LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

Skriptum
zur Vorlesung
Algorithmische Bioinformatik I&II

gehalten im Sommersemester 2019

und im Wintersemester 2019/20

am Lehrstuhl für Bioinformatik

Volker Heun

5. Februar 2020

Version 7.58

Vorwort

Dieses Skript entsteht parallel zu den Vorlesungen *Algorithmische Bioinformatik I und II*, die im Sommersemester 19 sowie im Wintersemester 19/20 an der Ludwig-Maximilians-Universität München für Studenten der Bioinformatik und Informatik im Rahmen des von der Ludwig-Maximilians-Universität und der Technischen Universität München gemeinsam veranstalteten Studiengangs Bioinformatik gehalten werden. Dieses Skript basiert dabei in Teilen auf einem früheren Skript zu den Vorlesungen *Algorithmische Bioinformatik I und II*, die im Wintersemester 01/02 und im Sommersemester 02, im Sommersemester 05 und im Wintersemester 05/06, im Sommersemester 08 und im Wintersemester 08/09, im Sommersemester 10 und im Wintersemester 10/11, im Sommersemester 12 und im Wintersemester 12/13, im Sommersemester 14 und im Wintersemester 14/15 sowie im Sommersemester 16 und im Wintersemester 16/17 für Studenten der Bioinformatik und Informatik sowie anderer Fachrichtungen im Rahmen des von der Ludwig-Maximilians-Universität und der Technischen Universität gemeinsam veranstalteten Studiengangs Bioinformatik gehalten wurde.

Teile, die im Sommersemester 19 und Wintersemester 19/20 nicht Teil der Vorlesung waren, sind mit einem Stern (*), Teile, die nur skizziert wurden, sind mit einem (+) markiert.

An dieser Stelle möchte ich mich für die Mithilfe bei der Erstellung des Skript aus dem Jahre 2002 bei folgenden Personen bedanken: Hamed Behrouzi, Michael Engelhardt, Jens Ernst, Peter Lücke, Moritz Maaß, Ingo Rohloff, Sabine Spreer, Hanjo Täubig. Weiterhin möchte ich für die Überarbeitung dieses Skripts insbesondere (in alphabetischer Reihenfolge) Benjamin Albrecht, Constantin Ammar, Florian Erhard, Johannes Fischer, Caroline Friedel, Simon W. Ginzinger, Markus Joppich, Maximilian Miller, Tobias Petri, Jens Quedenfeld, Bernhard Schauberger, Konrad Schreiber und Simone Wolf für ihre Unterstützung bei den Durchführungen der Veranstaltungen und für Hinweise zu Tippfehlern danken, die somit das aktuell vorliegende Skript erst möglich gemacht haben. Für weitere Hinweise zu Tippfehlern und inhaltlichen Fehlern danke ich Peter Heinig, Anton Smirnov, Hanjo Täubig und Stefan Wentzig.

Falls sich dennoch weitere (Tipp)Fehler unserer Aufmerksamkeit entzogen haben sollten, so bin ich für jeden Hinweis darauf (an Volker.Heun@bio.ifi.lmu.de) dankbar.

München, im Sommer- und Wintersemester 2019/20 Volker Heun

Inhaltsverzeichnis

Molekularbiol. Grundlagen (*) 0

0.1 Mendelsche Genetik

Dieses Kapitel ist nicht Bestandteil der Vorlesung, sondern dient nur als kurze Einführung in die Molkularbiologie für Hörer ohne besonderen biologischen Hintergrund.

In diesem Einführungskapitel wollen wir uns mit den molekularbiologischen Details beschäftigen, die für die informatische und mathematische Modellierung im Folgenden hilfreich sind. Zu Beginn stellen wir noch einmal kurz die Anfänge der systematischen Genetik, die Mendelsche Genetik, dar.

0.1.1 Mendelsche Experimente

Eine der ersten systematischen Arbeiten zur Vererbungslehre wurde im 19. Jahrhundert von Gregor Mendel geleistet. Unter anderem untersuchte Mendel die Vererbung einer Eigenschaft von Erbsen, nämlich ob die Erbsen eine glatte oder runzlige Oberfläche besitzen. Wie bei allen Pflanzen besitzt dabei jedes Individuum zwei Eltern (im Gegensatz beispielsweise zu Einzellern, die sich durch Zellteilung fortpflanzen).

Bei einer Untersuchung wurden in der so genannten *Elterngeneration* oder *Parentalgeneration* Erbsen mit *glatter* und Erbsen mit *runzliger* Oberfläche gekreuzt. Somit hatte in der nachfolgenden Generation, der so genannten *ersten Tochtergeneration* oder *ersten Filialgeneration* jede Erbse je ein Elternteil mit glatter und je ein Elternteil mit runzliger Oberfläche.

Überraschenderweise gab es bei den Nachkommen der Erbsen in der ersten Tochtergeneration nur noch glatte Erbsen. Man hätte wohl vermutet, dass sowohl glatte als auch runzlige Erbsen vorkommen oder aber leicht runzlige bzw. unterschiedlich runzlige Erbsen auftauchen würden.

Noch überraschender waren die Ergebnisse bei der nachfolgenden Tochtergeneration, der so genannten *zweiten Tochtergeneration* oder *zweiten Filialgeneration*, bei der nun beide Elternteile aus der ersten Tochtergeneration stammten. Hier kamen sowohl glatte als auch wieder runzlige Erbsen zum Vorschein. Interessanterweise waren jedoch die glatten Erbsen im Übergewicht, und zwar im Verhältnis 3 zu 1. Die Frage, die Mendel damals untersuchte, war, wie sich dieses Phänomen erklären lassen konnte.

0.1.2 Modellbildung

Als Modell schlug Gregor Mendel vor, dass die Erbse für ein bestimmtes Merkmal oder eine bestimmte Ausprägung von beiden Elternteilen je eine Erbinformation erhielt. Im Folgenden wollen wir eine kleinste Erbinformation als *Gen* bezeichnen. Zur Formalisierung bezeichen wir das Gen, das die glatte Oberfläche hervorruft mit G und dasjenige für die runzlige Oberfläche mit g. Da nun nach unserem Modell jede Erbse von beiden Elternteilen ein Gen erhält, muss jedes Gen für ein Merkmal doppelt vorliegen. Zwei Erbinformationen, also Gene, die für dieselbe Ausprägung verantwortlich sind, werden als *Allel* bezeichnet. Wir nehmen also an, dass unsere glatten Erbsen in der Elterngeneration die Allele GG und die runzligen die Allele gg enthalten.

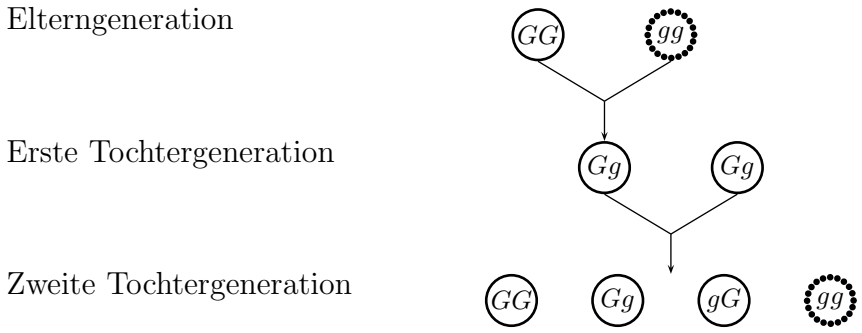


Abbildung 1: Skizze: Mendelsche Vererbung

Welche Erbinformation besitzt nun die erste Tochtergeneration? Sie erhält jeweils ein G und ein g von ihren Eltern und trägt als Erbinformation bezüglich der Oberfläche ein Gg. Was soll nun Gg eigentlich sein? Wir wissen nur, dass GG glatt und gg runzlig bedeutet. Ein Organismus, der bezüglich einer Ausprägung, dieselbe Erbinformation trägt, wird als *reinerbig* oder *homozygot* bezeichnet.

Wir haben nun mit Gg eine *mischerbige* oder *heterozygote* Erbinformation vorliegen. Wie oben bereits angedeutet, könnte die Ausprägung nun gemischt vorliegen, also ein „wenig runzlig“, oder aber einer der beiden Allelen zufällig die Ausprägung bestimmen.

Werden die Merkmale in Mischformen vererbt, wie in „ein wenig runzlig“, dann sagt man, dass das Merkmal *intermediär* vererbt wird. Mitglieder der ersten Tochtergeneration tragen dann also eine Mischung von beidem. Beispielsweise können die Nachfahren von Blumen mit roten bzw. weißen Blüten rosa-farbene Blüten besitzen oder aber auch weiße Blüten mit roten Tupfen etc.

Dies ist aber hier, wie die Experimente von Gregor Mendel gezeigt haben, nicht der Fall: Alle Erbsen der ersten Tochtergeneration sind glatt. Das bedeutet, dass

beide Gene eines Allels gegeneinander konkurrieren und in Abhängigkeit der Gene sich immer eins der beiden als dominant behauptet und den Wettkampf gewinnt. In unserem Falle, setzt sich also das Gen für die glatte Oberfläche gegenüber dem Gen für die runzlige durch. Das Gen, das sich durchsetzt, wird als *dominant* bezeichnet, und dasjenige, das unterliegt, wird als *rezessiv* bezeichnet.

Da nun sowohl die Erbinformation GG als auch Gg für glatte Erbsen stehen, muss man zwischen den so genannten Phänotypen und den Genotypen unterscheiden. Als *Phänotyp* bezeichnet man die sichtbare Ausprägung, also z.B. glatt. Als *Genotyp* bezeichnet man die Zusammensetzung der Erbinformation, also z.B. GG oder Gg für glatte Erbsen. Insbesondere kann also der Genotyp unterschiedlich, aber der Phänotyp gleich sein, wie bei den glatten Erbsen in der Elterngeneration und in der ersten Tochtergeneration.

Wie kann man jetzt die Erscheinung in der zweiten Tochtergeneration erklären? Betrachten wir nun die Eltern, also die Erbsen der ersten Tochtergeneration, die als Genotyp Gg tragen. Nimmt man nun an, das jedes Elternteil eines seiner Gene eines Allels zufällig (mit gleich hoher Wahrscheinlichkeit) an seine Kinder weitergibt, dann gibt es für die Erbsen der zweiten Tochtergeneration $2 \cdot 2 = 4$ Möglichkeiten, wie sich diese Gene dieses Alles vererben können (siehe Abbildung 2).

	G	g
G	GG	Gg
g	gG	gg

Abbildung 2: Skizze: Vererbung des Genotyps von zwei mischerbigen Eltern

Also sind drei der vier Kombinationen, die im Genotyp möglich sind (GG, gG sowie Gg), im Phänotyp gleich, nämlich glatt. Nur eine der Kombinationen im Genotyp liefert im Phänotyp eine runzlige Erbse. Dies bestätigt in eindrucksvoller Weise das experimentell ermittelte Ergebnis, dass etwa dreimal so viele glatte wie runzlige Erbsen zu beobachten sind.

An dieser Stelle müssen wir noch anmerken, dass diese Versuche nur möglich sind, wenn man in der Elterngeneration wirklich reinerbige Erbsen zur Verfügung hat und keine mischerbigen. Auf den ersten Blick ist dies nicht einfach, da man ja nur den Phänotyp und nicht den Genotyp einfach ermitteln kann. Durch vielfache Züchtung kann man jedoch die Elternteile identifizieren, die reinerbig sind (nämlich, die runzligen sowie die glatten, deren Kinder und Enkelkinder nicht runzlig sind).

0.1.3 Mendelsche Gesetze

Fassen wir hier noch einmal kurz die drei so genannten Mendelschen Gesetze zusammen, auch wenn wir hier nicht alle bis ins Detail erläutert haben:

1) Uniformitätsregel: Werden zwei reinerbige Individuen einer Art gekreuzt, die sich in einem einzigen Merkmal unterscheiden, so sind alle Individuen der ersten Tochtergeneration gleich.

2) Spaltungsregel: Werden zwei Mischlinge der ersten Tochtergeneration miteinander gekreuzt, so spalten sich die Merkmale in der zweiten Tochtergeneration im Verhältnis 1 zu 3 bei dominant-rezessiven Genen und im Verhältnis 1 zu 2 zu 1 bei intermediären Genen auf.

3) Unabhängigkeitsregel Werden zwei mischerbige Individuen, deren Elterngeneration sich in zwei Merkmalen voneinander unterschieden hat, miteinander gekreuzt, so vererben sich die einzelnen Erbanlagen unabhängig voneinander.

Die Unabhängigkeitsregel gilt in der Regel nur, wenn die Gene auf verschiedenen Chromosomen sitzen bzw. innerhalb eines Chromosoms so weit voneinander entfernt sind, dass eine so genannte *Crossing-Over-Mutation* hinreichend wahrscheinlich ist.

0.1.4 Wo und wie sind die Erbinformationen gespeichert?

Damit haben wir die Grundlagen der Genetik ein wenig kennen gelernt. Es stellt sich jetzt natürlich noch die Frage, wo und wie die Gene gespeichert werden. Dies werden wir in den folgenden Abschnitten erläutern.

Zum Abschluss noch ein paar Notationen. Wie bereits erwähnt, bezeichnen wir ein *Gen* als den Träger einer kleinsten Erbinformation. Alle Gene eines Organismus zusammen bilden das *Genom*. Wie bereits aus der Schule bekannt sein dürfte, ist das Genom auf dem oder den *Chromosom(en)* gespeichert (je nach Spezies).

0.2 Chemische Grundlagen

Bevor wir im Folgenden auf die molekularbiologischen Grundlagen näher eingehen, wiederholen wir noch ein paar elementare Begriffe und Eigenschaften aus der Chemie bzw. speziell aus der organischen und der Biochemie. Die in der Biochemie wichtigsten auftretenden Atome sind Kohlenstoff (C), Sauerstoff (O), Wasserstoff (H), Stickstoff (N), Schwefel (S), Kalzium (Ca), Eisen (Fe), Magnesium (Mg), Kalium (K)

und Phosphor (P). Diese Stoffe lassen sich beispielsweise mit folgendem Merkspruch behalten: **COHNS CaFe M***it* **g***roßem* **K***uchen-***P***aket*. Zunächst einmal wiederholen wir kurz die wichtigsten Grundlagen der chemischen Bindungen.

0.2.1 Kovalente Bindungen

Die in der Biochemie wichtigste Bindungsart ist die *kovalente Bindung*. Hierbei steuern zwei Atome je ein Elektron bei, die dann die beiden Atome mittels einer gemeinsamen Bindungswolke zusammenhalten. Im Folgenden wollen wir den Raum, für den die Aufenthaltswahrscheinlichkeit eines Elektrons bzw. eines Elektronenpaares (nach dem Pauli-Prinzip dann mit verschiedenem Spin) am größten ist, als *Orbital* bezeichnen.

Hierbei sind die Kohlenstoffatome von besonderer Bedeutung, die die organische und Biochemie begründen. Die wichtigste Eigenschaft der Kohlenstoffatome ist, dass sie sowohl Einfach-, als auch Doppel- und Dreifachbindungen untereinander ausbilden können. Das Kohlenstoffatom hat in der äußersten Schale 4 Elektronen. Davon befinden sich im Grundzustand zwei in einem so genannten *s-Orbital* und zwei jeweils in einem so genannten *p-Orbital*.

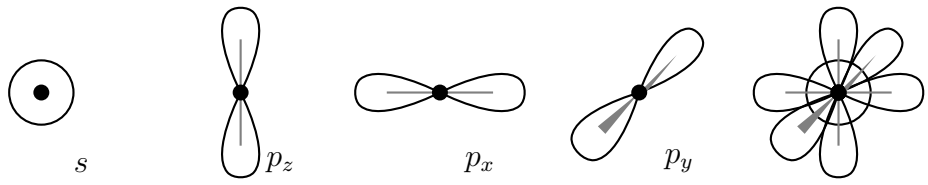


Abbildung 3: Skizze: Räumliche Ausdehnung der Orbitale

Das s-Orbital ist dabei kugelförmig, während die drei verschiedenen p-Orbitale jeweils eine *Doppelhantel* ausbilden, die paarweise orthogonal zueinander sind. In Abbildung 3 ist die räumliche Ausdehnung des s- und der drei p-Orbitale schematisch dargestellt, von denen jedes bis zu zwei Elektronen aufnehmen kann. Ganz rechts sind alle Orbitale gleichzeitig zu sehen, die in der Regel für uns interessant sein werden.

In Einfachbindungen befinden sich beim Kohlenstoffatom die einzelnen Elektronen in so genannten *sp^3-hybridisierten Orbitalen*, die auch als *q-Orbitale* bezeichnet werden. Hierbei bilden sich aus den 3 Hanteln und der Kugel vier energetisch äquivalente keulenartige Orbitale. Dies ist in der Abbildung 4 links dargestellt. Die Endpunkte der vier Keulen bilden dabei ein Tetraeder aus. Bei einer Einfachbindung überlappen sich zwei der Keulen, wie in Abbildung 4 rechts dargestellt. Die in der Einfachbindung überlappenden q-Orbitale bilden dann ein so genanntes *σ-Orbital*.

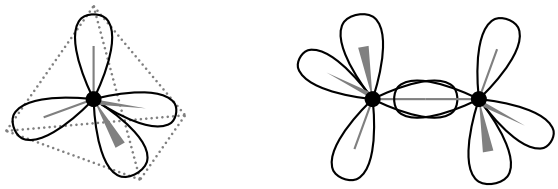


Abbildung 4: Skizze: sp^3 hybridisierte Orbitale sowie eine Einfachbindung

In Doppelbindungen sind nur zwei p-Orbitale und ein s-Orbital zu drei Keulen hybridisiert, so genannte *sp^2-Orbitale*. Ein p-Orbital bleibt dabei bestehen. Dies ist in Abbildung 5 links illustriert. Die in Doppelbindungen überlappenden p-Orbitale werden dann auch als *π-Orbital* bezeichnet. Bei einer Doppelbindung überlappen sich zusätzlich zu den zwei keulenförmigen hybridisierten q-Orbitalen, die die σ-Bindung bilden, auch noch die beiden Doppelhanteln der p-Orbitale, die dann das π-Orbital bilden. Dies ist schematisch in der Abbildung 5 rechts dargestellt (die Abbildungen sind nicht maßstabsgetreu).

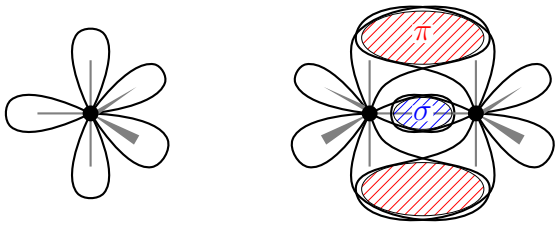


Abbildung 5: Skizze: sp^2 hybridisierte Orbitale und eine Doppelbindung

Die Bindung der Doppelbindung, die durch Überlappung von q-Orbitalen entsteht, wird auch *σ-Bindung* genannt, die Bindung der Doppelbindung, die durch Überlappung von p-Orbitalen entsteht, wird als *π-Bindung* bezeichnet. Ähnlich verhält es sich bei Dreifachbindungen, wo zwei p-Orbitale verbleiben und ein s- und nur ein p-Orbital zu einem *sp-Orbital* hybridisieren. Die konkrete Art der Hybridisierung der Orbitale der Kohlenstoffatome eines bestimmten Moleküls ist deshalb so wichtig, weil dadurch die dreidimensionale Struktur des Moleküls festgelegt wird. Die vier sp^3-Orbitale zeigen in die Ecken eines Tetraeders, die drei sp^2-Orbitale liegen in einer Ebene, auf der das verbleibende p-Orbital senkrecht steht, die zwei sp-Orbitale schließen einen Winkel von 180° ein und sind damit gerade gestreckt, auf ihnen stehen die verbleibenden beiden p-Orbitale senkrecht.

Bei zwei benachbarten Doppelbindungen (wie im Butadien, $H_2C{=}CH{-}HC{=}CH_2$) verbinden sich in der Regel die beiden benachbarten Orbitale, die die jeweilige π-Bindung zur Doppelbindung machen, um dann quasi eine π-Wolke über alle vier Kohlenstoffatome auszubilden, da dies energetisch günstiger ist. Daher spricht man

bei den Elektronen in dieser verschmolzenen Wolke auch von *delokalisierten π-Elektronen*.

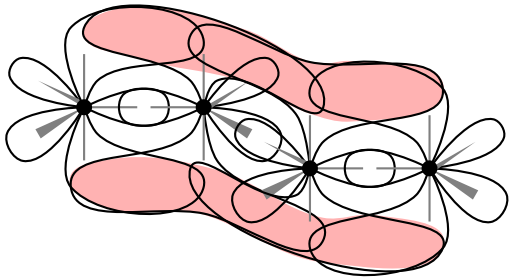


Abbildung 6: Skizze: Delokalisierte π-Bindung im Butadien

Ein Beispiel hierfür ist das *Benzol*-Molekül (C_6H_6). Aus energetischen Gründen bilden sich in dem Ring aus sechs Kohlenstoffatomen nicht drei einzelne alternierende Doppelbindungen aus, sondern eine große Wolke aus sechs delokalisierten π-Elektronen, die die starke Bindung des Benzolrings begründen.

Wie wir später noch sehen werden, kann sich eine Wolke aus delokalisierten π-Elektronen auch aus den π-Elektronen einer C=C Doppelbindung und dem nichtbindenden Orbital eines Sauerstoff- oder Stickstoffatoms bilden. Stickstoff bzw. Sauerstoff besitzen in der äußersten Schale mehr als vier Elektronen und daher kann sich ein *p*-Orbital mit zwei Elektronen ausbilden. Dieses hat dann bezüglich der Delokalisation von π-Elektronen ähnliche Eigenschaften wie eine π-Bindung.

Die Energie einer kovalenten Bindung variiert zwischen 200kJ/mol und 450kJ/mol (Kilojoule pro Mol), wobei Kohlenstoffatome untereinader relativ starke Bindungen besitzen (etwa 400kJ/mol). Die Angabe dieser absoluten Werte ist für uns eigentlich nicht von Interesse. Wir geben sie hier nur an, um die Stärken der verschiedenen Bindungsarten im Folgenden vergleichen zu können.

0.2.2 Ionische Bindungen

Bei *ionischen Bindungen* gibt ein Atom, das so genannte *Donatoratom*, ein Elektron an ein anderes Atom, das so genannte *Akzeptoratom*, ab. Damit sind die Donatoratome positiv und die Akzeptoratome negativ geladen. Durch die elektrostatische Anziehungskraft (und die Abstoßung gleichnamiger Ladung) bildet sich in der Regel ein Kristallgitter aus, das dann abwechselnd aus positiv und negativ geladenen Atomen besteht.

Ein bekanntes Beispiel hierfür ist Kochsalz, d.h. Natriumchlorid (NaCl). Dabei geben die Natriumatome jeweils das äußerste Elektron ab, das dann von den Chloratomen

aufgenommen wird. Dadurch sind die Natriumatome positiv und die Chloratome negativ geladen, die sich dann innerhalb eines Kristallgitters anziehen.

Hier wollen wir noch deutlich den Unterschied herausstellen, ob wir diese Bindungen in wässriger Lösung oder ohne Lösungsmittel betrachten. Ohne Wasser als Lösungsmittel sind ionische Bindungen sehr stark. In wässriger Lösung sind sie jedoch sehr schwach, sie werden etwa um den Faktor 80 schwächer. Beispielsweise löst sich das doch recht starke Kristallgitter des Kochsalzes im Wasser nahezu auf. In wässriger Lösung beträgt die Energie einer ionischen Bindung etwa 20 kJ/mol.

0.2.3 Wasserstoffbrücken

Eine andere für uns sehr wichtige Anziehungskraft, die keine Bindung im eigentlichen chemischen Sinne ist, sind die *Wasserstoffbrücken*. Diese Anziehungskräfte werden im Wesentlichen durch die unterschiedlichen Elektronegativitäten der einzelnen Atome bedingt.

Die Elektronegativität ist ein Maß dafür, wie stark die Elektronen in der äußersten Schale angezogen werden. Im Periodensystem der Elemente wächst der Elektronegativitätswert innerhalb einer Periode von links nach rechts, weil dabei mit der Anzahl der Protonen auch die Kernladung und damit auch die Anziehungskraft auf jedes einzelne Elektron ansteigt. Innerhalb einer Hauptgruppe sinkt die Elektronegativität mit zunehmender Ordnungszahl, weil die Außenelektronen sich auf immer höheren Energieniveaus befinden und der entgegengesetzt geladene Kern durch die darunterliegenden Elektronen abgeschirmt wird. Deshalb ist z.B. Fluor das Element mit dem größten Elektronegativitätswert.

Eine Liste der für uns wichtigsten Elektronegativitäten in für uns willkürlichen Einheiten ist in Abbildung 7 angegeben. Hier bedeutet ein größerer Wert eine größere Affinität zu Elektronen.

Atom	C	O	H	N	S	P
EN	2,55	3,44	2,20	3,04	2,58	2,19

Abbildung 7: Tabelle: Elektronegativitäten nach Pauling

Bei einer kovalenten Bindung sind die Elektronenwolken in Richtung des Atoms mit der stärkeren Elektronegativität hin verschoben. Dadurch bekommt dieses Atom eine teilweise negative Ladung, während das andere teilweise positiv geladen ist. Ähnlich wie bei der ionischen Bindung, wenn auch bei weitem nicht so stark, wirkt diese Polarisierung der Atome anziehend.

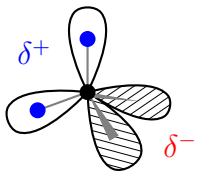


Abbildung 8: Skizze: Polarität bei einem Wassermolekül

Insbesondere Wasser ist für die Ausbildung von zahlreichen Wasserstoffbrücken bekannt. In Abbildung 8 ist ein Wassermolekül schematisch dargestellt.

Wie beim Kohlenstoffatom sind die drei p-Orbitale und das s Orbital zu vier q-Orbitalen hybridisiert. Da das Sauerstoffatom in der äußersten Schale sechs anstatt vier Elektronen besitzt, sind bereits zwei der q-Orbitale des Sauerstoffatoms mit je zwei Elektronen besetzt und können daher keine kovalente Bindung eingehen. Man bezeichnet diese Orbitale daher auch als *nichtbindend*.

Die beiden anderen werden im Wasser gemäß der Formel H_2O mit jeweils einem Wasserstoffatom protoniert. Da nun die beiden nichtbindenden Orbitale (zumindest aus dieser Richtung auf das Sauerstoffatom) negativ geladen sind und die beiden protonierten bindenden Orbitale positiv geladen sind, wirkt das Wasser als Dipol und die Wassermoleküle hängen sich wie viele kleine Stabmagneten aneinander. Einziger Unterschied ist hier dass die Teilladungen in den Ecken eine Tetraeders sitzen, so dass sich die Wassermoleküle ähnlich wie die Kohlenstoffatome im Kristallgitter des Diamanten anordnen.

Im gefrorenen Zustand ist das Kristallgitter von Wasser (also Eis) nahezu ein Diamantengitter, während im flüssigen Zustand die Wasserstoffbrücken häufig aufbrechen und sich wieder neu bilden. Daher ist es zum einen flüssig, und zum anderen kann es im flüssigen Zustand dichter gepackt werden als im gefrorenen Zustand. Erst dadurch nimmt Wasser den flüssigen Zustand bei Zimmertemperatur an, während sowohl Wasserstoff wie auch Sauerstoff einen sehr niedrigen Siedepunkt besitzen. Eine Wasserstoffbrückenbindung ist mit ca. 21 kJ/mol deutlich schwächer als eine kovalente oder eine ionische Bindung.

0.2.4 Van der Waals-Kräfte

Die *Van der Waals-Anziehung* bzw. *Van der Waals-Kräfte* treten insbesondere in großen bzw. langen Molekülen, wie Kettenkohlenwasserstoffen auf. Da der Ort der Elektronen ja nicht festgelegt ist (Heisenbergsche Unschärferelation), können sich durch die Verlagerung der Elektronen kleine Dipolmomente in den einzelnen Bindungen ergeben. Diese beeinflussen sich gegenseitig und durch positive Rückkopplungen

können diese sich verstärken. Somit können sich lange Moleküle fester aneinander legen als kürzere. Dies ist mit ein Grund dafür, dass die homologe Reihe der Alkane (C_nH_{2n+2}) mit wachsender Kohlenstoffanzahl bei Zimmertemperatur ihren Aggregatzustand von gasförmig über flüssig bis zu fest ändert. Die Energie der Van der Waals-Kraft liegt bei etwa 4kJ/mol.

0.2.5 Hydrophobe Kräfte

Nichtpolare Moleküle, wie Fette, können mit Wasser keine Wasserstoffbrücken ausbilden. Dies ist der Grund, warum nichtpolare Stoffe in Wasser unlöslich sind. Solche Stoffe werden auch als *hydrophob* bezeichnet, während polare Stoffe auch als *hydrophil* bezeichnet werden. Aufgrund der Ausbildung von zahlreichen Wasserstoffbrücken innerhalb des Wassers (auch mit hydrophilen Stoffen) tendieren hydrophobe Stoffe dazu, sich möglichst eng zusammenzulagern, um eine möglichst kleine Oberfläche (gleich Trennfläche zum Wasser) auszubilden. Diese Tendenz des Zusammenlagerns hydrophober Stoffe in wässriger Lösung wird als *hydrophobe Kraft* bezeichnet.

0.2.6 Funktionelle Gruppen

Wie schon zu Beginn bemerkt spielt in der organischen und Biochemie das Kohlenstoffatom die zentrale Rolle. Dies liegt insbesondere daran, dass es sich mit sich selbst verbinden kann und sich so eine schier unendliche Menge an verschiedenen Molekülen konstruieren lässt. Dabei sind jedoch auch andere Atome beteiligt, sonst erhalten wir bekanntlich Graphit oder den Diamanten.

Chem. Rest	Gruppe	gewöhnlicher Name
$-CH_3$	Methyl	
–OH	Hydroxyl	Alkohol
$-NH_2$	Amino	Amine
–NH–	Imino	
–CHO	Carbonyl	Aldehyde
–CO–	Carbonyl	Ketone
–COO–	Ester	Ester
–COOH	Carboxyl	organische Säure
–CN	Cyanid	Nitrile
–SH	Sulfhydril	Thiole

Abbildung 9: Tabelle: Einige funktionelle (organische) Gruppen

Um diese anderen vorkommenden Atome bezüglich ihrer dem Molekül verleihenden Eigenschaften ein wenig besser einordnen zu können, beschreiben wir die am meisten vorkommenden *funktionellen Gruppen*. Die häufigsten in der Biochemie auftretenden einfachen funktionellen Gruppen sind in der Tabelle 9 zusammengefasst.

0.2.7 Stereochemie und Enantiomerie

In diesem Abschnitt wollen wir einen kurzen Einblick in die *Stereochemie*, speziell in die *Enantiomerie* geben. Die Stereochemie beschäftigt sich mit der räumlichen Anordnung der Atome in einem Molekül. Beispielsweise ist entlang einer Einfachbindung die Rotation frei möglich. Bei Doppelbindungen ist diese aufgrund der π-Bindung eingeschränkt und es kann zwei mögliche räumliche Anordnungen desselben Moleküls geben. In Abbildung 10 sind zwei Formen für Äthendiol angegeben. Befinden sich beide (der bedeutendsten) funktionellen Gruppen auf derselben Seite der Doppelbindung, so spricht man vom *cis-Isomer* andernfalls von *trans-Isomer*. Bei der cis-trans-Isomerie kann durch Energiezufuhr die Doppelbindung kurzzeitig

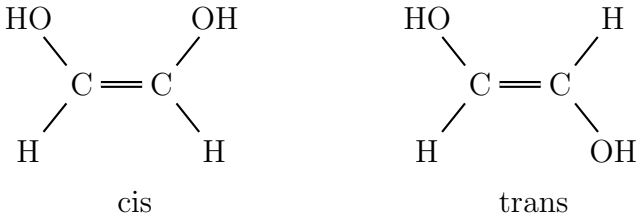


Abbildung 10: Skizze: Cis-Trans-Isomerie bei Äthendiol

geöffnet werden und um 180° gedreht werden, so dass die beiden Isomere ineinander überführt werden können.

Es hat sich herausgestellt, dass scheinbar identische Stoffe (aufgrund der Summen- und Strukturformel) sich unter bestimmten Bedingungen unterschiedlich verhalten können. Dies sind also solche Isomere, die sich nicht ineinander überführen lassen. Betrachten wir dazu in Abbildung 11 ein Kohlenstoffatom (schwarz dargestellt) und vier unterschiedliche funktionelle Gruppen (farbig dargestellt), die jeweils

Abbildung 11: Skizze: Asymmetrisches Kohlenstoffatom

mittels einer Einfachbindung an das Kohlenstoffatom gebunden sind. Das betrachtete Kohlenstoffatom wird hierbei oft als *zentrales Kohlenstoffatom* bezeichnet. Auf den ersten Blick sehen die beiden Moleküle in Abbildung 11 gleich aus. Versucht man jedoch, die beiden Moleküle durch Drehungen im dreidimensionalen Raum zur Deckung zu bringen, so wird man feststellen, dass dies gar nicht geht. Die beiden Moleküle sind nämlich Spiegelbilder voneinander.

Daher werden Moleküle als *chiral* (deutsch Händigkeit) bezeichnet, wenn ein Molekül mit seinem Spiegelbild nicht durch Drehung im dreidimensionalen Raum zur Deckung gebracht werden kann. Die beiden möglichen, zueinander spiegelbildlichen Formen nennen wir *Enantiomere*. Beide Formen werden auch als *enantiomorph* zueinander bezeichnet. Für Kohlenstoffatome, die mit vier *unterschiedlichen* Resten verbunden sind, gilt dies immer. Aus diesem Grund nennt man ein solches Kohlenstoffatom auch ein *asymmetrisches Kohlenstoffatom*.

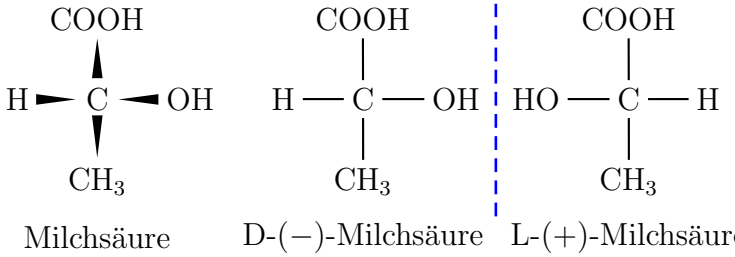


Abbildung 12: Skizze: Milchsäure

Ein einfaches (und bekanntes) Beispiel hierfür ist die Milchsäure. Hierbei sind die funktionellen Gruppen, die an einem zentralen Kohlenstoffatom sitzen, ein Wasserstoffatom, eine Hydroxyl-, eine Methyl und eine Carboxylgruppe. In Abbildung 12 sind rechts die beiden Formeln der beiden spiegelbildlichen Formen dargestellt. In einer zweidimensionalen Abbildung muss man jedoch eine Konvention einführen, wie man die Projektion vornimmt. Die längste Kohlenstoffkette wird dabei von oben nach unten dargestellt. Hier also das zentrale Kohlenstoffatom und das Kohlenstoffatom der Methylgruppe. Dabei wird oben von der charakteristischsten funktionellen Gruppe, hier die Carboxylgruppe, bestimmt. Dabei ist zu verstehen, dass die vertikale Kette hinter dem zentralen Kohlenstoffatom unter der Papierebene verschwindet, während die beiden restlichen Seitenketten aus der Papiereben dem Leser entgegen kommen (dies wird als *Fischer-Projektion* bezeichnet). Dies ist in der Abbildung 12 ganz links noch einmal illustriert.

Da nun im mittleren Teil der Abbildung 12 die bedeutendere funktionelle Gruppe, also die Hydroxylgruppe gegenüber dem Wasserstoffatom, rechts sitzt, wird dieses Enantiomer mit D-Milchsäure (latein. dexter, rechts) bezeichnet. Rechts handelt es sich dann um die L-Milchsäure (latein. laevis, links).

Diese Bezeichnungsweise hat nichts mit den aus der Werbung bekannten links- bzw. rechtsdrehenden Milchsäuren zu tun. Die Namensgebung kommt von der Tatsache, dass eine Lösung von Milchsäure, die nur eines der beiden Enantiomere enthält, polarisiertes Licht dreht. Dies gilt übrigens auch für die meisten Moleküle, die verschiedene Enantiomere besitzen. Je nachdem, ob es polarisiertes Licht nach rechts oder links verdreht, wird es als *rechtsdrehend* oder *linksdrehend* bezeichnet (und im Namen durch (+) bzw. (−) ausgedrückt).

Bei der Milchsäure stimmen zufällig die D- bzw. L-Form mit der rechts- bzw. linksdrehenden Eigenschaft überein. Bei Aminosäuren, die wir noch kennen lernen werden, drehen einige L-Form nach rechts! Hier haben wir einen echten Unterschied gefunden, mit dem sich Enantiomere auf makroskopischer Ebene unterscheiden lassen.

In der Chemie wurde die *DL-Nomenklatur* mittlerweile zugunsten der so genannten *RS-Nomenklatur* aufgegeben. Da jedoch bei Zuckern und Aminosäuren oft noch die DL-Nomenklatur verwendet wird, wurde diese hier angegeben. Für Details bei der DL - sowie der RS-Nomenklatur verweisen wir auf die einschlägige Literatur.

0.2.8 Tautomerien

Tautomerien sind intramolekulare Umordnungen von Atomen. Dabei werden chemisch zwei verschiedene Moleküle ineinander überführt. Wir wollen dies hier nur exemplarisch am Beispiel der *Keto-Enol-Tautomerie* erklären. Wir betrachten dazu die folgende Abbildung 13

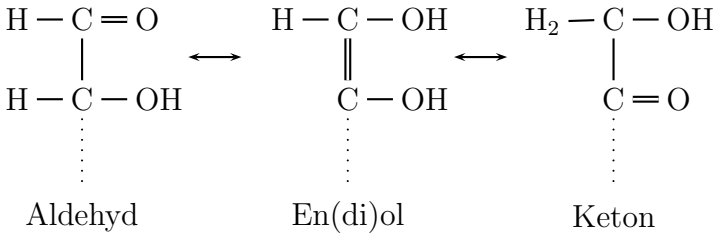


Abbildung 13: Skizze: Keto-Enol-Tautomerie

Im Aldehyd sind aufgrund der Doppelbindung in der Carbonylgruppe und der daraus resultierenden starken Elektronegativität die Elektronen zum Sauerstoffatom der Carbonylgruppe verschoben. Dies führt induktiv zu einer Verlagerung der Elektronen in der C-C Bindung zum Kohlenstoffatom der Carbonylgruppe. Auf der anderen Seite sind die anderen Elektronen im Bindungsorbital zur Hydroxylgruppe aufgrund

derer starken Elektronegativität zum Sauerstoffatom hin verschoben. Dadurch lässt sich das Wasserstoffatom am zweiten Kohlenstoffatom sehr leicht trennen und kann eines der nichtbindenden Orbitale des Sauerstoffatoms der benachbarten Carbonylgruppe protonieren. Diese wandelt sich somit zu einer Hydroxylgruppe und es entsteht zwischen den beiden Kohlenstoffatomen eine Doppelbindung.

Man beachte hierbei, dass die bindenden Orbitale der π-Bindung und die nichtbindenden Orbitale der angrenzenden Sauerstoffatome sich jetzt ebenfalls überlappen, um für die darin enthaltenen Elektronen ein größeres Orbital bereitzustellen. Durch eine Delokalisierung dieser Elektronen kann sich zwischen dem zweiten Kohlenstoffatom und dem Sauerstoffatom der Hydroxylgruppe eine Carbonylgruppe ausbilden. Das frei werdende Wasserstoffatom wird dann unter Aufbruch der Doppelbindung am ersten Kohlenstoffatom angelagert.

Aus ähnlichen Gründen kann sich diese intramolekulare Umlagerung auch auf dem Rückweg abspielen, so dass sich hier ein Gleichgewicht einstellt. Das genaue Gleichgewicht kann nicht pauschal angegeben werden, da es natürlich auch von den speziellen Randbedingungen abhängt. Wie schon erwähnt gibt es auch andere Tautomerien, d.h. intramolekulare Umlagerungen bei anderen Stoffklassen.

0.3 DNS und RNS

In diesem Abschnitt wollen wir uns um die chemische Struktur der *Desoxyribonukleinsäure* oder kurz *DNS* bzw. *Ribonukleinsäure* oder kurz *RNS* (engl. *deoxyribonucleic acid*, *DNA* bzw. *ribonucleic acid*, *RNA*) kümmern. In der DNS wird die eigentliche Erbinformation gespeichert und diese wird durch die RNS zur Verarbeitung weitergegeben. Wie diese Speicherung und Weitergabe im Einzelnen geschieht, werden wir später noch genauer sehen.

0.3.1 Zucker

Ein wesentlicher Bestandteil der DNS sind Zucker. Chemisch gesehen sind Zucker Moleküle mit der Summenformel $C_nH_{2n}O_n$ (weshalb sie oft auch als *Kohlenhydrate* bezeichnet werden). In Abbildung 14 sind die für uns wichtigsten Zucker in der Strukturformel dargestellt. Für uns sind insbesondere Zucker mit 5 oder 6 Kohlenstoffatomen von Interesse. Zucker mit 5 bzw. 6 Kohlenstoffatomen werden auch *Pentosen* bzw. *Hexosen* genannt.

Jeder Zucker enthält eine Carbonylgruppe, so dass Zucker entweder ein Aldehyd oder ein Keton darstellen. Daher werden Zucker entsprechend auch als *Aldose* oder

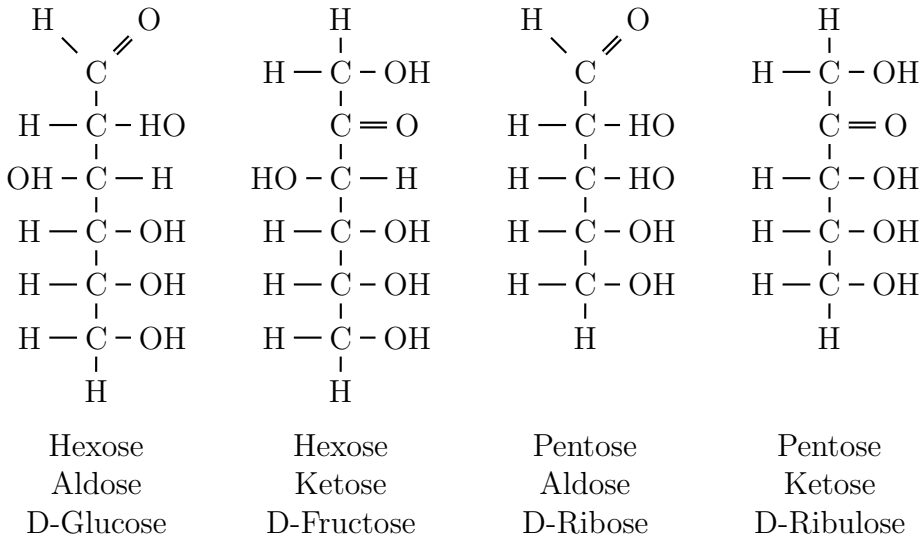


Abbildung 14: Skizze: Zucker (Hexosen und Pentosen sowie Aldosen und Ketosen)

Ketose bezeichnet. Diese Unterscheidung ist jedoch etwas willkürlich, da aufgrund der Keto-Enol-Tautomerie eine Aldose in eine Ketose überführt werden kann. In der Regel pendelt sich ein Gleichgewicht zwischen beiden Formen ein. An dieser Stelle wollen wir noch anmerken, dass es sich bei allen Kohlenstoffatomen (bis auf das erste und das letzte) um asymmetrische Kohlenstoffatome handelt. Somit bilden Zucker Enantiomere aus.

Warum haben jetzt eigentlich Glucose und Fructose unterschiedliche Namen, obwohl diese aufgrund der Keto-Enol-Tautomerie im Gleichgewicht miteinander stehen? In der Natur treten die Zucker kaum als Aldose oder Ketose auf. Die Aldehyd- bzw. Ketogruppe bildet mit einer der Hydroxylgruppen einen Ring aus. In der Regel sind diese 5er oder 6er Ringe. Als 5er Ringe werden diese Zucker als *Furanosen* (aufgrund ihrer Ähnlichkeit zu *Furan*) bezeichnet, als 6er Ringe als *Pyranosen* (aufgrund ihrer Ähnlichkeit zu *Pyran*).

Bei den Hexosen (die hauptsächlich in gewöhnlichen Zuckern und Stärke vorkommen) wird der Ringschluss über das erste und vierte bzw. fünfte Kohlenstoffatom gebildet. Bei Pentosen (mit denen wir uns im Folgenden näher beschäftigen wollen) über das erste und vierte Kohlenstoffatom. Dabei reagiert die Carbonylgruppe mit der entsprechenden Hydroxylgruppe zu einem so genannten *Halb-Acetal*, wie in der folgenden Abbildung 15 dargestellt. Aus der Carbonylgruppe entsteht dabei die so genannte glykosidische OH-Gruppe. Die Ausbildung zum Voll-Acetal geschieht über eine weitere Reaktion (Kondensation) dieser Hydroxylgruppe am zentralen Kohlenstoffatom der ehemaligen Carbonylgruppe.

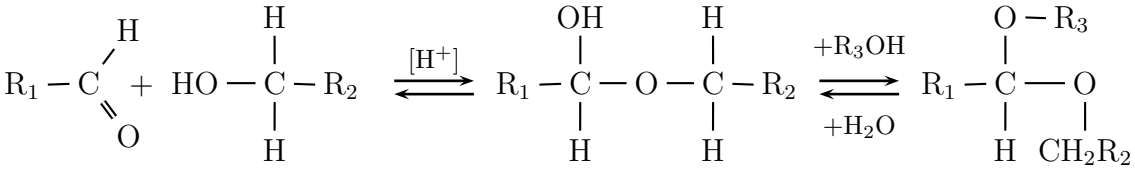


Abbildung 15: Skizze: Halb-Acetal- und Voll-Acetal-Bildung

In der Abbildung 16 sind zwei Furanosen, nämlich *Ribose* und *Desoxyribose* dargestellt. Der einzige Unterschied ist das quasi fehlende Sauerstoffatom am zweiten Kohlenstoffatom, dort ist eine Hydroxylgruppe durch ein Wasserstoffatom ersetzt. Daher stammt auch der Name *Desoxy*ribose. Wie man aus dem Namen schon vermuten kann tritt die Desoxyribose in der Desoxyribonukleinsäure (DNS) und die Ribose in der Ribonukleinsäure (RNS) auf. Die Kohlenstoffatome werden dabei zur Unterscheidung von 1 bis 5 durchnummeriert. Aus später verständlich werdenden Gründen, verwenden wir eine gestrichene Nummerierung 1' bis 5' (siehe auch Abbildung 16).

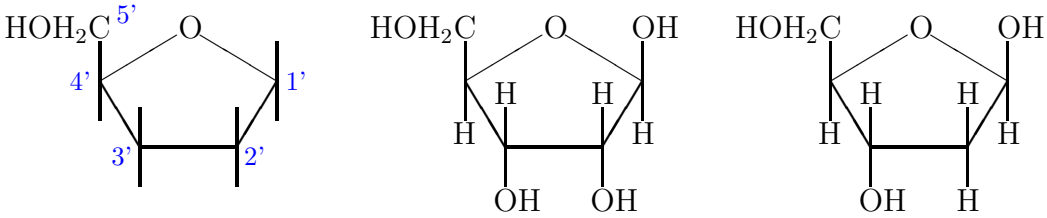


Abbildung 16: Skizze: Ribose und Desoxyribose als Furanosen

0.3.2 Basen

In diesem Abschnitt wollen wir einen weiteren wesentlichen Bestandteil der DNS bzw. RNS vorstellen, die so genannten *Basen*. Hiervon gibt es fünf verschiedene: Adenin, Guanin, Cytosin, Thymin und Uracil. Betrachten wir zuerst die von Purin abgeleiteten Basen. In Abbildung 17 ist links das Purin dargestellt und in der Mitte bzw. rechts *Adenin* bzw. *Guanin*. Die funktionellen Gruppen, die Adenin bzw. Guanin von Purin unterscheiden, sind rot dargestellt. Man beachte auch, dass sich durch die Carbonylgruppe im Guanin auch die Doppelbindungen in den aromatischen Ringen formal ändern. Da es sich hierbei jedoch um alternierende Doppelbindungen und nichtbindende Orbitale handelt, sind die Elektronen sowieso über die aromatischen Ringe delokalisiert.

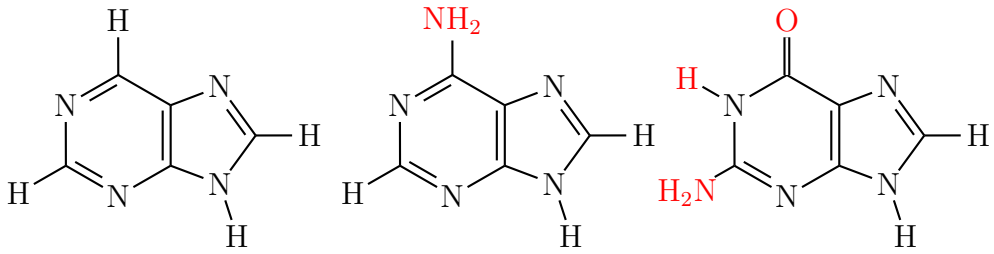


Abbildung 17: Skizze: Purine

Eine weitere Gruppe von Basen erhält man aus Pyrimidin, dessen Strukturformel in der Abbildung 18 links abgebildet ist. Hiervon werden *Cytosin*, *Thymin* und *Uracil* abgeleitet. Auch hier sind die funktionellen Gruppen, die den wesentlichen Unterschied zu Pyrimidin ausmachen, wieder rot bzw. orange dargestellt. Man beachte, dass sich Thymin und Uracil nur in der orange dargestellten Methylgruppe unterscheiden. Hier ist insbesondere zu beachten, dass Thymin nur in der DNS und Uracil nur in der RNS vorkommt. Auch hier beachte man, dass sich durch die Carbonyl-

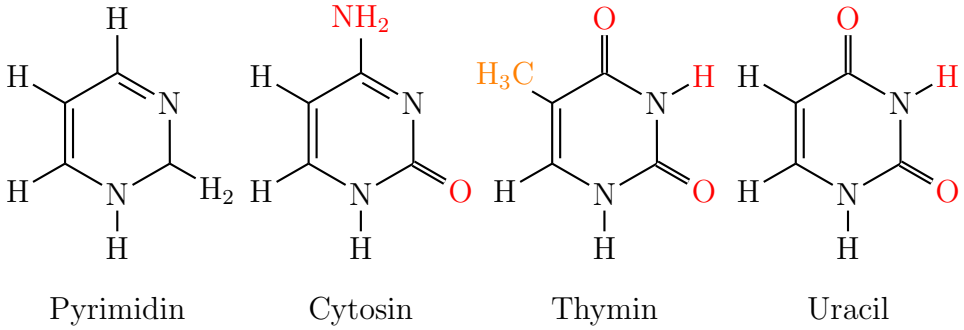


Abbildung 18: Skizze: Pyrimidine

gruppe im Thymin bzw. Uracil auch die Doppelbindungen in den aromatischen Ringen formal ändern. Jedoch bleiben auch hier die Elektronen über die aromatischen Ringe delokalisiert.

Ohne an dieser Stelle im Detail darauf einzugehen, merken wir noch an, dass Guanin über die Keto-Enol-Tautomerie mit einem ähnlichen Stoff in Wechselwirkung steht, ebenso Cytosin über eine so genannte Amino-Imino-Tautomerie. Wir kommen später noch einmal kurz darauf zurück.

Wie im Zucker werden auch die Atome in den aromatischen Ringen durchnummeriert. Da wir im Folgenden auf diese Nummerierung nie zurückgreifen werden, geben

wir sie an dieser Stelle auch nicht an. Um eine Verwechslung mit der Nummerierung in den Zuckern zu vermeiden, wurde die Nummerierung in den Zuckern gestrichen durchgeführt.

0.3.3 Polymerisation

Nun haben wir die wesentlichen Bausteine der DNS bzw. RNS kennen gelernt: die Zucker Desoxyribose bzw. Ribose sowie die Basen Adenin, Guanin, Cytosin und Thymin bzw. Uracil. Zusätzlich spielt noch die Phosphorsäure H_3PO_4 eine Rolle. Je ein Zucker, eine Base und eine Phosphorsäure reagieren zu einem so genannten *Nukleotid*, das sich dann seinerseits mit anderen Nukleotiden zu einem Polymer verbinden kann.

Dabei wird das Rückgrat aus der Phosphorsäure und einem Zucker gebildet, d.h. der Desoxyribose bei DNS und der Ribose bei RNS. Dabei reagiert die Phosphorsäure (die eine mehrfache Säure ist, da sie als Donator bis zu drei Wasserstoffatome abgeben kann) mit den Hydroxylgruppen der Zucker zu einer Esterbindung. Eine Bindung wird über die Hydroxylgruppe am fünften Kohlenstoffatom, die andere am dritten Kohlenstoffatom der (Desoxy-)Ribose gebildet. Somit ergibt sich für das Zucker-Säure-Rückgrat eine Orientierung.

Die Basen werden am ersten Kohlenstoffatom der (Desoxy-)Ribose über eine glykosidische Bindung (zum bereits erwähnten Voll-Acetal) angebunden. Eine Skizze eines Teilstranges der DNS ist in Abbildung 19 dargestellt. Man beachte, dass das Rückgrat für alle DNS-Stränge identisch ist. Die einzige Variabilität besteht in der Anbindung der Basen an die (Desoxy-)Ribose. Eine Kombination aus Zucker und Base (also ohne eine Verbindung mit der Phosphorsäure) wird als *Nukleosid* bezeichnet.

0.3.4 Komplementarität der Basen

Zunächst betrachten wir die Basen noch einmal genauer. Wir haben zwei Purin-Basen, Adenin und Guanin, sowie zwei Pyrimidin-Basen, Cytosin und Thymin (bzw. Uracil in der RNS). Je zwei dieser Basen sind *komplementär* zueinander. Zum einen sind Adenin und Thymin komplementär zueinander und zum anderen sind es Guanin und Cytosin. Die Komplementarität erklärt sich daraus, dass diese Paare untereinander Wasserstoffbrücken ausbilden können, wie dies in Abbildung 20 illustriert ist.

Dabei stellen wir fest, dass Adenin und Thymin zwei und Cytosin und Guanin drei Wasserstoffbrücken bilden. Aus energetischen Gründen werden diese Basen immer

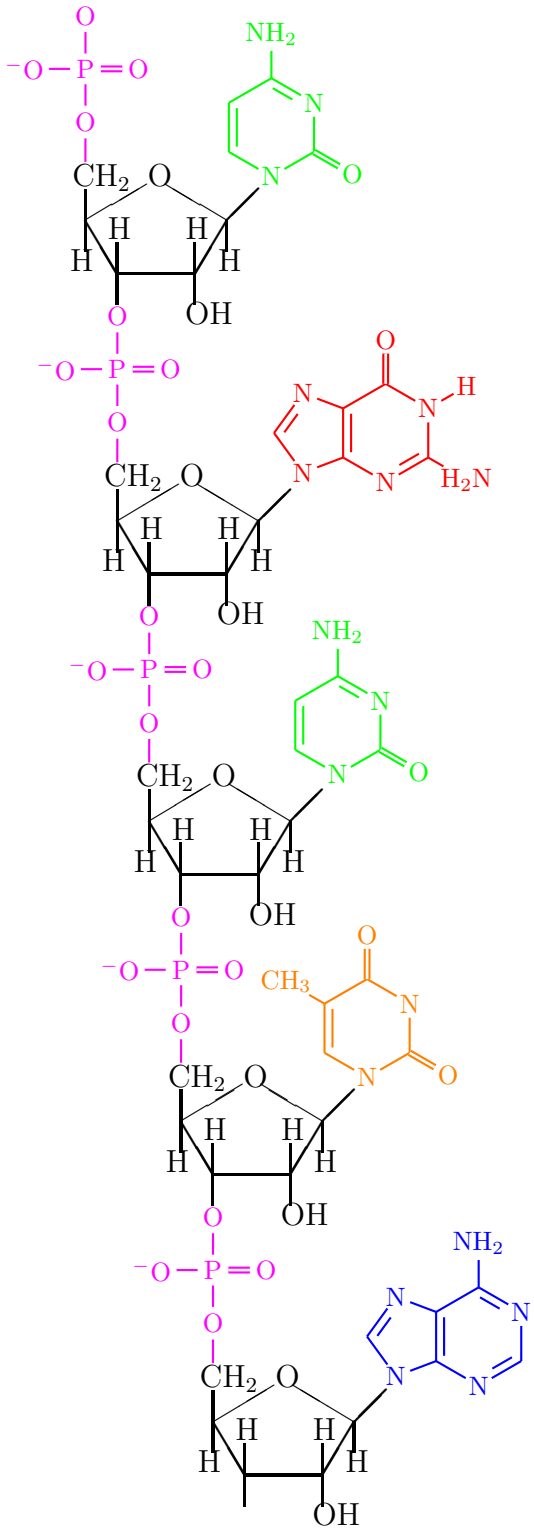


Abbildung 19: Skizze: DNS bzw. RNS als Polymerstrang

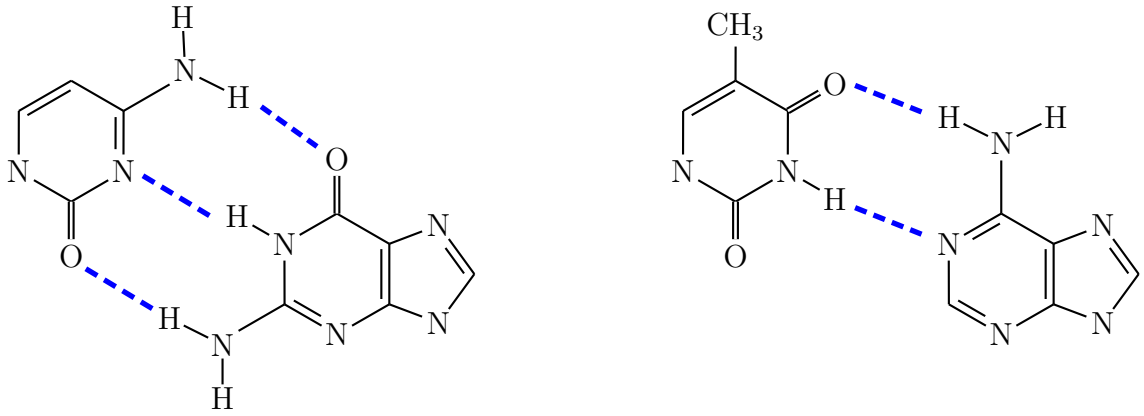


Abbildung 20: Skizze: Wasserstoffbrücken der komplementären Basen

versuchen, diese Wasserstoffbrücken auszubilden. Wir merken, an dass sich auch andere Brückenverbindungen ausbilden können, wie zwischen Thymin und Guanin sowie zwischen Adenin und Cytosin. Diese „falschen“ Wasserstoffbrücken sind aufgrund der Keto-Enol-Tautomerie von Guanin und der Amino-Imino-Tautomerie von Cytosin möglich. Diese sind aus energetischen Gründen zwar eher unwahrscheinlich, können aber dennoch zu Mutationen führen.

Als einfache Merkregel kann man sich merken, dass runde Buchstaben (C und G) bzw. eckige (A und T) zueinander komplementär sind. In der RNS ersetzt Uracil die Base Thymin, so dass man die Regel etwas modifizieren muss. Alles was wie C aussieht ist komplementär (C und G) bzw. alles was wie U aussieht (U und A).

0.3.5 Doppelhelix

Frühe Untersuchungen haben gezeigt, dass in der DNS einer Zelle die Menge von Adenin und Thymin sowie von Cytosin und Guanin immer gleich groß sind. Daraus kam man auf die Schlussfolgerung, dass diese Basen in der DNS immer in Paaren auftreten. Aus dem vorherigen Abschnitt haben wir mit der Komplementarität aufgrund der Wasserstoffbrücken eine chemische Begründung hierfür gesehen. Daraus wurde die Vermutung abgeleitet, dass die DNS nicht ein Strang ist, sondern aus zwei komplementären Strängen gebildet wird, die einander gegenüber liegen.

Aus sterischen Gründen liegen diese beiden Stränge nicht wie Gleise von Eisenbahnschienen parallel nebeneinander (die Schwellen entsprechen hierbei den Wasserstoffbrücken der Basen), sondern sind gleichförmig miteinander verdrillt. Jedes Rückgrat bildet dabei eine Helix (Schraubenlinie) aus. Eine schematische Darstellung ist in Abbildung 21 gegeben.

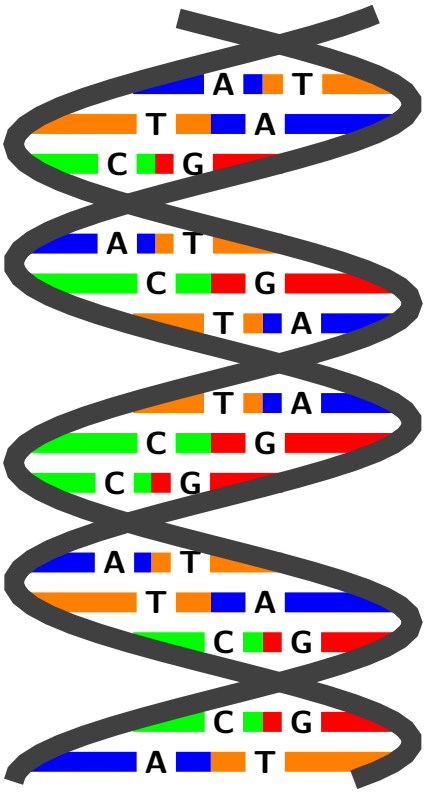


Abbildung 21: Skizze: Doppelhelix der DNS

In einer vollen Drehung sind ungefähr 10 Basenpaare involviert, wobei die Ebenen der Purine bzw. Pyrimidine in etwa orthogonal zur Achse der Doppelhelix liegen. Diese Struktur wurde 1953 von Watson und Crick mit Hilfe der Röntgenkristallographie bestätigt. Es sollte auch noch angemerkt werden, dass unter anderen Randbedingungen auch noch andere Formen von Doppelhelices ausgebildet werden können.

Wie wir schon gesehen haben, besitzt das Rückgrat eines DNS-Strangs eine Orientierung (von 5' nach 3'). Genaue sterische Untersuchungen haben gezeigt, dass die beiden Stränge der DNS innerhalb einer Doppelhelix gegenläufig sind. Läuft also der eine Strang quasi von unten nach oben, so läuft der andere von oben nach unten. Ferner haben die beiden Stränge keinen maximalen Abstand voneinander. Betrachtet man die Doppelhelix der DNS aus etwas „größerem" Abstand (wie etwa in der schematischen Zeichnung in Abbildung 21), so erkennt man etwas, wie ein kleinere und eine größere Furche auf einer Zylinderoberfläche.

Zum Abschluss noch ein paar Fakten zur menschlichen DNS. Die DNS des Menschen ist nicht eine lange DNS, sondern in 46 unterschiedlich lange Teile zerlegt. Insgesamt sind darin etwa 3 Milliarden Basenpaare gespeichert. Jedes Teil der gesamten DNS ist im Zellkern in einem Chromosom untergebracht. Dazu verdrillt und klumpt

sich die DNS noch weiter und wird dabei von Histonen (spezielle Proteine) unterstützt, die zum Aufwickeln dienen. Würde man die gesamte DNS eines Menschen hintereinander ausrollen, so wäre sie etwa 1 Meter(!) lang.

0.4 Proteine

In diesem Abschnitt wollen wir uns um die wichtigsten Bausteine des Lebens kümmern, die *Proteine*.

0.4.1 Aminosäuren

Zunächst einmal werden wir uns mit den *Aminosäuren* beschäftigen, die den Hauptbestandteil der Proteine darstellen. An einem Kohlenstoffatom (dem so genannten *zentralen Kohlenstoffatom* oder auch *α-ständigen Kohlenstoffatom* ist ein Wasserstoffatom, eine Carboxylgruppe (also eine Säure) und eine Aminogruppe gebunden, woraus sich auch der Name ableitet. Die letzte freie Bindung des zentralen Kohlenstoffatoms ist mit einem weiteren Rest gebunden. Hierfür kommen prinzipiell alle

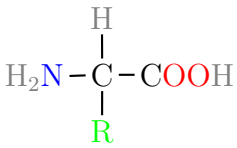


Abbildung 22: Aminosäure

möglichen organischen funktionellen Gruppen in Frage. In der Natur der Proteine kommt jedoch nur eine Auswahl von zwanzig verschiedenen Resten in Betracht. Dabei können die Reste so einfach sein wie ein Wasserstoffatom (Glyzin) oder eine Methylgruppe (Alanin), aber auch recht komplex wie zum Beispiel zwei aromatische Ringe (Tryptophan). In Abbildung 22 ist die Strukturformel einer generischen Aminosäure dargestellt.

In Abbildung 23 sind die Namen der zwanzig in Proteinen auftretenden Aminosäuren und ihre gebräuchlichsten Abkürzungen im so genannten Three-Letter-Code und One-Letter-Code angegeben. Auf die Angabe der genauen chemischen Formeln wollen wir an dieser Stelle verzichten. Hierfür sei auf die einschlägige Literatur verwiesen. In Abbildung 24 sind die grundlegendsten Eigenschaften der einzelnen Aminosäuren schematisch zusammengefasst (nach W.R. Taylor, J. Theor. Biol, 119(2):205–218).

Aminosäure	3LC	1LC
Alanin	Ala	A
Arginin	Arg	R
Asparagin	Asn	N
Asparaginsäure	Asp	D
Cystein	Cys	C
Glutamin	Gln	Q
Glutaminsäure	Glu	E
Glyzin	Gly	G
Histidin	His	H
Isoleuzin	Ile	I
Leuzin	Leu	L
Lysin	Lys	K
Methionin	Met	M
Phenylalanin	Phe	F
Prolin	Pro	P
Serin	Ser	S
Threonin	Thr	T
Tryptophan	Trp	W
Tyrosin	Tyr	Y
Valin	Val	V
Selenocystein	Sec	U
Asparaginsäure oder Asparagin	Asx	B
Glutaminsäure oder Glutamin	Glx	Z
Beliebige Aminosäure	Xaa	X

Abbildung 23: Tabelle: Liste der zwanzig Aminosäuren

Auch hier sind in der Regel (mit Ausnahmen von Glyzin) am zentralen Kohlenstoffatom vier verschiedene Substituenten vorhanden. Somit handelt es sich bei dem zentralen Kohlenstoffatom um ein asymmetrisches Kohlenstoffatom und die Aminosäuren können in zwei enantiomorphen Strukturen auftreten. In der Natur tritt jedoch die L-Form auf, die meistens rechtsdrehend ist! Nur diese kann in der Zelle mit den vorhandenen Enzymen verarbeitet werden.

0.4.2 Peptidbindungen

Auch Aminosäuren besitzen die Möglichkeit mit sich selbst zu langen Ketten zu polymerisieren. Dies wird möglich durch eine so genannte *säureamidartige Bindung* oder auch *Peptidbindung*. Dabei kondensiert die Aminogruppe einer Aminosäure mit der Carboxylgruppe einer anderen Aminosäure (unter Wasserabspaltung) zu einem

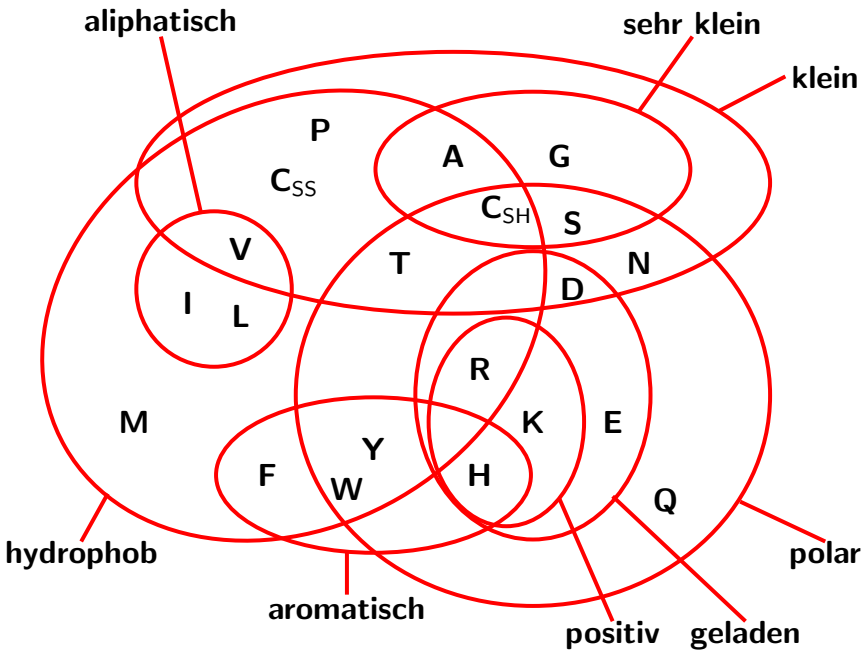


Abbildung 24: Skizze: Elementare Eigenschaften von Aminosäuren

neuen Molekül, einem so genannten *Dipeptid*. Die chemische Reaktionsgleichung ist in Abbildung 25 illustriert.

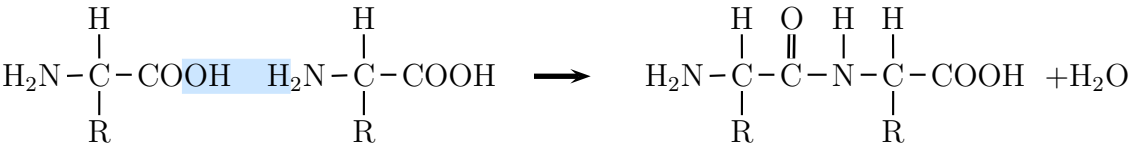


Abbildung 25: Skizze: Säureamidartige oder Peptidbindung

Man beachte, dass das Dipeptid an einem Ende weiterhin eine Aminogruppe und am anderen Ende eine Carboxylgruppe besitzt. Dieser Prozess kann also fortgesetzt werden, so dass sich aus Aminosäuren lange unverzweigte Polymere konstruieren lassen. Solche Polymere aus Aminosäuren nennt man *Polypeptide*. Auch hier bemerken wir wieder, dass ein Polypeptid eine Orientierung besitzt. Wir werden Polypeptide, respektive ihre zugehörigen Aminosäuren immer in der Leserichtung von der freien Aminogruppe zur freien Carboxylgruppe hin orientieren. Ein *Protein* selbst besteht dann aus einem oder mehreren miteinander verbundenen Polypeptiden.

Wir wollen uns nun eine solche Peptidbindung etwas genauer anschauen. Betrachten wir hierzu die Abbildung 26. Von unten links nach oben rechts durchlaufen wir eine Peptidbindung vom zentralen Kohlenstoffatom der ersten Aminosäure über

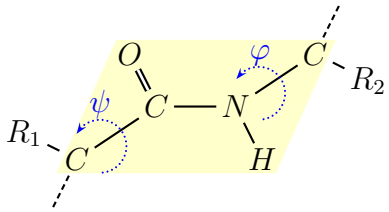


Abbildung 26: Skizze: freie Winkel in der Peptidbindung

das Kohlenstoffatom der ehemaligen Carboxylgruppe über das Stickstoffatom der ehemaligen Aminogruppe der zweiten Aminosäure bis hin zum zentralen Kohlenstoffatom der zweiten Aminosäure.

Auf den ersten Blick könnte man meinen, dass Drehungen um alle drei Bindungen C−C, C−N und N−C möglich wären. Eine genaue Betrachtung zeigt jedoch, dass der Winkel um die $C-N$ Bindung nur zwei Werte, nämlich 0° oder 180°, annehmen kann. Dies wird aus der folgenden Abbildung 27 deutlicher, die für die Bindungen O=C−N die beteiligten Elektronen-Orbitale darstellt. Man sieht hier, dass nicht nur die C=O-Doppelbindung ein π-Orbital aufgrund der Doppelbindung ausbildet, sondern dass auch das Stickstoffatom aufgrund seiner fünf freien Außenelektronen zwei davon in einem nichtbindenden p-Orbital unterbringt. Aus energetischen Gründen ist es günstiger, wenn sich das π-Orbital der Doppelbindung und das p-Orbital des Stickstoffatoms überlagern.

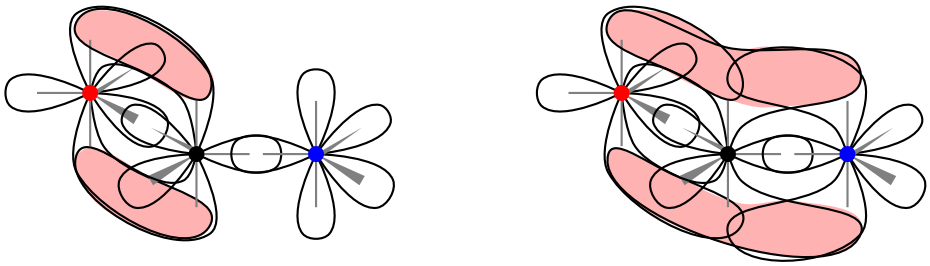


Abbildung 27: Skizze: Elektronenwolken in der Peptidbindung

Somit ist die Bindung zwischen dem Kohlenstoff- und dem Stickstoffatom auf 0° oder 180° festgelegt. In der Regel wird die trans-Konformation gegenüber der cis-Konformation bevorzugt, da dann die variablen Reste der Aminosäuren ziemlich weit auseinander liegen. Eine Ausnahme stellt nur Prolin dar, da hier die Seitenkette eine weitere Bindung mit dem Rückgrat eingeht.

Prinzipiell unterliegen die beiden anderen Winkel keinen Einschränkungen. Auch hier haben Untersuchungen gezeigt, dass jedoch nicht alle Winkel eingenommen werden. Ein Plot, der alle Paare von den beiden übrigen Winkeln darstellt, ist der so genannte *Ramachandran-Plot*, der schematisch in der Abbildung 28 dargestellt ist. Hier sieht man, dass es gewisse ausgezeichnete Gebiete gibt, die mögliche Winkelkombinationen angeben. Wir kommen auf die Bezeichnungen in diesem Plot später noch einmal zurück.

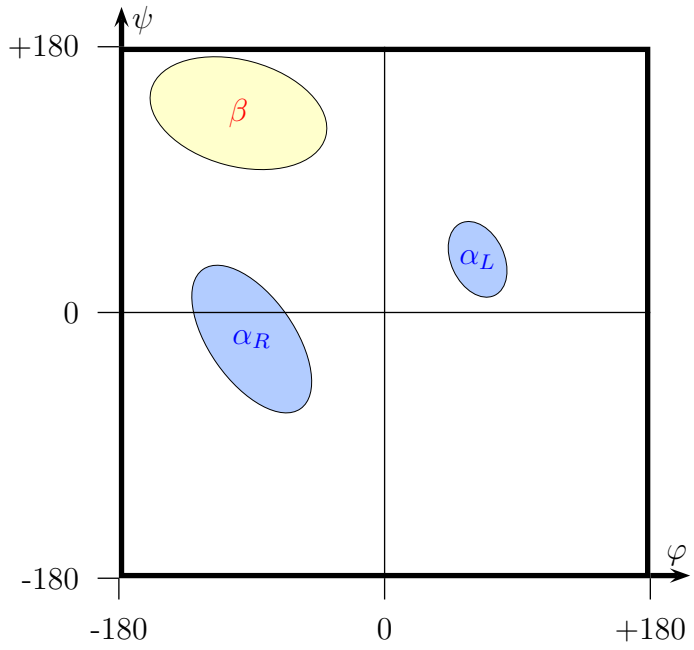


Abbildung 28: Skizze: Ramachandran-Plot (schematische Darstellung)

0.4.3 Proteinstrukturen

Man betrachtet die *Struktur* der Proteine auf vier verschiedenen Ebenen:

0.4.3.1 Primärstruktur

Die *Primärstruktur* (primary structure) eines Proteins ist die Abfolge der beteiligten Aminosäuren des Polypeptids, also seine *Aminosäuresequenz*. Hierbei hält man die Konvention ein, dass man die Aminosäuren von dem Ende mit der freien Aminogruppe her aufschreibt. Für uns ist dann ein Protein, respektive seine Primärstruktur nichts anderes als eine Zeichenreihe über einem zwanzig-elementigen Alphabet.

0.4.3.2 Sekundärstruktur

Als Sekundärstruktur (secondary structure) bezeichnet man Regelmäßigkeiten in der lokalen Struktur des Proteins, die sich nur über einige wenige Aminosäuren erstrecken. Die prominentesten Vertreter hierfür sind die spiralförmige α-*Helix* und der langgestreckte β-Strang (β-*strand*). Ursache für die Ausbildung dieser Sekundärstrukturmerkmale ist vor allem die Stabilisierung durch Wasserstoffbrückenbindungen. Zu einem großen Teil wird die Sekundärstruktur eines Proteinabschnitts durch seine eigene Primärstruktur bestimmt, d.h. bestimmte Aminosäuresequenzen bevorzugen (oder vermeiden) α-Helices, β-Strands oder Loops.

α-Helices: Wie bei der DNS kann ein Protein oder ein kurzes Stück hiervon eine helixartige (spiralförmige) Gestalt ausbilden. Dabei werden die Helices durch Wasserstoffbrücken innerhalb des Polypeptids stabilisiert, die sich zwischen dem Sauerstoffatom der Carbonylgruppe und dem Wasserstoffatom der Aminogruppe der viertnächsten Aminosäure im Peptidstrang ausbilden. Einen solchen Teil eines Peptids nennt man α-Helix. Dabei entfallen auf eine volle Drehung etwa 3,6 Aminosäuren. Hierbei hat die Helix in der Regel eine Rechtsdrehung, weil bei einer Linksdrehung die sterische Hinderung deutlich größer ist. Die zugehörigen Winkelpaare der Peptidbindung entsprechen im Ramachandran-Plot in Abbildung 28 dem mit α_R markierten Bereich. Einige wenige Helices bilden eine Linksdrehung aus. Die zugehörigen Winkelpaare sind im Ramachandran-Plot mit α_L gekennzeichnet.

Beispielsweise sind die Haare aus Proteinen gebildet, die eine Helix bilden. Auch hier können wie bei der DNS mehrere (sogar mehr als zwei) Helices zusammen verdrillt sein. Ebenso seien Proteine erwähnt, die in Muskeln eine wichtige Rolle spielen.

π- und 3_{10}-Helices: In seltenen Fällen treten Abwandlungen der α-Helix auf, bei denen die Wasserstoffbrücken nicht zwischen den Aminosäuren n und $n+4$, sondern zwischen den Aminosäuren n und $n+3$ oder $n+5$ gebildet werden. In diesen Fällen ist die Helix also etwas mehr oder etwas weniger verdreht. Man nennt diese beiden Formen die 3_{10}- und die π-Helix. Der Name 3_{10}-Helix entstammt dabei der Tatsache, dass die Helix 3 Aminosäuren pro Umdrehung enthält und dass zwischen den beiden Enden einer Wasserstoffbrücke zehn Atome (incl. Wasserstoffatom) liegen. In dieser Nomenklatur (nach Linus Pauling und Robert Corey) würde man die α-Helix mit 3.6_{13} und die π-Helix mit 4.4_{16} bezeichnen.

β-Strands: Eine andere Struktur sind langgezogene Bereiche von Aminosäuren. Meist lagern sich hier mehrere Stränge (oft von verschiedenen Polypeptidketten, aber durchaus auch nur von einer einzigen) nebeneinander an und bilden

so genannte β-Sheets oder β-Faltblätter aus. Hierbei ist zu beachten, dass sich diese wie Spaghetti nebeneinander lagern. Dies kann entweder parallel oder antiparallel geschehen (Polypeptide haben ja eine Richtung!). Auch hier werden solche Faltblätter durch Wasserstoffbrücken zwischen den Amino- und Carbonylgruppen stabilisiert. Diese Struktur heißt Faltblatt, da es entlang eines Polypeptid-Strangs immer wieder auf und ab geht, ohne insgesamt die Richtung zu ändern. Bilden sich ganze β-Faltblätter aus, so sehen diese wie ein gefaltetes Blatt aus, wobei die einzelnen Polypeptide quer zu Faltungsrichtung verlaufen. Die zugehörigen Winkelpaare der Peptidbindung entsprechen im Ramachandran-Plot in Abbildung 28 dem mit β markierten Bereich. Beispielsweise tauchen im Seidenfibroin (Baustoff für Seide) fast nur β-Faltblätter auf.

Reverse Turns: Zum Schluss seien noch kurze Sequenzen (von etwa fünf Aminosäuren) erwähnt, die einfach nur die Richtung des Polypeptids umkehren. Diese sind beispielsweise in antiparallelen β-Faltblätter zu finden, um die einzelne β-Strands zu einem β-Sheet anordnen zu können.

0.4.3.3 Supersekundärstruktur

Oft lagern sich zwei oder drei Sekundärstrukturelemente zu sogenannten *Motifs* zusammen.

Hairpins Reverse Turns, die zwischen zwei nebeneinander liegenden antiparallelen β-Strands liegen und deshalb die Form einer Haarnadel nachbilden

Coiled coils bestehen aus zwei verdrillten α-Helices und spielen eine wichtige Rolle in Faser-Proteinen

0.4.3.4 Tertiärstruktur

Die *Tertiärstruktur* (tertiary structure) ist die *Konformation*, also die räumliche Gestalt, eines einzelnen Polypeptids. Sie beschreibt, wie die Elemente der Sekundär- und Supersekundärstruktur sich zu so genannten *Domains* zusammensetzen. Hier ist für jedes Atom (oder Aminosäure) die genaue relative Lage zu allen anderen bekannt. Tertiärstrukturen sind insbesondere deshalb wichtig, da für globuläre Proteine die räumliche Struktur für ihre Wirkung wichtig ist (zumindest in fest umschriebenen Reaktionszentren eines Proteins).

0.4.3.5 Quartärstruktur

Besteht ein Protein aus mehreren Polypeptidketten, wie etwa Hämoglobin, dann spricht man von der *Quartärstruktur* (quaternary structure) eines Proteins. Proteine können aus einem einzelnen Polypeptid oder aus mehreren (gleichen oder unterschiedlichen) Polypeptidketten bestehen.

0.5 Der genetische Informationsfluss

Bislang haben wir die wichtigsten molekularbiologischen Bausteine kennen gelernt. Die DNS, die die eigentliche Erbinformation speichert, und sich in der Regel zu den bekannten Chromosomen zusammenwickelt. Über die Bedeutung der zur DNS strukturell sehr ähnlichen RNS werden wir später noch kommen. Weiterhin haben wir mit den Proteinen die wesentlichen Bausteine des Lebens kennen gelernt. Jetzt wollen wir aufzeigen, wie die in der DNS gespeicherte genetische Information in den Bau von Proteinen umgesetzt werden kann, d.h. wie die Erbinformation weitergegeben (vererbt) wird.

0.5.1 Replikation

Wie wird die genetische Information überhaupt konserviert, oder anders gefragt, wie kann man die Erbinformation kopieren? Dies geschieht durch eine Verdopplung der DNS. An einer bestimmten Stelle wird die DNS entspiralisiert und die beiden Stränge voneinander getrennt, was dem Öffnen eines Reißverschlusses gleicht. Diese Stelle wird auch *Replikationsgabel* genannt. Dann wird mit Hilfe der DNS-Polymerase an beiden Strängen die jeweils komplementäre Base oder genauer das zugehörige Nukleotid mit Hilfe der Wasserstoffbrücken angelagert und die Phosphatsäuren bilden mit den nachfolgenden Zuckern jeweils eine Esterbindung zur Ausbildung des eigentlichen Rückgrates aus. Dies ist schematisch in Abbildung 29 dargestellt, wobei die neu konstruierten DNS-Stücke rot dargestellt sind.

Die Polymerase, die neue DNS-Stränge generiert, hat dabei nur ein Problem. Sie kann einen DNS-Strang immer nur in der Richtung vom 5'-Ende zum 3'-Ende synthetisieren. Nach dem Öffnen liegt nun ein Strang in Richtung der Replikationsgabel (in der ja die Doppelhelix entspiralisiert wird und die DNS aufgetrennt wird) in Richtung vom 3'-Ende zum 5'-Ende vor, der andere jedoch in Richtung vom 5'-Ende zum 3'-Ende, da die Richtung der DNS-Stränge in der Doppelhelix ja antiparallel ist.

Der Strang in Richtung vom 3'-Ende zum 5'-Ende in Richtung auf die Replikationsgabel zu, lässt sich jetzt leicht mit der DNS-Polymerase ergänzen, da dann die

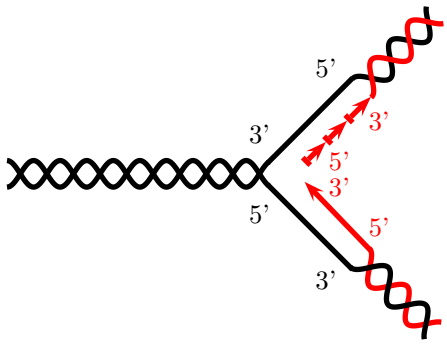


Abbildung 29: Skizze: DNS-Replikation

Synthese selbst in Richtung vom 5’-Ende zum 3’-Ende erfolgt und beim weiteren Öffnen der DNS einfach weiter synthetisiert werden kann.

Für den Strang in Richtung vom 5’-Ende zum 3’-Ende hat man herausgefunden, dass auch hier die Synthese in Richtung vom 5’-Ende zum 3’-Ende erfolgt. Die DNS-Polymerase wartet hier solange, bis ein hinreichend langes Stück frei liegt und ergänzt den DNS Strang dann von der Replikationsgabel weg. Das bedeutet, dass die DNS hier immer in kleinen Stücken synthetisiert wird und nicht im ganzen wie am komplementären Strang. Die dabei generierten kleinen DNS-Teilstränge werden nach ihrem Entdecker *Okazaki-Fragmente* genannt.

0.5.2 Transkription

Damit die in der DNS gespeicherte Erbinformationen genutzt werden kann, muss diese erst einmal abgeschrieben oder kopiert werden. Der Prozess ist dabei im Wesentlichen derselbe wie bei der Replikation. Hierbei wird allerdings nicht die gesamte DNS abgeschrieben, sondern nur ein Teil. Der hierbei abgeschriebene Teil bildet dann nicht eine Doppelhelix mit der DNS aus, sondern löst sich am nichtaktiven Ende wieder, so dass die beiden aufgetrennten Stränge der DNS wieder eine Doppelhelix bilden können.

Hierbei ist zu beachten, dass der abgeschriebene Teil keine DNS, sondern eine RNS ist. Hier wird also Ribose statt Desoxyribose verwendet und als Base Uracil statt Thymin. Die abgeschriebene RNS wird als *Boten-RNS* bzw. *messenger RNA* bezeichnet, da sie als Überbringer der Erbinformation dient.

In prokaryontischen Zellen ist der Vorgang damit abgeschlossen. In eukaryontischen Zellen ist der Vorgang etwas komplizierter, da die Chromosomen im Zellkern beheimatet sind und damit auch die Boten-RNS. Da die Boten-RNS jedoch außerhalb des Zellkerns weiterverarbeitet wird, muss diese erst noch durch die Membran des Zellkerns wandern.

Des Weiteren hat sich in eukaryontischen Zellen noch eine Besonderheit ausgebildet. Die Erbinformation steht nicht kontinuierlich auf der DNS, sondern beinhaltet dazwischen Teilstücke ohne Erbinformation. Diese müssen vor einer Weiterverarbeitung erst noch entfernt werden.

Es hat sich gezeigt, dass dieses Entfernen, *Spleißen* (engl. *Splicing*) genannt, noch im Zellkern geschieht. Dabei werden die Stücke, die keine Erbinformation tragen und *Introns* genannt werden, aus der Boten-RNS herausgeschnitten. Die anderen Teile, *Exons* genannt, werden dabei in der selben Reihenfolge wie auf der DNS aneinander gereiht. Die nach dem Spleißen entstandene Boten-RNS wird dann als *reife Boten-RNS* oder als *mature messenger RNA* bezeichnet.

Für Experimente wird oft aus der mRNA wieder eine Kopie als DNS dargestellt. Diese wird als *cDNS* bzw. *komplementäre DNS* (engl. *cDNA* bzw. *complementary DNA*) bezeichnet. Diese entspricht dann dem Original aus der DNS, wobei die Introns bereits herausgeschnitten sind. Die originalen Gene aus der DNS mit den Introns werden auch als *genetische DNS* (engl. *genetic DNA*) bezeichnet.

Dazu betrachten wir die schematische Darstellung des genetischen Informationsflusses innerhalb einer Zelle (hier einer eukaryontischen) in Abbildung 30.

0.5.3 Translation

Während der *Translation* (*Proteinbiosynthese*) wird die in der DNS gespeicherte und in der reifen Boten-RNS zwischengespeicherte komplementäre Erbinformation in Proteine übersetzt. Dies geschieht innerhalb der Ribosomen, die sich wie zwei Semmelhälften auf den Anfang der Boten-RNS setzen und den RNS-Strang in ein Protein übersetzen. Ribosomen selbst sind aus Proteinen und RNS, so genannter *ribosomaler RNS* oder kurz *rRNS* (engl. *ribosomal RNA*, *rRNA*), zusammengesetzt.

Wir erinnern uns, dass die RNS bzw. DNS im Wesentlichen durch die vier Basen Adenin, Guanin, Cytosin und Uracil bzw. Thymin die Information trägt. Nun ist also die in der RNS gespeicherte Information über einem vierelementigen Alphabet codiert, wobei ein Protein ein Polymer ist, das aus zwanzig verschiedenen Aminosäuren gebildet wird. Wir müssen also noch die Codierung eines zwanzig-elementigen durch ein vier-elementiges Alphabet finden.

Offensichtlich lassen sich nicht alle Aminosäuren durch je zwei Basen codieren. Es muss also Aminosäuren geben, die durch mindestens drei Basen codiert werden. Es hat sich herausgestellt, dass der Code immer dieselbe Länge hat und somit jeweils drei Basen, ein so genanntes *Basen-Triplett* oder *Codon*, jeweils eine Aminosäure codiert.

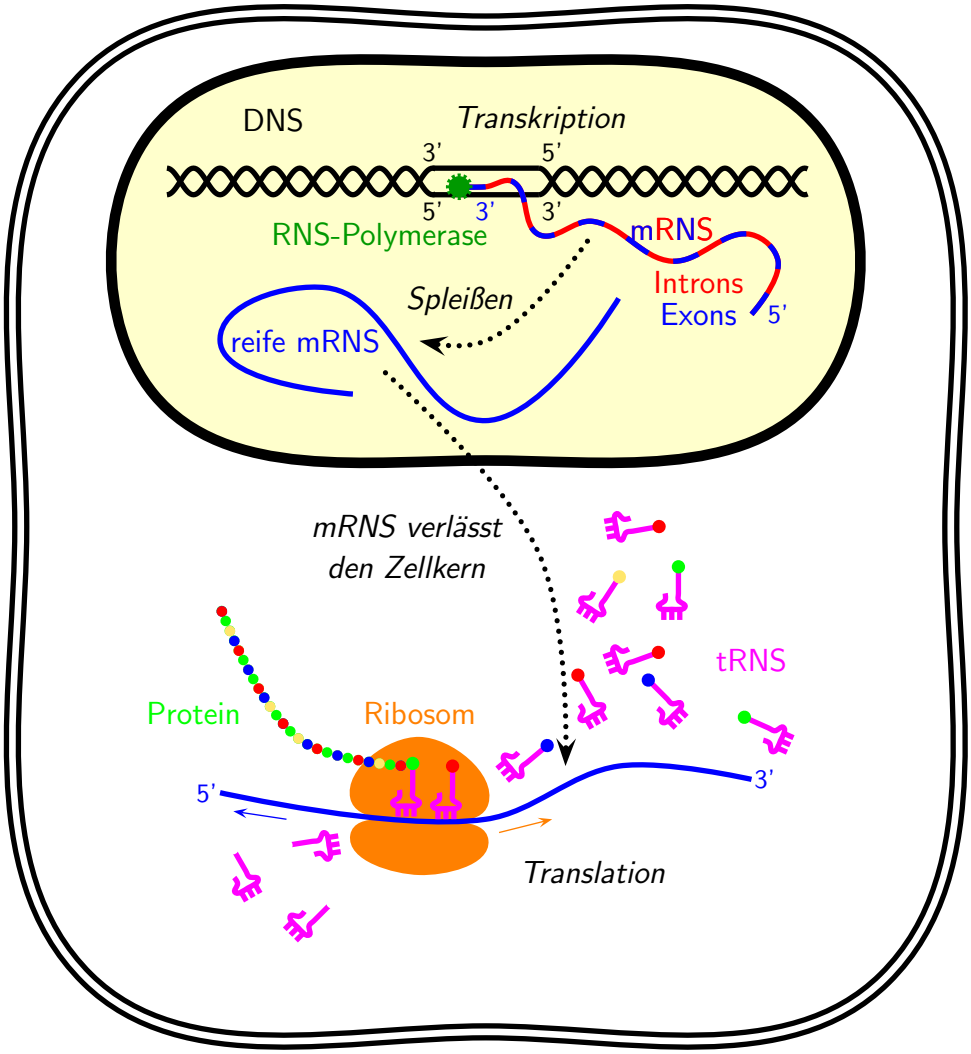


Abbildung 30: Skizze: Transkription und Transaltion in einer Zelle

Da es 64 verschiedene Tripletts aber nur zwanzig Aminosäuren gibt, werden einige Aminosäuren durch mehrere Tripletts codiert. In der folgenden Abbildung 31 ist der Code für die Umwandlung von Tripletts in Aminosäuren angegeben. In Abbildung 31 steht das erste Zeichen des Tripletts links, das zweite oben und das dritte in der rechten Spalte. AGU bzw. AGC codiert also Serin.

Zum genetischen Code ist noch folgendes zu sagen. Es gibt auch spezielle Stopp-Codons, die den Ribosomen mitteilen, dass die Übersetzung der RNS in ein Protein zu Ende ist: Diese sind UAG, UAA und UGA. Ebenfalls gibt es auch ein so genanntes Start-Codon, das jedoch nicht eindeutig ist. AUG übernimmt sowohl die Rolle der Codierung von Methionin als auch dem Anzeigen an das Ribosom, dass hier mit der

	U	**C**	**A**	**G**	
U	Phe	Ser	Tyr	Cys	**U**
	Phe	Ser	Tyr	Cys	**C**
	Leu	Ser	STOP	STOP	**A**
	Leu	Ser	STOP	Trp	**G**
C	Leu	Pro	His	Arg	**U**
	Leu	Pro	His	Arg	**C**
	Leu	Pro	Gln	Arg	**A**
	Leu	Pro	Gln	Arg	**G**
A	Ile	Thr	Asn	Ser	**U**
	Ile	Thr	Asn	Ser	**C**
	Ile	Thr	Lys	Arg	**A**
	Met	Thr	Lys	Arg	**G**
G	Val	Ala	Asp	Gly	**U**
	Val	Ala	Asp	Gly	**C**
	Val	Ala	Glu	Gly	**A**
	Val	Ala	Glu	Gly	**G**

Abbildung 31: Tabelle: Der genetische Code

Übersetzung begonnen werden kann. Ebenso ist dieser Code universell, d.h. fast alle Lebewesen benutzen diesen Code. Bislang sind nur wenige Ausnahmen bekannt, die einen anderen Code verwenden, der aber diesem weitestgehend ähnlich ist.

Auch sollte erwähnt werden, dass die Redundanz des Codes (64 Tripletts für 20 Aminosäuren) zur Fehlerkorrektur ausgenutzt wird. Beispielsweise ist die dritte Base für die Decodierung einiger Aminosäuren völlig irrelevant, wie für Alanin, Glycin, Valin und andere. Bei anderen Mutationen werden in der Regel Aminosäuren durch weitestgehend ähnliche (in Bezug auf Größe, Hydrophilie, Ladung oder ähnliches) ersetzt. Auch werden häufig auftretende Aminosäuren durch mehrere Tripletts, selten auftretende nur durch eines codiert.

Ebenfalls sollte man hierbei noch darauf hinweisen, dass es für jeden RNS-Strang eigentlich drei verschiedene Leseraster gibt. Dem RNS-Strang $s_1 \cdots s_n$ an sich sieht man nicht an, ob das codierte Gen in $(s_1 s_2 s_3)(s_4 s_5 s_6) \cdots$, $(s_2 s_3 s_4)(s_5 s_6 s_7) \cdots$ oder $(s_3 s_4 s_5)(s_6 s_7 s_8) \cdots$ codiert ist. Das Start-Codon kann dabei jedoch Hilfe leisten.

Zum Schluss bleibt nur noch die Übersetzung im Ribosom zu beschreiben. Dabei hilft die so genannte *Transfer-RNS* oder *tRNS*. Die Transfer-RNS besteht im Wesentlichen aus RNS und einer Bindungsstelle für eine Aminosäure. Dabei befinden sich drei Basen an exponierter Stelle, die den drei komplementären Basen der zugehörigen Aminosäure entsprechen, die an der Bindungsstelle angebunden ist.

Im Ribosom werden dann jeweils die komplementären tRNS zum betrachteten Triplett der mRNS mittels der Wasserstoffbrücken angebunden. Dabei wird anschließend die neue Aminosäure mittels der säureamidartigen Bindung an den bereits synthetisierten Polypeptid-Strang angebunden. Die tRNS, die den bereits synthetisierten Polypeptid-Strang festhielt, wird dann freigegeben, um in der Zelle wieder mit einer neuen, zum zugehörigen Triplett gehörigen Aminosäure, aufgeladen zu werden.

0.5.4 Das zentrale Dogma

Aus der bisherigen Beschreibung lässt sich die folgende Skizze für den genetischen Informationsfluss ableiten. Die genetische Information wird in der DNS gespeichert und mit Hilfe der Replikation vervielfacht. Mit Hilfe der Transkription wird die genetische Information aus der DNS ausgelesen und in die RNS umgeschrieben. Aus der RNS kann dann mit Hilfe der Translation die genetische Information in die eigentlichen Bausteine des Lebens, die Proteine, übersetzt werden. Damit ist der genetische Informationsfluss eindeutig von der DNS über die RNS zu den Proteinen gekennzeichnet. Dies wird als das zentrale Dogma der Molekularbiologie bezeichnet.

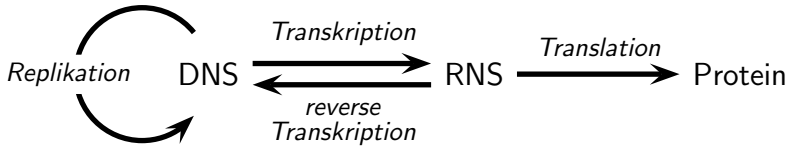


Abbildung 32: Skizze: Das zentrale Dogma der Molekularbiologie

In der Biologie gibt es kaum eine Regel ohne Ausnahmen. Trotz des zentralen Dogmas ist auch ein Informationsfluss in die umgekehrte Richtung möglich. Auch aus der RNS kann modifizierte genetische Erbinformation wieder zurück in die DNS eingebaut werden. Das zentrale Dogma ist in Abbildung 32 noch einmal schematisch dargestellt.

0.5.5 Promotoren

Für den letzten Abschnitt müssen wir uns nur noch überlegen, wie man die eigentliche Erbinformation, die Gene, auf der DNS überhaupt findet. Dazu dienen so genannte *Promotoren*. Dies sind mehrere kurze Sequenzen vor dem Beginn des

eigentlichen Gens, die die RNS-Polymerase überhaupt dazu veranlassen unter bestimmten Bedingungen die spiralisierte DNS an dieser Stelle aufzuwickeln und die beiden DNS-Stränge zu trennen, so dass das Gen selbst transkribiert werden kann. In Bakterien ist dies sehr einfach, da es dort im Wesentlichen nur einen Promotor gibt, der in Abbildung 33 schematisch dargestellt ist. In höheren Lebewesen und insbesondere in eukaryontischen Zellen sind solche Promotoren weitaus komplexer und es gibt eine ganze Reihe hiervon.

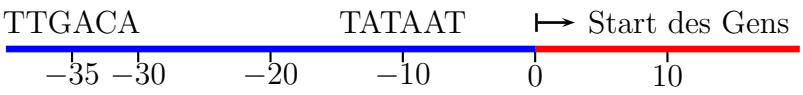


Abbildung 33: Skizze: Promotoren in Bakterien

0.6 Biotechnologie

In diesem Abschnitt wollen wir einige der wichtigsten biotechnologischen Methoden vorstellen, die für uns im Folgenden für eine informatische Modellbildung wichtig sein werden.

0.6.1 Hybridisierung

Mit *Hybridisierung* wird die Aneinanderlagerung zweier DNS-Stränge bezeichnet, die aufgrund der Komplementarität ihrer Basen über Wasserstoffbrücken gebildet wird. Dies haben wir prinzipiell schon bei der Replikation und Transkription kennen gelernt.

Eine Hybridisierung kann dazu ausgenutzt werden, um festzustellen, ob sich eine bestimmte, in der Regel recht kurze Teilsequenz innerhalb eines DNS-Strangs befindet. Dazu wird eine kurze Teilsequenz synthetisiert und an einem Ende markiert, z.B. mit einem fluoreszierenden oder radioaktiven Stoff. Werden die kurzen Teilsequenzen mit den durch Klonierung verfielfachten DNS-Strängen zusammengebracht, können die kurzen Sequenzen mit dem DNS-Strang hybridisieren. Nach Entfernung der kurzen synthetisierten Stücke kann dann festgestellt werden, ob die langen DNS-Stränge fluoreszent oder radioaktiv sind. Letzteres ist genau dann der Fall, wenn eine Hybridisierung stattgefunden hat.

0.6.2 Klonierung

Für biologische Experimente wird oft eine Vielzahl von identischen Kopien eines DNS-Stückes benötigt. Solche Vervielfältigungen lassen sich mit Hilfe niederer Organismen erledigen. Dazu wird das zu vervielfältigende DNS-Stück in die DNS des Organismus eingesetzt und dieser vervielfältigt diese wie seine eigene DNS. Je nachdem, ob Plasmide, Bakterien oder Hefe (engl. yeast) verwendet werden, spricht man vom *plasmid* (*PAC*), *bacterial* (*BAC*) oder *yeast artificial chromosomes* (*YAC*).

Die bei PACs verwendeten Plasmide sind ringförmige DNS-Stränge, die in Bakterien auftreten. Bei jeder Zellteilung wird dabei auch der zu klonierende, neu eingesetzte DNS-Strang vervielfältigt. Hierbei können jedoch nur Stränge bis zu 15.000 Basenpaaren kloniert werden.

Bei BACs werden Phagen (ein Virus) verwendet. Die infizierten Wirtszellen (Bakterien) haben dann das zu klonierende DNS-Stück, das in die Phage eingesetzt wurde, vervielfältigt. Hier sind Vervielfältigungen von bis zu 25.000 Basenpaaren möglich.

Bei YACs wird die gewöhnliche Brauerhefe zur Vervielfältigung ausgenutzt, in die die gewünschten DNS-Teilstücke eingebracht werden. Hierbei sind Vervielfältigungen bis zu 1 Million Basenpaaren möglich.

0.6.3 Polymerasekettenreaktion

Eine andere Art der Vervielfältigung ist mit Hilfe der *Polymerasekettenreaktion* (engl. *polymerase chain reaction*) möglich. Diese hatten wir ja schon bei Replikation von DNS-Doppelhelices kennen gelernt. Wir müssen von dem zu vervielfältigenden Bereich nur die Sequenzen der beiden Endstücke von etwa 10 Basenpaaren kennen. Diese kurzen Stücke werden *Primer* genannt

Zuerst werden die DNS-Stränge der Doppelhelix durch Erhitzen aufgespalten. Dann werden sehr viele komplementäre Sequenzen der Primer zugegeben, so dass sie an die Primer der DNS hybridisieren können. Mit Hilfe der Polymerase werden dann ab den Primern in die bekannte Richtung vom 5'-Ende zum 3'-Ende die Einzelstränge der aufgesplitteten DNS zu einem Doppelstrang vervollständigt. Dies ist in Abbildung 34 schematisch dargestellt. Dabei ist das zu vervielfältigende DNS-Stück grün, die Primer rot und der Rest der DNS grau dargestellt. Die Pfeile geben die Synthetisierungssrichtung der Polymerase an.

Einziges Problem ist, dass bei der ersten Anwendung die Polymerase immer bis zum Ende des Strangs läuft. Es wird also mehr dupliziert als gewünscht. Nun kann man dieses Experiment mehrfach (50 Mal) wiederholen. In jedem Schritt werden dabei

Abbildung 34: Skizze: Polymerasekettenreaktion

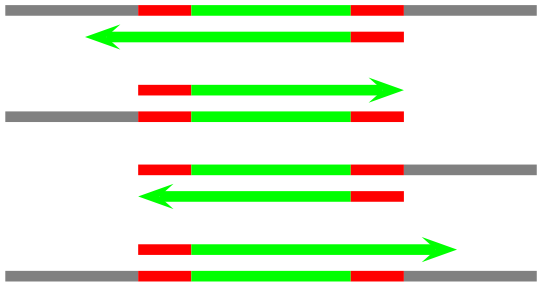


Abbildung 35: Skizze: PCR: nach der 2. Verdopplung

die vorher synthetisierten Doppelstränge verdoppelt. Nach n Phasen besitzt man also 2^n Doppelstränge!

Da nach der ersten Verdopplung bereits jeweils ein unnützes Ende nicht kopiert wurde, überlegt man sich leicht, dass nach der zweiten Verdopplung bereits zwei (von vier) Doppelsträngen nur den gewünschten Bereich verdoppelt haben. Zum Schluss ist jeder zweite DNS-Doppelstrang eine Kopie des gewünschten Bereichs und alle anderen sind nur im gewünschten Bereich doppelsträngig (ansonsten einsträngig, mit zwei Ausnahmen).

Mit dieser Technik lassen sich also sehr schnell und recht einfach eine Vielzahl von Kopien eines gewünschten, durch zwei kurze Primer umschlossenen DNS-Teilstücks herstellen.

0.6.4 Restriktionsenzyme

Restriktionsenzyme sind spezielle *Enzyme* (Proteine, die als Katalysator wirken), die eine DNS-Doppelhelix an bestimmten Stellen aufschneiden können. Durch solche Restriktionsenzyme kann also eine lange DNS geordnet in viele kurze Stücke zerlegt werden. Eines der ersten gefundenen Restriktionsenzyme ist EcoRI, das im Bakterium Escherichia Coli auftaucht. Dieses erkennt das Muster GAATTC. In Abbildung 36 ist dieses Muster in der Doppelhelix der DNS noch einmal schematisch mit den Bruchstellen dargestellt. Man beachte hierbei, dass die Sequenz zu sich selbst komplementär ist. Es handelt sich also um ein so genanntes *komplementäres Palindrom*.

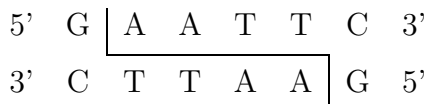


Abbildung 36: Skizze: Restriktionsenzym mit Muster $GAATTC$

0.6.5 Sequenzierung kurzer DNS-Stücke

In diesem Abschnitt wollen wir kurz die gebräuchlichsten Techniken zur Sequenzierung von DNS darstellen. Mit *Sequenzierung* ist das Herausfinden der Abfolge der vier verschiedenen Basen in einem gegebenen DNS-Strang gemeint.

Die Grundidee ist die Folgende: Zuerst wird für das zu sequenzierende Teilstück mit Hilfe der Klonierung eine ausreichende Anzahl identischer Kopien erzeugt. Dann werden die klonierten Teilstücke in vier Gruppen (quasi für jede Base eine) eingeteilt. In jeder Gruppe werden an jeweils einer Base die Teilstücke aufgebrochen. Zu Beginn hat man eines der Enden mit Hilfe eines fluoreszierenden oder radioaktiven Stoffes markiert. Im Folgenden interessieren nur die Bruchstücke, die das markierte Ende besitzen, wobei die anderen Bruchstücke jedoch nicht entfernt werden. Mit Hilfe der so genannten *Elektrophorese* werden die verschieden langen Bruchstücke getrennt.

Die Elektrophorese nutzt aus, dass unter bestimmten Randbedingungen die DNS-Bruchstücke nicht elektrisch neutral, sondern elektrisch geladen sind. Somit kann man die DNS-Bruchstücke mit Hilfe eines elektrischen Feldes wandern lassen. Dazu werden diese innerhalb eines Gels gehalten, dessen Zähflüssigkeit es erlaubt, dass sie sich überhaupt, aber auch nicht zu schnell bewegen. Da die Bruchstücke alle dieselbe Ladung tragen, aber aufgrund ihrer Länge unterschiedlich schwer sind, haben sie innerhalb des angelegten elektrischen Feldes eine unterschiedliche Wanderungsgeschwindigkeit. Die kurzen wandern naturgemäß sehr schnell, während die langen Bruchstücke sich kaum bewegen.

Führt man dieses Experiment gleichzeitig für alle vier Gruppen (also für jede Base) getrennt aus, so erhält man ein Bild der gewanderten Bruchstücke. Dies ist schematisch in Abbildung 37 dargestellt, das man sich als eine Fotografie einer Gruppe radioaktiv markierter Stücke vorstellen kann (auch wenn dies heute mit fluoreszierenden Stoffen durchgeführt wird). Hier ist links die (noch nicht bekannte) Sequenz der einzelnen Bruchstücke angegeben. Mit rot ist speziell das Ergebnis für die Base Adenin hervorgehoben. In den experimentellen Ergebnissen gibt es an sich jedoch keine farblichen Unterscheidungen.

Man kann nun die relativen Positionen einfach feststellen und anhand der Belichtung des Films feststellen in welcher Gruppe sich ein Bruchstück befindet und somit die Base an der entsprechenden Position ablesen. Es ist hierbei zu berücksichtigen, dass

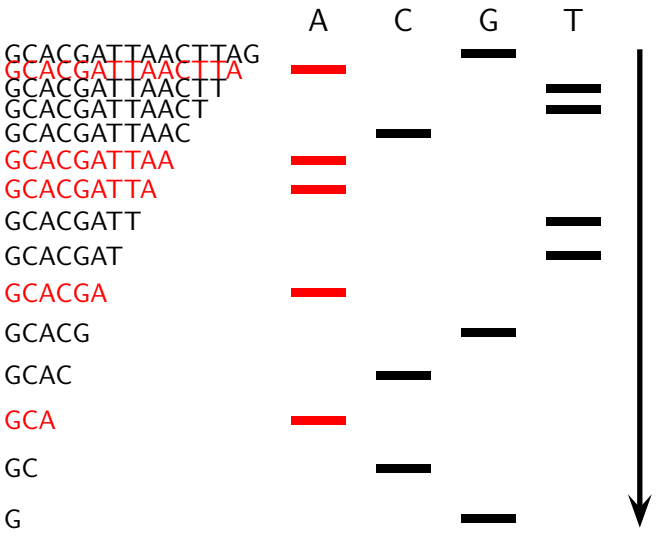


Abbildung 37: Skizze: Sequenzierung nach Sanger

die Wanderungsgeschwindigkeit umgekehrt proportional zu dessen Masse (und somit im Wesentlichen zur Länge des betrachteten Bruchstücks ist). Während also der Abstand der Linie der Bruchstücke der Länge eins und der Linie der Bruchstücke der Länge zwei relativ groß sind, ist der Abstand der Linie der Bruchstücke der Länge 100 und der Linie der Bruchstücke der Länge 101 relativ kurz.

0.6.5.1 Sanger-Methode

Das Aufspalten kann über zwei prinzipiell verschiedene Methoden erfolgen. Zum einen kann man bei der Vervielfältigung innerhalb einer Klasse dafür sorgen, dass neben der entsprechenden Base auch eine modifizierte zur Verfügung steht, an der die Polymerase gestoppt wird. Der Nachteil hierbei ist, dass längere Sequenzen immer weniger werden und man daher lange Sequenzen nicht mehr so genau erkennen kann. Diese Methode wird nach ihrem Erfinder auch *Sanger-Methode* genannt.

0.6.5.2 Maxam-Gilbert-Methode

Zum anderen kann man mit Hilfe chemischer Stoffe die Sequenzen entweder nach Adenin, einer Purinbase (Adenin und Guanin), Thymin oder einer Pyrimidin-Base (Thymin und Cytosin) aufbrechen. Man erhält also nicht für jede Base einen charakteristischen Streifen, sondern bei Adenin oder Thymin jeweils zwei, wie in der folgenden Abbildung illustriert. Nichts desto trotz lässt sich daraus die Sequenz

ablesen (siehe auch Abbildung 38). Diese Methode wird nach ihren Erfinder auch *Maxam-Gilbert-Methode* genannt.

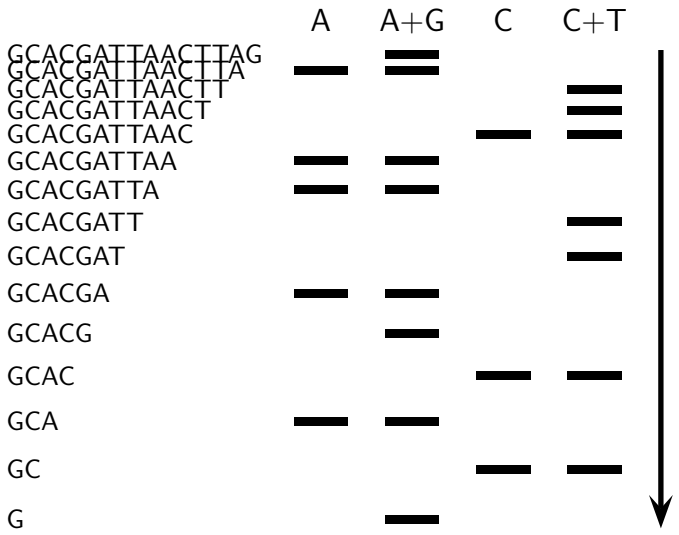


Abbildung 38: Skizze: Sequenzierung nach Maxam-Gilbert

Heutzutage wird in der Regel die Sanger-Methode mit fluoreszierenden Stoffen angewendet, die sich dann gleichzeitig in großen Sequenzierautomaten sehr leicht automatisch anwenden lässt. Dennoch lassen sich auch heute nur DNS-Sequenzen der Länge 500 gut sequenzieren. Mit den bekannten Methoden sind auch in Zukunft bei einem entsprechenden technologischen Fortschritt Sequenzierungen von mehr als 1000 Basenpaaren nicht denkbar.

0.6.6 Sequenzierung eines Genoms

Im letzten Abschnitt haben wir gesehen, wie sich kurze DNS-Stücke sequenzieren lassen und dass sich diese Methoden nicht auf beliebig lange DNS-Sequenzen ausdehnen lassen. In diesem letzten Abschnitt wollen wir uns überlegen, wie man ein ganzes Genom sequenzieren kann.

0.6.6.1 Primer Walking

Beim *Primer Walking* ist die Idee, dass man die ersten 500 Basen des Genoms sequenziert. Kennt man diese, so kann man am Ende einen möglichst eindeutigen Primer der Länge ca. zwanzig ablesen und mit der Polymerasekettenreaktion, die Sequenz ab dieser Stelle vervielfältigen. Der folgende Sequenzierungsschritt beginnt

daher am Ende (mit einer kleinen Überlappung) des bereits sequenzierten Anfangsstücks.

Dieses Verfahren lässt sich natürlich beliebig wiederholen. Somit läuft man also mit Primern über die Sequenz und sequenziert die DNS Stück für Stück. In der Anwesenheit von Repeats (Wiederholungen) versagt diese Methode, da man innerhalb eines langen Repeats per Definition keinen eindeutigen Primer mehr finden kann.

0.6.6.2 Nested Sequencing

Auch beim *Nested Sequencing* läuft man Stück für Stück über die Sequenz. Immer wenn man eine Sequenz der Länge 500 sequenziert hat, kann man diese mit Hilfe eines Enzyms (Exonuclease) entfernen und mit der restlichen Sequenz weitermachen.

Beide vorgestellten Verfahren, die die Sequenz Stück für Stück sequenzieren, haben den Nachteil, dass sie inhärent sequentiell und somit bei großen Genomen sehr langsam sind.

0.6.6.3 Sequencing by Hybridization

Beim *Sequenzieren durch Hybridisierung* oder kurz *SBH* (engl. sequencing by hybridization) werden die Sequenzen zuerst vervielfältigt und dann durch Restriktionsenzyme kleingeschnitten. Diese klein geschnittenen Sequenzen werden durch Hybridisierung mit allen Sequenzen der Längen bis etwa 8 verglichen. Somit ist bekannt, welche kurzen Sequenzen in der zu sequenzierenden enthalten sind. Aus diesen kurzen Stücken lässt sich dann mit Informatik-Methoden die Gesamtsequenz wiederherstellen.

Dieses Verfahren ist jedoch sehr aufwendig und hat sich bislang nicht durchgesetzt. Jedoch hat es eine bemerkenswerte Technik, die so genannten *DNA-Microarrays* oder *Gene-Chips* hervorgebracht, die jetzt in ganz anderen Gebieten der Molekularbiologie eingesetzt werden.

Ein DNA-Microarray ist eine kleine Glasplatte (etwa 1cm^2), die in etwa 100 mal 100 Zellen aufgeteilt ist. In jeder Zelle wird ein kleines Oligonukleotid der Länge von bis zu 50 Basen aufgebracht. Mit Hilfe der Hybridisierung können nun parallel 10.000 verschiedene Hybridisierungsexperimente gleichzeitig durchgeführt werden. Die zu untersuchenden Sequenzen sind dabei wieder fluoreszent markiert und können nach dem hochparallelen Hybridisierungsexperiment entsprechend ausgewertet werden.

0.6.6.4 Shotgun Sequencing

Eine weitere Möglichkeit, ein ganzes Genome zu sequenzieren, ist das so genannte *Shotgun-Sequencing*. Hierbei werden lange Sequenzen in viele kurze Stücke aufgebrochen. Dabei werden die Sequenzen in mehrere Klassen aufgeteilt, so dass (in der Regel) eine Bruchstelle in einer Klasse mitten in den Fragmenten der anderen Sequenzen liegt.

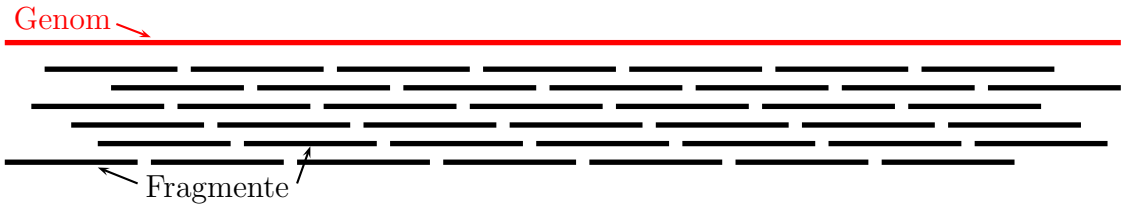


Abbildung 39: Skizze: Shotgun-Sequencing

Die kurzen Sequenzen können jetzt wieder direkt automatisch sequenziert werden. Es bleibt nur das Problem, aus der Kenntnis der Sequenzen wieder die lange DNS-Sequenz zu rekonstruieren. Dabei hilft, dass einzelne Positionen (oder sogar kurze DNS-Stücke) von mehreren verschiedenen Fragmenten, die an unterschiedlichen Positionen beginnen, überdeckt werden. In der Regel sind diese Überdeckungen relativ lang. Somit muss man nur noch die Fragmente wie in einem Puzzle-Spiel so anordnen, dass überlappende Bereiche möglichst gleich sind (man muss ja leider immer noch mit Sequenzierfehlern leben).

Zunächst dachte man, dass diese Methode nur für kürzere DNS-Stränge möglich ist. etwa für 100 000 Basenpaare. Celera Genomics zeigte jedoch mit der Sequenzierung des ganzen Genoms der Fruchtfliege (Drosophila melanogaster) und schließlich dem menschlichen Genom, dass diese (bzw. eine geeignet modifizierte) Methode auch für lange DNS-Sequenzen zum Ziel führt.

1 Algorithmik

1.1 Einführendes Beispiel: MSS

In diesem Kapitel befassen wir uns mit einer kurzen Einführung in die Algorithmik. Wir wollen hierzu die Grundlagen zum Entwurf und zur Analyse von Algorithmen besprechen.

Ziel dieses Abschnitts ist es, anhand eines einfachen Problems die verschiedenen Entwurfs- und Analysetechniken für Algorithmen exemplarisch vorzustellen. In den folgenden Abschnitten werden wir dann im einzelnen die benötigten Grundlagen und Techniken genauer behandeln.

1.1.1 Maximal Scoring Subsequence

Zunächst einmal wollen wir das hier betrachtete Problem definieren.

MAXIMAL SCORING SUBSEQUENCE (MSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$, die $\sigma(i,j)$ maximiert, wobei $\sigma(i,j) := \sum_{\ell=i}^{j} a_\ell$.

Bemerkung: Mit Teilfolgen sind in diesem Kapitel immer (sofern nicht anders erwähnt) zusammenhängende (d.h. konsekutive) Teilfolgen einer Folge gemeint (also anders als beispielsweise in der Analysis).

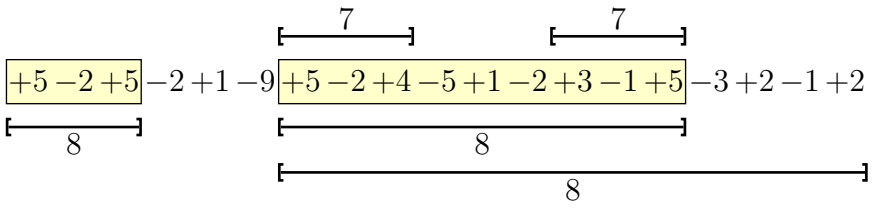


Abbildung 1.1: Beispiel: Maximal Scoring Subsequences

In Abbildung 1.1 ist ein Beispiel angegeben. Wie man dort sieht kann es mehrere (und auch nicht-disjunkte) Lösungen geben.

Bemerkungen:

- Wie man an dem Beispiel sieht, sind mehrere Lösungen möglich.
- Die verschiedenen Lösungen können sich ohne weitere Einschränkung auch überlappen. Ist eine Lösung in der anderen enthalten, so lassen wir im Folgenden als Lösung immer eine Lösung kürzester Länge als echte Lösung zu. Die anderen ergeben sich aus Anhängen von Teilfolgen mit dem Score Null. Darüber hinaus haben solche Lösungen noch einen schöne Eigenschaft, wie wir gleich sehen werden. Dann haben alle Präfixe bzw. Suffixe einer Lösungsfolge einen echt positiven Score.
- Mit der obigen Einschränkungen auf kurze Lösungen kann es keine echt überlappenden Lösungen mehr geben. Angenommen, es gäbe echt überlappende Lösungen a' und a'' einer gegebenen Folge a (siehe dazu auch Abbildung 1.2). Die Sequenz gebildet aus der Vereinigung beider Sequenzen (respektive ihrer Indizes) müsste dann einen höheren Score haben, da der Score der Endstücke > 0 ist (sonst würde er in den betrachteten Teilfolgen nicht berücksichtigt werden).

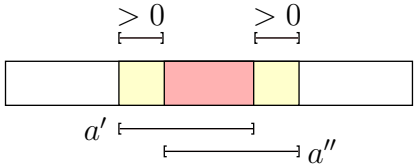


Abbildung 1.2: Skizze: Überlappende optimale Teilfolgen

- Die gleiche Eigenschaft erhält man, wenn man nur längste Lösungen zulässt.

Nur als kurze Motivation wollen wir hier noch anmerken, dass das Problem kein künstliches ist, sondern sogar in der Bioinformatik verschiedene Anwendung besitzt. Beispielsweise kann man diesen Algorithmus zur die Bestimmung der *transmembranen Regionen* eines *Transmembranproteins* verwenden. In die Membran eingelagerte Proteine sollten einen ähnlichen Aufbau wie die Membran selbst haben, damit die Gesamtstruktur stabiler ist. Somit sollten transmembrane Regionen hydrophob sein.

Mit einer geeigneten Gewichtung der verschiedenen Aminosäuren können solche hydrophoben Regionen mit Hilfe der Lösung eines Maximal Scoring Subsequence Problems gefunden werden. Für die einzelnen Aminosäuren werden die folgende Werte gemäß der Hydrophobizität der entsprechenden Aminosäure gewählt: Für hydrophobe Aminosäuren wählt man einen positiven Wert aus $[0, 3]$; für hydrophile Aminosäuren einen negativen Wert aus $[-5, 0]$.

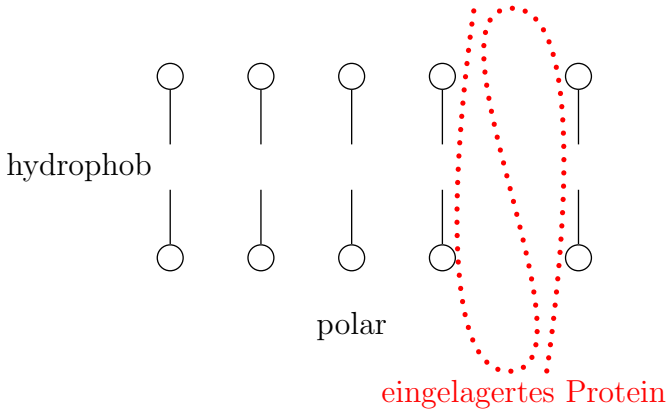


Abbildung 1.3: Beispiel: transmembrane Proteine

Es gibt noch viele weitere Anwendungen, für die wir den interessierten Leser auf das Skript zur *Algorithmen auf Sequenzen* verweisen.

Halten wir noch kurz die eben verwendete Notation formal fest.

Notation 1.1 *Für $a, b \in \mathbb{R}$ ist $[a : b] = [a, b] \cap \mathbb{Z} = \{z \in \mathbb{Z} : a \leq z \leq b\}$.*

Man beachte, dass wir dabei in der Regel $a, b \in \mathbb{Z}$ verwenden.

1.1.2 Naive Lösung

Im Folgenden wollen wir eine Reihe von Algorithmen zur Lösung des Maximal Scoring Subsequence Problems vorstellen, die jeweils effizienter als die vorher vorgestellte Variante ist.

Beginnen wollen wir mit einem naiven Ansatz. Wir benötigen jedoch vorher noch eine Notation.

Notation 1.2 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen, dann bezeichnet $\sigma(i, j) := \sum_{\ell=i}^{j} a_\ell$ für $i \leq j \in [1 : n]$.*

Die naive Methode bestimmt zuerst alle Werte $\sigma(i, j)$ für alle $i \leq j \in [1 : n]$. Eine simple Implementierung ist in Abbildung 1.4 angegeben. Wir merken an, dass wir den Pseudo-Code für die MSS-Algorithmen im Folgenden der Einfachheit halber für ganzzahlige Folgen angeben.

Die Korrektheit des Algorithmus folgt unmittelbar aus der Konstruktion, da wir ja alle Möglichkeiten der Reihe nach ausprobieren. Diese algorithmische Idee nennt

```
MSS_Naive (int[] a, int n)
begin
    maxscore := 0;   ℓ := 1;   r := 0;        /* always is maxscore = σ(ℓ, r) */
    for (i := 1; i ≤ n; i++) do
        for (j := i; j ≤ n; j++) do
            s := 0;                                  /* compute s = σ(i, j) */
            for (k := i; k ≤ j; k++) do
                s := s + a[k] ;
            if (s > maxscore) then
                maxscore := s;   ℓ := i;   r := j;
end
```

Abbildung 1.4: Algorithmus: naiver Algorithmus für MSS

man auch *vollständige Suche* oder *vollständige Aufzählung* (im Englischen *complete enumeration* oder *exhaustive search*).

Als nächstes stellt sich die Frage, wie gut dieser Algorithmus ist. Was heißt hier überhaupt gut? Zum einem will man natürlich sicher sein, dass der Algorithmus wirklich eine korrekte Lösung liefert. Das heißt, man muss zuerst die *Korrektheit* eines Algorithmus überprüfen. In diesem Fall ist das offensichtlich der Fall.

Zum anderen möchte man natürlich einen möglichst schnellen Algorithmus für das gegebene Problem. Was heißt überhaupt schnell bzw. schneller als was. Dazu muss man also die Laufzeiten von Algorithmen für dasselbe Problem vergleichen. Die einfachste Möglichkeit ist, die gemessenen Laufzeiten für die Implementationen der in Frage kommenden Algorithmen zu vergleichen. Da es sehr viele potenzielle Eingaben gibt, ist das in der Praxis nicht durchführbar, höchstens für eine paar exemplarische Eingaben.

Daher versucht man, die Laufzeit theoretisch abzuschätzen. Eine erste Möglichkeit ist, die Anzahl der Taktzyklen des erzeugten Maschinencodes zu bestimmen. Damit ist man zum einen von der verwendeten Architektur abhängig, zum anderen ist einen solche Bestimmung nicht einfach, da sie in der Regel auch von der Eingabe abhängt.

Ein einfacheres qualitatives Maß ist die Anzahl ausgeführter Befehle des Pseudo-Codes. Auch dies ist in der Regel sehr aufwendig. Daher ermittelt man meist nur die Anzahl von speziellen ausgeführten Anweisungen, die für den Ablauf des Programms typisch sind. Typsich heißt hier, dass sie die Anzahl aller anderen Operationen majorisieren. Auf jede typische Operation kommt maximal eine konstante Anzahl anderer ausgeführter Operationen.

Dies liefert natürlich keine genaue Vorhersage für die Laufzeit in Sekunden o.ä., aber für einen ersten qualitativen Vergleich von Algorithmen ist das völlig ausreichend, wie wir in diesem Abschnitt noch sehen werden.

Für unser Problem wählen wir als typische Operation die Addition eines Array-Elements. Mit solchen Array-Additionen sind hier und im Folgenden Additionen eines Array-Elements bzw. Additionen einer Variablen, die eine Summe von Array-Elementen enthält, mit einer anderen Zahl (die eventuell ebenfalls ein Array-Element bzw. eine Variable, die eine Summe von Array-Elementen enthält) gemeint.

Damit ergibt sich für unseren Algorithmus der folgende Aufwand $A_{\text{Naiv}}(n)$ für eine Folge mit n Elementen:

$$
\begin{aligned}
A_{\text{Naiv}}(n) &= \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 \\
&= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \\
&\quad \text{Indexverschiebung in der inneren Summe} \\
&\quad \left(\textstyle\sum_{i=a}^{b} f(i+c) = \sum_{i=a+c}^{b+c} f(i)\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j \\
&\quad \text{Gaußsche Summe} \\
&= \sum_{i=1}^{n}\binom{(n-i+1)+1}{2} \\
&\quad \text{Rückwärtige Summation} \\
&\quad \left(\textstyle\sum_{i=a}^{b} f(c-i) = \sum_{i=c-b}^{c-a} f(i)\right) \\
&= \sum_{i=1}^{n}\binom{i+1}{2} \\
&\quad \text{Indexverschiebung} \\
&= \sum_{i=2}^{n+1}\binom{i}{2} \\
&\quad \text{mit } \sum_{i=k}^{n}\binom{i}{k} = \binom{n+1}{k+1} \text{ für } k=2 \\
&= \binom{n+2}{3}
\end{aligned}
$$

$$= \frac{(n+2)(n+1)n}{3 \cdot 2 \cdot 1}$$
$$= \frac{n^3 + 3n^2 + 2n}{6}.$$

Lemma 1.3 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Der naive Algorithmus benötigt zur Bestimmung einer maximal scoring subsequence genau $\frac{n^3+3n^2+2n}{6}$ Array-Additionen.*

1.1.3 Rekursion

Ein anderes aus Informatik I bekanntes Paradigma zum Entwurf von Algorithmen ist die *Rekursion*. Aufgrund der Assoziativität der Addition gilt folgende Rekursionsgleichung:

$$\sigma(i,j) = \begin{cases} 0 & \text{für } i > j, \\ a_i & \text{für } i = j, \\ \sigma(i,k) + \sigma(k+1,j) & \text{für } i < j \text{ und ein } k \in [i : j-1]. \end{cases}$$

Basierend auf dieser Rekursionsgleichung ergibt sich der in Abbildung 1.5 angegebene Algorithmus.

24.04.19

MSS_rec (int[] a, int n)

```
begin
    maxscore := 0;    ℓ := 1;    r := 0;
    for (i := 1; i ≤ n; i++) do
        for (j := i; j ≤ n; j++) do
            s := σ(i, j);                          /* compute s = σ(i, j) */
            if (s > maxscore) then
                maxscore := s;    ℓ := i;    r := j;
end
```

int σ(int i, j);

```
begin
    if (i > j) then return 0;
    else if (i = j) then return a[i];
    else return σ(i, k) + σ(k + 1, j);          /* for some k ∈ [i : j − 1] */
end
```

Abbildung 1.5: Algorithmus: rekursive Berechnung des Scores für MSS

Man überlegt sich leicht, dass für die Berechnung Summe $\sigma(i,j)$ mittels Rekursion genau $j-i$ Additionen ausreichend sind. In der Rekursion wird das ursprüngliche Feld der Länge m jeweils in zwei Teile aufgeteilt und die Ergebnisse addiert. Die Aufteilung endet, wenn das Feld einelementig ist. Eine wiederholte Aufteilung eines Feldes in lauter einelementige Felder benötigt genau $m-1$ Aufteilungen. Also werden in der Rekursion auch genau $m-1$ Additionen ausgeführt. Damit ergibt sich für diesen Algorithmus der folgende Aufwand $A_{\text{rec}}(n)$ für eine Folge mit n Elementen:

$$\begin{aligned}
A_{\text{rec}}(n) &= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) \\
&= \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) - \sum_{i=1}^{n}\sum_{j=i}^{n}1 \\
&= A_{\text{Naiv}}(n) - \sum_{i=1}^{n}(n-i+1) \\
&\quad \text{Rückwärtige Summation} \\
&= A_{\text{Naiv}}(n) - \sum_{i=1}^{n} i \\
&\quad \text{Einsetzen von } A_{\text{Naiv}}(n) \text{ und Gaußsche Summe} \\
&= \frac{n^3+3n^2+2n}{6} - \frac{n(n+1)}{2} \\
&= \frac{n^3+3n^2+2n}{6} - \frac{3n^2+3n}{6} \\
&= \frac{n^3-n}{6}.
\end{aligned}$$

Lemma 1.4 *Sei $(a_1,\ldots,a_n) \in \mathbb{R}^n$ eine reelle Folge. Der auf Rekursion basierte Algorithmus benötigt zur Bestimmung einer maximal scoring subsequence genau $\frac{n^3-n}{6}$ Array-Additionen.*

1.1.4 Dynamische Programmierung

Die im vorherigen Unterabschnitt gefundene Rekursionsgleichung können wir auch noch anders verwenden:

$$\sigma(i,j) = \begin{cases} 0 & \text{für } i > j, \\ a_i & \text{für } i = j, \\ \sigma(i,k) + \sigma(k+1,j) & \text{für } i < j \text{ und ein } k \in [i:j-1]. \end{cases}$$

Man überlegt sich leicht, dass bestimmte Summen mehrfach berechnet werden. Mit Hilfe der so genannten *dynamischen Programmierung* können wir jedoch effizienter werden. Wir speichern alle berechneten Werte in einer Tabelle und schauen dort die Werte nach. Dabei wird die Tabelle diagonal von der Mitte nach rechts oben aufgefüllt (siehe auch Abbildung 1.6).

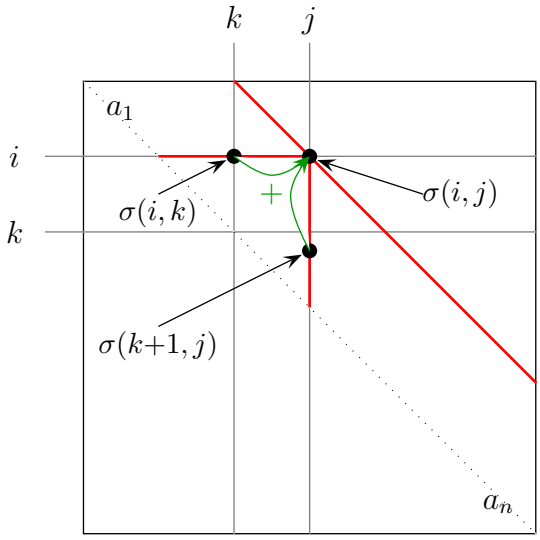


Abbildung 1.6: Skizze: Auffüllen der dynamischen Programmierungstabelle

Wählen wir $k = j - 1$, so können wir die Tabelle auch zeilenweise auffüllen, wie im in Abbildung 1.7 angegeben Algorithmus. Dabei kann das Feld $S[i, j]$ auch durch eine Variable s ersetzt werden, da immer nur auf das vorhergehende Feldelement in der Zeile zugegriffen wird.

MSS_DP (int[] a, int n)

```
begin
    maxscore := 0;    ℓ := 1;    r := 0;
    for (i := 1; i ≤ n; i++) do
        for (j := i; j ≤ n; j++) do
            if (i = j) then S[i, i] := a[i];
            else S[i, j] := S[i, j − 1] + a[j];
            if (S[i, j] > maxscore) then
                maxscore := S[i, j];    ℓ := i;    r := j;
end
```

Abbildung 1.7: Algorithmus: Dynamische Programmierung für MSS

Wie ist nun der Aufwand $A_{\mathrm{DP}}(n)$ für die Dynamische Programmierung:

$$\begin{aligned} A_{\mathrm{DP}}(n) &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 \\ &= \sum_{i=1}^{n} (n-i) \\ &\quad \text{Rückwärtige Summation} \\ &= \sum_{i=0}^{n-1} i \\ &= \binom{n}{2} \\ &= \frac{n^2-n}{2}. \end{aligned}$$

Lemma 1.5 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Der auf dynamischer Programmierung basierende Algorithmus benötigt zur Bestimmung einer maximal scoring subsequence genau $\frac{n^2-n}{2}$ Array-Additionen.*

1.1.5 Divide-and-Conquer-Ansatz

Eine andere Lösungsmöglichkeit erhalten wir mit einem *Divide-and-Conquer-Ansatz*, wie in Abbildung 1.8 illustriert. Dabei wird die Folge in zwei etwa gleich lange Folgen aufgeteilt und die Lösung in diesen rekursiv ermittelt.

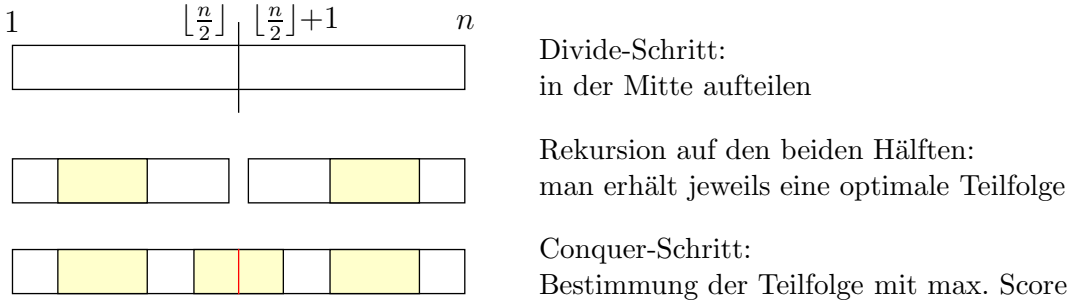
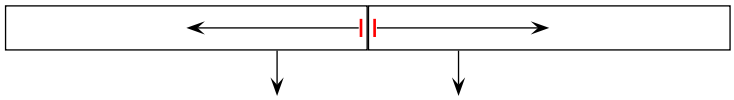


Abbildung 1.8: Skizze: Divide-and-Conquer bei Maximal Scoring Subsequences

Man kann dabei aber auch die optimale Teilfolge in der Mitte zerschneiden. Daher muss man zusätzlich von der Mitte aus testen, wie von dort nach rechts bzw. links eine optimale Teilfolge aussieht (siehe auch Abbildung 1.9).



$\max\left\{\sigma(i, \lfloor \frac{n}{2} \rfloor) \,:\, i \in [1 : \lfloor \frac{n}{2} \rfloor]\right\}$ $\max\left\{\sigma(\lfloor \frac{n}{2} \rfloor + 1, j) \,:\, j \in [\lfloor \frac{n}{2} \rfloor + 1 : n]\right\}$

Abbildung 1.9: Skizze: Conquer-Step

Wir halten hier noch schnell die üblichen Definitionen von unteren bzw. oberen Gauß-Klammern und ein paar elementare Eigenschaften fest.

Notation 1.6 *Für $x \in \mathbb{R}$ ist*

- $\lfloor x \rfloor = \max\{z \in \mathbb{Z} \,:\, z \leq x\}$ *(*untere Gauß-Klammer*)*
- $\lceil x \rceil = \min\{z \in \mathbb{Z} \,:\, z \geq x\}$ *(*obere Gauß-Klammer*)*

Beobachtung 1.7 *Sei $z \in \mathbb{Z}$, dann gilt $\lfloor z/2 \rfloor + \lceil z/2 \rceil = z$.*

Beobachtung 1.8 *Sei $x \in \mathbb{R}$, dann gilt $\lfloor x \rfloor = -\lceil -x \rceil$.*

Um die optimale Teilfolge zu bestimmen, die die Positionen $\lfloor n/2 \rfloor$ und $\lfloor n/2 \rfloor + 1$ enthält, bestimmen wir zunächst je eine optimale Teilfolge in der linken bzw. rechten Hälfte, die die Position $\lfloor n/2 \rfloor$ bzw. $\lfloor n/2 \rfloor + 1$ enthält. Dazu bestimmen wir jeweils das Optimum der Hälften, d.h

$$\begin{aligned} i' &:= \operatorname{argmax}\{\sigma(i, \lfloor n/2 \rfloor) \,:\, i \in [1 : \lfloor n/2 \rfloor]\}, \\ j' &:= \operatorname{argmax}\{\sigma(\lfloor n/2 \rfloor + 1, j) \,:\, j \in [\lfloor n/2 \rfloor + 1 : n]\}. \end{aligned}$$

Notation 1.9 *Sei $f : M \to N$ eine Abbildung von einer Menge M in eine total geordnete Menge N. Dann ist* $\operatorname{argmax}\{f(i) \,:\, i \in M\}$ *(und analog* argmin*) definiert als ein Indexwert $j \in M$ mit $f(j) = \max\{f(i) \,:\, i \in M\}$.*

Wir zeigen jetzt, dass die optimale Teilfolge, die die Positionen $\lfloor n/2 \rfloor$ und $\lfloor n/2 \rfloor + 1$ überdeckt, aus der Konkatenation der beiden berechneten optimalen Teilfolgen in den jeweiligen Hälften bestehen muss.

Lemma 1.10 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Sei weiter*

$$\begin{aligned} i' &:= \operatorname{argmax}\{\sigma(i, \lfloor n/2 \rfloor) \,:\, i \in [1 : \lfloor n/2 \rfloor]\}, \\ j' &:= \operatorname{argmax}\{\sigma(\lfloor n/2 \rfloor + 1, j) \,:\, j \in [\lfloor n/2 \rfloor + 1 : n]\}. \end{aligned}$$

Dann gibt es keine Teilfolge von a mit einem größeren Score als $\sigma(i', j')$, die die Positionen $\lfloor n/2 \rfloor$ und $\lfloor n/2 \rfloor + 1$ enthält.

Beweis: Wir führen den Beweis durch Widerspruch. Sei also $i^* \in [1 : \lfloor n/2 \rfloor]$ und $j^* \in [\lfloor n/2 \rfloor + 1 : n]$ mit $\sigma(i^*, j^*) > \sigma(i', j')$.

Nach Definition von i' und j' gilt:

$$\sigma(i', \lfloor n/2 \rfloor) \geq \sigma(i^*, \lfloor n/2 \rfloor) \quad \text{und} \quad \sigma(\lfloor n/2 \rfloor + 1, j') \geq \sigma(\lfloor n/2 \rfloor + 1, j^*).$$

Also gilt

$$\begin{aligned} \sigma(i', j') &= \sigma(i', \lfloor n/2 \rfloor) + \sigma(\lfloor n/2 \rfloor + 1, j') \\ &\geq \sigma(i^*, \lfloor n/2 \rfloor) + \sigma(\lfloor n/2 \rfloor + 1, j^*) \\ &= \sigma(i^*, j^*) \\ &> \sigma(i', j') \end{aligned}$$

Dies ist offensichtlich ein Widerspruch. ∎

Damit ergibt sich folgender Divide-and Conquer-Algorithmus, der in Abbildung 1.10 angegeben ist.

Wie sieht nun die Laufzeit $A_{\mathrm{DC}}(n)$ für diesen Divide-and-Conquer-Algorithmus aus? Zunächst stellen wir fest, dass für eine Folge der Länge 1 keine Additionen auf Array-Elementen ausgeführt werden, also gilt $A_{\mathrm{DC}}(1) = 0$.

Für die linke Hälfte der Folge bestimmen wir rekursiv eine optimale Teilfolge. Hierfür werden nach Definition $A_{\mathrm{DC}}(\lfloor n/2 \rfloor)$ Additionen von Array-Elementen benötigt. Analog werden für die rechte Hälfte $A_{\mathrm{DC}}(\lceil n/2 \rceil)$ Additionen von Array-Elementen benötigt. Für die Bestimmung der optimalen Teilfolge, die die Positionen $\lfloor n/2 \rfloor$ und $\lfloor n/2 \rfloor + 1$ enthält, sind $(\lfloor n/2 \rfloor - 1) + (\lceil n/2 \rceil - 1) + 1 = n - 1$ Additionen von Array-Elementen nötig. Die letzte plus 1 resultiert aus der Summe der beiden Maxima der linken bzw. rechten Hälfte, die das Element an Position $\lfloor n/2 \rfloor$ bzw. $\lfloor n/2 \rfloor + 1$ beinhalten. Somit erhalten wir für die Laufzeit:

$$A_{\mathrm{DC}}(n) = \begin{cases} 0 & \text{falls} \quad n = 1, \\ A_{\mathrm{DC}}(\lfloor n/2 \rfloor) + A_{\mathrm{DC}}(\lceil n/2 \rceil) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

```
MSS (int[] a, int n)
begin
    (maxscore, ℓ, r) := MSS_DC(a, 1, n);
end

(int,int,int) MSS_DC(int[] a, int i, j);
begin
    if (i = j) then
        if (a[i] > 0) then return (a[i], i, i);
        else return (0, i, i − 1);
    else
        m := ⌊(i+j−1)/2⌋;
        (s1, i1, j1) := MSS_DC(a, i, m);
        (s2, i2, j2) := MSS_DC(a, m + 1, j);

        i3 := m;
        s := a[i3];
        simax := s;
        for (k := i3 − 1; k ≥ i; k−−) do
            s := s + a[k];
            if (s > simax) then
                simax := s;   i3 := k;

        j3 := m + 1;
        s := a[j3];
        sjmax := s;
        for (k := j3 + 1; k ≤ j; k++) do
            s := s + a[k];
            if (s > sjmax) then
                sjmax := s;   j3 := k;

        s3 := simax + sjmax;                          /* s3 = σ(i3, j3) */
        if (max{s1, s2, s3} = s1) then return (s1, i1, j1);
        else if (max{s1, s2, s3} = s2) then return (s2, i2, j2);
        else return (s3, i3, j3);
end
```

Abbildung 1.10: Algorithmus: Divide-and-Conquer-Algorithmus für MSS

Leider lässt sich diese Laufzeit nun schlecht mit den anderen Laufzeiten vergleichen. Wir benötigen also noch eine geschlossene Form dieser *Rekursionsgleichung*. Da beim Rechnen die Gauß-Klammern einige Probleme bereiten, nehmen wir an, dass

n eine Zweierpotenz ist, also $n = 2^k$. Damit hat die Folge und alle resultierenden Teilfolgen eine gerade Länge, mit Ausnahmen der Folgen der Länge 1. Damit können wir leichter rechnen und das Ergebnis für Längen der Folgen 1 kennen wir ja. Damit ergibt sich

$$A_{\mathrm{DC}}(2^k) = \begin{cases} 0 & \text{falls} \quad k = 0 \quad (n = 1), \\ 2A_{\mathrm{DC}}(2^{k-1}) + (2^k - 1) & \text{falls} \quad k > 0 \quad (n > 1). \end{cases}$$

Wir können nun die Rekursionsgleichung auf der rechten Seite erneut einsetzen und erhalten:

$$\begin{aligned} A_{\mathrm{DC}}(2^k) &= 2A_{\mathrm{DC}}(2^{k-1}) + (2^k - 1) \\ &= 2\left(2A_{\mathrm{DC}}(2^{k-2}) + (2^{k-1} - 1)\right) + (2^k - 1) \\ &= 2^2 A_{\mathrm{DC}}(2^{k-2}) + 2(2^{k-1} - 1) + (2^k - 1) \end{aligned}$$

Da wir nicht viel schlauer geworden sind, setzen wir die Rekursionsgleichung noch einmal ein:

$$\begin{aligned} &= 2^2\left(2A_{\mathrm{DC}}(2^{k-3}) + (2^{k-2} - 1)\right) + 2(2^{k-1} - 1) + (2^k - 1) \\ &= 2^3 A_{\mathrm{DC}}(2^{k-3}) + 2^2(2^{k-2} - 1) + 2(2^{k-1} - 1) + (2^k - 1) \end{aligned}$$

Die Summanden am Ende schreiben wir als formale Summe:

$$= 2^3 A_{\mathrm{DC}}(2^{k-3}) + \sum_{j=0}^{2} 2^j(2^{k-j} - 1).$$

Nun können wir eine Vermutung anstellen, wie diese Formel nach i-fachen Einsetzen der Rekursion aussieht:

$$A_{\mathrm{DC}}(2^k) = 2^i A_{\mathrm{DC}}(2^{k-i}) + \sum_{j=0}^{i-1} 2^j(2^{k-j} - 1).$$

Wie können wir nun überprüfen, ob wir richtig geraten haben? Hierfür eignet sich am besten ein Beweis mittels vollständiger Induktion.

Beobachtung 1.11 *Es gilt für $i \in [1 : k]$:*

$$A_{\mathrm{DC}}(2^k) = 2^i A_{\mathrm{DC}}(2^{k-i}) + \sum_{j=0}^{i-1} 2^j(2^{k-j} - 1).$$

Beweis: Induktionsanfang ($i = 1$): Offensichtlich ist

$$A_{\mathrm{DC}}(2^k) = 2A_{\mathrm{DC}}(2^{k-1}) + (2^k - 1) = 2^1 A_{\mathrm{DC}}(2^{k-1}) + \sum_{j=0}^{1-1} 2^j (2^{k-j} - 1).$$

Induktionsschritt ($i \to i+1$): Es gilt nach Induktionsvoraussetzung:

$$\begin{aligned}
A_{\mathrm{DC}}(2^k) &= 2^i A_{\mathrm{DC}}(2^{k-i}) + \sum_{j=0}^{i-1} 2^j (2^{k-j} - 1) \\
&= 2^i \left(2A_{\mathrm{DC}}(2^{k-i-1}) + (2^{k-i} - 1)\right) + \sum_{j=0}^{i-1} 2^j (2^{k-j} - 1) \\
&= 2^{i+1} A_{\mathrm{DC}}(2^{k-(i+1)}) + 2^i (2^{k-i} - 1) + \sum_{j=0}^{i-1} 2^j (2^{k-j} - 1) \\
&= 2^{i+1} A_{\mathrm{DC}}(2^{k-(i+1)}) + \sum_{j=0}^{(i+1)-1} 2^j (2^{k-j} - 1)
\end{aligned}$$

Also gilt die Formel auch für $i + 1$ und der Induktionsschluss ist vollzogen. ■

In dieser Formel können wir nun $i = k$ setzen und erhalten:

$$\begin{aligned}
A_{\mathrm{DC}}(2^k) &= 2^i A_{\mathrm{DC}}(2^{k-i}) + \sum_{j=0}^{i-1} 2^j (2^{k-j} - 1) \\
&= 2^k A_{\mathrm{DC}}(2^{k-k}) + \sum_{j=0}^{k-1} 2^j (2^{k-j} - 1) \\
&= 2^k A_{\mathrm{DC}}(1) + \sum_{j=0}^{k-1} 2^j (2^{k-j} - 1) \\
&= 2^k \cdot 0 + \sum_{j=0}^{k-1} 2^j (2^{k-j} - 1) \\
&= \sum_{j=0}^{k-1} 2^j (2^{k-j} - 1)
\end{aligned}$$

Diese Summe können wir nun leicht lösen:

$$= \sum_{j=0}^{k-1} 2^j \cdot 2^{k-j} - \sum_{j=0}^{k-1} 2^j$$

$$= \sum_{j=0}^{k-1} 2^k - \sum_{j=0}^{k-1} 2^j$$

Mit der geometrischen Reihe $\sum_{j=0}^{k} x^j = \frac{x^{k+1}-1}{x-1}$ für $x \neq 1$:

$$= k2^k - \frac{2^k - 1}{2 - 1}$$
$$= k2^k - 2^k + 1.$$

Mit $2^k = n$ und damit $k = \log(n)$ erhalten wir

$$A_{\mathrm{DC}}(n) = n\log(n) - n + 1.$$

Diese Lösung gilt natürlich nur solche Werte von n, die Zweierpotenzen sind. Dies ist aber kein Problem, da wir jede Folge am Ende mit 0en auffüllen können und für den Algorithmus nur Folgenlängen betrachten können, die Zweierpotenzen sind. Man überlegt sich leicht, dass dadurch die Menge der Lösungen (mit unserer Vereinbarung) nicht verändert wird.

Dies ist natürlich nicht sehr effizient. Wir werden später noch sehen, dass die Einschränkung auf Zweierpotenzen keine große Einschränkung ist, und diese Lösung im Wesentlichen für alle $n \in \mathbb{N}$ korrekt ist.

Lemma 1.12 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Der Divide-and-Conquer-Algorithmus benötigt zur Bestimmung einer maximal scoring subsequence genau $n\log(n) - n + 1$ Array-Additionen, sofern n eine Zweierpotenz ist.*

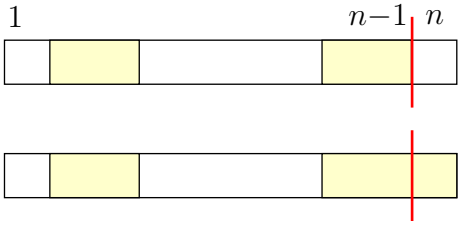
Wir merken hier noch an, dass mit log immer der Logarithmus zur Basis 2 gemeint ist. Andernfalls werden wir dies immer explizit angeben.

30.04.19

1.1.6 Clevere Lösung

Wenn wir wie beim Divide-and-Conquer-Ansatz das Problem nicht in der Mitte aufteilen, sondern am rechten Rand, so können wir eine effizientere Lösung finden (siehe auch Abbildung 1.11).

Wir können dies auch iterativ statt rekursiv interpretieren. Dazu nehmen wir an, dass wir für die Folge $(a_1, \ldots, a_{n-1})$ sowohl eine optimale Teilfolgen kennen, als auch eine optimale Teilfolge, die das Element an Position $n-1$ enthält. Wenn wir zu der Folgen $(a_1, \ldots, a_n)$ übergehen, müssen wir nur diese beide Folgen aktualisieren.



Nur das letzte Feld absplitten.

Am Rand gleich die optimale Teilfolge mitbestimmen, z.B. durch den Versuch die aktuelle Randfolge zu verlängern.

Abbildung 1.11: Skizze: Asymmetrische Divide-and-Conquer

Wir behaupten zuerst, dass für eine kürzeste optimale Teilfolge $(a_i, \ldots, a_n)$ von $(a_1, \ldots, a_n)$, die die Position n enthält und für die $i < n$ gilt, die kürzeste optimale Teilfolge von $(a_1, \ldots, a_{n-1})$, die die Position $n-1$ enthält, die Teilfolge $(a_i, \ldots, a_{n-1})$ ist. Würde $\sigma(j, n-1) > \sigma(i, n-1)$ sein, dann wäre auch

$$\sigma(j, n) = \sigma(j, n-1) + a_n > \sigma(i, n-1) + a_n = \sigma(i, n),$$

was offensichtlich nicht sein kann.

Somit brauchen wir beim Übergang von $n-1$ zu n für die optimale Folge am rechten Rand nur überprüfen, ob sich die bisherige, um a_n erweiterte Lösung oder die Folge (a_n) optimal ist. Wir müssen also nur überprüfen, ob $rmaxscore + a_n > a_n$ ist (d.h. eigentlich nur, ob $rmaxscore > 0$), wobei $rmaxscore$ das bisherige Optimum am rechten Rand ist. Damit erhalten wir sofort den folgenden, in Abbildung 1.12 angegebenen Algorithmus.

Man kann diese Idee rekursiv als Divide-and-Conquer-Algorithmus implementieren oder iterativ wie in Abbildung 1.12 angegeben auflösen.

MSS_Clever (int[] a, int n)

```
begin
    int maxscore := 0;    ℓ := 1;    r := 0;
    int rmaxscore := 0;   rstart := 1;
    for (i := 1; i ≤ n; i++) do
        if (rmaxscore + a[i] > a[i]) then
            rmaxscore := rmaxscore + a[i];
        else
            rmaxscore := a[i];    rstart := i;
        if (rmaxscore > maxscore) then
            maxscore := rmaxscore;    ℓ := rstart;    r := i;
end
```

Abbildung 1.12: Algorithmus: Die clevere Lösung

Da im Wesentlichen einmal über die Folge gelaufen wird und für jedes Array-Element nur einmal addiert wird, erhalten wir offensichtlich eine Laufzeit von $A_{\text{Clever}}(n) = n$ (genau genommen $A_{\text{Clever}}(n) \leq n$).

Da dies, wie gesehen, eine optimale Lösung ist, halten wir das Ergebnis im folgenden Satz fest.

Theorem 1.13 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Der clevere Algorithmus benötigt zur Bestimmung einer maximal scoring subsequence maximal n Array-Additionen.*

1.1.7 Zusammenfassung

In der Tabelle in Abbildung 1.13 sind alle Resultate der vorgestellten Algorithmen noch einmal zusammengefasst.

Algorithmus	Laufzeit
Naiv	$\frac{n^3 + 3n^2 + 2n}{6}$
rek.	$\frac{n^3 - n}{6}$
DP	$\frac{n^2 - n}{2}$
D&C	$n \log(n) - n + 1$
Clever	n

Abbildung 1.13: Tabelle: Theoretische Laufzeiten für die MSS Algorithmen

In Abbildung 1.14 sind noch einmal die Laufzeiten der verschiedenen Algorithmen graphisch für kleine Folgenlänge gezeichnet. Hieraus werden die einzelnen Effizienzsteigerungen der verschiedenen Algorithmen schon bei sehr kleinen Folgenlängen ersichtlich. Beachte, dass in der oberen Abbildung die y-Achse logarithmisch skaliert ist.

In der Tabelle in Abbildung 1.15 sind die gemessenen Laufzeiten für die verschiedenen MSS Algorithmen. Vergleicht man die Laufzeit pro Elementar-Operation in der theoretischen Laufzeitabschätzung, so kommt man bei allen Algorithmen auf ein Ausführungszeit von etwa 2 Nanosekunden pro hier veranschlagte Elementaroperation (also Addition von Array-Elementen). Die Zeiten in Klammern sind der Einfachheit halber hochgerechnete und nicht real ermittelte Werte.

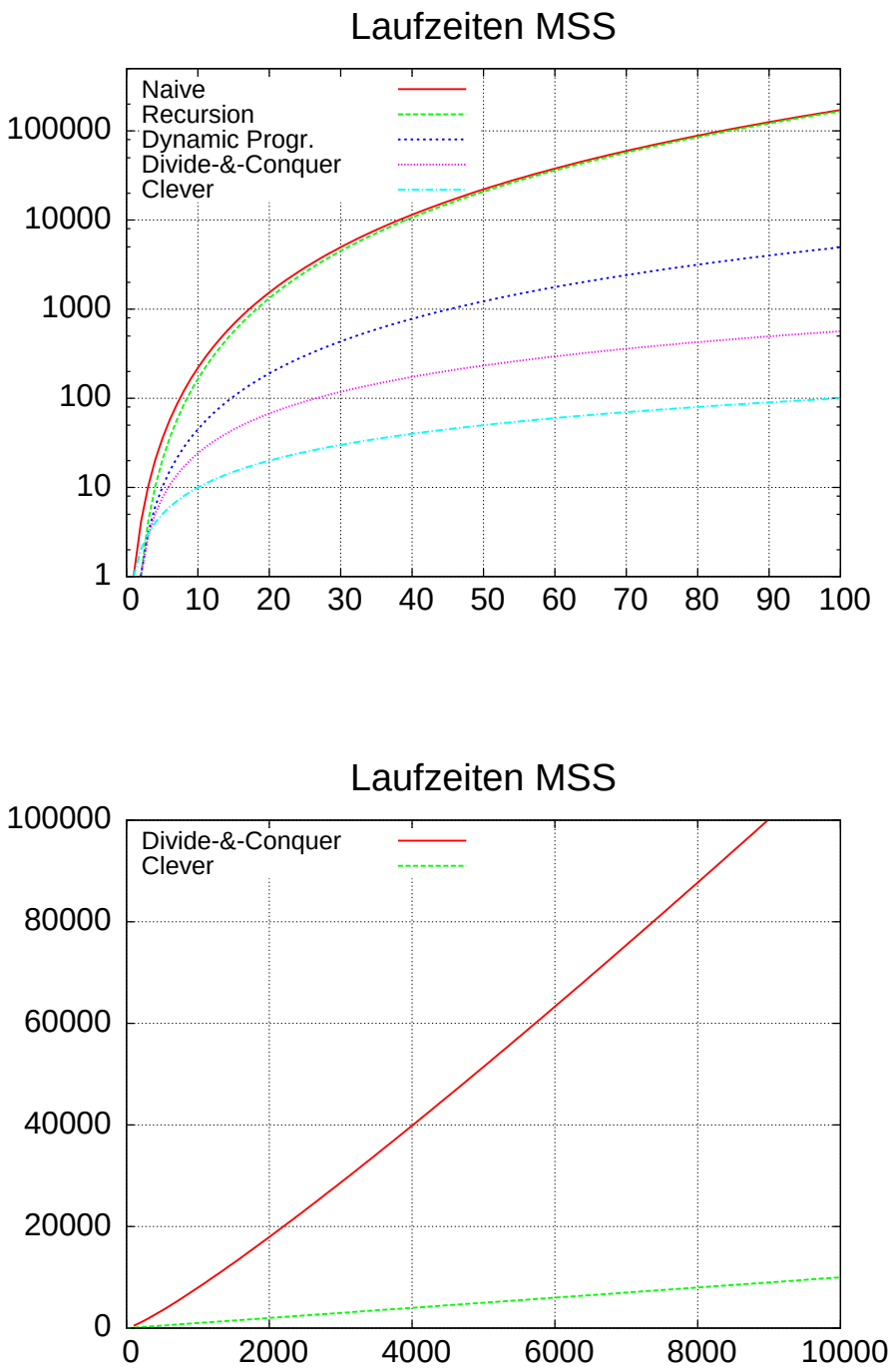


Abbildung 1.14: Skizze: Theoretische Laufzeiten im Vergleich

Algorithmus	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
Naiv	$0.30s$	$310s$	(86h)	—	—	—	—	—
DP	$0s$	$0.08s$	$7.73s$	$773s$	(21h)	—	—	—
D&C	$0s$	$0s$	$0s$	$0.05s$	$0.52s$	$5.68s$	$63.6s$	—
Clever	$0s$	$0s$	$0s$	$0s$	$0.02s$	$0.16s$	$1.55s$	$15.5s$

Abbildung 1.15: Tabelle: Praktische Laufzeiten für die MSS Algorithmen

In der Tabelle in der Abbildung 1.16 sind noch die Längen von Folgen angegeben, die in einer Sekunde bzw. einer Minute auf einem gewöhnlichen Rechner (Stand 2014, Intel Quad Core i7-3770, 3.40 GHz, 32 GB Hauptspeicher) verarbeitet werden können.

Seq-Len	1sec.		1min.
Naiv	1.500	$\xrightarrow{\times \approx 4}$	5.800
Dyn.Prog.	36.000	$\xrightarrow{\times \approx 8}$	280.000
D&C	18.000.000	$\xrightarrow{\times \approx 50}$	950.000.000
Clever	650.000.000	$\xrightarrow{\times \approx 60}$	$\approx$ 40.000.000.000

Abbildung 1.16: Tabelle: zu verarbeitenden Problemgrößen

Beachte hierbei, dass die Faktoren bei einer Versechzigfachung der Laufzeit beim naive Algorithmus etwa $\sqrt[3]{60} \approx 4$, bei der dynamischen Programmierung etwa $\sqrt{60} \approx 8$ und beim Cleveren Algorithmus etwa 60 ist. Die Funktion $n \log(n)$ besitzt keine einfache anzugebende Inverse, der Wert 50 entspricht aber in etwa der Inversen für 60 im Bereich der betrachteten Werte von n.

1.2 Komplexität von Algorithmen (*)

In diesem Abschnitt wollen wir etwas formaler die Komplexität von Algorithmen definieren.

1.2.1 Maschinenmodelle

Für eine saubere Analyse eines Algorithmus muss immer zuerst das zugrunde liegende Maschinenmodell festgelegt werden. Die einfachste Idee ist, einfachst die verwendete Rechner-Architektur als Maschinenmodell zugrunde zu legen und als Auf-

wand die Anzahl der benötigten Taktzyklen definieren. Damit wäre die Analyse sehr genau und man könnte eventuell sogar die Laufzeit daraus ableiten.

Nachteilig ist jedoch, dass eine solche Analyse sehr aufwendig ist, da man alle Belange der Architektur berücksichtigen müsste. Zum anderen wäre bei jedem Wechsel der Architektur, also bei jedem neuen Prozessor (oder sogar Architekturwechsel zu einen anderen Prozessorfamilie) eine Neubestimmung nötig. Dies ist viel zu aufwendig. Zum anderen verwenden intelligente Prozessorarchitekturen Methoden wie Caching, Instruction Prefetch und Branch Prediction, die einer exakten Analyse kaum oder nur sehr schwer zugänglich sind.

Daher betrachtet man für die Analyse meist abstrakte Maschinenmodelle wie Turing-Maschinen oder Registermaschinen, aber auch theoretischere Modelle wir den λ-Kalkül oder μ-rekursive Funktionen. Diese haben den Vorteil, dass diese zum einen konkret genug sind für realistische Ergebnisse und zum anderen noch einfach genug für eine exakte Analyse sind.

Für eine qualitative Analyse sind meist auch diese abstrakteren Maschinenmodelle noch zu komplex. Wie wir schon im einführenden Beispiel gesehen haben, genügt es, für eine qualitative Analyse in der Regel gewisse, für das Problem charakteristische Operationen zu zählen. Dabei muss man nur berücksichtigen, dass auf eine solche charakteristische Operation nur eine konstante Anzahl anderer Operationen kommt, die wir in der Analyse nicht berücksichtigen.

1.2.2 Worst-Case, Best-Case und Average-Case

Wenn das zugrunde gelegte Maschinenmodell bzw. die zu zählenden charakteristischen Operationen festgelegt sind, muss nur noch die Art der Analyse festgelegt werden.

Wir haben bisher in der Analyse immer den Aufwand für eine feste Eingabe angegeben. Mit $A(x)$ haben wir also die Anzahl charakteristischer Operationen für die Eingabe x gezählt. In unserem einführenden Beispiel also für eine gegebene Folge $x = (x_1, \ldots, x_n)$. Hier waren die eigentliche Werte $x_i \in \mathbb{R}$ für die Analyse uninteressant, da diese beim Zählen der charakteristischen Operationen keinen Einfluss hatten. Daher haben wir die Anzahl der Operationen in Abhängigkeit von der Länge der Folge n statt von der Folge x selbst angegeben.

Man kann sich leicht überlegen, dass die Anzahl der Operationen bei anderen Problemen durchaus von den Werten abhängen kann, wie beispielsweise beim Sortieren von n Zahlen. Da wir die Anzahl der Operationen nicht unbedingt in Abhängigkeit von der Eingabe, sondern von der Länge der Eingabe abhängig machen wollen, ergeben sich drei grundlegend unterschiedliche Definition der Laufzeit.

Definition 1.14 *Sei $A_M(x)$ die Anzahl von Operationen eines Algorithmus M im betrachteten Maschinenmodell für eine Eingabe x. Dann ist*

$$A_M^{\text{wc}}(n) = \max\{A_M(x) \,:\, \|x\| = n\}$$

die worst-case Laufzeit *des Algorithmus M.*

Hierbei bezeichnet $\|x\|$ die Größe der Eingabe. Die worst-case-Laufzeit ist die gängige Form der durchgeführten Laufzeit-Analysen.

Definition 1.15 *Sei $A_M(x)$ die Anzahl von Operationen eines Algorithmus M im betrachteten Maschinenmodell für eine Eingabe x. Dann ist*

$$A_M^{\text{bc}}(n) = \min\{A_M(x) \,:\, \|x\| = n\}$$

die best-case Laufzeit *des Algorithmus M.*

Diese Form der Laufzeit-Analyse ist eher als ein Hinweis dafür gedacht, wie lange eine Algorithmus mindestens braucht. Sie kann einen Hinweis auf eine bessere Effizienz des Algorithmus in der Praxis geben, wenn best-case und worst-case-Laufzeit sehr unterschiedlich sind.

Für die Praxis ist auch die average-case-Laufzeit von besonderer Bedeutung.

Definition 1.16 *Sei $A_M(x)$ die Anzahl von Operationen eines Algorithmus M im betrachteten Maschinenmodell für eine Eingabe x. Sei weiter $\{p_n(x) : n \in \mathbb{N}\}$ eine Familie von Verteilung auf den Eingaben der Länge n, d.h. $\sum_{\|x\|=n} p_n(x) = 1$ und $p_n(x) \geq 0$ für alle $n \in \mathbb{N}$. Dann ist*

$$A_M^{\text{ac}}(n) = \sum_{\|x\|=n} p_n(x) \cdot A_M(x)$$

die average-case Laufzeit *des Algorithmus M.*

Der Vorteil hierbei ist, dass dabei die Verteilung der Eingabe berücksichtigt wird und die angegebene Laufzeit für einen *durchschnittlichen* Fall genauer ist, auch wenn für spezielle (in der Praxis aber eventuell selten eintretenden Fälle) die worst-case-Laufzeit angenommen wird.

Nachteile sind zum einen die unbekannte Verteilung der Eingaben. Zum anderen sind Berechnungen für andere Verteilungen als die Gleichverteilung auch schwer durchzuführen. Weiterhin gibt die average-case-Laufzeitanalyse auch nur die zu erwar-

tenende Laufzeit an. Es kann dennoch im konkreten Fall die worst-case Lauftzeit eintreten.

Daher bleibt die worst-case-Analyse weiterhin die wichtigste Analyseform für einen Algorithmus. Die average-case-Analyse kann Hinweise darauf geben, in wie weit man mit einem besseren Laufzeitverhalten als mit der worst-case-Laufzeit hoffen darf. Da die average-case-Analyse in der Regel mathematisch aufwendig ist, führt man zuerst eine best-case-Analyse durch, die Hinweise liefert, um wie viel besser die average-case Laufzeit als die worst-case-Laufzeit überhaupt sein kann.

1.2.3 Eingabegröße

Im letzten Abschnitt haben wir von der Eingabegröße $\|x\|$ einer Eingabe x gesprochen, ohne diese genauer zu definieren. In unserem einführenden Beispiel war anscheinend klar, dass hiermit die Anzahl der Folgenglieder gemeint ist.

Betrachten wir jetzt das Problem der Multiplikation zweier natürlicher Zahlen, also $M(x_1, x_2) = x_1 \cdot x_2$. Egal welche Eingaben x wir betrachten, anscheinend ist die Eingabegröße immer 2. Damit ist der Aufwand aber für jeden Algorithmus, der dieses Problem löst, jedoch $A(x) = c$ für eine Konstante c, falls $\|x\| = 2$ ist und $A(x) = 0$ sonst.

Das kann aber für dieses Problem nicht in unserem Sinne sein. Als elementare Operationen würden wir vermutlich Bit-Operationen wählen, und dann benötigt $x_i \in \mathbb{N}$ in der Darstellung als Binärzahl ohne führende Nullen genau $\lfloor \log(x_i) \rfloor + 1$ Bits. Wir merken hier noch an, dass in der Informatik der Logarithmus auf $\mathbb{N}_0$ in der Regel mittels $\log(0) := 0.$ fortgesetzt wird.

Wir müssen uns also für jedes Problem nicht nur überlegen, was wir als charakteristische Operationen wählen wollen, sondern auch wie die Eingabegröße für das Problem definiert ist.

Ein weiteres Problem tritt beim Zählen der charakteristischen Operationen auf. Führen wir beispielsweise die Multiplikation von $x_1 \cdot x_2$ auf x_1 Additionen von x_2 zurück, dann wäre eine vernünftige Wahl der charakteristischen Operation ein Addition. Für $2 \cdot x$ erhalten wir $A(2, x) = 1$ und für $x \cdot 2$ den Aufwand $A(x, 2) = x - 1$.

Da die Multiplikation ja kommutativ ist, würden wir ja eher erwarten, dass $A(2, x)$ und $A(x, 2)$ denselben Aufwand verursachen. Verhält sich dieser Algorithmus tatsächlich so asymmetrisch oder haben wir einen methodischen Fehler gemacht? Im ersten Fall addieren wir zwei relativ lange Binärzahlen, im anderen Falle addieren wir relative viele Binärezahlen der Länge 2. Also sind unsere Algorithmen schon verschieden. Jedoch werden wir vermutlich bei der Implementierung der Addition

auf die Schulmethode zurückgreifen. Die Anzahl der dort ausgeführten Addition auf $\mathbb{B} := \{0, 1\}$ ist abhängig von der Länge der beteiligte Binärzahlen.

Wenn wir mit relativ großen Zahlen (im Vergleich zur Eingabelänge) umgehen, ist in der Regel das so genannte *logarithmische Kostenmaß* vernünftiger. Hierbei wird jede charakteristische Operation mit der maximalen Länge der beteiligten Operanden in einer sinnvollen Binärdarstellung gewichtet.

Beim logarithmischen Kostenmaß muss man darauf achten, dass auch alle impliziten Operanden berücksichtigt werden. Beispielsweise muss man bei einer indirekten Adressierung eines Operanden durch $A[B[i]]$ nicht nur die Länge des eigentlichen Operanden $A[B[i]]$ berücksichtigen, sondern auch die der impliziten Operanden i und $B[i]$. Eventuell muss man auch die Operandenlängen derjenigen nichtcharakteristischen Operationen berücksichtigen, die von der zugehörigen charakteristischen Operation majorisiert werden.

Wird jede charakteristische Operation als eine Operation gezählt, so spricht man vom *uniformen Kostenmaß*. Da sich diese meist einfacher analysieren lässt, wird auch meist dieses verwendet. Wenn man von einem Algorithmus weiß, dass alle verwendeten Operanden durch eine bestimmte Länge ℓ (die durchaus von der Eingabegröße abhängen kann) beschränkt ist, ist dieses Vorgehen in der Regel auch zulässig, da man den Aufwand im logarithmische Kostenmaß durch den Aufwand im uniformen Kostenmaß multipliziert mit ℓ abschätzen kann.

1.3 Entwurfsmethoden von Algorithmen (*)

In diesem Abschnitt wollen wir die Entwurfsmethoden noch einmal etwas genauer unter die Lupe nehmen. Hierfür betrachten wir im Folgenden zwei verschiedene Probleme. Als erstes betrachten wir das Problem des Sortierens Wir erinnern uns zunächst, was Relationen und Ordnungen sind.

Definition 1.17 *Eine* binäre Relation *R auf einer Menge M ist eine Menge R von geordneten Paaren aus M, d.h. $R \subseteq M \times M$. Eine binäre Relation R auf M heißt*

- reflexiv*, wenn $(x, x) \in R$ für alle $x \in M$;*
- symmetrisch*, wenn mit $(x, y) \in R$ auch $(y, x) \in R$;*
- antisymmetrisch*, wenn aus $(x, y) \in R$ und $(y, x) \in R$ folgt, dass $x = y$;*
- transitiv*, wenn aus $(x, y) \in R$ und $(y, z) \in R$ folgt, dass $(x, z) \in R$ ist.*

Nun kommen wir zu speziellen Relationen, die die Basis für das Sortieren bilden.

Definition 1.18 *Eine Relation R auf einer Menge M ist eine* Ordnung*, wenn R reflexiv, antisymmetrisch und transitiv ist. Eine Ordnung R auf einer Menge M heißt* total*, wenn für alle $x, y \in M$ entweder $(x, y) \in R$ oder $(y, x) \in R$ ist. Eine nichttotale Ordnung heißt* partielle Ordnung.

Oft bezeichnen wir eine totale Ordnung kurz als Ordnung. Für eine Ordnung R auf einer Menge M schreiben wir im folgenden statt $(x, y) \in R$ kürzer $x \leq y$ bzw. $y \geq x$. Mit $x < y$ bzw. $x > y$ ist $x \leq y$ bzw. $x \geq y$ und $x \neq y$ gemeint.

SORTING

Eingabe: Eine Folge $(x_1, \ldots, x_n) \in M^n$, wobei $(M, \leq)$ eine total geordnete Menge ist.
Gesucht: Eine Folge $(x_{\pi(1)}, \ldots, x_{\pi(n)}) \in M^n$, wobei $\pi \in S(n)$ und $x_{\pi(i)} \leq x_{\pi(i+1)}$ für alle $i \in [1 : n-1]$.

Als weiteres Problem betrachten wir das so genannte Geldwechsel-Problem (engl. Change-Making Problem oder Coin Change Problem). Hierbei betrachten wir eine Währung mit Münzen im Nennwert von $w_1 > \cdots > w_k$. Wir wollen dann einen Betrag B mit einer minimalen Menge von Münzen herausgeben, wobei wir für jeden Nennwert beliebig viele Münzen zur Verfügung haben.

In den USA sind dies beispielsweise die Münzen mit Nennwerten $w = (25, 10, 5, 1)$, in der EU $w = (200, 100, 50, 20, 10, 5, 2, 1)$. Für einen Betrag von 89 Cents erhält man im optimalen Fall in den USA 3 Quarters (25), 1 Dime (10) und 4 Pennies (1). In der EU einen 50er, einen 20er, einen 10er, einen 5er und zwei 2er.

CHANGE-MAKING PROBLEM (CMP)

Eingabe: $B \in \mathbb{N}_0$.
Lösung: $(n_1, \ldots, n_k) \in \mathbb{N}_0^k$ mit $\sum_{i=1}^{k} n_i w_i = B$ für $(w_1, \ldots, w_k) \in \mathbb{N}^k$ mit $w_i > w_{i+1}$ für alle $i \in [1 : k-1]$.
Optimum: Minimiere $\sum_{i=1}^{k} n_i$.

Man unterscheidet hierbei das Problem, ob die Nennwerte Teil der Eingabe sind (universelles Problem) oder nicht.

UNIVERSAL CHANGE-MAKING PROBLEM (UCMP)

Eingabe: $B \in \mathbb{N}_0$ und $(w_1, \ldots, w_k) \in \mathbb{N}^k$ mit $w_i > w_{i+1}$ für alle $i \in [1 : k-1]$.
Lösung: $(n_1, \ldots, n_k) \in \mathbb{N}_0^k$ mit $\sum_{i=1}^{k} n_i w_i = B$.
Optimum: Minimiere $\sum_{i=1}^{k} n_i$.

Das letzte Problem ist in seiner allgemeinsten Fassung sogar $\mathcal{NP}$-hart, d.h. es ist kein polynomieller Algorithmus hierfür bekannt. Die Problematik rührt allerdings im Wesentlichen daher, dass man im Allgemeinen nicht weiß, ob es für eine Liste von Nennwerten für einen gegebenen Betrag überhaupt eine Lösung gibt. Daher nimmt man beim universellen Geldwechsel-Problem oft (aber nicht immer) an, dass für die Eingabe $w_k = 1$ gilt.

1.3.1 Vollständige Aufzählung

Zunächst einmal wollen wir die beiden Probleme mit vollständiger Aufzählung lösen. Im Falle des Sortierens zählen wir einfach alle Permutationen über n Elementen auf (man überlege sich, wie man das geschickt macht), und testen, ob die so permutierte Folge sortiert ist. Dieser Algorithmus ist in Abbildung 1.17 angegeben.

Sorting (int[] x, int n)

```
begin
    forall (π ∈ S(n)) do
        for (i := 1; i < n; i++) do
            if (x[π[i]] > x[π[i + 1]]) then continue forall;
        return (x_π(1), ..., x_π(n));
end
```

Abbildung 1.17: Algorithmus: Sortieren mit vollständiger Aufzählung

Der Algorithmus ist offensichtlich korrekt, da ja eine der Permutationen die Folge sortieren muss. Da es genau $n!$ Permutationen gibt, ist die worst-case-Laufzeit durch $(n-1)\cdot n!$ beschränkt, wenn man die Anzahl der Vergleiche zählt. Da $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, ist das jedoch sehr aufwendig.

Für das universelle Geldwechsel-Problem müssen wir alle Möglichkeiten aufzählen, wie man den Betrag B darstellen kann. Dazu bedienen wir uns der Rekursion. Zuerst zählen wir alle Möglichkeiten auf, wie oft eine Münze mit Nennwert w_1 in einer möglichen Lösung enthalten sein kann: $n_1 \in [0 : \lfloor B/w_1 \rfloor]$. Anschließend zählen wir

```
UCMP (int B', ℓ, N; int[] n)
begin
    if (ℓ = k + 1) then
        if ((B' = 0) && (N < N̂)) then
            N̂ := N;
            for (i := 1; i ≤ k; i++) do
                n̂[i] := n[i];
    else
        for (i := 0; i ≤ ⌊B'/w[ℓ]⌋; i++) do
            n[ℓ] := i;
            UCMP(B' − n[ℓ] · w[ℓ], ℓ + 1, N + i, n);
end
```

Abbildung 1.18: Algorithmus: Change-Making Problem mit rekursiver vollständiger Enumeration

für den Restbetrag $B' := B - n_1 \cdot w_1$ alle Möglichkeiten für die Anzahlen von n_2 auf: $n_2 \in [0 : \lfloor B'/w_2 \rfloor]$, usw. Dieser Algorithmus ist in Abbildung 1.18 angegeben.

Bei jedem rekursiven Aufruf des Algorithmus CMP haben wir bereits eine zulässige Aufteilung der Münzen mit Nennwerten von w_1 mit $w_{\ell-1}$ vorgenommen (und zwar n_i Münzen mit Wert w_i für $i \in [1 : \ell - 1]$) und damit den Teilbetrag $B - B'$ mit $N = \sum_{i=1}^{\ell-1} n_i$ Münzen erzielt. Wir versuchen dann alle Möglichkeiten für die Münzen mit Nennwert w_ℓ auszuprobieren. Hierfür sind offensichtlich nur die Anzahlen aus $[0 : \lfloor B'/w_\ell \rfloor]$ möglich. Dabei beschreibt $(\hat{n}_1, \ldots, \hat{n}_k)$ eine bislang beste bekannte Lösung mit $\hat{N} = \sum_{i=1}^{k} \hat{n}_i$ Münzen. Für die eigentliche Berechnung initialisieren wir $\hat{N} := \infty$ und rufen die rekursive Prozedur mit $CMP(B, 1, 0, \vec{0})$ auf.

Diesen Ansatz nennt man auch *Backtracking*, da wir bei einem Lösungsversuch, der keine Lösung liefert (wenn im Algorithmus $B > 0$ und $\ell = k + 1$ ist), versuchen die zuletzt getroffene revidierbare Entscheidung zu revidieren.

1.3.2 Branch-and-Bound

Bei der vollständigen Aufzählung konstruieren wir implizit einen gewurzelten Baum, der die verschiedenen Möglichkeiten, den Betrag B zusammenzustellen, beschreibt. Ein kleiner Ausschnitt des Baumes ist in Abbildung 1.19 illustriert.

Der Knoten auf Level k (die Wurzel ist auf Level 0) sind dann Blätter. Ist der Eintrag eines Blattes 0, dann hat man eine gültige Darstellung des Betrages B durch Münzen

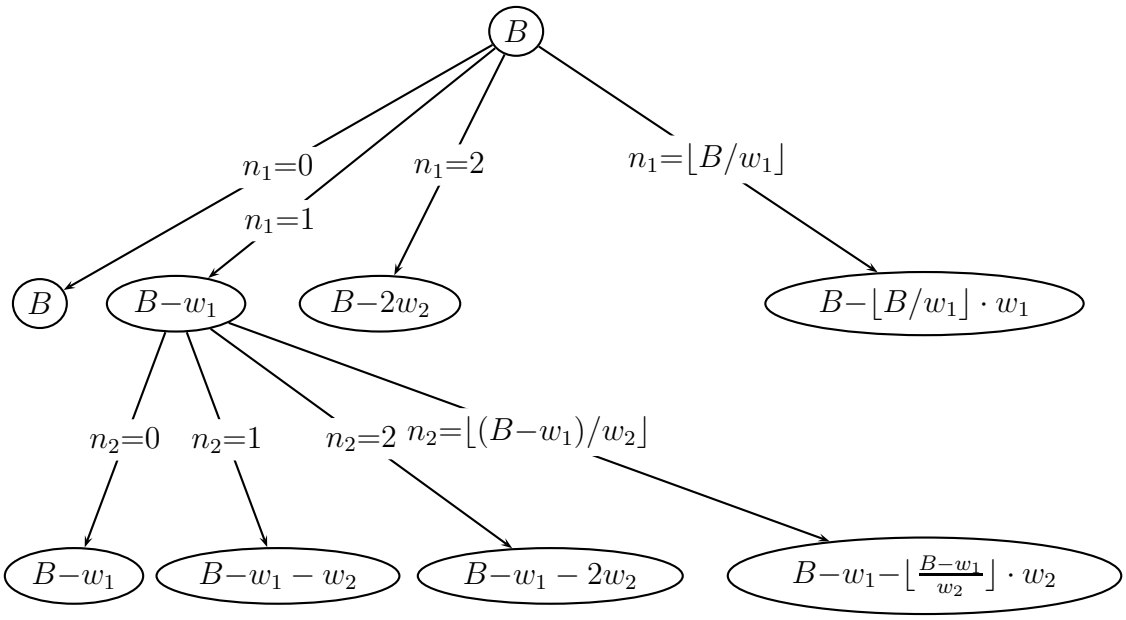


Abbildung 1.19: Skizze: Ausschnitt des Rekursionsbaums für CMP

gefunden, die durch die Kantenlabel auf dem Weg von der Wurzel zu diesem Blatt gegeben ist.

Anstatt den Baum jetzt systematisch vollständig zu durchlaufen, wie dies die vollständige Aufzählung tut, kann man versuchen, im Baum jeweils bei den Knoten den Baum weiterzuentwickeln, die am Erfolg versprechenden sind.

Dazu definieren wir für jeden Knoten v mit Label B' auf Level i (mit den zugehörigen Kantenlabeln $(n_1, \ldots, n_i)$ auf dem Weg von der Wurzel zu diesem Knoten) Wert $L(v) = \sum_{j=1}^{i} n_i + \lceil B'/w_{i+1} \rceil$. Dieser Wert ist für alle Blätter im zugehörigen Teilbaum eine untere Schranke für die verwendete Anzahl von Münzen, um den Betrag B darzustellen, was man wie folgt sieht. Nach Vorgabe werden schon $\sum_{j=1}^{i} n_i$ Münzen verwendet, um den Betrag $B - B'$ durch Münzen mit den Nennwerten $w_1 > \cdots > w_i$ darzustellen. Im günstigsten Fall können wir den Restbetrag B' durch Münzen mit dem verbleibenden größten Nennwert w_{i+1} darstellen. Falls B' nicht durch w_{i+1} teilbar ist, benötigen wir noch mindestens eine Münze mehr.

Wir werden jetzt immer von solchen Knoten die Kinder untersuchen, deren Wert $L(v)$ am kleinsten ist. Bei diesen Knoten haben wir vermutlich am ehesten die Chance, mit möglichst wenig Münzen auszukommen. Haben wir andererseits bereits eine Darstellung für den Betrag B mit N Münzen gefunden, so können wir jeden Knoten v deren zugehöriger Wert $L(v)$ größer gleich N ist verwerfen, da wir hier keine bessere Lösung mehr finden können. Damit werden also ganze Teilbäume, deren Wurzel einen zu großen L-Wert besitzt, abgeschnitten und müssen nicht mehr explizit durchsucht werden.

1.3.3 Dynamische Programmierung

Für unser Geldwechsel-Problem können wir auch die dynamische Programmierung einsetzen. Wir definieren $N[B]$ als die minimale Anzahl von Münzen, die nötig sind, um den Betrag B darzustellen. Es gilt dann die folgende Rekursionsgleichung:

$$N[B] = \begin{cases} 0 & \text{falls} \quad B = 0, \\ N[B] = \min\{1 + N[B - w_i] \,:\, i \in [1:k] \wedge w_i \leq B\} & \text{falls} \quad B > 0. \end{cases}$$

Die Überlegung hierfür ist einfach. Wenn in einer Darstellung von B die Münze mit Nennwert w_i enthalten ist, benötigt der Restbetrag $B - w_i$ in einer optimalen Darstellung genau $N[B - w_i]$ Münzen. Da wir nicht wissen, welche Münze in einer optimalen Darstellung enthalten ist (aber mindestens eine, wenn der Betrag nicht 0 ist), bilden wir das Minimum über alle Möglichkeiten (d.h. über alle Nennwerte).

Der zugehörige Algorithmus mittels dynamischer Programmierung ist in Abbildung 1.20 dargestellt. Hier wird allerdings nur die minimale Anzahl von Münzen einer optimalen Darstellung berechnet. Es bleibt dem Leser überlassen, diesen Algorithmus so zu ergänzen, dass auch die Verteilung der Münzen ausgegeben wird.

UCMP (int B)

```
begin
    for (b := 1; b ≤ B; b++) do
        N[b] := ∞;
        for (i := 1 i ≤ k; i++) do
            if (N[b − w[i]] < N[b]) then
                N[b] := min(N[b], 1 + N[b − w[i]]);
end
```

Abbildung 1.20: Algorithmus: Change-Making Problem mit dynamischer Programmierung

Eine einfache Analyse der worst-case-Laufzeit zeigt, dass der Algorithmus genau $k \cdot B$ Minimumsbildungen ausführt. Auf den ersten Blick sieht es also so aus, als hätten wir einen polynomiellen Algorithmus für UCMP gefunden. Aber hatten wir nicht zu Beginn des Abschnitts behauptet, UCMP sei $\mathcal{NP}$-hart und es daher keinen polynomiellen Algorithmus geben kann?

Die Laufzeit ist $k \cdot B$. Aber wie sieht die Eingabegröße aus? Im uniformen Kostenmaß $k + 1$. Aber was ist dann B in der Laufzeit, eine Konstante? Nein, B kann ja

beliebig groß werden. Also ist hier die genauere Betrachtung der Eingabegröße im logarithmischen Kostenmaß nötig:

$$\|(w_1, \ldots, w_k, B)\| = \sum_{i=1}^{k} (\lfloor \log(w_i) \rfloor + 1) + \lfloor \log B \rfloor + 1 \geq k + \log(B).$$

Damit ist der Term $k \cdot B$ in $n = k + \log(B)$ für große B (z.B. $B \geq 2^k$) nicht mehr polynomiell, sondern exponentiell!

1.3.4 Greedy-Algorithmen

Wir betrachten jetzt noch einen anderen Ansatz für Optimierungsprobleme, den so genannten *Greedy-Ansatz* (englisch für gierig). Hierbei wird im Hinblick auf das Optimierungsziel möglichst gierig vorgegangen, d.h. man versucht lokal eine möglichst beste Entscheidung zu treffen.

Für unser Change-Making Problem bedeutet dies, dass man so viele Münzen mit größtem Nennwert verwendet, die für den Restbetrag B' überhaupt noch möglich sind. Dieser Greedy-Algorithmus für das CMP ist in Abbildung 1.21 dargestellt.

Greedy (int B)

```
begin
    for (i := 1; i ≤ k; i++) do
        n[i] := ⌊B/w_i⌋;
        B := B − n[i] · w[i];
    return n;
end
```

Abbildung 1.21: Algorithmus: Greedy-Algorithmus für CMP

Zuerst überlegt man sich, dass der Algorithmus in der Regel nur dann eine korrekte Darstellung des Betrages B findet, wenn $w_k = 1$ gilt. Wir betrachten dazu beispielsweise die Nennwerte $w = (5, 2)$ und den Betrag $B = 8$. Der Greedy-Algorithmus wählt eine Münze mit Nennwert 5 und kann dann den Betrag 8 nicht mehr darstellen, obwohl die optimale Lösung vier Münzen mit Nennwert 2 wäre.

Auch wenn $w_k = 1$ ist, muss der Greedy-Algorithmus nicht immer eine optimale Lösung finden. Wir betrachten dazu eine Variante der US-Münzen, in der es keinen Nickel (5 Cent-Münze) gibt, also $w = (25, 10, 1)$. Für $B = 30$ wählt der Greedy-Algorithmus 1 Quarter und 5 Pennies, also 6 Münzen, obwohl eine Wahl von 3 Dimes optimal ist.

Für das normale System des US-Münzen ist der Greedy-Algorithmus jedoch optimal.

Lemma 1.19 *Für $w = (25, 10, 5, 1)$ konstruiert der Greedy-Algorithmus eine optimale Lösung.*

Beweis: Zuerst halten wir fest, dass eine optimalen Lösung

a) maximal 4 Pennies enthält (sonst könnte man 5 Pennies durch einen Nickel ersetzen);

b) maximal 1 Nickel enthält (sonst könnte man 2 Nickels durch einen Dime ersetzen);

c) maximal 2 Dimes enthält (sonst könnte man 3 Dimes durch einen Quarter und einen Nickel ersetzen);

d) maximal einen Dime enthält,wenn sie einen Nickel enthält (sonst könnte man 2 Dimes und 1 Nickel durch einen Quarter ersetzen).

Wir führen den Beweis durch Widerspruch. Wir nehmen also an, es gibt einen Betrag B, so dass die optimale Lösung $(\hat{n}_1, \ldots, \hat{n}_4)$ weniger Münzen enthält als die Greedy-Lösung $(n_1, \ldots, n_4)$.

Des Weiteren wählen wir unter allen solchen Gegenbeispielen ein kleinstes aus, d.h. eines mit dem kleinstem Betrag B. Für alle Beträge $B' < B$ ist die Greedy-Lösung also optimal.

Zuerst behaupten wir, dass für ein kleinstes Gegenbeispiel B die optimale Lösung keinen Quarter enthält. Würde sie einen Quarter enthalten, wäre $B \geq 25$ und der Greedy-Algorithmus würde ebenfalls mindestens einen Quarter auswählen. Wir behaupten, dass dann $B-25$ ein kleineres Gegenbeispiel ist (was einen Widerspruch zu unserer Annahme liefert). Wir können in der optimalen Lösungen einen Quarter entfernen und erhalten eine Lösung $\hat{n}' = (\hat{n}_1 - 1, \hat{n}_2, \hat{n}_3, \hat{n}_4)$. Der Greedy-Algorithmus liefert nach Konstruktion die Lösung $n' = (n_1 - 1, n_2, n_3, n_4)$. Die Lösung $\hat{n}'$ muss zwar für $B - 25$ nicht optimal sein, verwendet aber offensichtlich weniger Münzen als der die Lösung n' des Greedy-Algorithmus.

Jetzt behaupten wir, dass die optimale Lösung nicht 2 Dimes enthalten kann. Würde sie zwei Dimes beinhalten, könnte sie nach Eigenschaft d) keinen Nickel enthalten. Mit Eigenschaft a) folgt, dass dann $B \in [20 : 24]$ wäre. Dann würde die optimale Lösung aus 2 Dimes und $B-20$ Pennies bestehen, die auch der Greedy-Algorithmus wählt. Somit wäre B kein Gegenbeispiel.

Nun behaupten wir, dass die optimale Lösung überhaupt keinen Dime beinhalten kann. Würde sie genau einen Dime beinhalten, so folgt aus den Eigenschaften a)

und b), dass $B \in [10 : 19]$. Enthält die optimale Lösung des Weiteren einen Nickel, so gilt $B \in [15 : 19]$ und wäre mit der Greedy-Lösung äquivalent, was den gewünschten Widerspruch liefert. Wäre andererseits kein Nickel in der optimalen Lösung, so wäre $B \in [10 : 14]$ und die Greedy-Lösung ebenfalls mit der optimalen Lösung gleich. Wir erhalten also in jedem Fall einen Widerspruch.

Zuletzt behaupten wir, dass die optimale Lösung keinen Nickel enthalten kann. Ansonsten würde sie nach Eigenschaft b) genau einen Nickel enthalten und mit Eigenschaft a) folgt $B \in [5 : 9]$. Auch hier wäre die optimale Lösung mit der Greedy-Lösung identisch.

Somit kann die optimale Lösung unseres kleinsten Gegenbeispiels nur aus Pennies bestehen, und nach Eigenschaft a) gilt $B \in [0 : 4]$. Auch in diesem Fall liefert der Greedy-Algorithmus die optimale Lösung und der Beweis ist vollendet. ■

Die Methode bei einem Widerspruchsbeweis nicht ein beliebiges, sondern ein bestimmtes (hier ein gewissermaßen kleinstes) Gegenbeispiel auszuwählen, wird oft auch als die Methode des kleinsten Verbrechers genannt. Dieser Beweistrick kann viele Widerspruchsbeweise technisch einfacher machen.

Wir halten noch folgendes Lemma fest, das bei einer Überprüfung der Optimalität des Greedy-Algorithmus für CMP hilfreich sein kann, aber im Allgemeinen nicht effizient einsetzbar ist.

Lemma 1.20 *Wenn für das Geldwechsel-Problem mit $w_k = 1$ die Greedy-Strategie nicht optimal ist, dann gibt es ein Gegenbeispiel B mit $B \in [1 : w_1 + w_2]$.*

Für Details verweisen wir den Leser auf die Originalarbeit von D. Kozen und S. Zaks.

1.3.5 Rekursion

In den letzten Teilen dieses Abschnittes wenden wir uns wieder dem Sortierproblem zu. Wir hatten ja gesehen, dass Rekursion ein gängiges Mittel ist, um Probleme zu lösen. Nun wollen wir diese Paradigma auf das Sortierproblem anwenden.

Gegeben sei eine Folge $(x_1, \ldots, x_n)$ mit n Elementen. Wir sortieren zunächst die ersten $n - 1$ Elemente rekursiv. Wir müssen uns dann nur noch überlegen, wie wir mit dem n-ten Element umgehen. Wenn die ersten $n - 1$ Elemente jedoch sortiert sind, so können wir das n-te Element sehr einfach in diese sortierte Liste einfügen. Dazu vergleichen wir jeweils das Element an Position i (zu Beginn ist $i = n$) mit dem Element an Position $i - 1$ und vertauschen die beiden Elemente, wenn dass

```
Insertionsort (int[] x, int n)
begin
    if (n > 1) then
        Insertionsort(x,n − 1);
        for (i := n; i > 1; i−−) do
            if (x_{i−1} > x_i) then swap(i − 1,i);
            else break;
end
```

Abbildung 1.22: Algorithmus: Insertionsort

Element an Position $i-1$ das größere ist. Dieser Algorithmus heißt *Insertionsort* und ist in Abbildung 1.22 angegeben. Dabei vertauscht die Operation $\texttt{swap}(i,j)$ die beiden Elemente an Position i und j im Feld x.

Wie sieht es nun mit der Laufzeit $A^{\text{wc}}_{\text{Ins}}(n)$ hiervon aus? Als typische Operation zur Analyse von Sortieralgorithmen verwenden wir die Vergleichsoperation, da dies die einzige Operation ist, die auf beliebigen total geordneten Mengen zur Verfügung steht und auf die somit jeder allgemeine Sortieralgorithmus zurückgreifen muss.

Wir sortieren zuerst rekursiv mit maximal $A^{\text{wc}}_{\text{Ins}}(n-1)$ Vergleichen die verkürzte Anfangsfolge. Für das Einsortieren benötigen wir maximal $(n-1)$ Vergleiche. Weiterhin halten wir fest, dass einelementige Folgen trivialerweise sortiert sind. Somit erhalten wir:

$$
\begin{aligned}
A^{\text{wc}}_{\text{Ins}}(n) &= A^{\text{wc}}_{\text{Ins}}(n-1) + (n-1) \\
&\quad \text{erneutes Einsetzen liefert} \\
&= A^{\text{wc}}_{\text{Ins}}(n-2) + (n-2) + (n-1) \\
&= A^{\text{wc}}_{\text{Ins}}(n-3) + (n-3) + (n-2) + (n-1) \\
&\quad \text{nach } k\text{-maligen Einsetzen} \\
&= A^{\text{wc}}_{\text{Ins}}(n-k) + \sum_{i=1}^{k} (n-i) \\
&\quad \text{mit } k = n-1 \\
&= A^{\text{wc}}_{\text{Ins}}(1) + \sum_{i=1}^{n-1} (n-i) \\
&\quad \text{Rückwärtige Summation} \\
&= 0 + \sum_{i=1}^{n-1} i
\end{aligned}
$$

$$= \overset{\text{Gaußsche Summe}}{\frac{n(n-1)}{2}}.$$

Theorem 1.21 *Insertionsort benötigt zum Sortieren von n Elementen einer total geordneten Menge im worst-case maximal $\binom{n}{2}$ Vergleiche.*

Wir können bei der Rekursion auch erst ein sinnvolles Element auswählen und dann die Rekursion aufrufen. Beispielsweise können wir zuerst das größte Element an die n-te Position bewegen und dann die ersten $n-1$ Elemente rekursiv sortieren. Dies führt zum Algorithmus Selectionsort, der in Abbildung 1.23 angegeben ist.

```
Selectionsort (int[] x, int n)
begin
    if (n > 1) then
        for (i := 1; i < n; i++) do
            if (x_i > x_n) then swap(i,n);
        Selectionsort(x,n − 1);
end
```

Abbildung 1.23: Algorithmus: Selectionsort

Wie sieht es nun mit der Laufzeit $A_{\text{Sel}}(n)$ hiervon aus? Zuerst bestimmen wir mit genau $n-1$ Vergleichen das maximale Element der Folge und sortieren dann rekursiv die $n-1$ kleinsten Elemente mit genau $A_{\text{Sel}}(n-1)$ Vergleichen. Wiederum bemerken wir, dass einelementige Folgen trivialerweise sortiert sind. Somit erhalten wir

$$A_{\text{Sel}}(n) = \begin{cases} 0 & \text{falls} \quad n = 1, \\ A_{\text{Sel}}(n-1) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Wir erhalten also die gleiche Rekursionsgleichung wie beim Insertionsort. Damit ergibt sich die gleiche Lösung von genau $\binom{n}{2}$ Vergleichen.

Wir halten noch fest, dass es sich hierbei sowohl um eine worst-case- wie best-case- und somit auch average-case-Analyse handelt, wenn man die Anzahl von Vergleichen betrachtet. Für die Anzahl von Swaps wäre dies nur eine worst-case-Analyse. Der Leser möge sich überlegen warum.

Theorem 1.22 *Selectionsort benötigt zum Sortieren von n Elementen einer total geordneten Menge genau $\binom{n}{2}$ Vergleiche.*

1.3.6 Divide-and-Conquer

Nun wollen wir noch versuchen, den Divide-and-Conquer-Ansatz auf das Sortieren loszulassen. Wir stellen dazu zunächst einmal das abstrakte Konzept vor.

Das Prinzip des Divide-and-Conquer (oft auch als divide-et-impera oder Teile-und-Herrsche-Prinzip bezeichnet) lässt sich im Wesentlichen in die folgenden drei Phasen gliedern:

- Zuerst wird das Problem in mehrere kleinere Teilprobleme derselben Art aufgeteilt (im so genannten *Divide-Schritt*);
- Dann werden die kleineren Teilprobleme rekursiv gelöst;
- Schließlich wird aus den rekursiv gewonnen Lösungen der Teilprobleme eine Lösung für das Gesamtproblem konstruiert (im so genannten *Conquer-Schritt*).

Wie können wir dieses Paradigma also auf das Sortieren anwenden. Da uns zunächst einmal nichts besseres einfällt, teilen wir die Folge $(x_1, \ldots, x_n)$ zuerst in zwei Folgen $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ und $(x_{\lfloor n/2 \rfloor+1}, \ldots, x_n)$ der Länge $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ auf. Diese beiden Folgen sortieren wir dann rekursiv und erhalten die beiden sortierten Folgen $(x_{\pi(1)}, \ldots, x_{\pi(\lfloor n/2 \rfloor)})$ sowie $(x_{\pi'(\lfloor n/2 \rfloor+1)}, \ldots, x_{\pi'(n)})$.

Wir erhalten wir nun aus zwei sortierten Folge eine? Nun ja, wir können die beiden sortierten Folgen zu einer sortierten Folge zusammenmischen. Dazu vergleichen wir jeweils die beiden kleinsten Elemente der beiden verbleibenden sortierten Restfolgen und hängen das kleinere Element an die bereits vollständig sortierte Liste (bzw. Feld) hinten an. Damit erhalten wir den so genannten Mergesort, wie in Abbildung 1.24 dargestellt.

Mergesort (int[] x, int ℓ, r)

```
begin
    // sorts region [ℓ : r] of array x
    if (ℓ < r) then
        m := (ℓ + r)/2;
        Mergesort(ℓ,m);
        Mergesort(m + 1,r);
        merge(x,ℓ,m,r);          /* merging (x_ℓ, ..., x_m) and (x_{m+1}, ..., x_r) */
end
```

Abbildung 1.24: Algorithmus: Mergesort

Wie sieht es für die Laufzeit $A_{\mathrm{M}}(n)$ aus? Für das rekursive Sortieren benötigen wir $A_{\mathrm{M}}(\lfloor n/2 \rfloor)$ bzw. $A_{\mathrm{M}}(\lceil n/2 \rceil)$ Vergleiche. Für das Zusammenmischen benötigen wir im worst-case $n-1$ Vergleiche. Damit ergibt sich

$$A_{\mathrm{M}}(n) = \begin{cases} 0 & \text{falls} \quad n = 1, \\ A_{\mathrm{M}}(\lfloor n/2 \rfloor) + A_{\mathrm{M}}(\lceil n/2 \rceil) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Dies ist fast dieselbe Rekursionsgleichung wie im Falle des Divide-and-Conquer-Algorithmus für das Maximal Scoring Subsequence Problem bis auf den additiven Term. Eine analoge Analyse für Zweierpotenzen n ergibt $A_{\mathrm{M}}(n) = n \log(n) - n + 1$.

Können wir uns im Divide-Schritt auch etwas geschickter anstellen, so dass der Conquer-Schritt leichter wird? Ja, wenn die erste Hälfte nur relativ kleine Elemente enthält und die zweite Hälfte nur relativ große Elemente.

Dazu wählen wir ein beliebiges Element e aus der zu sortierenden Folge aus und erzeugen eine Folge, deren Element alle kleiner gleich e sind, sowie eine zweite Folge, deren Elemente alle größer als e sind. Diese beiden Folgen sortieren wir dann jeweils rekursiv. Da jedes Element der ersten Folge kleiner als jedes Element der zweite Folge ist, kann man im Conquer-Step die sortierten Folgen einfach zusammenhängen, um die ganze sortierte Folge zu erhalten. Dieser Algorithmus nennt sich *Quicksort* und ist in Abbildung 1.25 dargestellt, wobei die Funktion `partition` die Aufteilung der Folge in kleine und große Elemente vornimmt.

Quicksort (int[] x, int ℓ, r)

```
begin
    // sorts region [ℓ : r] of array x
    if (ℓ < r) then
        // partition subarray x[ℓ : r], such that
        // x_i ≤ x_m und x_j > x_m for i ∈ [ℓ : m] and j ∈ [m + 1 : r]
        m := partition(x, ℓ, r, r);
        Quicksort(ℓ,m − 1);
        Quicksort(m + 1,r);
end
```

Abbildung 1.25: Algorithmus: Quicksort

Wie kann man nun die Funktion `partition` auf einem Array realisieren? Dazu wählen wir zunächst ein Element als Aufteilungselement aus. Die Wahl dieses Elements kann die Effizienz des Algorithmus zwar beträchtlich beeinflussen, für die Korrektheit des Algorithmus, ist die Wahl jedoch beliebig.

Dann gehen wir zuerst von links nach rechts durch das Feld und finden das erste Element x_i größer als e. Anschließend gehen wir von rechts nach links durch das

Feld und finden das erste Element x_j kleiner gleich e. Dann vertauschen wir x_i mit x_j und beginnen ab Position $i+1$ nach rechts und ab Position $j-1$ nach links von neuem. Sobald $i > j$ wird, brechen wir ab. Dann befinden sich zu Beginn bis Position $i-1$ nur Elemente kleiner gleich e und ab Position i nur Elemente größer als e. Das bedeutet, dass die Position i unsere Trennposition ist.

Wie sieht es nun mit der Laufzeit $A_{\mathrm{Q}}(n)$ aus? Für das rekursive Sortieren benötigen wir $A_{\mathrm{Q}}(n_1)$ bzw. $A_{\mathrm{Q}}(n_2)$ Vergleiche, wobei $n_1 + n_2 = n$ ist. Für das Aufteilen benötigen wir offensichtlich genau $n-1$ Vergleiche. Damit ergibt sich

$$A_{\mathrm{Q}}(n) = \begin{cases} 0 & \text{falls} \quad n \leq 1, \\ A_{\mathrm{Q}}(n_1) + A_{\mathrm{Q}}(n_2) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Die explizite Laufzeit hängt anscheinend von der jeweiligen Größe der entstehenden Teilfelder ab.

Man kann sich überlegen, dass der worst-case vermutlich angenommen wird, wenn jeweils die zweite Hälfte (oder auch die erste) leer ist. Dann erhalten wir folgende Rekursionsgleichung

$$A_{\mathrm{Q}}^{\mathrm{wc}}(n) = \begin{cases} 0 & \text{falls} \quad n \leq 1, \\ A_{\mathrm{Q}}^{\mathrm{wc}}(n-1) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Dies ist wiederum dieselbe Rekursionsgleichung wie im Falle von Selection- oder Insertionsort.

Theorem 1.23 *Quicksort benötigt zum Sortieren von n Elementen einer total geordneten Menge im worst-case $\binom{n}{2}$ Vergleiche.*

Das dies wirklich der worst-case ist, weist man durch die Gültigkeit von $A_{\mathrm{Q}}^{\mathrm{wc}}(n) \leq \binom{n}{2}$ mittels vollständiger Induktion nach.

Man kann sich weiterhin überlegen, was wir hier nicht explizit tun wollen, dass der beste Fall eintritt, wenn die beiden Hälften etwa gleich groß sind. Dann erhalten wir folgende Rekursionsgleichung

$$A_{\mathrm{Q}}^{\mathrm{bc}}(n) = \begin{cases} 0 & \text{falls} \quad n \leq 1, \\ A_{\mathrm{Q}}^{\mathrm{bc}}(\lfloor n/2 \rfloor) + A_{\mathrm{Q}}^{\mathrm{bc}}(\lceil n/2 \rceil - 1) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Diese Rekursionsgleichung ist sehr ähnlich zu der von Mergesort. Wir wollen sie jetzt hier auch nicht weiter analysieren, sondern nur festhalten, dass auch hier ein ähnliches Ergebnis gilt: $A_{\mathrm{Q}}^{\mathrm{bc}}(n) \geq n \log(n) - cn$ für eine Konstante $c > 0$.

In diesem Fall unterscheidet sich der best-case vom worst-case deutlich und eine average-case Analyse scheint hier gerechtfertigt. Wir nehmen der Einfachheit halber

an, dass alle Eingaben gleich wahrscheinlich sind. Was bedeutet das bei einem Sortierproblem? Da wir nur die Vergleichsoperationen als charakteristische Operationen betrachten, wollen wir hier annehmen, dass alle Permutationen einer Folge von n Elementen der total geordneten Menge M gleich wahrscheinlich sind. Des Weiteren nehmen wir hier für die Analyse an, dass alle zu sortierenden Elemente paarweise verschieden sind.

Als Aufteilungselement nehmen wir jeweils das letzte Element der Folge. Bei Quicksort wird ein solches Aufteilungselement in der Regel als *Pivot-Element* bezeichnet. Für die weitere Analyse benötigen wir noch eine Definition

Definition 1.24 *Ein Element einer total geordneten Menge hat* Rang k*, wenn genau* $k-1$ *Elemente der Menge kleiner sind.*

Offensichtlich trennt ein ein Pivotelement mit Rang k die beiden rekursiv zu sortierenden Teilmengen in eine der Größe $k-1$ und eine der Größe $n-k$. Mit welcher Wahrscheinlichkeit erhalten wir nun ein Pivot mit Rang k. Es gibt genau $n!$ Permutationen einer Folge mit n Elementen und $(n-1)!$ Permutationen, in denen das Element mit Rang k am Ende steht. Also ist die Wahrscheinlichkeit ein Element mit Rang k als Pivot zu wählen genau $\frac{(n-1)!}{n!} = \frac{1}{n}$.

Somit ergibt sich folgende Rekursionsgleichung

$$A_{\mathrm{Q}}^{\mathrm{ac}}(n) = \begin{cases} 0 & \text{falls} \quad n \in [0:1], \\ \displaystyle\sum_{k=1}^{n} \frac{1}{n}\Big[A_{\mathrm{Q}}^{\mathrm{ac}}(k-1) + A_{\mathrm{Q}}^{\mathrm{ac}}(n-k) + (n-1)\Big] & \text{falls} \quad n > 1. \end{cases} \tag{1.1}$$

Allerdings muss man hier noch nachweisen, dass die nach dem Partitionsschritt entstehenden beiden Teilfolgen jeweils auch wieder gleichverteilt sind. Da dies technisch etwas aufwendig ist, wollen wir es hier nicht ausführen.

Für $n > 1$ lässt sich die Rekursionsgleichung umschreiben zu

$$A_{\mathrm{Q}}^{\mathrm{ac}}(n) = (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} A_{\mathrm{Q}}^{\mathrm{ac}}(k).$$

Die Lösung dieser komplexeren Rekursionsgleichung verschieben wir auf später.

1.4 Analyse von Algorithmen

In diesem Abschnitt wollen wir auf grundlegende Analysetechniken für Algorithmen eingehen, d.h. wie gibt man Laufzeit geschickt an, wie löst man Rekursionsgleichungen auf, etc.

1.4.1 Landausche Symbole

Wie wir bereits gesehen haben, sind genaue Angaben zur Laufzeitkomplexität nicht ganz einfach und zum Teil auch gar nicht durchführbar. Daher werden wir uns bei Angaben von Zeit- und Platzkomplexitätsschranken oft mit asymptotischen Abschätzungen zufrieden geben.

Bei der Analyse von Algorithmen werden wir daher die Komplexität meist mit Hilfe der *Landauschen Symbole* (auch *Groß-O-Notation* genannt) angeben. Diese verschleiern die genauen Größen dieser Proportionalitätskonstanten. Die asymptotischen Resultate sind jedoch für einen qualitativen Vergleich von Algorithmen im Allgemeinen aussagekräftig genug.

Definition 1.25 *Sei $f : \mathbb{N} \to \mathbb{R}_+$ eine Funktion, dann sind die Landauschen Symbole wie folgt definiert:*

$$
\begin{aligned}
O(f) &:= \{g : \mathbb{N} \to \mathbb{R}_+ \,:\, \exists c{\in}\mathbb{R}.c{>}0,\, n_0{\in}\mathbb{N}\colon \forall n{\in}\mathbb{N}.n{\geq}n_0\colon g(n) \leq c \cdot f(n)\}\,,\\
\Omega(f) &:= \{g : \mathbb{N} \to \mathbb{R}_+ \,:\, \exists c{\in}\mathbb{R}.c{>}0,\, n_0{\in}\mathbb{N}\colon \forall n{\in}\mathbb{N}.n{\geq}n_0\colon g(n) \geq c \cdot f(n)\}\,,\\
\Omega_\infty(f) &:= \{g : \mathbb{N} \to \mathbb{R}_+ \,:\, \exists c{\in}\mathbb{R}.c{>}0\colon \forall m{\in}\mathbb{N}\colon \exists n{\in}\mathbb{N}.n{>}m\colon g(n) \geq c \cdot f(n)\}\,,\\
\Theta(f) &:= \{g : \mathbb{N} \to \mathbb{R}_+ \,:\, g{\in}O(f) \wedge g{\in}\Omega(f)\} \quad = \quad O(f) \cap \Omega(f),\\
o(f) &:= \left\{g : \mathbb{N} \to \mathbb{R}_+ \,:\, \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\right\},\\
\omega(f) &:= \left\{g : \mathbb{N} \to \mathbb{R}_+ \,:\, \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\right\}.
\end{aligned}
$$

In der Informatik schreibt man im Zusammenhang mit den Landauschen Symbolen oft $=$ anstatt von $\in$, also z.B. $g = O(f)$ statt $g \in O(f)$. In der Regel wird aus dem Zusammenhang klar, was eigentlich gemeint ist. Bei dieser Schreibweise muss man allerdings etwas aufpassen, da $=$ im Allgemeinen eine symmetrische Relation bezeichnet. Bei den Landauschen Symbolen macht $O(f) = g$ wenig Sinn und sollte deshalb auch vermieden werden. Besonders aufpassen muss man auch bei folgender Schreibweise $f = O(g) = O(h)$, wobei $f \in O(g) \subseteq O(h)$ und damit $f \in O(h)$ gemeint ist. Dabei könnte man meinen, dass damit auch $O(g) = O(h)$ behauptet wird, was aber meist nicht der Fall ist und oft auch nicht gilt.

Betrachtet man Funktionen $f, g : \mathbb{N} \to \mathbb{R}$, so schreibt man $g = O(f)$, wenn $|g| = O(|f|)$ gilt, wobei hier $|f| : \mathbb{N} \to \mathbb{R}_+ : x \mapsto |f(x)|$ für $f : \mathbb{N} \to \mathbb{R}$ ist. Dies gilt analog auch für die anderen Landauschen Symbole.

Für das Folgende ist es wichtig, ein Gefühl dafür zu bekommen, welche Beziehungen bezüglich der Landauschen Symbole für zwei vorgegebene Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ gelten. Dazu betrachten wir zunächst ein paar Beispiele und elementare Rechenregeln.

1. $\forall k, \ell \in \mathbb{N}.k > \ell : \quad n^\ell = o(n^k)$:

   Weil $\lim\limits_{n\to\infty} \dfrac{n^\ell}{n^k} = \lim\limits_{n\to\infty} n^{\ell-k} = 0.$

2. $\forall k, \ell \in \mathbb{N}.k > \ell : \quad n^k + n^\ell = \Theta(n^k)$:

   Weil $\quad n^k + n^\ell \leq 2 \cdot n^k, \quad$ gilt $\quad n^k + n^\ell = O(n^k) \quad$ [mit $c = 2$, $n_0 = 1$].

   Und weil $\quad n^k + n^\ell \geq n^k, \quad$ gilt $\quad n^k + n^\ell = \Omega(n^k) \quad$ [mit $c = 1$, $n_0 = 1$].

3. Sei $p(n) \geq 0$ ein Polynom vom Grad k, dann gilt $p \in \Theta(n^k)$:

   Sei also $p(n) = \sum_{i=0}^{k} a_i n^i$ mit $a_k > 0$. Mit $a := \max\{|a_0/a_k|, \ldots, |a_{k-1}/a_k|\}$ können wir $p(n)$ für $n \geq 2$ wie folgt nach unten abschätzen:

$$\begin{aligned}
p(n) &= a_k n^k \left(1 + \sum_{i=0}^{k-1} \frac{a_i}{a_k} n^{i-k}\right) \\
&\geq a_k n^k \left(1 + \sum_{i=0}^{k-1} \frac{-a}{n^{k-i}}\right) \\
&= a_k n^k \left(1 - a \sum_{i=1}^{k} \frac{1}{n^i}\right) \\
&\quad \text{mit Hilfe der geometrischen Reihe } \textstyle\sum_{i=\ell}^{k} x^i = \frac{x^{k+1} - x^\ell}{x-1} \\
&= a_k n^k \left(1 - a \frac{\frac{1}{n^{k+1}} - \frac{1}{n}}{\frac{1}{n} - 1}\right) \\
&\quad \text{Erweitern des Bruchs mit } -n \\
&= a_k n^k \left(1 - a \frac{1 - \frac{1}{n^k}}{n-1}\right) \\
&\quad \text{da } \tfrac{1}{n^k} \geq 0 \\
&\geq a_k n^k \left(1 - a \frac{1}{n-1}\right) \\
&\quad \text{für } n \geq 2 \text{ gilt } n - 1 \geq n/2 \\
&\geq a_k n^k \left(1 - \frac{a}{n/2}\right)
\end{aligned}$$

   Weiter erhalten wir mit $c := a_k/2 > 0$ für alle $n \geq 4a$:

$$\geq a_k n^k \left(1 - \frac{a}{2a}\right)$$

$$\begin{aligned} &= \frac{a_k}{2} n^k \\ &= c \cdot n^k. \end{aligned}$$

Somit ist $p \in \Omega(n^k)$ [mit $c = \frac{a_k}{2}$, $n_0 = \max\{2, 4a\}$].

Auf der anderen Seite gilt offensichtlich mit $c := 2 \cdot \max\{a_0, \ldots, a_k\} > 0$ für alle $n \in \mathbb{N}$ mit $n \geq 2$:

$$\begin{aligned} p(n) &= \sum_{i=0}^{k} a_i n^i \\ &\leq \frac{c}{2} \sum_{i=0}^{k} n^i \\ &\quad \text{mit Hilfe der geometrischen Reihe } \textstyle\sum_{i=\ell}^{k} x^i = \frac{x^{k+1} - x^\ell}{x-1} \\ &= \frac{c}{2} \cdot \frac{n^{k+1} - 1}{n - 1} \\ &\quad \text{aus } n \geq 2 \text{ folgt, dass } n - 1 \geq n/2 \\ &\leq \frac{c}{2} \cdot \frac{n^{k+1}}{n/2} \\ &= c \cdot n^k. \end{aligned}$$

Damit ist $p \in O(n^k)$ [mit $c = 2 \cdot \max\{a_0, \ldots, a_k\}$, $n_0 = 2$].

07.05.19

4. $\forall k \in \mathbb{N}: \quad n^k = o(2^n)$:

   Dies gilt, weil $\lim\limits_{n\to\infty} \frac{n^k}{2^n} = \lim\limits_{n\to\infty} \frac{2^{k\log(n)}}{2^n} = \lim\limits_{n\to\infty} 2^{k\log(n)-n} = 0$.

   Es bleibt nur zu zeigen, dass $n - k\log(n) \to \infty$ für $n \to \infty$. Es gilt:

   $$\forall n \in \mathbb{N},\ \ell \in \mathbb{N}: \quad \ln(n) \leq \frac{n}{\ell} + \ell. \tag{1.2}$$

   Damit erhalten wir für $\ell = 4k$ mit (1.2):

   $$\lim_{n\to\infty} (n - k \underbrace{\log(n)}_{=\frac{\ln(n)}{\ln(2)} \leq 2\ln(n)}) \geq \lim_{n\to\infty} \left(n - 2k\left(\frac{n}{4k} + 4k\right)\right) = \lim_{n\to\infty} \left(\frac{n}{2} - 8k^2\right) = \infty.$$

   Wir müssen nur noch (1.2) zeigen. Für $n = \ell$ gilt (1.2) offensichtlich, da $\ln(\ell) < 1 + \ell$. Da für alle $x \in \mathbb{R}$ mit $x > \ell$

   $$\frac{\partial}{\partial x} \ln(x) = \frac{1}{x} < \frac{1}{\ell} = \frac{\partial}{\partial x} \left(\frac{x}{\ell} + \ell\right),$$

   gilt (1.2) für $n \geq \ell$.

   Für $n \leq \ell$ ist aber sicherlich $\ln(n) < n \leq \ell \leq \frac{n}{\ell} + \ell$. Also gilt (1.2) auch für $n \leq \ell$.

5. $\forall k \in \mathbb{N} : \ \forall \varepsilon \in \mathbb{R}.\varepsilon > 0 : \quad \log^k(n) = o(n^\varepsilon)$:
   Übungsaufgabe (ähnlich wie 4.).

6. $2^n = o(2^{2n})$:
   Weil $\lim\limits_{n\to\infty} \dfrac{2^n}{2^{2n}} = \lim\limits_{n\to\infty} \dfrac{1}{2^n} = 0.$

7. $f = O(g) \ \Leftrightarrow \ g = \Omega(f)$ und $f = o(g) \ \Leftrightarrow \ g = \omega(f)$:
   Folgt unmittelbar aus der Definition.

8. $f = \Omega(g) \ \Rightarrow \ f = \Omega_\infty(g)$:
   Folgt unmittelbar aus der Definition.

9. $f = \Omega_\infty(g) \ \not\Rightarrow \ f = \Omega(g)$:
   Wähle $f(n) = 1 + (1 + (-1)^n)n^2$ und $g(n) = n$.

10. $f = \mathcal{O}(g) \wedge g = \mathcal{O}(h) \ \Rightarrow \ f = \mathcal{O}(h) \quad$ für $\mathcal{O} \in \{O, \Omega, \Theta, o, \omega\}$:
    Die Landauschen Symbole sind also transitiv. Diese Beobachtung folgt ebenfalls unmittelbar aus den Definitionen.

11. $f = \Omega_\infty(g) \wedge g = \Omega_\infty(h) \ \not\Rightarrow \ f = \Omega_\infty(h)$:
    Wähle hierzu die Funktionen wie folgt $f(n) = 1 + (1 + (-1)^n)n$, $g(n) = n$, und $h(n) = n + (1 + (-1)^n)n^2 = n \cdot f(n)$.

12. $f_1 = \mathcal{O}(g) \ \wedge \ f_2 \in \mathcal{O}(g) \ \Rightarrow \ f_1 + f_2 \in \mathcal{O}(g) \quad$ für $\mathcal{O} \in \{O, \Omega, \Omega_\infty, \Theta, o, \omega\}$:
    Wir führen den Beweis nur für O. Nach Definition gilt für $i \in [1 : 2]$:

    $$\exists c_i \in \mathbb{R}.c_i > 0, n_i \in \mathbb{N} : \ \forall n \in \mathbb{N}.n \geq n_i : \ f_i(n) \leq c_i \cdot g(n).$$

    Also gilt mit $c := c_1 + c_2$ und $n_0 := \max\{n_1, n_2\}$:

    $$\forall n_0 \leq n \in \mathbb{N} : \ f_1(n) + f_2(n) \leq c \cdot g(n).$$

    Analoge Beziehungen gelten auch für Ω, Ω_∞, Θ, o und ω.

13. $f = \mathcal{O}(g) \ \wedge \ a \in \mathbb{R}_+^* \ \Rightarrow \ a \cdot f \in \mathcal{O}(g) \quad$ für $\mathcal{O} \in \{O, \Omega, \Omega_\infty, \Theta, o, \omega\}$:
    Wir führen den Beweis nur für O: Nach Definition gilt:

    $$\exists c \in \mathbb{R}.c > 0, n_0 \in \mathbb{N} : \ \forall n \in \mathbb{N}.n \geq n_0 : \ f(n) \leq c \cdot g(n).$$

    Dann gilt auch für alle $n \geq n_0$:

    $$a \cdot f(n) \leq (ac) \cdot g(n).$$

    Mit $c' = ac$ erhalten wir $a \cdot f \in O(g)$.

14. Wenn g nur endlich viele Nullstellen hat, dann gilt:

$$f = O(g) \quad \Leftrightarrow \quad \exists c \in \mathbb{R}.c > 0 : \quad \limsup_{n\to\infty} \frac{f(n)}{g(n)} \leq c.$$

15. Wenn g nur endlich viele Nullstellen hat, dann gilt:

$$f = \Omega(g) \quad \Leftrightarrow \quad \exists c \in \mathbb{R}.c > 0 : \quad \liminf_{n\to\infty} \frac{f(n)}{g(n)} \geq c.$$

16. $f = O(g) \not\Rightarrow f = (\Theta(g) \cup o(g))$:

    Wir beweisen nur den Fall, wenn $g(n) \neq 0$ für alle $n \in \mathbb{N}$. Dann definieren wir $f(n) := (1 + (-1)^n) \cdot g(n)$. Offensichtlich ist $f(n) \leq 2 \cdot g(n)$ und somit $f = O(g)$. Da $f(2k-1) = 0$ und $g(2k-1) > 0$ für alle $k \in \mathbb{N}$, gilt $f \neq \Omega(g)$. Auch ist $f(2k)/g(2k) = 2$ für alle $k \in \mathbb{N}$ und somit $f \neq o(g)$.

Nun führen wir noch ein paar abkürzende Sprechweisen ein, um das Wachstum von Funktionen zu beschreiben. Sei hierzu $f : \mathbb{N} \to \mathbb{N}$. Wir sagen f

- ist *konstant*, wenn $f(n) = \Theta(1)$;
- wächst *logarithmisch*, wenn $f(n) = O(\log(n))$;
- wächst *polylogarithmisch*, wenn $f(n) = O(\log^k(n))$ für ein $k \in \mathbb{N}$;
- wächst *linear*, wenn $f(n) = O(n)$;
- wächst *quadratisch*, wenn $f(n) = O(n^2)$;
- wächst *polynomiell*, wenn $f(n) = O(n^k)$ für ein $k \in \mathbb{N}$;
- wächst *superpolynomiell*, wenn $f(n) = \omega(n^k)$ für alle $k \in \mathbb{N}$;
- wächst *subexponentiell*, wenn $f(n) = o(2^{cn})$ für alle $0 < c \in \mathbb{R}$;
- wächst *exponentiell*, wenn $f(n) = O(2^{cn})$ für ein $0 < c \in \mathbb{R}$.

In der Regel fordert man für die obigen Sprechweisen auch, dass neben O auch Ω bzw. zumindest Ω_∞ gelten soll. Dass heißt, dass eine Funktion f logarithmisch wächst, wenn $f(n) = O(\log(n))$ und $f(n) = \Omega(\log(n))$ bzw. $f(n) = \Omega_\infty(\log(n))$ gilt.

Man beachte ferner, dass für Abschätzung mit den Landauschen Symbolen eine als konstant bezeichnete Funktion nicht im mathematischen Sinne konstant sein muss. Sie ist nur bis auf konstante Faktoren durch eine im mathematischen Sinne konstante Funktion von oben und unten beschränkt. Im Allgemeinen sind nach dieser

Namenskonvention auch lineare, exponentielle bzw. die anderen angegebenen Funktionen nicht notwendigerweise monoton wachsend.

Mit den Landauschen Symbolen sollte man, wie bereits erwähnt, sehr bewusst umgehen. Wir betrachten den folgenden „Induktionsbeweis“ für die Gaußsche Summe:

Beh. Für $S(n) := \sum_{i=1}^{n} i$ gilt $S(n) = O(n)$.

Induktionsanfang ($n = 1$): Offensichtlich ist $S(1) = 1 = O(1)$ und somit folgt die Behauptung.

Induktionsschritt ($n - 1 \to n$): Es gilt:

$$\begin{aligned} S(n) &= n + \sum_{i=1}^{n-1} i \\ &= n + S(n-1) \\ &\quad \text{Nach Induktionsvoraussetzung } S(n-1) \in O(n-1) \\ &= n + O(n-1) \\ &\quad \text{Offensichtlich ist } n \in O(n) \text{ und } O(n-1) \subseteq O(n) \\ &= O(n) + O(n) \\ &\quad \text{Wie wir gesehen haben, gilt für } f_i = O(g), \text{ dass } f_1 + f_2 = O(g) \\ &= O(n). \end{aligned}$$

Dieses Ergebnis ist offensichtlich falsch. Der Leser möge sich die Mühe machen, die Fehler im Beweis zu finden.

1.4.2 Euler-Maclaurin Summation (+)

In diesem Abschnitt wollen wir uns mit dem Ausrechnen von Summen beschäftigen, die ja, wie wir bereits gesehen haben, bei der Analyse von Algorithmen oft vorkommen. Ein einfaches Mittel ist, die Summe auf die Berechnung von Integralen zurückzuführen.

Wir betrachten dazu erst ein einfaches Beispiel, namentlich die harmonischen Zahlen.

Definition 1.26 *Mit $H_n := \sum_{i=1}^{n} \frac{1}{i}$ wird die n-te* harmonische Zahl *bezeichnet.*

Diese Summe kann man offensichtlich auch als Ober- bzw. Untersumme für das Integral über $1/x$ verstehen, siehe Abbildung 1.26

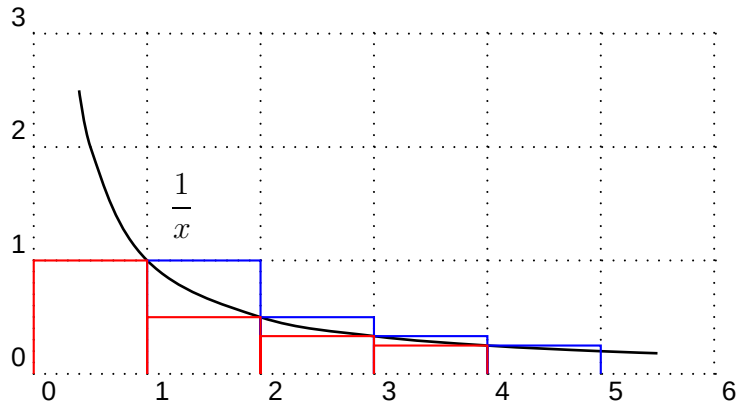


Abbildung 1.26: Skizze: Harmonische Zahlen als Ober- bzw. Untersumme

Daraus sieht man sofort, dass

$$\int_1^{n+1} \frac{dx}{x} \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 + \int_1^n \frac{dx}{x}.$$

Da $\int \frac{dx}{x} = \ln(x)$ folgt:

$$\ln(n+1) = [\ln(x)]_1^{n+1} \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 + [\ln(x)]_1^n = 1 + \ln(n).$$

Also gilt $H_n \in [\ln(n+1) : \ln(n) + 1]$. Man kann auch eine genauere Abschätzung mithilfe von Integralen finden. Hierzu definieren wir zuerst die so genannten Bernoulli-Zahlen.

Definition 1.27 *Sei $m \in \mathbb{N}_0$, dann ist die* m-te Bernoulli-Zahl *definiert durch:*

$$B_m := -\frac{1}{m+1} \left[\sum_{j=0}^{m-1} \binom{m+1}{j} B_j - \delta_{0,m} \right],$$

wobei $\delta_{i,j}$ das Kronecker-Symbol *mit $\delta_{i,i} = 1$ und 0 sonst ist.*

Beispielsweise gilt $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$, $B_4 = -\frac{1}{30}$, $B_6 = \frac{1}{42}$ und $B_{2k+1} = 0$ für alle $k \in \mathbb{N}$. Basierend auf den Bernoulli-Zahlen können wir jetzt die so genannten Bernoulli-Polynome definieren.

Definition 1.28 *Sei $m \in \mathbb{N}_0$, dann ist das* m-te Bernoulli-Polynom *definiert durch:*

$$B_m(x) = \sum_{k=0}^{m} \binom{m}{k} B_k x^{m-k}.$$

Mit diesen eingeführten Hilfsmitteln können wir jetzt die Summationsformel von Euler und Maclaurin angeben.

Theorem 1.29 (Euler-Maclaurin-Summation) *Sei $m \in \mathbb{N}$ und f eine m-fach stetig differenzierbare Funktion, dann gilt für $a < b \in \mathbb{Z}$:*

$$\sum_{i=a}^{b-1} f(i) = \int_a^b f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!}\left[f^{(k-1)}(b) - f^{(k-1)}(a)\right] + R_m,$$

wobei

$$R_m = (-1)^{m+1} \int_a^b \frac{B_m(x - \lfloor x \rfloor)}{m!} f^{(m)}(x)\,dx.$$

Wir wenden die Euler-Maclaurin Summation nun für die harmonischen Zahlen an. Für die Summanden der harmonischen Zahl H_n gilt $f(i) = 1/i$. Um Index-Probleme zu vermeiden, berechnen wir den Wert für H_{n-1}. Wir halten zunächst einmal fest, dass $\int f(x)\,dx = \ln(x)$ und $f^{(m)}(x) = \frac{(-1)^m m!}{x^{m+1}}$. Wir erhalten also für $m = 1$:

$$\begin{aligned} H_{n-1} &= \int_1^n \frac{dx}{x} + \sum_{k=1}^{m} \frac{B_k}{k!}\left[f^{(k-1)}(n) - f^{(k-1)}(1)\right] + R_m \\ &= \ln(n) - \frac{1}{2}\left[\frac{1}{n} - 1\right] + \int_1^n \frac{B_1(x - \lfloor x \rfloor)}{-x^2}\,dx \end{aligned}$$

Man kann zeigen, dass das uneigentliche Integral $\int_1^\infty \frac{B_1(x-\lfloor x \rfloor)}{-x^2}\,dx$ existiert. Wir definieren daher $C := \frac{1}{2} + \int_1^\infty \frac{B_1(x-\lfloor x \rfloor)}{-x^2}\,dx$.

$$= \ln(n) + C - \frac{1}{2n} - \int_n^\infty \frac{B_1(x - \lfloor x \rfloor)}{-x^2}\,dx$$

Weiter kann man zeigen, dass $|B_1(x - \lfloor x \rfloor)| \leq 1$.

$$\begin{aligned} &= \ln(n) + C - \frac{1}{2n} + O\left(\int_n^\infty \frac{1}{x^2}\,dx\right) \\ &= \ln(n) + C - \frac{1}{2n} + O\left(\left[-\frac{1}{x}\right]_n^\infty\right) \\ &= \ln(n) + C + O\left(\frac{1}{n}\right). \end{aligned}$$

Die Konstante $C \approx 0.5772$ wird auch *Euler-Mascheronische-Konstante* genannt.

1.4.3 Diskrete Differentiation und Integration

Jetzt wollen wir noch eine andere Methode zur Bestimmung von Summen herleiten, die der Analysis eng angelehnt ist. Wir definieren hierzu zuerst die so genannten fallenden Faktoriellen.

Definition 1.30 *Sei $n \in \mathbb{Z}$ und $x \in \mathbb{R}$, dann ist die n-te* fallende Faktorielle *von x definiert als*

$$x^{\underline{n}} := \begin{cases} \prod_{i=0}^{k-1}(x-i) & \text{für} \quad k = n \geq 0 \\ \prod_{i=1}^{k} \frac{1}{x+i} & \text{für} \quad k = -n \geq 0 \end{cases}$$

Man beachte, dass nach Definition $n^{\underline{n}} = n!$ und $x^{\underline{0}} = 1$ gilt.

Für die fallenden Faktoriellen gelten ähnliche Gesetze wie für normale Potenzen.

Lemma 1.31 *Seien $n, m \in \mathbb{Z}$ und $x \in \mathbb{R}$, dann gilt*

$$x^{\underline{n+m}} = x^{\underline{n}}(x-n)^{\underline{m}} = x^{\underline{m}}(x-m)^{\underline{n}}.$$

Beachte, dass dieses Lemma auch für negative n oder m gilt. Der Beweis wird durch Fallunterscheidung, ob n bzw. m positiv oder negativ ist, geführt und sei dem Leser zur Übung überlassen

Wir können jetzt die diskrete Ableitung, das diskrete Analogon zur kontinuierlichen Ableitung ($f'(x) = \lim_{\substack{h \to 0 \\ h \neq 0}} \frac{f(x+h)-f(x)}{h}$), definieren.

Definition 1.32 *Sei $f : \mathbb{Z} \to \mathbb{R}$, dann ist die* diskrete Ableitung Δf *von f definiert durch $(\Delta f)(x) := f(x+1) - f(x)$.*

08.05.19

Auch die diskrete Ableitung stellt einen linearen Operator wie im kontinuierlichen Fall dar.

Theorem 1.33 *Seien $f, g : \mathbb{Z} \to \mathbb{R}$ und $c \in \mathbb{R}$, dann gilt $\Delta(f+g) = \Delta f + \Delta g$ sowie $\Delta(c \cdot f) = c \cdot \Delta f$.*

Beweis: Es gilt für alle $x \in \mathbb{Z}$:

$$\begin{aligned}(\Delta(f+g))(x) &= (f+g)(x+1) - (f+g)(x)\\ &= [f(x+1)+g(x+1)] - [f(x)+g(x)]\\ &= f(x+1) - f(x) + g(x+1) - g(x)\\ &= (\Delta f)(x) + (\Delta g)(x)\\ &= (\Delta f + \Delta g)(x).\end{aligned}$$

Der Beweis für die skalare Multiplikation verläuft analog. ■

Für die fallende Faktorielle ist die diskrete Ableitung ähnlich zur normalen Potenz im kontinuierlichen Fall.

Lemma 1.34 *Es gilt für $n \in \mathbb{Z}$: $\Delta x^{\underline{n}} = n \cdot x^{\underline{n-1}}$.*

Beweis: Für $n = 0$ ist $x^{\underline{0}} = 1$ und es gilt: $\Delta x^{\underline{0}} = 1 - 1 = 0 = 0 \cdot x^{\underline{-1}}$.

Für $n > 0$ gilt:

$$\begin{aligned}\Delta x^{\underline{n}} &= (x+1)^{\underline{n}} - x^{\underline{n}}\\ &= (x+1)\cdots(x-n+2) - x\cdots(x-n+1)\\ &= [(x+1)-(x-n+1)]\,x\cdots(x-n+2)\\ &= n \cdot x^{\underline{n-1}}.\end{aligned}$$

Für $n < 0$ gilt mit $k = -n$:

$$\begin{aligned}\Delta x^{\underline{-k}} &= (x+1)^{\underline{-k}} - x^{\underline{-k}}\\ &= \frac{1}{(x+2)\cdots(x+k+1)} - \frac{1}{(x+1)\cdots(x+k)}\\ &= \frac{(x+1)-(x+k+1)}{(x+1)\cdots(x+k+1)}\\ &= \frac{-k}{(x+1)\cdots(x+k+1)}\\ &= (-k)\cdot x^{\underline{-(k+1)}}\\ &= (-k)\cdot x^{\underline{-k-1}}\\ &= n \cdot x^{\underline{n-1}}.\end{aligned}$$

Damit ist der Satz bewiesen. ■

Im diskreten Fall ist die Funktion 2^x diejenige, die mit ihrer diskreten Ableitung übereinstimmt.

Lemma 1.35 *Sei $c \in \mathbb{R}^*$, dann gilt $\Delta c^x = (c-1) \cdot c^x$ und insbesondere $\Delta 2^x = 2^x$.*

Nun geben wir die analoge Produktregel für die diskrete Differentiation an. Dazu benötigen wir zuerst noch eine Definition.

Definition 1.36 *Sei $f : \mathbb{Z} \to \mathbb{R}$ eine Funktion. Der* Shift-Operator *E ist definiert durch $(Ef)(x) = f(x+1)$ für alle $x \in \mathbb{Z}$.*

Theorem 1.37 *Seien $f, g : \mathbb{Z} \to \mathbb{R}$, dann gilt $\Delta(f \cdot g) = (\Delta f) \cdot (Eg) + f \cdot (\Delta g)$ sowie $\Delta(f \cdot g) = (\Delta f) \cdot g + (Ef) \cdot (\Delta g)$.*

Beweis: Es gilt für alle $x \in \mathbb{Z}$:

$$\begin{aligned}
\Delta(f \cdot g)(x) &= (f \cdot g)(x+1) - (f \cdot g)(x) \\
&= f(x+1) \cdot g(x+1) - f(x) \cdot g(x) \\
&= f(x+1) \cdot g(x+1) \underbrace{-f(x) \cdot g(x+1) + f(x) \cdot g(x+1)}_{=0} - f(x) \cdot g(x) \\
&= (f(x+1) - f(x)) \cdot g(x+1) + f(x) \cdot (g(x+1) - g(x)) \\
&= (\Delta f)(x) \cdot (Eg)(x) + f(x) \cdot (\Delta g)(x) \\
&= (((\Delta f) \cdot (Eg)) + (f \cdot (\Delta g)))(x)
\end{aligned}$$

Damit ist die erste Behauptung bewiesen, der Beweis der zweiten Behauptung sei dem Leser überlassen. ■

Nun können wir die diskrete Integration herleiten.

Definition 1.38 *Seien $f, g : \mathbb{Z} \to \mathbb{R}$. $g = \sum f$ heißt* diskrete Stammfunktion *von f, wenn $\Delta g = f$.*

Somit gilt für $n \neq -1$ bzw. $c \neq 1$:

$$\sum x^{\underline{n}} = \frac{x^{\underline{n+1}}}{n+1} \quad \text{bzw.} \quad \sum c^x = \frac{c^x}{c-1}.$$

Was uns momentan eventuell noch fehlt, ist das diskrete Analogon zur Stammfunktion $\ln(x)$ von $\frac{1}{x}$. Wir geben hier nur die diskrete Stammfunktion von $n^{\underline{-1}}$ als

Funktion auf $\mathbb{N}$ an und verweisen darauf, dass diese Funktion auch für komplexe Werte definiert werden kann.

Lemma 1.39 *Die diskrete Stammfunktion von $n^{\underline{-1}}$ ist $H(n) := H_n$.*

Beweis: Es gilt für alle $n \in \mathbb{N}$:

$$\begin{aligned}(\Delta H)(n) &= H(n+1) - H(n)\\ &= \sum_{i=1}^{n+1} \frac{1}{i} - \sum_{i=1}^{n} \frac{1}{i}\\ &= \frac{1}{n+1}\\ &= n^{\underline{-1}}.\end{aligned}$$

Damit ist H_n die diskrete Stammfunktion von $n^{\underline{-1}}$. ■

Wie im kontinuierlichen Fall kann man bestimmte Integrale definieren.

Theorem 1.40 *Seien $f, F : \mathbb{Z} \to \mathbb{R}$ und sei F eine diskrete Stammfunktion von f. Dann gilt*

$$\sum_{i=a}^{b} f(i) = F(b+1) - F(a) = \left[F(i)\right]_a^{b+1}.$$

Beweis: Da F eine diskrete Stammfunktion von f ist, gilt $\Delta F = f$ und somit für alle $i \in \mathbb{Z}$ die Beziehung $f(i) = F(i+1) - F(i)$. Damit erhalten wir:

$$\sum_{i=a}^{b} f(i) \;=\; \sum_{i=a}^{b} (F(i+1) - F(i)) \;=\; F(b+1) - F(a).$$

Damit ist der Beweis abgeschlossen. ■

Wir halten jetzt noch die eben verwendete Notation fest, die uns bereits aus der Integralrechnung der Analysis bekannt ist.

Notation 1.41 *Für $a < b \in \mathbb{R}$ bezeichnet $\left[f(x)\right]_a^b := f(b) - f(a)$.*

Wir wollen jetzt als Anwendung noch einen einfachen Beweis für die Summe $\sum_{i=1}^{n} i^2$ herleiten.

$$\begin{aligned}
\sum_{i=1}^{n} i^2 &= \sum_{i=1}^{n} i(i-1+1) \\
&= \sum_{i=1}^{n} i(i-1) + \sum_{i=1}^{n} i \\
&= \sum_{i=1}^{n} i^{\underline{2}} + \sum_{i=1}^{n} i^{\underline{1}} \\
&\quad \text{da } \tfrac{i^{\underline{3}}}{3} \text{ bzw. } \tfrac{i^{\underline{2}}}{2} \text{ die diskrete Stammfunktion von } i^{\underline{2}} \text{ bzw. } i^{\underline{1}} \text{ ist} \\
&= \left[\frac{i^{\underline{3}}}{3}\right]_1^{n+1} + \left[\frac{i^{\underline{2}}}{2}\right]_1^{n+1} \\
&= \frac{1}{3}\left[(n+1)^{\underline{3}} - 1^{\underline{3}}\right] + \frac{1}{2}\left[(n+1)^{\underline{2}} - 1^{\underline{2}}\right] \\
&= \frac{1}{3}\left[(n+1)n(n-1) - 0\right] + \frac{1}{2}\left[(n+1)n - 0\right] \\
&= \frac{2(n+1)n(n-1) + 3(n+1)n}{6} \\
&= \frac{(n+1)n(2n-2+3)}{6} \\
&= \frac{(n+1)n(2n+1)}{6}.
\end{aligned}$$

Wie man sieht, kann man mit einem analogen Apparat wie für die kontinuierliche Integration mit Hilfe der diskreten Differentiation und Integration auch viele Summen einfach direkt und exakt bestimmen.

Auch der aus dem kontinuierlichen bekannte Satz der partiellen Integration hat ein Analogon in der diskreten Integration.

Theorem 1.42 *Seien $f, g : \mathbb{Z} \to \mathbb{R}$, dann gilt $\sum (f \cdot \Delta g) = fg - \sum(\Delta f)(Eg)$.*

Beweis: Der Beweis folgt direkt aus den folgenden Ableitungen

$$\begin{aligned}
\Delta\left(\sum (f \cdot \Delta g)\right) &= f \cdot \Delta g \\
\Delta\left(fg - \sum(\Delta f)(Eg)\right) &= \Delta(fg) - (\Delta f)(Eg)
\end{aligned}$$

sowie der Produktregel für die diskrete Ableitung, da dann die beiden rechten Seiten der Gleichung gleich sind. ■

Für eine konkrete Summation impliziert dies

$$\sum_{i=a}^{b} (f(i) \cdot (\Delta g)(i)) = [f(i) \cdot g(i)]_a^{b+1} - \sum_{i=a}^{b} ((\Delta f)(i) \cdot (Eg)(i)).$$

Wir wollen jetzt den Wert der Summe $\sum_{i=1}^{n} i \cdot 2^i$ bestimmen. Dazu wählen wir zunächst $f(i) = i^{\underline{1}} = i$ und $(\Delta g)(i) = 2^i$ und verwenden die partielle Integration. Zuerst halten wir noch fest, dass hier $\Delta g = g$ gilt und somit $g(i) = 2^i$ ist. Wir erhalten dann:

$$\begin{aligned}
\sum_{i=1}^{n} i2^i &= \sum_{i=1}^{n} f(i) \cdot (\Delta g)(i) \\
&= [(f \cdot g)(i)]_1^{n+1} - \sum_{i=1}^{n} (\Delta f)(i) \cdot (Eg)(i) \\
&= \left[i2^i\right]_1^{n+1} - \sum_{i=1}^{n} 1 \cdot 2^{i+1} \\
&= \left[i2^i\right]_1^{n+1} - \sum_{i=2}^{n+1} 2^i \\
&= \left[i2^i\right]_1^{n+1} - \left[2^i\right]_2^{n+2} \\
&= ((n+1)2^{n+1} - 2) - (2^{n+2} - 4) \\
&= (n+1)2^{n+1} - 2 - 2 \cdot 2^{n+1} + 4 \\
&= (n-1)2^{n+1} + 2.
\end{aligned}$$

Damit haben wir das folgende Lemma bewiesen.

Lemma 1.43 *Für $n \in \mathbb{N}_0$ gilt $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$.*

In Abbildung 1.27 sind noch einmal alle uns bekannten diskreten Stammfunktionen bzw. Ableitungen zusammengefasst. Für eine vertiefende Lektüre verweisen wir auf das Buch von Graham, Knuth und Patashnik.

Wir halten an dieser Stelle noch fest, dass es kein direktes Analogon für die Kettenregel gibt.

$f = \sum g$	$\Delta f = g$
$x^{\underline{n}}$	$n \cdot x^{\underline{n-1}}$
$\frac{x^{\underline{n+1}}}{n+1}$	$x^{\underline{n}}$
$H(x)$	$x^{\underline{-1}}$
2^x	2^x
c^x	$(c-1)c^x$
$\frac{c^x}{c-1}$	c^x
$c \cdot f$	$c \cdot (\Delta f)$
$f + g$	$\Delta f + \Delta g$
$f \cdot g$	$(\Delta f) \cdot (Eg) + f \cdot (\Delta g)$

Abbildung 1.27: Tabelle: Diskrete Ableitungen und Stammfunktionen

1.4.4 Lineare Rekursionsgleichungen

Wir wollen zunächst eine allgemeine Lösung für einfache, so genannte lineare homogene Rekursionsgleichungen konstanter Ordnung angeben. Zunächst definieren wir, was wir unter einer linearen Rekursionsgleichung konstanter Ordnung verstehen.

Definition 1.44 *Eine* Rekursionsgleichung *mit Koeffizienten $c_i \in \mathbb{C}$ für $i \in [0:k]$ der Form*

$$c_0 \cdot a_n + c_1 \cdot a_{n-1} + \cdots + c_k \cdot a_{n-k} = \sum_{i=0}^{k} c_i \cdot a_{n-i} = b_k \qquad \text{für} \quad n \geq k$$

mit $c_0 \neq 0 \neq c_k$ und den Anfangsbedingungen $a_i = b_i \in \mathbb{C}$ für $i \in [0:k-1]$ heißt lineare Rekursionsgleichung k-ter Ordnung*. Gilt $b_k = 0$, so heißt die Rekursionsgleichung* homogen*, ansonsten heißt sie* inhomogen*.*

14.05.19

Beachte, dass nach der vorherigen Definition der Wert b_k nicht zu den Anfangsbedingungen gehört, wobei b_k hier nicht notwendigerweise eine Konstante sein muss, sondern auch von n abhängen kann. Andererseits ist k hier eine feste Konstante. Die Rekursionsgleichung für den Divide-and-Conquer-Algorithmus ist zwar auch eine lineare Rekursionsgleichung, aber keine konstanter (bzw. k-ter) Ordnung.

Für die allgemeine Lösung benötigen wir noch die Definition eines charakteristischen Polynoms.

Definition 1.45 *Sei $\sum_{i=0}^{k} c_i \cdot a_{n-i} = 0$ für $n \geq k$ eine homogene lineare Rekursionsgleichung k-ter Ordnung. Dann ist*

$$p(x) = \sum_{i=0}^{k} c_i \cdot x^{k-i} = c_0 \cdot x^k + c_1 \cdot x^{k-1} + \cdots + c_k$$

das zugehörige charakteristische Polynom.

Wir erinnern hier an den Fundamentalsatz aus der Algebra, nachdem jedes Polynom k-ten Grades genau k Nullstellen über den komplexen Zahlen $\mathbb{C}$ besitzt (Vielfachheiten mitgezählt).

Nun können wir die allgemeine Lösung für eine homogene lineare Rekursionsgleichung angeben.

Theorem 1.46 *Sei $\sum_{i=0}^{k} c_i \cdot a_{n-i} = 0$ für $n \geq k$ eine homogene lineare Rekursionsgleichung mit den Anfangsbedingungen $a_i = b_i$ für $i \in [0 : k-1]$ und sei $p(x)$ das zugehörige charakteristische Polynom. Seien weiter $(x_1, \ldots, x_m)$ die paarweise verschiedenen komplexen Nullstellen von p mit Vielfachheiten $(k_1, \ldots, k_m)$ (also mit $\sum_{i=1}^{m} k_i = n$). Dann ist die Lösung dieser Rekursionsgleichung die Folge $(a_n)_{n \in \mathbb{N}_0}$ mit*

$$a_n = \sum_{i=1}^{m} \sum_{j=0}^{k_i - 1} \alpha_{i,j} \cdot n^{\underline{j}} \cdot x_i^n.$$

Die genauen Werte der Parameter $\alpha_{i,j} \in \mathbb{C}$ mit $i \in [1 : m]$ und $j \in [0 : k_i - 1]$ hängen dabei von den Anfangsbedingungen $(a_0, \ldots, a_{k-1}) = (b_0, \ldots, b_{k-1})$ ab.

Als Beispiel wollen wir die Fibonacci-Zahlen betrachten:

$$f_n - f_{n-1} - f_{n-2} = 0 \qquad \text{und} \qquad f_0 = 0, \quad f_1 = 1.$$

Es handelt sich also um eine homogene lineare Rekursionsgleichung zweiter Ordnung mit dem charakteristischen Polynom $p(x) = x^2 - x - 1$. Die zugehörigen Nullstellen sind $x_{1,2} = \frac{1 \pm \sqrt{5}}{2}$. Hierbei ist $\varphi := \frac{1+\sqrt{5}}{2}$ als *goldener Schnitt* bekannt. Beachte hierbei, dass $\frac{1-\sqrt{5}}{2} = -\frac{1}{\varphi}$ gilt! Also lautet die Lösung

$$f_n = a \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + b \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n = a \cdot \varphi^n + b \cdot \left(\frac{-1}{\varphi}\right)^n,$$

wobei die Koeffizienten $a := \alpha_{1,0}$ und $b := \alpha_{2,0}$ von den Anfangsbedingungen abhängen und somit das folgende lineare Gleichungssystem lösen müssen:

$$\begin{aligned} 0 = f_0 &= a \cdot \left(\frac{1+\sqrt{5}}{2}\right)^0 + b \cdot \left(\frac{1-\sqrt{5}}{2}\right)^0 = a + b, \\ 1 = f_1 &= a \cdot \left(\frac{1+\sqrt{5}}{2}\right)^1 + b \cdot \left(\frac{1-\sqrt{5}}{2}\right)^1. \end{aligned}$$

Aus der ersten Gleichung folgt $b = -a$ und somit lautet die Lösung hierfür $a = \frac{1}{\sqrt{5}}$ und $b = -\frac{1}{\sqrt{5}}$. Somit gilt die bekannte Formel:

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \frac{\varphi^n}{\sqrt{5}} - \frac{\left(-\frac{1}{\varphi}\right)^n}{\sqrt{5}}.$$

Beachte, dass in der Lösung irrationale Zahlen vorkommen, obwohl $f_n \in \mathbb{N}_0$.

Können wir auch inhomogene lineare Rekursionsgleichungen derart lösen? Ja, wir können inhomogene lineare Rekursionsgleichungen nämlich homogen machen. Dazu betrachten wir das Beispiel $a_n - a_{n-1} = n - 1$ und $a_1 = 0$ wie beim Selectionsort. Wir betrachten diese Rekursionsgleichung für n und $n-1$ und ziehen die letzte Gleichung von der ersten ab:

$$\begin{aligned} a_n - a_{n-1} &= n - 1 \\ a_{n-1} - a_{n-2} &= n - 2 \end{aligned}$$

Wir erhalten $a_n - 2a_{n-1} + a_{n-2} = 1$. Diese Gleichung ist jedoch immer noch nicht homogen, aber statt des inhomogenen Terms $n-1$ haben wir jetzt eine Konstante. Wir können diese Idee also noch einmal anwenden:

$$\begin{aligned} a_n - 2a_{n-1} + a_{n-2} &= 1 \\ a_{n-1} - 2a_{n-2} + a_{n-3} &= 1 \end{aligned}$$

Wir erhalten $a_n - 3a_{n-1} + 3a_{n-2} - a_{n-3} = 0$. Wie wir sehen, haben wir nun eine homogene lineare Rekursionsgleichung dritter Ordnung mit den Anfangsbedingungen $a_1 = 0$, $a_2 = 1$, $a_3 = 3$ (diese lassen sich aus der ersten Rekursionsgleichung explizit bestimmen) erhalten.

Das charakteristische Polynom ist daher $p(x) = x^3 - 3x^2 + 3x - 1 = (x-1)^3$. Somit ist 1 eine dreifache Nullstelle des charakteristischen Polynoms und wir erhalten mit $\alpha_j := \alpha_{1,j}$, $x_1 = 1$ und $k_1 = 3$:

$$a_n = \sum_{j=0}^{2} \alpha_j \cdot n^{\underline{j}} \cdot 1^n = \sum_{j=0}^{2} \alpha_j \cdot n^{\underline{j}}$$

Die Koeffizienten α_0, α_1 und α_2 lassen sich aus den Anfangsbedingungen bestimmen:

$$\begin{aligned} 0 &= a_1 &&= \alpha_0 + \alpha_1 \cdot 1 + \alpha_2 \cdot 1^{\underline{2}} &&= \alpha_0 + \alpha_1, \\ 1 &= a_2 &&= \alpha_0 + \alpha_1 \cdot 2 + \alpha_2 \cdot 2^{\underline{2}} &&= \alpha_0 + 2\alpha_1 + 2\alpha_2, \\ 3 &= a_3 &&= \alpha_0 + \alpha_1 \cdot 3 + \alpha_2 \cdot 3^{\underline{2}} &&= \alpha_0 + 3\alpha_1 + 6\alpha_2. \end{aligned}$$

Somit ist $\alpha_0 = -\alpha_1$ und wir erhalten aus den letzten beiden Gleichungen.

$$\begin{aligned} 1 &= \alpha_1 + 2\alpha_2, \\ 3 &= 2\alpha_1 + 6\alpha_2. \end{aligned}$$

Nun multiplizieren wir die erste Gleichung mit -2 und addieren dann beide Gleichungen, dann erhalten wir $1 = 2\alpha_2$. Somit ist $\alpha_2 = \frac{1}{2}$ und daher $\alpha_1 = 1 - 2\alpha_2 = 0$ sowie $\alpha_0 = -\alpha_1 = 0$. Die Lösung ist also

$$a_n = \frac{1}{2} \cdot n^{\underline{2}} = \frac{1}{2} \cdot n(n-1) = \binom{n}{2}.$$

1.4.5 Weiteres Beispiel einer Homogenisierung (*)

Diesen Trick der Homogenisierung können wir auch auf die lineare Rekursionsgleichung (nichtkonstanter Ordnung) anwenden, die wir bei der Average-Case-Analyse von Quicksort erhalten haben (siehe auch die Herleitung von Gleichung 1.1 auf Seite 79. Im Folgenden schreiben wir kurz $\bar{A}(n)$ für $A_{\mathrm{Q}}^{\mathrm{ac}}(n)$. Wir wollen also folgende Rekursionsgleichung lösen:

$$\bar{A}(n) = (n-1) + \frac{2}{n} \sum_{k=1}^{n-1} \bar{A}(k).$$

wobei $\bar{A}(1) = 0$. Es gilt nach Multiplikation mit n

$$n\bar{A}(n) = n(n-1) + 2 \sum_{k=1}^{n-1} \bar{A}(k).$$

Wir betrachten diese Gleichung jetzt für $n-1$:

$$(n-1)\bar{A}(n-1) = (n-1)(n-2) + 2 \sum_{k=1}^{n-2} \bar{A}(k).$$

Ziehen wir die letzte von der vorletzten Gleichung ab, so erhalten wir:

$$n\bar{A}(n) - (n-1)\bar{A}(n-1) = n(n-1) - (n-1)(n-2) + 2\bar{A}(n-1).$$

Somit gilt:

$$n\bar{A}(n) = 2(n-1) + (n+1)\bar{A}(n-1).$$

Nach Division durch $n(n+1)$ erhalten wir

$$\frac{\bar{A}(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{\bar{A}(n-1)}{n}.$$

Diese Gleichung können wir jetzt wie schon so oft herunteriterieren und erhalten:

$$\frac{\bar{A}(n)}{n+1} = \sum_{i=2}^{n} \frac{2(i-1)}{i(i+1)}.$$

Durch Nachrechnen stellt man fest, dass $\frac{2(i-1)}{i(i+1)} = \frac{4}{i+1} - \frac{2}{i}$ gilt. Somit erhalten wir:

$$\begin{aligned}
\frac{\bar{A}(n)}{n+1} &= \sum_{i=2}^{n} \frac{2(i-1)}{i(i+1)} \\
&= \sum_{i=2}^{n} \frac{4}{i+1} - \sum_{i=2}^{n} \frac{2}{i} \\
&= \sum_{i=3}^{n+1} \frac{4}{i} - \sum_{i=2}^{n} \frac{2}{i} \\
&= 4\left(H_n + \frac{1}{n+1} - \frac{3}{2}\right) - 2(H_n - 1) \\
&= 2H_n - 4 + \frac{4}{n+1}
\end{aligned}$$

Also gilt $\bar{A}(n) = (n+1)(2H_n - 4 + \frac{4}{n+1})$. Da $H_n = \ln(n) + O(1)$, gilt $\bar{A} = \Theta(n \log(n))$.

Theorem 1.47 *Quicksort benötigt zum Sortieren von n Elementen einer total geordneten Menge im average-case $\Theta(n \log(n))$ Vergleiche.*

1.4.6 Umgang mit Gauß-Klammern

Kommen wir jetzt noch einmal zur Rekursionsgleichung von Mergesort bzw. zu der Rekursionsgleichung für den Divide-and-Conquer-Ansatz für das Maximal Scoring Subsequence Problem zurück. Beide haben in etwa die folgende Form:

$$A(n) = \begin{cases} 0 & \text{falls} \quad n = 1, \\ A(\lfloor n/2 \rfloor) + A(\lceil n/2 \rceil) + (n-1) & \text{falls} \quad n > 1. \end{cases}$$

Die Gauß-Klammern lassen sich auf mehrere Arten entfernen. Eine etwas trickreichere Variante, die viel Gespür erfordert, ist die folgende. Wir betrachten den Ausdruck $B(n) = A(n+1) - A(n)$ für $n > 1$ und $B(1) = 1$. Damit gilt für $n > 1$:

$$\begin{aligned}
B(n) &= A(n+1) - A(n) \\
&= A\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + A\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + n - \left(A\left(\left\lceil \frac{n}{2} \right\rceil\right) + A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + (n-1)\right) \\
&\quad \text{da für alle } n \in \mathbb{N} \text{ gilt: } \left\lfloor \tfrac{n+1}{2} \right\rfloor = \left\lceil \tfrac{n}{2} \right\rceil \\
&= A\left(\left\lceil \frac{n+1}{2} \right\rceil\right) - A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&\quad \text{da für alle } n \in \mathbb{N} \text{ gilt: } \left\lceil \tfrac{n+1}{2} \right\rceil = \left\lfloor \tfrac{n}{2} \right\rfloor + 1 \\
&= A\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) - A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \\
&= B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.
\end{aligned}$$

Die Rekursion $B(n) = B(\lfloor n/2 \rfloor) + 1$ mit $B(1) = 1$ hat eine einfache Interpretation, sie zählt die Bits in der Binärdarstellung von n ohne führende Nullen, also gilt $B(n) = \lceil \log(n+1) \rceil = \lfloor \log(n) \rfloor + 1$. Weiter gilt dann (mit $A(1) = 0$):

$$\begin{aligned}
A(n) &= A(n) - A(1) \\
&= \sum_{i=1}^{n-1} (A(i+1) - A(i)) \\
&= \sum_{i=1}^{n-1} B(i)
\end{aligned}$$

Wir zählen jetzt die Anzahl der Bits in der Binärdarstellung aller Zahlen im Intervall $[1 : n-1]$ anders. Dazu zählen wir für jede feste Anzahl von Bits j die Anzahl der verschiedenen Zahlen, deren Binärdarstellung genau j Bits benötigt. Dies sind 2^{j-1}, da das erste Bit ja immer 1 sein muss. Davon müssen wir noch die Anzahl von Bits abziehen, deren Wert im Intervall $[n : 2^{\lceil \log(n) \rceil} - 1]$ liegt. Somit erhalten wir:

$$\begin{aligned}
&= \sum_{j=1}^{\lceil \log(n) \rceil} j2^{j-1} - \left(2^{\lceil \log(n) \rceil} - 1 - (n-1)\right) \cdot \lceil \log(n) \rceil \\
&= \frac{1}{2} \sum_{j=1}^{\lceil \log(n) \rceil} j2^{j} - \left(2^{\lceil \log(n) \rceil} - n\right) \cdot \lceil \log(n) \rceil \\
&\quad \text{mit } \textstyle\sum_{j=1}^{k} j2^j = (k-1)2^{k+1} + 2 \text{ (siehe Lemma 1.43 auf Seite 93)} \\
&= \frac{1}{2}\left((\lceil \log(n) \rceil - 1)2^{\lceil \log(n) \rceil + 1} + 2\right) - \left(2^{\lceil \log(n) \rceil} - n\right) \cdot \lceil \log(n) \rceil
\end{aligned}$$

$$
\begin{aligned}
&= (\lceil \log(n) \rceil - 1) 2^{\lceil \log(n) \rceil} + 1 - \left(2^{\lceil \log(n) \rceil} - n\right) \cdot \lceil \log(n) \rceil \\
&= n \lceil \log(n) \rceil + 1 - 2^{\lceil \log(n) \rceil}.
\end{aligned}
$$

Theorem 1.48 *Mergesort bzw. der Divide-and-Conquer-Ansatz für das Maximal Scoring Subsequence Problem benötigt zum Sortieren von n Elementen einer total geordneten Menge bzw. für die Bestimmung einer maximalen Teilfolge im worst-case maximal $n\lceil \log(n) \rceil + 1 - 2^{\lceil \log(n) \rceil}$ Vergleiche bzw. Additionen von Array-Elementen.*

15.05.19

Wir wollen jetzt noch eine andere, einfacher anzuwendende, aber nicht ganz so genaue Methode kennen lernen. Die Lösung der Rekursionsgleichung wird durch die Gauß-Klammern doch erheblich erschwert. Aus diesem Grunde betrachten wir die Rekursionsgleichung nicht mehr über $\mathbb{N}$, sondern über $\mathbb{R}$ und nutzen dabei aus, dass $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq \frac{n+1}{2}$. Wir betrachten daher die folgende Rekursionsgleichung:

$$
\begin{aligned}
W(x) &= 2 \cdot W\left(\frac{x+1}{2}\right) + x - 1 && \text{für } x \in [2, \infty), \\
W(x) &= 0 && \text{für } x \in [1, 2).
\end{aligned}
$$

Hierbei bezeichne wie in der Analysis üblich $[a, b) = \{x \in \mathbb{R} : a \leq x < b\}$. Wir zeigen zunächst einmal, dass $A(n) \leq W(n)$ ist und es somit zulässig ist, sich auf W zu beschränken.

Lemma 1.49 *Für alle $n \in \mathbb{N}$ und $\varepsilon \geq 0$ gilt: $A(n) \leq W(n + \varepsilon)$.*

Beweis: Wir beweisen die Behauptung mit vollständiger Induktion über n:

Induktionsanfang ($n = 1$): Wie man leicht sieht, ist $W(x) \geq 0$ für alle $x \geq 1$. Damit ist $W(1 + \varepsilon) \geq 0 = A(1)$ für alle $\varepsilon \geq 0$.

Induktionsschritt ($\rightarrow n$): Wir unterscheiden zwei Fälle, je nachdem, ob n gerade oder ungerade ist.

Fall 1: Sei also n zunächst gerade und somit $n \geq 2$:

$$
\begin{aligned}
W(n + \varepsilon) &= 2 \cdot W\left(\frac{n + \varepsilon + 1}{2}\right) + n + \varepsilon - 1 \\
&= 2 \cdot W\left(\frac{n}{2} + \frac{\varepsilon + 1}{2}\right) + n + \varepsilon - 1 \\
&\quad \text{da } \tfrac{n}{2} \leq n - 1 \text{ und } \tfrac{\varepsilon+1}{2} \geq 0 \\
&\overset{\text{I.V.}}{\geq} 2 \cdot A\left(\frac{n}{2}\right) + n + \varepsilon - 1
\end{aligned}
$$

$$
\begin{aligned}
& \text{da } n \text{ gerade und somit } \tfrac{n}{2} = \lfloor \tfrac{n}{2} \rfloor = \lceil \tfrac{n}{2} \rceil \\
= \;& A\left(\left\lceil \frac{n}{2} \right\rceil\right) + A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1 + \varepsilon \\
= \;& A(n) + \varepsilon \\
\geq \;& A(n).
\end{aligned}
$$

Fall 2: Sei also n nun ungerade und somit $n \geq 3$:

$$
\begin{aligned}
W(n+\varepsilon) &= 2 \cdot W\left(\frac{n+\varepsilon+1}{2}\right) + n + \varepsilon - 1 \\
&= W\left(\frac{n+1}{2} + \frac{\varepsilon}{2}\right) + W\left(\frac{n-1}{2} + \frac{2+\varepsilon}{2}\right) + n + \varepsilon - 1 \\
&\quad \text{da } \tfrac{n-1}{2} < \tfrac{n+1}{2} \leq n-1 \text{ und } \tfrac{2+\varepsilon}{2} > \tfrac{\varepsilon}{2} \geq 0 \\
&\overset{\text{I.V.}}{\geq} A\left(\frac{n+1}{2}\right) + A\left(\frac{n-1}{2}\right) + n + \varepsilon - 1 \\
&\quad \text{da } n \text{ ungerade und somit } \tfrac{n+1}{2} = \lceil \tfrac{n}{2} \rceil \text{ sowie } \tfrac{n-1}{2} = \lfloor \tfrac{n}{2} \rfloor \\
&= A\left(\left\lceil \frac{n}{2} \right\rceil\right) + A\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1 + \varepsilon \\
&= A(n) + \varepsilon \\
&\geq A(n).
\end{aligned}
$$

Damit haben wir in beiden Fällen den Induktionsschritt vollzogen und somit das Lemma bewiesen. ■

Wir müssen also nun nur noch die Rekursion von W lösen, um eine obere Schranke für die Anzahl der Vergleiche von Mergesort zu bekommen. Dazu definieren wir erst einmal die Funktion h und die Folge N_i:

$$
h : \mathbb{R} \to \mathbb{R} : \; x \mapsto \frac{x+1}{2}, \qquad\qquad N_i := \frac{n + 2^i - 1}{2^i} > 1.
$$

Wie man leicht sieht, gilt dann $N_0 = n$ und $h(N_i) = N_{i+1}$. Damit können wir nun leicht die Rekursionsgleichung von W durch Iteration lösen:

$$
\begin{aligned}
W(N_0) &= 2 \cdot W(N_1) + (N_0 - 1) \\
&\quad \text{durch Anwendung der Rekursionsgleichung} \\
&= 2(2 \cdot W(N_2) + (N_1 - 1)) + (N_0 - 1) \\
&= 2^2 \cdot W(N_2) + 2^1(N_1 - 1) + 2^0(N_0 - 1) \\
&\quad \text{nach } j \text{ Iterationen erhalten wir} \\
&= 2^j \cdot W(N_j) + \sum_{i=0}^{j-1} 2^i (N_i - 1)
\end{aligned}
$$

Es gilt $N_j \in [1,2)$, wenn $j > \log(n-1)$. Also setzen wir $j := \lceil \log(n) \rceil$.

$$\begin{aligned}
&= 2^{\lceil \log(n) \rceil} \cdot \underbrace{W(N_{\lceil \log(n) \rceil})}_{=0} + \sum_{i=0}^{\lceil \log(n) \rceil - 1} 2^i (N_i - 1) \\
&= \sum_{i=0}^{\lceil \log(n) \rceil - 1} 2^i \left(\frac{n + 2^i - 1}{2^i} - \frac{2^i}{2^i} \right) \\
&= \sum_{i=0}^{\lceil \log(n) \rceil - 1} (n-1) \\
&= (n-1)\lceil \log(n) \rceil \\
&= \Theta(n \log(n)).
\end{aligned}$$

Damit haben wir folgenden Satz erneut bewiesen.

Theorem 1.50 *Mergesort sortiert ein Feld der Länge n mit maximal $n \cdot \lceil \log(n) \rceil$ Vergleichen.*

Wie man leicht sieht, ist die exakte obere Schranke von Vergleichen von Mergesort aus Theorem 1.48 nur geringfügig kleiner als die in Theorem 1.50 hergeleitete wesentlich einfachere, gröbere obere Schranke für die Anzahl der ausgeführten Vergleiche. Das zeigt, dass oft eine einfachere Laufzeitanalyse völlig ausreichend ist.

1.4.7 Master-Theorem

Für Rekursionsgleichungen, die durch den Einsatz des Divide-and-Conquer entstehen, gibt es eine allgemeine Lösung, die in vielen Fällen die Anwendung findet. Dabei werden sowohl die Gauß-Klammern unterschlagen als auch die Problemgrößen bis auf Konstante abgeschätzt. Dann hilft das so genannte *Master-Theorem*:

Theorem 1.51 *Seien $a, b, d \in \mathbb{N}$ mit $b > 1$, sei $f(n)$ eine Funktion und sei $\mathcal{C}(n)$ definiert durch die Rekursionsgleichung $\mathcal{C}(n) = a \cdot \mathcal{C}(n/b) + f(n)$ für $n > 1$ und $\mathcal{C}(1) = d$. Dann gilt:*

$$\mathcal{C}(n) = \begin{cases} \Theta(n^{\log_b(a)}) & \text{falls } f(n) = O(n^{\log_b(a)-e}) \text{ für ein konstantes } e > 0 \\ \Theta(n^{\log_b(a)} \log(n)) & \text{falls } f(n) = \Theta(n^{\log_b(a)}) \\ \Theta(f(n)) & \text{falls } f(n) = \Omega(n^{\log_b(a)+e}) \text{ für ein konstantes } e > 0 \\ & \text{und } a \cdot f(n/b) \leq c \cdot f(n) \text{ für ein konstantes } c < 1 \end{cases}$$

Wenden wir das Master-Theorem auch noch einmal auf die Rekursionsgleichung von Mergesort an: $\mathcal{C}(n) = 2 \cdot \mathcal{C}(\frac{n}{2}) + n - 1$ für $n \geq 1$ und $\mathcal{C}(1) = 0$. Hier gilt $a = 2$, $b = 2$, $d = 0$ und $f(n) = n - 1$. Somit ist also $f(n) \in \Theta(n^{\log_2(2)})$ und die Laufzeit beträgt: $\Theta(n^{\log_2(2)} \cdot \log(n)) = \Theta(n \log(n))$.

Betrachten wir im Folgenden noch eine bzgl. f leicht modifizierte Rekursionsgleichung: $\mathcal{C}(n) = 2 \cdot \mathcal{C}(\frac{n}{2}) + \log(n)$ für $n \geq 1$ und $\mathcal{C}(1) = 0$. Hier gilt $a = 2$, $b = 2$, $d = 0$ und $f(n) = \log(n)$. Somit ist also $f(n) = \log(n) \in O(n^{\frac{1}{2}}) = O(n^{\log_2(2)-\frac{1}{2}})$ und somit $f(n) \in O(n^{\log_2(2)-e})$ für $e = \frac{1}{2}$ und die Laufzeit beträgt: $\Theta(n^{\log_2(2)}) = \Theta(n)$.

Betrachten wir im Folgenden noch eine bzgl. f anders modifizierte Rekursionsgleichung: $\mathcal{C}(n) = 2 \cdot \mathcal{C}(\frac{n}{2}) + n^2$ für $n \geq 1$ und $\mathcal{C}(1) = 0$. Hier gilt $a = 2$, $b = 2$, $d = 0$ und $f(n) = n^2$. Somit ist also $f(n) = n^2 \in \Omega(n^{\frac{3}{2}}) = \Omega(n^{\log_2(2)+\frac{1}{2}})$ und somit $f(n) \in \Omega(n^{\log_2(2)+e})$ für $e = \frac{1}{2}$. Weiter gilt

$$a \cdot f(n/b) = 2 \cdot \left(\frac{n}{2}\right)^2 = 2 \cdot \frac{n^2}{4} = \frac{n^2}{2} = \frac{1}{2} \cdot n^2 = c \cdot f(n)$$

mit $c = \frac{1}{2} < 1$ und somit auch $a \cdot f(n/b) \leq c \cdot f(n)$. Die Laufzeit beträgt daher: $\Theta(f(n)) = \Theta(n^2)$.

1.4.8 Erzeugende Funktionen

In diesem letzten Abschnitt wollen wir ein letztes Werkzeug zur Lösung von Rekursionsgleichungen vorstellen. Dazu bedienen wir uns der so genannten erzeugenden Funktionen, die sich zu einer gegebenen Folge definieren lassen.

Definition 1.52 *Sei $(a_n)_{n\in\mathbb{N}_0}$ eine komplexwertige unendliche Folge. Dann heißt*

$$A(z) = \sum_{n=0}^{\infty} a_n z^n$$

die zu a gehörige erzeugende Funktion.

Nehmen wir wieder an, wir wollen die folgende Rekursionsgleichung lösen:

$$a_n = \begin{cases} 0 & \text{für} \quad n \in [0:1], \\ a_{n-1} + (n-1) & \text{für} \quad n > 1. \end{cases}$$

Die hierzu gehörige erzeugende Funktion lässt sich wie folgt berechnen:

$$\begin{aligned}
A(z) &= \sum_{n=0}^{\infty} a_n z^n \\
&= a_0 + a_1 z + \sum_{n=2}^{\infty} a_n z^n \\
&= 0 + 0 + \sum_{n=2}^{\infty} (a_{n-1} + n - 1) z^n \\
&= \sum_{n=1}^{\infty} a_n z^{n+1} + \sum_{n=1}^{\infty} n z^{n+1} \\
&= z \cdot \sum_{n=1}^{\infty} a_n z^n + z^2 \sum_{n=1}^{\infty} n z^{n-1} \\
&\quad \text{da } a_0 = 0 \\
&= z \sum_{n=0}^{\infty} a_n z^n + z^2 \sum_{n=1}^{\infty} \frac{d}{dz} z^n \\
&= z \cdot A(z) + z^2 \cdot \frac{d}{dz} \sum_{n=1}^{\infty} z^n \\
&\quad \text{da für } |z| < 1 \text{ gilt } \textstyle\sum_{n=1}^{\infty} z^n = \frac{1}{1-z} \\
&= z \cdot A(z) + z^2 \cdot \frac{d}{dz} \left(\frac{1}{1-z} - 1 \right) \\
&= z \cdot A(z) + z^2 \cdot \frac{0 \cdot (1-z) - 1 \cdot (-1)}{(1-z)^2} \\
&= z \cdot A(z) + \frac{z^2}{(1-z)^2}.
\end{aligned}$$

Beachte, dass wir gerade eben auch $\sum_{n=0}^{\infty} n z^n = \frac{z}{(1-z)^2}$ für $|z| < 1$ bewiesen haben.

Für die erzeugende Funktion $A(z)$ gilt also die folgende Funktionalgleichung:

$$A(z) = z \cdot A(z) + \frac{z^2}{(1-z)^2}.$$

Diese Gleichung für die erzeugenden Funktion $A(z)$ lässt sich nun auflösen zu:

21.06.19

$$A(z) = \frac{z^2}{(1-z)^3}.$$

Wir wissen bereits wissen bzw. oben implizit gezeigt haben, gilt $\sum_{n=0}^{\infty} z^n = \frac{1}{1-z}$ und $\sum_{n=0}^{\infty} nz^n = \frac{z}{(1-z)^2}$ für $|z| < 1$ gilt. Also gilt:

$$\begin{aligned} A(z) &= z \cdot \frac{1}{1-z} \cdot \frac{z}{(1-z)^2} \\ &= z \cdot \left(\sum_{n=0}^{\infty} z^n\right) \cdot \left(\sum_{n=0}^{\infty} nz^n\right) \end{aligned}$$

Mit Hilfe des so genannten Cauchy-Produkts für absolut konvergente Reihen aus der Analysis: $\left(\sum_{n=0}^{\infty} a_n\right) \cdot \left(\sum_{n=0}^{\infty} b_n\right) = \sum_{n=0}^{\infty} \sum_{i=0}^{n} a_i \cdot b_{n-i}$ folgt weiter

$$\begin{aligned} &= z \cdot \sum_{n=0}^{\infty} \sum_{i=0}^{n} z^i \cdot (n-i) z^{n-i} \\ &= z \cdot \sum_{n=0}^{\infty} \sum_{i=0}^{n} z^{n-i} \cdot i z^i \\ &= z \sum_{n=0}^{\infty} \sum_{i=0}^{n} i z^n \\ &= z \sum_{n=0}^{\infty} z^n \sum_{i=0}^{n} i \\ &= \sum_{n=0}^{\infty} z^{n+1} \binom{n+1}{2} \\ &= \sum_{n=1}^{\infty} \binom{n}{2} z^n. \end{aligned}$$

Also gilt $a_n = \binom{n}{2}$. Somit haben wir wieder eine Herleitung eines expliziten Ausdrucks für diese Rekursionsgleichung für a_n gefunden.

In Abbildung 1.28 sind die wichtigsten Korrespondenzen zwischen Beziehungen von unendlichen Folgen und ihren zugehörigen erzeugenden Funktionen angegeben. Diese können oft beim Rechnen mit erzeugenden Funktionen helfen.

Hier und in der folgenden Abbildung seien $(a_n)_{n\in\mathbb{N}_0}$, $(b_n)_{n\in\mathbb{N}_0}$ und $(c_n)_{n\in\mathbb{N}_0}$ unendliche Folgen und $A(z)$, $B(z)$ und $C(z)$ ihre zugehörigen erzeugenden Funktionen mit positiven Konvergenzradien.

In Abbildung 1.29 sind die wichtigsten Reihenentwicklungen von bekannten Funktionen bzw. die erzeugenden Funktionen von elementaren Folgen angegeben.

Wir fassen die einzelnen Schritte beim Einsatz von erzeugenden Funktionen zur Lösung von Rekursionsgleichungen noch einmal schematisch zusammen:

1. Aufstellen der erzeugenden Funktion:

$$A(z) = \sum_{n=0}^{\infty} a_n z^n.$$

2. Einsetzen der Rekursionsgleichung und der Anfangsbedingungen:

$$a_n = \ldots$$

3. Umformen in eine Funktionalgleichung:

$$f(A(z), z) = 0.$$

4. Lösen der Funktionalgleichung in eine explizite Darstellung von $A(z)$:

$$A(z) = g(z).$$

5. Reihenentwicklung von $A(z)$ um 0 (Taylor-Reihe):

$$A(z) = \sum_{n=0}^{\infty} \left(\frac{\frac{d^n g}{dz^n}(0)}{n!} \right) z^n.$$

6. Ablesen der Koeffizienten a_n vor z^n aus der Reihenentwicklung von $A(z)$ liefert die explizite Darstellung von a_n:

$$\sum_{n=0}^{\infty} \left(\underbrace{\frac{\frac{d^n g}{dz^n}(0)}{n!}}_{a_n} \right) z^n.$$

7. Überprüfen der gefunden Lösung.

Folgen	Funktionen
$\forall n \in \mathbb{N}_0 : c_n = a_n \pm b_n$	$C(z) = A(z) \pm B(z)$
$\forall n \in \mathbb{N}_0 : c_n = c \cdot a_n$	$C(z) = c \cdot A(z)$
$\forall n \in \mathbb{N}_0 : c_n = \sum_{k=0}^{n} a_k \cdot b_{n-k}$	$C(z) = A(z) \cdot B(z)$
$\forall n \in \mathbb{N}_0 : c_{n+k} = a_n, \forall n \in [0 : k-1] : c_n = 0$	$C(z) = z^k \cdot A(z)$
$\forall n \in \mathbb{N}_0 : c_n = a_{n+k}$	$C(z) = \frac{A(z) - \sum_{n=0}^{k-1} a_n z^n}{z^k}$
$b_0 = \frac{1}{a_0}, \forall n \in \mathbb{N} : b_n = -\frac{1}{a_0} \sum_{k=1}^{n} a_k \cdot b_{n-k}$	$A(z) \cdot B(z) = 1$
$\forall n \in \mathbb{N}_0 : b_n = \sum_{k=0}^{n} a_k$	$B(z) = \frac{A(z)}{1-z}$
$\forall n \in \mathbb{N}_0 : b_n = n \cdot a_n$	$B(z) = z \cdot \frac{d}{dz} A(z)$
$\forall n \in \mathbb{N}_0 : b_n = (n+1) a_{n+1}$	$B(z) = \frac{d}{dz} A(z)$
$\forall n \in \mathbb{N}_0 : b_n = (n+1) a_n$	$B(z) = \frac{d}{dz} (z \cdot A(z))$
$b_0 = 0, \forall n \in \mathbb{N} : b_n = \frac{a_{n-1}}{n}$	$B(z) = \int_0^z A(t)\, dt$

Abbildung 1.28: Tabelle: Korrespondenzen zwischen Beziehungen von Folgen und deren zugehörigen erzeugenden Funktionen

a_n	a	$A(z)$	$A(z)$
$a_n = 1$	$(1, 1, 1, 1, \ldots)$	$\sum_{n=0}^{\infty} z^n$	$\frac{1}{1-z}$
$a_n = (-1)^n$	$(1, -1, 1, -1, \ldots)$	$\sum_{n=0}^{\infty} (-1)^n z^n$	$\frac{1}{1+z}$
$a_n = c^n$	$(1, c, c^2, c^3, \ldots)$	$\sum_{n=0}^{\infty} c^n z^n$	$\frac{1}{1-cz}$
$a_n = n$	$(0, 1, 2, 3, \ldots)$	$\sum_{n=0}^{\infty} n \cdot z^n$	$\frac{z}{(1-z)^2}$
$a_n = \binom{m}{n}$	$(1, m, \frac{m(m-1)}{2}, \ldots)$	$\sum_{n=0}^{\infty} \binom{m}{n} z^n$	$(1+z)^m$
$a_n = \binom{m+n}{n}$	$(1, m+1, \frac{(m+2)(m+1)}{2}, \ldots)$	$\sum_{n=0}^{\infty} \binom{m+n}{n} z^n$	$\frac{1}{(1-z)^{m+1}}$
$a_0 = 0,\ a_n = \frac{1}{n}$	$(0, 1, \frac{1}{2}, \frac{1}{3}, \ldots)$	$\sum_{n=1}^{\infty} \frac{z^n}{n}$	$-\ln(1-z)$
$a_n = \frac{1}{n!}$	$(0, 1, \frac{1}{2}, \frac{1}{6}, \ldots)$	$\sum_{n=0}^{\infty} \frac{z^n}{n!}$	e^z
$a_n = H_n$	$(0, 1, \frac{3}{2}, \frac{11}{6}, \ldots)$	$\sum_{n=0}^{\infty} H_n \cdot z^n$	$\frac{\ln(\frac{1}{1-z})}{1-z}$

Abbildung 1.29: Tabelle: Elementare Reihenentwicklungen bekannter Funktionen

Suchen in Texten 2

2.1 Grundlagen

In diesem Abschnitt wollen wir uns mit der Suche in Texten beschäftigen. Zunächst wiederholen wir einige elementare Definitionen.

Definition 2.1 *Ein* Alphabet *ist eine endliche Menge von Symbolen.*

Bsp.: $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, C, G, T\}$.

Definition 2.2 *Ein* Wort *über Σ ist endliche Folgen von Symbolen aus Σ.*

Wir werden im Folgenden Wörter manchmal von 0 und manchmal von 1 an indizieren, d.h. $w = w_0 \cdots w_{n-1}$ bzw. $w = w_1 \cdots w_n$, je nachdem, was im gegebenen Kontext praktischer ist.

Bsp.: $\Sigma = \{a, b\}$, dann ist $w = abba$ ein Wort über Σ.

Definition 2.3 *Die* Länge *eines Wortes w wird mit $|w|$ bezeichnet und entspricht der Anzahl der Symbole in w. Das Wort der Länge 0 heißt* leeres Wort *und wird mit ε bezeichnet.*

Notation 2.4 *Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Die Menge aller Wörter der Länge größer gleich 1 über Σ wird mit $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ bezeichnet. Die Menge aller Wörter über Σ der Länge k wird mit $\Sigma^k \subseteq \Sigma^*$ bezeichnet.*

Definition 2.5 *Sei $w = w_1 \cdots w_n \in \Sigma^n$ ein Wort der Länge n über Σ, dann heißt*

- w' Präfix *von w, wenn $w' = w_1 \cdots w_\ell$ mit $\ell \in [0 : n]$;*
- w' Suffix *von w, wenn $w' = w_\ell \cdots w_n$ mit $\ell \in [1 : n + 1]$;*
- w' Teilwort *von w, wenn $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$ oder $w' = \varepsilon$.*

Allgemein gilt $w_i \cdots w_j = \varepsilon$ für $i > j \in \mathbb{N}$.

Das leere Wort ist also Präfix, Suffix und Teilwort eines jeden Wortes über Σ.

2.2 Der Algorithmus von Knuth, Morris und Pratt

Dieser Abschnitt ist dem Suchen in Texten gewidmet, d.h. es soll festgestellt werden, ob ein gegebenes Suchwort s in einem gegebenen Text t enthalten ist oder nicht.

TEXTSUCHE

Eingabe: $s \in \Sigma^m$, $t \in \Sigma^n$.
Gesucht: Existiert ein $i \in [0 : n-m]$ mit $t_i \cdots t_{i+m-1} = s$.

2.2.1 Ein naiver Ansatz

Das Suchwort s wird Buchstabe für Buchstabe mit dem Text t verglichen. Stimmen zwei Buchstaben nicht überein ($\rightarrow$ Mismatch), so wird s um eine Position „nach rechts“ verschoben und der Vergleich von s mit t beginnt von neuem. Dieser Vorgang wird solange wiederholt, bis s in t gefunden wird oder bis klar ist, dass s in t nicht enthalten ist (siehe auch Abbildung 2.1).

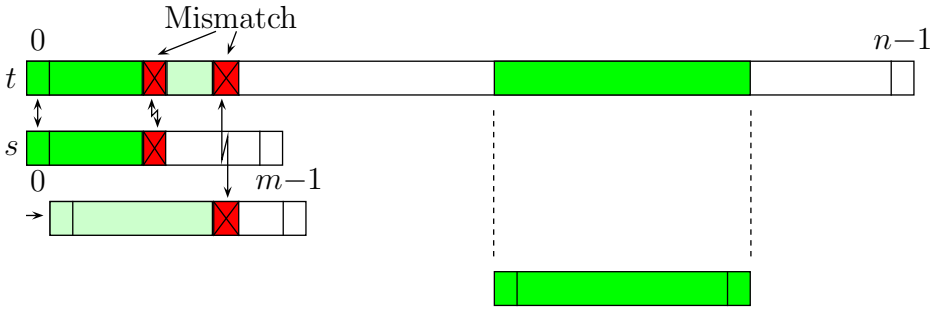


Abbildung 2.1: Skizze: Suchen mit der naiven Methode

In Abbildung 2.2 ist ein Beispiel für die Abarbeitung und in Abbildung 2.3 der naive Algorithmus im Pseudocode angegeben.

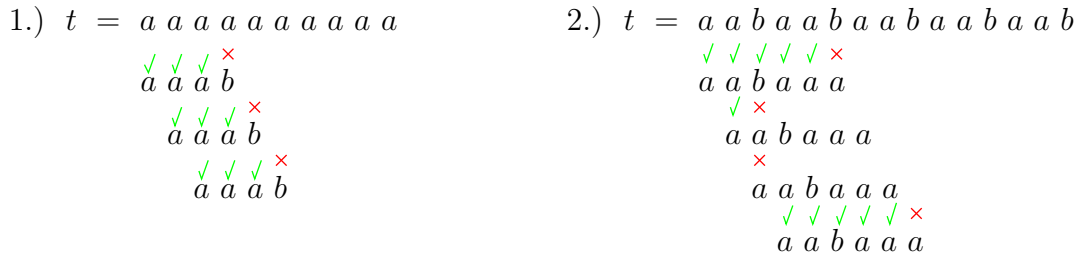


Abbildung 2.2: Beispiel: Suchen mit der naiven Methode

```
bool Naiv (char t[], int n, char s[], int m)
begin
    int i := 0, j := 0;
    while (i ≤ n − m) do
        while (t[i + j] = s[j]) do
            j++;
            if (j = m) then return TRUE;
        i++;
        j := 0;
    return FALSE;
end
```

Abbildung 2.3: Algorithmus: Die naive Methode

Definition 2.6 *Stimmen beim Vergleich zweier Zeichen diese nicht überein, so nennt man dies einen* Mismatch*, ansonsten einen* Match*.*

2.2.2 Laufzeitanalyse des naiven Algorithmus

Um die Laufzeit abzuschätzen, zählen wir die Vergleiche von Symbolen aus Σ. Die äußere Schleife wird $(n - m + 1)$-mal durchlaufen, die innere Schleife wird maximal m-mal durchlaufen. Also wird ein Vergleichstest auf Zeichen $((n - m + 1)m)$-mal ausgeführt. Somit ist die Laufzeit $O(nm)$. Das ist zu viel!

2.2.3 Eine bessere Idee

Ein Verbesserung ließe sich vermutlich dadurch erzielen, dass man die früheren erfolgreichen Vergleiche von zwei Zeichen ausnützt. Daraus resultiert die Idee, das Suchwort so weit nach rechts zu verschieben, dass in dem Bereich von t, in dem

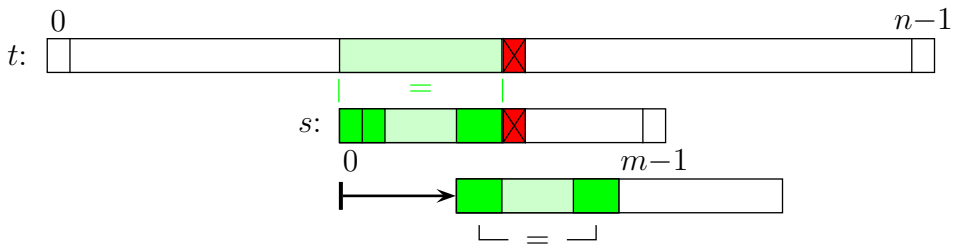


Abbildung 2.4: Skizze: Eine Idee für größere Shifts

bereits beim vorherigen Versuch erfolgreiche Zeichenvergleiche durchgeführt wurden, nun nach dem Verschieben auch wieder die Zeichen in diesem Bereich übereinstimmen (siehe auch die Skizze in Abbildung 2.4).

Um diese Idee genauer formalisieren zu können, benötigen wir noch einige grundlegende Definitionen.

Definition 2.7 *Ein Wort r heißt* Rand *eines Wortes w, wenn r sowohl Präfix als auch Suffix von w ist.*

Ein Rand r eines Wortes w heißt echter Rand, *wenn $r \neq w$.*

Ein Rand r eines Wortes w heißt eigentlicher Rand, *wenn r der längste echte Rand ist. Mit $\partial(s)$ wird der eigentliche Rand von s bezeichnet.*

Bemerkung: ε und w sind immer Ränder von w.

Beispiel: $a\ a\ b\ a\ a\ b\ a\ a$

Beachte: Ränder können sich überlappen!

Der eigentliche Rand von *aabaabaa* ist also *aabaa*. *aa* ist ein echter Rand, aber nicht der eigentliche Rand. ε besitzt als einziges Wort keinen echten Rand und somit auch keinen eigentlichen Rand

Definition 2.8 *Eine Verschiebung der Anfangsposition i des zu suchenden Wortes (d.h. eine Erhöhung des Index $i \to i'$) heißt* Shift.

Ein Shift der Form $i \to i'$ nach einem Mismatch der Zeichen an Position $i+j$ in t und j in s heißt zulässig, *wenn $s_0 \cdots s_{j-(i'-i)-1} = s_{i'-i} \cdots s_{j-1}$ gilt.*

Ein zulässiger Shift der Form $i \to i'$ heißt sicher, *wenn s nicht auch als Teilwort von t an einer Position $k \in [i+1 : i'-1]$ vorkommt, d.h. es gilt $s \neq t_k \cdots t_{k+m-1}$ für alle $k \in [i+1 : i'-1]$.*

Wir definieren:

$$border[j] = \begin{cases} -1 & \text{für } j = 0, \\ |\partial(s_0 \cdots s_{j-1})| & \text{für } j \geq 1. \end{cases}$$

Lemma 2.9 *Gilt $s_k = t_{i+k}$ für alle $k \in [0 : j-1]$ und $s_j \neq t_{i+j}$, dann ist der Shift $i \to i + j - border[j]$ sicher.*

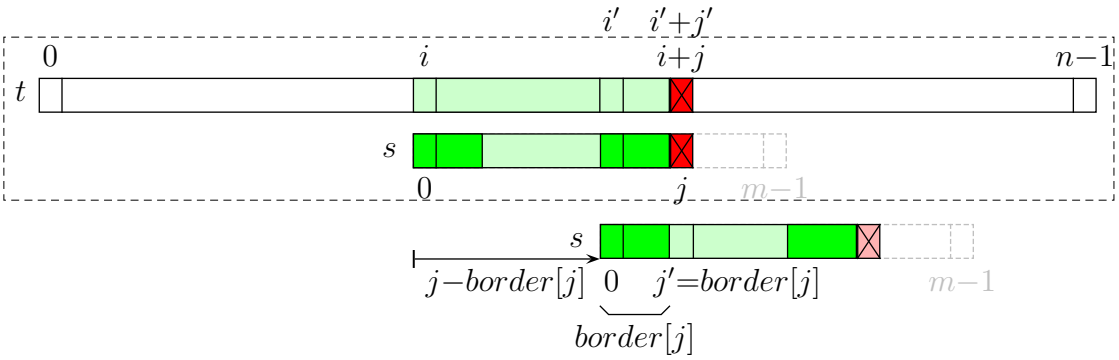


Abbildung 2.5: Skizze: Der Shift um $j - border[j]$

Beweis: Das Teilwort von s stimmt ab der Position i mit dem zugehörigen Teilwort von t von t_i bzw. s_0 mit t_{i+j-1} bzw. s_{j-1} überein, d.h. $t_{i+j} \neq s_j$ (siehe auch die Skizze in Abbildung 2.5). Der zum Teilwort $s_0 \cdots s_{j-1}$ gehörende eigentliche Rand hat laut Definition die Länge border$[j]$. Verschiebt man s um $j - border[j]$ nach rechts, so kommt der linke Rand des Teilwortes $s_0 \cdots s_{j-1}$ von s auf dem rechten Rand zu liegen, d.h. man schiebt „Gleiches“ auf „Gleiches“. Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt, der ungleich $s_0 \cdots s_{j-1}$ ist, ist dieser Shift sicher. ■

Im Falle der Voraussetzungen von Lemma 2.9 ist der Shift nicht nur sicher, sondern auch zulässig.

2.2.4 Der Knuth-Morris-Pratt-Algorithmus

Wenn wir die Überlegungen aus dem vorigen Abschnitt in einen Algorithmus gießen, erhalten wir den in Abbildung 2.6 angegeben, so genannten KMP-Algorithmus, benannt nach D.E. Knuth, J. Morris und V. Pratt.

2.2.5 Laufzeitanalyse des KMP-Algorithmus

Für die Analyse zählen wir die Anzahl der Zeichenvergleiche getrennt nach erfolglosen und erfolgreichen Zeichenvergleichen.

Definition 2.10 *Ein Vergleich von zwei Zeichen heißt erfolgreich, wenn die beiden Zeichen gleich sind, und* erfolglos *sonst.*

Es werden maximal $n-m+1$ erfolglose Zeichenvergleiche ausgeführt, da nach jedem erfolglosen Zeichenvergleich $i \in [0 : n - m]$ erhöht und im Verlaufe des Algorithmus nie erniedrigt wird.

```
bool Knuth_Morris_Pratt (char[] t, int n, char[] s, int m)
begin
    int border[m + 1];
    compute_borders(border[], m, s[]);
    int i := 0, j := 0;
    while (i ≤ n − m) do
        while (t[i + j] = s[j]) do
            j++;
            if (j = m) then return TRUE;
        i := i + (j − border[j]);          /* It holds that j − border[j] > 0 */
        j := max{0, border[j]};
    return FALSE;
end
```

Abbildung 2.6: Algorithmus: Die Methode von Knuth, Morris und Pratt

Wir werden zunächst zeigen, dass nach einem erfolglosen Zeichenvergleich der Wert von $i + j$ nie erniedrigt wird. Seien dazu i und j die Werte vor einem erfolglosen Zeichenvergleich und i' und j' die Werte nach einem erfolglosen Zeichenvergleich.

Somit ist der betrachtete Wert vor dem Zeichenvergleich $i+j$. Nach dem Zeichenvergleich beträgt er $i'+j' = (i+j-border[j])+(\max\{0, border[j]\})$. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob $border[j]$ negativ ist oder nicht.

Fall 1 ($\boldsymbol{border[j] \geq 0}$): Ist $border[j] \geq 0$, dann gilt offensichtlich $i' + j' = i + j$.

Fall 2 ($\boldsymbol{border[j] < 0}$): Ist $border[j] = -1$, dann muss $j = 0 = j'$ sein. Dann gilt aber offensichtlich $i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1 = i + j + 1$.

Also wird $i + j$ nach einem erfolglosen Zeichenvergleich nicht kleiner! Nach einem erfolgreichen Zeichenvergleich wird $i + j$ um 1 erhöht! Die maximale Anzahl erfolgreicher Zeichenvergleiche ist somit durch n beschränkt, da $i + j \in [0 : n - 1]$. Somit werden insgesamt maximal $2n - m + 1$ Zeichenvergleiche ausgeführt.

21.05.19

Lemma 2.11 *Der Algorithmus von Knuth, Morris und Pratt benötigt für die Suche von $s \in \Sigma^m$ in $t \in \Sigma^n$ maximal $2n-m+1$ Zeichenvergleiche, vorausgesetzt die Tabelle border steht zur Verfügung.*

2.2.6 Berechnung der Border-Tabelle

In der $border[]$-Tabelle wird für jedes Präfix $s_0 \cdots s_{j-1}$ der Länge $j \in [0 : m]$ des Suchstrings s der Länge m gespeichert, wie groß dessen eigentlicher Rand ist.

Wir initialisieren die Tabelle zunächst mit $border[0] = -1$ und $border[1] = 0$. Im Folgenden nehmen wir an, dass $border[0], \ldots, border[j-1]$ bereits berechnet sind. Wir wollen also jetzt den Wert von $border[j]$ (die Länge des eigentlichen Randes eines Suffixes der Länge j) berechnen. Wir betrachten dazu zuerst die Abbildung 2.7 und stellen fest, dass das um ein Zeichen kürzere Präfix des eigentlichen Randes von $s_0 \cdots s_{j-1}$ ein echter Rand von $s_0 \cdots s_{j-2}$ ist.

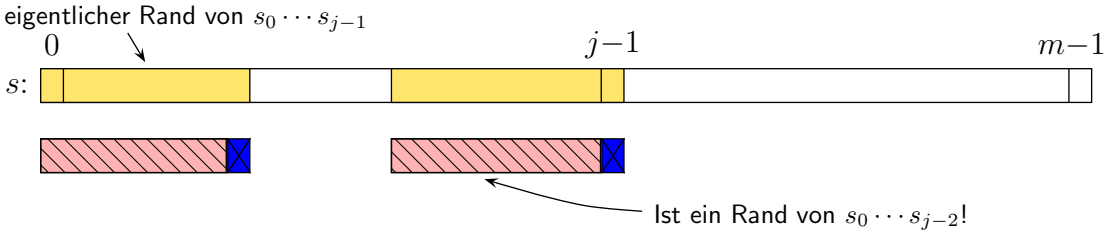


Abbildung 2.7: Skizze: Das eins kürzere Präfix des eigentlichen Randes von $s_0 \cdots s_{j-1}$ ist ein Rand von $s_0 \cdots s_{j-2}$

Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann daher um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$. Allgemein gilt, dass der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ daher nur eine Verlängerung um ein Zeichen eines echten Randes von $s_0 \cdots s_{j-2}$ sein kann. Wir müssen also nur alle echten Ränder von $s_0 \cdots s_{j-2}$ nach absteigender Länge aufzählen und prüfen, ob sich diese zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ verlängern lassen (siehe auch Abbildung 2.8).

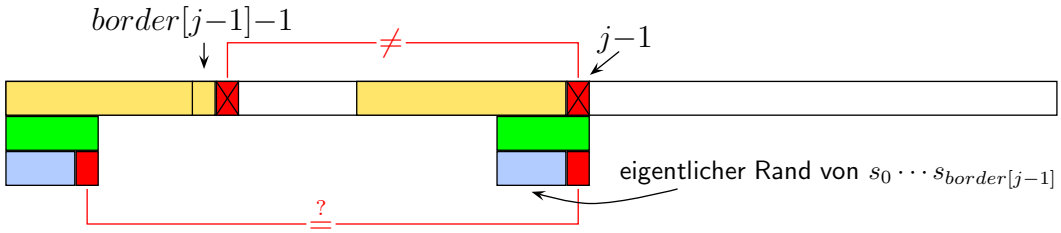


Abbildung 2.8: Skizze: Berechnung von $border[j]$ durch Verlängerung eines Randes

Ist $s_{border[j-1]} = s_{j-1}$, so lässt sich der Rand zum eigentlichen Rand von $s_0 \cdots s_{j-1}$ verlängern und es gilt $border[j] = border[j-1]+1$. Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist. Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes. Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft gerade das Präfix der Länge $border[border[j-1]]$.

Nun testen wir, ob sich dieser Rand von $s_0 \cdots s_{j-2}$ zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ erweitern lässt. Dies wiederholen wir solange, bis wir einen Rand gefunden

```
compute_borders (int[] border, int m, char[] s)
begin
    border[0] := −1;
    border[1] := 0;
    int i := border[1];
    for (int j := 2; j ≤ m; j++) do
        // Note that we have: i = border[j − 1]
        while ((i ≥ 0) && (s[i] ≠ s[j − 1])) do
            i := border[i];
        i++;
        border[j] := i;
end
```

Abbildung 2.9: Algorithmus: Berechnung der Tabelle *border*

haben, der sich zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt. Falls sich kein Rand von $s_0 \cdots s_{j-2}$ zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt, so ist der eigentliche Rand von $s_0 \cdots s_{j-1}$ das leere Wort und wir setzen $border[j] = 0$.

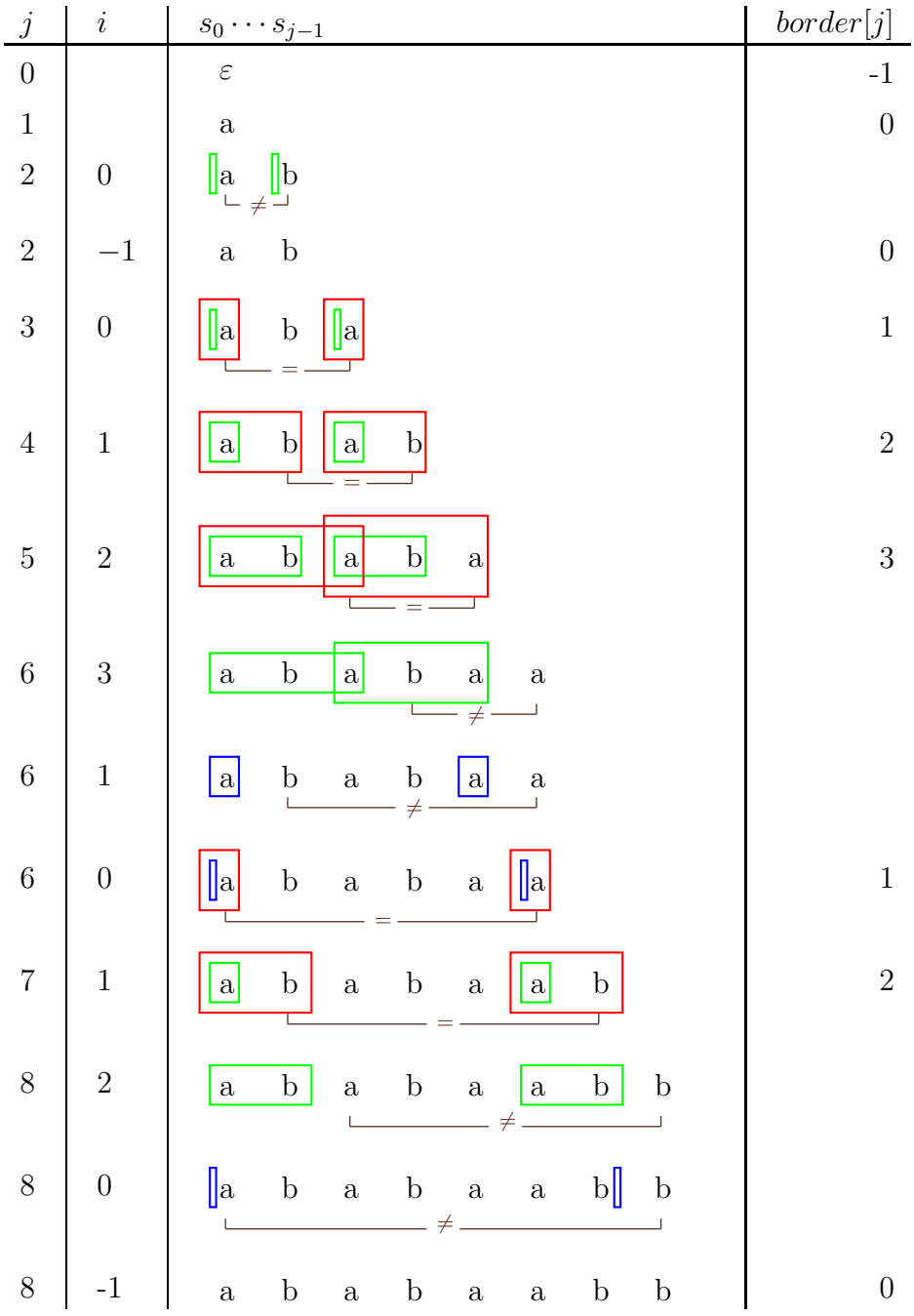
Damit erhalten wir den folgenden, in Abbildung 2.9 angegebenen Algorithmus zur Berechnung der Tabelle *border*, hierbei wird $border[j]$ auf Null gesetzt, wenn die While-Schleife wegen $i < 0$ abbricht (d.h. $i = -1$). Im Anschluss an die While-Schleife wird dann i auf den Wert 0 inkrementiert und $border[j] = 0$ gesetzt. In Abbildung 2.10 ist noch ein Beispiel zur Berechnung der Border-Tabelle für *ababaabb* angegeben.

2.2.7 Analyse der Gesamtlaufzeit

Wieder zählen wir die Zeichenvergleiche getrennt nach erfolgreichen und erfolglosen Zeichenvergleichen.

Es kann maximal $m - 1$ erfolgreiche Zeichenvergleiche geben, da jedes Mal für ein $j \in [2 : m]$ der Wert von j um 1 erhöht und nie erniedrigt wird.

Für die Anzahl erfolgloser Zeichenvergleiche betrachten wir den Wert i, zu Beginn ist $i = 0$. Nach jedem erfolgreichen Zeichenvergleich wird i inkrementiert. Also wird i genau $m - 1$ Mal um 1 erhöht, da die for-Schleife $m - 1$ Mal durchlaufen wird. Auf der anderen Seite kann i maximal $(m - 1) + 1 = m$ Mal erniedrigt werden, da immer $i \geq -1$ gilt. Es kann also nur das weggenommen werden, was schon einmal hinzugefügt wurde ($+1$ weil zu Beginn $i = 0$ ist und ansonsten immer $i \geq -1$ gilt. Also ist die Anzahl der Zeichenvergleiche insgesamt durch $2m - 1$ beschränkt.



grün: der bekannte eigentliche Rand
rot: der verlängerte (neu gefundene) eigentliche Rand
blau: der „längste echte Rand des letzten Randes“

Abbildung 2.10: Beispiel: Berechnung der Tabelle *border* für *ababaabb*

Lemma 2.12 *Für die Berechnung der Tabelle border sind maximal $2m - 1$ Zeichenvergleiche notwendig.*

Damit erhalten wir die Aussage über die Gesamtlaufzeit des Algorithmus.

Theorem 2.13 *Der Algorithmus von Knuth, Morris und Pratt benötigt maximal $2n + m$ Zeichenvergleiche, um festzustellen, ob ein Muster s der Länge m in einem Text t der Länge n enthalten ist.*

Der Algorithmus lässt sich leicht derart modifizieren, dass er alle Positionen der Vorkommen von s in t ausgibt, ohne dabei die asymptotische Laufzeit zu erhöhen. Die Details seien dem Leser als Übungsaufgabe überlassen.

2.3 Der Algorithmus von Aho und Corasick

Wir wollen jetzt nach mehreren Suchwörtern gleichzeitig im Text t suchen. Die kann zum Beispiel auftreten, wenn man nach mehreren interessierenden Proteinnamen in den Abstracts der pubmed suchen will.

MEHRFACHTEXTSUCHE

Eingabe: Ein Text $t \in \Sigma^n$ und eine Menge $S = \{s^1, \ldots, s^\ell\} \subseteq \Sigma^+$ von Suchwörtern mit $m := \sum_{s \in S} |s|$.
Gesucht: Taucht ein Suchwort $s \in S$ im Text t auf?

Wir nehmen hier zunächst an, dass in S kein Suchwort Teilwort eines anderen Suchwortes aus S ist.

2.3.1 Naiver Lösungsansatz

Eine einfache Lösungsmöglichkeit ist es, den KMP-Algorithmus für jedes Suchwort $s \in S$ auf t einzeln anzuwenden.

Die Kosten betragen für den naiven Ansatz für eine Suche nach s^i höchstens $2n + |s^i|$ und somit insgesamt:

$$\sum_{i=1}^{\ell} (2n + |s^i|) = 2\ell n + \sum_{i=1}^{\ell} |s^i| = 2\ell n + m.$$

Somit sind die Gesamtkosten $O(\ell n + m)$. Ziel ist die Elimination des Faktors ℓ.

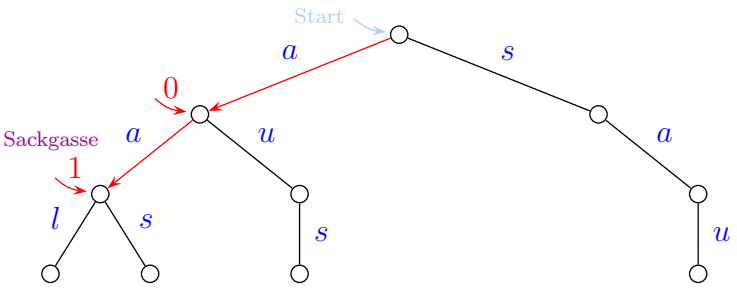
2.3.2 Der Algorithmus

Zuerst werden die Suchwörter in einem so genannten Suchwort-Baum organisiert.

Definition 2.14 *Ein* Suchwort-Baum *für eine Menge $S \subseteq \Sigma^+$ ist ein gewurzelter Baum mit folgenden Eigenschaften:*

- *Jede Kante ist mit genau einem Zeichen aus Σ markiert;*
- *Die von einem Knoten ausgehenden Kanten besitzen paarweise verschiedene Markierungen;*
- *Jedes Suchwort $s \in S$ wird auf einen Knoten v abgebildet, so dass s entlang des Pfades von der Wurzel zu v steht;*
- *Jedem Blatt ist ein Suchwort zugeordnet.*

In der Abbildung 2.11 auf der nächsten Seite ist ein solcher Suchwort-Baum für die Menge $\{aal, aas, aus, sau\}$ angegeben.



Beispiel: $t = \overset{0}{a}\ \overset{1}{a}\ u\ s$

Nach der Abarbeitung des Teilwortes aa gerät man in eine Sackgasse, da diesen Knoten keine Kante mit Label u verlässt. Somit wird das Teilwort aus, obwohl in t enthalten, nicht gefunden.

Abbildung 2.11: Beispiel: Aufbau des Suchwort-Baumes für $\{aal, aas, aus, sau\}$

Wie können wir nun mit diesem Suchwort-Baum im Text t suchen? Wir werden die Buchstaben des Textes t im Suchwort-Baum ablaufen. Sobald wir an einem Blatt gelandet sind, haben wir eines der gesuchten Wörter gefunden. Wir können jedoch auch in Sackgassen landen: Dies sind Knoten, von denen keine Kante mit einem gesuchten Kanten-Label ausgeht.

28.05.19

Damit es zu keinen Sackgassen kommt, werden in den Baum so genannte Failure-Links eingefügt.

Definition 2.15 *Ein* Failure-Link *eines Suchwort-Baumes ist ein Verweis von einem Knoten v auf einen Knoten w im Baum, so dass die Kantenmarkierung von der Wurzel zum Knoten w das längste echte Suffix des zu v korrespondierenden Wortes unter allen von der der Wurzel ausgehenden Wörtern ist.*

Die Failure-Links der Kinder der Wurzel werden so initialisiert, dass sie direkt zur Wurzel zeigen. Die Failure-Links der restlichen Knoten werden nun Level für Level von oben nach unten berechnet. Betrachten wir dazu den Knoten v mit Elter w, wobei das Kantenlabel von w zu v gerade a sei. Wir folgen dann dem bereits berechneten Failure-Link von w zum Knoten x. Hat der Knoten x eine ausgehende Kante zum Knoten y mit Markierung a, so setzen wir den Failure-Link vom Knoten v auf y. Andernfalls folgen wir wiederum dem Failure-Link von x.

Dieses Verfahren endet, wenn ein Knoten über Failure-Links erreicht wird, von dem eine ausgehende Kante mit Label a erreicht wird oder aber die Wurzel erreicht wird, von der keine ausgehende Kante die Markierung a trägt. Im letzten Fall setzen wir den Failure-Link von v auf die Wurzel des Baumes.

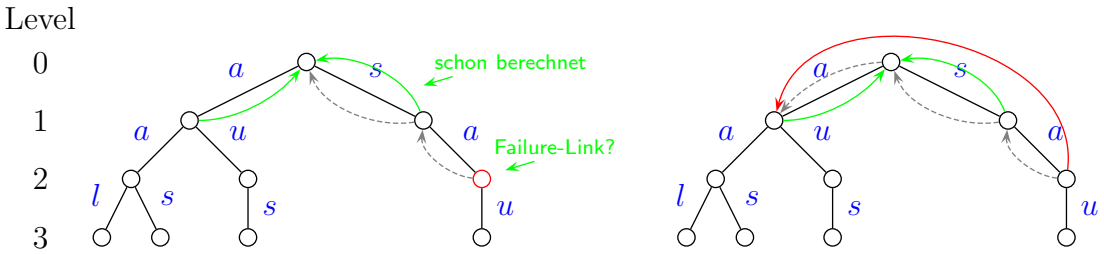


Abbildung 2.12: Beispiel: Berechnung eines Failure-Links

Für den Suchwort-Baum der Menge $\{aal, aas, aus, sau\}$ ist dies in Abbildung 2.12 für die Berechnung des Failure-Links des zu sa gehörigen Knotens illustriert. In Abbildung 2.13 sind alle Failure-Links des Suchwort-Baumes für die Suchwort-Menge $\{aal, aas, aus, sau\}$ illustriert. Der vollständige Algorithmus zur Bestimmung der Failure-Links ist in Abbildung 2.14 im Pseudo-Code angegeben.

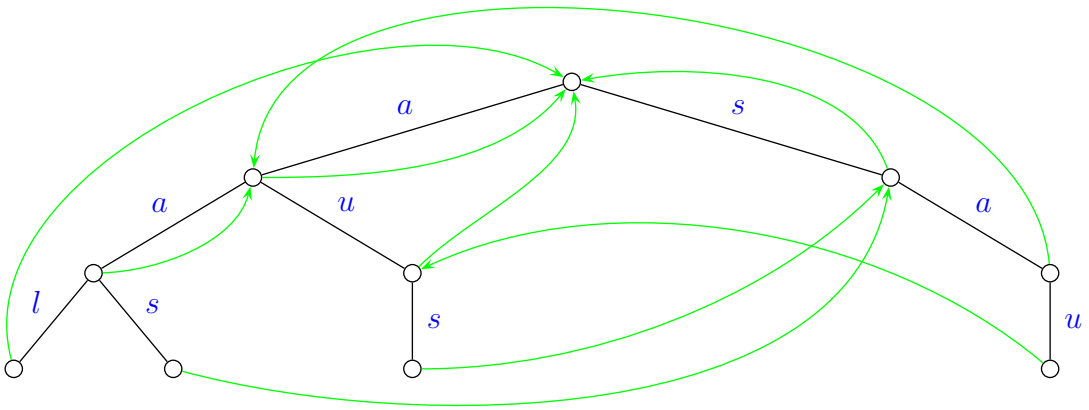


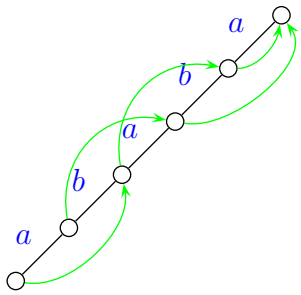
Abbildung 2.13: Beispiel: Der komplette Baum mit allen Failure-Links

Berechnung der Failure-Links (tree T)

begin
  `// T = (V, E) is the tree consisting of all search words`
  Failure_Link($r(T)$) = NULL;
  forall ($v \in V \setminus \{r(T)\}$) **do**
    `// traverse tree level by level`
    Let v' be the parent of v, where $(v' \xrightarrow{x} v) \in E$;
    $w :=$ Failure_Link(v');
    while (($w \neq$ NULL) && ($w \neq r(T)$) && ($\forall y \in V : (w \xrightarrow{x} y) \notin E$)) **do**
      $w :=$ Failure_Link(w)
    if (($w \neq$ NULL) && ($\exists w' \in V : (w \xrightarrow{x} w') \in E$)) **then**
      Failure_Link(v) := w';
    else
      Failure_Link(v) := $r(T)$;
end

Abbildung 2.14: Algorithmus: Berechnung der Failure-Links

Ist die Menge S einelementig, so erhalten wir als Spezialfall die Tabelle *border* des KMP-Algorithmus. Im Suchwort-Baum wird dabei auf das entsprechende längste Präfix (der auch Suffix ist) verwiesen. In der Tabelle *border* ist hingegen nur die Länge dieses Präfixes gespeichert. Da S einelementig ist, liefern beide Methoden dieselben Informationen. Dies ist für das Wort *ababa* noch einmal in Abbildung 2.15 illustriert.



Level 0, $border[0] = -1$
Level 1, $border[1] = 0$
Level 2, $border[2] = 0$
Level 3, $border[3] = 1$
Level 4, $border[4] = 2$
Level 5, $border[5] = 3$

Abbildung 2.15: Beispiel: $S = \{ababa\}$

Ein Suchwort $s \in S$ ist genau dann im Text t enthalten, wenn man beim Durchlaufen der Buchstaben von t im Suchwort-Baum in einem Blatt ankommt. Sind wir in einer Sackgasse gelandet, d.h. es gibt keine ausgehende Kante mit dem gewünschten Label, so folgen wir dem Failure-Link und suchen von dem so aufgefundenen Knoten

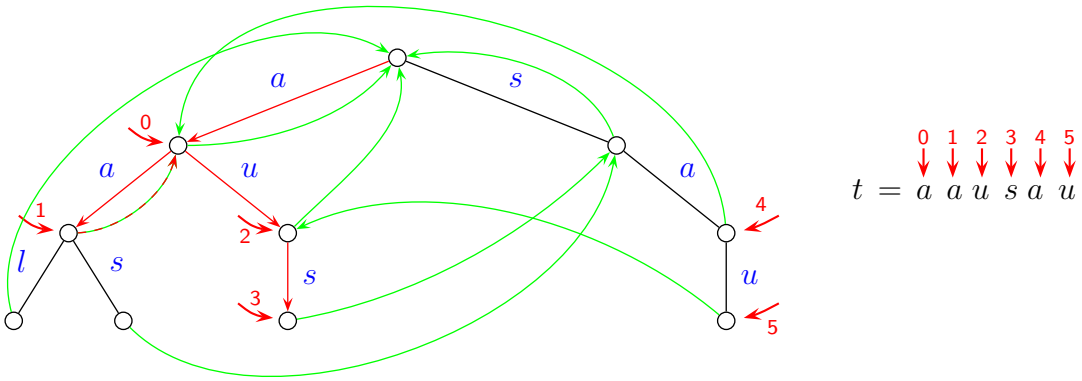


Abbildung 2.16: Beispiel: Suche in $t = aausau$, nun mit Failure-Links

aus weiter. Da wir angenommen haben, dass kein Suchwort Teilwort eines anderen Suchwortes ist, können wir beim Folgen von Failure-Links nie zu einem Blatt gelangen. In Abbildung 2.16 ist für die Suche noch einmal das vorherige Beispiel (etwas erweitert) vollständig dargestellt.

In der Abbildung 2.17 ist der Algorithmus von A. Aho und M. Corasick im Pseudocode angegeben.

bool Aho_Corasick (char $t[]$, int n, char $S[][]$, int m)

```
begin
    tree T := T(S);                                  // tree of search words in S
    node v := r(T);
    int i := 0;
    while (i < n) do
        while ((v --t_{i+lev(v)}--> v') in E(T)) do
            v := v';
            if (v' is a leaf) then
                return TRUE;
        if (v ≠ r(T)) then
            i := i + lev(v) − lev(Failure_Link(v));   // i will only be increased
            v := Failure_Link(v);                     // level of v will only be decreased
        else
            i++;
    return FALSE;
end
```

Abbildung 2.17: Algorithmus: Die Methode von Aho und Corasick

2.3.3 Laufzeitanalyse

Die Laufzeit zur Berechnung der Failure-Links beträgt $O(m)$. Um dies zu zeigen, betrachten wir ein festes Suchwort $s \in S$. Wir zeigen zunächst nur, dass für die Berechnung der Failure-Links der Knoten auf dem Pfad von s im Suchwort-Baum maximal $2|s|$ Zeichenvergleiche ausgeführt werden.

Wie bei der Analyse der Berechnung der Tabelle *border* des KMP-Algorithmus unterscheiden wir erfolgreiche und erfolglose Zeichenvergleiche. Zuerst halten wir fest, dass es maximal $|s|$ erfolgreiche Zeichenvergleiche (d.h., es gibt eine Kante $w \xrightarrow{x}$) geben kann, da wir dann zum nächsttieferen Knoten auf dem Pfad von s wechseln. Für die erfolglosen Zeichenvergleiche beachten wir, dass Failure-Links immer nur zu Knoten auf einem niedrigeren Level verweisen. Bei jedem erfolglosen Zeichenvergleich springen wir also zu einem Knoten auf einem niedrigeren Level (siehe auch Abbildung 2.18).

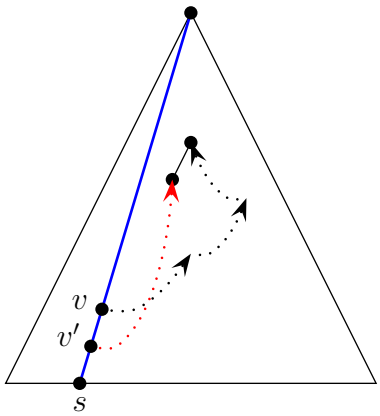


Abbildung 2.18: Skizze: Erweiterung Failure-Links

Da wir für einen neuen Knoten v' (der das Kind von v ist) versuchen den Failure-Link von v zu erweitern, gilt für die Folge der Levelwerte der Failure-Links für Knoten auf dem Pfad von der Wurzel zu dem Knoten für s, dass sich die Werte bei einem erfolglosen Zeichenvergleich verkleinern und bei einem erfolgreichen um 1 erhöhen. Somit kann es nur so viele erfolglose wie erfolgreiche Zeichenvergleiche geben.

Somit ist die Anzahl Zeichenvergleiche für jedes Wort $s \in S$ durch $2|s|$ beschränkt. Damit ergibt sich insgesamt für die Anzahl der Zeichenvergleiche, dass diese höchstens den folgenden Wert beträgt:

$$\sum_{s \in S} 2|s| = 2m = O(m).$$

Hierbei werden die Zeichenvergleiche für Berechnung der Failure-Links von manchen Knoten mehrfach gezählt, aber die Abschätzung ist für unsere Zwecke gut genug.

Auch für die eigentliche Suche des Aho-Corasick-Algorithmus verläuft die Laufzeitanalyse ähnlich wie beim KMP-Algorithmus. Da $lev(v) - lev(\text{Failure\_Link}(v)) > 0$ (analog zu $j - border[j] > 0$) ist, wird nach jedem erfolglosen Zeichenvergleich i um mindestens 1 erhöht. Also gibt es maximal n erfolglose Zeichenvergleiche. Weiterhin kann man zeigen, dass sich nach einem erfolglosen Zeichenvergleich $i + lev(v)$ nie erniedrigt und nach jedem erfolgreichen Zeichenvergleich um 1 erhöht (da sich $lev(v)$ um 1 erhöht). Da $i + j \in [0 : n - 1]$ ist, können maximal n erfolgreiche und somit maximal $2n$ Zeichenvergleiche ausgeführt worden sein.

2.3.4 Korrektheit des Algorithmus von Aho und Corasick

Es bleibt nur noch die Korrektheit des vorgestellten Algorithmus von Aho und Corasick nachzuweisen. Wenn kein Muster in t auftritt ist nach Kontsruktion klar, dass der Algorithmus nicht behauptet, dass ein Suchwort auftritt. Wir beschränken uns also auf den Fall, dass eines der Suchwörter aus S in t auftritt.

Für die folgende Argumentation betrachte auch die Abbildung 2.19. Was passiert, wenn $y \in S$ ein Teilwort von t ist und sich der Algorithmus unmittelbar nach dem Lesen von y in einem internen Knoten v befindet? Hierzu sei im Folgenden $t = t'yt''$ mit $y \in S$.

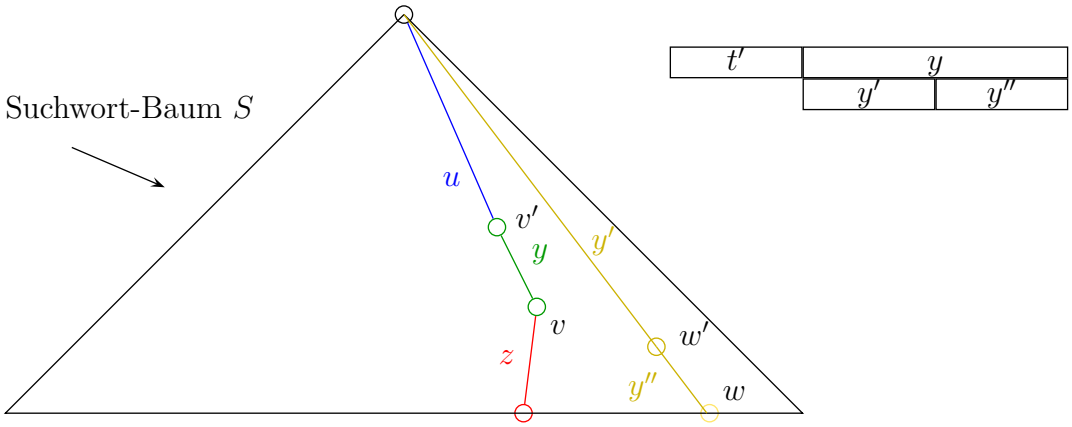


Abbildung 2.19: Skizze: Im Suchwort-Baum wird Treffer von y gemeldet

Zuerst betrachten wir dazu den Algorithmus nach der Abarbeitung eines Präfixes $t'y'$ von $t'y$ mit einer beliebigen Aufteilung von $y = y'xy''$ mit $y', y'' \in \Sigma^*$ und $x \in \Sigma$. Nach der Abarbeitung von t' befindet sich der Algorithmus trivialerweise in einem Konten mit Level mindestens 0. Weiterhin nehmen wir an, dass sich der Algorithmus dann nach Abarbeitung von $t'y'$ im Knoten u auf Level mindestens $|y'|$ befindet (für $y' = \varepsilon$ haben wir dies gerade gezeigt). Nach der Abarbeitung von $t'y'x$ befindet sich dann der Algorithmus in einem Knoten u' auf Level mindestens $|y'x|$, da beim Folgen

der Failure-Links in jedem Fall immer der Suffix y' auch als Präfix von $y \in S$ in der Suchwortmenge vorhanden ist. Somit kann kein Failure-Link auf einen Knoten mit Level kleiner als $|y'|$ zeigen und wir gelangen letztendlich zu einem Knoten auf Level mindestens $|y'x|$.

Wir nehmen nun an, dass wir uns nach Abarbeitung von $t'y$ am inneren Knoten v befinden, der sich damit mindestens auf Level $|y|$ befindet. Wäre v ein Blatt, so hätten wir y ja gefunden, da alle Blätter als Treffer markiert sind. Es könnte auch sein, dass v ein Blatt ist, das nicht mit y markiert ist, dann hätten wir eigentlich auch einen Fehler (zumindest in der Angabe des Suchworts). Dieser tritt nach sinnvoller Konstruktion der Suchwortbaumes jedoch nicht auf.

Also ist v ein innerer Knoten und somit muss man mit dem Wort z zu einem Blatt gelangen. Also ist $t'yz \in S$ und somit wäre y ein Teilwort eines anders Suchwortes in S, was wir zu Beginn ja ausgeschlossen hatten. Somit muss der Algorithmus nach der Abarbeitung von $t'y$ sich in einem Blatt befinden.

Theorem 2.16 *Sei $S \subseteq \Sigma^+$ eine Menge von Suchwörtern, so dass kein Wort $s \in S$ ein echtes Teilwort von $s' \in S$ (mit $s \neq s'$) ist, dann findet der Algorithmus von Aho-Corasick einen Match von $s \in S$ in $t \in \Sigma^n$ in der Zeit $O(n+m)$, wobei $m = \Sigma_{s \in S}|s|$.*

29.06.16

2.3.5 Erweiterung des Aho-Corasick-Algorithmus

Es stellt sich die Frage, wie der Algorithmus von Aho-Corasick zu erweitern ist, wenn ein Suchwort aus S ein echtes Teilwort eines anderen Suchwortes aus S sein darf. In diesem Fall ist es möglich, dass der Algorithmus beim Auftreten eines Suchwortes $s \in S$ in einem internen Knoten v' des Suchwort-Baumes endet. Sei im Folgenden s' die Kantenbeschriftung des Pfades von der Wurzel zum Knoten v' und, da $s \in S$, sei v der Endpunkt eines einfachen Pfades aus Baumkanten von der Wurzel, dessen Kantenbeschriftungen gerade s ergeben.

Wir überlegen uns zuerst, dass s ein Suffix von s' sein muss. Sei $t's$ das Präfix von t, der gelesen wurde, bis ein Suchwort $s \in S$ gefunden wird. Gemäß der Vorgehensweise des Algorithmus und der Definition der Failure-Links muss s' ein Suffix von $t's$ sein. Ist $|s'| \geq |s|$, dann muss s ein Suffix von s' sein. Andernfalls ist $|s'| < |s|$ und somit $lev(v') = |s'| < |s|$. Wir behaupten jetzt, dass dies nicht möglich sein kann. Betrachten wir hierzu die Abarbeitung von $t's$. Sei $\bar{s}$ das längste Präfix von s, so dass sich der Algorithmus nach der Abarbeitung von $t'\bar{s}$ in einem Knoten w mit $lev(w) \geq |\bar{s}|$ befindet. Da mindestens ein solches Präfix die Bedingung erfüllt (z.B. für $\bar{s} = \varepsilon$), muss es auch ein längstes geben. Anschließend muss der Algorithmus dem Failure-Link von w folgen, da ansonsten $\bar{s}$ nicht das Längste gewesen wäre. Sei

also $w' = \text{Failure-Link}(w)$. Dann gilt $lev(w') < |\bar{s}|$, da ansonsten $\bar{s}$ nicht das Längste gewesen wäre. Dies kann aber nicht sein, da es zu $\bar{s}$ einen Knoten im Suchwort-Baum auf Level $|\bar{s}|$ gibt, wobei die des Pfades von der Wurzel zu diesem Knoten gerade $\bar{s}$ ist (da ja $s \in S$ im Suchwort-Baum enthalten ist und $\bar{s}$ ein Präfix von s ist).

Betrachten wir nun den Failure-Link des Knotens v'. Dieser kann auf den Knoten v zeigen (da ja s als Suffix auf dem Pfad zu v' auftreten muss). Andernfalls kann er nach unserer obigen Überlegung nur auf andere Knoten auf einem niedrigeren Level zeigen, wobei die Beschriftung dieses Pfades dann s als Suffix beinhalten muss. Eine Wiederholung dieser Argumentation zeigt, dass letztendlich über die Failure-Links der Knoten v besucht wird. Daher genügt es, den Failure-Links zu folgen, bis ein Knoten erreicht wird, der zu einem Suchwort korrespondiert. In diesem Fall haben wir ein Suchwort im Text gefunden. Enden wir andernfalls an der Wurzel, so kann an der betreffenden Stelle in t kein Suchwort aus S enden.

Der Algorithmus von Aho-Corasick wird wie folgt erweitert. Wann immer ein neuer Knoten über eine Baumkante erreichbar ist, muss getestet werden, ob über eine Folge von Failure-Links (eventuell auch keine) auch ein zu einem Suchwort korrespondierender Knoten erreichbar ist. Falls ja, haben wir ein Suchwort gefunden, ansonsten nicht.

Anstatt nun jedes Mal den Failure-Links zu folgen, können wir dies auch in einem Vorverarbeitungsschritt durchführen. Zuerst markieren wir jeden Knoten, der zu einem Suchwort $s \in S$ korrespondiert, als Treffer (also insbesondere alle Blätter), die Wurzel als Nicht-Treffer und alle anderen (dann natürlicherweise internen) Knoten als unbekannt. Dann durchlaufen wir wieder levelweise alle Knoten des Baumes. Ein Knoten wird nun auch als Treffer markiert, wenn sein Failure-Link auf einen als Treffer markierten Knoten zeigt. Da der Baum levelweise von den niedrigeren zu den höheren Leveln durchlaufen wird, wird ein Knoten dann als Treffer markiert, wenn es von ihm einen Folge von Failure-Links gibt, die zu einem Knoten führt, der zu einem Suchwort aus S gehört.

Sobald wir nun im normalen Algorithmus von Aho-Corasick auf einen Knoten treffen, der mit Treffer markiert ist, haben wir ein Suchwort im Text gefunden, ansonsten nicht. Die Vorverarbeitung lässt sich in Zeit $O(m)$ implementieren, so dass die Gesamtlaufzeit bei $O(m + n)$ bleibt.

Wollen wir zusätzlich auch noch die Startpositionen aller Treffer ausgeben bzw. welche Suchwörter den Treffer ausgelöst haben, so müssen wir den Algorithmus noch weiter modifizieren. Wir hängen zusätzlich an jeden als Treffer markierten Knoten v noch einen Verweis $p(v) = w$, den so genannten *Hit-Link*, auf den zu einem Suchwort korrespondierenden Knoten w an, den man von dem als Treffer markierten Knoten als erstes über eine Folge von Failure-Links erreicht. Zu Beginn erhält jeder Knoten einen leeren Verweis. Wird ein Knoten v als Treffer markiert, weil der über seinen

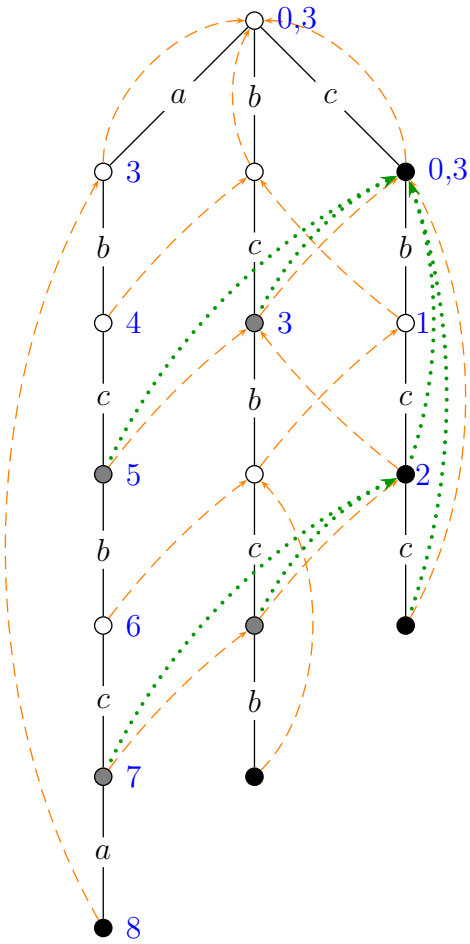


Abbildung 2.20: Beispiel: Suchwort-Baum mit Failure- und Hit-Links

Failure-Link erreichbare Knoten w zu einem Suchwort korrespondiert, dann setzen wir $p(v) = w$; andernfalls setzen wir $p(v) = p(w)$.

Treffen wir nun beim Algorithmus von Aho-Corasick auf einen als Treffer markierten Knoten, so folgen wir den Verweisen $p(\cdot)$ und geben für jeden über die Verweise besuchten Knoten das zugehörige Suchwort aus. Somit ergibt sich für die Laufzeit $O(m+n+k)$, wobei m die Anzahl der Zeichen in den Suchwörtern ist, n die Länge des Textes t und k die Anzahl der Vorkommen von Suchwörtern aus S in t.

Wir schauen uns das noch einmal in einem Beispiel in Abbildung 2.20 für die folgende Suchwortmenge S an:

$$S = \{abcbca, bcbcb, c, cbc, cbcc\}.$$

Failure-Links sind in dieser Abbildung orange und gestrichelt, Hit-Links grün und gepunktet dargestellt.

Schauen wir uns nun noch die Suche nach S in einem Text

$$t = t_0 \cdots t_8 = cbcabcbca$$

genauer an. In der Abbildung 2.20 gibt die blaue Zahl i den Knoten an, an dem versucht wird, mit s_i weiterzuarbeiten, bei $*$ befindet sich der Algorithmus am Ende des Textes t.

Theorem 2.17 *Der Algorithmus von Aho und Corasick findet alle Vorkommen von $s \in S \subseteq \Sigma^+$ in $t \in \Sigma^n$ in Zeit $O(n + m + k)$, wobei $m = \Sigma_{s \in S}|s|$ und k die Anzahl der Vorkommen von Suchwörtern aus S in t ist.*

2.4 Der Algorithmus von Boyer und Moore

In diesem Kapitel werden wir einen weiteren Algorithmus zum exakten Suchen in Texten vorstellen, der zwar im worst-case schlechter als der Knuth-Morris-Pratt-Algorithmus ist, der jedoch in der Praxis meist wesentlich bessere Resultate bzgl. der Laufzeit zeigt.

2.4.1 Ein zweiter naiver Ansatz

Ein weiterer Ansatzpunkt zum Textsuchen ist der, das gesuchte Wort mit einem Textstück nicht mehr von links nach rechts, sondern von rechts nach links zu vergleichen. Diese zweite naive Idee ist im Algorithmus in Abbildung 2.21 angegeben.

bool Naiv2 (char $t[]$, int n, char $s[]$, int m)

```
begin
    int i := 0, j := m − 1;
    while (i ≤ n − m) do
        while (t[i + j] = s[j]) do
            if (j = 0) then return TRUE;
            j−−;
        i++;
        j := m − 1;
    return FALSE;
end
```

Abbildung 2.21: Algorithmus: Eine naive Methode mit rechts-nach-links Vergleichen

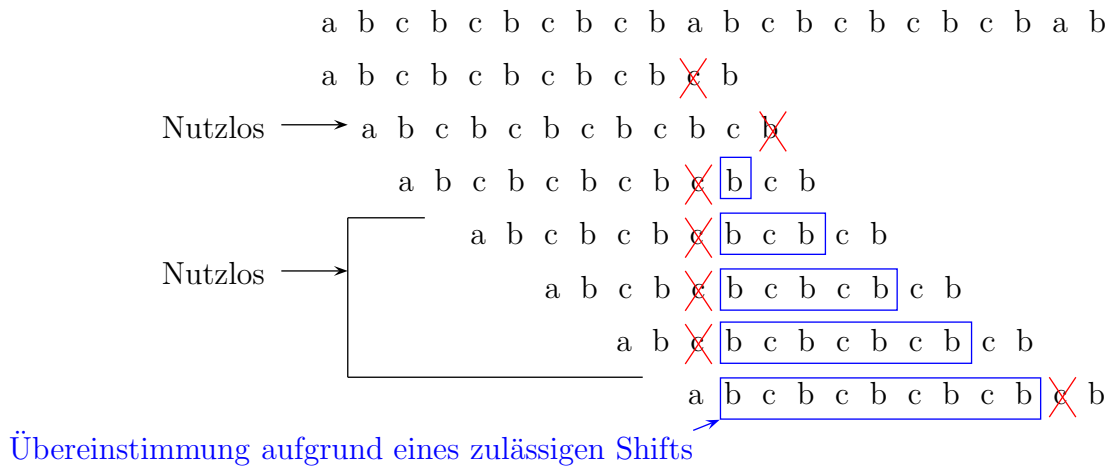


Abbildung 2.22: Beispiel: Suchen mit der zweiten naiven Methode

Das folgende Bild in Abbildung 2.22 zeigt ein Beispiel für den Ablauf des naiven Algorithmus. Dabei erkennt man recht schnell, dass es nutzlos ist (siehe zweiter Versuch von oben), wenn man die Zeichenreihe so verschiebt, dass im Bereich erfolgreicher Zeichenvergleiche nach einem Shift keine Übereinstimmung mehr herrscht. Diese Betrachtung ist völlig analog zur Schiebe-Operation im Algorithmus von Knuth, Morris und Pratt.

Weiter bemerkt man, dass es ebenfalls keinen Sinn macht, das Muster so zu verschieben, dass an der Position des Mismatches in t im Muster s wiederum dasselbe Zeichen zum Liegen kommt, das schon vorher den Mismatch ausgelöst hat. Daher kann man auch den vierten bis sechsten Versuch als nutzlos bezeichnen.

Ein weiterer Vorteil dieser Vorgehensweise ist, dass man in Alphabeten mit vielen verschiedenen Symbolen bei einem Mismatch von s_j mit t_{i+j}, i auf $i + j + 1$ setzen kann, falls das im Text t verglichene Zeichen t_{i+j} gar nicht im gesuchten Wort s vorkommt. D.h. in so einem Fall kann man das Suchwort gleich um $j + 1$ Positionen verschieben.

2.4.2 Der Algorithmus von Boyer-Moore

Der von R.S. Boyer und J.S. Moore vorgeschlagene Algorithmus unterscheidet sich vom zweiten naiven Ansatz nunmehr nur in der Ausführung größerer Shifts, welche in der Shift-Tabelle gespeichert sind. Die folgende Skizze in Abbildung 2.23 zeigt die möglichen Shifts beim Boyer-Moore-Algorithmus.

Prinzipiell gibt es zwei mögliche Arten eines „vernünftigen“ Shifts bei der Variante von Boyer-Moore. Im oberen Teil ist ein „kurzer“ Shift angegeben, bei dem im grünen

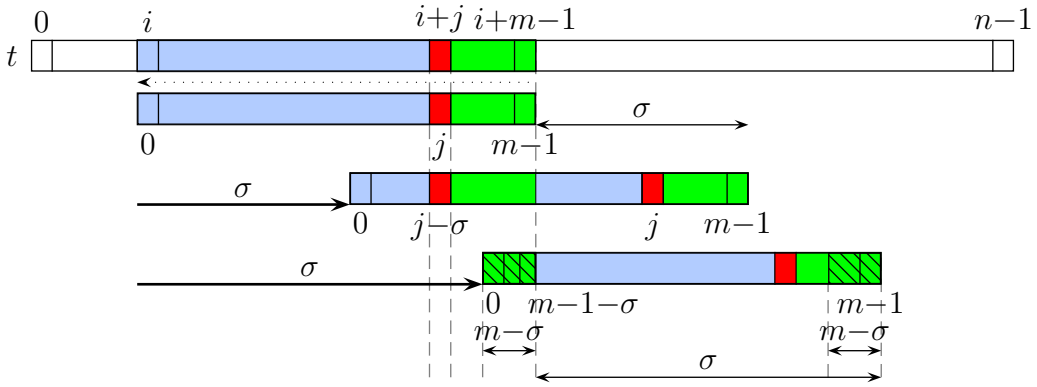


Abbildung 2.23: Skizze: Zulässige Shifts bei Boyer-Moore (Strong-Good-Suffix-Rule)

Bereich die Zeichen nach dem Shift weiterhin übereinstimmen. Das rote Zeichen in t, welches den Mismatch ausgelöst hat, wird nach dem Shift auf ein anderes Zeichen in s treffen, damit überhaupt die Chance auf Übereinstimmung besteht. Im unteren Teil ist ein „langer“ Shift angegeben, bei dem die Zeichenreihe s soweit verschoben wird, dass an der Position des Mismatches in t gar kein weiterer Zeichenvergleich mehr entsteht. Allerdings soll auch hier im schraffierten grünen Bereich wieder Übereinstimmung mit den bereits verglichenen Zeichen aus t herrschen.

Die Kombination der beiden obigen Regeln nennt man die *Good-Suffix-Rule*, da man darauf achtet, die Zeichenreihen so zu verschieben, dass im letzten übereinstimmenden Bereich nach dem Shift wieder Übereinstimmung herrscht. Achtet man noch speziell darauf, dass an der Position, in der es zum Mismatch gekommen ist, jetzt in s eine anderes Zeichen liegt, als das, welches den Mismatch ausgelöst hat, so spricht man von der *Strong-Good-Suffix-Rule*, andernfalls *Weak-Good-Suffix-Rule*. Im Folgenden werden wir nur diese Strong-Good-Suffix-Rule betrachten, da ansonsten die worst-case Laufzeit wieder quadratisch werden kann. Der Leser möge sich ein Beispiel hierfür überlegen.

Der sich aus dieser Idee abgeleitete Boyer-Moore-Algorithmus ist in der Abbildung 2.24 angegeben. Hierbei ist $S[]$ die inhaltlich eben beschriebene Shift-Tabelle, über deren Einträge wir uns im Detail im Folgenden noch Gedanken machen werden.

Man beachte hierbei, dass es nach dem Shift einen Bereich gibt, in dem Übereinstimmung von s und t vorliegt. Allerdings werden auch in diesem Bereich wieder Zeichenvergleiche ausgeführt, da es letztendlich doch zu aufwendig ist, sich diesen Bereich explizit zu merken und bei folgenden Zeichenvergleichen von s in t zu überspringen.

Offen ist nun nur noch die Frage nach den in der Shift-Tabelle zu speichernden Werten und wie diese effizient bestimmt werden können. Hierzu sind einige Vorüberlegungen zu treffen. Der erste Mismatch soll im zu durchsuchenden Text t an

```
bool Boyer_Moore (char t[], int n, char s[], int m)
begin
    int S[m];
    compute_shift_table(S, m, s);
    int i := 0, j := m − 1;
    while (i ≤ n − m) do
        while (t[i + j] = s[j]) do
            if (j = 0) then return TRUE;
            j−−;
        i := i + S[j];
        j := m − 1;
    return FALSE;
end
```

Abbildung 2.24: Algorithmus: Boyer-Moore mit Strong-Good-Suffix-Rule

der Stelle $i+j$ auftreten. Da der Boyer-Moore-Algorithmus das Suchwort von hinten nach vorne vergleicht, ergibt sich folgende Voraussetzung:

$$s_{j+1} \cdots s_{m-1} = t_{i+j+1} \cdots t_{i+m-1} \qquad \wedge \qquad s_j \neq t_{i+j}$$

Um nun einen nicht nutzlosen Shift um σ Positionen zu erhalten, muss gelten:

$$s_{j+1-\sigma} \cdots s_{m-1-\sigma} = t_{i+j+1} \cdots t_{i+m-1} = s_{j+1} \cdots s_{m-1} \qquad \wedge \qquad s_j \neq s_{j-\sigma}$$

Diese Bedingung ist nur für „kleine“ Shifts mit $\sigma \leq j$ sinnvoll. Ein solcher Shift ist im Bild der Abbildung 2.23 als erster Shift zu finden. Für „große“ Shifts $\sigma > j$ muss gelten, dass das Suffix des übereinstimmenden Bereichs mit dem Präfix des Suchwortes übereinstimmt, d.h.:

$$s_0 \cdots s_{m-1-\sigma} = t_{i+\sigma} \cdots t_{i+m-1} = s_\sigma \cdots s_{m-1}$$

Zusammengefasst ergibt sich für beide Bedingungen Folgendes:

$$\begin{array}{lllll} \sigma \leq j & \wedge & s_{j+1} \cdots s_{m-1} \in \mathcal{R}(s_{j+1-\sigma} \cdots s_{m-1}) & \wedge & s_j \neq s_{j-\sigma} \\ \sigma > j & \wedge & s_0 \cdots s_{m-1-\sigma} \in \mathcal{R}(s_0 \cdots s_{m-1}) & & \end{array}$$

wobei $\mathcal{R}(s)$ die Menge aller Ränder von s bezeichnet. Erfüllt ein Shift nun eine dieser Bedingungen, so nennt man diesen Shift *zulässig*. Um einen sicheren Shift zu erhalten, wählt man das minimale σ, das eine der Bedingungen erfüllt. Somit gilt:

$$S[j] = \min \left\{ \sigma : \begin{array}{l} (s_{j+1} \cdots s_{m-1} \in \mathcal{R}(s_{j+1-\sigma} \cdots s_{m-1}) \wedge s_j \neq s_{j-\sigma} \wedge \sigma \leq j) \vee \\ (s_0 \cdots s_{m-1-\sigma} \in \mathcal{R}(s_0 \cdots s_{m-1}) \wedge \sigma > j) \ \vee \ (\sigma = m) \end{array} \right\}.$$

2.4.3 Bestimmung der Shift-Tabelle

In Abbildung 2.25 ist der Algorithmus zur Konstruktion der Shift-Tabelle für den Boyer-Moore-Algorithmus angegeben. Wir gehen nun auf die Details der Berechnung näher ein.

Zu Beginn wird die Shift-Tabelle an allen Stellen mit der Länge des Suchstrings initialisiert. Im Wesentlichen entsprechen beide Fälle von möglichen Shifts (siehe obige Vorüberlegungen) der Bestimmung von Rändern von Teilwörtern des gesuchten Wortes. Dennoch unterscheiden sich die hier betrachteten Fälle vom KMP-

compute_shift_table (int $S[]$, char $s[]$, int m)

```
begin
    // Initialize S[]
    for (int j := 0; j < m; j++) do
        S[j] := m;

    // Part 1: σ ≤ j
    int border2[m + 1];
    border2[0] := −1;
    int i := border2[1] = 0;
    for (int j' := 2; j' ≤ m; j'++) do
        // Note that i = border2[j' − 1]
        while ((i ≥ 0) && (s[m − i − 1] ≠ s[m − j'])) do
            int σ := j' − i − 1;
            S[m − i − 1] := min(S[m − i − 1], σ);
            i := border2[i];
        i++;
        border2[j'] := i;

    // Part 2: σ > j
    int j := 0;
    for (int i := border2[m]; i ≥ 0; i := border2[i]) do
        int σ := m − i;
        while (j < σ) do
            S[j] := min(S[j], σ);
            j++;
end
```

Abbildung 2.25: Algorithmus: Berechnung der Shift-Tabelle für Boyer-Moore

Algorithmus, da hier, im Gegensatz zu den Rändern von Präfixen des Suchwortes, die Ränder von Suffixen des Suchwortes gesucht sind.

Dies gilt besonders für den ersten Fall ($\sigma \leq j$). Genauer gesagt, ist man im ersten Fall besonders daran interessiert, dass das Zeichen unmittelbar vor dem Rand des Suffixes ungleich dem Zeichen unmittelbar vor dem gesamten Suffix sein soll (siehe auch Abbildung 2.26). Diese Situation entspricht dem Fall in der Berechnung eines eigentlichen Randes, bei dem der vorhandene Rand nicht zu einem längeren Rand fortgesetzt werden kann. Daher sieht die erste for-Schleife genau so aus, wie in der Berechnung von compute_border.

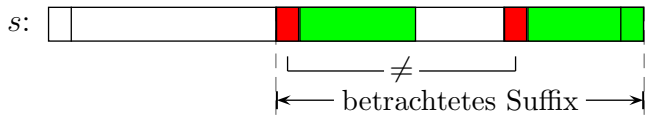


Abbildung 2.26: Skizze: Verlängerung eines Randes eines Suffixes

04.06.19

Da hier allerdings Suffixe statt Präfixe betrachtet werden, wird in Abbildung 2.25 im Teil 1 die Tabelle $border2$ berechnet, wobei $border2[j']$ die Länge des eigentlichen Randes des Suffixes der Länge j' von s ist. Lässt sich ein betrachteter Rand nicht verlängern, so wird die while-Schleife ausgeführt. In den grünen Anweisungen wird zunächst die Länge des Shifts berechnet und dann die entsprechende Position in der Shift-Tabelle (wo der Mismatch in s passiert ist) aktualisiert. Für die Aktualisierung der Shift-Tabelle im Teil 1 des Algorithmus siehe auch die Abbildung 2.27).

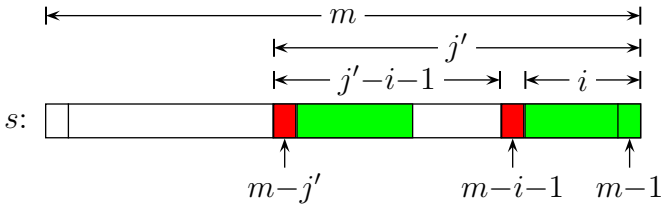


Abbildung 2.27: Skizze: Aktualisierung der Shift-Tabelle (Teil 1)

Wir müssen uns nur noch überlegen, warum man im Falle einer Verlängerung des Randes kürzere Ränder nicht mehr beachten muss. Betrachten wir hierfür die Skizze in Abbildung 2.28. Nehmen wir an, wir hätten eine erfolgreiche Verlängerung des grünen Randes x um das Zeichen a. Es könnte ja einen kürzerer Rand y geben, vor dem ein b steht (das dann ungleich a ist. Somit könnten man die Shifttabelle an der Position des b's auch aktualisieren. Hierfür wurde aber bereits früher diese Position in der Shift-Tabelle mit einem kleineren Wert aktualisiert, da der diese kürzere Rand mit einem a davor auch schon einmal früher (also weiter hinten im Suchwort s) auftreten musste.

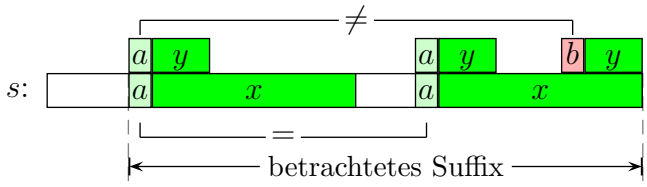


Abbildung 2.28: Skizze: Nichtbeachtung der Verlängerung kürzerer Ränder eines Suffixes

Im zweiten Fall ($\sigma > j$) müssen wir alle Ränder von s durchlaufen. Der längste Rand ungleich s ist der eigentliche Rand von s. Diesen erhalten wir über $border2[m]$, da ja das Suffix der Länge m von s gerade wieder s ist. Was ist der nächstkürzere Rand von s? Dieser muss ein Rand des eigentlichen Randes von s sein. Damit es der nächstkürzere Rand ist, muss s der eigentliche Rand des eigentlichen Randes von s sein, also das Suffix der Länge $border2[border2[s]]$.

Somit können wir alle Ränder von s durchlaufen, indem wir immer vom aktuell betrachteten Rand der Länge ℓ das Suffix der Länge $border2[\ell]$ wählen. Dies geschieht in der for-Schleife im zweiten Teil. Solange der Shift σ größer als die Position j eines Mismatches ist, wird die Shift-Tabelle aktualisiert. Dies geschieht in der inneren while-Schleife. Für die Aktualisierung der Shift-Tabelle im Teil 2 des Algorithmus siehe auch die Abbildung 2.29).

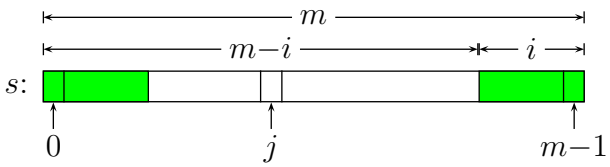


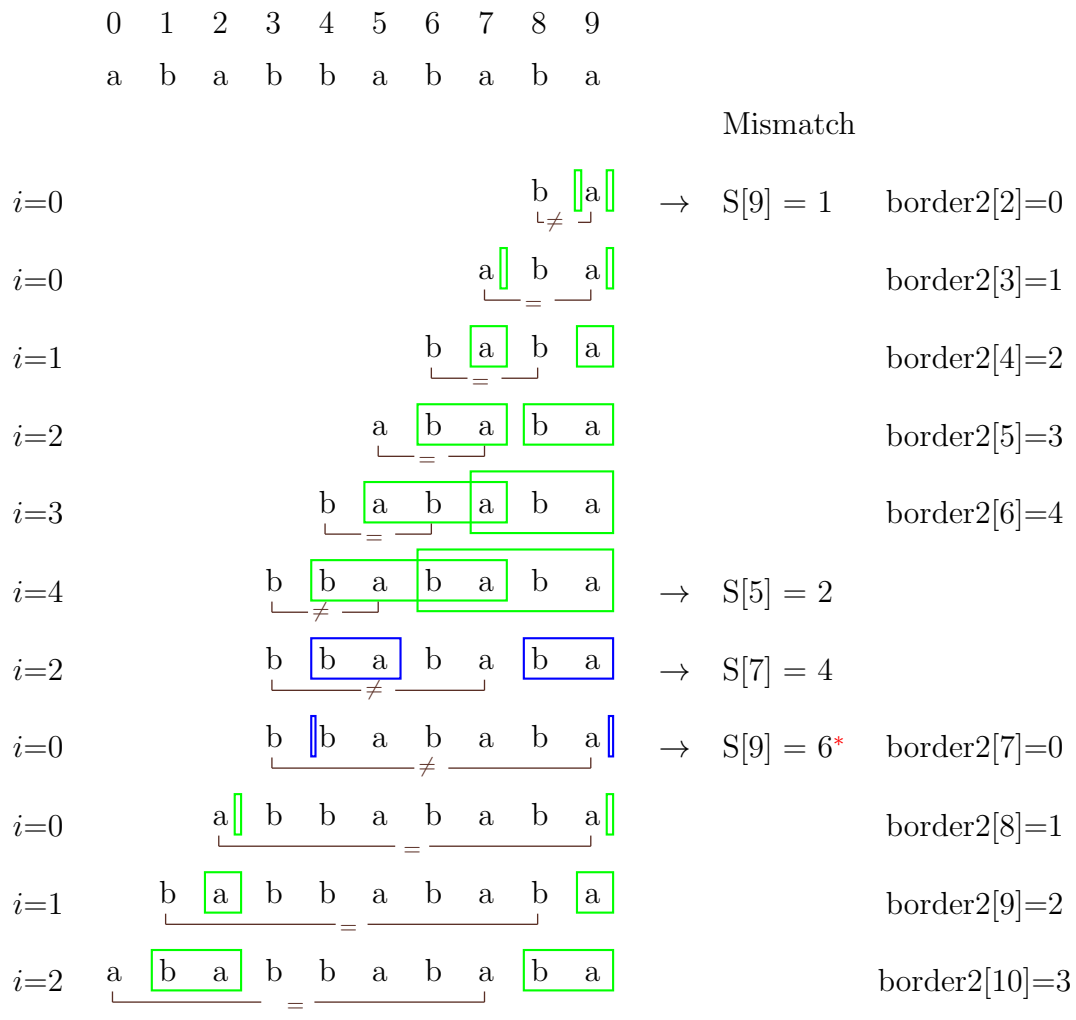
Abbildung 2.29: Skizze: Aktualisierung der Shift-Tabelle (Teil 2)

Wir haben bei der Aktualisierung der Shift-Tabelle nur zulässige Werte berücksichtigt. Damit sind die Einträge der Shift-Tabelle (d.h. die Länge der Shifts) nie zu klein. Eigentlich müsste jetzt noch gezeigt werden, dass die Werte auch nicht zu groß sind, d.h. dass es keine Situation geben kann, in der ein kleinerer Shift möglich wäre. Dass dies nicht der Fall ist, kann mit einem Widerspruchsbeweis gezeigt werden (man nehme dazu an, bei einem Mismatch an einer Position j in s gäbe es einen kürzeren Shift gemäß der Strong-Good-Suffix-Rule und leite daraus einen Widerspruch her). Die Details seien dem Leser zur Übung überlassen.

In Abbildung 2.30 und 2.31 ist ein Beispiel angegeben, wie für das Wort *ababbababa* die Shift-Tabelle berechnet wird. In Abbildung 2.32 ist die resultierende Shift-Tabelle mit den Aktualisierungen aus beiden Teilen angegeben.

Die Shift-Tabelle $S[j]$ wird für alle $j \in [0 : m-1]$ mit der Länge m des Suchstrings vorbesetzt.

Teil 1: $\sigma \leq j$



Die Besetzung der Shift-Tabelle erfolgt immer nach dem Prinzip, dass das Minimum des gespeicherten Wertes und des neu gefundenen Wertes gespeichert wird, d.h. $S[j] = \min\{S[j]; \text{neu gefundener Wert}\}$.

* Dieser Wert wird nicht gespeichert, da S[9] schon mit 1 belegt ist

Abbildung 2.30: Beispiel: $s = ababbababa$ (Teil 1: $\sigma \leq j$)

Teil 2: $\sigma > j$

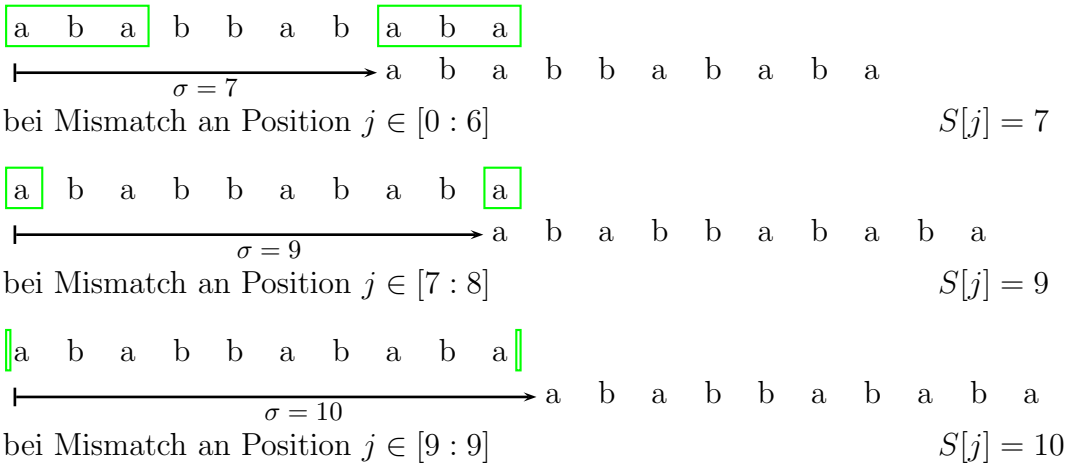


Abbildung 2.31: Beispiel: $s = ababbababa$ (Teil 2: $\sigma > j$)

Zusammenfassung:

S[0] =		7	7	S[5] =	2		2
S[1] =		7	7	S[6] =		7	7
S[2] =		7	7	S[7] =	4		4
S[3] =		7	7	S[8] =		9	9
S[4] =		7	7	S[9] =	1		1
	1.Teil	2.Teil	Erg		1.Teil	2.Teil	Erg

Abbildung 2.32: Beispiel: $s = ababbababa$ (Zusammenfassung)

2.4.4 Laufzeitanalyse des Boyer-Moore Algorithmus

Man sieht, dass die Prozedur *compute_shift_table* hauptsächlich auf die Prozedur *compute_border* des KMP-Algorithmus zurückgreift. Eine nähere Betrachtung der beiden Schleifen des zweiten Teils ergibt, dass die Schleifen nur m-mal durchlaufen werden und dort überhaupt keine Zeichenvergleiche ausgeführt werden.

Lemma 2.18 *Die Shift-Tabelle des Boyer-Moore-Algorithmus lässt sich für eine Zeichenkette der Länge m mit maximal $2m$ Zeichenvergleichen berechnen.*

Es bleibt demnach nur noch die Anzahl der Zeichenvergleiche in der Hauptprozedur zu bestimmen. Hierbei wird nur die Anzahl der Zeichenvergleiche für eine erfolglose Suche bzw. für das erste Auftreten des Suchwortes s im Text t betrachtet. Wir gehen später noch darauf ein, wie sich die Laufzeit verhält, wenn mehrere Vorkommen von

s in t auftreten. Wir werden zum Abzählen der Zeichenvergleiche wieder einmal zwischen zwei verschiedenen Arten von Zeichenvergleichen unterscheiden: so genannte *initiale* und *wiederholte* Zeichenvergleiche.

Initiale Zeichenvergleiche: Zeichenvergleiche von s_j mit t_{i+j}, so dass t_{i+j} zum ersten Mal beteiligt ist.

Wiederholte Zeichenvergleiche: Zeichenvergleiche s_j mit t_{i+j}, so dass t_{i+j} schon früher einmal bei einem Zeichenvergleich beteiligt war.

Wir betrachten jetzt die Situation, wie in Abbildung 2.33 angegeben, wenn wir bei einem Test ℓ erfolgreiche Zeichenvergleiche ausgeführt haben.

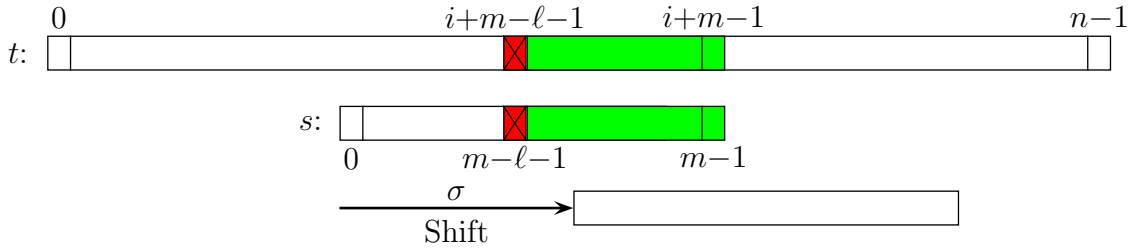


Abbildung 2.33: Skizze: Shift nach ℓ erfolgreichen Zeichenvergleichen

Lemma 2.19 *Sei $s_{m-\ell} \cdots s_{m-1} = t_{i+m-\ell} \cdots t_{i+m-1}$ für ein $i \in [0 : n-m]$ und ein $\ell \in [0 : m-1]$ sowie $s_{m-\ell-1} \neq t_{i+m-\ell-1}$ (es wurden also bei einem Versuch im Boyer-Moore-Algorithmus ℓ erfolgreiche und ein erfolgloser Zeichenvergleich durchgeführt). Dabei seien I initiale Zeichenvergleiche durchgeführt worden und weiter sei σ die Länge des folgenden Shifts gemäß der Strong-Good-Suffix-Rule. Dann gilt:*

$$\ell + 1 \leq I + 4 \cdot \sigma.$$

Aus diesem Lemma können wir sofort folgern, dass maximal $5n+m$ Zeichenvergleiche ausgeführt wurden.

Lemma 2.20 *Bei einer erfolglosen oder erfolgreichen Suche nach dem ersten Vorkommen von $s \in \Sigma^m$ in $t \in \Sigma^n$ treten maximal $5n + m$ Zeichenvergleiche auf.*

Beweis: Bezeichne dazu $V(n)$ die Anzahl aller Zeichenvergleiche, um in einem Text der Länge n zu suchen, und I_i bzw. V_i für $i \in [1 : k]$ die Anzahl der initialen bzw. aller Zeichenvergleiche, die beim i-ten Versuch, s in t zu finden, ausgeführt wurden.

Ferner sei σ_i die Länge des Shifts, der nach dem i-ten Versuch ausgeführt wird. War der k-te (und somit letzte) Versuch erfolgreich, sei $\sigma_k := 1$.

$$
\begin{aligned}
V(n) &= \sum_{i=1}^{k} V_i \\
&\quad \text{im letzten Versuch werden maximal } m \text{ Zeichenvergleiche ausgeführt} \\
&\leq \sum_{i=1}^{k-1} V_i + m \\
&\quad \text{mit Hilfe des obigen Lemmas (da dort } V_i = 1 + \ell) \\
&\leq \sum_{i=1}^{k-1} (I_i + 4\sigma_i) + m \\
&\quad \text{da es maximal } n \text{ initiale Zeichenvergleiche geben kann} \\
&\leq n + 4 \sum_{i=1}^{k-1} \sigma_i + m \\
&\quad \text{da die Summe der Shifts maximal } n \text{ sein kann} \\
&\quad \text{(der Anfang von } s \text{ bleibt innerhalb von } t) \\
&\leq n + 4n + m \\
&= 5n + m.
\end{aligned}
$$

Damit ist der Satz bewiesen. ■

Beweis von Lemma 2.19: Zum Beweis des obigen Lemmas nehmen wir also an, dass ℓ erfolgreiche und ein erfolgloser Zeichenvergleich stattgefunden haben. Wir unterscheiden jetzt den darauf folgenden Shift in Abhängigkeit seiner Länge im Verhältnis zu ℓ. Ist $\sigma \geq \lceil \ell/4 \rceil$, dann heißt der Shift *lang*, andernfalls ist $\sigma \leq \lceil \ell/4 \rceil - 1$ und der Shift heißt *kurz*.

Fall 1: Shift σ ist lang, d.h. $\sigma \geq \lceil \ell/4 \rceil$:

$$
\begin{aligned}
\text{Anzahl der ausgeführten Zeichenvergleiche} &= \ell + 1 \\
&\leq 1 + 4 \cdot \lceil \ell/4 \rceil \\
&\leq 1 + 4 \cdot \sigma \\
&\leq I + 4 \cdot \sigma
\end{aligned}
$$

Die letzte Ungleichung folgt aus der Tatsache, dass es bei jedem Versuch immer mindestens einen initialen Zeichenvergleich geben muss (zu Beginn und nach einem Shift wird dass letzte Zeichen von s mit einem Zeichen aus t verglichen, das vorher noch an keinem Zeichenvergleich beteiligt war).

Fall 2: Shift σ ist kurz, d.h. $\sigma \leq \ell/4$.

Wir zeigen zuerst, dass dann das Ende von s periodisch sein muss, d.h. es gibt ein $\alpha \in \Sigma^+$ und ein $k \in \mathbb{N}$, so dass s mit α^k enden muss, wobei $k \geq 5$. Betrachten wir dazu die folgende Skizze in Abbildung 2.34.

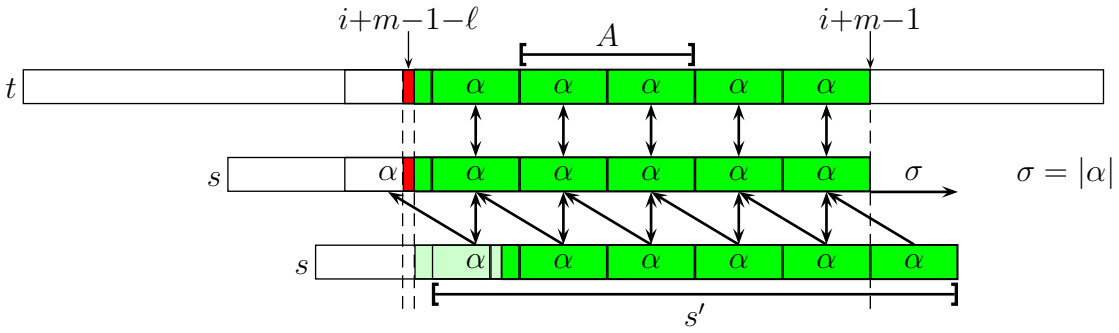


Abbildung 2.34: Skizze: Periodisches Ende von s bei kurzen Shifts

Wir wissen, dass der Shift der Länge σ kurz ist. Wegen der Strong-Good-Suffix-Rule wissen wir, dass im Intervall $[i+m-\ell : i+m-1]$ in t die Zeichen mit der verschobenen Zeichenreihe s übereinstimmen. Aufgrund der Kürze des Shifts der Länge σ folgt, dass das Suffix α von s der Länge σ mindestens $\lfloor \ell/\sigma \rfloor$ mal am Ende der Übereinstimmung von s und t vorkommen muss. Damit muss α also mindestens $k := 1 + \lfloor \ell/\sigma \rfloor \geq 5$ mal am Ende von s auftreten (siehe auch obige Abbildung). Nur falls das Wort s kürzer als $k \cdot \sigma$ ist, ist s ein Suffix von α^k (aber $|s| \geq 1 + \ell$). Sei im Folgenden s' das Suffix der Länge $k \cdot \sigma$ von s, falls $|s| \geq k \cdot \sigma$ ist, und $s' = s$ sonst. Wir halten noch fest, dass sich im Suffix s' die Zeichen im Abstand von σ Positionen wiederholen.

05.06.19

Wir werden jetzt mit Hilfe eines Widerspruchbeweises zeigen, dass die Zeichen in t an den Positionen im Intervall $A := [i+m-(k-2)\sigma : i+m-2\sigma-1]$ bislang noch an keinem Zeichenvergleich beteiligt waren. Dazu betrachten wir frühere Versuche, die Zeichenvergleiche im Abschnitt A hätten ausführen können.

Zuerst betrachten wir einen früheren Versuch, bei dem mindestens σ erfolgreiche Zeichenvergleiche ausgeführt worden sind. Dazu betrachten wir die Skizze in Abbildung 2.35 a), wobei der betrachtete Versuch oben dargestellt ist. Da mindestens σ erfolgreiche Zeichenvergleiche ausgeführt wurden, folgt aus der Periodizität von s', dass alle Zeichenvergleiche bis zur Position $i+m-\ell$ erfolgreich sein müssen. Der Zeichenvergleich an Position $i+m-\ell-1$ muss hingegen erfolglos sein, da auch der ursprüngliche Versuch, s an Position i in t zu finden, an Position $i+m-\ell-1$ erfolglos war. Wir behaupten nun, dass dann jeder sichere Shift gemäß der Strong-Good-Suffix-Rule die Zeichenreihe s auf eine Position größer als i verschoben hätte.

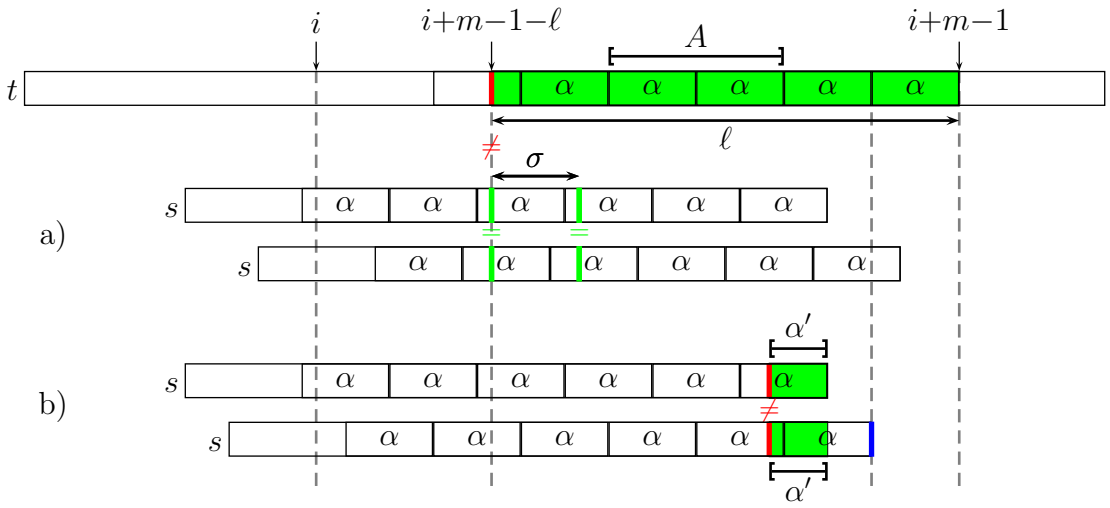


Abbildung 2.35: Skizze: Mögliche frühere initiale Vergleich im Bereich A

Nehmen wir an, es gäbe einen kürzeren sicheren Shift (wie in der Skizze in Abbildung 2.35 a) darunter dargestellt). Dann müsste das Zeichen an Position $i+m-\ell-1$ in t mit einem Zeichen aus s' verglichen werden. Dort steht im verschobenen s' aber dasselbe Zeichen wie beim erfolglosen Zeichenvergleich des früheren Versuchs, da sich in s' die Zeichen alle σ Zeichen wiederholen. Damit erhalten wir einen Widerspruch zur Strong-Good-Suffix-Rule.

Es bleiben noch die früheren Versuche, die weniger als σ erfolgreiche Zeichenvergleiche ausgeführt haben. Da wir nur Versuche betrachten, die Zeichenvergleiche im Abschnitt A ausführen, muss der Versuch in t an einer Startposition im Intervall $[i-(k-2)\sigma : i-\sigma-1]$ von rechts nach links begonnen haben. Nach Wahl von A muss der erfolglose Zeichenvergleich auf einer Position größer als $i+m-\ell-1$ erfolgen. Betrachte hierzu die erste Zeile in der Abbildung 2.35 b). Sei α' das Suffix von α (und damit von s' bzw. s), in dem die erfolgreichen Zeichenvergleiche stattgefunden haben. Seien nun $x \neq y$ die Zeichen, die den Mismatch ausgelöst haben, wobei x unmittelbar vor dem Suffix α' in s' steht. Offensichtlich liegt α' völlig im Intervall $[i+m-\ell : i+m-\sigma-1]$ von t.

Wir werden jetzt zeigen, dass ein Shift auf $i-\sigma$ (wie in Abbildung 2.35 b) darunter dargestellt) zulässig ist. Die alten erfolgreichen Zeichenvergleiche stimmen mit dem Teilwort in s' überein. Nach Voraussetzung steht an der Position unmittelbar vor α' in s' das Zeichen x und an der Position des Mismatches in s' das Zeichen y. Damit ist ein Shift auf $i-\sigma$ zulässig. Da dies aber nicht notwendigerweise der kürzeste sein muss, erfolgt ein sicherer Shift auf eine Position kleiner oder gleich $i-\sigma$.

Erfolgt ein Shift genau auf Position $i-\sigma$, dann ist der nächste Versuch bis zur Position $i+m-\ell-1$ in t erfolgreich. Da wir dann also mindestens σ erfolgreiche Zeichenvergleiche ausführen, folgt, wie oben erläutert, ein Shift auf eine Position

größer als i (oder der Versuch wäre erfolgreich abgeschlossen worden). Andernfalls haben wir einen Shift auf Position kleiner als $i - \sigma$. Aber auch dann gilt, dass in allen folgenden Fällen ein Shift genau auf die Position $i-\sigma$ zulässig bleibt. Egal, wie viele Shifts ausgeführt werden, irgendwann kommt ein Shift auf die Position $i - \sigma$. Also erhalten wir letztendlich immer einen Shift auf eine Position größer als i, aber nie einen Shift auf genau die Position i.

Damit ist bewiesen, dass bei einem kurzen Shift die Zeichen im Abschnitt A von t zum ersten Mal verglichen worden sind. Da auch der Zeichenvergleich des letzten Zeichens von s mit einem Zeichen aus t ein initialer gewesen sein musste, folgt dass mindestens $1 + |A|$ initiale Zeichenvergleiche ausgeführt wurden. Da $|A| \geq \ell - 4\sigma$, erhalten wir die gewünschte Abschätzung:

$$1 + \ell = (1 + |A|) + (\ell - |A|) \leq I + (\ell - (\ell - 4\sigma)) \leq I + 4\sigma.$$

Da wie schon weiter oben angemerkt, der Zeichenvergleich des letzten Zeichens von s mit dem entsprechenden Zeichen in t immer initial sein muss und alle Zeichenvergleiche im Bereich A initial sind, folgt damit die Behauptung des Lemmas. ■

Fassen wir das Gesamtergebnis noch einmal im folgenden Satz zusammen (inklusive der Zeichenvergleiche zur Bestimmung der Shift-Tabelle).

Theorem 2.21 *Der Boyer-Moore Algorithmus benötigt für das erste Auffinden des Suchmusters $s \in \Sigma^m$ in einem Text $t \in \Sigma^n$ maximal $5n + 3m$ Zeichenvergleiche.*

Mit Hilfe einer besseren, allerdings für die Vorlesung zu aufwendigen Analyse, kann man folgendes Resultat herleiten.

Theorem 2.22 *Der Boyer-Moore Algorithmus benötigt maximal $3(n+m)$ Zeichenvergleiche, um zu entscheiden, ob eine Zeichenreihe der Länge m in einem Text der Länge n enthalten ist.*

Die Behauptung im obigen Satz ist scharf, da man Beispiele konstruieren kann, bei denen mindestens $3n - o(n)$ Zeichenvergleiche bei einer Suche mit der Strong-Good-Suffix-Rule beim Boyer-Moore-Algorithmus ausgeführt werden müssen. Damit ist der Boyer-Moore-Algorithmus zwar im worst-case etwas schlechter als der Knuth-Morris-Pratt-Algorithmus, dafür jedoch im average-case deutlich besser.

Wir erinnern nochmals, dass unsere Analyse nur für den Fall einer erfolglosen Suche oder dem ersten Auffinden von s in t gültig ist. Das Beispiel $s = a^m$ und $t = a^n$ zeigt, dass der Boyer-Moore-Algorithmus beim Auffinden aller Vorkommen von s

in t wieder quadratischen Zeitbedarf (d.h. $\Theta(nm)$) bekommen kann (im Gegensatz zum KMP-Algorithmus, der immer seine Laufzeit von $2n + m$ beibehält).

Diese worst-case Laufzeit lässt sich jedoch vermeiden, wenn man einen kleinen, aber wirkungsvollen Trick nach Zvi Galil anwendet. Nach einem erfolgreichen Test $s = t_i \cdots t_{i+m-1}$ verschieben wir das Muster gemäß der Strong-Good-Suffix-Rule. Man überlegt sich leicht, dass man dann denselben Shift erhält, wie beim KMP-Algorithmus. Nach diesem erfolgreichen Test werden keine Vergleiche mehr mit Zeichen in t an einer Position kleiner als $i + m$ ausgeführt. Da wir ja wissen, dass $s = t_i \cdots t_{i+m-1}$ gilt, werden Zeichenvergleiche im Bereich einfach ausgelassen, da wir hier die Übereinstimmung kennen. Sobald ein erfolgloser Zeichenvergleich stattgefunden hat, vergessen wir diese Begrenzung wieder, siehe auch Abbildung 2.36.

Man kann nun zeigen, dass bei nicht periodischen Suchworten oder Suchworten mit großer Periode die Suchzeit auch ohne Änderung linear bleibt, da es einfach nur wenige Treffer geben kann. Für stark periodische Suchwörter (also Wörter mit kleiner Periode) kann man beweisen, dass die Einsparung der Zeichenvergleiche am Beginn des Suchworts genügen um die lineare Laufzeit beizubehalten. Für Details sei der Leser auf die Originalliteratur von Zvi Galil oder das Buch von Maxime Crochemore und Wojciech Rytter verwiesen.

bool Boyer_Moore_Galil (char $t[]$, int n, char $s[]$, int m)

```
begin
    int S[m + 1];
    compute_shift_table(S, m, s);
    int i := 0, j := m − 1; start := 0;
    int Border := border[m];
    while (i ≤ n − m) do
        while (t[i + j] = s[j]) do
            if (j = start) then
                output i;
                break;
            j−−;
        if (j = start) then start := Border;
        else start := 0;
        i := i + S[j];
        j := m − 1;
    return FALSE;
end
```

Abbildung 2.36: Algorithmus: Boyer-Moore mit Galil-Trick

2.4.5 Bad-Character-Rule

Eine andere mögliche Beschleunigung beim Boyer-Moore Algorithmus stellt, wie in der Einleitung zu diesem Abschnitt bereits angedeutet, die so genannte *Bad-Character-Rule* dar. Bei einem Mismatch von s_j mit t_{i+j} verschieben wir das Suchwort s so, dass das rechteste Vorkommen von t_{i+j} in s genau unter t_{i+j} zu liegen kommt (siehe auch Abbildung 2.37). Falls wir die Zeichenreihe s dazu nach links ver-

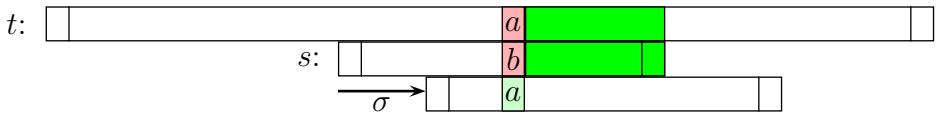


Abbildung 2.37: Skizze: Bad-Character-Rule

schieben müssten, schieben wir s stattdessen einfach um eine Position nach rechts. Falls das Zeichen in s gar nicht auftritt, so verschieben wir s so, dass das erste Zeichen von s auf der Position $i + j + 1$ in t zu liegen kommt. Hierfür benötigt man eine separate Tabelle mit $|\Sigma|$ Einträgen:

$$\mathrm{bc}[a] = \max\{k \in [-1 : m-1] \ : \ k = -1 \vee s_k = a\}\,.$$

Bei einem Mismatch an Position $j \in [0 : m-1]$ in s mit $t_{i+j} = a$ ergibt sich dann ein Shift um $\max\{1, j - \mathrm{bc}[a]\}$.

Diese Regel macht insbesondere bei kleinen Alphabeten Probleme, wenn beim vorigen Versuch schon einige Zeichenvergleiche ausgeführt wurden. Denn dann ist das gesuchte Zeichen in s meistens rechts von der aktuell betrachteten Position. Hier kann man die so genannte *Extended-Bad-Character-Rule* verwenden. Bei einem Mismatch von s_j mit t_{i+j} sucht man nun das rechteste Vorkommen von t_{i+j} im Präfix

BC-Boyer-Moore (char $t[]$, int n, char $s[]$, int m)

```
int i := 0, j := m − 1;
compute_ebc(ebc,s,m);
while (i ≤ n − m) do
    while (t[i + j] = s[j]) do
        if (j = 0) then return TRUE;
        j−−;
    i := i + j − ebc[t[i + j], j];
    j := m − 1;
return FALSE;
```

Abbildung 2.38: Algorithmus: Boyer-Moore nur mit bad character rule

$s_0 \cdots s_{j-1}$ (also links von s_j) und verschiebt nun s so nach rechts, dass die beiden Zeichen übereinander zu liegen kommen. Damit erhält man immer recht große Shifts. Einen Nachteil erkauft man sich dadurch, dass man für diese Shift-Tabelle nun $m \cdot |\Sigma|$ Einträge benötigt, was aber bei kleinen Alphabetgrößen nicht sonderlich ins Gewicht fällt.

$$\text{ebc}[a, j] = \max\{k \in [-1 : j-1] \,:\, k = -1 \vee s_k = a\}\,.$$

Bei einem Mismatch an Position $j \in [0 : m-1]$ in s mit $t_{i+j} = a$ ergibt sich dann ein Shift um $j - \text{ebc}[a, j]$ (siehe auch Abbildung 2.38).

In der Praxis wird man in der Regel sowohl die Strong-Good-Suffix-Rule als auch die Extended-Bad-Character-Rule verwenden. Hierbei darf der größere der beiden vorgeschlagenen Shifts ausgeführt werden. Das worst-case Laufzeitverhalten verändert sich dadurch nicht, aber die average-case Laufzeit wird besser. Wir weisen noch darauf hin, dass sich unser Beweis für die Laufzeit nicht einfach auf diese Kombination erweitern lässt.

2.5 Z-Box-Algorithmen

In diesem Abschnitt wollen wir noch eine alternative Implementierung der Algorithmen von Knuth, Morris und Pratt bzw. Boyer und Moore angeben. Diese basieren auf so genannten Z-Boxen

2.5.1 Z-Boxen

Als erstes definieren wir, was wir unter einer Z-Box verstehen wollen.

Definition 2.23 *Sei $s = s_0 \cdots s_{m-1} \in \Sigma^m$. Für jeden Index $i \in [1 : m-1]$ ist $Z_i = \max\{j \in [0 : m] \,:\, s_i \cdots s_{i+j-1} = s_0 \cdots s_{j-1}\}$ der* Z-Wert *an Position i. Das Teilwort $s_i \cdots s_{i+Z_i-1} \neq \varepsilon$ heißt auch* Z-Box *ab der Position i, wenn es nichtleer ist.*

Für das Wort $s = ananas$ ist die Folge der Z-Werte gegeben wie folgt: $Z_1 = 0$, $Z_2 = 3$, $Z_3 = 0$, $Z_4 = 1$ sowie $Z_5 = 0$. Für das Wort $s = abaabaabab$ ist die Folge der Z-Werte gegeben wie folgt: $Z_1 = 0$, $Z_2 = 1$, $Z_3 = 6$, $Z_4 = 0$, $Z_5 = 1$, $Z_6 = 3$, $Z_7 = 0$, $Z_8 = 2$ sowie $Z_9 = 0$.

Als erstes werden wir zeigen, wie man die Z-Werte für einen Wort der Länge m effizient berechnen kann. Aus der Definition folgt sofort ein Algorithmus mit Laufzeit $O(m^2)$. Wir werden jetzt einen Algorithmus mit linearer Laufzeit vorstellen.

Wir nehmen an, dass wir die Z-Werte bis zur Position $k-1$ bereits berechnet haben, und zeigen, wie wir mit deren Hilfe den Wert Z_k bestimmen können. Z_1 können wir in jedem Fall mit der naiven Methode mit Z_1+1 Zeichenvergleichen bestimmen Für den allgemeinen Fall nehmen wir an, dass wir wissen, wo das rechteste Ende einer Z-Box bisher ist, d.h. wir setzen

$$r := r_k := \max\{i + Z_i - 1 \ : \ i \in [1 : k-1]\}\,.$$

Für das Folgende nehmen wir auch noch an, dass wir den Anfang dieser Z-Box kennen und bezeichen die Position mit $\ell := \ell_k$, d.h. es gilt $\ell + Z_\ell - 1 = r$. Beachte, dass es für ein r_k durchaus mehrere ℓ_k geben kann. Wir wählen dann hierbei immer das kleinstmögliche ℓ (dies spielt für den Algorithmus keine Rolle, es muss nur $\ell < k$ gelten). Wenn es noch keine Z-Box gibt, dann setzen wir einfach $\ell = \ell_k = r = r_k = 0$.

Im Beispiel *abaabaabab* gilt für $k = 7$, dass $r = r_7 = 8$. An dieser Stelle enden zwei Z-Boxen, nämlich *aba* ab Position 6 und *abaaba* ab Position 3. Wir setzen dann $\ell = \ell_7 = 3$.

Beachte, dass also das Präfix $s_0 \cdots s_{Z_\ell - 1} = s_\ell \cdots s_r$ von s also ab Position ℓ nochmal in s vorkommt. Diese Notation ist in Abbildung 2.39 noch einmal illustriert.

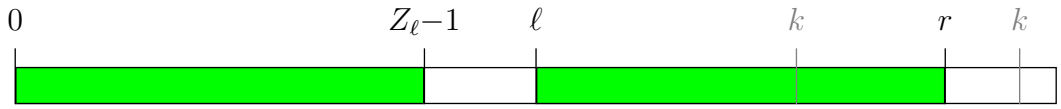


Abbildung 2.39: Skizze: Bezeichnung für die Berechnung von Z-Boxen

Für die Berechnung des Z-Wertes an Position k unterscheiden wir zwei Fälle, je nachdem, ob die Position k hinter der bislang rechtesten Z-Box liegt oder nicht.

Fall 1 ($k > r$): In diesem Fall befindet sich k also hinter dem Ende der bislang rechtesten Z-Box. Wie bei der Berechnung von Z_1 ermitteln wir Z_k durch $Z_k + 1$ Zeichenvergleiche der Teilwörter ab Position 0 bzw. k. Siehe hierzu auch die Skizze in Abbildung 2.40.

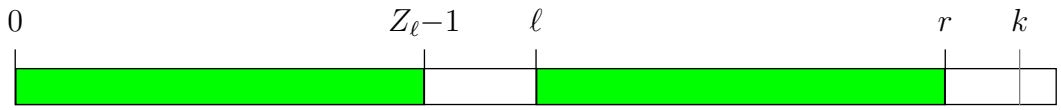


Abbildung 2.40: Skizze: Fall 1

Falls wir eine Z-Box (also einen Z-Wert größer als 0) bestimmt haben, müssen wir die rechteste Z-Box (und die Werte ℓ bzw. r) aktualisieren.

Fall 2 ($k \leq r$): Also kommt das Teilwort $s_k \cdots s_r$ ab Position k (als Suffix der rechtesten Z-Box) auch noch einmal ab Position $k-\ell$ vor, da $s_\ell \cdots s_r = s_0 \cdots s_{Z_\ell - 1}$. Siehe hierzu auch die Skizze in Abbildung 2.41. Wir betrachten nun die Z-Box ab

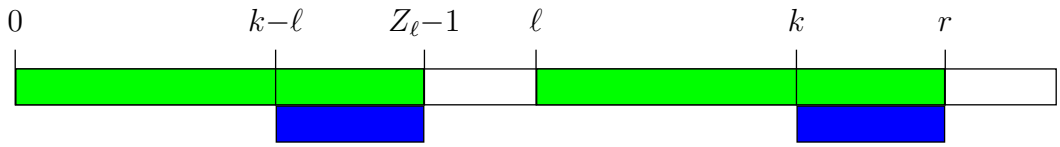


Abbildung 2.41: Skizze: Fall 2

Position $k-\ell$. Wir unterscheiden jetzt zwei Unterfälle, je nachdem, ob diese Z-Box ab Position $k-\ell$ vor oder nach Position $Z_\ell - 1$ endet.

Fall 2a: Wenn diese Z-Box ab Position $k-\ell$ vor der Position $Z_\ell - 1$ endet, ist also

$$k - \ell + Z_{k-\ell} - 1 < Z_\ell - 1.$$

Da $(Z_\ell - 1) - (k - \ell) + 1 = r - k + 1$, folgt $Z_{k-\ell} - 1 < r - k$. Siehe hierfür auch die Skizze in Abbildung 2.42. Dann folgt aber sofort, dass der Z-Wert an Position k genau $Z_{k-\ell}$ ist. Die rechteste Z-Box ist dann auch weiterhin die rechteste Z-Box.

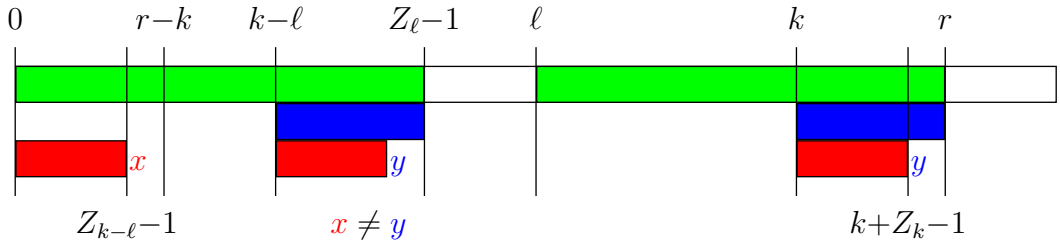


Abbildung 2.42: Skizze: Fall 2a

12.06.19

Fall 2b: Im anderen Fall endet die Z-Box ab Position $k-\ell$ an oder nach der Position $Z_\ell - 1$. Dies ist in Abbildung 2.43 illustriert.

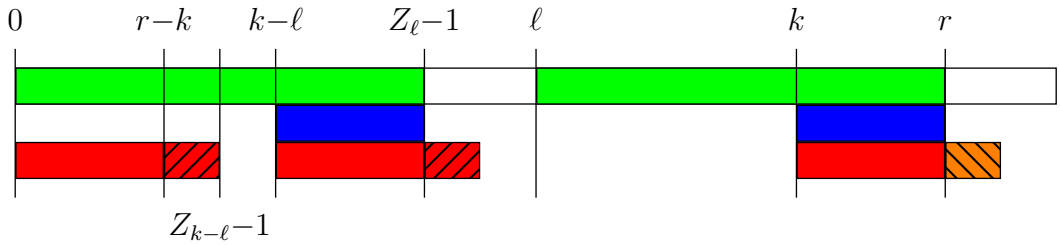


Abbildung 2.43: Skizze: Fall 2b

```
ZBoxes (char s[], int m)
begin
    int ℓ := r := 0;                                   /* no Z-box seen so far */
    int i := 1;                          /* Note that i will never be decremented */
    for (k := 1; k < m; k++) do
        if (k > r) then                                                  /* case 1 */
            i := k;
            while ((i < m) && (s[i] = s[i − k])) do
                i++;
            Z[k] := i − k;
            if (Z[k] > 0) then
                ℓ := k;
                r := i − 1;
        else
            if (Z[k − ℓ] < r − k + 1) then                               /* case 2a */
                Z[k] := Z[k − ℓ];
            else                                                         /* case 2b */
                i := r + 1;
                while ((i < m)&&(s[i] = s[i − k])) do
                    i++;
                Z[k] := i − k;
                if (i − 1 > r) then
                    ℓ := k;
                    r := i − 1;
end
```

Abbildung 2.44: Algorithmus: Berechnung der Z-Boxen

Der schraffierte Bereich ist nun nach der Position r, also nach der rechtesten bislang bekannten Z-Box. Von daher wissen wir nicht, ob in diesem Bereich eine Übereinstimmung herrscht oder nicht. Also müssen wird ab Position $r + 1$ wieder zeichenweise vergleichen. Abschließend müssen wir noch testen, ob wir eine neue rechtestete Z-Box gefunden haben (auf jeden Fall gilt hier $Z[k] \geq 1$).

Dies führt uns nun zu dem in Abbildung 2.44 angegeben Algorithmus. Beachte hierbei, dass die Berechnung von Z_1 wie bereits erwähnt durch den naiven Ansatz gelöst wird. Durch die Initialisierung von $r = 0$ wird in der for-Schleife für $k = 1$ die Anweisungen im if-Zweig mit $k = 1$ aufgerufen und somit die while-Schleife beginnend mit $i = 1$ durchlaufen. Somit wird dort die naive Konstruktion implementiert.

Für die Laufzeitanalyse unterscheiden wir auch hier wieder erfolgreiche und erfolglose Zeichenvergleiche. Nach jedem erfolglosen Zeichenvergleich wird ein Z-Wert festgelegt. Also kann es maximal $m-1$ erfolglose Zeichenvergleiche geben. Nach einem erfolgreichen Zeichenvergleich wird das Zeichen s_i, das im Algorithmus betrachtet wird, nie wieder als erstes Zeichen an einem Zeichenvergleich teilnehmen. Da das erste Zeichen s_0 nie als erstes Zeichen in einem Zeichenvergleich beteiligt ist, kann es also maximal $m-1$ erfolgreiche Zeichenvergleiche geben.

Halten wir das Ergebnis im folgenden Lemma fest.

Lemma 2.24 *Für ein Wort $s \in \Sigma^m$ können die Z-Werte Z_i für $i \in [1:m-1]$ mit maximal $2m-2$ Zeichenvergleichen bestimmt werden.*

In Abbildung 2.45 wird der Algorithmus anhand unseres Beispiel $s = abaabaabab$ illustriert. Hierbei ist die rechteste Z-Box orange umrandet, die betrachtete Position rot eingefärbt und neue Zeichenvergleiche grün umrandet. Bei der Betrachtung des Falls 1 bzw. 2b muss die rechteste Z-Box überprüft werden, also für $k \in \{1,2,3,6,8\}$. Für $k=2$, $k=3$ bzw. $k=8$ wird dabei die rechteste Z-Box aktualisiert: $(\ell_3, r_3) = (2,2)$, $(\ell_3, r_3) = (3,8)$, bzw. $(\ell_9, r_9) = (8,9)$

0	1	2	3	4	5	6	7	8	9	
a	b	a	a	b	a	a	b	a	b	
a	b	a	a	b	a	a	b	a	b	F. 1: $s_1 \neq s_0 \Rightarrow Z_1 = 0$
a	b	a	a	b	a	a	b	a	b	F. 1: $s_2 = s_0 \wedge s_3 \neq s_1 \Rightarrow Z_2 = 1$
a	b	a	a	b	a	a	b	a	b	F. 1: $s_3 \cdots s_8 = s_0 \cdots s_5 \wedge s_9 \neq s_6 \Rightarrow Z_3 = 6$
a	b	a	a	b	a	a	b	a	b	F. 2a: $Z_{k-\ell} = 0 < 5 = r-k+1 \Rightarrow Z_4 = Z_{4-3} = 0$
a	b	a	a	b	a	a	b	a	b	F. 2a: $Z_{k-\ell} = 1 < 4 = r-k+1 \Rightarrow Z_5 = Z_{5-3} = 1$
a	b	a	a	b	a	a	b	a	b	F. 2b: $Z_{k-\ell} = 6 \not< 3 = r-k+1 \wedge s_9 \neq s_3 \Rightarrow Z_6 = 3$
a	b	a	a	b	a	a	b	a	b	F. 2a: $Z_{k-\ell} = 0 < 2 = r-k+1 \Rightarrow Z_7 = Z_{7-3} = 0$
a	b	a	a	b	a	a	b	a	b	F. 2b: $Z_{k-\ell} = 1 \not< 1 = r-k+1 \wedge s_9 = s_1 \Rightarrow Z_8 = 2$
a	b	a	a	b	a	a	b	a	b	F. 2a: $Z_{k-\ell} = 0 < 1 = r-k+1 \Rightarrow Z_9 = Z_{9-8} = 0$

Abbildung 2.45: Beispiel: Berechnung der Z-Werte für $s = abaabaabab$

2.5.2 Knuth-Morris-Pratt-Algorithmus mittels Z-Boxen

Schauen wir uns nochmal die Situation der zulässigen Shifts beim KMP-Algorithmus in Abbildung 2.46 an. Ein Shift um σ Zeichen bei einem Mismatch an Position j in s ist also genau dann zulässig, wenn $Z_\sigma + \sigma = j$.

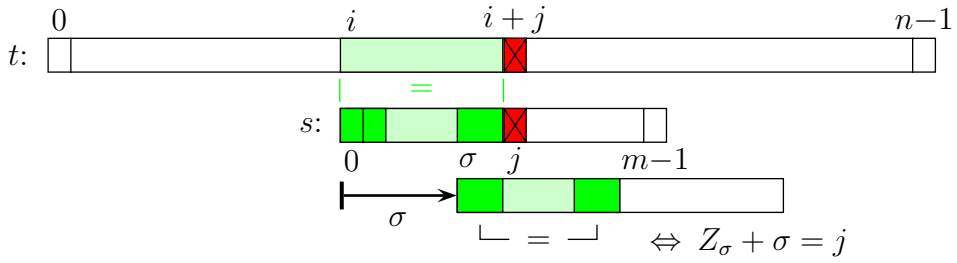


Abbildung 2.46: Skizze: Shift bei KMP

Daraus folgt für die Bestimmung der Shift-Tabelle S für den Algorithmus von Knuth, Morris und Pratt, da wir ja den minimalen zulässigen Shift als sicheren Shift wählen müssen:

$$S[j] = \begin{cases} \min\{j, \sigma \in [1:m-1] \,:\, Z_\sigma + \sigma = j\} & \text{falls } j > 0 \\ 1 & \text{sonst.} \end{cases}$$

Für eine analoge Definition einer Border-Tabelle namens $border'$ für den KMP-Algorithmus erhalten wir die folgenden Werte:

$$border'[j] = j - S[j].$$

Der auf den Z-Boxen basierende Algorithmus für die Berechnung der Shift-Tabelle für den Algorithmus Knuth, Morris und Pratt ist in Abbildung 2.47 noch einmal angegeben. Beachte, dass hier im Gegensatz zum bisherigen KMP-Algorithmus eine starke Shift-Regel implementiert wird, d.h. auf die Position, die ein Mismatch ausgelöst hat, wird ein anderes Zeichen geschoben als das, das den Mismatch ausgelöst hat.

computeShiftTableKMP (char $s[]$, int m)

```
begin
    Z := ZBoxes(s, m);
    for (j := 1; j < m; j++) do
        S[j] := j;
    S[0] := 1;
    for (σ := m − 1; σ > 0; σ−−) do
        j := Z[σ] + σ;
        S[j] := min(S[j], σ);                    /* i.e., S[j] := σ */
end
```

Abbildung 2.47: Algorithmus: Berechnung der Shift-Tabelle für KMP mit Z-Boxen

2.5.3 Boyer-Moore-Algorithmus mittels Z-Boxen

Wir wollen jetzt noch eine Implementierung des Algorithmus von Boyer und Moore basierend auf der Strong-Good-Suffix-Rule mittels Z-Boxen angeben. Dafür definieren wir zuerst die gespiegelten Z-Werte.

Definition 2.25 *Sei $s = s_0 \cdots s_{m-1} \in \Sigma^m$ und sei $s^R = s_{m-1} \cdots s_0$ der zugehörige gespiegelte String. Seien weiter Z_i^R die Z-Werte für s^R. Dann sind die* gespiegelten Z-Werte *Z'_i für $i \in [0 : m-2]$ von s gegeben durch $Z'_i = Z^R_{(m-1)-i}$.*

Die Definition der gespiegelten Z-Werte sind in der Abbildung 2.48 noch einmal illustriert.

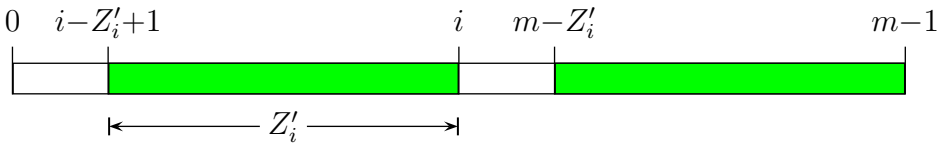


Abbildung 2.48: Skizze: Die gespiegelten Z-Werte)

Um für die Berechnung der Shift-Tabelle mithilfe der gespiegelten Z-Werte angeben zu können, betrachten wir noch einmal die beiden Fälle der Strong-Good-Suffix-Rule, die in Abbildung 2.49 noch einmal illustriert sind.

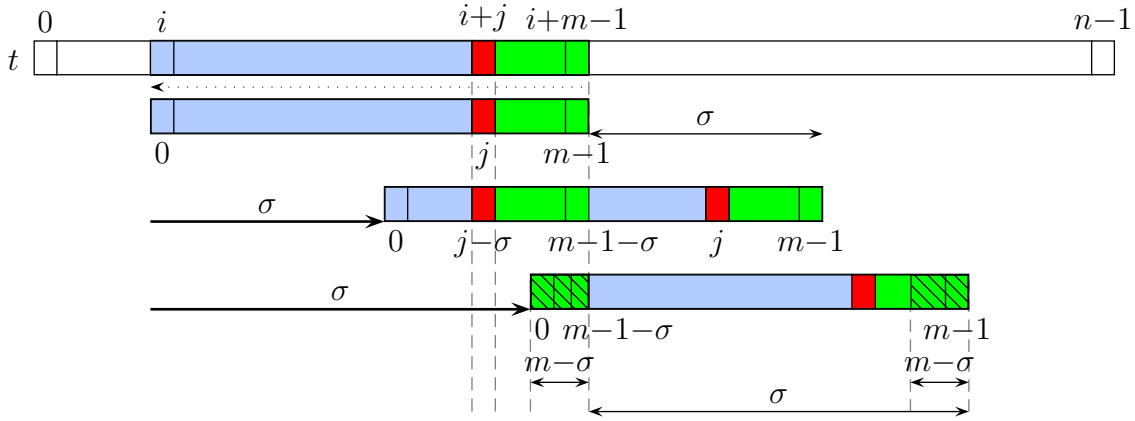


Abbildung 2.49: Skizze: Zulässige Shifts bei Boyer-Moore (Strong-Good-Suffix-Rule)

Ein kurzer Shift um σ ist zum einen zulässig, wenn $Z'_{m-1-\sigma} = m-1-j$ und $\sigma \leq j$ gilt (siehe oberer Teil der Abbildung 2.49). Das heißt ein Shift um σ Zeichen ist dann zulässig, wenn der Mismatch an Position $j = m-1-Z'_{m-1-\sigma}$ aufgetreten ist.

Ein langer Shift um σ ist zum anderen zulässig, wenn $Z'_{m-1-\sigma} = m - \sigma$ und $\sigma > j$ gilt (siehe unterer Teil der Abbildung 2.49).

Der vollständige Algorithmus zur Berechnung der Shift-Tabelle basierend auf der Strong-Good-Suffix-Rule mithilfe der gespiegelten Z-Werte ist in Abbildung 2.50 angegeben, wobei die beiden Fällen wieder in zwei for-Schleifen die Aktualisierung der Shift-Tabelle bewirken.

computeShiftTableBM (char $s[]$, int m)

```
begin
    Z^R := ZBoxes(s^R, m);
    for (i := 0; i < m; i++) do  Z'[i] = Z^R[m − 1 − i];

    for (j := 0; j < m; j++) do  S[j] = m;

    for (σ := 1; σ ≤ m; σ++) do
        j := m − 1 − Z'[m − 1 − σ];
        if (σ ≤ j) then
            S[j] := min(S[j], σ);

    j := 0;
    for (σ := 1; σ < m; σ++) do
        if (Z'_{m−1−σ} = m − σ) then
            while (j < σ) do
                S[j] := min(S[j], σ);
                j++;
end
```

Abbildung 2.50: Algorithmus: Berechnung der Shift-Tabelle für BM mittels Z-Boxen

2.6 Der Algorithmus von Karp und Rabin (*)

Dieser Abschnitt ist nur der Vollständigkeit halber im Skript enthalten und wurde in den Sommersemestern ab 2005 nicht mehr behandelt. Im Folgenden wollen wir einen Algorithmus zum Suchen in Texten vorstellen, der nicht auf dem Vergleichen von einzelnen Zeichen beruht. Wir werden vielmehr das Suchwort bzw. Teile des zu durchsuchenden Textes als Zahlen interpretieren und dann aufgrund von numerischer Gleichheit auf das Vorhandensein des Suchwortes schließen.

2.6.1 Ein numerischer Ansatz

Der Einfachheit halber nehmen wir im Folgenden an, dass $\Sigma = \{0, 1\}$ ist. Die Verallgemeinerung auf beliebige k-elementige Alphabete, die wir ohne Beschränkung der Allgemeinheit als $\Sigma = \{0, \ldots, k-1\}$ annehmen dürfen, sei dem Leser überlassen.

Da $\Sigma = \{0, 1\}$, kann jedes beliebige Wort aus Σ^* als Binärzahl interpretiert werden. Somit kann eine Funktion V angegeben werden, welche jedem Wort aus Σ^* ihre zugehörige Zahl zuordnet.

$$V : \{0,1\}^* \to \mathbb{N}_0 : V(s) = \sum_{i=0}^{|s|-1} 2^i \cdot s_i.$$

Für $\Sigma = \{0, \ldots, k-1\}$ gilt dementsprechend:

$$V_k : [0 : k-1]^* \to \mathbb{N}_0 : V_k(s) = \sum_{i=0}^{|s|-1} k^i \cdot s_i.$$

Beispielsweise gilt: $V_2(0011) = 12$; $V_2(1100) = V_2(11) = 3$. Man beachte, dass die hier definierte Funktion V im Allgemeinen nicht injektiv ist. Wir werden aber im Folgenden bei der Verwendung von V ihren Definitionsbereich so weit einschränken, dass sie hierauf injektiv ist.

Beobachtung 2.26 *s ist ein Teilwort von t an der Stelle i, falls es ein i gibt, $i \in [0 : n-m]$, so dass $V(s) = V(t_{i,m})$, wobei $t_{i,m} := t_i \cdots t_{i+m-1}$.*

Wird also s in t gesucht, müssen immer nur zwei Zahlen miteinander verglichen werden. Dabei wird $V(s)$ mit jeder Zahl $V(t_{i,m})$ für alle $i \in [0 : n-m]$ verglichen, die durch das Teilwort von t der Länge m an der Stelle i repräsentiert wird. Stimmen die beiden Zahlen überein, ist s in t an der Stelle i enthalten. Diese einfache Idee ist im naiven Algorithmus in Abbildung 2.51 wiedergegeben.

```
bool Naiv3 (char s[], int m, char t[], int n)
begin
    for (i := 0; i ≤ n − m; i++) do
        if (V(s) = V(t_i, m)) then
            return TRUE;
end
```

Abbildung 2.51: Algorithmus: Naive numerische Textsuche

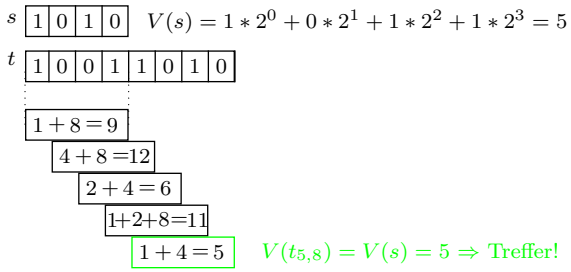


Abbildung 2.52: Beispiel: Numerische Suche von 1010 in 100110101

In Abbildung 2.52 ist diese naive numerische Suche an einem Beispiel illustriert.

Wie groß ist der Aufwand, um $V(x)$ mit $x \in \Sigma^m$, $x = x_0...x_{m-1}$ zu berechnen?

$$V(x) = \underbrace{\sum_{i=0}^{m-1} x_i \cdot 2^i}_{1)} = \underbrace{x_0 + 2(x_1 + 2(x_2 + 2(x_3...x_{m-2} + 2x_{m-1})))}_{2)}$$

Die zweite Variante zur Berechnung folgt aus einer geschickten Ausklammerung der einzelnen Terme und ist auch als *Horner-Schema* bekannt.

Aufwand für 1): $\sum_{i=0}^{m-1} i + (m-1) = \Theta(m^2)$ arithmetische Operationen.

Aufwand für 2): $\Theta(m)$ arithmetische Operationen.

Somit ist die Laufzeit für die gesamte Suche selbst mit Hilfe der zweiten Methode immer noch $\Theta(nm)$. Wiederum wollen wir versuchen, dieses quadratische Verhalten zu unterbieten.

Wir versuchen zuerst festzustellen, ob uns die Kenntnis des Wertes für $V(t_{i,m})$ für die Berechnung des Wertes $V(t_{i+1,m})$ hilft:

$$\begin{aligned} V(t_{i+1,m}) &= \sum_{j=0}^{m-1} t_{i+1+j} \cdot 2^j \\ &= \sum_{j=1}^{m} t_{i+1+j-1} \cdot 2^{j-1} \\ &= \frac{1}{2} \sum_{j=1}^{m} t_{i+j} \cdot 2^j \end{aligned}$$

$$= \frac{1}{2}\left(\sum_{j=0}^{m-1} t_{i+j}\cdot 2^j - t_{i+0}\cdot 2^0 + t_{i+m}\cdot 2^m\right)$$
$$= \frac{1}{2} * (V(t_{i,m}) - t_i + 2^m t_{i+m}).$$

$V(t_{i+1,m})$ lässt sich somit mit dem Vorgänger $V(t,m)$ sehr einfach und effizient in konstanter Zeit berechnen.

Damit ergibt sich für die gesamte Laufzeit $O(n+m)$: Für das Preprocessing zur Berechnung von $V(s)$ und 2^m benötigt man mit dem Horner-Schema $O(m)$. Die Schleife selbst wird dann n-Mal durchlaufen und benötigt jeweils eine konstante Anzahl arithmetischer Operationen, also $O(n)$.

Allerdings haben wir hier etwas geschummelt, da die Werte ja sehr groß werden können und damit die Kosten einer arithmetischen Operation sehr teuer werden können.

2.6.2 Der Algorithmus von Karp und Rabin

Nun kümmern wir uns um das Problem, dass die Werte von $V(s)$ bzw. von $V(t_{i,m})$ sehr groß werden können. Um dies zu verhindern, werden die Werte von V einfach modulo einer geeignet gewählten Zahl gerechnet. Dabei tritt allerdings das Problem auf, dass der Algorithmus Treffer auswirft, die eigentlich gar nicht vorhanden sind (siehe auch das folgende Beispiel in Abbildung 2.53).

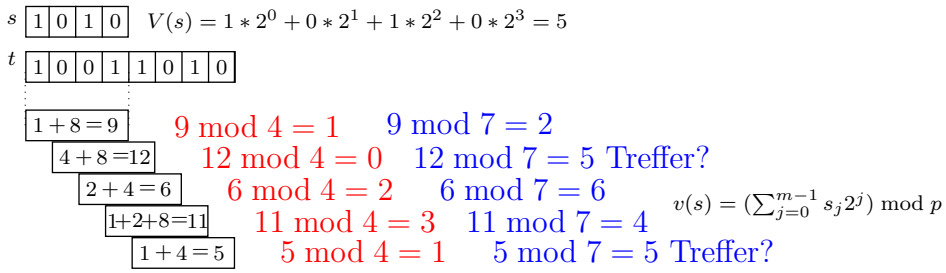


Abbildung 2.53: Beispiel: Numerische Suche von 1010 in 10011010 modulo 7

Die Zahl, durch welche modulo gerechnet wird, sollte teilerfremd zu $|\Sigma|$ sein, da ansonsten Anomalien auftreten können. Wählt man beispielsweise eine Zweierpotenz, so werden nur die ersten Zeichen des Suchwortes berücksichtigt, wohingegen die letzten Zeichen des Suchwortes bei dieser Suche überhaupt keine Relevanz haben. Um unabhängig von der Alphabetgröße die Teilerfremdheit garantieren zu können, wählt man p oft als eine Primzahl.

Um nun sicher zu sein, dass man wirklich einen Treffer gefunden hat, muss zusätzlich noch im Falle eines Treffers überprüft werden, ob die beiden Zeichenreihen auch wirklich identisch sind. Damit erhalten wir den Algorithmus von R. Karp und M. Rabin, der im Detail in Abbildung 2.54 angegeben ist. Den Wert $V(s) \bmod p$ nennt man auch einen *Fingerabdruck* (engl. *fingerprint*) des Wortes s.

bool Karp_Rabin (char $t[]$,int n, char $s[]s$, int m)

begin
  int p; `// sufficiently large prime number, but not too large ;-)`
  int $vs := v(s) \bmod p = \sum_{j=0}^{m-1} s_j \cdot 2^j \bmod p$;
  int $vt := (\sum_{j=0}^{m-1} t_j \cdot 2^j) \bmod p$;
  for ($i := 0$; $i \leq n - m$; i++) **do**
    if (($vs = vt$) && ($s = t_{i,m}$) /* Zusätzlicher Test auf Gleichheit */) **then**
      return TRUE;
    $vt := \frac{1}{2}(vt - t_i + (2^m \bmod p) - t_{i+m}) \bmod p$;
end

Abbildung 2.54: Algorithmus: Die Methode von Karp und Rabin

2.6.3 Bestimmung der optimalen Primzahl

Es stellt sich nun die Frage, wie die Primzahl, mit der modulo gerechnet wird, auszusehen hat, damit möglichst selten der Fall auftritt, dass ein „falscher Treffer“ vom Algorithmus gefunden wird.

Sei $\mathbb{P}$ ={2,3,5,7...} die Menge aller Primzahlen. Weiter sei $\pi(k) := |\mathbb{P} \cap [1 : k]|$ die Anzahl aller Primzahlen kleiner oder gleich k. Wir stellen nun erst einmal ein paar nützliche Tatsachen aus der Zahlentheorie zu Verfügung, die wir hier nur teilweise beweisen wollen.

Theorem 2.27 *Für alle $k \in \mathbb{N}$ gilt:*

$$\frac{k}{\ln(k)} \leq \pi(k) \leq 1.26\frac{k}{\ln(k)}.$$

Für den Beweis dieses Satzes verweisen auf die einschlägige Literatur zur Zahlentheorie.

Theorem 2.28 *Sei $k \geq 29$, dann gilt*

$$\prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p \geq 2^k.$$

Beweis: Mit Hilfe des vorherigen Satzes und der Stirlingschen Abschätzung, d.h. mit $n! \geq \left(\frac{n}{e}\right)^n$, folgt:

$$\prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p \geq \prod_{p=1}^{\pi(k)} p = \pi(k)! \geq \left(\frac{k}{\ln(k)}\right)! \geq \left(\frac{k}{e \cdot \ln(k)}\right)^{k/\ln(k)} \geq \left(\left(\frac{k}{e \cdot \ln(k)}\right)^{1/\ln(k)}\right)^k.$$

Da ferner gilt, dass

$$\lim_{k \to \infty} \left(\frac{k}{e \cdot \ln(k)}\right)^{1/\ln(k)} = \exp\left(\lim_{k \to \infty} \frac{\ln(k) - 1 - \ln(\ln(k))}{\ln(k)}\right) = e,$$

folgt, dass $\left(\frac{k}{e \cdot \ln(k)}\right)^{1/\ln(k)} \geq 2$ für große k sein muss und somit auch $\prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p > 2^k$. Eine genaue Analyse zeigt, dass dies bereits für $k \geq 29$ gilt. ■

Theorem 2.29 *Seien $k, x \in \mathbb{N}$ mit $k \geq 29$ und mit $x \leq 2^k$, dann besitzt x maximal $\pi(k)$ verschiedene Primfaktoren.*

Beweis: Wir führen den Beweis durch Widerspruch. Dazu seien $k, x \in \mathbb{N}$ mit $x \leq 2^k$ und mit $k \geq 29$. Wir nehmen an, dass x mehr als $\pi(k)$ verschiedene Primteiler besitzt. Seien $p_1, \ldots, p_\ell$ die verschiedenen Primteiler von x mit Vielfachheiten $m_1, \ldots, m_\ell \in \mathbb{N}$. Dann gilt unter Verwendung von $m_i \geq 1$ und der Annahme $\ell > \pi(k)$:

$$x = \prod_{i=1}^{\ell} p_i^{m_i} \geq \prod_{i=1}^{\ell} p_i = \prod_{\substack{p \in \mathbb{P} \\ \pi(p) \leq \ell}} p > \prod_{\substack{p \in \mathbb{P} \\ \pi(p) \leq \pi(k)}} p \geq \prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p.$$

Andererseits gilt mit Hilfe des vorherigen Satzes:

$$x \leq 2^k \leq \prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p.$$

Somit gilt

$$\prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p < \prod_{\substack{p \in \mathbb{P} \\ p \leq k}} p.$$

Dies liefert den gewünschten Widerspruch. ■

Theorem 2.30 *Sei $s, t \in \{0,1\}^*$ mit $|s| = m$, $|t| = n$, mit $nm \geq 29$. Sei $P \in \mathbb{N}$ und $p \in [1:P] \cap \mathbb{P}$ eine zufällig gewählte Primzahl, dann gilt für die Wahrscheinlichkeit eines irrtümlichen Treffers bei Karp-Rabin von s in t*

$$\text{Ws}(\textit{irrtümlicher Treffer}) \leq \frac{\pi(nm)}{\pi(P)}.$$

Damit folgt für die Wahl von P unmittelbar, dass $P > nm$ sein sollte.

Beweis: Sei $I = \{i \in [0 : n-m] \mid s \neq t_{i,m}\}$ die Menge der Positionen, an denen s in t nicht vorkommt. Dann gilt für alle $i \in I$, dass $V(s) \neq V(t_{i,m})$. Offensichtlich gilt

$$\prod_{i \in I} |V(s) - V(t_{i,m})| \;\leq\; \prod_{i \in I} 2^m \;\leq\; 2^{nm}.$$

Mit Satz 2.29 folgt dann, dass $\prod_{i \in I} |V(s) - V(t_{i,m})|$ maximal $\pi(nm)$ verschiedene Primteiler besitzt.

Sei s ein irrtümlicher Treffer an Position j, d.h. es gilt $V(s) \equiv V(t_{j,m}) \bmod p$ und $V(s) \neq V(t_j, m)$. Dann ist $j \in I$.

Da $V(s) \equiv V(t_{j,m}) \bmod p$, folgt, dass p ein Teiler von $|V(s) - V(t_{i,m})|$ ist. Dann muss p aber auch $\prod_{i \in I} |V(s) - V(t_{i,m})|$ teilen.

Für p gibt es $\pi(P)$ mögliche Kandidaten, aber es kann nur einer der $\pi(nm)$ Primfaktoren ausgewählt worden sein. Damit gilt

$$\text{Ws}(\text{irrtümlicher Treffer}) \leq \frac{\pi(nm)}{\pi(P)}.$$

Somit ist die Behauptung bewiesen. ■

Zum Abschluss zwei Lemmata, die verschiedene Wahlen von P begründen.

Lemma 2.31 *Wählt man $P = mn^2$, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers begrenzt durch $3/n$.*

Beweis: Nach dem vorherigen Satz gilt, dass die Wahrscheinlichkeit eines irrtümlicher Treffers durch $\frac{\pi(nm)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.27:

$$\frac{\pi(nm)}{\pi(n^2m)} \leq 1.26 \cdot \frac{nm \ln(n^2m)}{n^2m \ln(nm)} \leq \frac{1.26}{n} \cdot \frac{2\ln(nm)}{\ln(nm)} \leq \frac{2.52}{n}.$$

Somit ist die Behauptung bewiesen. ■

Ein Beispiel hierfür mit $n = 4000$ und $m = 250$. Dann ist $P = nm^2 < 2^{32}$. Somit erhält man mit einem Fingerabdruck, der sich mit 4 Byte darstellen lässt, eine Fehlerwahrscheinlichkeit von unter 0,1%.

Lemma 2.32 *Wählt man $P = nm^2$, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers begrenzt durch $3/m$.*

Beweis: Nach dem vorherigen Satz gilt, dass die Wahrscheinlichkeit eines irrtümlicher Treffers durch $\frac{\pi(nm)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.27:

$$\frac{\pi(nm)}{\pi(nm^2)} \leq 1.26 \cdot \frac{nm \ln(nm^2)}{nm^2 \ln(nm)} \leq \frac{1.26}{m} \cdot \frac{2 \ln(nm)}{\ln(nm)} \leq \frac{2.52}{m}.$$

Somit ist die Behauptung bewiesen. ■

Nun ist die Wahrscheinlichkeit eines irrtümlichen Treffers $O(\frac{1}{m})$. Da für jeden irrtümlichen Treffer ein Vergleich von s mit dem entsprechenden Teilwort von t ausgeführt werden muss (mit m Vergleichen), ist nun die erwartete Anzahl von Vergleichen bei einem irrtümlichen Treffer $O(m\frac{1}{m}) = O(1)$. Somit ist die erwartete Anzahl von Zeichenvergleichen bei dieser Variante wieder linear, d.h. $O(n)$.

Zum Schluss wollen wir noch anmerken, dass wir ja durchaus verschiedene Fingerabdrücke testen können. Nur wenn alle einen Treffer melden, kann es sich dann um einen Treffer handeln. Somit lässt sich die Fehlerwahrscheinlichkeit noch weiter senken bzw. bei gleicher Fehlerwahrscheinlichkeit kann man den kürzeren Fingerabdruck verwenden.

Suffix-Tries und -Trees 3

3.1 Suffix-Tries

In diesem Abschnitt wollen wir noch schnellere Verfahren zum Suchen in Texten vorstellen. Hier wollen wir den zu durchsuchenden Text vorverarbeiten (möglichst in Zeit und Platz $O(|t|)$), um dann sehr schnell ein Suchwort s (möglichst in Zeit $O(|s|)$) suchen zu können.

3.1.1 Definition von Suffix-Tries

Ein *Trie* ist eine Datenstruktur, in welcher eine Menge von Wörtern abgespeichert werden kann.

Definition 3.1 *Ein* Trie *für eine Menge $S \subseteq \Sigma^+$ ist ein gewurzelter Baum mit folgenden Eigenschaften:*

- *Jede Kante ist mit einem Zeichen aus Σ markiert;*
- *Die von einem Knoten ausgehenden Kanten besitzen paarweise verschiedene Markierungen;*
- *Jedes Wort $s \in S$ wird auf einen Knoten v abgebildet, so dass s entlang des Pfades von der Wurzel zu v steht;*
- *Jedem Blatt ist ein Wort aus S zugeordnet.*

Tries haben wir eigentlich schon kennen gelernt. Der Suchwort-Baum des Aho-Corasick-Algorithmus ohne die Failure-Links ist nichts anderes als ein Trie für die Menge der Suchwörter. In Abbildung 3.1 ist ein Beispiel für die dreibuchstabigen Abkürzungen der Monatsnamen angegeben.

Der Vorteil eines Tries ist, dass man in ihm nach einem Wort der Länge m in Zeit $O(m)$ suchen kann.

Definition 3.2 *Ein* Suffix-Trie *für ein Wort $t \in \Sigma^*$ ist ein Trie für alle Suffixe von $t = t_1 \cdots t_n$, d.h. $S = \{t_i \cdots t_n : i \in [1 : n+1]\}$, wobei $t_{n+1} \cdots t_n = \varepsilon$.*

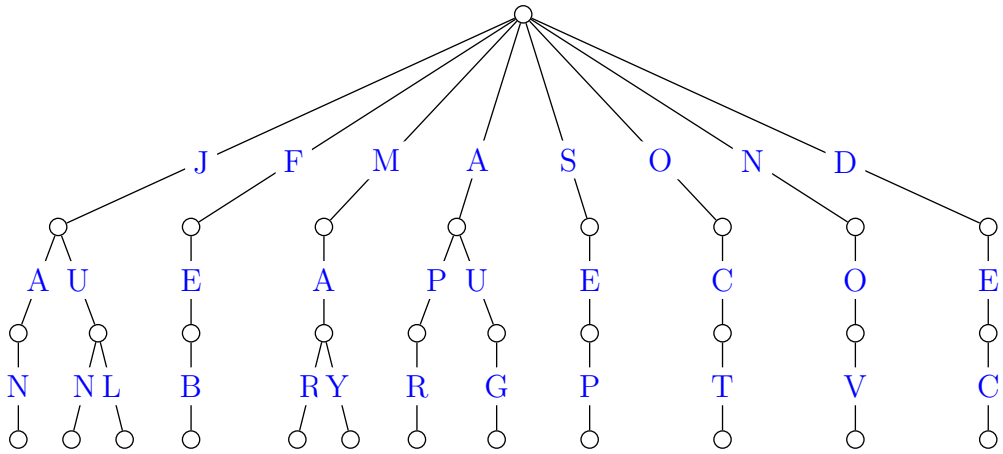


Abbildung 3.1: Beispiel: Trie für die dreibuchstabigen Monatsnamensabk.

In der Literatur werden Suffix-Tries oft auch als *atomare Suffix-Bäume* bezeichnet. Diese Bezeichnung sollte allerdings besser nicht mehr verwendet werden. Wir kommen später darauf zurück, warum.

In Abbildung 3.2 ist ein Suffix-Trie für das Wort $t = abbaba\$$ gezeigt. Damit kein Suffix von t ein Präfix eines anderen Suffixes ist und somit jedes Suffix von t in einem Blatt endet, hängt man an das Ende von t oft ein Sonderzeichen \$ ($\$ \notin \Sigma$) an.

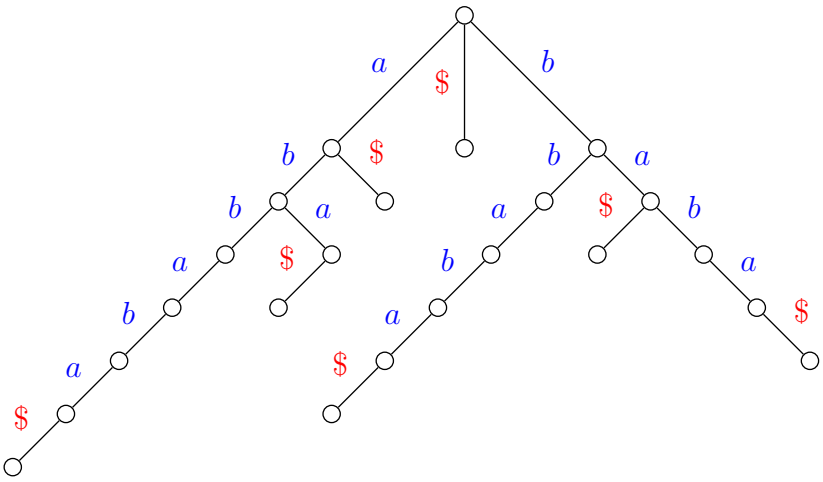


Abbildung 3.2: Beispiel: $t = abbaba\$$

Der Suffix-Trie kann also so aufgebaut werden, dass nach und nach die einzelnen Suffixe von t in den Trie eingefügt werden, beginnend mit dem längsten Suffix. Der einfache Algorithmus in Abbildung 3.3 beschreibt genau dieses Verfahren.

Im Folgenden betrachten wir für die Laufzeitanalyse als charakteristische Operationen die besuchten und modifizierten Knoten der zugrunde liegenden Bäume. Da

```
BuildSuffixTrie (char t[], int n)
begin
    tree T := empty();
    for (i := 1; i ≤ n; i++) do
        insert(T, t_i ··· t_n);
end
```

Abbildung 3.3: Algorithmus: Simple Methode zum Aufbau eines Suffix-Tries

das Einfügen des Suffixes $t_i \cdots t_n$ genau $O(|t_i \cdots t_n|) = O(n - i + 1)$ Operationen benötigt, folgt für die Laufzeit:

$$\sum_{i=1}^{n}(n - i + 1) \;=\; \sum_{i=1}^{n} i \;=\; O(n^2).$$

Halten wir noch schnell die folgende offensichtliche Beobachtung fest.

Beobachtung 3.3 *Sei $t \in \Sigma^*$, dann ist $w \in \Sigma^*$ genau dann ein Teilwort von t, wenn w ein Präfix eines Suffixes von t ist.*

Somit können wir in Suffix-Tries sehr schnell nach einem Teilwort w von t suchen. Wir laufen im Suffix-Trie die Buchstabenfolge $(w_1, \ldots, w_{|w|})$ ab. Sobald wir auf einen Knoten treffen, von dem wir nicht mehr weiter können, wissen wir, dass w dann nicht in t enthalten sein kann. Andernfalls ist w ein Teilwort von t.

18.06.19

3.1.2 Online-Algorithmus für Suffix-Tries

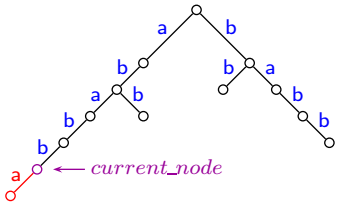
Sei $t \in \Sigma^n$ und sei T^i der Suffix-Trie für $t_1 \cdots t_i$. Ziel ist es nun, den Suffix-Trie T^i aus aus dem Trie T^{i-1} zu konstruieren. T^0 ist einfach zu konstruieren, da $t_1 \cdots t_0 = \varepsilon$ ist und somit T^0 der Baum ist, der aus nur einem Knoten (der Wurzel) besteht.

Auf der nächsten Seite ist in Abbildung 3.4 der sukzessive Aufbau eines Suffix-Tries für *ababba* aus dem Suffix-Trie für *ababb* ausführlich beschrieben. Dabei wird auch von so genannten Suffix-Links Gebrauch gemacht, die wir gleich noch formal einführen werden.

Das Bild in Abbildung 3.5 zeigt den kompletten Suffix-Trie T^7 für $t = ababbaa$ mit einigen Suffix-Links (auch die Suffix-Links zur Konstruktion des T^6 sind eingezeichnet).

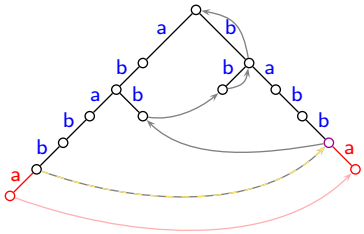
Beispiel: Aufbau eines Suffix-Tries

1. Schritt



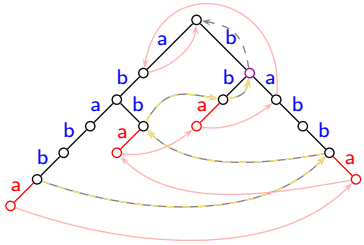
Der Trie für $t = ababb$ ist bereits konstruiert. An den Knoten, der am Ende des Pfades steht, dessen Labels das bisher längste Suffix ergibt, muss nun zuerst das neue Zeichen a angehängt werden. Anschließend muss an jedes kürzere Suffix ein a angehängt werden.

2. Schritt



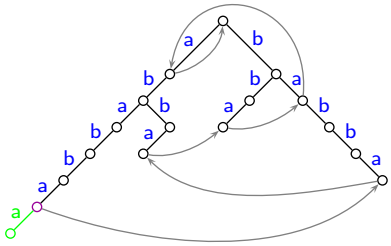
Um nun den Knoten schnell zu finden, der das nächst kürzere Suffix repräsentiert, sind im Trie so genannte *Suffix-Links* vorhanden. Diese zeigen immer auf den Knoten, der das nächst kürzere Suffix darstellt. Folge nun dem Suffix-Link des aktuellen Knotens und füge an dessen Endknoten wiederum die neue a-Kante ein und aktualisiere den Suffix-Link.

3. Schritt



Der zuvor beschriebene Vorgang wird nun so lange wiederholt, bis man auf einen Knoten trifft, der bereits eine a-Kante besitzt. Nun muss natürlich kein neuer Knoten mehr eingefügt werden, da dieser ja bereits existiert. Allerdings müssen noch die restlichen Suffix-Links aktualisiert werden.

4. Schritt



Nachdem T^6 vollständig aufgebaut ist, beginnt das ganze Spiel von vorne für den nächsten Buchstaben, im Beispiel a.

Abbildung 3.4: Beispiel: $t = ababbaa$; Konstruktion von T^6 aus T^5

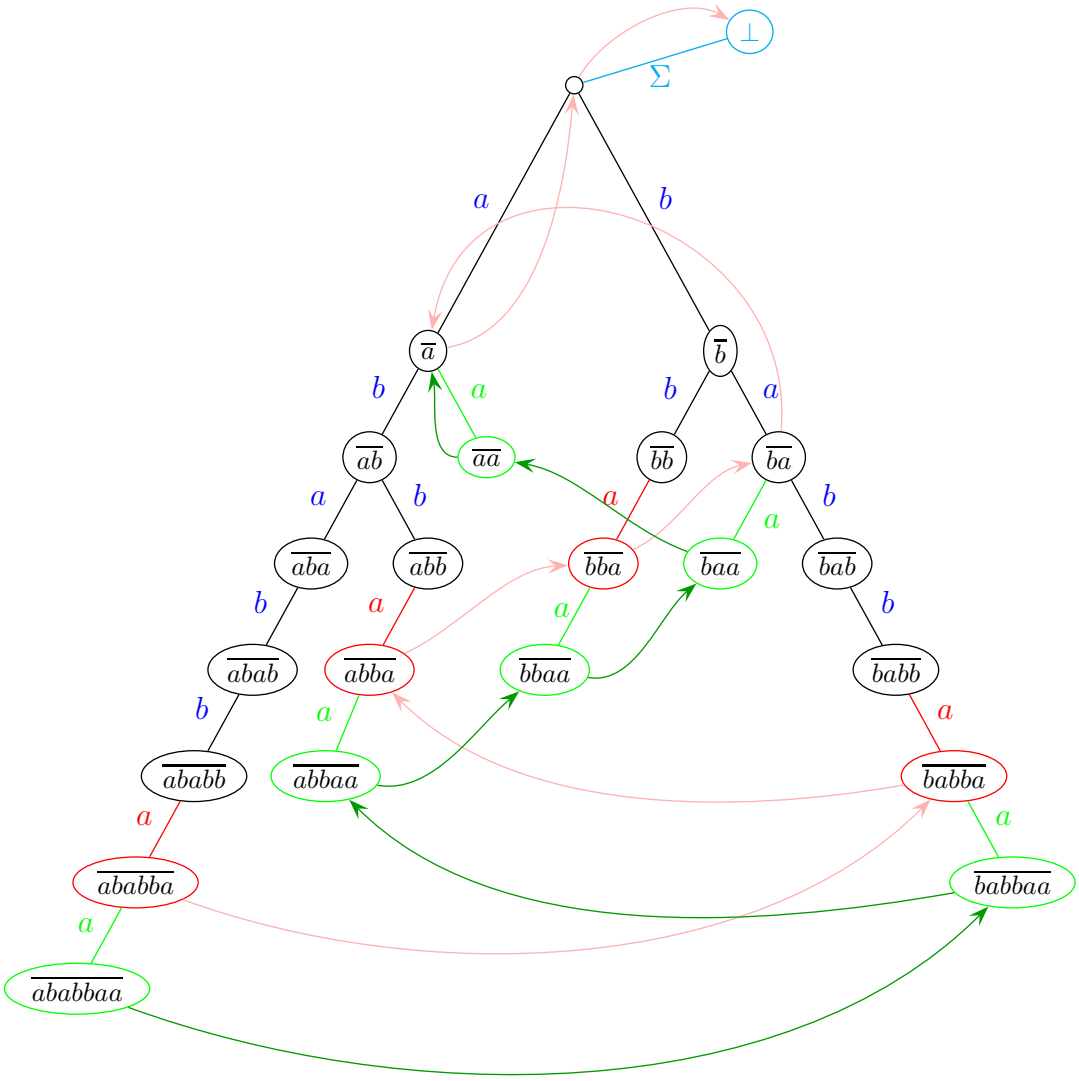


Abbildung 3.5: Beispiel: Suffix-Trie für $ababbaa$ mit Suffix-Links

Für das Folgenden führen wir eine einfache Notation zur Bezeichnung von Knoten eines Suffix-Tries ein.

Notation 3.4 *Sei $t \in \Sigma^*$ und sei w ein Teilwort von t. Den Knoten, der über w in einem Suffix-Trie für t erreichbar ist, wird mit $\overline{w}$ bezeichnet.*

Nun können wir den Begriff eines Suffix-Links definieren, der dem zu w gehörenden Knoten den zum längsten echten Suffix von w gehörenden Knoten zuordnet.

Definition 3.5 *Sei $t \in \Sigma^*$ und sei aw mit $a \in \Sigma$ und $w \in \Sigma^*$ ein Teilwort von t. Dann ist der Knoten $\overline{w}$ der* Suffix-Link *des Knotens $\overline{aw}$, kurz $suffix\_link(\overline{aw}) = \overline{w}$.*

Damit auch die Wurzel einen Suffix-Link besitzt, ergänzen wir den Suffix-Trie um eine *virtuelle Wurzel* $\perp$ und setzen $\mathit{suffix\_link}(root) = \perp$. Diese virtuelle Wurzel $\perp$ besitzt für jedes Zeichen $a \in \Sigma$ eine Kante zur eigentlichen Wurzel. Dies wird in Zukunft aus algorithmischer Sicht hilfreich sein, da für diesen Knoten dann keine Suffix-Links benötigt werden.

Die Idee der Konstruktion des Suffix-Tries T^i aus T^{i-1} ist als die Folgende. Wir laufen alle Knoten ab, die den Suffixen von $t_1 \cdots t_{i-1}$ im Suffix-Trie T^{i-1} entsprechen, und hängen an diese Knoten ein neues Kind über eine Kante mit Kantenlabel t_i an, sofern dieses noch nicht existiert.

Wie können wir alle Suffixe von $t_1 \cdots t_{i-1}$ effizient ablaufen? Wir beginnen hierzu am längsten Suffix, nämlich $t_1 \cdots t_{i-1}$ selbst, und erreichen das nächst kürzere Suffix jeweils über den zugehörigen Suffix-Link. Dies ist schematisch in Abbildung 3.6 dargestellt.

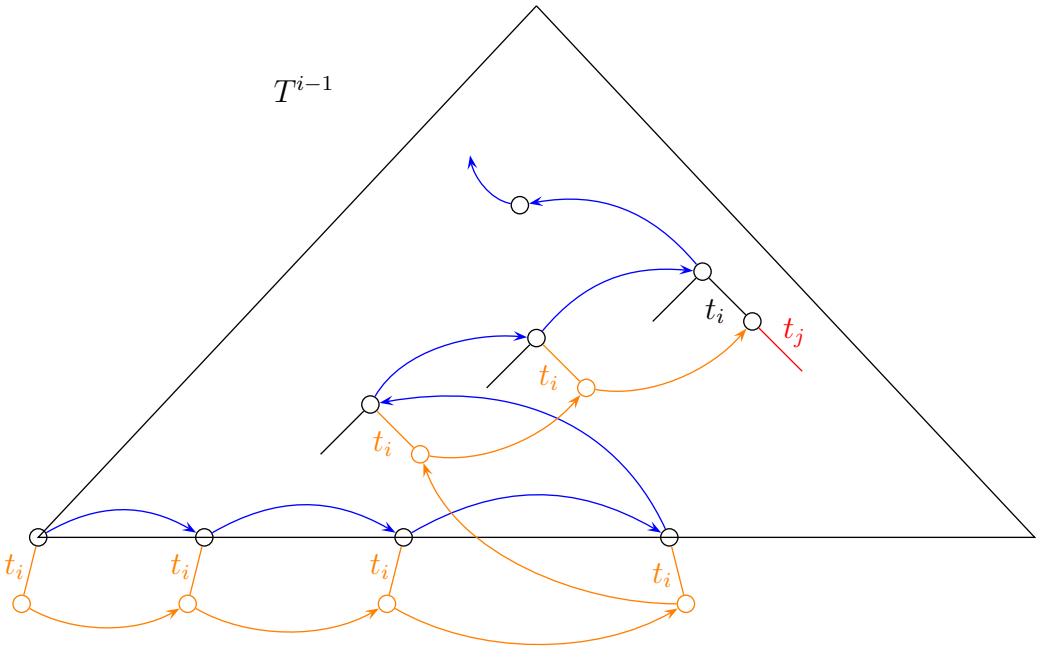


Abbildung 3.6: Skizze: Konstruktion eines Suffix-Tries

Aus diese Skizze lässt sich sofort der auf Suffix-Links basierende Algorithmus zur Konstruktion von Suffix-Tries in Abbildung 3.7 angegeben. Hierbei ist *curr_node* immer der aktuell betrachtete Knoten, an den ein neues Kind über eine Kante mit Kantenlabel t_i angehängt werden soll. *new_node* ist das jeweils neu angehängte Blatt und *prev_node* das zuletzt davor angehängte Blatt (außer zu Beginn jeder Phase, wo *prev_node* nicht wirklich sinnvoll definiert ist). *some_node* ist nur dann definiert, wenn die while-Schleife abbricht, und ist dann das Kind des aktuell betrachteten Knotens *curr_node*, von dem bereits eine Kante mit Kantenlabel t_i ausgeht.

BuildSuffixTrie (char $t[]$, int n)

```
begin
    T := ({root, ⊥}, {⊥ -x-> root : x ∈ Σ});
    suffix_link(root) := ⊥;
    longest_suffix := root;
    for (i := 1; i ≤ n; i++) do
        curr_node := longest_suffix;
        while (curr_node -t_i-> some_node does not exist) do
            Add curr_node -t_i-> new_node to T;
            if (curr_node = longest_suffix) then
                longest_suffix := new_node;
            else
                suffix_link(prev_node) := new_node;
            prev_node := new_node;
            curr_node := suffix_link(curr_node);
        suffix_link(prev_node) := some_node;
end
```

Abbildung 3.7: Algorithmus: Konstruktion von Suffix-Tries mittels Suffix-Links

3.1.3 Laufzeitanalyse für die Konstruktion von T^n

Bei der naiven Methode muss zur Verlängerung eines Suffixes das ganze Suffix durchlaufen werden. Daher sind zum Anhängen von t_i an $t_j \cdots t_{i-1}$ wiederum $O(i-j+1)$ Operationen erforderlich,

$$\sum_{i=1}^{n} \underbrace{\sum_{j=1}^{i} O(i-j+1)}_{\text{Zeit für } T^{i-1} \to T^i} = O\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) = O\left(\sum_{i=1}^{n} i^2\right) = O(n^3).$$

Dies ist also sogar schlechter als unser erster Algorithmus.

Durch die Suffix-Links benötigt das Verlängern des Suffixes $t_j \cdots t_{i-1}$ um t_i nur noch $O(1)$ Operationen.

$$\sum_{i=1}^{n}\sum_{j=1}^{i} O(1) = \sum_{i=1}^{n} O(i) = O(n^2)$$

Die Laufzeit ist zwar immer noch nicht besser geworden als mit unserem ersten Algorithmus, aber wir werden diese Idee später zur Konstruktion von Suffix-Bäumen recyceln können.

3.1.4 Wie groß kann ein Suffix-Trie werden?

Es stellt sich die Frage, ob wir unser Ziel aus der Einleitung zu diesem Abschnitt bereits erreicht haben. Betrachten wir das folgende Beispiel für $t = a^n b^n$ in Abbildung 3.8. Der Suffix-Trie für ein Wort t der Form $t = a^n b^n$ hat damit insgesamt

$$(2n + 1) + n^2 = (n + 1)^2 = \Theta(n^2)$$

viele Knoten, was nicht optimal ist. Zusammenfassend ist ein Suffix-Trie zwar eine Datenstruktur mit den gewünschten Eigenschaft, die allerdings zu speicherplatzintensiv ist.

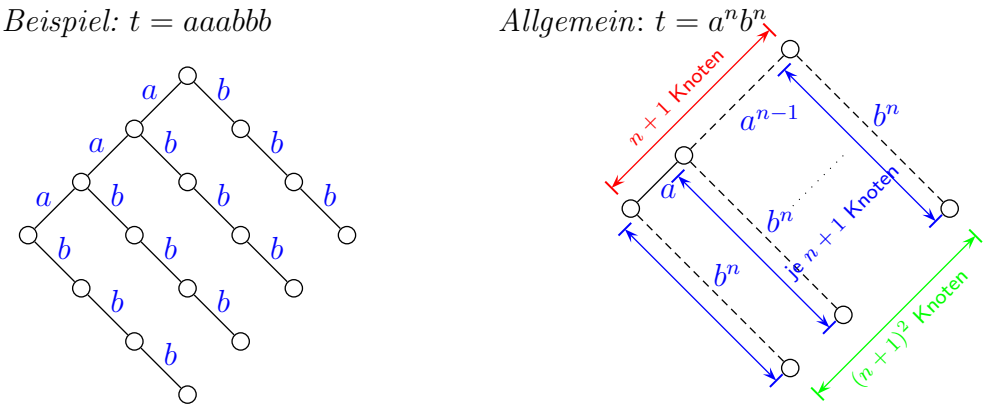


Abbildung 3.8: Beispiel: Potentielle Größe von Suffix-Tries (für $a^n b^n$)

3.2 Suffix-Bäume

Im Folgenden versuchen wir nun eine kompaktere Darstellung eines Suffix-Tries zu finden. Am vorherigen Beispiel können wir feststellen, dass sich in dem angegeben Suffix-Trie viele Pfade befinden. Die Idee hierfür ist also, möglichst viele Kanten sinnvoll zusammenzufassen, um den Trie zu kompaktifizieren. So können alle Pfade, die bis auf den Endknoten nur aus Knoten mit jeweils einem Kind bestehen, zu einer Kante zusammengefasst werden.

Diese so kompaktifizierten Tries werden *Patricia-Tries* genannt (für *Practical Algorithm To Retrieve Information Coded In Alphanumeric*), siehe auch unser Beispiel für die dreibuchstabigen Abkürzungen der Montansamen in Abbildung 3.9.

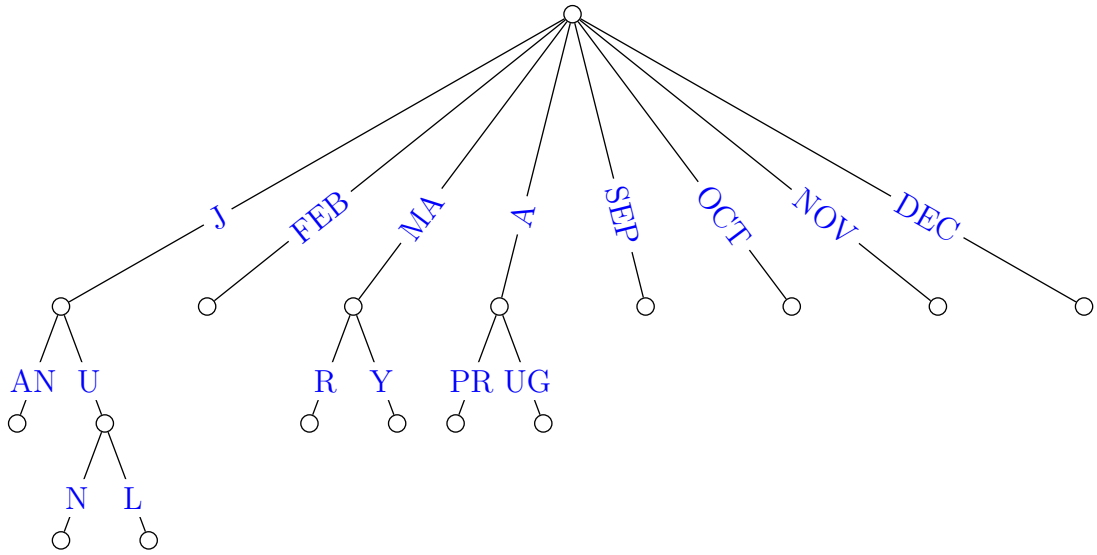


Abbildung 3.9: Beispiel: Patricia-Trie für die dreibuchstabigen Monatsnamensabk.

Definition 3.6 *Ein* Patricia-Trie *für eine Menge $S \subseteq \Sigma^+$ ist ein gewurzelter Baum mit folgenden Eigenschaften*

- *Jeder innere Knoten bis auf die Wurzel hat mindestens zwei Kinder;*
- *Jede Kante ist mit einem Zeichenkette aus Σ^+ markiert;*
- *Für jeden Knoten sind die ersten Zeichen der Markierungen der ausgehenden Kanten paarweise verschiedene;*
- *Für jedes Wort $s \in S$ gibt es einen Knoten v, so dass s ein Präfix des Wortes ist, das entlang des Pfades von der Wurzel zu v abgelesen wird;*
- *Jedem Blatt ist ein Wort aus S zugeordnet.*

3.2.1 Definition von Suffix-Bäumen

So kompaktifizierte Suffix-Tries werden *Suffix-Bäume* (engl. *suffix-trees*) genannt. Ein Beispiel ist in Abbildung 3.10 für $t = a^n b^n$ angegeben.

Definition 3.7 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Ein* Suffix-Baum *für t ist ein Patricia-Trie für alle Suffixe von t, also für $S = \{t_i \cdots t_n : i \in [1 : n+1]\}$.*

In der Literatur wird der hier definierte Suffix-Baum oft auch als *kompakter Suffix-Baum* benannt. Da mittlerweile der Begriff kompakter Suffix-Baum für verschiedene

weitere platzsparendere Varianten von Suffixbäumen verwendet wird, sollte man die Begriffe Suffix-Trie und Suffix-Baum anstelle von atomarer bzw. kompakter Suffix-Baum verwenden.

Das folgende Lemma ist aus der Informatik bzw. Graphentheorie wohlbekannt und wir überlassen den Beweis dem Leser zur Übung.

Lemma 3.8 *Ein gewurzelter Baum, in dem jeder innere Knoten mindestens zwei Kinder hat, besitzt mehr Blätter als innere Knoten.*

Da ein Suffix-Baum für $t \in \Sigma^n$ höchstens so viele Blätter wie Suffixe ungleich ε haben kann, also maximal n, besteht ein Suffix-Baum für t aus maximal $2n$ Knoten (nur die Wurzel darf nur ein Kind besitzen).

Durch das Zusammenfassen mehrerer Kanten wurde zwar die Anzahl der Knoten reduziert, dafür sind aber die Kantenlabels länger geworden. Deswegen werden als Labels der zusammengefassten Kanten nicht die entsprechenden Teilwörter verwendet, sondern die Position, an welcher das Teilwort im Gesamtwort auftritt. Für das Teilwort $t_i \cdots t_j$ wird die Referenz (i, j) verwendet (siehe auch Abbildung 3.10).

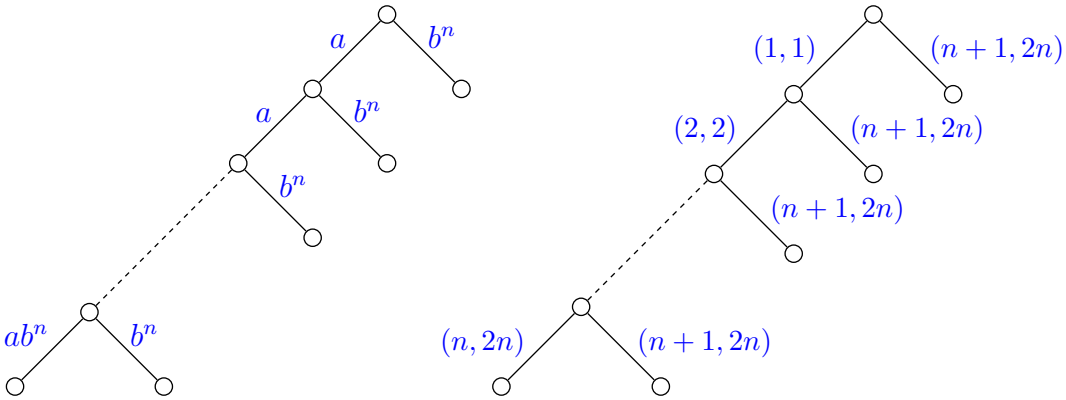


Abbildung 3.10: Beispiel: Kompaktifizierter Trie und echter Suffix-Baum für $t = a^n b^n$

Sei $t = t_1 \cdots t_n$ und $(\overline{t_1 \cdots t_j}, \overline{t_1 \cdots t_i})$ eine Kante des Baumes, dann verwenden wir als Label $(j+1, i)$ statt $t_{j+1} \cdots t_i$. Damit hat der Suffix-Baum nicht nur $O(|t|)$ viele Knoten, sondern er benötigt auch für die Verwaltung der Kantenlabels nur linear viel Platz (anstatt $O(n^2)$).

3.2.2 Ukkonens Online-Algorithmus für Suffix-Bäume

Im Folgenden bezeichnen wir mit $\hat{T}^i$ den Suffix-Baum für $t_1 \cdots t_i$. Wir wollen nun den Suffix-Baum $\hat{T}^i$ aus dem Suffix-Baum $\hat{T}^{i-1}$ aufbauen. Diese Konstruktion kann man aus der Konstruktion des Suffix-Tries T^i aus dem Suffix-Trie T^{i-1} erhalten.

Definition 3.9 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und $i \in [1:n]$. Weiter sei $s_j = t_j \cdots t_{i-1}$ für $j \in [1:i]$ und sei $\overline{s}_j$ der zugehörige Knoten im Suffix-Trie T^{i-1} für $t_1 \cdots t_{i-1}$. Dabei sei $\overline{s}_i = \overline{\varepsilon} = root$, und $\overline{s}_{i+1} = \perp$, wobei $\perp$ die virtuelle Wurzel ist, von der für jedes Zeichen $a \in \Sigma$ eine Kante zur eigentlichen Wurzel führt.*

Sei ℓ der größte Index, so dass für alle $j < \ell$ im Suffix-Trie T^{i-1} eine neue Kante für t_i an den Knoten $\overline{s}_j$ angehängt werden muss. Der Knoten $\overline{s}_\ell$ im Suffix-Trie T^{i-1} heißt Endknoten.

Sei k der größte Index, so dass für alle $j < k$ $\overline{s}_j$ ein Blatt ist. Der Knoten $\overline{s}_k$ heißt aktiver Knoten.

Da $\overline{s}_1$ immer ein Blatt ist, gilt $k > 1$, und da $\overline{s}_{i+1} = \perp$ immer eine Kante mit dem Label t_i besitzt, gilt $\ell \leq i+1$. Weiterhin sieht man auch leicht, dass nach Definition $k \leq \ell$ gilt. Für eine Illustration dieser Definition des aktiven und des Endknotens siehe auch Abbildung 3.11.

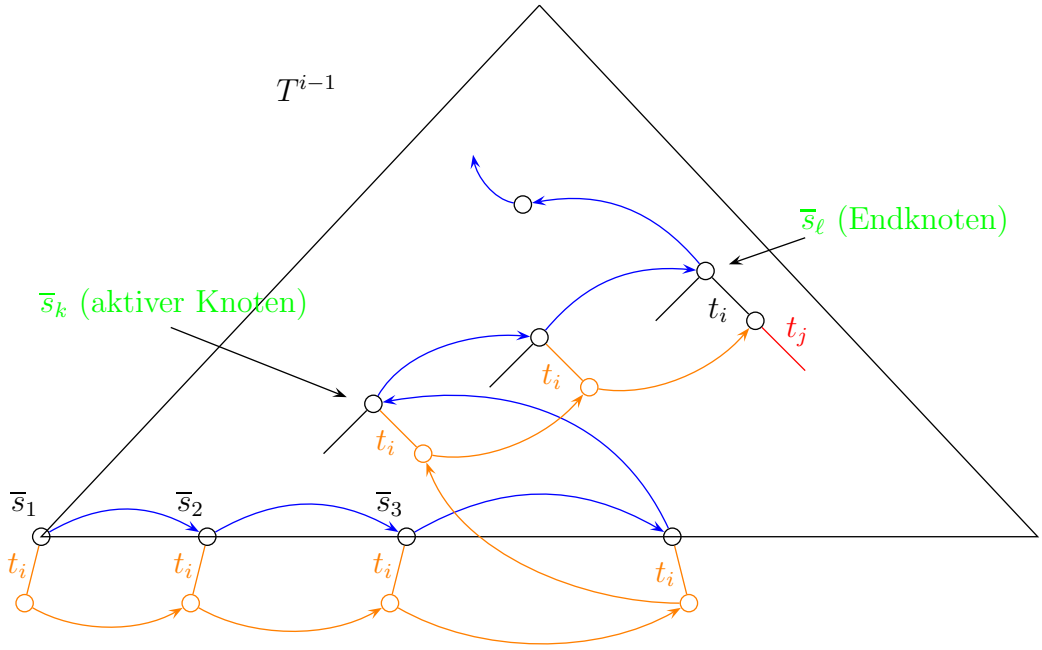


Abbildung 3.11: Skizze: Konstruktion eines Suffix-Tries

Definition 3.10 *Sei $t = t_1 \cdots t_n \in \Sigma^*$, sei $t' = t_1 \cdots t_i$ für ein $i \in [1:n]$ und sei w ein Teilwort von t'. Dann heißt ein Knoten $\overline{w}$ des Suffix-Tries T^i für t'* explizit, *wenn es auch im Suffix-Baum $\hat{T}^i$ für t' einen Knoten gibt, der über die Zeichenreihe w erreichbar ist. Andernfalls heißt er implizit.*

Es ist leicht einsehbar, dass die eigentliche Arbeit beim Aufbau des Suffix-Tries zwischen dem aktiven Knoten und dem Endknoten zu verrichten ist. Bei allen Knoten „vor“ dem aktiven Knoten, also bei allen bisherigen Blättern, muss einfach ein Kind mit dem Kantenlabel t_i eingefügt werden. Im Suffix-Baum bedeutet dies, dass an das betreffende Kantenlabel einfach das Zeichen t_i angehängt wird. Um sich nun diese Arbeit beim Aufbau des Suffix-Baumes zu sparen, werden wir gleich statt der Teilwörter als Kantenlabels so genannte (offene) Referenzen benutzt.

Definition 3.11 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $i \in [1:n]$. Weiter sei w ein Teilwort von $t_1 \cdots t_i$ mit $w = uv$, wobei $u = t_j \cdots t_{k-1}$ und $v = t_k \cdots t_p$. Ist $s = \overline{u}$ im Suffix-Baum $\hat{T}^i$ realisiert (also explizit in T^i) und kein Blatt, dann heißt $(s, (k, p))$ eine* Referenz *für $\overline{w}$ (im Suffix-Trie T^i).*

19.06.19

Im Folgenden werden wir dabei annehmen, dass $t_j \cdots t_p = w$ das erste Vorkommen von w in t ist, da es ja durchaus vorkommen kann, dass das Wort w mehrmals als Teilwort von t auftritt. Dies wird für den Algorithmus zur Konstruktion des Suffix-Baumes im Folgenden wichtig sein.

Beispiele für Referenzen in $t = ababbaa$ (siehe auch Abbildung 3.14) sind:

1. $w = \underset{2\,3\,4\,5}{babb} \mathrel{\hat{=}} (\overline{b}, (3,5)) \mathrel{\hat{=}} (\overline{ba}, (4,5)) \mathrel{\hat{=}} (\overline{\varepsilon}, (2,5))$.
2. $w = \underset{2\,3}{ba} \mathrel{\hat{=}} (\overline{\varepsilon}, (2,3)) \mathrel{\hat{=}} (\overline{b}, (3,3)) \mathrel{\hat{=}} (\overline{ba}, (4,3))$.
3. $w = \underset{4\,5\,6\,7}{bbaa} \mathrel{\hat{=}} (\overline{\varepsilon}, (4,7)) \mathrel{\hat{=}} (\overline{b}, (5,7)) \mathrel{\hat{=}} (\overline{b}, (5,\infty))$.

Beachte hierbei, dass im dritten Fall $(\overline{bbaa}, (8,7))$ keine Referenz ist, da $\overline{bbaa}$ ein Blatt ist, was in der Definition explizit ausgeschlossen ist.

Definition 3.12 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $i \in [1:n]$. Weiter sei w ein Teilwort von $t_1 \cdots t_i$ mit $w = uv$, wobei $u = t_j \cdots t_{k-1}$ und $v = t_k \cdots t_p$. Ist $s = \overline{u}$ im Suffix-Baum $\hat{T}^i$ realisiert und kein Blatt, dann heißt eine Referenz $(s, (k, p))$ für $\overline{w}$* kanonisch, *wenn $p - k$ minimal unter allen Referenzen für $\overline{w}$ ist.*

Im obigen Beispiel sind die Referenzen $(\overline{ba}, (4,5))$, $(\overline{ba}, (4,3))$ und $(\overline{b}, (5,7))$ kanonisch.

Für Blätter des Suffix-Baumes $\hat{T}^i$ werden alle Referenzen *offen* angegeben, d.h. es wird $(s, (k, \infty))$ statt $(s, (k, i))$ in $\hat{T}^i$ verwendet.

Des Weiteren werden die Kantenlabel zu Blättern im Suffix-Baum ebenfalls offen dargestellt, d.h. im Suffixbaum $\hat{T}^i$ für das Präfix $t_i \cdots t_i$ von t verwenden wir als als Label für Kanten zu Blättern (k, ∞) anstelle von (k, i).

Dadurch spart man sich dann die Arbeit, die Kantenlabels der Kanten, die zu Blättern führen, während der Konstruktion des Suffix-Baumes zu verlängern.

Definition 3.13 *Sei w ein Suffix von $t_1 \cdots t_n$ mit $w = uv$, wobei $u = t_j \cdots t_{k-1}$ und $v = t_k \cdots t_n$, so dass $(s, (k, n))$ eine kanonische Referenz von $\overline{w}$ ist und w als Teilwort nur einmal in $t_1 \cdots t_n$ auftritt. Dann heißt $(s, (k, \infty))$ die* offene Referenz *des Blattes $\overline{w}$.*

Es stellt sich nun noch die Frage, ob, wenn einmal ein innerer Knoten erreicht wurde, auch wirklich kein Blatt mehr um ein Kind erweitert wird, sondern nur noch innere Knoten. Des Weiteren ist noch von Interesse, ob, nachdem der Endknoten erreicht wurde, jeder weitere Knoten auch eine Kante mit dem Label t_i besitzt.

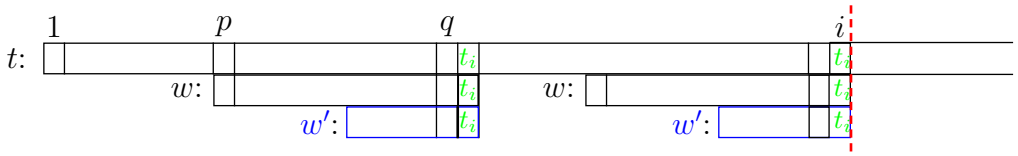


Abbildung 3.12: Skizze: Endknoten beendet Erweiterung von T^{i-1}

Dazu betrachten wir die Skizze in Abbildung 3.12. Im Folgenden sei $t' = t_1 \cdots t_{i-1}$. Zeigt ein Suffix-Link einmal auf einen inneren Knoten w, bedeutet dies, dass die Kantenlabels bis zu diesem Knoten ein Teilwort von t' ergeben, das nicht nur Suffix von t' ist, sondern auch sonst irgendwo in t' auftritt (siehe w in der Skizze). Da nun jeder weitere Suffix-Link auf einen Knoten verweist, der ein Suffix des zuvor gefundenen Teilwortes darstellt (blauer String w' in der Skizze), muss auch dieser Knoten ein innerer sein, da seine Kantenlabels ebenfalls nicht nur Suffixe von t' sind, sondern auch noch an einer anderen Stelle in t' auftauchen müssen.

Hat man nun den Endknoten erreicht, bedeutet dies, dass der betreffende Knoten bereits eine ausgehende Kante mit Kantenlabel t_i besitzt. Somit ergeben die Kantenlabels von der Wurzel bis zu diesem Knoten ein Teilwort von t', das nicht nur Suffix von t' ist, sondern auch sonst irgendwo mitten in t' auftritt. Außerdem kann aufgrund der Kante mit Kantenlabel t_i an das Teilwort das Zeichen t_i angehängt werden. Da nun jeder weitere Suffix-Link auf einen Knoten zeigt, der ein Suffix des vom Endknoten repräsentierten Wortes darstellt, muss auch an das neue Teilwort das Zeichen t_i angehängt werden können. Dies hat zur Folge, dass der betreffende Knoten eine Kante mit Kantenlabel t_i besitzen muss.

Lemma 3.14 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $\hat{T}^{i-1}$ bzw. $\hat{T}^i$ der Suffix-Baum für $t_1 \cdots t_{i-1}$ bzw. $t_1 \cdots t_i$ für ein $i \in [1:n]$. Ist $(s,(k,i-1))$ eine Referenz in $\hat{T}^{i-1}$ auf den Endknoten, dann ist $(s,(k,i))$ eine Referenz in $\hat{T}^i$ auf den aktiven Knoten.*

Beweis: Der aktive Knoten von T^i ist nach Definition der erste Knoten mit mindestens einem Kind auf dem Pfad bestehend aus Suffix-Links beginnend am Blatt $\overline{t_1 \cdots t_i}$. Wie man der Abbildung 3.11 sofort entnimmt, sind alle neuen Knoten, die bis zum Endknoten (ausschließlich) in T^{i-1} angehängt wurden, Blätter. Damit das Kind über die Kante t_i vom Endknoten in T^{i-1} der erste Kandidat für den aktiven Knoten von T^i.

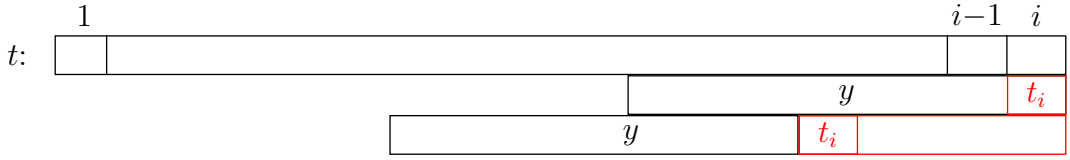


Abbildung 3.13: Skizze: Vom Endknoten von T^{i-1} zum aktiven Knoten von T^i

Wir müssen also nur noch nachweisen, dass dieser Knoten wirklich kein Blatt ist. Dazu betrachten wir die folgende Skizze in Abbildung 3.13. Bezeichnen wir mit $\overline{y}$ den Endknoten von T^{i-1}, der damit die Referenz $(s,(k,i-1))$ besitzt. Da $\overline{y}$ bereits in T^{i-1} die ausgehende Kante t_i besitzt, muss $y \cdot t_i$ ein Teilwort von $t_1 \cdots t_{i-1}$ sein. Damit muss $y \cdot t_i$ auch ein Teilwort von $t_1 \cdots t_i$ sein, das darüber hinaus mitten im Wort und nicht nur als Suffix auftauchen muss. Somit kann das Kind über die Kante t_i von $\overline{y}$ kein Blatt sein. ■

Jetzt zeigen wir noch, dass wir zumindest für innere Knoten eins Suffix-Baumes ebenfalls wieder Suffix-Links definieren können.

Lemma 3.15 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei aw ein Teilwort von t mit $a \in \Sigma$, so dass $\overline{aw}$ ein innerer Knoten im Suffix-Baum $\hat{T}$ für t ist. Dann ist auch $\overline{w}$ ein innerer Knoten in $\hat{T}$.*

Beweis: Wenn $\overline{aw}$ ein innerer Knoten in T ist, dann gibt es zwei ausgehende Kanten von $\overline{aw}$, deren Kantenlabel mit x bzw. y ($x \neq y \in \Sigma$) beginnt. Also ist sowohl awx als auch awy ein Teilwort von t. Somit sind auch wx und wy Teilwörter von t. Da im Suffix-Baum T alle Teilwörter von t dargestellt sind, muss $\overline{w}$ ein innerer Knoten sein, von dem mindestens zwei Kanten ausgehen, deren Kantenlabel mit x bzw. y beginnen. ■

Der obige Satz gilt nicht notwendigerweise für Blätter des Suffix-Baumes. Der Leser möge sich ein solches Beispiel überlegen. Daher sind Suffix-Links im Suffix-Baum in

der Regel nur für innere Knoten und nicht für Blätter definiert. In Abbildung 3.14 wurden auch Suffix-Links für die Blätter des Suffix-Baumes gepunktet in hellblau eingezeichnet, die hier zufälligerweise auch existieren.

Die Beispiele zu den folgenden Prozeduren beziehen sich alle auf den in Abbildung 3.14 angegeben Suffix-Baum für das Wort $t = ababbaa$.

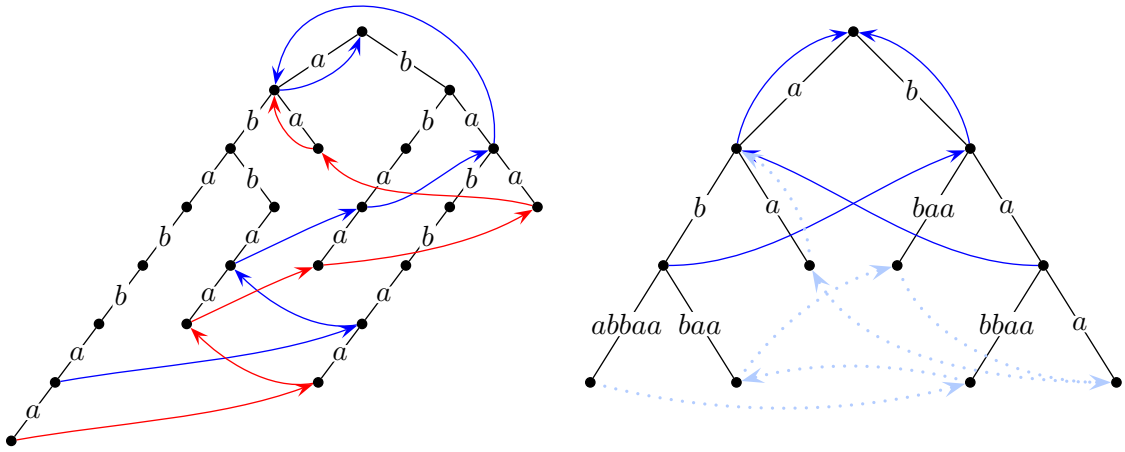


Abbildung 3.14: Beispiel: Suffix-Trie und Suffix-Baum für $t = ababbaa$

Ukkonens Algorithmus für Suffix-Bäume ist in der folgenden Abbildung 3.15 angegeben. In der for-Schleife des Algorithmus wird jeweils $\hat{T}^i$ aus $\hat{T}^{i-1}$ mit Hilfe der Prozedur Update konstruiert. Dabei ist jeweils $(s,(k,i-1))$ der aktive Knoten. Zu

BuildSuffixTree (char $t[]$, int n)

begin

$T := (\{root, \bot, v\}, \{root \xrightarrow{t_1} v\} \cup \{\bot \xrightarrow{x} root : x \in \Sigma\})$;
$suffix\_link(root) := \bot$;
$s := root$;
$k := 2$;
// $(s,(k,1))$ is a reference to $\overline{\varepsilon}$
for $(i := 2;\ i \leq n;\ i{+}{+})$ **do**
  // Constructing $\hat{T}^i$ from $\hat{T}^{i-1}$
  // $(s,(k,i-1))$ is active point in $\hat{T}^{i-1}$
  $(s,k) :=$ Update$(s,(k,i-1),i)$;
  // Now $(s,(k,i-1))$ is endpoint of $\hat{T}^{i-1}$

end

Abbildung 3.15: Algorithmus: Ukkonens Online-Algorithmus

Canonize (ref $(s,(k,p))$)

begin
  // $(s,(k,p))$ must be a reference
  while $(|t_k \cdots t_p| > 0)$ **do**
    let $e := s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;
    if $((|w| > |t_k \cdots t_p|)$ || (s' is a leaf)) **then break** ;
    $k := k + |w|$;
    $s := s'$;
  return (s, k);
  // $((s,k,p))$ is now the canonical reference
end

Abbildung 3.16: Algorithmus: Die Prozedur Canonize

Beginn für $i = 2$ ist dies für $(\overline{\varepsilon},(2,1)) \mathrel{\hat{=}} \overline{\varepsilon}$ klar. Wir bemerken hier, dass hier die Wurzel nur ein Kind besitzt (dies gilt aber für alle Suffix-Bäume zu Wörtern, die nur aus einem Buchstaben bestehen).

Zum Ende liefert die Prozedur Update, die aus $\hat{T}^{i-1}$ den Suffix-Baum $\hat{T}^i$ konstruiert, eine Referenz in $\hat{T}^{i-1}$ auf den Endknoten zurück, nämlich $(s,(k,i-1))$. Nach Lemma 3.14 ist dann die Referenz auf den aktiven Knoten von $\hat{T}^i$ gerade $(s,(k,i))$. Da in der for-Schleife i um eins erhöht wird, erfolgt der nächste Aufruf von Update wieder korrekt mit der Referenz auf den aktiven Knoten von $\hat{T}^i$, der jetzt ja $\hat{T}^{i-1}$ ist, wieder mit der korrekten Referenz $(s,(k,i-1))$.

Bevor wir die Prozedur Update erläutern, geben wir zunächst noch die Prozedur Canonize in Abbildung 3.16 an, die aus einer übergebenen Referenz eine kanonische Referenz konstruiert. Wir wollen ja eigentlich immer nur mit kanonischen Referenzen arbeiten. Der Prozedur Canonize wird eine Referenz $(s,(k,p))$ übergeben. Die Prozedur durchläuft dann den Suffix-Baum ab dem Knoten s entlang der Zeichen-

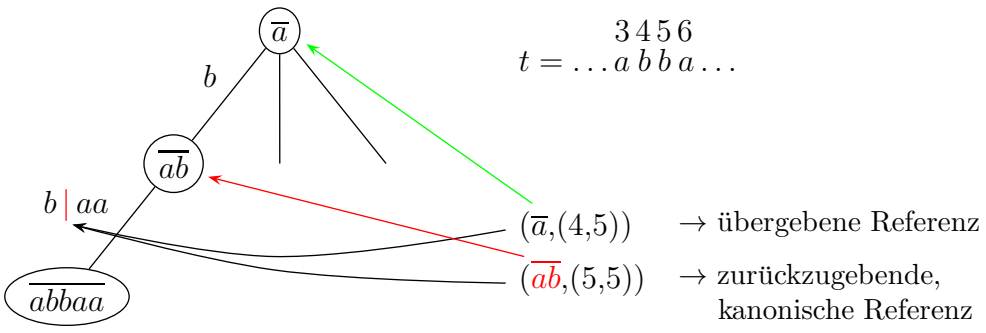


Abbildung 3.17: Beispiel: Erstellung kanonischer Referenzen mittels Canonize

Update (ref $(s,(k,p))$, int i)

begin
  // $(s,(k,p))$ `is active point`
  $old\_r := root$;
  $(s,k) :=$ Canonize$((s,(k,p)))$;
  $(done, r) :=$ TestAndSplit$((s,(k,p)), t_i)$;
  while (**!** $done$) **do**
    let m be a new leaf and add $r \xrightarrow{(i,\infty)} m$;
    if $(old\_r \neq root)$ **then**
      $suffix\_link(old\_r) = r$;
    $old\_r := r$;
    $(s,k) :=$ Canonize$((suffix_link(s),(k,p)))$;
    $(done, r) :=$ TestAndSplit$((s,(k,p)), t_i)$;
  if $(old\_r \neq root)$ **then**
    $suffix\_link(old\_r) := s$;
  return $(s,\ k)$;
end

Abbildung 3.18: Algorithmus: Die Prozedur Update

reihe $t_k \cdots t_p$. Kommt man nun mitten auf einer Kante zum stehen, entspricht der zuletzt besuchte Knoten dem Knoten, der für die kanonische Referenz verwendet wird. Dies ist in der folgenden Abbildung 3.17 noch einmal am Beispiel für die Referenz $(\overline{a},(4,5))$ illustriert.

Kommen wir noch einmal zur Implementierung vom Algorithmus in Abbildung 3.16 zurück. Die Bedingung $|t_k \cdots t_p| > 0$ wird in einer realen Implementierung natürlich durch die Abfrage $p - k + 1 > 0$ ersetzt. Ebenso wird $|w| > |t_k \cdots t_p|$ durch die Abfrage $p' - k' > p - k$ ersetzt, wobei (k', p') das implementierte Kantenlabel von e ist (also $w = t_{k'} \cdots t_{p'}$). Die Abfrage $w_1 = t_k$ ist dann gleichbedeutend mit $t_{k'} = t_k$. Hierbei ist zu beachten, dass für eine Referenz $(s,(k,p))$ gilt, dass aus $w_1 = t_k$ bereits $w_1 \cdots w_m = t_k \cdots t_{k+m-1}$ für $m = \min\{|w|, p - k + 1\}$ gelten muss (sonst wäre $(s,(k,p))$ keine Referenz gewesen).

In der Prozedur Update, die in Abbildung 3.18 angegeben ist, wird nun der Suffix-Baum $\hat{T}^i$ aus dem $\hat{T}^{i-1}$ aufgebaut. Die Prozedur Update bekommt eine, nicht zwingenderweise kanonische Referenz des Knotens übergeben, an welchen das neue Zeichen t_i, dessen Index i ebenfalls übergeben wird, angehängt werden muss. Dabei hilft die Prozedur Canonize, welche die übergebene Referenz kanonisch macht, und die Prozedur TestAndSplit, die den existierenden Knoten im Suffix-Baum zurückgibt, an welchen die neue Kante mit Kantenlabel t_i angehängt werden muss. Gegeben-

TestAndSplit (ref $(s,(k,p))$, char x)

begin
  // $(s,(k,p))$ must be a canonical reference
  if $(|t_k \cdots t_p| = 0)$ **then**
    // explicit reference
    if $(\exists\ s \xrightarrow{x\cdots})$ **then return** (TRUE, s);
    else return (FALSE, s);
  else
    // implicit reference
    let $e := s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;
    if $(x = w_{|t_k \cdots t_p|+1})$ **then return** (TRUE, s);
    else
      split $e := s \xrightarrow{w} s'$ s.t.: $s \xrightarrow{w_1 \cdots w_{|t_k \cdots t_p|}} r \xrightarrow{w_{|t_k \cdots t_p|+1} \cdots w_{|w|}} s'$;
      return (FALSE, r);
end

Abbildung 3.19: Algorithmus: Die Prozedur TestAndSplit

falls erzeugt die Prozedur TestAndSplit diesen Knoten neu, sofern dieser noch nicht vorhanden war.

Bevor wir jetzt die Prozedur Update weiter erläutern, gehen wir zunächst auf die Hilfsprozedur TestAndSplit ein. Die Prozedur TestAndSplit hat, wie oben bereits angesprochen, die Aufgabe, den Knoten auszugeben, an welchen die neue Kante mit Label t_i, falls noch nicht vorhanden, anzuhängen ist.

Die in Abbildung 3.19 angegebene Prozedur TestAndSplit geht dabei so vor, dass zunächst überprüft wird, ob die betreffende Kante im korrespondierenden Suffix-Trie bereits vorhanden ist oder nicht. Falls dies der Fall ist, liefert TestAndSplit TRUE und irgendeinen Knoten des Suffix-Baumes zurück. Ist die Kante jedoch noch nicht vorhanden, aber der Knoten, von welchem diese ausgehen sollte, bereits im Suffix-Baum existent, wird einfach dieser explizite Knoten zurückgegeben. Nun kann noch der Fall auftreten, dass der Knoten, an welchen die neue Kante angehängt werden muss, nur implizit im Suffix-Baum vorhanden ist. In diesem Fall muss die betreffende Kante aufgebrochen werden, und der benötigte Knoten als expliziter Knoten eingefügt werden. Anschließend wird dieser dann ausgegeben, und in der Prozedur Update wird die fehlende Kante eingefügt.

In der Prozedur TestAndSplit ist die erste Fallunterscheidung, ob die kanonische Referenz auf einen expliziten oder impliziten Knoten zeigt ($|t_k \cdots t_p| = 0$). Ist der Knoten explizit, muss in jedem Falle kein neuer innerer Knoten generiert werden und

die Kante mit Label t_i kann an den expliziten Knoten s im Suffix-Baum angehängt werden. Ist andernfalls der referenzierte Knoten implizit, so wird zuerst getestet, ob an diesem eine Kante mit Label t_i hängt ($t_i = w_{|t_k \cdots t_p|+1}$). Falls ja, ist nichts zu tun. Ansonsten muss diese Kante mit dem langen Label aufgebrochen werden, wie dies im folgenden Beispiel in Abbildung 3.20 illustriert ist. Dann wird der neue Knoten r, der in die lange Kante eingefügt wurde, von TestAndSplit zurückgegeben, damit die Prozedur Update daran die neue Kante mit Label t_i anhängen kann.

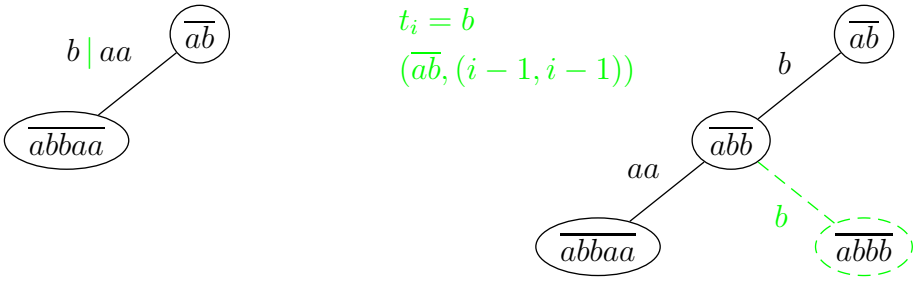


Abbildung 3.20: Beispiel: Vorgehensweise von TestAndSplit

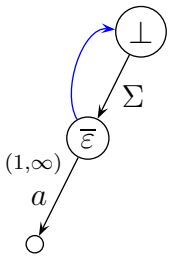
Auch hier wird wieder in einer realen Implementierung beispielsweise die Abfrage $|t_k \cdots t_p| = 0$ durch $k > p$ ersetzt. Analog entspricht die Abfrage $x = w_{|t_k \cdots t_p|+1}$ dem Wert $x = t_{p'+1}$, wenn wieder (k', p') das implementierte Kantenlabel von e ist. Die Kantenlabel der aufgesplitteten Kanten sind dann (k, p) und $(p+1, p')$.

Kehren wir nun zur Beschreibung der Prozedur Update zurück. Auch hier folgen wir wieder vom aktiven Knoten aus den Suffix-Links und hängen eine neue Kante mit Label t_i ein. Wir bemerken hier, dass wir im Suffix-Baum nur für interne Knoten die Suffix-Links berechnen und nicht für die Blätter. Mit *old_r* merken wir uns immer den zuletzt neu eingefügten internen Knoten, für den der Suffix-Link noch zu konstruieren ist. Zu Beginn setzen wir *old_r* auf *root*, um damit anzuzeigen, dass wir noch keinen neuen Knoten eingefügt haben.

Wann immer wir dem Suffix-Link gefolgt sind und im Schritt vorher einen neuen Knoten eingefügt haben, setzen wir für diesen zuvor eingefügten Knoten den Suffix-Link jetzt mittels Suffix-Link(*old_r*) $= r$. Der Knoten r ist gerade der Knoten, an den wir jetzt aktuell die Kante mit Label t_i anhängen wollen und somit der Elter des neuen Blattes. Somit wird der Suffix-Link von den korrespondierenden Eltern korrekt gesetzt.

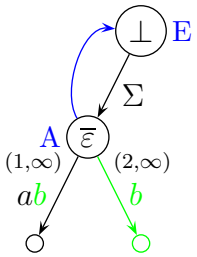
25.06.19

Wir wollen jetzt noch ein längeres *Beispiel* durchgehen, das in den Abbildungen 3.21 mit 3.27 abgebildet ist, und zeigt wie nach und nach der Suffix-Baum für das bereits öfter verwendete Wort $t = ababbaa$ aufgebaut wird. Im Folgenden wird mit A bzw. E die Position des aktiven Knoten bzw. des Endknoten des Suffix-Tries beschrieben.



$(\overline{\varepsilon}, (2,1))$; $i = 1$; $t_i = a$

Abbildung 3.21: Beispiel: Ukkonens Algorithmus: $\hat{T}^1$



$(\overline{\varepsilon}, (2,1))$; $i = 2$; $t_i = b$

↓ Canonize

$(\overline{\varepsilon}, (2,1))$

↯ TestAndSplit = (FALSE,$\overline{\varepsilon}$)

$(\overline{\varepsilon}, (2,1))$

↓ Suffix-Link

$(\bot, (2,1))$

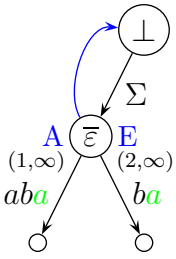
↓ Canonize

$(\bot, (2,1))$

↯ TestAndSplit = (TRUE,$\bot$)

$(\bot, (2,1))$

Abbildung 3.22: Beispiel: Ukkonens Algorithmus: $\hat{T}^2$



$(\bot, (2,2))$; $i = 3$; $t_i = a$

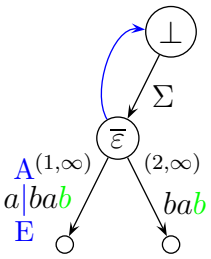
↓ Canonize

$(\overline{\varepsilon}, (3,2))$

↯ TestAndSplit = (TRUE,$\overline{\varepsilon}$)

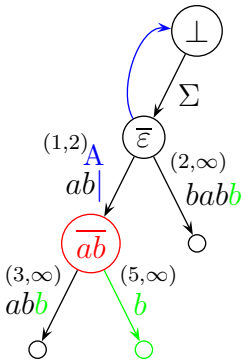
$(\overline{\varepsilon}, (3,2))$

Abbildung 3.23: Beispiel: Ukkonens Algorithmus: $\hat{T}^3$



$(\overline{\varepsilon}, (3,3))$; $i = 4$; $t_i = b$
↓ Canonize
$(\overline{\varepsilon}, (3,3))$
↯ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
$(\overline{\varepsilon}, (3,3))$

Abbildung 3.24: Beispiel: Ukkonens Algorithmus: $\hat{T}^4$



$(\overline{\varepsilon}, (3,4))$; $i = 5$; $t_i = b$
↓ Canonize
$(\overline{\varepsilon}, (3,4))$
↯ TestAndSplit = (FALSE,$\overline{ab}$)
$(\overline{\varepsilon}, (3,4))$
↓ Suffix-Link
$(\bot, (3,4))$
↓ Canonize
$(\overline{\varepsilon}, (4,4))$
↯ TestAndSplit = (FALSE,$\overline{b}$)
$(\overline{\varepsilon}, (4,4))$
↓ Suffix-Link
$(\bot, (4,4))$
↓ Canonize
$(\overline{\varepsilon}, (5,4))$
↯ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
$(\overline{\varepsilon}, (5,4))$

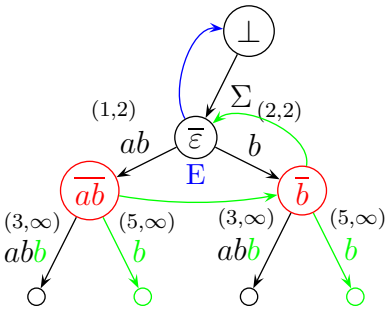
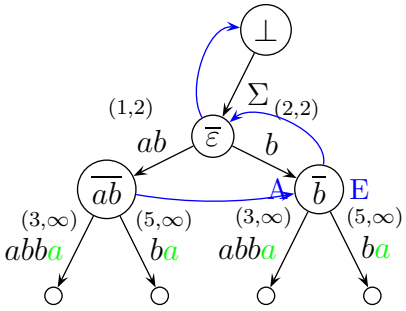
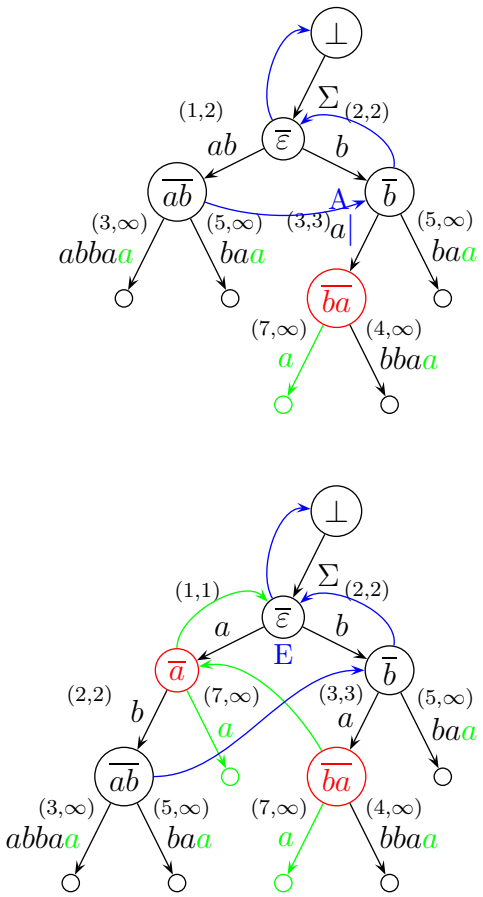


Abbildung 3.25: Beispiel: Ukkonens Algorithmus: $\hat{T}^5$



$(\overline{\varepsilon}, (5,5))$; $i = 6$; $t_i = a$
↓ Canonize
$(\overline{b}, (6,5))$
↯ TestAndSplit = (TRUE,$\overline{b}$)
$(\overline{b}, (6,5))$

Abbildung 3.26: Beispiel: Ukkonens Algorithmus: $\hat{T}^6$



$(\overline{b}, (6,6))$; $i = 7$; $t_i = a$

↓ Canonize

$(\overline{b}, (6,6))$

↯ TestAndSplit = (FALSE,$\overline{ba}$)

$(\overline{b}, (6,6))$

↓ Suffix-Link

$(\overline{\varepsilon}, (6,6))$

↓ Canonize

$(\overline{\varepsilon}, (6,6))$

↯ TestAndSplit = (FALSE,$\overline{a}$)

$(\overline{\varepsilon}, (6,6))$

↓ Suffix-Link

$(\bot, (6,6))$

↓ Canonize

$(\overline{\varepsilon}, (7,6))$

↯ TestAndSplit = (TRUE,$\overline{\varepsilon}$)

$(\overline{\varepsilon}, (7,6))$

Abbildung 3.27: Beispiel: Ukkonens Algorithmus: $\hat{T}^7$

3.2.3 Laufzeitanalyse

Zum Schluss kommen wir noch zur Laufzeitanalyse des Online-Algorithmus von Ukkonen für die Konstruktion von Suffix-Bäumen. Wie messen hier als Laufzeit wieder die Anzahl besuchter (und neu erzeugter) Knoten (inklusive Blätter). Alte Knoten werden insbesondere in der Prozedur Canonize besucht, neue innere Knoten werden in TestAndSplit erzeugt (an die dann in der Prozedur Update ein neues Blatt gehängt wird). Es genügt also, die betrachteten Knoten in den Prozeduren Canonize und TestAndSplit zu zählen.

In der Prozedur Canonize wird zunächst der aufgerufene Knoten besucht. Da nach jedem Aufruf von Canonize sofort ein Aufruf von TestAndSplit folgt, ist diese Zahl gleich der Anzahl Aufrufe von TestAndSplit, die wir später ermitteln. Mit jedem in der Prozedur Canonize neu aufgesuchten Knoten in der while-Schleife wird der Parameter k erhöht. Eine genaue Inspektion des Pseudo-Codes ergibt, dass k nie erniedrigt wird, zu Beginn den Wert 2 besitzt und nach oben durch n beschränkt

ist. Also können in der while-Schleife innerhalb verschiedener Aufrufe von Canonize maximal n Knoten des Suffix-Baumes besucht werden.

Bleibt noch die Analyse von TestAndSplit. Wir unterscheiden hier zwischen erfolgreichen Aufrufen (Rückgabe-Wert ist TRUE) und erfolglosen Aufrufen (Rückgabewert ist FALSE). Da nach jedem erfolgreichen Aufruf die Prozedur Update verlassen wird, kann es maximal n erfolgreiche Aufrufe von TestAndSplit geben. In jedem erfolglosen Aufruf von TestAndSplit wird der Suffix-Baum um mindestens einen Knoten (genau genommen um ein Blatt und eventuell ein innerer Knoten) erweitert. Wie wir gesehen haben hat der Suffix-Baum für ein Wort der Länge n maximal $O(n)$ Knoten. Also kann es maximal $O(n)$ erfolglose Aufrufe von TestAndSplit geben.

Damit kann es auch insgesamt nur maximal $O(n)$ Durchläufe der while-Schleife in allen Aufrufen von Canonize geben. Insgesamt werden also maximal $O(n)$ Knoten im Suffix-Baum besucht (bzw. neu erzeugt).

Theorem 3.16 *Ein Suffix-Baum für $t \in \Sigma^n$ lässt sich in Zeit $O(n)$ und Platz $O(n)$ mit Hilfe des Algorithmus von Ukkonen konstruieren.*

Für die hier angegebene Laufzeitanalyse wurde allerdings implizit angenommen, dass man für einen Knoten und ein Zeichen die ausgehende Kante von diesem Knoten, dessen Label mit dem angegeben Zeichen beginnt, in konstanter Zeit ermitteln kann. Dies ist nicht nicht zwangsläufig gegeben, wie wir im nächsten Abschnitt sehen werden. Für eine halbwegs sinnvolle Implementierung ist dies aber in Zeit $O(|\Sigma|)$ möglich, was hier im Allgemeinen als (wenn auch manchmal nicht kleine) Konstante angesehen wird.

26.06.19

Theorem 3.17 *Ein Suffix-Baum für $t \in \Sigma^n$ lässt sich in Zeit $O(|\Sigma| \cdot n)$ und Platz $O(n)$ mit Hilfe des Algorithmus von Ukkonen für ein Alphabet Σ beliebiger Größe konstruieren.*

3.3 Verwaltung der Kinder eines Knotens

Zum Schluss wollen wir uns noch kurz mit der Problematik des Abspeicherns der Verweise auf die Kinder eines Knotens in einem Suffix-Baum widmen. Ein Knoten kann entweder sehr wenige Kinder besitzen oder sehr viele, nämlich bis zu $|\Sigma|$ viele. Wir schauen uns verschiedene Methoden hierfür einmal an und bewerten sie nach Platz- und Zeitbedarf.

Wir bewerten den Zeitbedarf danach, wie lange es dauert, bis wir für ein Zeichen $a \in \Sigma$ das Kind gefunden haben, das über die Kante erreichbar ist, dessen Kantelabel

mit a beginnt. Für den Platzbedarf berechnen wir den Bedarf für den gesamten Suffix-Baum für einen Text der Länge n.

Felder: Die Kinder eines Knotens lassen sich sehr einfach mit Hilfe eines Feldes der Größe $|\Sigma|$ darstellen.

- Platz: $O(n \cdot |\Sigma|)$.
  Dies folgt daraus, dass für jeden Knoten ein Feld mit Platzbedarf $O(|\Sigma|)$ benötigt wird.
- Zeit: $O(1)$.
  Übliche Realisierungen von Feldern erlauben einen Zugriff in konstanter Zeit.

Der Zugriff ist also sehr schnell, wo hingegen der Platzbedarf, insbesondere bei großen Alphabeten doch sehr groß werden kann.

Lineare Listen: Wir verwalten die Kinder eines Knotens in einer linearen Liste, diese kann entweder sortiert sein (wenn es eine Ordnung, auch eine künstliche, auf dem Alphabet gibt) oder auch nicht.

- Platz: $O(n)$.
  Für jeden Knoten ist der Platzbedarf proportional zur Anzahl seiner Kinder. Damit ist Platzbedarf insgesamt proportional zur Anzahl der Knoten des Suffix-Baumes, da jeder Knoten (mit Ausnahme der Wurzel) das Kind eines Knotens ist. Im Suffix-Baum gilt, dass jeder Knoten (mit Ausnahme der Wurzel) entweder kein oder mindestens zwei Kinder hat. Für solche Bäume ist bekannt, dass die Anzahl der inneren Knoten kleiner ist als die Anzahl der Blätter. Da ein Suffix-Baum für einen Text der Länge n maximal n Blätter besitzt, folgt daraus die Behauptung für den Platzbedarf.
- Zeit: $O(|\Sigma|)$.
  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber auch größenordnungsmäßig im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

Balancierte Bäume: Die Kinder lassen sich auch mit Hilfe von balancierten Suchbäumen (AVL-, Rot-Schwarz-, B-Bäume, etc.) verwalten:

- Platz: $O(n)$
  Da der Platzbedarf für einen Knoten ebenso wie bei linearen Listen proportional zur Anzahl der Kinder ist, folgt die Behauptung für den Platzbedarf unmittelbar.

- Zeit: $O(\log(|\Sigma|))$.

  Da die Tiefe von balancierten Suchbäumen logarithmisch in der Anzahl der abzuspeichernden Schlüssel ist, folgt die Behauptung unmittelbar.

Hashfunktionen: Eine weitere Möglichkeit ist die Verwaltung der Kinder aller Knoten in einem einzigen großen Feld der Größe $O(n)$. Um nun für ein Knoten auf ein spezielles Kind zuzugreifen wird dann eine Hashfunktion verwendet:

$$h : V \times \Sigma \to \mathbb{N} : (v, a) \mapsto h(v, a)$$

Zu jedem Knoten und dem Symbol, die das Kind identifizieren, wird ein Index des globales Feldes bestimmt, an der die gewünschte Information enthalten ist.

Leider bilden Hashfunktionen ein relativ großes Universum von potentiellen Referenzen (hier Paare von Knoten und Symbolen aus Σ also $\Theta(n \cdot |\Sigma|)$) auf ein kleines Intervall ab (hier Indizes aus $[1 : \ell]$ mit $\ell = \Theta(n)$). Daher sind so genannte *Kollisionen* prinzipiell nicht auszuschließen. Ein Beispiel ist das so genannte *Geburtstagsparadoxon*. Ordnet man jeder Person in einem Raum eine Zahl aus dem Intervall $[1 : 366]$ zu (nämlich ihren Geburtstag), dann ist ab 23 Personen die Wahrscheinlichkeit größer als 50%, dass zwei Personen denselben Wert erhalten. Also muss man beim Hashing mit diesen Kollisionen leben und diese geeignet auflösen. Für die Details wird auf andere Vorlesungen (wie etwa zu Algorithmen und Datenstrukturen) verwiesen.

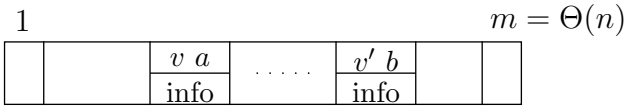


Abbildung 3.28: Skizze: Realisierung mittels eines Feldes und Hashing

Um solche Kollisionen überhaupt festzustellen, enthält jeder Feldeintrag i neben den normalen Informationen noch die Informationen, wessen Kind er ist und über welches Symbol er von seinem Elter erreichbar ist (siehe auch Abbildung 3.28. Somit lassen sich Kollisionen leicht feststellen und die üblichen Operationen zur Kollisionsauflösung anwenden.

- Platz: $O(n)$

  Das folgt unmittelbar aus der obigen Diskussion.

- Zeit: $O(1)$

  Im Wesentlichen erfolgt der Zugriff in konstanter Zeit, wenn man voraussetzt, dass sich die Hashfunktion einfach (d.h. in konstanter Zeit) berechnen lässt und dass sich Kollisionen effizient auflösen lassen.

Hybride Verfahren: Schaut man sich einen Suffix-Baum genauer an, so wird man feststellen, dass die Knoten auf niedrigem Level, d.h. nah bei der Wurzel, einen sehr großen Verzweigungsgrad haben. Dort sind also fast alle potentiellen Kinder auch wirklich vorhanden. Knoten auf recht großem Level haben hingegen relativ wenige Kinder. Aus diesem Grunde bietet sich auch eine hybride Implementierung an. Für Knoten auf niedrigem Level verwendet man für die Verwaltung der Kinder einfache Felder, während man bei Knoten auf großem Level auf eine der anderen Varianten umsteigt.

3.4 Anwendungen und Ausblick

Wir haben bislang nur Konstruktionsalgorithmen für Suffix-Bäume kennen gelernt, aber noch keine Anwendungen. Wir wollen mögliche Anwendungen hier nur skizzieren, die genaue Ausarbeitung hiervon wird in den Übungen behandelt.

Zuerst einmal wiederholen wir die Tatsache, dass ein Suffix-Baum ein Patricia-Trie für alle Suffixe eines gegebenen Wortes t ist. Somit gibt es für alle Teilwörter von t (die ja Präfixe eines Suffixes von t sind) einen Pfad im Suffixbaum startend an der Wurzel und endend irgendwo im Suffix-Baum (also auch mitten auf einer Kante).

Mit dieser Überlegung können wir einen sehr effizienten Suchalgorithmus beschreiben. Der zu durchsuchende Text wird in einen Suffix-Baum vorverarbeitet. Anschließend versuchen wir für ein Suchwort den zugehörigen Pfad im Suffix-Baum zu finden. Wenn uns das gelingt, ist die Suche erfolgreich, ansonsten erfolglos. Vorteil dieses Verfahrens ist, dass die Suchzeit mit $O(m)$ unabhängig von der Größe des zu durchsuchenden Textes ist.

Eine andere Anwendung ist die Analyse von Texten beispielsweise bzgl. Wiederholungen. Für ein Word w, zu dem es einen Pfad im Suffix-Baum für $t\$$ gibt, gilt als erstes, dass w ein Teilwort von t ist. Weiterhin gibt die Anzahl der Blätter, die von dieser Position noch erreichbar sind, die Anzahl der Vorkommen von w in t an. Somit können wir Repeats in t sehr einfach charakterisieren, klassifizieren und auch effizient erkennen.

Leider sind Suffix-Bäume trotz ihres linearen Platzbedarfs in der Praxis meist viel zu speicherintensiv und werden heutzutage durch Suffix-Arrays verdrängt.

Definition 3.18 *Ein* Suffix-Array S *für ein Wort* $t \in \Sigma^n$ *ist Feld der Größe* n *mit folgenden Eigenschaften:*

i) $\forall i < j \in [1:n]: \; t_{S[i]} \cdots t_n < t_{S[j]} \cdots t_n$,

ii) $\{S[i] \,:\, i \in [1:n]\} = [1:n]$.

Umgangssprachlich enthält das Suffix-Array für ein Wort t die lexikographisch sortierten Suffixe von t repräsentiert durch ihre Anfangspositionen in t. In Abbildung 3.29 ist ein Beispiel für das Suffix-Array für unser bekanntes Beispielwort $t = ababbaa$ angegeben.

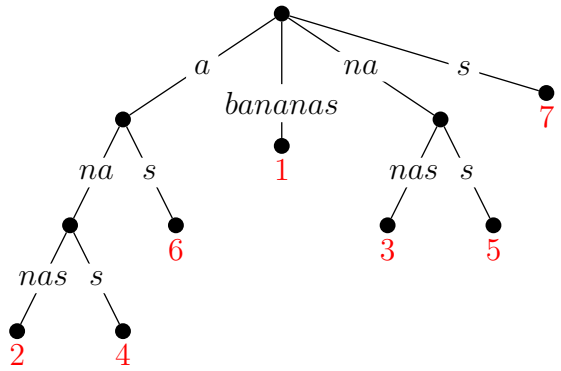
$$\begin{array}{lcll} \mathrm{S}[1] & = & 7 & \hat{=}\ a \\ \mathrm{S}[2] & = & 6 & \hat{=}\ aa \\ \mathrm{S}[3] & = & 1 & \hat{=}\ ababbaa \\ \mathrm{S}[4] & = & 3 & \hat{=}\ abbaa \\ \mathrm{S}[5] & = & 5 & \hat{=}\ baa \\ \mathrm{S}[6] & = & 2 & \hat{=}\ babbaa \\ \mathrm{S}[7] & = & 4 & \hat{=}\ bbaa \end{array}$$

Abbildung 3.29: Beispiel: Suffix-Array für $t = ababbaa$

Um alle Algorithmen für Suffix-Bäume mittels Suffix-Arrays implementieren zu können, werden noch wenige, zusätzliche Datenstrukturen benötigt, die jedoch sehr platzeffizient zu implementieren sind.

Rekapitulieren wir grob den Platzbedarf für eine einfache Implementierung eines Suffix-Baumes. Für ein Wort mit n Buchstaben hat der zugehörige Suffix-Baum n Blätter und somit etwa n innere Knoten. Um diese referenzieren zu können, werden jeweils etwa $\log(n)$ Bits benötigt, so dass eine einfache Implementierung des Baumes $2n\log(n)$ Bits benötigt. Dieser Baum hat dann auch etwa $2n$ Kanten, für die jeweils noch zwei Indizes für das Kantenlabel zu speichern sind, also noch einmal etwa $4n\log(n)$ Bits. Dazu ist auch das Wort selbst mit $n\log(|\Sigma|)$ Bits zu speichern. Es werden also insgesamt etwa $6n\log(n) + n\log(|\Sigma|)$ Bits benötigt.

Für das Suffix-Array bzw. das Wort selbst werden $n\log(n)$ bzw. $n\log(|\Sigma|)$ Bits benötigt. Die zusätzlich benötigten Datenstrukturen können mit $4n + o(n)$ Bits implementiert werden. Dies ist insgesamt deutlich weniger als für einen Suffix-Baum. Akutelle Varianten können den Text und das Suffix-Array selbst noch mit deutlich weniger Platz speichern. Der Platzbedarf ist dann abhängig von der Entropie des gegebenen Textes und somit noch einmal deutlich platzsparender. In Abbildung 3.30 ist noch einmal die Gegenüberstellung des Suffix-Baums und des Suffix-Arrays fürdas Wort *bananas* illustriert. Dabei ist in blau noch die so genannte *LCP-Tabelle* (LCP für longest common prefix) angegeben, die für ein Suffix im Suffix-Array und dessen unmittelbaren Vorgänger im Suffix-Array die Länge des längsten gemeinsame Präfixes angibt. Mit Hilfe dieser und dem Suffix-Array kann die Gestalt des Suffix-Baumes im Wesentliche rekonstruiert werden. Für die Details verweisen wir auf Spezialvorlesungen, wie etwa *Algorithmen auf Sequenzen*.



1	2	3	4	5	6	7
2	4	6	1	3	5	7
a	a	a	b	n	n	s
n	n	s	a	a	a	
a	a		n	n	s	
n	s		a	a		
a			n	s		
s			a			
			s			
–	3	1	0	0	2	0

Abbildung 3.30: Beispiel: Suffix-Baum und Suffix-Array für *bananas* inklusive LCP-Tabelle

Paarweises Sequenzen-Alignment 4

4.1 Distanz- und Ähnlichkeitsmaße

In diesem Kapitel beschäftigen wir uns mit dem so genannten *paarweisen Sequenzen-Alignment*. Dabei werden zwei Sequenzen bzw. Zeichenreihen miteinander verglichen und darauf untersucht, wie *ähnlich* sie sind und wie die eine Sequenz aus der anderen hervorgegangen sein kann.

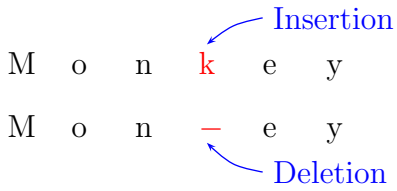


Abbildung 4.1: Beispiel: Ähnlichkeit mittels Insertion bzw. Deletion

Dies wird an folgenden Beispielen in Abbildung 4.1 illustriert. Wir betrachten die Worte MONKEY und MONEY. Wie in der Abbildung 4.1 bereits zeigt, kann das Wort MONEY aus dem Wort MONKEY dadurch hervorgegangen sein, dass im Wort MONKEY einfach das „k“ gelöscht wurde. Andersherum kann in das Wort MONEY ein „k“ eingefügt worden sein, so dass das Wort MONKEY entstand.

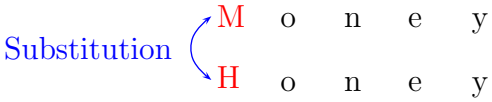


Abbildung 4.2: Beispiel: Ähnlichkeit durch Substitution

Ähnlich verhält es sich mit den Wörtern MONEY und HONEY, wie in Abbildung 4.2 gezeigt. Hier wurde entweder ein „M“ durch ein „H“ oder ein „H“ durch ein „M“ ersetzt.

Um nun tiefer in das Thema einsteigen zu können, sind zuerst einige grundlegende Definitionen nötig. Wir werden zuerst so genannte Distanz- und Ähnlichkeitsmaße einführen, die es uns überhaupt erst erlauben, ähnliche Sequenzen qualitativ und quantitativ zu beurteilen.

4.1.1 Edit-Distanz

Zuerst einmal führen wir die so genannte Edit-Distanz ein.

Notation 4.1 *Sei Σ ein Alphabet und sei $- \notin \Sigma$. Dann bezeichne $\overline{\Sigma} := \Sigma \cup \{-\}$ das um $-$ erweiterte Alphabet von Σ und $\overline{\Sigma}_0^2 := \overline{\Sigma} \times \overline{\Sigma} \setminus \{(-,-)\}$.*

Wir werden das neue Zeichen $-$ oft auch als *Leerzeichen* bezeichnen. Damit sind wir in der Lage, so genannte Edit-Operation zu definieren.

Definition 4.2 *Eine* Edit-Operation *ist ein Paar $(x,y) \in \overline{\Sigma}_0^2$ und (x,y) heißt*

- Match*, wenn $x = y \in \Sigma$;*
- Substitution*, wenn $x \neq y$ mit $x, y \in \Sigma$;*
- Insertion*, wenn $x = -$, $y \in \Sigma$;*
- Deletion*, wenn $x \in \Sigma$, $y = -$.*

Als Indel-Operation *bezeichnet man eine Edit-Operation, die entweder eine Insertion oder eine Deletion ist.*

Wir bemerken hier, dass $(x,x) \in \Sigma \times \Sigma$ explizit als Edit-Operation zugelassen wird. In vielen Büchern wird dies nicht erlaubt. Wie man leicht sieht, ist ein solcher Match als Edit-Operation eigentlich eine neutrale oder NoOp-Operation und kann daher meist problemlos weggelassen werden.

Definition 4.3 *Ist (x,y) eine Edit-Operation und sind $a, b \in \Sigma^*$, dann gilt $a \xrightarrow{(x,y)} b$, d.h. a kann mit Hilfe der Edit-Operation (x,y) in b umgeformt werden, wenn*

- $x, y \in \Sigma \ \wedge \ \exists\, i \in [1 : |a|] : \ (a_i = x) \wedge (b = a_1 \cdots a_{i-1} \cdot y \cdot a_{i+1} \cdots a_{|a|})$ *(Substitution oder Match);*
- $x \in \Sigma \ \wedge \ y = - \ \wedge \ \exists\, i \in [1 : |a|] : (a_i = x) \wedge (b = a_1 \cdots a_{i-1} \cdot a_{i+1} \cdots a_{|a|})$ *(Deletion);*
- $x = - \ \wedge \ y \in \Sigma \ \wedge \ \exists\, i \in [1 : |a| + 1] : (b = a_1 \cdots a_{i-1} \cdot y \cdot a_i \cdots a_{|a|})$ *(Insertion).*

Sei $s = ((x_1, y_1), \ldots, (x_m, y_m))$ eine (möglicherweise auch leere) Folge von Edit-Operationen mit $a_{i-1} \xrightarrow{(x_i,y_i)} a_i$, wobei $a_i \in \Sigma^*$ für $i \in [0 : m]$ und $a := a_0$ und $b := a_m$, dann schreibt man auch kurz $a \overset{s}{\Rightarrow} b$.

In den folgenden Abbildungen 4.3 und 4.4 sind zwei Beispiele von Transformationen einer Sequenz in eine andere mit Hilfe von Edit-Operationen dargestellt. Dabei wird immer die obere Sequenz in die untere umgewandelt.

$$AGTGTAGTA \overset{s}{\Rightarrow} ACGTGTTT \text{ mit } s = ((G,-),(T,C),(A,G),(G,T),(A,T))$$
$$\text{oder mit } s = ((G,T),(A,G),(G,-),(A,T),(T,C))$$

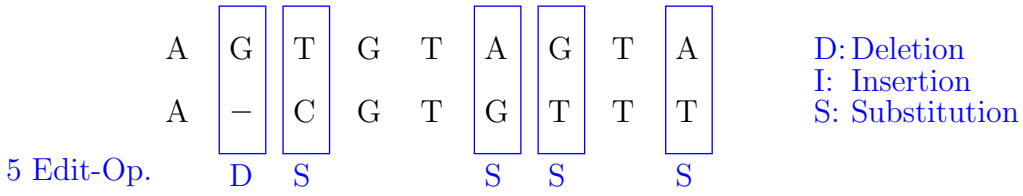


Abbildung 4.3: Beispiel: Transformation mit Edit-Operationen

Wie man im obigen Beispiel sieht, kommt es nicht unbedingt auf die Reihenfolge der Edit-Operationen an. Wir wollen hier nur anmerken, dass es durchaus Sequenzen von Edit-Operationen gibt, so dass es auf die Reihenfolge ankommt. Der Leser möge sich solche Beispiele überlegen.

Wir können dieselben Sequenzen auch mit Hilfe anderer Edit-Operationen ineinander transformieren.

$$AGTGTAGTA \overset{s'}{\Rightarrow} ACGTGTTT \text{ mit } s' = ((-,C),(A,-),(G,-),(A,T))$$

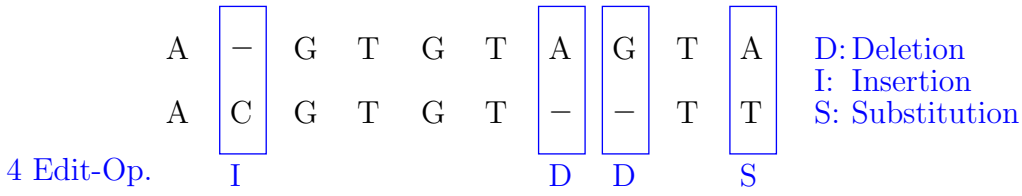


Abbildung 4.4: Beispiel: Transformation mit anderen Edit-Operationen

Um nun die Kosten einer solchen Folge von Edit-Operationen zu bewerten, müssen wir zunächst die Kosten einer einzelnen Edit-Operation bewerten. Dazu verwenden wir eine so genannte *Kostenfunktion* $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$. Beispielsweise kann man alle Kosten bis auf die Matches auf 1 setzen. Für Matches sind Kosten größer als Null in der Regel nicht sonderlich sinnvoll. In der Biologie, wenn die Sequenzen Basen oder insbesondere Aminosäuren repräsentieren, wird man jedoch intelligentere Kostenfunktionen wählen. Ja nach mathematischem Kontext wird die Kostenfunktion mal auf $\overline{\Sigma}^2$ und mal auf $\overline{\Sigma}_0^2$ definiert.

Definition 4.4 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion. Seien $a, b \in \Sigma^*$ und sei $s = (s_1, \ldots, s_\ell)$ eine Folge von Edit-Operationen mit $a \overset{s}{\Rightarrow} b$. Dann sind die* Kosten der Edit-Operationen *s definiert als*

$$w(s) := \sum_{j=1}^{\ell} w(s_j).$$

Die Edit-Distanz *von $a, b \in \Sigma^*$ ist definiert als*

$$d_w(a, b) := \min \left\{ w(s) \ : \ a \overset{s}{\Rightarrow} b \right\}.$$

Setzt man alle Kosten in der Kostenfunktion bis auf die Matches auf 1, erhält man die so genannte *Levenshtein-Distanz*. Zuerst einmal überlegen wir uns, dass folgende Beziehung sinnvollerweise gelten soll:

$$\forall x, y, z \in \overline{\Sigma} : \ w(x, y) + w(y, z) \geq w(x, z).$$

Wir betrachten zum Beispiel eine Mutation (x, z), die als direkte Mutation relativ selten (also teuer) ist, sich jedoch sehr leicht (d.h. billig) durch zwei Mutationen (x, y) und (y, z) ersetzen lässt. Dann sollten die Kosten für diese Mutation durch die beiden billigen beschrieben werden, da man in der Regel nicht feststellen kann, ob eine beobachtete Mutation direkt oder über einen Umweg erzielt worden ist. Diese Bedingung ist beispielsweise erfüllt, wenn w eine Metrik ist.

Definition 4.5 *Sei M eine beliebige Menge. Eine Funktion $w : M \times M \to \mathbb{R}_+$ heißt* Metrik *auf M, wenn die folgenden Bedingungen erfüllt sind:*

(M1) $\forall x, y \in M : w(x, y) = 0 \Leftrightarrow x = y$ *(Definitheit);*

(M2) $\forall x, y \in M : w(x, y) = w(y, x)$ *(Symmetrie);*

(M3) $\forall x, y, z \in M : w(x, z) \leq w(x, y) + w(y, z)$ *(Dreiecksungleichung).*

Dann heißt (M, w) auch metrischer Raum.

Oft nimmt man daher für eine Kostenfunktion für ein Distanzmaß an, dass es sich um eine Metrik handelt. Wir müssen uns nur noch M1 und M2 im biologischen Zusammenhang klar machen. M1 sollte gelten, da nur ein Zeichen mit sich selbst identisch ist und somit nur Zeichen mit sich selbst einen Abstand von 0 haben sollten. M2 ist nicht ganz so klar, da Mutationen in die eine Richtung durchaus wahrscheinlicher sein können als in die umgekehrte Richtung. Da wir allerdings immer nur die Sequenzen sehen und oft nicht die Kenntnis haben, welche Sequenz aus welcher

Sequenz durch Mutation entstanden ist, ist die Annahme der Symmetrie bei Kostenfunktionen sinnvoll. Des Weiteren kommt es bei Vergleichen von Sequenzen oft vor, dass beide von einer dritten Elter-Sequenz abstammen und somit aus dieser durch Mutationen entstanden sind. Damit ist die Richtung von Mutationen in den beobachteten Sequenzen völlig unklar und die Annahme der Symmetrie der einzig gangbare Ausweg.

Lemma 4.6 *Ist $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine Metrik, dann ist auch $d_w : \Sigma^* \times \Sigma^* \to \mathbb{R}_+$ eine Metrik.*

Beweis: Offensichtlich gilt $d_w(a,a) = 0$ für $a \in \Sigma^*$, da $a \overset{\varepsilon}{\Rightarrow} a$ und $w(\varepsilon) = 0$. Für $a, b \in \Sigma^*$ gilt andererseits:

$$\begin{aligned} 0 &= d_w(a,b) \\ &= \min\left\{w(s) \,:\, a \overset{s}{\Rightarrow} b\right\} \\ &= \min\left\{\sum_{i=1}^{r} w(s_i) \,:\, a \overset{s}{\Rightarrow} b \text{ mit } s = (s_1,\ldots,s_r)\right\}. \end{aligned}$$

Da w immer nichtnegativ ist, muss $w(s_i) = 0$ für alle $i \in [1:r]$ sein (bzw. $r = 0$). Somit sind alle ausgeführten Edit-Operationen Matches, d.h. $s_i = (x_i, x_i)$ für ein $x_i \in \Sigma$. Damit gilt $a = b$ und somit M1 für d_w.

Seien $a, b \in \Sigma^*$. Dann gilt:

$$\begin{aligned} d_w(a,b) &= \min\left\{w(s) \,:\, a \overset{s}{\Rightarrow} b \text{ mit } s = (s_1,\ldots,s_r)\right\} \\ &= \min\left\{\sum_{i=1}^{r} w(s_i) \,:\, a = a_0 \overset{s_1}{\to} a_1 \cdots a_{r-1} \overset{s_r}{\to} a_r = b\right\} \end{aligned}$$

Es bezeichne $\tilde{s}_i$ für eine Edit-Operation $s_i = (x,y)$ mit $x, y \in \overline{\Sigma}$ die inverse Edit-Operation $\tilde{s}_i = (y,x)$. Indem wir die inversen Edit-Operationen nur in der anderen Richtung anwenden, muss es auch eine Transformation $b \Rightarrow a$ geben.

$$\begin{aligned} &= \min\left\{\sum_{i=1}^{r} w(s_i) \,:\, b = a_r \overset{\tilde{s}_r}{\to} a_{r-1} \cdots a_1 \overset{\tilde{s}_1}{\to} a_0 = a\right\} \\ &\quad \text{da nach M2 } w \text{ symmetrisch ist (d.h. } w(\tilde{s}_i) = w(s_i)) \\ &= \min\left\{\sum_{i=1}^{r} w(\tilde{s}_i) \,:\, b = a_r \overset{\tilde{s}_r}{\to} a_{r-1} \cdots a_1 \overset{\tilde{s}_1}{\to} a_0 = a\right\} \end{aligned}$$

$$\begin{aligned} &= \min\left\{w(\tilde{s}^R) \,:\, b \overset{\tilde{s}^R}{\Rightarrow} a \text{ mit } \tilde{s}^R = (\tilde{s}_r, \ldots, \tilde{s}_1)\right\} \\ &= d_w(b, a). \end{aligned}$$

Damit ist auch M2 für d_w nachgewiesen.

Seien $a, b, c \in \Sigma^*$. Seien s und t zwei Folgen von Edit-Operationen, so dass $a \overset{s}{\Rightarrow} b$ und $w(s) = d_w(a, b)$ sowie $b \overset{t}{\Rightarrow} c$ und $w(t) = d_w(b, c)$. Dann ist offensichtlich die Konkatenation $s \cdot t$ eine Folge von Edit-Operationen mit $a \overset{s \cdot t}{\Longrightarrow} c$. Dann gilt weiter

$$d_w(a, c) \leq w(s \cdot t) = w(s) + w(t) = d_s(a, b) + d_w(b, c).$$

Damit ist auch die Dreiecksungleichung (M3) für d_w bewiesen. ■

4.1.2 Alignment-Distanz

In diesem Abschnitt wollen wir einen weiteren Begriff einer Distanz einzuführen, die auf Ausrichtungen der beiden bezeichneten Zeichenreihen beruht. Dazu müssen wir erst einmal formalisieren, was wir unter einer Ausrichtung (einem Alignment) von zwei Zeichenreihen verstehen wollen.

Definition 4.7 *Sei $u \in \overline{\Sigma}^*$. Dann ist die Restriktion von u auf Σ mit Hilfe eines Homomorphismus h definiert als $u|_\Sigma = h(u)$, wobei*

$$\begin{aligned} h(a) &= a && \text{für alle } a \in \Sigma, \\ h(-) &= \varepsilon, \\ h(u'u'') &= h(u')h(u'') && \text{für alle } u', u'' \in \Sigma^*. \end{aligned}$$

Die Restriktion von $u \in \overline{\Sigma}^*$ auf Σ ist also nichts anderes als das Löschen aller Leerzeichen $-$ aus u. Nun sind wir formal in der Lage, ein paarweises Alignment zu definieren.

Definition 4.8 *Ein* paarweises Alignment *ist ein Paar $(\overline{a}, \overline{b}) \in \overline{\Sigma}^* \times \overline{\Sigma}^*$ mit $|\overline{a}| = |\overline{b}|$ und $(\overline{a}_i, \overline{b}_i) \neq (-, -)$ für alle $i \in [1 : |\overline{a}|]$.*

$(\overline{a}, \overline{b})$ ist ein Alignment für $a, b \in \Sigma^*$, wenn $\overline{a}|_\Sigma = a$ und $\overline{b}|_\Sigma = b$ gilt. Weiterhin ist $\mathcal{A}(a, b) = \left\{(\overline{a}, \overline{b}) \in \overline{\Sigma}^* \times \overline{\Sigma}^* \,:\, (\overline{a}, \overline{b}) \text{ ist ein Alignment für } a, b\right\}$.

$$\begin{array}{cccccccc} A & - & G & G & C & G & T & T \\ A & G & C & G & C & - & T & T \end{array}$$

Abbildung 4.5: Beispiel: Alignment von $AGGCGTT$ mit $AGCGCTT$

Betrachten wir das folgende Beispiel in Abbildung 4.5, ein Alignment zwischen $a = AGGCGTT$ und $b = AGCGCTT$. Mit Hilfe solcher Alignments lassen sich jetzt ebenfalls Distanzen zwischen Zeichenreihen definieren.

Definition 4.9 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion. Die Kosten eines Alignments $(\overline{a}, \overline{b})$ für (a, b) sind definiert als*

$$w(\overline{a}, \overline{b}) := \sum_{i=1}^{|\overline{a}|} w(\overline{a}_i, \overline{b}_i).$$

Die Alignment-Distanz *von $a, b \in \Sigma^*$ ist definiert als*

$$\overline{d}_w(a, b) := \min \left\{ w(\overline{a}, \overline{b}) \ : \ (\overline{a}, \overline{b}) \in \mathcal{A}(a, b) \right\}.$$

Für die Kostenfunktion gilt hier dasselbe wie für die Kostenfunktion der Edit-Distanz. Wählt man im obigen Beispiel in Abbildung 4.5 $w(x, x) = 0$ für $x \in \Sigma$ und $w(x, y) = 1$ für $x \neq y \in \overline{\Sigma}$, dann hat das gegebene Alignment eine Distanz von 3. Es gibt jedoch ein besseres Alignment mit Alignment-Distanz 2.

4.1.3 Beziehung zwischen Edit- und Alignment-Distanz

In diesem Abschnitt wollen wir zeigen, dass Edit-Distanz und Alignment-Distanz unter bestimmten Voraussetzungen gleich sind.

Lemma 4.10 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion und seien $a, b \in \Sigma^*$. Für jedes Alignment $(\overline{a}, \overline{b})$ von a und b gibt es eine Folge s von Edit-Operationen, so dass $a \overset{s}{\Rightarrow} b$ und $w(s) = w(\overline{a}, \overline{b})$.*

Beweis: Sei $(\overline{a}, \overline{b})$ ein Alignment für a und b. Dann konstruieren wir eine Folge $s = (s_1, \ldots, s_{|\overline{a}|})$ von Edit-Operationen wie folgt: $s_i = (\overline{a}_i, \overline{b}_i)$. Dann gilt offensichtlich, da $(\overline{a}, \overline{b})$ ein Alignment für a und b ist: $a \overset{s}{\Rightarrow} b$. Für die Edit-Distanz erhalten

wir somit:

$$w(s) = \sum_{i=1}^{|\overline{a}|} w(s_i) = \sum_{i=1}^{|\overline{a}|} w(\overline{a}_i, \overline{b}_i) = w(\overline{a}, \overline{b}).$$

■

Aus diesem Lemma folgt sofort das folgende Korollar, das besagt, dass die Edit-Distanz von zwei Zeichenreihen höchstens so groß ist wie die Alignment-Distanz.

Korollar 4.11 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion, dann gilt für alle $a, b \in \Sigma^*$:*

$$d_w(a, b) \leq \overline{d}_w(a, b).$$

Nun wollen wir die umgekehrte Richtung untersuchen.

Lemma 4.12 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine metrische Kostenfunktion und seien $a, b \in \Sigma^*$. Für jede Folge s von Edit-Operationen mit $a \overset{s}{\Rightarrow} b$ gibt es ein Alignment $(\overline{a}, \overline{b})$ von a und b, so dass $w(\overline{a}, \overline{b}) \leq w(s)$.*

Beweis: Wir führen den Beweis durch Induktion über $n = |s|$.

Induktionsanfang ($n = 0$): Aus $|s| = 0$ folgt, dass $s = \varepsilon$ (also die leere Folge von Edit-Operationen). Also ist $a = b$ und $w(s) = 0$. Wir setzen nun $\overline{a} = a = b = \overline{b}$ und erhalten ein Alignment $(\overline{a}, \overline{b})$ für a und b mit $w(\overline{a}, \overline{b}) = 0 \leq w(s)$ (aufgrund der Definitheit von w).

02.07.19

Induktionsschritt ($n \to n + 1$): Sei $s = (s_1, \ldots, s_n, s_{n+1})$ eine Folge von Edit-Operationen mit $a \overset{s}{\Rightarrow} b$. Sei nun $s' = (s_1, \ldots, s_n)$ und $a \overset{s'}{\Rightarrow} c \xrightarrow{s_{n+1}} b$ für ein $c \in \Sigma^*$ (da $a \overset{s}{\Rightarrow} b$ gilt, muss es ein solches c geben). Aus der Induktionsvoraussetzung folgt nun, dass es ein Alignment $(\overline{a}, \overline{c})$ von a, c gibt, so dass $w(\overline{a}, \overline{c}) \leq w(s')$.

Wir betrachten zuerst den Fall, dass die letzte Edit-Operation $s_{n+1} = (x, y)$ eine Substitution, ein Match oder eine Deletion ist, d.h. $x \in \Sigma$ und $y \in \overline{\Sigma}$. Wir können dann, wie in Abbildung 4.6 schematisch dargestellt, ein Alignment für a und b erzeugen,

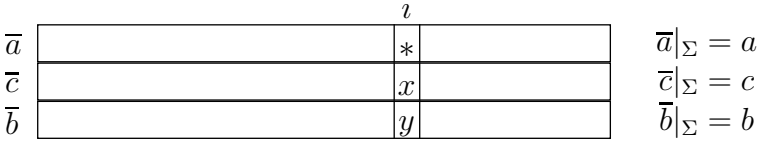


Abbildung 4.6: Skizze: $s_{n+1} = (x, y)$ ist eine Substitution, Match oder Deletion

indem wir die Zeichenreihe $\overline{b}$ geeignet aus $\overline{c}$ unter Verwendung der Edit-Operation (x, y) umformen.

Es gilt dann (außer wenn $\overline{a}_i = - = \overline{b}_i$):

$$\begin{aligned} w(\overline{a}, \overline{b}) &= w(\overline{a}, \overline{c}) \underbrace{-w(\overline{a}_i, \overline{c}_i) + w(\overline{a}_i, \overline{b}_i)}_{\leq w(\overline{c}_i, \overline{b}_i)} \\ &\quad \text{aufgrund der Dreiecksungleichung} \\ &\quad \text{d.h., } w(\overline{a}_i, \overline{b}_i) \leq w(\overline{a}_i, \overline{c}_i) + w(\overline{c}_i, \overline{b}_i) \\ &\leq \underbrace{w(\overline{a}, \overline{c})}_{\leq w(s')} + \underbrace{w(\overline{c}_i, \overline{b}_i)}_{= w(s_{n+1})} \\ &\overset{\text{I.V.}}{\leq} w(s') + w(s_{n+1}) \\ &= w(s). \end{aligned}$$

Den Fall $\overline{a}_i = \overline{b}_i = -$ muss man gesondert betrachten. Hier wird das verbotene Alignment von Leerzeichen im Alignment $(\overline{a}, \overline{b})$ eliminiert und wir erhalten das Alignment $(\overline{a'}, \overline{b'})$. Für dieses gilt nun:

$$\begin{aligned} w(\overline{a'}, \overline{b'}) &= w(\overline{a}, \overline{c}) - \underbrace{w(\overline{a}_i, \overline{c}_i)}_{\geq 0} \\ &\leq w(\overline{a}, \overline{c}) \\ &\leq \underbrace{w(\overline{a}, \overline{c})}_{\leq w(s')} + \underbrace{w(\overline{c}_i, \overline{b}_i)}_{\geq 0} \\ &\overset{\text{I.V.}}{\leq} w(s') + w(s_{n+1}) \\ &= w(s). \end{aligned}$$

Es bleibt noch der Fall, wenn $s_{n+1} = (-, y)$ mit $y \in \Sigma$ eine Insertion ist. Dann erweitern wir das Alignment $(\overline{a}, \overline{c})$ von a und c zu einem eigentlich *unzulässigen Alignment* $(\overline{a'}, \overline{c'})$ von a und c wie folgt. Es gibt ein $i \in [0 : |b|]$ mit $b_i = y$ und $b = c_1 \cdots c_i \cdot y \cdot c_{i+1} \cdots c_{|a|}$. Sei j die Position, nach der das Symbol y in $\overline{c}$ eingefügt wird. Dann setzen wir $\overline{a'} = \overline{a}_1 \cdots \overline{a}_j \cdot - \cdot \overline{a}_{j+1} \cdots \overline{a}_{|\overline{c}|}$, $\overline{c'} = \overline{c}_1 \cdots \overline{c}_j \cdot - \cdot \overline{c}_{j+1} \cdots \overline{c}_{|\overline{c}|}$ und $\overline{b'} = \overline{c}_1 \cdots \overline{c}_j \cdot y \cdot \overline{c}_{j+1} \cdots \overline{c}_{|\overline{c}|}$. Dies ist in Abbildung 4.7 noch einmal schematisch dargestellt.

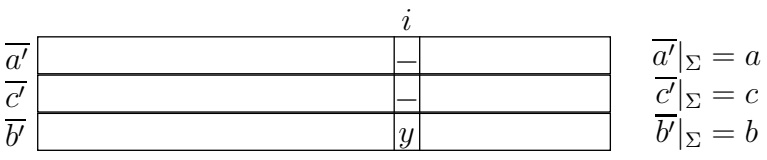


Abbildung 4.7: Skizze: $s_{n+1} = (x, y)$ ist eine Insertion

Nun ist $(\overline{a'}, \overline{c'})$ kein Alignment mehr, da es eine Spalte $(-,-)$ gibt. Wir interessieren uns jedoch jetzt nur noch für das Alignment $(\overline{a'}, \overline{b'})$ von a und b.

Damit gilt

$$\begin{aligned} w(\overline{a'}, \overline{b'}) &= w(\overline{a}, \overline{c}) + w(-, y) \\ &\quad \text{Nach Induktionsvoraussetzung} \\ &\leq w(s') + w(s_{n+1}) \\ &= w(s). \end{aligned}$$

Das Lemma ist somit bewiesen. ■

Aus diesem Lemma folgt jetzt, dass die Alignment-Distanz durch die Edit-Distanz beschränkt ist, sofern die zugrunde liegende Kostenfunktion eine Metrik ist.

Korollar 4.13 *Ist $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine metrische Kostenfunktion, dann gilt für alle $a, b \in \Sigma^*$:*

$$\overline{d}_w(a, b) \leq d_w(a, b).$$

Fasst man die beiden Korollare zusammen, so ergibt sich das folgende Theorem, dass die Edit-Distanz mit der Alignment-Distanz zusammenfallen, wenn die zugrunde gelegte Kostenfunktion eine Metrik ist.

Theorem 4.14 *Ist w eine Metrik, dann gilt: $d_w = \overline{d}_w$.*

Man überlege sich, ob alle drei Bedingungen der Metrik wirklich benötigt werden. Insbesondere für die Definitheit mache man sich klar, dass man zumindest auf die Gültigkeit von $w(x, x) = 0$ für alle $x \in \Sigma$ nicht verzichten kann.

Manchmal will man aber auf die Definitheit verzichten, d.h. manche Zeichen sollen gleicher als andere sein. Dann schwächt man die Bedingungen an die Kostenfunktion $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ wie folgt ab.

Definition 4.15 *Eine Kostenfunktion $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ für ein Distanzmaß heißt* sinnvoll*, wenn folgende Bedingungen erfüllt sind:*

(D1) $\forall x, y \in \Sigma : \; w(x, x) \leq w(x, y)$;

(D2) $\forall x, y \in \Sigma : w(x, x) \leq 2w(y, -)$;

(D3) $\forall x, y \in \overline{\Sigma} : w(x, y) = w(y, x)$;

(D4) $\forall x, y, z \in \overline{\Sigma} : w(x, z) \leq w(x, y) + w(y, z)$.

In der vorherigen Definition sind die (Un)gleichungen für $x, y, z \in \overline{\Sigma}$ nur dann gültig, wenn maximal ein Leerzeichen involviert ist.

Von der Definitheit bleibt hier also nur noch übrig, dass gleiche Zeichen einen geringeren Abstand (aber nicht notwendigerweise 0) haben müssen als verschiedene Zeichen (D1). Damit man in einem Alignment ein Match (x, x) nicht durch eine Deletion $(x, -)$ und eine Insertion $(-, x)$ von x ersetzen kann, wurde D2 eingeführt. D2 ist hier sogar noch etwas stärker formuliert, da es so in den folgenden Beweisen benötigt wird. Die Symmetrie (D3) und die Dreiecksungleichung (D4) übernehmen wir von einer Metrik mit derselben Argumentation wie dort.

Will man die Äquivalenz von Edit- und Alignment-Distanz weiterhin sicherstellen, so wird man ebenfalls $\max\{w(x, x) : x \in \Sigma\} = 0$ fordern. Im Folgenden werden wir für alle Kostenfunktionen für Distanzmaße voraussetzen, dass zumindest D1 mit D4 gelten.

4.1.4 Ähnlichkeitsmaße

Für manche Untersuchungen ist der Begriff der Ähnlichkeit von zwei Zeichen angemessener als der Begriff der Unterschiedlichkeit. Im letzten Abschnitt haben wir gesehen, wie wir Unterschiede zwischen zwei Zeichenreihen qualitativ und quantitativ fassen können. In diesem Abschnitt wollen wir uns mit der Ähnlichkeit von Zeichenreihen beschäftigen. Zum Ende dieses Abschnittes werden wir zeigen, dass sich die hier formalisierten Begriffe der Distanz und der Ähnlichkeit im Wesentlichen entsprechen.

Im Unterschied zu Distanzen werden gleiche Zeichen mit einem positiven Gewicht belohnt, während ungleiche Zeichen mit einen negativen Gewicht bestraft oder einem kleinen positiven Gewicht geringer belohnt werden. Man hat also insbesondere die Möglichkeit, die Gleichheit von gewissen Zeichen stärker zu bewerten als von anderen Zeichen. Formal lässt sich eine sinnvolle Kostenfunktion für Ähnlichkeitsmaße wie folgt definieren.

Definition 4.16 *Eine Kostenfunktion $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ für ein Ähnlichkeitsmaß heißt* sinnvoll*, wenn die folgenden Bedingungen erfüllt sind:*

(S1) $\forall x \in \Sigma : w'(x, x) \geq 0 \ \wedge \ w'(x, -) \leq 0$*;*

(S2) $\forall x, y \in \Sigma : w'(x, x) \geq w'(x, y)$*;*

(S3) $\forall x, y \in \overline{\Sigma} : w'(x, y) = w'(y, x)$*;*

(S4) $\forall x, y \in \Sigma : w'(x, y) \geq w'(x, -) + w'(-, y)$*.*

In der obigen Definition ist bei (S3) die Gleichung für $x, y \in \overline{\Sigma}$ nur dann gültig, wenn maximal ein Leerzeichen involviert ist.

In einer Kostenfunktion für ein Ähnlichkeitsmaß beschreiben positive Werte Ähnlichkeiten und negative Werte Unähnlichkeiten. Dies ist die Begründung der Forderung von S1. In jedem Fall sollte ein Zeichen mit sich selbst ähnlicher sein als zu einem anderen Zeichen (siehe S2). Wie bei Kostenfunktionen für Distanzmaße setzen wir auch für Ähnlichkeitsmaße mit derselben Argumentation die Symmetrie S3 für die zugrunde liegende Kostenfunktion voraus. Würde S4 nicht gelten, so wäre es in einem Alignment besser, die Substitution (x, y) durch eine Deletion von x und eine Insertion von y zu ersetzen, was in der Regel nicht sinnvoll ist.

Im Folgenden werden wir für alle Kostenfunktionen für Ähnlichkeitsmaße mindestens voraussetzen, dass S1 und S2 gelten. Basierend auf solchen Kostenfunktionen für Ähnlichkeitsmaße können wir jetzt die Ähnlichkeit von Alignments definieren.

Definition 4.17 *Sei $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine Kostenfunktion und sei $(\overline{a}, \overline{b})$ ein Alignment für $a, b \in \Sigma^*$. Dann ist die* Ähnlichkeit des Alignments *von $(\overline{a}, \overline{b})$ definiert als:*

$$w'(\overline{a}, \overline{b}) := \sum_{i=1}^{|\overline{a}|} w'(\overline{a}_i, \overline{b}_i).$$

Die Alignment-Ähnlichkeit *von $a, b \in \Sigma^*$ ist definiert als:*

$$s(a, b) := s_{w'}(a, b) := \max\left\{w'(\overline{a}, \overline{b}) \ : \ (\overline{a}, \overline{b}) \in \mathcal{A}(a, b)\right\}.$$

4.1.5 Beziehung zwischen Distanz- und Ähnlichkeitsmaßen

Die folgenden beiden Lemmata verdeutlichen die Beziehung, die zwischen Ähnlichkeitsmaß und Distanzmaß besteht.

Lemma 4.18 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine sinnvolle Kostenfunktion für ein Distanzmaß. Dann existiert ein $C \in \mathbb{R}_+$, so dass w' vermöge $w'(a, b) = C - w(a, b)$ und $w'(a, -) = \frac{C}{2} - w(a, -)$ für alle $a, b \in \Sigma$ eine sinnvolle Kostenfunktion für ein Ähnlichkeitsmaß ist.*

Beweis: Sei $C := \max\{w(x, x) \ : \ x \in \Sigma\} \geq 0$. Dann gilt für alle $a \in \Sigma$

$$w'(a, a) = C - w(a, a) = \max\{w(x, x) \ : \ x \in \Sigma\} - w(a, a) \geq 0$$

sowie mit D2

$$w'(a,-) = \frac{C}{2} - w(a,-) \leq \frac{1}{2}\left(\max\{w(x,x) \,:\, x \in \Sigma\} - 2w(a,-)\right) \overset{D2}{\leq} 0$$

und somit S1.

Für jedes $a \in \Sigma$ gilt aufgrund von D1 $w(a,a) \leq w(a,b)$ und somit:

$$w'(a,a) = C - w(a,a) \geq C - w(a,b) = w'(a,b).$$

Damit haben wir die Eigenschaft S2 nachgewiesen.

Die Eigenschaft S3 folgt offensichtlich aus der Definition von w' und D3.

Für den Nachweis von S4 gilt für $a,b \in \Sigma$ mit Hilfe der Dreiecksungleichung:

$$w'(a,b) = C - w(a,b) \geq C - w(a,-) - w(-,b) = w'(a,-) + w'(b,-).$$

Damit haben wir die Eigenschaft S4 nachgewiesen. ■

Lemma 4.19 *Sei $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine sinnvolle Kostenfunktion für ein Ähnlichkeitsmaß. Dann existiert ein $D \in \mathbb{R}_+$, so dass w vermöge $w(a,b) = D - w'(a,b)$ und $w(a,-) = \frac{D}{2} - w'(a,-)$ für alle $a,b \in \Sigma$ eine sinnvolle Kostenfunktion für ein Distanzmaß ist.*

Beweis: Wir wählen $A = \max\left\{w'(x,y) + w'(y,z) - w'(x,z) \,:\, x,y,z \in \overline{\Sigma}\right\} \geq 0$ und $B = \max\{w'(x,y) \,:\, x,y \in \Sigma\} \geq 0$ (jeweils unter Verwendung von $x = y(= z)$ und S1). Dann setzen wir $D = \max\{A,B\} \geq 0$. Nach Wahl von D gilt dann für $a,b \in \Sigma$:

$$w(a,b) = D - w'(a,b) \geq \max\{w'(x,y) \,:\, x,y \in \Sigma\} - w'(a,b) \geq 0.$$

Weiter gilt wegen S1, dass $w'(a,-) \leq 0$ und daher ist $w(a,-) = \frac{D}{2} - w'(a,-) \geq 0$. Somit haben wir nachgewiesen, dass $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$.

Wir weisen jetzt D1 nach. Wegen S2 gilt für alle $a,b \in \Sigma$, dass $w'(a,a) \geq w'(a,b)$, und somit gilt:

$$w(a,a) = D - w'(a,a) \leq D - w'(a,b) = w(a,b).$$

Nach S1 gilt $w'(a,a) \geq 0 \geq w'(b,-)$ für alle $a,b \in \Sigma$. Daraus folgt, dass auch $w'(a,a) \geq 0 \geq 2w'(b,-)$ für alle $a,b \in \Sigma$ gilt. Somit erhalten wir die Eigenschaft D2 mittels:

$$w(a,a) = D - w'(a,a) \leq D - 2w'(b,-) = 2w(b,-).$$

D3 folgt nach Definition von w unmittelbar aus S3.

Wir müssen nur noch D4 beweisen, d.h. dass $w(a,c) \leq w(a,b) + w(b,c)$ für alle $a,b,c \in \overline{\Sigma}$ gilt. Betrachten wir zuerst den Fall $a,b,c \in \Sigma$, dann gilt

$$w(a,c) = D - w'(a,c) \quad \text{und} \quad w(a,b) + w(b,c) = 2D - w'(a,b) - w'(b,c).$$

Damit genügt es zu zeigen, dass $w'(a,b)+w'(b,c)-w'(a,c) \leq D$ gilt. Nach Definition von A und D gilt jedoch $w'(a,b) + w'(b,c) - w'(a,c) \leq A \leq D$.

Es bleiben die Fälle, in denen genau ein Leerzeichen involviert ist. Sei zunächst $c = -$. Es gilt nun

$$w(a,-) = \frac{D}{2} - w'(a,-) \quad \text{und} \quad w(a,b) + w(a,-) = \frac{3D}{2} - w'(a,b) - w'(b,-).$$

Damit genügt es zu zeigen, dass $w'(a,b) + w'(b,-) - w'(a,-) \leq D$ gilt. Nach Definition von A und D gilt jedoch $w'(a,b) + w'(b,-) - w'(a,-) \leq A \leq D$. Der Fall $a = -$ ist analog.

Sei abschließend $b = -$. Es gilt jetzt

$$w(a,c) = D - w'(a,c) \quad \text{und} \quad w(a,-) + w(c,-) = D - w'(a,-) - w'(c,-).$$

Es genügt also zu zeigen, dass $w'(a,c) \geq w'(a,-) + w'(c,-)$. Dies ist aber gerade die Bedingung S4 und der Beweis ist damit abgeschlossen. ■

Damit haben wir trotz der unterschiedlichen Definition gesehen, dass Distanzen und Ähnlichkeiten eng miteinander verwandt sind. Dies unterstreicht insbesondere auch der folgende Satz.

Theorem 4.20 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine (sinnvolle) Kostenfunktion für ein Distanzmaß d und sei $C \in \mathbb{R}_+$ so gewählt, dass $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ vermöge $w'(a,b) = C - w(a,b)$ und $w'(a,-) = \frac{C}{2} - w(a,-)$ für alle $a,b \in \Sigma$ eine (sinnvolle) Kostenfunktion für ein Ähnlichkeitsmaß s ist. Dann gilt für alle $a,b \in \Sigma^*$:*

$$\overline{d}_w(a,b) + s_{w'}(a,b) = \frac{C}{2}(|a| + |b|).$$

Beweis: Im Folgenden bezeichne für ein $(\overline{a},\overline{b}) \in \mathcal{A}(a,b)$:

$$\begin{aligned} M \;:=\; M(\overline{a},\overline{b}) \;&:=\; \left\{ i \in [1 : |\overline{a}|] \;:\; \overline{a}_i, \overline{b}_i \in \Sigma \right\} \\ I \;:=\; I(\overline{a},\overline{b}) \;&:=\; [1 : |\overline{a}|] \setminus M(\overline{a},\overline{b}). \end{aligned}$$

Weiter bezeichne $\mu(\overline{a},\overline{b}) = |M(\overline{a},\overline{b})|$ bzw. $\iota(\overline{a},\overline{b}) = |I(\overline{a},\overline{b})|$ die Anzahl von Matches und Mismatches (Substitutionen) bzw. von Indel-Operationen eines Alignments $(\overline{a},\overline{b})$. Beachte, dass für jedes Alignment $(\overline{a},\overline{b})$ für $a,b \in \Sigma^*$ gilt:

$$|a| + |b| = 2\mu(\overline{a},\overline{b}) + \iota(\overline{a},\overline{b}).$$

Es gilt dann:

$$\begin{aligned}
& s_{w'}(a,b) \\
&= \max\left\{ \sum_{j=1}^{|\overline{a}|} w'(\overline{a}_j,\overline{b}_j) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \max\left\{ \sum_{j\in M(\overline{a},\overline{b})} w'(\overline{a}_j,\overline{b}_j) + \sum_{j\in I(\overline{a},\overline{b})} w'(\overline{a}_j,\overline{b}_j) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \max\left\{ \sum_{j\in M(\overline{a},\overline{b})} (C - w(\overline{a}_j,\overline{b}_j)) + \sum_{j\in I(\overline{a},\overline{b})} \left(\frac{C}{2} - w(\overline{a}_j,\overline{b}_j)\right) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \max\left\{ C\cdot\mu(\overline{a},\overline{b}) + \frac{C}{2}\cdot\iota(\overline{a},\overline{b}) \right. \\
&\qquad \left. - \sum_{j\in M(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) - \sum_{j\in I(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) \quad:\quad (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \max\left\{ \frac{C}{2}(|a|+|b|) - \sum_{j\in M(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) - \sum_{j\in I(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \frac{C}{2}(|a|+|b|) - \min\left\{ \sum_{j\in M(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) + \sum_{j\in I(\overline{a},\overline{b})} w(\overline{a}_j,\overline{b}_j) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \frac{C}{2}(|a|+|b|) - \min\left\{ \sum_{j=1}^{|\overline{a}|} w(\overline{a}_j,\overline{b}_j) \;:\; (\overline{a},\overline{b}) \in \mathcal{A}(a,b) \right\} \\
&= \frac{C}{2}(|a|+|b|) - \overline{d}_w(a,b).
\end{aligned}$$

Damit ist der Satz bewiesen. ■

Unter gewissen Voraussetzungen an die zugrunde liegenden Kostenfunktionen (siehe den vorhergehenden Satz) kann man also beliebig zwischen Edit-, Alignment-Distanz

oder Ähnlichkeiten wechseln, ohne in der Bewertung der Ähnlichkeit von Sequenzen andere Ergebnisse zu erhalten. Dies gilt jedoch so nur für globale Alignments.

4.2 Globale Alignments

In diesem Abschnitt geht es nun darum, optimale Alignments für zwei Zeichenreihen tatsächlich zu berechnen. Dabei werden die beiden Zeichenreihen zueinander aligniert, also so zueinander ausgerichtet, dass sie danach dieselbe Länge haben und so ähnlich wie möglich sind (bezüglich eines geeignet gewählten Distanz- oder Ähnlichkeitsmaßes).

Globales Alignment

Eingabe: Seien $s \in \Sigma^n$ und $t \in \Sigma^m$ und sei w eine Kostenfunktion für ein Distanz- oder Ähnlichkeitsmaß μ.

Gesucht: Ein optimales globales Alignment $(\overline{s}, \overline{t})$ für s und t, d.h. $\mu(s,t) = w(\overline{s}, \overline{t})$.

Wir werden uns zunächst hauptsächlich mit Distanzmaßen beschäftigen. Es ist meist offensichtlich, wie die Methoden für Ähnlichkeitsmaße zu modifizieren sind. Wir fordern den Leser an dieser Stelle ausdrücklich auf, dies auch jedes Mal zu tun.

4.2.1 Der Algorithmus nach Needleman-Wunsch

Wir nehmen an, wir kennen schon ein optimales Alignment $(\overline{s}, \overline{t})$ für s und t. Es gibt jetzt drei Möglichkeiten, wie die letzte Spalte $(\overline{t}_{|\overline{t}|}, \overline{s}_{|\overline{s}|})$ dieses optimalen Alignments aussehen kann. Entweder wurde $x = s_n$ durch $y = t_m$ substituiert (siehe Abbildung 4.8 oben) oder es wurde das letzte Zeichen $x = s_n$ in s gelöscht (siehe Abbildung 4.8 Mitte) oder es wurde das letzte Zeichen $y = t_m$ in t eingefügt (siehe Abbildung 4.8 unten).

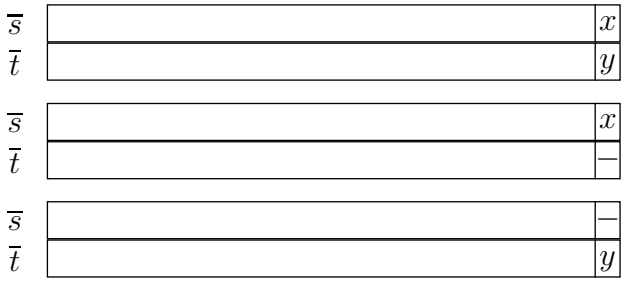


Abbildung 4.8: Skizze: Letzte Spalte eines optimalen Alignments

In allen drei Fällen überlegt man sich, dass das Alignment, das durch Streichen der letzten Spalte eines optimalen Alignments entsteht, also $(\overline{s}_1 \cdots \overline{s}_{|\overline{s}|-1}, \overline{t}_1 \cdots \overline{t}_{|\overline{t}|-1})$, ebenfalls ein optimales Alignment für das entsprechende Sequenzen-Paar $s_1 \cdots s_{n-1}$ mit $t_1 \cdots t_{m-1}$, $s_1 \cdots s_{n-1}$ mit $t_1 \cdots t_m$ oder $s_1 \cdots s_n$ mit $t_1 \cdots t_{m-1}$ sein muss. Gäbe es andernfalls ein besseres Alignment (mit geringerer Distanz), so könnte man daraus mit der Edit-Operation an der letzten Stelle ein besseres Alignment für s mit t konstruieren. Das folgende Lemma beweist dies formal in einem allgemeineren Fall.

Lemma 4.21 *Sei $(\overline{a}, \overline{b})$ ein optimales Alignment für $a, b \in \Sigma^*$ für ein gegebenes Distanz- oder Ähnlichkeitsmaß. Für alle $i \leq j \in [1 : |\overline{a}|]$ ist dann $(\overline{a}_i \cdots \overline{a}_j, \overline{b}_i \cdots \overline{b}_j)$ ein optimales Alignment für $a' = \overline{a}_i \cdots \overline{a}_j|_\Sigma$ und $b' = \overline{b}_i \cdots \overline{b}_j|_\Sigma$.*

Beweis: Sei $(\overline{a}, \overline{b})$ ein optimales Alignment für $a, b \in \Sigma^*$. Für einen Widerspruchsbeweis nehmen wir an, dass $(\overline{a}_i \cdots \overline{a}_j, \overline{b}_i \cdots \overline{b}_j)$ kein optimales Alignment für $a', b' \in \Sigma$ ist. Sei also $(\widetilde{a'}, \widetilde{b'})$ ein optimales Alignment für a' und b'. Dann ist aber nach Definition der Kosten eines Alignments (unabhängig, ob Distanz- oder Ähnlichkeitsmaß) das Alignment

$$(\overline{a}_1 \cdots \overline{a}_{i-1} \cdot \widetilde{a'} \cdot \overline{a}_{j+1} \cdots \overline{a}_n, \overline{b}_1 \cdots \overline{b}_{i-1} \cdot \widetilde{b'} \cdot \overline{b}_{j+1} \cdots \overline{b}_n)$$

ein besseres Alignment als $(\overline{a}, \overline{b})$ und wir erhalten einen Widerspruch. ■

Mit diesen Überlegungen können wir den *Algorithmus von Needleman und Wunsch* formulieren, der ein optimales Alignment für zwei Sequenzen $s = s_1 \cdots s_n \in \Sigma^n$ und $t = t_1 \cdots t_m \in \Sigma^m$ berechnet. Dazu wird eine Matrix

$$D(i, j) = \mu(s_1 \cdots s_i, t_1 \cdots t_j)$$

aufgestellt, in welcher jeweils die Distanz eines optimalen Alignments für $s_1, \ldots s_i$ und $t_1, \ldots t_j$ abgespeichert wird. Die Matrix kann rekursiv mit der folgenden Rekursionsformel berechnet werden (wobei wir im Folgenden μ der Einfachheit halber als Distanzmaß annehmen):

$$D(i,j) \;=\; \min \left\{ \begin{array}{lcl} D(i-1,j-1) & + & w(s_i, t_j), \\ D(i-1,j) & + & w(s_i, -), \\ D(i,j-1) & + & w(-, t_j) \end{array} \right\}.$$

Die folgenden Skizzen in Abbildung 4.9 illustrieren nochmals obige Rekursionsformel. Im ersten Fall ist das optimale Alignment für $s_1 \cdots s_{i-1}$ und $t_1 \cdots t_{j-1}$ bereits berechnet und in $D(i-1, j-1)$ abgelegt. Um nun die Distanz eines Alignments für $s_1 \cdots s_i$ und $t_1 \cdots t_j$ zu erhalten, müssen noch die Kosten $w(s_i, t_j)$ für die Substitution von s_i durch t_j hinzuaddieren. Im zweiten Fall wurde ein Zeichen in t gelöscht.

$\overline{s_1 \cdots s_{i-1}}$	s_i
$\overline{t_1 \cdots t_{j-1}}$	t_j

$D(i,j) = D(i-1, j-1) + w(s_i, t_j)$

$\overline{s_1 \cdots s_{i-1}}$	s_i
$\overline{t_1 \cdots t_j}$	$-$

$D(i,j) = D(i-1, j) + w(s_i, -)$

$\overline{s_1 \cdots s_i}$	$-$
$\overline{t_1 \cdots t_{j-1}}$	t_j

$D(i,j) = D(i, j-1) + w(-, t_j)$

Abbildung 4.9: Skizze: Erweiterung eines optimalen Alignments zu $s_1 \cdots s_i$ mit $t_1 \cdots t_j$

Um nun die Distanz eines Alignments für $s_1 \cdots s_i$ und $t_1 \cdots t_j$ zu erhalten, muss zu den Kosten $w(s_i, -)$ dieser Löschung noch die Distanz des bereits berechneten optimalen Alignments für $s_1 \cdots s_{i-1}$ und $t_1 \cdots t_j$ dazuaddiert werden. Im letzten Fall wurde ein Zeichen in die Sequenz t eingefügt. Wie bei den anderen beiden Fällen auch, müssen zur Distanz des bereits berechneten optimalen Alignments für $s_1 \cdots s_i$ und $t_1 \cdots t_{j-1}$, noch die Kosten $w(-, t_j)$ für die Einfügung hinzuaddiert werden, um die Distanz eines Alignments für $s_1 \cdots s_i$ und $t_1 \cdots t_j$ zu erhalten. Da das Optimum, wie vorher schon erläutert, einer dieser Fälle ist, genügt es, aus allen drei möglichen Werten das Minimum auszuwählen. Im Falle von Ähnlichkeitsmaßen wird dann entsprechend das Maximum genommen.

Die Abbildung 4.10 zeigt schematisch, wie der Wert eines Eintrags $D(i,j)$ in der Matrix D von den anderen Werten aus D abhängt. Nun fehlt nur noch der zuge-

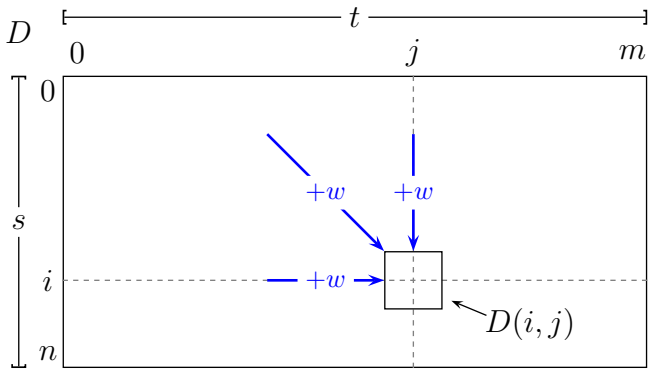


Abbildung 4.10: Skizze: Berechnung optimaler Alignments nach Needleman-Wunsch

SequenceAlignment (char $s[]$, int n, char $t[]$, int m)

begin

$D[0,0] := 0;$

for $(i := 1;\ i \leq n;\ i{+}{+})$ **do**

$D[i,0] := D[i-1,0] + w(s_i, -);$

for $(j := 1;\ j \leq m;\ j{+}{+})$ **do**

$D[0,j] := D[0,j-1] + w(-, t_j);$

for $(i := 1;\ i \leq n;\ i{+}{+})$ **do**

for $(j := 1;\ j \leq m;\ j{+}{+})$ **do**

$$D[i,j] := \min \left\{ \begin{array}{l} D[i-1,j] + w(s_i, -), \\ D[i,j-1] + w(-, t_j), \\ D[i-1,j-1] + w(s_i, t_j) \end{array} \right\};$$

end

Abbildung 4.11: Algorithmus: Verfahren von Needleman und Wunsch

hörigen Anfangswert: $D(0,0) = 0$, da für zwei leere Sequenzen das leere Alignment das optimale ist. Der vollständige Algorithmus von Needleman und Wunsch ist in Abbildung 4.11 angegeben.

03.07.19

Das folgende Beispiel zeigt ausführlich wie für zwei Sequenzen $s = AGGCTG$ und $t = ACCGGTA$ deren optimale Alignment-Distanz bestimmt wird. In den nachfolgenden Abbildungen gilt, dass die Zeichenreihe s immer vertikal und die Zeichenreihe t immer horizontal aufgetragen ist.

Der erste Schritt besteht darin, den so genannten *Edit-Graphen* aufzustellen, siehe Abbildung 4.12. Dazu wird die Sequenz t horizontal und s vertikal aufgetragen. Anschließend werden in den Graphen Pfeile eingefügt, abhängig davon, ob an der jeweiligen Stelle eine Substitution, eine Löschung, eine Einfügung oder aber ein Match auftritt. Ein horizontaler bzw. vertikaler blauer Pfeil steht für eine Insertion bzw. für eine Deletion, ein roter Pfeil für eine Substitution und schließlich ein grüner Pfeil für einen Match.

Daraufhin wird der Edit-Graph mit Zahlen gefüllt, siehe Abbildung 4.13. An die jeweilige Stelle im Graphen wird die korrekte Distanz des jeweiligen Alignments der entsprechenden Präfixe von s und t eingetragen, die mit der weiter oben angegeben Rekursionformel berechnet werden. Im Beispiel wird von einer Kostenfunktion ausgegangen, welche einer Löschung und einer Einfügung die Kosten 2 zuordnet, und bei welcher eine Substitution die Kosten 3 verursacht. Ein Match verursacht hier natürlich keine Kosten.

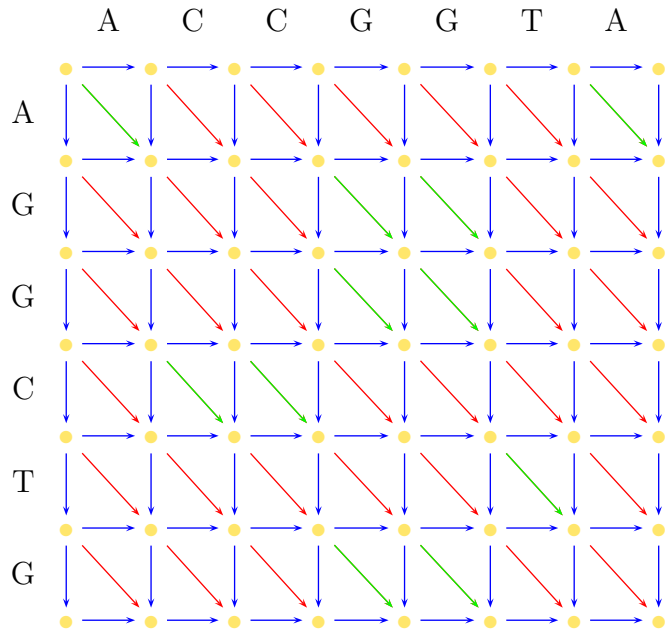
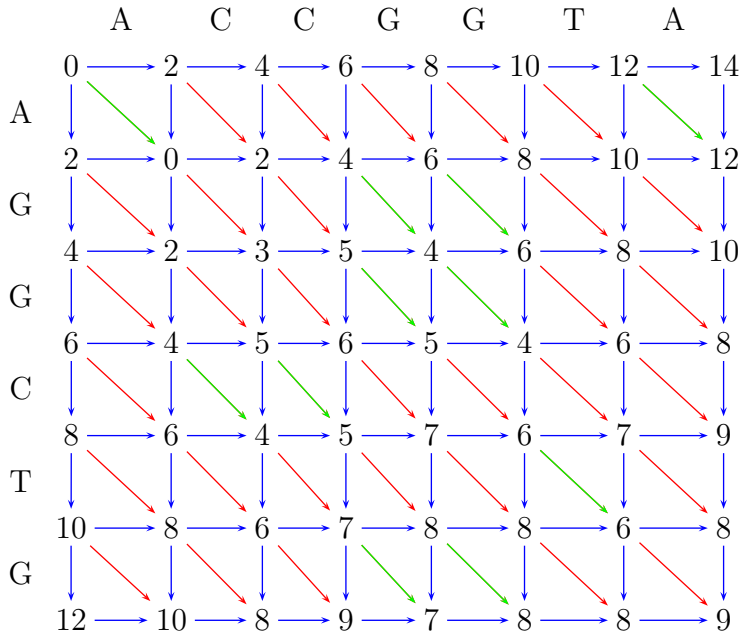
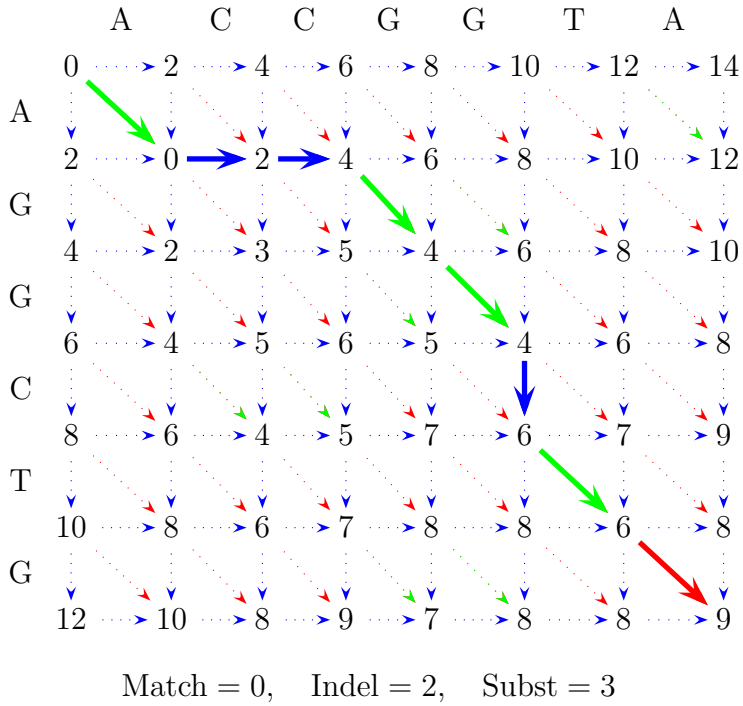


Abbildung 4.12: Skizze: Edit-Graph für s und t ohne Distanzen



Match = 0, Indel = 2, Subst = 3

Abbildung 4.13: Skizze: Edit-Graph für s und t mit Distanzen



Match = 0, Indel = 2, Subst = 3

Abbildung 4.14: Skizze: Pfad im Edit-Graphen zur Bestimmung des Alignments

In der rechten unteren Ecke, also in $D(n, m)$, steht nun die Distanz eines optimalen Alignments für s und t. Damit haben wir zwar den Wert eines optimalen Alignments für s und t bestimmt, kennen das Alignment an sich jedoch noch nicht. Um nun dieses zu erhalten, wird ein Pfad im Graphen von rechts unten nach links oben gesucht, der minimale Kosten verursacht.

Dieser Pfad wird folgendermaßen gefunden. Gestartet wird in der rechten unteren Ecke. Als Vorgängerknoten wird nun der Knoten gewählt, der zuvor als Sieger bei der Minimum-Bildung hervorging. Liefern mehrere Knoten die gleichen minimalen Kosten, kann einer davon frei gewählt werden. Meist geht man hier in einer vorher fest vorgegeben Reihenfolge bei Unentschieden vor, z.B. Insertion vor Substitution vor Deletion. So verfährt man nun immer weiter, bis man in der linken oberen Ecke ankommt.

Nun ist es nicht mehr schwer, das optimale Alignment für s und t anzugeben. Dieses muss nur noch aus dem Edit-Graphen (entlang des gefundenen Pfades) abgelesen werden, wie dies in Abbildung 4.15 dargestellt ist. Blaue horizontale bzw. vertikalen

s :	A	–	–	G	G	C	T	G
t :	A	C	C	G	G	–	T	A

Abbildung 4.15: Beispiel: Optimales globales Alignment von s mit t

Kanten entsprechen Einfügungen bzw. Löschungen, gründe bzw. rote Diagonalen entsprechen Matches bzw. Substitutionen.

Fassen wir zum Abschluss dieses Abschnittes noch das Ergebnis zusammen.

Theorem 4.22 *Das optimale globale paarweise Sequenzen-Alignment für s und t mit $n = |s|$ und $m = |t|$ sowie die zugehörige Alignment-Distanz lassen sich in Zeit $O(nm)$ und mit Platz $O(nm)$ berechnen.*

Zum Schluss dieses Abschnitts wollen wir noch anmerken, dass wir hier wiederum auf das bereits im zweiten Kapitel vorgestelle Programmierparadigma der *dynamischen Programmierung* zurückgegriffen haben.

4.2.2 Sequenzen-Alignment mit linearem Platz (Hirschberg)

Bisher wurde zur Bestimmung eines optimalen Alignments für s und t Platz in der Größenordnung $O(nm)$ benötigt. Dies soll nun dahingehend optimiert werden, dass nur noch linear viel Platz gebraucht wird.

Es fällt auf, dass während der Berechnung der $D(i, j)$ immer nur die momentane Zeile i und die unmittelbar darüber liegende Zeile $i-1$ benötigt wird. Somit bietet es sich an, immer nur diese beiden relevanten Zeilen zu speichern und somit nur linear viel Platz zu beanspruchen. Der Algorithmus in Abbildung 4.16 gibt ein Implementierung

SequenceAlignment (char $s[]$, int n, char $t[]$, int m)

begin
  $D[0] := 0$;
  for $(j := 1; j \leq m; j{+}{+})$ **do**
    $D[j] := D[j-1] + w(-, t_j)$;
  for $(i := 1; i \leq n; i{+}{+})$ **do**
    for $(j := 0; j \leq m; j{+}{+})$ **do** $D'[j] := D[j]$;
    $D[0] := D'[0] + w(s_i, -)$;
    for $(j := 1; j \leq m; j{+}{+})$ **do**

$$D[j] := \min \left\{ \begin{array}{l} D'[j] + w(s_i, -), \\ D[j-1] + w(-, t_j), \\ D'[j-1] + w(s_i, t_j) \end{array} \right\};$$

end

Abbildung 4.16: Algorithmus: platzsparende Variante von Needleman-Wunsch

genau dieses Verfahren an, wobei $D'[j]$ bzw. $D[j]$ quasi für $D[i-1,j]$ bzw. $D[i,j]$ steht.

Mit dem oben beschriebenen Verfahren lässt sich die Distanz der beiden Sequenzen s und t mit linearem Platz berechnen. Allerdings hat das Verfahren den Haken, dass das Alignment selbst nicht mehr einfach anhand des Edit-Graphen aufgebaut werden kann, weil ja die nötigen Zwischenergebnisse nicht gespeichert wurden.

Mit dem Verfahren nach Hirschberg kann ein optimales Sequenzen-Alignment selbst konstruiert werden, so dass insgesamt nur linear viel Platz benutzt wird. Dazu betrachten wir zunächst einmal ein optimales paarweises Alignment von s mit t, wie in der folgenden Abbildung 4.17 angegeben. Wir teilen nun dieses Alignment so in zwei Teil-Alignments auf, dass beide Teile in etwa die Hälfte der Zeichen aus s enthalten: der erste Teil enthalte $\lfloor n/2 \rfloor$ und der zweite Teil $\lceil n/2 \rceil$ Zeichen aus s.

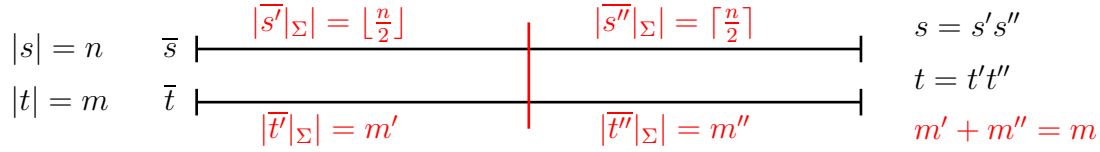


Abbildung 4.17: Skizze: Optimales Alignment für s und t

Wir merken an dieser Stelle noch an, dass dieser Aufteilungsschritt nicht eindeutig sein muss, da das Alignment $\overline{s}$ von s sehr viele Leerzeichen zwischen dem Zeichen $s_{\lfloor n/2 \rfloor}$ und dem Zeichen $s_{\lfloor n/2 \rfloor + 1}$ enthalten kann.

Im Folgenden bezeichnen wir mit m' die Anzahl der Zeichen aus t die bei der Aufteilung in der ersten Hälfte des Alignments sind und mit m'' die Anzahl der Zeichen in der zweiten Hälfte. Es gilt also $m = m' + m''$. Weiter bezeichne $\overline{s'}$ und $\overline{s''}$ bzw. $\overline{t'}$ und $\overline{t''}$ die Teile des Alignments nach der Aufteilung, d.h. $\overline{s} = \overline{s'} \cdot \overline{s''}$ und $\overline{t} = \overline{t'} \cdot \overline{t''}$. Ferner gilt $s' = \overline{s'}|_\Sigma$ und $s'' = \overline{s''}|_\Sigma$ bzw. $t' = \overline{t'}|_\Sigma$und $t'' = \overline{t''}|_\Sigma$. Es gilt also $s = s' \cdot s''$ und $t = t' \cdot t''$ sowie $|s'| = \lfloor n/2 \rfloor$ bzw. $|s''| = \lceil n/2 \rceil$ und $|t'| = m'$ bzw. $|t''| = m'' = m - m'$.

Zuerst einmal erinnern wir uns an Lemma 4.21, dass sowohl $(\overline{s'}, \overline{t'})$ ein optimales Alignment für s' mit t' sein muss, als dass auch $(\overline{s''}, \overline{t''})$ ein optimales Alignment für s'' mit t'' sein muss. Auch hier könnten wir andererseits aus besseren Alignments für s' mit t' bzw. s'' mit t'' ein besseres Alignment für s mit t konstruieren.

Dies führt uns unmittelbar auf die folgende *algorithmische Idee*: Berechne optimale Alignments für $s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_{m'}$ sowie für $s_{\lfloor n/2 \rfloor + 1} \cdots s_n$ mit $t_{m'+1} \cdots t_m$. Dieser Ansatz führt uns auf einen Divide-and-Conquer-Algorithmus, da wir nun rekursiv für kleinere Eingaben ein Problem derselben Art lösen müssen.

Der *Conquer-Schritt* ist dabei trivial, da wir einfach die beiden erhalten Alignments für beide Teile zu einem großen Alignment für s mit t zusammenhängen müssen, wie dies in der folgenden Abbildung 4.18 dargestellt ist.

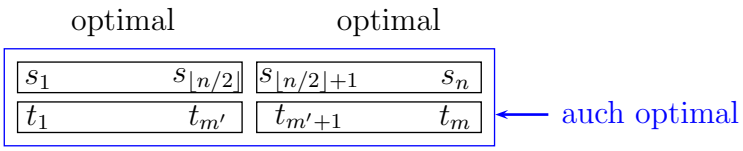


Abbildung 4.18: Skizze: Conquer-Schritt des Hirschberg-Algorithmus

Allerdings haben wir nun noch ein kleines Problem übersehen. Wir kennen nämlich m' noch nicht. Wir haben m' ja über ein optimales Alignment für s mit t definiert. Wenn wir also erst das optimale Alignment für s mit t berechnen wollen, kennen wir $m' = |t'|$ ja noch nicht.

Wie finden wir m' jetzt also? Ein erster naiver Gedanke ist, dass man alle möglichen $m' \in [0 : m]$ ausprobiert. Dies sind allerdings recht viele, die uns ja dann auch noch $m + 1$ rekursiv zu lösende Teilprobleme zur Aufgabe stellen.

So dumm, wie der naive Ansatz jedoch zuerst klingt, ist er gar nicht, denn wir wollen ja gar nicht die Alignments selbst berechnen, sondern nur wissen, wie m' für ein optimales Alignment von s mit t aussieht. Für die weitere Argumentation werden wir die folgende Abbildung 4.19 zu Hilfe nehmen.

In dieser Abbildung ist die Tabelle $D(i, j)$ wieder als Edit-Graph bildlich dargestellt. Ein optimales Alignment entspricht in diesem Graphen einem Weg von $(0, 0)$ nach (n, m). Offensichtlich muss dieser Weg die Zeile $\lfloor n/2 \rfloor$ an einem Punkt m' schnei-

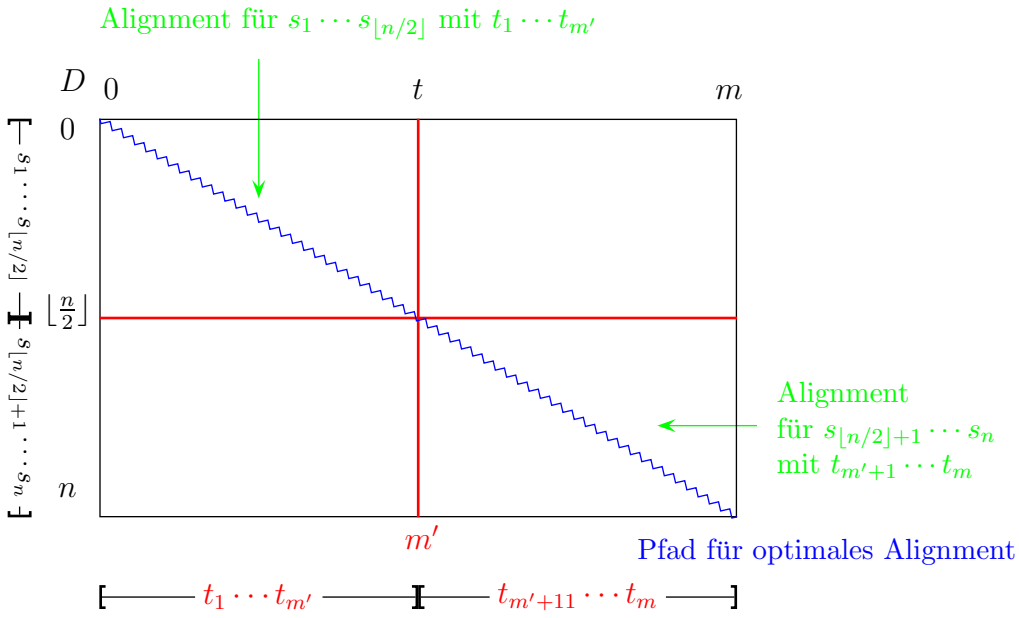


Abbildung 4.19: Skizze: Auffinden von m' (Divide-Schritt bei Hirschberg)

den. Wir merken hier nochmals an, dass dieser Punkt nicht eindeutig sein muss, da aufgrund von Insertionen der Pfad waagerecht innerhalb der Zeile $\lfloor n/2 \rfloor$ verlaufen kann.

Der Pfad von $(0,0)$ nach (n,m) zerfällt also in zwei Teile, nämlich in $(0,0)$ bis $(\lfloor n/2 \rfloor, m')$ und in $(\lfloor n/2 \rfloor, m')$ nach (n,m). Diese beiden Teile entsprechen dann genau den vorher diskutierten optimalen Alignments für s' mit t' und s'' mit t''.

Können wir die beiden Teilpfade jetzt schnell finden? Den ersten auf jeden Fall. Wir berechnen die optimalen Alignment-Distanzen von $s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_{m'}$ für alle $m' \in [0:m]$. Dies können wir mit unserem vorhin in Abbildung 4.16 vorgestellten Algorithmus in linearem Platz berechnen. Dort haben wir als Endergebnis das Feld $D[j]$ erhalten, dass die Alignment-Distanzen von s zu allen Präfixen von t enthielt.

Jetzt brauchen wir noch den zweiten Teil des Pfades. Dazu benötigen wir insbesondere die Alignment-Distanzen von $s_{\lfloor n/2 \rfloor+1} \cdots s_n$ mit $t_{m'+1} \cdots t_m$ für alle $m' \in [0:m]$. Diese können wir jedoch mit demselben Algorithmus berechnen. Wir stellen uns die Tabelle nur um 180 Grad um den Mittelpunkt gedreht vor. Wir berechnen dann alle Alignment-Distanzen von $(s'')^R = s_n \cdots s_{\lfloor n/2 \rfloor+1}$ mit $(t'')^R = t_m \cdots t_{m'+1}$, wobei hier x^R für eine Zeichenreihe $x = x_1 \cdots x_n$ die *gespiegelte oder reversierte Zeichenreihe* $x^R = x_n \cdots x_1$ bezeichnet.

Damit die Korrektheit dieses Ansatzes gilt, müssen wir nur den folgenden Satz beweisen.

Theorem 4.23 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ bzw. $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ bzw. eine Kostenfunktion für das Distanzmaß $\overline{d}_w$ bzw. für das Ähnlichkeitsmaß $s_{w'}$ und seien $s, t \in \Sigma^*$, dann gilt $\overline{d}_w(s,t) = \overline{d}_w(s^R, t^R)$ bzw. $s_{w'}(s,t) = s_{w'}(s^R, t^R)$.*

Beweis: Wir führen den Beweis nur für das Distanzmaß $\overline{d}_w$. Es gilt für beliebige $\overline{s}, \overline{t} \in \overline{\Sigma}^*$ mit $\overline{s}|_\Sigma = s$ und $\overline{t}|_\Sigma = t$:

$$w(\overline{s}, \overline{t}) = \sum_{i=1}^{|\overline{s}|} w(\overline{s}_i, \overline{t}_i) = \sum_{i=1}^{|\overline{s}|} w(\overline{s}_{|\overline{s}|-i+1}, \overline{t}_{|\overline{t}|-i+1}) = w(\overline{s}^R, \overline{t}^R).$$

Somit gilt auch:

$$\begin{aligned} \overline{d}_w(s,t) &= \min\left\{w(\overline{s}, \overline{t}) \ : \ (\overline{s}, \overline{t}) \in \mathcal{A}(s,t)\right\} \\ &= \min\left\{w(\overline{s}^R, \overline{t}^R) \ : \ (\overline{s}^R, \overline{t}^R) \in \mathcal{A}(s^R, t^R)\right\} \\ &= \overline{d}_w(s^R, t^R). \end{aligned}$$

Der Beweis für ein Ähnlichkeitsmaß verläuft völlig analog. ∎

Bezeichne $V(\lfloor n/2 \rfloor, k)$ die minimale Alignment-Distanz von $s' = s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_k$ und $V'(\lfloor n/2 \rfloor, k)$ die minimale Alignment-Distanz von $s'' = s_{\lfloor n/2 \rfloor + 1} \cdots s_n$ mit $t_{k+1} \cdots t_m$ was nach dem vorherigen Satz gleichbedeutend mit der minimalen Alignment-Distanz von $(s'')^R$ mit $t_m \cdots t_{k+1}$ ist.

$$\begin{aligned} V\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) &= \overline{d}(s_1 \cdots s_{\lfloor n/2 \rfloor}, t_1 \cdots t_k), \\ V'\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) &= \overline{d}(s_{\lfloor n/2 \rfloor + 1} \cdots s_n, t_{k+1} \cdots t_m) \;=\; \overline{d}(s_n \cdots s_{\lfloor n/2 \rfloor + 1}, t_m \cdots t_{k+1}). \end{aligned}$$

Nach unseren Überlegungen gilt für das optimale m', dass für die optimale Edit-Distanz gilt: $\overline{d}(s,t) = V(\lfloor n/2 \rfloor, m') + V'(\lfloor n/2 \rfloor, m')$. Wir können also $\overline{d}(s,t)$ und m' wie folgt berechnen:

$$\begin{aligned} \overline{d}(s,t) &= \min\left\{V\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) + V'\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) \;:\; k \in [0:m]\right\}, \\ m' &= \operatorname{argmin}\left\{V\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) + V'\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) \;:\; k \in [0:m]\right\}. \end{aligned}$$

Hierbei bezeichnet argmin einen Index-Wert, für den in der Menge das Minimum angenommen wird, d.h. es gilt für eine Menge $M = \{e_i \;:\; i \in I\}$ mit der zugehörigen Indexmenge I:

$$\min\{e_i \;:\; i \in I\} = e_{\operatorname{argmin}\{e_i : i \in I\}}.$$

Somit können wir also für zwei Zeichenreihen s und t den Schnittpunkt m' berechnen, der zwei optimale Teil-Alignments angibt, aus dem ein optimales Alignment für s und t berechnet wird.

In Abbildung 4.20 ist der vollständige Algorithmus von Hirschberg angegeben. Wir bestimmen also zunächst den Mittelpunkt des Pfades eines optimalen Alignments $(\lfloor n/2 \rfloor, m')$, dann lösen wir rekursiv die beiden entstehenden Alignment-Probleme und konstruieren zum Schluss aus diesen beiden Alignments eine neues optimales Alignment für s und t.

Wir müssen uns nur noch überlegen, wann wir die Rekursion abbrechen und ob sich diese Teilprobleme dann trivial lösen lassen. Wir brechen die Rekursion ab, wenn die erste Zeichenreihe, d.h. das Teilwort von s, die Länge 1 erreicht.

Sei also $(s_\ell, t_p \cdots t_q)$ eine solche Eingabe, in der die Rekursion abbricht. Ist $q < p$, d.h. $t_p \cdots t_q = \varepsilon$, dann wird eine Deletion von s_ℓ als Alignment

$$\begin{pmatrix} s_\ell \\ - \end{pmatrix}$$

zurückgeliefert. Ist andernfall $p \leq q$, dann bestimmen wir das Zeichen aus $t_p \cdots t_q$ (inklusive dem Leerzeichen) das mit s_ℓ den besten Score besitzt. Dazu bestimmen

1. Berechne die Werte optimaler Alignments für $s' = s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_k$ für alle $k \in [0:m]$, d.h. $V(\lfloor n/2 \rfloor, k)$ für alle $k \in [0:m]$.

   (In Wirklichkeit $s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_m$.)

2. Berechne die Werte optimaler Alignments für $s'' = s_{\lfloor n/2 \rfloor + 1} \cdots s_n$ mit $t_{k+1} \cdots t_m$ für alle $k \in [0:m]$, d.h. $V'(\lfloor n/2 \rfloor, k)$ für alle $k \in [0:m]$.

   (In Wirklichkeit $s_n \cdots s_{\lfloor n/2 \rfloor + 1}$ mit $t_m \cdots t_1$.)

3. Bestimme m' mittels $m' = \operatorname{argmin}\left\{V(\frac{n}{2}, k) + V'(\frac{n}{2}, k) \;:\; k \in [0:m]\right\}$.

4. Löse rekursiv die beiden Alignment-Probleme für $s' = s_1 \cdots s_{\lfloor n/2 \rfloor}$ mit $t_1 \cdots t_{m'}$ sowie $s'' = s_{\lfloor n/2 \rfloor + 1} \cdots s_n$ mit $t_{m'+1} \cdots t_m$.

Abbildung 4.20: Algorithmus: Verfahren von Hirschberg

wir zuerst

$$\begin{aligned} k &= \operatorname{argmin}\left\{ w(s_\ell, t_i) + \sum_{\substack{j=p \\ j \neq i}}^{q} w(-, t_j) \;:\; i \in [p:q] \right\} \\ &= \operatorname{argmin}\left\{ w(s_\ell, t_i) + W_{p,q} - w(-, t_i) \;:\; i \in [p:q] \right\}, \end{aligned}$$

wobei $W_{p,q} := \sum_{j=p}^{q} w(-, t_j)$. Gilt

$$\begin{aligned} w(s_\ell, t_k) + W_{p,q} - w(-, t_k) &= w(s_\ell, t_k) + \sum_{\substack{j=p \\ j \neq k}}^{q} w(-, t_j) \\ &\leq w(s_\ell, -) + \sum_{j=p}^{q} w(-, t_j) \\ &= w(s_\ell, -) + W_{p,q}, \end{aligned}$$

dann liefern wir als Alignment

$$\begin{pmatrix} - & \cdots\cdots & - & s_\ell & - & \cdots\cdots & - \\ t_p & \cdots\cdots & t_{k-1} & t_k & t_{k+1} & \cdots\cdots & t_q \end{pmatrix}$$

zurück. Andernfalls wird das Alignment

$$\begin{pmatrix} s_\ell & - & \cdots\cdots & - \\ - & t_p & \cdots\cdots & t_q \end{pmatrix}$$

zurückgegeben. Dies kann natürlich auch leicht mit dem Algorithmus aus Abbildung 4.16 generiert werden, da die Matrix ja nur aus 2 Zeilen besteht ($n = 1$).

Folgendes Beispiel verdeutlicht die Vorgehensweise beim Verfahren von Hirschberg zur Bestimmung eines optimalen Sequenzen-Alignments anhand von zwei Sequenzen $s = AGGT$ und $t = ACCGT$. Zuerst wird, wie oben beschrieben, der Wert des optimalen Alignments für $s_1 \cdots s_2$ mit $t_1 \cdots t_k$ und für $s_3 \cdots s_4$ mit $t_k \cdots t_5$ für alle $k \in [0:5]$ berechnet. Dies ist in Abbildung 4.21 illustriert.

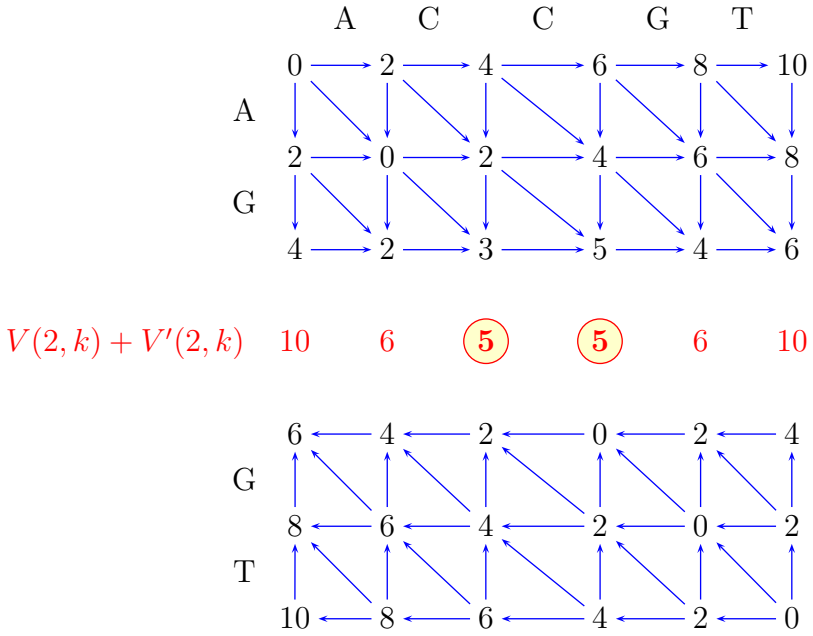


Abbildung 4.21: Beispiel: Bestimmung von m' im Hirschberg-Algorithmus

Der nächste Schritt besteht nun darin, m' zu bestimmen. In unserem Fall sind zwei verschiedene Werte möglich, da zweimal der Wert 5 auftritt. Für den weiteren Verlauf entscheiden wir uns für $m' = 3$. Jetzt müssen wir rekursiv die beiden Teile bearbeiten.

Zuerst betrachten wir den oberen linken Teil (siehe dazu auch Abbildung 4.22). Wieder haben wir zwei Schnittpunkte zur Wahl, nämlich 1 und 2. Wir entscheiden uns für 1. Damit erhalten wir jetzt Probleme, bei denen die erste Sequenz Länge 1 hat. Wir müssen jetzt also ein Alignment für A mit A und für G mit CC finden. Offensichtlich wählt man $\binom{A}{A}$ und $\binom{G-}{CC}$ (wobei das natürlich im Detail von der Kostenfunktion w abhängt). Dieses wird dann zu $\binom{AG-}{ACC}$ zusammengesetzt.

Jetzt fehlt noch der zweite rekursive Aufruf für $m' = 3$, d.h. der untere rechte Teil (siehe dazu Abbildung 4.23). Hier ist der Aufteilungspunkt eindeutig und die Zeichen stimmen ja auch überein, so dass wir zuerst zwei kurze Alignments $\binom{G}{G}$ und $\binom{T}{T}$ erhalten, die dann zu $\binom{GT}{GT}$ zusammengesetzt werden.

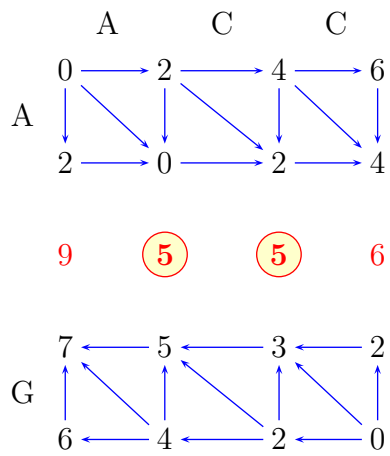


Abbildung 4.22: Beispiel: Erster rekursiver Aufruf im Hirschberg-Algorithmus

Setzt man nun die beiden Alignments aus dem ersten Aufruf, nämlich $\binom{AG-}{ACC}$, und dem zweiten rekursiven Aufruf, nämlich $\binom{GT}{GT}$, zusammen, so erhalten wir als gesamtes Alignment $\binom{AG-GT}{ACCGT}$, das auch die schon berechnete Distanz 5 besitzt.

Wir haben bereits die Korrektheit der Variante von Hirschberg bewiesen. Es bleibt noch zu zeigen, dass der Platzbedarf wirklich linear ist und wie groß die Laufzeit ist.

Zuerst zum Platzbedarf: Dazu betrachten wir noch einmal den in Abbildung 4.20 angegebenen Algorithmus. Schritte 1 und 2 können wir, wie bereits erläutert, in linearem Platz $O(m)$ berechnen. Schritt 3 benötigt keinen weiteren Platz. Im letzten Schritt rufen wir zweimal rekursiv die Prozeduren auf und benötigen für die erste Rekursion Platz $O(m')$ sowie für die zweite Rekursion Platz $O(m'')$ und somit wieder Platz von $O(m' + m'') = O(m)$. Da wir den Platz aus Schritt 1 und 2 wiederverwenden können, benötigen wir insgesamt nur Platz $O(m)$.

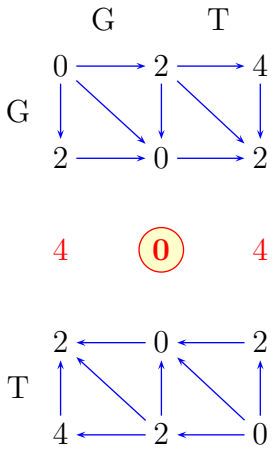


Abbildung 4.23: Beispiel: Zweiter rekursiver Aufruf im Hirschberg-Algorithmus

Betrachten wir das etwas formaler, dann gilt für den Platzbedarf $S(n,m)$ die folgende Rekursionsgleichung:

$$S(n,m) = \max\left\{3(m+1), S(\left\lfloor \frac{n}{2} \right\rfloor, m'), S(\left\lceil \frac{n}{2} \right\rceil, m-m')\right\}.$$

Hierbei kommt $3(m+1)$ aus Schritt 1 mit 3. und der Rest aus den beiden rekursiven Aufrufen. Wie gesagt, ist hierbei zu beachten, dass der Platz wiederverwendbar ist, und daher das Maximum aus den drei Werten zu bilden ist

Mit Induktion kann man zeigen, dass dann $S(n,m) \leq 3(m+1)$ ist:

$$\begin{aligned} S(n,m) &= \max\left\{3(m+1), S(\left\lfloor \frac{n}{2} \right\rfloor, m'), S(\left\lceil \frac{n}{2} \right\rceil, m-m')\right\} \\ &\leq \max\{3(m+1), 3(m'+1), 8(m''-m+1)\} \\ &\quad \text{da } \max\{m, m''\} \leq m \text{ und } 3(m+1) \leq 6m \text{ für } m \geq 1 \\ &\leq 3(m+1) \\ &\leq 6m. \end{aligned}$$

Es bleibt die Laufzeitanalyse. Wir bezeichnen hierzu mit $T(n,m)$ den Zeitbedarf für die Hirschberg-Variante für zwei Zeichenreihen mit Längen n und m. Wir stellen zuerst eine Rekursionsformel für T auf. Hierfür zählen wir die Anzahl berechneter Einträge der virtuellen Tabelle D und ähnliche solche Berechnungen.

09.07.19

Schritt 1 berechnet $(\lfloor \frac{n}{2} \rfloor + 1) \cdot (m+1)$ solche Felder, Schritt 2 $(\lceil \frac{n}{2} \rceil + 1) \cdot (m+1)$ solche Felder. Schritt 3 berechnet $(m+1)$ neue Werte, die wir hier als Berechnungen zählen. Die Schritte 1 mit 3 besuchen also maximal $(n+3)(m+1)$ Felder. Aufgrund der beiden rekursiven Aufrufe im Schritt 4, ist der Zeitbedarf hierfür durch $T(\lfloor n/2 \rfloor, k) + T(\lceil n/2 \rceil, m-k)$ gegeben, wobei $k \in [0:m]$. Somit erhalten wir folgende Rekursionsformel für den Zeitbedarf:

$$T(n,m) = (n+3)(m+1) + T\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) + T\left(\left\lceil \frac{n}{2} \right\rceil, m-k\right).$$

Wir könnten diese Rekursionsgleichung mit aufwendigen Mitteln direkt lösen. Wir machen es uns aber hier etwas leichter und verifizieren eine geratene Lösung mittels Induktion.

Beobachtung 4.24 *Es gilt* $T(n,m) \leq 3n(m+1) + 11m\log(n)$.

Beweis: Wir beweisen die Behauptung mittels vollständiger Induktion nach n.

Induktionsanfang ($n = 1$): $T(1,m)$ ist sicherlich durch $3(m+1)$ beschränkt, da wir nur ein Zeichen gegen eine Zeichenreihe der Länge m optimal ausrichten müssen, was eine Berechnung von $2(m+1)$ Feldern bedeutet.

Induktionsschritt ($\rightarrow n$): Wir setzen nun die Behauptung als Induktionsvoraussetzung in die Rekursionsformel ein (da $\lceil n/2 \rceil < n$ für $n \geq 2$) und formen zunächst für $m \geq 2$ um:

$$
\begin{aligned}
T(n,m) &= (n+3)(m+1) + T\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right) + T\left(\left\lceil \frac{n}{2} \right\rceil, m-k\right) \\
&\overset{\text{I.V.}}{\leq} (n+3)(m+1) \\
&\quad + 3\left\lfloor \frac{n}{2} \right\rfloor (k+1) + 11k \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\
&\quad + 3\left\lceil \frac{n}{2} \right\rceil (m-k+1) + 11(m-k) \log\left(\left\lceil \frac{n}{2} \right\rceil\right) \\
&\leq (n+3)m + (n+3) + 3\left\lceil \frac{n}{2} \right\rceil m + 3n + 11m \log\left(\left\lceil \frac{n}{2} \right\rceil\right) \\
&\quad \text{da } n \leq 2\lfloor n/2 \rfloor + 1,\ m \geq 2 \text{ und } \lceil \tfrac{n}{2} \rceil \leq \tfrac{2}{3}n \\
&\leq \left(2\left\lfloor \frac{n}{2} \right\rfloor + 4\right) m + 4n + 3 + 3\left\lceil \frac{n}{2} \right\rceil m + 11m \log\left(\frac{2n}{3}\right) \\
&\quad \text{mit } n \leq \left\lfloor \tfrac{n}{2} \right\rfloor m + 1 \text{ für } m \geq 2 \\
&\leq 2\left\lfloor \frac{n}{2} \right\rfloor m + 4m + 3n + \left\lfloor \frac{n}{2} \right\rfloor m + 1 + 3 + 3\left\lceil \frac{n}{2} \right\rceil m + 11m \log\left(\frac{2n}{3}\right) \\
&\leq 3nm + 4m + 3n + 4 + 11m \log(n) + 11m \log\left(\frac{2}{3}\right) \\
&\leq 3n(m+1) + 4m + 4 + 11m \log(n) + 11m \log\left(\frac{2}{3}\right) \\
&\leq 3n(m+1) + 11m \log(n) + m\left[4 + \frac{4}{m} + 11 \log\left(\frac{2}{3}\right)\right] \\
&\quad \text{da } [4 + \tfrac{4}{m} + 11 \log(2/3)] < 0 \text{ (wegen } m \geq 2) \\
&\leq 3n(m+1) + 11m \log(n).
\end{aligned}
$$

Beachte, dass wir auch den Fall $m \in [0:1]$ untersuchen müssen. Falls $m < 2$ ist ändern wir den Algorithmus so ab, dass wir das Alignment direkt ausrechnen. Für $m = 0$ gilt aber offensichtlich $T(n,0) = n \leq 3n$ für alle $n \geq 2$. Für $m = 1$ gilt weiter $T(n,1) = 2(n+1) \leq 3n + 11 \leq 3n + 11\log(n)$ für alle $n \geq 2$. Damit ist der Induktionsschluss vollzogen. ■

Damit ist die Laufzeit weiterhin $O(nm)$ und wir haben den folgenden Satz bewiesen.

Theorem 4.25 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Der Algorithmus von Hirschberg berechnet ein optimales globales paarweises Sequenzen-Alignment für s und t in Zeit $O(nm)$ mit Platzbedarf $O(\min\{n,m\})$.*

Wir haben zwar nur einen Platzbedarf von $O(m)$ gezeigt, aber es sollte klar sein, dass man auch die Sequenzen s und t vertauschen kann, so dass die kürzere der beiden Sequenzen im Wesentlichen den benötigten Platzbedarf impliziert.

4.3 Spezielle Lückenstrafen

In diesem Abschnitt wollen wir teilweise Alignments und Strafen für Lücken genauer untersuchen. Eine Lücke ist nichts anderes als eine aufeinander folgende Folge von Edit-Operationen, die entweder nur aus Deletionen oder nur aus Insertionen bestehen (jedoch nicht abwechselnd). In einem Alignment entspricht dies einem Teilwort, das nur aus Leerzeichen besteht. Solche zusammenhängenden Lücken der Länge ℓ haben ihre Ursache meist aus einer einzigen Mutation, die eine ganze Teilsequenz entfernt bzw. eingefügt hat. Aus diesem Grund ist eine Bestrafung, die proportional zur Länge der Lücke ist, nicht ganz gerecht, und sollte daher eher subadditiv in der Länge der Lücke sein.

4.3.1 Semiglobale Alignments

Im Falle *semiglobaler Alignments* wollen wir Lücken, die am Anfang oder am Ende eines Wortes auftreten, nicht berücksichtigen. Semiglobale Alignments werden oft auch als *Free-Shift Alignments* bezeichnet. Dies ist insbesondere dann von Interesse, wenn die Wörter sehr unterschiedlich lang sind oder wenn klar ist, dass diese Sequenzen zwar eine Ähnlichkeit besitzen, aber man nicht weiß, ob man die Sequenzen korrekt aus einer großen Sequenz herausgeschnitten hat. Dann können an den Enden Fehler aufgetreten sein (etwas zu kurze oder zu lange Sequenzen gewählt).

Beispiel: Betrachten wir die beiden Sequenzen $CGTACGTGATGA$ und $CGATTA$. Wenn wir hierfür die optimale Alignment-Distanz berechnen (mit $w(x,y) = 3$ für $x \neq y \in \Sigma$ und $w(x,-) = 2$ für $x \in \Sigma$), so erhalten wir das folgende optimale Alignment:

$$\begin{array}{cccccccccccc} C & G & T & A & C & G & T & G & A & G & T & G & A \\ C & G & - & A & - & - & T & - & - & - & T & - & A \end{array}$$

Dieses hat einen Alignment-Distanz von $7 * 2 = 14$.

Alternativ betrachten wir folgendes Alignment:

$$\begin{array}{cccccccccccc} C & G & T & A & C & G & - & T & G & A & G & T & G & A \\ - & - & - & - & C & G & A & T & T & A & - & - & - & - \end{array}$$

Dieses hat natürlich eine größere Alignment-Distanz von $9 * 2 + 1 * 3 = 21$.

Berücksichtigen wir jedoch die Deletionen am Anfang und Ende nicht, da diese vermutlich nur aus einer zu lang ausgewählten ersten (oder zu kurz ausgewählten zweiten) Sequenz herrühren, so erhalten wir eine Alignment-Distanz von $1*2+1*3=5$. Aus diesem Grund werden bei einem semiglobalen Alignment Folgen von Insertionen bzw. Deletionen zu Beginn oder am Ende nicht berücksichtigt.

Es gibt jedoch noch ein kleines Problem. Man kann nämlich dann immer eine Alignment mit Alignment-Distanz 0 basteln:

$$\begin{array}{cccccccccccccccccc} C & G & T & A & C & G & T & G & A & G & T & G & A & - & - & - & - & - \\ - & - & - & - & - & - & - & - & - & - & - & - & - & C & G & A & T & T \end{array}$$

Bei solchen Distanzen sollte man natürlich den Wert der Distanz im Verhältnis zur Länge des Bereiches in Beziehung setzen, in dem das eigentliche, bewertete Alignment steht. Man kann jetzt die Distanz bezüglich der wirklich ausgerichteten Zeichen um jeweils einen konstanten Betrag erniedrigen. Wir können uns das Leben jedoch viel einfacher machen, wenn wir statt dessen Ähnlichkeitsmaße verwenden. Wie wir schon gesehen haben, entsprechen diese im Wesentlichen den Distanzmaßen, sind aber bei solchen semiglobalen Alignments wesentlich einfacher zu handhaben.

Wir verwenden jetzt als Kostenfunktion für ein Ähnlichkeitsmaß für Matches $+2$, für Insertionen sowie Deletionen -1 und für Substitutionen -1. Dieses Kostenmaß ist aus der Kostenfunktion für das obige Distanzmaß mittels $2-w(x,y)$ bzw. $1-w(x,-)$ gewonnen worden. Somit erhält man für das erste, optimale globale Alignment einen Score von

$$6*(+2)+7*(-1)=+5,$$

was man mithilfe von Theorem 4.20 auch wie folgt ermitteln kann:

$$s(s,t)=\frac{C}{2}(|a|+|b|)-\overline{d}(s,t)=1(13+6)-14=+5.$$

Für das zweite Alignment erhält man als globales Alignment einen Ähnlichkeitswert von

$$4*(+2)+9*(-1)+1*(-1)=-2.$$

und als semiglobales-Alignment einen Score von

$$4*(+2)+1*(-1)+1*(-2)=+6.$$

Für das künstliche Alignment erhalten wir jedoch auch hier für einen Score von 0 für das Ähnlichkeitsmaß. Wir weisen an dieser Stelle darauf hin, dass die hier

verwendeten Kostenfunktionen nicht besonders gut gewählt, aber für die Beispiele ausreichend sind.

Semiglobales Alignment

Eingabe: Seien $s = s_1 \cdots s_n \in \Sigma^n$ und $t = t_1 \cdots t_m \in \Sigma^m$ und sei σ ein Ähnlichkeitsmaß.

Gesucht: Ein optimales Alignment zweier Teilwörter $s' = s_{i_1} \cdots s_{j_1}$ von s und $t' = t_{i_2} \cdots t_{j_2}$ von t mit einem optimalen Ähnlichkeitswert $\sigma(s', t')$, wobei $\min(i_1, i_2) = 1$ und $\min(n - j_1, m - j_2) = 0$.

Wie äußert sich jetzt die Nichtberücksichtigung von Lücken am Anfang und Ende eines Alignments in der Berechnung dieser mit Hilfe der Dynamischen Programmierung nach Needleman-Wunsch. Betrachten wir zuerst noch einmal Abbildung 4.24, in der schematisch semiglobale Alignments dargestellt sind.

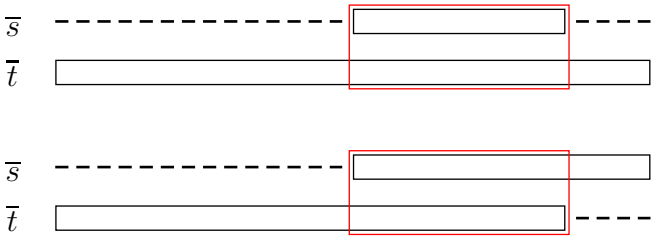


Abbildung 4.24: Skizze: semiglobale Alignments

Wenn in der ersten Sequenz s am Anfang Lücken auftreten dürfen, bedeutet dies, dass wir in der zweiten Sequenz t Einfügungen gemacht haben. Damit diese nicht zählen, dürfen diese Einfügungen zu Beginn nicht gewertet werden. Daher werden wir die erste Zeile der Tabelle mit 0 initialisieren. Analoges gilt für Lücken zu Beginn von t. Dann dürfen die Deletionen von s nicht bewertet werden und wir initialisieren auch die erste Spalte mit 0.

Nun betrachten wir Lücken am Ende. Tritt am Ende von s eine Lücke auf, dann dürfen wir die letzten Insertionen von Zeichen in t nicht berücksichtigen. Wie können wir dies bewerkstelligen? Dazu betrachten wir die letzte Zeile der Tabelle. Wenn die letzten Insertionen nicht zählen sollen, dann hört ein solches semiglobales Alignment irgendwo in der letzten Zeile auf. Wenn wir nun ein semiglobales Alignment mit maximaler Ähnlichkeit wollen, müssen wir einfach nur in der letzten Zeile den maximalen Wert suchen. Die Spalten dahinter können wir für unser semiglobales Alignment dann einfach vergessen.

Dasselbe gilt für Deletionen in s. Dann hört das semiglobale Alignment irgendwo in der letzten Spalte auf und wir bestimmen für ein optimales semiglobales Alignment den maximalen Wert in der letzten Spalte und vergessen die Zeilen danach.

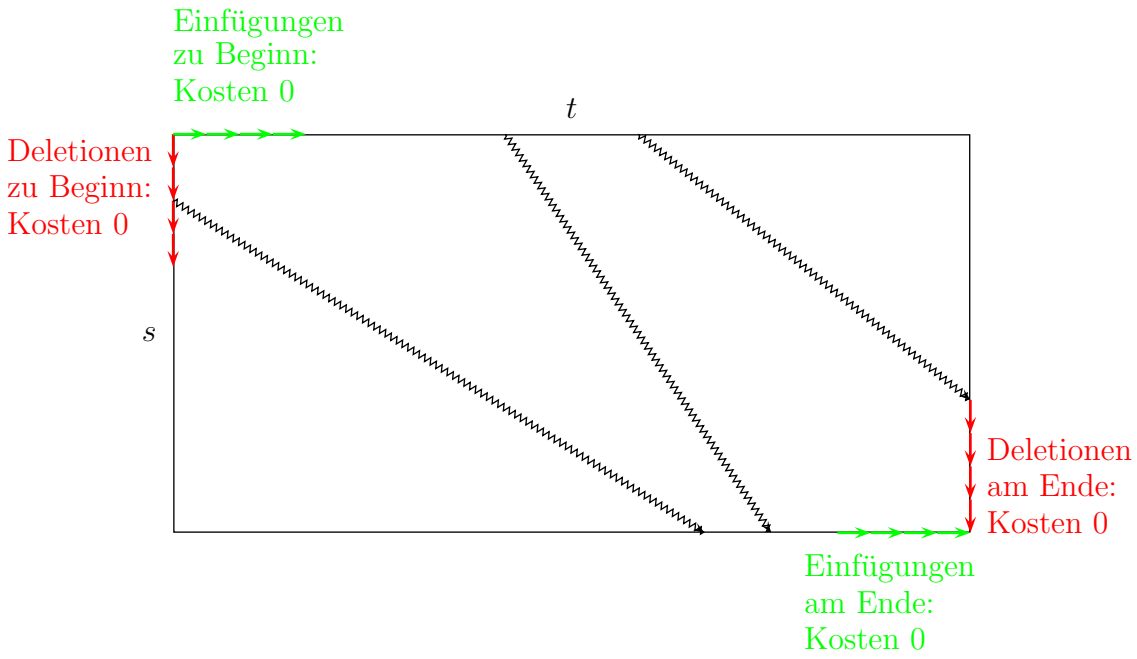


Abbildung 4.25: Skizze: Semiglobale Alignments in der Ähnlichkeitstabelle

In der folgenden Abbildung 4.25 sind die Pfade solcher semiglobaler Alignments in der berechneten Tabelle bildlich dargestellt.

Um also insgesamt ein optimales semiglobales Alignment zu erhalten, setzen wir die erste Zeile und erste Spalte gleich 0 und bestimmen den maximalen Ähnlichkeitswert, der in der letzten Zeile oder Spalte auftritt. Dieser gibt dann die Ähnlichkeit an. Das Alignment selbst erhalten wir dann genauso wie im Falle des globalen Alignments, indem wir einfach von diesem Maximalwert rückwärts das Alignment bestimmen. Wir hören auf, sobald wir die auf die erste Spalte oder die erste Zeile treffen. Damit ergibt sich für die Tabelle S (wie similarity):

$$S(i,j) = \begin{cases} 0 & \text{für } (i=0) \vee (j=0), \\ \max \left\{ \begin{array}{lcl} S(i-1,j-1) & + & w(s_i,t_j), \\ S(i-1,j) & + & w(s_i,-), \\ S(i,j-1) & + & w(-,t_j) \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

Natürlich lässt sich auch für semiglobale Alignments das Verfahren von Hirschberg zur Platzreduktion anwenden. Die Details seien dem Leser als Übungsaufgabe überlassen. Fassen wir unser Ergebnis noch zusammen.

Theorem 4.26 *Seien $s,t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales semiglobales paarweises Sequenzen-Alignment für s und t lässt sich in Zeit $O(nm)$ mit Platzbedarf $O(\min\{n,m\})$ berechnen.*

4.3.2 Lokale Alignments (Smith-Waterman)

Eine weitere Einschränkung sind so genannte *lokale Alignments*. Hier suchen wir in zwei Sequenzen zwei Teilwörter, die möglichst ähnlich zueinander sind. Damit wir nicht wieder zwei leere Teilwörter mit Alignment-Distanz 0 bekommen, verwenden wir auch hier wieder Ähnlichkeitsmaße. In der Abbildung 4.26 sind zwei solche Teilwörter, die ein solches lokales Alignment besitzen, schematisch dargestellt.

Abbildung 4.26: Skizze: Lokales Alignment

Lokales Alignment

Eingabe: Seien $s = s_1 \cdots s_n \in \Sigma^n$ und $t = t_1 \cdots t_m \in \Sigma^m$ und σ ein Ähnlichkeitsmaß.

Gesucht: Ein optimales Alignment zweier Teilwörter $s' = s_{i_1} \cdots s_{j_1}$ von s und $t' = t_{i_2} \cdots t_{j_2}$ von t mit einem maximalen Ähnlichkeitswert $\sigma(s', t')$.

Betrachten wir zunächst ein Beispiel, nämlich ein lokales Alignment zwischen den Sequenzen $s = ACGATTATTT$ und $t = TAGTAATCG$, wie es in Abbildung 4.27 dargestellt ist. Das lokales Alignment besteht aus den beiden Teilwörtern, die in dem grauen Rahmen eingefasst sind, und hat den Ähnlichkeitswert 7 (hierbei ist $w(x, x) = 3$, $w(x, y) = -3$ und $w(x, -) = -2$ für $x \neq y \in \Sigma$).

s:	A	C	G	A	T	T	A	T	T	T
t:	T	A	G	−	T	A	A	T	C	G

Abbildung 4.27: Beispiel: Ein lokales Alignment zwischen s und t

Wie können wir nun ein solches lokales Alignment berechnen? Wir werden auch hier die Methode von Needleman-Wunsch wiederverwenden. Dazu definieren wir $S(i, j)$ als den Wert eines besten (globalen) Alignments von zwei Teilwörtern von $s_{i'} \cdots s_i$ und $t_{j'} \cdots t_j$ mit $i' \leq i + 1$ und $j' \leq j + 1$. Dann können wir wieder eine

Rekursionsgleichung aufstellen:

$$S(i,j) = \begin{cases} 0 & \text{für } (i=0) \vee (j=0), \\ \max \left\{ \begin{array}{lcl} S(i-1,j-1) & + & w(s_i,t_j), \\ S(i-1,j) & + & w(s_i,-), \\ S(i,j-1) & + & w(-,t_j), \\ & 0 & \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

Mathematisch exakt müsste die Rekursionsgleichungen für $i = 0$ oder $j = 0$ wie folgt lauten:

$$S(i,j) = \begin{cases} \max \left\{ \sum_{k=\ell}^{j} w(-,t_\ell) \,:\, \ell \in [1:j+1] \right\} & \text{für } i=0, \\ \max \left\{ \sum_{k=\ell}^{i} w(s_\ell,-) \,:\, \ell \in [1:i+1] \right\} & \text{für } j=0. \end{cases}$$

Da aber für eine (biologisch) sinnvolle Kostenfunktion w das Maximum bei $\ell = j+1$ bzw. $\ell = i+1$ mit dem Wert 0 angenommen wird, können wir diese Werte auch gleich auf Null setzen.

Die Rekursionsgleichung sieht fast so aus, wie im Falle der semiglobalen Alignments. Wir müssen hier nur in der Maximumsbildung im Falle von $i \neq 0 \neq j$ den Wert 0 berücksichtigen. Das folgt daraus, dass ein lokales Alignment ja an jeder Stelle innerhalb der beiden gegebenen Sequenzen i und j beginnen kann. Wie finden wir nun ein optimales lokales Alignment? Da ein lokales Alignment ja an jeder Stelle innerhalb der Sequenzen s und t enden darf, müssen wir einfach nur den maximalen Wert innerhalb der Tabelle S finden. Dies ist dann der Ähnlichkeitswert eines optimalen lokalen Alignments.

Die Korrektheit folgt wiederum aus der Tatasche, dass ein optimales lokales Alignment, das Suffix von $s_1 \cdots s_i$ bzw. $t_1 \cdots t_j$ umfasst, entweder mit einer Substitution, einem Match, einer Insertion, einer Deletion endet oder ganz und gar leer ist und somit den Wert 0 erhält.

Das Alignment selbst finden wir dann wieder durch Rückwärtsverfolgen der Sieger aus der Maximumbildung. Ist der Sieger letztendlich der Wert 0 in der Maximumsbildung, so haben wir den Anfangspunkt eines optimalen lokalen Alignments gefunden. Die auf dieser Rekursionsgleichung basierende Methode wird oft auch als *Algorithmus von Smith und Waterman* bezeichnet.

In der folgenden Abbildung 4.28 ist noch einmal der Pfad, der zu einem lokalen Alignment gehört, innerhalb der Tabelle S schematisch dargestellt.

Auch für das lokale paarweise Sequenzen-Alignment lässt sich die Methode von Hirschberg zur Platzreduktion anwenden. Wir überlassen es wieder dem Leser sich

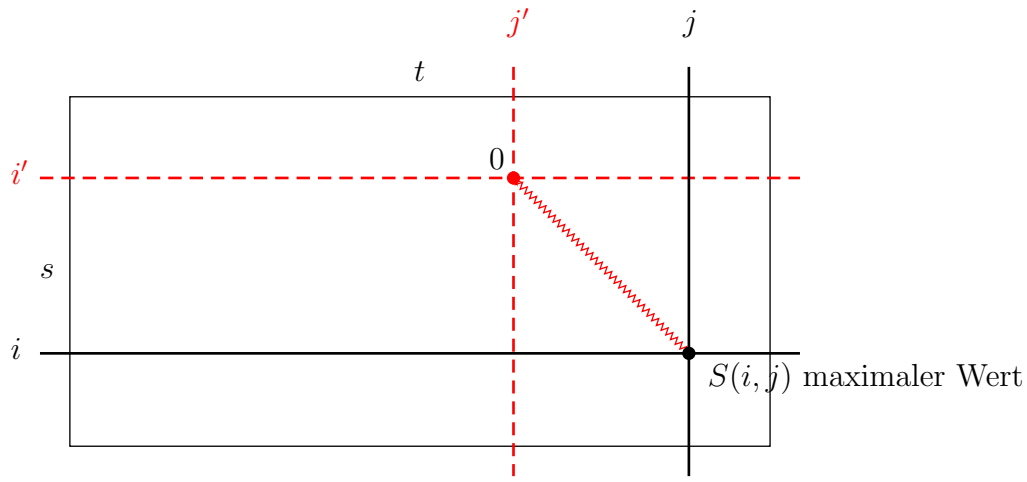


Abbildung 4.28: Skizze: lokales Alignment in der Tabelle

die genauen Details zu überlegen. Zusammenfassend erhalten wir für lokale Alignments das folgende Ergebnis.

Theorem 4.27 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales lokales paarweises Sequenzen-Alignment für s und t sowie der zugehörige Ähnlichkeitswert lässt sich in Zeit $O(nm)$ mit Platzbedarf $O(\min\{n, m\})$ berechnen.*

Kehren wir noch einmal zu unserem konkreten Beispiel vom Anfang des Abschnitts zurück und berechnen die Tabelle S für die Sequenzen $s = ACGATTATTT$ und $t = TAGTAATCG$. Die Tabelle mit den zugehörigen Werten ist in Abbildung 4.30 angegeben.

Wie man leicht sieht ist der maximale Wert 8 (siehe Position $(8, 7)$ in der Tabelle für S). Der zurückverfolgte Weg für das optimale lokale Alignment ist in der Abbildung durch die dicken Pfeile dargestellt.

Auf die Null trifft man an der Position $(0, 1)$ in der Tabelle S. Aus diesem Pfad lässt sich wie üblich wieder das zugehörige lokale Alignment ablesen. Dies ist explizit in der Abbildung 4.29 angegeben. Das zu Beginn angegebene lokale Alignment war also nicht optimal, aber schon ziemlich nahe dran. Durch eine Verlängerung kommt man auf das optimale lokale Alignment.

s:	−	A	C	G	A	T	T	A	T	T	T
t:	T	A	−	G	−	T	A	A	T	C	G

Abbildung 4.29: Beispiel: Ein optimales lokales Alignment zwischen s und t

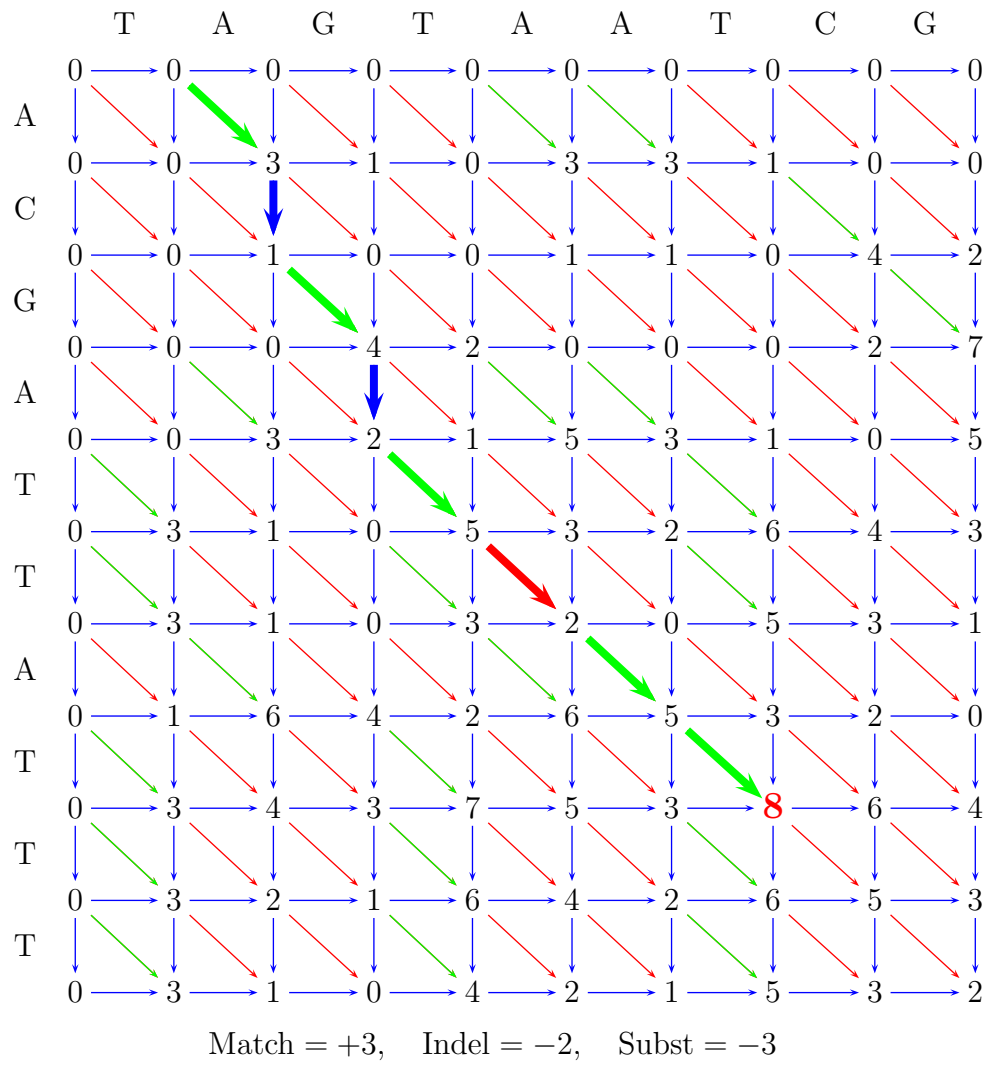


Abbildung 4.30: Beispiel:Tabelle für lokale Alignments zwischen s und t

4.3.3 Lückenstrafen

Manchmal tauchen in Alignments mittendrin immer wieder lange Lücken auf (siehe Abbildung 4.31). Eine Lücke der Länge ℓ nun mit den Kosten von ℓ Insertionen

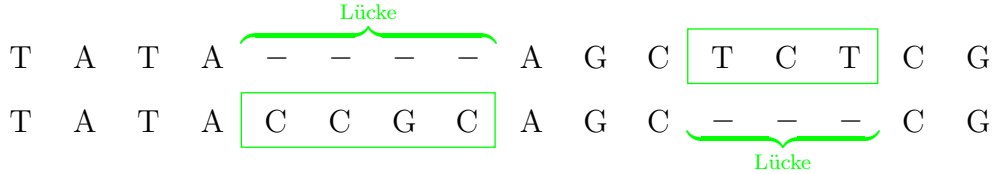


Abbildung 4.31: Skizze: Lücken in Alignments

oder Deletionen zu belasten ist nicht unbedingt fair, da diese durch eine Mutation entstanden sein kann. Dass kurze Lücken wahrscheinlicher als lange Lücken sind, ist noch einzusehen. Daher sollte die Strafe monoton in der Länge sein.

Zur Bestrafung für Lücken verwenden wir eine Lückenstrafe (engl. gap-penalty), die durch eine Funktion

$$g : \mathbb{N}_0 \to \mathbb{R}$$

gegeben ist. Hierbei gibt $g(k)$ die Strafe für k konsekutive Insertionen bzw. Deletionen an. Im Falle von Distanzmaßen ist g immer nichtnegativ und im Falle von Ähnlichkeitsmaßen nichtpositiv. Dabei sollte immer $g(0) = 0$ gelten und

$$|g| : \mathbb{N}_0 \to \mathbb{R}_+ : k \mapsto |g(k)|$$

eine monoton wachsende Funktion sein. Außerdem nehmen wir an, dass die Lückenstrafe g subadditiv ist.

Definition 4.28 *Eine Funktion* $g : \mathbb{N}_0 \to \mathbb{R}$ *heißt* subadditiv*, wenn für alle* $k', k'' \in \mathbb{N}_0$ *gilt*

$$g(k' + k'') \leq g(k') + g(k'').$$

Wir bemerken hier noch, dass wir jetzt Insertionen und Deletionen explizit immer gleich bewerten, unabhängig davon, welche Zeichen gelöscht oder eingefügt werden.

In Abbildung 4.32 ist skizziert, wie Funktionen für vernünftige Lückenstrafen aussehen. Lineare Strafen haben wir bereits berücksichtigt, da ja die betrachteten Distanz- und Ähnlichkeitsmaße linear waren. Im nächsten Abschnitt beschäftigen wir uns mit beliebigen Lückenstrafen, dann mit affinen und zum Schluss geben wir noch einen Ausblick auf konkave Lückenstrafen.

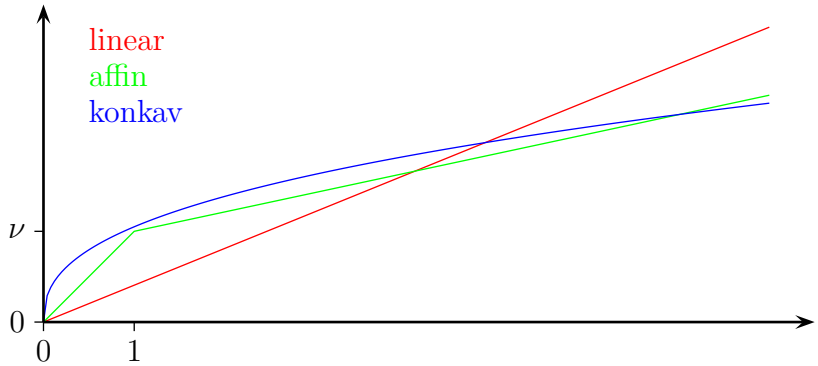


Abbildung 4.32: Skizze: Funktionsgraphen einiger typischer Straffunktionen

4.3.4 Allgemeine Lückenstrafen (Waterman-Smith-Beyer)

Nun wollen wir uns damit beschäftigen, wie wir die Rekursionsgleichungen für Alignments für allgemeine Lückenstrafen anpassen können. Wir beschränken uns hier wieder auf Distanzmaße und globale Alignments. Die Übertragung auf Ähnlichkeitsmaße und nichtglobale Alignments sei dem Leser zur Übung überlassen

Für allgemeine Lückenstrafen ergeben sich die folgenden Rekursionsgleichungen nach dem *Algorithmus von Waterman-Smith-Beyer*.

$$D(i,j) = \begin{cases} g(j) & \text{für } i = 0, \\ g(i) & \text{für } j = 0, \\ \min\limits_{k} \left\{ \begin{array}{lcl} D(i-1,j-1) & + & w(s_i,t_j), \\ D(i-k,j) & + & g(k), \\ D(i,j-k) & + & g(k) \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

Im Gegensatz zum Algorithmus von Needleman-Wunsch muss hier bei der Aktualisierung von $D(i,j)$ auf alle Werte in derselben Zeile bzw. Spalte bei Insertionen und Deletionen zurückgegriffen werden, da die Kosten der Lücken ja nicht linear sind und somit nur im ganzen und nicht einzeln berechnet werden können. Im Prinzip werden hier auch zwei unmittelbar aufeinander folgende Lücken berücksichtigt, da aber die Strafe von zwei unmittelbar aufeinander folgenden Lücken der Länge k' und k'' größer als die einer Lücke der Länge $k' + k''$ ist dies kein Problem. Wir haben hierbei ausgenutzt, dass g subadditiv ist, d.h. $g(k' + k'') \leq g(k') + g(k'')$ für alle $k', k'' \in \mathbb{N}_0$.

10.07.19

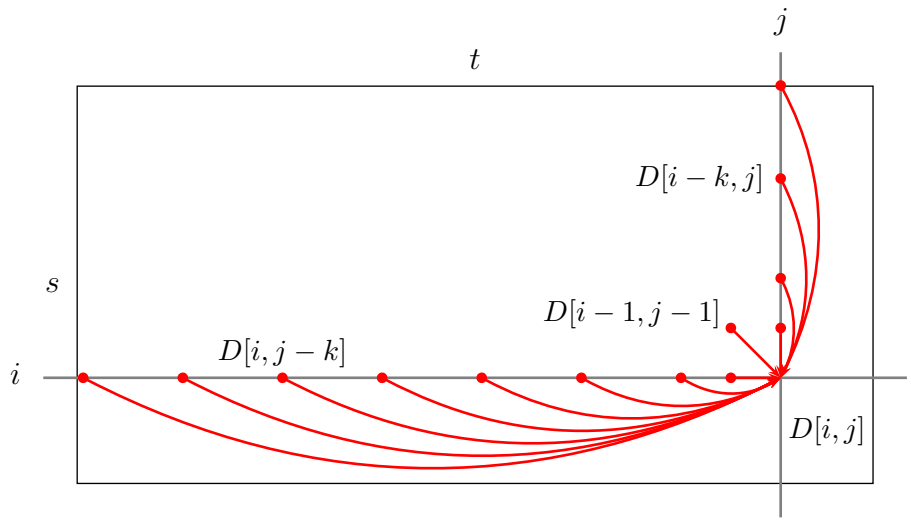


Abbildung 4.33: Skizze: Berechnung optimaler Alignments nach Waterman-Smith-Beyer

In der Abbildung 4.33 ist noch einmal schematisch dargestellt, auf welche Werte die Berechnung von $D[i,j]$ zurückgreift.

Die Laufzeit für die Variante von Waterman-Smith-Beyer ist jetzt größer geworden, da für jeden Tabellen-Eintrag eine Minimumbildung von $O(n+m)$ Elementen involviert ist. Damit wird die Laufzeit im Wesentlichen kubisch nämlich $O(nm(n+m))$. Fassen wir das Ergebnis zusammen.

Theorem 4.29 *Seien $s,t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen-Alignment für s und t mit allgemeinen Lückenstrafen lässt sich in Zeit $O(nm(n+m))$ mit Platzbedarf $O(nm)$ berechnen.*

Leider lässt sich hier die Methode von Hirschberg nicht anwenden, da zur Bestimmung optimaler Alignment-Distanzen, alle vorherigen Zeilen benötigt werden.

4.3.5 Affine Lückenstrafen (Gotoh)

Da für allgemeine Lückenstrafen sowohl Laufzeit- als auch Platzbedarf zu hoch sind, schauen wir uns spezielle Lückenstrafen an, nämlich zunächst so genannte affine Lückenstrafen. Solche affinen Lückenstrafen lassen sich wie folgt beschreiben:

$$g : \mathbb{N} \to \mathbb{R}_+ : k \mapsto \mu \cdot k + \nu$$

für Konstanten $\mu, \nu \in \mathbb{R}_+$. Für $g(0)$ setzen wir, wie zu Beginn gefordert, $g(0) = 0$, so dass nur die Funktion auf $\mathbb{N}$ im bekannten Sinne affin ist. Dennoch werden wir solche Funktionen für eine Lückenstrafe affin nennen. Hierbei sind ν die Kosten, die für das Auftauchen einer Lücke prinzipiell berechnet werden (so genannte *Strafe für Lückeneröffnung*), und μ die proportionalen Kosten für die Länge der Lücke (so genannte *Strafe für Lückenfortsetzung*). In der Literatur wird für die Lückenstrafe oft auch die Funktion $g(0) = 0$ und $g(k) = \mu(k-1) + \nu$ für alle $k \in \mathbb{N}$ verwendet.

Wieder können wir eine Rekursionsgleichung zur Berechnung optimaler Alignment-Distanzen angeben. Der daraus resultierende Algorithmus wird der *Algorithmus von Gotoh* genannt. Die Rekursionsgleichungen sind etwas komplizierter, insbesondere deswegen, da wir jetzt vier Tabellen berechnen müssen, die wie folgt definiert sind:

- $E[i,j]$ = Distanz eines optimalen Alignments von $s_1 \cdots s_i$ mit $t_1 \cdots t_j$, das mit einer Einfügung endet.
- $F[i,j]$ = Distanz eines optimalen Alignments von $s_1 \cdots s_i$ mit $t_1 \cdots t_j$, das mit einer Löschung endet.

- $G[i,j]$ = Distanz eines optimalen Alignments von $s_1 \cdots s_i$ mit $t_1 \cdots t_j$, das mit einer Substitution oder einem Match endet.
- $D[i,j]$ = Distanz eines optimalen Alignments von $s_1 \cdots s_i$ mit $t_1 \cdots t_j$.

Letztendlich ist man natürlich nur an der Tabelle D interessiert, zu deren Berechnung jedoch die anderen Tabellen benötigt werden. Die Rekursionsgleichungen ergeben sich wie folgt:

- Betrachten wir zuerst die Tabelle E, d.h. das Alignment endet mit einer Insertion. Dann muss davor eine Substitution, eine Insertion oder eine Deletion gewesen sein. Im ersten und dritten Fall wird eine Lücke eröffnet (Kosten $\nu + \mu$), im zweiten Fall eine fortgesetzt (Kosten μ). Somit erhalten wir:

$$E[i,j] = \min \left\{ \begin{array}{l} G[i,j-1] + \mu + \nu, \\ E[i,j-1] + \mu, \\ F[i,j-1] + \mu + \nu \end{array} \right\}.$$

  (Der Term $F[i,j-1] + \mu + \nu$, dritter Fall, kann weggelassen werden werden, wenn eine Substitution (a,b) billiger als eine Deletion von a und einer Insertion von b ist, das hängt allerdings auch davon ab, inwieweit Lückeneröffnungs- und Lückenfortsetzungsstrafen dabei im speziellen Fall involviert sind; Allerdings ist dann nur D, aber nicht unbedingt E korrekt.)

  Dies ist in der folgenden Abbildung 4.34 noch einmal schematisch dargestellt.

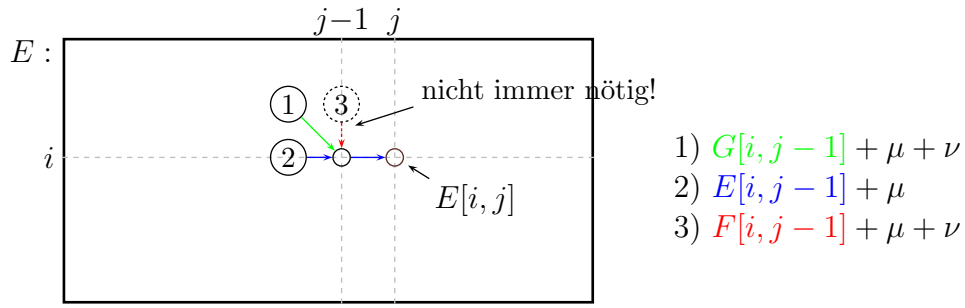


Abbildung 4.34: Skizze: Erweiterung eines Alignments mit einer Insertion

- Betrachten wir jetzt die Tabelle F, d.h. das Alignment endet mit einer Deletion. Dann muss davor eine Substitution, eine Deletion oder eine Einfügung gewesen sein. Im ersten und dritten Fall wird eine Lücke eröffnet (Kosten $\nu + \mu$), im zweiten Fall eine fortgesetzt (Kosten μ). Somit erhalten wir:

$$F[i,j] = \min \left\{ \begin{array}{l} G[i-1,j] + \mu + \nu, \\ F[i-1,j] + \mu, \\ E[i-1,j] + \mu + \nu \end{array} \right\}.$$

(Der Term $E[i-1,j]+\mu+\nu$, dritter Fall, kann weggelassen werden, wenn eine Substitution (a,b) billiger als eine Deletion von a und einer Insertion von b ist, das hängt allerdings auch davon ab, inwieweit Lückeneröffnungs- und Lückenfortsetzungsstrafen dabei im speziellen Fall involviert sind; Allerdings ist dann nur D, aber nicht unbedingt F korrekt.)

Dies ist in der folgenden Abbildung 4.35 noch einmal schematisch dargestellt.

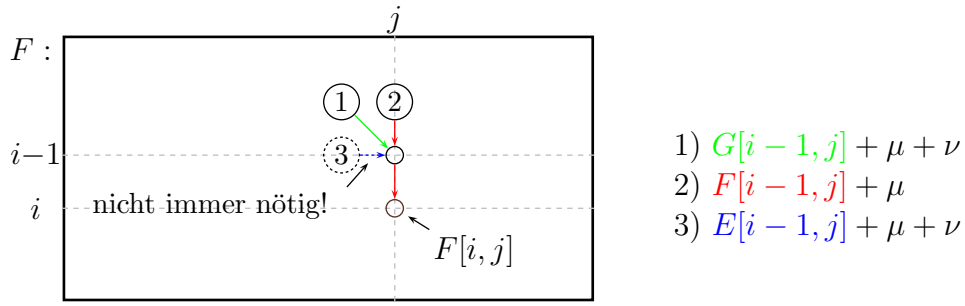


Abbildung 4.35: Skizze: Erweiterung eines Alignments mit einer Deletion

- Betrachten wir jetzt die Tabelle G, d.h. das Alignment endet mit einer Substitution. Wir müssen nur berücksichtigen, ob das Alignment zuvor mit einer Substitution, Deletion oder Insertion geendet hat. Dann erhalten wir:

$$G[i,j] = \min \left\{ \begin{array}{l} G[i-1,j-1] + w(s_i,t_j), \\ E[i-1,j-1] + w(s_i,t_j), \\ F[i-1,j-1] + w(s_i,t_j) \end{array} \right\} = D[i-1,j-1] + w(s_i,t_j).$$

Dies ist in der folgenden Abbildung 4.36 noch einmal schematisch dargestellt.

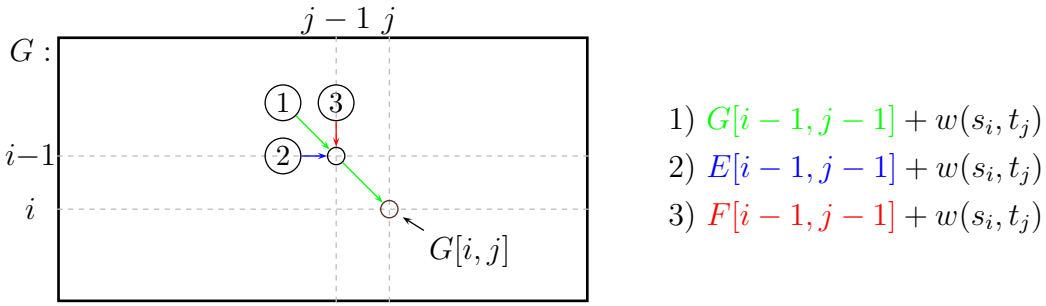


Abbildung 4.36: Skizze: Erweiterung eines Alignments mit einer Substitution

- Die Tabelle D berechnet sich offensichtlich aus dem Minimum aller drei Tabellen

$$D[i,j] = \min\{E[i,j], F[i,j], G[i,j]\}.$$

Bei Ähnlichkeitsmaßen sehen die Rekursionsgleichungen im Wesentlichen gleich aus, es wird nur die Minimumsbildung durch eine Maximumsbildung ersetzt und im Falle der Tabellen E und F müssen alle Deletionen und Insertionen berücksichtigt werden, da bei Ähnlichkeitsmaßen aufgrund der fehlenden Dreiecksungleichung auch Insertionen und Deletionen unmittelbar benachbart sein dürfen.

Es stellt sich nun noch die Frage, welche Werte jeweils in der 1. Zeile bzw. in der 1. Spalte der Matrizen stehen. Es gilt für $i > 0$ und $j > 0$:

$$\begin{aligned} E[0,j] &= j * \mu + \nu, \\ E[i,0] &= \infty, \\ E[0,0] &= \infty, \end{aligned}$$

$$\begin{aligned} F[i,0] &= i * \mu + \nu, \\ F[0,j] &= \infty, \\ F[0,0] &= \infty, \end{aligned}$$

$$\begin{aligned} G[i,0] &= \infty, \\ G[0,j] &= \infty, \\ G[0,0] &= 0. \end{aligned}$$

Eigentlich müsste auch $G[0,0] = \infty$ sein, da es kein Alignment leerer Sequenzen gibt, die mit einer Substitution bzw. Match enden. Dann würde jedoch $G[1,1]$ falsch berechnet werden. Also interpretieren wir das Alignment von ε mit sich selbst als Match. Das kann man so umgehen, wenn man fordert, dass die letzte Spalte eines Alignments eine Substitution oder ein Match enthält, ein leeres Alignment erfüllt dies, da es keine letzte Spalte gibt. $E[0,0]$ bzw. $F[0,0]$ darf jedoch nicht mit 0 initialisiert werden, da sonst die Eröffnung einer ersten Lücke von Deletionen bzw. Insertionen straffrei bleiben würde.

Auch hier kann man wieder die Methode von Hirschberg zur Platzreduktion anwenden. Hierbei muss man allerdings bei der Berechnung der optimalen Alignment-Distanz zur Bestimmung des Schnittpunkts der zweiten Sequenz aufpassen, dass für einen Lücke an der Nahtstelle des Alignments die Lückeneröffnungsstrafe nicht zweimal bezahlt wird. Halten wir noch das Ergebnis fest.

Theorem 4.30 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen-Alignment für s und t mit affinen Lückenstrafen lässt sich in Zeit $O(nm)$ mit Platzbedarf $O(\min(n,m))$ berechnen.*

4.3.6 Konkave Lückenstrafen

Zum Abschuss wollen wir noch auf konkave Lückenstrafen eingehen.

Definition 4.31 *Eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}$ heißt* konkav*, wenn gilt:*

$$\forall n \in \mathbb{N} : \; f(n) - f(n-1) \geq f(n+1) - f(n).$$

Anschaulich bedeutet dies, dass die Funktion immer langsamer wächst. Mithilfe der Diskreten Differentiation lässt sich die Bedingung als $(\Delta f)(n-1) \geq (\Delta f)(n)$ schreiben. In der kontinuierlichen Analysis ist dies gleichbedeutend damit, dass die erste Ableitung monoton fallend ist bzw. die zweite Ableitung kleiner gleich Null ist (natürlich nur, sofern die Funktion zweimal differenzierbar ist). Ein bekannter Vertreter von konkaven Funktionen ist der Logarithmus.

Lemma 4.32 *Sei $g : \mathbb{N} \to \mathbb{R}$ eine konkave Funktion. Für alle $q \leq q' \in \mathbb{N}$ und $d \in \mathbb{N}$ gilt*

$$g(q+d) - g(q) \geq g(q'+d) - g(q').$$

Beweis: Offensichtlich gilt durch wiederholtes Anwenden der Definition einer konkaven Funktion für alle $k' \geq k \in \mathbb{N}$:

$$g(k) - g(k-1) \geq g(k') - g(k'-1).$$

Addieren wir die linke Seite für alle $k \in [q+1 : q+d]$ und die rechte Seite für alle $k' \in [q'+1 : q'+d]$, erhalten wir für alle $q \leq q'$:

$$\begin{aligned} g(q+d) - g(q) &= \sum_{k=q+1}^{q+d} g(k) - g(k-1) \\ &\geq \sum_{k'=q'+1}^{q'+d} g(k') - g(k'-1) \\ &= g(q'+d) - g(q'). \end{aligned}$$

Damit die Behauptung gezeigt. ∎

Wiederholen wir noch einmal kurz in leicht modifizierter Fassung die Rekursionsgleichungen für eine beliebige Lückenstrafe g (basierend auf der Variante für affine

Lückenstrafen) mit $i, j \in \mathbb{N}$:

$$\begin{aligned}
D[i,j] &:= \min\{E[i,j], F[i,j], G[i,j]\},\\
G[i,j] &:= D[i-1,j-1] + w(s_i, t_j),\\
E[i,j] &:= \min\left\{D[i,k] + g(j-k) \;:\; k \in [0:j-1]\right\},\\
F[i,j] &:= \min\left\{D[k,j] + g(i-k) \;:\; k \in [0:i-1]\right\},\\
D[0,0] &:= 0,\\
D[i,0] &:= g(i),\\
D[0,j] &:= g(j),\\
E[0,j] &:= g(j),\\
F[i,0] &:= g(i).
\end{aligned}$$

Alle anderen hier nicht definierten Werte sind auf ∞ zu setzen.

Im Folgenden betrachten wir die vereinfachte Notation für eine feste Zeile i der Tabelle der dynamischen Programmierung:

$$\begin{aligned}
D(j) &:= D[i,j],\\
E(j) &:= \min\left\{D(k) + g(j-k) \;:\; k \in [0:j-1]\right\}.
\end{aligned}$$

Wir führen nun folgende Abkürzung ein:

$$\mathrm{Cand}(k,j) := D(k) + g(j-k).$$

Damit gilt dann:

$$E(j) = \min\left\{\mathrm{Cand}(k,j) \;:\; k \in [0:j-1]\right\}.$$

Die Berechnung von E kann mit Hilfe des in Abbildung 4.37 angegebenen Algorithmus erfolgen. Dabei wird bei der Bestimmung des Wertes von $E(j)$ darauf verzichtet, rückwärts alle Alignments mit einem Einfüge-Block zu betrachten. Stattdessen wird jedes Mal, wenn der Wert von $D(j)$ berechnet wird, der bislang optimale Wert von $E(j')$ berechnet (in $\hat{E}(\cdot)$), der durch eine Einfüge-Lücke von $j+1$ bis j' entsteht. In diesem Algorithmus wird nicht nur der bislang beste Score $\hat{E}(j')$ gespeichert, sondern auch die linkeste Position, bei der für einen optimalen Score die Lücke beginnt (in $b(\cdot)$).

Notation 4.33 *Für* $j' \in [1:m]$ *sei* $b(j') = \min\{k \in [0:j'-1] : \mathit{Cand}(k,j') = \hat{E}(j')\}$.

Beachte, dass die Werte $b(j')$ vom aktuellen betrachteten Wert j des in Abbildung 4.37 angegebenen Algorithmus abhängen.

Forward_Propagation

```
begin
    for (j := 1; j ≤ m; j++) do
        Ê(j) := Cand(0, j);
        b(j) := 0;

    for (j := 1; j ≤ m; j++) do
        E(j) := Ê(j);
        D(j) := min{E(j), F(j), G(j)};
        for (j' := j + 1; j' ≤ m; j'++) do
            if (Ê(j') > Cand(j, j')) then
                Ê(j') := Cand(j, j');
                b(j') = j;
end
```

Abbildung 4.37: Algorithmus: Vorwärts-Berechnung der Tabelle E

Dieses Vorgehen bringt für eine Zeile noch keine Zeiteinsparung, das folgende Lemma zeigt jedoch, dass für ein j nicht unbedingt alle $j' > j$ betrachtet werden müssen.

05.07.16

Lemma 4.34 *Sei $k < j < j' \leq j'' \in [1 : m]$. Wenn $Cand(k, j') \leq Cand(j, j')$ gilt, dann ist $Cand(k, j'') \leq Cand(j, j'')$.*

In Abbildung 4.38 ist die Aussage des Lemmas noch einmal illustriert. Im Wesentlichen folgt die Aussage aus der Tatsache, dass die beiden Terme $\text{Cand}(k, x)$ und $\text{Cand}(j, x)$ interpretiert als Funktionen in x zwei verschobene Instanzen von g sind und sich daher nur einmal schneiden können.

Beweis: Sei $q := j' - j \geq 0$, $q' := j'' - j \geq j' - j = q$ und $d := j - k > 0$. Es gilt wegen der Konkavität von g und mit Lemma 4.32:

$$g(q + d) - g(q) \geq g(q' + d) - g(q').$$

Dann gilt auch

$$g(j' - k) - g(j' - j) \geq g(j'' - k) - g(j'' - j),$$

und damit auch (nach Multiplikation mit -1):

$$g(j' - j) - g(j' - k) \leq g(j'' - j) - g(j'' - k). \tag{4.1}$$

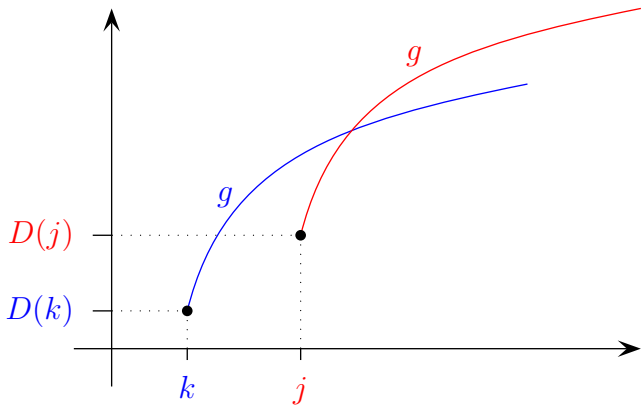


Abbildung 4.38: Skizze: Illustration von Lemma 4.34

Weiterhin gilt wegen der Voraussetzung $\text{Cand}(k, j') \leq \text{Cand}(j, j')$:

$$D(k) + g(j' - k) \leq D(j) + g(j' - j). \tag{4.2}$$

Somit gilt:

$$\begin{aligned}
\text{Cand}(k, j'') &= D(k) + g(j'' - k) \\
&\quad \text{wegen Ungleichung 4.2} \\
&\leq D(j) + g(j' - j) - g(j' - k) + g(j'' - k) \\
&\quad \text{wegen Ungleichung 4.1} \\
&\leq D(j) + g(j'' - j) - g(j'' - k) + g(j'' - k) \\
&= D(j) + g(j'' - j) \\
&= \text{Cand}(j, j'').
\end{aligned}$$

Damit ist das Lemma bewiesen. ■

Aus diesem Lemma folgt sofort das folgende Korollar, das die Abbruch-Bedingung in der for-Schleife im Algorithmus in Abbildung 4.37 einschränkt.

Korollar 4.35 *Wenn $\text{Cand}(j, j') \geq \hat{E}(j')$ für ein $j' > j$ gilt, dann gilt für alle $j'' \geq j'$ die Beziehung $\text{Cand}(j, j'') \geq \hat{E}(j'')$.*

Das hilft jedoch für eine wesentliche Zeitersparnis immer noch nicht. Das folgende Lemma gibt jedoch weiteren Aufschluss, an welchen Positionen in einer Zeile die besten Einfüge-Lücken beginnen.

Lemma 4.36 *Sei $j \in [0 : m]$ und betrachte den Zeitpunkt, wenn $D(j)$ aktualisiert wurde, aber $\hat{E}(\cdot)$ noch nicht. Dann ist $b(j') \geq b(j' + 1)$ für alle $j' \in [j + 1 : m]$.*

Beweis: Es gilt nach Definition von $b(j')$:

$$\text{Cand}(b(j'), j') \leq \text{Cand}(b(j'+1), j'). \tag{4.3}$$

Für einen Widerspruchsbeweis sei $b(j') < b(j'+1)$. Dann gilt nach dem vorigen Lemma 4.34, da die Voraussetzungen $b(j') < b(j'+1) < j' < j'+1$ erfüllt sind (wobei nach dem Algorithmus alle Einträge von b dann kleiner als j sind und somit $b(j'+1) < j < j'$ gilt) sowie mit Ungleichung 4.3:

$$\text{Cand}(b(j'), j'+1) \leq \text{Cand}(b(j'+1), j'+1).$$

Dann wäre jedoch $b(j'+1) \leq b(j')$, da immer der linkest mögliche Index für den bislang optimalen Score in $b(\cdot)$ gesetzt wird, und wir erhalten den gewünschten Widerspruch zu $b(j') < b(j'+1)$. ■

Somit ist also die Folge b eine monoton fallende Folge. Dies ist in Abbildung 4.39 noch einmal schematisch dargestellt.

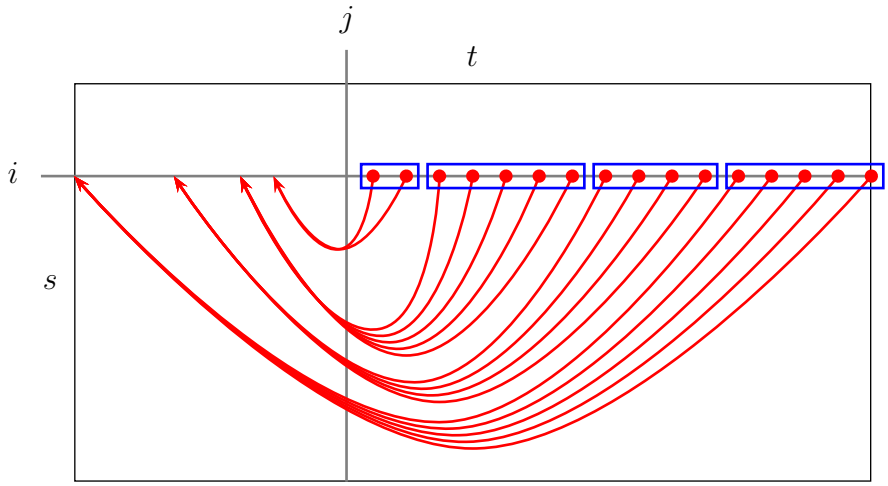


Abbildung 4.39: Skizze: Monotonie der $b(\cdot)$-Werte

Korollar 4.37 *Zum Ende jeder Iteration von j ist die Folge $(b(j+1), \ldots, b(m))$ monoton fallend.*

Für das Folgende betrachten wir nun die Partition von $[j+1 : m]$ die durch die Werte von b induziert wird. Diese Partition ist in Abbildung 4.39 durch die blauen Boxen dargestellt.

Definition 4.38 *Die Partition von $[j+1 : m]$ in disjunkte Intervalle mit gleichen $b(\cdot)$-Wert wird* Block-Partition *genannt.*

Betrachten wir zunächst den Fall $j = 1$ und die zugehörige triviale Blockpartition $[2 : m]$ von $[2 : m]$, da $b(j) = 0$ für alle $j \in [2 : m]$ ist. Nach dem Setzen von $E(1)$ mittels $\hat{E}(1)$ kann sich die Block-Partition ändern. Dabei können sich nur einige Werte von 0 auf 1 ändern. Nach dem vorherigen Lemma müssen diese einen Block aus den ersten Elementen der monoton fallenden Folge b bilden.

Fall 1: Es bleibt $b(2) = \cdots = b(m) = 0$ unverändert. Nach Korollar 4.35 ist dies genau dann der Fall, wenn $\text{Cand}(1, 2) \geq \hat{E}(2) = \text{Cand}(0, 2)$ ist.

Fall 2: Gelte nun $b(2) = \cdots = b(k) = 1$ und $b(k+1) = \cdots = b(m) = 0$. Dies ist genau dann der Fall, wenn $\text{Cand}(1, j') < \hat{E}(j')$ für alle $j' \in [2 : k]$ und $\text{Cand}(1, j') \geq \hat{E}(j')$ für alle $j' \in [k + 1 : m]$.

Fall 3: Gelte abschließend $b(2) = \cdots = b(m) = 1$. Dies kann nur genau dann der Fall sein, wenn $\text{Cand}(1, j') < \hat{E}(j')$ für alle $j' \in [2 : m]$.

Diese drei Fälle (bzw. die Ermittlung von k im Fall 2) lassen sich mit Hilfe einer binären Suche auf dem Intervall $[2 : m]$ mit der Funktion $\text{Cand}(1, j') - \hat{E}(j')$ bewerkstelligen. Ist $\text{Cand}(1, j') - \hat{E}(j') \geq 0$, sucht man in der linken Hälfte weiter, andernfalls in der rechten. Dabei ist zu beachten, dass $\text{Cand}(1, j') - \hat{E}(j')$ äquivalent zu folgendem Ausdruck ist:

$$\underbrace{D(1) + g(j' - 1)}_{\text{Cand}(1,j')} - (\underbrace{D(0) + g(j')}_{\hat{E}(j')=\text{Cand}(0,j')}).$$

Betrachten wir nun den Fall $j \geq 2$. Die Blockstruktur von $(b(j + 1), \ldots, b(m))$ sei durch eine Liste von Indizes $(p_1, \ldots, p_r)$ gegeben, d.h.

$$b(j+1) = \cdots = b(p_1) > b(p_1+1) = \cdots = b(p_{r-1}) > b(p_{r-1}+1) = \cdots = b(p_r) = b(m).$$

Sei im Folgenden $b_i = b(p_i)$. Siehe dazu auch die Skizze in Abbildung 4.40. Nachdem $E(j)$ und $D(j)$ aktualisiert wurden, erfolgt die Forward-Propagation.

Fall 1: Sei $\text{Cand}(k, j+1) = \hat{E}(j+1) \leq \text{Cand}(j, j+1)$ für ein k, das $\hat{E}(j+1)$ bisher definiert, also $k = b(j + 1) = b_x$ für ein geeignetes $x \in [1 : r]$. Nach Lemma 4.34

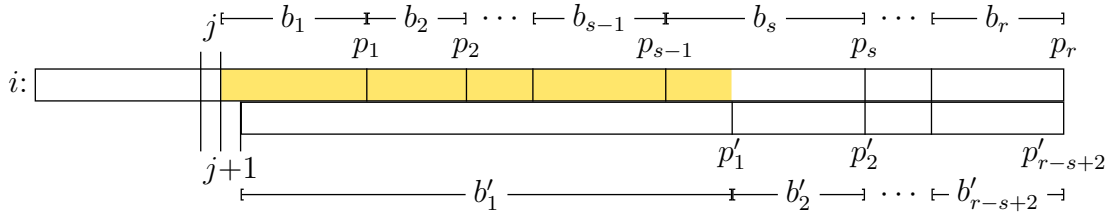


Abbildung 4.40: Skizze: Veränderung der Blockstruktur

gilt dies genau dann, wenn $\text{Cand}(k, j') = \hat{E}(j') \leq \text{Cand}(j, j')$ für alle $j' > j$. Die Blockpartition bleibt in diesem Fall unverändert.

Fall 2: Sei nun $\hat{E}(j+1) > \text{Cand}(j, j+1)$. Nun vergleichen wir nacheinander $\hat{E}(p_i)$ mit $\text{Cand}(j, p_i)$ bis die Block-Partition abgearbeitet wurde oder $\hat{E}(p_s) \leq \text{Cand}(j, p_s)$ gilt.

Im ersten Fall wird $[j+1 : m]$ ein Block mit $p_1 = m$ und $b_1 = j$. Im zweiten Fall werden die Blöcke $[1 : p_1]$, bis $[p_{s-2} + 1 : p_{s-1}]$ einschließlich zu einem Block verschmolzen. Eventuell gehört noch ein Anfangsstück des Intervalls $[p_{s-1} + 1 : p_s]$ zu diesem Block, der Rest des Blocks überlebt als eigener Block in der neuen Block-Partition. Dieses Anfangsstück ermitteln wir wie im Falle für $j = 1$ mit einer binären Suche (in diesem Block sind b-Werte wieder alle gleich).

Wir halten fest, dass $\hat{E}(j) = \text{Cand}(b(j), j)$ gilt. Weiterhin wissen wir, dass im Block $[p_{k-1} + 1 : p_k]$ die Beziehung $b(j) = b_k$ für alle $j \in [p_{k-1} + 1 : p_k]$ gilt. Also gilt:

$$\hat{E}(j) = \text{Cand}(b(j), j) = \text{Cand}(b_k, j) = D(b_k) + g(j - b_k),$$

wenn sich j im k-ten Block befindet.

Wir brauchen daher $\hat{E}$ nicht explizit speichern, sondern können die Werte mit Hilfe der Liste der b-Werte zur Liste der Blockgrenzen p_i der aktuellen Block-Partition in konstanter Zeit berechnen. Auch die b-Werte speichern wir nicht für jedes i, sondern nur einmal für jeden Block der Block-Partition als Liste. Die Aktualisierung dieser Listen ist jeweils sehr einfach, wie wir gesehen haben, da immer nur jeweils eine Präfix der Liste durch einen neuen Wert zu ersetzen ist. Aus diesen Gründen wird die binäre Suche auch nur in einem Block mit festen b_k-Wert ausgeführt, da ansonsten der direkte Zugriff bei der binären Suche auf Listenelemente schwierig wäre.

Die Laufzeit der binären Suchen in jeder Phase für $j \in [1 : m]$ benötigt Zeit $O(\log(m))$. Somit benötigen alle binären Suchen einer Zeile der Tabelle der dynamischen Programmierung Zeit $O(m \log(m))$.

Wir müssen uns nur noch überlegen, wie viele verschiedene Blöcke der Blockpartition innerhalb einer Zeile geprüft werden. Werden in einer Block-Partition maximal 2 Blöcke inspiziert, so werden maximal 2 Tests ausgeführt und die Blockpartition verlängert sich anschließend um höchstens 1. Werden $\ell > 2$ Blöcke einer Block-Partition inspiziert, so werden anschließend aus diesen ℓ Blöcken maximal 2. Damit verringert sich die Anzahl der Blöcke einer Block-Partition um mindestens $\ell - 2$.

Für jeden Durchgang $j \in [1 : m]$ berechnen wir zuerst zwei Block-Inspektionen direkt ab, das macht eine Aufwand von $O(m)$. Wenn es mehr als zwei Inspektionen gab, so sind diese proportional zur Anzahl der eliminierten Blocks. Da wir mit einem Block beginnen und für jedes $j \in [1 : m]$ maximal einen neuen Block generieren können

und auch nur so viele Blöcke eliminiert werden können, wie erzeugt werden, können maximal $O(m)$ Blöcke eliminiert werden. Somit können maximal $O(m)$ Tests auf Blöcken ausgeführt werden.

Damit sind für die Berechnung der E-Werte maximal $O(mn \log(m))$ Operationen nötig. Dieselbe Argumentation gilt auch für die F-Werte. Hierbei ist nur zu beachten, dass die F-Werte zwar pro Spalte gelten, aber ebenfalls zeilenweise berechnet werden. Damit sind für jede Spalte die Listen $(p_i)_{i\in[1:r]}$ der aktuellen Block-Partition und die Liste $(b_i)_{i\in[1:r]}$ der zugehörigen b-Werte zu speichern. Dennoch werden pro Spalte maximal $n \log(n)$ Operationen ausgeführt, also insgesamt $O(mn \log(n))$. Wir halten jetzt noch das Ergebnis fest.

Theorem 4.39 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen-Alignment für s und t mit konkaven Lückenstrafen lässt sich in Zeit $O(nm \log(n + m))$ berechnen.*

Beweis: Wir haben bereits gezeigt, dass die Laufzeit $O(mn(\log(m) + \log(n)))$ beträgt. Mit $O(mn(\log(m) + \log(n))) \subseteq O(mn \log(n + m))$ folgt die Behauptung. ■

17.07.19

Wir merken noch an, dass gewisse konkave Funktionen die Berechnung sogar in Zeit $O(nm)$ zulassen.

Man kann den Algorithmus natürlich auch leicht auf Ähnlichkeitsmaße übertragen. Eine platzsparende Variante nach Hirschberg ist aufgrund der vielen und weiten Forward-Propagation nicht möglich.

Leider wird in der Bioinformatik-Literatur der Begriff konkav und konvex oft verwechselt. Eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}$ heißt *konvex*, wenn gilt:

$$\forall n \in \mathbb{N} : \; f(n) - f(n-1) \leq f(n+1) - f(n).$$

Anschaulich bedeutet dies, dass die Funktion immer stärker wächst. In der kontinuierlichen Analysis ist dies gleichbedeutend damit, dass die erste Ableitung monoton steigend ist bzw. die zweite Ableitung größer gleich Null ist (natürlich nur, sofern die Funktion zweimal differenzierbar ist). Ein bekannter Vertreter von konvexen Funktionen ist die Exponentialfunktion oder $x \mapsto x^2$.

Des Weiteren gibt es auch für konvexe Lückenstrafen effiziente Algorithmen für das paarweise Alignment, die sogar in Zeit $O(nm)$ laufen. Konvexe Lückenstrafen (in unserem Sinne) sind jedoch für die biologische Anwendungen eher uninteressant.

4.4 Hybride Verfahren

In diesem Abschnitt wollen wir uns mit so genannten hybriden Verfahren beschäftigen. Dies sind Verfahren, die mehrere verschiedene Techniken gleichzeitig einsetzen. Hier wird es eine Alignment-Berechnung mit Hilfe von Suffix-Bäumen sein.

4.4.1 One-Against-All-Problem

Im *One-Against-All-Problem* wollen wir für zwei gegebene Sequenzen $s, t \in \Sigma^*$ alle globalen Alignments von s gegen alle Teilwörter von t berechnen. Formal wird das Problem wie folgt beschrieben.

ONE-AGAINST-ALL-PROBLEM
Eingabe: Sei $s \in \Sigma^n$, $t \in \Sigma^m$ und $\vartheta \in \mathbb{R}_+$.
Gesucht: Berechne $d(s, t')$ für alle $t' \sqsubseteq t$ mit $d(s, t') \leq \vartheta$.

Hierbei gilt $t' \sqsubseteq t$ für ein gegebenes $t \in \Sigma^*$, wenn t' ein Teilwort von t ist.

Wir betrachten zuerst einen naiven Ansatz und berechnen für jedes Teilwort t' von t dessen Alignment gegen s. Die Anzahl der Teilwörter von t mit $|t| = m$ beträgt $\Theta(m^2)$. Da der Aufwand pro Alignment $O(nm)$ beträgt, ist der Gesamtaufwand $O(nm^3)$.

Etwas geschickter können wir vorgehen, wenn wir uns an die Tabelle $D(i, j)$ erinnern und bemerken, dass wir ja nicht nur die optimale Alignment-Distanz $s = s_1 \cdots s_n$ mit $t = t_1 \cdots t_m$ berechnen, sondern auch gleich für alle Paare s mit $t_1 \cdots t_j$ für $j \in [0 : m]$. Diese Distanzen stehen in der letzten Zeile. Somit brauchen wir die Distanzen nur für alle Suffixe $t^k := t_k \cdots t_m$ von t mit s zu berechnen. Wir können dann die Ergebnisse der Distanzen von $t_k \cdots t_\ell$ für $k \leq \ell \in [0 : m]$ mit s auslesen. Da es nur $\Theta(m)$ Suffixe von t gibt, ist der Zeitbedarf dann nur noch $O(nm^2)$.

Wir können noch ein wenig effizienter werden, wenn wir die Suffixe von t mit Hilfe eines Suffix-Baumes $T(t\$)$ verwalten. Wir durchlaufen jetzt diesen Suffix-Baum mit Hilfe der Tiefensuche. Für jedes Blatt, das wir besuchen, erhalten wir ein Suffix von t und berechnen für dieses die Tabelle der optimalen Alignment-Distanzen gegen s.

Hierbei können wir einige Einträge jedoch geschickt recyceln. Betrachten wir zwei Knoten v und w die durch eine Kante $(v, w) \in E(T(t\$))$ verbunden sind (w ist also ein Kind von v) und die beiden zugehörigen Teilwörter (Präfixe von Suffixen) t_v und t_w von t. Ist $\text{label}(v, w)$ das Kantenlabel der Kante (v, w), dann gilt nach Definition eines Suffix-Baumes: $t_w = t_v \cdot \text{label}(v, w)$. Um nun die Tabelle für s und

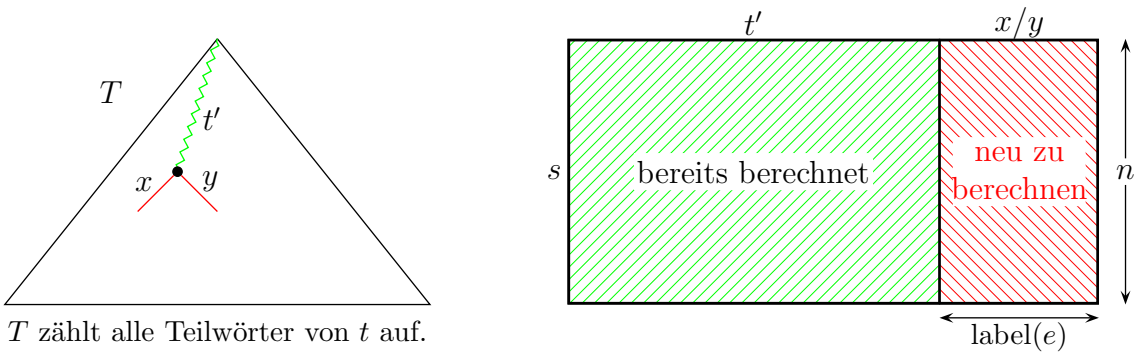


Abbildung 4.41: Skizze: Hybrides Verfahren für One-Against-All

t_w zu berechnen, können wir die linke Hälfte für s und t_v wiederverwenden und müssen nur die letzten $|\text{label}(v,w)|$ Spalten neu berechnen. Dies ist schematisch in Abbildung 4.41 dargestellt.

Damit ergibt sich unmittelbar der folgende Algorithmus, der in Abbildung 4.42 angegeben ist. Hierbei berechnet compute_table(D, s, t', k, ℓ) die Tabelle mit den optimalen Alignment-Distanzen von s mit $t' = t_1 \cdots t_\ell$, wobei die Einträge $D(i,j)$ für $i \in [0:n]$ und $j \in [0:k]$ schon berechnet sind.

DFS (node v, char $s[]$, char $t'[]$)

```
begin
    forall ((v,w) ∈ E(T(t$))) do
        compute_table(D, s, t' · label(v,w), |t'|, |t' · label(v,w)|);
        DFS(w, s, t' · label(v,w));
end
```

Abbildung 4.42: Algorithmus: Hybrides Verfahren für One-Against-All

Wir können bei der Ausgabe natürlich bei alle Varianten leicht die Ergebnisse herausfiltern, deren Wert ϑ nicht überschreitet. In einer echten Implementierung werden natürlich nicht die Zeichenreihen selbst, sondern wieder Referenzen auf das zugehörige benötigte Kantenlabel in t angegeben. Die Angabe der vollständigen Zeichenreihen dient hier nur dem besseren Verständnis des Algorithmus. Der Leser möge sich selbst überlegen, wie die Implementierung aussehen würde.

Die Laufzeit beträgt dann $O(m)$ für die Konstruktion des Suffix-Baumes für t, da der Suffix-Baum ja $O(|t|) = O(m)$ Knoten besitzt. Für die Berechnung der Tabellenerweiterungen fällt für jede Kante $(v,w) \in E(T(t\$))$ die Zeit $O(|s| \cdot |\text{label}(v,w)|)$

an. Somit ergibt sich für die gesamte Laufzeit $T(n,m)$:

$$\begin{aligned} T(n,m) &= O(m) + O\Bigg(\sum_{(v,w)\in E(T(t))} n \cdot |\text{label}(v,w)| \Bigg) \\ &= O(m) + O\Bigg(n \cdot \underbrace{\sum_{(v,w)\in E(T(t\$))} |\text{label}(v,w)|}_{=:\text{size}(T(t\$))} \Bigg) \\ &= O(m + n \cdot \text{size}(T(t\$))). \end{aligned}$$

Hierbei ist die Größe $\text{size}(T)$ eines Suffix-Baumes T durch die Summe aller Längen der Kantenlabels von T gegeben. Wie wir uns bei den Suffix-Tries schon überlegt haben, gilt dann für einen Suffix-Baum $T(t\$)$: $\text{size}(T(t\$)) = O(m^2) \cap \Omega(m)$.

Theorem 4.40 *Seien $s \in \Sigma^n$, $t \in \Sigma^m$ und $\vartheta \in \mathbb{R}_+$. Alle optimalen Alignment-Distanzen für s und t' mit $t' \sqsubseteq t$ und $d(s,t') \leq \vartheta$ lassen sich mit Hilfe des hybriden Verfahrens in Zeit $O(n \cdot \text{size}(T(t)) + D) \subseteq O(nm^2)$ bestimmen, wobei $D = |\{t' \sqsubseteq t : d(s,t') \leq \vartheta\}|$.*

Experimente mit realistischen (biologischen) Daten haben ergeben, dass $\text{size}(T(t\$))$ in der Regel ungefähr $m^2/10$ entspricht.

In der Praxis wird man aus Platzgründen wiederum Suffix-Arrays gegenüber Suffix-Bäumen vorziehen. In einem Suffix-Array S kommen ja alle Suffixe von t in lexikographischer Reihenfolge vor. Mit Hilfe der LCP-Tabelle L weiß man dann auch auch wieviele Buchstaben des i-ten Suffix (an Position $S[i]$) mit dem vorherigen Suffix übereinstimmen, man kennt also $|t'| = L[i]$ in der Abbildung 4.41.

Wir können den Algorithmus auch so modifizieren, dass er neben der Distanz auch ein zugehöriges optimales Alignment ausgibt. Wir haben ja die Matrizen D gespeichert, so dass der Traceback leicht ermittelt werden kann. Dann ist D im Satz nicht mehr die Anzahl der augegebenen Werte, sondern die Ausgabegröße. Wenn man nur an den Distanzen interessiert ist, braucht man sich von der Distanzmatrix D nur die Spalten zu speichern, deren zugehörige Wörter in der horizontalen Knoten im Suffix-Baum entsprechne. Dann lässt sich der Speicherplatzbedarf noch verringern.

4.4.2 All-Against-All-Problem

Im *All-Against-All-Problem* wollen wir für zwei gegebene Sequenzen $s, t \in \Sigma^*$ alle globalen Alignments von s' gegen alle Teilwörter von t berechnen, sofern diese

Distanz ein gewisse Schranke ϑ unterschreitet. Formal wird das Problem wie folgt beschrieben.

ALL-AGAINST-ALL-PROBLEM

Eingabe: Sei $s \in \Sigma^n$ und $t \in \Sigma^m$ und $\vartheta \in \mathbb{R}_+$.
Gesucht: Berechne $d(s', t')$ für alle $s' \sqsubseteq s$ und $t' \sqsubseteq t$ mit $d(s', t') \leq \vartheta$.

Wir betrachten zuerst einen naiven Ansatz und berechnen für jedes Teilwort s' von s sowie jedes Teilwort t' von t deren Alignment. Die Anzahl der der Paare von Teilwörtern von s' bzw. t' von s bzw. t mit $|s| = n$ bzw. $|t| = m$ beträgt $\Theta(n^2m^2)$. Da der Aufwand pro Alignment $O(nm)$ ist, beträgt der Gesamtaufwand $O(n^3m^3)$.

Etwas geschickter können wir wieder vorgehen, wenn wir uns an die Tabelle $D(i, j)$ erinnern und bemerken, dass wir ja nicht nur die optimale Alignment-Distanz von $s = s_1 \cdots s_n$ mit $t = t_1 \cdots t_m$ berechnen, sondern auch gleich für alle Paare $s_1 \cdots s_i$ mit $t_1 \cdots t_j$ für $i \in [0 : n]$ und $j \in [0 : m]$. Diese Distanzen stehen ja über die gesamte Tabelle verteilt in $D(i, j)$. Somit brauchen wir die Distanzen nur für alle Suffixe $s^k = s_k \cdots s_n$ und $t^{k'} := t_{k'} \cdots t_m$ von t mit s zu berechnen. Wir können dann die Ergebnisse der Distanzen von $s_k \cdots s_\ell$ und $t_{k'} \cdots t_{\ell'}$ für $k \leq \ell \in [0 : n]$ und $k' \leq \ell' \in [0 : m]$ aus D auslesen. Da es nur $\Theta(n)$ Suffixe von s und $\Theta(m)$ Suffixe von t gibt, ist der Zeitbedarf dann nur noch $O(n^2m^2)$.

Ist $\vartheta = \infty$ so ist diese Methode optimal, da wir ja $\Theta(n^2m^2)$ Paare von Ergebnissen ausgeben müssen. Da wir jetzt nur noch die Paare ausgeben wollen, deren Alignment-Distanz kleiner gleich ϑ ist, können wir mit Hilfe eines hybriden Verfahrens effizienter vorgehen.

Wir werden wieder die Suffixe von s und t mit Hilfe von Suffix-Bäumen verwalten. Hierzu sei $T(s\$)$ bzw. $T(t\$)$ der Suffix-Baum von s bzw. t. Wir durchlaufen jetzt beide Suffix-Bäume von s und t parallel mit Hilfe der Tiefensuche. Für jedes Paar von Knoten $(v, v') \in V(T(s\$)) \times V(T(t\$))$ die wir besuchen, erhalten wir ein Paar von Suffixen von s' bzw. t' und berechnen für diese die Tabelle der optimalen Alignment-Distanzen.

Hierbei können wir wiederum einige Einträge geschickt recyceln. Betrachten wir zwei Paare von Knoten v und w sowie v' und w', die durch eine Kante $(v, w) \in E(T(s\$))$ sowie $(v', w') \in E(T(t\$))$ verbunden sind (w bzw. w' ist also ein Kind von v bzw. v') und die beiden zugehörigen Suffixe s_v von s und $t_{v'}$ von t. Ist $\text{label}(v, w)$ bzw. $\text{label}(v', w')$ das Kantenlabel der Kante (v, w) bzw. (v', w'), dann gilt nach Definition eines Suffix-Baumes: $s_w = s_v \cdot \text{label}(v, w)$ bzw. $t_{w'} = t_{v'} \cdot \text{label}(v', w')$. Um nun die Tabelle für s_w und $t_{w'}$ zu berechnen, können wir den größten Teil links und oben für s_v bzw. s_w sowie $t_{v'}$ wiederverwenden und müssen nur den rechten unteren Teil

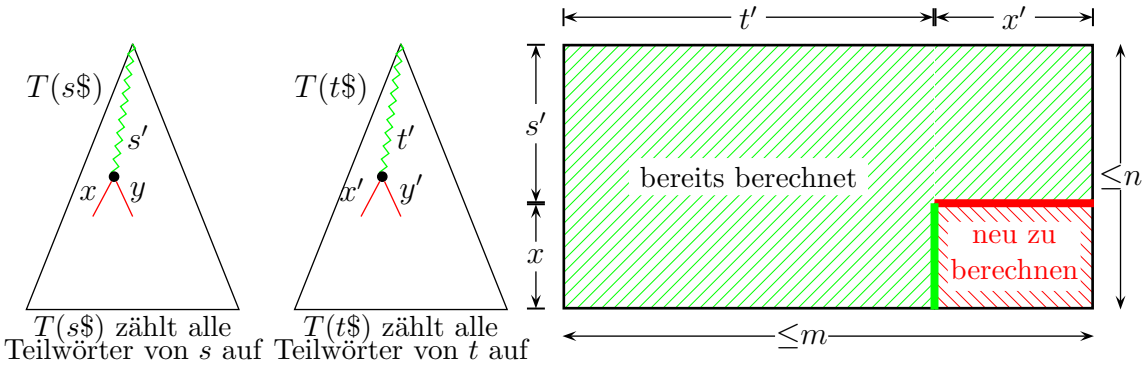


Abbildung 4.43: Skizze: Hybrides Verfahren für All-Against-All

($|\text{label}(v,w)|$ Zeilen sowie $|\text{label}(v',w')|$ Spalten) neu berechnen. Dies ist schematisch in Abbildung 4.43 dargestellt.

Die Werte im dicken grünen Balken sind dabei schon berechnet und stehen auch gerade in der Tabelle der dynamischen Programmierung zur Verfügung. Die Werte im dicken roten Balken sind ebenfalls schon berechnet worden (beim Alignment von s' gegen $t' \cdot x'$) und wurden seinerzeit gespeichert und können jetzt sehr einfach in die Tabelle der dynamischen Programmierung kopiert werden, um anschließend damit weiterzurechnen.

Damit ergibt sich der in Abbildung 4.44 angegebene Algorithmus, der zu Beginn mit $\text{DFS}_\text{s}(r(T(s)), \varepsilon)$ aufgerufen wird. Die Prozedur $\text{compute\_table}(D, s', t', k, k', \ell, \ell')$

DFS_s (node v, char $s'[]$)

```
begin
    forall ((v, w) ∈ E(T(s$))) do
        DFS_t(r(T(t$)), s' · label(v, w), ε, |s'|, |s' · label(v, w)|);
        DFS_s(w, s' · label(v, w));
end
```

DFS_t (node w, char $s'[]$, char $t'[]$, int k, int k')

```
begin
    forall ((w, w') ∈ E(T(t$))) do
        compute_table(D, s', t' · label(v', w'), k, k', |t'|, |t' · label(v', w')|);
        DFS_t(w', s', t' · label(v', w'), |s'|.|t' · label(v', w')|);
end
```

Abbildung 4.44: Algorithmus: Hybrides Verfahren für All-Against-All

berechnet die Tabelle mit den optimalen Alignment-Distanzen von $s' = s_k \cdots s_\ell$ mit $t' = t_{k'} \cdots t_{\ell'}$, wobei einige Einträge in $D(i,j)$ schon berechnet sind.

Für die Laufzeit $T(n,m)$ gilt (wobei der Term $O(n+m)$ von der Konstruktion der Suffix-Bäume für $s\$$ und $t\$$ herrührt):

$$\begin{aligned} T(n,m) &= O(n+m) + O\Bigg(\sum_{(v,w)\in E(T(s))} \sum_{(v',w')\in E(T(t))} |\text{label}(v,w)| \cdot |\text{label}(v',w')| \Bigg) \\ &= O(n+m) + O\Bigg(\sum_{(v,w)\in E(T(s))} |\text{label}(v,w)| \sum_{(v',w')\in E(T(t))} |\text{label}(v',w')| \Bigg) \\ &= O(n+m) + O(\text{size}(T(s\$)) \cdot \text{size}(T(t\$))) \\ &= O(\text{size}(T(s\$)) \cdot \text{size}(T(t\$))). \end{aligned}$$

Halten wir das Ergebnis noch in dem folgenden Satz fest.

Theorem 4.41 *Seien $s \in \Sigma^n$, $t \in \Sigma^m$ und $\vartheta \in \mathbb{R}_+$. Alle optimalen Alignment-Distanzen für s' und t' mit $s' \sqsubseteq s$ und $t' \sqsubseteq t$ und $d(s',t') \leq \vartheta$ lassen sich mit Hilfe des hybriden Verfahrens in Zeit $O(\text{size}(T(s\$)) \cdot \text{size}(T(t\$)) + D) \subseteq O(n^2m^2)$ bestimmen, wobei D die Anzahl der Paare (s',t') mit $d(s',t') \leq \vartheta$ ist.*

Man kann sich auch leicht überlegen, dass man gewisse Teile der Suffix-Bäume nicht weiter durchsucht, nämlich dann, wenn die bereits gefundenen Distanz zu groß ist. Diese Verfahren lassen sich natürlich auch auf Ähnlichkeitsmaße übertragen, wobei hier das Auslassen der Teilbäume zu einer Heuristik wird.

23.07.19

Approximative Algorithmen 5

5.1 $\mathcal{NP}$-Vollständigkeit

In diesem Kapitel wollen wir uns mit Entscheidungsproblemen beschäftigen, die zwar lösbar sind, für die allerdings bislang nur superpolynomielle Algorithmen bekannt sind und für die vermutet wird, dass sie keine polynomiellen Algorithmen besitzen.

Für unsere Untersuchungen werden Funktionen mit Wertebereich $\mathbb{B} := \{0, 1\}$ von besonderem Interesse sein. Wir bezeichnen eine solche Funktion $f : \mathbb{N}^m \to \mathbb{B}$ als ein *Entscheidungsproblem*. Dabei interpretieren wir die Ausgabe 1 als Ja und die Ausgabe 0 als Nein. Wir haben solche Entscheidungsprobleme schon kennengelernt: „Ist die gegebene Folge sortiert?“, „Ist die Zeichenkette s in der Zeichenkette t enthalten?“ Hierbei waren die Argumente allerdings nicht immer natürliche Zahlen. Man kann jedoch jede Zeichenfolge über einem festen Alphabet als eine natürliche Zahl interpretieren. Auch mehrere Argumente können in einer natürlichen Zahl kodiert werden, so dass wir im Folgenden nur Funktionen $f : \mathbb{N} \to \mathbb{B}$ betrachten werden.

5.1.1 Rechenmodelle

Bevor wir uns dem eigentlichen Problem zuwenden wollen, müssen wir uns erst noch über das zugrunde liegende Rechenmodell klar werden, bzw. ob dieses überhaupt einen wesentlichen Unterschied macht. In der Informatik werden insbesondere die folgenden Rechenmodelle betrachtet:

- Turingmaschinen,
- 2-Zähler-Automaten,
- Registermaschinen (RAM),
- WHILE-Programme, GOTO-Programme,
- verallgemeinerte Petri-Netze (mit 2 Inhibitor-Kanten),
- Markov-Algorithmen,
- Semi-Thue-Systeme,
- zelluläre Automaten,
- μ-rekursive Funktionen,
- λ-Kalkül.

Es hat sich jedoch herausgestellt, dass Funktionen, die in einem dieser Rechenmodelle berechenbar sind, auch in jedem anderen Rechenmodell berechenbar sind. Wir wiederholen hierzu kurz die Church-Turing-These aus Einführung in die Theoretische Informatik (die auch bei Ersetzung von Turingmaschinen durch jedes andere vorhergenannte Modell gilt).

These 5.1 (Church-Turing-These) *Turingmaschinen sind berechnungsuniversell, d.h. jede Funktion, die auf einem effektiven Rechenmodell berechnet werden kann, kann auch auf der Turingmaschine effektiv berechnet werden.*

Da nicht definiert ist (und da es auch nicht die Absicht ist, zu definieren), was ein *effektives* Rechenmodell ist, ist die These nicht beweisbar (sie kann höchstens widerlegt werden).

Weiterhin hat sich gezeigt, dass selbst die Laufzeiten sich nur um ein Polynom verändern, wenn man das zugrunde liegende Rechenmodell ändert. Dies wird als die erweiterte Church-Turing-These bezeichnet.

These 5.2 (Erweiterte Church-Turing-These) *Jede Funktion, die auf einem effektiven Rechenmodell $\mathcal{R}$ in Zeit $t(n)$ berechnet werden kann, kann auch auf einer deterministischen Turingmaschine effektiv in Zeit $O(p(t(n)+n))$ berechnet werden, wobei p ein Polynom ist, dass nur von $\mathcal{R}$ abhängt.*

Hierbei muss jedoch angemerkt werden, dass diese erweiterte Church-Turing-These durchaus falsch sein kann. Für Quanten-Computer hat sich gezeigt, dass das Problem der Primfaktorzerlegung auf Quanten-Computern (Stichwort Shors Algorithmus) in polynomieller Zeit lösbar ist. Zum einen ist momentan noch nicht klar, ob sich Quanten-Computer in beliebiger wirklich Größe bauen lassen (was für gängige Kryptographie-Verfahren das Aus bedeuten würde). Zum anderen würde selbst dies nicht die Widerlegung der erweiterten Church-Turing-These bedeuten, da es durchaus möglich sein kann, dass es einen polynomiellen Algorithmus in einem gängigen Rechenmodell gibt, der die Primfaktorzerlegung lösen kann.

Zur Bestimmung, ob eine gegebene natürliche Zahl prim ist oder nicht, war auch lange Zeit nicht bekannt, ob es einen polynomiellen Algorithmus hierfür gibt. Erst seit dem Jahr 2002 ist ein solcher Algorithmus bekannt. Jedoch hat die Entscheidbarkeit, ob eine Zahl zusammengesetzt ist oder nicht (also prim ist), nicht allzuviel damit zu tun, auch wirklich einen Teiler (oder auch Primteiler) dieser Zahl zu finden.

Aufgrund der (erweiterten) Church-Turing-These wird im Folgenden die Turingmaschine als Rechenmodell verwendet (auch wenn wir diese nicht formal definieren wollen), da diese einem Standardrechner am nächsten kommt.

5.1.2 Die Klassen $\mathcal{P}$ und $\mathcal{NP}$

In der Praxis hat sich gezeigt, dass polynomiell zeitbeschränkte Algorithmen noch praktikabel sind. Aus diesem Grunde definiert man die so genannte Klasse $\mathcal{P}$.

Definition 5.3 *Ein Entscheidungsproblem P gehört zur Klasse $\mathcal{P}$, wenn eine polynomiell zeitbeschränkte Turingmaschine M existiert, wobei $M(x) = P(x)$ für alle $x \in \mathbb{N}$ gilt.*

Wir weisen darauf hin, dass in diesem Kapitel immer die logarithmische Zeitkomplexität zugrunde gelegt ist. Die Klasse $\mathcal{P}$ ist also der Versuch, den informalen Begriff der effizient lösbaren Entscheidungsprobleme zu definieren. Dabei geht es hier im Wesentlichen um das asymptotische Laufzeitverhalten. Für kleine Eingabegrößen, sagen wir 32, kann ein exponentieller Algorithmus mit Laufzeit 2^n wesentlich effizienter sein als ein polynomieller Algorithmus mit Laufzeit n^8.

Entscheidungsprobleme lassen sich oft in der folgenden Art formulieren: „Hat ein Objekt eine gewisse Eigenschaft?“, wobei es einen „kurzen“ Beweis für das Vorhandensein dieser Eigenschaft gibt. Dabei ist es meist einfacher, den Beweis für eine Eingabe zu verifizieren, anstatt diesen Beweis selbst zu finden. Betrachten wir das folgende Entscheidungsproblem.

HC (Hamiltonian-Circuit)

Eingabe: Ein ungerichteter Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$.
Gesucht: Gibt es einen Hamiltonschen Kreis, d.h. eine Permutation der Knoten $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ mit $\{v_{\pi(i)}, v_{\pi((i \bmod n)+1)}\} \in E$?

Im Entscheidungsproblem HC wollen wir also feststellen, ob ein gegebener Graph einen einfachen Kreis auf allen Knoten als Teilgraphen enthält. Um dieses Entscheidungsproblem zu lösen, können wir alle Permutationen von V betrachten und feststellen, ob für eine dieser Permutationen die benötigten Kanten in G enthalten sind. Leider liefert diese Methode einen exponentiellen Algorithmus, da es $|V|!$ viele Permutationen von V gibt.

Wird uns zu der Eingabe von einem Helferlein noch eine Permutation der Knoten $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ mitgegeben, so können wir sehr leicht in polynomieller Zeit entscheiden, ob der durch diese Permutation induzierte Kreis in G enthalten ist. Würde uns dieses Helferlein zu einem Graphen immer eine Permutation liefern, so dass der induzierte Kreis in G enthalten ist, sofern ein solcher existiert, dann wäre das Entscheidungsproblem sehr leicht lösbar. Falls der gegebene Graph keinen Hamiltonschen Kreis enthält, kann uns andererseits keine Permutation von V in die Irre führen.

Ein solches Hilfsmittel wollen wir ein *Zertifikat* nennen. Man beachte, dass Zertifikate von dem gegebenen Entscheidungsproblem abhängen. Mit Hilfe solcher Zertifikate lässt sich die Komplexitätsklasse $\mathcal{NP}$ definieren.

Definition 5.4 *Ein Entscheidungsproblem P gehört zur Klasse $\mathcal{NP}$, wenn es eine polynomiell zeitbeschränkte deterministische Turingmaschine M und ein Polynom q gibt, so dass*

- *für jede Eingabe x mit $P(x) = 1$ ein Zertifikat z mit $\|z\| \leq q(\|x\|)$ existiert, so dass $M(x, z) = 1$ ist, und*
- *für jede Eingabe x mit $P(x) = 0$ und für jedes Zertifikat z mit $\|z\| \leq q(\|x\|)$ gilt, dass $M(x, z) = 0$ ist.*

Aus der Definition folgt sofort, dass $\mathcal{P} \subseteq \mathcal{NP}$ (für das Problem P wird das Zertifikat z gar nicht benötigt). Man beachte dabei, dass es für ein Entscheidungsproblem $P \in \mathcal{NP}$ und Eingabe x mit $P(x) = 0$ kein Zertifikat z (mit $\|z\| \leq q(\|x\|)$) geben darf, auf der die Turingmaschine M auf (x, z) mit 1 antwortet.

Wie sich herausstellen wird, ist die obige Definition für viele Komplexitätsbetrachtungen sehr fruchtbar. Für die Praxis ist die Definition auf den ersten Blick nutzlos, da sie für ein Entscheidungsproblem $P \in \mathcal{NP}$ eine polynomiell zeitbeschränkte deterministische Turingmaschine postuliert, die P unter Zuhilfenahme eines Zertifikats entscheidet. Aber woher soll man in der Praxis diese Zertifikate hernehmen? Uns wird im positiven Fall ja nur deren Existenz und im negativen Fall deren Nichtexistenz zugesichert. Leider gibt es keine Aussage darüber, wie man diese Zertifikate effizient (d.h. in polynomieller Zeit) konstruieren kann.

Wofür steht nun eigentlich das $\mathcal{N}$ in $\mathcal{NP}$? Die Klasse $\mathcal{NP}$ ist die Menge der Entscheidungsprobleme, die *nichtdeterministisch* in polynomieller Zeit gelöst werden können. Wie gesagt, wissen wir nur von der Existenz eines Zertifikates, wenn eine Eingabe positiv beantwortet werden kann. Rein mathematisch können wir nun von einer so genannten *nichtdeterministischen Turingmaschine* fordern, dass sie zu Beginn der Berechnung einfach ein korrektes Zertifikat für die gegebene Eingabe zur Verfügung stellt, vorausgesetzt, es gibt ein solches. Die Konstruktion eines solchen Zertifikats erfolgt nichtdeterministisch. Mathematisch ist das absolut zulässig, auch wenn man eine solche Maschine in der Praxis so überhaupt nicht konstruieren kann.

Eine nichtdeterministische Turingmaschine ist somit ein Beispiel für ein nichteffektives Rechenmodell (das man allerdings durch Aufzählen aller möglichen Zertifikate zu einem solchen erweitern kann, wobei sich dann die Laufzeit natürlich drastisch erhöht).

5.1.3 Reduktionen

Nun wollen wir ein mächtiges und zugleich alltägliches Konzept vorstellen: die so genannte Reduktion. Bei einer Reduktion wird ein Entscheidungsproblem in ein anderes transformiert, so dass man aus der Lösung des transformierten Entscheidungsproblems die Lösung des ursprünglichen Entscheidungsproblems erhält.

Reduktionen sind in zweierlei Hinsicht hilfreich. Zum einen kann man damit versuchen, wie oben schon erwähnt, ein Entscheidungsproblem zu lösen, indem man es auf ein anderes reduziert und dieses löst. Zum anderen bieten sie komplexitätstheoretisch eine Ordnung auf den Entscheidungsproblemen. Kann man ein Entscheidungsproblem A auf ein Entscheidungsproblem B reduzieren, so ist wohl B als schwieriger anzusehen als A, sofern man nicht zu viel Aufwand in die Reduktion gesteckt hat. Uns wird die letztere Interpretation später noch beschäftigen. Formal erhalten wir die folgende Definition einer Reduktion.

Definition 5.5 *Seien $P, P' \subseteq \mathbb{N}$ zwei Entscheidungsprobleme. Eine Abbildung $\rho : \mathbb{N} \to \mathbb{N}$ ist eine* Reduktion *von P auf P', wenn gilt:*

$$\forall x \in \mathbb{N} : \; P(x) = P'(\rho(x)).$$

Hierfür schreibt man auch $P \leq^{\rho} P'$ bzw. $P \leq P'$.

Können wir nun das Entscheidungsproblem P' lösen, so können wir auch P lösen, indem wir die Eingabe x von P mittels ρ auf eine Eingabe $\rho(x)$ für P' transformieren und die Antwort für $\rho(x) \in P'$ übernehmen. Diese Vorgehensweise ist im Bild 5.1 schematisch dargestellt.

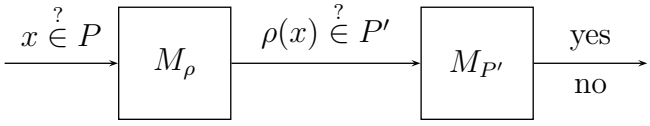


Abbildung 5.1: Schematische Darstellung einer Reduktion von P auf P'

Wir wollen an dieser Stelle noch deutlich darauf hinweisen, dass bei dieser Definition der Reduktion die Antwort aus dem reduzierten Entscheidungsproblem übernommen werden muss. Es ist also nicht möglich, dass wir die Antwort von „$\rho(x) \in P'$?“ negieren. Es lassen sich natürlich solche allgemeineren Reduktionen definieren (und werden es auch), aber wir kommen im Folgenden mit dieser stärkeren Definition einer Reduktion aus.

Lemma 5.6 *Reduktionen sind transitiv, d.h. $P \leq P' \wedge P' \leq P'' \Rightarrow P \leq P''$.*

Wenn wir uns mit Entscheidungsproblemen beschäftigen, die in polynomieller Zeit verifiziert oder gelöst werden sollen, dann sind insbesondere Reduktionen von Interesse, die selbst auf einer polynomiell zeitbeschränkten Turingmaschine berechnet werden können.

15.10.19

Definition 5.7 *Eine Reduktion ρ heißt* polynomiell*, wenn ρ in polynomieller Zeit auf einer Turingmaschine berechnet werden kann. Für eine polynomielle Reduktion ρ von P nach P' schreibt man kurz $P \leq_p^\rho P'$ bzw. $P \leq_p P'$.*

Zu Ehren von R. Karp werden solche polynomielle Reduktionen oft auch als *Karp-Reduktionen* bezeichnet. Wir werden nun noch festhalten, dass Karp-Reduktionen transitiv sind. Diese Transitivität wird noch eine wichtige Rolle spielen.

Lemma 5.8 *Karp-Reduktionen sind transitiv.*

5.1.4 $\mathcal{NP}$-harte und $\mathcal{NP}$-vollständige Probleme

Nun können wir definieren, was wir im Folgenden unter einem schwierigen Entscheidungsproblem verstehen wollen.

Definition 5.9 *Ein Entscheidungsproblem P heißt $\mathcal{NP}$-*hart*, wenn sich jedes Entscheidungsproblem aus der Klasse $\mathcal{NP}$ auf P polynomiell reduzieren lässt, d.h.*

$$\forall P' \in \mathcal{NP} : \exists \rho : \mathbb{N} \to \mathbb{N} : P' \leq_p^\rho P.$$

Nun können wir die schwierigsten Entscheidungsprobleme aus der Klasse $\mathcal{NP}$ definieren.

Definition 5.10 *Ein Entscheidungsproblem P heißt $\mathcal{NP}$-*vollständig*, wenn P $\mathcal{NP}$-hart ist und zugleich $P \in \mathcal{NP}$ gilt.*

Aus der Vorlesung Einführung in die Theoretische Informatik kennen wir bereits ein generisches $\mathcal{NP}$-vollständiges Entscheidungsproblem, nämlich SAT, worauf wir später noch zu sprechen kommen.

Wir werden jetzt noch einige Folgerungen aus der Existenz von $\mathcal{NP}$-vollständigen Entscheidungsproblemen ziehen. Zuerst halten wir fest, dass mit der Kenntnis bereits

eines $\mathcal{NP}$-vollständigen Entscheidungsproblems und der Transitivität der Karp-Reduktion der Nachweis anderer $\mathcal{NP}$-vollständiger Entscheidungsprobleme leichter wird.

Lemma 5.11 *Sei Q $\mathcal{NP}$-hart, $P \in \mathcal{NP}$ und $Q \leq_p P$, dann ist P $\mathcal{NP}$-vollständig.*

Wir halten noch fest, dass bereits die Kenntnis eines polynomiell zeitbeschränkten Algorithmus für *nur ein einziges* $\mathcal{NP}$-hartes Entscheidungsproblem ausreicht, um $\mathcal{P} = \mathcal{NP}$ zu beweisen.

Theorem 5.12 *Sei P ein $\mathcal{NP}$-hartes Entscheidungsproblem und $P \in \mathcal{P}$, dann ist $\mathcal{P} = \mathcal{NP}$.*

Allerdings hat man auch nach intensiver Suche noch keinen polynomiell zeitbeschränkten Algorithmus für ein $\mathcal{NP}$-hartes Entscheidungsproblem gefunden. Daraus leitet man die allgemein anerkannte Vermutung ab, dass $\mathcal{NP}$-harte Entscheidungsprobleme keinen polynomiell zeitbeschränkten Algorithmus besitzen können.

Vermutung 5.13 *Es gilt $\mathcal{P} \neq \mathcal{NP}$.*

Damit wird der Nachweis der $\mathcal{NP}$-Härte eines Entscheidungsproblems als ein Nachweis dafür angesehen, dass dieses Problem *nicht* effizient lösbar ist.

Bevor wir im Folgenden das Entscheidungsproblem SAT formulieren können, benötigen wir noch einige elementare Begriffe aus der Booleschen Algebra bzw. Booleschen Aussagenlogik.

Definition 5.14 *Eine* Boolesche Variable *ist eine Variable, die einen der beiden Booleschen Werte aus $\mathbb{B} = \{0, 1\}$ annehmen kann.*

Der Einfachheit halber identifizieren wir mit 1 bzw. 0 den Booleschen Wert `true` bzw. `false`. Die Werte 0 und 1 bezeichnen wir auch als die *Booleschen Konstanten* und fassen sie in der Menge $\mathbb{B} = \{0, 1\}$ zusammen.

Definition 5.15 *Eine* Boolesche Formel *ist induktiv über einer abzählbar unendlichen Menge von Variablen $X = \{x_1, x_2, x_3, \ldots\}$ wie folgt definiert:*

- *Jede Boolesche Variable und jede Boolesche Konstante ist eine Boolesche Formel.*
- *Sind F_1 und F_2 Boolesche Formeln, dann ist auch*
  - *die* Konjunktion $(F_1 \wedge F_2)$,
  - *die* Disjunktion $(F_1 \vee F_2)$,
  - *die* Negation $\overline{F}_1$

  eine Boolesche Formel.

Die in einer Booleschen Formel F verwendeten Variablen werden mit $V(F) \subset X$ bezeichnet.

Man beachte, dass für jede Boolesche Formel F die Menge $V(F)$ endlich ist. Die Menge der Booleschen Formeln über X bezeichnen wir mit $\mathcal{B}(X)$ oder einfach mit $\mathcal{B}$. Nach der Beschreibung der Syntax von Booleschen Formeln geben wir nun die Semantik an.

Definition 5.16 *Sei F eine Boolesche Formel über den Booleschen Variablen $V(F)$ und $B : V(F) \to \mathbb{B}$ eine* Belegung *der Booleschen Variablen. Eine Belegung B auf den Booleschen Variablen* induziert *eine* Interpretation $\mathcal{I}_B : \mathcal{B} \to \mathbb{B}$ *einer Booleschen Formel wie folgt:*

- *Die Interpretation der Booleschen Konstanten ist von der Belegung unabhängig: $\mathcal{I}_B(1) = 1$ bzw. $\mathcal{I}_B(0) = 0$.*
- *Die Interpretation $\mathcal{I}_B$ auf einer Booleschen Variablen $x \in V(F)$ wird durch die Belegung der Variablen induziert: $\mathcal{I}_B(x) = B(x)$.*
- *Ist eine Boolesche Formel $F = (F_1 \wedge F_2)$ eine Konjunktion zweier Boolescher Formeln F_1, F_2, so ist die Interpretation $\mathcal{I}_B(F) = \min\{\mathcal{I}_B(F_1), \mathcal{I}_B(F_2)\}$.*
- *Ist eine Boolesche Formel $F = (F_1 \vee F_2)$ eine Disjunktion zweier Boolescher Formeln F_1, F_2, so ist die Interpretation $\mathcal{I}_B(F) = \max\{\mathcal{I}_B(F_1), \mathcal{I}_B(F_2)\}$.*
- *Ist eine Boolesche Formel $F = \overline{F}_1$ eine Negation einer Booleschen Formel F_1, so ist die Interpretation $\mathcal{I}_B(F) = 1 - \mathcal{I}_B(F_1)$.*

Man kann sich leicht überlegen, dass die Operationen $\vee$ und $\wedge$ unter der gegebenen Interpretation assoziativ und kommutativ sind. Aufgrund der Assoziativität werden

wir oft die Klammerungen weglassen. Wir schreiben dann z.B. $x \vee y \vee z$ anstatt von $(x \vee y) \vee z$ oder $x \vee (y \vee z)$.

Definition 5.17 *Eine Boolesche Formel F heißt* erfüllbar*, wenn es eine Belegung B der Booleschen Variablen $V(F)$ gibt, so dass für die induzierte Interpretation $\mathcal{I}_B(F) = 1$ gilt.*

Die Menge der erfüllbaren Formeln bezeichnen wir mit

$$\textsc{Sat} = \{F \in \mathcal{B} \,:\, \exists B : V(F) \to \mathbb{B} \text{ mit } \mathcal{I}_B(F) = 1\} \subsetneq \mathcal{B}$$

Damit handelt es sich bei Sat um ein Entscheidungsproblem.

Theorem 5.18 (Satz von Cook und Levin) Sat *ist $\mathcal{NP}$-vollständig.*

Dieses Theorem wurde von S. Cook zu Beginn der 70er Jahre des 20. Jahrhunderts westlich des Eisernen Vorhangs bewiesen. Unabhängig davon wurde zur selben Zeit von L.A. Levin ein analoges Resultat östlich des Eisernen Vorhangs bewiesen.

Ein wichtiger Aspekt dieses Theorem ist, dass wir nun ein allererstes Entscheidungsproblem als $\mathcal{NP}$-vollständig erkannt haben. Dies macht aufgrund der Transitivität der Karp-Reduktion den Nachweis der $\mathcal{NP}$-Härte wesentlich leichter (siehe Lemma 5.11).

5.1.5 Beispiele $\mathcal{NP}$-vollständiger Probleme

In diesem Abschnitt wollen wir noch einige wichtige $\mathcal{NP}$-vollständige Entscheidungsprobleme vorstellen, die wir im weiteren Verlauf dieses Kapitels noch benötigen werden. Dazu benötigen wir erst noch einige Begriffe.

Definition 5.19 *Eine Boolesche Variable bzw. ihre Negation heißt* Literal*. Eine* Klausel *ist eine Disjunktion von Literalen. Eine Boolesche Formel F ist in* konjunktiver Normalform*, wenn F eine Konjunktion von Klauseln ist. Eine Boolesche Formel F ist in k-*konjunktiver Normalform*, wenn F in konjunktiver Normalform ist und zusätzlich jede Klausel aus maximal k Literalen besteht.*

Mit Cnf-Sat bezeichnen wir die Menge der erfüllbaren Booleschen Formeln in konjunktiver Normalform.

Cnf-Sat

Eingabe: Eine Boolesche Formel F in konjunktiver Normalform.
Gesucht: Gibt es eine Belegung B von $V(F)$, so dass $\mathcal{I}_B(F) = 1$?

Wir weisen darauf hin, dass in der Literatur manchmal auch die Notation SAT anstelle von CNF-SAT verwendet wird.

Dem Leser sei als Hausaufgabe überlassen, das folgende nicht ganz triviale Lemma zu beweisen.

Lemma 5.20 *Es gilt* SAT $\leq_p$ CNF-SAT.

Mit Lemma 5.11 folgt aus der obigen Reduktion sofort das folgende Korollar:

Korollar 5.21 CNF-SAT *ist $\mathcal{NP}$-vollständig.*

Mit k-SAT bezeichnen wir die Menge der erfüllbaren Booleschen Formeln in k-konjunktiver Normalform.

k-SAT

Eingabe: Eine Boolesche Formel F in k-konjunktiver Normalform.
Gesucht: Gibt es eine Belegung B von $V(F)$, so dass $\mathcal{I}_B(F) = 1$?

Mit 3SAT bzw. mit 2SAT bezeichnen wir zwei Spezialfälle von k-SAT für $k = 3$ bzw. $k = 2$. Dem Leser sei als Hausaufgabe überlassen, das folgende Lemma zu beweisen.

Lemma 5.22 *Es gilt* CNF-SAT $\leq_p$ k-SAT $\leq_p$ 3SAT *für* $k \geq 3$..

Mit Lemma 5.11 folgt aus der obigen Reduktion sofort das folgende Korollar:

Korollar 5.23 *Für $k \geq 3$ sind* k-SAT *und* 3SAT *$\mathcal{NP}$-vollständig.*

2SAT hat hingegen eine ganz andere Komplexität.

Lemma 5.24 *Es gilt* 2SAT $\in \mathcal{P}$.

Als nächstes betrachten wir das Entscheidungsproblem des Hamiltonschen Kreises in gerichteten Graphen.

DHC (DIRECTED HAMILTONIAN CIRCUIT)

Eingabe: Ein gerichteter Graph $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$.
Gesucht: Gibt es einen Hamiltonschen Kreis, d.h. eine Permutation der Knoten $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ mit $(v_{\pi(i)}, v_{\pi((i \bmod n)+1)}) \in E$?

In die Einführung in die Theoretische Informatik wurde bereits das folgende Resultat gezeigt.

Theorem 5.25 DHC *ist $\mathcal{NP}$-vollständig.*

Daraus folgt mit einer einfachen Reduktion, dass auch das Problem, ob ein ungerichteter Graph einen Hamiltonschen Kreis enthält, $\mathcal{NP}$-vollständig ist.

Theorem 5.26 HC *ist $\mathcal{NP}$-vollständig.*

Wir geben jetzt noch ohne Beweis zwei $\mathcal{NP}$-vollständige Entscheidungsprobleme an, die wir später noch benötigen werden.

PARTITION

Eingabe: Eine Folge $(s_1, \ldots, s_n) \in \mathbb{N}^n$.
Gesucht: Gibt es eine Partition (I_1, I_2) von $[1:n]$, so dass für diese Partition gilt: $\sum_{i \in I_1} s_i = \sum_{i \in I_2} s_i$?

In die Einführung in die Theoretische Informatik wurde ebenfalls das folgende Resultat gezeigt.

Theorem 5.27 PARTITION *ist $\mathcal{NP}$-vollständig.*

Für das zweite Problem benötigen wir noch den Begriff einer unabhängigen Menge. Eine Teilmenge $V' \subseteq V(G)$ eines ungerichteten Graphen G, in der es keine Kante zwischen zwei Knoten aus V' gibt, nennen wir eine *unabhängige Menge* von G.

IS (INDEPENDENT SET)

Eingabe: Ein ungerichteter Graph $G = (V, E)$ und ein $B \in \mathbb{N}$.
Gesucht: Gibt es eine unabhängige Teilmenge von V der Größe mindestens B, d.h. existiert $V' \subseteq V$ mit $|V'| \geq B$ und $\binom{V'}{2} \cap E = \emptyset$?

Das Problem ist ebenfalls $\mathcal{NP}$-vollständig.

Theorem 5.28 IS *ist $\mathcal{NP}$-vollständig.*

5.2 Optimierungsprobleme und Approximationen

Wir haben eben gesehen, dass es unter der Annahme von $\mathcal{P} \neq \mathcal{NP}$ Entscheidungsprobleme gibt, die nicht in polynomieller Zeit gelöst werden können. Dies ist auf den ersten Blick niederschmetternd, da alle nicht-polynomiell zeitbeschränkten Algorithmen für große Eingaben deutlich zu langsam sind. In der Praxis begegnet man allerdings eher Optimierungsproblemen als Entscheidungsproblemen.

5.2.1 Optimierungsprobleme

Im Gegensatz zu Entscheidungsproblemen, die nur die Antwort ja oder nein kennen, besitzen Optimierungsprobleme sehr viele zulässige Lösungen, wobei man aber an einer besten interessiert ist. Nehmen wir als Beispiel die Berechnung eines paarweisen Alignments. Es gibt exponentiell viele paarweise Alignments für zwei gegebene Sequenzen. Gesucht ist aber ein paarweises Alignment mit optimalen Score bzgl. des zugrunde liegenden Distanz- oder Ähnlichkeitsmaßes.

In der Praxis kann aber durchaus eine Lösung genügen, die nahe am Optimum ist, wenn man dadurch Rechenzeit für die Berechnung der Lösung einsparen kann. Daher wollen wir uns in diesem Abschnitt mit der Approximation von $\mathcal{NP}$-harten Problemen beschäftigen. Zunächst einmal müssen wir definieren, was wir unter einem Optimierungsproblem und dessen näherungsweisen Lösung verstehen wollen.

Definition 5.29 *Ein* Optimierungsproblem *ist ein 4-Tupel* $P = (I, S, \mu, \text{opt})$ *mit den folgenden Eigenschaften:*

- $I \subseteq \Sigma^*$ *ist die Menge der zulässigen Eingaben (Instanzen).*
- $S : I \to 2^{\Sigma^*}$ *ist eine Abbildung der Eingaben auf die Mengen der zulässigen Lösungen, d.h. für* $x \in I$ *ist* $S(x)$ *die Menge der zulässigen Lösungen von* x*.*
- $\mu(x, y) \in \mathbb{Q}_+$ *ist das* Maß der Lösung $y \in S(x)$ *für die Eingabe* $x \in I$*. Weiterhin ist genau dann* $\mu(x, y) = 0$*, wenn* $y \notin S(x)$ *gilt.*
- $\text{opt} \in \{\max, \min\}$ *gibt an, ob ein Maximierungs- oder ein Minimierungsproblem betrachtet wird.*

Ist $\text{opt} = \max$ *(bzw.* $\text{opt} = \min$*), dann wird für eine Eingabe* $x \in I$ *eine Lösung* $y \in S(x)$ *gesucht, so dass* $\mu(x, y)$ *maximal (bzw. minimal) ist.*

Anschaulich ist I die Menge der korrekten Codierungen der Eingaben des Optimierungsproblems P (über einem geeignet gewählten Alphabet Σ). Die Abbildung S liefert uns zu einer konkreten Eingabe x alle zulässigen Lösungen der Eingabe x für das Optimierungsproblem P. Man beachte, dass wir noch keine Aussagen über die

Komplexität der Entscheidbarkeit von $x \in I$ bzw. $y \in S(x)$ und der Berechenbarkeit von μ gemacht haben.

17.10.19

Definition 5.30 *Sei $P = (I, S, \mu, \text{opt})$ ein Optimierungsproblem. Für eine Eingabe $x \in I$ heißt eine Lösung $y^* \in S(x)$* optimal*, wenn*

$$\mu(x, y^*) = \text{opt}\{\mu(x, y) \,:\, y \in S(x)\}.$$

Das Maß einer optimalen Lösung *bezeichnen wir mit*

$$\mu^*(x) := \text{opt}\{\mu(x, y) \,:\, y \in S(x)\}.$$

Das aysmptotische Maß einer optimalen Lösung *bezeichnen wir mit*

$$\mu^+(x) := \text{opt}^+\{\mu(x, y) \,:\, y \in S(x)\}.$$

Hierbei ist $\min^+ = \inf$ *und* $\max^+ = \sup$.

Beachte, dass in der obigen Definition weder y^* noch $\mu^*(x)$ existieren muss. Wenn $\mu^*(x)$ existiert, dann ist $\mu^*(x) = \mu^+(x)$.

Zu einem Optimierungsproblem können wir das zugehörige Entscheidungsproblem definieren. Hierbei soll entschieden werden, ob die optimale Lösung besser als eine vorgegebene Schranke ist.

Definition 5.31 *Sei $P = (I, S, \mu, \text{opt})$ ein Optimierungsproblem und sei $B \in \mathbb{Q}_+$. Dann ist $\{x \in I : \exists y \in S(x) : \mu(x, y) = \text{opt}(\mu(x, y), B)\}$ das* zugehörige Entscheidungsproblem.

Wir wollen hier noch anmerken, dass genau dann $\mu(x, y) = \text{opt}(\mu(x, y), B)$ für ein $y \in S(x)$ ist, wenn die optimale Lösung mindestens den Wert B erreicht. Wir nennen ein Optimierungsproblem $\mathcal{NP}$*-hart*, wenn das zugehörige Entscheidungsproblem $\mathcal{NP}$-hart ist.

Um die Definition eines Optimierungsproblems etwas anschaulicher darzustellen, betrachten wir ein konkretes Optimierungsproblem.

MinBinPacking

Eingabe:	Eine Folge $(s_1, \ldots, s_n) \in \mathbb{N}^n$ und $B \in \mathbb{N}$.
Lösung:	Eine Partition $(I_1, \ldots, I_m)$ von $[1 : n]$, so dass $\sum_{i \in I_j} s_i \leq B$ für alle $j \in [1 : m]$ gilt.
Optimum:	Minimiere m.

Anschaulich haben wir hier n Objekte, wobei das i-te Objekt Gewicht s_i hat. Die Objekte sollen nun in möglichst wenig Kisten gepackt werden, wobei jede Kiste eine Tragkraft von B hat.

Für die Formulierung der Eingaben verwenden wir ein geeignetes Alphabet Σ. Für eine Eingabe $x \in I$ gibt nun $S(x)$ alle möglichen Verteilungen der n Objekte auf Kisten an, so dass keine Kiste ihre Tragfähigkeit überschreitet (oder leer bleibt). Das Maß $\mu(x, y)$ für ein $y \in S(x)$ gibt dann an, wie viele Kisten bei der Verteilung y verwendet wurden (oder ist 0, wenn $y \notin S(x)$). Bei MinBinPacking handelt es sich um ein Minimierungsproblem, da wir möglichst wenig Kisten verwenden wollen.

5.2.2 Approximationen und Approximationsgüte

Nachdem wir uns klar gemacht haben, was ein Optimierungsproblem ist, wollen wir nun den Begriff der Approximation definieren.

Definition 5.32 *Sei $P = (I, S, \mu, \text{opt})$ ein Optimierungsproblem. Ein Algorithmus A ist eine r-*Approximation *für P, wenn für alle $x \in I$ mit $S(x) \neq \emptyset$ gilt, dass $A(x) \in S(x)$ und dass*

$$\Gamma_A(x) := \max\left(\frac{\mu(x, A(x))}{\mu^+(x)}, \frac{\mu^+(x)}{\mu(x, A(x))}\right) \leq r.$$

Mit $\Gamma_A(x)$ wird die Güte *des Algorithmus A auf der Eingabe x bezeichnet. Mit $\Gamma_A = \sup\{\Gamma_A(x) : x \in I\}$ wird die* maximale Güte *des Algorithmus A bezeichnet. Mit $\Gamma_A^\infty = \limsup_{\gamma\to\infty}\{\Gamma_A(x) : x \in I \wedge \mu^+(x) \geq \gamma\}$ wird die* asymptotische Güte *des Algorithmus A bezeichnet.*

Die Maximumbildung in der Definition von $\Gamma_A(x)$ ist auf den ersten Blick etwas verwirrend. Für Minimierungsprobleme liefert das erste, für Maximierungsprobleme das zweite Argument das Maximum. Damit gilt unabhängig von der Art des Optimierungsproblems, dass $\Gamma_A(x) \geq 1$ gilt. Damit erhalten wir eine von der Art des Optimierungsproblems unabhängige Definition der Güte einer Approximation, die Güten leichter vergleichbar machen.

Ist ein Algorithmus A eine r-Approximation des Optimierungsproblems P, so folgt aus der Definition, dass die vom Algorithmus A generierte Lösung im schlimmsten Fall um den Faktor r vom Optimum abweicht. Für MinBinPacking bedeutet dies, dass wir bei einer r-Approximation anstatt einer optimalen Lösung mit k Kisten eine näherungsweise Lösung mit maximal $r \cdot k$ Kisten erhalten.

5.2.3 Beispiel: MinBinPacking

Wir wollen nun noch einen sehr einfachen Approximationsalgorithmus für MinBinPacking angeben und nachweisen, dass dieser eine 2-Approximation liefert.

Theorem 5.33 *Es gibt eine 2-Approximation für* MinBinPacking*, die sich in polynomieller Zeit berechnen lässt.*

Beweis: Wir werden einen Greedy-Algorithmus für MinBinPacking angeben. Die Kisten werden von 1 an durchnummeriert. Der Algorithmus sucht nun für jedes Objekt eine Kiste, wobei die Kisten in der Reihenfolge ihrer Nummerierung betrachtet werden. Passt das i-te Objekt mit Gewicht s_i noch in die j-te Kiste, dann werfen wir das Objekt in diese. Andernfalls testen wir die $j+1$-te Kiste. Da wir beliebig viele Kisten zur Verfügung haben, finden wir auf jeden Fall eine Kiste, in die das i-te Objekt passt (siehe auch Abbildung 5.2).

Nachdem wir alle Objekte auf m Kisten verteilt haben, stellen wir fest, dass es keine zwei Kisten geben kann, die zusammen maximal Gewicht B haben. Gäbe es zwei Kisten $j_1 < j_2$, die zusammen Gewicht maximal B hätten, dann wären nach unserem Algorithmus alle Objekte aus der Kiste j_2 spätestens in die Kiste j_1 platziert worden, da diese für den Inhalt beider Kisten noch genügend Tragkraft gehabt hätte.

MinBinPacking(int $s[]$, int n, int B)

```
// Assuming s_i ≤ B for all i
begin
    int m := 0;
    for (i := 1; i ≤ n; i++) do
        A[i] := 0;
        for (j := 1; j ≤ m; j++) do
            if (K_j + s_i ≤ B) then
                K_j := K_j + s_i;
                A[i] := j;
                break;
        if (A[i] = 0) then
            m++;
            K_m := s_i;
            A[i] := m;
end
```

Abbildung 5.2: Algorithmus: MinBinPacking

Jetzt lässt sich zeigen, dass das Gesamtgewicht aller Kisten größer als $m \cdot B/2$ ist. Dazu sei K_j das Gewicht der j-ten Kiste. Dann erhalten wir für das Gesamtgewicht aller verwendeten Kisten (bzw. Objekte):

$$\sum_{i=1}^{n} s_i = \sum_{j=1}^{m} K_j = \sum_{j=1}^{m-1} \frac{K_j + K_{j+1}}{2} + \frac{K_m + K_1}{2} > m \cdot \frac{B}{2} = \frac{m}{2} \cdot B.$$

Da die Tragkraft einer Kiste B ist, sind also für eine optimale Lösung mehr als $m/2$ Kisten notwendig. Damit hat unser Greedy-Algorithmus höchstens doppelt so viele Kisten verwendet wie nötig. Somit liefert unser Greedy-Algorithmus eine 2-Approximation. Der Beweis der Laufzeit sei dem Leser zur Übung überlassen. ■

Mit etwas mehr Aufwand lässt sich zeigen, dass der obige Algorithmus sogar eine 1,7-Approximation ist. Wendet man denselben Algorithmus auf die gemäß ihrer Gewichte absteigend sortierte Liste von Objekten an, dann erhält man sogar eine 1,5-Approximation. Auf der anderen Seite werden wir später noch sehen, dass dies die bestmögliche Approximationsgüte eines Algorithmus für MinBinPacking sein kann. Für weitere Details verweisen wir den Leser auf das Buch von G. Ausiello et al. über Komplexität und Approximation.

5.3 Komplexitätsklassen für Optimierungsprobleme

In diesem Abschnitt geben wir die Definitionen von den wichtigsten Komplexitätsklassen für Optimierungsprobleme und deren Implikationen für den Entwurf effizienter Approximationsalgorithmen an.

5.3.1 Die Klassen $\mathcal{NPO}$ und $\mathcal{PO}$

Wir definieren zunächst die wichtige Klasse $\mathcal{NPO}$, die die meisten der in der Praxis bedeutsamen Optimierungsprobleme umfasst.

Definition 5.34 *Ein Optimierungsproblem $P = (I, S, \mu, \text{opt})$ gehört zur Klasse $\mathcal{NPO}$, wenn folgendes gilt:*

- *Es kann in polynomieller Zeit entschieden werden, ob $x \in I$ ist oder nicht, d.h. $I \in \mathcal{P}$.*
- *Die Größe jeder Lösung ist polynomiell in der Eingabegröße beschränkt, d.h. es existiert ein Polynom p, so dass für jedes $x \in I$ und jedes $y \in S(x)$ gilt: $\|y\| \leq p(\|x\|)$.*
- *Das Maß μ ist in polynomieller Zeit berechenbar.*

Man beachte, dass man aufgrund der polynomiellen Berechenbarkeit von μ, auch in polynomieller Zeit entscheiden kann, ob $y \in S(x)$ ist oder nicht, da ja $\mu(x, y)$ genau dann den Wert 0 annimmt, wenn $y \notin S(x)$ gilt.

Die Klasse $\mathcal{NPO}$ besteht also im Wesentlichen aus allen Optimierungsproblemen, für die es einen nichtdeterministischen polynomiell zeitbeschränkten Algorithmus gibt, der zu einer gegebenen Eingabe eine Lösung verifiziert und deren Güte berechnet. Es ist allerdings nicht klar, wie man eine optimale Lösung mit Hilfe von solchen nichtdeterministischen polynomiell zeitbeschränkten Algorithmen finden kann, da man nur die Güte einer vorgegebenen Lösung berechnen kann, aber nicht notwendigerweise die Güte der optimalen Lösung.

Wir merken auch noch an, dass für Optimierungsprobleme aus $\mathcal{NPO}$ die Menge $S(x)$ für jedes $x \in I$ endlich ist, da ja jedes Element $y \in S(x)$ polynomiell in der Eingabegröße beschränkt ist ($\|y\| \leq p(\|x\|)$). Somit existiert für $x \in I$ auch eine optimale Lösung und μ^* ist für alle x definiert.

Als Übungsaufgabe überlassen wir dem Leser den Beweis der folgenden wichtigen Beziehung zwischen $\mathcal{NPO}$ und $\mathcal{NP}$.

Theorem 5.35 *Sei $P \in \mathcal{NPO}$ ein Optimierungsproblem, dann ist das zugehörige Entscheidungsproblem in $\mathcal{NP}$.*

Nun können wir die Klasse $\mathcal{PO}$ definieren, die alle Optimierungsprobleme enthält, die sich in polynomieller Zeit optimal lösen lassen.

Definition 5.36 *Ein Optimierungsproblem $P = (I, S, \mu, \text{opt})$ gehört zur Klasse $\mathcal{PO}$, wenn $P \in \mathcal{NPO}$ und wenn es einen polynomiell zeitbeschränkten Algorithmus A gibt, der für jede Eingabe $x \in I$ eine optimale Lösung $A(x) \in S(x)$ berechnet, d.h. $\mu(x, A(x)) = \mu^*(x)$, bzw. feststellt, dass es gar keine Lösung gibt.*

Aus den Definitionen folgt unmittelbar, dass $\mathcal{PO} \subseteq \mathcal{NPO}$. Als Beispiel der Klasse $\mathcal{PO}$ sei hier wieder das Optimierungsproblem des paarweisen Sequenzen-Alignments erwähnt.

SeqAlign

Eingabe:	Zwei Sequenzen $s, t \in \Sigma^*$.
Lösung:	Ein Alignment $(\overline{s}, \overline{t})$ von s und t mit $\sigma(\overline{s}, \overline{t}) > 0$, wobei σ ein Ähnlichkeitsmaß ist.
Optimum:	Maximiere $\sigma(\overline{s}, \overline{t})$.

Manchmal muss man die übliche Definition eines gegebenen Optimierungsproblems etwas anpassen, damit es nach unserer Definition ein Optimierungsproblem ist bzw. es in $\mathcal{NPO}$ enthalten ist. Zum einen ist hier wichtig, dass in Alignments die Alignierung von Leerzeichen gegeneinenader verboten ist, sonst gäbe es Lösungen deren Größe nicht polynomiell in der Eingabegröße ist. Zum anderen haben wir hier nur Alignments mit einem positiven Ähnlichkeitsmaß als Lösung zugelassen, da sonst die Eigenschaft der Positivität des Maßes $\mu((s,t),(\overline{s},\overline{t}))$ nach der Definition eines Optimierungsproblems nicht erfüllt wäre. Diese Einschränkungen stellen in der Regel keine Einschränkung dar, da diese Lösungen in der Regel sowieso nicht von Interesse sind. Der Leser möge sich überlegen, wie man in diesen Rahmenbedingungen das Sequenzen-Alignment mit Distanzfunktionen definieren würde.

22.10.19

Ebenfalls als Übungsaufgabe überlassen wir dem Leser den Beweis der folgenden wichtigen Beziehung zwischen $\mathcal{PO}$ und $\mathcal{P}$.

Theorem 5.37 *Sei $P \in \mathcal{PO}$ ein Optimierungsproblem, dann ist das zugehörige Entscheidungsproblem in $\mathcal{P}$.*

5.3.2 Die Klasse $\mathcal{APX}$

Zuerst definieren wir die Klasse $\mathcal{APX}$, die aus all solchen Optimierungsproblemen besteht, die sich in polynomieller Zeit bis auf einen konstanten Faktor approximieren lassen.

Definition 5.38 *Ein Optimierungsproblem $P = (I, S, \mu, \text{opt})$ gehört zur Klasse $\mathcal{APX}(r)$, wenn $P \in \mathcal{NPO}$ ist und es einen polynomiell zeitbeschränkten Algorithmus A gibt, so dass A eine r-Approximation für P ist. Die Klassen $\mathcal{APX}$ und $\mathcal{APX}^*$ sind definiert als $\mathcal{APX} = \bigcup_{r\geq 1} APX(r)$ bzw. $\mathcal{APX}^* = \bigcap_{r>1} APX(r)$.*

Aus der Definition folgt unmittelbar, dass $\mathcal{PO} \subseteq \mathcal{APX}^* \subseteq \mathcal{APX} \subseteq \mathcal{NPO}$. Mit Hilfe von Lemma 5.33 folgt nun, dass MinBinPacking in der Klasse $\mathcal{APX}$ enthalten ist. Wir wollen nun noch zeigen, dass die Klasse $\mathcal{APX}$ eine echte Teilmenge von $\mathcal{NPO}$ ist. Dazu betrachten wir das Problem der Handlungsreisenden:

TSP (Traveling Salesperson)

Eingabe:	Ein vollständiger Graph $G = (V, E)$ und Kantengewichten w.
Lösung:	Ein Hamiltonscher Kreis, d.h. eine Permutation $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ der Knotenmenge V mit $(v_{\pi(i)}, v_{\pi((i \bmod n)+1)}) \in E$ und $n = \lvert V\rvert$.
Optimum:	Minimiere $\sum_{i=1}^{n} w(v_{\pi(i)}, v_{\pi((i \bmod n)+1)})$ mit $n = \lvert V\rvert$.

Das Problem der Handlungsreisenden (TSP) kann man sich wie folgt vorstellen. Gegeben sind n Knoten, die z.B. die Großstädte von Europa repräsentieren. Zwischen zwei Städten gibt es eine Kante, wenn sie per Flugzeug oder Bahn verbunden sind. Das Gewicht einer Kante kann man als den Zeitaufwand werten, um von der einen in die andere Stadt zu gelangen. Unsere Handlungsreisende will nun jede Großstadt Europas besuchen, um ihre Artikel zu verkaufen, und möchte dabei möglichst wenig Zeit für die Reisen verschwenden. Die optimale Rundreise ergibt sich gerade als Lösung des zugehörigen TSP. Wir werden nun zeigen, dass sich diese Rundreise im Allgemeinen wohl nicht bis auf eine Konstante approximieren lässt. Dass TSP $\in \mathcal{NPO}$ gilt, überlassen wir dem Leser als Übungsaufgabe.

Theorem 5.39 *Es gilt* TSP $\notin \mathcal{APX}$*, außer wenn* $\mathcal{P} = \mathcal{NP}$*.*

Beweis: Wir zeigen, dass aus der Approximierbarkeit von TSP folgt, dass HC in $\mathcal{P}$ wäre. Da HC $\mathcal{NP}$-vollständig ist, folgt mit Theorem 5.12, dass dann $\mathcal{P} = \mathcal{NP}$ wäre.

Nehmen wir an, es gäbe eine r-Approximation A für TSP. Sei $G = (V, E)$ mit $|V| = n$ eine Eingabe für HC. Wir konstruieren daraus eine Eingabe $G' = (V, \binom{V}{2})$ mit den folgenden Kantengewichten $w(e)$ für $e \in \binom{V}{2}$:

$$w(e) = \begin{cases} 1 & \text{falls } e \in E, \\ 1 + r \cdot n & \text{sonst.} \end{cases}$$

Enthält G einen Hamiltonschen Kreis, dann enthält G' eine optimale Rundreise mit Gewicht genau n. Andernfalls hat jede Rundreise im Graphen G' mindestens das Gewicht $(n-1) + (1+rn) = (r+1)n$. Da A eine r-Approximation für das TSP ist, findet A eine Rundreise mit Gewicht maximal rn, falls G einen Hamiltonschen Kreis enthält. Andernfalls hat jede Rundreise, die A liefert, mindestens das Gewicht $(r+1)n$. Also enthält G genau dann einen Hamiltonschen Kreis, wenn A eine Rundreise in G' mit Gewicht maximal rn liefert (die dann genau n sein muss). ■

Wir wollen hier noch anmerken, dass es eine 1,5-Approximation gibt, wenn man voraussetzt, dass die Dreiecksungleichung für die Kantengewichte gilt. Diese Approximation wurde im Jahr 1976 von N. Christofides entwickelt. Für den Fall, dass die Knoten Punkte im Euklidischen Raum sind und die Kantengewichte dem Euklidischen Abstand entsprechen, wurden im Jahre 1996 von S. Arora bessere Approximationsalgorithmen gefunden, so genannte polynomielle Approximationsschemata (siehe nächster Abschnitt). Die zugehörigen Entscheidungsprobleme für diese Spezialfälle von TSP bleiben allerdings $\mathcal{NP}$-hart.

Korollar 5.40 *Es gilt* $\mathcal{APX} \subsetneq \mathcal{NPO}$*, außer wenn* $\mathcal{P} = \mathcal{NP}$*.*

5.3.3 Die Klasse $\mathcal{PTAS}$

Optimierungsprobleme aus der Klasse $\mathcal{APX}$ haben die schöne Eigenschaft, dass sich Lösungen bis auf eine Konstante approximieren lassen. Noch schöner wäre es jedoch, wenn man sich die Approximationsgüte wünschen könnte, d.h. wenn es zu jedem $r > 1$ einen polynomiell zeitbeschränkten Approximationsalgorithmus geben würde. Dies leistet die Klasse $\mathcal{APX}^*$. Leider hängt hierbei der zu wählende Algorithmus von der Approximationsgüte ab. Schöner wäre es einen einheitlichen Algorithmus zu haben, der neben der Eingabe x auch noch die gewünschte Approximationsgüte $1+\varepsilon$ als Eingabe bekommt. Das motiviert die folgende Definition eines Approximationsschemas.

Definition 5.41 *Sei $P = (I, S, \mu, \text{opt}) \in \mathcal{NPO}$ ein Optimierungsproblem. Ein Algorithmus A ist ein* polynomielles Approximationsschema*, wenn es für jedes $\varepsilon > 0$ ein Polynom p_ε gibt, so dass für jede Eingabe $x \in I$ gilt, dass $A(x, \varepsilon) \in S(x)$ sowie:*

$$\Gamma_{A(\varepsilon)}(x) \leq 1 + \varepsilon \qquad \text{und} \qquad T_{A(\varepsilon)}(x) \leq p_\varepsilon(\|x\|).$$

Ein Optimierungsproblem gehört zur Klasse $\mathcal{PTAS}$, wenn es ein polynomielles Approximationsschema besitzt.

Man beachte, dass der Algorithmus A nicht nur $x \in I$, sondern auch $\varepsilon > 0$ als Eingabe erhält. Mit $A(\varepsilon)$ bezeichnen wir dann den Algorithmus, den wir aus A für ein festes $\varepsilon > 0$ erhalten. Damit ist A eigentlich nicht ein Algorithmus, sondern er besteht aus einer ganzen Schar von Algorithmen. Dies erklärt auch den Namen Approximationsschema. Man beachte, dass wir hier allerdings eine gewisse Uniformität der Approximationsalgorithmen fordern, da wir eine einheitliche und endliche Beschreibung aller Approximationsalgorithmen fordern.

Die Klasse $\mathcal{PTAS}$ (engl. polynomial time approximation scheme) enthält also genau die Optimierungsprobleme aus $\mathcal{NPO}$, die wir beliebig genau in polynomieller Zeit approximieren können. Nach Definition gilt, dass $\mathcal{PO} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX}$. Wir wollen nun zunächst zeigen, dass $\mathcal{PTAS}$ echt in $\mathcal{APX}$ enthalten ist.

Theorem 5.42 MinBinPacking *lässt sich nicht besser als mit einer Approximationsgüte von $3/2$ in polynomieller Zeit approximieren, außer wenn $\mathcal{P} = \mathcal{NP}$.*

Beweis: Wir zeigen nun, dass wir Partition in polynomieller Zeit entscheiden können, wenn wir MinBinPacking beliebig genau approximieren können. Wir zeigen sogar, dass aus jeder r-Approximation von MinBinPacking mit $r < 1{,}5$ folgt, dass Partition $\in \mathcal{P}$ ist.

Nehmen wir an, dass es eine $(3/2-\varepsilon)$-Approximation für MinBinPacking gäbe. Sei $S = (s_1, \ldots, s_n)$ eine Eingabe von Partition, dann konstruieren wir eine Eingabe $S' = ((s_1, \ldots, s_n), B)$ mit $B = \frac{1}{2}\sum_{i=1}^{n} s_i$ für MinBinPacking. Falls $B \notin \mathbb{N}$ ist, sind wir fertig, da in diesem Fall die Eingabe von Partition gar keine Lösung zulässt.

Nun stellen wir fest, dass es genau dann eine Lösung für Partition für S gibt, wenn es eine Lösung für MinBinPacking für S' mit 2 Kisten gibt. Da wir nun eine $(3/2-\varepsilon)$-Approximation für MinBinPacking haben, gibt es genau dann eine Partition, wenn unsere Approximation mit maximal $(3/2-\varepsilon)\cdot 2 = 3-2\varepsilon$ Kisten auskommt. Da unsere Approximation immer nur ganzzahlige Lösungen konstruieren kann, erhalten wir eine Lösung mit 2 Kisten, also eine optimale. Damit wäre dann Partition $\in \mathcal{P}$. Da Partition ein $\mathcal{NP}$-vollständiges Problem ist, folgt mit Theorem 5.12, dass $\mathcal{P} = \mathcal{NP}$. ■

Korollar 5.43 *Es gilt $\mathcal{PTAS} \subsetneq \mathcal{APX}$, außer wenn $\mathcal{P} = \mathcal{NP}$.*

Was wir hier nicht beweisen, aber erwähnen wollen, ist, dass die asymptotische Approximationsgüte von MinBinPacking gleich 1 ist. Die erzielbare maximale und asymptotische Approximationsgüte eines Problems können sich also unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ durchaus unterscheiden.

Wir wollen jetzt noch ein Problem vorstellen, das unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ keine kleinen asymptotischen Approximationsgüten besitzt.

Definition 5.44 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine* k-Färbung *(genauer eine* k-Knotenfärbung*) ist eine Abbildung $\varphi : V \to [1:k]$. Eine k-Färbung φ heißt* zulässig*, wenn $\varphi(v) \neq \varphi(w)$ für alle $\{v, w\} \in E$. Die* chromatische Zahl *eines Graphen G ist definiert durch $\chi(G) = \min\{k \in \mathbb{N}_0 : \exists \varphi : V \to [1:k] \wedge \varphi \text{ ist zulässig}\}$.*

MinGC (Minimum Graph Coloring)

Eingabe: Ein ungerichteter Graph $G = (V, E)$.
Lösung: Eine zulässige k-Färbung von G für ein $k \in [1 : |V|]$.
Optimum: Minimiere k.

Das zugehörige Entscheidungsproblem ist selbst für $k = 3$ $\mathcal{NP}$-hart. Für $k = 2$ gibt es einen einfachen polynomiellen Algorithmus. Der Leser sei aufgefordert, sich diesen zu überlegen.

Theorem 5.45 MinGC *lässt sich nicht besser als mit einer asymptotischen Approximationsgüte von 4/3 in polynomieller Zeit approximieren, außer wenn $\mathcal{P} = \mathcal{NP}$.*

Beweis: Wir zeigen dazu, dass eine asymptotische $(4/3-\varepsilon)$-Approximation für MinGC für ein $\varepsilon > 0$ einen polynomiellen Algorithmus für 3-GC impliziert.

Sei also A ein Approximationsalgorithmus für MinGC mit $\Gamma_A^\infty = 4/3 - \varepsilon$. Dann gibt es nach Definition der asymptotischen Approximationsgüte ein $k \in \mathbb{N}$, so dass $\Gamma_A(G) \leq 4/3 - \frac{\varepsilon}{2}$ für alle Graphen G mit $\chi(G) \geq k$.

Gegeben sei die Eingabe $G = (V, E)$ für 3-GC. Wir konstruieren einen Graphen $G' = (V', E')$ als Eingabe für MinGC wie folgt. Jeder Knoten $v \in V$ wird durch einen vollständigen Graphen G_v auf k Knoten ersetzt. Für jede Kante $\{v, w\} \in E$ werden k^2 Kanten zwischen jedem Knoten in G_v und jedem Knoten in G_w gezogen. Man überlegt sich, dass $\chi(G') = k \cdot \chi(G)$ gilt.

Wir wenden jetzt A auf G' an. Ist $\chi(G) = 3$, dann ist nach Konstruktion $\chi(G') = 3k$. Unser Approximationsalgorithmus liefert dann ein Lösung mit einer Färbung von maximal $(4/3-\frac{\varepsilon}{2})3k < 4k$. Da wir aufgrund der Konstruktion wissen, dass die chromatische Zahl von G' ein Vielfaches von k ist, muss $\chi(G') \in \{1k, 2k, 3k\}$ sein. Daraus können wir folgern, dass $\chi(G) \leq 3$ gilt. Ist $\chi(G) \geq 4$, dann ist nach Konstruktion $\chi(G') \geq 4k$. Damit können wir leicht feststellen, ob der gegebene Graph eine chromatische Zahl kleiner gleich 3 besitzt (nämlich genau dann, wenn $\chi(G') < 4k$ ist). Das liefert den gewünschten Widerspruch, außer wenn $\mathcal{P} = \mathcal{NP}$. ■

Wir wollen noch anmerken, dass sich MinGC auch nicht auf einen Faktor $|V|^{1/7-\varepsilon}$ approximieren lässt, also noch nicht einmal zu $\mathcal{APX}$ gehört.

Wir wollen jetzt noch ein Optimierungsproblem angeben, das in $\mathcal{PTAS}$ enthalten ist: die Bestimmung einer Medians der Länge L von Sequenzen mit kleinster Hamming-Distanz. Definieren wir zuerst die Hamming-Distanz zweier Sequenzen.

Definition 5.46 *Seien $s, t \in \Sigma^n$ zwei Sequenzen gleicher Länge, dann ist die* Hamming-Distanz *von s und t definiert als $d_H(s,t) = \#\{i \in [1:n] : s_i \neq t_i\}$.*

MinConsPat (Minimum Consensus Pattern)

Eingabe: Eine Folge von Sequenzen $s^{(1)}, \ldots, s^{(n)} \in \Sigma^m$ und ein $L \in \mathbb{N}$.
Lösung: Ein Median-String $s \in \Sigma^L$, $t^{(i)} \in \Sigma^L$ mit $t^{(i)} \sqsubseteq s^{(i)}$ für $i \in [1:n]$.
Optimum: Minimiere $\sum_{i=1}^n d_H(s, t^{(i)})$.

Wir zeigen jetzt noch, dass das zugehörige Entscheidungsproblem ConsPat von MinConsPat $\mathcal{NP}$-vollständig ist.

Theorem 5.47 ConsPat *ist $\mathcal{NP}$-vollständig.*

Beweis: Zuerst zeigen wir, dass ConsPat $\in \mathcal{NP}$. Als Zertifikat wählen wir eine geeignete Kodierung von $(j_1, \ldots, j_n) \in [1 : m - L + 1]^n$ und dem Median-String $s \in \Sigma^L$. Mit Hilfe des Zertifikates können wir die folgenden Bedingung in polynomieller Zeit testen:

$$\sum_{i=1}^{n} d_H(s, t^{(i)}) \leq B,$$

wobei $t^{(i)} := s^{(i)}_{j_i} \cdots s^{(i)}_{j_i+L-1}$. Damit ist ConsPat $\in \mathcal{NP}$.

29.10.19

Nun zeigen wir, dass 3Sat $\leq_p$ MinConsPat gilt. Sei $F \in \mathcal{B}$ eine Boolesche Formel in 3-konjunktiver Normalform. Ohne Beschränkung der Allgemeinheit nehmen wir jetzt hier an, dass jede Klausel in F aus genau 3 verschiedenen Literalen besteht und dass die Menge der Variablen in F wie folgt gegeben ist: $V(F) = \{x_1, \ldots, x_\ell\}$. Der Leser möge sich überlegen, dass dies keine Einschränkung darstellt. Sei also $F = \bigwedge_{i=1}^{k} C_i$, wobei jedes C_i eine Disjunktion von genau drei Literalen ist.

Für jede Klausel C_i definieren wir einen Zeichenreihe $s^{(i)}$ für $i \in [1 : k]$ wie folgt:

$$s^{(i)} := 0c_i^{(1)}0\sharp_1^L 0c_i^{(2)}0\sharp_1^L \cdots \sharp_1^L 0c_i^{(7)}0,$$

wobei $c_i^{(j)} \in \{0, 1, *\}^\ell$ und $L = \ell + 2$ ist. Dabei beschreiben $c_i^{(1)}, \ldots, c_i^{(7)}$ die sieben verschiedenen erfüllenden Belegungen der Klausel C_i, wobei die Variablen, die in C_i nicht vorkommen mit $*$ besetzt werden. Für ein Beispiel betrachte $C = (x_1 \vee \overline{x_3} \vee x_4)$. Dann ist

$$\begin{aligned}
c^{(1)} &= 0 * 0\, 0 *^{\ell-4}, \\
c^{(2)} &= 0 * 0\, 1 *^{\ell-4}, \\
c^{(3)} &= 0 * 1\, 1 *^{\ell-4}, \\
c^{(4)} &= 1 * 0\, 0 *^{\ell-4}, \\
c^{(5)} &= 1 * 0\, 1 *^{\ell-4}, \\
c^{(6)} &= 1 * 1\, 0 *^{\ell-4}, \\
c^{(7)} &= 1 * 1\, 1 *^{\ell-4}.
\end{aligned}$$

Weiter definieren wir $s^{(k+i)} := 0^L\sharp_2^L 0^L\sharp_2^L \cdots \sharp_2^L 0^L$ und $s^{(2k+i)} := 1^L\sharp_3^L 1^L\sharp_3^L \cdots \sharp_3^L 1^L$ für jedes $i \in [1 : k]$. Wir erhalten somit also $n := 3k$ Sequenzen der Länge jeweils $m := 13L = 13(\ell + 2)$. Offensichtlich lässt sich die Reduktion in polynomieller Zeit durchführen. Diese Zeichenreihen sind in Matrixschreibweise noch einmal in Abbildung 5.3 illustriert.

Zuerst zeigen wir, dass es eine optimales Lösung für dieses Problem mit $s \in \{0, 1\}^L$ gibt. Sei also s eine optimale Lösung. Nehmen wir als erstes an, s enthalte das Symbol $\sharp_3$. Da $\sharp_3$ nur in den Zeichenreihen $s^{(2k+1)}, \ldots, s^{(3k)}$ vorkommt, können auch

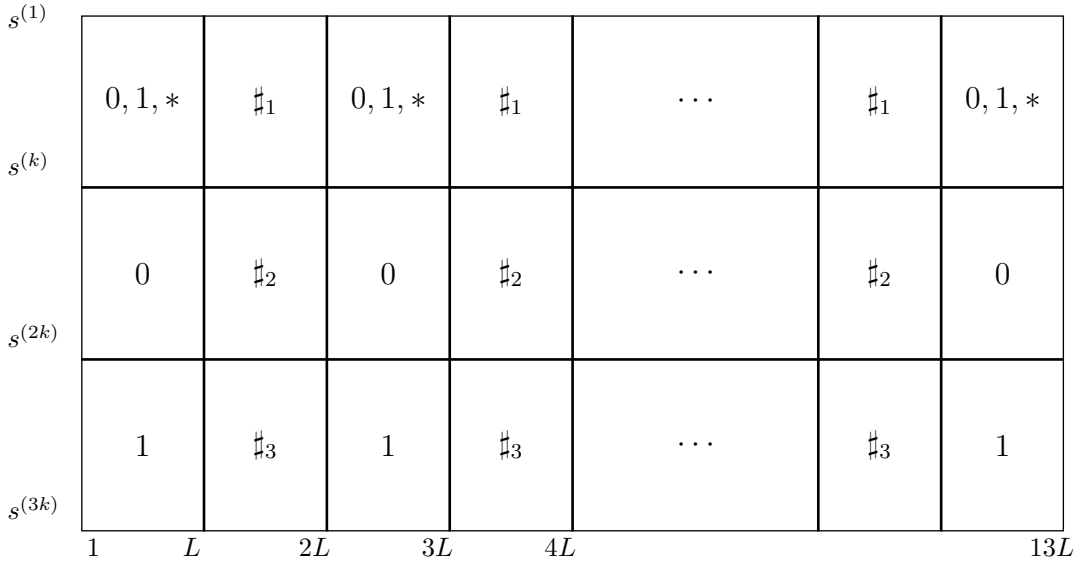


Abbildung 5.3: Skizze: Die Struktur der Sequenzen $s^{(1)}, \ldots, s^{(3k)}$

nur dort Matches vorkommen. Sei also s' die Zeichenreihe, die aus s entsteht, wenn alle $\sharp_3$ durch 1 ersetzt werden. Dann gilt $\sum_{i=1}^{2k} d_H(s', t^{(i)}) \leq \sum_{i=1}^{2k} d_H(s, t^{(i)})$. Wenn wir jetzt $t^{(i)} := s_1^{(i)} \cdots s_L^{(i)}$ für $i \in [2k+1 : 3k]$ setzen, kann sich die Hamming-Distanz von s' zu diesem Teilwörtern nicht erhöhen und wir erhalten eine optimale Lösung s' ohne das Symbol $\sharp_3$ und mit den festen Zeichenreihen $t^{(i)} := s_1^{(i)} \cdots s_L^{(i)}$ für $i \in [2k+1 : 3k]$.

Analog können wir annehmen, dass es eine optimale Lösung s ohne das Symbol $\sharp_2$ gibt. Also wissen wir jetzt, dass es eine optimale Lösung $s \in \{0, 1, *, \sharp_1\}^L$ gibt, wobei $t^{(i)} = s_1^{(i)} \cdots s_L^{(i)}$ für $i \in [k+1 : 3k]$ gilt.

Wir zeigen nun, dass es eine optimale Lösung gibt, die kein $*$ beinhaltet. Wir können das Symbol $*$ in einer optimalen Lösung s einfach durch 0 ersetzen (und erhalten die Zeichenreihe s'). Da das Symbol $*$ in der optimalen Lösung s nur mit maximal k Sternen in $t^{(1)}, \ldots, t^{(k)}$ übereinstimmen kann, kommen also maximal k Mismatches in der Hamming-Distanz für die Lösung s' hinzu. Diese werden aber durch k Matches von s' mit $t^{(k+1)}, \ldots, t^{(2k)}$ wieder ausgleichen, da dort die Mismatches $(*, 0)$ durch Matches $(0, 0)$ ersetzt werden.

Mit der gleichen Argumentation kann man die verbleibenden Trennzeichen $\sharp_1$ in einer optimalen Lösung s durch 0 ersetzen (und erhält dann die Zeichenreihe s'). Jedes Trennzeichen $\sharp_1$ in s kann maximal k Matches in $s^{(1)}, \ldots, s^{(k)}$ besitzen. Durch die Substitution werden diese zu Mismatches, hierfür werden aber die Mismatches des Symbols $\sharp_1$ von s mit 0 in den Teilwörtern $t^{(k+1)}, \ldots, t^{(2k)}$ zu Matches mit s', so dass die gesamte Hamming-Distanz unverändert bleibt.

Also halten wir fest, dass es eine optimale Lösung mit $s \in \{0,1\}^L$ gibt (mit zugehörigen $t^{(i)}$ für $i \in [1:k]$). Als Schlussbemerkung halten wir fest, dass einige der Modifikationen die gesamte Hamming-Distanz möglicherweise hätte erniedrigen können. Da wir aber von einer optimalen Lösung ausgegangen sind, kann dies nicht passieren.

Sei nun $s = s_0 \cdots s_{\ell+1} \in \{0,1\}^L$ für eine erfüllende Belegung B von F, d.h. es gilt $I_B(F) = 1$, wie folgt gewählt: $s_i := B(x_i)$ für $i \in [1:\ell]$ und $s_0 := s_{\ell+1} := 0$. Dann lässt sich leicht nachrechnen, dass gilt:

$$\sum_{i=1}^{3k} d_H(s, t^{(i)}) = kL + k(L-5) = 2kL - 5k,$$

wenn man für jedes $i \in [1:k]$ die Teilwörter $t^{(i)} = 0c^{(p_i)}0$ von $s^{(i)}$ so wählt, dass $d_H(0c^{(p_i)}0, s) = L - 5$ (mit einem geeigneten $p_i \in [1:7]$), und $t^{(i)} = s_1^{(i)} \cdots s_L^{(i)}$ für $i \in [k+1:3k]$. Da B eine erfüllende Belegung ist, ist dies möglich. Der erste Summand stellt die Hamming-Distanz von s zu den 0en bzw. 1en in $t^{(k+1)}$ mit $t^{(3k)}$ dar. Der zweite Summand den Abstand von den Sternen in $t^{(1)}$ mit $t^{(k)}$ zu s, da im $(0,1,*)$-Block je Zeile jeweils genau 3 0en und 1en der Belegung auftreten sowie die jeweils führende und letzte 0 (und damit insgesamt genau $kL - 5k$ Sterne).

Man überlegt sich weiter leicht, dass immer Folgendes gelten muss.

$$\sum_{i=1}^{3k} d_H(s, t^{(i)}) \geq 2kL - 5k =: C.$$

Gibt es keine erfüllende Belegung von F, so muss

$$\sum_{i=1}^{3k} d_H(s, t^{(i)}) > 2kL - 5k$$

gelten. Für eine optimale Lösung $s = s_0 \cdots s_{\ell+1} \in \{0,1\}^L$ können nie fünf 0en und 1en in jedem Teilwort von $s^{(i)}$ für $i \in [1:k]$ übereinstimmen, wenn s eine nichterfüllende Belegung darstellt, d.h. $I_B(F) = 0$ für $B(x_i) := s_i$.

Liefert also ConsPat eine Lösung mit einer Güte von C oder besser (also genau gleich C), so ist F erfüllbar, ansonsten nicht. ■

Wir wollen jetzt noch ein polynomielles Approximationsschema für MinConsPat in Abbildung 5.4 angeben.

31.10.19

Betrachten wir zuerst die Laufzeit des polynomiellen Approximationsschemas.

Lemma 5.48 *Die Laufzeit des Algorithmus aus Abbildung 5.4 hat eine Laufzeit von* $O((nm)^{r+1}L)$.

PTAS_MCP $(s^{(1)}, \ldots, s^{(n)} \in \Sigma^m, r, L \in \mathbb{N})$

```
begin
    forall (1 ≤ i_1 < ··· < i_r ≤ n) do
        forall ((u^(1), ..., u^(r)) ∈ (Σ^L)^r s.t. u^(j) ⊑ s^(i_j)) do
            let u be the columnwise majority-string of (u^(1), ..., u^(r));
            for (k := 1; k ≤ n; k++) do
                let t_u^(k) ∈ Σ^L s.t. t_u^(k) ⊑ s^(k) and d_H(t_u^(k), u) is minimal;
            let c(u) := Σ_{k=1}^{n} d_H(u, t^(k));
    output u, t_u^(1), ..., t_u^(n) s.t. c(u) is minimal;
end
```

Abbildung 5.4: Algorithmus: polynomielles Approximationschema für MinConsPat

Beweis: Die Bestimmung des spaltenweisen Majority-Strings in $(u^{(1)}, \ldots, u^{(r)})$ lässt sich in Zeit $O(Lr)$ erledigen. Die Bestimmung der $t^{(i)}$ kann als Sequenzen-Alignment mit der Hamming-Distanz interpretiert werden und benötigt Zeit $O(nLm)$. Die Bestimmung von $c(u)$ geht offensichtlich in Zeit $O(nL)$. Mit der Berücksichtigung von $r \leq n$ benötigt ein Schleifendurchlauf also Zeit $O(nmL)$. Die äußeren Schleifen werden $(nm)^r$ Mal durchlaufen, also beträgt die Laufzeit insgesamt $O((nm)^{r+1}L)$. ■

Da nur Eingaben mit $L \leq m$ sinnvoll sind, ist die Eingabegröße $\Theta(nm)$ und es handelt sich somit um einen polynomiellen Algorithmus.

Mit der Behauptung des folgenden Lemmas beweisen wir die Approximationsgüte von PTAS_MCP.

Lemma 5.49 *Der Algorithmus PTAS_MCP ist ein polynomielles Approximationsschema für* MinConsPat*, d.h. für jedes $r \geq 3$ konstruiert der Algorithmus eine zulässige Lösung, die höchstens um den Faktor*

$$1 + O\left(\sqrt{\frac{\log(r)}{r}}\right)$$

schlechter als die optimale Lösung ist, in Zeit $O((nm)^{r+1}L)$.

Beweis: Im Folgenden sei s^* und $t^{(1)}, \ldots, t^{(n)}$ eine optimale Lösung der Eingabe $(s^{(1)}, \ldots, s^{(n)}, L)$ mit Güte c^*. Dann können wir ohne Beschränkung der Allgemeinheit annehmen, dass:

- $t^{(i)} \in \Sigma^L$ und $t^{(i)} \sqsubseteq s^{(i)}$, wobei $d_H(s^*, t^{(i)})$ minimal ist.

- $s^* \in \Sigma^L$ ist der spaltenweise Majority-String von $t^{(1)}, \ldots, t^{(n)}$.

Weiter sei $1 \leq i_1 < i_2 < \cdots < i_r \leq n$ und sei $s(i_1, \ldots, i_r) \in \Sigma^L$ der spaltenweise Majority-String von $t^{(i_1)}, \ldots, t^{(i_r)}$. Es gilt also

$$\begin{aligned} c^* &= \sum_{i=1}^{n} d_H(s^*, t^{(i)}), \\ c(i_1, \ldots, i_r) &:= \sum_{i=1}^{n} d_H(s(i_1, \ldots, i_r), t^{(i)}) \geq c^*. \end{aligned}$$

Wir nehmen nun an, dass $i_1, \ldots, i_r$ unabhängig gezogene Zahlen aus dem Intervall $[1:n]$ ohne Zurücklegen sind. Wir werden zeigen, dass

$$\mathbb{E}(c(i_1, \ldots, i_r)) \leq \left(1 + O\left(\sqrt{\frac{\log(r)}{r}}\right)\right) \cdot c^*.$$

Wir setzen jetzt $\rho := 1 + O(\sqrt{\log(r)/r})$, wobei die explizite Konstante in der O-Notation versteckt ist (der genaue Wert ergibt sich aus einer Inspektion des Beweises). Dann existiert eine Wahl von $(i_1, \ldots, i_r)$ mit $c(i_1, \ldots, i_r) \leq \rho \cdot c^*$. Da wir im Algorithmus alle möglichen $(i_1, \ldots, i_r)$ ausprobieren, müssen wir ein Tupel erwischen, dessen Maß mindestens so gut wie der des Erwartungswertes ist.

Für $a \in \Sigma$ sei $h_j(a) = \#\left\{i \in [1:n] \,:\, t_j^{(i)} = a\right\}$. Dann ist

$$c^* = \sum_{i=1}^{n} d_H(t^{(i)}, s^*) = \sum_{j=1}^{L} (n - h_j(s_j^*)).$$

Mit derselben Überlegung und der Linearität des Erwartungswertes gilt:

$$\begin{aligned} \mathbb{E}(c(i_1, \ldots, i_r)) &= \mathbb{E}\left(\sum_{i=1}^{n} d_H(t^{(i)}, s(i_1, \ldots, i_r))\right) \\ &= \mathbb{E}\left(\sum_{j=1}^{L} (n - h_j(s(i_1, \ldots, i_r)_j))\right) \\ &= \sum_{j=1}^{L} \mathbb{E}\left(n - h_j(s(i_1, \ldots, i_r)_j)\right). \end{aligned}$$

Es genügt also zu zeigen, dass für allr $j \in [1:L]$

$$\mathbb{E}\left(n - h_j(s(i_1, \ldots, i_r)_j)\right) \leq \rho(n - h_j(s_j^*)).$$

Ziehen wir von beiden Seite $n - h_j(s_j^*)$ ab, ist also nur noch zu zeigen, dass

$$\mathbb{E}\left(h_j(s_j^*) - h_j(s(i_1,\ldots,i_r)_j)\right) \leq O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot \left(n - h_j(s_j^*)\right).$$

Dies beweisen wir im Lemma 5.52 und der Beweis dieses Satzes ist beendet. ■

Definition 5.50 *Eine Zufallsvariable X heißt* binäre Zufallsvariable *bzw.* reelle Zufallsvariable*, wenn X nur Werte aus $\{0,1\}$ bzw. aus $\mathbb{R}$ annehmen kann.*

Damit können wir den folgende Satz formulieren.

Theorem 5.51 (Chernoff-Schranken) *Seien $X_1,\ldots,X_n$ n unabhängige binäre Zufallsvariablen mit $\mathrm{Ws}(X_i = 1) = p_i$. Sei $X = \sum_{i=1}^n X_i$ eine reelle Zufallsvariable, dann gilt für $\delta > 0$:*

$$\begin{aligned}
\mathrm{Ws}[X \geq (1+\delta)\cdot\mathbb{E}(X)] &\leq \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mathbb{E}(X)} \leq \exp\left(-\frac{\mathbb{E}(X)\cdot\delta^2}{3}\right), \\
\mathrm{Ws}[X \leq (1-\delta)\cdot\mathbb{E}(X)] &\leq \left[\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right]^{\mathbb{E}(X)} \leq \exp\left(-\frac{\mathbb{E}(X)\cdot\delta^2}{2}\right).
\end{aligned}$$

Der Beweis der Chernoff-Schranken wird normalerweise in einer Einführungsvorlesung zu Stochastik und Statistik ausgeführt, so dass wir den Beweis hier auslassen wollen und den Leser auf die Literatur (z.B. von Angelika Steger) verweisen.

Lemma 5.52 *Seien $a_1,\ldots,a_n \in \Sigma$. Sei weiter $h(a) = \#\{i \in [1:n] : a_i = a\}$ für jedes $a \in \Sigma$. Seien $i_1,\ldots,i_r$ unabhängig zufällig gezogenen Zahlen aus $[1:n]$. Sei a^r der Majority-Letter von $a_{i_1},\ldots,a_{i_r}$ und a^* der Majority-Letter von $a_1,\ldots,a_n$. Für $r \geq 3$ gilt:*

$$\mathbb{E}\left(h(a^*) - h(a^r)\right) \leq O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot \left(n - h(a^*)\right).$$

Beweis: Bezeichne $m(a) = \#\left\{j \in [1:r] : a_{i_j} = a\right\}$ und sei $p_a = \frac{h(a)}{n}$. Dann ist $\mathrm{Ws}(a_{i_j} = a) = p_a$. Wir unterscheiden nun zwei Fälle, je nachdem, wie groß $h(a^*)$ ist.

Fall 1 ($h(a^*) \leq \frac{5}{6}n$): Mit Hilfe der Chernoff-Schranken erhalten wir zunächst für alle $\varepsilon > 0$ und alle $a \in \Sigma$:

$$\begin{aligned}
\mathrm{Ws}[m(a) \geq rp_a + \varepsilon r] &= \mathrm{Ws}[m(a) \geq (1 + \varepsilon/p_a) rp_a] \\
&\leq \exp\left(-\frac{rp_a(\frac{\varepsilon}{p_a})^2}{3}\right) \\
&= \exp\left(-\frac{r\varepsilon^2/p_a}{3}\right) \\
&\leq \exp\left(-\frac{r\varepsilon^2}{3}\right)
\end{aligned}$$

sowie

$$\begin{aligned}
\mathrm{Ws}[m(a) \leq rp_a - \varepsilon r] &= \mathrm{Ws}[m(a) \leq (1 - \varepsilon/p_a) rp_a] \\
&\leq \exp\left(-\frac{rp_a(\frac{\varepsilon}{p_a})^2}{2}\right) \\
&= \exp\left(-\frac{r\varepsilon^2/p_a}{2}\right) \\
&\leq \exp\left(-\frac{r\varepsilon^2}{2}\right) \\
&\leq \exp\left(-\frac{r\varepsilon^2}{3}\right).
\end{aligned}$$

Sei nun $\delta := \sqrt{3\log(r)/r}$. Wir setzen dann $\Sigma_1 = \{a \in \Sigma : h(a) \geq h(a^*) - 2\delta n\}$ und $\Sigma_2 = \Sigma \setminus \Sigma_1$. Dann gilt für $a \in \Sigma_2$:

$$p_a + \delta \quad = \quad \frac{h(a)}{n} + \delta \quad < \quad \frac{h(a^*) - 2\delta n}{n} + \delta \quad = \quad p_{a^*} - \delta.$$

Dann gilt:

05.11.19

$$\begin{aligned}
&\mathbb{E}[h(a^*) - h(a^r)] \\
&\quad = \sum_{a \in \Sigma} \mathrm{Ws}[a^r = a](h(a^*) - h(a)) \\
&\quad = \sum_{a \in \Sigma_1} \mathrm{Ws}[a^r = a](h(a^*) - h(a)) + \sum_{a \in \Sigma_2} \mathrm{Ws}[a^r = a](h(a^*) - h(a)) \\
&\quad \leq \sum_{a \in \Sigma_1} \mathrm{Ws}[a^r = a] \cdot 2\delta n + \sum_{a \in \Sigma_2} \mathrm{Ws}[a^r = a] \cdot n
\end{aligned}$$

Wenn a der Majority-Letter von $(a_{i_1}, \ldots, a_{i_r})$ ist, muss $m(a) \geq m(a^*)$ gelten. Folglich muss also $\text{Ws}[a^r = a] \leq \text{Ws}[m(a) \geq m(a^*)]$ gelten.

$$
\begin{aligned}
&\leq 2\delta n + \sum_{a \in \Sigma_2} \text{Ws}[m(a) \geq m(a^*)] \cdot n \\
&\quad \text{da } p_a + \delta < p_{a^*} - \delta \\
&\leq 2\delta n + n \sum_{a \in \Sigma_2} \text{Ws}[(m(a) \geq (p_a + \delta)r) \vee (m(a^*) \leq (p_{a^*} - \delta)r)] \\
&\leq 2\delta n + n \sum_{a \in \Sigma_2} (\text{Ws}[m(a) \geq (p_a + \delta)r] + \text{Ws}[m(a^*) \leq (p_{a^*} - \delta)r]) \\
&\leq 2\delta n + n|\Sigma| \left(\exp\left(-\frac{\delta^2 r}{3}\right) + \exp\left(-\frac{\delta^2 r}{3}\right)\right) \\
&\leq 2\delta n + 2n|\Sigma| \exp\left(-\frac{\delta^2 r}{3}\right) \\
&\quad \text{da } \delta^2 r/3 = \log(r) \\
&= 2\delta n + 2n|\Sigma| \cdot r^{-1} \\
&= O\left(\sqrt{\frac{\log(r)}{r}} + \frac{1}{r}\right) \cdot n \\
&\quad \text{da } r^{-1} = o\left(\sqrt{\log(r)/r}\right) \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot n \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot (n - h(a^*)) \cdot \frac{n}{n - h(a^*)} \\
&\quad \text{da } h(a^*) \leq \tfrac{5}{6}n, \text{ gilt } \tfrac{n}{n-h(a^*)} \leq 6 \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot (n - h(a^*)).
\end{aligned}
$$

Damit ist in diesem Fall die Behauptung bewiesen.

Fall 2 ($h(a^*) > \frac{5}{6}n$): Wir betrachten nun die Zufallsvariable $r - m(a^*)$, die als eine Summe von r unabhängigen binären Zufallsvariablen betrachtet werden kann, wobei der Wert 1 jeder binären Zufallsvariablen mit Wahrscheinlichkeit $1 - p_{a^*}$ angenommen wird. Mithilfe der Chernoff-Schranken erhalten wir für $\beta > 0$:

$$
\text{Ws}[r - m(a^*) \geq (1 + \beta)((1 - p_{a^*})r)] \leq \left(\frac{e^\beta}{(1+\beta)^{1+\beta}}\right)^{(1-p_{a^*})r}.
$$

Sei nun $x := \frac{n-h(a^*)}{n} = 1 - p_{a^*} \in [0, \frac{1}{6})$ (da $h(a^*) > \frac{5}{6}n$), dann gilt:

$$\mathrm{Ws}[r - m(a^*) \geq (1+\beta)(xr)] \leq \left(\frac{e^\beta}{(1+\beta)^{1+\beta}}\right)^{xr}.$$

Wir setzen nun $\beta := \frac{1}{2x} - 1 \geq 2$, d.h. es gilt $1 + \beta = \frac{1}{2x}$, und erhalten:

$$\begin{aligned}
\mathrm{Ws}[r - m(a^*) \geq r/2] &\leq \left(\frac{e^{\frac{1}{2x}-1}}{(\frac{1}{2x})^{\frac{1}{2x}}}\right)^{xr} \\
&= \left(\frac{(2ex)^{\frac{1}{2x}}}{e}\right)^{xr} \\
&= \left(\frac{\sqrt{2ex}}{e^x}\right)^{r} \\
&\quad \text{mit } x \geq 0 \text{ ist } e^x \geq 1 \\
&\leq \left(\sqrt{2ex}\right)^{r} \\
&= \left(\sqrt{2ex}\right)^{r-2} \cdot 2ex
\end{aligned}$$

Da $h(a^*) > \frac{5}{6}n$, gilt $x = \frac{n-h(a^*)}{n} < \frac{1}{6}$ und somit $\sqrt{2ex} \leq \sqrt{e/3}$.Damit gilt:

$$\begin{aligned}
&\leq (\sqrt{e/3})^{r-2} \cdot 2ex \\
&= \left(\frac{3}{e}\right)^{-\frac{r-2}{2}} \cdot 2ex \\
&\quad \text{da } e < 3 \text{ und somit } (3/e)^{-(r-2)/2} = o\left(\sqrt{\log(r)/r}\right) \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot 2ex \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot x.
\end{aligned}$$

Insgesamt erhalten wir also:

$$\begin{aligned}
\mathbb{E}(h(a^*) - h(a^r)) &= \sum_{a \in \Sigma} \mathrm{Ws}[a^r = a] \cdot (h(a^*) - h(a)) \\
&= \sum_{\substack{a \in \Sigma \\ a \neq a^*}} \mathrm{Ws}[a^r = a] \cdot (h(a^*) - h(a)) \\
&\leq \sum_{\substack{a \in \Sigma \\ a \neq a^*}} \mathrm{Ws}[a^r = a] \cdot n
\end{aligned}$$

$$
\begin{aligned}
&\leq n \cdot \sum_{\substack{a \in \Sigma \\ a \neq a^*}} \mathrm{Ws}[a^r = a] \\
&= n \cdot \mathrm{Ws}[a^r \neq a^*] \\
&\quad \text{da } a^r \neq a^* \Rightarrow m(a^*) \leq r/2 \\
&\leq n \cdot \mathrm{Ws}[m(a^*) \leq r/2] \\
&= n \cdot \mathrm{Ws}[r - m(a^*) \geq r/2] \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot xn \\
&\quad \text{da } x = \tfrac{n - h(a^*)}{n} \\
&= O\left(\sqrt{\frac{\log(r)}{r}}\right) \cdot (n - h(a^*)).
\end{aligned}
$$

Aus beiden Fällen folgt nun die Behauptung des Lemmas. ■

Eine genaue Inspektion des Beweises zeigt, dass die Konstante in der O-Notation im Wesentlichen $12|\Sigma|$ ist.

Theorem 5.53 *Es gibt ein polynomielles Approximationsschema für* MinConsPat *mit Laufzeit* $O((mn)^{2(\varepsilon/c)^{-2}\log((\varepsilon/c)^{-2})+1}L)$ *für ein geeignetes* $c > 0$.

Beweis: Wir setzen $r = 2(\varepsilon/c)^{-2}\log((\varepsilon/c)^{-2})$. Dann gilt

$$
\begin{aligned}
c \cdot \sqrt{\frac{\log(r)}{r}} &= c \cdot \sqrt{\frac{\log((\varepsilon/c)^{-2}) + \log(2\log((\varepsilon/c)^{-2}))}{2(\varepsilon/c)^{-2}\log((\varepsilon/c)^{-2})}} \\
&= c \cdot \sqrt{\frac{1 + \frac{\log(2\log((\varepsilon/c)^{-2}))}{\log((\varepsilon/c)^{-2})}}{2(\varepsilon/c)^{-2}}}
\end{aligned}
$$

Da $\frac{\log(2\log((\varepsilon/c)^{-2}))}{\log((\varepsilon/c)^{-2})} \leq 1$, weil $\frac{\log(2x)}{x} < 1$ für $x > 0$ und $\log((\varepsilon/c)^{-2}) > 0$ für $\varepsilon > 0$.

$$
\begin{aligned}
&\leq c \cdot \sqrt{\frac{\varepsilon^2}{c^2}} \\
&= \varepsilon.
\end{aligned}
$$

Die Behauptung folgt nun aus den vorherigen Lemmata. ■

Damit erhalten wir das folgende Korollar.

Korollar 5.54 *Es gilt* MinConsPat $\in \mathcal{PTAS}$.

5.3.4 Die Klasse $\mathcal{FPTAS}$

Der Nachteil eines polynomiellen Approximationsschemas ist, dass die Laufzeit polynomiell in der Eingabe, aber nicht unbedingt in der Approximationsgüte ist. Für die Praxis wäre es schön, wenn die Laufzeit solcher $(1+\varepsilon)$-Approximationen auch polynomiell in ε^{-1} wäre. Bislang haben wir auch Laufzeiten zugelassen, die z.B. exponentiell in ε^{-1} sind, wie das polynomielle Approximationsschema für MinConsPat (siehe Theorem 5.53). Approximationsschemata, deren Laufzeit exponentiell in ε^{-1} ist, sind in der Praxis oft schon für $\varepsilon \approx 20\%$ ungeeignet. Das motiviert die folgende Definition von echt polynomiellen Approximationsschemata (engl. fully polynomial time approximation scheme).

Definition 5.55 *Sei $P = (I, S, \mu, \text{opt}) \in \mathcal{NPO}$ ein Optimierungsproblem. Ein Algorithmus A ist ein* echt polynomielles Approximationsschema*, wenn es ein Polynom p gibt, so dass für jedes $\varepsilon > 0$ und jede Eingabe $x \in I$ gilt, dass $A(\varepsilon, x) \in S(x)$ sowie:*

$$\Gamma_{A(\varepsilon)}(x) \leq 1 + \varepsilon \qquad \text{und} \qquad T_{A(\varepsilon)}(x) \leq p(\|x\| + \varepsilon^{-1}).$$

Ein Optimierungsproblem gehört zur Klasse $\mathcal{FPTAS}$, wenn es ein echt polynomielles Approximationsschema besitzt.

Aus der Definition folgt wieder unmittelbar, dass $\mathcal{PO} \subseteq \mathcal{FPTAS} \subseteq \mathcal{PTAS}$ gilt. Wir wollen nun ein Kriterium angeben, das uns hilft festzustellen, ob ein Optimierungsproblem ein echt polynomielles Approximationsschema besitzen kann oder nicht.

Definition 5.56 *Ein Optimierungsproblem $P = (I, S, \mu, \text{opt})$ heißt* polynomiell beschränkt*, wenn es ein Polynom q gibt, so dass für jedes $x \in I$ und für jedes $y \in S(x)$ gilt: $\mu(x, y) \in \mathbb{N}$ und $\mu(x, y) \leq q(\|x\|)$.*

Wir können nun zeigen, dass jedes $\mathcal{NP}$-harte, polynomiell beschränkte Optimierungsproblem kein echt polynomielles Approximationsschema besitzen kann.

Theorem 5.57 *Ist $\mathcal{P} \neq \mathcal{NP}$, dann kann kein $\mathcal{NP}$-hartes, polynomiell beschränktes Optimierungsproblem zur Klasse $\mathcal{FPTAS}$ gehören.*

Beweis: Sei $P = (I, S, \mu, \max)$ ein $\mathcal{NP}$-hartes, polynomiell beschränktes Optimierungsproblem. Wir nehmen hier ohne Beschränkung der Allgemeinheit an, dass es sich um ein Maximierungsproblem handelt. Der Beweis für ein Minimierungsproblem verläuft analog. Nehmen wir an, es gäbe ein echt polynomielles Approximationsschema A für P. Dann gibt es ein Polynom p, so dass die Laufzeit von $A(\varepsilon)$

durch $p(\|x\| + \varepsilon^{-1})$ beschränkt ist, wobei ε sich auf die Approximationsgüte bezieht. Da P polynomiell beschränkt ist, gibt es ein weiteres Polynom q, so dass für jede Eingabe $x \in I$ gilt: $\mu^*(x) \leq q(\|x\|)$. Wählen wir nun $\varepsilon := 1/q(\|x\|)$, dann liefert $A(\varepsilon)$ für jedes x eine $(1 + 1/q(\|x\|))$-Approximation der optimalen Lösung, d.h. es gilt:

$$\frac{\mu^*(x)}{\mu(x, A(\varepsilon, x))} \leq 1 + \varepsilon = 1 + \frac{1}{q(\|x\|)} = \frac{q(\|x\|) + 1}{q(\|x\|)}.$$

Damit erhalten wir sofort:

$$\begin{aligned}
\mu(x, A(\varepsilon, x)) &\geq \mu^*(x) \frac{q(\|x\|)}{q(\|x\|) + 1} \\
&= \mu^*(x) \frac{q(\|x\|) + 1 - 1}{q(\|x\|) + 1} \\
&= \mu^*(x) - \frac{\mu^*(x)}{q(\|x\|) + 1} \\
&\quad \text{da } \mu^*(x) \leq q(\|x\|) \\
&\geq \mu^*(x) - \frac{q(\|x\|)}{q(\|x\|) + 1} \\
&> \mu^*(x) - 1.
\end{aligned}$$

Da $\mu^*(x)$ nach Definition ganzzahlig ist, muss $\mu(x, A(\varepsilon, x)) = \mu^*(x)$ sein und $A(\varepsilon)$ liefert somit eine optimale Lösung.

Mit $\varepsilon^{-1} = q(\|x\|)$ ist die Laufzeit des Algorithmus dann $p(\|x\| + q(\|x\|))$. Da das Polynom eines Polynoms wiederum ein Polynom ist, bleibt die Laufzeit des Algorithmus polynomiell in $\|x\|$. Damit hätten wir für ein $\mathcal{NP}$-hartes Problem einen polynomiellen Algorithmus gefunden, was unmittelbar $\mathcal{P} = \mathcal{NP}$ impliziert. ■

Mit Hilfe dieses Satzes können wir nun zeigen, dass $\mathcal{FPTAS}$ echt in $\mathcal{PTAS}$ enthalten ist.

Korollar 5.58 *Es gilt* MinConsPat $\notin \mathcal{FPTAS}$*, außer wenn* $\mathcal{P} = \mathcal{NP}$*.*

Beweis: Sei MinConsPat $= (I, S, \mu, \max)$, dann ist für $x \in I$ und $y \in S(x)$ offensichtlich $\mu(x, y) \leq L \cdot n \leq mn \leq \|x\|$. Damit ist MinConsPat ein polynomiell beschränktes Optimierungsproblem. Da das zu MinConsPat gehörende Entscheidungsproblem $\mathcal{NP}$-vollständig ist, ist MinConsPat $\mathcal{NP}$-hart. Mit Hilfe des vorherigen Satzes folgt dann die Behauptung. ■

Korollar 5.59 *Es gilt* $\mathcal{FPTAS} \subsetneq \mathcal{PTAS}$*, außer wenn* $\mathcal{P} = \mathcal{NP}$*.*

Zum Abschluss wollen wir für ein $\mathcal{NP}$-hartes Optimierungsproblem noch ein echt polynomielles Approximationsschema konstruieren. Dazu betrachten wir das Packen eine Rucksacks.

MaxKnapsack

Eingabe: Eine Folge $((s_1, p_1), \ldots, (s_n, p_n)) \in (\mathbb{N} \times \mathbb{N})^n$ und $C \in \mathbb{N}$.
Lösung: Eine Teilmenge $I \subseteq [1:n]$ mit $\sum_{i \in I} s_i \leq C$
Optimum: Maximiere $\sum_{i \in I} p_i$.

Anschaulich haben wir n Objekte für eine Wanderung zur Verfügung. Jedes Objekt hat ein Gewicht s_i und einen Profit p_i. Außerdem darf der Rucksack nur mit einem maximalen Gewicht von C bepackt werden. Wir wollen nun den Profit der mitgenommenen Objekte bei Einhaltung der Gewichtsschranke maximieren. Wir betrachten zuerst das zu MaxKnapsack gehörige Entscheidungsproblem:

07.11.19

Knapsack

Eingabe: Eine Folge $((s_1, p_1), \ldots, (s_n, p_n)) \in (\mathbb{N} \times \mathbb{N})^n$ und $B, C \in \mathbb{N}$.
Gesucht: Gibt es eine Teilmenge $I \subseteq [1:n]$ mit $\sum_{i \in I} s_i \leq C$ und $\sum_{i \in I} p_i \geq B$?

Lemma 5.60 Knapsack *ist $\mathcal{NP}$-vollständig.*

Beweis: Offensichtlich ist Knapsack in $\mathcal{NP}$ enthalten. Wir reduzieren jetzt Partition polynomiell auf Knapsack. Da Partition $\mathcal{NP}$-vollständig ist, folgt dann die Behauptung. Sei $x = (s_1, \ldots, s_n) \in \mathbb{N}^n$ eine Eingabe für Partition. Dann ist $x' = (((s_1, s_1), \ldots, (s_n, s_n)), B, C)$ mit $B := C := \frac{1}{2} \sum_{i=1}^{n} s_i$ eine Eingabe für Knapsack. Man sieht leicht, dass Partition für x genau dann lösbar ist, wenn Knapsack für x' lösbar ist. ∎

Zur Berechnung einer optimalen Lösung von MaxKnapsack verwenden wir wieder einmal die Dynamische Programmierung. Sei dazu $S(k, p)$ das minimale Gewicht und höchstens C, um den Rucksack mit Profit p zu packen, wenn nur die ersten k Objekte zur Auswahl zur Verfügung stehen. Dann gilt:

$$S(k,p) = \begin{cases} 0 & \text{falls } k = 0 \text{ und } p = 0, \\ \infty & \text{falls } k = 0 \text{ und } p \neq 0, \\ S(k-1, p-p_k) + s_k & \text{falls } k \geq 1 \text{ und } p - p_k \geq 0 \\ & \text{und } S(k-1, p-p_k) < \infty \\ & \text{und } S(k-1, p-p_k) + s_k \leq C \\ & \text{und } S(k-1, p-p_k) + s_k \leq S(k-1, p), \\ S(k-1, p) & \text{sonst.} \end{cases}$$

Die Rekursion ergibt sich einfach aus der Tatsache, dass wir entweder das k-te Objekt hinzunehmen oder nicht. Wir nehmen es genau dann hinzu, wenn das Gewicht bei vorgegebenen Profit minimal wird. Sei also $\Pi(k,p)$ eine solche Packung mit Gewicht $S(k,p)$, dann erhalten wir analog:

$$\Pi(k,p) = \begin{cases} \emptyset & \text{falls } k = 0, \\ \Pi(k-1, p-p_k) \cup \{k\} & \text{falls } k \geq 1 \text{ und } p - p_k \geq 0 \\ & \text{und } S(k-1,p-p_k) < \infty \\ & \text{und } S(k-1,p-p_k) + s_k \leq C \\ & \text{und } S(k-1,p-p_k) + s_k \leq S(k-1,p), \\ \Pi(k-1,p) & \text{sonst.} \end{cases}$$

Die beste Packung erhalten wir als $\Pi(n,p^*)$, wobei $p^* \in [1:P]$ mit $P := \sum_{i=1}^{n} p_i$ der maximale Index ist, so dass $S(n,p^*) < \infty$ und somit auch $S(n,p^*) \leq C$. Damit erhalten wir unmittelbar das folgende Lemma.

Lemma 5.61 MaxKnapsack *kann in Zeit* $O(n \cdot P)$ *mit* $P = \sum_{i=1}^{n} p_i$ *optimal gelöst werden.*

Beweis: Die Werte von S und Π lassen sich mit zwei Schleifen über $k \in [1:n]$ und $p \in [1:P]$ berechnen. Dabei speichern wir in $\Pi(k,p)$ nicht explizit ab, sondern nur, ob $k \in \Pi(k,p)$ ist oder nicht. Die restlichen Elemente lassen sich dann rekursiv aus Π ermitteln. Der Zeitbedarf ist also $O(n \cdot P)$. Der maximale Nutzen einer Packung des Rucksacks lässt sich aus dem maximalen Wert p^* mit $S(n,p^*) \leq C$ ablesen. ■

Damit haben wir anscheinend einen polynomiellen Algorithmus für MaxKnapsack gefunden. Aber Moment, MaxKnapsack ist doch ein $\mathcal{NP}$-hartes Problem! Haben wir also eben gerade $\mathcal{P} = \mathcal{NP}$ bewiesen? Die Laufzeit ist zwar polynomiell, allerdings ist sie polynomiell in P. Für polynomielle Algorithmen fordern wir, dass sie in der Eingabegröße polynomiell sind. Leider lässt sich der Wert von P mit logarithmisch vielen Bits repräsentieren. Der oben angegebene Algorithmus ist in Wirklichkeit exponentiell in der Eingabegröße, wie beim Problem UCMP. Für kleine Werte von P (etwa wenn P polynomiell in n ist) haben wir allerdings wirklich einen polynomiellen Algorithmus gefunden. Aus diesem Grund nennt man solche Algorithmen oft auch pseudo-polynomiell.

Definition 5.62 *Ein Algorithmus hat* pseudo-polynomielle Laufzeit*, wenn die Laufzeit polynomiell in der Eingabegröße und in der größten auftretenden Zahl in der Eingabe ist.*

Wir versuchen nun mit Hilfe dieses pseudo-polynomiellen Algorithmus ein echt polynomielles Approximationsschema für MaxKnapsack zu konstruieren. Ist der Profit der einzelnen Objekte klein, so haben wir bereits einen optimalen Algorithmus gefunden. Wir müssen uns nur noch überlegen, wie wir mit dem Problem umgehen, wenn die Profite groß werden. Die Hauptidee ist, dass wir den Profit auf kleinere Werte abrunden. Wenn wir dafür mit unserem pseudo-polynomiellen Algorithmus eine optimale Lösung finden, können wir hoffen, dass diese Lösung eine näherungsweise Lösung für das ursprüngliche Optimierungsproblem ist.

Wir konstruieren eine neue Eingabe für MaxKnapsack wie folgt. Sei dazu

$$x = ((s_1, p_1), \ldots, (s_n, p_n), C)$$

die ursprüngliche Eingabe, dann definieren wir eine neue Eingabe

$$x' = ((s_1, p_1'), \ldots, (s_n, p_n'), C) \quad \text{mit} \quad p_i' := \lfloor p_i / 2^t \rfloor,$$

wobei wir t später noch genauer spezifizieren werden. Anschaulich haben wir von jedem p_i die letzten t Bits abgeschnitten.

Wir überlegen uns nun, inwieweit sich die optimale Lösung von x und die Lösung unterscheiden, die man aus einer optimalen Lösung von x' erhält. Seien dazu $\Pi^*(x)$ bzw. $\Pi^*(x')$ die optimalen Lösungen von x bzw. x'. Der Algorithmus A konstruiert zuerst aus ε das zugehörige t (wie genau erläutern wir später) sowie x' aus x, berechnet dann die optimale Lösung für x' und gibt diese als Lösung für x aus. Nach Konstruktion unseres Algorithmus A erhalten wir:

$$\begin{aligned}
\mu(x, A(\varepsilon, x)) &= \sum_{i \in \Pi^*(x')} p_i \\
&\quad \text{da } p_i \geq 2^t \cdot p_i' \\
&\geq 2^t \sum_{i \in \Pi^*(x')} p_i' \\
&\quad \text{da } \Pi^*(x') \text{ eine optimale Lösung für } x' \text{ ist} \\
&\geq 2^t \sum_{i \in \Pi^*(x)} p_i' \\
&\quad \text{da } p_i' = \lfloor p_i / 2^t \rfloor \\
&= 2^t \sum_{i \in \Pi^*(x)} \lfloor p_i / 2^t \rfloor \\
&\geq 2^t \sum_{i \in \Pi^*(x)} (p_i / 2^t - 1)
\end{aligned}$$

$$\begin{aligned} &\geq \sum_{i \in \Pi^*(x)} p_i - 2^t \cdot |\Pi^*(x)| \\ &\quad \text{da } |\Pi^*(x)| \leq n \text{ und } \Pi^*(x) \text{ eine optimale Lösung für } x \text{ ist} \\ &\geq \mu^*(x) - n \cdot 2^t. \end{aligned}$$

Also gilt $0 \leq \mu^*(x) - \mu(x, A(\varepsilon, x)) \leq n \cdot 2^t$. Für unsere Approximationsgüte gilt dann

$$\begin{aligned} \Gamma_{A(\varepsilon)}(x) &= \frac{\mu^*(x)}{\mu(x, A(\varepsilon, x))} \\ &= \frac{\mu^*(x) - \mu(x, A(\varepsilon, x)) + \mu(x, A(\varepsilon, x))}{\mu(x, A(\varepsilon, x))} \\ &\leq 1 + \frac{n \cdot 2^t}{\mu(x, A(\varepsilon, x))} \\ &\quad \text{da } \mu(x, A(\varepsilon, x)) \geq \mu^*(x) - n \cdot 2^t > 0 \text{ (für } > 0 \text{ siehe unten)} \\ &\leq 1 + \frac{n \cdot 2^t}{\mu^*(x) - n \cdot 2^t} \\ &\leq 1 + \frac{1}{\frac{\mu^*(x)}{n \cdot 2^t} - 1}. \end{aligned}$$

Damit die obige Abschätzung gilt, muss t so gewählt werden, dass $\mu^*(x) - n \cdot 2^t > 0$ bzw. $\frac{\mu^*(x)}{n \cdot 2^t} > 1$ ist. Weiter sollte $\frac{\mu^*(x)}{n \cdot 2^t} - 1 \approx \varepsilon^{-1}$ sein, damit wir ein Approximationsschema erhalten.

Wir wählen daher

$$t = \left\lfloor \log \left(\frac{\varepsilon}{1+\varepsilon} \cdot \frac{p_{max}}{n} \right) \right\rfloor,$$

wobei $p_{max} := \max \{p_i : i \in [1:n]\}$. Dann gilt

$$\frac{\mu^*(x)}{n \cdot 2^t} \geq \frac{p_{max}}{n \cdot \left(\frac{\varepsilon}{1+\varepsilon} \cdot \frac{p_{max}}{n}\right)} = \frac{1+\varepsilon}{\varepsilon} > 1.$$

Dabei haben wir ausgenutzt, dass wir ohne Beschränkung der Allgemeinheit annehmen können, dass $p_{max} \leq \mu^*(x)$. Dies kann nur dann nicht der Fall sein, wenn das zugehörige Gewicht größer als C ist, d.h. $s_{max} > C$ ist. Dies können wir aber sehr einfach verhindern, indem wir alle Paare (s_i, p_i) mit $s_i > C$ aus der Eingabe entfernen. Solche Paare sind sowieso irrelevant, da sie nicht in einer optimalen Lösung auftauchen können. Insgesamt erhalten wir dann also für die Güte unseres Approximationsalgorithmus:

$$\Gamma_{A(\varepsilon)}(x) \leq 1 + \frac{1}{\frac{\mu^*(x)}{n \cdot 2^t} - 1} \leq 1 + \frac{1}{\frac{1+\varepsilon}{\varepsilon} - 1} = 1 + \frac{1}{\frac{1+\varepsilon-\varepsilon}{\varepsilon}} = 1 + \varepsilon.$$

Damit haben wir nun ein Approximationsschema konstruiert. Wir müssen nun noch nachweisen, dass es sich um ein echt polynomielles Approximationsschema handelt. Da wir den optimalen Algorithmus für die modifizierte Eingabe x' laufen lassen und die Lösung einfach übernehmen sowie die Konstruktion von x' aus x in Zeit $O(n)$ erledigt werden kann, beträgt die Laufzeit des Algorithmus dann

$$\begin{aligned} O\left(n + n \cdot \sum_{i=1}^{n} p'_i\right) &= O\left(n \cdot \sum_{i=1}^{n} \frac{p_i}{2^t}\right) \\ &= O\left(n \cdot (n \cdot p_{max}) \cdot \frac{1+\varepsilon}{\varepsilon} \cdot \frac{n}{p_{max}}\right) \\ &= O\left(n^3 \cdot \frac{1+\varepsilon}{\varepsilon}\right) \\ &= O\left(n^3 \cdot \varepsilon^{-1}\right) \\ &= O\left((n+\varepsilon^{-1})^4\right). \end{aligned}$$

Also ist die Laufzeit des Algorithmus sowohl in n als auch in ε^{-1} polynomiell und wir halten das Ergebnis im folgenden Satz fest.

Theorem 5.63 *Es gibt für* MaxKnapsack *ein echt polynomielles Approximationsschema mit Laufzeit* $O(n^3 \cdot \varepsilon^{-1}) = O((n+\varepsilon^{-1})^4)$.

Damit erhalten wir unmittelbar das folgende Korollar.

Korollar 5.64 *Es gilt* MaxKnapsack $\in \mathcal{FPTAS}$.

Wir halten abschließend noch fest, dass $\mathcal{PO}$ echt in $\mathcal{FPTAS}$ enthalten ist.

Lemma 5.65 *Es gilt* $\mathcal{PO} \subsetneq \mathcal{FPTAS}$, *außer wenn* $\mathcal{P} = \mathcal{NP}$.

Beweis: Da wir eben gesehen haben, dass MaxKnapsack in $\mathcal{FPTAS}$ ist und da Knapsack ein $\mathcal{NP}$-hartes Problem ist, würde aus $\mathcal{FPTAS} = \mathcal{PO}$ folgen, dass $\mathcal{P} = \mathcal{NP}$ ist. ■

Die Hierarchie von Komplexitätsklassen, die zu den verschiedenen Begriffen der Approximierbarkeit korrespondieren, ist im Bild 5.5 schematisch dargestellt.

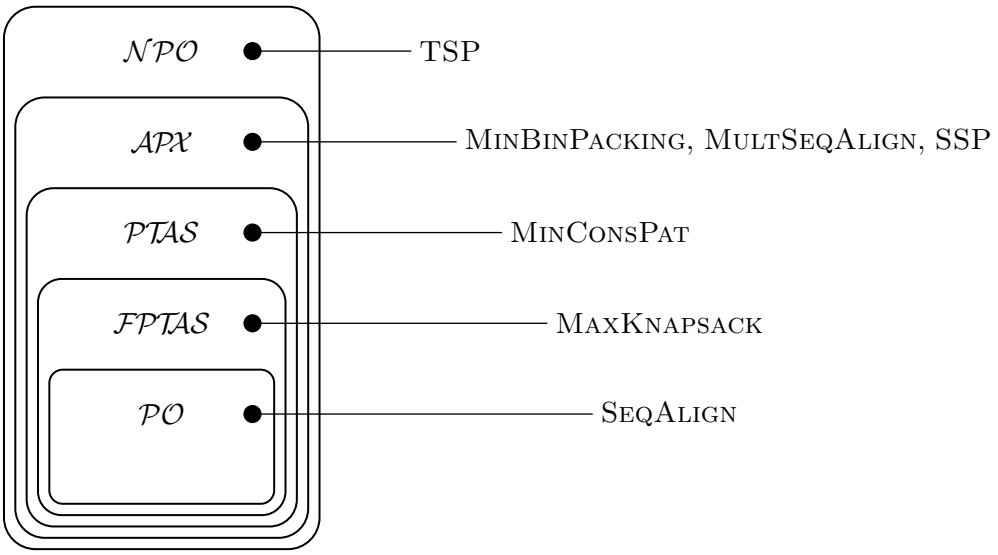


Abbildung 5.5: Die Hierarchie zwischen $\mathcal{PO}$ und $\mathcal{NPO}$

5.3.5 Approximationserhaltende Reduktionen

Wie in der Klasse $\mathcal{NP}$ möchte man auch in den Klassen $\mathcal{NPO}$, $\mathcal{APX}$ oder $\mathcal{PTAS}$ die schwierigsten Probleme als harte bzw. vollständige Probleme klassifizieren. Dazu benötigen wir jedoch einen angepassten Begriff der Reduktion, da sich Approximationsgüten möglichst gut übertragen lassen sollen. Hierfür definieren wir die so genannte PTAS-Reduktion, die nicht nur die Lösung, sondern auch die Approximationsgüte erhalten müssen.

12.11.19

Definition 5.66 *Ein Tripel (f, g, α) heißt* PTAS-Reduktion *eines Optimierungsproblems $A = (I, S, \mu, \text{opt})$ auf ein Optimierungsproblem $B = (I', S', \mu', \text{opt}')$ (abgekürzt $A \leq_{\text{PTAS}} B$), wenn folgende Bedingungen erfüllt sind:*

- *$f : I \times \mathbb{Q}_+^* \to I'$, wobei $f(x, \varepsilon) \in I'$ für alle $x \in I$ und $\varepsilon \in \mathbb{Q}_+^*$, $S'(f(x, \varepsilon)) \neq \emptyset$ aus $S(x) \neq \emptyset$ folgt und f in polynomieller Zeit in $\|x\|$ berechenbar ist.*
- *$g : I \times S'(I') \times \mathbb{Q}_+^* \to S(I)$, wobei $g(x, y', \varepsilon) \in S(x)$ für alle $x \in I$, $\varepsilon \in \mathbb{Q}_+^*$ und $y' \in S'(f(x, \varepsilon))$ und g in polynomieller Zeit in $\|x\| + \|y'\|$ berechenbar ist.*
- *$\alpha : \mathbb{Q}_+^* \to \mathbb{Q}_+^*$ ist eine invertierbare, in polynomieller Zeit berechenbare Funktion.*
- *Für alle $x \in I$, $\varepsilon \in \mathbb{Q}_+^*$ und $y' \in S'(f(x, \varepsilon))$ mit $\Gamma_\mu(\bar{x}, \bar{y}) := \max\left\{\frac{\mu(\bar{x},\bar{y})}{\mu^*(\bar{x})}, \frac{\mu^*(\bar{x})}{\mu(\bar{x},\bar{y})}\right\}$ gilt: Ist $\Gamma_{\mu'}(f(x, \varepsilon), y') \leq 1 + \alpha(\varepsilon)$, dann ist $\Gamma_\mu(x, g(x, y', \varepsilon)) \leq 1 + \varepsilon$.*

Die Funktion f hat hier wieder die übliche Aufgabe eine Instanz des Problems A in eine Instanz des Problems B zu konvertieren. Die Funktion g berechnet aus einer Lösung von $f(x, \varepsilon)$ des Problems B eine Lösung der Instanz x des Problems A zurück. Bei beiden Funktion kann dabei die Umrechnung auch von der zu erzielenden Approximationsgüte abhängen. Dass die Güte der mithilfe von g konstruierten Lösung in Relation zur Güte der Lösung von $f(x, \varepsilon)$ steht, wird durch den letzten Punkt der Definition festgelegt. Oft wird für die Funktion $\alpha(x)$ eine lineare Funktion $c \cdot x$ verwendet. In Abbildung 5.6 sind die einzelnen Komponenten der PTAS-Reduktion noch einmal bildlich dargestellt.

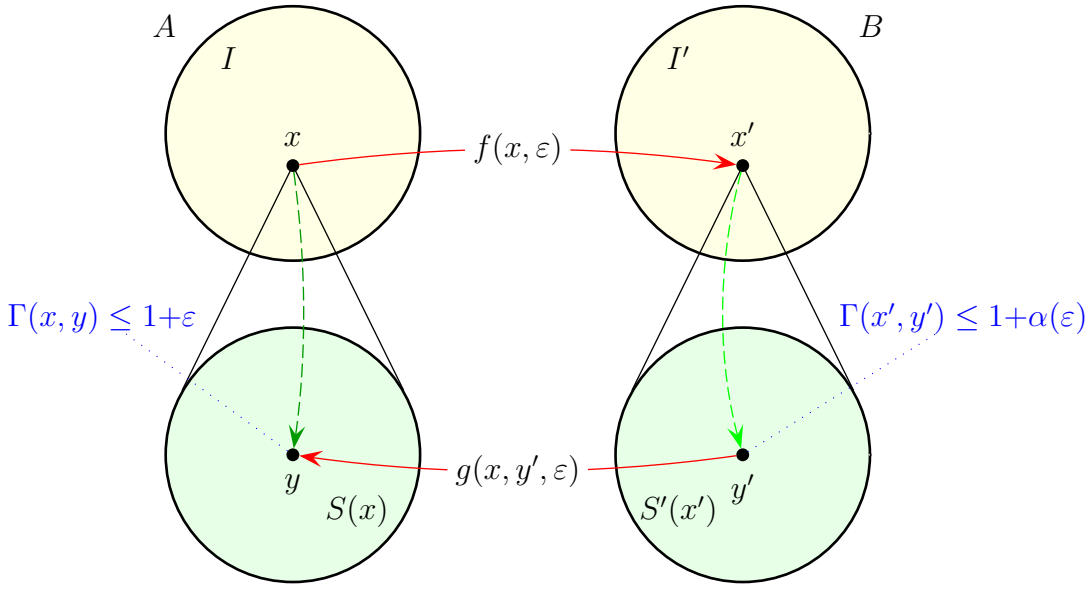


Abbildung 5.6: Skizze: PTAS-Reduktionen

Wir wollen diese Definition an einem Beispiel veranschaulichen.

MaxE-m-Sat (Maximum Exactly m-SAT)

Eingabe: Eine Boolesche Formel in konjunktiver Normalform $F = \bigwedge_{i=1}^{k} C_i$ mit $V(F) = X = \{x_1, \ldots, x_n\}$, wobei jede Klausel C_i genau m paarweise verschiedene Literale enthält.
Lösung: Eine Belegung $B : X \to \mathbb{B}$.
Optimum: Maximiere $\mu(F, B) := |\{i \in [1 : k] : I_B(C_i) = 1\}|$.

Wir geben nun in einem Beispiel eine solche PTAS-Reduktion an.

Theorem 5.67 *Es gilt* MaxE3Sat $\leq_{PTAS}$ MaxE4Sat.

Beweis: Wir zeigen im Folgenden, dass MaxE3Sat $\leq_{PTAS}$ MaxE4Sat. Wir beschreiben zuerst die Transformation f, sei dazu $F = \bigwedge_{i=1}^{k} C_i$ eine Eingabe für MaxE3Sat. Dann ist

$$F' := f(F, \varepsilon) = \bigwedge_{i=1}^{k} ((C_i \vee z_i) \wedge (C_i \vee \overline{z}_i))$$

eine Eingabe für MaxE4Sat, wobei $z_1, \ldots, z_k \notin V(F)$ neue Variablen sind.

Hat man eine Belegung für F, die m Klauseln gleichzeitig erfüllt, so induziert diese eine Belegung für F', die $m+k$ Klauseln gleichzeitig erfüllt (egal wie man die Variablen $z_1, \ldots, z_n$ belegt). Umgekehrt gilt, dass jede Belegung für F' immer mindestens k Klauseln in F' erfüllt (allein die Belegung der z_i setzt immer eine der beiden korrespondierenden Klauseln auf wahr). Hat man eine Belegung für F', die $k+m$ Klauseln gleichzeitig erfüllt, so induziert diese Belegung eine Belegung für F, die m Klauseln gleichzeitig erfüllt. Somit erstellt die Funktion g im Wesentlichen aus einer Belegung für $V(F')$ die zugehörige Restriktion auf $V(F)$. Halten wir das noch formal fest:

$$\exists B : V(F) \to \mathbb{B} : \ \mu(F, B) = m \quad \Leftrightarrow \quad \exists B' : V(F') \to \mathbb{B} : \ \mu(F', B') = m + k.$$

Sei im Folgenden m^* die maximale Anzahl gleichzeitig erfüllbarer Klauseln in F. Dann kann man mittels der obigen Aussage leicht nachprüfen, dass die maximale Anzahl gleichzeitig erfüllbarer Klauseln in F' gerade m^*+k ist. Umgekehrt gilt auch, dass wenn $m^* + k$ die maximale Anzahl gleichzeitig erfüllbarer Klauseln in F' ist, dass m^* die maximale Anzahl gleichzeitig erfüllbarer Klauseln in F ist. Halten wir auch das noch formal fest:

$$\mu^*(F') = \mu^*(F) + k.$$

Für den folgenden Beweis der Erhalt der Approximationsgüte benötigen wir noch einen Fakt, der in den Übungen bewiesen wird. Für jede E3Sat-Formel F kann man immer in polynomieller Zeit eine 8/7-Approximation finden (d.h. 7/8 der Klauseln sind immer gleichzeitig erfüllbar). Es gilt sogar, dass $7k/8$ der Klauseln immer gleichzeitig erfüllbar sind. Halten wir auch dies noch formal fest:

$$\exists \tilde{B} : V(F) \to \mathbb{B} : \ \mu(F, \tilde{B}) \geq \frac{7}{8} \cdot k \geq \frac{7}{8} \cdot \mu^*(F).$$

Damit verwenden wir die folgende Rücktransformation:

$$g(F, B', \varepsilon) = \begin{cases} B'|_{V(F)} & \text{falls } \varepsilon < \frac{1}{3}, \\ \tilde{B} & \text{sonst.} \end{cases}$$

Für die Funktion α wählen wir $\alpha(\varepsilon) = \varepsilon^2$. Man könnte auch $\alpha(\varepsilon) = \varepsilon/3$ wählen (der Leser möge dies bitte selbst verifizieren).

Fall 1 ($\varepsilon \geq 1/3$): Dann liefert die oben genannte 8/7-Approximation für F eine hinreichend gute Lösung, da hier $\frac{8}{7} < \frac{4}{3} \leq 1 + \varepsilon$ gilt.

Fall 2 ($\varepsilon < 1/3 \wedge m \geq k/2$): Nach Voraussetzung gilt mit $\alpha(\varepsilon) = \varepsilon^2$:

$$\frac{m^* + k}{m + k} = \Gamma_\mu(F', B') \leq 1 + \alpha(\varepsilon) = 1 + \varepsilon^2.$$

Dann gilt auch

$$m^* + k \leq (m + k) + (m + k)\varepsilon^2.$$

Nach Subtraktion von k und Division durch m erhalten wir

$$\begin{aligned}
\Gamma_\mu(F, B) &= \frac{m^*}{m} \\
&\leq 1 + \frac{m + k}{m}\varepsilon^2 \\
&= 1 + \left(1 + \frac{k}{m}\right)\varepsilon^2 \\
&\quad \text{da } m \geq k/2 \\
&\leq 1 + \left(1 + \frac{k}{k/2}\right)\varepsilon^2 \\
&= 1 + 3\varepsilon^2 \\
&\quad \text{da } \varepsilon < 1/3 \\
&\leq 1 + \varepsilon.
\end{aligned}$$

Also folgt $\Gamma_\mu(F, B) = \frac{m^*}{m} \leq 1 + \varepsilon$.

Fall 3 ($\varepsilon < 1/3 \wedge m \leq k/2$): Dann gilt aber mit $m^* + k \geq 7k/8 + k$:

$$\begin{aligned}
\Gamma_\mu(F', B') &= \frac{m^* + k}{m + k} \\
&\geq \frac{7k/8 + k}{k/2 + k} \quad = \quad \frac{\frac{15}{8}}{\frac{3}{2}} \\
&= \frac{5}{4} \\
&> \frac{10}{9} \\
&= 1 + \left(\frac{1}{3}\right)^2 \\
&> 1 + \varepsilon^2.
\end{aligned}$$

Somit erfüllt die gegebene Belegung B' für F' nicht die geforderte Approximationsgüte. ■

Auch PTAS-Reduktionen sind transitiv. Den Beweis überlassen wird dem Leser als Übungsaufgabe.

Lemma 5.68 *PTAS-Reduktionen sind transitiv.*

Wie bei Karp-Reduktionen, erhalten wir auch hier die entsprechenden Resultate.

Lemma 5.69 *Wenn $A \leq_{\text{PTAS}} B$ und $B \in \mathcal{PTAS}$ (bzw. $B \in \mathcal{APX}$), dann ist $A \in \mathcal{PTAS}$ (bzw. $A \in \mathcal{APX}$).*

Das vorhergehende Lemma gilt auch für $\mathcal{NPO}$, sofern wir für das Problem A voraussetzen, dass sich das zugehörige Maß μ in polynomieller Zeit berechnen lässt.

5.3.6 Vollständige Probleme

Nun können wir auch für die Approximationsklassen die Vollständigkeit bzgl. der gewählten Reduktion definieren.

Definition 5.70 *Ein Optimierungsproblem P heißt* $\mathcal{NPO}$-vollständig *bzw.* $\mathcal{APX}$-vollständig *(bzgl. der PTAS-Reduktion), wenn $P \in \mathcal{NPO}$ bzw. $P \in \mathcal{APX}$ gilt und für jedes Problem $Q \in \mathcal{NPO}$ bzw. $Q \in \mathcal{APX}$ gilt, dass $Q \leq_{\text{PTAS}} P$.*

Wir wollen hier noch erwähnen, dass es noch eine Vielzahl anderer approximationserhaltender Reduktionen (wie L-Reduktion, AP-Reduktion, etc.) gibt. Die PTAS-Reduktion hat sich jedoch als allgemein und leicht handhabbar erwiesen und wird daher heutzutage oft verwendet.

Wie bei der Theorie der $\mathcal{NP}$-Vollständigkeit ist es ein Problem, das erste vollständige Problem aufzufinden. Dazu definieren wir zuerst das folgende Theorem, dessen $\mathcal{NPO}$-Vollständigkeit nachgewiesen worden ist.

MaxWSat (Maximum Weighted Satisfiability)

Eingabe: Eine Boolesche Formel in konjunktiver Normalform $F = \bigwedge_{i=1}^{k} C_i$ über $X = \{x_1, \ldots, x_n\}$ und Variablengewichten $\gamma_i \in \mathbb{N}$ für $i \in [1:n]$.
Lösung: Eine Belegung $B : X \to \mathbb{B}$ mit $I_B(F) = 1$.
Optimum: Maximiere $\mu(B) = \sum_{i=1}^{n} B(x_i) \cdot \gamma_i$.

Theorem 5.71 MaxWSat *ist $\mathcal{NPO}$-vollständig.*

Im Wesentlichen folgt der Beweis dem Beweis des Satzes von Cook-Levin. Eine geeignete nichtdeterministische Turingmaschine (oder Turingmaschine) rät zu einer Instanz eine Lösung, berechnet die Güte der Lösung (und verifiziert diese dabei auch) in polynomieller Zeit und schreibt dann Lösung und Güte auf die Ausgabe. Alle Variablen erhalten Gewicht 0, nur die Variablen die die Zellen beschreiben, in denen die Güte steht werden so mit Gewichte versehen, dass die Güte der Belegung der Güte der geratenen Lösung entspricht.

Für einen Beweis dieses Theorems verweisen wir auf die einschlägigen Lehrbücher oder eine Vorlesung zur Komplexitätstheorie. Für die Klasse $\mathcal{APX}$ konnten mit klassischen Beweismitteln keine Vollständigkeitsresultate gezeigt werden. Dies wurde erst mit der Kenntnis des so genannten PCP-Theorems möglich.

5.3.7 PCP-Theorem

Um das PCP-Theorem genauer erläutern zu können, müssen wir erst definieren, was wir unter einem randomisierten Verifizierer verstehen wollen.

Definition 5.72 *Ein* randomisierter Verifizierer *V ist eine polynomiell zeitbeschränkte Turingmaschine, die Zugriff auf die Eingabe x, ein Zertifikat z und ein zufälligen Bitstring τ besitzt. Aufgrund der Eingabe x und des zufälligen Bitsrings τ berechnet V eine Menge I von Positionen des Zertifikats z, die V liest und entscheidet in Abhängigkeit von x und $\{z_i : i \in I\}$, ob er die Eingabe x akzeptiert oder nicht. Das Ergebnis wird mit $V(x, \tau, z) \in \mathbb{B}$ bezeichnet.*

Definition 5.73 *Ein randomisierter Verifizierer heißt* (r, q)-beschränkt*, wenn er nur die ersten $O(r(\|x\|))$ Zufallsbits von τ liest und maximal $O(q(\|x\|))$ Positionen des Zertifikats inspiziert.*

Basierend auf (r, q)-beschränkten randomisierten Verifizierern können wir nun die Klasse PCP definieren.

Definition 5.74 *Ein Entscheidungsproblem P gehört zur Klasse $\mathcal{PCP}(r, q)$, wenn es einen (r, q)-beschränkten randomisierten Verifizierer V gibt, für den gilt:*

- *Für alle x mit $P(x) = 1$ existiert ein Zertifikat z_x mit $\mathrm{Ws}_\tau[V(x, \tau, z_x) = 1] = 1$.*
- *Für alle x mit $P(x) = 0$ und für alle Zertifikate z gilt $\mathrm{Ws}_\tau[V(x, \tau, z) = 1] < \frac{1}{2}$.*

Hierbei ist mit Ws_τ die Wahrscheinlichkeit gemeint, wenn alle zufälligen Bitstrings τ gleichwahrscheinlich sind.

Das fundamentale PCP-Theorem beschreibt, dass man die Klasse $\mathcal{NP}$ auch durch eine der PCP-Klassen charakterisieren kann, wobei nur logarithmisch viele Zufallsbits benötigt werden und nur konstant viele Bits des Zertifikats gelesen werden müssen. Dies ist auf den ersten Blick ein sehr verblüffendes Ergebnis, insbesondere wenn nach aktuellem Stand der Forschung 5 Anfragen ausreichend sind.

14.11.19

Theorem 5.75 (PCP-Theorem) $\mathcal{NP} = \mathcal{PCP}(\log(n), 1)$.

5.3.8 Ein $\mathcal{APX}$-vollständiges Problem (+)

Mit Hilfe des PCP-Theorems konnte dann gezeigt werden, dass Max3Sat ein $\mathcal{APX}$-vollständiges Problem ist.

Max3Sat

Eingabe: Eine Boolesche Formel F in 3-konjunktiver Normalform.
Lösung: Eine Belegung $B : X \to \mathbb{B}$.
Optimum: Maximiere $\mu_F(B)$, wobei $\mu_F(B)$ die Anzahl gleichzeitig erfüllter Klauseln in F unter der Belegung B ist.

Folgender Satz zeigt, dass man jedes Minimierungsproblem auf ein Maximierungsproblem PTAS-reduzieren kann. Diese PTAS-Reduktion ist nicht ganz so einfach, wie sie auf den ersten Blick scheint, da man ja die Approximationsgüten mehr oder weniger erhalten muss.

Theorem 5.76 *Für jedes Minimierungsproblem $P \in \mathcal{APX}$ existiert ein Maximierungsproblem $P' \in \mathcal{APX}$ mit $P \leq_{\text{PTAS}} P'$.*

Bevor zum Beweis der $\mathcal{APX}$-Vollständigkeit von Max3Sat kommen, benötigen wir noch einige Definition und Hilfsergebnisse.

ε-Rob3Sat

Eingabe: Eine Boolesche Formel $F = \bigwedge_{i=1}^{k} C_i$ über X in 3-konjunktiver Normalform. Wenn F nicht erfüllbar ist, dann gilt für jede Belegung $B : X \to \mathbb{B}$: $\sum_{i=1}^{k} I_B(C_i) \leq \varepsilon \cdot k$.
Gesucht: Existiert eine Belegung $B : X \to \mathbb{B}$ mit $I_B(F) = 1$.

Beachte, dass man für ein beliebige Formel vermutlich nicht effizient entscheiden kann, ob die Eingabe für Rob3Sat zulässig ist.

Theorem 5.77 *Sei $L \in \mathcal{NP}$, dann existiert ein $\varepsilon > 0$ und eine Karp-Reduktion ρ mit $L \leq_p^\rho \varepsilon$-Rob3Sat mit den folgenden Eigenschaften.*

- *Ist $x \notin L$, dann sind in $\rho(x)$ maximal $(1-\varepsilon) \cdot k(\rho(x))$ Klauseln gleichzeitig erfüllbar.*
- *Für eine Belegung B von Variablen in $\rho(x)$, die mehr als $(1-\varepsilon) \cdot k(\rho(x))$ Klauseln in $\rho(x)$ erfüllt, kann eine erfüllende Belegung in polynomieller Zeit berechnet werden.*

Hierbei ist $k(F)$ die Anzahl von Klauseln einer Booleschen Formel F in konjunktiver Normalform.

Beweis: Aufgrund des PCP-Theorems gilt $\mathcal{NP} = \mathcal{PCP}(\log n, 1)$. Damit existiert ein randomisierter Verifizierer V für L, der $r(n) = O(\log(n))$ Zufallsbits liest und $q = O(1)$ Anfragen an das Zertifikat stellt. Weiterhin gilt:

- Für jedes $x \in L$ existiert ein Zertifikat $z \in \{0,1\}^*$, so dass $V(x,z,\tau) = 1$ für alle Bitstrings $\tau \in \{0,1\}^{r(n)}$ gilt.
- Für jedes $x \notin L$ und alle Zertifikate z gilt: $\mathrm{Ws}_\tau[V(x,z,\tau) = 1] \leq 1/2$.

Für die Konstruktion von ρ müssen wir also für eine Eingabe x eine Boolesche Formel $F(x)$ konstruieren, die die Eigenschaften des Satzes erfüllt.

Sei also $x \in L$. Für ein festes, aber beliebiges $\tau \in \{0,1\}^{r(n)}$ simulieren wir V auf der Eingabe x mit dem Zufallsstring τ bis der Verifizierer das Zertifikat z an genau q Stellen $(i_1(x,\tau), \ldots, i_q(x,\tau))$ inspiziert. Da v nicht adaptiv ist, hängen die inspizierten Positionen nur von der Eingabe x und dem Zufallsstring τ (aber nicht von bereits inspizierten Stellen des Zertifikats) ab. Weiterhin hängt dann nach Definition des Verifizierer die Antwort nur von den inspizierten Stellen des Zertifikats ab, also von $(z_{i_1(x,\tau)}, \ldots, z_{i_q(x,\tau)})$. Die Antwort $V(x,\tau,z)$ lässt sich als Boolesche Formel $F'_{x,\tau}(z)$ in den Variablen $(z_{i_1(x,\tau)}, \ldots, z_{i_q(x,\tau)})$ beschreiben. Da diese nur von q Variablen abhängt, kann die konjunktive Normalform hiervon mit weniger als 2^q Klauseln mit je q Literalen beschrieben werden. Ein Umwandlung in eine 3-konjunktive Normalform $F_{x,\tau}(z)$ vergrößert die Anzahl Klauseln um den Faktor q. Damit besteht $F_{x,\tau(x)}(z)$ aus $q \cdot 2^q$ Klauseln. Für eine Eingabe x definieren wir nun die Boolesche Formel $F_x(z)$ wie folgt:

$$F_x(z) \quad := \quad \bigwedge_{\tau \in \{0,1\}^{r(n)}} F_{x,\tau}(z).$$

$F_x(z)$ besteht also aus genau $q \cdot 2^q \cdot 2^{r(n)}$ Klauseln. Da $q = O(1)$ und $r(n) = O(\log(n))$ gilt, sind dies polynomiell viele Klauseln.

Ist $x \in L$, dann ist nach Definition $F_x(z)$ erfüllbar. Das aus der Definition für $x \in L$ zugehörige Zertifikat z stellt dann eine erfüllende Belegung dar. Ist $x \notin L$, dann ist nach Definition von $L \in \mathcal{PCP}(\log(n), 1)$ unabhängig vom betrachteten Zertifikat mindestens die Hälfte der Formeln $F_{x,\tau}(z)$ ist nicht erfüllbar, d.h. mindestens $\frac{1}{2} \cdot 2^{r(n)}$ Klauseln sind nicht erfüllbar. Damit ist der Anteil nicht gleichzeitig erfüllbarer Klauseln mindestens

$$\frac{\frac{1}{2} \cdot 2^{r(n)}}{q \cdot 2^q \cdot 2^{r(n)}} = \frac{1}{q \cdot 2^{q+1}} =: \varepsilon.$$

Die zweite Eigenschaft der Behauptung wollen wir hier nicht beweisen, da dessen Beweis eng mit dem Beweis des PCP-Theorems zusammenhängt. ■

Theorem 5.78 Max3Sat*ist $\mathcal{APX}$-vollständig.*

Beweisidee: Wir werden nur eine Skizze dieses aufwendigeren Beweises angeben. Sei $P = (I, S, \mu, \max)$ ein Maximierungsproblem aus $\mathcal{APX}$ und sei A ein Approximationsalgorithmus für P mit Güte r. Wir müssen also im Folgenden zeigen, dass $P \leq_{PTAS}$ Max3Sat gilt. Sei weiter δ die Konstante aus δ-Rob3Sat, wenn wir im Satz 5.77 $L =$ Max3Sat wählen. Im Folgenden sei $\bar{\mu}$ das Maß von Max3Sat, also die Anzahl erfüllbarer Klauseln.

Wir unterscheiden zwei Fälle bei unserer Reduktion. Gilt $r \leq 1 + \varepsilon$, so brauchen wir keine echte Reduktion ausführen. In der Funktion g simulieren wir die Berechnung von A und erhalten eine Approximation mit Güte $r \leq 1 + \alpha(\varepsilon)$. Dies wird jedoch nur für große Werte ε funktionieren.

Sei also nun $r > 1 + \varepsilon$. Der Approximationsalgorithmus A auf der Eingabe x liefert nun eine Lösung y mit Maß $M := \mu(x, y)$. Nach Definition der Approximationsgüte gilt $\mu^*(x) \in [M, r \cdot M]$. Wir wählen jetzt $b \in \mathbb{Q}$ und $m \in \mathbb{N}$, so dass $r \in (b^{m-1}, b^m]$ gilt (also $b \approx 1 + \varepsilon$ und $m = \lceil \log_b(r) \rceil$), und zerlegen dieses Intervall

$$[M, r \cdot M] \subseteq [M, b^m \cdot M] = [b^0 \cdot M, b^1 \cdot M] \cup [b^1 \cdot M, b^2 \cdot M] \cup \cdots \cup [b^{m-1} \cdot M, b^m \cdot M].$$

Wir werden im Folgenden versuchen festzustellen, in welchem dieser Intervalle sich die optimale Lösung befindet. Dazu betrachten wir die m zugehörigen Entscheidungsprobleme E_i, wobei für $i \in [0 : m-1]$ genau dann $E_i(x) = 1$ gilt, wenn $\mu^*(x) \geq b^i \cdot \mu(x, y)$ ist. Dabei wurde b und m so gewählt, dass $b \leq 1 + \varepsilon$ ist und dass es sich um eine konstante Anzahl von Intervallen handelt (nur abhängig von ε und r).

Da das ursprüngliche Problem in $\mathcal{APX} \subseteq \mathcal{NPO}$ liegt, gilt für die zugehörigen Entscheidungsprobleme $E_i \in \mathcal{NP}$. Es gibt also jeweils eine $\mathcal{NP}$-Maschine M_i, die das zugehörige Entscheidungsproblem E_i (d.h. $\mu^*(x) \geq b^i \cdot M$) entscheidet. Wie im

Beweis des Satzes von Cook konstruieren wir aus den $\mathcal{NP}$-Maschinen M_i 3SAT-Formeln F_i', die genau dann erfüllbar sind, wenn die $\mathcal{NP}$-Maschine M_i mit Ja antworte. Man beachte, dass hier das zugehörige Zertifikat so gewählt werden kann, dass es eine Lösung des ursprünglichen Entscheidungsproblems E_i kodiert, und auch leicht wieder dekodiert werden kann.

Diese Formeln F_i' in konjunktiver Normalform überführen wir mit dem Satz 5.77 jeweils in eine ROB3SAT-Instanz F_i. Auch hierbei ist zu beachten, dass wir aus einer erfüllenden Belegung von F_i leicht eine erfüllende Belegung von F_i' rekonstruieren können. Ohne Beschränkung der Allgemeinheit nehmen wir dabei an, dass jede Formel F_i aus genau k Klauseln besteht.

Wir setzen jetzt

$$F = \bigwedge_{i=0}^{m-1} F_i.$$

Dann besteht F aus genau km Klauseln. Sei jetzt j der maximale Index, so dass F_j erfüllbar ist (damit auch F_0 mit F_j) und F_{j+1} nicht (damit auch F_{j+1} mit F_{m-1} nicht). Wir betrachten jetzt eine $(1+\alpha(\varepsilon))$-Approximation für F, also eine Belegung B mit $\bar{\mu}^*(F) \leq (1+\alpha(\varepsilon))\bar{\mu}(F,B)$. Hierbei wählen wir $\alpha(x) = \frac{\delta \cdot x}{2\ln(r)}$. Daraus folgt unmittelbar $\bar{\mu}(F,B) \geq \frac{1}{1+\alpha(\varepsilon)}\bar{\mu}^*(F)$ und damit (da $\bar{\mu}^*(F) \leq km$):

$$\bar{\mu}^*(F) - \bar{\mu}(F,B) \leq \left(1 - \frac{1}{1+\alpha(\varepsilon)}\right)\bar{\mu}^*(F) \leq \left(1 - \frac{1}{1+\alpha(\varepsilon)}\right) km. \tag{5.1}$$

Sei nun $r_i := \bar{\mu}^*(F_i)/\bar{\mu}(F_i, B_i)$, wobei B_i die Belegung der Variablen von F_i ist, die sich durch Einschränkung der Belegung der Variablen aus F ergibt (beachte, dass die Variablenmengen der verschiedenen Booleschen Formeln F_i disjunkt sind). Damit gilt auch

$$\bar{\mu}^*(F_i) - \bar{\mu}(F_i, b_i) = (1 - 1/r_i)\bar{\mu}^*(F_i). \tag{5.2}$$

Wir werden jetzt zeigen, dass $r_i < 1/(1-\delta)$ (bzw. $1/r_i > 1-\delta$) für $i \in [0:j]$ gilt. Dann wissen wir, dass unsere Approximation von den erfüllbaren Formeln $F_1, \ldots, F_j$ mehr als den Anteil $(1-\delta)$ der Klauseln erfüllt und wir können nach Lemma 5.77 auch eine erfüllbare Belegung finden. Zunächst gilt für $i \in [0:j]$ (da dann $\bar{\mu}^*(F_i) = k$ gilt):

$$\begin{aligned} \bar{\mu}^*(F) - \bar{\mu}(F,B) &= \sum_{p=0}^{m-1} \bar{\mu}^*(F_p) - \bar{\mu}(F_p, B_p) \\ &\geq \bar{\mu}^*(F_i) - \bar{\mu}(F_i, B_i) \end{aligned}$$

$$\begin{aligned} & \text{mit Gleichung (5.2)} \\ = & \left(1 - \frac{1}{r_i}\right) \bar{\mu}^*(F_i) \\ \geq & \left(1 - \frac{1}{r_i}\right) k. \end{aligned}$$

Zusammen mit Ungleichung (5.1) erhalten wir:

$$\left(1 - \frac{1}{1+\alpha(\varepsilon)}\right) km \geq \bar{\mu}^*(F) - \bar{\mu}(F,B) \geq \left(1 - \frac{1}{r_i}\right) k.$$

Also gilt

$$\left(1 - \frac{1}{1+\alpha(\varepsilon)}\right) m \geq \left(1 - \frac{1}{r_i}\right).$$

Wir müssen also nur noch zeigen, dass $1-(1-1/(1+\alpha(\varepsilon)))m > 1-\delta$ ist, d.h. dass $(1-1/(1+\alpha(\varepsilon)))m < \delta$ gilt. Mit der Wahl von $\alpha(x) = \frac{\delta \cdot x}{2\ln(r)}$ kann dies leicht gezeigt werden. Zunächst gilt für m aufgrund der Wahl von b:

$$\begin{aligned} m &= \lceil \log_b(r) \rceil \\ &< 1 + \frac{\ln(r)}{\ln(b)} \\ &\quad \text{da wir } b \in \mathbb{Q}_+^* \text{ mit } b \in [1+3\varepsilon/4, 1+\varepsilon] \text{ wählen} \\ &\leq 1 + \frac{\ln(r)}{\ln(1+3\varepsilon/4)} \\ &\quad \text{da } \ln(z) > 1 - 1/z \text{ für } z > 1 \text{ und somit } \ln(1+\varepsilon) > \tfrac{\varepsilon}{1+\varepsilon} \text{ für } \varepsilon > 0 \\ &< 1 + \frac{(1+3\varepsilon/4)\ln(r)}{3\varepsilon/4} \\ &= 1 + \frac{(4/3+\varepsilon)\ln(r)}{\varepsilon} \\ &< \frac{2\ln(r)}{\varepsilon}. \end{aligned}$$

Die letzte Ungleichung gilt beispielsweise, wenn $r > e$ und $\varepsilon < 1/3$ ist, wovon wir ohne Beschränkung der Allgemeinheit ausgehen können. Somit gilt:

$$\begin{aligned} \left(1 - \frac{1}{1+\alpha(\varepsilon)}\right) \cdot m &< \left(1 - \frac{1}{1+\frac{\varepsilon\delta}{2\ln(r)}}\right) \frac{2\ln(r)}{\varepsilon} \\ &= \left(1 - \frac{\ln(r)}{\ln(r)+\frac{1}{2}\varepsilon\delta}\right) \frac{2\ln(r)}{\varepsilon} \\ &= \left(\frac{\ln(r)+\frac{1}{2}\varepsilon\delta-\ln(r)}{\ln(r)+\frac{1}{2}\varepsilon\delta}\right) \frac{2\ln(r)}{\varepsilon} \end{aligned}$$

$$\begin{aligned} &= \frac{\delta \ln(r)}{\ln(r) + \frac{1}{2}\varepsilon\delta} \\ &< \frac{\delta \ln(r)}{\ln(r)} \\ &= \delta. \end{aligned}$$

Somit gilt also $r_i < 1/(1-\delta)$.

Nach Satz 5.77 wissen wir, dass wenn die Belegung B_i in der Booleschen Formel F_i mehr als $(1-\delta)k$ Klauseln erfüllt, F_i erfüllbar ist und sich diese Belegung B_i' in polynomieller Zeit berechnen lässt. Rückwärts können wir daraus eine erfüllende Belegung für F_i' berechnen und aus dieser das richtige Zertifikat z_i für die $\mathcal{NP}$-Maschine M_i rekonstruieren. Daraus können wir dann eine Lösung für $\tilde{y} \in S(x)$ für das ursprüngliche Maximierungsproblem P rekonstruieren.

Die so gefundene Lösung hat eine Güte $\mu(x, \tilde{y}) \in [b^j \cdot \mu^*(x), b^{j+1}\mu^*(x)]$. Also ist die Approximationsgüte der so konstruierten Lösung y kleiner gleich b. Da $b \leq 1 + \varepsilon$ gewählt ist, folgt die Behauptung. ■

Damit können wir auch polynomielle Approximationsschemata für $\mathcal{APX}$-harte Probleme ausschließen. Für den Beweis des folgenden Satzes verweisen wir auf die einschlägige Literatur.

Theorem 5.79 *Kein $\mathcal{APX}$-vollständiges Problem besitzt ein polynomielles Approximationsschema, außer wenn $\mathcal{P} = \mathcal{NP}$.*

Mit Hilfe des PCP-Theorems konnten viele Nichtapproximierbarkeitsresulate von Optimierungsproblemen nachgewiesen werden, außer wenn $\mathcal{P} = \mathcal{NP}$. Wir wollen es hier jedoch mit dieser sehr knappen Einführung bewenden lassen und verweisen statt dessen auf die einschlägige Literatur.

Bevor das PCP-Theorem bekannt war, konnte man also keine $\mathcal{APX}$-vollständigen Probleme nachweisen. Früher wurde der Nachweis der Härte von Approximationsproblemen daher mit Hilfe der syntaktisch definierten Klasse *MAX-SNP* geführt. Allein schon die Definition dieser Klasse bzw. Vollständigkeitsbeweise darin waren wesentlich aufwendiger. Es konnte jedoch gezeigt werden, dass der Abschluss von MAX-SNP unter PTAS-Reduktionen gerade die Klasse der Maximierungsprobleme in $\mathcal{APX}$ liefert. Daher können ältere Resultate der MAX-SNP-Härte eines Problems als ein Resultat der $\mathcal{APX}$-Härte desselben Problems interpretiert werden.

5.4 Beispiel: Ein $\mathcal{APX}$-Algorithmus für SSP (*)

Dieser Abschnitt ist nur der Vollständigkeit im Skript enthalten und wurde seit dem Wintersemester 2008/09 nicht mehr behandelt. In diesem Abschnitt wollen wir das so genannte *Shortest Superstring Problem* (*SSP*) und eine algorithmische Lösung hierfür vorstellen. Formal ist das Problem wie folgt definiert.

Shortest Superstring Problem

Eingabe: Sei $S = \{s_1, \ldots, s_k\} \subset \Sigma^*$.
Lösung: Ein $s^* \in \Sigma$, so dass $s_i \sqsubseteq s^*$ für jedes $i \in [1:k]$ gilt.
Optimum: Minimiere $|s^*|$.

Dies ist eine Formalisierung des Fragment Assembly Problems. Allerdings gehen wir hierbei davon aus, dass die Sequenzierung fehlerfrei funktioniert hat. Ansonsten müssten die Fragmente nur sehr ähnlich zu Teilwörtern des Superstrings, aber nicht identisch sein. Ferner nehmen wir an, dass die gefunden Überlappungen auch wirklich echt sind und nicht zufällig sind. Zumindest bei langen Überlappungen kann man davon jedoch mit hoher Wahrscheinlichkeit ausgehen. Bei kurzen Überlappungen (etwa bei 5 Basenpaaren), kann dies jedoch auch rein zufällig sein. Wie wir später sehen werden, werden wir daher auch den längeren Überlappungen ein größeres Vertrauen schenken als den kürzeren.

Obwohl wir hier die Existenz von Fehlern negieren, ist das Problem und dessen Lösung nicht nur von theoretischem Interesse. Auch bei vorhandenen Fehlern wird die zugrunde liegende Lösungsstrategie von allgemeinem Interesse sein, da diese prinzipiell auch beim Vorhandensein von Fehlern angewendet werden kann.

Zuerst die schlechte Nachricht: Das Shortest Superstring Problem ist $\mathcal{NP}$-hart. Wir können also nicht hoffen, dass wir eine optimale Lösung in polynomieller Zeit finden können. Wie schon früher werden wir versuchen, eine möglichst gute Näherungslösung zu finden. Leider ist das SSP sogar $\mathcal{APX}$-hart.

Im Folgenden wollen wir zeigen, dass mithilfe einer Greedy-Strategie eine Lösung gefunden werden kann, die höchstens viermal so lang wie eine optimale Lösung ist. Mithilfe derselben Idee und etwas mehr technischen Aufwand, lässt sich sogar eine 2,5-Approximation finden.

5.4.1 Ein Approximationsalgorithmus

Für die Lösung des SSP wollen wir in Zukunft ohne Beschränkung der Allgemeinheit annehmen, dass kein s_i Teilwort von s_j für $i \neq j$ sei (andernfalls ist s_i ja bereits in einem Superstring für $S \setminus \{s_i\}$ als Teilwort enthalten).

Definition 5.80 *Sei $s, t \in \Sigma^*$ und sei v das längste Wort aus Σ^*, so dass es $u, w \in \Sigma^+$ mit $s = uv$ und $t = vw$ gibt. Dann bezeichne*

- $o(s,t) = v$ *den* Overlap *von s und t,*
- $p(s,t) = u$ *das* Präfix *von s in t.*

In der Abbildung 5.7 ist der Overlap $v := o(s,t)$ von s und t sowie das Präfix $u = p(s,t)$ von s in t noch einmal graphisch dargestellt. Beachte, dass der Overlap

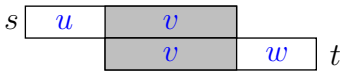


Abbildung 5.7: Skizze: Overlap und Präfix von s und t

ein echtes Teilwort von s und t sein muss. Daher ist der Overlap von $s = aabab$ und $t = abab$ eben $o(s,t) = ab$ und nicht $abab$. Dies spielt hier keine allzu große Rolle, da wir zu Beginn dieses Abschnitts ohne Beschränkung der Allgemeinheit angenommen haben, dass kein Wort Teilwort eines anderen Wortes der gegebenen Menge ist. Dies ist jedoch wichtig, wenn wir den Overlap und das Präfix eines Wortes mit sich selbst berechnen wollen. Beispielsweise ist für $s = aaa$ der Overlap $o(s,s) = aa$ und somit $p(s,s) = a$ sowie für $s' = abbaba$ ist der Overlap sogar das leere Wort: $o(s',s') = \varepsilon$. Wir wollen an dieser Stelle noch die folgende einfache, aber wichtige Beziehung festhalten.

Lemma 5.81 *Sei $s, t \in \Sigma^*$, dann gilt $s = p(s,t) \cdot o(s,t)$.*

In der Abbildung 5.8 ist ein Beispiel zur Illustration der obigen Definitionen anhand von drei Sequenzen angegeben.

Bsp:	s_1=ACACG	$o(s_1, s_1) = \varepsilon$	$p(s_1, s_1)$ =ACACG
	s_2=ACGTT	$o(s_1, s_2)$ =ACG	$p(s_1, s_1)$ =AC
	s_3=GTTA	$o(s_1, s_3)$ =G	$p(s_1, s_3)$ =ACAC
		$o(s_2, s_1) = \varepsilon$	$p(s_2, s_1)$ =ACGTT
		$o(s_2, s_2) = \varepsilon$	$p(s_2, s_2)$ =ACGTT
		$o(s_2, s_3)$ =GTT	$p(s_2, s_3)$ =AC
		$o(s_3, s_1)$ =A	$p(s_3, s_1)$ =GTT
		$o(s_3, s_2)$ =A	$p(s_3, s_2)$ =GTT
		$o(s_3, s_3) = \varepsilon$	$p(s_3, s_3)$ =GTTA

Abbildung 5.8: Beispiel: Overlaps und Präfixe

Lemma 5.82 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$, dann gilt für eine beliebige Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$, dass*

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_2}, s_{i_3}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}$$

ein Superstring von S ist.

Beweis: Wir führen den Beweis mittels Induktion über k.

Induktionsanfang ($k = 1$): Hierfür ist die Aussage trivial, da s_1 offensichtlich s_1 als Teilwort enthält.

Induktionsschritt ($k \to k + 1$): Nach Induktionsvoraussetzung gilt

$$s' = \underbrace{p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k})}_{s''} \cdot s_{i_k},$$

wobei s' ein Superstring für $\{s_{i_1}, \ldots, s_{i_k}\}$ ist.

Nach Lemma 5.81 ist $s_{i_k} = p(s_{i_k}, s_{i_{k+1}}) \cdot o(s_{i_k}, s_{i_{k+1}})$. Daher enthält $p(s_{i_k}, s_{i_{k+1}}) \cdot s_{i_{k+1}}$ sowohl s_{i_k} als auch $s_{i_{k+1}}$, da $o(s_{i_k}, s_{i_{k+1}})$ ein Präfix von $s_{i_{k+1}}$ ist. Dies ist in der Abbildung 5.9 noch einmal graphisch dargestellt. Also ist

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot p(s_{i_k}, s_{i_{k+1}}) \cdot s_{i_{k+1}}$$

ein Superstring für die Zeichenreihen in $S = \{s_1, \ldots, s_{k+1}\}$. ■

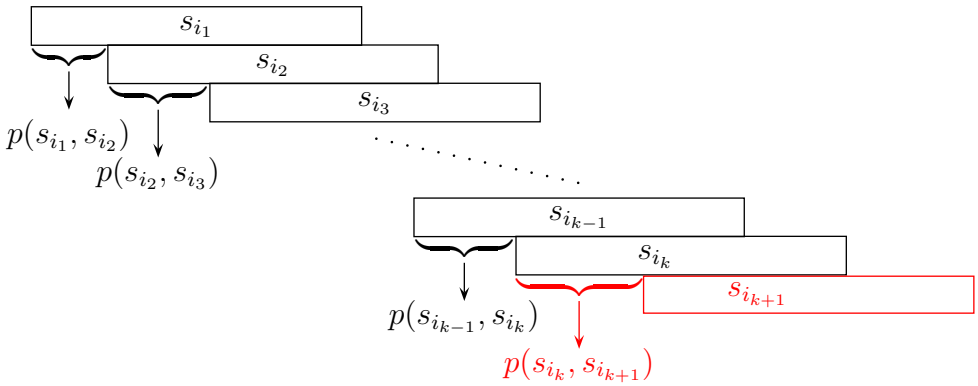


Abbildung 5.9: Skizze: Erweiterung des Superstrings

Korollar 5.83 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$, dann ist für eine beliebige Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ die Zeichenfolge*

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot p(s_{i_k}, s_{i_1}) \cdot o(s_{i_k}, s_{i_1})$$

ein Superstring.

Beweis: Dies folgt aus dem vorhergehenden Lemma und dem Lemma 5.81, das besagt, dass $s_{i_k} = p(s_{i_k}, s_1) \cdot o(s_{i_k}, s_1)$. ■

Lemma 5.84 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und s^* der kürzeste Superstring für S. Dann gibt es eine Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ mit*

$$s^* = p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}.$$

Beweis: Sei s^* ein (kürzester) Superstring für $S = \{s_1, \ldots, s_k\}$. Wir definieren a_i als die kleinste ganze Zahl, so dass $s^*_{a_i} \cdots s^*_{a_i+|s_i|-1} = s_i$ gilt. Umgangssprachlich ist a_i die erste Position, an der s_i als Teilwort von s^* auftritt. Da s^* ein Superstring von $S = \{s_1, \ldots, s_k\}$ ist, sind alle a_i für $i \in [1:k]$ wohldefiniert. Wir merken noch an, dass die a_i paarweise verschieden sind, da wir ja ohne Beschränkung der Allgemeinheit angenommen haben, dass in s kein Wort Teilwort eines anderen Wortes ist.

Sei nun $(i_1, \ldots, i_k) \in S(k)$ eine Permutation über $[1:k]$, so dass

$$a_{i_1} < a_{i_2} < \cdots < a_{i_k}.$$

Dann ist $(i_1, \ldots, i_k)$ die gesuchte Permutation. Dies ist in Abbildung 5.10 noch einmal illustriert. Man beachte, dass $a_{i_1} = 1$ und $a_{i_k} + |s_{i_k}| - 1 = |s^*|$ gilt, da sonst s^* nicht der kürzeste Superstring von S wäre. ■

Korollar 5.85 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und s^* der kürzeste Superstring für S. Dann gibt es eine Permutation der Indizes $(i_1, \ldots, i_k) \in S(k)$ mit*

$$s^* = p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot p(s_{i_k}, s_{i_1}) \cdot o(s_{i_k}, s_{i_1}).$$

Aus den beiden letzten Korollaren folgt, dass den Präfixen von je zwei Zeichenreihen ineinander eine besondere Bedeutung für einen kürzesten Superstring zukommt. Dies motiviert die folgenden Definition eines Präfix-Graphen.

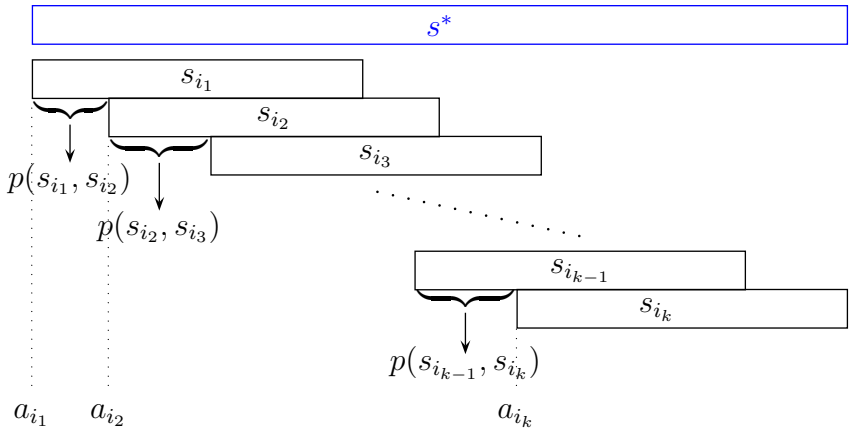


Abbildung 5.10: Skizze: Auffinden der s_i im Superstring

Definition 5.86 *Der gewichtete gerichtete Graph $G_S = (V, E, \gamma)$ für eine Menge von Sequenzen $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ heißt* Präfix-Graph *von S, wobei $V = S$, $E = V \times V = S \times S$ und das Gewicht für $(s,t) \in E$ durch $\gamma(s,t) = |p(s,t)|$ definiert ist.*

In Abbildung 5.11 ist der Präfix-Graphen samt drei Beispielen von Zyklenüberdeckungen für das vorherige Beispiel von Sequenzen dargestellt. Natürlich ist in diesem Beispiel auch (s_1, s_2) und s_3 eine Zyklenüberdeckung mit einem Gewicht von $2 + 3 + 4 = 9$.

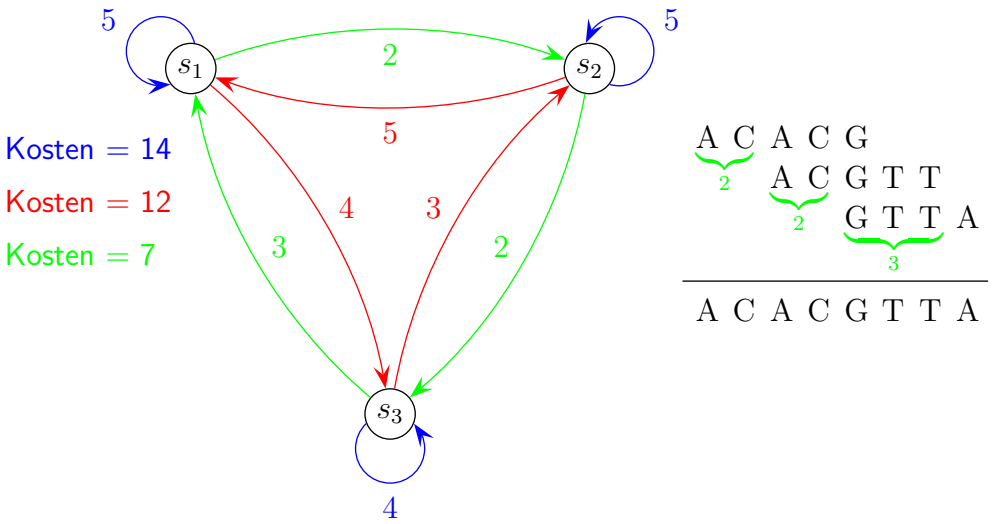


Abbildung 5.11: Beispiel: Zyklenüberdeckung für {ACACG,ACGTT,GTTA}

5.4.2 Hamiltonsche Kreise und Zyklenüberdeckungen

Definition 5.87 *Sei $G = (V, E)$ ein Graph. Ein Pfad $(v_1, \ldots, v_k) \in V^k$ heißt* hamiltonsch*, wenn $(v_{i-1}, v_i) \in E$ für alle $i \in [2 : k]$ ist und $\{v_1, \ldots, v_k\} = V$ sowie $|V| = k$ gilt. $(v_1, \ldots, v_k) \in V^k$ ist ein* hamiltonscher Kreis*, wenn $(v_1, \ldots, v_k)$ ein hamiltonscher Pfad ist und wenn $(v_k, v_1) \in E$ gilt. Ein Graph heißt* hamiltonsch*, wenn er einen hamiltonschen Kreis besitzt.*

Damit haben wir eine wichtige Beziehung gefunden: Superstrings von S und hamiltonsche Kreise im Präfix-Graphen von S korrespondieren zueinander. Das Gewicht eines hamiltonschen Kreises im Präfix-Graphen von S entspricht fast der Länge des zugehörigen Superstrings, nämlich bis auf $|o(s_j, s_{(j \bmod k)+1})|$, je nachdem, an welcher Stelle j man den hamiltonschen Kreis aufschneidet.

Im Folgenden werden wir also statt kürzester Superstrings für S kürzeste hamiltonsche Kreis im entsprechenden Präfix-Graphen suchen. Dabei werden wir von der Hoffnung geleitet, dass die Länge des gewichteten hamiltonschen Kreises im Wesentliche der Länge des kürzesten Superstrings entspricht und die Größe $|o(s_j, s_{(j \bmod k)+1})|$ ohne spürbaren Qualitätsverlust vernachlässigt werden kann.

Definition 5.88 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und G_S der zugehörige Präfix-Graph. $C(G_S)$ bezeichnet den kürzesten (bzgl. des Gewichtes) Hamiltonschen Kreis in G_S:*

$$C(G_S) := min \left\{ \sum_{j=1}^{k} |p(s_{i_j}, s_{i_{j+1}})| \ : \ (s_{i_1}, \ldots, s_{i_k}) \in \mathcal{H}(G_S) \right\},$$

wobei $\mathcal{H}(G)$ die Menge aller hamiltonscher Kreise in einem Graphen G bezeichnet.

Definition 5.89 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei S^* die Menge aller Superstrings von S. Dann bezeichnet*

$$SSP(S) := min \{|s| \ : \ s \in S^*\}$$

die Länge eines kürzesten Superstrings für S.

Das folgende Korollar fasst die eben gefundene Beziehung noch einmal zusammen.

Korollar 5.90 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und G_S der zugehörige Präfix-Graph, dann gilt $C(G_S) \leq SSP(S)$.*

Leider ist die Berechnung von hamiltonschen Kreisen mit minimalem Gewicht ebenfalls ein algorithmisch schwer lösbares Problem. In der Literatur ist es als *Traveling Salesperson Problem(TSP)* bekannt und ist $\mathcal{NP}$-hart. Daher gibt es auch wieder keine optimale Lösung, die sich in polynomieller Zeit berechnen lässt (außer, wenn $\mathcal{P} = \mathcal{NP}$ gilt). Daher werden wir die Problemstellung etwas relaxieren und zeigen, dass wir dafür eine optimale Lösung in polynomieller Zeit berechnen können. Leider wird die optimale Lösung des relaxierten Problems nur eine Näherungslösung für das ursprüngliche Problem liefern

Für die weiteren Untersuchungen wiederholen wir noch ein paar elementare graphentheoretische Bezeichnungen. Sei im Folgenden $G = (V, E)$ ein ungerichteter Graph. Für $v \in V$ bezeichnen wir mit

$$N(v) = \{w \,:\, \{v, w\} \in E\}$$

die *Nachbarschaft* des Knotens v. Mit dem *Grad*

$$d(v) := |N(v)|$$

des Knotens v bezeichen wir die Anzahl seiner Nachbarn. Einen Knoten mit Grad 0 nennen wir einen *isolierter Knoten*. Mit

$$\begin{aligned} \Delta(G) &:= \max\{d(v) \,:\, v \in V\} \qquad \text{bzw.} \\ \delta(G) &:= \min\{d(v) \,:\, v \in V\} \end{aligned}$$

bezeichnen wir den *Maximal-* bzw. *Minimalgrad* eines Knotens in G.

Im Falle gerichteter Graphen gibt es folgenden Ergänzungen und Modifikationen. Sei also im Folgenden $G = (V, E)$ ein gerichteter Graph. Die Menge der Nachbarn eines Knotens v bezeichnen wir weiterhin mit $N(v)$. Wir unterteilen die Nachbarschaft in die Menge der direkten Nachfolger und der direkten Vorgänger, die wir mit $N^+(v)$ und $N^-(v)$ bezeichnen wollen:

$$\begin{aligned} N^+(v) &= \{w \in V \,:\, (v, w) \in E\}, \\ N^-(v) &= \{w \in V \,:\, (w, v) \in E\}, \\ N(v) &= N^+(v) \cup N^-(v). \end{aligned}$$

Der *Eingangsgrad* bzw. *Ausgangsgrad* eines Knotens $v \in V(G)$ ist die Anzahl seiner direkten Vorgänger bzw. Nachfolger und wird mit $d^- = |N^-(v)|$ bzw. $d^+ = |N^+(v)|$ bezeichnet. Der *Grad* eines Knotens $v \in V(G)$ ist definiert als $d(v) := d^-(v) + d^+(v)$ und es gilt somit $d \geq |N(v)|$. Mit

$$\begin{aligned} \Delta(G) &:= \max\{d(v) \,:\, v \in V\} \qquad \text{bzw.} \\ \delta(G) &:= \min\{d(v) \,:\, v \in V\} \end{aligned}$$

bezeichnen wir den *Maximal-* bzw. *Minimalgrad* eines Knotens in G. Mit

$$\begin{aligned} \Delta^-(G) &:= \max\left\{d^-(v) \,:\, v \in V\right\} \qquad \text{bzw.} \\ \delta^-(G) &:= \min\left\{d^-(v) \,:\, v \in V\right\} \end{aligned}$$

bezeichnen wir den *maximalen* bzw. *minimalen Eingangsgrad* eines Knotens in G. Analog bezeichen wir mit

$$\begin{aligned} \Delta^+(G) &:= \max\left\{d^+(v) \,:\, v \in V\right\} \qquad \text{bzw.} \\ \delta^+(G) &:= \min\left\{d^+(v) \,:\, v \in V\right\} \end{aligned}$$

den *maximalen* bzw. *minimalen Ausgangsgrad* eines Knotens in G.

Jetzt können wir formal, die im Folgenden wichtige Zyklenüberdeckung definieren.

Definition 5.91 *Sei $G = (V, E)$ ein gerichteter Graph. Eine* Zyklenüberdeckung *(engl.* cycle cover*) von G ist ein Teilgraph $C = (V', E')$ mit den folgenden Eigenschaften:*

- $V' = V$,
- $E' \subseteq E$,
- $\Delta^+(G) = \Delta^-(G) = \delta^+(G) = \delta^-(G) = 1$.

Mit $\mathcal{C}(G)$ bezeichnen wir die Menge aller Zyklenüberdeckungen von G.

Ist $C \in \mathcal{C}(G)$, dann bezeichne C_i mit $C = \bigcup_i C_i$ die einzelnen Zusammenhangskomponenten von C. Dabei ist dann jede Komponente C_i ein gerichteter Kreis.

Definition 5.92 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und G_S der zugehörige Präfix-Graph. Dann bezeichnet $CS(G_S)$ das Gewicht einer minimalen Zyklenüberdeckung:*

$$CS(G_S) := min\left\{\sum_{i=1}^{r} \gamma(C_i) \,:\, C = \bigcup_i C_i \wedge C \in \mathcal{C}(G_S)\right\}.$$

Notation 5.93 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und G_s der zugehörige Präfix-Graph. Weiter sei $C = \bigcup_{i=1}^{r} C_i$ eine Zyklenüberdeckung für G_S. Dann bezeichne*

$$\begin{aligned} \ell(C_i) &:= \ell_i := \max\{|s_j| \,:\, s_j \in V(C_i)\}, \\ w(C_i) &:= w_i := \gamma(C_i) = \sum_{c \in E(C_i)} \gamma(c) = \sum_{(u,v) \in E(C_i)} |p(u,v)|. \end{aligned}$$

Der einfache Beweis des folgenden Lemmas sei dem Leser überlassen.

Lemma 5.94 *Ist s' ein Superstring von $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$, der aus einer Zyklenüberdeckung $C = \bigcup_{i=1}^{r} C_i$ konstruiert wurde, dann gilt:*

$$\sum_{i=1}^{r} w_i \leq |s'| \leq \sum_{i=1}^{r} (w_i + \ell_i).$$

5.4.3 Berechnung einer optimalen Zyklenüberdeckung

In diese Abschnitt wollen wir nun zeigen, dass sich eine optimale Zyklenüberdeckung effizient berechnen lässt. Dazu benötigen wir der einfacheren Beschreibung wegen noch eine Definition.

Definition 5.95 *Der gewichtete gerichtete Graph $B_S = (V, E, \gamma)$ für eine Menge von Sequenzen $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ heißt* Overlap-Graph *von S, wobei:*

- $V = S \cup S'$ *wobei* $S' = \{s' : s \in S\}$ *mit* $S \cap S' = \emptyset$,
- $E = \{\{s, s'\} : s \in S \wedge s' \in S'\}$,
- $\gamma(s, t') = |o(s, t)| = |s| - |p(s, t)|$ *für* $s, t \in S$.

In Abbildung 5.12 ist der Overlap-Graph B_S für unser bereits bekannten Beispielsequenzen angegeben.

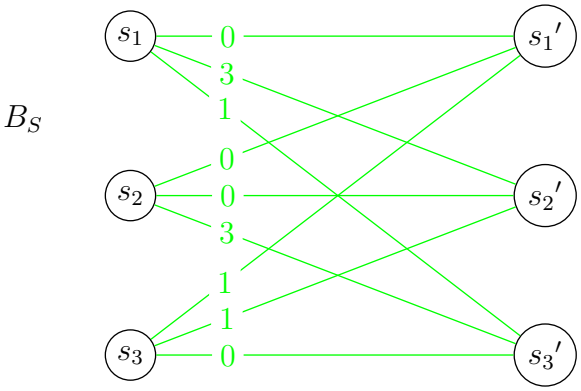


Abbildung 5.12: Beispiel: Overlap-Graph für {ACACG,ACGTT,GTTA}

Es drängt sich nun die Idee auf, dass minimale Zyklenüberdeckungen in G_S gerade gewichtsmaximalen Matchings in B_S entsprechen. Ob dies nun tatsächlich der Fall ist, soll im Folgenden untersucht werden.

Definition 5.96 *Sei G ein ungerichteter Graph. Die Kantenmenge $M \subseteq E(V)$ heißt* Matching*, wenn $\Delta(G(M)) = 1$ für den Graphen $G(M) = (V(G), M)$ gilt. Ein Matching M heißt* perfekt*, wenn auch $\delta(G(M)) = 1$ gilt.*

Man beachte, dass nur Graphen mit einer geraden Anzahl von Knoten ein perfektes Matching besitzen können. Im Graphen in der Abbildung 5.13 entsprechen die rot hervorgehobenen Kanten einem perfekten Matching M des Graphen G_S.

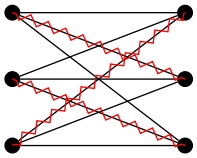


Abbildung 5.13: Beispiel: Matching in G_S

Aus einer Zyklenüberdeckung im Präfix-Graphen können wir sehr einfach ein perfektes Matching in einem Overlap-Graphen konstruieren. Für jede Kante (s, t) in der Zyklenüberdeckung im Präfix-Graphen nehmen wir $\{s, t'\}$ in das Matching des Overlap-Graphen auf. Umgekehrt können wir eine Zyklenüberdeckung im Präfix-Graphen aus einen Matching des Overlap-Graphen konstruieren, indem wir für jede Matching-Kante $\{s, t'\}$ die gerichtete Kante (s, t) in die Zyklenüberdeckung aufnehmen. Man überlegt sich leicht, dass man aus der Zyklenüberdeckung im Präfix-Graphen ein perfektes Matching im Overlap-Graphen erhält und umgekehrt.

In der Abbildung 5.14 wird nochmals der Zusammenhang zwischen einem minimalen CC in G_S und einem gewichtsmaximalen Matching in B_S anhand des bereits bekannten Beispiels illustriert. Hier ist der Übersichtlichkeit halber ein Zyklus und

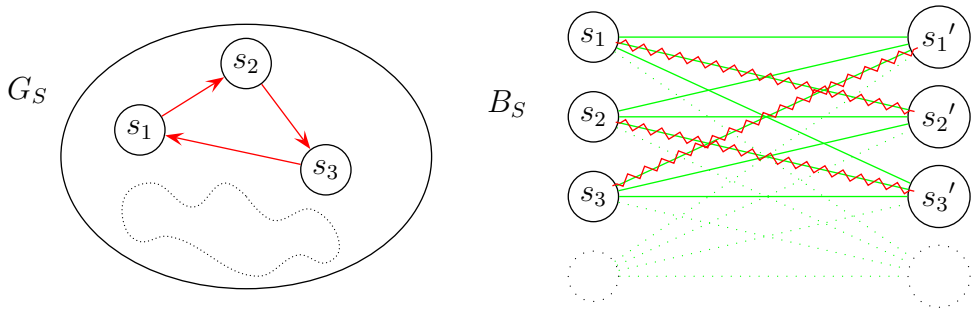


Abbildung 5.14: Skizze: Cycle Cover in G_S entspricht perfektem Matching in B_S

das korrespondierende perfekte Matching auf den entsprechende Knoten besonders hervorgehoben.

Sei $C = \bigcup_{i=1}^{r} C_i$ eine Zyklenüberdeckung von G_S. Es gilt dann:

$$\gamma(C) = \sum_{(u,v)\in E(C)} |p(u,v)|.$$

Für ein perfektes Matching M in B_s erhalten wir entsprechend:

$$\begin{aligned}
\gamma(M) &= \sum_{(u,v)\in M} |o(u,v)| \\
&= \sum_{(u,v)\in M} (\underbrace{|u| - |p(u,v)|}_{|o(u,v)|}) \\
&= \underbrace{\sum_{u\in S} |u|}_{=:N} - \sum_{(u,v)\in M} |p(u,v)|.
\end{aligned}$$

Damit erhalten wir, dass für ein zu einer Zyklenüberdeckung C in G_s korrespondierendes perfektes Matching M in B_S gilt:

$$\begin{aligned}
M &= \{(u,v) \,:\, (u,v) \in E(C)\}, \\
\gamma(M) &= N - \gamma(C).
\end{aligned}$$

Umgekehrt erhalten wir, dass für eine zu einem perfekten Matching M in B_s korrespondierende Zyklenüberdeckung C in G_S gilt:

$$\begin{aligned}
C &= \{(u,v) \,:\, (u,v') \in M\}, \\
\gamma(C) &= N - \gamma(M).
\end{aligned}$$

Hierbei ist $N = \sum_{s\in S} |s|$ eine nur von der Menge $S = \{s_1, \ldots, s_k\}$ abhängige Konstante. Somit können wir nicht aus der Zyklenüberdeckungen im Präfix-Graphen sehr einfach ein perfektes Matching konstruieren und umgekehrt, sondern auch die gewichteten Werte der auseinander konstruierten Teilgraphen lassen sich sehr leicht berechnen. Fassen wir das im folgenden Satz noch einmal zusammen.

Theorem 5.97 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ mit $N = \sum_{s\in S} |s|$ und G_S bzw. B_S der zugehörige Präfix- bzw. Overlap-Graph. Zu jeder minimalen Zyklenüberdeckung C in G_S existiert ein gewichtsmaximales Matching M in B_S mit $\gamma(M) = N - \gamma(C)$ und zu jedem gewichtsmaximalen Matching M in B_S existiert eine minimale Zyklenüberdeckung C in G_S mit $\gamma(C) = N - \gamma(M)$.*

5.4.4 Berechnung gewichtsmaximaler Matchings

Wenn wir nun ein gewichtsmaximales Matching in B_s gefunden haben, haben wir also sofort eine Zyklenüberdeckung von G_S mit minimalem Gewicht. Zum Auffinden eines gewichtsmaximalen perfekten Matchings in B_S werden wir wieder einmal einen Greedy-Ansatz verwenden. Wir sortieren zunächst die Kanten absteigend nach ihrem Gewicht. Dann testen wir in dieser Reihenfolge jede Kante, ob wir diese zu unserem bereits konstruierten Matching hinzunehmen dürfen, um ein Matching mit einer größeren Kardinalität zu erhalten. Dieser Ansatz ist noch einmal im Pseudo-Code in Abbildung 5.15 angegeben.

W_Max_Matching (graph (V, E, γ))

```
begin
    set M = ∅;
    Sortiere E nach den Gewichten γ;
    Sei also E = {e_1, ..., e_m} mit γ(e_1) ≥ ··· ≥ γ(e_m);
    for (i = 1; i ≤ m; i++) do
        if (e_i ist zu keiner anderen Kante aus M inzident) then
            M = M ∪ {e_i};
    return M;
end
```

Abbildung 5.15: Algorithmus: Greedy-Methode für ein gewichtsmaximales Matching

Zunächst einmal halten wir fest, dass wir immer ein perfektes Matching erhalten. Dies folgt unmittelbar aus dem Heiratssatz (auch Satz von Hall). Für die Details verweisen wir auf die entsprechenden Vorlesungen oder die einschlägige Literatur. Wir werden später noch zeigen, dass wir wirklich ein gewichtsmaximales Matching erhalten, da der Graph B_S spezielle Eigenschaften aufweist, die mit Hilfe eines Greedy-Ansatzes eine optimale Lösung zulassen. Wir merken an dieser Stelle noch kurz an, dass es für bipartite Graphen einen Algorithmus zum Auffinden gewichtsmaximaler Matchings gibt, der eine Laufzeit von $O(k^3)$ besitzt. Auf die Details dieses Algorithmus sei an dieser Stelle auf die einschlägige Literatur verwiesen.

Nun wollen wir uns um die Laufzeit unseres Greedy-Algorithmus kümmern. Wir werden zeigen, dass er ein besseres Laufzeitverhalten als $O(k^3)$ besitzt. Für das Sortieren der k^2 Kantengewichte benötigen wir eine Laufzeit von $O(k^2 \log(k))$. Das Abtesten jeder einzelnen Kante, ob sie zum Matching hinzugefügt werden darf, kann in konstanter Zeit realisiert werden. Somit ist die gesamte Laufzeit $O(k^2 \log(k))$.

Wenn man annehmen kann, dass die Kantengewichte nur aus einem kleinen Intervall möglicher Werte vorkommen, so kann man die Sortierphase mit Hilfe eines Bucket-

Sorts noch auf $O(k^2)$ beschleunigen. Auch hier verweisen wir für die Details auf die einschlägige Literatur.

Lemma 5.98 *Der Greedy-Algorithmus liefert für einen gewichteten vollständigen bipartiten Graphen auf $2k$ Knoten in Zeit $O(k^2 \log(k))$ ein gewichtsmaximales Matching.*

Jetzt wollen wir uns nur noch darum kümmern, dass der Greedy-Algorithmus wirklich ein gewichtsmaximales Matching findet. Dazu benötigen wir die so genannte Monge-Ungleichung.

Definition 5.99 *Sei $G = (A, B, E, \gamma)$ ein gewichteter vollständiger bipartiter Graph mit $E = \{\{a, b\} : a \in A \wedge b \in B\}$ und $\gamma : E \to \mathbb{R}$. Der Graph G erfüllt die* Monge-Ungleichung *oder* Monge-Bedingung*, wenn für beliebige vier Knoten $s, p \in A$ und $t, q \in B$ mit $\gamma(s,t) \geq \max\{\gamma(s,q), \gamma(p,t), \gamma(p,q)\}$ gilt, dass*

$$\gamma((s,t)) + \gamma((p,q)) \geq \gamma((s,q)) + \gamma((p,t)).$$

Die Monge-Bedingung ist in der folgenden Abbildung 5.16 illustriert. Anschaulich besagt diese, dass auf Knoten das perfekte Matching mit der gewichtsmaximalen Kante ein Gewicht besitzt, das mindestens so groß ist wie das andere mögliche perfekte Matching auf diesen vier Knoten.

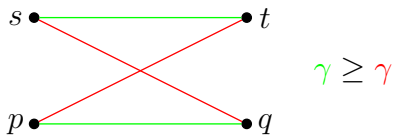


Abbildung 5.16: Skizze: Monge-Bedingung

Wir werden zuerst zeigen, dass unser Overlap-Graph B_S die Monge-Bedingung erfüllt und anschließend, dass der Greedy-Algorithmus zur Bestimmung gewichtsmaximaler Matchings auf gewichteten vollständigen bipartiten Graphen mit der Monge-Bedingung eine optimale Lösung liefert.

Lemma 5.100 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und B_S der zugehörige Overlap-Graph. Dann erfüllt B_S die Monge-Ungleichung.*

Beweis: Seien $s, p \in A$ und t, q beliebige vier Knoten des Overlap-Graphen B_S, so dass die für die Monge-Ungleichung die Kante (s,t) maximales Gewicht besitzt.

Insbesondere gelte $\gamma(s,t) \geq \max\{\gamma(s,q), \gamma(p,t), \gamma(p,q)\}$. Betrachten wir die vier zugehörigen Kanten und ihre entsprechenden Zeichenreihen aus S. Der Einfachheit wegen identifizieren wir die Knoten des Overlap-Graphen mit den entsprechenden Zeichenreihen aus S. Da die Kantengewichte gleich den Overlaps der Zeichenreihen sind, ergibt sich das folgende in Abbildung 5.17 illustrierte Bild.

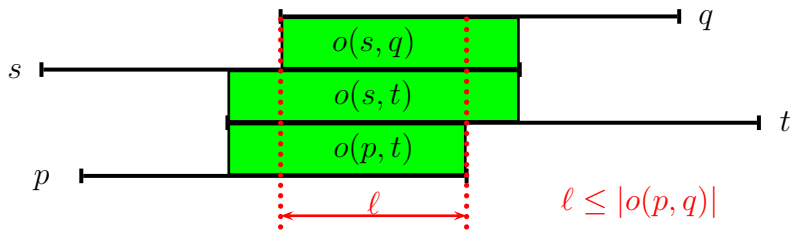


Abbildung 5.17: Skizze: Overlap-Graph erfüllt Monge-Bedingung

Da s und t nach Voraussetzung den längsten Overlaps (das maximale Gewicht) besitzen, kann der Overlap von s und q sowie von p und t nicht länger sein (grüne Bereiche im Bild). Betrachten man nun den roten Bereich der Länge ℓ, so stellt man fest, dass hier sowohl s und q sowie s und t als auch p und t übereinstimmen. Daher muss in diesem Bereich auch p und q übereinstimmen und wir haben eine untere Schranke für $|o(p,q)|$ gefunden. Man beachte, dass der rote Bereich ein echtes Teilwort ist (wie in der Definition des Overlaps gefordert). Daher können wir sofort folgern, dass gilt:

$$|o(s,t)| + |o(p,q)| \geq |o(s,q)| + |o(p,t)|.$$

Damit gilt dann auch, dass $\gamma(s,t) + \gamma(p,q) \geq \gamma(s,q) + \gamma(p,t)$ und das Lemma ist bewiesen. ■

Theorem 5.101 *Sei $G = (A, B, E, \gamma)$ ein gewichteter vollständiger bipartiter Graph mit $E = \{\{a,b\} : a \in A \wedge b \in B\}$ und $\gamma : E \to \mathbb{R}_+$, der die Monge-Bedingung erfüllt. Der Greedy-Algorithmus für gewichtsmaximale Matchings in G liefert eine optimale Lösung.*

Beweis: Wir führen den Beweis durch Widerspruch. Sei M das Matching, das vom Greedy-Algorithmus konstruiert wurde und sei M^* ein optimales Matching in B_S, d.h wir nehmen an, dass $\gamma(M^*) > \gamma(M)$. Wir wählen unter allen möglichen Gegenbeispielen ein „kleinstes“ (so genannter kleinster Verbrecher), d.h. wir wählen unter allen optimalen Matchings das aus, so dass $|M^* \triangle M|$ minimal ist, wobei M das vom Greedy-Algorithmus erzeugte Matching ist. Hierbei bezeichnet $A \triangle B := (A \setminus B) \cup (B \setminus A)$ die symmetrische Differenz von A und B.

Da $\gamma(M^*) > \gamma(M)$ muss $M^* \neq M$ sein. Da alle Kanten nichtnegativ sind, können wir ohne Beschränkung der Allgemeinheit annehmen, dass ein gewichtsmaximales Matching auch perfekt sein muss.

Wir wählen jetzt eine gewichtsmaximale Kante aus, die im Matching des Greedy-Algorithmus enthalten ist, die aber nicht im optimalen Matching ist, d.h. wir wählen $(s,t) \in M \setminus M^*$, so dass $\gamma(s,t)$ maximal unter diesen ist.

Da (wie oben bereits angemerkt) das gewichtsmaximale Matching perfekt sein muss, muss M^* zwei Kanten beinhalten, die die Knoten s und t überdecken. Also seien p und q so gewählt, dass $\{s,q\} \in M^*$ und $\{p,t\} \in M^*$ gilt. Zusätzlich betrachten wir noch die Kante $\{p,q\}$, die nicht im optimalen Matching M^* enthalten ist. Die Kante $\{p,q\}$ kann, muss aber nicht im Matching des Greedy-Algorithmus enthalten sein. Diese vier Knoten und Kanten sind in der Abbildung 5.18 noch einmal illustriert.

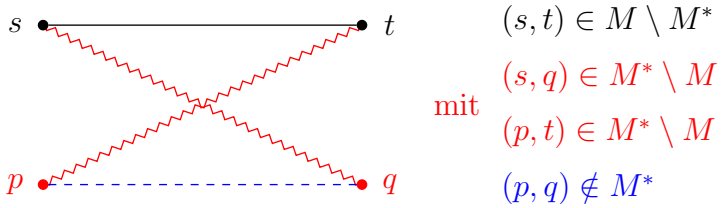


Abbildung 5.18: Skizze: Widerspruchsbeweis zur Optimalität

Da $\{s,t\}$ eine schwerste Kante aus $M \setminus M^*$ ist, muss

$$\gamma(s,t) \geq \gamma(s,q) \quad \wedge \quad \gamma(s,t) \geq (p,t)$$

gelten, da der Greedy-Algorithmus ansonsten $\{s,q\}$ oder $\{p,t\}$ anstatt $\{s,t\}$ gewählt hätte. Man sollte hier noch anmerken, dass zu diesem Zeitpunkt der Greedy-Algorithmus keine Kante ins Matching aufgenommen hat, die p oder q überdeckt. Eine solche Kante wäre ebenfalls in $M \setminus M^*$ und da die Kante (s,t) mindestens eine schwerste ist, wird diese vom Greedy-Algorithmus zuerst gewählt.

Da für den Overlap-Graphen B_S die Monge-Ungleichung erfüllt ist, gilt

$$\gamma((s,t)) + \gamma((p,q)) \;\geq\; \gamma((s,q)) + \gamma((p,t)).$$

Wir betrachten nun folgende Menge $M' := M^* \setminus \{(s,q),(p,t)\} \cup \{(s,t),(p,q)\}$. Offensichtlich ist M' ein perfektes Matching von B_S. Aus der Monge-Ungleichung folgt, dass $\gamma(M') \geq \gamma(M^*)$. Da M^* ein gewichtsmaximales Matching war, gilt also $\gamma(M') = \gamma(M^*)$. Offensichtlich gilt aber auch

$$|M' \triangle M| \;<\; |M^* \triangle M|.$$

Dies ist der gewünschte Widerspruch, da wir ja mit M einen kleinsten Verbrecher als Gegenbeispiel gewählt haben und jetzt angeblich M' ein noch kleinere Verbrecher wäre. ■

5.4.5 Greedy-Algorithmus liefert eine 4-Approximation

Bis jetzt haben wir jetzt gezeigt, dass wir eine Näherungslösung für das SSP effizient gefunden haben. Wir müssen jetzt noch die Güte der Näherung abschätzen. Hierfür müssen wir erst noch ein noch ein paar grundlegende Definitionen und elementare Beziehungen für periodische Zeichenreihen zur Verfügung stellen.

Definition 5.102 *Ein Wort $s \in \Sigma^*$ hat eine* Periode *p, wenn $p < |s|$ ist und es gilt:*

$$\forall i \in [1 : |s| - p] : \; s_i = s_{i+p}.$$

Man sagt dann auch, s besitzt die Periode p.

Beispielsweise besitzt $w = aaaaaa$ die Periode 3, aber auch jede andere Periode aus $[1 : |w| - 1]$. Das Wort $w' = ababc$ besitzt hingegen gar keine Periode.

Zunächst werden wir zeigen, dass zwei verschiedene Perioden für dasselbe Wort gewisse Konsequenzen für die kleinste Periode dieses Wortes hat. Im Folgenden bezeichnet $\text{ggT}(a, b)$ für zwei natürliche Zahlen $a, b \in \mathbb{N}$ den größten gemeinsamen Teiler: $\text{ggT}(a, b) = \max\{k \in \mathbb{N} : (k \mid a) \wedge (k \mid b)\}$, wobei $k \mid a$ gilt, wenn es ein $n \in \mathbb{N}$ gibt, so dass $n \cdot k = a$.

Lemma 5.103 (GGT-Lemma für Zeichenreihen) *Sei $s \in \Sigma^*$ ein Wort mit Periode p und mit Periode q, wobei $p > q$ und $p + q \leq |s|$. Dann hat s auch eine Periode von $\text{ggT}(p, q)$.*

Beweis: Wir zeigen zunächst, dass das Wort s auch die Periode $p - q$ besitzt. Dazu unterscheiden wir zwei Fälle, je nachdem, wie sich i zu q verhält.

Fall 1 ($i \leq q$): Da $i \leq q$ ist ist $i + p \leq p + q \leq |s|$. Weiter besitzt s die Periode p und es gilt $s_i = s_{i+p}$. Da $p > q$ ist, gilt $p - q > 0$ und somit ist $i + p - q > i$. Weiter besitzt s auch die Periode q und es $s_{i+p} = s_{i+p-q}$. Insgesamt ist also $s_i = s_{i+(p-q)}$ für $i \leq q$. Dies ist in der Abbildung 5.19 illustriert.

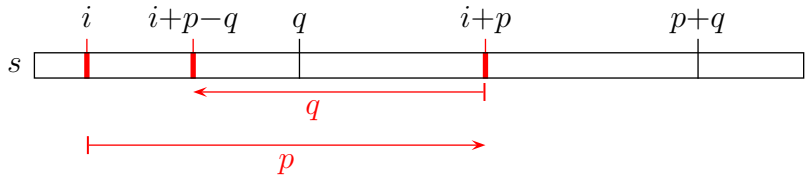


Abbildung 5.19: Skizze: 1. Fall $i \leq q$

Fall 2 ($i > q$): Da $i > q$ ist ist $i - q \geq 1$. Weiter besitzt s die Periode q und es gilt $s_i = s_{i-q}$. Da $p > q$ ist, gilt $p - q > 0$ und somit ist $i + p - q > i$. Weiter besitzt s auch die Periode p und es $s_{i-q} = s_{i+p-q}$. Insgesamt ist also $s_i = s_{i+(p-q)}$ für $i \leq q$. Dies ist in der Abbildung 5.20 illustriert.

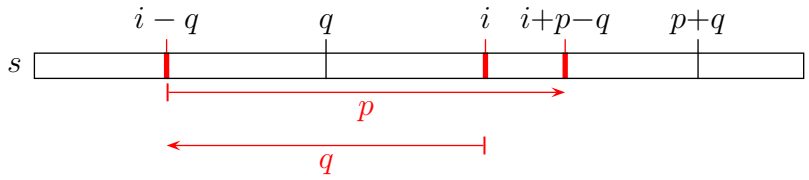


Abbildung 5.20: Skizze: 2.Fall $i > q$

Damit gilt für alle $i \in [1 : |s| - (p - q)]$, dass $s_i = s_{i+(p-q)}$. Somit besitzt s auch die Periode $p - q$.

Erinnern wir uns an den Euklidischen Algorithmus. Dieser kann für $p > q$ durch $\text{ggT}(p, q) = \text{ggT}(q, p - q)$ rekursiv definiert werden; dabei ist die Abbruchbedingung dann durch $\text{ggT}(p, p) = p$ gegeben. Da die Perioden von s dieselbe Rekursionsgleichung erfüllen, muss also auch $\text{ggT}(p, q)$ eine Periode von s sein. ■

Mithilfe dieses Lemmas können wir jetzt eine nahe liegende, jedoch für die Approximierbarkeit wichtige Eigenschaft von einer optimalen Zyklenüberdeckung beweisen. Diese besagt umgangssprachlich, dass zwei Wörter, die in verschiedenen Kreisen der Zyklenüberdeckung vorkommen, keine allzu große Überlappung besitzen können.

Lemma 5.104 (Overlap-Lemma) *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und G_S der zugehörige Präfix-Graph. Sei $C = \bigcup_{i=1}^{r} C_i$ eine gewichtsminimale Zyklenüberdeckung für G_S. Sei $t_i \in V(C_i)$ und $t_j \in V(C_j)$ mit $i \neq j$. Dann gilt:*

$$|o(t_i, t_j)| \leq w_i + w_j = \gamma(C_i) + \gamma(C_j).$$

Beweis: Wir führen den Beweis durch Widerspruch und nehmen hierzu an, dass $|o(t_i, t_j)| > w_i + w_j$. Wir beobachten dann das Folgende:

- t_i hat Periode w_i und damit hat auch $o(t_i, t_j)$ die Periode w_i;
- t_j hat Periode w_j und damit hat auch $o(t_i, t_j)$ die Periode w_j;
- Es gilt $|t_i| > |o(t_i, t_j)| > w_i + w_j \geq w_i$;
- Es gilt $|t_j| > |o(t_i, t_j)| > w_i + w_j \geq w_j$.

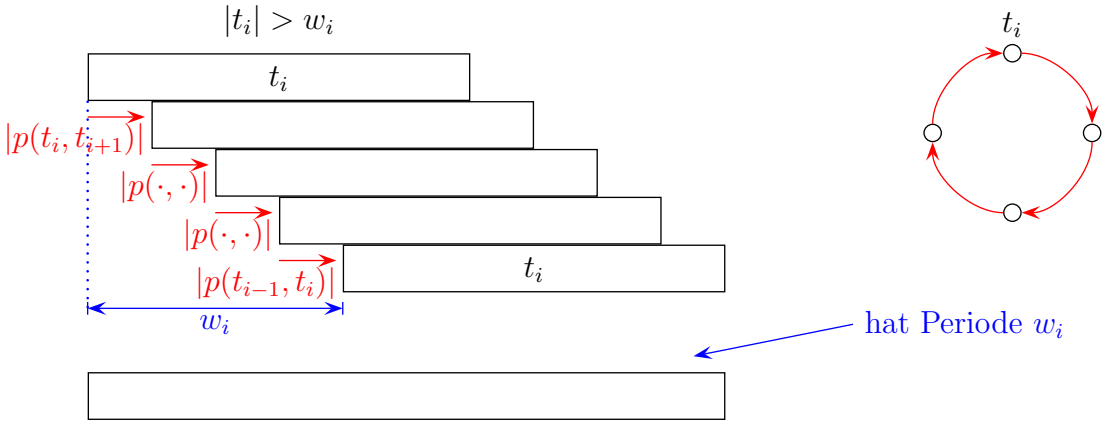


Abbildung 5.21: Skizze: Wörter eines Zyklus

Wir betrachten jetzt alle Knoten im Kreis C_i; genauer betrachten wir alle Wörter, deren korrespondierende Knoten sich im Kreis C_i befinden, siehe dazu die folgende Skizze in Abbildung 5.21. Hier sind die Wörter entsprechend der Reihenfolge des Auftretens in C_i beginnend mit t_i angeordnet. Der Betrag der Verschiebung der Wörter entspricht gerade der Länge des Präfixes im vorausgehenden zum aktuell betrachteten Wort in C_i. Man beachte, dass auch die Wortenden monoton aufsteigend sind. Andernfalls wäre ein Wort Teilwort eines anderen Wortes, was wir zu Beginn dieses Abschnitts ausgeschlossen haben.

Sind alle Wörter des Kreises einmal aufgetragen, so wird das letzte Wort am Ende noch einmal wiederholt. Da die Wörter in den überlappenden Bereichen übereinstimmen (Definition des Overlaps), kann man aus dem Kreis den zugehörigen Superstring für die Wörter aus C_i ableiten und dieser muss eine Periode von w_i besitzen.

Wir unterscheiden jetzt zwei Fälle, je nachdem, ob $w_i = w_j$ ist oder nicht.

Fall 1 ($w_i = w_j$): Da nun beide Zyklen die gleiche Periode besitzen können wir diese in einen neuen Zyklus zusammenfassen. Da der Overlap von t_i und t_j nach Widerspruchsannahme größer als $2w_i$ ist und beide Wörter die Periode w_i besitzen, muss das Wort t_j in den Zyklus von t_i einzupassen sind. Siehe dazu auch Abbildung 5.22. Man kann also die beiden Zyklen zu einem verschmelzen, so dass das Gewicht des Zyklus kleiner als $2w_i = w_i + w_j$ wäre. Dies ist aber ein Widerspruch zur Optimalität der Zyklenüberdeckung.

Fall 2 ($w_i > w_j$): Jetzt ist sicherlich $w_i \neq w_j$ und außerdem ist $|o(t_i, t_j)| \geq w_i + w_j$. Also folgt mit dem GGT-Theorem, dass $o(t_i, t_j)$ eine Periode von $g := \text{ggT}(w_i, w_j)$ besitzt. Da t_i auch die Periode w_i besitzt und g ein Teiler von w_i ist, muss das ganze Wort t_i die Periode g besitzen. Dies ist Abbildung 5.23 veranschaulicht. Eine analoge Überlegung gilt natürlich auch für t_j, so dass also auch t_j eine Periode von g besitzt.

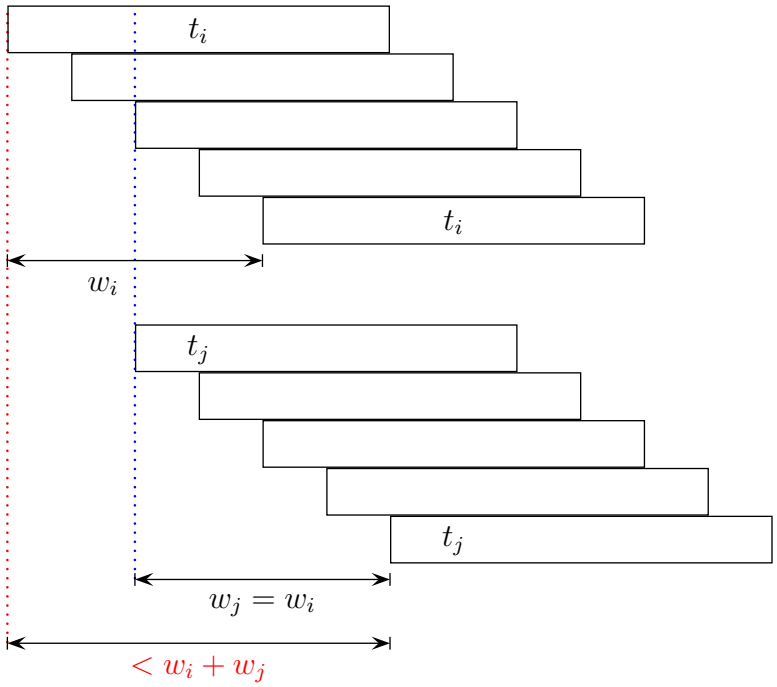


Abbildung 5.22: Skizze: 2 Zyklen mit demselben Gewicht

Wir werden auch jetzt wieder zeigen, dass sich die beiden Zyklen, die t_i bzw. t_j beinhalten, zu einem neuen Zyklus verschmelzen lassen, dessen Gewicht geringer als die Summe der beiden Gewichte der ursprünglichen Zyklen ist. Für die folgende Argumentation betrachten wir die Illustration in Abbildung 5.24.

Da sowohl t_i als auch t_j einen Periode von g besitzen und die Zeichen innerhalb der Periode (die ja auch innerhalb des Overlaps liegt) gleich sind, lässt sich der Zyklus, der t_j enthält, in den Zyklus, der t_i enthält, integrieren. Somit hat der neue Zyklus ein Gewicht von $g+w_j < w_i+w_j$, was offensichtlich ein Widerspruch zur Optimalität der Zyklenüberdeckung ist. Somit ist das Overlap-Lemma bewiesen. ■

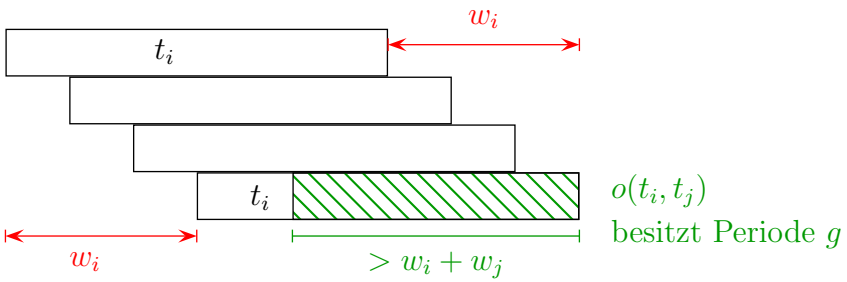


Abbildung 5.23: Skizze: Übertragung der Periode g von $o(t_i, t_j)$ auf t_i

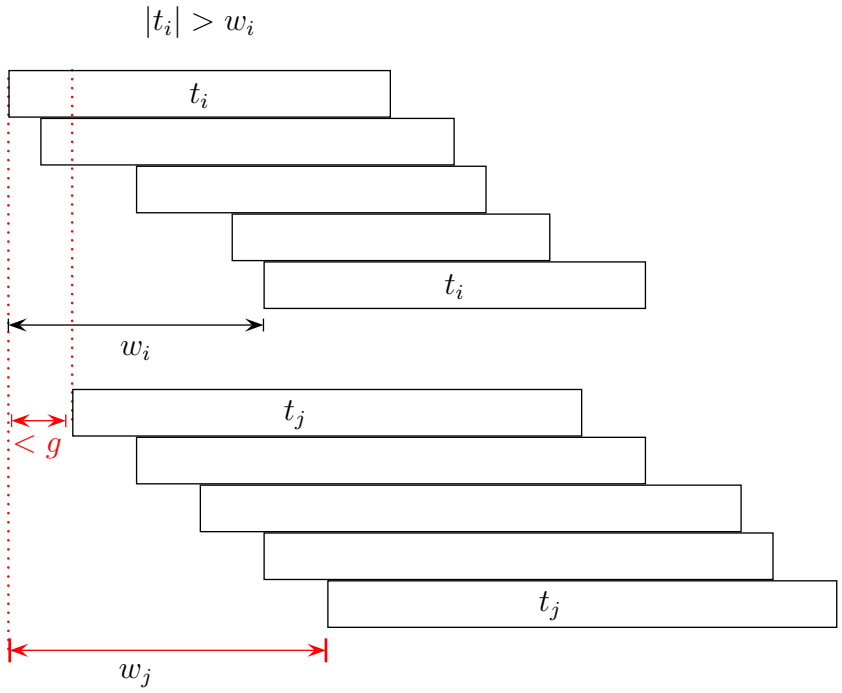


Abbildung 5.24: Skizze: Verschmelzung zweier Zyklen mit $w_i \neq w_j$

Mit Hilfe des eben bewiesenen Overlap-Lemmas können wir jetzt die Approximationsgüte des von uns vorgestellten Greedy-Algorithmus abschätzen.

Theorem 5.105 *Sei s' der durch den Greedy-Algorithmus konstruierte Superstring für $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$. Dann gilt $|s'| \leq 4 \cdot SSP(S)$.*

Beweis: Sei $\tilde{s}_i$ für $i \in [1:r]$ der jeweils längste String aus C_i in einer optimalen Zyklenüberdeckung $C = \bigcup_{i=1}^{r} C_i$ für G_S. Sei jetzt $\tilde{s}$ ein kürzester Superstring für $\tilde{S} = \{\tilde{s}_1, \ldots, \tilde{s}_k\}$. Nach Lemma 5.84 gibt es eine Permutation $(j_1, \ldots, j_r)$ von $[1:r]$, so dass sich $\tilde{s}$ schreiben lässt als:

$$\tilde{s} = p(\tilde{s}_{j_1}, \tilde{s}_{j_2}) \cdots p(\tilde{s}_{j_{r-1}}, \tilde{s}_{j_r}) \cdot p(\tilde{s}_{j_r}, \tilde{s}_{j_1}) \cdot o(\tilde{s}_{j_r}, \tilde{s}_{j_1}).$$

Dann gilt:

$$\begin{aligned} |\tilde{s}| &= \sum_{i=1}^{r} |p(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| + |\underbrace{o(\tilde{s}_{j_r}, \tilde{s}_{j_1})}_{\geq 0}| \\ &\geq \sum_{i=1}^{r} \Big(\underbrace{|\tilde{s}_{j_i}|}_{=\ell_i} - |o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \Big) \end{aligned}$$

aufgrund des Overlap-Lemmas gilt:
$|o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \leq (w_{j_i} + w_{j_{(i \bmod r)+1}})$

$$
\begin{aligned}
&\geq \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} \left(w_{j_i} + w_{j_{(i \bmod r)+1}} \right) \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_{(i \bmod r)+1}} \\
&\quad \text{Verschiebung der Indizes um 1} \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_i} \\
&\quad \text{da } (j_1, \ldots, j_r) \text{ nur eine Permutation von } [1:r] \text{ ist} \\
&= \sum_{i=1}^{r} \ell_i - 2 \sum_{i=1}^{r} w_i \\
&= \sum_{i=1}^{r} (\ell_i - 2w_i).
\end{aligned}
$$

Der kürzeste Superstring für S ist sicherlich nicht kürzer als der für $\tilde{S}$, da ja $\tilde{S} \subseteq S$ gilt. Also gilt:

$$SSP(S) \geq SSP(\tilde{S}) = |\tilde{s}| \geq \sum_{i=1}^{r} (\ell_i - 2w_i). \tag{5.3}$$

Nach Konstruktion des Superstrings s' für s mit Hilfe des Greedy-Algorithmus gilt:

$$
\begin{aligned}
|s'| &\leq \sum_{i=1}^{r} (w_i + \ell_i) \\
&\leq \underbrace{\sum_{i=1}^{r} (\ell_i - 2w_i)}_{\leq SSP(S)} + \sum_{i=1}^{r} 3w_i \\
&\quad \text{mit Hilfe von Ungleichung 5.3} \\
&\leq SSP(S) + 3 \cdot SSP(S) \\
&\leq 4 \cdot SSP(S)
\end{aligned}
$$

Damit haben wir das Theorem bewiesen. ■

Somit haben wir nachgewiesen, dass der Greedy-Algorithmus eine 4-Approximation für das Shortest Superstring Problem liefert, d.h. der generierte Superstring ist höchstens um den Faktor 4 zu lang.

Wir wollen an dieser Stelle noch anmerken, dass die Aussage, das der Greedy-Algorithmus eine 4-Approximation liefert, nur eine obere Schranke ist. Wir können für den schlimmsten Fall nur beweisen, dass der konstruierte Superstring maximal um den Faktor 4 zu lang ist. Es ist nicht klar, ob die Analyse scharf ist, dass heißt, ob der Algorithmus nicht im worst-case bessere Resultate liefert. Für den average-case können wir davon ausgehen, dass die Ergebnisse besser sind.

Wir können auch eine 3-Approximation beweisen, wenn wir etwas geschickter vorgehen. Nach der Erzeugung einer optimalen Zyklenüberdeckung generieren wir für jeden Zyklus einen Superstring, wie in Korollar 5.83 angegeben. Um den gesamten Superstring zu erhalten, werden die Superstrings für die einzelnen Zyklen einfach aneinander gehängt. Auch hier können wir Überlappungen ausnutzten. Wenn wir dies und noch ein paar weitere kleinere Tricks anwenden, erhalten wir eine 3-Approximation. Der bislang beste bekannte Approximationsalgorithmus für das Shortest Superstring Problem liefert eine 2,5-Approximation.

5.4.6 Zusammenfassung und Beispiel

In Abbildung 5.25 auf Seite 320 ist die Vorgehensweise für die Konstruktion eines kürzesten Superstrings mit Hilfe der Greedy-Methode noch einemal skizziert.

Zum Abschluss vervollständigen wir unser Beispiel vom Beginn dieses Abschnittes und konstruieren mit dem Greedy-Algorithmus einen kürzesten Superstring, der hier optimal sein wird. In Abbildung 5.26 auf Seite 321 ist links der zugehörige Präfix-Graph und rechts der zugehörige Overlap-Graph angegeben.

Zuerst bestimmen wir das maximale Matching mit dem Greedy-Algorithmus im Overlap-Graphen. Dazu wird zuerst die Kanten $\{s_1, s_2'\}$ gewählt. Wir hätten auch $\{s_2, s_3'\}$ wählen können. Egal welche hier zuerst gewählt wird, die andere wird als zweite Kante ins Matching aufgenommen. Zum Schluss bleibt nur noch die Kante $\{s_3, s_1'\}$ übrig, die aufzunehmen ist.

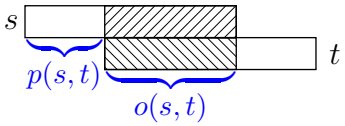
Dies entspricht der folgenden Zyklenüberdeckung im zugehörigen Präfix-Graphen (s_1, s_2, s_3) (oder aber (s_2, s_3, s_1) bzw. (s_3, s_1, s_2), je nachdem, wo wir den Kreis bei willkürlich aufbrechen). Wir erhalten hier also sogar einen hamiltonschen Kreis.

Nun müssen wir aus der Zyklenüberdeckung nur noch den Superstring konstruieren. Wie wir gesehen haben, ist es am günstigsten den Kreis nach der Kante aufzubre-

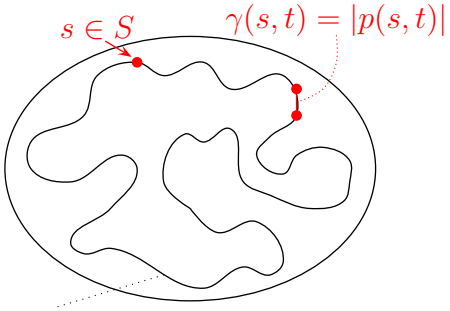
(1) Fragment-Generation

$S = \{s_1, \ldots, s_k\}$

Overlap-Detection

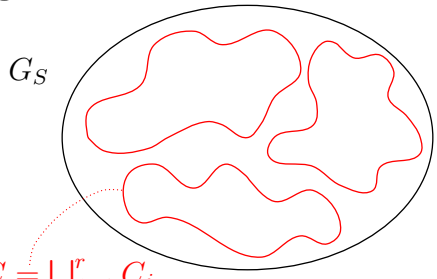


(2) Präfix-Graph G_S



Hamiltonscher Kreis

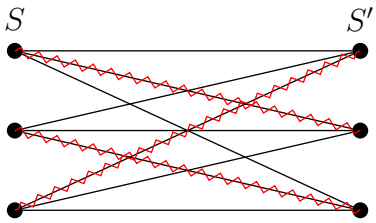
(3) Relaxierung: Cycle Cover in G_S



$C = \bigcup_{j=1}^{r} C_j$

$|s'| = \sum_{j=1}^{r} \left(\gamma(C_j) + |o(s_{i_r}^j, s_{i_1}^j)|\right)$

(3) Overlap-Graph B_S

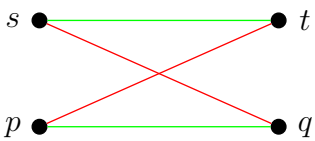


$\gamma(s, t') = |o(s,t)| \geq 0$

$\gamma(M; B_S) = N - \gamma(C; G_S)$

gewichtsmaximales Matching

(5) Monge-Bedingung

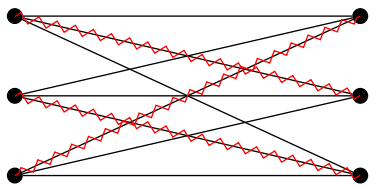


$\gamma(s,t) \geq \max\{\gamma(s,q), \gamma(p,t), \gamma(p,q)\}$

$\Rightarrow \gamma(s,t) + \gamma(p,q) \geq \gamma(s,q) + \gamma(p,t)$

Monge-Bedingung in B_S erfüllt

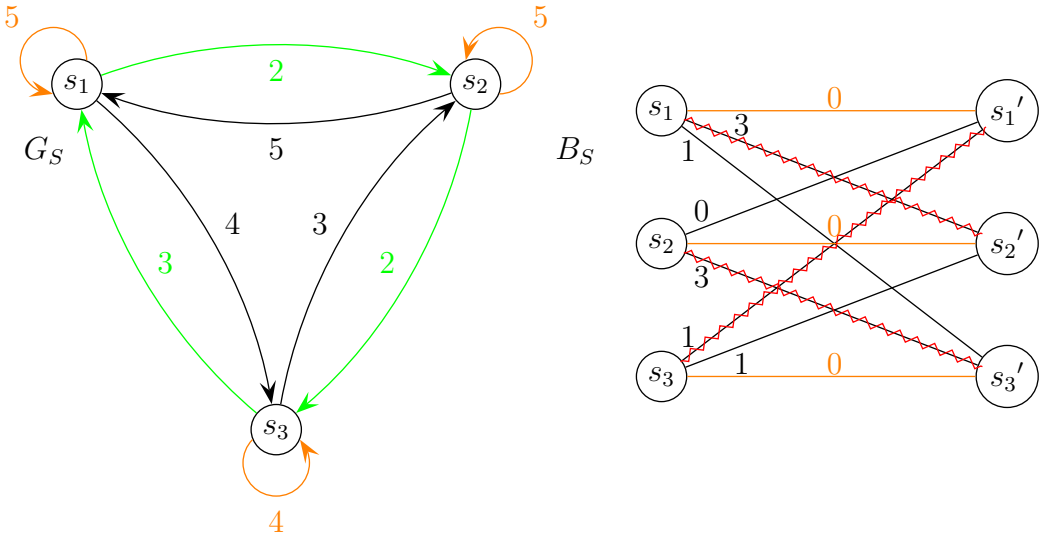
(6) Greedy-Algorithmus



Greedy liefert opt. Matching in B_S
in Zeit $O(k^2 \log(k) + nk)$
Ergebnis ist 4-Approximation

Abbildung 5.25: Skizze: Zusammenfassung der Greedy-SSP-Approximation

$$s_1 = \text{ACACG} \qquad s_2 = \text{ACGTT} \qquad s_3 = \text{GTTA} \qquad \sum_{i=1}^{3} |s_i| \;=\; 14$$



$\gamma(C) \;=\; 7 \;=\! 14 \;-\; 7$ $\qquad$ $\gamma(M) \;=\; 7 \;=\! 14 \;-\; 7$

Abbildung 5.26: Beispiel: Greedy-Algorithmus für Superstrings

chen, wo der zugehörige Overlap-Wert klein ist. Daher wählen wir als Kreisdarstellung $(s_1, s_2, s_{,3})$ und erhalten folgenden Superstring:

		A	C	A	C	G			
				A	C	G	T	T	
						G	T	T	A
s	=	A	C	A	C	G	T	T	A

Mit Bezug zum Präfix-Graphen ergibt sich folgenden Korrespondenz zu den Knoten im hamiltonschen Kreis:

$$s = \underbrace{AC}_{p(s_1,s_2)}\ \underbrace{AC}_{p(s_2,s_3)}\ \underbrace{GTT}_{p(s_3,s_1)}\ \underbrace{A}_{o(s_3,s_1)}\ .$$

Mehrfaches Sequenzen-Alignment 6

6.1 Maße für mehrfache Sequenzen-Alignments

In diesem Kapitel wollen wir uns mit der gleichzeitigen Ausrichtungen mehrerer (insbesondere mehr als zwei) Sequenzen beschäftigen. Bevor wir zum algorithmischen Teil kommen, müssen wir das Problem erst noch formalisieren.

6.1.1 Mehrfache Sequenzen-Alignments

Zuerst definieren wir mehrfache Sequenzen-Alignments analog wie im Falle paarweiser Sequenzen-Alignments. Wie im Falle des paarweisen Alignments bezeichnet $\overline{\Sigma} = \Sigma \cup \{-\}$, wobei $- \notin \Sigma$.

Definition 6.1 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Eine Folge $(\overline{s}_1, \ldots, \overline{s}_k) \in (\overline{\Sigma}^n)^k$ heißt* mehrfaches Alignment *(MSA) für die Sequenzen $s_1, \ldots, s_k$, wenn gilt:*

- *Für alle $j \in [1:k]$ gilt $|\overline{s}_j| = n$.*
- *Für alle $j \in [1:k]$ gilt $\overline{s}_j|_\Sigma = s_j$.*
- *Für alle $i \in [1:n]$ gilt: aus $\overline{s}_{1,i} = \overline{s}_{2,i} = \cdots = \overline{s}_{k,i}$ folgt $\overline{s}_{1,i} \neq -$.*

Mit $\mathcal{A}(s_1, \ldots, s_k)$ bezeichnen wir die Menge aller mehrfachen Alignments von $s_1, \ldots, s_k \in \Sigma^*$.

Nun können wir auch von einem mehrfachen Sequenzen-Alignment projizierte und induzierte paarweise Sequenzen-Alignments definieren.

Definition 6.2 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und $(\overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Alignment hierfür. Dann ist $(\overline{s}_i, \overline{s}_j)$ mit $i \neq j \in [1:k]$ ein* projiziertes paarweises Alignment*. Das paarweise Alignment $(\overline{t}_i, \overline{t}_j)$ heißt* induziertes paarweises Alignment*, wenn es aus dem projizierten paarweisen Alignment $(\overline{s}_i, \overline{s}_j)$ entstanden ist, in dem alle Paare $(-,-)$ eliminiert wurden.*

6.1.2 Alignment-Distanz und -Ähnlichkeit

Im Folgenden bezeichnen wir mit $\overline{\Sigma}_0^k$ alle k-Tupel aus $\overline{\Sigma}$, die nicht nur aus Leerzeichen bestehen, d.h. $\overline{\Sigma}_0^k := \overline{\Sigma}^k \setminus \{(-,\cdots,-)\}$. Wie im Falle paarweiser Sequenzen-Alignments können wir auch hier wieder die Alignment-Distanz und Alignment-Ähnlichkeit von mehreren Sequenzen definieren.

Definition 6.3 *Sei $w : \overline{\Sigma}_0^k \to \mathbb{R}_+$ eine Kostenfunktion für ein Distanzmaß eines k-fachen Alignments $(\overline{s}_1, \ldots, \overline{s}_k)$ für $s_1, \ldots, s_k$, dann ist*

$$w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{i=1}^{|\overline{s}_1|} w(\overline{s}_{1,i}, \ldots, \overline{s}_{k,i})$$

die Distanz des Alignments $(\overline{s}_1, \ldots, \overline{s}_k)$. *Die so genannte* Alignment-Distanz *der Sequenzen $s_1, \ldots, s_k$ ist definiert als*

$$\overline{d}_w(s_1, \ldots, s_k) := \min\left\{w((\overline{s}_1, \ldots, \overline{s}_k)) \;:\; (\overline{s}_1, \ldots, \overline{s}_k) \in \mathcal{A}(s_1, \ldots, s_k)\right\}.$$

Wie im Falle paarweiser Sequenzen-Alignments sollte die dem Distanzmaß zugrunde liegenden Kostenfunktion wieder den wesentlichen Bedingungen einer Metrik entsprechen. Die Kostenfunktion w sollte wiederum symmetrisch sein:

$$\forall (a_1, \ldots, a_k) \in \overline{\Sigma}_0^k : \forall \pi \in S(k) : w(a_1, \ldots, a_k) = w(a_{\pi(1)}, \ldots, a_{\pi(k)}).$$

Weiter sollte die Dreiecks-Ungleichung gelten:

$$\begin{aligned}\forall (a_1, \ldots, a_k) \in \overline{\Sigma}_0^k : \forall x \in \overline{\Sigma} : \quad & w(a_1, \ldots, a_i, \ldots, a_j, \ldots, a_k) \\ \leq \; & w(a_1, \ldots, a_i, \ldots, x, \ldots, a_k) + w(a_1, \ldots, x, \ldots, a_j, \ldots a_k).\end{aligned}$$

Auch sollte wieder die Definitheit gelten:

$$\forall (a_1, \ldots, a_k) \in \overline{\Sigma}_0^k : w(a_1, \ldots, a_k) = 0 \quad \Leftrightarrow \quad a_1 = \cdots = a_k.$$

Definition 6.4 *Sei $w : \overline{\Sigma}_0^k \to \mathbb{R}$ eine Kostenfunktion für ein Ähnlichkeitsmaß eines k-fachen Alignments $(\overline{s}_1, \ldots, \overline{s}_k)$ für $s_1, \ldots, s_k$, dann ist*

$$w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{i=1}^{|\overline{s}_1|} w(\overline{s}_{1,i}, \ldots, \overline{s}_{k,i})$$

die Ähnlichkeit des Alignments $(\overline{s}_1, \ldots, \overline{s}_k)$. *Die* Alignment-Ähnlichkeit *der Sequenzen $s_1, \ldots, s_k$ ist definiert als*

$$\overline{s}_w(s_1, \ldots, s_k) := \max\left\{w((\overline{s}_1, \ldots, \overline{s}_k)) \;:\; (\overline{s}_1, \ldots, \overline{s}_k) \in \mathcal{A}(s_1, \ldots, s_k)\right\}.$$

Auch diese sollte wieder symmetrisch sein. Weiter sollen positive Werte Ähnlichkeit und negative Werte Unähnlichkeit von Spalten repräsentieren.

Wenn die zugrunde liegenden Kostenfunktionen und die Verwendung von Alignments klar ist, schreibt man auch $d(s_1, \ldots, s_k)$ statt $\overline{d}_w(s_1, \ldots, s_k)$ und $s(s_1, \ldots, s_k)$ statt $\overline{s}_w(s_1, \ldots, s_k)$.

Definition 6.5 *Sei $w : \overline{\Sigma}_0^k \to \mathbb{R}$ eine Kostenfunktion für ein Distanzmaß d bzw. Ähnlichkeitsmaß s eines k-fachen Alignments. Ein k-faches Alignment $(\overline{s}_1, \ldots, \overline{s}_k)$ für $s_1, \ldots, s_k \in \Sigma^*$ heißt* optimal*, wenn*

$$w(\overline{s}_1, \ldots, \overline{s}_k) = d(s_1, \ldots, s_k) \qquad \textit{bzw.} \qquad w(\overline{s}_1, \ldots, \overline{s}_k) = s(s_1, \ldots, s_k).$$

Wir merken noch an, dass bei den verwendeten konkreten Kostenfunktionen von mehrfachen Alignments manchmal auch auf die Symmetrie verzichtet wird.

Wir werden dabei im Folgenden die Kostenfunktion auf Zeichentupel $w : \overline{\Sigma}^k \to \mathbb{R}$ auf Sequenzen von Zeichentupeln (d.h. auf mehrere Sequenzen gleicher Länge) in kanonischer Weise fortsetzen: $w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{r=1}^{n} w(\overline{s}_{1,r}, \ldots, \overline{s}_{k,r})$ für $\overline{s}_1, \ldots, \overline{s}_k \in \overline{\Sigma}^n$.

6.1.3 Spezielle Kostenfunktionen

Da es aufwendig ist Kostenfunktionen auf $\overline{\Sigma}_0^k$ zu definieren, betrachtet man spezielle Kostenfunktionen, die aus Kostenfunktionen auf $\overline{\Sigma}^2$ aufgebaut sind. Dabei gilt hier allerdings $w(-,-) = 0$, da dies in einer Projektion eines mehrfachen Sequenzen-Alignment zum einen nicht wirklich die Distanz zwischen den beiden betrachteten Sequenzen erhöht und zum anderen die Ähnlichkeit dadurch weder erhöht noch erniedrigt wird.

Definition 6.6 *Sei $G = ([1:k], E)$ ein (zusammenhängender) Graph auf $[1:k]$. Sei weiter $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine Kostenfunktion für ein Distanz- bzw. ein Ähnlichkeitsmaß. Dann heißt $w : \overline{\Sigma}_0^k \to \mathbb{R}$ eine* von G induzierte Kostenfunktion *für ein Distanz- bzw. Ähnlichkeitsmaß, wenn für alle $(a_1, \ldots, a_k) \in \overline{\Sigma}_0^k$ gilt:*

$$w(a_1, \ldots, a_k) = \sum_{\{i,j\} \in E} w'(a_i, a_j),$$

wobei hier $w'(-,-) := 0$ definiert ist.

Die im Folgenden definierte einfache Kostenfunktionen für mehrfache Sequenzen-Alignments sind im Prinzip alles induzierte Kostenfunktionen.

6.1.3.1 Sum-of-Pairs Cost (SP)

Eine Standardkostenfunktion ist die so genannte Sum-of-Pairs-Kostenfunktion.

Definition 6.7 *Ist G ein vollständiger Graph, so heißt die durch G induzierte Kostenfunktion* Sum-of-Pairs-Kostenfunktion *(kurz* SP-Kostenfunktion*):*

$$w(a_1, \ldots, a_k) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \tilde{w}(a_i, a_j),$$

wobei $\tilde{w} : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine gewöhnliche Kostenfunktion für ein Distanz- oder Ähnlichkeitsmaß eines paarweisen Alignments ist.

Dies impliziert den Abstand oder Ähnlichkeit für mehrfache Alignments:

$$w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{i=1}^{k} \sum_{j=i+1}^{k} \tilde{w}(\overline{s}_i, \overline{s}_j).$$

Die Berechnungszeit für ein $(a_1, \ldots, a_k) \in \overline{\Sigma}_0^k$ ist $O(k^2)$.

Wir merken hier noch an, dass wir im Folgenden meist als Kostenfunktion die oben erwähnte Sum-of-Pairs-Kostenfunktion verwenden werden. Das zugehörige Distanz bzw. Ähnlichkeitsmaß wird dann oft auch als Sum-of-Pairs-Distanz bzw. Sum-of-Pairs-Ähnlichkeit oder kurz als SP-Distanz bzw. SP-Ähnlichkeit bezeichnet.

Theorem 6.8 *Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion für ein SP-Distanzmaß d und sei $C \in \mathbb{R}_+$ so gewählt, dass $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ vermöge $w'(a, b) = C - w(a, b)$ und $w'(a, -) = \frac{C}{2} - w(a, -)$ für alle $a, b \in \Sigma$ eine Kostenfunktion für ein SP-Ähnlichkeitsmaß s ist. Dann gilt für alle $(s_1, \ldots, s_k) \in \Sigma^*$:*

$$d(s_1, \ldots, s_k) + s(s_1, \ldots, s_k) = \frac{C(k-1)}{2} \sum_{i=1}^{k} |s_i|.$$

Beweis: Im Folgenden verwenden wir folgende Bezeichnungen:

$$\begin{aligned}
M = M(\overline{s}_1, \ldots, \overline{s}_k) &= \{(i, j, \lambda) : i < j \in [1:k] \wedge \lambda \in [1 : |\overline{s}_1|] \wedge \overline{s}_{i,\lambda}, \overline{s}_{j,\lambda} \in \Sigma\}, \\
N = N(\overline{s}_1, \ldots, \overline{s}_k) &= \{(i, j, \lambda) : i < j \in [1:k] \wedge \lambda \in [1 : |\overline{s}_1|] \wedge \overline{s}_{i,\lambda} = - = \overline{s}_{j,\lambda}\}, \\
D = D(\overline{s}_1, \ldots, \overline{s}_k) &= \{(i, j, \lambda) : i < j \in [1:k] \wedge \lambda \in [1 : |\overline{s}_1|]\} \setminus (M \cup N).
\end{aligned}$$

Beachte, dass für jedes mehrfache Alignment $(\overline{s}_1, \ldots, \overline{s}_k)$ gilt, dass jedes Zeichen in den Sequenzen zweimal an einem (Mis)Match, aber nur einmal bei einer Indel-Operation beteiligt ist:

$$(k-1)\sum_{i=1}^{k}|s_i| = 2|M| + |D|.$$

Weiter gilt dann (wobei das Maximum bzw. Minimum jeweils über alle Alignments $(\overline{s}_1, \ldots, \overline{s}_k) \in \mathcal{A}(s_1, \ldots, s_k)$ gebildet wird) und M bzw. D jeweils als $M(\overline{s}_1, \ldots, \overline{s}_k)$ bzw. $D(\overline{s}_1, \ldots, \overline{s}_k)$ zu lesen sind:

$$\begin{aligned}
& s(s_1, \ldots, s_k) \\
&= \max\left\{ \sum_{(i,j,\lambda)\in M} w'(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) + \sum_{(i,j,\lambda)\in D} w'(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) \right\} \\
&= \max\left\{ \sum_{(i,j,\lambda)\in M} (C - w(\overline{s}_{i\lambda}, \overline{s}_{j\lambda})) + \sum_{(i,j,\lambda)\in D} \left(\frac{C}{2} - w'(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) \right) \right\} \\
&= \max\left\{ C\cdot|M| + \frac{C}{2}\cdot|D| - \sum_{(i,j,\lambda)\in M} w(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) - \sum_{(i,j,\lambda)\in D} w'(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) \right\} \\
&= \frac{C}{2}\sum_{i=1}^{k}(k-1)|s_i| - \min\left\{ \sum_{(i,j,\lambda)\in M} w(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) + \sum_{(i,j,\lambda)\in D} w(\overline{s}_{i\lambda}, \overline{s}_{j\lambda}) \right\} \\
&= \frac{C(k-1)}{2}\sum_{i=1}^{k}|s_i| - d(s_1, \ldots, s_k).
\end{aligned}$$

Damit ist der Satz bewiesen. ■

Ein analoger Satz gilt, wenn der durch einen Graph induzierten Kostenfunktion ein regulärer Graph zugrunde liegt.

Theorem 6.9 *Sei G ein r-regulärer Graph auf $[1:k]$. Sei $w : \overline{\Sigma}_0^2 \to \mathbb{R}_+$ eine Kostenfunktion für ein durch G induziertes Distanzmaß d und sei $C \in \mathbb{R}_+$ so gewählt, dass $w' : \overline{\Sigma}_0^2 \to \mathbb{R}$ vermöge $w'(a,b) = C - w(a,b)$ und $w'(a,-) = \frac{C}{2} - w(a,-)$ für alle $a, b \in \Sigma$ eine Kostenfunktion für ein durch G induziertes Ähnlichkeitsmaß s. Dann gilt für alle $(s_1, \ldots, s_k) \in \Sigma^*$:*

$$d(s_1, \ldots, s_k) + s(s_1, \ldots, s_k) = \frac{Cr}{2}\sum_{i=1}^{k}|s_i|.$$

Der Beweis folgt im Wesentlichen aus der folgenden Beobachtung:

$$r \sum_{i=1}^{k} |s_i| = 2|M| + |D|.$$

Hierbei sind M, D und N jetzt nur noch für die Paare $\{i, j\} \in E$ definiert. Damit wird jedes Zeichen $s_{i,\lambda}$ genau r mal auf der linken Seite aufaddiert und ist dabei zweimal in einem (Mis)Match und einmal in einer Deletion enthalten.

6.1.3.2 Fixed-Tree Cost

Die feste Baum-Kostenfunktion wird durch einen ungewurzelten Baum auf allen Sequenzen induziert.

Definition 6.10 *Ist G ein Baum mit $V(G) = [1 : k]$, so heißt die durch G induzierte Kostenfunktion* feste Baum-Kostenfunktion*:*

$$w(a_1, \ldots, a_k) = \sum_{\{i,j\} \in E} \tilde{w}(a_i, a_j),$$

wobei $\tilde{w} : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine gewöhnliche Kostenfunktion für ein Distanz- oder Ähnlichkeitsmaß eines paarweisen Alignments ist.

Die Berechnungszeit für ein $(a_1, \ldots, a_k) \in \overline{\Sigma}_0^k$ ist $O(k)$.

6.1.3.3 Center-Star Cost

Die feste Stern-Kostenfunktion wird durch einen speziellen Baum induziert, wobei alle $k - 1$ Blätter an einem inneren Knoten hängen

Definition 6.11 *Ist $G = ([1 : k], \{\{1, j\} : j \in [2 : k]\})$ ein Baum, so heißt die durch G induzierte Kostenfunktion* Stern-Kostenfunktion

$$w(a_1, \ldots, a_k) = \sum_{\{i,j\} \in E} \tilde{w}(a_i, a_j),$$

wobei $\tilde{w} : \overline{\Sigma}_0^2 \to \mathbb{R}$ eine gewöhnliche Kostenfunktion für ein Distanz- oder Ähnlichkeitsmaß eines paarweisen Alignments ist.

Die Berechnungszeit für ein $(a_1, \ldots, a_k) \in \overline{\Sigma}_0^k$ ist $O(k)$.

6.1.3.4 Consensus Cost

Hierbei handelt es sich im Wesentlichen auch um eine Stern-Kostenfunktion, wobei die Sequenzen jedoch nur den Blätter zugeordnet sind, und für die Wurzel das Zeichen ausgesucht wird, das die Stern-Kostenfunktion optimiert.

Definition 6.12 *Die* Konsensus-Kostenfunktion *ist definiert durch*

$$w(a_1, \ldots, a_k) = \text{opt} \left\{ \sum_{i=1}^{k} \tilde{w}(a_i, c) \ : \ c \in \overline{\Sigma} \right\},$$

wobei $\tilde{w} : \overline{\Sigma}_0^2 \to \mathbb{R}$ *eine gewöhnliche Kostenfunktion für ein Distanz- bzw. Ähnlichkeitsmaß eines paarweisen Alignments ist und* $\text{opt} \in \{\min \max\}$*, je nach Maß.*

Die Berechnungszeit für ein $a \in \overline{\Sigma}_0^k$ ist $O(|\Sigma| \cdot k)$.

6.1.3.5 Tree Cost

Eine Verallgemeinerung der Konsensus-Kostenfunktion ist die allgemeine Baum-Kostenfunktion. Hierbei hat jeder innere Knoten Grad genau drei und man sucht die beste Belegung der inneren Knoten mit Zeichen aus $\overline{\Sigma}$.

Definition 6.13 *Sei* $([1 : 2k-2], E)$ *ein freier Baum, wobei alle inneren Knoten den Grad 3 besitzen und die inneren Knoten durch* $([k+1 : 2k-2])$ *bezeichnet sind. Die* allgemeine Baum-Kostenfunktion *ist definiert durch*

$$w(a_1, \ldots, a_k) = \text{opt} \left\{ \sum_{\{i,j\} \in E} \tilde{w}(a_i, a_j) \ : \ (a_{k+1}, \ldots, a_{2k-1}) \in \overline{\Sigma}^{k-2} \right\},$$

wobei $\tilde{w} : \overline{\Sigma}_0^2 \to \mathbb{R}$ *eine gewöhnliche Kostenfunktion für ein Distanz- oder Ähnlichkeitsmaß eines paarweisen Alignments ist und* $\text{opt} \in \{\min \max\}$*, je nach Maß.*

Die Berechnungszeit für ein $(a_1, \ldots, a_k) \in \overline{\Sigma}_0^k$ ist $O(|\Sigma| \cdot k^2)$. Der Beweis der Berechnungszeit ist etwas aufwendiger und bedarf einer geschickten dynamischen Programmierung.

In Varianten dieser Kostenfunktion werden auch andere Bäume mit größerem Grad zugelassen, oder es wird auch über alle möglichen Topologien optimiert (oder einer eingeschränkten Teilmenge davon).

19.11.19

6.2 Dynamische Programmierung

In diesem Abschnitt verallgemeinern wir die Methode der Dynamischen Programmierung von paarweisen auf mehrfache Sequenzen-Alignments. Aufgrund der großen Laufzeit ist dieses Verfahren aber eher von theoretischem Interesse bzw. nur für wenige oder kurze Sequenzen sinnvoll einsetzbar.

6.2.1 Rekursionsgleichungen

Im Folgenden sei $D[\vec{x}]$ für $\vec{x} = (x_1, \ldots, x_k) \in \mathbb{N}_0^k$ der Wert eines optimalen mehrfachen Sequenzen-Alignments für $s_{1,1} \cdots s_{1,x_1}$, $s_{2,1} \cdots s_{2,x_2}$, $\ldots$, $s_{k-1,1} \cdots s_{k-1,x_{k-1}}$ und $s_{k,1} \cdots s_{k,x_k}$. Der folgende Satz lässt sich analog wie für das paarweise Sequenzen-Alignment beweisen.

Theorem 6.14 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und sei $w : \overline{\Sigma}_0^k \to \mathbb{R}_+$ eine Kostenfunktion für ein Distanzmaß. Es gilt für $\vec{x} \in \bigtimes_{i=1}^{k} [0 : |s_i|]$:*

$$D[\vec{x}] := \min \left\{ D[\vec{x} - \vec{\eta}] + w(\vec{x} \bullet \vec{\eta}) \ : \ \vec{\eta} \in [0:1]^k \setminus \vec{0} \wedge \vec{\eta} \leq \vec{x} \right\}.$$

Hierbei ist $\min \emptyset = 0$ und $(x_1, \ldots, x_k) \bullet (\eta_1, \ldots, \eta_k) = (s_{1,x_1} \bullet \eta_1, \ldots, s_{k,x_k} \bullet \eta_k)$ mit $a \bullet 0 := -$ und $a \bullet 1 := a$ für $a \in \Sigma$.

Bei der Implementierung kann die Abfrage $\vec{\eta} \geq \vec{x}$ etwas aufwendig werden, so dass man wie im Falle paarweiser Alignments die Randbedingungen oft explizit definiert. Nun stellt sich noch die Frage, wie die Anfangswerte für $\vec{x} \in [0:n]^k \setminus [1:n]^k$ eines solchen mehrfachen Sequenzen-Alignments aussehen. Dies wird am Beispiel von drei Sequenzen im folgenden Bild in Abbildung 6.1 erklärt.

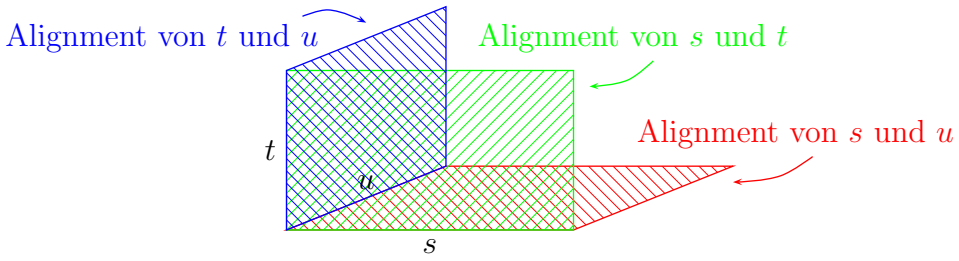


Abbildung 6.1: Skizze: Anfangswerte für ein 3-faches Sequenzen-Alignment

Für ein 3-faches Sequenzen-Alignment von $s^{(1)}$, $s^{(2)}$ und $s^{(3)}$ mit $n_i = |s(i)|$ für alle $i \in [1:3]$ wollen wir noch explizit die Rekursionsformeln und Anfangsbedingungen angeben. Es gilt dann für ein globales mehrfaches Sequenzen-Alignment mit $(i,j,k) \in [1:n_1] \times [1:n_2] \times [1:n_3]$:

$$
\begin{aligned}
D[0,0,0] &= 0, \\
D[i,0,0] &= D[i-1,0,0] + w(s_i^{(1)},-,-), \\
D[0,j,0] &= D[0,j-1,0] + w(-,s_j^{(2)},-), \\
D[0,0,k] &= D[0,0,k-1] + w(-,-,s_k^{(3)}), \\
D[i,j,0] &= \min \left\{ \begin{array}{c} D[i-1,j,0] + w(s_i^{(1)},-,-), \\ D[i,j-1,0] + w(-,s_j^{(2)},-), \\ D[i-1,j-1,0] + w(s_i^{(1)},s_j^{(2)},-) \end{array} \right\}, \\
D[i,0,k] &= \min \left\{ \begin{array}{c} D[i-1,0,k] + w(s_i^{(1)},-,-), \\ D[i,0,k-1] + w(-,-,s_k^{(3)}), \\ D[i-1,0,k-1] + w(s_i^{(1)},-,s_k^{(3)}) \end{array} \right\}, \\
D[0,j,k] &= \min \left\{ \begin{array}{c} D[0,j-1,k] + w(-,s_j^{(2)},-), \\ D[0,j,k-1] + w(-,-,s_k^{(3)}), \\ D[0,j-1,k-1] + w(-,s_j^{(2)},s_k^{(3)}) \end{array} \right\}, \\
D[i,j,k] &= \min \left\{ \begin{array}{c} D[i-1,j,k] + w(s_i^{(1)},-,-), \\ D[i,j-1,k] + w(-,s_j^{(2)},-), \\ D[i,j,k-1] + w(-,-,s_k^{(3)}), \\ D[i-1,j-1,k] + w(s_i^{(1)},s_j^{(2)},-), \\ D[i-1,j,k-1] + w(s_i^{(1)},-,s_k^{(3)}), \\ D[i,j-1,k-1] + w(-,s_j^{(2)},s_k^{(3)}), \\ D[i-1,j-1,k-1] + w(s_i^{(1)},s_j^{(2)},s_k^{(3)}) \end{array} \right\}.
\end{aligned}
$$

Hierbei ist $w : \overline{\Sigma}^3 \to \mathbb{R}_+$ die zugrunde gelegte Kostenfunktion. Für das Sum-of-Pairs-Maß gilt dann: $w(x,y,z) = w'(x,y) + w'(x,z) + w'(y,z)$, wobei $w' : \overline{\Sigma}^2 \to \mathbb{R}_+$ die Standard-Kostenfunktion für Paare ist.

Die Übertragung auf lokale mehrfache Alignments oder auf Ähnlichkeitsmaße sei dem Leser zur Übung überlassen.

6.2.2 Zeitanalyse

Für die Zeitanalyse nehmen wir an, dass $|s_i| = \Theta(n)$ für alle $i \in [1:k]$ gilt. Wir überlegen uns zuerst, dass die gesamte Tabelle $\Theta(n^k)$ viele Einträge besitzt. Für jeden Eintrag ist eine Minimumsbildung von $2^k - 1$ Elemente durchzuführen, wobei sich jeder Wert in Zeit $\Theta(k^2)$ berechnen lässt (wenn wir das SP-Maß zugrunde legen). Insgesamt ist der Zeitbedarf also $O(k^2 \cdot 2^k \cdot n^k)$.

Dies ist leider exponentiell und selbst für moderat große k inakzeptabel. Für $k = 3$ ist dies gerade noch verwendbar, für größere k in der Regel unpraktikabel (außer die Sequenzen sind sehr kurz).

Leider gibt es für die Berechnung eines mehrfachen Sequenzen-Alignment kein effizientes Verfahren. Man kann nämlich nachweisen, das die Entscheidung, ob eine gegebene Menge von Sequenzen, ein mehrfaches Alignment besitzt, das eine vorgegebene Distanz unterschreitet (oder Ähnlichkeit überschreitet), $\mathcal{NP}$-hart ist. Für das SP-Maß und einige andere kann sogar die $\mathcal{APX}$-Vollständigkeit nachgewiesen werden.

Auch für andere Kostenfunktionen ist die dynamische Programmierung nicht wesentlich schneller, da in jedem Fall $O(\gamma(k) \cdot 2^k \cdot n^k)$ Operationen benötigt werden, wobei $\gamma(k)$ die Anzahl der Operationen beschreibt um die induzierte Kostenfunktion für ein k-Tupel zu ermitteln.

6.2.3 Forward Dynamic Programming

Im Folgenden wollen wir eine Heuristik zeigen, mit der sich die dynamische Programmierung oft etwas schneller durchführen lässt. Wir werden wie beim paarweisen Alignment für konkave Lückenstrafen vom backward dynamic programming auf das forward dynamic programming umschalten. Wenn also die Zelle $D[\vec{x}]$ vollständig berechnet wurde, werden dann die davon abhängigen Zellen $\vec{x} + \vec{\eta}$ (für $\vec{\eta} \in \{0,1\}^k \setminus \{\vec{0}\}$) mittels der folgenden Gleichung aktualisiert:

$$D[\vec{x} + \vec{\eta}] := \min\{D[\vec{x} + \vec{\eta}], D[\vec{x}] + w((\vec{x} + \vec{\eta}) \bullet \vec{\eta})\}.$$

Ersetzt man in dieser Gleichung $\vec{x} + \vec{\eta}$ durch $\vec{x}$ sowie $\vec{x}$ durch $\vec{x} - \vec{\eta}$, dann erhält man die übliche Form der Rekursionsgleichung im backward dynamic programming. Dieser Trick mithilfe anderer kann die dynamische Programmierung beschleunigen, wenn auch nicht im worst-case.

6.2.4 Relevanz-Test

Bei der geschickteren Berechnung der Tabelle wollen wir nur die Zellen aktualisieren, die in einer optimalen Lösung auch wirklich benötigt werden können.

Theorem 6.15 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und $(\overline{s}_1, \ldots, \overline{s}_k)$ ein optimales k-faches Alignment von $s_1, \ldots, s_k$. Gilt $d(s_1, \ldots, s_k) = w(\overline{s}_1, \ldots, \overline{s}_k) \leq C$ für das zur Kostenfunktion w gehörige SP-Distanzmaß d, dann gilt für alle $\mu < \nu \in [1:k]$*

$$w(\overline{s}_\mu, \overline{s}_\nu) \leq C_{\mu\nu},$$

wobei

$$C_{\mu\nu} := C - \sum_{\substack{i<j\in[1:k]\\(i,j)\neq(\mu,\nu)}} d(s_i, s_j) = C' + d(s_\mu, s_\nu),$$

wobei $C' = C - \sum_{i<j\in[1:k]} d(s_i, s_j)$.

Beweis: Nach Voraussetzung gilt:

$$w(\overline{s}_1, \ldots, \overline{s}_k) \leq C.$$

Also gilt nach Definition des SP-Maßes

$$\sum_{i<j\in[1:k]} w(\overline{s}_i, \overline{s}_j) = w(\overline{s}_1, \ldots, \overline{s}_k) \leq C.$$

Somit gilt auch:

$$\sum_{\substack{i<j\in[1:k]\\(i,j)\neq(\mu,\nu)}} w(\overline{s}_i, \overline{s}_j) \leq C - w(\overline{s}_\mu, \overline{s}_\nu).$$

Weiter gilt dann

$$\sum_{\substack{i<j\in[1:k]\\(i,j)\neq(\mu,\nu)}} d(s_i, s_j) \leq \sum_{\substack{i<j\in[1:k]\\(i,j)\neq(\mu,\nu)}} w(\overline{s}_i, \overline{s}_j) \leq C - w(\overline{s}_\mu, \overline{s}_\nu)$$

und damit die Behauptung. ∎

Wenn wir also eine obere Schranke für die Distanz des mehrfachen Sequenzen-Alignments haben, können wir für die projizierten paarweisen Alignments ebenfalls eine obere Schranke herleiten.

Dazu müssen wir zuerst noch definieren, welche Zellen der Tabelle der dynamischen Programmierung relevant sind. Dazu benötigen wir erst noch ein paar weitere altbekannte Ergebnisse.

Notation 6.16 *Seien $s \in \Sigma^n$ sowie $t \in \Sigma^m$ und d ein Distanzmaß. Dann ist*

$$\begin{aligned} P(i,j) &:= d(s_1 \cdots s_i, t_1 \cdots t_j), \\ S(i,j) &:= d(s_{i+1} \cdots s_n, t_{j+1} \cdots t_m). \end{aligned}$$

Definition 6.17 *Seien $s \in \Sigma^n$ und $t \in \Sigma^m$ sowie $i \in [1:n]$ und $j \in [1:m]$. Ein Alignment $(\overline{s}, \overline{t}) \in \mathcal{A}(s,t)$* respektiert den Schnitt *(i,j), wenn es ein $\ell \in [1:|\overline{s}|]$ gibt, so dass gilt*

$$\begin{aligned} ((\overline{s}_1, \ldots, \overline{s}_\ell), (\overline{t}_1, \ldots, \overline{t}_\ell)) &\in \mathcal{A}(s_1 \cdots s_i, t_1 \cdots t_j), \\ ((\overline{s}_{\ell+1}, \ldots, \overline{s}_{|\overline{s}|}), (\overline{t}_{\ell+1}, \ldots, \overline{t}_{|\overline{t}|})) &\in \mathcal{A}(s_{i+1} \cdots s_n, t_{j+1} \cdots t_m). \end{aligned}$$

Dann gilt das folgende Lemma.

Lemma 6.18 *Seien $s \in \Sigma^n$ sowie $t \in \Sigma^m$ und d ein Distanzmaß. Die minimale Distanz eines Alignments von s und t, das den Schnitt (i,j) für $i \in [1:n]$ und $j \in [1:m]$ respektiert, ist genau $P(i,j) + S(i,j)$.*

Der einfache Beweis sei dem Leser zur Übung überlassen.

Notation 6.19 *Seien $s^{(1)}, \ldots, s^{(k)} \in \Sigma^*$ und d ein Distanzmaß. Dann ist*

$$\begin{aligned} P_{\mu,\nu}(i,j) &:= d(s_1^{(\mu)} \cdots s_i^{(\mu)}, s_1^{(\nu)} \cdots s_j^{(\nu)}), \\ S_{\mu,\nu}(i,j) &:= d(s_{i+1}^{(\mu)} \cdots s_{|s_\mu|}^{(\mu)}, s_{j+1}^{(\nu)} \cdots s_{|s_\nu|}^{(\nu)}). \end{aligned}$$

Wenn wir also eine obere Schranke C für die Distanz eines optimalen Sequenzen-Alignments haben, können wir eine Zelle $\vec{x}$ als *relevant* definieren, für die gilt:

$$P_{\mu,\nu}(x_\mu, x_\nu) + S_{\mu,\nu}(x_\mu, x_\nu) \leq C_{\mu,\nu}.$$

Wir werden also nur noch für relevante Zellen die Tabelle der Dynamische Programmierung ausfüllen. Für die Bestimmung relevanter Zellen sind nur die $O(k^2)$ quadratische Tabellen $P_{\mu,\nu}$ und $S_{\mu,\nu}$ der Größe $O(|s_\mu| \cdot |s_\nu|)$ nötig. Ferner benötigen wir nur eine obere Schranke C für die optimale Distanz des k-fachen Sequenzen-Alignments.

```
Carrillo_Lipman (char[] (s_1, ..., s_k); int k, int C)
begin
    forall (μ < ν ∈ [1 : k]) do
        compute P_{μ,ν}[], S_{μ,ν}[];
        compute C_{μ,ν} := C − Σ_{(i,j)≠(μ,ν)} d(s_i, s_j); ;        /* = C' + P_μν(|s_μ|, |s_ν|) */
    x⃗ := 0⃗;
    D[0⃗] := 0;
    heap h := empty();
    h.insert(x⃗);
    while ((not h.is_empty()) && (x⃗ ≠ (|s_1|, ..., |s_k|))) do
        x⃗ := h.delete_min(); ;                    /* the lexicographic smallest */
        if (P_{μ,ν}(x_μ, x_ν) + S_{μ,ν}(x_μ, x_ν) ≤ C_{μ,ν} for all μ < ν ∈ [1 : k]) then
            forall (η⃗ ∈ {0,1}^k \ {0⃗}) do
                if (not h.is_member(x⃗ + η⃗)) then
                    h.insert(x⃗ + η⃗);
                    D[x⃗ + η⃗] := D[x⃗] + w((x⃗ + η⃗) • η⃗);
                else
                    D[x⃗ + η⃗] := min{D[x⃗ + η⃗], D[x⃗] + w((x⃗ + η⃗) • η⃗)};
    return D[|s_1|, ..., |s_k|];
end
```

Abbildung 6.2: Algorithmus: Dynamische Programmierung nach Carrillo-Lipman

6.2.5 Algorithmus nach Carrillo und Lipman

Für den Algorithmus definieren wir zuerst nur den Ursprung $\vec{0}$ als relevant und speichern ihn in einem Heap ab. Wenn wir ein kleinstes Element (gemäß der lexikographischen Ordnung) aus dem Heap entnehmen, testen wir zuerst, ob die Zelle relevant ist. Wenn Sie relevant ist, werden alle Nachfolger, die noch nicht im Heap sind, in den Heap aufgenommen, andernfalls nicht. In Abbildung 6.2 ist der Algorithmus von Carrillo und Lipman im Detail aufgelistet.

21.11.19

Zur Erinnerung: ein Heap ist eine Datenstruktur, der folgende Operationen zur Verfügung stellt:

insert: Fügt ein Element in den Heap ein.

delete_min: Liefert das kleinste Element des Heaps und löscht es aus dem Heap.

is_member: Liefert genau dann true, wenn das Element bereits im Heap gespeichert ist.

is_empty: Liefert genau dann true, wenn der Heap kein Element enthält.

Nun zur Korrektheit des Algorithmus von Carrillo und Lipman. Zuerst halten wir fest, dass wir nur Elemente im Heap halten, deren Wert noch nicht als endgültig berechnet gelten und direkte Nachfolger von relevanten Zellen sind. Da wir jeweils die lexikographisch kleinste Zelle $\vec{x}$ aus dem Heap entfernen, wissen wir dass alle relevanten lexikographisch kleineren Zellen ihre Aktualisierung bereits an $\vec{x}$ mitgeteilt haben müssen. Also ist $D[\vec{x}]$ korrekt berechnet und wir können die Zelle auf ihre Relevanz hin testen. Ist $\vec{x}$ relevant, so schickt sie ihrerseits ihre Aktualisierungen an alle direkten Nachfolger $\vec{x} + \vec{\eta}$ mit $\vec{\eta} \in \{0,1\}^k \setminus \{\vec{0}\}$.

6.2.6 Laufzeit des Carrillo-Lipman-Algorithmus

Die Berechnung der $\Theta(k^2)$ paarweisen Sequenzen-Alignments für die Tabellen P und S kosten Zeit $O(k^2n^2)$. Die Berechnung der $\Theta(k^2)$ Konstanten für den Relevanz-Test kostet ebenfalls $O(k^2n^2)$. Mit einer geschickten Implementierung geht dies auch in Zeit $O(k^2)$. Hierzu bemerkt man zuerst dass, $d(s_i, s_j) = P_{ij}(|s_i|, |s_j|)$. Dann berechnet man in zeit $O(k^2)$ die Konstante $C' := C - \sum_{i<j\in[1:k]} d(s_i, s_j)$. Die gesuchten Werte $C_{\mu,\nu}$ könne nun jeweils mit einer weiteren Addition bestimmt werden: $C_{\mu,\nu} = C' + P_{\mu\nu}(|s_\mu|, |s_\nu|)$.

Weiterhin lassen sich Heaps so implementieren, dass sich die insert- und delete_min-Operationen in Zeit $O(\log(N))$ realisieren lassen, wobei N die maximale Anzahl der Elemente im Heap ist. In unserem Falle ist also $\log(N) = O(k \log(n))$. Die restlichen Operationen auf Heaps können in konstanter Zeit ausgeführt werden (wobei man bei is_member bei der Implementierung vorsichtig sein muss).

Sei im Folgenden als N die Anzahl der inspizierten Zellen, wobei offensichtlich $N \leq n^k$ gilt. Dann wird die while-Schleife genau N-mal durchlaufen. Der Zeitbedarf der Heap-Operationen einer while-Schleife ist durch $O(\log(N))$ beschränkt. Der Relevanz-Test von $\vec{x}$ erfordert Zeit $O(k^2)$. Im positiven Falle werden $O(2^k)$ Nachfolger betrachtet. Der Zeitbedarf für jeden Nachfolger ist konstant mit Ausnahme der Heap-Operation insert, die wir allerdings schon vorher abgeschätzt hatten, und der Berechnung der Kostenfunktion w, die wir mit k^2 angeben. Also beträgt der Gesamtzeitbedarf $O((k^2 \cdot 2^k + \log(N))N)$

Theorem 6.20 *Der Algorithmus von Carrillo und Lipman berechnet mit einem Zeitbedarf von $O((k^2 \cdot 2^k + \log(N))N)$ die Distanz eines optimalen mehrfachen Alignments der Sequenzen $s_1, \ldots, s_k \in \Sigma^*$, wobei N die Anzahl der betrachteten Zellen ist.*

Dies ergibt im worst-case wieder die asymptotisch gleiche Laufzeit wie beim backward dynamic programming Ansatz.

6.2.7 Mehrfaches Alignment durch kürzeste Pfade

Wie beim paarweisen Sequenzen-Alignment, kann man sich auch das mehrfache Sequenzen-Alignment als ein Wege-Problem in einem Graphen vorstellen.

Definition 6.21 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und sei $w : \overline{\Sigma}_0^k \to \mathbb{R}$ eine Kostenfunktion. Der* Alignment-Graph *für die Sequenzen $s_1, \ldots, s_k$ ist definiert als gewichteter gerichteter $G(s_1, \ldots, s_k) = (V, E, \gamma)$:*

$$\begin{aligned} V &:= \bigtimes_{i=1}^{k} [0 : |s_i|], \\ E &:= \left\{ (\vec{x}, \vec{x} + \vec{\eta}) \;:\; \vec{x}, \vec{x} + \vec{\eta} \in V \wedge \vec{\eta} \in \{0,1\}^k \setminus \{\vec{0}\} \right\}, \\ \gamma(\vec{x}, \vec{x} + \vec{\eta}) &:= w((\vec{x} + \vec{\eta}) \bullet \vec{\eta}). \end{aligned}$$

Für ein Distanz- bzw. Ähnlichkeitsmaß suchen wir in $G(s_1, \ldots, s_k)$ also einen kürzesten bzw. längsten Weg von $\vec{0}$ zu $(|s_1|, \ldots, |s_k|)$. Im Allgemeinen ist es jedoch $\mathcal{NP}$-hart einen längsten Pfad in einem Graphen zu finden. Wir haben hier jedoch spezielle Graphen, in denen dies effizient möglich ist.

Definition 6.22 *Ein* azyklischer gerichteter Graph*, kurz* DAG *(für directed acyclic graph), ist ein gerichteter Graph, der keinen gerichteten Kreis enthält.*

Beobachtung 6.23 *Der Alignment-Graph für $s_1, \ldots, s_k \in \Sigma^*$ ist ein DAG.*

Definition 6.24 *Sei $G = (V, E)$ ein DAG, dann heißt eine Aufzählung der Knoten $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ mit $n := |V|$ eine* topologische Aufzählung *der Knoten von G, wenn für jede Kante $(v_{\pi(i)}, v_{\pi(j)}) \in E$ gilt, dass $i < j$.*

Lemma 6.25 *Sei $G = (V, E)$ ein DAG, dann kann eine topologische Aufzählung der Knoten in Zeit $O(|V| + |E|)$ berechnet werden.*

Durchläuft man das forward dynamic Programming mittels der topologischen Aufzählung, so lassen sich kürzte und längste Wege von einer gegebenen Quelle zu einer gegeben Senke in linearer Zeit berechnen, siehe Abbildung 6.3.

extreme_path_DAG (DAG $G = (V, E, \gamma)$)

```
begin
    // (v_1, ..., v_n) is a topological ordering of V
    // v_1 is the source and v_n the sink
    D[v_1] := 0;
    for (i := 2; i ≤ n; i++) do
        D[v_i] := − opt{−∞, +∞};
    for (i := 1; i < n; i++) do
        forall ((v_i, v_j) ∈ E) do    /* i.e. j > i */
            D[v_j] := opt{D[v_j], D[v_i] + γ(v_i, v_j)};
    return D[v_n];
end
```

Abbildung 6.3: Algorithmus: Bestimmung extremaler Pfade in DAGs

Wenn wir diesen Algorithmus auf unser Alignment-Problem anwenden, haben wir auch hier wieder das Problem, dass wir alle n^k Knoten und $\Theta(2^k n^k)$ Kanten ablaufen müssen, was zu viel ist. Besser können wir hier mit Dijkstras Algorithmus werden. Hierfür noch kurz zur Erinnerung die Datenstruktur PQ einer *Priority Queue* oder *Vorrang-Warteschlange*, die die folgenden Operationen unterstützt:

is_empty: Liefert genau dann true, wenn die Priority Queue leer ist.

is_member: Liefert genau dann true, wenn das Element bereits im der Priority Queue gespeichert ist.

insert: Fügt ein Element mit dem zugehörigen Schlüssel in die Priority Queue ein.

delete_min Liefert bezüglich der vorgegebenen totalen Ordnung ein minimales Element auf den Schlüsseln und entfernt dieses Element aus der Priority Queue.

decrease_key: Erniedrigt den Schlüssel des angegebenen Elements in der Priority Queue auf den mitgegeben Wert (sofern nötig). Dabei wird vorausgesetzt, dass das Element bereits in der Priority Queue enthalten ist.

Wir zitieren hier nur das Resultat, das man erhält, wenn man Priority Queues mithilfe von Fibonacci-Heaps implementiert. Für die Details verweisen wir auf die einschlägige Literatur (oder auch auf das Skript zu Algorithmischen Bioinformatik: Bäume und Graphen).

Theorem 6.26 *Beginnend mit einem leeren Fibonacci-Heap kann eine Folge von ℓ Operationen* `is_empty`*,* `is_member`*,* `insert`*,* `decrease_key` *und* `delete_min` *in Zeit $O(\ell + k \log(n))$ ausgeführt werden, wobei n die maximale Zahl von Elementen im Fibonacci-Heap und $k \leq \ell$ die Anzahl der* `delete_min` *Operationen ist.*

```
Dijkstra_Alignment(char[] s_1, ..., s_k, int k, int C)
begin
    forall (μ < ν ∈ [1 : k]) do
        compute P_{μ,ν}[], S_{μ,ν}[];
        compute C_{μ,ν} := C − Σ_{(i,j)≠(μ,ν)} d(s_i, s_j) [= C' + P_{μν}(|s_μ|, |s_ν|)];
    x⃗ := 0⃗;
    D[0⃗] := 0;
    PQ q := empty();
    q.insert(x⃗, 0);
    while ((not q.is_empty()) && (x⃗ ≠ (|s_1|, ..., |s_k|))) do
        x⃗ := q.delete_min();
        if (P_{μ,ν}(x_μ, x_ν) + S_{μ,ν}(x_μ, x_ν) ≤ C_{μ,ν} for all μ < ν ∈ [1 : k]) then
            forall (η⃗ ∈ {0,1}^k \ {0⃗}) do
                if (not q.is_member(x⃗ + η⃗)) then
                    D[x⃗ + η⃗] := D[x⃗] + w((x⃗ + η⃗) • η⃗);
                    q.insert(x⃗ + η⃗, D[x⃗ + η⃗]);
                else
                    D[x⃗ + η⃗] := min{D[x⃗ + η⃗], D[x⃗] + w((x⃗ + η⃗) • η⃗)};
                    q.decrease_key(x⃗ + η⃗, D[x⃗ + η⃗]);
    return (D[|s_1|, ..., |s_k|]);
end
```

Abbildung 6.4: Dijkstra-Algorithmus mit Relevanz-Test für mehrfaches Alignment

Damit können wir den in Abbildung 6.4 angegebenen Algorithmus implementieren. Lassen wir den Relevanz-Test weg (erste If-Anweisung sowie die zugehörigen Vorberechnungen) und lassen die forall-Schleife über alle Nachfolgerknoten laufen, erhalten wir den bekannten Dijkstra-Algorithmus.

Theorem 6.27 *Der Dijkstra-Algorithmus basierend auf Fibonacci-Heaps berechnet die kürzesten Pfade von einem bestimmten Knoten zu allen anderen Knoten in einem Graphen $G = (V, E)$ in Zeit $O(|E| + |V| \log(|V|))$.*

Damit können wir einen kürzesten Weg (jedoch keinen längsten Weg, da die Korrektheit von Dijkstras Algorithmus nur mit nichtnegativen Kantengewichten gewährleistet ist, die bei Ähnlichkeitsmaßen allerdings auftreten) im Alignment-Graphen in Zeit $O((k^2 \cdot 2^k + \log(N))N)$ berechnen (wobei $N \leq n^k$). Für DAGs mit nichtnegativen Kantengewichten können wir die Idee von Dijkstra recyceln, indem wir

den nächsten Knoten nicht nach der topologischen Sortierung auswählen, sondern den mit dem bislang kürzesten Pfad, da dieser dann (aufgrund der Nichtnegativität der Kantengewichte) korrekt berechnet sein muss. Auch hier können wir zusätzlich den Relevanz-Test einsetzen, d.h. wir propagieren Ergebnisse nur von relevanten Knoten weiter. Ein Knoten $\vec{x}$ ist hier relevant, wenn $P_{i,j}(x_i, x_j) + S_{ij}(x_i, x_j) \leq C_{ij}$ für alle $i < j \in [1 : k]$ gilt.

Theorem 6.28 *Der Algorithmus von Dijkstra basierend auf Fibonacci-Heaps und dem Relevanz-Test von Carrillo und Lipman berechnet die Distanz eines optimalen mehrfachen Alignments von $s_1, \ldots, s_k \in \Sigma^*$ in Zeit $O(k^2n^2 + (k^2 \cdot 2^k + \log(N))N)$, wobei N die Anzahl der betrachteten Zellen ist.*

Wir wollen noch erwähnen, dass es noch andere Varianten des Relevanz-Tests gibt und dass es andere heuristische Methoden gibt, um einen kürzesten bzw. längsten Pfad in einem DAG zu finden, wie beispielsweise den A^*-Algorithmus.

6.3 Divide-and-Conquer-Alignment

In diesem Abschnitt wollen wir einen anderen exakten Algorithmus vorstellen, der sich mittels geeigneter Heuristiken beschleunigen lässt, das so genannte *Divide-and-Conquer-Alignment* (kurz *DCA*).

6.3.1 Divide-and-Conquer-Ansatz

Die Grundidee des Divide-and-Conquer-Ansatzes ist im Wesentlichen dieselbe wie bei Quicksort. Wir teilen zuerst jede Sequenz geschickt in zwei Teile auf, so dass wir jeweils alle Präfixe bzw. alle Suffixe rekursiv alignieren. Die zwei so erhaltenen mehrfachen Alignments werden dann zu einem zusammengesetzt. Für hinreichend kurze Sequenzen kann man das mehrfache Sequenzen-Alignment natürlich optimal lösen. Die meiste Energie muss man dabei in die geschickte Aufteilung in Präfix und Suffix für jede Sequenz stecken. Für eine Skizze dieser Idee siehe Abbildung 6.5.

Wenn man alle möglichen Aufteilungsschritte betrachtet, liefert dies natürlich eine optimale Lösung. Halbiert man die erste Sequenz (hier ist die Schnittposition noch relativ egal) und testet für die restlichen $k-1$ Sequenzen alle andere Möglichkeiten für das Aufteilen in Präfix und Suffix, so muss man hier allein schon etwa $O(n^{k-1})$ Schnittpositionen betrachten.

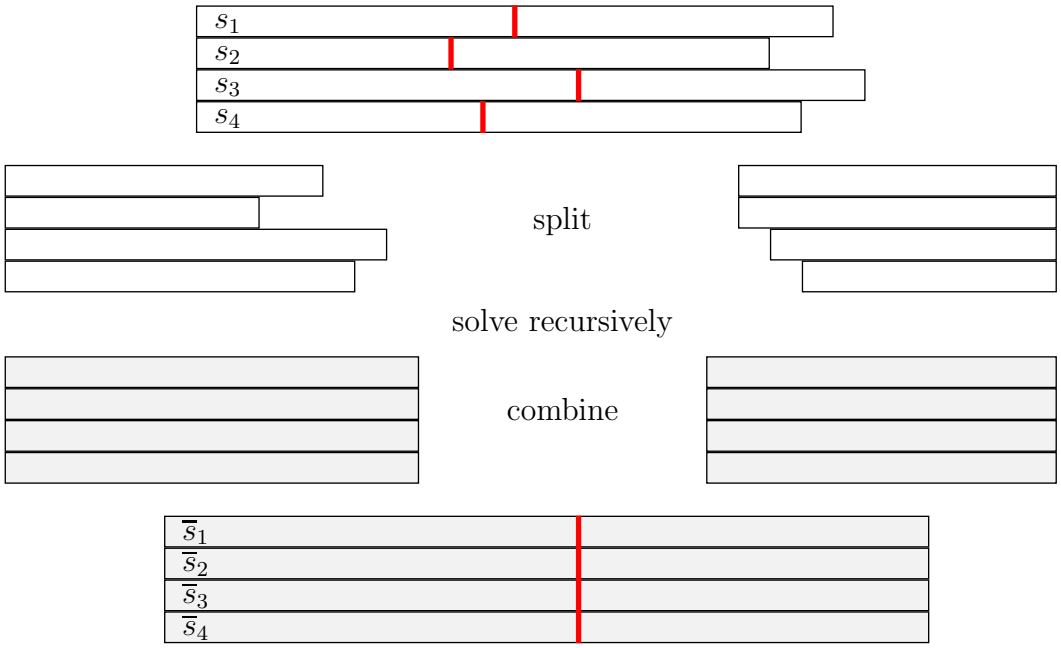


Abbildung 6.5: Skizze: Divide-and-Conquer-Alignment

6.3.2 C-optimale Schnittpositionsfamilien

Wir müssen uns nun also überlegen, wie wir gute Schnittpositionen effizient finden können. Hierbei orientieren wir uns wieder an den SP-Kosten der einzelnen Projektionen von mehrfachen Alignments.

Definition 6.29 *Seien $s \in \Sigma^n$ und $t \in \Sigma^m$ sowie $i \in [1:n]$ und $j \in [1:m]$. Sei w die Kostenfunktion für ein Distanzmaß d. Die* Zusatzkosten *eines Schnitts (i,j) für s und t sind definiert durch:*

$$C_{s,t}(i,j) := \min\left\{w(\overline{s},\overline{t}) \;:\; (\overline{s},\overline{t}) \in \mathcal{A}_{(i,j)}(s,t)\right\} - d(s,t),$$

wobei $\mathcal{A}_{(i,j)}(s,t)$ die Menge aller Alignments von s und t ist, die den Schnitt (i,j) respektieren. Die Matrix $C_{s,t}$ wird auch Zusatzkostenmatrix *genannt.*

Die Zusatzkostenmatrix lässt sich wie folgt leicht berechnen:

$$C_{s,t}(i,j) = P_{s,t}(i,j) + S_{s,t}(i,j) - d(s,t).$$

Hierbei gilt natürlich wieder die Identität $d(s,t) = P_{s,t}(|s|,|t|) = S_{s,t}(0,0)$. Aus den so definierten Zusatzkosten lassen sich (gewichtete) mehrfache SP-Zusatzkosten für

26.11.19

eine feste mehrfache Schnittposition einer gegebenen Menge von Sequenzen definieren.

Definition 6.30 *Seien* $s_1, \ldots, s_k \in \Sigma^*$, $(c_1, \ldots, c_k) \in \bigtimes_{i=1}^{k} [0 : |s_i|]$ *und* $\gamma(i,j) \in \mathbb{R}$ *für* $i < j \in [1 : k]$. *Die* (gewichteten) mehrfachen SP-Zusatzkosten *einer* k*-fachen Schnittposition* $(c_1, \ldots, c_k)$ *ist definiert durch:*

$$C(c_1, \ldots, c_k) := \sum_{i=1}^{k} \sum_{j=i+1}^{k} \gamma(i,j) C_{s_i,s_j}(c_i, c_j).$$

Wir werden in der Regel als Gewicht immer $\gamma(i,j) = 1$ verwenden, die Gewichte können aber auch in Abhängigkeit der entsprechenden paarweise Alignment-Scores gewählt werden. Nun können wir unter allen Schnittpositionen optimale, d.h. solche mit minimalen SP-Zusatzkosten, definieren.

Definition 6.31 *Seien* $s_1, \ldots, s_k \in \Sigma^*$ *und* $c \in [0 : |s_1|]$. *Eine Familie von* k*-fachen Schnittpositionen* $\mathcal{C} = \{(c_1, \ldots, c_k) : c_1 = c\}$ *heißt* C-optimal *mit Respekt zu* c*, wenn die (gewichteten) mehrfachen SP-Zusatzkosten minimal sind. Mit*

$$\begin{aligned} C^*(c) &= \min \{C(c_1, \ldots, c_k) : c_1 = c \wedge c_i \in [0 : |s_i|]\}, \\ \mathcal{C}(c) &= \{(c_1, \ldots, c_k) : C(c_1, \ldots, c_k) = C^*(c) \wedge c_1 = c \wedge c_i \in [0 : |s_i|]\} \end{aligned}$$

bezeichnet man die minimalen SP-Zusatzkosten bzw. die zugehörigen optimalen Schnittpositionen.

Wir sollten an dieser Stelle noch darauf hinweisen, dass eine C-optimale Schnittposition nicht notwendigerweise zu einem optimalen mehrfachen Alignment führen muss. Dies wird auch im folgenden Beispiel deutlich (siehe auch Abbildung 6.6 und Abbildung 6.7). Wir betrachten hierzu die Sequenzen $s_1 = \text{AGGA}$, $s_2 = \text{ACCG}$ und $s_3 = \text{CGA}$. In der Abbildung 6.6 sind für jedes Paar von Sequenzen die beiden Distanzmatrizen für die Präfixe $(P_{\mu,\nu})$ und die Suffixe $(S_{\mu,\nu})$ dargestellt. Hierbei wurde eine Lücke und ein Mismatch jeweils mit dem Wert 1 bestraft, d.h. $w(a,b) = 1 - \delta_{a,b}$ für $a, b \in \overline{\Sigma}$. Daraus wurden dann in der letzten Spalte der Abbildung 6.6 die zugehörigen paarweisen Zusatzkostenmatrizen berechnet. Man beachte, dass die Distanz eines optimalen Alignments beispielsweise in der Matrix $S_{\mu,\nu}(0,0)$ abgelesen werden kann.

Für die Berechnung der Matrix $C(2,i,j)$ (ganz unten in Abbildung 6.6) werden dann nur die beiden mittleren (in der Abbildung 6.6 fett gedruckten) Zeilen der Zusatzkostenmatrizen C_{s_1,s_2} und C_{s_1,s_3} benötigt, da ja gilt:

$$C(2,i,j) := C_{s_1,s_2}(2,i) + C_{s_1,s_3}(2,j) + C_{s_2,s_3}(i,j).$$

P	A	C	C	G
	0	1	2	3	4
A					
	1	0	1	2	3
G					
	2	1	1	2	2
G					
	3	2	2	2	2
A					
	4	3	3	3	3

S	A	C	C	G
	3	3	3	3	4
A					
	4	3	2	2	3
G					
	4	3	2	1	2
G					
	3	3	2	1	1
A					
	4	3	2	1	0

C	A	C	C	G
	0	1	2	3	5
A					
	2	0	0	1	3
G					
	3	**1**	**0**	**0**	**1**
G					
	3	2	1	0	0
A					
	5	3	2	1	0

P	C	G	A
	0	1	2	3
A				
	1	1	2	2
G				
	2	2	1	2
G				
	3	3	2	2
A				
	4	4	3	2

S	C	G	A
	2	2	3	4
A				
	1	1	2	3
G				
	1	0	1	2
G				
	2	1	0	1
A				
	3	2	1	0

C	C	G	A
	0	1	3	5
A				
	0	0	2	3
G				
	1	**0**	**0**	**2**
G				
	3	2	0	1
A				
	5	4	2	0

P	C	G	A
	0	1	2	3
A				
	1	1	2	2
C				
	2	1	2	3
C				
	3	2	2	3
G				
	4	3	2	3

S	C	G	A
	3	4	3	4
A				
	2	3	3	3
C				
	1	2	2	2
C				
	2	1	1	1
G				
	3	2	1	0

C	C	G	A
	0	2	2	4
A				
	0	1	2	2
C				
	0	0	1	2
C				
	2	0	0	1
G				
	4	2	0	0

c=2	C	G	A
	4	5	5	9
A				
	2	2	3	5
C				
	1	**0**	1	4
C				
	3	**0**	**0**	3
G				
	6	3	1	3

Abbildung 6.6: Beispiel: Die Matrizen $P_{\mu,\nu}$, $S_{\mu,\nu}$ und $C(2, c_2, c_3)$ für die Sequenzen $s_1 = AGGA$, $s_2 = ACCG$ und $s_3 = CGA$ mit $C^*(2) = 0$

Man sieht leicht, dass die mehrfachen SP-Zusatzkosten für C-optimale Schnittpositionen mit Respekt zu 2 genau 0 sind und dass es genau drei solcher optimaler Schnittpositionen gibt.

In der Abbildung 6.7 sind dann jeweils für diese Schnittpositionen die beiden zugehörigen rekursiv zu bestimmenden (und hier jeweils optimal ausgewählten) Alignments angegeben. Dabei sieht man, dass die optimale Schnittposition $(2, 3, 2)$ kein optimales mehrfaches Alignment generieren muss.

```
A G - | G A        A G | - G A        A - G | G A
A C C | G -        A C | C G -        A C C | G -
- C - | G A        - C | - G A        - C G | - A
```

Abbildung 6.7: Beispiel: Beste mehrfache Alignments für die C-optimalen Schnittpositionen $\mathcal{C}(2) = \{(2, 3, 1), (2, 2, 1), (2, 3, 2)\}$ mit den zugehörigen SP-Distanzen 8, 8 und 10.

Es stellt sich jetzt nur die Frage, wie man solche C-optimalen Schnittpositionen effizient finden kann. Man kann natürlich wieder alle möglichen Schnittpositionen für ein gegebenes $c \in [0 : |s_1|]$ aufzählen und das minimale auswählen. Durch simple Aufzählung aller Schnittpositionen erhält man das folgende Lemma.

Lemma 6.32 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Für ein gegebenes $c \in [0 : |s_1|]$ lässt sich eine C-optimale Menge von Schnittpositionen in Zeit $O(k^2n^2 + k^2n^{k-1})$ berechnen.*

Beweis: Der Beweis ist sehr einfach. Für die Berechnung der k^2 Matrizen C_{s_i,s_j} für $i, j \in [1 : k]$ benötigt man jeweils Zeit $O(n^2)$. Das Aufzählen aller weiteren Schnittpositionen $(c_2, \ldots, c_k) \in \bigtimes_{i=2}^{k}$ ist in Zeit $O(n^{k-1})$ realisierbar. Zur Bestimmung jedes $C(c_1, \ldots, c_k)$ sind dabei $O(k^2)$ Additionen nötig. ■

Man kann an dieser Stelle auch weitere Heuristiken bei der Aufzählung der Schnittpositionen anwenden und mehrfache Schnittpositionen, bei denen zu große SP-Zusatzkosten zu erwarten sind, bei der Aufzählung einfach auslassen. Für weitere Beschleunigungstricks verweisen wir auf die einschlägige Literatur.

6.3.3 Der DCA-Algorithmus und seine Laufzeit

Nimmt man die Bestimmung von C-optimalen Schnittpositionen als Black-Box an, so erhält man den folgenden in Abbildung 6.8 angegebenen Divide-and-Conquer-Alignment-Algorithmus.

DCA (char[] $s_1, \ldots, s_k$, int L)

```
begin
    if (max{|s_1|, ..., |s_k|} ≤ L) then
        return MSA(s_1, ..., s_k);
    else
        c_1 := ⌈|s_1|/2⌉;
        let (c_1, ..., c_k) ∈ C(c_1, ..., c_k);
        return DCA(s_1^1 ··· s_{c_1}^1, ..., s_1^k ··· s_{c_k}^k, L)·
               DCA(s_{c_1+1}^1 ··· s_{|s_1|}^1, ..., s_{c_k+1}^k ··· s_{|s_k|}^k, L);
end
```

Abbildung 6.8: Algorithmus: Skelett des DCA-Algorithmus

Um die Laufzeit des DCA-Algorithmus unter Verwendung C-optimaler Schnittpositionen so einfach wie mögliche analysieren zu können, nehmen wir an, dass wir für jede Menge von Sequenzen, die alle kürzer als L sind, ein optimales Verfahren anwenden (beispielsweise mittels dynamischer Programmierung).

Weiterhin nehmen wir an, dass alle Sequenzen genau die Länge $n = L \cdot 2^D$ besitzen und dass jeweils genau in der Hälfte der Sequenzen geschnitten wird. Damit beträgt die Rekursionstiefe genau D.

Damit ergibt sich die folgende Rekursionsgleichung (in gewissem Sinne ähnlich wie bei der Analyse von Mergesort):

$$\begin{aligned}
T(n,L,D) &= O\left(\sum_{i=0}^{D-1} 2^i \left(k^2\left(\frac{n}{2^i}\right)^2 + k^2\left(\frac{n}{2^i}\right)^{k-1}\right) + 2^D k^2 2^k L^k\right) \\
&= O\left(k^2 n^2 \sum_{i=0}^{D-1} \frac{1}{2^i} + k^2 n^{k-1} \sum_{i=0}^{D-1} \frac{1}{(2^{(k-2)})^i} + \frac{n}{L} k^2 2^k L^k\right) \\
&\quad \text{da } k > 2 \text{ und für jedes } x \in (0,1) \text{ gilt } \textstyle\sum_{i=0}^{\infty} x^i = \Theta(1) \\
&= O\left(k^2 n^2 + k^2 n^{k-1} + k^2 2^k n L^{k-1}\right) \\
&= O\left(k^2 n^{k-1} + k^2 2^k n L^{k-1}\right).
\end{aligned}$$

Falls L klein genug ist (z.B. $L = O(\sqrt{n})$), ist die Laufzeit sogar durch $O(k^2 n^{k-1})$ beschränkt. Damit haben wir den folgenden Satz motiviert.

Theorem 6.33 *Der DCA-Alignment benötigt bei Verwendung C-optimaler Schnittpositionen und Berechnung optimaler mehrfacher Alignment für Sequenzen kürzer als L für ein mehrfachen Sequenzen-Alignments für Sequenzen $s_1, \ldots, s_k \in \Sigma^*$ mit $|s_i| = O(n)$ Zeit $O(k^2 n^{k-1} + k^2 2^k n L^{k-1})$ und Platz $O(k^2 n^2 + L^k)$.*

Für den Platzbedarf tragen wir jetzt noch nach:

$$S(n, L, D) \leq \max\left\{\frac{kn}{2^d} + \frac{k^2n^2}{4^d} \;:\; d \in [0 : D-1]\right\} + L^k \leq \Theta(k^2n^2 + L^k).$$

Der Term $kn/2^d + k^2n^2/4^d$ stammt dabei für die Berechnung der C-optimalen Schnittpositionen auf jedem Rekursionslevel (Platz kann wiederverwendet werden) und der Term L^k für die Berechnung eines optimalen Alignments am Rekursionsanfang. Die Kosten zur Memorierung der rekursiven Aufrufen sind dabei nicht berücksichtigt, da sie von geringerer Größenordnung sind (man muss sich in jedem Level nur die Schnittpositionen merken).

6.4 Center-Star-Approximation

Da die exakte Lösung eines mehrfachen Sequenzen-Alignments, wie eben bereits angemerkt, in aller Regel sehr schwer zu finden ist, wollen wir uns jetzt mit Approximationen beschäftigen.

6.4.1 Mit Bäumen konsistente Alignments

Dazu definieren wir zuerst mit Bäumen konsistente Alignments.

Definition 6.34 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und sei $T = ([1 : k], E)$ ein Baum. Ein mehrfaches Sequenzen-Alignment $(\overline{s}_1, \ldots, \overline{s}_k)$ für $s_1, \ldots, s_k$ ist* konsistent mit T*, wenn jedes induzierte paarweise Sequenzen-Alignment $(\tilde{s}_i, \tilde{s}_j)$ für $(i, j) \in E$ optimal ist.*

In der Abbildung 6.9 ist ein mit einem Baum konsistentes mehrfaches Alignment für den dort angegebenen Baum und die dort angegebenen Sequenzen illustriert.

$$\begin{array}{ccccc} A & G & G & T & - \\ A & G & C & T & - \\ A & G & - & T & A \\ A & G & A & T & A \\ A & G & A & T & - \\ - & G & A & T & A \end{array}$$

Abbildung 6.9: Skizze: mehrfaches Alignment, das mit einem Baum konsistent ist

Nun wollen wir uns überlegen, ob man solche mit vorgegebenen Bäumen konsistente Alignments effizient berechnen kann.

Lemma 6.35 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und sei $T = ([1:k], E)$ ein Baum. Ein mehrfaches Sequenzen-Alignment für $s_1, \ldots, s_k$, das konsistent zu T ist, lässt sich in Zeit $O(kn(n+k))$ konstruieren, wobei $|s_i| = \Theta(n)$ für $i \in [1:k]$.*

Beweis: Wir führen den Beweis mittels Induktion über k und nehmen hierfür an, dass sich die Laufzeit durch $cnk(n+k)$ für ein geeignetes c beschränkt ist.

Induktionsanfang ($k = 2$): Hierfür ist die Aussage trivial.

Induktionsschritt ($k \to k+1$): Ohne Beschränkung der Allgemeinheit sei s_{k+1} ein Blatt von T und s_k adjazent zu s_{k+1} in T.

Nach Induktionvoraussetzung existiert ein mehrfaches Alignment $(\overline{s}_1, \ldots, \overline{s}_k)$ für $s_1, \ldots, s_k$, das konsistent zu T ist und in Zeit $ckn(n+k)$ konstruiert werden kann. Ein solches mehrfaches Alignment ist in Abbildung 6.10 illustriert.

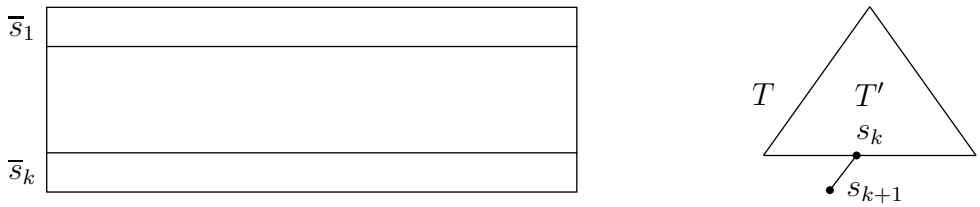


Abbildung 6.10: Skizze: Induktionsvoraussetzung

Wir berechnen nun ein optimales paarweises Alignment $(\tilde{s}_k, \tilde{s}_{k+1})$ von s_k mit s_{k+1} in Zeit cn^2. Dann erweitern wir das mehrfaches Sequenzen-Alignment um das Alignment $(\tilde{s}_k, \tilde{s}_{k+1})$ wie in der Abbildung 6.11 angegeben. Dazu müssen wir im Wesentlichen nur die Zeile $\tilde{s}_{k+1}$ hinzufügen, wobei wir in der Regel sowohl in $\tilde{s}_{k+1}$ als auch im

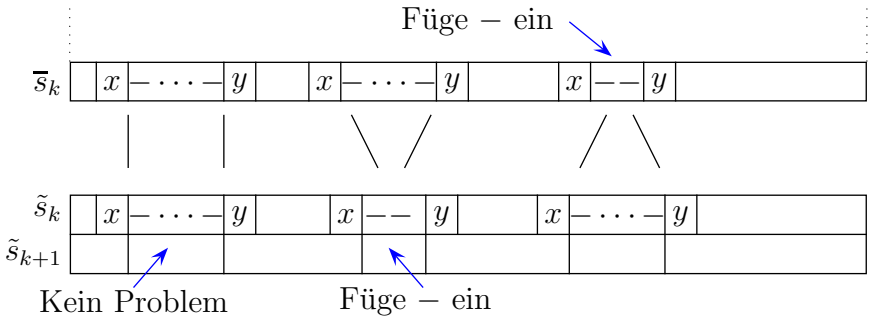


Abbildung 6.11: Skizze: Erweiterung des mehrfaches Sequenzen-Alignments

bereits konstruierten mehrfachen Sequenzen-Alignment Leerzeichen einfügen müssen. Diese bestimmen sich im Wesentlichen aus dem Paar $(\overline{s}_k, \tilde{s}_k)$. Wir fügen im Prinzip so wenig wie möglich Leerzeichen hinzu, so dass $w(\overline{s}_k, \tilde{s}_k) = 0$ wird.

Für die Rechenzeit benötigen wir für das Alignment cn^2 Operationen (hinreichend großes c vorausgesetzt). Im schlimmsten Fall, müssen wir in das bestehende mehrfache Sequenzenalignment n Spalten mit je k Leerzeichen (also kn Leerzeichen) einfügen und in das paarweise Alignment ebenfalls maximal kn Leerzeichen (da in $\overline{s}_k$ maximal kn Leerzeichen auftreten können). Dies ist in Zeit cnk möglich (sofern $c > 2$), wenn die Spalten des Alignments in einer linearen Liste verwaltet werden. Damit erhalten wir

$$\begin{aligned} T(n, k+1) &\leq cnk(n+k) + cn^2 + ckn \\ &= cnk(n+k) + cn(n+k) \\ &= cn(k+1)(n+k) \\ &\leq cn(k+1)(n+k+1) \end{aligned}$$

und der Induktionsschluss ist vollzogen. ■

Somit können wir mehrfache Alignments, die zu Bäumen konsistent sind, sehr effizient konstruieren. Im Weiteren wollen wir uns damit beschäftigen, wie gut solche mehrfachen Alignments sind.

6.4.2 Die Wahl des Baumes

Nun wollen wir ausgehend von dem eben vorgestellten Verfahren zur Konstruktion von mehrfachen Sequenzen-Alignments mit Hilfe von Bäumen einen Algorithmus vorstellen, der ein mehrfaches Sequenzen-Alignment bestimmter Güte konstruiert. Bei der Center-Star-Methode besteht die Idee darin, den Baum T so zu wählen, dass er einen Stern darstellt. Das Problem besteht nun darin, welche Sequenz als Zentrum des Sterns gewählt werden soll. Dazu werden wir diejenige Sequenz wählen, die den kleinsten Abstand zu allen anderen besitzt.

Definition 6.36 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Eine Sequenz s_c mit $c \in [1:k]$ heißt* Center-String*, wenn $\sum_{j=1}^{k} d(s_c, s_j)$ minimal ist.*

6.4.3 Approximationsgüte

Sei $M_c = (\overline{s}_1, \ldots, \overline{s}_k)$ das mehrfache Sequenzen-Alignment, das zu T (dem Stern mit Zentrum s_c) konsistent ist und wie vorhin beschrieben konstruiert ist. Dann

28.11.19

bezeichne im Folgenden $D(s_i, s_j) := D_{M_c}(s_i, s_j) := w(\overline{s}_i, \overline{s}_j)$ den Wert des durch M_c projizierten Alignments $(\overline{s}_i, \overline{s}_j)$ für s_i und s_j. Es gilt dann offensichtlich:

$$\begin{aligned} D(s_i, s_j) &\geq d(s_i, s_j), \\ D(s_c, s_j) &= d(s_c, s_j), \\ D(M_c) &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} D(s_i, s_j). \end{aligned}$$

Hierbei bezeichnen wir im Folgenden für ein mehrfaches Sequenzen-Alignment M die zugehörige SP-Distanz $D(M) := w(M)$ bzgl. eines Distanzmaßes d basierend auf der Kostenfunktion w.

Wir erinnern zuerst noch einmal an folgende elementare Beziehung für ein optimales paarweises Alignment $(\overline{s}, \overline{t})$ für s und t:

$$d(s,t) = w(\overline{s}, \overline{t}) = \sum_{i=1}^{|\overline{s}|} w(\overline{s_i}, \overline{t_i}).$$

Lemma 6.37 *Sei d ein Distanzmaß, dessen zugehörige Kostenfunktion eine Metrik ist, dann gilt für jede Wahl von s_c als Zentrum mit $c \in [1:k]$:*

$$D(s_i, s_j) \leq D(s_i, s_c) + D(s_c, s_j) = d(s_i, s_c) + d(s_c, s_j).$$

Beweis: Der Beweis folgt unmittelbar aus der Abbildung 6.12 unter der Berück-

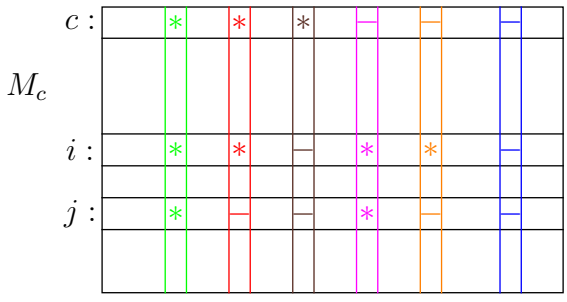


Abbildung 6.12: Skizze: Beweis der Dreiecksungleichung

sichtigung, dass die Dreiecksungleichung $w(\overline{s}_{i,p}, \overline{s}_{j,p}) \leq w(\overline{s}_{i,p}, \overline{s}_{c,p}) + w(\overline{s}_{c,p}, \overline{s}_{j,p})$ für w gilt (mit $w(-,-) = 0$) und dass D der Summe der Spalten entspricht. Der zweite Teil folgt aus der Tatsache, dass wir ein zu einem Stern mit Zentrum s_c konsistentes Alignment betrachten. ■

Sei $M^* = (\tilde{s}_1, \ldots, \tilde{s}_k)$ ein optimales mehrfaches Sequenzen-Alignment für $s_1, \ldots, s_k$ und sei $D^*(s_i, s_j) = w(\tilde{s}_i, \tilde{s}_j)$ der Wert des durch M^* projizierten paarweisen Alignments für s_i und s_j. Dann gilt:

$$d(s_1, \ldots, s_k) = D(M^*) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} D^*(s_i, s_j).$$

Theorem 6.38 *Seien $s_1, \ldots, s_k \in \Sigma^*$, d das SP-Distanzmaß, das auf einer metrischen Kostenfunktion basiert, und $T = (S, E)$ ein Stern, dessen Zentrum der Center-String von $s_1, \ldots, s_k$ ist. Sei M_c ein mehrfaches Sequenzen-Alignment für $s_1, \ldots, s_k$, das zu T konsistent ist, und M^* ein optimales mehrfaches Sequenzen-Alignment von $s_1, \ldots, s_k$. Dann gilt:*

$$\frac{D(M_c)}{D(M^*)} \leq 2 - \frac{2}{k}.$$

Beweis: Zuerst eine Vereinfachung:

$$\begin{aligned} D(M^*) &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} D^*(s_i, s_j) = \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D^*(s_i, s_j), \\ D(M_c) &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} D(s_i, s_j) = \frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D(s_i, s_j). \end{aligned}$$

Dies folgt aus der Tatsache, dass $D(s_i, s_i) = 0 = D^*(s_i, s_i)$ sowie $D(s_i, s_j) = D(s_j, s_i)$ und $D^*(s_i, s_j) = D^*(s_j, s_i)$ (da die Kostenfunktion eine Metrik ist).

Dann gilt für den Quotienten:

$$\begin{aligned} \frac{D(M_c)}{D(M^*)} &= \frac{\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D(s_i, s_j)}{\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D^*(s_i, s_j)} \\ &\quad \text{da } D(s_i, s_i) = 0 \\ &= \frac{\sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(s_i, s_j)}{\sum_{i=1}^{k} \sum_{j=1}^{k} D^*(s_i, s_j)} \\ &\quad \text{mit Lemma 6.37 und } D^*(s_i, s_j) \geq d(s_i, s_j) \\ &\leq \frac{\sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} [d(s_i, s_c) + d(s_c, s_j)]}{\sum_{i=1}^{k} \sum_{j=1}^{k} d(s_i, s_j)} \\ &\quad \textstyle\sum_{j=1}^{k} d(s_i, s_j) \text{ wird minimal für } i = c \text{ nach Wahl von } s_c \\ &\leq \frac{(k-1) \sum_{i=1}^{k} d(s_i, s_c) + (k-1) \sum_{j=1}^{k} d(s_c, s_j)}{\sum_{i=1}^{k} \sum_{j=1}^{k} d(s_c, s_j)} \end{aligned}$$

$$
\begin{aligned}
&= \frac{2(k-1)}{k} \cdot \frac{\sum_{i=1}^{k} d(s_i, s_c)}{\sum_{j=1}^{k} d(s_c, s_j)} \\
&= \frac{2k-2}{k} \\
&= 2 - \frac{2}{k}.
\end{aligned}
$$

Damit ist die Behauptung bewiesen. ■

6.4.4 Laufzeit für Center-Star-Methode

Wie groß ist nun die Laufzeit der Center-Star-Methode? Für die Bestimmung des Centers müssen wir für jede Sequenz die Summe der paarweisen Distanzen zu den anderen Sequenzen berechnen. Dies kostet pro Sequenz $(k-1) \cdot O(n^2) = O(kn^2)$. Für alle Sequenzen ergibt sich daher eine Laufzeit von $O(k^2n^2)$. Für die Konstruktion des mehrfachen Sequenzen-Alignments, das konsistent zum Stern mit dem gewählten Center-String als Zentrum ist, benötigen wir nach Lemma 6.35 nur noch $O(kn(n+k))$. Der Gesamtzeitbedarf ist also $O(k^2n^2)$, wobei die meiste Zeit für die Auswahl des Zentrums verbraucht wurde.

Theorem 6.39 *Die Center-Star-Methode liefert für k Sequenzen der Länge $O(n)$ eine $(2-\frac{2}{k})$-Approximation für ein mehrfaches Sequenzen-Alignment unter dem SP-Distanzmaß, das auf einer metrischen Kostenfunktion basiert, in Zeit $O(k^2n^2)$.*

Man kann ein mehrfaches Sequenzen-Alignment statt aus der Berechnung paarweiser Sequenzen-Alignments mit einem Zentrum auch aus optimalen ℓ-fachen Sequenzen-Alignments mit einem gemeinsamen Zentrum konstruieren. Dann erhält man einen wesentlich aufwendigeren Algorithmus mit einer etwas besseren Approximationsgüte, allerdings ist die Laufzeit dann auch exponentiell in ℓ.

Theorem 6.40 *Für k Sequenzen liefert die verallgemeinerte Center-Star-Methode mit $\ell \in [2:k]$ eine $(2-\frac{\ell}{k})$-Approximation für ein mehrfaches Sequenzen-Alignment in Zeit $O(k^{\ell+1}(2^k + k \cdot A(\ell, n)))$ oder $O(k^3 \cdot A(2\ell+5, n))$, wobei $A(\ell, n)$ die Laufzeit ist, um ℓ Sequenzen der Länge n optimal zu alignieren.*

Für die Details verweisen wir auf die Originalliteratur von Bineet Bafna, Eugene L. Lawler und Pavel A. Pevzner.

6.4.5 Randomisierte Varianten

Wir wollen im Folgenden zeigen, dass man nur einige Zentren ausprobieren muss und dass dann bereits der beste der ausprobierten Zentrum schon fast eine 2-Approximation liefert. Wir können also die Laufzeit noch einmal senken. Für den Beweis benötigen wir die folgende Notation und das darauf folgende Lemma.

Notation 6.41 *Seien $s_1, \ldots, s_k \in \Sigma^*$, dann bezeichne $M(i) := \sum_{j=1}^{k} d(s_i, s_j)$ und $M := \min\{M(i) : i \in [1:k]\}$.*

Mit Hilfe dieser Notation können wir zeigen, dass es „relativ viele“ Sterne gibt, deren „Güte“ nicht all zu weit vom Optimum abweicht.

Lemma 6.42 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und $1 < r \in \mathbb{R}$, dann existieren mehr als $\lfloor \frac{k}{r} \rfloor$ Sterne s_i mit $M(i) \leq \frac{2r-1}{r-1} M$.*

Beweis: Wir berechnen zuerst den Mittelwert von $M(i)$. Sei dazu $c \in [1:k]$ mit $M = M(c)$.

$$
\begin{aligned}
\frac{1}{k}\sum_{i=1}^{k} M(i) &= \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{mit } d(s_i, s_i) = 0 \\
&= \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} d(s_i, s_j) \\
&\quad \text{mit Hilfe der Dreiecksungleichung} \\
&\leq \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} [d(s_i, s_c) + d(s_c, s_j)] \\
&\leq \frac{1}{k}\sum_{i=1}^{k} d(s_i, s_c) \sum_{\substack{j=1 \\ j\neq i}}^{k} 1 + \frac{1}{k}\sum_{j=1}^{k} d(s_c, s_j) \sum_{\substack{i=1 \\ i\neq j}}^{k} 1 \\
&\quad \text{da } s_c \text{ Zentrum eines optimalen Sterns ist} \\
&= \frac{2}{k}(k-1)M \\
&< 2M
\end{aligned}
$$

Wir führen den Beweis des Lemmas mit Hilfe eines Widerspruchs. Dazu nehmen wir also an, dass maximal $\lfloor \frac{k}{r} \rfloor$ Sterne mit $M(i) \leq \frac{2r-1}{r-1} M$ existieren. Dann gilt:

$$
\begin{aligned}
2M &> \frac{1}{k} \sum_{i=1}^{k} M(i) \\
&\quad \text{mit Hilfe der Widerspruchsannahme} \\
&\geq \frac{1}{k} \left(\left\lfloor \frac{k}{r} \right\rfloor M + \left(k - \left\lfloor \frac{k}{r} \right\rfloor \right) \frac{2r-1}{r-1} M \right) \\
&\quad \text{da offensichtlich } M \leq \tfrac{2r-1}{r-1} M \text{ ist} \\
&\quad \text{(siehe dazu auch Abbildung 6.13)} \\
&\geq \frac{1}{k} \left(\frac{k}{r} M + \left(k - \frac{k}{r} \right) \frac{2r-1}{r-1} M \right) \\
&= M \left(\frac{1}{r} + \left(1 - \frac{1}{r} \right) \frac{2r-1}{r-1} \right) \\
&= M \left(\frac{1}{r} + \frac{r-1}{r} \cdot \frac{2r-1}{r-1} \right) \\
&= 2M.
\end{aligned}
$$

Also gilt $2M > 2M$, was offensichtlich der gewünschte Widerspruch ist. ■

Zum Beweis der Abschätzung bei Vergrößerung des Parameters α einer Konvex-Kombinationen $\alpha \cdot x + (1-\alpha) \cdot y$ für $x < y$ mit $\alpha \in [0,1]$ siehe auch die folgende Skizze in Abbildung 6.13 (hier mit $x = M$, $y = \frac{2r-1}{r-1} M$, $\alpha = \frac{1}{k} \lfloor \frac{k}{r} \rfloor$). Dies bedeutet, das bei einer Vergrößerung von α die Konvex-Kombination kleiner wird.

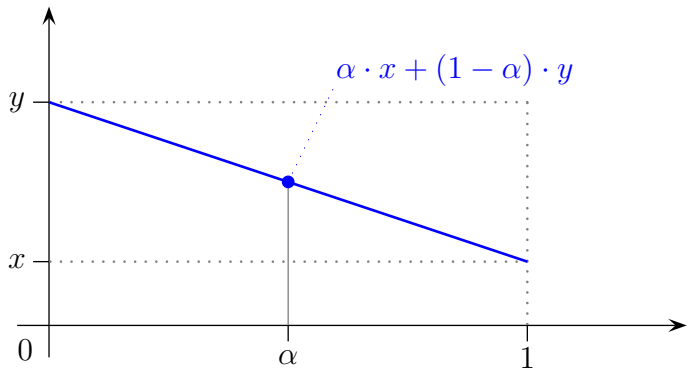


Abbildung 6.13: Skizze: Konvex-Kombination von x und y mit $x < y$ bei Vergrößerung von α

Damit können wir nun zeigen, dass es relative viele Sterne mit einem kleinen Score geben muss.

Lemma 6.43 *Seien $s_1, \ldots, s_k \in \Sigma^*$ mit $M(1) \leq M(2) \leq \cdots \leq M(k)$ und sei $r \in \mathbb{R}$ mit $r > 1$. Sei weiter M^* ein optimales mehrfaches Sequenzen-Alignment für $s_1, \ldots, s_k$ (bzgl. des SP-Distanzmaßes) und M_c ein Center-Star-Alignment mit Zentrum s_c für $s_1, \ldots, s_k$, wobei $c = \lfloor \frac{k}{r} \rfloor + 1$. Dann gilt $\frac{D(M_c)}{D(M^*)} < \frac{2r-1}{r-1} = 2 + \frac{1}{r-1}$.*

Beweis: Sei $c := \lfloor \frac{k}{r} \rfloor + 1$, dann gilt aufgrund der Voraussetzungen des vorherigen Lemmas 6.42:

$$M(c) = \varepsilon \cdot M \quad \text{mit} \quad \varepsilon \in \left[1, \frac{2r-1}{r-1}\right].$$

Damit gilt für die Approximationsgüte:

$$\begin{aligned}
\frac{D(M_c)}{D(M^*)} &= \frac{\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k} D(s_i, s_j)}{\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k} D^*(s_i, s_j)} \\
&\quad \text{da } D(s_j, s_j) = 0 \\
&= \frac{\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} D(s_i, s_j)}{\sum_{i=1}^{k}\sum_{j=1}^{k} D^*(s_i, s_j)} \\
&\quad \text{da } D(s_i, s_j) \leq d(s_i, s_c) + d(s_c, s_j) \text{ und } D^*(s_i, s_j) \geq d(s_i, s_j) \\
&\leq \frac{\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} [d(s_i, s_c) + d(s_c, s_j)]}{\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j)} \\
&= \frac{2\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} d(s_i, s_c)}{\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j)} \\
&= \frac{2\sum_{i=1}^{k} d(s_i, s_c)\sum_{\substack{j=1 \\ j\neq i}}^{k} 1}{\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j)} \\
&\quad \text{da } \sum_{i=1}^{k} d(s_i, s_c) = M(c) \\
&= \frac{2(k-1)M(c)}{\sum_{i=1}^{k} M(i)} \\
&\quad \text{da } M(i) \geq M \text{ für alle } i \text{ und } M(i) \geq \varepsilon M \text{ für } i \geq c \geq \lfloor \tfrac{k}{r} \rfloor + 1 \\
&\leq \frac{2(k-1)\cdot\varepsilon\cdot M}{\lfloor \frac{k}{r} \rfloor \cdot M + (k - \lfloor \frac{k}{r} \rfloor)\cdot\varepsilon\cdot M} \\
&\quad \text{da offensichtlich } M \leq \varepsilon M \text{ gilt} \\
&\leq \frac{2(k-1)\cdot\varepsilon\cdot M}{\frac{k}{r}\cdot M + (k - \frac{k}{r})\cdot\varepsilon\cdot M}
\end{aligned}$$

$$
\begin{aligned}
& \text{Kürzen mit } \tfrac{\varepsilon \cdot M}{r} \\
= \; & \frac{2r(k-1)}{k/\varepsilon + k(r-1)} \\
& \text{da } \varepsilon \leq \tfrac{2r-1}{r-1} \\
\leq \; & \frac{2r(k-1)}{k\frac{r-1}{2r-1} + k(r-1)} \\
= \; & \frac{2r(k-1)(2r-1)}{k(r-1) + k(r-1)(2r-1)} \\
= \; & \frac{2r(k-1)(2r-1)}{2rk(r-1)} \\
= \; & \frac{k-1}{k} \cdot \frac{2r-1}{r-1} \\
< \; & \frac{2r-1}{r-1} \\
= \; & 2 + \frac{1}{r-1}.
\end{aligned}
$$

Damit ist das Lemma bewiesen. ■

Aus dem Satz und dessen Beweis (mit $M(c') \leq M(c) = M(\lfloor \frac{k}{r} \rfloor + 1) = \varepsilon \cdot M$) folgt sofort das folgende Korollar.

Korollar 6.44 *Seien $s_1, \ldots, s_k \in \Sigma^*$ mit $M(1) \leq M(2) \leq \cdots \leq M(k)$ und sei $r \in \mathbb{R}$ mit $r > 1$. Sei weiter M^* ein optimales mehrfaches Sequenzen-Alignment für $s_1, \ldots, s_k$ (bzgl. des SP-Distanzmaßes) und $M_{c'}$ ein Center-Star-Alignment mit Zentrum $s_{c'}$ für $s_1, \ldots, s_k$, wobei $c' \in [1 : \lfloor \frac{k}{r} \rfloor + 1]$. Dann gilt $\frac{D(M_{c'})}{D(M^*)} < \frac{2r-1}{r-1} = 2 + \frac{1}{r-1}$.*

03.12.19

Daraus kann man sofort den folgenden Satz ableiten.

Theorem 6.45 *Seien $s_1, \ldots, s_k \in \Sigma^*$ und sei $1 < r \in \mathbb{R}$. Wählt man ℓ Sterne (d.h. ihre Zentren) zufällig aus $s_1, \ldots, s_k$ aus, dann ist das beste mehrfache Sequenzen-Alignment, das von diesen Sternen generiert wird, eine $(2 + \frac{1}{r-1})$-Approximation mit der Wahrscheinlichkeit größer gleich $1 - \left(\frac{r-1}{r}\right)^\ell$.*

Beweis: Seien $(c_1, \ldots, c_\ell) \in [1 : k]^\ell$ zufällig gewählte Zahlen. Sei X_i für $i \in [1 : \ell]$ eine binäre Zufallsvariable (auch eine so genannte Indikatorvariable), die genau

dann 1 ist, wenn die Wahl von s_{c_i} als Center-String eine Approximationsgüte von $(2 + \frac{1}{r-1})$ oder besser liefert. Es gilt mit Korollar 6.44:

$$\text{Ws}[X_i = 0] \leq \frac{k - (\lfloor \frac{k}{r} \rfloor + 1)}{k} \leq \frac{k - \frac{k}{r}}{k} = \frac{r-1}{r}.$$

Betrachten wir nun die Zufallsvariable $X = \sum_{i=1}^{\ell} X_i$. Wir berechnen nun die Wahrscheinlichkeit, dass kein Center-String eine hinreichend gute Approximationsgüte liefert, d.h. $\text{Ws}[X = 0]$. Da die X_i stochastisch unabhängig sind, gilt:

$$\text{Ws}[X = 0] = \text{Ws}[X_1 = 0 \wedge \cdots \wedge X_\ell = 0] = \prod_{i=1}^{\ell} \text{Ws}[X_i = 0] \leq \left(\frac{r-1}{r}\right)^{\ell}.$$

Damit erhalten wir sofort

$$Ws[X \geq 1] = 1 - \text{Ws}[X = 0] \geq 1 - \left(\frac{r-1}{r}\right)^{\ell}$$

und damit ist der Satz bewiesen. ■

Betrachten wir für ein Beispiel eine Approximationsgüte von 2,1 (bzw. 2,2), dann ist $r = 11$ (bzw. $r = 6$). Damit wird mit einer Wahrscheinlichkeit von mehr als 99% die gewünschte Approximation erhalten, müssen wir $\ell = 48$ (bzw. $\ell = 25$) Sequenzen als Zentrum der Sterne zufällig auswählen.

Sicherlich werden wir dabei ein Ziehen ohne Zurücklegen durchführen. Die vorherige Analyse für das Ziehen mit Zurücklegen ist jedoch einfacher und die Wahrscheinlichkeit beim Ziehen ohne Zurücklegen, ein geeignetes Zentrum zu wählen, ist nur größer.

Korollar 6.46 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Für $1 < r \in \mathbb{R}$ sei $C(r)$ die Anzahl von Sternen, die zufällig gewählt werden müssen, bis das beste mehrfache Alignment, das mit der Center-Star-Methode und den gewählten Zentren erstellt wird, eine $(2+\frac{1}{r-1})$-Approximation ist. Dann ist $\mathbb{E}[C(r)] \leq r$.*

Beweis: Wir haben im letzten Lemma nachgewiesen, dass sich die Wahl eines guten Center-Strings durch eine Binomialverteilung zum Parameter $p \geq 1-(1-1/r) = 1/r$ abschätzen lässt. Nun wollen wir quasi wissen, wie oft man würfeln muss, bis man eine Sechs (einen guten Center-String) erwischt:

$$\mathbb{E}[C(r)] = \sum_{\ell=1}^{\infty} p(1-p)^{\ell-1} \cdot \ell.$$

Das entspricht einer geometrischen Verteilung zum Parameter p. Hierfür ist aus der Einführungsvorlesung zur Stochastik bekannt, dass $\mathbb{E}[C(r)] = 1/p \leq r$. Daraus folgt die Behauptung. ■

6.5 Konsensus eines mehrfachen Alignments

Nachdem wir nun verschiedene Möglichkeiten kennen gelernt haben, wie wir mehrfache Sequenzen-Alignments konstruieren können, wollen wir uns jetzt damit beschäftigen, wie wir daraus eine Referenz-Sequenz (einen so genannten Konsensus-String) ableiten können.

6.5.1 Konsensus-Fehler und Steiner-Strings

Zuerst definieren wir den Konsensus-Fehler und einen optimalen Steiner-String einer Menge von Sequenzen.

Definition 6.47 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und $s' \in \Sigma^*$ eine beliebige Zeichenreihe. Der* Konsensus-Fehler *von s' zu S ist definiert durch*

$$E_S(s') := \sum_{j=1}^{k} d(s', s_j).$$

Ein optimaler Steiner-String s^* *für S ist eine Zeichenreihe $s' \in \Sigma^*$ mit minimalem Konsensus-Fehler, d.h.*

$$E_S(s^*) := \min \{E_S(s') \ : \ s' \in \Sigma^*\}.$$

Im Allgemeinen ist s^* nicht eindeutig und es gilt $s^* \notin S$. Dennoch kann s^* in einigen wenigen Fällen durchaus eindeutig sein bzw. $s^* \in S$ gelten.

Woher kommt der Name Steiner-String? In der Graphentheorie sind so genannte Steiner-Bäume bekannt. Dort versucht man in einem gegebenen Graphen einen minimalen Spannbaum für eine ebenfalls vorgegebene Menge von Punkten zu finden, wobei auch Punkte des Graphen, die außerhalb der vorgegeben Punktemenge liegen, in dem Spannbaum enthalten sein dürfen (aber nicht müssen). Ein solcher minimaler Spannbaum, der Punkte außerhalb der gegebenen Punktemenge verwendet, wird Steiner-Baum genannt. Hier ist das ja ähnlich. Der gesuchte Stern ist auch ein minimaler Spannbaum, wobei wir nur das Zentrum als Punkt außerhalb der Punktemenge (der Menge der vorgegebene Sequenzen) zusätzlich verwenden dürfen.

Eine vollständige Enumeration zum Auffinden eines optimalen Steiner-Strings scheidet aus, da der Suchraum unendlich groß ist. Selbst nach einer gewissen Einschränkung aufgrund des zu erwartenden Konsensusfehlers gibt es immer noch exponentiell viele Kandidaten. Man kann sogar zeigen, dass das Problem $\mathcal{NP}$-hart und für

gewisse nichtmetrische Kostenfunktionen sogar $\mathcal{APX}$-hart ist. Der folgende Satz sagt uns, dass sich bereits in der Menge S eine gute Approximation für einen optimalen Steiner-String befindet.

Lemma 6.48 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei d eine metrisches Distanzmaß. Dann existiert ein $s' \in S$ mit*

$$\frac{E_S(s')}{E_S(s^*)} \leq 2 - \frac{2}{k},$$

wobei s^* ein optimaler Steiner-String für S ist.

Beweis: Wir schätzen zuerst den Konsensus-Fehler für ein Zentrum aus der Menge S ab. Wir wählen dazu $s_i \in S$ beliebig, aber fest.

$$\begin{aligned}
E_S(s_i) &= \sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{da } d(s_i, s_i) = 0 \\
&= \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s_i, s_j) \\
&\quad \text{aufgrund der Dreiecksungleichung} \\
&\leq \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s_i, s^*) + \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s^*, s_j) \\
&= \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s_i, s^*) + \sum_{j=1}^{k} d(s^*, s_j) - d(s^*, s_i) \\
&= (k-1) d(s_i, s^*) + E_S(s^*) - d(s^*, s_i) \\
&= (k-2) d(s_i, s^*) + E_S(s^*).
\end{aligned}$$

Sei s_i nun so gewählt, dass $d(s_i, s^*) \leq d(s_j, s^*)$ für alle $j \in [1:k]$. Dann gilt:

$$\begin{aligned}
E_S(s^*) &= \sum_{j=1}^{k} d(s^*, s_j) \\
&\geq \sum_{j=1}^{k} d(s^*, s_i) \\
&= k \cdot d(s^*, s_i).
\end{aligned}$$

Fassen wir beide Zwischenergebnisse zusammen, dann gilt:

$$\begin{aligned}\frac{E_S(s_i)}{E_S(s^*)} &\leq \frac{(k-2)d(s_i,s^*)+E_S(s^*)}{E_S(s^*)}\\ &= 1+\frac{(k-2)d(s_i,s^*)}{E_S(s^*)}\\ &\leq 1+\frac{(k-2)d(s_i,s^*)}{k\cdot d(s^*,s_i)}\\ &= 1+\frac{k-2}{k}\\ &= 2-\frac{2}{k}.\end{aligned}$$

Damit ist das Lemma bewiesen. ■

Wir wissen nun, dass eine der Zeichenreihen aus S eine gute Approximation für den optimalen Steiner-String ist, aber leider nicht welche. Eine genauere Inspektion des Beweises zeigt, dass es derjenige String ist, der den kleinsten Abstand zu einem optimalen Steiner-String besitzt. Diesen wollen jedoch erst approximieren, ohne ihn explizit kennen zu müssen. Aber auch das ist möglich, wie die folgende Argumentation zeigt.

Für den Center-String s_c gilt, dass der Wert $\sum_{j=1}^{k} d(s_i,s_j)$ für $i=c$ minimal wird, d.h. es gilt für alle $i\in[1:k]$:

$$E_S(s_i)=\sum_{j=1}^{k} d(s_i,s_j)\geq\sum_{j=1}^{k} d(s_c,s_j)=E_S(s_c).$$

Damit gilt insbesondere

$$E_S(s_c)\leq E_S(s_i),$$

wobei s_i auch die Zeichenreihe aus Lemma 6.48 sein kann. Daraus folgt sofort

$$\frac{E_S(s_c)}{E_S(s^*)}\leq\frac{E_S(s_i)}{E_S(s^*)}\leq 2-\frac{2}{k}$$

und somit das folgende Korollar:

Korollar 6.49 *Sei $S=\{s_1,\ldots,s_k\}\subseteq\Sigma^*$ und sei s_c ein Center-String von S sowie s^* ein optimaler Steiner-String für S, dann gilt*

$$\frac{E_S(s_c)}{E_S(s^*)}\leq 2-\frac{2}{k}.$$

6.5.2 Randomisierte Verfahren

Da die Kosten zur Bestimmung des Center-Strings ja $O(k^2n^2)$ ist und sich dies im Falle von Center-Star-Approximationen senken lässt, wollen wir auch hier zeigen, dass mit randomisierten Verfahren eine geringere Laufzeit möglich ist.

Lemma 6.50 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und $1 < r \in \mathbb{R}$. Es existieren mehr als $\lfloor \frac{k}{r} \rfloor$ Sequenzen $s_i \in S$ mit $E_S(s_i) \leq \frac{2r-1}{r-1} \cdot E_S(s^*)$, wobei s^* ein optimaler Steiner-String ist.*

Beweis: Wir berechnen zuerst den Mittelwert von $E_S(s_i)$:

$$\begin{aligned}
\frac{1}{k}\sum_{i=1}^{k} E_S(s_i) &= \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{mit } d(s_i, s_i) = 0 \\
&= \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} d(s_i, s_j) \\
&\quad \text{mit Hilfe der Dreiecksungleichung} \\
&\leq \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1 \\ j\neq i}}^{k} [d(s_i, s^*) + d(s^*, s_j)] \\
&= \frac{1}{k}\sum_{i=1}^{k} d(s_i, s^*) \sum_{\substack{j=1 \\ j\neq i}}^{k} 1 + \frac{1}{k}\sum_{j=1}^{k} d(s^*, s_j) \sum_{\substack{i=1 \\ i\neq j}}^{k} 1 \\
&= \frac{2}{k}(k-1)E_S(s^*) \\
&< 2E_S(s^*).
\end{aligned}$$

Wir führen den Beweis des Lemmas mit Hilfe eines Widerspruchs. Dazu nehmen wir an, dass maximal $\lfloor \frac{k}{r} \rfloor$ Sequenzen mit $E_S(s_i) \leq \frac{2r-1}{r-1} \cdot E_S(s^*)$ existieren. Dann gilt:

$$\begin{aligned}
2E_S(s^*) &> \frac{1}{k}\sum_{i=1}^{k} E_S(S_i) \\
&\quad \text{mit Hilfe der Widerspruchsannahme} \\
&\geq \frac{1}{k}\left(\left\lfloor \frac{k}{r} \right\rfloor E_S(s^*) + \left(k - \left\lfloor \frac{k}{r} \right\rfloor\right)\frac{2r-1}{r-1}E_S(s^*)\right) \\
&\quad \text{da offensichtlich } E_S(s^*) \leq \tfrac{2r-1}{r-1}E_S(s^*) \\
&\geq \frac{1}{k}\left(\frac{k}{r}E_S(s^*) + \left(k - \frac{k}{r}\right)\frac{2r-1}{r-1}E_S(s^*)\right)
\end{aligned}$$

$$\begin{aligned} &= E_S(s^*)\left(\frac{1}{r} + \frac{r-1}{r} \cdot \frac{2r-1}{r-1}\right) \\ &= E_S(s^*)\frac{1+2r-1}{r} \\ &= 2E_S(s^*). \end{aligned}$$

Also gilt $2E_S(s^*) < 2E_S(s^*)$, was offensichtlich ein Widerspruch ist. ■

Theorem 6.51 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei $1 < r \in \mathbb{R}$. Wählt man ℓ Sequenzen zufällig aus, dann ist die Sequenz mit dem kleinsten Konsensus-Fehler eine $(2 + \frac{1}{r-1})$-Approximation für einen optimalen Steiner-String mit der Wahrscheinlichkeit größer gleich $1 - \left(\frac{r-1}{r}\right)^\ell$.*

Beweis: Seien $(c_1, \ldots, c_\ell) \in [1:k]^\ell$ zufällig gewählte Zahlen. Sei X_i für $i \in [1:\ell]$ eine binäre Zufallsvariable (auch eine so genannte Indikatorvariable), die genau dann 1 ist, wenn die Wahl von s_{c_i} als Steiner-String eine Approximationsgüte von $(2 + \frac{1}{r-1})$ liefert. Es gilt mit Lemma 6.50

$$\mathrm{Ws}[X_i = 0] = \frac{k - (\lfloor \frac{k}{r} \rfloor + 1)}{k} \leq \frac{k - \frac{k}{r}}{k} = \frac{r-1}{r}.$$

Betrachten wir nun die Zufallsvariable $X = \sum_{i=1}^{\ell} X_i$. Wir berechnen nun die Wahrscheinlichkeit, dass kein Steiner-String eine hinreichend gute Approximationsgüte liefert, d.h. $X = 0$. Da die X_i stochastisch unabhängig sind, gilt:

$$\mathrm{Ws}[X = 0] = \mathrm{Ws}[X_1 = 0 \wedge \cdots \wedge X_\ell = 0] = \prod_{i=1}^{\ell} \mathrm{Ws}[X_i = 0] \leq \left(\frac{r-1}{r}\right)^\ell.$$

Damit erhalten wir sofort

$$Ws[X \geq 1] = 1 - \mathrm{Ws}[X = 0] \geq 1 - \left(\frac{r-1}{r}\right)^\ell$$

und damit ist der Satz bewiesen. ■

Korollar 6.52 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Für $1 < r \in \mathbb{R}$ sei $C(r)$ die Anzahl von Sequenzen, die zufällig gewählt werden müssen, bis der beste Konsensus-Fehler der gewählten Sequenz eine $(2 + \frac{1}{r-1})$-Approximation ist. Dann gilt $\mathbb{E}[C(r)] \leq r$.*

Beweis: Wir haben im letzten Lemma nachgewiesen, dass sich die Wahl eines guten Steiner-Strings durch eine Binomialverteilung zum Parameter $p \geq 1 - (1 - \frac{1}{r})$ abschätzen lässt. Nun wollen wir quasi wissen, wie oft man würfeln muss, bis man eine Sechs (einen guten Center-String) erwischt:

$$\mathbb{E}[C(r)] = \sum_{\ell=1}^{\infty} p(1-p)^{\ell-1} \cdot \ell.$$

Das entspricht einer geometrischen Verteilung zum Parameter p. Hierfür ist aus der Einführungsvorlesung zur Stochastik bekannt, dass $\mathbb{E}[C(r)] = 1/p \leq r$. Daraus folgt die Behauptung. ■

Damit stellen Steiner-Strings also eine Möglichkeit dar, für eine Folge von Sequenzen eine Referenz-Sequenz zu generieren.

6.5.3 Alignment-Fehler und Konsensus-String

Jetzt stellen wir eine weitere Methode vor, die auf mehrfachen Sequenzen-Alignments basiert.

Definition 6.53 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Alignment für S der Länge $n := |\overline{s}_1|$. Ein* Konsensus-Zeichen *an der Position $i \in [1:n]$ ist ein Zeichen $\mathcal{S}_M(i) := x \in \overline{\Sigma}$ mit*

$$\sum_{j=1}^{k} w(x, \overline{s}_{j,i}) = \min \left\{ \sum_{j=1}^{k} w(a, \overline{s}_{j,i}) \ : \ a \in \overline{\Sigma} \right\} =: \ \delta_M(i).$$

Ein Konsensus-String $\mathcal{S}_M \in \overline{\Sigma}^n$ *des mehrfachen Alignments M für S ist dann definiert als $\mathcal{S}_M := \mathcal{S}_M(1) \cdots \mathcal{S}_M(n)$.*

Beachte, dass weder ein Konsensus-Zeichen noch ein Konsensus-String eindeutig sein muss und dass beide Leerzeichen enthalten können. Zur Erinnerung sei angemerkt, dass $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine Kostenfunktion ist, bei der für mehrfache Sequenzen-Alignments $w(-,-) = 0$ gilt.

Wir können nun ganz allgemein einen so genannten Alignment-Fehler für eine mehrfaches Sequenzen-Alignment definieren.

Definition 6.54 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Alignment für S der Länge $n = |\overline{s}_1|$, dann ist der* Alignment-Fehler *einer Sequenz $s' \in \overline{\Sigma}^n$ definiert als*

$$E_M(s') = \sum_{j=1}^{k} \sum_{i=1}^{n} w(s'_i, \overline{s}_{j,i}) = \sum_{j=1}^{k} w(s', \overline{s}_j),$$

wobei w wieder die zugrunde liegende Kostenfunktion ist.

Speziell gilt dann

$$E_M(\mathcal{S}_M) = \sum_{i=1}^{n} \delta_M(i).$$

Weiterhin haben wir das folgende Lemma.

Lemma 6.55 *Sei $S \subseteq \Sigma^*$ und sei M ein mehrfaches Sequenzen-Alignment für S der Länge n, dann gilt für alle $s' \in \Sigma^n$:*

$$E_M(\mathcal{S}_M) \leq E_M(s').$$

Der Beweis folgt unmittelbar aus der Tatsache, dass wir jeweils über n Spalten summieren und in jeder Spalte ein Konsensus-Zeichen gewählt wird, das für die entsprechende Spalte den Wert minimiert.

Basierend auf dem Maß des Alignment-Fehlers können wir nun optimale Konsensus-Alignments definieren.

Definition 6.56 *Ein optimales* Konsensus-Alignment *für $S \subseteq \Sigma^*$ ist ein mehrfaches Sequenzen-Alignment M für S, dessen Konsensus-String einen minimalen Alignment-Fehler besitzt.*

Wir kommen später darauf zurück, wie man ein solches Konsensuns-Alignment gut approximieren kann.

6.5.4 Beziehung zwischen Steiner-String und Konsensus-String

Wir haben jetzt also mehrere, unterschiedliche Definitionen für einen so genannten *Konsensus-String*, der die beste Annäherung an ein mehrfaches Sequenzen-Alignment liefert, kennen gelernt. Bevor wir auf die Beziehungen zwischen diesen eingehen, zählen wir diese noch einmal auf:

- optimaler Steiner-String (ohne ein mehrfaches Sequenzen-Alignment!);
- Konsensus-String für ein optimales mehrfaches Sequenzen-Alignment M (die Optimalität wird hierbei bezüglich minimalem Alignment-Fehler definiert, also ein Konsensus-Alignment);
- Konsensus-String für ein optimales mehrfaches Sequenzen-Alignment M (die Optimalität wird hierbei bezüglich des Sum-of-Pairs-Maßes definiert).

Zuerst beweisen wir das folgende fundamentale Lemma.

Lemma 6.57 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und dessen optimaler Steiner-String s^*. Für ein beliebiges mehrfaches Alignment M für S und dessen Konsensus-String $\mathcal{S}_M$ gilt:*

$$E_M(\mathcal{S}_M) \geq E_S(\mathcal{S}_M|_\Sigma) \geq E_S(s^*).$$

Beweis: Sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ ein *beliebiges* mehrfaches Sequenzen-Alignment für S. Sei $\mathcal{S}_M$ ein Konsensus-String für M. Für das projizierte paarweise Alignment von $\mathcal{S}_M$ mit $\overline{s}_j$ gilt:

$$w(\mathcal{S}_M, \overline{s}_j) \geq d(\mathcal{S}_M|_\Sigma, s_j). \tag{6.1}$$

Also gilt nach Definition des Alignment- und Konsensus-Fehlers:

$$\begin{aligned}
E_M(\mathcal{S}_M) &= \sum_{j=1}^{k} \sum_{i=1}^{n} w(\mathcal{S}_M(i), \overline{s}_{j,i}) \\
&= \sum_{j=1}^{k} w(\mathcal{S}_M, \overline{s}_j) \\
&\quad \text{mit Ungleichung (6.1)} \\
&\geq \sum_{j=1}^{k} d(\mathcal{S}_M|_\Sigma, s_j) \\
&= E_S(\mathcal{S}_M|_\Sigma) \\
&\geq E_S(s^*).
\end{aligned}$$

Die letzte Ungleichung folgt, da ein optimaler Steiner-String immer den minimalen Konsensus-Fehler für S besitzt. ∎

Prinzipiell gilt also, dass der Alignment-Fehler eines Konsensus-Strings für ein Alignment M mindestens so groß ist wie der minimale Konsensus-Fehler der zugrunde liegenden Sequenz-Menge S.

Theorem 6.58 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$.*

i) Sei s' ein Konsensus-String eines optimalen Konsensus-Alignments für S, dann ist $s'|_\Sigma$ ein optimaler Steiner-String für S.

ii) Sei s^* ein optimaler Steiner-String für S und sei M ein mehrfaches Sequenzen-Alignment für $S \cup \{s^*\}$, welches konsistent zu einem Stern mit Zentrum s^* ist. Dann ist M ohne die Zeile für s^* ein optimales Konsensus-Alignment für S.

Beachte, dass in ii) Spalten, die nur aus Leerzeichen bestehen, ebenfalls gestrichen werden (was nicht wirklich geschehen kann, siehe Details im Beweis). Teil i) ist eine Reduktion des optimalen Steiner-String-Problems auf optimale Konsensus-Alignments. Somit ist auch die Bestimmung eines optimalen Konsensus-Alignment ein $\mathcal{NP}$-hartes Problem.

Beweis: Sei $S = \{s_1, \ldots, s_k\}$ und sei $M' = (\overline{s}_1, \ldots, \overline{s}_k)$ ein *beliebiges* mehrfaches Sequenzen-Alignment für S. Sei $\mathcal{S}_{M'}$ ein Konsensus-String für M'. Also gilt nach Lemma 6.57 für einen optimalen Steiner-String s^* für S:

$$E_{M'}(\mathcal{S}_{M'}) \geq E_S(\mathcal{S}_{M'}|_\Sigma) \geq E_S(s^*). \tag{6.2}$$

Sei weiterhin s^* ein optimaler Steiner-String für S und sei jetzt $M^* = (\overline{s}^*, \overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Sequenzen-Alignment für $S \cup \{s^*\}$, das konsistent zu einem Stern mit Zentrum s^* ist. Für das projizierte Alignment $\overline{s}^*$ mit $\overline{s}_j$ gilt

$$w(\overline{s}^*, \overline{s}_j) = d(s^*, s_j), \tag{6.3}$$

da M^* konsistent zu einem Stern mit s^* als Zentrum ist.

Sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ das mehrfache Sequenzen-Alignment, das aus M^* durch Streichen von $\overline{s}^*$ entsteht. Dabei könnten gegebenfalls auch Spalten in M gestrichen, die nur aus Leerzeichen bestehen. Dies kann aber nicht passieren, da dann das Streichen des korrespondierenden Buchstaben im optimalen Steiner-String einen Steiner String mit kleinerem Konsensus-Fehler als dem optimalen liefern würde. Letzteres folgt aus der Tatsache, dass das ursprüngliche Alignment zu einem Stern mit Zentrum s^* konsistent ist und somit die projizierten paarweisen Alignments von s^* und s_i optimal sind.

Mit $n = |\overline{s}^*|$ gilt nach Definition des Alignment- und Konsensus-Fehlers:

$$\begin{aligned} E_M(\overline{s}^*) &= \sum_{j=1}^{k} \sum_{i=1}^{n} w(\overline{s}_i^*, \overline{s}_{j,i}) \\ &= \sum_{j=1}^{k} w(\overline{s}^*, \overline{s}_j) \\ &\quad \text{mit Hilfe der Gleichung 6.3} \\ &= \sum_{j=1}^{k} d(s^*, s_j) \\ &= E_S(s^*). \end{aligned}$$

Es gibt also ein mehrfaches Sequenzen-Alignment M für S, wobei der Alignment-Fehler der zum optimalen Steiner-Strings gehörigen Zeile gleich dem Konsensus-Fehler von s^* ist.

Wir zeigen nun, dass $E_M(\mathcal{S}_M) = E_M(\overline{s}^*)$ gilt. Wäre nun $E_M(\mathcal{S}_M) > E_M(\overline{s}^*)$, dann wäre jedoch $\overline{s}^*$ ein Konsensus-String von M mit einem kleineren Alignment-Fehler, was nach Definition nicht sein kann. Wäre andererseits $E_M(\mathcal{S}_M) < E_M(\overline{s}^*)$, dann wäre auch $E_M(\mathcal{S}_M) < E_S(s^*)$, da ja $E_M(\overline{s}^*) = E_S(s^*)$. Da nach Lemma 6.57 aber $E_M(\mathcal{S}_M) \geq E_S(\mathcal{S}_M|_\Sigma) \geq E_S(s^*)$ gilt, erhalten wir einen Widerspruch. Somit gilt also $E_M(\mathcal{S}_M) = E_M(\overline{s}^*) = E_S(s^*)$.

Damit folgt die zweite Behauptung des Satzes, da M dann ein optimales Konsensus-Alignment ist, da $E_M(\overline{s}^*)$ nach Lemma 6.57 bzw. Ungleichung (6.2) bereits den kleinstmöglichen Wert $E_S(s^*)$ annimmt.

05.12.19

Die erste Behauptung folgt, da es ein Konsensus-Alignment M mit Alignment-Fehler $E_M(\mathcal{S}_M) = E_S(s^*)$ gibt und da dann für ein optimales Konsensus-Alignment A die Beziehung $E_A(\mathcal{S}_A) \leq E_S(s^*)$ gelten muss. Also gilt:

$$E_S(s^*) \geq E_A(\mathcal{S}_A) \geq E_S(\mathcal{S}_A|_\Sigma) \geq E_S(s^*).$$

Die mittlere Ungleichung folgt wieder aus Lemma 6.57 und die letzte, da s^* ein optimaler Steiner-String ist. Also gilt $E_S(\mathcal{S}_A|_\Sigma) = E_S(s^*)$ und $\mathcal{S}_A|_\Sigma$ ist auch ein optimaler Steiner-String. ■

Wir wollen noch anmerken, das der Beweis von Teil ii) zumindest teilweise konstruktiv ist, wir können hier einen approximativen Algorithmus für die Konstruktion des Center-Star-Alignments anwenden. Der Beweis von Teil i) ist hingegen ein reiner Existenzbeweis, der erst einmal keinen direkten Algorithmus für den optimalen Steiner-String impliziert.

Weiterhin haben wir bereits gezeigt, dass der Center-String eine gute Approximation des Steiner-String einer Menge von Sequenzen S ist:

$$\frac{E_S(s_c)}{E_S(s^*)} \leq 2 - \frac{2}{k},$$

wobei s_c ein Center-String von S ist und s^* ein optimaler Steiner-String für S ist. Wir werden dies auch auf optimale Konsensus-Alignments übertragen können. Fassen wir das Ergebnis dieses Abschnitts noch einmal im folgenden Satz zusammen.

Theorem 6.59 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei M das mehrfache Alignment für S, das mit Hilfe der Center-Star-Methode konstruiert wurde, wobei als Zentrum der Center-String $s_c \in S$ verwendet wird.*

i) Das Alignment M besitzt eine Sum-of-Pairs-Distanz, die maximal um den Faktor $(2 - \frac{2}{k})$ vom Optimum entfernt ist;

ii) Der Center-String von S ist eine $(2 - \frac{2}{k})$-Approximation für einen optimalen Steiner-String für S;

iii) Das Alignment M besitzt einen Alignment-Fehler, der maximal um den Faktor $(2 - \frac{2}{k})$ vom Optimum entfernt ist.

Beweis: Die Aussage i) wurde bereits in Theorem 6.38 bewiesen. Die Aussage ii) wurde in Korollar 6.49 gezeigt.

Es bleibt also nur noch die Aussage iii) zu beweisen. Dazu zeigen wir, dass das von der Center-Star-Approximation mit dem Center-String $s_c \in S$ als Zentrum gelieferte mehrfaches Sequenzen-Alignment M eine Approximationsgüte von ebenfalls $(2 - \frac{2}{k})$ bzgl. des Alignment-Fehlers besitzt. Sei dazu s^* ein optimaler Steiner-String für S. Es gilt dann (da nach Definition zunächst $\mathcal{S}_M$ bzgl. M einen minimalen Alignment-Fehler besitzt):

$$\begin{aligned} E_M(\mathcal{S}_M) &\leq E_M(\overline{s}_c) \\ &\quad \text{nach Definition des Alignment-Fehlers} \\ &= \sum_{j=1}^{k} w(\overline{s}_c, \overline{s}_j) \\ &\quad \text{da } M \text{ zu einem Stern konsistent ist} \\ &= \sum_{j=1}^{k} d(s_c, s_j) \end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=1}^{k} d(s_c, s_j) \\
&\quad \text{nach Definition des Konsensus-Fehlers} \\
&= E_S(s_c) \\
&\quad \text{Mit Korollar 6.49} \\
&\leq \left(2 - \frac{2}{k}\right) E_S(s^*).
\end{aligned}
$$

Weiter gilt für ein optimales Alignment M^* bzgl. des Alignment-Fehlers mit zugehörigem Konsensus-String $\mathcal{S}_{M^*}$ (mit Lemma 6.57 und der Eigenschaft eines optimalen Steiner-Strings):

$$E_{M^*}(\mathcal{S}_{M^*}) \;\geq\; E_S(\mathcal{S}_{M^*}|_\Sigma) \;\geq\; E_S(s^*).$$

Daher gilt für die Approximationsgüte:

$$\frac{E_M(\mathcal{S}_M)}{E_{M^*}(\mathcal{S}_{M^*})} \;\leq\; \frac{\left(2 - \frac{2}{k}\right) E_S(s^*)}{E_S(s^*)} \;=\; 2 - \frac{2}{k}.$$

Damit ist auch diese Behauptung bewiesen. ■

6.6 Phylogenetische Alignments

Im letzten Abschnitt haben wir gesehen, wie wir mit Hilfe mehrfacher Sequenzen-Alignments, die zu Sternen konsistent sind, eine Approximation für ein optimales mehrfaches Sequenzen-Alignment oder einen Konsensus-String konstruieren können. Manchmal ist für die gegebenen Sequenzen ja mehr bekannt, zum Beispiel ein phylogenetischer Baum der zugehörigen Spezies. Diesen könnte man für die Konstruktion zu Bäumen konsistenter mehrfacher Sequenzen-Alignments ja ausnutzen.

6.6.1 Definition phylogenetischer Alignments

Wir werden jetzt Alignments konstruieren, die wieder zu Bäumen konsistent sind. Allerdings sind jetzt nur die Sequenzen an den Blättern bekannt und die inneren Knoten sind ohne Sequenzen. Dies folgt daher, da für einen phylogenetischen Baum in der Regel nur die Sequenzen der momentan noch nicht ausgestorbenen Spezies bekannt sind, und das sind genau diejenigen, die an den Blättern stehen. An den

inneren Knoten stehen ja die Sequenzen, von den Vorahnen der bekannten Spezies, die in aller Regel heutzutage ausgestorben sind. Wir geben zuerst die formale Definition so genannter phylogenetischer mehrfacher Sequenzen-Alignments an.

Definition 6.60 *Sei $S \subseteq \Sigma^*$ und $T = (V, E)$ ein gewurzelter Baum mit $V = I \cup B$, wobei I bzw. B die Menge der inneren Knoten bzw. der Blätter des Baumes bezeichnet. Weiterhin sei $\varphi : B \to S$ eine Bijektion mit*

$$\varphi : B \to S : v \mapsto s_v.$$

Ein solcher markierter Baum (T, φ) heißt konsistent *zu S.*

Ein phylogenetisches mehrfaches Sequenzen-Alignment *(M, φ, T) (kurz:* PMSA*) ist eine Zuordnung von Zeichenreihen aus Σ^* an I, d.h.*

$$\varphi : I \to \Sigma^* : v \mapsto s_v$$

und ein mehrfaches Sequenzen-Alignment M, das mit T konsistent ist.

Beachte, dass in der obigen Definition formal der Definitionsbereich von φ von B auf $B \cup I$ erweitert wird. Manchmal schreiben wir für ein phylogenetisches mehrfaches Alignment (M, φ, T) für (S, T) auch kurz (M, T') , wenn T' aus T durch vollständige Zuweisung von Sequenzen aus S durch φ entstanden ist (unabhängig davon, ob φ nur auf B oder ganz auf V definiert ist). Der Baum aus der Definition ist in Abbildung 6.14 illustriert.

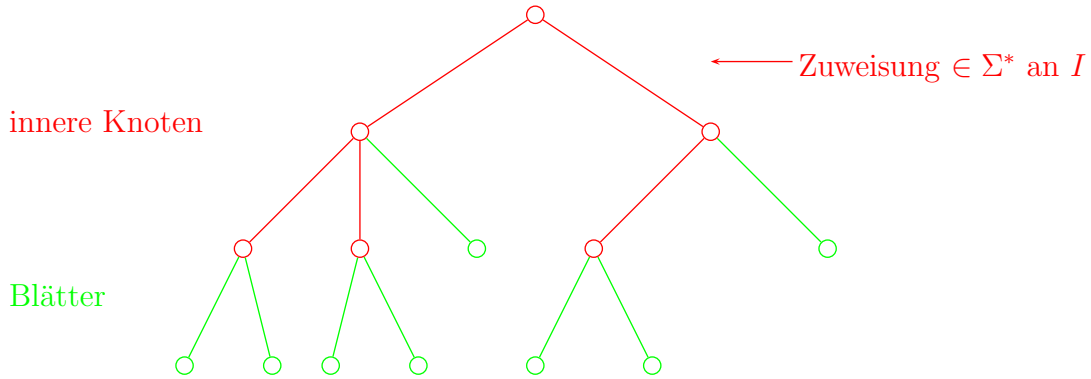


Abbildung 6.14: Skizze: Phylogenetisches mehrfaches Sequenzen-Alignment

Für $(v, w) \in E(T)$ bezeichne $D_M(v, w)$ die Alignment-Distanz des induzierten Alignments aus einem phylogenetischen mehrfachen Alignment M für die zu v und w zugeordneten Sequenzen. Da das mehrfache Sequenzen-Alignment zu T konsistent ist, entspricht diese Alignment-Distanz des induzierten Alignments daher genau der

Alignment-Distanz des optimalen paarweisen Sequenzen-Alignments dieser beiden Sequenzen, d.h. $D_M(v, w) = d(s_v, s_w)$.

Definition 6.61 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Für ein phylogenetisches mehrfaches Alignment (M, φ, T) für S bezeichnet*

$$D_M(T) := \sum_{e \in E(T)} D_M(e) = \sum_{(v,w) \in E(T)} D_M((v, w)) = \sum_{(v,w) \in E(T)} d(s_v, s_w)$$

die Distanz eines phylogenetischen mehrfachen Alignments.

Somit können wir nun optimale phylogenetische mehrfache Sequenzen-Alignments definieren.

Definition 6.62 *Ein optimales phylogenetisches mehrfaches Sequenzen-Alignment ist ein phylogenetisches mehrfaches Sequenzen-Alignment (M, φ, T) für T, das $D_M(T)$ minimiert.*

Leider ist auch hier wieder die Entscheidung, ob es ein phylogenetisches mehrfaches Sequenzen-Alignment mit einer Distanz kleiner gleich D gibt, ein $\mathcal{NP}$-hartes Problem. Wir können also auch hierfür wieder nicht auf ein effizientes Verfahren für eine optimale Lösung hoffen.

Im Folgenden nehmen wir an, dass alle Sequenzen in S paarweise verschieden sind. Des Weiteren nehmen wir an, dass jeder innere Knoten von T mindestens 2 Kinder besitzt. Letzteres führt dazu, dass T maximal $O(k)$ Knoten bzw. Kanten besitzt, wenn S aus k Sequenzen besteht. Weiterhin sei d eine Metrik.

6.6.2 Geliftete Alignments

Um wieder eine Approximation konstruieren zu können, betrachten wir so genannte geliftete mehrfache phylogenetische Sequenzen-Alignments. Ähnlich wie wir bei der Center-Star-Methode für das Zentrum eine Sequenz aus der Menge S wählen, werden wir uns bei der Zuordnung der Sequenzen an die inneren Knoten auch wieder auf die Sequenzen aus S beschränken können. Wir schränken uns sogar noch ein wenig mehr ein, wie die folgenden Definitionen zeigen.

Definition 6.63 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Ein Knoten $v \in V(T)$ heißt* geliftet*, wenn er entweder ein Blatt ist oder ein Knoten $w \in V(T)$ mit $(v, w) \in E(T)$ und $s_v = s_w$ existiert.*

Somit können wir nun geliftete phylogenetische mehrfache Sequenzen-Alignments definieren.

Definition 6.64 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Ein phylogenetisches mehrfaches Sequenzen-Alignment heißt* geliftet*, wenn jeder Knoten $v \in V$ geliftet ist.*

Wir werden uns im Folgenden nur noch mit gelifteten phylogenetischen mehrfachen Alignments beschäftigen.

6.6.3 Konstruktion gelifteter aus optimalen Alignments

Nun zeigen wir, wie wir aus einem optimalen phylogenetischen mehrfachen Alignment ein geliftetes konstruieren. Dies ist an und für sich nicht sinnvoll, da wir mit einem optimalen Alignment natürlich glücklich wären und damit an dem gelifteten kein Interesse mehr hätten. Wir werden aber nachher sehen, dass uns diese Konstruktion beim Beweis der Approximationsgüte behilflich sein wird.

Zu Beginn sind alle Blätter geliftet. Wir betrachten jetzt einen Knoten v, dessen Kinder alle geliftet sind, und werden diesen Knoten selbst liften (siehe Abbildung 6.15).

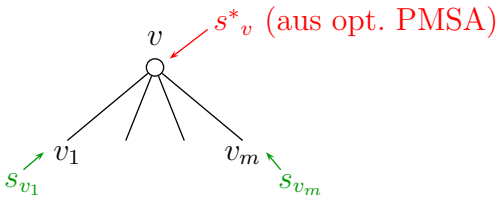


Abbildung 6.15: Skizze: Liften eines Knotens

Wir ersetzen jetzt die Zeichenreihe s^*_v des Knotens v durch die Zeichenreihe s_{v_j} seines Kindes v_j, wobei $d(s_{v_j}, s^*_v) \leq d(s_{v_i}, s^*_v)$ für alle $i \in [1:m]$ gelten soll, d.h.:

$$j := \operatorname{argmin} \{d(s^*_v, s_{v_i}) \ : \ i \in [1:m]\} .$$

Wir ersetzen also die Zeichenreihe s^*_v des Knotens v durch die Zeichenreihe eines seiner Kinder, die zu s^*_v am nächsten ist (im Sinne der Alignment-Distanz).

6.6.4 Güte gelifteter Alignments

Nun wollen wir die Güte dieses phylogenetischen mehrfachen Alignments bestimmen, das durch Liften aus einem optimalen phylogenetischen mehrfachen Alignment entstanden ist.

Theorem 6.65 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Ist (M^*, φ^*, T) ein optimales phylogenetisches mehrfaches Alignment für S, dann gilt für das daraus konstruierte geliftete phylogenetische mehrfache Alignment (M_L, φ^L, T):*

$$D_{M_L}(T) \leq 2 \cdot D_{M^*}(T).$$

Beweis: Wir betrachten zuerst eine beliebige Kante $(v, w) \in E(T)$ (siehe auch Abbildung 6.16). Beachte dabei, dass der Baum T immer gleich ist und nur die Zuordnung der Sequenzen an den inneren Knoten für unterschiedliche phylogenetische mehrfache Alignments verschieden sein kann.

Gilt $s_v = s_w$, dann ist logischerweise $D(v, w) = d(s_v, s_w) = 0$. Ist andernfalls $s_v \neq s_w$, dann erhalten wir

$$D_{M_L}(v, w) = d(s_v, s_w) \leq d(s_v, s_v^*) + d(s_v^*, s_w).$$

Aufgrund des vorgenommenen Liftings in (T, φ^L) gilt außerdem $d(s_v^*, s_v) \leq d(s_v^*, s_w)$. Somit erhalten wir insgesamt:

$$D_{M_L}(v, w) = d(s_v, s_w) \leq d(s_v, s_v^*) + d(s_v^*, s_w) \leq 2 \cdot d(s_v^*, s_w).$$

Betrachten wir eine Kante (v, w) in T, wobei w ein Kind von v ist. Diese Kante definiert in T mittels φ^L einen Pfad $p_{v,w}$ von v über w zu einem Blatt x, indem wir vom Knoten v aus immer zu dem Kind gehen, das ebenfalls mit der Sequenz s_w markiert ist. Letztendlich landen wir dann im Blatt x. Dieser Pfad ist eindeutig, da

10.12.19

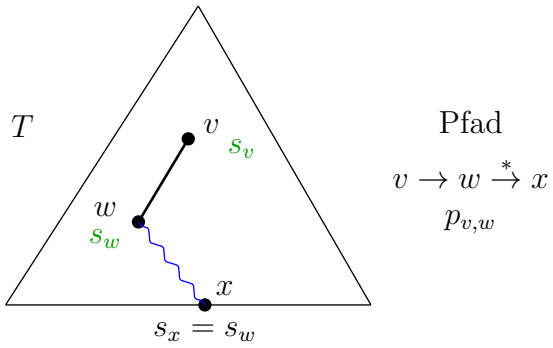


Abbildung 6.16: Skizze: Kante (v, w) definiert mithilfe von T einen Pfad $p_{v,w}$ in T

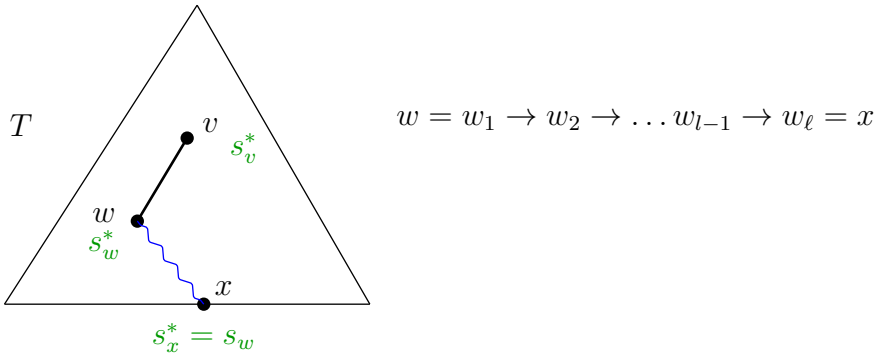


Abbildung 6.17: Skizze: Pfad $p_{v,w}$ im optimalen Baum (T, φ^*)

wir angenommen haben, dass die Sequenzen aus S paarweise verschieden sind (siehe auch Abbildung 6.16).

Betrachten wir jetzt diesen Pfad $p_{v,w}$ mit $v \to w = w_1 \to \cdots \to w_\ell = x$ (siehe Abbildung 6.17) in einem optimalen phylogenetischen mehrfachen Sequenzen-Alignment, d.h. mit der zugehörigen Zuordnung φ^*, dann gilt aufgrund mehrfacher Anwendung der Dreiecksungleichung (durch Einfügen der zu den Knoten $w_1, \ldots, w_\ell$ gehörigen Sequenzen $s_{w_1}, \ldots, s_{w_\ell}$ auf dem Pfad $p_{v,w}$):

$$\begin{aligned}
d(s_v^*, s_w) &\leq d(s_v^*, s_{w_1}^*) + d(s_{w_1}^*, s_w) \\
&\leq d(s_v^*, s_{w_1}^*) + d(s_{w_1}^*, s_{w_2}^*) + d(s_{w_2}^*, s_w) \\
&\vdots \\
&\leq d(s_v^*, s_{w_1}^*) + d(s_{w_1}^*, s_{w_2}^*) + \cdots + d(s_{w_{\ell-1}}^*, s_w) \\
&\leq d(s_v^*, s_{w_1}^*) + d(s_{w_1}^*, s_{w_2}^*) + \cdots + d(s_{w_{\ell-1}}^*, s_{w_\ell}^*) + d(s_{w_\ell}^*, s_w) \\
&\quad \text{da } s_{w_\ell}^* = s_{w_\ell} = s_w \text{ ist, folgt } d(s_{w_\ell}^*, s_w) = 0 \\
&= d(s_v^*, s_{w_1}^*) + d(s_{w_1}^*, s_{w_2}^*) + \cdots + d(s_{w_{\ell-1}}^*, s_{w_\ell}^*) \\
&=: D^*(p_{v,w}).
\end{aligned}$$

Insgesamt erhalten wir dann

$$D_{M_L}(v, w) \leq 2 \cdot d(s_v^*, s_w) \leq 2 \cdot D^*(p_{v,w}). \tag{6.4}$$

Wir können also die Distanz des gelifteten phylogenetischen Alignments geschickt wie folgt berechnen (siehe auch Abbildung 6.18). Für alle grünen Kanten e gilt $D_{M_L}(e) = 0$. Wir müssen also nur die roten Kanten in (T, φ^L) aufaddieren. Jede rote Kante (v, w) in T korrespondiert zu einem Pfad $p_{v,w}$ in T. Es gilt weiter, dass für alle Kanten (v, w), für die $D_{M_L}(v, w) > 0$ in (T, φ^L) ist, die zugehörigen Pfade disjunkt sind. Somit kann die Summe der roten Kantengewichte in (T, φ^L) durch die

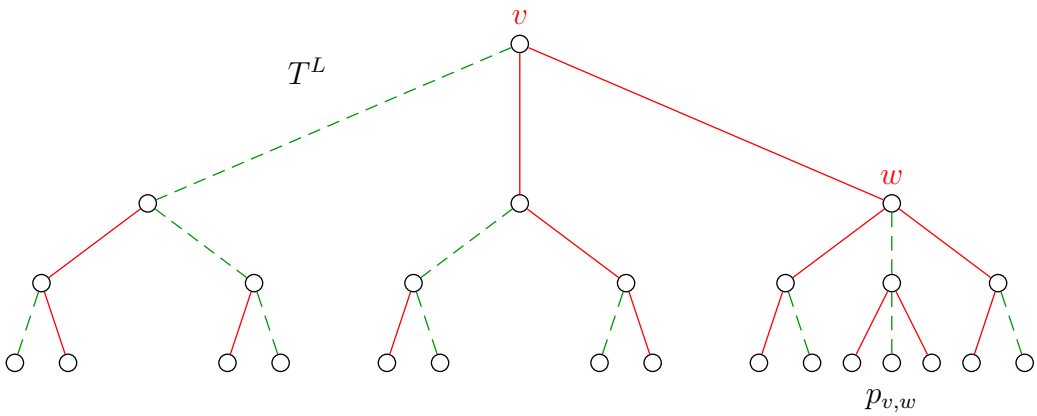


Abbildung 6.18: Skizze: Beziehung Kantengewichte in T^L zu T^*

doppelte Summe aller Kantengewichte im Baum (T, φ^*) abgeschätzt werden. Damit ergibt sich sich nun Folgendes:

$$\begin{aligned}
D_{M_L}(T) &= \sum_{(v,w)\in E(T)} D_{M_L}(v,w) \\
&= \sum_{\substack{(v,w)\in E(T) \\ D_{M_L}(v,w)>0}} D_{M_L}(v,w) \\
&\quad \text{da } D_{M_L}(v,w) \leq 2 \cdot D^*(p_{v,w}), \text{ siehe Gleichung 6.4} \\
&\leq 2 \sum_{\substack{(v,w)\in E(T) \\ D_{M_L}(v,w)>0}} D^*(p_{v,w}) \\
&\leq 2 \sum_{(v,w)\in E(T)} D^*((v,w)) \\
&= 2 \cdot D(T^*).
\end{aligned}$$

Damit ist der Satz bewiesen. ■

Damit haben wir gezeigt, dass es ein geliftetes phylogenetisches Alignment gibt, dass höchstens um den Faktor zwei vom Optimum entfernt ist. Man kann sogar noch eine stärkere Version beweisen, was wir hier nicht tun wollen.

Theorem 6.66 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Ist (M^*, φ^*, T) ein optimales phylogenetisches mehrfaches Alignment für S, dann gilt für das daraus konstruierte geliftete phylogenetische mehrfache Alignment (M_L, φ^L, T):*

$$D_{M_L}(T) \leq \left(2 - \frac{2}{k}\right) \cdot D_{M^*}(T).$$

Wenn wir jetzt ein optimales geliftetes phylogenetisches mehrfaches Sequenzen-Alignment konstruieren, so gelten die oben genannten Approximationsgüten natürlich auch für dieses.

6.6.5 Berechnung eines optimalen gelifteten PMSA

Wir wollen jetzt mit Hilfe der dynamischen Programmierung ein optimales geliftetes phylogenetisches mehrfaches Sequenzen-Alignment konstruieren. Dieses muss dann eine Distanz besitzen, die höchstens so groß wie die des gelifteten optimalen phylogenetischen mehrfachen Sequenzen-Alignments ist und muss daher ebenfalls die Approximationsgüte 2 erreichen. Bevor wir die Rekursionsgleichung beschreiben können, benötigen wir noch eine Notation.

Notation 6.67 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Für $v \in V(T)$ bezeichnet $S(v)$ die Menge aller Sequenzen, die an Blättern in T_v vorkommen, wobei T_v der am Knoten v gewurzelte Teilbaum von T ist.*

Für die dynamische Programmierung bezeichnet $D(v, s)$ die Distanz eines besten gelifteten phylogenetischen mehrfachen Alignments für den am Knoten v gewurzelten Teilbaum T_v, so dass v mit der Sequenz $s \in S(v)$ markiert ist. Für die Berechnung von $D(v, s)$ stellen wir zunächst wieder einmal eine Rekursionsgleichung auf. Es gilt dann (siehe auch Abbildung 6.19):

$$D(v,s) = \begin{cases} \infty & \text{wenn } s \notin S(v), \\ 0 & \text{wenn } v \text{ ein Blatt ist,} \\ \displaystyle\sum_{(v,w)\in E(T)} \min\{d(s,s') + D(w,s') \;:\; s' \in S(w)\} & \text{sonst.} \end{cases}$$

Während eines Preprocessings ist es nötig, für alle Paare $(s, s') \in S^2$ die optimale Distanz $d(s, s')$ zu berechnen. Dafür ergibt sich folgender Zeitbedarf:

$$O\left(\sum_{i=1}^{k}\sum_{j=1}^{k} |s_i| \cdot |s_j|\right) = O\left(\left(\sum_{i=1}^{k} |s_i|\right) \cdot \left(\sum_{j=1}^{k} |s_j|\right)\right) = O(k^2 n^2),$$

wenn wir wieder $|s_i| = O(n)$ annehmen. Nach dem Preprocessing kann dann jeder Wert in der Minimumbildung in konstanter Zeit bestimmt werden.

Wir müssen uns nur noch überlegen, wie viele Terme bei allen Minimumsbildungen in allen Summen insgesamt betrachtet werden. Dies geschieht für jede Baumkante

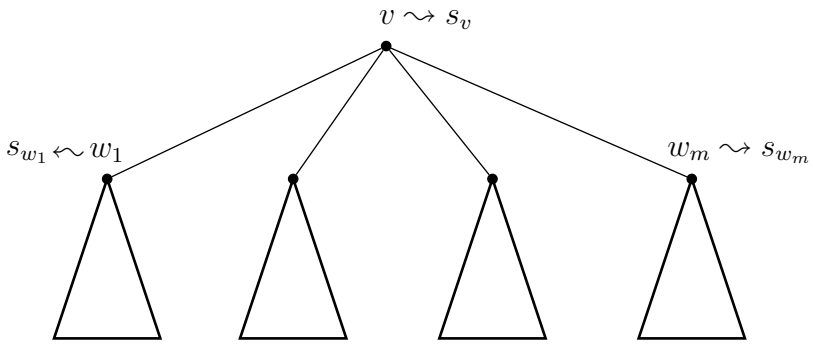
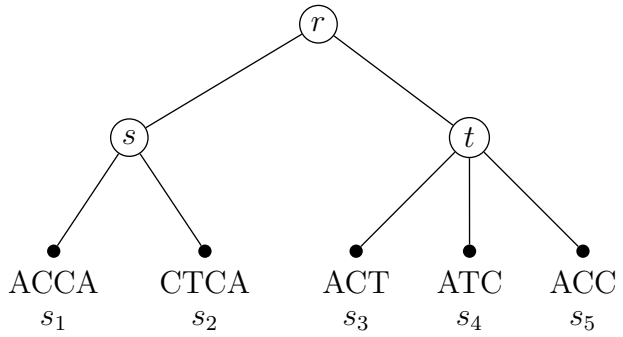


Abbildung 6.19: Skizze: Berechnung von $D(v, s)$

(v, w) und jedes Paar von Sequenzen (s, s') genau einmal. Da ein gewurzelter Baum, bei dem es keine Knoten mit genau einem Kind gibt, weniger innere Knoten als Blätter besitzt, hat der Baum maximal $O(k)$ Knoten. Weiterhin gilt, dass jeder Baum weniger Kanten als Knoten besitzt und es somit maximal $O(k)$ Kanten in T gibt. Offensichtlich gibt es k^2 Paare von Sequenzen aus S. Also gilt insgesamt für die Laufzeit: $O(k^2n^2 + k^3)$. Hierbei haben wird die Berücksichtigung der Bedingung $s \notin S(v)$ bzw. $s' \in S(w)$ erst einmal vernachlässigt und somit die betrachtete Anzahl der betrachteten Bäume nur vergrößert. Für die Berechnung des zum Baum konsistenten Alignments benötigen wir dann noch Zeit $O(nk(n + k)) = O(k^2n^2)$. Fassen wir das Ergebnis noch zusammen.

Theorem 6.68 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ mit $|s_i| = O(n)$ und sei T ein zu S konsistenter Baum. Ein phylogenetisches mehrfaches Alignment für S und T, dessen Distanz maximal um 2 von einem optimalen phylogenetischen mehrfachen Alignment für S und T abweicht, kann in Zeit $O(k^2n^2 + k^3)$ konstruiert werden.*

In Abbildung 6.20 ist ein Beispiel für ein geliftetes phylogenetisches mehrfaches Sequenzen-Alignment angegeben. Die Kostenfunktion für das Distanzmaß sei hier-



d	s_1	s_2	s_3	s_4	s_5
s_1	0	2	2	2	1
s_2		0	3	2	3
s_3			0	2	1
s_4				0	1
s_5					0

Abbildung 6.20: Beispiel: Dynamische Programmierung für geliftetes PMSA

bei $w(a,a) = 0$ und $w(a,b) = 1$ für alle $a \neq b \in \overline{\Sigma}$. Im Folgenden geben wir die vollständige Dynamische Programmierung an, wobei wir allerdings annehmen, dass die Sequenzen an der Wurzel tatsächlich geliftet sind (also etwas anders als in der allgemeinen Rekursionsgleichung für die dynamische Programmierung). Dabei identifizieren wir die Blätter mit ihren Sequenzen.

$$\begin{aligned}
D[s,s_1] &= (d(s_1,s_1)+D[s_1,s_1])+(d(s_1,s_2)+D[s_2,s_2]) \\
&= 0+0+2+0 = 2 \\
D[s,s_2] &= (d(s_2,s_1)+D[s_1,s_1])+(d(s_2,s_2)+D[s_2,s_2]) \\
&= 2+0+0+0 = 2
\end{aligned}$$

$$\begin{aligned}
D[t,s_3] &= (d(s_3,s_3)+D[s_3,s_3])+(d(s_3,s_4)+D[s_4,s_4])+(d(s_3,s_5)+D[s_5,s_5]) \\
&= 0+0+2+0+1+0 = 3 \\
D[t,s_4] &= (d(s_4,s_3)+D[s_3,s_3])+(d(s_4,s_4)+D[s_4,s_4])+(d(s_4,s_5)+D[s_5,s_5]) \\
&= 2+0+0+0+1+0 = 3 \\
D[t,s_5] &= (d(s_5,s_3)+D[s_3,s_3])+(d(s_5,s_4)+D[s_4,s_4])+(d(s_5,s_5)+D[s_5,s_5]) \\
&= 1+0+1+0+0+0 = 2
\end{aligned}$$

$$\begin{aligned}
D[r,s_1] &= d(s_1,s_1)+D[s,s_1]+ \\
&\quad +\min\{d(s_1,s_3)+D[t,s_3], d(s_1,s_4)+D[t,s_4], d(s_1,s_5)+D[t,s_5]\} \\
&= 0+2+\min\{2+3,2+3,1+2\} = 5 \\
D[r,s_2] &= d(s_2,s_2)+D[s,s_2]+ \\
&\quad +\min\{d(s_2,s_3)+D[t,s_3], d(s_2,s_4)+D[t,s_4], d(s_2,s_5)+D[t,s_5]\} \\
&= 0+2\min\{3+3,2+3,3+2\} = 7 \\
D[r,s_3] &= \min\{d(s_3,s_1)+D[s,s_1], d(s_3,s_2)+D[s,s_2]\}+d(s_3,s_3)+D[t,s_3] \\
&= \min\{2+2,3+2\}+0+3 = 7 \\
D[r,s_4] &= \min\{d(s_4,s_1)+D[s,s_1], d(s_4,s_2)+D[s,s_2]\}+d(s_4,s_4)+D[t,s_4] \\
&= \min\{2+2,2+2\}+0+3 = 7 \\
D[r,s_5] &= \min\{d(s_5,s_1)+D[s,s_1], d(s_5,s_2)+D[s,s_2]\}+d(s_5,s_5)+D[t,s_5] \\
&= \min\{1+2,3+2\}+0+2 = 5
\end{aligned}$$

Damit erhalten wir die folgenden beiden Lösungen:

$r = \text{ACCA}$	$r = \text{ACC}$
$s = \text{ACCA}$	$s = \text{ACCA}$
$t = \text{ACC}$	$t = \text{ACC}$

Wenn wir uns bei der dynamischen Programmierung auf geliftete phylogenetische mehrfache Sequenzen-Alignments beschränken wollen, kann man mit einer etwas trickreichen Analyse den Summanden k^3 noch auf k^2 drücken. Dazu benötigen wir erst noch eine Definition.

Definition 6.69 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Ein Paar $(s, s') \in S^2$ wird als ein* legales Paar *für eine Kante $(v, v') \in E(T)$ bezeichnet, wenn entweder $s = s' \in S(v')$ oder $s \in S(v) \setminus S(v')$ sowie $s' \in S(v')$ gilt.*

Man überlegt sich leicht, dass für die Berechnung von $D(v, s)$ für $v \in V(T)$ und $s \in S$ bei der Betrachtung des Kindes v' von v nur legale Paare für die Kante (v, v') zu berücksichtigen sind.

Lemma 6.70 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Jedes Paar $(s, s') \in S^2$ mit $s \neq s'$ kann nur für eine Kante in T legal sein.*

Beweis: Sei $(s, s') \in S^2$ mit $s \neq s'$. Sei weiter v der niedrigste gemeinsame Vorfahre der Blätter, die mit s bzw. s' markiert sind (siehe auch Abbildung 6.21). Sei weiter

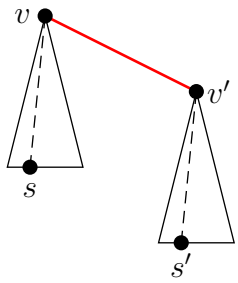


Abbildung 6.21: Skizze: Kante zu der (s, s') legal sein kann

v' das Kind von v, das auf einem einfachen Pfad von v zu dem Blatt mit Label s' liegt. Offensichtlich kann (s, s') nur für die Kante (v, v') legal sein, sonst gäbe es nicht geliftete Knoten im Baum. ■

Wir wollen nun die Anzahl legaler Paare, die für alle Kanten auftreten können, abschätzen. Dazu benötigen wir noch eine kurze Definition.

Definition 6.71 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Das Paar $((s, s'), (v, v'))$ wird als* legales Kanten-Paar *bezeichnet, wenn (s, s') ein legales Paar für $(v, v') \in E(T)$ ist.*

Mit dieser Definition können wir das folgende Korollar formulieren.

Korollar 6.72 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ und sei T ein zu S konsistenter Baum. Es gibt nur $O(k^2)$ legale Kanten-Paare im Baum T.*

Beweis: Sei $(s, s') \in S^2$ mit $s \neq s'$. Dann gibt es nach dem vorherigen Lemma nur eine Kante, zu dem dieses Paar legal sein kann. Da es $O(k^2)$ solcher Paare $(s, s') \in S^2$ gibt, kann es hiervon maximal $O(k^2)$ legale Kanten-Paare geben.

Sei nun $(s, s) \in S^2$. Dieses Paar kann ein legales Paar für maximal $O(k)$ Kanten sein (mehr Kanten hat T nicht). Da es genau k solcher Paare gibt, kann es auch hiervon maximal $O(k^2)$ legale Kanten-Paare geben. ■

Wenn wir also wissen, welche Paare für eine Kante legal sind, müssen wir pro Kante nicht mehr wie bisher $O(k^2)$ Sequenzen ausprobieren (also insgesamt $O(k^3)$), sondern nur insgesamt $O(k^2)$ ausprobieren. Wir müssen uns also nur einen Algorithmus überlegen, der die Menge legaler Paare für alle Kanten aufzählt.

Wir betrachten dazu eine Kante $(v, w_i) \in E(T)$. Wir nehmen an, wir hätten für jedes Kind w_j von v für $j \in [1 : r]$ bereits die Menge $S(w_j)$ berechnet. Nach dem vorherigen Lemma ist dann die Menge der legalen Paare gerade $L(v, w_i)$, wobei

$$L(v, w_i) = \left(\bigcup_{\substack{j=1 \\ j \neq i}}^{r} S(w_j) \right) \times S(w_i) \ \cup \ \{(s, s) \ : \ s \in S(w_i)\} .$$

Die Mengen $S(v)$ lassen sich im Baum T bottom-up sehr leicht berechnen. Für die Blätter sind diese Mengen ja schon definiert. Sind für einen Knoten v für alle seine Kinder $w_1, \ldots, w_r$ die Mengen $S(w_1), \ldots, S(w_r)$ bekannt, dann ist

$$S(v) = \bigcup_{j=1}^{r} S(w_j).$$

Wir können diese als lineare Listen leicht verwalten und bei geeigneter Implementierung in konstanter Zeit zwei Mengen vereinigen (Listen konkatenieren). Die Konstruktion der Mengen $S(v)$ kann also in Zeit $O(k)$ erledigt werden. Also ist der Zeitbedarf für die Konstruktion legaler Paare insgesamt $O(k^2)$, da es nach dem Lemma ja nur nur $O(k^2)$ legale Kanten-Paare gibt.

Für jede Kante kann die Liste legaler Paare in Zeit proportional zur Anzahl dieser legalen Paare erzeugt werden. Die Laufzeit ergibt sich wie folgt, wobei $L(v, w)$ die

Menge aller legaler Paare der Kante (v, w) ist:

$$\sum_{v \in I} \sum_{(v,w) \in E(T)} O(|L(v,w)|) = O\left(\sum_{(v,w) \in E(T)} |L(v,w)| \right) = O(k^2).$$

Tatsächlich implementieren wir dann die folgende Rekursionsgleichung mittels dynamischer Programmierung:

$$D(v,s) = \begin{cases} 0 & \text{falls } v \text{ ein Blatt mit } s_v = s, \\ \sum_{(v,w) \in E(T)} \min_{(s,s') \in L(v,w)} \{d(s,s') + D(w,s')\} & \text{sonst.} \end{cases}$$

Beachte hierbei, dass $\min\{\} = \infty$ gilt, das heißt, dass sobald in der Summe eine der Mengen in der Minimumsbildung leer ist, ist der Wert ∞ (insbesondere wenn v ein Blatt mit $s_v \neq s$ ist).

Somit erhalten wir das folgende Theorem.

Theorem 6.73 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ mit $|s_i| = O(n)$ und sei T ein zu S konsistenter Baum. Ein phylogenetisches mehrfaches Alignment für S und T, dessen Distanz maximal um 2 von einem optimalen phylogenetischen mehrfachen Alignment für S und T abweicht, kann in Zeit $O(k^2 n^2)$ konstruiert werden.*

12.12.19

6.6.6 Berechnung eines optimal uniform gelifteten PMSA

Eine weitere Verbesserung kann mit einer modifizierten Version dieses Algorithmus erzielt werden. Wir erinnern zunächst an die Definition eines Levels in einem gewurzelten Baum.

Definition 6.74 *Sei $T = (V, E)$ ein gewurzelter Baum. Der* Level $\ell(v)$ *eines Knotens $v \in V$ ist die Länge des einfachen Pfades (dies entspricht der Anzahl der Kanten auf dem Pfad) von der Wurzel des Baumes T zu diesem Knoten v. Weiterhin bezeichnet $V_i = \{v \in V : \ell(v) = i\}$ die Menge aller Knoten, die sich auf dem Level i befinden.*

Als nächstes wollen wir noch kurz vollständige Bäume definieren.

Definition 6.75 *Sei T ein gewurzelter Baum. T heißt k-*är*, wenn jeder Knoten maximal k Kinder besitzt. T heißt* binär*, wenn er 2-är ist. Ein k-ärer Baum heißt* regulär*, wenn jeder innere Knoten genau k Kinder besitzt. Ein k-ärer Baum heißt* vollständig*, wenn er regulär ist und sich alle Blätter auf demselben Level befinden.*

Wir werden uns im Folgenden der Einfachheit halber auf vollständige binäre Bäume beschränken. Dass jeder Knoten maximal zwei Kinder hat, ist aus biologischer Sicht nicht so einschränkend, wie dass sich alle Blätter auf demselben Level befinden. Wir können aber solche unvollständigen Bäume einfach zu einem vollständigen Baum erweitern, indem man die neuen Blätter alle mit der Sequenz markiert sind, mit der auch ihr Vorgänger (ein ursprüngliches Blatt) markiert ist.

Definition 6.76 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum. Ein zu T gelifteter Baum T' heißt* uniform*, wenn für jeden Knoten eines Levels entweder alle gelifteten Sequenzen vom linken oder alle vom rechten Kind stammen.*

Gegenüber einem normalen Lifting ist dies eine starke, künstlich wirkende Einschränkung. Den einfachen Beweis des folgenden Lemmas überlassen wir dem Leser.

Lemma 6.77 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum. Eine Markierung der Wurzel von T mit $s \in S$ beschreibt eindeutig ein uniformes Lifting in T.*

Notation 6.78 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum. Mit T^s bezeichnen wir den uniform gelifteten Baum, so dass die Wurzel mit $s \in S$ markiert ist.*

Somit gibt es in einem vollständigen binären Baum genau k verschiedene uniforme Liftings. Im Gegensatz zu normalen binären Bäumen gibt es verhältnismäßig wenige uniforme Liftings. In normalen Bäumen kann es $2^{O(k)}$ verschiedene Liftings geben.

Lemma 6.79 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum der Tiefe d. Es gibt genau 2^d verschiedene uniforme Liftings für T.*

Im Falle unseres vollständigen binären Baumes gilt dann natürlich sofort $2^d = k$, d.h. $d = \log(k)$.

Notation 6.80 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum. Für einen inneren Knoten $v \in V(T)$ mit Markierung s bezeichnet $\bar{v}$ bzw. $\hat{v}$ das Kind mit der Markierung s bzw. s'. Hierbei ist s' die Markierung der beiden Kinderknoten von v, für die $s' \neq s$ gilt.*

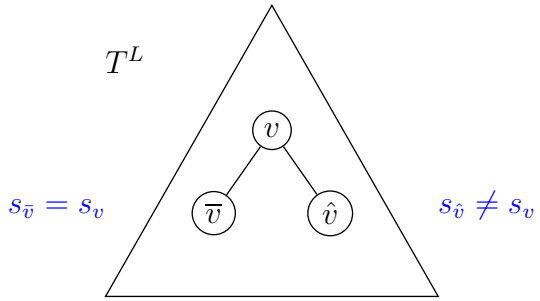


Abbildung 6.22: Skizze: Bild zur Notation der gelifteten Kindsequenzen

In Abbildung 6.22 ist diese Notation noch einmal visualisiert. Mit diesen Notationen können wir das folgende grundlegende Lemma formulieren.

Lemma 6.81 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum der Tiefe d. Es gilt*

$$\sum_{s \in S} D(T^s) \leq 2 \sum_{s \in S} \sum_{v \in V(T)} D^*(p^s_{v,\bar{v}}) = 2 \sum_{s \in S} \sum_{v \in V(T)} D^*(p^s_{v,\hat{v}}).$$

Hierbei bezeichnet $p^s_{v,w}$ den Pfad von v über w zu dem Blatt im Baum T^s, das dieselbe Markierung wie w trägt. Weiterhin bezeichnet $D^*(p)$ die Summe der Kantengewichte im Pfad p im Baum T, wobei die inneren Knoten mit den Sequenzen markiert sind, die zu einem optimalen phylogenetischen mehrfachen Sequenzen-Alignment gehören.

Beweis: Sei T^* der Baum, der zu einem optimalen phylogenetischen mehrfachen Sequenzen-Alignment für (S, T) gehört, wobei s^*_v die dem Knoten v zugeordnete Sequenz ist. Für die Kante $v, \overline{v}$ gilt wiederum $D_{T^s}(v, \overline{v}) = 0$ und mithilfe der Dreiecksungleichung gilt mit der Markierung s^*_v für v im Baum T^* für das optimale phylogenetische mehrfachen Sequenzen-Alignment für die Distanz der Kante $(v, \hat{v})$ in T^s:

$$\begin{aligned}
D_{T^s}(v, \hat{v}) &= d(s_v, s_{\hat{v}}) \\
&\leq d(s_v, s^*_v) + d(s^*_v, s_{\hat{v}}) \\
&\quad \text{da } s_v = s_{\bar{v}} \\
&= d(s^*_v, s_{\bar{v}}) + d(s^*_v, s_{\hat{v}}) \\
&\quad \text{mit } p^s_{v,\bar{v}} = (v, v_1, \ldots, v_\ell) \text{ und } p^s_{v,\hat{v}} = (v, v'_1, \ldots, v'_\ell) \\
&\leq d(s^*_v, s^*_{v_1}) + d(s^*_{v_1}, s_{\bar{v}}) + d(s^*_v, s^*_{v'_1}) + d(s^*_{v'_1}, s_{\hat{v}}) \\
&\vdots
\end{aligned}$$

$$
\begin{aligned}
&\vdots \\
\leq\ & d(s^*_v, s^*_{v_1}) + d(s^*_{v_1}, s^*_{v_2}) + \cdots + d(s^*_{v_{\ell-1}}, s^*_{v_\ell}) + d(s^*_{v_\ell}, s_{\bar{v}}) \\
& + d(s^*_v, s^*_{v'_1}) + d(s^*_{v'_1}, s^*_{v'_2}) + \cdots + d(s^*_{v'_{\ell-1}}, s^*_{v'_\ell}) + d(s^*_{v'_\ell}, s_{\hat{v}}) \\
& \text{da } s^*_{v_\ell} = s_{\bar{v}} \text{ und } s^*_{v'_\ell} = s_{\hat{v}} \\
=\ & d(s^*_v, s^*_{v_1}) + d(s^*_{v_1}, s^*_{v_2}) + \cdots + d(s^*_{v_{\ell-1}}, s^*_{v_\ell}) \\
& + d(s^*_v, s^*_{v'_1}) + d(s^*_{v'_1}, s^*_{v'_2}) + \cdots + d(s^*_{v'_{\ell-1}}, s^*_{v'_\ell}) \\
=\ & D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}}).
\end{aligned}
$$

Die Ungleichungen folgen aus der mehrfachen Anwendung der Dreiecksungleichung auf alle Knoten im Pfad $p^s_{v,\bar{v}}$ bzw. $p^s_{v,\hat{v}}$ analog zum Beweis von Satz 6.65. Damit gilt

$$D(T^s) = \sum_{v \in V(T)} D_{T^s}(v, \hat{v}) \leq \sum_{v \in V(T)} (D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}})).$$

Als nächstes zeigen wir, dass für ein festes $v \in V(T)$ und jedes $s \in S$ genau ein $s' \in S$ mit $s' \neq s$ gibt, so dass

$$D^*(p^s_{v,\bar{v}}) = D^*(p^{s'}_{v,\hat{v}}) \qquad \text{und} \qquad D^*(p^s_{v,\hat{v}}) = D^*(p^{s'}_{v,\bar{v}}).$$

Dazu muss im Lifting von T^s nur das Lifting von v zu seinen Kindern vertauscht werden. Hierbei ist zu beachten, dass die Pfade über die summiert wird, zwar in den uniform gelifteten Bäumen T^s und $T^{s'}$ anders definiert werden, aber die Menge der Pfade in T identisch bleibt und im optimalen uniform gelifteten Baum T^* dieselbe Summe besitzen. Damit ergibt sich

$$\sum_{s \in S} (D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}})) = 2 \sum_{s \in S} (D^*(p^s_{v,\bar{v}}))$$

bzw. analog

$$\sum_{s \in S} (D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}})) = 2 \sum_{s \in S} (D^*(p^s_{v,\hat{v}})).$$

Damit folgt:

$$
\begin{aligned}
\sum_{s \in S} D(T^s) &\leq \sum_{s \in S} \sum_{v \in V(T)} (D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}})) \\
&= \sum_{v \in V(T)} \sum_{s \in S} (D^*(p^s_{v,\bar{v}}) + D^*(p^s_{v,\hat{v}})) \\
&= \sum_{v \in V(T)} 2 \cdot \sum_{s \in S} D^*(p^s_{v,\hat{v}}) \\
&= 2 \cdot \sum_{s \in S} \sum_{v \in V(T)} D^*(p^s_{v,\hat{v}})
\end{aligned}
$$

Die andere Ungleichung folgt analog. Damit ist das Lemma bewiesen. ■

Damit erhalten wir sofort das folgende Lemma.

Lemma 6.82 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum der Tiefe d. Sei weiter T^* ein optimales phylogenetisches Alignment für (S, T). Dann gilt*

$$\sum_{s \in S} D(T^s) \leq 2 \cdot |S| \cdot D(T^*) = 2^{d+1} \cdot D(T^s).$$

Beweis: Es gilt:

$$\begin{aligned}
\sum_{s \in S} D(T^s) &\leq 2 \sum_{s \in S} \sum_{v \in V(T)} D^*(p^s_{v,\hat{v}}) \\
&\quad \text{da die Pfade ja wieder kantendisjunkt sind} \\
&\leq 2 \sum_{s \in S} \sum_{(v,w) \in E(T)} D^*(v, w) \\
&= 2 \sum_{s \in S} D(T^*) \\
&= 2 \cdot |S| \cdot D(T^*) \\
&\quad \text{mit } |S| = 2^d \\
&= 2^{d+1} D(T^*).
\end{aligned}$$

und damit ist der Satz bewiesen. ■

Durch leichte Umformung erhalten wir sofort

$$\frac{1}{|S|} \sum_{s \in S} D(T^s) \leq 2 \cdot D(T^*).$$

Damit ist der Mittelwert der Distanzen über alle uniformen gelifteten phylogenetischen mehrfachen Sequenzen-Alignment höchstens um den Faktor 2 schlechter als das optimale phylogenetische mehrfache Sequenzen-Alignment. Daraus können wir sofort das folgende gewünschte Korollar ableiten.

Korollar 6.83 *Sei $S \subseteq \Sigma^*$ und sei T ein zu S konsistenter vollständiger binärer Baum der Tiefe d. Sei weiter T^* ein optimales phylogenetisches Alignment für (S, T). Es existiert ein uniform geliftetes Alignment T' mit $D(T') \leq 2D(T^*)$.*

Wir können nun wieder mit Hilfe der dynamischen Programmierung ein optimales uniform geliftetes Alignment konstruieren. Dazu sei wiederum $D(v, s)$ mit $v \in V(T)$

und $s \in S$ die optimale Distanz eines uniform gelifteten Alignments für den am Knoten v gewurzelten Teilbaum T_v mit der Markierung s.

Für alle $s \notin S(v)$ gilt $D(v,s) = \infty$ und für alle Blätter von v von T mit Markierung s gilt $D(v,s) = 0$. Sei v ein interner Knoten v und $s \in S(v)$ eine Sequenz. Für die beiden Kinder $\bar{v}$ und $\hat{v}$ von v gilt nach unserer Konvention, dass $s \in S(\bar{v})$. Aufgrund des uniformen Liftings ist dann die Sequenz $s' \in S(\hat{v})$ für das zweite Kind dann durch s auch schon eindeutig bestimmt. Es gilt dann:

$$D(v,s) = \begin{cases} \infty & \text{falls } s \notin S(v), \\ 0 & \text{falls } v \text{ ein Blatt}, \\ D(\bar{v},s) + D(\hat{v},s') + d(s,s') & \text{sonst.} \end{cases}$$

Wie kann man nun leicht s' aus s bestimmen. Dazu nehmen wir an, dass die Blätter von links nach rechts mit 0 bis $2^d - 1$ durchnummeriert sind (siehe auch Abbildung 6.23). Ist ein Knoten auf Level ℓ (die Wurzel hat Level 0) und die zugehörige geliftete Sequenz hat Nummer j, dann findet man den zugehörigen String des anderen Kindes an dem Knoten Nummer j', wobei j durch Tauschen des $(\ell+1)$-ersten Bits geschieht. Ein Knoten auf Level 0 bzw. Level 1 mit Sequenz $s = s_5$ findet seinen anderen String $s' = s_1$ bzw. $s' = s_7$, da das erste bzw. zweite Bit (von vorne) komplementiert wurde.

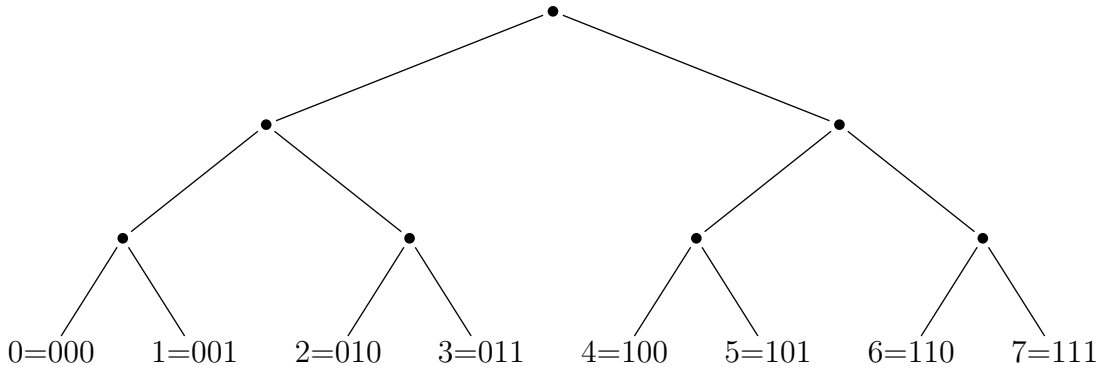


Abbildung 6.23: Skizze: Nummerierung der Blattsequenzen

Im Falle uniformer Liftings kann man nun zeigen, dass es insgesamt nur $2dk$ legale Kanten-Paare in T gibt. Man betrachte dazu für eine Sequenz $s \in S$ den Pfad vom Blatt v mit $s_v = s$ zur Wurzel. An jedem inneren Knoten v auf diesem Pfad ist aufgrund des *uniformen* Lifting klar, welche Sequenz $s' \in S$ am Knoten $\hat{v}$ stehen muss. Somit hat diese Sequenz in einem vollständigen Baum der Tiefe d genau d legale Kanten-Paare mit $s \neq s'$ und genau d legale Kantenpaare mit $s = s'$. Da es k Sequenzen in S gibt, folgt die Behauptung.

Die Rekursion lässt sich in dann Zeit $O(dkn^2)$ lösen. Für die Rekursion werden zum einen allerdings nur die dk legalen Kantenpaare mit verschiedenen Sequenzen

benötigt und zum anderen nur die dk paarweisen Distanzen $d(s, s')$ benötigt die in der Rekursion tatsächlich auftauchen.

Theorem 6.84 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ mit $|s_i| = O(n)$ und sei T ein zu S konsistenter vollständiger binärer Baum der Tiefe d. Ein phylogenetisches mehrfaches Alignment für S und T, dessen Distanz maximal um 2 von einem optimalen phylogenetischen mehrfachen Alignment für S und T abweicht, kann in Zeit $O(dkn^2 + k^2n)$ konstruiert werden.*

Eine Verallgemeinerung auf beliebige q-äre Bäume ist möglich. Wir halten hier nur das Ergebnis ohne Beweis fest und verweisen den Leser auf die Originalliteratur von Lushent Wang und Dan Gusfield.

Theorem 6.85 *Sei $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ mit $|s_i| = O(n)$ und sei T ein zu S konsistenter regulärer q-ärer Baum der Tiefe d. Ein phylogenetisches mehrfaches Alignment für S und T, dessen Distanz maximal um 2 von einem optimalen phylogenetischen mehrfachen Alignment für S und T abweicht, kann in Zeit $O(dkn^2+k^2n)$ konstruiert werden.*

6.6.7 Polynomielles Approximationsschema

Wir wollen zum Abschluss noch kurz skizzieren, wie man für das phylogenetische Alignment ein polynomielles Approximationsschema konstruieren kann.

Definition 6.86 *Sei $S \subseteq \Sigma^*$ und sei T ein gewurzelter Baum der Tiefe d. Für $v \in V(T)$ und $t \in \mathbb{N}$ ist der Teilbaum $T_{v,t} = T[V']$ mit*

$$V' = \left\{ v' \in V(T) \,:\, v \xrightarrow{*} v' \wedge \ell(v') - \ell(v) \leq t \right\},$$

*eine t-*Komponente *von T, wobei $T[V'] = (V(T) \cap V', E(T) \cap (V' \times V'))$ der durch die Knotenmenge V' induzierte Teilbaum von T ist.*

17.12.19

Wir setzen nun $G_i = \bigcup_{j \equiv i \,(\mathrm{mod} t)} V_j$, wobei $V_j \subseteq V$ die Teilmenge der Knoten ist, die sich auf Level j befinden. Dann erlaubt der Baum T für jedes $i \in [0 : t-1]$ die folgende Darstellung:

$$T = T_{r(T),i} \cup \bigcup_{v \in G_i} T_{v,t} =: T_i.$$

Dabei werden den Knoten aus $G_i \cup \{r(T)\}$ in T_i genau die Sequenzen zugewiesen, die den Knoten auch im Baum T in einem optimalen *gelifteten* Alignment zugeordnet

werden. Diesen so markierten Baum bezeichnen wir mit $\hat{T}_i$ für $i \in [0:t]$. Wenn wir jetzt für jede t-Komponente von $T_{v,t}$ von $\hat{T}_i$ ein optimales phylogenetisches mehrfaches Alignment $T'_{v,t}$ bestimmen und diese zu einem phylogenetisches mehrfaches Alignment T'_i zusammensetzen, dann kann man zeigen, dass gilt

$$\sum_{i=0}^{t-1} D(T'_i) \leq (t+3)D(T^*),$$

wobei T^* ein optimales phylogenetisches mehrfaches Alignment für (S,T) ist. Wir führen also für jede t-Komponente eine unabhängige lokale Optimierung durch.

Theorem 6.87 *Sei T ein zu $S \subseteq \Sigma^*$ konsistenter Baum und sei $t \in \mathbb{N}$. Sei weiter T^* ein optimales phylogenetisches mehrfaches Sequenzen-Alignment für S und T und seien T'_i mit $i \in [0:t-1]$ die wie eben beschrieben konstruierten phylogenetischen mehrfachen Sequenzen-Alignments. Dann existiert ein $i \in [0:t-1]$, so dass $D(T'_i) \leq (1+3/t) \cdot D(T^*)$.*

Wir müssen also nur noch für die t-Komponenten ein optimales phylogenetisches mehrfaches Alignment berechnen und das beste unter allen (also ein geeignetes $i \in [0:t-1]$) auswählen. Hierfür gibt es mittlerweile verschiedene Algorithmen unterschiedlicher Effizienz. Wir halten hier nur den folgenden Satz fest.

Theorem 6.88 *Es existiert ein polynomielles Approximationsschema für das phylogenetische mehrfache Sequenzen-Alignment, wenn eine metrische Distanzfunktion zugrunde gelegt wird.*

Wenn die zugrunde liegende Kostenfunktion die Dreiecksungleichung nicht erfüllen muss, kann man zeigen, dass die Ermittlung eines phylogenetischen mehrfachen Alignments bereits $\mathcal{APX}$-hart ist.

Wir merken hier nur noch an, dass für eingeschränkte Versionen des SP-Alignments und Konsensus-Alignments ebenfalls polynomielle Approximationsschema bekannt sind. Hierbei wird angenommen, dass die Leerzeichen in dem resultierenden Alignment relativ gleichmäßig verteilt sind. Andernfalls ist es noch unbekannt, ob es dann immer noch ein polynomielles Approximationsschema gibt.

6.7 Heuristische Methoden

In diesem Abschnitt wollen wir noch kurz ein paar heuristische Methoden vorstellen, die in der Praxis oft Anwendung finden. Des Weiteren wollen wir auch die verwen-

deten algorithmischen Ideen der beiden gebräuchlichsten Tools zur Suche in großen Sequenz-Datenbanken vorstellen: FASTA und BLAST.

6.7.1 Progressives Alignment

Das progressive Alignment ist eine Variante eines Alignments, dass zu einem Baum konsistent ist. Dabei wird zuerst aus den Sequenzen selbst der Baum erstellt. Dazu wird zunächst wieder für jedes Paar von Sequenzen aus $S = \{s_1, \ldots, s_k\} \subseteq \Sigma^*$ ein optimales paarweise Alignment erstellt. Dann wird aufgrund dieser Ähnlichkeitsmatrix ein Baum (auch Dendrogramm genannt) konstruiert. Mit Hilfe des so konstruierten Baumes wird dann das mehrfache Sequenzen-Alignment konstruiert. CLUSTAL und T-Coffee sind Beispiele für progressive Alignment-Methoden.

Das prinzipielle Problem hierbei ist, dass man versucht mit mehrfachen Sequenzen-Alignments nicht zuletzt die Phylogenie zu rekonstruieren und dabei eben diese Phylogenie benötigt. Somit hat man das bekannte Henne-Ei-Problem. Man kann dennoch versuchen, aus den gegebenen Sequenzähnlichkeiten erst einmal eine grobe Phylogenie zu rekonstruieren, die für das mehrfache Sequenzen-Alignment noch brauchbar ist. Das Hauptproblem bei progressiven Alignments ist eine fälschliche Ähnlichkeit von Sequenzen, die dann über das gesamte mehrfache Sequenzen-Alignment durchgezogen wird.

Für die Erstellung der Bäume stehen zum einen bekannte Algorithmen für minimale (bzw. maximale) Spannbäume als auch die gängigen hierarchische Clustering-Verfahren (agglomerative und divisive) zur Verfügung, wie beispielsweise die folgenden Verfahren:

- Single Linkage Clustering (auch als Nearest-Neighbor bekannt),
- Complete-Linkage-Clustering (auch als Farthest-Neighbor bekannt),
- Average-Linkage-Clustering (auch als UPGMA bekannt),
- Weighted-Average-Linkage-Clustering (auch als WPGMA genannt), etc.

Beim Clustering versucht man ähnliche Objekte (was immer das genau heißt, dies wird meist durch eine Distanz beschrieben) in Gruppen zusammenzufassen. Beim agglomerativen hierarchischen Clustering versucht man ausgehend von der Menge der einelementigen Mengen der Objekte durch Zusammenfassen der beiden ähnlichsten Gruppen (Cluster) eine neue Partition der Objekte zu finden. Dabei wird neben einer Distanz von Objekten, also $d : S^2 \to \mathbb{R}_+$ auch eine Distanz auf disjunkten Teilmengen von S (Clustern) benötigt, also $d' : (2^S)^2 \to \mathbb{R}_+$, die in der Regel von d

Clustering (data [] a; int[][] d)

```
begin
    P := {{a_i} : i ∈ [1 : n]};
    T := (P, ∅);
    while (|P| > 1) do
        let A, B ∈ P s.t. d(A, B) is minimal;
        X = A ∪ B;
        P := (P ∪ {X}) \ {A, B};
        T := T ∪ ({X}, {(X, A), (X, B)});
        forall (Y ∈ P) do
            update d(X, Y);
    return T;
end
```

Abbildung 6.24: Algorithmus: Generisches agglomeratives hierarchisches Clustering

induziert wird. In Abbildung 6.24 ist der generische Algorithmus zum agglomerativen hierarchischen Clustering basierend auf einer Distanzfunktion gegeben.

Die beim Clustering entstehenden verschiedenen Partitionen induzieren eine Graphstruktur, besser eine Baumstruktur, mittels der folgenden Definition.

Definition 6.89 *Sei S eine Menge und sei $V \subseteq 2^S$. Dann ist $G = (V, E)$ mit $E = \{(A, B) : A \supseteq B\}$ der von V* induzierte Graph.

Beim agglomerativen hierarchischen Clustering entstehen bei der Vereinigung von Punktmengen solche induzierte Graphen, die dann eben gewurzelte Bäume darstellen (die Dendrogramme). Man kann sie auch als ungewurzelte, d.h. freie Bäume interpretieren.

Die erste Gleichung stellt in den folgenenden Definitionen die eigentliche Definition der Distanz dar, die zweite Gleichung eine einfachere Berechnung der Distanzen, wenn andere Distanzen (wie eben beim agglomerativen hierarchischen Clustering) bekannt sind.

Definition 6.90 *Sei S eine Menge und sei $d : S^2 \to \mathbb{R}_+$ ein Distanzfunktion auf S. Für $X = A \cup B$ mit $A, B \in 2^S$ ist die induzierte* single-linkage-Distanz *für X mit einer beliebigen anderen Menge $Y \in 2^S$ gegeben durch:*

$$d(X, Y) := \min \{d(x, y) \ : \ x \in X \wedge y \in Y\} = \min(d(A, Y), d(B, Y)).$$

Definition 6.91 *Sei S eine Menge und sei $d : S^2 \to \mathbb{R}_+$ ein Distanzfunktion auf S. Für $X = A \cup B$ mit $A, B \in 2^S$ ist die induzierte* complete-linkage-Distanz *für X mit einer beliebigen anderen Menge $Y \in 2^S$ gegeben durch:*

$$d(X, Y) := \max\{d(x, y) \,:\, x \in X \wedge y \in Y\} = \max(d(A, Y), d(B, Y)).$$

Der Abstand ist hier allgemein definiert, beim agglomerativen Clustering wird jedoch dabei immer sowohl $A \cap B = \emptyset$ als auch $X \cap Y = \emptyset$ gelten. Im Folgenden wird jedoch explizit vorausgesetzt, dass A und B eine disjunkte Zerlegung von X ist.

Definition 6.92 *Sei S eine Menge und sei $d : S^2 \to \mathbb{R}_+$ ein Distanzfunktion auf S. Für $X = A \cup B$ mit $A, B \in 2^S$ und $A \cap B = \emptyset$ ist die induzierte* average-linkage-Distanz *(auch* UPGMA *für unweighted pair group method with arithmetic mean) für X mit einer beliebigen anderen Menge $Y \in 2^S$ gegeben durch:*

$$d(X, Y) = \frac{\sum_{x \in X} \sum_{y \in Y} d(x, y)}{|X||Y|} = \frac{|A|d(A, Y) + |B|d(B, Y)}{|A| + |B|}.$$

Definition 6.93 *Sei S eine Menge und sei $d : S^2 \to \mathbb{R}_+$ ein Distanzfunktion auf S. Für $X = A \cup B$ mit $A, B \in 2^S$ und $A \cap B = \emptyset$ ist die induzierte* weighted-average-linkage-Distanz *(auch* WPGMA *für weighted pair group method with arithmetic mean) für X mit einer beliebigen anderen Menge $Y \in 2^S$ gegeben durch:*

$$d(X, Y) = \frac{d(A, Y) + d(B, Y)}{2}.$$

Für diese Verfahren können die neuen Distanzen also leicht aus den Distanzen des vorherigen Clustering-Schrittes bestimmt werden. Eine allgemeine Definition ist aber nur für die ersten der drei Verfahren möglich, was für WPGMA nicht zutrifft. Diese letzte Distanz hängt insbesondere auch vom Aufbau der Cluster-Struktur ab, so dass es keine allgemeine Formel für die Distanz von zwei Mengen gibt.

Eine andere Abart des progressiven Alignment ist das Mischen mehrerer mehrfacher Sequenzen-Alignments, insbesondere auch, wenn in beiden mehrfachen Sequenzen-Alignments bereits mehrere Sequenzen in beiden enthalten sind. Dies ist quasi eine Erweiterung eines mit einem Baum konsistenten Alignments.

Auch kann man sich überlegen, zwei mehrfache Sequenzen-Alignments selbst zu alignieren. Dafür muss man sich nur sinnvolle Scoring-Funktionen ausdenken, mit denen man die Ähnlichkeit von zwei Alignment-Spalten bewertet. In der Regel bleiben dabei aber alignierte Bereiche fix, es können nur neue Gaps zwischen den Spalten eingefügt werden.

Diese letzten beiden Varianten werden tatsächlich auch von CLUSTAL verwendet. Das mehrfache Sequenzen-Alignment wird dabei bottom-up im Baum generiert wobei die Reihenfolge der betrachteten inneren Knoten der Reihenfolge des entstehenden Knoten beim Clustering entspricht (d.h. die zu den jeweils am nächsten benachbarten Cluster gehörigen Alignments werden auch als nächstes vereinigt). Dabei werden an die inneren Knoten nicht direkt Sequenzen zugewiesen, sondern abstrakt für die Konsensus-Strings (ohne Leerzeichen) der beiden Alignments an den Kindern ein optimales paarweise Alignment erstellt. Anhand dieses optimalen Alignments werden ähnlich wie im zu einem Baum konsistenten mehrfachen Alignment die beiden mehrfachen Alignments gemischt. Die weiter unten im Baum generierten Alignments überleben dabei im gesamten mehrfachen Sequenzen Alignment, was eine Schwäche von CLUSTAL ist.

Tatsächlich werden bei CLUSTAL nicht die Konsensus-Strings (ohne Leerzeichen) aligniert, sondern quasi die Profile der beiden mehrfachen Sequenzen der Kinder aligniert. Dies entspricht abstrakt einem Alignment von Alignments. Dabei wird die Ähnlichkeit von zwei Spalten eines Alignments über die Summe der Ähnlichkeiten der einzelnen Zeichen der beiden Spalten des Alignments bestimmt. Bei CLUSTALW können dabei noch die einzelnen Sequenzen gewichtet werden (in der Regel in Abhängigkeit vom konstruierten Baum), wobei sehr ähnliche Sequenzen ein niedrigeres Gewicht bekommen, um das mehrfache Alignments nicht aufgrund sehr ähnlicher Sequenzen in diese Richtung zu ziehen.

6.7.2 Iteratives Alignment

Iterative Alignment-Verfahren sind eher ein Postprocessing-Schritt zur Verbesserung bestehender Alignments. Dort werden Teile (einige Sequenzen) eines mehrfachen Alignments unter veränderten Randbedingungen neu berechnet. Auch kann es wieder einen Baum geben, an den sich das Alignment orientiert. Hier werden für Teilbäume aufgrund des bereits berechnet Alignments neue Topologien konstruiert und darauf basierend für die beteiligten Sequenzen neu mehrfache Alignments berechnet, die dann wieder in das mehrfache Alignment integriert werden.

Bekannte iterative Methoden (die teilweise auch progressiv sind) sind unter anderem MUSCLE und DIALIGN. Bei MUSCLE wird zuerst mit einem Distanzbegriffe über in beiden Sequenzen vorkommenden k-mere ein Baum mittels Clustering erstellt. Das daraus resultierende mit diesem Baum konsistente Alignment wird dann verwendet, um einen genaueren Distanzbegriff für Sequenzpaare zu bekommen. Damit wird dann ein neuer Baum erstellt und daraus ein weiteres mehrfaches Alignment konstruiert. Zum Schluss wird dieses mehrfache Alignment (im eigentlichen iterativen Schritt) weiter verbessert. Hier wird durch Kantenelimination die Menge in zwei

Mengen aufgeteilt und für beide Teile ein mehrfaches Alignment berechnet. Das daraus erstellte gemeinsache mehrfachen Alignment wird bezüglich der Güte mit dem bisher besten verglichen und ggf. beibehalten. Das letzte wird so oft iteriert, bis eine Konvergenz oder eine bestimmte Anzahl Iterationen erreicht wird.

Das in Abschnitt 6.6.7 vorgestellt polynomielle Approximationsschema ist also so ein iteratives Verfahren. Auch kann man sich für phylogenetische Alignments vorstellen, zuerst ein geliftetes Alignments zu konstruieren und dann für jeden inneren Knoten mit Hilfe eines besseren Algorithmus ausgehend von den Nachbar-Sequenzen einen Steiner-String oder eine sehr gute Approximation hiervon zu berechnen.

6.7.3 FASTA (FAST All oder FAST Alignments)

Wir stellen hier nur die Hauptvariante von FASTA vor, es gibt auch noch zahlreiche abgeleitete Varianten, die zuerst Mitte der 80er Jahre von David J. Lipman und William R. Pearson entwickelt wurde. Im Folgenden suchen wir nach einer Sequenz s in einer Datenbank t.

(1) Wir wählen zuerst eine Konstante ktup (die in der Beschreibung von FASTA so genannt wird und die wir der Einfachheit halber im Folgenden k nennen werden) in Abhängigkeit vom Inhalt der Datenbank, wie z.B.:

$$k \;:=\; \text{ktup} = \begin{cases} 6 & \text{für DNS} \\ 2 & \text{für Proteine} \end{cases}$$

Dann suchen wir nach perfekten Treffern von Teilwörtern von s der Länge k in t, d.h. für solche Treffer (i, j) gilt $s_i \cdots s_{i+k-1} = t_j \cdots t_{j+k-1}$. Dies erfolgt mit Hilfe einer Hash-Tabelle entweder für die Datenbank oder für das Suchmuster. Da es nur wenige kurze Sequenzen gibt ($4^6 = 4096$ bei DNS und $20^2 = 400$ bei Proteinen), kann man für jede solche kurze Sequenz eine Liste mit den zugehörigen Positionen in t speichern, an der solche kurzen Sequenzen auftreten.

Diese kurzen Treffer von $s_i \cdots s_{i+k-1}$ werden *Hot Spots* genannt. Diese Hot Spots sind in der Abbildung 6.25 noch einmal in der (nicht wirklich berechneten) Tabelle für die Alignment-Distanzen visualisiert (also quasi ein Dot-Plot). Man beachte, dass sich dabei auch überlappende Hot Spots (aufgrund längerer gemeinsamer Teilsequenzen) auch zu längeren Diagonalen anhäufen können, was in der Abbildung allerdings so nicht illustriert ist (hier haben alle roten Diagonalen die Länge k).

Angenommen für jedes Wort aus $w \in \Sigma^k$ liegt eine Liste der Indexpositionen vor, an der w in t auftritt. Beim Durchlaufen der Teilwörter $s_i \cdots s_{i+k-1}$ wird dann jeweils das Trefferpaar (i, j) in eine Liste der Diagonalen $(j - i)$ aufgenommen.

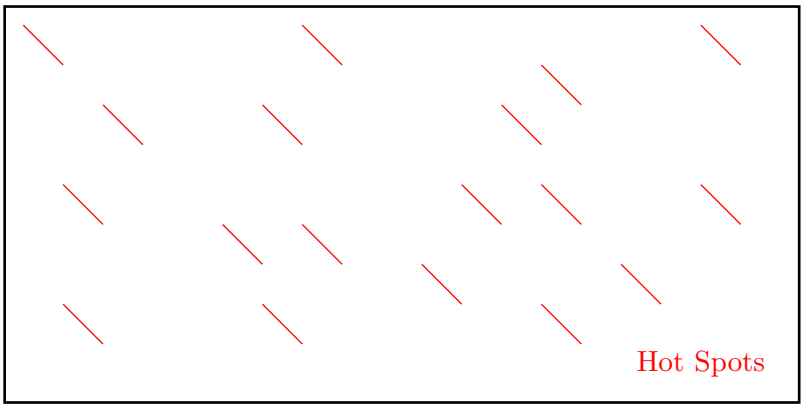


Abbildung 6.25: Skizze: Hot Spots

Innerhalb dieser Diagonalenliste treten die Hot Spots nun fortlaufend auf. Der Leser möge sich überlegen, warum dies so ist.

Man beachte, dass man k aufgrund modernerer Technologien mittlerweile auch wesentlich größer wählen könnte. Dann verliert man allerdings möglich Treffer, die eben nicht durch längere Bereiche exakter Übereinstimmung geprägt sind, da trotz guter Übereinstimmung auch immer Mismatches dazwischen liegen können.

(2) Jetzt werden auf den Diagonalen der Tabelle mit den Alignment-Distanzen (wiederum ohne diese explizit zu berechnen) so genannte *Diagonal Runs* gesucht. Das sind mehrere Hot Spots, die sich in derselben Diagonalen befinden, so dass die Lücken dazwischen kurz sind. Dies ist in Abbildung 6.26 noch einmal illustriert.

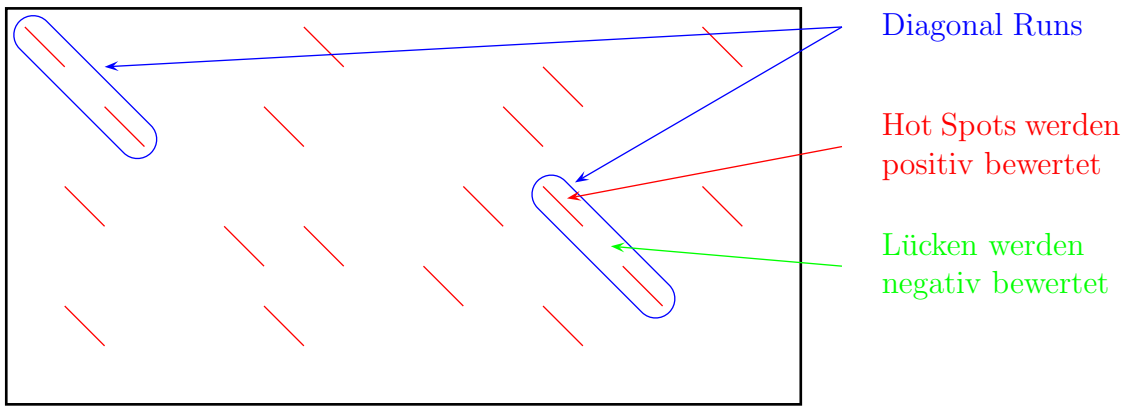


Abbildung 6.26: Skizze: Diagonal Runs

Dazu bewertet man die Hot-Spots positiv und die Lücken negativ, wobei längere Lücken einen kleineren (negativeren!) Wert erhalten als kürzere Lücken. Wir bewerten nun die Folgen von Hot Spots in ihren Diagonalen, ähnlich wie bei einem lokalen Alignment. Die etwa zehn besten werden zum Schluss aufgesammelt und in den Folgeschritten weiter bearbeitet.

Wir merken hier noch an, dass nicht alle Hot Spots einer Diagonalen in einem Diagonal Run zusammengefasst werden müssen und dass es in einer Diagonalen durchaus mehr als einen Diagonal Run geben kann.

(3) Nun erzeugen wir einen gerichteten Graphen. Die Knoten entsprechen den Diagonal Runs aus dem vorherigen Schritt und erhalten die positiven Gewichte, die im vorhergehenden Schritt bestimmt wurden.

Zwei Diagonal Runs werden mit einer Kante verbunden, wenn der Endpunkt des ersten Diagonal Runs links oberhalb des Anfangspunktes des zweiten Diagonal Runs liegt. Die Kanten erhalten wiederum ein negatives Gewicht, das entweder konstant oder proportional zum Abstand der Endpunkte ist.

Der so entstandene Graph ist azyklisch (d.h. kreisfrei, also ein DAG) und wir können darin wieder sehr einfach gewichtsmaximale Pfade suchen. Wir merken uns dann die besten gewichtsmaximalen Pfade für den nächsten Bearbeitungsschritt. Dieser Graph ist noch einmal in Abbildung 6.27 illustriert.

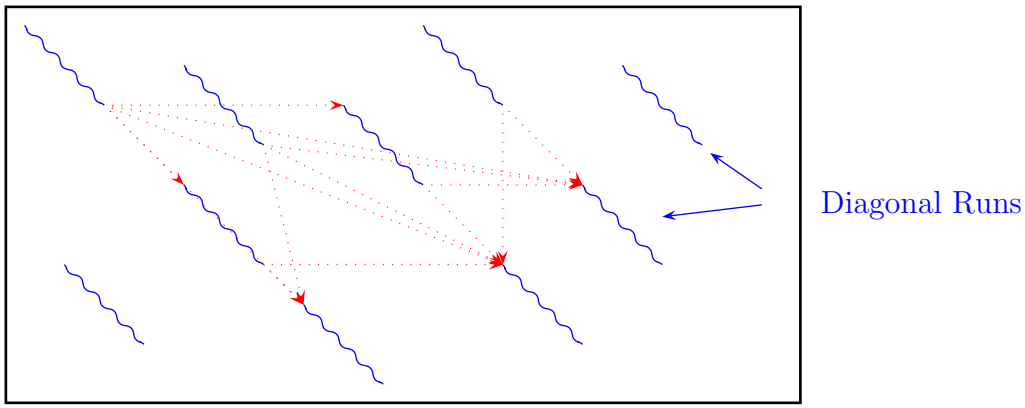


Abbildung 6.27: Skizze: Graph aus Diagonal Runs

(4) Für die gewichtsmaximalen Pfade aus Diagonal Runs berechnen wir ein semiglobales Alignment. Da wir nur an kleinen Distanzen interessiert sind, brauchen wir nur kleine Umgebungen dieser Pfade von Diagonal Runs zu berücksichtigen, was zu einer linearen Laufzeit (in $|s|$) führt. Dies ist in Abbildung 6.28 dargestellt.

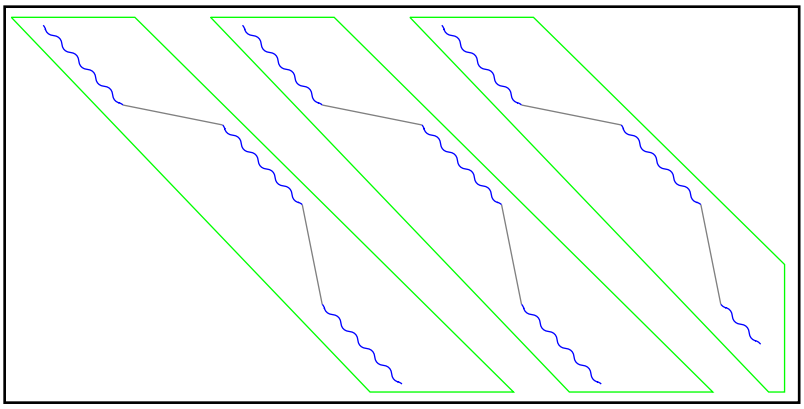


Abbildung 6.28: Skizze: Optimale Alignments um die Pfade aus Diagonal Runs

6.7.4 BLAST (Basic Local Alignment Search Tool)

Wir stellen hier nur die Hauptvariante vor, die zuerst in den 90er Jahren von Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, David J. Lipman eingeführt wurde. Es gibt auch noch zahlreiche abgeleitete Varianten. Wieder nehmen wir an, dass wir nach einer Sequenz s in einer Datenbank t suchen.

(1) Zuerst konstruieren wir alle Wörter $w \in \Sigma^k$ und testen, ob diese ähnlich zu einem Teilwort von s der Länge k ist, d.h. ob für das verwendete Ähnlichkeitsmaß S gilt:

$$S(s_i \cdots s_{i+k-1}, w) \geq \vartheta.$$

Ist dies der Fall, so nehmen wir dieses Wort in die von uns konstruierte Suchmustermenge $M_i = \left\{w \in \Sigma^k \,:\, S(s_i \cdots s_{i+k-1}, w) \geq \vartheta\right\}$ auf. Hierbei wird k auch wieder relativ klein gewählt:

$$k \;:=\; \begin{cases} k \in [3:5] & \text{für Proteine,} \\ k \approx 12 & \text{für DNS.} \end{cases}$$

Diese Menge $M = \bigcup_i M_i$ beinhaltet nun Wörter, die ziemlich ähnlich zu Teilwörtern aus dem ursprünglichen Suchmuster s sind. Der Vorteil ist der, dass wir die Fehler jetzt extrahiert haben und im Weiteren mit einer exakten Suche nach Wörtern in M weitermachen können. Damit können wir jetzt nach längeren Mustern in kürzerer Zeit bei gleicher Sensitivität exakt suchen.

(2) Jetzt suchen wir in der Datenbank t nach Wörtern aus M, z.B. mit Hilfe des Algorithmus von *Aho-Corasick* (bzw. einem ähnlichem Algorithmus)) oder auch wieder mithilfe eine Hash-Tabelle, und merken uns die Treffer. Die eigentlichen Details der Implementierung ist dabei viel tiefgründiger und erlauben erst die schnelle Laufzeit.

(3) Sei $j \in [1 : m]$ mit $t_j \cdots t_{j+k-1} = w \in M$ und $s_i \cdots s_{i+k-1}$ das Teilwort s' aus s ist, für den $S(s', w)$ maximal wurde; s' ist also ein Zeuge dafür, dass w in M_i aufgenommen wurde. Jetzt berechnen wir den Wert $S(s_i \cdots s_{i+k-1}, t_j \cdots t_{j+k-1})$. Ist dieser Ähnlichkeitswert größer als ϑ, so nennen wir $(s_i \cdots s_{i+k-1}, t_j \cdots t_{j+k-1})$ ein *Sequence Pair* oder *Segment Pair* (siehe auch Abbildung 6.29).

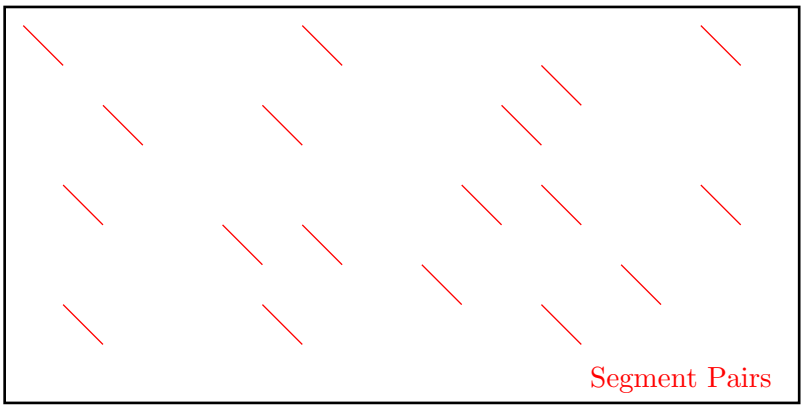


Abbildung 6.29: Skizze: Segment Pairs

(4) Solche Segment Pairs sind Startwerte für mögliche gute lokale Alignments von s und t. Zuerst wird versucht, diese entlang der Diagonalen zu so genannten *Maximal Segment Pairs* (MSP) zu erweitern (siehe auch Abbildung 6.30).

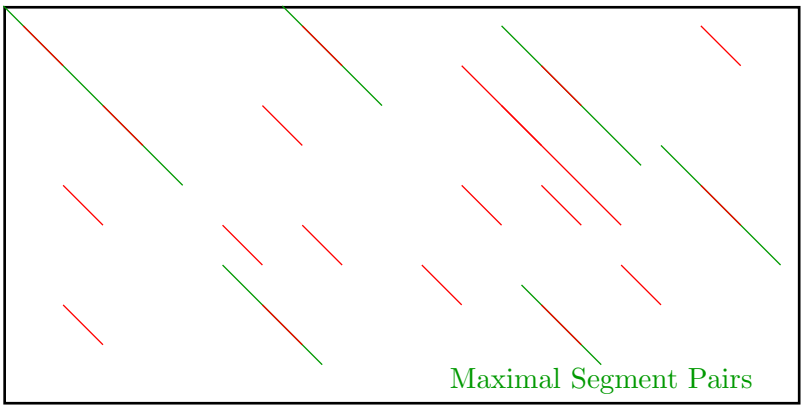


Abbildung 6.30: Skizze: Maximal Segment Pairs

Dazu wird versucht an beiden Enden das lückenlose lokale Alignment zu verlängern. Dabei wird das Erweitern abgebrochen, wenn die Verlängerung einen gewissen negativen Wert gegenüber dem ursprünglichen Score des betrachteten

Segment Pairs überschreitet. Dies ist natürlich eine Heuristik, mit der man sich auch falsche Treffer einkauft bzw. gute Treffer auslässt.

(5) Unter allen so generierten MSP's werden dann die am besten gewerteten als so genannte *High Scoring Segments Pairs* (*HSP*) ausgewählt und zusammen mit dem erzielten Score und einigen Statistiken als Ergebnis zurückgeliefert, siehe hierzu auch die Abbildung 6.31.

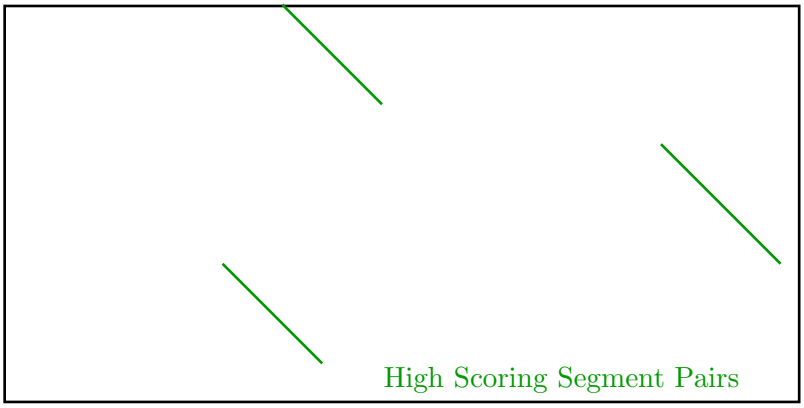


Abbildung 6.31: Skizze: High Scoring Segment Pairs

(6) In modernen Versionen werden diese HSP's weiter zu lokalen Alignments verarbeitet. Im Gegensatz zu FASTA werden hierbei keine lokalen Sequenzen-Alignments in einer näheren Umgebung bestimmt, sondern eine Position (i, j) des HSP als so genanntes *Seed Pair* ausgewählt. Ausgehend von diesem Seed Pair wird dann ein Sequenzen-Alignment nach links oben und rechts unten berechnet.

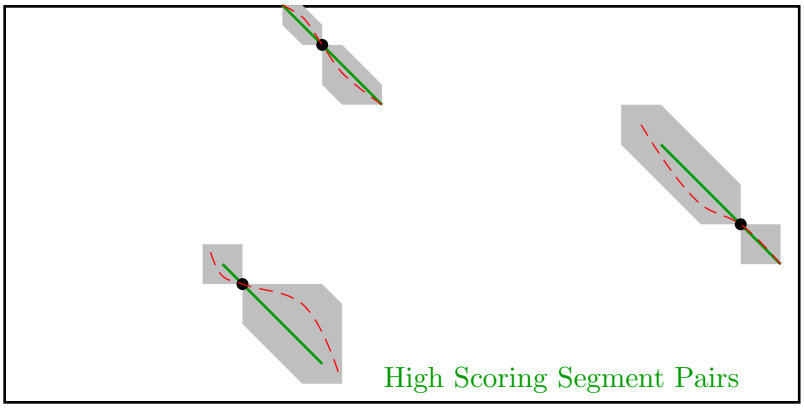


Abbildung 6.32: Skizze: Erweiterung der High Scoring Segment Pairs

Auch hier wird wieder heuristisch das Alignment abgebrochen, wenn die berechneten Scores zu klein sind. Die Wahl des geeigneten Seed Pairs beeinflusst die Güte des Ergebnisses natürlich, so dass man dieses Seed Pair sehr sorgfältig wählen muss. Dies ist in Abbildung 6.32 illustriert.

6.7.5 Der Algorithmus von Baeza-Yates und Perleberg

In diesem Abschnitt wollen wir noch einen exakten Algorithmus zur approximativen Suche mit einer erwarteten linearen Laufzeit vorstellen, den Algorithmus von R. Baeza-Yates und C. Perleberg. Die Grundidee ist, das Suchmuster in mindestens $k+1$ Teile gleicher Länge zu zerlegen. Wenn man dann nach einem approximativen Treffer mit maximal k Fehlern sucht, muss eines der Teile exakt übereinstimmen.

Sei also im Folgenden t der gegebene Text der Länge n und s das Suchmuster der Länge m. Wir nehmen an, wir wollen höchstens k Fehler gemäß der Edit-Distanz (Insertionen, Deletionen, Substitutionen) erlauben.

Lemma 6.94 *Sei $t \in \Sigma^n$ und $s \in \Sigma^m$. Sei weiter $r = \lfloor \frac{m}{k+1} \rfloor$ und sei $s = s_1 \cdots s_{k+2}$ mit $|s_i| = r$ für $i \in [1 : k+1]$. Entspricht s einem Teilwort von t mit einer Edit-Distanz von maximal k, dann existiert ein $i \in [1 : k+1]$, so dass s_i ein Teilwort von t ist*

Beweis: Beachte zuerst, dass nach Definition $|s_{k+2}| < r$ (wobei s_{k+2} auch das leere Wort sein kann) gilt. Betrachte das semi-globale Alignment von s gegen t und die zugehörige Partition von s (siehe Abbildung 6.33). Wenn jedes der ersten $k+1$

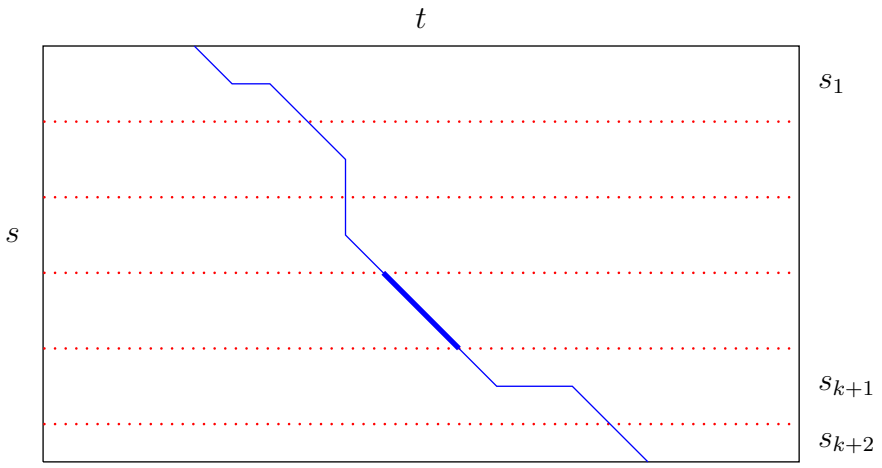


Abbildung 6.33: Skizze: Alignment mit Editdistanz $k = 4$

Teil-Alignments in den ersten $k+1$ Streifen einen Fehler enthalten würde, müssten das gesamte Alignment mindestens $k+1$ Fehler enthalten, was den gewünschten Widerspruch liefert. ■

Unser Algorithmus versucht dann die Suchwort-Menge $S = \{s_1, \ldots, s_{k+1}\}$ beispielsweise mithilfe des Aho-Corasick-Algorithmus zu suchen. Für jeden exakten Treffer können wir dann mithilfe der dynamischen Programmierung versuchen diese Treffer zu einem semi-globalen Alignment mit einer Edit-Distanz von maximal k zu erweitern.

Die Laufzeit für den Aho-Corasick-Algorithmus ist offensichtlich $O(n+m)$ Ein Erweiterung eines Treffers mit einem semi-globalen Alignment kann in Zeit $O(m^2)$ erledigt werden (oder auch in Zeit $O(km)$, wenn wir die Ähnlichkeit des zu suchenden Teilwortes mit maximal k Fehlern voraussetzen). Die Gesamtlaufzeit $O(n+\nu \cdot m^2)$ hängt nun im Wesentlichen von der Anzahl ν der Treffer ab, die wir in der ersten Phase finden.

Für eine Analyse im Erwartungswert nehmen wir an, dass jeder Buchstabe in Suchmuster bzw. im Text mit Wahrscheinlichkeit $\frac{1}{\sigma}$ auftritt, wobei $\sigma := |\Sigma|$.

Lemma 6.95 *Sei $t \in \Sigma^n$ und $s \in \Sigma^m$. Sei weiter $r = \lfloor \frac{m}{k+1} \rfloor$ und sei $s = s_1 \cdots s_{k+2}$ mit $|s_i| = r$ für $i \in [1 : k+1]$. Für $S = \{s_1, \ldots, s_{k+1}\}$ ist die erwarte Anzahl von Treffern in t gegeben durch $\frac{n(k+1)}{\sigma^r}$.*

Beweis: Die Wahrscheinlichkeit, dass zwei Zeichenreihen der Länge r übereinstimmen ist in unserem Modell $\frac{1}{\sigma^r}$. Somit kann ist die erwarte Anzahl von Vorkommen eines Suchwortes $s \in S$ für lange Texte t gerade $\frac{n}{\sigma^r}$. Für die Suchwortmenge S gibt es dann im Erwartungswert $\frac{n(k+1)}{\sigma^r}$ Treffer. ■

Somit beträgt also die erwartete Laufzeit des Algorithmus von Baeza-Yates und Perleberg $O(\frac{m^2 n(k+1)}{\sigma^r})$. Beschränken wir jetzt $k \leq \frac{\log(\sigma)m}{6\log(m)} - 1$, dann erhalten wir für r:

$$r = \left\lfloor \frac{m}{k+1} \right\rfloor \geq \left\lfloor \frac{m}{\frac{\log(\sigma)m}{6\log(m)}} \right\rfloor = \left\lfloor \frac{6\log(m)}{\log(\sigma)} \right\rfloor \geq \frac{3\log(m)}{\log(\sigma)} = 3\log_\sigma(m).$$

Die letzte Ungleichung folgt aus der Tatsache, dass für $x > 1$ gilt: $\lfloor 2x \rfloor \geq x$. Somit erhalten wir (für eine konstante Alphabetgröße σ):

$$\sigma^r \geq \sigma^{3\log_\sigma(m)} = \left(\sigma^{\log_\sigma(m)}\right)^3 = m^3.$$

Damit erhalten für die erwartete Laufzeit, wenn wir hier $k+1 \leq m$ verwenden:

$$O\left(\frac{m^2 n(k+1)}{\sigma^r}\right) \leq O\left(\frac{m^3 n}{m^3}\right) = O(n).$$

Theorem 6.96 *Alle Vorkommen eines Suchworts $s \in \Sigma^m$ können in einem Text $t \in \Sigma^n$ mit einer Edit-Distanz kleiner als $\frac{\log(\sigma)m}{6\log(m)}$ mithilfe des Algorithmus von Baeza-Yates und Perleberg mit einer linearen erwarteten Laufzeit gefunden werden.*

19.12.19

Probabilistic Modeling 7

7.1 Signifikanz von Alignment-Scores

In diesem Kapitel wollen wir uns mit der probabilistischen Modellierung von bioinformatischen Problemstellungen und statistischer Inferenz beschäftigen. Zuerst wollen wir die Aussagekraft von Alignments (beispielsweise aus BLAST) genauer untersuchen.

7.1.1 Wahrscheinlichkeitsmodell

Zunächst einmal müssen wir uns ein Wahrscheinlichkeitsmodell überlegen, in dem wir die Signifikanz von lokalen Alignments bewerten wollen. Wir werden hier nur lokale Alignments ohne Lücken betrachten, da die Analyse hierfür schon sehr aufwendig ist.

Im Folgenden sei Σ das von uns betrachte Alphabet und seien x bzw. y zwei Sequenzen der Länge n bzw. m über Σ. Dabei sei x die (Menge von) Sequenz(en), nach denen wir suchen, und y stellt die Datenbank dar. Für ein gefundenes lokales Alignment zwischen x und y wollen wir im Wesentlichen wissen, wie groß die Wahrscheinlichkeit ist, dass dieses Alignment auch zufällig in zwei Sequenzen der gleichen Länge vorkommen kann. Je kleiner diese Wahrscheinlichkeit ist, desto signifikanter ist das Alignment.

Dazu müssen wir zuerst die Wahrscheinlichkeitsverteilungen der Zeichen aus Σ in den Sequenzen x bzw. y festlegen. Sei dazu p_a bzw. p'_a für $a \in \Sigma$ die Wahrscheinlichkeit, dass an einer Position das Zeichen a in x bzw. y vorkommt. Diese Wahrscheinlichkeiten seien unabhängig von der Position in x bzw. y. Wir betrachten hier zwei verschiedene Verteilungen, wobei sich die für y in der Regel aus der in der Datenbank vorkommenden Verteilung der Buchstaben oder einer generellen Analyse ergibt. Für x kann natürlich auch dieselbe Verteilung wie für y angenommen werden, oder eine die sich aus der Sequenz (bzw. den Sequenzen) aus x ergibt. Wir nehmen hierbei an, dass die Verteilung unabhängig von der Position und der Nachbarschaft ist, also eine unabhängige identische Verteilung.

Mit $\mathcal{S} = \{(x', y') : x' \in \Sigma^n \wedge y' \in \Sigma^m\}$ bezeichnen wir die Menge aller Sequenz-Paare, die die gleiche Länge wie unser untersuchtes Anfragepaar $(x, y) \in \Sigma^n \times \Sigma^m$

haben. Für ein Sequenzen-Paar (x, y) ergibt sich dann als Wahrscheinlichkeit in $\mathcal{S}$ der Wert:

$$\mathrm{Ws}_{\mathcal{S}}[\{(x,y)\}] = \prod_{i=1}^{n} p_{x_i} \cdot \prod_{j=1}^{m} p'_{y_j}.$$

Bezeichnen wir wieder mit $s(\cdot,\cdot)$ das Ähnlichkeitsmaß für lokale Alignments von zwei Sequenzen, so sind wir an folgender Wahrscheinlichkeit interessiert:

$$\mathrm{Ws}_{\mathcal{S}}[s(x',y') \geq \alpha],$$

wobei $\alpha = s(x,y)$ den Wert eines gefundenen (optimalen) lokalen Alignments zwischen x und y angibt und $s(x',y')$ als Zufallsvariable über dem Ereignisraum $\mathcal{S}$ interpretiert wird. Dieser Wert wird allgemein auch als *P-Wert* oder *P-Value* bezeichnet. Stattdessen betrachten man oft auch den so genannten *E-Wert* oder *E-Value*. Dieser ist definiert als die erwartete Anzahl von Paaren mit einem höheren Score:

$$\mathbb{E}_{\mathcal{S}}|\{(x',y') \,:\, s(x',y') \geq \alpha\}| = \sum_{\substack{(x',y')\in S \\ s(x',y')\geq\alpha}} \mathrm{Ws}[(x',y')].$$

7.1.2 Random Walks

Um den E-Value für ein gegebenes HSP bestimmen zu können, benötigen wir noch das Konzept eines Random Walks.

Definition 7.1 *Sei $M = [-c : d]$ für $c, d \in \mathbb{N}$ und sei p_i eine Wahrscheinlichkeitsverteilung auf M mit $p_{-c} > 0$ und $p_d > 0$. Ein* Random Walk *(auch* Irrfahrt*) auf $\mathbb{Z}$ mittels M ist eine unendliche Folge von ganzzahligen Zufallsvariablen $(X_i)_{i\in\mathbb{N}_0}$ mit*

$$\mathrm{Ws}[X_i = x] = \sum_{j=-c}^{d} p_j \cdot \mathrm{Ws}[X_{i-1} = x - j].$$

Man beachte hierbei, dass über die Wahrscheinlichkeitsverteilung von X_0 nichts ausgesagt wird.

Im Folgenden nehmen wir die folgenden Bedingungen an:

(R1) $\sum_{j\in M} p_j \cdot j < 0$,

(R2) $\mathrm{ggT}\,\{j \in [1:d] \,:\, p_j > 0\} = 1$,

Wir wollen nun unsere High Scoring Segment Pairs (HSP) als Random Walks simulieren. Dazu definieren wir für eine Scoring-Matrix w die Menge M und die dazugehörige Wahrscheinlichkeitsverteilung. Wir setzen $c := -\min\{w(a,b) : a,b \in \Sigma\}$ und $d := \max\{w(a,b) : a,b \in \Sigma\}$ für $M = [-c:d]$ (hierbei nehmen wir an, dass w nur ganzzahlige Werte annimmt). Die Wahrscheinlichkeitsverteilung wird durch

$$p_i := \sum_{\substack{a,b\in\Sigma \\ w(a,b)=i}} p_a \cdot p'_b$$

für $i \in [-c:d]$ definiert. Hierbei ist $p_a \cdot p'_b$ die Wahrscheinlichkeit, dass das Buchstabenpaar (a,b) in einem Alignment auftritt, in dem von uns verwendeten zufälligen Nullmodell. Damit übersetzen sich die Bedingungen R1 und R2 zu:

(S1) $\sum_{(a,b)\in\Sigma^2} p_a \cdot p'_b \cdot w(a,b) < 0$,

(S2) $\text{ggT}\{w(a,b) : w(a,b) > 0 \wedge p_a \cdot p'_b > 0\} = 1$.

Die Bedingung S1 ist für eine sinnvolle Scoring-Matrix erfüllt. Andernfalls würde für ein lokales Alignment die Verlängerung um ein zufälliges Alignment den Score im Erwartungswert nur erhöhen. Dies ist für eine Scoring-Matrix zur Verwendung für lokale Alignments nicht besonders sinnvoll. Die Bedingung S2 ist von beweistechnischer Natur, deren Sinn wir im Rahmen dieses Skripts nicht näher erläutern können. Wenn S2 nicht gilt, kann man alle Werte der Scoring-Matrix mit einer 10er-Potenz multiplizieren und auf einen der positiven Werte den Wert 1 aufaddieren, dann ist die Bedingung S2 erfüllt und die Scoring-Matrix unterscheidet sich nur marginal. Als nächstes definieren wir die Begriffe von Leiterpunkten und Exkursionen.

Definition 7.2 *Sei $(X_i)_{i\in\mathbb{N}_0}$ mit $X_0 = 0$ ein Random Walk auf $\mathbb{Z}$ durch die Wahrscheinlichkeitsverteilung p_i auf $M = [-c:d]$. Ein Zeitpunkt t heißt* Leiterpunkt *des Random Walks, wenn $X_t < \min\{X_i : i \in [0:t-1]\}$.*

Sind X_t und $X_{t'}$ zwei aufeinander folgende Leiterpunkte des Random Walks, so wird der Abschnitt $[t:t'-1]$ auch Exkursion *genannt.*

Wir betrachten im Folgenden ein Alignment als einen Random Walk, der durch die zugehörige Scoring-Matrix induziert wird. Der maximale Wert eines Präfixes einer Exkursion entspricht dann dem Score eines HSP des zugehörigen gaplosen lokalen Alignments. Dies wird leichter einsichtig, wenn wir einen korrespondierenden Random Walk $(Y_i)_{i\in\mathbb{N}_0}$ zu $(X_i)_{i\in\mathbb{N}_0}$ wie folgt definieren:

$$Y_i := \max\{0, Y_{i-1} + (X_i - X_{i-1})\}.$$

Anschaulich bedeutet dies, dass der Random Walk auf $\mathbb{N}_0$ und nicht auf $\mathbb{Z}$ verläuft, da wir das Absinken unter 0 explizit verbieten. Dies entspricht den gaplosen lokalen

Alignments zweier Sequenzen, in denen ja negative Scores ebenfalls explizit verboten wird. Dies ist auch in Abbildung 7.1 illustriert, wobei Leiterpunkte in rot markiert sind.

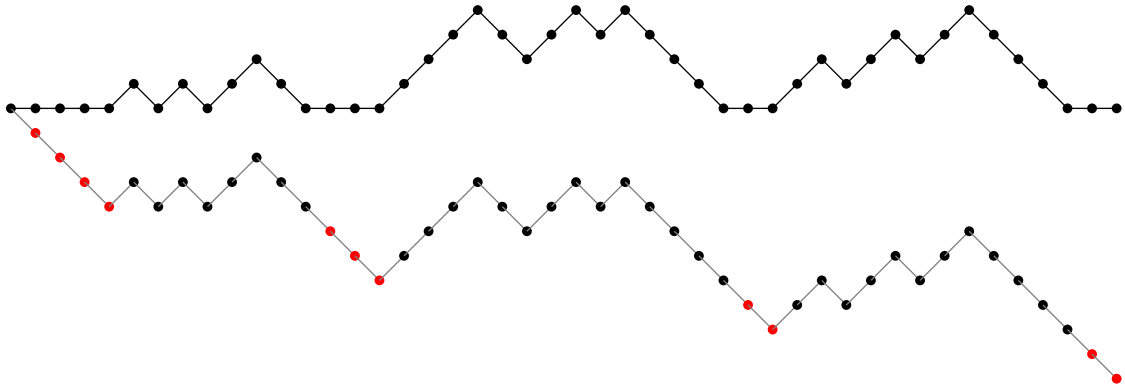


Abbildung 7.1: Skizze: Random Walk mit negativer Drift ($p_{+1} = \frac{3}{7}$, $p_0 = 0$, $p_{-1} = \frac{4}{7}$)

7.1.3 Ein einfaches Modell

Der Einfachheit halber wollen wir im Folgenden eine vereinfachte Scoring-Matrix betrachten, die den wesentlichen Gedankengang zur Signifikanz von HSP's verdeutlichen soll. Wir betrachten als Scoring-Matrix eine Kostenfunktion der Gestalt $w(a, b) := 2 \cdot \delta_{a,b} - 1$ für $a, b \in \Sigma$. Das heißt ein Match wird mit $+1$ und ein Mismatch wird mit -1 bewertet. Weiterhin nehmen wir an, dass $p_a = p'_a$ für alle $a \in \Sigma$.

Was bedeutet dies für den zugehörigen Random Walk? In jedem Schritt geht es um eine Einheit nach oben oder nach unten (siehe auch Abbildung 7.1). Mit Wahrscheinlichkeit

$$p := \sum_{a \in \Sigma} p_a^2$$

geht es dabei nach oben und mit

$$\sum_{a \in \Sigma} \sum_{a \neq b \in \Sigma} p_a p_b = \sum_{a \in \Sigma} p_a (1 - p_a) = 1 - \sum_{a \in \Sigma} p_a^2 = 1 - p$$

nach unten. Damit die Bedingung S1 erfüllt ist, muss $p - (1 - p) < 0$, also $p < 0.5$ gelten.

Um nun die Wahrscheinlichkeiten für den besten Score eines HSP's in zwei zufälligen Sequenzen zu ermitteln, definieren wir die folgende Wahrscheinlichkeit w_h.

Notation 7.3 *Für $a \leq h \leq b \in \mathbb{Z}$ bezeichne w_h die Wahrscheinlichkeit, dass ein bei h startender Random Walk den Wert b vor dem Wert a erreicht.*

Es gilt dann:

$$\begin{aligned} w_a &= 0, \\ w_b &= 1, \\ w_h &= p \cdot w_{h+1} + (1-p) \cdot w_{h-1} \quad \text{für } h \in (a:b). \end{aligned}$$

Nach dem Lösen von homogenen linearen Rekursionsgleichungen wissen wir, dass es sich bei der Lösung um eine Linearkombination von Potenzen der Wurzeln des zugehörigen charakteristischen Polynoms handelt. Somit ist $w_h = k \cdot y^h$ mit noch näher zu bestimmenden Konstanten $y, k \in \mathbb{C}$ eine Lösung der Rekursionsgleichung. Zur Ermittlung von y setzen wir diesen Ansatz in die Rekursionsgleichung ein:

$$k \cdot y^h = p \cdot k \cdot y^{h+1} + (1-p)k \cdot y^{h-1}.$$

Nach Division durch $k \cdot y^{h-1}$ erhalten wir:

$$y = p \cdot y^2 + (1-p).$$

Damit erhalten wir folgende quadratische Gleichung:

$$y^2 - \frac{1}{p}y + \frac{1-p}{p} = 0.$$

Die Lösungen ergeben sich zu

$$\begin{aligned} y_{1,2} &= \frac{1}{2p} \pm \sqrt{\frac{1}{4p^2} - \frac{1-p}{p}} \\ &= \frac{1 \pm \sqrt{4p^2 - 4p + 1}}{2p} \\ &= \frac{1 \pm \sqrt{4(p^2 - p + \frac{1}{4})}}{2p} \\ &= \frac{1 \pm \sqrt{4(p - \frac{1}{2})^2}}{2p} \\ &= \frac{1 \pm 2(p - \frac{1}{2})}{2p}. \end{aligned}$$

Damit ist $y_1 = 1$ und $y_2 = \frac{2-2p}{2p} = \frac{1-p}{p}$. Damit wissen wir also, dass die Lösung von w_h wie folgt aussieht:

$$w_h = C_1 \cdot \left(\frac{1-p}{p}\right)^h + C_2.$$

Berücksichtigen wir die Randbedingungen $w_a = 0$ und $w_b = 1$, dann gilt:

$$\begin{aligned} 0 &= C_1 \left(\frac{1-p}{p}\right)^a + C_2, \\ 1 &= C_1 \left(\frac{1-p}{p}\right)^b + C_2. \end{aligned}$$

Daraus folgt

$$1 = C_1 \left[\left(\frac{1-p}{p}\right)^b - \left(\frac{1-p}{p}\right)^a\right].$$

Also gilt (mit $C_2 = -C_1(\frac{1-p}{p})^a$):

$$\begin{aligned} C_1 &= \left[\left(\frac{1-p}{p}\right)^b - \left(\frac{1-p}{p}\right)^a\right]^{-1}, \\ C_2 &= -\left(\frac{1-p}{p}\right)^a \cdot \left[\left(\frac{1-p}{p}\right)^b - \left(\frac{1-p}{p}\right)^a\right]^{-1}. \end{aligned}$$

Somit erhalten wir für w_h mit $h \in [a, b]$:

$$w_h = C_1 \left(\frac{1-p}{p}\right)^h + C_2 = \frac{\left(\frac{1-p}{p}\right)^h - \left(\frac{1-p}{p}\right)^a}{\left(\frac{1-p}{p}\right)^b - \left(\frac{1-p}{p}\right)^a}.$$

Der für uns interessierende Fall ist $a = -1$ für einen großen Score b mit $h = 0$ (Beginn eines HSP):

$$w_0 = \frac{1 - \left(\frac{1-p}{p}\right)^{-1}}{\left(\frac{1-p}{p}\right)^b - \left(\frac{1-p}{p}\right)^{-1}} \sim \left(1 - \frac{p}{1-p}\right) \cdot \left(\frac{p}{1-p}\right)^b.$$

Hierbei gilt für zwei Funktionen $f, g : \mathbb{N} \to \mathbb{R}_+$ die folgende Definition:

$$f(n) \sim g(n) \quad :\Leftrightarrow \quad f(n) = g(n) \cdot (1 + o(1)).$$

Beachte, dass aus $p < 0.5$ die Beziehung $\frac{1-p}{p} > 1$ (und $\frac{p}{1-p} < 1$) folgt.

Setzen wir nun $\lambda := \log\left(\frac{1-p}{p}\right) > 0$, dann gilt:

$$w_0 \sim (1 - e^{-\lambda}) \cdot e^{-\lambda b}.$$

Wir bezeichnen mit W_b die Wahrscheinlichkeit, dass wir den Score b von einem Leiterpunkt aus erreichen ohne vorher negative Werte erreicht zu haben. Um W_b aus w_0 bestimmen zu können, müssen wir w_0 nur noch berücksichtigen, dass wir von einem Leiterpunkt starten. Man kann zeigen, dass dann gilt:

$$W_b \sim k \cdot e^{-\lambda b}.$$

Hierbei hängt die Konstante k natürlich auch von λ ab.

Auch im allgemeinen Fall konnte gezeigt werden, dass für die Wahrscheinlichkeit W_b, ab einer festen Position einen HSP mit Score mindestens b zu finden, für viele Scoring-Matrizen gilt:

$$W_b \sim k \cdot e^{-\lambda b} \qquad \text{bzw.} \qquad W_b = \Theta(k \cdot e^{-\lambda b}),$$

wobei wir hierfür auf die einschlägige Literatur verweisen müssen.

Durch eine Zerlegung des HSP's nach dem ersten Zeichenpaar gilt:

$$\begin{aligned} k \cdot e^{-\lambda B} &\sim W_B \\ &\approx \sum_{a,b \in \Sigma} p_a \cdot p_b \cdot W_{B-w(a,b)} \\ &\sim \sum_{a,b \in \Sigma} p_a \cdot p_b \cdot k \cdot e^{-\lambda(B-w(a,b))}. \end{aligned}$$

Dies ist äquivalent zu

$$1 \approx \sum_{a,b \in \Sigma} p_a \cdot p_b \cdot e^{\lambda \cdot w(a,b)},$$

woraus sich λ numerisch bestimmen lässt. Wir weisen hier explizit darauf hin, dass bei dieser Analyse diese Vorgehensweise mathematisch nicht ganz sauber ist, aber man diese Argumentation auf mathematisch solide Beine stellen kann. Auch die Konstante k kann numerisch ermittelt werden, worauf wir hier allerdings auch nicht näher eingehen wollen und auch hier auf die entsprechende Literatur verweisen müssen.

Da es etwa $n \cdot m$ Positionspaare gibt, an denen ein HSP beginnen kann, gilt für den E-Value $E_{S \geq b}$:

$$E_{S \geq b} \sim n \cdot m \cdot k \cdot e^{-\lambda \cdot b}.$$

Somit zeigen kleine E-Values eine höhere Signifikanz des gefundenen lokalen Alignments ohne Lücken an. Ein Fund wird dabei als signifikant erachtet, wenn der E-Value deutlich kleiner als 10^{-3} ist.

Für lokale Alignments mit Indels gelten ähnliche Abschätzungen für den E-Value. Dies konnte bislang noch nicht vollständig mathematisch sauber bewiesen werden, aber die ersten Resultate und empirische Studien weisen deutlich darauf hin.

7.1.4 Normalisierte BLAST Scores (Bit-Scores)

Der Score selbst ist als Maß zum Vergleichen von Alignments eher ungeeignet, insbesondere wenn auch noch verschiedenen Scoring-Matrizen verwendet werden. Allerdings bietet die Verwendung eines normalisierten Scores (auch *Bit-Score* genannt) eine Alternative. Sei S der Score, der von BLAST geliefert wird, dann ist der Bit-Score S' definiert durch

$$S' := \frac{\lambda \cdot S - \ln(k)}{\ln(2)} \qquad \text{bzw.} \qquad S = \frac{S' \cdot \ln(2) + \ln(k)}{\lambda}.$$

Dies wird durch folgende Rechnung gerechtfertigt:

$$\begin{aligned} E_{S' \geq b} &= E_{S \geq \frac{b \cdot \ln(2) + \ln(k)}{\lambda}} \\ &\sim nmk \cdot e^{-\lambda \cdot \frac{b \cdot \ln(2) + \ln(k)}{\lambda}} \\ &= nmk \cdot e^{-b \cdot \ln(2)} \cdot e^{-\ln(k)} \\ &= nm \cdot 2^{-b}. \end{aligned}$$

Da diese Bit-Scores nun nicht mehr unmittelbar von λ und k abhängen, kann man die Bit-Scores nun für verschiedene Scoring-Matrizen besser miteinander vergleichen.

07.01.20

7.1.5 P-Value

Zum Abschluss wollen wir noch eine Asymptotik für den zu Beginn des Abschnitts gesuchten P-Value $\text{Ws}_S[s(x', y') \geq b]$ angeben. Problematisch sind hier die vielen auftretenden stochastischen Abhängigkeiten. Für die Berechnung des P-Wertes sei $I_{i,j}$ eine Indikatorvariable des Ereignisses, dass am Positionspaar (i, j) ein HSP mit Score mindestens b auftritt. Somit ist der P-Wert dann durch $\text{Ws}[\sum_{i,j} I_{i,j} \geq 1]$ gegeben. Da beispielsweise nach Definition nicht gleichzeitig $I_{i,j}$ und $I_{i+1,j+1}$ jeweils 1 sein kann, müssten hier eine Menge von stochastischen Abhängigkeiten berücksichtigt werden. Es konnte jedoch gezeigt werden, dass die stochastischen Abhängigkeiten insgesamt sehr gering sind und hier vernachlässigt werden können.

Wir können daher als Verteilung eine Binomialverteilung zum Parameter $(nm, ke^{-\lambda b})$ annehmen, ohne dass wir dies hier beweisen wollen. Der Parameter $(nm, ke^{-\lambda b})$ erklärt sich aus der Anzahl nm möglicher Anfangsposition, an denen ein HSP beginnen kann. Dabei wollen wir Effekte, die an den Enden der Sequenzen (für Werte von i bzw. j nahe an n bzw, m) auftreten, hier ebenfalls vernachlässigen. Die Wahrscheinlichkeit, dass an einer Position ein HSP mit großem Score beginnt ist $k \cdot e^{-\lambda b}$ (wie im Unterabschnitt 7.1.3 gezeigt). Dann gilt also

$$\text{Ws}\left[\sum_{i,j} I_{i,j} = \ell\right] \approx \binom{nm}{\ell} \cdot \left(ke^{-\lambda b}\right)^{\ell} \cdot \left(1 - ke^{-\lambda b}\right)^{nm-\ell}.$$

Da die Wahrscheinlichkeiten des positiven Ausgangs sehr klein zur Anzahl der Stichproben ist, konvergiert die Binomialverteilung zum Parameter $(nm, ke^{-\lambda b})$ gegen die Poisson-Verteilung zum Parameter $\mu := nmke^{-\lambda b}$. Damit gilt

$$\text{Ws}\left[\sum_{i,j} I_{i,j} = \ell\right] \approx \frac{\mu^\ell}{\ell!} e^{-\mu}.$$

Daraus folgt für den P-Value:

$$\text{Ws}\left[\sum_{i,j} I_{i,j} \geq 1\right] = 1 - \text{Ws}\left[\sum_{i,j} I_{i,j} = 0\right] \approx 1 - \frac{\mu^0}{0!} e^{-\mu} = 1 - e^{-nmke^{-\lambda b}}.$$

Auch für lokale Alignments mit Lücken konnte gezeigt werden, dass es Konstanten λ und k gibt, so dass die oben gezeigten Beziehungen gelten. Mathematisch konnte dies allerdings noch nicht vollständig bewiesen werden.

Bei wiederholten Datenbanksuchen muss man allerdings bei der Interpretation der P-Values vorsichtig sein. Für ein übliches Signifikanz-Niveau von 0,01 bekommt man bereits bei einer Datenbank mit 100 Einträgen einen Treffer mit diesen Wert rein zufällig. Berechnen wir die Wahrscheinlichkeit für einen vermeintlich signifikanten Treffern (P-Value kleiner als 0,01) bei einer Datenbank mit 60 Einträgen. Sei dazu I_j die Indikator-variable, dass der j-te Datenbankeintrag einen vermeintlich signifikanten Treffer (P-Value kleiner als 0,01) liefert. Dann gilt (unter der Annahme, dass die Einträge der Datenbank quasi stochastisch unabhängig sind):

$$\begin{aligned}
\text{Ws}\left[\sum_{j=1}^{60} I_j \geq 1\right] &= 1 - \text{Ws}\left[\sum_{j=1}^{60} I_j = 0\right] \\
&= 1 - \left(\text{Ws}\left[I_j = 0\right]\right)^{60} \\
&= 1 - \left(1 - \text{Ws}\left[I_j = 1\right]\right)^{60} \\
&= 1 - (1 - 0{,}01)^{60} \\
&\approx \frac{1}{2}.
\end{aligned}$$

Also mit Wahrscheinlichkeit 1/2 hat man hier bereits vermeintlich signifikante Treffer, die allerdings rein zufällig sind. Dieses Phänomen nennt man *Multiple Testing*. Die P-Values müssen in Abhängigkeit der Anfrage-Zahlen bewertet werden. Dazu kann man entweder Multiple-Testing-Korrekturen verwenden, oder man betrachtet den E-Value, der die Aussage gleich unter Berücksichtigung der Datenbankgröße angibt.

Aus dem tägliche Leben kennt man das vom Lotto. Die Wahrscheinlichkeit (also der P-Value) für sechs Richtige plus Superzahl ist etwa 1 zu $1.4 * 10^8$. Hierbei sind die

Tipps der Teilnehmer die Datenbank und die Ziehung die entsprechende Anfrage Dennoch gewinnt etwa bei jeder vierten Ausspielung ein Teilnehmer. Man müsste bei einem signifikanten Treffer diesem unterstellen, dass er hellsehen kann oder weiß wie man beim Lotto betrügen kann.

7.2 Konstruktion von Ähnlichkeitsmaßen

In diesem Abschnitt wollen wir einen kurzen Einblick geben, wie man aus experimentellen biologischen Daten gute Kostenfunktionen für Ähnlichkeitsmaße konstruieren kann.

7.2.1 Allgemeiner Ansatz

Für das Problem des Sequenzen-Alignments kann man sich zwei simple Modelle vorstellen. Das erste ist das so genannte *Zufallsmodell R*. Hier nehmen wir an dass zwei ausgerichtete Sequenzen gar nichts miteinander zu tun haben und gewisse Übereinstimmungen rein zufällig sind.

Für die Wahrscheinlichkeit für das Auftreten eines (lückenlosen) Alignments (s,t) für $s,t \in \Sigma^n$ gilt dann in diesem Modell R:

$$\mathrm{Ws}[(s,t) \mid R] = \prod_{i=1}^{n} p_{s_i} \prod_{i=1}^{n} p_{t_i}.$$

Hierbei ist p_a die Wahrscheinlichkeit, dass in einer Sequenz das Zeichen $a \in \Sigma$ auftritt. Wir haben für diese Alignments jedoch angenommen, dass Leerzeichen nicht erlaubt sind (also keine Deletionen und Insertionen, sondern nur Substitutionen).

Ein anderes Modell ist das so genannte *Mutationsmodell* M, wobei wir annehmen, dass ein Alignment (s,t) für $s,t \in \Sigma^n$ durchaus biologisch erklärbar ist, nämlich beispielsweise mit Hilfe von Mutationen. Hier gilt für die Wahrscheinlichkeit für ein Alignment (s,t)

$$\mathrm{Ws}[(s,t) \mid M] = \prod_{i=1}^{n} q_{s_i,t_i}.$$

Hierbei bezeichnet $q_{a,b}$ die Wahrscheinlichkeit, dass in einem Sequenz-Paar a mit b aligniert wird. Wir nehmen an, dass $q_{a,b} = q_{b,a}$ und $\sum_{a,b\in\Sigma} q_{a,b} = 1$ gilt.

Vergleichen wir jetzt beide Modelle, d.h. wir dividieren die Wahrscheinlichkeiten für ein gegebenes Alignment (s,t) mit $s,t \in \Sigma^n$:

$$\frac{\mathrm{Ws}[(s,t) \mid M]}{\mathrm{Ws}[(s,t) \mid R]} = \prod_{i=1}^{n} \frac{q_{s_i,t_i}}{p_{s_i} \cdot p_{t_i}} \lesseqgtr 1.$$

Ist nun dieser Bruch größer als 1, so spricht diese für das Mutationsmodell, andernfalls beschreibt das Zufallsmodell dieses Alignment besser.

Leider wäre dieses Maß multiplikativ und nicht additiv. Da können wir uns jedoch sehr einfach mit einem arithmetischen Trick behelfen. Wir logarithmieren die Werte:

$$\text{Score}(s,t) := \sum_{i=1}^{n} \underbrace{\log\left(\frac{q_{s_i,t_i}}{p_{s_i} \cdot p_{t_i}}\right)}_{\text{Kostenfunktion}}$$

Aus diesem Ähnlichkeitsmaß können wir nun eine zugehörige Kostenfunktion für alle Paare $(a,b) \in \Sigma \times \Sigma$ sehr leicht ableiten, nämlich den logarithmierten Quotienten der einzelnen Wahrscheinlichkeiten, dass sich ein solches Paar innerhalb eines Alignments gegenübersteht:

$$w(a,b) := \log\left(\frac{q_{a,b}}{p_a \cdot p_b}\right).$$

Es bleibt die Frage, wie man p_a bzw. $q_{a,b}$ für $a,b \in \Sigma$ erhält? Damit werden wir uns in den folgenden Abschnitten beschäftigen.

7.2.2 PAM-Matrizen

In diesem Abschnitt wollen für die obige Frage eine Lösung angeben. Wir nehmen hierzu an, wir erhalten eine Liste von so genannten *akzeptierten Mutationen* $L = (\{a_1,b_1\},\ldots,\{a_n,b_n\})$, d.h. wir können relativ sicher sein, dass die hier vorgekommenen Mutationen wirklich passiert sind. Solche Listen kann man über mehrfache Sequenzen-Alignments von gut konservierten Regionen ähnlicher Spezies oder aus guten evolutionären Bäumen über den betrachteten Sequenzen erhalten. Mit $n_{a,b}$ bezeichnen wir die Paare $\{a,b\}$ in der Liste L und mit n die Anzahl aller Paare in L. Wir nehmen hier an, dass $n_{a,b} = n_{b,a} > 0$ und $n_{a,a} = 0$ für $a \neq b \in \Sigma$. Damit folgt $\sum_{a,b\in\Sigma} n_{a,b} = 2n$.

Für p_a mit $a \in \Sigma$ ist es am einfachsten, wenn man hierfür die relative Häufigkeit von a in allen Sequenzen annimmt:

$$p_a = \frac{1}{2n} \sum_{b\in\Sigma} n_{a,b}.$$

Man kann stattdessen auch die relative Häufigkeit aus anderen geeigneten Daten als Schätzer verwenden. Es bleibt insbesondere $q_{a,b}$ zu bestimmen. Hierfür definieren wir zuerst die Wahrscheinlichkeit $p_{a,b}$, dass eine gegebene Aminosäure a zu b mutiert. Für die Wahrscheinlichkeit, dass dann in einem solchen Alignment a mit b aligniert wird, ist dann $q_{a,b} = p_a \cdot p_{a,b}$.

Die Mutation $a \to b$ ist also nichts anderes, als die bedingte Wahrscheinlichkeit, dass in einer Sequenz ein b auftritt, wo vor der Mutation ein a stand. Für diese bedingte Wahrscheinlichkeit schreiben wir $\text{Ws}[b \mid a]$. Nach Definition der bedingten Wahrscheinlichkeiten gilt, dass $\text{Ws}[b \mid a] = \frac{\text{Ws}[a,b]}{\text{Ws}[a]}$, wobei $\text{Ws}[a,b]$ die Wahrscheinlichkeit ist, dass einem Alignment a und b gegenüberstehen. Also gilt:

$$p_{a,b} = \text{Ws}[b \mid a] = \frac{\text{Ws}[a,b]}{\text{Ws}[a]} \propto \frac{\frac{n_{a,b}}{2n}}{p_a} = \frac{n_{a,b}}{2n} \cdot \frac{1}{p_a}.$$

Die Proportionalität folgt daher, dass wir für die Wahrscheinlichkeit $\text{Ws}[a,b]$ annehmen, dass diese durch die relative Häufigkeit von Mutationen ziemlich gut angenähert wird (sofern a in $b \neq a$ mutiert). Da in unserer Liste L nur Mutationen stehen, wissen wir natürlich nicht, mit welcher Wahrscheinlichkeit eine Mutation wirklich auftritt. Daher setzen wir zunächst etwas willkürlich für $a \neq b$ an:

$$\begin{aligned} p_{a,b} &:= \frac{n_{a,b}}{2n} \cdot \frac{1}{p_a} \cdot \frac{1}{100}, \\ p_{a,a} &:= 1 - \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b}. \end{aligned}$$

Zunächst gilt für alle $a \in \Sigma$:

$$\sum_{b \in \Sigma} p_{a,b} = p_{a,a} + \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b} = 1 - \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b} + \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b} = 1.$$

Da außerdem nach Definition $p_{a,b} \in [0,1]$ gilt, handelt es sich um eine zulässige Wahrscheinlichkeitsverteilung.

Weiter gilt

$$\begin{aligned} \sum_{a \in \Sigma} p_a \cdot p_{a,a} &= \sum_{a \in \Sigma} p_a \left(1 - \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b} \right) \\ &= \underbrace{\sum_{a \in \Sigma} p_a}_{=1} - \sum_{a \in \Sigma} \sum_{\substack{b \in \Sigma \\ b \neq a}} p_a \cdot p_{a,b} \\ &= 1 - \sum_{a \in \Sigma} \sum_{\substack{b \in \Sigma \\ b \neq a}} \frac{n_{a,b}}{2n} \cdot \frac{1}{p_a} \cdot \frac{1}{100} \cdot p_a \\ &= 1 - \frac{1}{200n} \cdot \underbrace{\sum_{a \in \Sigma} \sum_{\substack{b \in \Sigma \\ b \neq a}} n_{a,b}}_{=2n} \\ &= 0{,}99. \end{aligned}$$

Somit gilt, dass mit Wahrscheinlichkeit 99% keine Mutation auftritt und mit Wahrscheinlichkeit 1% eine Mutation auftritt. Aus diesem Grund werden diese Matrizen $(p_{a,b})_{a,b\in\Sigma}$ auch *1-PAM* genannt. Hierbei steht PAM für *Percent Accepted Mutations* oder *Point Accepted Mutations*. Als zugehörige *Kostenfunktion* erhalten wir dann für $a \neq b \in \Sigma$

$$\begin{aligned} w(a,b) &= \log\left(\frac{q_{a,b}}{p_a \cdot p_b}\right) \\ &= \log\left(\frac{p_a \cdot p_{a,b}}{p_a \cdot p_b}\right) \\ &= \log\left(\frac{p_a \cdot \frac{1}{p_a} \cdot \frac{n_{a,b}}{2n} \cdot \frac{1}{100}}{p_a \cdot p_b}\right) \\ &= \log\left(\frac{n_{a,b}}{200 \cdot n \cdot p_a \cdot p_b}\right) \end{aligned}$$

sowie für $a \in \Sigma$

$$\begin{aligned} w(a,a) &= \log\left(\frac{q_{a,b}}{p_a \cdot p_b}\right) \\ &= \log\left(\frac{p_a \cdot p_{a,a}}{p_a \cdot p_a}\right) \\ &= \log\left(\frac{1 - \sum_{\substack{b\in\Sigma \\ b\neq a}} p_{a,b}}{p_a}\right) \\ &= \log\left(\frac{1 - \sum_{b\in\Sigma} \frac{n_{a,b}}{200n \cdot p_a}}{p_a}\right) \\ &= \log\left(\frac{200 \cdot n \cdot p_a - \sum_{b\in\Sigma} n_{a,b}}{200 \cdot n \cdot p_a^2}\right). \end{aligned}$$

Diese PAM-Matrizen wurden erfolgreich für kleine evolutionäre Abstände von Margaret Dayhoff auf der Basis von Aminosäuren entwickelt und eingesetzt.

Diese 1-PAM Matrizen sind jetzt jedoch nur für sehr kurze evolutionäre Abstände geeignet. Man kann diese jedoch auch auf so genannte k-PAM-Matrizen hochskalieren, indem man die stochatsische Matrix $P = (a_{a,b})_{a,b\in\Sigma}$ durch $P^k = (a^{(k)}_{a,b})_{a,b\in\Sigma}$ ersetzt und dann entsprechend in die Kostenfunktion einsetzt. Diese erhält man durch Multiplikation der Matrix der Mutationswahrscheinlichkeiten.

Diese Methode liefert zum Beispiel so genannte 120- oder 250-PAM-Matrizen, die dann für größere evolutionäre Abstände einsetzbar sind. Für wirklich große evolutionäre Abstände haben sich jedoch PAM-Matrizen als nicht so brauchbar erwiesen. Hier werden dann meist so genannte BLOSUM-Matrizen eingesetzt, auf die wir im nächsten Abschnitt eingehen wollen.

7.2.3 BLOSUM-Matrizen

Ein anderer Ansatz zur Erzeugung von Scoring-Matrizen über Aminosäuren wurde von Steven Henikoff und Jorja Henikoff vorgeschlagen, die so genannten *BLOSUM-Matrizen* (für BLOcks SUbstition Matrices). Dieser sollte insbesondere die Schwachstelle von PAM-Matrizen umgehen, die für fern verwandte Sequenzen keine gute Ergebnisse lieferte.

Die Scoring-Matrizen werden hierbei von einer Menge von Blöcken von Sequenzen gebildet, wobei Sequenzen innerhalb der Blöcke sehr ähnlich sind und insbesondere alle die gleiche Länge besitzen. Sei also

$$\mathcal{S} = \left\{ s_j^{(\beta)} \in \Sigma^+ \ : \ \beta \in [1:B] \wedge j \in [1:n_\beta] \right\}.$$

Dabei gilt für alle $\beta \in [1:B]$ und alle $i, j \in [1:n_\beta]$, dass $|s_i^{(\beta)}| = |s_j^{(\beta)}|$.

Zur Erstellung einer *BLOSUM-r* Matrix mit $r \in [0, 100]$ werden zunächst innerhalb jedes Blockes $\beta \in [1:B]$ die Sequenzen nach ihrer Ähnlichkeit geclustert, d.h. für jedes $\beta \in [1:B]$ existiert eine Partition $C_1^{(\beta)} \cup \cdots \cup C_{m_\beta}^{(\beta)} = [1:n_\beta]$, wobei m_β die Anzahl der Cluster in Block β ist. Hierbei gehören zwei Sequenzen zum gleichen Cluster, wenn ihre Sequenzidentität mindestens $r\%$ beträgt. Das Clustering selbst wird dann als reflexiver und transitiver Abschluss definiert. Algorithmisch entspricht dies genau einem single-linkage Clustering, wobei das Verfahren endet, wenn die Sequenzidentität der verbleibenden Cluster kleiner als $r\%$ wird. In Abbildung 7.2 ist ein Beispiel einer solchen Eingabe $\mathcal{S}$ für die BLOSUM-Konstruktion mit $B = 3$ gegeben.

$s_1^{(1)} = \text{ABCAB}$	$s_1^{(2)} = \text{ABC}$	$s_1^{(3)} = \text{AAACBABC}$
$s_2^{(1)} = \text{ABCAC}$	$s_2^{(2)} = \text{ABC}$	$s_2^{(3)} = \text{BAACBABC}$
$s_3^{(1)} = \text{BBCAB}$	$s_3^{(2)} = \text{AAC}$	$s_3^{(3)} = \text{AAACBACB}$
$s_4^{(1)} = \text{CBCAB}$	$s_4^{(2)} = \text{CBC}$	$s_4^{(3)} = \text{AAACBACC}$
$s_5^{(1)} = \text{AAACB}$	$s_5^{(2)} = \text{AAB}$	
	$s_6^{(2)} = \text{BAB}$	

Abbildung 7.2: Beispiel: Eingabe-Blöcke für die Erstellung einer BLOSUM-Matrix

Wird nun ein Clustering der Sequenzen innerhalb der Blöcke des Beispiels in Abbildung 7.2 für $r = 80$ vorgenommen, so liefert das folgendes Clustering:

$$\begin{aligned} [1:5] &= [1:4] \cup \{5\} \\ [1:6] &= [1:2] \cup \{3\} \cup \{4\} \cup \{5\} \cup \{6\} \\ [1:4] &= [1:4] \end{aligned}$$

Mit Hilfe des Clusterings innerhalb der Blöcke will man verhindern, dass zu ähnliche Sequenzen einen zu großen Einfluss auf die Scoring-Matrix-erhält. Im Folgenden werden Sequenzen eines Cluster quasi als eine Sequenz (besser als ein Profil) behandelt. Beachte, dass zwei Sequenzen zu einem Cluster gehören, wenn ihre Sequenzidentität mindestens $r\%$ beträgt. Allerdings können sich aufgrund des single-linkage Clusterings auch zwei Sequenzen in einem Cluster befinden, deren Sequenzidentität deutlich niedriger als $r\%$ ist.

Innerhalb eines jeden Blockes bestimmen wir nun zunächst die Häufigkeit von geordneten Paaren $(a, b) \in \Sigma^2$, wobei wir nur Paare in verschiedenen Cluster berücksichtigen und die Anzahlen jeweils durch die beteiligten Clustergrößen normalisieren. Wir betrachten zuerst einen Block $\beta \in [1 : B]$ und darin zwei verschiedene Cluster $p, q \in [1 : m_\beta]$, dann definieren für jedes $a, b \in \Sigma$:

$$H_{p,q}^{(\beta)}(a,b) = \sum_{\ell=1}^{|s_1^{(\beta)}|} \sum_{i \in C_p^{(\beta)}} \sum_{j \in C_q^{(\beta)}} \frac{\delta_{s_{i,\ell}^{(\beta)},a}}{|C_p^{(\beta)}|} \cdot \frac{\delta_{s_{j,\ell}^{(\beta)},b}}{|C_q^{(\beta)}|} \cdot (1 - \delta_{p,q}).$$

Für einen festen Block $\beta \in [1 : n]$ erhalten wir dann für jedes $a, b \in \Sigma$:

$$H^{(\beta)}(a,b) = \sum_{p=1}^{m_\beta} \sum_{q=1}^{m_\beta} H_{p,q}^{(\beta)}(a,b).$$

Insgesamt gilt dann für jedes $a, b \in \Sigma$:

$$H(a,b) = \sum_{\beta=1}^{B} H^{(\beta)}(a,b).$$

Für die in Abbildung 7.2 angegebenen Sequenzen ergibt sich dann die in Abbildung 7.3 angegebene Häufigkeitsmatrix. Manchmal wird auch die Diagonale noch durch 2 dividiert (was wir hier im Folgenden jedoch nicht tun wollen), da je nach Sichtweise es so interpretieren kann, dass Paare (A, A) für $A \in \Sigma$ doppelt gezählt werden.

$H(\cdot,\cdot)$	A	B	C
A	13	41/4	21/4
B	41/4	11/2	29/4
C	21/4	29/4	6

Abbildung 7.3: Beispiel: BLOSUM-Häufigkeitsmatrix

Wir setzen dann für $a, b \in \Sigma$

$$q_{ab} = \frac{H(a,b)}{\sum_{a,b \in \Sigma} H(a,b)}$$

und

$$p_a = \frac{\sum_{b\in\Sigma} H(a,b)}{\sum_{a,b\in\Sigma} H(a,b)},$$

wobei $\sum_{a,b\in\Sigma} H(a,b) = 70$ ist. Hier kann man die Wahrscheinlichkeit für $q_{a,b}$ direkt aus den relativen Häufigkeiten abschätzen, da wir keine Liste sicherer Mutationen bekommen, sondern tatsächlich die Anzahl beobachtete Alignierungen von zwei Symbolen.

Daraus lassen sich die folgende Auftrittswahrscheinlichkeiten der einzelnen Buchstaben berechnen:

$$\begin{aligned} p_a &:= \frac{1}{70}\left(13 + \frac{41}{4} + \frac{21}{4}\right) = \frac{114}{280} \\ p_b &:= \frac{1}{70}\left(\frac{41}{4} + \frac{11}{2} + \frac{29}{4}\right) = \frac{92}{280} \\ p_c &:= \frac{1}{70}\left(\frac{21}{4} + \frac{29}{4} + 6\right) = \frac{74}{280} \end{aligned}$$

Die daraus resultierende BLOSUM80-Matrix vermöge $w(a,b) = \log_2\left(\frac{q_{a,b}}{p_a \cdot p_b}\right)$ ist in Abbildung 7.3 dargestellt.

$w(\cdot,\cdot)$	A	B	C
A	0.164	0.130	-0.441
B	0.130	-0.458	0.254
C	-0.441	0.254	0.295

Abbildung 7.4: Beispiel: BLOSUM80

Normalerweise wird die BLOSUM-Matrix anschließend noch mit einer Konstanten multipliziert und auf ganzzahlige Werte gerundet (oder auch nur auf eine Nachkommastelle). Auf die Wahl einer sinnvollen Normierungskonstanten gehen wir im folgenden Abschnitt ein.

09.01.20

7.2.4 Wahl einer sinnvollen Scoring-Matrix

Wie wir eben schon gesehen haben, liegt den Scoring-Matrizen im Wesentlichen die folgende Formel zugrunde:

$$w(a,b) = c \cdot \ln\left(\frac{q_{a,b}}{p_a \cdot p_b}\right),$$

wobei $(p_a)_{a\in\Sigma}$ den relativen Häufigkeiten der Buchstaben aus Σ und $(q_{a,b})_{a,b\in\Sigma}$ der relativen Häufigkeiten von homologen Paaren über Σ entspricht.

Daraus ergibt sich die Wahl von $(q_{a,b})_{a,b\in\Sigma}$, wenn man die Scoring-Matrix w als bekannt voraussetzt:

$$q_{a,b} = p_a \cdot p_b \cdot e^{\frac{w(a,b)}{c}}.$$

Damit $(q_{a,b})_{a,b\in\Sigma}$ eine Wahrscheinlichkeitsverteilung auf Σ^2 ist, muss also weiter gelten:

$$1 = \sum_{a,b\in\Sigma} q_{a,b} = \sum_{a,b\in\Sigma} p_a \cdot p_b \cdot e^{\frac{w(a,b)}{c}}.$$

Nach der Definition aus dem vorherigen Abschnitt muss also $c = 1/\lambda$ sein, da wir dort λ genau so definiert hatten.

Für eine so definierte Wahrscheinlichkeitsverteilung der homologen Paare gilt für den Erwartungswert des Score-Beitrags eines Paares:

$$\sum_{a,b\in\Sigma} q_{a,b} \cdot \lambda^{-1} \ln\left(\frac{q_{a,b}}{p_a \cdot p_b}\right) = \frac{H(Q\|P)}{\lambda \cdot \log_2(e)},$$

wobei $H(Q\|P)$ die relative Entropie der Verteilung Q bezüglich P ist.

Definition 7.4 *Seien $P = (p_i)_{i\in\mathcal{I}}$ und $Q = (q_i)_{i\in\mathcal{I}}$ zwei Wahrscheinlichkeitsverteilungen auf derselben Menge von Elementarereignissen $\mathcal{I}$, dann heißt*

$$H(Q\|P) = \sum_{i\in\mathcal{I}} q_i \log_2\left(\frac{q_i}{p_i}\right)$$

die relative Entropie *von Q bezüglich P (oft auch als* Kullback-Leibler-Distanz *oder als* Kullback-Leibler-Information *bezeichnet).*

Der Buchstabe H ist dabei der große griechische Buchstabe Eta (Abkürzung für Entropie). Streng genommen muss im Lemma noch folgendes vorausgesetzt werden: $p_i = 0 \Rightarrow q_i = 0$. Beachte, dass die relative Entropie nicht symmetrisch ist, d.h. im Allgemeinen gilt **nicht** $H(P\|Q) = H(Q\|P)$, und dass im Allgemeinen die Dreiecksungleichung nicht gilt. Allerdings gilt $H(P\|Q) \geq 0$, wie das folgende Lemma zeigt.

Lemma 7.5 *Seien $P = (p_i)_{i\in\mathcal{I}}$ und $Q = (q_i)_{i\in\mathcal{I}}$ zwei Wahrscheinlichkeitsverteilungen auf derselben Menge von Elementarereignissen $\mathcal{I}$. Dann ist $H(Q\|P) \geq 0$. Die Gleichheit gilt genau dann, wenn $P \equiv Q$ ist.*

Beweis: Zunächst halten wir fest, dass $\ln(x) \leq x-1$ für $x \in \mathbb{R}_+^*$ gilt. Dann erhalten wir:

$$\begin{aligned}
H(Q\|P) &= \sum_{i\in\mathcal{I}} q_i \log_2\left(\frac{q_i}{p_i}\right) \\
&= \frac{1}{\ln(2)} \sum_{i\in\mathcal{I}} q_i \ln\left(\frac{q_i}{p_i}\right) \\
&= \frac{1}{\ln(2)} \sum_{i\in\mathcal{I}} q_i \left(-\ln\left(\frac{p_i}{q_i}\right)\right) \\
&\geq \frac{-1}{\ln(2)} \sum_{i\in\mathcal{I}} q_i \left(\frac{p_i}{q_i} - 1\right) \\
&= \frac{1}{\ln(2)} \sum_{i\in\mathcal{I}} q_i \left(1 - \frac{p_i}{q_i}\right) \\
&= \frac{1}{\ln(2)} \left(\sum_{i\in\mathcal{I}} q_i - \sum_{i\in\mathcal{I}} p_i\right) \\
&= 0.
\end{aligned}$$

Die Gleichheit in $\ln(x) \leq x-1$ gilt genau dann, wenn $x = 1$. Somit gilt oben die Gleichheit genau dann, wenn $p_i/q_i = 1$ für alle $i \in \mathcal{I}$. ∎

Also gilt für den Erwartungswert des Score-Beitrags eines Paares aus Σ^2 gemäß der Wahrscheinlichkeitsverteilung $q_{a,b}$:

$$\sum_{a,b\in\Sigma} q_{a,b} \cdot \lambda^{-1} \ln\left(\frac{q_{a,b}}{p_a \cdot p_b}\right) = \frac{H(Q\|P)}{\lambda \cdot \log_2(e)} > 0.$$

Andererseits gilt jedoch für den Erwartungswert des Score-Beitrags eines Paares aus Σ^2 gemäß der Wahrscheinlichkeitsverteilung $p_a \cdot p_b$:

$$\sum_{a,b\in\Sigma} p_a \cdot p_b \cdot \lambda^{-1} \ln\left(\frac{q_{a,b}}{p_a \cdot p_b}\right) = -\sum_{a,b\in\Sigma} p_a \cdot p_b \cdot \lambda^{-1} \ln\left(\frac{p_a \cdot p_b}{q_{a,b}}\right) = -\frac{H(P\|Q)}{\lambda \cdot \log_2(e)} < 0,$$

wie wir es bereits gefordert hatten.

7.3 Statistische Inferenz

In diesem Abschnitt werden wir einige Konzepte zur statistischen Inferenz aus der Einführungsvorlesung zur Stochastik wiederholen, die wir zum einen bereits implizit verwendet haben und zum anderen im Folgenden noch verwenden werden.

7.3.1 Maximum-Likelihood-Schätzer

Oft ist bei der Modellierung das probabilistische Modell relativ schnell klar. Beispielsweise wie beim Münzwurf, da wir hier davon ausgehen können, dass der Aufgang nicht wirklich vom Zeitpunkt oder vom Werfenden abhängt, sofern die Münze nur hoch genug geworfen wird. Allerdings hängt die Wahrscheinlichkeit, ob Kopf oder Zahl oben liegt, von der Massenverteilung innerhalb der Münze ab.

Will man nun in einem gegebenen probabilistischen Modell M die zugehörigen Parameter θ schätzen, bedient man sich gerne der Maximum-Likelihood-Methode. Wir wählen dann die Parameter des Modells so, dass sie die Daten x am besten reproduzieren.

Definition 7.6 *Sei $M(\theta)$ ein probabilistisches Modell mit Parametern $\theta \in \Theta$. Dann ist die* Likelihood-Funktion *für ein Ereignis x definiert als:*

$$L(\theta) := L(\theta; x) := \mathrm{Ws}[x \mid M(\theta)].$$

Dann ist die Log-Likelihood-Funktion *für ein Ereignis x definiert als:*

$$\hat{L}(\theta) := \hat{L}(\theta; x) := \log\left(\mathrm{Ws}[x \mid M(\theta)]\right).$$

Also gibt $L(\theta; x)$ die Wahrscheinlichkeit an, mit dem die beobachteten Daten x für den Parameter θ reproduziert werden können. Daher wählen wir als Parameter einen so genannten *Maximum-Likelihood-Schätzer*:

$$\theta^* := \operatorname{argmax}\left\{L(\theta) \;:\; \theta \in \Theta\right\} = \operatorname{argmax}\left\{\mathrm{Ws}[x \mid M(\theta)] \;:\; \theta \in \Theta\right\},$$

wobei hier wieder Θ die Menge der mögliche Werte des/der Parameter θ ist.

Definition 7.7 *Sei $M(\theta)$ ein probabilistisches Modell mit Parameterraum Θ. Für einen gegebenen Datensatz x heißt θ^** Maximum-Likelihood-Schätzer*, wenn*

$$\theta^* := \operatorname{argmax}\left\{L(\theta) \;:\; \theta \in \Theta\right\} = \operatorname{argmax}\left\{\mathrm{Ws}[x \mid M(\theta)] \;:\; \theta \in \Theta\right\}.$$

Häufig ist es mathematisch einfacher, das Maximum der Log-Likelihood-Funktion anstatt der Likelihood-Funktion zu bestimmen. Da der Logarithmus eine monoton wachsende Funktion ist, werden die Maxima an denselben Stellen angenommen.

Beispielsweise hatten wir die Wahrscheinlichkeiten $(p_a)_{a\in\Sigma}$ des Auftretens von Buchstaben eines Alphabets (eigentlich Aminosäuren) als Maximum-Likelihood-Schätzer bestimmt. Der Leser möge nachrechnen, dass die relative Häufigkeit eines Ereignisses der Maximum-Likelihood-Schätzer für seine Wahrscheinlichkeitsverteilung ist.

7.3.2 Einfache Hypothesen-Tests

Oft will man wissen ob, ein vorgegebenes probabilistisches Modell eines zu untersuchenden Sachverhalt gut oder weniger gut beschreibt. In einem ersten Schritt beschreibt man ein probabilistisches Modell, das in der Regel auch einen oder mehrere Parameter zur Wahrscheinlichkeitsverteilung enthält. Die so genannte *Null-Hypothese* besagt, dass dieses Modell die (erst noch zu ermittelnden) Daten gut erklären kann. Gut erklären ist hier im Sinne von gut vorhersagen gemeint, d.h. dass die Wahrscheinlichkeit, dass ein solches oder ähnliches Ereignis eintritt, groß ist. Dabei wird in der Regel ein Parameter im Modell angenommen, dass die Daten im Weitesten Sinne rein zufällig erzeugt werden (ohne eine systematische Abweichung bzw. Verzerrung). Im zweiten Schritt wird dann die so genannte *Alternativ-Hypothese* definiert. Meist besagt sie nur, dass die Null-Hypothese nicht stimmt oder gibt eine Richtung vor, in der man annimmt, dass die Parameter in diese Richtung verschoben sind.

Definition 7.8 *Sei f die Dichte der Wahrscheinlichkeitsverteilung, die der Null-Hypothese zugrunde liegt. Dann heißt $\alpha \in [0,1]$ das* Signifikanz-Niveau*, wenn*

$$\alpha = \int_C f(x)\,dx \qquad \text{bzw.} \qquad \frac{\alpha}{2} = \int_{C_1} f(x)\,dx = \int_{C_2} f(x)\,dx$$

mit $C = [k, \infty)$ bzw. $C_1 = (-\infty : -k_1]$ und $C_2 = [k_2 : +\infty)$. Hierbei heißt C bzw. $C_1 \cup C_2$ der kritische Bereich*. Für $x \in C$ bzw. $x \in C_1 \cup C_2$ wird die Null-Hypothese verworfen, ansonsten nicht. Der Wert k wird als* Signifikanz-Punkt *bezeichnet.*

Dann wird auch noch ein Signifikanz-Niveau festgelegt, in der Regel 0.05, 0.01 oder 0.001. Man beachte, dass es beim Hypothesen-Test zwei Arten von Fehlern gibt.

Definition 7.9 *Betrachte die Null-Hypothese bei einem einfachen Hypothesen-Test.*

- *Der* Fehler erster Art *tritt ein, wenn die Null-Hypothese fälschlicherweise verworfen wird.*
- *Der* Fehler zweiter Art *tritt ein, wenn die Null-Hypothese fälschlicherweise nicht verworfen wird.*

Wenn man das probabilistische Modell und die zugehörigen Hypothesen definiert hat sowie das Signifikanz-Niveau festgelegt hat, werden experimentelle Daten erhoben und analysiert, wie gut sich diese mit der Null-Hypothese erklären lassen. Dazu wird die Wahrscheinlichkeit ermittelt, mit der sich ein Ereignis wie das beobachtete oder ein noch extremeres (bzgl. der Alternativ-Hypothese) in dem zugrunde liegenden

probabilistischen Modell beobachten lässt. Daher hängt die Definition extremerer Ereignisse auch von der Alternativ-Hypothese ab.

Ist nun diese Wahrscheinlichkeit, die als *P-Value* bezeichnet wird, kleiner als das vorher festgelegte Signifikanz-Niveau, so verwirft man die Null-Hypothese, andernfalls verwirft man sie nicht. Wenn man die Null-Hypothese nicht verwirft, bedeutet dies nicht, dass diese korrekt ist. Man hat nur keine hinreichenden Anhaltspunkte, um Sie für falsch zu halten.

Betrachten wir ein kurzes Beispiel, nämlich das Werfen einer Münze. In der Null-Hypothese nehmen wir an, dass Kopf und Zahl gleichwahrscheinlich sind, also rein zufällig mit gleicher Wahrscheinlichkeit. Die Wahrscheinlichkeit für Kopf sei also $p = 0.5$ und für Zahl $q = 1 - p = 0.5$. Die Alternativ-Hypothese besagt, dass die Münze unfair ist, also eine Seite bevorzugt. Je nachdem, in welchem Kontext wir uns befinden, kann es sein, dass wir keine Annahme darüber haben, ob Kopf oder Zahl wahrscheinlicher ist, oder, wenn wir um Geld mit einem Partner spielen, der bei Kopf gewinnt, dass die Kopf-Seite häufiger ist. Als Signifikanz-Niveau legen wir in unserem Beispiel $\alpha := 0.05$ fest.

Wir werfen nun 100 mal eine Münze und erhalten 60 mal Kopf und 40 mal Zahl. Sei also X_i die Indikatorvariable, dass der i-te Wurf Kopf zeigt. Wir wollen jetzt also die Wahrscheinlichkeit berechnen, dass dieses oder ein extremeres Ereignis eintritt. Wir suchen also die Wahrscheinlichkeit:

$$\mathrm{Ws}\left[\sum_{i=1}^{100} X_i \geq 60\right] \quad \text{bzw.} \quad \mathrm{Ws}\left[\sum_{i=1}^{100} X_i \geq 60 \vee \sum_{i=1}^{100} X_i \leq 40\right],$$

je nachdem, welche Alternativ-Hypothese wir betrachten. Hierbei beschreibt der P-Value im ersten Fall, dass wir mit einem Partner um Geld spielen, im zweiten Fall, dass wir nicht wissen, wie die Münze manipuliert ist (bzw. von einer idealen Münze abweicht). Wenn wir nicht wissen, wie die Münze manipuliert sein könnte, ist das Erscheinen von 40 Mal Zahl genauso abwegig, wie 40 Mal Kopf. Daher sind die extremeren Ereignis mindestens 60 Mal Kopf oder höchstens 40 Mal Kopf. Bei der zweiten Alternativ-Hypothese, wenn wir um Geld spielen, ist das Ereignis 40 Mal Zahl hingegen kein extremes Ergebnis, da wir ja eine Abweichung Richtung Kopf in dieser Hypothese unterstellen.

Dies ist wieder eine Binomialverteilung, es gilt also im ersten Fall:

$$\mathrm{Ws}\left[\sum_{i=1}^{100} X_i \geq 60\right] = \sum_{j=60}^{100} \binom{100}{j} p^j q^{100-j} = \frac{1}{2^{100}} \sum_{j=60}^{100} \binom{100}{j}.$$

Im zweiten Fall haben wir aufgrund der Symmetrie der Binomialverteilung die doppelte Wahrscheinlichkeit. Wir können nun diese Wahrscheinlichkeit berechnen, was

nicht immer unbedingt einfach möglich ist, oder auch approximieren (beispielsweise mit der Normalverteilung oder mit Chernov-Schranken). Hier gilt nun

$$\text{Ws}\left[\sum_{i=1}^{100} X_i \geq 60\right] \approx 0.028.$$

Im ersten Fall ($p \neq \frac{1}{2}$) würden wir also die Null-Hypothese verwerfen, im zweiten Fall ($p > \frac{1}{2}$) jedoch nicht.

Bei der Signifikanz vor Scores von lokalen Alignments haben wir also implizit einen einfachen Hypothesen-Test gemacht, ohne das Signifikanz-Niveau explizit festgelegt zu haben. Zu beachten ist, dass der P-Value nicht die Wahrscheinlichkeit angibt, dass die Null-Hypothese falsch ist, sondern das Signifikanz-Niveau angibt, mit der die Null-Hypothese fälschlicherweise verworfen wird.

7.3.3 Likelihood-Ratio-Tests

Bei einfachen Hypothesen-Tests werden in der Alternativ-Hypothese (wenn sie überhaupt explizit angegeben wird) in der Regel keine besonderen Parameter zum verwendeten probabilistischen Modell angegeben. Wenn man eine konkrete Alternative im Auge hat, kann man die P-Values für die Null- und Alternativ-Hypothese explizit miteinander vergleichen. Beispielsweise hatten wir beim Erstellen von Scoring-Matrizen zwei explizite Parameter für das Alignment von Sequenzen hergeleitet: Im Null-Modell R war $p_a \cdot p_b$ die Wahrscheinlichkeit, dass in einem Alignment das Paar (a, b) auftritt, im Alternativ-Modell M war es $q_{a,b}$. Daher betrachtet man dann die so genannte Likelihood-Ratio.

Definition 7.10 *Sei M ein probabilistisches Modell und sei θ_0 bzw. θ_1 die Parameter für die Null- bzw. Alternativ-Hypothese. Dann ist die* Likelihood-Ratio *definiert als:*

$$\Lambda(x) := \frac{L(\theta_0; x)}{L(\theta_1; x)}.$$

Darauf basiert der so genannte *Likelihood-Ratio-Test*. Ist der Wert $\Lambda(x)$ zu klein (was immer das genau heißt, werden wir gleich sehen), so wird die Null-Hypothese zu Gunsten der Alternativ-Hypothese verworfen. Ansonsten wird die Null-Hypothese nicht verworfen.

Definition 7.11 *Seien $\theta_0 \in \Theta_M$ bzw. $\theta_1 \in \Theta_M$ die Parameter für die Null- bzw. Alternativ-Hypothese des probabilistischen Modells M. Sei weiter $\alpha \in [0,1]$ sowie $\lambda \in \mathbb{R}_+$ mit*

$$\mathrm{Ws}[\Lambda(x) \leq \lambda \mid \theta_0] = \alpha.$$

Für das Signifikanz-Niveau α *heißt dann* λ *der* Signifikanz-Punkt *des Likelihood-Ratio-Tests. Mit $C = \{x : \Lambda(x) \leq \lambda\}$ wird die* kritische Region *des Likelihood-Ratio-Tests bezeichnet.*

Nehmen wir an, wir verwerfen die Null-Hypothese, wenn $\Lambda(x) \leq \lambda$, dann gilt $\mathrm{Ws}[\Lambda(x) \leq \lambda \mid \theta_0] = \alpha$. Das bedeutet, dass wir mit Wahrscheinlichkeit α die Null-Hypothese fälschlich verwerfen. Das bedeutet, dass wir auch hier wieder durch Einführen eines Signifikanz-Niveaus α den Parameter λ zur Ablehnung der Null-Hypothese festlegen. Die kritische Region bezeichnet also die Menge, für die wir die Null-Hypothese verwerfen.

Betrachten wir noch einmal das Beispiel mit dem Münzwurf: 60 von hundert Würfen waren Kopf. Wir nehmen für die Null-Hypothese wiederum $p = 1/2 =: \theta_0$ an. Für die Alternativ-Hypothese gehen wir von $p = 2/3 =: \theta_1$ aus, zum Beispiel, weil wir wissen, dass unser Partner neben normalen (also fairen) Münzen, eine gezinkte Münze besitzt, deren Wahrscheinlichkeit für Kopf bei θ_1 liegt. Die zugehörige Verteilung lautet dann

$$\mathrm{Ws}[X = x] = \binom{100}{x} p^x (1-p)^{100-x}.$$

Damit ergibt sich (wobei x die Anzahl der Versuche ist, in der Kopf oben lag):

$$\begin{aligned} L(x;\theta_0) &= \binom{100}{x} \cdot \frac{1}{2^{100}}, \\ L(x;\theta_1) &= \binom{100}{x} \cdot \frac{2^x}{3^{100}}. \end{aligned}$$

Damit erhalten wir für die Likelihood-Ratio

$$\Lambda(x) = \frac{L(\theta_0;x)}{L(\theta_1;x)} = \left(\frac{3}{2}\right)^{100} \cdot \left(\frac{1}{2}\right)^x.$$

Wir wollen also für folgende Wahrscheinlichkeit $\mathrm{Ws}[\Lambda(X) \leq \lambda \mid \theta_0]$ nach Möglichkeit einen geschlossenen Ausdruck erhalten. Man rechnet leicht nach, dass $\Lambda(x) \leq \lambda$ genau dann gilt, wenn

$$x \geq \log_2 \left(\frac{1}{\lambda} \cdot \left(\frac{3}{2}\right)^{100}\right)$$

gilt. Sei X die Zufallsvariable, die die Anzahl Köpfe bei unseren Versuchen zählt. Dann ist

$$\mathrm{Ws}[\Lambda(X) \leq \lambda \mid \theta_0] = \mathrm{Ws}\left[X \geq \log_2\left(\frac{1}{\lambda} \cdot \left(\frac{3}{2}\right)^{100}\right) \mid \theta_0\right].$$

Für ein gegebenes Signifikanz-Niveau wollen wir jetzt λ so bestimmen, dass

$$\mathrm{Ws}\left[X \geq \log_2\left(\frac{1}{\lambda} \cdot \left(\frac{3}{2}\right)^{100}\right) \mid \theta_0\right] = \alpha.$$

Wie wir schon wissen, gilt

$$\mathrm{Ws}[X \geq B \mid \theta_0] = \sum_{i=B}^{100} \binom{100}{i} \frac{1}{2^{100}}.$$

Die letzte Summe ist für $B \approx 59$ etwa 0.05. Diese Berechnung sind in der Regel aufwendig und können meist nur mit Approximationen gelöst werden Damit lässt sich λ aus B berechnen:

$$B = \log_2\left(\frac{1}{\lambda} \cdot \left(\frac{3}{2}\right)^{100}\right).$$

Also gilt

$$\lambda = \left(\frac{3}{2}\right)^{100} \cdot \frac{1}{2^B}.$$

Somit ist $\lambda \approx 0.71$. Da $\Lambda(60) \approx 0.35 < \lambda$ gilt, verwerfen wir die Null-Hypothese.

14.01.20

Lemma 7.12 (Lemma von Neyman und Pearson) *Seien θ_0 bzw. θ_1 die Parameter für die Null- bzw. Alternativ-Hypothese des probabilistischen Modells M, wobei $f_0(x) := \mathrm{Ws}[x \mid \theta_0]$ und $f_1(x) := \mathrm{Ws}[x \mid \theta_1]$ den gleichen Träger besitzen. Dann ist unter allen Tests mit einem Signifikanz-Niveau kleiner gleich $\alpha \in [0,1]$ der Test mit dem kleinsten Fehler zweiter Art gegeben durch die kritische Menge $C = \{x : \Lambda(x) \leq \lambda\}$, wobei λ implizit definiert ist durch*

$$\alpha = \mathrm{Ws}[x \in C \mid \theta_0] = \int_C f_0(x)\, dx.$$

Zur Erinnerung: Der Träger einer Funktion ist die Menge der Elemente, auf denen die Funktion nicht verschwindet ($\mathrm{supp}(f) = \{x \in D(f) : f(x) \neq 0\}$).

Beweis: Wir betrachten einen Test mit einem Signifikanz-Niveau $\beta \leq \alpha$. Für die zugehörige kritische Region D dieses Tests gilt nach Definition des Signifikanz-Niveaus β, dass $\mathrm{Ws}[x \in D \mid \theta_0] = \beta$. Weiter sei für eine beliebige Menge E

$$\chi_E(x) = \begin{cases} 1 \text{ wenn } x \in E, \\ 0 \text{ wenn } x \notin E. \end{cases}$$

Für alle x gilt dann (da nach Definition $\Lambda(x) = \frac{\mathrm{Ws}[x|M(\theta_0)]}{\mathrm{Ws}[x|M(\theta_1)]} = \frac{f_o(x)}{f_1(x)}$)

$$0 \leq (\chi_C(x) - \chi_D(x)) \cdot \left(f_1(x) - \frac{1}{\lambda} \cdot f_0(x) \right),$$

da beide Faktoren immer dasselbe Vorzeichen besitzen, wie wir gleich sehen werden. Gilt $\chi_C(x) = \chi_D(x)$, dann ist der rechte Term 0 und die Ungleichung stimmt. Sei nun $x \in C \setminus D$, dann gilt $\frac{f_0(x)}{f_1(x)} \leq \lambda$ nach Definition von C. Dann ist der erste Faktor positiv und der zweite nichtnegativ. Sei jetzt $x \in D \setminus C$, dann gilt $\frac{f_0(x)}{f_1(x)} > \lambda$ nach Definition von C. Also ist sowohl der erste als auch der zweite Faktor negativ.

Somit erhalten wir

$$\begin{aligned}
0 &\leq \int (\chi_C(x) - \chi_D(x)) \cdot \left(f_1(x) - \frac{1}{\lambda} f_0(x) \right) dx \\
&= \int (\chi_C(x) - \chi_D(x)) \cdot f_1(x)\, dx - \frac{1}{\lambda} \int (\chi_C(x) - \chi_D(x)) \cdot f_0(x)\, dx \\
&= \int_C f_1(x)\, dx - \int_D f_1(x)\, dx - \frac{1}{\lambda} \left(\int_C f_0(x)\, dx - \int_D f_0(x)\, dx \right) \\
&= \mathrm{Ws}[x \in C \mid \theta_1] - \mathrm{Ws}[x \in D \mid \theta_1] - \frac{1}{\lambda} \left(\mathrm{Ws}[x \in C \mid \theta_0] - \mathrm{Ws}[x \in D \mid \theta_0] \right) \\
&\quad \text{da } \mathrm{Ws}[x \in C \mid \theta_0] = \alpha \text{ und } \mathrm{Ws}[x \in D \mid \theta_0] \leq \beta \\
&\leq \mathrm{Ws}[x \in C \mid \theta_1] - \mathrm{Ws}[x \in D \mid \theta_1] - \frac{1}{\lambda} (\alpha - \beta) \\
&\quad \text{da } \beta \leq \alpha \\
&\leq \mathrm{Ws}[x \in C \mid \theta_1] - \mathrm{Ws}[x \in D \mid \theta_1].
\end{aligned}$$

Also gilt

$$\mathrm{Ws}[x \in D \mid \theta_1] \leq \mathrm{Ws}[x \in C \mid \theta_1]$$

und somit auch

$$1 - \mathrm{Ws}[x \in D \mid \theta_1] \geq 1 - \mathrm{Ws}[x \in C \mid \theta_1].$$

Nach Definition der Gegenwahrscheinlichkeit erhalten wird dann aber

$$\mathrm{Ws}[x \notin C \mid \theta_1] \leq \mathrm{Ws}[x \notin D \mid \theta_1],$$

was gerade die Behauptung des Satzes ist. ■

Der Likelihood-Ratio-Test kann auch für Mengen von Parametern definiert werden. Dann ist $\Theta_0 \subseteq \Theta_M$ die Menge der möglichen Parameter, die man für die Null-Hypothese verwenden will und $\Theta_1 := \Theta_M \setminus \Theta_0$ die Menge der möglichen Parameter für die Alternativ-Hypothese. Diese ist eine Verallgemeinerung unseres bisherigen Likelihood-Ratio-Test, für den dann $\Theta_M = \{\theta_0, \theta_1\}$ war. Die Likelihood-Ratio ist dann definiert als

$$\Lambda(x) := \frac{\sup\{L(\theta_0; x) \,:\, \theta_0 \in \Theta_0\}}{\sup\{L(\theta_1; x) \,:\, \theta_1 \in \Theta_1\}}.$$

7.3.4 Bayes'scher Ansatz

Bisher haben wir nur den so genannten frequentistischen Ansatz verfolgt. Hierbei überlegt man sich für die gegebenen Daten, was die Wahrscheinlichkeit ist, dass das betrachtete Modell diese Daten generiert. Hauptmanko des frequentistischen Ansatzes ist, dass man keine quantitative Aussagen über die Korrektheit der getroffenen Aussage über das Modell machen kann. Der vorhergesagte Parameter ist entweder korrekt oder falsch.

Alternativ hierzu gibt es den Bayes'schen Ansatz, der letztendlich fragt, was ist die Wahrscheinlichkeit, dass das Modell (die Hypothese) wahr ist, wenn man die gegebenen Daten betrachtet. Hierbei wird alles, insbesondere auch die Parameter als Zufallsvariable betrachtet. Dabei wird die Wahrscheinlichkeit nicht mehr nur als eine messbare Größe (beispielsweise Verhältnis der Anzahl aller günstigen Ereignisse zu allen möglichen Ereignissen) betrachtet, sondern als eine subjektive Sicherheit in dem Ereignis. Auch hier gelten die übrigen Eigenschaften einer Wahrscheinlichkeitsmaßes. Um den Unterschied deutlich zu machen, schreiben wir im Folgenden kurzzeitig statt Ws jedoch Prob. Die zugehörige Dichtefunktion (insbesondere im kontinuierlichen Fall) bezeichnen wir wiederum mit f.

Ausgangspunkt ist die folgende Beziehung für zwei nicht notwendigerweise stochastisch unabhängiger Ereignisse A und B:

$$\mathrm{Ws}[A, B] = \mathrm{Ws}[A] \cdot \mathrm{Ws}[B \mid A] = \mathrm{Ws}[B] \cdot \mathrm{Ws}[A \mid B].$$

Daraus leitet sich das so genannte *Bayes'sche Theorem* ab, das für zwei Ereignisse A und B besagt:

$$\mathrm{Ws}[A \mid B] = \frac{\mathrm{Ws}[A, B]}{\mathrm{Ws}[B]} = \frac{\mathrm{Ws}[A] \cdot \mathrm{Ws}[B \mid A]}{\mathrm{Ws}[B]}.$$

Im Bayes'schen Ansatz, will man also $\mathrm{Prob}[H \mid D]$ berechnen, also die Sicherheit, mit der die Hypothese H gegeben die Daten D korrekt ist. Mit

$$\mathrm{Prob}[H \mid D] = \frac{\mathrm{Prob}[H] \cdot \mathrm{Prob}[D \mid H]}{\mathrm{Prob}[D]}$$

kann man dann letztendlich diese Sicherheit aus einer vorgegebenen Sicherheit der Hypothese H und der Wahrscheinlichkeit der Daten D gegeben die Hypothese H bestimmen. Beachte, dass der Nenner $\text{Prob}[D]$ von der Hypothese unabhängig ist und quasi nur einen Normalisierungsfaktor darstellt.

Definition 7.13 *In der Gleichung*

$$\text{Prob}[H \mid D] = \frac{\text{Prob}[D, H]}{\text{Prob}[D]} = \frac{\text{Prob}[H] \cdot \text{Prob}[D \mid H]}{\text{Prob}[D]}$$

werden

- $\text{Prob}[D \mid H]$ *als* Likelihood Function*,*
- $\text{Prob}[H]$ *als* Prior Distribution*,*
- $\text{Prob}[H \mid D]$ *als* Posterior Distribution
- *und* $\text{Prob}[D]$ *als* Evidence *bezeichnet.*

Für die Dichtefunktion gilt dann für ein Ereignis x und einen Parameter θ des zugrunde liegenden probabilistischen Modells gilt dann:

$$f(\theta \mid x) = \frac{f_0(\theta) \cdot f(x \mid \theta)}{f(x)}.$$

Definition 7.14 *In der Gleichung*

$$f(\theta \mid x) = \frac{f_0(\theta) \cdot f(x \mid \theta)}{f(x)}$$

werden ebenfalls $f(x \mid \theta)$ *als* Likelihood Function*,* $f_0(\theta)$ *als* Prior Density*,* $f(\theta \mid x)$ *als* Posterior Density *und* $f(x)$ *als* Evidence *bezeichnet.*

Der Vollständigkeit halber erwähnen wir noch die folgende Definition als Erinnerung aus der Stochastik.

Definition 7.15 $\text{Prob}[A, B] = \text{Prob}[A \cap B]$ *bezeichnet man die* gemeinsame Verteilung *bzw.* Joint Distribution *von* A *und* B*.* $\text{Prob}[A] = \sum_B \text{Prob}[A, B]$ *bezeichnet die* Randverteilung *bzw.* Marginal Distribution *von* A *und* B*.*

Für Dichtefunktion gelten die Definitionen analog. Wir erinnern noch einmal an den Satz der totalen Wahrscheinlichkeit. Für eine Partition $(B_1, \ldots, B_n)$ eines Ereignisraums, der dem Ereignis A zugrunde liegt, gilt:

$$\mathrm{Ws}[A] = \sum_{j=1}^{n} \mathrm{Ws}[A \wedge B_j] = \sum_{j=1}^{n} \mathrm{Ws}[A \mid B_j] \cdot \mathrm{Ws}[B_j].$$

Im diskreten Fall gilt für $\theta_i \in \Theta = \{\theta_1, \ldots, \theta_n\}$:

$$\mathrm{Ws}[\theta_i \mid x] = \frac{\mathrm{Ws}[\theta_i] \cdot \mathrm{Ws}[x \mid \theta_i]}{\sum_{j=1}^{n} \mathrm{Ws}[\theta_j] \cdot \mathrm{Ws}[x \mid \theta_j]}$$

Die Expansion des Nenners folgt dem Satz von der totalen Wahrscheinlichkeit. Im kontinuierlichen Fall (beispielsweise sei $p \in [0, 1]$, wobei p die Wahrscheinlichkeit, dass beim Münzwurf Kopf erscheint) gilt dann entsprechend:

$$f(\theta_0 \mid x) = \frac{f_0(\theta_0) \cdot f(x \mid \theta_0)}{\int f_0(\theta) \cdot f(x \mid \theta)\, d\theta}.$$

Betrachten wir wieder unser Beispiel mit den Münzen. Wir haben also eine faire Münze ($p = 1/2 =: \theta_0$) und eine gezinkte Münze ($p = 2/3 =: \theta_1$). Wir haben wieder 100 Mal geworfen und als Ergebnis X 60 Mal Kopf erzielt. Nun wollen wir die Wahrscheinlichkeit bestimmen, mit welcher Münze das Ergebnis erzielt wurde. Als Prior nehmen wir $\mathrm{Ws}[\theta_0] = \mathrm{Ws}[\theta_1] = 1/2$ an, weil wir keine weitergehenden Informationen vor dem Experiment habe, welche Münze benutzt sein könnte (außer dass unser Partner beide besitzt). Es gilt also

$$\begin{aligned}
\mathrm{Ws}[\theta_0 \mid X] &= \frac{\mathrm{Ws}[\theta_0] \cdot \mathrm{Ws}[X \mid \theta_0]}{\mathrm{Ws}[X \mid \theta_0] \cdot \mathrm{Ws}[\theta_0] + \mathrm{Ws}[X \mid \theta_1] \cdot \mathrm{Ws}[\theta_1]} \\
&= \frac{\frac{1}{2} \cdot \binom{100}{60} \left(\frac{1}{2}\right)^{60} \left(\frac{1}{2}\right)^{40}}{\binom{100}{60} \left(\frac{1}{2}\right)^{60} \left(\frac{1}{2}\right)^{40} \cdot \frac{1}{2} + \binom{100}{60} \left(\frac{2}{3}\right)^{60} \left(\frac{1}{3}\right)^{40} \cdot \frac{1}{2}} \\
&= \frac{\left(\frac{1}{2}\right)^{60} \left(\frac{1}{2}\right)^{40}}{\left(\frac{1}{2}\right)^{60} \left(\frac{1}{2}\right)^{40} + \left(\frac{2}{3}\right)^{60} \left(\frac{1}{3}\right)^{40}} \\
&= \frac{\frac{1}{2^{100}}}{\frac{1}{2^{100}} + \frac{2^{60}}{3^{100}}} \\
&= \frac{3^{100}}{3^{100} + 2^{160}} \\
&\approx 0.26.
\end{aligned}$$

Analog ergibt sich dann $\mathrm{Ws}[\theta_1 \mid X] \approx 0.74$. Damit hat man nun explizite Wahrscheinlichkeiten für beide Modelle und kann darauf eine Entscheidung treffen. Wenn

wir beispielsweise wissen, dass der Partner die zufällige Münze deutlich öfter als die gezinkte verwendet (oder umgekehrt), kann man den Prior entsprechend wählen und ggf. kann die Wahrscheinlichkeit $\mathrm{Ws}[\theta_0 \mid X]$ für das Modell θ_0 kann größer als die für θ_1 werden (oder umgekehrt). Für eine kontinuierliche Verteilung auf $[0,1]$ kann man auch eine Verteilung ähnlich der Standard-Verteilung wählen, die an einem Punkt $p \in [0,1]$ stärker konzentriert ist (sie muss nicht notwendigerweise symmetrisch sein, die hängt vom Vorwissen ab) .

Definition 7.16 *Sei $M(\theta)$ ein probabilistisches Modell mit Parameterraum Θ. Sei weiter f eine Dichte für den Parameterraum und x ein Ereignis von M, dann heißt*

$$\theta^* = \operatorname{argmax}\{f(\theta \mid x) \ : \ \theta \in \Theta\}$$

der Maximum-A-Posteriori-Schätzer *oder kurz* MAP-Schätzer.

Vergleichen wir nun den Maximum-Likelihood Ansatz mit dem MAP-Schätzer. Wir betrachten dabei statt der ursprünglichen Funktionen wieder die logarithmierten Funktionen. Da der Logarithmus streng monoton wachsend ist, tritt das Maximum an derselben Stelle auf.

Zunächst gilt für den Maximum-Likelihood-Schätzer:

$$\theta^*_{ML} = \operatorname{argmax}\{\log(f(x \mid \theta)) \ : \ \theta \in \Theta\}\,.$$

Für den Maximum-A-Posteriori-Schätzer gilt (mittels der zuvor hergeleiteten Beziehung $f(\theta \mid x) = \frac{f_0(\theta)\cdot f(x|\theta)}{f(x)}$):

$$\begin{aligned}
\theta^*_{MAP} &= \operatorname{argmax}\{\log(f(\theta \mid x)) \ : \ \theta \in \Theta\} \\
&= \operatorname{argmax}\{\log(f(x \mid \theta)) + \log(f_0(\theta)) - \log(f(x)) \ : \ \theta \in \Theta\} \\
&\quad \text{da } \log(f(x)) \text{ unabhängig von } \theta \text{ ist} \\
&= \operatorname{argmax}\{\log(f(x \mid \theta)) + \log(f_0(\theta)) \ : \ \theta \in \Theta\}\,.
\end{aligned}$$

Damit spielt also neben der Likelihood-Function nur die Prior Density bzw. Prior Distribution eine Rolle. Verwendet man also einen uniformen Prior, dann ergeben Maximum-Likelihood- und MAP-Schätzer das gleiche Ergebnis. In der Regel wird man jedoch in den Prior eine gewisse Vorinformation über die Verteilung der Parameter einfließen lassen. Insbesondere in der Bioinformatik macht dies Sinn, da man genau hier biologisches Vorwissen über die Wahl des Parameter einfließen lassen kann.

Man kann diesen Bayes'schen Ansatz auch als ein Hypothesen-Test mit vielen Hypothesen ansehen, wobei der Prior eine Dichte über die Sinnhaftigkeit der Hypothesen (ohne Kenntnis des Experiments) ist.

7.4 EM-Methode (*)

In diesem Abschnitt wollen wir ein Verfahren zur Bestimmung eines ML- oder MAP-Schätzers angeben, den so gennannten Expectation-Maximization-Algorithm oder kurz EM-Algorithmus.

7.4.1 Fehlende Daten

Manchmal lassen sich nicht alle Ergebnisse der Experimente messen. Leider kann aber die Nichtmessbarkeit einen Einfluss auf die Parameter haben. Es kann durchaus sein, dass bei der probabilistischen Modellierung sichtbare Ergebnisse $(\vec{x}, \vec{y})$ vorkommen, wobei sich die Ergebnisse $\vec{x}$ in einem biologischen Experiment messen lassen, die Werte $\vec{y}$ aber nicht messbar sind. In einem anderen Fall liefern einige Messungen kein Ergebnis. Das kann zufällig sein, es kann aber auch einen mehr oder weniger engen Zusammenhang mit den eigentlichen (nicht messbaren) Ergebnissen haben.

Für die Berechnung der Likelihood $L(x;\theta)$ muss man auch nicht messbare bzw. nicht gemessene Daten y berücksichtigen. Wir wollen also den Parameter θ^* finden, so dass $f(x \mid \theta)$ maximal wird, also den so genannten Modus oder Modalwert finden.

7.4.2 Mathematischer Hintergrund

Der EM-Algorithmus ist nun ein Verfahren zum Bestimmen des *Modalwerts* oder *Modus* von der folgenden Randdichte:

$$f(x \mid \theta) = \int f(x, y \mid \theta)\, dy \qquad \text{bzw.} \qquad f(x \mid \theta) \cdot f_0(\theta) = \int f(x, y \mid \theta) \cdot f_0(\theta)\, dy$$

für den ML- bzw. MAP-Schätzer. Hierbei ist $f_0(\theta)$ die Dichte des Priors.

Für das Weitere benötigen wir erst noch ein paar Vorüberlegungen. Zunächst gilt

$$f(y \mid x, \theta) = \frac{f(x, y \mid \theta)}{f(x \mid \theta)}.$$

Dies kann man auch schreiben als

$$\log(f(x \mid \theta)) = \log(f(x, y \mid \theta)) - \log(f(y \mid x, \theta)).$$

Wir versuchen nun eine Folge von Parametern $(\theta^{(0)}, \ldots, \theta^{(t)}, \ldots)$ zu konstruieren, die gegen θ^* konvergieren soll. Um eine Beziehung zwischen den approximierenden

Parametern zu erhalten, multiplizieren wir diese Gleichung mit $f(y \mid x, \theta^{(t)})$ und integrieren über alle möglichen y. Da $\int f(y \mid x, \theta^{(t)})\, dy = 1$ gilt, folgt:

$$\log(f(x \mid \theta)) = \int f(y \mid x, \theta^{(t)}) \cdot \log(f(x, y \mid \theta))\, dy - \int f(y \mid x, \theta^{(t)}) \cdot \log(f(y \mid x, \theta))\, dy.$$

Wir definieren nun

$$Q(\theta \mid \theta^{(t)}) := \int f(y \mid x, \theta^{(t)}) \cdot \log(f(x, y \mid \theta))\, dy.$$

Man kann $Q(\theta \mid \theta^{(t)})$ als einen bedingten Erwartungswert interpretieren:

$$Q(\theta \mid \theta^{(t)}) = \mathbb{E}[\log(f(X, Y \mid \theta)) \mid X = x, \theta^{(t)}].$$

Damit gilt:

$$\begin{aligned}
&\log(f(x \mid \theta)) - \log(f(x \mid \theta^{(t)})) \\
&= \int f(y \mid x, \theta^{(t)}) \cdot \log(f(x, y \mid \theta))\, dy - \int f(y \mid x, \theta^{(t)}) \cdot \log(f(y \mid x, \theta))\, dy \\
&\quad - \int f(y \mid x, \theta^{(t)}) \cdot \log(f(x, y \mid \theta^{(t)}))\, dy + \int f(y \mid x, \theta^{(t)}) \cdot \log(f(y \mid x, \theta^{(t)}))\, dy \\
&= Q(\theta \mid \theta^{(t)}) - Q(\theta^{(t)} \mid \theta^{(t)}) + \int f(y \mid x, \theta^{(t)}) \cdot \log\left(\frac{f(y \mid x, \theta^{(t)})}{f(y \mid x, \theta)}\right) dy.
\end{aligned}$$

Der letzte Summand ist gerade die Kullback-Leibler-Distanz zwischen den Dichten $f(y \mid x, \theta^{(t)})$ und $f(y \mid x, \theta)$ und ist daher größer gleich 0. Also gilt

$$\log(f(x \mid \theta)) - \log(f(x \mid \theta^{(t)})) \geq Q(\theta \mid \theta^{(t)}) - Q(\theta^{(t)} \mid \theta^{(t)}).$$

Wenn wir nun $\theta^{(t+1)}$ wie folgt definieren

$$\theta^{(t+1)} := \operatorname{argmax} \left\{ Q(\theta \mid \theta^{(t)}) \ : \ \theta \in \Theta \right\},$$

gilt dann $Q(\theta^{(t+1)} \mid \theta^{(t)}) - Q(\theta^{(t)} \mid \theta^{(t)}) \geq 0$ (da $\theta^{(t+1)} \in \Theta$) und somit:

$$\log(f(x \mid \theta^{(t+1)})) \geq \log(f(x \mid \theta^{(t)})).$$

Also ist $\theta^{(t+1)}$ ein besserer ML-Schätzer als $\theta^{(t)}$. Man muss hierbei jedoch beachten, dass dieses Verfahren zwar konvergiert, aber durchaus in einem lokalen Extremum stecken bleiben kann. Um letzteres zu vermeiden, widerholt man diese EM-Methode mit verschiedenen Startwerten für $\theta^{(0)}$.

Falls wir den MAP-Schätzer bestimmen wollen beginnen wir mit:

$$f(y \mid x, \theta) = \frac{f(x, y \mid \theta)}{f(x \mid \theta)} = \frac{f(x, y \mid \theta) \cdot f_0(\theta)}{f(x \mid \theta) \cdot f_0(\theta)}.$$

und erhalten dann

$$\log(f(x \mid \theta) \cdot f_0(\theta)) = \log(f(x, y \mid \theta)) - \log(f(y \mid x, \theta)) + \log(f_0(\theta)),$$

wobei auch hier $f_0(\theta)$ die Dichte des Priors ist. In diesem Fall definieren wir Q' wie folgt:

$$Q'(\theta \mid \theta^{(t)}) := \int f(y \mid x, \theta^{(t)}) \cdot \log(f(x, y \mid \theta))\, dy + \log(f_0(\theta)).$$

Dann gilt wiederum

$$\log(f(x \mid \theta) \cdot f_0(\theta)) - \log(f(x \mid \theta^{(t)}) \cdot f_0(\theta^{(t)})) \geq Q'(\theta \mid \theta^{(t)}) - Q'(\theta^{(t)} \mid \theta^{(t)}).$$

Letztes ist ebenfalls wieder nichtnegativ, da $\theta^{(t)} \in \Theta$. Wir wählen also $\theta^{(t+1)}$ wieder wie folgt:

$$\theta^{(t+1)} := \operatorname{argmax} \left\{ Q'(\theta \mid \theta^{(t)}) \ : \ \theta \in \Theta \right\}.$$

7.4.3 EM-Algorithmus

Damit können wir den EM-Algorithmus nun konkret, wie in Abbildung 7.5 angeben. Die Iteration wird hierbei solange durchgeführt bis wir hinreichend nahe an dem gefundenen Extremum sind (also beispielsweise $L(\theta^{(t)}; x) \approx L(\theta^{(t-1)}; x)$ gilt). Hierbei ist zu beachten, dass sich der bedingte Erwartungswert $Q(\theta \mid \theta^{(t)})$ nicht unbedingt leicht berechnen lassen muss (und meist auch gar nicht durch einen einfachen Ausdruck angeben lässt). Wir werden später darauf zurückkommen, wie wir Integrale von Dichten approximieren können.

EM (data x)

begin
  choose $\theta^{(0)}$ appropriately;
  for $(t := 0;\ (t = 0) \ ||\ (L(\theta^{(t)}; x) \not\approx L(\theta^{(t-1)}; x));\ t{+}{+})$ **do**
    compute $Q(\theta \mid \theta^{(t)})$; `/* E-Step */`
    let $\theta^{(t+1)} := \operatorname{argmax} \left\{ Q(\theta \mid \theta^{(t)}) \ : \ \theta \in \Theta \right\}$; `/* M-Step */`
end

Abbildung 7.5: Algorithmus: EM-Algorithmus

7.5 Markov-Ketten

Wir wiederholen in diesem Abschnitt aus der Stochastik bzw. der Wahrscheinlichkeitstheorie kurz den Begriff einer Markov-Kette, den wir im Folgenden noch oft benötigen.

7.5.1 Grundlegende Definitionen

Zuerst definieren wir formal eine Markov-Kette.

Definition 7.17 *Sei Q eine endliche Menge. Eine unendliche Folge von Zufallsvariablen $(X_i)_{i\in\mathbb{N}_0}$ mit Wertebereich Q heißt* Markov-Kette k-ter Ordnung *auf Q, wenn für alle $n \in \mathbb{N}_0$ und alle $q_n \in Q$ gilt:*

$$\begin{aligned}\mathrm{Ws}[X_n = q_n \mid X_0 = q_0, \ldots, X_{n-1} = q_{n-1}] \\ = \ \mathrm{Ws}[X_n = q_n \mid X_{n-k} = q_{n-k}, \ldots, X_{n-1} = q_{n-1}].\end{aligned}$$

Eine Markov-Kette erster Ordnung wird auch kurz als Markov-Kette *bezeichnet.*

Je nach Anwendung definieren wir die unendliche Folge von Zufallsvariablen mal als $(X_i)_{i\in\mathbb{N}_0}$ (wie oben) und mal als $(X_i)_{i\in\mathbb{N}}$ (also $X_1, \ldots$).

Markov-Ketten erster Ordnung sind *gedächtnislos*, da die Wahrscheinlichkeit des nächsten Zustandes nur vom aktuellen und nicht von den vergangenen Zuständen abhängt. Formal bedeutet dies, dass die folgende Beziehung gilt, die oft auch als *Markov-Eigenschaft* bezeichnet wird:

$$\mathrm{Ws}[X_i = q_i \mid X_0 = q_0, \ldots, X_{i-1} = q_{i-1}] = \mathrm{Ws}[X_i = q_i \mid X_{i-1} = q_{i-1}].$$

Man kann sich überlegen, dass auch Random-Walks eine Markov-Kette darstellen. Allerdings war dort die Zustandsmenge $Q = \mathbb{Z}$ eine unendliche Menge. Es ist auch möglich Markov-Ketten auf unendlichen Zustandsmengen zu definieren, was wir aber hier nicht tun wollen.

Bei Markov-Ketten k-ter Ordnung hängt die Übergangswahrscheinlichkeit von den letzten k eingenommenen Zuständen ab. Wie man sich leicht überlegt, lassen sich solche Markov-Ketten k-ter Ordnung durch Markov-Ketten erster Ordnung simulieren. Wir wählen als Zustandsmenge nur Q^k, wobei dann ein Zustand der neuen Markov-Kette erster Ordnung ein k-Tupel von Zuständen der Markov-Kette k-ter Ordnung ist, die dann die k zuletzt besuchten Zustände speichern. Die Details dieser Simulation seien dem Leser zur Übung überlassen.

Kommen wir jetzt noch zum Begriff der Zeithomogenität bei Markov-Ketten.

Definition 7.18 *Sei $(X_i)_{i\in\mathbb{N}_0}$ eine Markov-Kette auf Q. Hängt die Wahrscheinlichkeit* $\mathrm{Ws}[X_i = q \mid X_{i-1} = q']$ *nur von den Zuständen $q, q' \in Q$ ab, so nennt man die Markov-Kette* zeithomogen.

Bevor wir zur Definition von Markov-Modellen kommen, wiederholen wir noch einmal die Definition einer stochastischen Matrix.

Definition 7.19 *Ein $n \times m$-Matrix M heißt* stochastisch*, wenn $M_{i,j} \in [0,1]$ für alle $i \in [1:n]$ und $j \in [1:m]$ sowie $\sum_{j=1}^{m} M_{i,j} = 1$ für alle $i \in [1:n]$. Ein Vektor $x = (x_1, \ldots, x_n)$ (Zeilen- oder Spaltenvektor) heißt* stochastisch*, wenn $x_i \in [0,1]$ und $\sum_{i=1}^{n} x_i = 1$ gilt.*

Damit können wir die folgende Definition angeben.

Definition 7.20 *Eine* Markov-Modell *ist ein Tripel $M = (Q, P, \pi)$, wobei*

- *$Q = \{q_1, \ldots, q_n\}$ eine endliche Menge von Zuständen ist;*
- *$P = (p_{i,j})_{(i,j)\in[1:n]^2}$ eine stochastische $n \times n$-Matrix der so genannten Zustandsübergangswahrscheinlichkeiten ist;*
- *$\pi = (\pi_1, \ldots, \pi_n)$ ein stochastischer Vektor der so genannten Anfangswahrscheinlichkeiten ist.*

Eine Markov-Kette ist also nichts anderes als eine Menge von Zuständen zwischen denen man in jedem (diskreten) Zeitschritt mit einer gewissen Wahrscheinlichkeit, die vom momentanen Zustand abhängt, in einen anderen Zustand wechselt.

Die Anfangswahrscheinlichkeiten gibt hierbei an, mit welcher Wahrscheinlichkeit π_i wir uns zu Beginn im Zustand $q_i \in Q$ befinden. Die Zustandsübergangswahrscheinlichkeiten $p_{i,j}$ geben an, mit welcher Wahrscheinlichkeit wir vom Zustand q_i in den Zustand q_j wechseln. Damit muss die Summe der Wahrscheinlichkeiten vom Zustand q_i aus 1 sein und daher fordern wir, dass P eine stochastische Matrix sein muss.

Definition 7.21 *Die Markov-Kette $(X_i)_{i\in\mathbb{N}_0}$ ist durch ein Markov-Modell (Q, P, π) induziert, wenn gilt:*

a) $\mathrm{Ws}[X_0 = q_i] = \pi_i$*;*

b) $\mathrm{Ws}[X_i = q \mid X_{i-1} = q'] = p_{q',q}$.

Solche induzierten Markov-Ketten sind zeithomogen, da die Wahrscheinlichkeit eines Zustandswechsels nur von den Zuständen, aber nicht von den betrachteten Zeitpunkten abhängt.

Betrachten wir dazu ein einfaches Beispiel, nämlich eine simple Modellierung des Wetters, wie sie in Abbildung 7.6 illustriert ist. Wir besitzen zwei Zustände, nämlich schönes Wetter und schlechtes Wetter. Die Übergangswahrscheinlichkeiten geben

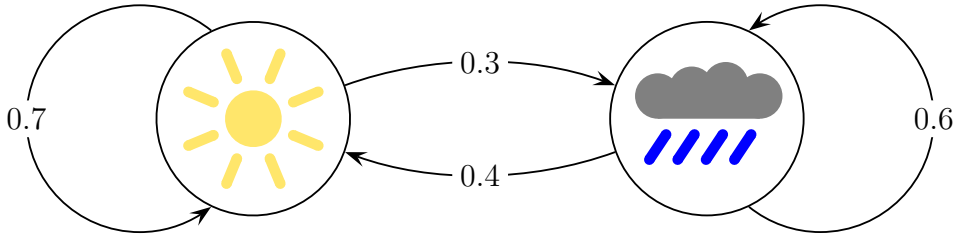


Abbildung 7.6: Beispiel: Einfache Markov-Kette für das Wetter

dabei an, mit welcher Wahrscheinlichkeit man beispielsweise vom schönen Wetter aus wechseln kann. Mit Wahrscheinlichkeit 0.7 bleibt es schön und mit Wahrscheinlichkeit 0.3 wird es schlecht, sofern momentan schönes Wetter herrscht. Umgekehrt wird es mit Wahrscheinlichkeit 0.4 schön und mit Wahrscheinlichkeit 0.6 bleibt es schlecht, sofern das Wetter gerade schlecht ist.

Mit $(x_1, \ldots, x_\ell) \in Q^\ell$ bezeichnen wir im Folgenden die Folge von Zuständen, die eine Markov-Kette zu den (diskreten) Zeitpunkten $i \in [1 : \ell]$ durchläuft. Dabei interpretieren wir gemäß der Definition einer Markov-Kette

$$\forall i \in [2 : \ell] : \ \forall r, s \in Q : \ \mathrm{Ws}[X_i = s \mid X_{i-1} = r] = p_{r,s}$$

und

$$\forall r \in Q : \ \mathrm{Ws}[X_1 = r] = \pi_r.$$

Wir bemerken hier insbesondere wieder die Zeithomogenität an, d.h. dass die Wahrscheinlichkeiten der Zustandsübergänge nur von den Zuständen und nicht von den Zeitpunkten i mit $i \in [1 : \ell]$ selbst abhängen.

Im Folgenden werden wir oft Sequenzen betrachten. Dann entsprechen die Zeitpunkte i den Positionen i innerhalb der Sequenz, d.h. wir stellen uns den Aufbau einer Sequenz als den Prozess einer Markov-Kette vor. Mit dieser Modellierungen bekommen wir eine Abhängigkeiten der Wahrscheinlichkeiten der Zeichen innerhalb einer Sequenz von ihrer Nachbarschaft.

7.5.2 Wahrscheinlichkeiten von Pfaden

Wir wollen jetzt die Wahrscheinlichkeit von bestimmten Pfaden durch eine Markov-Kette bestimmen. Hierbei benutzen wir oft die folgenden Abkürzung für eine gegebene Folge $X = (X_1, \ldots, X_\ell)$ von Zuständen.

$$\mathrm{Ws}[X = x] := \mathrm{Ws}[X = (x_1, \ldots, x_\ell)] := \mathrm{Ws}[X_1 = x_1 \wedge \cdots \wedge X_\ell = x_\ell].$$

Sei $x = (x_1, \ldots, x_\ell)$ eine Folge von Zuständen. Die Wahrscheinlichkeit, dass diese Zustandsfolge durchlaufen wird, berechnet sich dann zu

$$\mathrm{Ws}[X = x] = \pi_{x_1} \cdot \prod_{i=2}^{\ell} p_{x_{i-1},x_i}.$$

Die Anfangswahrscheinlichkeiten π kann auch durch einen neuen Startzustand q_0, der zum Zeitpunkt X_0 eingenommen wird, mit einer Erweiterung der Matrix P der Zustandsübergangswahrscheinlichkeiten eliminiert werden. Wir setzen dann

$$p_{0,j} := \mathrm{Ws}[X_1 = q_j \mid X_0 = q_0] = \pi_{x_j}$$

für $j \in [1 : n]$. Weiter definieren wir noch für $i \in [0 : n]$:

$$p_{i,0} := 0.$$

Damit vereinfacht sich die Formel zur Berechnung der Wahrscheinlichkeiten für eine Folge $x = (x_1, \ldots, x_\ell)$ von Zuständen zu:

$$\mathrm{Ws}[X = x] = \prod_{i=1}^{\ell} p_{x_{i-1},x_i}.$$

7.5.3 Beispiel: CpG-Inseln

Im Folgenden wollen wir für ein Beispiel mit biologischem Hintergrund eine Markov-Kette angeben. Im Genom kommt die Basenabfolge CG sehr selten vor (wir schreiben hierfür CpG, damit es nicht mit dem Basenpaar CG verwechselt werden kann, wobei p für den Phosphatrest zwischen den entsprechenden Nukleosiden steht). Dies hat den Hintergrund, dass die beiden Basen in dieser Abfolge chemischen Reaktionen unterworfen sind, die für eine Mutation in der DNS sorgen würde. Man beachte hier, dass mit CG im komplementären Strang der DNS ebenfalls die Basenabfolge CG vorkommt.

Es gibt jedoch Bereiche, wo diese Abfolge überaus häufig auftritt. Es hat sich herausgestellt, dass solche Bereiche, in denen die Folge CpG überdurchschnittlich häufig vorkommt, oft Promotoren enthält. In diesen Bereichen wird die chemische Reaktion von CpG-Paaren in der Regel verhindert. Daher kann man die Kenntnis von Bereichen mit vielen CpG-Teilsequenzen als Bereiche für Kandidaten für Promotoren betrachten, die dann natürlich für das Auffinden von Genen im Genom besonders wichtig sind. Wir formalisieren die Problemstellung wie folgt.

Identifikation von CpG-Inseln

Eingabe: Eine kurze DNS-Sequenz $x = (x_1, \ldots, x_\ell) \in \{A, C, G, T\}^\ell$.
Gesucht: Befindet sich x innerhalb einer CpG-Insel.

Wir versuchen jetzt mit Hilfe von Markov-Ketten solche CpG-Inseln zu identifizieren. Dazu sind in Abbildung 7.7 die Wahrscheinlichkeiten angegeben, mit denen im Genom auf eine Base X eine Base Y folgt. Dabei ist P^+ die Matrix der Wahrscheinlichkeiten innerhalb einer CpG-Insel und P^- die außerhalb einer solchen.

P^+	A	C	G	T
A	0.18	0.27	0.43	0.12
C	0.17	0.37	0.27	0.19
G	0.16	0.34	0.37	0.13
T	0.08	0.36	0.38	0.18

P^-	A	C	G	T
A	0.30	0.21	0.28	0.21
C	0.32	0.30	0.08	0.30
G	0.25	0.32	0.30	0.21
T	0.18	0.24	0.29	0.29

Abbildung 7.7: Skizze: Zustandsübergangswahrscheinlichkeiten innerhalb und außerhalb von CpG-Inseln

Wir modellieren jeweils ein Markov-Modell für die Bereiche innerhalb bzw. außerhalb der CpG-Inseln. Mit $M^+ = (Q, P^+, \pi)$ bzw. $M^- = (Q, P^-, \pi)$ bezeichnen wir zwei Markov-Ketten: eine für Sequenzen innerhalb (M^+) und eine außerhalb der CpG-Inseln (M^-). Dabei besteht die Zustandsmenge $Q = \{A, C, G, T\}$ gerade jeweils aus den vier Basen und π ist die Anfangswahrscheinlichkeiten, die man aus den relativen Häufigkeiten der Basen im gesamten Genom ermittelt hat.

Wir berechnen dann die Wahrscheinlichkeit des Pfades X innerhalb der beiden Modelle:

$$\begin{aligned} \mathrm{Ws}[X = x \mid M^+] &= \pi(x_1) \cdot \prod_{i=2}^{\ell} p^+_{x_{i-1},x_i}, \\ \mathrm{Ws}[X = x \mid M^-] &= \pi(x_1) \cdot \prod_{i=2}^{\ell} p^-_{x_{i-1},x_i}. \end{aligned}$$

Wie bereits erwähnt, nehmen wir hier für die Anfangswahrscheinlichkeiten an, dass diese für beide Modell gleich sind.

Für die Entscheidung betrachten wir dann den Quotienten der entsprechenden Wahrscheinlichkeiten, also die Likelihood-Ratio:

$$\frac{\mathrm{Ws}[X = x \mid M^+]}{\mathrm{Ws}[X = x \mid M^-]} = \frac{\pi(x_1) \cdot \prod_{i=2}^{\ell} p^+_{x_{i-1},x_i}}{\pi(x_1) \cdot \prod_{i=2}^{\ell} p^-_{x_{i-1},x_i}} = \prod_{i=2}^{\ell} \frac{p^+_{x_{i-1},x_i}}{p^-_{x_{i-1},x_i}}.$$

Wir können dann einen einfachen Hypothesen-Test, also den Likelihood-Ratio-Test anwenden, um die Null- von der Alternativ-Hypothese zu trennen.

Statt einen echten Hypothesen-Test durchzuführen, wird die Likelihood-Ratio auch als Score verwendet. Da es sich zum einen leichter mit Summen als mit Produkten rechnen lässt und zum anderen die numerischen Stabilität der Ergebnisse erhöht, definieren wir den Score als den Logarithmus des obigen Quotienten:

$$\text{Score}(x) := \log\left(\frac{\text{Ws}[X = x \mid M^+]}{\text{Ws}[X = x \mid M^-]}\right) = \sum_{i=2}^{\ell} \left(\log(p^+_{x_{i-1},x_i}) - \log(p^-_{x_{i-1},x_i})\right).$$

Positive Scores deuten dann auf eine CpG-Inseln hin, während negative Scores Bereiche außerhalb einer CpG-Insel charakterisieren.

In der Regel wollen wir nicht für eine kurze Sequenz feststellen, ob sie sich innerhalb oder außerhalb einer CpG-Insel befindet, sondern wir wollen herausfinden, wo sich im Genom solche CpG-Inseln befinden. Daher betrachten wir die folgende Problemstellung, die diesem Ziel Rechnung trägt.

Lokalisierung von CpG-Inseln

Eingabe: Eine lange DNS-Sequenz $x = (x_1, \ldots, x_\ell) \in \{A, C, G, T\}^\ell$.
Gesucht: Teilsequenzen in x, die CpG-Inseln bilden.

Ein naiver Ansatz würde für alle kurzen Sequenzen der Länge k mit $k \in [100 : 1000]$ den vorherigen Algorithmus anwenden. Problematisch ist hier weniger der Rechenaufwand, der sich durch geeignete Tricks wie beim Karp-Rabin-Algorithmus in Grenzen halten lässt, sondern die Festlegung von k.

Wählt man k zu groß, so wird man CpG-Inseln sicherlich nur schwerlich hinreichend sicher identifizieren können. Wählt man k zu klein, so können die Sequenzen so kurz sein, dass man keine signifikanten Unterscheidungen mehr erhält. Eine Alternative dazu besteht einerseits in einem festerlosen Ansatz, indem man auf die Scores einen Algorithmus für das Maximum Scoring Subsequence Problem bzw. eine geeignete Variante hierzu anwendet. Eine weitere Alternative andererseits werden wir im nächsten Kapitel kennen lernen, in dem wir beide Modelle in einem Modell zu vereinigen versuchen.

7.5.4 Fundamentale Eigenschaften von Markov-Ketten

Wir halten jetzt noch ein paar fundamentale Eigenschaften von Markov-Ketten fest. Wir sind insbesondere an der so genannten stationären Verteilung einer Markov-Kette interessiert.

Definition 7.22 *Sei (Q, P, π) ein Markov-Modell. Ein stochastischer Vektor ρ heißt* stationäre Verteilung *der induzierten Markov-Kette, wenn $\rho \cdot P = \rho$.*

Wir wollen uns nun überlegen, wann eine Markov-Kette eine stationäre Verteilung besitzt. Zuerst überlegen wir uns, wie man aus einer Verteilung $\rho^{(i)}$ zum Zeitpunkt i die Verteilung $\rho^{(i+1)}$ zum Zeitpunkt $i+1$ berechnet. Zuerst halten wir fest, dass $\rho^{(0)} = \pi$ ist. Nach Definition einer Markov-Kette gilt für $Q = \{q_1, \ldots, q_m\}$:

$$\mathrm{Ws}[X_{i+1} = q_k] = \sum_{j=1}^{m} \mathrm{Ws}[X_i = q_j] \cdot \mathrm{Ws}[X_{i+1} = q_k \mid X_i = q_j].$$

Das lässt sich für die k-te Komponente auch schreiben als

$$\rho_k^{(i+1)} = \sum_{j=1}^{m} \rho_j^{(i)} \cdot p_{j,k}.$$

In der Matrix-Schreibweise gilt dann

$$\rho^{(i+1)} = \rho^{(i)} \cdot P.$$

Wegen der Zeithomogenität gilt dann auch

$$\rho^{(i+k)} = \rho^{(i)} \cdot P^k.$$

Notation 7.23 *Sei P eine Matrix, die Einträge von P^k werden mit $p_{i,j}^{(k)}$ bezeichnet.*

Man überlegt sich leicht, dass P^k eine stochastische Matrix ist, wenn P bereits stochastisch ist. Dazu zeigen wir, dass für zwei stochastische $n \times n$-Matrizen P, P' auch $P \cdot P'$ stochastisch ist. Für alle $i \in [1:n]$ gilt:

$$\sum_{j=1}^{n} (P \cdot P')_{i,j} = \sum_{j=1}^{n} \sum_{k=1}^{n} p_{i,k} \cdot p'_{k,j} = \sum_{k=1}^{n} p_{i,k} \sum_{j=1}^{n} p'_{k,j} = \sum_{k=1}^{n} p_{i,k} = 1.$$

Wir zeigen zunächst, dass jede endliche zeithomogene Markov-Kette zumindest eine stationäre Verteilung besitzt.

Lemma 7.24 *Jede endliche zeithomogene Markov-Kette $(X_i)_{i \in \mathbb{N}_0}$ besitzt eine stationäre Verteilung ρ.*

Für unendliche zeithomogene Markov-Ketten gilt die Aussage des Lemmas nicht. Man möge sich eine Gegenbeispiel überlegen.

Beweis: Sei P die Matrix der Zustandsübergangswahrscheinlichkeiten, d.h.

$$p_{i,j} = \mathrm{Ws}[X_t = q_j \mid X_{t-1} = q_i]$$

für alle $t \in \mathbb{N}$. Sei α eine beliebige Verteilung auf Q. Wir definieren dann

$$\alpha^{(n)} = \frac{1}{n} \sum_{k=0}^{n-1} \alpha \cdot P^k.$$

Offensichtlich ist $\alpha^{(1)} = \alpha$. Zuerst bemerkt man, dass $\alpha \cdot P^k$ für $k \in \mathbb{N}_0$ ebenfalls eine Wahrscheinlichkeitsverteilung auf Q ist:

$$\sum_{j=1}^{m} \left(\alpha \cdot P^k\right)_j = \sum_{j=1}^{m} \sum_{i=1}^{m} \alpha_i \cdot p_{i,j}^{(k)} = \sum_{i=1}^{m} \alpha_i \sum_{j=1}^{m} p_{i,j}^{(k)} = \sum_{i=1}^{m} \alpha_i = 1.$$

Somit sind auch $\alpha^{(n)}$ für alle $n \in \mathbb{N}$ Wahrscheinlichkeitsverteilungen auf Q.

Also ist $(\alpha_i^{(n)})_{n\in\mathbb{N}}$ für jedes $i \in [1:m]$ eine beschränkte Folge. Nach dem Satz von Bolzano-Weierstraß gibt es eine unendliche Teilfolge $(\alpha^{(j'_n)})_{n\in\mathbb{N}}$, so dass

$$\lim_{n\to\infty} \alpha_1^{(j'_n)} = \rho_1$$

existiert. Aus dieser Teilfolge kann man nach dem Satz von Bolzano-Weierstraß wiederum eine Teilfolge $(\alpha^{(j''_n)})_{n\in\mathbb{N}}$ von $(\alpha^{(j'_n)})_{n\in\mathbb{N}}$ finden, so dass

$$\lim_{n\to\infty} \alpha_2^{(j''_n)} = \rho_2$$

existiert. Nach m-maligen Anwenden des Satzes von Bolzano-Weierstraß folgt, dass es eine Teilfolge $(\alpha^{(j_n)})_{n\in\mathbb{N}}$ gibt, für die gilt:

$$\lim_{n\to\infty} \alpha^{(j_n)} = \rho = (\rho_1, \ldots, \rho_m).$$

Dies folgt aus der Tatsache, dass eine Teilfolge einer konvergenten Folge denselben Grenzwert wie die konvergente Folge selbst besitzt.

Wir zeigen jetzt, dass ρ auch eine Wahrscheinlichkeitsverteilung auf Q ist:

$$\sum_{i=1}^{m} \rho_i = \sum_{i=1}^{m} \lim_{n\to\infty} \alpha_i^{(j_n)} = \lim_{n\to\infty} \sum_{i=1}^{m} \alpha_i^{(j_n)} = \lim_{n\to\infty} 1 = 1.$$

Nach Definition von $\alpha^{(n)}$ gilt:

$$\begin{aligned}
\alpha^{(n+1)} &= \frac{1}{n+1} \sum_{k=0}^{n} \alpha \cdot P^k \\
&= \frac{1}{n+1} \cdot \alpha \cdot P^n + \frac{1}{n+1} \cdot \frac{n}{n} \sum_{k=0}^{n-1} \alpha \cdot P^k \\
&= \frac{n}{n+1} \cdot \alpha^{(n)} + \frac{1}{n+1} \cdot \alpha \cdot P^n
\end{aligned}$$

sowie

$$\begin{aligned}\alpha^{(n+1)} &= \frac{1}{n+1}\sum_{k=0}^{n}\alpha\cdot P^k\\ &= \frac{1}{n+1}\cdot\alpha+\frac{1}{n+1}\cdot\frac{n}{n}\sum_{k=1}^{n}\alpha\cdot P^{k-1}\cdot P\\ &= \frac{n}{n+1}\cdot\alpha^{(n)}\cdot P+\frac{1}{n+1}\cdot\alpha.\end{aligned}$$

Aus der ersten Beziehung folgt

$$\lim_{n\to\infty}\alpha^{(j_n+1)}=\lim_{n\to\infty}\left(\frac{j_n}{j_n+1}\cdot\alpha^{(j_n)}+\frac{1}{j_n+1}\cdot\alpha\cdot P^{j_n}\right)=\lim_{n\to\infty}\alpha^{(j_n)}=\rho.$$

Mit der zweiten Beziehung folgt dann

$$\lim_{n\to\infty}\alpha^{(j_n+1)}=\lim_{n\to\infty}\left(\frac{j_n}{j_n+1}\cdot\alpha^{(j_n)}\cdot P+\frac{1}{j_n+1}\cdot\alpha\right)=\rho\cdot P.$$

Damit folgt die Behauptung. ■

16.01.20

Lemma 7.25 *Sei $(X_i)_{i\in\mathbb{N}_0}$ eine durch (Q,P,π) induzierte Markov-Kette. Wenn $\lim_{n\to\infty}\pi\cdot P^n=\rho$ existiert, dann ist ρ eine stationäre Verteilung.*

Beweis: Wir zeigen zunächst, dass ρ eine Verteilung ist. Offensichtlich gilt $\rho_q\geq 0$ für alle $q\in Q$. Weiter gilt

$$\begin{aligned}\sum_{q\in Q}\rho_q &= \sum_{q\in Q}\lim_{n\to\infty}(\pi P^n)_q\\ &= \lim_{n\to\infty}\sum_{q\in Q}(\pi P^n)_q\\ &= \lim_{n\to\infty}\sum_{q\in Q}\sum_{q'\in Q}\pi_{q'}p^{(n)}_{q',q}\\ &= \lim_{n\to\infty}\sum_{q'\in Q}\pi_{q'}\sum_{q\in Q}p^{(n)}_{q',q}\\ &\quad\text{da mit } P \text{ auch } P^n \text{ eine stochastische Matrix ist}\\ &= \lim_{n\to\infty}\sum_{q'\in Q}\pi_{q'}\\ &\quad\text{da } \pi \text{ ein stochastischer Vektor ist}\\ &= \lim_{n\to\infty}1\\ &= 1.\end{aligned}$$

Weiter gilt

$$\begin{aligned}\rho &= \lim_{n\to\infty} \pi \cdot P^n \\ &= \lim_{n\to\infty} (\pi \cdot P^{n-1}) \cdot P \\ &= \lim_{n\to\infty} (\pi \cdot P^n) \cdot P \\ &= \rho \cdot P\end{aligned}$$

und damit die Behauptung. ∎

Kommen wir nun zur Definition des Begriffs der Irreduzibilität.

Definition 7.26 *Sei (Q, P, π) ein Markov-Modell. Die dadurch induzierte Markov-Kette heißt* irreduzibel*, wenn es für alle Paare $(q, q') \in Q^2$ ein $k \in \mathbb{N}$ gibt, so dass $p^{(k)}_{q,q'} > 0$.*

Irreduzible Markov-Ketten haben die schöne Eigenschaft, dass sie eine eindeutige stationäre Verteilung besitzen. Bevor wir dazu kommen, benötigen wir noch den Begriff der Übergangszeit.

Definition 7.27 *Sei (Q, P, π) ein Markov-Modell und $(X_i)_{i\in\mathbb{N}_0}$ eine dadurch induzierte Markov-Kette. Die Zufallsvariable*

$$H_{q,q'} = \min\{k \in \mathbb{N} \,:\, X_0 = q \wedge X_k = q'\}\,.$$

heißt Übergangszeit *(engl.* hitting time*) von q nach q' (hierbei gilt $\min\emptyset = +\infty$). Weiter ist $h_{q,q'} = \mathbb{E}[H_{q,q'}]$ die* erwartete Übergangszeit*.*

Man beachte, dass $H_{q,q'}$ eine Zufallsvariable ist und somit keinen konkreten Wert darstellt, man kann nur die Wahrscheinlichkeit für spezielle Werte ermitteln. Wir wollen dies an einem kleinen Beispiel deutlich machen und bemühen wieder den Wurf einer Münze, die mit Wahrscheinlichkeit $\frac{1}{2}$ Kopf zeigt. Für das Markov-Modell sei die Zustandsmenge die Menge der ganzen Zahlen, also ausnahmsweise sowie der Einfachheit halber eine unendliche, und die Anfangsverteilung sei auf 0 konzentriert. Der Zustand $z \in \mathbb{Z}$ repräsentiert dabei die Anzahl geworfener Köpfe minus der der geworfener Zahlen. Die dadurch induzierte Markov-Kette sei $(X_i)_{i\in\mathbb{N}_0}$.

Beachte zunächst, dass für das ℓ-malige Werfen einer Münze für ein gegebenes Ergebnis $(x_1, \ldots, x_\ell) \in \{K, Z\}^\ell$ gilt: $\mathrm{Ws}[(M_1, \ldots, M_{\ell 1}) = (x_1, \ldots, x_\ell)] = \frac{1}{2^\ell}$ (bei einer geeigneten Wahrscheinlichkeitsverteilung, hier basierend auf der Gleichverteilung).

Die Zufallsvariable $H_{0,1}$ ist dann nicht 1, obwohl man mit einem Wurf von Kopf vom Zustand 0 in den Zustand 1 gelangen kann. Vielmehr kann man nur $\text{Ws}[H_{0,1} = 1] = \frac{1}{2}$ ermitteln, da man nur mit einem (dem ersten Münzwurf) mit Wahrscheinlichkeit $\frac{1}{2}$ vom Zustand 0 nach 1 gelangt, d.h. nur Sequenzen die mit Kopf beginnen, können in einem Schritt den Zustand 1 erreichen. Die Minimumsbildung in der Definition der Übergangszeit besagt nur, dass man für eine Folge von Münzwürfen nur den ersten Besuch vom Zustand 0 berücksichtigt.

Es gilt weiter $\text{Ws}[H_{0,1} = 3] = \frac{1}{8}$, da nur die Folgen, die mit ZKK beginnen, erstmalig den Zustands 1 nach 3 Schritten erreichen. Ferner gilt $\text{Ws}[H_{01} = 5] = \frac{2}{32} = \frac{1}{16}$, da es nur die Sequenzen mit den beiden folgenden Präfixen zum erstmaligen Erreichen des Zustands 1 nach genau 5 Schritten gibt: $(ZZKKK)$ und $(ZKZKK)$. Es gilt insbesondere auch $\text{Ws}[H_{0,1} = 2n] = 0$ für alle $n \in \mathbb{N}_0$, da man nur mit einer ungeraden Anzahl von Münzwürfen den Zustand 1 von 0 aus erreichen kann (dem Markov-Modell liegt ein bipartiter Graph zugrunde).

Damit kommen wir zu dem bereits angekündigten Theorem, dass wir hier nicht beweisen wollen.

Theorem 7.28 *Sei (Q, P, π) ein Markov-Modell und $(X_i)_{i\in\mathbb{N}_0}$ eine dadurch induzierte irreduzible Markov-Kette. Dann besitzt $(X_i)_{i\in\mathbb{N}_0}$ eine eindeutige stationäre Verteilung $\rho = (1/h_{11}, \ldots, 1/h_{mm})$.*

Wir müssten hier eigentlich nur noch zeigen, dass die stationäre Verteilung eindeutig ist und die Beziehung zur erwarteten Übergangszeit erfüllt ist. Für den Beweis verweisen wir auf die entsprechenden Vorlesungen über Stochastik. Dort findet man auch einfache und effiziente Algorithmen zur Bestimmung der für die stationäre Verteilung benötigten erwarteten Übergangszeiten.

Eine Markov-Kette, die nicht irreduzibel ist, kann hingegen viele stationäre Verteilung besitzen. Betrachte dazu das Markov-Modell (Q, I, π), wobei I die Einheitsmatrix ist. Dann ist jede Verteilung π stationär.

Leider gibt es irreduzible Markov-Modelle, die stationäre Verteilungen haben, aber nicht dagegen konvergieren. Betrachte die Matrix $P = \left(\begin{smallmatrix}01\\10\end{smallmatrix}\right)$, in der man für beliebige $q, q' \in Q$ nach einem oder zwei Schritten von q nach q' gelangt (also ist P irreduzibel). Man kann leicht nachrechnen, dass $(0.5, 0.5)$ eine stationäre Verteilung ist, aber die Markov-Kette gestartet in $(1, 0)$ immer zwischen den Verteilungen $(1, 0)$ und $(0, 1)$ alterniert. Wir wollen uns daher im Folgenden damit beschäftigen, wann eine Markov-Kette von einer beliebigen Anfangsverteilung in die stationäre Verteilung konvergiert. Dazu benötigen wir den Begriff der Aperiodizität.

Definition 7.29 *Sei (Q, P, π) ein Markov-Modell. Die* Periode d_q *eines Zustands $q \in Q$ ist definiert als*

$$d_q := \text{ggT}\left\{ k \in \mathbb{N} : \begin{array}{l} \exists (q_0, \ldots, q_k) \in Q^{k+1} \quad \wedge \quad q_0 = q_k = q \\ \wedge \quad \forall i \in [0 : k-1] p_{q_i, q_{i+1}} > 0 \end{array} \right\}.$$

Ein Zustand mit Periode 1 heißt aperiodisch. *Eine durch (Q, P, π) induzierte Markov-Kette heißt* aperiodisch, *wenn alle Zustände $q \in Q$ aperiodisch sind.*

Ist $d_q > 1$, dann beträgt die Anzahl der Schritte, um von q nach q zurückzukehren, immer ein Vielfaches von d_q. Es liegt dann für diese Zustand also eine gewisse Periodizität vor.

Damit können wir nun den Begriff der Ergodizität einführen.

Definition 7.30 *Eine durch ein Markov-Modell induzierte Markov-Kette, die irreduzibel und aperiodisch ist, heißt* ergodisch.

Wie wir gleich sehen werden, sind wir insbesondere an ergodischen Markov-Ketten interessiert.

Theorem 7.31 *Sei $(X_i)_{i \in \mathbb{N}_0}$ eine durch das Markov-Modell (Q, P, π) induzierte ergodische Markov-Kette. Dann existiert*

$$\lim_{t \to \infty} \pi \cdot P^t = \rho$$

und ρ ist die eindeutige stationäre Verteilung der Markov-Kette.

Für den Beweis, den wir hier nicht führen wollen, ist nur noch zu zeigen, dass die Existenz des Grenzwertes aus der Aperiodizität folgt. Auch hier verweisen wir wieder auf die entsprechenden Vorlesungen bzw; Lehrbücher über Stochastik.

Kommen wir nach dieser qualitativen Aussage über die Konvergenz von ergodischen Markov-Ketten zu einer qualitativen Aussage über deren Konvergenzgeschwindigkeit. Der folgende Satz quantifiziert, wie schnell bzw. wie gut ein Markov-Modell von der Anfangsverteilung aus hinreichend nahe an die eindeutige stationäre Verteilung konvergiert.

Theorem 7.32 *Sei $(X_i)_{i\in\mathbb{N}_0}$ eine durch (Q, P, π) induzierte Markov-Kette. Wenn ρ eine stationäre Verteilung der Markov-Kette ist und wenn*

$$\nu := \min\left\{\frac{p_{q,q'}}{\rho_{q'}} \;:\; q, q' \in Q \wedge \rho_{q'} > 0\right\} > 0$$

ist, dann ist die Markov-Kette ergodisch. Für die Konvergenzgeschwindigkeit gilt dann für $\rho^{(t)} = \pi \cdot P^t$ mit $q \in Q$

$$|\rho_q - \rho_q^{(t)}| \leq (1-\nu)^t.$$

Ist Y eine reellwertige Zufallsvariable auf Q, dann gilt

$$|\,\mathbb{E}_\rho[Y] - \mathbb{E}_{\rho^{(t)}}[Y]|\;\leq (1-\nu)^t \cdot \max\left\{Y(q) - Y(q') \;:\; q, q' \in Q\right\}.$$

Beachte, dass die Aussage des Satze unabhängig von der Anfangsverteilung π ist.

Beweis: Wir beweisen hier nur die Ergebnisse über die Konvergenzgeschwindigkeit. Für den Beweis der Ergodizität verweisen wir auf einschlägige Lehrbücher. Für den einfachen Fall, dass $\rho_q > 0$ für alle $q \in Q$ gilt, ist mit $\nu > 0$ auch $p_{q,q'} > 0$ für alle $q, q' \in Q$. Somit ist das Markov-Modell offensichtlich irreduzibel und aperiodisch, also ergodisch.

Zuerst zeigen wir, dass $\nu \leq 1$ gilt. Für einen Widerspruchsbeweis nehmen wir an, dass $\nu > 1$. Dann müsste zumindest für alle $q' \in Q$ mit $\rho_{q'} > 0$ gelten, dass $\frac{p_{q,q'}}{\rho_{q'}} > 1$, d.h. $p_{q,q'} > \rho_{q'}$. Falls $\rho_{q'} = 0$, gilt zumindest $p_{q,q'} \geq \rho_{q'}$ Dann wäre aber (da nicht alle Komponenten von ρ Null sein können)

$$1 = \sum_{q'\in Q} p_{q,q'} > \sum_{q'\in Q} \rho_{q'} = 1,$$

was den gewünschten Widerspruch liefert.

Wir überlegen uns jetzt, was aus $\nu = 1$ folgt. Dann gilt $p_{q,q'} \geq \rho_{q'}$ für alle $q, q' \in Q$. Ware nun $p_{q'',q'} > \rho_{q'}$, dann wäre die Zeilesumme der Zeile q'' größer als 1 und P somit nicht stochastisch. Also hat im Falle $\nu = 1$ die Matrix P die Gestalt $p_{q,q'} = \rho_{q'}$. Dann ist aber $\rho^{(t)} = \rho$ für alle $0 \neq t \in \mathbb{N}$ und die Aussagen sind trivial.

Wir behaupten jetzt, dass sich $\rho_q^{(n)}$ wie folgt schreiben lässt:

$$\rho_q^{(n)} = [1 - (1-\nu)^n]\rho_q + (1-\nu)^n r_n(q),$$

wobei r_n ein Wahrscheinlichkeitsverteilung auf Q ist.

Diese Behauptung beweisen wir durch vollständige Induktion. Für den Induktionsanfang $n = 0$ gilt $\pi_q = \rho_q^{(0)} = r_0(q)$. Für den Induktionsschluss von n nach $n+1$ gilt:

$$\begin{aligned}
\rho_q^{(n+1)} &= \sum_{q' \in Q} \rho_{q'}^{(n)} p_{q',q} \\
&\quad \text{nach Induktionsvoraussetzung} \\
&= \sum_{q' \in Q} \Big([1-(1-\nu)^n]\rho_{q'} + (1-\nu)^n r_n(q') \Big) p_{q',q} \\
&= [1-(1-\nu)^n] \sum_{q' \in Q} \rho_{q'} p_{q',q} + (1-\nu)^n \sum_{q' \in Q} r_n(q') p_{q',q} \\
&\quad \text{da } \rho \text{ eine stationäre Verteilung ist} \\
&= [1-(1-\nu)^n]\rho_q + (1-\nu)^n \sum_{q' \in Q} r_n(q')[p_{q',q} - \nu\rho_q + \nu\rho_q] \\
&\quad \text{da } r_n \text{ eine Wahrscheinlichkeitsverteilung ist} \\
&= [1-(1-\nu)^n]\rho_q + (1-\nu)^n \nu\rho_q + (1-\nu)^n \sum_{q' \in Q} r_n(q')[p_{q',q} - \nu\rho_q] \\
&= [1-((1-\nu)^n - (1-\nu)^n\nu)]\rho_q + (1-\nu)^{n+1} \sum_{q' \in Q} r_n(q') \frac{p_{q',q} - \nu\rho_q}{1-\nu} \\
&= [1-(1-\nu)^{n+1}]\rho_q + (1-\nu)^{n+1} r_{n+1}(q).
\end{aligned}$$

Hierbei ist

$$r_{n+1}(q) := \sum_{q' \in Q} r_n(q') \frac{p_{q',q} - \nu\rho_q}{1-\nu}.$$

Es bleibt zu zeigen, dass r_{n+1} wiederum eine Wahrscheinlichkeitsverteilung ist, wenn r_n bereits eine war. Zunächst gilt

$$r_{n+1}(q) = \sum_{q' \in Q} r_n(q') \frac{p_{q',q} - \nu\rho_q}{1-\nu} \geq 0,$$

da zum einen $1-\nu > 0$, da $\nu < 1$. Zum anderen ist $p_{q',q} - \nu\rho_q \geq 0$, da $\frac{p_{q',q}}{\rho_q} \geq \nu$ nach Definition von ν. Weiterhin gilt:

$$\begin{aligned}
\sum_{q \in Q} r_{n+1}(q) &= \sum_{q \in Q} \sum_{q' \in Q} r_n(q') \frac{p_{q',q} - \nu\rho_q}{1-\nu} \\
&= \frac{1}{1-\nu} \sum_{q' \in Q} r_n(q') \sum_{q \in Q} (p_{q',q} - \nu\rho_q) \\
&= \frac{1}{1-\nu} \sum_{q' \in Q} r_n(q') \left(\sum_{q \in Q} p_{q',q} - \nu \sum_{q \in Q} \rho_q \right)
\end{aligned}$$

$$\begin{aligned} &= \frac{1}{1-\nu} \sum_{q' \in Q} r_n(q')(1-\nu) \\ &= \sum_{q' \in Q} r_n(q') \\ &= 1. \end{aligned}$$

Nun zeigen wir die Konvergenz von $\rho_q^{(n)}$:

$$\begin{aligned} |\rho_q - \rho_q^{(n)}| &= |\rho_q - [1-(1-\nu)^n]\rho_q - (1-\nu)^n r_n(q)| \\ &= |(1-\nu)^n \rho_q - (1-\nu)^n r_n(q)| \\ &= (1-\nu)^n \cdot |\rho_q - r_n(q)| \\ &\quad \text{da } \rho(q), r_n(q) \in [0,1] \\ &\leq (1-\nu)^n. \end{aligned}$$

Weiterhin gilt

$$\begin{aligned} &|\mathbb{E}_\rho[Y] - \mathbb{E}_{\rho^{(n)}}[Y]| \\ &\quad = \left| \sum_{q' \in Q} \rho_{q'} \cdot Y(q') - \sum_{q' \in Q} \rho_{q'}^{(n)} \cdot Y(q') \right| \\ &\quad = \left| \sum_{q' \in Q} \rho_{q'} \cdot Y(q') - \sum_{q' \in Q} [(1-(1-\nu)^n)\rho_{q'} + (1-\nu)^n r_n(q')] Y(q') \right| \\ &\quad = \left| \sum_{q' \in Q} [(1-\nu)^n \rho_{q'} - (1-\nu)^n r_n(q')] Y(q') \right| \\ &\quad = (1-\nu)^n \left| \sum_{q' \in Q} \rho_{q'} \cdot Y(q') - \sum_{q' \in Q} r_n(q') \cdot Y(q') \right| \\ &\quad \leq (1-\nu)^n \cdot \max\{|Y(q') - Y(q'')| \,:\, q', q'' \in Q\}. \end{aligned}$$

Damit ist der Beweis abgeschlossen. ■

Eine andere Überlegung ist, dass eine stationäre Verteilung ρ nichts anderes als ein linker Eigenvektor zum Eigenwert $\lambda = 1$ ist, da $\rho \cdot P = \lambda \cdot \rho$. Im Gegensatz zur linearen Algebra, wo in der Regel nur rechte Eigenwerte und rechte Eigenvektoren betrachtet werden, betrachtet man hier linke. Da aber aus $A \cdot x = \lambda x$ durch Transposition auch $x^T \cdot A^T = \lambda x^T$ folgt, kann man die übliche Theorie der Eigenwerte aus der linearen Algebra auch hier anwenden. Man überlege sich, dass 1 auch ein rechter Eigenwert von P ist und wie der zugehörige Eigenvektor aussieht.

Theorem 7.33 (Satz von Perron und Frobenius (1907)) *Sei P eine stochastische Matrix mit positiven Einträgen. Dann gilt:*

i) P besitzt den einfachen Eigenwert 1 und für alle anderen Eigenwerte λ von P gilt $|\lambda| < 1$.

ii) Der linke Eigenvektor des Eigenwertes 1 ist ein nichtnegativer Vektor.

iii) Die Eigenvektoren zu anderen Eigenwerten sind nicht nichtnegativ.

Aus Punkt ii) folgt, dass der Eigenvektor zu 1 durch Normalisieren auch zu einem stochastischen Vektor gemacht werden kann. Der Satz wurde zuerst von Perron sogar für allgemeine Matrizen mit positiven Einträgen gezeigt (der größte Eigenwert muss dann nicht unbedingt 1 sein). Ein verallgemeinerter Satz von Frobenius gilt für so genannte irreduzible Matrizen mit nichtnegativen Einträgen.

Etwas zur Historie, der obige Satz wurde zuerst vor etwa 100 Jahren von Oskar Perron entdeckt, der seinerzeit am Mathematischen Institut der Ludwig-Maximilians-Universität arbeitete und später auch dort nach Aufenthalten in Heidelberg, Göttingen und Tübingen zum Professor ernannt wurde.

21.01.20

7.5.5 Simulation von Verteilungen

In diesem Abschnitt wollen wir uns mit der Simulation von Dichtefunktion beschäftigen, die, wie wir gesehen haben, für den EM-Algorithmus wichtig sind, wenn sich diese nicht als geschlossene Formel angeben lassen. Wie bei der EM-Methode oder bei der konkreten Berechnung des Posteriori-Wahrscheinlichkeit (im Nenner) will man oft Integrale für eine Dichtefunktion $f(x)$ und eine beliebige Funktion g der folgenden Form berechnen:

$$\int g(x, y) \cdot f(x)\, dx.$$

Diese lassen sich oft nicht analytisch genau bestimmen, so dass man für interessierende y zufällige Werte $(x_1, \ldots, x_n)$ gemäß der Dichtefunktion f zieht. Dann lässt sich diese Integral durch den Mittelwert approximieren:

$$\int g(x, y) \cdot f(x)\, dx \approx \frac{1}{n} \sum_{i=1}^{n} g(x_i, y).$$

Ein Beispiel ist die Berechnung der Kreiszahl π. Hier werden mehrfach zufällig n Zahlenpaare $(x_1, x_2) \in [-1 : +1]^2$ uniform gezogen (interpretiert als Punkte im Quadrat $[-1 : +1]^2$). Die k Zahlen mit $x_1^2 + x_2^2 \leq 1$ (interpretiert als Punkte im Kreis mit Radius 1) bekommen für $g(x_1, x_2)$ den Wert 1, die anderen den Wert 0.

Die Dichtefunktion f ist in diesem Fall dann das Produkt der Dichten der Gleichverteilung auf $[-1:1]$. Somit nähert sich der Quotienten k/n dem Wert des Integrals $\int g(x) \cdot f(x)\,dx$ an, dass den Wert des Quotienten aus Kreisfläche durch Quadratfläche entspricht, also $\frac{\pi \cdot r^2}{(2r)^2} = \frac{\pi}{4}$ für $r = 1$. Das Integral wird also näherungsweise durch

$$\int g(x) \cdot f(x)\,dx \approx \frac{1}{n} \sum_{i=1}^{n} g(x) = \frac{k}{n},$$

wobei $x = (x_1, x_2)$. Dies ist in Abbildung 7.8 illustriert.

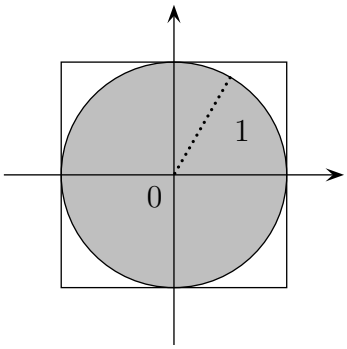


Abbildung 7.8: Beispiel: Berechnung der Kreiszahl π

7.5.5.1 Inversion-Method

Wir nehmen im Folgenden an, dass sich die uniforme Verteilung auf $[0, 1)$ leicht simulieren lässt. In der Regel ist diese durch Zufallszahlengeneratoren in den meisten Programmiersprachen vorhanden.

Wenn die Verteilungsfunktion $F(z) = \int_{-\infty}^{z} f(x)\,dx$ einer Verteilung (und insbesondere ihre Inverse) bekannt bzw. leicht berechenbar oder approximierbar ist, können wir die Verteilung F leicht simulieren. Hierbei ist die Inverse zu F wie folgt definiert:

$$F^{-1}(u) = \inf \{x \in \mathbb{R} \,:\, F(x) \geq u\}.$$

Es handelt sich dabei nur dann um eine echte Inverse, wenn F streng monoton wachsend ist. Aufgrund der Monotonie von F und F^{-1} sowie der rechtsseitigen Stetigkeit von F gilt auf jeden Fall:

$$F^{-1}(u) \leq x \quad \Leftrightarrow \quad u \leq F(x). \tag{7.1}$$

Für unsere Simulation wählen wir zuerst ein uniform auf $[0, 1)$ verteiltes u, die zugehörige Zufallsvariable nennen wir U. Dann sei die Zufallsvariable X wie folgt

definiert: $X = F^{-1}(U)$. Es gilt dann:

$$\begin{aligned} \text{Ws}[X \leq x] &= \text{Ws}[F^{-1}(U) \leq x] \\ &\quad \text{wegen Gleichung 7.1} \\ &= \text{Ws}[U \leq F(x)] \\ &\quad \text{da } U \text{ uniform auf } [0,1) \text{ verteilt ist} \\ &= F(x). \end{aligned}$$

Diese Methode ist die so genannte *Inversionsmethode*.

7.5.5.2 Rejection-Method

Angenommen, wir wollen eine Verteilung nach einer Dichte π konstruieren, die aber explizit nicht bekannt ist. Wir nehmen aber an, dass wir die Funktion $\ell(x) = c \cdot \pi(x)$ kennen (die Konstante c aber unbekannt ist). Weiterhin ist eine Dichte $g(x)$ und eine Konstante M mit der folgenden Eigenschaft bekannt: $\ell(x) \leq M \cdot g(x)$. Ferner nehmen wir an, dass wir auch Elemente gemäß der Dichtefunktion g zufällig ziehen können.

Wir ziehen nun x gemäß der Dichte g und berechnen $r = \frac{\ell(x)}{Mg(x)} \leq 1$. Dann ziehen wir weiterhin ein uniform auf $[0,1)$ verteiltes u. Ist $u \leq r$, dann geben wir x aus. Ansonsten ziehen wir ein neues x und u und wiederholen das Prozedere. Dies ist die so genannte *Rejection-Method* (siehe auch Abbildung 7.9).

Wir zeigen nun noch, dass diese Methode tatsächlich die gewünschte Verteilung generiert. Sei hierfür $I(x)$ eine Indikatorvariable, die genau dann 1 ist, wenn x nach der Dichtefunktion g gezogen und akzeptiert wird (also $u \leq r$ gilt). Da wir nur

Rejection-Method

```
begin
    repeat
        Ziehe x gemäß der Dichte g;
        r := ℓ(x) / (M·g(x));
        Ziehe u gleichverteilt aus [0,1);
    until (u ≤ r) ;
    return x;
end
```

Abbildung 7.9: Algorithmus: Rejection-Method

dann x akzeptieren, wenn $u \leq r = \frac{\ell(x)}{M \cdot g(x)}$ gilt, und da u gleichverteilt aus $[0,1)$ gewählt wird, gilt zunächst einmal:

$$\mathrm{Ws}[I(x) = 1 \mid X = x] = r = \frac{\ell(x)}{M \cdot g(x)} = \frac{c \cdot \pi(x)}{M \cdot g(x)},$$

Weiter gilt dann mit dem Satz der totalen Wahrscheinlichkeit

$$\begin{aligned}
\mathrm{Ws}[I(x) = 1] &= \int \mathrm{Ws}[I(x) = 1 \mid X = x] \cdot Ws[X = x]\, dx \\
&= \int \frac{c \cdot \pi(x)}{M \cdot g(x)} \cdot g(x)\, dx \\
&= \frac{c}{M} \int \pi(x)\, dx \\
&= \frac{c}{M}.
\end{aligned}$$

Somit gilt nach dem Satz von Bayes:

$$\begin{aligned}
\mathrm{Ws}[X = x \mid I(x) = 1] &= \frac{Ws[I(x) = 1 \mid X = x] \cdot \mathrm{Ws}[X = x]}{\mathrm{Ws}[I(x) = 1]} \\
&= \frac{\frac{c \cdot \pi(x)}{M \cdot g(x)} \cdot g(x)}{\frac{c}{M}} \\
&= \pi(x).
\end{aligned}$$

7.5.5.3 Metropolis-Hastings-Algorithmus

Nun wollen wir mithilfe von ergodischen Markov-Ketten eine Verteilung simulieren. Hierzu nehmen wir wieder an, dass die Dichtefunktion ρ wieder nur bedingt bekannt ist. Wie in der Rejection-Methode, sei ρ bis auf einen normalisierenden konstanten Faktor bekannt. Ziel ist es nun eine ergodische Markov-Kette zu konstruieren, die die gesuchte Verteilung als stationäre Verteilung besitzt. Wir müssen also im Wesentlichen eine geeignete Matrix der Zustandsübergangswahrscheinlichkeiten konstruieren.

Wir nehmen hierfür an, dass wir eine beliebige stochastische Matrix S haben, die uns quasi einen Random Walk auf der Markov-Kette liefert. $s_{i,j}$ soll also die Wahrscheinlichkeit angeben, mit der man in den Zustand j gelangt, wenn man sich im Zustand i befindet. Im Folgenden habe S nur echt positive Einträge (d.h. $s_{i,j} > 0$).

Wir definieren dann den so genannten *Metropolis-Hastings-Quotienten* als

$$a_{i,j} := \min\left\{1, \frac{\rho_j \cdot s_{j,i}}{\rho_i \cdot s_{i,j}}\right\},$$

wobei hier ρ die zu simulierende Wahrscheinlichkeitsverteilung ist. Beachte, dass der Quotient berechenbar ist, wenn ρ bis auf einen konstanten Faktor bekannt ist. Im Folgenden gelte $\rho_i > 0$. Weiterhin gilt offensichtlich $0 < a_{i,j} \leq 1$. Dann definieren wir die Matrix P für $i \neq j$ mittels:

$$\begin{aligned} p_{i,j} &:= s_{i,j} \cdot a_{i,j}, \\ p_{i,i} &:= 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} p_{i,j}. \end{aligned}$$

Nach Definition gilt offensichtlich $p_{i,j} \in (0:1]$ für $i \neq j$, da $s_{i,} \in (0,1]$ und $a_{i,j} \in (0,1]$ ist. Damit P nun eine stochastische Matrix ist, muss man also nur noch zeigen, dass $p_{i,i} \geq 0$ gilt (es gilt sogar $p_{i,i} > 0$). Dies gilt, da S eine stochastische Matrix ist:

$$\begin{aligned} p_{i,i} &= 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} p_{i,j} \\ &= 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} a_{i,j} \cdot s_{i,j} \\ &\geq 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} s_{i,j} \\ &= s_{i,i} \\ &> 0. \end{aligned}$$

Die letzte echte Ungleichung folgt wieder aus der Annahme, dass S echt positiv ist.

Beachte, dass für alle $i \neq j$ gilt:

$$a_{i,j} = \frac{\rho_j \cdot s_{j,i}}{\rho_i \cdot s_{i,j}} \leq 1 \quad \Leftrightarrow \quad a_{j,i} = 1.$$

Dies lässt sich leicht durch eine Fallunterscheidung von $a_{i,j} < 1$ und $a_{i,j} = \frac{\rho_j \cdot s_{j,i}}{\rho_i \cdot s_{i,j}} = 1$ einsehen.

Wir zeigen jetzt, dass ρ eine stationäre Verteilung der durch (Q, P, π) induzierten Markov-Kette ist (d.h. $\rho \cdot P = \rho$):

$$\begin{aligned} (\rho \cdot P)_j &= \sum_{i=1}^{m} \rho_i \cdot p_{i,j} \\ &= \sum_{\substack{i=1 \\ i \neq j}}^{m} \rho_i \cdot p_{i,j} + \rho_j \left(1 - \sum_{\substack{i=1 \\ i \neq j}}^{m} p_{j,i} \right) \end{aligned}$$

$$
\begin{aligned}
&= \rho_j + \sum_{\substack{i=1 \\ i \neq j}}^{m} (\rho_i \cdot p_{i,j} - \rho_j \cdot p_{j,i}) \\
&\quad \text{siehe folgender Absatz} \\
&= \rho_j
\end{aligned}
$$

Die letzte Gleichung folgt aus der Tatsache, dass wir fordern, dass die so genannte *detailed balance equation* erfüllt sein muss

$$\forall i, j : \ \rho_i \cdot p_{i,j} = \rho_j \cdot p_{j,i}.$$

Diese besagt nichts anderes, als dass für eine stationäre Verteilung die Änderung des Beitrags von i nach j genau so groß wie der Beitrag der Änderung von j nach i sein muss. Das ist für eine stationäre Verteilung ρ natürlich ein hinreichender Grund (aber nicht notwendigerweise ein notwendiger Grund). Es bleibt also noch zu zeigen, dass für alle i und j gilt:

$$\rho_i \cdot p_{i,j} = \rho_j \cdot p_{j,i}.$$

Für $i = j$ ist das trivial, also sei im Folgenden $i \neq j$. Wir unterscheiden nun zwei Fälle, je nachdem, ob $a_{i,j} < 1$ gilt oder nicht.

Fall 1 ($a_{i,j} < 1$): Es gilt also $a_{i,j} = \frac{\rho_j \cdot s_{j,i}}{\rho_i \cdot s_{i,j}} < 1$ und somit ist $a_{j,i} = 1$.

$$
\begin{aligned}
\rho_i \cdot p_{i,j} &= \rho_i \cdot s_{i,j} \cdot a_{i,j} \\
&= \rho_i \cdot s_{i,j} \cdot \frac{\rho_j \cdot s_{j,i}}{\rho_i \cdot s_{i,j}} \\
&= \rho_j \cdot s_{j,i} \\
&\quad \text{da } a_{j,i} = 1 \\
&= \rho_j \cdot s_{j,i} \cdot a_{j,i} \\
&= \rho_j \cdot p_{j,i}.
\end{aligned}
$$

Fall 2 ($a_{i,j} = 1$): Es gilt also $a_{i,j} = 1$ und somit $a_{j,i} = \frac{\rho_i \cdot s_{i,j}}{\rho_j \cdot s_{j,i}} \leq 1$.

$$
\begin{aligned}
\rho_i \cdot p_{i,j} &= \rho_i \cdot s_{i,j} \cdot a_{i,j} \\
&= \rho_i \cdot s_{i,j} \\
&= \rho_i \cdot s_{i,j} \cdot \frac{\rho_j \cdot s_{j,i}}{\rho_j \cdot s_{j,i}} \\
&= \rho_j \cdot s_{j,i} \cdot a_{j,i} \cdot \\
&= \rho_j \cdot p_{j,i}.
\end{aligned}
$$

Es bleibt noch zu überlegen, dass durch P tatsächlich eine ergodische Markov-Kette konstruiert wird. Wird S als eine positive stochastische Matrix gewählt, dann gilt

für $i \neq j$:

$$\frac{p_{i,j}}{\rho_j} = \frac{a_{i,j} \cdot s_{i,j}}{\rho_j} = \min\left\{1, \frac{\rho_j s_{j,i}}{\rho_i s_{i,j}}\right\} \cdot \frac{s_{i,j}}{\rho_j} = \min\left\{\frac{s_{i,j}}{\rho_j}, \frac{s_{j,i}}{\rho_i}\right\} > 0.$$

Weiterhin haben wir bereits gezeigt, dass $p_{i,i} > 0$, und angenommen, das $\rho_i > 0$ ist, und somit auch $\frac{p_{i,i}}{\rho_i} > 0$ für alle $i, j \in [1:m]$. Nach Satz 7.32 ist die Markov-Kette dann ergodisch.

Man beachte weiterhin, dass dann auch

$$\begin{aligned}
\nu &:= \min\left\{\frac{p_{i,j}}{\rho_j} \ : \ i,j \in Q \wedge \rho_j > 0\right\} \\
&= \min\left\{\frac{s_{i,j}}{\rho_j}, \frac{s_{j,i}}{\rho_i}, \frac{p_{i,i}}{\rho_i} \ : \ i \neq j \in Q\right\} \\
&= \min\left\{\frac{s_{i,j}}{\rho_j}, \frac{p_{i,i}}{\rho_i} \ : \ i \neq j \in Q\right\} \\
&> 0
\end{aligned}$$

und damit letztendlich die Konvergenzgeschwindigkeit des Markov-Modells (Q, P, π) beeinflusst wird. Die Wahl der stochastischen Matrix S zu Beginn ist somit für die Konvergenzgeschwindigkeit des Markov-Modells entscheidend.

Es gibt jetzt zwei Möglichkeiten eine Markov-Kette zu realisieren. Entweder man konstruiert direkt die Markov-Kette für das Markov-Modell (Q, P, π). Hauptproblem dabei ist, dass bei einer großen Zustandsmenge Q die Berechnung der vollständigen Zustandsübergangswahrscheinlichkeitsmatrix P sehr lange dauern kann. Die Konvergenz kann aber schon nach 100 Iterationen sehr gut sein, während die Berechnung einer 1000×1000-Matrix (oder noch größeren) sehr rechenaufwendig sein kann.

Eine andere Möglichkeit ist, dass man das Markov-Modell für die Zustandsübergangswahrscheinlichkeitsmatrix S wiederverwendet. Damit können wir jetzt den Metropolis-Hastings-Algorithmus wie in Abbildung 7.10 definieren. Wir starten in einem zufälligen Zustand. Gemäß der Übergangsmatrix S ermitteln wir einen möglichen Nachfolgezustand q'. Ist der zugehörige Metropolis-Hastings-Quotient nicht größer als eine uniform aus $[0, 1]$ gezogenen Zahl, so wechseln wir in den Zustand q' andernfalls bleiben wir im aktuellen Zustand. Sofern ρ die stationäre Verteilung dieser Markov-Kette ist, folgt die Wahrscheinlichkeit nach einer hinreichenden Anzahl von Iterationen, dass wir uns in einem bestimmten Zustand befinden, gerade der Verteilung ρ.

Wir müssen uns nur überlegen, dass der Übergang von $q^{(t)}$ nach q' mit der gewünschten Wahrscheinlichkeit geschieht. Aufgrund des uniform gewählten Wertes u, wird der Wert $q^{(t+1)}$ mit Wahrscheinlichkeit $a_{q^{(t)},q'}$ auf q' gesetzt. Weiterhin wird q' mit

Metropolis-Hastings

begin

```
// let ϑ be a threshold for the number of iterations
```

choose $q^{(0)}$ at random;

for $(t := 0;\ t \leq \vartheta\ ; t{+}{+})$ **do**

choose q' according to $s_{q^{(t)},q'}$;

choose u according to $\mathit{Unif}([0,1))$;

compute $a_{q^{(t)},q'} := \min\left\{\frac{\rho_{q'} \cdot s_{q',q^{(t)}}}{\rho_{q^{(t)}} \cdot s_{q^{(t)},q'}}, 1\right\}$;

if $(u \leq a_{q^{(t)},q'})$ **then**

$q^{(t+1)} := q'$;

else

$q^{(t+1)} := q^{(t)}$;

end

Abbildung 7.10: Algorithmus: Metropolis-Hastings

Wahrscheinlichkeit $s_{q^{(t)},q'}$ unter allen möglichen Zuständen Augewählt. D.h., wenn aufgrund des Metropolis-Hasting-Quotienten auf den Zustand q' geändert wird, geschieht dies wie gewünscht mit Wahrscheinlichkeit:

$$s_{q^{(t)},q'} \cdot a_{q^{(t)},q'} = p_{q^{(t)},q'}.$$

Letzteres ist dann sinnvoll, wenn man für eine stochastische Matrix S die zugehörige Markov-Kette bereits als black-box wiederverwenden kann. Die Matrix S kann dabei auch einfach definiert sein, d.h. man muss nicht notwendigerweise eine $m \times m$-Matrix S erzeugen und speichern (z.B. wenn alle Einträge $\frac{1}{m}$ sind). Weiter kann das Markov-Modell dabei auch anders realisiert sein, d.h. es muss nicht unbedingt als Zustandsübergang implementiert werden. Für den Metropolis-Hastings-Algorithmus spielen die Details der Implementierung des Markov-Modells (Q, S, π) also überhaupt keine Rolle. Man muss nur effizient auf die Werte $s_{i,j}$ der Matrix S zugreifen bzw. diese für ein Paar (i, j) effizient berechnen können oder aber randomisiert über die Zustände des Markov-Modells bzgl. S laufen können.

Im Metropolis-Hastings-Algorithmus, der in Abbildung 7.10 angegeben ist, kann man auch andere Werte als den Metropolis-Hasting-Quotienten einsetzen. Wählt man z.B. für symmetrische stochastische Matrizen S

$$a_{i,j} = \min\left\{\frac{\rho_j}{\rho_i}, 1\right\}$$

so erhält man den von Metropolis vorgeschlagenen *Metropolis-Quotienten* und der zugehörige Algorithmus heißt dann *Metropolis-Algorithmus*.

Damit der Algorithmus die gewünschte stationäre Verteilung liefert, muss nur die detailed balance equation erfüllt sein. Damit dies die einzige stationäre Verteilung ist, muss man dann noch jeweils (beispielsweise) die Irreduzibilität der so definierten Markov-Kette nachweisen.

23.01.20

Hidden Markov Models 8

8.1 Grundlegende Definitionen und Beispiele

Wir kommen jetzt zu einer Verallgemeinerung von Markov-Ketten, den so genannten Hidden Markov Modellen. Hierbei werden die im Modell vorhanden Zustände und die nach außen sichtbaren Ereignisse, die in den Markov-Ketten streng miteinander verbunden waren, voneinander trennen.

8.1.1 Definition

Wir geben zuerst die formale Definition eines Hidden Markov Models an.

Definition 8.1 *Ein 5-Tupel $M = (Q, \Sigma, P, S, \pi)$ heißt* Hidden Markov Model *M (kurz* HMM*), wobei:*

- *$Q = \{q_1, \ldots, q_n\}$ eine endliche Menge von Zuständen ist;*
- *$\Sigma = \{a_1, \ldots, a_m\}$ eine endliche Menge von Symbolen ist;*
- *P eine stochastische $n \times n$-Matrix der Zustandsübergangswahrscheinlichkeiten ist;*
- *S eine stochastische $n \times m$-Matrix der Emissionswahrscheinlichkeiten ist;*
- *$\pi = (\pi_1, \ldots, \pi_n)$ ein stochastischer Vektor der Anfangswahrscheinlichkeiten ist.*

Ein Pfad $x = (x_1, \ldots, x_\ell) \in Q^\ell$ in M ist wiederum eine Folge von Zuständen. Solche Pfade verhalten sich wie gewöhnliche Markov-Ketten. Nach außen sind jedoch nur Symbole aus Σ sichtbar, wobei zu jedem Zeitpunkt t_i ein Symbol gemäß der Emissionswahrscheinlichkeiten sichtbar wird. Für einen Pfad $x = (x_1, \ldots, x_\ell) \in Q^\ell$ in M sind die sichtbaren Symbolen eine Folge $y = (y_1, \ldots, y_\ell) \in \Sigma^\ell$, wobei

$$\mathrm{Ws}[y_i = a \mid x_i = q] = s_{q,a}$$

für $a \in \Sigma$ und $q \in Q$.

Gilt $\Sigma = Q$ und $S = E$ (wobei E die $n \times n$-Einheitsmatrix ist), dann ist ein Hidden Markov Model M nichts anderes als unser bekanntes Markov-Modell.

Die Wahrscheinlichkeit einer Zustandsfolge $x = (x_1, \ldots, x_\ell)$ mit der emittierten Symbolfolge $y = (y_1, \ldots, y_\ell)$ ist für ein Hidden Markov Model M wie folgt gegeben:

$$\mathrm{Ws}_M[X = x \wedge Y = y] = \pi_{x_1} \cdot s_{x_1,y_1} \cdot \prod_{i=2}^{\ell} (p_{x_{i-1},x_i} \cdot s_{x_i,y_i}).$$

Im Allgemeinen ist uns bei den Hidden Markov Models jedoch nur y, nicht aber x bekannt. Dies ist auch der Grund für den Namen, da die Zustandfolge für uns versteckt ist. Allerdings wird die Zustandsfolge x in der Regel die Information beinhalten, an der wir interessiert sind. Wir werden uns daher später mit Verfahren beschäftigen, wie wir aus der emittierten Folge y die Zustandsfolge x mit großer Wahrscheinlichkeit rekonstruieren können.

8.1.2 Modellierung von CpG-Inseln

Wir werden jetzt die CpG-Inseln mit Hilfe von Hidden Markov Ketten modellieren. Das Hidden Markov Model $M = (Q, \Sigma, P, S, \pi)$ wird dann wie folgt definiert. Als Zustandsmenge wählen wir $Q = \{A^+, C^+, G^+, T^+, A^-, C^-, G^-, T^-\}$. Für jede der möglichen Basen wählen wir zwei Ausprägungen: eine, sofern sich die Base innerhalb der CpG-Insel befindet (die mit + indizierten), und eine, sofern sie sich außerhalb befindet (die mit − indizierten). Das Symbolalphabet ist dann $\Sigma = \{A, C, G, T\}$. Nach außen können wir ja zunächst nicht feststellen, ob wir innerhalb oder außerhalb einer CpG-Insel sind.

Die Zustandsübergangsmatrix sieht wie folgt aus

$$P = \left(\begin{array}{ccc|ccc} & & & \frac{p}{4} & \cdots & \frac{p}{4} \\ & P^+ \cdot (1-p) & & \vdots & & \vdots \\ & & & \frac{p}{4} & \cdots & \frac{p}{4} \\ \hline \frac{q}{4} & \cdots & \frac{q}{4} & & & \\ \vdots & & \vdots & & P^- \cdot (1-q) & \\ \frac{q}{4} & \cdots & \frac{q}{4} & & & \end{array}\right).$$

Wir haben hier zusätzlich noch Zustandsübergangswahrscheinlichkeiten definiert mit denen wir aus einer CpG-Insel bzw. in eine CpG-Insel übergehen können. Dabei gelangen wir mit Wahrscheinlichkeit p aus einer CpG-Inseln und mit Wahrscheinlichkeit q in eine CpG-Insel. Dazu mussten wir natürlich die Zustandsübergangswahrscheinlichkeiten innerhalb bzw. außerhalb der CpG-Inseln mit $(1-p)$ bzw. $(1-q)$ normieren.

Für die Übergänge in bzw. aus einer CpG-Insel haben wir uns das Leben leicht gemacht und haben die Wahrscheinlichkeit von p bzw. q auf die einzelnen Zustände aus $\{A^+, C^+, G^+, T^+\}$ bzw. $\{A^-, C^-, G^-, T^-\}$ gleich verteilt.

Für die Emissionswahrscheinlichkeiten gilt $s_{a^+,a} = s_{a^-,a} = 1$ und $s_{a^+,b} = s_{a^-,b} = 0$ für $a \neq b \in \Sigma$. Hier finden die Emission deterministisch, also anhand des Zustandes statt. Im folgenden Beispiel werden wir sehen, dass es bei Hidden Markov Modellen auch möglich sein kann, von allen Zuständen aus alle Zeichen (eben mit unterschiedlichen Wahrscheinlichkeiten) zu emittieren.

8.1.3 Modellierung eines gezinkten Würfels

Wir geben nun noch ein Hidden Markov Model an, bei dem die Emissionswahrscheinlichkeiten weniger mit dem Zustand korreliert sind als im vorherigen Beispiel der Modellierung der CpG-Inseln. Wir nehmen an, ein Croupier besitzt zwei Würfel, einen normalen und einen gezinkten. Daher besteht die Zustandsmenge $Q = \{F, U\}$ aus nur zwei Zuständen, je nachdem, ob der normale (F=Fair) oder der gezinkte Würfel (U=Unfair) benutzt wird.

Das Alphabet $\Sigma = [1:6]$ modelliert dann die gewürfelte Anzahl Augen. Die Emissionswahrscheinlichkeiten sind dann $s_{F,i} = \frac{1}{6}$ für $i \in [1:6]$ sowie $s_{U,i} = \frac{1}{10}$ für $i \in [1:5]$ und $s_{U,6} = \frac{1}{2}$. Beim fairen Würfel sind alle Seiten gleich wahrscheinlich, während beim unfairen Würfel der Ausgang von sechs Augen erheblich wahrscheinlicher ist.

Die Matrix der Zustandsübergangswahrscheinlichkeiten legen wir mittels

$$P = \begin{pmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{pmatrix}$$

fest. Dies sind die Wahrscheinlichkeiten, mit denen der Croupier den gerade verwendeten Würfel wechselt oder auch nicht. Beispielsweise behält der Croupier mit Wahrscheinlichkeit 0.9 den normalen Würfel bei.

8.2 Viterbi-Algorithmus

In diesem Abschnitt wollen wir zeigen, wie sich zu einer gegebenen emittierten Symbolfolge eine Zustandsfolge konstruieren lässt, die in dem betrachten Hidden Markov Model die größte Wahrscheinlichkeit für die gegebene Symbolfolge besitzt.

8.2.1 Decodierungsproblem

Zuerst einmal wollen wir die Aufgabenstellung noch ein wenig formalisieren und präzisieren.

DECODIERUNGSPROBLEM

Eingabe: Ein HMM $M = (Q, \Sigma, P, S, \pi)$ und eine Folge $y \in \Sigma^\ell$.
Gesucht: Ein wahrscheinlichster Pfad $x \in Q^\ell$ für y in M, d.h. ein $x \in Q^\ell$, so dass $\mathrm{Ws}_M[X = x \mid Y = y]$ maximal ist.

Das bedeutet wir suchen eine Folge $x \in Q^\ell$ mit

$$\mathrm{Ws}[X = x \mid Y = y] = \max\left\{\mathrm{Ws}[X = z \mid Y = y] \ : \ z \in Q^\ell\right\}.$$

Dafür schreiben wir auch

$$x \in \operatorname{argmax}\left\{\mathrm{Ws}[X = z \mid Y = y] \ : \ z \in Q^\ell\right\}.$$

Wir schreiben hier $\in$ statt $=$, da es ja durchaus mehrere verschiedene Folgen mit der gleichen höchsten Wahrscheinlichkeit geben kann. Oft verwendet man hier auch etwas schlampig das Gleichheitszeichen, ohne das nachgewiesen wurde, dass das Maximum eindeutig ist. Man verwendet sogar das Gleichheitszeichen, wenn das Maximum eben nicht eindeutig ist.

8.2.2 Dynamische Programmierung

Wir lösen jetzt das Decodierungsproblem mit Hilfe der Dynamischen Programmierung. Dazu definieren wir:

$$\tilde{W}_q(i) := \max\left\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q) \mid Y = (y_1, \ldots, y_i)] \ : \ x_1, \ldots, x_{i-1} \in Q\right\}.$$

Offensichtlich sind wir eigentlich am Wert von $\max\{\tilde{W}_q(|Y|) : q \in Q\}$ interessiert. Wie wir sehen werden, sind die anderen Werte Hilfsgrößen, die uns bei der Bestimmung einer wahrscheinlichsten Zustandsfolge helfen werden.

Mit Hilfe der Definition der bedingten Wahrscheinlichkeit $\mathrm{Ws}[X \mid Y] = \frac{\mathrm{Ws}[X \wedge Y]}{\mathrm{Ws}[Y]}$ erhalten wir:

$$\tilde{W}_q(i) := \max\left\{\frac{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q) \wedge Y = (y_1, \ldots, y_i)]}{\mathrm{Ws}[Y = (y_1, \ldots, y_i)]} \ : \ x_1, \ldots, x_{i-1} \in Q\right\}.$$

Da der Nenner ja nicht von der Zustandsfolge X abhängt, über die ja maximiert werden soll, genügt es den Zähler allein zu maximieren. Wir versuchen daher im Folgenden eine Zustandsfolge x zu finden, so dass

$$\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q) \wedge Y = (y_1, \ldots, y_i)]$$

maximal wird, und definieren hierfür:

$$W_q(i) := \max\left\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q) \wedge Y = (y_1, \ldots, y_i)] \ : \ x_1, \ldots, x_{i-1} \in Q\right\}.$$

Es gilt offensichtlich

$$W_q(1) \ = \ \mathrm{Ws}[X = q \wedge Y = y_1] \ = \ \pi_q \cdot s_{q,y_1},$$

sowie (mit mehrfacher Anwendung der der Markov-Eigenschaft und des Multiplikationssatzes $\mathrm{Ws}[A \wedge B] = \mathrm{Ws}[A] \cdot \mathrm{Ws}[B \mid A]$):

$W_q(i{+}1)$

$$\begin{aligned}
&= \max\left\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q', q) \wedge Y = (y_1, \ldots, y_{i+1})] \ : \ x_1, \ldots, x_{i-1}, q' \in Q\right\} \\
&\quad \text{(Multiplikationssatz)} \\
&= \max\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q') \wedge Y = (y_1, \ldots, y_i)] \\
&\qquad \cdot \mathrm{Ws}[X_{i+1} = q \wedge Y_{i+1} = y_{i+1} \mid X = (x_1, \ldots, x_{i-1}, q') \wedge Y = (y_1, \ldots, y_i)] \ : \\
&\qquad x_1, \ldots, x_{i-1}, q' \in Q\} \\
&\quad \text{(Markov-Eigenschaft)} \\
&= \max\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q') \wedge Y = (y_1, \ldots, y_i)] \\
&\qquad \cdot \mathrm{Ws}[X_{i+1} = q \wedge Y_{i+1} = y_{i+1} \mid X_i = q'] \ : \ x_1, \ldots, x_{i-1}, q' \in Q\} \\
&= \max\{\max\{\mathrm{Ws}[X = (x_1, \ldots, x_{i-1}, q') \wedge Y = (y_1, \ldots, y_i)] \ : \ x_1, \ldots, x_{i-1} \in Q\} \\
&\qquad \cdot \mathrm{Ws}[X_{i+1} = q \wedge Y_{i+1} = y_{i+1} \mid X_i = q'] \ : \ q' \in Q\} \\
&\quad \text{(Definition von } W\text{)} \\
&= \max\left\{W_{q'}(i) \cdot \mathrm{Ws}[X_{i+1} = q \wedge Y_{i+1} = y_{i+1} \mid X_i = q'] \ : \ q' \in Q\right\} \\
&\quad \text{(Multiplikationssatz)} \\
&= \max\{W_{q'}(i) \cdot \mathrm{Ws}[X_{i+1} = q | X_i = q'] \\
&\qquad \cdot \mathrm{Ws}[Y_{i+1} = y_{i+1} | X_i = q' \wedge X_{i+1} = q] \ : \ q' \in Q\} \\
&\quad \text{(Definition von } P \text{ sowie HMM und Markov-Eigenschaft)} \\
&= \max\left\{W_{q'}(i) \cdot p_{q',q} \cdot \mathrm{Ws}[Y_{i+1} = y_{i+1} | X_{i+1} = q] \ : \ q' \in Q\right\} \\
&= \max\left\{W_{q'}(i) \cdot p_{q',q} \cdot s_{q,y_{i+1}} \ : \ q' \in Q\right\}.
\end{aligned}$$

Offensichtlich gilt $\mathrm{Ws}[Y = y \wedge X = x] = \max\{W_q(\ell) : q \in Q\}$. Somit müssen wir eine Matrix der Größe $O(|Q| \cdot \ell)$ berechnen. Für jeden Eintrag muss dabei ein

Maximum über $|Q|$ Werte gebildet werden. Somit ist die Laufzeit $O(|Q|^2 \cdot \ell)$. Der obige, auf dynamischer Programmierung basierende Algorithmus ist in der Literatur unter dem Namen *Viterbi-Algorithmus* bekannt.

Damit haben wir jedoch nur die Wahrscheinlichkeit einer wahrscheinlichsten Folge von Zuständen berechnet. Die Zustandsfolge selbst können wir jedoch wie beim paarweisen Sequenzen Alignment berechnen, wenn wir uns bei jeder Maximumsbildung merken, welcher Wert gewonnen hat. Ebenso wie beim paarweisen Sequenzen Alignment können wir den Platzbedarf mit Hilfe derselben Techniken des Hirschberg-Algorithmus auf $O(|Q|)$ senken.

Theorem 8.2 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Model und $y \in \Sigma^\ell$. Eine Zustandsfolge $x \in Q^\ell$, die $\mathrm{Ws}[X = x \mid Y = y]$ maximiert, kann in Zeit $O(|Q|^2 \cdot |Y|)$ und Platz $O(|Q|)$ konstruiert werden.*

8.2.3 Implementierungstechnische Details

Die Berechnung von Produkten und Division ist zum einen nicht sonderlich einfach und zum anderen auch nicht immer numerisch stabil, insbesondere wenn die Werte (wie hier) ziemlich klein werden. Daher rechnen wir mit den Logarithmen der Wahrscheinlichkeiten, damit wir es nur mit Addition und Subtraktion zu tun haben. Außerdem erhöht dies die numerische Stabilität, da im Rechner sich kleine Zahlen nur schwer speichern lassen und durch das Logarithmieren betragsmäßig größer werden.

Wir definieren also $\hat{W}_q(i)$ anstelle $W_q(i)$ wie folgt:

$$\hat{W}_q(i) := \log(W_q(i)).$$

Dafür ergeben sich dann die folgenden Rekursionsgleichungen (da aufgrund der Monotonie des Logarithmus die Vertauschung von Logarithmus-Anwendung und Maximumsbildung zulässig ist):

$$\begin{aligned} \hat{W}_q(1) &= \log(\pi_q) + \log(s_{q,y_1}), \\ \hat{W}_q(i+1) &= \log(s_{q,y_{i+1}}) + \max\left\{\hat{W}_{q'}(i) + \log(p_{q',q}) \;:\; q' \in Q\right\}. \end{aligned}$$

Wie man sieht, muss man nur einmal zu Beginn die teuren Kosten für das Logarithmieren investieren.

8.3 Posteriori-Decodierung

Wir wollen jetzt noch eine andere Möglichkeit angeben, wie man die Zustandsfolge im Hidden Markov Model für eine gegebene emittierte Sequenz y rekonstruieren kann. Hierzu wollen wir nun den Zustand zum Zeitpunkt t_i bestimmen, der die größte Wahrscheinlichkeit unter der Annahme besitzt, dass insgesamt die Sequenz y emittiert wurde.

Wir geben hier also explizit die Forderung auf, eine Zustandsfolge im Hidden Markov Model zu betrachten. Wir werden später sehen, dass sich aus der Kenntnis der wahrscheinlichsten Zuständen zu den verschiedenen Zeitpunkten t_i wieder eine Zustandsfolge rekonstruieren lässt. Allerdings kann diese Folge illegal in dem Sinne sein, dass für zwei aufeinander folgende Zustände x_{i-1} und x_i gilt, dass $p_{x_{i-1},x_i} = 0$ ist. Dies folgt methodisch daher, dass wir nur die Zustände gewählt haben, die zu einem festen Zeitpunkt am wahrscheinlichsten sind, aber nicht die Folge samt Zustandsübergangswahrscheinlichkeiten berücksichtigt haben.

Wir kommen darauf später noch einmal zurück und geben jetzt die formalisierte Problemstellung an.

Posteriori-Decodierung

Eingabe: Ein HMM $M = (Q, \Sigma, P, S, \pi)$ und $y \in \Sigma^\ell$.
Gesucht: Bestimme $\mathrm{Ws}[X_i = q \mid Y = y]$ für alle $i \in [1 : \ell]$ und für alle $q \in Q$.

8.3.1 Ansatz zur Lösung

Um jetzt für die Posteriori-Decodierung einen Algorithmus zu entwickeln, formen wir die gesuchte Wahrscheinlichkeit erst noch einmal ein wenig mit Hilfe der Definition für bedingte Wahrscheinlichkeiten

$$\mathrm{Ws}[X = x \mid Y = y] = \frac{\mathrm{Ws}[X = x \wedge Y = y]}{\mathrm{Ws}[Y = y]}$$

um und erhalten damit für $\mathrm{Ws}[X_i = q \mid Y = y]$ mit $y \in \Sigma^\ell$:

$$\begin{aligned} &\mathrm{Ws}[X_i = q \mid Y = y] \\ &= \frac{\mathrm{Ws}[X_i = q \wedge Y = y]}{\mathrm{Ws}[Y = y]} \end{aligned}$$

Nach dem Multiplikationssatz gilt $\mathrm{Ws}[X \wedge Y] = \mathrm{Ws}[X] \cdot \mathrm{Ws}[Y \mid X]$ und wir erhalten somit:

$$= \frac{\mathrm{Ws}[Y'{=}(y_1, \ldots, y_i) \wedge X_i{=}q] \cdot \mathrm{Ws}[Y''{=}(y_{i+1}, \ldots, y_\ell) \mid Y'{=}(y_1, \ldots, y_i) \wedge X_i{=}q]}{\mathrm{Ws}[Y = y]}$$

Aufgrund der Definition von Hidden Markov Models folgt, dass die emittierten Symbole $(y_{i+1}, \ldots, y_\ell)$ nur von den Zuständen $(x_{i+1}, \ldots, x_\ell)$ abhängen. Aufgrund der Markov-Eigenschaft hängen die Zustände $(x_{i+1}, \ldots, x_\ell)$ aber nur vom Zustand x_i ab. Somit folgt:

$$\begin{aligned} &= \frac{\mathrm{Ws}[Y' = (y_1, \ldots, y_i) \wedge X_i = q] \cdot \mathrm{Ws}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_i = q]}{\mathrm{Ws}[Y = y]} \\ &= \frac{f_q(i) \cdot b_q(i)}{\mathrm{Ws}[Y = y]}. \end{aligned}$$

Hierbei haben wir für die Umformung in der letzten Zeile die folgende Notation verwendet:

$$\begin{aligned} f_q(i) &:= \mathrm{Ws}[X_i = q \wedge Y' = (y_1, \ldots, y_i)], \\ b_q(i) &:= \mathrm{Ws}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_i = q]. \end{aligned}$$

Die hier eingeführten Wahrscheinlichkeiten $f_q(i)$ bzw. $b_q(i)$ werden wir im Folgenden als *Vorwärtswahrscheinlichkeiten* bzw. als *Rückwärtswahrscheinlichkeiten* bezeichnen. Wie wir noch sehen werden ist der Grund hierfür ist, dass sich $f_q(i)$ in der Zustandsfolge vorwärts aus $f_q(1), \ldots, f_q(i-1)$ berechnen lässt, während sich $b_q(i)$ in der Zustandsfolge rückwärts aus $b_q(\ell), \ldots, b_q(i+1)$ berechnen lässt.

8.3.2 Vorwärts-Algorithmus

Wir werden uns jetzt überlegen, wie sich die Vorwärtswahrscheinlichkeiten mit Hilfe der dynamische Programmierung berechnen lassen. Dazu stellen wir im Folgenden fest, dass die folgenden Rekursionsgleichungen gelten:

$$\begin{aligned} f_q(1) &= \pi_q \cdot s_{q,y_1}, \\ f_q(i+1) &= s_{q,y_{i+1}} \cdot \sum_{q' \in Q} (f_{q'}(i) \cdot p_{q',q}). \end{aligned}$$

Die Anfangsbedingung für $f_q(1)$ ergeben sich direkt aus den gegebenen Anfangswahrscheinlichkeiten π und den Emissionswahrscheinlichkeiten s. Für die Herleitung der Rekursionsgleichung erhalten wir nach dem Satz der totalen Wahrscheinlichkeiten folgendes:

$$\begin{aligned} &f_q(i+1) \\ &\quad = \mathrm{Ws}[X_{i+1} = q \wedge Y' = (y_1, \ldots, y_{i+1})], \end{aligned}$$

$$
\begin{aligned}
&= \sum_{q' \in Q} \mathrm{Ws}[X_{i+1} = q \wedge Y' = (y_1, \ldots, y_{i+1}) \mid X_i = q'] \cdot \mathrm{Ws}[X_i = q'], \\
&\quad \text{mittels } \mathrm{Ws}[A \wedge B] = \mathrm{Ws}[A] \cdot \mathrm{Ws}[B \mid A] \\
&= \sum_{q' \in Q} \mathrm{Ws}[X_{i+1} = q \mid X_i = q'] \cdot \mathrm{Ws}[Y' = (y_1, \ldots, y_{i+1}) \mid X_{i+1} = q \wedge X_i = q'] \\
&\qquad \cdot \mathrm{Ws}[X_i = q'] \\
&\quad \text{da } \mathrm{Ws}[X_{i+1} = q \mid X_i = q'] = p_{q',q} \\
&= \sum_{q' \in Q} p_{q',q} \cdot \mathrm{Ws}[Y' = (y_1, \ldots, y_{i+1}) \mid X_{i+1} = q \wedge X_i = q'] \cdot \mathrm{Ws}[X_i = q'] \\
&\quad \text{mittels } \mathrm{Ws}[A \wedge B] = \mathrm{Ws}[A] \cdot \mathrm{Ws}[B \mid A] \\
&= \sum_{q' \in Q} p_{q',q} \cdot \mathrm{Ws}[Y_{i+1} = y_{i+1} \mid X_{i+1} = q \wedge X_i = q'] \\
&\qquad \cdot \mathrm{Ws}[Y'' = (y_1, \ldots, y_i) \mid X_{i+1} = q \wedge X_i = q \wedge Y_{i+1} = y_{i+1}] \cdot \mathrm{Ws}[X_i = q']
\end{aligned}
$$

Da $\mathrm{Ws}[Y_{i+1} = y_{i+1} \mid X_{i+1} = q \wedge X_i = q']$ nur von X_{i+1} und nicht von X_i abhängt, ist $\mathrm{Ws}[Y_{i+1} = y_{i+1} \mid X_{i+1} = q \wedge X_i = q'] = \mathrm{Ws}[Y_{i+1} = y_{i+1} \mid X_{i+1} = q] = s_{q,y_{i+1}}$.

$$
\begin{aligned}
&= \sum_{q' \in Q} p_{q',q} \cdot s_{q,y_{i+1}} \\
&\qquad \cdot \mathrm{Ws}[Y'' = (y_1, \ldots, y_i) \mid X_{i+1} = q \wedge X_i = q' \wedge Y_{i+1} = y_{i+1}] \cdot \mathrm{Ws}[X_i = q']
\end{aligned}
$$

Da $\mathrm{Ws}[Y'' = (y_1, \ldots, y_i) \mid X_{i+1} = q \wedge X_i = q' \wedge Y_{i+1} = y_{i+1}]$ ebenfalls nur von X_i, aber nicht von X_{i+1} und Y_{i+1} abhängt, erhalten wir weiter.

$$
\begin{aligned}
&= \sum_{q' \in Q} p_{q',q} \cdot s_{q,y_{i+1}} \cdot \mathrm{Ws}[Y'' = (y_1, \ldots, y_i) \mid X_i = q'] \cdot \mathrm{Ws}[X_i = q'] \\
&\quad \text{mit } \mathrm{Ws}[Y = y \mid X = x] \cdot \mathrm{Ws}[X = x] = \mathrm{Ws}[Y = y \wedge X = x] \text{ um:} \\
&= \sum_{q' \in Q} p_{q',q} \cdot s_{q,y_{i+1}} \cdot \mathrm{Ws}[Y'' = (y_1, \ldots, y_i) \wedge X_i = q'] \\
&= \sum_{q' \in Q} p_{q',q} \cdot s_{q,y_{i+1}} \cdot f_{q'}(i).
\end{aligned}
$$

Diese Rekursionsgleichungen sind denjenigen aus dem Viterbi-Algorithmus sehr ähnlich. Wir haben hier nur die Summe statt einer Maximumsbildung, da wir über alle Pfade in den gesuchten Zustand gelangen dürfen. Dies ist in letzter Instanz die Folge dessen, dass wir nach einem wahrscheinlichsten Zustand zum Zeitpunkt t_i suchen und nicht nach einer wahrscheinlichen Zustandsfolge für die Zeitpunkte $(t_1, \ldots, t_i)$.

Wir halten noch fest, dass offensichtlich $\mathrm{Ws}[Y = y] = \sum_{q \in Q} f_q(\ell)$ gilt.

Lemma 8.3 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Model und $y \in \Sigma^\ell$. Die Vorwärtswahrscheinlichkeiten $f_q(i) = \mathrm{Ws}[X_i = q \wedge Y' = (y_1, \ldots, y_i)]$ können für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot \ell)$ und Platz $O(|Q|)$ berechnet werden.*

Den Algorithmus zur Berechnung der Vorwärtswahrscheinlichkeiten wird in der Literatur auch oft als *Vorwärts-Algorithmus* bezeichnet.

8.3.3 Rückwärts-Algorithmus

Wir werden uns jetzt überlegen, wie sich die Rückwärtswahrscheinlichkeiten mit Hilfe der dynamische Programmierung berechnen lässt. Dazu stellen wir fest, dass die folgenden Rekursionsgleichungen gelten:

$$\begin{aligned} b_q(\ell) &= 1, \\ b_q(i) &= \sum_{q' \in Q} (p_{q,q'} \cdot s_{q',y_{i+1}} \cdot b_{q'}(i+1)). \end{aligned}$$

Die Anfangsbedingung und die Rekursionsgleichung können wir analog wie bei, Vorwärts-Algorithmus herleiten. Auch hier gilt $\mathrm{Ws}[Y = y] = \sum_{q \in Q} \pi_q \cdot s_{q,y_1} \cdot b_q(1)$.

Lemma 8.4 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Model und $y \in \Sigma^\ell$. Die Rückwärtswahrscheinlichkeiten $b_q(i) = \mathrm{Ws}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_i = q]$ können für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot \ell)$ und Platz $O(|Q|)$ berechnet werden.*

Den Algorithmus zur Berechnung der Rückwärtswahrscheinlichkeiten wird in der Literatur auch oft als *Rückwärts-Algorithmus* bezeichnet.

Aus den beiden letzten Lemmata folgt aus unserer vorherigen Diskussion unmittelbar das folgende Theorem.

Theorem 8.5 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Model und $y \in \Sigma^\ell$. Die Posteriori-Wahrscheinlichkeit $\mathrm{Ws}[X_i = q \mid Y = y]$ kann für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot \ell)$ und Platz $O(|Q|)$ berechnet werden.*

8.3.4 Implementierungstechnische Details

Auch hier ist die Berechnung von Produkten und Division zum einen nicht sonderlich einfach und zum anderen auch nicht immer numerisch stabil, insbesondere wenn die Werte (wie hier) ziemlich klein werden. Daher rechnen wir mit den Logarithmen der Wahrscheinlichkeiten, da wir es dann nur mit Addition und Subtraktion zu tun haben. Wie bereits erwähnt, erhöht sich dadurch ebenfalls die numerische Stabilität.

Wir definieren also jetzt die *logarithmischen Vorwärts- und Rückwärtswahrscheinlichkeiten*:

$$\begin{aligned}\hat{f}_q(i) &:= \log(f_q(i)) = \log(\mathrm{Ws}[X_i = q \wedge Y' = (y_1, \ldots, y_i)]),\\ \hat{b}_q(i) &:= \log(b_q(i)) = \log(\mathrm{Ws}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_i = q]).\end{aligned}$$

Dazu stellen wir fest, dass dann die folgenden Rekursionsgleichungen

$$\begin{aligned}\hat{f}_q(1) &= \log(\pi_q) + \log(s_{q,y_1}),\\ \hat{f}_q(i+1) &= \log(s_{q,y_{i+1}}) + \log\left(\sum_{q' \in Q} (\exp(\hat{f}_{q'}(i)) \cdot p_{q',q}\right)\end{aligned}$$

für die logarithmischen Vorwärtswahrscheinlichkeiten gelten und analog für die logarithmischen Rückwärtswahrscheinlichkeiten

$$\begin{aligned}\hat{b}_q(n) &= 0,\\ \hat{b}_q(i) &= \log\left(\sum_{q' \in Q} (p_{q,q'} \cdot s_{q',y_{i+1}} \cdot \exp(\hat{b}_{q'}(i+1)))\right)\end{aligned}$$

gelten. Da hier nun statt der Maximumsbildung eine Summe involviert ist, müssen wir bei der dynamischen Programmierung sowohl Logarithmieren als auch Exponenzieren.

28.02.20

8.3.5 Anwendung und Beispiel

Mit Hilfe der Posteriori-Decodierung können wir jetzt alternativ den gesuchten Pfad $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_\ell)$ wie folgt definieren:

$$\tilde{x}_i := \mathrm{argmax}\left\{\mathrm{Ws}[X_i = q \mid Y = y] \;:\; q \in Q\right\}.$$

Das beideutet, dass wir den Pfad aus den zum Zeitpunkt i wahrscheinlichsten Zuständen zusammensetzten. Wir merken hier noch an, dass die Zustandsfolge nicht unbedingt eine zulässige Zustandsfolge sein muss. In der Folge $\tilde{x}$ können durchaus

die Teilsequenz $(\tilde{x}_{i-1}, \tilde{x}_i)$ auftreten, so dass $p_{\tilde{x}_{i-1},\tilde{x}_i} = 0$. Dies folgt daraus, dass wir bei der Konstruktion der Folge $\tilde{x}$ für den Zeitpunkt t_i immer nur den Zustand auswählen, der die größte Wahrscheinlichkeit für die beobachtete emittierte Sequenz Y besitzt. Wir achten dabei im Gegensatz zum Viterbi-Algorithmus nicht darauf, eine ganze konsistente Sequenz zu betrachten.

Dies wollen wir uns noch einmal an dem folgenden Beispiel des Hidden Markov Models $M = (Q, \Sigma, P, S, \pi)$ klar machen. Hierbei ist $Q = \{q_1, \ldots, q_5\}$, $\Sigma = \{a, b\}$ und $\pi = (1, 0, 0, 0, 0)$. Die Matrix P der Zustandsübergangswahrscheinlichkeiten ist in der Abbildung 8.1 illustriert.

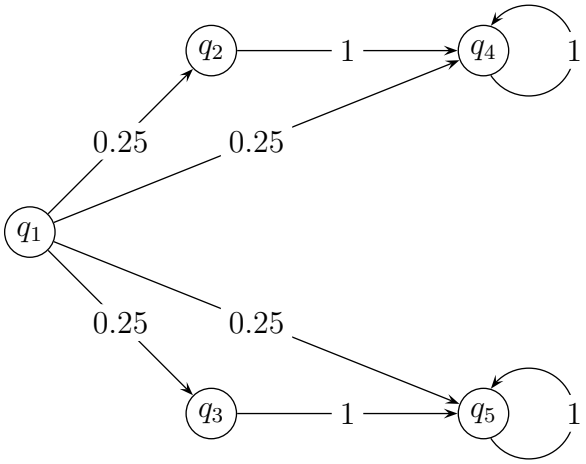


Abbildung 8.1: Beispiel: Ein HMM mit illegaler a posteriori Zustandsfolge

Die Matrix S der Emissionswahrscheinlichkeiten ist wie folgt gegeben:

$$S = \begin{pmatrix} s_1 := s_{q_1,a} & s_{q_1,b} = 1 - s_1 \\ s_2 := s_{q_2,a} & s_{q_2,b} = 1 - s_2 \\ s_3 := s_{q_3,a} & s_{q_3,b} = 1 - s_3 \\ s_4 := s_{q_4,a} & s_{q_4,b} = 1 - s_4 \\ s_5 := s_{q_5,a} & s_{q_5,b} = 1 - s_5 \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 \\ 0.8 & 0.2 \\ 0.6 & 0.4 \\ 0.4 & 0.6 \\ 0.5 & 0.5 \end{pmatrix},$$

wobei $s_1 = 1$, $s_2 = 0.8$, $s_3 = 0.6$, $s_4 = 0.4$ und $s_5 = 0.5$

Für die Zeichenfolge *aaa* kann man jetzt die Posteriori-Wahrscheinlichkeit (ausgehend von den zugehörigen Vorwärts- und Rückwärtswahrscheinlichkeiten) zu den Zeitpunkten t_1, t_2 und t_3 für alle Zustände bestimmen. In der Abbildung 8.2 wurden die Vorwärts- und Rückwärtswahrscheinlichkeiten $f_q(i)$ und $b_q(i)$ sowie die daraus resultierenden Posteriori-Wahrscheinlichkeiten $f_q(i) \cdot b_q(i)$ berechnet.

Wie man daraus abliest, ist für die Ausgabefolge *aaa* der wahrscheinlichste Zustand zum Zeitpunkt t_2 bzw. t_3 gerade q_2 bzw. q_5. Es gilt jedoch $p_{q_2,q_5} = 0$. Somit enthält

$f_q(i)$	1	2	3
q_1	1.0	0.0	0.0
q_2	0.0	$\frac{s_2}{4}$	0.0
q_3	0.0	$\frac{s_3}{4}$	0.0
q_4	0.0	$\frac{s_4}{4}$	$\frac{s_2 \cdot s_4 + s_4^2}{4}$
q_5	0.0	$\frac{s_5}{4}$	$\frac{s_3 \cdot s_5 + s_5^2}{4}$

$b_q(i)$	1	2	3
q_1	$\cdots$	$\frac{s_2+s_3+s_4+s_5}{4}$	1.0
q_2	$\cdots$	s_4	1.0
q_3	$\cdots$	s_5	1.0
q_4	$\cdots$	s_4	1.0
q_5	$\cdots$	s_5	1.0

$f_q(i)$	1	2	3
q_1	1.0000	0.0000	0.0000
q_2	0.0000	0.2000	0.0000
q_3	0.0000	0.1500	0.0000
q_4	0.0000	0.1000	0.1200
q_5	0.0000	0.1250	0.1375

$b_q(i)$	1	2	3
q_1	$\cdots$	0.5750	1.0000
q_2	$\cdots$	0.4000	1.0000
q_3	$\cdots$	0.5000	1.0000
q_4	$\cdots$	0.4000	1.0000
q_5	$\cdots$	0.5000	1.0000

$f_q(i) \cdot b_q(i)$	1	2	3
q_1	$b_{q_1}(1)$	0.0000	0.0000
q_2	0.000	0.0800	0.0000
q_3	0.000	0.0750	0.0000
q_4	0.000	0.0400	0.1200
q_5	0.000	0.0625	0.1375

Abbildung 8.2: Beispiel: Die expliziten Vorwärts- und Rückwärts- sowie Posteriori-Wahrscheinlichkeiten für M zur Folge der emittierten Symbole $y = aaa$

die mit dieser Methode vorhergesagte Zustandsfolge für die emittierte Folge aaa einen illegalen Zustandsübergang (wie wir dies bereits als Möglichkeit angekündigt hatten).

Weiterhin lassen sich mit Hilfe der Posteriori-Decodierung auch wichtige Eigenschaften von der konstruierten Folge $\tilde{x}$ ableiten, die sich mit Hilfe einer Funktion $g : Q \to \mathbb{R}$ beschreiben lassen. Dazu definieren wir basierend auf der emittierten Folge y und der gegebenen Funktion g eine Funktion G, die jedem Zeitpunkt einen Wert wie folgt zuordnet (ggf. kann g auch von der Folge der emittierten Symbole y abhängen):

$$G(i \mid Y = y) := \sum_{q \in Q} \mathrm{Ws}[X_i = q \mid Y = y] \cdot g(q).$$

Ein Beispiel hierfür ist unser Problem der CpG-Inseln. Hierfür definieren wir die Funktion g wie folgt:

$$g:\ Q \to \mathbb{R}:\ Q \ni q \mapsto \begin{cases} 1 & \text{für } q \in \{A^+, C^+, G^+, T^+\} \\ 0 & \text{für } q \in \{A^-, C^-, G^-, T^-\} \end{cases}.$$

$G(i \mid Y)$ kann man dann als die a posteriori Wahrscheinlichkeit interpretieren, dass sich das Zeichen an Position i (d.h. zum Zeitpunkt t_i) innerhalb einer CpG-Insel befindet.

8.4 Schätzen von HMM-Parametern

In diesem Abschnitt wollen wir uns mit dem Schätzen der Parameter eines Hidden Markov Models beschäftigen. Hierbei nehmen wir an, dass uns die Zustände des Hidden Markov Models selbst bekannt sind. Wir wollen also versuchen, die stochastischen Matrizen P und S zu schätzen.

Um diese stochastischen Matrizen schätzen zu können, werden wir einige Folgen der emittierten Symbole, also $y^{(1)}, \ldots, y^{(m)} \in \Sigma^*$ der Länge $\ell^{(1)}, \ldots, \ell^{(m)}$ betrachten und versuchen, auf die gesuchten Wahrscheinlichkeiten zu schließen. Wir werden jetzt zwei Fälle unterscheiden, je nachdem, ob uns auch die zugehörige Zustandsfolgen $x^{(1)}, \ldots, x^{(m)}$ bekannt sind oder nicht. Die Folgen $y^{(i)}$ werden oft auch *Trainingssequenzen* genannt. Wir schreiben im Folgenden $\vec{y} = (y^{(1)}, \ldots, y^{(m)})$ und mit y bezeichen wir Element aus $\vec{y}$, wenn uns der obere Index nicht wichtig ist, d.h. $y = y^{(i)}$ für ein $i \in [1:m]$.

8.4.1 Zustandsfolge bekannt

Ein Beispiel, für das auch die Zustandsfolgen unter gewissen Umständen bekannt sind, ist das Problem der CpG-Inseln. Hier können wir aus anderen, biologischen Untersuchungen auch noch einige CpG-Inseln und somit die dort durchlaufenen Zustandfolgen kennen. In diesem Beispiel ist uns sogar die stochastische Matrix S der Emissionswahrscheinlichkeiten bereits bekannt.

Wir zählen jetzt die Anzahl $\hat{p}_{q,q'}$ der Transition $q \to q'$ und die Anzahl $\hat{s}_{q,a}$ der Emission von a aus dem Zustand q. Als Schätzer wählen wir dann ganz einfach

die entsprechenden relativen Häufigkeiten (was nichts anderem als dem Maximum-Likelihood-Schätzer entspricht):

$$p_{q,q'} := \frac{\hat{p}_{q,q'}}{\sum_{r \in Q} \hat{p}_{q,r}}, \tag{8.1}$$

$$s_{q,a} := \frac{\hat{s}_{q,a}}{\sum_{\sigma \in \Sigma} \hat{s}_{q,\sigma}}. \tag{8.2}$$

Ist die Anzahl der Trainingssequenzen sehr klein, so verändern wir die Schätzer ein wenig, um künstliche Wahrscheinlichkeiten von 0 zu vermeiden (die nur auftreten, weil wir kein entsprechendes Ereignis in unseren Trainingssequenzen haben). Dazu erhöhen wir die jeweilige beobachtete Anzahl um 1 und passen die Gesamtanzahl entsprechend an. Dies entspricht der so genannten Laplace-Regel. Die Anzahl der Transitionen aus dem Zustand q erhöhen wir also um $|Q|$, da wir für jede mögliche Transition (q, q') eine künstliche hinzugenommen haben. Die Anzahl der emittierten Zeichen aus dem Zustand q um $|\Sigma|$, da wir für jedes Zeichen $a \in \Sigma$ eine weitere künstliche Emission hinzugefügt haben.

$$p'_{q,q'} := \frac{1 + \hat{p}_{q,q'}}{|Q| + \sum_{r \in Q} \hat{p}_{q,r}}, \tag{8.3}$$

$$s'_{q,a} := \frac{1 + \hat{s}_{q,a}}{|\Sigma| + \sum_{\sigma \in \Sigma} \hat{s}_{q,\sigma}}. \tag{8.4}$$

Der Grund für die letzte Modifikation ist, dass es ein qualitativer Unterschied ist, ob eine Übergangswahrscheinlichkeit gleich Null oder nur sehr klein ist. Bei einer geringen Anzahl von Trainingssequenzen haben wir keinen Grund, eine Übergangswahrscheinlichkeit von Null anzunehmen, da das Nichtauftreten nur aufgrund der kleinen Stichprobe möglich ist. Daher vergeben wir bei einer relativen Häufigkeit von Null bei kleinen Stichproben lieber eine kleine Wahrscheinlichkeit.

8.4.2 Zustandsfolge unbekannt — Baum-Welch-Algorithmus

In der Regel werden die Zustandsfolgen, die zu den bekannten Trainingssequenzen gehören, jedoch unbekannt sein. In diesem Fall ist das Problem wieder $\mathcal{NP}$-hart (unter geeigneten Definitionen der gewünschten Eigenschaften der Schätzer). Daher werden wir in diesem Fall eine Heuristik zum Schätzen der Parameter vorstellen.

Auch hier werden wir wieder versuchen die relative Häufigkeit als Schätzer zu verwenden. Dazu werden gemäß gegebener Matrizen P bzw. S für die Zustandsübergangswahrscheinlichkeiten bzw. für die Emissionswahrscheinlichkeiten die Zustandsfolge ermittelt und wir können dann wieder die relativen Häufigkeiten abschätzen.

Woher erhalten wir jedoch die Matrizen P und S? Wir werden zuerst mit beliebigen stochastischen Matrizen P und S starten und dann mit Hilfe der neuen Schätzungen versuchen die Schätzungen immer weiter iterativ zu verbessern.

Seien P und S also beliebige stochastische Matrizen. Wir versuchen zuerst die Anzahl der Transitionen (q, q') unter Annahme dieser Matrizen als Zustandsübergangs- bzw. Emissionswahrscheinlichkeiten zu bestimmen. Es gilt für eine beliebige emittierte Folge Y nach dem Satz von Bayes (wobei wir wie bei der Berechnung der Vorwärts- und Rückwärtswahrscheinlichkeiten im Folgenden den Multiplikationssatz ($\mathrm{Ws}[X, Y] = \mathrm{Ws}[X] \cdot \mathrm{Ws}[Y \mid X]$) und die Markov-Eigenschaft bei fast jeder Zeile ausnutzen):

$$
\begin{aligned}
&\mathrm{Ws}_{P,S}[X_i = q, X_{i+1} = q' \mid Y = y] \\
&= \frac{\mathrm{Ws}_{P,S}[Y = y, X_i = q, X_{i+1} = q']}{\mathrm{Ws}_{P,S}[Y = y]} \\
&= \frac{\mathrm{Ws}_{P,S}[Y'{=}(y_1, \ldots, y_i), X_i{=}q] \cdot \mathrm{Ws}_{P,S}[Y''{=}(y_{i+1}, \ldots, y_\ell), X_{i+1}{=}q' \mid X_i{=}q]}{\mathrm{Ws}_{P,S}[Y = y]} \\
&= \frac{1}{\mathrm{Ws}_{P,S}[Y = y]} \Big(\mathrm{Ws}_{P,S}[Y' = (y_1, \ldots, y_i), X_i = q] \cdot \mathrm{Ws}_{P,S}[X_{i+1} = q' \mid X_i = q] \\
&\qquad\qquad \cdot \mathrm{Ws}_{P,S}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_{i+1} = q'] \Big) \\
&= \frac{1}{\mathrm{Ws}_{P,S}[Y = y]} \Big(\mathrm{Ws}_{P,S}[Y' = (y_1, \ldots, y_i), X_i = q] \cdot \mathrm{Ws}_{P,S}[X_{i+1} = q' \mid X_i = q] \\
&\qquad\qquad \cdot \mathrm{Ws}_{P,S}[Y'' = y_{i+1} \mid X_{i+1} = q'] \\
&\qquad\qquad \cdot \mathrm{Ws}_{P,S}[Y'' = (y_{i+2}, \ldots, y_\ell) \mid X_{i+1} = q'] \Big) \\
&= \frac{f_q(i) \cdot p_{q,q'} \cdot s_{q',y_{i+1}} \cdot b_{q'}(i+1)}{\mathrm{Ws}_{P,S}[Y = y]}.
\end{aligned}
$$

Die in der letzten Umformung auftretenden Terme $f_q(i)$ und $b_q(i)$ sind die bereits bekannten (und somit auch effizient berechenbaren) Vorwärts- und Rückwärtswahrscheinlichkeiten unter Berücksichtigung der Matrix P der Zustandsübergangswahrscheinlichkeiten und der Matrix S der Emissionswahrscheinlichkeiten. Damit lässt sich natürlich auch $\mathrm{Ws}_{P,S}[Y = y]$ leicht berechnen.

Damit haben wir jetzt die Wahrscheinlichkeit der Transitionen $(q \to q')$ für die emittierte Symbolfolge y unter Berücksichtigung der Matrizen P bzw. S für die Zustandsübergangs- bzw. Emissionswahrscheinlichkeiten bestimmt, die sich mit Hilfe der verwendeten Vorwärts- und Rückwärtswahrscheinlichkeiten effizient berechnen können. Somit können wir jetzt einen neuen Schätzer (basierend auf P und S) für die erwartete Anzahl der Zustandsübergange $(q \to q')$ in den Trainingssequenzen

$\vec{Y} = (Y^{(1)}, \ldots, Y^{(m)})$ (wobei $Y^{(i)} = (Y_1^{(i)}, \ldots, Y_{\ell_i}^{(i)})$) jeweils eine beobachte Folge von beobachteten Symbolen eine HMM-Durchlaufs ist) und somit mithilfe der Gleichung 8.1 bzw. 8.3 auch für die zugehörigen Zustandsübergangswahrscheinlichkeit angeben:

$$\begin{aligned}
\hat{p}_{q,q'} &:= \mathbb{E}_{P,S}[(q \to q') \mid \vec{Y} = \vec{y}] \\
&= \sum_{j=1}^{m} \sum_{i=1}^{\ell^{(j)}-1} \mathrm{Ws}_{P,S}[X_i = q, X_{i+1} = q' \mid Y^{(j)} = y^j] \\
&\quad \text{mit der eben hergeleiteten Gleichung} \\
&= \sum_{j=1}^{m} \sum_{i=1}^{\ell^{(j)}-1} \frac{f_q^{(j)}(i) \cdot p_{q,q'} \cdot s_{q,y_{i+q}} \cdot b_{q'}^{(j)}(i+1)}{\mathrm{Ws}[Y^{(j)} = y^{(j)}]}.
\end{aligned}$$

Beachte, dass wir hier wieder einen Erwartungswert berechnen und dass wir aus den absoluten Häufigkeiten $\hat{p}_{q,q'}$ letztendlich die relativen Häufigkeiten (siehe Gleichung 8.1 bzw. 8.3) berechnen und als einen neuen Schätzer für die Zustandsübergangswahrscheinlichkeiten verwenden, was eine Maximierung gemäß dem Maximum-Likelihood-Schätzer ist. Wie wir noch sehen werden, verwenden wir hier die EM-Methode.

Analog können wir auch für die Emissionwahrscheinlichkeiten vorgehen. Berechnen wir zuerst wieder die Wahrscheinlichkeit für eine Emissionen eines Zeichens $a \in \Sigma$ aus dem Zustand $q \in Q$ für eine gegebene emittierte Folge y unter Berücksichtigung der Matrizen P bzw. S der Zustandsübergangs- bzw. Emissionswahrscheinlichkeiten. Nach dem Satz von Bayes gilt (wobei wir wie bei der Berechnung der Vorwärts- und Rückwärtswahrscheinlichkeiten im Folgenden den Multiplikationssatz ($\mathrm{Ws}[X, Y] = \mathrm{Ws}[X] \cdot \mathrm{Ws}[Y \mid X]$) und die Markov-Eigenschaft bei fast jeder Zeile ausnutzen):

$$\begin{aligned}
&\mathrm{Ws}_{P,S}[X_i = q \mid Y = y] \\
&= \frac{\mathrm{Ws}_{P,S}[X_i = q, Y = y]}{\mathrm{Ws}_{P,S}[Y = y]} \\
&= \frac{\mathrm{Ws}_{P,S}[Y' = (y_1, \ldots, y_i), X_i = q] \cdot \mathrm{Ws}_{P,S}[Y'' = (y_{i+1}, \ldots, y_\ell) \mid X_i = q]}{\mathrm{Ws}_{P,S}[Y = y]} \\
&= \frac{f_q(i) \cdot b_q(i)}{\mathrm{Ws}_{P,S}[Y = y]}.
\end{aligned}$$

Somit können wir die erwartete Anzahl von Emissionen des Symbols $a \in \Sigma$ aus dem Zustand $q \in Q$ für die Trainingssequenzen $\vec{Y}$ unter Berücksichtigung der alten Schätzer P bzw. S für die Zustandsübergangs- bzw. Emissionswahrscheinlichkeiten und somit mithilfe der Gleichung 8.2 bzw. 8.4 auch die zugehörigen Emissionswahrscheinlichkeiten angeben. Man beachte hierbei, dass durch die Bedingung $Y = Y$

bereits klar ist, dass hierbei im Zustand q das Zeichen y_i emittiert wird. Somit gilt nun für die absolute Häufigkeit:

$$\begin{aligned}\hat{s}_{q,a} &:= \mathbb{E}_{P,S}[(q \to a) \mid \vec{Y} = \vec{y}] \\ &= \sum_{j=1}^{m} \sum_{\substack{i=1 \\ y_i^{(j)}=a}}^{\ell^{(j)}} \mathrm{Ws}_{P,S}[X_i = q \mid Y^{(j)} = y^{(j)}] \\ &= \sum_{j=1}^{m} \sum_{\substack{i=1 \\ y_i^{(j)}=a}}^{\ell^{(j)}} \frac{f_q^{(j)}(i) \cdot b_q^{(j)}(i)}{\mathrm{Ws}_{P,S}[Y^{(j)} = y^{(j)}]}.\end{aligned}$$

Wir können also aus einer Schätzung der stochastischen Matrizen P und S und der gegebenen emittierten Folgen $\vec{Y}$ (der Trainingssequenzen) neue Schätzer für P und S generieren, indem wir versuchen, die entsprechenden geschätzten Wahrscheinlichkeiten mit den Trainingssequenzen zu adjustieren.

Diese Methode wird in der Literatur als *Baum-Welch-Algorithmus* bezeichnet. Die algorithmische Idee ist in Abbildung 8.3 noch einmal skizziert.

1) Wähle Wahrscheinlichkeiten S und P;

2) Bestimme die erwartete relative Häufigkeit $p'_{q,q'}$ von Zustandsübergängen (q, q') und die erwartete relative Emissionshäufigkeit $s'_{q,a}$ unter der Annahme von P und S.

3) War die Verbesserung von (P, S) zu (P', S') eine deutliche Verbesserung so setzte $P = P'$ und $S = S'$ und gehe zu Schritt 2). Ansonsten gib P' und S' als Lösung aus.

Abbildung 8.3: Algorithmus: Skizze des Baum-Welch-Algorithmus

8.4.3 Baum-Welch-Algorithmus als EM-Methode

Der Baum-Welch-Algorithmus ist ein Spezialfall der Erwartungswert-Maximierungs-Methode oder kurz EM-Methode. Auch hier versuchen wir aus einem Schätzer der Wahrscheinlichkeiten und den Trainingssequenzen, den Erwartungswert der gegebenen Trainingssequenzen unter den geschätzten Wahrscheinlichkeiten zu bestimmen und anschließend den Schätzer zu verbessern, so dass sie den Erwartungswert in Richtung der beobachteten Werte zu maximieren.

Wir wollen die Parameter P bzw. S der Zustandsübergangs- bzw. Emissionswahrscheinlichkeiten schätzen, die wir im Folgenden kurz $\Psi := (P, S)$ nennen wollen. Dazu verwenden wir wieder die Maximum-Likelihood-Methode Methode, d.h. wir versuchen Ψ so zu wählen, dass $\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]$ maximal wird. Wir wissen jedoch zusätzlich, dass die Folgen $\vec{y}$ mit Hilfe von Zustandsfolgen $\vec{X}$ im Hidden Markov Model generiert werden. Daher wissen wir mit Hilfe des Satzes der totalen Wahrscheinlichkeit und der Definition der bedingten Wahrscheinlichkeit, dass

$$\begin{aligned}\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}] &= \sum_{\vec{x} \in Q^{|\vec{y}|}} \mathrm{Ws}_\Psi[\vec{Y} = \vec{y} \mid \vec{X} = \vec{x}] \cdot \mathrm{Ws}_\Psi[\vec{X} = \vec{x}] \\ &= \sum_{\vec{x} \in Q^{|\vec{y}|}} \mathrm{Ws}_\Psi[\vec{Y} = \vec{y}, \vec{X} = \vec{x}],\end{aligned}$$

wobei $\vec{x} \in Q^{|\vec{y}|}$ wieder Zustandsfolgen im gegebenen Hidden Markov Model sind.

Für eine algorithmisch besser zugängliche Darstellung von $\mathrm{Ws}_\Psi[\vec{Y} = \vec{y} \mid \vec{X} = \vec{x}]$ bestimmen wir eine weitere Beziehung. Mit Hilfe der Definition der bedingten Wahrscheinlichkeit gilt:

$$\mathrm{Ws}_\Psi[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}] = \frac{\mathrm{Ws}_\Psi[\vec{X} = \vec{x}, \vec{Y} = \vec{y}]}{\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]}.$$

Durch Umformung und Logarithmieren erhalten wir

$$\log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]) = \log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}, \vec{X} = \vec{x}]) - \log(\mathrm{Ws}_\Psi[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}]). \quad (8.5)$$

Sei im Folgenden Ψ_i der Parametersatz, den wir im i-ten Iterationsschritt ausgewählt haben. Wir multiplizieren jetzt beiden Seiten der Gleichung 8.5 mit dem Term $\mathrm{Ws}_{\Psi_i}[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}]$ und summieren über alle Zustandfolgen $\vec{x} \in Q^{|\vec{y}|}$. Wir erhalten dann:

$$\begin{aligned}\log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]) &= \underbrace{\sum_{\vec{x} \in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}]}_{=1} \cdot \log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]) \\ &= \sum_{\vec{x} \in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}] \cdot \log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}, \vec{X} = \vec{x}]) \\ &\quad - \sum_{\vec{x} \in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}] \cdot \log(\mathrm{Ws}_\Psi[\vec{X} = \vec{x} \mid \vec{Y} = \vec{y}]).\end{aligned}$$

Anstatt nun $\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}]$ zu maximieren können wir aber auch $\log(\mathrm{Ws}_\Psi[\vec{Y} = \vec{y}])$ maximieren, da der Logarithmus ein streng monoton wachsende Funktion ist. Somit müssen beide Maxima an derselben Stelle angenommen werden, d.h. für dieselbe Wahl von Ψ.

Wir werden jetzt iterativ vorgehen und versuchen, stattdessen ein neues Ψ so zu wählen, dass $\log(\mathrm{Ws}_{\Psi}[\vec{Y}=\vec{y}]) > \log(\mathrm{Ws}_{\Psi_i}[\vec{Y}=\vec{y}])$ wird und setzen dieses als Ψ_{i+1}. Damit suchen wir also nach einem Ψ mit

$$\begin{aligned}
&\sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log(\mathrm{Ws}_{\Psi}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]) \\
&- \sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log(\mathrm{Ws}_{\Psi}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}]) \\
&\qquad > \sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log(\mathrm{Ws}_{\Psi_i}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]) \\
&\qquad\quad - \sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log(\mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}]).
\end{aligned}$$

Dies ist äquivalent dazu, dass die folgende Ungleichung gilt:

$$\begin{aligned}
&\sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log\left(\frac{\mathrm{Ws}_{\Psi}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]}{\mathrm{Ws}_{\Psi_i}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]}\right) \\
&\qquad + \sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log\left(\frac{\mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}]}{\mathrm{Ws}_{\Psi}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}]}\right) > 0.
\end{aligned}$$

In dieser Ungleichung ist der zweite Term auf der linken Seite wieder die *Kullback-Leibler-Distanz* und dieses ist stets nichtnegativ. Es genügt daher ein Ψ zu finden, so dass

$$\sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log\left(\frac{\mathrm{Ws}_{\Psi}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]}{\mathrm{Ws}_{\Psi_i}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]}\right) > 0$$

Definieren wir

30.01.20

$$\Delta(\Psi,\Psi_i) := \sum_{\vec{x}\in Q^{|\vec{y}|}} \mathrm{Ws}_{\Psi_i}[\vec{X}=\vec{x} \mid \vec{Y}=\vec{y}] \log\left(\mathrm{Ws}_{\Psi}[\vec{Y}=\vec{y},\vec{X}=\vec{x}]\right),$$

so wollen wir als ein Ψ mit

$$\Delta(\Psi,\Psi_i) - \Delta(\Psi_i,\Psi_i) > 0.$$

finden. Da für $\Psi = \Psi_i$ der Term $\Delta(\Psi,\Psi_i) - \Delta(\Psi_i,\Psi_i)$ zumindest gleich Null wird, können wir zumindest einen nichtnegativen Wert finden. Es genügt also, Ψ so zu wählen, dass der $\Delta(\Psi,\Psi_i)$ maximal wird ($\Delta(\Psi_i,\Psi_i)$ ist ja konstant), d.h. wir wählen Ψ mittels

$$\Psi_{i+1} := \mathrm{argmax}\left\{\Delta(\Psi,\Psi_i) \;:\; \Psi\right\}.$$

Somit wächst die Folge in jedem Schritt und da der Zuwachs beim Optimum aufhören muss, konvergiert diese Methode zumindest gegen ein lokales Maximum.

Wir bemerken jetzt noch, dass der Ausdruck $\Delta(\Psi, \Psi_i)$ der Erwartungswert der Variablen $\log(\mathrm{Ws}_\Psi[\vec{Y} \wedge \vec{X}]$ unter der Verteilung $\mathrm{Ws}_{\Psi_i}[\vec{X} \mid \vec{Y}]$ ist. Somit wird auch der Name Erwartungswert-Maximierungs-Methode klar, da wir zuerst einen Erwartungswert bestimmen und dann mit Hilfe einer Maximierung die neuen Verteilungen wählen.

8.5 Mehrfaches Sequenzen-Alignment mit HMM

Wir wollen jetzt noch einmal auf das mehrfache Sequenzen Alignment aus dem vierten Kapitel zurückgreifen und versuchen dieses mit Hilfe von Hidden Markov Models zu lösen.

8.5.1 Profile

Zunächst einmal benötigen wir noch eine Definition, was wir unter einem Profil verstehen wollen.

Definition 8.6 *Ein* Profil $\mathcal{P}$ *der Länge* ℓ *über ein Alphabet* Σ *ist eine Matrix* $(e_i(a))_{i\in[1:\ell],a\in\Sigma}$ *von Wahrscheinlichkeiten, wobei* $e_i(a)$ *die Wahrscheinlichkeit angibt, dass an der Position* i *ein* a *auftritt.*

Ein Profil ist also eine Wahrscheinlichkeitsverteilung von Buchstaben über ℓ Positionen. Wenn wir schon ein mehrfaches Sequenzen-Alignment hätten, dann könnte man an den einzelnen Positionen die Verteilung der Buchstaben bestimmen, das man dann als Profil interpretieren kann.

Sei $y \in \Sigma^\ell$, dann ist für ein Profil $\mathcal{P}$ die Wahrscheinlichkeit, dass es die Folge y generiert hat:

$$\mathrm{Ws}_{\mathcal{P}}[Y = y] = \prod_{i=1}^{\ell} e_i(y_i).$$

Wir werden daher den Score eines Wortes $y \in \Sigma^\ell$ wieder über den Logarithmus des Bruchs der Wahrscheinlichkeit dieser Zeichenreihe unter dem Profil $\mathcal{P}$ und der Wahrscheinlichkeit, wenn die Wahrscheinlichkeiten der einzelnen Zeichen unabhängig von der Position ist:

$$\mathrm{Score}(y \mid \mathcal{P}) = \sum_{i=1}^{\ell} \log\left(\frac{e_i(y_i)}{p(y_i)}\right),$$

wobei p eine Wahrscheinlichkeitsverteilung ist, die angibt, mit welcher Wahrscheinlichkeit $p(a)$ ein Zeichen $a \in \Sigma$ (unabhängig von der Position) auftritt.

Wir konstruieren jetzt ein Hidden Markov Model $M = (Q, \Sigma, P, S)$ für ein gegebenes Profil $\mathcal{P}$. Dazu führen wir für jede Position einen Zustand ein, sowie zwei Hilfszustände am Anfang und am Ende, die linear miteinander verbunden sind (s.a. Abbildung 8.4)

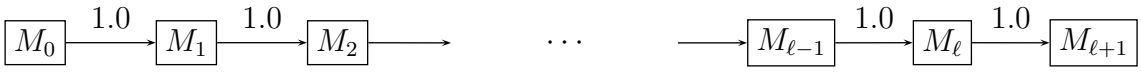


Abbildung 8.4: Skizze: HMM für ein gegebenes Profil

Für die Zustandsübergangswahrscheinlichkeiten gilt offensichtlich $p_{i,j} = 1$, wenn $j = i+1$ und 0 sonst. Die Emissionswahrscheinlichkeiten sind natürlich $s_{M_i,a} = e_i(a)$ für $i \in [1 : \ell]$. Die Zustände M_0 und $M_{\ell+1}$ widersprechen eigentlich der Definition eines Hidden Markov Models, da sie keine Zeichen ausgeben. Daher müssen wir unser Modell noch etwas erweitern.

Definition 8.7 *Ein Zustand eines Hidden Markov Models, der keine Zeichen emittiert heißt* stiller Zustand *(engl.* silent state*).*

Somit sind die Zustände M_0 und $M_{\ell+1}$ unseres Hidden Markov Models für ein gegebenes Profil der Länge ℓ stille Zustände.

8.5.2 Erweiterung um InDel-Operationen

Ein Durchlauf durch das Profil-HMM entspricht einem Alignment gegen das Profil (was man als Verteilung des Konsensus-Strings ansehen kann). Was ist aber mit den InDel-Operationen, die bei Alignments essentiell sind? Wir werden jetzt das Hidden Markov Model für solche Operationen erweitern.

Für die Einfügungen zwischen zwei Zeichen des Konsensus-Strings führen wir jetzt neue, weitere Zustände ein, die diese modellieren sollen. Dies ist in Abbildung 8.5 illustriert.

Nun sind die Zustandsübergangswahrscheinlichkeiten wieder offen. Wir werden später darauf zurückkommen, wie wir diese und die Emisisionswahrscheinlichkeiten bestimmen können. Letzte können auch allgemein durch $s_{I_i,a} = p(a)$ gegeben sein.

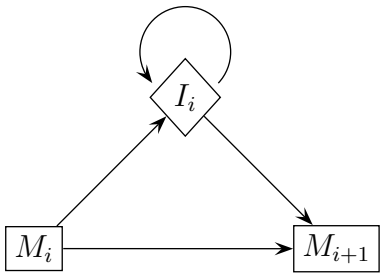


Abbildung 8.5: Skizze: Einbau von Insertionen

Betrachten wir jetzt noch eine Einfügung von $k > 0$ Zeichen $(a_1, \ldots, a_k)$ zwischen zwei Matching-Zuständen, d.h. der Zustandsfolge $(M_i, I_i, \ldots, I_i, M_{i+1})$. Was bedeutet dies für den Score für die $k > 0$ Einfügungen? Der additive Teil für den Score beträgt dann:

$$\log\left(p_{M_i,I_i} \cdot \prod_{j=1}^{k} \left(p_{I_i,I_i} \cdot s_{I_i,a_j}\right) \cdot p_{I_i,M_{i+1}}\right)$$

$$= \underbrace{\log(p_{M_i,I_i} \cdot p_{I_i,M_{i+1}})}_{\text{Gap}-\text{Eröffnungs}-\text{Strafe}} + \underbrace{k \cdot p_{I_i,I_i} + \sum_{j=1}^{k} \log(s_{I_i,a_j})}_{\text{Gap}-\text{Verlängerungs}-\text{Strafe}}.$$

Man sieht, dass hier die Gaps gemäß einer affinen Lücken-Strafe bewertet werden. Allerdings hängen hier die Konstanten von der Position (und ggf. von den emittierten Symbolen) ab, d.h. sie sind nicht notwendigerweise für alle Lücken gleicher Länge gleich.

Des Weiteren müssen wir noch die Deletionen modellieren. Eine einfache Möglichkeit ist, weitere Möglichkeiten der Übergänge von M_i nach M_j für $i < j$ einzufügen, wie dies in Abbildung 8.6 schematisch dargestellt ist.

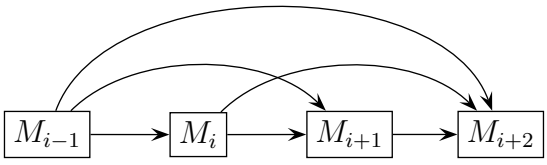


Abbildung 8.6: Skizze: Einbau von Deletionen

Wir werden jetzt noch eine andere Modellierung vorstellen, die später eine effizientere Berechnung der Vorwärts- und Rückwärtswahrscheinlichkeiten erlaubt. Hierfür ist die Anzahl der Pfeile, die in einem Zustand endet entscheidend. Im obigen Modell können dies bis zu ℓ Pfeile sein, wenn wir ein Profil der Länge ℓ zugrunde legen.

Wir werden jetzt ℓ spezielle Zustände einführen, die die Deletionen an den ℓ Positionen modellieren werden. Diese Zustände kann man als Bypässe für das entsprechende Zeichen interpretieren. Die genaue Anordnung dieser Zustände ist in Abbildung 8.7 illustriert.

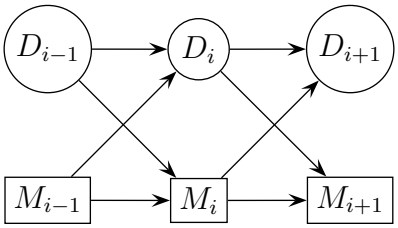


Abbildung 8.7: Skizze: Einbau von Deletions-Zuständen

Hierbei ist jetzt wichtig, dass diese Deletions-Zustände auch stille Zustände sind, da ja beim Löschen nichts ausgegeben werden kann. Wir wollen hier noch anmerken, dass wir mit Hilfe von Deletions-Zuständen nicht alle Verteilungen modellieren können, die mit Hilfe von Deletionsübergängen ohne zusätzliche Zustände möglich sind.

Auch hier wollen wir uns jetzt wieder den resultierenden Score für eine Folge von k Deletionen $(M_i, D_{i+1}, \ldots, D_{i+k}, M_{i+k+1})$ mit $k > 0$ genauer betrachten:

$$\log\left(p_{M_i,D_{i+1}} \cdot \prod_{j=1}^{k-1} p_{D_{i+j},D_{i+j+1}} \cdot p_{D_{i+k},M_{i+k+1}}\right)$$
$$= \underbrace{\log(p_{M_i,D_{i+1}} \cdot p_{D_{i+k},M_{i+k+1}})}_{\text{Gap-Eröffnungs-Strafe}} + \underbrace{\sum_{j=1}^{k-1} \log(p_{D_{i+j},D_{i+j+1}})}_{\text{Gap-Verlängerungs-Strafe}}.$$

Auch hier haben wir im Wesentlichen affine Lücken-Strafen. Allerdings werden die Deletionen unterschiedich bewertet, was natürlich von den entsprechenden Wahrscheinlichkeiten und auch der Position abhängt.

In der Abbildung 8.8 ist jetzt ein vollständiges Hidden Markov Model für eine Sequenz der Länge 4 angegeben. Man beachte, dass die fehlenden Pfeile für die Wahrscheinlichkeit 0 für die entsprechenden Zustandsübergangswahrscheinlichkeiten stehen. Wie wir in diesem vollständigen Beispiel eines Hidden Markov Models für ein Profil der Länge 4 sehen, hat jeder Zustand nur maximal 3 eingehende Pfeile. Somit können wir die Vorwärts- und Rückwärtswahrscheinlichkeiten viel einfacher berechnen, da wir nur jeweils maximal vier mögliche vorherige Zustände und nicht bis zu 3ℓ verschiedenen berücksichtigen müssen. Daher können wir diese in Zeit $O(|Q| \cdot \ell)$ sehr effizient bestimmen.

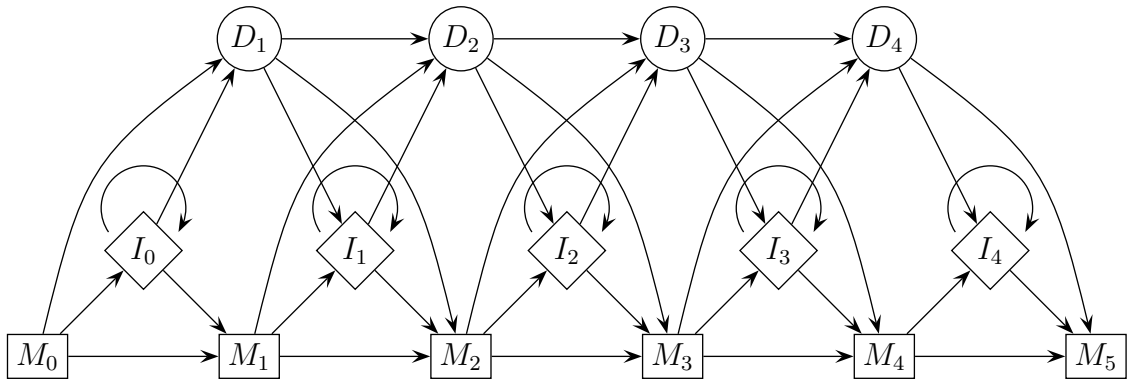


Abbildung 8.8: Skizze: Ein vollständiges HMM für $\ell = 4$

8.5.3 Alignment gegen ein Profil-HMM

Um ein mehrfaches Sequenzen-Alignment zu konstruieren, berechnen wir einen wahrscheinlichsten Pfad durch das zugehörige Hidden Markov Model, das wir eben konstruiert haben. Der Pfad gibt uns dann für jede Sequenz die entsprechende Alignierung an. Zu Beginn versuchen wir für ein gegebenes Profil ein HMM zu konstruieren und dazu neue Sequenzen zu alignieren.

Wir merken an dieser Stelle noch an, dass wir hierfür die Zustandsübergangswahrscheinlichkeiten und die Emissionswahrscheinlichkeiten benötigen. Wir werden uns später genauer überlegen, wie wir diese aus den gegebenen Sequenzen abschätzen können.

Für die Berechnung des wahrscheinlichsten Pfades durch unser Profil-HMM definieren wir die folgenden Scores analog zum Viterbi-Algorithmus, wobei ℓ die Länge des Profils und $i \in [1 : |Y|]$ (Y sei die gegebene Sequenz):

$$\begin{aligned} M_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i), \text{ der in } M_j \text{ endet} \\ D_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i), \text{ der in } D_j \text{ endet} \\ I_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i), \text{ der in } I_j \text{ endet} \end{aligned}$$

Hierbei ist der Score wieder die Log-Odds-Ratio der Wahrscheinlichkeiten der gegebenen Sequenz, die durch das Profil bzw. durch eine uniforme Verteilung p der Buchstaben beschrieben ist.

Wie man sich leicht überlegt, kann man die beim Viterbi-Algorithmus verwendeten Rekursionsgleichungen für unser Profil-HMM wie folgt vereinfachen. Für die Matching-Zustände erhalten wir:

$$M_j(i+1) = \log\left(\frac{s_{M_j, y_{i+1}}}{p_{y_{i+1}}}\right) + \max\left\{ \begin{array}{c} M_{j-1}(i) + \log(p_{M_{j-1}, M_j}), \\ I_{j-1}(i) + \log(p_{I_{j-1}, M_j}), \\ D_{j-1}(i) + \log(p_{D_{j-1}, M_j}) \end{array} \right\}.$$

Wenn man die $\log(p_{y_{i+1}})$ aus den Logarithmen additiv herausrechnet, erhält man den Term $\sum_{k=1}^{\ell} \log(p_{y_k}) = \log\left(\prod_{k=1}^{\ell} p_{y_k}\right)$. Dieser hängt vom HMM gar nicht ab, sondern nur von y, d.h. es wird wie beim Viterbi-Algorithmus die wahrscheinlichste Zustandsfolge berechnet.

Für die Insertions-Zustände erhalten wir:

$$I_j(i+1) = \log\left(\frac{s_{I_j,y_{i+1}}}{p_{y_{i+1}}}\right) + \max\left\{\begin{array}{c} M_j(i) + \log(p_{M_j,I_j}), \\ I_j(i) + \log(p_{I_j,I_j}), \\ D_j(i) + \log(p_{D_j,I_j}) \end{array}\right\}.$$

Wählt man für die Wahrscheinlichkeiten der eingefügten Symbole $s_{I_j,a} = p_a$, dann fällt der additive logarithmische Term weg und die Bestrafung für die eingefügten Symbole hängt nur von der Position ab.

Für die Deletions-Zustände erhalten wir:

$$D_j(i+1) = \max\left\{\begin{array}{c} M_{j-1}(i+1) + \log(p_{M_{j-1},D_j}), \\ I_{j-1}(i+1) + \log(p_{I_{j-1},D_j}), \\ D_{j-1}(i+1) + \log(p_{D_{j-1},D_j}) \end{array}\right\}.$$

Für die Anfangsbedingung gilt offensichtlich $M_0(0) = 1$. Alle undefinierten Werte sind als $-\infty$ zu interpretieren. Was wir letztendlich suchen ist der Wert $M_{\ell+1}(|Y|)$, der den Score der Sequenz, der die Log-Odds-Ration gemäß des Profils gegen eine zufällige Verteilung angibt.

Theorem 8.8 *Sei $M = (Q, \Sigma, P, S)$ ein Hidden Markov Model für ein mehrfaches Sequenzen-Alignment der Länge ℓ. Für eine gegebene Sequenz Y lässt sich eine wahrscheinlichste zugehörige Zustandsfolge in Zeit $O(|Q| \cdot (\ell + |Y|))$ und mit Platz $O(|Q|)$ berechnen.*

Mit Hilfe des obigen Theorems können wir jetzt auch ein mehrfaches Sequenzen-Alignment gegen ein gegebenes Profil konstruieren. Wir berechnen zuerst für jede Folge mit Hilfe des Viterbi-Algorithmus die wahrscheinlichste zugehörige Zustandsfolge durch das Hidden Markov Models. Anhand dieser Zustandsfolge können wir aus den Zuständen das Alignment der zugehörigen Zeichenreihe ablesen.

Als einziges Problem bleibt hierbei natürlich noch die Bestimmung der stochastischen Matrizen P und S. Diese können wir wiederum mit Hilfe der Erwartungswert-Maximierungs-Methode, speziell dem Baum-Welch-Agorithmus schätzen. Aufgrund der speziellen Gestalt des hier verwendeten Hidden Markov Models für das mehrfache Sequenzen-Alignment ergibt sich für die Vorwärts- und Rückwärtswahrschein-

lichkeiten folgende einfachere Rekursionsgleichungen:

$$\begin{aligned} f_{M_j}(i) &:= \text{Ws[Ausgabe ist } (y_1,\ldots,y_i) \text{ und Endzustand ist } M_j] \\ f_{I_j}(i) &:= \text{Ws[Ausgabe ist } (y_1,\ldots,y_i) \text{ und Endzustand ist } I_j] \\ f_{D_j}(i) &:= \text{Ws[Ausgabe ist } (y_1,\ldots,y_i) \text{ und Endzustand ist } D_j] \end{aligned}$$

$$\begin{aligned} b_{M_j}(i) &:= \text{Ws[Ausgabe ist } (y_{i+1},\ldots,y_{|Y|}) \mid \text{Anfangszustand ist } M_j] \\ b_{I_j}(i) &:= \text{Ws[Ausgabe ist } (y_{i+1},\ldots,y_{|Y|}) \mid \text{Anfangszustand ist } I_j] \\ b_{D_j}(i) &:= \text{Ws[Ausgabe ist } (y_{i+1},\ldots,y_{|Y|}) \mid \text{Anfangszustand ist } D_j] \end{aligned}$$

Mit Hilfe dieser Vorwärts- und Rückwärtswahrscheinlichkeiten können wir wieder mit der Baum-Welch Methode neue, verbesserte Zustandsübergangs- und Emissionswahrscheinlichkeiten aus dem bestehenden Modell berechnen. Wir überlassen die Details der Implementierung dem Leser und halten nur noch ein hierzu benötigtes Ergebnis fest.

Lemma 8.9 *Die Vorwärts- und Rückwärtswahrscheinlichkeiten für ein Hidden Markov Model $M = (Q, \Sigma, P, S)$ für ein mehrfaches Sequenzen-Alignment der Länge ℓ können in Zeit $O(|Q| \cdot \ell)$ berechnet werden.*

Wenn wir für eine Menge von Sequenzen $\vec{y} = (y^{(1)}, \ldots, y^{(m)})$ jetzt allgemein ein mehrfaches Sequenzen-Alignment berechnen wollen, müssen wir zuerst den Parameter ℓ des Hidden Markov Models festlegen, das ist die Anzahl der eigentlichen (nichtstillen) Match-Zustande. Diese kann beispielsweise als die mittele Sequenzlänge von $\vec{y}$ gewählt werden. Dann trainieren wir mit den Sequenzen $\vec{y}$ mithilfe des Baum-Welch-Algorithmus die Zustandsübergangs- und Emissionswahrscheinlichkeiten des Hidden Markov Models. Letztendlich berechnen wir dann wider den wahrscheinlichsten Pfad jeder Sequenz mithilfe des Viterbi-Algorithmus (oder des vorhin angepassten Scoring-Algorithmus) durch dieses Hidden Markov Model. Durch die Zustandsfolgen wissen wir nun für jedes Zeichen der Sequenz, ob Sie durch ein Match-, Insertion- oder Deletion-Zustand gelaufen sind und fügen die Sequenzen im Wesentlichen gemäß der Match-Zustande zu einem mehrfachen Sequenz-Alignment zusammen. Zeichen, die zu einem Insertions-Zustand gehören, werden in neuen Spalten aufgenommen, so dass die anderen Sequenzen mit Gaps gefüllt werden (außer sie enthalten an dieser Stelle auch Insertionen). Deletionszustände führen zu Gaps in der Sequenz beim Alignieren mit den anderen Sequenzen an den entsprechenden Match-Positionen.

04.02.20

9 Fragment Assembly (*)

9.1 Sequenzierung ganzer Genome

In diesem Kapitel wollen wir uns mit algorithmischen Problemen beschäftigen, die bei der Sequenzierung ganzer Genome (bzw. einzelner Chromosome) auftreten. Im ersten Kapitel haben wir bereits biotechnologische Verfahren hierzu kennen gelernt. Nun geht es um die informatischen Methoden, um aus kurzen sequenzierten Fragmenten die Sequenz eines langen DNS-Stückes zu ermitteln.

9.1.1 Shotgun-Sequencing

Trotz des rasanten technologischen Fortschritts ist es nicht möglich, lange DNS-Sequenzen im Ganzen biotechnologisch zu sequenzieren. Selbst mit Hilfe großer Sequenzierautomaten lassen sich nur Sequenzstücke der Länge von etwa 500-1000 Basenpaaren sequenzieren. Wie lässt sich dann allerdings beispielsweise das ganze menschliche Genom mit etwa drei Milliarden Basenpaaren sequenzieren?

Eine Möglichkeit ist das so genannte *Shotgun-Sequencing*. Hierbei werden lange Sequenzen in sehr viele kurze Stücke aufgebrochen. Dabei werden die Sequenzen in mehrere Klassen aufgeteilt, so dass eine Bruchstelle in einer Klasse mitten in den Fragmenten der anderen Sequenzen einer anderen Klasse liegt (siehe auch Abbildung 9.1).

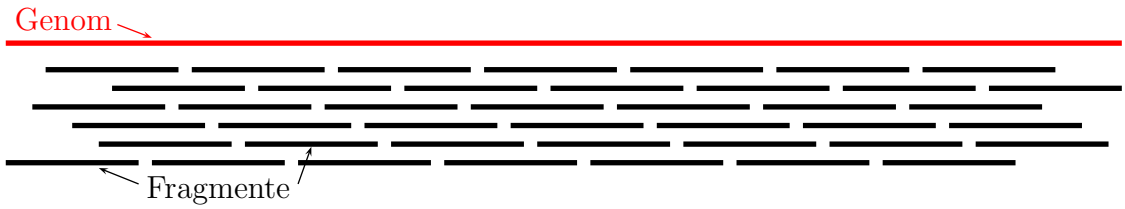


Abbildung 9.1: Skizze: Shotgun-Sequencing

Die kurzen Sequenzen können jetzt direkt automatisch sequenziert werden. Es bleibt nur das Problem, aus der Kenntnis der kurzen Sequenzen wieder die lange DNS-Sequenz zu rekonstruieren. Dabei hilft, dass einzelne Positionen (oder vielmehr kurze DNS-Stücke) von mehreren verschiedenen Fragmenten, die an unterschiedlichen Positionen beginnen, überdeckt werden. Man muss also nur noch die Fragment

wie in einem Puzzle-Spiel so anordnen, dass überlappende Bereiche möglichst gleich sind.

9.1.2 Sequence Assembly

Damit ergibt sich für das Shotgun-Sequencing die folgende prinzipielle Vorgehensweise:

Overlap-Detection: Zuerst bestimmen wir für jedes Paar von zwei Fragmenten, wie gut diese beiden überlappen, d.h. für eine gegebene Menge $S = \{s_1, \ldots, s_k\}$ von k Fragmenten bestimmen wir für alle $i, j \in [1:k]$ die beste Überlappung $d(s_i, s_j)$ zwischen dem Ende von s_i und dem Anfang von s_j (siehe Abbildung 9.2).

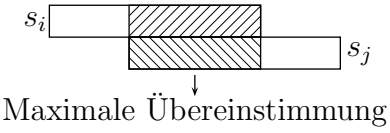


Abbildung 9.2: Skizze: Overlap-Detection

Fragment-Layout: Dann müssen die Fragmente so angeordnet werden, dass die Überlappungen möglichst gleich sind. Mit diesem Problem werden wir uns in diesem Kapitel hauptsächlich beschäftigen.

Konsensus-String: Zum Schluss interpretieren wir das Fragment Layout wie ein mehrfaches Sequenzen Alignment und bestimmen den zugehörigen Konsensus-String, den wir dann als die ermittelte Sequenz für die zu sequenzierende Sequenz betrachten.

In den folgenden Abschnitten gehen wir auf die einzelnen Schritte genauer ein. Im Folgenden werden wir mit $S = \{s_1, \ldots, s_k\}$ die Menge der Fragmente bezeichnen. Dabei werden in der Praxis die einzelnen Fragmente ungefähr die Länge 500 haben. Wir wollen ganz allgemein mit $n = \lfloor \frac{1}{k} \sum_{i=1}^{k} |S_i| \rfloor$ die mittlere Länge der Fragmente bezeichnen. Wir wollen annehmen, dass für alle Sequenzen in etwa gleich lang sind, also $|s_i| = \Theta(n)$ für alle $i \in [1:k]$. In der Praxis gilt hierbei, dass kn in etwa 5 bis 10 Mal so groß ist wie die Länge der zu sequenzierenden Sequenz, da wir bei der Generierung der Fragmente darauf achten werden, dass jede Position der zu sequenzierenden Sequenz von etwa 5 bis 10 verschiedenen Fragmenten überdeckt wird.

9.2 Overlap-Detection

Zuerst wollen wir uns mit der Overlap-Detection beschäftigen. Wir wollen hier zwei verschiedene Alternativen unterscheiden, je nachdem, ob wir Fehler zulassen wollen oder nicht.

9.2.1 Overlap-Detection mit Fehlern

Beim Sequenzieren der Fragmente treten in der Regel Fehler auf, so dass man damit auch rechnen muss. Ziel wird es daher sein, ein Suffix von s_i zu bestimmen, der ziemlich gut mit einem Präfix von s_j übereinstimmt. Da wir hierbei Sequenzierfehler zulassen, entspricht dies im Wesentlichen einem semi-globalen Alignment (siehe Abbildung 9.3).

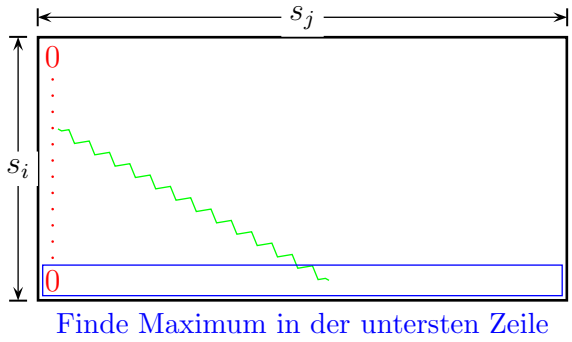


Abbildung 9.3: Semi-globale Alignments mit Ähnlichkeitsmaßen

Hierbei ist zu beachten, dass Einfügungen zu Beginn von s_i und Löschungen am Ende von s_j nicht bestraft werden. Für den Zeitbedarf gilt (wie wir ja schon gesehen haben) $O(n^2)$ pro Paar. Da wir $k^2 - k$ verschiedene Paare betrachten müssen, ergibt dies insgesamt eine Laufzeit von: $O(k^2n^2)$.

Theorem 9.1 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen mit $|s_i| = \Theta(n)$ für alle $i \in [1:k]$. Die längsten Überlappung für jedes Paar (s_i, s_j) für alle $i, j \in [1:k]$ kann in Zeit $O((kn)^2)$ berechnet werden.*

9.2.2 Overlap-Detection ohne Fehler

Wir wollen jetzt noch eine effizientere Variante vorstellen, wenn wir keine Fehler zulassen. Dazu verwenden wir wieder einmal Suffix-Bäume.

Wir konstruieren zuerst einen verallgemeinerten Suffix-Baum für $S = \{s_1, \ldots, s_k\}$. Wie wir schon gesehen haben, ist der Platzbedarf $\Theta(kn)$ und die Laufzeit der Konstruktion $O(kn)$. Dieser Suffix-Baum ist in Abbildung 9.4 schematisch dargestellt.

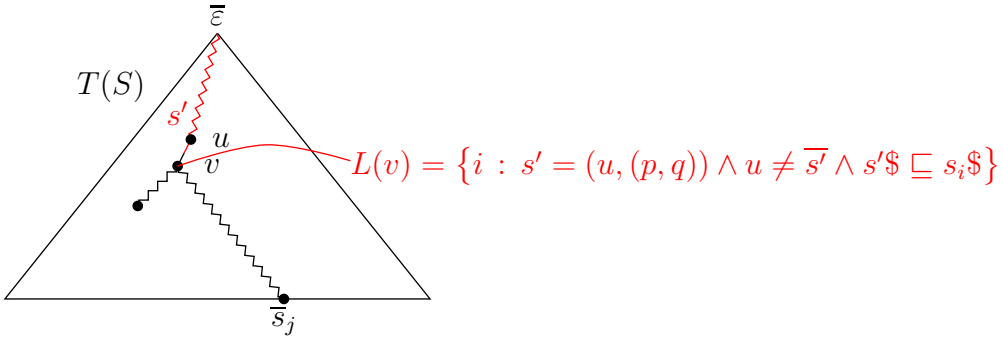


Abbildung 9.4: Skizze: Verallgemeinerter Suffix-Baum für $S = \{s_1, \ldots, s_k\}$

Für jeden Knoten v des Suffix-Baumes mit Elter u generieren wir eine Liste $L(v)$, die wie folgt definiert ist:

$$L(v) := \left\{ i \in [1:k] \ : \ \exists j \in [1:|s_i|] : \ s_{i,j} \cdots s_{i,|s_i|} = (u,(p,q)) \right\},$$

wobei $(u,(p,q))$ eine kanonische Referenz ist. In der Liste $L(v)$ befinden sich also alle Indizes i, so dass ein Suffix von s_i im Knoten v oder auf der Kante zum Knoten v endet. Genau genommen ist die obige Definition nicht ganz korrekt, da hier der Knoten u statt v berücksichtigt wird. Um komplexere Definitionen zu vermeiden, ist dies alles etwas lockerer zu verstehen.

Betrachten wir also jetzt einen Knoten v im verallgemeinerten Suffix-Baum mit seiner Liste $L(v)$, wie in Abbildung 9.4 illustriert. Sei dabei s' die Zeichenfolge, mit der der Knoten s' erreicht wird. Dann gilt offensichtlich:

- s' ist Suffix von s_i für alle $i \in L(v)$,
- s' ist Präfix von s_j für alle j, so dass $\overline{s}_j$ ein Blatt im vom v gewurzelten Teilbaum ist.

Der längste Suffix-Präfix-Match von s_i mit s_j ist damit im Wesentlichen durch den tiefsten Knoten v mit $i \in L(v)$ auf einem Pfad von $\overline{\varepsilon}$ zu $\overline{s}_j$ gegeben, genau genommen durch die größte Stringstiefe einer kanonischen Referenz in $L(v)$.

Überlegen wir uns jetzt, wie man diese Listen effizient erstellen kann. Für alle $s_i \in S$ tun wir das Folgende. Starte an $\overline{s}_i$ und folge den Suffix-Links bis zur Wurzel (implizites Durchlaufen aller Suffixe von s). Für jeden besuchten Knoten v füge i in die Liste

eine: $L(v) := L(v) \cup \{i\}$. Die Kosten hierfür entsprechen der Anzahl der Suffixe von s_i, dies sind $|s_i| + 1 = O(n)$ viele. Für alle $s \in S$ mit $|S| = k$ ist dann der Zeitbedarf insgesamt $O(kn)$.

Hier muss man genau genommen auch wieder etwas aufpassen. Zum einen haben Blätter keine Suffix-Links, so dass wir mit kanonischen Referenzen arbeiten müssen, für deren innere Knoten die Suffix-Links definiert sind. Dabei müssen nach dem Folgen eines Suffix-Links die Referenzen auch wieder kanonisiert werden. Zum anderen werden wieder nicht nur Knoten, sondern auch Positionen auf Kanten (also kanonische Referenzen hierfür) besucht und betrachtet.

Eine alternative, einfachere Möglichkeit besteht darin, wenn man den verallgemeinerten Suffix-Baum als Suffix-Baum für die Zeichenreihe $s_1\sharp_1 s_2\sharp_2 \cdots \sharp_{k-1} s_k\sharp_k$ aufbaut. In einem solchen Suffix-Baum wird man alle Teilbäume eliminieren die nur über ein Zeichen $\sharp_i$ für $i \in [1:k]$ erreichbar sind. Nun kann man mit einer Tiefensuche den Baum durchlaufen und an jedem Knoten v diejenigen Indizes $i \in [1:k]$ in die Menge $L(v)$ aufnehmen, die ein Blatt als Kind besitzen, deren Kantenlabel gerade $\sharp_i$ ist. Hierbei muss man darauf achten, dass Knoten mit nur einem Kind entstehen können. Diese müssen dann repariert werden.

Wie finden wir jetzt mit Hilfe dieses verallgemeinerten Suffix-Baumes und den Listen längste Suffix-Präfix-Matches? Für jedes $i \in [1:k]$ legen wir einen Keller Stack[i] an. Wenn wir mit einer Tiefensuche den verallgemeinerten Suffix-Baum durchlaufen, soll Folgendes gelten. Befinden wir uns am Knoten w des verallgemeinerten Suffix-Baumes, dann soll der Stack[i] alle Knoten v (genauer genommen Referenzen) beinhalten, die zum einen Vorfahren von w sind und in denen zum anderen ein Suffix von s_i endet (bzw. auf der Kante zu w).

Wenn wir jetzt eine Tiefensuche durch den verallgemeinerten Suffixbaum durchführen, werden wir für jeden neu aufgesuchten Knoten zuerst die Stacks aktualisieren. Wenn wir nach der Abarbeitung des Knotens wieder im Baum aufsteigen, entfernen wir die Elemente wieder, die wir beim ersten Besuch des Knotens auf die Stacks gelegt haben. Nach Definition einer Tiefensuche müssen sich diese Knoten jetzt wieder oben auf den Stacks befinden.

Sobald wir ein Blatt gefunden haben, das ohne Beschränkung der Allgemeinheit zum String s_j gehört, betrachten wir für jedes $i \in [1:k]$ das oberste Element w vom Stack (sofern eines existiert). Da dies das zuletzt auf den Stack gelegte war, war dieses eines, das den längsten Überlappung von s_i mit s_j ausgemacht hat. Wenn wir für v noch den *String-Tiefe* mitberechnen, entspricht der String-Tiefe einem längsten Overlap. Hier ist die String-Tiefe der Wurzel 0, und die String-Tiefe eines Knoten entspricht der Länge der Zeichenreihe, die man beim Ablaufen des einfachen Pfades von der Wurzel zu diesem Knoten erhält.

Suffix-Prefix-Matches (int[] S)

```
begin
    tree T = T(S) ;                          /* generalized suffix tree T_S */
    stack_of_nodes Stack[k] ;                        /* one for each s_i ∈ S */
    int str_depth[V(T_S)];
    int Overlap[k, k];
    str_depth[ε̄] := 0;
    computeLvalues(T);
    DFS(T, ε̄);
end
```

DFS (tree T, node v)

```
begin
    forall (i ∈ L(v)) do
        Stack[i].push(v);

    if (v = s̄_j) then
        forall (i ∈ [1 : k]) do
            if (not Stack[i].isEmpty()) then
                Overlap(s_i, s_j) := str_depth[Stack[i].top()]};
                // actually the string depth of the refernce, cf. text

    forall ((v, w) ∈ E(T)) do
        str_depth[w] := str_depth[v] + |label(v, w)|;
        DFS(T,w);

    forall (i ∈ L(v)) do
        Stack[i].pop();
end
```

Abbildung 9.5: Algorithmus: modifizierte Depth-First-Search

Damit ergibt sich zur Lösung der in Abbildung 9.5 angegebene Algorithmus. Dabei müsste bei der Aktualisierung der Matrix `Overlap` nicht die String-Tiefe des Knotens v hergenommen werden, sondern die der tiefsten Referenz auf der Kante zum Knoten v. In einer vollständigen Implementierung würde dies in der Liste $L(v)$ mitvermerkt und auch auf dem Stack vermerkt werden. der Einfachheit halber haben wir dies im Pseudo-Code unterlassen.

Kommen wir nun zur Bestimmung der Laufzeit. Für den reinen DFS-Anteil (der grüne Teil) der Prozedur benötigen wir Zeit $O(kn)$. Die Kosten für die Pushs und

Pops auf die Stacks (der schwarze Teil) betragen:

$$\sum_{v \in V(T_S)} |L(v)| = |\left\{t \in \Sigma^* \,:\, \exists s \in S : \exists j \in [1:|s|] : t = s_j \cdots s_{|s|}\right\}| = O(kn),$$

da in der zweiten Menge alle Suffixe von Wörtern aus S auftauchen.

Die Aktualisierungen der Overlaps (der rote Teil) verursachen folgende Kosten. Da insgesamt nur k Knoten eine Sequenz aus S darstellen, wird die äußere if-Anweisung insgesamt genau k mal betreten. Darin wird innere for-Schleife jeweils k mal aufgerufen. Da die Aktualisierung in konstanter Zeit erledigt werden kann, ist der gesamte Zeitbedarf $O(k^2)$.

Theorem 9.2 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen mit $|s_i| = \Theta(n)$ für alle $i \in [1:k]$. Die längsten Überlappung für jedes Paar (s_i, s_j) für alle $i, j \in [1:k]$ kann in Zeit $O(nk + k^2)$ berechnet werden.*

An dieser Stelle sei noch darauf hingewiesen, dass die Rechenzeit im vorherigen Theorem optimal ist, da die Eingabegröße $\Theta(kn)$ und die Ausgabegröße des Problems $\Theta(k^2)$ ist.

9.2.3 Hybrid Overlap-Detection

In der Praxis kann keines der beiden vorgenannte Verfahren angewandt werdem, es wird meist eine Mischform aus beiden angewandt. Dabei wird im ersten Schritt nach exakten Übereinstimmungen gesucht und diese werden in einem zweiten Schritt zu perfekten Suffix-Prefix-Matches erweitert.

Für den ersten Schritt konstruieren wir für die Menge aller Suffixe der Wörter aus S:

$$\hat{S} = \{\hat{s} \,:\, \exists s \in S : \hat{s}\$ \sqsubseteq s\$\}$$

beispielsweise den Suchwort-Baum für den Aho-Corasick-Algorithmus, was in diesem Fall mehr oder weniger einem verallgemeinerten Suffix-Baum für S entspricht. Wir suchen dabei nicht nur nach Suffixen, sondern auch nach Präfixen von Suffixen (also Teilwörtern, wobei wir hier nur an nichtverlängerbaren exakten Übereinstimmungen in zwei Fragmenten interessiert sind.

Wir suchen dann mittels Aho-Corasick (oder einem ähnlichen Verfahren) alle exakten Übereinstimmungen von Teilwörtern aus $\hat{S}$ in einem Fragment s_ℓ. Dann versucht man diese Treffer (wie bei BLAST oder FASTA) zu einem größeren semiglobalen Alignment auszubauen. Für einen potentiellen Hit lässt man dann noch ein

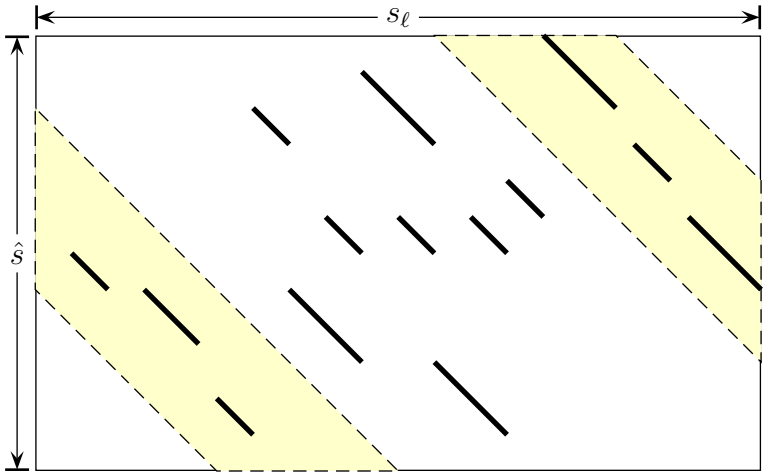


Abbildung 9.6: Skizze: Berechnung geeigneter Alignments für potentielle Kandidaten

beschränktes semiglobales Sequenzen-Alignment um den Treffer laufen, um eine lokale optimale Lösung zu erhalten.

Auf die Details wollen wir an dieser Stelle verzichten und verweisen noch auf die Skizze in Abbildung 9.6. Hierbei sind die exakten Hits schwarze Diagonalen, und die Bereiche eines beschränkten semiglobalen Sequenz-Alignments gelb dargestellt. Wie in den Übungen diskutiert, lassen sich diese beschränkten Alignments in Zeit $O(n)$ konstruieren. Damit beträgt die Gesamtlaufzeit $O(k^2 n)$.

In der ersten Phase können wir statt nach den nicht mehr zu verlängernden exakten Matches von Teilwörtern auch nur nach Teilwörtern mit einer bestimmten Länge suchen. Wie bei BLAST können wir diese Menge auch noch um die zugehörigen dazu ähnlichen Wörter erweitern.

9.3 Fragment Layout

Nachdem wir uns im vorherigen Abschnitt mit der Overlap-Detection beschäftigt haben, kümmern wir uns nun um das Layout der Fragmente.

9.3.1 Layout mit hamiltonschen Pfaden

Wir versuchen nun ein Layout mit Hilfe eines so genannten Overlap-Graphen zu finden. Sei im Folgenden $S = \{s_1, \ldots, s_k\}$ die Menge der Fragmente und $D_{ij} = d(s_i, s_j)$ ein Score für den besten Suffix-Prefix-Overlap von s_i mit s_j. Im fehlerfreien Fall ist

dies die Länge des Overlaps, ansonsten etwa ein Score, der sich beispielsweise aus dem semiglobalen Alignment ergibt.

Definition 9.3 *Der gewichtet, gerichtete Graph (ohne Schleifen)* $G_S = (V, E, \gamma)$ *heißt* Overlap-Graph*, wenn*

- $V = S$,
- $E = V \times V \setminus \{(v, v) : v \in V\}$,
- $\gamma(v, w) = D_i j = d(s_i, s_j)$ *für* $v = s_i$ *und* $w = s_j$.

In unserem Overlap-Graphen werden hamiltonsche Pfade eine besondere Rolle spielen.

Definition 9.4 *Sei* $G = (V, E)$ *ein gerichteter Graph. Ein Pfad (bzw. Kreis)* $(v_1, \ldots, v_\ell)$ *heißt* hamiltonsch*, wenn* $\{v_1, \ldots, v_\ell\} = V$ *und* $|\{v_1, \ldots, v_\ell\}| = |V|$*. Ein Graph heißt* hamiltonsch*, wenn er einen hamiltonschen Pfad besitzt.*

Wir wollen noch anmerken, dass in der Literatur Graphen oft als hamiltonsch bezeichnet werden, wenn sie einen hamiltonschen Kreis enthalten. Die oben angegebene Definition ist für die folgende Diskussion sinnvoller.

Wir werden zeigen, dass jedem Layout L ein hamiltonscher Pfad im zugehörigen Overlap-Graphen entspricht. Betrachten wir dazu die Abbildung 9.7. Der Abfolge der Fragmente im Layout L entspricht offensichtlich einem hamiltonschen Pfad im Overlap-Graphen. Berücksichtigen wir jetzt noch die Gewichte im Overlap-Graphen. Da wir annehmen, dass lange (und damit auch längste) Overlaps nicht durch Zufall entstehen, entspricht die Anordnung der Fragmente im optimalen (d.h. realen) Layout einem hamiltonschen Pfades maximalen Gewichtes, wobei das Gewicht eines Pfades als die Summe seiner Kantengewichte definiert ist.

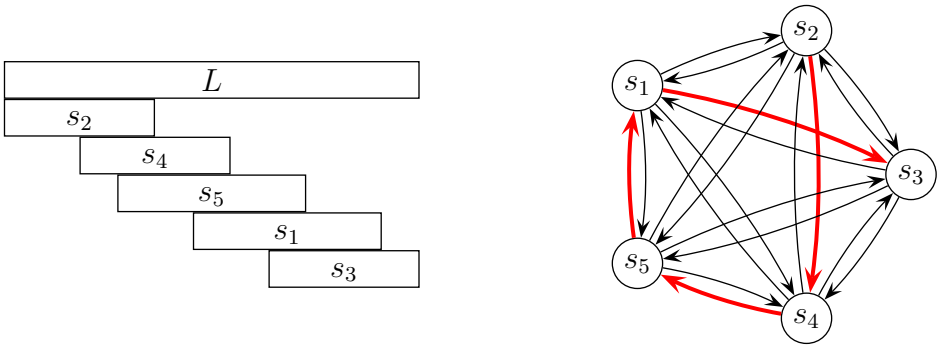


Abbildung 9.7: Skizze: Layout und hamiltonsche Pfade im Overlap-Graphen

Andererseits können wir aus einen hamiltonschen Pfad immer ein Layout konstruieren, indem wir aus den Überlappungender Fragmente ein Layout bilden. Bei der Berücksichtigung von Fehlern, sind an manchem Positionen des Layout die entsprechenden Nukleotide nicht definiert (bzw. als X). Dies ist jedoch eine Aufgabe des dritten Schrittes, der Konstruktion des Konsensus-Strings. Auch hier gilt, dass wir ein optimales (vermutlich kürzestes) Layout finden, wenn wir einen hamiltonschen Pfad maximalen Gewichtes bestimmen (sieh auch hierzu Abbildung 9.7).

Somit lässt sich das Problem zum Fragment Layout auf das Finden gewichtsmaximaler hamiltonscher Pfade im Overlap-Graphen reduzieren. Hierfür gibt es eine Reihe von Ansätzen, wir werden im nächsten Semester vermutlich einen Algorithmus für das so genannte Shortest Superstring Problem kennen lernen. Leider erweist sich das Problem zur Bestimmung gewichtsmaximaler hamiltonscher Pfade in einem gegebenen Graph als algorithmisch nicht leicht.

Wie aus der Vorlesung zur Theoretischen Informatik bekannt ist, ist die Bestimmung hamiltonscher Pfade $\mathcal{NP}$-vollständig.

Hamiltonian Path

Eingabe: Ein gerichteter Graph G.
Gesucht: Besitzt G einen hamiltonschen Pfad.

Wie aus der Einführung in die theoretische Informatik bekannt ist, handelt es sich um ein schwieriges Problem.

Proposition 9.5 *Das Hamiltonian Path Problem ist $\mathcal{NP}$-vollständig.*

Somit haben wir das Problem anscheinen auf ein algorithmisch schwieriges Problem zurückgeführt, was nicht besonders klug erscheint. Offensichtlich besitzt aber jeder Overlap-Graph einen hamiltonschen Pfad, da er ja vollständig ist. Ist das Problem also doch nicht algorithmisch schwierig? Dennoch ist die Bestimmung gewichtsmaximaler Pfade $\mathcal{NP}$-hart. Dazu definieren wir zunächst das Traveling Salesperson Problem.

Traveling Salesperson Problem (TSP)

Eingabe: Ein gewichteter, gerichteter Graph $G = (V, E, \gamma)$ und $K \in \mathbb{N}$.
Gesucht: Besitzt G einen hamiltonschen Kreis C, so dass $\gamma(C) \geq K$.

Auch hierbei handelt es sich um ein schwieriges Problem.

Proposition 9.6 *Das Traveling Salesperson Problem ist $\mathcal{NP}$-vollständig.*

Auch wenn man beim Traveling Salesperson Problem statt eines Kreises einen Pfad fordert, bleibt das Problem $\mathcal{NP}$-vollständig. Dies ist aber genau die Formulierung zum Auffinden eines gewichtsmaximalen hamiltonschen Pfades. Dennoch gibt es Ansätze, die auf diesem Konzept beruhen. Wir werden im zweiten Teil der Vorlesung darauf zurückkommen.

9.3.2 Layout mit eulerschen Pfaden

Nun wollen wir ein alternatives Konzept zum Layout vorschlagen. Um eine der dabei verwendeten Methode kennen zu lernen, gehen wir noch einmal auf die Methode der Sequenzierung durch Hybridisierung zurück. Hierbei werden mithilfe von DNA-Microarrays alle Teilfolgen einer festen Länge ermittelt, die in der zu sequenzierenden Sequenz auftreten. Betrachten wir hierzu ein Beispiel dass in Abbildung 9.8 angege-

T	G	A	C	G	A	C	A	G	A	C	T
T	G	A	C								
	G	A	C	G							
		A	C	G	A						
			C	G	A	C					
				G	A	C	A				
					A	C	A	G			
						C	A	G	A		
							A	G	A	C	
								G	A	C	T

Abbildung 9.8: Beispiel: Teilsequenzen der Länge 4, die bei SBH ermittelt werden

ben ist. In unserem Beispiel erhalten wir also die folgende Menge an (sehr kurzen, wie für SBH charakteristisch) so genannten *Oligos*:

$$\{\text{ACGA}, \text{ACAG}, \text{AGAC}, \text{CAGA}, \text{CGAC}, \text{GACA}, \text{GACG}, \text{GACT}, \text{TGAC}\}.$$

Auch hier müssen wir wieder einen Layout für diese Menge konstruieren. Allerdings würden wir mehrfach vorkommenden Teilsequenzen nicht feststellen. Diese Information erhalten wir über unser Experiment erst einmal nicht, so dass wie eine etwas andere Modellierung finden müssen. Außerdem versuchen wie die Zusatzinformation auszunutzen, dass (bei Nichtberücksichtigung von Fehlern) an jeder Position des DNS-Strangs ein Oligo der betrachteten Länge bekannt ist.

Beim Layout mit hamiltonschen Pfaden haben wir in einem Graphen einen hamiltonschen Kreis gesucht. Dies war ein schwieriges Problem. In der Graphentheorie

gibt es ein sehr ähnliches Problem, nämlich das Auffinden eines eulerschen Kreises, das hingegen algorithmisch sehr leicht ist.

Definition 9.7 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Pfad bzw. ein Kreis $(v_1, \ldots, v_\ell)$ heißt* eulersch*, wenn*

$$\{(v_{i-1}, v_i) \,:\, i \in [2:\ell]\} = E \quad \textit{und} \quad |\{(v_{i-1}, v_i) \,:\, i \in [2:\ell]\}| = |E|$$

bzw.

$$\{(v_{i-1}, v_i), (v_\ell, v_1) \,:\, i \in [2:\ell]\} = E \quad \textit{und} \quad |\{(v_{i-1}, v_i), (v_\ell, v_1) \,:\, i \in [2:\ell]\}| = |E|.$$

Ein Graph heißt eulersch*, wenn er einen eulerschen Pfad besitzt.*

Wir weisen hier darauf hin, dass wir aus gegebenem Anlass den Begriff eulerscher Graph anders definieren als in der Literatur üblich. Dort wird ein Graph als eulersch definiert, wenn er einen eulerschen Kreis besitzt. Da wir hier aber an einem Pfad als Ergebnis und nicht an einem Kreis interessiert sind, wird der Grund für unsere Definition klar. Wir wiederholen noch kurz das Ergebnis, dass es sich sehr effizient feststellen lässt ob ein Graph eulersch ist bzw. einen eulerschen Pfad enthält.

Lemma 9.8 *Sei $G = (V, E)$ ein gerichteter Graph. Der Graph G ist genau dann eulersch, wenn es zwei Knoten $u, w \in V$ gibt, so dass folgendes gilt:*

- *$d^-(v) = d^+(v)$ für alle $v \in V \setminus \{u, w\}$,*
- *$d^-(u) + 1 = d^+(u)$ und*
- *$d^-(w) = d^+(w) + 1$.*

Ein eulerscher Pfad in G kann in Zeit $O(|V| + |E|)$ ermittelt werden, sofern ein solcher existiert.

Der Beweis sei dem Leser überlassen bzw. wir verweisen auf die einschlägige Literatur hierfür. Wir werden jetzt sehen, wie wir diese Eigenschaft ausnutzen können. Dazu definieren wir für eine Menge von Oligos einen so genannten Oligo-Graphen.

Definition 9.9 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von ℓ-Oligos über Σ, d.h. $|s_i| = \ell$ für alle $i \in [1:k]$. Der gerichtete Graph $G_S = (V, E)$ heißt* Oligo-Graph*, wobei*

- *$V = \{s_1 \cdots s_{\ell-1}, s_2 \cdots s_\ell \,:\, s \in S\} \subseteq \Sigma^{\ell-1}$,*
- *$E = \{(v, w) \,:\, \exists s \in S : v = s_1 \cdots s_{\ell-2} \wedge w = s_2 \cdots s_{\ell-1}\}$.*

Als Knotenmenge nehmen wir alle $(\ell-1)$-Tupel aus $\Sigma^{\ell-1}$ her. Damit die Knotenmenge im Zweifelsfall nicht zu groß wird, beschränken wir uns auf alle solchen $(\ell-1)$-Tupel, die ein Präfix oder Suffix eines Oligos sind. Kanten zwischen zwei solcher $(\ell-1)$-Tupel führen wir von einem Präfix zu einem Suffix desselben Oligos. Wir wollen uns diese Definition noch an unserem Beispiel in Abbildung 9.9 veranschaulichen. Wie man dem Beispiel ansieht, kann es durchaus mehrere eulersche Pfade im Oligo-Graphen geben. Einer davon entspricht der ursprünglichen Sequenz.

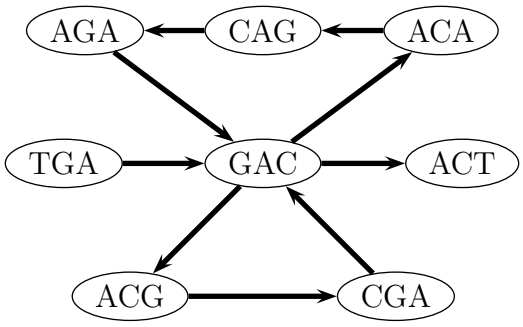


Abbildung 9.9: Beispiel: Oligo-Graph

Probleme hierbei stellen natürlich Sequenzierfehler dar, die den gesuchten eulerschen Pfad zerstören können. Ebenso können lange Repeats (größer gleich ℓ) zu Problemen führen. Wäre im obigen Beispiel das letzte Zeichen der Sequenz ein A, so gäbe es ein Repeat der Länge 4, nämlich GACA. Im Oligo-Graphen würde das dazu führen dass die Knoten ACT und ACA verschmelzen würden. Der Graph hätte dann ebenfalls keinen eulerschen Pfad mehr (außer wir würden Mehrfachkanten erlauben, hier eine Doppelkante zwischen GAC nach ACA).

Könnte uns die Technik der eulerschen Pfade beim Fragment Assembly helfen? Ja, die Idee ist die Folgende. Wir kennen ja Sequenzen der Länge 500. Diese teilen wir in überlappende Oligos der Länge ℓ (in der Praxis wählt man $\ell \approx 20$) wie folgt ein. Sei $s = s_1 \cdots s_n$ ein Fragment, dann erhalten wir daraus $n - \ell + 1$ ℓ-Oligos durch $s^{(i,\ell)} = s_i \cdots s_{i+\ell-1}$ für $i \in [1 : n - \ell + 1]$.

Diese Idee geht auf Idury und Waterman zurück und funktioniert, wenn es keine Sequenzierfehler und nur kurze Repeats gibt. Natürlich müssen wir auch hier voraussetzen, dass die zu sequenzierende Sequenz gut überdeckt ist, dass heißt jedes Nukleotid wird durch mindestens ℓ verschiedene Oligos überdeckt.

Dieser Ansatz hat allerdings auch den Vorteil, dass man versuchen kann die Fehler zu reduzieren. Ein Sequenzierfehler erzeugt genau ℓ fehlerhafte Oligos (außer der Fehler taucht am Rand des Fragments auf, dann natürlich entsprechend weniger). Hierbei nutzt man aus, dass eine Position ja von vielen Fragmenten und somit auch Oligos an derselben Position überdeckt wird (in der Praxis etwa 10) und dass pro

Oligo aufgrund deren Kürze (in der Praxis etwa 20) nur wenige Sequenzierfehler (möglichst einer) vorliegen.

Dazu ein paar Definitionen. Ein Oligo heißt *solide*, wenn es in einer bestimmen Mindestanzahl der vorliegenden Fragmente vorkommt (beispielsweise mindestens in der Hälfte der Fragmente, die eine Position überdecken sollen). Zwei Oligos heißen *benachbart*, wenn sie durch eine Substitution ineinander überführt werden können. Ein Oligo heißt *Waise*, wenn es nicht solide ist, und es zu genau einem anderen soliden Oligo benachbart ist.

Beim Korrekturvorgang suchen wir nach Waisen und ersetzen diese in den Fragmenten durch ihren soliden Nachbarn. Mithilfe dieser Prozedur kann die Anzahl der Fehler deutlich reduziert werden. Hierbei ist anzumerken, dass Fehler hier nicht bezüglich der korrekten Sequenz gemeint ist, sondern so zu verstehen ist, dass Fehler reduziert werden, die im zugehörigen Oligo-Graphen eulersche Pfade eliminieren.

Wie wir schon vorher kurz angemerkt haben, können Repeats ebenfalls eulersche Pfade eliminieren. Um dies möglichst gering zu halten, erlauben wir in unserem Graphen mehrfache Kanten. Außerdem haben wir in unserem Oligo-Graphen ja noch eine wichtige Zusatzinformation. Die Oligos sind ja nicht durch Hybridisierungsexperimente entstanden, sondern wir haben sie aus den Sequenzinformationen der Fragmente abgelesen.

Ein Fragment der Länge n induziert daher nicht nur $n - \ell + 1$ Oligos, sondern wir kennen ja auch die Reihenfolge dieser Oligos. Das heißt nichts anderes, als dass jedes Fragment einen Pfad der Länge $n - \ell$ auf den $n - \ell + 1$ Oligos induziert (siehe auch Abbildung 9.10 für die Fragmente TGACGA, ACGACAG, ACAGACT). Somit suchen wir jetzt nach einem eulerschen Pfad im Oligo-Graphen, der diese Pfade nach Möglichkeit respektiert. Dies macht die Aufgabe in der Hinsicht leichter, dass bei mehreren möglichen eulerschen Pfaden leichter ersichtlich ist, welche Variante zu wählen ist. Bei langen Repeats bleibt dieses prinzipielle Problem jedoch weiterhin bestehen.

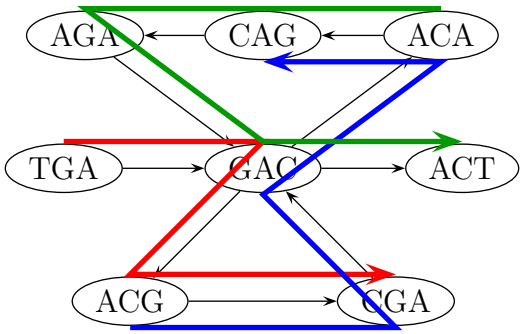


Abbildung 9.10: Beispiel: Aus Fragmenten generierter Oligo-Graph

9.3.3 Layout mit Spannbäumen

Das Fragment-Layout kann man aus einem Overlap-Graphen auch mit Hilfe eines Greedy-Algorithmus aufbauen. Hierbei werden in das Fragment Layout die Overlaps in der Reihenfolge nach ihrem Score eingearbeitet, beginnend mit dem Overlap mit dem größten Score.

Sind beide Sequenzen noch nicht im Layout enthalten, so werden sie mit dem aktuell betrachteten Overlap in dieses neu aufgenommen. Ist eine der beiden Sequenzen bereits im Layout enthalten, so wird die andere Sequenz mit dem aktuell betrachteten Overlap in das Layout aufgenommen. Hierbei muss beachtet werden, wie sich die neue Sequenz in das bereits konstruierte aufnehmen lässt. Kommt es hier zu großen Widersprüchen, so wird die Sequenz mit dem betrachteten Overlap nicht aufgenommen. Sind bereits beide Sequenzen im Overlap enthalten und befinden sich in verschiedenen Zusammenhangskomponenten des Layouts, so wird versucht die beiden Komponenten mit dem aktuell betrachteten Overlap zusammenzufügen. Auch hier wird der betrachtete Overlap verworfen, wenn sich mit diesem Overlap die beiden bereits konstruierten Layout nur mit großen Problemen zusammenfügen lassen.

Bei dieser Vorgehensweise gibt es wiederum insbesondere Probleme bei den Repeats. Repeats sind ja Teilsequenzen, die mehrfach auftreten. Bei Prokaryonten ist dies eher selten, bei Eukaryonten treten jedoch sehr viele Repeats auf. In der Regel werden dann solche Repeats, die ja mehrfach in der Sequenz auftauchen, auf eine Stelle im Konsensus abgebildet. Ist die vorhergesagte Sequenz deutlich zu kurz, so deutet dies auf eine fehlerhafte Einordnung von Repeats hin.

Versucht man nun beim Aufbau nicht darauf zu achten, ob ein Pfad entsteht, sondern nur, dass Kreise vermieden werden, erhält man einen so genannten Spannbaum oder Spannwald des Overlap-Graphen. Hierzu benötigen wir zuerst noch einige grundlegende Definitionen aus der Graphentheorie.

Definition 9.10 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Teilgraph $G' \subset G$ heißt* aufspannend*, wenn $V(G') = V(G)$ und G' zusammenhängend ist.*

Der aufspannende Teilgraph enthält also alle Knoten des aufgespannten Graphen. Nun können wir den Begriff eines Spannbaumes bzw. Spannwaldes definieren.

Definition 9.11 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Teilgraph $G' \subset G$ heißt* Spannbaum*, wenn G' ein aufspannender Teilgraph von G ist und G' ein Baum ist.*

Definition 9.12 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Teilgraph $G' \subset G$ heißt* Spannwald*, wenn $V(G') = V(G)$ ist und G' ein Wald ist.*

Für gewichtete Graphen brauchen wir jetzt noch das Konzept eines minimalen bzw. maximalen Spannbaumes.

Definition 9.13 *Sei $G = (V, E, \gamma)$ ein gewichteter gerichteter Graph und T ein Spannbaum von G. Das* Gewicht des Spannbaumes *T von G ist definiert durch*

$$\gamma(T) := \sum_{e \in E(T)} \gamma(e).$$

Ein Spannbaum T für G heißt minimal *bzw.* maximal*, wenn er unter allen möglichen Spannbäumen für G minimales bzw. maximales Gewicht besitzt, d.h.*

$$\gamma(T) \leq \min \{\gamma(T') \ : \ T' \text{ ist ein Spannbaum von } G\}$$

bzw.

$$\gamma(T) \leq \max \{\gamma(T') \ : \ T' \text{ ist ein Spannbaum von } G\}.$$

In der Literatur sind minimale Spannbäume gebräuchlicher, wir benötigen hier jedoch das Konzept eines maximalen Spannbaumes, was jedoch algorithmisch äquivalent ist.

Proposition 9.14 *Sei $G = (V, E, \gamma)$ ein gewichteter Graph mit $\gamma(e) < B$ für alle $e \in E$. Sei $G =' (V, E, \gamma')$ ein gewichteter Graph mit $\gamma(e) := B - \gamma(e)$ für alle $e \in E$. Dann ist ein Spannbaum T in G genau dann ein minimaler Spannbaum, wenn T ein maximaler Spannbaum in G' ist.*

In der Abbildung 9.11 ist noch einmal ein Layout mit dem zugehörigen Overlap-Graphen gezeigt. Dort sind nur die Kanten eingezeichnet, deren Gewicht einen gewissen Threshold übersteigen.

Konstruiert man nun aus dem Overlap-Graphen aus Abbildung 9.11 mit Hilfe der Greedy-Methode eine Pfad, so werden zuerst die Kanten (s_3, s_2) und (s_1, s_3)in den Pfad aufgenommen. Die Kante (s_5, s_3) kann dann nicht mehr aufgenommen werden, da sonst der Knoten (s_3) einen Eingangsgrad von mindestens 2 besitzen würde, was der Definition eines Pfades widerspricht. Der resultierende Pfad ist im linken Teil der Abbildung 9.12 dargestellt. Man beachte hierbei, dass der Knoten s_5 nicht in den Pfad integriert werden kann. Dies ist die Schwäche des Greedy-Ansatzes, er kann den Pfad nicht vollständig rekonstruieren bzw. sogar ganz falsche Layouts konstruieren.

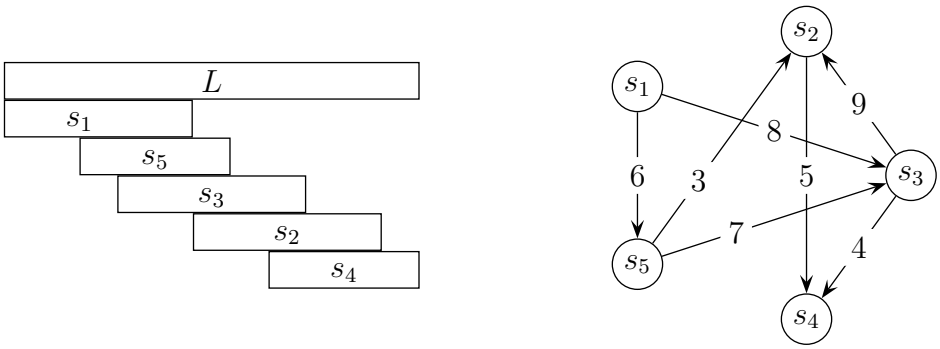


Abbildung 9.11: Skizze: Overlap-Graph G mit den schwersten Kanten

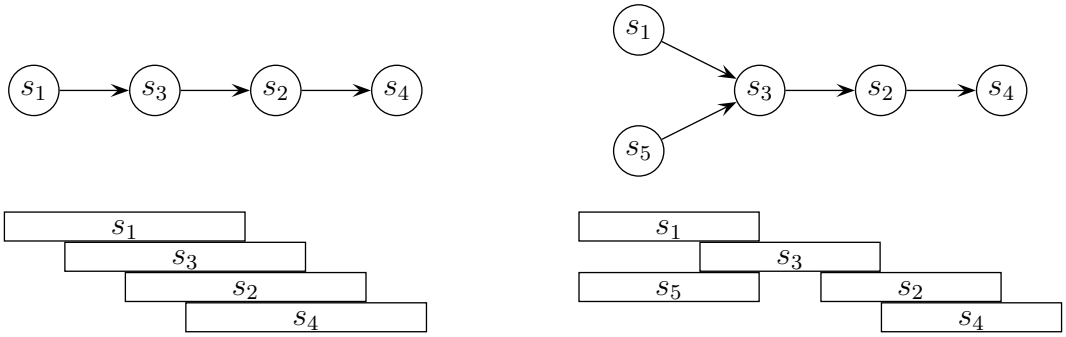


Abbildung 9.12: Skizze: Greedy-Path und minimaler Spannbaum für G aus Abbildung 9.11

Konstruieren wir nun ebenfalls mit der Greedy-Methode einen Spannbaum des Overlap-Graphen, so erhalten wir den im rechten Teil von Abbildung 9.12 angegebenen Baum. Hieraus lässt sich leicht ein Layout rekonstruieren, wobei nur die relative Lage von s_1 und s_5 ungeklärt bleibt.

Die Greedy-Methode zur Konstruktion von minimalen Spannbäumen ist auch als Algorithmus von Kruskal bekannt. Man beachte, dass in unserem Fall nicht unbedingt ein Spannbaum entstehen muss, sondern nur ein Spannwald, da wir nur Kanten mit einem gewissen Mindestgewicht berücksichtigen.

Literaturhinweise A

A.1 Lehrbücher zur Vorlesung

S. Aluru (Ed.): *Handbook of Computational Molecular Biology*; Chapman and Hall/CRC, 2006.

H.-J. Böckenhauer, D. Bongartz: *Algorithmische Grundlagen der Bioinformatik: Modelle, Methoden und Komplexität*, Teubner, 2003.

Ph. Compeau, P. Pevzner: *Bioinformatics Algorithms — An Active Learning Approach*, Active Learning Publishers, 3rd Ed., 2018.

P. Clote, R. Backofen: *Introduction to Computational Biology*, John Wiley and Sons, 2000.

R.C. Deonier, S. Tavare, M.S. Waterman: *Computational Genome Analysis*, Springer, 2005.

R. Durbin, S. Eddy, A. Krogh, G. Mitchison: *Biological Sequence Analysis*, Cambridge University Press, 1998.

D. Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*, Cambridge University Press, 1997.

N.C. Jones, P.A. Pevzner: *An Introduction to Bioinformatics Algorithms*, MIT Press, 2004.

V. Mäkinen, F. Cunial, D. Belazzougui, A.I. Tomescu: *Genome-Scale Algorithm Design*, Cambridge University Press, 2015.

J.C. Setubal, J. Meidanis: *Introduction to Computational Molecular Biology*, PWS Publishing Company, 1997.

Wing-Kin Sung: *Algorithms in Bioinformatics — A Practical Introduction*, CRC Press, 2009.

M.S. Waterman: *Introduction to Computational Biology: Maps, Sequences, and Genomes*, Chapman and Hall, 1995.

A.2 Lehrbücher zur Bioinformatik

I. Eidhammer, I. Jonassen, W.R. Taylor: *Protein Bioinformatics — An Algorithmic Approach to Sequence and Structure Analysis*, Jon Wiley and Sons, 2004.

W.J. Ewens, G.R. Grant: *Statistical Methods in Bioinformatics*, Springer, 2001.

A. Isaev: *Introduction to Mathematical Methods in Bioinformatics*, Springer, 2004.

T. Koski: *Hidden Markov Models for Bioinformatics*, Kluwer Academics Publishers, 2001.

J.S. Liu: *Bayesian Modeling and Computation in Bioinformatics Research*, in: Current Topics in Computational Molecular Biology, T. Jiang, Y. Xu, M.Q. Zhang (Eds.), MIT Press, 2002.

D.W. Mount: *Bioinformatics — Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.

E. Ohlebusch: *Bioinformatics Algorithms — Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstrution*, Oldenbusch Verlag, 2013.

P.A. Pevzner: *Computational Molecular Biology — An Algorithmic Approach*, MIT Press, 2000.

A.3 Lehrbücher zur Algorithmik und Komplexität

G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Potasi: *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability*, Springer, 1999.

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*, MIT Press, 2nd edition, 2001.

M. Crochemore, W. Rytter: *Jewels of Stringology — Text Algorithms*, World Scientific Press, 2003.

V. Heun: *Grundlegende Algorithmen*, 2. Auflage, Vieweg, 2003.

J. Kleinberg, E. Tardos: *Algorithm Design*, Addison Wesley, 2005.

E.W. Mayr, A. Steger, H.J. Prömel: *Lectures on Proof Verification and Approximation Algorithms*, Lecture Notes in Computer Science, Vol. 1367, Springer, 1998.

I. Wegener: *Komplexitätstheorie — Grenzen der Effizienz von Algorithmen*, Springer, 2003.

A.4 Lehrbücher zur Algorithmenanalyse

R.L. Graham, D.E. Knuth O. Patashnik: *Concrete Mathematics: A Foundation of Computer Science*, Addison-Wesley, 1989.

D.H. Greene, D.E. Knuth: *Mathematics for the analysis of Algorithms*, Birkhäuser, 1990.

M. Hofri: *Analysis of Algorithms — Computational methods and Mathematical Tools*, Oxford Press, 1995.

R. Sedgewick, Ph. Flajolet: *An Introduction to the Analysis of Algorithms*, Addison-Wesley Publushing company, 1996.

A. Steger: *Diskrete Strukturen I*, Springer, 2001.

Th. Schickinger, A. Steger: *Diskrete Strukturen II*, Springer, 2001.

R. Sedgewick, Ph. Flajolet: *An Introduction to the Analysis of Algorithms*, Addison Wesley, 1996.

H.S. Wilf: *generatingfunctionology*, A K Peters, Ltd., 2005.
http://www.math.upenn.edu/~wilf/DownldGF.html

Index B

Symbole

A

D

E

F

G

N

Q

R

S

T

U

V

W

Y

Z