INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
LEHRSTUHL FÜR EFFIZIENTE ALGORITHMEN

Skriptum

zur Vorlesung

Algorithmische Bioinformatik I/II

gehalten im Wintersemester 2001/2002

und im Sommersemester 2002 von

Volker Heun

Erstellt unter Mithilfe von:

Peter Lücke – Hamed Behrouzi – Michael Engelhardt

Sabine Spreer – Hanjo Täubig

Jens Ernst – Moritz Maaß

14. Mai 2003

Version 0.96

Vorwort

Dieses Skript entstand parallel zu den Vorlesungen *Algorithmische Bioinformatik I* und *Algorithmische Bioinformatik II*, die im Wintersemester 2001/2002 sowie im Sommersemester 2002 für Studenten der Bioinformatik und Informatik sowie anderer Fachrichtungen an der Technischen Universität München im Rahmen des von der Ludwig-Maximilians-Universität und der Technischen Universität gemeinsam veranstalteten Studiengangs Bioinformatik gehalten wurde. Einige Teile des Skripts basieren auf der bereits im Sommersemester 2000 an der Technischen Universität München gehaltenen Vorlesung *Algorithmen der Bioinformatik* für Studierende der Informatik.

Das Skript selbst umfasst im Wesentlichen die grundlegenden Themen, die man im Bereich Algorithmische Bioinformatik einmal gehört haben sollte. Die vorliegende Version bedarf allerdings noch einer Ergänzung weiterer wichtiger Themen, die leider nicht in den Vorlesungen behandelt werden konnten.

An dieser Stelle möchte ich insbesondere Hamed Behrouzi, Michael Engelhardt und Peter Lücke danken, die an der Erstellung des ersten Teils dieses Skriptes (Kapitel 2 mit 5) maßgeblich beteiligt waren. Bei Sabine Spreer möchte ich mich für die Unterstützung bei Teilen des siebten Kapitels bedanken. Bei meinen Übungsleitern Jens Ernst und Moritz Maaß für deren Unterstützung der Durchführung des Übungsbetriebs, aus der einige Lösungen von Übungsaufgaben in dieses Text eingeflossen sind. Bei Hanjo Täubig möchte ich mich für die Mithilfe zur Fehlerfindung bedanken, insbesondere bei den biologischen Grundlagen.

Falls sich dennoch weitere (Tipp)Fehler unserer Aufmerksamkeit entzogen haben sollten, so bin ich für jeden Hinweis darauf (an heun@in.tum.de) dankbar.

München, im September 2002 Volker Heun

Inhaltsverzeichnis

Molekularbiologische Grundlagen

1.1 Mendelsche Genetik

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.

1.1.1 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.

1.1.2 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.

Elterngeneration

Erste Tochtergeneration

Zweite Tochtergeneration

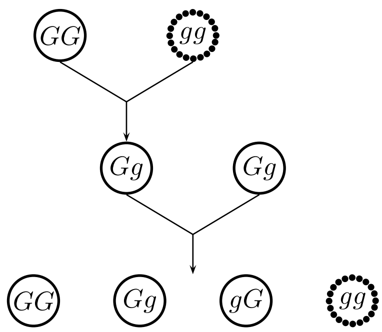


Abbildung 1.1: Skizze: Mendelsche Vererbung

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 1.2).

	G	g
G	GG	Gg
g	gG	gg

Abbildung 1.2: 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).

1.1.3 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.

1.1.4 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).

1.2 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.

1.2.1 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*.

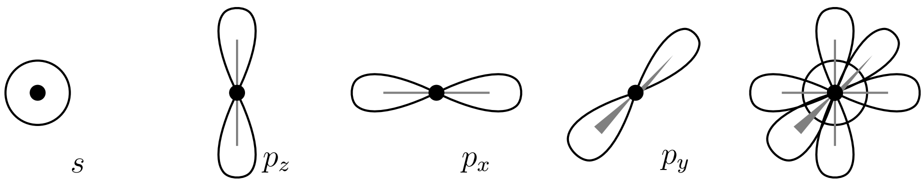


Abbildung 1.3: Skizze: Räumliche Ausdehnung der Orbitale

Das s-Orbital ist dabei kugelförmig, während die drei verschiedenen p-Orbitale jeweils eine *Doppelhantel* ausbilden, die paarweise orthogonal zueinander sind. In Abbildung 1.3 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 1.4 links dargestellt. Die Endpunkte der vier Keulen bilden dabei ein Tetraeder aus. Bei einer Einfachbindung überlappen sich zwei der Keulen, wie in Abbildung 1.4 rechts dargestellt. Die in der Einfachbindung überlappenden q-Orbitale bilden dann ein so genanntes *σ-Orbital*.

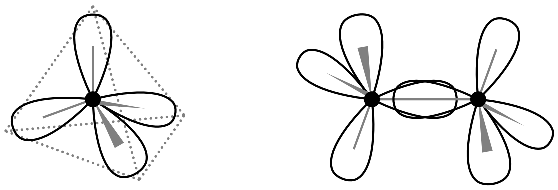


Abbildung 1.4: Skizze: sp^3 hybridisierte Orbitale sowie eine Einfachbindung

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 1.5 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 1.5 rechts dargestellt (die Abbildungen sind nicht maßstabsgetreu).

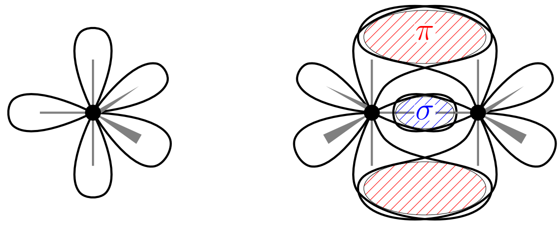


Abbildung 1.5: Skizze: sp^2 hybridisierte Orbitale und eine Doppelbindung

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*.

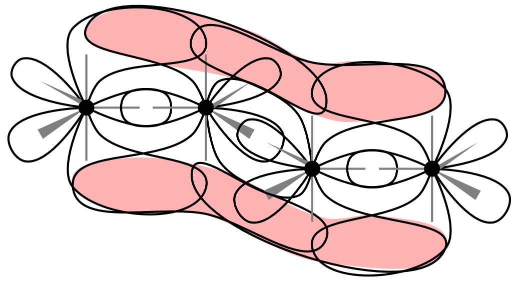


Abbildung 1.6: Skizze: Delokalisierte π-Bindung im Butadien

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.

1.2.2 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.

1.2.3 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 1.7 angegeben. Hier bedeutet ein größerer Wert eine größere Affinität zu Elektronen.

Atom	C	O	H	N	S	P
EN	2.5	3.5	2.1	3.0	2.5	2.1

Abbildung 1.7: 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.

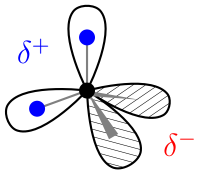


Abbildung 1.8: Skizze: Polarität bei einem Wassermolekül

Insbesondere Wasser ist für die Ausbildung von zahlreichen Wasserstoffbrücken bekannt. In Abbildung 1.8 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.

1.2.4 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.

1.2.5 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.

1.2.6 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.

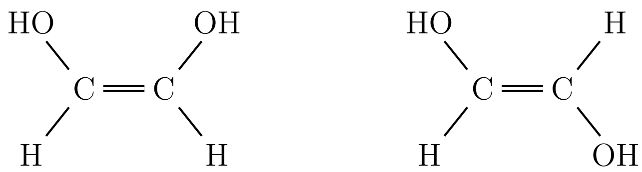
Chem. Rest	Gruppe	gewöhnlicher Name
$-CH_3$	Methyl	
–OH	Hydroxyl	Alkohol
$-NH_2$	Amino	Amine
–NH–	Imino	
–CHO	Carbonyl	Aldehyde
–CO–	Carbonyl	Ketone
–COO–	Ester	Ester
–COOH	Carboxyl	organische Säure
–CN	Cyanid	Nitrile
–SH	Sulfhydril	Thiole

Abbildung 1.9: 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 1.9 zusammengefasst.

1.2.7 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 aufrund der π-Bindung eingeschränkt und es kann zwei mögliche räumliche Anordnungen desselben Moleküls geben. In Abbildung 1.10 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



cis trans

Abbildung 1.10: Skizze: Cis-Trans-Isomerie bei Äthendiol

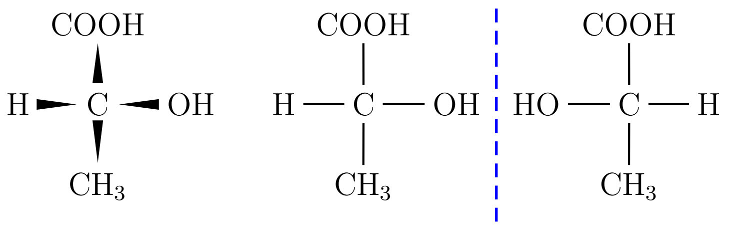
geöffnet werden und um 180° gedreht werden, so dass die beiden Isomere ineinander überführt werden können.

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 1.11 ein Kohlenstoffatom (schwarz darge-

Abbildung 1.11: Skizze: Asymmetrisches Kohlenstoffatom

stellt) und vier unterschiedliche funktionelle Gruppen (farbig dargestellt), die jeweils 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 1.11 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*.



Milchsäure D-(−)-Milchsäure L-(+)-Milchsäure

Abbildung 1.12: 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 1.12 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 1.12 ganz links noch einmal illustriert.

Da nun im mittleren Teil der Abbildung 1.12 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.

1.2.8 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 1.13

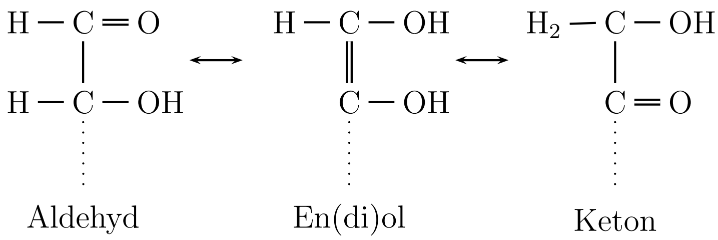


Abbildung 1.13: Skizze: Keto-Enol-Tautomerie

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.

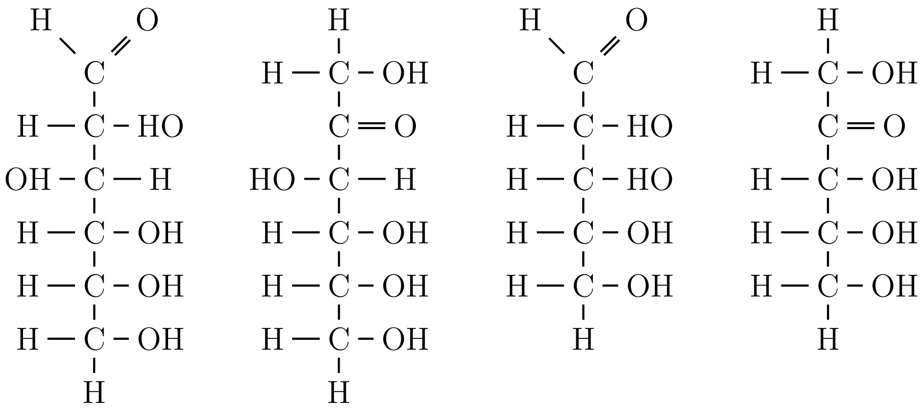
1.3 DNS und RNS

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.

1.3.1 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 1.14 sind die für uns wichtigsten Zucker in der Strukturformel dargestellt. Für uns sind insbesondere Zucker mit 5 oder 6 Kohlenstoffatomen von Interessen. 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



Hexose	Hexose	Pentose	Pentose
Aldose	Ketose	Aldose	Ketose
D-Glucose	D-Fructose	D-Ribose	D-Ribulose

Abbildung 1.14: 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 1.15 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.

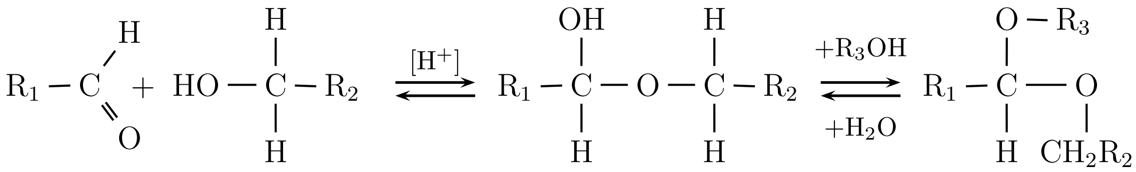


Abbildung 1.15: Skizze: Halb-Acetal- und Voll-Acetal-Bildung

In der Abbildung 1.16 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 1.16).

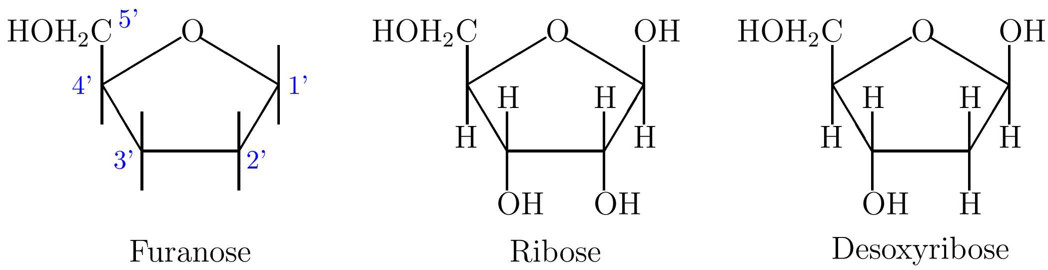


Abbildung 1.16: Skizze: Ribose und Desoxyribose als Furanosen

1.3.2 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 1.17 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.

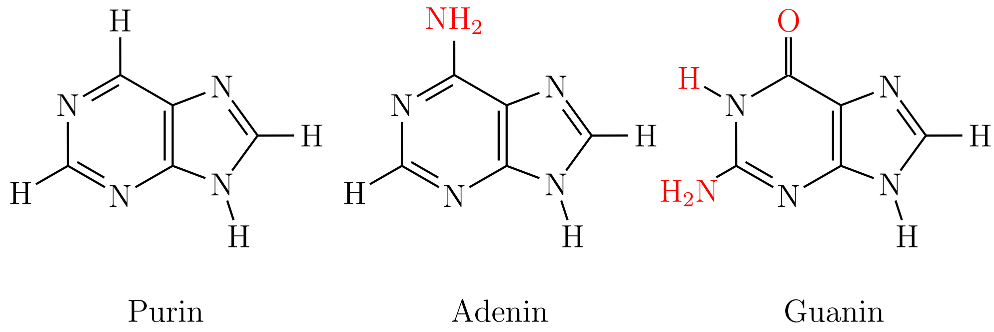


Abbildung 1.17: Skizze: Purine

Eine weitere Gruppe von Basen erhält man aus Pyrimidin, dessen Strukturformel in der Abbildung 1.18 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-

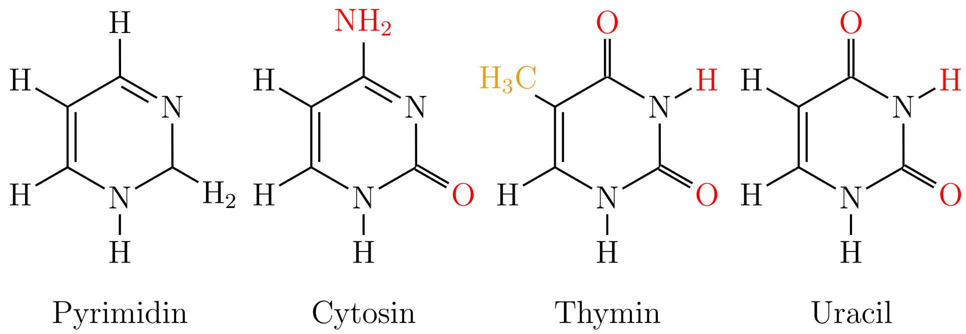


Abbildung 1.18: Skizze: Pyrimidine

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.

1.3.3 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, gebildet. 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 1.19 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.

1.3.4 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 1.20 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

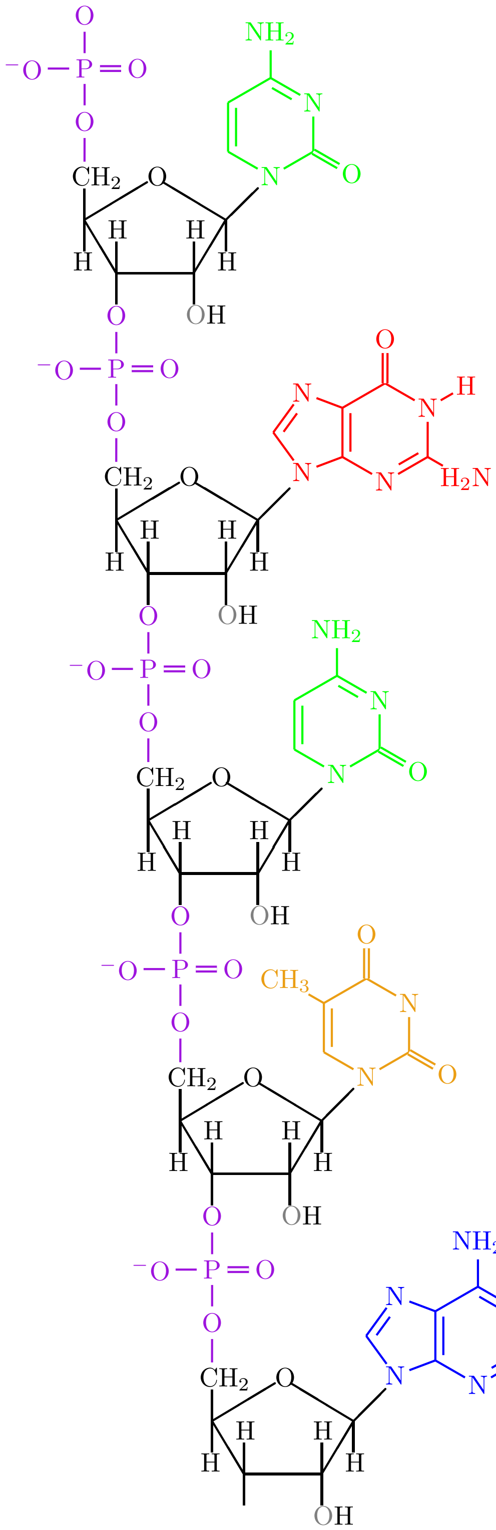


Abbildung 1.19: Skizze: DNS bzw. RNS als Polymerstrang

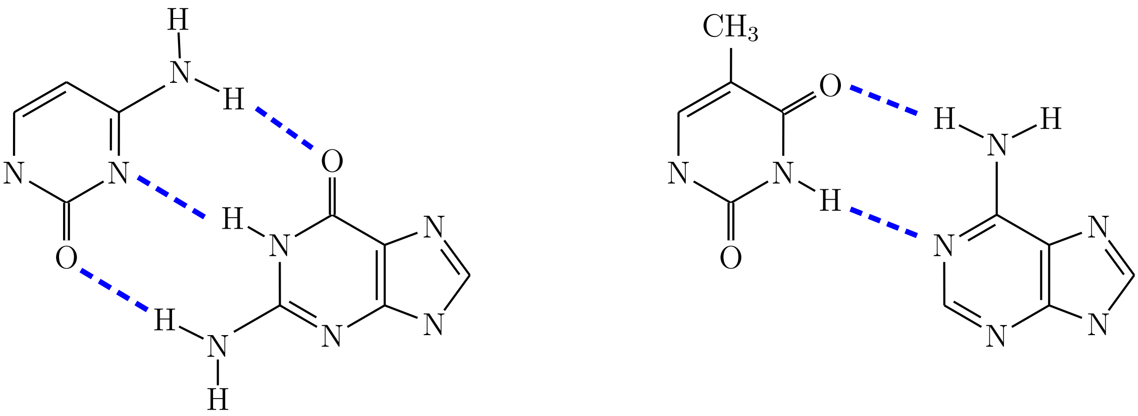


Abbildung 1.20: Skizze: Wasserstoffbrücken der komplementären Basen

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).

1.3.5 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 nebeneinader (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 1.21 gegeben.

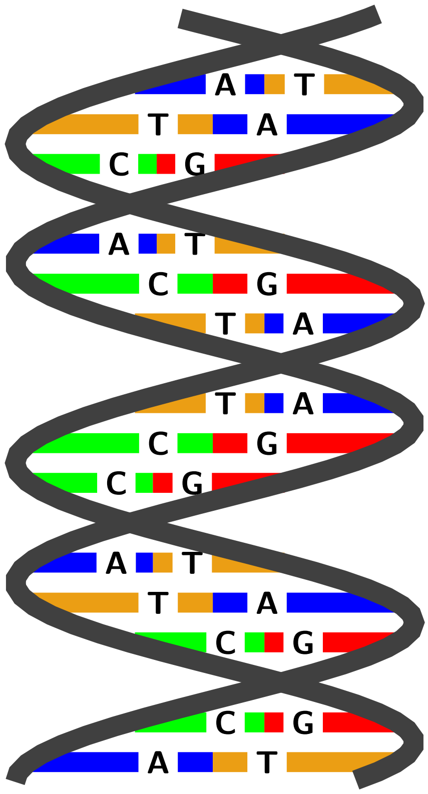


Abbildung 1.21: Skizze: Doppelhelix der DNS

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 1.21), 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.

1.4 Proteine

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

1.4.1 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

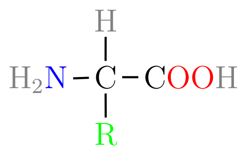


Abbildung 1.22: Aminosäure

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 1.22 ist die Strukturformel einer generischen Aminosäure dargestellt.

In Abbildung 1.23 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 1.24 sind die grundlegendsten Eigenschaften der einzelnen Aminosäuren schematisch zusammengefasst.

Auch hier sind in der Regel (mit Ausnahmen von Glyzin) am zentralen Kohlenstoffatom vier verschiedene Substituenten vorhanden. Somit handelt es sich bei dem

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
Aspartamsäure oder Asparagin	Asx	B
Glutaminsäure oder Glutamin	Glx	Z
Beliebige Aminosäure	Xaa	X

Abbildung 1.23: Tabelle: Liste der zwanzig Aminosäuren

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.

1.4.2 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

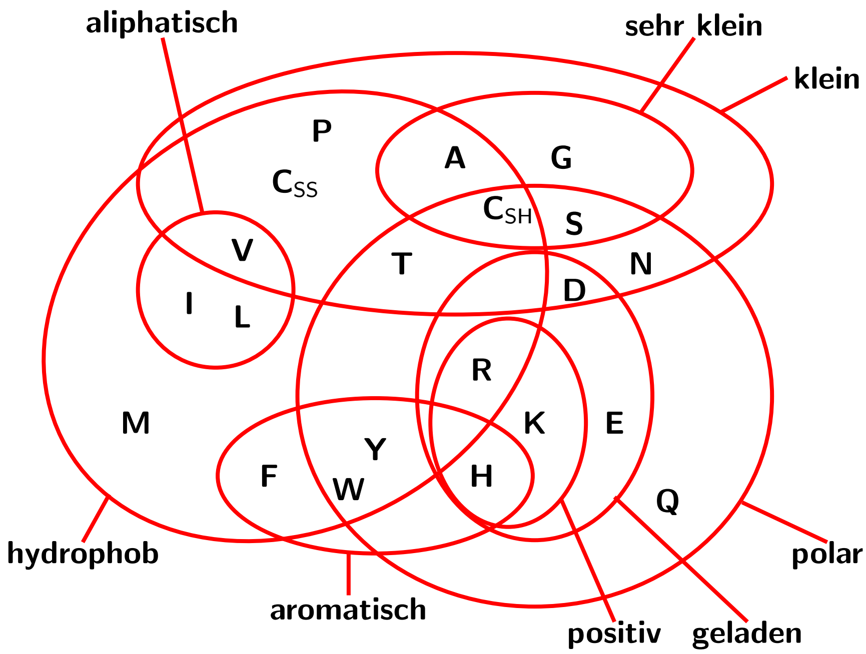


Abbildung 1.24: Skizze: Elementare Eigenschaften von Aminosäuren

neuen Molekül, einem so genannten *Dipeptid*. Die chemische Reaktionsgleichung ist in Abbildung 1.25 illustriert.

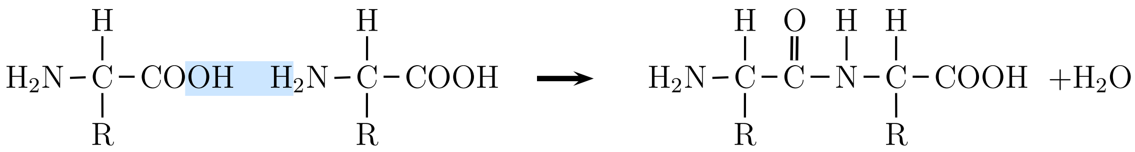


Abbildung 1.25: Skizze: Säureamidartige oder Peptidbindung

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 verwundenen Polypeptiden.

Wir wollen uns nun eine solche Peptidbindung etwas genauer anschauen. Betrachten wir hierzu die Abbildung 1.26. Von unten links nach oben rechts durchlaufen wir eine Peptidbindung vom zentralen Kohlenstoffatom der ersten Aminosäure über

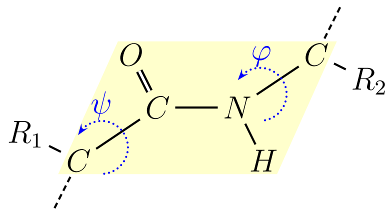


Abbildung 1.26: Skizze: freie Winkel in der Peptidbindung

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 1.27 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.

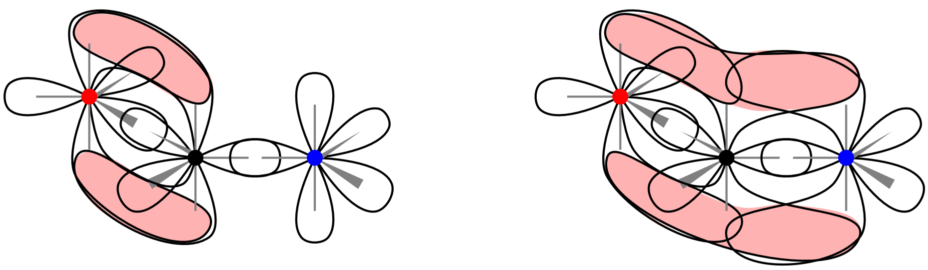


Abbildung 1.27: Skizze: Elektronenwolken in der Peptidbindung

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 1.28 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.

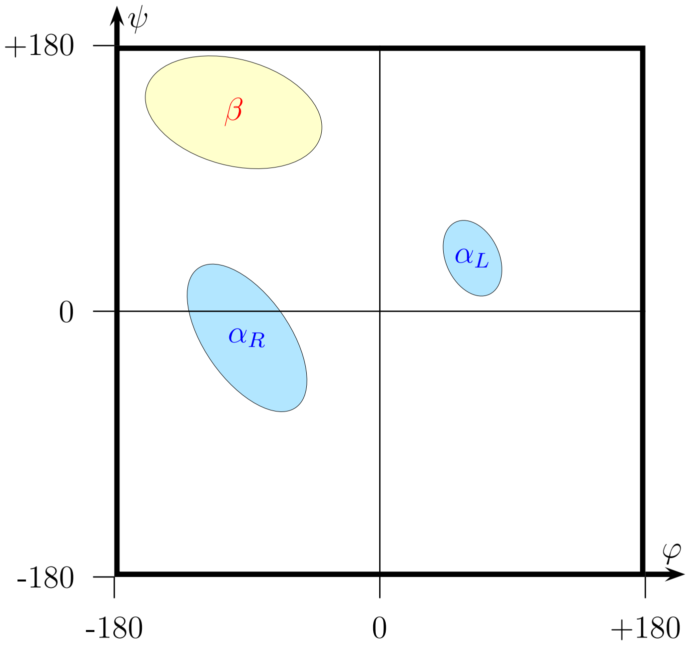


Abbildung 1.28: Skizze: Ramachandran-Plot (schematische Darstellung)

1.4.3 Proteinstrukturen

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

1.4.3.1 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.

1.4.3.2 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 Linksdrehung, weil bei einer Rechtsdrehung die sterische Hinderung deutlich größer ist. Die zugehörigen Winkelpaare der Peptidbindung entsprechen im Ramachandran-Plot in Abbildung 1.28 dem mit α_L markierten Bereich. Einige wenige Helices bilden eine Rechtsdrehung aus. Die zugehörigen Winkelpaare sind im Ramachandran-Plot mit α_R 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 1.28 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.

1.4.3.3 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

1.4.3.4 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).

1.4.3.5 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.

1.5 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.

1.5.1 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 1.29 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 Strand 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

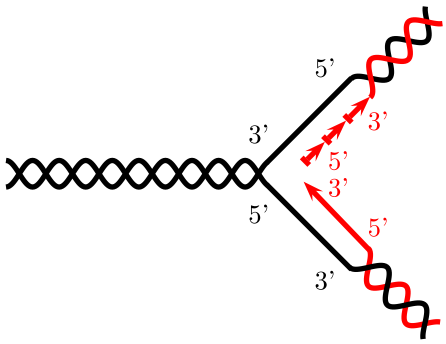


Abbildung 1.29: Skizze: DNS-Replikation

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 die DNS hier immer in kleinen Stücken synthetisiert wird und nicht im ganzen wie am komplementären Strang. Die dabei generierten kleine DNS-Teilstränge werden nach ihrem Entdecker *Okazaki-Fragmente* genannt.

1.5.2 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 wird 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 1.30.

1.5.3 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.

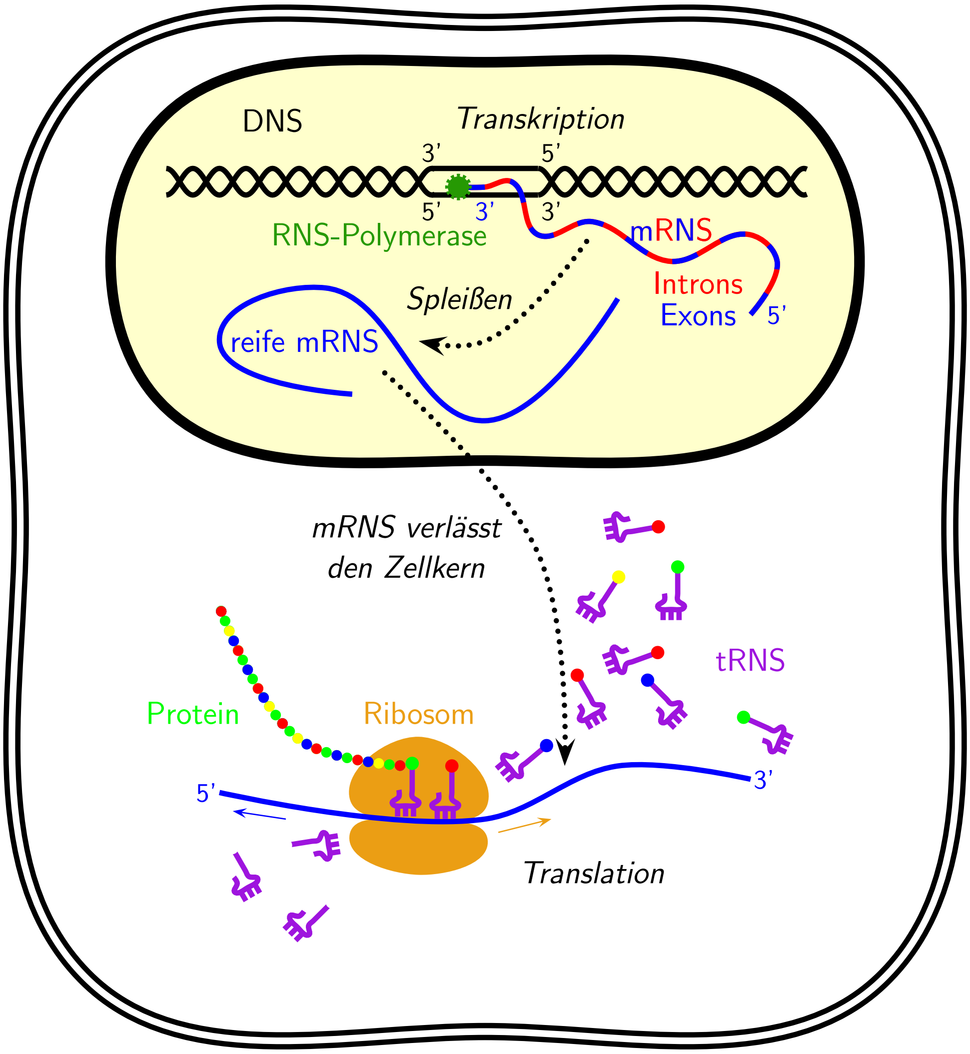


Abbildung 1.30: Skizze: Transkription und Transaltion in einer Zelle

Da es 64 verschiedene Tripletts aber nur zwanzig Aminosäuren gibt, werden einige Aminosäuren durch mehrere Tripletts codiert. In der folgenden Abbildung 1.31 ist der Code für die Umwandlung von Tripletts in Aminosäuren angegeben. In Abbildung 1.31 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

	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 1.31: 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, Glyzin, 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, 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 mit drei Basen und einer Bindungsstelle für eine Aminosäure. Dabei entsprechen die drei komplementären Basen der zugehörigen Aminosäure, 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 aufzuladen.

1.5.4 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.

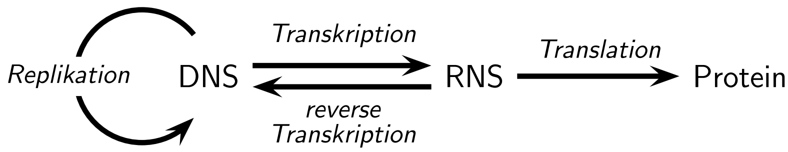


Abbildung 1.32: Skizze: Das zentrale Dogma der Molekularbiologie

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 1.32 noch einmal schematisch dargestellt.

1.5.5 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 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 1.33 schematisch dargestellt ist. In höheren Lebewesen und insbesondere in eukaryontischen Zellen sind solche Promotoren weitaus komplexer und es gibt eine ganze Reihe hiervon.

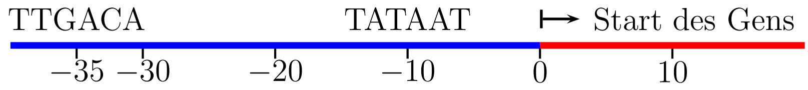


Abbildung 1.33: Skizze: Promotoren in Bakterien

1.6 Biotechnologie

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

1.6.1 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, so 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.

1.6.2 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 eingesetzt 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.

1.6.3 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 sich 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 1.34 schematisch dargestellt. Dabei ist das zu vervielfältigende DNS-Stück grün, die Primer rot und Rest der DNS grau dargestellt. Die Pfeile geben die Synthetisierungssrichtung der Polymerase an.

Abbildung 1.34: Skizze: Polymerasekettenreaktion

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

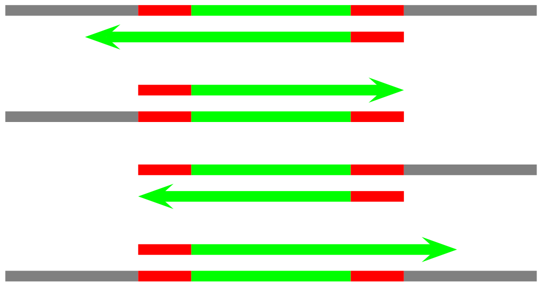


Abbildung 1.35: Skizze: PCR: nach der 2. Verdopplung

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.

1.6.4 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 Abbil-

5'	G	A	A	T	T	C	3'
3'	C	T	T	A	A	G	5'

Abbildung 1.36: Skizze: Restriktionsenzym mit Muster *GAATTC*

dung 1.36 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*.

1.6.5 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 1.37 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 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

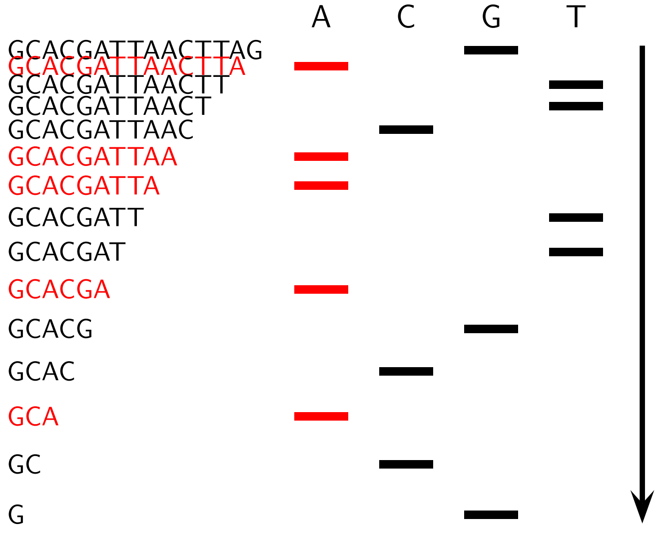


Abbildung 1.37: Skizze: Sequenzierung nach Sanger

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.

1.6.5.1 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.

1.6.5.2 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 1.38). Diese Methode wird nach ihren Erfinder auch *Maxam-Gilbert-Methode* genannt.

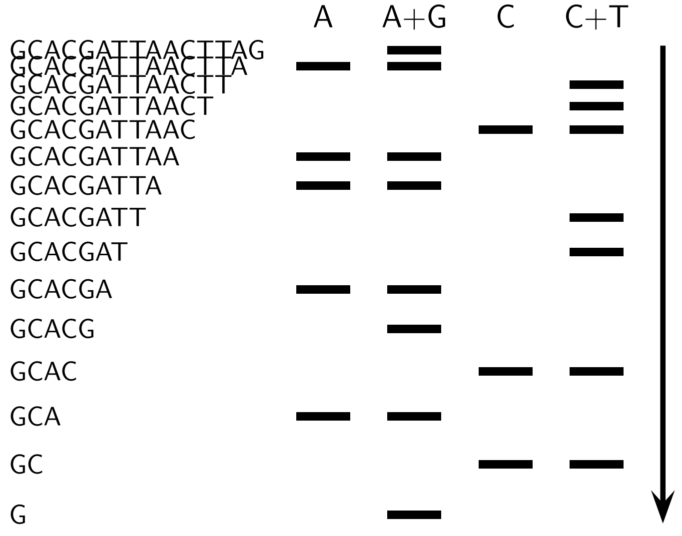


Abbildung 1.38: Skizze: Sequenzierung nach Maxam-Gilbert

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.

1.6.6 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 lässt. In diesem letzten Abschnitt wollen wir uns überlegen, wie man ein ganzes Genom sequenzieren kann.

1.6.6.1 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.

1.6.6.2 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.

1.6.6.3 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.

1.6.6.4 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.

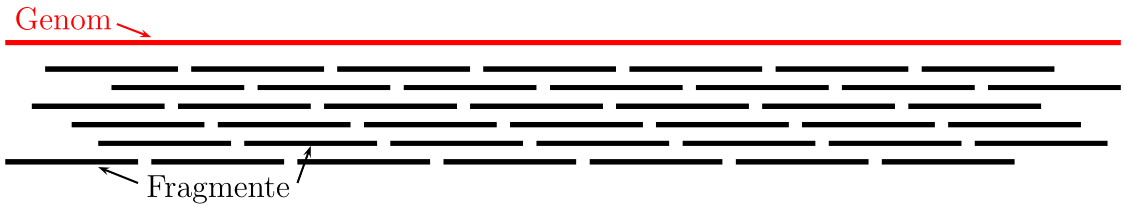


Abbildung 1.39: Skizze: Shotgun-Sequencing

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.

Suchen in Texten

2.1 Grundlagen

- Ein *Alphabet* ist eine endliche Menge von Symbolen.
  Bsp.: $\Sigma = \{a, b, c, \ldots, z\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{A, C, G, T\}$.
- *Wörter* über Σ sind endliche Folgen von Symbolen aus Σ. Wörter werden manchmal 0 und manchmal von 1 an indiziert, d.h. $w = w_0 \cdots w_{n-1}$ bzw. $w = w_1 \cdots w_n$, je nachdem, was im Kontext praktischer ist.
  Bsp.: $\Sigma = \{a, b\}$, dann ist $w = abba$ ein Wort über Σ.
- 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.
- Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Die Menge aller Wörter der Länge größer gleich 1 über Σ wird mit $\Sigma^+ := \Sigma^* \backslash \{\varepsilon\}$ bezeichnet. Die Menge aller Wörter über Σ der Länge k wird mit $\Sigma^k \subseteq \Sigma^*$ bezeichnet.
- Sei $w = w_1 \cdots w_n$ ein Wort der Länge n ($w_i \in \Sigma$). Im Folgenden bezeichne $[a : b] = \{n \in \mathbb{Z} \mid a \leq n \land n \leq b\}$ für $a, b \in \mathbb{Z}$.
  - Ist $w' = w_1 \cdots w_l$ mit $l \in [0 : n]$, dann heißt w' *Präfix* von w.
    (für $l = 0 \Rightarrow w' = \varepsilon$)
  - Ist $w' = w_l \cdots w_n$ mit $l \in [1 : n + 1]$, dann heißt w' *Suffix* von w.
    (für $l = n + 1 \Rightarrow w' = \varepsilon$)
  - Ist $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$, dann heißt w' *Teilwort* von w.
    (wobei $w_i \cdots w_j = \varepsilon$ für $i > j$)

  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.

Problem:
Geg.: $s \in \Sigma^*$; $|s| = m$; $t \in \Sigma^*$; $|t| = n \geq m$
Ges.: $\exists\, i \in [0 : n - m]$ 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 (→ 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.

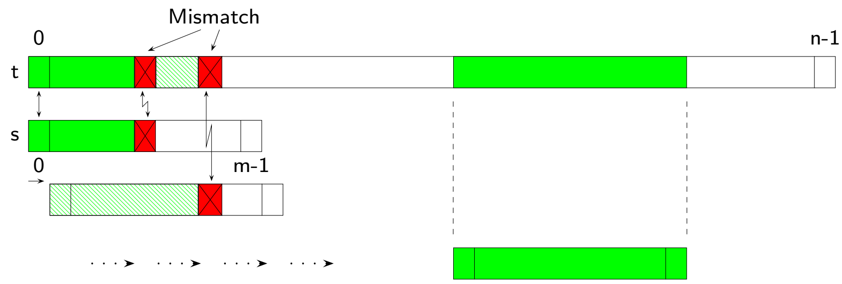


Abbildung 2.1: Skizze: Suchen mit der naiven Methode

Definition 2.1 *Stimmen beim Vergleich zweier Zeichen diese nicht überein, so nennt man dies einen* Mismatch.

In der folgenden Abbildung ist der naive Algorithmus in einem C-ähnlichen Pseudocode angegeben.

```
BOOL NAIV (char t[], int n, char s[], int m)
{
    int i = 0, j = 0;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i++;
        j = 0;
    }
    return FALSE;
}
```

Abbildung 2.2: Algorithmus: Die naive Methode

2.2.2 Laufzeitanalyse des naiven Algorithmus:

Um die Laufzeit abzuschätzen, zählen wir die Vergleiche von Symbolen aus Σ:

- Äußere Schleife wird $(n - m + 1)$-mal durchlaufen.
- Innere Schleife wird maximal m-mal durchlaufen.

$\Rightarrow$ Test wird $((n - m + 1) * m)$-mal durchlaufen $= O(n * m) \longleftarrow$ Das ist zu viel!

Beispiel:

1.) $t \;=\; a\,a\,a\,a\,a\,a\,a\,a\,a\,a$

$\checkmark\,\checkmark\,\checkmark\,\times$
$a\,a\,a\,b$
$\quad\checkmark\,\checkmark\,\checkmark\,\times$
$\quad a\,a\,a\,b$
$\qquad\checkmark\,\checkmark\,\checkmark\,\times$
$\qquad a\,a\,a\,b$

2.) $t \;=\; a\,a\,b\,a\,a\,b\,a\,a\,b\,a\,a\,b\,a\,a\,b$

$\checkmark\,\checkmark\,\checkmark\,\checkmark\,\checkmark\,\times$
$a\,a\,b\,a\,a\,a$
$\quad\checkmark\,\times$
$\quad a\,a\,b\,a\,a\,a$
$\qquad\times$
$\qquad a\,a\,b\,a\,a\,a$
$\qquad\quad\checkmark\,\checkmark\,\checkmark\,\checkmark\,\checkmark\,\times$
$\qquad\quad a\,a\,b\,a\,a\,a$

Abbildung 2.3: Beispiel: Suchen mit der naiven Methode

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

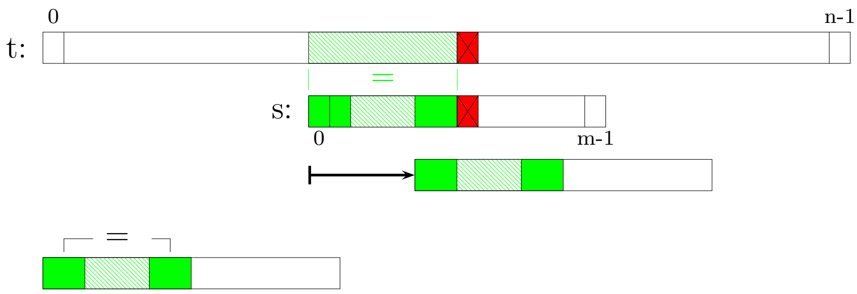


Abbildung 2.4: Skizze: Eine Idee für größere Shifts

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

Definition 2.2 *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 eigentlicher Rand, *wenn $r \neq w$ und wenn es außer w selbst keinen längeren Rand gibt.*

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 Rand, aber nicht der eigentliche Rand.

Definition 2.3 *Ein Verschiebung der Anfangsposition i des zu suchenden Wortes (d.h. eine Erhöhung des Index $i \to i'$) heißt* Shift.

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

Wir definieren:

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

wobei $\partial(s)$ den eigentlichen Rand von s bezeichnet.

Lemma 2.4 *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.*

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 folgende 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 rechte Rand des Teilwortes $s_0 \cdots s_{j-1}$ von s auf dem linken 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. ∎

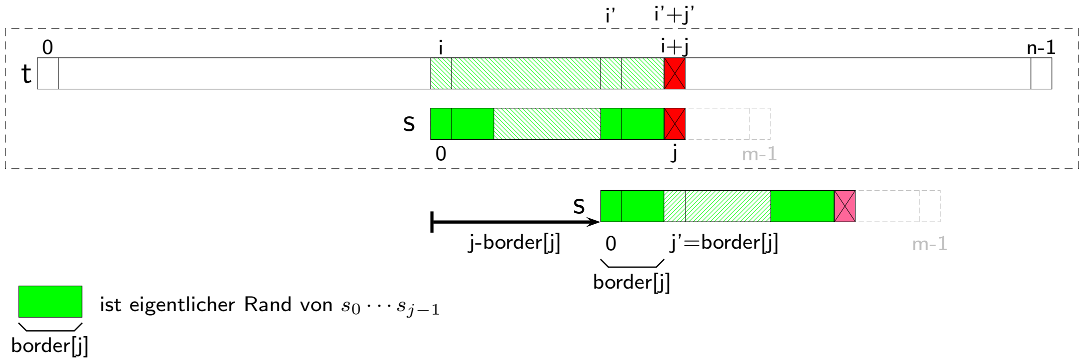


Abbildung 2.5: Skizze: Der Shift um $j - border[j]$

2.2.4 Der Knuth-Morris-Pratt-Algorithmus

Wenn wir die Überlegungen aus dem vorigen Abschnitt in einen Algorithmus übersetzen, erhalten wir den sogenannten KMP-Algorithmus, benannt nach D.E. Knuth, J. Morris und V. Pratt.

BOOL KNUTH-MORRIS-PRATT (char $t[]$, int n, char $s[]$, int m)

```
{
    int border[m + 1];
    compute_borders(int border[], int m, char s[]);
    int i = 0, j = 0;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            j++;
            if (j == m) return TRUE;
        }
        i = i + j − border[j];   // j − border[j] > 0
        j = max{0, border[j]};
    }
    return FALSE;
}
```

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

2.2.5 Laufzeitanalyse des KMP-Algorithmus:

Strategie: Zähle Anzahl der Vergleiche getrennt nach erfolglosen und erfolgreichen Vergleichen. Ein Vergleich von zwei Zeichen heißt erfolgreich, wenn die beiden Zeichen gleich sind, und *erfolglos* sonst.

erfolglose Vergleiche:

Es werden maximal $n - m + 1$ erfolglose Vergleiche ausgeführt, da nach jedem erfolgten Vergleich $i \in [0 : n - m]$ erhöht und nie erniedrigt wird.

erfolgreiche Vergleiche:

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

Wert von $i + j$ vorher: $i + j$
Wert von $i' + j'$ nachher: $\underbrace{(i + j - border[j])}_{i'} + \underbrace{(max\{0, border[j]\})}_{j'}$

<u>1.Fall:</u> $border[j] \geq 0 \quad \Rightarrow \; i' + j' = i + j$
<u>2.Fall:</u> $border[j] = -1 \; (\Leftrightarrow j = 0) \; \Rightarrow \; i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$

$\Rightarrow$ Nach einem erfolglosen Vergleich wird $i + j$ <u>nicht</u> kleiner!
Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht!

$\Rightarrow$ Die maximale Anzahl erfolgreicher Vergleiche ist durch n beschränkt, da $i + j \in [0 : n - 1]$.

$\Rightarrow$ Somit werden insgesamt maximal $2n - m + 1$ Vergleiche ausgeführt.

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.

Initialisierung: $border[0] = -1$
$border[1] = 0$

Annahme: $border[0] \cdots border[j - 1]$ seien bereits berechnet.

Ziel: Berechnung von $border[j]$ = Länge des eigentlichen Randes eines Suffixes der Länge j.

Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j - 1] + 1$. 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

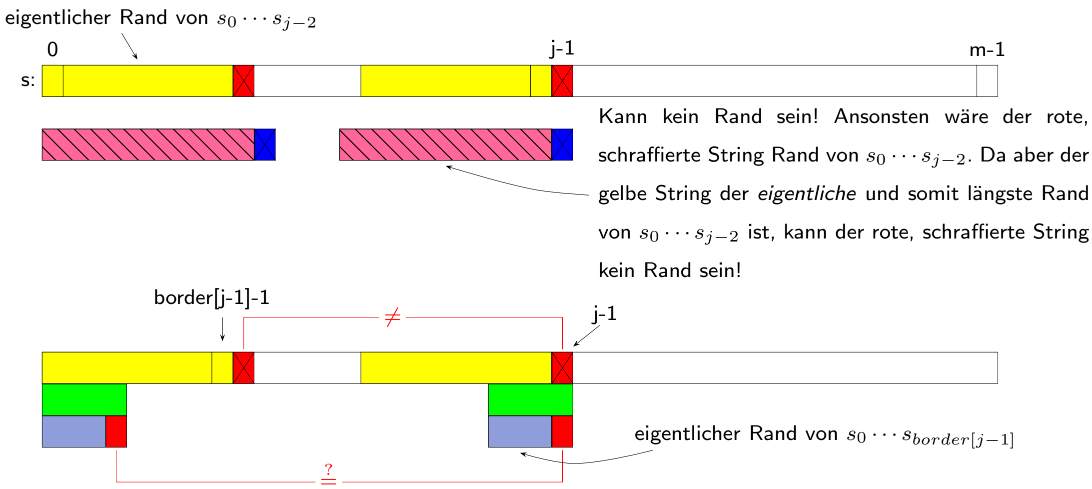


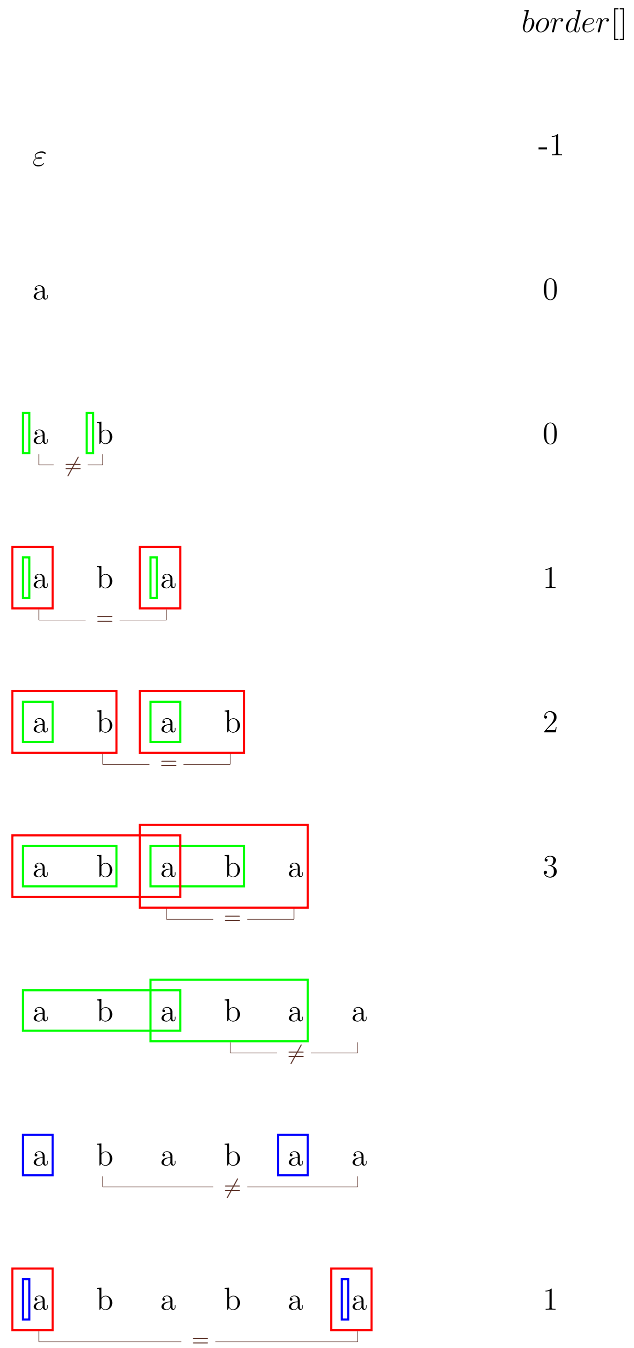
Abbildung 2.7: Skizze: Berechnung von $border[j]$

betrachteten Randes dieses Wortes. Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das 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 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 Algorithmus zur Berechnung der Tabelle *border*.

```
COMPUTE_BORDERS (int border[], int m, char s[])
{
    border[0] = −1;
    int i = border[1] = 0;
    for (int j = 2; j ≤ m; j++)
    { /* Beachte, dass hier gilt: i == border[j − 1] */
        while ((i ≥ 0) && (s[i] ≠ s[j − 1]))
            i = border[i];
        i++;
        border[j] = i;
    }
}
```

Abbildung 2.8: Algorithmus: Berechnung der Tabelle *border*



grün: der bekannte Rand
rot: der verlängerte (neu gefundene) Rand
blau: der "Rand des Randes"

Abbildung 2.9: Beispiel: Berechnung der Tabelle *border* für *ababaa*

2.2.7 Laufzeitanalyse:

Wieder zählen wir die Vergleiche getrennt nach erfolgreichen und erfolglosen Vergleichen.

Anzahl erfolgreicher Vergleiche:
Es kann maximal $m-1$ erfolgreiche Vergleiche geben, da jedes Mal $j \in [2:m]$ um 1 erhöht und nie erniedrigt wird.

Anzahl erfolgloser Vergleiche:
Betrachte i zu Beginn: $i = 0$
Nach jedem erfolgreichen Vergleich wird i inkrementiert
$\Rightarrow$ i wird $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
$\stackrel{i \geq -1}{\Rightarrow}$ i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da $i \geq -1$
(Es kann nur das weggenommen werden, was schon einmal hinzugefügt wurde; das „plus eins" kommt daher, dass zu Beginn $i = 0$ und ansonsten immer $i \geq -1$ gilt.).

$$\Rightarrow \text{Anzahl der Vergleiche} \leq 2m - 1$$

Theorem 2.5 *Der Algorithmus von Knuth, Morris und Pratt benötigt maximal $2n + m$ Vergleiche, 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.

Geg.: Ein Text t der Länge n und eine Menge $S = \{s^1, \ldots, s^l\}$ mit $\sum_{s \in S} |s| = m$.
Ges.: 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

Wende den KMP-Algorithmus für jedes Suchwort $s \in S$ auf t an.

Kosten des Preprocessing (Erstellen der Border-Tabellen):

$$\sum_{i=1}^{l}(2|s^i| - 2) \leq \sum_{i=1}^{l} 2|s^i| = 2m.$$

Kosten des eigentlichen Suchvorgangs:

$$\sum_{i=1}^{l}(2n - |s^i| + 1) \leq 2l * n - m + l.$$

Somit sind die Gesamtkosten $O(l * n + m)$. Ziel ist die Elimination des Faktors l.

2.3.2 Der Algorithmus von Aho und Corasick

Zuerst werden die Suchwörter in einem so genannten Suchwort-Baum organisiert. In einem *Suchwort-Baum* gilt folgendes:

- Der Suchwort-Baum ist gerichteter Baum mit Wurzel r;
- Jeder Kante ist als Label ein Zeichen aus Σ zugeordnet;
- Die von einem Knoten ausgehenden Kanten besitzen verschiedene Labels;
- Jedes Suchwort $s \in S$ wird auf einen Knoten v abgebildet, so dass s entlang des Pfades von r nach v steht;
- Jedem Blatt ist ein Suchwort zugeordnet.

In der Abbildung auf der nächsten Seite ist ein solcher Suchwort-Baum für die Menge $\{aal, aas, aus, sau\}$ angegeben.

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.

Damit es zu keinen Sackgassen kommt, werden in den Baum so genannten Failure-Links eingefügt.

Failure-Links: Verweis von einem Knoten v auf einen Knoten w im Baum, so dass die Kantenbezeichnungen von der Wurzel zu dem Knoten w den längsten Suffix des bereits erkannten Teilwortes bilden (= Wort zu Knoten v).

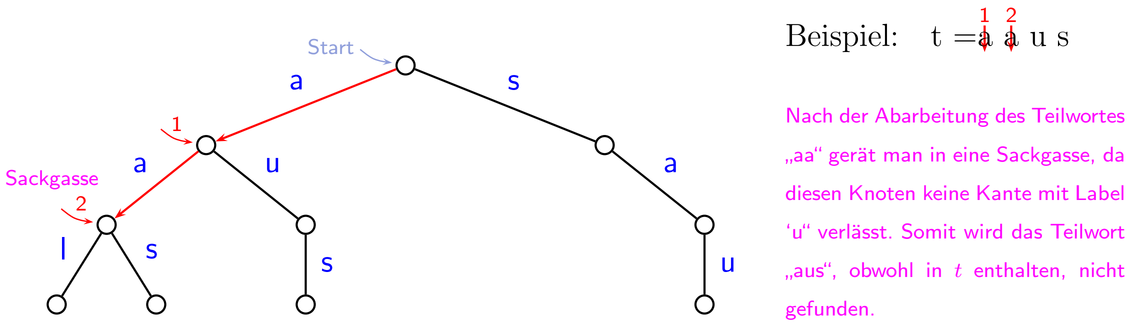


Abbildung 2.10: Beispiel: Aufbau des Suchwort-Baumes für $\{aal, aas, aus, sau\}$

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.

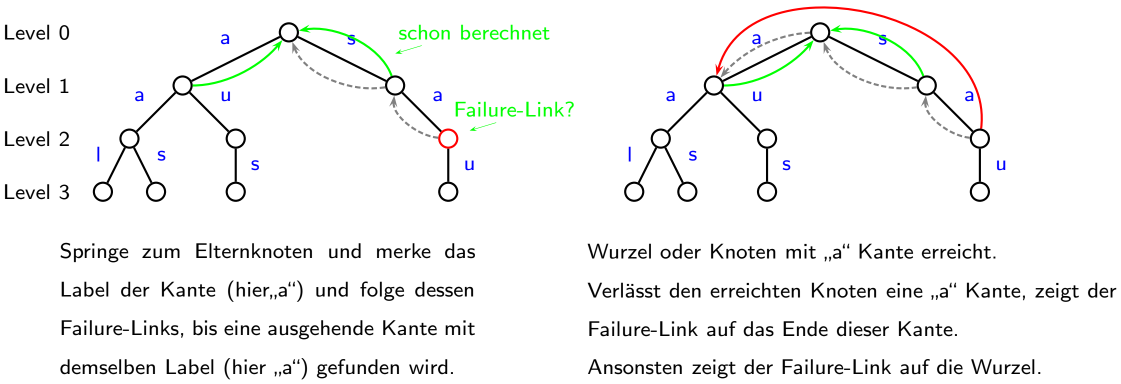


Abbildung 2.11: Beispiel: für die Berechnung eines Failure-Links

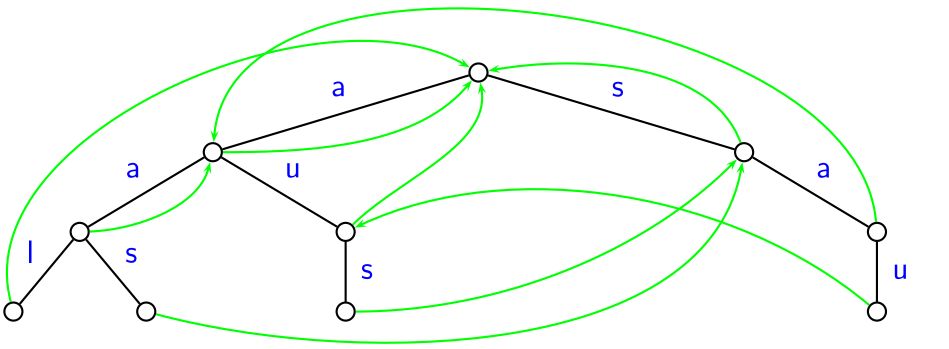


Abbildung 2.12: Beispiel: Der komplette Baum mit allen Failure-Links

Ein Suchwort 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

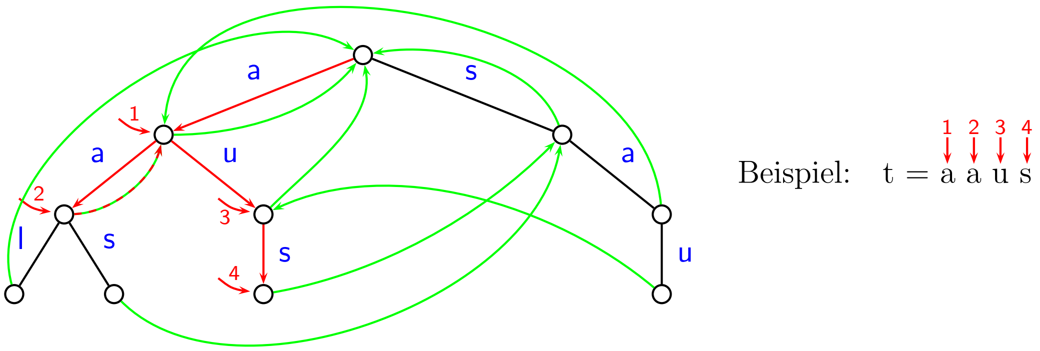
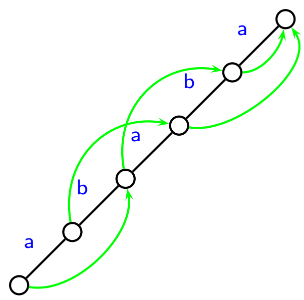


Abbildung 2.13: Beispiel: Suche in $t = aaus$, nur mit Failure-Links

Label, so folgen wir dem Failure-Link und suchen von dem so aufgefundenen Knoten aus weiter.

Ist die Menge S einelementig, so erhalten wir als Spezialfall die Tabelle *border* des KMP-Algorithmus. Im Suchwort-Baum wird dabei auf den entsprechenden längsten 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.



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 2.14: Beispiel: $S = \{ababa\}$

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 $O(|s|)$ Vergleiche ausgeführt werden.

Wie bei der Analyse der Berechnung der Tabelle *border* des KMP-Algorithmus unterscheiden wir erfolgreiche und erfolglose Vergleiche. Zuerst halten wir fest, dass es maximal $O(|s|)$ erfolgreiche Vergleiche (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 Vergleiche beachten wir, dass Failure-Links immer nur zu Knoten auf

Berechnung der Failure-Links (tree $T = (V, E)$)

```
{
    forall v ∈ V
    {
        Sei v' der Elternknoten von v mit v' --x--> v ∈ E
        w = Failure_Link(v')
        while (!(w --x-->) && (w ≠ root))
            w = Failure_Link(w)
        if (w --x--> w') Failure_Link(v) = w'
        else Failure_Link(v) = root
    }
}
```

Abbildung 2.15: Algorithmus: Berechnung der Failure-Links

einem niedrigeren Level verweisen. Bei jedem erfolglosen Vergleich springen wir also zu einem Knoten auf einem niedrigeren Level. Da wir nur bei einem erfolgreichen Vergleich zu einem höheren Level springen können, kann es nur so viele erfolglose wie erfolgreiche Vergleiche geben.

Somit ist die Anzahl Vergleiche für jedes Wort $s \in S$ durch $O(|s|)$ beschränkt. Damit ergibt sich insgesamt für die Anzahl der Vergleiche

$$\leq \sum_{s \in S} O(|s|) = O(m).$$

In der Abbildung auf der nächsten Seite ist der Algorithmus von A. Aho und M. Corasick im Pseudocode angegeben.

Auch hier verläuft die Laufzeitanalyse ähnlich wie beim KMP-Algorithmus. Da $level(v)$-$level(\text{Failure\_Link}(v)) > 0$ (analog zu $j - border[j] > 0$) ist, wird nach jedem erfolglosen Vergleich i um mindestens 1 erhöht. Also gibt es maximal $n-m+1$ erfolglose Vergleiche. Weiterhin kann man zeigen, dass sich nach einem erfolglosen Vergleich $i + level(v)$ nie erniedrigt und nach jedem erfolgreichen Vergleich um 1 erhöht (da sich $level(v)$ um 1 erhöht). Da $i + j \in [0 : n-1]$ ist, können maximal n erfolgreiche und somit maximal $2n - m + 1$ Vergleiche überhaupt ausgeführt worden sein.

2.3.3 Korrektheit von Aho-Corasick

Es bleibt nur noch, die Korrektheit des vorgestellten Algorithmus von Aho und Corasick nachzuweisen. Wenn kein Muster in t auftritt ist klar, dass der Algorithmus

```
BOOL AHO-CORASICK (char t[], int n, char S[], int m)
{
    int i = 0
    tree T(S) // Suchwort-Baum, der aus den Wörtern in s konstruiert wurde
    node v = root
    while (i < n)
    {
        while ((v --t_{i+level(v)}--> v') in T)
        {
            v = v'
            if (v' ist Blatt) return TRUE;
        }
        i = i + level(v) − level(Failure_Link(v))
        v = Failure_Link(v)
    }
}
```

Abbildung 2.16: Algorithmus: Die Methode von Aho und Corasick

nicht behauptet, dass ein Suchwort auftritt. Wir beschränken uns also auf den Fall, dass eines der Suchwörter aus S in t auftritt.

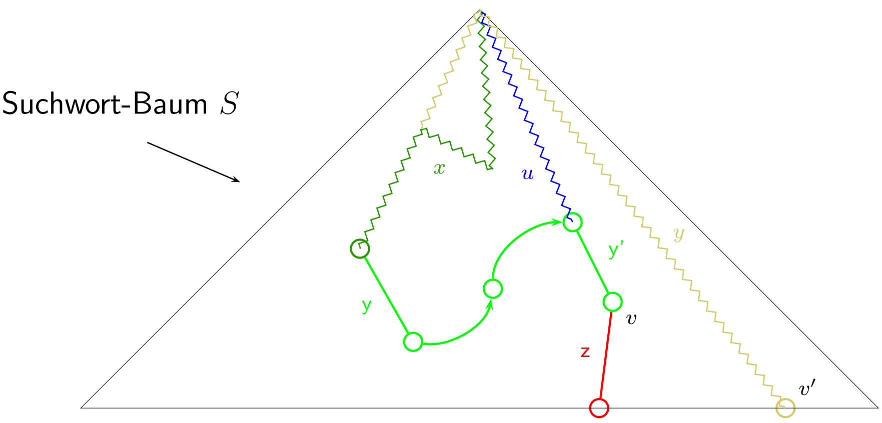


Abbildung 2.17: Skizze: Im Suchwort-Baum wird Treffer von y gemeldet

Was passiert, wenn $y \in S$ als ein Teilwort von t auftritt und sich der Algorithmus unmittelbar nach dem Lesen von y in einem internen Knoten v befindet? Sei uy' das Wort, über den man den Knoten v auf einem einfachen Pfad von der Wurzel aus erreicht, wobei y' das Teilwort des Pfades ist, das während des Algorithmus auf diesem Pfad abgelaufen wurde. Zuerst halten wir fest, dass y' ein Suffix von y ist.

Wir behaupten, dass y ein Suffix von uy' ist. Da $y \in S$, gibt es im Suchwortbaum einen Pfad von der Wurzel zu einem Blatt v', der mit y markiert ist. Somit kann man nach der Verarbeitung von y als Teilwort von t nur an einem Knoten landen, dessen Level mindestens $|y|$ ist (ansonsten müssten wir bei v' gelandet sein). Somit ist $level(v) \geq |y|$. Nach Definition der Failure-Links muss dann die Beschriftung uy' des Pfades von der Wurzel zu v mindestens mit y enden.

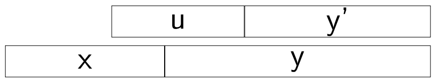


Abbildung 2.18: Skizze: y muss Suffix von uy' sein

Ist z die Beschriftung eines Pfades von der Wurzel zu einem Blatt, das von v aus über normale Baumkanten erreicht werden kann, dann muss y ein echtes Teilwort von $uy'z \in S$ sein. Dies widerspricht aber der Annahme, dass kein Suchwort aus S ein echtes Teilwort eines anderen Suchwortes in S sein kann. Damit befindet sich der Algorithmus von Aho-Corasick nach dem Auffinden eines Suchwortes $s \in S$ in einem Blatt des Baumes. Da y ein Suffix von uy' ist, ist y ein Teilwort von $uy'z \in S$.

Theorem 2.6 *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 $t \in \Sigma^*$ in der Zeit $O(n+m)$, wobei $n = |t|$ und $m = \Sigma_{s \in S}|s|$.*

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 $t's$ der 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 $level(v') = |s'| < |s|$. 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

$level(w) \geq |\bar{s}|$ 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 $level(w') < |\bar{s}|$, 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 Kantenbeschriftungen 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 das Blatt 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 höheren Level zeigen, wobei die Beschriftung dieses Pfades dann s als Suffix beinhalten muss. Eine Wiederholung dieser Argumentation zeigt, dass letztendlich über die Failure-Links das Blatt v besucht wird. Daher genügt es, den Failure-Links zu folgen, bis ein Blatt erreicht wird. 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 muss also wie folgt erweitert werden. Wann immer wir einen neuen Knoten über eine Baumkante erreichen, müssen wir testen, ob über eine Folge von Failure-Links (eventuell auch keine) ein Blatt 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 alle Blätter als Treffer, die Wurzel als Nicht-Treffer und alle internen Knoten als unbekannt. Dann durchlaufen wir alle als unbekannt markierten Knoten des Baumes. Von jedem solchen Knoten aus folgen wir solange den Failure-Links bis wir auf einen mit Treffer oder Nicht-Treffer markierten Knoten treffen. Anschließend markieren wir alle Knoten auf diesem Pfad genau so wie den gefundenen Knoten. 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 jedoch zusätzlich auch noch die Endpositionen aller Treffer ausgeben, so müssen wir den Algorithmus noch weiter modifizieren. Wir hängen zusätzlich an die Knoten mit Treffer noch eine Liste mit den Längen der Suchworte an, die an dieser Position enden können. Zu Beginn erhält jedes Blatt eine einelementige Liste, die die Länge des zugehörigen Suchwortes beinhaltet. Alle andere Knoten bekommen eine leere Liste. Finden wir nun einen Knoten, der als Treffer markiert ist, so wird dessen Liste an alle Listen der Knoten auf dem Pfad dorthin als Kopie angefügt. Treffen wir nun beim Algorithmus von Aho-Corasick auf einen als Treffer markierten Knoten, so müssen jetzt mehrere Antworten ausgegeben werden (die Anzahl entspricht der Elemente der Liste). 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.

Theorem 2.7 *Sei $S \subseteq \Sigma^*$ eine Menge von Suchwörtern, dann findet der Algorithmus von Aho-Corasick alle Vorkommen von $s \in S$ in $t \in \Sigma^*$ in der Zeit $O(n+m+k)$, wobei $n = |t|$, und $m = \Sigma_{s \in S}|s|$ 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 Suchen in Texten ist der, das gesuchte Wort mit einem Textstück nicht mehr von links nach rechts, sondern von rechts nach links zu vergleichen. Ein Vorteil dieser Vorgehensweise ist, dass man in Alphabeten mit vielen verschiedenen Symbolen bei einem Mismatch an der Position $i+j$ in t, i auf $i+j+1$ setzen kann, falls das im Text t gesuchte Zeichen t_{i+j} gar nicht im gesuchten Wort s vorkommt. D.h. in so einem Fall kann man das Suchwort gleich um $j+1$ verschieben.

Diese naive Idee ohne den Trick, bei einen Mismatch mit einem Zeichen aus t, das in s gar nicht auftritt, das Muster s möglichst weit zu schieben, ist im Algorithmus auf der nächsten Seite wiedergegeben.

Das folgende Bild 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 Vergleiche nach einem Shift keine Übereinstimmung mehr herrscht. Diese Betrachtung ist völlig analog zur Schiebe-Operation im KMP-Algorithmus.

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

```
BOOL NAIV2 (char t[], int n, char s[], int m)
{
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i++;
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 2.19: Algorithmus: Eine naive Methode mit rechts-nach-links Vergleichen

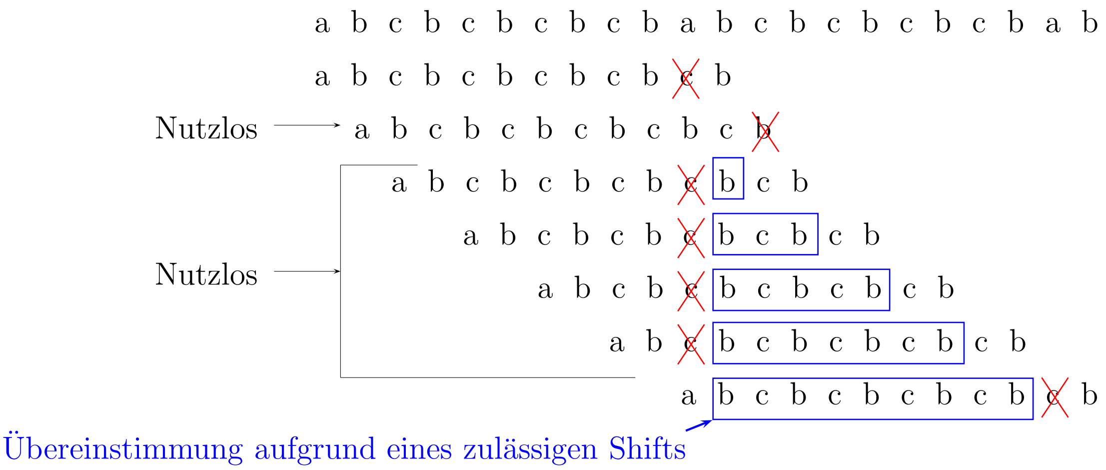


Abbildung 2.20: Beispiel: Suchen mit der zweiten naiven Methode

Zeichen zum Liegen kommt, das schon vorher den Mismatch ausgelöst hat. Daher kann man auch den dritten bis sechsten Versuch als nutzlos bezeichnen.

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. Folgende Skizze zeigt die möglichen Shifts beim Boyer-Moore-Algorithmus:

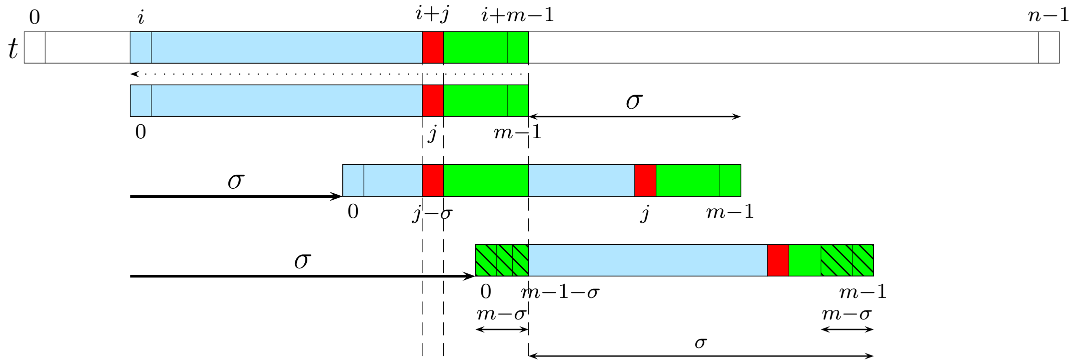


Abbildung 2.21: Skizze: Zulässige Shifts bei Boyer-Moore (Strong-Good-Suffix-Rule)

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 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, so dass an der Position des Mismatches in t gar kein weiterer Vergleich 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 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 Boyer-Moore-Algorithmus sieht dann wie in der Abbildung auf der nächsten Seite aus. Hierbei ist $S[]$ die 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 Vergleiche ausgeführt, da es letztendlich doch zu aufwendig ist, sich diesen Bereich explizit zu merken und bei folgenden Vergleichen von s in t zu überspringen.

```
BOOL Boyer-Moore (char t[], int n, char s[], int m)
{
    int S[m+1];
    compute_shift_table(S, m, s);
    int i = 0, j = m − 1;
    while (i ≤ n − m)
    {
        while (t[i + j] == s[j])
        {
            if (j == 0) return TRUE;
            j−−;
        }
        i = i + S[j];
        j = m − 1;
    }
    return FALSE;
}
```

Abbildung 2.22: Algorithmus: Boyer-Moore mit Strong-Good-Suffix-Rule

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 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 obigen Bild 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 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.

2.4.3 Bestimmung der Shift-Tabelle

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-Algorithmus, da hier, zusätzlich zu den Rändern von Präfixen des Suchwortes, auch die Ränder von Suffixen des Suchwortes gesucht sind.

```
COMPUTE_SHIFT_TABLE (int S[], char s[], int m)
{
    /* Initialisierung von S[] */
    for (int j = 0; j ≤ m; j++) S[j] = m;
    /* Teil 1: σ ≤ j */
    int border2[m + 1];
    border2[0] = −1;
    int i = border2[1] = 0;
    for (int j' = 2; j' ≤ m; j'++)
    {   /* Beachte, dass hier gilt: i == border2[j' − 1] */
        while ((i ≥ 0) && (s[m − i − 1] ≠ s[m − j']))
        {
            int σ = j' − i − 1;
            S[m − i − 1] = min(S[m − i − 1], σ);
            i = border2[i];
        }
        i++;
        border2[j'] = i;
    }
    /* Teil 2: σ > j */
    int j = 0;
    for (int i = border2[m]; i ≥ 0; i = border2[i])
    {
    int σ = m − i;
        while (j < σ)
        {
            S[j] = min(S[j], σ);
            j++;
        }
    }
}
```

Abbildung 2.23: Algorithmus: Berechnung der Shift-Tabelle für Boyer-Moore

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. Diese Situation entspricht dem Fall in der Berechnung eines eigentlichen Randes, bei dem der vorhandene Rand nicht zu einem längeren Rand fortgesetzt werden kann (nur werden Suffixe statt Präfixe betrachtet). Daher sieht die erste for-Schleife genau so aus, wie in der Berechnung von compute_border. 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.

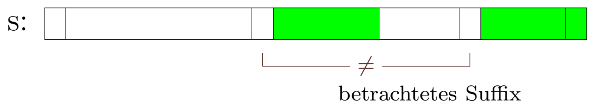


Abbildung 2.24: Skizze: Verlängerung eines Randes eines Suffixes

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 der Suffix der Länge m von s gerade wieder s ist. Was ist der nächstkürzerere 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 den 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.

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.

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

Die Shift-Tabelle $S[j]$ wird für alle $j \in [0 : m-1]$ mit der Länge m des Suchstrings vorbesetzt.

Teil 1: $\sigma \leq j$

	0	1	2	3	4	5	6	7	8	9		
	a	b	a	b	b	a	b	a	b	a		
												Mismatch
border2[1]=0									b	a	$\longrightarrow$	S[9] = 1
border2[2]=0								a	b	a		
border2[3]=1							b	a	b	a		
border2[4]=2						a	b	a	b	a		
border2[5]=3					b	a	b	a	b	a		
border2[6]=4				b	b	a	b	a	b	a	$\longrightarrow$	S[5] = 2
				b	b	a	b	a	b	a	$\longrightarrow$	S[7] = 4
border2[7]=0				b	b	a	b	a	b	a	$\longrightarrow$	S[9] = 6*
border2[8]=1		b	a	b	b	a	b	a	b	a		
border2[9]=2	a	b	a	b	b	a	b	a	b	a		
border2[10]=3	a	b	a	b	b	a	b	a	b	a		

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 2.25: Beispiel: $s = ababbababa$ (Teil 1: $\sigma \leq j$)

Teil 2: $\sigma > j$

$\boxed{\text{a b a}}$ b b a b $\boxed{\text{a b a}}$

$\xrightarrow[\sigma = 7]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [0:6]$ S[j] = 7

$\boxed{\text{a}}$ b a b b a b a b $\boxed{\text{a}}$

$\xrightarrow[\sigma = 9]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [7:8]$ S[j] = 9

a b a b b a b a b a

$\xrightarrow[\sigma = 10]{}$ a b a b b a b a b a

bei Mismatch an Position $j \in [9]$ S[j] = 10

Zusammenfassung:

	1.Teil	2.Teil	Erg		1.Teil	2.Teil	Erg
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

Abbildung 2.26: Beispiel: $s = ababbababa$ (Teil 2: $\sigma > j$)

beiden Schleifen des zweiten Teils ergibt, dass die Schleifen nur m-mal durchlaufen werden und dort überhaupt keine Vergleiche ausgeführt werden.

Lemma 2.8 *Die Shift-Tabelle des Boyer-Moore-Algorithmus lässt sich für eine Zeichenkette der Länge m mit maximal $2m$ Vergleichen berechnen.*

Es bleibt demnach nur noch, die Anzahl der Vergleiche in der Hauptprozedur zu bestimmen. Hierbei wird nur die Anzahl der Vergleiche für eine erfolglose Suche bzw. für das erste Auftreten des Suchwortes s im Text t betrachtet. Wir werden zum Abzählen der Vergleiche wieder einmal zwischen zwei verschiedenen Arten von Vergleichen unterscheiden: *initiale* und *wiederholte* Vergleiche.

Initiale Vergleiche:
Vergleiche von s_j mit t_{i+j}, so dass t_{i+j} zum ersten Mal beteiligt ist.
Wiederholte Vergleiche:
Beim Vergleich s_j mit t_{i+j} war t_{i+j} schon früher bei einem Vergleich beteiligt.

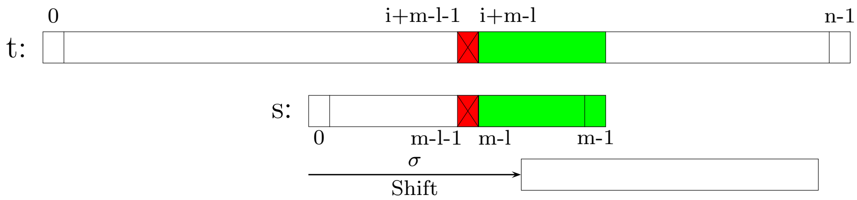


Abbildung 2.27: Skizze: Shift nach ℓ erfolgreichen Vergleichen

Lemma 2.9 *Sei $s_{m-\ell} \cdots s_{m-1} = t_{i+m-\ell} \cdots t_{i+m-1}$ für $i \in [0 : n - m]$ und $\ell \in [0 : m - 1]$ sowie $s_{m-\ell-1} \neq t_{i+m-\ell-1}$ (es wurden also ℓ erfolgreiche und ein erfolgloser Vergleich durchgeführt). Dabei seien I initiale Vergleiche durchgeführt worden, und weiter sei σ die Länge des folgenden Shifts gemäß der Strong-Good-Suffix-Rule. Dann gilt:*

$$\ell + 1 \leq I + 4 * \sigma.$$

Aus diesem Lemma folgt, dass maximal $5n$ Vergleiche ausgeführt wurden. Bezeichne dazu $V(n)$ die Anzahl aller Vergleiche, um in einem Text der Länge n zu suchen, und I_i bzw. V_i die Anzahl der initialen bzw. aller Vergleiche, die beim i-ten Versuch ausgeführt wurden. Ferner sei σ_i die Länge des Shifts, der nach dem i-ten Vergleich ausgeführt wird. War der i-te (und somit letzte) Vergleich erfolgreich, so sei $\sigma_i := m$ (Unter anderem deswegen gilt die Analyse nur für einen erfolglosen bzw. den ersten erfolgreichen Versuch).

$$\begin{aligned}
V(n) &= \sum_i V_i \\
& \text{ mit Hilfe des obigen Lemmas} \\
&\leq \sum_i (I_i + 4\sigma_i) \\
& \text{ da es maximal } n \text{ initiale Vergleiche geben kann} \\
&\leq n + 4\sum_i \sigma_i \\
& \text{ da die Summe der Shifts maximal } n \text{ sein kann} \\
& \text{ (der Anfang von } s \text{ bleibt innerhalb von } t\text{)} \\
&\leq n + 4n \; = \; 5n
\end{aligned}$$

Zum Beweis des obigen Lemmas nehmen wir an, dass ℓ erfolgreiche und ein erfolgloser Vergleich 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*.

1.Fall: Shift σ ist lang, d.h. $\sigma \geq \lceil \ell/4 \rceil$:

$$\begin{aligned} \text{Anzahl der ausgeführten Vergleiche} &= \ell + 1 \\ &\leq 1 + 4 * \lceil \ell/4 \rceil \\ &\leq 1 + 4 * \sigma \\ &\leq I + 4 * \sigma \end{aligned}$$

Die letzte Ungleichung folgt aus der Tatsache, dass es bei jedem Versuch immer mindestens einen initialen Vergleich 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 Vergleich beteiligt war).

2.Fall: 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 $\alpha \in \Sigma^+$, so dass s mit α^5 enden muss. Betrachten wir dazu die folgende Skizze:

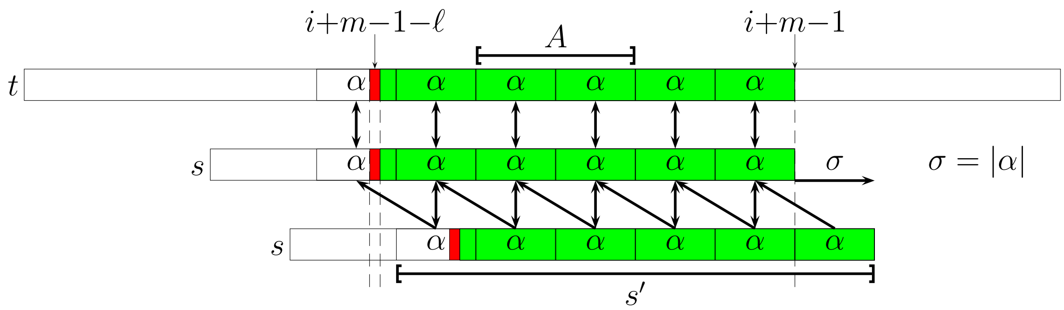


Abbildung 2.28: Skizze: Periodisches Ende von s bei kurzen Shifts

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. 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.

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 Vergleich beteiligt waren. Dazu betrachten wir frühere Versuche, die Vergleiche im Abschnitt A hätten ausführen können.

Zuerst betrachten wir einen früheren Versuch, bei dem mindestens σ erfolgreiche Vergleiche ausgeführt worden sind. Dazu betrachten wir die folgende Abbildung, wobei der betrachtete Versuch oben dargestellt ist. Da mindestens σ erfolgreiche Vergleiche ausgeführt wurden, folgt aus der Periodizität von s', dass alle Vergleiche bis zur Position $i+m-\ell-2$ erfolgreich sein müssen. Der Vergleich 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.

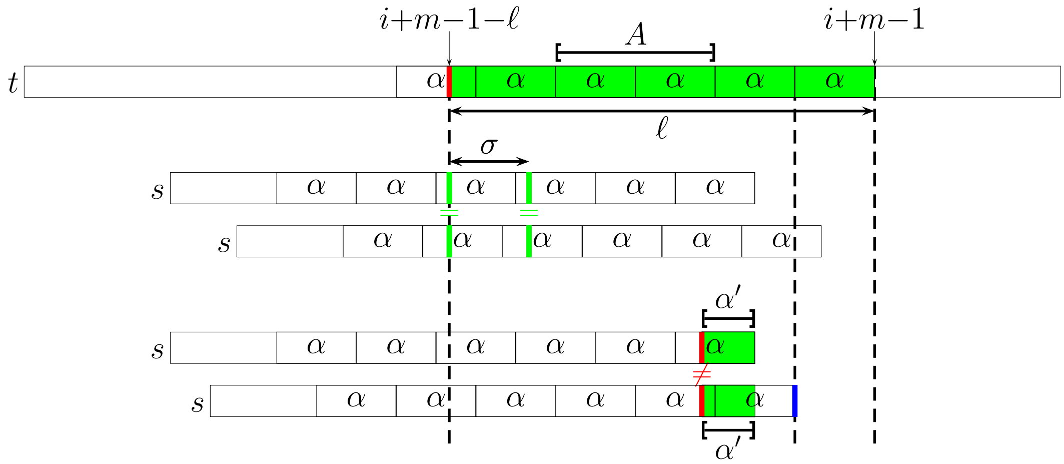


Abbildung 2.29: Skizze: Mögliche frühere initiale Vergleich im Bereich A

Nehmen wir an, es gäbe einen kürzeren sicheren Shift (wie in der Abbildung 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 Vergleich des früheren Versuchs, da sich in s' die Zeichen alle σ Zeichen wiederholen. Damit erhalten wir einen Widerspruch zur Strong-Good-Suffix-Rule. Falls der Shift um s Zeichen zu kurz ist, damit es zu einem Vergleich mit dem Zeichen an Position $i+m-\ell-1$ kommen kann, hätten wir s in t gefunden und wir hätten keinen Versuch an Position i unternommen.

Es bleiben noch die früheren Versuche, die weniger als σ Vergleiche ausgeführt haben. Da wir nur Versuche betrachten, die Vergleiche im Abschnitt A ausführen, muss der Versuch an einer Position im Intervall $[i-(k-2)\sigma : i-\sigma-1]$ von rechts nach links begonnen haben. Nach Wahl von A muss der erfolglose Vergleich auf einer Position größer als $i+m-\ell-1$ erfolgen. Betrachte hierzu die erste Zeile im unteren Teil der obigen Abbildung. Sei α' das Suffix von α (und damit von s' bzw. s), in dem die erfolgreichen Vergleiche 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 $[i - m - \ell : i + m - \sigma - 1]$.

Wir werden jetzt zeigen, dass ein Shift auf $i - \sigma$ (wie im unteren Teil der Abbildung darunter dargestellt) zulässig ist. Die alten erfolgreichen Vergleiche 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 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 Vergleiche 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 $i - \sigma$ und wir können dieselbe Argumentation wiederholen. Also erhalten wir letztendlich immer einen Shift auf eine Position größer als i, aber nie einen Shift auf 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 Vergleich des letzten Zeichens von s mit einem Zeichen aus t ein initialer gewesen sein musste, folgt dass mindestens $1 + |A|$ initiale Vergleiche ausgeführt wurden. Da $|A| \geq \ell - 4\sigma$, erhalten wir die gewünschte Abschätzung:

$$1 + \ell = (1 + |A|) + (\ell - |A|) \leq (1 + |A|) + 4\sigma.$$

Da wie schon weiter oben angemerkt, der Vergleich des letzten Zeichens von s immer initial sein muss, und alle Vergleiche im Bereich A initial sind, folgt damit die Behauptung des Lemmas.

Theorem 2.10 *Der Boyer-Moore Algorithmus benötigt für das erste Auffinden des Musters s in t maximal $5n$ Vergleiche ($|s| \leq |t| = n$)*

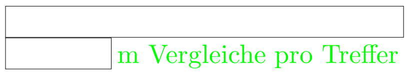
Mit Hilfe einer besseren, allerdings für die Vorlesung zu aufwendigen Analyse, kann man folgendes Resultat herleiten.

Theorem 2.11 *Der Boyer-Moore Algorithmus benötigt maximal $3(n + m)$ Vergleiche 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 $3n - o(n)$ Vergleiche 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. Folgendes Beispiel zeigt, dass der Boyer-Moore-Algorithmus beim Auffinden aller Vorkommen von s in t wieder quadratischen Zeitbedarf bekommen kann (im Gegensatz zum KMP-Algorithmus, der immer seine Laufzeit von $2n + m$ beibehält).

Suche a^m in a^n



n-m Vorkommen von a^m

$\Rightarrow$ Laufzeit $\Theta(n * m)$

Abbildung 2.30: Skizze: Anzahl Vergleiche bei Boyer-Moore bei Suche a^m in a^n

Diese worst-case Laufzeit lässt sich jedoch vermeiden, wenn man einen Trick anwendet. Nach einem erfolgreichen Vergleich verschieben wir das Muster jetzt wie beim KMP-Algorithmus. Da ja alle Vergleiche erfolgreich waren, gibt es ja nun keinen Präfix des Suchwortes, das noch unverglichen wäre. Dann merken wir uns (ähnlich wie beim KMP-Algorithmus),bis wohin wir beim Vergleichen von rechts nach links im Folgenden noch vergleichen müssen, da wir im Präfix des Suchwortes, das einen Rand darstellt, nach dem erfolgreichen Vergleich sicher sein können, dass dort eine Übereinstimmung herrscht.

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 Such-

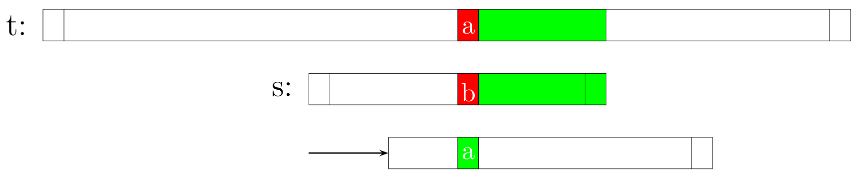


Abbildung 2.31: Skizze: Bad-Character-Rule

wort s so, dass das rechteste Vorkommen von t_{i+j} in s genau unter t_{i+j} zu liegen kommt. Falls wir die Zeichenreihe s dazu nach links verschieben 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.

Diese Regel macht insbesondere bei kleinen Alphabeten Probleme, wenn beim vorigen Vergleich 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 $s_0 \cdots s_{j-1}$ (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.

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 Der Algorithmus von Karp und Rabin

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.

2.5.1 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 Dezimalzahl zuordnet.

$$V:\ \{0,1\}^* \longrightarrow \mathbb{N}_0:\ V(s) = \sum_{i=0}^{|s|-1} 2^i \cdot s_i$$

Für $\Sigma = \{0, \ldots, k-1\}$ gilt dementsprechend:

$$V_k:\ [0:k-1]^* \longrightarrow \mathbb{N}_0:\ V_k(s) = \sum_{i=0}^{|s|-1} k^i \cdot s_i$$

Bsp.: $V_2(0011) = 12$; $V_2(1100) = 3$;

Fakt: 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 Dezimalzahlen miteinander verglichen werden. Dabei wird $V(s)$ mit jeder Dezimalzahl $V(t_{i,m})$ 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 Idee ist im folgenden einfachen Algorithmus wiedergegeben:

```
BOOL NAIV3 (char s[], int m, char t[], int n)
{
    for (i = 0; i ≤ n − m; i++) {
        if (V(s) == V(t_i, m)) return TRUE;
    }
}
```

Abbildung 2.32: Algorithmus: Naive numerische Textsuche

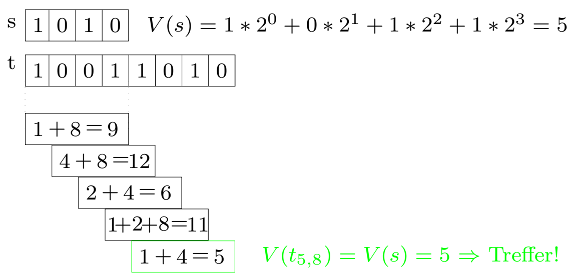


Abbildung 2.33: Beispiel: Numerische Suche von 1010 in 100110101

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 $\Theta(n * m)$. 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:

$$\begin{aligned}
V(t_{i+1,m}) &= \sum_{j=0}^{m-1} t_{i+1+j} \cdot 2^j \\
&= \sum_{j=1}^{m} t_{i+1+j-1} \cdot 2^{j-1} \\
&= \frac{1}{2} \sum_{j=1}^{m} t_{i+j} \cdot 2^j \\
&= \frac{1}{2} \left(\sum_{j=0}^{m-1} t_{i+j} \cdot 2^j - t_{i+0} \cdot 2^0 + t_{i+m} \cdot 2^m \right) \\
&= \frac{1}{2} * (V(t_{i,m}) - t_i + 2^m t_{i+m})
\end{aligned}$$

$V(t_{i+1,m})$ lässt sich somit mit dem Vorgänger $V(t,m)$ sehr einfach und effizient berechnen, nämlich in konstanter Zeit.

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.

2.5.2 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 folgendes Beispiel).

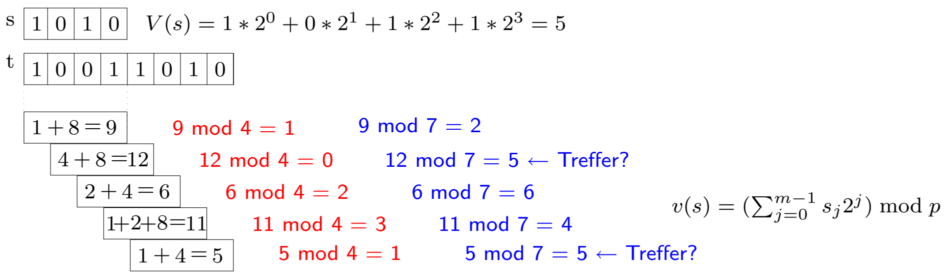


Abbildung 2.34: Beispiel: Numerische Suche von 1010 in 10011010 modulo 7

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, wo hingegen 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 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 auf der nächsten Seite angegeben ist. Den Wert $V(s) \bmod p$ nennt man auch einen *Fingerabdruck* (engl. *fingerprint*) des Wortes s.

2.5.3 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 $\pi(k) := |P \cap [1:k]|$ 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.

bool Karp-Rabin (char t[],int n, char s[]s, int m)
{
  int p; // hinreichend große Primzahl, die nicht zu groß ist ;-)
  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++)
  {
    if (($vs == vt$) && ($s == t_{i,m}$)) // Zusätzlicher Test auf Gleichheit
      return TRUE;
    $vt = \frac{1}{2}(vt - t_i + (2^m \bmod p) - t_{i+m}) \bmod p$;
  }
}

Abbildung 2.35: Algorithmus: Die Methode von Karp und Rabin

Theorem 2.12 *Für alle $k \in \mathbb{N}$ gilt:*

$$\frac{k}{\ln(k)} \leq \pi(k) \leq 1.26 \frac{k}{\ln(k)}.$$

Theorem 2.13 *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 2.14 *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 Vielfachheit $m_1, \ldots, m_\ell$. Nach Annahme ist $\ell > \pi(k)$. Dann gilt mit Hilfe des vorherigen Satzes:

$$x = \prod_{i=1}^{\ell} p_i^{m_i} \geq \prod_{i=1}^{\ell} p_i \geq \prod_{\substack{p \in \mathbb{P} \\ p \leq \ell}} p \geq 2^\ell > 2^{\pi(k)}.$$

Dies liefert den gewünschten Widerspruch. ■

Theorem 2.15 *Sei $s, t \in \{0,1\}^*$ mit $|s| = m$, $|t| = n$, mit $n*m \geq 29$. Sei $P \in \mathbb{N}$ und $p \in [1:P] \cap \mathbb{P}$ eine zufällig gewählte Primzahl, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers bei Karp-Rabin von s in t*

$$\mathrm{Ws}(\textit{irrtümlicher Treffer}) \leq \frac{\pi(n*m)}{\pi(P)}.$$

Damit folgt für die Wahl von P unmittelbar, dass $P > n*m$ 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 2.14 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 $\pi(n * m)$ Primfaktoren ausgewählt worden sein. Damit gilt

$$\text{Ws(irrtümlicher Treffer)} \leq \frac{\pi(n * m)}{\pi(P)}.$$

■

Zum Abschluss zwei Lemmas, die verschiedene Wahlen von P begründen.

Lemma 2.16 *Wählt man $P = m * n^2$, dann ist die Wahrscheinlichkeit eines irrtümlichen Treffers begrenzt durch $3/n$.*

Beweis: Nach dem vorherigen Satz gilt, dass die Wahrscheinlichkeit eines irrtümlicher Treffers durch $\frac{\pi(n*m)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.12:

$$\frac{\pi(n * m)}{\pi(n^2m)} \leq 1.26 \cdot \frac{nm \ln(n^2m)}{n^2m \ln(nm)} \leq \frac{1.26}{n} \cdot \frac{2\ln(nm)}{\ln(nm)} \leq \frac{2.52}{n}.$$

■

Ein Beispiel hierfür mit $n = 4000$ und $m = 250$. Dann ist $P = n * m^2 < 2^{32}$. Somit erhält man mit einem Fingerabdruck, der sich mit 4 Byte darstellen lässt, eine Fehlerwahrscheinlichkeit von unter 0,1%.

Lemma 2.17 *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 $\frac{\pi(n*m)}{\pi(P)}$ beschränkt ist. Damit folgt mit dem Satz 2.12:

$$\frac{\pi(n * m)}{\pi(nm^2)} \leq 1.26 \cdot \frac{nm \ln(nm^2)}{nm^2 \ln(nm)} \leq \frac{1.26}{m} \cdot \frac{2\ln(nm)}{\ln(nm)} \leq \frac{2.52}{m}.$$

■

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.

2.6 Suffix-Tries und Suffix-Bäume

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.

2.6.1 Suffix-Tries

Ein *Trie* ist eine Datenstruktur, in welcher eine Menge von Wörtern abgespeichert werden kann. Tries haben wir eigentlich schon kennen gelernt. Der Suchwort-Baum des Aho-Corasick-Algorithmus ist nichts anderes als ein Trie für die Menge der Suchwörter.

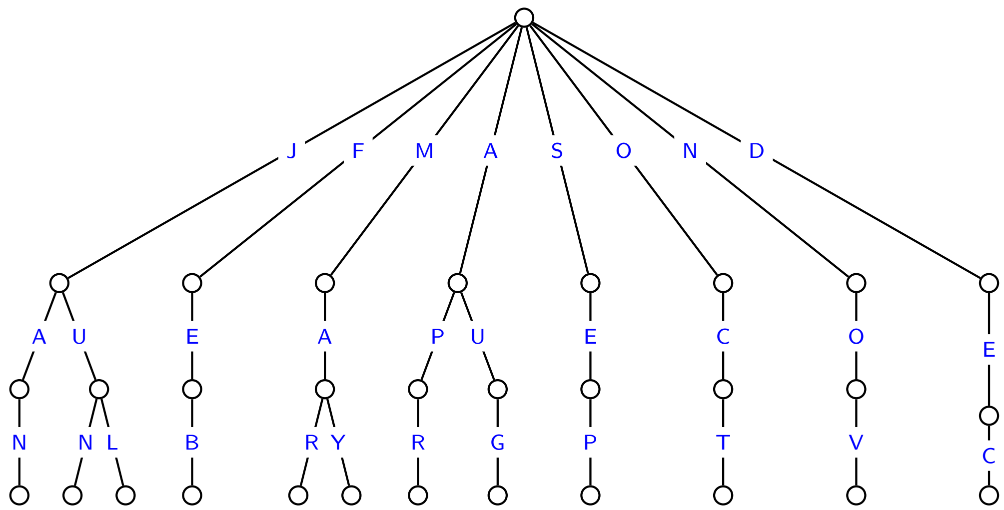


Abbildung 2.36: Beispiel: Trie für die dreibuchstabigen Monatsnamensabk.

Definition 2.18 *Ein* Trie *für ein gegebenes $M \subseteq \Sigma^*$ ist ein Baum mit Kantenmarkierungen, wobei alle Kantenbeschriftungen von Pfaden von der Wurzel zu einem Blatt gerade die Menge M ergeben und es von jedem Knoten aus für ein Zeichen $a \in \Sigma$ maximal eine ausgehende Kante mit diesem Zeichen als Markierung geben darf.*

Vorteil: Schnelles Suchen mit der Komplexität $O(m)$.

Definition 2.19 *Ein* Suffix-Trie *für ein Wort $t \in \Sigma^*$ ist ein Trie für alle Suffixe von t, d.h. $M = \{t_i \cdots t_n \mid i \in [1 : n+1]\}$ mit $t = t_1 \cdots t_n$ $(t_{n+1} \cdots t_n = \varepsilon)$.*

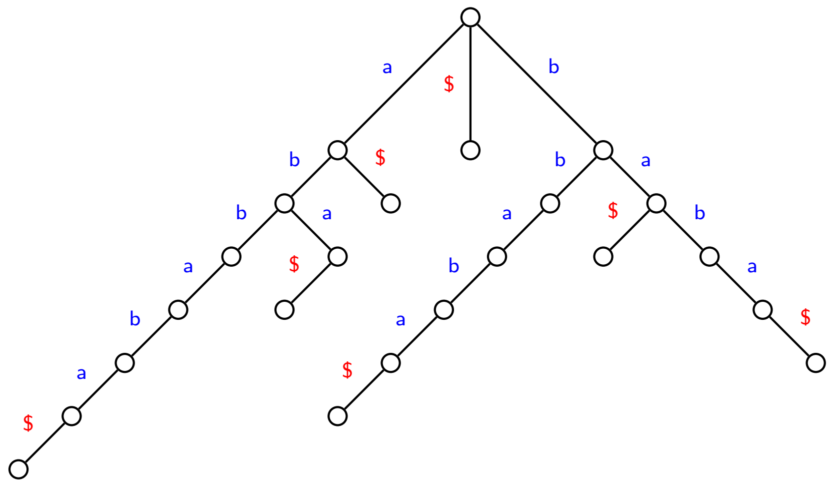


Abbildung 2.37: Beispiel: $t = abbaba\$$

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.

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 folgende einfache Algorithmus beschreibt genau dieses Verfahren.

Im Folgenden zählen wir als Elementaroperationen die Anzahl der besuchten und modifizierten Knoten der zugrunde liegenden Bäume.

Laufzeit: Da 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:

$$\sum_{i=1}^{n}(n-i+1) \;=\; \sum_{i=1}^{n} i \;=\; O(n^2).$$

```
BuildSuffixTrie (char t[], int n)
{
    tree T;
    for (i = 0; i ≤ n; i++)
        insert(T, t_i ··· t_n)
}
```

Abbildung 2.38: Algorithmus: Simple Methode zum Aufbau eines Suffix-Tries

Fakt: w ist 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.

2.6.2 Ukkonens Online-Algorithmus für Suffix-Tries

Sei $t \in \Sigma^n$ und sei T^i der Suffix-Trie für $t_1..t_i$. 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 $t_1 t_0 = \varepsilon$ ist und somit T^0 der Baum ist, der aus nur einem Knoten (der Wurzel) besteht.

Auf der nächsten Seite ist 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.

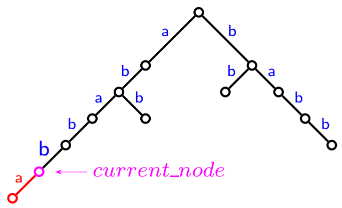
Das darauf folgende Bild zeigt den kompletten Suffix-Trie T^7 mit allen Suffix-Links (auch die Suffix-Links zur Konstruktion des T^6 sind eingezeichnet).

Sei $w \in \Sigma^*$ ein Teilwort von t. Den Knoten, der über w in einem Suffix-Trie erreichbar ist, bezeichnen wir mit $\overline{w}$. Sei $\overline{aw}$ der Knoten, der über aw von der Wurzel aus erreichbar ist, wobei $a \in \Sigma$ und $w \in \Sigma^*$. Dann zeigt der *Suffix-Link* von $\overline{aw}$ auf $\overline{w}$.

Damit auch die Wurzel einen Suffix-Link besitzt, ergänzen wir den Suffix-Trie um eine virtuelle Wurzel $\perp$ und setzen $\mathit{suffix\_link}(\mathit{root}) = \perp$. 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.

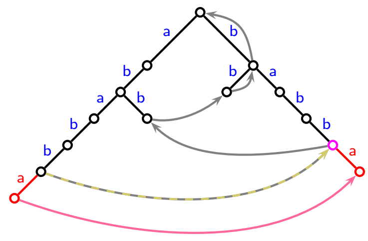
Beispiel: Aufbau eines Suffix-Tries

1. Schritt



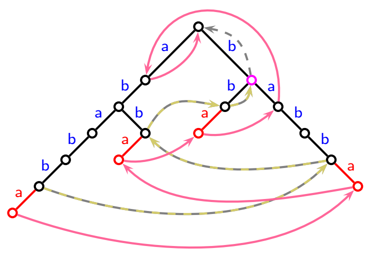
Der Trie für $t = ababb$ ist bereits konstruiert. An den Knoten, der am Ende des Pfades steht, dessen Labels das bisher längste Suffix ergibt, muss nun zuerst das neue Zeichen a angehängt werden. Anschließend muss an jedes kürzere Präfix ein a angehängt werden.

2. Schritt



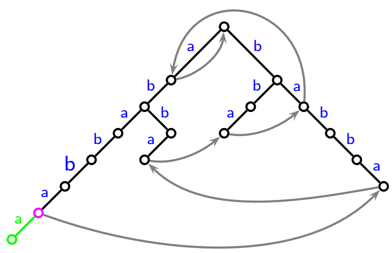
Um nun den Knoten schnell zu finden, der das nächst kürzere Suffix repräsentiert, sind im Trie so genannte *Suffix-Links* vorhanden. Diese zeigen immer auf den Knoten, der das nächst kürzere Suffix darstellt. Folge nun dem Suffix-Link des aktuellen Knotens und füge an dessen Endknoten wiederum die neue 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 2.39: Beispiel: $t = ababbaa$; Konstruktion von T^6 aus T^5

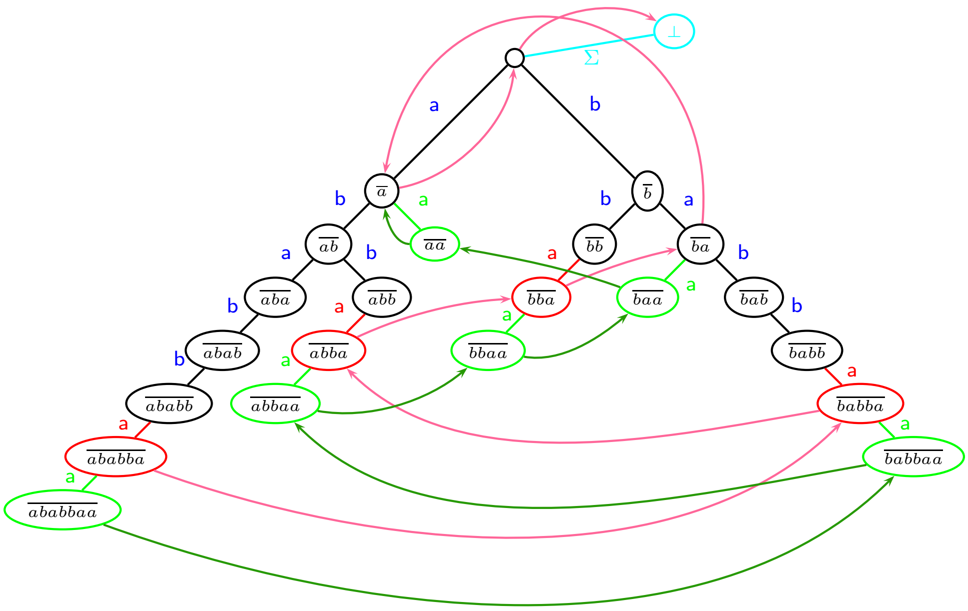


Abbildung 2.40: Beispiel: Suffix-Trie für $ababbaa$ mit Suffix-Links

2.6.3 Laufzeitanalyse für die Konstruktion von T^n

Naive Methode: Bei der naiven Methode muss zur Verlängerung eines Suffixes der 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,

$$\sum_{i=1}^{n} \underbrace{\sum_{j=1}^{i} O(i-j+1)}_{\text{Zeit für } T^{i-1} \to T^i} = O\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) = O\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) = O\left(\sum_{i=1}^{n} i^2\right) = O(n^3).$$

Laufzeit mit Suffix-Links: Durch die Suffix-Links benötigt das Verlängern des Suffixes $t_j \cdots t_{i-1}$ um t_i nur noch $O(1)$ Operationen.

$$\sum_{i=1}^{n}\sum_{j=1}^{i} O(1) = \sum_{i=1}^{n} O(i) = O(n^2)$$

2.6.4 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 $t = a^n b^n$:

```
BuildSuffixTrie (char t[], int n)
{
    T = ({root, ⊥}, {⊥ -x-> root : x ∈ Σ});
    suffix_link(root) = ⊥;
    longest_suffix = prev_node = root;
    for (i = 1; i ≤ n; i++)
    {
        curr_node = longest_suffix;
        while (curr_node -t_i-> some_node does not exist)
        {
            Add curr_node -t_i-> new_node to T;
            if (curr_node == longest_suffix)
                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;
    }
}
```

Abbildung 2.41: Algorithmus: Konstruktion von Suffix-Tries mit Hilfe von Suffix-Links

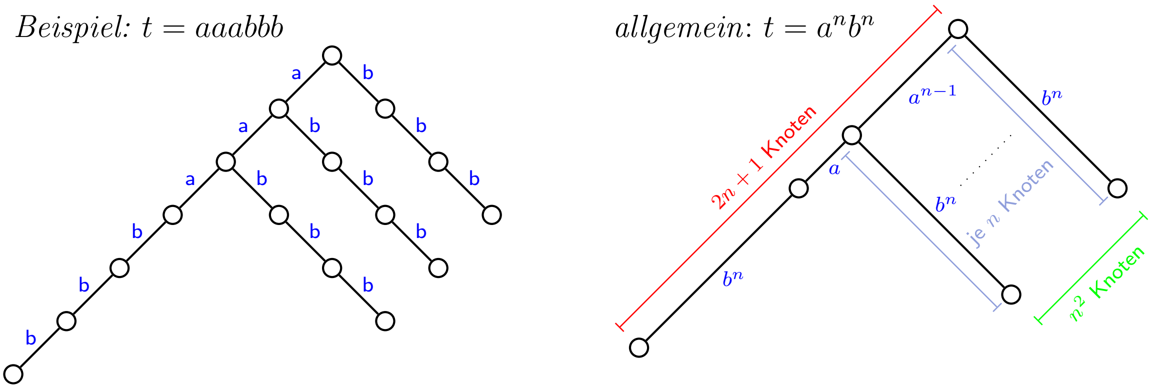


Abbildung 2.42: Beispiel: Potentielle Größe von Suffix-Tries (für $a^n b^n$)

Der Suffix-Trie für ein Wort t der Form $t = a^n b^n$ hat damit insgesamt

$$(2n + 1) + n^2 = (n + 1)^2 = O(n^2)$$

viele Knoten, was optimiert werden muss.

Die Idee hierfür ist, 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 kompaktifizierten Tries werden *Patricia-Tries* genannt (für *Practical Algorithm To Retrieve Information Coded In Alphanumeric*), siehe auch folgendes Beispiel.

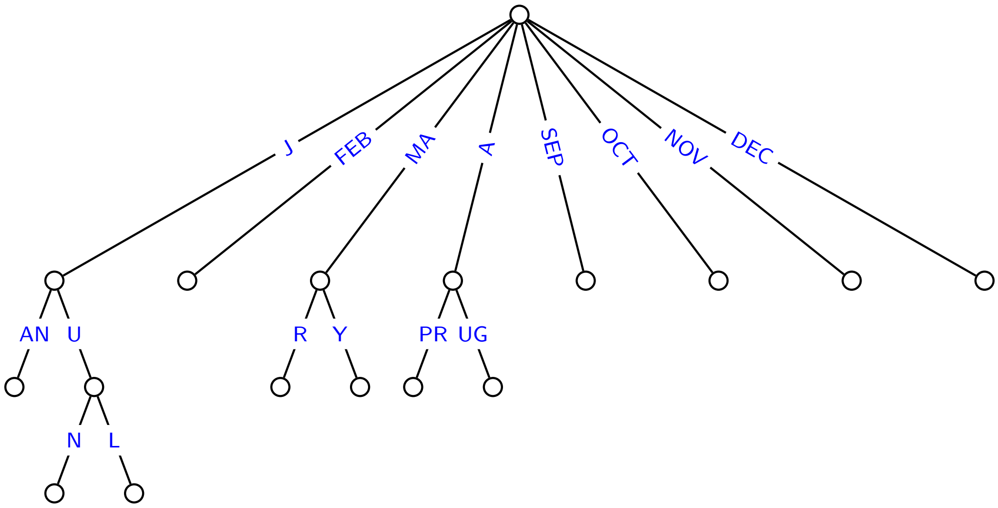


Abbildung 2.43: Beispiel: Patricia-Trie für die dreibuchstabigen Monatsnamensabk.

2.6.5 Suffix-Bäume

So kompaktifizierte Suffix-Tries werden *Suffix-Bäume* (engl. *suffix-trees*) genannt. 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.

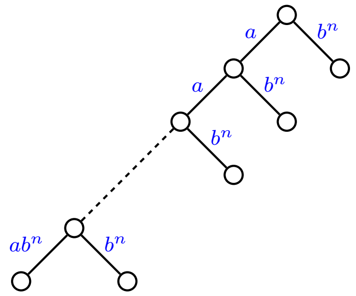


Abbildung 2.44: Beispiel: Kompaktifizierter Trie für $t = a^n b^n$

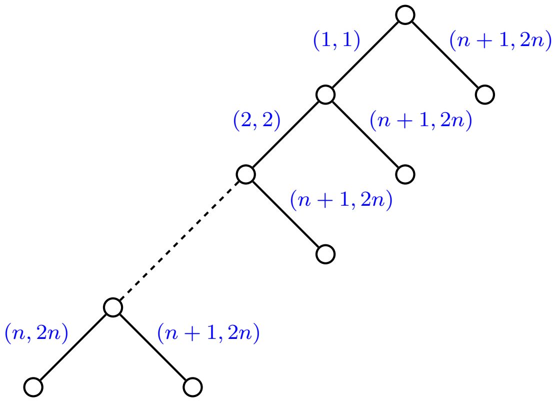


Abbildung 2.45: Beispiel: Echter Suffix-Baum für$t = a^n b^n$

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 $O(n^2)$).

2.6.6 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 $\hat{T}^{i-1}$ aufbauen. Diese Konstruktion kann man aus der Konstruktion des Suffix-Tries T^i aus dem Suffix-Trie T^{i-1} erhalten.

Dazu folgende Festlegungen:

- Sei $t \in \Sigma^*$, wobei $t = t_1 \cdots t_{i-1}$.
- Sei $s_j = t_j \cdots t_{i-1}$ und sei $\overline{s_j}$ der zugehörige Knoten im Suffix-Trie, 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 $j < l$ im Suffix-Trie T^{i-1} eine neue Kante für t_i an den Knoten $\overline{s_j}$ angehängt werden muss ($\rightarrow \overline{s_l}$ heißt Endknoten).
- Sei k der größte Index, so dass für alle $j < k$ $\overline{s_j}$ ein Blatt ist ($\rightarrow \overline{s_k}$ heißt aktiver Knoten).

Da $\overline{s_1}$ immer ein Blatt ist, gilt $k > 1$, und da $\overline{s_{i+1}} = \perp$ immer die Kante mit dem Label t_i besitzt, gilt $l \leq i + 1$.

Sei w ein Teilwort von t, dann heißt ein Knoten $\overline{w}$ des Suffix-Tries *explizit*, wenn es auch im Suffix-Baum einen Knoten gibt, der über die Zeichenreihe w erreichbar ist. Andernfalls heißt er implizit.

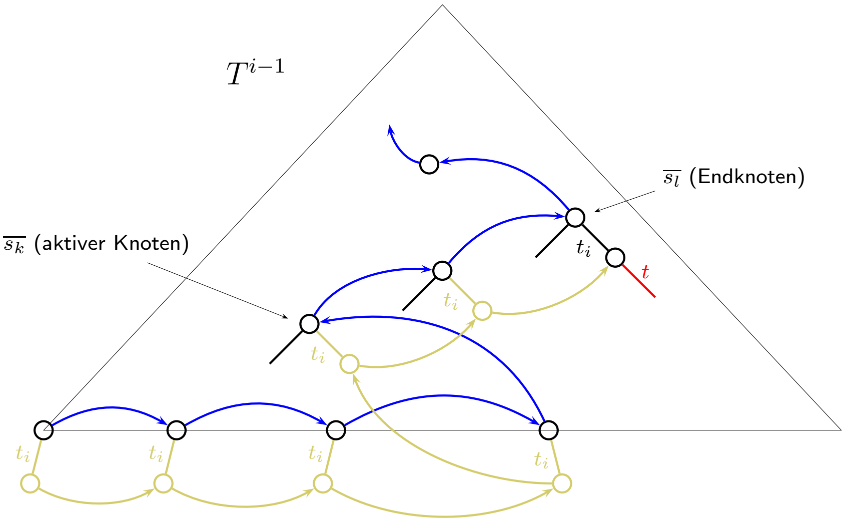


Abbildung 2.46: Skizze: Konstruktion eines Suffix-Tries

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 statt der Teilwörter als Kantenlabels so genannte (offene) Referenzen benutzt.

Definition 2.20 *Sei w ein Teilwort von $t_1 \cdots t_n$ mit $w = uv = t_j \cdots t_{k-1} t_k \cdots t_p$, wobei $u = t_j \cdots t_{k-1}$ und $v = t_k \cdots t_p$. Ist $s = \overline{u}$ im Suffix-Baum realisiert, dann heißt $(s,(k,p))$ eine* Referenz *für $\overline{w}$ (im Suffix-Trie).*

Beispiele für Referenzen in $t = ababbaa$ (siehe auch Abbildung 2.48):

1. $w = \genfrac{}{}{0pt}{}{babb}{2345} \mathrel{\hat{=}} (\overline{b},(3,5)) \mathrel{\hat{=}} (\overline{ba},(4,5))$.

2. $w = \genfrac{}{}{0pt}{}{bbaa}{4567} \mathrel{\hat{=}} (\overline{b},(5,7)) \mathrel{\hat{=}} (\overline{bb},(6,7))$.

Definition 2.21 *Eine Referenz (s,(k,p)) heißt* kanonisch, *wenn $p - k$ minimal ist.*

Für Blätter des Suffix-Baumes werden alle Referenzen *offen* angegeben, d.h. es wird $(s,(k,\infty))$ statt $(s,(k,n))$ in $\hat{T}^n$ verwendet. Dadurch spart man sich dann die Arbeit, die Kantenlabels der Kanten, die zu Blättern führen, während der Konstruktion des Suffixbaumes zu verlängern.

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. Desweiteren ist noch von Interesse, ob, nachdem der Endknoten erreicht wurde, jeder weitere Knoten auch eine Kante mit dem Label t_i besitzt.

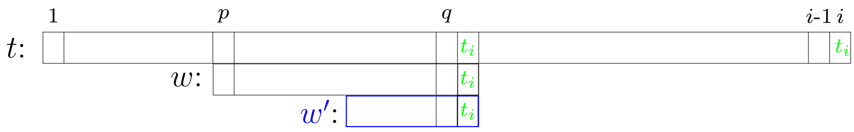


Abbildung 2.47: Skizze: Endknoten beendet Erweiterung von T^{i-1}

Dazu betrachten wir obige Skizze. 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 ($\rightarrow$ blauer String w' in der Skizze), muss auch dieser Knoten ein innerer sein, da seine Kantenlabels ebenfalls nicht nur Suffix 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 t_i Kante 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 t_i Kante 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 t_i Kante besitzen muss.

Somit haben wir das folgende für die Konstruktion von Suffix-Bäumen wichtige Resultat bewiesen:

Lemma 2.22 *Ist (s,(k,i-1)) der Endknoten von $\hat{T}^{i-1}$ (Suffix-Baum $t_1 \cdots t_{i-1}$), dann ist (s,(k,i)) der aktive Knoten von $\hat{T}^i$.*

Die Beispiele zu den folgenden Prozeduren beziehen sich alle auf den unten abgebildeten Suffix-Baum für das Wort $t = ababbaa$.

Ukkonens Algorithmus für Suffix-Bäume ist in der folgenden Abbildung angegeben. In der for-Schleife des Algorithmus wird jeweils $\hat{T}^i$ aus $\hat{T}^{i-1}$ mit Hilfe der Prozedur

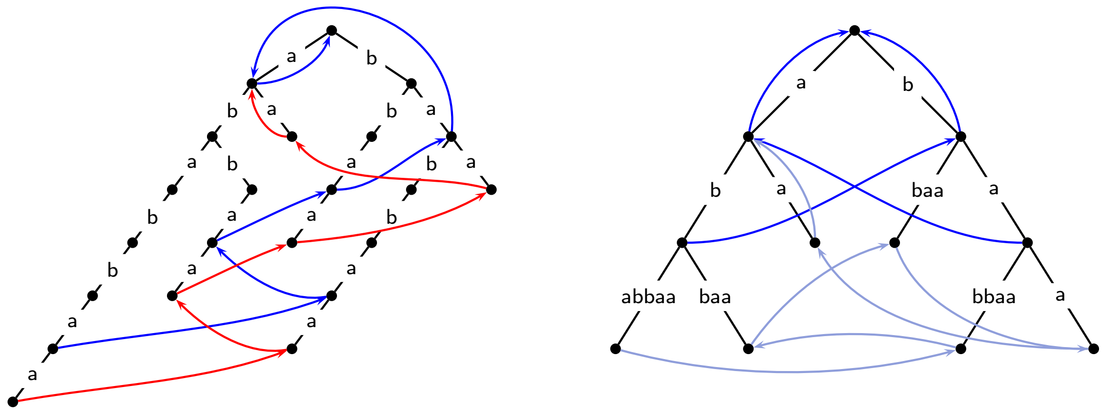


Abbildung 2.48: Beispiel: Suffix-Trie und Suffix-Baum für $t = ababbaa$

Update konstruiert. Dabei ist jeweils $(s,(k,i-1))$ der aktive Knoten. Zu Beginn für $i = 1$ ist dies für $(\overline{\varepsilon},(1,0))$ klar, da dies der einzige Knoten ist (außer dem Virtuellen). Wir bemerken hier, dass die Wurzel eines Baumes für uns per Definition kein Blatt ist, obwohl sie hier keine Kinder besitzt. Zum Ende liefert die Prozedur Update, die

BuildSuffixTree (char $t[]$, int n)
{
    $T = (\{root, \perp\}, \{\perp \xrightarrow{x} root : x \in \Sigma\})$;
    $suffix\_link(root) = \perp$;
    $s = root$;
    $k = 1$;
    for $(i = 1;\ i \leq n;\ i{+}{+})$
        // Constructing T^i from T^{i-1}
        // $(s,(k,i-1))$ is active point in T^{i-1}
        $(s,k) = \text{Update}(s,(k,i-1),i)$;
        // Now $(s,(k,i-1))$ is endpoint of T^{i-1}
}

Abbildung 2.49: Algorithmus: Ukkonens Online-Algorithmus

aus T^{i-1} den Suffix-Baum T^i konstruiert, den Endknoten von T^{i-1} zurück, nämlich $(s,(k,i-1))$. Nach Lemma 2.22 ist dann der aktive Knoten von 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 dem aktiven Knoten von T^i, der jetzt ja T^{i-1} ist, wieder mit der korrekten Referenz $(s,(k,i-1))$.

Bevor wir die Prozedur Update erläutern, geben wir noch die Prozedur Canonize an, die aus einer übergegebenen Referenz eine kanonische Referenz konstruiert. Wir wollen ja eigentlich immer nur mit kanonischen Referenzen arbeiten.

CANONIZE (node s, ref (k, p))

{
  while $(|t_k \cdots t_p| > 0)$
  {
    let $e = s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;
    if $(|w| > |t_k \cdots t_p|)$ **break**;
    $k = k + |w|$;
    $s = s'$
  }
  return (s, k);
}

Abbildung 2.50: Algorithmus: Die Prozedur Canonize

Der Prozedur Canonize wird eine gegebene Referenz $(\overline{s}, (k, p))$ übergeben. Die Prozedur durchläuft dann den Suffix-Baum ab dem Knoten $\overline{s}$ entlang der Zeichenreihe $t_k \cdots t_p$. 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 noch einmal am Beispiel für die Referenz $(\overline{ab}, (6, 7))$ illustriert.

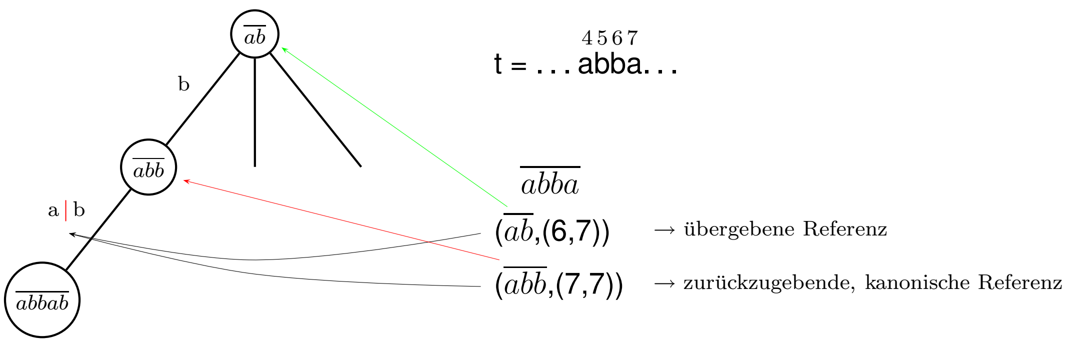


Abbildung 2.51: Beispiel: Erstellung kanonischer Referenzen mittels Canonize

In der Prozedur Update 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 der Prozedur TestAndSplit, die den tatsächlichen Knoten zurückgibt, an welchen die neue t_i Kante angehängt werden muss, falls sie noch nicht vorhanden ist. 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 t_i Kante, falls noch nicht vorhanden, anzuhängen ist.

```
UPDATE (node s, ref (k,p), int i)
{
    // (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 )
    {
        let m be a new node and add r --(i,∞)--> m;
        if (old_r ≠ root)
            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 ≠ root)
        suffix_link(old_r) = s;
    return (s, k);
}
```

Abbildung 2.52: Algorithmus: Die Prozedur Update

Die Prozedur TestAndSplit geht dabei so vor, dass zunächst überprüft wird, ob die betreffende Kante bereits vorhanden ist oder nicht. Falls dies der Fall ist, wird der Knoten, von welchem diese Kante ausgeht, ausgegeben. Ist die Kante jedoch noch nicht vorhanden, aber der Knoten, von welchem sie ausgehen sollte, existiert bereits im Suffix-Baum, 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 Referenz auf einen expliziten oder impliziten Knoten zeigt ($|t_k \cdots t_p| == 0$). Ist der Knoten explizit, muss in jedem Falle kein neuer 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 ja nichts zu tun. Ansonsten muss diese Kante mit dem langen Label aufgebrochen werden, wie dies im folgenden Beispiel illustriert ist. Dann wird der neue Knoten r, der in die lan-

TESTANDSPLIT (node s, ref (k,p), char x)

{
    if $(|t_k \cdots t_p| == 0)$
        if $(\exists\ s \xrightarrow{x\cdots})$ **return** (TRUE, s);
        else return (FALSE, s);
    else
    {
        let $e = s \xrightarrow{w} s'$ s.t. $w_1 = t_k$;
        if $(x == w_{|t_k \cdots t_p|+1})$ **return** (TRUE, s);
        else
        {
            split $e = s \xrightarrow{w} s'$ s.t.:
            $s \xrightarrow{w_1 \cdots w_{|t_k \cdots t_p|}} m \xrightarrow{w_{|t_k \cdots t_p|+1} \cdots w_{|w|}} s'$
            return (FALSE, m)
        }
    }
}

Abbildung 2.53: Algorithmus: Die Prozedur TestAndSplit

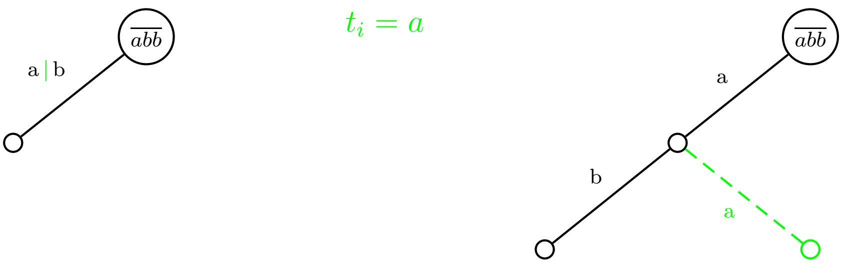


Abbildung 2.54: Beispiel: Vorgehensweise von TestAndSplit

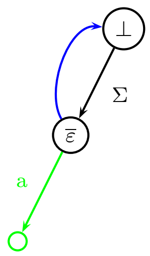
gen Kante eingefügt wurde, von TestAndSplit zurückgegeben, damit die Prozedur Update daran die neue Kante mit Label t_i anhängen kann.

Zurück zur Beschreibung der Prozedur Update. 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, da wir vom aktiven Knoten starten, der ja der erste interne Knoten ist. 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.

Folgendes längeres *Beispiel* zeigt, wie nach und nach der Suffix-Baum für das bereits öfter verwendete Wort $t = ababbaa$ aufgebaut wird:

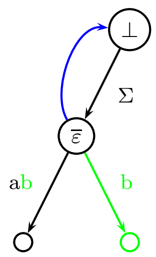
	1	2	3	4	5	6	7
t=	a	b	a	b	b	a	a

Buchstabe a wird in den noch leeren Tree eingefügt:



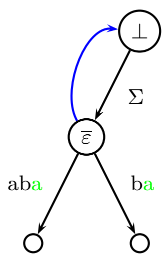
($\overline{\varepsilon}$,(1,0)) $i = 1; t_i = a$
$\rightsquigarrow$ TestAndSplit = (FALSE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(1,0))
$\downarrow$ Suffix-Link
($\perp$,(1,0))
$\downarrow$ Canonize
($\perp$,(1,0))

Buchstabe b wird in den $\hat{T}^1$ eingefügt:



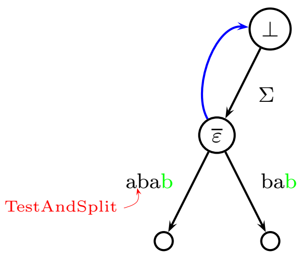
($\perp$,(1,1)) $i = 2; t_i = b$
$\downarrow$ Canonize
($\overline{\varepsilon}$,(2,1))
$\rightsquigarrow$ TestAndSplit = (FALSE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(2,1))
$\downarrow$ Suffix-Link
($\perp$,(2,1))
$\downarrow$ Canonize
($\perp$,(2,1))

Buchstabe a wird in den $\hat{T}^2$ eingefügt:



($\perp$,(2,2)) $i = 3; t_i = a$
$\downarrow$ Canonize
($\overline{\varepsilon}$,(3,2))
$\rightsquigarrow$ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(3,2))

Buchstabe b wird in den $\hat{T}^3$ eingefügt:



$(\overline{\varepsilon}$,(3,3)) $i = 4$; $t_i = b$

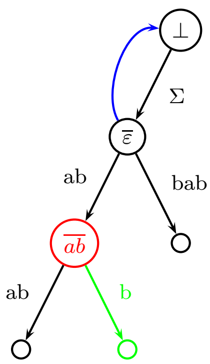
↓ Canonize

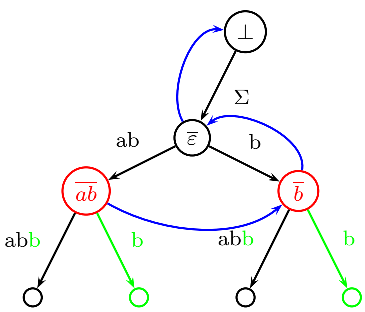
$(\overline{\varepsilon}$,(3,3))

⦚ TestAndSplit = (TRUE,$\overline{\varepsilon}$)

$(\overline{\varepsilon}$,(3,3))

Buchstabe b wird in den $\hat{T}^4$ eingefügt:





$(\overline{\varepsilon}$,(3,4)) $i = 5$; $t_i = b$

↓ Canonize

$(\overline{\varepsilon}$,(3,4))

⦚ TestAndSplit = (FALSE,$\overline{ab}$)

$(\overline{\varepsilon}$,(3,4))

↓ Suffix-Link

$(\perp$,(3,4))

↓ Canonize

$(\overline{\varepsilon}$,(4,4))

⦚ TestAndSplit = (FALSE,$\overline{b}$)

$(\overline{\varepsilon}$,(4,4))

↓ Suffix-Link

$(\perp$,(4,4))

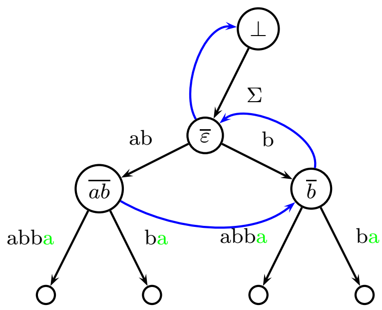
↓ Canonize

$(\overline{\varepsilon}$,(5,4))

⦚ TestAndSplit = (TRUE,$\overline{\varepsilon}$)

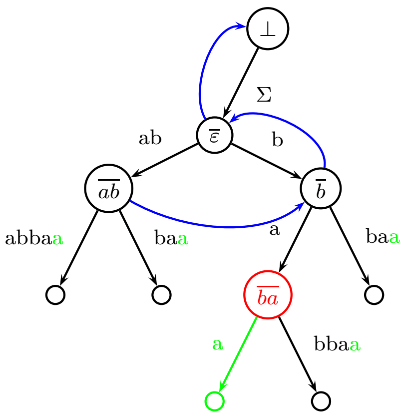
$(\overline{\varepsilon}$,(5,4))

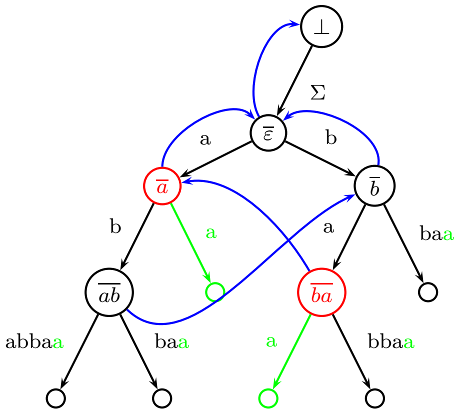
Buchstabe a wird in den $\hat{T}^5$ eingefügt:



($\overline{\varepsilon}$,(5,5)) $i = 6; t_i = a$
↓ Canonize
($\overline{b}$,(6,5))
⇝ TestAndSplit = (TRUE,$\overline{b}$)
($\overline{b}$,(6,5))

Buchstabe a wird in den $\hat{T}^6$ eingefügt:





($\overline{b}$,(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
($\perp$,(6,6))
↓ Canonize
($\overline{\varepsilon}$,(7,6))
⇝ TestAndSplit = (TRUE,$\overline{\varepsilon}$)
($\overline{\varepsilon}$,(7,6))

Jetzt ist der Suffix-Baum für $t = ababbaa$ vollständig aufgebaut.

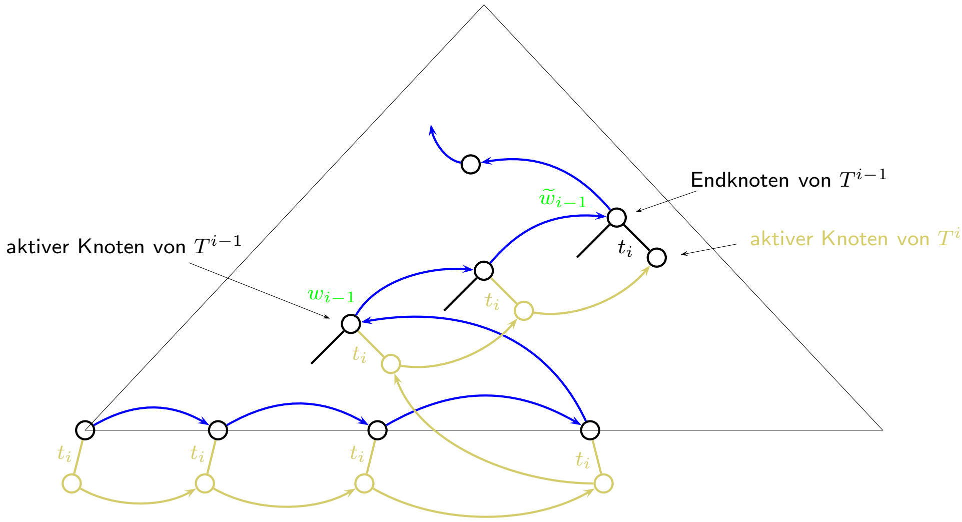


Abbildung 2.55: Skizze: Erweiterung von T^{i-1} auf T^i

2.6.7 Laufzeitanalyse

Zum Schluss kommen wir noch zur Laufzeitanalyse des Online-Algorithmus von Ukkonen für Suffix-Bäume. Wir teilen die Zeitanalyse in zwei Teile auf. Zunächst analysieren wir den Zeitbedarf für alle Ausführungen der while-Schleife in der Prozedur Canonize. Ein Durchlauf der while-Schleife kann offensichtlich in Zeit $O(1)$ ausgeführt werden. Wir stellen fest, dass nur hier der Wert von k verändert wird, und zwar wird er immer nur vergrößert. Da k zu Beginn 1 ist und nie größer als n werden kann, wird die while-Schleife der Prozedur Canonize maximal n-mal durchlaufen. Somit ist auch der Zeitbedarf höchstens $O(n)$.

Nun betrachten wir alle übrigen Kosten. Die verbleibenden Kosten für einen Aufruf der Prozedur Canonize sind jetzt noch $O(1)$. Ebenso benötigt jeder Aufruf der Prozedur TestAndSplit Zeit $O(1)$. Damit sind die verbleibenden Kosten durch die Anzahl der betrachteten Knoten des Suffix-Baumes beschränkt.

Sei w_{i-1} die Zeichenreihe, um den aktiven Knoten von T^{i-1} zu erreichen und $\widetilde{w}_{i-1}$ die Zeichenreihe, um den Endknoten von T^{i-1} zu erreichen. Dann ist $\overline{w_i}$ der aktive Knoten von T^i und $\overline{\widetilde{w}_i}$ der Endknoten von T^i. Betrachte hierzu auch die folgende Skizze. Wie viele Knoten werden nun bei der Konstruktion von $\hat{T}^i$ aus $\hat{T}^{i-1}$ besucht? Ist $\overline{w_i}$ der aktive Knoten von T^i, dann war der Endknoten von T^{i-1} gerade $\overline{\widetilde{w}_i}$, wobei $w_i = \tilde{w}_{i-1} \cdot t_i$. Die Anzahl der Durchläufe einer *while-Schleife* entspricht $(|w_{i-1}| - |\widetilde{w}_{i-1}| + 1)$. Weiterhin ist

$$|w_i| - |\widetilde{w}_{i-1}| + 1 = (|w_{i-1}| - (|w_i| - 1) + 1) = |w_{i-1}| - |w_i| + 2.$$

Wie viele Knoten werden für alle Abläufe der *while-Schleife* in Update getestet?

$$\begin{aligned}
&\sum_{i=1}^{n}(|w_{i-1}| - |w_i| + 2)\\
&= |w_0| - |w_1| + |w_1| - |w_2| + |w_2| - |w_3| + \cdots + |w_{n-1}| - |w_n| + \sum_{i=1}^{n} 2\\
&= |w_0| - |w_n| + \sum_{i=1}^{n} 2\\
&= 2n - |w_n|\\
&\leq 2n
\end{aligned}$$

Die Laufzeit für *while-Schleife* bei allen Updates entspricht $O(n)$. Die Laufzeit für *nicht-while-Schleife* ist ebenfalls insgesamt $O(n)$. Somit ist die Gesamtlaufzeit $O(n)$.

Theorem 2.23 *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.*

2.6.8 Problem: 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 die verschiedenen Methoden einmal an und bewerten sie nach Platz- und Zeitbedarf. Wir bewerten den Zeitbedarf danach, wie lange es dauert, bis wir für ein Zeichen aus Σ das entsprechende Kind gefunden haben. Für den Platzbedarf berechnen wir den Bedarf für den gesamten Suffix-Baum für einen Text der Länge m.

Realisierungsmöglichkeiten:

Felder Die Kinder eines Knotens lassen sich sehr einfach mit Hilfe eines Feldes der Größe $|\Sigma|$ darstellen.

- Platz: $O(m \cdot |\Sigma|)$.
  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: $O(m)$.

  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 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 m maximal m 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 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.

Hashfunktion Eine weitere Möglichkeit ist die Verwaltung der Kinder aller Knoten in einem einzigen großen Feld der Größe $O(m)$. 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 $\Theta(m \cdot |\Sigma|)$) auf ein kleines Intervall ab (hier Indizes aus $[1 : \ell]$ mit $\ell = \Theta(m)$). 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 Werrt erhalten. Also muss man beim Hashing mit diesen Kollisionen leben und diese geeignet auflösen. Für die Details wird auf andere Vorlesungen (wie etwa Informatik IV) verwiesen.

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. Somit lassen sich Kollisionen leicht feststellen und die üblichen Operationen zur Kollisionsauflösung anwenden.

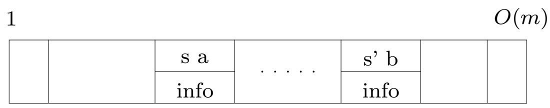


Abbildung 2.56: Skizze: Realisierung mittels eines Feldes und Hashing

- Platz: $O(m)$

  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.

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.

Paarweises Sequenzen Alignment

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. Dies wird an folgenden Beispielen illustriert:

Betrachte die Worte Monkey und Money:

$$\begin{array}{cccccc} & & & & \text{Insertion} & \\ \mathrm{M} & \mathrm{o} & \mathrm{n} & \mathrm{k} & \mathrm{e} & \mathrm{y} \\ \mathrm{M} & \mathrm{o} & \mathrm{n} & \_ & \mathrm{e} & \mathrm{y} \\ & & & & \text{Deletion} & \end{array}$$

Wie obiges Bild 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.

Ähnlich verhält es sich mit den Wörtern Money und Honey, wie folgendes Bild zeigt:

$$\begin{array}{cccccc} & \mathrm{M} & \mathrm{o} & \mathrm{n} & \mathrm{e} & \mathrm{y} \\ \text{Substitution} & & & & & \\ & \mathrm{H} & \mathrm{o} & \mathrm{n} & \mathrm{e} & \mathrm{y} \end{array}$$

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.

3.1 Distanz- und Ähnlichkeitsmaße

In diesem Abschnitt werden wir so genannte Distanzmaße einführen, die es uns überhaupt erst erlauben, ähnliche Sequenzen qualitativ und quantitativ zu beurteilen.

3.1.1 Edit-Distanz

Sei hier und im Folgenden Σ ein Alphabet. Weiter sei „$-$“ ein neues Zeichen, d.h. $- \notin \Sigma$. Dann bezeichne $\overline{\Sigma} := \Sigma \cup \{-\}$ das um $-$ erweiterte Alphabet von Σ. Wir werden $-$ oft auch als *Leerzeichen* bezeichnen.

Definition 3.1 *Eine* Edit-Operation *ist ein Paar*

$$(x, y) \in \overline{\Sigma} \times \overline{\Sigma} \setminus \{-, -\}.$$

Eine Edit-Operation (x, y) heißt

- Match*, wenn $x = y \in \Sigma$;*
- Substitution*, wenn $x \neq y \in \Sigma$;*
- Insertion*, wenn $x = -, y \in \Sigma$;*
- Deletion*, wenn $x \in \Sigma, y = -$.*

Wir bemerken hier, dass $(x, x) \in \Sigma \times \Sigma$ explizit als Edit-Operation zugelassen wird. In vielen Büchern wird dies nicht erlaubt. Wir werden im Folgenden sehen, dass ein solcher Match als Edit-Operation eigentlich eine neutrale oder NoOp-Operation ist und meist problemlos weggelassen werden kann. Als *Indel-Operation* bezeichnet man eine Edit-Operation, die entweder eine Insertion oder Deletion ist.

Definition 3.2 *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 $\exists\, i \in [1 : |a|] : \ (x, y \in \Sigma) \wedge (a_i = x) \wedge (b = a_1 \cdots a_{i-1} \cdot y \cdot a_{i+1} \cdots a_{|a|})$ (Substitution oder Match);*
- *oder wenn $\exists\, i \in [1 : |a|] : (a_i = x) \wedge (y = -) \wedge (b = a_1 \cdots a_{i-1} \cdot a_{i+1} \cdots a_{|a|})$ (Deletion);*
- *oder wenn $\exists\, j \in [1 : |b|] : (b_j = y) \wedge (x = -) \wedge (a = b_1 \cdots b_{j-1} \cdot b_{j+1} \cdots b_{|b|})$ (Insertion).*

Sei $s = ((x_1, y_1), \ldots, (x_m, y_m))$ eine 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 $a \stackrel{s}{\Rightarrow} b$.

Folgende Abbildungen zeigen zwei Beispiele von Transformationen einer Sequenz in eine andere mit Hilfe von Edit-Operationen. Dabei wird immer die obere Sequenz in die untere umgewandelt.

$$AGTGTAGTA \stackrel{s}{\Rightarrow} ACGTGTTT \text{ mit } s = ((G,-),(T,C),(A,G),(G,T),(A,T))$$
$$\text{oder mit } s = ((G,T),(A,G),(G,-),(A,T),(T,C)).$$

	A	G	T	G	T	A	G	T	A	D: Deletion
	A	_	C	G	T	G	T	T	T	I: Insertion
5 Edit-Op.		D	S			S	S		S	S: Substitution

Abbildung 3.1: 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 durchaus auf die Reihenfolge ankommt. Der Leser möge sich solche Beispiele überlegen.

Wir können dieselben Sequenzen auch mit Hilfe anderer Edit-Operationen ineinander transformieren.

$$AGTGTAGTA \stackrel{s'}{\Rightarrow} ACGTGTTT \text{ mit } s' = ((-,C),(A,-),(G,-),(A,T)).$$

	A	_	G	T	G	T	A	G	T	A	D: Deletion
	A	C	G	T	G	T	_	_	T	T	I: Insertion
4 Edit-Op.		I					D	D		S	S: Substitution

Abbildung 3.2: 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 (Kosten-)Funktion $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$. 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.

Definition 3.3 *Sei* $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ *eine (Kosten-)Funktion. Seien* $a, b \in \Sigma^*$ *und sei* $s = (s_1, \ldots, s_l)$ *eine Folge von Edit-Operationen mit* $a \xrightarrow{s_1} \cdots \xrightarrow{s_l} b$ *(kurz* $a \stackrel{s}{\Longrightarrow} b$*). Dann sind die* Kosten der Edit-Operationen s *definiert als*

$$w(s) := \sum_{j=1}^{l} w(s_j).$$

Die Edit-Distanz *von* $a, b \in \Sigma^*$ *ist definiert als*

$$d_w(a, b) := min\{w(s) \mid a \stackrel{s}{\Longrightarrow} b\}.$$

Zuerst einmal überlegen wir uns, was für eine Kostenfunktion w sinnvollerweise gelten soll:

$\boldsymbol{w(x, y) + w(y, z) \geq w(x, z)}$ **für** $\boldsymbol{x, y, z \in \Sigma}$**:** 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.

$\boldsymbol{w(x, -) + w(-, z) \geq w(x, z)}$ **für** $\boldsymbol{x, z \in \Sigma}$**:** Es ist wohl im Allgemeinen wahrscheinlicher, dass eine Mutation durch eine Substitution und nicht erst durch Deletion und eine anschließende Insertion (oder umgekehrt) zustande gekommen ist. Daher sollte die Kosten der Substitution billiger als die Kosten der entsprechenden Deletion und Insertion sein.

Diese Bedingungen sind beispielsweise erfüllt, wenn w eine Metrik ist.

Definition 3.4 *Eine Funktion* $w : M \times M \to \mathbb{R}_+$ *heißt* Metrik*, 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).*

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 3.5 *Ist $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine Metrik, dann ist auch $d_w : \overline{\Sigma}^* \times \overline{\Sigma}^* \to \mathbb{R}_+$ eine Metrik.*

Beweis: Seien $a, b \in \Sigma^*$. Dann gilt:

$$\begin{aligned} 0 &= d_w(a,b) \\ &= \min\left\{ w(s) \ : \ a \stackrel{s}{\Rightarrow} b \right\} \\ &= \min\left\{ \sum_{i=1}^{r} w(s_i) \ : \ a \stackrel{s}{\Longrightarrow} b \text{ mit } s = (s_1, \ldots, s_r) \right\}. \end{aligned}$$

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:

$$\begin{aligned} d_w(a,b) &= \min\left\{ w(s) \ : \ a \stackrel{s}{\Rightarrow} b \text{ mit } s = (s_1, \ldots, s_r) \right\} \\ &= \min\left\{ \sum_{i=1}^{r} w(s_i) \ : \ a = a_0 \stackrel{s_1}{\to} a_1 \cdots a_{r-1} \stackrel{s_r}{\to} a_r = b \right\} \\ &= \min\left\{ \sum_{i=1}^{r} w(s_i) \ : \ b = a_r \stackrel{\tilde{s}_r}{\to} a_{r-1} \cdots a_1 \stackrel{\tilde{s}_1}{\to} a_0 = a \right\} \\ &= \min\left\{ w(\tilde{s}^R) \ : \ a \stackrel{\tilde{s}^R}{\Rightarrow} b \text{ mit } \tilde{s}^R = (\tilde{s}_1, \ldots, \tilde{s}_r) \right\} \\ &= d_w(b,a). \end{aligned}$$

Hierbei bezeichnet $\tilde{s}_i$ für eine Edit-Operation $s_i = (x, y)$ mit $x, y \in \overline{\Sigma}$ die inverse Edit-Operation $\tilde{s}_i = (y, x)$. 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 $a \stackrel{s}{\Rightarrow} b$ und $w(s) = d_w(a, b)$ sowie $b \stackrel{t}{\Rightarrow} c$ und $w(t) = d_w(b, c)$. Dann ist offensichtlich die Konkatenation $s \cdot t$ eine Folge von Edit-Operationen mit $a \stackrel{s \cdot t}{\Longrightarrow} c$. 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 bewiesen. ■

3.1.2 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 3.6 *Sei $u \in \overline{\Sigma}^*$. Dann ist Restriktion von u auf Σ 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 $u \in \overline{\Sigma}^*$ ist also nichts anderes als das Löschen aller Zeichen $-$ aus u. Nun sind wir formal in der Lage, ein Alignment zu definieren.

Definition 3.7 *Ein Alignment ist ein Paar $(\overline{a}, \overline{b}) \in \overline{\Sigma}^*$ mit $|\overline{a}| = |\overline{b}|$ und*

$$\forall i \in [1 : |\overline{a}|] : \overline{a}_i = \overline{b}_i \Rightarrow \overline{a}_i \neq - \neq \overline{b}_i.$$

$(\overline{a}, \overline{b})$ ist ein Alignment für $a, b \in \Sigma^*$, wenn gilt:

$$\overline{a}|_\Sigma = a \text{ und } \overline{b}|_\Sigma = b.$$

Betrachten wir das folgende Beispiel in Abbildung 3.3, ein Alignment zwischen $a = AGGCATT$ und $b = AGCGCTT$. Mit Hilfe solcher Alignments lassen sich jetzt ebenfalls Distanzen zwischen Zeichenreihen definieren.

$$\begin{array}{cccccccc} A & - & G & G & C & A & T & T \\ A & G & C & G & C & - & T & T \end{array}$$

Abbildung 3.3: Beispiel: Alignment von $AGGCATT$ mit $AGCGCTT$

Definition 3.8 *Sei $\overline{w} : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine (Kosten-)Funktion. Die Kosten eines Alignments $(\overline{a}, \overline{b})$ für (a, b) sind definiert als*

$$\overline{w}(\overline{a}, \overline{b}) := \sum_{i=1}^{|\overline{a}|} \overline{w}(\overline{a}_i, \overline{b}_i).$$

Die Alignment-Distanz von $a, b \in \Sigma^*$ ist definiert als

$$\overline{d}_{\overline{w}}(a, b) := \min\{\overline{w}(\overline{a}, \overline{b}) \mid (\overline{a}, \overline{b}) \textit{ ist Alignment für } a, b\}.$$

Für die Kostenfunktion gilt hier dasselbe wie für die Kostenfunktion für die Edit-Distanz. Wählt man im obigen Beispiel in Abbildung 3.3 $w(x, x) = 0$ 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.

3.1.3 Beziehung zwischen Edit- und Alignment-Distanz

In diesem Abschnitt wollen wir zeigen, dass Edit-Distanz und Alignment-Distanz unter bestimmten Voraussetzungen gleich sind.

Lemma 3.9 *Sei $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ 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 $a \stackrel{s}{\Longrightarrow} b$ 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): $a \stackrel{s}{\Longrightarrow} b$. Für die Edit-Distanz erhalten wir:

$$w(s) = \sum_{i=1}^{|\overline{a}|} w(s_i) = \sum_{i=1}^{|\overline{a}|} w(\overline{a}_i, \overline{b}_i) = w(\overline{a}, \overline{b}).$$

■

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 3.10 *Sei $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine Kostenfunktion, dann gilt für alle $a, b \in \overline{\Sigma}^*$:*

$$d_w(a,b) \leq \overline{d}_w(a,b).$$

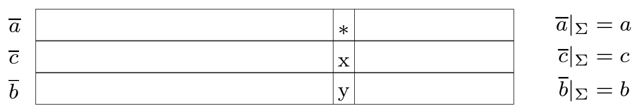
Nun wollen wir die umgekehrte Richtung untersuchen.

Lemma 3.11 *Sei $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine metrische Kostenfunktion und seien $a, b \in \Sigma^*$. Für jede Folge von Edit-Operationen mit $a \stackrel{s}{\Longrightarrow} b$ 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 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 $w(\overline{a}, \overline{b}) = 0 \leq w(s)$.

Induktionsschritt ($n \to n+1$): Sei $s = (s_1, \ldots, s_n, s_{n+1})$ eine Folge von Edit-Operationen mit $a \stackrel{s}{\Longrightarrow} b$. Sei nun $s' = (s_1, \ldots, s_n)$ und $a \stackrel{s'}{\Longrightarrow} c \stackrel{s_{n+1}}{\longrightarrow} b$ für ein $c \in \Sigma^*$. Aus der Induktionsvoraussetzung folgt nun, dass es ein Alignment $(\overline{a}, \overline{c})$ von a, c gibt, so dass $w(\overline{a}, \overline{c}) \leq w(s')$.



Sei $s_{n+1} = (x, y)$ eine Substitution, Match oder Deletion.

Abbildung 3.4: Skizze: zum Induktionsbeweis (Substitution)

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 3.4 schematisch dargestellt, ein Alignment für a und b erzeugen, indem wir die Zeichenreihe $\overline{b}$ geeignet aus $\overline{c}$ unter Verwendung der Edit-

Operation(x, y) umformen. Es gilt dann:

$$
\begin{aligned}
w(\overline{a}, \overline{b}) &= w(\overline{a}, \overline{c}) \underbrace{-w(\overline{a}_i, \overline{c}_i) + w(\overline{a}_i, \overline{b}_i)}_{\leq w(\overline{b}_i, \overline{c}_i)} \\
&\quad \text{aufgrund der Dreiecksungleichung und der Symmetrie} \\
&\quad \text{d.h., } w(\overline{a}_i, \overline{b}_i) \leq w(\overline{a}_i, \overline{c}_i) + w(\overline{b}_i, \overline{c}_i) \\
&\leq \underbrace{w(\overline{a}, \overline{c})}_{\leq w(s')} + \underbrace{w(\overline{b}_i, \overline{c}_i)}_{= w(s_{n+1})} \\
&\leq w(s') + w(s_{n+1}) \\
&= w(s).
\end{aligned}
$$

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 3.5 noch einmal schematisch

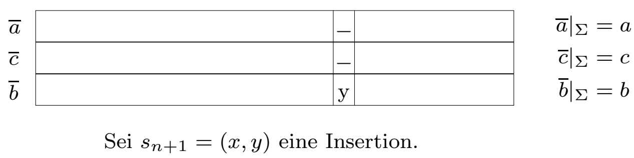


Abbildung 3.5: Skizze: zum Induktionsbeweis(Insertion)

dargestellt. $(\overline{a}, \overline{c})$ ist jetzt 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

$$
\begin{aligned}
w(\overline{a}, \overline{b}) &= w(\overline{a}, \overline{c}) + w(-, y) \\
&\quad \text{Nach Induktionsvoraussetzung} \\
&\leq w(s') + w(s_n) \\
&= w(s).
\end{aligned}
$$

■

Aus diesem Lemma folgt jetzt, dass die Alignment-Distanz durch die Edit-Distanz beschränkt ist, sofern die zugrunde liegende Kostenfunktion eine Metrik ist.

Korollar 3.12 *Ist $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ 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 3.13 *Ist w eine Metrik, dann gilt: $d_w = \overline{d}_w$*

Man überlege sich, dass 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 $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ wie folgt ab.

Definition 3.14 *Eine Kostenfunktion $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ für ein Distanzmaß heißt* sinnvoll*, wenn folgende Bedingungen erfüllt sind:*

(D1) $\max\{w(x,x) \mid x \in \Sigma\} \leq \min\{w(x,y) \mid x \neq y \in \overline{\Sigma}\}$*;*

(D2) $\forall x, y \in \overline{\Sigma} : w(x,y) = w(y,x)$ *(Symmetrie);*

(D3) $\forall x, y, z \in \overline{\Sigma} : w(x,z) \leq w(x,y) + w(y,z)$ *(Dreiecksungleichung).*

Von der Definitheit bleibt hier also nur noch übrig, dass gleiche Zeichen einen geringeren Abstand (aber nicht notwendigerweise 0) haben müssen als verschiedene Zeichen. Will man die Äquivalenz von Edit- und Alignment-Distanz weiterhin sicherstellen, so wird man ebenfalls $\max\{w(x,x) \mid x \in \Sigma\} = 0$ fordern. Im Folgenden werden wir für alle Kostenfunktionen für Distanzmaße voraussetzen, dass zumindest D1, D2 und D3 gelten.

3.1.4 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 werden. Man hat also insbesondere die Möglichkeit, die Gleichheit von gewissen Zeichen stärker zu bewerten als von anderen Zeichen. Formal lässt sich eine Kostenfunktion für Ähnlichkeitsmaße wie folgt definieren.

Definition 3.15 *Ein Kostenfunktion* $w' : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}$ *für ein Ähnlichkeitsmaß heißt* sinnvoll, *wenn die folgenden Bedingungen erfüllt sind:*

(S1) $\forall x \in \Sigma : w'(x, x) \geq 0$*;*

(S2) $\forall x \neq y \in \overline{\Sigma} : w'(x, y) \leq 0$*;*

(S3) $\forall x, y \in \overline{\Sigma} : w'(x, y) = w'(y, x)$*.*

Im Folgenden werden wir für alle Kostenfunktionen für Ähnlichkeitsmaße voraussetzen, dass S1, S2 und S3 gelten. Basierend auf solchen Kostenfunktionen für Ähnlichkeitsmaße können wir jetzt die Ähnlichkeit von Alignments definieren.

Definition 3.16 *Sei* $w' : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}$ *eine Kostenfunktion und sei* $(\overline{a}, \overline{b})$ *ein Alignment für* $a, b \in \Sigma^*$*. Dann ist die Ähnlichkeit 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 Ähnlichkeit von $a, b \in \Sigma^*$ *ist definiert als:*

$$s(a, b) := \max\{w'(\overline{a}, \overline{b}) \mid (\overline{a}, \overline{b}) \text{ ist Alignment für } a, b\}$$

3.1.5 Beziehung zwischen Distanz- und Ähnlichkeitsmaßen

Die folgenden beiden Lemmata verdeutlichen die Beziehung, die zwischen Ähnlichkeitsmaß und Distanzmaß besteht.

Lemma 3.17 *Sei w eine sinnvolle Kostenfunktion für ein Distanzmaß. Dann existiert ein $C \in \mathbb{R}_+$, so dass $w'(a,b) = C - w(a,b)$ eine sinnvolle Kostenfunktion für ein Ähnlichkeitsmaß ist.*

Beweis: Sei $A := \max\{w(a,a) : a \in \Sigma\}$ und $B := \min\left\{w(a,b) : a \neq b \in \overline{\Sigma}\right\}$. Da w eine sinnvolle Kostenfunktion für ein Distanzmaß ist, gilt $A \leq B$. Wir wählen daher $C \in [A : B]$.

Für $a \in \Sigma$ gilt:

$$w'(a,a) = C - w(a,a) \geq A - w(a,a) = \max\{w(a,a) : a \in \Sigma\} - w(a,a) \geq 0.$$

Somit haben wir die Eigenschaft S1 nachgewiesen.

Für $a \neq b \in \overline{\Sigma}$ gilt:

$$w'(a,b) = C - w(a,b) \leq B - w(a,b) = \min\left\{w(a,b) : a \neq b \in \overline{\Sigma}\right\} - w(a,b) \leq 0.$$

Somit haben wir die Eigenschaft S2 nachgewiesen.

Die Eigenschaft S3 folgt offensichtlich aus der Definition von w' und D2. ■

Lemma 3.18 *Sei $w' : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine sinnvolle Kostenfunktion für ein Ähnlichkeitsmaß. Dann existiert ein $D \in \mathbb{R}_+$, so dass $w(a,b) = D - w'(a,b)$ eine sinnvolle Kostenfunktion für ein Distanzmaß ist.*

Beweis: Wir wählen $D = \max\left\{w'(a,b) : a,b \in \overline{\Sigma}\right\}$. Nach Wahl von D gilt dann

$$\begin{aligned} w(a,b) &= D - w'(a,b) \\ &= \max\left\{w'(a,b) : a,b \in \overline{\Sigma}\right\} - w'(a,b) \\ &\geq 0. \end{aligned}$$

Somit haben wir nachgewiesen, dass $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$.

Wir weisen jetzt D1 nach. Da wegen S1 und S2 gilt

$$\min\{w'(a,a) \mid a \in \Sigma\} \geq 0 \geq \max\{w'(a,b) \mid a \neq b \in \overline{\Sigma}\},$$

folgt, dass

$$\begin{aligned}\max\{w(a,a) \mid a \in \Sigma\} &= \max\{D - w'(a,a) \mid a \in \Sigma\}\\ &= D - \min\{w'(a,a) \mid a \in \Sigma\}\\ &\leq D - \max\{w'(a,b) \mid a \neq b \in \overline{\Sigma}\}\\ &= \min\{D - w'(a,b) \mid a \neq b \in \overline{\Sigma}\}\\ &= \min\{w(a,b) \mid a \neq b \in \overline{\Sigma}\}.\end{aligned}$$

D2 folgt nach Definition von w unmittelbar aus S3.

Wir müssen nur noch die Dreiecksungleichung beweisen: $w(x,z) \leq w(x,y) + w(y,z)$ für alle $x,y,z \in \Sigma$. Dies geschieht durch eine vollständige Fallunterscheidung.

Fall 1 ($\boldsymbol{x = y = z}$): Dann gilt offensichtlich $w(x,x) \leq w(x,x) + w(x,x)$, da $w(x,x) \geq 0$.

Fall 2 ($\boldsymbol{x = y \neq z}$): Dann gilt

$$w(x,z) = w(y,z) \leq w(x,y) + w(y,z),$$

da wir wissen, dass $w(x,y) = w(x,x) \geq 0$.

Fall 3 ($\boldsymbol{x = z \neq y}$): Es gilt nun mit Hilfe von S3, dass

$$w(x,z) = w(x,x) = D - w'(x,x) \quad \text{und} \quad w(x,y) + w(y,z) = 2w(y,z) = 2D - w'(y,z).$$

Wir müssen also nur noch zeigen, dass

$$D - w'(x,x) \leq 2D - 2w'(y,z) \quad \text{bzw.} \quad 2w'(y,z) \leq D + w'(x,x)$$

gilt. Dies gilt aber offensichtlich, da $D \geq 0$ und da nach Definition eines Ähnlichkeitsmaßes $w'(y,z) \leq 0$ für $y \neq z \in \overline{\Sigma}$ und $w'(x,x) \geq 0$ für $x \in \Sigma$ gilt.

Fall 4 ($\boldsymbol{|\{x,y,z\}| = 3}$): Es gilt wiederum

$$w(x,z) = D - w'(x,z) \quad \text{und} \quad w(x,y) + w(y,z) = 2D - w'(x,y) - w'(y,z).$$

Damit genügt es zu zeigen, dass $w'(x,y) + w'(y,z) \leq D + w'(x,z)$ gilt. Nach Definition eines Ähnlichkeitsmaßes gilt $w'(x,y) + w'(y,z) \leq 0$ für paarweise verschiedene $x,y,z \in \overline{\Sigma}$. Nach Wahl von D gilt außerdem, dass $D + w'(x,z) \geq 0$ ist. Damit ist der Beweis abgeschlossen. ■

Damit haben wir trotz der unterschiedlichen Definition gesehen, dass Distanzen und Ähnlichkeiten eng miteinander verwandt sind. Dies unterstreicht insbesondere auch noch einmal der folgende Satz.

Theorem 3.19 *Sei $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine sinnvolle Kostenfunktion für ein Distanzmaß d und sei $C \in \mathbb{R}_+$ so gewählt, dass $C - w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}$ eine sinnvolle Kostenfunktion für ein Ähnlichkeitsmaß s ist und dass gilt:*

$$\forall x, y \in \Sigma : \quad C \leq w(x, -) + w(y, -) - w(x, y). \tag{3.1}$$

Dann gilt für ein Alignment $(\overline{a}, \overline{b})$ für $a, b \in \Sigma^*$:

$$\overline{d}(a, b) + s(a, b) = C \cdot \ell$$

für eine geeignet gewählte Konstante $\ell \in \mathbb{N}_0$.

Beweis: Für ein beliebiges Alignment $(\overline{a}, \overline{b})$ für $a, b \in \Sigma^*$ gilt offensichtlich, dass

$$w(\overline{a}, \overline{b}) + s(\overline{a}, \overline{b}) = w(\overline{a}, \overline{b}) + (C - w)(\overline{a}, \overline{b}) = w(\overline{a}, \overline{b}) + C \cdot |\overline{a}| - w(\overline{a}, \overline{b}) = C \cdot |\overline{a}|.$$

Beachte hierbei, dass $|\overline{a}| = |\overline{b}|$. Da nach Definition

$$\begin{aligned}
\overline{d}(a, b) &= \min \left\{ w(\overline{a}, \overline{b}) \ : \ (\overline{a}, \overline{b}) \text{ ist Alignment für } a, b \right\}, \\
s(a, b) &= \max \left\{ (C - w)(\overline{a}, \overline{b}) \ : \ (\overline{a}, \overline{b}) \text{ ist Alignment für } a, b \right\},
\end{aligned}$$

gilt, müssen wir nur noch zeigen, dass das Minimum beim Distanzmaß und das Maximum beim Ähnlichkeitsmaß an denselben Stellen angenommen wird.

Wir nehmen für einen Widerspruchsbeweis an, es gäbe bezüglich des Ähnlichkeitsmaßes ein besseres Alignment. Sei $(\overline{a}, \overline{b})$ ein optimales Alignment für a und b für das Distanzmaß, d.h. $w(\overline{a}, \overline{b}) = \overline{d}(a, b)$. Dann gilt $s(a, b) > (C - w)(\overline{a}, \overline{b})$ und es gibt ein optimales Alignment $(\tilde{a}, \tilde{b})$ für a und b mit $s(a, b) = (C - w)(\tilde{a}, \tilde{b})$. Also gilt $(C - w)(\overline{a}, \overline{b}) < (C - w)(\tilde{a}, \tilde{b})$ und somit (mit $|\overline{a}| = |\overline{b}|$ und $|\tilde{a}| = |\tilde{b}|$)

$$w(\tilde{a}, \tilde{b}) < w(\overline{a}, \overline{b}) + C(|\tilde{a}| - |\overline{a}|). \tag{3.2}$$

Da nach Voraussetzung $w(\overline{a}, \overline{b}) \leq w(\tilde{a}, \tilde{b})$ gilt, folgt mit 3.2 sofort $|\overline{a}| < |\tilde{a}|$. Somit hat das Alignment $(\tilde{a}, \overline{a})$ mehr Indel-Stellen als das Alignment $(\overline{a}, \overline{b})$. Somit müssen Matches oder Substitutionen durch Insertionen und Deletionen ersetzt worden sein. Die Ungleichung 3.1 impliziert jedoch, dass jede solche Verlängerung eines Alignments die Distanz um mindestens C erhöht. Somit gilt

$$w(\tilde{a}, \tilde{b}) \geq w(\overline{a}, \overline{b}) + C(|\tilde{a}| - |\overline{a}|) \tag{3.3}$$

Aus den Ungleichungen 3.2 und 3.3 folgt sofort

$$w(\overline{a}, \overline{b}) + C(|\tilde{a}| - |\overline{a}|) > w(\tilde{a}, \tilde{b}) \geq w(\overline{a}, \overline{b}) + C(|\tilde{a}| - |\overline{a}|),$$

was offensichtlich ein Widerspruch ist. Die umgekehrte Richtung lässt sich analog beweisen. ■

Wir merken hier noch an, dass die Ungleichung 3.1 im Satz äquivalent zu der Bedingung $s(x, y) \geq s(x, -) + s(-, y)$ für alle $x, y \in \Sigma$ ist. Somit wird hier nur gefordert, dass auch im Ähnlichkeitsmaß eine Substitution besser bewertet wird als deren Ersetzung durch eine Insertion und eine Deletion.

Unter gewissen Voraussetzungen kann man also beliebig zwischen Edit-, Alignment-Distanz oder Ähnlichkeitswerten wechseln, ohne in der Bewertung der Ähnlichkeit von Sequenzen andere Ergebnisse zu erhalten.

3.2 Bestimmung optimaler globaler Alignments

In diesem Abschnitt geht es nun darum, optimale Alignments für zwei Zeichenreihen tatsächlich zu berechnen. Dabei werden die beiden Zeichenreihen zueinander „aligned“, also so zueinander ausgerichtet, dass sie danach dieselbe Länge haben und so ähnlich wie möglich sind (bezüglich eines geeignet gewählten Distanz- oder Ähnlichkeitsmaßes).

Geg.: $s = s_1 \cdots s_n \in \Sigma^n$, $t = t_1 \cdots t_m \in \Sigma^m$ und ein Distanz- oder Ähnlichkeitsmaß d.
Ges.: Ein optimales Alignment $(\overline{s}, \overline{t})$ für s, 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.

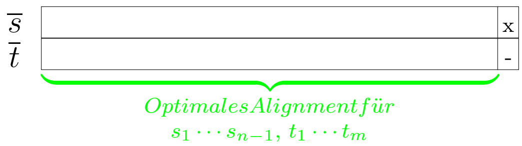
3.2.1 Der Algorithmus nach Needleman-Wunsch

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:

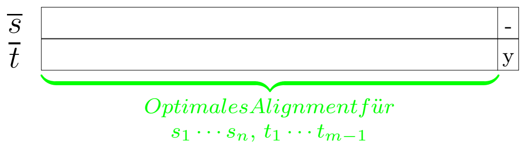
1. $x = s_n$ wurde durch $y = t_m$ substituiert:

$$\begin{array}{c} \overline{s} \\ \overline{t} \end{array} \underbrace{\begin{array}{|c|} \hline \qquad\qquad\qquad\qquad \\ \hline \qquad\qquad\qquad\qquad \\ \hline \end{array}}_{\substack{Optimales Alignment für \\ s_1 \cdots s_{n-1},\, t_1 \cdots t_{m-1}}} \begin{array}{|c|} \hline x \\ \hline y \\ \hline \end{array}$$

2. Das letzte Zeichen $x = s_n$ in s wurde gelöscht:



3. Das letzte Zeichen $y = t_m$ in t wurde eingefügt:

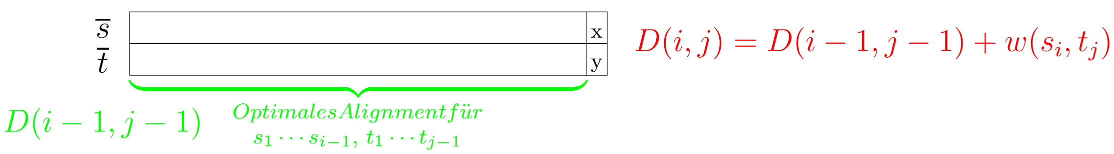


In allen drei Fällen überlegt man sich leicht, dass das Alignment, das durch Streichen der letzten Spalte entsteht, also $(\overline{s}_1 \cdots \overline{s}_{|\overline{s}|-1}, \overline{t}_1 \cdots \overline{t}_{|\overline{t}|-1})$, ebenfalls ein optimales Alignment für $s_1 \cdots s_{n-1}$ mit $t_1 \cdots t_{m-1}$, $s_1 \cdots s_{n-1}$ mit $t_1 \cdots t_m$ bzw. $s_1 \cdots s_n$ 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.

Mit diesen Vorüberlegungen können wir das Verfahren von Needleman-Wunsch formulieren, das 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 $D(i, j)$ 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:

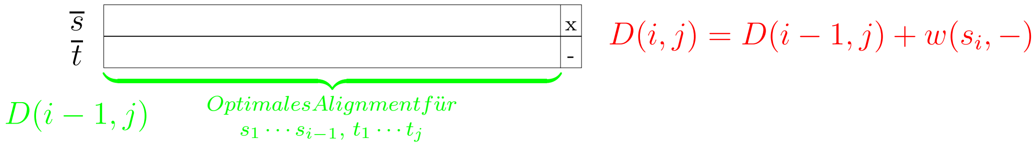
$$D(i,j) \quad = \quad \min \left\{ \begin{array}{lcl} D(i-1, j-1) & + & w(s_i, t_j), \\ D(i-1, j) & + & w(s_i, -), \\ D(i, j-1) & + & w(-, t_j) \end{array} \right\} .$$

Die folgenden Bilder 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 für die Substitution von s_n durch t_m hinzuaddieren:

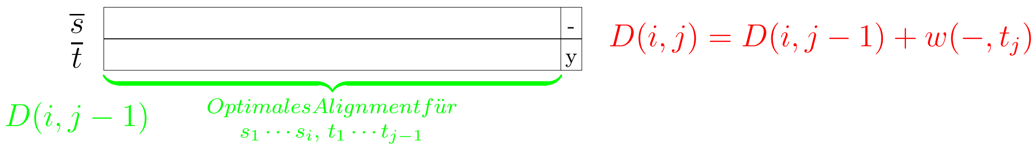


Im zweiten Fall wurde ein Zeichen in t gelöscht. Um nun die Distanz eines Alignments für $s_1 \cdots s_i$ und $t_1 \cdots t_j$ zu erhalten, muss zu den Kosten dieser Löschung noch die

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 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.

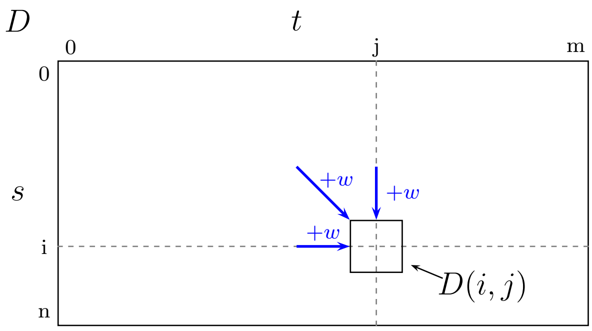


Abbildung 3.6: Skizze: Berechnung optimaler Alignments nach Needleman-Wunsch

Abbildung 3.6 zeigt schematisch, wie der Wert eines Eintrags $D(i,j)$ in der Matrix D von den anderen Werten aus D abhängt. Nun fehlen nur noch die zugehörigen Anfangswerte:

$$D(0,0) = 0, \qquad D(i,0) = \sum_{k=1}^{i} w(s_k, -), \qquad D(0,j) = \sum_{k=1}^{j} w(-, t_k).$$

Die Korrektheit folgt aus der Überlegung, dass in der ersten Spalte $D(i,0)$ die Abstände optimaler Alignments von $s_1 \cdots s_i$ mit $t_1 \cdots t_0 = \varepsilon$ stehen. Es bleibt einem

SEQUENCEALIGNMENT (char $s[]$, int n, char $t[]$, int m)

```
{
    D[0,0] = 0;
    for (i = 1; i ≤ n; i++)
        D[i,0] = D[i − 1,0] + w(s_i, −);
    for (j = 1; j ≤ m; j++)
        D[0,j] = D[0,j − 1] + w(−, t_j);
    for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ m; j++)
            D[i,j] = min { D[i − 1,j] + w(s_i, −),
                           D[i,j − 1] + w(−, t_j),
                           D[i − 1,j − 1] + w(s_i, t_j) };
}
```

Abbildung 3.7: Algorithmus: Verfahren von Needleman und Wunsch

gar nichts anderes übrig, als alle Zeichen aus $s_1 \cdots s_j$ zu löschen. Analoges gilt für die erste Zeile.

Folgendes Beispiel zeigt ausführlich für zwei Sequenzen s und t, wie deren optimale Alignment-Distanz bestimmt wird.

$$
\begin{array}{lcccccccc}
s & = & A & G & G & C & T & G & \\
t & = & A & C & C & G & G & T & A
\end{array}
$$

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. 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". An die jeweilige Stelle im Graphen wird die momentane Distanz des jeweiligen Alignments eingetragen, berechnet mit der Rekursionformel von weiter oben. 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 natürlich keine Kosten.

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

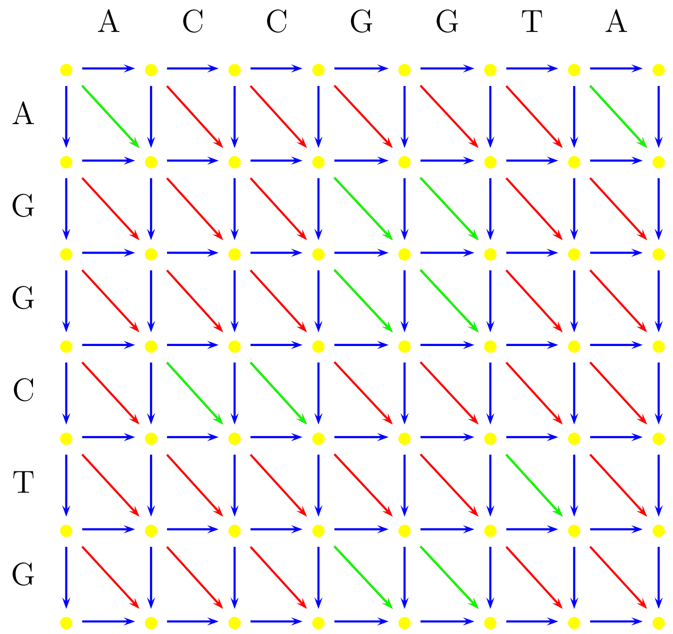
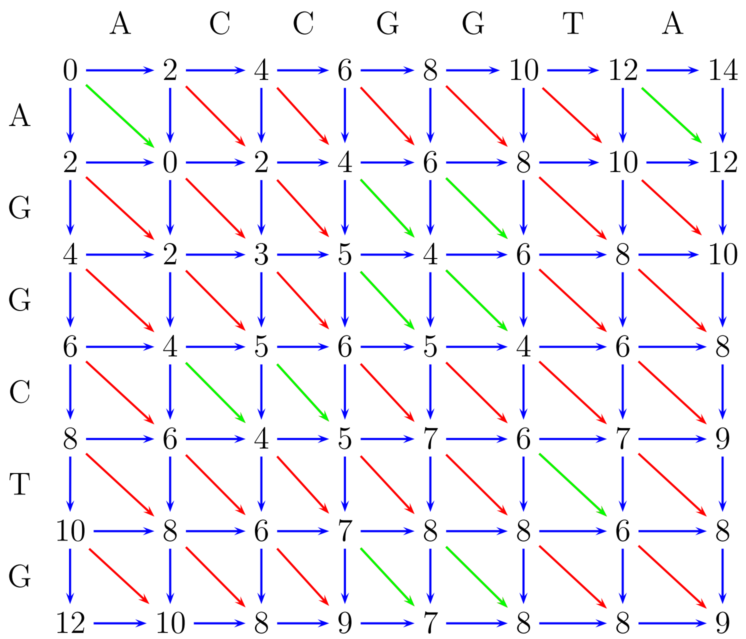


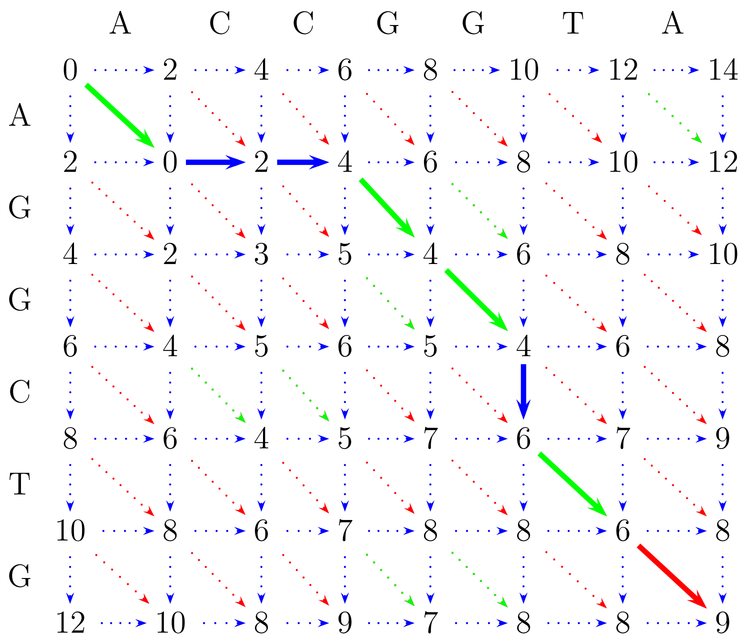
Abbildung 3.8: Skizze: Edit-Graph für s und t ohne Distanzen



Match = 0, Indel = 2, Subst = 3

Abbildung 3.9: Skizze: Edit-Graph für s und t mit Distanzen 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.



Match = 0, Indel = 2, Subst = 3

Abbildung 3.10: Skizze: Pfad im Edit-Graphen zur Bestimmung des Alignments

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 3.11 dargestellt ist.

s :	A	–	–	G	G	C	T	G
t :	A	C	C	G	G	–	T	A

Abbildung 3.11: Beispiel: Optimales globales Alignment von s mit t

Fassen wir zum Abschluss dieses Abschnittes noch das Ergebnis zusammen.

Theorem 3.20 *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 das algorithmische Lösen von Rekursionsgleichungen mit Hilfe von Tabellen *dynamische Programmierung* genannt wird. Normalerweise würde man Rekursionsgleichungen mit Hilfe von Rekursionen lösen. Werden hierbei jedoch sehr oft gleiche Teilprobleme rekursiv gelöst, so ist dies sehr ineffizient. Mit Hilfe der Tabellen werden dabei die Ergebnisse von bereits gelösten Teilproblemen gespeichert, so dass sie wiederverwendet werden können, was in diesen Fällen die Effizienz erheblich erhöht.

Musterbeispiel hierfür ist die Berechnung der n-ten Fibonacci-Zahl f_n die rekursiv durch $f_n = f_{n-1} + f_{n-2}$ und $f_1 = 1$ und $f_0 = 0$ definiert ist. Würde man diese Rekursionsgleichung rekursiv lösen, so würde man exponentiell oft den Wert f_2 berechnen (das gilt im Prinzip für alle Werte). Somit würde eine rekursive Lösung exponentiellen Aufwand in n benötigen. Berechnet man hingegen die Fibonacci-Zahlen mit Hilfe der dynamischen Programmierung, so füllt man eine Tabelle $F(i)$ für $i = 2, \ldots, n$ aus. Dafür sind nur linear (in n) viele Additionen nötig.

3.2.2 Sequenzen Alignment mit linearem Platz (Modifikation von 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 3.12 beschreibt genau dieses Verfahren.

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, da ja die nötigen Zwischenergebnisse nicht gespeichert wurden.

SequenceAlignment (char $s[]$, int n, char $t[]$, int m)

```
{
    D[0] = 0;
    for (j = 0; j ≤ m; j++)
        D[j] = D[j − 1] + w(−, t_j);
    for (i = 1; i ≤ n; i++)
    {
        for (j = 0; j ≤ m; j++)
            D'[j] = D[j];
        D[0] = D'[0] + w(s_i, 0);
        for (j = 1; j ≤ m; j++)
            D[j] = min { D'[j] + w(s_i, −),
                         D[j − 1] + w(−, t_j),
                         D'[j − 1] + w(s_i, t_j) };
    }
}
```

Abbildung 3.12: Algorithmus: platzsparende Variante von Needleman-Wunsch

Mit dem Verfahren nach Hirschberg kann ein optimales Sequenzen Alignment selbst konstruiert werden, so dass nur linear viel Platz benutzt werden muss. Dazu betrachten wir zunächst einmal ein optimales paarweises Alignment von s mit t, wie in der folgenden Abbildung 3.13 angegeben. Wir teilen nun dieses Alignment so in zwei

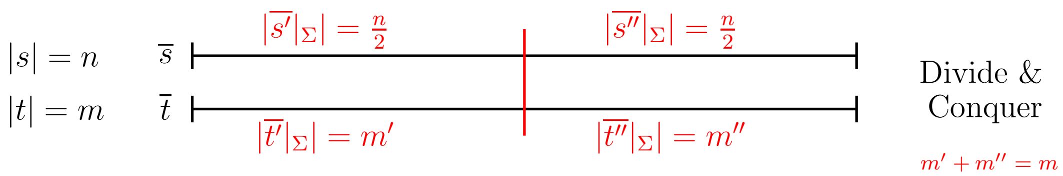


Abbildung 3.13: Skizze: Optimales Alignment für s und t

Teil-Alignments auf, dass beide Teile in etwa die Hälfte der Zeichen aus s enthalten: der erste Teil enthalte $\lceil n/2 \rceil$ und der zweite Teil $\lfloor n/2 \rfloor$ Zeichen aus s. Um uns im Folgenden das Leben etwas leichter zu machen, nehmen wir an, dass n gerade ist und wir somit ohne die Gauß-Klammern weiter arbeiten können.

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 $s_{n/2}$ und dem Zeichen $s_{n/2+1}$ 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 $|s'| = |s''| = n/2$ und $|t'| = m'$ bzw. $|t''| = m''$.

Zuerst einmal bemerken wir, 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 $s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_{m'}$ sowie für $s_{n/2} \cdots s_n$ 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 3.14 dargestellt ist.

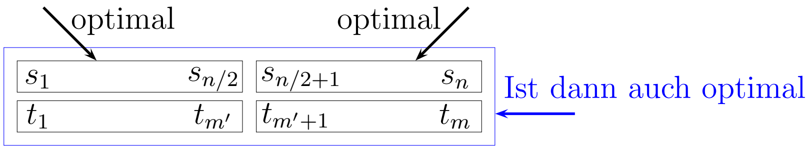


Abbildung 3.14: Skizze: Conquer-Schritt des Hirschberg-Algorithmus

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 3.15 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 $n/2$ an einem Punkt m' schneiden. Wir merken hier nochmals an, dass dieser Punkt nicht eindeutig sein muss, da aufgrund von Insertionen der Pfad waagerecht innerhalb der Zeile $n/2$ verlaufen kann.

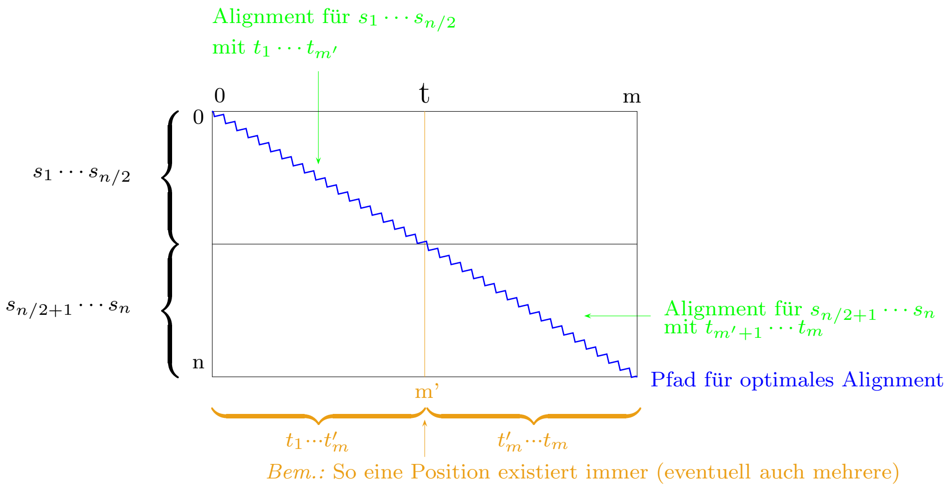


Abbildung 3.15: Skizze: Auffinden von m' (Divide-Schritt bei Hirschberg)

Der Pfad von $(0, 0)$ nach (n, m) zerfällt also in zwei Teile, nämlich in $(0, 0)$ bis $(n/2, m')$ und in $(n/2, m')$ 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 $s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_{m'}$ für alle $m' \in [0 : m]$. Dies können wir mit unserem vorhin in Abbildung 3.12 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 $s_{n/2} \cdots s_n$ mit $t_{m'} \cdots t_m$ 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 $(s'')^R = s_n \cdots s_{n/2}$ mit $(t'')^R = t_m \cdots t_{m'}$, 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 3.21 *Sei* $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ *eine Kostenfunktion und seien* $s, t \in \Sigma^*$, *dann gilt* $\overline{d}_w(s, t) = \overline{d}_w(s^R, t^R)$.

Beweis: Es gilt für beliebige $\overline{s}, \overline{t} \in \overline{\Sigma}^*$ mit $\overline{s}|_\Sigma = s$ und $\overline{t}|_\Sigma = t$:

$$\begin{aligned} w(\overline{s}, \overline{t}) &= \sum_{i=1}^{|\overline{s}|} w(\overline{s}_i, \overline{t}_i) \\ &= \sum_{i=1}^{|\overline{s}|} w(\overline{s}_{|\overline{s}|-i+1}, \overline{t}_{|\overline{t}|-i+1}) \\ &= w(\overline{s^R}, \overline{t^R}). \end{aligned}$$

Somit gilt auch:

$$\begin{aligned} \overline{d}_w(s,t) &= \min\{w(\overline{s}, \overline{t}) \mid (\overline{s}, \overline{t}) \text{ ist ein Alignment für } s,t\} \\ &= \min\{w(\overline{s^R}, \overline{t^R}) \mid (\overline{s^R}, \overline{t^R}) \text{ ist ein Alignment für } s^R, t^R\} \\ &= \overline{d}_w(s^R, t^R). \end{aligned}$$

■

Bezeichne $V(n/2, k)$ die minimale Alignment-Distanz von $s' = s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_k$ und $V'(n/2, k)$ die minimale Alignment-Distanz von $s'' = s_{n/2+1} \cdots s_n$ mit $t_k \cdots t_m$ was nach obigem Satz gleichbedeutend mit der minimalen Alignment-Distanz von $(s'')^R$ mit $t_m \cdots t_k$ ist.

$$\begin{aligned} V(\frac{n}{2}, k) &= \overline{d}(s_1 \cdots s_{n/2}, t_1 \cdots t_k) \\ V'(\frac{n}{2}, k) &= \overline{d}(s_{n/2+1} \cdots s_n, t_{k+1} \cdots t_m) \;=\; \overline{d}(s_n \cdots s_{n/2+1}, t_m \cdots t_{k+1}) \end{aligned}$$

Nach unseren Überlegungen gilt für das optimale m', dass für die optimale Edit-Distanz gilt: $\overline{d}(s,t) = V(n/2, m') + V'(n/2, m')$. Wir können also $\overline{d}(s,t)$ und m' wie folgt berechnen:

$$\begin{aligned} \overline{d}(s,t) &= \min\left\{V(\frac{n}{2}, k) + V'(\frac{n}{2}, k)) \;:\; k \in [0:m]\right\} \\ m' &= \operatorname{argmin}\left\{V(\frac{n}{2}, k) + V'(\frac{n}{2}, k)) \;:\; k \in [0:m]\right\} \end{aligned}$$

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 der folgenden Abbildung 3.16 ist der vollständige Hirschberg-Algorithmus angegeben. Wir bestimmen also zunächst den „Mittelpunkt des Pfades eines optimalen Alignments“ $(n/2, m')$, 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.

1. Berechne die Werte optimaler Alignments für $s' = s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_k$ für alle $k \in [0 : m]$, d.h. $V(n/2, k)$ für alle $k \in [0 : m]$.

   (In Wirklichkeit $s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_m$.)

2. Berechne die Werte optimaler Alignments für $s'' = s_{n/2+1} \cdots s_n$ mit $t_k \cdots t_m$ für alle $k \in [0 : m]$, d.h. $V'(n/2, k)$ für alle $k \in [0 : m]$.

   (In Wirklichkeit $s_n \cdots s_{n/2+1}$ mit $t_m \cdots t_1$.)

3. Bestimme m' mittels $m' = \text{argmin}\{V(\frac{n}{2}, k) + V'(\frac{n}{2}, k)) \mid k \in [0 : m]\}$.

4. Löse rekursiv die beiden Alignment-Probleme für $s' = s_1 \cdots s_{n/2}$ mit $t_1 \cdots t_{m'}$ sowie $s'' = s_{n/2} \cdots s_n$ mit $t_{m'+1} \cdots t_m$.

Abbildung 3.16: Algorithmus: Verfahren von Hirschberg

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.

$$
\begin{array}{llcll}
(s_l, t_p \cdots t_{p'}): & \text{für} \quad t_p \cdots t_{p'} & = & \varepsilon & \Rightarrow \binom{s_l}{-} \\
 & \phantom{\text{für}} \quad t_p \cdots t_{p'} & \neq & \varepsilon & \Rightarrow \binom{-\cdots-s_l-\cdots-}{t_p \cdots\cdots t_{p'}}
\end{array}
$$

$\hookrightarrow$ hier steht das Zeichen s_l
sofern s_l in $t_p \cdots t_{p'}$ vorkommt,
ansonsten ist die Position egal.

Folgendes Beispiel verdeutlicht die Vorgehensweise beim Verfahren von Hirschberg zur Bestimmung eines optimalen Sequenzen Alignments anhand von zwei Sequenzen s und t.

$$
\begin{array}{lcccccc}
s & = & A & G & G & T & \\
t & = & A & C & C & G & T
\end{array}
$$

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 3.17 bildlich dargestellt.

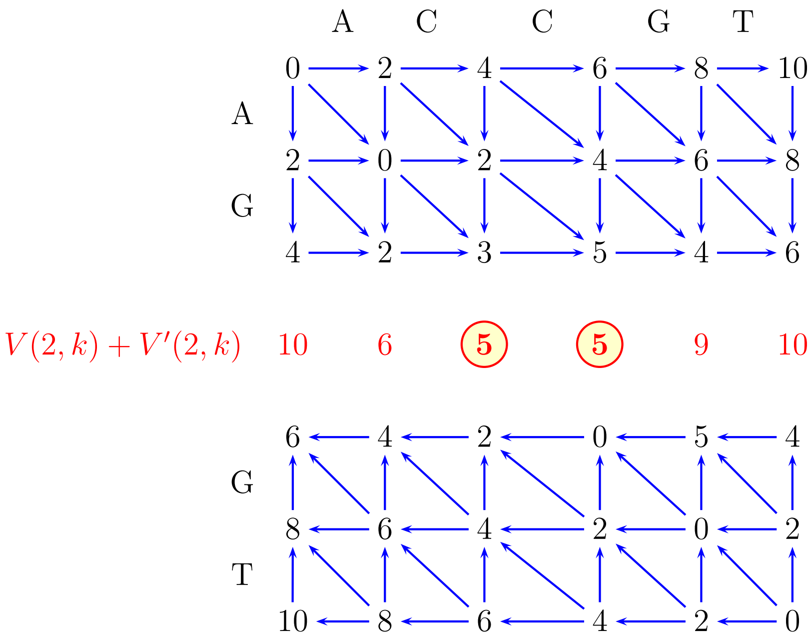


Abbildung 3.17: Beispiel: Bestimmung von m' im Hirschberg-Algorithmus

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 3.18). Wieder haben wir zwei Schnittpunkte zur Wahl, nämlich 1 und 2. Wir entscheiden

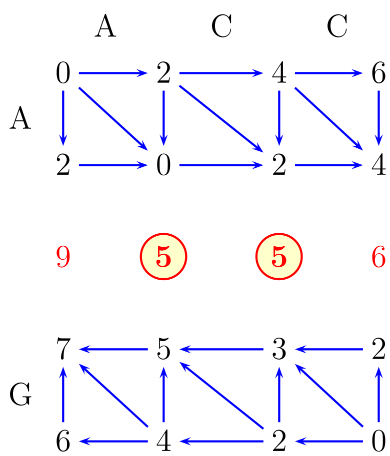


Abbildung 3.18: Beispiel: Erster rekursiver Aufruf im Hirschberg-Algorithmus

und 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 $\binom{G-}{CC}$. 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 3.19). Hier ist der Aufteilungspunkt eindeutig und die Zeichen

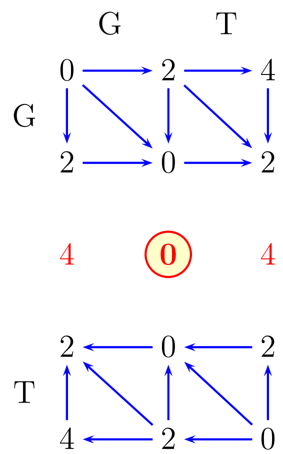


Abbildung 3.19: Beispiel: Zweiter rekursiver Aufruf im Hirschberg-Algorithmus

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.

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 3.16 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 $O(m)$.

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.

Schritt 1 benötigt Laufzeit $c \cdot \frac{n}{2} \cdot m$ für eine geeignet gewählte Konstante c. Schritt 2 benötigt ebenfalls Laufzeit $c \cdot \frac{n}{2} \cdot m$. Schritt 3 benötigt $O(m)$ Operationen. Wir können hier ebenfalls, davon ausgehen, dass dies maximal $c \cdot m$ Operationen sind (im Zweifelsfall erhöhen wir c auf das Maximum der beiden Konstanten). Aufgrund der beiden rekursiven Aufrufe im Schritt 4, ist der Zeitbedarf hierfür durch $T(n/2, k) + T(n/2, m-k)$ gegeben, wobei $k \in [0 : m]$. Somit erhalten wir folgende Rekursionsformel für den Zeitbedarf:

$$\begin{aligned} T(n,m) &= 2c \cdot \frac{n}{2} \cdot m + cm + T\left(\frac{n}{2}, k\right) + T\left(\frac{n}{2}, m-k\right) \\ &= cnm + cm + T\left(\frac{n}{2}, k\right) + T\left(\frac{n}{2}, m-k\right) \end{aligned}$$

Wir könnten diese Rekursionsgleichung mit aufwendigen Mitteln direkt lösen. Wir machen es uns aber hier etwas leichter und verifizieren eine geratene Lösung mittels Induktion.

Behauptung: Es gibt eine Konstante $c \in \mathbb{R}_+$, so dass $T(n,m) \leq 2cnm + cm\log(n)$.

Induktionsanfang ($\boldsymbol{n = 1}$): $T(1,m)$ ist sicherlich $O(m)$, da wir nur ein Zeichen gegen eine Zeichenreihe der Länge m optimal ausrichten müssen. Wenn wir c in der Behauptung hinreichend groß gewählt haben, so gilt die Behauptung sicherlich.

Induktionsschritt ($\boldsymbol{\to n}$): Wir setzen nun die Behauptung als Induktionsvoraussetzung in die Rekursionsformel ein (da $\lceil n/2 \rceil < n$ für $n \geq 2$) und formen um:

$$\begin{aligned} T(n,m) &= cnm + cm + T\left(\frac{n}{2}, k\right) + T\left(\frac{n}{2}, m-k\right) \\ &\leq cnm + cm + 2c\cdot\frac{n}{2}\cdot k + ck\log\left(\frac{n}{2}\right) + 2c\cdot\frac{n}{2}\cdot(m-k) + c(m-k)\log\left(\frac{n}{2}\right) \\ &= cnm + cm + cnk + cn(m-k) + ck\log\left(\frac{n}{2}\right) + c(m-k)\log\left(\frac{n}{2}\right) \\ &= cnm + cm + cnm + cm\log\left(\frac{n}{2}\right) \\ &= 2cnm + cm + cm\log(n) - cm\log(2) \\ &\ \text{da mit log der Logarithmus zur Basis 2 gemeint ist, gilt } \log(2) = 1 \\ &= 2cnm + cm + cm\log(n) - cm \\ &= 2cnm + cm\log(n). \end{aligned}$$

Damit ist die Laufzeit weiterhin $O(nm)$, da $m\log(n) \leq mn$ ist, und wir haben den folgenden Satz bewiesen.

Theorem 3.22 *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.

3.3 Besondere Berücksichtigung von Lücken

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 sublinear in der Länge der Lücke sein.

3.3.1 Semi-Globale Alignments

Im Falle *semi-globaler Alignments* wollen wir Lücken, die am Anfang oder am Ende eines Wortes auftreten, nicht berücksichtigen. 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:

$$\begin{array}{ccccccccccccc} C & G & T & A & C & G & T & G & A & G & T & G & A \\ C & G & - & A & - & - & T & - & - & - & T & - & A \end{array}$$

Dieses hat einen Alignment-Abstand von $7 * 2 = 14$.

Alternativ betrachten wir folgendes Alignment:

$$\begin{array}{cccccccccccccc} C & G & T & A & C & G & - & T & G & A & G & T & G & A \\ - & - & - & - & C & G & A & T & T & A & - & - & - & - \end{array}$$

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 semi-globalen 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 semi-globalen Alignments wesentlich einfacher zu handhaben.

Wir verwenden jetzt als Kostenfunktion für ein Ähnlichkeitsmaß für Matches $+1$, für Insertionen sowie Deletionen -1 und für Substitutionen -2. Dieses Kostenmaß ist aus der Kostenfunktion für das obige Distanzmaß mittels $1 - w(x, y)$ gewonnen worden. Somit erhält man für das erste globale Alignment einen Score von

$$6*(+1)+7*(-1)=-1.$$

Für das zweite Alignment erhält man als globales Alignment einen Ähnlichkeitswert von

$$4*(+1)+8*(-1)+1*(-2)=-6$$

und als semi-globales-Alignment einen Score von

$$4*(+1)+1*(-1)+1*(-2)=+1.$$

Für das künstliche Alignment jedoch einen Score von 0. Wir weisen an dieser Stelle darauf hin, dass die hier verwendeten Kostenfunktionen nicht besonders gut gewählt, aber für die Beispiele ausreichend sind.

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 3.20, in der schematisch semi-globale Alignments dargestellt sind.

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

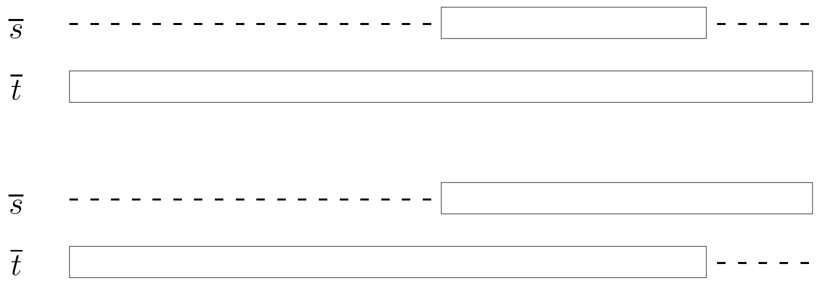


Abbildung 3.20: Skizze: semi-globale Alignments

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 semi-globales Alignment irgendwo in der letzten Zeile auf. Wenn wir nun ein semi-globales Alignment mit maximaler Ähnlichkeit wollen, müssen wir einfach nur in der letzten Zeile den maximalen Wert suchen. Die Spalten dahinter können wir für unser semi-globales Alignment dann einfach vergessen.

Dasselbe gilt für Deletionen in s. Dann hört das semi-globale Alignment irgendwo in der letzten Spalte auf und wir bestimmen für ein optimales semi-globales Alignment den maximalen Wert in der letzten Spalte und vergessen die Zeilen danach.

In der folgenden Abbildung 3.21 sind die Pfade solcher semi-globaler Alignments in der berechneten Tabelle bildlich dargestellt.

Um also insgesamt ein optimales semi-globales 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):

$$S(i,j) = \begin{cases} 0 & \text{für } (i=0) \vee (j=0), \\ \max \left\{ \begin{array}{lcl} S(i-1,j-1) & + & w(s_i,t_j), \\ S(i-1,j) & + & w(s_i,-), \\ S(i,j-1) & + & w(-,t_j) \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

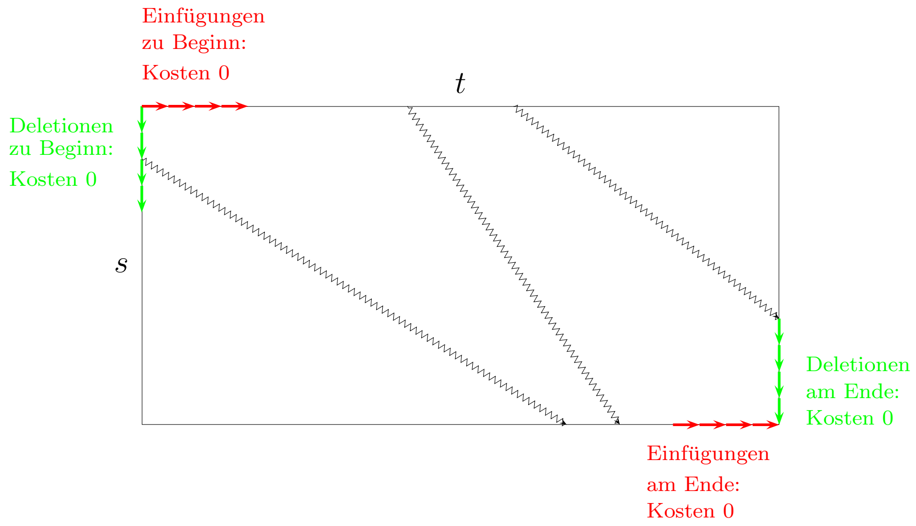


Abbildung 3.21: Skizze: Semi-Globale Alignments in der Ähnlichkeits-Tabelle

Natürlich lässt sich auch für semi-globale Alignments das Verfahren von Hirschberg zur Platzreduktion anwenden. Die Details seien dem Leser als Übungsaufgabe überlassen. Fassen wir unser Ergebnis noch zusammen.

Theorem 3.23 *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.*

3.3.2 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 3.22 ist ein solches lokales Alignment schematisch dargestellt.

Abbildung 3.22: Skizze: lokales Alignment

Betrachten wir zunächst ein Beispiel, nämlich ein lokales Alignment zwischen den Sequenzen $s = ACGATTATTT$ und $t = TAGTAATCG$, wie es in Abbildung 3.23 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 3.23: 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 lokalen Alignments von zwei Teilwörtern von $s_1 \cdots s_i$ und $t_1 \cdots t_j$. Dann können wir wieder eine Rekursionsgleichung aufstellen:

$$S(i,j) = \begin{cases} 0 & \text{für } (i=0) \vee (j=0), \\ \max \left\{ \begin{array}{lcl} S(i-1,j-1) & + & w(s_i,t_j), \\ S(i-1,j) & + & w(s_i,-), \\ S(i,j-1) & + & w(-,t_j), \\ & 0 & \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

Die Rekursionsgleichung sieht fast so aus, wie im Falle der semi-globalen 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.

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-Waterman bezeichnet.

In der folgenden Abbildung 3.24 ist noch einmal der Pfad, der zu einem lokalen Alignment gehört, innerhalb der Tabelle S schematisch dargestellt.

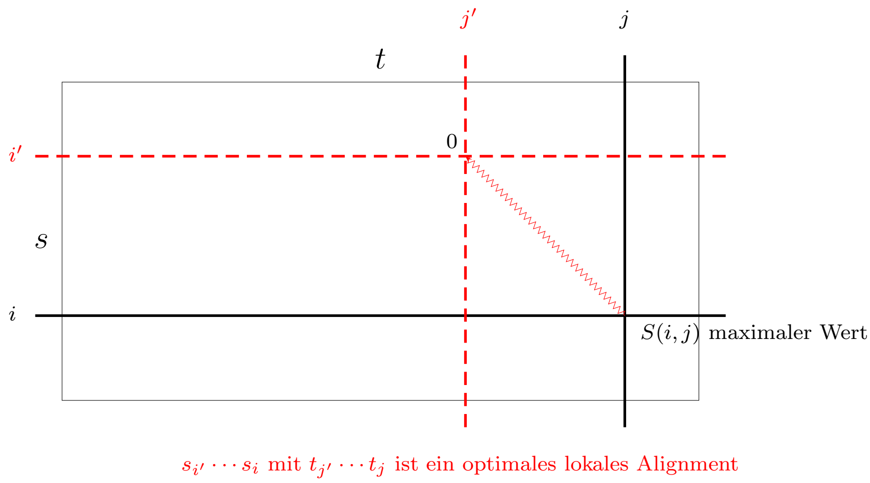


Abbildung 3.24: Skizze: lokales Alignment in der Tabelle

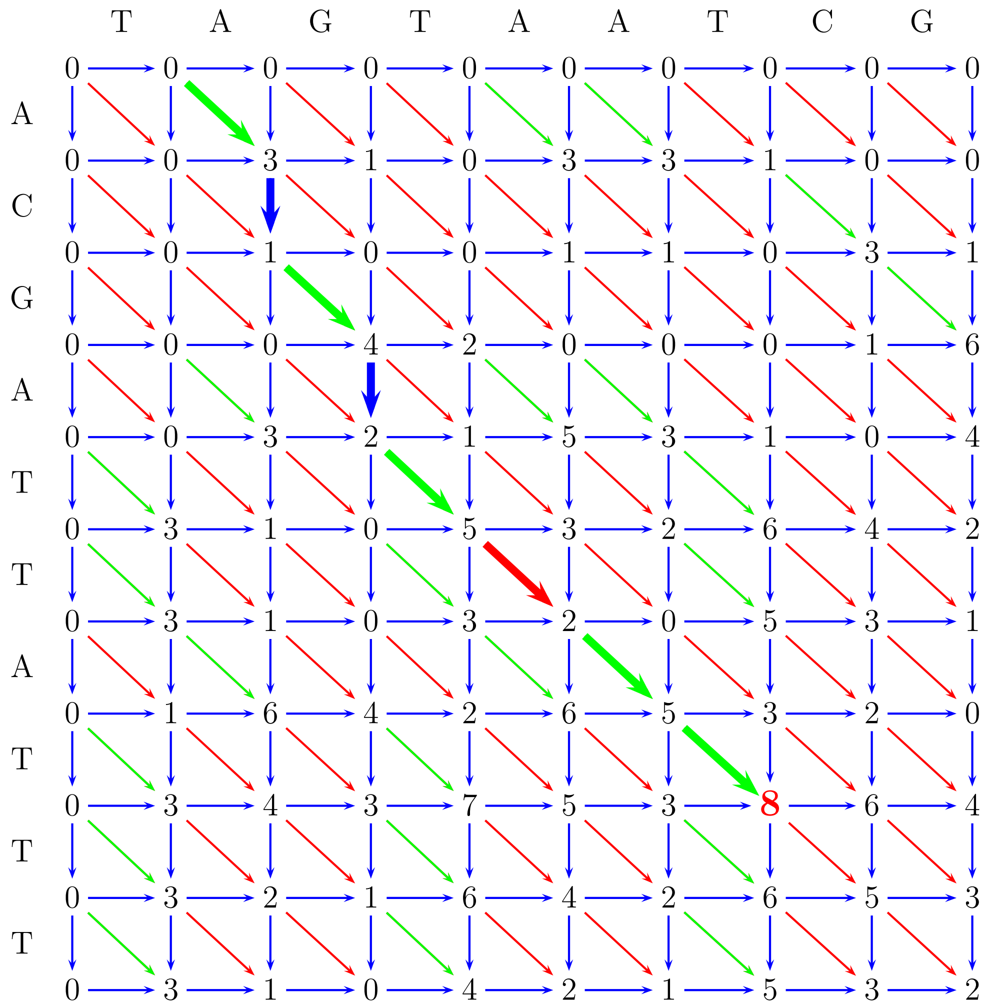
Auch für das lokale paarweise Sequenzen Alignment lässt sich die Methode von Hirschberg zur Platzreduktion anwenden. Wir überlassen es wieder dem Leser sich die genauen Details zu überlegen. Zusammenfassend erhalten wir für lokale Alignments das folgende Ergebnis.

Theorem 3.24 *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 3.25 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 3.26 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.



Match = +3, Indel = −2, Subst = −3

Abbildung 3.25: Beispiel:Tabelle für lokale Alignments zwischen s und t

s:	-	A	C	G	A	T	T	A	T	T	T
t:	T	A	-	G	-	T	A	A	T	C	G

Abbildung 3.26: Beispiel: Ein optimales lokales Alignment zwischen s und t

3.3.3 Lücken-Strafen

Manchmal tauchen in Alignments mittendrin immer wieder lange Lücken auf (siehe Abbildung 3.27). Eine Lücke der Länge ℓ nun mit den Kosten von ℓ Insertionen 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.

				Lücke											
T	A	T	A	_	_	_	_	A	G	C	T	C	T	C	G
T	A	T	A	C	C	G	C	A	G	C	_	_	_	C	G
											Lücke				

Abbildung 3.27: Skizze: Lücken in Alignments

Zur Bestrafung für Lücken verwenden wir eine Lücken-Strafe (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 Lücken-Strafe g sublinear ist, d.h. $g(k' + k'') \leq g(k') + g(k'')$ für alle $k', k'' \in \mathbb{N}_0$. Wir bemerken hier noch, dass wir jetzt Insertionen und Deletionen explizit immer gleich bewerten, unabhängig davon, welche Zeichen gelöscht oder eingefügt werden. In Abbildung 3.28 ist skizziert, wie Funktionen für „vernünftige“ Lücken-Strafen 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.

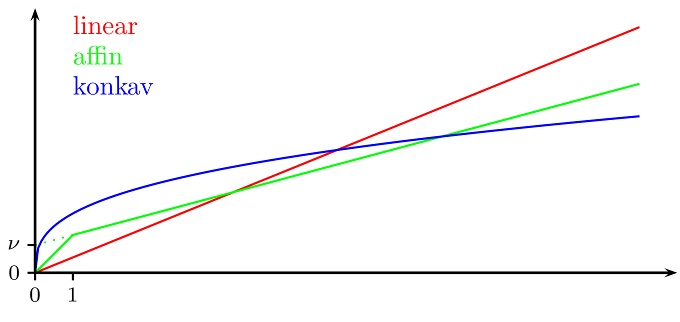


Abbildung 3.28: Skizze: Funktionsgraphen einiger typischer Lücken-Strafen

3.3.4 Allgemeine Lücken-Strafen (Waterman-Smith-Byers)

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 Lücken-Strafen ergeben sich die folgenden Rekursionsgleichungen nach Waterman-Smith-Byers.

$$D(i,j) = \begin{cases} g(i) & \text{für } i = 0, \\ g(j) & \text{für } j = 0, \\ \min\limits_{k} \left\{ \begin{array}{lcl} D(i-1,j-1) & + & w(s_i,t_j), \\ D(i-k,j) & + & g(k), \\ D(i,j-k) & + & g(k) \end{array} \right\} & \text{für } (i>0) \wedge (j>0). \end{cases}$$

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 sublinear ist, d.h. $g(k'+k'') \leq g(k') + g(k'')$ für alle $k', k'' \in \mathbb{N}_0$.

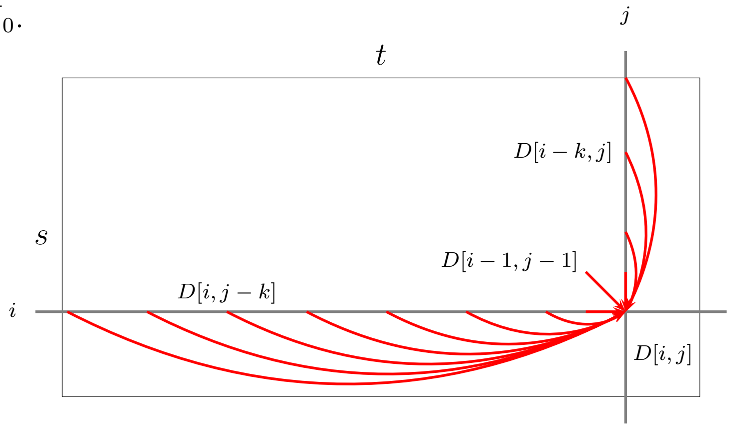


Abbildung 3.29: Skizze: Berechnung optimaler Alignments nach Waterman-Smith-Byers

In der Abbildung 3.29 ist noch einmal schematisch dargestellt, auf welche Werte die Berechnung von $D[i,j]$ zurückgreift.

Die Laufzeit für die Variante von Waterman-Smith-Byers 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 3.25 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen Alignment für s und t mit allgemeinen Lücken-Strafen 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.

3.3.5 Affine Lücken-Strafen (Gotoh)

Da für allgemeine Lücken-Strafen sowohl Laufzeit- als auch Platzbedarf zu hoch sind, schauen wir uns spezielle Lücken-Strafen an, nämlich affine Lücken-Strafen. Solche affinen Lücken-Strafen 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 Lücken-Strafe 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*).

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 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 oder eine Insertion gewesen sein, da aufgrund der Dreiecksungleichung eine Insertion nicht auf eine Deletion folgen kann. Im ersten Fall wird eine Lücke eröffnet (Kosten $\nu + \mu$), im anderen Fall eine fortgesetzt (Kosten μ). Somit erhalten wir:

$$E[i,j] = \min\{G[i,j-1] + \mu + \nu, E[i,j-1] + \mu\}.$$

Dies ist in der folgenden Abbildung noch einmal schematisch dargestellt.

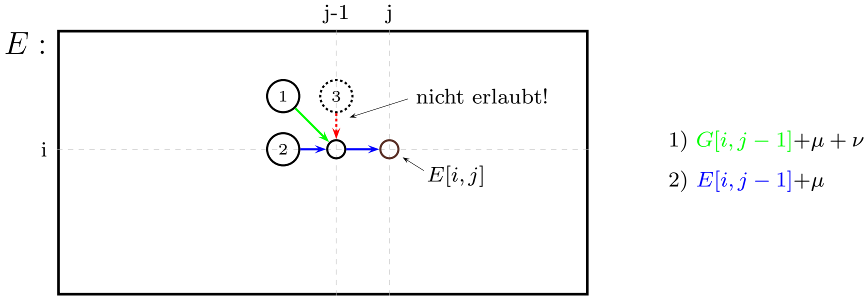


Abbildung 3.30: Skizze: Erweiterung eines Alignments mit einer Insertion

- Betrachten wir jetzt die Tabelle F, d.h. das Alignment endet mit einer Deletion. Dann muss davor eine Substitution oder eine Deletion gewesen sein, da aufgrund der Dreiecksungleichung eine Deletion nicht auf eine Insertion folgen kann. Im ersten Fall wird eine Lücke eröffnet (Kosten $\nu + \mu$), im anderen Fall eine fortgesetzt (Kosten μ). Somit erhalten wir:

$$F[i,j] = \min\{G[i-1,j] + \mu + \nu, F[i-1,j] + \mu\}.$$

Dies ist in der folgenden Abbildung noch einmal schematisch dargestellt.

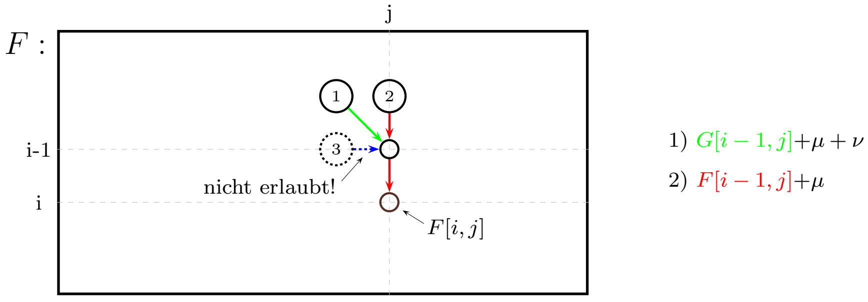


Abbildung 3.31: Skizze: Erweiterung eines Alignments mit einer Deletion

- 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:

$$G[i,j] = \min \left\{ \begin{array}{l} G[i-1,j-1] + w(s_i,t_j), \\ E[i-1,j-1] + w(s_i,t_j), \\ F[i-1,j-1] + w(s_i,t_j) \end{array} \right\}.$$

Dies ist in der folgenden Abbildung noch einmal schematisch dargestellt.

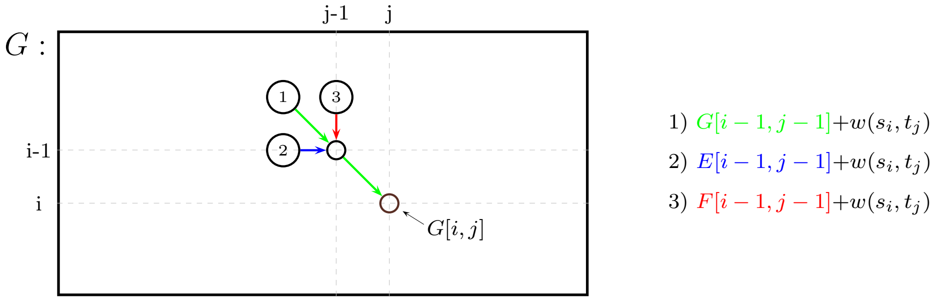


Abbildung 3.32: Skizze: Erweiterung eines Alignments mit einer Substitution

- 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 auch 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 $j > 0$:

$$\begin{aligned}
E[0, j] &= j * \mu + \nu \\
E[i, 0] &= \infty \\
E[0, 0] &= \infty \\
\\
F[i, 0] &= i * \mu + \nu \\
F[0, j] &= \infty \\
F[0, j] &= \infty \\
\\
G[i, 0] &= \infty \\
G[0, j] &= \infty \\
G[0, 0] &= 0
\end{aligned}$$

Auch hier kann man wieder die Methode von Hirschberg zur Platzreduktion anwenden. Halten wir noch das Ergebnis fest.

Theorem 3.26 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen Alignment für s und t mit affinen Lücken-Strafen lässt sich in Zeit $O(nm)$ mit Platzbedarf $O(\min(n, m))$ berechnen.*

3.3.6 Konkave Lücken-Strafen

Zum Abschuss wollen wir noch kurz konkave Lücken-Strafen erwähnen. Eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}$ heißt *konkav*, wenn gilt:

$$\forall n \in \mathbb{N} : \; f(x) - f(x-1) \leq f(x+1) - f(x).$$

Anschaulich bedeutet dies, dass die Funktion immer langsamer wächst. 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.

Wir wollen an dieser Stelle nicht näher auf konkave Funktionen als Lücken-Strafen eingehen, sondern nur ein Ergebnis festhalten.

Theorem 3.27 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Ein optimales globales paarweises Sequenzen Alignment für s und t mit konkaven Lücken-Strafen lässt sich in Zeit $O(nm\log(n))$ berechnen.*

Wir merken noch an, dass gewisse konkave Funktionen die Berechnung sogar in Zeit $O(nm)$ zulassen.

3.4 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.

3.4.1 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.

Geg.: $s, t \in \Sigma^*$ mit $|s| = n$, $|t| = m$.
Ges.: Berechne $d(s, t')$ für alle $t' \sqsubseteq t$.

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 $O(nm)$ ist, 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 $O(m)$ 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 jeden Knoten, den 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 $(v, w) \in T_t$ verbunden sind (w ist also ein Kind von v) und die beiden zugehörigen Suffixe 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 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 3.33 dargestellt.

Damit ergibt sich unmittelbar der folgende Algorithmus, der in Abbildung 3.34 angegeben ist. Hierbei berechnet $\text{compute\_table}(D, s, t', k, \ell)$ 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.

Die Laufzeit beträgt dann $O(m)$ für den reinen DFS-Durchlauf, da der Suffix-Baum ja $O(|t|) = O(m)$ Knoten besitzt. Für die Berechnung der Tabellenerweiterungen

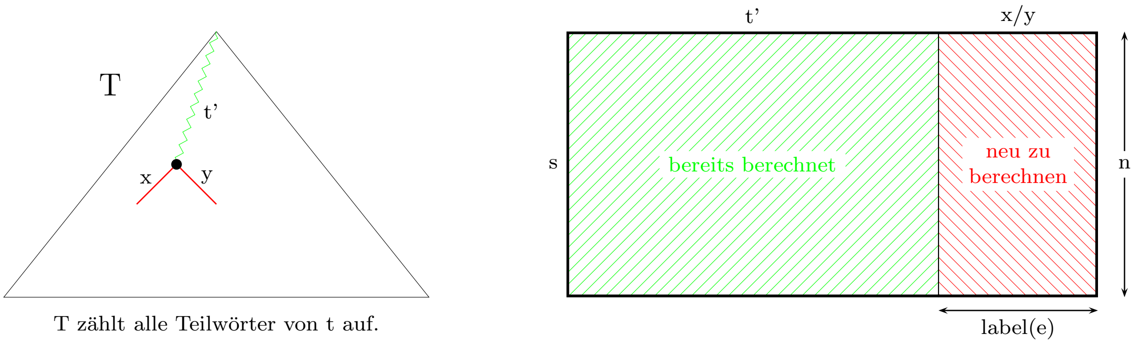


Abbildung 3.33: Skizze: Hybrides Verfahren für One-Against-All

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 $T(n, m)$:

$$\begin{aligned} T(n,m) &= O(m) + O\left(\sum_{(v,w)\in E(T_t)} n \cdot |\text{label}(v,w)| \right) \\ &= O(m) + O\left(n \cdot \underbrace{\sum_{(v,w)\in E(T_t)} |\text{label}(v,w)|}_{=:\text{size}(\text{T}_t)} \right) \\ &= O(m + n \cdot \text{size}(T_t)). \end{aligned}$$

Hierbei ist die Größe size eines Suffixbaumes durch die Summe aller Längen der Kantenlabels 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 3.28 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$. Alle optimalen Alignment-Distanzen für s und t' mit $t \sqsubseteq t$ lassen sich mit Hilfe des hybriden Verfahrens in Zeit $O(n \cdot \text{size}(T_t)) \subseteq O(nm^2)$ bestimmen.*

```
DFS (node v, char s[], char t[])
{
    for all ((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));
    }
}
```

Abbildung 3.34: Algorithmus: Hybrides Verfahren für One-Against-All

Experimente mit realistischen (biologischen) Daten haben ergeben, dass $size(T_t)$ in der Regel ungefähr $m^2/10$ entspricht.

3.4.2 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.

Geg.: $s, t \in \Sigma^*$ mit $|s| = n$ sowie $|t| = m$ und $\vartheta \in \mathbb{R}_+$.
Ges.: 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 Teilwörter 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 $O(n)$ Suffixe von s und $O(m)$ 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 label(v, w) bzw.

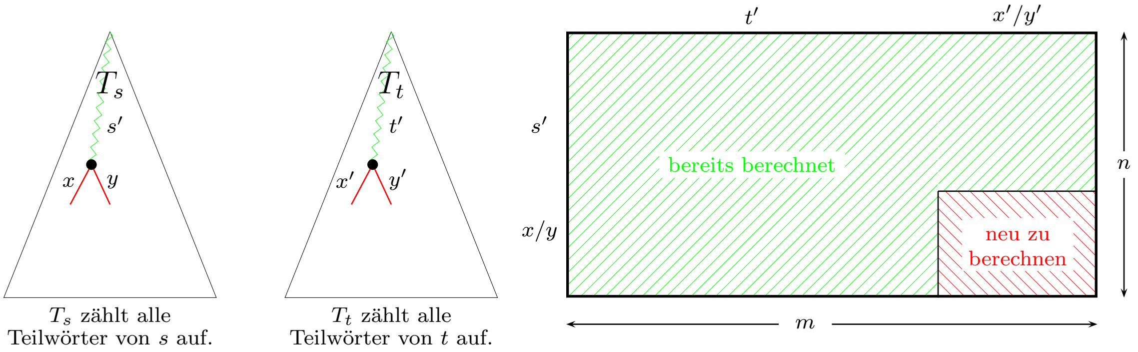


Abbildung 3.35: Skizze: Hybrides Verfahren für All-Against-All

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 ($|\text{label}(v, w)|$ Zeilen sowie $|\text{label}(v', w')|$ Spalten) neu berechnen. Dies ist schematisch in Abbildung 3.35 dargestellt.

Damit ergibt sich der in Abbildung 3.36 angegebene Algorithmus, der zu Beginn mit $\text{DFS}_\text{s}(r(T_s), \varepsilon)$ aufgerufen wird. Die Prozedur compute_table$(D, s', t', k, k', \ell, \ell')$ berechnet die Tabelle mit den optimalen Alignment-Distanzen von $s' = s_1 \cdots s_\ell$ mit $t' = t_1 \cdots t_{\ell'}$, wobei einige Einträge in $D(i, j)$ schon berechnet sind.

Für die Laufzeit $T(n, m)$ gilt (wobei der Term $O(nm)$ vom reinen parallelen Durchlaufen der Suffix-Bäume mit der Tiefensuche herrührt):

$$\begin{aligned}
T(n,m) &= O(nm) + O\left(\sum_{(v,w)\in E(T_s)} \sum_{(v',w')\in E(T_t)} |\text{label}(v,w)| \cdot |\text{label}(v',w')| \right) \\
&= O(nm) + O\left(\sum_{(v,w)\in E(T_s)} |\text{label}(v,w)| \sum_{(v',w')\in E(T_t)} |\text{label}(v',w')| \right) \\
&= O(n+m) + O(\text{size}(T_s) \cdot \text{size}(T_t)).
\end{aligned}$$

Theorem 3.29 *Seien $s, t \in \Sigma^*$ mit $n = |s|$ und $m = |t|$ sowie $\vartheta > 0$. 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 $O(\text{size}(T_s) \cdot \text{size}(T_t) + D)$ bestimmen, wobei D die Anzahl der Paare (s', t') mit $d(s', t') \leq \vartheta$ ist.*

```
DFS_s (node v, char s[])
{
    for all ((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))
    }
}

DFS_t (node w, char s[], char t[], int k, int k')
{
    for all ((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'));
    }
}
```

Abbildung 3.36: Algorithmus: Hybrides Verfahren für All-Against-All

3.5 Datenbanksuche

In diesem Abschnitt wollen wir noch kurz die verwendeten algorithmischen Ideen der beiden gebräuchlichsten Tools zur Suche in großen Sequenz-Datenbanken vorstellen: FASTA und BLAST. Wir stellen hier nur jeweils die Hauptvariante vor. Von beiden Verfahren gibt es zahlreiche abgeleitete Varianten.

3.5.1 FASTA (FAST All oder FAST Alignments)

Im Folgenden suchen wir nach einer Sequenz s in einer Datenbank t.

(1) Wir wählen zuerst eine Konstante ktup in Abhängigkeit vom Inhalt der Datenbank:

$$k := \text{ktup} = \begin{cases} 6 & \text{für DNS} \\ 2 & \text{für Proteine} \end{cases}$$

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 Sequenz auftreten. Diese kurzen Treffer von $s_i \cdots s_{i+k-1}$ werden *Hot Spots* genannt. Diese Hot Spots sind

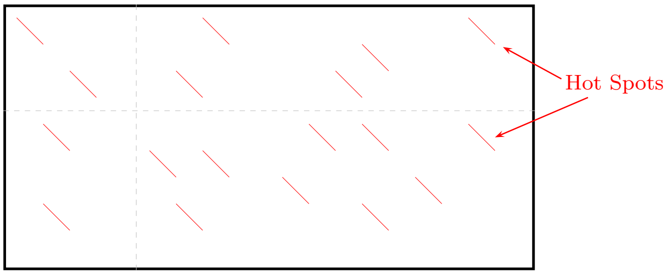


Abbildung 3.37: Skizze: Hot Spots

in der Abbildung 3.37 noch einmal in der (nicht wirklich berechneten) Tabelle für die Alignment-Distanzen visualisiert.

(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 3.38 noch einmal illustriert. Dazu bewertet man die Hot-Spots positiv und die Lücken negativ, wobei

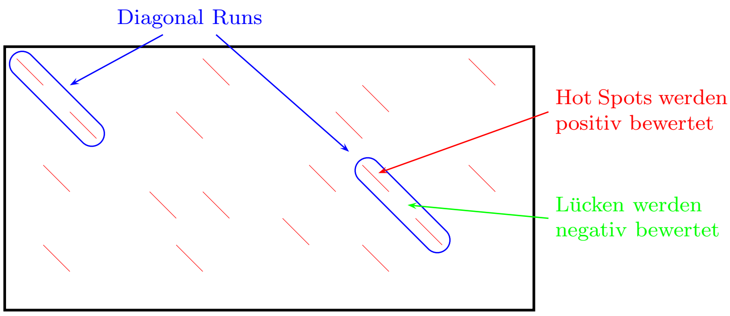


Abbildung 3.38: Skizze: Diagonal Runs

längere Lücken einen kleineren (negativen!) Wert erhalten als kurze 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. 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.

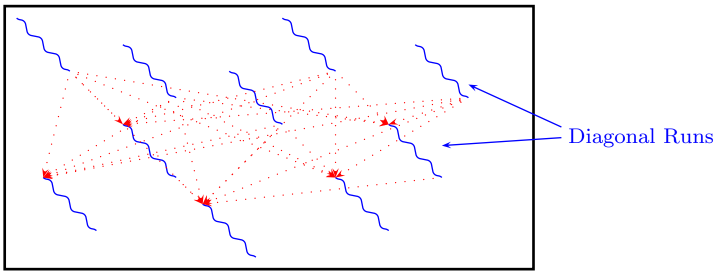


Abbildung 3.39: Skizze: Graph aus Diagonal Runs

(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 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 Graph ist noch einmal in Abbildung 3.39 illustriert. Der so entstandene Graph ist azyklisch (d.h. kreisfrei) und wir können darin wieder sehr einfach gewichtsmaximale Pfade suchen.

(4) Für die gewichtsmaximalen Pfade aus Diagonal Runs berechnen wir jetzt noch 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 3.40 bildlich dargestellt.

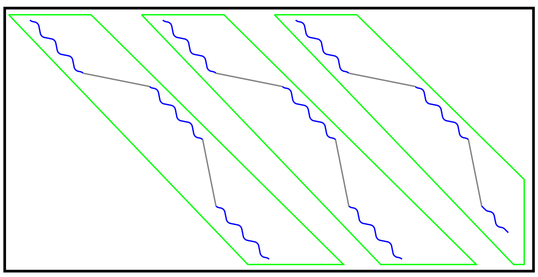


Abbildung 3.40: Skizze: Optimale Alignments um die Pfade aus Diagonal Runs

3.5.2 BLAST (Basic Local Alignment Search Tool)

Wieder nehmen wir an, dass wir nach einer Sequenz s in einer Datenbank t suchen.

(1) Zuerst konstruieren wir alle Wörter aus Σ^k und testen, 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 eine Suchmustermenge M auf Hierbei wird k relativ klein gewählt:
$$k \begin{cases} \in [3:5] & \text{für Proteine,} \\ \approx 12 & \text{für DNS.} \end{cases}$$
Diese Menge M 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 weitermachen können.

(2) Jetzt suchen wir in der Datenbank t nach Wörtern aus M, z.B. mit Hilfe des Algorithmus von *Aho-Corasick*, und merken uns die Treffer.

(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 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*.

(4) Solche Sequence Pairs sind Startwerte für mögliche gute lokale Alignments von s und t. Zum Schluss erweitern wir Sequence Pairs zu einem optimalen lokalen Alignment und geben diese als Treffer zusammen mit dem erzielten Score aus.

3.6 Konstruktion von Ähnlichkeitsmaßen

In diesem Abschnitt wollen wir einen kurzen Einblick geben, wie man aus experimentellen biologischen Daten gute Kostenfunktionen für Ähnlichkeitsmaße konstruieren kann.

3.6.1 Maximum-Likelihood-Prinzip

Zuerst erläutern wir kurz das so genannte *Maximum-Likelihood-Prinzip*, dass hinter der Entscheidung für ein bestimmtes Modell aufgrund experimenteller Daten steckt.

Definition 3.30 (Maximum-Likelihood-Prinzip) *Gibt es mehrere Modelle zur Erklärung eines Sachverhalts (experimentell ermittelte Daten), so wählt man das Modell, das für die ermittelten Daten die größte Wahrscheinlichkeit vorhersagt.*

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 Alignments (s, t) für $s, t \in \Sigma^n$ gilt dann in diesem Modell:

$$\mathrm{Ws}((s,t) \mid R) = \prod_{i=1}^{n} p_{s_i} \prod_{i=1}^{n} p_{t_i}.$$

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 erklärbar ist, nämlich mit Hilfe von Mutationen. Hier gilt für die Wahrscheinlichkeit für ein Alignment (s, t)

$$\mathrm{Ws}((s,t) \mid M) = \prod_{i=1}^{n} (p_{s_i} \cdot p_{s_i,t_i}).$$

Hierbei bezeichnet $p_{a,b}$ die Wahrscheinlichkeit, dass in einer Sequenz das Symbol a zu einem Symbol b mutiert. Wir nehmen an, dass $p_{a,b} = p_{b,a}$ gilt.

Vergleichen wir jetzt beide Modelle, d.h. wir dividieren die Wahrscheinlichkeiten für ein gegebenes Alignment (s, t) mit $s, t \in \Sigma^n$:

$$\frac{\mathrm{Ws}((s,t) \mid M)}{\mathrm{Ws}((s,t) \mid R)} = \prod_{i=1}^{n} \frac{p_{s_i} \cdot p_{s_i,t_i}}{p_{s_i} \cdot p_{t_i}} \lesseqgtr 1.$$

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:

$$\mathrm{Score}(s,t) := \sum_{i=1}^{n} \underbrace{\log\left(\frac{p_{s_i} \cdot p_{s_i,t_i}}{p_{s_i} \cdot p_{t_i}}\right)}_{Kostenfunktion}$$

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:

$$w(a,b) := \log\left(\frac{p_a \cdot p_{a,b}}{p_a \cdot p_b}\right).$$

Es bleibt die Frage, wie man p_a bzw. $p_{a,b}$ für $a, b \in \Sigma$ erhält?

3.6.2 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 sicher sein, dass die hier vorgekommenen Mutationen wirklich passiert sind. Solche Listen kann man über mehrfache Sequenzen Alignments von gut konservierten Regionen ähnlicher Spezies erhalten. Mit $n_{a,b}$ bezeichnen wir die Paare $\{a, b\}$ in der Liste L und mit n die Anzahl aller Paare in L.

Für p_a mit $a \in \Sigma$ ist es am einfachsten, wenn man hierfür die relative Häufigkeit von a in allen Sequenzen annimmt. Es bleibt insbesondere $p_{a,b}$ zu bestimmen. Die Mutation $a \to b$ ist 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 $\mathrm{Ws}(b \mid a)$. Nach dem Satz von Bayes für bedingte Wahrscheinlichkeiten gilt, dass $\mathrm{Ws}(b \mid a) = \frac{\mathrm{Ws}(a,b)}{\mathrm{Ws}(a)}$, wobei $\mathrm{Ws}(a,b)$ die Wahrscheinlichkeit ist, dass einem Alignment a und b gegenüberstehen. Also gilt:

$$p_{a,b} = \mathrm{Ws}(b \mid a) = \frac{\mathrm{Ws}(a,b)}{\mathrm{Ws}(a)} \sim \frac{\frac{n_{a,b}}{n}}{p_a} = \frac{n_{a,b}}{n} \cdot \frac{1}{p_a}.$$

Die letzte Proportionalität folgt daher, dass wir für die Wahrscheinlichkeit $\mathrm{Ws}(a,b)$ annehmen, dass diese durch die relative Häufigkeit von Mutationen ziemlich gut angenähert wird (bis auf einen konstanten Faktor). 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:

$$\begin{aligned} p_{a,b} &:= \frac{n_{a,b}}{n} \cdot \frac{1}{p_a} \cdot \frac{1}{100}, \\ p_{a,a} &:= 1 - \sum_{\substack{b \in \Sigma \\ b \neq a}} p_{a,b}. \end{aligned}$$

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

$$\begin{aligned}
\sum_{a\in\Sigma} p_a \cdot p_{a,a} &= \sum_{a\in\Sigma} p_a \left(1 - \sum_{\substack{b\in\Sigma\\ b\neq a}} p_{a,b}\right) \\
&= \underbrace{\sum_{a\in\Sigma} p_a}_{=1} - \sum_{a\in\Sigma}\sum_{\substack{b\in\Sigma\\ b\neq a}} p_a \cdot p_{a,b} \\
&= 1 - \sum_{a\in\Sigma}\sum_{\substack{b\in\Sigma\\ b\neq a}} \frac{n_{ab}}{n} \cdot \frac{1}{p_a} \cdot \frac{1}{100} \cdot p_a \\
&= 1 - \frac{1}{100n} \cdot \underbrace{\sum_{a\in\Sigma}\sum_{\substack{b\in\Sigma\\ b\neq a}} n_{ab}}_{=n} \\
&= 0{,}99.
\end{aligned}$$

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

$$w(a,b) = \log\left(\frac{p_a \cdot p_{a,b}}{p_a \cdot b}\right) = \log\left(\frac{p_a \cdot \frac{1}{p_a} \cdot \frac{n_{ab}}{n} \cdot \frac{1}{100}}{p_a \cdot p_b}\right) = \log\left(\frac{n_{ab}}{100 \cdot n \cdot p_a \cdot p_b}\right)$$

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 Matrix $P = (p_{a,b})_{a,b\in\Sigma}$ durch $P^k = (p^{(k)}_{a,b})_{a,b\in\Sigma}$ ersetzt und dann entsprechend in die Kostenfunktion einsetzt. 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 an dieser Stelle jedoch nicht eingehen wollen.

Mehrfaches Sequenzen Alignment

4.1 Distanz- und Ähnlichkeitsmaße

In diesem Kapitel wollen wir uns mit der gleichzeitigen Ausrichtungen mehrerer (mehr als zwei) Sequenzen beschäftigen.

4.1.1 Mehrfache Alignments

Zuerst müssen wir mehrfache Alignments sowie deren Distanz bzw. Ähnlichkeit analog wie im Falle paarweiser Sequenzen Alignments definieren.

Definition 4.1 *Seien* $s_1, \ldots, s_k \in \Sigma^*$. *Eine Folge* $\overline{s}_1, \ldots, \overline{s}_k$ *heißt* mehrfaches Sequenzen Alignment (MSA) *für die Sequenz* $s_1, \ldots, s_k$, *wenn gilt:*

- $|\overline{s}_1| = \cdots = |\overline{s}_k| = n$,
- $\overline{s}_{1,i} = \overline{s}_{2,i} = \cdots = \overline{s}_{k,i} \quad \Rightarrow \quad \overline{s}_{1,i} \neq -$,
- $\overline{s}_j|_\Sigma = s_j$ *für alle* $j \in [1:k]$.

4.1.2 Alignment-Distanz und -Ähnlichkeit

Definition 4.2 *Sei* $w : \overline{\Sigma}^k \to \mathbb{R}_+$ *eine Kostenfunktion für ein Distanzmaß bzw. Ähnlichkeitsmaß eines* k*-fachen Sequenzen Alignments* $(\overline{s}_1, \ldots, \overline{s}_k)$ *für* $s_1, \ldots, s_k$, *dann ist*

$$w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{i=1}^{n} w(\overline{s}_{1,i}, \ldots, \overline{s}_{k,i})$$

mit $n = |\overline{s}_1|$ *die Distanz bzw. Ähnlichkeit des Alignments* $(\overline{s}_1, \ldots, \overline{s}_k)$ *für* $s_1, \ldots, s_k$.

Wie im Falle paarweiser Sequenzen Alignments sollte die Kostenfunktion wieder den wesentlichen Bedingungen einer Metrik entsprechen. Die Kostenfunktion w sollte wiederum symmetrisch sein:

$$w(a_1, \ldots, a_k) = w(a_{\pi_1}, \ldots, a_{\pi_k})$$

für eine beliebige Permutation $\pi = (\pi_1, \ldots, \pi_k) \in S([1:k])$. Weiter sollte die Dreiecks-Ungleichung gelten:

$$\begin{aligned} & w(a_1, \ldots, a_i, \ldots, a_j, \ldots, a_k) \\ & \qquad \leq \; w(a_1, \ldots, a_i, \ldots, x, \ldots, a_k) + w(a_1, \ldots, x, \ldots, a_j, \ldots a_k). \end{aligned}$$

Weiterhin sollte auch wieder die Definitheit gelten:

$$w(a_1, \ldots, a_k) = 0 \quad \Leftrightarrow \quad a_1 = \ldots = a_k.$$

Eine Standardkostenfunktion ist die so genannte *Sum-of-Pairs-Funktion*:

$$w(a_1, \ldots, a_k) = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \tilde{w}(a_i, a_j),$$

wobei $\tilde{w} : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ eine gewöhnliche Kostenfunktion für Alignment- oder Ähnlichkeitsmaße eines paarweisen Alignments ist. Hierbei nehmen wir jedoch an, dass $\tilde{w}(-,-) = 0$.

Dies impliziert den Abstand oder Ähnlichkeit für mehrfache Alignments:

$$w(\overline{s}_1, \ldots, \overline{s}_k) := \sum_{i=1}^{k} \sum_{j=i+1}^{k} \tilde{w}(\overline{s}_i, \overline{s}_j).$$

Definition 4.3 *Ein mehrfaches Sequenzen Alignment* $(\overline{s}_1, \ldots, \overline{s}_k) \in \overline{\Sigma}^n$ *für* $s_1, \ldots, s_k \in \Sigma^*$ *heißt optimal, wenn*

$$w(\overline{s}_1, \ldots, \overline{s}_k) = \min\{w(\overline{t}_1, \ldots, \overline{t}_k) \mid (\overline{t}_1, \ldots, \overline{t}_k) \text{ ist ein MSA für } s_1, \ldots, s_k\}.$$

Dann ist $d_w(s_1, \ldots, s_k) := w(\overline{s}_1, \ldots, \overline{s}_k)$ *die mehrfache Alignment-Distanz bzw. -Ähnlichkeit von* $s_1, \ldots, s_k$.

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-Maß oder kurz SP-Maß bezeichnet.

4.2 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.

4.2.1 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 4.4 *Seien $s_1, \ldots, s_k \in \Sigma^*$. Es gilt für $\vec{x} \in [1 : |s_1|] \times \cdots \times [1 : |s_k|]$:*

$$D[\vec{x}] := \min\{D[\vec{x} - \vec{\eta}] + w(\vec{x} \bullet \vec{\eta}) \mid \vec{\eta} \in [0 : 1]^k \setminus \vec{0}\}.$$

Hierbei ist

$$(x_1, \ldots, x_k) \bullet (\eta_1, \ldots, \eta_n) = (s_{1,x_1} \otimes \eta_1, \ldots, s_{k,x_k} \otimes \eta_k) \text{ mit } \begin{array}{l} a \otimes 0 = - \\ a \otimes 1 = a \end{array} \text{ für } a \in \Sigma.$$

Nun stellt sich noch die Frage, wie die Anfangswerte für $\vec{x} \in [0 : n]^k \setminus [1 : n]^k$ eines solchen mehrfaches Sequenzen Alignments aussehen. Dies wird am Beispiel von drei Sequenzen im folgenden Bild erklärt.

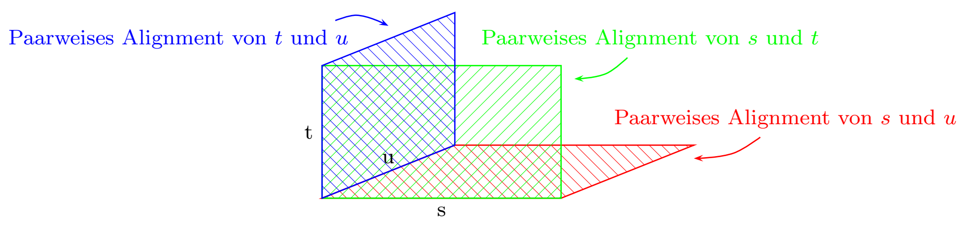


Abbildung 4.1: Skizze: Anfangswerte für ein 3-faches Sequenzen-Alignment

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 eine globales mehrfaches Sequenzen Alignment mit $(i, j, k) \in [1 : n_1] \times [1 : n_2] \times [1 : n_3]$:

$$D[0, 0, 0] = 0,$$

$$
\begin{aligned}
D[i,0,0] &= D[i-1,0,0] + w(s_i^1,-,-),\\
D[0,j,0] &= D[0,j-1,0] + w(-,s_j^2,-),\\
D[0,0,k] &= D[0,0,k-1] + w(-,-,s_k^3),\\
D[i,j,0] &= \min\left\{\begin{array}{c} D[i-1,j,0] + w(s_i^1,-,-),\\ D[i,j-1,0] + w(-,s_j^2,-),\\ D[i-1,j-1,0] + w(s_i^1,s_j^2,-) \end{array}\right\},\\
D[i,0,k] &= \min\left\{\begin{array}{c} D[i-1,0,k] + w(s_i^1,-,-),\\ D[i,0,k-1] + w(-,-,s_k^3),\\ D[i-1,0,k-1] + w(s_i^1,-,s_k^3) \end{array}\right\},\\
D[0,j,k] &= \min\left\{\begin{array}{c} D[0,j-1,k] + w(-,s_j^2,-),\\ D[0,j,k-1] + w(-,-,s_k^3),\\ D[0,j-1,k-1] + w(-,s_j^2,s_k^3) \end{array}\right\},\\
D[i,j,k] &= \min\left\{\begin{array}{c} D[i-1,j,k] + w(s_i^1,-,-),\\ D[i,j-1,k] + w(-,s_j^2,-),\\ D[i,j,k-1] + w(-,-,s_k^3),\\ D[i-1,j-1,k] + w(s_i^1,s_j^2,-),\\ D[i-1,j,k-1] + w(s_i^1,-,s_k^3),\\ D[i,j-1,k-1] + w(-,s_j^2,s_k^3),\\ D[i-1,j-1,k-1] + w(s_i^1,s_j^2,s_k^3) \end{array}\right\}.
\end{aligned}
$$

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 z.B. semi-globale oder lokale mehrfache Alignments sei dem Leser zur Übung überlassen.

4.2.2 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 dass SP-Maß zugrunde legen). Insgesamt ist der Zeitbedarf also $O(k^2 * 2^k * n^k)$.

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. Nach gängiger Lehrmeinung lassen sich $\mathcal{NP}$-harte Probleme nicht in polynomieller Zeit lösen, so dass eine Berechnung optimaler mehrfacher Sequenzen Alignments praktisch nicht effizient lösbar ist.

4.3 Alignment mit Hilfe eines Baumes

Da die exakte Lösung eines mehrfachen Alignments, wie eben angedeutet, in aller Regel sehr schwer lösbar ist, wollen wir uns mit so genannten *Approximationen* beschäftigen. Hierbei konstruieren wir Lösungen, die nur um einen bestimmten Faktor von einer optimalen Lösung entfernt ist.

4.3.1 Mit Bäumen konsistente Alignments

Dazu definieren wir zuerst mit Bäumen konsistente Alignments.

Definition 4.5 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ und sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Sequenzen Alignment für S. Das Paar $(\overline{s}_i, \overline{s}_j)$ heißt* von M induziertes paarweises Alignment*.*

In der Regel werden wir bei induzierten Alignments annehmen, dass Spalten, die nur aus Leerzeichen $-$ bestehen, gestrichen werden, da wir bei paarweisen Sequenzen Alignments solche Spalten verboten haben. Wir bemerken hier noch einmal, dass die Distanz bzw. Ähnlichkeit dadurch nicht verändert wird, da wir hier für die Kostenfunktion annehmen, dass $w(-,-) = 0$ gilt.

Definition 4.6 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ und sei $T = (S, E)$ ein Baum. Ein mehrfaches Sequenzen Alignment $(\overline{s}_1, \ldots, \overline{s}_k)$ für S ist* konsistent *mit T, wenn jedes induzierte paarweise Sequenzen Alignment $(\overline{s}_i, \overline{s}_j)$ für $(s_i, s_j) \in E$ optimal ist.*

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

Abbildung 4.2: Skizze: mehrfaches Alignment, das mit einem Baum konsistent ist

4.3.2 Effiziente Konstruktion

Lemma 4.7 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ und sei $T = (S, E)$ ein Baum. Ein mehrfaches Sequenzen Alignment für S, das konsistent zu T ist, lässt sich in Zeit $O(kn^2)$ konstruieren, wobei $|s_i| = \Theta(n)$ für $i \in [1 : k]$.*

Beweis: Wir führen den Beweis mittels Induktion über k.

Induktionsanfang ($k = 0, 1, 2$): 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. Dieses wurde in Zeit $O(kn^2)$ konstruiert.

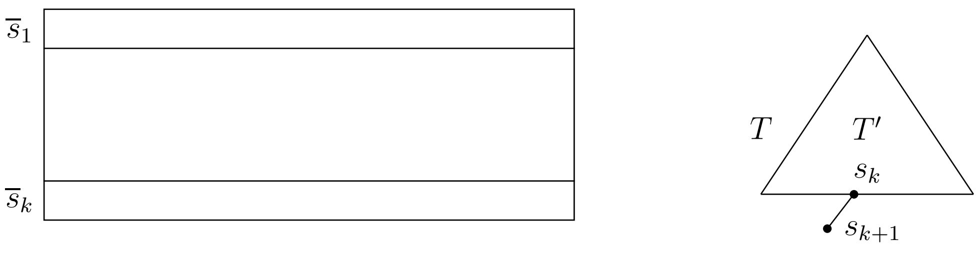


Abbildung 4.3: Skizze: Induktionsvoraussetzung

Wir berechnen ein optimales paarweises Alignment $(\tilde{s}_k, \tilde{s}_{k+1})$ von s_k mit s_{k+1} in Zeit $O(n^2)$. Dann erweitern wir das mehrfaches Sequenzen Alignment um das Alignment $(\tilde{s}_k, \tilde{s}_{k+1})$ wie in der Abbildung angegeben. Dazu müssen wir im Wesentlichen nur

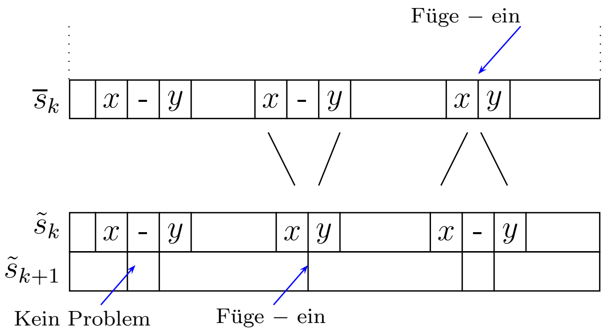


Abbildung 4.4: Skizze: Erweiterung des mehrfaches Sequenzen Alignments

die Zeile $\tilde{s}_{k+1}$ hinzufügen, wobei wir in der Regel sowohl in $\tilde{s}_{k+1}$ als auch im 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. ■

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.

4.4 Center-Star-Approximation

In diesem Abschnitt wollen wir ausgehend von dem im letzten Abschnitt vorgestellten Verfahren zur Konstruktion von mehrfachen Sequenzen Alignments mit Hilfe von Bäumen einen Algorithmus vorstellen, der ein mehrfaches Sequenzen Alignment bestimmter Güte konstruiert.

4.4.1 Die Wahl des Baumes

Bei der Center-Star-Methode besteht die Idee darin, den Baum T so zu wählen, dass er einen Stern darstellt. Also $T \cong \ast$. Das Problem besteht nun darin, welche Sequenz als Zentrum des Sterns gewählt werden soll.

Definition 4.8 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ. Die Sequenz s_c mit $c \in [1 : k]$ heißt* Center-String*, wenn $\sum_{j=1}^{k} d(s_c, s_j)$ minimal ist.*

4.4.2 Approximationsgüte

Sei M_c das mehrfache Sequenzen Alignment, das zu T (dem Stern mit Zentrum s_c) konstruiert ist. Dann bezeichne $D(s_i, s_j)$ den Wert des durch M_c induzierten Alignments für s_i und s_j. Es gilt

$$\begin{aligned} D(s_i, s_j) &\geq d(s_i, s_j), \\ D(s_c, s_j) &= d(s_c, s_j), \\ D(M_c) &= \sum_{i=1}^{k} \sum_{j=i+1}^{k} D(s_i, s_j). \end{aligned}$$

Lemma 4.9 *Es gilt:*

$$D(s_i, s_j) \leq D(s_i, s_c) + D(s_c, s_j) = d(s_i, s_c) + d(s_c, s_j)$$

Beweis: Der Beweis folgt unmittelbar aus der folgenden Abbildung unter Berück-

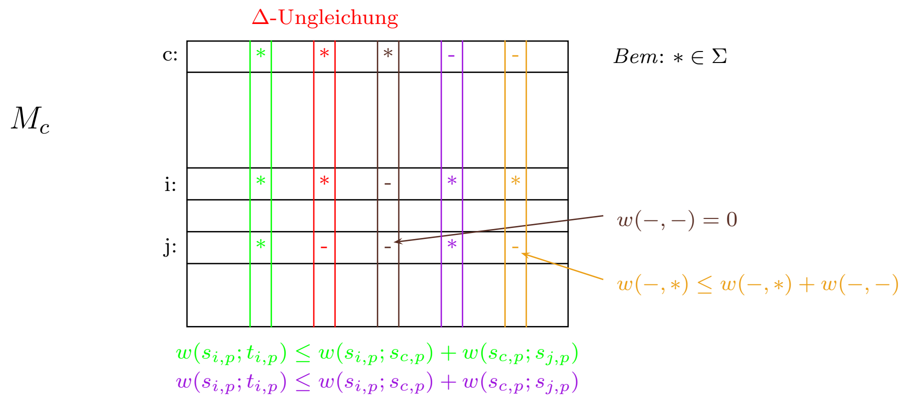


Abbildung 4.5: Skizze: Beweis

sichtigung, dass für w die Dreiecksungleichung gilt. ■

Erinnerung: $D(s, t) = w(\overline{s}, \overline{t}) = \sum_{i=1}^{|\overline{s}|} w(\overline{s_i}, \overline{t_i})$.

Sei M^* ein optimales mehrfaches Sequenzen Alignment für S und sei $D^*(s_i, s_j)$ der Wert des durch M^* induzierten 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 4.10 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen und $T = (S, E)$ ein Stern, dessen Zentrum der Center-String von S ist. Sei M_c ein mehrfaches Sequenzen Alignment, das zu T konsistent ist, und M^* ein optimales mehrfaches Sequenzen Alignment von S. 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 $D^*(s_i, s_j) = D^*(s_j, s_i)$.

Dann gilt für den Quotienten:

$$\begin{aligned} \frac{D(M_c)}{D(M^*)} &= \frac{\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D(s_i, s_j)}{\frac{1}{2} \sum_{i=1}^{k} \sum_{j=1}^{k} D^*(s_i, s_j)} \\ & \quad \text{da } D(s_i, s_i) = 0 \\ &= \frac{\sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(s_i, s_j)}{\sum_{i=1}^{k} \sum_{j=1}^{k} D^*(s_i, s_j)} \\ & \quad \text{mit Lemma 4.9 und } D^*(s_i, s_j) \geq d(s_i, s_j) \\ &\leq \frac{\sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} [d(s_i, s_c) + d(s_c, s_j)]}{\sum_{i=1}^{k} \underbrace{\sum_{j=1}^{k} d(s_i, s_j)}_{\text{minimal für} i=c}} \\ & \quad \text{Nach Wahl von } s_c \\ &\leq \frac{(k-1) \sum_{i=1}^{k} d(s_i, s_c) + (k-1) \sum_{j=1}^{k} d(s_c, s_j)}{\sum_{i=1}^{k} \sum_{j=1}^{k} d(s_c, s_j)} \\ &= \frac{2(k-1)}{k} * \underbrace{\frac{\sum_{i=1}^{k} d(s_i, s_c)}{\sum_{j=1}^{k} d(s_c, s_j)}}_{=1} \end{aligned}$$

$$
\begin{aligned}
&= \frac{2k-2}{k} \\
&= 2 - \frac{2}{k}.
\end{aligned}
$$

■

4.4.3 Laufzeit für Center-Star-Methode

Wie groß ist 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 $O(k^2n^2)$. Für die Konstruktion des mehrfachen Sequenzen Alignments, das konsistent zum Stern mit dem gewählten Center-String als Zentrum ist, benötigen wir nur noch $O(kn^2)$.

Der Gesamtzeitbedarf ist also $O(k^2n^2)$, wobei die meiste Zeit für die Auswahl des Zentrums verbraucht wurde.

Theorem 4.11 *Die Center-Star-Methode liefert eine $(2 - \frac{2}{k})$-Approximation für ein optimales mehrfaches Sequenzen Alignment für k Sequenzen der Länge $\Theta(n)$ in Zeit $O(k^2n^2)$.*

4.4.4 Randomisierte Varianten

Wir wollen im Folgenden zeigen, dass man nur einige Zentren ausprobieren muss und dann bereits der beste der ausprobierten Zentrum schon fast eine 2-Approximation liefert. Wir können also die Laufzeit noch einmal senken.

Theorem 4.12 *Für r sei $C(r)$ die erwartete Anzahl von Sternen, die zufällig gewählt werden müssen, bis das beste mehrfache Sequenzen Alignment, der mit der Center-Star-Methode und den gewählten Zentren, eine $(2 + \frac{1}{r-1})$-Approximation ist. Dann ist $C(r) \leq r$.*

Für den Beweis (und den nächsten Satzes) benötigen wir das folgende Lemma.

Theorem 4.13 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen. Es existieren mehr als $\left\lfloor \frac{k}{r} \right\rfloor$ Sterne mit $M(i) \leq \frac{2r-1}{r-1}M$. Hierbei ist $M(i) := \sum_{j=1}^{k} d(s_i, s_j)$ und $M := \min\{M(i) \mid i \in [1:k]\}$.*

Beweis: Wir berechnen zuerst den Mittelwert von $M(i)$:

$$\begin{aligned}
\frac{1}{k}\sum_{i=1}^{k} M(i) &= \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{mit } d(s_i, s_i) = 0 \\
&= \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} d(s_i, s_j) \\
&\quad \text{mit Hilfe der Dreiecksungleichung} \\
&\leq \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} d(s_i, s_c) + d(s_c, s_j) \\
&\quad \text{da } s_c \text{ ist Zentrum eines optimalen Sterns} \\
&= \frac{1}{k} 2(k-1)M \\
&< 2M
\end{aligned}$$

Wir führen den Beweis des Lemmas mit Hilfe eines Widerspruchsbeweises.

Annahme: Es existieren maximal $\left\lfloor \frac{k}{r} \right\rfloor$ Sterne mit $M(i) \leq \frac{2r-1}{r-1}M$.

Dann gilt:

$$\begin{aligned}
2M &> \frac{1}{k}\sum_{i=1}^{k} M(i) \\
&\quad \text{mit Hilfe der Widerspruchsannahme} \\
&\geq \frac{1}{k}\left(\frac{k}{r}M + \left(k - \frac{k}{r}\right)\frac{2r-1}{r-1}M\right) \\
&= M\left(\frac{1}{r} + \underbrace{\left(1 - \frac{1}{r}\right)}_{=\frac{r-1}{r}}\frac{2r-1}{r-1}\right) \\
&= \frac{M}{r}(1 + 2r - 1) \\
&= 2M.
\end{aligned}$$

Also gilt $2M < 2M$, was offensichtlich der gewünschte Widerspruch ist. ■

Beweis von Satz 4.12: Im Folgenden gelte ohne Beschränkung der Allgemeinheit, dass $M(1) \leq M(2) \leq \cdots M(k)$. Sei $c := \left\lfloor \frac{k}{r} \right\rfloor + 1$, dann gilt aufgrund von Lemma 4.13:

$$M(c) = \varepsilon \cdot M \text{ mit } \varepsilon \in \left[1 : \frac{2r-1}{r-1}\right]$$

Damit gilt:

$$
\begin{aligned}
\frac{D(M_c)}{D(M^*)} &\leq \frac{\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k} D(s_i,s_j)}{\frac{1}{2}\sum_{i=1}^{k}\sum_{j=1}^{k} D^*(s_i,s_j)} \\
&\quad \text{da } D(s_k,s_j)=0 \\
&\leq \frac{\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} D(s_i,s_j)}{\sum_{i=1}^{k}\sum_{j=1}^{k} D^*(s_i,s_j)} \\
&\quad \text{da } D(s_i,s_j) \leq d(s_i,s_c)+d(s_c,s_j) \text{ und } D(s_i,s_j) \geq d(s_i,s_j) \\
&\leq \frac{\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} [d(s_i,s_c)+d(s_c,s_j)]}{\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i,s_j)} \\
&\leq \frac{2\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} d(s_i,s_c)}{\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i,s_j)} \\
&\quad \text{da } M(i) \geq M \text{ für alle } i \text{ und } M(i) \geq \varepsilon M \text{ für } i > c \geq \tfrac{k}{r} \\
&\leq \frac{2(k-1)M(c)}{\sum_{i=1}^{k} M(i)} \\
&\leq \frac{2(k-1)\cdot\varepsilon\cdot M}{\frac{k}{r}\cdot M + \frac{kr-k}{r}\cdot\varepsilon\cdot M} \\
&\leq \frac{2r(k-1)\cdot\varepsilon}{k+k(r-1)\cdot\varepsilon} \\
&\leq \frac{2r(k-1)}{k/\varepsilon+k(r-1)} \\
&\quad \text{da } \varepsilon \leq \tfrac{2r-1}{r-1} \\
&\leq \frac{2r(k-1)}{k\frac{r-1}{2r-1}+k(r-1)} \\
&\leq \frac{2r(k-1)(2r-1)}{k(r-1)+k(r-1)(2r-1)} \\
&\leq \frac{2r(k-1)(2r-1)}{2rk(r-1)} \\
&\leq \frac{(k-1)}{k}\cdot\frac{(2r-1)}{(r-1)} \\
&\leq \frac{(2r-1)}{(r-1)} \\
&\leq 2+\frac{1}{(r-1)}
\end{aligned}
$$

■

Theorem 4.14 *Wählt man p Sterne (d.h. ihre Zentren) zufällig 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* $1-\left(\frac{r-1}{r}\right)^p$.

Beweis: Es gilt mit Lemma 4.13

$$\mathrm{Ws}[\underbrace{\text{schlechter Stern}}_{\substack{\text{liefert keine } (2+\frac{1}{r-1})\text{-}\\ \text{Approximation}}}] \leq \frac{k-\frac{k}{r}}{k} = \frac{r-1}{r}.$$

Daraus folgt sofort

$$\mathrm{Ws}[\text{Es werden p schlechte Sterne gewählt}] \leq \left(\frac{r-1}{r}\right)^p,$$

und somit

$$Ws[\text{Einer der p Sterne liefert } (2+\frac{1}{r-1})\text{Approximation}] \geq 1-\left(\frac{r-1}{r}\right)^p.$$

■

Bsp:	Approximation	Fehler	p
	2,2 $(r=6)$	$< 1\%$	≈ 13
	2,1 $(r=11)$	$< 1\%$	≈ 25

4.5 Konsensus eines mehrfachen Alignments

Nachdem wir nun Möglichkeiten kennen gelernt haben, wie wir mehrfache Sequenzen Alignments effizient konstruieren können, wollen wir uns jetzt damit beschäftigen, wie wir daraus eine Referenz-Sequenz (einen so genannten Konsensus-String) ableiten können.

4.5.1 Konsensus-Fehler und Steiner-Strings

Definition 4.15 *Seien* $S = \{s_1, \ldots, s_k\}$ *Sequenzen über* Σ *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 Sequenz aus* Σ^* *mit minimalem Konsensus-Fehler*

$$E_S(s^*) := \min\{E_S(s') \mid s' \in \Sigma^*\}.$$

Im Allgemeinen ist s^* nicht eindeutig und es gilt $s^* \notin S$. Dennoch kann s^* in einigen wenigen Fällen durchaus eindeutig sein bzw. $s^* \in S$ sein.

Lemma 4.16 *Sei* $S = \{s_1, \ldots, s_k\}$ *und* d *sei eine Metrik. Dann existiert ein* $s' \in S$ *mit*

$$\frac{E_S(s')}{E_S(s^*)} \leq 2 - \frac{2}{k},$$

wobei s^* *ein Steiner-String für* S *ist.*

Beweis: Wähle $s_i \in S$ beliebig, aber fest.

$$\begin{aligned}
E_S(s_i) &= \sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{da } d(s_i, s_i) = 0 \\
&= \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s_i, s_j) \\
&\quad \text{aufgrund der Dreiecksungleichung} \\
&\leq \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s_i, s^*) + \sum_{\substack{j=1 \\ j \neq i}}^{k} d(s^*, s_j) \\
&= (k-1)d(s_i, s^*) + E_S(s^*) - d(s^*, s_i) \\
&= (k-2)d(s_i, s^*) + E_S(s^*).
\end{aligned}$$

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:

$$\begin{aligned} \frac{E_S(s_i)}{E_S(s^*)} &\leq \frac{(k-2)d(s_i, s^*) + E_S(s^*)}{E_S(s^*)} \\ &= 1 + \frac{(k-2)d(s_i, s^*)}{E_S(s^*)} \\ &\leq 1 + \frac{(k-2)d(s_i, s^*)}{k \cdot d(s^*, s_i)} \\ &= 1 + \frac{k-2}{k} \\ &= 2 - \frac{2}{k}. \end{aligned}$$

■

Da für den Center-String s_c gilt, dass $\sum_{j=1}^{k} d(s_u, s_j)$ für $i = c$ minimal wird:

$$\sum_{j=1}^{k} d(s_i, s_j) \geq \sum_{j=1}^{k} d(s_c, s_j) = E_S(s_c).$$

Damit gilt

$$E_S(s_c) \leq E_S(s_i),$$

wobei s_i aus Lemma 4.16 ist. Somit erhalten wir das folgende Korollar:

Korollar 4.17 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ und sei s_c ein Center-String von S und s^* ein optimaler Steiner-String für S, dann gilt*

$$\frac{E_S(s_c)}{E_S(s^*)} \leq 2 - \frac{2}{k}.$$

Theorem 4.18 *Für r sei $C(r)$ die erwartete Anzahl von Sternen (Zentren), die zufällig gewählt werden müssen, bis der beste Konsensus-Fehler bis auf den Faktor $(2+\frac{1}{r-1})$vom Optimum entfernt ist. Dann gilt $C(r) \leq r$.*

Zum Beweis des Satze benötigen wir das folgende Lemma.

Lemma 4.19 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ. Es existieren mehr als $\left\lfloor \frac{k}{r} \right\rfloor$ Sterne mit $E_S(s_i) \leq \frac{2r-1}{r-1} E_S(s^*)$. Hierbei ist $E_S(s_i) := \sum_{j=1}^{k} d(s_i, s_j)$ und s^* ist ein Konsensus-String.*

Beweis: Wir berechnen zuerst den Mittelwert von $E(s_i)$:

$$
\begin{aligned}
\frac{1}{k}\sum_{i=1}^{k} E_S(s_i) &= \frac{1}{k}\sum_{i=1}^{k}\sum_{j=1}^{k} d(s_i, s_j) \\
&\quad \text{mit } d(s_i, s_i) = 0 \\
&= \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} d(s_i, s_j) \\
&\quad \text{mit Hilfe der Dreiecksungleichung} \\
&\leq \frac{1}{k}\sum_{i=1}^{k}\sum_{\substack{j=1\\ j\neq i}}^{k} d(s_i, s^*) + d(s^*, s_j) \\
&\quad \text{da } s^* \text{ ein optimaler Steiner-String} \\
&= \frac{1}{k} 2(k-1) E_S(s^*) \\
&< 2E_S(s^*)
\end{aligned}
$$

Wir führen den Beweis des Lemmas mit Hilfe eines Widerspruchsbeweises.

Annahme: Es existieren maximal $\left\lfloor \frac{k}{r} \right\rfloor$ Sterne mit $E_S(s_i) \leq \frac{2r-1}{r-1} E_S(s^*)$.

Dann gilt:

$$
\begin{aligned}
2E_S(s^*) &> \frac{1}{k}\sum_{i=1}^{k} E_S(S_i) \\
&\quad \text{mit Hilfe der Widerspruchsannahme} \\
&\geq \frac{1}{k}\left(\frac{k}{r} E_S(s^*) + \left(k - \frac{k}{r}\right)\frac{2r-1}{r-1} E_S(s^*)\right)
\end{aligned}
$$

$$\begin{aligned} &= E_S(s^*)\left(\frac{1}{r} + \underbrace{(1-\frac{1}{r})}_{=\frac{r-1}{r}}\frac{2r-1}{r-1}\right) \\ &= \frac{E_S(s^*)}{r}(1+2r-1) \\ &= 2E_S(s^*). \end{aligned}$$

Also gilt $2E_S(s^*) < 2E_S(s^*)$, was offensichtlich der gewünschte Widerspruch ist. ∎

Beweis von Satz 4.18: Es gelte ohne Beschränkung der Allgemeinheit

$$E_S(s_1) \leq E_S(s_2) \leq \cdots \leq E_S(s_k).$$

Mit $c = \lfloor \frac{k}{r} \rfloor + 1$ gilt $E_S(S_c) \leq \frac{2r-1}{r-1} \cdot E_S(s^*)$. ∎

Damit stellen Steiner-Strings also eine Möglichkeit dar, für eine Folge von Sequenzen eine Referenz-Sequenz zu generieren.

4.5.2 Alignment-Fehler und Konsensus-String

Jetzt stellen wir eine weitere Methode vor, die auf mehrfachen Sequenzen Alignments basiert.

Definition 4.20 *Sei $M = (\overline{s}_1, \ldots, \overline{s}_k)$ ein mehrfaches Sequenzen Alignment für $S = \{s_1, \ldots, s_k\}$. Das* Konsensus-Zeichen *an der Position i ist das Zeichen $x \in \overline{\Sigma}$ mit*

$$\sum_{j=1}^{k} w(x, \overline{s}_{j,i}) = \min\left\{\sum_{j=1}^{k} w(a, \overline{s}_{j,i}) \ : \ a \in \overline{\Sigma}\right\} =: \ \delta_M(i).$$

Zur Erinnerung: $w : \overline{\Sigma} \times \overline{\Sigma} \to \mathbb{R}_+$ war eine Kostenfunktion, bei der für mehrfache Sequenzen Alignments $w(-,-) = 0$ gilt.

Definition 4.21 *Der* Konsensus-String $\mathcal{S}_M$ *eines mehrfachen Sequenzen Alignments M für S ist $\mathcal{S}_M := s_1 \cdots s_m$, wobei s_i das Konsensus-Zeichen an der Position i ist.*

Definition 4.22 *Sei* $S = \{s_1, \ldots, s_k\}$ *eine Menge von Sequenzen über* Σ *und sei* $M = (\overline{s}_1, \ldots, \overline{s}_k)$ *ein mehrfaches Sequenzen Alignment für* S *mit* $n = |\overline{s}_1| = \cdots = |\overline{s}_k|$, *dann ist der* Alignment-Fehler *einer Sequenz* $s \in \Sigma^n$ *definiert als*

$$E_M(s) = \sum_{j=1}^{k} \sum_{i=1}^{n} w(s_i, \overline{s}_{j,i}),$$

wobei w *wieder die zugrunde liegende Kostenfunktion ist. Speziell gilt dann*

$$E_M(\mathcal{S}_M) = \sum_{i=1}^{m} \delta_M(i).$$

Definition 4.23 *Sei* $S = \{s_1, \ldots, s_k\}$ *eine Menge von Sequenzen. Das optimale* Konsensus-MSA *für* S *ist ein mehrfaches Sequenzen Alignment* M *für* S *mit minimalem Alignment-Fehler*

4.5.3 Beziehung zwischen Steiner-String und Konsensus-String

Wir haben jetzt mehrere Definitionen für einen *Konsensus-String* kennen gelernt:

- Steiner-String (ohne Mehrfaches Sequenzen Alignment!);
- Konsensus-String für ein Mehrfaches Sequenzen Alignment M (die Optimalität wird hierbei bezüglich minimalem Alignment-Fehler definiert);
- Konsensus-String für ein Mehrfaches Sequenzen Alignment M (die Optimalität wird hierbei bezüglich des Sum-of-Pairs-Maßes definiert).

Theorem 4.24 *i) Sei* s' *der Konsensus-String eines optimalen Konsensus-MSA für* S, *dann ist* $s'|_\Sigma$ *ein optimaler Steiner-String für* S.
ii) Ist M *ein mehrfaches Sequenzen Alignment für* $S \cup \{s^*\}$, *das konsistent zu einem Stern mit Zentrum* s^* *ist, dann ist* M *ohne die Zeile für* s^* *ein optimales Konsensus-MSA, wenn* s^* *ein optimaler Steiner-String für* S *ist.*

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 $\mathcal{S}_M$ der Konsensus-String für M.

Für das induzierte paarweise Alignment von $\mathcal{S}_M$ mit $\overline{s}_j$ gilt:

$$D(\mathcal{S}_M, \overline{s}_j) \;\geq\; d(\mathcal{S}_M, s_j)$$

Also gilt

$$\begin{aligned}
E_M(\mathcal{S}_M) &= \sum_{i=1}^{n} \delta_M(i) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{k} w(\mathcal{S}_{M,i}, \overline{s}_{j,i}) \\
&= \sum_{j=1}^{k} \sum_{i=1}^{n} w(\mathcal{S}_{M,i}, \overline{s}_{j,i}) \\
&= \sum_{j=1}^{k} D(\mathcal{S}_M, \overline{s}_j) \\
&\geq \sum_{j=1}^{k} d(\mathcal{S}_M, s_j) \\
&= E_S(\mathcal{S}_M) \\
&\geq E_S(s^*).
\end{aligned}$$

Hierbei ist s^* ein optimaler Steiner-String für S. Prinzipiell gilt also, dass der Alignment-Fehler des Konsensus-Strings für M mindestens so groß ist wie dessen Konsensus-Fehler.

Sei jetzt M^* ein mehrfaches Sequenzen Alignment für $S \cup \{s^*\}$, das konsistent zu einem Stern mit Zentrum s^* ist, wobei s^* ein optimaler Steiner-String für S ist.

Für das induzierte Alignment $\overline{s^*}$ mit $\overline{s}_j$ gilt:

$$D(\overline{s^*}, \overline{s}_j) = d(s^*, s_j), \tag{4.1}$$

da M konsistent zu einem Stern mit s^* als Zentrum ist.

Sei M das mehrfache Sequenzen Alignment, das aus M^* durch Streichen von $\overline{s^*}$ entsteht. Mit $n = |\overline{s^*}|$ gilt:

$$\begin{aligned}
E_M(\overline{s^*}) &= \sum_{i=1}^{n} \sum_{j=1}^{k} w(\overline{s^*}_i, \overline{s}_{j,i}) \\
&= \sum_{j=1}^{k} \underbrace{\sum_{i=1}^{n} w(\overline{s^*}_i, \overline{s}_{j,i})}_{D_{M^*}(s^*, s_j)}
\end{aligned}$$

$$\begin{aligned} &= \sum_{j=1}^{k} D_{M^*}(\overline{s^*}, s_j) \\ &\quad \text{mit Hilfe der Gleichung 4.1} \\ &= \sum_{j=1}^{k} d(s^*, s_j) \\ &= E_S(s^*). \end{aligned}$$

Es gibt also ein mehrfaches Sequenzen Alignment M für S, dessen Alignment-Fehler des zugehörigen Konsensus-Strings gleich dem Konsensus-Fehler von S ist.

Damit folgt die zweite Behauptung des Satzes, da M ein optimales Konsensus-MSA ist, da $E_M(s^*)$ minimal ist.

Die erste Behauptung folgt, da es eine Konsensus-MSA mit Alignment-Fehler $E_S(s^*)$ gibt und da für ein optimales Konsensus-MSA $\bar{M}$ gilt:

$$E_S(s^*) = E_{\bar{M}}(\mathcal{S}_{\bar{M}}) \overset{Beh.}{\geq} E_S(\mathcal{S}_{\bar{M}}|_\Sigma) \geq E_S(s^*).$$

Also ist $\mathcal{S}_{\bar{M}}|_\Sigma$ auch ein optimaler Steiner-String. ■

Es gilt:

$$\frac{E_S(s_c)}{E_S(s^*)} \leq 2 - \frac{2}{k},$$

wobei s_c ein Center-String von S ist und s^* ein optimaler Steiner-String ist.

Fassen wir das Ergebnis dieses Abschnitts noch einmal zusammen.

Theorem 4.25 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen. Das mehrfache Sequenzen Alignment M_c für S, das mit Hilfe der Center-Star-Methode konstruiert wurde, hat eine Sum-of-Pairs-Distanz, die maximal $(2 - \frac{2}{k})$ vom Optimum entfernt ist, und es hat einen Alignment-Fehler, der maximal $(2 - \frac{2}{k})$ vom Optimum entfernt ist.*

4.6 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 von Sequenzen Alignments ja ausnutzen.

4.6.1 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 Sequenz 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.

Definition 4.26 *Sei* $S = \{s_1, \ldots, s_k\}$ *eine Menge von Sequenzen über* Σ *und* $T = (V, E)$ *ein Baum mit*

$$V = I \cup B, \text{ wobei } \begin{array}{ll} I = \{v \in V | deg(v) > 1\} & \textit{(innere Knoten)} \\ B = \{v \in V | deg(v) = 1\} & \textit{(Blätter)} \end{array}$$

Weiterhin existiert eine Bijektion $\varphi : B \to S$ *mit*

$$\varphi : B \to S : v \mapsto s_v$$

Einen solcher Baum T *heißt* konsistent *zu* S.

Ein phylogenetisches mehrfaches Sequenzen Alignment *(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, das mit T *konsistent ist.*

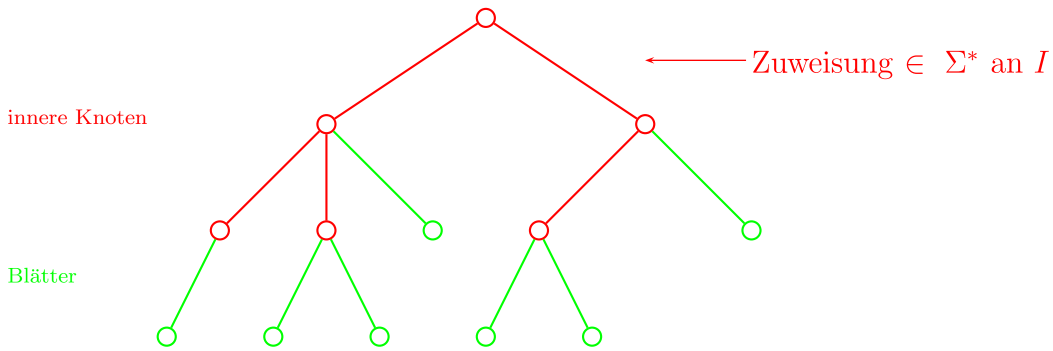


Abbildung 4.6: Skizze: Phylogenetisches mehrfaches Sequenzen Alignment

Für $(v, w) \in E(T)$ bezeiche $D_M(v, w) := d(s_v, s_w)$ die Alignment-Distanz des induzierten Alignments aus einem PMSA 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 der Alignment-Distanz des optimalen paarweisen Sequenzen Alignments.

Definition 4.27 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen und T ein zu S konsistenter Baum. Für ein PMSA M für S, das zu T konsistent ist, bezeichnet*

$$D_M(T) := \sum_{e \in E(T)} D_M(e) = \sum_{(v,w) \in E(T)} D_M((v,w))$$

die Distanz eines PMSA.

Definition 4.28 *Ein optimales phylogenetisches mehrfaches Sequenzen Alignment ist ein phylogenetisches mehrfaches Sequenzen Alignment M 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.

4.6.2 Geliftete Alignments

Um wieder eine Approximation generieren zu können, betrachten wir so genannte geliftete 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. Wir schränken uns sogar noch ein wenig mehr ein.

Definition 4.29 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen und T ein zu S konsistenter Baum. Ein phylogenetisches mehrfaches Sequenzen Alignment heißt* geliftet, *wenn für jeden inneren Knoten $v \in I$ gilt, dass ein Knoten $w \in V(T)$ mit $s_v = s_w$ und $(v, w) \in E(T)$ existiert.*

4.6.3 Konstruktion eines gelifteten aus einem optimalem Alignment

Nun zeigen wir, wie wir aus aus einem optimalen phylogenetischen mehrfachen Sequenzen 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.

Definition 4.30 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen und T ein zu S konsistenter Baum. Ein Knoten v heißt geliftet, wenn ein Knoten $w \in V(T)$ mit $s_v = s_w$ und $(v, w) \in E(T)$ existiert.*

Zu Beginn sind alle Blätter geliftet. Wir betrachten jetzt einen Knoten v, so dass alle seine Kinder geliftet sind und werden diesen Knoten selbst liften.

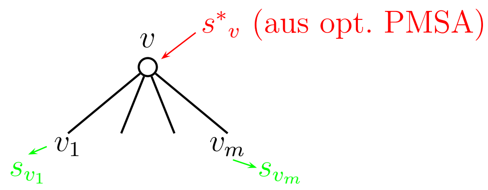


Abbildung 4.7: Skizze: Liften eines Knotens

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. 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).

4.6.4 Güte gelifteter Alignments

Definition 4.31 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen und sei und sei T ein zu S konsistenter Baum. Ist T^* ein optimales PMSA M^* für S und T, dann gilt für das aus T^* konstruierte geliftete PMSA M_L T^L:*

$$D_{M_L}(T^L) \leq 2 \cdot D_{M^*}(T^*)$$

Beweis: Wir betrachten zuerst eine Kante $(v, w) \in E(T)$. 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 gilt

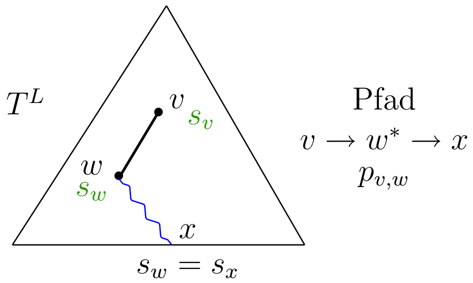


Abbildung 4.8: Skizze: Kante (v, w) definiert Pfad $p_{v,w}$

$D(v, w) = d(s_v, s_w) \leq d(s_v, {s^*}_v) + d({s^*}_v, s_w)$. Aufgrund des Liftings gilt außerdem: $d({s^*}_v, s_v) \leq d({s^*}_v, s_w)$. Somit erhalten wir insgesamt:

$$D(v, w) = d(s_v, s_w) \leq d(s_v, {s^*}_v) + d({s^*}_v, s_w) \leq 2 \cdot d({s^*}_v, s_w).$$

Betrachten wir eine Kante (v, w) in T^L, wobei w ein Kind von v ist. Diese Kante definiert in T^L einen Pfad $p_{v,w}$ von w zu einem Blatt x , indem wir vom Knoten w 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 wir angenommen haben, dass die Sequenzen aus S paarweise verschieden sind.

Betrachten wir jetzt diesen Pfad $p_{v,w}$ mit $v \to w = w_1 \to \cdots \to w_\ell = x$ in einem optimalen phylogenetischen mehrfachen Sequenzen Alignment, dann gilt aufgrund der Dreiecksungleichung:

$$\begin{aligned} d({s^*}_v, s_w) &\leq d({s^*}_v, {s^*}_{w_1}) + d({s^*}_{w_1}, {s^*}_{w_2}) + \cdots + d({s^*}_{w_{\ell-1}}, \underbrace{{s^*}_x}_{s^*_x = s_{w_\ell}}) \\ &\leq D^*(p_{v,w}) \end{aligned}$$

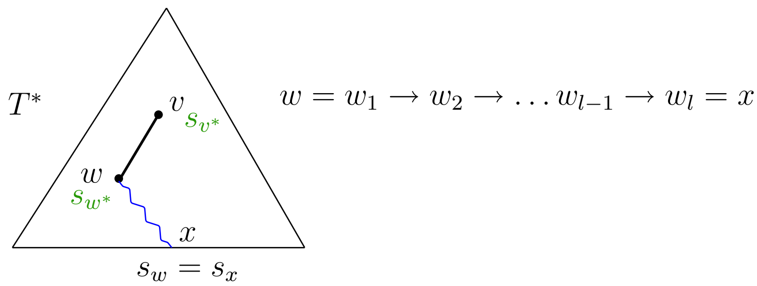


Abbildung 4.9: Skizze: Pfad $p_{v,w}$ im optimalen Baum T^*

Insgesamt erhalten wir dann

$$D(v, w) \leq 2d({s^*}_v, s_w) \leq 2D^*(p_{v,w}).$$

Wir können also die Distanz des geliſteten phylogenetischen Alignments geschickt wie folgt berechnen: Für alle grünen Kanten e gilt $D(e) = 0$. Wir müssen also nur

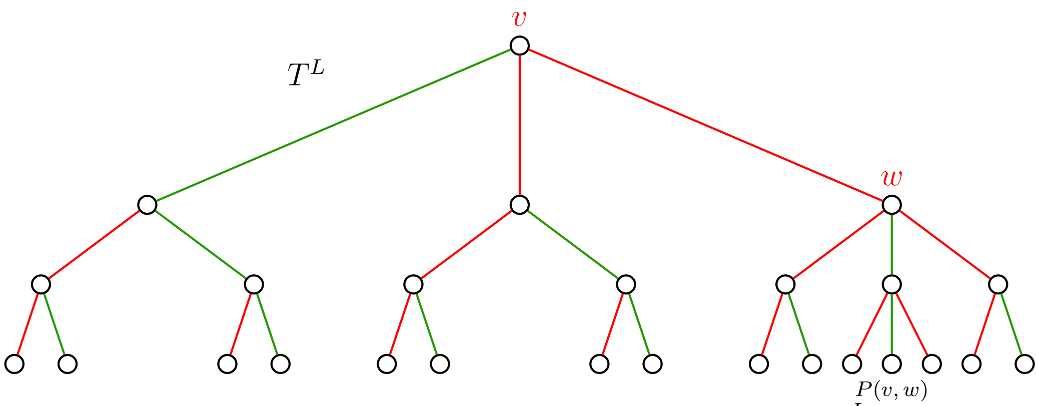


Abbildung 4.10: Skizze: Beziehung Kantengewichte in T^L zu T^*

die roten Kanten in T^L aufaddieren. Jede rote Kante (v, w) in T^L korrespondiert zu einem Pfad $p_{v,w}$ in T^*. Es gilt weiter, dass für alle Kanten (v, w), für die in T^L $D(v, w) > 0$ ist, die zueghörigen Pfade disjunkt sind. Somit kann die Summe der roten Kantengewichte durch die doppelte Summe aller Kantengewichte im Baum T^* abgeschätzt werden. Damit ergibt sich sich nun Folgendes:

$$\begin{aligned}
D(T^L) &= \sum_{(v,w)\in E(T)} D(v, w) \\
&= \sum_{\substack{(v,w)\in E(T) \\ D(v,w)>0}} \underbrace{D(v, w)}_{\leq 2D^*(p_{v,w})} \\
&\leq 2 \cdot \underbrace{\sum_{\substack{(v,w)\in E(T) \\ D(v,w)>0}} D^*(p_{v,w})}_{\leq D(T^*)} \\
&\leq 2 \cdot D(T^*).
\end{aligned}$$

■

Damit haben wir gezeigt, dass es ein geliftetes phylogenetisches Sequenzen Alignment gibt, dass höchstens um den Faktor zwei vom Optimum entfernt ist. Wenn wir jetzt ein optimales geliftetes phylogenetisches mehrfaches Sequenzen Alignment konstruieren, so gilt dies natürlich auch für dieses.

4.6.5 Berechnung eines optimalen gelifteten PMSA

Wir wollen jetzt mit Hilfe der Dynamischen Programmierung ein optimales phylogenetisches mehrfaches Sequenzen Alignment konstruieren. Dazu stellen wir zunächst wieder einmal eine Rekursionsgleichung auf. Hierfür bezeichnet $D(v, s)$ den Wert eines besten gelifteten PMSA für den am Knoten v gewurzelten Teilbaum, so dass v mit der Sequenzen $s \in S$ markiert ist (dabei muss natürlich s an einem der Kinder von v bereits vorkommen). Es gilt dann:

$$D(v,s) = \begin{cases} \sum\limits_{(v,w)\in E(T)} d(s, s_w) & \text{wenn alle Kinder von } v \text{ Blätter sind} \\ \sum\limits_{(v,w)\in E(T)} \min \left\{ d(s,s') + D(w,s') : \begin{array}{c} s' \text{ ist eine Markierung} \\ \text{eines Blattes in } T_v \end{array} \right\} & \end{cases}$$

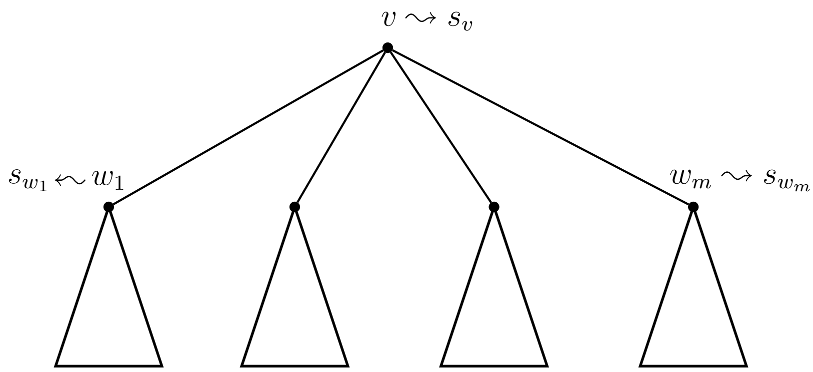


Abbildung 4.11: Skizze: Berechnung von $D(v, s)$

Während des Preprocessings ist es nötig, für alle Paare $(s, s') \in S^2$ $d(s, s')$ zu berechnen. Dafür ergibt sich folgender Zeitbedarf:

$$O(\sum_{i=1}^{k} \sum_{j=1}^{k} |s_i| * |s_j|) = O(\underbrace{\sum_{i=1}^{k} |s_i|}_{N} \underbrace{\sum_{j=1}^{k} |s_j|}_{N}) = O(N^2)$$

Nach dem Preprocessing kann jede Minimumbildung in konstanter Zeit erfolgen. Wir müssen uns nur noch überlegen, wie oft das Minimum gebildet wird. Dies geschieht für jede Baumkante (v, w) und jedes Paar von Sequenzen (s, s') genau einmal. Da ein binärer Baum, bei dem es keine Knoten mit genau einem Kind gibt, höchstens so viele innere Knoten wie 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: $O(N^2 + k^3)$. Fassen wir das Ergebnis noch zusammen.

Theorem 4.32 *Sei $S = \{s_1, \ldots, s_k\}$ eine k-elementige Menge von Sequenzen über Σ mit $N = \sum_{i=1}^{k} |s_i|$ und sei und sei T ein zu S konsistenter Baum. Ein PMSA für S und T, dessen Distanz maximal um 2 von einem optimalen PMSA für S und T abweicht, kann in Zeit $O(N^2 + k^3)$ konstruiert werden.*

Mit etwas Aufwand kann man den Summanden k^3 noch auf k^2 drücken.

Fragment Assembly

5.1 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.

5.1.1 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 Sequenzen der Länge von etwa 500 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 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 5.1).

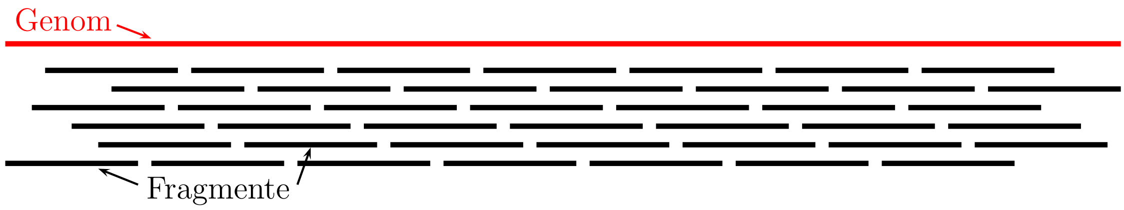


Abbildung 5.1: Skizze: Shotgun-Sequencing

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 einzelnen 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.

5.1.2 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 5.2).

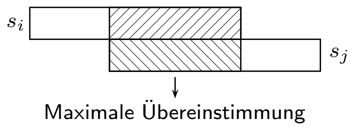


Abbildung 5.2: Skizze: Overlap-Detection

Fragment Layout Dann müssen die Fragmente so angeordnet werden, dass die Überlappungen möglichst gleich sind. Mit diesem Problem werden wir und in diesem Kapitel hauptsächlich beschäftigen.

Konsensus-String für gefundenes Layout Zum Schluss interpretieren wir das Fragment Layout wie ein mehrfaches Sequenzen Alignment und bestimmen den zugehörigen Konsensus-String, denn 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.

5.2 Overlap-Detection und Fragment-Layout

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.

5.2.1 Overlap-Detection mit Fehlern

Beim Sequenzieren der Fragmente treten in der Regel Fehler auf, so dass man damit rechnen muss. Ziel wird es daher sein, einen Suffix von s_i zu bestimmen, der ziemlich gut mit einem Präfix von s_j übereinstimmt. Da wir hierbei (Sequenzier-)Fehler zulassen, entspricht dies im Wesentlichen einem semi-globalen Alignment. Hierbei

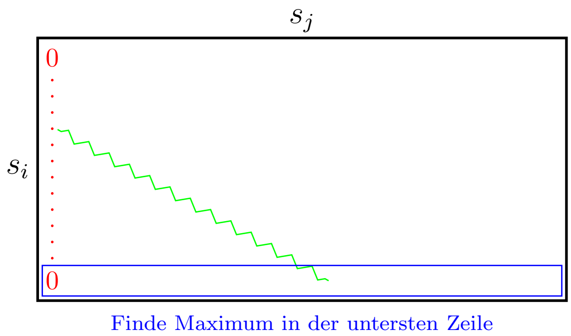


Abbildung 5.3: Semi-globale Alignments mit Ähnlichkeitsmaßen

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 5.1 *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 $i, j \in [1 : k]$ mit einem vorgegebenen beschränkten Fehler kann in Zeit $O((kn)^2)$ berechnet werden.*

5.2.2 Overlap-Detection ohne Fehler

Wir wollen jetzt noch eine effizientere Variante vorstellen, wenn wir keine Fehler zulassen. Dazu verwenden wir wieder einmal Suffix-Bäume.

5.2.2.1 Definition von $L(v)$

Wir konstruieren zuerst einen verallgemeinerter Suffix-Baum für $S = \{s_1, \ldots, s_k\}$. Wie wir schon gesehen haben ist der Platzbedarf $\Theta(kn)$ und die Laufzeit der Konstruktion $O(kn)$. Für jeden Knoten v des Suffix-Baumes generieren wir eine Liste

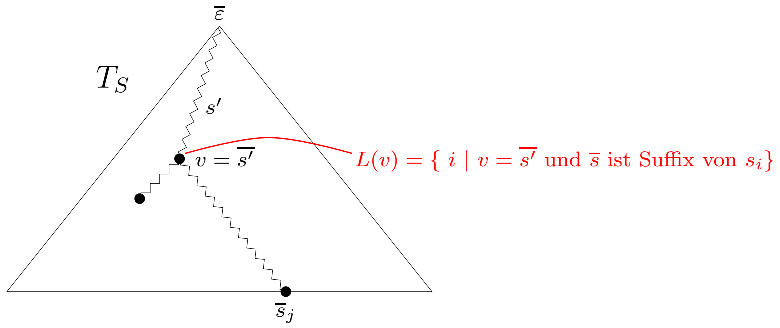


Abbildung 5.4: Skizze: Verallgemeinerter Suffix-Baum für $S = \{s_1, \ldots, s_k\}$

$L(v)$, die wie folgt definiert ist:

$$L(v) := \left\{ i \in [1:k] \ : \ \exists j \in [1:|s_i|] : v = \overline{s_{i,j} \cdots s_{i,|s_i|}} \right\}.$$

In der Liste $L(v)$ befinden sich also alle Indizes i, so dass ein Suffix von s_i im Knoten v endet.

Betrachten wir also jetzt einen Knoten v im verallgemeinerten Suffix-Baum mit seiner Liste $L(v)$, wie in Abbildung 5.4 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 durch den tiefsten Knoten v mit $i \in L(v)$ auf einem Pfad von $\overline{\varepsilon}$ zu $\overline{s}_j$ gegeben.

5.2.2.2 Erzeugung von $L(v)$

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 $O(kn)$.

5.2.2.3 Auffinden längster Suffix-Präfix-Matches

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 $S[i]$ 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 $S[i]$ alle Knoten v beinhalten, die zum einen Vorfahren von w sind und zum anderen soll in v ein Suffix von s_i enden.

Wenn wir jetzt eine Tiefensuche durch den verallgemeinerten Suffixbaum durchführen, werden wir für jeden neu aufgesuchten Knoten zuerst die Stack aktualisieren, d.h. wir füllen die Stacks geeignet auf. Wenn 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 wieder oben auf den Stacks befinden.

```
SUFFIX-PREFIX-MATCHES (int[] S)
{
    tree T_S;   /* verallgemeinerter Suffixbaum T_S */
    stack_of_nodes S[k];   /* je einen für jedes s_i ∈ S */
    int level[V(T_S)];
    int Overlap[k, k];
    level[ε̄] = 0;
    DFS(T_S, ε̄);
}

DFS (tree T, node v)
{
    for all (i ∈ L(v)) do S[i].push(v);
    if (v = s̄_j)
        for all (i ∈ [1 : k]) do
            if (not S[i].isEmpty())
                Overlap(s_i, s_j) = level[S[i].top()]};
    for all (v, w) do
    {
        level[w] = level[v] + 1;
        DFS(w);
    }
    for all (i ∈ L(v)) do S[i].pop();
}
```

Abbildung 5.5: Algorithmus: modifizierte Depth-First-Search

Sobald wir ein Blatt gefunden haben, das ohne Beschränkung der Allgemeint zum String s_j gehört, holen 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 Level mitberechnen, entspricht der Level einem längsten Overlap. Wie üblich ist der Level der Wurzel 0, und der Level eines Knoten ist um eines größer, als der Level seines Elters. Damit ergibt sich zur Lösung der in Abbildung 5.5 angegebene Algorithmus.

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:

$$\sum_{v \in V(T_S)} |L(v)| = \left| \left\{ t \in \Sigma^* \, : \, \exists s \in S : \exists j \in [1:|s|] : t = s_j \cdots s_{|s|} \right\} \right| = O(kn),$$

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 nur $O(k)$ 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 5.2 *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.

5.2.3 Greedy-Ansatz für das Fragment-Layout

Das Fragment-Layout kann man beispielsweise 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 Seqeunz in das bereits konstruierte aufnehmen lässt. Kommt es hier zu großen Widersprüchen, so wird die Seqeunz 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 insbesondere Probleme bei so genannten Repeats. Repeats sind 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.

5.3 Shortest Superstring Problem

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.

Geg.: $S = \{s_1, \ldots, s_k\}$
Ges.: $s^* \in \Sigma$, so dass s_i Teilwort von s^* und $|s^*|$ minimal ist

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 auch noch $\mathcal{APX}$-hart. Die Komplexitätsklasse $\mathcal{APX}$ umfasst alle Optimierungsprobleme, die man in polynomieller Zeit bis auf einen konstanten Faktor approximieren kann. Gehört nun ein Problem zu den schwierigsten Problemen der Klasse $\mathcal{APX}$ (ist also $\mathcal{APX}$-hart bezüglich einer geeigneten Reduktion, für Details verweisen wir auf Vorlesungen über Komplexitätstheorie), dann gibt es eine Zahl $\alpha > 1$, so dass einen Näherungslösung die eine Approximation bis auf einen Faktor kleiner als α liefert, nicht in polynomieller Zeit konstruiert werden kann (außer $\mathcal{P} = \mathcal{NP}$).

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.

5.3.1 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 5.3 *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.*

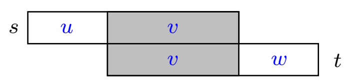


Abbildung 5.6: Skizze: Overlap und Präfix von s und t

In der Abbildung 5.6 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 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 den 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 $s' = abbab$ 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 5.4 *Sei $s, t \in \Sigma^*$, dann gilt*

$$s = p(s, t) \cdot o(s, t).$$

In der Abbildung 5.7 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 5.7: Beispiel: Overlaps und Präfixe

Lemma 5.5 *Sei $S = \{s_1, \ldots, s_k\}$, dann gilt für eine beliebige Permutation der Indizes $(i_1, \ldots, i_k) \in \mathcal{S}(k)$, 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 $p(s_1, s_1) \cdot s_1$ 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 5.4 ist $s_{i_k} = p(s_{i_k}, s_{i_{k+1}}) \cdot o(s_{i_k}, s_{i_{k+1}})$. Daher enthält $p(s_{i_k}, s_{i_{k+1}}) \cdot s_{i_k}$ 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 5.8noch einmal graphisch dargestellt. Also ist

$$p(s_{i_1}, s_{i_2}) \cdots p(s_{i_{k-1}}, s_{i_k}) \cdot s_{i_k}$$

ein Superstring für die Zeichenreihen in $S = \{s_1, \ldots, s_{k+1}\}$. ■

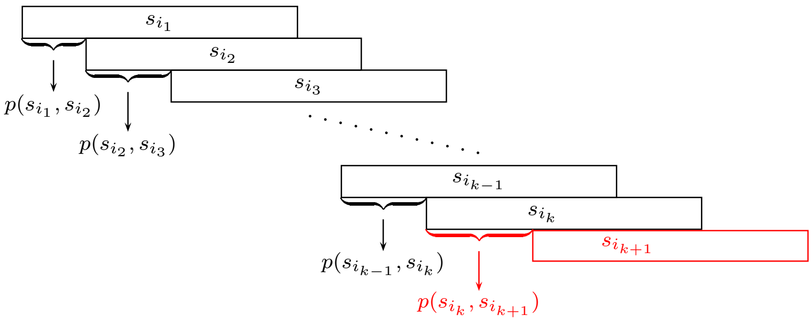


Abbildung 5.8: Skizze: Erweiterung des Superstrings

Korollar 5.6 *Sei $S = \{s_1, \ldots, s_k\}$, 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 5.4, das besagt, dass $s_{i_k} = p(s_{i_k}, s_1) \cdot o(s_{i_k}, s_1)$. ■

Lemma 5.7 *Sei $S = \{s_1, \ldots, s_k\}$ 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.

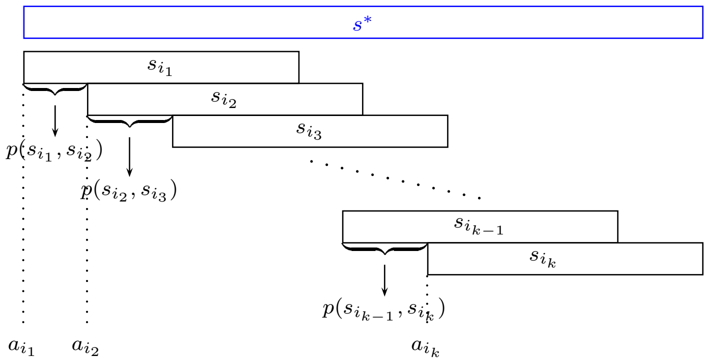


Abbildung 5.9: Skizze: Auffinden der s_i im Superstring

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 5.9 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 5.8 *Sei $S = \{s_1, \ldots, s_k\}$ 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 eine kürzesten Superstring zukommt. Dies motiviert die folgenden Definition eines Präfix-Graphen.

Definition 5.9 *Der gewichtete gerichtete Graph $G_S = (V, E, \gamma)$ für eine Menge S von Sequenzen $S = \{s_1, \ldots, s_k\}$ 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.*

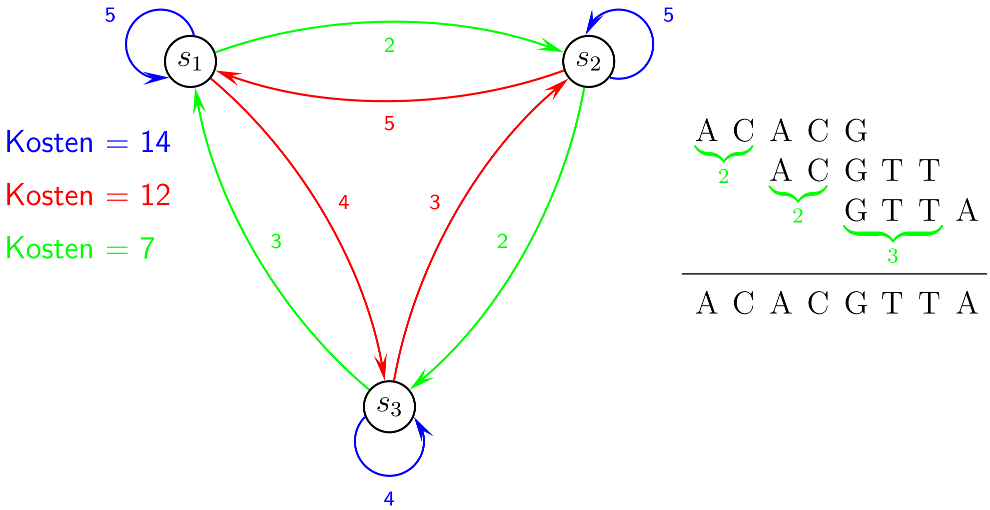


Abbildung 5.10: Beispiel: Zyklenüberdeckung für {ACACG,ACGTT,GTTA}

In Abbildung 5.10 ist der Präfix-Graphen samt einiger Zyklenüberdeckungen für das vorherige Beispiel dargestellt. Natürlich ist in diesem Beispiel auch (s_1, s_2) und s_3 eine Zyklenüberdeckung mit einem Gewicht von $2 + 3 + 4 = 9$.

5.3.2 Hamiltonsche Kreise und Zyklenüberdeckungen

Definition 5.10 *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 Präfixgraphen 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 5.11 *Sei $S = \{s_1, \ldots, s_k\}$ 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 5.12 *Sei $S = \{s_1, \ldots, s_k\}$ 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 5.13 *Sei $S = \{s_1, \ldots, s_k\}$ 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

$$\begin{array}{rcll} \Delta(G) & := & \max\{d(v) \; : \; v \in V\} & \quad \text{bzw.} \\ \delta(G) & := & \min\{d(v) \; : \; v \in V\} & \end{array}$$

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.

Definition 5.14 *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 eine Zyklenüberdeckung von 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 5.15 *Sei* $S = \{s_1, \ldots, s_k\}$ *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 5.16 *Sei* $S = \{s_1, \ldots, s_k\}$ *und* G_s *der zugehörige Präfix-Graph. Weiter sei* $C = \bigcup_{i=1}^{r} C_i$ *eine Zyklenüberdeckung für* G_S*. Dann bezeichne*

$$\begin{aligned} \ell(C_i) &:= \ell_i := \max\{|s_j| \; : \; s_j \in V(C_i)\}, \\ w(C_i) &:= w_i := \gamma(C_i) = \sum_{c \in E(C_i)} \gamma(c) = \sum_{(u,v) \in E(C_i)} |p(u,v)|. \end{aligned}$$

Lemma 5.17 *Ist* s' *ein Superstring von* $S = \{s_1, \ldots, s_k\}$*, der aus einer Zyklenüberdeckung* $C = \bigcup_{i=1}^{r} C_i$ *konstruiert wurde, dann gilt:*

$$\sum_{i=1}^{r} w_i \leq |s'| \leq \sum_{i=1}^{r} (w_i + \ell_i).$$

5.3.3 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 5.18 *Der gewichtete gerichtete Graph* $B_S = (V, E, \gamma)$ *für eine Menge von Sequenzen* $S = \{s_1, \ldots, s_k\}$ *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 5.11 ist der Overlap-Graph B_S für unser bereits bekannten Beispielsequenzen angegeben.

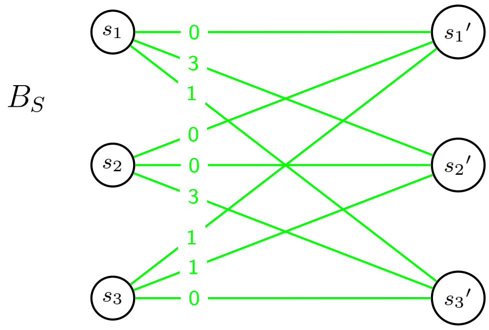


Abbildung 5.11: Beispiel: Overlap-Graph für {ACACG,ACGTT,GTTA}

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 5.19 *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Kantenmenge $M \subseteq E$ heißt* Matching*, wenn für den Graphen $G(M) = (V, M)$ gilt, dass $\Delta(G(M)) = 1$ und $\delta(G(M) \geq 0$. Eine Kantenmenge M heißt* perfektes Matching*, wenn gilt, dass $\Delta(G(M)) = 1$ und $\delta(G(M)) = 1$.*

Man beachte, dass nur Graphen mit einer geraden Anzahl von Knoten ein perfektes Matching besitzen können. Im Graphen in der Abbildung 5.12 entsprechen die rot hervorgehobenen Kanten einem perfekten Matching M des Graphen G_S.

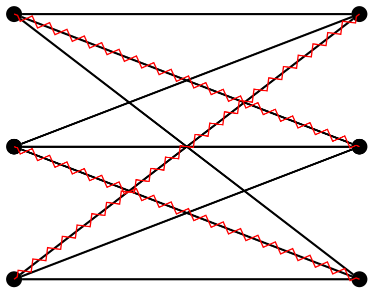


Abbildung 5.12: Beispiel: Matching in G_S

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 folgenden Abbildung 5.13 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 das korrespondierende perfekte Matching auf den entsprechende Knoten besonders hervorgehoben.

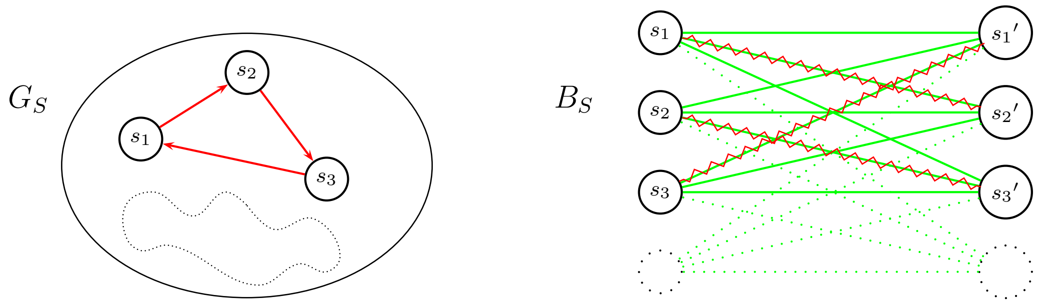


Abbildung 5.13: Skizze: Cycle Cover in G_S entspricht perfektem Matching in B_S

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 5.20 *Sei $S = \{s_1, \ldots, s_k\}$ eine Menge von Sequenzen über Σ 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 $\gamma(C) = N - \gamma(M)$.*

5.3.4 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 5.14 angegeben.

Zunächst einmal halten wir fest, dass wir immer ein perfektes Matching erhalten. Dies folgt unmittelbar aus dem Heiratssatz (oder 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.

```
W_Max_Matching (graph (V, E, γ))
{
    set M = ∅;
    Sortiere E nach den Gewichten γ                      → O(m log m)
    E = {e_1, ..., e_m} mit γ(e_1) ≥ ··· ≥ γ(e_m)
    for (i = 1; i ≤ m; i++)                               → O(m)
        if (e_i ist zu keiner anderen Kante aus M inzident)
            M = M ∪ {e_i};
    return M;
}
```

Abbildung 5.14: Algorithmus: Greedy-Methode für ein gewichtsmaximales Matching

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 5.21 *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 5.22 *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 5.15 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.

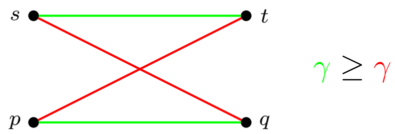


Abbildung 5.15: Skizze: Monge-Bedingung

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 5.23 *Sei $S = \{s_1, \ldots, s_k\}$ 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 5.16 illustrierte Bild.

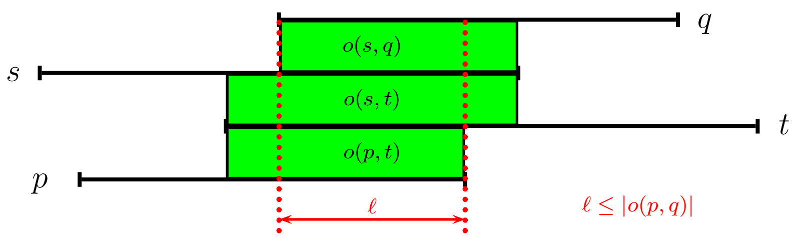


Abbildung 5.16: Skizze: Overlap-Graph erfüllt Monge-Bedingung

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 5.24 *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 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 die Ausgabe M eines Ablaufs des Greedy-Algorithmus, so dass $|M^* \triangle M|$ minimal 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 5.17 noch einmal illustriert.

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

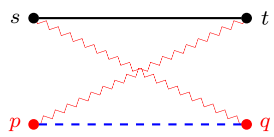


Abbildung 5.17: Skizze: Widerspruchsbeweis zur Optimalität

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. ■

5.3.5 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 5.25 *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 $\operatorname{ggT}(a, b)$ für zwei natürliche Zahlen $a, b \in \mathbb{N}$ den größten gemeinsamen Teiler: $\operatorname{ggT}(a, b) = \max\{k \in \mathbb{N} : (k \mid a) \wedge (k \mid b)\}$, wobei $k \mid a$ gilt, wenn es ein $n \in \mathbb{N}$ gibt, so dass $n \cdot k = a$.

Lemma 5.26 (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 $\operatorname{ggT}(p, q)$.*

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 folgenden Abbildung illustriert.

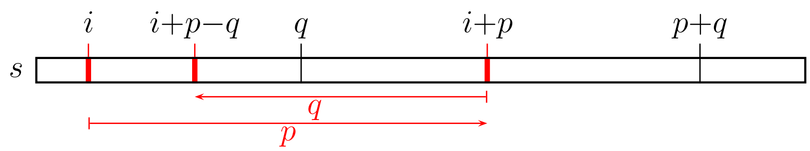


Abbildung 5.18: Skizze: 1. Fall $i \leq q$

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 folgenden Abbildung illustriert.

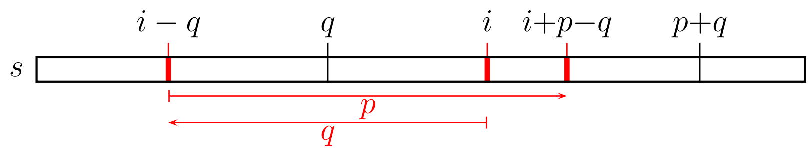


Abbildung 5.19: Skizze: 2.Fall $i > q$

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 ist für $p > q$ durch $\mathrm{ggT}(p, q) = \mathrm{ggT}(q, p - q)$ rekursiv definiert; dabei ist die Abbruchbedingung durch $\mathrm{ggT}(p, p) = p$ gegeben. Da die Perioden von s dieselbe Rekursionsgleichung erfüllen, muss also auch $\mathrm{ggT}(p, q)$ 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 5.27 (Overlap-Lemma) *Sei $S = \{s_1, \ldots, s_k\}$ 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 $|t_j| > |o(t_i, t_j)| > w_i + w_j \geq w_j$.

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 5.20. 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.

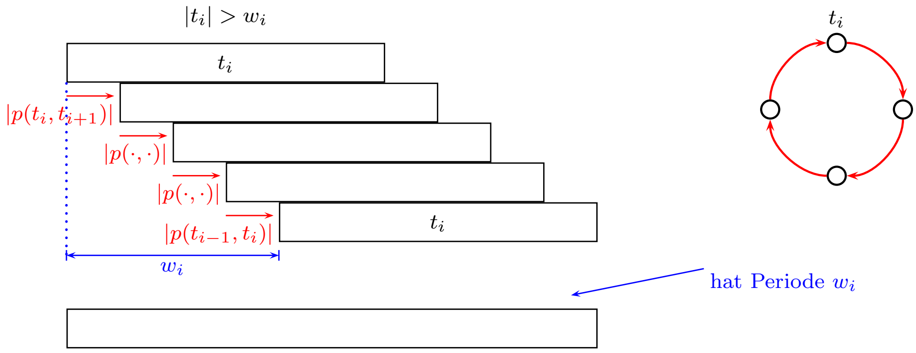


Abbildung 5.20: Skizze: Wörter eines Zyklus

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 5.21. Man kann also die beiden Zyklen zu einem verschmelzen, so dass das

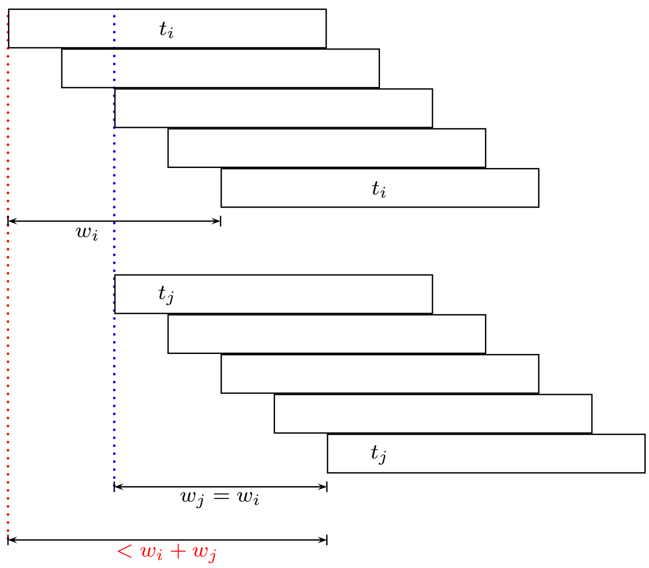


Abbildung 5.21: Skizze: 2 Zyklen mit demselben Gewicht

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 5.22 veranschaulicht. Eine analoge Überlegung gilt natürlich auch für t_j, so dass also auch t_j eine Periode von g besitzt.

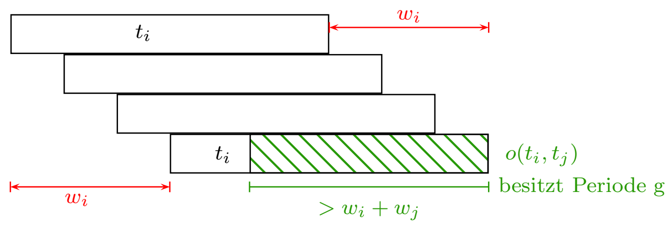


Abbildung 5.22: Skizze: Übertragung der Periode g von $o(t_i, t_j)$ auf t_i

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. Dazu betrachten wir die Illustration in Abbildung 5.23. Da sowohl t_i als auch t_j einen Periode von g

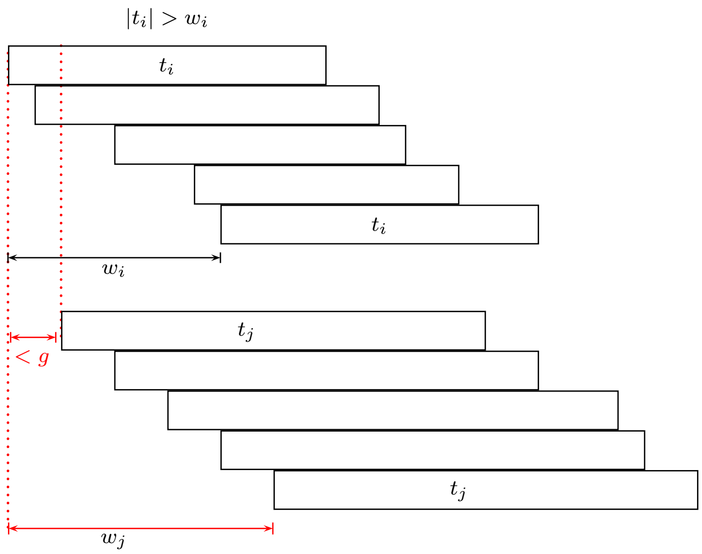


Abbildung 5.23: Skizze: Verschmelzung zweier Zyklen mit $w_i \neq w_j$

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. ■

Mit Hilfe des eben bewiesenen Overlap-Lemmas können wir jetzt die Approximationsgüte des von uns vorgestellten Greedy-Algorithmus abschätzen.

Theorem 5.28 *Sei s' der durch den Greedy-Algorithmus konstruierte Superstring für $S = \{s_1, \ldots, s_k\}$. 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 5.7 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:

$$\begin{aligned}
|\tilde{s}| &= \sum_{i=1}^{r} |p(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| + |\underbrace{o(\tilde{s}_{j_r}, \tilde{s}_{j_1})}_{\geq 0}| \\
&\geq \sum_{i=1}^{r} \Big(\underbrace{|\tilde{s}_{j_i}|}_{=\ell_i} - |o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \Big) \\
&\quad \text{aufgrund des Overlap-Lemmas gilt:} \\
&\quad |o(\tilde{s}_{j_i}, \tilde{s}_{j_{(i \bmod r)+1}})| \leq (w_{j_i} + w_{j_{(i \bmod r)+1}}) \\
&\geq \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} \Big(w_{j_i} + w_{j_{(i \bmod r)+1}} \Big) \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_{(i \bmod r)+1}} \\
&\quad \text{Verschiebung der Indizes um 1} \\
&= \sum_{i=1}^{r} \ell_i - \sum_{i=1}^{r} w_{j_i} - \sum_{i=1}^{r} w_{j_i} \\
&\quad \text{da } (j_1, \ldots, j_r) \text{ nur eine Permutation von } [1 : r] \text{ ist} \\
&= \sum_{i=1}^{r} \ell_i - 2 \sum_{i=1}^{r} w_i \\
&= \sum_{i=1}^{r} (\ell_i - 2w_i).
\end{aligned}$$

Der kürzeste Superstring für S ist sicherlich nicht kürzer als der für $\tilde{S}$, da ja $\tilde{S} \subseteq S$ gilt. Also gilt:

$$SSP(S) \geq SSP(\tilde{S}) = |\tilde{s}| \geq \sum_{i=1}^{r}(\ell_i - 2w_i). \tag{5.1}$$

Nach Konstruktion des Superstrings s' für s mit Hilfe des Greedy-Algorithmus gilt:

$$\begin{aligned}
|s'| &\leq \sum_{i=1}^{r}(w_i + \ell_i) \\
&\leq \underbrace{\sum_{i=1}^{r}(\ell_i - 2w_i)}_{\leq SSP(S)} + \sum_{i=1}^{r} 3w_i \\
&\quad \text{mit Hilfe von Ungleichung 5.1} \\
&\leq SSP(S) + 3 \cdot SSP(S) \\
&\leq 4 \cdot SSP(S)
\end{aligned}$$

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 5.6 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.

5.3.6 Zusammenfassung und Beispiel

In Abbildung 5.24 ist die Vorgehensweise für die Konstruktion eines kürzesten Superstrings mit Hilfe der Greedy-Methode noch einemal skizziert.

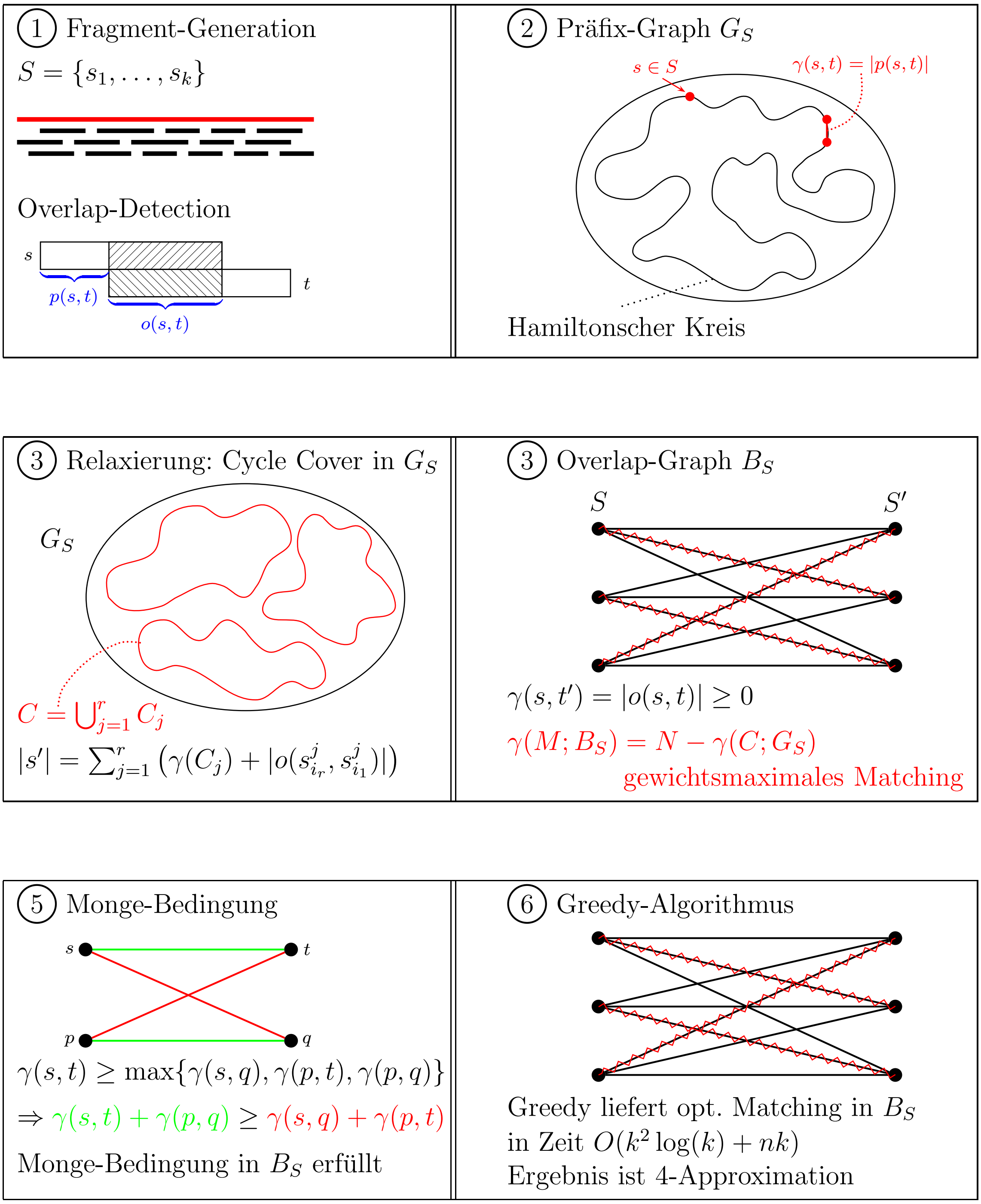
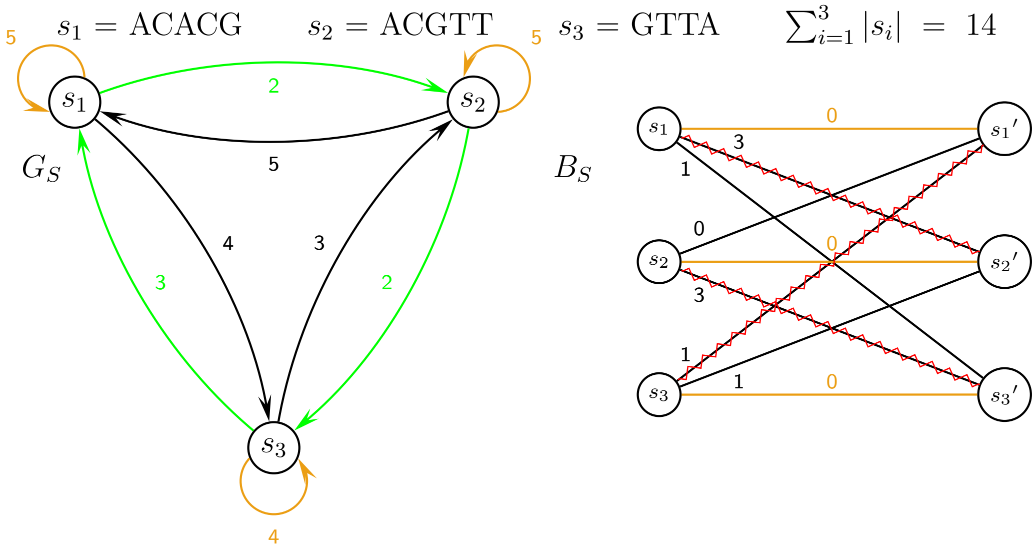


Abbildung 5.24: Skizze: Zusammenfassung der Greedy-SSP-Approximation



$\gamma(C) = 7 = 14 - 7$ $\qquad\qquad$ $\gamma(M) = 7 = 14 - 7$

Abbildung 5.25: Beispiel: Greedy-Algorithmus für Superstrings

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 5.25 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 aufzubrechen, wo der zugehörige Overlap-Wert klein ist. Daher wählen wir als Kreisdarstellung $(s_1, s_2, s_{,3})$ und erhalten folgenden Superstring:

$$
\begin{array}{cccccccccc}
 & & \mathrm{A} & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{G} & & & \\
 & & & & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \\
 & & & & & & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A} \\
\hline
s & = & \mathrm{A} & \mathrm{C} & \mathrm{A} & \mathrm{C} & \mathrm{G} & \mathrm{T} & \mathrm{T} & \mathrm{A}
\end{array}
$$

Mit Bezug zum Präfix-Graphen ergibt sich folgenden Korrespondenz zu den Knoten im hamiltonschen Kreis:

$$s = \underbrace{AC}_{p(s_1,s_2)}\ \underbrace{AC}_{p(s_2,s_3)}\ \underbrace{GTT}_{p(s_3,s_1)}\ \underbrace{A}_{o(s_3,s_1)}\ .$$

5.4 (*) Whole Genome Shotgun-Sequencing

Wie schon angemerkt, wurde vermutet, dass die eben vorgestellte Methode nur für nicht zu lange oder einfachere Genome (ohne Repeats) anwendbar ist. Celera Genomics hat mit der Sequenzierung der Fruchtfliege Drosophila Melanogaster und dem menschlichen Genom bewiesen, dass sich dieses Verfahren prinzipiell auch zur Sequenzierung ganzer Genome anwenden lässt. Natürlich sind hierzu noch ein paar weitere Tricks nötig, auf die wir hier noch ganz kurz eingehen wollen.

5.4.1 Sequencing by Hybridization

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 Seqeunz auftreten. Betrachten wir hierzu ein Beispiel dass in Abbildung 5.26 angegeben ist. In unserem Beispiel erhalten wir also die folgende

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 5.26: Beispiel: Teilsequenzen der Länge 4, die bei SBH ermittelt werden

Menge an (sehr kurzen, wie für SBH charakteristisch) so genannten *Oligos*:

$$\{\mathrm{ACGA}, \mathrm{ACAG}, \mathrm{AGAC}, \mathrm{CAGA}, \mathrm{CGAC}, \mathrm{GACA}, \mathrm{GACG}, \mathrm{GACT}, \mathrm{TGAC}\}.$$

Auch hier müssen wir wieder einen Superstring 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-Stranges ein Oligo der betrachteten Länge bekannt ist.

Beim SSP 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, was hingegen algorithmisch sehr leicht ist.

Definition 5.29 *Sei $G = (V, E)$ ein gerichteter Graph. Ein Pfad $p = (v_1, \ldots, v_\ell)$ heißt* eulersch*, wenn alle Kanten des Graphen genau einmal in diesem Pfad enthalten sind, d.h.:*

- $(v_{i-1}, v_i) \in E$ *für alle* $i \in [2 : \ell]$,
- $|\{(v_{i-1}, v_i) : i \in [2 : \ell]\}| = |E|$.

Ein Graph heißt eulersch*, wenn er einen eulerschen Pfad besitzt.*

Wir weisen hier darauf hin, dass wir aus gegebenem Anlass der 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 5.30 *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

Definition 5.31 *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 5.27 veranschaulichen. Wie man dem Beispiel ansieht, kann es durchaus mehrere eulersche

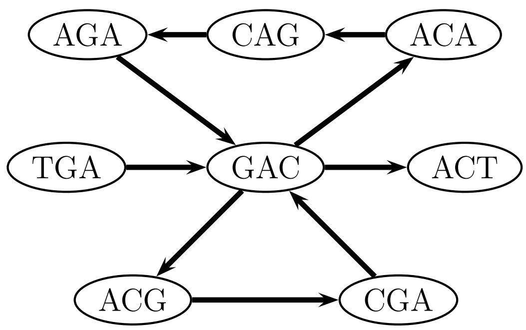


Abbildung 5.27: Beispiel: Oligo-Graph

Pfade im Oligo-Graphen geben. Einer davon entspricht der ursprünglichen Sequenz.

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 Seqeunz 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).

5.4.2 Anwendung auf Fragment Assembly

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, das 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). 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. Somit suchen wir jetzt nach einem eulerschen Pfad im Oligo-Graphen, der diese Pfade respektiert. Dies macht die Aufgabe in der Hinsicht leichter, dass bei mehreren möglichen eulerschen Pfaden leichter ersichtlich ist, welche Variante zu wählen ist.

Ein weiterer Trick den Celera Genomics bei der Sequenzierung des menschlichen Genoms angewendet hat ist, dass nicht Fragmente der Länge 500 sequenziert worden sind, sondern dass man hat Bruchstücke der Länge von etwa 2000 und 10000 Basenpaaren konstruiert. Diese werden dann von beiden Seiten her auf 500 Basenpaare ansequenziert. Dies hat den Vorteil, dass man für die meisten sequenzierten Teile (nicht für alle, aufgrund von Sequenzierfehlern) auch noch jeweils ein Geschwister-Teil im Abstand von 2000 bzw. 10000 Basenpaaren kennt. Dies erlaubt beim Zusammensetzen der Teile eine weitere Überprüfung, ob Abstand und Orientierung der Geschwister-Fragmente korrekt sind.

Physical Mapping

6.1 Biologischer Hintergrund und Modellierung

Bei der *genomischem Kartierung* (engl. *physical mapping*) geht es darum, einen ersten groben Eindruck des Genoms zu bekommen. Dazu soll für „charakteristische“ Sequenzen der genaue Ort auf dem Genom festgelegt werden. Im Gegensatz zu *genetischen Karten* (engl. *genetic map*), wo es nur auf die lineare und ungefähre Anordnung einiger bekannter oder wichtiger Gene auf dem Genom ankommt, will man bei *genomischen Karten* (engl. *physical map*) die Angaben nicht nur ungefähr, sondern genau bis auf die Position der Basenpaare ermitteln.

6.1.1 Genomische Karten

Wir wollen zunächst die Idee einer genomischen Karte anhand einer „Landkarte aus Photographien“ für Deutschland beschreiben. Wenn man einen ersten groben Überblick der Orte von Deutschland bekommen will, dann wäre ein erster Schritt, die Kirchtürme aus ganz Deutschland so zu erfassen. Kirchtürme bieten zum einen den Vorteil, dass sich ein Kirchturm als solcher sehr einfach erkennen lässt, und zum anderen, dass Kirchtürme verschiedener Kirchen in der Regel doch deutlich unterschiedlich sind. Wenn man nun die Bilder der Kirchtürme den Orten in Deutschland zugeordnet hat, dann kann man für die meisten Photographien entscheiden, zu welchem Ort sie gehören, sofern ein Kirchturm darauf zu sehen ist. Die äquivalente Aufgabe bei der genomischen Kartierung ist die Zuordnung der Kirchtürme auf die Orte in Deutschland. Ein Genom ist dabei im Gegensatz zu Deutschland ein- und nicht zweidimensional.

Ziel der genomischen Kartierung ist es nun ungefähr alle 10.000 Basenpaare eine charakteristische Sequenz auf dem Genom zu finden und zu lokalisieren. Dies ist wichtig für einen ersten Grob-Eindruck für ein Genom. Für das Human Genome Project war eine solche Kartierung wichtig, damit man das ganze Genom relativ einfach in viele kleine Stücke aufteilen konnte, so dass die einzelnen Teile von unterschiedlichen Forscher-Gruppen sequenziert werden konnten. Die einzelnen Teile konnten daher dann unabhängig und somit hochgradig parallel sequenziert werden. Damit zum Schluss die einzelnen sequenzierten Stücke wieder den Orten im Genom zugeordnet werden konnten, wurde dann eine genomische Karte benötigt.

Obwohl Celera Genomics mit dem Whole Genome Shotgun Sequencing gezeigt hat, dass für die Sequenzierung großer Genome eine genomische Karte nicht unbedingt benötigt wird, so ist diese zum einen doch hilfreich und zum anderen auch unerlässlich beim Vergleich von ähnlichen Genomen, da auch in absehbarer Zukunft aus Kostengründen nicht jedes beliebige Genom einfach einmal schnell sequenziert werden kann.

6.1.2 Konstruktion genomischer Karten

Wie erstellt man nun solche genomischen Karten. Das ganze Genom wird in viele kleinere Stücke, so genannte *Fragmente* zerlegt. Dies kann mechanisch durch Sprühdüsen oder biologisch durch Restriktionsenzyme geschehen. Diese einzelnen kurzen Fragmente werden dann auf spezielle Landmarks hin untersucht.

Als Landmarks können zum Beispiel so genannte *STS*, d.h. *Sequence Tagged Sites*, verwendet werden. Dies sind kurze Sequenzabschnitte, die im gesamten Genom eindeutig sind. In der Regel sind diese 100 bis 500 Basenpaare lang, wobei jedoch nur die Endstücke von jeweils 20 bis 40 Basenpaaren als Sequenzfolgen bekannt sind. Vorteil dieser STS ist, dass sie sich mit Hilfe der Polymerasekettenreaktion sehr leicht nachweisen lassen, da gerade die für die PCR benötigten kurzen Endstücke als Primer bekannt sind. Somit lassen sich die einzelnen Fragmente daraufhin untersuchen, ob sie ein STS enthalten oder nicht.

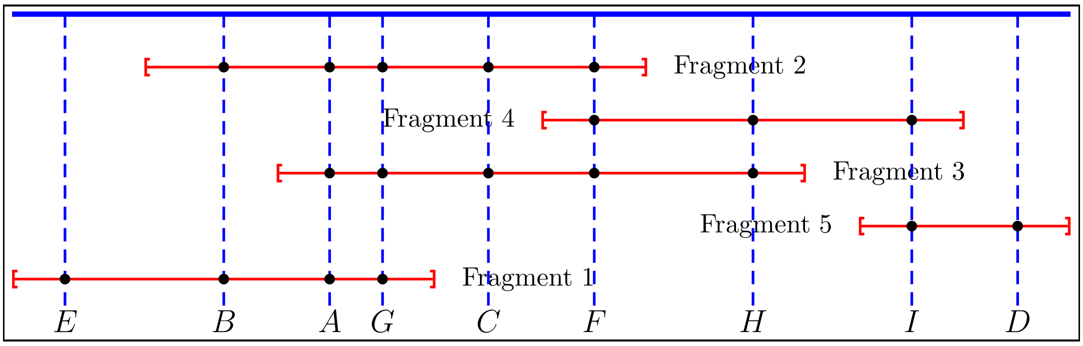


Abbildung 6.1: Skizze: Genomische Kartierung

Dies ist in Abbildung 6.1 illustriert. Dabei ist natürlich weder die Reihenfolge der STS im Genom, noch die Reihenfolge der Fragmente im Genom (aufsteigend nach Anfangspositionen) bekannt. Die Experimente liefern nur, auf welchem Fragment sich welche STS befindet. Die Aufgabe der genomischen Kartierung ist es nun, die Reihenfolge des STS im Genom (und damit auch die Reihenfolge des Auftretens der Fragmente im Genom) zu bestimmen. In dem Beispiel, dass in der Abbildung 6.1 angegeben ist, erhält man als Ergebnis des Experiments nur die folgende Information:

$$\begin{aligned} S_1 &= \{A, B, C, F, G\}, \\ S_2 &= \{F, H, I\}, \\ S_3 &= \{A, C, F, G, H\}, \\ S_4 &= \{D, I\}, \\ S_5 &= \{A, B, E, G\}. \end{aligned}$$

Hierbei gibt die Menge S_i an, welche STS das Fragment i enthält. In der Regel sind natürlich die Fragmente nicht in der Reihenfolge ihres Auftretens durchnummeriert, sonst wäre die Aufgabe ja auch zu trivial.

Aus diesem Beispiel sieht man schon, das sich die Reihenfolge aus diesen Informationen nicht immer eindeutig rekonstruieren lässt. Obwohl im Genom A vor G auftritt, ist dies aus den experimentellen Ergebnissen nicht ablesbar.

6.1.3 Modellierung mit Permutationen und Matrizen

In diesem Abschnitt wollen wir zwei recht ähnliche Methoden vorstellen, wie man die Aufgabenstellung mit Mitteln der Informatik modellieren kann. Eine Modellierung haben wir bereits kennen gelernt: Die Ergebnisse werden als Mengen angegeben. Was wir suchen ist eine Permutation der STS, so dass für jede Menge gilt, dass die darin enthaltenen Elemente in der Permutation zusammenhängend vorkommen, also durch keine andere STS separiert werden. Für unser Beispiel wären also $EBAGCFHID$ und $EBGACFHID$ sowie $DIHFCGABE$ und $DIHFCAGBE$ zulässige Permutationen, da hierfür gilt, dass die Elemente aus S_i hintereinander in der jeweiligen Permutation auftreten.

Wir merken hier bereits an, dass wir im Prinzip immer mindestens zwei Lösungen erhalten, sofern es eine Lösung gibt. Aus dem Ergebnis können wir nämlich die Richtung nicht feststellen. Mit jedem Ergebnis ist auch die rückwärts aufgelistete Reihenfolge eine Lösung. Dies lässt sich in der Praxis mit zusätzlichen Experimenten jedoch leicht lösen.

Eine andere Möglichkeit wäre die Darstellung als eine $n \times m$-Matrix, wobei wir annehmen, dass wir n verschiedene Fragmente und m verschiedene STS untersuchen. Der Eintrag an der Position (i, j) ist genau dann 1, wenn die STS j im Fragment i enthalten ist, und 0 sonst. Diese Matrix für unser Beispiel ist in Abbildung 6.2 angegeben. Hier ist es nun unser Ziel, die Spalten so permutieren, dass die Einsen in jeder Zeile aufeinander folgend (konsekutiv) auftreten. Wenn es eine solche Permutation gibt, ist es im Wesentlichen dieselbe wie die, die wir für unsere andere Modellierung erhalten. In der Abbildung 6.2 ist rechts eine solche Spaltenpermutation angegeben.

	A	B	C	D	E	F	G	H	I
1	1	1	1	0	0	1	1	0	0
2	0	0	0	0	0	1	0	1	1
3	1	0	1	0	0	1	1	1	0
4	0	0	0	1	0	0	0	0	1
5	1	1	0	0	1	0	1	0	0

	E	B	A	G	C	F	H	I	D
1	0	1	1	1	1	1	0	0	0
2	0	0	0	0	0	1	1	1	0
3	0	0	1	1	1	1	1	0	0
4	0	0	0	0	0	0	0	1	1
5	1	1	1	1	0	0	0	0	0

Abbildung 6.2: Beispiel: Matrizen-Darstellung

Daher sagt man auch zu einer 0-1 Matrix, die eine solche Permutation erlaubt, dass sie die *Consecutive Ones Property*, kurz *C1P*, erfüllt.

6.1.4 Fehlerquellen

Im vorigen Abschnitt haben wir gesehen, wir wir unser Problem der genomischen Kartierung geeignet modellieren können. Wir wollen jetzt noch auf einige biologische Fehlerquellen eingehen, um diese bei späteren anderen Modellierungen berücksichtigen zu können.

False Positives: Leider kann es bei den Experimenten auch passieren, dass eine STS in einem Fragment i identifiziert wird, obwohl sie gar nicht enthalten ist. Dies kann zum Beispiel dadurch geschehen, dass in der Sequenz sehr viele Teilsequenzen auftreten, die den Primern der STS zu ähnlich sind, oder aber die Primer tauchen ebenfalls sehr weit voneinander entfernt auf, so dass sie gar keine STS bilden, jedoch dennoch vervielfältigt werden. Solche falschen Treffer werden als *False Positives* bezeichnet.

False Negatives: Analog kann es passieren, dass, obwohl eine STS in einem Fragment enthalten ist, diese durch die PCR nicht multipliziert wird. Solche fehlenden Treffer werden als *False Negatives* bezeichnet.

Chimeric Clones: Außerdem kann es nach dem Aufteilen in Fragmente passieren, dass sich die einzelnen Fragmente zu längeren Teilen rekombinieren. Dabei könnten sich insbesondere Fragmente aus ganz weit entfernten Bereichen des untersuchten Genoms zu einem neuen Fragment kombinieren und fälschlicherweise Nachbarschaften liefern, die gar nicht existent sind. Solche Rekombinationen werden als *Chimeric Clones* bezeichnet.

6.2 PQ-Bäume

In diesem Abschnitt wollen wir einen effizienten Algorithmus zur Entscheidung der Consecutive Ones Property vorstellen. Obwohl dieser Algorithmus mit keinem, der im vorigen Abschnitt erwähnten Fehler umgehen kann, ist er dennoch von grundlegendem Interesse.

6.2.1 Definition von PQ-Bäumen

Zur Lösung der C1P benötigen wir das Konzept eines PQ-Baumes. Im Prinzip handelt es sich hier um einen gewurzelten Baum mit besonders gekennzeichneten inneren Knoten und Blättern.

Definition 6.1 *Sei* $\Sigma = \{a_1, \ldots, a_n\}$ *ein endliches Alphabet. Dann ist ein* PQ-Baum *über* Σ *induktiv wie folgt definiert:*

- *Jeder einelementige Baum (also ein Blatt), das mit einem Zeichen aus* Σ *markiert ist, ist ein PQ-Baum.*
- *Sind* $T_1, \ldots, T_k$ *PQ-Bäume, dann ist der Baum, der aus einem so genannten* P-Knoten *als Wurzel entsteht und dessen Kinder die Wurzeln der Bäume* $T_1, \ldots, T_k$ *sind, ebenfalls ein PQ-Baum.*
- *Sind* $T_1, \ldots, T_k$ *PQ-Bäume, dann ist der Baum, der aus einem so genannten* Q-Knoten *als Wurzel entsteht und dessen Kinder die Wurzeln der Bäume* $T_1, \ldots, T_k$ *sind, ebenfalls ein PQ-Baum.*

Abbildung 6.3: Skizze: Darstellung von P- und Q-Knoten

In der Abbildung 6.3 ist skizziert, wie wir in Zukunft P- bzw. Q-Knoten graphisch darstellen wollen. P-Knoten werden durch Kreise, Q-Knoten durch lange Rechtecke dargestellt. Für die Blätter führen wir keine besondere Konvention ein. In der Abbildung 6.4 ist das Beispiel eines PQ-Baumes angegeben.

Im Folgenden benötigen wir spezielle PQ-Bäume, die wir jetzt definieren wollen.

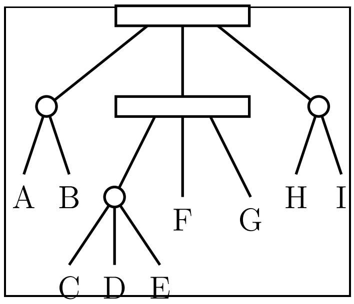


Abbildung 6.4: Beispiel: Ein PQ-Baum

Definition 6.2 *Ein PQ-Baum heißt* echt*, wenn folgende Bedingungen erfüllt sind:*

- *Jedes Element $a \in \Sigma$ kommt genau einmal als Blattmarkierung vor;*
- *Jeder P-Knoten hat mindestens zwei Kinder;*
- *Jeder Q-Knoten hat mindestens drei Kinder.*

Der in Abbildung 6.4 angegebene PQ-Baum ist also ein echter PQ-Baum.

An dieser Stelle wollen wir noch ein elementares, aber fundamentales Ergebnis über gewurzelte Bäume wiederholen, dass für PQ-Bäume im Folgenden sehr wichtig sein wird.

Lemma 6.3 *Sei T ein gewurzelter Baum, wobei jeder innere Knoten mindestens zwei Kinder besitzt, dann ist die Anzahl der inneren Knoten echt kleiner als die Anzahl der Blätter von T.*

Da ein echter PQ-Baum diese Eigenschaft erfüllt (ein normaler in der Regel nicht), wissen wir, dass die Anzahl der P- und Q-Knoten kleiner als die Kardinalität des betrachteten Alphabets Σ ist.

Die P- und Q-Knoten besitzen natürlich eine besondere Bedeutung, die wir jetzt erläutern wollen. Wir wollen PQ-Bäume im Folgenden dazu verwenden, Permutation zu beschreiben. Daher wird die Anordnung der Kinder an P-Knoten willkürlich sein (d.h. alle Permutationen der Teilbäume sind erlaubt). An Q-Knoten hingegen ist die Reihenfolge bis auf das Umdrehen der Reihenfolge fest. Um dies genauer beschreiben zu können benötigen wir noch einige Definitionen.

Definition 6.4 *Sei T ein echter PQ-Baum über Σ. Die* Frontier *von T, kurz $f(T)$ ist die Permutation über Σ, die durch das Ablesen der Blattmarkierungen von links nach rechts geschieht (also die Reihefolge der Blattmarkierungen in einer Tiefensuche unter Berücksichtigung der Ordnung auf den Kindern jedes Knotens).*

Die Frontier des Baumes aus Abbildung 6.4 ist dann ABCDEFGHI.

Definition 6.5 *Zwei echte PQ-Bäume T und T' heißen* äquivalent*, kurz $T \cong T'$, wenn sie durch endliche Anwendung folgender Regeln ineinander überführt werden können:*

- *Beliebiges Umordnen der Kinder eines P-Knotens;*
- *Umkehren der Reihenfolge der Kinder eines Q-Knotens.*

Definition 6.6 *Sei T ein echter PQ-Baum, dann ist die Menge der konsistenten Frontiers von T, d.h.:*

$$\textit{consistent}(T) = \{f(T') \ : \ T \cong T'\}\,.$$

Beispielsweise befinden sich dann in der Menge consistent(T) für den Baum aus der Abbildung 6.4: BADCEFGIH, ABGFCDEHI oder HIDCEFGBA.

Definition 6.7 *Sei Σ ein endliches Alphabet und $\mathcal{F} = \{F_1, \ldots, F_k\} \subseteq 2^\Sigma$ eine so genannte Menge von* Restriktionen*, d.h. von Teilmengen von Σ. Dann bezeichnet $\Pi(\Sigma, \mathcal{F})$ die Menge der Permutationen über Σ, in der die Elemente aus F_i für jedes $i \in [1:k]$ konsekutiv vorkommen.*

Mit Hilfe dieser Definitionen können wir nun das Ziel dieses Abschnittes formalisieren. Zu einer gegebenen Menge $\mathcal{F} \subset 2^\Sigma$ von Restriktionen (nämlich den Ergebnissen unserer biologischen Experimente zur Erstellung einer genomischen Karte) wollen wir einen PQ-Baum T mit

$$\text{consistent}(T) = \Pi(\Sigma, \mathcal{F})$$

konstruieren, sofern dies möglich ist.

6.2.2 Konstruktion von PQ-Bäumen

Wir werden versuchen, den gewünschten PQ-Baum für die gegebene Menge von Restriktionen iterativ zu konstruieren, d.h. wir erzeugen eine Folge $T_0, T_1, \ldots, T_k$ von PQ-Bäumen, so dass

$$\text{consistent}(T_i) = \Pi(\Sigma, \{F_1, \ldots, F_i\})$$

gilt. Dabei ist $T_0 = T(\Sigma)$ der PQ-Baum, dessen Wurzel aus einem P-Knoten besteht und an dem n Blätter hängen, die eineindeutig mit den Zeichen aus $\Sigma = \{a_1, \ldots, a_n\}$ markiert sind. Wir müssen daher nur noch eine Prozedur reduce entwickeln, für die $T_i = \text{reduce}(T_{i-1}, F_i)$ gilt.

Prinzipiell werden wir zur Realisierung dieser Prozedur den Baum T_{i-1} von den Blättern zur Wurzel hin durchlaufen, um gleichzeitig die Restriktion F_i einzuarbeiten. Dazu werden alle Blätter, deren Marken in F_i auftauchen markiert und wir werden nur den Teilbaum mit den markierten Blättern bearbeiten. Dazu bestimmen wir zuerst den niedrigsten Knoten $r(T_{i-1}, F_i)$ in T_i, so dass alle Blätter aus F_i in dem an diesem Knoten gewurzelten Teilbaum enthalten sind. Diesen Teilbaum selbst bezeichnen wir mit $T_r(T_{i-1}, S_i)$ als den *reduzierten Teilbaum*.

Weiterhin vereinbaren wir noch den folgenden Sprachgebrauch. Ein Blatt heißt *voll*, wenn es in F_i vorkommt und ansonsten *leer*. Ein innerer Knoten heißt *voll*, wenn alle seine Kinder voll sind. Analog heißt ein innerer Knoten *leer*, wenn alle seine Kinder leer sind. Andernfalls nennen wir den Knoten *partiell*. Im Folgenden werden wir auch Teilbäume als *voll* bzw. *leer* bezeichnen, wenn alle darin enthaltenen Knoten voll bzw. leer sind (was äquivalent dazu ist, dass dessen Wurzel voll bzw. leer ist). Andernfalls nennen wir einen solchen Teilbaum *partiell*.

Da es bei P-Konten nicht auf die Reihenfolge ankommt, wollen wir im Folgenden immer vereinbaren, dass die leeren Kinder und die vollen Kinder eines P-Knotens immer konsekutiv angeordnet sind (siehe Abbildung 6.5).

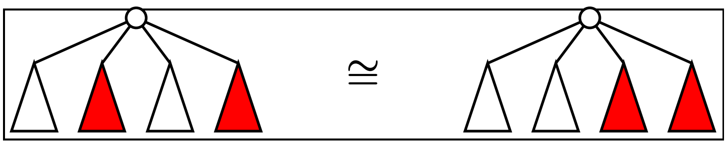


Abbildung 6.5: Skizze: Anordnung leerer und voller Kinder eines P-Knotens

Im Folgenden werden wir volle und partielle Knoten bzw. Teilbäume immer rot kennzeichnen, während leere Knoten bzw. Teilbäume weiß bleiben. Man beachte, dass ein PQ-Baum nie mehr als zwei partielle Knoten besitzen kann, von denen nicht einer ein Nachfahre eines anderen ist. Würde ein PQ-Baum drei partielle Knoten besitzen, von den keiner ein Nachfahre eines anderen ist, dann könnten die gewünschten Permutationen aufgrund der gegebenen Restriktionen nicht konstruiert werden. Die Abbildung 6.6 mag dabei helfen, sich dies klar zu machen.

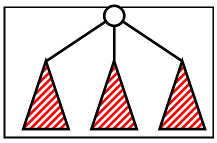


Abbildung 6.6: Skizze: Drei partielle Teilbäume

Im Folgenden werden wir jetzt verschiedene Schablonen beschreiben, die bei unserer bottom-up-Arbeitsweise im reduzierten Teilbaum angewendet werden, um die aktuelle Restriktion einzuarbeiten. Wir werden also immer annehmen, dass die Teilbäume des betrachteten Knoten (oft auch als Wurzel bezeichnet) bereits abgearbeitet sind. Wir werden dabei darauf achten, folgende Einschränkung aufrecht zu erhalten. Wenn ein Knoten partiell ist, wird es ein Q-Knoten sein. Wir werden also nie einen partiellen P-Knoten konstruieren.

6.2.2.1 Schablone P_0

Die Schablone P_0 in Abbildung 6.7 ist sehr einfach. Wir betrachten einen P-Knoten, an dem nur leere Teilbäume hängen. Somit ist nichts zu tun.

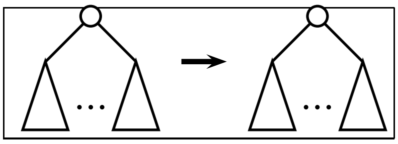


Abbildung 6.7: Skizze: Schablone P_0

6.2.2.2 Schablone P_1

Die Schablone P_1 in Abbildung 6.8 ist auch nicht viel schwerer. Wir betrachten einen P-Knoten, an dem nur volle Unterbäume hängen. Wir markieren daher die Wurzel als voll und gehen weiter bottom-up vor.

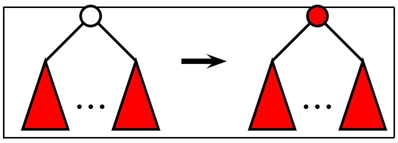


Abbildung 6.8: Skizze: Schablone P_1

6.2.2.3 Schablone P_2

Jetzt betrachten wir einen P-Knoten p, an dem nur volle und leere (also keine partiellen) Teilbäume hängen (siehe Abbildung 6.9). Weiter nehmen wir an, dass der Knoten p die Wurzel des reduzierten Teilbaums T_r ist In diesem Fall fügen wir einen neuen P-Knoten als Kind der Wurzeln ein und hängen alle volle Teilbäume der ursprünglichen Wurzel an diesen Knoten. Das wir die Wurzel des reduzierten Teilbaumes erreicht haben können wir mit der Umordnung des PQ-Baumes aufhören, da nun alle markierten Knoten aus F in den durch den PQ-Baum dargestellten Permutationen konsekutiv sind.

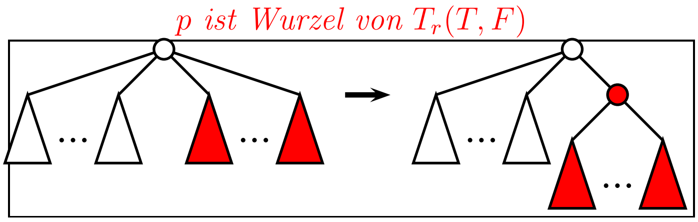


Abbildung 6.9: Skizze: Schablone P_2

Hierbei ist nur zu beachten, dass wir eigentlich nur echte PQ-Bäume konstruieren wollen. Hing also ursprünglich nur ein voller Teilbaum an der Wurzel, so führen wir die oben genannte Transformation nicht aus und belassen alles so wie es war.

In jedem Falle überzeugt man sich leicht, dass alle Frontiers, die nach der Transformation eines äquivalenten PQ-Baumes abgelesen werden könne, auch schon vorher abgelesen werden konnten. Des Weiteren haben wir durch die Transformation erreicht, dass alle Zeichen der aktuell betrachteten Restriktion nach der Transformation konsekutiv auftreten müssen.

6.2.2.4 Schablone P_3

Nun betrachten wir einen P-Knoten, an dem nur volle oder leere Teilbäume hängen, der aber noch nicht die Wurzel der reduzierten Teilbaumes ist (siehe Abbildung 6.10).

Wir führen als neue Wurzel einen Q-Knoten ein. Alle leeren Kinder der ursprünglichen Wurzel belassen wird diesem P-Knoten und machen diesen P-Knoten zu einem Kind der neuen Wurzel. Weiter führen wir einen neuen P-Knoten ein, der ebenfalls ein Kind der neuen Wurzel wird und schenken ihm als Kinder alle vollen Teilbäume der ehemaligen Wurzel.

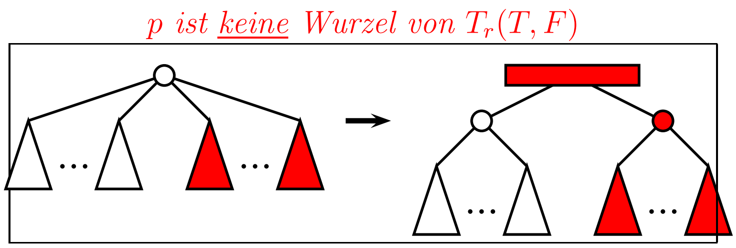


Abbildung 6.10: Skizze: Schablone P_3

Auch hier müssen wir wieder beachten, dass wir einen korrekten PQ-Baum generieren. Gab es vorher nur einen leeren oder einen vollen Unterbaum, so wird das entsprechende Kind der neuen Wurzel nicht wiederverwendet bzw. eingefügt, sondern der leere bzw. volle Unterbaum wird direkt an die neue Wurzel gehängt. Des Weiteren haben wir einen Q-Knoten konstruiert, der nur zwei Kinder besitzt. Dies würde der Definition eines echten PQ-Baumes widersprechen. Da wir jedoch weiter bottom-up den reduzierten Teilbaum abarbeiten müssen, werden wir später noch sehen, dass dieser Q-Knoten mit einem anderen Q-Knoten verschmolzen wird, so dass auch das kein Problem sein wird.

6.2.2.5 Schablone P_4

Betrachten wir nun den Fall, dass die Wurzel p ein P-Knoten ist, der neben leeren und vollen Kindern noch ein partielles Kind hat, das dann ein Q-Knoten sein muss. Dies ist in Abbildung 6.11 illustriert, wobei wir noch annehmen, dass der betrachtete Knoten die Wurzel des reduzierten Teilbaumes ist

Wir werden alle vollen Kinder, die direkt an der Wurzel hängen, unterhalb des partiellen Knotens einreihen. Da der partielle Knoten ein Q-Knoten ist, müssen die vollen Kinder an dem Ende hinzugefügt werden, an dem bereits volle Kinder hängen. Da die Reihenfolge der Kinder, die an der ursprünglichen Wurzel (einem P-Knoten) hingen, egal ist, werden wir die Kinder nicht direkt an den Q-Knoten hängen, sondern erst einen neuen P-Knoten zum äußersten Kind dieses Q-Knotens machen und daran die vollen Teilbäume anhängen. Dies ist natürlich nicht nötig, wenn an der ursprünglichen Wurzel nur ein vollen Teilbaum gehangen hat.

p *ist Wurzel von* $T_r(T, F)$

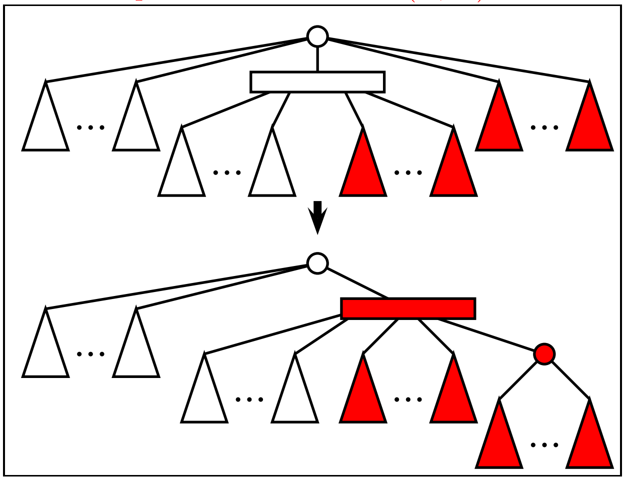


Abbildung 6.11: Skizze: Schablone P_4

Auch hier machen wir uns wieder leicht klar, dass die Einschränkungen der Transformation lediglich die aktuell betrachtete Restriktion widerspiegelt und wir den Baum bzw. seine dargestellten Permutationen nicht mehr einschränken als nötig.

Wir müssen uns jetzt nur noch Gedanken machen, wenn der Q-Knoten im vorigen Schritt aus der Schablone P_3 entstanden ist. Dann hätte dieser Q-Knoten nur zwei Kinder gehabt. Besaß die ehemalige Wurzel p vorher noch einen vollen Teilbaum. so hat sich dieses Problem erledigt, das der Q-Knoten nun noch ein drittes Kind erhält. Hätte p vorher kein volles Kind gehabt (also nur einen partiellen Q-Knoten und lauter leere Bäume als Kinder), dann hätten wir ein Problem, da der Q-Knoten dann weiterhin nur zwei Kinder hätte. In diesem Fall ersetzen wir den Q-Knoten durch einen P-Knoten, da ein Q-Knoten mit zwei Kindern dieselben Permutationen beschreibt wie ein P-Knoten.

6.2.2.6 Schablone P_5

Nun betrachten wir den analogen Fall, dass an der Wurzel ein partielles Kind hängt, aber der betrachtete Knoten nicht die Wurzel des reduzierten Teilbaumes ist. Dies ist in Abbildung 6.12 illustriert.

Wir machen also den Q-Knoten zur neuen Wurzel des betrachteten Teilbaumes und hängen die ehemalige Wurzel des betrachteten Teilbaumes mitsamt seiner leeren Kinder ganz außen am leeren Ende an den Q-Knoten an. Die vollen Kinder der ehemaligen Wurzel des betrachteten Teilbaumes hängen wir am vollen Ende des Q-Knotens über einen neuen P-Knoten an. Man beachte wieder, dass die P-Knoten

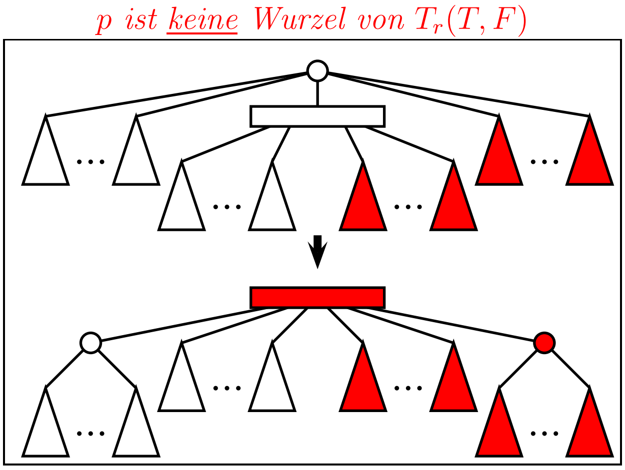


Abbildung 6.12: Skizze: Schablone P_5

nicht benötigt werden, wenn es nur einen leeren bzw. vollen Teilbaum gibt, der an der Wurzel des betrachteten Teilbaumes hing.

Auch hier machen wir uns wieder leicht klar, dass die Einschränkungen der Transformation lediglich die aktuell betrachtete Restriktion widerspiegelt und wir den Baum bzw. seine dargestellten Permutationen nicht mehr einschränken als nötig.

Falls der Q-Knoten vorher aus der Schablone P_3 neu entstanden war, so erhält er nun die benötigten weiteren Kinder, um der Definition eines echten PQ-Baumes zu genügen. Man beachte hierzu nur, dass die Wurzel p vorher mindestens einen leeren oder einen vollen Teilbaum besessen haben muss. Andernfalls hätte der P-Knoten p als Wurzel nur ein Kind besessen, was der Definition eines echten PQ-Baumes widerspricht.

6.2.2.7 Schablone P_6

Es bleibt noch der letzte Fall zu betrachten, dass an die Wurzel des betrachteten Teilbaumes ein P-Knoten ist, an der neben vollen und leeren Teilbäume genau zwei partielle Kinder hängen (die dann wieder Q-Knoten sein müssen). Dies ist in Abbildung 6.13 illustriert.

Man überlegt sich leicht, dass die Wurzel p des betrachteten Teilbaumes dann auch die Wurzel des reduzierten Teilbaumes sein muss, da andernfalls die aktuell betrachtete Restriktion sich nicht mit den Permutationen des bereits konstruierten PQ-Baumes unter ein Dach bringen lässt.

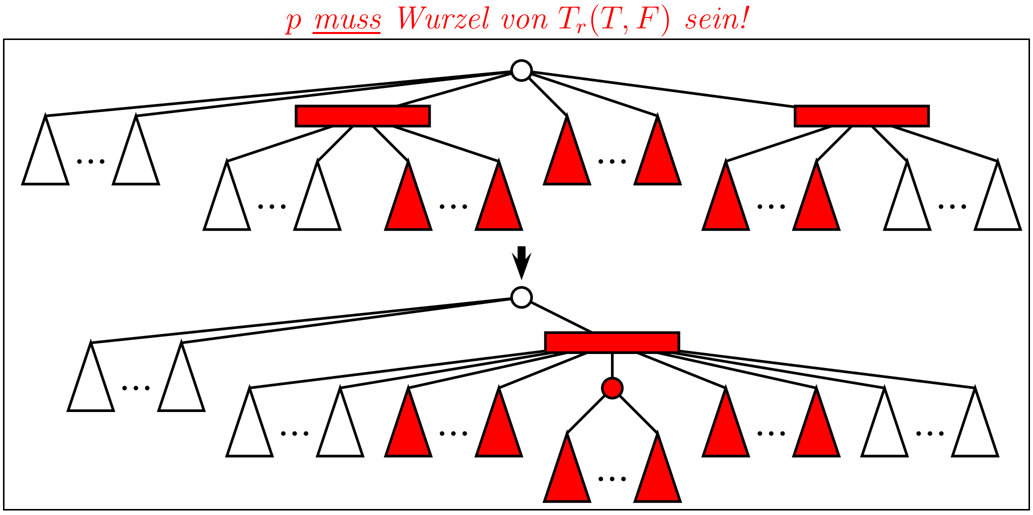


Abbildung 6.13: Skizze: Schablone P_6

Wir vereinen einfach die beiden Q-Knoten zu einem neuen und hängen die vollen Kinder der Wurzel des betrachteten Teilbaumes über eine neu einzuführenden P-Knoten in der Mitte des verschmolzenen Q-Knoten ein.

Falls hier einer oder beide der betrachteten Q-Knoten aus der Schablone P_3 entstanden ist, so erhält er auch hier wieder genügend zusätzliche Kinder, so dass die Eigenschaft eines echten PQ-Baumes wiederhergestellt wird.

6.2.2.8 Schablone Q_0

Nun haben wir alle Schablonen für P-Knoten als Wurzeln angegeben. Es folgen die Schablonen, in denen die Wurzel des betrachteten Teilbaumes ein Q-Knoten ist. Die Schablone Q_0 ist analog zur Schablone P_0 wieder völlig simpel. Alle Kinder sind leer und es ist also nichts zu tun (siehe Abbildung 6.14).

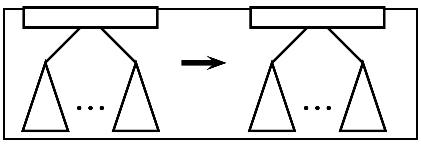


Abbildung 6.14: Skizze: Schablone Q_0

6.2.2.9 Schablone Q_1

Auch die Schablone Q_1 ist völlig analog zur Schablone P_1. Alle Kinder sind voll und daher markieren wir den Q-Knoten als voll und arbeiten uns weiter bottom-up durch den reduzierten Teilbaum (siehe auch Abbildung 6.15).

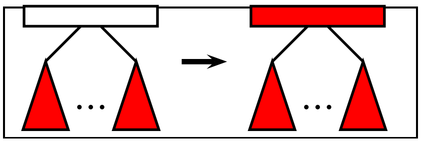


Abbildung 6.15: Skizze: Schablone Q_1

6.2.2.10 Schablone Q_2

Betrachten wir nun den Fall, dass sowohl volle wie leere Teilbäume an einem Q-Knoten hängen. In diesem Fall tun wir gar nichts, denn dann ist die Wurzel ein partieller Q-Knoten. Wir steigen also einfach im Baum weiter auf.

Kommen wir also gleich zu dem Fall, an dem an der Wurzel p des aktuell betrachteten Teilbaumes volle und leere sowie genau ein partieller Q-Knoten hängt. Wir verschmelzen nun einfach den partiellen Q-Knoten mit der Wurzel (die ebenfalls ein Q-Knoten ist), wie in Abbildung 6.16 illustriert. Falls der partielle Q-Knoten aus der Schablone P_3 entstanden ist, erhält er auch her wieder ausreichend viele Kinder.

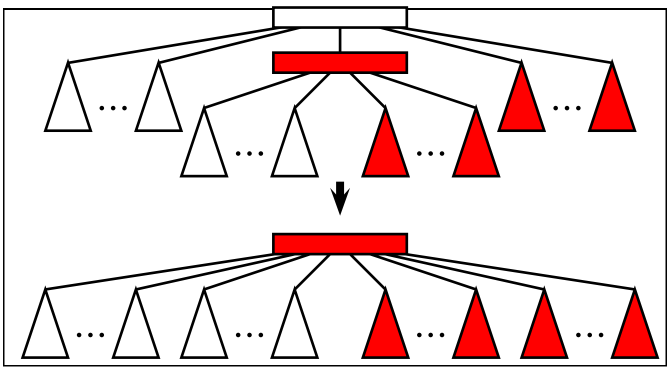


Abbildung 6.16: Skizze: Schablone Q_2

6.2.2.11 Schablone Q_3

Als letzter Fall bleibt der Fall, dass an der Wurzel des aktuell betrachteten Teilbaumes zwei partielle Q-Knoten hängen (sowie volle und leere Teilbäume). Auch hier vereinen wir die drei Q-Knoten zu einem neuen wie in Abbildung 6.17 angegeben. In diesem Fall muss der betrachtete Q-Knoten bereits die Wurzel des reduzierten Teilbaumes und die Prozedur bricht ab.

p muss Wurzel sein! $\Rightarrow$ *Prozedur abbrechen!*

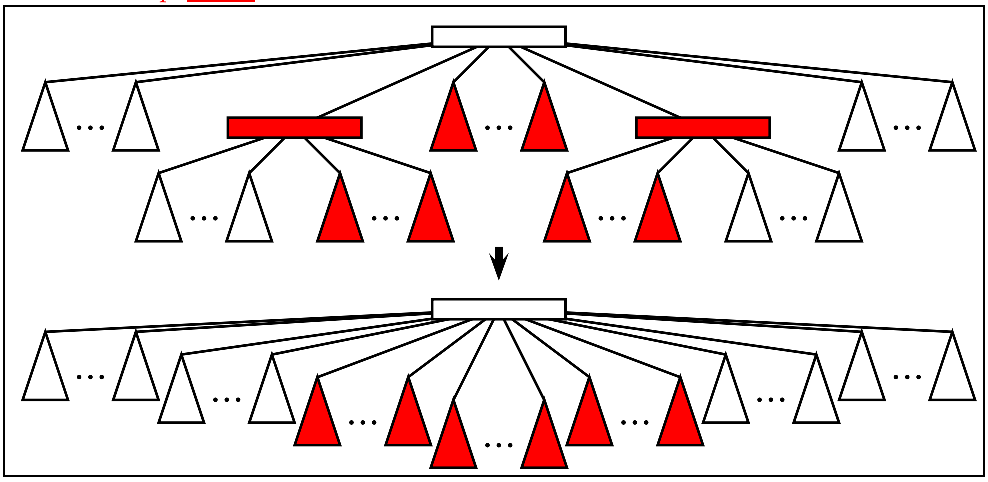


Abbildung 6.17: Skizze: Schablone Q_3

In der Abbildung 6.18 auf Seite 235 ist ein Beispiel zur Konstruktion eines PQ-Baumes für die Restriktionsmenge

$$\{\{B, E\}, \{B, F\}, \{A, C, F, G\}, \{A, C\}, \{A, C, F\}, \{D, G\}\}$$

angegeben.

6.2.3 Korrektheit

In diesem Abschnitt wollen wir kurz die Korrektheit beweisen, d.h. dass der konstruierte PQ-Baum tatsächlich die gewünschte Menge von Permutationen bezüglich der vorgegebenen Restriktionen darstellt. Dazu definieren wir den *universellen PQ-Baum* $T(\Sigma, F)$ für ein Alphabet Σ und eine Restriktion $F = \{a_{i_1}, \ldots, a_{i_r}\}$. Die Wurzel des universellen PQ-Baumes ist ein P-Knoten an dem sich lauter Blätter, je eines für jedes Zeichen aus $\Sigma \setminus F$, und ein weiterer P-Knoten hängen, an dem sich seinerseits lauter Blätter befinden, je eines für jedes Element aus F.

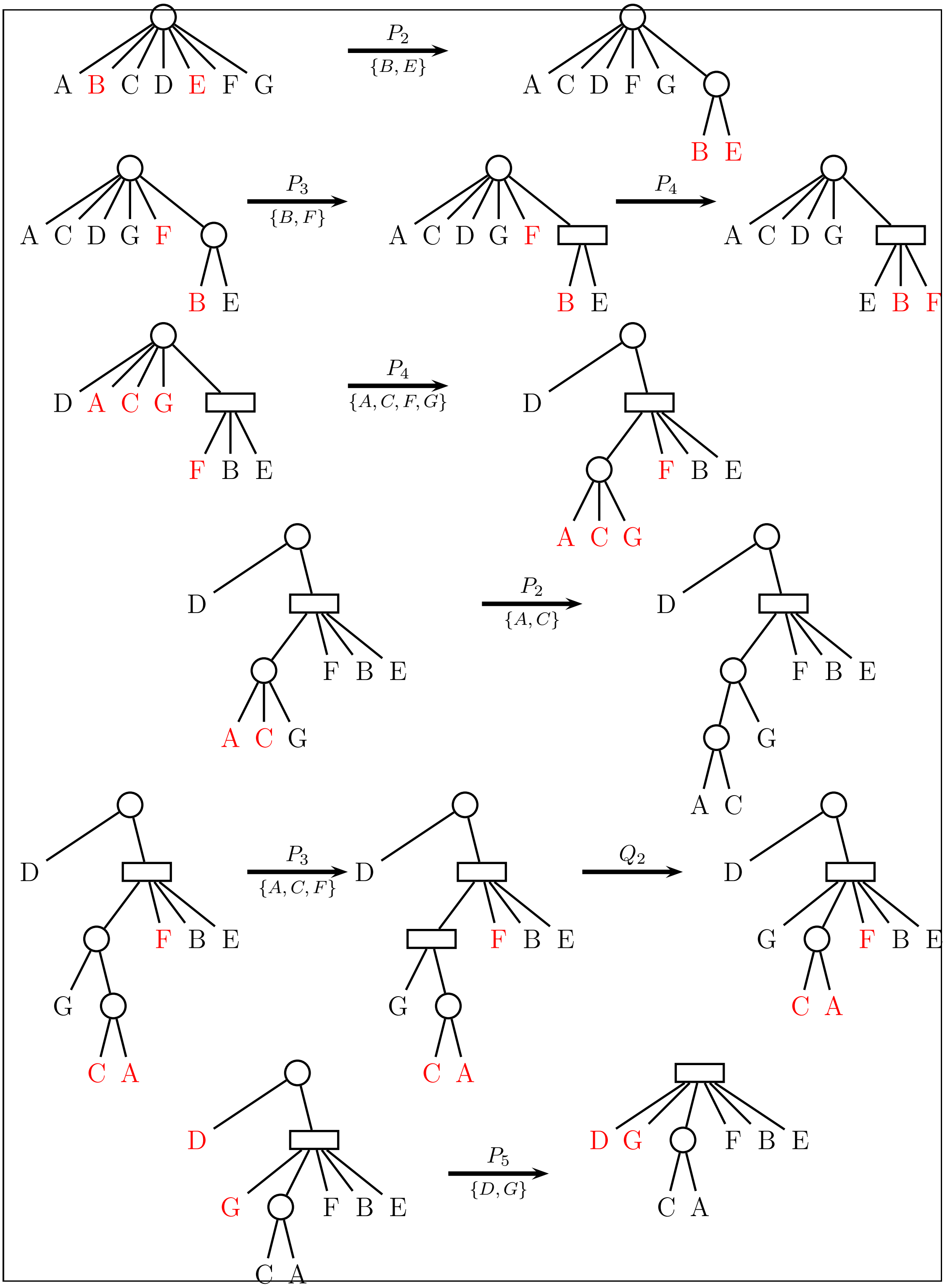


Abbildung 6.18: Beispiel: Konstruktion eines PQ-Baumes

Theorem 6.8 *Sei T eine beliebiger echter PQ-Baum und $F \subseteq \Sigma$. Dann gilt:*

$$\text{consistent}(\text{reduce}(\text{T}, \text{F})) = \text{consistent}(T) \cap \text{consistent}(T(\Sigma, F)).$$

Beweis: Zuerst führen wir zwei Abkürzungen ein:

$$\begin{aligned} A &:= \text{consistent}(\text{reduce}(\text{T}, \text{F})) \\ B &:= \text{consistent}(T) \cap \text{consistent}(T(\Sigma, F)) \end{aligned}$$

$\boldsymbol{A \subseteq B}$ **:** Ist $A = \emptyset$, so ist nichts zu zeigen. Ansonsten existiert ein

$$\pi \in \text{consistent}(\text{reduce}(T, F)) \quad \text{und} \quad T' \cong \text{reduce}(T, F) \quad \text{mit} \quad f(T') = \pi.$$

Nach Konstruktion gilt $\pi \in \text{consistent}(T)$. Andererseits gilt nach Konstruktion für jeden erfolgreich abgearbeiteten Knoten x eine der folgenden Aussagen:

- x ist ein Blatt und $x \in F$,
- x ist ein voller P-Knoten,
- x ist ein Q-Knoten, dessen markierte Unterbäume alle konsekutiv vorkommen und die partiellen markierten Unterbäume (sofern vorhanden) am Rand diese konsekutiven Bereichs vorkommen.

Daraus folgt unmittelbar, dass $\pi \in \text{consistent}(T(\Sigma.F))$.

$\boldsymbol{B \subseteq A}$ **:** Sei also $\pi \in B$. Sei T' so gewählt, dass $T' \cong T$ und $f(T') = \pi$. nach Voraussetzung kommen die Zeichen aus F in π hintereinander vor. Somit hat im reduzierten Teilbaum $T_r(T', F)$ jeder Knoten außer der Wurzel maximal ein partielles Kind und die Wurzel maximale zwei partielle Kinder. Jeder partielle Knoten wird nach Konstruktion durch einen Q-Knoten ersetzt, dessen Kinnder entweder alle voll oder leer sind und deren volle Unterbäume konsekutiv vorkommen. Damit ist bei der bottom-up-Vorgehensweise immer eine Schablone anwendbar und es gilt $\pi \in \text{consistent}(\text{reduce}(T', F))$. Damit ist auch $\pi \in \text{consistent}(\text{reduce}(T, F))$. ■

6.2.4 Implementierung

An dieser Stelle müssen wir noch ein paar Hinweise zur effizienten Implementierung geben, da mit ein paar Tricks die Laufzeit zur Generierung von PQ-Bäumen drastisch gesenkt werden kann. Überlegen wir uns zuerst die Eingabegröße. Die Eingabe selbst ist $(\Sigma, \mathcal{F})$ und somit ist die Eingabegröße $\Theta(|\Sigma| + \sum_{F \in \mathcal{F}} |F|)$.

Betrachten wir den Baum T auf den wir die Operation reduce(T, F) loslassen. Mit $T_r(T, F)$ bezeichnen wir den reduzierten Teilbaum von T bezüglich F. Dieser ist über die niedrigste Wurzel beschrieben, so dass alle aus F markierten Blätter Nachfahren dieser Wurzel sind. Der Baum $T_r(T, F)$ selbst besteht aus allen Nachfahren dieser Wurzel. Offensichtlich läuft die Hauptarbeit innerhalb dieses Teilbaumes ab. Diesee Teilbaum von T sind in Abbildung 6.19 schematisch dargestellt.

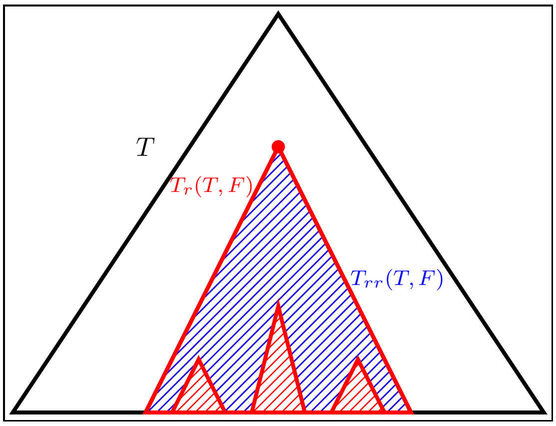


Abbildung 6.19: Skizze: Bearbeitete Teilbäume bei reduce(T, F)

Aber selbst bei nur zwei markierten Blättern, kann dieser Teilbaum sehr groß werden. Also betrachten wir den so genannten *relevanten reduzierten Teilbaum* $T_{rr}(T, F)$ Dieser besteht aus dem kleinsten zusammenhängenden Teilgraphen von T, der alle markierten Blätter aus F enthält. Offensichtlich ist $T_{rr}(T, F)$ ein Teilbaum von $T_r(T, F)$, wobei die Wurzeln der beiden Teilbäume von T dieselben sind. Man kann auch sagen, dass der relevante reduzierte Teilbaum aus dem reduzierten Teilbaum entsteht, indem man leer Teilbäume herausschneidet. Diese Teilbäume von T sind in Abbildung 6.19 schematisch dargestellt.

Wir werden zeigen, dass die gesamte Arbeit im Wesentlichen im Teilbaum $T_{rr}(T, F)$ erledigt wird und diese somit für eine reduce-Operation proportional zu $|T_{rr}(T, F)|$ ist. Somit ergibt sich für die Konstruktion eines PQ-Baumes für eine gegebene Menge $\mathcal{F} = \{F_1, \ldots, F_n\}$ von Restriktionen die folgende Laufzeit von

$$\sum_{i=1}^{n} O(|T_{rr}(T_{i-1}, F_i)|),$$

wobei $T_0 = T(\Sigma)$ ist und $T_i = \text{reduce}(T_{i-1}, F_i)$. Wir müssen uns jetzt noch um zwei Dinge Gedanken machen: Wie kann man die obige Laufzeit besser, anschaulicher abschätzen und wie kann man den relevanten reduzierten Teilbaum $T_{rr}(T, F)$ in Zeit $O(|T_{rr}(T, F)|)$ ermitteln.

Zuerst kümmern wir uns um die Bestimmung des relevanten reduzierten Teilbaumes. Dazu müssen wir uns aber erst noch ein paar genauere Gedanken zur Implementierung des PQ-Baumes selbst machen. Die Kinder eines Knotens werden als doppelt verkettete Liste abgespeichert, da ja für die Anzahl der Kinder keine obere Schranke a priori bekannt ist. Bei den Kindern eines P-Knoten ist die Reihenfolge, in der sie in der doppelt verketteten Liste abgespeichert werden, beliebig. Bei den Kindern eines Q-Knoten respektiert die Reihenfolge innerhalb der doppelt verketteten Liste gerade die Ordnung, in der sie unter dem Q-Knoten hängen.

Zusätzlich werden wir zum bottom-up Aufsteigen auch noch von jedem Knoten den zugehörigen Elter wissen wollen. Leider wird sich herausstellen, dass es zu aufwendig ist, für jeden Knoten einen Verweis zu seinem Elter aktuell zu halten. Daher werden wie folgt vorgehen. Ein Kind eines P-Knotens erhält jeweils eine Verweis auf seinen Elter. Bei Q-Knoten werden nur die beiden äußersten Kinder einen Verweis auf ihren Elter erhalten. Wir werden im Folgenden sehen, dass dies völlig ausreichend sein wird.

In Abbildung 6.20 ist der Algorithmus zum Ermitteln des relevanten reduzierten Teilbaumes angegeben. Prinzipiell versuchen wir ausgehend von der Menge der markierten Blätter aus F einen zusammenhängen Teilgraphen von T zu konstruieren, indem wir mit Hilfe der Verweise auf die Eltern im Baum T von den Blätter aus F nach oben laufen.

Um diesen Algorithmus genauer verstehen zu können, müssen wir erst noch ein paar Notationen vereinbaren. Wir halten zwei Listen als FIFO-Queue vor: die Menge free der so genannten *freien Konten* und eine Menge blocked der so genannten *blockierten Knoten*. Dazu müssen wir jedoch zuerst noch *aktive Knoten* definieren.

Ein Knoten heißt aktiv, wenn wir wissen, dass er ein Vorfahr eines markierten Blattes aus F ist. Ein aktiver Knoten heißt frei, wenn die Kante zu seinem Elter noch nicht betrachtet wurde. Ein aktiver Knoten ist blockiert, wenn wir festgestellt haben, dass wir seinen Elter nicht kennen. Es kann also durchaus freie Knoten geben, die keinen Verweis auf ihren Elter haben oder deren Eltern selbst schon frei sind (wir haben dies nur noch nicht bemerkt). Um die Notation einfacher zu halten, werden wir blockierte Knoten nicht als frei bezeichen (auch wenn diese nach obiger Definition eigentlich der Fall ist).

Wenn wir jetzt versuchen den kleinsten zusammenhängenden Teilbaum, der alle markierten Blätter enthält, konstruieren, gehen wir bottom-up durch den Baum und konstruieren dabei viele kleine Teilbäume, die durch Verschmelzen letztendlich im Wesentlichen den relevanten reduzierten Teilbaum ergeben. Zu Beginn besteht diese Menge der Teilbäume aus allen markierten Blättern.

Ein Folge von blockierten Knoten, die aufeinander folgende Kinder desselben Knotens sind (der dann ein Q-Knoten sein muss), nennen wir einen *Sektor*. Beachte,

```
FIND_TREE (tree T, set F)
{
    int sectors = 0;    set free, blocked;    for all (f ∈ F) do free.add(f);
    while (free.size() + sectors > 1)
    {
        if (free.is_empty()) return (∅, ∅);
        else
        {
            v = free.remove_FIFO();
            if (parent(v) ≠ nil)
            {
                if (parent(v) ∉ V(T_rr))
                {
                    V(T_rr) = V(T_rr) ∪ {parent(v)};
                    free.add(parent(v));
                }
                E(T_rr) = E(T_rr) ∪ {{v, parent(v)}};
            }
            else
            {
                blocked.add(v);
                if (∃x ∈ N(v) s.t. parent(x) ≠ nil)
                {
                    let y s.t. x ⇌ v ⇌ y;
                    let S be the sector containing v;
                    for all (s ∈ S) do
                    {
                        blocked.remove(s);
                        parent(s) = parent(x);
                        E(T_rr) = E(T_rr) ∪ {{s, parent(s)}};
                    }
                    if (y ∈ blocked) sectors−−;
                }
                elsif (both neighbors of v are blocked) sectors−−;
                elsif (both neighbors of v are not blocked) sectors++;
            }
        }
    }
    return T_rr;
}
```

Abbildung 6.20: Algorithmus: Ermittlung von $T_{rr}(T, F)$

dass ein Sektor nie eines der äußersten Kinder eines Q-Knoten enthalten kann, da diese nach Definition frei sind.

Zuerst überlegen wir uns, wann wir die Prozedur abbrechen. Wenn es nur noch einen freien Knoten und keine blockierten Knoten (und damit auch keine Sektoren) mehr gibt, brechen wir ab. Dann haben wir entweder die Wurzel des relevanten reduzierten Teilbaumes gefunden, oder wir befinden uns mit der freien Wurzel bereits auf dem Weg von der gesuchten Wurzel zur Wurzel des gesamtbaumes T. Wir wissen ja leider nicht in welcher Reihenfolge wir die Knoten des relevanten reduzierten Teilbaumes aufsuchen. Es kann durchaus passieren, dass wir die Wurzel recht schnell finden und den restlichen Teil des Baumes noch gar nicht richtig untersucht haben. Dies passiert insbesondere dann, wenn an der Wurzel bereits ein Blatt hängt. Andererseits brechen wir ab, wenn wir nur noch einen Sektor bearbeiten. Der Elter der Knoten dieses Sektors muss dann die gesuchte Wurzel des relevanten reduzierten Teilbaumes sein.

Wenn immer wir mindestens zwei Sektoren und keine freie Wurzel mehr besitzen, ist klar, dass wir im Fehlerfall sind, d.h für die gegebene Menge $\mathcal{F}$ von Restriktionen kann es keinen korrespondierenden PQ-Baum geben. Andernfalls müssten wir die Möglichkeit haben, diese beide Sektoren mithilfe von freien Wurzeln zu verschmelzen.

Was tut unser Algorithmus also, wenn es noch freie Wurzeln gibt? Er nimmt eine solche freie Wurzel v her und teste, ob der Elter von v bekannt ist. Falls ja, fügt er die Kante zum Elter in den relevanten reduzierten Teilbaum ein. Ist der Elter selbst noch nicht im relevanten reduzierten Teilbaum enthalten, so wird auch dieser darin aufgenommen und der Elter selbst als frei markiert.

Andernfalls wird der betrachtete Knoten v als blockiert erkannt. Jetzt müssen wir nur die Anzahl der Sektoren aktualisieren. Dazu stellen wir zunächst fest, ob v ein direktes Geschwister (Nachbar in der doppelt verketteten Liste) besitzt, der seinen Elter schon kennt. Wenn ja, dann sei y das andere direkte Geschwister von v (man überlege sich, dass dieses existieren muss). Die Folge (x, v, y) kommt also so oder in umgekehrter Reihenfolge in der doppelt verketteten Liste der Geschwister vor. Mit S bezeichnen wir jetzt den Sektor, der v enthält (wir wir diesen bestimmen, ist im Algorithmus nicht explizit angegeben und die technischen Details seien dem Leser überlassen).

Da S nun mit v einen blockierten Knoten enthält, der eine Geschwister hat, der seinen Elter kennt, können wir jetzt auch allen Knoten dieses Sektors S seinen Elter zuweisen und die die entsprechenden Kanten in den relevanten reduzierten Teilbaum aufnehmen. War y vorher blockiert, so reduziert sich die Anzahl der Sektoren um eins, da alle Knoten im Sektor von y jetzt ihren Elter kennen

Es bleibt der Fall übrig, wo kein direktes Geschwister von v seinen Elter kennt. In diesem Fall muss jetzt nur noch die Anzahl der Sektoren aktualisiert werden. Ist v

ein isolierter blockierte Knoten (besitzt also kein blockiertes Geschwister), so muss die Anzahl der Sektoren um eine erhöht werden. Waren beide Geschwister blockiert, so werden diese Sektoren mithilfe von v zu einem verschmolzen und die Anzahl der Sektoren sinkt um eins. War genau ein direktes Geschwister blockiert, so erweitert v diesen Sektor und die Anzahl der Sektoren bleibt unverändert.

Damit haben wir die Korrektheit des Algorithmus zur Ermittlung des relevanten reduzierten Teilbaumes bewiesen. Bleibt am Ende des Algorithmus eine frei Wurzel oder ein Sektor übrig, so haben wir den relevanten reduzierten Teilbaum im Wesentlichen gefunden. Im ersten Fall befinden wir uns mit der freien Wurzel auf dem Pfad von der eigentlichen Wurzel zur Wurzel der Gesamtbaumes. Durch Absteigen können wir die gesuchte Wurzel als den Knoten identifizieren, an dem eine Verzweigung auftritt. Im zweiten Fall ist, wie gesagt, der Elter der blockierten Knoten im gefunden Sektor die gesuchte Wurzel.

6.2.5 Laufzeitanalyse

Wir haben die Lauzeit bereits mit

$$\sum_{i=1}^{n} O(|T_{rr}(T_{i-1}, F_i)|)$$

abgeschätzt, wobei $T_0 = T(\Sigma, \emptyset)$ ist und $T_i = \text{reduce}(T_{i-1}, F_i)$. Zuerst wollen wir uns noch wirklich überlegen, dass diese Behauptung stimmt. Das einzige Problem hierbei ist, dass ja aus dem relevanten reduzierte Teilbaum Kanten herausführen, an denen andere Knoten des reduzierten Teilbaumes hängen, die jedoch nicht zum relevanten reduzierten Teilbaum gehören (in Abbildung 6.19 sind dies Kanten aus dem blauen in den roten Bereich). Wenn wir für jede solche Kante nachher bei der Anwendung der Schablonen den Elterverweis in den relevanten reduzierten Teilbaum aktualisieren müssten, hätten wir ein Problem. Dies ist jedoch wie gleich sehen werden, glücklicherweise nicht der Fall.

6.2.5.1 Die Schablonen P_0, P_1, Q_0 und Q_1

Zuerst bemerken wir, dass die Schablonen P_0 und Q_0 nie angewendet werden, da diese erstens nichts verändern und zweitens nur außerhalb des relevanten reduzierten Teilbaums anwendbar sind. Bei den Schablonen P_1 und Q_1 sind keine Veränderungen des eigentlichen PQ-Baumes durchzuführen.

6.2.5.2 Die Schablone P_2

Bei der Schablone P_2 (siehe Abbildung 6.9 auf Seite 228) bleiben die Knoten außerhalb des relevanten reduzierten Teilbaumes unverändert und auch die Wurzel ändert sich nicht. Wir müssen nur die Wurzeln der vollen Teilbäume und den neuen Knoten aktualisieren.

6.2.5.3 Die Schablone P_3

Bei der Schablone P_3 (siehe Abbildung 6.10 auf Seite 229) verwenden wir den Trick, dass wir die alte Wurzel als Wurzel der leeren Teilbäume belassen. Somit muss ebenfalls nur an den Wurzeln der vollen Teilbäumen und der neu eingeführten Knoten etwas verändert werden. Dass wir dabei auch den Elter-Zeiger der alten Wurzel des betrachteten Teilbaumes aktualisieren müssen ist nicht weiter tragisch, da dies nur konstante Kosten pro Schablone (und somit pro Knoten des betrachteten relevanten reduzierten Teilbaumes) verursacht.

6.2.5.4 Die Schablone P_4

Bei der Schablone P_4 (siehe Abbildung 6.11 auf Seite 230) ist dies wieder offensichtlich, da wir nur ein paar volle Teilbäume umhängen und einen neuen P-Knoten einführen.

6.2.5.5 Die Schablone P_5

Bei der Schablone P_5 (siehe Abbildung 6.12 auf Seite 231) verwenden wir denselben Trick wie bei Schablone P_3. Die alte Wurzel mitsamt ihrer Kinder wird umgehängt, so dass die eigentliche Arbeit an der vollen und neuen Knoten stattfindet.

6.2.5.6 Die Schablone P_6

Bei der Schablone P_6 (siehe Abbildung 6.13 auf Seite 232) gilt dasselbe. Hier werden auch zwei Q-Knoten verschmolzen und ein P-Knoten in deren Kinderliste mitaufgenommen. Da wir die Menge der Kinder als doppelt verkettet Liste implementiert haben, ist dies ebenfalls wieder mit konstantem Aufwand realisierbar.

6.2.5.7 Die Schablone Q_2

Bei der Schablone Q_2 (siehe Abbildung 6.16 auf Seite 233) wird nur ein Q-Knoten in einen anderen Knoten hineingeschoben. Da die Kinder eines Knoten als doppelt verkettete Liste implementiert ist, kann dies in konstanter Zeit geschehen.

Einziges Problem ist die Aktualisierung der Kinder des Kinder-Q-Knotens. Würde jedes Kind einen Verweis auf seinen Elter besitzen, so könnte dies teuer werden. Da wir dies aber nur für die äußersten Kinder verlangen, müssen nur von den äußersten Kindern des Kinder-Q-Knotens die Elter-Information eliminiert werden, was sich in konstanter Zeit realisieren lässt. Alle inneren Kinder eines Q-Knotens sollen ja keine Informationen über ihren Elter besitzen. Ansonsten könnte nach ein paar Umorganisationen des PQ-Baumes diese Information falsch sein. Da ist dann keine Information besser als eine falsche.

6.2.5.8 Die Schablone Q_3

Bei der Schablone Q_3 (siehe Abbildung 6.17 auf Seite 234) gilt die Argumentation von der Schablone Q_2 analog.

6.2.5.9 Der Pfad zur Wurzel

Zum Schluss müssen wir uns nur noch überlegen, dass wir eventuell Zeit verbraten, wenn wir auf dem Weg von der Wurzel des relevanten reduzierten Teilbaumes zur eigentlichen Wurzel des Baumes weit nach oben laufen. Dieser Pfad könnte wesentlich größer sein als die Größe des relevanten reduzierten Teilbaumes.

Hierbei hilft uns jedoch, dass wir die Knoten aus der Menge free in FIFO-Manier (first-in-first-out) entfernen. Das bedeutet, bevor wir auf diesem Wurzelweg einen Knoten nach oben steigen, werden zunächst alle anderen freien Knoten betrachtet. Dies ist immer mindestens ein anderer. Andernfalls gäbe es nur einen freien Knoten und einen Sektor. Aber da der freie Knoten auf dem Weg von der Wurzel des relevanten reduzierten Teilbaumes zur Wurzel des Baumes könnte den blockierten Sektor nie befreien. In diesem Fall könnten wir zwar den ganzen Weg bis zur Wurzel hinauflaufen, aber dann gäbe es keine Lösung und ein einmaliges Durchlaufen des Gesamt-Baumes können wir uns leisten.

Sind also immer mindestens zwei freie Knoten in der freien Menge. Somit wird beim Hinauflaufen jeweils der relevante reduzierte Teilbaum um eins vergrößert. Damit können wir auf dem Weg von der relevanten reduzierten Wurzel zur Wurzel des Baumes nur so viele Knoten nach oben ablaufen wie es insgesamt Knoten im relevanten reduzierten teilbaum geben kann. Diese zusätzlichen Faktor können wir jedoch in unserer Groß-O-Notation verstecken.

6.2.6 Anzahlbestimmung angewendeter Schablonen

Da die Anzahl die Knoten im relevanten reduzierten Teilbaum gleich der angewendete Schablonen ist, werden wir für die Laufzeitabschätzung die Anzahl der angewendeten Schablonen abzählen bzw. abschätzen. Mit $\sharp P_i$ bzw. $\sharp Q_i$ bezeichnen wir die Anzahl der angewendeten Schablonen p_i bzw. Q_i zur Konstruktion des PQ-Baumes für $\Pi(\Sigma, \mathcal{F})$.

6.2.6.1 Bestimmung von $\sharp P_0$ und $\sharp Q_0$

Diese Schablonen werden wie bereits erwähnt nie wirklich angewendet.

6.2.6.2 Bestimmung von $\sharp P_1$ und $\sharp Q_1$

Man überlegt sich leicht, dass solche Schablonen nur in Teilbäumen angewendet werden kann, in denen alle Blätter markiert sind. Da nach Lemma 6.3 die Anzahl der inneren Knoten durch die Anzahl der markierten Blätter beschränkt sind, gilt:

$$\sharp P_1 + \sharp Q_1 = O\left(\sum_{F \in \mathcal{F}} |F|\right).$$

6.2.6.3 Bestimmung von $\sharp P_2$, $\sharp P_4$, $\sharp P_6$ und $\sharp Q_3$

Dann nach diesen Schablonen die Prozedur reduce(T, F) abgeschlossen ist, können diese nur einmal für jede Restriktion angewendet werden uns daher gilt:

$$\sharp P_2 + \sharp P_4 + \sharp P_6 + \sharp Q_3 = O(|\mathcal{F}|).$$

6.2.6.4 Bestimmung von $\sharp P_3$

Diese Schablone generiert einen neuen partiellen Q-Knoten, der vorher noch nicht da war (siehe auch Abbildung 6.10). Da in einem PQ-Baum mit mehr als zwei partiellen Q-Knoten (die nicht Vorfahr eines anderen sind) auftreten können und partielle Q-Knoten nicht wieder verschwinden können, kann für jede Anwendung reduce(T, F) nur zweimal die Schablone P_3 angewendet werden. Daher gilt

$$\sharp P_3 \leq 2|\mathcal{F}| = O(|\mathcal{F}|).$$

6.2.6.5 Bestimmung von $\sharp P_5 + \sharp Q_2$

Hierfür definieren zunächst einmal recht willkürlich die *Norm eines PQ-Baumes* wie folgt: Die Norm eine PQ-Baumes T, in Zeichen $\|T\|$, ist die Summe aus der Anzahl der Q-Knoten plus der Anzahl der inneren Knoten von T, die Kinder eines P-Knotens sind. Man beachte, dass Q-Knoten in der Norm zweimal gezählt werden können, nämlich genau dann, wenn sie ein Kind eines P-Knotens sind.

Zuerst halten wir ein paar elementare Eigenschaften dieser Norm fest:

1. Es gilt $\|T\| \geq 0$ für alle PQ-Bäume T;
2. $\|T(\Sigma)\| = 0$;
3. Die Anwendung einer beliebige Schablone erhöht die Norm um maximal eins, d.h $\|S(T)\| \leq \|T\| + 1$ für alle PQ-Bäume T, wobei $S(T)$ der PQ-Baum ist, der nach Ausführung einer Schablone S entsteht.
4. Die Schablonen P_5 und Q_2 erniedrigen die Norm um mindestens eins, d.h. $\|S(T)\| \leq \|T\| - 1$ für alle PQ-Bäume T, wobei $S(T)$ der PQ-Baum ist, der nach Ausführung einer Schablone $S \in \{P_5, Q_2\}$ entsteht.

Die ersten beiden Eigenschaften folgen unmittelbar aus der Definition der Norm. Die letzten beiden Eigenschaften werden durch eine genaue Inspektion der Schablonen klar (dem Leser sei explizit empfohlen, dies zu verifizieren).

Da wir mit den Schablonen P_5 und Q_2 die Norm ganzzahlig erniedrigen und mit jeder anderen Schablone die Norm ganzzahlig um maximal 1 erhöhen, können die Schablonen P_5 und Q_2 nur so oft angewendet werden, wie die anderen. Grob gesagt, es kann nur das weggenommen werden, was schon einmal hingelegt wurde. Es gilt also:

$$\begin{aligned} \sharp P_5 + \sharp Q_2 &\leq \sharp P_1 + \sharp P_2 + \sharp P_3 + \sharp P_4 + \sharp P_6 + \sharp Q_1 + \sharp Q_3 \\ &= O\left(|\mathcal{F}| + \sum_{F \in \mathcal{F}} |F|\right) \\ &= O\left(\sum_{F \in \mathcal{F}} |F|\right). \end{aligned}$$

Die letzte Gleichung folgt aus der Annahme, dass jedes Alphabetsymbol zumindest in einer Restriktion auftritt. Im Allgemeinen kann man dies zwar nicht annehmen, aber in unserem Kontext der genomischen Kartierung ist dies durchaus sinnvoll, da Landmarks die in keinem Fragment auftreten, erst gar nicht berücksichtigt werden.

Damit haben wir die Laufzeit für einen erfolgreichen Fall berechnet. Wir müssen uns nur noch überlegen, was im erfolglosen Fall passiert, wenn also der leere PQ-Baum die Lösung darstellt. In diesem Fall berechnen wir zuerst für eine Teilmenge $\mathcal{F}' \subsetneq \mathcal{F}$ einen konsistenten PQ-Baum. Bei Hinzunahme der Restriktion F stellen wir fest, dass $\mathcal{F}'' := \mathcal{F}' \cup \{F\}$ keine Darstellung durch einen PQ-Baum besitzt. Für die Berechnung des PQ-Baumes von $\mathcal{F}'$ benötigen wir, wie wir eben gezeigt haben:

$$O\left(|\Sigma| + \sum_{F \in \mathcal{F}'} |F|\right) = O\left(|\Sigma| + \sum_{F \in \mathcal{F}} |F|\right).$$

Um festzustellen, dass $\mathcal{F}''$ keine Darstellung durch einen PQ-Baum besitzt, müssen wir im schlimmsten Fall den PQ-Baum T' für $\mathcal{F}'$ durchlaufen. Da dieser ein PQ-Baum ist und nach Lemma 6.3 maximal $|\Sigma|$ innere Knoten besitzt, da er genau $|\Sigma|$ Blätter besitzt, folgt, dass der Aufwand höchsten $O(|\Sigma|)$ ist. Fassen wir das Ergebnis noch einmal zusammen.

Theorem 6.9 *Die Menge* $\Pi(\Sigma, \mathcal{F})$ *kann durch einen PQ-Baum mit*

$$\text{consistent}(T) = \Pi(\Sigma, \mathcal{F})$$

dargestellt und in Zeit $O\left(|\Sigma| + \sum_{F \in \mathcal{F}} |F|\right)$ *berechnet werden*

Somit haben wir einen effizienten Algorithmus zur genomischen Kartierung gefunden, wenn wir voraussetzten, dass die Experimente fehlerfrei sind. In der Regel wird dies jedoch nicht der Fall sein, wie wir das schon am Ende des ersten Abschnitt dieses Kapitels angemerkt haben. Wollten wir False Negatives berücksichtigen, dann müssten wir erlauben, dass die Zeichen einer Restriktion nicht konsekutiv in einer Permutation auftauchen müssten, sondern durchaus wenige (ein oder zwei) sehr kurze Lücken (von ein oder zwei Zeichen) auftreten dürften. Für False Positives müssten wir zudem wenige einzelne isolierte Zeichen einer Restriktion erlauben. Und für Chimeric Clones müsste auch eine oder zwei zusätzliche größere Lücken erlaubt sein. Leider hat sich gezeigt, dass solche modifizierten Problemstellung bereits $\mathcal{NP}$-hart sind und somit nicht mehr effizient lösbar sind.

6.3 Intervall-Graphen

In diesem Abschnitt wollen wir eine andere Modellierung zur genomischen Kartierung vorstellen. Wie wir im nächsten Abschnitt sehen werden, hat diese Modellierung den Vorteil, dass wir Fehler hier leichter mitmodellieren können.

6.3.1 Definition von Intervall-Graphen

Zuerst benötigen wir die Definition eines Intervall-Graphen.

Definition 6.10 *Ein Menge $\mathcal{I} = \{[\ell_i, r_i] \subset \mathbb{R} : i \in [1:n]\}$ von reellen Intervallen $[\ell_i, r_i]$ mit $\ell_i < r_i$ für alle $i \in [1:n]$ heißt* Intervall-Darstellung.

Der zugehörige Graph $G(\mathcal{I}) = (V, E)$ ist gegeben durch

- $V = \mathcal{I} \cong [1:n]$,
- $E = \{\{I, I'\} : I, I' \in \mathcal{I} \wedge I \cap I' \neq \emptyset\}$.

Ein Graph G heißt Intervall-Graph *(engl.* interval graph*), wenn es eine Intervall-Darstellung $\mathcal{I}$ gibt, so dass $G \cong G(\mathcal{I})$.*

In Abbildung 6.21 ist ein Beispiel eines Intervall-Graphen samt seiner zugehörigen Intervall-Darstellung gegeben.

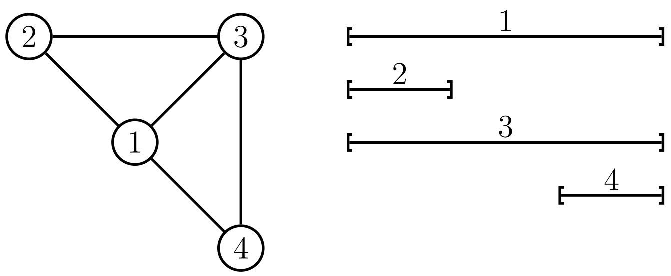


Abbildung 6.21: Beispiel: Ein Intervall-Graph samt zugehöriger Intervall-Darstellung

Zuerst bemerken wir, dass die Intervalle so gewählt werden können, dass die Intervallgrenzen paarweise verschieden sind, d.h. für einen Intervall-Graphen G kann eine Intervall-Darstellung $\mathcal{I} = \{[\ell_i, r_i] : [i \in 1:n]\}$ gefunden werden, so dass $G \cong G(\mathcal{I})$ und $|\{\ell, ir_i : i \in [1:n]\}| = 2n$. Dazu müssen gleich Intervallgrenzen nur um ein kleines Stück verschoben werden. Ferner merken wir hier noch an, dass die Intervall-Grenzen der Intervalle einer Intervalldarstellung ohne Beschränkung der Allgemeint aus $\mathbb{N}$ gewählt werden können. Dazu müssen nur die Anfangs- und Endpunkte der Intervallgrenzen einer Intervall-Darstellung nur aufsteigend durchnummeriert werden.

Wir definieren jetzt noch zwei spezielle Klassen von Intervall-Graphen, die für die genomische Kartierung von Bedeutung sind.

Definition 6.11 *Ein Intervall-Graph G heißt* echt *(engl.* proper interval graph*), wenn er eine Intervall-Darstellung $\mathcal{I}$ besitzt (d.h. $G \cong G(\mathcal{I})$), so dass*

$$\forall I \neq I' \in \mathcal{I} : (I \not\subseteq I') \wedge (I \not\supseteq I').$$

Ein Intervall-Graph G heißt Einheits-Intervall-Graph *(engl.* unit interval graph*), wenn er eine Intervall-Darstellung $\mathcal{I}$ besitzt (d.h. $G \cong G(\mathcal{I})$), so dass $|I| = |I'|$ für alle $I, I' \in \mathcal{I}$.*

Zunächst zeigen wir, dass sich trotz unterschiedlicher Definition diese beiden Klassen gleich sind.

Lemma 6.12 *Ein Graph ist genau dann ein Einheits-Intervall-Graph, wenn er ein echter Intervall-Graph ist.*

Den Beweis dieses Lemmas überlassen wir dem Leser als Übungsaufgabe.

6.3.2 Modellierung

Warum sind Intervall-Graphen für die genomische Kartierung interessant. Schauen wir uns noch einmal unsere Aufgeb der genomischen Kartierung in Abbildung 6.22 an. Offensichtlich entsprechen die Fragmente gerade Intervallen, nämliche den Posi-

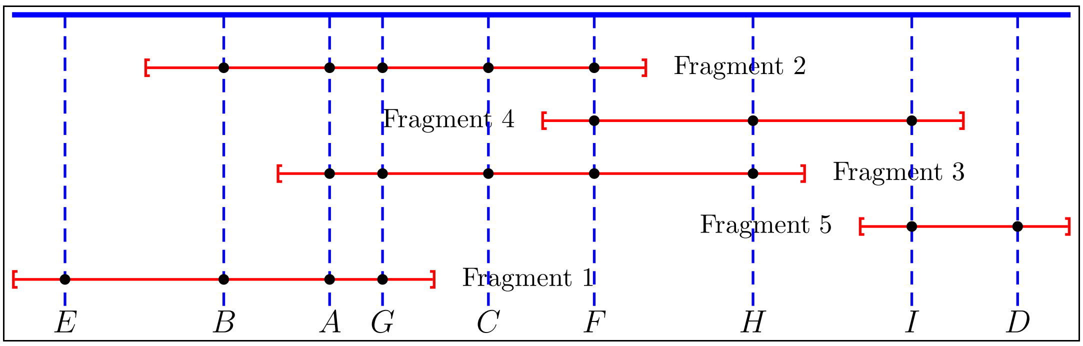


Abbildung 6.22: Skizze: Genomische Kartierung

tionen die sie überdecken. Mit Hilfe unserer Hybridisierungs-Experimente erhalten wir die Information, ob sich zwei Fragmente bzw. Intervalle überlappen, nämlich genau dann, wenn beide Fragmente dasselbe Landmark, also STS, enthalten. Somit bilden die Fragmente mit den Knoten und den Überschneidungen als Kanten einen

Intervall-Graphen. Was in der Aufgabe der genomischen Kartierung gesucht ist, ist die Anordnung der Fragmente auf dem Genom. Dies ist aber nichts anderes als eine Intervall-Darstellung des Graphen, den wir über unsere biologischen Experimente erhalten. Aus diesem Grund sind oft auch Einheits-Intervall-Graphen von Interessen, da in den biologischen Experimenten die Fragmente im Wesentlichen dieselbe Länge besitzen und somit eine Intervall-Darstellung durch gleich lange Intervalle erlauben sollte.

Wir formulieren nun einige Probleme für Intervall-Graphen, die die Problemstellung bei der genomischen Kartierung widerspiegeln soll.

PROPER INTERVALL COMPLETION (PIC)

Eingabe: Ein Graph $G = (V, E)$ und $k \in \mathbb{N}$.
Ausgabe: Ein echter Intervall-Graph $G' = (V, E \cup F)$ mit $|F| \leq k$.

Mit PIC wird versucht das Vorhandensein von False Negatives zu simulieren. Es wird angenommen, dass bei den Experimenten einige Überschneidungen von Fragmenten (maximal k) nicht erkannt wurden.

PROPER INTERVALL SELECTION

Eingabe: Ein Graph $G = (V, E)$ und $k \in \mathbb{N}$.
Ausgabe: Ein echter Intervall-Graph $G' = (V, E \setminus F)$ mit $|F| \leq k$.

Mit PIS wird versucht das Vorhandensein von False Positives zu simulieren. Es wird angenommen, dass bei den Experimenten einige Überschneidungen von Fragmenten (maximal k) zu Unrecht erkannt wurden.

INTERVALL SANDWICH (IS)

Eingabe: Ein Tripel (V, D, F) mit $D, F \subset \binom{V}{2}$.
Ausgabe: Ein Intervall-Graph $G = (V, E)$ mit $D \subset E \subset F$.

Mit IS soll in gewissen Sinne versucht werden sowohl False Positive als auch False Negatives zu simulieren. Hierbei repräsentiert die Menge D die Überschneidungen von Fragmenten, von denen man sich sicher ist, dass sich gelten. Diese werden eine Teilmenge der aus den experimentell gewonnen Überschneidungen sein. Mit der Menge F versucht ein Menge von Kanten anzugeben, die höchstens benutzt werden dürfen. Diese werden eine Obermenge der experimentellen Überschneidungen sein. Man kann das IS-Problem auch anders formulieren.

Intervall Sandwich (IS)

Eingabe: Ein Tripel (V, M, F) mit $D, F \subset \binom{V}{2}$.
Ausgabe: Ein Intervall-Graph $G = (V, E)$ mit $M \subset E$ und $E \cap F = \emptyset$.

Hierbei bezeichnet M (wie vorher D) die Menge von Kanten, die in jedem Falle im Intervall-Graphen auftreten sollen (engl. mandatory). Die Menge F bezeichnet jetzt die Menge von Kanten, die im zu konstruierenden Intervall-Graphen sicherlich nicht auftreten dürfen (engl. forbidden). Wie man sich leicht überlegt, sind die beiden Formulierungen äquivalent.

Bevor wir unser letztes Problem formalisieren, benötigen wir noch die Definition von Färbungen in Graphen.

Definition 6.13 *Sei $G = (V, E)$ ein Graph. Eine Abbildung $c : V \to [1 : k]$ heißt* k-Färbung. *Eine k-Färbung heißt* zulässig, *wenn $c(v) \neq c(w)$ für alle $\{v, w\} \in E$.*

Intervalizing Colored Graphs

Eingabe: Ein Graph $G = (V, E)$ und eine k-Färbung c.
Ausgabe: Ein Intervall-Graph $G' = (V, E')$ mit $E \subseteq E'$, so dass c eine zulässige k-Färbung für G' ist.

Die Motivation hinter dieser Formalisierung ist, dass man bei der Herstellung der Fragmente darauf achten kann, welche Fragmente aus einer Kopie des Genoms gleichzeitig generiert wurden. Damit weiß man, dass sich diese Fragmente sicherlich nicht überlappen können und gibt ihnen daher dieselbe Farbe.

Man beachte, dass ICG ist ein Spezialfall des Intervall Sandwich Problems ist. Mit $F = \{\{i, j\} : c(i) \neq c(j)\}$ können wie aus einer ICG-Instanz eine äquivalente IS-Instanz konstruieren.

6.3.3 Komplexitäten

In diesem Abschnitt wollen kurz auf die Komplexität der im letzten Abschnitt vorgestellten Probleme eingehen. Leider sind für diese fehlertolerierenden Modellierungen die Entscheidungsproblem, ob es den gesuchten Graphen gibt oder nicht, in der Regel bereits $\mathcal{NP}$-hart.

PIC: Proper Interval Completion ist $\mathcal{NP}$-hart, wenn k Teil der Eingabe ist. Für feste k ist das Problem in polynomieller Zeit lösbar, aber die Laufzeit bleibt exponentiell in k.

ICG und IS: Intervalizing Colored Graphs ist ebenfalls $\mathcal{NP}$-hart. Somit ist auch das Intervall Sandwich Problem, das ja ICG als Teilproblem enthält, ebenfalls $\mathcal{NP}$-hart Selbst für eine festes $k \geq 4$ bleibt ICG $\mathcal{NP}$-hart. Für $k \leq 3$ hingegene lassen sich jedoch polynomielle Algorithmen für ICG finden. Der Leser sei dazu eingeladen, für die Fälle $k = 2$ und $k = 3$ polynomielle Algorithmen zu finden. Leider taucht in der Praxis doch eher der Fall $k \geq 4$ auf.

6.4 Intervall Sandwich Problem

In diesem Abschnitt wollen wir das Intervall Sandwich Problem vom algorithmischen Standpunkt aus genauer unter die Lupe nehmen. Wir wollen zeigen, wie man dieses Problem prinzipiell, leider mit eine exponentiellen Laufzeit löst, und wie man daraus für einen Speziallfall einen polynomiellen Algorithmus ableiten kann.

Wir wollen an dieser Stelle noch anmerken, dass wir im Folgenden ohne Beschränkung der Allgemeinheit annehmen, dass der Eingabe-Graph (V, M) zusammenhängend ist. Andernfalls bestimmen wir eine Intervall-Darstellung für jede seine Zusammenhangskomponenten und hängen dieses willkürlich aneinander. Für praktische Eingaben in Bezug auf die genomische Kartierung können wir davon ausgehen, dass die Eingabe zusammenhängend ist, da andernfalls die Fragmente so dünn gesät waren, dass eine echte Kartierung sowieso nicht möglich ist,

6.4.1 Allgemeines Lösungsprinzip

Zunächst definieren wir einige für unsere algorithmische Idee grundlegende, dennoch sehr einfache Begriffe.

Definition 6.14 *Sei $S = (V, M, F)$ eine Eingabe für IS. Eine Teilmenge $X \subseteq V$ heißt* Kern. *Der* Rand $\beta(X) \subseteq M$ *eines Kerns X ist definiert als*

$$\beta(X) = \{e \in M \ : \ e \cap X \neq \emptyset\}.$$

Die aktive Region $\mathcal{A}(X) \subseteq V$ *eines Kerns X ist definiert als*

$$\mathcal{A}(X) = \{v \in X \ : \ \exists e \in \beta(X) : v \in e\}.$$

Der Hintergrund für diese Definition ist der folgende. Der aktuell betrachtete Kern in unserem Algorithmus wird eine Knotenteilmenge sein, für die wir eine Intervall-Darstellung bereits konstruiert haben. Die aktive Region beschreibt dann die Menge von Knoten des Kerns, für die noch benachbarte Knoten außerhalb des Kerns existieren, die dann über die Kanten aus dem Rand verbunden sind.

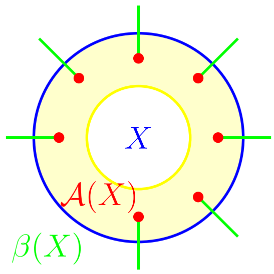


Abbildung 6.23: Skizze: Aktive Region $\mathcal{A}(X)$ und Rand $\beta(X)$ des Kerns X

Kommen wir nun dazu genauer zu formalisieren, was ein Intervall-Darstellung eines Kerns ist.

Definition 6.15 *Sei V eine Knotenmenge und $M, F \subseteq \binom{V}{2}$. Ein* Layout $L(X)$ *eines Kerns $X \subseteq V$ ist eine Funktion $I : X \to \{[a, b] \mid a < b \in \mathbb{R}\}$ mit*

1. $\forall \{v, w\} \in M \cap \binom{X}{2} : I(v) \cap I(w) \neq \emptyset$,
2. $\forall \{v, w\} \in F \cap \binom{X}{2} : I(v) \cap I(w) = \emptyset$,
3. $\forall v \in \mathcal{A}(X) : r(I(v)) = \max\{r(I(w)) \ : \ w \in X\}$, *wobei* $r([a, b]) = b$ *für alle* $a < b \in \mathbb{R}$.

Ein Kern heißt zulässig, *wenn er ein Layout besitzt.*

Damit ist ein zulässiger Kern also der Teil der Knoten, für den bereits ein Layout bzw. eine Intervall-Darstellung konstruiert wurde. Mit der nächsten Definition geben wir im Prinzip die algorithmische Idee an, wie wir zulässige Kerne erweitern wollen. Wir werden später sehen, dass diese Idee ausreichend sein wird, um für eine Eingabe des Intervall Sandwich Problems eine Intervall-Darstellung zu konstruieren.

Definition 6.16 *Ein zulässiger Kern $Y = X \cup \{v\}$* erweitert *genau dann einen zulässigen Kern X, wenn $L(Y)$ aus $L(X)$ durch Hinzufügen eines Intervalls $I(v)$ entsteht, so dass*

1. $\forall w \in X \setminus \mathcal{A}(X) : r(I(w)) < \ell(I(v))$;
2. $\forall w \in \mathcal{A}(X) : r(I(w)) = r(I(v))$.

In Abbildung 6.24 ist ein Beispiel für eine solche Erweiterung des zulässigen Kerns $\{1, 2, 3, 4\}$ zu einem zulässigen Kern $\{1, 2, 3, 4, 5\}$ dargestellt.

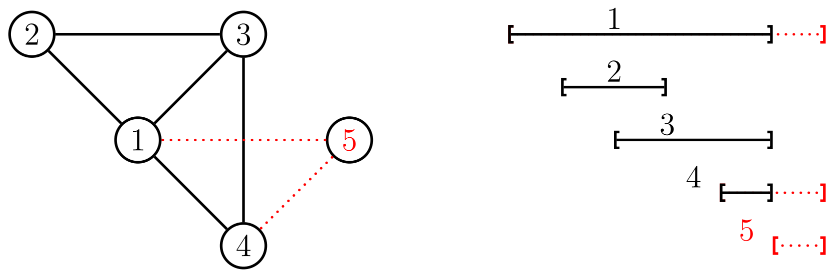


Abbildung 6.24: Skizze: Erweiterung eines Layouts

Wir kommen im folgenden Lemma zu einer einfachen Charakterisierung, wann ein zulässiger Kern eine Erweiterung eines anderen zulässigen Kerns ist. Hierbei ist insbesondere wichtig, dass diese Charakterisierung völlig unabhängig von den zugrunde liegenden Layouts ist, die den Kernen ihre Zulässigkeit bescheinigen.

Lemma 6.17 *Sei X ein zulässiger Kern. $Y = X \cup \{v\}$ ist genau dann ein zulässiger Kern und erweitert X, wenn $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$.*

Beweis: **$\Rightarrow$:** Da Y eine Erweiterung von X ist, überschneidet sich das Intervall von v mit jedem Intervall aus $\mathcal{A}(X)$. Da außerdem $Y = X \cup \{v\}$ ein zulässiger Kern ist, gilt $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$.

$\Leftarrow$: Sei also X ein zulässiger Kern. Wir betrachten das Layout von X in Abbildung 6.25 Da $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$, können wir nun alle Intervalle der

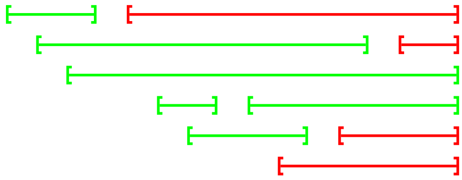


Abbildung 6.25: Skizze:

Knoten aus $\mathcal{A}(X)$ verlängern und ein neues Intervall für v einfügen, das nur mit den Intervallen aus $\mathcal{A}(X)$ überlappt. ■

Wir zeigen jetzt noch, dass es zu jedem zulässigen Kern einen kleineren zulässigen Kern gibt, der sich zu diesem erweitern lässt.

Lemma 6.18 *Jeder zulässige Kern Y erweitert mindestens einen zulässigen Kern $X \subsetneq Y$.*

Beweis: Sei $L(Y)$ mit $I : V \to \mathcal{J}(\mathbb{R})$ ein Layout für Y, wobei $\mathcal{J}(\mathbb{R})$ die Menge aller abgeschlossen reellen Intervalle bezeichnet. Wir wählen jetzt $y \in Y$, so dass $\ell(I(y))$ maximal ist. Siehe dazu auch Abbildung 6.26. Beachte, dass y nicht notwendigerweise aus $\mathcal{A}(Y)$ sein muss.

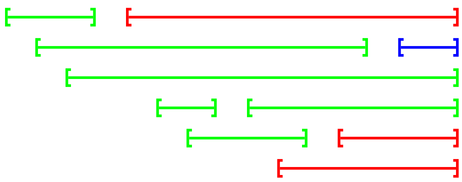


Abbildung 6.26: Skizze: Layout $L(Y)$ für Y

Wir definieren jetzt ein Layout $L(X)$ für X aus $L(Y)$ wie folgt um:

$$\forall (x, y) \in M : r(I(x)) := \max \{r(z) \ : \ z \in \mathcal{A}(Y)\}.$$

Alle anderen Werte von I auf X bleiben unverändert und y wird aus dem Definitionsbereich von I entfernt.

Wir müssen jetzt lediglich die drei Bedingungen aus der Definition eines zulässigen Layouts nachweisen. Offensichtlich gilt weiterhin $I(v) \cap I(w) \neq \emptyset$ für alle $\{v, w\} \in M \cap \binom{X}{2} \subseteq M \cap \binom{Y}{2}$. Außerdem gilt ebenfalls $I(v) \cap I(w) = \emptyset$ für alle

$\{v, w\} \in F \cap \binom{X}{2} \subseteq F \cap \binom{Y}{2}$. Letztendlich gilt nach unserer Konstruktion, dass $r(I(v)) = \max\{r(I(w)) : w \in X\}$ für alle $v \in \mathcal{A}(X)$, da man sich leicht überlegt, dass

$$\mathcal{A}(X) = \{x \in X : \{x, y\} \in M\} \cup (A(Y) \setminus \{y\}).$$

Damit ist der Beweis abgeschlossen. ■

Als unmittelbare Folgerung erhält man das folgende Korollar, dass die Basis für unseren Algorithmus sein wird.

Korollar 6.19 *Für eine Eingabe $S = (V, M, F)$ des Interval Sandwich Problems existiert genau dann eine Lösung, wenn V ein zulässiger Kern ist.*

Mit Hilfe dieses Korollars wissen wir nun, dass es eine aufsteigende Folge

$$\emptyset = X_0 \subset X_1 \subset \cdots \subset X_{n-1} \subset X_n = V$$

mit $|X_{i+1} \setminus X_i| = 1$ gibt. Das bedeutet, dass wir ein Layout iterativ für unsere Problemeingabe konstruieren können. Nach dem Lemma 6.17 wissen wir ferner, dass die Erweiterungen unabhängig vom betrachteten Layout möglich sind. Insbesondere folgt daraus, dass wenn ein Layout eines zulässigen Kerns nicht erweitert werden kann, es auch kein anderes Layout dieses Kerns geben kann, dass sich erweitern lässt.

Somit erhalten wir den in Abbildung 6.27 angegeben Algorithmus zur Konstruktion einer Intervall-Darstellung für eine gegebene Eingabe S des Intervall Sandwich Problems. Hierbei testen wir alle möglichen Erweiterungen der leeren Menge zu einem zulässigen Kern V. Da es leider exponentiell viele Erweiterungspfade gibt, nämlich genau $n!$, wenn $n = |V|$ ist, ist dieser Algorithmus sicherlich nicht praktikabel. Da der Test, ob sich ein Kern erweitern lässt nach Lemma 6.17 in Zeit $O(|V|^2)$ implementieren lässt, erhalten wir das folgende Theorem.

Theorem 6.20 *Für eine Eingabe $S = (V, M, F)$ des Interval Sandwich Problems lässt sich in Zeit $O(|V|! \cdot |V|^2)$ feststellen, ob es eine Lösung gibt, und falls ja, kann diese auch konstruiert werden.*

6.4.2 Lösungsansatz für Bounded Degree Interval Sandwich

Wir wollen nun zwei modifizierte Varianten des Intervall Sandwich Problems vorstellen, die sich in polynomieller Zeit lösen lassen. Dazu erst noch kurz die Definition eine Clique.

```
SANDWICH (S = (V, M, F))
{
    Queue Q;
    Q.enqueue(∅);
    while (not Q.is_empty())
    {
        X = Q.dequeue();
        for each v ∉ X do
            if (Y := X ∪ {v} is feasible and extends X)
                if (Y = V)
                    output Solution found;
                else
                    Q.enqueue(Y);
    }
    output No solutions found;
}
```

Abbildung 6.27: Algorithmus: Allgemeines Intervall Sandwich Problem

Definition 6.21 *Sei $G = (V, E)$ ein Graph. Eine Teilgraph $G' = (V', E')$ heißt* Clique *oder k-*Clique*, wenn folgendes gilt:*

- $|V'| = k$,
- $E' = \binom{V}{2}$ *(d.h. G' ist ein vollständiger Graph),*
- *Für jedes $v \in V \setminus V'$ ist $G'' = (V'', \binom{V''}{2})$ mit $V'' = V \cup \{v\}$ kein Teilgraph von G (d.h. G' ist ein maximaler vollständiger Teilgraph von G).*

Die Cliquenzahl $\omega(G)$ *des Graphen G ist Größe einer größten Clique von G.*

Mit Hilfe dieser Definition können wir folgende Spezialfälle des Intervall Sandwich Problems definieren.

BOUNDED DEGREE AND WIDTH INTERVAL SANDWICH

Eingabe: Ein Tripel (V, M, F) mit $D, F \subset \binom{V}{2}$ sowie zwei natürliche Zahlen $d, k \in \mathbb{N}$ mit $\Delta((V, M)) \leq d$.

Ausgabe: Ein Intervall-Graph $G = (V, E)$ mit $M \subset E$ und $E \cap F = \emptyset$ sowie $\omega(G) \leq k$.

Wir beschränken also hier die Eingabe auf Graphen mit beschränkten Grad und suchen nach Intervall-Graphen mit einer beschränkten Cliquenzahl.

Bounded Degree Interval Sandwich (BDIS)

Eingabe: Ein Tripel (V, M, F) und $d \in \mathbb{N}$ mit $D, F \subset \binom{V}{2}$.
Ausgabe: Ein Intervall-Graph $G = (V, E)$ mit $M \subset E$ und $E \cap F = \emptyset$ sowie $\delta(G) \leq d$.

Beim Bounded Degree Interval Sandwich Problem beschränken wir den Suchraum nur dadurch, dass wir für die Lösungen gradbeschränkte Intervall-Graphen zulassen. Wir werden jetzt für dieses Problem einen polynomiellen Algorithmus vorstellen. Für das erstgenannte Problem lässt sich mit ähnlichen Methoden ebenfalls ein polynomieller Algorithmus finden. Die beiden hier erwähnten Probleme sind auch für die genomische Kartierung relevant, da wir bei den biologischen Experimenten davon ausgehen, dass die Überdeckung einer Position im Genom sehr gering ist und aufgrund der kurzen, in etwa gleichlangen Länge der resultierende Intervall-Graph sowohl einen relativ kleinen Grad als auch eine relativ kleine Cliquenzahl besitzt.

Um unseren Algorithmus geeignet modifizieren zu können, müssen wir auch die grundlegende Definition anpassen.

Definition 6.22 *Sei V eine Menge und $M, F \subseteq \binom{V}{2}$. Ein d-*Layout *(oder kurz* Layout*) $L(X)$ eines Kerns $X \subseteq V$ ist eine Funktion $I : X \to \{[a, b] \mid a < b \in \mathbb{R}\}$ mit*

1. *$\forall \{v, w\} \in M \cap \binom{X}{2} : I(v) \cap I(w) \neq \emptyset$,*
2. *$\forall \{v, w\} \in F \cap \binom{X}{2} : I(v) \cap I(w) = \emptyset$,*
3. *$\forall v \in \mathcal{A}(X) : r(I(v)) = \max\{r(I(w)) : w \in X\}$, wobei $r([a, b]) = b$ für alle $a < b \in \mathbb{R}$,*
4. *Für alle $v \in X \setminus \mathcal{A}(X)$ schneidet $I(v)$ höchstens d andere Intervalle,*
5. *Für alle $v \in \mathcal{A}(X)$ schneidet $I(v)$ höchstens $d - |E(v, X)|$ andere Intervalle, wobei $E(v, X) = \{\{v, w\} \in M \mid w \notin X\}$*

*Ein Kern heißt d-*zulässig *(oder auch kurz* zulässig*), wenn er ein d-Layout besitzt und $\mathcal{A}(X) \leq d - 1$.*

Im Folgenden werden wir meist die Begriffe Layout bzw. zulässig anstatt von d-Layout bzw. d-zulässig verwenden. Aus dem Kontext sollte klar sein, welcher Begriff wirklich gemeint ist.

Im Wesentlichen sind die Bedingungen 4 und 5 in der Definition neu hinzugekommen. Die Bedingung 4 ist klar, da wir ja nur Intervall-Graphen mit maximalen Grad kleiner gleich d konstruieren wollen. Daher darf sich jeder fertig konstruierte Knoten mit maximal d anderen Intervalle schneiden. Analog ist es bei Bedingung 5. Hier gibt $|E(v, X)|$ gerade die Anzahl der Nachbarn an, die noch nicht in der aktuelle Intervall-darstellung bzw. Layout realisiert sind. Daher darf ein Intervall der aktiven Region also vorher maximal $d - |E(v, X)|$ andere Intervalle schneiden.

Es bleibt noch zu überlegen, warum man die Einschränkung gemacht hat, dass $|\mathcal{A}(X)| \leq d - 1$ ist. Wäre $|\mathcal{A}(X)| \geq d + 1$, dann würde jedes Intervall der aktiven Region bereits d andere Intervalle schneiden. Da die Knoten jedoch noch aktiv sind, gibt es noch nicht realisierte Nachbarn und der resultierenden Graph würde einen Grad von größer als d bekommen.

Warum verbieten wir auch noch $|\mathcal{A}(X)| = d$? Angenommen wir hätten einen aktive Region mit d Knoten. Dann hätte jeder Knoten der aktiven Region bereits eine Grad von $d-1$, da sich alle Intervall der aktiven Region überschneiden. Die aktive Region bildet also eine d-Clique. Wenn nun ein Knoten hinzukommt, wird er zu allen Knoten der aktiven Region benachbart. Somit konstruieren wir eine $(d+1)$-Clique, in der jeder Knoten Grad d besitzt. Würde die aktive Region also einmal aus d Knoten bestehen so müsste eine erfolgreiche Ausgabe des Algorithmus eine $(d+1)$-Clique sein. Da wir voraussetzen, dass der Eingabegraph (V, M) zusammenhängend ist und somit auch der zu konstruierende Ausgabegraph zusammenhängend sein muss, kann dies nur der Fall sein, wenn $|V| = d + 1$ ist. Andernfalls, gäbe es einen Knoten mit Grad größer als d. Wir können also vorher abprüfen, ob der vollständige Graph auf V eine zulässige Ausgabe ist und hinterher diesen Fall ausschließen.

Lemma 6.23 *Sei X ein d-zulässiger Kern. $Y = X \cup \{v\}$ ist genau dann ein d-zulässiger Kern und erweitert X, wenn $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$ und X besitzt ein d-Layout L, so dass $I(u)$ höchstens $d - |E(u, X)| - 1$ andere Intervalle schneidet, für alle $u \in \mathcal{A}(X)$ mit $\{u, v\} \notin M$, und $|\mathcal{A}(X)| \leq d - |E(v, Y)|$.*

Beweis: **$\Rightarrow$:** Nach Lemma 6.17 wissen wir, dass $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$.

Sei $Y = X \cup \{v\}$ ein zulässiger Kern, der X erweitert. Sei $L(Y)$ ein Layout von Y und $L(X)$ das Layout für X, das durch Entfernen des Intervalls für v aus dem Layout $L(Y)$ entsteht. Sei weiter $u \in \mathcal{A}(X)$ mit $\{u, v\} \notin M$. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob u auch in der aktiven Region von Y ist oder nicht.

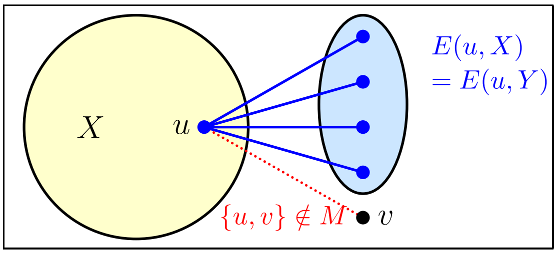


Abbildung 6.28: Skizze: Erweiterung von X um v

Fall 1 ($u \notin \mathcal{A}(Y)$): Da u sich nicht mehr in der aktiven Region von Y befindet, obwohl es in der aktiven Region von X war, muss v der letzte verbliebene Nachbar von u außerhalb von X gewesen sein. Damit ist $(u, v) \in M$ und dieser Fall kann nach der Wahl von u gar nicht auftreten.

Fall 2 ($u \in \mathcal{A}(Y)$): Da $L(Y)$ ein Layout von Y ist und sich u in der aktiven Region von Y befindet, schneidet $I(u)$ maximal $d - |E(u, Y)|$ andere Intervalle in $L(Y)$. Durch Hinzunahme von v zu X bleibt die Menge der Nachbarn von u außerhalb von X bzw. Y unverändert, d.h. $E(u, X) = E(u, Y)$ (siehe auch Abbildung 6.28).

Nach Definition der Erweiterung müssen sich jedoch die Intervalle von u und v schneiden, obwohl $\{u, v\} \notin M$. Damit schneidet das Intervall $I(u)$ im Layout von Y maximal $d - |E(u, Y)| = d - |E(u, X)|$ andere Intervalle. Da im Layout von $L(X)$ nun das Intervall von v nicht mehr enthalten ist, das sich mit dem Intervall von u schneidet, gilt im Layout von X, dass $I(u)$ maximal $d - |E(u, X)| - 1$ andere Intervalle schneidet.

Es bleibt noch zu zeigen, dass $|\mathcal{A}(X)| \leq d - |E(v, Y)|$ gilt. Da Y ein zulässiger Kern ist, schneidet das Intervall von v maximal $d - |E(v, Y)|$ andere Intervalle im Layout $L(Y)$ von Y. Die zu diesen Intervalle zugehörigen Knoten bilden gerade die aktive Region von X. Somit gilt $\mathcal{A} \leq d - |E(v, Y)|$.

$\Leftarrow$: Nach Lemma 6.17 folgt aus $(v, w) \notin F$ für alle $w \in \mathcal{A}(X)$ bereits, dass die Y eine Erweiterung von X und die Bedingungen 1 mit 3 für das Layout $L(Y)$ für Y gelten. Wir müssen also nur noch die Bedingungen 4 und 5 überprüfen.

Zum Nachweis der Bedingung 4 halten wir zunächst fest, dass Knoten aus X, die in X nicht mehr aktiv sind, sicherlich auch in Y nicht aktiv sind, d.h. es gilt $X \setminus \mathcal{A}(X) \subseteq Y \setminus \mathcal{A}(Y)$. Für alle Knoten aus $X \setminus \mathcal{A}(X)$ gilt also Bedingung 4. Sei jetzt also $y \in Y \setminus \mathcal{A}(Y) \setminus (X \setminus \mathcal{A}(X))$. Daher wird v jetzt inaktiv und es muss daher $\{v, y\} \in M$ gelten. Aufgrund der Bedingung 5 für das Layout $L(X)$ von X schneidet das Intervall von y maximal d andere Intervalle und die Bedingung 4 gilt.

Es bleibt noch der Fall, dass auch der neue Knoten v nicht in Y nicht mehr zur aktiven Region gehört. Dann schneidet v maximal $d-1$ andere Intervalle, da immer $|\mathcal{A}(X)| \leq d-1$ gilt

Zum Nachweis der Bedingung 5 für das Layout $L(Y)$ für Y machen wir wieder eine Fallunterscheidung und betrachten hierbei $y \in \mathcal{A}(Y) \subseteq (\mathcal{A}(X) \cup \{v\})$:

Fall 1 ($\boldsymbol{y \in \mathcal{A}(X)}$): Ist $\{y, v\} \in M$, dann schneidet das Intervall $I(y)$ im Layout $L(X)$ von X nach Voraussetzung maximal $d - |E(y, X)|$ andere Intervalle. Da $E(y, Y) = E(y, x) \setminus \{v\}$ und somit $|E(y, X)| = |E(y, Y)| + 1$ ist, kann $I(y)$ im erweiterten Layout $L(y)$ maximal

$$d - |E(y, X)| + 1 = d - (|E(y, Y)| + 1) + 1 = d - |E(y, Y)|$$

andere Intervalle schneiden.

Ist andererseits $\{y, v\} \notin M$, dann schneidet $I(y)$ im Layout $L(X)$ von X nach Voraussetzung maximal $d - |E(y, X)| - 1$ andere Intervalle. Somit kann $I(v)$ im erweiterten Layout $L(Y)$ maximal $d - |E(y, X)| - 1 + 1 = d - |E(y, X)|$ andere Intervalle schneiden.

Fall 2 ($\boldsymbol{y = v}$): Da $\mathcal{A}(X) \leq d - |E(v, Y)|$ ist kann $y = v$ im Layout $L(Y)$ maximal $d - |E(y, Y)|$ andere Intervalle schneiden. ■

Somit haben wir auch wieder eine Charakterisierung gefunden, die eine Erweiterung von X zu Y beschreibt, ohne auf die konkreten Layouts einzugehen. Im Gegensatz zum allgemeinen Fall müssen wir hier jedoch die Grade der Knoten in der aktiven Region bzgl. des bereits konstruierten Intervall-Graphen kennen.

Lemma 6.24 *Jeder d-zulässige Kern Y erweitert mindestens einen d-zulässigen Kern $X \subsetneq Y$.*

Beweis: Sei Y ein zulässiger Kern und sei $L(Y)$ ein zugehöriges Layout. Sei $y \in Y$ so gewählt, dass $\ell(I(y))$ maximal ist. Weiter sei $L(X)$ das Layout für $X = Y \setminus \{y\}$, das durch Entfernen von $I(y)$ aus $L(Y)$ entsteht. Aus dem Beweis von Lemma 6.18 folgt, dass die Bedingungen 1 mit 3 für das Layout $L(X)$ erfüllt sind. Wir müssen jetzt nur noch zeigen, dass auch die Bedingungen 4 und 5 gelten.

Zuerst zur Bedingung 4. Für alle $v \in X \setminus \mathcal{A}(X)$ gilt offensichtlich, dass $I(v)$ maximal d andere Intervalle schneidet. Ansonsten wäre $L(Y)$ schon kein Layout für Y, da dieses dann ebenfalls die Bedingung 4 verletzten würde.

Kommen wir jetzt zum Beweis der Gültigkeit von Bedingung 5. Zuerst stellen wir fest, dass $\mathcal{A}(X) \supseteq \mathcal{A}(Y) \setminus \{y\}$ gilt.

Fall 1 ($\boldsymbol{v \notin \mathcal{A}(Y)}$): Damit gilt, dass $(v, y) \in M$ sein muss. In $L(Y)$ schneidet $I(v)$ dann maximal d andere Intervalle. In $L(X)$ schneidet $I(v)$ dann maximal $d-1 = d-|E(v,x)|$ andere Intervalle, da sich $I(v)$ und $I(y)$ in $L(Y)$ schneiden und da $E(v,Y) = \{y\}$.

Fall 2 ($\boldsymbol{v \in \mathcal{A}(Y)}$): Nach Voraussetzung schneidet $I(v)$ maximal $d - |E(v,Y)|$ andere Intervalle im Layout $L(Y)$. Da sich die Intervalle von v und y nach Wahl von y schneiden müssen, schneidet $I(v)$ maximal $d - |E(v,Y)| - 1 = d - |E(v,X)|$ andere Intervalle in $L(X)$, da $E(v,Y) = E(v,Y) \cup \{y\}$. ■

Korollar 6.25 *Für die Eingabe $S = (V, M, F)$ des Bounded Degree Interval Sandwich Problems existiert genau dann eine Lösung, wenn V ein d-zulässiger Kern ist.*

Wir merken hier noch an, dass man X durchaus zu Y erweitern kann, obwohl ein konkretes Layout für X sich nicht zu einem Layout für Y erweitern lässt. Dennoch haben wir auch hier wieder festgestellt, dass es eine bzgl. Mengeninklusion aufsteigende Folge von zulässigen Kernen gibt, anhand derer wir von der leeren Menge als zulässigen Kern einen zulässigen Kern für V konstruieren können, sofern das Problem überhaupt eine Lösung besitzt.

Da wir für die Charakterisierung der Erweiterbarkeit nun auf die Grade der der Knoten der aktiven Region bzgl. des bereits konstruierten Intervall-Graphen angewiesen sind, ist die folgende Definition nötig.

Definition 6.26 *Für ein d-Layout $L(X)$ von X ist der* Grad *von $v \in \mathcal{A}(X)$ definiert als die Anzahl der Intervalls, die $I(v)$ schneiden.*

Ein Kern-Paar *(X, f) ist ein d-zulässiger Kern X zusammen mit einer Gradfolge $f : \mathcal{A}(X) \to \mathbb{N}$, die jedem Knoten in der aktiven Region von X ihren Grad zuordnet.*

Das vorherige Lemma impliziert, dass zwei Layouts mit demselben Grad für jeden Knoten $v \in \mathcal{A}(X)$ entweder beide erweiterbar sind oder keines von beiden. Damit können wir unseren generische Algorithmus aus dem vorigen Abschnitt wie folgt für das Bounded Degree Interval Sandwich Problem erweitern. Wenn ein Kern Paar (X, f) betrachtet wird, wird jedes mögliche Kern-Paar (Y, g) hinzugefügt, das ein

Layout für Y mit Gradfolge g besitzt und ein Layout von X mit Gradfolge f erweitert. Aus den vorherigen Lemmata folgt bereits die Korrektheit. Wir wollen uns im nächsten Abschnitt nun noch um die Laufzeit kümmern.

6.4.3 Laufzeitabschätzung

Für die Laufzeitabschätzung stellen wir zunächst einmal fest, dass wir im Algorithmus eigentlich nichts anderes tun, als einen so genannten *Berechnungsgraphen* per Tiefensuche zu durchlaufen. Die Knoten dieses Berechnungsgraphen sind die Kern-Paare und zwei Kern-Paare (X, f) und (Y, g) sind mit einer gerichteten Kante verbunden, wenn sich (X, f) zu (Y, g) erweitern lässt. Unser Startknoten ist dann die leere Menge und unser Zielknoten ist die Menge V.

Wir müssen also nur noch (per Tiefen- oder Breitensuche oder einen andere optimierten Suchstrategie) feststellen, ob sich der Zielknoten vom Startknoten aus erreichen lässt. Die Laufzeit ist dann proportional zur Anzahl der Knoten und Kanten im Berechnungsgraphen. Daher werden wir diese als erstes abschätzen.

Lemma 6.27 *Ein zulässiger Kern X ist durch das Paar $(\mathcal{A}(X), \beta(X))$ eindeutig charakterisiert.*

Beweis: Wir müssen jetzt nur feststellen, wie wir anhand des gegeben Paares feststellen können welche Knoten sich im zulässigen Kern befinden. Dazu stellen wir fest, dass genau dann $x \in X$ ist, wenn es einen Pfad von x zu einem Knoten $v \in \mathcal{A}(X)$ der aktiven Region im Graphen $(V, M \setminus \beta(X))$ gibt. ■

Somit können wir jetzt die die Anzahl der zulässigen Kerne abzählen, indem wir die Anzahl der oben beschriebenen charakterisierenden Paare abzählen.

Zuerst einmal stellen wir fest, dass es maximal

$$\sum_{i=0}^{d-1} \binom{n}{i} \leq \sum_{i=0}^{d-1} n^i \leq \frac{n^d - 1}{n - 1} = O(n^{d-1})$$

Möglichkeiten gibt, eine aktive Region aus V auszuwählen, da $|\mathcal{A}(X)| \leq d - 1$.

Für die mögliche Ränder der aktiven Region gilt, dass deren Anzahl durch $2^{d(d-1)}$ beschränkt ist. Wir müssen nämlich von jedem der $(d-1)$ Knoten jeweils festlegen welche ihrer maximal d Nachbarn bzgl. M im Rand liegen.

Jetzt müssen wir noch die Anzahl möglicher Gradfolgen abschätzen. Diese ist durch $d^{d-1} < d^d \leq 2^{\varepsilon \cdot d^2}$ für ein $\varepsilon > 0$ beschränkt, da nur für jeden Knoten aus der aktiven Region ein Wert aus d möglichen Werten in $[0 : d-1]$ zu vergeben ist. Somit ist die Anzahl der Kern-Paare ist beschränkt durch $O(2^{(1+\varepsilon)d^2} n^{d-1})$.

Lemma 6.28 *Die Anzahl der Kern-Paare, deren aktive Region maximal $d-1$ Knoten besitzt, ist beschränkt durch $O(2^{(1+\varepsilon)d^2} n^{d-1})$ für ein $\varepsilon > 0$.*

Nun müssen wir noch die Anzahl von Kanten im Berechnungsgraphen ermitteln. Statt dessen werden wir jedoch den maximalen Ausgangsgrad der Knoten ermitteln.

Wir betrachten zuerst die Kern-Paare, deren aktive Region maximale Größe, also $d-1$ Knoten, besitzen. Wie viele andere Kernpaare können ein solches Kern-Paar erweitern? Zuerst bemerken wir, dass zu einem solchen Kern nur Knoten hinzugefügt werden können, die zu Knoten der aktiven Region benachbart sind. Andernfalls würde die aktive Region auf d Knoten anwachsen, was nicht zulässig ist.

Wir müssen also nur eine Knoten aus der Nachbarschaft der aktiven Region auswählen. Da diese aus weniger als d Knoten besteht und jeder Knoten im Graphen (V, M) nur maximal d Nachbarn hat, kommen nur d^2 viele Knoten in Frage.

Nachdem wir einen dieser Knoten ausgewählt haben, können wir mit Hilfe des Graphen (V, M) und der aktuell betrachteten aktiven Region sofort die aktive Region sowie deren Rand bestimmen. Ebenfalls die Gradfolge der aktiven Region lässt sich leicht ermitteln. Bei allen Knoten, die in der aktiven Region bleiben, erhöht sich der Grad um 1. Alle, die aus der aktiven Region herausfallen, sind uninteressant, da wir uns hierfür den Grad nicht zu merken brauchen. Der Grad des neu hinzugenommen Knotens ergibt sich aus der Kardinalität der alten aktiven Region.

Somit kann der Ausgangsgrad der Kern-Paare mit eine aktiven Region von $d-1$ Knoten durch d^2 abgeschätzt werden. Insgesamt gibt es also $O(2^{(1+\varepsilon)d^2} \cdot n^{d-1})$ viele Kanten, die aus Kern-Paaren mit einer aktiven Region von $d-1$ Knoten herausgehen.

Jetzt müssen wir noch den Ausgangsgrad der Kern-Paare abschätzen, deren aktive Region weniger als $d-1$ Knoten umfasst. Hier kann jetzt jeder Knoten, der sich noch nicht im Kern befindet hinzugenommen werden. Wiederum können wir sofort die aktive Region und dessen Rand mithilfe des Graphen (V, M) berechnen. Auch die Gradfolge folgt unmittelbar.

Wie viele Kern-Paare, deren aktive Region maximal $d-2$ Knoten umfasst, gibt es denn überhaupt? Wir haben dies vorhin im Lemma 6.28 für $d-1$ angegeben. Also gibt es $O(2^{(1+\varepsilon)d^2} n^{d-2})$. Da von all diesen jeweils maximal n Kanten in unserem Berechnungsgraphen ausgehen, erhalten wir also insgesamt gibt es also

$O(2^{(1+\varepsilon)d^2} \cdot n^{d-1})$ viele Kanten, die aus Kern-Paaren mit einer aktiven Region von maximal $d-2$ Knoten herausgehen.

Damit erhalten wir zusammenfassend das folgende Theorem.

Theorem 6.29 *Das Bounded Degree Interval Sandwich Problem kann in Zeit* $O(2^{(1+\varepsilon)d^2} n^{d-1})$ *für ein* $\varepsilon > 0$ *gelöst werden.*

Da das Intervalizing Colored Graphs als Spezialfall des Interval Sandwich Problems aufgefasst werden kann, erhalten wir auch hier für eine gradbeschränkte Lösung eine polynomielle Laufzeit.

Korollar 6.30 *Das ICG Problem ist in* $\mathcal{P}$*, wenn der maximale Grad der Lösung beschränkt ist.*

Phylogenetische Bäume

7.1 Einleitung

In diesem Kapitel wollen wir uns mit *phylogenetischen Bäumen* bzw. *evolutionären Bäumen* beschäftigen. Wir wollen also die Entwicklungsgeschichte mehrerer verwandter Spezies anschaulich als Baum darstellen bzw. das Auftreten von Unterschieden in den Spezies durch Verzweigungen in einem Baum wiedergeben.

Definition 7.1 *Ein* phylogenetischer Baum *für eine Menge* $S = \{s_1, \ldots, s_n\}$ *von* n *Spezies ist ein ungeordneter gewurzelter Baum mit* n *Blättern und den folgenden Eigenschaften:*

- *Jeder innere Knoten hat mindestens zwei Kinder;*
- *Jedes Blatt ist mit genau einer Spezies* $s \in S$ *markiert;*
- *Jede Spezies taucht nur einmal als Blattmarkierung auf.*

Ungeordnet bedeutet hier, dass die Reihenfolge der Kinder eines Knotens ohne Belang ist. Die bekannten und noch lebenden (zum Teil auch bereits ausgestorbenen) Spezies werden dabei an den Blättern dargestellt. Jeder (der maximal $n-1$) inneren Knoten entspricht dann einem Ahnen der Spezies, die in seinem Teilbaum die Blätter bilden. In Abbildung 7.1 ist ein Beispiel eines phylogenetischen Baumes angegeben.

Wir wollen uns hier mit der mathematischen und algorithmischen Rekonstruktion von phylogenetischen Bäumen anhand der gegebenen biologischen Daten beschäftigen. Die daraus resultierenden Bäume müssen daher nicht immer mit der biologischen Wirklichkeit übereinstimmen. Die rekonstruierten phylogenetischen Bäume mögen Ahnen vorhersagen, die niemals existiert haben.

Dies liegt zum einen daran, dass die biologischen Daten nicht in der Vollständigkeit und Genauigkeit vorliegen, die für eine mathematische Rekonstruktion nötig sind. Zum anderen liegt dies auch an den vereinfachenden Modellen, da in der Natur nicht nur Unterscheidungen (d.h. Verzweigungen in den Bäumen) vorkommen können, sondern dass auch Vereinigungen bereits getrennter Spezies vorkommen können.

Biologisch würde man daher eher nach einem gerichteten azyklischen Graphen statt eines gewurzelten Baumes suchen. Da diese Verschmelzungen aber eher vereinzelt

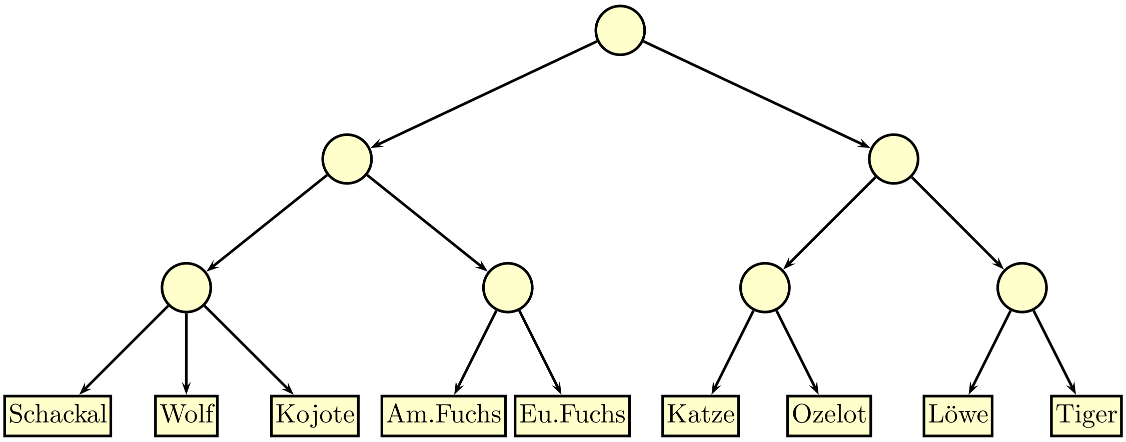


Abbildung 7.1: Beispiel: Ein phylogenetischer Baum

vorkommen, bilden phylogenetischer Bäume einen ersten Ansatzpunkt, in die dann weiteres biologisches Wissen eingearbeitet werden kann.

In der Rekonstruktion unterscheidet man zwei prinzipiell unterschiedliche Verfahren: distanzbasierte und charakterbasierte Verfahren, die wir in den beiden folgenden Unterabschnitten genauer erörtern werden.

7.1.1 Distanzbasierte Verfahren

Bei den so genannten *distanzbasierten Verfahren* wird zwischen den Spezies ein Abstand bestimmt. Man kann diesen einfach als die Zeitspanne in die Vergangenheit interpretieren, vor der sich die beiden Spezies durch Spezifizierung aus einem gemeinsamen Urahn auseinander entwickelt haben.

Für solche Distanzen, also evolutionäre Abstände, können beispielsweise die EDIT-Distanzen von speziellen DNS-Teilsträngen oder Aminosäuresequenzen verwendet werden. Hierbei wird angenommen, dass durch Mutationen die Sequenzen sich auseinander entwickeln und dass die Anzahl der so genannten akzeptierten Mutationen (also derer, die einem Weiterbestehen der Art nicht im Wege standen) zur zeitlichen Dauer korreliert ist. Hierbei muss man vorsichtig sein, da unterschiedliche Bereiche im Genom auch unterschiedliche Mutationsraten besitzen.

Eine andere Möglichkeit aus früheren Tagen sind Hybridisierungsexperimente. Dabei werden durch vorsichtiges Erhitzen die DNS-Doppelstränge zweier Spezies voneinander getrennt. Bei der anschließenden Abkühlung hybridisieren die DNS-Stränge wieder miteinander. Da jetzt jedoch DNS-Einzelstränge von zwei Spezies vorliegen, können auch zwei Einzelstränge von zwei verschiedenen Spezies miteinander hybridisieren, vorausgesetzt, die Stränge waren nicht zu verschieden.

Beim anschließenden erneuten Erhitzen trennen sich diese gemischten Doppelstränge umso schneller, je verschiedener die DNS-Sequenzen sind, da dann entsprechend weniger Wasserstoffbrücken aufzubrechen sind. Aus den Temperaturen, bei denen sich dann diese gemischten DNS-Doppelstränge wieder trennen, kann man dann ein evolutionäres Abstandsmaß gewinnen.

Ziel der evolutionären Verfahren ist es nun, einen Baum mit Kantengewichten zu konstruieren, so dass das Gewicht der Pfade von den zwei Spezies zu ihrem niedrigsten gemeinsamen Vorfahren dem Abstand entspricht. Ein solcher phylogenetischer Baum, der aufgrund von künstlichen evolutionären Distanzen konstruiert wurde, ist in der Abbildung 7.2 illustriert.

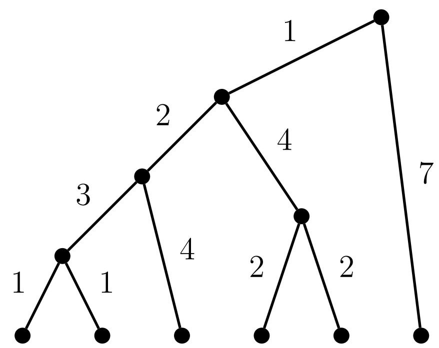


Abbildung 7.2: Beispiel: Ein distanzbasierter phylogenetischer Baum

7.1.2 Charakterbasierte Methoden

Bei den so genannten *charakterbasierten Verfahren* verwendet man gewisse Eigenschaften, so genannte *Charaktere*, der Spezies. Hierbei unterscheidet man *binäre Charaktere*, wie beispielsweise „ist ein Säugetier“, „ist ein Wirbeltier“, „ist ein Fisch“, „ist ein Vogel“, „ist ein Lungenatmer“, etc., *numerische Charaktere*, wie beispielsweise Anzahl der Extremitäten, Anzahl der Wirbel, etc, und *zeichenreihige Charaktere*, wie beispielsweise bestimmte Teilsequenzen in der DNS. Bei letzterem betrachtet man oft Teilsequenzen aus nicht-codierenden und nicht-regulatorischen Bereichen der DNS, da diese bei Mutationen in der Regel unverändert weitergegeben werden und nicht durch Veränderung einer lebenswichtigen Funktion sofort aussterben.

Das Ziel ist auch hier wieder die Konstruktion eines phylogenetischen Baumes, wobei die Kanten mit Charakteren und ihren Änderungen markiert werden. Eine Markierung einer Kante mit einem Charakter bedeutet hierbei, dass alle Spezies in dem

Teilbaum nun eine Änderung dieses Charakters erfahren. Die genaue Änderung dieses Charakters ist auch an der Kante erfasst.

Bei charakterbasierten Verfahren verfolgt man das Prinzip der minimalen Mutationshäufigkeit bzw. der maximalen Parsimonie (engl. parsimony, Geiz). Das bedeutet, dass man einen Baum sucht, der so wenig Kantenmarkierungen wie möglich besitzt. Man geht hierbei davon aus, dass die Natur keine unnötigen Mutationen verwendet.

In der Abbildung 7.3 ist ein Beispiel für einen solchen charakterbasierten phylogenetischen Baum angegeben, wobei hier nur binäre Charaktere verwendet wurden. Außerdem werden die binären Charaktere hier so verwendet, dass sie nach einer Kantenmarkierung in den Teilbaum eingeführt und nicht gelöscht werden. Bei binären Charakteren kann man dies immer annehmen, da man ansonsten den binären Charakter nur negieren muss.

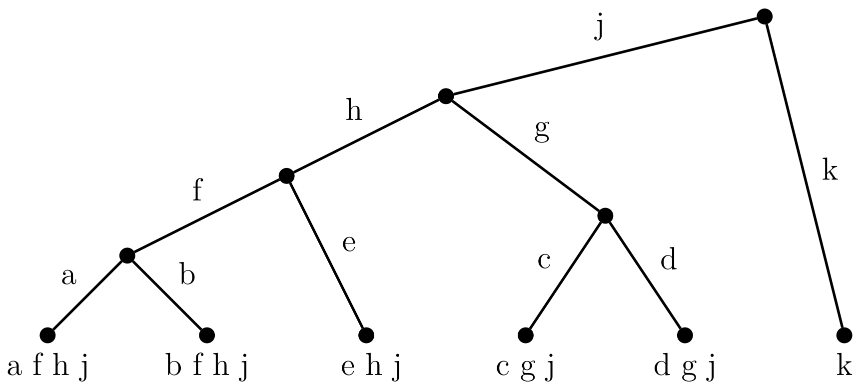


Abbildung 7.3: Beispiel: Ein charakterbasierter phylogenetischer Baum

7.2 Ultrametriken und ultrametrische Bäume

Wir wollen uns zuerst mit distanzbasierten Methoden beschäftigen. Dazu stellen wir zuerst einige schöne und einfache Charakterisierungen vor, ob eine gegebene Distanzmatrix einen phylogenetischen Baum besitzt oder nicht.

7.2.1 Metriken und Ultrametriken

Zuerst müssen wir noch ein paar Eigenschaften von Distanzen wiederholen und einige hier nützliche zusätzliche Definitionen angeben. Zuerst wiederholen wir die Definition einer Metrik (siehe auch Definition 3.4).

Definition 7.2 *Eine Funktion* $d : M^2 \to \mathbb{R}_+$ *heißt* Metrik*, wenn*

(M1) $\forall x, y \in M : d(x, y) = 0 \Leftrightarrow x = y$ *(Definitheit),*

(M2) $\forall x, y \in M : d(x, y) = d(y, x)$ *(Symmetrie),*

(M3) $\forall x, y, z \in M : d(x, z) \leq d(x, y) + d(y, z)$ *(Dreiecksungleichung).*

Im Folgenden werden wir auch die folgende verschärfte Variante der Dreiecksungleichung benötigen.

Definition 7.3 *Eine Metrik heißt* Ultrametrik*, wenn zusätzlich die so genannte* ultrametrische Dreiecksungleichung *gilt:*

$$\forall x, y, z \in M : d(x, z) \leq \max\{d(x, y), d(y, z)\}.$$

Eine andere Charakterisierung der ultrametrischen Ungleichung wird uns im Folgenden aus beweistechnischen Gründen nützlich sein.

Lemma 7.4 *Sei d eine Ultrametrik auf M. Dann sind für alle $x, y, z \in M$ die beiden größten Zahlen aus $d(x, y)$, $d(y, z)$ und $d(x, z)$ gleich.*

Beweis: Zuerst gelte die ultrametrische Dreiecksungleichung für alle $x, y, z \in M$:

$$d(x, z) \leq \max\{d(x, y), d(y, z)\}.$$

Ist $d(x, y) = d(y, z)$, dann ist nichts zu zeigen. Sei also ohne Beschränkung der Allgemeinheit $d(x, y) < d(y, z)$. Dann ist auch $d(x, z) \leq d(y, z)$.

Aufgrund der ultrametrischen Ungleichung gilt ebenfalls:

$$d(y, z) \leq \max\{d(y, x), d(x, z)\} = d(x, z).$$

Die letzte Ungleichung folgt aus der obigen Tatsache, dass $d(y, z) > d(y, x)$.

Zusammen gilt also $d(x, z) \leq d(y, z) \leq d(x, z)$. Also gilt $d(x, z) = d(y, z) > d(x, y)$ und das Lemma ist bewiesen. ∎

Nun zeigen wir auch noch die umgekehrte Richtung.

Lemma 7.5 *Sei $d : M^2 \to \mathbb{R}_+$, wobei für alle $x, y \in M$ genau dann $d(x, y) = 0$ gilt, wenn $x = y$. Weiter gelte, dass für alle $x, y, z \in M$ die beiden größten Zahlen aus $d(x, y)$, $d(y, z)$ und $d(x, z)$ gleich sind. Dann ist d eine Ultrametrik.*

Beweis: Die Definitheit (M1) gilt nach Voraussetzung.

Für die Symmetrie (M2) betrachten wir beliebige $x, y \in M$. Aus der Voraussetzung folgt mit $z = x$, dass von $d(x, y)$, $d(y, x)$ und $d(x, x)$ die beiden größten Werte gleich sind. Da nach Voraussetzung $d(x, x) = 0$ sowie $d(x, y) \geq 0$ und $d(y, x) \geq 0$ gilt, folgt, dass $d(x, y) = d(y, x)$ die beiden größten Werte sind und somit nach Voraussetzung gleich sein müssen.

Für die ultrametrische Dreiecksungleichung ist Folgendes zu zeigen:

$$\forall x, y, z \in M : d(x, z) \leq \max\{d(x, y), d(y, z)\}.$$

Wir unterscheiden drei Fälle, je nachdem, welche beiden Werte der drei Distanzen die größten sind und somit nach Voraussetzung gleich sind.

Fall 1 ($\boldsymbol{d(x, z) \leq d(x, y) = d(y, z)}$): Die Behauptung lässt sich sofort verifizieren.

Fall 2 ($\boldsymbol{d(y, z) \leq d(x, y) = d(x, z)}$): Die Behauptung lässt sich sofort verifizieren.

Fall 3 ($\boldsymbol{d(x, y) \leq d(x, z) = d(y, z)}$): Die Behauptung lässt sich sofort verifizieren. ■

Da die beiden Lemmata bewiesen haben, dass von drei Abständen die beiden größten gleich sind, nennt man diese Eigenschaft auch *3-Punkte-Bedingung*.

Zum Schluss dieses Abschnittes definieren wir noch so genannte Distanzmatrizen, aus denen wir im Folgenden die evolutionären Bäumen konstruieren wollen.

Definition 7.6 *Sei $D = (d_{i,j})$ eine symmetrische $n \times n$-Matrix mit $d_{i,i} = 0$ und $d_{i,j} > 0$ für alle $i, j \in [1 : n]$. Dann heißt D eine* Distanzmatrix.

7.2.2 Ultrametrische Bäume

Zunächst einmal definieren wir spezielle evolutionäre Bäume, für die sich, wie wir sehen werden, sehr effizient die gewünschten Bäume konstruieren lassen. Bevor wir diese definieren können, benötigen wir noch den Begriff des niedrigsten gemeinsamen Vorfahren von zwei Knoten in einem Baum.

Definition 7.7 *Sei $T = (V, E)$ ein gewurzelter Baum. Seien $v, w \in V$ zwei Knoten von T. Der* niedrigste gemeinsame Vorfahr *von v und w, bezeichnet als* $\mathrm{lca}(v, w)$ *(engl.* least common ancestor*), ist der Knoten $u \in V$, so dass u sowohl ein Vorfahr von v als auch von w ist und es keinen echten Nachfahren von u gibt, der ebenfalls ein Vorfahr von v und w ist.*

Mit Hilfe des Begriffs des niedrigsten gemeinsamen Vorfahren können wir jetzt ultrametrische Bäume definieren.

Definition 7.8 *Sei D eine $n \times n$-Distanzmatrix. Ein* (strenger) ultrametrischer Baum *T für D ist ein Baum $T = T(D)$ mit*

1. *T besitzt n Blätter, die bijektiv mit $[1 : n]$ markiert sind;*
2. *Jeder innere Knoten von T besitzt mindestens 2 Kinder, die mit Werten aus D markiert sind;*
3. *Entlang eines jeden Pfades von der Wurzel von T zu einem Blatt ist die Folge der Markierungen an den inneren Blättern (streng) monoton fallend;*
4. *Für je zwei Blätter i und j von T ist die Markierung des niedrigsten gemeinsamen Vorfahren gleich d_{ij}.*

In Folgenden werden wir hauptsächlich strenge ultrametrische Bäume betrachten. Wir werden jedoch der Einfachheit wegen immer von ultrametrischen Bäume sprechen. In Abbildung 7.4 ist ein Beispiel für eine 6×6-Matrix angegeben, die einen ultrametrischen Baum besitzt.

Wir wollen an dieser Stelle noch einige Bemerkungen zu ultrametrischen Bäumen festhalten.

- Nicht jede Matrix D besitzt einen ultrametrischen Baum. Dies folgt aus der Tatsache, dass jeder Baum, in dem jeder innere Knoten mindestens zwei Kinder besitzt, maximal $n - 1$ innere Knoten besitzen kann. Dies gilt daher insbesondere für ultrametrische Bäume Also können in Matrizen, die einen ultrametrischen Baum besitzen, nur $n - 1$ von Null verschiedene Werte auftreten.

D	1	2	3	4	5	6
1	0	7	6	7	7	3
2		0	7	4	4	7
3			0	7	7	6
4				0	4	7
5					0	7
6						0

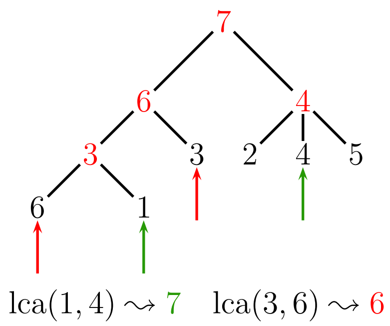


Abbildung 7.4: Beispiel: ultrametrischer Baum

- Der Baum $T(D)$ für D heißt auch *kompakte Darstellung* von D, da sich eine Matrix mit n^2 Einträgen durch einen Baum der Größe $O(n)$ darstellen lässt.

- Die Markierung an den inneren Knoten können als Zeitspanne in die Vergangenheit interpretiert werden. Vor diesem Zeitraum haben sich die Spezies, die in verschiedenen Teilbäumen auftreten, auseinander entwickelt.

Unser Ziel wird es jetzt sein, festzustellen, ob eine gegebene Distanzmatrix einen ultrametrischen Baum besitzt der nicht. Zuerst einmal überlegen wir uns, dass es es sehr viele gewurzelte Bäume mit n Blättern gibt. Somit scheidet ein einfaches Ausprobieren aller möglichen Bäume aus.

Lemma 7.9 *Die Anzahl der ungeordneten binären gewurzelten Bäume mit n Blättern beträgt*

$$\prod_{i=2}^{n}(2i-3) = \frac{(2n-3)!}{2^{n-2}\cdot(n-2)!}.$$

Um ein besseres Gefühl für dieses Anzahl zu bekommen, rechnet man leicht nach, dass

$$\prod_{i=2}^{n}(2i-3) \geq (n-1)! \geq 2^{n-2}$$

gilt. Eine bessere Abschätzung lässt sich natürlich mit Hilfe der Stirlingschen Formel bekommen.

Beweis: Wir führen den Beweis durch vollständige Induktion über n.

Induktionsanfang ($n = 2$): Hierfür gilt die Formel offensichtlich, das es genau einem Baum mit zwei markierten Blättern gibt.

Induktionsanfang ($n - 1 \to n$): Sei T ein ungeordneter binärer gewurzelter Baum mit n Blättern. Der Einfachheit nehmen wir im Folgenden an, dass unser Baum noch eine Superwurzel besitzt, deren einziges Kind die ursprüngliche Wurzel von T ist.

Wir entfernen jetzt das Blatt v mit der Markierung n. Der Elter w davon hat jetzt nur noch ein Kind und wir entfernen es ebenfalls. Dazu wird das andere Kind von w jetzt ein Kind des Elters von w (anstatt von w). Den so konstruierten Baum nennen wir T'. Wir merken noch an, dass genau eine Kante eine „Erinnerung“ an das Entfernen von v und w hat. Falls w die Wurzel war, so bleibt die Superwurzel im Baum und die Kante von der Superwurzel hat sich das Entfernen „gemerkt“.

Wir stellen fest, dass T' ein ungeordneter binärer gewurzelter Baum ist. Davon gibt es nach Induktionsvoraussetzung $\prod_{i=2}^{n-1}(2i - 3)$ viele. Darin kann an jeder Kante v mit seinem Elter w entfernt worden sein.

Wie viele Kanten besitzt ein binärer gewurzelter Baum mit $n - 1$ Blättern? Ein binärer gewurzelter Baum mit $n-1$ Blättern besitzt genau $n-2$ innere Knoten plus die von uns hinzugedachte Superwurzel. Somit besitzt der Baum

$$(n - 1) + (n - 2) + 1 = 2n - 2$$

Knoten. Da in einem Baum die Anzahl der Kanten um eines niedriger ist als die Anzahl der Knoten, besitzt unser Baum $2n-3$ Kanten, die sich an einen Verlust eines Blattes „erinnern“ können. Somit ist die Gesamtanzahl der ungeordneten binären gewurzelten Bäume mit n Blättern genau

$$(2n - 3) \cdot \prod_{i=2}^{n-1}(2i - 3) = \prod_{i=2}^{n}(2i - 3)$$

und der Induktionschluss ist vollzogen. ■

Wir fügen noch eine ähnliche Behauptung für die Anzahl ungewurzelter Bäume an. Der Beweis ist im Wesentlichen ähnlich zu dem vorherigen.

Lemma 7.10 *Die Anzahl der ungewurzelten (freien) Bäume mit n Blättern, deren innere Knoten jeweils den Grad 3 besitzen, beträgt*

$$\prod_{i=3}^{n}(2i - 5) = \frac{(2n - 5)!}{2^{n-3} \cdot (n - 3)!}.$$

Wir benötigen jetzt noch eine kurze Definition, die Distanzmatrizen und Metriken in Beziehung setzen.

Definition 7.11 *Eine $n \times n$-Distanzmatrix M* induziert *eine Metrik bzw. Ultrametrik auf $[1:n]$, wenn die Funktion $d : [1:n]^2 \to \mathbb{R}_+$ mit $d(x,y) = M_{x,y}$ eine Metrik bzw. Ultrametrik ist.*

Mit Hilfe oben erwähnten Charakterisierung einer Ultrametrik, dass von den drei Abständen zwischen drei Punkten, die beiden größten gleich sind, können wir sofort einen einfachen Algorithmus zur Erkennung ultrametrischer Matrizen angeben. Wir müssen dazu nur alle dreielementigen Teilmengen aus $[1:n]$ untersuchen, ob von den drei verschiedenen Abständen, die beiden größten gleich sind. Falls ja, ist die Matrix ultrametrisch, ansonsten nicht.

Dieser Algorithmus hat jedoch eine Laufzeit $O(n^3)$. Wir wollen im Folgenden einen effizienteren Algorithmus zur Konstruktion ultrametrischer Matrizen angeben, der auch gleichzeitig noch die zugehörigen ultrametrischen Bäume mitberechnet.

7.2.3 Charakterisierung ultrametrischer Bäume

Wir geben jetzt eine weitere Charakterisierung ultrametrischer Matrizen an, deren Beweis sogar einen effizienten Algorithmus zur Konstruktion ultrametrischer Bäume erlaubt.

Theorem 7.12 *Eine symmetrische $n \times n$-Distanzmatrix D besitzt genau dann einen ultrametrischen Baum, wenn D eine Ultrametrik induziert.*

Beweis: **$\Rightarrow$:** Die Definitheit und Symmetrie folgt unmittelbar aus der Definition einer Distanzmatrix. Wir müssen also nur noch die ultrametrische Dreiecksungleichung zeigen:

$$\forall i,j,k \in [1:n] : \; d_{ij} \leq \max\{d_{ik}, d_{jk}\}.$$

Wir betrachten dazu den ultrametrischen Baum $T = T(D)$. Wir unterscheiden dazu drei Fälle in Abhängigkeit, wie sich die niedrigsten gemeinsamen Vorfahren von i, j und k zueinander verhalten. Sei dazu $x = \mathrm{lca}(i,j)$, $y = \mathrm{lca}(i,k)$ und $z = \mathrm{lca}(j,k)$. In einem Baum muss dabei gelten, dass mindestens zwei der drei Knoten identisch sind. Die ersten beiden Fälle sind in Abbildung 7.5 dargestellt.

Fall 1 ($\boldsymbol{y = z \neq x}$): Damit folgt aus dem rechten Teil der Abbildung 7.5 sofort, dass $d(i,j) \leq d(i,k) = d(j,k)$.

Fall 2 ($\boldsymbol{x = y \neq z}$): Damit folgt aus dem linken Teil der Abbildung 7.5 sofort, dass $d(j,k) \leq d(i,k) = d(i,j)$.

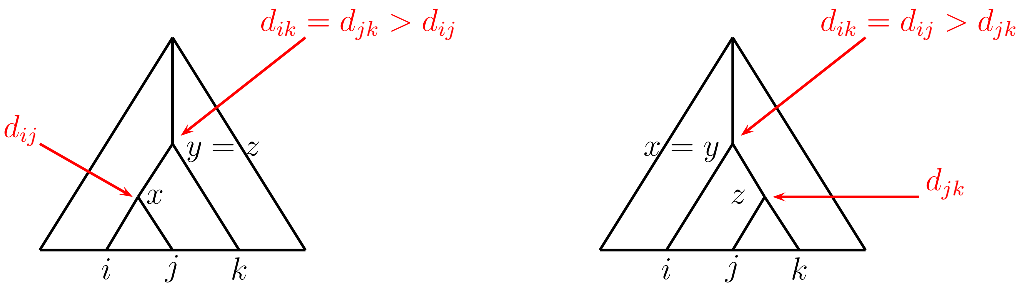


Abbildung 7.5: Skizze: Fall 1 und Fall 2

Fall 3 ($\boldsymbol{x = z \neq y}$): Dieser Fall ist symmetrisch zu Fall 2 (einfaches Vertauschen von i und j).

Fall 4 ($\boldsymbol{x = y = z}$): Aus der Abbildung 7.6 folgt auch hier, dass die ultrametrische Dreiecksungleichung gilt, da alle drei Abstände gleich sind. Wenn man statt

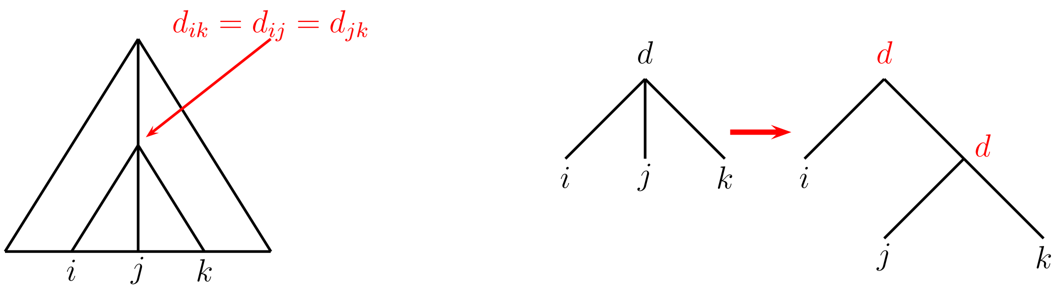


Abbildung 7.6: Skizze: Fall 4 und binäre Neukonstruktion

eines strengen ultrametrischen Baumes lieber einen binären ultrametrischen Baum haben möchte, so kann man ihn beispielsweise wie im rechten Teil der Abbildung 7.6 umbauen.

$\Leftarrow$: Wir betrachten zuerst die Abstände von Blatt 1 zu allen anderen Blättern. Sei also $\{d_{11}, \ldots, d_{1n}\} = \{\delta_1, \ldots, \delta_k\}$, d.h. $\delta_1, \ldots, \delta_k$ sind die paarweise verschiedenen Abstände, die vom Blatt 1 aus auftreten. Ohne Beschränkung der Allgemeinheit nehmen wir dabei an, dass $\delta_1 < \cdots < \delta_k$. Wir partitionieren dann $[2 : n]$ wie folgt:

$$D_i = \{\ell \in [2 : n] \ : \ d_{1\ell} = \delta_i\}.$$

Es gilt dann offensichtlich $[2 : n] = \biguplus_{i=1}^{k} D_i$. Wir bestimmen jetzt für die Mengen D_i rekursiv die entsprechenden ultrametrischen Bäume. Anschließend konstruieren wir einen Pfad von der Wurzel zum Blatt 1 mit k inneren Knoten, an die die rekursiv konstruierten Teilbäume $T(D_i)$ angehängt werden. Dies ist in Abbildung 7.7 schematisch dargestellt.

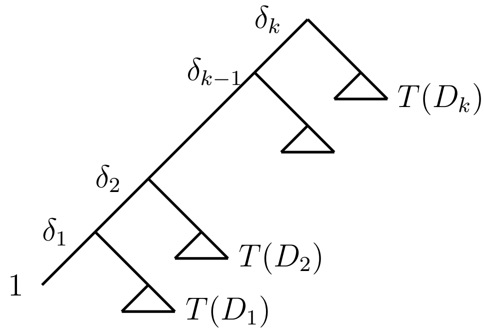


Abbildung 7.7: Skizze: Rekursive Konstruktion ultrametrischer Bäume

Wir müssen jetzt nachprüfen, ob der konstruierte Baum ultrametrisch ist. Dazu müssen wir zeigen, dass die Knotenmarkierungen auf einem Pfad von der Wurzel zu einem Blatt streng monoton fallend sind und dass der Abstand von den Blättern i und j gerade die Markierung von $\text{lca}(i, j)$ ist.

Für den ersten Teil überlegen wir uns, dass diese sowohl auf dem Pfad von der Wurzel zu Blatt 1 gilt als auch in den Teilbäumen $T(D_i)$. Wir müssen nur noch die Verbindungspunkte überprüfen. Dies ist in Abbildung 7.8 illustriert. Hier sind x und

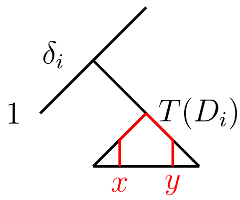


Abbildung 7.8: Skizze: Abstände von x und y und geforderte Monotonie

y zwei Blättern in $T(D_i)$, deren niedrigster gemeinsamer Vorfahre die Wurzel von $T(D_i)$ ist. Wir müssen als zeigen, dass $d_{x,y} < \delta_i$ gilt. Es gilt zunächst aufgrund der ultrametrischen Dreiecksungleichung:

$$d_{xy} \leq \max\{d_{1x}, d_{1y}\} = d_{1x} = d_{1y} = \delta_i.$$

Gilt jetzt $d_{xy} < \delta_i$ dann ist alles gezeigt. Andernfalls gilt $\delta_{xy} = \delta_i$ und wir werden den Baum noch ein wenig umbauen, wie in der folgenden Abbildung 7.9 illustriert. Dabei wird die Wurzel des Teilbaums $T(D_i)$ mit dem korrespondierenden Knoten des Pfades von der Wurzel des Gesamtbaumes zum Blatt 1 miteinander identifiziert und die Kante dazwischen gelöscht. Damit haben wir die strenge Monotonie der Knotenmarkierungen auf den Pfaden von der Wurzel zu den Blättern nachgewiesen.

Es ist jetzt noch zu zeigen, dass die Abstände von zwei Blättern x und y den Knotenmarkierungen entsprechen. Innerhalb der Teilbäume $T(D_i)$ gilt dies nach Kon-

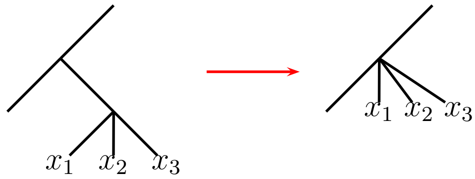


Abbildung 7.9: Skizze: Umbau im Falle nicht-strenger Monotonie

struktion. Ebenfalls gilt dies nach Konstruktion für das Blatt 1 mit allen anderen Blättern.

Wir müssen diese Eigenschaft nur noch nachweisen, wenn sich zwei Blätter in unterschiedlichen Teilbäumen befinden. Sei dazu $x \in V(T(D_i))$ und $y \in V(T(D_j))$, wobei wir ohne Beschränkung der Allgemeinheit annehmen, dass $\delta_i > \delta_j$ gilt. Dies ist in der Abbildung 7.10 illustriert.

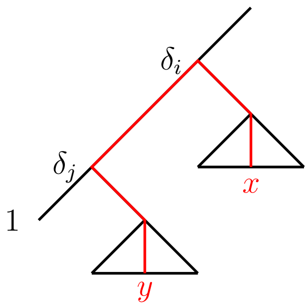


Abbildung 7.10: Skizze: Korrektheit der Abstände zweier Blätter in unterschiedlichen rekursiv konstruierten Teilbäumen

Nach Konstruktion gilt: $\delta_i = d_{1x}$ und $\delta_j = d_{1y}$. Mit Hilfe der ultrametrischen Dreiecksungleichung folgt:

$$\begin{aligned}
d_{xy} &\leq \max\{d_{1x}, d_{1y}\} \\
&= \max\{\delta_i, \delta_j\} \\
&= \delta_i \\
&= d_{1x} \\
&\leq \max\{d_{1y}, d_{yx}\} \\
&\quad \text{da } d_{1y} = \delta_j < \delta_i = d_{1x} \\
&= d_{xy}
\end{aligned}$$

Daraus folgt also $d_{xy} \leq \delta_i \leq d_{xy}$ und somit $\delta_i = d_{xy}$. Damit ist der Beweis abgeschlossen. ■

Aus dem Beweis folgt unter Annahme der strengen Monotonie (d.h. für strenge ultrametrische Bäume), dass der konstruierte ultrametrische Baum eindeutig ist. Die Konstruktion des ultrametrischen Baumes ist ja bis auf Umordnung der Kinder eines Knoten eindeutig festgelegt.

Korollar 7.13 *Sei D eine streng ultrametrische Matrix, dann ist der zugehörige strenge ultrametrische Baum eindeutig.*

Für nicht-strenge ultrametrische Bäume kann man sich überlegen, wie die Knoten mit gleicher Markierung umgeordnet werden können, so dass der Baum ein ultrametrischer bleibt. Bis auf diese kleinen Umordnungen sind auch nicht-streng ultrametrische Bäume im Wesentlichen eindeutig.

7.2.4 Konstruktion ultrametrischer Bäume

Wir versuchen jetzt aus dem Beweis der Existenz eines ultrametrischen Baumes einen effizienten Algorithmus zur Konstruktion eines ultrametrischen Baumes zu entwerfen und dessen Laufzeit zu analysieren.

Wir erinnern noch einmal an die Partition von $[2 : n]$ durch $D_i = \{\ell : d_{1\ell} = \delta_i\}$ mit $n_i := |D_i|$. Dabei war $\{d_{11}, \ldots, d_{1n}\} = \{\delta_1, \ldots, \delta_k\}$, wobei $\delta_1 < \cdots < \delta_k$. Wir erinnern hier auch noch einmal an Skizze der Konstruktion des ultrametrischen Baumes, wie in Abbildung 7.11.

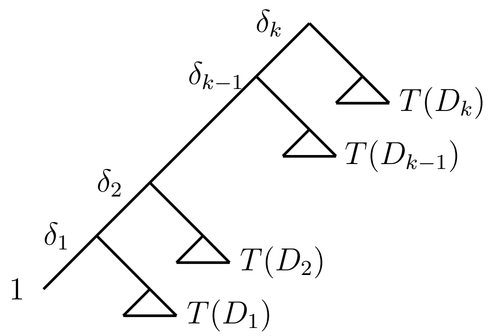


Abbildung 7.11: Skizze: Konstruktion eines ultrametrischen Baumes

Daraus ergibt sich der erste naive Algorithmus, der in Abbildung 7.12 aufgelistet ist. Da jeder Schritt Zeitbedarf $O(n \log(n))$ und es maximal n rekursive Aufrufe geben kann (mit jedem rekursiven Aufruf wird ein Knoten des ultrametrischen Baumes explizit konstruiert), ist die Laufzeit insgesamt $O(n^2 \log(n))$.

1. Sortiere die Menge $\{d_{11}, \ldots, d_{1n}\}$ und bestimme anschließend $\delta_1, \ldots, \delta_k$ mit $\{\delta_1, \ldots, \delta_k\} = \{d_{11}, \ldots, d_{1n}\}$ und partitioniere $[2 : n] = \biguplus_{i=1}^{k} D_i$. $O(n \log(n))$
2. Bestimme für $D_1, \ldots D_k$ ultrametrische Bäume $T(D_1), \ldots, T(D_k)$ mittels Rekursion. $\sum_{i=1}^{k} T(n_j)$
3. Setze die Teillösungen und den Pfad von der Wurzel zu Blatt 1 zur Gesamtlösung zusammen. $O(k) = O(n)$

Abbildung 7.12: Algorithmus: Naive Konstruktion eines ultrametrischen Baumes

Wir werden jetzt noch einen leicht modifizierten Algorithmus vorstellen, der eine Laufzeit von nur $O(n^2)$ besitzt. Dies ist optimal, da die Eingabe, die gegebene Distanzmatrix, bereits eine Größe von $\Theta(n^2)$ besitzt.

Dazu beachten wir, dass der aufwendige Teil des Algorithmus das Sortieren der Elemente in $\{d_{11}, \ldots, d_{1n}\}$ ist. Insbesondere ist dies sehr teuer, wenn es nur wenige verschieden Elemente in dieser Menge gibt, da dann auch nur entsprechend wenig rekursive Aufrufe folgen.

Daher werden wir zuerst feststellen, wie viele verschiedene Elemente es in der Menge $\{d_{11}, \ldots, d_{1n}\}$ gibt und bestimmen diese. Die paarweise verschiedenen Elemente dieser Menge können wir in einer linearen Liste (oder auch in einem balancierten Suchbaum) aufsammeln. Dies lässt sich in Zeit $O(k \cdot n)$ (bzw. in Zeit $O(n \log(k))$ bei Verwendung balancierter Bäume) implementieren. Anschließend müssen wir nur noch k Elemente sortieren. Der Algorithmus selbst ist in Abbildung 7.13 aufgelistet.

1. Bestimme zuerst $k = |\{d_{11}, \ldots, d_{1n}\}|$ und $\{\delta_1, \ldots, \delta_k\} = \{d_{11}, \ldots, d_{1n}\}$.

   Dies kann mit Hilfe linearer Listen in Zeit $O(k \cdot n)$ erledigt werden. Mit Hilfe balancierter Bäume können wir dies sogar in Zeit $O(n \log(k))$ realisieren.
2. Sortiere die k paarweise verschiedenen Werte $\{\delta_1, \ldots, \delta_k\}$.

   Dies kann in Zeit $O(k \log(k)$ erledigt werden.
3. Bestimme die einzelnen Teilbäume $T(D_i)$ rekursiv.
4. Setze die Teillösungen und den Pfad von der Wurzel zu Blatt 1 zur Gesamtlösung zusammen.

   Dies lässt sich wiederum in Zeit $O(k)$ realisieren

Abbildung 7.13: Algorithmus: Konstruktion eines ultrametrischen Baumes

Damit erhalten wir als Rekursionsgleichung für diesen modifizierten Algorithmus:

$$T(n) = d \cdot k \cdot n + \sum_{i=1}^{k} T(n_i)$$

mit $\sum_{i=1}^{k} n_i = n - 1$, wobei $n_i \geq 1$ für $i \in [1 : n]$, und einer geeignet gewählten Konstanten d. Hierbei ist zu beachten, dass $O(k \log(k)) = O(k \cdot n)$ ist, da ja $k < n$ ist.

Lemma 7.14 *Ist D eine ultrametrische $n \times n$-Matrix, dann kann der zugehörige ultrametrische Baum in Zeit $O(n^2)$ konstruiert werden.*

Beweis: Es ist nur noch zu zeigen, dass $T(n) \leq c \cdot n^2$ für eine geeignet gewählte Konstante c gilt. Wir wählen c jetzt so, dass zum einen $c \geq 2d$ und zum anderen $T(n) \leq c \cdot n^2$ für alle $n \leq 3$ gilt. Den Beweis selbst führen wir mit vollständiger Induktion über n.

Induktionsanfang ($n \leq 3$): Nach Wahl von c gilt dies offensichtlich.

Induktionsschritt ($\to n$): Es gilt dann nach Konstruktion des Algorithmus mit $n = \sum_{i=1}^{k} n_i$:

$$\begin{aligned}
T(n) &\leq d \cdot k \cdot n + \sum_{i=1}^{k} T(n_i) \\
&\quad \text{nach Induktionsvoraussetzung ist } T(n_i) \leq c \cdot n_i^2 \\
&\leq d \cdot k \cdot n + \sum_{i=1}^{k} c \cdot n_i^2 \\
&= d \cdot k \cdot n + c \sum_{i=1}^{k} n_i(n - n + n_i) \\
&= d \cdot k \cdot n + c \sum_{i=1}^{k} n_i \cdot n - c \sum_{i=1}^{k} n_i(n - n_i) \\
&\quad \text{da } x(c-x) > 1(c-1) \text{ für } x \in [1 : c-1] \text{, siehe auch Abbildung 7.14} \\
&\leq d \cdot k \cdot n + cn^2 - c \sum_{i=1}^{k} 1(n-1)
\end{aligned}$$

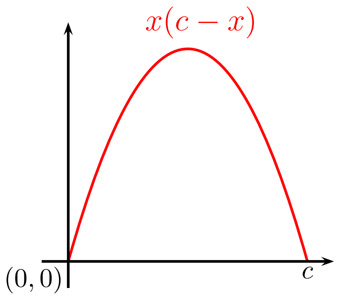


Abbildung 7.14: Skizze: Die Funktion $[0, c] \to \mathbb{R}_+ : x \mapsto x(c-x)$

$$
\begin{aligned}
&\leq \quad d \cdot k \cdot n + c \cdot n^2 - c \cdot k(n-1) \\
&\qquad \text{da } c \geq 2d \\
&\leq \quad d \cdot k \cdot n + c \cdot n^2 - 2d \cdot k(n-2) \\
&\qquad \text{da } 2(n-1) \geq n \text{ für } n \geq 3 \\
&\leq \quad d \cdot k \cdot n + c \cdot n^2 - d \cdot k \cdot n \\
&= \quad c \cdot n^2.
\end{aligned}
$$

Damit ist der Induktionsschluss vollzogen und der Beweis beendet. ■

7.3 Additive Distanzen und Bäume

Leider sind nicht alle Distanzmatrizen ultrametrisch. Wir wollen jetzt ein größere Klasse von Matrizen vorstellen, zu denen sich evolutionäre Bäume konstruieren lassen, sofern wir auf eine explizite Wurzel in diesen Bäumen verzichten können.

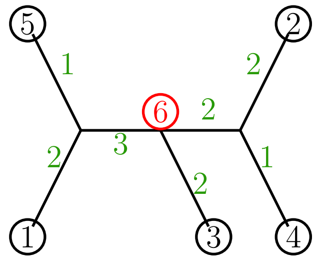
7.3.1 Additive Bäume

Zunächst einmal definieren wir, was wir unter additiven Bäumen, d.h. evolutionären Bäumen ohne Wurzel, verstehen wollen.

Definition 7.15 *Sei D eine $n \times n$-Distanzmatrix. Sei T ein Baum mit mindestens n Knoten und positiven Kantengewichten, wobei einige Knoten bijektiv mit Werten aus $[1:n]$ markiert sind. Dann ist T ein* additiver Baum *für D, wenn der Pfad vom Knoten mit Markierung i zum Knoten mit Markierung j in T das Gewicht d_{ij} besitzt.*

Wie bei den ultrametrischen Bäumen besitzt auch nicht jede Distanzmatrix einen additiven Baum. Des Weiteren sind nicht markierte Blätter in einem additiven Baum überflüssig, da jeder Pfad innerhalb eines Baum nur dann ein Blatt berührt, wenn dieses ein Endpunkt des Pfades ist. Daher nehmen wir im Folgenden ohne Beschränkung der Allgemeinheit an, dass ein additiver Baum nur markierte Blätter besitzt. In der folgenden Abbildung 7.15 ist noch einmal ein Beispiel einer Matrix samt ihres zugehörigen additiven Baumes angegeben.

D	1	2	3	4	5	6
1	0	9	7	8	3	5
2		0	6	3	8	4
3			0	5	6	2
4				0	7	3
5					0	4
6						0



Gewicht des Pfades zwischen 5 und 4: $\Rightarrow 1+3+2+1=7$
Gewicht des Pfades zwischen 1 und 5: $\Rightarrow 2+1=3$

Abbildung 7.15: Beispiel: Eine Matrix und der zugehörige additive Baum

Definition 7.16 *Sei D eine Distanzmatrix. Besitzt D einen additiven Baum, so heißt D eine* additive Matrix. *Ein additiver Baum heißt* kompakt, *wenn alle Knoten markiert sind (insbesondere auch die inneren Knoten). Ein additiver Baum heißt* extern, *wenn nur Blätter markiert sind.*

In der Abbildung 7.15 ist der Baum ohne Knoten 6 (und damit die Matrix ohne Zeile und Spalte 6) ein externer additiver Baum. Durch Hinzufügen von zwei Markierungen (und zwei entsprechenden Spalten und Zeilen in der Matrix) könnte dieser Baum zu einem kompakten additiven Baum gemacht werden.

Lemma 7.17 *Sei D eine additive Matrix, dann induziert D eine Metrik.*

Beweis: Da D eine Distanzmatrix ist, gelten die Definitheit und Symmetrie unmittelbar. Die Dreiecksungleichung sieht man leicht, wenn man sich die entsprechenden Pfade im zu D gehörigen additiven Baum betrachtet. ∎

Die Umkehrung gilt übrigens nicht, wie wir später noch zeigen werden.

7.3.2 Charakterisierung additiver Bäume

Wir wollen nun eine Charakterisierung additiver Matrizen angeben, mit deren Hilfe sich auch gleich ein zugehöriger additiver Baum konstruieren lässt. Zunächst einmal zeigen wir, dass ultrametrische Bäume spezielle additive Bäume sind.

Lemma 7.18 *Sei D eine additive Matrix. D ist genau dann ultrametrisch, wenn es einen additiven Baum T für D gibt, so dass es in T einen Knoten v (zentraler Knoten) gibt, der zu allen markierten Knoten denselben Abstand besitzt.*

Beweis: **⇒:** Sei T ein ultrametrischer Baum für D mit Knotenmarkierungen μ. Wir erhalten daraus einen additiven Baum T', indem wir als Kantengewicht einer Kante (v, w) folgendes wählen:

$$\gamma(v, w) = \frac{1}{2}(\mu(v) - \mu(w)).$$

Man rechnet jetzt leicht nach, dass für zwei Blätter i und j das sowohl das Gewicht des Weges von i nach $\mathrm{lca}(i, j)$ als auch das Gewicht von j nach $\mathrm{lca}(i, j)$ gerade $\frac{1}{2} \cdot d_{ij}$ beträgt. Die Länge des Weges von i nach j beträgt daher im additiven Baum wie gefordert d_{ij}.

Falls einen Knoten mit Grad 2 in einem solchen additiven Baum stören (wie sie beispielweise von der Wurzel konstruiert werden), der kann diese, wie in Abbildung 7.16 illustriert, eliminieren. In dieser Abbildung stellt der rote Knoten den zentralen Knoten des additiven Baumes dar.

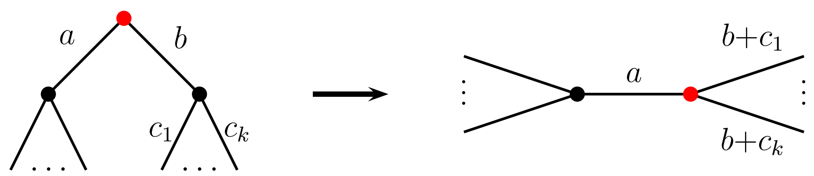


Abbildung 7.16: Skizze: Elimination von Knoten mit Grad 2

⇐: Sei D additiv und sei v der Knoten des additiven Baums T, der von allen Blättern den gleichen Abstand hat. Man überlegt sich leicht, dass T dann extern additiv sein muss.

Betrachtet man v als Wurzel, so gilt für beliebige Blätter i, j, dass der Abstand zwischen dem kleinsten gemeinsamen Vorfahren ℓ für beide Blätter gleich ist. (Da der Abstand $d_{\ell v}$ fest ist, wäre andernfalls der Abstand der Blätter zu v nicht gleich). Wir zeigen jetzt, dass für drei beliebige Blätter i, j, k die ultrametrische Dreiecksungleichung $d_{ik} \leq \max\{d_{ij}, d_{jk}\}$ gilt. Sei dazu $\text{lca}(i, j)$ der kleinste gemeinsame Vorfahre von i, j.

Falls $\text{lca}(i, j) = \text{lca}(i, k) = \text{lca}(j, k)$ die gleichen Knoten sind, so ist $d_{ik} = d_{ij} = d_{jk}$ und die Dreiecksungleichung gilt mit Gleichheit.

Daher sei jetzt ohne Beschränkung der Allgemeinheit $\text{lca}(i, j) \neq \text{lca}(i, k)$ und dass $\text{lca}(i, k)$ näher an der Wurzel liegt. Da $\text{lca}(i, j)$ und $\text{lca}(i, k)$ auf dem Weg von i zur Wurzel liegen gilt entweder $\text{lca}(j, k) = \text{lca}(j, i)$ oder $\text{lca}(j, k) = \text{lca}(i, k)$ sein.

Sei ohne Beschränkung der Allgemeinheit $\text{lca}(j, k) = \text{lca}(i, k) = \text{lca}(i, j, k)$. Dann gilt:

$$\begin{aligned}
d_{ij} &= w(i, \text{lca}(i, j)) + w(j, \text{lca}(i, j)) \\
&= 2 \cdot w(j, \text{lca}(i, j)), \\[1em]
d_{ik} &= w(i, \text{lca}(i, j)) + w(\text{lca}(i, j), \text{lca}(i, j, k)) + w(\text{lca}(i, j, k), k) \\
&= 2 \cdot w(\text{lca}(i, j, k), k), \\[1em]
d_{jk} &= w(j, \text{lca}(i, j)) + w(\text{lca}(i, j), \text{lca}(i, j, k)) + w(\text{lca}(i, j, k), k) \\
&= 2 \cdot w(\text{lca}(i, j, k), k)
\end{aligned}$$

Daher gilt unter anderem $d_{ij} \leq d_{ik}$, $d_{ik} \leq d_{jk}$ und $d_{jk} \leq d_{ik}$. Somit ist die Dreiecksungleichung immer erfüllt. ■

Wie können wir nun für eine Distanzmatrix entscheiden, ob sie additiv ist, und falls ja, beschreiben, wie der zugehörige additive Baum aussieht? Im vorherigen Lemma haben wir gesehen, wie wir das Problem auf ultrametrische Matrizen zurückführen können.

Wir wollen dies noch einmal mir einer anderen Charakterisierung tun. Wir betrachten zuerst die gegebene Matrix D. Diese ist genau dann additiv, wenn sie einen additiven Baum T_D besitzt. Wenn wir diesen Baum wurzeln und die Kanten zu den Blättern so verlängernd, dass alle Pfade von der Wurzel zu den Blättern gleiche Gewicht besitzen, dann ist T'_D ultrametrisch. Daraus können wir dann eine Distanzmatrix $D(T'_D)$ ablesen, die ultrametrisch ist, wenn D additiv ist. Dies ist in der folgenden Abbildung 7.17 schematisch dargestellt.

Wir wollen also die gebebene Matrix D so modifizieren, dass daraus eine ultrametrische Matrix wird. Wenn diese Idee funktioniert können wir eine Matrix auf Additivität hin testen, indem wir die zugehörige, neu konstruierte Matrix auf Ultrametrik hin testen. Für Letzteres haben wir ja bereits einen effizienten Algorithmus kennen gelernt.

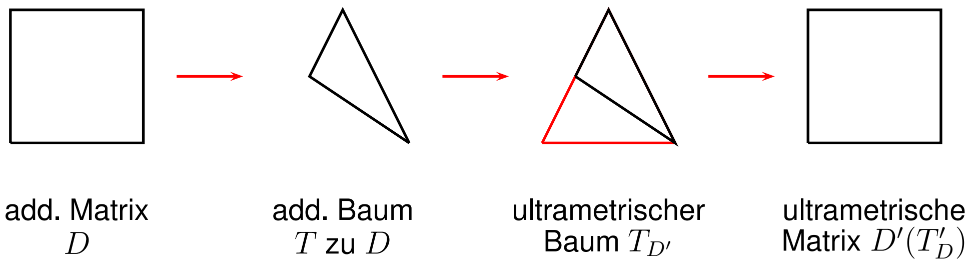


Abbildung 7.17: Skizze:

Sei im Folgenden D eine additive Matrix und d_{ij} ein maximaler Eintrag,d h.

$$d_{ij} \geq \max \{d_{k\ell} \, : \, k, \ell \in [1:n]\} \, .$$

Weiter sei T_D der additive Baum zu D. Wir wurzeln jetzt diesen Baum am Knoten i. Man beachte, dass i ein Blatt ist. Wir kommen später noch darauf zurück, wie wir dafür sorgen dass i ein Blatt bleibt Dies ist in der folgenden Abbildung 7.18 illustriert.

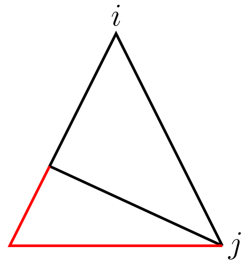


Abbildung 7.18: Skizze: Wurzeln von T_D am Blatt i

Unser nächster Schritt besteht jetzt darin, alle Blätter (außer i) auf denselben Abstand zur neuen Wurzel zu bringen. Wir betrachten dazu jetzt ein Blatt k des neuen gewurzelten Baumes. Wir versuchen die zu k inzidente Kante jetzt so zu verlängern, dass der Abstand von k zur Wurzel i auf d_{ij} anwächst. Dazu setzen wir das Kantengewicht auf d_{ij} und verkürzen es um das Gewicht des restlichen Pfades von k zu i, d.h. um $(d_{ik} - d)$, wobei d das Gewicht der zu k inzidenten Kante ist. Dies ist in Abbildung 7.19 noch einmal illustriert. War der Baum vorher additiv, so ist er jetzt ultrametrisch, wenn wir als Knotenmarkierung jetzt das Gewicht des Pfades einen Knotens zu einem seiner Blätter (die jetzt alle gleich sein müssen) wählen.

Somit haben wir aus einem additiven einen ultrametrischen Baum gemacht. Jedoch haben wir dazu den additiven Baum benötigt, den wir eigentlich erst konstruieren

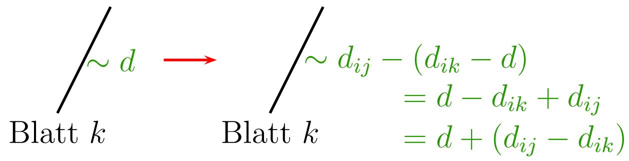


Abbildung 7.19: Skizze: Verlängern der zu Blättern inzidenten Kanten

wollen. Es stellt sich nun die Frage, ob wir die entsprechende ultrametrische Matrix aus der additiven Matrix direkt, ohne Kenntnis des zugehörigen additiven Baumes berechnen können. Betrachten wir dazu noch einmal zwei Blätter im additiven Baum und versuchen den entsprechenden Abstand im ultrametrischen Baum berechnen. Dazu betrachten wir die Abbildung 7.20.

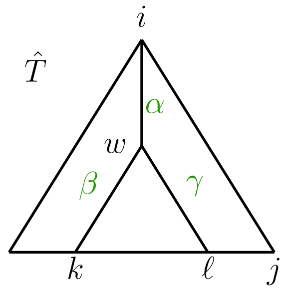


Abbildung 7.20: Skizze: Abstände im gewurzelten additiven Baum $\hat{T}$

Seien k und ℓ zwei Blätter, für die wir den Abstand in der entsprechenden ultrametrischen Matrix bestimmen wollen. Sei w der niedrigste gemeinsame Vorfahre von k und ℓ im gewurzelten additiven Baum $\hat{T}$ und α, β bzw. γ die Abstände im gewurzelten additiven Baum $\hat{T}$ zwischen der Wurzel und w, w und k bzw. w und ℓ. Es gilt dann

$$\begin{aligned}\beta &= d_{ik} - \alpha\\ \gamma &= d_{il} - \alpha\end{aligned}$$

Im ultrametrischen Baum T'_D gilt dann:

$$\begin{aligned}d_{T'}(w,k) &= d_{T'}(w,\ell)\\ &= \beta + (d_{ij} - d_{ik})\\ &= \gamma + (d_{ij} - d_{i\ell}).\end{aligned}$$

Damit gilt:

$$d_{T'}(w,k) = \beta + d_{ij} - d_{ik}$$

$$\begin{aligned} &= d_{ik} - \alpha + d_{ij} - d_{ik} \\ &= d_{ij} - \alpha \end{aligned}$$

und analog:

$$\begin{aligned} d_{T'}(w,\ell) &= \gamma + d_{ij} - d_{i\ell} \\ &= d_{i\ell} - \alpha + d_{ij} - d_{i\ell} \\ &= d_{ij} - \alpha. \end{aligned}$$

Wir müssen jetzt nur noch α bestimmen. Aus der Skizze in Abbildung 7.20 folgt sofort, wenn wir die Gewichte der Pfade von i nach k sowie ℓ addieren und davon das Gewicht des Pfades von k nach ℓ subtrahieren:

$$2\alpha = d_{ik} + d_{i\ell} - d_{k\ell}.$$

Daraus ergibt sich:

$$\begin{aligned} d_{T'}(w,k) &= d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell}), \\ d_{T'}(w,\ell) &= d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell}). \end{aligned}$$

Somit können wir jetzt die ultrametrische Matrix D' direkt aus der additiven Matrix D berechnen:

$$d'_{k\ell} := d_{T'}(w,k) = d_{T'}(w,\ell) = d_{ij} - \frac{1}{2}(d_{i\ell} + d_{ik} - d_{k\ell}).$$

Wir müssen uns jetzt nur noch überlegen, dass dies wirklich eine ultrametrische Matrix ist, da wir den additiven Baum ja am Blatt i gewurzelt haben. Damit würden wir sowohl ein Blatt verlieren, nämlich i, als auch keinen echten ultrametrischen Baum generieren, da dessen Wurzel nur ein Kind anstatt mindestens zweier besitzt. In Wirklichkeit wurzeln wir den additiven Baum am zu i adjazenten Knoten, wie in Abbildung 7.21 illustriert. Damit erhalten wir einen echten ultrametrischen Baum.

Damit haben wir das folgende Lemma bewiesen.

Lemma 7.19 *Sei D eine additive Matrix, deren maximaler Eintrag d_{ij} ist, dann ist D' mit*

$$d'_{kl} = d_{ij} - \frac{1}{2}(d_{i\ell} + d_{ik} - d_{kl\ell})$$

eine ultrametrische Matrix.

Es wäre schön, wenn auch die umgekehrte Richtung gelten würde, nämlich, dass wenn D' ultrametrisch ist, dass dann bereits D additiv ist. Dies ist leider nicht der

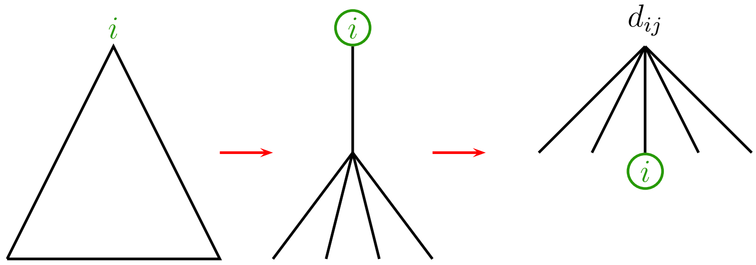


Abbildung 7.21: Skizze: Wirkliches Wurzeln des additiven Baumes

D	1	2	3
1	0	8	4
2		0	2
3			0

D'	1	2	3
1	0	8	8
2		0	3
3			0

$$d'_{12} = 8 - \frac{1}{2}(0 + 8 - 8) = 8$$
$$d'_{13} = 8 - \frac{1}{2}(0 + 4 - 4) = 8$$
$$d'_{23} = 8 - \frac{1}{2}(8 + 4 - 2) = 3$$

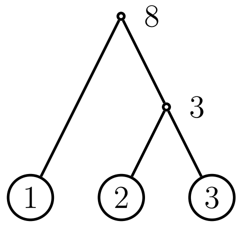


Abbildung 7.22: Gegenbeispiel: D nicht additiv, aber D' ultrametrisch

.

Fall, auch wenn dies in vielen Lehrbüchern und Skripten fälschlicherweise behauptet wird. Wir geben hierfür ein Gegenbeispiel in Abbildung 7.22. Man sieht leicht, dass D' ultrametrisch D ist hingegen nicht additiv, weil $d(1,2) = 8$, aber

$$d(1,3) + d(3,2) = 4 + 2 = 6 < 8$$

gilt. Im Baum muss also der Umweg über 3 größer sein als der direkte Weg, was nicht sein kann (siehe auch Abbildung 7.23), denn im additiven Baum gilt die normale Dreiecksungleichung (siehe Lemma 7.17).

Dennoch können wir mit einer weiteren Zusatzbedingung dafür sorgen, dass das Lemma in der von uns gewünschten Weise retten können.

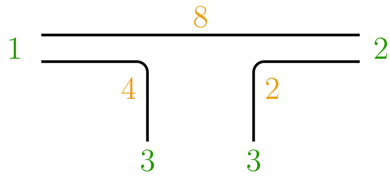


Abbildung 7.23: Gegenbeispiel: „Unmöglicher“ additiver Baum

Lemma 7.20 *Sei D eine Distanzmatrix und sei d_{ij} ein maximaler Eintrag von D. Weiter sei D' durch $d'_{k\ell} = d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell})$ definiert. Wenn D' eine ultrametrische Matrix ist und wenn für jedes Blatt b im zugehörigen ultrametrischen Baum $T(D')$ für das Gewicht γ der zu b inzidenten Kante gilt: $\gamma \geq (d_{ij} - d_{bi})$, dann ist D additiv.*

Beweis: Sei $T' := T(D')$ ein ultrametrischer Baum für D'. Weiter sei $(v, w) \in E(T')$ und es seien p, q, r, s Blätter von T', so dass $\text{lca}(p, q) = v$ und $\text{lca}(r, s) = w$. Dies ist in Abbildung 7.24 illustriert. Hierbei ist q zweimal angegeben ist, da a priori nicht klar ist, ob q ein Nachfolger von w ist oder nicht. Wir definieren dann das

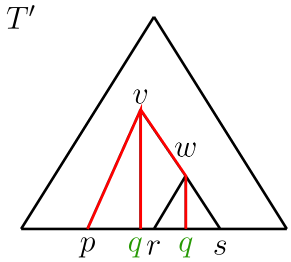


Abbildung 7.24: Skizze: Kante (v, w) in T'

Kantengewicht von (v, w) durch:

$$\gamma(v, w) = d'_{pq} - d'_{rs}.$$

Damit ergibt sich für

$$\begin{aligned} d_{T'}(k, \ell) &= 2 \cdot d'_{k,\ell} \\ &= 2(d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell})) \\ &= 2d_{ij} - d_{ik} - d_{i\ell} + d_{k\ell} \end{aligned}$$

Wenn wir jetzt für jedes Blatt b das Gewicht der inzidenten Kante um $d_{ij} - d_{bi}$ erniedrigen, erhalten wir einen neuen additiven Baum T. Dann nach Voraussetzung, das Gewicht einer solchen zu b inzidenten Kante größer als $d_{ij} - d_{bi}$ ist, bleiben die Kantengewichte von T positiv. Weiterhin gilt in T:

$$\begin{aligned} d_T(k,\ell) &= d_{T'}(k,\ell) - (d_{ij} - d_{ik}) - (d_{ij} - d_{i\ell}) \\ &= (2d_{ij} - d_{ik} - d_{i\ell} + d_{k\ell}) - d_{ij} + d_{ik} - d_{ij} + d_{i\ell} \\ &= d_{k\ell}. \end{aligned}$$

Somit ist T ein additiver Baum für D und das Lemma ist bewiesen. ■

Gehen wir noch einmal zurück zu unserem Gegenbeispiel, das besagte, dass die Ultrametrik von D' nicht ausreicht um die Additivität von D zu zeigen. Wir schauen einmal was hier beim Kürzen der Gewichte von zu Blättern inzidenten Kanten passieren kann. Dies ist in Abbildung 7.25 illustriert. Wir sehen hier, dass der Baum zwar die gewünschten Abstände besitzt, jedoch negative Kantengewichte erzeugt, die bei additiven Bäumen nicht erlaubt sind.

D	1	2	3
1	0	8	4
2		0	2
3			0

D'	1	2	3
1	0	8	8
2		0	3
3			0

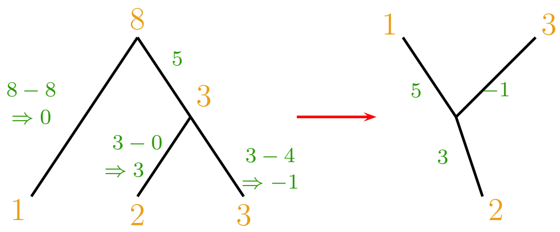


Abbildung 7.25: Gegenbeispiel: D nicht additiv und D' ultrametrisch (Fortsetzung)

7.3.3 Algorithmus zur Erkennung additiver Matrizen

Aus der Charakterisierung der additiven Matrizen mit Hilfe ultrametrischer Matrizen lässt sich der folgende Algorithmus zur Erkennung additiver Matrizen und der Konstruktion zugehöriger additiver Bäume herleiten. Dieser ist in Abbildung 7.26 aufgelistet. Es lässt sich leicht nachrechnen, dass dieser Algorithmus eine Laufzeit von $O(n^2)$ besitzt.

Wir müssen uns nur noch kurz um die Korrektheit kümmern. Nach Lemma 7.19 wissen wir, dass in Schritt 2 die richtige Entscheidung getroffen wird. Nach Lemma 7.20 wird auch im Schritt 4 die richtige Entscheidung getroffen. Also ist der Algorithmus korrekt. Fassen wir das Ergebnis noch zusammen.

Theorem 7.21 *Es kann in Zeit $O(n^2)$ entschieden werden, ob eine gegebene $n \times n$-Distanzmatrix additiv ist oder nicht. Falls die Matrix additiv ist, kann der zugehörige additive Baum in Zeit $O(n^2)$ konstruiert werden.*

1. Konstruiere aus der gegebenen $n \times n$-Matrix D eine neue $n \times n$-Matrix D' mittels $d'_{k\ell} = d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell})$, wobei d_{ij} ein maximaler Eintrag von D ist.
2. Teste D' auf Ultrametrik. Ist D' nicht ultrametrisch, dann ist D nicht additiv.
3. Konstruiere den zu D' gehörigen ultrametrischen Baum T'.
4. Teste, ob sich die Kantengewichte für jedes Blatt b um $d_{ij} - d_{ib}$ erniedrigen lässt. Falls dies nicht geht, ist D nicht additiv, andernfalls erhalten wir einen additiven Baum T für D.

Abbildung 7.26: Algorithmus: Erkennung additiver Matrizen

7.3.4 4-Punkte-Bedingung

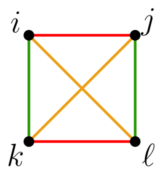
Zum Abschluss wollen wir noch eine andere Charakterisierung von additiven Matrizen angeben, die so genannte *4-Punkte-Bedingung von Buneman*.

Lemma 7.22 (Bunemans 4-Punkte-Bedingung) *Eine $n \times n$-Distanzmatrix D ist genau dann additiv, wenn für je vier Punkte $i, j, k, \ell \in [1:n]$ mit*

$$d_{i\ell} + d_{jk} = \min\{d_{ij} + d_{k\ell}, d_{ik} + d_{j\ell}, d_{i\ell} + d_{jk}\}$$

gilt, dass $d_{ij} + d_{k\ell} = d_{ik} + d_{j\ell}$.

Diese 4-Punkte-Bedingung ist in Abbildung 7.27 noch einmal illustriert



$$d_{i\ell} + d_{jk} = \min\{d_{ij} + d_{k\ell}, d_{ik} + d_{i\ell}, d_{i\ell} + d_{jk}\}$$
$$\Rightarrow \quad d_{ij} + d_{k\ell} = d_{ik} + d_{j\ell}$$

Abbildung 7.27: Skizze: 4-Punkte-Bedingung

Beweis: **⇒:** Sei T ein additiver Baum für D. Wir unterscheiden jetzt drei Fälle, je nachdem, wie die Pfade in T verlaufen.

Fall 1: Im ersten Fall nehmen wir an, die Pfade von i nach j und k nach ℓ knotendisjunkt sind. Dies ist in Abbildung 7.28 illustriert. Dann ist aber $d_{i\ell} + d_{jk}$ nicht minimal und somit ist nichts zu zeigen.

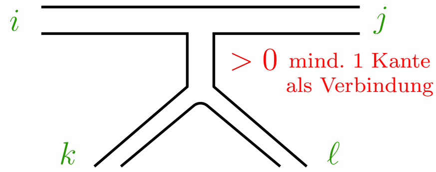


Abbildung 7.28: Skizze: Die Pfade $i \to j$ und $k \to \ell$ sind knotendisjunkt

Fall 2: Im zweiten Fall nehmen wir an, die Pfade von i nach k und j nach ℓ knotendisjunkt sind. Dies ist in Abbildung 7.29 illustriert. Dann ist jedoch ebenfalls $d_{i\ell} + d_{jk}$ nicht minimal und somit ist nichts zu zeigen.

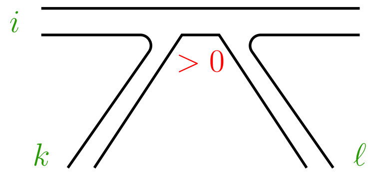


Abbildung 7.29: Skizze: Die Pfade $i \to k$ und $j \to \ell$ sind knotendisjunkt

Fall 3: Im dritten und letzten Fall nehmen wir an, die Pfade von i nach ℓ und j nach k kantendisjunkt sind und sich daher höchstens in einem Knoten schneiden. Dies ist in Abbildung 7.30 illustriert. Nun gilt jedoch, wie man leicht der Abbildung 7.30

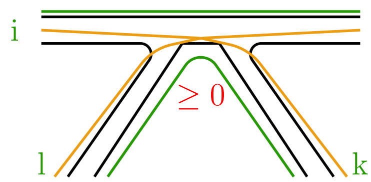


Abbildung 7.30: Skizze: Die Pfade $i \to \ell$ und $j \to k$ sind kantendisjunkt

entnehmen kann:

$$d_{ij} = d_{ik} + d_{j\ell} - d_{k\ell}$$

und somit

$$d_{ij} + d_{k\ell} = d_{ik} + d_{j\ell}.$$

$\Leftarrow$: Betrachte D' mit $d'_{k\ell} := d_{ij} - \frac{1}{2}(d_{ik} + d_{i\ell} - d_{k\ell})$, wobei d_{ij} ein maximaler Eintrag von D ist. Es genügt zu zeigen, dass D' ultrametrisch ist.

Betrachte $p, q, r \in [1 : n]$ mit $d'_{pq} \leq d'_{pr} \leq d'_{qr}$. Es ist dann zu zeigen: $d'_{pr} = d'_{qr}$. Aus $d'_{pq} \leq d'_{pr} \leq d'_{qr}$ folgt dann:

$$2d_{ij} - d_{ip} - d_{iq} + d_{pq} \; \leq \; 2d_{ij} - d_{ip} - d_{ir} + d_{pr} \; \leq \; 2d_{ij} - d_{iq} - d_{ir} + d_{qr}.$$

Daraus folgt:

$$d_{pq} - d_{ip} - d_{iq} \; \leq \; d_{pr} - d_{ip} - d_{ir} \; \leq \; d_{qr} - d_{iq} - d_{ir}$$

und weiter

$$d_{pq} + d_{ir} \; \leq \; d_{pr} + d_{iq} \; \leq \; d_{qr} + d_{ip}.$$

Aufgrund der 4-Punkt-Bedingung folgt unmittelbar

$$d_{pr} + d_{iq} = d_{qr} + d_{ip}.$$

Nach Addition von $(2d_{ij} - d_{ir})$ folgt:

$$(2d_{ij} - d_{ir}) + d_{pr} + d_{iq} = (2d_{ij} - d_{ir}) + d_{qr} + d_{ip}.$$

Nach kurzer Rechnung folgt:

$$2d_{ij} - d_{ir} - d_{ip} + d_{pr} = 2d_{ij} - d_{ir} - d_{iq} + d_{qr}$$

und somit gilt

$$2d'_{pr} = 2d'_{qr}.$$

Also ist D' nach Lemma 7.4 ultrametrisch und somit ist nach Lemma7.20 D additiv. ∎

Mit Hilfe dieser Charakterisierung werden wir noch zeigen, dass es Matrizen gibt, die eine Metrik induzieren, aber keinen additiven Baum besitzen. Dieses Gegenbeispiel

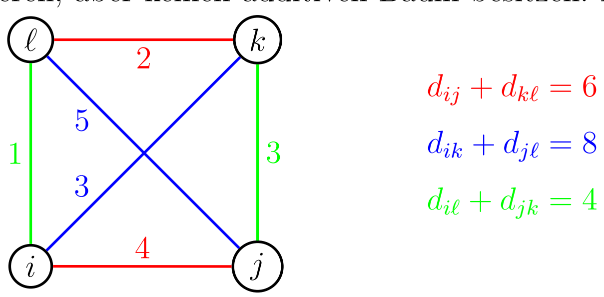


Abbildung 7.31: Gegenbeispiel: Metrische Matrix, die nicht additiv ist

ist in Abbildung 7.31 angegeben. Man sieht leicht, dass die 4-Punkte-Bedingung nicht gilt und somit die Matrix nicht additiv ist. Man überprüft leicht, dass für jedes Dreieck die Dreiecksungleichung gilt, die Abstände also der Definition einer Metrik genügen.

7.3.5 Charakterisierung kompakter additiver Bäume

In diesem Abschnitt wollen wir eine schöne Charakterisierung kompakter additiver Bäume angeben. Zunächst benötige wir noch einige grundlegenden Definitionen aus der Graphentheorie.

Definition 7.23 *Sei $G = (V, E)$ ein ungerichteter 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 definieren.

Definition 7.24 *Sei $G = (V, E)$ ein ungerichteter Graph. Ein Teilgraph $G' \subset G$ heißt* Spannbaum, *wenn G' ein aufspannender Teilgraph von G ist und G' ein Baum ist.*

Für gewichtete Graphen brauchen wir jetzt noch das Konzept eines minimalen Spannbaumes.

Definition 7.25 *Sei $G = (V, E, \gamma)$ ein gewichteter ungerichteter 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, *wenn er unter allen möglichen Spannbäumen für G minimales Gewicht besitzt, d.h.*

$$\gamma(T) \leq \min \{\gamma(T') \,:\, T' \text{ ist ein Spannbaum von } G\}.$$

Im Folgenden werden wir der Kürze wegen einen minimalen Spannbaum oft auch mit MST (engl. *minimum spanning tree*) abkürzen.

Definition 7.26 *Sei D eine $n \times n$-Distanzmatrix. Dann ist $G(D) = (V, E)$ der zu D* gehörige gewichtete Graph*, wobei*

$$\begin{aligned} V &= [1 : n], \\ E &= \binom{V}{2} := \{\{v, w\} \,:\, v \neq w \in V\}\,, \\ \gamma(v, w) &= D(v, w). \end{aligned}$$

Nach diesen Definitionen kommen wir zu dem zentralen Lemma dieses Abschnittes, der kompakte additive Bäume charakterisiert.

Theorem 7.27 *Sei D eine $n \times n$-Distanzmatrix. Besitzt D einen kompakten additiven Baum T, dann ist T der minimale Spannbaum von $G(D)$ und ist eindeutig bestimmt.*

Beweis: Sei T ein kompakter additiver Baum für D (dieser muss natürlich nicht gewurzelt sein). Wir betrachten ein Knotenpaar (x, y), das durch keine Kante in T

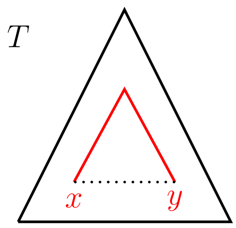


Abbildung 7.32: Skizze: Pfad von x nach y in T

verbunden ist. Sei $p = (v_0, v_1, \ldots, v_k)$ mit $v_0 = x$ und $v_k = y$ sowie $k \geq 2$ der Pfad von x nach y in T. $\gamma(p)$ entspricht $D(x, y)$, weil T ein additiver Baum für D ist. Da nach Definition einer Distanzmatrix alle Kantengewichte positiv sind und der Pfad aus mindestens zwei Kanten besteht, ist $\gamma(v_{i-1}, v_i) < D(x, y) = \gamma(x, y)$ für alle Kanten (v_{i-1}, v_i) des Pfades p.

Sei jetzt T' ein minimaler Spannbaum von $G(D)$. Wir nehmen jetzt an, dass die Kante (x, y) eine Kante des minimalen Spannbaumes ist, d.h. $(x, y) \in E(T')$. Somit verbindet die Kante (x, y) zwei Teilbäume von T'. Dies ist in Abbildung 7.32 illustriert.

Da (x, y) keine Kante im Pfad von x nach y im kompakten additiven Baum von T ist, verbindet der Pfad p die beiden durch Entfernen der Kante (x, y) separierten

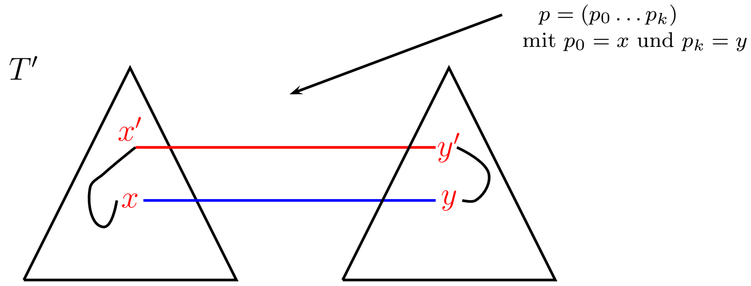


Abbildung 7.33: Skizze: Spannbaum T' von $D(G)$

Teilbäume des minimalen Spannbaumes T'. Sei (x', y') die erste Kante auf dem Pfad p von x nach y, die nicht innerhalb einer der beiden separierten Spannbäume verläuft. Wie wir oben gesehen haben, gilt $\gamma(x', y') < \gamma(x, y)$.

Wir konstruieren jetzt einen neuen Spannbaum T'' für $G(D)$ wie folgt:

$$\begin{aligned}
V(T'') &= V(T') = V \\
E(T'') &= (E(T') \setminus \{\{x, y\}\}) \cup \{\{x', y\}\}
\end{aligned}$$

Für das Gewicht des Spannbaumes T'' gilt dann:

$$\begin{aligned}
\gamma(T'') &= \sum_{e \in E(T'')} \gamma(e) \\
&= \sum_{e \in E(T')} \gamma(e) - \gamma(x, y) + \gamma(x', y') \\
&= \sum_{e \in E(T')} \gamma(e) + \underbrace{(\gamma(x', y') - \gamma(x, y))}_{<0} \\
&< \gamma(T').
\end{aligned}$$

Somit ist $\gamma(T'') < \gamma(T)$ und dies liefert den gewünschten Widerspruch zu der Annahme, dass T' ein minimaler Spannbaum von $G(D)$ ist.

Somit gilt für jedes Knotenpaar (x, y) mit $(x, y) \notin E(T)$, dass dann die Kante (x, y) nicht im minimalen Spannbaum von $G(D)$ sein kann, d.h. $(x, y) \notin T'$. Also ist eine Kante, die sich nicht im kompakten additiven Baum befindet, auch keine Kante des Spannbaums.

Dies gilt sogar für zwei verschiedene minimale Spannbäume für $G(D)$. Somit kann es nur einen minimalen Spannbaum geben. ■

7.3.6 Konstruktion kompakter additiver Bäume

Die Information aus dem vorherigen Lemma kann man dazu ausnutzen, um einen effizienten Algorithmus zur Erkennung kompakter additiver Matrizen zu entwickeln. Sei D die Matrix, für den der kompakte additive Baum konstruiert werden soll, und somit $G(D) = (V, E, \gamma)$ der gewichtete Graph, für den der minimale Spannbaum berechnet werden soll.

Wir beginnen mit der Knotenmenge $V' = \{v\}$ für eine beliebiges $v \in V$ und der leeren Kantenmenge $E' = \emptyset$. Der Graph (V', E') soll letztendlich der gesuchte minimale Spannbaum werden. Wir verwenden hier Prims Algorithmus zur Konstruktion eines minimalen Spannbaumes, der wieder ein Greedy-Algorithmus sein wird. Wir versuchen die Menge V' in jedem Schritt um einen Knoten aus $V \setminus V'$ zu erweitern. Von allen solchen Kandidaten wählen wir eine Kante minimalen Gewichtes. Dies ist schematisch in Abbildung 7.34 dargestellt.

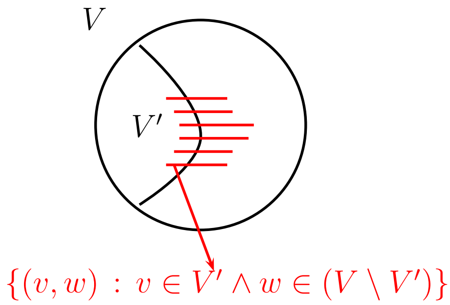


Abbildung 7.34: Skizze: Erweiterung von V'

Den Beweis, dass wir hiermit einen minimalen Spannbaum konstruieren, wollen wir an dieser Stelle nicht führen. Wir verweisen hierzu auf die einschlägige Literatur. Den formalen Algorithmus haben wir in Abbildung 7.35 angegeben. Bis auf die blauen Zeilen ist dies genau der Pseudo-Code für Prims Algorithmus zur Konstruktion eines minimalen Spannbaums.

Die blaue for-Schleife ist dazu da, um zu testen, ob der konstruierte minimale Spannbaum wirklich ein kompakter additiver Baum für D ist. Zum einen setzen wir den konstruierten Abstand d_T für den konstruierten minimalen Spannbaum. Der nachfolgende Test überprüft, ob dieser Abstand d_T mit der vorgegebenen Distanzmatrix γ übereinstimmt.

Wir müssen uns jetzt nur noch die Laufzeit überlegen. Die while-Schleife wird genau $n = |V|$ Mal durchlaufen. Danach ist $V' = V$. Die Minimumsbestimmung lässt sich sicherlich in Zeit $O(n^2)$ erledigen, da maximal n^2 Kanten zu durchsuchen sind. Daraus folgt eine Laufzeit von $O(n^3)$.

```
Modified_Prim (V, E, γ)
{
    E' := ∅;
    V' := {v} für ein beliebiges v ∈ V;
    /* (V', E') wird am Ende der minimale Spannbaum sein */
    while (V' ≠ V)
    {
        Sei e = (x, y), so dass γ(x, y) = min {γ(v, w) : v ∈ V' ∧ w ∈ V \ V'};
        /* oBdA sei y ∈ V' */
        E' := E' ∪ {e};
        V' := V' ∪ {y};
        for all (v ∈ V');
        {
            d_T(v, y) := d_T(v, x) + γ(x, y);
            if (d_T(v, y) ≠ γ(v, y)) reject;
        }
    }
    return (V', E');
}
```

Abbildung 7.35: Algorithmus: Konstruktion eines kompakten additiven Baumes

Implementiert man Prims-Algorithmus ein wenig geschickter, z.B. mit Hilfe von Priority-Queues, so lässt sich die Laufzeit auf $O(n^2)$ senken. Für die Details verweisen wir auch hier auf die einschlägige Literatur. Auch die blaue Schleife kann insgesamt in Zeit $O(n^2)$ implementiert werden, da jede for-Schleife maximal n-mal durchlaufen wird und die for-Schleife selbst maximal n-mal ausgeführt wird

Theorem 7.28 *Sei D eine $n \times n$-Distanzmatrix. Dann lässt sich in Zeit $O(n^2)$ entscheiden, ob ein kompakter additiver Baum für D existiert. Falls dieser existiert, kann dieser ebenfalls in Zeit $O(n^2)$ konstruiert werden.*

7.4 Perfekte binäre Phylogenie

In diesem Abschnitt wollen wir uns jetzt um charakterbasierte Methoden kümmern. Wir beschränken uns hier auf den Spezialfall, dass die Charaktere binär sind, d.h. nur zwei verschiedene Werte annehmen können.

7.4.1 Charakterisierung perfekter Phylogenie

Zunächst benötigen wir noch ein paar grundlegende Definitionen, um das Problem der Konstruktion phylogenetischer Bäume formulieren zu können.

Definition 7.29 *Eine* binäre Charaktermatrix M *ist eine binäre* $n \times m$*-Matrix. Ein* phylogenetischer Baum T *für eine binäre* $n \times m$*-Charaktermatrix ist ein ungeordneter gewurzelter Baum mit genau* n *Blättern, so dass:*

1. *Jedes der* n *Objekte aus* $[1 : n]$ *markiert genau eines der Blätter;*
2. *Jeder der* m *Charaktere aus* $[1 : m]$ *markiert genau eine Kante;*
3. *Für jedes Objekt* $p \in [1 : n]$ *gilt für die Menge* C_T *der Kantenmarkierungen auf dem Pfad von der Wurzel von* T *zu dem Blatt mit Markierung* p*, dass* $C_T = \{c : M_{c,p} = 1\}$.

In Abbildung 7.36 ist eine binäre Charaktermatrix samt seines zugehörigen phylogenetischen Baumes angegeben.

M	1	2	3	4	5	6	7
1	1	0	1	0	0	1	0
2	1	0	1	0	1	0	1
3	1	0	1	0	1	0	0
4	1	0	0	0	0	0	0
5	0	1	1	0	0	0	0
6	0	1	0	0	0	0	0

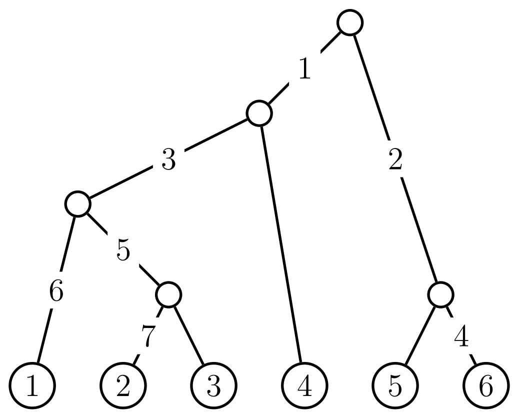


Abbildung 7.36: Beispiel: binäre Charaktermatrix und zugehöriger phylogenetischer Baum

Somit können wir das Problem der perfekten Phylogenie formulieren.

Perfekte Phylogenie

Eingabe: Eine binäre $n \times m$-Charaktermatrix M.
Ausgabe: Ein phylogenetischer Baum T für M.

Zunächst benötigen wir noch eine weitere Notation bevor wir uns der Lösung des Problems zuwenden können.

Notation 7.30 *Sei M eine binäre $n \times m$-Charaktermatrix, dann umfasst die Menge $O_j = \{i \in [1:n] : M_{i,j} = 1\}$ die Objekte, die den Charakter j besitzen.*

Wir geben zunächst wieder eine Charakterisierung an, wann eine binäre Charaktermatrix überhaupt einen phylogenetischen Baum besitzt.

Theorem 7.31 *Eine binäre $n \times m$-Charaktermatrix M besitzt genau dann einen phylogenetischen Baum, wenn für jedes Paar $i, j \in [1:n]$ entweder $O_i \cap O_j = \emptyset$ oder $O_i \subset O_j$ oder $O_i \supset O_j$ gilt.*

Beweis: **$\Rightarrow$:** Sei T ein phylogenetischer Baum für M. Sei e_i bzw. e_j die Kante in T mit der Markierung i bzw. j. Wir unterscheiden jetzt vier Fälle, je nachdem, wie sich die Kanten e_i und e_j zueinander im Baum T verhalten.

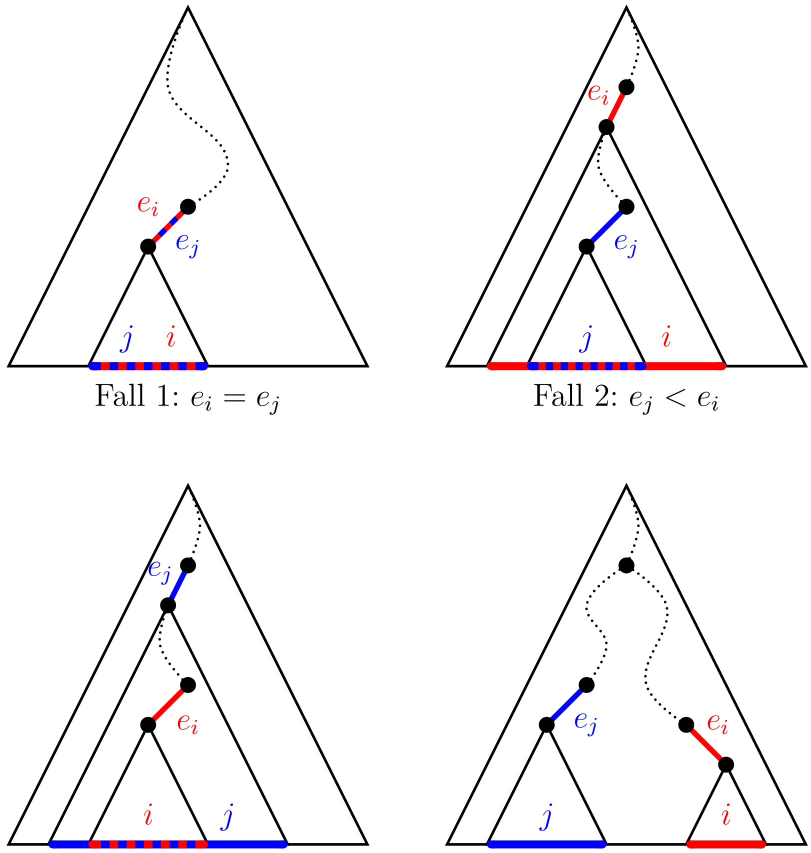
Fall 1: Wir nehmen zuerst an, dass $e_i = e_j$ ist (siehe auch Abbildung 7.37). In diesem Fall gilt offensichtlich $O_i = O_j$.

Fall 2: Nun nehmen wir an, dass sich im Teilbaum vom unteren Knoten vom e_i die Kante e_j befindet (siehe auch Abbildung 7.37). Also sind alle Nachfahren des unteren Knotens von e_j auch Nachfahren des unteren Knoten von e_i und somit gilt $O_j \subset O_i$.

Fall 3: Nun nehmen wir an, dass sich im Teilbaum vom unteren Knoten vom e_j die Kante e_i befindet (siehe auch Abbildung 7.37). Also sind alle Nachfahren des unteren Knotens von e_i auch Nachfahren des unteren Knoten von e_j und somit gilt $O_i \subset O_j$.

Fall 4: Der letzte verbleibende Fall ist, dass keine Kante Nachfahre eine anderen Kante ist (siehe auch Abbildung 7.37). In diesem Fall gilt offensichtlich $O_i \cap O_j = \emptyset$.

$\Leftarrow$: Wir müssen jetzt aus der Matrix M einen phylogenetischen Baum konstruieren. Zuerst nehmen wir ohne Beschränkung der Allgemeinheit an, dass die Spalten der Matrix M interpretiert als Binärzahlen absteigend sortiert sind. Dabei nehmen wir an, dass jeweils in der ersten Zeile das höchstwertigste Bit und in der letzten Zeile das niederwertigste Bit steht. Wir machen uns dies noch einmal anhand unserer Beispiel aus der Einleitung klar und betrachten dazu Abbildung 7.38. Der besseren Übersichtlichkeit wegen, bezeichnen wir die Spalten jetzt mit a–g anstatt mit 1–7.



Fall 1: $e_i = e_j$ Fall 2: $e_j < e_i$

Fall 3: $e_i > e_j$ Fall 4: $e_i \not\sim e_j$

Abbildung 7.37: Skizze: Phylogenetischer Baum für M

M	a	b	c	d	e	f	g
1	1	0	1	0	0	1	0
2	1	0	1	0	1	0	1
3	1	0	1	0	1	0	0
4	1	0	0	0	0	0	0
5	0	1	1	0	0	0	0
6	0	1	0	0	0	0	0

M'	a	c	f	e	g	b	d
1	1	1	1	0	0	0	0
2	1	1	0	1	1	0	0
3	1	1	0	1	0	0	0
4	1	0	0	0	0	0	0
5	0	0	0	0	0	1	1
6	0	0	0	0	0	1	0

Abbildung 7.38: Beispiel: Sortierte Charaktermatrix

Wir betrachten jetzt zwei beliebige Objekte p und q. Sei k der größte gemeinsame Charakter, den sowohl p als auch q besitzt, d.h.

$$k = \max\{j : M(p,j) = M(q,j) = 1\}.$$

Wir betrachten jetzt einen Charakter $i < k$ von p, d.h. $M(p,i) = 1$. Somit gilt $p \in O_i \cap O_k$ und nach Voraussetzung gilt entweder $O_i \subset O_k$ oder $O_k \subset O_i$. Da die Spalten absteigend sortiert sind, muss die größere Zahl (also die zur Spalte i korrespondierende) mehr 1-Bits besitzen und es gilt somit $O_k \subset O_i$. Da $q \in O_k$ gilt, ist nun auch $q \in O_i$. Analoges gilt für ein $i < k$ mit $M(q, i) = 1$. Damit gilt für alle Charaktere $i \leq k$ (nach der Spaltensortierung), dass entweder beide den Charakter i besitzen oder keiner. Da k der größte Index ist, so dass beide einen Charakter gemeinsam besitzen, gilt für $i > k$, dass aus $M(p, i) = M(q, i)$ folgt, dass beide den Charakter i nicht besitzen, d.h. $M(p, i) = M(q, i) = 0$.

Betrachten wir also zwei Objekte (also zwei Zeilen in der spaltensortierten Charaktermatrix), dann besitzen zu Beginn entweder beide einen Charakter gemeinsam oder keinen. Sobald wir einen Charakter finden, der beide Objekte unterscheidet, so finden wir später keine gemeinsamen Charaktere mehr.

Mit Hilfe dieser Eigenschaft können wir jetzt einen phylogenetischen Baum konstruieren. Dazu betrachten wir die Abbildung $p \mapsto s_{p,1} \cdots s_{p,\ell}\$$, wobei wir jedem Objekt eine Zeichenkette über $[1 : m]$ zuordnen, so dass jedes Symbol aus $[1 : m]$ maximal einmal auftaucht und ein $i \in [1 : m]$ nur dann auftaucht, wenn p den entsprechenden Charakter besitzt, also wenn $M(p, i) = 1$. Die Reihenfolge ergibt sich dabei aus der Spaltensortierung der Charaktermatrix.

Für unser Beispiel ist diese Zuordnung in der Abbildung 7.39 angegeben.

$$\begin{aligned}
s_1 &= acf\$ \\
s_2 &= aceg\$ \\
s_3 &= ace\$ \\
s_4 &= a\$ \\
s_5 &= bd\$ \\
s_6 &= b\$
\end{aligned}$$

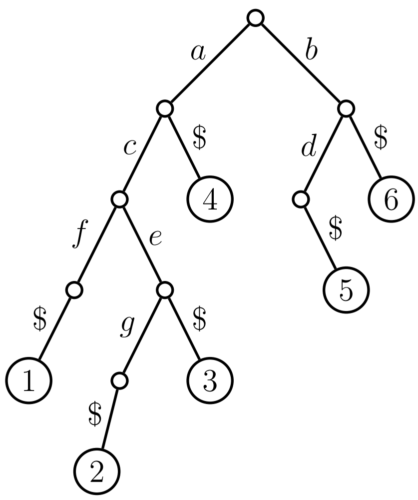


Abbildung 7.39: Skizze: Trie für die Zeichenreihen der Charaktermatrix

Wenn wir jetzt für diese Zeichenreihen einen Trie konstruieren (dies entspricht dem Suchmuster-Baum aus dem Aho-Corasick-Algorithmus), so erhalten wir den phylogenetischen Baum für unsere Charaktermatrix M. Wir müssen nur noch Knoten mit nur einem Kind kontrahieren und die Kantenmarkierungen mit dem \$ entfernen.

Aus der Konstruktion folgt, dass alle Blätter die benötigten Kantenmarkierungen auf dem Pfad von der Wurzel zum Blatt besitzen. Nach der Spaltensortierung und der daran anschließenden Diskussion kommt auch jeder Charakter nur einmal als Kantenmarkierung vor. Somit haben wir nachgewiesen, dass der konstruierte Baum ein phylogenetischer Baum ist und der Beweis ist beendet. ■

Aus dem vorherigen Beweis erhalten wir unmittelbar einen Algorithmus zur Berechnung einer perfekten binären Phylogenie, der in Abbildung 7.40 aufgelistet ist.

1. Sortieren der Spalten der binären Charaktermatrix. $O(nm)$
2. Konstruktion der Zeichenreihen $s_1, \ldots, s_n$. $O(nm)$
3. Konstruktion des Tries T für $s_1, \ldots, s_n$. $O(nm)$
4. Kompaktifiziere den Trie T und teste, ob der kompaktifizierte Trie ein phylogenetischer Baum ist. $O(nm)$

Abbildung 7.40: Algorithmus: Konstruktion einer perfekten binäre Phylogenie

Anstatt die Bedingung aus dem Lemma zu überprüfen, konstruieren wir einfach einen kompaktifizierten Trie. Ist dieser ein phylogenetischen Baum, so muss dieser der gesuchte sein. Andernfalls gibt es keinen phylogenetischen Baum für die gegebene binäre Charaktermatrix.

Theorem 7.32 *Für eine binäre $n \times m$-Charaktermatrix M lässt sich in Zeit $O(nm)$ entscheiden, ob sie eine perfekte Phylogenie besitzt. Falls eine perfekte Phylogenie existiert, lässt sich der zugehörige phylogenetische Baum in Zeit $O(nm)$ konstruieren.*

7.4.2 Binäre Phylogenien und Ultrametriken

Nun wollen wir noch zeigen, dass sich das Problem der perfekten Phylogenie auch auf das Problem der Bestimmung eines ultrametrischen Baumes zurückführen lässt. Zunächst definieren wir den Begriff einer phylogenetischen Distanzmatrix

Definition 7.33 *Sei M eine binäre $n \times m$-Charaktermatrix, dann heißt die $n \times n$-Matrix eine* phylogenetische Distanzmatrix, *wenn*

$$D_M(i,j) = m - |\{c \,:\, M(i,c) = M(j,c) = 1\}|.$$

In Abbildung 7.41 ist ein Beispiel für eine binäre Charaktermatrix und der zugehörigen phylogenetischen Distanzmatrix angegeben.

M	1	2	3	4	5	6	7
1	1	0	1	0	0	1	0
2	1	0	1	0	1	0	1
3	1	0	1	0	1	0	0
4	1	0	0	0	0	0	0
5	0	1	1	0	0	0	0
6	0	1	0	0	0	0	0

D_M	1	2	3	4	5	6
1	0	5	5	6	7	7
2		0	4	6	7	7
3			0	6	7	7
4				0	7	7
5					0	6
6						0

Abbildung 7.41: Beispiel: phylogenetische Distanzmatrix einer binären Charaktermatrix

Nun kommen wir zu der gesuchten Charakterisierung.

Lemma 7.34 *Besitzt eine binäre Charaktermatrix eine perfekte Phylogenie, dann ist die zugehörige phylogenetische Distanzmatrix ultrametrisch.*

Beweis: Sei T ein phylogenetischer Baum für eine binäre Charaktermatrix M. Wir konstruieren jetzt daraus einen ultrametrischen Baum T', indem wir zu T Knotenmarkierungen hinzufügen. Markiere die Wurzel mit n. Betrachte nun die Kinder rekursiv. Ist dabei die Kante zu einem Kind mit $k \geq 0$ Charakteren markiert, so besitzen diese alle gemeinsam diesen Charakter im betrachteten Teilbaum und der Maximal-Abstand kann nur noch $m-k$ sein. Mit Hilfe einer Tiefensuche können wir so den Baum durchlaufen und die Knotenmarkierungen vergeben. ■

Aus diesem Beweis ergibt sich der folgende, in Abbildung 7.42 angegebene Algorithmus. Wir schauen uns jetzt nur noch die Details von Schritt 3 genauer an. Alle vier Schritte können jeweils in $O(nm)$ ausgeführt werden. Es ergibt sich somit eine Gesamtlaufzeit von $O(mn^2)$.

Falls M eine perfekte Phylogenie besitzt, so kann diese mit den angegebenen Schritten konstruiert werden. Durch die angegebene Methode zur Konstruktion von D_M liegen alle Werte d_{ij} im Bereich $[0 : m]$. Es wird genau ein Blatt pro Spezies angelegt und im Schritt 3c) wird jeder Charakter eines Blattes zu einer Kante von der Wurzel zum Blatt zugewiesen (ohne Beschränkung der Allgemeinheit sei der Fall ausgeschlossen, dass alle Spezies eine bestimmte Eigenschaft haben). In Schritt 3d) wird dann, dass jeder Charakter nur einmal auftritt. Falls dies gelingt, so ist der entstandene Baum eine perfekte Phylogenie für M.

1. Konstruiere D_M aus M. $O(nm)$
2. Konstruiere den ultrametrischen Baum U für D_M. Lässt sich U nicht konstruieren, dann besitzt M keine perfekte Phylogenie. $O(n^2)$
3. Versuche die Kanten des ultrametrischen Baumes U mit binären Charaktere zu markieren. Lässt sich dies nicht bewerkstelligen, dann besitzt M keine perfekte Phylogenie. $O(n^2m)$
   a) Jedem Blatt (entspricht einer Spezies) wird sein Zeilenvektor aus der gegebenen Matrix M zugewiesen.
   b) Jedem inneren Knoten wird ein $\{0, 1\}$-Vektor der Länge m zugewiesen, der aus der komponentenweisen Multiplikation der Vektoren seiner Kinder entsteht.
   c) Jeder Kante werden die Charaktere zugewiesen, die im Kind auftauchen, aber im Elter nicht.
   d) Wir jeder Buchstabe nur einmal verwendet, entsteht so eine perfekte Phylogenie.

Abbildung 7.42: Algorithmus: Konstruktion eine perfekten binären Phylogenie

Falls M eine perfekte Phylogenie hat, bleibt zu zeigen, dass jeder Charakter nur genau einmal vorkommt (er muss mindestens einmal vorkommen). Die technischen Details überlassen wir dem Leser als Übungsaufgabe und fassen das Ergebnis zusammen.

Theorem 7.35 *Für eine binäre $n \times m$-Charaktermatrix M lässt sich in Zeit $O(mn^2)$ entscheiden, ob sie eine perfekte Phylogenie besitzt. Falls eine perfekte Phylogenie existiert, lässt sich der zugehörige phylogenetische Baum in Zeit $O(mn^2)$ konstruieren.*

7.5 Sandwich Probleme

Hauptproblem bei den bisher vorgestellten Verfahren war, dass wir dazu die Distanzen genau wissen mussten, da beispielsweise eine leicht modifizierte ultrametrische Matrix in der Regel nicht mehr ultrametrisch ist. Aus diesem Grund werden wir in diesem Abschnitt einige Problemestellungen vorstellen und lösen, die Fehler in den Eingabedaten modellieren.

7.5.1 Fehlertolerante Modellierungen

Bevor wir zur Problemformulierung kommen, benötigen wir noch einige Notationen, wie wir Matrizen zwischen zwei anderen Matrizen einschachteln können.

Notation 7.36 *Seien M und M' zwei $\times n$-Matrizen, dann gilt $M \leq M'$, wenn $M_{i,j} \leq M'_{i,j}$ für alle $i, j \in [1 : n]$ gilt. Für drei Matrizen M, M' und M'' gilt $M \in [M', M'']$, wenn $M' \leq M \leq M''$ gilt.*

Nun können wir die zu untersuchenden Sandwich-Probleme angeben.

ADDITIVES SANDWICH PROBLEM

Eingabe: Zwei $n \times n$-Distanzmatrizen D_ℓ und D_h.
Ausgabe: Eine additive Distanzmatrix $D \in [D_\ell, D_h]$, sofern eine existiert.

ULTRAMETRISCHES SANDWICH PROBLEM

Eingabe: Zwei $n \times n$-Distanzmatrizen D_ℓ und D_h.
Ausgabe: Eine ultrametrische Distanzmatrix $D \in [D_\ell, D_h]$, sofern eine existiert.

Im Folgenden bezeichnet $\|M\|$ eine Norm einer Matrix. Hierbei wird diese Norm jedoch nicht eine Abbildungsnorm der durch M induzierten Abbildung sein, sondern wir werden Normen verwenden, die eine $n \times n$-Matrix als einen Vektor mit n^2 Einträgen interpretiert. Beispielsweise können wir die so genannten p-Normen verwenden:

$$\begin{aligned} \|D\|_p &= \left(\sum_{i=1}^{n} \sum_{j=1}^{n} |M_{i,j}|^p \right)^{1/p}, \\ \|D\|_\infty &= \max \{ |M_{i,j}| \ : \ i, j \in [1 : n] \}. \end{aligned}$$

ADDITIVES APPROXIMATIONSPROBLEM

Eingabe: Eine $n \times n$-Distanzmatrix D.
Ausgabe: Eine additive Distanzmatrix D', die $\|D - D'\|$ minimiert.

Ultrametrisches Approximationsproblem

Eingabe: Eine $n \times n$-Distanzmatrix D.
Ausgabe: Eine ultrametrische Distanzmatrix D', die $\|D - D'\|$ minimiert.

Für die weiteren Untersuchungen benötigen wir noch einige Notationen.

Notation 7.37 *Sei M eine $n \times n$-Matrix, dann ist $\mathrm{MAX}(M)$ der maximale Eintrag von M, d.h. $\mathrm{MAX}(M) = \max\{M_{i,j} : i, j \in [1:n]\}$.*

Sei T ein additiver Baum mit n markierten Knoten (mit Markierungen aus $[1:n]$) und $d_T(i,j)$ der durch den additiven Baum induzierte Abstand zwischen den Knoten mit Markierung i und j, dann ist $\mathrm{MAX}(T) = \max\{d_T(i,j) : i, j \in [1:n]\}$.

Sei M eine $n \times n$-Matrix und T ein additiver Baum mit n markierten Knoten, dann schreiben wir $T \leq M$ bzw. $T \geq M$, wenn $d_T(i,j) \leq M_{i,j}$ bzw. $d_T(i,j) \geq M_{i,j}$ für alle $i, j \in [1:n]$ gilt.

Für zwei Matrizen M und M' sowie einen additiven Baum T gilt $T \in [M, M']$, wenn $M \leq T \leq M'$ gilt.

Wir wollen noch einmal kurz daran erinnern, wie man aus einem ultrametrischen Baum $T = (V, E, \mu)$ mit der Knotenmarkierung $\mu : V \to \mathbb{R}_+$ einen additiven Baum $T = (V, E, \gamma)$ mit der Kantengewichtsfunktion $\gamma : E \to \mathbb{R}_+$ erhält, indem man γ wie folgt definiert:

$$\forall (v,w) \in E : \gamma(v,w) := \frac{\mu(v) - \mu(w)}{2} > 0.$$

Somit können wir ultrametrische Bäume immer auch als additive Bäume auffassen.

7.5.2 Eine einfache Lösung

In diesem Abschnitt wollen wir zuerst eine einfache Lösung angeben, um die Ideen hinter der Lösung zu erkennen. Im nächsten Abschnitt werden wir dann eine effizientere Lösung angeben, die von der Idee her im Wesentlichen gleich sein wird, aber dort nicht so leicht bzw. kaum zu erkennen sein wird.

Zuerst zeigen wir, dass wenn es einen ultrametrischen Baum gibt, der der Sandwich-Bedingung genügt, es auch einen ultrametrischen Baum gibt, dessen Wurzelmarkierung gleich dem maximalem Eintrag der Matrix D_ℓ ist. Dies liefert einen ersten Ansatzpunkt für einen rekursiven Algorithmus.

Lemma 7.38 *Seien D_ℓ und D_h zwei $n \times n$-Distanzmatrizen. Wenn ein ultrametrischer Baum T mit $T \in [D_\ell, D_h]$ existiert, dann gibt es einen ultrametrischen Baum T' mit $T' \in [D_\ell, D_h]$ und $\mathrm{MAX}(T') = \mathrm{MAX}(D_\ell)$.*

Beweis: Wir beweisen die Behauptung durch Widerspruch. Wir nehmen also an, dass es keinen ultrametrischen Baum $T' \in [D_\ell, D_h]$ mit $\mathrm{MAX}(T') = \mathrm{MAX}(D_\ell)$ gibt.

Sei $T' \in [D_\ell, D_h]$ ein ultrametrischer Baum, der $s := \mathrm{MAX}(T') - \mathrm{MAX}(D_\ell)$ minimiert. Nach Voraussetzung gibt es mindestens einen solchen Baum und nach unserer Annahme für den Widerspruchsbeweis ist $s > 0$.

Wir konstruieren jetzt aus T' einen neuen Baum T'', indem wir nur die Kantengewichte der Kanten in T'' ändern, die zur Wurzel von T' bzw. T'' inzident sind. Zuerst definieren wir $\alpha := \frac{1}{2}\min\{\gamma(e), s\} > 0$. Wir bemerken, dass dann $\alpha \leq \frac{\gamma(e)}{2}$ und $\alpha \leq \frac{s}{2}$ gilt.

Sei also e ein Kante, die zur Wurzel von T'' inzident ist. Dann setzen wir

$$\gamma''(e) = \gamma'(e) - \alpha.$$

Hierbei bezeichnet γ' bzw. γ'' die Kantengewichtsfunktion von T' bzw. T''. Zuerst halten wir fest, dass die Kantengewichte der Kanten, die zur Wurzel inzident sind, weiterhin positiv sind, da

$$\gamma(e) - \alpha \;\; \geq \;\; \gamma(e) - \frac{\gamma(e)}{2} \;\; = \;\; \frac{\gamma(e)}{2} \;\; > \;\; 0.$$

Da wir Kantengewichte nur reduzieren, gilt offensichtlich weiterhin $T'' \leq D_h$. Wir müssen also nur noch zeigen, dass auch $D_\ell \leq T''$ weiterhin gilt. Betrachten wir hierzu zwei Blätter v und w in T'' und die Wurzel $r(T'')$ von T''. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob der kürzeste Weg von v nach w über die Wurzel führt oder nicht.

Fall 1 ($\mathbf{lca}(v, w) \neq r(T'')$): Dann wird der Abstand von $d_{T''}$ gegenüber $d_{T''}$ für diese Blätter nicht verändert und es gilt:

$$d_{T''}(v, w) = d_{T'}(v, w) \geq D_\ell(v, w).$$

Fall 2 ($\mathbf{lca}(v, w) = r(T'')$): Dann werden die beiden Kantengewichte der Kanten auf dem Weg von v zu w die inzident zur Wurzel sind um jeweils α erniedrigt und wir erhalten:

$$\begin{aligned} d_{T''}(v,w) &= d_{T'}(v,w) - 2\alpha \\ &\quad \text{da } d_{T'}(v,w) = s + \mathrm{MAX}(D_\ell) \\ &= s + \mathrm{MAX}(D_\ell) - 2\alpha \\ &\quad \text{da } 2\alpha \leq s \\ &\geq s + \mathrm{MAX}(D_\ell) - s \\ &= \mathrm{MAX}(D_\ell) \\ &\geq D_\ell(v,w). \end{aligned}$$

Also ist $D_\ell \leq T''$. Somit haben wir einen ultrametrischen Baum $T'' \in [D_\ell, D_h]$ konstruiert, für den

$$\begin{aligned} \mathrm{MAX}(T'') - \mathrm{MAX}(D_\ell) &\leq \mathrm{MAX}(T') - 2\alpha - \mathrm{MAX}(D_\ell) \\ &\quad \text{da } \alpha > 0 \\ &< \mathrm{MAX}(T') - \mathrm{MAX}(D_\ell) \\ &= s. \end{aligned}$$

gilt. Dies ist offensichtlich ein Widerspruch zur Wahl von T' und das Lemma ist bewiesen. ■

Das vorherige Lemma legt die Definition niedriger ultrametrische Bäume nahe.

Definition 7.39 *Ein ultrametrischer Baum $T \in [D_\ell, D_h]$ heißt* niedrig, *wenn* $\mathrm{MAX}(T) = \mathrm{MAX}(D_\ell)$.

Um uns im weiteren etwas leichter zu tun, benötigen wir einige weitere Notationen.

Notation 7.40 *Sei $T = (V, E)$ ein gewurzelter Baum. Dann bezeichnet $\mathcal{L}(T)$ die Menge der Blätter von T. Für $v \in \mathcal{L}(T)$ bezeichnet*

$$\mathcal{L}(T, v) := \{w \in \mathcal{L}(T) \;:\; \mathrm{lca}(v,w) \neq r(T)\}$$

die Menge aller Blätter die sich im selben Teilbaum der Wurzel von T befinden wie v selbst.

Aus dem vorherigen Lemma und den Notationen folgt jetzt unmittelbar das folgende Korollar.

Korollar 7.41 *Für jeden niedrigen ultrametrischen Baum $T \in [D_\ell, D_h]$ gilt:*

$$\forall x, y \in \mathcal{L}(T) : (D_\ell(x, y) = \mathrm{MAX}(D_\ell)) \Rightarrow (d_T(x, y) = \mathrm{MAX}(D_\ell)).$$

Bevor wir zum zentralen Lemma kommen, müssen wir noch eine weitere grundlegende Definition festlegen.

Definition 7.42 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Matrizen und seien $k, \ell \in [1 : n]$, dann ist der Graph $G_{k,\ell} = (V, E_{k,\ell})$ wie folgt definiert:*

$$\begin{aligned} V &:= [1 : n], \\ E_{k,l} &:= \{(i, j) \; : \; D_h(i, j) < D_\ell(k, \ell)\}\,. \end{aligned}$$

Mit $C(G_{k,\ell}, v)$ bezeichnen wir die Zusammenhangskomponente von $G_{k,\ell}$, die den Knoten v enthält.

Nun kommen wir zu dem zentralen Lemma für unseren einfachen Algorithmus, das uns beschreibt, wie wir mit Kenntnis des Graphen $G_{k\ell}$ (oder besser dessen Zusammenhangskomponenten) den gewünschte ultrametrischen Baum konstruieren können.

Lemma 7.43 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Matrizen und seien $k, \ell \in [1 : n]$ so gewählt, dass $D_\ell(k, \ell) = \mathrm{MAX}(D_\ell)$. Wenn ein niedriger ultrametrischer Baum $T \in [D_\ell, D_h]$ existiert, dann gibt es auch einen niedrigen ultrametrischen Baum $T' \in [D_\ell, D_h]$ mit*

$$\mathcal{L}(T', v) = V(C(G_{k,\ell}, v))$$

für alle $v \in \mathcal{L}(T)$.

Beweis: Wir beweisen zuerst die folgende Behauptung:

$$\forall T \in [D_\ell, D_h] : V(C(G_{k,l}, v)) \subseteq \mathcal{L}(T, v).$$

Wir führen diesen Beweis durch Widerspruch. Sei dazu $x \in V(C(G_{k,\ell}, v)) \setminus \mathcal{L}(T, v)$. Da $x \in V(C(G_{k,\ell}, v))$ gibt es einen Pfad p von v nach x in $G_{k\ell}$ (siehe dazu auch Abbildung 7.43). Sei $(y, z) \in p$ die erste Kante des Pfades von v nach x in $G_{k\ell}$, so dass $y \in \mathcal{L}(t, v)$, aber $z \notin \mathcal{L}(T, v)$ gilt. Da $(y, z) \in E(C(G_{k,\ell}, v)) \subseteq E_{k,\ell}$ ist, gilt $D_h(y, z) < D_\ell(k, \ell)$.

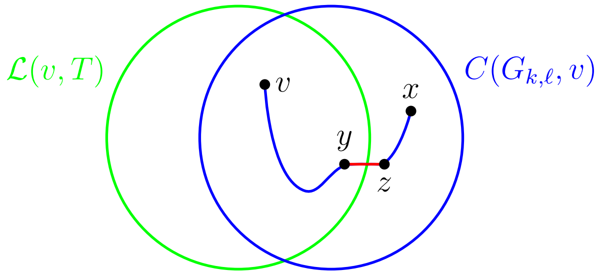


Abbildung 7.43: Skizze: $\mathcal{L}(T,v)$ und $C(G_{k,\ell},v)$

Da $y \in \mathcal{L}(T,v)$, aber $z \notin \mathcal{L}(T,v)$ ist, gilt $\mathrm{lca}(y,z) = r(T)$. Somit ist

$$d_T(y,z) \geq \mathrm{MAX}(D_\ell) = D_\ell(k,\ell).$$

Daraus folgt unmittelbar, dass

$$d_T(y,z) \geq D_\ell(k,\ell) > D_h(y,z).$$

Dies ist aber offensichtlich ein Widerspruch zu $T \in [D_\ell, D_h]$ und somit ist die Behauptung gezeigt.

Wir zeigen jetzt, wie ein Baum T umgebaut werden kann, so dass er die gewünschte Eigenschaft des Lemmas erfüllt. Sei also T ein niedriger ultrametrischer Baum mit $T \in [D_\ell, D_h]$. Sei weiter $S := \mathcal{L}(T,v) \setminus V(C(G_{k,\ell},v))$. Ist S leer, so ist nichts mehr zu zeigen. Sei also $s \in S$ und $x \in V(C(G_{k,\ell},v))$. Nach Wahl von s und x gibt es in $G_{k,\ell}$ keinen Pfad von s nach x. Somit gilt

$$D_h(x,s) \geq D_\ell(k,\ell) = \mathrm{MAX}(D_\ell) = \mathrm{MAX}(T) \geq d_T(x,s) \geq D_\ell(x,s).$$

Wir bauen jetzt T zu T' wie folgt um. Betrachte den Teilbaum T_1 der Wurzel $r(T)$ von T der sowohl x als auch s enthält. Wir duplizieren T_1 zu T_2 und hängen an die Wurzel von T'' sowohl T_1 als auch T_2 an. In T_1 entfernen wir alle Blätter aus S und in T_2 entfernen wir alle Blätter aus $V(C(G_{k,\ell},v))$ (siehe auch Abbildung 7.44). Anschließend räumen wir in den Bäumen noch auf, indem wir Kanten entfernen die zu keinen echten Blättern mehr führen. Ebenso löschen wir Knoten, die nur noch ein Kind besitzen, indem wir dieses Kind zum Kind des Elters des gelöschten Knoten machen. Letztendlich erhalten wir einen neuen ultrametrischen Baum T''.

Wir zeigen jetzt, dass $T'' \in [D_\ell, D_h]$ ist. Da wir die Knotenmarkierung der überlebenden Knoten nicht ändern, bleibt der ultrametrische Baum niedrig. Betrachten wir zwei Blätter x und y , die nicht zu T_1 oder T_2 gehören. Da der Pfad in T'' derselbe

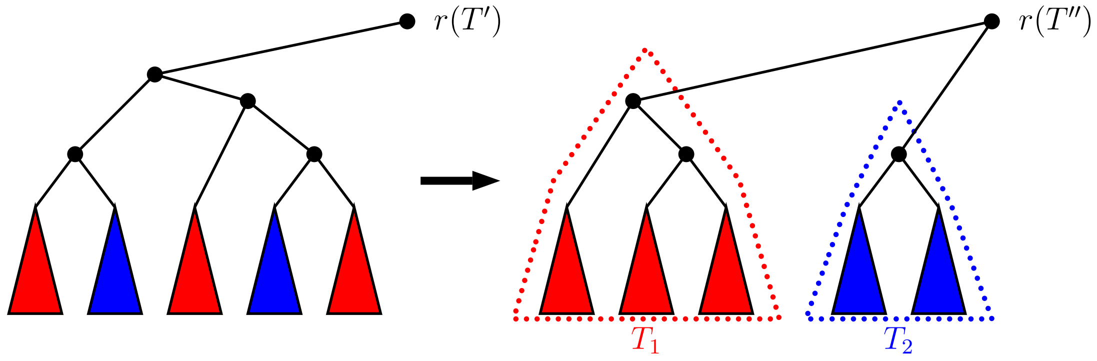


Abbildung 7.44: Skizze: Umbau des niedrigen ultrametrischen Baumes

wir T' ist, gilt weiterhin

$$D_\ell(x,y) \leq d_{T''}(x,y) \leq D_h(x,y).$$

Gehört ein Knoten x zu T_1 oder T_2 und der andere Knoten y nicht zu T_1 und T_2, so ist der Pfad nach die Duplikation bezüglich des Abstands derselbe. Es gilt also wiederum

$$D_\ell(x,y) \leq d_{T''}(x,y) \leq D_h(x,y).$$

Gehören beide Knoten v und w entweder zu T_1 oder zu T_2, dann hat sich der Pfad nach der Duplikation bzgl. des Abstands auch wieder nicht geändert, also gilt

$$D_\ell(x,y) \leq d_{T''}(x,y) \leq D_h(x,y).$$

Es bleibt der Fall zu betrachten, dass ein Knoten zu T_1 und einer zu T_2 gehört. Hier hat sich der Pfad definitiv geändert, da er jetzt über die Wurzel von T'' führt. Sei s der Knoten in T_1 und x der Knoten in T_2. Wir haben aber bereits gezeigt, dass für solche Knoten gilt:

$$D_h(x,s) \geq d_{T''}(x,s) \geq D_\ell(x,s).$$

Somit ist der Satz bewiesen. ■

Aus diesem Beweis ergibt sich unmittelbar die folgende Idee für unseren Algorithmus. Aus der Kenntnis des Graphen $G_{k\ell}$ für ein größte untere Schranke $D_\ell(k,\ell)$ für zwei Spezies k und ℓ beschreiben uns die Zusammenhangskomponenten die Partition der Spezies, wie diese in den verschiedenen Teilbäumen an der Wurzel des ultrametrischen Baumes hängen müssen, sofern es überhaupt einen gibt. Somit können wir die Spezies partitionieren und für jede Menge der Partition einen eigenen ultrametrischen Baum rekursiv konstruieren, deren Wurzeln dann die Kinder der Gesamtwurzel des zu konstruierenden ultrametrischen Teilbaumes werden. Damit ergibt sich der folgende, in Abbildung 7.45 angegebene Algorithmus.

- Bestimme $k, \ell \in [1:n]$, so dass $D_\ell(k,\ell) = \mathrm{MAX}(D_\ell)$. $O(n^2)$
- Konstruiere $G_{k,\ell}$. $O(n^2)$
- Bestimme die Zusammenhangskomponenten $C_1, \ldots, C_m$ von $G_{k,\ell}$. $O(n^2)$
- Konstruiere rekursiv ultrametrische Bäume für die einzelnen Zusammenhangskomponenten. $\sum_{i=1}^{m} T(|C_i|)$
- Baue aus den Teillösungen $T_1, \ldots, T_m$ für $C_1, \ldots, C_m$ einen ultrametrischen Baum, indem man die Wurzeln der $T_1, \ldots, T_m$ als Kinder an eine neue Wurzel hängt, die als Knotenmarkierung $\mathrm{MAX}(D_\ell)$ erhält. $O(n)$

Abbildung 7.45: Algorithmus: Algorithmus für das ultrametrische Sandwich-Problem

Für die Korrektheit müssen wir nur noch zeigen, dass die Partition nicht trivial ist, d.h., dass der Graph $G_{k,\ell}$ nicht zusammenhängend ist.

Lemma 7.44 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen und seien $k, \ell \in [1:n]$ so gewählt, dass $D_\ell(k,\ell) = \mathrm{MAX}(D_\ell)$ gilt. Wenn $G_{k,\ell}$ zusammenhängend ist, dann kann es keine ultrametrische Matrix $U \in [D_\ell, D_h]$ geben.*

Beweis: Wir führen den Beweis durch Widerspruch. Angenommen $G_{k,\ell}$ ist zusammenhängend und es existiert eine ultrametrische Matrix U mit zugehörigem Baum T. Sei $D_\ell(k,\ell) = \mathrm{MAX}(D_\ell)$. Wir wissen, dass $U(k,\ell) \geq D_\ell(k,\ell)$, weil $U \in [D_\ell, D_h]$. Weiterhin ist der kleinste gemeinsame Vorfahre $v = \mathrm{lca}(k,\ell)$ von k und ℓ mit $U(k,\ell)$ markiert.

Sei r_k bzw. r_ℓ das Kind von v, deren Teilbaum k bzw. ℓ enthält. Für alle Blätter i, j aus den Teilbäumen T_{r_k}, T_{r_ℓ} mit $i \in T_{r_k}$ und $j \in T_{r_\ell}$ gilt: $U(i,j) = U(k,\ell)$. Außerdem sind die Blattmengen $\mathcal{L}(T_{r_k})$ und$\mathcal{L}(T_{r_\ell})$ disjunkt.

Seien C_k die von k in $G_{k,\ell}$ erreichbaren Knoten. Aus der Definition folgt, dass

$$\forall i, j \in C_k : \; U(i,j) \leq D_h(i,j) < \mathrm{MAX}(D_\ell) \leq U(k,\ell).$$

Aus $U(i,k) < U(k,\ell)$ folgt, dass die Blätter i und k einen niedrigsten gemeinsamen Vorfahren haben, deren Markierung größer ist als die von v ist. Daher gilt $i \in T_{r_k}$.

Unter der Annahme, dass $G_{k,\ell}$ zusammenhängend ist, gilt aber $\ell \in C_k$. Daraus würde auch $\ell \in T_{r_k}$ und letztlich $v = r_k$. Dies liefert den gewünschten Widerspruch ■

Damit ist die Korrektheit bewiesen. In Abbildung 7.46 ist ein Beispiel zur Illustration der Vorgehensweise des einfachen Algorithmus angegeben.

Für die Laufzeit erhalten wir

$$T(n) = O(n^2) + \sum_{i=1}^{m} T(n_i)$$

mit $\sum_{i=1}^{m} n_i = n$. Wir überlegen uns, was bei einem rekursiven Aufruf geschieht. Bei jedem rekursiven Aufruf wird eine Partition der Menge $[1:n]$ verfeinert. Da wir mit der trivialen Partition $\{[1:n]\}$ starten und mit $\{\{1\},\ldots,\{n\}\}$ enden, kann es also maximal $n-1$ rekursive Aufrufe geben. Somit ist die Laufzeit durch $O(n \cdot n^2)$ beschränkt.

Theorem 7.45 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen. Ein ultrametrischer Baum $T \in [D_\ell : D_h]$ kann in Zeit $O(n^3)$ konstruiert werden, sofern überhaupt einer existiert.*

7.5.3 Charakterisierung einer effizienteren Lösung

In diesem Abschnitt wollen wir zeigen, dass wir das ultrametrische Sandwich Problem sogar in Zeit $O(n^2)$ lösen können. Dies ist optimal, da ja bereits die Eingabematrizen die Größe $\Theta(n^2)$ besitzen. Dazu benötigen wir erst noch die Definition der Kosten eines Pfades.

Definition 7.46 *Sei $G = (V, E, \gamma)$ ein gewichteter Graph. Die* Kosten $c(p)$ *eines Pfades $p = (v_0, \ldots, v_n)$ ist definiert durch*

$$c(p) := \max\left\{\gamma(v_{i-1}, v_i) \,:\, i \in [1:n]\right\}.$$

Mit $D(G, v, w)$ bezeichnen wir die minimalen Kosten eines Pfades von v nach w in G.

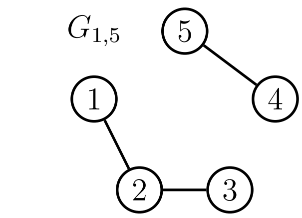
$$D(G, v, w) := \min\left\{c(p) \,:\, p \text{ ist ein Pfad von } v \text{ nach } w\right\}.$$

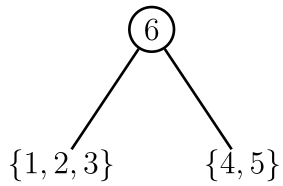
Warum interessieren wir uns überhaupt für die Kosten eines Pfades? Betrachten wir einen Pfad $p = (v_0, \ldots, v_n)$ in einem gewichteten Graphen $G(D)$, der von einer ultrametrischen Matrix D induziert wird. Seien dabei $\gamma(v, w)$ die Kosten der Kante (v, w). Wie groß ist nun der Abstand $d_D(v_0, v_n)$ von v_0 zu v_n? Unter Berücksichtigung der ultrametrischen Dreiecksungleichung erhalten wir:

$$d_D(v_0, w_n) \;\leq\; \max\{d_D(v_0, v_{n-1}), \gamma(v_{n-1}, v_n)\}$$

D_ℓ	1	2	3	4	5
1	0	1	2	3	6
2		0	4	5	5
3			0	4	5
4				0	1
5					0

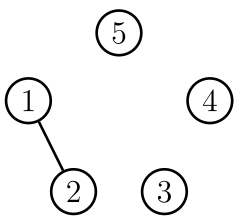
D_h	1	2	3	4	5
1	0	3	6	8	8
2		0	5	6	8
3			0	6	8
4				0	3
5					0

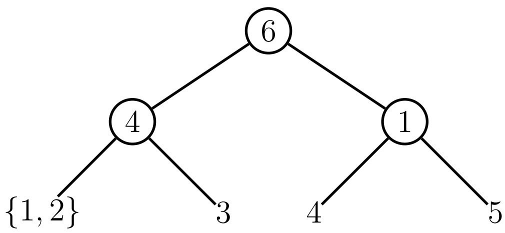




D_ℓ	1	2	3	4	5
1	0	1	2	3	6
2		0	4	5	5
3			0	4	5
4				0	1
5					0

D_h	1	2	3	4	5
1	0	3	6	8	8
2		0	5	6	8
3			0	6	8
4				0	3
5					0





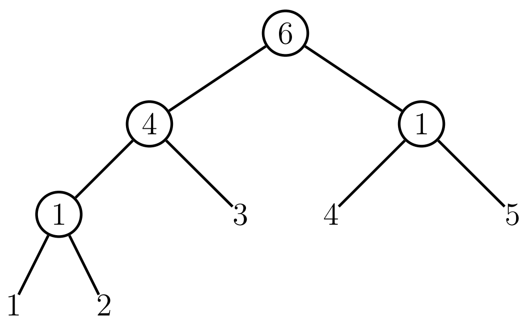


Abbildung 7.46: Beispiel: Einfache Lösung des ultrametrischen Sandwich-Problems

$$
\begin{aligned}
&\leq \max\{\max\{d_D(v_0, v_{n-2}), \gamma(v_{n-2}, v_{n-1})\}, \gamma(v_{n-1}, v_n)\} \\
&= \max\{d_D(v_0, v_{n-2}), \gamma(v_{n-2}, v_{n-1}), \gamma(v_{n-1}, v_n)\} \\
&\vdots \\
&\leq \max\{\gamma(v_0, v_1), \ldots, \gamma(v_{n-2}, v_{n-1}), \gamma(v_{n-1}, v_n)\} \\
&= c(p)
\end{aligned}
$$

Die letzte Gleichung folgt nur für den Fall, dass wir einen Pfad mit minimalen Kosten gewählt haben. Somit sind die Kosten eines Pfades in der zur ultrametrischen Matrix gehörigem gewichteten Graphen eine obere Schranke für den Abstand der Endpunkte dieses Pfades.

Im folgenden Lemma erhalten wir dann eine weitere Charakterisierung, wann zwei Knoten k und ℓ im zugehörigen Graphen $G_{k\ell}$ durch einen Pfad verbunden sind. Man beachte, dass wir hier nicht beliebige Knotenpaare betrachten, sondern genau das Knotenpaar, dessen maximaler Abstand gemäß der unteren Schranke den Graphen $G_{k\ell}$ definiert.

Lemma 7.47 *Seien $D_\ell \leq D_h$ zwei Distanzmatrizen. Zwei Knoten k und ℓ befinden sich genau dann in derselben Zusammenhangskomponente von $G_{k,\ell}$, wenn $D_\ell(k, \ell) > D(G(D_h), k, \ell)$.*

Beweis: **$\Rightarrow$:** Wenn sich k und ℓ in derselben Zusammenhangskomponente von $G_{k,\ell}$ befinden, dann gibt es einen Pfad p von k nach ℓ. Für alle Kanten $(v, w) \in p$ gilt daher (da sie Kanten in $G_{k,\ell}$ sind): $\gamma(v, w) < D_\ell(k, l)$. Somit ist auch das Maximum der Kantengewichte durch $D_\ell(k, \ell)$ beschränkt und es gilt $D(G(D_h), k, \ell) < D_\ell(k, \ell)$.

$\Leftarrow$: Gelte nun $D(G(D_h), k, \ell) < D_\ell(k, \ell)$. Dann existiert nach Definition von $D(\cdot, \cdot)$ und $G_{k,\ell}$ ein Pfad p in $G(D_h)$, so dass das Gewicht jeder Kante durch $D_\ell(k, \ell)$ beschränkt ist. Somit ist p auch ein Pfad in $G_{k,\ell}$ von v nach w. Also befinden sich v und w in derselben Zusammenhangskomponente von $G_{k,\ell}$. ■

Notation 7.48 *Sei T ein Baum. Den eindeutigen einfachen Pfad von $v \in V(T)$ nach $w \in V(T)$ bezeichnen wir mit $p_T(v, w)$.*

Im Folgenden sei T ein minimaler Spannbaum von $G(D_h)$. Mit obiger Notation gilt dann, dass $D(T, v, w) = c(p_T(v, w))$. Wir werden jetzt zeigen, dass wir die oberen Schranken, die eigentlich durch die Matrix D_h gegeben sind, durch den zugehörigen minimalen Spannbaum von $G(D_h)$ mit viel weniger Speicherplatz darstellen können.

Lemma 7.49 *Sei D_h eine Distanzmatrix und sei T ein minimaler Spannbaum des Graphen $G(D_h)$. Dann gilt $D(T, v, w) = D(G(D_h), v, w)$ für alle Knoten $v, w \in V(T) = V(G(D_h))$.*

Beweis: Zuerst halten wir fest, dass jeder Pfad in T auch ein Pfad in $G(D_h)$ ist. Somit gilt in jedem Falle

$$D(G(D_h), v, w) \leq D(T, v, w).$$

Für einen Widerspruchsbeweis nehmen wir jetzt an, dass es zwei Knoten v und w mit

$$D(G(D_h), v, w) < D(T, v, w)$$

gibt. Dann existiert ein Pfad p in $G(D_h)$ von v nach w mit $c(p) < D(T, v, w)$.

Wir betrachten jetzt den eindeutigen Pfad $p_T(v, w)$ im minimalen Spannbaum T. Sei jetzt (x, y) eine Kante in $p_T(v, w)$mit maximalem Gewicht, also $\gamma(x, y) = c(p_T)$. Wir entfernen jetzt diese Kante aus T und erhalten somit zwei Teilbäume T_1 und T_2, die alle Knoten des Graphen $G(D_h)$ beinhalten. Dies ist in der Abbildung 7.47 illustriert.

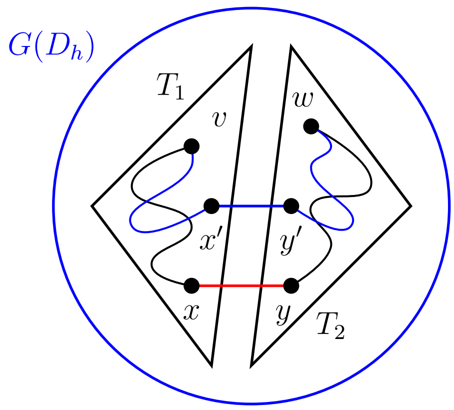


Abbildung 7.47: Skizze: Spannender Wald $\{T_1, T_2\}$ von $G(D_h)$ nach Entfernen von $\{x, y\}$ aus dem minimalen Spannbaum T

Sei (x', y') die erste Kante auf dem Pfad p in $G(D_h)$, die die Bäume T_1 und T_2 verbindet, d.h. $x' \in V(T_1)$ und $y' \in V(T_2)$. Nach Voraussetzung gilt, dass

$$D_h(x', y') < D_h(x, y),$$

da jede Kante des Pfades p nach Widerspruchsannahme leichter sein muss als die schwerste Kante in p_T und da (x, y) ja eine schwerste Kante in p_T war.

Dann könnten wir jedoch einen neuen Spannbaum T' mittels

$$E(T') = (E(T) \setminus \{(x,y)\}) \cup \{(x',y')\}$$

konstruieren. Dieser hat dann Gewicht

$$\gamma(T') \;=\; \gamma(T) + \underbrace{\gamma(x',y' - \gamma(x,y)}_{<0} \;<\; \gamma(T).$$

Somit hätten wir einen Spannbaum mit einem kleineren Gewicht konstruiert als der des minimalen Spannbaumes, was offensichtlich ein Widerspruch ist. ■

Damit haben wir gezeigt, dass wir im Folgenden die Informationen der Matrix D_h durch die Informationen im minimalen Spannbaum T von $G(D_h)$ ersetzen können.

Definition 7.50 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen. Zwei Knoten $v, w \in [1:n]$ heißen* separabel*, wenn $D_\ell(v,w) \leq D(G(D_h), v, w)$ gilt.*

Die Separabilität von zwei Knoten ist eine nahe liegende Eigenschaft, die wir benötigen werden, um zu zeigen, dass es möglich ist einen ultrametrischen Baum zu konstruieren, in dem der verbindende Pfad sowohl die untere als auch die obere Schranke für den Abstand einhält. Wir werden dies gleich formal beweisen, aber wir benötigen dazu erst noch ein paar Definitionen und Lemmata. Zuerst halten wir aber noch das folgenden Korollar fest, das aus dem vorherigen Lemma und der Definition unmittelbar folgt.

Korollar 7.51 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen und sei T ein minimaler Spannbaum von $G(D_h)$. Zwei Knoten $v, w \in [1:n]$ sind genau dann separabel, wenn $D_\ell(v,w) \leq D(T, v, w)$ gilt.*

Bevor wir den zentralen Zusammenhang zwischen paarweiser Separabilität und der Existenz ultrametrischer Sandwich-Bäume zeigen, benötigen wir noch die folgende Definition.

Definition 7.52 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen und sei T ein minimaler Spannbaum von $G(D_h)$. Eine Kante (x, y) des Pfades $p_T(v, w)$ im minimalen Spannbaum, der v und w verbindet, heißt* link-edge*, wenn sie eine Kante maximalen Gewichtes in $p_T(v, w)$ ist, d.h. wenn*

$$D_h(x, y) = c(p_T(v, w)) = D(T, v, w)$$

gilt. Mit $\mathrm{Link}(v, w)$ *bezeichnen wir die Menge der Link-edges für das Knotenpaar (v, w).*

Anschaulich ist die Link-Edge eines Pfades im zur oberen Schrankenmatrix gehörigen Graphen diejenige, die den maximalen Abstand von zwei Knoten bestimmt, die durch diesen Pfad verbunden werden. Aus diesem Grund werden diese eine besondere Rolle spielen.

Definition 7.53 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen und sei T ein minimaler Spannbaum von $G(D_h)$. Für jede Kante $(x, y) \in E(T)$ im minimalen Spannbaum ist die* cut-weight $\mathrm{CW}(x, y)$ *wie folgt definiert:*

$$\mathrm{CW}(x, y) := \max \{D_\ell(v, w) \ : \ (x, y) \in \mathrm{Link}(x, y)\}.$$

Die cut-weight einer Kante ist der maximale Mindestabstand zweier Knoten, deren Verbindungspfad diese Kante als link-edge besitzt. Um ein wenig mehr Licht in die Bedeutung der cut-weight zu bringen, betrachten wir jetzt nur die maximale auftretende cut-weight eines minimalen Spannbaumes des zu den Maximalabständen gehörigen Graphen.

Für diese maximale cut-weight c^* gilt dann $c^* = \mathrm{MAX}(D_\ell)$, wie man sich leicht überlegt. Wie im einfachen Algorithmus werden wir jetzt versuchen alle schwereren Kanten in $G_{k\ell}$ (der ja ein Teilgraph von $G(D_h)$ ist) zu entfernen. Statt $G_{k\ell}$ betrachten wir jedoch den minimalen Spannbaum T von $G(D_h)$ (der ja nach Definition auch ein Teilgraph von $G(D_H)$ ist).

In $G_{k\ell}$ werden alle schwereren Kanten entfernt, damit $G_{k\ell}$ in mehrere Zusammenhangskomponenten zerfällt. Zuerst überlegt man sich, dass es auch genügen würde, so viele von den schwersten Kanten zu entfernen, bis zwei Zusammenhangskomponenten entstehen. Dies wäre jedoch algorithmisch sehr aufwendig und würde in der Regel keinen echten Zeitvorteil bedeuten. Im minimalen Spannbaum T lässt sich dies hingegen sehr leicht durch Entfernen der Kante mit der maximalen cut-weight erzielen. Im Beweis vom übernächsten Lemma werden wir dies noch genauer sehen.

Das folgende Lemma stellt noch einen fundamentalen Zusammenhang zwischen Kantengewichten in Kreisen zu additiven Matrizen gehörigen gewichteten Graphen und der Eigenschaft einer Ultrametrik dar, die wir im Folgenden benötigen, um leicht von einer additiven Matrix nachweisen zu können, dass sie bereits ultrametrisch ist.

Lemma 7.54 *Eine additive Matrix M ist genau dann ultrametrisch ist, wenn für jede Folge $(i_1, \ldots, i_k) \in \mathbb{N}^k$ paarweise verschiedener Werte mit $k \geq 3$ gilt, dass in der Folge $(M(i_1, i_2), M(i_2, i_3), \ldots, M(i_{k-1}, i_k), M(i_k, i_1))$ das Maximum mehrfach angenommen wird.*

Beweis: **$\Leftarrow$:** Sei M eine additive Matrix und es gelte für alle $k \geq 3$ und für alle Folgen $(i_1, \ldots, i_k) \subset \mathbb{N}^k$ mit paarweise verschiedenen Folgengliedern, dass

$$\max\{M(i_1, i_2), M(i_2, i_3), \ldots, M(i_{k-1}, i_k), M(i_k, i_1)\} \tag{7.1}$$

nicht eindeutig ist.

Wenn man in (7.1) $k = 3$ einsetzt, gilt insbesondere für beliebige $a, b, c \in \mathbb{N}$, dass

$$\max\{M(a, b), M(b, c), M(c, a)\}$$

nicht eindeutig ist und damit ist M also ultrametrisch.

$\Rightarrow$: Sei M nun ultrametrisch, dann ist M auch additiv. Wir müssen nur noch (7.1) zu zeigen. Sei $S = (i_1, \ldots, i_k) \in \mathbb{N}^k$ gegeben. Wir betrachten zunächst i_1, i_2 und i_3. Aus der fundamentalen Eigenschaft einer Ultrametrik wissen wir, dass das Maximum von $(M(i_1, i_2), M(i_2, i_3), M(i_3, i_1))$ nicht eindeutig ist. Wir beweisen (7.1) per Induktion:

Gilt für j, dass das Maximum von $(M(i_1, i_2), M(i_2, i_3), \ldots, M(i_{j-1}, i_j), M(i_j, i_1))$ nicht eindeutig ist, so gilt dies auch für $j + 1$. Dazu betrachten wir i_1, i_j und i_{j+1}. Weiter wissen wir, dass $\max\{M(i_1, i_j), M(i_j, i_{j+1}), M(i_{j+1}, i_1)\}$ nicht eindeutig ist, d.h. einer der Werte ist kleiner als die anderen beiden. Betrachten wir wie sich das Maximum ändert, wenn wir $M(i_j, i_1)$ herausnehmen und dafür $M(i_j, i_{j+1})$ und $M(i_{j+1}, i_1)$ dazugeben.

Fall 1: $M(i_1, i_j)$ ist der kleinste Wert. Durch das Hinzufügen von $M(i_j, i_{j+1})$ und $M(i_{j+1}, i_1)$ kann das Maximum nicht eindeutig werden, denn beide Werte sind gleich. Liegen sie unter dem alten Maximum ändert sich nichts, liegen sie darüber, bilden beide das nicht eindeutige neue Maximum. Das Entfernen von $M(i_1, i_j)$ spielt nur eine Rolle, falls $M(i_1, i_j)$ vorher ein Maximum war. In dem Fall sind aber $M(i_j, i_{j+1})$ und $M(i_{j+1}, i_1)$ das neue nicht eindeutige Maximum.

Fall 2: $M(i_{j+1}, i_1)$ ist der kleinste Wert. Dann ist $M(i_1, i_j) = M(i_j, i_{j+1})$. Das Entfernen von $M(i_1, i_j)$ wird durch das Hinzufügen von $M(i_j, i_{j+1})$ wieder ausgeglichen. $M(i_{j+1}, i_1)$ wird auch hinzugefügt, spielt aber keine Rolle.

Fall 3: $M(i_j, i_{j+1})$ ist der kleinste Wert. Dann ist $M(i_1, i_j) = M(i_1, i_{j+1})$. Dieser Fall ist analog zu Fall 2. ■

Nun kommen wir zum Beweis unseres zentralen Lemmas, dass es genau dann einen ultrametrischen Baum für einen gegebene Sandwich-Bedingung gibt, wenn alle Knoten paarweise separabel sind. Der Bewies in die eine Richtung wird konstruktiv sein und wird uns somit einen effizienten Algorithmus an die Hand geben.

Lemma 7.55 *Seien $D_\ell \leq D_h$ zwei $n \times n$-Distanzmatrizen. Ein ultrametrischer Baum $U \in [D_\ell, D_h]$ existiert genau dann, wenn jedes Paar $v, w \in [1 : n]$ von Knoten separabel ist.*

Beweis: **⇒:** Für einen Widerspruchsbeweis nehmen wir an, dass v und w nicht separabel sind. Dann ist $D_\ell(v, w) > D_h(x, y)$ für alle $\{x, y\} \in \text{Link}(v, w)$. Für jede Kante $\{a, b\}$ in $p_T(v, w)$ gilt dann $D_h(a, b) \leq D_h(x, y)$ für alle $\{x, y\} \in \text{Link}(v, w)$. Also ist $\{v, w\} \notin p_T(v, w)$. Damit ist $D_\ell(x, y) > D_h(a, b)$ für alle $\{a, b\} \in \text{Link}(v, w)$. Damit muss die Kante $\{x, y\}$ im Kreis gebildet aus $p_T(x, y)$ und der Kante $\{x, y\}$ in einer ultrametrischen Matrix das eindeutige Maximum sein. Dies steht jedoch im Widerspruch zu Lemma 7.54 und diese Implikation ist bewiesen.

⇐: Sei T ein minimaler Spannbaum von $G(D_h)$ und sei $E(T) = \{e_1, \ldots, e_{n-1}\}$, wobei $\text{CW}(e_1) \geq \cdots \geq \text{CW}(e_{n-1})$. Wir entfernen jetzt sukzessive die Kanten aus T gemäß der cut-weight der Kanten. Dabei wird durch jedes Entfernen ein Baum in zwei neue Bäume zerlegt.

Betrachten wir jetzt nachdem Entfernen der Kanten $e_1, \ldots, e_{i-1}$ den entstandenen Wald und darin den Baum T', der die Kante $e_i = \{v, w\}$ enthält. Das Entfernen der Kante $e_i = \{v, w\}$ zerlegt den Baum T' in zwei Bäume T'_1 und T'_2. Wir setzten dann $d_U(v, w) := \text{CW}(e_i)$ für alle $v \in V(T'_1)$ und $w \in V(T'_2)$.

Wir zeigen als erstes, dass $d_U(v, w) \geq D_\ell(v, w)$ für alle $v, w \in [1 : n]$. Wir betrachten die Kante e, nach deren Entfernen die Knoten v und w im Rest des minimalen Spannbaums nicht mehr durch einen Pfad verbunden sind. Ist e eine link-edge in $p_T(v, w)$, dann gilt nach Definition der cut-weight: $\text{CW}(e) \geq D_\ell(v, w)$. Ist e hingegen keine link-edge in $p_T(v, w)$, dann gilt $\text{CW}(e) \geq \text{CW}(e')$ für jede link-edge $e' \in \text{Link}(v, w)$, da wir die Kanten gemäß ihrer absteigenden cut-weight aus dem minimalen Spannbaum T entfernen. Somit gilt nach Definition der cut-weight:

$$\mathrm{CW}(e) \geq \mathrm{CW}(e') \geq D_\ell(v, w).$$

Wir zeigen jetzt, dass ebenfalls $d_U(v, w) \leq D_h(v, w)$ für alle $v, w \in [1 : n]$ gilt. Nach Definition der cut-weight gilt, dass ein Knotenpaar $\{x, y\}$ existiert, so dass für alle $\{a, b\} \in \mathrm{Link}(x, y)$ gilt: $D_\ell(x, y) = \mathrm{CW}(a, b)$. Da x und y nach Voraussetzung separabel sind, gilt

$$\mathrm{CW}(a, b) = D_\ell(x, y) \leq D(T, x, y) = D_h(a, b).$$

Die letzte Gleichheit folgt aus der Tatsache, dass $\{a, b\} \in \mathrm{Link}(x, y)$. Aufgrund der Konstruktion des minimalen Spannbaumes gilt weiterhin $D_h(a, b) \leq D_h(v, w)$. Somit gilt $d_U(v, w) = \mathrm{CW}(a, b) \leq D_h(v, w)$.

Damit haben wir gezeigt, dass $U \in [D_\ell, D_h]$. Wir müssen zum Schluss nur noch zeigen, dass U ultrametrisch ist. Nach Lemma 7.54 genügt es zu zeigen, dass in jedem Kreis in $G(U)$ das Maximum nicht eindeutig ist. Sei also C ein Kreis in $G(U)$ und (v, w) eine Kante maximalen Gewichtes in C. Falls es mehrere davon geben sollte, wählen wir eine solche, deren Endpunkte in unserem Konstruktionsverfahren durch Entfernen von Kanten im minimalen Spannbaum T zuerst separiert wurden.

Wir betrachten jetzt den Zeitpunkt in unserem Konstruktionsverfahren von d_U, als $d_U(v, w)$ gesetzt wurde. Zu diesem Zeitpunkt wurde v und w in zwei Bäume T' und T'' aufgeteilt. Ferner sind die Knoten dieses Kreises nach Wahl der Kante $\{v, w\}$ alle Knoten des Kreises in diesen beiden Teilbäumen enthalten. Da es in C noch einen anderen Weg von v nach w gibt, muss zu diesem Zeitpunkt auch für eine andere Kante $\{v', w'\}$ der Wert $d_U(v', w')$ ebenfalls festgelegt worden sein. Nach unserer Konstruktion gilt dann natürlich $d_U(v, w) = d_U(v', w')$ und in C ist das Maximum nicht eindeutig. ■

7.5.4 Algorithmus für das ultrametrische Sandwich-Problem

Der Beweis des vorherigen Lemmas liefert unmittelbar den folgenden Algorithmus, der in Abbildung 7.48 angegeben ist. Die Korrektheit des Algorithmus folgt im Wesentlichen aus dem Beweis des vorherigen Lemmas (wir gehen später noch auf ein paar implementierungstechnische Besonderheiten ein). Wir müssen uns nur noch um die effiziente Implementierung kümmern. Einen Algorithmus zur Bestimmung minimaler Spannbäume haben wir bereits kennen gelernt. Wir werden hier noch eine andere Möglichkeit darstellen, die für unsere Zwecke besser geeignet ist. Ebenso müssen wir uns noch um die effiziente Berechnung der cut-weights kümmern. Außerdem müssen wir noch erklären was kartesische Bäume sind und wie wir sie konstruieren können.

1. Bestimme minimalen Spannbaum T für $G(D_h)$. $O(n^2)$
2. Bestimme dabei den Kartesischen Baum R für den Zusammenbau von T $O(n^2)$
3. Bestimme den cut-weight der einzelnen Kanten aus T mit Hilfe des Kartesischen Baumes. $O(n^2)$
4. Baue den minimalen Spannbaum durch Entfernen der Kanten absteigend nach den cut-weights der Kanten ab und bauen parallel den ultrametrischen Baum U wieder auf. $O(n)$

Abbildung 7.48: Algorithmus: Effiziente Lösung des ultrametrische Sandwich-Problems

7.5.4.1 Kruskals Algorithmus für minimale Spannbäume

Zuerst stellen wir eine alternative Methode zu Prims Algorithmus zur Berechnung minimaler Spannbäume vor. Auch hier werden wir wieder den Greedy-Ansatz verwenden. Wir beginnen jedoch anstatt mit einem Baum aus einem Knoten, den wir sukzessive zu einem Spannbaum erweitern, mit einem Wald von Bäumen, die zu Beginn aus einelementigen Bäumen bestehen. Dabei stellt jeder Knoten genau einen Baum dar. Dann fügen wir sukzessive Kanten in diesem Wald hinzu, um dabei zwei Bäume mittels dieser Kante zu einem neuen größeren Baum zu verschmelzen. Nachdem wir $n-1$ Kanten hinzugefügt haben, haben wir unseren Spannbaum konstruiert.

Da wir gierig vorgehen, d.h. leichte Kanten bevorzugen, werden wir zuerst die Kanten aufsteigend nach Gewicht sortieren und versuchen diese in dieser Reihenfolge zu verwenden. Dabei müssen wir nur beachten, dass wir keine Kreise generieren (da Bäume durch Kreisfreiheit und Zusammenhang charakterisiert sind). Dazu merken wir uns mit einer so genannten *Union-Find-Datenstruktur* wie die Mengen der Knoten auf die verschiedenen Menge im Wald verteilt sind. Wenn wir feststellen, dass eine Kante innerhalb eines Baumes (also innerhalb einer Menge) verlaufen würde, dann verwerfen wir sie, andernfalls wird sie aufgenommen. Dieser Algorithmus ist im Detail in Abbildung 7.49 angegeben.

Halten wir noch, dass Ergebnis fest, wobei wir im nächsten Teilabschnitt noch zeigen werden, dass $\mathcal{UF}(n) = O(n^2)$ ist.

Lemma 7.56 *Sei $G = (V, E, \gamma)$ ein gewichteter Graph. Ein minimaler Spannbaum T für G kann mit Hilfe des Algorithmus von Kruskal in Zeit $O(n^2 + \mathcal{UF}(n))$ berechnet werden, wobei $\mathcal{UF}(n)$ die Zeit ist, die eine Union-Find-Datenstruktur bei n^2 Find- und n Union-Operationen benötigt.*

```
KRUSKAL (G = (V, E, γ))
{
    /* O.B.d.A. gelte γ(e_1) ≤ ··· ≤ γ(e_m) mit E = {e_1, ..., e_m} */
    set E' = ∅;
    /* (V, E') wird der konstruierte minimale Spannbaum sein */
    for (i = 1; i ≤ m; i++)
    {
        Sei e_i = {v, w};
        k = FIND(v);
        ℓ = FIND(w);
        if (k ≠ ℓ)
        {
            E' = E' ∪ {e_i};
            UNION(k, ℓ);
            if (|E'| = |V| − 1) return (V, E');
        }
    }
}
```

Abbildung 7.49: Algorithmus: Kruskals Algorithmus für minimale Spannbäume

7.5.4.2 Union-Find-Datenstruktur

Jetzt müssen wir noch genauer auf die so genannte Union-Find-Datenstruktur eingehen. Eine Union-Find-Datenstruktur für eine Grundmenge U beschreibt eine Partition $\mathcal{P} = \{P_1, \ldots, P_\ell\}$ für U mit $U = \bigcup_{i=1}^{\ell} P_i$ und $P_i \cap P_j = \emptyset$ für alle $i \neq j \in [1 : \ell]$. Dabei ist zu Beginn $\mathcal{P} = \{P_1, \ldots, P_{|U|}\}$ mit $P_u = \{u\}$ für alle $u \in U$. Weiterhin werden die beiden folgenden elementaren Operationen zur Verfügung gestellt:

Find-Operation: Gibt für eine Element $u \in U$ den Index der Menge in der Mengenpartition zurück, die u enthält.

Union-Operation: Vereinigt die beiden Menge mit Index i und Index j und vergibt für diese vereinigte Menge einen neuen Index.

Die Realisierung erfolgt durch zwei Felder. Dabei nehmen wir der Einfachheit halber an, dass $U = [1 : n]$ ist. Ein Feld von ganzen Zahlen namens Index gibt für jedes Element $u \in U$ an, welchen Index die Menge besitzt, die u enthält. Ein weiteres Feld von Listen namens Liste enthält für jeden Mengenindex eine Liste von Elementen, die in der entsprechenden Menge enthalten sind.

Die Implementierung der Prozedur Find ist nahe liegend. Es wird einfach der Index zurück gegeben, der im Feld Index gespeichert ist. Für die Union-Operation wer-

```
Union-Find (int n)
function initialize(int n)
{
    int Index[n];
    for (i = 1; i ≤ n; i++)
        Index[i] = i;
    <int> Liste[n];
    for (i = 1; i ≤ n; i++)
        Liste[i] = <i>;
}

function Find(int u)
{
    return Index[u];
}

function Union(int i, j)
{
    if (Liste[i].size() ≤ Liste[j].size())
    {
        for all (u ∈ Liste[i]) do
        {
            Liste[j].add(u);
            Index[u] = j;
            Liste[i].remove(u);
        }
    }
    else
    {
        for all (u ∈ Liste[j]) do
        {
            Liste[i].add(u);
            Index[u] = i;
            Liste[j].remove(u);
        }
    }
}
```

Abbildung 7.50: Algorithmus: Union-Find

den wir die Elemente einer Menge in die andere kopieren. Die umkopierte Menge wird dabei gelöscht und der entsprechende Index der wiederverwendeten Menge wird dabei recycelt. Um möglichst effizient zu sein, werden wir die Elemente der kleineren Menge in die größere Menge kopieren. Die detaillierte Implementierung ist in Abbildung 7.50 angegeben.

Wir überlegen uns jetzt noch die Laufzeit dieser Union-Find-Datenstruktur. Hierbei nehmen wir an, dass wir k Find-Operationen ausführen und maximal $n-1$ Union-Operationen. Mehr Union-Operationen machen keinen Sinn, da sich nach $n-1$ Union-Operationen alle Elemente in einer Menge befinden.

Offensichtlich kann jede Find-Operation in konstanter Zeit ausgeführt werden. Für die Union-Operation ist der Zeitbedarf proportional zur Anzahl der Elemente in der kleineren Menge, die in der Union-Operation beteiligt ist. Somit ergibt sich für die maximal $n-1$ möglichen Union-Operationen.

$$\sum_{\sigma=(L,L')} \sum_{\substack{i \in L \\ |L| \leq |L'|}} O(1).$$

Um diese Summe jetzt besser abschätzen zu können vertauschen wir die Summationsreihenfolge. Anstatt die äußere Summe über die Union-Operation zu betrachten, summieren wir für jedes Element in der Grundmenge, wie oft es bei einer Union-Operation in der kleineren Menge sein könnte.

$$\sum_{\sigma=(L,L')} \sum_{\substack{i \in L \\ |L| \leq |L'|}} O(1) = \sum_{u \in U} \sum_{\substack{\sigma=(L,L') \\ u \in L \\ |L| \leq |L'|}} O(1).$$

Was passiert mit einem Element, dass sich bei einer Union-Operation in der kleineren Menge befindet? Danach befindet es sich in einer Menge die mindestens doppelt so groß wie vorher ist, da diese mindestens so viele neue Element in die Menge hinzubekommt, wie vorher schon drin waren. Damit kann jedes Element maximal $\log(n)$ Mal in einer kleineren Menge bei einer Union-Operation gewesen sein, da sich dieses Element dann in einer Menge mit mindestens n Elementen befinden muss.

Da die Grundmenge aber nur n Elemente besitzt, kann danach überhaupt keine Union-Operation mehr ausgeführt werden, da sich dann alle Elemente in einer Menge befinden. Somit ist die Laufzeit für ein Element durch $O(\log(n))$ beschränkt. Da es maximal n Elemente gibt, ist die Gesamtlaufzeit aller Union-Operationen durch $O(n \log(n))$ beschränkt.

Theorem 7.57 *Sei U mit $|U| = n$ die Grundmenge für die vorgestellte Union-Find-Datenstruktur. Die Gesamtlaufzeit von k Find- und maximal $n-1$ Union-Operationen ist durch $O(k + n\log(n))$ beschränkt.*

Es gibt noch effizienter Implementierungen von Union-Find-Operationen, die wir hier aber nicht benötigen und daher auch nicht näher darauf eingehen wollen. Wir verweisen statt dessen auf die einschlägige Literatur.

7.5.4.3 Kartesische Bäume

Kommen wir nun zur Definition eine kartesischen Baumes.

Definition 7.58 *Sei M eine Menge von Objekten, $E \subset \binom{M}{2}$ eine Menge von Paaren, so dass der Graph (M, E) ein Baum ist und $\gamma : E \to \mathbb{R}$ eine Gewichtsfunktion auf E. Ein* kartesischer Baum *für (M, E) ist rekursiv wie folgt definiert.*

- *Ist $|M| = 1$, dann ist der einelementige Baum mit der Wurzelmarkierung $m \in M$ ein kartesischer Baum.*
- *Ist $|M| \geq 2$ und $\{v_1, v_2\} \in E$ mit $\gamma(\{v_1, v_2\}) = \max\{\gamma(e) : e \in E\}$. Sei weiter T' der Wald, der aus T durch Entfernen von $\{v_1, v_2\}$ entsteht, d.h. $V(T') = V(T)$ und $E(T') = E(T) \setminus \{v_1, v_2\}$. Sind T_1 bzw. T_2 kartesische Bäume für $C(T', v_1)$ bzw. $C(T', v_2)$, dann ist der Baum mit der Wurzel, die mit $\{v_1, v_2\}$ markiert ist und deren Teilbäume der Wurzel gerade T_1 und T_2 sind, ebenfalls ein kartesischer Baum.*

Wir bemerken hier noch explizit an, dass die Elemente aus M nur als Blattmarkierungen auftauchen und dass alle inneren Knoten mit den Elementen aus E markiert sind.

Betrachtet man auf der Menge $[1 : n]$ als Baum eine lineare Liste mit

$$E = \{\{i, i+1\} : i \in [1 : n-1]\} \quad \text{mit} \quad \gamma(i, i+1) = \max\{F[i], F[i+1]\},$$

wobei F ein Feld von Zahlen ist, so erhält man im Wesentlichen einen Heap, wobei hier die Werte an den Blättern stehen und die inneren Knoten den maximalen Wert ihres Teilbaumes besitzen, wenn man, wie hier üblich, anstatt der Kante in den inneren Knoten das Gewicht der Kante einträgt. Diese Struktur wird manchmal auch als kartesischer Baum bezeichnet wird.

Der kartesische Baum für einen minimalen Spannbaum lässt sich jetzt bei der Konstruktion mit Hilfe des Kruskal-Algorithmus sehr leicht mitkonstruieren. Man überlegt sich leicht, dass bei der Union-Operation, die zugehörigen kartesischen Bäume mit einer neuen Wurzel vereinigt werden, wobei die Kante in der Wurzel genau die Kante ist, die bei der Konstruktion des minimalen Spannbaumes hinzugefügt wird.

Lemma 7.59 *Sei $G = (V, E, \gamma)$ ein gewichteter Graph und T ein minimaler Spannbaum von G. Der kartesische Baum für T kann bei der Konstruktion des minimalen Spannbaumes T basierend auf Kruskals Algorithmus parallel mit derselben Zeitkomplexität mitkonstruiert werden.*

7.5.4.4 Berechnung der cut-weights

Wir wollen jetzt zeigen, dass wir die cut-weights einer Kante $e \in E$ im minimalen Spannbaum T mit Hilfe von lca-Anfragen im kartesischen Baumes T bestimmen können. Dazu gehen wir durch die Matrix D_ℓ und bestimmen für jedes Knotenpaar (i, j) den niedrigsten gemeinsamen Vorfahren $\mathrm{lca}(i, j)$ im kartesischen Baum R.

Die dort gespeicherte Kante e ist nach Konstruktion des kartesischen Baumes eine Kante mit größtem Gewicht auf dem Pfad von i nach j im minimalen Spannbaum T, d.h. $e \in \mathrm{Link}(i, j)$. Daher werden wir dort die cut-weight $\mathrm{CW}(e)$ dieser Kante mittels $\mathrm{CW}(e) = \max\{\mathrm{CW}(e), D_\ell(i, j)\}$ aktualisieren. Wir bemerken an dieser Stelle, dass es durchaus noch andere link-edges auf dem Pfad von i nach j im minimalen Spannbaum geben kann. Daher wird die cut-weight nicht für jede Kante korrekt berechnet. Wir gehen auf diesen „Fehler“ am Ende diese Abschnittes noch ein.

Lemma 7.60 *Sei $G = (V, E, \gamma)$ ein gewichteter Graph und T ein minimaler Spannbaum von G. Der kartesische Baum R für den minimalen Spannbaum T kann mit derselben Zeit konstruiert werden wie der minimale Spannbaum, sofern hierfür der Algorithmus von Kruskal verwendet wird.*

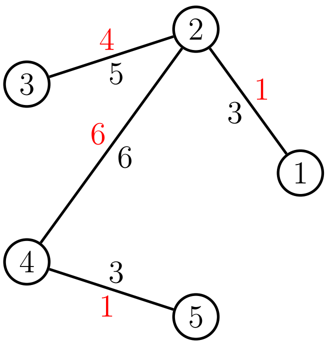
Nachdem wir jetzt die wesentlichen Schritte unseres effizienten Algorithmus weitestgehend verstanden haben, können wir uns einem Beispiel zuwenden. In der Abbildung 7.51 ist ein Beispiel angegeben, wie unser effizienter Algorithmus vorgeht.

7.5.4.5 Least Common Ancestor Queries

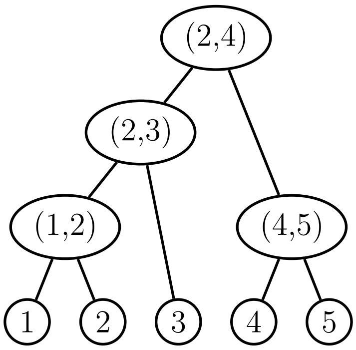
Wir müssen uns nun nur noch überlegen, wie wir $O(n^2)$ lca-Anfragen in Zeit $O(n^2)$ beantworten können. Dazu werden wir das lca-Problem auf das Range Minimum Query Problem reduzieren, das wie folgt definiert ist.

D_ℓ	1	2	3	4	5
1	0	1	2	3	6
2		0	4	5	5
3			0	4	5
4				0	1
5					0

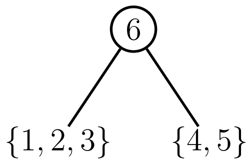
D_h	1	2	3	4	5
1	0	3	6	8	8
2		0	5	6	8
3			0	6	8
4				0	3
5					0



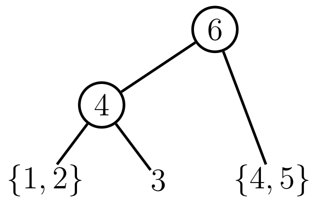
Minimaler Spannbaum T



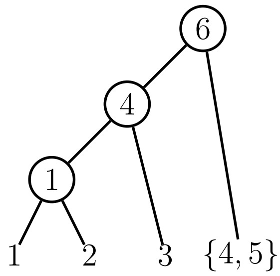
Kartesischer Baum R



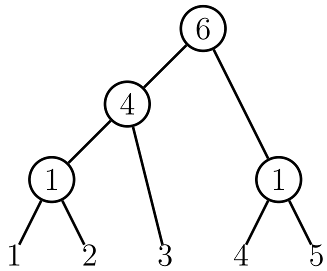
U_1 nach Entfernen $\{2, 4\}$



U_2 nach Entfernen $\{2, 3\}$



U_3 nach Entfernen $\{1, 2\}$



$U = U_4$ nach Entfernen $\{4, 5\}$

Abbildung 7.51: Beispiel: Lösung eines ultrametrischen Sandwich-Problems

RANGE MINIMUM QUERY

Eingabe: Eine Feld F der Länge n von reellen Zahlen und $i \leq j \in [1:n]$.
Ausgabe: Ein Index k mit $F[k] = \min\{F[\ell] : \ell \in [i:j]\}$.

Wir werden später zeigen, wie wir mit einem Preprocessing in Zeit $O(n^2)$ jede Anfrage in konstanter Zeit beantworten können. Für die Reduktion betrachten wir die so genannte *Euler-Tour* eines Baumes.

Definition 7.61 *Sei $T = (V, E)$ ein gewurzelter Baum mit Wurzel r und seien $T_1, \ldots, T_\ell$ die Teilbäume, die an der Wurzel hängen. Die* Euler-Tour *durch T ist eine Liste von $2n-1$ Knoten, die wie folgt rekursiv definiert ist:*

- *Ist $\ell = 0$, d.h. der Baum besteht nur aus dem Blatt r, dann ist diese Liste durch (r) gegeben.*
- *Für $\ell \geq 1$ seien $L_1, \ldots, L_\ell$ mit $L_i = (v_1^{(i)}, \ldots, v_{n_i}^{(i)})$ für $i \in [1:\ell]$ die Euler-Touren von $T_1, \ldots, T_\ell$. Die Euler-Tour von T ist dann durch*

$$(r, v_1^{(1)}, \ldots, v_{n_1}^{(1)}, r, v_1^{(2)}, \ldots, v_{n_2}^{(2)}, r, \ldots, r, v_1^{(\ell)}, \ldots, v_{n_\ell}^{(\ell)}, r)$$

definiert.

Der Leser sei dazu aufgefordert zu verifizieren, dass die oben definierte Euler-Tour eines Baumes mit n Knoten tatsächlich Liste mit $2n-1$ Elementen ist.

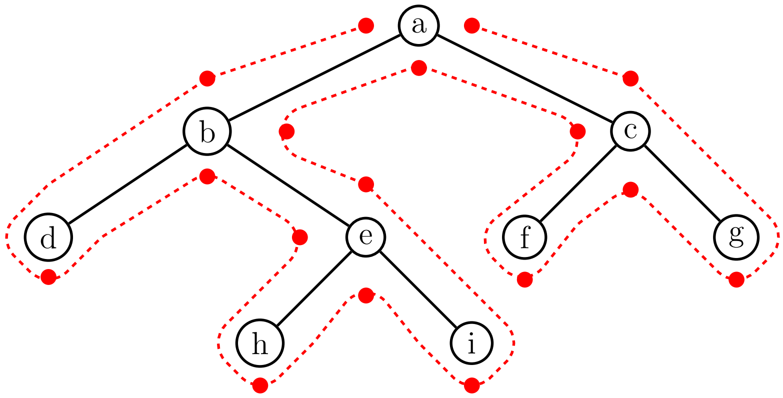
Die Euler-Tour kann sehr leicht mit Hilfe einer Tiefensuche in Zeit $O(n)$ berechnet werden. Der Algorithmus hierfür ist in Abbildung 7.52 angegeben. Man kann sich die Euler-Tour auch bildlich sehr schön als das Abmalen der Bäume anhand ihrer äußeren Kontur vorstellen, wobei bei jedem Antreffen eines Knotens des Baumes dieser in die Liste aufgenommen wird. Die ist in Abbildung 7.53 anhand eines Beispiels illustriert.

Zusammen mit der Euler-Tour, der Liste der abgelaufenen Knoten, betrachten wir zusätzlich noch DFS-Nummern des entsprechenden Knotens, die bei der Tiefensuche in der Regel mitberechnet werden (siehe auch das Beispiel in der Abbildung 7.53).

Betrachten wir jetzt die Anfrage an zwei Knoten des Baumes i und j. Zuerst bemerken wir, dass diese Knoten, sofern sie keine Blätter sind, mehrfach vorkommen können. Wir wählen jetzt für i und j willkürlich einen der Knoten, der in der Euler-Tour auftritt, als einen Repräsentanten aus. Zuerst stellen wir fest, dass in der Euler-Tour der niedrigste gemeinsame Vorfahren von i und j in der Teilliste, die durch die beiden Repräsentanten definiert ist, vorkommen muss, was man wie folgt sieht.

```
EULER-TOUR (tree T = (V, E))
{
    /* r(T) bezeichne die Wurzel von T */
    output r(T);
    /* N(r(T)) bezeichne die Menge der Kinder der Wurzel von T */
    for all (v ∈ N(r(T)))
    {
        /* T(v) bezeichne den Teilbaum mit Wurzel v */
        EULER-TOUR(T(v))
        output r(T);
    }
}
```

Abbildung 7.52: Algorithmus: Konstruktion einer Euler-Tour



Euler-Tour	a	b	d	b	e	h	e	i	e	b	a	c	f	c	g	c	a
DFS-Nummer	1	2	3	2	4	5	4	6	4	2	1	7	8	7	9	7	1

Abbildung 7.53: Beispiel: Euler-Tour

Wir nehmen an, dass i in der Euler-Tour vor j auftritt. In der Euler-Tour sind alle Knoten v, die nach einem Repräsentanten von i auftauchen und eine kleinere DFS-Nummer als i besitzen, ein Vorfahre von i. Da die DFS-Nummer von v kleiner als die von i ist und v in der Euler-Tour nach i auftritt, ist die DFS-Prozedur von v noch aktiv, als der Knoten i besucht wird. Das ist genau die Definition dafür, dass i ein Nachfahre von v ist. Analoges gilt für den Knoten j.

Wir betrachten jetzt nur die Knoten der Teilliste zwischen den beiden Repräsentanten von i und j, deren DFS-Nummer kleiner als die von i ist. Nach dem obigen sind dies Vorfahren von i. Nur einer davon, nämlich der mit der kleinsten DFS-Nummer, ist auch ein Vorfahre von j und muss daher der niedrigste gemeinsame Vorfahre von i und j sein. Dieser Knoten haben wir nämlich besucht, bevor wir in den Teilbaum von j eingedrungen sind. Alle Vorfahren davon haben wir mit betrachten der Teilliste eliminiert, die mit einem Repräsentanten von j endet.

Damit können wir das folgende Zwischenergebnis festhalten.

Lemma 7.62 *Gibt es eine Lösung für das Range Minimum Query Problem, dass für das Preprocessing Zeit $O(p(n))$ und für eine Anfrage $O(q(n))$ benötigt, so kann da Problem des niedrigsten gemeinsamen Vorfahren mit einen Zeitbedarf für das Preprocessing in Zeit $O(n + p(2n-1))$ und für eine Anfrage in Zeit $O(q(2n-1))$ gelöst werden.*

7.5.4.6 Range Minimum Queries

Damit können wir uns jetzt ganz auf das Range Minimum Query Problem konzentrieren. Offensichtlich kann ohne ein Preprocessing eine einzelne Anfrage mit $O(j-i) = O(n)$ Vergleichen beantwortet werden. Das Problem der Range Minimum Queries ist jedoch insbesondere dann interessant, wenn für ein gegebenes Feld eine Vielzahl von Range Minimum Queries durchgeführt werden. In diesem Fall kann mit Hilfe einer Vorverarbeitung die Kosten der einzelnen Queries gesenkt werden.

Ein triviale Lösung würde alle möglichen Anfragen vorab berechnen. Dazu könnte eine zweidimensionale Tabelle $Q[i, j]$ angelegt werden. Dazu würde für jedes Paar (i, j) das Minimum der Bereichs $F[i : j]$ mit $j - i$ Vergleiche bestimmt werden. Dies würde zu einer Laufzeit für die Vorverarbeitung von

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i) = \Theta(n^3)$$

führen. In der Abbildung 7.54 ist ein einfacher, auf dynamischer Programmierung basierender Algorithmus angegeben, der diese Tabelle in Zeit $O(n^2)$ berechnen kann. Damit erhalten wir das folgende Resultat für das Range Minimum Query Problem.

Theorem 7.63 *Für das Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n^2)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

Es gibt bereits wesentlich effizienter Verfahren für das Range Minimum Query Problem. In unserem Zusammenhang ist dieses leicht zu erzielende Ergebnis jedoch bereits völlig ausreichend und wir verweisen für die anderen Verfahren auf die einschlägige Literatur. Wir halten das für uns wichtige Ergebnis noch fest.

```
RMQ ( int F[], int n)
{
    for (i = 1; i ≤ n; i++)
        T[i, i] = i;

    for (i = 1; i ≤ n; i++)
        for (j = 1; i + j ≤ n; j++)
        {
            if (F[T[i, i + (j − 1)]] ≤ F[i + j])
                T[i, i + j] = T[i, i + j − 1];
            else
                T[i, i + j] = i + j;
        }
}
```

Abbildung 7.54: Algorithmus: Preprocessing für Range Minimum Queries

Theorem 7.64 *Sei T ein gewurzelter Baum mit n Knoten. Nach einer Vorverarbeitung, die in Zeit $O(n^2)$ durchgeführt werden kann, kann jede Anfrage nach einem niedrigsten gemeinsamen Vorfahren zweier Knoten aus T in konstanter Zeit beantwortet werden.*

7.5.4.7 Reale Berechnung der cut-weights

Wie bereits schon angedeutet berechnet unser effizienter Algorithmus nicht wirklich die cut-weights der Kanten des minimalen Spannbaumes. Dies passiert genau dann, wenn es im minimalen Spannbäume mehrere Kanten desselben Gewichtes gibt. Dazu betrachten wir das Beispiel, das in Abbildung 7.55 angegeben ist.

D_ℓ	1	2	3
1	0	1	3
2		0	2
3			0

D_h	1	2	3
1	0	5	6
2		0	5
3			0

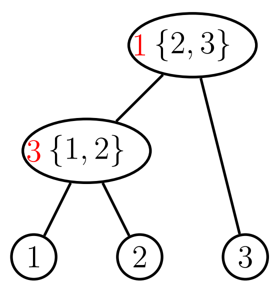


Abbildung 7.55: Beispiel: Falsche Berechnung der cut-weights

Hier stellen wir fest, das alle Kanten des minimalen Spannbaumes eine Kantengewicht von 5 besitzen. Somit sind alle Kanten des Baumes link-edges. Zuerst stellen wir fest, dass die cut-weight der Kante $\{2, 3\}$, die in der Wurzel des kartesischen Spannbaumes, mit 3 korrekt berechnet wird, da ja auch die Kante $\{1, 3\}$ der niedrigste gemeinsame Vorfahre von 1 und 3 im kartesischen Baum ist.

Im Gegensatz dazu wir die cut-weight der Kante $\{1, 2\}$ des minimalen Spannbaumes falsch berechnet. Diese Kante erhält die cut-weight 1, da die Kante $\{1, 2\}$ der niedrigste gemeinsame Vorfahre von 1 und 2 ist. Betrachten wir jedoch den Pfad von 1 über 2 nach 3, dann stellen wir fest, dass auch die Kante $\{1, 2\}$ eine link-edge im Pfad von 1 nach 3 im minimalen Spannbaum ist und die cut-weight der Kante $\{1, 2\}$ im minimalen Spannbaum daher ebenfalls 3 sein müsste.

Eine einfache Lösung wäre es die Kantengewicht infinitesimal so zu verändern, dass alle Kantengewichte im minimalen Spannbaum eindeutig wären. Wir können jedoch auch zeigen, dass der Algorithmus weiterhin korrekt ist, obwohl er die cut-weights von manchen Kanten falsch berechnet.

Zuerst überlegen wir uns, welche Kanten eine falsche cut-weight bekommen. Dies kann nur bei solchen Kanten passieren, deren Kantengewicht im minimalen Spannbaum nicht eindeutig ist. Zum anderen müssen diese Kanten bei der Konstruktion des minimalen Spannbaumes vor den anderen Kanten gleichen Gewichtes im minimalen Spannbaum eingefügt worden sein. Dies bedeutet, dass diese Kante im kartesischen Baum ein Nachfahre einer anderen Kante des minimalen Spannbaum gleichen Gewichtes sein muss.

Überlegen wir uns, was im Algorithmus passiert. Wir entfernen ja die Kanten aus dem minimalen Spannbaum nach fallendem cut-weight. Somit werden von zwei Kanten im minimalen Spannbaum, die dasselbe Kantengewicht besitzen und dieselbe cut-weight besitzen sollten, die Kante entfernt, die sich weiter oben im kartesischen Baum befindet. Damit zerfällt der minimale Spannbaum in zwei Teile und die beiden Knoten der untere Schranke die für die cut-weight der entfernten Kante verantwortlich sind befinden sich in zwei verschiedenen Zusammenhangskomponenten.

Somit ist die cut-weight der Kante, die ursprünglich falsch berechnet worden, in der neuen Zusammenhangskomponente jetzt nicht mehr ganz so falsch. Entweder ist sie für die konstruierte Zusammenhangskomponente korrekt und wir können mit unserem Algorithmus fortfahrten und er bleibt korrekt. War sie andernfalls falsch, befindet sich in minimalen Spannbaum dieser Zusammenhangskomponente eine weitere Kante mit demselben Gewicht, die im kartesischen Baum ein Vorfahre der betrachteten Kante ist und die ganze Argumentation wiederholt sich.

Also obwohl der Algorithmus nicht die richtigen cut-weights berechnet, ist zumindest immer die cut-weight der Kante mit der schwersten cut-weight korrekt berechnet und dies genügt für die Korrektheit des Algorithmus völlig, wie eine kurze Inspektion des zugehörigen Beweises ergibt.

7.5.5 Approximationsprobleme

Wir kommen jetzt noch einmal zu dem Approximationsproblem für Distanzmatrizen zurück.

ULTRAMETRISCHES APPROXIMATIONSPROBLEM

Eingabe: Eine $n \times n$-Distanzmatrizen D.
Ausgabe: Eine ultrametrische Distanzmatrix D', die $\|D - D'\|$ minimiert.

Das entsprechende additive Approximationsproblem ist leider $\mathcal{NP}$-hart. Das ultrametrische Approximationsproblem kann für die Maximumsnorm $\|\cdot\|_\infty$ in Zeit $O(n^2)$ gelöst werden. Für die anderen p-Normen ist das Problem ebenfalls wieder $\mathcal{NP}$-hart.

Theorem 7.65 *Das ultrametrische Approximationsproblem für die Maximumsnorm kann in linearer Zeit (in der Größe der Eingabe) gelöst werden.*

Beweis: Sei D die gegebene Distanzmatrix. Eigentlich müssen wir nur ein minimales $\varepsilon > 0$ bestimmen, so dass es eine ultrametrische Matrix U mit $U \in [D - \varepsilon, D + \varepsilon]$ gibt. Hierbei ist $D + x$ definiert durch $D = (d_{i,j} + x)_{i,j}$.

Wir berechnen also zuerst wieder den minimalen Spannbaum T für $G(D)$. Dann berechnen wir die cut-weights für die Kanten des minimalen Spannbaumes T. Dabei wählen wir für jede Kante e ein minimales ε_e, so dass die Knotenpaare separabel sind, d.h. $\mathrm{CW}(e) - \varepsilon_e \leq D(e) + \varepsilon_e$ für alle Kanten $e \in E(T)$.

Damit dies für alle Kanten des Spannbaumes gilt wählen ε als das Maximum dieser, d.h.

$$\varepsilon := \max\{\varepsilon_e \ : \ e \in E(T)\} = \frac{1}{2}\max\{\mathrm{CW}(e) - D(e) \ : \ e \in E(T)\}.$$

Dann führen wir denselben Algorithmus wie für das ultrametrische Sandwich Problem mit $D_\ell := D - \varepsilon$ und $D_h = D + \varepsilon$ durch. ■

Hidden Markov Modelle

8.1 Markov-Ketten

Um im Folgenden einige Problemstellungen mit Hilfe von so genannten Hidden Markov Modellen modellieren zu können, müssen wir uns zuerst mit den so genannten gewöhnlichen Markov-Ketten vertraut machen.

8.1.1 Definition von Markov-Ketten

Bevor wir zur Definition von Markov-Ketten kommen, wiederholen wir noch einmal die Definition einer stochastischen Matrix.

Definition 8.1 *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.*

Eine Markov-Kette ist 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.

Definition 8.2 *Eine* Markov-Kette *ist ein Tripel $M = (Q, P, \pi)$, wobei*

- *$Q = \{q_1, \ldots, q_n\}$ eine endliche Menge von Zuständen ist;*
- *P eine stochastische $n \times n$-Matrix der Zustandsübergangswahrscheinlichkeiten ist;*
- *$\pi = (\pi_1, \ldots, \pi_n)$ ist ein stochastischer Vektor der Anfangswahrscheinlichkeiten.*

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.

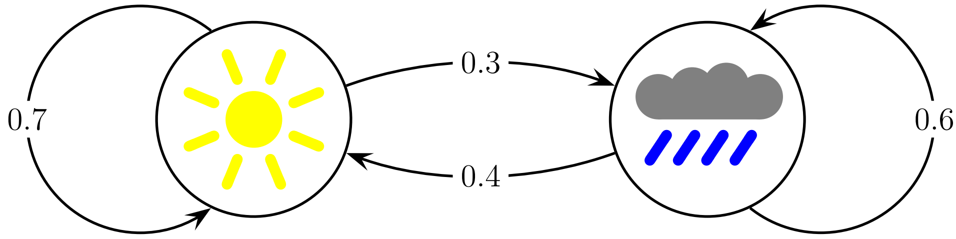


Abbildung 8.1: Beispiel: Einfache Markov-Kette für das Wetter

Betrachten wir dazu ein einfaches Beispiel, nämlich eine simple Modellierung des Wetters, wie sie in Abbildung 8.1 illustriert ist. Wir besitzen zwei Zustände, nämlich schönes Wetter und schlechtes Wetter. Die Übergangswahrscheinlichkeiten geben 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 $(x_1, \ldots, x_\ell) \in Q^*$ bezeichnen wir die Folge von Zuständen, die eine Markovkette zu den (diskreten) Zeitpunkten t_i mit $i \in [1 : \ell]$ und $t_1 < \cdots < t_\ell$ durchläuft. Dabei interpretieren wir gemäß der Definition einer Markov-Kette

$$\forall i \in [2 : \ell] : \ \forall r, s \in Q : \ \mathrm{Ws}(t_i = s \mid t_{i-1} = r) = p_{r,s}$$

und

$$\forall r \in Q : \ \mathrm{Ws}(t_1 = r) = \pi_r.$$

Wir bemerken hier insbesondere, dass die Wahrscheinlichkeiten der Zustandsübergänge nur von den Zuständen und nicht von den Zeitpunkten t_i mit $i \in [1 : \ell]$ selbst abhängen.

Man sagt Markov-Ketten 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:

$$\mathrm{Ws}(t_i = x_i \mid t_1 = x_1 \wedge \cdots \wedge t_{i-1} = x_{i-1}) = \mathrm{Ws}(t_i = x_i \mid t_{i-1} = x_{i-1}).$$

Die letzte Gleichung wird oft auch als *Markov-Eigenschaft* bezeichnet.

Die Markov-Eigenschaft bzw. Gedächtnislosigkeit besagt nicht anderes als dass die Wahrscheinlichkeit des Folgezustandes nur vom letzten Zustand abhängt.

Der Vollständigkeit halber wollen wir hier noch erwähnen, dass es auch *Markov-Ketten k-ter Ordnung* gibt. Hierbei 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 (dies sind genau die, die wir in Definition 8.2 definiert haben). 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 der Simulation seien dem Leser zur Übung überlassen.

Im Folgenden werden wir oft Sequenzen betrachten. Dann entsprechen die Zeitpunkte t_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.

8.1.2 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.

$$\begin{aligned}\mathrm{Ws}(X) &:= \mathrm{Ws}((x_1, \ldots, x_\ell)) \\ &:= \mathrm{Ws}((t_1, \ldots, t_\ell) = (x_1, \ldots, x_\ell)) \\ &:= \mathrm{Ws}(t_1 = x_1 \wedge \cdots \wedge t_\ell = x_\ell).\end{aligned}$$

Sei $X = (x_1, \ldots, x_\ell)$ eine Folge von Zuständen. Die Wahrscheinlichkeit, dass diese Zustandsfolge durchlaufen wird, berechnet sich dann zu

$$\mathrm{Ws}(X) = \pi_{x_1} \cdot \prod_{i=2}^{\ell} p_{x_{i-1},x_i}.$$

Die Anfangswahrscheinlichkeiten π kann auch durch einen neuen Startzustand q_0, der zum Zeitpunkt t_0 eingenommen wird, mit einer Erweiterung der Matrix P der Zustandsübergangswahrscheinlichkeiten eliminiert werden. Wir setzen dann

$$p_{0,j} := \mathrm{Ws}(t_1 = q_j \mid t_0 = q_0) = \pi_{x_j}$$

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:

$$\mathrm{Ws}(X) = \prod_{i=1}^{\ell} p_{x_{i-1},x_i}.$$

8.1.3 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 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 $X = (x_1, \ldots, x_\ell) \in \{A, C, G, T\}^\ell$.
Ausgabe: Befindet sich X innerhalb einer CpG-Insel.

Wir versuchen jetzt mit Hilfe von Markov-Ketten solche CpG-Inseln zu identifizieren. Dazu sind in Abbildung 8.2 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 8.2: Skizze: Zustandsübergangswahrscheinlichkeiten innerhalb und außerhalb von CpG-Inseln

Wir modellieren jeweils eine Markov-Kette 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:

$$
\begin{aligned}
\mathrm{Ws}_{P^+}(X) &= \pi(x_1) \cdot \prod_{i=2}^{\ell} p^+_{x_{i-1},x_i}, \\
\mathrm{Ws}_{P^-}(X) &= \pi(x_1) \cdot \prod_{i=2}^{\ell} p^-_{x_{i-1},x_i}.
\end{aligned}
$$

Wie bereits erwähnt, nehmen wir hier für die Anfangswahrscheinlichkeiten an, dass diese für beide Modell gleich ist.

Für die Entscheidung betrachten wir dann den Quotienten der entsprechenden Wahrscheinlichkeiten:

$$
\frac{\mathrm{Ws}_{P^+}(X)}{\mathrm{Ws}_{P^-}(X)} = \frac{\pi(x_1) \cdot \prod_{i=2}^{\ell} p^+_{x_{i-1},x_i}}{\pi(x_1) \cdot \prod_{i=2}^{\ell} p^-_{x_{i-1},x_i}} = \prod_{i=2}^{\ell} \frac{p^+_{x_{i-1},x_i}}{p^-_{x_{i-1},x_i}}.
$$

Je größer und weiter weg von 1 der Quotient ist, desto wahrscheinlicher ist es, dass wir uns innerhalb einer CpG-Insel befinden.

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:

$$
\mathrm{Score}(X) := \log\left(\frac{\mathrm{Ws}_{P^+}(X)}{\mathrm{Ws}_{P^-}(X)}\right) = \sum_{i=2}^{\ell} \left(\log(p^+_{x_{i-1},x_i}) - \log(p^-_{x_{i-1},x_i})\right).
$$

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.

Lokalisieren von CpG-Inseln

Eingabe: Eine lange DNS-Sequenz $X = (x_1, \ldots, x_\ell) \in \{A, C, G, T\}^\ell$.
Ausgabe: 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. Daher werden wir im nächsten Abschnitt versuchen, beide Modelle in einem Modell zu vereinigen.

8.2 Hidden Markov Modelle

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.

8.2.1 Definition

Wir geben zuerst die formale Definition eines Hidden Markov Modells an.

Definition 8.3 *Ein* Hidden Markov Modell M *(kurz* HMM*) ist ein 5-Tupel* $M = (Q, \Sigma, P, S, \pi)$*, 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 $X = (x_1, \ldots, x_\ell) \in Q^*$ 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 $X = (x_1, \ldots, x_\ell) \in Q^*$ in M sind die sichtbaren Symbolen eine Folge $Y = (y_1, \ldots, y_\ell) \in \Sigma^*$, 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 $s_{q,q} = 1$ und $s_{q,q'} = 0$ für $q \neq q' \in Q$, dann ist ein Hidden Markov Modell M nichts anderes als unsere bekannte Markov-Kette.

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 Modell M wie folgt gegeben:

$$\mathrm{Ws}_M(X \wedge Y) = \pi_{x_1} \cdot s_{x_1,y_1} \cdot \prod_{i=2}^{\ell} (p_{x_{i-1},x_i} \cdot s_{x_i,y_i}).$$

Im Allgemeinen ist uns bei den Hidden Markov Modellen 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.

8.2.2 Modellierung von CpG-Inseln

Wir werden jetzt die CpG-Inseln mit Hilfe von Hidden Markov Ketten modellieren. Das Hidden Markov Modell $M = (Q, \Sigma, P, S, \pi)$ 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

$$P = \left(\begin{array}{ccc|ccc} & & & \frac{p}{4} & \cdots & \frac{p}{4} \\ & P^+ \cdot (1-p) & & \vdots & & \vdots \\ & & & \frac{p}{4} & \cdots & \frac{p}{4} \\ \hline \frac{q}{4} & \cdots & \frac{q}{4} & & & \\ \vdots & & \vdots & & P^- \cdots (1-q) & \\ \frac{q}{4} & \cdots & \frac{q}{4} & & & \end{array}\right).$$

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.

8.2.3 Modellierung eines gezinkten Würfels

Wir geben nun noch ein Hidden Markov Modell 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.

8.3 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 Modell die größte Wahrscheinlichkeit für die gegebene Symbolfolge besitzt.

8.3.1 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 $Y \in \Sigma^*$.
Ausgabe: Ein wahrscheinlichster Pfad $X \in Q^*$ für Y in M, d.h. ein $X \in Q^*$, so dass $\mathrm{Ws}_M(X \mid Y)$ maximal ist.

Das bedeutet wir suchen eine Folge $X \in Q^*$ mit

$$\mathrm{Ws}(X \mid Y) = \max \{\mathrm{Ws}(Z \mid Y) \,:\, Z \in Q^*\}.$$

Dafür schreiben wir auch

$$X \in \operatorname{argmax} \{\mathrm{Ws}(Z \mid Y) \,:\, Z \in Q^*\}.$$

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.

8.3.2 Dynamische Programmierung

Wir lösen jetzt das Decodierungsproblem mit Hilfe der Dynamischen Programmierung. Dazu definieren wir:

$$\tilde{W}_q(i) := \max \{\mathrm{Ws}((x_1, \ldots, x_{i-1}, q) \mid (y_1, \ldots, y_i)) \,:\, (x_1, \ldots, x_{i-1}) \in Q^*\}.$$

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:

$$\tilde{W}_q(i) := \max\left\{ \frac{\mathrm{Ws}((x_1,\ldots,x_{i-1},q) \wedge (y_1,\ldots,y_i))}{\mathrm{Ws}((y_1,\ldots,y_i))} \;:\; (x_1,\ldots,x_{i-1}) \in Q^* \right\}.$$

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 $\mathrm{Ws}((x_1,\ldots,x_{i-1},q) \wedge (y_1,\ldots,y_i))$ maximal wird, und definieren hierfür:

$$W_q(i) := \max\left\{ \mathrm{Ws}((x_1,\ldots,x_{i-1},q) \wedge (y_1,\ldots,y_i)) \;:\; (x_1,\ldots,x_{i-1}) \in Q^* \right\}.$$

Es gilt offensichtlich:

$$\begin{aligned} W_q(1) &= \mathrm{Ws}(y_1 \wedge q) = \pi_q \cdot s_{q,y_1}, \\ W_q(i+1) &= \max\left\{ W_{q'}(i) \cdot p_{q',q} \cdot s_{q,y_{i+1}} \;:\; q' \in Q \right\}. \end{aligned}$$

Offensichtlich gilt $\mathrm{Ws}(Y \wedge X) = \max\{W_q(\ell) \;:\; q \in Q\}$. 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 $(|Q|)$ senken.

Theorem 8.4 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Modell und $Y \in \Sigma^*$. Eine Zustandsfolge $X \in Q^*$, die $\mathrm{Ws}(X \mid Y)$ maximiert, kann in Zeit $O(|Q|^2 \cdot |Y|)$ und Platz $O(|Q|)$ konstruiert werden.*

8.3.3 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:

$$\hat{W}_q(i) := \max\left\{\log(\mathrm{Ws}((x_1,\ldots,x_{i-1},q) \wedge (y_1,\ldots,y_i))) \,:\, (x_1,\ldots,x_{i-1}) \in Q^*\right\}.$$

Dafür ergeben sich dann die folgenden Rekursionsgleichungen:

$$\begin{aligned}
\hat{W}_q(1) &= \log(\pi_q) + \log(s_{q,y_1}), \\
\hat{W}_q(i+1) &= \log(s_{q,y_{i+1}}) + \max\left\{\hat{W}_{q'}(i) + \log(p_{q',q}) \,:\, q' \in Q\right\}.
\end{aligned}$$

Wie man sieht muss man nur einmal zu Beginn die teuren Kosten für das Logarithmieren investieren.

8.4 Posteriori-Decodierung

Wir wollen jetzt noch eine andere Möglichkeit angeben, wie man die Zustandsfolge im Hidden Markov Modell 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 Modell 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 $Y \in \Sigma^\ell$.
Ausgabe: Bestimme $\mathrm{Ws}(x_i = q \mid Y)$ für alle $i \in [1 : \ell]$ und für alle $q \in Q$.

8.4.1 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 $\mathrm{Ws}(X|Y) = \frac{\mathrm{Ws}(X \wedge Y)}{\mathrm{Ws}(Y)}$ um:

$$\begin{aligned} &\mathrm{Ws}(x_i = q \mid Y) \\ &\quad = \frac{\mathrm{Ws}(x_i = q \wedge Y)}{\mathrm{Ws}(Y)} \end{aligned}$$

Nach dem so genannten Multiplikationssatz gilt $\mathrm{Ws}(X \wedge Y) = \mathrm{Ws}(X) \cdot \mathrm{Ws}(Y \mid X)$ und wir erhalten somit:

$$= \frac{\mathrm{Ws}((y_1, \ldots, y_i) \wedge x_i = q) \cdot \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid (y_1, \ldots, y_i) \wedge x_i = q)}{\mathrm{Ws}(Y)}$$

Aufgrund der Definition von Hidden Markov Modellen 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:

$$\begin{aligned} &= \frac{\mathrm{Ws}((y_1, \ldots, y_i) \wedge x_i = q) \cdot \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_i = q)}{\mathrm{Ws}(Y)} \\ &= \frac{f_q(i) \cdot b_q(i)}{\mathrm{Ws}(Y)} \end{aligned}$$

Hierbei haben wir für die Umformung in der letzten Zeile die folgende Notation verwendet:

$$\begin{aligned} f_q(i) &:= \mathrm{Ws}(x_i = q \wedge (y_1, \ldots, y_i)), \\ b_q(i) &:= \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_i = q). \end{aligned}$$

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 bei $f_q(i)$ in der Zustandsfolge vorwärts aus $f_q(1), \ldots, f_q(i-1)$ berechnen lässt, während $b_q(i)$ in der Zustandsfolge rückwärts aus $b_q(\ell), \ldots, b_q(i+1)$ berechnen lässt.

8.4.2 Vorwärts-Algorithmus

Wir werden uns jetzt überlegen, wie sich die Vorwärtswahrscheinlichkeiten mit Hilfe der dynamische Programmierung berechnen lässt. Dazu stellen wir fest, dass die folgenden Rekursionsgleichungen gelten:

$$\begin{aligned} f_q(1) &= \pi_q \cdot s_{q,y_1}, \\ f_q(i+1) &= s_{q,y_{i+1}} \cdot \sum_{q' \in Q} (f_{q'}(i) \cdot p_{q',q}). \end{aligned}$$

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 wahrscheinlichen 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 $\mathrm{Ws}(Y) = \sum_{q \in Q} f_q(\ell)$ gilt.

Lemma 8.5 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Modell und $Y \in \Sigma^\ell$. Die Vorwärtswahrscheinlichkeit $f_q(i) = \mathrm{Ws}(x_i = q \wedge (y_1, \ldots, y_i))$ kann für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot |Y|)$ und Platz $O(|Q|)$ berechnet werden.*

Den Algorithmus zur Berechnung der Vorwärtswahrscheinlichkeiten wird in der Literatur auch oft als *Vorwärts-Algorithmus* bezeichnet.

8.4.3 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:

$$\begin{aligned} b_q(n) &= 1, \\ b_q(i) &= \sum_{q' \in Q} (p_{q,q'} \cdot s_{q',y_{i+1}} \cdot b_{q'}(i+1)). \end{aligned}$$

Auch hier gilt wieder $\mathrm{Ws}(Y) = \sum_{q \in Q} \pi_1 \cdot s_{q,y_1} \cdot b_q(1)$.

Lemma 8.6 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Modell und $Y \in \Sigma^\ell$. Die Rückwärtswahrscheinlichkeit $b_q(i) = \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_i = q)$ kann für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot |Y|)$ 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 8.7 *Sei $M = (Q, \Sigma, P, S, \pi)$ ein Hidden Markov Modell und $Y \in \Sigma^\ell$. Die Posteriori-Wahrscheinlichkeit $\mathrm{Ws}(x_i = q \mid Y)$ kann für alle $i \in [1 : \ell]$ und alle $q \in Q$ in Zeit $O(|Q|^2 \cdot |Y|)$ und Platz $O(|Q|)$ berechnet werden.*

8.4.4 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*:

$$
\begin{aligned}
\hat{f}_q(i) &:= \log(f_q(i)) = \log(\mathrm{Ws}(x_i = q \wedge (y_1, \ldots, y_i))), \\
\hat{b}_q(i) &:= \log(b_q(i)) = \log(\mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_i = 1)).
\end{aligned}
$$

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

$$
\begin{aligned}
\hat{b}_q(n) &= \log(s_{q,y_\ell}), \\
\hat{b}_q(i) &= \log\left(\sum_{q' \in Q} (p_{q,q'} \cdot s_{q',y_{i+1}} \cdot \exp(b_{q'}(i+1)))\right)
\end{aligned}
$$

gelten. Da hier nun statt der Maximumsbildung eine Summe involviert ist, müssen wir bei der dynamischen Programmierung sowohl Logarithmieren als auch Exponenzieren.

8.4.5 Anwendung

Mit Hilfe der Posteriori-Decodierung können wir jetzt einen alternativen Pfad $\tilde{X}$ wie folgt definieren:

$$\tilde{x}_i :\in \operatorname{argmax} \{ \mathrm{Ws}(x_i = q \mid Y) \,:\, q \in Q \} .$$

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 Modells $M = (Q, \Sigma, P, S, \pi)$ 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 8.3 illustriert.

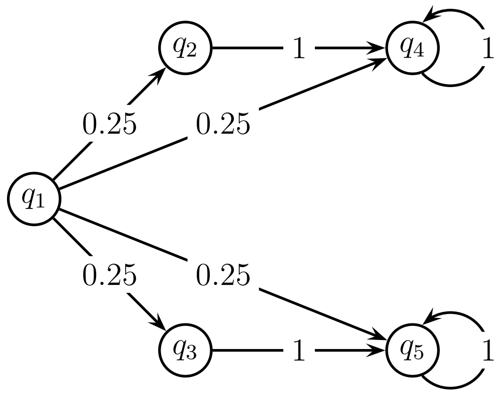


Abbildung 8.3: Beispiel: Ein HMM mit illegaler a posteriori Zustandsfolge

Die Matrix S der Emissionswahrscheinlichkeiten ist wie folgt gegeben:

$$S = \begin{pmatrix} s_1 := s_{q_1,a} & s_{q_1,b} = 1 - s_1 \\ s_2 := s_{q_2,a} & s_{q_2,b} = 1 - s_2 \\ s_3 := s_{q_3,a} & s_{q_3,b} = 1 - s_3 \\ s_4 := s_{q_4,a} & s_{q_4,b} = 1 - s_4 \\ s_5 := s_{q_5,a} & s_{q_5,b} = 1 - s_5 \end{pmatrix} .$$

Wir wählen jetzt $s_1 = 1$, $s_2 = 0.8$, $s_3 = 0.6$, $s_4 = 0.4$ 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 8.4 wurde 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.

$f_q(i)$	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	$\cdots$	$\frac{1}{4}$	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	$\cdots$	0.2500	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	$\cdots$	0.0000	0.0000
q_2	$\cdots$	0.0800	0.0000
q_3	$\cdots$	0.0750	0.0000
q_4	$\cdots$	0.0400	0.1200
q_5	$\cdots$	0.0625	0.1375

Abbildung 8.4: Beispiel: Die expliziten Vorwärts- und Rückwärts- sowie Posteriori-Wahrscheinlichkeiten für M

Wie man daraus abliеßt, 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 die mit dieser Methode vorhergesagte Zustandsfolge für die emittierte Folge *aaa* einen illegalen Zustandsübergang.

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:

$$G(i \mid Y) := \sum_{q \in Q} \mathrm{Ws}(x_i = q) \cdot g(q).$$

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.

8.5 Schätzen von HMM-Parametern

In diesem Abschnitt wollen wir uns mit dem Schätzen der Parameter eines Hidden Markov Modells beschäftigen. Hierbei nehmen wir an, dass uns die Zustände des Hidden Markov Modells 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 $i \in [1 : m]$.

8.5.1 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:

$$p_{q,q'} := \frac{\hat{p}_{q,q'}}{\sum_{r \in Q} \hat{p}_{q,r}}, \tag{8.1}$$

$$s_{q,a} := \frac{\hat{s}_{q,a}}{\sum_{\sigma \in \Sigma} \hat{s}_{q,\sigma}}. \tag{8.2}$$

Ist die Anzahl der Trainingssequenzen sehr klein, so verändern wir die Schätzer ein wenig, um künstliche Wahrscheinlichkeiten von 0 zu vermeiden. Dazu erhöhen wir die jeweilige beobachtete Anzahl um 1 und passen die Gesamtanzahl entsprechend an. 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.

8.5.2 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 (wobei wir wie bei der Berechnung der Vorwärts- und Rückwärtswahrscheinlichkeiten den Multiplikationssatz ($\mathrm{Ws}(X, Y) = \mathrm{Ws}(X) \cdot \mathrm{Ws}(Y \mid X)$) und die Markov-Eigenschaft bei fast jeder Zeile ausnutzen):

$$
\begin{aligned}
&\mathrm{Ws}(x_i = q, x_{i+1} = q' \mid Y, P, S) \\
&= \frac{\mathrm{Ws}(Y, x_i = q, x_{i+1} = q' \mid P, S)}{\mathrm{Ws}(Y)} \\
&= \frac{\mathrm{Ws}((y_1, \ldots, y_i), x_i = q \mid P, S) \cdot \mathrm{Ws}((y_{i+1}, \ldots, y_\ell), x_{i+1} = q' \mid x_i = q, P, S)}{\mathrm{Ws}(Y)} \\
&= \frac{1}{\mathrm{Ws}(Y)} \Big(\mathrm{Ws}((y_1, \ldots, y_i), x_i = q \mid P, S) \cdot \mathrm{Ws}(x_{i+1} = q' \mid x_i = q, P, S) \cdot \\
&\qquad \cdot \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_{i+1} = q', P, S) \Big) \\
&= \frac{1}{\mathrm{Ws}(Y)} \Big(\mathrm{Ws}((y_1, \ldots, y_i), x_i = q \mid P, S) \cdot \mathrm{Ws}(x_{i+1} = q' \mid x_i = q, P, S) \cdot \\
&\qquad \cdot \mathrm{Ws}(y_{i+1} \mid x_{i+1} = q', P, S) \cdot \mathrm{Ws}((y_{i+2}, \ldots, y_\ell) \mid x_{i+1} = q', P, S) \Big) \\
&= \frac{f_q(i) \cdot p_{q,q'} \cdot s_{q',y_{i+1}} \cdot b_{q'}(i+1)}{\mathrm{Ws}(Y)}.
\end{aligned}
$$

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 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 und somit mithilfe der Gleichung 8.1 bzw. 8.3 auch für die zugehörigen Zustandübergangswahrscheinlichkeit angeben:

$$
\begin{aligned}
\hat{p}_{q,q'} &:= \mathbb{E}((q \to q') \mid \vec{Y}, P, S) \\
&= \sum_{j=1}^{m} \sum_{i=1}^{\ell^{(j)}} \mathrm{Ws}(x_i = q, x_{i+1} = q' \mid Y^{(j)}, P, S)
\end{aligned}
$$

$$= \sum_{j=1}^{m} \frac{1}{\mathrm{Ws}(Y^{(j)})} \sum_{i=1}^{\ell(j)} f_q^{(j)}(i) \cdot p_{q,q'} \cdot s_{q,y_{i+q}} \cdot b_{q'}^{(j)}(i+1).$$

Analog können wir für die Emissionwahrscheinlichkeiten vorgehen. Berechnen wir zuerst wieder die Wahrscheinlichkeit für eine Emissionen von $a \in \Sigma$ aus dem Zustand $q \in Q$ für eine gegebene emittierte Folge Y unter Berücksichtigung der Matrizen P bzw. S der Zustandsübergangs- und Emissionswahrscheinlichkeiten:

$$\begin{aligned}
&\mathrm{Ws}(x_i = q \mid Y, P, S)\\
&\quad = \frac{\mathrm{Ws}(x_i = q, Y \mid P, S)}{\mathrm{Ws}(Y)}\\
&\quad = \frac{\mathrm{Ws}((y_1, \ldots, y_i), x_i = q \mid P, S) \cdot \mathrm{Ws}((y_{i+1}, \ldots, y_\ell) \mid x_i = q, P, S)}{\mathrm{Ws}(Y)}\\
&\quad = \frac{1}{\mathrm{Ws}(Y)} \cdot f_q(i) \cdot b_q(i+1).
\end{aligned}$$

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:

$$\begin{aligned}
\hat{s}_{q,a} &:= \mathbb{E}((q \to a) \mid \vec{Y}, P, S)\\
&= \sum_{j-1}^{m} \sum_{\substack{i=1\\ y_i^{(j)}=a}}^{\ell(j)} \mathrm{Ws}(x_i = q \mid Y^{(j)}, P, S)\\
&= \sum_{j=1}^{m} \frac{1}{\mathrm{Ws}(Y^{(j)})} \sum_{\substack{i=1\\ y_i^{(j)}=a}}^{\ell(j)} f_q^{(j)}(i) \cdot b_q^{(j)}(i).
\end{aligned}$$

Wir können also aus einer Schätzung der stochastischen Matrizen P und S und der der gegebenen emittierten Folgen 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 8.5 noch einmal skizziert.

8.5.3 Erwartungswert-Maximierungs-Methode

Der Baum-Welch-Algorithmus ist ein Spezialfall einer Methode, die in der in der Literatur als *Erwartungswert-Maximierungs-Methode* oder kurz *EM-Methode* bekannt

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 8.5: Algorithmus: Skizze des Baum-Welch-Algorithmus

ist, da wir versuchen aus einem Schätzer der Wahrscheinlichkeiten und den Trainingssequenzen versuchen, den Erwartungswert der gegebenen Trainingssequenzen unter den geschätzten Wahrscheinlichkeit zu bestimmen und anschließend die Schätzer zu verbessern, so dass sie den Erwartungswert in Richtung der beobachteten Werte zu maximieren.

Wir wollen jetzt allgemein zeigen, dass die EM-Methode auch in gewisser Weise konvergiert. Es könnte ja auch sein, dass wir zwischen zwei oder mehreren Verteilungen hin- und herhüpfen und nie einen endgültigen Schätzer erhalten.

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 $\mathrm{Ws}(\vec{Y} \mid \Psi)$ maximal wird. Wir wissen jedoch zusätzlich, dass die Folgen $\vec{Y}$ mit Hilfe von Zustandsfolgen $\vec{X}$ im Hidden Markov Modell generiert werden. Daher wissen wir mit Hilfe des Satzes der totalen Wahrscheinlichkeit und der Definition der bedingten Wahrscheinlichkeit, dass

$$\mathrm{Ws}(\vec{Y} \mid \Psi) \;=\; \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{Y} \mid \vec{X}, \Psi) \cdot \mathrm{Ws}(\vec{X} \mid \Psi) \;=\; \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi),$$

wobei $\vec{X} \in Q^{|\vec{Y}|}$ wieder Zustandsfolgen im gegebenen Hidden Markov Modell sind.

Für eine algorithmisch besser zugängliche Darstellung von $\mathrm{Ws}(\vec{Y} \mid X)$ bestimmen wir eine weitere Beziehung. Mit Hilfe der Definition der bedingten Wahrscheinlichkeit gilt:

$$\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi) = \frac{\mathrm{Ws}(\vec{X}, \vec{Y} \mid \Psi)}{\mathrm{Ws}(\vec{Y} \mid \Psi)}.$$

Durch Umformung und Logarithmieren erhalten wir

$$\log(\mathrm{Ws}(\vec{Y} \mid \Psi)) = \log(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi)) - \log(\mathrm{Ws}(\vec{X} \mid \Psi)). \tag{8.5}$$

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 $\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i)$

und summieren über alle Zustandfolgen $\vec{X} \in Q^{|\vec{Y}|}$. Wir erhalten dann:

$$\begin{aligned}
\log(\mathrm{Ws}(\vec{Y} \mid \Psi)) &= \underbrace{\sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i)}_{=1} \cdot \log(\mathrm{Ws}(\vec{Y} \mid \Psi)) \\
&= \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \cdot \log(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi)) \\
&\quad - \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \cdot \log(\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi)).
\end{aligned}$$

Anstatt nun $\mathrm{Ws}(\vec{Y} \mid \Psi)$ zu maximieren können wir aber auch $\log(\mathrm{Ws}(\vec{Y} \mid \Psi))$ 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 $\log(\mathrm{Ws}(\vec{Y} \mid \Psi)) > \log(\mathrm{Ws}(\vec{Y} \mid \Psi_i))$ wird und setzen dieses als Ψ_{i+1}. Damit suchen wir also nach einem Ψ mit

$$\sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi)) - \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log(\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi))$$

$$> \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi_i)) - \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log(\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i)).$$

Dies ist äquivalent dazu, dass die folgende Ungleichung gilt:

$$\sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log\left(\frac{\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi)}{\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi_i)}\right)$$

$$+ \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log\left(\frac{\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i)}{\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi)}\right) > 0.$$

In dieser Ungleichung ist der zweite Term auf der linken Seite die so genannte *Kullback-Leibler-Distanz* und dieses ist stets nichtnegativ (was man beispielsweise mit einigen Tricks mit Hilfe der Jensenschen Ungleichung beweisen kann). Es genügt

```
EM (int n, float Ψ[], float ε)
{
    Ψ_0 = Ψ;
    int i = 0;
    repeat
    {
        i++;
        /* E-Schritt */
        compute Δ(Ψ, Ψ_{i−1});
        /* M-Schritt */
        Ψ_i :∈ argmax {Δ(Ψ, Ψ_{i−1}) : Ψ};
    }
    until (‖Ψ_i − Ψ_{i−1}‖ < ε)
}
```

Abbildung 8.6: Skizze: Erwartungswert-Maximierungs-Methode

daher ein Ψ zu finden, so dass

$$\sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log \left(\frac{\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi)}{\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi_i)} \right) > 0$$

Definieren wir

$$\Delta(\Psi, \Psi_i) := \sum_{\vec{X} \in Q^{|\vec{Y}|}} \mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi_i) \log \Big(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi) \Big),$$

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

$$\Psi_{i+1} :\in \operatorname{argmax} \{\Delta(\Psi, \Psi_i) \ : \ \Psi\}.$$

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 $\log(\mathrm{Ws}(\vec{Y}, \vec{X} \mid \Psi))$ unter der Verteilung $\mathrm{Ws}(\vec{X} \mid \vec{Y}, \Psi)$ ist. Somit wird auch der Name Erwartungswert-Maximierungs-Methode klar, da wir zuerst einen

Erwartungswert bestimmen und dann mit Hilfe einer Maximierung de neuen Verteilungen wählen. Das Verfahren der EM-Methode ist in Abbildung 8.6 noch einmal schematisch dargestellt. Man kann zeigen , dass der Baum-Welch-Algorithmus ein Spezialfall der EM-Methode ist und somit auch konvergiert.

8.6 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 Modellen zu lösen.

8.6.1 Profile

Zunächst einmal benötigen wir noch eine Definition, was wir unter einem Profil verstehen wollen.

Definition 8.8 *Ein* Profil $\mathcal{P}$ *der Länge* ℓ *für ein Alphabet* Σ *ist eine Matrix* $(e_i(a))_{\substack{i\in[1:\ell]\\ a\in\Sigma}}$ *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 Sequenz 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:

$$\mathrm{Ws}(Y \mid \mathcal{P}) = \prod_{i=1}^{\ell} e_i(y_i).$$

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 Modell $M = (Q, \Sigma, P, S)$ 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 8.7)

$$\boxed{M_0} \xrightarrow{1.0} \boxed{M_1} \xrightarrow{1.0} \boxed{M_2} \longrightarrow \quad \cdots \quad \longrightarrow \boxed{M_{\ell-1}} \xrightarrow{1.0} \boxed{M_\ell} \xrightarrow{1.0} \boxed{M_{\ell+1}}$$

Abbildung 8.7: 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 Modells, da sie keine Zeichen ausgeben. Daher müssen wir unser Modell noch etwas erweitern.

Definition 8.9 *Ein Zustand eines Hidden Markov Modells, der keine Zeichen emittiert heißt* stiller Zustand *(engl.* silent state*).*

Somit sind die Zustände M_0 und $M_{\ell+1}$ unseres Hidden Markov Modells für ein gegebenes Profil der Länge ℓ stille Zustände.

8.6.2 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 Modell für solche Operationen erweitern.

Für die Einfügungen zwischen zwei Zeichen des Konsensus-Strings führen wir jetzt neue Zustände ein, die diese modellieren wollen. Dies ist in Abbildung 8.8 illustriert.

Nun sind die Zustandsübergangswahrscheinlichkeiten wieder offen. Wir werden später darauf zurückkommen, wie diese zu bestimmen sind. Die Emissionswahrscheinlichkeiten für die Insertions-Zustände sind durch $s_{I_i,a} = p(a)$ gegeben.

Betrachten wir jetzt noch eine Einfügung von k Zeichen zwischen zwei Matching-Zuständen, d.h. der Zustandsfolge $(M_i, I_i, \ldots, I_i, M_{i+1})$. Was bedeutet dies für den

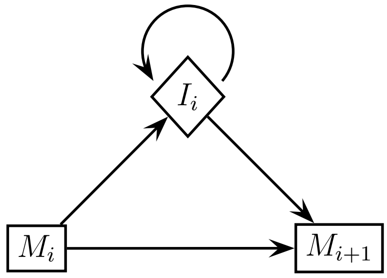


Abbildung 8.8: Skizze: Einbau von Insertionen

Score für die k Einfügungen? Der additive Teil für den Score beträgt dann:

$$\log(p_{M_i,I_i} \cdot p_{I_i,I_i}^{k-1} \cdot p_{I_i,M_{i+1}}) = \underbrace{\log(p_{M_i,I_i} \cdot p_{I_i,M_{i+1}})}_{\text{Gap-Eröffnungs-Strafe}} + \underbrace{(k-1)\log(p_{I_i,I_i})}_{\text{Gap-Verlängerungs-Strafe}} .$$

Man sieht, dass hier die Gaps gemäß einer affinen Lücken-Strafe bewertet werden. Allerdings hängt hier die Konstanten von der Position ab, d.h. sie sind nicht für alle Lücken 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 8.9 schematisch dargestellt ist.

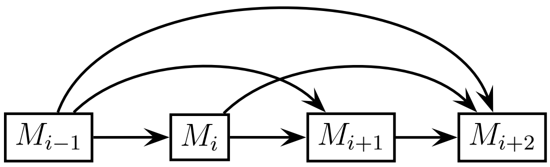


Abbildung 8.9: Skizze: Einbau von Deletionen

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 8.10 illustriert.

Hierbei ist jetzt wichtig, dass diese Deletions-Zustände stille Zustände sind, da ja beim Löschen nichts ausgegeben werden kann. Wir wollen hier noch anmerken, dass

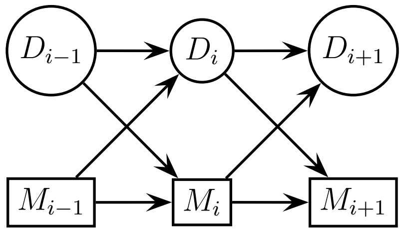


Abbildung 8.10: Skizze: Einbau von Deletions-Zuständen

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 den resultierenden Score für eine Folge von Deletionen $(M_i, D_{i+1}, \ldots, D_{i+k}, M_{i+k+1})$ genauer betrachten:

$$\log\left(p_{M_i,D_{i+1}} \cdot \prod_{j=1}^{k-1} p_{D_{i+j},D_{i+j+1}} \cdot p_{D_{i+k},M_{i+k+1}}\right)$$
$$= \underbrace{\log(p_{M_i,D_{i+1}} \cdot p_{D_{i+k},M_{i+k+1}})}_{\text{Gap}-\text{Eröffnungs}-\text{Strafe}} + \underbrace{\sum_{j=1}^{k-1} \log(p_{D_{i+j},D_{i+j+1}})}_{\text{Gap}-\text{Verlängerungs}-\text{Strafe}}.$$

Auch hier haben wir im Wesentlichen affine Lücken-Strafen. Allerdings werden die Deletionen unterschiedich bewertet, was natürlich von den entsprechenden Wahrscheinlichkeiten abhängt.

In der Abbildung 8.11 ist jetzt ein vollständiges Hidden Markov Modell 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 Modells 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.

8.6.3 Alignment gegen ein Profil-HMM

Um ein mehrfaches Sequenzen Alignment zu konstruieren, berechnen wir einen wahrscheinlichsten Pfad durch das zugehörige Hidden Markov Modell, das wir eben konstruiert haben. Der Pfad gibt uns dann für jede Sequenz die entsprechende Alignierung an.

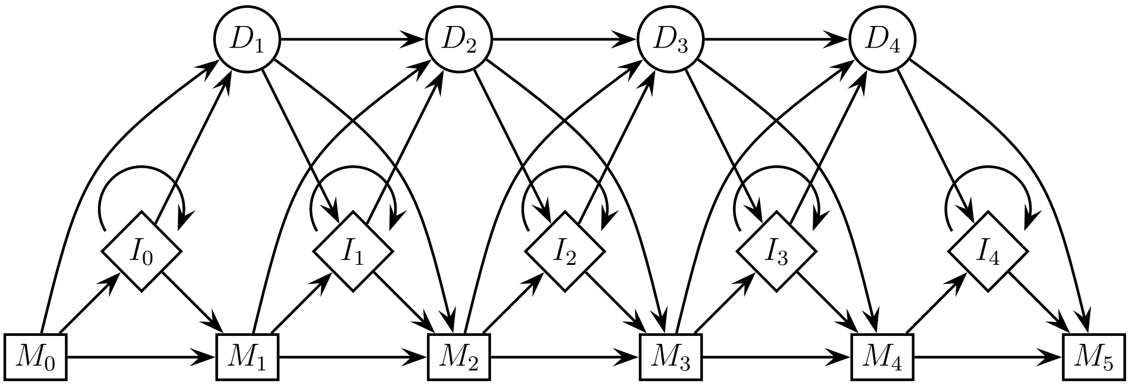


Abbildung 8.11: Skizze: Ein vollständiges HMM für $\ell = 3$

Wir merken an dieser Stelle noch an, dass wir hierfür die Zustandsübergangswahrscheinlichkeiten und die Emissionswahrscheinlichkeiten benötigen. Wir werden uns später 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:

$$\begin{aligned} M_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i)\text{, der in } M_j \text{ endet} \\ D_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i)\text{, der in } D_j \text{ endet} \\ I_j(i) &:= \text{Max. Score für } (y_1, \ldots, y_i)\text{, der in } I_j \text{ endet} \end{aligned}$$

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:

$$M_j(i+1) = \log\left(\frac{s_{M_j,y_{i+1}}}{p_{y_{i+1}}}\right) + \max\left\{\begin{array}{c} M_{j-1}(i) + \log(p_{M_{j-1},M_j}), \\ I_{j-1}(i) + \log(p_{I_{j-1},M_j}), \\ D_{j-1}(i) + \log(p_{D_{j-1},M_j}) \end{array}\right\}.$$

Für die Insertions-Zustände erhalten wir:

$$I_j(i+1) = \log\left(\frac{s_{I_j,y_{i+1}}}{p_{y_{i+1}}}\right) + \max\left\{\begin{array}{c} M_j(i) + \log(p_{M_j,I_j}), \\ I_j(i) + \log(p_{I_j,I_j}), \\ D_j(i) + \log(p_{D_j,I_j}) \end{array}\right\}.$$

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.

Theorem 8.10 *Sei $M = (Q, \Sigma, P, S)$ ein Hidden Markov Modell für ein mehrfaches Sequenzen Alignment der Länge ℓ. Für eine gegebene Sequenz Y lässt sich wahrscheinlichste zugehörige Zustandsfolge in Zeit $O(|Q| \cdot (\ell + |Y|)$ und mit Platz $O(|Q|)$ berechnen.*

Mit Hilfe des obigen Theorems können wir jetzt ein mehrfaches Sequenzen Alignment konstruieren. Wir berechnen zuerst für jede Folge die wahrscheinlichste zugehörige Zustandsfolge durch das Hidden Markov Modells. 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 so genannten Erwartungswert-Maximierungs-Methode schätzen. Aufgrund der speziellen Gestalt des Hidden Markov Modells für mehrfache Sequenzen Alignments ergibt sich für die Vorwärts- und Rückwärtswahrscheinlichkeiten folgende einfachere Rekursionsgleichungen:

$$\begin{aligned}
f_{M_j}(i) &:= \text{Ws(Ausgabe ist } (y_1, \ldots, y_i) \text{ und Endzustand ist } M_j) \\
f_{I_j}(i) &:= \text{Ws(Ausgabe ist } (y_1, \ldots, y_i) \text{ und Endzustand ist } I_j) \\
f_{D_j}(i) &:= \text{Ws(Ausgabe ist } (y_1, \ldots, y_i) \text{ und Endzustand ist } D_j)
\end{aligned}$$

$$\begin{aligned}
b_{M_j}(i) &:= \text{Ws(Ausgabe ist } (y_{i+1}, \ldots, y_{|Y|}) \text{ und Anfangszustand ist } M_j) \\
b_{I_j}(i) &:= \text{Ws(Ausgabe ist } (y_{i+1}, \ldots, y_{|Y|}) \text{ und Anfangszustand ist } I_j) \\
b_{D_j}(i) &:= \text{Ws(Ausgabe ist } (y_{i+1}, \ldots, y_{|Y|}) \text{ und Anfangszustand ist } D_j)
\end{aligned}$$

Mit Hilfe dieser Vorwärts- und Rückwärtswahrscheinlichkeiten können wir wieder mit der Baum-Welch Methode neue, verbesserte Zustandsübergans- und Emissionswahrscheinlichkeiten aus dem bestehenden Modell berechnen. Wir überlassen die Details der Implementierung dem Leser und halten nur noch das Ergebnis fest.

Lemma 8.11 *Die Vorwärts- und Rückwärtswahrscheinlichkeiten für ein Hidden Markov Modell $M = (Q, \Sigma, P, S)$ für ein mehrfaches Sequenzen Alignment der Länge ℓ können in Zeit $O(|Q| \cdot \ell)$ berechnet werden.*

Literaturhinweise

A.1 Lehrbücher zur Vorlesung

Peter Clote, Rolf Backofen: *Introduction to Computational Biology*; John Wiley and Sons, 2000.

Richard Durbin, Sean Eddy, Anders Krogh, Graeme Mitchison: *Biological Sequence Analysis*; Cambridge University Press, 1998.

Dan Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*; Cambridge University Press, 1997.

David W. Mount: *Bioinformatics — Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.

Pavel A. Pevzner: *Computational Molecular Biology - An Algorithmic Approach*; MIT Press, 2000.

João Carlos Setubal, João Meidanis: *Introduction to Computational Molecular Biology*; PWS Publishing Company, 1997.

Michael S. Waterman: *Introduction to Computational Biology: Maps, Sequences, and Genomes*; Chapman and Hall, 1995.

A.2 Skripten anderer Universitäten

Bonnie Berger: *Introduction to Computational Molecular Biology*, Massachusetts Institute of Technology, http://theory.lcs.mit.edu/~bab/01-18.417-home.html;

Bonnie Berger, *Topics in Computational Molecular Biology*, Massachusetts Institute of Technology, Spring 2001, http://theory.lcs.mit.edu/~bab/01-18.418-home.html;

Paul Fischer: *Einführung in die Bioinformatik* Universität Dortmund, Lehrstuhl II, WS2001/2002, http://ls2-www.cs.uni-dortmund.de/lehre/winter200102/bioinf/

Richard Karp, Larry Ruzzo: *Algorithms in Molecular Biology*; CSE 590BI, University of Washington, Winter 1998. http://www.cs.washington.edu/education/courses/590bi/98wi/

Larry Ruzzo: *Computational Biology*, CSE 527, University of Washington, Fall 2001; http://www.cs.washington.edu/education/courses/527/01au/

Georg Schnittger: *Algorithmen der Bioinformatik*, Johann Wolfgang Goethe-Universität Frankfurt am Main, Theoretische Informatik, WS 2000/2001, http://www.thi.informatik.uni-frankfurt.de/BIO/skript2.ps.

Ron Shamir: *Algorithms in Molecular Biology* Tel Aviv University, http://www.math.tau.ac.il/~rshamir/algmb.html; http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html.

Ron Shamir: *Analysis of Gene Expression Data, DNA Chips and Gene Networks*, Tel Aviv University, 2002; http://www.math.tau.ac.il/~rshamir/ge/02/ge02.html;

Martin Tompa: *Computational Biology*, CSE 527, University of Washington, Winter 2000. http://www.cs.washington.edu/education/courses/527/00wi/

A.3 Lehrbücher zu angrenzenden Themen

Teresa K. Attwood, David J. Parry-Smith; *Introduction to Bioinformatics*; Prentice Hall, 1999.

Maxime Crochemore, Wojciech Rytter: *Text Algorithms*; Oxford University Press: New York, Oxford, 1994.

Martin C. Golumbic: *Algorithmic Graph Theory and perfect Graphs*; Academic Press, 1980.

Benjamin Lewin: *Genes*; Oxford University Press, 2000.

Milton B. Ormerod: *Struktur und Eigenschaften chemischer Verbindungen*; Verlag Chemie, 1976.

Hooman H. Rashidi, Lukas K. Bühler: *Grundriss der Bioinformatik — Anwendungen in den Biowissenschaften und der Medizin*,

Klaus Simon: *Effiziente Algorithmen für perfekte Graphen*; Teubner, 1992.

Maxine Singer, Paul Berg: *Gene und Genome*; Spektrum Akademischer Verlag, 2000.

Lubert Stryer: *Biochemie*, Spektrum Akademischer Verlag, 4. Auflage, 1996.

A.4 Originalarbeiten

Kellogg S. Booth, George S. Lueker: Testing for the Consecutive Ones property, Interval Graphs, and Graph Planarity Using PS-Tree Algorithms; *Journal of Computer and System Science*, Vol.13, 335–379, 1976.

Ting Chen, Ming-Yang Kao: On the Informational Asymmetry Between Upper and Lower Bounds for Ultrametric Evolutionary Trees, *Proceedings of the 7th Annual European Symposium on Algorithms, ESA'99*, Lecture Notes in Computer Science 1643, 248–256, Springer-Verlag, 1999.

Richard Cole: Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm; *SIAM Journal on Computing*, Vol. 23, No. 5, 1075–1091, 1994.
s.a. *Technical Report*, Department of Computer Science, Courant Institute for Mathematical Sciences, New York University, TR1990-512, June, 1990, http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2fTR1990-512

Martin Farach, Sampath Kannan, Tandy Warnow: A Robust Model for Finding Optimal Evolutionary Trees, *Algorithmica*, Vol. 13, 155–179, 1995.

Wen-Lian Hsu: PC-Trees vs. PQ-Trees; *Proceedings of the 7th Annual International Conference on Computing and Combinatorics, COCOON 2001*, Lecture Notes in Computer Science 2108, 207–217, Springer-Verlag, 2001.

Wen-Lian Hsu: A Simple Test for the Consecutive Ones Property; *Journal of Algorithms*, Vol.43, No.1, 1–16, 2002.

Haim Kaplan, Ron Shamir: Bounded Degree Interval Sandwich Problems; *Algorithmica*, Vol. 24, 96–104, 1999.

Edward M. McCreight: A Space-Economical Suffix Tree Construction Algorithm; *Journal of the ACM*, Vol. 23, 262–272,1976.

Moritz Maaß: *Suffix Trees and Their Applications*, Ausarbeitung von der Ferienakademie '99, Kurs 2, Bäume: Algorithmik und Kombinatorik, 1999. http://www14.in.tum.de/konferenzen/Ferienakademie99/

Esko Ukkonen: On-Line Construction of Suffix Tress, *Algorithmica*, Vol. 14, 149–260, 1995.

Index

Symbole

A

B

P

Q

R

S

T