Ein graphtheoretischer Ansatz für das *multiple sequence Alignment*-Problem

Bachelorarbeit

vorgelegt von:
Joschka Strüber

Matrikelnummer: 418702

Studiengang: B.Sc. Informatik

Thema gestellt von:
Prof. Dr. Jan Vahrenhold

Arbeit betreut durch:
Prof. Dr. Jan Vahrenhold
Prof. Dr. Xiaoyi Jiang

Münster, 17. April 2018

Inhaltsverzeichnis

1 Einleitung und Motivation

1.1 Multiple Sequence Alignments

1.2 Einsatzgebiete

1.3 Komplexität

2 Dynamische Programmierung

2.1 Trivia

2.2 Das Paradigma

2.3 Der Algorithmus von Needleman und Wunsch

3 DIALIGN

In diesem Kapitel stelle ich zunächst das DIALIGN-Verfahren für multiples Sequenzalignment nach Morgenstern *et al.* (1996) vor. Dabei werde ich alle Anpassungen und Verbesserungen des Verfahrens vorstellen, die bis zur Version 2.2 umgesetzt wurden. Anders als der im letzten Kapitel vorgestellte Algorithmus von Needleman-Wunsch aligniert DIALIGN keine einzelnen Symbole, sondern gleich ganze Segmente der Eingabesequenzen. Das hat die Vorteile, dass man zum einen auf die Kosten zum Einfügen von Lücken verzichten kann und dadurch weitgehend von benutzerdefinierten Eingaben unabhängig wird, und weiterhin ist man so in der Lage sowohl global, als auch lokal verwandte Sequenzen einander auszurichten: Wenn man feststellt, dass in einem Bereich keine Segmente vorliegen, die einander ähnlich sind, dann verzichtet man darauf diese sich gegenseitig zuzuweisen und sie werden nicht Teil des *Alignments*.

DIALIGN kann genau wie Needleman-Wunsch im Sinne der jeweiligen Zielfunktion mathematisch optimale paarweise *Alignments* berechnen. Anders als bei letzterem, kann man aber auch mit Hilfe einer Heuristik effizient multiple Alignments berechnen, die aus drei oder mehr Sequenzen bestehen. Das grobe Vorgehen sieht dabei wie folgt aus:

Algorithmus 1 DIALIGN

Require: Menge S von Sequenzen mit $|S| = n$
1: **procedure** DIALIGN(S)
2: Weise allen möglichen Fragmenten f ein Gewicht $w^*(f)$ zu
3: Berechne mit dynamischer Programmierung alle möglichen $\binom{n}{2}$ paarweisen Alignments aus S
4: Sortiere alle Fragmente der paarweisen Alignments nach ihrem Gewicht als $f_{1,\ldots,n}$
5: $A \leftarrow \emptyset$ ▷ Initialisiere Ausgabe für Alignment
6: **for** i=1, …, n **do**
7: **if** f_i ist zu allen bisher gewählten Fragmenten *konsistent* **then**
8: $A \cup \{f_i\}$ ▷ Füge f_i zum *Alignment* hinzu
9: **end if**
10: **end for**
11: **return** A
12: **end procedure**

Unter *Konsistenz* können wir uns zunächst informell vorstellen, dass es bei einer Zuweisung weder zu Überkreuzungen kommt, noch dazu, dass ein Symbol einer Sequenz gleichzeitig mehreren einer anderen zugewiesen wird.

Möchte ich das lieber hier haben oder zwischen Definition und Beispielen zu Konsistenz

3.1 Theoretische Grundlagen

Um multiple Sequenzalignments genauer zu verstehen und die dazu nötigen Algorithmen analysieren zu können, brauchen wir einige Definitionen. Diese sind Morgenstern

et al. (1996), Abdeddaïm und Morgenstern (2000) und Corel *et al.* (2010) entnommen. Dazu betrachten wir im Folgenden eine n-stellige Menge von Sequenzen S über einem endlichen Alphabet. Dabei gibt L_i die Länge der i-ten Sequenz an.

3.1.1 Definition (Stelle und Stellenraum)
Eine *Stelle* ist ein Tupel (i, p), bei dem i die Sequenz und p die Position eines Zeichens innerhalb dieser Sequenz angibt. Als *Stellenraum* bezeichnen wir die Menge aller Stellen über unseren Sequenzen S: $\mathcal{S} := \{(i, p) | 1 \leq i \leq n, 1 \leq p \leq L_i\}$
Der Einfachheit identifizieren wir die *Stellen* der i-ten Sequenz als $\mathcal{S}_i$. Auf dem *Stellenraum* existiert eine Halbordnung '$\preceq$', wobei $(i, p) \preceq (i', p')$ genau dann gilt, falls $i = i'$ und $p \leq p'$.

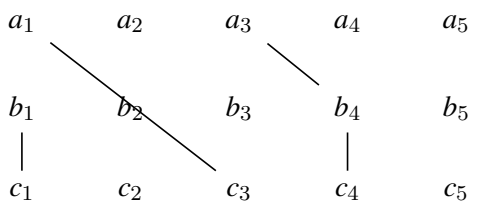
Nachdem wir bis jetzt nur umgangssprachlich mit *Alignments* und *Konsistenz* zu tun hatten, möchte ich diese Begriffe nun formalisieren.

3.1.2 Definition (Alignment und Konsistenz)
Ein *Alignment* $\mathcal{A}$ ist eine Äquivalenzrelation auf der Menge $\mathcal{S}$, die ein bestimmtes *Konsistenzkriterium* erfüllt. Sei zunächst $\mathcal{R}$ eine beliebige binäre Relation auf $\mathcal{S}$. Wir können diese mit '$\preceq$' zu der Präordnung (auch Quasiordnung genannt) $\preceq_{\mathcal{R}=(\preceq \cup \mathcal{R})_t}$ erweitern, also einer zweistelligen Relation, die reflexiv und transitiv, aber nicht antisymmetrisch ist. Hierbei bezeichnet $\mathcal{X}_t$ die transitive Hülle einer Relation $\mathcal{X}$.

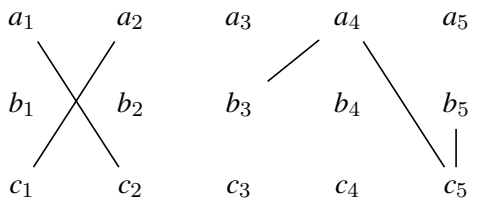
Wir bezeichnen $\mathcal{R}$ als *konsistent*, wenn $\preceq_{\mathcal{R}} = (\preceq \cup \mathcal{R})_t$ die natürliche Ordnung auf jeder Sequenz erhält, also $x \preceq_{\mathcal{R}} y \implies x \preceq y$ für alle $x, y \in S_i \forall 1 \leq i \leq n$ gilt. Außerdem nennen wir eine Menge von Relationen $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ *konsistent*, wenn ihre Vereinigung $\cup_i \mathcal{R}_i$ *konsistent* ist, sowie ein Paar $(x, y) \in \mathcal{S}^2$ *konsistent* mit einer Relation $\mathcal{R}$, falls $\mathcal{R} \cup \{(x, y)\}$ *konsistent* ist.

Für ein Alignment $\mathcal{A}$ und (x, y) gilt $x \mathcal{A} y$ genau dann, wenn die *Stellen* x und y durch $\mathcal{A}$ aligniert werden oder identisch sind.

Im Folgenden wollen wir zwei Beispiele betrachten, um das Konzept der *Konsistenz* und *Alignments* besser zu veranschaulichen. Informell können wir uns ein *Alignment* als eine Relation vorstellen, bei der es weder zu einer Überkreuzung von Zuweisungen kommt, noch zu Fällen, bei denen ein Symbol (transitiv) gleichzeitig mehreren Symbolen aus einer einzigen anderen Sequenz zugewiesen ist.

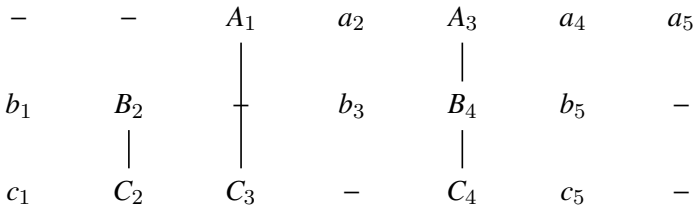


Für alle *Stellen*, die aus der selben kommen Sequenz stammen, gilt $x \preceq_{\mathcal{R}} y \implies x \preceq y$, wie beispielsweise für a_1 und a_5: $a_1 \mathcal{A} c_3, c_3 \preceq c_4, c_4 \mathcal{A} b_4, b_4 \mathcal{A} a_3$ und $a_3 \preceq a_5$. Es folgt $a_1 \preceq_{\mathcal{R}} a_5$. Also ist die Relation auf $\mathcal{S}$ *konsistent* und somit ein *Alignment*.



Hier handelt es sich um kein *Alignment*, denn die *Konsistenz* ist gleich an mehreren Stellen verletzt. Erstens gilt $a_2 \preceq_{\mathcal{R}} a_1$, denn $a_2 \mathcal{A} c_1, c_1 \preceq c_2$ und $c_2 \mathcal{A} c_1$. Da aber $c_1 \preceq c_2$ gilt, erhält die Relation die natürliche Ordnung auf der erstens Sequenz nicht. Der Grund liegt hier an der Überkreuzung von mehreren Zuweisungen. Des Weiteren gilt $b_5 \preceq_{\mathcal{R}} b_3$, weil $b_5 \mathcal{A} c_5, c_5 \mathcal{A} a_4$ und $a_4 \mathcal{A} b_3$, aber $b_5 \not\preceq b_3$. Hier ist das Problem eine transitive Mehrfachzuweisung von mehreren Symbolen der einen Sequenz auf das gleiche einer anderen (sowohl b_3 als auch b_5 stehen in Relation zu beispielsweise a_4).

Es lässt sich zeigen, dass eine Relation $\mathcal{A}$ genau dann ein *Alignment* ist, wenn es möglich ist zwischen den alignierten Symbolen Lücken einzufügen, sodass gerade die einander zugewiesenen untereinander stehen. Deshalb bezeichnet man die Äquivalenzklassen $[x]_{\mathcal{A}} = \{y \in \mathcal{S} \colon x \mathcal{A} y\}$ von $\mathcal{A}$ auch als *(Zuweisungs-)Spalten*. Man kann sich leicht überlegen, dass das bei Überkreuzungen und transitiven Mehrfachzuweisungen nicht möglich ist. Bei unserem ersten Beispiel von oben würde das so aussehen:



Alle Symbole, die Teil einer Zuweisungsspalte sind, also einer Äquivalenzklasse mit mehr als einer *Stelle*, wurden als Großbuchstabe dargestellt, während die unalignierten kleingeschrieben wurden.

Da DIALIGN ein segmentbasiertes Alignmentverfahren ist, brauchen wir noch eine Bezeichnung für eine paarweise, lückenlose Zuweisung von direkt aufeinanderfolgenden Elementen zweier Sequenzen.

3.1.3 Definition (Fragment)
Gegeben seien zwei Sequenzen S_1 und S_2 und ein *Alignment* $\mathcal{A}$ auf diesen Sequenzen. Dann definieren wir das *Fragment* mit Länge l, das an den Stellen i in S_1 und j in S_2 endet mit $1 \leq i \leq l(S_1), 1 \leq j \leq l(S_2)$ und $i - l \geq 0 \leq j - l$, als $f_{i,j,l}$, wenn $S_1[i-k] \mathcal{A} S_2[j-k] \forall\, 0 \leq k \leq l-1$ gilt. Manchmal werden *Fragmente* auch als *diagonals* bezeichnet, weil sie in der Matrix des Needleman-Wunsch-Verfahrens als Diagonale von mehreren aufeinanderfolgenden einander zugeordneten Symbolen stehen würden.

Wir können unter einem *Alignment* auch eine Kette von zueinander *konsistenten Fragmenten* verstehen.

3.2 Gewichtsfunktionen und Substitutionsmatrizen

3.2.1 Gewichtsfunktionen in DIALIGN 1

Um zwei *Fragmente* miteinander vergleichen zu können, müssen wir die Ähnlichkeit zwischen ihnen quantifizieren. Je ähnlicher sich zwei *Fragmente* sind, desto eher können wir davon ausgehen, dass sie einen gemeinsamen evolutionären Ursprung haben und als desto wichtiger schätzen wir sie für unser *Alignment* ein. In der ersten Variante von DIALIGN hat man eine starre stochastische Gewichtsfunktion benutzt, indem man davon ausging, dass alle Symbole gleichverteilt mit Wahrscheinlichkeit $p = 0,25$ für DNA und $p = 0,05$ für Proteine auftreten (Morgenstern *et al.*, 1996). Gegeben sei ein *Fragment* f der Länge l, mit m in beiden Sequenzen übereinstimmenden Symbolen. Dann lautet die Wahrscheinlichkeit, dass ein solches *Fragment* der Länge l m oder mehr Übereinstimmungen hat wie folgt:

$$P(l,m) = \sum_{i=m}^{l} \binom{l}{i} \cdot p^i \cdot (1-p)^{l-i} \tag{3.1}$$

Wie in anderen Disziplinen, wie der Informationstheorie oder statistischen Mechanik benutzen wir als Gewichtsfunktion nun den negativen Logarithmus von $P(l,m)$. Dadurch bekommen wir ein umso höheres Gewicht, je niedriger die Wahrscheinlichkeit ist, dass das vorliegende *Fragment* zufällig entstanden ist. Ziel wird es im Folgenden sein die Summe der Gewichte aller *Fragmente* eines *Alignments* zu maximieren. Diese bezeichnen wir als *Score* des *Alignments*.

$$w(f) \coloneqq -\ln(P(l,m)) \tag{3.2}$$

3.2.2 Substitutionsmatrizen

Es hat sich jedoch herausgestellt, dass diese Gewichtsfunktion nicht immer zielführend ist. Nicht alle Aminosäuren sind gleich ähnlich und die Übergangswahrscheinlichkeiten zwischen ihnen können dramatisch verschieden sein. So ist beispielsweise eine Veränderung von Arginin zu Lysin recht wahrscheinlich, während jene von Tryptophan zu Glycin nur sehr selten vorkommt (Pearson, 2013).

Deswegen verwenden wir genau wie bei Needleman-Wunsch Substitutionsmatrizen, wie beispielsweise BLOSUM62, um die Ähnlichkeit zwischen zwei *Fragmenten* zu berechnen. Sei dazu $f_{i,j,l}$ ein *Fragment* aus den zwei Sequenzen S_1 und S_2 und M eine Substitutionsmatrix. Dann berechnet folgende Formel das Gewicht von $f_{i,j,l}$:

$$w(f_{i,j,l}) \coloneqq \sum_{k=1}^{l} M[i-l+k, j-l+k] \tag{3.3}$$

Dieses Vorgehen hat einige Vorteile gegenüber der alten Gewichtsberechnung. Zum einen kann man das Gewicht eines *Fragments* $f_{i,j,l}$ sehr einfach berechnen, wenn man das Gewicht des *Fragments* $f_{i-1,j-1,l-1}$ bereits kennt, indem man einen einzigen Ähnlichkeitswert zur Summe hinzu addiert. Zum anderen kann man die Berechnung vieler Gewichte frühzeitig abbrechen und zwar, wenn eine Teilsumme der Ähnlichkeitswerte negativ ist. Dann weiß man, dass ein *Alignment* mit höherem *Score* berechnen werden kann, wenn man diesen Teil des Fragments weglässt. Diese beiden Eigenschaften werden wir

uns im nächsten Abschnitt über die effiziente Berechnung der paarweisen *Alignments* zunutze machen.

Nun wollen wir Substitutionsmatrizen und die Theorie dahinter genauer betrachten. Das werden wir anhand der von Henikoff und Henikoff (1992) entwickelten BLOcks Substitution Matrix (BLOSUM) tun, da andere verbreitete Substitutionsmatrizen ähnlich entstanden sind. Die Matrizen wurden empirisch bestimmt, indem man sich Blöcke von Proteinmotiven anguckte, bei denen ein korrektes *Alignment* bekannt war. Als *Block* bezeichnen wir einen längeren, zusammenhängenden alignierten Bereich ohne gelöschte oder eingefügte Segmente. Für die Berechnung eines Eintrags der Matrix $M_{i,j}$ brauchen wir die Wahrscheinlichkeit, mit der die beiden Aminosäuren auftreten q_i und q_j, sowie die Wahrscheinlichkeit, dass gerade diese beide Aminosäuren miteinander aligniert werden $p_{i,j}$.

$$M_{i,j} := \frac{1}{\lambda} \log\left(\frac{p_{i,j}}{q_i \cdot q_j}\right) \tag{3.4}$$

Der Korrekturterm λ wird benutzt, um die Werte auf ganze Zahlen zu runden, die weniger anfällig für Rundungsfehler und andere Ungenauigkeiten in der Computerarithmetik sind. Diese Vorgehensweise wird, da man den Logarithmus einer Wahrscheinlichkeit berechnet, als *log-odd*-Verfahren bezeichnet. Der Eintrag $M_{i,j}$ gibt ein Maß für die Wahrscheinlichkeit an, dass das betrachtete Paar in einem *Alignment* aus genau diesen beiden Aminosäuren auftritt und die Wahrscheinlichkeit für eine längere Folge aufeinanderfolgender Paare wird mit der Summe der Einträge berechnet. Das funktioniert aufgrund der Rechenregeln des Logarithmus: $\log(p_1 \cdot p_2) = \log(p_1) + \log(p_2)$. Möchte man die ursprünglichen Wahrscheinlichkeiten berechnen, muss man lediglich die Summe der Ähnlichkeitswerte exponentieren.

Henikoff und Henikoff (1992) haben mehrere Substitutionsmatrizen entwickelt. Die Zahl hinter jeder BLOSUM gibt die Ähnlichkeit der zur Berechnung der Matrix verwendeten Proteinsequenzen an. Für die BLOSUM62 wurden beispielsweise nur Blöcke benutzt, bei denen es eine Ähnlichkeit von höchstens 62% gab. Im Allgemeinen wird dazu geraten BLOSUMs mit geringen Suffixen wie beispielsweise BLOSUM45 zum alignieren von entfernt verwandten, mit großen wie BLOSUM80 für eng verwandte und BLOSUM62 für durchschnittlich eng verwandte Sequenzen zu benutzen.

BLOSUM62

BLOSUM62 IM ANHANG???

Bei DNA wird meistens nur eine simple Unterscheidung zwischen Treffern und Nichttreffern gemacht. Als Substitutionsmatrix entspräche dies der Einheitsmatrix. Dies hat aber die Nachteile, dass alle *Fragmente* positive Gewichte haben und damit potentiell für unser *Alignment* in Betracht kommen. Besser sind positive Werte für ähnliche und negative für sehr unähnliche Abschnitte, weil sich so der Rechenaufwand verringern lässt. Außerdem kann man mit Matrizen, die dem Einsatzgebiet angepasst sind, oft bessere Ergebnisse erzielen. Nach Pearson (2013) sind die Ähnlichkeiten zwischen zu vergleichenden DNA-Sequenzen deutlich größer, als bei Proteinen. Sie betragen zwischen homologen menschlichen DNA-Abschnitten etwa 99,9% und bei proteinkodierenden Regionen zwischen Mensch und Maus immer noch 80%, während Ähnlichkeiten von unter 50%, anders als bei Proteinen, quasi nicht mehr zu entdecken sind. Dementsprechend können +1/-3 für Treffer und Nichttreffer bei 99%, +2/-3 bei 90% und +5/-4 bei 70% Übereinstimmung benutzt werden.

Ich muss jedoch zugeben, dass die Wahl der richtigen Substitutionsmatrix ein bisschen

dem Henne-Ei-Problem ähnelt: um die Ähnlichkeit von zwei Sequenzen zu bestimmen, müssen wir sie mit der passenden Matrix alignieren. Für die Wahl dieser Matrix sollten wir jedoch wissen, wie ähnlich sich die beiden Sequenzen sind.

Wann genau sich die Berechnung der Gewichte in DIALIGN verändert hat, steht leider in keiner Veröffentlichung, auch wenn sie bereits in Morgenstern *et al.* (1996) als kommende Ergänzung in Betracht gezogen wurde. Spätestens in DIALIGN TX, der neuesten Version des Programms wie wir sie auf der Website der Göttinger Bioinformatik finden (Subramanian *et al.*, 2008), wird diese Technik jedoch angewendet. Dort dient eine modifizierte BLOSUM 62, die nur nichtnegative Werte enthält, als Matrix für Proteinsequenzen, während bei DNA lediglich die Einheitsmatrix benutzt wird.

3.2.3 Gewichtsfunktionen in DIALIGN 2

In der ursprünglichen Version von DIALIGN gab es noch einen benutzerdefinierten Parameter T, der das minimale Gewicht eines in Betracht zu nehmenden *Fragments* angab. Dieser wurde eingeführt, damit nicht kleine, zufällige Übereinstimmungen ihren Weg in das *Alignment* finden. Denn für ein gutes *Alignment* ist es genauso wichtig, dass nicht miteinander verwandte Abschnitte einander auch nicht zugewiesen werden, wie es wichtig ist, dass dies bei verwandten getan wird. Bei Tests mit DIALIGN 1 hat man jedoch festgestellt, dass ein Großteil der ausgewählten *Fragmente* nur knapp über der Gewichtsgrenze T lagen und wenn man diese senkte, sank das Gewicht der *Fragmente* auch (Morgenstern *et al.*, 1998).

Das liegt daran, dass die Gewichtsfunktion w einem langen *Fragment* f quasi das gleiche Gewicht zuordnet, wie die Summe der Gewichte der Teilfragmente $f_1, \ldots, f_n$, wenn man f in diese teilt. Das sorgt dafür, dass man oft bessere *Scores* erhält, wenn man größere Fragmente aufteilt und dazwischen einzelne Regionen mit geringen Übereinstimmungen weglässt, statt große *Fragmente* auszuwählen. Neben der Abhängigkeit vom willkürlichen Parameter T und der Tendenz kleine, unbedeutende Übereinstimmungen auszuwählen, hat dies auch den Nachteil, dass die rechenintensive Aktualisierung der Konsistenzgrenzen öfter durchgeführt werden muss.

Deshalb ist man in DIALIGN 2 dazu übergegangen statt der Wahrscheinlichkeit $P(l, m)$, dass in einem *Fragment* der Länge l mindestens m Übereinstimmungen auftreten, zu berechnen, wie wahrscheinlich es ist, dass in den beiden Gesamtsequenzen S_1 und S_2 mit Längen l_1 respektive l_2 überhaupt eine Sequenz mit Länge l und m Übereinstimmungen auftritt.

$$P^*(l, m) \approx l_1 \cdot l_2 \cdot P(l, m) \tag{3.5}$$

Als neue Gewichtsfunktion w^* ergibt sich dann mit $K := \log(l_1) + \log(l_2)$:

$$w^*(f) := w(f) - K \tag{3.6}$$

Wenn man f nun in $f_1, \ldots f_n$ aufteilt, wird der Korrekturterm K nicht nur einmal, sondern n-mal abgezogen. Das sorgt dafür, dass tendentiell längere *Fragmente* ausgewählt werden (Morgenstern, 1999). Ein weiterer Vorteil ist, dass der Erwartungswert des Gewichts eines zufälligen *Fragments* nicht mehr 1, sondern 0 ist. Dadurch haben alle Abschnitte mit unterdurchschnittlicher Ähnlichkeit automatisch negative Gewichte und wir haben eine einfache und schnelle Möglichkeit zu entscheiden, ob ein *Fragment* weiter für unser *Alignment* in Betracht gezogen werden muss.

Wie sich der Effekt von w^* auswirkt, wenn man eine Substitutionsmatrix benutzt, die einem zufälligen *Fragment* im Schnitt ein negatives Gewicht zuordnet, werden wir später nach der Programmierung empirisch feststellen. Möglicherweise ist es dann besser auf ihn zu verzichten. Mit einer (+2/-3)-Matrix haben wir beispielsweise einen Erwartungswert von $E(w(f_{i,j,l})) = \left(\frac{3}{4} \cdot (-3) + \frac{1}{4} \cdot 2\right) \cdot l = -\frac{7}{4} \cdot l$ für ein zufälliges DNA-*Fragment* der Länge l. Ein anderer Ansatz verbindet die Substitutionsmatrix mit dem Korrekturterm K, indem wir wie DIALIGN TX eine Substitutionsmatrix benutzen, die aber keine negativen Werte enthält. Dafür ziehen wir aber weiterhin K vom Gewicht ab.

3.3 Paarweise Alignments mit dynamischer Programmierung

Nachdem wir uns jetzt genauer mit den Gewichten von Fragmenten beschäftigt haben, können wir uns der Berechnung der paarweisen *Alignments* mit Hilfe von dynamischer Programmierung widmen. Dabei beziehe ich mich, außer wenn anders gekennzeichnet, auf die speichereffiziente Umsetzung aus DIALIGN 2.2, die in Morgenstern (2002) vorgestellt wurde.

Wie bei dynamischer Programmierung üblich, stellen wir zunächst eine Rekursionsgleichung auf. Sei dazu $Sc[i, j]$ der maximal mögliche *Score* aller *Fragmente* bis zu den Elementen $S_1[i]$ und $S_2[j]$ zweier Sequenzen S_1 und S_2. An dieser Stelle tritt sehr ähnlich zu Needleman-Wunsch eine von drei Situationen auf: Die ersten beiden Möglichkeiten sind, dass wir die *Stelle* $(1, i)$ oder die *Stelle* $(2, j)$ nicht zu unserem *Alignment* hinzufügen. Oder aber wir wählen ein *Fragment* $f_{i,j,l}$ aus, das in (i, j) endet. In diesem Fall wählen wir genau das aus, welches den *Score* aller in (i, j) endenden *Alignments* maximiert. Welcher der drei Fälle der richtige ist, um den höchstmöglichen *Score* bis (i, j) zu berechen, erfahren wir, indem wir das Maximum über sie berechnen.

$$Sc[i, j] = \max \begin{cases} Sc[i-1, j], \\ Sc[i, j-1], \\ \max_{l \geq 1}\{Sc[i-l, j-l] + w^*(f_{i,j,l})\} \end{cases} \tag{3.7}$$

3.3.1 Satz
Mit der obigen Rekursionsgleichung lässt sich ein optimales paarweises *Alignment* zweier Sequenzen mit Längen L_1 und L_2 in $O(L^3)$ Zeit und $O(L^2)$ Speicherplatz berechnen für $L = \max(L_1, L_2)$. Außerdem gilt für die Menge der möglichen Fragmente $F : |F| \in O(L^3)$.

Beweis. Insgesamt müssen wir $L_1 \cdot L_2 \in O(L^2)$-viele Tabelleneinträge berechnen, die wir im Allgemeinen auch gleichzeitig im Speicher vorhalten. Für jeden zu berechnenden Eintrag $Sc[i, j]$ brauchen wir Zugriffe auf $(\min(i, j) + 2)$-viele Einträge in der Matrix und müssen $\min(i, j)$ Gewichte neu berechnen. Dabei dominiert die Berechnung der Gewichte, wobei jedes Gewicht nur genau einmal berechnet werden muss (für den Score des Tabelleneintrags, in dem das *Fragment* endet). Im schlimmsten Fall gilt $L_1 = L_2$. Dann gibt es *Fragmente* der Länge 1 mit jeweils L möglichen Endpunkten in S_1 und S_2, der Länge zwei mit jeweils $L - 1$ möglichen Endpunkten und so weiter. Die Anzahl aller *Fragmente* $|F| = \sum_{k=0}^{L-1}(L-k)^2 = \frac{1}{6} \cdot L(2L^2 + 3L + 1) \in O(L^3)$ und die naiv berechnete

Anzahl der Zugriffe ist $\sum_{k=0}^{L-1}(L-k)^2 \cdot k = \frac{1}{12} \cdot (L-1)L^2(L+1) \in O(L^4)$. Glücklicherweise kann man das Gewicht jedes Fragments $f_{i,j,l}$ in $O(1)$ Zeit aus $f_{i,j,l-1}$ berechnen, denn $w^*(f_{i,j,l}) = w^*(f_{i,j,l-1}) + M[i–l+1, j–l+1]$, wodurch sich die Laufzeit auf $O(L^3)$ verkleinern lässt. □

Um nicht nur den *Score* eines perfekten paarweisen *Alignments* berechenen zu können, sondern auch dieses *Alignment* selbst, müssen wir zunächst noch einige Definitionen einführen. Zunächst definieren wir für ein *Fragment* $f \in F$ das *Präfixgewicht* $W(f)$, das die maximale Summe der Gewichte einer Kette von *Fragmenten* bezeichnet, die mit f endet.

$$W(f) := \max \left\{ \sum_{k=0}^{M} w^*(f_k) : f_1 \ll \ldots \ll f_M = f \right\} \tag{3.8}$$

3.3.2 Definition (Vorgänger)
Sei $f_1 \ll \ldots \ll f_M$ eine Kette von *Fragmente*, die das Maximum der vorherigen Gleichung erreicht. Dann bezeichnen wir $P(f) = f_{M-1}$ als den *Vorgänger* von f. Außerdem sei $Pr[i, j]$ das letzte *Fragment* einer optimalen Kette, die spätestens in (i, j) endet.

Jetzt können wir für ein *Fragment* $f \in F$, das in (i, j) startet, das *Gesamtgewicht* und den *Vorgänger* genau definieren. Das *Präfixgewicht* ist genau das Gewicht von f addiert mit dem *Score* der *Fragmente*, die vor f stehen. $P(f)$ und $Pr[i, j]$ sind zwar strenggenommen nicht wohldefiniert und es könnte mehrere *Fragmente* mit diesen Eigenschaften geben. Wie auch schon in Morgenstern *et al.* (1996) wählen wir dann das in den Sequenzen am weitesten rechts stehende aus.

$$W(f) = Sc[i-1, j-1] + w^*(f) \tag{3.9}$$

Der *Vorgänger* von f ist das letzte Element einer Kette von *Fragmente*, die vor f enden.

$$P(f) = Pr[i-1, j-1] \tag{3.10}$$

Damit können wir jetzt (3.7) mit unseren neuen Definitionen umformulieren, denn der dritte Fall der obigen Gleichung ist genau das maximale *Präfixgewicht* eines *Fragments*, das in (i, j) endet.

$$Sc[i, j] = \max \begin{cases} Sc[i-1, j], \\ Sc[i, j-1], \\ \max W(f) : f \text{ endet in } (i, j) \end{cases} \tag{3.11}$$

Analog zu den Fällen von $W(f)$ können wir jetzt auch $Pr[i, j]$ setzen. Das letzte *Fragment* einer optimalen Kette bis (i, j) ist das selbe wie bei $(i-1, j)$ beziehungsweise $(i, j-1)$, wenn diese in keinem dieser beiden Stellenpaare endet. Endet sie hingegen in (i, j), dann ist das gesuchte *Fragment* das, welches das *Präfixgewicht* aller in (i, j) endenden *Fragmente* maximiert.

$$Pr[i, j] = \begin{cases} Sc[i-1, j], & \text{if } Sc[i, j] = Sc[i-1, j] \\ Sc[i, j-1], & \text{if } Sc[i, j] = Sc[i, j-1] \\ \hat{f}, & \text{if } Sc[i, j] = \max\{W(f) : f \text{ endet in } (i, j)\} \end{cases} \tag{3.12}$$

Hier gilt $\hat{f} = argmax\{W(f) : f \text{ endet in } (i,j)\}$. Jetzt stehen uns alle Informationen zur Verfügung, um neben dem *Score* einer optimalen Kette von *Fragmenten* auch diese selbst zu berechnen. Zunächst sei $f_{\max} = argmax_{f \in F}(W(f))$ das letzte Element dieser Kette. Man erhält es, indem man sich das letzte Element einer optimalen Kette anguckt, die bis ganz ans Ende von S_1 und S_2 reichen kann: $f_{\max} = Pr[L_1, L_2]$. Mit einem Backtrackingalgorithmus sind wir nun in der Lage das optimale paarweise *Alignment* zu berechnen, indem wir mit $f_{\max}$ starten und immer den direkten *Vorgänger* des aktuellen *Fragments* auswählen.

$$f_0 = f_{\max} \text{ und } f_{k+1} = P(f_k) \tag{3.13}$$

3.3.1 Speichereffiziente Berechnung der paarweisen *Alignments*

In diesem Abschnitt beschäftigen wir uns mit einer sehr speichereffizienten und schnellen Implementierung des soeben gesehenen Ansatzes. Zunächst beschränken wir die maximale Länge eines *Fragments* $l_{\max}$ auf eine kleine, feste Zahl, beispielsweise 40. Je nach gewünschter Genauigkeit und benötigter Geschwindigkeit kann man diesen Wert vergrößern oder verkleinern. Auch wenn diese Einschränkung den maximal zu erreichenden *Score* senkt und wir daher keine perfekten *Alignments* mehr berechnen, hat $l_{\max}$ in der Praxis kaum einen Einfluss auf die Güte der Ergebnisse. Das liegt daran, dass wir im Fall von geringen Ähnlichkeiten zwischen Sequenzen nur selten *Fragmente* mit Längen haben, die $l_{\{max\}}$ überschreiten und im Fall von sehr ähnlichen Sequenzen können wir lange *Fragmente* auch in mehrere kleinere in der Größenordnung unserer Begrenzung aufteilen. Wir werden zeigen, dass mit dieser Einschränkung ein paarweises *Alignment* in $O(L^2)$ Zeit und $O(L + N_{\max})$ Speicherplatz berechnet werden kann, wobei $N_{\max}$ die Anzahl an gleichzeitig gespeicherten *Fragmenten* ist (Morgenstern, 2002), die durch $|F|$ begrenzt wird.

Wir gehen unsere *Score*matrix Spalte für Spalte von links nach rechts durch. An jeder Position (i, j) berechnen wir mit 3.9 und 3.10 $W(f)$ und $P(f)$ für alle *Fragmente* $f \in \{f_{i+k,j+k,k} : 1 \leq k \leq l_{\max}\}$, die an der Stelle (i, j) beginnen. Dabei speichern wir Pointer auf $W(f)$ und $P(f)$ in den Listen F_{j+k}, die mit der Spalte $j + k$ assoziiert werden in denen die jeweiligen *Fragmente enden*. Alles was wir dafür an Informationen benötigen sind $Sc[i-1, j-1]$ und $Pr[i-1, j-1]$. Deshalb müssen wir nicht permanent die ganze Matrix vorhalten, sondern benötigen nur die zuletzt berechnete und die aktuelle Spalte für Sc und Pr, also vier eindimensionale Arrays der Länge L_1.

Bevor wir zur $(j+1)$-ten Spalte übergehen, berechnen wir alle Einträge von 1 bis i für die j-te Spalte. Dazu greifen wir auf die Werte der vorhergehenden Spalte $(j-1)$ und auf die zuvor gespeicherten Listen aller *Fragmente* F_j zu , die in der j-ten Spalte enden, wobei wir die Formeln 3.11 und 3.12 benutzen. Man kann sich überlegen, dass für jeden Eintrag (i, j) höchstens $l_{\max} \in O(1)$ *Fragmente* gespeichert wurden. Sobald wir mit der Berechnung der $j - ten$ Spalte fertig sind, können wir die Werte von $Pr[i, j-1]$ und $Sc[i, j-1]$ für $1 \leq i \leq L_1$ löschen.

Diesen Vorgang wiederholen wir, bis wir schlussendlich auch alle Werte der letzten, also L_2-ten, Spalte berechnet haben. Dann kennen wir mit $Sc[L_1, L_2]$ den *Score* des paarweisen *Alignments* und können mithilfe der Backtrackingprozedur 3.13 die *Fragmente* aus denen es besteht bestimmen. Dazu brauchen wir die Mengen F_j, deren Einträge aber glücklicherweise nicht alle dauerhaft gespeichert werden müssen. Sobald $Sc[i, j]$ und

$Pr[i, j]$ für eine Position $i, j)$ berechnet wurden, können wir alle *Fragmente*, die dort enden, löschen, abgesehen von $Pr[i, j]$, für das immer noch in Frage kommt, dass es Teil der optimalen Kette von *Fragmenten* ist. Sollte $Pr[i, j]$ nicht in (i, j) enden, können wir sogar alle Einträge aus F_j löschen, die in Zeile i enden.

Algorithmus 2 Speichereffizientes paarweises DIALIGN

Require: Zwei Sequenzen S_1 und S_2 mit den Längen L_1 und L_2

procedure PAIRWISEALIGNMENT(S_1, S_2, l_max)
  for $i \leftarrow 0$ **do**L_1
    $Sc[i, 0] \leftarrow 0$
  end for
  for $j \leftarrow 1$ **to** L_2 **do**
    for $i \leftarrow 1$ **to** L_1 **do**
      for $l \leftarrow 1$ **to** $l_{\max}$ **do**
        $W(f_{i+l,j+l,l}) \leftarrow w^*(f_{i+l,j+l,l}) + Sc[i-1, j-1]$
        $P(f_{i+l,j+l,l}) \leftarrow Pr[i-1, j-1]$
        $F_{j+l} \leftarrow F_{j+l} \cup f_{i+l.j+l,l}$ ▷ Speichere *Fragment* für Spalte in der es endet
      end for
      $$Sc[i, j] = \max \begin{cases} Sc[i-1, j], \\ Sc[i, j-1], \\ \max W(f) : f \text{ endet in } (i, j) \end{cases}$$
      Setze $Pr[i, j]$ analog zu $Sc[i, j]$
      lösche $Sc[i, j-1]$ und $Pr[i, j-1]$ ▷ Lösche alte Spalteneinträge
      for all $f_{i,j,k} \in F_j$ mit $f \neq Pr[i, j]$ **do**
        lösche $F_{i,j,k}$ ▷ Lösche die, die in keiner opt. Kette in (i, j) enden
      end for
    end for
  end for
  $f_0 \leftarrow Pr[L_1, L_2]$
  while $f_k \neq$ **NIL do** ▷ Backtracking, um *Alignment* zu bestimmen
    $f_{k+1} \leftarrow P(f_k)$
    $k \leftarrow k + 1$
  end while
end procedure

Da wir wissen, dass nur *Fragmente* mit positiven Gewichten Teil unseres *Alignments* sein können, sind wir in der Lage die Mengen F_j von gespeicherten *Fragmenten* weiter einzuschränken. Wenn die Teilsumme von Ähnlichkeitswerten bis zu einem bestimmten Punkt negativ, können wir den Durchlauf von 2.7 sofort abbrechen, weil wir wissen, dass wir ein besseres *Alignment* finden, wenn wir den Teil mit der negativen Summe von Gewichten ignorieren. Außerdem gibt es zwei Situationen bei denen wir die Berechnung der Gewichte für Fragmente zwar nicht abbrechen, aber wissen, dass das aktuell betrachtete Element nicht gespeichert werden muss:

- Bei negativem Gewicht. Es kann beispielsweise sein, dass $w(f)$ zwar positiv ist, aber $w^* = w(f) - K < 0$ gilt. Dann kann dieses *Fragment* den Score zwar nicht erhöhen, aber vielleicht ist es Teil eines größeren, das Teil des finalen *Alignments* sein kann.

- Wenn das Gewicht kleiner ist, als das größte bisher gefundene eines *Fragments*, das in (i, j) startet. In diesem Fall wissen wir, dass ersteres auf jeden Fall ein besseres *Alignment* liefern würde.

- Bei DNA: Wenn das Residuenpaar direkt hinter dem Ende des aktuellen *Fragments* einen positiven Ähnlichkeitswert hat. Das bedeutet, dass dieses auf jeden Fall bessere Ergebnisse liefert und wir das aktuelle nicht speichern müssen. Im Programm können wir dies so umsetzen, dass wir das Fragment $f_{i+l,j+l,l}$ erst dann zu F_{j+l} hinzufügen, wenn wir im nächsten Durchlauf der Schleife 2.7 keins mit einem größeren Gewicht finden. Auf diese Art und Weise suchen wir quasi nach lokalen Maxima der Fragmentgewichte und speichern nur diese. Bei Proteinsequenzen funktioniert dieses Vorgehen nicht, weil es sein könnte, dass es für eins der beiden Symbole weiter hinten in der jeweils anderen Sequenz einen besseren Partner gibt. In diesem Fall brauchen wir aber das andere

Guckt man sich 2.12 genauer an, stellt man fest, dass man gar nicht alle *Fragmente* kennen muss, die in (i, j) enden. Es reicht das zu kennen, welches das *Präfixgewicht* $W(f)$ aller dort endenden Ketten maximiert. Anstatt alle dieser *Fragmente* in F_j zu speichern reicht es zu überprüfen, ob der dritte Fall von 2.12 eintritt und erst dann in 2.10 zu sichern. Das bedeutet, dass wir keine ganze Liste von *Fragmenten* für jede Stelle unserer Tabelle speichern müssen, sondern nur ein einziges.

Widmen wir uns nun $N_{\max}$, der Anzahl an *Fragmenten*, die maximal gleichzeitig gespeichert werden. Die Anzahl an gesicherten *Fragmenten*, die wir noch nicht für $Sc[i, j]$ betrachtet haben, beträgt $l_{\max} \cdot L_1 \in O(L)$, weil wir für jede der nächsten $l_{\max}$ Spalten und dort jede der L_1-vielen Zeilen das *Fragment* speichern, das $W(f)$ für alle dort endenden maximiert. Zusätzlich wird die Reihe von *Vorgängern* für jeden aktuellen Spalteneintrag gesichert, indem wir in $Pr[i, j]$ einen Pointer auf das letzte *Fragment* einer optimalen Kette für die Teilsequenzen bis zu den Stellen i und j in den beiden Sequenzen speichern. Dieses wiederum speichert einen Pointer auf seinen eigenen *Vorgänger* und so weiter. Im schlimmsten Fall befinden wir uns in der letzten Spalte der Tabelle und die in den Einträgen endenden optimalen Ketten sind alle unabhängig voneinander. Dann kann es sein, dass diese jeweils aus $O(L_2)$ nah aufeinanderfolgenden *Fragmenten* der Länge $O(1)$ bestehen. In diesem Fall ist $N_{\max} \in O(L^2)$, genau wie der insgesamt benötigte Speicherplatz. In der Praxis kann man aber erwarten, dass $N_{\max}$ deutlich kleiner ist.

Morgenstern (2002) hat sein Verfahren mit verschiedenen Sequenzen getestet. Dabei hat er festgestellt, dass $N_{\max}$ für unabhängige zufällig erstelle Sequenzen im Vergleich zur Größe L zu vernachlässigen ist. Und selbst wenn sehr ähnliche Sequenzen miteinander aligniert wurden, befand sich $N_{\max}$ in der Größenordnung von $L \cdot l_{\max}$. So gesehen bietet dieser Ansatz einen großen Vorteil gegenüber der naiven Umsetzung der Rekursionsformel für paarweise *Alignments*.

3.3.2 Laufzeit

3.3.3 Satz
Ein paarweises optimales *Alignment* zwischen zwei Sequenzen S_1 und S_2 mit Längen L_1 und L_2, gegeben eine maximale Fragmentlänge $l_{\max}$, lässt sich in $O(L^2)$ Zeit berechnen für $L = \max\{L_1, L_2\}$.

Beweis. Das Allokieren des Speicherplatzes für die vier Tabellenspalten (je zwei für $Sc[i,j]$ und $Pr[i,j]$) und das initialisieren der ersten Spalten benötigt $O(L)$ Zeit. Das Berechnen *Vorgänger* und *Präfixgewichte*, sowie das Speichern in $F_{j'}$, der in (i,j) startenden *Fragmente* benötigt jeweils $O(1)$ Zeit, da ihre Länge durch $l_{\max}$ beschränkt ist. $Sc[i,j]$ und $Pr[i,j]$ lassen sich auch in konstanter Zeit berechnen, da wir nur das Maximum von drei Werten bestimmen müssen. Sollte nicht das *Fragment* gewählt werden, welches das *Präfixgewicht* aller in (i,j) endenden *Fragmente* maximiert, löschen wir diesen einzelnen Eintrag in $O(1)$ Zeit. Dies wird für jeden möglichen der $L_1 \cdot L_2 \in O(L^2)$ Tabelleneinträge berechnet, was auch die Laufzeit der geschachtelten *for*-Schleifen ist. Der Backtrackingprozess zur Berechnung des optimalen *Alignments* ist in $O(L)$ Zeit möglich, da wir lediglich der Pointerkette von *Vorgänger* zu *Vorgänger* folgen müssen, bis wir am Anfang der Sequenzen angelangt sind. Es folgt die behauptete Laufzeit von $O(L^2)$. □

Da wir *Alignments* zwischen allen $\binom{n}{2} \in O(n^2)$-vielen Paaren mit jeweils $O(L^2)$ Laufzeit berechnen müssen, kommen wir für die paarweisen *Alignments* insgesamt auf eine Laufzeit von $O(n^2 \cdot L^2)$

3.3.3 Beispiel zur Berechnung paarweiser *Alignments*

Um das Verfahren, das diese Bachelorarbeit behandelt, genauer zu verstehen, widmen wir uns jetzt einem Beispiel mit vier DNA-Sequenzen. Zu diesen werden wir im Lauf der Kapitel immer wieder zurückkehren und an ihnen die verschiedenen Schritte unseres Algorithmus der Reihe nach durchführen.

1. `ADGTCTCA`
2. `GTCADCTCA`
3. `TATCADGG`
4. `DGTCADATC`

Als erstes berechnen wir nach dem oben beschriebenen Algorithmus ein paarweises *Alignment* zwischen den ersten beiden Sequenzen.

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	0	1	0	1	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	"	"	"	"	$F_2[4] = \{f_{4,2,2}, W(f) = 2, P(f) = \text{NIL}\}$
3	"	"	"	"	$F_3[5] = \{f_{5,3,3}, W(f) = 5, P(f) = \text{NIL}\}$
4	"	"	"	"	
5	"	"	"	"	
6	"	"	"	"	
7	"	"	"	"	
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	1	2	1	2	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	"	"	"	"	
3	"	"	"	"	
4	"	2	"	$f_{4,2,2}$	$F_3[5]$ wird nicht aktualisiert, da akt. *Fragment* größeres *Präfixgewicht* hat
5	"	"	"	"	
6	"	"	"	"	$F_4[8] = \{f_{4,8,3}, W(f = 5, P(f) = \text{NIL})\}$
7	"	"	"	"	
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	2	3	2	3	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	"	"	"	"	
3	"	"	"	"	
4	2	2	$f_{4,2,2}$	$f_{4,2,2}$	
5	"	5	"	$f_{5,3,3}$	
6	"	"	"	"	
7	"	"	"	"	$F_4[8]$ wird nicht aktualisiert
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	3	4	3	4	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	$F_5[2] = \{f_{2,5,2}, W(f) = 2, P(f) = \text{NIL}\}$, $F_7[4] = \{f_{4,7,4}, W(f) = 4, P(f) = \text{NIL}\}$, $F_8[5] = \{f_{5,8,5}, W(f) = 7, P(f) = \text{NIL}\}$
1	"	"	"	"	
2	"	"	"	"	
3	"	"	"	"	
4	2	2	$f_{4,2,2}$	$f_{4,2,2}$	
5	5	5	$f_{5,3,3}$	$f_{5,3,3}$	
6	"	"	"	"	
7	"	"	"	"	
8	"	"	"	"	$f_{5,3,3}$ statt $f_{8,4,3}$, wähle vorderes *Fragment* bei Gleichstand; lösche $f_{8,4,3}$ aus $F_4[8]$

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	4	5	4	5	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	$F_7[4]$ und $F_8[5]$ werden nicht aktualisiert
2	"	2	"	$f_{2,5,2}$	
3	"	"	"	"	
4	2	"	$f_{4,2,2}$	$f_{4,2,2}$	
5	5	5	$f_{5,3,3}$	$f_{5,3,3}$	bevorzuge $(i, j-1)$ gegenüber $(i-1, j)$
6	"	"	"	"	
7	"	"	"	"	
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	5	6	5	6	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	2	2	$f_{2,5,2}$	$f_{2,5,2}$	
3	"	"	"	"	
4	"	"	$f_{4,2,2}$	$f_{4,2,2}$	
5	5	5	$f_{5,3,3}$	$f_{5,3,3}$	$F_7[6] = \{f_{6,7,2}, W(f) = 4, P(f) = f_{4,2,2}\}$, $F_8[7] = \{f_{7,8,3}, W(f) = 7, P(f) = f_{4,2,2}\}$ $F_9[8] = \{f_{9,8,4}, W(f) = 10, P(f) = f_{4,2,2}\}$
6	"	"	"	"	
7	"	"	"	"	
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	6	7	6	7	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	2	2	$f_{2,5,2}$	$f_{2,5,2}$	
3	"	"	"	"	
4	2	4	$f_{4,2,2}$	$f_{4,7,4}$	
5	5	5	$f_{5,3,3}$	$f_{5,3,3}$	$F_8[7]$ und $F_9[8]$ nicht aktualisiert; *Score* erreicht, aber nicht übertroffen
6	"	"	"	"	
7	"	"	"	"	
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	7	8	7	8	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	2	2	$f_{2,5,2}$	$f_{2,5,2}$	
3	"	"	"	"	
4	4	4	$f_{4,7,4}$	$f_{4,7,4}$	
5	5	7	$f_{5,3,3}$	$f_{5,8,5}$	$F_8[7]$ und $F_9[8]$ nicht aktualisiert; *Score* zwar erreicht, aber nicht übertroffen
6	"	"	"	"	
7	"	"	"	"	lösche $F_8[7]$, $F_9[8]$ wird nicht aktualisiert
8	"	"	"	"	

	$Sc[i,j]$		$Pr[i,j]$		ADGTCTCA GTCADCTCA
i \ j	8	9	8	9	Hier beginnende *Fragmente* und Kommentare
0	0	0	NIL	NIL	
1	"	"	"	"	
2	2	2	$f_{2,5,2}$	$f_{2,5,2}$	
3	"	"	"	"	
4	4	4	$f_{4,7,4}$	$f_{4,7,4}$	
5	7	7	$f_{5,8,5}$	$f_{5,8,5}$	
6	"	"	"	"	
7	"	"	"	"	
8	"	10	"	$f_{9,8,4}$	

$f_0 = f_{\max} = Pr[8,9] = f_{9,8,4}$, $f_1 = P(f_0) = f_{4,2,2}$ und zuletzt $f_2 = P(f_1) =$ NIL. Das paarweise Alignment zwischen ADGTCTCA und GTCADCTCA sieht also wie folgt aus:

```
adGT---CTCA
--GTcadCTCA
```

Hierbei wurden alignierte *Stellen* großgeschrieben und als *Zuweisungsspalten* genau übereinander gereiht.

Dies sind die Ergebnisse der anderen *Alignments*:

Sequenzen	*Alignments*	*Score*	Sequenzen	*Alignments*	*Score*
1 3	adgTCTCA--- ---TATCAdgg	10	1 4	aDGTC---TCa -DGTCadaTC-	7
2 3	-gTCADctca taTCADgg--	8	2 4	-GTCADCTCa dGTCADATC-	16
3 4	taTCADgg- dgTCADatc	8			

3.4 Überlappgewichte

Beim multiplen Sequenzalignment werden normalerweise DNA- oder Proteinsequenzen miteinander verglichen bei denen man davon ausgeht, dass sie einen gemeinsamen evolutionären Ursprung haben. Gibt es diesen, dann sind fast ausnahmslos auch gemeinsame Motive erhalten geblieben, die in vielen oder sogar allen Sequenzen vorkommen. Für ein biologisch korrektes *Alignment* ist es notwendig diese zu finden und über möglichst viele Sequenzen hinweg einander zuzuweisen. Hat man erstmal diese verwandten Abschnitte gefunden und miteinander aligniert, werden in der Regel auch die Zuweisungen zwischen diesen sogenannten *Ankerpunkten* besser (Morgenstern *et al.*, 2006).

Es ist jedoch nicht immer leicht diese Motive zu finden, weil es sein kann, dass sie im Vergleich zu zufälligen Übereinstimmungen klein sind. Dann bekommen diese nur geringe Gewichte durch unsere Gewichtsfunktion und wenn wir am Ende von DIALIGN durch gieriges Auswählen der *Fragmente* das multiple *Alignment* bestimmen, kann es sein, dass sie nicht berücksichtigt werden, weil andere höher gewichteten Zuweisungen zu ihnen *inkonsistent* sind.

Um dieses Problem zu verhindern und Motive zu bevorzugen, die in möglichst vielen Sequenzen vorkommen, führen wir das Konzept der sogenannten *Überlappgewichte* ein (Morgenstern *et al.*, 1996). Betrachten wir dazu drei verschiedene Sequenzen S_1, S_2 und S_3 und zwei *Fragmente* $f^{1,2}$ und $f^{2,3}$ zwischen diesen. Dann kann es sein, dass die beiden *Fragmente* eine Überlappung in S_2 haben. In diesem Fall ist an dem *Alignment* ein drittes implizites *Fragment* $f^{1,3}$ zwischen S_1 und S_3 beteiligt, das auf ein gemeinsames Motiv zwischen allen drei Sequenzen hindeutet. Daher ist es angemessen die ursprünglichen *Fragmente* stärker zu gewichten, indem wir zu ihnen das Gewicht der Überlappung addieren.

$$\tilde{w}(f^{1,2}, f^{2,3}) := w(f^{1,3}) \tag{3.14}$$

Das *Überlappgewicht* eines *Fragments* mit sich selbst und zwischen zwei *Fragmenten*, die sich nicht überschneiden, definieren wir als 0.

Analog definieren wir das *Überlappgewicht* eines einzelnen *Fragments* als sein Gewicht addiert mit der Summe aller *Überlappgewichte* zwischen sich selbst und allen anderen *Fragmenten*:

$$\hat{w}(f) := w^*(f) + \sum_{e \in F} \tilde{w}(f, e) \tag{3.15}$$

Benutzt man *Überlappgewichte*, muss man jedoch die Zusammensetzung der Sequenzen stärker beachten. Hat man nämlich eine große Subfamilie von sehr ähnlichen Sequenzen, dann werden alle *Fragmente* zwischen einer Sequenz innerhalb und einer Sequenz außerhalb dieser Familie durch hohe *Überlappgewichte* gegenüber denen bevorzugt, die zwischen zwei Sequenzen berechnet wurden, die nicht aus der Sequenzfamilie stammen. Vingron und Sibbald (1993) stellen Methoden vor, die solchen Problemen vorbeugen.

3.4.1 Umsetzung im Programm und Laufzeit

Bei DIALIGN werden naiv alle *Fragmente* der paarweisen *Alignments* miteinander verglichen und auf Überschneidungen untersucht. Da es in den $O(n^2)$ *Alignments* zwischen n

Sequenzen jeweils bis zu $O(L)$ *Fragmente* gibt, kommt man so auf eine Gesamtlaufzeit von $O(n^4 \cdot L^2)$ (Morgenstern, 1999).

Untersucht man das Problem jedoch genauer, stellt man fest, dass man für ein *Fragment* $f^{k,l}$ gar nicht alle *Fragmente* auf Überlappungen überprüfen muss, sondern nur die, an denen eine der beiden Sequenzen S_k oder S_l unseres *Fragments* beteiligt ist. Außerdem müssen wir nicht jedes *Fragment* eines anderen paarweisen *Alignments* betrachten, sondern wir können in sortierten Fragmentketten durch die Start- und Endpunkte sehr genau abschätzen welche für Überlappungen in Frage kommen.

3.4.1 Satz
Für eine Menge S von n Sequenzen und eine Menge F von paarweisen *Fragmenten* zwischen diesen Sequenzen, lassen sich in $O(n^3 \cdot L)$ Zeit die *Überlappgewichte* berechnen.

Beweis. Zwischen den n Sequenzen gibt es $\binom{n}{2} \in O(n^2)$ paarweise *Alignments*. Wir gehen davon aus, dass der vorherige Schritt unseres Verfahrens diese in einer Tabelle A gespeichert hat, wobei $A_{i,j}$ die *Fragmente* des paarweisen *Alignments* zwischen S_i und S_j in einer sortierten Liste enthält. Das können wie o.B.d.A. annehmen, weil der Algorithmus diese ohnehin in sortierter Reihenfolge berechnet.

Betrachten wir ein *Alignment* zwischen den Sequenzen S_i und S_k. Dann müssen wir für die *Überlappgewichte* nur die Einträge $A_{i,k}$ und $A_{l,j}$ mit $1 \leq k, l \leq n$ betrachten, denn es sind nur die *Alignments* relevant, bei denen eine der Sequenzen übereinstimmt. In einer vollständigen Tabelle sind das alle Listen, die in der selben Spalte oder Zeile stehen, also $O(n)$ viele.

Seien $A_{i,k}$ und $A_{k,j}$ zwei *Alignments* von denen wir die Überlappgewichte berechnen wollen. Dazu müssen wir die Überlappung zwischen allen *Fragmenten* in S_k bestimmen. Dies können wir in linearer Zeit machen, indem wir parallel über die beiden sortierten Listen traversieren und anhand der Start- und Endpunkte in S_k die impliziten *Fragmente* zwischen S_i und S_j bestimmen, sowie die Gewichte der *Fragmente* aktualisieren. Dafür benötigen wir nur $O(L)$ Zeit, weil wir einmalig jedes Element der beiden Listen betrachten, es bis zu $O(L)$ *Fragmente* pro *Alignment* gibt und jedes von diesen in der Länge durch $l_{\max} \in O(1)$ beschränkt ist. Insgesamt haben wir also $O(n^2)$ paarweise *Alignments* für die mit jeweils $O(n)$ anderen *Alignments Überlappgewichte* berechnet werden müssen, was jeweils $O(L)$ Zeit kostet. Es folgt die Gesamtlaufzeit von $O(n^3 \cdot L)$. □

Genau genommen brauchen wir keine quadratische Tabelle, weil der Eintrag $A_{i,j}$ aus Symmetriegründen identisch zu $A_{j,i}$ ist. Auch die Diagonale können wir uns sparen, denn das alignieren einer Sequenz mit sich selbst ist unnötig. Des Weiteren kann man sich beim obigen Algorithmus noch die Hälfte des Aufwands sparen, denn die *Überlappgewicht* zwischen $A_{i,k}$ und $A_{k,j}$ müssen wir nicht doppelt berechnen, sondern können sie gleich zu den Gewichten in beiden *Alignments* addieren. Obgleich das nichts an der asymptotischen Laufzeit ändert, macht es in der Praxis einen Unterschied.

2	$A_{2,1}$			
3	$A_{3,1}$	$A_{3,2}$		
⋮	⋮	⋮	⋱	
n	$A_{n,1}$	$A_{n,2}$	...	$A_{n,n-1}$
i / j	1	2	...	n-1

Tabelle 3.1: Jeder Tabelleneintrag $A_{i,j}$ enthält Liste von *Fragmenten*

3.4.2 Beispiel Überlappgewichte

Widmen wir uns den *Überlappgewichten* an unserem Beispiel und betrachten dazu das *Alignment* zwischen den Sequenzen S_1 und S_3. Um das Gewicht zu aktualisieren, müssen wir alle *Alignments* auf Überlappungen überprüfen, in denen eine der beiden Sequenzen vorkommt.

Da unsere Tabelle nicht vollständig ist, reicht es nicht die Einträge der selben Spalte und Zeile zu überprüfen, weil diese unter Umständen nicht vollständig ist. Stattdessen müssen wir alle Einträge in der ersten und dritten Spalte oder Zeile betrachten. Dann gehen wir alle *Fragmente* der Reihe nach durch und gucken anhand der Start- und Endpunkte in der gemeinsamen Sequenz, ob es Überschneidungen gibt. Falls ja, bestimmen wir diese und addieren das Gewicht zu dem unseres *Fragments*.

i \ j	1	2	3
2	$A_{2,1}$		
3	$A_{3,1}$	$A_{3,2}$	
4	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$

Tabelle 3.2: Auf Überlappungen zu überprüfende *Alignments*

Zur Erinnerung hier nochmal der bisherige Stand mit den paarweisen *Alignments*:

Sequenzen	*Alignments*	*Score*	Sequenzen	*Alignments*	*Score*
1	`adgTCTCA---`	10	1	`aDGTC---TCa`	7
3	`---TATCAdgg`		4	`-DGTCadaTC-`	
2	`-gTCADctca`	8	2	`-GTCADCTCa`	16
3	`taTCADgg--`		4	`dGTCADATC-`	
3	`taTCADgg-`	8	1	`adGT---CTCA`	
4	`dgTCADatc`		2	`--GTcadCTCA`	

Wie wir sehen enthält das von uns betrachtete *Alignment* nur das eine Fragment $f_{8,5,5}$ mit drei Übereinstimmungen und einer Abweichung. Als erstes überprüfen wir die Überlappung mit $A_{2,1}$. Beide haben den gemeinsamen Abschnitt `CTCA` in S_1, woraus sich das neue *Fragment* $\binom{\texttt{CTCA}}{\texttt{ATCA}}$ zwischen S_2 und S_3 ergibt. Dieses hat drei Übereinstimmungen, eine Abweichung und somit ein Gewicht von 8. In der Folge addieren wir diese Zahl zum Gewicht von $f^{1,3}_{8,5,5}$ und zu dem von $f^{1,2}_{8,9,4}$. Wenn wir diese Anweisungen auch mit und zwischen allen anderen *Alignments* durchführen, kommen wir zu den folgenden *Überlappgewichten*:

Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*
2	`GTCADCTC`	69	1	`TCTCA`	41	1	`GT`	20
4	`GTCADATC`		3	`TATCA`		2	`GT`	
3	`TCAD`	47	1	`CTCA`	34	1	`TC`	20
4	`TCAD`		2	`CTCA`		4	`TC`	
2	`TCAD`	44	1	`DGTC`	31			
3	`TCAD`		4	`DGTC`				

Da wir im nächsten Schritt die *Fragmente* für unser multiples *Alignment* basierend auf ihren Gewichten gierig auswählen, wurden die Abschnitte bereits sortiert. Ein Algorithmus wird als *gierig* bezeichnet, wenn er eine Entscheidung lokal optimal trifft und danach nicht wieder ändert. In unserem Fall bedeutet das, dass die *Fragmente* mit höheren Gewichten früher gewählt werden und wir sie selbst dann nicht wieder aus unserem multiplen *Alignment* entfernen, wenn sie durch *Inkonsistenzen* verhindern, dass andere, zusammen möglicherweise bessere *Fragment* gewählt werden können. Durch diese radikale Vorgehensweise haben gierige Algorithmen oft gute Laufzeiten, liefern aber nicht

immer die bestmöglichen Ergebnisse. Zu bekannten Vertretern dieses Paradigmas gehören beispielsweise der Algorithmus von Dijkstra zur Bestimmung der kürzesten Pfade aus einem einzelnen Knoten oder die Algorithmen von Prim und Kruskal zur Bestimmung von minimalen Spannbäumen.

3.5 Konsistenz

Die nach ihrem Gewicht sortierten *Fragmente* möchten wir der Reihe nach in unser multiples *Alignment* einfügen, vorausgesetzt, das gerade gewählte ist nicht *inkonsistent* zu den zuvor integrierten. Wenn wir uns an den Abschnitt über die theoretischen Grundlagen erinnern, dann hatten wir formal definiert, dass eine Relation $\mathcal{R}$ genau dann ein *Alignment* ist, wenn „$\preceq_{\mathcal{R}} = (\preceq \cup \mathcal{R})_t$ die natürliche Ordnung auf jeder Sequenz erhält, also $x \preceq_{\mathcal{R}} y \implies x \preceq y$ für alle $x, y \in S_i \forall 1 \leq i \leq n$ gilt."

Mengentheoretisch können wir uns unser vorgehen so vorstellen, dass wir eine Menge von *Fragmenten* $f_1, \cdot f_k$ haben, die wir der Reihe nach in unser wachsendes multiples *Alignment* hinzufügen wollen, vorausgesetzt sie sind *konsistent* zueinander. Die hinzugefügten *Fragmente* bilden dabei für $i = 2, \ldots, k$ eine monoton wachsende Menge $A_1 \subset \ldots \subset A_k$:

$$\begin{aligned} \mathcal{A}_1 &= f_1 \\ \mathcal{A}_i &= \begin{cases} (\mathcal{A}_{i-1} \cup f_i) & \text{falls } f_i \textit{ konsistent} \text{ ist zu } A_{i-1} \\ A_{i-1} & \text{sonst} \end{cases} \end{aligned} \tag{3.16}$$

Das finale *Alignment* $\mathcal{A}$ ist dann genau das resultierende größte *Alignment* $\mathcal{A}_k$. Die Frage, die man sich natürlich jetzt stellt, lautet: „Wie kann ich bestimmen, ob ein *Fragment konsistent* zu einem *Alignment* ist?" Und noch besser: „Wie kann ich das effizient bestimmen?"

3.5.1 Definition (Konsistenzgrenze)
Gegeben seien ein *Alignment* $\mathcal{A}$ auf einer Menge von Sequenzen S mit Stellenraum $\mathcal{S}$. Dann existiert für eine Stelle $s \in \mathcal{S}$ und eine Sequenz $S_i \in S$ eine kleinste und größte Stelle in S_i, die mit s alignierbar ist, ohne zu *Inkonsistenzen* zu führen.

$$\begin{aligned} \underline{b}_{\mathcal{A}}(s,i) &= \min(p : (s,[i,p]) \text{ ist } \textit{konsistent} \text{ zu} \mathcal{A}) \\ \overline{b}_{\mathcal{A}}(s,i) &= \max(p : (s,[i,p]) \text{ ist } \textit{konsistent} \text{ zu} \mathcal{A}) \end{aligned} \tag{3.17}$$

Diese beiden *Stellen* nennen wir die *Konsistenzgrenzen* von s in S_i.

In der ersten Version von DIALIGN wurden diese Konsistenzgrenzen für alle *Stellen* gespeichert und jedes Mal aktualisiert, wenn eine neue Sequenz zum multiplen *Alignment* hinzugefügt wurde (Morgenstern *et al.*, 1996). Das bedeutete benötigten Speicherplatz in der Größenordnung $\theta(n^2 \cdot L)$, denn für jede *Stelle* aus jeder Sequenz ($\theta(n \cdot L)$ viele) mussten die *Konsistenzgrenzen* für jede der n Sequenzen gespeichert werden. Noch schlechter ist die Laufzeit bei diesem naiven Ansatz, denn im schlimmsten Fall gibt es $O(n^2 \cdot L)$ *Fragmente*, die der Reihe nach zum *Alignment* hinzugefügt werden, für die jeweils alle $\theta(n \cdot L)$

Konsistenzgrenzen überprüft und gegebenenfalls angepasst werden müssen. Es folgt eine Laufzeit von $O(n^4 \cdot L^2)$, die auch die Gesamtlaufzeit des DIALIGN-Verfahrens dominiert hat (Morgenstern, 1999).

Weil die Laufzeit im Vergleich zu anderen Verfahren, wie beispielsweise Clustal W, sehr schlecht war, entschied man sich den von Abdeddaïm (1997) veröffentlichten und in der GABIOS-LIB („Greedy Alignment of BIOlogical Sequences LIBrary“) implementierten besseren Ansatz auch in DIALIGN einzubauen. Wie wir im Folgenden sehen werden, ist es möglich das Problem der *Konsistenzgrenzen* durch den Erhalt der transitiven Hülle eines Graphen abzubilden. Dadurch kann man die *Fragmente* deutlich effizienter der Reihe nach in unser multiples *Alignment* einfügen, wodurch sich die praktische Laufzeit in etwa um den Faktor zehn verbessern ließ (Abdeddaïm und Morgenstern, 2000).

3.5.2 Definition (Transitivitätsgrenzen)
Die sogenannten *Transitivitätsgrenzen* erlauben es uns die *Konsistenzgrenzen* als graphtheoretisches Problem zu verstehen. Wir definieren zu unserer *Stelle* s die *Vorgängergrenze* $Pred_{\mathcal{A}}(s,i)$ als die *Stelle* y aus der Sequenz S_i, die von allen *Stellen* mit $y \preceq_{\mathcal{A}} s$ am weitesten rechts steht. Analog wird die *Nachfolgergrenze* $Succ_{\mathcal{A}}(s,i)$ als am weitesten links stehende *Stelle* mit $s \preceq_{\mathcal{A}} y$ festgelegt.

$$\begin{aligned} Pred_{\mathcal{A}}(s,i) &= \max(p : [i,p] \preceq_{\mathcal{A}} s) \\ Succ_{\mathcal{A}}(s,i) &= \min(p : s \preceq_{\mathcal{A}} [i,p]) \end{aligned} \tag{3.18}$$

Zwischen *Transitivitäts-* und *Konsistenzgrenzen* herrscht ein direkter Zusammenhang. Ist unsere *Stelle* s bereits mit einer anderen aus der Sequenz S_i aligniert, dann gibt es zwischen allen vier Grenzen keinen Unterschied.

$$Pred_{\mathcal{A}}(s,i) = Succ_{\mathcal{A}}(s,i) = \underline{b}_{\mathcal{A}}(s,i) = \overline{b}_{\mathcal{A}}(s,i) = p \tag{3.19}$$

Gibt es hingegen keine *Stelle* in S_i, die bereits s zugewiesen wurde, dann unterscheiden sich $Pred_{\mathcal{A}}(s,i)$ und $\underline{b}_{\mathcal{A}}(s,i)$, sowie $Succ_{\mathcal{A}}(s,i)$ und $\overline{b}_{\mathcal{A}}(s,i)$ jeweils nur um genau eine Position.

$$\begin{aligned} Pred_{\mathcal{A}}(s,i) &= \underline{b}_{\mathcal{A}}(s,i) - 1 \\ Succ_{\mathcal{A}}(s,i) &= \overline{b}_{\mathcal{A}}(s,i) + 1 \end{aligned} \tag{3.20}$$

Man kann also sagen, dass *Transitivitätsgrenzen* und *Konsistenzgrenzen* äquivalent sind, denn wenn man das aktuelle *Alignment* kennt, kann man aus dem einen das jeweils andere bestimmen. Daraus folgt auch, dass man mit beiden darstellen kann, ob zwei *Stellen* miteinander alignierbar sind oder nicht.

Beispiel hinzufügen?

In den nächsten beiden Abschnitten zeige ich eine Technik mit der man in konstanter Zeit entscheiden kann, ob zwei *Stellen* miteinander alignierbar sind und einen inkrementellen Algorithmus, der es uns in angemessener Zeit erlaubt ein *Alignment* zwischen zwei *Stellen* in unser multiples *Alignment* hinzuzufügen.

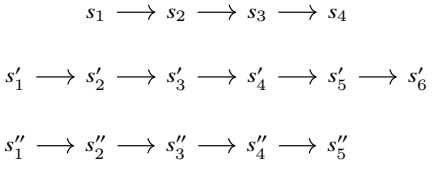


Abbildung 3.1: Graph dreier Sequenzen mit den durch sie induzierten Pfaden

Das wird erreicht, indem wir unseren
Stellenraum $\mathcal{S}$ als gerichteten Graph auffassen, in dem die einzelnen *Stellen* Knoten entsprechen und jede Sequenz einen Pfad durch den Graphen darstellt. Jedes mal, wenn eine Zuweisung zwischen zwei *Stellen* ausgewählt wird, fügen wir eine neue Kante zu unserem Graphen hinzu. Ziel ist es in jedem Schritt unseres Verfahrens die transitive Hülle des Graphen zu kennen, denn mit ihr lässt sich entscheiden, ob das *Fragment*, das wir gerade wählen wollen, *konsistent* zum bisherigen *Alignment* ist oder nicht.

3.5.1 Berechnung der transitiven Hülle eines gerichteten Graphen

Sei $G = (V, E)$ ein gerichteter Graph mit einer Menge Knoten V und einer Menge an Kanten E zwischen diesen Knoten. Unter einem Pfad P auf G verstehen wir ein k-Tupel von Knoten $(v_1, \ldots, v_k)$, sodass $(v_i, v_{i+1}) \in E \forall 1 \leq i \leq k-1$, also eine Folge von Knoten, die alle direkt durch Kanten miteinander verbunden sind. Dabei nennen wir den Index eines Knotens innerhalb dieses Tupels seine *Position* ($pos(x)$ für $x \in P$).

Die transitive Hülle eines Graphen G ist der Graph G^*, in dem es eine Kante $(u, v) \in E^*$ gibt, falls es in G einen Pfad von u nach v gibt. Sollte es $(u, v) \in E^*$ geben, dann nennen wir u den Vorgänger von v und v den Nachfolger von u. Diese Vorgänger und Nachfolger sind nicht mit den gleichnamigen Definitionen aus dem Abschnitt über die paarweisen *Alignments* zu verwechseln. Ich habe mich hier an den Begriffen der Quellen orientiert, die den selben Begriff an verschiedenen Stellen verwenden. Da die beiden Bedeutungen des Begriffs aber nur in zueinander disjunkten und klar getrennten Themengebieten vorkommen, hoffe ich hier nicht allzu viel Verwirrung zu stiften.

Zunächst definieren wir ein paar Variablen für den Kontext dieses Abschnitts: Die Anzahl der Knoten $|V| = \nu$, die der Kanten $|E| = \mu$, die der Kanten in der transitiven Hülle $|E^*| = \mu^*$ und die Anzahl Kanten μ_0, die sich vor dem Hinzufügen von neuen Verbindungen in unserem Graphen befinden. Zuletzt brauchen wir noch μ_p für die Anzahl der Kanten, über die ein Pfad P traversiert.

Sei im Folgenden eine Menge $\mathcal{P} = \{P_1, \ldots, P_k\}$ von Pfaden auf unserem ursprünglichen Graphen gegeben, für die gilt, dass jeder Knoten aus V in genau einem der Pfade vorkommt. Eine solche Menge nennen wir SSDP (*Spanning Set of Disjoint Paths*), also eine Menge von disjunkten Pfaden, die den ganzen Graphen aufspannen. Man kann sich vorstellen, dass man auf unserem Graphen sehr leicht einen solche Menge aufstellen kann, indem man jede einzelne dieser Sequenzen mit ihren Kanten als Pfad dieser Menge auffasst.

3.5.3 Satz
Es sei ein SSDP mit k disjunkten Pfaden gegeben. Dann lässt sich die transitive Hülle G^* unseres Graphen in $O(k^2 \cdot (\mu - \mu_0) + \nu \cdot \min\{\nu, (\mu - \mu_p)\})$ Zeit und mit $O(k \cdot \nu)$ Speicherplatz erhalten, nachdem Kanten zu ihm hinzugefügt wurden (Abdeddaïm, 1997).

Für einen Knoten x sei $P(x)$ das Tupel, das in jedem Eintrag $P(x)[i]$ für $1 \leq i \leq k$ die Anzahl an Vorgängern von x im Pfad P_i angibt. In anderen Worten ist $P(x)[i]$ die maximale Position eines Vorgängers in P_i. Analog definieren wir $S(x)$, wobei $S(x)[i]$ der minimale Nachfolger von x in S_i ist. Wie man sich denken kann, entsprechen $S(x)$ und $P(x)$ genau unseren *Transitivitätsgrenzen* aus dem Kontext von *Alignments* und wir verwenden diese Begriffe synonym. Das werden wir später auch noch beweisen. Gibt es Vorgänger oder

Nachfolger nicht, setzen wir diese Werte auf 0 beziehungsweise die Länge des Pfads, um den es geht, addiert mit eins.

Für Elemente x und y auf zwei Pfaden P_i und P_j kann man sich leicht überlegen, dass aus $pos(x) \leq P(y)[i]$ folgt, dass $(x, y) \in E^*$ gilt. Die gleiche Aussage folgt aus $pos(y) \geq S(x)[j]$. Wie man sieht ist es also möglich die transitive Hülle auf $O(k \cdot v)$ Speicherplatz zu sichern, wenn man ein SSDP mit k Pfaden hat. Das kann signifikant weniger sein, als die $O(v^2)$ Kanten, die man im naiven Ansatz speichern muss.

Beispiel S(x) und P(x)

Als nächstes führen wir eine Funktion ein, die die *Transitivitätsgrenzen* nach dem Hinzufügen einer einzelnen Kante aktualisiert. Wollen wir unseren Graphen um mehr als eine Kante ergänzen, fügen wir sie der Reihe nach hinzu und rufen jedes Mal inkrementell die Funktion auf. Das ist beispielsweise der Fall, wenn wir ein längeres *Fragment* zu unserem *Alignment* hinzufügen. Im Folgenden seien die transitive Hülle und die *Transitivitätsgrenzen* unseres ursprünglichen Graphen gegeben. Sei außerdem (x, y) die Kante, die wir ergänzen.

Nun wählen wir einen beliebigen, aber festen Knoten $u \in V$, dessen *Transitivitätsgrenzen* wir aktualisieren. $P(u)$ und $S(u)$ bezeichnen dabei die Grenzen vor und $P(u)'$ und $S(u)'$ die Grenzen nach dem Hinzufügen. Als erstes kann man sich überlegen, dass die *Vorgängergrenze* immer nur wachsen kann, während die *Nachfolgergrenze* höchstens sinkt. Des Weiteren kann sich durch eine Kante von x nach y nur die *Vorgängergrenze* von u in einer Sequenz verändern, wenn u ein Nachfolger von y war und die *Nachfolgergrenze*, wenn es ein Vorgänger von x war. Im ersten Fall, wählt man dann das Maximum der beiden möglichen Werte aus und in letzterem das Minimum.

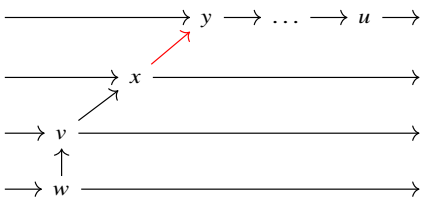


Abbildung 3.2: Füge (x, y) ein.

Mit diesen zwei Formeln können wir für ein $u \in P_i$ und jeden Pfad P_j die neuen *Transitivitätsgrenzen* von u bestimmen:

$$P'(u)[j] = \begin{cases} \max\{P(u)[j], P(x)[j]\}, & \text{falls } u \text{ Nachfolger von y war} \\ P(u)[j], & \text{sonst} \end{cases} \tag{3.21}$$

$$S'(u)[j] = \begin{cases} \min\{S(u)[j], S(y)[j]\}, & \text{falls } u \text{ Vorgänger von x war} \\ S(u)[j], & \text{sonst} \end{cases} \tag{3.22}$$

Falls bereits ein Pfad zwischen x und y existiert, wissen wir, dass sich die transitive Hülle nicht verändert und dass wir die *Transitivitätsgrenzen* dementsprechend nicht anpassen müssen. Die obigen Beobachtungen können wir jetzt direkt in einen Algorithmus $\mathrm{EdgeAddition}$ münden lassen, indem wir über alle Paarkombinationen von Pfaden iterieren und dabei für jeden in Frage kommenden Knoten im jeweiligen Pfad die obigen Formeln anwenden. Wir wählen dabei die Knoten für die *Nachfolgergrenze* in absteigender und die für die *Vorgängergrenze* in aufsteigender Reihenfolge. Weil wir wissen, dass die *Nachfolgergrenze* immer nur sinken kann und wir die in Frage kommenden Knoten sortiert betrachten, wissen wir, dass der aktuelle Schleifendurchlauf abgebrochen werden kann, wenn die *Nachfolgergrenze* kleiner oder genauso groß ist, wie die von y. In diesem Fall würde sie eh nicht mehr aktualisiert werden und wir können uns die Berechnung

sparen. Für die *Vorgängergrenzen* gehen wir analog vor. Jetzt haben wir alles Rüstzeug zusammen, um Satz 3.5.3 zu beweisen.

Algorithmus 3 EdgeAddition

Require: Gerichteter Graph G mit SSDP $P_1, \ldots, P_k$

```
1:  procedure EdgeAddition(x, y)
2:      if (x, y) ∉ E* then
3:          for each Pfad P_i do
4:              for each Pfad P_j do
5:                  for each u in P_i startend von P(x)[i] in absteigender Reihenfolge do
6:                      if S(u)[j] > S(y)[j] then
7:                          S(u)[j] ← S(y)[j]                ▷ aktualisiere Nachfolgergrenze
8:                      else
9:                          breche den aktuellen Schleifendurchlauf für u ab
10:                     end if
11:         for each Pfad P_i do
12:             for each Pfad P_j do
13:                 for each u in P_i startend von S(y)[i] in aufsteigender Reihenfolge do
14:                     if P(u)[j] < P(x)[j] then
15:                         P(u)[j] ← P(x)[j]                ▷ aktualisiere Vorgängergrenze
16:                     else
17:                         breche den aktuellen Schleifendurchlauf für u ab
18:                     end if
19:     end if
20: end procedure
```

Beweis. Im schlimmsten Fall werden die ineinander verschachtelten Schleifen je $O(k^2 \cdot (\mu - \mu_0))$-mal durchlaufen. Das ergibt sich aus der Kombination aller $O(k^2)$ Paare von Pfaden und dadurch, dass es sein kann, dass nur für die ursprünglichen Kanten μ_0 im Graphen und die sie verbindenden Knoten die if-Bedingungen 6 und 14 nicht erfüllt sind.

Ist das wirklich der Grund? Denk nochmal drüber nach.

Wenn wir darüber nachdenken wie oft es wirklich zu einer Aktualisierung der *Transitivitätsgrenzen* kommen kann, stellen wir fest, dass diese je nach Graph möglicherweise viel geringer ist, als oben abgeschätzt. Um Redundanz zu vermeiden betrachten wir hier nur die *Vorgängergrenzen*. Für *Nachfolgergrenzen* gilt aus Symmetriegründen aber genau das selbe. Die einzige Situation, in der ein Knoten $v \in V$ die *Vorgängergrenzen* eines anderen Knotens u verändern kann, ist wenn v ein Vorgänger von x, aber kein Vorgänger von u ist, bevor die Kante (x, y) hinzugefügt wird (vgl. **??**). Gleichzeitig muss gelten, dass u ein Nachfolger von y ist. Das liegt daran, dass die Pfade $\mathcal{P}$ disjunkt sind. Also folgt, dass die *Vorgängergrenzen* von u höchstens v-mal angepasst werden und es insgesamt nicht mehr als v^2 Anpassungen gibt.

Sei (w, v) eine Kante unseres Graphen, die auf keinem unserer Pfade aus dem SSDP liegt. Diese Kante kann die *Vorgängergrenze* von u in der Methode $\mathrm{EdgeAddition}$ nur dann verändern, wenn v ein Vorgänger von x ist, aber kein Vorgänger von u, bevor die Kante (x, y) addiert wird. Nach dieser Aktualisierung kann die Kante (w, v) die *Transitivitätsgrenzen* von u nie wieder anpassen. Das bedeutet, dass dies für einen Knoten u höchstens so oft passieren, wie es Kanten gibt, die in keinem Pfad des SSDP liegen. Das sind genau $O(\mu - \mu_p)$ und es folgt, dass es insgesamt höchstens $v \cdot (\mu - \mu_p)$ solcher Aktualisierungen geben kann.

Da sowohl die Anzahl dieser Knoten, als auch der Kanten im Graph die Anzahl der Aktualisierungen beschränken, kann es nicht zu mehr Aktualisierungen als dem Minimum dieser beiden Werte kommen, also $O(\nu \cdot \min\{\nu, \mu - \mu_p\})$. Fasst man alles zusammen, so folgt die behauptete Laufzeit von $O(k^2 \cdot (\mu - \mu_0) + \nu \cdot \min\{\nu, \mu - \mu_p\})$. □

3.5.2 Konsistenzgrenzen durch Berechnung der transitiven Hülle

Jetzt möchten wir den allgemeinen graphtheoretischen Ansatz auf unser multiples *Alignment* anwenden.

3.5.4 Definition (Alignmentgraph)
Zunächst übertragen wir daher unsere Sequenzen $S = \{S_1, \ldots, S_n\}$ auf einen Graphen mit SSDP. Hierbei ist jede *Stelle* aus $\mathcal{S}$ ein Knoten und zwischen zwei Knoten u und v gibt es genau dann eine Kante $(u, v) \in E$, wenn die dazugehörigen *Stellen* aus der selben Sequenz kommen und direkt aufeinanderfolgen. Dadurch ergibt sich automatisch unser SSDP mit n Pfaden, denn jede Sequenz liefert genau einen solchen und da jeder Knoten mit genau einer Sequenz assoziiert ist, spannen diese Pfade auch den ganzen Graphen. Den Pfad einer Sequenz S_i bezeichnen wir als P_i. Jedes Mal, wenn wir in unserem *Alignment* von S zwei *Stellen* (i, p) und (j, q) miteinander alignieren, fügen wir je eine Kante in beide Richtungen zwischen den dazugehörigen Knoten im Graphen ein.

Gibt es zwischen u und v Pfade in beiden Richtungen ((u, v) und $(v, u) \in E^*$), dann symbolisieren wir dies durch $u \rightleftharpoons^* v$. Wir sagen dann, dass u und v miteinander *koinzidieren*.

Wie bereits angedeutet ist es bei einem *Alignment* möglich Lücken so in die Sequenzen einzufügen, dass genau die einander zugewiesenen Symbole übereinander stehen. Diese Tupel von *Stellen* nennen wir *Zuweisungsspalten* und zu ihnen zählen wir nur die miteinander alignierten Symbole. Andere, nicht mit den gerade betrachteten *Stellen* alignierte Symbole, gehören nicht zur *Zuweisungsspalte*, selbst wenn sie in der selben Spalte der Sequenzen stehen, nachdem man Lücken eingefügt hat. Eine Menge von miteinander *koinzidierenden* Knoten nennen wir einen *Anker*, wenn es keine weiteren Knoten gibt, die mit Knoten aus dieser Menge *koinzidieren*.

Beispiel

Wir nennen einen *Alignmentgraphen* mit seinem SSDP (G, P) auf einer Menge von Sequenzen S genau dann *regulär*, wenn die Relation auf der er beruht ein *Alignment* ist. Das heißt, wenn die Zuweisungen keine *Inkonsistenzen* enthalten.

3.5.5 Lemma
Sei G ein *Alignmentgraph* und P das ihm zugewiesene SSDP. (G, P) ist genau dann regulär, wenn jeder dazugehörige *Anker* höchstens einen Knoten aus jedem Pfad von P hat. In diesem Fall entspricht ein *Anker* genau einer *Zuweisungsspalte*.

Beweis. „⇒“:Angenommen jeder *Anker* unseres *Alignmentgraphen* G enthält höchstens einen Knoten aus jedem Pfad in P. Betrachten wir jetzt die Halbordnung $\preceq^*$, bei der für zwei *Anker* a und b $a \preceq^* b$ genau dann gilt, wenn zwischen zwei Knoten $u \in a$ und $v \in b$ ein Pfad in G existiert. Da keiner der *Anker* von G zwei oder mehr verschiedene Knoten vom selben Pfad aus P enthält und da auf ihnen unsere Halbordnung $\preceq^*$ existiert, können wir anhand dieser die *Anker* immer in *Zuweisungsspalten* übereinander schreiben und haben

somit ein *Alignment*. $\preceq^*$ entspricht dabei genau der topologischen Sortierung auf G. Da auf (G, P) ein *Alignment* existiert, ist es *regulär*.

„$\Leftarrow$“:Sei G regulär. Dann ist die zugehörige Relation zwischen Knoten per Definition ein *Alignment* und für dieses können wir, wie wir wissen, die Zuweisungen in *Zuweisungsspalten* untereinander schreiben. Da dies möglich ist, sind alle *Anker Zuweisungsspalten* und keiner enthält zwei oder mehr Knoten aus der selben Sequenz. □

Für den Rest des Abschnitts nehmen wir an, dass unser *Alignmentgraph* G mit SSDP *regulär* ist. Ziel ist es jetzt zu überprüfen, ob er auch regulär bleibt, nachdem man eine Zuweisung zwischen zwei *Stellen* s und s und die korrespondierenden Kanten in G hinzugefügt hat. Falls ja, dann nennen wir s und t *alignierbar*, einen Begriff, den wir zuvor bereits informell genutzt haben.

3.5.6 Lemma
Zwei *Stellen* s und t mit den korrespondierenden Knoten u und v sind genau dann *alignierbar*, wenn eine der folgenden zwei Bedingungen eintritt:
(1) Es existiert kein Pfad zwischen u und v, das heißt $(u, v), (v, u) \notin E^*$,
(2) u und v *koinzidieren* miteinander, also $u \rightleftharpoons^* v$.

Beweis. Sei G' der Graph nach dem Hinzufügen der Kanten (u, v) und (v, u). Wir müssen zeigen, dass (G', P) genau dann regulär ist, wenn eine der beiden obigen Bedingungen eintritt.

„$\Rightarrow$“:(1) Angenommen es existiert kein Pfad zwischen u und v und seien a_1 und a_2 die dazugehörigen *Anker* der beiden Knoten. Es sei $a = a_1 \cup a_2$ der neue *Anker* von G' nach dem Hinzufügen von (u, v). Dann kann a nicht mehrere Knoten vom selben Pfad aus P enthalten, weil es sonst bereits einen Pfad von u nach v oder andersrum gegeben hätte und die Vorbedingung verletzt gewesen wäre. Das liegt daran, dass es zwischen diesen Knoten vom selben Pfad ja eine Kante gegeben haben müsste über die dann auch eine einseitige Verbindung zwischen u und v bestünde. Weil alle anderen *Anker* von G nicht verändert werden, folgt, dass G' regulär ist.

(2) Es gelte $u \rightleftharpoons^* v$ für G. Dann sind die *Anker* von G' die selben wie von G, denn u und v haben bereits zuvor *koinzidiert*. Also ist auch G' *regulär*, da G es war.

„$\Leftarrow$“:Die Rückrichtung des Beweises führen wir per Kontrapositionsbeweis. Es sei also nur eine einseitige Verbindung zwischen u und v gegeben. O.B.d.A. nehmen wir an, dass $(u, v) \in E^*$ und $(v, u) \notin E^*$. Da G ein *Alignmentgraph* ist, gibt es zwangsläufig zwei Knoten x und y, die auf dem selben Pfad P_i aus P liegen und über die der Pfad von u nach v läuft. Gäbe es keinen solchen Knoten, dann wäre die Vorbedingung nicht erfüllt, weil nur Kanten auf den Pfaden P keine antiparallele Kante haben. Nach dem Hinzufügen der Kante (u, v) mit dem resultierenden *Alignmentgraphen* G' gäbe es dann einen Pfad von y nach x, denn $(y, v), (v, u), (u, x) \in E^*$. Daraus folgt dann $x \rightleftharpoons^* y$, also eine *Koinzidenz* zwischen x und y vom selben Pfad P_i, wodurch die *Regularität* von G' verletzt ist. □

Sei ein Symbol s aus der Sequenz S_i mit korrespondierendem Knoten v aus dem *Alignmentgraphen* gegeben. Dann sind genau die Symbole einer zweiten Sequenz S_j mit s *alignierbar*, die zwischen $P(v)[j]+1$ und $S(v)[j]-1$ liegen. Das liegt an einer Beobachtung, die wir bereits zu Beginn dieses Unterabschnitts gemacht haben: „Für Elemente x und y auf zwei Pfaden P_i und P_j kann man sich leicht überlegen, dass aus $pos(x) \leq P(y)[i]$ folgt, dass $(x, y) \in E^*$ gilt. Die gleiche Aussage folgt aus $pos(y) \geq S(x)[j]$.“ Haben wir

jetzt einen Knoten u auf P_j mit $P(v)[j] + 1 \leq u \leq S(v)[j] - 1$, dann gilt $u \rightleftharpoons^* v$ und die beiden korrespondierenden Symbole sind miteinander *alignierbar* nach dem letzten Lemma. Diese Überprüfung ist in $O(1)$ Zeit möglich. Wie bereits angedeutet entsprechen die *Transitivitätsgrenzen* aus dem Kontext der Graphen genau denen auf unseren Sequenzen, denn der größte Vorgänger von u auf dem Graphen P_i $P(u)[i]$ ist wie wir gezeigt haben auch gleichzeitig auch der am weitesten rechts stehende Knoten v, für den in unserem *Alignment* $v \preceq_{\mathcal{A}} u$ gilt. Es folgt $P(u)[i] = Pred_{\mathcal{A}}(u, i)$ und für die *Nachfolgergrenzen* gilt das selbe.

3.5.7 Korollar
Für eine Menge an Sequenzen $S = \{S_1, \ldots, S_n\}$ und zugehörigem *Alignmentgraphen* mit SSDP (G, P) kann die transitive Hülle von G in $O(n^3 \cdot L + n^2 \cdot L^2)$ Zeit und mit $O(n^2 \cdot L)$ Speicherplatz erhalten werden, wenn L die maximale Länge aller Sequenzen ist.

Beweis. Da (G, P) *regulär* ist, enthält jeder *Anker* höchstens einen Pfad aus P. Die Anzahl der Kanten pro *Anker* ist also beschränkt durch die Anzahl an möglichen Paaren zwischen den Sequenzen mal zwei, denn für zwei *alignierte* Symbole werden zwischen den dazugehörigen Knoten auch zwei Kanten hinzugefügt. Von diesen $n \cdot (n-1)$ Kanten können aber nur $2 \cdot (n-1)$-viele für Aktualisierungen im Algorithmus $\mathrm{EdgeAddition}$ sorgen. Das liegt daran, dass jeder Knoten nur mit $(n-1)$ neuen Knoten verbunden werden kann. Alle weiteren Kanten ändern die transitive Hülle des *Alignmentgraphen* nicht und werden direkt in Zeile 32 abgefangen.

Die maximale Anzahl an möglichen Kanten wird erreicht, wenn man $O(L)$ *Anker* hat, die sich über möglichst viele Sequenzen spannen. Es ist zwar möglich, dass man deutlich mehr *Anker* hat (bis zu $O(n \cdot L)$), aber das ginge mit einer Reduzierung der zu überprüfenden Kanten einher. Im schlimmsten Fall kommt es also zu $O(n \cdot L)$ rechenintensiven Aufrufen von $\mathrm{EdgeAddition}$. Erinnern wir uns an die Laufzeit von $\mathrm{EdgeAddition}$ auf unserem Graphen: $O(n^2 \cdot (\mu - \mu_0) + \nu \cdot \min\{\nu, \mu - \mu_p\})$, mit $\mu - \mu_0$ der Anzahl an hinzugefügten Verbindungen, ν der Anzahl an *Stellen* aller Sequenzen und $\mu - \mu_p$ der Anzahl an Kanten zwischen allen Knoten, die nicht zur gerade betrachteten Sequenz gehören. $\mu - \mu_p$ liegt im schlimmsten Fall über ν, sodass dieses als Minimum im entsprechenden Term ausgewählt wird. Es gelten $\mu - \mu_0 \in O(n \cdot L)$ und $\nu \in O(n \cdot L)$. Die Laufzeit von $\mathrm{EdgeAddition}$ aller Kanten im rechenintensiven Fall liegt also in $O(n^3 \cdot L + n^2 \cdot L^2)$. Die anderen Kanten, die nicht in Zeile 32 abgefangen werden, sind nur $O(n^2 \cdot L)$-viele und da ihre Bearbeitung nur konstante Zeit kostet, beträgt die Laufzeit auch insgesamt $O(n^3 \cdot L + n^2 \cdot L^2)$ Zeit.

Während des Algorithmus müssen wir stets unseren *Alignmentgraphen* vorhalten. Dieser enthält für jede *Stelle* einen Knoten, also $O(n \cdot L)$-viele und $O(n \cdot L)$ Kanten für unsere Pfade für jede Sequenz, sowie im Worst-Case für $O(L)$ Anker die jeweils bis zu n^2 Verbindungen zwischen den Symbolen dieser *Zuweisungsspalte*. Der benötigte Speicherplatz beträgt somit $O(n^2 \cdot L)$. □

Die Schlussfolgerung des letzten Satzes ist, dass es möglich ist die *Transitivitäts-* und somit *Konsistenzgrenzen* für ein ganzes multiples *Alignment* in $O(n^3 \cdot L + n^2 \cdot L^2)$ Zeit und mit $O(n^2 \cdot L)$ Speicherplatz zu berechnen.

Durch ein paar geschickte Tricks ist es Abdeddaïm und Morgenstern (2000) gelungen die Laufzeit und den Speicherplatz für die DIALIGN-Implementierung mit GABIOS-LIB weiter zu verbessern. Wenn zwei *Stellen* aus $\mathcal{S}$ miteinander aligniert sind, dann folgt daraus automatisch, dass ihre *Transitivitätsgrenzen* identisch sind. Das nutzen wir aus, indem

wir nicht die Grenzen für jede *Stelle* speichern und aktualisieren, sondern stattdessen für ganze Äquivalenzklassen $[x]_{\mathcal{A}}$ unseres *Alignments*.

Die zweite Verbesserung zielt auf *verwaiste Stellen* ab, das heißt solche, die mit keiner anderen aligniert sind. Sei $x = (i, p)$ ein solcher *Waise* aus der Sequenz S_i an der Stelle p. Dann stimmen die *Nachfolgergrenzen* von x genau mit denen der am weitesten links stehenden *Stelle* $y = (i, p')$ überein mit $p < p'$, sodass y kein *Waise* ist. Jetzt genügt es ein Feld mit Werten nextClass$[x] = p'$ für alle *Waisen* unserer Sequenzen zu speichern. Dadurch ist es uns möglich die *Transitivitätsgrenzen* $Succ_{\mathcal{A}}(x, j)$ dieser *Stellen* in konstanter Zeit zu aktualisieren, wenn eine neue Zuweisung zwischen unseren Sequenzen und somit eine neue Kante in unserem *Alignmentgraphen* hinzugefügt wird. Möglicherweise kann man dieses Array auch durch eine Baum- oder andere Datenstruktur ersetzen, die es uns ermöglicht Einträge zu löschen, nachdem die entsprechenden *Stellen* keine *Waisen* mehr sind. Das verhindert, dass beim Alignieren von sehr ähnlichen Sequenzen viel Speicher durch Arrays belegt sind, deren Einträge nie wieder benötigt werden. Dieser Ansatz könnte insbesondere deshalb gut funktionieren, weil *Stellen* innerhalb eines Intervalls zwischen zwei *Nichtwaisen* die gleichen *Vorgänger* und *Nachfolger* haben. Man kann jetzt einen AVL- oder B-Baum pro Sequenz nehmen über der Ordnung der Startpunkte aller Intervalle. Jedes Knotenelement speichert dann zusätzlich seinen Endpunkt, sowie *Vorgänger* und *Nachfolger*. Wird ein neues *Fragment* ins *Alignment* eingefügt, wird das Intervall höchstens in zwei neue aufgeteilt, was die Löschung eines und das Hinzufügen zwei neuer Knoten zur Folge hätte. Asymptotisch wäre die Laufzeit vermutlich schlechter, aber durch den mitunter deutlich verringerten Speicherplatz in der Praxis letztendlich besser. Das weiterzuführen ginge aber zu weit und soll nicht Teil dieser Bachelorarbeit sein.

Sobald Beispiele für Transitivitätsgrenzen gegeben sind, hier darauf verweisen

3.5.3 Einbindung in DIALIGN

Auch wenn dieses Thema in Abdeddaïm und Morgenstern (2000) nicht behandelt wird, möchte ich kurz darauf eingehen was notwendig ist, um den obigen Ansatz zur Überprüfung von *Konsistenz* im konkreten Fall von DIALIGN anzuwenden.

DIALIGN ist ein segmentbasiertes Verfahren für multiples Sequenzalignment. Unser Algorithmus $\mathrm{EdgeAddition}$ addiert aber immer nur eine einzelne Kante, die eine symbolweise Zuweisung darstellt, zu unserem *Alignmentgraphen*. Um sicherzustellen, dass ein ganzes *Fragment*, das wir zu unserem *Alignment* hinzufügen wollen, zu den bisher gewählten Zuweisungen *konsistent* ist, betrachten wir dieses auf der Ebene einfacher Paare von *Stellen*. Zunächst überprüfen wir für alle *Stellenpaare* nacheinander, ob diese alignierbar sind. Falls ja, können wir sie der Reihe nach zum *Alignment* mit $\mathrm{EdgeAddition}$ hinzufügen und falls nicht, würde das *Fragment* für *Inkonsistenzen* sorgen und wir fügen nichts hinzu. Die Überprüfung kann einmalig für alle Paare am Anfang durchgeführt werden, weil jedes Paar definitiv zu einem eigenen *Anker* gehört und die *Alignierbarkeit* nicht durch andere Paare desselben *Fragments* eingeschränkt wird.

Es mag sein, dass man für die „inneren“ *Stellen* der Fragmente effizientere Wege finden kann, um diese zum *Alignment* hinzuzufügen als $\mathrm{EdgeAddition}$. In diesem Fall wäre der Algorithmus nur für die beiden Randpaare unseres *Fragments* nötig.

3.5.4 Evaluation von DIALIGN mit der GABIOS-LIB

Nachdem die GABIOS-LIB in DIALIGN 2.1 integriert wurde, haben Abdeddaïm und Morgenstern die neue Version auf verschiedenen simulierten und echten Datensätzen getestet und mit der vorherigen verglichen (Abdeddaïm und Morgenstern, 2000). Die Datensätze umfassten dabei bis zu 200 Sequenzen mit Durchschnittslängen von 100, 63,3 und 119,3 je nach Satz.

Es hat sich herausgestellt, dass die Laufzeit der alten Version sich proportional zu n^4 verhielt, während die von DIALIGN 2.1 sogar besser als n^3 war. Wie nicht anders zu erwarten, wurde der Unterschied bei steigender Anzahl der Sequenzen n tendenziell größer. Die Laufzeit verbesserte sich im besten Fall um den Faktor 120 und im Durchschnitt etwa um den Faktor 10. Insbesondere die höhere Geschwindigkeit bei großen *Alignments* mit mehr Sequenzen ist erfreulich, da diese ohnehin komplexer zu berechnen sind.

Ein weiterer Vorteil ist der verringerte Speicherverbrauch im Verhältnis zur alten Version 2.0. Zwar beträgt der Speicherverbrauch der neuen Implementierung im schlimmsten Fall nach wie vor $O(n^2 \cdot L)$, was zuvor der echte benötigte Speicher war. In der Praxis sank er aber um einen konstanten Faktor abhängig von der Ähnlichkeit der alignierten Sequenzen (Ausnahme: randomisierte Sequenzen). Die Anzahl n hatte hierbei keinen signifikanten Einfluss auf den Grad der Verbesserung. Dieser lag bei den nichtrandomisierten Datensätzen etwa zwischen fünf und zehn.

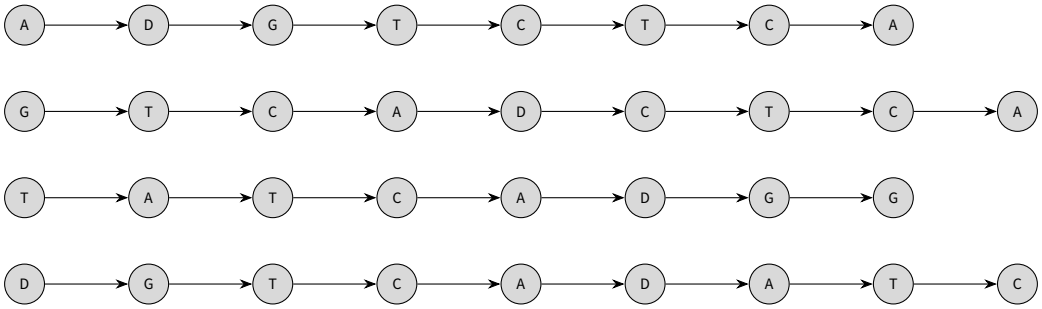
3.5.5 Beispiel Konsistenzgrenzen

Für die *Konsistenzgrenzen* brauchen wir in jedem Schritt unseren *Alignmentgraphen*, alle Äquivalenzklassen unseres *Alignments*, deren *Vorgänger-* und *Nachfolgergrenzen* und die Felder $\mathrm{nextClass}$ und $\mathrm{prevClass}$ für alle *Waisen*. Der Übersichtlichkeit halber gebe ich nicht die ganzen Felder an, sondern nur die Werte ungleich „NIL“. Zur Erinnerung hier nochmal der aktuelle Zwischenstand mit unseren *Fragmenten* sortiert nach *Überlappgewichten*.

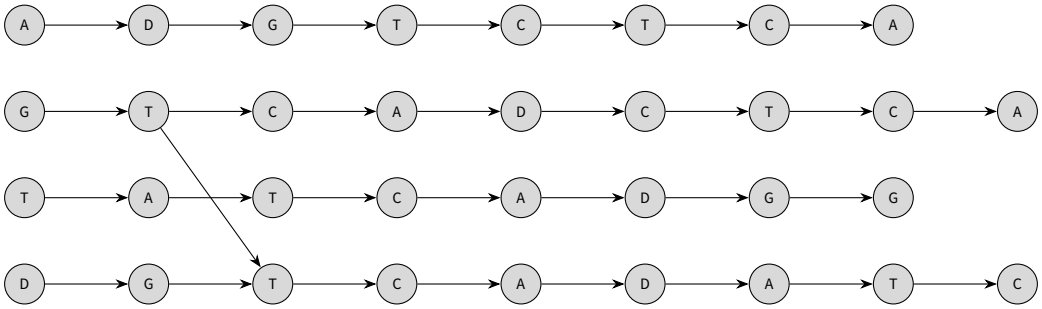
Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*
2	GTCADCTC	69	1	TCTCA	41	1	GT	20
4	GTCADATC		3	TATCA		2	GT	
3	TCAD	47	1	CTCA	34	1	TC	20
4	TCAD		2	CTCA		4	TC	
2	TCAD	44	1	DGTC	31			
3	TCAD		4	DGTC				

Diese werden wir jetzt der Reihe nach in unseren *Alignmentgraphen* einfügen, wenn die *Konsistenz* dadurch erhalten bleibt.

Vor dem ersten Schritt enthält der *Alignmentgraph* genau unsere *Stellen* $\mathcal{S}$ mit den Pfaden $\mathcal{P}$, die der natürlichen Ordnung auf den jeweiligen Sequenzen entsprechen. Weil es noch keine *Nichtwaisen* gibt, können wir auch die Felder $\mathrm{nextClass}$ und $\mathrm{prevClass}$ zunächst ignorieren, da es keine *Stellen* gibt auf die diese verweisen können.



Zu Beginn haben wir das *Fragment* $\binom{\texttt{GTCADCTC}}{\texttt{GTCADATC}}$ zwischen den Sequenzen S_2 und S_4. Da unser *Alignment* noch leer ist, bleibt die *Konsistenz* auf jeden Fall gewahrt. Wir fügen daher die Kanten in unseren *Alignmentgraphen* ein und berechnen die *Vorgänger-* und *Nachfolgergrenzen* für alle Äquivalenzklassen und *Waisen*.



3.6 Zusammenfassung

Als abschließenden Schritt müssen wir noch unsere Ausgabe vorbereiten. Das Ziel ist dabei alignierte Symbole als Groß- und *Waisen* als Kleinbuchstaben darzustellen. Zudem sollen die *Zuweisungsspalten* genau übereinander stehen, wofür unter Umständen das Einfügen von Leerzeichen in die Ausgabesequenzen nötig ist. Hierfür sortieren wir alle *Fragmente* für jede Sequenz nach Startpunkten und traversieren diese, sowie die Sequenzen selbst, der Reihe nach durch, wobei wir bei Bedarf Lücken einfügen. Da die Länge jeder Sequenz in $O(1)$ ist, ist dies in $O(n^2 \cdot L)$ möglich (Morgenstern, 1999).

3.6.1 Gesamtkomplexität

3.6.1 Korollar
Mit DIALIGN lässt sich ein multiples *Sequenzalignment* in $O(n^3 \cdot L + n^2 \cdot L^2)$ Zeit und mit $O(N_{\max} + n^2 \cdot L)$ Speicherplatz berechnen.

Beweis. Mit unserem speichereffizienten Algorithmus lässt sich jedes paarweise *Alignment* in $O(L^2)$ Zeit und mit $O(N_{\max}) \in O(L^2)$ Speicherplatz berechnen (Morgenstern, 2002). Da es $O(n^2)$ paarweise *Alignments* gibt, folgt hierfür die Laufzeit in $O(n^2 \cdot L^2)$. Die Werte für unsere *Fragmente* speichern wir sortiert in einer $O(n^2)$ großen Tabelle, wobei jeder Eintrag auf die Liste von *Fragmenten* eines paarweisen *Alignments* verweist. Da jedes von diesen bis zu $O(L)$ Segmente enthält, folgt der Speicherverbrauch von $O(n^2 \cdot L)$ zu diesem Zeitpunkt.

Mit Hilfe unserer Tabelle und dem parallelen traversieren sortierter Listen können wir in-place und in $O(n^3 \cdot L)$ Zeit unsere *Überlappgewichte* berechnen.

Bevor wir die *Fragmente* gierig in unser multiples *Alignment* einfügen können, müssen wir sie alle auf Basis ihrer *Überlappgewichte* sortieren. Mit einem effizienten Algorithmus wie beispielsweise Heapsort ist dies in $O(n^2 \cdot L \cdot \log(n^2 \cdot L))$ Rechenschritten möglich.

Im vorletzten Schritt von DIALIGN konstruieren wir unseren *Alignmentgraphen* und benutzen ihn, um die *Transitivitäts-* und somit die *Konsistenzgrenzen* zu berechnen. Er enthält $O(n \cdot L)$ Knoten mit bis zu $O(n^2 \cdot L)$ Kanten und wie in Satz 3.5.3 bewiesen kostet das Aktualisieren der *Transitivitätsgrenzen* $O(n^3 \cdot L + n^2 \cdot L^2)$ Rechenschritte.

Das vorbereiten der Ausgabe ist wie gerade erwähnt in $O(n^2 \cdot L)$ Zeit möglich.

Somit dominieren die *Konsistenzgrenzen* die Laufzeit und je nach $N_{\max}$ in unseren paarweisen *Alignments* auch den Speicherverbrauch. Es folgt die behauptete Laufzeit von $O(n^3 \cdot L + n^2 \cdot L^2)$ Rechenschritten und benötigter Speicherplatz in $O(N_{\max} + n^2 \cdot L)$. □

3.7 Evaluierung und Schwächen des Ansatzes

4 Ein Min-Cut-Ansatz für das Konsistenzproblem

-skizzierter Ablauf des Algorithmus

4.1 Flussnetzwerke

4.1.1 Einführung

4.1.2 Wichtige Algorithmen

4.1.3 Der *Max-Flow-Min-Cut-Satz*

4.2 Inzidenzgraphen und das Auflösen von Inkosistenzen mit Hilfe von Flussnetzwerken

4.2.1 Komplexität

$O(n^4 * l^{7/2})$

4.3 Sukzessorgraphen und der Algorithmus von Pitschi

$O(n^3 * l)$ für n-faches Topological Sort auf einem Knoten mit $O(n^2 * l)$-vielen Kanten im schlimmsten Fall.

4.4 Ankerpunkte

4.5 Gesamtkomplexität

$O(n^4 * l^{7/2})$

5 Programmierung

5.1 Speichereffiziente Umsetzung der dynamischen Programmierung

6 Validierung der Ergebnisse

6.1 Vorstellung BAliBase und (D)IRMBASE

6.2 Test auf BAliBase

6.3 Test auf DIRMBASE und IRMBASE

7 Fazit

7.1 Zusammenfassung

7.2 Future Works

- Statt paarweisen Alignments, Alignments von je drei Sequenzen berechnen
- Conditional Random Fields statt Gewichtsfunktionen
- Bessere Heuristik zum Löschen der Kanten aus dem Sukzessorgraph benutzen (gerade wenn Tests zeigen, dass oft sehr viele Kanten gelöscht werden) - das könnte vor allem dann vorkommen, wenn die Sequenzen große Überkreuzungen enthalten, weil man dann große Zyklen mit sehr hohen Kantengewichten hat. -> mögliches Paper?
- Statt DIALIGN 2 DIALIGN TX zwischen den Ankerpunkten nutzen
- Wir wollen aus dem Sukzessorgraph möglichst wenige Knoten löschen. Dafür sind Sites, die aber als einzige in ihren Knoten vorkommen, irrelevant.
    1. Jede Kante hinter Knoten hat Gewicht von Anzahl an Sites - 1 => bevorzugt Zuweisungen über möglichst viele Sequenzen hinweg. Es kann aber sein, dass dadurch viele kleine Zuweisungen aufgelöst werden
    2. Jede Kante hinter Knoten hat Gewicht 1, falls mehr als eine Sequenz beteiligt und Gewicht 0, falls nicht => bevorzugt viele kleine Alignments und minimiert Anzahl der Löschungen
- Teile des Algorithmus lassen sich gut parallelisieren:
    1. Die $(\frac{1}{2} * (n^2 - n))$-vielen paarweisen Alignments lassen sich parallelisiert berechnen.
    2. Die Überlappgewichte lassen sich gut parallelisiert berechnen, weil keins der Ergebnisse von denen der anderen abhängt. Paralleles Lesen der paarweisen Alignments ist kein Problem.
    3. Min-Cuts können innerhalb jeder Zusammenhangskomponente parallelisiert berechnet werden.
    4. Bei der alten Methode können auch die kürzesten Pfade durch den Sukzessionsgraphen parallelisiert werden, bei den beiden neuen Ansätzen nicht. Das ist aber nicht so schlimm, weil dieser Abschnitt die Laufzeit nicht dominiert.
- Ersetzen der Felder für die *Waisen* durch eine bessere Datenstruktur
- Hinzufügen von Kanten im *Alignmentgraphen* für ganze *Fragmente*

Literaturverzeichnis

Abdeddaïm, S. On Incremental Computation of Transisitve Closure and Greedy Alignment. In *Pattern Matching Algorithms*, Seiten 168–179. Oxford University Press (1997).

Abdeddaïm, S. und Morgenstern, B. Speeding up the DIALIGN multiple alignment program by using the 'Greedy Alignment of BIOlogical Sequences LIBrary' (GABIOS-LIB. In *Computational Biology: First International Conference on Biology, Informatics, and Mathematics*, Seiten 1–11 (2000).

Corel, E., Pitschi, F. und Morgenstern, B. A *min-cut* algorithm for the consistency problem in multiple sequence alignment. *Bioinformatics*, Band 26:8:1015–1021 (2010).

Henikoff, S. und Henikoff, J. Amino acid substitution matrices from protein blocks. In *Proceedings of the National Academy of Sciences, USA*, Band 89:22, Seiten 10915–10919. Natl. Acad. Sci. USA (1992).

Morgenstern, B. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, Band 15:3:211–218 (1999).

Morgenstern, B. A Simple and Space-Efficient Fragment-Chaining Algorithm for Alignment of DNA and Protein Sequences. *Applied Mathematics Letters*, Band 15:1:11–16 (2002).

Morgenstern, B., Atchley, W., Hahn, K. und Dress, A. Segment-based scores for pairwise and multiple sequence alignments. In *ISMB-98 Proceedings*, Seiten 115–121. AAAI (1998).

Morgenstern, B., Dress, A. und Werner, T. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. In *Proceedings of the National Academy of Sciences, USA*, Band 93, Seiten 12098–12103. Natl. Acad. Sci. USA (1996).

Morgenstern, B., Prohaska, S., D., P. und Stadler, P. Multiple sequence alignment with user-defined anchor points. *Algorithms for Molecular Biology*, Band 1:6 (2006).

Pearson, W. Selecting the Right Similarity Matrix. *Curr Protoc Bioinformatics*, Band 43:3.5:1–9 (2013).

Subramanian, A., Kaufman, M. und Morgenstern, B. DIALIGN TX download. `http://dialign-tx.gobics.de/download` (2008). Accessed: 2018-03-31.

Vingron, M. und Sibbald, P. Weighting in sequence space: a comparison of methods in terms of generalized sequences. In *Proceedings of the National Academy of Sciences, USA*, Band 90:19, Seite 8777–8781. Natl. Acad. Sci. USA (1993).

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über *„Titel“* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Vorname Nachname, Münster, 17. April 2018

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Vorname Nachname, Münster, 17. April 2018