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, 30. 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 $p \leq p'$.

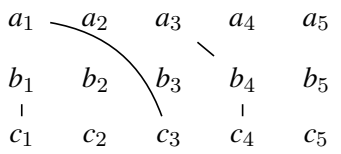
Nachdem wir bis jetzt nur umgangssprachlich mit *Alignments* und *Konsistenz* zu tun hatten, möchte ich diese Begriffe nun formalisieren.

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 $\mathcal{X}$.

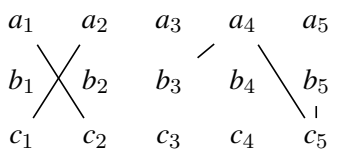
Wir bezeichnen $\mathcal{R}$ als *konsistent*, wenn $\preceq_{\mathcal{R}} = (\preceq \cup \mathcal{R})_t$ die natürliche Ordnung auf jeder Sequenz erhält, also $x \preceq_{\mathcal{R}} y \implies x \preceq y$ für alle $x, y \in S_i \forall 1 \leq i \leq n$ gilt. Außerdem nennen wir eine Menge von Relationen $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ *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 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 $[x]_{\mathcal{A}} = \{y \in \mathcal{S} : x \mathcal{A} y\}$ 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:

$$\begin{array}{ccccccc} - & - & A_1 & a_2 & A_3 & a_4 & a_5 \\ & & \vert & & \vert & & \\ b_1 & B_2 & \vert & b_3 & B_4 & b_5 & - \\ & \vert & \vert & & \vert & & \\ c_1 & C_2 & C_3 & - & C_4 & c_5 & - \end{array}$$

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

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

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 $f_{i,j,l}$:

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

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* bezeich-

nen 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 (Subra-

manian *et al.*, 2008), wird diese Technik jedoch angewendet. Dort dient eine modifizierte BLOSUM 62, die nur nicht-negative 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 tendenziell 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 berechnen, erfahren wir, indem das Maximum von ihnen bestimmt wird.

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

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* berechnen zu können, sondern auch dieses *Alignment* selbst, müssen wir zunächst noch einige Definitionen ein-

führen. Zuerst 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.

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

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 *Fragmenten*, 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 erstellte 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. Dazu nehmen wir eine Substitutionsmatrix mit $+3$ für Übereinstimmungen und -1 für Abweichungen. Der Korrekturterm $K = \ln(L_i) + \ln(L_j)$ aus unserer Gewichtsfunktion w^* beträgt für alle Sequenzpaare gerundet 4.

	$Sc[i,j]$		$Pr[i,j]$		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	adgTCTCA---	10	1	aDGTC---TCa	7
3	---TATCAdgg		4	-DGTCadaTC-	
2	-gTCADctca	8	2	-GTCADCTCa	16
3	taTCADgg--		4	dGTCADATC-	
3	taTCADgg-	8			
4	dgTCADatc				

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.

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

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*:

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

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*.

2	$A_{2,1}$		
3	$A_{3,1}$	$A_{3,2}$	
4	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$
i / j	1	2	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* $\binom{\text{CTCA}}{\text{ATCA}}$ 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 wei-

testen 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.

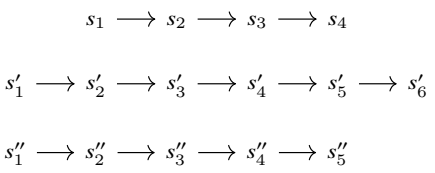


Abbildung 3.1: Graph dreier Sequenzen mit den durch sie induzierten Pfaden

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 $|V| = v$, 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 $O(k^2 \cdot (\mu - \mu_0) + v \cdot \min\{v, (\mu - \mu_p)\})$ Zeit und mit $O(k \cdot v)$ 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

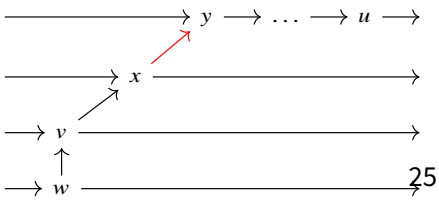


Abbildung 3.2: Füge (x, y) ein.

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.

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.

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. 3.2). 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

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
```

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 $O(v \cdot \min\{v, \mu - \mu_p\})$. Fasst man alles zusammen, so folgt die behauptete Laufzeit von $O(k^2 \cdot (\mu - \mu_0) + v \cdot \min\{v, \mu - \mu_p\})$. □

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

Beispiel

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*.

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*.

„⇐“: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.

„⇒“:(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.

„⇐“: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 $\mathrm{nextClass}[x] = p'$ 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

Sobald Beispiele für Transitivitätsgrenzen gegeben sind, hier darauf verweisen

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.

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

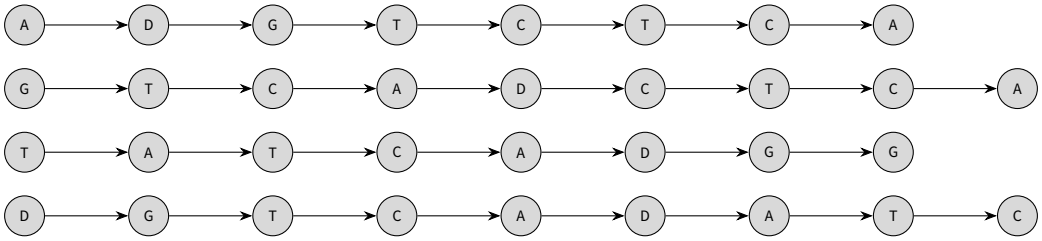
Für die *Konsistenzgrenzen* brauchen wir in jedem Schritt unseren *Alignmentgraphen*, alle Äquivalenzklassen unseres *Alignments*, deren *Vorgänger-* und *Nachfolgergrenzen* und die Felder $\mathrm{nextClass}$ und $\mathrm{prevClass}$ für alle *Waisen*. Der Übersichtlichkeit halber gebe ich nicht die 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.

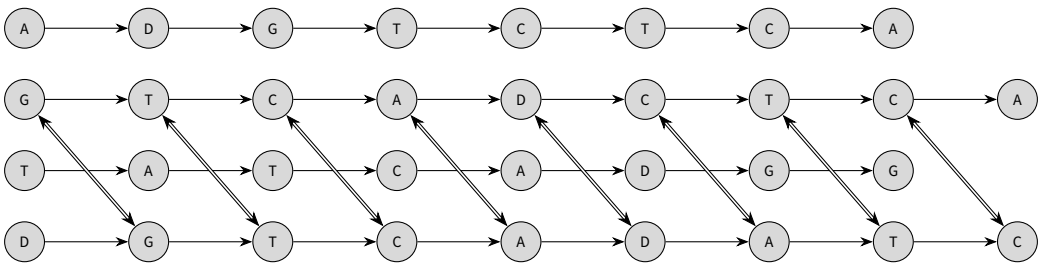
Ursprungszustand

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.



Erstes Fragment

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*.



Um zu verdeutlichen, dass durch doppelte Kanten verbundene Knoten im Sinne unseres *Alignments* äquivalent sind, wurden diese Verbindungen mit Äquivalenzpfeilen dargestellt.

Bezeichner	$[(2,1)]_{\mathcal{A}}$	$[(2,2)]_{\mathcal{A}}$	$[(2,3)]_{\mathcal{A}}$	$[(2,4)]_{\mathcal{A}}$
Äqu.klasse	$\{(2,1),(4,2)\}$	$\{(2,2),(4,3)\}$	$\{(2,3),(4,4)\}$	$\{(2,4),(4,5)\}$
	$[(2,5)]_{\mathcal{A}}$	$[(2,6)]_{\mathcal{A}}$	$[(2,7)]_{\mathcal{A}}$	$[(2,8)]_{\mathcal{A}}$
	$\{(2,5),(4,6)\}$	$\{(2,6),(4,7)\}$	$\{(2,7),(4,8)\}$	$\{(2,8),(4,9)\}$

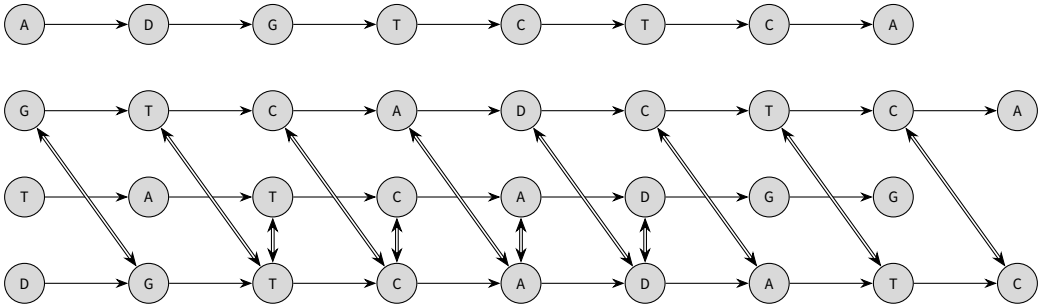
Bei den *Waisen* werden nur die *Stellen* beeinflusst, die in den Sequenzen liegen, die bereits ein *Fragment* enthalten, aber nicht selbst Teil davon sind.

	$Pred((i,p),j)$				$Succ((i,p),j)$				$\mathrm{prevClass}$		$\mathrm{nextClass}$	
(i,p) \ j	1	2	3	4	1	2	3	4	(i,p)	$P((i,p))$	(i,p)	$S((i,p))$
$[(2,1)]_{\mathcal{A}}$	0	1	0	2	9	1	9	2	$(2,9)$	8	$(4,1)$	1
$[(2,2)]_{\mathcal{A}}$	0	2	0	3	9	2	9	3				
$[(2,3)]_{\mathcal{A}}$	0	3	0	4	9	3	9	4				
$[(2,4)]_{\mathcal{A}}$	0	4	0	5	9	4	9	5				
$[(2,5)]_{\mathcal{A}}$	0	5	0	6	9	5	9	6				
$[(2,6)]_{\mathcal{A}}$	0	6	0	7	9	6	9	7				
$[(2,7)]_{\mathcal{A}}$	0	7	0	8	9	7	9	8				
$[(2,8)]_{\mathcal{A}}$	0	8	0	9	9	8	9	9				

Wie nicht anders zu erwarten, wurden die *Vorgänger* und *Nachfolger* für alle Sequenzen, die nicht mit einem *Fragment* verbunden sind, auf 0 beziehungsweise die Länge der Sequenz plus eins gesetzt. Bei allen *Ankern* ist der Eintrag für die jeweilige Sequenz genau die Position der an dieser Äquivalenzklasse beteiligten *Stelle*.

Zweites Fragment

Als nächstes fügen wir das *Fragment* $\binom{\mathtt{TCAD}}{\mathtt{TCAD}}$ aus der dritten und vierten Sequenz ein. Weil die dritte Sequenz noch nicht Teil des multiplen *Alignments* ist, kann es auch hier zu keinen *Inkonsistenzen* kommen.



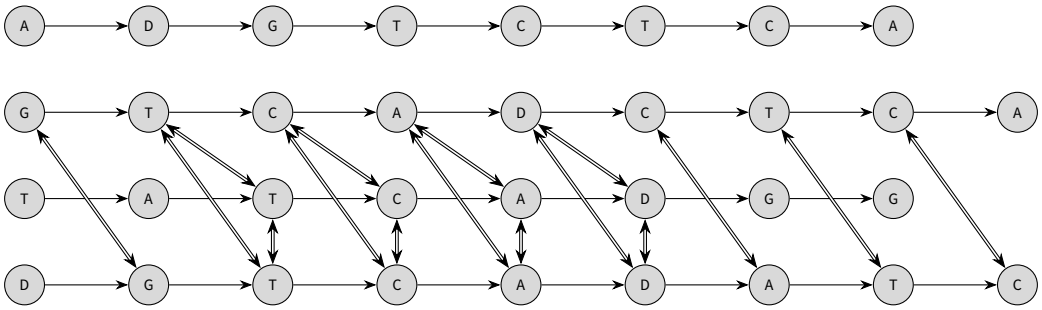
Bezeichner	$[(2,1)]_{\mathcal{A}}$	$[(2,2)]_{\mathcal{A}}$	$[(2,3)]_{\mathcal{A}}$	$[(2,4)]_{\mathcal{A}}$
Äqu.klasse	$\{(2,1),(4,2)\}$	$\{(2,2),(3,3),(4,3)\}$	$\{(2,3),(3,4),(4,4)\}$	$\{(2,4),(3,5),(4,5)\}$
	$[(2,5)]_{\mathcal{A}}$	$[(2,6)]_{\mathcal{A}}$	$[(2,7)]_{\mathcal{A}}$	$[(2,8)]_{\mathcal{A}}$
	$\{(2,5),(3,6),(4,6)\}$	$\{(2,6),(4,7)\}$	$\{(2,7),(4,8)\}$	$\{(2,8),(4,9)\}$

Zu aktualisieren sind die *Transitivitätsgrenzen* für die dritte Sequenz und die Prev- und $\mathrm{NextClass}$-Einträge der *Stellen* von S_3, die vor oder hinter unserem *Fragment* liegen. Da diese jeweils am Start oder Ende positioniert sind, ist nur einer der beiden Einträge relevant.

	$Pred((i,p),j)$				$Succ((i,p),j)$				prevClass		nextClass	
(i,p) \ j	1	2	3	4	1	2	3	4	(i,p)	$P((i,p))$	(i,p)	$S((i,p))$
$[(2,1)]_{\mathcal{A}}$	0	1	0	2	9	1	3	2	$(2,9)$	8	$(4,1)$	1
$[(2,2)]_{\mathcal{A}}$	0	2	3	3	9	2	3	3	$(3,7)$	6	$(3,1)$	3
$[(2,3)]_{\mathcal{A}}$	0	3	4	4	9	3	4	4	$(3,8)$	6	$(3,2)$	3
$[(2,4)]_{\mathcal{A}}$	0	4	5	5	9	4	5	5				
$[(2,5)]_{\mathcal{A}}$	0	5	6	6	9	5	6	6				
$[(2,6)]_{\mathcal{A}}$	0	6	6	7	9	6	9	7				
$[(2,7)]_{\mathcal{A}}$	0	7	6	8	9	7	9	8				
$[(2,8)]_{\mathcal{A}}$	0	8	6	9	9	8	9	9				

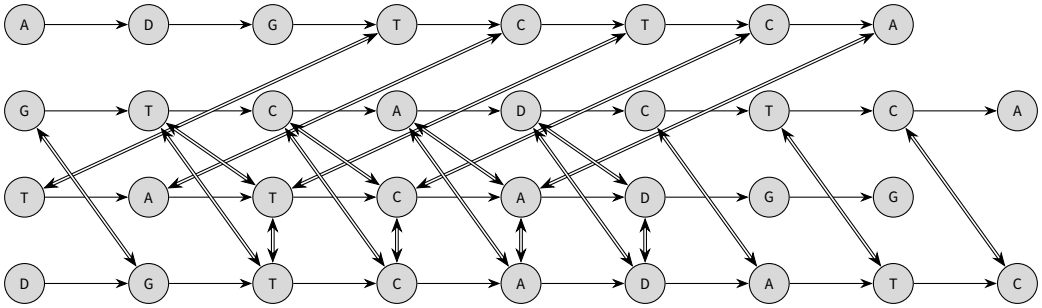
Drittes Fragment

In diesem Schritt fügen wir das *Fragment* $\binom{\texttt{TCAD}}{\texttt{TCAD}}$ aus S_2 und S_3 ein. Hier fügen wir nur zusätzliche Kanten in bereits existierende *Anker* ein. Das hat auch zur Folge, dass sich die *Vorgänger-* und *Nachfolgergrenzen* nicht verändern können, weil bereits in beide Richtungen Kanten existierten (s. Algorithmus 3).



Viertes Fragment

Nun fügen wir das *Fragment* $\binom{\texttt{TCTCA}}{\texttt{TATCA}}$ aus der ersten und dritten Sequenz ein. Auch hier kann es wieder zu keinen *Inkonsistenzen* kommen, weil S_1 noch nicht Teil des *Alignments* war.



Bezeichner Äqu.klasse	$[(2,1)]_{\mathcal{A}}$ $\{(2,1),(4,2)\}$	$[(2,2)]_{\mathcal{A}}$ $\{(2,2),(3,3),(4,3),(1,6)\}$	$[(2,3)]_{\mathcal{A}}$ $\{(2,3),(3,4),(4,4),(1,7)\}$
	$[(2,4)]_{\mathcal{A}}$ $\{(2,4),(3,5),(4,5),(1,8)\}$	$[(2,5)]_{\mathcal{A}}$ $\{(2,5),(3,6),(4,6)\}$	$[(2,6)]_{\mathcal{A}}$ $\{(2,6),(4,7)\}$
	$[(2,7)]_{\mathcal{A}}$ $\{(2,7),(4,8)\}$	$[(2,8)]_{\mathcal{A}}$ $\{(2,8),(4,9)\}$	$[(1,4)]_{\mathcal{A}}$ $\{(1,4),(3,1)\}$
	$[(1,4)]_{\mathcal{A}}$ $\{(1,4),(3,1)\}$		

.

	$Pred((i,p),j)$				$Succ((i,p),j)$				prevClass		nextClass	
(i,p) \ j	1	2	3	4	1	2	3	4	(i,p)	$P((i,p))$	(i,p)	$S((i,p))$
$[(2,1)]_{\mathcal{A}}$	0	1	0	2	6	1	3	2	$(2,9)$	8	$(4,1)$	1
$[(2,2)]_{\mathcal{A}}$	6	2	3	3	6	2	3	3	$(3,7)$	6	$(3,1)$	3
$[(2,3)]_{\mathcal{A}}$	7	3	4	4	7	3	4	4	$(3,8)$	6	$(3,2)$	3
$[(2,4)]_{\mathcal{A}}$	8	4	5	5	8	4	5	5			$(1,1)$	4
$[(2,5)]_{\mathcal{A}}$	0	5	6	6	9	5	6	6			$(1,2)$	4
$[(2,6)]_{\mathcal{A}}$	0	6	6	7	9	6	9	7			$(1,3)$	4
$[(2,7)]_{\mathcal{A}}$	0	7	6	8	9	7	9	8				
$[(2,8)]_{\mathcal{A}}$	0	8	6	9	9	8	9	9				
$[(1,4)]_{\mathcal{A}}$	4	0	1	0	4	2	1	3				
$[(1,5)]_{\mathcal{A}}$	5	0	2	0	5	2	2	4				

Weitere Fragmente

Die restlichen *Fragmente* unserer paarweisen *Alignments* sind leider nicht *konsistent* zu unserem bisherigen multiplen *Alignment*. Betrachtet man $\binom{\texttt{GT}}{\texttt{Gt}}$ zwischen S_1 und S_2, das als drittnächstes an der Reihe ist, stellt man fest, dass die jeweiligen *Stellen* nicht *alignierbar* sind. Für $(1,2) = (\texttt{G})$ ist das Alignieren noch möglich, denn $\mathrm{NextClass}((1,3)) = 4$ und $Succ((1,4),2) - 1 = 1$. Für $(1,3) = (\texttt{T})$ gilt das nicht mehr, denn $Succ((1,4),2) - 1 = 1 < 2$. Für die anderen drei übrigen *Fragmente* gilt analoges.

3.6 Abschluss des Verfahrens

Nach diesem ersten Durchlauf unseres Ansatzes sind wir jedoch noch nicht fertig. Es kann beispielsweise sein, dass wir ein wichtiges Motiv haben, dass in mehreren Sequenzen vorkommt und somit Teil unseres multiplen *Alignments* sein sollte. Bei zwei Sequenzen war es jedoch nicht im paarweisen *Alignment* enthalten, weil stattdessen ein zufälliges, aber größeres *Fragment* ausgewählt wurde. Wenn wir Glück haben, findet dieses aber nicht den Weg in unser multiples *Alignment*, weil es im Vergleich zu anderen *Fragmenten* mit denen es *inkonsistent* war ein geringeres Gewicht hat. Deshalb führen wir zwischen allen Teilsequenzen, die durch unsere *Ankerpunkte* vorgegeben sind, unser DIALIGN-Verfahren nochmal durch und wiederholen dies, bis es keine neuen *Fragmente* mit positivem Gewicht mehr gibt (Morgenstern *et al.*, 1996). In der Praxis stellt man fest, dass es in der Regel zu höchstens drei Iterationen kommt (Morgenstern, 1999). Wir können daher annehmen, dass die Anzahl an Durchläufen unseres Verfahrens konstant ist und für die Laufzeit nicht weiter in Betracht gezogen werden muss.

Denken wir an unser Beispiel zurück, dann stellen wir fest, dass das Fragment $f_{3,2,2} = \binom{\texttt{DG}}{\texttt{DG}}$ aus der ersten und vierten Sequenz das positive Gewicht 2 hat. Es kann also unter Aktualisierung unseres *Alignmentgraphen* in unser multiples *Alignment* aufgenommen werden. Weitere geeignete *Fragmente* der Teilsequenzen gibt es nicht. Bei der nächsten Iteration stellt man demnach fest, dass kein neues *Fragment* ein positives Gewicht hat und wir können das Alignieren abbrechen.

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).

Alignmentgraphen für das Einfügen von Lücken benutzen?

Das finale *Alignment* unserer vier Beispielsequenzen mit DIALIGN 2.2 und einer $+3/-1$-Substitutionsmatrix sieht dann so aus:

```
1. aDGTCTCA-----
2. --G--TCADCTCa
3. ---TATCADgg--
4. -DG--TCADATC-
```

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. Wir gehen davon aus, dass die Anzahl an Iterationen wie oben beschrieben in $O(1)$ liegt. Es folgt, dass die Laufzeit einer Iteration asymptotisch auch der Gesamtlaufzeit entspricht.

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. Mein Verbesserung dieses Schrittes ist an dieser Stelle wichtig, weil sonst der naive Ansatz mit Laufzeit $O(n^4 \cdot L^2)$ die Gesamtkomplexität dominiert hätte.

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 der benötigte Speicherplatz von $O(N_{\max} + n^2 \cdot L)$. □

3.7 Evaluierung, Zusammenfassung und Schwächen des Ansatzes

3.7.1 Evaluierung

DIALIGN wurde im Laufe der Zeit und nach jeder neuen Verbesserung ausführlichen Tests und Vergleichen zu anderen Programmen für multiple *Sequenzalignments* unterzogen. Diese möchte ich im Folgenden kurz zusammenfassen.

DIALIGN 1.0 wurde zunächst an einem Satz von elf DNA-Sequenzen getestet, die jeweils Helix-Loop-Helix-Bindungstellen für Proteine haben (Morgenstern *et al.*, 1996). Diese Bindungsstellen mit einer Länge von etwa 30 DNA-Basen wurden experimentell gefunden und außerhalb dieser Regionen existieren keine erkennbaren Ähnlichkeiten zwischen den Sequenzen. Während DIALIGN in der Lage war alle elf Sequenzen korrekt miteinander zu alignieren, waren das Optimum der anderen Verfahren zwei korrekte Zuordnungen. Die verglichenen Programme waren dabei DFALIGN, PILEUP, CLUSTAL und GENALIGN. Auch bei den Vergleichen mit elf Programmen zum alignieren von Proteinsequenzen schnitt DIALIGN neben CLUSTAL V und DFALIGN unter den besten dreien ab.

Es ist sinnvoll bei zu alignierenden Sequenzen zwischen lokal und global verwandten zu unterscheiden. Lokal verwandte haben nur Ähnlichkeiten in begrenzten Abschnitten, während der Rest höchstens zufällige Übereinstimmungen enthält. Hier ist es optimal lediglich die ähnlichen Segmente ins *Alignment* aufzunehmen und die nicht verwandten zu ignorieren. Global verwandte Sequenzen haben hingegen Ähnlichkeiten, die sich über die volle Länge der Sequenzen ziehen und wo es nur vereinzelt zu Deletionen, Insertionen oder Punktmutationen kommt.

Beim Testen von DIALIGN 2.0 an verschiedenen Testsequenzen hat man festgestellt, dass DIALIGN 1 und 2.0 auf global verwandten Sequenzen vergleichbar zu globalen Alignern wie CLUSTAL W und DCA zu sein scheinen (Morgenstern *et al.*, 1998). Bei lokal verwandten Sequenzen war DIALIGN 2.0 allen anderen getesteten Programmen hingegen weit überlegen. Ein weiterer Vorteil ist, dass DIALIGN 2.0 anders als der Vorgänger und viele der anderen Programme unabhängig von benutzerdefinierten Eingaben war.

Da große *Alignments* sehr rechenintensiv sein können, ist es außerdem wichtig die Laufzeit der Verfahren im Auge zu behalten. Progressive Alignierer wie beispielsweise CLUSTAL W starten damit nach dem Berechnen paarweiser *Alignments* zunächst die beiden ähnlichsten Sequenzen zu verwenden und dann der Reihe nach die jeweils ähnlichste Sequenz ins multiple *Alignment* einzufügen, bis alle Sequenzen Teil der Zuordnung sind. Die erste Version von DIALIGN hatte eine Laufzeit, die etwa um den Faktor 100 langsamer war, als vergleichbare globale Alignierer. Durch die Verwendung des graphtheoretischen Ansatzes für die *Konsistenzgrenzen* von Abdeddaïm konnte dieser Unterschied in etwa um den Faktor zehn gesenkt werden (Abdeddaïm und Morgenstern, 2000). Es ist denkbar, dass die speichereffiziente Berechnung der paarweisen *Alignments* die Lücke weiter geschlossen hat, indem aufgrund des geringeren Speicherverbrauchs weniger Seitenfehler und damit teure I/O-Operationen auftreten.

3.7.2 Zusammenfassung

Wir haben in diesem Kapitel den DIALIGN-Ansatz für multiple *Sequenzalignments* kennengelernt. DIALIGN erstellt multiple *Alignments*, indem zuerst mit einem speichereffizienten Algorithmus über dynamische Programmierung paarweise *Alignments* berechnet

werden. Mit Hilfe vom parallelen Traversieren über sortierte Listen können wir dann effektiv die Überschneidungen der *Fragmente* finden und diese damit neu gewichten. Nachdem wir unsere *Fragmente* nach ihren Gewichten sortiert haben, fügen wir sie der Reihe nach in unser multiples *Alignment* und unseren *Alignmentgraphen* ein, vorausgesetzt die *Konsistenz* wird durch sie nicht verletzt. Die so entstandenen *Anker* wählen wir dann als Grenzen für Teilsequenzen und führen zwischen diesen iterativ das selbe Verfahren solange durch, bis es keine *Fragmente* mit positiven Gewichten mehr gibt. Zum Schluss konstruieren wir die Ausgabe, indem wir nichtalignierte Symbole klein und alignierte groß schreiben, sowie Lücken in unsere Sequenzen einfügen, sodass alle *Stellen* eines *Ankers* genau in einer Spalte stehen.

DIALIGN aligniert nur die Segmente der Sequenzen miteinander, die auch wirklich ähnlich zueinander sind, während globale Alignierer selbst dann versuchen die ganzen Sequenzen einander zuzuordnen, wenn es nur lokale Übereinstimmungen gibt. Das bietet einen großen Vorteil bei lokal verwandten Sequenzfamilien. Viele dezidierte Algorithmen für lokale *Alignments*, wie beispielsweise der auf Needleman-Wunsch basierende Smith-Waterman-Algorithmus, sind nur in der Lage eine einzige Region mit Übereinstimmungen zu finden. DIALIGN hingegen kann auch mehrere weit voneinander entfernte lokale Übereinstimmungen finden oder gleich globale *Alignments* mit angemessener Genauigkeit berechnen (Morgenstern *et al.*, 1996). In diesem Sinne ist es ein sehr flexibler Algorithmus, der für eine Vielzahl an Anwendungen angemessen ist.

Erinnern wir uns zurück an die gängigsten Mutationen, die auf den von uns untersuchten Sequenzen vorkommen: Deletionen und Insertionen von ganzen Abschnitten oder Punktmutationen, bei denen einzelne Basen oder Aminosäuren durch andere ersetzt werden. Man stellt fest, dass jede dieser Situationen sich in DIALIGN wiederfindet. Punktmutationen zeigen sich durch Abweichungen einzelner *Stellen* in längeren *Fragmenten* und mit gelöschten oder eingefügten Segmenten gehen wir um, indem ein einzelnes längeres *Fragment* in zwei oder mehr kürzere aufgeteilt wird (Morgenstern *et al.*, 1997). Es ist außerdem anzunehmen, dass der segmentbasierte Ansatz von DIALIGN insofern für die Benutzer angenehmer ist, als dass nicht auf Biegen und Brechen versucht wird die kompletten Sequenzen in ein *Alignment* einzufügen, selbst wenn es keine Ähnlichkeit gibt. Man stellt fest, dass es für einen Alignierer fast genauso wichtig ist, nichtverwandte Abschnitte auch nicht zuzuordnen, wie es wichtig ist, dies für verwandte zu tun. Denn sonst muss der Forscher, der das *Alignment* später benutzt, erst mühsam von Hand feststellen, welche Abschnitte eigentlich die gesuchten Motive sind (Morgenstern, 1999).

3.7.3 Schwächen von DIALIGN

Es gibt im Allgemeinen zwei Gründe, warum automatisierte *Alignmentverfahren* wie beispielsweise DIALIGN biologisch unzureichende Ergebnisse liefern können: Entweder die Gütefunktion, also in unserem Fall der *Score* unseres *Alignments*, bildet die Wirklichkeit unzureichend ab und weist bedeutungslosen Abschnitten größere Werte zu, als anderen wichtigen Motiven. Oder aber die Gütefunktion ist grundsätzlich richtig, aber dafür werden beim zusammensetzen des multiplen *Alignments* die falschen *Fragmente* ausgewählt. In diesem Fall liegt das Problem bei unserer Heuristik, die gierig die *Fragmente* nach Gewicht und *Konsistenz* auswählt.

Um festzustellen wie sich DIALIGN weiter verbessern lässt, muss man zuerst die Güte unserer Ergebnisse quantifizieren und dann überprüfen, ob die obigen Situationen

auftreten. Bei der Quantifizierung gibt es zunächst zwei verschiedene Möglichkeiten, wie man feststellen kann, ob und wie gut ein *Alignment* wirklich ist. Als erstes brauchen wir dafür fertige *Alignments* zum Vergleich von denen wir wissen, dass sie korrekt sind. Solche finden sich beispielsweise in Testdatenbanken wie BAliBASE, die eine Vielzahl von Zuordnungen auf unterschiedlichen Klassen von Sequenzen bieten. Das erste Gütekriterium ist der *sum-of-pairs-Score*, der in Prozent angibt wie viele paarweise korrekte Übereinstimmungen es im Vergleich zur Referenz gab. Die zweite Möglichkeit ist der *column-Score*, der überprüft wie viele ganze Spalten korrekt *aligniert* wurden (Morgenstern *et al.*, 2006). Grundsätzlich ist letzterer das bessere Maß, weil wir Ähnlichkeiten über die komplette Menge an Sequenzen hinweg finden wollen. Es kann jedoch passieren, dass in beinahe jeder Spalte eine einzige fehlerhafte Zuordnung auftritt, was in einem sehr niedrigen *column-Score* resultieren würde, obwohl das *Alignment* eigentlich nicht so schlecht ist.

Morgenstern *et al.* (2006) haben DIALIGN auf einer Vielzahl von Sequenzen untersucht, um festzustellen welche Probleme beim *Alignieren* auftreten. Die schlechte Nachricht: beide der oben genannten treten auf, abhängig von den zu *alignierenden* Sequenzen. Das erste *Testalignment* war eine Menge von Genen des Pufferfischs *Takifugu rubripes*. Diese enthalten sogenannte *Hoxgene*, die die Ausbildung von Vorder- (Kopf) und Hinterteilen (Schwanz) bei der Klasse der *Bilateria* regulieren. Zu den *Bilateria* gehören alle Tiere, die einen bilateralsymmetrisch aufgebauten Körper haben, also beispielsweise alle Säugetiere, Würmer, Fische, Fische oder Amphibien. Die *Hoxgene* sind evolutionär gesehen sehr alt und ihre Geschichte wurde durch viele *Tandemduplikate* dominiert. *Tandemduplikate* sind gleiche oder sehr ähnliche Abschnitte, die an mehreren Stellen innerhalb der Sequenzen vorkommen. Im schlimmsten Fall kommt es dann dazu, dass die falschen *Hoxgene* miteinander *aligniert* werden. Weil diese sehr ähnlich sind, ist es wahrscheinlich, dass sie einander zugewiesen werden und die *Alignments* hohe Gewichte haben. Biologisch sind diese Zuweisungen aber von geringem Wert, weil es eben nicht die richtigen Paare von *Hoxgenen* in unserem *Alignment* waren. Genau diese Erfahrung hat man auch bei *Takifugu rubripes* gemacht. Das DIALIGN-Ergebnis hatte zwar einen um 13% höheren *Score*, als das korrekte Ergebnis, der *column-Score* lag dagegen bei 0% und selbst die Residuenpaare waren nur zu 33% richtig. Ergebnisse wie dieses sind ein klares Ergebnis dafür, dass die Gütefunktion von DIALIGN in manchen Fällen versagt. Bei diesem extremen Beispiel ist es aufgrund der geringeren Korrelation zwischen der mathematischen und der biologischen Ähnlichkeit der *Hoxgene* schwer eine entsprechende Gütefunktion zu finden. Mögliche Ansätze können Machine Learning oder semiautomatisierte Verfahren sein, wo ein Mensch mit Expertenwissen bestimmte Abschnitte von Hand zuweist (Verankerung [1]) und lediglich die Abschnitte dazwischen maschinell *aligniert* (Morgenstern *et al.*, 2006). Man hat jedoch auch weitere Sequenzfamilien in BAliBASE gefunden, bei denen der *Score* bei schlechteren *Alignments* stieg. In diesem Fall kann man über bessere Gütefunktionen nachdenken, weil bessere Heuristiken das Problem nicht lösen. Schließlich basieren sie auf den Ergebnissen der anscheinend unzureichenden Gütefunktionen.

Bei anderen Sequenzen hat man hingegen festgestellt, dass die korrekten *Alignments Scores* haben können, die bis zu 15% über den Ergebnissen von DIALIGN liegen. In diesem Fall ist das Problem vermutlich die Heuristik, die gierig die *Fragmente* mit den höchsten Gewichten wählt. Wenn man Pech hat läuft man so in ein lokales Maximum, indem man *Fragmente* wählt, die hohe Gewichte, aber geringen biologischen Wert haben und über *Inkonsistenzen* verhindern, dass andere bedeutungsvollere Zuweisungen gewählt wer-

[1] Im englischen Original auch „anchor“. Um Verwirrung mit unseren *Ankern* vorzubeugen hier umbenannt.

den. Auch mathematisch kann der gierige Ansatz suboptimale Ergebnisse liefern, indem ein einziges großes *Fragment* das Hinzufügen mehrerer kleiner verhindert, die summiert ein größeres Gesamtgewicht hätten. Weil das nicht optimale, große *Fragment* nie wieder aus unserem multiplen *Alignment* entfernt werden kann, sind wir auch nicht in der Lage die anderen Segmente auszuwählen (Morgenstern, 1999).

Um diesem Problem vorzubeugen lernen wir im nächsten Kapitel eine neue, verbesserte Heuristik für DIALIGN kennen. Das Verfahren von Corel *et al.* (2010) basiert auf Flussnetzen und weiteren graphtheoretischen Ansätzen, die eine flexiblere Auswahl der *Anker* unseres multiplen *Alignments* ermöglichen.

4 Ein Min-Cut-Ansatz für das Konsistenzproblem

Da es, wie im letzten Kapitel gezeigt, auf manchen Sequenzfamilien durch die gierige Heuristik von DIALIGN zu suboptimalen *Scores* und *Alignments* kommt, werden wir jetzt einen verbesserten graphtheoretischen Ansatz von Corel *et al.* (2010) betrachten.

Dazu benötigen wir zwei verschiedene Graphen: zum einen den *Inzidenzgraphen*, bei dem alle *Stellen* Knoten sind und ihre *Anker* Zusammenhangskomponenten. Der zweite ist der *Sukzessionsgraph*, der die Zusammenhangskomponenten unseres *Inzidenzgraphen* als Knoten und die natürliche Ordnung auf den Sequenzen als Kanten benutzt. Man kann sich vorstellen, dass genau dann eine Kante zwischen zwei Diese beiden Datenstrukturen werden wir benutzen, um *Inkonsistenzen* aufzulösen. Wenn wir uns an die Definition von *Konsistenz* aus dem letzten Kapitel erinnern, dann stellen wir fest, dass es zwei Arten von ihr gibt: zum einen implizite, transitive Mehrfachzuweisungen bei denen einer *Stelle* einer Sequenze mehrere *Stellen* einer anderen Sequenz zugeordnet sind und zum anderen überkreuzte Zuweisungen.

Als Ausgangspunkt starten wir wieder mit unseren paarweisen *Alignments* aus DIALIGN. *Überlappgewichte* brauchen wir in unserem Fall nicht. Dann konstruieren wir mit Hilfe dieser Zuweisungen unseren *Inzidenzgraphen* und benutzen einen Algorithmus zur Berechnung des minimalen Schnitts („min-cut“) auf den Zusammenhangskomponenten, um alle *Inkonsistenzen* aufgrund von transitiven Mehrfachzuweisungen aufzulösen. Die so entstehenden Zusammenhangskomponenten benutzen wir, um einen *Sukzessionsgraphen* aufzubauen. Dank eines Algorithmus von Pitschi *et al.* (2010) können wir mit diesem Überkreuzungen aus unserer Relation löschen. Alle dieser Konzepte werden wir im Laufe dieses Kapitel formal definieren, genauer analysieren und die Korrektheit der Aussagen beweisen.

4.1 Flussnetzwerke

4.1.1 Einführung

Stell dir eine Produktionsstätte vor, wo ein bestimmtes Produkt hergestellt wird und eine Verwendungsstätte bei der das Produkt benötigt wird. Zwischen dem Ort der Produktion und dem Ort der Verwendung gibt es ein Schienennetz über das die Produkte geliefert werden können. Über jeden Abschnitt der Schienen kann aber nur eine bestimmte Anzahl an Waggons geleitet werden. So mag es weniger stark befestigte Strecken geben, wo nur kleinere oder leichtere Züge fahren können und andere gut ausgebaute mit mehreren Gleisen nebeneinander. Die Anzahl an Einheiten, die über einen Abschnitt geleitet werden können, nennt man Kapazität, während Produktionsstätte und Zielort Quelle und Senke heißen. Ziel ist es zu bestimmen wie viele Produktionseinheiten über dieses Netz von der Fabrik zum Verbraucher geliefert werden können.

Neben diesem Problem kann man mit Flussnetzwerken noch viele andere Anwendungen modellieren, zum Beispiel eine chemische Produktionsstraße mit vielen Rohren, die jeweils nur den Durchfluss einer bestimmten Menge erlauben. Oder Stromnetze mit Kraftwerken und Endverbrauchern zwischen denen es Stromleitungen gibt, die jeweils höchstens einen bestimmten Stromfluss erlauben. Für eine Hochspannungsleitung mag dieser sehr hoch und für die Leitungen, die direkt zu den Häusern gehen, sehr klein sein. Stromnetze bieten uns auch eine direkte Analogie, wie mit den Knotenpunkten zwischen den Verbindungen umzugehen ist. An jedem Knoten, außer der Quelle und Senke, wird genauso viel Fluss hinein- wie wieder hinausgeleitet. Es wird nichts gespeichert oder geht verloren. Dieses Konzept, das man *Flusserhaltung* nennt, funktioniert genau wie das erste *Kirchhoffsche Gesetz* bei Strömen (Cormen *et al.*, 2009).

Ursprünglich stammt das Konzept des Flussnetzwerkes aus dem Kalten Krieg und der militärischen Forschung. Es wurde das Schienennetzwerk des Sowjetunion in Osteuropa untersucht, um herauszufinden wie viel Material die UdSSR im Kriegsfall aus dem russischen Kerngebiet nach Mitteleuropa transportieren könnte und welche Strecken man zerstören müsste, um den Nachschub am schnellsten Abzuschneiden. Die Untersuchung bietet uns sogleich einen intuitiven Begriff des *maximalen Flusses* und des *minimalen Schnitts*. Der *maximale Fluss* ist die maximale Anzahl an Einheiten, die von der Quelle zur Senke transportiert werden kann, während der *minimale Schnitt* den „Bottleneck“ des Netzwerkes darstellt, der den Fluss von der Quelle zur Senke minimiert. Wie wir später formal zeigen werden sind diese beiden Werte in einem Flussnetzwerk äquivalent.

Zunächst betrachten wir Flussnetzwerke auf einer formalen Ebene, dann widmen wir uns kurz und skizziert einigen der wichtigsten Algorithmen zum berechnen von *maximalen Flüssen* und danach beweisen wir die Äquivalenz von diesen mit *minimalen Schnitten*. Diese *minimalen Schnitte* brauchen wir später zum Auflösen von *Inkonsistenzen* im *Inzidenzgraphen*.

4.1.1 Definition (Flussnetzwerk)
Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$ bei dem jeder Kante $(u, v) \in E$ eine nicht-negative Kapazität $c(u, v) \geq 0$ zugeordnet ist und bei dem es zwei ausgezeichnete Knoten $s, t \in V$ gibt, die wir *Quelle* und *Senke* nennen.

Der Einfachheit halber nehmen wir im Folgenden an, dass jeder Knoten von G auf einem Pfad von s nach t liegt. Sollte es doch solche Knoten geben, dann wären sie ohnehin nicht relevant, weil kein *Fluss* durch sie von der *Quelle* zur *Senke* geschickt werden kann.

4.1.2 Definition (Fluss)
Sei $G = (V, E)$ mit Kapazitätsfunktion c ein *Flussnetzwerk*. Dann ist definieren wir einen *Fluss* als eine Funktion $f : V \times V \to \mathbb{R}$, die die folgenden beiden Eigenschaften erfüllt:

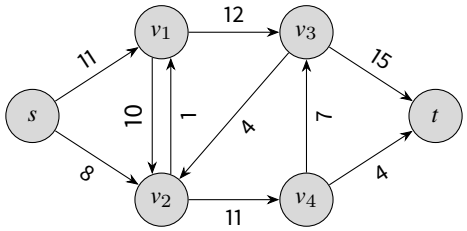
Kapazitätsbeschränkung: Für alle Knoten $u, v \in V$ gelte $0 \leq f(u, v) \leq c(u, v)$.

Flusserhaltung: Für alle Knoten $u \in V \setminus \{s, t\}$ gelte $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

Für Knoten $u, v \in V$, die nicht durch eine Kante verbunden sind ($(u, v) \notin E$), setzen wir den Fluss auf 0: $f(u, v) = 0$.

Ein *Fluss* ist also eine Zuordnung von Werten an die Kanten unseres *Flussnetzwerkes*, sodass keine Kapazität verletzt wird und in jeden Knoten soviel hinein wie hinaus fließt. Wir nennen auch den Wert $f(u, v)$ *Fluss* zwischen den beiden Knoten u und v.

Hier ein Beispiel für ein einfaches *Flussnetzwerk* mit sechs Knoten:

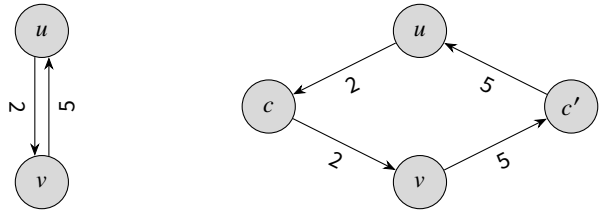


Viele Autoren setzen voraus, dass es keine Kanten in die *Senke* und keine aus der *Quelle* gibt. Diese Beschränkung ist zwar hilfreich fürs Verständnis, wird aber grundsätzlich nicht benötigt. Man kann sich leicht überlegen, dass selbst wenn es solche Kanten gibt, ihr *Fluss* 0 sein muss. Bei den *Flussnetzwerken*, die wir für unseren Algorithmus benutzen, wird es diese Kanten geben, weshalb wir auf die Beschränkung verzichten.

4.1.3 Definition (Wert eines Flusses)
Der *Wert* eines *Flusses* ist definiert als

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s). \tag{4.1}$$

Eine andere Einschränkung, die man in vielen Definitionen sieht, ist der Verzicht auf antiparallele Kanten. Das heißt, dass es gleichzeitig Kanten $(u, v), (v, u) \in E$ gibt. Manche Implementierungen für Algorithmen, die den maximalen Fluss berechnen, sind so programmiert, dass sie vom Benutzer eine Menge von Gegenkanten mit *Fluss* 0 erwarten, die intern benötigt werden. Grundsätzlich sind antiparallele Kanten aber unproblematisch, weil man jeden *Fluss* mit positiven Werten auf antiparallelen Kanten in einen *Fluss* umwandeln kann, bei dem zwischen zwei Knoten höchstens eine Kante einen positiven *Fluss* hat. Sei beispielsweise $f(u, v) = x, f(v, u) = y$ mit $x \not/ 0 \not/ y$. Dann kann man einen *Fluss* mit dem selben *Wert* definieren, wenn man den *Fluss* auf den antiparallelen Kanten wie folgt definiert: $f(u, v) = x - \min\{x, y\}, f(v, u) = y - \min\{x, y\}$. Alternativ lässt sich jedes *Flussnetzwerk* mit antiparallelen Kanten auch in ein äquivalentes übertragen, das keine solchen Kanten enthält.



4.1.2 Wichtige Algorithmen

Es gibt eine Vielzahl von Algorithmen zur Berechnung eines maximalen Flusses auf einem Flussnetzwerk. Die wichtigsten und bekanntesten Vertreter möchte ich kurz vorstellen.

Der erste Algorithmus, der speziell zum berechnen des *maximalen Flusses* entwickelt wurde war der Algorithmus von Ford-Fulkerson (Goldberg und Tarjan, 2014). Dieser verwendete einen sogenannten *Restwertgraphen*, der für jede Kante im ursprünglichen *Flussnetzwerk* eine Vorwärts- und eine Rückwärtskante enthält. Die Vorwärtskante hat als Kapazität die der Kante im ursprünglichen Graph minus den *Fluss* im *Flussnetzwerk*. Die der Rückwärtskante im *Restwertgraphen* ist genau die Kapazität des *Flusses* der zugehörigen Kante im *Flussnetzwerk*. Für einen Pfad von der *Quelle* zur *Senke* im *Restwertgraph* kann man den *Fluss* im *Flussnetzwerk* um die kleinste Kapazität auf diesem Pfad erhöhen, ohne die *Kapazitätsbeschränkung* zu verletzen. Diese Pfade nennt man *augmentierend* und nach jeder Aktualisierung des Flusses, müssen auch die Kapazitäten im *Restwertgraphen* angepasst werden. Der Algorithmus von Ford-Fulkerson besucht solange *augmentierende* Pfade, bis es keine solchen mehr gibt. Ist das der Fall, dann kann der *Fluss* nicht mehr vergrößert werden und der *maximale Fluss* wurde berechnet. Der Algorithmus von Ford-Fulkerson hat für einen zusammenhängenden Graphen mit ganzzahligen Kapazitäten eine pseudopolynomielle Laufzeit von $O(|E| \cdot U)$, wobei U die Summe der Kapazitäten der ausgehenden Kanten von der *Quelle* ist.

Oft wird Ford-Fulkerson eher als eine „Methode“ statt als *Algorithmus* bezeichnet, weil die Reihenfolge nach der *augmentierende* Pfade gewählt werden nicht spezifiziert ist. In der Folge wurde der Algorithmus von Ford-Fulkerson an mehreren Stellen verbessert. Edmond-Karps benutzt eine Breitensuche, um den Pfad mit den wenigsten Knoten von der *Quelle* zur *Senke* zu finden. Dadurch lässt sich die Laufzeit auf $O(|V| \cdot |E|^2)$ verbessern (Cormen *et al.*, 2009). Auch der Algorithmus von Dinic [1] sucht nach kürzesten Pfaden im *Restwertgraphen*. Zusätzlich wird das Konzept von sogenannten *blockierenden Flüssen* benutzt, bei denen jeder $s-t$-Pfad mindestens eine Kante enthält, deren Kapazität komplett ausgereizt ist. Der Algorithmus von Dinic benötigt nur $O(|V|^2 \cdot |E|)$ Rechenschritte (Dinitz, 2006).

Die nächste Klasse von Algorithmen waren die sogenannten *Push-Relabel-Algorithmen*. Diese benutzen bei ihren Schritten statt *Flüssen* nur sogenannten *Präflüsse* bei denen zwar die *Kapazitätsbeschränkung* gilt, die *Flusserhaltung* aber nicht. Das heißt, dass jeder Knoten einen positiven *Überfluss* haben kann, wenn mehr in ihn hinein als hinaus fließt. Jeder Knoten hat einen Index, der seine Höhe im Netz angibt, wobei *Fluss* immer nur von oben nach unten geleitet werden kann. Wie der Name schon sagt, sind die zwei Grundschritte bei allen *Push-Relabel-Algorithmen* die Methoden *Push* und *Relabel*. Bei *Push* verschieben wir den *Überfluss* eines Knotens zu einem tiefer gelegenen Nachbarn. Bei *Relabel* wird hingegen die Höhe eines Knotens vergrößert, wenn er noch *Überfluss* hat, aber alle benachbarten Knoten einen größeren Index haben. Die *Push*- und *Relabelmethoden* werden solange angewendet, bis es keine Knoten mehr gibt auf die sie anwendbar sind (Cormen *et al.*, 2009). Je nachdem wie der nächste zu bearbeitende Knoten ausgewählt wird, kann die Laufzeit stark variieren. Ein generischer Algorithmus, der die Knoten mehr oder minder zufällig auswählt, läuft in $O(|V|^2 \cdot |E|)$. Mit einer FIFO-Warteschlange in dem jeder bearbeitete Knoten, der noch *Überfluss* hat, wieder ans Ende eingereiht wird, lässt

[1] Eigentlich entwickelt vom sowjetischen Forscher Yefim A. Dinitz. Die verbreitete Schreibweise beruht auf einem Übersetzungsfehler. Später wanderte Dinitz nach Israel aus (Dinitz, 2006).

sich die Laufzeit auf $O(|V|^3)$ verbessern (Goldberg und Tarjan, 1988). Der in der Praxis verbreitetste Ansatz benutzt die „Highest-Label"-Regel, bei der alle Knoten in Töpfen nach ihrem Index eingeordnet sind und jeweils ein Knoten mit maximalem Index ausgewählt wird. Hier verbessert sich die Laufzeit auf $O(|V|^2 \cdot \sqrt{|E|})$ Rechenschritte im schlimmsten Fall (Ahuja *et al.*, 1997).

Weiter verbessern ließ sich die asymptotische Laufzeit durch die Verwendung von *dynamischen Bäumen*. *Dynamische Bäume* sind eine Datenstruktur, die die nicht-saturierten Teile von *augmentierenden* Pfaden speichern. Mit Operationen auf dieser Datenstruktur ist es möglich die Laufzeit zu senken. Beschränkt man beispielsweise die maximale Baumgröße und benutzt eine weitere Datenstruktur, dann lässt sich der maximale Fluss mit Hilfe von *blockierenden Flüssen* in $O(|V| \cdot |E| \cdot \log(|V|^2/|E|))$ berechnen (Goldberg und Tarjan, 2014). In der Praxis sind diese Algorithmen aber nicht von Relevanz, weil die verwendete Baumstruktur einen großen konstanten Vorfaktor benötigt. Einer der schnellsten bekannten Algorithmen ist der von Orlin. Er verwendet eine Kombination verschiedener Techniken in unterschiedlichen Situationen und kommt insgesamt auf eine Laufzeit von $O(|V| \cdot |E|)$ (Goldberg und Tarjan, 2014).

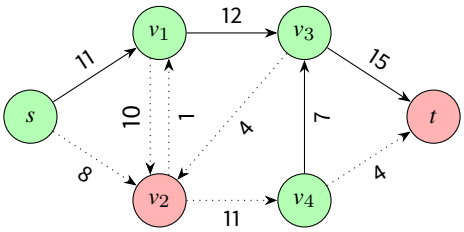
4.1.3 Schnitte und der *Max-Flow-Min-Cut-Satz*

4.1.4 Definition (Schnitt)
Sei $G = (V, E)$ mit *Senke* und *Quelle* $s, t \in V$ und einer Kapazitätsfunktion c. Dann ist ein *Schnitt* von G eine Partitionierung von V in zwei Mengen A und B, sodass $s \in A$ und $t \in B$ gelten.

Die Kapazität $c_{A,B}$ dieses *Schnitts* ist definiert als die Summe der Kapazitäten aller Kanten von A nach B:

$$c_{A,B} := \sum_{(v,w) \in E \cap (A \times B)} c(v, w) \tag{4.2}$$

Betrachten wir ein Beispiel für einen *Schnitt* auf einem *Flussnetzwerk*. Alle Knoten in A wurden grün und alle in B rot markiert. Kanten zwischen den beiden Mengen wurden gepunktet dargestellt.



Es gilt $A = \{s, v_1, v_3, v_4\}$ und $B = \{v_2, t\}$. Zur Kapazität des *Schnitts* tragen alle Kanten bei, die von Knoten aus A zu solchen aus B verlaufen. Die Kante (v_3, v_2) also schon, die Kante (v_2, v_4) aber nicht. Insgesamt beträgt die Kapazität des *Schnitts* $c(s, v_2) + c(v_1, v_2) + c(v_3, v_2) + c(v_3, t) + c(v_4, t) = 8 + 10 + 4 + 15 + 4 = 41$.

4.1.5 Definition (Minimaler Schnitt)
Als *minimalen Schnitt* bezeichnet man den oder einen Schnitt mit minimaler Kapazität. Es kann mehrere solcher *minimaler Schnitte* geben.

Dem geneigten Leser mag die Asymmetrie zwischen *Flüssen*, die maximal sein können und *Schnitten*, die minimal sein können, aufgefallen sein. Den Zusammenhang der beiden Konzepte möchte ich im Folgenden kurz beschreiben. Dazu werden wir zwei formale Aussagen betrachten, die hier aber nicht bewiesen werden. Diese sind zwar für unseren Algorithmus von Bedeutung, aber für einen formalen Beweis bräuchten wir einen größeren theoretischen Hintergrund, der obgleich furchtbar spannend, hier den Rahmen sprengte und wenig zielführend wäre.

4.1.6 Lemma
Der *Wert* eines beliebigen *Flusses* über einem *Flussnetzwerk* ist beschränkt durch einen beliebigen Schnitt.

Beweis. Siehe Cormen *et al.* (2009). □

4.1.7 Satz (Max-Flow-Min-Cut-Satz)
Für ein *Flussnetzwerk* $G = (V, E)$ mit *Senke* und *Quelle* $s, t \in V$ entspricht die minimale Kapazität aller *Schnitte* auf G dem maximalen *Wert* aller Flüsse von s nach t.

Beweis. Wenn wir an den Algorithmus von Ford-Fulkerson denken, dann erinnern wir uns, dass dieser solange *augmentierende* Pfade mit positiver *Restwertkapazität* von der *Quelle* zur *Senke* gesucht hat, bis es keinen solchen mehr gab. Sobald das der Fall war, hatte man den *maximalen Fluss* berechnet. Das bedeutet gleichzeitig, dass man den *Bottleneck* zwischen s und t gefunden hat. Genau dieser trennt die Knoten von G in zwei Hälften und ist unser *minimaler Schnitt*. Für einen formalen Beweis neben dieser Skizze siehe auch hier Cormen *et al.* (2009). □

Die Erkenntnis, die man aus diesem Satz zieht, ist, dass man Algorithmen zur Berechnung des *maximalen Flusses* benutzen kann, um den *minimalen Schnitt* eines *Flussnetzwerkes* zu bestimmen. Das werden wir im nächsten Abschnitt ausnutzen, um *Inkonsistenzen* zwischen unseren Sequenzen aufzulösen.

4.2 Inzidenzgraphen und das Auflösen von Inkonsistenzen mit Hilfe von Flussnetzwerken

4.2.1 Konstruieren des Inzidenzgraphen

Wir starten wie bei DIALIGN zunächst mit paarweisen *Alignments*. Unser Ziel ist es Übereinstimmungen zu finden, die in möglichst vielen Sequenzen gleichzeitig vorkommen. Das ist ein vielversprechender Ansatz, weil die gesuchten Motive oft in vielen sich überlappenden *Fragmente* vorkommen, während Überlappungen bei zufälligen Übereinstimmungen unwahrscheinlich sind. Dazu konstruieren wir einen sogenannten *Inzidenzgraphen*, der alle *Stellen* als Knoten enthält, die durch Kanten verbunden sind, falls es ein sie verbindendes *Fragment* gibt. Mit Hilfe eines Algorithmus zur Berechnung des *maximalen Flusses* bestimmen wir *minimale Schnitte*, bis nur noch dichte Zusammenhangskomponenten übrig bleiben, die jeweils höchstens einen Knoten aus jeder Sequenz enthalten (Corel *et al.*, 2010).

4.2.1 Definition (Mehrdeutigkeit und partielle Zuweisungsspalten)
Es sei eine Menge an Sequenzen S mit *Stellenraum* $\mathcal{S}$ gegeben. Eine Teilmenge $C \subset \mathcal{S}$ nennen wir *mehrdeutig*, wenn es mindestens eine Sequenz S_i gibt, sodass $C \cap S_i$ zwei oder mehr *Stellen* $(i, p), (i, p') \in \mathcal{S}$ enthält. In diesem Fall nennen wir auch (i, p) und (i, p') *mehrdeutig*. Analog nennen wir eine Äquivalenzrelation $\mathcal{R}$ *mehrdeutig*, wenn $\mathcal{R}$ eine *mehrdeutige* Äquivalenzklasse enthält.

Eine *nicht-mehrdeutige* Teilmenge $C \subset \mathcal{S}$ bezeichnen wir als *partielle Zuweisungsspalte*.

Es lässt sich folgern, dass eine *konsistente* Äquivalenzrelation auch nicht *mehrdeutig* ist, während das Gegenteil im Allgemeinen nicht gilt. Das liegt daran, dass es überkreuzte Zuweisungen geben kann, die aber *partielle Zuweisungsspalten* sind. *Nicht-mehrdeutige* Äquivalenzrelationen bestehen nur aus *partiellen Zuweisungsspalten*.

Im Folgenden sei eine Menge von *Fragmenten* $\mathcal{F} = \{f_1, \ldots, f_k\}$ gegeben. Normalerweise werden dies die *Fragmente* unserer paarweisen *Alignments* sein, aber theoretisch kann man auch anders bestimmte benutzen. Wir wollen möglichst wenige Verbindungen aus der durch die *Fragmente* induzierte Relation $\mathcal{R}$ löschen, bis eine *nicht-mehrdeutige* Äquivalenzrelation $\mathcal{R}'$ bleibt.

4.2.2 Definition (Inzidenzgraph)
Es sei ein *Stellenraum* $\mathcal{S}$ mit einer Menge von *Fragmenten* $\mathcal{F}$ auf diesem gegeben. Dann bezeichnen wir den ungerichteten Graphen $G_{\mathcal{F}} = (\mathcal{S}, E_{\mathcal{F}})$ als *Inzidenzgraphen*. In diesem Graph existiert genau dann eine Kante $(u, v) \in E_{\mathcal{F}}$, wenn die *Stellen* u und v in einem gemeinsamen *Fragment* $f_i \in \mathcal{F}$ vorkommen.

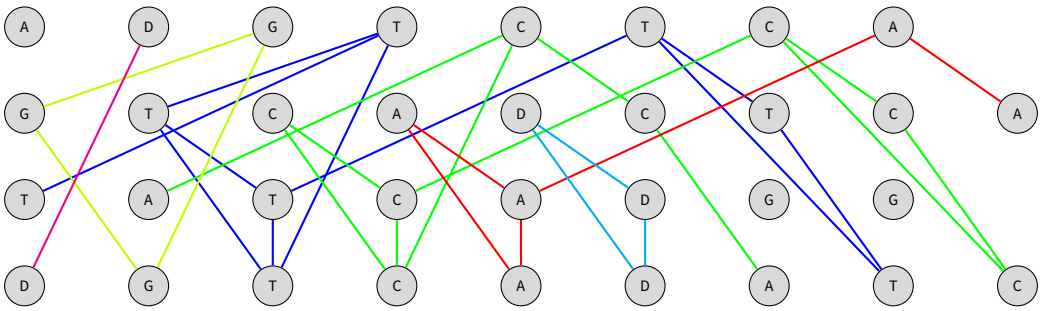
Wie man sieht, sind die Zusammenhangskomponenten unseres *Inzidenzgraphen* genau die Äquivalenzklassen der durch $\mathcal{F}$ induzierten Kanten. Weil diese nicht weiter nützlich sind, kann man *Stellen*, die nicht mit anderen verbunden sind, von vornherein ignorieren beziehungsweise sie löschen, wenn sie nach dem Entfernen von Kanten Grad 0 haben. So lässt sich etwas Speicherplatz sparen und die Ergebnisse dieses Algorithmus können direkt für den nächsten Schritt weiterverwendet werden. Um die Positionierung der Zusammenhangskomponenten im *Inzidenzgraph* zu verdeutlichen, lassen wir sie in unseren graphischen Beispiele aber stehen.

4.2.2 Beispiel Inzidenzgraph

Auch hier werden wir wieder unser Beispiel aus dem letzten Kapitel benutzen. Erinnern wir uns zunächst an die *Fragmente* der paarweisen *Alignments*. Die *Überlappgewichte* brauchen wir für diesen Schritt des Algorithmus nicht, weil wir versuchen mit Hilfe von *minimalen Schnitten* ein ähnliches aber besseres Ergebnis zu erreichen.

Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*	Seq.	*Frag.*	*Ü-Gew.*
2	GTCADCTC	16	1	TCTCA	7	1	GT	2
4	GTCADATC		3	TATCA		2	GT	
3	TCAD	8	1	CTCA	8	1	TC	2
4	TCAD		2	CTCA		4	TC	
2	TCAD	8	1	DGTC	8			
3	TCAD		4	DGTC				

Diese *Fragmente* überführen wir direkt in einen *Inzidenzgraphen* bei denen die *Stellen* als Knoten und die Verbindungen in gemeinsamen *Fragmenten* als Kanten übertragen werden. Um die Übersichtlichkeit halbwegs zu wahren, wurden die Kanten jeder Zusammenhangskomponente jeweils in einer Farbe markiert.



4.2.3 Mehrdeutigkeiten Auflösen

Die Zusammenhangskomponenten unseres *Inzidenzgraphen* werden wir jetzt als *Flussnetzwerke* interpretieren, um mit Hilfe eines Algorithmus zur Berechnung eines *minimalen Schnitts* solange Kanten zu entfernen, bis alle *Mehrdeutigkeiten* aufgelöst sind.

Sei C eine Zusammenhangskomponente von $G_{\mathcal{F}}$, die *mehrdeutig* ist, also zwei Knoten x, y aus der selben Sequenz enthält. Wir wählen die *mehrdeutigen* Knoten x und y als *Quelle* und *Senke* unseres *Flussnetzwerks*. Die ungerichteten Kanten des *Inzidenzgraphen* werden durch zwei antiparallele gerichtete Kanten ersetzt. Corel *et al.* (2010) benutzen als Kapazitäten jeweils 1. Meiner Meinung nach wäre es hingegen sinnvoller die Kapazitäten so zu wählen, dass ähnliche Symbole seltener voneinander durch den *minimalen Schnitt* und das damit einhergehende Löschen von Kanten getrennt werden. Das können wir beispielsweise erreichen, indem wir uns an den Ähnlichkeitswerten unserer Substitutionsmatrix orientieren. Weil *Flussnetzwerke* nicht-negative Kapazitäten erwarten, müssen die Werte gegebenenfalls modifiziert werden, indem wir den betragsmäßig größten Eintrag der Matrix zu allen Einträgen addieren. Das bedeutet, dass bei unserer $(+3/-1)$-Substitutionsmatrix aus dem Beispiel zwischen übereinstimmenden Symbolen eine Kapazität von 6 und zwischen nicht übereinstimmendes eine von 2 benutzt wird. Leider war es aufgrund der begrenzten Zeit und des hohen Aufwands nicht möglich das ganze Verfahren zu implementieren. Eine Evaluierung mit echten *Alignments* fehlt also leider. Weil ich den Ansatz mit den nicht-unitären Kapazitäten als sehr sinnvoll erachte, habe ich ihn bei unseren Beispielen verwendet.

Als nächstes benutzen wir einen Algorithmus zur Bestimmung des *maximalen Flusses*. Wie mit dem *Max-Flow-Min-Cut-Satz* gezeigt, bestimmen wir durch die Bestimmung des *maximalen Flusses* auch gleichzeitig einen *minimalen Schnitt*. Das ist die „schmalste“ Verbindungsstelle zwischen der *Quelle* und der *Senke* und wir hoffen durch Löschen der dazugehörigen Kanten die *Mehrdeutigkeit* aufzulösen und die dichtbesetzten Untergraphen zu erhalten. Im Original wird der Algorithmus von Edmonds-Karp benutzt, während wir stattdessen einen *Push-Relabel-Algorithmus* mit Laufzeit $O(|V|^2 \cdot \sqrt{|E|})$ für die Komplexität betrachten und in unserer Implementierung verwenden. Nachdem der *minimale Schnitt* bestimmt wurde, löschen wir alle Kanten zwischen den Mengen A und B. Auf diese Art und Weise wird unsere Zusammenhangskomponente C in zwei neue Zusammenhangskomponenten A und B aufgeteilt. Dieses Verfahren wiederholen wir solange, bis es

keine *mehrdeutigen* Äquivalenzklassen mehr gibt.

Graphik aus CPM10 mit Zusammenhangskomponenten einfügen.

Algorithmus 4 Algorithmus zum Auflösen von Mehrdeutigkeiten

Require: *Inzidenzgraph* $G_{\mathcal{F}} = (\mathcal{S}, E_{\mathcal{F}})$ über einem Satz *Fragmente* $\mathcal{F}$
1: **procedure** ResolveAmbiguities($G_{\mathcal{F}}$)
2: $E \leftarrow E_{\mathcal{F}}$
3: Berechne Zusammenhangskomponenten von $G_{\mathcal{F}}$
4: **while** es ex. *mehrdeutige* Zusammenhangskomponente C von $G_{\mathcal{F}}$ **do**
5: **while** es ex. *mehrdeutige* Knoten x, y aus der selben Sequenz **do**
6: Wähle Sequenz S_i mit max. Anzahl an *mehrdeutigen* Knoten in C
7: Wähle $s = \text{argmin}\{\deg(v) | v \in C \text{ und } v \in S_i\}$
8: Wähle $t = \text{argmax}\{\deg(v) | v \in C \text{ und } v \in S_i\}$
9: Definiere *Flussnetzwerk* auf C mit *Quelle* s und *Senke* t
10: Benutze $\mathrm{PushRelabel}$, um *minimalen Schnitt* C_1 und C_2 zu bestimmen
11: Lösche Kanten zwischen C_1 und C_2 aus E
12: **end while**
13: **end while**
14: **return** *nicht-mehrdeutigen* Subgraphen $(\mathcal{S}, E)$ von $G_{\mathcal{F}}$
15: **end procedure**

Unglücklicherweise können die Zusammenhangskomponenten bei *Alignments* zwischen vielen langen Sequenzen sehr groß werden. Corel *et al.* (2010) haben daher eine Grenze $k = \max\{\deg(v) | v \in \mathcal{S}\}$ eingeführt, die sukzessive gesenkt wird, bis alle *Mehrdeutigkeiten* aufgelöst wurden. k wird so benutzt, dass alle Kanten zwischen Knoten mit Grad $< k$ zunächst nicht betrachtet werden, sodass andere kleinere Zusammenhangskomponenten vorliegen. Als erstes werden für den reduzierten Kantensatz $E_k = \{(u, v) \in E | \min\{\deg(u), \deg(v)\}\}$ solange *minimale Schnitte* berechnet und Kanten gelöscht, bis für den Graph mit weniger Kanten keine *mehrdeutigen* Zusammenhangskomponenten mehr existierten. Danach wird k um eins reduziert und das Vorgehen auf dem neuen Kantensatz wiederholt. Sobald k auf null gesetzt wurde, hat man alle Knoten betrachtet und es liegen nur noch *partielle Zuweisungsspalten* vor.

keine eigene Implementierung, also Text über k anpassen.

Abgesehen von der verbesserten Laufzeit scheint dieses Vorgehen aber keine Vorteile zu bieten und falls doch, werden diese in Corel *et al.* (2010) nicht genannt. Deshalb verzichte ich auf die Grenze k und gehe davon aus, dass aufgrund des deutlich effizienteren Algorithmus für den *maximalen Flusses* und die Verwendung einer schnellen Graphimplementierung aus einer Bibliothek keine Laufzeitprobleme für die üblichen Anwendungsgrößen auftreten.

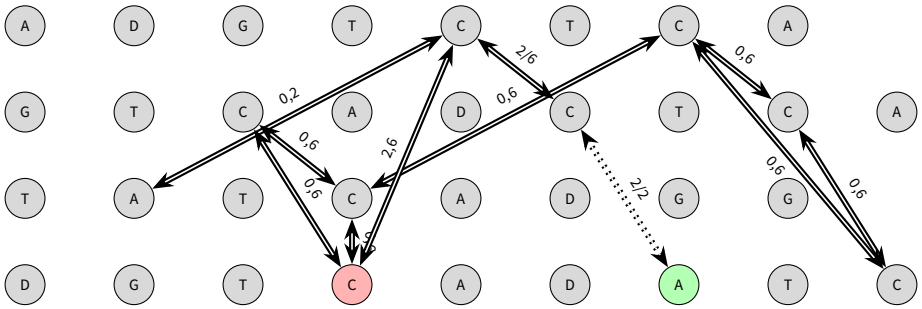
Bis jetzt wurde nur gesagt, dass wir *mehrdeutige* Zusammenhangskomponenten auswählen und für diese jeweils einen Knoten aus der selben Sequenz als *Quelle* und *Senke* wählen. Abschließend müssen wir also noch festlegen in welcher Reihenfolge diese gewählt werden. Für die Zusammenhangskomponenten müssen wir keine Reihenfolge festlegen, weil diese unabhängig voneinander sind. Sei eine *mehrdeutige* Zusammenhangskomponente C gegeben. Dann mag es mehrere Sequenzen geben, die alle zu mehr als einem Knoten in C korrespondieren. Außerdem muss es nicht unbedingt einen eindeutigen *minimalen Schnitt* geben, sondern es kann mehrere solche geben. Wir entscheiden uns für dieses Vorgehen:

1. Falls es mehrere Sequenzen gibt, die zwei oder mehr *mehrdeutige Stellen* in C ent-

halten, wähle die Sequenz S_i mit den meisten *mehrdeutigen* Knoten in dieser Zusammenhangskomponente.
2. Sobald S_i bestimmt ist, wähle unter allen *Stellen* aus dieser Sequenz in C den Knoten mit dem niedrigsten Knotengrad als Quelle und den mit dem höchsten als *Senke*.
3. Sollte es mehr als einen *minimalen Schnitt* geben, dann orientieren wir uns an dem Ergebnis des Algorithmus der Implementierung und wählen die Partitionierung, die sich durch die nur durch Kanten mit *Restwertkapazität* 0 verbundenen Subgraphen im *Restwertgraphen* ergibt.

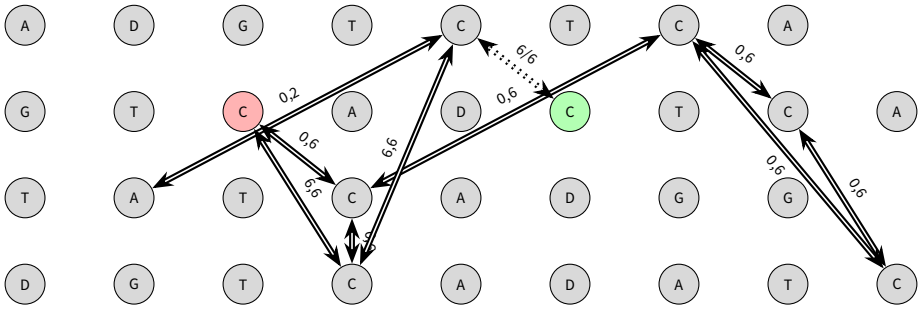
4.2.4 Beispiel von $\mathrm{ResolveAmbiguities}$

Exemplarisch betrachten wir die grüne Zusammenhangskomponente, weil diese die meisten Knoten enthält und werden an ihrem Beispiel solange den *minimalen Schnitt* durchführen, bis keine *Inkonsistenzen* mehr vorliegen. Die *Quelle* ist grün, während die *Senke* rot dargestellt ist.

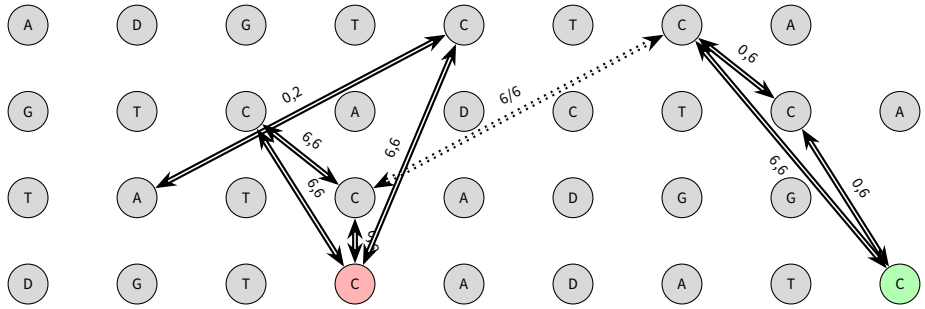


Sowohl S_2, als auch S_4 haben drei Knoten in C. Wir wählen hier $S_4[7]$ als *Quelle* und $S_4[4]$ als *Senke*, weil diese den kleinsten bzw. größten Knotengrad haben. In diesem Fall passiert nichts spannendes, weil der *minimale Schnitt* aufgrund der Kante mit Kapazität 2 direkt die *Quelle* vom Rest der Zusammenhangskomponente trennt.

Im nächsten Schritt hat S_2 mit 3 die meisten Knoten in C. Da alle zwei der Knoten Grad 2 haben, entscheiden wir uns für $S_2[6]$ als *Quelle* und $S_2[8]$ als *Senke*. Auch hier wird direkt die erste Kante an der *Quelle* gelöscht.

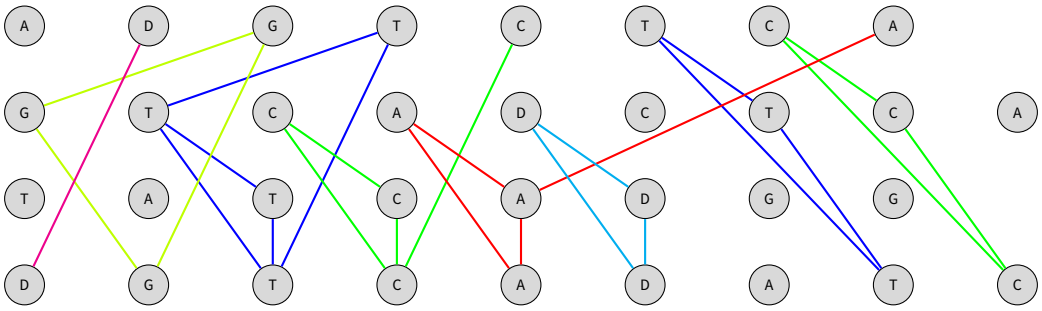


Gehen nun wieder zu Sequenz S_4 und wählen $S_4[9]$ als *Quelle* und $S_4[4]$ als *Senke*. Hier haben wir zwei relativ dichte Subgraphen, die nur durch die eine Kante $(S_1[7] \rightarrow S_3[4])$ miteinander verbunden sind. Diese löschen wir und danach gibt es in der rechten Zusammenhangskomponente keine *Mehrdeutigkeiten* mehr.



Abschließend wird noch die Verbindung zwischen dem A aus der zweiten Sequenz und dem C an der fünften Stelle der ersten gelöscht, was hier nicht mehr graphisch dargestellt wurde. Danach sind alle *Mehrdeutigkeiten* aufgelöst und es liegen nur noch *partielle Zuweisungsspalten* vor.

Wenn wir den Algorithmus auch auf allen anderen Zusammenhangskomponenten anwenden, dann kommen wir zu folgendem *Inzidenzgraphen* ohne *Mehrdeutigkeiten*:



Obwohl in diesem Graphen keine *Mehrdeutigkeiten* mehr existieren, ist die Zuordnung noch nicht *konsistent*. Das liegt an Überkreuzungen, wie der zwischen $(S_1[8] \rightarrow S_3[5] \rightarrow S_4[5])$ (rot) und $(S_1[7] \rightarrow S_4[9])$ (grün).

4.2.5 Komplexität

Der *Inzidenzgraph* lässt sich aus den *Fragmenten* der paarweisen *Alignments* in $O(n^2 \cdot L)$ berechnen, weil es $O(n^2)$ davon gibt, die bis zu $O(L)$ Verbindungen enthalten.

Die Laufzeit von $\mathrm{ResolveAmbiguities}$ wird von der Laufzeit zur Berechnung des *minimalen Schnitts* durch $\mathrm{PushRelabel}$ dominiert (Corel *et al.*, 2010). Diese hängt von der Größe unserer Zusammenhangskomponenten ab. Im schlimmsten Fall besteht der *Inzidenzgraph* aus einer einzigen Zusammenhangskomponente bei der jeder Knoten mit mindestens einem Knoten aus jeder anderen Sequenz verbunden ist. In diesem Fall muss sie in vielen Schritten zerkleinert werden muss, bis nur noch *partielle Zuweisungsspalten* übrig bleiben. Eine Zusammenhangskomponente kann $n \cdot L$ Knoten und $O(n^2 \cdot L)$ Kanten haben. Das liegt daran, dass wir als Grundlage unsere paarweisen *Alignments* benutzen, bei denen jede *Stelle* für jede andere Sequenz mit höchstens einer *Stelle* aus dieser verbunden ist.

$\mathrm{PushRelabel}$ hat eine Komplexität von $O(|V|^2 \cdot \sqrt{|E|})$, um einen *minimalen Schnitt* zu berechnen. Weil $|V| \in O(n \cdot L)$ und $|E| \in O(n^2 \cdot L)$ gelten, braucht man für einen Durchlauf also $O(n^3 \cdot L^{5/2})$. Wenn wir Pech haben, entfernt jeder Aufruf unseres Algorithmus für den *maximalen Fluss* nur einen einzigen Knoten aus der Zusammenhangskomponente.

Das resultiert in $n \cdot L$ Aufrufen und einer Gesamtlaufzeit von $O(n^4 \cdot L^{7/2})$. Die Kosten für das Bestimmen der Zusammenhangskomponenten, das Finden der *Quellen* und *Senken* und das Löschen von Kanten liegen jeweils in linearer Größe zur Größe des Graphen und werden daher hier nicht weiter betrachtet. Für ersteres benutzt man beispielsweise eine Tiefensuche, fürs zweite eine Breitensuche und letzteres ist simples Iterieren über alle Kanten.

Möglicherweise lässt sich eine bessere Laufzeit erreichen, wenn alle Kanten eine uniforme Kapazität haben, wie das bei der Variante der Autoren der Fall ist. Es lässt sich zeigen, dass das ein deutlich leichteres Problem ist (Goldberg und Tarjan, 2014) und nach Karzanov und Even hat eine Variante des Algorithmus von Dinic in diesem Fall gute Laufzeiten.

Glücklicherweise sind die Zusammenhangskomponenten im Allgemeinen nicht so groß und in den meisten Fällen trennt man mit dem *minimalen Schnitt* auch nicht nur einzelne Knoten ab. Bei Testläufen auf der Protein-Referenzdatenbank BAliBASE haben Corel *et al.* (2010) die Referenzmenge RV12 genauer betrachtet. RV12 besteht aus 88 Sequenzfamilien, die im Schnitt zehn Sequenzen enthielt. Messungen haben ergeben, dass die *Inzidenzgraphen* auf diesen Sequenzfamilien im Schnitt 2877 Knoten und 10952 in 223 Zusammenhangskomponenten enthalten haben. Auf diesen Sequenzen lassen sich in guten Laufzeiten von weniger als einer Minute multiple *Sequenzalignments* berechnen. Anders sah es auf der Sequenzfamilie BB30003 aus. Diese besteht aus 142 Sequenzen und resultiert in einem monströsen *Inzidenzgraphen*, der nur aus einer einzigen Zusammenhangskomponente besteht. Nach einer Laufzeit von 20 Stunden ohne Ergebnis wurde der Lauf auf Graphen, der $1,5 \cdot 10^6$ Kanten enthält, erfolglos abgebrochen. Erst mit einer Begrenzung des minimalen *Fragmentgewichts* auf 4 ließ sich in 13 Stunden ein Ergebnis erzielen.

4.3 Sukzessionsgraphen und der Algorithmus von Pitschi

4.3.1 Aufbau des Sukzessionsgraphen

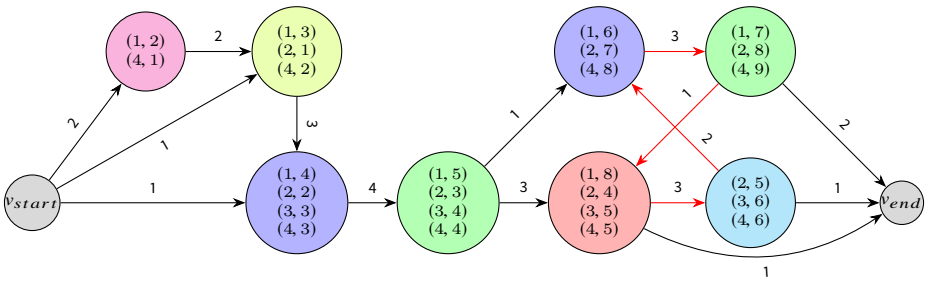
Wie wir gesehen haben, ist die Menge an Zuweisungen nach dem $\mathrm{ResolveAmbiguities}$-Aufruf noch nicht *konsistent*, aber es liegen keine *Mehrdeutigkeiten* mehr vor. Das zwingt uns dazu weitere Verbindungen aus der Äquivalenzrelation zu löschen, bis diese ein *Alignment*, also *konsistent* ist. Wir führen dazu eine Datenstruktur ein, die die Zusammenhangskomponenten des reduzierten *Inzidenzgraphen* nach ihrer Ordnung in den beteiligten Sequenzen ordnet.

4.3.1 Definition (Sukzessionsgraph)
Es sei eine Menge an Sequenzen S mit *Stellenraum* $\mathcal{S}$ gegeben. Für diese Sequenzen liegt eine Menge C von Zusammenhangskomponenten vor, die alle *partielle Zuweisungsspalten* sind. Dann definieren wir den *Sukzessionsgraph* $SG(C) = (C, E)$ als gerichteten, gewichteten Graphen. In $SG(C)$ fügen wir genau dann eine Kante $e = (C, C')$ für $C, C' \in C$ ein, wenn eine Sequenz $S_i \in S$ existiert, für die *Stellen* $s = (i, p) \in C$ und $s' = (i, p') \in C'$ vorliegen mit $p < p'$ und außerdem keine *Stelle* $s = (i, p'')$ in einem anderen Knoten C'' mit $p < p'' < p'$ vorhanden ist. Das Gewicht von e ist die Anzahl an Sequenzen für die die obige Bedingung gilt. Des Wei-

teren setzen wir voraus, dass alle Zusammenhangskomponenten mindestens zwei Knoten enthalten. Außerdem fügen wir zwei zusätzliche Knoten v_{start} und v_{end} ein. v_{start} ist mit allen Knoten verbunden, die die erste *Stelle* einer Sequenz enthalten. Zusätzlich sind alle Knoten, die die letzte *Stelle* einer Sequenz enthalten mit v_{end} über eine Kante verbunden.

Wie man sieht sind zwei Knoten C und C' aus dem *Sukzessionsgraphen* genau dann miteinander verbunden, wenn es in diesen *Stellen* aus der selben Sequenz gibt, bei denen die aus C links von der in C' stehen. Die zusätzlich eingefügten Knoten v_{start} und v_{end} brauchen wir im Algorithmus von Pitschi, um die Konnektivität des Graphen zu erhalten, wenn wir Kanten aus dem Graph löschen. Das werden wir später tun, um *Inkonsistenzen* zu entfernen. Setzen wir nicht voraus, dass die Knoten von SG mindestens zwei *Stellen* enthalten, dann liefert der Algorithmus von Pitschi suboptimale Ergebnisse, wie wir später sehen werden. Corel *et al.* (2010) und Pitschi *et al.* (2010) sind an dieser Stelle nicht eindeutig.

Als Beispiel konstruieren wir aus dem reduzierten *Inzidenzgraphen* des letzten Schritts einen *Sukzessionsgraphen*. Die Knotenfarben wurden in der Farbe der Zusammenhangskomponenten des *Inzidenzgraphen* aus dem letzten Schritt gewählt.



4.3.2 Lemma
Die Menge C ist genau dann *konsistent*, wenn $SG(C)$ ein gerichteter, azyklischer Graph (DAG) ist.

Beweis. „⇒“: Sei C eine *konsistente* Menge von Zusammenhangskomponenten mit der induzierten Äquivalenzrelation $\mathcal{R}$. Dann folgt aus $x \preceq_{\mathcal{R}} y$ auch $x \preceq y$ für alle Sequenzen. Angenommen es gibt in $SG(C)$ einen Zyklus. Dann gibt es Knoten C, C' mit *Stellen* $s_1, s_2 \in C$ und $s_1', s_2' \in C'$ mit folgenden Eigenschaften:

1. $s_1, s_1' \in S_1$ und $s_2, s_2' \in S_2$
2. $pos(s_1) < pos(s_1')$ und $pos(s_2') < pos(s_2)$

Diese muss es geben, sonst gäbe es keinen Zyklus im Graphen. Für diese *Stellen* gelten $s_1' \mathcal{R} s_2'$, $s_2' \preceq s_2$ und $s_2 \mathcal{R} s_1$, woraus $s_1' \preceq_{\mathcal{R}} s_1$ folgt. Es gilt aufgrund von $pos(s_1) < pos(s_1')$, weshalb $s_1 \not\preceq s_1'$ folgt. Das steht im Widerspruch dazu, dass C *konsistent* war.

„⇐“: Wir benutzen einen Kontrapositionsbeweis und es sei eine *inkonsistente* Menge C gegeben. Dann existiert eine Sequenz S_1, die *Stellen* $s_1, s_1' \in S_1$ mit folgenden Eigenschaften enthält:

1. $s_1 \preceq_{\mathcal{R}} s_1'$
2. $s_1 \not\preceq s_1'$

Aus $s_1 \preceq_{\mathcal{R}} s_1'$ folgt, dass es zwei Knoten C und C' gibt mit $s_1 \in C$ und $s_1' \in C'$, die in $SG(C)$ über einen Pfad verbunden sind. Der Pfad kann aber nicht über eine Kante laufen,

die zu S_1 gehört, denn $s_1 \not\preceq s_1'$ gilt. Aus dieser Bedingung folgt aber, dass einen Pfad von C' zu C' geben muss. Da es sowohl von C nach C', als auch von C' nach C Pfade gibt, kann $SG(C)$ nicht azyklisch sein. □

4.3.2 Der Algorithmus von Pitschi

Aufgrund des gerade gezeigten Lemmas folgt, dass wir eine *konsistente* Menge von *partiellen Zuweisungsspalten* finden, wenn deren *Sukzessionsgraph* azyklisch ist. Der Algorithmus von Pitschi *et al.* (2010) sieht zwei Schritte vor, um aus dem potentiell zyklischen *Sukzessionsgraphen* einen kreisfreien zu konstruieren, der mit einer *konsistenten* Menge von *partiellen Zuweisungsspalten* korrespondiert:

1. Lösche Kanten aus dem *Sukzessionsgraphen*, bis dieser kreisfrei ist.
2. Benutze den so entstandenen DAG, um zu entscheiden, welche *Stellen* aus den jeweiligen Knoten gelöscht werden müssen, um *Inkonsistenzen* zu entfernen.

Entfernen der Kanten

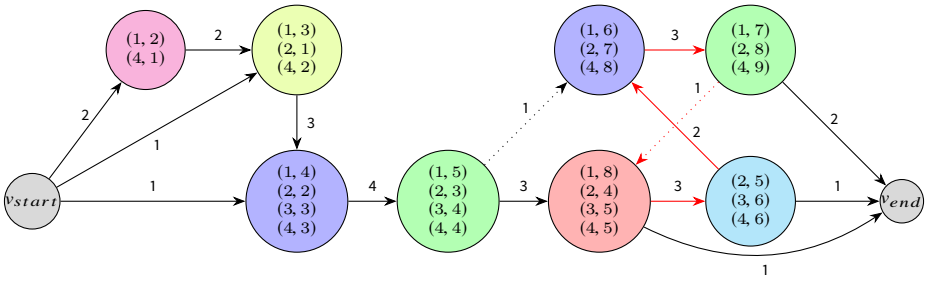
Wir beginnen mit der Transformation des zyklischen Graphen in einen azyklischen, indem wir Kanten entfernen. Optimal wäre es Kanten mit einer minimalen Summe von Kantengewichten zu entfernen. Leider ist dieses Problem, das auch als „minimal weighted feedback arc set"-Problem bekannt ist, NP-schwer. Als Folge bedienen wir uns einfach einer einfachen Heuristik, indem wir sukzessive eine Grenze k erhöhen und in jedem Schritt alle Kanten mit einem Gewicht kleiner als k löschen, bis es keinen Zyklus im Graphen mehr gibt. Formal definieren wir die Menge an Kanten, die mindestens das Gewicht $k \in \mathbb{N}$ hat als

$$\begin{aligned} &E_k := \{(u,v) \in E | w(u,v) > k \text{ oder } u = v_{start} \text{ oder } v = v_{end}\} \text{ mit} \\ &k^* := \min\{k > 0 | (V, E_k) \text{ ist azyklisch}\} \end{aligned} \tag{4.3}$$

Für den Fall, dass durch das Löschen von Kanten der Graph nicht mehr zusammenhängend ist, fügen wir für jede Zusammenhangskomponente und jede an dieser beteiligten Sequenz eine Kante vom Startzustand v_{start} zum Knoten mit der kleinsten *Stelle* aus der gewählten Sequenz ein. Analog gehen wir mit den größten *Stellen* und dem Endzustand v_{end} vor. Diese Menge dieser Kanten nennen wir E_c. Auf diese Weise ist sichergestellt, dass der azyklische Graph $G^* = (V, E_{k^*} \cup E_c)$ zusammenhängend ist und über jeden Knoten ein Pfad vom Start- zum Endzustand führt.

Beispiel zum Entfernen von Kanten

Bei unserem Beispiel ist glücklicherweise nur Zyklus vorhanden, der bereits in (V, E_1) nicht mehr vorhanden ist. Wir löschen also alle Kanten mit Kantengewicht 1 oder weniger, die nicht zum Start- oder Endknoten gehören. Entfernte Kanten wurden gestrichelt dargestellt. Das sieht dann so aus:



Entfernen von Stellen

Als nächstes lernen wir einen von Pitschi *et al.* (2010) entwickelten Algorithmus kennen, der eine minimale Anzahl an *Stellen* aus dem verkleinerten *Sukzessionsgraphen* G^* löscht, um alle *Inkonsistenzen* zu entfernen. Dabei orientiert sich der Algorithmus an der linearen Halbordnung $\preceq$ über $\mathcal{S}$ und der Halbordnung auf dem DAG, die ersterer angepasst werden soll. Wir bezeichnen die Halbordnung auf C, die durch den Graph G^* induziert wird, als $\preceq^*$.

Sei eine Sequenz $S_i \in S$ gegeben. Dann ist C_{S_i} die Teilmenge von Zusammenhangskomponenten aus C, die *Stellen* aus S_i enthalten. Wir definieren außerdem eine Beschränkung der Knoten aus G^* von C_{S_i} als $V_{S_i} = C_{S_i} \cup \{v_{start}, v_{end}\}$. Auf diesen existiert eine Ordnung $\preceq_{S_i}$, die durch die natürliche Ordnung auf S_i gegeben ist, und die Halbordnung $\preceq^*_{S_i}$ des Graphen mit reduzierter Knotenmenge. Wir definieren $\mathcal{R}_{S_i} = \preceq_{S_i} \cap \preceq^*_{S_i}$. Diese Relation entspricht genau den Verbindungen der transitiven Hülle im ursprünglichen Graph, wenn man G^* auf die Knoten aus C_{S_i} beschränkt. Sei dafür $G^+ = (V, E^+)$ die transitive Hülle $TC(G^*)$ des DAG. Weil G^* keine Zyklen enthält, kann auch $TC(G^*)$ keine enthalten. Wir können G^* auf jede unserer Sequenzen S_i beschränken, indem wir den reduzierten Knotensatz V_{S_i} benutzen und genau dann eine Kante zwischen zwei Knoten einfügen, falls diese in unserer Relation $\mathcal{R}_{S_i}$ liegen.

4.3.3 Definition
Der Graph G_{S_i} einer Sequenz S_i ist definiert als Graph über der Knotenmenge V_{S_i} und den Kanten E_{S_i}. Es gilt

$$(u, v) \in E_{S_i} \iff u, v \in V_{S_i}, (u, v) \in E^+ \text{ und } u \preceq_{S_i} v \iff u \mathcal{R}_{S_i} v \tag{4.4}$$

Pfade von v_{start} nach v_{end} in G_{S_i} sind genau die Teilmengen von *partiellen Zuweisungsspalten*, die bezüglich S_i *konsistent* sind. Das liegt daran, dass für zwei Knoten $u, v \in V_{S_i}$ auf einem solchen Pfad sowohl $u \preceq_{S_i} v$, als auch $u \preceq^*_{S_i} v$ gelten und damit $u \preceq^*_{S_i} v \implies u \preceq_{S_i} v$. Das war aber genau die Definition für *Konsistenz*: dass die Relation die natürliche Ordnung auf den Sequenzen erhält. Wie entfernen daher alle *Stellen* unserer Sequenz aus den Knoten, die nicht auf dem gewählten Pfad liegen, weil diese die *Konsistenz* verletzen würden.

Reicht das als Beweis oder muss das mit dem DAG aus dem Lemma gezeigt werden?

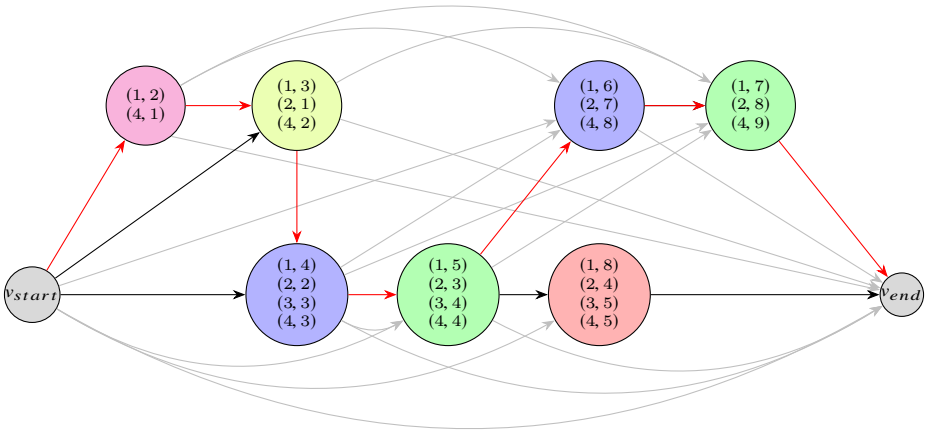
Wenn wir jetzt für jede unserer Sequenzen S_j einen Pfad von v_{start} nach v_{end} in G_{S_j} wählen und alle nicht-besuchten *Stellen* aus ihren Knoten entfernen, dann hält die *Konsistenzbedingung* auf der Relation, die durch die übriggebliebenen Zusammenhangskomponenten induziert wird. Das Resultat ist also ein *Alignment*. Da wir aber nicht irgendein *Alignment* erhalten wollen, sondern ein möglichst großes, wählen wir für jede Sequenz $S_i \in S$ den Pfad maximaler Länge durch G_{S_i}. Auf diese Weise löschen wir die minimale

Anzahl an *Stellen* aus ihren Zusammenhangskomponenten. Formal: Sei für eine Sequenz S_i g_{S_i} ein Pfad maximaler Länge $(v_{start}, u_1, \ldots, u_n, v_{end})$ gegeben. Wir iterieren über alle Knoten $C \in C_{S_i}$ und entfernen *Stellen* $(i, p) \in C$, falls $C \notin g_{S_i}$. Wenn wir das für alle Sequenzen tun, dann nennen wir die Menge der reduzierten Zusammenhangskomponenten C. Weil die mit C korrespondierende Relation ein *Alignment* ist, wäre der *Sukzessionsgraph* $SG(C)$ ein DAG. Da die Pfade für jede Sequenz unabhängig voneinander sind, können wir diese in beliebiger Reihenfolge wählen, ohne dass dies einen Einfluss auf unser Ergebnis hat.

Letztendlich folgt, dass das Problem die *partiellen Zuweisungsspalten* auf *konsistente Zuweisungsspalten* zu reduzieren auf die Suche nach Pfaden maximaler Länge durch gerichtete azyklische Graphen abbildbar ist (Corel *et al.*, 2010).

Beispiel zum Entfernen von Stellen mit dem Algorithmus von Pitschi

Bevor wir im nächsten Abschnitt genauer kennenlernen, wie man einen längsten Pfad im DAG eigentlich genau berechnet, betrachten wir unser Beispiel, in diesem Fall nur für die Sequenz S_1. Weil der Algorithmus zum Berechnen des längsten Pfades die Kantengewichte ignoriert und nur Knoten zählt, wurden diese nicht mehr angegeben. Formal handelt es sich hierbei um die Suche nach einem längsten Pfad in einem ungewichteten Graphen. Die Pfade, die über die Bildung der transitiven Hülle neu dazu kamen, sind grau angedeutet.



Im Graphen G_{S_1} gibt es eine Kante vom $(1, 5)$er-Knoten zum $(1, 6)$er, obwohl diese direkte Kante in G^* gelöscht wurde. Das liegt an der Verbindung $(2, 3) \rightarrow (2, 4) \rightarrow (2, 5) \rightarrow (2, 7)$ und $(1, 5) \preceq_{S_1} (1, 6)$. Die Kante von $(1, 8) \rightarrow (1, 6)$ gibt es aber nicht, obwohl diese in der transitiven Hülle verbunden sind. Das liegt an $(1, 8) \not\preceq_{S_i} (1, 6)$. Der hellblaue Knoten enthält keine *Stelle* aus S_1 und ist daher nicht Teil von V_{S_1}.

Wie schon beim Blick auf den ersten Sukzessionsgraphen vermutet, ist das Problem die *Stelle* $(1, 8)$, die für Überkreuzungen sorgt. Sie liegt nicht auf dem längsten Pfad (rot markiert) und wird daher aus der Zusammenhangskomponente des Knoten entfernt. Die längsten Pfade aller anderen Sequenzen besuchen auch alle Knoten der jeweiligen Graphen, weshalb keine weiteren Knoten modifiziert werden müssen.

4.3.3 Algorithmus zur Bestimmung des längsten Pfads

Es sei ein gerichteter, azyklischer Graph $G = V, E$ gegeben. Ein einfacher Ansatz wäre es einen Algorithmus, der auch mit negativen Kantengewichten rechnen kann zu benutzen, um im Graphen mit negierten Kantengewichten $G^- = (V, E^-)$ den kürzesten Pfad zu berechnen. Der kürzeste Pfad mit umgedrehten Kantengewichten entspricht genau dem längsten Pfad im Ursprungsgraph. Der Algorithmus von Bellman-Ford berechnet den kürzesten Pfad eines Ausgangsknotens (hier v_start) zu allen anderen Knoten im Graphen in $O(|V| \cdot |E|) = O(n^3 \cdot L^3)$ in unserem Fall Cormen *et al.* (2009). Anders als beispielsweise der gierige Algorithmus von Dijkstra kann Bellman-Ford auch mit negativen Kanten umgehen, solange der Graph keine negativen Zyklen enthält.

Weil unsere Graphen G_{S_i} azyklisch sind lässt sich die Berechnung der längsten Pfade mit der *topologischen Sortierung* der Knoten effizienter umsetzen.

4.3.4 Definition (Topologische Sortierung)
Eine *topologische Sortierung* von DAG $G = (V, E)$ ist eine lineare Aufzählung der Knoten in V bei der ein Knoten v hinter einem Knoten u auftaucht, wenn es eine Kante $(u, v) \in E$ gibt.

Die *topologische Sortierung* liefert eine mögliche Aufzählung der Knoten, die die Halbordnung des Graphen erhält. Im Allgemeinen ist sie nicht eindeutig. Das Standardmotivation dafür ist das Anziehen von Kleidungsstücken: Ich muss meine Socken anziehen, bevor ich in meine Schuhe schlüpfen kann und es kommt zuerst die Unterwäsche und dann die Hose. Erst muss ich Unterhemd und oder T-Shirt anziehen, bevor der Pullover kommt. Diese Reihenfolgen lassen sich als DAG modellieren, wobei jede Kante zwischen Kleidungsstück a und b bedeutet, dass a vor b angezogen werden muss. Gültige *topologische Sortierungen* für dieses vereinfachte Beispiel [2] sind also beispielsweise diese: „(Socken, Schuhe, Unterhose, Hose, Unterhemd, T-Shirt,Pullover)“, „(Socken, Unterhose, Schuhe, Unterhemd, T-Shirt, Pullover, Hose)“ oder „(Unterhose, Hose, Socken, Unterhemd, T-Shirt, Pullover, Schuhe)“.

Mit Hilfe einer Tiefensuche lässt sich die *topologische Sortierung* in $\theta(|V| + |E|)$ berechnen Cormen *et al.* (2009). Das Ergebnis ist eine Liste der Knoten in topologischer Reihenfolge. Die Liste können wir zur Berechnung des längsten Pfades nutzen, indem wir zusätzlich für jeden Knoten die Länge des längsten Pfades und den Vorgänger in diesem speichern. Wir starten bei v_{start} mit Länge 0 und Vorgänger „NIL“. Aufgrund der gegebenen *topologischen Sortierung* wissen wir, dass jeder Knoten erst dann bearbeitet wird, wenn seine Vorgänger bereits behandelt wurden. Vorgänger und längster Pfad bis zu diesem Knoten lassen sich also rekursiv berechnen:

$$\begin{aligned} distance(v_{start}) &= 0 \\ distance(v) &= \max\{distance(u) + w((u, v)) \mid (u, v) \in E \ \wedge v \neq v_{start}\} \end{aligned} \tag{4.5}$$

Der Vorgänger für die Knoten wird dann abhängig von der gewählten Kante gesetzt, die die größte Gesamtlänge liefert. Das Kantengewicht $w((u, v))$ ist in unserem Fall 1, weil wir nur die Anzahl der Knoten von v_{start} bis v_{end}. Die Korrektheit des Algorithmus folgt

[2] Mr. Bean hat mit seiner Badehose bewiesen, dass die übliche Reihenfolge des An- und Ausziehens nicht unbedingt eingehalten werden muss.

aufgrund der Korrektheit der Rekursionsgleichung und weil $distance(u)$ wegen der *topologischen Sortierung* bereits berechnet wurde für Knoten v, wenn eine Kante (u, v) existiert.

Jetzt kennen wir die Länge des längsten Pfades, müssen aber noch die Knoten, die zu diesem gehören, berechnen. Dazu nutzen wir einen Backtracking-Algorithmus, indem wir mit dem letzten Knoten starten und solange den Vorgänger auswählen, bis es keinen mehr gibt. Bei unserem Graphen G_{S_i} ist das einfach, weil mit v_{start} und v_{end} auf jeden Fall der erste und letzte Knoten unseres längsten Pfads bekannt sind. In allgemeinen DAGs muss man zuerst in linearer Zeit einmal über die Liste iterieren und den Knoten mit der maximalen gespeicherten Länge als Endknoten des Pfads und Startknoten des Backtrackings wählen.

Algorithmus 5 LongestPath

Require: DAG $G = (V, E)$
1: **procedure** LONGESTPATH(G)
2: $list_{ts} \leftarrow topologicalSort(G)$
3: $v_{start}.distance \leftarrow 0$
4: $v_{start}.pred \leftarrow$ NIL
5: $current \leftarrow list_{ts}.head$
6: **while** $current.next \neq$ NIL **do** ▷ Berechne Distanz zu v_{end}
7: $current \leftarrow current.next$
8: $maxPred \leftarrow argmax\{u.distance \mid (u, current.key) \in E\}$
9: $current.key.distance \leftarrow maxPred.distance + 1$
10: $current.key.pred \leftarrow maxPred$
11: **end while**
12: $longestPath \leftarrow empty$ ▷ initialisiere Ausgabeliste
13: $longestPath.addFirst(v_{end}$
14: $currentNode \leftarrow v_{end}$
15: **while** $currentNode.pred \neq$ NIL **do** ▷ Backtracking
16: $currentNode \leftarrow currentNode.pred$
17: $longestPath.addFirst(currentNode)$
18: **end whilereturn** $longestPath$
19: **end procedure**

Im kommenden Kapitel wird die Implementierung dieses Algorithmus beispielhaft für das ganze Verfahren beschrieben. Da dort auch ein konkretes Beispiel behandelt wird, verzichte ich darauf an dieser Stelle.

Laufzeit

Im Allgemeinen hat dieser Algorithmus eine Laufzeit von $O(|V| + |E|)$. Die *topologische Sortierung* benötigt $\theta(|V| + |E|)$ Rechenschritte. Für die Berechnung der Distanzwerte müssen in jedem der $O(|V|)$ Knoten das Maximum der Knoten über eingehende Kanten berechnet werden. Jede Kante wird nur einmalig betrachtet und somit ist die Summe der Kosten aller Maximumsoperationen in $O(|E|)$. Der Backtrackingprozess ist in linearer Zeit in der Größe der Anzahl der Knoten möglich.

Wenn wir G_{S_i} betrachten, dann stellen wir fest, dass $|V| \in O(L)$ und $|E| \in O(L^2)$ liegen

aufgrund der zuvor berechneten transitiven Hülle. Für die Berechnung des längsten Pfads folgt somit eine Laufzeit von $O(L^2)$.

4.3.4 Verankerungen

Morgenstern *et al.* (2006) haben einen semiautomatisierten Ansatz für das *multiple sequence alignment*-Problem entwickelt. Bei diesem kann der Benutzer, der über Expertenwissen verfügt, vorgeben, welche Positionen der Sequenzen auf jeden Fall miteinander *aligniert* werden sollen. Das ist insbesondere dann nützlich, wenn es Abschnitte gibt, die sich mathematisch gesehen sehr ähnlich sind, deren Zuordnung biologisch aber falsch wäre. Ein Beispiel für solche Sequenzen haben wir mit den *Hoxgenen* des Pufferfisches im Abschnitt über die Schwächen von DIALIGN bereits kennengelernt. Es hat sich herausgestellt, dass das semiautomatisierte Verfahren bei vielen Tests auf der Referenzdatenbank BAliBASE bessere Ergebnisse geliefert hat, wenn die Startpunkte aller Motive als *Verankerungen* gesetzt wurden.

Das Verfahren funktioniert so, dass der Experte *Fragmente* (die auch nur paarweise Zuweisungen sein können) wählt und diese mit einer Dringlichkeit an DIALIGN übergibt. Wie im Standardalgorithmus werden dann gierig die *Fragmente* mit maximaler Dringlichkeit gewählt, die zu allen bereits gewählten *konsistent* sind. Zwischen diesen *Verankerungen* wird dann ganz normal DIALIGN durchgeführt, wodurch die restlichen Abschnitte automatisiert *aligniert* werden.

Corel *et al.* (2010) benutzten die bereits vorliegende Infrastruktur in DIALIGN, um die *Zuweisungsspalten*, die der Algorithmus von Pitschi geliefert hat, als *Verankerungen* ins *Alignment* zu integrieren. Weil diese Möglichkeit in unserer Implementierung nicht bereits vorhanden ist, werden wir die Resultate des *Min-Cut*-Ansatzes direkt mit Hilfe von $\mathrm{Edge\text{-}Addition}$ und dem *Alignmentgraphen* in unser Ergebnis integrieren. Neben dem Übergeben von *konsistenten Zuweisungsspalten* als *Verankerungen* hat man auch Versuche mit *partiellen Zuweisungsspalten* gemacht. Hier war die gierige Heuristik von DIALIGN aber nicht erfolgreich und die Ergebnisse waren schlechter als mit dem Algorithmus von Pitschi. Aus diesem Grund werden wir den zweiten Ansatz nicht weiter behandeln.

4.3.5 Komplexität

Der *Sukzessionsgraph* $SG(C)$ kann aus dem *Inzidenzgraphen* in $O(n \cdot L)$ Rechenschritten berechnet werden, weil es höchstens so viele *partielle Zuweisungsspalten* gegeben kann. Außerdem ist die Anzahl der direkten Nachfolger pro Sequenz auch durch L begrenzt, was eine obere Grenze für die Anzahl der Kanten liefert.

Die transitive Hülle für G^+ berechnen wir beispielsweise durch Breitensuche für jeden Knoten. Weil es $O(n \cdot L)$ Knoten geben kann und jeder dieser Aufrufe $O(n \cdot L)$ Rechenschritte benötigt, weil dies die maximale Anzahl an Kanten und Knoten ist, folgt hierfür die Laufzeit von $O(n^2 \cdot L^2)$. Überlegungen für einen effizienteren Algorithmus haben sich als fruchtlos erwiesen. Für diesen hätten wir zuerst eine topologische Sortierung berechnet und von hinten nach vorne bearbeitet. Die transitive Hülle eines Knotens u kann dabei durch Vereinigung aller Knoten v mit $(u, v) \in E$ sowie deren transitive Hülle berechnet werden. Im schlimmsten Fall habe ich $O(L)$ Ebenen in meinem Graphen mit je bis zu $O(n)$ Knoten und $O(n)$ Kanten in andere Ebenen. Es kommt pro Ebene also zu bis zu $O(n)$ vielen Vereinigungen. Weil die Mengen, die miteinander vereinigt werden, aber bis zu $O(n \cdot L)$ Elemente enthalten (potentiell erreichbare Knoten), komme ich selbst unter Verwendung

von Bit-Sets und Vereinigung über das bitweise Oder auf Kosten von $O(n^2 \cdot L)$ pro Ebene. Auf Grund der $O(L)$ Ebenen bleibt es bei der Laufzeit von $O(n^2 \cdot L^2)$, selbst wenn dieser Algorithmus in der Praxis sehr schnell sein dürfte. Weil ich im Allgemeinen nicht davon ausgehen kann, dass mein Graph signifikant viel weniger starke Zusammenhangskomponenten als Knoten insgesamt enthält, liefert auch der Algorithmus von Purdom keine bessere Laufzeit.

Für das Konstruieren der Graphen G_{S_i} unserer Sequenzen $S_1, \ldots, S_n$ benötigen wir jeweils lineare Zeit in der Größe des Graphen G^+, um zu überprüfen welche Knoten *Stellen* der jeweiligen Sequenz enthalten und welche Kanten die Relation $\mathcal{R}_{S_i}$ erfüllen. Zwar kann es grundsätzlich $O(n \cdot L)$ Knoten geben und jeder Knoten bis zu $O(n)$ *Stellen*, aber die Gesamtanzahl an *Stellen* ist durch $n \cdot L$ begrenzt. Bei n Sequenzen folgt die Laufzeit in $O(n^2 \cdot L)$.

Die längsten Pfade für alle Sequenzen benötigen n-mal $O(L^2)$. Zudem müssen die nicht besuchten *Stellen* für jede Sequenz aus ihren Knoten gelöscht werden. Dazu iterieren wir wieder n-mal über die Graphen mit bis zu $O(L)$ Knoten. Wie schnell es möglich ist die *Stellen* aus dem dazugehörigen Knoten zu löschen, hängt von der intern verwendeten Datenstruktur ab. Im schlimmsten Fall brauchen wir $O(n)$ Zeit für die Entfernung einer *Stelle* aus einem Knoten. Alle Löschoperationen sind dementsprechend in $O(n^2 \cdot L)$ Rechenschritten möglich.

Insgesamt dominiert das Berechnen der transitiven Hülle die Komplexität des Algorithmus von Pitschi mit $O(n^2 \cdot L^2)$. Weil der vorangegangene Schritt mit den *minimalen Schnitten* aber ohnehin eine deutlich höhere Komplexität hat, ist es müßig hier nach effizienteren Ansätzen zu suchen.

4.4 Abschluss und Zusammenfassung

Nach dem Algorithmus von Pitschi liegen uns *konsistente Zuweisungsspalten* vor. Diese können aufgrund ihrer *Konsistenz* direkt in das finale *Alignment* eingefügt werden. Trotzdem müssen wir die *Transitivitätsgrenzen* mit dem *Alignmentgraphen* verwalten. Das liegt daran, dass wir wie bei DIALIGN zwischen den ursprünglichen *Konsistenzgrenzen* auf den Teilsequenzen DIALIGN benutzen. Auch hier kann davon ausgegangen werden, dass die Anzahl der Durchläufe konstant ist. Zu guter Letzt wird die Ausgabe vorbereitet. Das Vorgehen ist an dieser Stelle identisch mit dem der ursprünglichen DIALIGN-Implementierung, das in Kapitel 3 beschrieben wurde.

4.4.1 Gesamtkomplexität

4.4.1 Korollar
Mit dem *Min-Cut*-Ansatz von Corel *et al.* (2010) lässt sich ein multiples *Sequenzalignment* in $O(n^4 \cdot L^{7/2})$ Zeit berechnen.

Beweis. DIALIGN berechnet die $O(n^2)$ paarweisen *Alignments* in $O(n^2 \cdot L^2)$ Rechenschritten. Die Laufzeit der Berechnung der *partiellen Zuweisungsspalten* mit Hilfe des *minimalen Schnitts* liegt in $O(n^4 \cdot L^{7/2})$. Der Algorithmus von Pitschi hat eine Komplexität in $O(n^2 \cdot L^2)$. Für den *Alignmentgraphen* sind nach wie vor $O(n^3 \cdot L^2 + n^2 \cdot L^2)$ Berechnungen nötig. Es folgt, dass der *minimalen Schnitte* auf dem *Inzidenzgraphen* die Laufzeit dominiert, womit sich eine Gesamtkomplexität von $O(n^4 \cdot L^{7/2})$ ergibt. □

4.4.2 Probleme bei der Heuristik zum Entfernen von Kanten

Die Heuristik zum Entfernen von Kanten kann bei einigen Sequenzen zu größeren Problemen führen. Diese Heuristik funktionierte so, dass sie eine Grenze k für das minimale Kantengewicht sukzessive erhöhte und alle Kanten unter k entfernte, bis die Menge an Kanten ohne Zyklus war.

In den meisten Fällen ist das kein Problem, weil man davon ausgeht, dass die wichtigen Motive, die wir finden wollen, in der gleichen Reihenfolge innerhalb der Eingabesequenzen liegen. Problematisch sind hingegen Sequenzfamilien, bei denen zwei (oder mehr)größere Abschnitte in ihrer Reihenfolge vertauscht wurden und einige Sequenzen diese in der einen, die anderen aber in den der anderen Reihenfolge haben. Grafik einfügen.

Sind die vertauschten Abschnitte lang genug, dann werden sie Teil der paarweisen *Alignments*. Weil es sich bei ihnen um Überkreuzungen handelt, entfernt der *Min-Cut*-Ansatz sie im Allgemeinen auch nicht. Kommen diese Permutationen in genug Sequenzen vor (beispielsweise allen), dann ist das Ergebnis ein *Sukzessionsgraph* mit einem Zyklus, der Kanten mit sehr hohen Kantengewichten enthält. Um diese Zyklen aufzulösen, wird dann auch eine sehr hohe Grenze k^* benötigt. Im schlimmsten Fall werden so alle Kanten im Graph gelöscht, in jedem Fall aber sehr viele, die nichts mit dem eigentlichen Zyklus zu tun haben, weil der Schritt global Kanten löscht. Somit gehen viele wichtige oder alle Verbindungen in unserem *Alignment* verloren. Wir wollen am liebsten, dass die überkreuzten Stellen nicht aligniert werden, der Rest hingegen ganz normal.

Bis jetzt haben wir dieses Problem nur auf theoretischer Ebene betrachtet. Es gibt aber Sequenzfamilien, bei denen ein solches Verhalten wirklich auftritt. Ein Beispiel dafür ist die sogenannte *zirkuläre Permutation* bei Proteinen. Bei diesen kommen zwar die selben Abschnitte, aber in veränderter Reihenfolge vor. Ein Beispiel sind die abstrahierten Proteinsequenzen [A,B,C,D] und [B,C,D,A]. Erstmalig wurde dieses Phänomen von Cunningham *et al.* (1979) beschrieben. Wenn es im Laufe der Zeit zu einer zirkulären Permutation gekommen ist, deren Ergebnis zwei Sequenzen sind und wir eine Sequenzfamilie vorliegen haben, bei der die eine Hälfte von der einen und die andere von der anderen dieser Sequenzen abstammt, dann kann es durchaus zum oben beschriebenen Verhalten kommen.

Mit einer verbesserten Heuristik zum Entfernen von Kanten aus dem *Sukzessionsgraph*, um die Zyklen aufzulösen, könnte man diesem Problem vorbeugen. Ein solcher Algorithmus wurde beispielsweise von Eades *et al.* (1993) beschrieben, der in $O(|E|)$ Zeit einen DAG berechnet und dabei höchstens $|E|/2 - |V|/6$ Kanten entfernt. Leider entfernt auch dieser im Allgemeinen sehr viele Kanten, die nichts mit dem Zyklus zu tun haben. Weil man bei nahezu allen Sequenzfamilien davon ausgehen kann, dass Permutationen kein Problem sind, halte ich die ursprüngliche Heuristik zunächst für praktikabler. Denkbar ist aber ein hybrider Ansatz, bei dem wir überprüfen wie viele Kanten E_{k^*} im Verhältnis zur alten Kantenanzahl enthält. Ist dieses Verhältnis zu gering, benutzen wir stattdessen den Algorithmus von Eades, Lin und Smyth auf den ursprünglichen Kanten unseres *Sukzessionsgraphen*.

4.4.3 Evaluierung

5 Programmierung

-Test, ob Kapazitäten 1 oder nach Ähnlichkeit besser sind -Testen, welche Pfadbestimmung bei Algorithmus von Pitschi am besten ist

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).

Ahuja, R., Kodialam, M., Mishra, A. und Orlin, J. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, Band 97:3:509–542 (1997).

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).

Cormen, T., Leiserson, C., Rivest, R. und Stein, C. *Introduction to Algorithms*. The MIT Press, 3rd Auflage (2009).

Cunningham, B., Hemperly, J., Hopp, T. und Edelman, G. Favin versus concanavalin A: Circularly permuted amino acid sequences. In *Proceedings of the National Academy of Sciences, USA*, Band 76:7, Seiten 3218–3222. Natl. Acad. Sci. USA (1979).

Dinitz, Y. Dinitz' Algorithm: The Original Version and Even's Version. In *Theoretical Computer Science. Lecture Notes in Computer Science*, Band 3895. Springer-Verlag (2006).

Eades, P., Lin, X. und Smyth, W. A fast and effective heuristic for the feedback arc set problem. Band 47:319–323 (1993).

Goldberg, A. und Tarjan, R. A new approach to the maximum-flow problem. *Journal of the ACM*, Band 35:4:921–940 (1988).

Goldberg, A. und Tarjan, R. Efficient Maximum Flow Algorithms. *Communications of the ACM*, Band 57:8:82–89 (2014).

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., Frech, K., Dress, A. und Werner, T. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics*, Band 14:3:290–294 (1997).

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).

Pitschi, F., Dechauvelle, C. und Corel, E. Automatic detection of anchor points for multiple sequence alignment. *BMC Bioinformatics*, Band 11:445 (2010).

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, 30. 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, 30. April 2018