Spaltenerzeugung für die multiple Sequenzalinierung

Bachelorarbeit in Mathematik
eingereicht an der Fakultät für Mathematik und Informatik
der Georg-August-Universität Göttingen
am 31. Januar 2013

von

Erik Pohl

Erstgutachter:

Jun.-Prof. Dr. Stephan Westphal

Zweitgutachterin:

Prof. Dr. Anita Schöbel

Inhaltsverzeichnis

1 Einleitung

Viele Bereiche der Biologie beschäftigen sich mit der Analyse von DNA-Sequenzen. Die DNA ist, ein aus den vier Basen Adenin, Thymin, Guanin und Cytosin bestehendes Molekül, welches in allen Lebenwesen als Träger der Erbinformationen dient. Unter anderem wird diese Information zum Bau der Proteine verwendet, welche die meisten Abläufe in einem Organismus steuern.
Mithilfe der DNA werden diese Informationen von einer Generation zur nächsten vererbt. Während der Replikation (Vervielfältigung) der DNA können, neben Replikationsfehlern, Mutationen auftreten. Mutationen können allgemein als Veränderung im Genom definiert werden. Es gibt unterschiedliche Arten von Mutationen, zum Beispiel können einzelne Basen verändert, ausgelöscht oder eingeschoben werden. Treten diese Mutationen in den für die Funktion wichtigen Bereich der DNA auf, führt dies meist zu nicht überlebensfähigen Nachfahren. Dennoch können sich einige Mutationen positiv auf die Anpassungsfähigkeit der Nachfahren auswirken, was zu einer höheren Überlebenswahrscheinlichkeit führen kann. Man geht davon aus, dass letztlich durch Mutationen und Anpassung die heutige Artenvielfalt entstanden ist.
Die Ähnlichkeit der DNA zweier Individuen oder Arten gibt somit Aufschluss über die Verwandtschaft. Analog dazu können, mithilfe der Ähnlichkeit in der Reihenfolge von Aminosäuren in Proteinen, Aussagen über Funktion, Sekundär- und Tertiärstruktur getroffen werden.
Um Aussagen über die Ähnlichkeit von Sequenzen zu treffen, benötigt man ein Verfahren um diese zu vergleichen. Ein einfacher Vergleich der Zeichenkette ist nicht möglich, da Mutationen auch zu Veränderungen in der Länge der Sequenzen führen können. Zum Vergleich zweier Sequenzen kann die Methode der paarweisen Alinierung verwendet werden. Um eine Alinierung berechnen zu können, aus der sich tatsächlich Unterschiede in der Sequenz, Funktion bzw. Struktur ableiten lassen, muss man viele lange Sequenzen miteinander alinieren. Dies führt auf das Problem der multiplen Sequenzalinierung. Man kann die multiple Sequenzalinierung (MSA) als Verallgemeinerung der paarweisen Alinierung betrachten. Es werden hier gleichzeitig mehr als zwei Sequenzen miteinander aliniert. Dies erlaubt Aussagen über die Gemeinsamkeiten der Sequenzen und Verwandtschaft von mehreren Individuen. Diese Verallgemeinerung erhöht die Komplexität des Problems, sodass das optimale Lösen sehr aufwändig ist. Es werden also heuristische Verfahren entwickelt, die einen gewissen Grad an Optimalität garantieren können.
1953 wurde der strukturelle Aufbau der DNA von James Watson und Francis Crick ent-

deckt und in dem Artikel „Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid“ [8] veröffentlich. Dies diente als Grundlage der Bioinformatik, welche sich vorallem mit dem Umgang großer Datenmengen befasst. Der erste Algorithmus zur Lösung des paarweisen Alinierungsproblems von DNA Sequenzen wurde bereits 1970 von Needleman und Wunsch in [2] vorgestellt. Eine Variation dieses Algorithmus wurde 1981 von Smith und Waterman in [9] vorgestellt. Dieser Algorithmus benutzt die Methode der dynamischen Programmierung und berechnet die paarweise Alinierung in polynomieller Zeit. Die multiple Sequenzalinierung ist, bis heute, eines der wichtigsten Probleme in der Bioinformatik (s. z.B. Lander et al. 1991). Als Bewertungsfunktion für die Kompabilität zweier Symbole in unterschiedlichen Sequenzen wird hier oft die sogenannte „Sum of Pairs“-Funktion verwendet. Es wurde in [11] gezeigt, dass das Problem der multiplen Sequenzalinierung mit dieser Funktion NP-vollständig ist. Historisch gesehen gibt es zwei Möglichkeiten der Herangehensweise: Globale Algorithmen, die die gegebene Sequenzen als ganzes miteinander alinieren und lokale Algorithmen, welche nach Abschnitten suchen, die in allen Sequenzen gleich sind und daraus eine Alinierung erstellen. Eine neue Herangehensweise ist die auch in [12] verwendete Möglichkeit, die Sequenzen zuerst paarweise zu alinieren und daraus eine multiple Alinierung zu erstellen. Es werden also die Vorteile beider Herangehensweisen kombiniert. In [12] wird ein Verfahren vorgestellt, um, mithilfe der paarweisen Alinierung und einem Min-Cut-Algorithmus, potentielle Spalten zu suchen, die in die endgültige multiple Alinierung übernommen werden können.
In dieser Bachelorarbeit wird ein Verfahren vorgestellt, diese Spalten mithilfe eines Spaltenerzeugungsalgorithmus und Greedy-Algorithmen zu suchen.
Die Arbeit gliedert sich in folgende Bereiche. Im ersten Teil werden die mathematischen Grundlagen gelegt, um im zweiten Teil das Problem der multiplen Sequenzalinierung mithilfe Linearer Programmierung zu beschreiben. Im dritten Teil wird das Simplex-Verfahren und das Verfahren der Spaltenerzeugung eingeführt, welche dann im vierten Teil auf die im zweiten Teil gefundene Formulierung angewandt wird. Des Weiteren werden hier Heuristiken für das Subproblem der Spaltenerzeugung entwickelt. Alle gefundenen Formulierungen und diskutierten Algorithmen wurden implementiert und miteinander verglichen. Eine ausführliche Diskussion der verwendeten Heuristik und die Ergebnisse sind im fünften Teil zu finden.

2 Einführung

2.1 Grundlagen und Definitionen

Um das in der Einleitung genannte Problem der multiplen Sequenzalinierung weiter untersuchen zu können werden einige Definitionen benötigt.

Definition 2.1 (**(Multiple) Alinierung**, nach [1])**.** Sei $S = \{S_1, \ldots, S_k\}$ eine Menge von Sequenzen über einem (endlichen) Alphabet Σ. Sei $\hat{\Sigma} = \Sigma \cup \{-\}$, wobei "$-$" eine Lücke in einer Sequenz bezeichnen soll. Eine Alinierung der Sequenzen S ist eine Menge $\hat{S} = \{\hat{S}_1, \ldots, \hat{S}_k\}$ von Sequenzen über $\hat{\Sigma}$ mit

- alle Sequenzen in $\hat{S}$ haben die gleiche Länge
- bis auf eingefügte "$-$" sind alle Sequenzen unverändert.

Für DNA-Sequenzen setzt man $\Sigma = \{A, T, G, C\}$ stellvertretend für Adenin, Thymin, Guanin und Cytosin.

Definition 2.2 (**Optimale Alinierung**)**.** Sei $\hat{S}$ eine Alinierung und $c : \hat{\Sigma} \times \hat{\Sigma} \to \mathbb{R}$ eine Funktion. Sei $s_{i,k}$ der k-te Buchstabe der i-ten Sequenz. Eine optimale Alinierung ist eine Alinierung, welche eine gegebene Kostenfunktion minimiert. Es wird im Folgenden die „Sum of Pairs"-Funktion

$$SP = \sum_{k=1}^{l} \sum_{i<j} c(s_{i,k}, s_{j,k})$$

betrachtet, wobei l die Länge der Sequenzen bezeichnet.

Die Funktion c bewertet also die Kompatibilität von zwei Symbolen und wird meistens so gewählt, dass gleiche Symbole keine Kosten und unterschiedliche Symbole hohe Kosten verursachen.

Beispiel 2.3. Gegeben seien die beiden (DNA)-Sequenzen S1=GGACTGGTCG und S2=GACTGTCG. Man wählt die Funktion c so, dass gleiche Symbole mit 0 und unterschiedlich Symbole mit 1 bewertet werden. Da die Sequenz $S2$ zwei Symbole kürzer ist als $S1$ müssen mindestens zwei Lücken eingefügt werden. Eine mögliche Alinierung, mit Kosten $SP = 5$, ist also

S1	G	G	A	C	T	G	G	T	C	G	
S2	G	-	A	C	T	G	T	C	G	-	
$c(s_{1,k}, s_{2,k})$	0	1	0	0	0	0	1	1	1	1	$SP = 5$

Durch Verschieben der Lücken in der unteren Sequenz kann man viele weitere Alinierungen erzeugen. Die optimale Alinierung bezüglich dieser Kostenfunktion ist, wegen der unteren Schranke, gegeben durch

S1	G	G	A	C	T	G	G	T	C	G	
S2	G	-	A	C	T	G	-	T	C	G	
$c(s_{1,k}, s_{2,k})$	0	1	0	0	0	0	1	0	0	0	$SP = 2$

Da mindestens zwei Lücken eingefügt werden müssen, damit die Sequenzen die gleiche Länge haben, entstehen mindestens Kosten von 2. Damit ist dies eine optimale Lösung.

Jede Alinierung kann auch mithilfe eines Graphen $G = (V, E)$ dargestellt werden. Hierzu benötigt man die folgende

Definition 2.4 (**Alinierungsgraph**, nach [1])**.** Sei wieder $S = \{S_1, \ldots, S_k\}$ eine Menge von Sequenzen über einem (endlichen) Alphabet Σ. Ein Knoten im Graph repräsentiert genau ein $s_{i,k}$. Kanten existieren nur zwischen Knoten unterschiedlicher Sequenzen und bedeuten, dass die entsprechenden Symbole miteinander aliniert werden.

2.2 Der paarweise Alinierungsgraph

Das Problem der paarweisen Alinierung von Sequenzen kann effizient und optimal gelöst werden. Ein bekannter Algorithmus zur Lösung dieses Problems ist der von Needleman und Wunsch der bereits 1970 in [2] vorgestellt wurde. Der Algorithmus verwendet die Methode der dynamischen Programmierung und kann mithilfe von Scoring-Modellen angepasst werden.
Die paarweise Alinierung aller Sequenzen wird als Ausgangspunkt verwendet um den paarweisen Alinierungsgraph wie folgt zu definieren.

Definition 2.5 (**Paarweise Alinierungsgraph**, nach [3])**.** Der paarweise Alinierungsgraph enthält alle optimalen, paarweisen Alinierungen der gegebenen Sequenzen. Die Kanten erhalten ein Gewicht, welches die Güte der zu alinierenden Symbole angibt.

Die Idee hinter der paarweisen Alinierung und dem Speichern in einem Graphen ist, dass man nun versucht, den paarweisen Alinierungsgraphen in einen Alinierungsgraphen zu

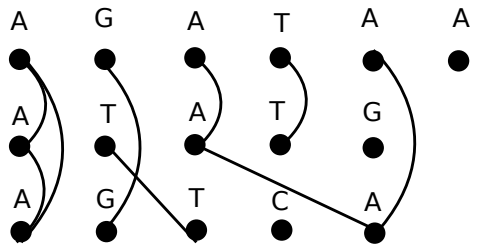

Abbildung 1: Paarweiser Alinierungsgraph von 3 Sequenzen, ohne Kantengewichte

überführen. Das heißt, man versucht aus paarweisen Alinierungen eine multiple Alinierung zu erstellen mit dem Ziel, auf diese Weise nahe an das optimale MSA zu gelangen.
Um den paarweisen Alinierungsgraphen untersuchen zu können benötigt man noch folgende Definition.

Definition 2.6 (**Komponente**)**.** Sei $G = (V, E)$ ein Graph. Durch die Bedingung: „ $v_1, v_2 \in V$ sind durch einen Weg verbunden“ ist eine Äquivalenzrelation auf V definiert. Eine Äquivalenzklasse $U \subset V$ heißt Komponente von G.

Der Alinierungsgraph ist charakterisiert durch folgende Bedingungen:

1. Kanten existieren nur zwischen Knoten unterschiedlicher Sequenz
2. Jeder Knoten ist mit maximal einem Knoten aus einer anderen Sequenz verbunden
3. Es muss möglich sein, alle miteinander alinierten Symbole, duch Einschieben von Lücken, in eine Spalte zu „schieben“, s. dazu [12]
4. Jede Komponente enthält maximal einen Knoten pro Sequenz

Falls Bedingung 3 verletzt ist, ist es nicht mehr möglich das Überkreuzen durch Einschieben von Lücken zu beseitigen und dadurch eine gültige multiple Alinierung zu erstellen (siehe dazu Abb 2). In einem paarweisen Alinierungsgraphen sind Bedingung 1 und 2 nach Konstruktion immer erfüllt. Für Bedingung 3 und 4 kann aber nicht garantiert werden. Das Ziel ist es also jetzt, Kanten aus dem paarweisen Alinierungsgraphen auszuwählen und in die multiple Alinierung zu übernehmen. Als Zielfunktion soll die Summe der Kantengewichte der übernommenen Kanten maximiert werden. Andersherum kann man auch Kanten, die Bedingung 3 oder 4 verletzen, aus dem paarweisen Alinierungsgraphen löschen. Dann ist es natürlich sinnvoll, die Summe der Kantengewichte der gelöschten Kanten zu

minimieren.

$$\sum_{e \in E^{\times}} w(e) \to \min$$

Wobei $E^{\times}$ die Menge der gelöschten Kanten bezeichnen soll. Diese Sichtweise soll beibehalten werden und im Folgenden soll nur noch Bedingung 4 betrachtet und untersucht werden.

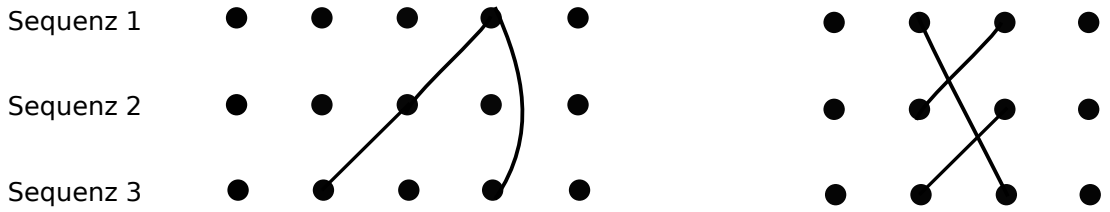

Abbildung 2: Minimalbeispiel zu Bedingung 3 und 4: Beide Fälle können im paarweisen Alinierungsgraphen auftreten, dürfen aber nicht ins MSA übernommen werden

3 Formulierung als lineares Programm

In diesem Kapitel sollen drei Möglichkeiten vorgestellt werden, das oben eingeführte Problem als lineares Programm zu formulieren.

Definition 3.1 (**Lineares Programm**, nach [4])**.** Sei A eine Matrix, $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ Spaltenvektoren. Das Ziel in der linearen Programmierung ist, entweder das Finden eines Vektors $x \in \mathbb{R}^n_+$, sodass $Ax \geq b$ und $c^t x$ ist minimal, zu entscheiden, dass $\{x \in \mathbb{R}^n_+ : Ax \geq b\}$ leer ist oder zu entscheiden, dass das Problem unbeschränkt ist. Eine Instanz des obigen Problems heißt lineares Programm (LP). Man schreibt häufig

$$\begin{aligned} \text{(LP)} \quad & \min c^t x && (3.2)\\ & \text{s.t. } Ax \geq b && (3.3)\\ & \quad x \geq 0. && (3.4) \end{aligned}$$

Eine zulässige Lösung eines linearen Programms ist ein Vektor x mit $Ax \geq b$. Eine Lösung, die das Minimum annimmt, heißt optimale Lösung.

Ein sehr bekannter Algorithmus zur Lösung linearer Programme ist der von George Dantzig stammende Simplex-Algorithmus (s. Kapitel 3). Es ist oft leichter, kombinatorische Optimierungsprobleme als ganzzahliges oder binäres lineares Programm zu formulieren.

Definition 3.5 (**Ganzzahliges lineares Programm**, nach [4])**.** Sei $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$. Ein lineares Programm der folgenden Form heißt ganzzahliges lineares Programm.

$$\begin{aligned} \text{(IP)} \quad & \min c^t x && (3.6)\\ & \text{s.t. } Ax \geq b && (3.7)\\ & \quad x \in \mathbb{Z}^n_+ && (3.8) \end{aligned}$$

Ein binäres lineares Programm ist ein Spezialfall des ganzzahligen lineares Programms.

$$\begin{aligned} \text{(BP)} \quad & \min c^t x && (3.9)\\ & \text{s.t. } Ax \geq b && (3.10)\\ & \quad x \in \{0,1\}^n && (3.11) \end{aligned}$$

Das Finden einer (beweisbaren) Optimallösung für ein Programm der Form (IP) oder der Form (BP) ist ein NP-schweres Problem (siehe z.B. [4], Theorem 15.40). Es gibt

exakte Lösungsverfahren, wie zum Beispiel das „branch and bound“-Verfahren oder das Schnittebenen-Verfahren. Dennoch ist das Lösen dieser Programme eine noch immer schwere Aufgabe und man versucht, durch das Ausnutzen der problemspezifischen Struktur der Programme, angepasste Algorithmen zu entwickeln, welche weniger Rechenzeit benötigen. Trotzdem sollen im Folgenden drei mögliche exakte und relaxierte Formulierungen für das Problem der multiplen Sequenzalinierung vorgestellt werden.

3.1 (BP1): Eine exakte Formulierung

Gegeben seien n Sequenzen, die paarweise optimal aliniert sind. Es sei $G = (V, E)$ der paarweise Alinierungsgraph. Ziel ist es, aus den paarweisen Alinierungen eine multiple Alinierung zu erstellen.

Hierzu führt man für jede Kante e_{ij}, $i, j \in V$ eine binäre Variable $\alpha(e_{ij})$ ein, die anzeigt, ob die jeweilige Kante aus dem Graph entfernt ($\alpha(e_{ij}) = 0$) wird, oder nicht. Als Zielfunktion wird die Summe der Kantengewichte der entfernten Kannten minimiert. Um nun sicherzustellen, dass am Ende alle Komponenten nur noch maximal einen Knoten pro Sequenz enthalten, benutzt man die folgende zyklische Bedingung.

Es sei $\hat{E} = \{e_{ij} : i, j \in V\}$ die Menge aller möglichen Kanten in dem Graph. Man setzt nun den Wert der Entscheidungsvariable aller Kanten innerhalb einer Sequenz auf Null und fordert $\alpha(e_{ij}) + \alpha(e_{ik}) \leq 1 + \alpha(e_{kj}) \quad \forall e_{ij} \neq e_{ik} \in \hat{E}$. Zur Verdeutlichung betrachte man die ungültige Komponente in Abbildung 3.1 und nehme an, dass $\alpha(e_{ij}) = 1$ für alle Kanten in dieser Komponente gilt.

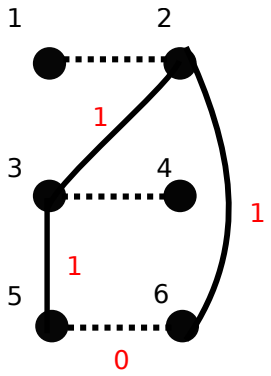

Abbildung 3: Ungültige Komponente

Wegen $\alpha(e_{23}) = \alpha(e_{35}) = 1$ muss $\alpha(e_{25}) = 1$ sein. Da aber $\alpha(e_{56}) = 0$ ist, ist die Bedingung in dem Dreieck mit den Knoten $\{5, 2, 6\}$ verletzt. Das komplette Programm hat die folgende Form:

$$\text{(BP1)} \quad \min \sum_{e_{ij} \in E} (1 - \alpha(e_{ij}))w_{ij} \tag{3.12}$$

$$\text{s.t.} \quad \alpha(e_{ij}) = 0 \qquad \forall e_{ij} \text{ in einer Sequenz} \tag{3.13}$$

$$\alpha(e_{ij}) + \alpha(e_{ik}) \leq 1 + \alpha(e_{kj}) \qquad \forall e_{ij} \neq e_{ik} \in \hat{E} \tag{3.14}$$

$$\alpha(e_{ij}) \in \{0, 1\} \tag{3.15}$$

Das Problem, Kanten aus einem paarweisen Alinierungsgraph, die Bedingung 3 verletzten, zu löschen, wird von diesem Programm optimal gelöst. Für die Anwendung ist dieses Programm aber nicht geeignet, da der Speicherbedarf und die Laufzeit zu hoch sind. Es sei m die Anzahl der Knoten in dem Graph. Die Menge $\hat{E}$ enthält somit $(m-1)!$ Kanten. Für jede dieser Kanten wird eine Entscheidungsvariable angelegt. Damit ergibt sich die Anzahl der Nebenbedingungen zu

$$\begin{pmatrix} (m-1)! \\ 3 \end{pmatrix},$$

nämlich die Anzahl der Möglichkeiten aus $(m-1)!$ Elementen, ohne Beachtung der Reihenflolge, Tripel zu bilden. Wenn man dieses Programm implementiert und davon ausgeht, dass jede Nebenbedingung eine Speicherstelle benötigt, die addressiert werden muss, dann kann es ein 64-bit Computer Graphen mit $|V| = 12$ nicht mehr lösen, denn

$$\begin{pmatrix} 11! \\ 3 \end{pmatrix} \approx 1,06 \times 10^{22} > 1,84 \times 10^{19} \approx 2^{64}$$

Da einzelne DNA-Sequenzen deutlich über 11 Basen besitzen, ist es nötig, andere Formulierungen zu finden, die vielleicht besser gestellt sind. Im folgenden Abschnitt wird eine Möglichkeit vorgestellt, die auf die Optimalität der Lösung verzichtet.

3.2 (BP2): Eine relaxierte Formulierung

Die Idee hinter der folgenden Formulierung ist, eine bestimmte Anzahl an Komponenten vorzugeben, die von dem Programm „befüllt“ werden dürfen. Als binäres lineares Programm kann man das wie folgt formulieren: Seien $K_1, \ldots, K_m$ m Komponenten. Es sei

$$\delta(v, K) = \begin{cases} 1, & v \text{ liegt in } K \\ 0, & \text{sonst} \end{cases}$$

die Indikatorfunktion, die anzeigt, ob der Knoten v in der Komponente K liegt und für $e \in E$ seien $v_1(e)$ und $v_2(e)$ die zugehörigen Knoten. Des Weiteren sei V_i die Menge der Knoten aus der i-ten Sequenz. Hiermit:

$$\text{(BP2)} \quad \min \sum_{e \in E} c(e)w(e) \tag{3.16}$$

$$\text{s.t.} \sum_{k=K_1}^{K_m} \delta(v,k) = 1 \qquad , \forall v \in V \tag{3.17}$$

$$\sum_{v \in V_i} \delta(v,k) \leq 1 \qquad , \forall k = K_1, \ldots, K_m, \forall i = 1, \ldots, n \tag{3.18}$$

$$\delta(v_1(e),k) - \delta(v_2(e),k) \leq c(e), \ \forall e \in E, k = K_1, \ldots, K_m \tag{3.19}$$

$$\delta(v_2(e),k) - \delta(v_1(e),k) \leq c(e), \ \forall e \in E, k = K_1, \ldots, K_m \tag{3.20}$$

$$c(e) \in [0,1], \qquad \forall e \in E \tag{3.21}$$

$$\delta(v,k) \in \{0,1\}, \qquad \forall v \in V, k = K_1, \ldots, K_m. \tag{3.22}$$

Die Variable $c(e)$ gibt an, ob die Kante e entfernt ($c(e) = 1$) werden muss oder in die Alinierung übernommen werden kann ($c(e) = 0$). Man minimiert über die Summe der Kantengewichte der entfernten Kanten (Gl. 3.16). Unter den folgenden Nebenbedingungen: Gleichung 3.17 stellt sicher, dass jeder Knoten in genau einer Komponente liegt, Gleichung 3.18 stellt sicher, dass jede Komponente maximal einen Knoten jeder Sequenz enthält. Die Bedingungen 3.19 und 3.20 erzwingen, dass die zu einer Kante gehörenden Knoten jeweils in der gleichen Komponente liegen. So wie gerade beschrieben, muss in Gleichung 3.21 gefordert werden, dass $c(e) \in \{0,1\}$ ist. Aufgrund der Zielfunktion und den Bedingungen 3.19 und 3.20 wird die Variable $c(e)$ trotzdem nur Werte in $\{0,1\}$ annehmen. Die Fortsetzung von $\{0,1\}$ auf $[0,1]$ reduziert die Rechenzeit, die ein Solver benötigt, um dieses Programm zu lösen. Durch das Einschränken der Anzahl zur Verfügung stehender Komponenten verliert man die Optimalität der Lösung. Denn angenommen, um die Bedingungen 3.17 und 3.18 zu erfüllen, werden 2 unterschiedliche Knoten, die durch eine Kante e verbunden sind, in unterschiedliche Komponenten gesetzt, dann wird durch Bedingung 3.19 oder 3.20 die Kante e gelöscht, obwohl es vielleicht möglich gewesen wäre, den beiden Knoten eine eigene Komponente zuzuweisen. Man muss beachten, dass die Anzahl der Komponenten größer als die Anzahl der Sequenzen gewählt werden muss, da sonst Bedingung 3.18 von Anfang an nicht erfüllt werden könnte. Wird m größer oder gleich der Anzahl an Komponenten gewählt, die ein optimaler Algorithmus benötigt um die Knoten aufzuteilen, erhält man

auch mit (BP2) die optimale Lösung.

Lemma 3.23 (Symmetriereduzierung)**.** *Um die Rechenzeit zur Lösung des Programms zu verkürzen können einige der $\delta(v, K)$, von vornherein, wie folgt festgelegt werden. Hierdurch wird die Lösung nicht verändert.*

$$\delta(v_1, K_j) = 0 \qquad \forall j = 2, \ldots, m \tag{3.24}$$

$$\delta(v_2, K_j) = 0 \qquad \forall j = 3, \ldots, m \tag{3.25}$$

$$\vdots$$

$$\delta(v_{m-1}, K_j) = 0 \qquad j = m \tag{3.26}$$

Beweis. Man beachte zuerst, dass es keine Rolle spielt, in welcher Komponente K_i sich ein Knoten befindet, es kommt nur darauf an, welche anderen Knoten sich noch in dieser Komponente befinden. Beweis durch vollständige Induktion über die Anzahl der Knoten.
Induktionsanfang: Es sei $|V| = 1$. Dann kann der einzige Knoten in die Komponente K_1.
Induktionsannahme: Es gelte diese Aussage bis zu einem $(n-1) \in \mathbb{N}$.
Induktionsschritt: Der neue Knoten kann entweder zu einer der Komponenten K_i, $i = 1, \ldots, m-1$ hinzugefügt werden, oder zu der noch nicht verwendeten Komponente K_m. □

3.3 (BP3): Eine dritte Formulierug

Für eine dritte Formulierung soll der Gedanke, die Komponenten des Graphen zu betrachten, fortgeführt werden. Hierzu ist es nötig, den Begriff der Komponente etwas zu erweitern: Mit einer Komponente wird im Folgenden eine Menge zusammenhängender Knoten bezeichnet. Insbesondere ist ein einzelner Knoten auch eine Komponente. In der folgenden Formulierung sollen alle Komponenten des paarweisen Alinierungsgraphs betrachtet werden. Ein lineares Programm trifft dann die Entscheidung, welche Komponenten in die multiple Alinierung übernommen werden. Hierzu werden die Kosten einer Komponente wie folgt definiert:

$$\delta(K) = \sum_{e=\{u,v\}:u\in K, v\notin K} w(e) \tag{3.27}$$

Das heißt, falls Komponente K in die multiple Alinierung übernommen werden soll, fallen Kosten in Höhe aller „durchgeschnittener“ Kanten an. Wie sich hier schon andeutet, wer-

den, als Zielfunktion in dem linaren Programm, die Kosten der benutzten Komponenten minimiert. Als Nebenbedingung muss nun sichergestellt werden, dass sich jeder Knoten in einer Komponente befindet und jede Komponente maximal einen Knoten pro Sequenz enthält. Es wird davon ausgegangen, dass die zulässigen Komponenten des Graphen bekannt sind. Das heißt, in einem ersten Schritt wurden die Komponeten des Graphen berechnet, so dass diese maximal einen Knoten pro Sequenz enthalten. Die Komponenten werden dann wie folgt in einer Matrix A gespeichert. Die n Knoten des Graphen seien durchnummeriert. Jede Zeile der Matrix entspricht dann einem Knoten, jede Spalte entspricht einer Komponente und es gilt $a_{ij} = 1$, falls sich Knoten i in Komponente j befindet, $a_{ij} = 0$ falls nicht. Als Beispiel betrachte folgenden Graph und Matrix A die alle zulässigen Komponenten enthält:

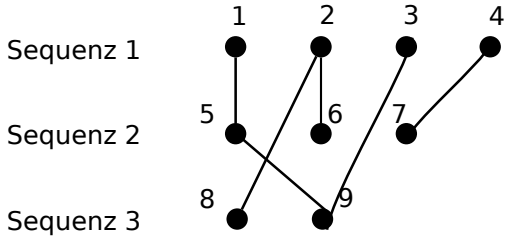

Abbildung 4: Beispielgraph mit 3 Sequenzen und 9 Knoten

$$A = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Die erste Spalte repräsentiert z.B. die Komponente die nur Knoten 1 enthält. Spalte 12 steht für die Komponente mit Knoten 1, 5 und 9. Man beachte, dass die Komponente mit den Knoten $\{1, 3, 5, 9\}$ nicht zulässig ist. Nun kann folgendes linerare Programm aufgestellt werden.

$$\text{(BP3)} \quad \min \ (\delta(K_1), \delta(K_2), \dots)x \tag{3.28}$$

$$\text{s.t.} \ Ax = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{3.29}$$

$$x_i \in \{0, 1\} \tag{3.30}$$

Mit K_j wird die Komponente bezeichnet, die der Spalte $A_{\cdot j}$ der Matrix entspricht. Gleichung 3.29 stellt sicher, dass sich jeder Knoten in genau einer Komponente befindet, da aufgrund der Gleichung 3.30 $x_i \in \{0, 1\}$ ist. Das heißt, falls $x_i = 1$ gesetzt wird, bedeutet das, dass Komponente i übernommen wird. Dabei fallen Kosten von $\delta(K_i)$ an.
Dieses lineare Programm löst das vorgestellte Problem, aus dem paarweisen Alinierungsgraphen Kanten zu löschen, so dass die Summe der Kantengewichte minimiert wird und eine gültige multiple Alinierung entsteht, optimal. Auch bei diesem Programm treten Probleme mit der Laufzeit und dem Speicherbedarf auf. Bevor dieses Problem von dem Programm gelöst werden kann müssen in einem vorangegangenen Schritt alle zulässigen Komponenten des Graphen berechnet werden. Des Weiteren hat die Matrix A für jede dieser Komponenten eine Spalte. Dies hat eine sehr hohe Anzahl an Entscheidungsvariablen x_i zur Folge.
Dieses Programm wird weiter unten wieder aufgegriffen und mithilfe eines Spaltenerzeugungsalgorithmus gelöst.

4 Spaltenerzeugung

4.1 Simplex

Das Simplex-Verfahren ist ein bekanntes Verfahren zur Lösung linearer Optimierungsprobleme. Das Verfahren löst ein solches Problem entweder in endlich vielen Schritten oder stellt fest, dass das Problem unlösbar oder unbeschränkt ist. Im Folgenden soll kurz auf den Simplex-Algorithmus eingegangen werden, um im Anschluss die Methode der Spaltenerzeugung einführen zu können.
Der Algorithmus wird zur Lösung linearer Programme der folgenden Form verwendet:

$$\min \; c^t x \tag{4.1}$$

$$\text{s.t.} \; Ax = b \tag{4.2}$$

$$x \geq 0. \tag{4.3}$$

wobei $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$. Man kann jedes lineare Programm durch Äquivalenzumformungen auf die obige Form bringen. Man geht davon aus, dass $m \leq n$ gilt. Zudem kann man annehmen, dass $Rang(A) = m$ ist. Falls dies nicht der Fall sein sollte, d.h. $Rang(A) < m$, enthält A linear abhängige Zeilen. Man erhält durch Entfernen dieser Zeilen ein äquivalentes System mit vollem Zeilenrang und $m \leq n$. Für das weitere Vorgehen benötigt man die folgende Definition.

Definition 4.4 (Basis). Es sei A eine $m \times n$-Matrix mit $Rang(A) = m$. Eine Indexmenge $B \subset \{1, \ldots, n\}$ heißt Basis, falls $|B| = m$ und die zugehörige Spaltenmenge $\{A_{B(1)}, \ldots, A_{B(m)}\}$ linear unabhängig ist. Man bezeichnet die entsprechende Teilmatrix mit A_B, dem Basisanteil von A. Analog heißt $N = \{1, \ldots, n\} \setminus B$ Nichtbasis und A_N Nichtbasisanteil von A.

Der Grundform des Simplex-Algorithmus wird nun ein lineares Programm und eine Basis B übergeben. Gleichung 4.2 und 4.3 bilden ein Polyender der zulässigen Lösungen. Ein Punkt x ist genau dann Extremalpunkt des Polyeders, wenn x eine Basislösung ist. Der Simplex-Algorithmus startet mit einer zulässigen Basislösung und wechselt so lange Variablen zwischen Basis und Nichtbasis aus, bis eine optimale Basis erreicht ist. D.h. man wandert auf dem Polyeder von einen Extremalpunkt zu einem benachbarten, ohne dabei den Wert der Zielfunktion zu verschlechtern, bis man einen optimalen Extremalpunkt

erreicht hat. Die eintretende Variable in einer Iteration wird dabei wie folgt bestimmt. Ist B eine Basis von Matrix A, so kann jeder Vektor $x \in \mathbb{R}^m$ in die Basisvariablen x_B und Nichtbasisvariablen x_N zerlegt werden, d.h. $x = (x_B|x_N)$. Damit erhält man aus der Gleichung 4.2

$$\begin{aligned} & Ax = b \\ \Leftrightarrow \quad & A_B x_B + A_N x_N = b \\ \Leftrightarrow \quad & x_B = A_B^{-1} b - A_B^{-1} A_N x_N. \end{aligned}$$

Dies verleitet zu folgender

Definition 4.5 (zulässige Basislösung). Es sei A eine $m \times n$- Matrix mit $Rang(A) = m$ und B eine Basis. Ein Vektor $x = (x_B|x_N)$ bestehend aus Basis- und Nichtbasisvariablen heißt Basislösung von Gleichung 4.2, falls $x_N = 0$ und $x_B = A_B^{-1} b$ gilt. Falls zusätzlich $x_B \geq 0$ gilt, so heißt x zulässig.

Nun kann die Zielfunktion geschrieben werden als

$$\begin{aligned} c^t x &= c_B^t x_B + c_N^t x_N \\ &= c_B^t (A_B^{-1} b - A_B^{-1} A_N x_N) + c_N^t x_N \\ &= c_B^t A_B^{-1} b + (c_N^t - c_B^t A_B^{-1} A_N) x_N \end{aligned} \tag{4.6}$$

Der Vektor

$$\tilde{c}_N = c_N^t - c_B^t A_B^{-1} A_N \tag{4.7}$$

enthält die sogenannten reduzierten Kosten. Falls die Einträge von $\tilde{c}_N$ positiv sind, ist der aktuelle Extremalpunkt optimal, da der Zielfunktionwert wegen $x \geq 0$ nicht mehr verkleinert werden kann. Angenommen ein Eintrag von $\tilde{c}_N$ ist negativ, so wird der Zielfunktionswert verkleinert, falls die entsprechende Variable in die Basis genommen wird. Wichtig bei der Wahl der eintretenden Variable ist also, dass der Zielfunktionswert nur verbessert wird, falls die reduzierten Kosten dieser Variable negativ sind. D.h. man wählt als eintretende Variable $j \in N$ mit $\tilde{c}_j < 0$ möglichst klein. Als verlassende Variable kann zum Beispiel

$$k^* = \arg\min_{k \in B} \left\{ \frac{x_{B_k}}{w_k} : w_k > 0 \right\},$$

mit $w = A_B^{-1} A_j$ und j wie oben, gewählt werden.

4.2 Spaltenerzeugung

In jeder Iteration des Simplexalgorithmus wird eine Variable gesucht, die den Zielfunktionswert verbessert, die dann in die Basis genommen wird. Man sucht somit eine Nichtbasisvariable deren reduzierte Kosten negativ sind. Eine Möglichkeit, eine solche zu finden, ist es, das Minimum über alle reduzierten Kosten zu berechnen. Mit dem dualen Vektor $\pi = c_B A_B^{-1}$ gilt es

$$\arg\min_{j \in N} \{\tilde{c}_j = c_j - \pi^t A_{\cdot j}\}$$

zu lösen, wobei $A_{\cdot j}$ wieder eine Spalte der Matrix A bezeichnet. Da Variablen mit negativen reduzierten Kosten gesucht werden, können die Spalten aus A_B hinzugenommen werden, denn die reduzierten Kosten dieser Spalten sind immer $c_B - c_B A_B^{-1} A_B = 0$. Mit $J = B \cup N = \{1, \ldots, n\}$ kann stattdessen

$$\arg\min_{j \in J} \{\tilde{c}_j = c_j - \pi^t a_j\}$$

gelöst werden. Diese Suche benötigt einen hohen Rechenaufwand, falls $|J|$ groß ist. Bei näherer Betrachtung fällt auf, dass es nicht nötig ist ein exlpizites $j \in J$ zu bestimmen. Es genügt, die Spalte $A_{\cdot j}$ zu bestimmen, die dann eine Spalte der Basismatrix A_B ersetzt. In der Anwendung entsprechen diese Spalten meist der Lösung eines kombinatorischen Problems, wie zum Beispiel Pfade in einem Graphen, Mengen oder Permutationen. Um das explizite Bestimmen von $j \in J$ zu vermeiden beginnt man mit einer kleinen Teilmenge $J' \subset J$ von Spalten und nennt dieses kleinere Problem das „Restricted Master Problem (RMP)“ zu dem Ausgangsproblem. Man betrachte weiterhin das Programm aus Gleichung 4.1 - 4.3 als Ausgangsprogramm (Master Problem). Umgeschrieben hat es die Form

$$\min \quad \sum_{j \in J} c_j x_j \tag{4.8}$$

$$\text{s.t.} \quad \sum_{j \in J} A_{\cdot j} x_j = b \tag{4.9}$$

$$x_j \geq 0 \quad j \in J. \tag{4.10}$$

Mit einer Teilmenge $J' \subset J$ erhält man analog das RMP

$$\min \sum_{j \in J'} c_j x_j \tag{4.11}$$

$$\text{s.t.} \sum_{j \in J'} A_{\cdot,j} x_j = b \tag{4.12}$$

$$x_j \geq 0 \quad j \in J'. \tag{4.13}$$

Angenommen dieses Programm hat eine zulässige Lösung. Sei x^* die primäre und π^* die duale Lösung des RMP und sei $\mathbb{A} = \{A_{\cdot,j} : j \in J\}$ die Menge aller Spalten. Falls der Kostenkoeffizient c_j der Spalte $A_{\cdot,j}$ in der Zielfunktion mithilfe einer Funktion c berechnet werden kann, so heißt

$$c^* := \min\{c(a) - (\pi^*)^t a : a \in \mathbb{A}\}$$

das Subproblem. Falls $c^* \geq 0$, gibt es kein negatives $\tilde{c}_j, j \in J$ und die Lösung x^* des RMP löst auch das Ausgangsproblem optimal (siehe Lemma 4.14). Andernfalls wurde eine Spalte gefunden, die den Zielfunktionswert des Probelms verbessern kann. Man kann die Spalte, die zur optimalen Lösung des Subproblems gehört, dem Restricted Master Problem hinzufügen und kann dieses nochmal lösen. Das oben beschriebene Verfahren heißt Spaltenerzeugung, da man bei jeder Iteration Spalten des Ausgangsprogramms erzeugt und sie dem kleineren Programm hinzufügt. Man löst also statt des Ausgangsprogramms, welches eine hohe Anzahl an Spalten hat, viele kleinere Programme mit weniger Spalten. Diese Methode zur Lösung linearer Programme bietet einen weiteren großen Vorteil. Zur Lösung des Subproblems ist es nicht nötig, alle Spalten der Matrix A_N zu kennen. Es ist ausreichend eine Funktion zu kennen, die eine Spalte mit negativen reduzierten Kosten liefert, oder anzeigt, dass keine solche existiert. Im folgenden Pseudocode sind nochmal die grundlegenden Schritte zusammengefasst.

Wie oben schon angedeutet, müssen die Initialisierung und das Lösen des Subproblems problemspezifisch implementiert werden. Die Lösung des RMP kann ein beliebiger LP-Solver übernehmen.

Lemma 4.14. *Die Lösung des Restricted Master Problems löst auch das Master Problem optimal.*

Procedure 1 Spaltenerzeugung

Input: Master Problem
Output: Optimale Lösung
  Initialisierung: Erstelle RMP mit zul. Lösung
  while Subproblem erzeugt Spalte mit $c^* < 0$ **do**
    füge Spalte zum RMP hinzu
    optimiere/löse RMP
  end while
  gebe Lösung des RMP zurück

Beweis. Es sei c^* die optimale Lösung des RMP. Dann gibt es kein $c' < 0$ mit $c' = \min\{c(a) - (\pi^*)^t a : a \in \mathbb{A}\}$. Wegen der Gleichungen 4.6 und 4.7 ist dies auch eine optimale Lösung des Master Problems. □

5 Anwendung auf das MSA-Problem

5.1 Das Restricted Master Problem und die Initialisierung

Wie am Ende von Kapitel 2 erwähnt, soll nun versucht werden, das dritte lineare Programm mthilfe eines Spaltenerzeugungs-Algorithmus zu lösen. Gegeben seien also n Sequenzen mit insgesamt m Symbolen. Diese Sequenzen seien paarweise optimal miteinander aliniert und der paarweise Alinierungsgraph sei gegeben durch $G = (V, E)$. Des Weiteren seien die Kanten des Graphen gewichtet mit $w(e_i) \geq 0$, welche die Güte der zu alinierenden Symbole angibt.
In (BP3) wurden nun in einem ersten Schritt die Komponenten von G berechnet und wie folgt in die Matrix A geschrieben:

$$a_{ij} = \begin{cases} 1, & i \text{ liegt in Komponente } j \\ 0, & \text{sonst} \end{cases} \tag{5.1}$$

Die Zeilen repräsentieren also die Knoten und die Spalten die Komponenten des Graphen G. Unser Master Problem hat also die folgende Gestalt.

$$\min \ (\delta(K_1), \delta(K_2), \dots)x \tag{5.2}$$

$$\text{s.t. } Ax = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{5.3}$$

$$x_i \in \{0, 1\} \tag{5.4}$$

Wobei $\delta(K)$, wie oben, die Kosten der Komponente K sind. Um dies in die Form 4.9 zu bringen schreibt man die Gleichung 5.3 um und erhält folgendes äquivalentes Programm.

$$\min \sum_{i \in J} \delta(K_i) x_i \tag{5.5}$$

$$\text{s.t.} \sum_{i \in J} A_{\cdot i} x = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{5.6}$$

$$x_i \in \{0, 1\} \tag{5.7}$$

Wobei J die Indexmenge der Komponenten des Graphen G bezeichnen soll. Um das restricted Master Problem aufzustellen, benötigt man eine Teilmenge der Spaltenmenge $J' \subset J$, für die eine zulässige Lösung existiert. Man wählt J' so, dass $A_{J'} = I_m$ die Einheitsmatrix ist. Dies ist möglich, da die Matrix A alle möglichen Komponenten des Graphen enthält, also insbesondere die Komponenten, die jeweils nur einen Knoten enthalten. Zusätzlich ist die Reihenfolge der Komponenten in der Matrix nicht relevant, so dass man die obige Wahl treffen kann. Das Ergebnis ist folgendes RMP:

$$\min \sum_{i=1}^{m} \delta(v_i) x_i \tag{5.8}$$

$$\text{s.t. } I_m x = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{5.9}$$

$$x_i \in \{0, 1\}. \tag{5.10}$$

Dabei bezeichnet v_i den i-ten Knoten im Graph. Folgendes Lemma garantiert die Lösbarkeit dieses Programms.

Lemma 5.11. *Das gewählte RMP ist für jeden Graph, bzw. für jede gegebene Menge an Eingabesequenzen lösbar.*

Beweis. Betrachte $x = (1, \ldots, 1)$. Diese Wahl erfüllt offensichtlich die Bedingung 5.9 und jeder Eintrag des Vektors x liegt in $\{0, 1\}$, so dass Bedingung 5.10 ebenfalls erfüllt ist. □

5.2 Das Subproblem

Da nun die Lösbarkeit des Restricted Master Problems sichergestellt ist, wird sich im Folgenden mit dem Subproblem befasst. Nach Definition ist das Subproblem gegeben durch

$$c^* := \min\{c(a) - (\pi^*)^t a : a \in \mathbb{A}\}.$$

In dem vorliegenden Fall entspricht $\mathbb{A}$ der Menge aller zulässigen Komponenten des Graphen G und $c(\cdot)$ der Kostenfunktion $\delta(\cdot)$. Eine Komponente heißt zulässig, falls sie maximal einen Knoten pro Sequenz enthält. Man erhält also für die reduzierten Kosten einer Komponente $K \in \mathbb{A}$ folgenden Ausdruck. Es sei dazu j der Index der Spalte in der Matrix A, die der Komponente K entspricht und sei $\tilde{K} = \{i \in K : a_{ij} = 1\}$. Hiermit erhält man

$$\bar{c}_j = c(K) - \pi^t A_{\cdot j} = \delta(K) - \pi^t A_{\cdot j} \overset{5.1}{=} \delta(K) - \sum_{i \in \tilde{K}} \pi_i.$$

Man kann die reduzierten Kosten einer Komponente des Graphen G nun mithilfe der Kostenfunktion $\delta(\cdot)$ und den Dualvariablen π berechnen. Das Subproblem hat also die folgende Gestalt:

$$\min\{\delta(K) - \sum_{i \in \tilde{K}} \pi_i : K \in \mathbb{A}\}.$$

Es ist nun möglich, mithilfe des Spaltenerzeugungs-Algorithmus aus Kapitel 3, dieses Problem zu lösen. Dennoch muss man in einem ersten Schritt alle Komponenten, also die Menge $\mathbb{A}$, des Graphen berechnen. Dies ist oft zeitaufwändig und die Speicherung der Komponenten benötigt unter Umständen viel Speicherplatz. Man kann dieses Problem umgehen, indem man von dem Vorsatz, eine optimale Lösung zu finden, abweicht und sich mit einer „guten“ Lösung zufrieden gibt.

5.3 Heuristik zur Lösung des Subproblems

Verfahren bzw. Algorithmen, die nach einer möglichst „guten“ Lösung von Problemen suchen, heißen Heuristik. Eine Heuristik bezeichnet also ein Vorgehen, bei dem man mit wenig Wissen und begrenzter Zeit, Schlussfolgerungen über ein System aufstellt, wobei die gefolgerten Aussagen von den Optimalen abweichen können. Falls die optimale Lösung des Problems bekannt ist, kann man durch Vergleich der Lösungen die Güte der verwendeten

Heuristik bestimmen. Im Folgenden sollen Heuristiken zur Lösung des Subproblems gefunden und betrachtet werden. Hierzu werden weiterhin die reduzierten Kosten einer Komponente des Graphegn G betrachtet. In jeder Iteration des Spaltenerzeugungs-Algorithmus wird eine Komponente des Graphen gesucht, deren reduzierte Kosten kleiner als Null sind. Die reduzierten Kosten einer Komponente setzen sich aus zwei Summanden zusammen. Zum einen die Kosten der Komponente

$$\delta(K) = \sum_{e=\{u,v\}:u\in K, v\notin K} w(e)$$

zum anderen die (negative) Summe der Dualvariablen in der Komponente

$$-\sum_{i\in\tilde{K}} \pi_i.$$

Da nach Komponenten mit negativen reduzierten Kosten gesucht wird, ist nun eine Strategie, die Komponenten K zu suchen, für die $\delta(K)$ möglichst klein ist und $\sum_{i\in\tilde{K}} \pi_i$ möglichst groß. Diese beiden Bedingungen sollen näher betrachtet werden und es soll versucht werden, die gesuchten Komponenten besser zu charakterisieren. Die Bedingung „$\delta(K) \to \min$“ heißt, möglichst wenig Kanten „durchzuschneiden“, bzw. nur die Kanten mit wenig Gewicht „durchzuschneiden“. Da die Komponenten nach Voraussetzung endlich viele Knoten haben, heißt dass, das man zum Beispiel möglichst große Komponenten wählt. Damit kann man im besten Fall $\delta(K) = 0$ erreichen. Da $\delta(K) \geq 0$ nach Definition ist, ist das der minimale Wert. Um die Summe der Dualvariablen zu maximieren benötigen wir folgende Vorüberlegung. Es sei m wie oben die Anzahl der Knoten im Graph G, d.h. $|V| = m$. Somit ist A eine $m \times *$-Matrix. Nach Definition der Dualvariablen hat π also auch m Einträge. Man kann die einzelnen Werte $\pi_i, i = 1, \ldots, m$ der Dualvariablen in jeder Iteration kanonisch den Knoten des Graphen zuordnen. Um nun Komponenten zu finden, die die Summe der Dualvariablen maximieren, ordnet man die einzelnen Werte dieser den Knoten zu und sucht z.B. nach Komponenten, die viele Knoten mit großen Werten enthalten. Man stellt fest, dass es auch hier sinnvoll ist, möglichst große Komponenten zu suchen.

Beispiel 5.12. Man betrachte erneut den Graph aus Abbildung 3.3 mit den Kantengewichten $w(e) = 1\ \forall e \in E$ und dem Dualvektor $\pi = (1, \ldots, 1)$. Es sei K_1 die Komponente mit den Knoten $\{1, 3, 5, 9\}$ und $K_2 = \{2, 6, 8\}$. Man erhält für die reduzierten Kosten der Komponenten $\bar{c}_1 = 0 - 4 = -4$ und $\bar{c}_2 = 0 - 3 = -3$.

Wie das Beispiel zeigt ist es nicht ausreichend nur die reduzierten Kosten zu betrach-

ten. Man muss zusätzlich sicherstellen, dass nur zulässige Komponenten gefunden werden. Hierzu wird im folgenden Kapitel ein sogenannter Greedy-Algorithmus entworfen.

5.3.1 Ein Greedy-Algorithmus

Ein Greedy-Algorithmus (greedy: engl. gierig) ist ein Algorithmus, der, zur Lösung eines Systems, schrittweise den Folgezustand auswählt, der in diesem Moment die bester Verbesserung oder den größten Gewinn garantiert. D.h. Algorithmen dieser Form suchen lokal nach Verbesserungen und übernehmen diese. Es gibt Greedy-Algorithmen, die optimale Lösungen zu Problemen finden, meistens ist dies jedoch nicht der Fall, da diese Algorithmen die „Details“ der alternativen Zustände nicht betrachten. Aus diesem Grund sind diese Greedy-Algorithmen deutlich schneller als exakte Agorithmen, die oft exponentiell viele Möglichkeiten überprüfen. Im vorliegenden Fall bietet sich ein Greedy-Algorithmus besonders an. Nach Voraussetzung sind die n Sequenzen paarweise optimal miteinander aliniert. Das heißt, im günstigsten Fall, ist ein zufällig ausgewählter Knoten des Graphen G genau einmal mit einem Knoten aus anderen Sequenzen aliniert. Betrachte dazu folgende Abbildung.

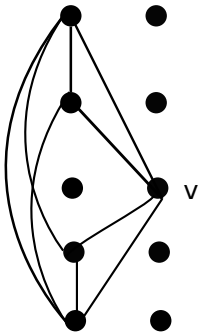

Abbildung 5: Mögliche Alinierung von 5 Knoten in 5 Sequenzen

Abbildung 5 zeigt einen Knoten v, der mit 4 Knoten aus unterschiedlichen Sequenzen aliniert ist. Die 4 Knoten sind auch jeweils miteinander aliniert. Falls man nun eine Komponente konstruiert, die aus allen zu v adjazenten Knoten besteht, erhält man eine Komponente, deren Kosten Null sind. D.h. eine Komponente, die wahrscheinlich die Lösung des RMP verbessert. Hier soll der Greedy Algorithmus ansetzen. Man wählt (zufällig) einen Knoten des Graphen aus, bildet aus diesem und allen adjazenten Knoten eine Komponente K und berechnet die reduzierten Kosten $\bar{c}(K)$ dieser Komponente. Falls $\bar{c}(K) < 0$, fügt man die Komponente dem RMP hinzu.

Procedure 2 Greedy 1

Input: Paarweiser Alinierungsgraph $G = (V, E)$ mit Dualvariablen
Output: Komponente K mit $\bar{c}(K) < 0$

```
Wähle (zufällig) Knoten v ∈ V
while true do
  K = {v} ∪ {v}
  for w ∈ V do
    if w adjazent zu v then
      K = K ∪ {w}
    end if
  end for
  if c̄(K) < 0 then
    return K
  else
    if ∃v' ∈ V : v wurde noch nicht gewählt then
      v = v'
    else
      return STOP
    end if
  end if
end while
```

Lemma 5.13. *Der Algorithmus Greedy 1 liefert immer zulässige Komponenten.*

Beweis. Da die Sequenzen paarweise optimal miteinander aliniert sind, ist ein gewählter Knoten $v \in V$ maximal mit einem anderen Knoten in jeder anderen Sequenz aliniert. □

5.3.2 Eine Verbesserung von Greedy 1

Die Annahme, dass ein ausgewählter Knoten schon einmal mit einem Knoten aus vielen anderen Sequenzen aliniert ist, ist ziemlich stark. Folgendes Beispiel zeigt, dass Greedy 1 nicht immer die größtmögliche Komponente findet.

Man betrachte den Beispielgraph aus Abbildung 6. Angenommen Greedy 1 wählt als Anfangsknoten v. Dann wäre die gefundene Komponente $K = \{u, v, w\}$ nicht so groß wie möglich. Man könnte die anderen beiden Knoten noch hinzunehmen, ohne die Zulässigkeit der Komponente zu verletzen. Die Verbesserung von Greedy 1 kann wie folgt aussehen. Nachdem eine Komponente gefunden wurde, wird in einem zweiten Schritt für jeden Knoten v in der Komponente überprüft, ob zu v adjazente Knoten noch in die Komponente genommen werden können, ohne die Zulässigkeit zu verletzen. Falls das der Fall ist, wird der Knoten hinzugenommen. Dies wird solange wiederholt, bis in einem Schritt kein weiterer

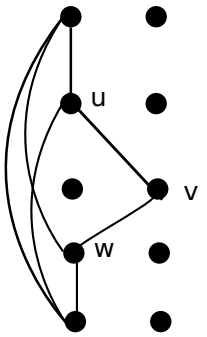

Abbildung 6: weitere mögliche Alinierung von 5 Knoten in 5 Sequenzen

Knoten in die Komponente aufgenommen wurde. Algorithmus 3 stellt das eben beschriebene nochmal als Pseudocode dar. Sei dazu $Adj(v)$ die Menge der zu Knoten v adjazenten Knoten und sei $Seq(v)$ die Sequenz, in der sich v befindet.

Procedure 3 Greedy 2

Input: Paarweisen Alinierungsgraph $G = (V, E)$ mit Dualvariablen
Output: Komponente K mit $\bar{c}(K) < 0$

```
Wähle (zufällig) Knoten v ∈ V
while true do
  K = Adj(v)
  S = {Seq(v) : v ∈ K}
  repeat
    for v ∈ K do
      for w ∈ Adj(v) do
        if Seq(w) ∉ S then
          K = K ∪ {w}
          S = S ∪ {Seq(w)}
        end if
      end for
    end for
  until kein neuer Knoten hinzugefügt
  if c̄(K) < 0 then
    return K
  else
    if ∃v' ∈ V : v' wurde noch nicht gewählt then
      v = v'
    else
      return STOP
    end if
  end if
end while
```

6 Implementierung

6.1 Benutzte Werkzeuge

Alle Algorithmen und Programme wurden unter Ubuntu Linux 10.04 und 12.04 entwickelt und getestet. Als Programmiersprache wurde *C++* und als Compiler *g++ Version 4.4.3 und 4.6.3* aus der GNU Compiler Collection verwendet. Des Weiteren spielt die *C++ API of ILOG Concert Technology* eine wichtige Rolle. Sie stellt eine C++ Bibliothek zur Modellierung und Lösung (ganzzahliger) linearer Programme zur Verfügung. Zudem bietet sie grundlegende Funktionen zur Programmierung von Spaltenerzeugungsalgorithmen und stellt einen Solver zur Lösung der Programme bereit. Für den leichteren Umgang mit Listen und Komponenten wurden die Bibliotheken *vector* und *set* aus den *C++*-Standardbibliotheken verwendet. Die linearen Programm (BP1) und (BP2) wurden mithilfe von *OPL* aus dem *IBM ILOG CPLEX Optimization Studio* implementiert.

6.2 Programmablauf, Klassen und Funktionen

Abbildung 7 stellt den Ablauf des Programms dar. Zur Speicherung des Graphen wurde eine entsprechende Klasse implementiert. Nachfolgend werden kurz die Klassen, ihre Attribute und Funktionen dargestellt.

- **Node:** Diese Klasse repräsentiert einen Knoten des Graphen. Ein Knoten hat folgende Attribute: Eine Liste (*list*) mit Referenzen zu inzidenten Kanten, sowie ein Liste (*vector*) mit Referenzen auf inzidente Knoten. Des Weiteren wird die Nummer des Knotens im Graphen und die Sequenz in der er sich befindet gespeichert. Es werden Funktionen zum Hinzufügen von inzidenten Knoten und Kanten bereitgestellt.

- **Edge:** Diese Klasse repräsentiert eine Kanten im Graphen. Als Attribute enthält sie jeweils eine Referenz auf den Anfangs-und Endknoten der Kante sowie ein Gewicht.

- **Graph:** Diese Klasse speichert den Graphen an sich. Die Klasse enthält jeweils Listen mit Referenzen auf die Knoten bzw. Kanten im Graphen, sowie Funktionen um Knoten und Kanten hinzuzufügen. Desweiteren wird die Anzahl der eingegebenen Sequenzen gespeichert.

Alle weiteren Funktionen wurden als solche ausgelagert. Die wichtigsten werden in der folgenden Tabelle 1 dargestellt.

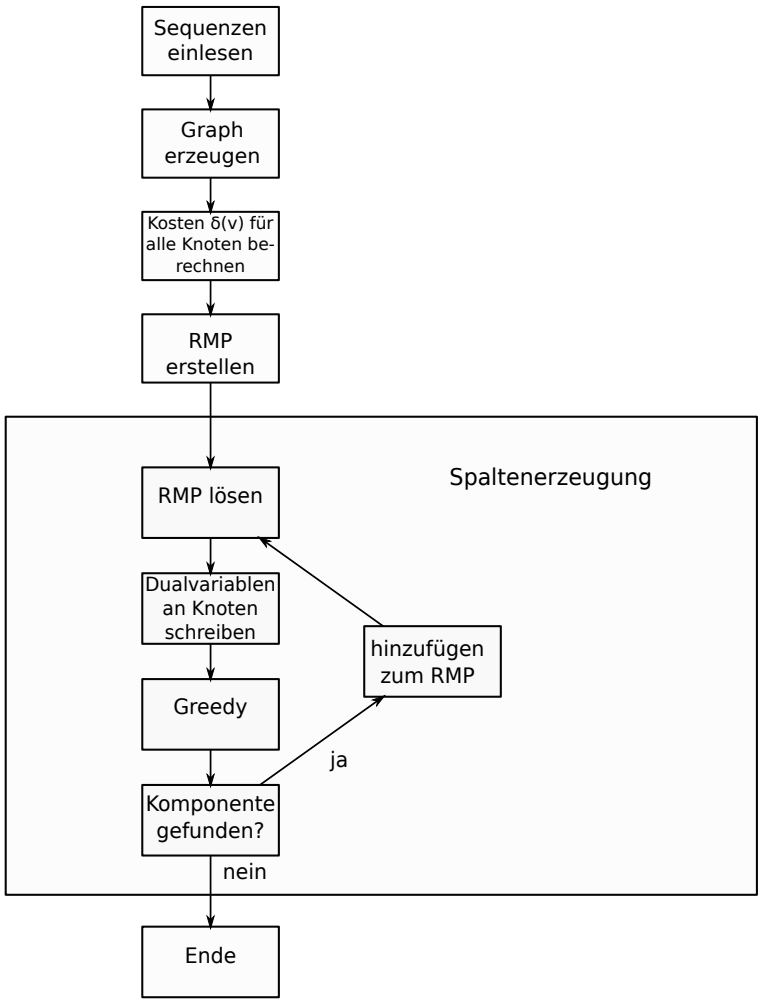

Abbildung 7: Programmablauf

6.3 Kompilierung und Eingabedateien

Für die Kompilierung des Programms genügt unter Ubuntu-Linux der Aufruf

```
make all
```

im entsprechenden Ordner. Eventuell muss der Pfad zur *CPLEX*-Installation und zur *CONCERT-TECHNOLOGY*-Installation im *makefile* angepasst werden. Das Programm kann nun mit

```
.\main DATEI
```

gestartet werden. Als Eingabedatei muss ein paarweiser Alinierungsgraph übergeben werden, der wie folgt in einer Textdatei gespeichert wurde:

Name	Verwendung/Funktion
main	Enthält den Hauptteil des Spaltenerzeugungsalgorithmus. Zur Modellierung und Lösung der linearen Programme werden Objekte und Funktionen der *C++ API of ILOG Concert Technology* verwendet.
buildGraph	Liest einen Graphen ein und erzeugt das Objekt, sowie die entsprechenden Knoten und Kanten. Desweiteren werden diese miteinander verknüpft.
bubbleSort	Liefert einen Permutationsvektor zurück, dessen Einträge, die nach den Dualvariablen sortierten Knoten, enthalten.
setDuals	Schreibt die Dualvariablen an die Knoten des Graphen.
compCost	Berechnet die Kosten einer Komponente nach Formel. 3.27
newSearch	Implementiert den Greedy2-Algorithmus aus Procedure. 3.
report	Auslesen und Ausgeben der aktuellen Lösung des RMP.

Tabelle 1: Funktionenübersicht

```
nodes={
        <1,1>
        <2,1>
        <3,1>
        <4,1>
        <5,2>
        <6,2>
        <7,2>
        <8,3>
        <9,3>
```

```
};
edges={
        <1,5,1.0>
        <2,6,1.0>
        <2,8,1.0>
        <4,7,1.0>
        <5,9,1.0>
        <9,3,1.0>
        };
```

Jeder Knoten, bzw. jede Kante wird in Form eines Tupels, bzw. Tripels in der entsprechenden Menge angegeben. Hierbei steht der erste Eintrag eines Knotens für seine Nummer im Graphen, der zweite Eintrag enthält sie Sequenz, in der sich der Knoten befindet. Bei den Kanten geben die ersten beiden Einträge die Nummer des Anfangs-und Endknotens an, der dritte Eintrag enthält das Kantengewicht. Das Beispiel stellt den Graph aus Abbildung 3.3, mit Kantengewichten 1.0 für alle Kanten, dar.

7 Analyse der Heuristik

7.1 Allgemeines

Man betrachte nochmal die Formulierung von (BP3) auf welche der Spaltenerzeugungsalgorithmus angewandt wurde:

$$\text{(BP3)} \quad \min \ (\delta(K_1), \delta(K_2), \dots)x \tag{7.1}$$

$$\text{s.t.} \ \ Ax = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{7.2}$$

$$x_i \in \{0, 1\}. \tag{7.3}$$

Dieses lineare Programm ist insbesondere binär. D.h. aufgrund der Gleichung 7.3 hat der Vektor x nur Einträge in $\{0, 1\}$. Das allgemeine Masterproblem für die Spaltenerzeugung ist nicht in binärer Form. D.h. die Methode der Spaltenerzeugung wurde für lineare Programme in standardform hergeleitet und entwickelt. Durch das Lösen des in Kapitel 5.2 hergeleiteten Subproblems wird nicht zwangsläufig die Lösung von (BP3) verbessert, sondern vielmehr die der Relaxation. Die Relaxation von (BP3) hat die folgenden Form.

$$\text{(BP3-Relax)} \quad \min \ (\delta(K_1), \delta(K_2), \dots)x \tag{7.4}$$

$$\text{s.t.} \ \ Ax = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \tag{7.5}$$

$$x_i \geq 0. \tag{7.6}$$

Ein Solver erstellt mithilfe eines Brach-and-Bound-Verfahrens oder eines Schnittebenen-Verfahrens eine Lösung des binären Programms (s. z.B. [4]). Diese Lösung hat im allgemeinen einen schlechteren Zielfunktionswert als die Lösung der Relaxation. Damit kann die Lösung der Relaxation als untere Schranke für die Lösung der binären Variante aufgefasst werden. Durch das Hinzufügen von Spalten, die die Relaxation verbessern, kann somit letztendlich auch die Lösung des binären Programms (BP3) verbessert werden. Eine Garantie für diese Verbesserung gibt es nicht.

7.2 Analyse von Greedy 2

In diesem Kapitel soll die gefundene Heuristik für das Subproblem näher analysiert werden. Hierzu wird beispielhaft untersucht, was für Komponenten der Greedy 2 Algorithmus findet und ob diese immer optimal in Bezug auf die multiple Sequenzalinierung sind. In Kapitel 5.3 wurden, zur Herleitung des Algorithmus, die reduzierten Kosten des Subproblems betrachtet, welche in die Kosten der Komponente $\delta(K)$ und die Summe der Dualvariablen aufgeteilt wurden. Durch die Minimierung der Kosten und die Maximierung der Dualvariablen wurde dann geschlossen, dass es sinnvoll ist, möglichst große Komponenten zu suchen und zu übernehmen. Folgende Abbildung zeigt, dass diese Annahme nicht immer richtig ist. Die Abbildung 8 zeigt einen Graphen, der aus einer nicht zulässigen Kompo-

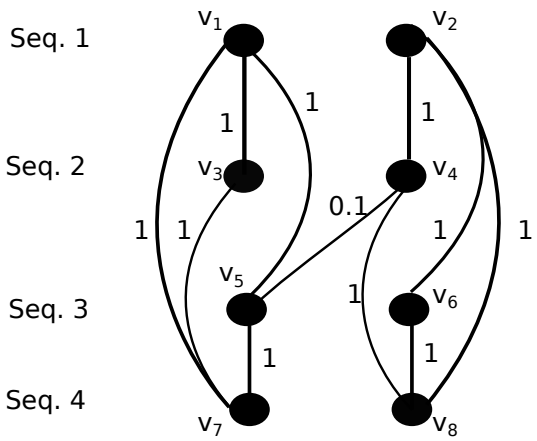

Abbildung 8: Beispielgraph

nente besteht. Alle Kantengewichte betragen 1, bis auf das Gewicht der Kante $\{v_5, v_4\}$, welches $0,1$ beträgt. Angenommen, man sucht in diesem Graph nun eine möglichst große, zulässige Komponente, so könnte man die Knoten der linken Spalte und zusätzlich v_4 wählen. Die Kosten dieser Komponente $K = \{v_1, v_3, v_4, v_5, v_7\}$ belaufen sich auf $\delta(K) = 2$. Im Hinblick auf die multiple Sequenzalinierung ist es wahrscheinlich sinnvoller, diesen Graphen in 2 Komponenten an der Kante $\{v_5, v_4\}$ zu teilen. Die Kosten dieser Komponenten $K_1 = \{v_1, v_3, v_5, v_7\}$ und $K_2 = \{v_2, v_4, v_6, v_8\}$ wären dann $\delta(K_1) = \delta(K_2) = 0,1$. Da die gefundene Heuristik auf der oben beschriebenen Annahme beruht, findet sie teilweise nicht sinnvolle Komponenten. Es soll im Folgenden untersucht werden, was der Greedy 2 Algorithmus auf dem oben dargestellten Graphen macht. Sei $v = v_7$ der Startknoten. Im ersten Schritt wird $K = Adj(v) \cup \{v\} = \{v_3, v_5, v_1, v_7\}$ und $S = \{Seq(v_v) : v \in K\} = \{1, 2, 4\}$ gesetzt. Durch die Verbesserung wird nun für alle Knoten in K überprüft, ob sie adjazent

zu Knoten sind, die sich in Sequenzen befinden, die noch nicht in S enthalten sind. Falls ja, werden diese Knoten zu K hinzugefügt. Die Knoten in K werden, sowohl beim Pseudocode, als auch bei der Implementierung, der Reihenfolge nach überprüft. Es wird nun also zuerst v_3 untersucht. Es gilt: $Adj(v_3) = \{v_1, v_7\}$ und $Seq(v_i) \in S$, für $i = 1, 7$. Das heißt, es wird kein neuer Knoten hinzugefügt. Für v_5 gilt: $Adj(v_5) = \{v_7, v_1, v_4\}$ und $Seq(v_4) = 4 \notin S$, dass heißt, durch das Hinzufügen von v_4 wird die Gültigkeit der Komponente nicht verletzt und Greedy 2 setzt $K = K \cup \{v_4\}$. Bei der Überprüfung von v_7 wird offensichtlich kein neuer Knoten gefunden. In der nächsten Iteration werden zusätzlich noch alle zu v_4 adjazenten Knoten untersucht. Da v_4 nur zu Knoten aus den Sequenzen 1,3 und 4 adjazent ist, wird auch hier kein neuer Knoten hinzugefügt. Falls die reduzierten Kosten dieser Komponente kleiner als Null sind, wird sie dem Spaltenerzeugungs-Algorithmus übergeben, falls nicht, wird ein neuer Startknoten v' gewählt, der noch nicht überprüft wurde. In dem obigen Graphen wird für die Knoten $\{v_1, v_3, v_5, v_7\}$ immer die gleiche Komponente gefunden. Analog wird für Startknoten aus $K_2 = \{v_2, v_4, v_6, v_8\}$ immer die Komponente $K_2 \cup \{v_5\}$ gefunden. Wie sich oben schon andeutet, kann die endgültig gefundene Komponente von der Reihenfolge der Knoten in der Menge K abhängen. Als Beispiel dient hier Abbildung 9. Für den Startknoten $v = v_9$ setzt Greedy 2 die Komponente im ersten Schritt auf

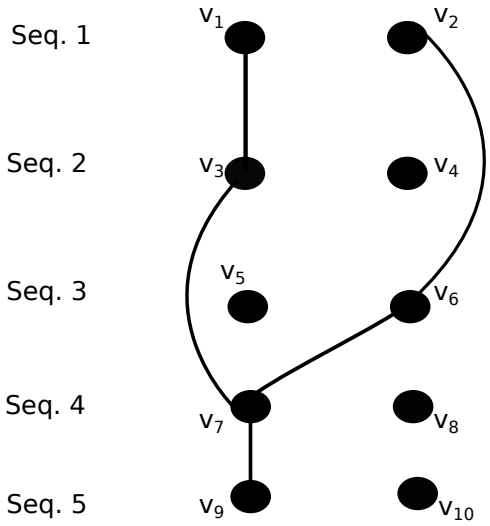

Abbildung 9: Beispielgraph

$K = \{v_9, v_7\}$. Für die Menge $Adj(v_7)$ gilt $Adj(v_7) = \{v_6, v_3\}$. Da die Sequenzen von v_6 und v_3 noch nicht besetzt sind, werden beide Knoten zu K hinzugefügt. Man erhält also $K = \{v_9, v_7, v_6, v_3\}$ oder $K' = \{v_9, v_7, v_3, v_6\}$. Für die Komponente K erhält man nun als endgültige Komponente $K_1 = K \cup \{v_1\}$ und für K' die Komponente $K_2 = K' \cup \{v_2\}$.

Dies ist nicht weiter problematisch, denn angenommen die Komponente K_1 wird gefunden und dem Spaltenerzeugungsalgorithmus übergeben, dann gilt für jede darauffolgende Iteration des Algorithmus, dass die reduzierten Kosten $\tilde{c}(K_2) = 0$ sind. D.h. Greedy 2 wird weitere Startknoten durchprobieren, also insbesondere irgendwann Knoten aus K_2 und somit diese Komponente finden. Man stellt also fest, dass das Subproblem von dem Greedy-Algorithmus nicht zwangsläufig optimal gelöst wird. Insbesondere heißt das, dass noch weitere Komponenten existieren, für die gilt

$$\min\{\delta(K) - \sum_{i \in \tilde{K}} \pi_i : K \in \mathbb{A}\} < 0,$$

obwohl Greedy keine solchen mehr findet. Da der Spaltenerzeugungsalgorithmus abbricht, sobald Greedy keine Komponente liefert, ist die gefundene Lösung nicht unbedingt optimal. In dem folgenden Kapitel werden die Ergebnisse von unterschiedlichen Datensätzen miteinander verglichen.

8 Auswertung

8.1 Systemkonfiguration

Prozessor	Intel Core i5-2540M @ 2.50 GHz
Arbeitsspeicher	4 GB
Betriebssystem	(L)Ubuntu 12.04
Kernel	Version 3.2.0-35
Compiler	GNU g++ 4.6.3
IBM ILOG CPLEX	Version 12.4.0.0

Tabelle 2: Systemkonfiguration

8.2 Datensätze

Die getesteten Datensätze stammen aus der von Thomson et al. (2005) entwickelten Datenbank *BAliBASE 3.0*, siehe [13]. Diese wurden mit dem Programm *DIALIGN* paarweise optimal aliniert. Es wurden 4 Datensätze unterschiedlicher Länge, d.h. verschiedener Knoten-und Kantenanzahl getestet. Die folgende Tabelle gibt Aufschluss über dem Umfang der Daten.

Name	#Sequenzen	#Knoten	#Kanten
D1	3	9	6
D2	4	318	407
D3	4	1198	1261
D4	11	1481	3327

Tabelle 3: Übersicht der Testdaten

8.3 Ergebnisse von (BP1)

Das binäre Programm (BP1) wurde mithilfe der Skriptsprache *OPL* aus dem *IBM ILOG CPLEX Optimization Studio* implementiert und getestet. Trotz der „intelligenten“ Im-

plementierung, d.h. trotz Vorabreduktion einiger Nebenbedingungen war es nicht mehr möglich den Datensatz D2 zu lösen. Das verwendete Betriebssystem brach die Berechnung nach Verwendung des gesamten zur Verfügung gestellten Speichers ab. Die folgende Tabelle gibt Aufschluss über die Ergebnisse.

Daten	Laufzeit(s)	#gelöschter Kanten	Zielfunktionswert
D1	0	1	1
D2	/	/	/
D3	/	/	/
D4	/	/	/

Tabelle 4: Ergebnis (BP1)

8.4 Ergebnisse von (BP2)

Wie auch (BP1) wurde (BP2) mithilfe der Skriptsprache *OPL* implementiert. Für die Anzahl der zur Verfügung stehenden Komponenten m in dieser Formulierung wurden verschiedene Werte der Form

$$m = M * \#(\text{Knoten in der Längsten Sequenz}) = M * \left(\max_{i=1,\dots,N} |V_i| \right)$$

getestet. Hierbei bezeichnet N die Anzahl der Sequenzen und V_i die i-te Sequenz. Die Ergebnisse sind in der folgenden Tabelle angegeben. Bei Datensatz D4 für $M = 1,3$ und $M = 1,5$ wurde die Berechnung vom Betriebssystem abgebrochen wegen unzureichender Speicherkapazität. Für $M < 1$ konnte keiner der Datensätze gelöst werden.

Daten	M	Laufzeit(s)	#gelöschter Kanten	Zielfunktionswert
D1	1	0,04	1	1
D2	1	0,44	66	55,1
D3	1	698,71	31	17,54
D4	1	>8h	/	/
D1	1,1	0,05	1	1
D2	1,1	0,47	66	55,1
D3	1,1	12,26	22	16,67
D4	1,1	>8h	/	/
D1	1,2	0,0	1	1
D2	1,2	0,47	66	55,1
D3	1,2	35,29	22	16,67
D4	1,2	>8h	/	/
D1	1,3	0,1	1	1
D2	1,3	0,62	66	55,1
D3	1,3	134,47	22	16,67
D4	1,3	/	/	/
D1	1,5	0,0	1	1
D2	1,5	0,57	66	55,1
D3	1,5	536,37	22	16,67
D4	1,5	/	/	/

Tabelle 5: Ergebnis (BP2)

8.5 Ergebnisse von (BP3)

Die Implementierung des Spaltenerzeugungsalgorithmus wurde bereits oben beschrieben. Die Ergebnisse der vier Datensätze sind in Tabelle 5 dargestellt. In der Tabelle wurde der halbe Zielfunktionswert eingetragen, welcher genau die Summe der Gewichte aller entfernter Kanten ist.

Daten	Laufzeit(s)	# gelöschter Kanten	Zielfunktionswert
D1	0	1	1
D2	0,22	54	114,3
D3	4	24	38,79
D4	7	1000	2743,77

Tabelle 6: Ergebnis (BP3) mit Spaltenerzeugung

9 Gesamtauswertung und Diskussion

Wie erwartet, ist die Formulierung (BP1) auf den heute aktuellen Computern nicht effizient lösbar. Schon bei dem noch kleinen Datensatz D2 reichen 4 GB Arbeitsspeicher und 4 GB Swap nicht aus.
Das Problem wird für kleine Datensätze von der Formulierung (BP2) in akzeptabler Zeit gelöst. Bemerkenswert ist dabei der Ausreißer in der Laufzeit bei D3 und $M = 1$. Es fällt auf, dass sich die Anzahl der gelöschten Kanten, sowie der Zielfunktionswert ab $M = 1,2$ und $M = 1,3$ nicht weiter verbessern. Aufgrund dieser Ergebnisse kann man davon ausgehen, für $M = 1,3$ recht nahe an der Optimallösung zu sein. Das Gleiche kann man auch bei der Berechnung mit anderen Datensätzen beobachten.
Um die Güte der gefundenen Heuristik, d.h. die Güte von Greedy 2, ermitteln zu können, müsste man die Relaxation von (BP3), einmal mit vorher berechneten Komponenten und einmal mit dem Greedy-Verfahren lösen. Man könnte dann für unterschiedliche Datensätze die Lösungen miteinander vergleichen und somit die Güte von Greedy 2 abschätzen.
Da die Berechnung der Komponenten eines Alinierungsgraphen, auch für Testzwecke, sehr viel Zeit und Speicherplatz benötigt, sollte versucht werden, die Güte der Heuristik anders zu bestimmen. Wenn man, zum einen, davon ausgeht, dass die mit (BP2) und $M = 1,3$ gefundene Lösung „nahe“ am Optimum liegt. Und zum anderen, dass durch die Verbesserung der Lösung von (BP3-relax) durch Greedy 2 auch die Lösung des binären Programms (BP3) verbessert wird, kann man die Ergebnisse der beiden Programme miteinander vergleichen und Vermutungen über die Güte aufstellen.
Im Folgenden sollen die oben dargestellten Lösungen miteinander verglichen werden. Man sieht an den Datensätzen D2 und D3, dass der Zielfunktionswert der Heuristik um einen Faktor $2-2,5$ schlechter ist, als der von (BP2). Zum Beispiel löscht (BP2) beim Datensatz D3 22 Kanten mit einem Gesamtgewicht von 16,67. Die Implementierung von (BP3) mit Greedy 2 löscht hier 24 Kanten mit einem Gesamtgewicht von 38,79.
Positiv fällt die Laufzeit der Heuristik auf. Sie ist, vorallem bei D3, deutlich kleiner (4 Sekunden) als die von (BP2) (134 Sekunden).
Die gefundene Heuristik findet auch für große Datensätze, wie z.B. D4, schnell eine Lösung. Da es hierfür keine Vergleichsergebnisse gibt, kann man nur abschätzen, wie nahe diese Lösung am Optimum ist. Da das Verhältnis von Kanten zu Knoten in D4 deutlich höher ist als bei den anderen Datensätzen, kann man davon ausgehen, dass viele Kanten gelöscht werden mussten. Dennoch ist der gefundene Zielfunktionswert und die Anzahl der gelöschten Kanten recht hoch.

Man kann also allgemein festhalten, dass es sinnvoll ist, in einem ersten Schritt mit einem Greedy-Verfahren nach Komponenten zu suchen, um mögliche untere Schranken zu finden. Diese gefundenen Schranken könnte man z.B. für einen Branch-and-Bound-Algorithmus verwenden.

Literatur

[1] J. D. Kececiouglu, H.-P. Lenhof, Kurt Mehlhorn, Petra Mutzel, Knut Reinert, and Martin Vingron. A Polyhedral Approach to Sequence Alignment Problems. Discrete Applied Mathematics, 104:143–186, 2000

[2] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48:443–453, 1970

[3] J. D. Kececiouglu. The maximum weight trace problem in multiple sequence alignment. In Proceedings of the 4th Symposium on Combinatorial Pattern Matching, number 684 in LNCS, Seite 106–119. Springer, 1993

[4] B. Korte, J. Vygen. Combinatorial Optimization, Theory and Algorithms. Third Edition, Springer 2005

[5] John M. Smith. Evolutionary Genetics, Second Edition, Oxford University Press, 1998

[6] Robert J. Vanderbei. Linear Programming, Foundations and Extensions. Third Edition, Springer 2008

[7] J. Desrosiers, M. E. Lübbecke. A Primer in Columngeneration. In Column Generation, G. Desaulniers, J. Desrosiers, and M.M. Solomon (Eds.), Springer, 2005

[8] Watson, J.D. und Crick F.H. (1953): Molecular structure of nucleic acids. A structure for deoxyribose nucleic acid. In: Nature. Bd. 171, Nr. 4356, S. 737–738

[9] Smith, Temple F. und Waterman, Michael S. (1981). Identification of Common Molecular Subsequences. Journal of Molecular Biology 147: 195–197

[10] Lander, E.S., Langridge, R., and Saccocio, D.M. 1991. Mapping and interpreting biological information. Commun. ACM 34,33-39

[11] Wang, L., Jiang, T. (1994). On the Complexity of Multiple Sequence Alignment. Journal of Computational Biology, Vol 1, Nr. 4, S. 337-348.

[12] Corel, E., Pitschi, F., Morgenstern P., (2010). A min-cut algorithm for the consistency problem in multi sequence alignment. Oxford Journals, Bioinformatics, Vol. 26, Nr. 8, S. 1015-1021

[13] Thompson,J.D. et al. (2005) BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark. Proteins Struct. Funct. Bioinform., 61, S.127–136

Erklärung nach §13(6) der Prüfungsordnung für den Bachelor-Studiengang Mathematik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 31. Januar 2013

Erik Pohl