DIPLOMARBEIT

Ein metaheuristischer Lösungsnsatz für das Multiple Sequence Alignment Problem

ausgeführt am Institut für
Computergraphik und Algorithmen
der
Technischen Universität Wien

unter der Anleitung von
Ass.Prof. Univ.Doz. Dr. Günther Raidl
und
Univ.Ass. Dr. Gabriele Kodydek
durch

Janosch Fauster

Kölblgasse 13/3, 1030 Wien, Österreich
Zwölfmalgreinerstr. 9/3 39100 Bozen, Italien

24. Februar 2004

Abstract

Wir präsentieren in dieser Arbeit heuristische Algorithmen zur Lösung einer NP-schwierigen Aufgabenstellung der Bioinformatik, dem Multiple Sequence Alignment. Das Problem wird hier in ein Problem auf einem Graphen, dem sog. Maximum Weight Trace Problem, umformuliert. Dazu werden mit Hilfe des progressiven Verfahren ClustalW alle optimale paarweisen Alignments berechnet, deren Ergebnisse in einem Alignmentgraph verwaltet werden. Darauf führen wir eine sog. Alignmentgraph Extension durch, welche zusätzliche (globale) Informationen in den Alignmentgraph einbringen soll. Außerdem wurden Methoden entwickelt, welche jeder Kante im Alignmentgraphen ein möglichst sinnvolles Gewicht vergeben. Die Aufgabe ist es nun, einen Lösungstrace zu finden, der Kanten des Alignemntgraphen mit möglichsta hohem Gesamtgewicht beinhaltet und ein gültiges multiples Alignment beschreibt. Hierzu wurden zwei schnelle Greedy-Heuristiken entwickelt. So gefundene Traces werden weiter durch verschiedene lokale Verbesserungsverfahren, wie das Verschieben von Lücken oder von zusammenhängenden Teilstücken, und einer Tabu-Suche optimiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Diese Arbeit stellt Algorithmen zur näherungsweisen Lösung eines schwierigen, bedeutenden Problems der Bioinformatik, dem Multiple Sequence Alignment, vor. Ziel des Multiple Sequence Alignment Problem ist es, eine Reihe von Aminosäuresequenzen (Proteine) oder Basensequenzen (DNA, RNA), die in Form von Zeichenketten aus einem bestimmten Alphabet dargestellt werden, so in Relation zu einander zu bringen (sie zu *alinieren*), dass eine gegebene Kostenfunktion minimiert wird.

Mit Hilfe von bereits bestehenden Methoden, Erweiterungen von diesen, sowie neuen Elementen wurde ein Programm entwickelt, welches imstande ist, dieses NP-schwierige Problem in vielen Fällen optimal zu lösen. Zur Erstellung einer Ausgangslösung wird das Multiple Sequence Alignment-Problem in ein graphentheoretisches Modell umgewandelt und ein Greedy-Algorithmus darauf angewandt, welcher eine erste Lösung erstellt. Ein Tabu-Search Algorithmus sowie mehrere lokale Verbesserungsheuristiken optimieren diese Ausgangslösung, sodass schlussendlich sehr gute Alignments der Sequenzen erzielt werden können.

Dieses Kapitel führt in den nächsten Abschnitten in das Problem ein. Kapitel 2 beschreibt bereits existierende Arbeiten und Lösungsansätze, darauf folgt in Kapitel 3 die Vorstellung des Alignmentgraphen, einer zentralen Datenstruktur zur Lösung dieser Problemstellung. Kapitel 4 zeigt, wie man mit Hilfe von zwei Greedy-Algorithmen eine Ausgangslösung für die weiteren Heuristiken aus Kapitel 5 erzeugen kann. Kapitel 6 geht schließlich detailiert auf den entwickelten Tabu-Search-Ansatz ein, Kapitel 7 beschreibt die Implementierung aller Algorithmen genauer und Kapitel 8 analysiert anschließend die Qualität der erhaltenen Ergebnisse. Schlussendlich vervollständigen wir in Kapitel 9 mit abschließenden Bemerkungen diese Arbeit.

1.1 Beispiel

Das Beispiel in Abbildung 1.1 zeigt eine Problemstellung für das Multiple Sequence Alignment. Gegeben sind vier unalinierte Sequenzen, welche Fragmente eines Proteins darstellen. Die Symbole, aus welchen die Sequenzen bestehen, repräsentieren in diesem Falle Aminosäuren.

```
MPQILLLV
LRLL
MKILLL
MPPVLILV
```

Abbildung 1.1: Beispiel: Sequenzen vor Alignment

```
MPQILLLV
MLR-LL--
M-KILLL-
MPPVLILV
```

Abbildung 1.2: Beispiel: Sequenzen nach Alignment

Die Abbildung 1.2 zeigt nun ein mögliches Alignment für die Eingabesequenzen. Man sieht, dass nun alle Sequenzen die gleiche Länge haben und dass das multiple Alignment aus den ursprünglichen Sequenzen dadurch gewonnen wurde, dass Leerzeichen eingefügt worden sind, sodass ähnliche Bereiche in der gleichen Spalte untergebracht werden (um damit eine gegebene Kostenfunktion zu minimieren).

1.2 Anwendungsbereiche

Folgende Aufgaben sollen mit Hilfe des Multiple Sequence Alignment erleichtert bzw. ermöglicht werden:

- Vorhersage von Proteinstrukturen: evolutionär oder funktionell ähnliche Proteine bzw. DNA-Sequenzen haben häufig ähnliche Teilstücke in ihren Aminosäuresequenzen, die für eine gemeinsame dreidimensionale Struktur (Sekundär- und Tertiärstruktur von Proteinen bzw. DNA-Strängen) verantwortlich sind.

- Phylogenetische Analyse: evolutive Aussagen werden möglich: Erkennen gemeinsamer Vorfahren, Erstellen von Stammbäumen, Erkennen von Mutationen in Genen.

- Genomsequenzierung: gleiche Gene in der DNA von verschiedenen Menschen haben Ähnlichkeiten in der Sequenz ihrer Basen Adenin,

Cytosin, Thymin und Guanin (A, C, T, G). Ähnliches gilt auch für die verschiedenen Typen von RNA. Mit Hilfe des Multiple Sequence Alignment können diese Ähnlichkeiten herausgearbeitet werden.

1.3 Grundlagen

Definition 1.1 (Multiple Sequence Alignment, MSA) *Sei* Σ *ein endliches Alphabet ohne Lücke (gap)* $'-'$ *und* $\Sigma' = \Sigma \cup \{'-'\}$*. Seien weiter* $s_1, \ldots, s_k$ *k Sequenzen über* Σ *mit Längen* $l_1, \ldots, l_k$*. Ein (globales) multiples Alignment A von* $s_1, \ldots, s_k$ *ist eine Matrix der Dimension* $k \times l$ *mit folgenden Eigenschaften:*

- $max(l_1, \ldots, l_k) \leq l \leq \sum_{i=1}^{k} l_i$
- $A[i][j] \in \Sigma' \ \forall 1 \leq i \leq k, 1 \leq j \leq l$
- *für je zwei Symbole einer Sequenz* $s_{k,i}, s_{k,j}$ *mit* $i < j$ *gilt, dass deren relative Ordnung im Alignment erhalten bleibt. Ist ihre neue Position im MSA* $s_{k,i'}$ *und* $s_{k,j'}$*, so gilt* $i' < j'$*.*
- *keine Spalte besteht nur aus Lücken*
- *Anzahl der Sequenzen* $k \geq 2$ *(*$k = 2$ *ist ein Spezialfall: Paarweises Alignment)*

Man sucht jedoch im Normalfall nicht irgendein Alignment, sondern ein optimales Alignment.

Definition 1.2 (Optimales Alignment) *Ein optimales Alignment ist ein Alignment welches eine Kostenfunktion minimiert. Dabei wird als Kostenfunktion häufig die* Sum of Pairs*-Funktion benutzt. Diese ist wie folgt definiert:*

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

wobei:

- s_{ih} *das h-te Zeichen der i-ten Sequenz darstellt;*
- $c : \Sigma' \times \Sigma' \to \mathbb{R}$ *eine Kostenfunktion für Paare von Symbolen ist;*
- $c(-,-) = 0$ *gilt.*

Es kann gezeigt werden, dass das Multiple Sequence Alignment mit der Kostenfunktion SP ein NP-vollständiges Problem ist.

Das *paarweise* Alignment, also ein Alignment von zwei Sequenzen, ist ein (wichtiger) Spezialfall, da es mittels dynamischer Optimierung effizient

optimal gelöst werden kann. Deshalb stellen die paarweisen Alignments bei vielen Lösungsansätzen für das MSA-Problem die Ausgangsinformationen zur Verfügung. Ein *globales* Alignment aliniert zwei oder mehrere Sequenzen vollständig, ein *lokales* Alignment hingegen kann auch nur Teile von Sequenzen alinieren.

1.4 Maximum Weight Trace Formulierung des MSA

Jedes Alignment kann auch in Form eines Graphen $G = (V, E)$ dargestellt werden. Dieser ist wie folgt definiert:

Definition 1.3 (Alignmentgraph) *Ein Knoten des Alignmentgraphen repräsentiert genau ein Symbol $s_{i,h}$ für $i = 1, \ldots, k$ und $h = 1, \ldots, l_i$. Wir bezeichnen daher diese Knoten genauso wie die entsprechenden Symbole. Kanten gibt es immer nur zwischen Knoten zweier unterschiedlicher Sequenzen.*

Existiert eine Kante e_m zwischen $s_{i,h}$ und $s_{j,g}$, so bedeutet dies, dass diese beiden Symbole eventuell miteinander aliniert werden sollten, also in der Matrix A in derselben Spalte stehen sollten. Die Kanten haben ein Gewicht w_m, das ein heuristisches Maß für die Bedeutung des Alignments der beiden Symbole darstellt. Kanten die z.B. zwei gleiche Symbole verbinden, sollten ein hohes Gewicht bekommen, während weniger gut passende Kanten mit einem niedrigen Gewicht versehen werden sollten.

Definition 1.4 (Erweiterter Alignmentgraph) *Wir definieren nach [13] einen erweiterten Alignmentgraph als einen Graphen, der zusätzlich zu der oben beschriebenen Kantenmenge weitere Kanten, sog.* Arcs*, enthält. Arcs (gerichtete Kanten) existieren dabei von allen Knoten $s_{i,h}$ zu den Nachfolgeknoten $s_{i,h+1}$ für $i = 1, \ldots, k$ und $h = 1, \ldots, l_{i-1}$.*

Definition 1.5 (Trace) *Ein erweiterter Alignmentgraph enthält genau dann eine* gültige *Lösung, wenn alle im Graphen vorkommenden Kreise keine Arcs enthalten. Ein solcher Graph wird nach [12] auch als* Trace *bezeichnet.*

Abbildung 1.3 zeigt ein Beispiel für einen gültigen und einen ungültigen Trace. Ein weiteres Merkmal eines Traces ist, dass jedes MSA eine eindeutigen Trace hat, jedoch ein Trace nicht ein eindeutiges Alignment repräsentiert.

Durch diese Trace-Formulierung des MSA ergibt sich auch eine neue Zielfunktion, die optimiert werden soll:

$$\sum_{i=1}^{n} w(e_i) \rightarrow max,$$

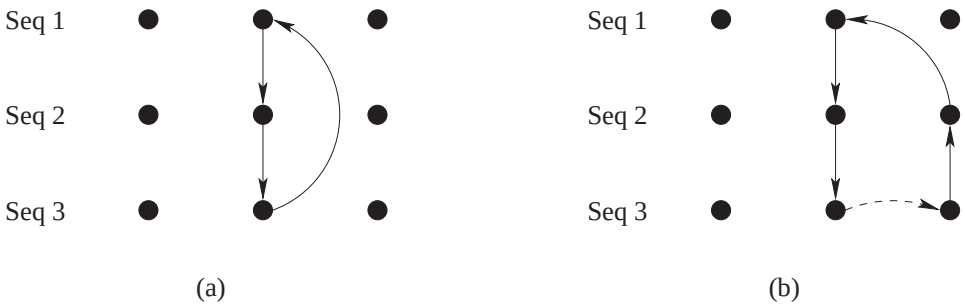

Abbildung 1.3: Ein (a) gültiger und ein (b) ungültiger Trace

wobei $w(e_i)$ das Gewicht der Kante e_i bezeichnet und $e_1, \ldots, e_m$ alle Kanten des Trace darstellen. Das heißt also, dass jener Trace mit dem größtmöglichem Gesamtgewicht aller realisierten Kanten gesucht wird.

Abbildung 1.4 zeigt eine Beispielsequenz, einen dazugehörigen Alignmentgraphen, welcher gleichzeitig einen gültigen Trace darstellt sowie das multiple Alignment, das daraus abgeleitet werden kann.

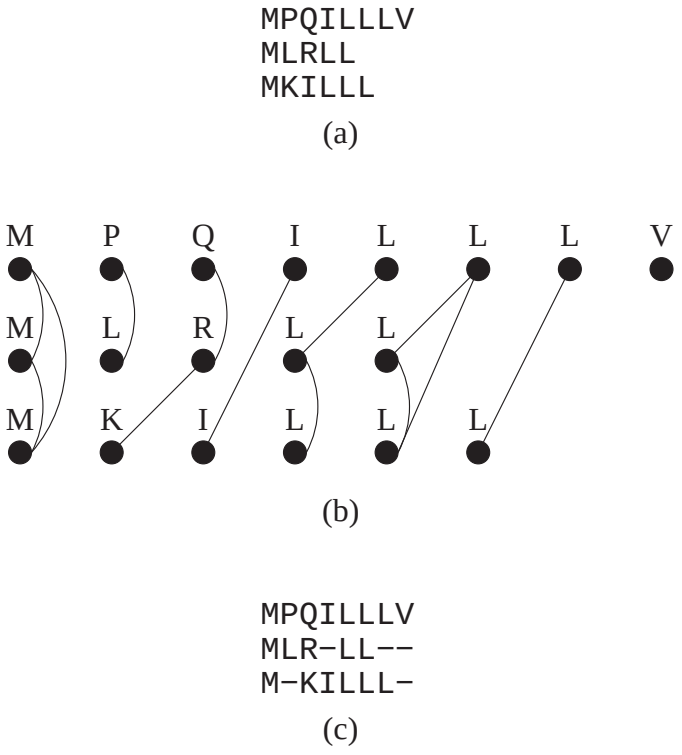

Abbildung 1.4: Alignmentgraph - Trace - Multiples Alignment
(a) Eingabesequenz, aus der ein multiples Alignment erstellt werden soll.
(b) Ein möglicher Alignmentgraph zu (a), welcher auch einen gültigen Trace darstellt.
(c) Das mulitple Alignment, welches aus dem Trace von (b) gewonnen werden kann.

Kapitel 2

Vorhandene Arbeiten

2.1 Übersicht

Die folgenden Abschnitte sollen die wichtigsten Ergebnisse einiger wissenschaftlicher Arbeiten der letzten Jahre zusammenfassen. Notredame unterteilt in Artikel [18] die Verfahren zum Lösen des MSA-Problems in vier verschiedene Kategorien:

- *Progressive Algorithmen:* Diese Klasse von Algorithmen gehört zu den einfachsten, aber effektivsten Möglichkeiten, das MSA-Problem zu lösen. In einem progressiven Ansatz werden alle Sequenzen einzeln zum multiplen Alignment hinzugefügt, wobei die Bearbeitungsreihenfolge der Sequenzen zuvor durch Berechnung unterschiedlicher Kriterien, wie die Ähnlichkeit von je zwei Sequenzen festgelegt wird. Ein prominenter Vertreter dieser Klasse ist ClustalW [29].

- *Exakte Algorithmen:* Die Klasse der exakten Algorithmen berechnet im Gegensatz zu den anderen Ansätzen immer optimale multiple Alignments. Da der Zeitaufwand jedoch immer exponentiell ist, ist die Größe der Eingabeinstanzen stark limitiert. Der Branch-and-Bound-Ansatz aus [12] gehört zu dieser Klasse von Algorithmen.

- *Iterative Algorithmen:* Iterative Algorithmen versuchen eine suboptimale Ausgangslösung durch wiederholtes Anwenden von Verbesserungsheuristiken so zu verbessern, um ein (fast) optimales Alignment zu erzielen. Diese Klasse kann noch in zwei Unterbereiche eingeteilt werden: den *stochastischen iterativen Algorithmen* und den *nicht stochastisch iterativen Algorithmen*. Zur ersten Unterteilung zählen Verfahren, welche *Simulated Annealing*, *genetische Algorithmen* oder Ähnliches benutzen, zur zweiten Klasse gehören vor allem Algorithmen die auf dynamischer Programmierung basieren. Zu den iterativen Algorithmen zählen SAGA [19], aber auch unser Ansatz.

Name	Klasse	Referenz
MSA	Exakt	[14]
DCA	Exakt	[26]
OMA	Iterativer DCA	[22]
ClustalW, ClustalX	Progressive	[29]
MultAlin	Konsistenz-basierend	[4]
DiAlign	Konsistenz-basierend	[16]
ComAlign	Konsistenz-basierend	[3]
T-Coffee	Konsistenz-basierend/Progressiv	[20]
Praline	Iterativ/Progressiv	[8]
IterAlign	Iterativ	[2]
Prrp	Iterativ/Stochastisch	[1]
SAM	Iterativ/Stochastisch	[10]
HMMER	Iterativ/Stochastisch	[5]
SAGA	Iterativ/Stochastisch	[19]
GA	Iterativ/Stochastisch	[30]

Tabelle 2.1: Überblick über existierende Verfahren (übernommen aus [18])

- *Konsistenz-basierende Algorithmen:* Algorithmen dieser Klasse versuchen unabhängige Beobachtungen so zu verarbeiten, dass sie möglichst konsistent werden. Bezogen auf das MSA werden paarweise Alignments gebildet (voneinander unabhängige Daten/Beobachtungen) und daraus eine Lösung erzeugt, welche die paarweisen Alignments möglichst gut abbildet, wobei konsistente Subalignments bevorzugt behandelt werden. Ein Beispiel dafür ist T-Coffee [20], das jedoch eine Mischform von progressiven Algorithmen und Konsistenz-basierenden Algorithmen darstellt.

Tabelle 2.1 gibt einen Überblick über viele wichtige Verfahren und deren Einteilung in den oben beschriebenen Klassen.

2.2 Erstellen der paarweisen Alignments

Das alinieren von zwei Sequenzen kann optimal mittels dynamischer Programmierung gelöst werden. Die zwei (historisch) wichtigsten Ansätze werden in den folgenden beiden Abschnitten kurz erläutert.

2.2.1 Needleman-Wunsch-Algorithmus

Der Needleman-Wunsch-Algorithmus zum berechnen eines paarweisen Alignments wurde schon 1970 in [17] vorgestellt. Das Ziel dieses Algorithmus

ist es, einen sogenannten *similarity score* durch das erstellen von *globalen Alignments* zu maximieren. Die drei Hauptphasen des Algorithmus sind die folgenden:

1. Zuerst werden alle möglichen Paare zwischen den beiden Sequenzen in einer Matrix dargestellt. Darauf wird jedem Paar ein Ähnlichkeitskoeffizient zugeordnet der das Zueinanderpassen der beiden Symbole beschreibt. Dieser Koeffizient kann auf einem sehr einfachen Prinzip beruhen oder er kann auch biochemische und evolutive Informationen enthalten wie zum Beispiel die Vertauschungswahrscheinlichkeit verschiedener Aminosäuren oder Basen.

2. Nun wird jeder Pfad durch die Matrix bewertet. Wir wollen hier nicht genauer in die Details eingehen. Die Idee dahinter ist jedoch für jede Zelle in der Matrix das beste Alignment zu bestimmen, welches genau an diesem Punkt endet.

3. Schließlich muss noch das beste Alignment bestimmt werden. Dies ist genau jenes, welches den höchsten Gesamtscore erreicht.

2.2.2 Smith-Waterman-Algorithmus

Dieser Algorithmus aus [25] basiert auf jenem von Needleman und Wunsch. Im Gegensatz zum Algorithmus von Needleman und Wunsch berechnen Smith und Waterman jedoch *lokale Alignments*. Dazu werden zwei Sequenzen nicht als Ganzes aliniert, sondern Subsequenzen aller möglichen Längen von diesen. Damit dieser Algorithmus effizient arbeiten kann müssen zusätzlich zu den Ähnlichkeitskoeffizienten sogenannte *gap penalties* berechnet werden. Dies sind Strafpunkte, welche bei dem Auftreten von Lücken vom Gesamtscore subtrahiert werden.

2.3 ClustalW

ClustalW ist wohl der bekannteste und am häufigsten benutzte Algorithmus um das MSA-Problem zu lösen. Er basiert auf einer progressiven Alignment-Strategie [29]. Wir wollen hier die Vorgehensweise von diesem Algorithmus kurz erläutern:

Zuerst werden mit Hilfe der dynamischen Programmierung alle paarweisen optimalen Alignments berechnet. Dazu wird jede Sequenz mit jeder anderen aliniert. Es müssen also bei n Sequenzen insgesamt

$$\binom{n}{2} = \frac{n * (n-1)}{2}$$

paarweise Alignments berechnet werden. Um eine Kante zwischen zwei Symbolen zu bewerten, wurden von Experten Matrizen erstellt, welche die chemische Ähnlichkeit für alle Kombinationen zweier Symbole angeben.

Zu ihnen zählen folgende Matrizenserien:

- PAM
- BLOSUM
- GONNET

Je nachdem, ob es sich bei den Eingabesequenzen um Proteine oder DNA handelt, und abhängig von den Einstellungen der Benutzer wird eine Matrizenserie vom Programm ausgewählt. Von jeder Serie existieren mehrere Matrizen, von welchen schließlich in Abhängigkeit von der Ähnlichkeit des betrachteten Sequenzenpaareseine eine bestimmte für die Bewertung ausgewählt wird. ClustalW berechnet dazu einen heuristischen Ähnlichkeitskoeffizienten der beiden Sequenzen. Wenn die zu analysierenden Sequenzen sehr ähnlich sind, wird eine Matrix gewählt, welche Identitäten, also zwei gleiche Symbole, sehr hoch bewertet, unterschiedlichen Symbolen jedoch prinzipiell eine schlechte Bewertung gibt. Ist zwischen den Eingabesequenzen jedoch eine große evolutionäre Distanz vorhanden, ist dieser Ansatz nicht erfolgversprechend und deshalb wird eine Matrix ausgewählt, welche auch unterschiedlichen, aber strukturell ähnlichen Aminosäuren eine höhere Bewertung gibt.

Nachdem das gerade betrachtete Sequenzenpaar gemäß des gewählten Bewertungsschemas (mathematisch) optimal miteinander aliniert wurde, wird die ermittelte Ähnlichkeit in einer Distanzmatrix eingetragen.

Auf Basis dieser Distanzmatrix wird daraufhin unter Verwendung der *neighbor-joining*-Methode [23] ein *guide tree (phylogenetischer Baum)* aufgebaut, der das progressive Alignment leitet. Der Baum bestimmt nun, welche beiden Sequenzen als nächstes aliniert werden sollen. Sind beide Sequenzen noch nicht mit anderen Sequenzen aliniert, so wird einfach das paarweise Alignment in die Lösung übernommen. Ist eine Sequenz bereits vorher mit einem anderen Subalignment aliniert worden, so wird die zweite Sequenz bestmöglich mit dem Subalignment aliniert, wobei vorher eingefügte Lücken nicht mehr entfernt werden können. Der Baum hat dabei die Aufgabe, die ähnlichsten Sequenzen zuerst an den Algorithmus zu übergeben und erst im späteren Verlauf ungünstige Paarungen zu berücksichtigen. Der *guide tree* retourniert so lange neu auszurichtende Sequenzen, bis alle Sequenzen in *einem* Alignment aliniert worden sind.

Die Praxis zeigt, dass durch diesen Greedy-artigen Ansatz gute Ergebnisse erzielt werden können. Es darf jedoch nicht verschwiegen werden, dass mit diesem Algorithmus häufig nur lokale Optimas erreicht werden, da einmal festgelegte Lücken nicht mehr verändert werden können und auch kein Versuch unternommen wird, aus diesen herauszukommen.

2.4 T-Coffee

Auch T-Coffee von Notredame et al. [20] basiert auf einer progressiven Alignment Strategie, wobei die Autoren jedoch versuchen die Nachteile, welche durch den Greedy-Charakter einer progressiven Heuristik entstehen, zu beseitigen. Benutzt man nämlich eine reine Greedy-Strategie wie z.B. in ClustalW, so können Entscheidungen, welche beim Alinieren der ersten Sequenzen gemacht worden sind, nicht mehr revidiert werden, falls sich herausstellt, dass diese ungünstig waren. Dieses Problem kann dadurch überwunden werden, indem man bereits von Beginn an alle Sequenzen gleichzeitig aliniert. Dies wurde u.a. in MSA [14] und DCA [26] verwirklicht. Es zeigt sich jedoch, dass solche Verfahren äußerst speicher- und rechenintensiv sind.

Alle drei eben erwähnten Verfahren benutzen Methoden um *globale* Alignments zu erstellen, d.h. sie versuchen immer Sequenzen vollständig miteinander zu alinieren. Ein anderer Ansatz wäre jedoch, *lokale* Ähnlichkeiten in Sequenzen herauszuarbeiten. T-Coffee versucht sowohl globale Alignment-Informationen als auch lokale Ähnlichkeiten zu berücksichtigen. Die globalen Informationen übernimmt es von den paarweisen ClustalW-Alignments, lokale Daten werden mit Hilfe eines lokalen Alignment-Algorithmus ermittelt.

Ein wichtiger Algorithmus für lokale Alignments zweier Sequenzen wurde von Smith und Waterman [24, 21] entwickelt. Das Programm Lalign[9] ist eine Variante dieses Algorithmus und wird von T-Coffee verwendet um auch kurze, lokal gut alinierte Stücke zu erkennen.

Schließlich wird mittles einer weiteren Heuristik (*library extension*) versucht, jedes paarweise Alignment mit allen anderen Sequenzen zu vergleichen, um dadurch noch mehr Informationen zu erhalten, die den Alignment-Prozess steuern können. Nachdem sowohl globale als auch lokale Alignments gebildet wurden, wird ein phylogenetischer Baum aufgebaut und ein ClustalW-ähnlicher progressiver Algorithmus gestartet. Dieser entscheidet widerum aufgrund der Informationen im Baum, welche Sequenzen als nächstes in die Lösung eingebaut werden sollen.

2.5 Maximum Weight Trace: Ein Branch and Bound Ansatz

J. D. Kececioglu beschreibt in [12] einen Branch and Bound Algorithmus zur Lösung des Multiple Sequence Alignment Problems. Als Ausgangspunkt dienen optimale paarweise Alignments, welche sich auf effektive Weise mittels dynamischer Programmierung berechnen lassen. Daraus lässt sich nun ein Alignmentgraph wie in Kapitel 1.3 beschrieben aufbauen. Nun kann mit Hilfe der Informationen, welche im Alignmentgraph enthalten sind, das so genannte *Maximum Weight Trace* Problem gelöst werden, aus welchem sich schließlich eine *optimale* Lösung für das Multiple Sequence Alignment ablei-

ten lässt. Ziel des Maximum Weight Trace Problem ist es, einen Subgraphen im Alignment zu finden, in welchem keine Kreise vorkommen, die Arcs enthalten. Dabei soll das Gesamtgewicht der Kanten im Trace maximal sein.

Natürlich ist auch das Maximum Weight Trace Problem ein NP-vollständiges Problem und die Laufzeit eines optimalen Algorithmus unter Umständen exponentiell, aber durch das Branch and Bound Verfahren sollte die Laufzeit in den meisten Fällen stark begrenzt werden. Folgende zwei Schritte sind im Branch and Bound Ansatz wichtig und werden deshalb kurz behandelt:

Branch: Dieser Schritt generiert aus einer bestehenden Teillösung, durch Hinzunahme weiterer Kanten im Trace neue Teillösungen, sodass die Qualität der Lösungen verbessert wird. Dazu verwendet Kececioglu graphentheoretische Methoden, die bestimmen über welche Kanten das Branching stattfindet.

Bound: mittels zweier Heuristiken wurden eine untere und obere Schranke bestimmt, die dazu benutzt werden, um Teilprobleme, welche diese Schranken nicht erfüllen und somit nicht zur Verbesserung der Lösung beitragen können, frühzeitig zu verwerfen.

2.6 Integer Lineare Programmierung & Branch and Cut

Diesen von Kececiouglu et al. [13] entwickelter Algorithmus benutzt das Konzept der *Ganzzahligen Linearen Programmierung (ILP)* sowie einen *Branch and Cut*-Algorithmus um ein (optimales) Alignment zu berechnen. Kleine Probleminstanzen können dadurch gelöst werden, indem man Ungleichungen aufstellt, die das Problem definieren und darauf einen ILP-Solver anwendet. Bei großen Probleminstanzen ist dieser Weg zu aufwändig, und deshalb wird für solche Sequenzen ein Branch-and-Cut-Algorithmus angewandt. Um den Lösungsraum durch sog. *cutting planes* einzuschränken, wird ein lineares Programm ähnlich dem vorigen ILP erstellt, wobei die Bedingung der Ganzzahligkeit fallen gelassen wird. Dadurch lässt sich auf effiziente Weise die Suche nach der optimalen Lösung beschleunigen.

2.7 SAGA: Ein Genetischer Algorithmus

Notredame und Higgins beschreiben in [19] einen genetischen Algorithmus zur Erstellung eines multiplen Alignments.

Zuerst wird eine Menge von zufälligen Startlösungen generiert. Nun werden auf iterativer Weise immer wieder neue Lösungen erzeugt, welche andere aus der Population verdrängen. Ziel ist, dass die Güte der Lösungen in der

Population in Richtung Optimum konvergieren soll. Um aus bestehenden Lösungen neue zu erzeugen, wurden 22 Operatoren implementiert. Diese werden in zwei Klassen eingeteilt:

Crossover Operatoren: Dieser Operatorentyp erzeugt aus zwei Eltern-Alignments ein neues Kind-Alignment.

Mutationsoperatoren: Diese Operatoren verändern (mutieren) eine bestehende Lösung um ein neues, aber ähnliches Alignment zu erzeugen. Dazu zählen unter anderem *block shuffling* (also das Verschieben von ganzen Blöcken im Alignment) und *gap insertion* (das Einfügen von Lücken an bestimmten Stellen).

Diese eben beschriebenen Operatoren werden auf die bestehende Population so angewandt, dass eine Menge von neuen Alignments entsteht. Diese neuen Lösungen werden danach mit Hilfe einer Zielfunktion (*objective function*) bewertet. Zur Berechnung der Zielfunktion werden die Ähnlichkeit der alinierten Paare, sowie sog. *gap penalties* berücksichtigt. Je besser ein Alignment ist, desto höher ist dabei die Wahrscheinlichkeit, dass es in die nächste Generation der Population übernommen wird. Durch geeignete Maßnahmen wird dabei sichergestellt, dass die Population zu einem (unter Umständen nur lokalen) Optimum konvergiert. Tests belegen, dass dieser Algorithmus im Allgemeinen sehr robust ist, also meistens zum globalen Optimum hin konvergiert.

2.8 BAliBASE-Benchmark

Um die entwickelten Algorithmen zu testen, benutzen wir die Referenzsequenzen aus der BAliBASE 1.0-Datenbank, die unter anderem in [28] und [27] genauer beschrieben wird. Die Testdaten dieser Sammlung wurde von Biochemikern so ausgewählt, dass ein breites Spektrum an möglichen Aminosäuresequenzen abgedeckt wird und damit zuverlässige Aussagen über die Qualität eines Lösungsalgorithmus gemacht werden können. Außerdem wurden alle BAliBASE-Sequenzen „händisch“ von Fachpersonal optimal aliniert und annotiert. Kernbereiche, die als sicher gelten, wurden zusammen mit anderen Informationen in Annotationsdateien vermerkt. Mit Hilfe eines speziellen Bewertungsprogrammes (*bali_score*) können nun maschinell berechnete Alignments mit den optimalen Alignments verglichen werden. Dieses Programm vergibt dabei zwei verschieden Noten zwischen Null (keine Übereinstimmung) und Eins (vollkommene Übereinstimmung). Die zwei Arten der Benotung sind:

Sum of Pairs Score (SPS) Ein Alignment mit k Sequenzen und l Spalten ist gegeben. Dann definieren wir $p_{a,b,i}$ für das Symbolpaar $(s_{a,i}, s_{b,i})$ so, dass $p_{a,b,i} = 1$, falls $s_{a,i}$ auch im Referenzalignment mit $s_{b,i}$ aliniert

ist, ansonsten sei $p_{a,b,i} = 0$. Der Score für die i-te Spalte S_a berechnet sich wie folgt:

$$S_i = \sum_{a=1,a\neq b}^{k} \sum_{b=1}^{k} p_{a,b,i}.$$

Der Sum of Pairs Score ist dann:

$$SPS = \frac{\sum_{i=1}^{l} S_i}{\sum_{i=1}^{lr} S_{ir}},$$

wobei lr die Anzahl der Spalten und S_{ir} der Score S_i der i-ten Spalte im Referenzalignment ist.

Column Score (CS) Für die i-te Spalte im Alignment ist der Score $C_i = 1$, falls alle Symbole der i-ten Spalte auch im Referenzalignment so aliniert wurden, ansonsten gilt $C_i = 0$. Der Column Score für ein multiples Alignment ist dann:

$$CS = \sum_{i=1}^{l} \frac{C_i}{l}.$$

Zusätzlich gibt es noch die sogenannten annotierten Scores *annSPS* und *annCS*, bei denen noch die Informationen aus den sog. *annotation files* mit einfließen, welche unter anderem die Kernbereiche der Sequenzen kennzeichnen. Bei diesen werden die richtig alinierten Kernbereiche stärker bewertet, als andere, biochemisch nicht so relevante Teile.

Die BAliBASE-Alignments können weiters in verschiedene Kategorien (*References*) unterteilt werden. Insgesamt gibt es fünf Kategorien, wobei *Reference 1* in drei weitere Untergruppen unterteilt ist. Tabelle 2.2 gibt einen genauen Überblick.

Die Unterschiede zwischen den einzelnen Gruppen liegen vor allem in der Länge der Sequenzen, der Anzahl an kolinearen Sequenzen, sowie einigen weiteren biochemischen Fakten wie zum Beispiel dem Verwandtschaftsgrad der Proteine, worauf hier jedoch nicht weiter eingegangen werden soll.

Reference	Kurz (¡100 Residuen)	Mittel (200-300 Residuen)	Lang (¿400 Residuen)
Ref. 1: equidistante Sequenzen mit ähnlicher Länge			
V1 (¡25% Identität)	7	8	8
V2 (20-40% Identität)	10	9	10
V3 (¿35% Identität)	10	10	8
Ref. 2: Familien und weit entfernte Verwandte	9	8	7
Ref. 3: equidistante divergente Familien	5	3	5
Ref. 4: N/C-terminale Extensionen	12		
Ref. 5: Insertionen	12		

Tabelle 2.2: Unterteilungen in BAliBASE 1.0

Kapitel 3

Erstellung des Alignmentgraphen

3.1 Paarweiser Alignmentgraph

Der paarweise Alignmentgraph stellt eine zentrale Datenstruktur für alle implementierten Algorithmen dar. Er speichert alle *optimalen* paarweisen Alignments der Eingabesequenzen, die mit Hilfe von ClustalW erzeugt wurden. Abbildung 3.1 zeigt einen Beispiel-Alignmentgraphen.

Mit Hilfe von Gewichtungsfunktionen, die im nächsten Kapitel beschrieben werden, wird jeder Kante im Alignmentgraphen ein Gewicht zugeordnet, welches die Güte der alinierten Symbole widerspiegelt. Um darauf möglichst gute Lösungen zu erzeugen, muss ein Algorithmus Kanten aus dem paarweisen Alignmentgraphen auswählen und in den Lösungstrace übernehmen. Dabei muss darauf geachtet werden, dass diese nicht in Konflikt mit anderen Kanten stehen, der Trace also immer gültig bleibt. Außerdem soll das Gesamtgewicht der Kanten, die in den Trace übernommen worden sind, maximal sein.

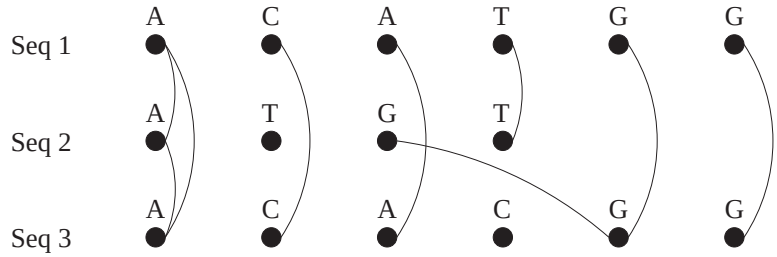

Abbildung 3.1: Paarweise Alignments in einem Alignmentgraphen

3.2 Kantengewichtung

Allen Kanten des paarweisen Alignments soll ein Gewicht zugeordnet werden, welches die Güte der Kante repräsentiert. Diese Güte der Kanten hängt von mehreren Faktoren ab. Der wichtigste Faktor ist jedoch, wie gut die beiden Symbole zusammenpassen. So ist im Allgemeinen das Alignment von zwei gleichen Symbolen gegenüber jenes von unterschiedlichen Symbolen zu bevorzugen. Für die Implementierung der Gewichtung von Kanten ist es nun sehr wichtig, dass vielversprechende Kanten höher gewichtet werden, als jene Kanten, welche zwei weniger gut übereinstimmende Symbole verbinden. Hier gibt es natürlich mehrere Möglichkeiten, welche in den folgenden Unterkapiteln ausführlich beschrieben werden. Sie unterscheiden sich vor allem durch den Berechnungsaufwand, der Granularität der Gewichtsvergabe sowie der Lokalität.

3.2.1 Bewertung nach Notredame

Bei der Bewertung nach Notredame et al. [20] handelt es sich um eine sehr einfache Bewertungsfunktion. Sie bewertet immer nur die Ähnlichkeit von zwei Sequenzen, das heißt, alle Kanten zwischen zwei Symbolen von genau diesen zwei Sequenzen haben das gleiche Gewicht.

Um ein Gewicht für diese Kanten zu erhalten, wird zuerst die Anzahl jener Symbole errechnet, welche übereinstimmen (*matches*), sowie die Anzahl jener Symbole, welche im paarweisen Alignment unterschiedlich sind (*mismatches*). Das Gewicht w eines paarweisen Alignment (und somit aller Kanten, die daraus resultieren) wird dann wie folgt berechnet:

$$w = \frac{matches}{matches + mismatches} * 100$$

Somit erhält man für alle paarweisen Alignments einen Wert, welcher das Zusammenpassen der beiden Sequenzen widerspiegelt. Nachteilig wirkt sich allerdings die „Ungenauigkeit“ der Analyse aus. Falls kurze, sehr gut zusammenpassende Teile im Sequenzenpaar vorhanden sind, so werden diese unter Umständen schlecht bewertet, wenn das restliche Alignment schlecht ist, da wie bereits vorher erwähnt, alle Kanten im paarweisen Alignment das gleiche Gewicht erhalten. So werden diese guten Kanten im Algorithmus, welcher die Lösung erstellen soll, nicht genügend beachtet. Lokale Verbesserungen können ebenfalls nicht effektiv angewandt werden, da diese häufig darauf beruhen, eine Kante aus der Lösung herauszunehmen und eine andere, bessere Kante aus dem gleichen Sequenzenpaar in die Lösung aufzunehmen. Haben jedoch alle das gleiche Gewicht, führt diese Vorgehensweise zu keinem sinnvollen Verbesserungsalgorithmus.

3.2.2 Bewertung nach ClustalW-Pairwise-Score

Auch diese Bewertungsfunktion berechnet für alle Kanten zwischen einem Sequenzenpaar im Alignmentgraphen das gleiche Gewicht. Somit unterliegt diese Bewertungsfunktion den gleichen Einschränkungen wie jene nach Notredame. Die Bewertung der Paare erfolgt mit Hilfe des ClustalW-Programms. Die genaue Realisierung findet man in [29]. Sie ist jedoch jener von Notredame sehr ähnlich.

3.2.3 Bewertung mittels ClustalW Bewertungsmatrizen

ClustalW wählt beim Erstellen der paarweisen Alignments eine Matrix aus, die für Aminosäuren Ähnlichkeitskoeffizienten angibt (siehe auch Abschnitt 2.3).

Da Symbole Aminosäuren oder Basen darstellen, muss man hier differenziert vorgehen. Wenn man auf zwei verschiedene Aminosäuren trifft, kann deren chemische Struktur trotzdem sehr ähnlich sein. Daraus folgt, dass es beim Vergleich von Proteinsequenzen, der in der Praxis häufiger ist, als der Vergleich von DNA-Sequenzen, sinnvoll sein kann, eine Matrix zu verwenden, in welcher Gewichte angegeben werden, die beschreiben wie gut zwei Aminosäuren (also Symbole) zusammenpassen.

Unsere Gewichtsberechnung für eine Alignmentkante (s_1, s_2) greift auf die gewählte Matrix *scorematrix* vom ClustalW-Programm zu und gibt dieser Kante im Alignmentgraphen das normalisierte Gewicht vom entsprechenden Matrixeintrag. Im Konkreten wird das Gewicht w wie folgt berechnet:

$$w = \frac{scorematrix[s_1][s_2] - offset}{maxValue - offset}$$

wobei *offset* der betragsgrößte negative Wert und *maxValue* der größte positive Wert ist. Das bedeutet einfach, dass die Bewertungen der Matrix auf einen Bereich [0,1] skaliert werden.

Durch diesen Typ von Bewertung können sich gute Kanten in einem paarweisen Alignment von den schlechten abheben, gute Teilsequenzen werden jedoch nicht als Ganzes erfasst. Das heißt, dass es einem auf dieser Bewertungsfunktion aufbauendem Algorithmus nicht möglich ist, vollständige Teile aus einem paarweisen Alignment zu übernehmen, sondern immer nur einzelne Kanten, welche viele andere, nur durchschnittlich bewertete Kanten, die in einem relativ gut alinierten Teilstück liegen, ungültig machen können. Dieser Umstand kann sich auf die damit erzielbaren Resultate negativ auswirken.

3.2.4 Bewertung mit fixer Fenstergröße

Um nun auch die Nachbarkanten in die Bewertung einfließen zu lassen, wurde das oben beschriebene Schema erweitert. Unser Ziel soll es ja sein, dass gute Teilsequenzen möglichst als Ganzes in die Lösung übernommen werden. Das bedeutet, dass wir eine Kante, welche inmitten einer hochwertigen Sequenz liegt, höher bewerten wollen, als eine Kante, die nur von weniger passenden Symbolen umgeben ist. So eine Bewertungsfunktion erreichen wir, indem wir ein ganzes Fenster rund um die aktuell zu untersuchende Kante betrachten, wobei Kanten in der Nähe der momentan betrachteten „Hauptkante“ stärker gewichten wollen, als Kanten in größerer Entfernung. Dabei scheint sich ein zur Entfernung proportionaler linearer Abstieg der Gewichtung besonders gut zu eignen. In Vortests zeigte sich, dass eine lineare Skalierung anderen Skalierungen, wie zum Beispiel einer geometrischen, überlegen ist.

Ein weiterer wichtiger Faktor ist die Fenstergröße. Denkbar ist eine konstante Größe, mit n Elementen in beide Richtungen. Man kann die Länge des Fensters aber auch in Abhängigkeit der Gesamtlänge des Sequenzenpaares berechnen. Folgt man den dargelegten Überlegungen, so wird das Gewicht w einer Kante e_m wie folgt berechnet:

$$w = w(e_m) + \sum_{i=1}^{n} \left[\left(w\left(e_{m-i}\right) + w\left(e_{m+i}\right) \right) * \left(1 - \frac{1}{n+1} * i \right) \right],$$

wobei $w(e_m)$ das Gewicht der Kante e_m aus der ClustalW-Bewertungsmatrix bezeichnet.

Diese Bewertungsfunktion besitzt jedoch noch den Nachteil, dass keine Anstrengungen unternommen werden, um das Ende eines gut alinierten Teilstückes zu erkennen. Es könnte vorteilhaft sein, wenn die Fenstergröße nicht „blind“ gewählt, sondern von der Güte der Nachbarkanten abhängen würde. Eine Annäherung an dieses Prinzip stellt die nächste Berechnungsmethode dar.

3.2.5 Bewertung mittels durch Lücken beschränkter Fenstergröße

Dieses Bewertungsschema basiert auf der „Bewertung mit fixer Fenstergröße“ (siehe Abschnitt 3.2.4), jedoch wird jetzt auch die Güte der Nachbarkanten miteinbezogen um die Fenstergröße zu berechnen. Eine mögliche Realisierung geht davon aus, dass die Qualität der Nachbarschaftskanten genau dann einen Schwellenwert unterschreitet, wenn in einem der beiden betrachteten Sequenzen eine Lücke (*gap*) auftritt. Mit dieser Methode wird vereinfachend angenommen, dass genau eine solche Lücke das Ende einer guten Teilsequenz darstellt, und dass deshalb darauf folgende Symbole nicht mehr betrachtet werden sollten. Wir betrachteten zwei konkrete Realisie-

rungen:

Im ersten Ansatz wird in beide Richtungen von der Hauptkante ausgehend die Anzahl der Symbole bis zur ersten Lücke (bzw. bis zum Ende einer Sequenz) bestimmt. Danach werden die Gewichte in beide Richtungen unabhängig skaliert, sodass das Gewicht der Hauptkante mit eins multipliziert wird und die Lücke mit Null. Dazwischen wird linear abfallend gewichtet:

$$w = w(e_m) + \sum_{i=1}^{n1} \left[w\left(e_{m-i}\right) + * \left(1 - \frac{1}{n1+1} * i\right)\right] + \\ + \sum_{i=1}^{n2} \left[w\left(e_{m+i}\right) + * \left(1 - \frac{1}{n2+1} * i\right)\right],$$

wobei $w(e_m)$ das Gewicht der Kante e_m aus der ClustalW-Bewertungsmatrix bezeichnet. $n1$ kennzeichnet jene Position links und $n2$ jene Position rechts von e_m, bei der die erste Lücke auftritt oder das Ende der Sequenz erreicht wird.

Meistens treten jedoch nur sehr wenige Lücken auf, sodass die betrachteten Teilsequenzen häufig sehr groß werden. Dies kann zwei Probleme mit sich bringen: (a) wenn die betrachtete Teilsequenz sehr lang ist, haben wir nur einen sehr flachen Abfall in der Skalierung, sodass relativ weit entfernte Symbolpaare noch einen sehr starken Einfluss auf die Hauptkante haben können; (b) dies bedeutet, dass auch andere Faktoren als das Auftreten einer Lücke das Ende einer guten Sequenz kennzeichnen. Deshalb wurde der zuerst beschriebene Ansatz wie folgt modifiziert: Es gibt eine obere Schranke für die Fenstergröße, welche festlegt, wie viele Nachbarschaftskanten die Bewertungsfunktion maximal berücksichtigt. Die Skalierungsfaktoren werden nun so gewählt, dass das Element maximaler Entfernung mit Null skaliert wird, während die Hauptkante mit Eins multipliziert wird. Tritt bei der Berechnung des Gewichtes innerhalb dieses Fensters eine Lücke auf, so werden weitere Elemente nicht mehr betrachtet, ansonsten wird beim Erreichen der oberen Schranke abgebrochen:

$$w = w(e_m) + \sum_{i=1}^{n} \left[w\left(e_{m-i}\right) * \left(1 - \frac{1}{n+1} * i\right) * p\right] + \\ + \sum_{i=1}^{n} \left[w\left(e_{m+i}\right) * \left(1 - \frac{1}{n+1} * i\right) * q\right].$$

Es gilt $p = 1$, falls noch keine Lücke bei Position $m - i$ aufgetreten ist, ansonsten ist $p = 0$; ähnlich gilt $q = 1$ falls noch keine Lücke bei Position $m + i$ aufgetreten ist, ansonsten ist auch $q = 0$.

Durch diese Maßnahmen ergeben sich folgende Eigenschaften im Gegensatz zu der vorher beschriebenen Methode:

- Nur ein Durchlauf des Fensters ist nötig um das Gewicht einer Kante zu berechnen, während vorher ein Durchlauf gebraucht wurde, um die Entfernung zur nächsten Lücke zu berechnen, mit deren Hilfe die Skalierungsfaktoren berechnet werden, und schließlich noch ein zweiter, um das Gesamtgewicht zu berechnen.

- Symmetrische Skalierungsfunktion: sei s_m die gerade betrachtete Hauptkante, dann werden die Kanten s_{m-i} und s_{m+i} mit dem gleichen Skalierungsfaktor versehen, während im ersten Versuch durchaus asymmetrische Skalierungen vorkommen können, da diese ja dynamisch zur Entfernung der Lücken auf der linken bzw. der rechten Seite gewählt werden.

- Die betrachteten Teilstücke sind eher kurz, während sie im vorigen Ansatz unter Umständen über ein ganzes paarweises Alignment gehen können (falls dieses vollständig lückenfrei ist).

3.2.6 Weitere Ansätze zur Bestimmung der Kantengewichte

Es wurden noch einige weitere Vorgehensweisen zu Bestimmung der Kantengewichte implementiert, die jedoch vor allem zum Testen und Vergleichen eingesetzt werden:

- Falls die beiden Symbole, welche eine Kante verbindet, gleich sind, bekommt die Kante ein Gewicht von 1, ansonsten 0.

- Die Kantengewichte werden zufällig zugewiesen.

- Wenn die beiden Symbole übereinstimmen, bekommt die Kante ein Gewicht gleich dem ClustalW-Score, ansonsten wird das Gewicht gleich ClustalW-Score/100 gesetzt.

- Eine Kombination von zwei Bewertungsschemata: Primär wird dabei der ClustalW-Score benutzt. Damit kann dann zum Beispiel mit einer Greedy-Heuristik eine Lösung erzeugt werden. Nun ist es jedoch normalerweise nicht möglich, lokale Verbesserungsalgorithmen zu benutzen, da mit dieser Bewertungsmethode alle Kanten in einem paarweisen Alignment das gleiche Gewicht haben. Um dies trotzdem zu ermöglichen, wird eine zweite Bewertungsfunktion angewandt. Hier eignet sich unter anderem eine „Bewertung mittels durch Lücken beschränkter Fenstergröße“ (siehe Abschnitt 3.2.5). Testergebnisse bestätigen, dass mit Hilfe dieser Kombination von Bewertungsverfahren bessere Ergebnisse erzielt werden, als mit einer reinen ClustalW-Bewertung. Dieses Verfahren unterliegt in seiner Güte jedoch den meisten anderen Bewertungsverfahren, die mit Hilfe von Gewichtsmatrizen und Nachbarkanten arbeiten.

3.2.7 Auswertung

Alle oben beschriebenen Verfahren zur Bewertung von Kantengewichten wurden ausführlichen Tests unterzogen, deren Resultate in diesem Kapitel zusammengefasst werden. Tabelle 3.1 zeigt jene Resultate, welche man erhält, wenn der Algorithmus Greedy 2 aus Kapitel 4.2 (siehe Abschnitt 4.2) angewandt wird und als Bewertungsschema Notredame bzw. ClustalW benutzt werden. Diese Tabelle kann wie folgt interpretiert werden: Für alle

	Notredame		ClustalW-Pairwise Score	
Referenz	SP	annSP	SP	annSP
ref 1	0,735	0,813	0,743	0,819
ref 2	0,719	0,794	0,725	0,791
ref 3	0,561	0,630	0,575	0,665
ref 4	0,585	0,731	0,576	0,726
ref 5	0,721	0,842	0,727	0,848
Summe	0,701	0,788	0,706	0,794

Tabelle 3.1: Notredame und ClustalW-Pairwise-Score

fünf Referenzen wurde der Durchschnitt berechnet, sodass man eventuelle schlechte Ergebnisse leichter auf ihre Ursachen zurückführen kann. Um zum Schluss mit Hilfe einer Zahl die Gesamtgüte angeben zu können, wurde auch der Durchschnitt der Benotungen über das gesamte Testfeld berechnet. Für beide Gewichtungsverfahren werden dabei die zwei Benotungen *SP* und *annSP*.

Wie zu erwarten, sind die Ergebnisse der beiden Verfahren sehr ähnlich. Wie bereits vorher erwähnt, gibt es auch viele Ähnlichkeiten in den beiden Gewichtungsfunktionen. ClustalW berücksichtigt jedoch mehr Faktoren als die Bewertung nach Notredame (u.a. werden auch Lücken in der Bewertung berücksichtigt und sog. *gap penalties* vergeben). Dies schlägt sich auch in den erhaltenen Resultaten nieder.

Beinahe überall erhält man mit Hilfe des ClustalW-Scores bessere Ergebnisse als mit der einfacheren Variante von Notredame. Die beiden gerade besprochenen Varianten vergeben an alle Kanten zwischen einem Paar von Sequenzen die gleichen Gewichte, die nächsten beiden Gewichtungsfunktionen bestimmen für jede Kante ein individuelles Gewicht. Eines dieser Verfahren vergibt ein Gewicht entsprechend der ClustalW-Bewertungsmatrizen, die zweite betrachtet nicht nur eine einzelne Kante sondern einen ganzen Teilstring (2% der Gesamtlänge einer Sequenz). Die Ergebnisse fasst Tabelle 3.2 zusammen.

Wir sehen, dass eine individuelle Bewertung der Kanten nicht unbedingt vorteilhaft ist. Die Bewertungsfunktion mit ClustalW-Matrizen schneidet in den Tests am schlechtesten ab. Dies liegt primär daran, dass dieser Ansatz das andere Extrem zu den paarweisen Alignmentscores darstellt. Nun wird

	ClustalW-Matrizen		Fenster fixer Größe	
Referenz	SP	annSP	SP	annSP
ref 1	0,706	0,793	0,773	0,853
ref 2	0,593	0,677	0,768	0,833
ref 3	0,413	0,481	0,613	0,700
ref 4	0,515	0,691	0,590	0,734
ref 5	0,613	0,735	0,779	0,918
Summe	0,635	0,733	0,739	0,830

Tabelle 3.2: ClustalW-Scoringmatrizen und mit Fenster mit fixer Größe

jede Kante unabhängig von allen anderen bewertet. Somit werden längere, sehr gute Sequenzen nicht gemeinsam erfasst und auch in den Lösungen nicht als solche realisiert. Eine ideale Bewertungsfunktion sollte wohl jeder Kante ein unterschiedliches Gewicht zuordnen können, sie sollte jedoch auch in der Lage sein, die umgebenden Symbole mit einzubeziehen. Genau dies versucht die Bewertungsfunktion, welche ganze Teilstrings zur Bewertung einer einzelnen Kante hinzuzieht. Diese Vorgehensweise wird uns auch von den Testergebnissen als gut bestätigt. Sowohl die *SP*-, als auch die *annSP*-Werte von allen Gruppen sind (teilweise deutlich) besser als jene Ergebnisse, welche man mit dem ClustalW-Pairwise-Score erhält.

Als letzte Bewertungsfunktion wollen wir die „Bewertung mittels durch Lücken beschränkter Fenstergröße“ genauer untersuchen. Dabei wurden mehrere Testläufe mit verschiedener maximaler Fenstergröße durchgeführt. Einmal wurde für die Fenstergröße keine obere Schranke festgelegt und als Grenze des Fensters die erste auftretende Lücke auf beiden Seiten gewählt. Die Skalierung für den linken und den rechten Teilstring wurde exakt berechnet, ansonsten testeten wir eine maximale Größe des Fensters von insgesamt 13, 17, 21, 31 und 51 Elementen, wobei hierbei die Skalierung der Nachbarkanten so gewählt wurde, dass das maximal entfernte Element mit Null und die Hauptkante selbst mit Eins gewichtet werden und die Skalierung dazwischen linear abfällt. Die Tabellen 3.3 und 3.4 zeigen einen Auszug der dabei erhaltenen Resultate.

Die Resultate zeigen, dass die variable Fenstergröße einer fixen Fenstergröße überlegen ist. Ein weiteres interessantes Detail ist, dass eine nur durch Lücken beschränkte Fenstergröße einer variablen Fenstergröße mit einer oberen Schranke unterlegen ist. Dies liegt daran, dass in einem multiplen Alignment meist nur sehr wenige Lücken vorkommen und somit der betrachtete Teilstring sehr lang werden kann, falls keine Einschränkungen getroffen werden. Das heißt jedoch widerum, dass sehr weit entfernte Elemente die Hauptkante noch sehr stark beeinflussen können. Unser Ziel sollte jedoch sein, dass nur die unmittelbare Umgebung die Bewertung einer Kante beeinflussen sollte. Eine obere Schranke zu wählen ist somit ein guter Ansatz.

	Variable Fenstergröße (keine obere Schranke)	
Referenz	SP	annSP
ref 1	0,770	0,850
ref 2	0,774	0,833
ref 3	0,653	0,738
ref 4	0,610	0,761
ref 5	0,789	0,921
Summe	0,745	0,834

Tabelle 3.3: Ergebnisse mit Fenster variabler Größe nur durch Lücken beschränkt

	Variable Fenstergröße (21 Elemente)		Variable Fenstergröße (31 Elemente)	
Referenz	SP	annSP	SP	annSP
ref 1	0,780	0,862	0,770	0,850
ref 2	0,780	0,844	0,774	0,833
ref 3	0,643	0,723	0,653	0,738
ref 4	0,616	0,775	0,610	0,761
ref 5	0,795	0,924	0,789	0,921
Summe	0,752	0,843	0,745	0,834

Tabelle 3.4: Ergebnisse mit Fenster variabler Größe mit oberer Schranke

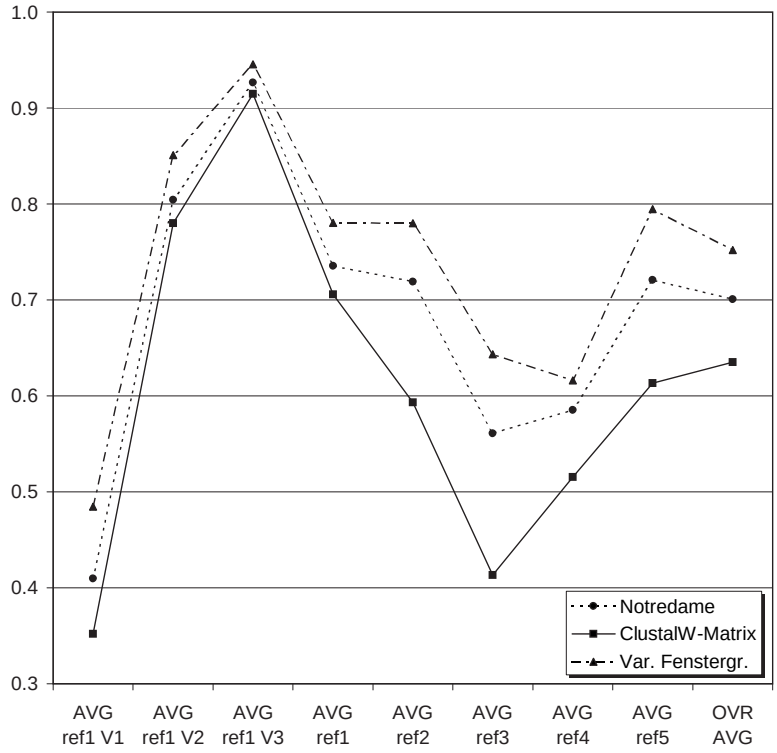

Abbildung 3.2: Diagramm: Verschiedene Bewertungsmethoden für Kanten

Bei den Tests stellte sich heraus, dass mit einer maximalen Anzahl von 21 berücksichtigten Elementen (also der Hauptkante und maximal 10 Elementen auf jeder Seite) die besten Ergebnisse erzielt werden. Das Diagramm in Abbildung 3.2 verbildlicht die wichtigsten Ergebnisse noch einmal. Zusätzlich werden dort noch die Werte für die Teilbereiche V1, V2 und V3 der Referenz 1 aufgeführt.

3.3 Alignmentgraph Extension

Um in den paarweisen Alignments nun auch globale Informationen aus anderen Sequenzen zu berücksichtigen, wurde eine *Alignmentgraph Extension* implementiert, die eine Erweiterung der *Library Extension* von Notredame aus [20] darstellt.

Führt eine Kante von $s_{i,x}$ nach $s_{j,y}$ und eine weitere Kante von $s_{j,y}$ nach $s_{k,z}$, so kann der paarweise Alignmentgraph um die transitive Kante $(s_{i,x}, s_{j,y})$ erweitert werden, oder, falls diese schon existiert, deren Gewicht angepasst werden. Diese Vorgehensweise kann auf mehr als drei Symbolen verallgemeinert werden:

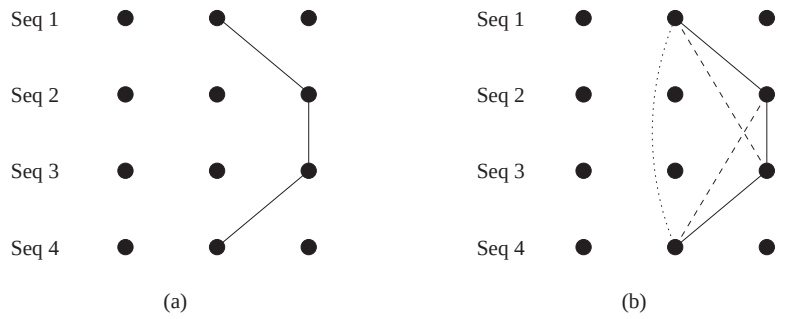

Abbildung 3.3: Alignmentgraph Extension
(a) zeigt den Ausgangsgraph, (b) den Ergebnisgraph, wobei - - - einen Grad $k = 2$ und $\cdots$ einen Grad $k = 3$ kennzeichnet.

Definition 3.1 (Edge Trace) *Seien s_{a_i,p_i} Symbole von $k \geq 2$ verschiedenen Sequenzen S_{a_i}. Dann wird eine Sequenz von Kanten von s_{a_1,p_1} nach s_{a_k,p_k}* edge trace *genannt, wenn eine Kante zwischen allen s_{a_i,p_i} und $s_{a_{i+1},p_{i+1}}$ $\forall i = 1 \ldots k-1$ existiert.*

Definition 3.2 (Transitive Kante) *Existiert ein edge trace von s_{a_1,p_1} nach s_{a_k,p_k}, so heißt die Kante $(s_{a_1,p_1}, s_{a_k,p_k})$* transitive Kante mit Grad *k.*

Das Gewicht einer transitiven Kante e_t mit Grad k wird wie folgt berechnet:

$$w(e_t) = \frac{w_{min}(e_i)}{(k-1)}, k \geq 2,$$

wobei w_{min} das minimale Gewicht aller Kanten e_i bezeichnet.

Der Aufwand um eine Alignmentgraph Extension zu berechnen, liegt in $O(l \cdot n^{k+1})$, wobei n die Anzahl der Sequenzen, l die Länge der Sequenzen und k den Grad der Alignmentgraph Extension bezeichnet (meist gilt $l \gg n$).

3.3.1 Auswertung

Tabelle 3.5 zeigt die Ergebnisse von Greedy 2 mit und ohne Anwendung der Alignmentgraph Extension ($k = 2$). Abbildung 3.4 zeigt dieselben Ergebnisse in einem Liniendiagramm.

	Greedy 2		Greedy 2 + Alignment-graph Extension ($k = 2$)	
Referenz	SP	annSP	SP	annSP
ref 1	0,780	0,862	0,789	0,865
ref 2	0,780	0,844	0,797	0,852
ref 3	0,643	0,723	0,688	0,783
ref 4	0,616	0,775	0,646	0,803
ref 5	0,795	0,924	0,802	0,904
Summe	0,752	0,843	0,767	0,852

Tabelle 3.5: Ergebnisse mit Alignmentgraph Extension bei der Anwendung von Greedy 2. Als Kantengewichtung wurde die Bewertung „Fenster variabler Größe mit oberer Schranke (21 Elemente) gewählt“

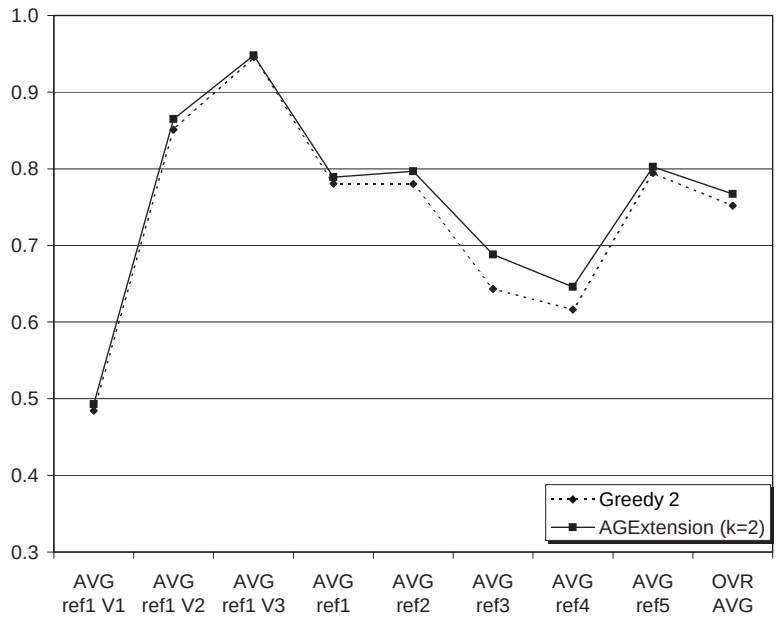

Abbildung 3.4: Diagramm: Greedy2 mit und ohne Alignmentgraph Extension

Kapitel 4

Greedy Algorithmen

4.1 Greedy 1

Dieser Algorithmus basiert auf einem Ansatz, welcher von Kececioglu in [12] beschrieben und in Kapitel 2.5 zusammengefasst wird. In diesem Paper benutzt der Autor ein Branch and Bound-Verfahren um das Multiple Alignment Problem zu lösen. Dabei ist es wichtig mit Hilfe einer Heuristik eine gute untere Schranke zu errechnen um schlechte Teillösungen frühzeitig verwerfen zu können. Dies bewerkstelligt er mit einem Greedy-Algorithmus, der eine Ausgangsbasis für Greedy 1 bildet.

Prinzipiell versucht der Greedy-1-Algorithmus Spalte für Spalte des multiplen Alignments zu erzeugen. Dabei sollen jene Kanten des Alignmentgraphen, der die paarweisen Alignments abspeichert, realisiert werden, die ein möglichst hohes Gesamtgewicht haben. Dieser Algorithmus durchläuft mehrere Phasen, welche im Folgenden genauer beschrieben werden:

1. *Bestimmung des durch die Front definierten Subgraphen:* Das multiple Alignment wird von links nach rechts erzeugt. Anfänglich ist noch kein Symbol der Eingabesequenzen in der Lösung vorhanden (d.h. die Symbole wurden noch nicht plaziert). Alle Symbole, welche in weiteren Schritten in das Lösungsalignment aufgenommen werden, werden dementsprechend als *plaziert* markiert. Um nun eine neue Spalte der Lösung zu erzeugen muss zu allererst die *Front* bestimmt werden.

   Definition 4.1 (Front) *Bei der Front handelt es sich um eine Menge von Symbolen, die im Alignmentgraphen als Knoten repräsentiert werden. Zu dieser Menge V_F gehört jeweils das erste Symbol in jeder Sequenz, welches noch nicht platziert wurde.*

   Abbildung 4.1 zeigt ein Beispiel zur Bestimmung der Front. Ausgehend von dieser Front kann die Kantenmenge berechnet werden, welche in dernächsten Spalte des multiplen Alignment realisiert wird. Um dies

AGATAGTAGATCACAAA
AGCARAGAGTACCTGTA
AATAGTACATGCAAATA

Graue Symbole kennzeichnen die Front, unterstrichene die bereits verarbeiteten Codesequenzen

Abbildung 4.1: Bestimmung der Front

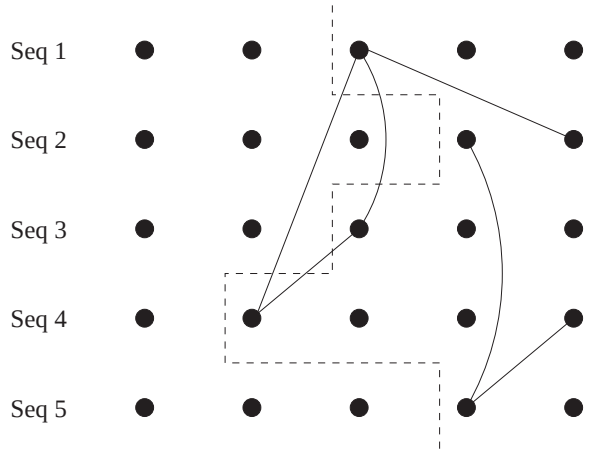

Abbildung 4.2: Alignmentgraph mit dem durch die Front induzierten Subgraphen; (- - -) kennzeichnet die aktuelle Front

zu erreichen muss nun ein Subgraph $G_S = (V_S, E_S)$ des Alignmentgraphen $G_A = (V_A, E_A)$ erstellt werden.

Dabei sind alle Knoten der Front in V_S enthalten, sowie die rechten Nachbarn dieser Knoten, sofern es diese gibt und sie über eine Kante mit einem Knoten der Front verbunden sind. Die Menge E_S beinhaltet genau jene Kanten des Alignmentgraphen, die zwei Knoten $v_i, v_j \in V_S$ verbinden, wobei mindestens ein Knoten v_i oder v_j aus der Front sein muss. Mit der Bestimmung von diesem Subgraphen G_S ist die erste Phase abgeschlossen. Abbildung 4.2 zeigt einen Alignmentgraphen und den durch die Front induzierten Subgraphen.

2. *Bestimmung der Komponenten:* Der gerade erstellte Subgraph muss nicht unbedingt zusammenhängend sein. Da wir daran interessiert sind, so viele gute Kanten wie möglich in die Lösung zu übernehmen, um das Gesamtgewicht aller Kanten zu maximieren, können wir alle zusammenhängende Teile dieses Graphen voneinander unabhängig betrachten, da durch diese Maßnahme sicherlich keine Kanten verloren

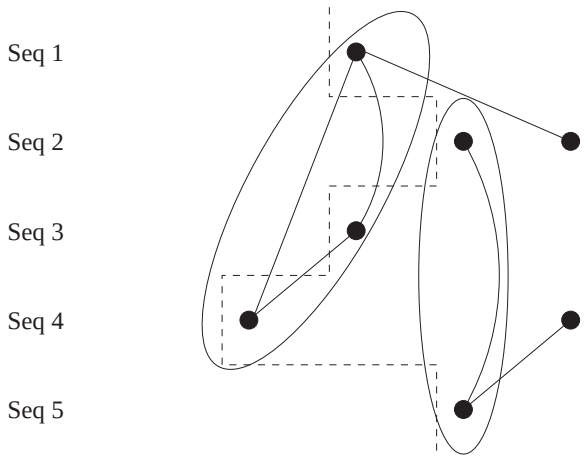

Abbildung 4.3: Komponenten des Subgraphen und deren *exposed edges*

gehen können, die Komplexität des Problems jedoch reduziert wird. Der Beweis dazu kann in [12] gefunden werden. Daraus folgt, dass wir in unserem nächsten Schritt die einzelnen Komponenten des Subgraphen G_S bestimmen. Hier werden nun zwei Arten von Kanten unterschieden. Zum Einen gibt es Kanten, welche zwei Knoten verbinden, die beide zur Front gehören, zum Anderen können auch Kanten existieren, die einen Knoten aus der Front mit einem Knoten rechts von der Front verbinden (sog. *exposed edges*). Eine Komponente $G_K = (V_K, E_K, E_{EXP})$ ist nun ein maximal zusammenhängender Teil des Subgraphen G_S, wobei die zwei oben beschriebenen Mengen von Kanten unterschieden werden, also $e_k = (v_i, v_k) \in E_K$ genau dann, wenn $v_i, v_j \in V_K$, sowie $v_i, v_j \in V_F$. Eine Kante $e_k = (v_i, v_k) \in E_{EXP}$ genau dann, wenn $v_i, v_j \in V_K$ sowie $(v_i \in V_F \wedge v_j \notin V_F) \vee (v_i \notin V_F \wedge v_j \in V_F)$. Abbildung 4.3 zeigt ein Beispiel für diese Berechnung.

3. *Bestimmung der Superkomponenten:* Da es das Ziel dieser Heuristik ist, so viele Kanten wie möglich in die Lösung zu übernehmen, versucht sie so wenig Kanten wie möglich zu „cutten“. Da unter Umständen nicht alle Kanten aus E_{EXP} realisiert werden können, muss zwischen den einzelnen verfügbaren Komponenten jene ausgewählt werden, welche am wenigsten Kanten verwirft (bzw. deren Summe der Gewichte aller verworfener Kanten minimal ist). Dies bedeutet für die Heuristik, dass sie bestimmte Komponenten bevorzugen soll. Welche nun für die weitere Verarbeitung gewählt wird, soll nun bestimmt werden.

   Um diese Aufgabe zu bewerkstelligen, werden sog. stark zusammenhängende Superkomponenten bestimmt. Dazu wird jede ermittelte Komponente $G_K = (V_K, E_K)$ zu einem einzelnen Knoten reduziert.

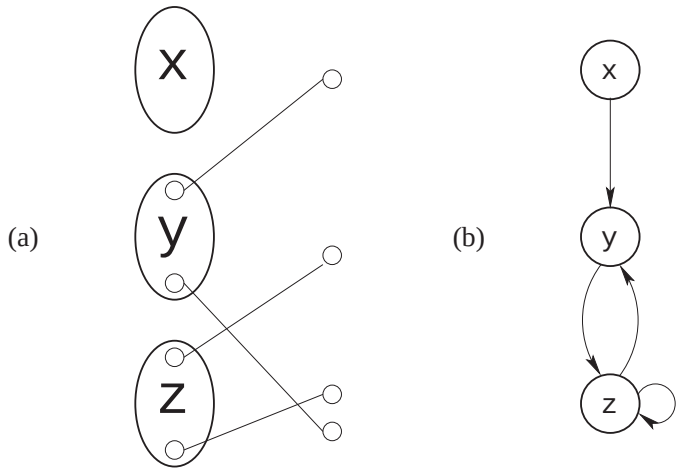

Abbildung 4.4: (a) Komponenten an der Front zusammen mit deren exposed edges. (b) Komponentengraph (Abbildung übernommen aus [12]).

Die Menge der so erhaltenen Knoten nennen wir V_{SK}. Nun wird genau dann eine gerichtete Kante von Komponente X nach Komponente Y gezogen, wenn rechts von der Komponente X ein Knoten existiert, welcher Teil einer *exposed edge* von Komponente Y ist. Durch dieses Vorgehen entsteht ein neuer gerichteter Graph $G_{SK} = (V_{SK}, E_{SK})$, den wir Superkomponentengraph nennen.

In diesem Graphen G_{SK} werden nun die stark zusammenhängenden (Super-)Komponenten bestimmt. Nun gibt es immer mindestens eine Superkomponente ohne Eingangskanten. Diese Superkomponente wird nun für die weitere Bearbeitung ausgewählt. Es lässt sich dabei zeigen, dass dies ein optimales Vorgehen ist (für einen Beweis dieser Behauptung siehe [12]).

4. *Bestimmung des Minimum Cut:* Nun haben wir eine Superkomponente ausgewählt, welche im Allgemeinen aus mehreren Komponenten bestehen kann, wobei über die *exposed edges* Beziehungen zwischen den einzelnen Komponenten existieren. Wählen wir eine bestimmte Komponente aus, deren Kanten E_K und E_{EXP} wir realisieren wollen, so werden andere mögliche Kanten unter Umständen im Multialignment nicht mehr realisierbar. Um hier eine möglichst gute Lösung zu erzeugen, sollte jene Komponente ausgewählt werden, welche am wenigsten Kanten (bzw. Kanten, deren Gesamtgewicht minimal ist) wegschneidet. Dazu wird ein Minimum Cut Algorithmus benutzt, der dieses Problem in einer optimalen Weise löst. Für jede Komponente in der ausgewählten Superkomponente erzeugen wir einen neuen Graphen. Kanten, die zwei Knoten aus der Front verbinden, können

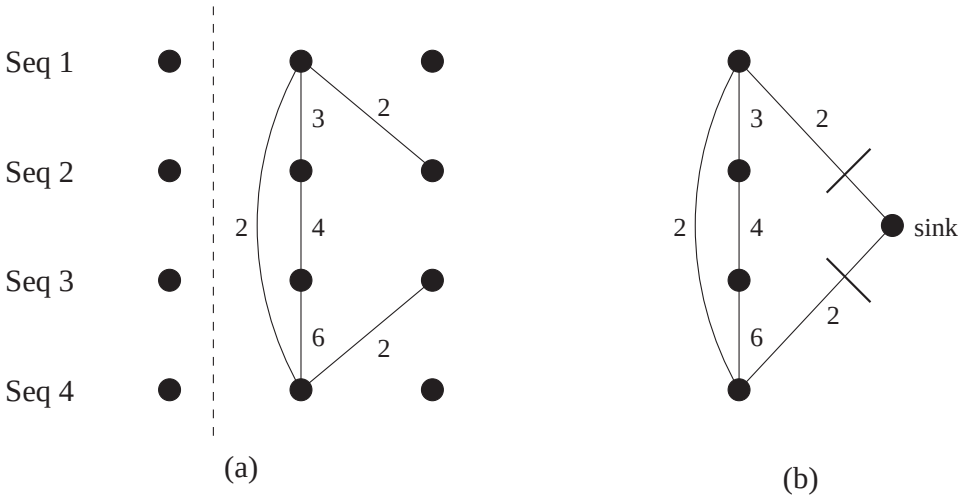

Abbildung 4.5: Minimum Cut einer Komponente:
(a) Ursprüngliche Komponente
(b) Graph, der die Komponente aus (a) darstellt mit neuem Knoten *sink* und der entsprechende Minimum Cut

direkt zusammen mit den inzidenten Knoten in den neuen Graphen übernommen werden. Alle Knoten, die nicht Teil der Front sind, werden durch eine Senke (*sink* ersetzt. Die *exposed edges* werden darauf dem neuen Graphen so hinzugefügt, dass jener Knoten, welcher nicht in der Front ist, durch die Senke ersetzt wird. Von diesem Graphen wird nun der Minimum Cut berechnet (siehe Abbildung 4.5), welcher jene Kantenmenge bestimmt, die verworfen werden muss, um vorhandene Konflikte zu beseitigen. Jene Komponente mit dem minimalen Minimum Cut wird für die weitere Verarbeitung ausgewählt.

5. *Hinzufügen der selektierten Kanten zur Lösung:* Nun werden alle Kanten, welche Teil der ausgewählten Komponente sind und nicht in der Menge ihres Minimum Cuts sind, in der Lösung eingetragen und im Alignmentgraphen als „platziert“ markiert. Solange noch mindestens eine Sequenz im multiplen Alignment existiert, dessen letztes Symbol noch nicht bearbeitet bzw. platziert wurde, wird der Algorithmus in Punkt 1 fortgesetzt, ansonsten terminiert er.

In jeder Iteration wird im Worst Case genau eine Kante zur Lösung hinzugefügt. Dies heißt, dass im schlimmsten Fall $|E|$ Iterationen nötig sein können. In jeder Iteration muss die Front bestimmt werden. Dazu ist ein Aufwand von $O(k)$ nötig, wobei k die Anzahl der Eingabesequenzen bezeichnet. Um eine Komponente zu bestimmen, müssen wir ebenfalls mit einer Laufzeit von $\Theta(k)$ rechnen, da auch hier für jede Sequenz mindestens

ein Knoten in einer Komponente liegt. Im schlechtesten Fall liegt jedoch jeder Knoten in einer anderen Komponente und alle Knoten an der Front können mit bis zu insgesamt k^2 Kanten verbunden sein. Dadurch ergibt sich für das Berechnen der Komponenten im Worst Case eine Laufzeit von $O(k^2)$. Bei der Bestimmung der Superkomponenten gilt die gleiche Argumentation wie bei der Berechnung der Komponenten. Deshalb haben wir auch hier eine Laufzeit von $O(k^2)$ pro Iteration. Das Hinzufügen der Kanten liegt schließlich zwischen $O(1)$ und $O(k^2)$ pro Iteration. Daraus ergibt sich eine Gesamtlaufzeit im Worst Case von $O(|E| \cdot k^2)$.

4.2 Greedy 2

Dieser Algorithmus arbeitet nach folgender Grundidee: Die Kanten des Alignmentgraphen werden fallend nach Gewicht sortiert und der Reihe nach im Alignment realisiert, sofern die neue Kante sich nicht mit vorangegangenen widerspricht. Genauer wird zunächst von einem Trace ausgegangen, der nur Arcs, aber keine Kanten aus dem Alignmentgraphen enthält. Eine Kante des Alignmentgraphen wird jeweils hinzugefügt, wenn sie mit den bisherigen Kanten und den Arcs keinen (gerichteten) Kreis bildet, der mindestens einen Arc beinhaltet. Wurden alle möglichen Kanten aus dem Alignmentgraphen im Trace eingefügt, wird daraus das multiple Alignment abgeleitet. Dieses ist jedoch im Allgemeinen nicht eindeutig. Unser Algorithmus erzeugt das kürzest mögliche mutliple Alignment mit einer vereinfachten Version des Greedy 1-Algorithmus Spalte für Spalte, indem zu jedem Zeitpunkt in einer Spalte alle möglichen nächsten Symbole der Front aufgenommen werden, die keine „Forward-Kanten“ (also Kanten zu nicht auch in die Spalte kommenden Symbolen an der Front) besitzen. Daraus folgt, dass immer alle Kanten des Trace realisiert werden.

Der wesentliche Punkt ist somit die Bestimmung, ob eine Kante des Alignmentgraphen zulässig ist und im Trace aufgenommen werden kann oder nicht.

Es gilt: Eine Kante $(s_{a,i}, s_{b,j})$ ist zulässig, wenn im Trace

1. ein Pfad zwischen den beiden Knoten bereits existiert, der nur aus Kanten des Alignmentgraphen und keinen Arcs besteht (d.h., die entsprechenden Symbole sind schon aliniert).

2. oder kein Pfad von $s_{a,i}$ nach $s_{b,j}$ und kein Pfad von $s_{b,j}$ nach $s_{a,i}$ (über Arcs) existiert.

Es ist daher notwendig, nach einem Pfad zwischen zwei Knoten effizient zu suchen. Wir beschränken uns in der folgenden Beschreibung auf die Richtung von $s_{a,i}$ nach $s_{b,j}$.

Dabei können wir festhalten:

1. Wir brauchen nur Pfade zu berücksichtigen, die maximal einmal zu jeder Sequenz führen und diese auch maximal einmal verlassen. Andere Pfade können auf solche durch entsprechende Abkürzungen über die Arcs abgebildet werden.

2. Ein gesuchter Pfad beinhaltet damit maximal $k-1$ Kanten plus den Arcs, deren Anzahl nicht durch k beschränkt ist.

3. Ist einmal von $s_{a,i}$ ausgehend ein Pfad zu einem $s_{c,l}$ gefunden, so sind durch die Arc-Verbindungen auch automatisch Pfade zu allen Knoten $s_{c,l+1}, \ldots, s_{c,|S_c|}$ gefunden.

4. Ist die Kante $(s_{a,m}, s_{c,l})$ jene mit minimalem $m \geq i$ zwischen den Sequenzen a und c, so braucht keine weitere Kante zwischen diesen beiden Sequenzen betrachtet zu werden. Für eine „spätere“ Kante $(s_{a,m'}, s_{c,l'})$ mit $m' > m$ muss auch $l' > l$ gelten, und damit kann diese Kante keine Verbesserung bringen. Es kann auch keine Kante $(s_{a,m'}, s_{d,l'})$ über eine dritte Sequenz d mit $m' > m$ geben, über die indirekt eine Verbesserung in Bezug auf Sequenz c erzielt wird.

Wir machen eine von $s_{a,i}$ ausgehende Suche und speichern global für jede Sequenz, welche minimale Position wir schon erreichen können. Seien $M_1, \ldots, M_k$ diese Variablen. Ist eine Sequenz noch nicht erreicht worden, so ist $M_c = \infty$. Weiterverfolgen brauchen wir nur Kanten, die zu einer Verbesserung in M führen. Bewirkt eine Kante keine Verbesserung, so brauchen wir vom Zielknoten weitere Pfade nicht zu untersuchen.

Um von einem Knoten $s_{a,m}$ die nächste Kante $(s_{a,m'}, s_{c,l})$ mit minimalem $m' >= m$ möglichst rasch zu finden, verwenden wir folgende Datenstruktur: Für jede zwei Sequenzen $c = 1, \ldots, k$ und $d = 1, \ldots, k$, $c \neq d$, gibt es eine *sorted sequence* $A_{c,d}$ (implementiert als Skip-Liste, AVL-Baum etc.), die Verweise auf alle Kanten zwischen den Sequenzen c und d sortiert nach der Position in der Sequenz c speichert. Der Aufwand, um die „nächste“ Kante zu einer Sequenz c zu finden, ist so $O(\log n)$.

Die Suche muss von einem aktuellen Knoten $s_{a,m}$ aus, sofern dieser M_a verbessert hat, alle Kanten $A_{a,d}(m)$ weiter verfolgen, wobei d für alle noch nicht besuchten Sequenzen auf dem aktuellen Pfad steht. Der Suchbaum hat damit Maximaltiefe k, in jedem Knoten der Tiefe t gibt es bis zu $k - t - 1 = O(k)$ noch nicht besuchte Sequenzen, die eventuell zu weiteren Verzweigungen führen. Das würde maximal $O(k!)$ zu untersuchende Pfade für die Suche bedeuten.

Tatsächlich sind aber wesentlich weniger Knoten zu betrachten, wenn eine Breitensuche eingesetzt wird und wir nur Pfade weiter verfolgen, die tatsächlich eine Verbesserung bewirkt haben.

Zu jeder Zeit gibt es immer nur maximal $k-1$ aktuelle Pfade, die weiter verfolgt werden müssen, nämlich jene, die zuletzt die M_c, $c = 1, \ldots, k$, $c \neq a$, verbessert haben.

Durch die Breitensuche ist im Gegensatz zu einer Tiefensuche gewährleistet, dass alle Pfade mit weniger Kanten als ein aktueller Pfad bereits vorher untersucht wurden. Nehmen wir an, alle Pfade bis zur Tiefe t wurden untersucht. Es kann, wie gesagt, max. $k-1$ solche Pfade geben, die wir weiter betrachten müssen. Bei jedem dieser Pfade müssen maximal $k-1-t$ Folgekanten untersucht werden, um alle in Frage kommenden Pfade bis zur Tiefe $t+1$ betrachtet zu haben. Das bedeutet, dass der Aufwand hierfür $O((k-1)(k-1-t)) = O(k^2)$ ist. Da wir bis maximal zur Tiefe k gehen müssen, ist der Gesamtaufwand für die Breitensuche im schlechtesten Fall $O(k^3)$.

In der Praxis wird die Suche noch verbessert, indem wir für die Reihenfolge der Bearbeitung aktiver Pfade nicht nur die Anzahl der Kanten, sondern auch die Anzahl der Arcs berücksichtigen. Pfade mit weniger Arcs sind naturgemäß vielversprechender. Insbesondere stellen Pfade, die keine Arcs beinhalten Sonderfälle dar. Sie beschreiben nämlich direkt alinierte Knoten. Die erreichten Positionen so verbundener Sequenzen können in der Suche nicht verbessert werden und brauchen daher nicht weiter verfolgt werden. Pfade ohne Arcs werden daher grundsätzlich immer anderen Pfaden vorgezogen.

Der Gesamtaufwand für Greedy 2 im Worst Case ist daher $O(|E| \log |E| + |E|k^3 \log n)$, wobei normalerweise der zweite Term den Aufwand dominiert. Im Erwartungsfall ist der Aufwand deutlich geringer.

4.3 Verbesserung von Greedy 2

Die Laufzeit von Greedy 2 konnte durch eine Erweiterung der Trace-Datenstruktur und der damit nicht mehr notwendigen Tiefensuche bei der Bestimmung, ob ein Knoten von einem anderen aus erreichbar ist, wie folgt wesentlich verbessert werden.

Im Trace werden neben den „echten“ *Tracekanten*, also Kanten, welche das Multiple Alignment definieren, zwei neue Arten von Kanten eingeführt:

abgeleitete Kanten: Sind zwei Knoten $s_{a,i}$ und $s_{b,j}$ (noch) nicht direkt über eine Kante $(s_{a,i}, s_{b,j})$ verbunden, gibt es aber eine indirekte Verbindung über eine oder mehrere andere Knoten (aber keine Arcs), also z.B. die Kanten $(s_{a,i}, s_{c,k})$ und $(s_{c,k}, s_{b,j})$, so wird eine abgeleitete Kante $(s_{a,i}, s_{b,j})$ im Trace gespeichert, die die indirekte Verbindung direkt repräsentiert und später durch eine „echte“ Tracekante aus dem Alignment-Graphen ersetzt werden kann.

Pfad-Kanten: Eine Pfad-Kante $(s_{a,i}, s_{b,j})$ hat die Bedeutung, dass ein über zumindest einen Arc führender gerichteter Pfad von Knoten $s_{a,i}$ zu Knoten $s_{b,j}$ existiert.

Abbildung 4.6: Erweiterter Trace:
(a) Ausgangstrace ohne erweiterte Kanten
(b) Neuer Trace mit abgeleiteten Kanten (- - -) und Pfadkanten ($\cdots$)

Abbildung 4.6 zeigt, wie ein Trace mit diesen erweiterten Kanten gegenüber einem ohne Zusatzkanten aussieht.

Im Trace gilt zu jeder Zeit für alle Paare von Sequenzen a und b, $a \neq b$:

1. Von jedem Knoten $s_{a,i}$ geht zur Sequenz b immer nur maximal eine Kante egal welchen Typs weg bzw. führt auch nur maximal eine Kante von Sequenz b zu $s_{a,i}$. Es kann aber vorkommen, dass zu einem Knoten eine Pfad-Kante führt und eine andere davon weggeht.

2. Es kommt nie zu Kreuzungen zwischen zwei Kanten mit einer Ausnahme: Eine von Sequenz a nach b führende Pfad-Kante kann sich mit einer Pfad-Kante von b nach a kreuzen.

3. Für einen Knoten $s_{a,i}$ beschreibt die von Sequenz a „nächste“ ausgehende Kante $(s_{a,i'}, s_{b,j})$ beliebigen Typs mit $i' \geq i$ (d.h., $\nexists(s_{a,i''}, s_{b,j'}) \mid i'' < i'$) alle in Sequenz b über einen Pfad erreichbaren Knoten. Das sind nämlich all jene Knoten mit Positionen $\geq j$. Gibt es keine solche Kante $(s_{a,i'}, s_{b,j})$ mit $i' \geq i$, so bedeutet das, dass noch kein Pfad von $s_{a,i}$ zu Sequenz b existiert.

Unter diesen Bedingungen kann somit auf eine Tiefensuche zur Bestimmung der Zulässigkeit einer Kante $(s_{a,i}, s_{b,j})$ des Alignment-Graphen in einem Trace verzichtet werden. Es muss lediglich die „nächste“ entsprechende Kante von jedem der beiden Knoten nachgeschlagen werden, um festzustellen, ob ein Kreis erzeugt werden würde.

Eine neue wichtige Aufgabe ist es nun aber, alle abgeleiteten Kanten und Pfad-Kanten korrekt zu erzeugen bzw. im weiteren Verlauf auch zu ersetzen. Dabei werden für eine im Trace aufgenommene „echte“ Tracekante beide Richtungen $(s_{a,i}, s_{b,j})$ und $(s_{b,j}, s_{a,i})$ untersucht. Wir betrachten hier den ersten Fall genauer, der zweite wird entsprechend behandelt:

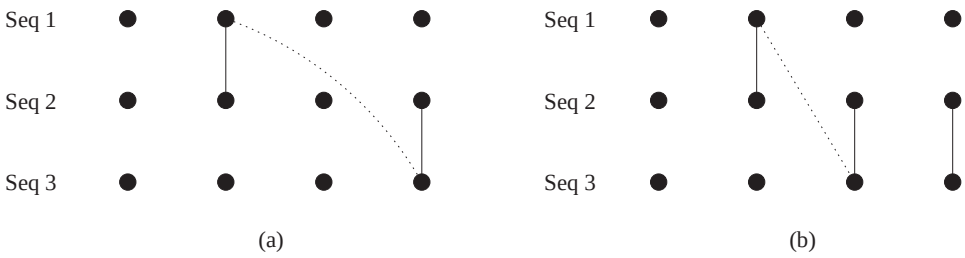

Abbildung 4.7: Aktualisieren einer Pfadkante:
(a) Ausgangstrace mit einer dargestellten Pfadkante ($\cdots$)
(b) Neuer Trace mit einer neuen Kante und aktualisierter Pfadkante

Grundsätzlich wird für jede Sequenz $c \neq b$ der erste von $s_{b,j}$ aus erreichbare Knoten $s_{c,l}$ bestimmt. Außerdem wird für jedes $d \neq a$ der „späteste“ Knoten $s_{d,m}$ mit maximalem m bestimmt, für den ein Pfad zu $s_{a,i}$ existiert. Für alle möglichen Paare $(s_{d,m}, s_{c,l})$ wird dann untersucht, ob über die neue Erreichbarkeit eine Verbesserung vorliegt und daher eine entsprechende Kante im Trace notwendig geworden ist.

Für das Ersetzen von Kanten gilt dabei:

- Eine bestehende abgeleitete Kante wird immer nur durch eine entsprechende echte Tracekante des Alignment-Graphen ersetzt.
- Eine Pfad-Kante $(s_{a,i'}, s_{b,j'})$ wird immer nur durch eine „bessere“ Kante $(s_{a,i''}, s_{b,j''})$ mit $i'' \geq i' \wedge j'' \leq j'$ ersetzt.

Abbildung 4.7 zeigt beispielhaft die Aktualisierung einer Pfadkante, nachdem eine neue Tracekante eingefügt worden ist.

Der Aufwand zum Aktualisieren des Trace nach dem Einfügen einer Kante vom Alignmentgraphen ist $O(k^2 \log n)$, da für alle k^2 Sequenzenpaare die nächste Pfadkante gesucht werden muss ($\log n$). Der Gesamtaufwand für die verbesserte Version von Greedy 2 im Worst Case ist $O(|E| \log |E| + |E_T| k^2 \log n + |E_T| \log n)$, wobei $E_T \subseteq E$ jene Kanten sind, die tatsächlich im Trace realisiert werden ($= O(k^2 n)$) und das anfängliche Sortieren aller Kanten in $O(|E| \log |E|$, das Einfügen aller realisierten Kanten in $|E_T| \log n)$ und das Aktualisieren des Traces nach dem Einfügen aller Kanten in $O(|E_T| k^2 \log n)$ liegen.

4.4 Auswertung

Dieses Kapitel soll einen Überblick über die Qualität der erhaltenen Ergebnisse geben.
Es hat sich gezeigt, dass Greedy 2 dem Algorithmus Greedy 1 in der Praxis sowohl von der Geschwindigkeit als auch von der Qualiät überlegen ist.

Je nach gewählter Kantengewichtung und Instanz unterscheidet sich jedoch die Qualitätsdifferenz zwischen den beiden Algorithmen. Benutzt man Gewichtungsverfahren, die an allen Kanten in einem Sequenzpaar das gleiche Gewicht vergeben, liegen beide Algorithmen fast auf gleichem Niveau, werden an alle Kanten individuelle Gewichte verteilt, steigt die Lösungsqualität bei Greedy 2 stärker als bei Greedy 1. Tabelle 4.1 zeigt die erhaltenen Resultate von Greedy 1 und Tabelle 4.2 jene von Greedy 2, wenn als Kantengewichtung das Verfahren aus 3.2.2 verwendet wird. Wie bereits bei anderen experimentellen Auswertungen wird auch hier BAliBase 1.0 verwendet.

	$\sum w_i$	SP		annSP	
Referenz	MW	MW	STABW	MW	STABW
ref 1	2066.162	0,762	0.213	0,833	0.188
ref 2	12336.962	0,717	0.175	0,778	0.201
ref 3	15513.999	0,486	0.276	0,659	0.316
ref 4	3847.732	0,552	0.195	0,810	0.188
ref 5	643.560	0,716	0.230	0,842	0.226
Summe	9220.178	0,705	0.226	0,778	0.216

Tabelle 4.1: Greedy1-Alg. mit ClustalW-Score

	$\sum w_i$	SP		annSP	
Referenz	MW	MW	STABW	MW	STABW
ref 1	2053.109	0.743	0.209	0,819	0.185
ref 2	12033.059	0.725	0.146	0,791	0.167
ref 3	15537.002	0.575	0.231	0,665	0.258
ref 4	4307.228	0.576	0.160	0,726	0.139
ref 5	632.210	0,727	0.231	0,848	0.231
Summe	9368.796	0,706	0.207	0,794	0.190

Tabelle 4.2: Greedy2-Alg. mit ClustalW-Score

Die Tabellen 4.3 und 4.4 stellen hingegen die Ergebnisse dar, welche man mit Hilfe der Bewertung aus Kapitel 3.2.5 erhält. Die SP- und annSP können dabei direkt mit den vorigen Ergebnissen verglichen werden, das Gesamtgewicht der Kanten jedoch nicht, da hier eine andere Gewichtungsfunktion zugrunde liegt. Das Diagramm in Abbildung 4.8 zeigt die SP-Scores graphisch aufbereitet.

	$\sum w_i$	SP		annSP	
Referenz	MW	MW	STABW	MW	STABW
ref 1	2323.936	0,771	0.211	0,842	0.186
ref 2	13894.789	0,737	0.244	0,800	0.278
ref 3	14204.388	0,426	0.279	0,466	0.269
ref 4	4694.775	0,565	0.204	0,709	0.184
ref 5	516.971	0,737	0.227	0,828	0.218
Summe	10168.056	0,711	0.237	0,788	0.224

Tabelle 4.3: Greedy1-Alg. mit Fenster-Bewertung

	$\sum w_i$	SP		annSP	
Referenz	MW	MW	STABW	MW	STABW
ref 1	2458.062	0,780	0.194	0,862	0.150
ref 2	13253.401	0,780	0.142	0,844	0.159
ref 3	14468.606	0,643	0.265	0,723	0.265
ref 4	5151.906	0,616	0.135	0,775	0.122
ref 5	508.518	0,795	0.211	0,924	0.184
Summe	10743.211	0,752	0.197	0,843	0.167

Tabelle 4.4: Greedy2-Alg. mit Fenster-Bewertung

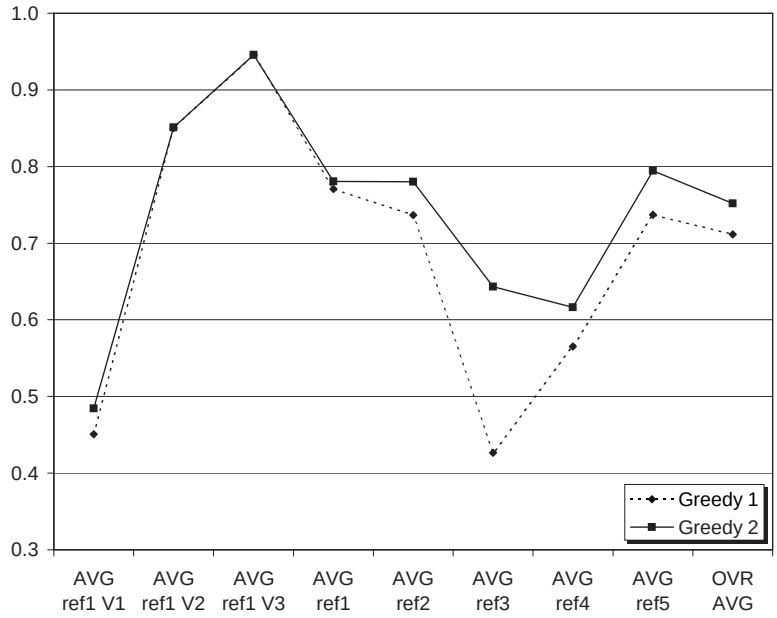

Abbildung 4.8: Diagramm: Greedy-Alg. mit Fensterbewertung

Kapitel 5

Lokale Verbesserungsheuristiken

In diesem Kapitel werden wir genauer auf die implementierten lokalen Verbesserungsheuristiken eingehen. MoveGap und MoveBlock versuchen beide durch das Verschieben von Symbolen in einem bestehenden multiplen Alignment weitere Verbesserungen zu erzielen. InsertEdge hingegen arbeitet direkt mit der Trace-Datenstruktur und versucht weitere Kanten zum Lösungstrace hinzuzufügen mit dem Ziel das Gesamtgewicht aller realisierten Kanten zu erhöhen.

5.1 MoveGap

All jene Symbole, die an einer Lücke angrenzen bzw. alle Lücken werden in dieser Heuristik bearbeitet. Der Algorithmus geht dabei Spalte für Spalte durch das Alignment (in einer zufälligen Permutation der Spalten). Trifft er in einer Spalte auf ein Symbol, das an eine Lücke grenzt, so versucht er die Lücke und das entsprechende Symbol zu vertauschen. Darauf wird das Alignment neu bewertet. Ergibt sich eine Verbesserung, so wird diese registriert, jedoch noch nicht durchgeführt. Die Heuristik versucht nämlich ein möglichst gutes lokales Optimum zu finden und deshalb kann es vorteilhaft sein, die endgültige Entscheidung über die Verbesserung nach hinten zu verschieben. Erst wenn alle möglichen Verbesserungen in einer Spalte bekannt sind, wird jene mit dem höchsten Gewinn durchgeführt. Darauf muss die Spalte von neuem untersucht werden, da sich durch das Verschieben von einem Symbol neue Verbesserungsmöglichkeiten eröffnen können. Nach (meist) wenigen Iterationen lässt sich in einer Spalte keine Verbesserung mehr erwirken und der Algorithmus schreitet zur nächsten Spalte in der Permutation fort.

Trifft der Algorithmus hingegen direkt auf eine Lücke, so versucht er diese sowohl mit dem rechts stehenden, als auch mit dem links stehenden Symbol

zu vertauschen. Eventuelle Verbesserungen werden auch hier registriert.

Um weitere Verbesserungen zu ermöglichen wurde diese Heuristik noch erweitert. In ihrer erweiterten Form kann ein Symbol über mehr als eine Lücke gezogen werden. Dadurch kann im Allgemeinen ein größerer Suchraum abgedeckt werden. Natürlich steigt mit dieser Modifikation der Heuristik auch deren Aufwand. Die Abbildung 5.1 zeigt eine Beispiel-Anwendung dieser Heuristik.

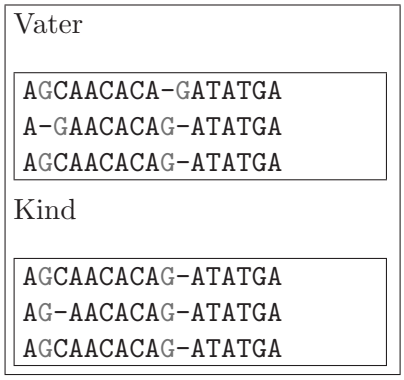

Abbildung 5.1: Beispiel für MoveGap

5.2 MoveBlock

MoveBlock funktioniert ähnlich wie das oben beschriebene MoveGap, nur ist es mit Hilfe von dieser Heuristik möglich, ganze Blöcke, also Teilsequenzen über eine Lücke zu schieben. Auch hier soll ein Beispiel (Abbildung 5.2) die Vorgehensweise verdeutlichen.

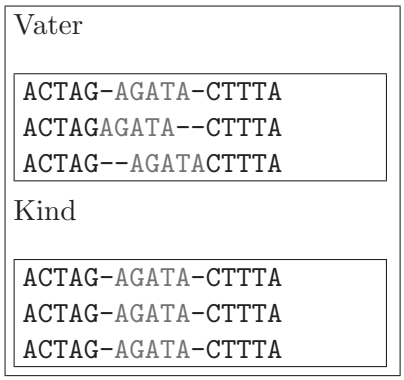

Abbildung 5.2: Beispiel für MoveBlock

Da dieser Algorithmus vom Rechenaufwand um einiges höher als der MoveGap-Algorithmus ist, wird hier darauf verzichtet, mehrere mögliche Verschiebungen durchzurechnen um schließlich die beste zu wählen. In der

Implementierung von dieser Methode wird jede erzielte Verbesserung sofort in die Lösung übernommen. Auch hier wurde aber ähnlich wie in der MoveGap-Heuristik die Möglichkeit implementiert, einen Block über mehrere Lücken zu ziehen. Dadurch lassen sich weitere Verbesserungen erzielen, auch wenn sich in Praxistests zeigt, dass eher selten mehrere Lücken hintereinander folgen und in diesen Fällen die Verschiebung über mehrere Lücken (wegen den großen Änderungen im Alignment) beinahe nie erfolgreich ist. Eine ähnliche Vorgehensweise (Block Shuffling) wurde bereits von Notredame et al. in [20] verwendet.

5.3 InsertEdge

Dieser Algorithmus versucht eine noch nicht realisierte Kante der paarweisen Alignments in den Trace einzubauen, welcher die bisherige Lösung repräsentiert, um so die Qualität der Lösung zu steigern. Dazu müssen im Allgemeinen jedoch andere Kanten entfernt werden um ein gültiges Alignment zu erzeugen. Laut Kapitel 1.3 ist ein erweiterter Alignmentgraph, also ein Graph welcher auch Arcs enthält, genau dann gültig, wenn alle im Graphen vorkommenden Kreise keine Arcs enthalten. Wird nun eine Kante hinzugefügt, können jedoch neue Kreise entstehen, welche über Arcs führen und so die Lösung ungültig machen.

Sei $e = (s, t)$ jene Kante, die eingefügt werden soll. Ein Kreis kann genau dann entstehen, wenn bereits vor dem Einfügen der neuen Kante e ein Pfad von s nach t bzw. von t nach s existiert. Um dies festzustellen wird ein minimaler s-t Schnitt (*minimum s-t Cut*) berechnet.

Definition 5.1 (minimum s-t Cut) *Ein minimum s-t Cut partitioniert die Knoten eines Graphen so in zwei Komponenten S und T, dass der Knoten s in S und der Knoten t in T liegt. Die Summe der Gewichte aller entfernten Kanten muss minimal sein.*

Damit kann einerseits ein evtl. vorhandener Pfad ermittelt werden, andererseits kann damit auch genau jene Kantenmenge minimalen Gewichtes ermittelt werden, welche aus dem Graph gelöscht werden muss, sodass kein Pfad mehr von s nach t oder umgekehrt existiert. Um dies zu bewerkstelligen wird der minimum s-t Cut auf ein maximales Flussproblem zurückgeführt. Der Algorithmus, der diese Funktion implementiert, basiert auf einem Ansatz von Goldberg und Tarjan, der unter anderem in [11] beschrieben wird. Daraus lässt sich nun leicht der minimum s-t-Cut ableiten. Details zum Erstellen eines minimum s-t-Cuts aus einem maximalen Fluss werden in [15] beschrieben. Nachdem die Kantenmenge des minimum s-t-Cuts aus dem Ergebnistrace entfernt wurden, kann die neue Kante gefahrlos in die Lösung aufgenommen werden. Meistens können nun noch weitere Kanten eingefügt

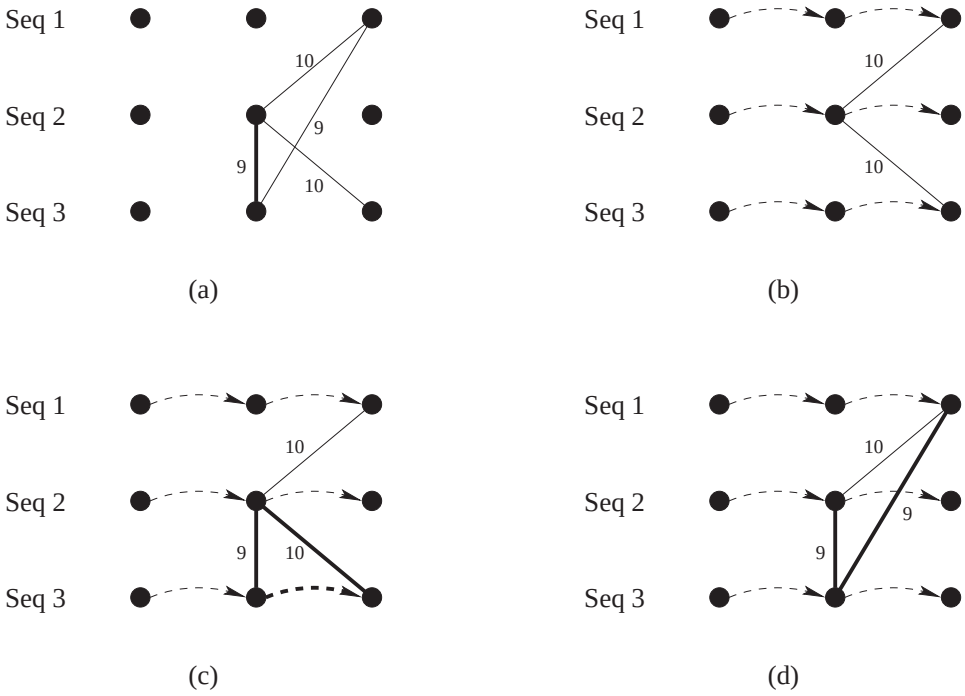

Abbildung 5.3: Anwendung von InsertEdge
(a) Ausschnitt eines Alignmentgraphen, der paarweise Alignments repräsentieren; die fett markierte Kante soll in den Lösungstrace eingefügt werden
(b) Der Trace der die momentane Lösung vor der InsertEdge-Verbesserung zeigt (alle Arcs haben ein Gewicht von ∞.
(c) Trace nach Einfügen der neuen Kante $\rightarrow$ ein Kreis mit einem Arc ist entstanden und der Trace ist dadurch ungültig geworden.
(d) Der minimale s-t Schnitt entfernt eine Kante und dadurch kann neben der gerade eingefügten Kante eine weitere eingefügt werden.
Durch InsertEdge ließ sich das Gesamtgewicht von 20 auf 28 steigern.

werden. Deshalb bestimmt der Algorithmus die Knotenmenge, die alle adjazenten Knoten der entfernten Kanten enthält. Darauf werden alle zu diesen Knoten adjazenten Kanten im paarweisen Alignmentgraphen bestimmt und als Kandidaten für eine Einfügeoperation markiert. Wir bezeichnen diese Kantenmenge mit K. Für jede Kante $k = (i, j) \in K$ wird mit Hilfe eines Tiefensuchealgorithmus ähnlich dem Verfahren aus Greedy 2 ein Pfad von i nach j oder von j nach i gesucht. Wird ein Pfad gefunden, welcher mindestens einen Arc enthält, kann die Kante *nicht* eingefügt werden, da ansonsten ein Kreis entstehen würde. Wird kein solcher Pfad gefunden, wird die Kante k eingefügt. Um möglichst Kanten mit hohen Gewichten einzufügen wird die Menge K vor den Einfügeversuchen absteigend nach Gewichten sortiert. Somit wird versucht, zuerst die am vielversprechendsten Kanten einzufügen.

Der eben beschriebene Algorithmus wird nun iterativ angewandt. Dazu werden zuerst alle noch nicht realisierten Kanten bestimmt und absteigend nach Gewicht sortiert. Darauf wird sowohl der Verlust am Kantengesamtgewicht durch das Entfernen von Kanten, als auch der Gewinn durch das Hinzufügen von neuen Kanten bestimmt. Ist die Differenz Gewinn−Verlust positiv, dann wird die Veränderung übernommen, ansonsten wird sie verworfen und mit der nächsten Kante fortgefahren. Dies wird so lange wiederholt, bis keine Kanten mehr eingefügt werden können oder bis eine zuvor festgelegte Anzahl an Iterationen erreicht wurde.

5.4 Verbesserung von InsertEdge

Um den InsertEdge-Algorithmus zu beschleunigen, wurde er um eine heuristische Funktion erweitert. Falls bereits nach dem minimalen s-t Schnitt abzusehen ist, dass durch die aktuell betrachtete Kante keine Verbesserung mehr zu erreichen ist, wird die gerade laufende Iteration abgebrochen und mit der nächsten Kante fortgefahren. Dadurch muss nicht die Menge jener Kanten berechnet werden, die evtl. noch hinzugefügt werden könnten. Da dies der rechenintensivste Teil des InsertEdge-Improvements ist, kann die Laufzeit vor allem bei großen Eingabeinstanzen beträchtlich verbessert werden. Hierbei ist es wichtig, dass durch die Schranke, welche bestimmt, ob eine Kante weiter bearbeitet werden soll oder nicht, die Lösungsqualität keine signifikanten Einschränkungen erfährt. Wir definieren diese Schranke wie folgt:
Sei x die Anzahl an Kanten, welche beim Einfügen der aktuell untersuchten Kante entfernt werden müssten. Überschreitet x einen vorher festgelegte Schranke k, so wird die momentane Iteration abgebrochen, ansonsten wird mit der Berechnung fortgefahren.
Diese Einschränkung ist deshalb sinnvoll, da die Laufzeit zur Berechnung der zusätzlich einzufügenden Kanten linear von der Anzahl der zu löschenden Kanten abhängt. Genauer: Der lineare Proportionalitätsfaktor entspricht

$2 \cdot \overline{deg} \cdot nr$. nr bezeichnet dabei die Anzahl der zu löschenden Kanten und $\overline{deg}$ ist der durchschnittliche Grad der Knoten im Trace. Dies kommt dadurch zustande, da alle adjazenten Kanten der inzidenten Knoten einer gelöschten Kante untersucht werden müssen.

Nun gilt weiters, dass die Wahrscheinlichkeit eine bessere Lösung zu erzeugen sinkt, wenn viele Kanten gelöscht werden müssen. Tests zeigen, dass sich durch diese Maßnahme (bei großen Instanzen) die Laufzeit eines Optimierungsvorganges halbiert.

Die Anzahl der zu löschenden Kanten x ist jedoch erst nach der Berechnung des minimalen s-t Schnittes bekannt. Eine weitere (kleine) Beschleunigung ließ sich dadurch erreichen, dass unter bestimmten Umständen bereits vor der Berechnung des minimalen s-t Schnittes eine ungünstige Iteration abgebrochen wird. Überschreitet schon der maximale Fluss eine bestimmte Schranke kann auch vor dem minimum Cut abgebrochen werden. Dazu ist jedoch eine Abschätzung nötig. Der maximale Fluss gibt an, wie viel insgesamt an Kantengewicht verloren geht. Dies muss immer in Relation zu der eingefügten Kante gebracht werden. Somit wird die Abbruchbedingung wie folgt formuliert:

$$\text{if } \left(\frac{maxflow}{edgeweight} > k \right) \text{ then Abbruch ,}$$

wobei k ein zuvor festgelegter Schwellenwert, $maxflow$ der maximale Fluss und $edgeweight$ das Gewicht jener Kante ist, die eingefügt werden soll. Durch diese Maßnahme entfällt bei besonders hohen Kantengewichtsverlusten die Berechnung der s-Komponente des maximalen Flusses und die Bestimmung der abgeschnittenen Kanten.

5.5 Auswertung

Alle drei lokalen Verbesserungsheuristiken wurden ausführlichen Tests unterzogen, deren Ergebnisse hier kurz zusammengefasst werden sollen. Die Ausgangslösung für die zu testenden Heuristiken wurde mit Greedy 2 erzeugt und die Kantengewichte wurden mit dem Algorithmus aus Unterkapitel 3.2.5 („Bewertung mittels durch Lücken beschränkter Fenstergröße“) berechnet. Die MoveGap-Heuristik sowie auch die MoveBlock-Heuristik wurden in ihrer erweiterten Form ausgeführt, für InsertEdge wurden folgende Parameter gewählt:

- maximal 100 Kanten werden eingefügt (wobei Kanten, die nach dem Einfügen der gerade betrachteten Kanten zusätzlich dazukommen können, nicht beachtet werden)
- sind mehr als 30 Kanten in einer Iteration zu entfernen, wird die Iteration abgebrochen

- ergibt der Quotient maximaler Fluss/aktuelles Kantengewicht einen Wert höher als k=35, wird die Iteration ebenfalls abgebrochen und keine Kante hinzugefügt.

Diese Werte haben sich aufgrund von experimentellen Tests als geeignet herausgestellt. Tabelle 5.1 zeigt die Ergebnisse der Greedy 2 Heuristik ohne lokale Verbesserungen sowie die Resultate mit MoveGap. Tabelle 5.2 zeigt

Referenz	Greedy 2		Greedy 2 + MoveGap	
	SP	annSP	SP	annSP
AVG ref1	0,780	0,862	0,788	0,871
AVG ref2	0,780	0,844	0,778	0,823
AVG ref3	0,643	0,723	0,638	0,759
AVG ref4	0,616	0,775	0,619	0,762
AVG ref5	0,795	0,924	0,796	0,935
OVR AVG	0,752	0,843	0,756	0,847

Tabelle 5.1: Greedy 2 mit und ohne Anwendung von MoveGap

die BAliBase-Scores, die wir durch die Anwendung von MoveBlock und InsertEdge mit den oben genannten Parametern erhalten haben.

Referenz	Greedy 2 + MoveBlock		Greedy 2 + InsertEdge	
	SP	annSP	SP	annSP
AVG ref1	0,777	0,860	0,790	0,871
AVG ref2	0,776	0,839	0,786	0,849
AVG ref3	0,639	0,718	0,661	0,750
AVG ref4	0,613	0,771	0,624	0,783
AVG ref5	0,792	0,924	0,801	0,928
OVR AVG	0,749	0,841	0,761	0,852

Tabelle 5.2: Greedy 2 mit MoveBlock oder InsertEdge

Vor allem InsertEdge, aber auch MoveGap bringen entscheidende Verbesserungen, MoveBlock jedoch scheint nicht sehr gut zu funktionieren. Abhängig von der Methode, welche die Kantenbewertung durchführt, werden die Ergbnisse teils besser, teils sogar schlechter. MoveBlock scheint zwar das Gesamtgewicht zu optimieren, nähert sich dabei jedoch nicht dem optimalen Alignment.

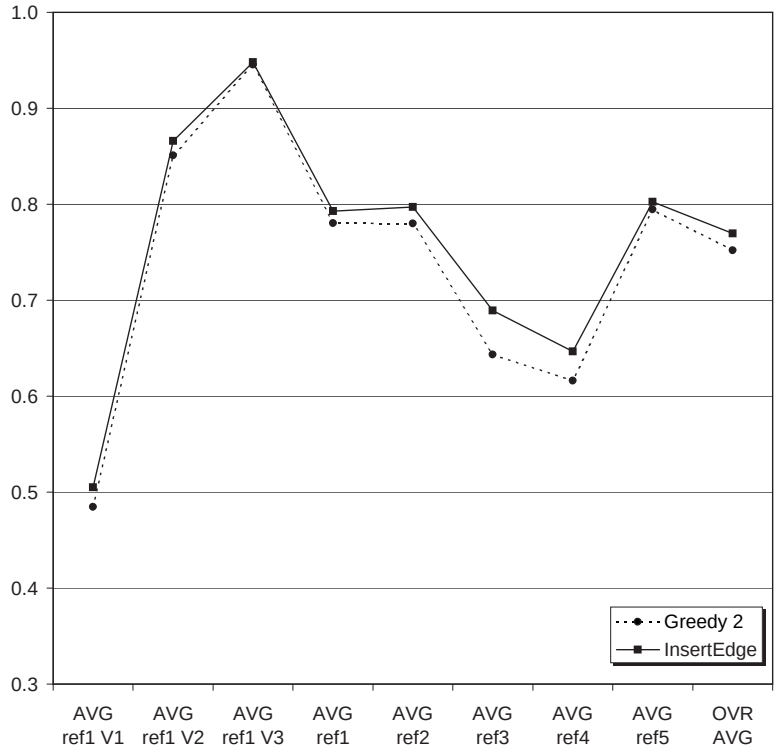

Abbildung 5.4: Diagramm: Greedy2 mit und ohne InsertEdge

Kapitel 6

Tabu Search

6.1 Einführung

Bei Tabu Search handelt es sich um eine sog. *Metaheuristik*, also einer Heuristikdie auf lokale Suche aufbaut. Ziel von Tabu Search ist das Vermeiden des Hängenbleibens bei lokal optimalen Lösungen und von Zyklen bei der Lösungssuche, indem durch Verbieten oder Bestrafen von bestimmten Lösungswegen verhindert wird, dass in einer folgenden Iteration idente Lösungen durchmustert werden.

Eine Datenstruktur (*Tabu Search Memory*) speichert die letzten k Schritte der Metaheuristik, um damit den Suchprozess im Lösungsraum zu steuern. Dieser Speicher wird normalerweise durch eine oder mehrere *Tabu-Listen* realisiert. Hauptziel dieser Listen ist die Vermeidung von Zyklen, die unter Umständen entstehen können, wenn eine durchgeführte Änderung an der momentanen Lösung in den nächsten Zyklen wieder rückgängig gemacht wird und somit eine Schleife entsteht, welche eine Änderung vornimmt und gleich darauf wieder verwirft. Glover hat auf diesem Gebiet bedeutende Entwicklungsarbeit geleistet. Eine gute Einführung findet man in [7] und [6], zwei von seinen zahlreichen Publikationen.

6.2 Der Algorithmus

Basierend auf die InsertEdge-Heuristik aus Kapitel 5.3 haben wir eine Tabu-Suche implementiert. Damit sollen die Ergebnisse weiter verbessert werden und im Gegensatz zu den bisher vorgestellten Verbesserungsheuristiken soll der Algorithmus auch lokalen Optima entfliehen können. Durch die Notwendigkeit von mehr Iterationen (da der Suchraum größer wird) steigt der Berechnungsaufwand erheblich, sodass die Berechnung teilweise deutlich mehr Zeit kostet.
Der Tabu-Search-Ansatz durchläuft sieben Phasen, die in Abbildung 6.1 aufgezählt werden.

1. Menge der nicht realisierten Kanten berechnen.
2. Gewinn g durch Einfügen der Kante mit höchstem Gewicht in den Trace berechnen, wobei die Gültigkeit des Traces erhalten bleibt.
3. Falls $g >$ Schranke s, Kante einfügen, sonst gibt Kante in Tabu-Liste NotInsert und gehe zu 6.
4. Alle eingefügten Kanten in Tabu-Liste NotDelete geben.
5. Alle gelöschten Kanten in Tabu-Liste NotInsert geben.
6. Kanten, welche seit mehr als k Iterationen in den Tabu-Listen sind, aus diesen entfernen.
7. Falls weniger als i Iterationen gemacht wurden, gehe zu 1, sonst Ende.

Abbildung 6.1: Tabu-Search Hauptphasen

Zu Beginn des Tabu-Search-Algorithmus muss die Menge der möglichen Kanten berechnet werden. Das sind alle Kanten aus dem Alignmentgraphen, welche nicht im momentanen Lösungstrace realisiert sind. Diese werden in einem binären Suchbaum abgespeichert. In der zweiten Phase wird nun jene Kante mit dem maximalen Gewicht aus den möglichen Kanten bestimmt und mit einem leicht modifizierten InsertEdge-Algorithmus wird darauf der Gewinn g berechnet, welcher erreicht werden würde, falls die Kante im Trace eingefügt wird und daraus wiederum eine gültige Lösung erzeugt wird. Dazu müssen im Allgemeinen, ähnlich wie in InsertEdge, bestimmte Kanten entfernt werden, und häufig können weitere Kanten hinzugefügt werden. Ist nun g größer als eine zuvor festgelegte Schranke s, wird die Veränderung beibehalten, ansonsten wird sie verworfen. Für s eignet sich ein betragsmäßig kleiner negativer Wert, da dadurch ein lokales Optimum verlassen werden kann, die Lösung jedoch nicht viel schlechter werden kann. Eine andere Möglichkeit für s wäre, diese Schranke im Laufe des Algorithmus dynamisch zu verändern und zwar in dem Sinne, dass Anfangs größere Verschlechterungen akzeptiert werden, während am Ende nur mehr geringfügige Verschlechterungen in Kauf genommen werden. Damit kann zu Beginn relativ leicht ein lokales Optimum verlassen werden und zum Schluss das beste Alignment im neuen betrachteten Teil des Suchraumes gefunden werden (*hill climing*).

Wurde die Kante nicht eingefügt, dann wird sie der Tabu-Liste NotInsert hinzugefügt, welche dafür Sorge trägt, dass für die nächsten k Iterationen nicht mehr versucht wird, diese Kante einzufügen. Wurde die Kante jedoch in die Lösung eingebaut, werden die insgesamt zwei Tabu-Listen wie folgt

aktualisiert: Alle Kanten, die eingefügt worden sind, sollen in den nächsten Iterationen nicht gleich wieder entfernt werden, also geben wir diese Kanten in die Tabu-Liste NotDelete. Alle entfernten Kanten sollen in naher Zukunft nicht mehr eingefügt werden, da ansonsten Zyklen entstehen könnten und die gerade erreichte Änderung wieder rückgängig gemacht würde. Deshalb wird diese Kantenmenge in die Tabu-Liste NotInsert gegeben. Da die Kanten in den beiden Tabu-Listen jedoch nur für eine bestimmte Zeit gesperrt werden sollen, werden am Ende jeder Iteration alle Kanten, welche bereits länger als k Iterationen in einer Tabu-Liste vorhanden sind, aus dieser entfernt. Schließlich muss noch der binäre Suchbaum aktualisiert werden. Alle Kanten, welche in die Lösung eingefügt werden konnten, müssen aus dem Baum entfernt werden. Außerdem müssen alle Kanten, welche aus der Tabu-Liste NotInsert entfernt wurden, wieder in den binären Baum eingefügt werden, da sie ja wieder gültige Kandidaten für eine Einfügeoperation darstellen. Damit endet eine Itertion. Falls bereits i Iterationen durchgeführt worden, sind, bricht der Algorithmus ab, ansonsten startet eine neue Iteration. Wird in einer Iteration eine neue, beste Lösung gefunden, wird diese abgespeichert. Somit kann am Ende der Tabu-Suche der beste berechnete Trace als Endlösung realisiert werden.

6.3 Auswertung

Kapitel 7

Implementierung

7.1 Benutzte Werkzeuge

Alle Algorithmen wurden für SuSE Linux 8.2 in C++ entwickelt. Als Compiler kommt *g++ 3.2* aus der GNU Compiler Collection zum Einsatz. Die Bibliothek *EAlib 1.1* wird für die Verwaltung der Paramter sowie zur Erzeugung von Zufallszahlen benutzt. Eine zentrale Rolle spielt auch das Programmpaket *LEDA 4.4* von Algorithmic Solutions. Es stellt eine Vielfalt von Datenstrukturen wie Listen, Mengen, Sorted Sequences und Hash-Arrays, sowie geeignete Strukturen zur Darstellung von Graphen zur Verfügung. Außerdem gibt es generische Methoden für viele grundlegende Graphenalgorithmen, unter anderem Minimum Cut, Maximal Flow und das Erkennen von (starken) Zusammenhangskomponenten. Weitere Informationen zu LEDA findet man unter [15]. Die paarweisen Alignments, welche die Berechnungsgrundlage unserer Algorithmen darstellen, werden mit Hilfe einer angepassten Version von *ClustalW 1.82* erzeugt. Zur Dokumentation des Source-Codes verwendeten wir Doxygen.

7.2 Klassen und Module

Abbildung 7.1 stellt den prinzipiellen Ablauf des Programmes dar. Alle Funktionen bzw. Datenstrukturen wurden in Klassen gekapselt, die in Abbildung 7.2 als UML-Diagramm dargestellt werden. Tabelle 7.1 hingegen zeigt einen Überblick der zentralen Klassen und Module und deren primäre Funktion.

Wir wollen nun diese Klassen genauer betrachten:

- `Sequence`: Diese Klasse kapselt eine einzelne Sequenz (ohne Lücken), welche in einem String gespeichert wird. Es existieren Operationen für einen einfachen Zugriff auf einzelne Zeichen, sowie zum Ausgeben einer Sequenz.

Klasse/Modul	Verwendung
Sequence	Speichert eine zu alinierende Sequenz
MultiSequences	Verwaltet alle zu alinierenden Sequenzen
ClustalW	Interface zu ClustalW
AGEdge	Stellt eine einzelne Kante im Alignmentgraphen dar
AGEdgeList	Verwaltet alle Kanten im Alignmentgraphen
CLWScoringMatrix	Speichert die gewählte ClustalW-Bewertungsmatrix ab
ScoringMatrix	Stellt Methoden für den Zugriff auf die ClustalW-Bewertungsmatrizen zur Verfügung
AlignmentGraph	Repräsentiert den eigentlichen Alignmentgraphen
MATraceEntry	Ein (Kanten-)Eintrag in einem Trace
MATrace	Trace, der eine mögliche Lösung repräsentiert
MATracePathSearch	Diese Klasse ermöglicht u.a. das Überprüfen der Gültigkeit einer Kante
MultiAlignment	Entgültige Lösung aus einem Trace
Greedy1	Greedy 1 Algorithmus
Greedy2	Greedy 2 Algorithmus
LocalImprovements	Enthält Methoden der lokalen Verbesserungsheuristiken
TabuSearch	Enthält den Tabu-Search Algorithmus

Tabelle 7.1: Auflistung der zentralen Klassen/Module

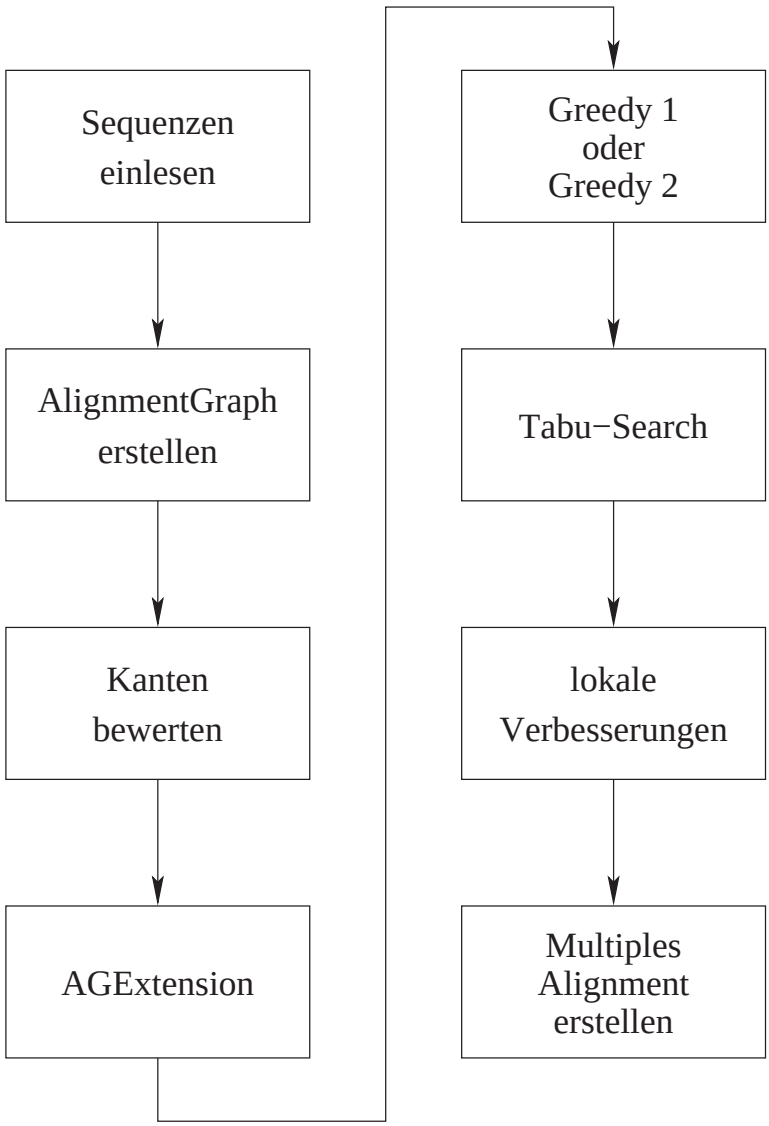

Abbildung 7.1: Abblauf des Programmes

- `MultiSequences`: Diese Klasse speichert ein Feld von Sequenzen (jeweils vom Typ `Sequence`), sowie die Anzahl der Sequenzen und deren maximale Länge. Es werden Funktionen zum Zugriff, Ausgeben und Laden der Sequenzen angeboten.

- `ClustalW`: Diese Klasse stellt ein komfortables C++-Interface zu ClustalW her. Damit kann auf einfache Weise die von ClustalW benutzte Bewertungsmatrix ermittelt werden, mit Hilfe einer weiteren Methode können die paarweisen Alignments erstellt werden oder es kann ein vollständiges ClustalW-Multialignment erstellt werden.

- `AGEdge`: Ein Objekt von dieser Klasse rapräsentiert eine Kante im Alignmentgraphen. Es wird für beide Knoten der Kante die Sequenz und die Position abgespeichert. Weiters kann das Gewicht der Kante festgelegt werden.

- `AGEdgeList`: Hier werden alle Kanten des AlignmentGraphen in einer (doppelt verketteten) Liste abgespeichert. Es wurde eine Operation zum effizienten Anlegen neuer Kanten, sowie ein Iterator implementiert.

- `CLWScoringMatrix`: Ein Objekt dieser Klasse enthält einen Verweis auf die von ClustalW benutzte Bewertungsmatrix (ein zweidimensionales Feld, welches für jede Kombination zweier Symbole einen Score speichert), sowie das darin betragsmäßig größte positive und negative Element.

- `ScoringMatrix`: Diese Klasse kapselt eine Klasse vom Typ `CLWScoringMatrix` und stellt einige zusätzliche Methoden, wie zum Beispiel zum Setzen einer neuen Matrix zur Verfügung. Außerdem wird in dieser Klasse die Notredame- und ClustalW-Bewertung für jedes Sequenzenpaar abgespeichert.

- `AlignmentGraph`: Diese Klasse wird zur Verwaltung eines Alignmentgraphen eingesetzt. Eine Instanz dieser Klasse enthält Referenzen zu einem `MultiSequences`-Objekt, dessen Sequenzen er darstellen soll, sowie zu einer `AGEdgeList`, welche alle Kanten des Graphen speichert. Weiters gibt es noch ein Feld, welches alle Knoten des Graphen abspeichert und eine Referenz zu einem `ScoringMatrix`-Objekt, welches unter anderem die benutzte Bewertungsmatrix enthält. Es werden Methoden zur Verfügung gestellt, mit deren Hilfe neue Kanten hinzugefügt oder bereits existierende Kanten zurückgegeben werden können. Es wurden auch Methoden implementiert, welche die Kantenbewertung und die AlignmentGraph-Extension durchführen. Schließlich gibt es noch Funktionen zum Speichern, Laden und Ausgeben des

Alignmentgraphen. Der Alignmentgraph wurde als zweidimensionales Feld von AGNodes implementiert. Ein `AGNode` stellt ein konkretes Symbol in einer Sequenz dar und enthält eine Liste von allen adjazenten `AGEdges`. Diese Liste wurde die mit Hilfe einer sog. *sorted sequence* implementiert, welche widerum mit *skip lists* realisiert wurde.

- `MATraceEntry`: Ein Objekt dieser Klasse stellt eine Kante in einem Trace dar. Es gibt insgesamt vier Kantentypen: EDGE (normale Alignmentgraph-Kanten), DERDEDGE (abgeleitete Kanten, also Kanten welche zwei Knoten verbinden, zwischen denen ein Pfad ohne Arcs existiert), PATH (eine Kante, welche zwei Knoten verbindet, die durch einen Pfad mit mindestens einem Arc verbunden sind) und REVPATH (d.h. ein PATH-Eintrag existiert vom Zielknoten zum Startknoten).

- `MATrace`: Diese Klasse rapräsentiert einen Trace, also einen Graphen, der eine gültige Lösung enthält. Zu jedem Trace wird ein `AlignmentGraph` und ein `Multisequences`-Objekt assoziiert. Es existieren Funktionen zum Einfügen oder Löschen von Kanten, die Objekte des Typs `MATraceEntry` sind. Außerdem gibt es Methoden, welche überprüfen, ob eine Kante eingefügt werden kann, ohne den Trace ungültig zu machen, sowie die Möglichkeit erweiterte Kantenypen (abgeleitete Kanten, Pfadkanten) korrekt einzufügen. Diese Datenstruktur wurde folgendermaßen implementiert: Ein zweidimensionales Feld von *sorted sequences* bietet Speicherplatz für alle Kanten von je zwei verschiedene Sequenzen. Nach der Auswahl einer sorted sequence (i, j), welche die beiden Sequenzen der Kante $(s_{i,a}, s_{j,b})$ repräsentiert, welche eingefügt oder gesucht werden soll, kann der Index a der Kante innerhalb der ersten Sequenz als Schlüssel für den Zugriff auf die sorted sequence benutzt werden. Dort befindet sich ein Objekt `MATraceEntry` (bzw. dort kann ein neues `MATraceEntry`-Objekt eingefügt werden), welches den Typ der Kante und die Position b in der zweiten Sequenz enthält.

- `MultiAlignment`: Ein Objekt dieser Klasse speichert eine Multialignment, welches eine Lösung des MSA-Problems darstellt. Es enthält eine Referenz auf ein `MultiSequences`-Objekt und ein zweidimensionales Feld, das für jede Position in jedem Alignment entweder einen Wert für *Lücke* oder eine Position im `MultiSequences`-Objekt enthält. Dadurch lässt sich jedes multiples Alignment darstellen. Es gibt eine Reihe von Zugriffsmethoden, eine Funktion zum Löschen von Spalten, welche ausschließlich Lücken enthalten, sowie einige weitere nützliche Methoden.

- `Greedy1`: Diese Klasse kapselt den Greedy 1-Algorithmus. Zwei weitere

Hilfsklassen `SubGraph` und `Component` dienen dazu, die Front bzw. die einzelnen Komponenten und Superkomponenten darzustellen. Operationen sind das Hinzufügen von *front edges* und *exposed edges*, das Analysieren der Komponenten usw.

- `Greedy2`: Hier befindet sich die Funktion `greedy2Heuristic`, welche den Greedy 2-Algorithmus startet. Die eigentliche Funktionalität des Algorithmus, also jene Methoden, die den Trace verändern, befinden sich in der Klasse `MATracePath`.

- `MATracePath`: Diese Klasse entählt die eigentliche Funktionalität des Greedy 2 Algorithmus und Teile der InsertEdge-Heuristik. Mit Hilfe dieser Klasse ist es möglich, Pfade zwischen zwei Knoten zu ermitteln und dadurch festzustellen, ob eine Kante eingefügt werden kann, ohne die Gültigkeitsbedingungen eines Traces zu verletzen.

- `LocalImprovements`: Diese Klasse enthält alle Methoden, die für die lokalen Verbesserungen zuständig sind (also MoveBlock, MoveGap und InsertEdge). Natürlich sind auch Referenzen auf ein AlignmentGraph-Objekt, sowie auf ein Objekt, welches eine bestehende Lösung markiert nötig. Dies ist je nach Optimierungsfunktion ein `MATrace`-Objekt oder, falls eine Optimierungsmethode direkt auf einem multiplen Alignment operiert, ein `MultiAlignment`-Objekt. Schließlich gibt es noch eine Methode, die dem Tabu-Search-Algorithmus erlaubt, auf eine modifizierte InsertEdge-Heuristik zuzugreifen.

- `TabuSearch`: Diese Klasse enthält eine Methode zum Starten des TabuSearch-Algorithmus sowie die dazu benötigten Datenstrukturen. Diese sind zwei Listen, in welchen die gesperrten Kanten abgespeichert werden (also einmal jene Kanten, die aus dem Trace nicht gelöscht werden dürfen und einmal jene Kanten, die in den Trace nicht eingefügt werden dürfen), sowie ein binärer Suchbaum, der alle Kanten enthält, welche in den Trace eingefügt werden könnten, absteigend sortiert nach dem Gewicht.

7.3 Parameter

Beinahe alle Module aus Abbildung 7.1 können über Parameter gesteuert werden. Tabelle 7.2 gibt einen Überblick über die wichtigsten Parameter, welche in der Kommandozeile beim Ausführen des Programmes gesetzt werden können. Der Aufruf hat dabei folgende allgemeine Form:

```
./msea [param1 wert1] [param2 wert2] ...
```

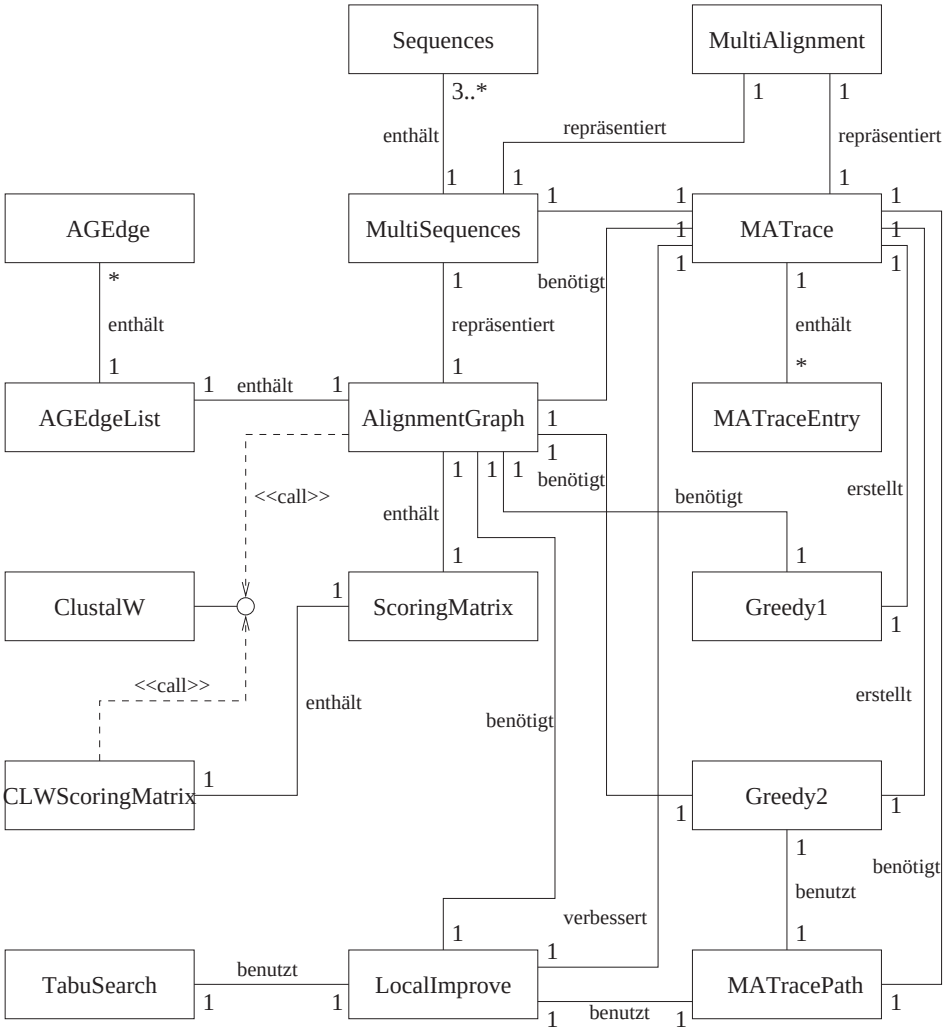

Abbildung 7.2: (Vereinfachtes) UML-Diagramm

Als Beispiel wollen wir zeigen wie der Algorithmus mit den Standardparametern gestartet wird um ein Alignment der Eingabesequenz `examplesequence.msf` zu starten:

```
./msea ifile examplesequence.msf
```

Das Programm akzeptiert als Ein- und Ausgabeformat alle Sequenzen, mit denen auch Clustalw 1.82 umgehen kann. Diese sind:

- 0: Clustal-Alignment
- 1: NBRF/PIR
- 2: GCG (Standard)
- 3: PHYLIP
- 4: GDE
- 5: NEXUS

Die Nummer gibt an, welchen Wert `aftype` annehmen muss, um ein Alignment im entsprechenden Format abzuspeichern.

Parameter	Verwendung
agfile	Dateiname zum Laden/Speichern eines Alignmentgraphen
agmode	Spezifiziert, ob ein Alignmentgraph geladen, gespeichert oder erstellt werden soll
anfile	Optionales Annotation-File zum Vergleichen mit einer optimalen Lösung
dfsk	Rekursionstiefe für AGExtension
edgew	Art der Kantenbewertung
clwfile	Dateiname für temporäre Dateien von ClustalW
local	Art der lokalen Verbesserungsheuristik, welche durchgeführt werden soll
aftype	Dateityp des abzuspeichernden Lösungsalignments
rfile	Name der Ergebnisdatei
ifile	Name der Eingabedatei

Tabelle 7.2: Parameter des Programmes

Kapitel 8

Gesamtauswertung

In diesem Kapitel wollen wir alle bisherigen Teilergebnisse und deren Auswertungen vereinen um somit die Gesamtqualität unserer Heuristiken beurteilen zu können. Außerdem werden die von uns implementierten Algorithmen mit anderen, bereits existierenden Programmen verglichen um die Qualität unserer Lösungen in Relation mit etablierten Projekten zu bringen.

8.1 Systemkonfiguration

Alle Testläufe wurden mit folgender Konfiguration durchgeführt:

System	32 Bit x86
Prozessor	P4 - 1.9 GHz
Hauptspeicher	1 GByte
Betriebssystem	SuSE Linux 8.1
Kernel	Version 2.4.19
Compiler	GNU gcc 3.2
Compileroptionen	-O4
ClustalW	Version 1.82
LEDA	Version 4.4.1
BAliBASE	Version 1.0

Tabelle 8.1: Systemkonfiguration

8.2 Auswertung

...

8.3 Vergleich mit bestehenden Programmen

In dieser Sektion wollen wir die Ergebnisse unserer Heuristiken mit drei anderen Projekten vergleichen:

- ClustalW (siehe Kapitel 2.3)
- Saga (siehe Kapitel 2.7)
- T-Coffee (siehe Kapitel 2.4)

ClustalW wurde in der Version 1.82 mit allen Standardparametern verwendet. Für Saga und TCoffee wurde das Paket SAGA V0.95 benutzt, das sowohl die Saga-, als auch die TCoffee-Algorithmen enthält. Auch hier wurden beinahe alle Default-Werte der Parameter benutzt. Folgende Außnahmen wurden gemacht:

- Der ClustalW-Path wurde gesetzt (Parameter `clustal_path`).
- Der Parameter `rooted_tree` wurde auf `MAKE:SIMILARITY_1:DEFAULT` gesetzt.
- Der Parameter `aln_lib` wurde auf `MAKE:CLUSTALW:DEFAULT` gesetzt.

Durch diese Maßnahme kann Saga/TCoffee auf ClustalW zurückgreifen um die phylogenetischen Bäume zu berechnen. Außerdem werden dadurch genauere und schnellere Berechnungen der Schranken möglich.

Unser Programm wurde ebenfalls mit den Standardparametern aufgerufen. Tabelle 8.2 zeigt deren Einstellung. Die Bedeutung der einzelnen Parameter wurden bereits in Tabelle 7.2 auf Seite 63 beschrieben.

Parameter	Wert	Bemerkung
agfile	Dateiname	Name des Ausgabe-Alignmentgraphen
agmode	2	Alignmentgraph wird erstellt und gespeichert
anfile	x	wird nicht benutzt
dfsk	2	Rekursionstiefe von 2 für AGExtension
edgew	7	Bewertung mit variabler Fenstergröße
clwfile	clw.dnd	
local	?7?	Tabu Search
aftype	2	GCG (*.msf) Datei
rfile	Ausgabedatei	
ifile	Eingabedatei	

Tabelle 8.2: Parameter des Programmes

Kapitel 9

Schlussbemerkungen

Anhang A

English Summary

This work presents an algorithm for solving an important problem in bioinformatics, the multiple sequence alignment problem. The goal of a multiple sequence alignment is to find similarities, in particular corresponding parts, between several amino acid or base sequences (i.e. between proteins or DNA, RNA).
By using already existing methods, enhancements of them and new elements, we have developed a new algorithm beeing capable to solve this NP-complete problem heuristically to near optimality. First of all the multiple sequence alignment problem is transformed into a graph theoretic model, the so-called *maximum weight trace formulation*, to which a greedy-algorithm is applied in order to generate a first solution. After that, tabu-search and a variety of local improvement methods are used to further improve the base solution.

Figure A.1 shows an example input of four short sequences and figure A.2 presents a possible alignment of them.

```
MPQILLLV
LRLL
MKILLL
MPPVLILV
```

Abbildung A.1: Sequences before alignment

```
MPQILLLV
MLR-LL--
M-KILLL-
MPPVLILV
```

Abbildung A.2: Aligned sequences

Chapter 1 starts with an introduction to the multiple sequence alignment problem (MSA). It defines the MSA-problem as follows:

Σ is a finite alphabet without space $'-'$ and $\Sigma' = \Sigma \cup \{'-'\}$. Let $s_1, \ldots, s_k$ be k sequences with length $l_1, \ldots, l_k$. A (global) multiple alignment A of $s_1, \ldots, s_l$ is a matrix of dimension $k \times l$ with the following properties:

- $max(l_1, ..., l_k) \leq l \leq \sum_{i=1}^{k} l_i$
- $A[i][j] \in \Sigma' \; \forall 1 \leq i \leq k, 1 \leq j \leq l$
- for each pair of symbols in a sequence $s_{k,i}, s_{k,j}$ with $i < j$ and their position $s_{k,i'}$ and $s_{k,j'}$ in the MSA, $i' < j'$ is valid, i.e. they maintain their relative order
- no column contains only spaces
- The number of sequences is $k \geq 2$ ($k = 2$ is a special case: the pairwise alignment)

An optimal alignment is a multiple alignment that minimizes an *objective function*. An often used objective function is the so-called *Sum of Pairs*(SPS)-function, which is defined in the following way:

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

where:

- s_{ih} represents the h^{th} symbol in the i^{th} sequence;
- $c : \Sigma' \times \Sigma' \to \mathbb{R}$ is an objective function for pairs of symbols
- c(-,-)=0 holds.

The MSA-problem with the SPS-objective function is NP-hard.

In chapter 2 we summarize some of the most important work done in this field. We used some concepts or ideas from those papers, like the *alignment graph* or the *trace*-data structure in our work.

Chapter 3 introduces the *alignment graph*. We use *ClustalW* (see [29]) to produce optimal pairwise alignments for all sequences. Each symbol is represented as a node in a graph. If two symbols are aligned in one of the pairwise alignments, we insert an edge in the alignment graph, connecting those two symbols. By using the *alignment graph extension*, an enhancement of the library extension described in [20], we add further edges into the alignment graph, which describe the transitive relation between two or more edges. This chapter puts also some attention to the weight of the edges in the graph. The weight of an edge describes, how "good" the alignment of the two symbols is. Equal symbols should get a high weight, whereas two

symbols that represent two absolutely different amino acids should get a very low weight. There are several possibilities to assign a weight to each edge. It has been shown, that using the scoring matrix contained in ClustalW together with a function that considers also the neighborhood of the alignment edge, produces good results, when used in combination with our algorithms. We construct a valid solution by building a *trace*, i.e. an alignment graph representing a feasible alignment. The goal of the implemented algorithms is to maximize the sum of weights of all edges realized in the trace. This means that the objective function used by us is:

$$\sum_{i=1}^{n} w(e_i) \rightarrow max,$$

where $w(e_i)$ is the weight of edge e_i and $e_1, \ldots, e_m$ are the edges of the trace.

The next chapter (chapter 4) describes two implemented greedy-algorithms. The first algorithm is based upon an idea of Kececioglu introduced in [12]. He uses a progressive strategy for calculating a lower bound in his branch and bound approach. The algorithm performes the following steps:

1. Determine the current front, i.e. the first symbol in each sequence, that was not considered up to now.
2. Determine the subgraph induced by the current front of the alignment graph;
3. Detect the components in this subgraph.
4. Create a new (directed) graph, where each component is reduced to a node and edges represent relations between the components.
5. Determine the so called *supercomponents*, i.e. the components of this new graph.
6. Select a supercomponent with no incoming edges.
7. Now, for each component in the selected supercomponent, we use a *minimum cut* algorithm to determine the edges that have to be deleted, if that component should be added to the solution trace in order to generate a *valid* solution.
8. Finally we add all edges from that component, which had the overall minimum cut, excluding the set of edges marked by the mincut-algorithm.
9. If there are still symbols, that were not considered until now, goto 1, else terminate.

In the second greedy algorithm all edges in the alignment graph are sorted descending to their weight. This algorithm tries to insert the edge with the highest weight in a trace-structure, if this results in a valid solution. A trace represents an alignment graph that can be mapped to a feasible solution for the MSA-problem. A trace is said to be valid, if there aren't any circles containing an arc. An arc is a directed edge from symbol $s_{i,h}$ to $s_{i,h+1}$ $\forall i = 1 \ldots k$ und $h = 1, \ldots, l_{i-1}$, if there are k sequences and sequence i has a length of l_i.

So before inserting a new edge, we use a depth first search algorithm to test, if a circle containing an arc is created by inserting that edge. If there is such a circle, we ignore the selected edge, else it is inserted into the trace. By inserting repeatedly new edges we build up a multiple sequence alignment.

We have developed also an enhancement for the trace-structure, so that we don't need to perform the time consuming depth first search algorithm for determining valid edges and substantial performace improvements could be realized.

Chapter 5 describes three local improvements that we have implemented. The first one, called *MoveGap* tries to exchange a gap in a multiple alignment with an adjacent symbol. If the overall weight increases, this change is kept, else it is undone. An enhancement of this scheme allows the moving of one symbol over more than one single gap to achive even better results.

The second implemented local heuristic (*MoveBlock*) tries to move whole blocks of symbols over one ore more gaps. Here too, if we can achive an improvement we keep the change, else we undo it. Tests have shown, that this improvement is not often successful, if the input-alignment has already a high quality.

The last local improvement method (*InsertEdge*) doesn't work on multiple alignments, but on the trace. It tries to insert a not realized edge of the alignment graph into the trace. Generally an edge cannot be inserted without making the trace invalid. To prevent this, we have to delete some other edges of the trace, in order that there aren't any circles containing arcs after the insertion of the new edge. With the help of a minimum cut algorithm, we find the set of edges with minimal weight that we have to delete. After this step, it is often possible to insert even other edges. If the sum of weights of all inserted edges is larger than the weight of all deleted edges, we keep the change, else we restore the original trace. This procedure is repeated for a predefined number of iterations.

Chapter 6 proposes a tabu search approach for solving the multiple sequence alignment problem. It performes the following steps:

1. Calculation of the set containing all feasible edges, i.e. edges of the alignment graph, which aren't realized in the current trace.

2. Determine the gain g of weights in the trace, if the edge with the highest weight of the above set is inserted into the trace. This step

uses a modificated InsertEdge-algorithm.

3. If $g >$ bound b, insert the edge, else add the edge to the tabu-list NotInsert and goto 6.
4. Add all inserted edges to the tabu-list NotDelete.
5. Add all deleted edges to the tabu-list NotInsert.
6. Delete all edges from the tabu-lists that were for more than k iterations in these lists.
7. If we are in the i^{th} iteration, then exit, else goto 1. i is a predefined value.

Chapter 7 describes the used tools for implementing all algorithms and gives a short overview of the source code. It describes the most important classes and modules and the order in which each module or algorithm is executed.

...

Literaturverzeichnis

[1] L. Brocchieri and S. Karlin. Significant Improvement in Accuracy of Multiple Protein Sequence Alignments by Iterative Refinements as Assessed by Reference to Structural Alignments. *Journal of Molecular Biology*, 276(4):823–838, 1996.

[2] L. Brocchieri and S. Karlin. Asymetric-iterated multiple alignment of protein sequences. *Journal of Molecular Biology*, 276:249–264, 1998.

[3] K. Bucka-Lassen, O. Caprani, and J. Hein. Combining many multiple alignments in one improved alignment. *Bioinformatics*, 15(2):122–130, 1999.

[4] F. Corpet. Multiple sequence alignment with hierarchical clustering. *Nucleic Acids Research*, 16:10881–10890, 1988.

[5] S. R. Eddy. Multiple alignment using hidden Markov models. In *Third international conference on intelligent systems for molecular biology*, Cambridge England. AAAI Press, 1995.

[6] F. Glover and M. Laguna. Tabu Search. In *Modern Heuristic Techniques for Combinatorial Problems*, Colin R. Reeves (Ed.), pages 70–150. Blackwell, 1993.

[7] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[8] J. Heringa. Two strategies for sequence comparison: profile-preprocessed and secondary structure-induced multiple alignment. *Computers and Chemistry*, 23:341–364, 1999.

[9] X. Huang and W. Miller. A time-efficient, linear-space local similarity algorithm. *Advances in Applied Mathematics*, 12:337–357, 1991.

[10] R. Hughey and A. Krogh. Hidden Markov models for sequence analysis: extension and analysis of the basic method. *Computer Applications in Biological Science*, 12:95–107, 1996.

[11] M. Jünger, G. Rinaldi, and S. Thienel. The Minimum Capacity Cut Problem. Algorithms, Implementations, and Experiments. Manuscript, 1997.

[12] 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, pages 106–119. Springer, 1993.

[13] 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.

[14] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Science, USA*, 86:4412–4415, 1989.

[15] K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[16] B. Morgenstern, A. Dress, and T. Wener. Multiple DNA and protein sequence based on segment-to-segment comparison. *Proceedings of the National Academy of Science, USA*, 93:12098–12103, 1996.

[17] 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.

[18] Cédric Notredame. Recent Progresses in Multiple Sequence Alignment: A survey. *Pharmacogenomics*, 3(1):131–144, 2001.

[19] Cédric Notredame and Desmond G. Higgins. SAGA: Sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524, 1996.

[20] Cédric Notredame, Desmond G. Higgins, and Jaap Heringa. T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment. In *JMB*, volume 302, pages 205–217. Academic Press, 2000.

[21] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Science, USA*, 85:2444–2448, 1988.

[22] K. Reinert, J. Stoye, and T. Will. An iterative method for faster sum-of-pair multiple sequence alignment. *Bioinformatics*, 16(9):808–814, 2000.

[23] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.

[24] R. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[25] T. F. Smith and M. S. Waterman. Comparative biosequence metrics. *Journal of Molecular Biology*, 147:195–197, 1981.

[26] J. Stoye, V. Moulton, and A. W. M. Dress. DCA: An efficient implementation of the divide-and-conquer approach to simultaneous multiple sequence alignment. *Computer Applications in the Biosciences*, 13:625–626, 1997.

[27] Julie D. Thompson, Frédéric Plewniak, and Olivier Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[28] Julie D. Thompson, Frédéric Plewniak, and Olivier Poch. BAliBASE: A benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics Application Note*, 15(1):87–88, 1999.

[29] Julie D. Thomson, Desmond G. Higgins, and Toby J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting position specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[30] C. Zhang and A. K. Wong. A genetic algorithm for multiple molecular sequence alignment. *Computer Applications in the Biosciences*, 13(6):565–81, 1997.