Diplomarbeit

„Design und Evaluierung von Parsing-Techniken für das Sequenz-Struktur-Alignment von RNA mit Pseudoknoten“

Albert-Ludwigs-Universität Freiburg
Fakultät für angewandte Wissenschaften
Lehrstuhl für Bioinformatik

Jörg Bruder - 1792498

Gutachter:
Prof. Dr. Rolf Backofen
Dr. Sebastian Will

21. Januar 2009

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1 Zusammenfassung

Sehr viele Probleme, die sich mit Berechnungen von RNA beschäftigen, die Pseudoknoten enthalten, sind NP-hart. Am bekanntesten sind hier die Fälle der Strukturvorhersage und der Berechnung des Alignments. Im Bereich der Strukturvorhersage wurden schon verschiedenste Algorithmen vorgestellt. Diese beinhalten allerdings Einschränkungen was die Art der vorkommenden Pseudoknoten angeht, um effizient arbeiten zu können. Diese hier vorgestellte Arbeit leistet einen Beitrag zu einem neuen Algorithmus, der vom Lehrstuhl für Bioinformatik vorgestellt wurde. Mit seiner Hilfe ist es möglich, effizient Sequenz-Struktur-Alignments von RNA Sequenzen zu berechnen, die Pseudoknoten enthalten. Hierfür wird auf das Prinzip der dynamischen Programmierung zurückgegriffen.

Der Algorithmus besteht dabei im Wesentlichen aus zwei Teilen. Zuerst wird eine der beiden Sequenzen in einen Parsetree zerlegt und anschließend das Alignment gebildet. Das Alignment profitiert hierbei von Einschränkungen auf bestimmte Klassen von Pseudoknoten auf die selbe Weise wie die jeweilige Strukturvorhersage. Dies hat zur Folge, dass die Komplexität des Alignments nur um einen linearen Faktor in Bezug auf den jeweiligen Vorhersagealgorithmus steigt.

Diese Arbeit beschäftigt sich mit dem ersten Teil, dem Berechnen einer Zerlegung der ersten Sequenz. Hier werden verschiedene Methoden untersucht, wie dies geschehen kann, sowie diese hinsichtlich ihrer Auswirkungen auf das spätere Alignment analysiert.

2 Einleitung

2.1 Motivation und Ziel der Arbeit

Die Idee für diese Arbeit entstammt einem Projekt des Lehrstuhls für Bioinformatik der Universität Freiburg. Der Lehrstuhl entwickelt einen neuen Algorithmus zur effizienteren Berechnung eines paarweisen Sequenz-Struktur-Alignments von RNA-Sequenzen, die Pseudoknoten enthalten.

Während es schon viele Arbeiten gibt, die sich mit Sequenzen beschäftigen, die keine Pseudoknoten enthalten, sieht es auf diesem Gebiet weniger gut aus. Dies hat drei Gründe, die voneinander abhängen:

- Bisher sind wenige Pseudoknoten bekannt.
- Die Vorhersage ist sehr rechenintensiv, das Gesamtproblem des Alignments verschiedener Pseudoknoten ist NP-hart [LP00].
- Die Zuverlässigkeit der bisher vorhandenen Algorithmen zur Vorhersage ist nicht sehr gut.

In der Praxis zeigt es sich, dass in der Evolution die Strukturen von RNA besser konserviert worden sind, als allein deren Sequenz. Aus diesem Grund ist es von entscheidender Bedeutung, bei Untersuchungen nach Verwandschaft oder evolutionären Zusammenhängen nicht nur die Abfolge der einzelnen Basen, sondern auch die Struktur, die diese Sequenz bildet, zu betrachten und damit ein Sequenz-Struktur-Alignment[1] durchzuführen.

Es gibt verschiedene Algorithmen, die sich mit RNA Sequenzen und deren Struktur beschäftigen. Zunächst einmal gibt es die sogenannten Strukturvorhersagealgorithmen. Diese versuchen zu einer gegebenen Sequenz die optimalste Struktur zu berechnen. Diejenigen, die dabei Pseudoknoten berücksichtigen, benutzen meistens heuristische Such-

[1] Alignment von Sequenz *und* Struktur

prozeduren und vernachlässigen dabei oft die Optimalität [RE99a]. Ansätze die dagegen die Optimalität garantieren, haben eine sehr hohe Komplexität.

Die Algorithmen von Abrahams et al. (1990), Gultyaev et al. (1995) und van Batenburg et al. (1995) verwenden quasi-Monte-Carlo Suchverfahren bzw. genetische Algorithmen. Diese können allerdings keine optimale Vorhersage garantieren und auch nicht bestimmen, wie groß die Differenz zwischen der gefundenen und der optimalen Lösung ist. Cary & Stormo (1995) und Tabaska et al. (1998) verwenden einen anderen Ansatz. Deren Algorithmen basieren auf dem *maximum weighted matching*. Sie sind in der Lage für Sequenzen, für die bereits ein multiples Alignment existiert, eine optimale Lösung zu finden. Die hierfür benötigte Komplexität liegt in $O(n^3)$ für die Zeit und in $O(n^2)$ für den Speicherbedarf.

Rivas & Eddy stellten 1999 einen Algorithmus vor, der die sogenannte *minimal free energy RNA structure* findet, also die Struktur mit der minimalsten freien Energie. Dafür benutzen sie das thermodynamische Model, welches bei vielen Algorithmen Verwendung findet. Mit Hilfe dieses Ansatzes lassen sich überlappende Pseudoknoten berechnen, also solche, die in einer planaren Abbildung ohne Kreuzungen darzustellen sind. Es können aber nicht alle möglichen Strukturen gefunden werden. Hierfür benötigt das Verfahren $O(n^6)$ Zeit und $O(n^4)$ Speicherplatz. Von diesem Ansatz gibt es in der Zwischenzeit weitere Varianten und Verbesserungen [CDR+04]. Diese haben zwar eine niedrigere Komplexität, können aber auch nur eingeschränktere Klassen von Pseudoknoten vorhersagen. Solche Algorithmen wurden zum Beispiel von Akutsu und Uemura et al. [UHKY99, Aku00], Dirks und Pierce [DP03], Lyngso und Pedersen [LP00] sowie von Reeder und Giegerich [RG04] entworfen. Einen kurzen Vergleich der Komplexitäten dieser Varianten liefert folgende Tabelle:

class		**R&E**	**A&U**	**L&P**	**D&P**	**R&G**
prediction	time	$O(m^6)$	$O(m^4)$	$O(m^5)$	$O(m^5)$	$O(m^4)$
	space	$O(m^4)$	$O(m^3)$	$O(m^3)$	$O(m^4)$	$O(m^2)$

Tabelle 2.1: Komplexitätsvergleich von Strukturvorhersageansätzen [MWB09]

Werkzeuge, die eine Strukturvorhersage allerdings ohne Berücksichtigung von Pseudoknoten durchführen sind zum Beispiel LocARNA [WRH+07], MARNA [SB05] und FOLDALIGN [HLSG05, HLG05], um nur einige zu nennen. Diese führen aber keine reine Strukturvorhersage durch, sondern kombinieren diese bzw. benutzen sie zur Berechnung eines Alignments.

Es werden aber nicht nur Strukturen vorhergesagt, sondern auch mit anderen verglichen. Algorithmen zur Berechnung eines hierfür notwendigen Sequenz-Struktur-Alignments unter der Berücksichtigung von Pseudoknoten sind dagegen noch nicht so verbreitet.

Das von Patricia Evans vorgestellte Verfahren [Eva06] beschränkt sich dabei auf die Betrachtung von Strukturen, deren Strukturkanten 2-färbbar sind. Das bedeutet, dass Kanten, die sich überkreuzen, unterschiedlich gefärbt sein müssen und hierfür maximal zwei Farben benutzt werden dürfen. Nicht erlaubt sind dabei 3-Knoten und Strukturen mit ineinander geschachtelten Endpunkten. Für die Berechnung benötigt dieser Ansatz allerdings $O(n^8)$ Zeit und $O(n^{10})$ Speicher.

Ein anderes vom Lehrstuhl für Bioinformatik vorgestelltes Verfahren [MWB08] benötigt für ein solches Alignment $O(n^2 s^{8k})$ Zeit und benutzt $O(n^2 s^{16k})$ Speicherplatz. Hierbei sind n die Länge der längsten Sequenz, s die maximale Anzahl von Kanten in sich kreuzenden stems und k die maximale Anzahl von Kreuzungen. Allerdings ist dieser Algorithmus in der Lage, beliebige Pseudoknoten zu berechnen. Hierfür wird die Sequenz in einfache und schwierige Teile geteilt. So muss der komplexe Algorithmus nur auf die schwierigen Teile angewandt werden, während für die einfachen Teile die bisher bekannten und gut funktionierenden Methoden zum Einsatz kommen können.

Da die bisher existierenden Ansätze zur Lösung sehr rechenintensiv sind, ist die Idee, einen neuen Algorithmus zu entwickeln, der auf den Vorhersagealgorithmen und deren Einschränkungen beruht. Dieser Algorithmus basiert auf der Strategie der Dekomposition der Strukturvorhersage. Zur Berechnung des Alignments wird in einem Vorverarbeitungsschritt eine der beiden Sequenzen bis in ihre kleinsten Teile zerlegt und ein Baum der Zerlegung aufgebaut. Da die Komplexität der späteren Alignmentberechnung von diesem Baum abhängt, ist es wichtig, für die Sequenz eine effiziente und „günstige“ Zerlegung zu finden. Mit dieser Problematik beschäftigt sich diese Arbeit und stellt mehrere Ansätze vor, von denen auch deren jeweilige Vor- und Nachteile diskutiert, sowie deren Ergebnisse betrachtet werden.

2.2 Aufbau der Arbeit

Um einen Überblick über diese Ausarbeitung zu geben, folgt ein kurzer Abriss über die Inhalte der folgenden Kapitel.

In Kapitel 3 finden sich zu erst eine Auswahl an Hintergrundinformationen, um der Arbeit besser folgen zu können. Diese werden sowohl biologischer als auch algorithmischer Natur sein sein. Nachfolgend wird in Kapitel 4 der zugrunde liegende Alignmentalgorithmus beschrieben und einige Festlegungen getroffen, was die Zerlegung der einen Sequenz angeht.

Kapitel 5 und 6 behandeln zwei Möglichkeiten eine solche Zerlegung zu erlangen. Hier wird sowohl auf die Funktionsweise eingegangen als auch mögliche Probleme und Lösungsansätze diskutiert.

In Kapitel 7 werden die Ergebnisse vorgestellt und anschließend in Kapitel 8 diskutiert. In Kapitel 9 wird zum Abschluss der Arbeit noch ein Fazit gezogen, welches die gesamte Arbeit kurz zusammenfasst.

3 Hintergrundinformationen

3.1 Biologische Grundlagen

3.1.1 RNA allgemein

Die RNA[1] ist eine Kette aus einzelnen Nukleotiden, also ein Polynukleotid. Jedes dieser einzelnen Nukleotide besteht aus einem Phosphatteil, einem Zuckerteil und aus einer Base. Als Basen kommen *Adenin (A)*, *Cytosin (C)*, *Guanin (G)* und *Uracil (U)* in Frage. Hier ist ein Unterschied zur DNA, denn in ihr kommt als vierte Base statt Uracil Thymin (T) vor. Desweiteren wird in der RNA ein anderer Zucker verwendet (RNA Ribose, DNA Desoxiribose). Während die DNA aus zwei Strängen besteht und die bekannte Helixstruktur bildet, kommt die RNA nur als Einzelstrang vor. Die in der RNA enthaltenen Zucker werden durch die Phosphatgruppen mit Hilfe von Phosphodiesterbindungen zusammengehalten. Diese Bindung tritt zwischem dem 3. und 5. Kohlenstoffatom der nebeneinanderliegenden Zuckerringe auf. Dadurch wird auch die „Richtung“ einer RNA-Kette festgelegt, da man diese immer vom 5’ zum 3’ Ende hin betrachtet. Wie die DNA auch, kann die RNA zwischen komplementären Basen Wasserstoffbrückenbindungen (Watson-Crick) [WC53] ausbilden. Diese treten zwischen Adenin und Uracil und zwischen Cytosin und Guanin auf. In gewissen Fällen kann es auch auftreten, dass sogenannte *wobble pairings* zwischen Guanin und Uracil auftreten. Eine Verbindung zwischen zwei Basen wird als *Basenpaar* bezeichnet. Während bei der DNA solche Basenpaare zwischen Basen zweier Nukleotidstränge auftreten, geschieht dies bei der RNA innerhalb eines Stranges.

Man kann zwischen kodierender und nicht kodierender RNA unterscheiden. Erstere hat die Funktion der Speicherung und des Transports genetischer Information, nicht kodierende RNAs sind Hilfsmittel in der Genregulation und sind in verschiedenen Gebieten involviert [HBB02]:

- Translation (transfer RNA (tRNA), ribosomal RNA (rRNA))

[1] engl: ribonucleid acid, Ribonukleinsäure

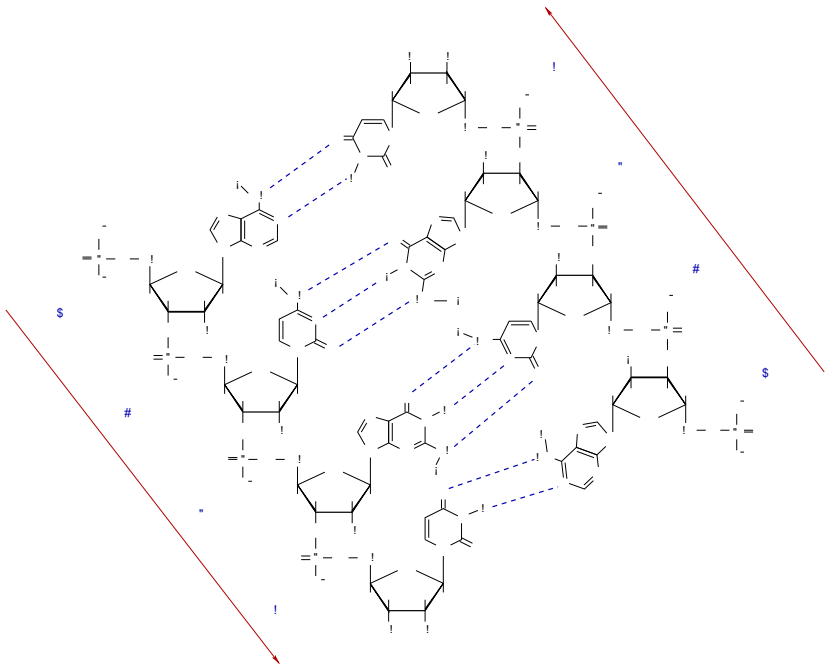

Abbildung 3.1: Darstellung der chemischen Zusammensetzung und Aufbau eines RNA Doppelstranges [Bus08]

- Splicing (small nuclear RNA (snRNA))
- Verarbeitung anderer RNA (smaller nucleolar RNA (snoRNA), nuclear ribonuclease P (RNAseP))
- regulatorische Prozesse (micro RNA (miRNA), small interfering RNA (siRNA))

Die tRNA (transfer oder transport RNA) ist ein relativ kurzes RNA Molekül, welches im Zellkern synthetisiert wird. Sie bindet eine der im Zytoplasma vorhandenen Aminosäuren für deren Transport zu den Ribosomen. Auf den Ribosomen sitzt die mRNA mit den kopierten Basentriplets. Diese gehört anders als die bisher angesprochenen Arten zur Gruppe der kodierenden RNA. Die tRNA-Moleküle sind jeweils für eine Aminosäure und das dazugehörende Basentriplet spezifisch. Die RNA, die für den Transport der Informationen zu den Ribosomen zuständig ist, ist die mRNA oder auch Boten-RNA (messenger RNA). Deren Bindung an die Ribosomen erfolgt durch Basenpaarungen, welche die mRNA mit der erwähnten tRNA eingeht. Die Verknüpfung der verschiedenen Aminosäuren zu einer Proteinkette erfolgt mit Hilfe von ribosomalen Enzymen. Die Information zu deren Bildung wiederum liefert die rRNA, welche innerhalb des Nucleolus

produziert wird. Somit spielt die RNA wichtige Rollen bei der Transkription, aber auch bei der Translation des genetischen Codes innerhalb der Zelle [AF99].

Forschungen haben ergeben, dass die Funktion von RNA sowohl von der eigentlichen Nukleotidsequenz, aber auch von der ausgebildeten Struktur abhängig ist. In vielen Fällen ist die Struktur sogar wichtiger als die Sequenz, da sie auch evolutionär oft besser konserviert ist.

3.1.2 Primär-, Sekundär- und Tertiärstruktur

Bei diesen drei Strukturarten handelt es sich um verschiedene Arten, die RNA zu beschreiben. Die einfachste Betrachtungsweise ist es, die sequentielle Abfolge der Basen zu betrachten. Dies geschieht immer vom 5' zum 3' Ende hin. Diese wird als Primärstruktur bezeichnet und kann formal wie folgt definiert werden:

Definition 3.1. Eine Sequenz $S = S_1 \dots S_n$ von n Basen S_i für die $1 \leq i \leq n$ und $S_i \in \{A, C, G, U\}$ gilt, wird **Primärstruktur** der RNA genannt.

Da wie schon erwähnt ein RNA-Strang Bindungen mit sich selbst eingehen kann, muss dies auch dargestellt werden können. Die Darstellung dieser Bindungen wird im zweidimensionalen Raum als Sekundärstruktur (siehe Abbildung 3.2) bezeichnet.

Definition 3.2. Die **Sekundärstruktur** einer RNA Sequenz S ist als die Menge der Basenpaare $T = \{(S_i, S_j) \mid 1 \leq\ i < j \leq n$, mit Bindung zwischen S_i und $S_j\}$ definiert. (S_i, S_j) oder im folgenden kürzer (i, j), bezeichnet dabei das Basenpaar der Basen S_i und S_j. Jede Base kann höchstens eine Bindung mit einer anderen Base eingehen. Dies hat zur Folge, dass zwei Basenpaare (i, j) und (k, l) entweder identisch sind $(i = k \wedge j = l)$, oder sich aber in beiden Stellen unterscheiden $(i \neq k \wedge j \neq l \wedge j \neq k \wedge i \neq l)$ [Bus08].

Betrachtet man allerdings die tatsächliche räumliche Ausformung der RNA, so spricht man von einer Tertiärstruktur. Diese wird benötigt, um die katalytische Aktivitäten der RNA besser verstehen zu können. Sie zeigt die tatsächliche Position und Bindungen der einzelnen Basen und bildet im Falle von tRNA eine typische L-Form, welche ebenfalls in Abbildung 3.2 zu sehen ist. In manchen Fällen, kann die Tertiärstruktur einer RNA durch Kristallographie bestimmt werden. Da eine RNA aber im Gegensatz zu Proteinen sehr flexibel ist, ist dies nur schwer möglich. Desweiteren ist eine Sekundärstruktur stärker und formt sich weit schneller, als die Tertiärstruktur [LTM06, Woo00]. Außerdem bedingt die zweidimensionale die dreidimensionale Struktur. Im weiteren Verlauf dieser Arbeit ist, wenn von *Struktur* die Rede ist, die Sekundärstruktur gemeint.

Zusätzlich zu der in den Abbildungen 3.2 und 3.3 verwendeten Darstellungsart, werden hier im weiteren Verlauf Sekundärstrukturen durch einen sogenannten Klammerstring dargestellt (*dot-bracket Notation*). In diesem wird jedes Basenpaar (i, j) durch eine öffnende bzw. schließende Klammer an den Stellen i und j symbolisiert, während das Zeichen für ungepaarte Basen ein Punkt ist. Ein Beispiel wie ein solcher Klammerstring aussehen kann wäre: ...(((((((...)))...)))).. . Müssen Pseudoknoten berücksichtig werden (siehe nächster Abschnitt), so werden die Basenpaare, die zu dem Pseudoknoten gehören, also die anderen Bindungen „schneiden“, mit anderen Klammernsymbolen dargestellt (meist „[„, “]“, “{„, “}“).

Abbildung 3.2: Sekundär- und Tertiärstruktur von RNA
Quelle: http://www.molecularstation.com/science-news/wp-content/uploads/2008/03/trna-structure.jpg

Solch eine Struktur kann man nicht nur im Ganzen betrachten, kann kann auch die einzelnen Teilsegmente einzeln untersuchen. Abbildung 3.3 zeigt, wie diese Teile dann benannt werden [AZC05, Bus08].

- Ein *hairpin loop* ist eine „Schlaufe“, die nur ungepaarte Basen enthält. Eine solche Schleife muss dabei im Normalfall mindestens 2 Basen enthalten.

- Ein *stack* oder *stem* ist eine Schleife, die nur Basenpaare und keine freien Basen enthält.

- Ein *internal loop* beinhaltet ein Basenpaar und hat eine Größe > 0. Kommen alle ungepaarten Basen auf einer Seite des loops vor, spricht man von einem *bulge loop*.

- Ein *multiloop* ist eine Schleife, die mehr als ein Basenpaar enthält.

Diese Tatsache, dass man eine komplexe Struktur aus einfacheren, kleinen Teilen zusammensetzen kann, machen sich viele Algorithmen zu Nutze, deren Aufgabe es ist, eine Sekundärstruktur zu einer gegebenen Basensequenz vorherzusagen. Einer der prominentesten Vertreter aus diesem Gebiet wurde bereits 1981 vorgestellt [ZS81].

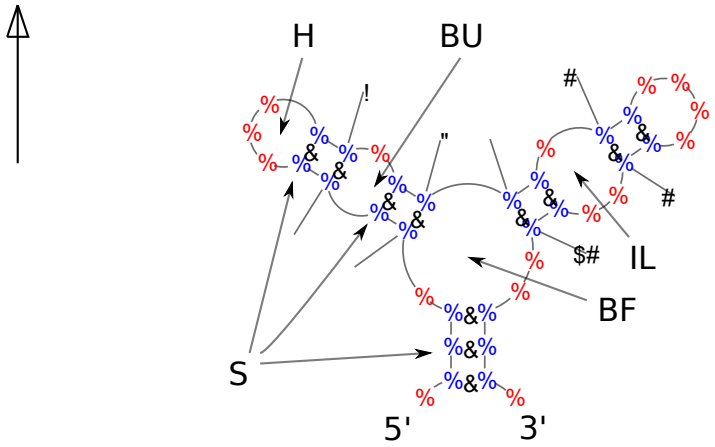

Abbildung 3.3: Substrukturen einer Sekundärstruktur
(S) stem, stack, Stamm; (H) haipin loop, Haarnadelstruktur; (BU) bulge loop; (BF) bifurcation loop, multiloop; (IL) internal loop;

3.1.3 Pseudoknoten

Unter bestimmten Voraussetzungen kann eine RNA sogenannte Pseudoknoten (siehe Abbildung 3.4) ausbilden. Dieser Begriff des Pseudoknotens wurde nach bisherigen Erkenntnissen zum ersten Mal Ende der siebziger Jahre in Zusammenhang mit der Entwicklung von Algorithmen zur Sekundärstrukturvorhersage von RNA eingeführt [SS78]. Heutzutage ist die Anzahl der bekannten Strukturen, die Pseudoknoten enthalten, derart gestiegen, dass sie in geradezu allen Klassen der RNA vorkommen. Hier spielen sie wichtige Rollen in der Funktion oder kommen an funktionell strategisch wichtigen Positionen vor [PRB85].

Ein Pseudoknoten ist ein Strukturelement, welches dann entsteht, wenn eine normale Basenpaarung zwischen den ungebundenen Basen innerhalb eines Loops und Basen die außerhalb des Loops stehen, auftritt. Innerhalb eines Loops bezeichnet damit all die Abschnitte in der Sequenz, die ungepaarte Basen enthalten und von mindestens einem Stem begrenzt werden (siehe Abbildung 3.4 die Basen 14-16 und 34-42). Dies hat zur

Folge, dass jeder Pseudoknoten durch mindestens zwei Stems definiert wird. Welcher der Stems dabei zuerst gebildet wird spielt für die eigentliche Definition keine Rolle, könnte sich aber bei der Bildung der Tertiärstruktur bemerkbar machen. In dem abgebildeten Fall sind die beiden Stems, die den Pseudoknoten bilden, adjazent zueinander. Eine solche Struktur nennt man dann *H-type Pseudoknoten* (hairpin) [PRB85]. Des weiteren besteht zum Beispiel die Möglichkeit, dass die ungepaarten Basen zweier hairpin loops eine Bindung aufbauen. Dies wird dann als *kissing hairpins* bezeichnet. Der H-Typ Pseudoknoten ist der in der Literatur am meisten behandelte und auch am meisten vorkommende der bisher bekannten Pseudoknoten. Ein Beispiel für natürlich vorkommende Pseudoknoten sind die tmRNA-Sequenzen aus Escherichia coli (E. coli). Diese besitzen insgesamt vier Pseudoknoten [ZWW99].

RNA Pseudoknoten spielen eine fundamentale Rolle bei der strukturellen Organisation komplexer RNAs, im Aufbau von Ribonukleoproteinkomplexen sowie zur Regulation, Übersetzung und Neucodierung im Zusammenhang mit messenger RNA. RNA Strukturen, die Pseudoknoten enthalten, wurden in nahezu allen Typen natürlich vorkommender RNA gefunden. So zum Beispiel in messenger RNAs (mRNA) und transfer-messenger RNA (tmRNA). Sie spielen ebenso verschiedene Rollen bei der Synthese von Proteinen. So kommt manchmal ein Pseudoknoten bei der Einleitung einer Übersetzung am Anfang einer nicht kodierenden Sequenz des Proteins vor. Es kann allerdings ebenfalls auftreten, dass die Pseudoknoten innerhalb der kodierenden Region vom mRNA liegen [GTN00].

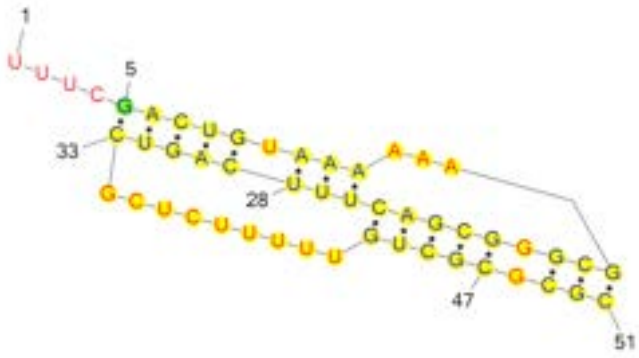

Abbildung 3.4: Darstellung eines natürlich vorkommenden Pseudoknotens
Erstellt mit: PseudoViewer Web Service (http://pseudoviewer.inha.ac.kr/)

Die meisten bekannten Vorhersagealgorithmen, so auch der oben angesprochene [ZS81], können allerdings nicht auf Strukturen angewandt werden, die Pseudoknoten enthalten, bzw. sie würden ein pseudoknoten-freies Ergebnis liefern. Mit bisherigen Vorgehensweisen ist es sehr schwer, diese Aufgabe der Strukturvorhersage mit Pseudoknoten zu lösen [AP90]. Trotzdem gibt es Lösungsansätze für dieses Problem, auch wenn diese noch sehr

hohe Komplexitäten ausweisen. Hier hinken sie den bekannten Lösungen zur Strukturvorhersage noch weit hinterher [RE99b, UHKY99, Aku00, DDKM04, LP00, DP03, RG04]. Man sollte allerdings diesen Algorithmen und ihren Ergebnissen nicht blind vertrauen, denn alle Programme haben ihre Grenzen. So bleibt es stets wichtig, dass die errechneten Pseudoknoten auf andere Weise, zum Beispiel durch stammesgeschichtliche Nachforschungen, verifiziert werden [TDPD92].

Die Verschachtelung macht aber die Berücksichtigung dieser Pseudoknoten für „normale“ Algorithmen sehr schwierig. Da diese auf dem Prinzip der dynamischen Programmierung beruhen, können sie diese Pseudoknoten aufgrund der Überlappung der Basenpaare nicht berücksichtigen. Wie schon erwähnt, sind Pseudoknoten alles andere als rar, so dass es durchaus notwendig ist, diese zu betrachten.

3.2 Algorithmische Grundlagen

Die meisten der auf diesem Gebiet bereits existierenden Algorithmen beruhen auf dem Prinzip der dynamischen Programmierung (DP). Dies ist eine Lösung für Optimierungsprobleme und kann immer dann eingesetzt werden, wenn sich das Gesamtproblem aus mehreren gleichartigen Teilproblemen zusammensetzt und sich dadurch auch die optimale Lösung des Gesamtproblems durch die Kombination der optimalen Lösungen für die Teilprobleme berechnen lässt [RG96]. Für die Zwischenspeicherung der Teilergebnisse werden häufig Tabellen verwendet. Dieses Prinzip kann zeit- und speicherplatzintensive Rekursionen vermeiden, da auf zuvor berechnete Ergebnisse einfach zugegriffen werden kann, anstatt sie jedes mal neu zu berechnen. Zu den in Bezug auf RNA am meisten auftretenden Problemstellungen der Informatik ist die Vorhersage, welche Struktur ein gegebener RNA-Strang bilden wird und die Berechnung des Alignments zweier oder mehrerer RNA-Sequenzen. Bei letzterem wird allerdings noch unterschieden, ob man nur die Sequenz der aufeinanderfolgenden Basen betrachtet (Sequenzalignment) oder aber auch zusätzlich die Struktur (Sequenz-Struktur-Alignment).

3.2.1 Verwendung von Heuristiken

In der Informatik kommen in vielen Fällen Heuristiken zum Einsatz. Dabei wird versucht, für ein gegebenes Problem, mit nur begrenztem Wissen und Ressourcen, vornehmlich Zeit, eine möglichst exakte Lösung zu finden. Im Normalfall wird durch einen Algorithmus versucht, sowohl die optimale Rechenzeit, als auch die optimale Lösung zu garantieren. Bei komplexen Problemstellungen kann es aber durchaus dazu kommen,

dass der Rechenaufwand eine optimale Lösung zu finden, explodiert. In diesen Fällen wird mit heuristischen Ansätzen und Methoden versucht, einen Kompromiss zwischen der Güte des Ergebnisses und der dafür aufgewendeten Zeit zu finden. Dies geschieht auf unterschiedliche Weise, so zum Beispiel durch Schätzungen, gewissen Regeln oder situationsbedingtes „raten“. So wird eine Lösung gesucht, ohne aber für deren Optimalität zu garantieren. Heuristiken werden auch Approximationsalgorithmen genannt und lassen sich formal wie folgt beschreiben:

Sei $S(x)$ ein zu einer gegebenen Eingabe x gehörender Lösungsraum. Für jede mögliche Lösung $y \in S(x)$ sei nun $v(y)$ deren Güte und die Güte der optimalen Lösung sei definiert als v^*. Das Ziel einer Heuristik oder eines Approximationsalgorithmus ist es nun eine Lösung $y \in S(x)$ zu finden, so dass $v(y)$ möglichst nah an v^* liegt. Dies kann sowohl auf Minimierungs- als auch auf Maximierungsprobleme angewendet werden. Sie kommen auch zum Einsatz, wenn es gilt, kürzeste Wege in Graphen zu finden oder auch bei Virenscannern, wenn es es darum geht, unbekannte Viren zu erkennen. Sie sind in der Regel das Herz von „intelligenten“ Suchverfahren.

3.2.2 Alignment

Das DP-Prinzip wird sehr häufig in der Bioinformatik eingesetzt, um Alignments zu berechnen. Alignments dienen zum Vergleich von Strings (in der Bioinformatik RNA oder DNA Sequenzen) und werden benutzt, um funktionelle und / oder evolutionäre Verwandtschaft verschiedener Sequenzen zu untersuchen, um konservierte Muster zu entdecken oder um bei unbekannten Proteinen durch Vergleich mit bereits bekannten Rückschlüsse auf deren Funktion schließen zu können. Hierbei werden die Elemente des untersuchten Strings in richtiger Reihenfolge denen des anderen zugeordnet. Somit kann ein jedes Element (in unserem Fall jede Base) einem anderen zugeordnet werden, oder aber auf eine Leerstelle treffen (Gap). Je nachdem was für Alignments man berechnen möchte, werden diese Fälle mit Hilfe von Kostenfunktionen anders bewertet. Hierbei ist es meist am billigsten, wenn man eine Base auf die selbe Base in der zweiten Sequenz aligniert. Teurer ist es, wenn diese Base auf eine andere trifft und am teuersten ist es, aligniert man die Base auf ein Gap. Am Ende eines solchen Alignments erhält man einen „Score“, welcher ein Indikator dafür ist, wie ähnlich sich die Sequenzen sind. Abbildung 3.5 zeigt ein einfaches Sequenz-Struktur-Alignment.

Im Bereich der Bioinformatik gibt es verschiedene Arten, Alignments zu unterscheiden. Hier ist zum einen die oben schon angesprochene Unterscheidung, ob man nur die Basensequenz oder auch die daraus resultierende Sekundärstruktur betrachtet. Des weiteren gibt es paarweise und multiple Alignments, also ob nur zwei Sequenzen oder aber mehrere betrachtet werden. Man kann auch jeweils die beiden Sequenzen komplett alignieren

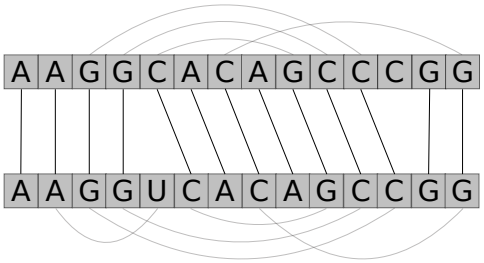

Abbildung 3.5: Darstellung eines einfachen Sequenz-Struktur-Alignments

oder nur den am besten passenden Teil finden. Ersteres ist dann ein globales, zweiteres ein lokales Alignment.

Für das paarweise Sequenzalignment gibt es drei prominente Algorithmen, die dieses Problem lösen [NW70, Got82, SW81]. Diesen gemein ist das verwendete Prinzip der dynamischen Programmierung. Sie führen mindestens eine Tabelle für die Speicherung der jeweiligen Teillösungen und benutzen einen Backtrace um den genauen Pfad zu ermitteln, der eben die Lösung bringt. Ebenfalls verwenden sie eine Kostenfunktion, um zu berechnen, wie gut die jeweiligen Lösungen sind. Es gibt aber durchaus Fälle, bei denen diese Algorithmen unbrauchbar für die Aufgabenstellung sind. Diese betrachten jeweils nur die Basensequenz und nicht deren Struktur. Aus diesem Grund liefern sie schlechte Ergebnisse, wenn nur die Struktur, nicht aber die Sequenz konserviert wurde. Dies liegt daran, dass sie keine Pseudoknoten berücksichtigen.

Für das hier behandelte Gebiet des Sequenz-Struktur-Alignments gibt es auch verschiedene Ansätze. Manche davon können keine Pseudoknoten behandeln [JLMZ02], andere sind dazu durchaus in der Lage. Die prominentesten hierzu wurden in Kapitel 2.1 erwähnt und kurz beschrieben.

3.2.3 Parsing

Beim *Parsing* (der syntaktischen Analyse) handelt es sich um einen der am besten verstanden Zweige der Informatik. Bei einem Parser handelt es sich im allgemeinen um ein Computerprogramm, welches eine beliebige Eingabe in ein für die Weiterverwendung benötigtes Format umwandelt. Sie werden in vielen verschiedenen Einsatzgebieten verwendet. So zum Beispiel in der Compilerkonstruktion, bei der künstlichen Intelligenz, für Datenbankinterfaces, in der Linguistik für Textanalyse und Maschinenübersetzung, in der Dokumentenvorbereitung und bei der Erkennung von Chromosomen, um exem-

plarisch ein paar aufzuzählen [Gru90]. Im allgemeinen werden in der Informatik Parser dazu verwendet, Texte in andere Strukturen zu übersetzten, zum Beispiel in Syntaxbäume. Am meisten bekannt dürfte das Parsing in Verbindung mit Grammatiken sein, wobei ein Parsevorgang dazu dient, festzustellen, ob ein gegebenes Wort zu jener Grammatik gehört, oder ob die Grammatik diese Wort erzeugen kann. Hierbei wird zwischen *top down parsing* und *bottom up parsing* unterschieden. Beim top down Parsing wird dabei mit einem Startsymbol angefangen und durch wiederholte Anwendung von Ableitungsschritten versucht, die zu analysierende Kette zu erzeugen. Genau von der anderen Seite arbeitet ein bottom up Parser. Dieser startet mit der zu analysierenden Sequenz und versucht das Startsymbol durch wiederholtes ableiten in „umgekehrter“ Richtung zu erreichen (siehe Tabelle 3.2). Durch diese verschiedenen Ableitungsschritte wird ein sogenannter Syntaxbaum oder *Parsetree* erzeugt. Ein Beispiel soll folgende kurze kontextfreie Grammatik verdeutlichen, mit welcher man den Syntaxbaum in Abbildung 3.6 erzeugen kann:

S → NP VP	Det → *der*	V → *liebt*
NP → Det N	Det → *ein*	V → *schläft*
VP → V NP	N → *Mann*	V → *sieht*
VP → V	N → *Frau*	V → *denkt*

Tabelle 3.1: einfache Grammatik

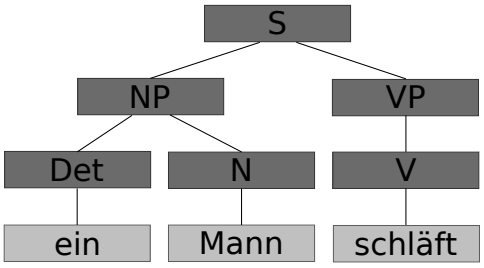

Abbildung 3.6: möglicher Syntakbaum zu der in Tabelle 3.1 gegebenen Grammatik

Als Parse wird dabei die Zerlegung eines Inputstrings in kleine Teile anhand einer gegebenen Grammatik bezeichnet. Hier kann es natürlich durchaus vorkommen, dass es zu einem gegebenen Input viele verschiedene Zerlegungen gibt. Da es bei der Berechnung der einzelnen Teile schnell zu Schwierigkeiten kommen kann, was den Speicherbedarf und die Laufzeit angeht, werden meist die Zwischenergebnisse gespeichert und bei späteren Berechnungen darauf zugegriffen anstatt alles jeweils neu zu berechnen. Dies macht vor allem auch dadurch Sinn, dass ein solches Teilstück einmal berechnet wird und dann von vielen anderen verwendet werden kann. Die Ergebnisse werden meist in Tabellen zwischengespeichert. Daher spricht man auch von *tabular parsing* oder *chart parsing*. Diese Methoden wurden in der Informatik um 1970 entwickelt und zählen zum Bereich

der *dynamischen Programmierung*. Das größte Einsatzgebiet der chart parser ist die syntaktische Analyse natürlicher Sprachen.

Vorgehen top down parsing	**Vorgehen bottom up parsing**
gesucht wird S	nimm das Wort *ein*
für S braucht man NP und VP	*ein* ist Det
für NP braucht man Det und N	nimm anderes Wort *Mann*
für Det kann man *ein* verwenden - gefunden	*Mann* ist N
für N kann man *Mann* verwenden - gefunden	Det und N sind zusammen NP
NP ist vollständig	nimm anderes Wort *schläft*
für VP braucht man V	*schläft* ist V
für V kann man *schläft* verwenden - gefunden	V ist VP
VP ist vollständig	NP und VP bilden zusammen S
S ist vollständig	

Tabelle 3.2: unterschiedliche Arbeitsweisen von Parsern

4 Zugrunde liegender Alignmentalgorithmus

4.1 Grundsätzliche Festlegungen

4.1.1 Allgemeines

Zuerst folgen hier nun grundlegende Definitionen. Dies soll dazu dienen, die anschließenden Ausführungen zu dem hier vorgestellten Alignmentalgorithmus [MWB09] an sich, aber auch zu der Zerlegung der Sequenzen im späteren Teil besser verstehen zu können.

Definition 4.1. Eine **Kanten-annotierte Sequenz** (arc-annotated sequence) ist ein Paar (S, P), wobei S ein String über der Menge der Basen $\{A, C, G, U\}$ ist und P eine Menge von Kanten (l, r) mit $1 \leq l < r \leq |S|$, welche die Bindungen zwischen den Basen repräsentiert, so dass jede Base adjazent zu maximal einer Kante ist.

Im folgenden wird die i-te Stelle in der Sequenz mit $S[i]$ bezeichnet. Für eine Kante $p = (l, r)$ bezeichnen wir die beiden Enden mit p^L und p^R. Zwei Kanten $p, p' \in P$ bilden einen Pseudoknoten, wenn entweder $p^L < p'^L < p^R < p'^R$ oder $p'^L < p^L < p'^R < p^R$ gilt. Ist dies der Fall, spricht man von kreuzenden Kanten (*crossing arc*), andernfalls wird die Struktur *nested* genannt.

Definition 4.2. Ein **Alignment A** von zwei Kanten-annotierten Sequenzen (S_a, P_a) und (S_b, P_b) ist eine Menge $A_1 \cup A_2$, wobei $A_1 \subseteq [1..|S_a|] \times [1..|S_b|]$ eine Menge von „match edges“ ist, so dass für alle $(i, j), (i', j') \in A_1$ gilt 1.) $i > i'$ impliziert $j > j'$ und 2.) $i = i'$ nur genau dann, wenn $j = j'$ und A_2 ist eine Menge von „gap edges“ $\{(x, -) | x \in [1..|S_a|] \wedge \nexists y (x, y) \in A_1\} \cup \{(-, y) | y \in [1..|S_b|] \wedge \nexists x (x, y) \in A_1\}$.

Zwei Basen $S_a[i], S_b[j]$ werden durch A *gematched* (einander zugeordnet), wenn $(i, j) \in A$, und zwei Kanten $p_a \in P_a, p_b \in P_b$ werden gematched, wenn $(p_a^L, p_b^L) \in A$ und $(p_a^R, p_b^R) \in A$.

Jedes Alignment hat dabei bestimmte Kosten, die auf der *edit distance* mit zwei verschiedenen Arten von *edit* Operationen beruht. Diese Operationen wurden von Jiang et al. [JLMZ02] eingeführt.

Definition 4.3. Ein **Fragment** F einer kanten-annotierten Sequenz (S, P) ist ein k-Tupel von *Intervallen* $([l_1, r_1] \dots [l_k, r_k])$ mit $1 \leq l_1 \leq r_1 + 1 \leq \dots \leq l_k \leq r_k + 1 \leq |S|$.

Zu beachten ist hierbei, dass diese Definition *leere* Intervalle $[i + 1, i]$ erlaubt. Die Bereiche zwischen den jeweiligen Intervallen werden *gaps von* F genannt. k wird im folgenden als der *Grad* des Intervalls bezeichnet, $l_1, r_1 \dots l_k, r_k$ als die *Grenzen (boundaries)*. $F[i]$ bezeichnet das i-te Intervall, $F[i]^L, F[i]^L$ seine jeweiligen Grenzen.

Definition 4.4. Ein Fragment F wird **arc-complete** (kantenvollständig) genannt, genau dann, wenn $l \in \hat{F} \Leftrightarrow r \in \hat{F}$ für alle $(l, r) \in P$.

Definition 4.5. Weiterhin wird ein Fragment F **atomar** genannt, wenn F entweder genau die beiden Enden einer Kante von P überdeckt, oder eine einzelne Basenposition darstellt, die nicht adjazent zu einer Kante ist.

Seien nun F, F^1 und F^2 Fragmente der selben Sequenz. Das Paar (F^1, F^2) ist ein *split* von F, genau dann, wenn $\hat{F} = \hat{F^1} \uplus \hat{F^2}$.[1] Weiterhin nennen wir im folgendem F^1 sowie F^2 die *Kinder* und F das *Elternfragment* von dem *Split*. Gilt für beide Kinder und das Elternfragment des Splits, dass sie jeweils arc-complete sind, so nennen wir den Split *arc-preserving* (kantenerhaltend).

4.1.2 Parsetree

Wie bereits erwähnt, verwendet der Algorithmus für eine der beiden Sequenzen eine zuvor berechnete Zerlegung. Diese Zerlegung lässt sich in einem Parsetree abbilden.

Definition 4.6. Ein **Parsetree** ist ein binärer Baum, bei dem jeder innere Knoten ein arc-complete Fragment von (S, P) ist, so dass folgende Bedingungen erfüllt sind:

- Die Wurzel des Baumes ist das Fragment $([1, |S|])$.
- Jeder innere Knoten ist ein Fragment F und hat genau zwei Kinder F^1 und F^2, so dass es sich um einen *arc-preserving*-Split handelt.

[1] Der Einfachheit halber werden nur binäre Splits betrachtet, die Methode kann aber auch für beliebig komplexe Splits angewandt werden.

- Jedes Blatt des Baumes stellt ein *atomares* Fragment dar.

Beispiele für die Darstelllung von Parsetrees zeigt Abbildung 4.1. Ein solcher Baum stellt die feste rekursive Zerlegung einer der beiden zu alignierenden Sequenzen dar. Der Alignmentalgorithmus behandelt die beiden Sequenzen nicht gleich, wie es die meisten Algorithmen in diesem Gebiet tun. Die Rekursion des Alignments folgt dabei eben dieser fix vorgegebenen Zerlegung der ersten Sequenz. An jedem inneren Knoten (also jedem Splitpunkt) werden dann ***alle*** kompatiblen Splits (Splits der gleichen Art) der zweiten Sequenz untersucht.

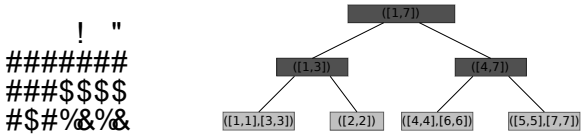

Abbildung 4.1: Darstellungsmöglichkeiten von Parsetrees

Die beiden Graphiken in Abbildung 4.1 zeigen zwei Möglichkeiten, eine solchen Parsetree zu visualisieren. Bei beiden wurde die gleiche Zerlegung dargestellt. In der linken Darstellung steht ein Buchstabe immer für ein Fragment, in der Zeile darunter dann, wie dieses Zerlegt wird (Beispiel aaaaaaa wird in aaa und bbbb zerlegt). Zu beachten ist, dass jedes Blatt eines solchen Zerlegungsbaumes ein atomares Fragment darstellt.

4.1.3 Der Splittyp

Im folgendem definieren wir einen String T über der Menge $\{1, 2, G\}$ als den *Basistyp* eines Splits. Jeder auftretende Split hat genau einen passenden Basistyp, zu dem er gehört (siehe Abbildung 4.2). Die Intervalle der beiden Kinder des Fragments werden dabei in aufsteigender Reihenfolge geordnet und alle Intervalle des ersten Kindes F^1 durch 1 ersetzt, die des zweiten Kindes durch 2. Handelt es sich bei dem betrachteten Elternfragment um ein Fragment mit Grad 2, so wird im Basistyp das „Loch“ mit dem G gekennzeichnet. Diese Einteilung kann durch bestimmte Bedingungen noch verfeinert werden. So werden weiterhin alle Intervalle, die die Länge 1 besitzen, zusätzlich durch ein „ ‟“ gekennzeichnet. Diese Größeneinschränkungen werden verwendet, um das Abspalten eines atomaren Fragmentes bei einem Split anzugeben. Ein Typ, der eine solche Einschränkung enthält, wird auch *constrained Typ* genannt. Zu beachten ist allerdings, dass jede Größen-Einschränkung die Anzahl der Splits von diesem Typ um eine Größenordnung reduziert, da es die Freiheitsgrade um eines verringert.

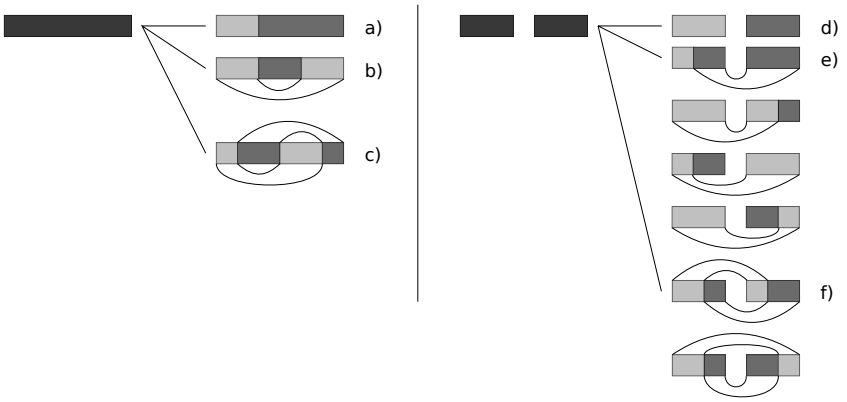

Abbildung 4.2: Die Basissplittypen, die hier betrachtet werden
Mit der oben genannten Schreibweise sind diese Splittypen folgende: a) 12, b) 121, c) 1212, d) 1G2, e) 12G2 bzw. 1G12 bzw. 12G1 bzw. 1G21, f) 12G12 bzw. 12G21.

Die Komplexität des Alignments hängt von der Anzahl der Eltern- und Kindinstanzen der betrachteten Splittypen ab. Von der Anzahl der Eltern deswegen, weil für ein gegebenes Fragment in der ersten Sequenz, *alle* Fragmente in der zweiten betrachtet werden, die den gleichen Splittyp haben. Von der Anzahl der Kinder aus dem Grund, das die Anzahl bedingt, auf wieviele verschiedene Möglichkeiten ein aligniertes Fragment in Teilalignments zerlegt werden kann. Diese Anzahlen hängen von der Länge m der zweiten Sequenz ab und sind wie folgt definiert:

Definition 4.7. Die **Anzahl der Kinder**

$$\#_C^m(T) = \left|\left\{(F^1, F^2) \mid (F^1, F^2)\text{ist ein T-Split von } F \text{ und } \hat{F} \subseteq [1, m]\right\}\right| \tag{4.1}$$

Definition 4.8. Die **Anzahl der Eltern**

$$\#_P^m(T) = \left|\left\{F \mid \exists(F^1, F^2)\text{der ein T-Split von } F \text{ ist und } \hat{F} \subseteq [1, m]\right\}\right| \tag{4.2}$$

Lemma 4.1. Für Sequenzen mit der Länge m und einem Splittyp T seien die Grade der Eltern und Kinder k_p,k_1 und k_2. Weiterhin gibt c die Anzahl der Freiheitsgrade an, die durch die Einschränkungen von T reduziert wurden und $c' \leq c$ sei die entsprechende Reduzierung der Elterninstanzen. Dann gilt $\#_C^m(T) \in O(m^{k_p+k_1+k_2-c})$ und $\#_P^m(T) \in O(m^{2k_p-c'})$.

Einen ausführlichen Beweis für dieses Lemma findet sich im Originalpaper [MWB09].

4.2 Funktionsweise

Als Input erhält der Alignmentalgorithmus zwei kantenannotierte Sequenzen (S_a, P_a) und (S_b, P_b), sowie einen Parsetree für die Sequenz (S_a, P_a)[2]. Für jedes Fragment des Parsetrees (also jeden inneren Knoten und die Wurzel) berechnet der Algorithmus rekursiv Alignments mit allen Fragmenten von (S_b, P_b), die den selben Basistyp haben. Um eine präzise Darstellung der Rekursion zu erhalten, braucht man folgende formale Notation für das Alignment von Fragmenten:

Definition 4.9. Die Einschränkung eines Alignments A auf Fragmente F_a und F_b ist definiert als:

$$A\,|_{F_a \times F_b} := \left\{(i,j) \in A \mid i \in \hat{F}_a \cup \{-\}, j \in \hat{F}_b \cup \{-\}\right\} \tag{4.3}$$

A aligniert dabei zwei Fragmente F_a und F_b mit dem selben Grad k (kurz: $\text{align}_A(F_a, F_b)$), genau dann, wenn für alle $(a_1, a_2) \in A$ und $i \in 1 \ldots k$ gilt, dass $a_1 = -$, oder $a_2 = -$ oder aber $a_1 \in F_a[i] \Leftrightarrow a_2 \in F_b[i]$. Zu beachten gilt es, dass für ein gegebenes Alignment A ein Fragment der einen Sequenz mit verschiedenen anderen aus der zweiten Sequenz aligniert werden kann.

Definition 4.10. Die optimalen Kosten um zwei Fragmente miteinander zu alignieren sind definiert als $C(F_a, F_b) := \min_{\text{A mit } \text{align}_A(F_a,F_b)} \{C_A(F_a, F_b)\}$. $C_A(F_a, F_b)$ sind dabei die Kosten um die beiden Fragmente zu alignieren im Alignment A [MWB09].

Das Alignment geht dabei die durch den Parsetree gegebene Zerlegung der ersten Sequenz rekursiv durch. Dies macht am Besten die folgende Rekursionsgleichung deutlich, die die Kostenberechnung an den inneren Knoten repräsentiert:

Lemma 4.2. Seien F_a und F_b Fragmente von (S_a, P_a) bzw. (S_b, P_b). Sei weiter (F_a^1, F_a^2) ein *arc-preserving Split* von F_a mit dem Basistyp T, so gilt für die Kosten des Splits

$$C(F_a, F_b) = \min_{\text{T-Split}(F_b^1, F_b^2) \text{von} F_b} \left\{C(F_a^1, F_b^1) + C(F_a^2, F_b^2)\right\} \tag{4.4}$$

Der Parsetree zerlegt dabei F_a in (F_a^1, F_a^2) durch einen Split des Basistyps T.

Die vorangestellte Gleichung 4.4 gibt an, wie an jedem Knoten des Parsebaumes die Kosten für den besten Split berechnet werden. Der Knoten gibt dabei eine feste Zerlegung des Fragments F_a in (F_a^1, F_a^2) vor. Hierfür müssen nun alle möglichen Zerlegungen des

[2] Mit den Möglichkeiten, einen solchen Parsetree aufzubauen, beschäftigt sich die Arbeit in den weiteren Kapiteln

Fragments F_b in (F_b^1, F_b^2) betrachtet werden, die den selben Splittyp aufweisen, wie der Split von F_a. Von all diesen möglichen Splits wird der günstigste gewählt.

Ist diese Rekursion an den Blättern des Baumes angekommen, so greift eine andere Funktion zur Kostenberechnung. Diese findet sich in [MWB09], ebenso wie ein Beweis des Lemma 4.2. Bei der Rekursion verwendet der Algorithmus Tabellen, um die Zwischenwerte aller $C(F_a, F_b)$ zu speichern. So wird jede Instanz nur einmal berechnet. Es erfolgt also eine dynamische Programmierung. Das aktuelle Alignment kann mit Hilfe gewöhnlicher Backtrace-Techniken ermittelt werden.

Werden allerdings die im Abschnitt *der Splittyp* genannten Größeneinschränkungen berücksichtigt, so ändert sich auch die Rekursionsgleichung. Wie diese dann aussieht, zeigt Formel 4.5. Die Ergänzung der Rekursionsgleichungen wird notwendig, wenn Längeneinschränkungen auf eine Intervalllänge 1 mit betrachtet werden. Abbildung 4.3 zeigt hierfür ein Beispiel. Wird eines der Fragmente des Splits auf eine Intervalllänge 1 eingeschränkt (siehe F_a^2 in der Abbildung), müssen für den entsprechenden Split in der zweiten Sequenz F_b^2 durchaus längere Intervalle betrachtet werden. Dies muss auf Grund der Tatsache geschehen, dass auch Teile der Sequenz auf *gaps* aligniert werden. Genau für diese Fälle sind die zusätzlichen Fälle der Rekursion.

$$C(F_a, F_b) = \min_{\text{T-Split}(F_b^1, F_b^2)\text{von}F_b} \min \begin{cases} C(F_a^1, F_b^1) + C(F_a^2, F_b^2 \\ C(F_a, F_b^2) + C(-, F_b^1) \\ \qquad\qquad \text{wenn } T \text{ Fragment 1 einschränkt} \\ C(F_a, F_b^1) + C(-, F_b^2) \\ \qquad\qquad \text{wenn } T \text{ Fragment 2 einschränkt} \end{cases} \tag{4.5}$$

4.3 Komplexität

Für die Analyse der Komplexität sei nun im Folgenden n die Länge der ersten Sequenz und m die Länge der zweiten Sequenz. Weiterhin sei k der höchste Grad, den ein Fragment annehmen kann (hier also 2). Wie in der Rekursionsgleichung 4.4 zu sehen, müssten eigentlich alle gültigen Fragmente F_a und F_b betrachtet werden. Durch die hier getroffene Einschränkung auf den maximalen Grad k hat jedes Fragment also maximal $2k$ Boundaries. Das bedeutet, dass sich die Anzahl der Fragmente in $O(n^{2k})$ bzw. in $O(m^{2k})$ bewegt. Das Alignment betrachtet nun aber nicht alle gültigen Splits von F_a

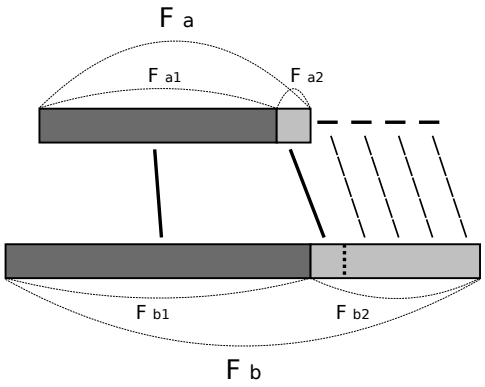

Abbildung 4.3: Spezialfall mit Einschränkungen der Länge

in (F_a^1, F_a^2), sondern nur diese, die durch den Parsetree vorgegeben sind. Diese Anzahl entspricht also der Anzahl der Knoten in dem Parrsetree und liegt somit in $O(n)$.

Die Komplexität des Speicherplatzes hängt von der Anzahl der Fragmente ab, für die Kosten abgelegt werden müssen. Aus obiger Überlegung lässt sich ableiten, dass der für das Alignment benötigte Speicherplatz in $O(nm^{2k})$ liegt.

Für die Abschätzung der Zeitkomplexität sieht dies etwas anders aus. Für jedes dieser abgespeicherten $O(nm^{2k})$ Fragmente müssen alle möglichen (F_b^1, F_b^2) betrachtet werden. Da jedes dieser Fragmente wieder maximal Grad k besitzt, würden sich weitere $4k$ Boundaries ergeben. Dies trifft allerdings nicht zu, da diese durch das Elternfragment F_b beschränkt werden, welches durch Zusammensetzung der beiden Kindfragmente entsteht. Von den $4k$ neuen Boundaries sind schon $2k$ durch das Elternfragment festgelegt. Bleiben also weiter $2k$ Boundaries, wobei hier jeweils wieder 2 zusammenfallen, bedingt durch den Split. Hieraus ergibt sich, dass für jedes Fragment, das abgespeichert wird, wieder k Möglichkeiten betrachtet werden müssen, was in einer Zeitkomplexität resultiert, die in $O(nm^{3k})$ liegt.

Hierbei handelt es sich um eine *worst-case* Abschätzung. Es lässt sich sehen, dass wenn der Split von F_a in (F_a^1, F_a^2) einen einfachen Typ hat, an jedem Knoten weniger Berechnungen notwendig sind, da es weniger mögliche Instanzen gibt. Die angesprochenen Einschränkungen die Längen der Intervalle betreffend setzen diese Komplexität noch einmal herab, da dann weniger Boundaries frei wählbar sind.

Auf Grund der ermittelten Komplexität des Alignmentalgorithmus ist es wichtig, eine Zerlegung der ersten Sequenz zu finden, welche diese Komplexität so niedrig wie möglich hält. Das folgende Kapitel 5 beschreibt einen solchen Parser.

5 Ein vollständiger Parser

Bei dem Ansatz des vollständigen Parsers geht es darum, eine optimale Zerlegung der Sequenz zu finden. Dies geschieht durch Berechnung aller möglichen Fragmente[1] und deren jeweiliger binärer Zerlegung (im nachfolgenden Splits) in Kindfragmente. Optimal bedeutet in diesem Sinne, dass dadurch die kürzest mögliche Laufzeit des Alignments erreicht werden kann. Realisiert wird diese Optimierung über die Minimierung auftretender Kosten. Jedes Fragment und der jeweils dazugehörende Split besitzt einen Kostenwert, der sich an der Anzahl der freien bzw. optimierten Boundaries orientiert. Dies führt zu einem elementaren Baustein des vollständigen Parsers, der Kostenfunktion.

5.1 Kostenfunktion

Wie bereits aus dem Kapitel mit der Alignmentbeschreibung hervorgeht, hängt die Komplexität des späteren Alignments direkt davon ab, wie die Sequenz unterteilt wird. Aus diesem Grund ist es wichtig, einen möglichst günstigen Parse zu finden. Diese Tatsache muss nun bei der Bewertung der Zerlegungen während des Parses und des Backtraces berücksichtigt werden und möglichst realistisch abgebildet werden. Es kommt also darauf an, welche Kosten genau an einem Knoten des Baumes entstehen. Ein durch einen solchen Knoten gegebener Split hat die Kosten, die an genau diesem Knoten entstehen und zusätzlich die, die durch die darunter liegenden Teilbäume gegeben sind.

Allgemein setzen sich die Kosten für ein Fragment $cost_F$ mit festem Split zusammen aus den Kosten des Splits $F = (F^1, F^2)$ sowie den Kosten der beiden Kindfragmente F^1 und F^2. Formal ist dies:

$$cost_F = \underbrace{cost_{Split\ von\ F\ in\ (F^1,F^2)}}_{\text{Kosten an diesem Knoten}} + \underbrace{cost_{F^1}}_{\text{Kosten 1. Teilbaum}} + \underbrace{cost_{F^2}}_{\text{Kosten 2. Teilbaum}} \tag{5.1}$$

[1] wie im vorherigen Kapitel festgelegt, beschränkt sich auf die Betrachtung auf Fragmente mit Grad 1 oder Grad 2

Da an jedem dieser Knoten wie im Kapitel 4 beschrieben alle möglichen Splits des gleichen Typs betrachtet werden müssen, sind die exakten Kosten also genau so groß wie die Anzahl solcher Splits, es gilt also:

$$cost_{Split\ von\ F\ in\ (F^1,F^2)} = \#_C^m \tag{5.2}$$

m sei hier wieder die Länge der zweiten Sequenz und T der Typ des Splits von F in (F^1, F^2).

Die Anzahl $\#_C^m$ ist allerdings nur mit erheblichem Aufwand zu berechnen. Hierfür müsste die zweite Sequenz ähnlich untersucht werden, wie die erste. Da dies zusätzlich sehr rechenintensiv ist, wurde für diese Arbeit die Entscheidung getroffen, diese Anzahl abzuschätzen, was allerdings ziemlich genau getan werden kann. Eine erste Möglichkeit der Abschätzung wäre die Komplexitätsklasse anzunehmen, in der sich die obige Anzahl bewegt. Da hier maximal Fragmente mit Grad 2 betrachtet werden, die also höchstens 4 Intervallgrenzen haben, wäre die Anzahl in $O(n^4)$. Die Kostenabschätzung unter dieser Annahme wäre $cost_{Split\ von\ F\ in\ (F^1,F^2)} = n^4$. Dies ist aber ein zu großer Wert, der noch viel genauer abgeschätzt werden kann. Die Abschätzung sieht dabei wie folgt aus:

$$cost_{Split\ von\ F\ in\ (F^1,F^2)} = \binom{n+k-1}{k} \cdot 2^c \tag{5.3}$$

Hierbei steht n für die Länge der zweiten Sequenz, c ist die Anzahl der optimierten Boundaries und k die Anzahl der freien Boundaries. Durch die Verwendung der Binomialkoeffizienten wird die Reihenfolge der Boundaries außer Acht gelassen, welche bei der Abschätzung durch n^4 eine Rolle spielt. Dies geschieht durch das Prinzip der *ungeordneten Probe* (siehe hierzu Abschnitt 5.2.3).

Definition 5.1. Bei einem beliebigen Split können maximal 6 Boundaries auftreten (siehe Abbildung 4.2). Für jedes auftretende Intervall der Länge 1 kann die Anzahl der zu betrachtenden Boundaries um eins vermindert werden. Treten zwei solcher Intervalle direkt hintereinander auf, ist dies nur einmal möglich. Diese Boundaries werden **optimierte Boundaries** c genannt.

Definition 5.2. Die Anzahl der **freien Boundaries** k ergibt sich wie folgt: $k =$ Anzahl der Boundaries $- c$.

5.2 Funktionsweise

Die Funktion des vollständigen Parsers beruht auf zwei grundlegenden Blöcken:

1. Berechnung des Parses (die Kosten aller möglicher Fragmente)
2. Backtracke zur Bestimmung der endgültigen Aufteilung

5.2.1 Der Parse

Im Kapitel zur Klärung des algorithmischen Hintergrundes wurden Parser erklärt, die meist dazu verwendet werden, Grammatiken zu parsen. Das Konzept, das hinter diesem Parser steckt ist dabei etwas anders. Die herkömmlichen Ableitungsregeln gibt es nicht, da ja nicht versucht wird, eine Grammatik zu parsen. Hier ist die Zerlegung einer Sequenz gefragt und die einzige Bedingung die hier gilt, ist die, dass alle entstandenen Fragmente *arc completed* sein müssen. In gewissem Sinne könnte man dies als Regel des Parsers verstehen.

Für alle weiteren Erklärungen gehen wir davon aus, dass n die Gesamtlänge einer Sequenz angibt, während die Positionen innerhalb einer Sequenz bei 0 beginnend gezählt werden. Da es sich hier um einen bottom-up-Ansatz handelt, werden erst die kleinen Fragmente betrachtet, erst später dann die größeren. Wie bereits erwähnt, werden um eine optimale Entscheidung treffen zu können, alle möglichen Kindfragmente des jeweiligen Elternfragmentes betrachtet. Daher ist es für diesen Parsealgorithmus überaus wichtig, dass alle Fragmente, die zur Betrachtung notwendig sind, im Vorfeld bereits berechnet wurden.

Ziel des Parsers ist es, eine optimale (also günstigste) Zerlegung der ersten Sequenz zu finden. Dies geschieht rekursiv anhand folgender Definition:

Definition 5.3. Die **optimalen Kosten** für ein Fragment und seinen optimalsten Split $cost_F^{opt.}$ ist definiert durch:

$$cost_F^{opt.} = \min_{\text{über alle gültigen Splits } (F^1,F^2)} cost_{Split\ von\ F\ in\ (F^1,F^2)} + cost_{F^1}^{opt.} + cost_{F^2}^{opt.} \tag{5.4}$$

Definition 5.4. Der Split von F in die Kindfragmente (F^1, F^2) ist dann **gültig**, wenn F^1 und F^2 *arc-completed* sind.

Die Kosten für den gesamten Parse, also die komplette Zerlegung der Sequenz S erhält man, wenn man als Fragment F das Fragment annimmt, welches die gesamte Sequenz darstellt, also $F = ([1, |S|])$.

5.2.2 Backtrace

Beim Backtrace geht es nun anschließend darum, den tatsächlich günstigsten Parse zu ermitteln. Dies wird über eine rekursive Funktion realisiert, welche anfangs mit einem Fragment gestartet wird, welches die gesamte Sequenz repräsentiert. Genau wie beim eigentlichen Parsen weiter oben, werden nun für dieses Fragment wieder alle möglichen Zerlegungen und deren Kosten ermittelt. Treten hierbei die selben Kosten auf, die für dieses Fragment aus dem Parse gespeichert sind, so ist die optimale Zerlegung gefunden. In diesem Fall wird die Zerlegung in den aufzubauenden Parsetree übertragen und die Backtrace-Funktion rekursiv für die beiden nun ermittelten Kinder aufgerufen. Die Rekursion endet, wenn man bei den atomaren Fragmenten angekommen ist, welche später die Blätter in dem Parsetree darstellen.

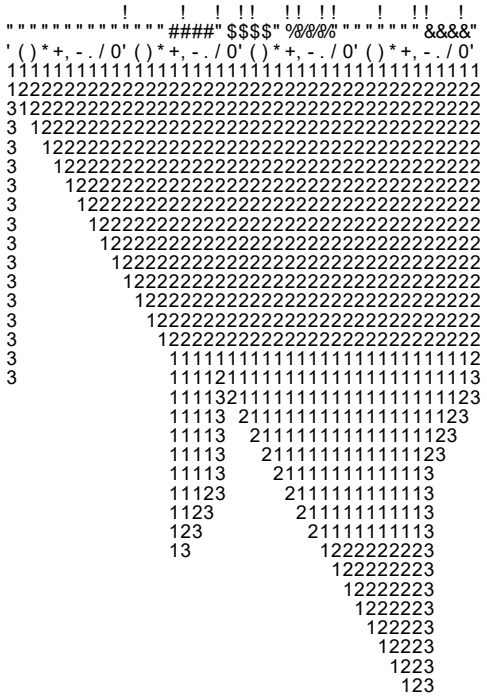

Abbildung 5.1: Beispiel eines Parsetrees als Ergebnis des vollständigen Parsers

5.2.3 Exkurs zur Kombinatorik

Weiter oben in der Kostenfunktion werden kombinatorische Hilfsmittel verwendet, um abzuschätzen, wieviele Fragmente eines Splittyps es in der zweiten Sequenz gibt. Ein solcher Typ ist in dem Fall durch die Anzahl der freien Boundaries gegeben. Dies geschieht unter anderem mit Hilfe des kombinatorischen *Ziehens*.

Aber was heißt Ziehen? Sei im folgenden N eine endliche Menge mit n verschiedenen Elementen. Aus N wird nun eine Stichprobe oder Teilmenge von k Elementen gezogen. Nun interessiert die Anzahl der verschiedenen Stichproben. Dabei kann allerdings unter *Ziehen* unterschiedliches gemeint sein:

- unter Berücksichtigung der Reihenfolge:
  - Reihenfolge wird berücksichtig: geordnete Probe
  - Reihenfolge der Ziehung wird nicht berücksichtigt: ungeordnete Probe
- Mehrfachziehungen des selben Elementes (zurücklegen):
  - möglich
  - nicht möglich

Somit ergeben sich also insgesamt vier Kombinationsmöglichkeiten. Diejenige, die hier verwendet wird, ist die der *ungeordneten Probe mit zurücklegen*. Da die Möglichkeit betrachtet wird, dass man ein und dasselbe Element mehrfach auswählt, muss die „Grundmenge“ N um die Anzahl der möglichen Mehrfachziehungen erweitert werden. Geht man davon aus, dass k Elemente aus N gezogen werden sollen, muss die Menge N um $k-1$ Elemente vergrößert werden. Es bestehen also insgesamt

$$\frac{(n+k-1)!}{k!(n-1)!} = \binom{n+k-1}{k}$$

Möglichkeiten, eine solche *k-elementige Multimenge* einer bestimmten Grundmenge zu bilden. Dies wird *Binomialkoeffizient* genannt.

5.3 Anmerkungen zur Implementation

Um die Arbeitsweise dieses Parsers besser zu verstehen, wird hier beschrieben, wie er funktioniert. Wie schon bekannt, muss die Aufzählung aller erforderlichen Fragmente von „klein“ nach „groß“ erfolgen, dabei werden alle möglichen Fragmente einer festen Länge aufgezählt. Begonnen wird daher mit allen Fragmenten der Länge 1, der Parse ist fertig wenn man bei der Länge n angekommen ist, denn dies repräsentiert die Gesamtsequenz. Innerhalb einer solchen „Längenstufe“ kommen dann erst die Fragmente mit Grad 1 dran.

Diese bestehen also nur aus einem Intervall. Dies geschieht in dem derzeitigen Ansatz so, dass dieses Fragment mit der bestimmten Länge Stück für Stück über die gesamte Sequenz „geschoben“ wird und an jeder Stelle geprüft wird, ob es gültig[2] ist. Sollte dies nicht der Fall sein, wird es solange weiter nach rechts versetzt, bis es an einer gültigen Position angekommen ist, oder aber am Ende.

Tritt dies ein, so werden die Fragmente betrachtet, die aus zwei Intervallen bestehen. Hier gibt es mehr Möglichkeiten. Die Initialisierung dieses Schritts erfolgt mit folgenden Werten (Fragmentgesamtlänge n wieder angenommen): Länge des linken Intervalls: $n-1$, Länge des rechten Intervalls: 1, Position: soweit links wie möglich, das bedeutet die erste vom linken Intervall überdeckten Base ist 0, die des rechten Intervalls liegt so, dass zwischen dem Ende des linken und dem Anfang des rechten Intervalls genau eine Base liegt. Wie vorher werden dann die Intervalle über die Gesamtsequenz „geschoben“, dies geschieht nun allerdings nach folgendem Muster:

- schiebe das rechte Intervall so lange nach rechts, bis es am Ende angekommen ist
- ziehe dann das linke Intervall so lange nach, bis zwischen ihm und dem rechten wieder genau eine Base steht
- führe diese beiden Schritte nochmal aus, beginne dabei aber eine Stelle weiter rechts und ende eine Stelle weiter links
- wenn bis hier alles durchgeführt wurde und es die Länge des Intervalls hergibt, verkleinere das linke Intervall um eine Stelle bei gleichzeitiger Vergrößerung des rechten, setze diese Intervalle wieder ganz an den linken Rand und wiederhole alle anderen Schritte, bis das linke Intervall nur noch die Länge 1 hat.

[2] Ein Fragment ist gültig, wenn es entweder *atomar* oder aber *arc-completed* ist. Siehe dazu vorheriges Kapitel.

Somit werden alle Fragmente aufgezählt und zwar in einer Reihenfolge, die für die Betrachtung der Zerlegung notwendig ist, da die beiden entstehenden Kindfragmente garantiert vorher schon einmal aufgezählt worden und somit schon berechnet worden sind.

Für jedes dieser nun erhaltenen Fragmente gilt es, sämtliche mögliche Zerlegungen in zwei Kindfragmente zu betrachten. Die Grafik zeigt alle Möglichkeiten, wie eine solche Zerlegung aussehen kann. Dabei gilt es natürlich zu beachten, dass jedes der beiden entstehenden Kindfragmente ebenfalls *arc-completed* ist. Hierzu werden die möglichen Stellen, an denen die Austeilung erfolgen soll, ähnlich wie oben beschrieben über die überdeckten Positionen geschoben, bis gültige Kindfragmente entstehen.

Für alle gefunden Kombinationen aus Elternfragment und Kindfragmenten $F = (F^1, F^2)$ werden nun die jeweiligen Kosten berechnet. Das bedeutet, dass die optimalen Kosten $cost_F^{opt.}$ für dieses Elternfragment ermittelt werden . Die Formel 5.4 zeigt, wie dies genau geschieht. Für jedes Elternfragment werden dann diese optimalsten Kosten abgespeichert. Das letzte Fragment, welches dann schließlich betrachtet wird, ist das, das die gesamte Sequenz repräsentiert. Wurden auch für dieses alle Zerlegungen betrachtet steht fest, welche Kosten der Parse über die Sequenz hat.

Für diese Variante des Parsens der Ursprungssequenz ist es zwingend notwendig, alle möglichen Kombinationen zu betrachten und zu berechnen. In den vorangegangenen Abschnitten wurde davon gesprochen, dass diese „aufgezählt“ werden. Dies wurde mit Hilfe von selbst implementierten Iteratoren gelöst. Diese kennt man in Programmiersprachen um zum Beispiel Listen der Reihe nach zu durchlaufen. Ähnlich funktionieren diese Iteratoren, nur dass diese nicht eine feste Datenstruktur durchlaufen, sondern selbst im Prinzip Werte berechnen und diese der Reihe nach zurückliefern.

Für den vollständigen Parser wurden drei solcher Iteratoren implementiert. Der erste sorgt dafür, dass alle Fragmente der Reihe nach aufgezählt werden. Der zweite Iterator berechnet für ein fest gegebenes Fragment alle möglichen Aufteilungen und der dritte liefert zu diesen Aufteilungen dann jeweils die passenden Fragmente zurück. Diese drei Iteratoren sind dabei ineinander geschachtelt, Abbildung 5.2 zeigt dies in Pseudocode.

Des weiteren müssen für jedes Fragment Kosten gespeichert werden. Diese einfach in einer Liste oder einem Vektor zu speichern, wäre zwar was den Speicherplatz angeht sehr gut, allerding macht die Zeit, die benötigt wird, bis das entsprechende Fragment gefunden wird, dies sehr ineffizient. Aus diesem Grund wurde für einen ersten Ansatz ein zwei- bzw. vierdimensionales Array gewählt, jeweils eines für Fragmente mit Grad 1 und eines für die mit Grad 2. Dies wurde durch verschachtelte Vektoren realisiert und ermöglicht den Zugriff in $O(1)$ auf die Kosten eines Fragmentes. Dies sollte sich aber bald als wenig praktikable Lösung herausstellen, wie es im nächsten Abschnitt erläutert wird.

```
while (es gibt noch Fragmente){
 while (es gibt für diese noch weitere Aufteilungen) {
  while (es gibt zur Aufteilung gültige Kindfragmente) {
   liefere diese Kindfragmente zurück;
   berechne deren Kosten;
   if(diese Kosten sind kleiner als die bisherigen für dieses Fragment)
    speichere diese;
   else
    ignoriere diese und mache im nächsten Schritt weiter
  }
 }
}
```

Abbildung 5.2: Pseudocode der Arbeitsweise der Iteratoren

5.4 Probleme - erste Verbesserungen

Bei der ersten Implementierung des vollständigen Parsers traten zweierlei Probleme auf. Wie oben erwähnt wurden für erste Tests die Kosten der Fragmente in einer 2-dimensionalen bzw. 4-dimensionalen Matrix (je nach Grad des Fragments) abgelegt, was für die kleinen Sequenzen auch schnell und effizient ist. Doch mit zunehmender Länge der Sequenzen gab es hier schnell Speicherplatzprobleme, zumal von den Matritzen nur wenige Einträge wirklich gebraucht werden. Unnötig sind zum Beispiel alle Einträge, die auf ungültige Fragmente verweisen würden und weitere Permutationen von gültigen Fragmenten. Benötigt wird also für ein Fragment $F = ([i, j], [k, l])$ nur der Matrixeintrag $M[i][j][k][l]$, nicht aber $M[j][i][k][l], M[j][k][i][l]$ und so weiter.

Dies wurde durch die Verwendung einer Hashmap[3] gelöst. Somit wird nur noch soviel Speicherplatz gebraucht, wie es gültige Fragmente gibt. Die verwendete Hashfunktion bildet dabei das Fragment anhand seiner Boundaries auf einen vorzeichenlosen 64-Bit-Integer[4] ab. Durch die Verwendung der STL[5] ist es dabei möglich, die Kosten für ein solches Fragment in logarithmischer Zeit abzurufen. Dies ist natürlich langsamer als bei der obigen Matritzenlösung, wo ein Zugriff in Konstantzeit möglich ist, allerdings werden da sehr viel unnötige Werte initialisiert. In der Praxis ist dadurch keine nennenswerte Laufzeitverschlechterung bei der Hashmap-Lösung zu beobachten.

Um diese Hashmap verwenden zu können, mussten sowohl eine Vergleichsfunktion und eine Hashfunktion implementiert werden. Die Vergleichsfunktion wird benötigt, um die Schlüssel auf Gleichheit zu testen, die Hashfunktion berechnet den Hashwert des ein-

[3] http://www.sgi.com/tech/stl/hash_map.html
[4] Datentyp unsigned long long
[5] STL = Standard Template Library

deutigen Schlüssels. In diesem Fall ist ein Fragment, welches den Schlüssel darstellt, eindeutig durch seine Intervallgrenzen gegeben. Die Hashfunktion sollte die Bedingung erfüllen, dass sie möglichst wenige Schlüssel auf den gleichen Wert abbildet. Aus diesem Grunde wurde eine Funktion gewählt, die als Schlüssel einen 64-Bit Integer Wert liefert, in dem nacheinander die Grenzen der Intervalle stehen. Einen Abdruck dieser Funktion zeigt Abbildung 5.3.

```
struct MyHash16Bit {
 size_t operator()(const fragment & f) const {
  assert (f.degree() > 0 && f.degree() <= 2);
  const int BIT_OFFSET = 16;
  unsigned long long hash = 0;
  if (f.degree() == 1) {
   hash =  ((unsigned long long )(f.get_left_boundary(0) ) )  +
           ((unsigned long long )(f.get_right_boundary(0) ) << (1*BIT_OFFSET) );
  }
  else {
   hash =  ((unsigned long long )(f.get_left_boundary(0) ))  +
           ((unsigned long long )(f.get_right_boundary(0) ) << (1*BIT_OFFSET) ) +
           ((unsigned long long )(f.get_left_boundary(1) ) << (2*BIT_OFFSET) ) +
           ((unsigned long long )(f.get_right_boundary(1) ) << (3*BIT_OFFSET) );
  }
  return hash;
  }
};
```

Abbildung 5.3: verwendete Hashfunktion

Mit Hilfe dieser Funktion ist garantiert, dass bis zu einer Sequenzlänge von 2^{16} Zeichen keine Mehrfachabbildung auf einen Schlüssel erfolgt. Die verwendete Hashmap der STL sorgt dabei selbständig durch Sortierung für eine Zugriffszeit in $O(\log n)$, was hier einen guten Kompromiss zwischen benötigter Laufzeit und verwendetem Speicherplatz darstellt.

Weiterhin ist bei diesem Ansatz des Algorithmus zu beobachten, dass durch das naive Aufzählen aller möglichen Fragmente viel Zeit verloren geht. Dies kann in manchem Fällen auch verbessert werden. Implementiert ist diese Verbesserung momentan für alle Fragmente, die Intervalle enthalten, die *arc completed* sind. In einem Vorverarbeitungsschritt werden dabei Tabellen erstellt, in denen die Positionen und Längen aller *arc completed*-Intervalle stehen. Werden also Fragmente mit Grad 1 aufgezählt, kann direkt auf diese Vorverarbeitung zugegriffen werden. Somit werden viele kleine „Schiebeschritte“ und auch die permanente Gültigkeitsprüfung eingespart. Bei Betrachtung von Fragmenten mit Grad 2 kann in manchen Fällen auch darauf zugegriffen werden und zwar immer genau dann, wenn eines der beiden betrachteten Intervalle *arc completed* ist. Dies hat nämlich laut Definition automatisch zur Folge, dass dies auch für das zweite Intervall gelten muss.

5.5 Bewertung

Der vollständige Parser liefert durch die vollständige Betrachtung aller Teilmöglichkeiten ein optimales Ergebnis. Dies müsste in einer optimalen Laufzeit des späteren Alignments resultieren. Hierzu ist aber immens viel Zeit zur Berechnung notwendig. Eine genauere Auswertung der Ergebnisse, auch im Hinblick auf das Alignment, erfolgt im übernächsten Kapitel.

PKB-Nr.	Länge	Zeit in sec	PKB-Nr.	Länge	Zeit in sec
279	21	0,390	65	46	17,250
299	25	1,000	205	48	24,985
278	29	3,203	147	51	18,328
49	30	1,312	52	52	9,218
73	33	2,390	50	59	23,266
277	37	9,437	47	61	375,562
25	37	89,953	48	61	401,625
12	40	99,468	98	62	531,046
14	40	83,937	78	62	528,343
5	41	109,968	72	67	935,922
207	45	52,453	191	113	870,281
206	45	74,547	135	116	833,765
51	46	13,765			

Tabelle 5.1: Parselaufzeiten auf dem Testsystem (siehe Kapitel D)

In Tabelle 5.1 sind für eine Auswahl an Testdaten die Laufzeiten des Parses angegeben. Betrachtet man aufmerksam die Werte in dieser Tabelle, so fällt auf, dass es vorkommt, dass die Laufzeit des Parses, bei zwei Sequenzen mit gleicher Länge oder mit nur minimaler Differenz in der Anzahl ihrer Basen, teilweise sehr unterschiedlich ist. Da dies in deren Struktur begründet liegt, sind in Abbildung 5.4 beispielhaft vier Strukturen abgedruckt. Hierbei handelt es sich bei a) um PKB 277, bei b) um PKB 25, bei c) um PKB 50 und schließlich bei d) um PKB 47. Dabei sind jeweils die linken Strukturen diejenigen, die eine weit kürzere Laufzeit aufweisen, als die rechten. Hierfür sind die relativ vielen ungebundenen Basen an den beiden Enden der Struktur verantwortlich, die den vollständigen Parser verlangsamen. Tauscht man zu Testzwecken in der Sequenz die Basenpaarungen so, dass diese atomaren Basen nach innen und dafür die Paarungen nach außen wandern, kann man schon eine große Verbesserung bemerken, was die benötigte Zeit angeht. Der Grund liegt darin, dass es in einem solchen Fall, in dem die Basen, die eine Bindung eingehen, weit auseinander liegen, weit weniger *arc completed* Fragmente in der Struktur gibt, die berücksichtigt und berechnet werden müssen, als wenn diese nahe beisammen sind. Ebenfalls aus diesem Grunde spielt die generelle Anzahl der vorhandenen Bindungen eine Rolle, denn je mehr Bindungen es gibt, desto weniger Berechnungen sind notwendig.

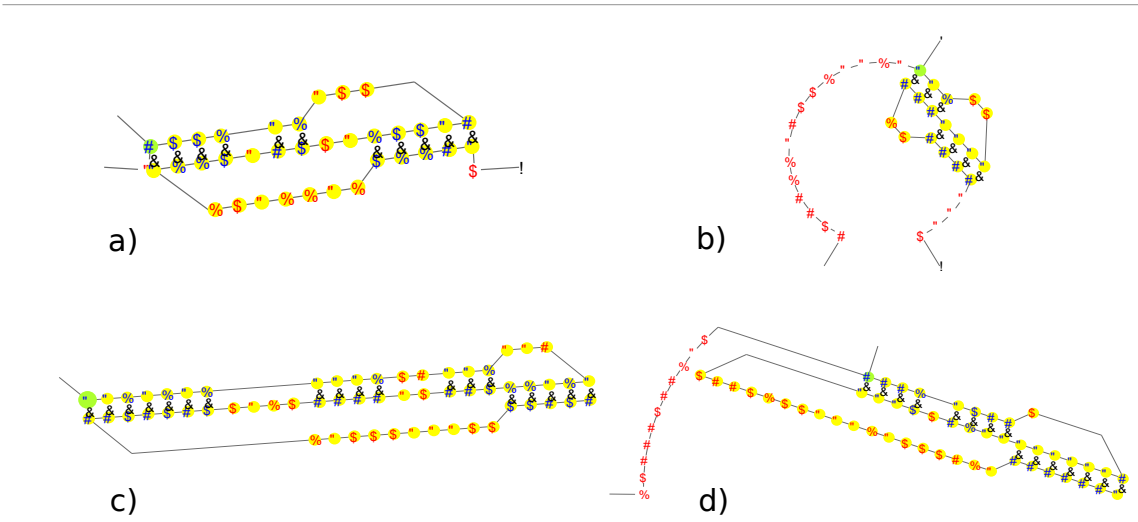

Abbildung 5.4: Strukturen mit sehr ähnlicher Länge und großen Unterschieden in den Parselaufzeiten
Darstellung der Sekundärstruktur ausgewählter Testdaten: a) PKB 277, b) PKB 25, c) PKB 50, d) PKB 47 Erstellt mit: PseudoViewer Web Service (http://pseudoviewer.inha.ac.kr/)

5.6 Analyse der Komplexität

Um einen Vergleich der verschiedenen Parsemöglichkeiten zu vereinfachen, folgt hier eine Analyse, in welchen Komplexitätsklassen sich Laufzeit und Speicherbedarf dieses vollständigen Ansatzes bewegen. Diese hängt wieder direkt von der Anzahl der möglichen Instanzen der betrachteten Fragmenttypen ab, die betrachtet werden müssen. Die Abschätzung erfolgt dadurch analog zu der in Kapitel 4.3 vorgestellten Komplexitätsanalyse.

Die Zeitkomplexität liegt für die vollständige und optimale Zerlegung in $O(n^6)$. Dies liegt daran, dass die Anzahl aller möglichen Fragmente in $O(n^4)$ liegt, und für jedes alle möglichen Splits in $O(n^2)$ betrachtet werden (siehe hierzu Kapitel 4.3). Für den benötigten Speicherplatz sieht dies anders aus. Durch die Verwendung der hashmap wird exakt soviel Speicher verbraucht, wie es Fragmente gibt. Die Anzahl der Fragmente kann man hierbei durch $\binom{n}{4} \in O(n^4)$ abschätzen, da im worst case jedes betrachtete Fragment maximal 4 Boundaries besitzt.

6 Parsen mit Heuristik

Die bisher vorgestellte Methode eine Zerlegung der Sequenz zu berechnen, liefert zwar optimale Ergebnisse, ist aber sowohl was die Laufzeit angeht, als auch speicherplatztechnisch sehr komplex. Hier stößt man bei längeren RNA Sequenzen schnell an die Grenzen dessen, was noch effektiv berechenbar ist. Die Intension ist also, eine Strategie zu entwickeln, die das Auffinden eines Parses schneller und mit weniger Speicherbedarf erledigt, als obige Variante. An dieser Stelle kommen die am Anfang erwähnten Heuristiken zum Einsatz. Das Ziel ist es, einen möglichst effektiven Parsealgorithmus zu finden, der nicht alle möglichen Fragmente und Splits berechnet, sondern versucht, möglichst einfache Entscheidungen zu treffen. Der springende Punkt hierbei ist, dass, anders als bei dem vollständigen Parser, keine vollständige Exploration des Suchraumes durchgeführt wird. Durch den Einsatz von *greedy*-Verfahren wird versucht, eine möglichst gute lokale Entscheidung zu treffen. Dadurch sinkt sowohl die Zeit- als auch die Speicherkomplexität. Hierzu gibt es zwei mögliche Strategien, die implementiert wurden. Beide wurden getestet und werden im folgenden analysiert.

6.1 Ansatz 1: bottom up

Die erste der beiden möglichen Varianten ist ein sogenannter *bottom up* Ansatz. Dieser geht von den kleinsten Fragmenten aus und arbeitet sich bis zu dem Gesamtfragment, also der kompletten Sequenz vor. Ähnlich wie der vollständige Parser arbeitet dieser Algorithmus in mehreren Stufen. In einem ersten Schritt werden alle atomaren Fragmente mit Grad 1 (also alle einzelnen Basen) zusammengefasst, sofern sie nebeneinander liegen. Als nächstes geschieht das selbe mit aufeinander folgenden Kanten, es werden also sogenannte *arc stems* gebildet. Anschließend gibt es nun drei Möglichkeiten, die der Reihe nach durchprobiert werden. Wird dabei eine der Möglichkeiten angewendet, wird wieder von vorne versucht, einen der Ansätze zu verwenden. Diese drei möglichen Bearbeitungen sind:

- Betrachte ein Fragment mit Grad 2 und versuche durch Kombination mit anderen ein Fragment mit Grad 1 zu erstellen (Einfügen eines Fragmentes mit Grad 1

in die Lücke oder Kombination mit einem Fragment vom Grad 2 wobei eines der beiden angefügten Intervalle die Lücke des ersten komplett schließt und das andere Intervall perfekt am Anfang oder Ende passt.

- Ist das nicht möglich, so versuche zumindest die Lücke im ersten Fragment zu verkleinern (also ein Grad-1-Fragment einbauen, das die Lücke nicht ganz schließt oder ein Fragment mit Grad 2, dessen eines Intervall die Lücke verkleinert und das andere direkt an den Anfang oder das Ende passt.

- Ist nichts der beiden Dinge möglich, versuche das Fragment nach außen durch anfügen von passenden Fragmenten mit Grad 1 zu vergrößern.

Bei den Möglichkeiten gibt es zweierlei zu beachten. Zum ersten ist die erste Möglichkeit günstiger als die zweite und dieser wieder günstiger als die dritte. Es wird also immer versucht, eine Möglichkeit zu wählen, die „weiter oben“ steht. Zweitens dürfen durch die Anwendung dieser Möglichkeiten per Definition nur Fragmente mit maximal einer Lücke entstehen. Die möglichen Ansätze werden solange durchgeführt, bis ein Fragment entsteht, das die gesamte Sequenz darstellt, ein separates Backtrace ist daher auch nicht notwendig.

6.1.1 Komplexität

Da die Sequenz aus n einzelnen Basen besteht, werden bei diesem Algorithmus maximal n verschiedene Fragmente betrachtet. Da keine weiteren Daten gespeichert werden und die gefunden Fragmente direkt den Parsetree bilden, liegt die Speicherkomplexität in $O(n)$. Die Zeitkomplexität liegt bei diesem Ansatz in $O(n^2)$, da für jedes der eben genannten n Fragmente versucht wird, aus den restlichen $n-1$ Fragmenten eine Möglichkeit zur Kombination zu finden.

6.2 Ansatz 2: top down

In einem weiteren Ansatz, kann ein solcher Parser auch *top down* entwickelt werden. Hierbei wird von der Gesamtsequenz ausgegangen und diese immer weiter zerlegt, bis man bei den atomaren Fragmenten angekommen ist. Genau wie der andere heuristische Ansatz versucht dieser Algorithmus lokal optimale Lösungen zu finden. Anhand der eingeführten Bewertungsfunktion und einer Analyse der Parsetrees, die der vollständige

Parser liefert, wurden hier Entscheidungen eincodiert, die den Parse merklich beschleunigen und im Vergleich zum ersten heuristischen Parser wesentlich bessere Ergebnisse liefern.

Der Aufbau des Parsetrees erfolgt hier durch eine Zerlegung der Ursprungssequenz. Ist es möglich, eine atomare Base abzuspalten, so wird dies in einem ersten Schritt durchgeführt und danach die restliche Sequenz betrachtet. Ist dies nicht mehr möglich, wird versucht, ein einzelnes Baasenpaar abzutrennen, welches durch eine Kante verbunden ist. Ist beides nicht mehr möglich, so kann man sicher sein, dass sowohl am Anfang als auch am Ende des betrachteten Fragmentes eine Klammer steht, die aber nicht zusammen gehören. Im folgenden Schritt wird also versucht, die Sequenz in zwei arc-completed Fragmente mit Grad 1 zu zerlegen. Sollte auch dies nicht möglich sein, wird das Stück mit der größten Anzahl zusammenhängender atomarer Basen als einzelnes Fragment abgespalten.

Sollte dann auch dieses nicht mehr machbar sein, wird ähnlich verfahren, wie es der vollständige Ansatz tun würde. In diesem Fall wird der erste von allen möglichen Splits gewählt, welche auch von dem vollständigen Parser in Betracht gezogen werden. Ein Vergleich auf Testsequenzen hat gezeigt, dass an dieser Stelle die Berechnung des lokalen Optimums nicht global zu einem besseren Ergebnis führt, da hier im Gegensatz zum vollständigen Parser die Informationen über die entstehenden Kindfragmente fehlen. Daher fiel die Entscheidung zu Gunsten des ersten gefundenen Splits.

Bei der hier beschriebenen Zerlegung wird dabei der Grad des betrachteten Fragmentes unterschieden. Die Abspaltung atomarer Basen und der Basenpaare kann immer erfolgen, die weiteren beschriebenen Schritte werden dann aber nur bei Fragmenten mit Grad 1 durchgeführt. Wird dagegen ein Fragment mit Grad 2 betrachtet, wird sofort darauf zurückgegriffen, die Zerlegung mittels der im vollständigen Parser implementierten Methode zu berechnen. In Abbildung 6.1 finden sie eine graphische Darstellung hierzu.

Definition 6.1. Die Funktion **grad(F)** sei eine Funktion, die zu jedem Fragment F dessen Grad, also die Anzahl der Intervalle, zurückliefert.

Die Funktion des Parsers kann man auch etwas formaler in einer Rekursionsschreibweise darstellen (siehe Gleichung 6.1). Sind mehrere Möglichkeiten denkbar, wird sich für die erstgenannte entschieden.

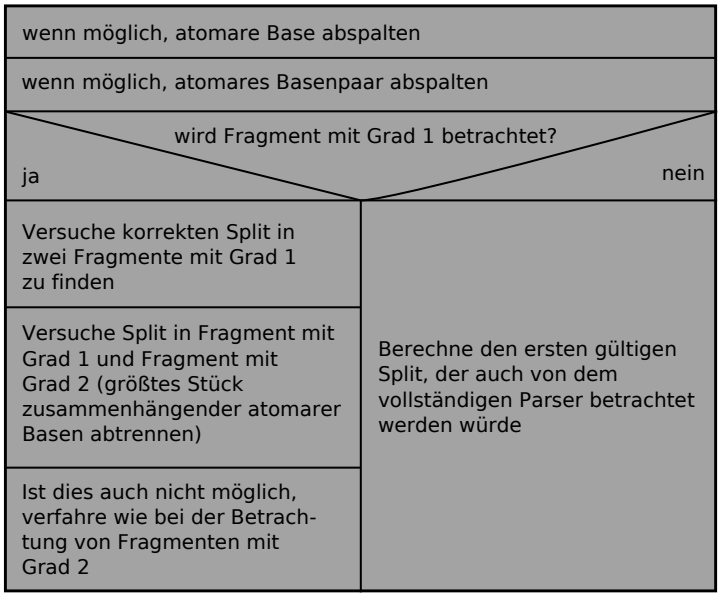

Abbildung 6.1: Verarbeitung des betrachteten Fragmentes
Die Entscheidungsreihenfolge anhand derer der heuristische Parser mit top-down-Ansatz die Zerlegung des jeweils gerade berachteten Fragmentes berechnet.

$$\text{zerlege } F \text{ durch } \begin{cases} \text{Abspaltung einer atomaren Base} \\ \text{Abspaltung einer atomaren Kante} \\ \text{Aufteilung in zwei Fragmente mit Grad 1} \\ \qquad \text{if grad(F)} = 1 \\ \text{Aufteilung in ein Fragment mit Grad 1 und eines mit Grad 2} \\ \qquad \text{if grad(F)} = 1 \\ \text{andere mögliche Zerlegung (erste gültige, die der} \\ \qquad \text{vollständige Ansatz betrachten würde)} \end{cases} \tag{6.1}$$

6.2.1 Komplexität

Genau wie bei dem anderen vorgestellten Ansatz werden auch hier maximal n Fragmente betrachtet und in einen Parsetree umgesetzt. Dies hat zur Folge, dass die Speicherkomplexität auch hier in $O(n)$ liegt. Allerdings werden hier nicht Fragmente miteinander

kombiniert sondern welche nach festen Regeln zerteilt. Dies resultiert in einer Zeitkomplexität von $O(n)$, was eine Klasse schneller ist, als der bottom-up-Ansatz.

6.3 Vergleich beider Ansätze

Um die beiden Arten des heuristischen Parsers miteinander vergleichen zu können, wird auf die beim vollständigen Parser eingeführte Kostenfunktion zurückgegriffen. Da hier ja während des Parsevorgangs keinerlei Kosten berechnet werden, werden diese dann später anhand des entstandenen Parsetrees rekursiv ermittelt. Um einen objektiven Vergleich zu erhalten, werden beide Heuristiken auf den gleichen Satz Testdaten angewendet (eine Übersicht über die verwendeten Testdaten findet sich im Anhang). Um eine gut darstellbare Graphik zu erhalten, werden die Kosten bzw. die Güte im Verhältnis zu denen des optimalen Parses angegeben. Wie sich aus Abbildung 6.2 erkennen lässt, gibt es doch beträchtliche Unterschiede in den Kosten, die von den beiden verwendeten Heuristiken verursacht werden (je höher die Kosten, desto kleiner die Güte).

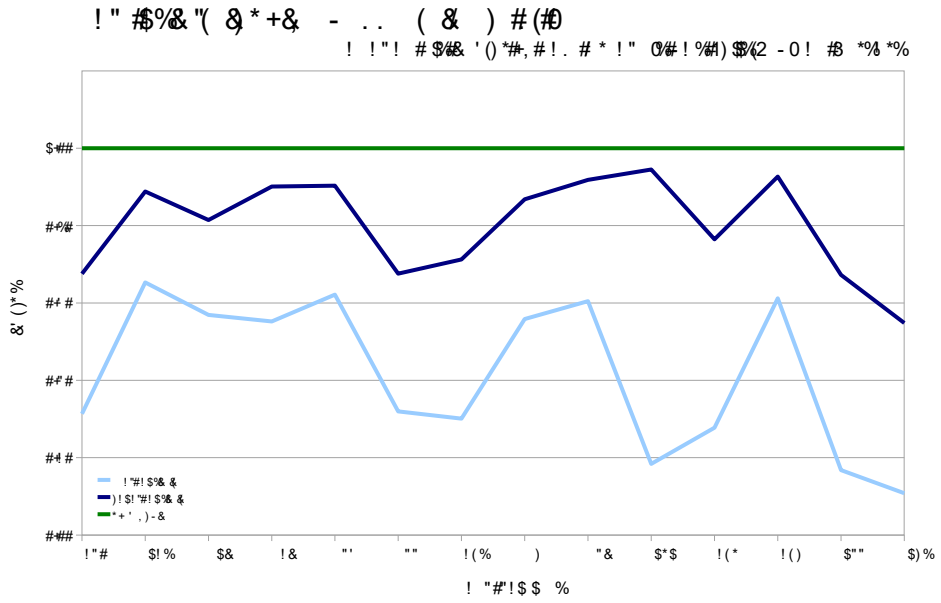

Abbildung 6.2: Vergleich der Güte beider heuristischen Ansätze
Die Abbildung zeigt den Unterschied in der Güte der beiden Ansätze. Als Referenz ist mit dem Faktor 1 die Güte des optimalen Parsers angegeben. Die Güte ist hier durch den Kehrwert der Kosten angegeben.

Durch die unterschiedlichen Ansätze unterscheiden sich die entstehen Kosten und natürlich auch die daraus resultierenden Parsetrees. Die Bäume aus dem verbesserten Parser haben eine deutlich größere Tiefe, da wie oben beschrieben bevorzugt atomare Basen abgespalten werden. Einen beispielhaften Vergleich zweier Parsebäume der selben Ursprungssequenz zeigt Abbildung 6.3.

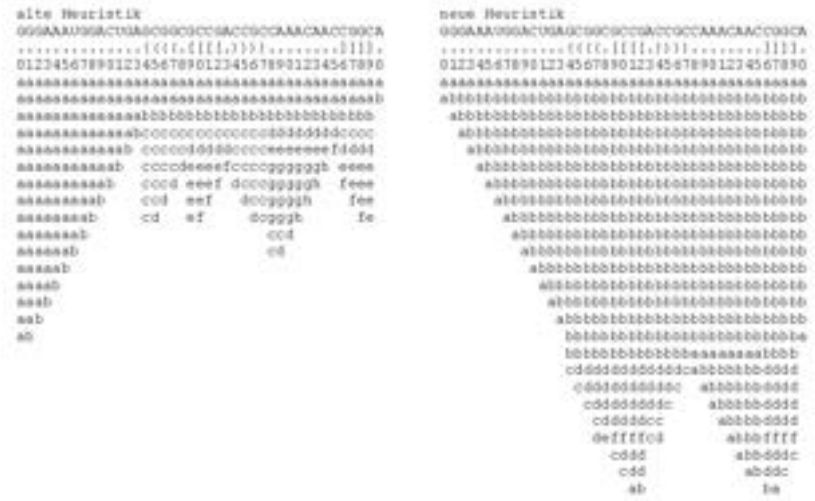

Abbildung 6.3: zwei unterschiedliche Parsebäume
Zu beachten ist, dass der durch den zweiten heuristischen Parser entstandene Parsetree deutlich mehr Ähnlichkeit zu dem hat, den der vollständige Parser erzeugt (siehe Abbildung 5.1).

6.4 Bewertung

Beide Ansätze sind wie beschrieben sehr effektiv was Laufzeit und Speicherverbrauch angeht. Allerdings liefern sie kein optimales Ergebnis (abgesehen von besonderen Fällen), was zu einer größeren Laufzeit des Alignmentalgorithmus führen kann. Eine Übersicht über die verursachten Kosten auf einem Testdatensatz zeigt Abbildung 6.2. Aus den Ergebnissen des Vergleiches beider Parsevarianten lässt sich eine Entscheidung zu Gunsten des top-down-Ansatzes vornehmen. Begründet wird dies vor allem durch die geringeren Kosten, da die Laufzeit beider im Vergleich zum Alignment kaum ins Gewicht fällt. Wird also in den folgenden Kapiteln auf die heuristische Variante verwiesen, so ist wenn nicht explizit anders angegeben die verbesserte Variante gemeint. Auf die Möglichkeiten, diese Heuristik vielleicht weiter zu verbessern, wird zu einem späteren Zeitpunkt noch eingegangen.

7 Ergebnisse

Die verschiedenen Ansätze für die Zerlegung einer Sequenz wurden auf Testdaten getestet. Diese wurden aus der PseudoBase[1]-Datenbank entnommen und so gewählt, dass sie auf der Testumgebung noch berechenbar waren. Aufgrund der hohen Komplexitätsklassen von Alignment und vollständigem Parse mussten diese auf gewisse Längen eingeschränkt werden. Im Anhang findet sich eine Übersicht aller Testdaten und auch Spezifikationen zur Testumgebung. Ziel dieses Abschnittes ist ein genauer Vergleich beider Parser hinsichtlich ihrer zur Berechnung benötigten Zeit und deren Auswirkungen auf das spätere Alignment. Es wird auch untersucht, wie gut die gefundenen Parsetrees sind. Was das berechnete Alignment angeht, ist es nicht erheblich, welcher Parser verwendet wird, da der Alignmentalgorithmus das selbe Resultat liefert. Daher kommt es auf die Zeiten an, die das Parsen der einen und das Alignieren beider Sequenzen benötigt.

7.1 Reine Parselaufzeiten

In einem ersten Schritt erfolgt nun ein Vergleich der reinen Laufzeiten beider Parsealgorithmen. In der Abbildung 7.1 sind die beiden Zeiten dargestellt. Da sich die Werte sehr in ihrem Betrag unterscheiden, wurde für die beiden eine logarithmische Einteilung der Werte-Achse gewählt. Trotzdem ist der Graphik zu entnehmen, dass die Laufzeit des heuristischen Parseansatzes um Potenzen kleiner ist, als die der vollständigen Variante. Im folgenden kann also die Zeit, die der heuristische Ansatz zum Aufbau des Parsetrees benötigt, vernachlässigt werden.

Im nächsten Abschnitt wird die „Güte“ der beiden Varianten untersucht, ähnlich wie dies in Abbildung 6.2 bereits für die beiden verschiedenen heuristischen Ansätze gemacht wurde.

[1] http://www.ekevanbatenburg.nl/PKBASE/PKB.HTML

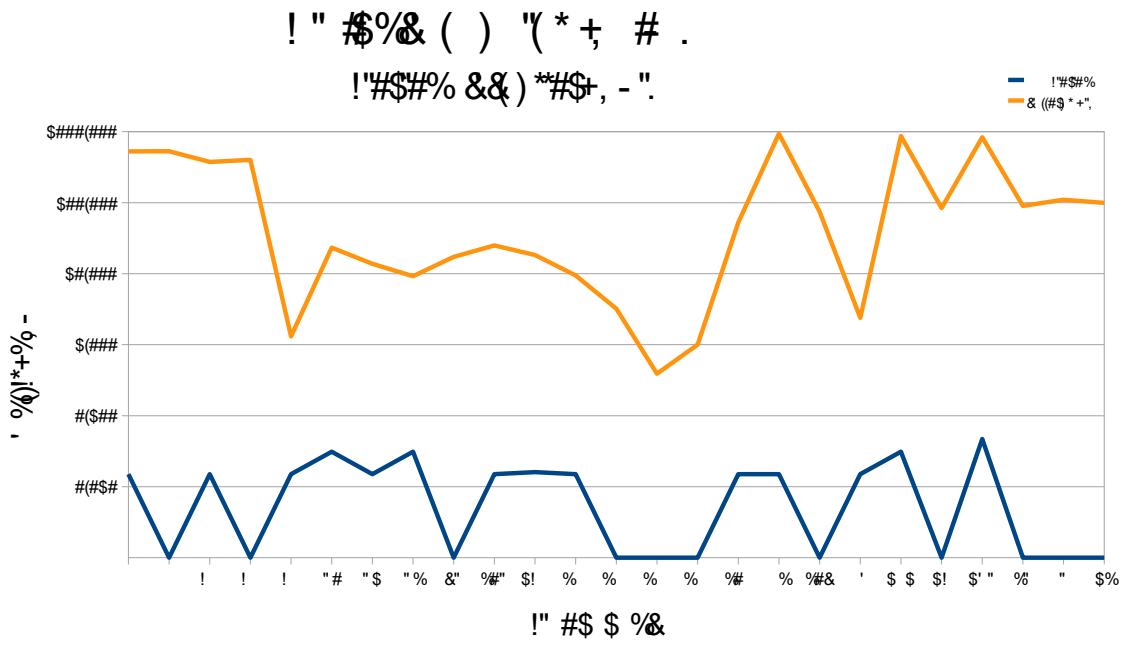

Abbildung 7.1: Vergleich der reinen Parsezeiten zwischen heuristischem und dem vollständigen Parser
Zu beachten ist die logarithmische Skala der y-Achse, die gewählt wurde, um eine Darstellung im selben Diagramm zu ermöglichen.

7.2 Bewertung des Parses

Um eine Aussage darüber treffen zu können, „wie gut“ ein gefundener Parse bzw. Parsetree ist, dient die Kostenfunktion, die im Abschnitt über den vollständigen Parser eingeführt wurde. Da diese Kosten bei dem heuristischen Ansatz während der Berechnung nicht berücksichtigt werden, werden die Kosten nach Aufbau des Parsetrees rekursiv anhand dessen Knoten durchgeführt. In folgendem Diagramm (Abbildung 7.2) ist ein relativer Kostenwert für die Kosten des heuristischen Parsers dargestellt. Dieser wird als Faktor angegeben, um wie viel die Kosten höher liegen, als die der optimalen Variante, die somit als Referenz benutzt wird (Faktor 1,0 bedeutet identische Kosten beider Parser).

Nachdem die Parsezeiten und die Güte der entstandenen Zerlegungen betrachtet wurdend, folgt anschließend die Untersuchung der reinen Alignmentzeiten, also die Zeit, die der Algorithmus bei bereits gegebenem Parse braucht, die Lösung zu errechnen.

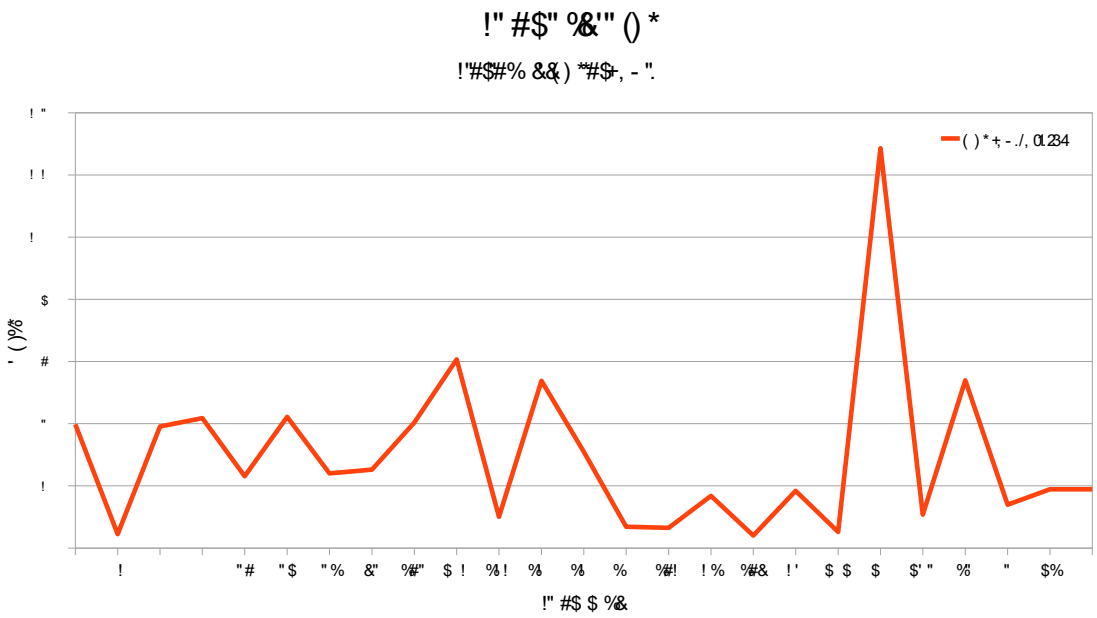

Abbildung 7.2: Vergleich der Kosten des heuristischen Parsers zu denen des vollständigen Darstellung als Faktor, als Referenz gelten die optimalen Kosten, also die, die der vollständige Parser berechnet hat.

7.3 Laufzeiten des Alignments

In diesem Abschnitt werden die Ergebnisse des Vergleiches zwischen der heuristischen Parsevariante und der vollständigen bezüglich der Gesamtlaufzeit des Alignments verglichen. Wie bereits gesehen, ist der heuristische Zerlegungsansatz immer viel schneller, als der vollständige. Was das Alignment angeht, geht man ja von der kürzesten Laufzeit bei einem optimalen Parse aus. Es stellt sich also die Frage, wann es sich lohnt, vollständig zu parsen und wann die heuristische Variante ausreicht.

In der folgenden Abbildung 7.3 sehen sie die Vergleiche für alle Testdaten[2]. Um eines gleich vorweg zunehmen: Einen Trend zeigen alle ausgewerteten Daten gleichermaßen. Wie vermutet, ist die Laufzeit des eigentlichen Alignmentalgorithmus in allen Fällen schneller, wenn der vollständige Parser benutzt wird. Teilweise lässt sich eine sehr drastische Reduzierung der Laufzeit erkennen (bis zu 50% und mehr), in anderen Fällen

[2]siehe Kapitel D, Testdaten

macht es keinen großen Unterschied. In wie weit diese Beobachtung auf die berechneten Kosten, und damit auf die Güte des ermittelten Parses zurückzuführen ist, wird später untersucht.

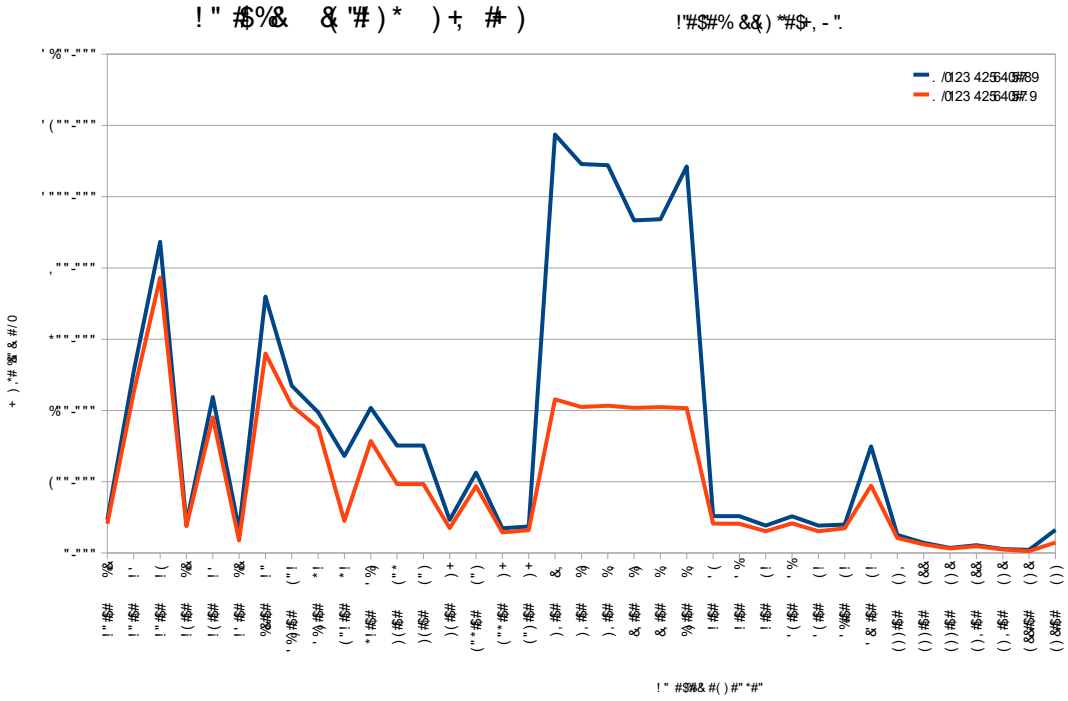

Abbildung 7.3: Vergleich der erhaltenen Ergebnisse auf allen Testdaten
(h) bezeichnet die Ergebnisse des heuristischen Parsers, (v) die des vollständigen

In Abbildung 7.4 ist auch deutlich zu erkennen, dass die berechneten Kosten tatsächlich einen Rückschluss auf die zu erwartende Laufzeit zulassen, da sich der Verlauf der Kurven sowohl für den heuristischen Fall, als auch für die optimale Variante sehr ähnelt.

7.4 Betrachtung der Gesamtlaufzeit

Nachdem bisher die reinen Parselaufzeiten und die Zeiten, die das Alignment bei gegebenem Parsetree benötigt, betrachtet worden sind, wird nun noch ein Blick auf die Gesamtzeit geworfen, die der Algorithmus für die Verarbeitung benötigt, also die Summe von Parsezeit und Alignmentzeit. Die graphische Darstellung des Vergleiches zeigt Abbildung 7.5.

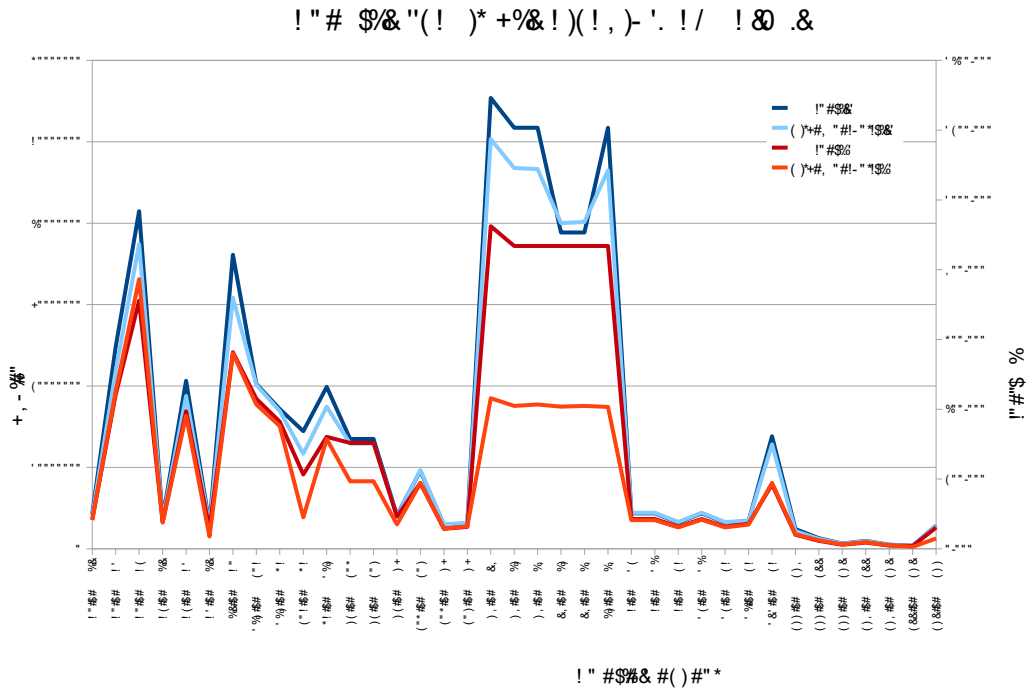

Abbildung 7.4: Zusammenhang bzw. Relation zwischen berechneten Kosten und der für das Alignment benötigten Zeit
(h) bezeichnet die Ergebnisse des heuristischen Parsers, (v) die des vollständigen

Anders als bei den bisher gesehenen Diagrammen, kann man hier kein eindeutiges Ergebnis erkennen. In einigen Fällen fällt die Zeitdifferenz beider Ansätze kaum ins Gewicht, bei anderen aber lässt sich ein extremer Unterschied erkennen, und das in diesem Fall zu Gunsten des Ansatzes mit heuristischem Parse. Doch wie ist dies weiterhin zu bewerten?

7.5 Schlussfolgerung

Nach Betrachtung des Vergleiches der Gesamtzeit lässt sich leider kein eindeutiger Trend mehr ausmachen, welcher der beiden Ansätze nun tatsächlich besser ist. Allein die Betrachtung der reinen Alignmentzeit ließe den Schluss zu, dass die Verwendung des vollständigen Parsers immer besser ist. Was die reine Alignmentzeit angeht, trifft dies auch zu. Allerdings bleibt dabei die doch sehr komplexe Laufzeit des Parsers unberücksichtigt,

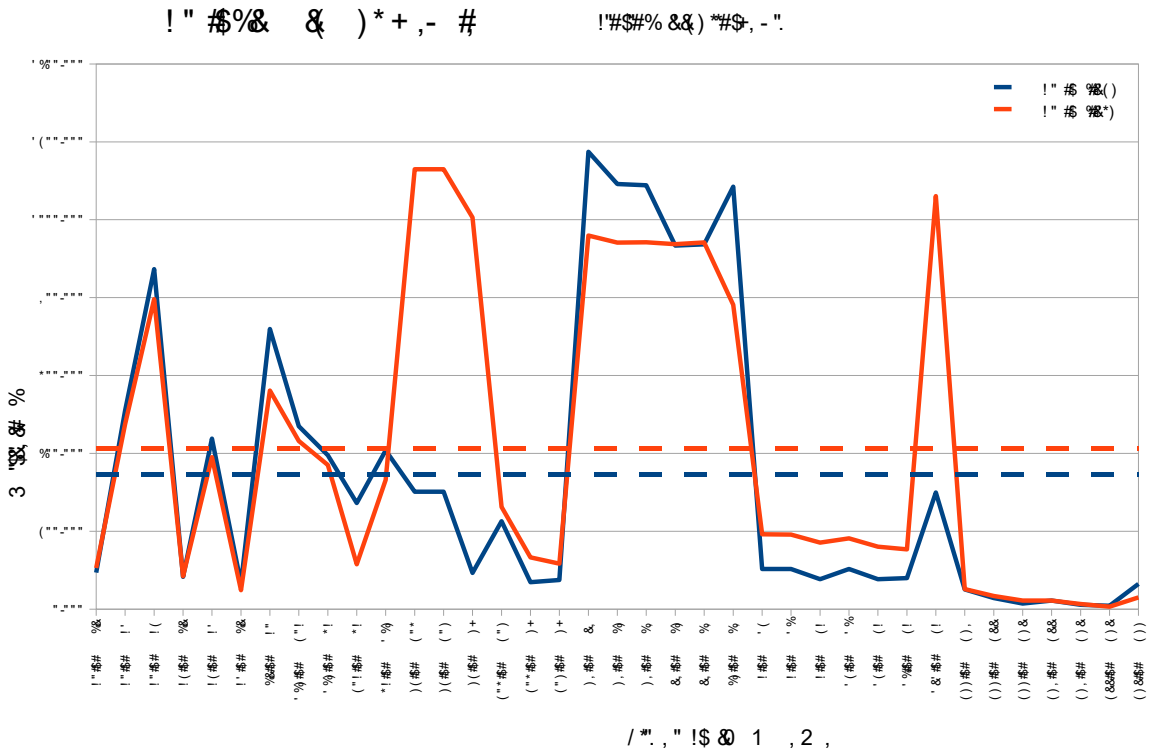

Abbildung 7.5: Vergleich der Gesamtlaufzeiten zwischen Alignment mit heuristischem Parse und Alignment mit optimalem Parse
(h) bezeichnet die Ergebnisse des heuristischen Parsers, (v) die des vollständigen. Die gestrichelt dargestellten Linien geben den Mittelwert über alle Alignments an

so dass es vorkommen kann, dass der Mehraufwand, den das Alignment mit einem nicht optimalen Parse der heuristischen Variante hat, durch die lange Berechnungszeit einer optimalen Zerlegung ausgeglichen wird. Ein Beispiel hierfür sind die Alignments der Sequenzen PKB78 und PKB98 mit den anderen Sequenzen dieser Familie. Hier ist die Zeit des Alignments optimal geparst um den Faktor 2,5 bis 3 schneller, bei der Gesamtzeit macht der Unterschied aber nur noch 10 bis 20 Prozent aus. Bei den Alignments der Sequenzen PKB72 und PKB191 macht es anderseits in der Gesamtzeit einen Unterschied um den Faktor ca. 3,5 aus, während der Zeitunterschied des reinen Alignments bei ca. 50 Prozent liegt.

In letzterem Fall würde das also bedeuten, dass es weit effektiver ist, das Alignment mit einem heuristischen Parse zu berechnen, da der Laufzeitgewinn des Alignments durch

einen optimalen Parse die Dauer dessen Berechnung bei weitem nicht ausgleichen kann. Auf der anderen Seite ist es bei den ersten angesprochenen Fällen so, dass hier der heuristische Ansatz die schlechtere Wahl wäre, da er die Zeit zur Berechnung des Alignments fast verdreifacht, so dass die Gesamtzeit doch schneller ist, wenn der vollständige Parser verwendet wird.

8 Diskussion der Ergebnisse

Die Beobachtungen der Ergebnisse aus dem Kapitel 7 werfen verschiedene Fragen auf, die nun im Anschluss diskutiert werden. Im folgenden ist immer, wenn davon die Rede ist, dass ein Parser schneller ist als der andere, die Kombination aus Parse und Alignment gemeint, also die benötigte Gesamtzeit. Sollte anderes zutreffen, wird dies explizit erwähnt.

Warum bestehen die Unterschiede zwischen beiden Parsevarianten?

Dies liegt daran, dass der heuristische Parser bei der Zerlegung der Sequenz *top down* vorgeht und damit keinerlei Wissen darüber hat, ob die Entscheidung, die er in jedem Schritt trifft (diese Entscheidungen sind fest eincodiert), im Sinne des globalen Gesamtzerlegung optimal ist. Anhand der Sequenz PKB78 macht dies Abbildung 8.1 deutlich.

Macht es einen Unterschied, ob man beim Alignment die längere Sequenz parst und gegen die kürzere aligniert oder andersrum?

Rein intuitiv muss man diese Frage mit einem „Ja“ beantworten. Betrachtet man allein den Vorgang des Parsens, so hat man ja eine Laufeit von $O(n^6)$ für eine optimale Lösung (n Länge der zu parsenden Sequenz). Also würde man meinen, dass es besser ist, die kürzere Sequenz zu parsen, da das Parsing von der zweiten Sequenz unabhängig ist. Für das Alignment ist es aber anders. Die Zeitkomplexität des Alignments liegt in $O(nm^6)$ (n Länge erste Sequenz, m Länge zweite Sequenz). Hier ist also die zweite Sequenz ausschlaggebend. Es ist also zu empfehlen, die längere der beiden Sequenzen zu Parsen und gegen die kürzere zu alignieren. Hierbei gilt: je kleiner die Längendifferenz der beiden betrachteten Sequenzen, desto weniger spielt die Reihenfolge der zu betrachtenden Sequenzen eine Rolle. Abbildung 8.2 zeigt hier exemplarisch Ergebnisse auf den verwendeten Testdaten. Dass es im mittleren Fall vorkommt, dass das Alignment der kürzeren Sequenz gegen die lange doch geringfügig schneller ist, liegt an der leicht anderen Struktur (siehe Abbildung 8.3). Eine Struktur wie bei PKB 65 ist schneller zu parsen als die andere, daher der Zeitgewinn.

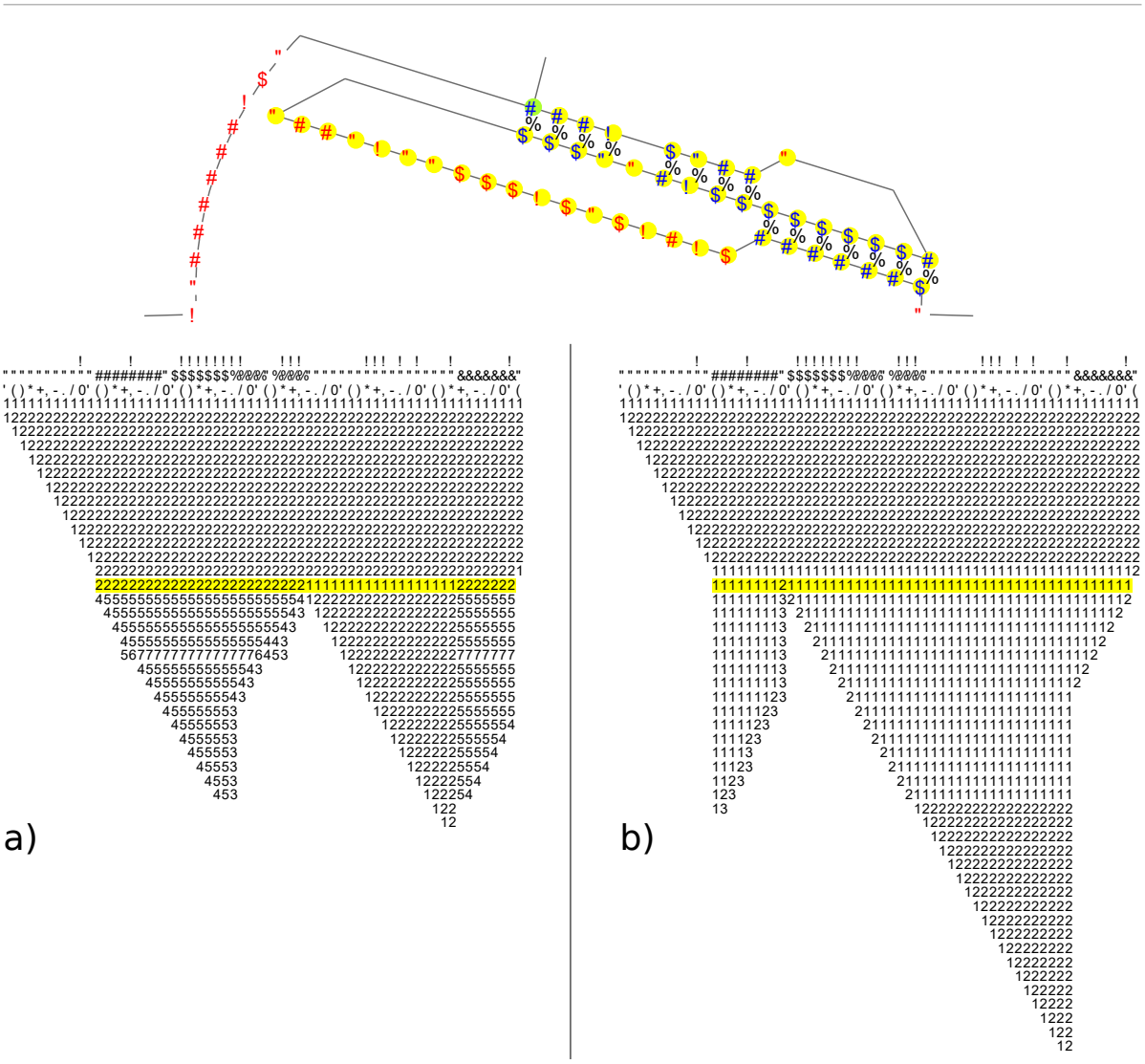

Abbildung 8.1: Struktur und Parsetrees von PKB 78
a) durch heuristisches Parsen entstandener Parsetree, b) durch optimales Parsen entstandene Zerlegung
Gelb markiert ist die Stelle, an der die unterschiedliche Entscheidung stattfindet.

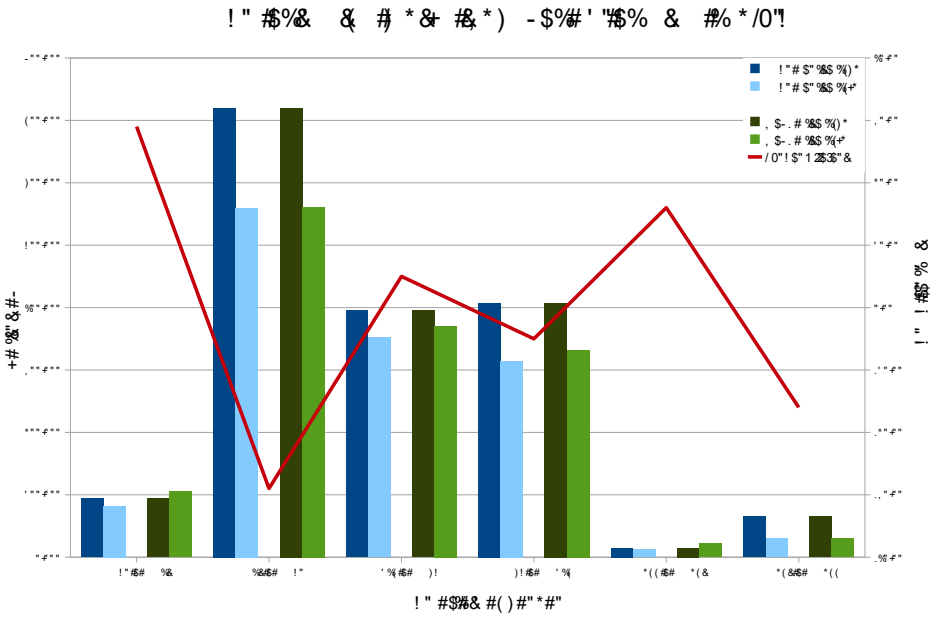

Abbildung 8.2: Vergleich der Laufzeiten bei unterschiedlicher Reihenfolge der Sequenzen Jeweils nebeneinander stehen die Zeiten des Alignments Seq. A gegen Seq. B und Seq. B gegen Seq. A. Die Linie zeigt die Längendifferenz beider Sequenzen an (auf der rechten y-Achse). Verglichen werden sowohl die reinen Alignmentzeiten, als auch die Gesamtzeit. (h) bezeichnet die Ergebnisse des heuristischen Parsers, (v) die des vollständigen

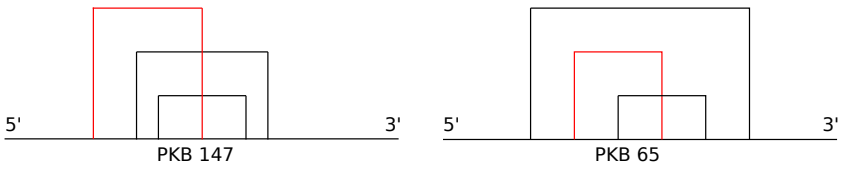

Abbildung 8.3: Einfache Strukturdarstellung der Sequenzen PKB 147 und PKB 65

Welchen Einfluss hat die Komplexität einer Struktur auf die zur Berechnung einer Zerlegung benötigten Zeit?

Um dies zu untersuchen, wurden spezielle Testdaten erzeugt, welche alle eine unterschiedliche Komplexität haben. Um möglichst andere Einflüsse auszuschließen, haben alle die gleiche Länge (30 Basen) und die gleiche Anzahl an Basenpaaren (12 Paarungen). Abbildung 8.4 zeigt verschiedene Strukturklassen und deren jeweilige Parsezeit. Auf das heuristische Parsen wurde bei diesem Vergleich verzichtet, da dessen Parsezeit nicht ins Gewicht fällt. Die Klasse a) hat dabei keinerlei überkreuzender Kanten, die Klasse e) hat deren zwei und alle anderen eine. Man könnte nun eine erste Vermutung aufstellen und behaupten, dass je komplexer die Struktur, also je mehr überkreuzende Kanten diese besitzt, desto länger dauert ihre Zerlegung. Interessanter Weise kann man aber in dem Diagramm sehen, dass es genau umgekehrt der Fall ist, also die Struktur mit der größten Anzahl überkreuzender Kanten, hat die kürzeste Laufzeit.

In diesem Fall spielt wieder der Begriff der *gültigen Fragmente* eine Rolle. Denn je mehr kreuzende Kanten eine Struktur beinhaltet, desto weniger dieser gültigen Fragmente gibt es und somit müssen auch weniger betrachtet werden. Andererseits kann man feststellen, auch wenn in dieser Graphik nicht extra erwähnt, ist es durchaus wahrscheinlich, dass die heuristische Parsevariante mit zunehmender Komplexität schlechtere Parses findet, da die Entscheidungen, die dieser Algorithmus trifft, fest einprogrammiert sind und daher global gesehen (also auf die gesamte Sequenz) zu schlechteren Ergebnissen führt.

Allerdings muss hierbei natürlich auch immer die Gesamtlaufzeit des Algorithmus berücksichtigt werden, da die Gefahr besteht, dass durch schlechte Parses die Laufzeit des Alignments explodiert. Abbildung 8.5 zeigt genau diesen Vergleich der Gesamtzeit. Hier kann man nun beobachten, dass sich in zwei Fällen die Laufzeiten kaum unterscheiden, in den anderen aber doch deutlich.

Kann schon vor der gesamten Berechnung eine Aussage getroffen werden, ob es in dem individuell vorliegenden Fall besser ist, vollständig zu parsen oder ob ein heuristischer Versuch ein hinreichend gutes Zerlegungsergebnis für schnelle Alignmentzeiten liefert?

Wie bisher gesehen, sind die Zeiten, die die beiden Parser benötigen eine Zerlegung zu berechnen, die das eigentliche Alignment benötigt und somit auch die Gesamtzeiten teils sehr unterschiedlich. Es treten Fälle auf, da ist der heuristische Ansatz schneller, in anderen aber der vollständige. Nun wäre es gut, wenn man schon vor Berechnung der Parses eine Aussage darüber treffen könnte, für welche der beiden Parsevarianten man sich nun entscheiden sollte.

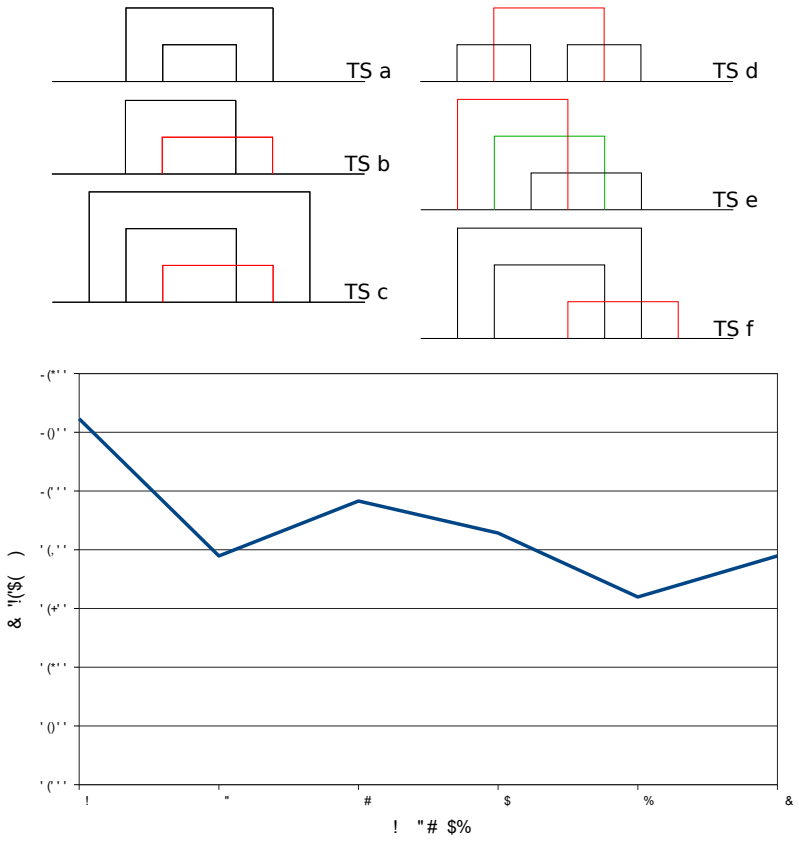

Abbildung 8.4: Verschiedene Strukturen und deren Parselaufzeit zur Bestimmg einer optimalen Zerlegung

Anhand dem direkten Kostenvergleich wäre eine solche Entscheidung einfach, allerdings muss man um diesen zu erhalten eben beide Berechnungen durchführen und somit hätte man ja keinen Zeitgewinn mehr. Es gilt also eine Strategie zu finden, die es möglich macht, eine qualifizierte Aussage zu treffen.

Wirft man einen Blick auf den Vergleich der für Parsing und Alignment benötigten Gesamtzeiten beider Parser in Abbildung 7.5, und insbesondere auf den darin gestrichelt eingezeichneten Mittelwert, so lässt sich diese Frage für die Summe der Alignments recht einfach beantworten. Betrachtet man diese mittlere Laufzeit, ist es günstiger, immer den heuristischen Parser zu verwenden, da dieser im Schnitt einen Laufzeitgewinn gegenüber dem vollständigen Parser liefert. Dies liegt daran, dass er in vielen Fällen nur wenig länger braucht als das Alignment mit vollständigem Parse, in manchen aber schneller ist. In ein paar Fällen, ist er sogar sehr deutlich schneller. Diesen Vorteil kann man aber erst mit zunehmender Anzahl von berechneten Alignments wirklich nutzen. Soll nur ein

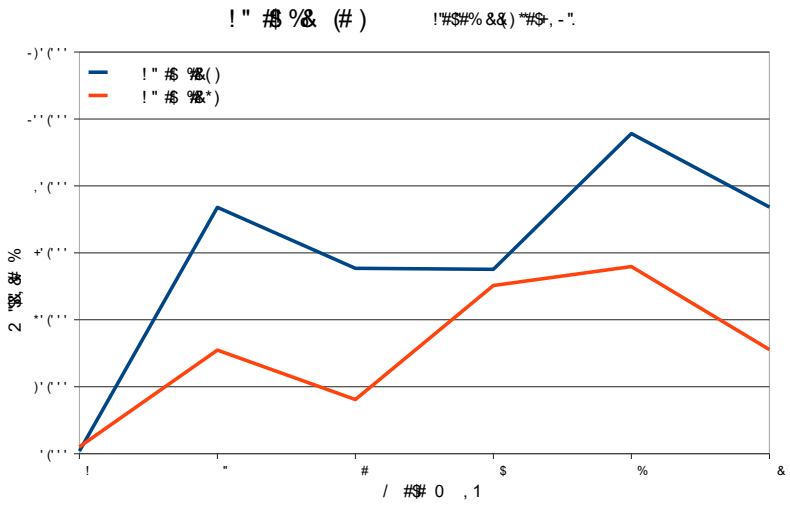

Abbildung 8.5: Gegenüberstellung der benötigten Gesamtzeiten zur Berechnung der Alignments von den Testsequenzen

einzelnes berechnet werden, wäre eine Entscheidungshilfe immer noch von Vorteil, da es doch vorkommen kann, dass der vollständige Parse im Einzelfall bessere Laufzeiten erzielt.

Was steht an Informationen vor der Berechnung zur Verfügung, die für eine Entscheidung verwendet werden können? Das sind eigentlich nur die beiden Eingabesequenzen mit allen Daten, die über sie vorliegen, also deren jeweilige Länge, die Anzahl und Positionen atomarer Basen und Basenpaare. Abbildung 8.6 zeigt einen Vergleich der Gesamtzeit auf verschiedenen Testdaten. Diese wurden anhand von Abbildung 7.5 generiert, um zu untersuchen, warum in vier Fällen ein solch großer Unterschied zwischen den beiden Zerlegungsmöglichkeiten besteht. Eine Vermutung bestand darin, dass es an der langen Teilsequenz liegt, die nur atomare Basen enthält. Auf Grund dieser Annahme wurden die Testdaten PKB72a bis PKB72f generiert (siehe Tabelle D.3). Hier wurde die lange Teilsequenz atomarer Basen innerhalb der Gesamtsequenz verschoben und auch aufgeteilt, bei PKB72e und PKB72f wurde diese Stelle mit einer Struktur gefüllt.

Hieraus ist der Schluss zu ziehen, dass dem vollständigen Parser die lange Teilsequenz ohne Struktur (also nur atomare Basen) Probleme bereitet, was dessen Laufzeit angeht. Dies bedeutet, dass in diesem Falle die durch ihn gefundene Zerlegung das Alignment nicht so viel beschleunigen kann, damit sich die Zeit zur Bestimmung des optimalen Parses rechnet. Dies kann man dann als Regel abfassen, die zumindest eine Teilantwort auf die oben gestellte Frage ist: Enthält die zu zerlegende Sequenz lange Subsequenzen ohne

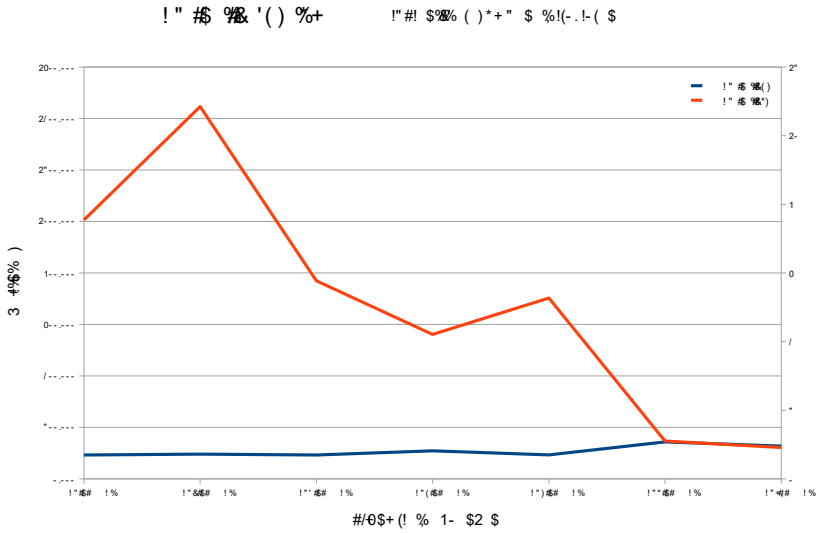

Abbildung 8.6: Vergleich von Testdaten mit leicht unterschiedlicher Struktur
h) durch heuristisches Parsen entstandener Parsetree, v) durch optimales Parsen entstandene Zerlegung

Strukturinformation (viele ungepaarte Basen am Stück), so ist eine heuristische Zerlegung mit anschließendem Alignment mit großer Wahrscheinlichkeit deutlich schneller, als ein vollständiger Parse.

Begründet ist dies wie bereits weiter oben im Abschnitt über den vollständigen Parser schon erwähnt, in dessen Funktionsweise. Treten viele aufeinander folgende atomare Basen auf, müssen sehr viele Möglichkeiten betrachtet, berechnet und auf Gültigkeit geprüft werden. Dies macht die große Laufzeit aus. Zumal nicht viele sinnvolle Entscheidungen in Betracht kommen, denn es ist sehr unwahrscheinlich, dass der optimale Split irgendwo dazwischen liegt. Die Abtrennung einer Base am Anfang oder Ende ist wahrscheinlicher.

Abschließend bleibt allerdings keine klare Antwort auf die Frage, ob vorher schon entschieden werden kann, welcher Parse die schnellere Laufzeit liefert. Zu unterschiedlich sind hier die Parameter. Dies wird hier immer davon abhängen, wie die Sequenz und deren Struktur aussieht, und wie günstig diese für eine der beiden Varianten ist.

8.1 Ausblick / weitere Verbesserungen

Der vorangegangene Abschnitt hat sich ausführlich mit den verschiedenen Ergebnissen der Untersuchungen beider vorgestellten Parseansätzen beschäftigt. Zusammenfassend kann man dazu nochmal folgendes erwähnen:

- Die Verwendung des vollständigen Parsers führt zu einer optimalen Zerlegung der ersten betrachteten Sequenz und somit auch zu einer optimalen, also der kürzest möglichen Laufzeit des Alignmentalgorithmus.
- Die Laufzeit der heuristischen Variante ist eigentlich immer vernachlässigbar klein, selbst auf langen Sequenzen.
- Da das Vorgehen des heuristischen Ansatzes fest implementiert ist, kann es zu teils gravierend schlechteren Ergebnissen kommen, die sich in langen Alignmentlaufzeiten wiederspiegeln.
- Es ist von Fall zu Fall die Entscheidung nötig, welcher Parser eingesetzt werden soll.

Um für die jeweils gerade vorliegende Situation die beste Variante wählen zu können, muss diese untersucht werden und anhand der Resultate eine Entscheidung zu Gunsten des heuristischen oder aber des vollständigen Parsers getroffen werden. Das Ziel ist es also, die Alignments vollständig zu parsen, wenn dadurch eine so gute Zerlegung gefunden wird, dass der Alignmentalgorithmus so viel schneller abläuft, dass er die Parsezeit wieder ausgleicht. Andererseits kann der heuristische Ansatz verwendet werden, wenn man davon ausgehen kann, dass er einen möglichst guten Parsebaum erzeugt, so dass das Alignment mit diesem nicht soviel länger dauert, als die Zeit, die benötigt würde, um die optimale Zerlegung zu ermitteln.

Am einfachsten wäre es natürlich, man müsste sich nicht zwischen den beiden Möglichkeiten entscheiden. In der derzeitigen Implementierung ist dies aber unumgänglich. Für die spätere Verwendung des Alignmentalgorithmus besteht aber die Möglichkeit, die beiden Parser zu verbessern. Bei dem heuristischen Ansatz betrifft dies die Güte der von ihm gefundenen Lösung, bei dem vollständigen die Zeit, die dieser zur Ermittlung der optimalen Lösung benötigt.

8.1.1 Beschleunigung der optimalen Lösung

Die Problematik liegt darin, dass alle gültigen Fragmente der Sequenz betrachtet, und für diese alle möglichen Zerlegungen berücksichtigt und bewertet werden müssen. Man müsste also das Aufzählen dieser Fragmente effektiver gestalten.

Ein erster Ansatz hierzu, der teilweise auch schon implementiert ist, ist es, Vorberechnungen durchzuführen, um darauf zurückgreifen zu können und viele „Schiebeschritte“ zu sparen. Für Fragmente mit Grad 1 wird derzeit schon eine Tabelle verwendet, in der zu jeder aktuellen Position i in der Sequenz bei gegebener Fragmentlänge j an der Stelle (i, j) die Anfangsposition des nächsten gültigen Grad-1-Fragmentes der Länge j steht. Somit wird einmal das schrittweise Schieben des betrachteten Intervalls gespart, aber auch die Zeit die immer benötigt wird, um das Fragment auf Gültigkeit zu überprüfen. Denn dieses ist ja schon im Schritt der Vorverarbeitung geschehen. In einem weiteren Schritt könnte diese Idee auch auf Fragmente mir Grad 2 ausgedehnt werden. Hier kann man durch einen weiteren Vorverarbeitungsschritt eine Tabelle erstellen, deren Einträge Paare (k, l) sind. So würde man dann aus der Tabelle in Zeile i und Spalte j auslesen können, dass wenn man das linke Intervall (i, j) betrachtet, dieses mindestens die Positionen k bis l überdecken muss. Dies wäre zum Beispiel durch linke Enden von Kanten gegeben, die in dem linken Intervall kein rechtes Ende haben. Somit müssen diese Kantenenden ja im rechten Intervall vorkommen, um ein gültiges Intervall zu erhalten.

8.1.2 Verbesserung der Heuristik

Bei der heuristischen Variante geht es nicht darum, die Berechnung zu beschleunigen, sondern die Güte des erhaltenen Ergebnisses zu verbessern. Was dies aber grundsätzlich schwierig macht, ist der *top down* Ansatz, den diese Implementierung verfolgt. Denkbar wäre zum Beispiel eine Art Kombination mit den Techniken des vollständigen Parsers, so dass zuerst einmal die fest hinterlegten Regeln angewendet werden und erst wenn dies nicht mehr möglich ist, versuchen für diesen Schritt eine optimale Lösung zu finden. Man hätte dann für ein Teilstück eine optimale Lösung und das dann auch durch die Reduktion der betrachteten Länge in vertretbarer Zeit. Dies ist natürlich erheblich von der jeweiligen Eingabe abhängig.

Ein anderer denkbarer Ansatz wäre, dem Parser eine Art Datenbank bereitzustellen, in der er für nicht eindeutige Fälle nach Vorschriften suchen kann, was wohl am Besten anzuwenden ist. So könnte man zum Beispiel mit Hilfe des optimalen Parsers eine Datenbasis bilden, worin die zu alignierenden Sequenzen nach deren Strukturen und Strukturklassen hinterlegt sind. So dass man durch die optimale Lösung Lösungsansätze

finden kann wie: „Es liegt eine Strukturklasse x vor und die Struktur sieht aus wie y, also ist es am besten, die Basen zwischen k und l abzuspalten.“

Es wäre auch denkbar, die Heuristik durch einen Backtracking-Algorithmus zu verbessern. So könnte man die Splits nicht nur auf der jeweils gerade betrachteten Ebene untersuchen, sondern ein paar Tiefenstufen weiter. Stellt man dabei fest, dass wieder viele „gute“ Regeln angewandt werden können, scheint der Weg gut zu sein, müssen wiederholt „schlechtere“ Regeln benutzt werden, macht man die bisherige Zerlegung teilweise rückgängig und versucht eine neue. Dies müsste dann über Parameter eingeschränkt werden, da ansonsten die Laufzeit wieder explodieren könnte. Ebenfalls wird hier dann wohl eine neue Bewertungsfunktion notwendig werden, um objektiv einzuschätzen, wann ein Backtracking notwenig wird oder nicht.

8.1.3 Neukonstruktion eines Parsers

Als weitere Möglichkeit könnte man auch einen weiteren Parser konstruieren, der etwas an die Funktionsweise der anderen bisher vorgestellten angelehnt ist. Wie schon bekannt gibt es Situationen, in denen eine vollständige Exploration des Suchraumes nicht von Nöten ist. Hier können bei einfachen Fragmenten greedy Entscheidungen getroffen werden, die trotz allem optimal sind. Ein einfaches Fragment würde zum Beispiel eine der folgenden Bedingungen erfüllen:

- Es handelt sich um ein Fragment mit Grad 1 mit atomaren Basen am Anfang oder Ende.
- Es handelt sich um ein Fragment mit Grad 1 mit einer Strukturkante von der ersten zur letzten Base.
- Es handelt sich um ein Fragment mit Grad 1 , das in zwei weitere 0-Loch-Fragmente aufgeteilt werden kann.
- Es handelt sich um ein Fragment mit Grad 2 , das sich in zwei 0-Loch-Fragmente aufteilen lässt.
- Es handelt sich um ein Fragment mit Grad 2 mit den Intervallen $([i, j], [k, l])$ mit einer atomaren Base an den Positionen i, j, k oder l.
- Es handelt sich um ein Fragment mit Grad 2 mit den Intervallen $([i, j], [k, l])$ mit einer Kante

- zwischen i und l,
- zwischen j und k,
- zwischen i und j,
- zwischen k und l,
- zwischen i und k oder
- zwischen j und l.

Liegt aber kein einfaches Fragment vor, muss eine vollständige Suche nach allen Möglichkeiten einer Zerteilung erfolgen und diese dann auch weiter betrachtet werden. Durch dieses Verfahren wäre Optimalität bei trotzdem sehr geringer Laufzeit garantiert. Dies kann man durch die bei jedem Split entstehenden Kosten sehen, welche sich nach der Formel 5.3 berechnen.

Als Vorbild zur Implementierung eines solchen Parsers könnte das Prinzip von bekannten *Agenda-basierten Chart-Parsern* dienen. Dieser verwendet einerseites einen Chart, der alle vollständigen bzw. optimalen Teillösungen enthält, andererseits eine sogenannte Agenda, die noch nicht optimale Teillösungen enthält. Diese müssen also nochmal betrachtet werden, da noch keine endgültige Aussage getroffen werden kann. Der ganze Algorithmus würde hier dann auch *top down* arbeiten, also die gesamte Sequenz bis in die atomaren Fragmente aufteilen. Die Fragmente und die Zerlegung, die durch Anwendung der oben genannten einfachen Fälle entstehen, würden direkt in den Chart wandern. Muss man allerdings alle möglichen Aufteilungen berechnen, so speichert man diese in der Agenda, bis man sicher sein kann, dass man für diese auch die optimale Lösung gefunden hat. Eine solche Agenda würde man wohl als *stack* implementieren während als Chart durchaus wieder eine *hashmap* dienen kann.

Hieraus ergibt sich dann noch die Frage nach der garantierten Optimalität. Rein intuitiv kann man davon ausgehen, schlussendlich beantwortet wird sie aber hier nicht. Einen Beweis dafür zu führen obliegt späteren Ausführungen und eventuellen Implementationen eines solchen Ansatzes.

9 Fazit

Der Lehrstuhl für Bioinformatik der Universität Freiburg hat einen neuen effektiveren Algorithmus zum paarweisen Sequenz-Struktur-Alignment von RNA Sequenzen vorgestellt. Dieser berücksichtigt auch vorkommende Pseudoknoten. Da das Verfahren auf einen Vorverarbeitungsschritt aufbaut (Parsing), müssen Untersuchungen vorgenommen werden, wie dieser Schritt gestaltet werden muss, um ein möglichst effektives Alignment zu ermöglichen. Mit der Entwicklung und Evaluierung solcher Möglichkeiten beschäftigte sich diese Arbeit.

Hierzu wurde sowohl ein vollständiger und somit optimaler Ansatz, aber auch zwei heuristische Ansätze analysiert. Besonderen Wert wurde dabei auf deren Laufzeit und Güte des gefundenen Ergebnisses gelegt, da dieses direkt die Komplexität des späteren Alignments beeinflusst. Der optimale Ansatz liefert dabei die optimalsten Alignmentzeiten, braucht aber selbst sehr viel Rechenzeit. Die Variante, die Heuristiken verwendet ist dagegen deutlich schneller, kann aber nicht optimale Ergebnisse liefern, wodurch sich die Zeit vergrößert, die zur Berechnung des Alignments benötigt wird.

Es gilt also mit derzeitigem Stand einen *trade-off* zwischen zwei Zeiten zu finden. Zum einen die Zeit, die für das Parsing investiert werden soll und zum anderen die, die sich beim Alignment einsparen lässt. Es zeigte sich, dass beide vorgestellten Parseansätze nutzbar sind. Welcher nun genau besser ist, hängt von Fall zu Fall von den verschiedenen Sequenzen ab. Aus diesem Grund obliegt es zukünftigen Arbeiten die schon angesprochenen Verbesserungsmöglichkeiten zu untersuchen, damit ein einzelner Parser gefunden werden kann, der sowohl schnell ist aber auch optimale Ergebnisse liefert.

Glossar

adjazent	[lateinisch] der, Anwohner, Anrainer, Grenznachbar, hier soviel wie 'zugeordnet zu'
Algorithmus	genau definierte Handlungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen in endlich vielen Schritten
Alignment	Als Alignment in der Bioinformatik wird eine Untersuchung z.B. von RNA-Sequenzen auf deren Ähnlichkeit oder Verwandtschaft bezeichnet. Beim Sequenzalignment wird nur die Sequenz selbst betrachtet (also die Reihenfolge der einzelnen Basen), beim Sequenz-Strukturalignment wird zusätzlich die Sekundärstruktur betrachtet und berücksichtigt.
Backtrace	Als Backtrace wird die Rückverfolgung eines Berechnungsvorganges zur Ermittlung des eingeschlagenen Weges bezeichnet. Man sucht sich hierbei den richtigen Weg vom Ergebnis zum Ausgang der Berechnung.
Base	Nukleobase, der Bestandteil, durch den sich die Nukleotide voneinander unterscheiden
DNA	Desoxyribonukleinsäure, in allen Lebewesen und DNA-Viren vorkommendes Biomolekül und Träger der Erbinformationen, meist vorkommend als Doppelhelix
Grammatik	In Bezug auf Parsing eine Sammlung fester Regeln, anhand derer ein Input zerlegt werden kann.
Heuristik	bezeichnet die Kunst, mit begrenztem Wissen und wenig Zeit gute Lösungen für ein Problem zu finden

Nucleolus	Kernkörperchen eukaryotischer Zellen, kugelförmiges Gebilde innerhalb des Zellkerns
Nukleotid	Ein Molekühl, dient als Grundbaustein der RNA und DNA, innerhalb der Arbeit oft gleichverwendet mit Base
Parser, to parse	Zerlegung (zerlegen) eines gegebenen Inputs anhand bestimmter Regeln.
Parsetree	Syntaxbaum, Darstellung der gefunden Zerlegung eines Inputs.
planar	Eine planare Abbildung ist eine Abbildung von Graphen, die auf einer Ebene mit Punkten für Knoten und Linien für Kanten dargestellt werden können, so dass sich diese Kanten nicht überschneiden.
RNA	Ribonukleinsäure, eine Kette aus vielen Nukleotiden
Splicing	Extraktion der genetisch relevanten Daten aus der RNA, welche in Proteine umgewandelt werden
Transkription	Umwandlung eines Gens von DNA in RNA
Translation	die Übersetzung von genetischen Informationen aus einer Sequenz von Nukleotiden in Proteine

A Literaturverzeichnis

[AF99] A. Faller, M. Schünke, G. Schünke: *Der Körper des Menschen*. Georg Thieme Verlag, 1999.

[Aku00] Akutsu, T.: *Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots*. Discrete Applied Mathematics, 104:45–62, 2000.

[AP90] Abrahams, J.P., van der Berg M. van Batenburg E und C. Pleij. *Nucleic Acids Research*, (18):3035–3044, 1990.

[AZC05] Andronescu, Mirela, Zhi Chuan Zhang und Anne Condon: *Secondary structure prediction of interacting RNA molecules*. 345(5):987–1001, 2005.

[Bus08] Busch, Anke: *RNA Secondary Structure Design under Simple and Complex Constraints*. Doktorarbeit, Albert-Ludwigs-University Freiburg, January 2008.

[CDR+04] Condon, Anne, Beth Davy, Baharak Rastegari, Shelly Zhao und Finbarr Tarrant: *Classifying RNA pseudoknotted structures*. Theoretical Computer Science, 320(1):35–50, 2004.

[DDKM04] Deogun, Jitender S., Ruben Donis, Olga Komina und Fangrui Ma: *RNA secondary structure prediction with simple pseudoknots*. In: *APBC '04: Proceedings of the second conference on Asia-Pacific bioinformatics*, Seiten 239–246, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[DP03] Dirks, Robert M. und Niles A. Pierce: *A partition function algorithm for nucleic acid secondary structure including pseudoknots*. J Comput Chem, 24(13):1664–77, 2003.

[Eva06] Evans, Patricia A.: *Finding Common RNA Pseudoknot Structures in Polynomial Time.* In: *Combinatorial Pattern Matching (CPM 2006)*, Band 4009/2006 der Reihe *Lecture Notes in Computer Science*, Seiten 223–232. Springer Berlin / Heidelberg, 2006.

[Got82] Gotoh, O.: *An improved algorithm for matching biological sequences.* 162:705–708, 1982.

[Gru90] Grune, D., Jacobs C.: *Parsing Techniques - A practical guide.* Ellis Horwood Limited, 1990.

[GTN00] Giedroc, D. P., C. A. Theimer und P. L. Nixon: *Structure, Stability and Function of RNA Pseudoknots Involved in Stimulating Ribosomal Frameshifting.* 298(2):167–186, 2000.

[HBB02] Huttenhofer, Alexander, Jurgen Brosius und Jean Pierre Bachellerie: *RNomics: identification and function of small, non-messenger RNAs.* 6(6):835–43, 2002.

[HLG05] Havgaard, Jakob H., Rune B. Lyngso und Jan Gorodkin: *The FOLDALIGN web server for pairwise structural RNA alignment and mutual motif search.* 33(Web Server issue):W650–3, 2005.

[HLSG05] Havgaard, Jakob Hull, Rune B. Lyngso, Gary D. Stormo und Jan Gorodkin: *Pairwise local structural alignment of RNA sequences with sequence similarity less than 40%.* Bioinformatics, 21(9):1815–24, 2005.

[JLMZ02] Jiang, Tao, Guohui Lin, Bin Ma und Kaizhong Zhang: *A General Edit Distance between RNA Structures.* J. Comput. Biol., 9(2):371–88, 2002.

[LP00] Lyngso, Rune B. und Christian N. S. Pedersen: *Pseudoknots in RNA Secondary Structures.* ACM Press, 2000. BRICS Report Series RS-00-1.

[LTM06] Lu, Zhi John, Douglas H. Turner und David H. Mathews: *A set of nearest neighbor parameters for predicting the enthalpy change of RNA secondary structure formation.* 34(17):4912–24, 2006.

[MWB08] Möhl, Mathias, Sebastian Will und Rolf Backofen: *Fixed Parameter Tractable Alignment of RNA Structures Including Arbitrary Pseudoknots.* In: *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008)*, 2008. accepted.

[MWB09] Möhl, Mathias, Sebastian Will und Rolf Backofen: *Lifting Prediction to Alignment of RNA Pseudoknots.* 2009. submitted.

[NW70] Needleman, S. B. und C. D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* 48(3):443–53, 1970.

[PRB85] Pleij, C. W., K. Rietveld und L. Bosch: *A new principle of RNA folding based on pseudoknotting.* Nucleic Acids Research, 13(5):1717–1731, March 1985.

[RE99a] Rivas, E. und S. R. Eddy: *A dynamic programming algorithm for RNA structure prediction including pseudoknots.* 285(5):2053–68, 1999.

[RE99b] Rivas, E. und S. R. Eddy: *A dynamic programming algorithm for RNA structure prediction including pseudoknots.* J Mol Biol, 285(5):2053–2068, Feb 1999.

[RG96] R. Giegerich, D. Wheeler: *Pairwise Sequence Alignment*, 1996.

[RG04] Reeder, Jens und Robert Giegerich: *Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics.* BMC Bioinformatics, 5:104, Aug 2004.

[SB05] Siebert, Sven und Rolf Backofen: *MARNA: multiple alignment and consensus structure prediction of RNAs based on sequence structure comparisons.* Bioinformatics, 21(16):3352–9, 2005.

[SS78] Studnicka, G.M.; Rahn, G.M.; Cummings I.W. und W. Salser: *Computer method for predicting the secondary structure of single-stranded RNA.* Nucleic Acids Research, 5(9):3365–3388, 1978.

[SW81] Smith, T.F. und M.S. Waterman: *Comparison of Biosequences.* Adv. appl. Math., 2:482–489, 1981.

[TDPD92] Ten Dam, Edwin, Kees Pleij und David Draper: *Structural and functional aspects of RNA pseudoknots.* Biochemistry, 31(47):11665–11676, 1992.

[UHKY99] Uemura, Yasuo, Aki Hasegawa, Satoshi Kobayashi und Takashi Yokomori: *Tree adjoining grammars for RNA structure prediction.* Theoretical Computer Science, 210:277 – 303, 1999. Paper as Print Copy.

[WC53] Watson, J. D. und F. H. Crick: *Molecular structure of nucleic acids. A structure for deoxyribose nucleic acid.* Nature, 171:737–741, 1953.

[Woo00] Woodson, S. A.: *Recent insights on RNA folding mechanisms from catalytic RNA.* Cell Mol Life Sci, 57(5):796–808, 2000.

[WRH+07] Will, Sebastian, Kristin Reiche, Ivo L. Hofacker, Peter F. Stadler und Rolf Backofen: *Inferring Non-Coding RNA Families and Classes by Means of Genome-Scale Structure-Based Clustering.* PLoS Comput. Biol., 3(4):e65, 2007.

[ZS81] Zuker, M. und P. Stiegler: *Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information.* 9(1):133–48, 1981.

[ZWW99] Zwieb, C., I. Wower und J. Wower: *Comparative sequence analysis of tmRNA.* Nucleic Acids Res, 27(10):2063–2071, May 1999.

B Abbildungsverzeichnis

C Tabellenverzeichnis

D Testumgebung / Testdaten

D.1 Testumgebung

Für die Tests zur Laufzeit- und Kostenanalyse wurde, wenn nicht explizit etwas anderes erwähnt wurde, ein Computer mit folgender Ausstattung benutzt, der auch mit keinen anderen Prozessen während den Berechnungen belastet wurde:

- Dell Optiplex 330
- Intel Pentium DualCore CPU E2180, 2GHz
- 1,0 GB RAM
- 160 GB HDD
- Microsoft Windows XP, ServicePack 2

D.2 Testdaten

Die hier abgedruckten Testdaten entstammen alle der PseudoBase[1], einer Sammlung von RNA Pseudoknoten, die über ihre Homepage der Wissenschaft zur Verfügung steht. Die in der Tabelle angegebene PKB Nummer ist eine fortlaufende Nummer, die von der PseudoBase verwendet wird. Über die können weitere Informationen über die genaue Struktur eingeholt werden.

[1] http://www.ekevanbatenburg.nl/PKBASE/PKB.HTML

Nummer	Definition	Organismus	Familie
PKB49	PseudoknotPK1 of E.coli tmRNA	E.coli	tmRNA
PKB50	PseudoknotPK2 of E.coli tmRNA	E.coli	
PKB51	PseudoknotPK3 of E.coli tmRNA	E.coli	
PKB52	PseudoknotPK4 of E.coli tmRNA	E.coli	
PKB65	The pseudoknot 505-507/524-526 of 16S ribosomal RNA of E.coli	E.coli	rRNA
PKB205	Pseudoknot E23-9/12 of 18S ribosomal RNA	Palmaria palmata	
PKB147	The pseudoknot E21-7/8 of 18S ribosomal RNA of S.cerevisiae	Saccharomyces cerevisiae	
PKB207	Pseudoknot in the PrP-encoding mRNA	Bos Taurus (cow)	mRNA
PKB71	Pseudoknot of the regulatory region of the alpha ribosomal protein operon	E.coli	
PKB72	Pseudoknot of the regulatory region of S15 ribosomal protein mRNA	E.coli	
PKB206	Pseudoknot in the PrP-encoding mRNA	Homo sapiens	
PKB73	Pseudoknot of the regulatory region of bacteriophage T2 gene 32 mRNA	T2 bacteriophage	
PKB78	Gag/pol translational readthrough site of AKV murine leukemia virus	AKV murine leukemia virus	viral ribosomal readthrough signals
PKB98	Gag/pol translational readthrough site of baboon endogenous virus	baboon endogenous virus	
PKB47	Gag/pol translational readthrough site of Moloney murine leukemia virus	Moloney murine leukemia virus	
PKB48	Gag/pol translational readthrough site of spleen necrosis virus	spleen necrosis virus	
PKB191	tRNA-like structure 3'end pseudoknot of RNA3	alfalfa mosaic virus	viral tRNA like structures
PKB14	tRNA-like structure 3'end pseudoknot of andean potato latent virus	andean potato latent virus	
PKB135	tRNA-like structure 3'end pseudoknot of RNA3 of broad bean mottle virus	broad bean mottle virus	
PKB25	tRNA-like structure 3'end pseudoknot of turnip vein-clearing virus	turnip vein-clearing virus	
PKB5	tRNA-like structure from turnip yellow mosaic virus	turnip yellow mosaic virus	
PKB12	tRNA-like structure 3'end pseudoknot of wild cucumber mosaic virus	wild cucumber mosaic virus	
PKB277	The 5'-leader pseudoknot PK1 of TEV	tobacco etch virus	other viral 5' utr
PKB278	The 5'-leader pseudoknot PK2 of TEV	tobacco etch virus	
PKB279	The 5'-leader pseudoknot PK3 of TEV	tobacco etch virus	
PKB299	5'UTR pseudoknot	Theiler's murine encephalomyelitis virus	

Tabelle D.1: verwendete Testdaten

Nummer	Definition	Organismus	Familie
PKB240	ribosomal frameshift signal ORF-2/3	E.coli	viral ribosomal frameshifting signals
PKB128	ORF1a/ORF1b (polymerase) ribosomal frameshift site of Berne virus	Berne virus	
PKB1	Gag/pro ribosomal frameshift site of Bovine Leukemia Virus	Bovine Leukemia Virus	
PKB2	Orf2-orf3 ribosomal frameshift site of Beet Western Yellows Virus	Beet Western Yellows Virus	
PKB46	ORF2/ORF3 (putative RNA-dependent RNA polymerase) ribosomal frameshift site of barley yellow dwarf virus, NY-RPV isolate	barley yellow dwarf virus	
PKB44	ORF2/ORF3 (putative RNA-dependent RNA polymerase) ribosomal frameshift site of cucurbit aphid-borne yellows virus	cucurbit aphid-borne yellows virus	
PKB258	Ma3 gene ribosomal frameshift signal	Homo sapiens	
PKB3	Gag-pol ribosomal frameshift site of Equine Infectious Anemia Virus	Equine Infectious Anemia Virus	
PKB4	Gag-pol ribosomal frameshift site of Feline Immunodeficiency Virus	Feline Immunodeficiency Virus	
PBK171	ORF1a/ORF1b (polymerase) ribosomal frameshift site	human coronavirus 229E	
PKB257	Edr gene ribosomal frameshift signal	Mus musculus (mouse)	
PKB253	ORF1a/1b ribosomal frameshift site of White Bream Virus	White Bream Virus	
PKB144	tRNA-like structure bulge pseudoknot of odontoglossum ringspot virus, Singapore isolate	odontoglossum ringspot virus	viral tRNA like structures
PKB138	tRNA-like structure 3'end pseudoknot of RNA beta of barley stripe mosaic virus	barley stripe mosaic virus	

Tabelle D.2: Für den Vergleich der heuristischen Parser verwendete Testdaten

Nummer	Sekundärstruktur
TS a	((((.((((.((((..)))).)))).))))
TS b	((((.((((.[[[[..)))).)))).]]]]
TS c	((((.((((.[[[[..)))).]]]].))))
TS d	((((.[[[[.)))).)..((((.]]]].))))
TS e	((((.[[[[.{{{{..)))).]]]].}}}}
TS f	((((.[[[[.{{{{..]]]].)))).}}}}
PKB72	((((.((((((.[[[[[[[)))))))))).....]]]]]]]
PKB72a	((((.((((((.[[[[[[[)))))))))).]]]]]]].....]]]]]]]
PKB72b	((((.((((((.[[[[[[[)))))))))).....]]]]]]].....]]]]]]]
PKB72c	((((.((((((.....[[[[[[[)))))))))).....]]]]]]]
PKB72d	.....((((.((((((.[[[[[[[)))))))))).....]]]]]]]
PKB72e	.....(((((((((((((.((((((.[[[[[[[)))))))))))))))))))).]]]]]]]
PKB72f	((((.((((((....((((()))(()))))(((()))))...[[[[[[[)))))))))).....]]]]]]]

Tabelle D.3: weitere selbsterstellte Testdaten

E Danksagung

Zum Schluss dieser Arbeit möchte ich nun noch die Gelegenheit nutzen, und mich bei einigen Leuten zu bedanken, die zum Gelingen dieser Arbeit beigetragen haben:

Für die Hilfe bei der Suche und der anschließenden Bereitstellung eines sehr interessanten Themas im Bereich der Bioinformatik, für die Betreuung und die abschließende Bewertung meiner Arbeit bedanke ich mich bei Prof. Dr. Rolf Backofen und Dr. Sebastian Will.

Einen besonderen Dank gebührt dabei meinen Betreuern der letzten Monate Mathias Möhl und Dr. Sebastian Will, die jederzeit für Fragen und Diskussionen zur Verfügung standen, sei es persönlich, telefonisch oder auf elektronischem Weg. Vielen Dank für die Unterstützung, auch wenn es wohl manchmal großer Geduld mit mir bedurfte.

Auch den Menschen, die zum Schluss einen Blick auf meine Arbeit warfen und das Korrekturlesen übernahmen, möchte ich meinen Dank aussprechen, auch und gerade aus dem Grund, da nicht immer viel Zeit blieb und es sich für die meisten um keine leichte Kost handelte.

Nicht vergessen möchte ich auch meine Kommilitonen Daniel Schüssele und Joachim Krempel, die mich während des ganzen Studiums begleitet haben. Auch sie hatten immer ein offenes Ohr für mich und halfen mir bei Problemen, selbst dann als sie selbst mit ihrer Diplomarbeit beschäftigt waren. Danke für die gemeinsam verbrachte und erlebte Zeit und die Freundschaft, die entstanden ist.

Zu guter Letzt bleibt noch ein ganz großes „DANKESCHÖN“, das ich an meine Eltern und meine Familie richten möchte. Ohne deren langjährige Unterstützung auf meinem Weg, ihrer Hilfe und Beratung meine selbst gesteckten Ziele zu erreichen, wäre ich nie soweit gekommen, diese Arbeit überhaupt zu schreiben.