technische universität
dortmund

Ein progressiver Algorithmus für Multiple Sequence Alignment auf Basis von A*-Suche

Christine Zarges

Algorithm Engineering Report
TR07-1-009
November 2007
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Diplomarbeit

Ein progressiver Algorithmus für Multiple Sequence Alignment auf Basis von A*-Suche

Christine Zarges

INTERNE BERICHTE
INTERNAL REPORTS

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

14. Mai 2007

Gutachter:

Prof. Dr. Petra Mutzel
Maria Kandyba

GERMANY · D-44221 DORTMUND

Ein progressiver Algorithmus für Multiple Sequence Alignment auf Basis von A*-Suche

Christine Zarges

Ich danke Melanie Schmidt für zahlreiche Anmerkungen und Diskussionen, Daniel Göhring für viele hilfreiche Tipps, Maria Kandyba, Markus Chimani und Frau Prof. Dr. Petra Mutzel für die Betreuung dieser Arbeit sowie meinen Eltern für die Unterstützung während des gesamten Studiums.

Christine Zarges, E-Mail: christine.zarges@uni-dortmund.de

Fachbereich Informatik, Universität Dortmund

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbolverzeichnis

Folgende Liste gibt einen Überblick über die in der vorliegenden Arbeit verwendeten Bezeichnungen und Symbole.

Sequenzen:

Σ	Alphabet
Σ^+	$\Sigma \cup \{-\}$
Σ^*	Menge aller möglichen Strings über Σ
$S_1, \ldots, S_n$	Sequenzen mit $S_i = (s_{i,1} \ldots s_{i,\ell_i}) \in \Sigma^*$, $s_{i,j} \in \Sigma$
$S_i^{-1} = (s_{i,\ell_i} \ldots s_{i,1})$	Die zu S_i inverse Sequenz
α_i^k	Präfix von S_i, der die ersten k Buchstaben umfasst
σ_i^k	Suffix von S_i, der die letzten $\ell_i - k$ Buchstaben umfasst
$S = \{S_1, \ldots, S_n\}$	Menge von Sequenzen
n	Anzahl Sequenzen
$\ell_1, \ldots, \ell_n$	Länge der Sequenzen $S_1, \ldots, S_n$
ℓ_{max}	Länge der längsten Eingabesequenz

Alignments:

A	Ein Alignment mit $A == \begin{pmatrix} s'_{1,1} & s'_{1,2} & \ldots & s'_{1,\ell} \\ s'_{2,1} & s'_{2,2} & \ldots & s'_{2,\ell} \\ \vdots & \vdots & \ddots & \vdots \\ s'_{n,1} & s'_{n,2} & \ldots & s'_{n,\ell} \end{pmatrix}$
A^*	Optimales Alignment
A_{ij}	Projektion von A auf S_i und S_j (induziertes paarweises Alignment)

$S'_i = (s'_{i,1} \dots s'_{i,\ell})$	Zeile eines Alignments A bzgl. S_i
ℓ	Anzahl Spalten eines Alignments A
$D = (d_{ij})$	$(\Sigma \times \Sigma)$-Substitutionsmatrix mit $d_{ij} \in \mathbb{Q}$
$d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$	Distanzfunktion zu einer Substitutionsmatrix D
$w(A)$	Bewertung eines Alignments A
$w(A_{ij})$	Bewertung für ein induziertes paarweises Alignment A_{ij}
$w^*(S_i, S_j)$	Wert eines optimalen Alignments von S_i und S_j
μ	Obere Schranke für den Wert eines optimalen Alignments
$g : \mathbb{N} \to \mathbb{Q}$	Lückenstrafe
γ	Kosten für eine einzelne Lücke bei Verwendung von linearen Lückenstrafen
g_{op}	Lückenöffnungskosten (Gap Opening Penalty)
g_{ext}	Lückenfortsetzungskosten (Gap Extension Penalty)

Graphentheorie:

$G = (V, E, c)$	Edit-Graph mit Quelle $s = (\ell_1, \dots, \ell_n)$, Senke $t = (0, \dots, 0)$ und Kantenkosten c
$G^{-1} = (V^{-1}, E^{-1}, c^{-1})$	Inverser Edit-Graph mit Quelle $t = (0, \dots, 0)$, Senke $s = (\ell_1, \dots, \ell_n)$ und Kantenkosten c^{-1}
(v, w), $v \to w$	Gerichtete Kante von v nach w
$v \xrightarrow{*} w$	Weg von v nach w
$\lambda(v \xrightarrow{*} w)$, $\lambda(p)$	Länge eines Weges von v nach w bzw. eines Weges p

Algorithmus von Dijkstra:

$\text{dist}_s[v]$	Länge eines kürzesten Weges von s nach v in G
$\text{dist}_t[v]$	Länge eines kürzesten Weges von t nach v in G^{-1}
$\delta_s(v)$	Distanzlabel für Knoten v in G
$\delta_t(v)$	Distanzlabel für Knoten v in G^{-1}
$\pi_s(v)$	Vorgänger von v auf einem (kürzesten) Weg in G

$\pi_t(v)$	Vorgänger von v auf einem (kürzesten) Weg in G^{-1}
Q, Q^{-1}	Prioritätswarteschlange

A*-Suche:

$h_s(v)$	Funktion, die die Länge einen kürzesten Weg von s nach v in G abschätzt
$h_t(v)$	Funktion, die die Länge einen kürzesten Weg von t nach v in G^{-1} abschätzt
c_{h_s}	Reduzierte Kantenkosten bzgl. h_s in G
$c_{h_t}^{-1}$	Reduzierte Kantenkosten bzgl. h_t in G^{-1}
P, P^{-1}	Datenstruktur zur Verwaltung endgültig markierter Knoten

k-Band-Algorithmus:

$q_i = (q_{i,1}, \ldots, q_{i,n})$	i-tes Diagonalelement

Kapitel 1

Einleitung

Die Bioinformatik ist ein aufstrebendes interdisziplinäres Gebiet der heutigen Wissenschaft und verfolgt das Ziel, effiziente algorithmische Methoden für die biologische Forschung zu entwickeln und einzusetzen. Beginnend mit der Entdeckung der Doppelhelix-Struktur der DNS durch Watson und Crick im Jahre 1953 hat sich das Wissen im Bereich der Molekularbiologie rasant vermehrt. Die Bioinformatik arbeitet daher meist auf großen Datenbeständen, um neue Erkenntnisse zu gewinnen und diese für andere Wissenschaften nutzbar zu machen. Viele molekularbiologische Erkenntnisse wären heute ohne den Einsatz informatischer Methoden undenkbar.

Die vorliegende Arbeit beschäftigt sich mit einem fundamentalen Teilgebiet der Bioinformatik – der Sequenzanalyse. Hierbei werden molekularbiologische Sequenzen wie z. B. Nukleinsäuren (DNS, RNS) oder Aminosäuresequenzen (Proteine) miteinander verglichen und Ähnlichkeiten zwischen ihnen identifiziert. Grundlage dieser Untersuchungen ist die Annahme, dass eine hohe Ähnlichkeit derartiger Sequenzen funktionelle und strukturelle Ähnlichkeiten sowie eine evolutionäre Verwandtschaft impliziert [19]. Diese Hypothese kann damit begründet werden, dass bewährte Strukturen im Laufe der Evolution wiederverwendet, kopiert und dabei modifiziert werden (Zentrales Paradigma der Evolution von Proteinen [19]).

Die ersten Untersuchungen im Bereich der Sequenzanalyse begannen in den 60er Jahren. Zu diesem Zeitpunkt waren nur wenige molekularbiologische Sequenzen verfügbar, wie z. B. die Proteine Cytochrome C[1] und Hämoglobin[2]. Da diese Proteine für verschiedene Organismen bekannt waren, wurden aufgrund der Annahme, dass verwandte Organismen auch ähnliche Proteinsequenzen besitzen, Abstammungsbäume konstruiert [47].

Ein erster großer wissenschaftlicher Erfolg auf dem Gebiet der Sequenzanalyse gelang 1983 mit der Feststellung eines Zusammenhangs zwischen einem krebsverursachenden Gen und einem Gen, das am normalen Wachstums- und Entwicklungsprozess von Zellen beteiligt ist. Aufgrund der Ähnlichkeit dieser beiden Gene wurde die Vermutung, dass Krebs durch eine falsche Aktivierung dieses Wachstumsgens entsteht, bestätigt [34].

[1] Cytochrome dienen der Übertragung von Elektronen.

[2] Hämoglobin ist zuständig für den Sauerstofftransport im Blut.

Den Durchbruch auf dem Gebiet der Bioinformatik stellt schließlich der erfolgreiche Abschluss des Human-Genom-Projektes im Jahre 2003 dar, bei dem mit Hilfe der Informatik die menschliche DNS vollständig entschlüsselt wurde. Hieraus ergaben sich neue Aufgabenfelder und Herausforderungen. Die gewonnenen Erkenntnisse helfen häufig bei der Diagnose und Vorbeugung von genetisch beeinflussten Krankheiten.

1.1 Problembeschreibung und verwandte Arbeiten

Wir werden uns im Folgenden insbesondere mit dem Vergleich von Proteinen beschäftigen. Diese Ähnlichkeitsanalyse hat zwei Hauptanwendungsgebiete: Einerseits sollen Zusammenhänge zwischen Aminosäuresequenzen und der Funktion bzw. der Struktur der durch sie kodierten Proteine identifiziert werden. Andererseits dient die Analyse der Bestimmung von Verwandtschaftsbeziehungen und damit der Rekonstruktion von Evolutionsprozessen.

Eine wichtige Methode für die Struktur- und Funktionsanalyse von Proteinen sind Alignment-Verfahren. Ziel dieser Verfahren ist die Bestimmung eines Maßes für den Verwandtschaftsgrad der gegebenen Sequenzen. Diese werden dabei durch das Einfügen von Lücken so gegeneinander verschoben, dass ähnliche Bereiche übereinander stehen. Hieraus können wiederum evolutionäre Verwandtschaften (Homologien) abgeleitet werden. Darüber hinaus stellen Alignment-Verfahren die Grundlage für komplexere Aufgaben dar. Zu nennen sind hier z. B. DNS-Sequenzierung, Datenbanksuche und die Erstellung phylogenetischer Bäume.

Aufgrund dieser Anwendungen ist die Entwicklung effizienter Alignment-Verfahren sehr wichtig. Ein weit verbreitetes Framework zur Berechnung von Alignments stellt das Sum-of-Pairs-Maß [7] dar. Während sich das paarweise Alignment, bei dem nur zwei Sequenzen gegeben sind, in quadratischer Zeit lösen lässt, ist das Multiple Sequence Alignment mit n Sequenzen gemäß Sum-Of-Pairs-Maß NP-schwer [46]. In der Praxis beschäftigt man sich jedoch meistens mit dem Multiple Sequence Alignment. Da es dort wichtig ist, möglichst schnell ein Ergebnis zu erhalten, werden in der Regel Heuristiken [19, 32] eingesetzt.

Trotz der Komplexität des vorliegenden Problems ist die Entwicklung effizienter exakter Algorithmen sinnvoll. Sie kommen z. B. bei der Evaluierung von Heuristiken oder als Unterprogramm für Heuristiken zum Einsatz. Eine klassische exakte Methode zur Berechnung von Alignments ist der auf dem Prinzip der Dynamischen Programmierung basierende Needleman-Wunsch-Algorithmus [33]. Er wurde u. a. von Carillo und Lipman [7] verbessert.

Ein anderer Ansatz für das vorliegende Problem basiert auf Kürzeste-Wege-Algorithmen. Gupta et al. präsentierten in [18] einen Branch-and-Bound-Ansatz, der den Algorithmus von Dijkstra [11] verwendet. Horton et al. [24, 29, 38] nutzten stattdessen die A*-Suche [20] für die Berechnung des kürzesten Weges und konnten auf diese Weise den Suchraum des Algorithmus deutlich reduzieren.

Einen Kompromiss aus Heuristiken und exakten Algorithmen stellen progressive Algorithmen[3] dar. Hierbei werden zunächst nur suboptimale Lösungen berechnet, die dann abhängig von den zur Verfügung stehenden Ressourcen (Zeit, Speicher) immer weiter verbessert werden und beweisbar gegen das Optimum konvergieren.

Der erste und einzige uns bekannte progressive Algorithmus für das Multiple Sequence Alignment wurde von Reinert et al. [36] vorgestellt. Er basiert auf der Verknüpfung einer heuristischen Divide-and-Conquer-Methode [41] mit der A*-Suche.

1.2 Ziele der Arbeit

Aufgrund der bisherigen Ergebnisse auf dem Gebiet der Alignment-Verfahren soll ein neuer progressiver Algorithmus auf Basis der A*-Suche entwickelt werden. Hierzu soll anstelle der Divide-and-Conquer-Heuristik eine k-Band-Heuristik [19] verwendet werden, die ursprünglich für das paarweise Alignment entworfen wurde. Die zentrale Idee dieses Algorithmus ist, den Suchraum zunächst auf einen kleinen Bereich (k-Band) einzuschränken, der dann sukzessive erweitert wird. Auf dem eingeschränkten Suchraum wird dann z. B. eine A*-Suche ausgeführt. Die Größe des aktuell betrachteten k-Bandes kann dabei zusätzlich einen Hinweis auf die Ähnlichkeit der betrachteten Sequenzen geben.

Des Weiteren gibt es im Bereich Kürzester-Wege-Algorithmen vielversprechende Resultate bei der Verwendung einer bidirektionalen Suche [35]. Hierbei werden im Gegensatz zur unidirektionalen Suche zwei parallele Berechnungen durchgeführt: eine vom Startknoten s und eine vom Zielknoten t aus. Sobald sich die beiden Suchrichtungen treffen, werden die Wege geeignet konkateniert. In dieser Arbeit soll untersucht werden, ob sich die Verwendung einer bidirektionalen Suche für das vorliegende Problem lohnt. Soweit uns bekannt ist, wurde bisher weder die k-Band-Heuristik noch die bidirektionale Suche für das Multiple Sequence Alignment eingesetzt.

Ziel der vorliegenden Arbeit ist, aus den vorgestellten Ansätzen einen progressiven Algorithmus für das Multiple Sequence Alignment zu entwickeln. Dieser soll außerdem implementiert und experimentell analysiert werden. Abschließend sollen die erzielten Ergebnisse mit dem progressiven Algorithmus von Reinert et al. [36] verglichen werden.

1.3 Gliederung der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert: Zunächst wird in Kapitel 2 eine geeignete Problemmodellierung vorgestellt und es werden notwendige Begriffe eingeführt. Kapitel 3 behandelt Standardmethoden zur Berechnung von Alignments,

[3] Im Kontext von Alignment-Verfahren wird der Begriff „progressiv“ häufig mit einer anderen, im Rahmen dieser Arbeit irrelevanten, Bedeutung verwendet und beschreibt eine Klasse von Heuristiken zur Berechnung von Alignments.

die auf dem Prinzip der Dynamischen Programmierung basieren. Danach werden wir in Kapitel 4 zeigen, wie Kürzeste-Wege-Algorithmen auf das vorliegende Problem angewendet werden können.

Schließlich werden in Kapitel 5 verschiedene progressive Algorithmen behandelt. Das Kapitel beginnt dabei mit der Beschreibung des progressiven Algorithmus von Reinert et al. [36]. Im Anschluss wird der im Rahmen dieser Arbeit entwickelte progressive k-Band-Algorithmus präsentiert.

Den Hauptteil der Arbeit stellt die experimentelle Analyse in Kapitel 6 dar. Hier werden verschiedene Varianten des entwickelten Algorithmus auf praktisch relevanten Daten getestet und mit dem bestehenden Verfahren aus [36] verglichen. Außerdem werden verwendete Datenstrukturen erläutert.

In Kapitel 7 werden die in dieser Arbeit erzielten Ergebnisse zusammengefasst und Ideen für Weiterentwicklungen gegeben.

Kapitel 2

Mathematische Problemmodellierung

Dieses Kapitel gibt eine Einführung in die Grundlagen und Formalismen, die zur Beantwortung der in Kapitel 1 formulierten Fragestellungen notwendig sind. Die Güte einer berechneten Lösung hängt dabei einerseits entscheidend davon ab, wie realistisch die gewählte Problemmodellierung ist. Andererseits kann ein zu komplexes Modell dazu führen, dass eine effiziente Lösung der Problemstellung nicht mehr möglich ist. Im Rahmen dieser Diplomarbeit werden übliche Vereinfachungen gewählt.

Der erste Schritt der Modellierung besteht darin, die Eingabedaten, d. h. Nukleinsäuren und Proteine, in geeigneter Form zu kodieren. Als zweites wird das gewünschte Resultat spezifiziert. Hierzu wird erklärt, wie der Verwandtschaftsbegriff für Sequenzen in der vorliegenden Arbeit aufgefasst wird.

2.1 Kodierung von Nukleinsäuren und Proteinen durch Strings

Eine übliche Repräsentation von Nukleinsäuren und Proteinen ist die Beschreibung der linearen Abfolge ihrer einzelnen Bausteine als String über einem Alphabet Σ. Diese einfache Art der Darstellung ist für die Beantwortung der in dieser Arbeit aufgeworfenen Fragen ausreichend, jedoch muss für weitergehende Untersuchungen, wie z. B. die Analyse der räumlichen Struktur der Nukleinsäure DNS, eine komplexere Repräsentation gewählt werden. Für eine detaillierte Beschreibung der gewählten Darstellung sei auf Anhang A verwiesen.

Die in dieser Arbeit vorgestellten Algorithmen können sowohl für den Vergleich von Nukleinsäuren als auch von Proteinen eingesetzt werden, da sie stringbasiert arbeiten. Die gegebenen Strings unterscheiden sich dabei abhängig von der betrachteten molekularbiologischen Sequenz im zugrunde liegenden Alphabet Σ. In der Regel gilt für DNS und RNS $|\Sigma| = 4$ und für Proteine $|\Sigma| = 20$ (siehe Anhang A).

S_1 :	G	C	C	T	G	A	T	G
S_2 :	G	A	C	T	G	G	A	G
S_3 :	A	C	G	A	G			

(a) Eingabe-Sequenzen

S'_1 :	G	C	C	T	–	G	A	T	G
S'_2 :	G	A	C	T	G	G	A	–	G
S'_3 :	–	A	C	–	G	–	A	–	G

(b) Ein mögliches Alignment

Abbildung 2.1: Multiples Alignment für $n = 3$.

Für die Behandlung von Strings führen wir zunächst einige Notationen ein, die im Verlauf der Arbeit verwendet werden:

Notation 2.1 Sei Σ^* die Menge aller möglichen Strings über Σ und $\Sigma^+ := \Sigma \dot\cup \{-\}$. Dabei stellt „$-$“ mit „$-$“ $\notin \Sigma$ ein zusätzliches ausgewiesenes Symbol (*Lückensymbol*, engl.: *Gap*) dar, mit dem Löschungen bzw. Einfügungen modelliert werden.

Mit $S = (s_1 \dots s_\ell) \in \Sigma^*$ bzw. $S_i = (s_{i,1} \dots s_{i,\ell_i}) \in \Sigma^*$ wird eine einzelne Sequenz bezeichnet. Die zu S_i *inverse Sequenz* $(s_{i,\ell_i} \dots s_{i,1})$ heißt S_i^{-1}. Für $S_i = \alpha_i^k \cdot \sigma_i^k$ ist α_i^k der Präfix von S_i, der die ersten k Zeichen umfasst, und σ_i^k der Suffix, der aus den letzten $\ell_i - k$ Zeichen besteht. Dabei stellt $\cdot$ die Konkatenation der beiden Sequenzen dar.

2.2 Verwandtschaftsbegriff für Sequenzen

Im Folgenden wird die in dieser Arbeit gewählte Modellierung des Verwandtschaftsbegriffs für Sequenzen vorgestellt. Die verwendete Modellierung definiert dabei ein Maß für den Verwandtschaftsgrad der gegebenen Sequenzen. Außerdem gibt sie mögliche Erklärungen, wie die Sequenzen im Laufe der Evolution auseinander hervorgegangen sein können. Dabei werden evolutionäre Vorgänge wie Einfügungen, Löschungen und Substitutionen (Mutationen) betrachtet.

2.2.1 Definition von Sequenzalignments

Eine Standardmethode zum Vergleich verschiedener Sequenzen stellen *Alignment-Verfahren* dar. Die Idee dieser Verfahren besteht darin, gegebene Sequenzen durch das Einfügen von Lücken so gegeneinander zu verschieben, dass an einer Position möglichst *ähnliche* Zeichen untereinander stehen. Man sagt, die Sequenzen werden *aligniert*. Das Resultat wird als *Alignment* bezeichnet. Der Ähnlichkeitsbegriff für verschiedene Zeichen wird in Abschnitt 2.2.2 näher erläutert. Abbildung 2.1 zeigt ein Beispiel für ein Alignment von drei Sequenzen.

Übereinstimmende Zeichen in einer Spalte werden als *Match*, unterschiedliche als *Mismatch* bezeichnet. In der Biologie entspricht ein Mismatch einer Mutation. Falls in einer Spalte eine Lücke mit einem Zeichen aligniert wird, kann dies biologisch als Löschung bzw. Einfügung eines Bausteins interpretiert werden. Auf diese Weise können anhand eines Alignments Aussagen über evolutionäre Verwandtschaften oder funktionelle Zusammenhänge der Sequenzen gemacht werden. Anhand

eines Alignments kann mit Hilfe einer geeigneten Bewertungsfunktion eine Ähnlichkeitsmessung durchgeführt werden. Wir werden derartige Bewertungsfunktionen in Abschnitt 2.2.2 vorstellen und zunächst eine formale Definition für ein Alignment angeben [41]:

Definition 2.2 (Alignment) Sei $S = \{S_1, \ldots, S_n\}$ eine Menge von $n \geq 2$ Sequenzen über Σ und ℓ_i die Länge der Sequenz $S_i \in \Sigma^*$, $1 \leq i \leq n$. Bezeichne $s_{i,j}$ den j-ten Buchstaben der i-ten Sequenz.

Ein *Alignment* von S ist gegeben durch eine $(n \times \ell)$-Matrix

$$A = \begin{pmatrix} s'_{1,1} & s'_{1,2} & \ldots & s'_{1,\ell} \\ s'_{2,1} & s'_{2,2} & \ldots & s'_{2,\ell} \\ \vdots & \vdots & \ddots & \vdots \\ s'_{n,1} & s'_{n,2} & \ldots & s'_{n,\ell} \end{pmatrix}$$

mit $s'_{i,j} \in \Sigma^+$. Dabei gilt:

- Die Anzahl der Spalten von A ist mindestens so groß wie die Länge der längsten Sequenz und höchstens so groß wie die Summe der Längen aller gegebenen Sequenzen, d. h. es muss gelten:

$$\max\{\ell_1, \ldots, \ell_n\} \leq \ell \leq \sum_{i=1}^{n} \ell_i$$

- Für jede Zeile $S'_i = (s'_{i,1} s'_{i,2} \ldots s'_{i,\ell})$ erhält man durch Streichung der Lückensymbole die Sequenz S_i.
- Keine Spalte von A enthält ausschließlich das Zeichen „−“.

Für $n = 2$ spricht man von einem *paarweisen Alignment*, für $n > 2$ heißt A *multiples Alignment*. Für letzteres wird außerdem noch der Begriff der Projektion benötigt, den wir wie folgt definieren [41]:

Definition 2.3 (Projektion) Sei $S = \{S_1, \ldots, S_n\}$ eine Menge von Sequenzen und A ein zugehöriges multiples Alignment. Weiter sei $\hat{S} \subseteq S$ eine Teilmenge der gegebenen Sequenzen und S'_i die zu S_i gehörende Zeile in A. Die *Projektion* von A auf $\hat{S}$ ist dann das Alignment $\hat{A}$, für das gilt:

- $\hat{A}$ enthält nur Zeilen S'_i aus A mit $S_i \in \hat{S}$.
- Sämtliche Spalten, die durch die Modifikation nur das Zeichen „−“ enthalten, werden aus $\hat{A}$ entfernt.

Von besonderer Bedeutung für die weiteren Betrachtungen sind paarweise und dreifache Projektionen, d. h. $|\hat{S}| = 2$ und $|\hat{S}| = 3$. Häufig ist in der Literatur anstelle von Projektionen auch von *induzierten Alignments* die Rede.

2.2.2 Bewertung von Sequenzalignments

Die in der vorliegenden Arbeit betrachtete Fragestellung lässt sich wie folgt als Optimierungsproblem formulieren [41]:

Definition 2.4 (Multiple Sequence Alignment) Sei $S = \{S_1, \ldots, S_n\}$ eine Menge von Sequenzen und $w : \mathbb{A}(S) \to \mathbb{Q}$ eine beliebige Bewertungsfunktion auf der Menge der möglichen Alignments zu S. Gesucht ist ein Alignment $A \in \mathbb{A}(S)$, das $w(A)$ optimiert.

Je nach Art der Anwendung kann die Berechnung eines Alignments als Minimierungs- oder Maximierungsproblem formuliert werden. Bei der Maximierungsvariante spricht man von *Ähnlichkeit*. Dabei werden Matches honoriert und Mismatches sowie Lücken bestraft. Alternativ beschreibt die Minimierungsvariante die sog. *Edit-Distanz* zwischen den gegebenen Sequenzen. Unter der Edit-Distanz versteht man die (minimale) Anzahl an Einfüge-, Lösch- und Ersetzungsoperationen, um einen String S_i in einen String S_j zu überführen.

Um die Qualität eines Alignments, d. h. seine biologische Relevanz, bewerten zu können, muss ein geeignetes Bewertungsmodell aufgestellt werden. Die Wahl dieses Modells ist dabei entscheidend für den praktischen Nutzen des berechneten Alignments. Wir werden zunächst Bewertungsfunktionen für paarweise Alignments vorstellen und diese anschließend auf multiple Alignments erweitern.

2.2.2.1 Paarweises Alignment

Es wird häufig vereinfacht angenommen, dass Mutationen, Löschungen und Einfügungen an verschiedenen Stellen der Sequenzen unabhängig voneinander geschehen. Daher basieren übliche Bewertungsmodelle auf dem Vergleich von jeweils zwei einzelnen Zeichen und sind häufig durch eine *Ähnlichkeits-* oder *Distanzfunktion* $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$ gegeben. Das folgende Beispiel gibt zwei einfache Funktionen für die beiden Optimierungsvarianten an:

Beispiel 2.5 (Einfache Ähnlichkeits- und Distanzfunktionen) Sei $s, t \in \Sigma^+$ und $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$. Dann stellt (2.1) eine Ähnlichkeitsfunktion und (2.2) eine Distanzfunktion dar. (2.2) wird auch Levenshtein-Distanz [30] genannt.

$$d(s,t) := \begin{cases} 1 & \text{falls } s = t \\ -1 & \text{falls } s \neq t \\ -2 & \text{falls } s = - \text{ oder } t = - \end{cases} \tag{2.1}$$

$$d(s,t) := \begin{cases} 0 & \text{falls } s = t \\ 1 & \text{falls } s \neq t \\ 1 & \text{falls } s = - \text{ oder } t = - \end{cases} \tag{2.2}$$

Die Bewertungen für Substitutionen werden meistens durch eine (symmetrische) $(\Sigma \times \Sigma)$-*Substitutionsmatrix* $D = (d_{ij})$ mit $d_{ij} \in \mathbb{Q}$ repräsentiert. Abhängig von der Art der Problemformulierung wird dabei zwischen *Ähnlichkeits-* und *Distanzmatrizen* unterschieden. Die beiden Varianten lassen sich einfach ineinander umformen [40, 51]. Außerdem wurde von Smith et al. gezeigt, dass beide Varianten unter bestimmten Umständen äquivalent sind [39]. Wir werden uns daher im Folgenden auf das Minimierungsproblem und damit auf Distanzmatrizen beschränken.

Für den Vergleich von DNS-Sequenzen reichen meist einfache Substitutionsmatrizen (vgl. (2.1) und (2.2)) aus. Da miteinander verwandte Proteine aber häufig nur wenige identische Aminosäuren besitzen, sind derartige Matrizen nicht in der Lage, existierende Verwandtschaften zwischen diesen zu erkennen. Daher werden für Proteinvergleiche meist spezielle Substitutionsmatrizen verwendet. Diese berücksichtigen Ähnlichkeiten bzgl. der chemischen Eigenschaften sowie Mutationswahrscheinlichkeiten unterschiedlicher Aminosäuren. Die beiden gängigsten Substitutionsmatrizen für Proteine sind PAM- [10] und BLOSUM-Matrizen [21]. Wir werden für unsere experimentelle Analyse in Kapitel 6 eine Distanzvariante der PAM-250-Matrix verwenden (siehe Abbildung B.1 in Anhang 6). Für Details zu den genannten Substitutionsmatrizen sei auf Anhang B verwiesen.

Es kann nun wie folgt eine Bewertungsfunktion $w : \mathbb{A}(S) \to \mathbb{R}_0$ auf der Menge der möglichen Alignments definiert werden [41]:

Definition 2.6 (Bewertungsfunktion für paarweise Alignments) Sei A ein Alignment der Sequenzen S_1 und S_2 und ℓ die Anzahl der Spalten von A. Die Bewertung von A entspricht dann der Summe aller Spaltenbewertungen von A gemäß $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$:

$$w(A) := \sum_{k=1}^{\ell} d(s'_{1,k}, s'_{2,k}) \tag{2.3}$$

Diese Bewertungsfunktion weist jedoch eine Schwachstelle auf: Biologisch gesehen ist das Auftreten einer großen zusammenhängenden Lücke der Länge k wahrscheinlicher als viele kleine Lücken, die zusammen ebenfalls die Länge k haben. Dies liegt daran, dass meistens ganze Sequenzteile gelöscht bzw. eingefügt werden. (2.3) bewertet diese beiden Fälle jedoch identisch.

Abhängig von den zur Verfügung stehenden Ressourcen und der Art der Anwendung können verschiedenen Arten von *Lückenstrafen* (engl.: *Gap Costs*) definiert werden [22]. Lückenstrafen werden dabei durch eine Funktion $g : \mathbb{N}_0 \to \mathbb{Q}$ definiert, wobei $g(k)$ die Kosten für eine fortlaufende Lücke der Länge k angibt. Es gelte $g(0) = 0$. Da kürzere Lücken wahrscheinlicher sind als lange, sollte g monoton in der Länge der Lücke sein.

Wir unterscheiden im Rahmen dieser Arbeit folgende Funktionstypen von g:

Lineare Lückenstrafen: Lineare Lückenstrafen werden durch eine lineare Funktion $g(k) = k \cdot \gamma$ beschrieben. Dabei sind γ die Kosten für eine einzelne Lücke. Diese Art der Lückenstrafe wird in (2.3) verwendet.

Affine Lückenstrafen: Affine Lückenstrafen werden durch eine affine Funktion definiert:

$$\begin{aligned} g &: \mathbb{N}_0 \to \mathbb{Q} \\ k &\mapsto g_{\text{ext}} \cdot k + g_{\text{op}} \end{aligned}$$

Der Parameter g_{op} gibt die zusätzlichen Kosten zum Öffnen einer neuen Lücke (*Lückenöffnungskosten*, engl.: *Gap Opening Penalty*) an und stellt sicher, dass eine fortlaufende Lücke der Länge $(k_1 + k_2)$ günstiger ist als zwei Lücken der Längen k_1 und k_2. Der Wert g_{ext} beschreibt die proportionalen Kosten für die Länge der Lücke (*Lückenfortsetzungskosten*, engl.: *Gap Extension Penalty*).

Allgemeine Lückenstrafen: Bei allgemeinen Lückenstrafen kann eine beliebige Funktion g definiert werden. Hier kann z. B. berücksichtigt werden, dass das Löschen oder Einfügen bestimmter Bausteine wahrscheinlicher ist. Es wird nur gefordert, dass g *sublinear* ist, d. h. dass $g(k_1 + k_2) \leq g(k_1) + g(k_2)$ für alle $k_1, k_2 \in \mathbb{N}_0$ gilt.

Optimal für den praktischen Nutzen des berechneten Alignments ist sicherlich die Verwendung von allgemeinen Lückenstrafen. Diese erfordern jedoch einen sehr hohen Rechenzeit- und Speicherplatzbedarf [22]. Lineare Lückenstrafen sind zwar am schnellsten zu berechnen, liefern jedoch Alignments von geringer biologischer Relevanz. Daher werden in der Praxis meistens Varianten von affinen Lückenstrafen verwendet.

Um affine und allgemeine Lückenstrafen bei der Bewertung eines Alignments berücksichtigen zu können, muss die Bewertungsfunktion (2.3) erweitert werden. Dabei werden Spalten, in denen Lücken vorkommen, separat betrachtet.

Definition 2.7 (Erweiterte Bewertungsfunktion für paarweise Alignments) Sei A ein Alignment der Sequenzen S_1 und S_2 und ℓ die Anzahl der Spalten von A. Die Bewertungsfunktion $w : \mathbb{A}(S) \to \mathbb{Q}$ ergibt sich dann für allgemeine Lückenstrafen durch [41]:

$$w(A) := \sum_{\substack{1 \leq i \leq \ell, \\ s'_{1,i} \neq -,\, s'_{2,i} \neq -}} d(s'_{1,i}, s'_{2,i}) \; + \; \sum_{k>0} g(k) \cdot \#\text{ Lücken der Länge } k \text{ in } A \qquad (2.4)$$

Wir werden die vorgestellten Lückenstrafen in Kapitel 3 näher betrachten und Algorithmen für die verschiedenen Funktionen vorstellen.

2.2.2.2 Multiple Sequence Alignment

Für multiple Alignments gibt es verschiedene Bewertungsmöglichkeiten. Im Rahmen dieser Arbeit wird das weit verbreitete *Sum-Of-Pairs-Maß* (SP-Maß, [7]) verwendet, da dieses auch im betrachteten Referenzalgorithmus OMA [36] zum Einsatz kommt. Für weitere Maße siehe z. B. [19, 22].

Das Sum-Of-Pairs-Maß basiert auf den Bewertungen der paarweisen Projektionen eines gegebenen Alignments und lässt sich wie folgt definieren:

Definition 2.8 (Sum-Of-Pairs-Maß) Sei A ein Alignment und $A_{i,j}$ die paarweise Projektion von A auf S_i und S_j. Der Zielfunktionswert $w(A)$ ergibt sich dann gemäß des Sum-Of-Pairs-Maßes durch:

$$w(A) := \sum_{i=1}^{n} \sum_{j=i+1}^{n} w(A_{i,j})$$

Es gibt insgesamt $\binom{n}{2}$ verschiedene paarweise Projektionen eines multiplen Alignments. Daher lässt sich das Sum-Of-Pairs-Maß zu einem gegebenen Alignment in Zeit $O(n^2)$ berechnen.

Bemerkung 2.9 Die Kosten $w(A_{i,j})$ für die paarweise Projektion $A_{i,j}$ eines – gemäß Sum-Of-Pairs-Maß – optimalen multiplen Alignments A^* auf S_i und S_j entsprechen im Allgemeinen nicht den optimalen Kosten $w^*(S_i, S_j)$, um S_i und S_j zu alignieren. Es gilt:

$$w^*(S_i, S_j) \leq w(A_{i,j})$$

2.3 Zusammenfassung

Wir haben in diesem Kapitel eine Modellierung für die vorliegende Fragestellung vorgestellt. Hierzu wurde zunächst eine übliche Kodierung für molekularbiologische Sequenzen angegeben. Anschließend haben wir den Begriff eines Alignments eingeführt und verschiedene Möglichkeiten zur Bewertung von Alignments präsentiert, sowie deren Vor- und Nachteile aufgezeigt. Auf Basis der vorgestellten Problemmodellierung können wir im Folgenden verschiedene Algorithmen für die Lösung der betrachteten Fragestellung beschreiben.

Kapitel 3

Alignments und Dynamische Programmierung

Das folgende Kapitel beschäftigt sich mit Standardverfahren zur Berechnung optimaler Alignments. Diese Verfahren basieren auf dem Prinzip der Dynamischen Programmierung und gehören zu den ältesten Methoden zur Berechnung optimaler Alignments.

Dynamische Programmierung beruht auf der Idee, die optimale Lösung des Gesamtproblems aus den optimalen Lösungen geeigneter Teilprobleme zusammenzusetzen (*Bellmansches Optimalitätsprinzip*, [9]). Ein derartiger Algorithmus kann in zwei Phasen gegliedert werden:

1. Berechnung des Wertes einer optimalen Lösung.
   (a) Charakterisierung der Struktur einer optimalen Lösung.
   (b) Konstruktion einer rekursiven Vorschrift für den optimalen Wert einer Lösung.
   (c) Bottom-up-Berechnung des optimalen Wertes, d. h. zunächst werden die Werte für die kleinsten Teilprobleme direkt berechnet und daraus dann die Lösungen für die jeweils nächstgrößeren Teilprobleme bestimmt. Um bereits berechnete Zwischenlösungen wieder verwenden zu können, werden diese in einer Tabelle gespeichert. Wir nennen diese Tabelle im Folgenden *Rekursionstabelle*, da sie auf einer rekursiven Vorschrift basiert.
2. Rekonstruktion einer zugehörigen optimalen Lösung.

Dynamische Programmierung bietet insbesondere dann Vorteile, wenn die Teilprobleme abhängig voneinander sind, d. h. wenn unterschiedliche Teilprobleme gleiche Unterprobleme haben. Bei der Berechnung optimaler Alignments wird die Alignierung beliebiger Präfixe der gegebenen Sequenzen als Teilproblem aufgefasst. Diese Definition erfüllt die genannte Abhängigkeitsbedingung.

Im Folgenden werden zunächst verschiedene Algorithmen für paarweise Alignments vorgestellt. Dabei wird zwischen linearen, affinen und allgemeinen Lückenstrafen (siehe Abschnitt 2.2.2.1) unterschieden. Zum Schluss des Kapitels werden Erweiterungsmöglichkeiten für multiple Alignments aufgezeigt.

3.1 Paarweises Alignment

Beim paarweisen Alignment sind zwei Sequenzen $S_1 = (s_{1,1} \ \ldots \ s_{1,\ell_1})$ und $S_2 = (s_{2,1} \ \ldots \ s_{2,\ell_2})$ gegeben. Historisch gesehen gibt es verschiedene unabhängige Quellen, die Dynamische Programmierung zur Berechnung optimaler paarweiser Alignments verwenden. Nach Stoye [41] sind hier neben Needleman und Wunsch [33] noch Wagner und Fischer [45] zu nennen. Auch wenn sich die Algorithmen geringfügig unterscheiden, basieren sie auf derselben Grundidee. Kruskal gibt in [26] einen Überblick über den Entwicklungsprozess.

Wir beginnen mit der Beschreibung des einfachsten Algorithmus – dem Needleman-Wunsch-Algorithmus [33]. Dieser berücksichtigt nur lineare Lückenstrafen. Daher werden danach Erweiterungen für allgemeine (Waterman-Smith-Beyer, [48]) und affine Lückenstrafen vorgestellt (Gotoh [15]).

3.1.1 Lineare Lückenstrafen: Needleman-Wunsch

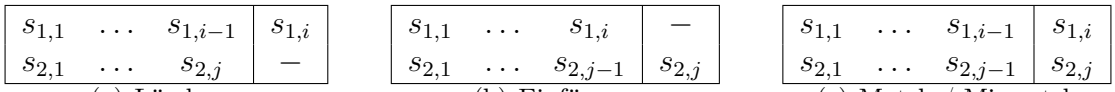
Der Needleman-Wunsch-Algorithmus wurde bereits 1970 für den Vergleich von zwei Proteinsequenzen vorgestellt [33]. Er arbeitet gemäß des Prinzips der Dynamischen Programmierung in zwei Phasen:

1. Berechnung des Wertes eines optimalen Alignments.
2. Rekonstruktion eines zugehörigen optimalen Alignments.

Phase 1: Berechnung des Wertes eines optimalen Alignments

Die Idee des Needleman-Wunsch-Algorithmus besteht darin, optimale Alignments von beliebigen Präfixen zu bestimmen und aus diesen das optimale Alignment der gesamten Sequenzen zu berechnen. Um Lücken am Anfang der Sequenzen zu ermöglichen, wird eine Rekursionstabelle OPT der Größe $(\ell_1+1) \times (\ell_2+1)$ verwendet. Ein Tabelleneintrag $\mathrm{OPT}(i,j)$ wird als Wert eines optimalen Alignments der Präfixe α_1^i und α_2^j interpretiert. Für $i = 0$ ist α_1^i leer, so dass α_2^j nur mit Lücken aligniert wird. Entsprechendes gilt für $j = 0$. Der Eintrag $\mathrm{OPT}(\ell_1, \ell_2)$ entspricht der Bewertung eines optimalen Alignments der gegebenen Sequenzen.

Die Rekursionstabelle kann zeilenweise ausgefüllt werden. Dabei wird bei der Berechnung von $\mathrm{OPT}(i,j)$ angenommen, dass der Wert der optimalen Lösung für alle kürzeren Präfixe bekannt ist. Es kann dann entschieden werden, wie die nächsten beiden Zeichen aligniert werden sollen. Seien $s_{1,i}$ und $s_{2,j}$ die nächsten beiden Zeichen



(a) Löschung (b) Einfügung (c) Match / Mismatch

Abbildung 3.1: Entstehung der Rekursionsgleichung nach Needleman-Wunsch.

und $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$ eine Distanzfunktion. Dann gibt es für die Alignierung drei verschiedene Möglichkeiten (siehe Abbildung 3.1):

1. **Löschung:** $s_{1,i}$ wird mit einer Lücke aligniert.

   Dies entspricht einem Schritt nach unten in der Tabelle. Die Bewertung setzt sich aus dem Tabelleneintrag $\mathrm{OPT}(i-1,j)$ und $d(s_{1,i},-)$ zusammen.

2. **Einfügung:** $s_{2,j}$ wird mit einer Lücke aligniert.

   Dies entspricht einem Schritt nach rechts. Die Kosten berechnen sich symmetrisch zum ersten Fall.

3. **Match / Mismatch:** $s_{1,i}$ und $s_{2,j}$ werden aligniert.

   Dies entspricht in der Tabelle einem Diagonalschritt nach rechts unten. Der Wert dieser Alternative ergibt sich als Summe von $\mathrm{OPT}(i-1,j-1)$ und $d(s_{1,i},s_{2,j})$.

Der Wert eines optimalen Alignments der Präfixe α_1^i und α_2^j ergibt sich dann als Minimum über diese drei Möglichkeiten. Die erste Zeile bzw. Spalte der Tabelle wird mit der Bewertung eines Alignments des entsprechenden Präfixes nur mit Lücken initialisiert. Abbildung 3.2 visualisiert den Ablauf der ersten Phase des Needleman-Wunsch-Algorithmus. Formal lässt sich diese durch folgende rekursive Berechnungsvorschrift darstellen:

$$\mathrm{OPT}(i,0) = i \cdot d(s_{1,i},-) \qquad \forall i \in \{1,\ell_1\} \tag{3.1}$$

$$\mathrm{OPT}(0,j) = j \cdot d(-,s_{2,j}) \qquad \forall j \in \{1,\ell_2\} \tag{3.2}$$

$$\mathrm{OPT}(i,j) = \min \begin{cases} \mathrm{OPT}(i-1,j) + d(s_{1,i},-) \\ \mathrm{OPT}(i,j-1) + d(-,s_{2,j}) \\ \mathrm{OPT}(i-1,j-1) + d(s_{1,i},s_{2,j}) \end{cases} \tag{3.3}$$

Phase 2: Rekonstruktion eines zugehörigen optimalen Alignments

Falls das optimale Alignment selbst ausgegeben werden soll, ist es sinnvoll, während der Berechnung der Rekursionstabelle Zeiger auf die jeweils relevanten Vorgängerzellen zu speichern. Dann erhält man ein optimales Alignment, wenn man nach der Durchführung von Phase 1 von $\mathrm{OPT}(\ell_1,\ell_2)$ aus entlang der Zeiger zu $\mathrm{OPT}(0,0)$ läuft. Eine Rekonstruktion des optimalen Alignments ist mit zusätzlichem Zeitaufwand aber auch ohne diese Zeiger möglich [4].

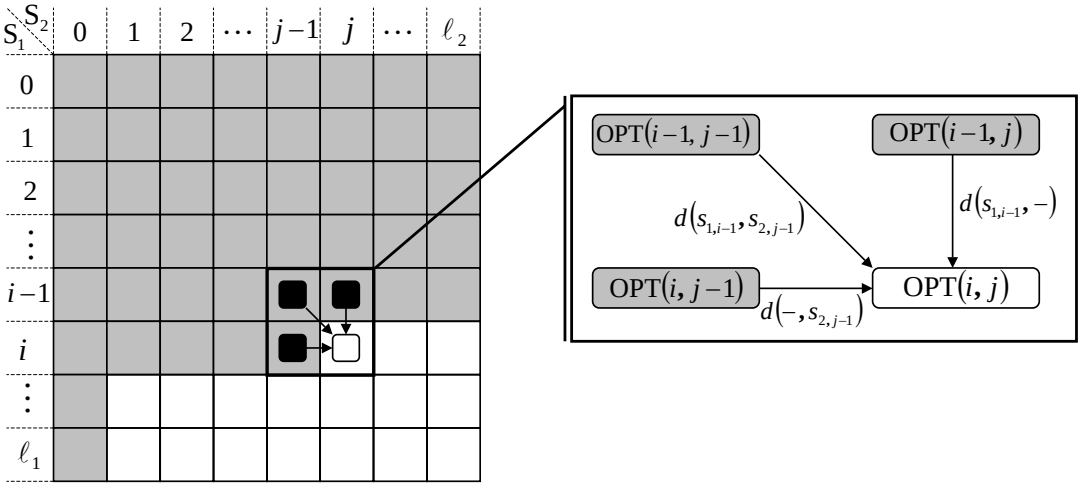


Abbildung 3.2: Berechnung eines optimalen Alignments nach Needleman-Wunsch.

Satz 3.1 (Laufzeit und Speicherbedarf) Der Needleman-Wunsch-Algorithmus berechnet für zwei Sequenzen unter Verwendung von linearen Lückenstrafen in Zeit $O(\ell_1\ell_2)$ und Platz $O(\ell_1\ell_2)$ ein optimales paarweises Alignment.

Beweis: Die Rekursionstabelle OPT hat die Größe $(\ell_1 + 1) \cdot (\ell_2 + 1) = O(\ell_1\ell_2)$. Die Laufzeit setzt sich aus der Berechnung der Tabelleneinträge und der Rekonstruktion des optimalen Alignments zusammen.

Die Berechnung eines jeden Tabelleneintrags benötigt konstante Zeit. Die Rekonstruktion des optimalen Alignments ist in Zeit $O(\ell_1 + \ell_2)$ und mit einem Speicheraufwand von $O(\ell_1\ell_2)$ möglich. Damit folgt direkt die Behauptung. □

Optimale Alignments sind im Allgemeinen nicht eindeutig. Der Algorithmus kann so erweitert werden, dass er sämtliche optimale Alignments ausgibt. Die Anzahl optimaler Alignments kann jedoch exponentiell in der Sequenzlänge sein. So gibt es z. B. insgesamt $\binom{2z}{z}$ viele optimale Alignments der Sequenzen $S_1 = X^z$ und $S_2 = X^{2z}$, da es $\binom{2z}{z}$ verschiedene Möglichkeiten gibt, die z notwendigen Lücken einzufügen, und die Bewertung nicht von dieser Positionierung abhängt. Ein solcher Algorithmus hat demnach im schlechtesten Fall eine Laufzeit, die exponentiell in der Größe der Eingabe ist.

Bemerkung 3.2 Es gibt diverse Varianten des Needleman-Wunsch-Algorithmus, die der Senkung des Speicherbedarfs dienen: Falls nur der Wert des optimalen Alignments berechnet werden soll, genügt es, jeweils die direkte Vorgängerzeile der Rekursionstabelle im Speicher zu halten, da nur diese für die Berechnung der darauf folgenden Zeile benötigt wird. Der Algorithmus benötigt dann einen Platz von $O(\max\{\ell_1, \ell_2\})$.

Wird auch das Alignment selbst benötigt, kommen fortgeschrittenere Algorithmen zum Einsatz. Ein solches Verfahren ist z. B. der Hirschberg-Algorithmus [23].

Er benötigt nur linearen Speicherplatz, dies jedoch bei einem doppelt so hohen Berechnungsaufwand.

3.1.2 Allgemeine Lückenstrafen: Waterman-Smith-Beyer

Mit der einfachen Rekursionsgleichung des Needleman-Wunsch-Algorithmus können keine allgemeinen Lückenstrafen berücksichtigt werden, da es dabei nicht ausreicht, nur die direkten Vorgängerzellen zu betrachten. Vielmehr muss bei der Berechnung von $\mathrm{OPT}(i,j)$ auf alle vorhergehenden Werte der i-ten Zeile und j-ten Spalte zurückgegriffen werden, da die Kosten der Lücken nicht mehr linear sind und von sämtlichen vorhergehenden Zeichen abhängen können.

Waterman et. al [48] haben die folgende Rekursionsgleichung vorgestellt. Dabei beschreibt die Funktion $g(k)$ die allgemeine Lückenstrafe für eine Lücke der Länge k:

$$\begin{aligned}
\mathrm{OPT}(0,0) &= 0 \\
\mathrm{OPT}(i,0) &= g(i) \qquad \forall i \in \{1, \ell_1\} \\
\mathrm{OPT}(0,j) &= g(j) \qquad \forall j \in \{1, \ell_2\} \\
\mathrm{OPT}(i,j) &= \min \begin{cases} \min\limits_{1 \le k \le i} \{\mathrm{OPT}(i-k,j) + g(k)\} \\ \min\limits_{1 \le k \le j} \{\mathrm{OPT}(i,j-k) + g(k)\} \\ \mathrm{OPT}(i-1,j-1) + d(s_{1,i}, s_{2,j}) \end{cases}
\end{aligned}$$

Bei dieser Berechnungsvorschrift werden auch zwei direkt aufeinander folgende Lücken berücksichtigt, d. h. eine Lücke der Länge $(k_1 + k_2)$ wird als zwei Lücken der Längen k_1 und k_2 aufgefasst. Da wir jedoch angenommen haben, dass g sublinear ist, gilt $g(k_1 + k_2) \le g(k_1) + g(k_2)$ für alle $k_1, k_2 \in \mathbb{N}_0$. Damit werden zwei aufeinander folgende Lücken der Längen k_1 und k_2 schlechter bewertet als eine Lücke der Länge $(k_1 + k_2)$ und es wird auf jeden Fall der Wert für die längste fortlaufende Lücke in die Berechnung des $\mathrm{OPT}(i,j)$-Wertes einbezogen.

Dieser Ansatz hat eine Laufzeit von $O(\ell_1 \cdot \ell_2 \cdot (\ell_1 + \ell_2))$, da die Berechnung eines einzelnen Tabelleneintrags nun Zeit $O(\ell_1 + \ell_2)$ benötigt. Außerdem lassen sich die in Bemerkung 3.2 erwähnten Verbesserungen nicht auf allgemeine Lückenstrafen übertragen. Aus diesen Gründen werden allgemeine Lückenstrafen in der Praxis meist nicht eingesetzt.

3.1.3 Affine Lückenstrafen: Gotoh

Affine Lückenstrafen stellen einen Kompromiss aus linearen und allgemeinen Lückenstrafen dar. Sie modellieren biologische Eigenschaften und Zusammenhänge besser als lineare Funktionen, können aber effizienter berechnet werden als allgemeine Lückenstrafen [22]. Affine Lückenstrafen unterscheiden dabei zwischen Lückenöffnungskosten und Lückenfortsetzungskosten (siehe Abschnitt 2.2.2.1).

Der Algorithmus von Gotoh [15] stellt eine Erweiterung des Needleman-Wunsch-Algorithmus für affine Lückenstrafen dar. Die zentrale Idee dieses Verfahrens ist, zusätzlich zum Wert eines optimalen Alignment für zwei gegebene Präfixe, den Wert eines besten Alignments dieser Präfixe abhängig vom Aussehen der letzten Spalte zu speichern. Neben der bisherigen Rekursionstabelle OPT existieren also drei Hilfstabellen A, B und C [22]:

- $A(i,j)$: Distanz eines optimalen Alignments der Präfixe σ_1^i und σ_2^j, das mit einem Match oder Mismatch endet, d. h. $s_{1,i}$ ist mit $s_{2,j}$ aligniert.
- $B(i,j)$: Distanz eines optimalen Alignments der Präfixe σ_1^i und σ_2^j, das mit einer Löschung endet, d. h. $s_{1,i}$ ist mit einer Lücke aligniert.
- $C(i,j)$: Distanz eines optimalen Alignments der Präfixe σ_1^i und σ_2^j, das mit einer Einfügung endet, d. h. $s_{2,j}$ ist mit einer Lücke aligniert.

Der Eintrag $\mathrm{OPT}(\ell_1, \ell_2)$ entspricht wie beim Needleman-Wunsch-Algorithmus der Bewertung eines optimalen Alignments der gegebenen Sequenzen. Es ergeben sich folgende Rekursionsgleichungen, wobei g_{op} die Lückenöffnungskosten und g_{ext} die Lückenfortsetzungskosten bezeichnet:

$$
\begin{array}{llll}
A(0,0) = 0 & B(0,0) = \infty & C(0,0) = \infty & \mathrm{OPT}(0,0) = 0 \\
A(i,0) = \infty & B(i,0) = i \cdot g_{\mathrm{ext}} + g_{\mathrm{op}} & C(i,0) = \infty & \mathrm{OPT}(i,j) = B(i,0) \\
A(0,j) = \infty & B(0,j) = \infty & C(0,j) = j \cdot g_{\mathrm{ext}} + g_{\mathrm{op}} & \mathrm{OPT}(i,j) = C(0,j)
\end{array}
$$

$$
\begin{aligned}
A(i,j) &= \mathrm{OPT}(i-1,j-1) + d(s_{1,i}, s_{2,j}) \\
B(i,j) &= \min \begin{cases} B(i-1,j) + g_{\mathrm{ext}} \\ A(i-1,j) + g_{\mathrm{op}} + g_{\mathrm{ext}} \\ C(i-1,j) + g_{\mathrm{op}} + g_{\mathrm{ext}} \end{cases} \\
C(i,j) &= \min \begin{cases} B(i,j-1) + g_{\mathrm{op}} + g_{\mathrm{ext}} \\ A(i,j-1) + g_{\mathrm{op}} + g_{\mathrm{ext}} \\ C(i,j-1) + g_{\mathrm{ext}} \end{cases} \\
\mathrm{OPT}(i,j) &= \min\{A(i,j), B(i,j), C(i,j)\}
\end{aligned}
$$

Laufzeit und Speicherbedarf des Algorithmus von Gotoh ergeben sich analog zum Needleman-Wunsch-Algorithmus wie folgt:

Satz 3.3 (Laufzeit und Speicherbedarf) Der Algorithmus von Gotoh berechnet für zwei Sequenzen in Zeit $O(\ell_1 \ell_2)$ und Platz $O(\ell_1 \ell_2)$ den Wert eines optimalen paarweisen Alignments bzgl. affiner Lückenstrafen.

Beweis: Es werden vier Rekursionstabellen der Größe $(\ell_1 + 1) \cdot (\ell_2 + 1) = O(\ell_1 \ell_2)$ verwendet. Die Berechnung eines jeden Tabelleneintrags benötigt konstante Laufzeit. Damit folgt direkt die Behauptung. □

Die Rekonstruktion des zugehörigen optimalen Alignments kann wie beim Needleman-Wunsch-Algorithmus durch zusätzliche Zeiger realisiert werden. Die Methoden zur Speicherplatzreduktion können auch hier angewendet werden, so dass eine Berechnung mit einem Platzbedarf von $O(\min\{\ell_1, \ell_2\})$ möglich ist.

3.2 Multiple Sequence Alignment

Zum Abschluss dieses Kapitels sollen Möglichkeiten zur Erweiterung der vorgestellten Algorithmen auf n Sequenzen aufgezeigt werden. Der Needleman-Wunsch-Algorithmus lässt sich einfach modifizieren, indem statt einer 2-dimensionalen eine n-dimensionale Rekursionstabelle (*Gitter*) verwendet wird. Der Platzbedarf steigt damit auf $O(\ell_{\max}^n)$, wobei $\ell_{\max}$ die Länge der längsten Sequenz ist. In der Rekursionsformel müssen nun 2^n-1 mögliche Vorgänger betrachtet werden. Außerdem wird eine Bewertungsfunktion für multiple Alignments benötigt, wie z. B. das Sum-of-Pairs-Maß. Die Auswertung dieser Funktion erfordert Zeit $O(n^2)$ (siehe Abschnitt 2.2.2.2). Man erhält also einen Algorithmus, der in Zeit $O(n^2 \cdot 2^n \cdot \ell_{\max}^n)$ und Platz $O(\ell_{\max}^n)$ ein optimales multiples Alignment berechnet [22].

Die Erweiterung des Algorithmus von Gotoh auf den allgemeinen Fall ist leider nicht ganz so einfach. Gotoh hat einen kubischen Algorithmus für drei Sequenzen vorgestellt [16]. Wir verzichten jedoch auf dessen Beschreibung, da er im Rahmen dieser Arbeit nicht verwendet wird.

Das Hauptproblem bei affinen Lückenstrafen besteht darin, dass die Anzahl der relevanten Vorgänger, die gespeichert werden müssen, in der Größenordnung $O\left(\left[\frac{n}{e \ln 2}\right]^n \cdot \sqrt{n}\right)$ wächst [3]. Wir betrachten an dieser Stelle erneut das Sum-Of-Pairs-Maß (siehe Definition 2.8):

$$w(A) := \sum_{i=1}^{n} \sum_{j=i+1}^{n} w(A_{i,j})$$

Bei der Bestimmung von $w(A)$ werden für jedes Sequenzenpaar die Lücken bewertet, die in der entsprechenden Projektion vorkommen. Dabei fallen übereinander stehende Lücken weg (siehe Definition 2.3). Bei der Übertragung auf affine Lückenstrafen können daher folgende Fälle auftreten:

1. In einer Spalte werden entweder zwei Lücken oder zwei Zeichen aligniert. In diesem Fall wird in der zugehörigen paarweisen Projektion keine neue Lücke geöffnet.

2. Es wird eine Lücke mit einem Zeichen aligniert, d. h. die Entscheidung, ob eine neue Lücke aufgemacht wird oder nicht, hängt von den vorhergehenden Spalten des Alignments ab. Sind in der direkten Vorgängerspalte zwei Lücken aligniert, muss solange zurückgegangen werden, bis eine Spalte mit mindestens einem „richtigen“ Zeichen erreicht wird.

Vorgänger	Neue Spalte	Anzahl neuer Lücken
/	x x	0
/	– –	0
x x	– x	1
x –	– x	1
– x	– x	0
x – ··· – x – ··· –	– x	1
x – ··· – – – ··· –	– x	1
– – ··· – x – ··· –	– x	0

Tabelle 3.1: Relevante Vorgänger für zwei Sequenzen aus einem multiplen Alignment gemäß Sum-Of-Pairs-Maß mit affinen Lückenstrafen [3].

Tabelle 3.1 visualisiert diese Problematik anhand zweier Sequenzen aus einem multiplen Alignment gemäß Sum-Of-Pairs-Maß.

Altschul [3] hat daher die Verwendung einer einfacheren Lückenstrafe vorgeschlagen: die *quasi-affinen Lückenstrafen* (engl.: *Quasi Natural Gap Costs*). Hierbei wird nur anhand der direkten Vorgängerspalte des Alignments entschieden, ob eine neue Lücke geöffnet wird oder nicht. Es brauchen also nicht mehr alle möglichen Vorgänger, sondern nur noch die direkte Vorgängerspalte gespeichert werden. Tabelle 3.2 verdeutlicht den Größenunterschied des Platzbedarfs bei der Berechnung der beiden Lückenstrafen. Für paarweise Alignments sind affine und quasi-affine Lückenstrafen identisch.

Tabelle 3.3 zeigt eine Gegenüberstellung der beiden Lückenstrafen. Es kann passieren, dass quasi-affine Lückenstrafen mehr Lücken zählen als affine Lückenstrafen. Die letzten beiden Zeilen der Tabelle zeigen solche Fälle. Affine Lückenstrafen würden z. B. beim vorletzten Alignment aus Tabelle 3.3 drei Lücken zählen, wohingegen quasi-affine Kosten vier Lücken bestimmen. Dies liegt daran, dass bei quasi-affinen Lückenstrafen zwischen der ersten und zweiten Sequenz zwei Lücken erkannt werden. Die zusätzlich erkannten Lücken wurden in der Tabelle grau schattiert. Hierbei fällt auf, dass bei quasi-affinen Lückenstrafen übereinander stehende Lücken nicht notwendigerweise wegfallen.

Analog lässt sich der Unterschied im letzten Beispiel erklären. Da dieser Fehler aber verhältnismäßig gering ist, werden quasi-affine Lückenstrafen als eine gute Approximation für affine Lückenstrafen angesehen. Es gibt nur wenige Fälle, in denen sich das optimale Alignment bzgl. affiner Lückenstrafen von denen hinsichtlich quasi-affiner Lückenstrafen unterscheidet [3].

n	Affin	Quasi-affin
2	3	3
3	13	7
4	75	15
5	541	31
6	4.683	63
7	47.293	127
8	545.835	255
9	7.087.261	511
10	102.247.563	1.023

Tabelle 3.2: Anzahl relevanter Vorgänger bei affinen und quasi-affinen Lückenstrafen [3].

Alignment	Affin	Quasi-affin
x − x x x x x x x	2	2
x − x x − x x x x	2	2
x − − x x x − x x x x x	3	3
x − x x x − − x x x − x	4	4
x − − x x x x − − x x x x x x	4	4
x − x x x x x − x x x x − − x	5	5
x − − − x x x − x x x x x x x	3	4
x − − − − − x x x − x − x x x x x x x x x	4	6

Tabelle 3.3: Anzahl gezählter Lücken für affine und quasi-affine Lückenstrafen [3].

3.3 Zusammenfassung

Wir haben in diesem Kapitel verschiedene Algorithmen für die Berechnung paarweiser Alignments kennen gelernt, die auf dem Prinzip der Dynamischen Programmierung basieren und unterschiedliche Arten von Lückenstrafen berücksichtigen. Außerdem wurden Probleme, die bei der Übertragung dieser Algorithmen auf das Multiple Sequence Alignment auftreten können, aufgezeigt und Lösungsansätze hierfür vorgestellt. Wir werden im Folgenden einen anderen Ansatz zur Lösung des vorliegenden Problems betrachten.

Kapitel 4

Alignments und Kürzeste-Wege-Algorithmen

Basierend auf den Betrachtungen des vorhergehenden Kapitels werden wir nun einen graphentheoretischen Ansatz zur Berechnung optimaler multipler Alignments betrachten. Bei dieser Herangehensweise werden Wege in einem Graphen als Alignments interpretiert. Daher beginnen wir zunächst mit der Definition eines 2-dimensionalen Edit-Graphen für paarweise Alignments [4]:

Definition 4.1 (2-dimensionaler Edit-Graph) Seien $S_1 = (s_{1,1} \dots s_{1,\ell_1})$ und $S_2 = (s_{2,1} \dots s_{2,\ell_2})$ zwei Sequenzen über Σ und $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$ eine Distanzfunktion. Dann heißt der gerichtete, kantengewichtete Graph $G = (V, E, c)$ mit

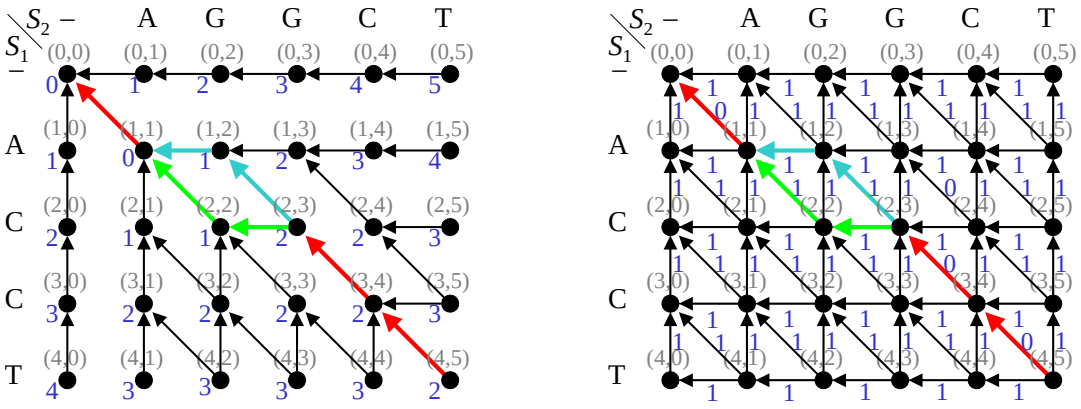
$$\begin{aligned} V &= \{0, \dots, \ell_1\} \times \{0, \dots, \ell_2\} \\ E &= \{((i,j),(i,j-1)) \mid 0 \le i \le \ell_1 \text{ und } 1 \le j \le \ell_2\} \\ &\quad \cup \{((i,j),(i-1,j)) \mid 1 \le i \le \ell_1 \text{ und } 0 \le j \le \ell_2\} \\ &\quad \cup \{((i,j),(i-1,j-1)) \mid 1 \le i \le \ell_1 \text{ und } 1 \le j \le \ell_2\} \end{aligned}$$

und

$$\begin{aligned} c : E \to \mathbb{Q} \quad \text{mit} \quad & c((i,j),(i,j-1)) = d(-, s_{2,j}), \text{ für } 0 \le i \le \ell_1, 1 \le j \le \ell_2 \\ & c((i,j),(i-1,j)) = d(s_{1,i}, -), \text{ für } 1 \le i \le \ell_1, 0 \le j \le \ell_2 \\ & c((i,j),(i-1,j-1)) = d(s_{1,i}, s_{2,j}), \text{ für } 1 \le i \le \ell_1, 1 \le j \le \ell_2 \end{aligned}$$

Edit-Graph für S_1 und S_2 bzgl. d.

Abbildung 4.1 veranschaulicht die Konstruktion eines 2-dimensionalen Edit-Graphen. Dabei zeigt Abbildung 4.1(a) die Zeigerstruktur, die beim Needleman-Wunsch-Algorithmus (siehe Abschnitt 3.1.1) unter Verwendung der Levenshtein-Distanz aus (2.2) zur Rekonstruktion des optimalen Alignments gespeichert wird. In Abbildung 4.1(b) ist der vollständige Edit-Graph bzgl. der Levenshtein-Distanz dargestellt. Die farbigen Kanten zeigen die beiden optimalen Lösungen der Beispielinstanz.



(a) Rekursionsgraph (b) Edit-Graph

Abbildung 4.1: Beispiel für die Konstruktion eines 2-dimensionalen Edit-Graphen.

Anschaulich entsprechen die horizontalen Kanten den Lücken, die in S_1 eingefügt werden, die vertikalen Kanten den Lücken in S_2. Eine diagonale Kante symbolisiert eine Alignierung von einem Zeichen aus S_1 mit einem Zeichen aus S_2, d.h. ein Match oder Mismatch. Die Knoten des Graphen entsprechen also den Matrixeinträgen $\mathrm{OPT}(i, j)$ des Needleman-Wunsch-Algorithmus, die Kanten den Rekursionseinträgen und die Kantengewichte den Kosten der entsprechenden Rekursion. Auf diese Weise erhält man einen gerichteten, azyklischen Gitter-Graphen.

Ein Weg von $v_0 = (\ell_1, \ell_2)$ nach $v_\ell = (0, 0)$ im Edit-Graphen kann als ein mögliches Alignment $A = (s'_{j,x})$ von S_1 und S_2 interpretiert werden. Die Kante (v_i, v_{i+1}) mit $v_i = (v_{i,1}, v_{i,2})$ und $v_{i+1} = (v_{i+1,1}, v_{i+1,2})$ definiert dann die $(\ell - i)$-te Spalte von A und es gilt für $j \in \{1, 2\}$:

$$s'_{j,\ell-i} = \begin{cases} - & \text{falls } v_{i,j} = v_{i+1,j} \\ s_{j,v_{i,j}} & \text{falls } v_{i,j} \neq v_{i+1,j} \end{cases} \tag{4.1}$$

Die Bewertung von A ergibt sich dann aus der Summe der Kantengewichte $c(v_i, v_{i+1})$ auf dem Weg $v_0 \xrightarrow{*} v_\ell$:

$$w(A) = \sum_{i=0}^{\ell-1} c(v_i, v_{i+1}) = \sum_{i=0}^{\ell-1} d(s'_{1,i}, s'_{2,i}) \tag{4.2}$$

Falls die Minimierungsvariante des Problems verwendet wird, entspricht der kürzeste v_0-v_ℓ-Weg einem optimalen Alignment der Sequenzen S_1 und S_2 (siehe Abbildung 4.1(b)).

Die Definition des Edit-Graphen lässt sich kanonisch auf n Sequenzen erweitern. Hierbei entsteht ein Gitter-Graph, bei dem jeder Knoten ein n-dimensionaler Vektor ist und $2^n - 1$ Nachfolger hat. Es gilt [22]:

$$|V| = O(\ell_{\max}^n) \tag{4.3}$$

$$|E| = O(|V| \cdot 2^n) = O(\ell_{\max}^n \cdot 2^n) \tag{4.4}$$

Wir definieren einen n-dimensionalen Edit-Graphen durch [29]:

Definition 4.2 (n-dimensionaler Edit-Graph) Seien $S_i = (s_{i,1} \dots s_{i,\ell_i})$ für $i \in \{1, \dots, n\}$ Sequenzen über Σ und $d : (\Sigma^+ \times \Sigma^+) \to \mathbb{Q}$ eine Distanzfunktion. Der n-dimensionale Edit-Graph $G = (V, E, c)$ für $S_1, \dots, S_n$ ist dann definiert durch:

$$\begin{aligned} V &= \{0, \dots, \ell_1\} \times \dots \times \{0, \dots, \ell_n\} \\ E &= \{(v, w) \mid v, w \in V, v \neq w, v - w \in \{0, 1\}^n - \{0\}^n\} \end{aligned}$$

Die durch eine Kante bestimmte Spalte x eines Alignments ergibt sich mit $j \in \{1, \dots, n\}$ analog zu (4.1). Die Kantengewichte $c : E \to \mathbb{Q}$ berechnen sich für $(v, w) \in E$ gemäß Sum-Of-Pairs-Maß durch:

$$c(v, w) = \sum_{j=1}^{n} \sum_{k=j+1}^{n} d(s'_{j,x}, s'_{k,x})$$

Wir haben gesehen, dass sich das Multiple Sequence Alignment als ein Kürzeste-Wege-Problem formulieren lässt. Daher betrachten wir im Folgenden das sog. *Single-Pair-Shortest-Paths-Problem* (SPSP), bei dem es darum geht, einen kürzesten Weg zwischen einem gegebenen Knotenpaar (s, t) zu bestimmen. Das SPSP stellt einen Spezialfall des *Single-Source-Shortest-Paths-Problem* (SSSP) dar, bei dem von einer gegebenen Quelle s die kürzesten Wege zu allen anderen Knoten des Graphen berechnet werden.

Es gibt bereits vielversprechende Resultate bei der Verwendung von Kürzesten-Wege-Algorithmen für die Berechnung optimaler Alignments: Gupta et al. [18] verwendeten den Algorithmus von Dijkstra [11] zur Berechnung optimaler Alignments. Lermen et al. [29, 36] wählten einen Ansatz auf Basis der A*-Suche [20].

Wir werden im Folgenden zunächst diese beiden Kürzeste-Wege-Algorithmen vorstellen. Danach präsentieren wir mit der bidirektionalen Suche eine Methode, die nach unserem Wissen bisher noch nicht zur Berechnung von Alignments eingesetzt wurde, aber im Bereich Kürzester-Wege-Algorithmen sehr gute Ergebnisse liefert [12, 13, 14]. Wir beschränken uns dabei zunächst auf lineare Lückenstrafen und werden die Verfahren im weiteren Verlauf des Kapitels auf quasi-affine Lückenstrafen erweitern.

4.1 Algorithmus von Dijkstra

Der Algorithmus von Dijkstra [11] löst das SSSP auf einem gerichteten Graphen mit nicht-negativen Kantengewichten. Dabei werden für jeden Knoten folgende Informationen verwaltet:

- $\delta_s(v) \in \mathbb{N}$: Distanzlabel, das die Länge des kürzesten bisher bekannten s-v-Weges angibt.
- $\pi_s(v) \in V$: Vorgängerknoten von v auf diesem Weg.

Mit $\text{dist}_s[v] \in \mathbb{N}$ wird die tatsächliche Länge eines kürzesten s-v-Weges bezeichnet. Es gilt stets $\delta_s(v) \geq \text{dist}_s[v]$, d. h. $\delta_s(v)$ ist eine obere Schranke für die Länge eines kürzesten s-v-Weges. Sobald der Algorithmus terminiert, muss $\delta_s(v) = \text{dist}_s[v]$ gelten.

Die Variablen werden für alle $v \in V$ mit $\delta_s(v) = \infty$ und $\pi_s(v) = \texttt{NIL}$ initialisiert. Ein Knoten v wird im Laufe des Algorithmus *vorläufig markiert*, sobald eine obere Schranke kleiner als unendlich für die Entfernung von s nach v bekannt ist, also $\delta_s(v) < \infty$ gilt. Ein Knoten wird *endgültig markiert*, wenn der kürzeste s-v-Weg gefunden wurde. Knoten v mit $\delta_s(v) = \infty$ heißen *unerreicht*. Jeder Knoten kann sich also in einem von drei Markierungszuständen befinden. Daher wird der Algorithmus von Dijkstra in der Literatur auch als *Markierungsalgorithmus* [50] bezeichnet.

Der Algorithmus beginnt beim Startknoten s: Der Knoten wird vorläufig markiert und $\delta_s(s) = 0$ gesetzt. Solange noch vorläufig markierte Knoten existieren, wird einer von ihnen – o. B. d. A. sei dies der Knoten v – ausgewählt und bearbeitet, d. h. er wird endgültig markiert und alle von v ausgehenden Kanten werden *relaxiert*. Bei der *Relaxation* einer Kante (v, w) wird überprüft, ob der Weg von s zu w über v kürzer ist als der bisher kürzeste s-w-Weg, d. h. ob $\delta_s(w) > \delta_s(v) + c(v, w)$ gilt. Ist dies der Fall, werden die entsprechenden Variablen durch $\delta_s(w) = \delta_s(v) + c(v, w)$ und $\pi_s(w) = v$ aktualisiert. Der Algorithmus terminiert, sobald keine vorläufig markierten Knoten mehr existieren. Die Vorgänger $\pi_s(v)$ definieren dann einen Kürzeste-Wege-Baum und $\delta_s(v)$ gibt die Länge eines kürzesten s-v-Weges an.

Die Effizienz des Verfahrens hängt davon ab, wie der nächste endgültig zu markierende Knoten ausgewählt wird. Der Algorithmus von Dijkstra [11] gehört dabei zur Klasse der *Greedy-Algorithmen* [9] und wählt jeweils den Knoten v mit minimalem Distanzlabel $d_s(v)$. Daher wird für die Verwaltung der Menge aller vorläufig markierten Knoten v eine Prioritätswarteschlange Q verwendet, die folgende Operationen unterstützt:

- `INSERT`$(v, \delta_s(v))$: Fügt v mit Priorität $\delta_s(v)$ in Q ein.
- `EXTRACT-MIN()`: Gibt den Knoten mit minimaler Priorität zurück und entfernt ihn aus Q.
- `DECREASE-KEY`$(v, \delta_s(v))$: Aktualisiert die Priorität von v in Q auf $\delta_s(v)$.

In Algorithmus 1 ist der Algorithmus von Dijkstra für das SSSP in Pseudocode dargestellt. Wir sind für unsere Anwendung nur an einem kürzesten Weg von $s = (\ell_1, \ldots, \ell_n)$ nach $t = (0, \ldots, 0)$ interessiert. Daher können die Berechnungen abgebrochen werden, sobald t endgültig markiert wurde. Auf diese Weise erhält man eine Variante des Algorithmus von Dijkstra für das SPSP.

Voraussetzung für die korrekte Arbeitsweise des Algorithmus sind nicht-negative Kantengewichte. Die Methode kann so modifiziert werden, dass sie auch negative Kantengewichte bearbeiten kann (Bellman-Ford-Algorithmus, [9]). Da im vorliegenden Fall sämtliche Kantengewichte nicht-negativ sind, ist dies für die weitere Betrachtungen jedoch nicht relevant.

Algorithmus 1 Algorithmus von Dijkstra

1. Initialisierung:
   (a) $Q = \emptyset$
   (b) $\delta_s(s) = 0$, $\pi_s(s) = \texttt{NIL}$, $\texttt{INSERT}(s, 0)$, s wird vorläufig markiert
   (c) `FOR ALL` $v \in V$, $v \neq s$:
   $\delta_s(v) = \infty$, $\pi_s(v) = \texttt{NIL}$
2. `WHILE` $Q \neq \emptyset$:
   (a) $v =$ `EXTRACT-MIN()`, v wird endgültig markiert
   (b) `FOR ALL` $(v, w) \in E$:
      i. w wird vorläufig markiert
      ii. `If` $\delta_s(w) > \delta_s(v) + c(v, w)$ `THEN`
         A. $\delta_s(w) = \delta_s(v) + c(v, w)$
         B. `IF` $\pi_s(w) = \texttt{NIL}$ `THEN`
            `INSERT`$(w, \delta_s(w))$
            `ELSE`
            `DECREASE-KEY`$(w, \delta_s(w))$
         C. $\pi_s(w) = v$

Korrektheit

Die Korrektheit des Algorithmus von Dijkstra basiert auf dem sog. *Optimalitätsprinzip* [9]:

Satz 4.3 (Optimalitätsprinzip) Sei $G = (V, E, c)$ ein gerichteter Graph mit Kantengewichten $c : E \to \mathbb{R}$ und $p = v_0 \xrightarrow{*} v_\ell$ ein kürzester v_0-v_ℓ-Weg. Dann ist jeder Teilweg $p_{ij} = v_i \xrightarrow{*} v_j$ ein kürzester v_i-v_j-Weg.

Beweis: Der Weg p kann wie folgt zerlegt werden:

$$v_0 \xrightarrow{*} v_i \xrightarrow{*} v_j \xrightarrow{*} v_\ell$$

Mit $p_{0i} = v_0 \xrightarrow{*} v_i$, $p_{ij} = v_i \xrightarrow{*} v_j$ und $p_{j\ell} = v_j \xrightarrow{*} v_\ell$ beträgt die Länge $\lambda(p)$ dann:

$$\lambda(p) = \lambda(p_{0i}) + \lambda(p_{ij}) + \lambda(p_{j\ell})$$

Angenommen, es gäbe einen Weg p'_{ij} mit $\lambda(p'_{ij}) < \lambda(p_{ij})$. Dann folgt:

$$\lambda(p') = \lambda(p_{0i}) + \lambda(p'_{ij}) + \lambda(p_{j\ell}) < \lambda(p)$$

Dies ist jedoch im Widerspruch zur Annahme, dass p ein kürzester v_0-v_ℓ-Weg ist. □

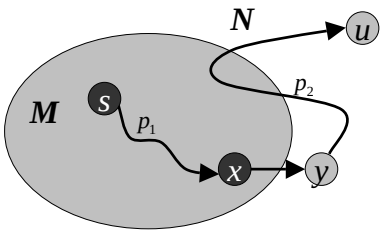


Abbildung 4.2: Korrektheitsbeweis des Algorithmus von Dijkstra [9].

Wir beweisen nun die Korrektheit des Algorithmus von Dijkstra. Dafür muss gezeigt werden, dass nach Terminierung für alle Knoten $v \in V$ $\delta_s(v) = \text{dist}_s[v]$ gilt. Da die Distanzlabel $\delta_s(v)$ während der Berechnungen nie erhöht werden und stets $\delta_s(v) \geq \text{dist}_s[v]$ gilt, genügt es zu zeigen, dass $\delta_s(v) = \text{dist}_s[v]$ bei der Extrahierung von v erfüllt ist. Hieraus folgt direkt, dass jeder Knoten maximal einmal aus der Prioritätswarteschlange extrahiert wird. Die Knotenmenge V zerfällt damit in zwei disjunkte Teilmengen M und $N = V \backslash M$, wobei M alle endgültig markierten Knoten enthält.

Satz 4.4 (Korrektheit [9]) Wenn $v \in V$ aus Q extrahiert wird, gilt:

$$\delta_s(v) = \text{dist}_s[v]$$

Beweis: Wir zeigen die Behauptung durch Widerspruch. Sei dazu u der erste Knoten, für den die Behauptung nicht gilt, d. h. $\delta_s(u) > \text{dist}_s[u]$. Es ist $u \neq s$, da s der erste endgültig markierte Knoten ist und $\delta_s(s) = \text{dist}_s[s] = 0$ nach Initialisierung. Wenn u endgültig markiert wird, gilt $\delta_s(u) \leq \delta_s(w)$ für alle Knoten $w \in N$, da stets ein Knoten mit minimalem Distanzlabel aus Q extrahiert wird.

Des Weiteren muss es mindestens einen kürzesten s-u-Weg p geben, da sonst $\delta_s(u) = \text{dist}_s[u] = \infty$ im Widerspruch zur Annahme gilt. Sei $y \in N$ der erste noch nicht endgültig markierte Knoten auf p und $x \in M$ der Vorgängerknoten von y. Der Weg p kann dann wie folgt zerlegt werden (siehe Abbildung 4.2):

$$\underbrace{s \xrightarrow{*} x}_{p_1} \rightarrow \underbrace{y \xrightarrow{*} u}_{p_2}$$

Dabei kann sowohl p_1 als auch p_2 leer sein, d. h. $s = x$ oder $y = u$ gelten.

Da x vor u aus Q extrahiert wurde und u nach Annahme der erste Knoten mit $\delta_s(u) > \text{dist}_s[u]$ ist, gilt $\delta_s(x) = \text{dist}_s[x]$. Bei der Extrahierung von x wurde (x, y) relaxiert und damit der Weg $p_{sy} = s \xrightarrow{*} x \rightarrow y$ gefunden. Da es sich hierbei um einen Teilweg von p handelt, muss p_{sy} nach Satz 4.3 ein kürzester s-y-Weg sein. Es gilt demnach $\delta_s(y) = \text{dist}_s[y]$.

Es gilt weiter $\text{dist}_s[y] \leq \text{dist}_s[u]$, da alle Kantengewichte nicht-negativ sind, und daher:

$$\delta_s(y) = \text{dist}_s[y] \leq \text{dist}_s[u] \leq \delta_s(u)$$

Da aber u vor y extrahiert wurde, folgt $\delta_s(u) = \delta_s(y)$ und damit im Widerspruch zur Annahme $\delta_s(u) = \text{dist}_s[u]$. □

Operation	unsortierte Liste	Binärer Heap	Fibonacci Heap
`INSERT`$(v, \delta_s(v))$	$O(1)$	$\log(\|V\|)$	$O(1)$ (amortisiert)
`EXTRACT-MIN()`	$O(\|V\|)$	$\log(\|V\|)$	$\log(\|V\|)$
`DECREASE-KEY`$(v, \delta_s(v))$	$O(\|V\|)$	$\log(\|V\|)$	$O(1)$ (amortisiert)

Tabelle 4.1: Laufzeiten für verschiedene Prioritätswarteschlangen.

Laufzeit

Die Wahl der Prioritätswarteschlange hat entscheidende Auswirkungen auf die Laufzeit des Algorithmus. In der Originalarbeit von Dijkstra [11] wurde eine einfache lineare Liste verwendet. Mit Hilfe fortgeschrittener Datenstrukturen ist es jedoch möglich, effizientere Algorithmen zu entwerfen [8, 28]. Wir verwenden in unsere Implementierung (siehe Abschnitt 6.1) einen Binärheap [9].

Tabelle 4.1 gibt einen Überblick über die Laufzeiten von ausgewählten Prioritätswarteschlangen für die benötigten Operationen. Ein detaillierter experimenteller Vergleich verschiedener Datenstrukturen befindet sich in [8]. Für weiterführende Literatur sei u. a. auf [9] verwiesen.

Die Laufzeit des Algorithmus wird dominiert von den Laufzeiten der Operationen auf der Prioritätswarteschlange und setzt sich aus folgenden Termen zusammen:

- $|V| \cdot T(\texttt{INSERT}(v, \delta_s(v)))$, da jeder Knoten genau einmal in Q eingefügt wird.
- $|V| \cdot T(\texttt{EXTRACT-MIN}())$, da jeder Knoten genau einmal extrahiert wird.
- $|E| \cdot T(\texttt{DECREASE-KEY}(v, \delta_s(v)))$, da während der Kanten-Relaxation für jeden Knoten jede ausgehende Kante genau einmal betrachtet wird. Für jede Kante wird also höchstens einmal die Priorität eines Knotens in Q gesenkt.

Außerdem wird vorausgesetzt, dass für jeden Knoten die aktuelle Position in der Prioritätswarteschlange bekannt ist, so dass ein Zugriff bei einer `DECREASE-KEY`-Operation in konstanter Zeit realisierbar ist. Mit den Laufzeiten aus Tabelle 4.1 ergeben sich dann folgende Gesamtlaufzeiten für den Algorithmus von Dijkstra:

- Unsortierte Liste:

$$|V| \cdot O(1) + |V| \cdot O(|V|) + |E| \cdot O(1) = O(|V|^2 + |E|) = O(|V|^2)$$

- Binärheap:

$$\begin{aligned} & |V| \cdot \log(|V|) + |V| \cdot \log(|V|) + |E| \cdot \log(|V|) \\ = \; & O((|V| + |E|) \cdot \log(|V|)) = O(|E| \cdot \log(|V|)) \end{aligned}$$

- Fibonacci-Heap:

$$|V| \cdot O(1) + |V| \cdot \log(|V|) + |E| \cdot O(1) = O(|E| + |V| \log(|V|))$$

Wir analysieren abschließend die benötigte Laufzeit auf dem gegebenen Edit-Graphen. Hierbei müssen wir beachten, dass für jede betrachtete Kante zunächst eine Berechnung der Kantenkosten nötig ist. Bei Verwendung des Sum-Of-Pairs-Maßes erhalten wir also pro Kante zusätzliche Kosten von $O(n^2)$. Mit (4.3) und (4.4) ergibt sich dann folgende Laufzeit zur Berechnung eines kürzesten Weges auf einem n-dimensionalen Edit-Graphen mit Fibonacci-Heaps:

$$\begin{aligned} O(|E| + |V| \cdot \log(|V|)) &= O(n^2 \cdot \ell_{\max}^n \cdot 2^n + \ell_{\max}^n \cdot \log(\ell_{\max}^n)) \\ &= O(n^2 \cdot \ell_{\max}^n \cdot 2^n) \end{aligned}$$

Die Laufzeiten bei Verwendung eines Binärheaps oder einer unsortierten Liste sind sogar höher. Im Vergleich zum Needleman-Wunsch-Algorithmus (siehe Kapitel 3.1.1) erhalten wir also keine theoretische Verbesserung, da dieser ebenfalls in Zeit $O(n^2 \cdot 2^n \cdot \ell_{\max}^n)$ ein optimales multiples Alignment berechnet. Experimente zeigen jedoch eine deutliche Beschleunigung. Dies kann dadurch erklärt werden, dass beim Algorithmus von Dijkstra nicht notwendigerweise der ganze Edit-Graph betrachtet wird. Die Dynamische Programmierung hingegen berechnet stets die komplette Rekursionstabelle.

4.2 A*-Suche

Die A*-Suche wurde erstmalig von Hart et al. [20] vorgestellt. Im Gegensatz zum Algorithmus von Dijkstra [11] stellt die A*-Suche eine sog. *informierte Suchstrategie* dar, die neben der Problemdefinition zusätzliches problemspezifisches Wissen nutzt [37]. Auf diese Weise erhält man zwar keine theoretischen Laufzeitverbesserungen, kann in der Praxis aber häufig effizienter optimale Lösungen berechnen als bei *uniformierten Strategien*. In der Literatur findet sich auch die Bezeichnung *Goal Directed Unidirectional Search* (GDUS) [28] für die A*-Suche, da zunächst diejenigen Knoten untersucht werden, die am wahrscheinlichsten zum Ziel führen.

Die A*-Suche basiert auf dem Algorithmus von Dijkstra. Es wird zusätzlich eine Funktion $h_t : V \to \mathbb{Q}$ benutzt, die den kürzesten Weg von einem Knoten u zur Senke t abschätzt [12] und daher *Schätzfunktion* genannt wird. Bei der Kanten-Relaxation wird dann für jeden Nachfolger v von u der Wert von h_t berechnet und v mit einem modifizierten Distanzlabel $\delta'_s(v)$ in die Prioritätswarteschlange eingefügt:

$$\delta'_s(v) := \delta_s(v) + h_t(v)$$

Die Schätzfunktion h_t ist eine Potenzialfunktion auf der Menge der Knoten. Mit Hilfe dieser Potenzialfunktion können die Kantengewichte im Edit-Graphen wie folgt modifiziert werden [12]:

$$c_{h_t}(u, v) := c(u, v) - [h_t(u) - h_t(v)]$$

Wenn wir die Kantenkosten $c : E \to \mathbb{Q}$ durch $c_{h_t} : E \to \mathbb{Q}$ ersetzen, reduzieren sich die Kosten eines jeden u-v-Weges um $h_t(v) - h_t(u)$. Dies gilt insbesondere auch für die Menge der kürzesten s-t-Wege.

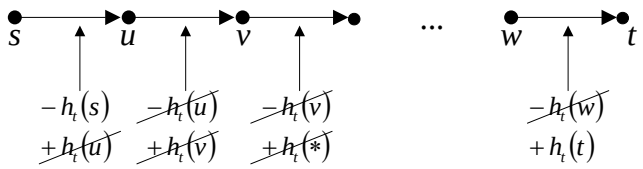


Abbildung 4.3: Modifikation der Kantenkosten durch die A*-Suche.

Satz 4.5 Die Modifikation der Kantengewichte ändert die Menge der kürzesten s-t-Wege nicht.

Beweis: Wir betrachten einen beliebigen s-t-Weg p. Abbildung 4.3 visualisiert, wie die Kantenkosten auf diesem Weg modifiziert werden. Die Länge von p ergibt sich dann durch die Teleskopsumme:

$$w(p) = \sum_{(u,v)\in p} [c(u,v) - h_t(u) + h_t(v)] = \left[\sum_{(u,v)\in p} c(u,v)\right] - h_t(s) + h_t(t)$$

Da $h_t(s)$ für alle s-t-Wege gleich ist und $h_t(t) = 0$ gilt, ist die Länge eines jeden s-t-Weges bzgl. der modifizierten Kostenfunktion um eine Konstante $h_t(s)$ kleiner als hinsichtlich der ursprünglichen Kostenfunktion. Demnach ändert sich die Menge der kürzesten s-t-Wege nicht. □

Wird h_t so gewählt, dass sämtliche modifizierte Kantengewichte nicht-negativ sind, kann auf der so modifizierten Instanz $G' = (V,\ E,\ c_{h_t})$ der Algorithmus von Dijkstra ausgeführt werden. Es ist leicht zu sehen, dass die A*-Suche auf G äquivalent zum Algorithmus von Dijkstra auf G' ist.

Analog zu [37] definieren wir zwei verschiedene Arten von Schätzfunktionen:

Zulässige (optimistische) Schätzfunktion: Eine Schätzfunktion $h_t : V \to \mathbb{Q}$ heißt *zulässig* oder *optimistisch*, wenn sie den kürzesten Weg von $v \in V$ nach $t \in V$ nie überschätzt, d. h. es gilt:

$$h_t(v) \leq \text{dist}_t[v] \tag{4.5}$$

Eine optimistische Schätzfunktion stellt also eine untere Schranke für die Länge des Weges $v \xrightarrow{*} t$ dar.

Konsistente (monotone) Schätzfunktion: Eine optimistische Schätzfunktion $h_t : V \to \mathbb{Q}$ heißt *konsistent* oder *monoton*, wenn die geschätzten Kosten $h_t(u)$ von einem Knoten u zum Ziel t nie größer sind als die geschätzten Kosten $h_t(v)$ eines beliebigen Nachfolgers v von u plus die tatsächlichen Kosten $c(u,v)$, um direkt von u nach v zu gelangen, d. h. es gilt:

$$h_t(u) \leq c(u,v) + h_t(v) \tag{4.6}$$

Konsistente Schätzfunktionen stellen sicher, dass alle modifizierten Kantengewichte nicht-negativ sind, da

$$h_t(u) \leq c(u,v) + h_t(v) \Leftrightarrow c(u,v) \geq h_t(u) - h_t(v)$$

und somit

$$c_{h_t}(u,v) = c(u,v) - [h_t(u) - h_t(v)] \geq 0.$$

Aus diesem Grund beschränken wir uns auf die Verwendung von konsistenten Schätzfunktionen.

Die Effizienz der A*-Suche hängt von der Güte der verwendeten Schätzfunktion ab. Hierbei existiert ein Trade-Off zwischen Güte der Schätzfunktion und der benötigten Rechenzeit zur Bestimmung dieser Schätzfunktion. Falls die Schätzfunktion den exakten kürzesten Weg von v nach t bestimmt, extrahiert die A*-Suche nur genau die Knoten, die auf einem kürzesten Weg von s nach t liegen. Die Verwendung einer solchen Schätzfunktion ist aber in der Regel nicht sinnvoll, da in diesem Fall allein durch die Berechnung der Schätzfunktion das ursprüngliche Problem gelöst wird. Andererseits führt die Verwendung einer sehr schlechten Schätzfunktion dazu, dass sich die A*-Suche nicht wesentlich vom Algorithmus von Dijkstra unterscheidet. Der Algorithmus von Dijkstra entspricht einer A*-Suche mit $h_t(v) = 0$ für alle $v \in V$.

Im Folgenden beschreiben wir zwei Schätzfunktionen, die den oben genannten Anforderungen gerecht werden. Anschließend stellen wir vor, wie mit Hilfe dieser Schätzfunktionen weitere Suchraumreduktionen für unser Problem vorgenommen werden können.

4.2.1 Verwendete Schätzfunktionen

Sei $w^*(S_i, S_j)$ der Wert eines optimalen paarweisen Alignments der Sequenzen S_i und S_j und $w(A_{i,j})$ der Wert der paarweisen Projektion eines – gemäß Sum-Of-Pairs-Maß – optimalen multiplen Alignment A^* auf S_i und S_j. Die beiden verwendeten Schätzfunktionen basieren auf folgender Beobachtung:

$$w^*(S_i, S_j) \leq w(A_{ij}) \tag{4.7}$$

Der Wert des optimalen paarweisen Alignments ist also eine untere Schranke für den Wert der paarweisen Projektion (siehe Bemerkung 2.9).

4.2.1.1 Paarweise Alignments als untere Schranke

Eine einfache Schätzfunktion verwendet direkt (4.7). Jeder einzelne Term des Sum-Of-Pairs-Maßes wird mit Hilfe der Werte für das zugehörige optimale paarweise Alignment abgeschätzt. Man erhält so eine untere Schranke für den Wert $w(A)$ eines optimalen multiplen Alignments:

$$w(A) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} w(A_{ij}) \geq \sum_{i=1}^{n} \sum_{j=i+1}^{n} w^*(S_i, S_j)$$

Ein Knoten $v = (v_1, \ldots, v_n)$ teilt jede Sequenz in einen Präfix und in einen Suffix, so dass ein zugehöriges Gesamtalignment aus einer Alignierung der entsprechenden Präfixe konkateniert mit einer Alignierung der Suffixe besteht. Da $t = (0, \ldots, 0)$ ist, entspricht ein Weg von v nach t einer Alignierung der durch v definierten Präfixe $\alpha_i^{v_i}$, die jeweils die ersten v_i Zeichen umfassen. Um eine untere Schranke für den kürzesten Weg von v nach t zu bestimmen, genügt es demnach, eine untere Schranke $h_t(v)$ für das Alignment der Präfixe $\alpha_1^{v_1}, \alpha_2^{v_2}, \ldots, \alpha_n^{v_n}$ zu bestimmen. Wir definieren:

$$h_t(v) := \sum_{i=1}^{n} \sum_{j=i+1}^{n} w^*(\alpha_i^{v_i}, \alpha_j^{v_j}) \tag{4.8}$$

Es lässt sich einfach zeigen, dass die Konsistenzbedingung

$$h_t(u) \leq c(u, v) + h_t(v)$$

für (4.8) erfüllt ist.

Satz 4.6 Die in (4.8) definierte Schätzfunktion ist konsistent [29].

Beweis: Da $\alpha_i^{u_i}$ ein Teilstring von $\alpha_i^{v_i}$ ist, gilt:

$$w^*(\alpha_i^{v_i}, \alpha_j^{v_j}) \geq w^*(\alpha_i^{u_i}, \alpha_j^{u_j})$$

Daraus folgt:

$$\begin{aligned} c(u, v) + h_t(v) &= c(u, v) + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w^*(\alpha_i^{v_i}, \alpha_j^{v_j}) \\ &\geq \sum_{i=1}^{n} \sum_{j=i+1}^{n} w^*(\alpha_i^{u_i}, \alpha_j^{u_j}) \\ &= h_t(u). \end{aligned}$$

□

4.2.1.2 Dreifache Alignments als untere Schranke

Lermen et al. haben in [29] eine Verbesserung der unteren Schranken vorgeschlagen. Dabei werden bei der Berechnung der Schätzfunktion $h_t : V \to \mathbb{Q}$ die Werte von den drei optimalen paarweisen Alignments der Sequenzen S_i, S_j und S_k durch den Wert des optimalen (multiplen) Alignments von S_i, S_j und S_k ersetzt. Wir bezeichnen das Alignment eines Tripels von drei Sequenzen im Folgenden als *dreifaches Alignment*.

Durch diese Ersetzung können wir erwarten, eine höhere untere Schranke zu erhalten. Wir können aus folgenden Beobachtungen schließen, dass die so modifizierte Schätzfunktion $h_t^3 : V \to \mathbb{Q}$ zulässig ist, d. h. eine gültige untere Schranke liefert. Außerdem erhalten wir keine kleinere untere Schranke:

- Der Wert $w^*(S_i, S_j, S_k)$ eines optimalen Alignments von drei Sequenzen ist mindestens so groß wie die Summe der Bewertungen der entsprechenden optimalen paarweisen Alignments, d. h. es gilt:

$$w^*(S_i, S_j, S_k) \geq w^*(S_i, S_j) + w^*(S_i, S_k) + w^*(S_j, S_k) \tag{4.9}$$

- Die Kosten für die dreifache Projektion $A_{i,j,k}$ eines optimalen multiplen Alignment A^* auf S_i, S_j und S_k sind mindestens so groß wie die Bewertung des optimalen Alignments der drei Sequenzen, d. h. es gilt:

$$w^*(S_i, S_j, S_k) \leq w(A_{i,j,k}) \tag{4.10}$$

Um möglichst gute Schranken zu erhalten, müssen geeignete Tripel ausgewählt werden. Die Menge T dieser Tripel darf keine zwei Elemente enthalten, die zwei Sequenzen gemeinsam haben, weil die Alignierung dieser beiden Sequenzen sonst doppelt in die Schätzfunktion eingeht und h_t^3 nicht mehr zulässig wäre. Da mit dieser Bedingung nicht alle möglichen Tripel abgedeckt werden können, werden zusätzlich zu den dreifachen Alignments auch weiterhin einige optimale paarweise Alignments berücksichtigt. Hierfür werden die Paare ausgewählt, die durch kein Tripel repräsentiert werden. Als Hilfsmenge werden alle bereits durch T repräsentierten Paare in einer Menge P verwaltet. Formal lassen sich T und P wie folgt definieren [29]:

$$\begin{aligned} T &:= \{\{i,j,k\} \mid i,j,k \in \{1,\ldots,n\}, i \neq j \neq k\} \text{ mit} \\ &\quad\ \forall a, b \in T, a \neq b : |a \cap b| < 2 \\ P &:= \{(i,j),(j,k),(i,k) \mid \forall \{i,j,k\} \in T\} \end{aligned}$$

Mit Hilfe dieser beiden Mengen definieren wir eine neue Schätzfunktion $h_t^3 : V \to \mathbb{Q}$ [29]. Dabei bezeichnet A^* ein optimales Alignment der entsprechenden Sequenzen und $\alpha_i^{v_i}$ den Präfix von S_i, der die ersten v_i Zeichen umfasst:

$$h_t^3(v) := \sum_{\{i,j,k\} \in T} c(A^*(\alpha_i^{v_i}, \alpha_j^{v_j}, \alpha_k^{v_k})) + \sum_{(i,j) \notin P} c(A^*(\alpha_i^{v_i}, \alpha_j^{v_j})) \tag{4.11}$$

$$\overset{(4.9)}{\geq} \sum_{i=1}^{n} \sum_{j=i+1}^{n} c(A^*(\alpha_i^{v_i}, \alpha_j^{v_j})) = h_t(v) \tag{4.12}$$

Analog zu h_t kann für h_t^3 die Konsistenzbedingung gezeigt werden. In der Regel liefert h_t^3 jedoch eine bessere untere Schranke als h_t. Die Güte der Schranke hängt dabei von der Wahl der Tripel ab. Um eine möglichst hohe untere Schranke zu erhalten, sollten diejenigen Tripel (S_i, S_j, S_k) ausgewählt werden, für die die Differenz aus dem Wert eines zugehörigen optimalen dreifachen Alignments und der Summe der Werte der entsprechenden optimalen paarweisen Alignments maximal ist. In [29] wird daher folgende Heuristik vorgeschlagen, die nach Lermen et al. gute Resultate liefert:

1. Berechne für jedes Tripel $\{i,j,k\}$:

$$D(i,j,k) := c(A^*(S_i, S_j, S_k)) - c(A^*(S_i, S_j)) - c(A^*(S_j, S_k)) - c(A^*(S_i, S_k))$$

2. Wähle Tripel $\{i, j, k\}$ falls:
   - $D(i, j, k) > 0$
   - $\nexists\, \{i', j', k'\} : |\{i, j, k\} \cap \{i', j', k'\}| \geq 2 \quad \text{und} \quad D(i', j', k') \geq D(i, j, k)$

4.2.2 Pruning

Beim Pruning werden Knoten, die beweisbar nicht auf einem kürzesten s-t-Weg liegen können, vorzeitig ausgeschlossen. In unserem Fall heißt dies, dass derartige Knoten während der Kanten-Relaxation nicht in die Prioritätswarteschlange eingefügt werden.

Es gibt viele effiziente Verfahren, um ein suboptimales Alignment zu berechnen [19, 32]. Mit Hilfe eines solchen Algorithmus kann in einem Vorverarbeitungsschritt eine obere Schranke μ für den Wert des optimalen multiplen Alignments berechnet werden. Bei der Relaxation einer Kante (u, v) wird dann unter Verwendung der Schätzfunktion h_t eine untere Schranke für einen s-t-Weg $p = (s, \ldots, u, v, \ldots, t)$ bestimmt:

$$w(p) \geq d_s(u) + c(u, v) + h_t(v)$$

Falls $d_s(u)+c(u, v)+h_t(v) \geq \mu$ gilt, kann v nicht auf einem kürzesten s-t-Weg liegen, da bereits ein besserer Weg bekannt ist. Der Knoten v kann also ignoriert werden.

4.3 Bidirektionale Suche

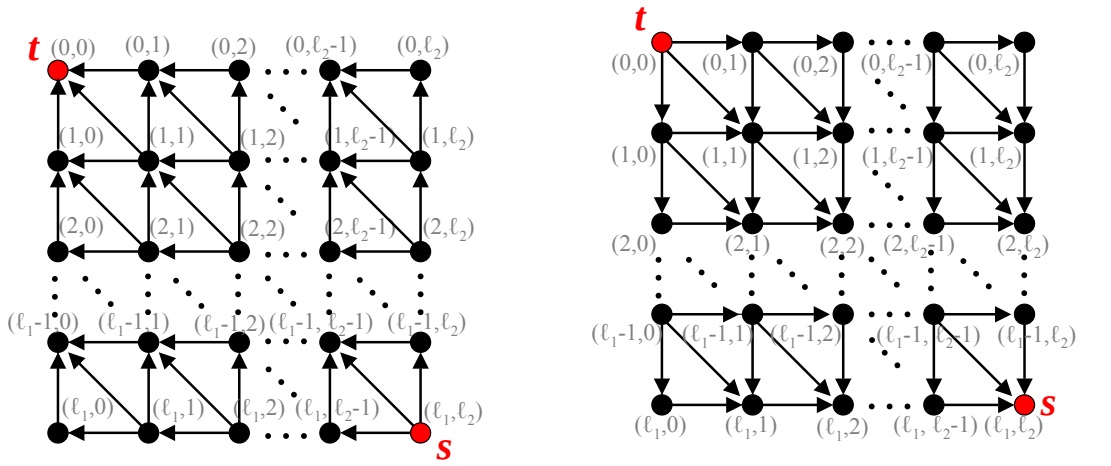
Wir wollen nun eine weitere Modifikation der Berechnung kürzester Wege betrachten. Hierbei wird die Idee verfolgt, zwei simultane Suchprozesse (einen vom Startknoten s und einen vom Zielknoten t aus) durchzuführen und durch geeignete Konkatenation der beiden Suchrichtungen einen kürzesten s-t-Weg zu bestimmen. Diese Vorgehensweise wird als bidirektionale Suche [35] bezeichnet. Sie kann sowohl für den Algorithmus von Dijkstra als auch für die A*-Suche angewendet werden. In beiden Fällen wird die Suche von t aus auf dem zu G inversen Graphen G^{-1} ausgeführt. Der Graph $G^{-1} = (V^{-1}, E^{-1}, c^{-1})$ entsteht dabei durch Umdrehen aller Kanten aus $G = (V, E, c)$. Er entspricht dem Edit-Graphen für die zu $S_1, \ldots, S_n$ inversen Sequenzen $S_1^{-1}, \ldots, S_n^{-1}$ und lässt sich wie folgt formalisieren:

$$\begin{aligned} V^{-1} &= V \\ E^{-1} &= \{(v, u) \mid (u, v) \in E\} \end{aligned}$$

und

$$c^{-1} : E \to \mathbb{Q} \quad \text{mit} \quad c^{-1}(v, u) = c(u, v)$$

Abbildung 4.4 zeigt eine schematische Darstellung der beiden Edit-Graphen. Ein kürzester t-s-Weg in G^{-1} entspricht einem kürzesten s-t-Weg in G. Daraus folgt, dass ein optimales Alignment für $S_1, \ldots, S_n$ auch ein optimales Alignment für $S_1^{-1}, \ldots, S_n^{-1}$ ist und umgekehrt.



(a) Edit-Graph $G = (V, E, c)$ (b) Inverser Edit-Graph $G^{-1} = (V^{-1}, E^{-1}, c^{-1})$

Abbildung 4.4: Edit-Graph für Vorwärts- und Rückwärtssuche mit $n = 2$.

Im Folgenden bezeichne *Vorwärtssuche* den Algorithmus auf G und *Rückwärtssuche* den auf G^{-1}. Die Rückwärtssuche verwendet Distanzlabel $\delta_t(v)$. Die tatsächlichen Längen kürzester t-v-Wege werden mit $\text{dist}_t[v]$, die Prioritätswarteschlange mit Q^{-1} bezeichnet. Falls eine bidirektionale A*-Suche durchgeführt wird, kommt zusätzlich eine Schätzfunktion $h_s : V \to \mathbb{Q}$ zum Einsatz.

Die Schätzfunktion h_s und die zugehörigen modifizierten Kantenkosten $c_{h_s} : E \to \mathbb{Q}$ der Rückwärtssuche können analog zur Vorwärtssuche berechnet werden (siehe Abschnitt 4.2). Anstelle der durch $v = (v_1, \ldots, v_n)$ definierten Präfixe werden die entsprechenden Suffixe $\sigma_i^{v_i}$ der Sequenzen verwendet. Dabei umfasst $\sigma_i^{v_i}$, die letzten $\ell_i - v_i$ Zeichen der Sequenzen:

$$\begin{aligned} h_s(v) &:= \sum_{i=1}^{n} \sum_{j=i+1}^{n} w(\sigma_i^{v_i}, \sigma_j^{v_j}) \\ c_{h_s}^{-1}(u, v) &:= c^{-1}(u, v) - [h_s(u) - h_s(v)] \end{aligned}$$

Wir wollen nun parallel eine Vorwärtssuche von s und eine Rückwärtssuche von t aus durchzuführen. Dabei verfolgen wir einerseits das Ziel, einen möglichst ähnlichen Aufwand für Vorwärts- und Rückwärtssuche zu erhalten. Unter Aufwand verstehen wir dabei nicht nur die Anzahl der jeweils extrahierten Knoten, sondern vielmehr die Gesamtrechenzeit der beiden Suchprozesse. Diese wird neben der Anzahl extrahierter Knoten auch von der Größe der Prioritätswarteschlangen beeinflusst. Andererseits können die Distanzlabel der extrahierten Knoten das Terminierungskriterium und somit die Gesamtlaufzeit beeinflussen. Wir unterscheiden daher verschiedene Strategien, um zwischen den beiden Suchrichtungen zu alternieren:

alt: Es wird abwechselnd jeweils ein Knoten in Vorwärts- und Rückwärtsrichtung aus der entsprechenden Prioritätswarteschlange extrahiert.

minQ: Auswahl der Richtung mit der kleineren Prioritätswarteschlange.

minP: Auswahl der Richtung, in der das nächste zu extrahierende Element den kleineren Schlüssel hat.

Die durch Vorwärts- und Rückwärtssuche gefundenen Wege müssen geeignet konkateniert werden. Dabei bezeichne μ die Länge des bisher kürzesten gefundenen s-t-Weges: Falls bei der Relaxation der Kante (v, w) der Knoten w bereits von der entgegengesetzten Suchrichtung endgültig markiert wurde, kennen wir einen kürzesten s-v-Weg und einen kürzesten w-t-Weg. Der Weg $s \xrightarrow{*} v \to w \xrightarrow{*} t$ ist demnach kostenminimal unter allen s-t-Wegen, die die Kante (v, w) benutzen [35]. Die Länge dieses Weges beträgt:

$$\kappa = \delta_s(v) + c(v, w) + \delta_t(w)$$

Falls $\mu > \kappa$ gilt, wurde ein neuer kürzester s-t-Weg gefunden. Wir aktualisieren μ dementsprechend und setzen die Suche fort.

Wir verwenden in unserer Implementierung eine weitere von Kwa [27] vorgeschlagene Verbesserung dieses Ansatzes: Bei der Relaxation der Kante (v, w) muss w nicht in die Prioritätswarteschlange der entsprechenden Suche eingefügt werden, falls w bereits von der entgegengesetzten Suche endgültig markiert wurde, da in diesem Fall bereits ein kürzester v-t- bzw. v-s-Weg bekannt ist. Auf diese Weise erhalten wir ein zusätzliches Pruning für Vorwärts- und Rückwärtssuche.

Bei der Verwendung des Algorithmus von Dijkstra für Vorwärts- und Rückwärtssuche terminiert das Verfahren aufgrund des Optimalitätsprinzips (siehe Satz 4.3), sobald ein Knoten von beiden Suchrichtungen endgültig markiert wurde. Wird die A^*-Suche benutzt, hängt das Terminierungskriterium jedoch von den verwendeten Schätzfunktionen ab. Wir halten daher folgende Eigenschaft fest:

Definition 4.7 (Symmetrische und antisymmetrische Schätzfunktionen)
Zwei Schätzfunktionen $h_s : V \to \mathbb{Q}$ und $h_t : V \to \mathbb{Q}$ heißen *antisymmetrisch*, falls für alle Knoten $u, v \in V$ gilt:

$$c_{h_t}(u, v) = c_{h_s}(v, u) \quad \text{bzw.} \quad h_s(u) + h_t(u) \equiv \texttt{konstant}$$

Ist die Bedingung nicht erfüllt, sprechen wir von *symmetrischen* Schätzfunktionen.

Falls antisymmetrische Schätzfunktionen zum Einsatz kommen, benutzen Vorwärts- und Rückwärtssuche dieselbe Kostenfunktion. In diesem Fall kann die bidirektionale A*-Suche wie der bidirektionale Algorithmus von Dijkstra beendet werden, sobald ein Knoten in beiden Richtungen endgültig markiert wurde.

Es ist leicht zu sehen, dass die in Abschnitt 4.2.1 vorgestellten Schätzfunktionen symmetrisch sind, d. h. die beiden Suchrichtungen verwenden unterschiedliche Kostenfunktionen. Wenn ein Knoten in beiden Richtungen endgültig markiert wurde, kann in diesem Fall nicht garantiert werden, dass der kürzeste s-t-Weg gefunden wurde [13].

In [13] werden zwei Lösungsansätze für dieses Problem vorgestellt: Beim *symmetrischen Ansatz* wird ein neues Terminierungskriterium für symmetrische Schätzfunktionen entworfen. Der *konsistente Ansatz* konstruiert aus den gegebenen Schätzfunktionen zwei antisymmetrische Funktionen.

4.3.1 Symmetrischer Ansatz

Ein symmetrischer Ansatz hat den Vorteil, dass für beide Richtungen die bestmögliche Schätzfunktion verwendet werden kann. Es wird der oben beschriebene bidirektionale Algorithmus inklusive der Verbesserung von Kwa [27] benutzt.

Nach Pohl [35] kann die Berechnung beendet werden, sobald ein Knoten v mit $\delta_s(v)+h_t(v) \geq \mu$ oder $\delta_t(v)+h_s(v) \geq \mu$ in einer der beiden Suchrichtungen endgültig markiert wird, oder falls beide Prioritätswarteschlangen leer sind. Wir verwenden diesen Ansatz in unserer Implementierung, da eine Verschlechterung der unteren Schranken in der vorliegenden Anwendung von Nachteil wäre.

Abbildung 4.5 visualisiert den Ablauf der bidirektionalen A*-Suche mit linearen Lückenstrafen. Dabei bezeichnen wir die Menge der endgültig markierten Knoten mit P bzw. P^{-1}. Des Weiteren heißt ein Nachfolger *gültig*, falls er sich noch innerhalb des Gitters befindet und die Pruning-Bedingung für ihn nicht erfüllt ist. „Q aktualisieren“ bedeutet, dass entweder eine `INSERT`$(v, \delta_s(v))$- oder eine `DECREASE-KEY`$(v, \delta_s(v))$-Operation auf Q durchgeführt wird.

4.3.2 Konsistenter Ansatz

Beim konsistenten Ansatz wird die Berechnung beendet, sobald ein Knoten von beiden Richtungen endgültig markiert wurde. Dafür müssen jedoch Einschränkungen bei der Wahl der Schätzfunktion in Kauf genommen werden.

Der grundsätzliche Ablauf des Algorithmus ist identisch zum symmetrischen Ansatz. Es werden jedoch antisymmetrische Schätzfunktionen ρ und $-\rho$ bzw. ρ und $-\rho+c$ mit einer beliebigen Konstante $c \in \mathbb{Q}$ verwendet. In der Literatur werden die folgenden einfachen Funktionen vorgeschlagen, um aus symmetrischen Schätzfunktionen h_t und h_s zwei antisymmetrische Schätzfunktionen zu bestimmen:

- Durchschnitt [25]:

$$\begin{aligned} \rho_t(v) &:= \frac{h_t(v) - h_s(v)}{2} \\ \rho_s(v) &:= \frac{h_s(v) - h_t(v)}{2} = -p_t(v) \end{aligned}$$

- Maximum [12]:

$$\rho_t(v) := \max(h_t(v), h_s(t) - h_s(v) + \beta)$$

  Dabei ist β eine von $h_s(t)$ und / oder $h_t(s)$ abhängige Konstante.

4.4 Erweiterung für quasi-affine Lückenstrafen

Wir haben uns bei den Betrachtungen der vorhergehenden Abschnitte auf lineare Lückenstrafen beschränkt. Für diese gilt das in Satz 4.3 vorgestellte Optimalitätsprinzip. Die Korrektheit des Algorithmus von Dijkstra und der A*-Suche basieren

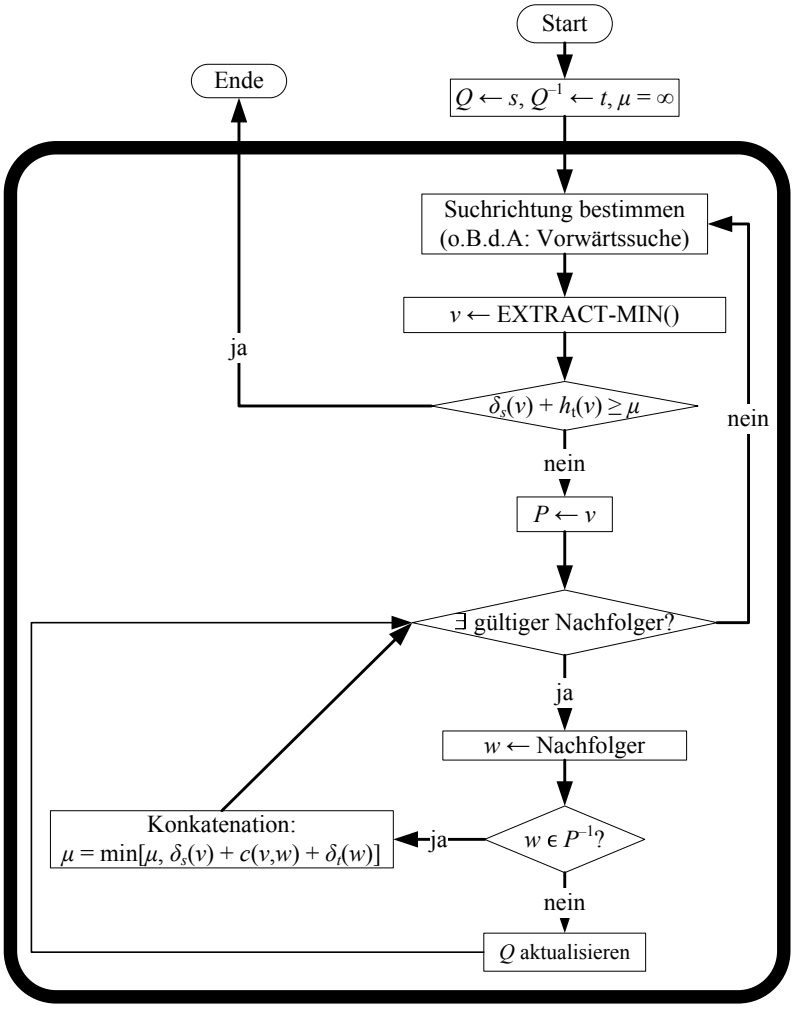


Abbildung 4.5: Ablauf der bidirektionalen A*-Suche bei Verwendung von linearen Lückenstrafen.

auf dieser Eigenschaft. Kürzeste s-u-Wege p_{su} werden hierbei jeweils durch Hinzunahme einer weiteren Kante (u, v) expandiert. Die Länge eines kürzesten s-v-Weges $p_{sv} = s \xrightarrow{*} v$ ergibt sich dann durch:

$$\lambda(p_{sv}) = \min_{(u,v)\in E} \{\lambda(p_{su}) + c(u, v)\}$$

Das Optimalitätsprinzip kann jedoch nicht einfach auf die Verwendung von affinen oder quasi-affinen Lückenstrafen übertragen werden. Hierbei kann es sich lohnen, einen nicht-optimalen Weg von s nach v zu verwenden, wenn dieser bereits eine Lücke aufmacht, die dann durch die Kante (v, w) fortgesetzt wird. Abbildung 4.6 zeigt ein Beispiel für diesen Fall. Der Weg $s \xrightarrow{*} u_2 \rightarrow v$ ist kürzester s-v-Weg. Der kürzeste s-w-Weg ist jedoch $s \xrightarrow{*} u_1 \rightarrow v \rightarrow w$, hat also einen suboptimalen s-v-Weg als Teilweg.

Das zentrale Problem besteht also darin, dass die Kosten für eine Kante nicht mehr feststehen, sondern vom vorhergehenden Weg abhängen. Aus diesem Grund genügt es nicht mehr, nur einen optimalen s-v-Weg zu betrachten. Bei Verwendung

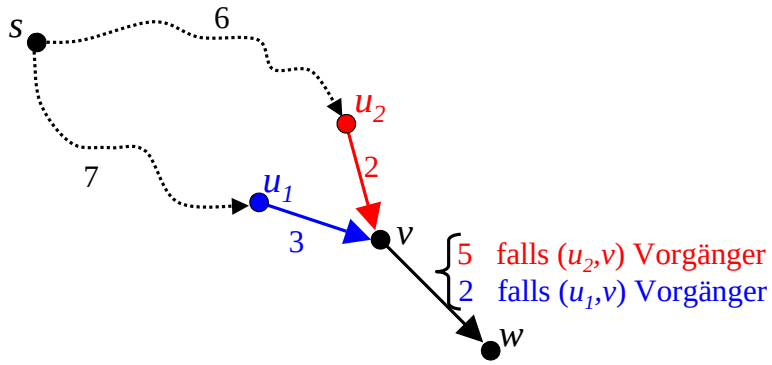


Abbildung 4.6: Gegenbeispiel für das Optimalitätsprinzip bei affinen oder quasi-affinen Lückenstrafen.

von affinen Lückenstrafen müssen sämtliche möglichen s-v-Wege einbezogen werden, da jeder von ihnen potenzieller Teilweg des kürzesten s-t-Weges ist.

Da wir aus den in Abschnitt 3.2 dargestellten Gründen nur quasi-affine Lückenstrafen berücksichtigen, genügt es jedoch, alle Vorgängerkanten der Kante (v, w) zu betrachten. Dies kann beispielsweise dadurch gelöst werden, dass eine Liste von Zeigern auf sämtliche bereits endgültig markierte Vorgängerknoten von v sowie die entsprechenden Distanzlabel gespeichert werden. Wird v endgültig markiert, müssen alle Knoten in dieser Liste bei der Kanten-Relaxation berücksichtigt werden.

Außerdem kann nun nicht mehr garantiert werden, dass jeder Knoten nur einmal aus der Prioritätswarteschlange extrahiert wird. Bei dem Graphen in Abbildung 4.6 kann es z. B. passieren, dass v vor u_1 extrahiert wird. Dann ist es bei der Relaxation der Kante (u_1, v) nötig, den Knoten v erneut in die Prioritätswarteschlange einzufügen, da sonst der Weg $s \xrightarrow{*} u_1 \to v \to w$ nie betrachtet wird.

Diese Art der Implementierung ist jedoch recht umständlich und hat einen hohen Zeit- und Platzbedarf. Daher verwenden wir einen anderen Ansatz [18]. Dafür wird der Algorithmus so modifiziert, dass Kanten anstelle von Knoten betrachtet werden. Dadurch werden nun kürzeste Wege von s zu jeder Kante des Graphen berechnet. Kanten übernehmen also für quasi-affine Lückenstrafen die Rolle der Knoten und jeder Weg wird durch seine Endkante repräsentiert. Auf diese Weise berücksichtigt der Algorithmus alle möglichen Vorgängerkanten der Kante (v, w) auf einem optimalen s-w-Weg.

Um vollständig auf der Menge der Kanten arbeiten zu können, wird zusätzlich ein Dummy-Startknoten s', der nur eine Kante zu s hat, in den Graphen eingefügt (siehe Abbildung 4.7). Es gilt $c(s', s) = 0$. Die Prioritätswarteschlange verwaltet Kanten statt Knoten, wobei jede Kante $e = (v, w)$ mit einem Distanzlabel $\delta_s(e) \in \mathbb{N}$ bzw. $\delta_t(e) \in \mathbb{N}$ in die Prioritätswarteschlange eingefügt wird. Das Distanzlabel entspricht dabei dem Distanzwert des Endknoten w von e. Für jede Kante wird die Vorgängerkante $\pi_s(e) \in E$ bzw. $\pi_t(e) \in E$ gespeichert. Wird eine Kante endgültig markiert, werden alle Nachfolger ihres Endknotens berechnet.

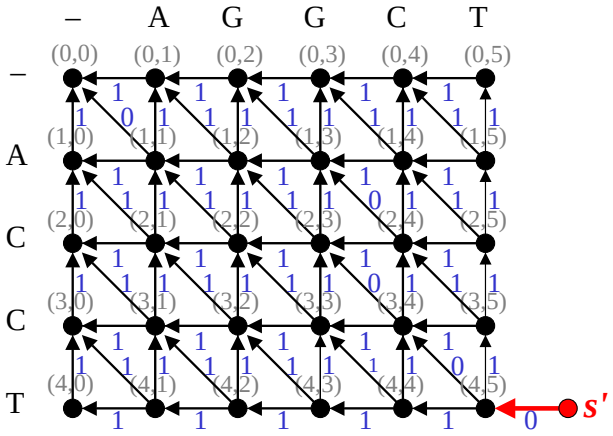


Abbildung 4.7: Edit-Graph für quasi-affine Lückenstrafen mit $n = 2$.

Neben dieser grundlegenden Modifikation müssen bei der Verwendung von quasi-affinen Lückenstrafen noch einige weitere Aspekte beachtet werden, die wir im Folgenden näher erläutern werden.

Berechnung der Kantenkosten

Die Kosten für eine Kante f entsprechen bei quasi-affinen Lückenstrafen einer von f und seiner Vorgängerkante e abhängige Funktion. Sei $e = (u, v)$ und $f = (v, w)$ mit $u = (u_1, \ldots, u_n)$, $v = (v_1, \ldots, v_n)$ und $w = (w_1, \ldots, w_n)$. Die Berechnung der Kantenkosten wird in zwei Teilterme aufgeteilt:

- $c(f)$: Teilkosten für f, an denen keine Lücken beteiligt sind, d. h. die Kosten, die durch Matches und Mismatches erzeugt werden.
- $c_{\text{gap}}(e, f)$: Kosten für f in Abhängigkeit von e, die durch geöffnete und fortgesetzte Lücken entstehen.

Die Kosten von f mit Vorgängerkante e ergeben sich nach [36] als:

$$
\begin{aligned}
c(f \mid e) &= c(f) + c_{\text{gap}}(e, f) \\
c_{\text{gap}}(e, f) &= \sum_{i=0}^{n} \sum_{j=i+1}^{n} \begin{cases} 0 & \text{falls } (w_i = v_i \text{ und } w_j = v_j) \text{ oder} \\ & \quad (w_i = v_i + 1 \text{ und } w_j = v_j + 1) \\ g_{\text{ext}} & \text{falls } (w_i = v_i + 1 = u_i + 2 \text{ und } w_j = v_j = u_j) \text{ oder} \\ & \quad (w_i = v_i = u_i \text{ und } w_j = v_j + 1 = u_j + 2) \\ g_{\text{op}} & \text{sonst} \end{cases}
\end{aligned}
$$

Bei $c_{\text{gap}}(e, f)$ wird also für jedes Sequenzenpaar geprüft, ob f im Vergleich zu e eine Lücke öffnet oder fortsetzt.

Berechnung der Schätzfunktion für die A*-Suche

Bei der A*-Suche muss sichergestellt werden, dass die verwendete Schätzfunktion weiterhin konsistent ist. Die zuvor vorgestellten Schätzfunktionen basieren auf Algorithmen für paarweise und dreifache Alignments und verwenden den Needleman-Wunsch-Algorithmus (siehe Abschnitt 3.1.1). Wir erinnern noch einmal daran, dass affine und quasi-affine Lückenstrafen für paarweise Alignments identisch sind. Daher wäre es nahe liegend, bei quasi-affinen Lückenstrafen und paarweisen Alignments zur Berechnung der unteren Schranke den Algorithmus von Gotoh (siehe Abschnitt 3.1.3) zu verwenden, da dieser affine Lückenstrafen berücksichtigt. Die dadurch entstehenden unteren Schranken erfüllen jedoch nicht mehr die Konsistenzbedingung aus (4.6) und können somit negative Kantengewichte verursachen.

Abbildung 4.8 zeigt ein Beispiel hierzu. Für jeden Knoten wurden untere Schranken mit Hilfe des Algorithmus von Gotoh berechnet. Dabei kam eine PAM-250-Distanzmatrix (siehe Abbildung B.1 in Anhang B) mit $g_{\text{op}} = 8$ und $g_{\text{ext}} = 12$ zum Einsatz. Es gilt im Widerspruch zu (4.6) für die markierten Knoten und Kanten:

$$\underbrace{67}_{h_t(v)} > \underbrace{12}_{c(v,w)} + \underbrace{52}_{h_t(w)} = 64$$

Es genügt also nicht mehr, nur eine einzige untere Schranke für jeden Knoten zu betrachten. Die untere Schranke für die Länge eines kürzesten w-t-Weges ist abhängig davon, welche Lücken von der gerade betrachteten Kante $f = (v, w)$ eingefügt wurden. Bei Verwendung von paarweisen Alignments zur Berechnung der unteren Schranken gibt es hierfür vier verschiedene Fälle:

1. Keine Lücken.
2. Lücke in v und w.
3. Lücke in w, aber nicht in v.
4. Lücke in v, aber nicht in w.

Wir speichern für jeden möglichen Fall eine eigene Tabelle und wählen bei der Berechnung der modifizierten Kantenkosten abhängig von $f = (v, w)$ die passende Tabelle aus. Die Tabellen für die Fälle 2-4 entsprechen dabei den Hilfstabellen A, B und C des Algorithmus von Gotoh (siehe Abschnitt 3.1.3).

Entsprechend gibt es bei dreifachen Alignments neun verschiedene Fälle. Die Berechnung ist jedoch komplizierter. Hierbei könnte eine Modifikation des von Gotoh vorgestellten Algorithmus für drei Sequenzen und affine Lückenstrafen verwendet werden [16].

Alternativ können weiterhin die bisherigen unteren Schranken für lineare Lückenstrafen verwendet werden, da der Wert eines Alignments bzgl. linearer Lückenstrafen eine untere Schranke für den Wert hinsichtlich quasi-affiner Lückenstrafen darstellt. Diese Schranken sind zwar schlechter als die oben beschriebenen, lassen sich jedoch

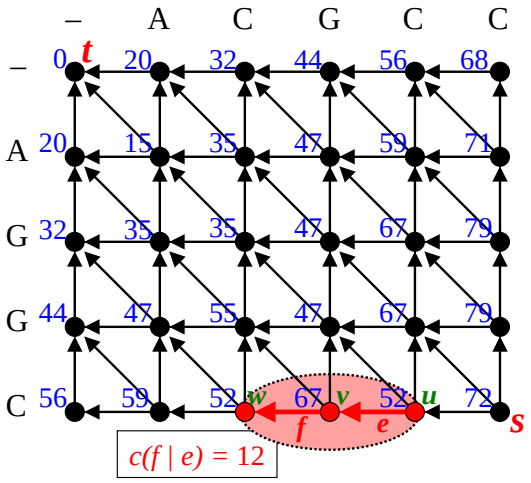


Abbildung 4.8: Beispiel für eine nicht-konsistente Schätzfunktion bei affinen Lückenstrafen.

effizienter berechnen und benötigen zudem weniger Speicherplatz. Wir verzichten daher im Rahmen dieser Arbeit auf die Modifikation der unteren Schranken und verwenden weiterhin die in Abschnitt 4.2.1 beschriebenen Schätzfunktionen h_t und h_t^3 bzw. h_s und h_s^3.

Konkatenation von Vorwärts- und Rückwärtssuche

Bei der Konkatenation von Vorwärts- und Rückwärtssuche der bidirektionalen A*-Suche unterscheiden wir zwei Fälle zum Zeitpunkt der Relaxation der Kante (v, w):

1. (w, v) wurde bereits von der entgegengesetzten Suche endgültig markiert.
2. Die entgegengesetzte Suche hat eine Kante der Form (w, x) mit $x \neq v$ endgültig markiert, nicht aber die Kante (w, v) selbst.

In beiden Fällen werden Vorwärts- und Rückwärtssuche konkateniert und gegebenenfalls μ aktualisiert. Im ersten Fall ist bereits ein kürzester Weg von der Kante (v, w) zur Senke bekannt. Es kann demnach die Verbesserung von Kwa [27] übertragen und auf die Einfügung von (v, w) in die Prioritätswarteschlange verzichtet werden. Für den zweiten Fall ist diese Bedingung jedoch nicht erfüllt und (v, w) wird in die Prioritätswarteschlange aufgenommen.

Abbildung 4.9 visualisiert den Ablauf der bidirektionalen A*-Suche für quasi-affine Lückenstrafen. Dabei bezeichnen P und P^{-1} erneut die Mengen der endgültig markierten Knoten für Vorwärts- und Rückwärtssuche. Ein Nachfolger heißt *gültig*, falls er sich noch innerhalb des Gitters befindet und die Pruning-Bedingung für ihn nicht erfüllt ist. „Q aktualisieren“ bedeutet, dass entweder eine `INSERT`$(v, \delta_s(v))$- oder eine `DECREASE-KEY`$(v, \delta_s(v))$-Operation auf Q durchgeführt wird.

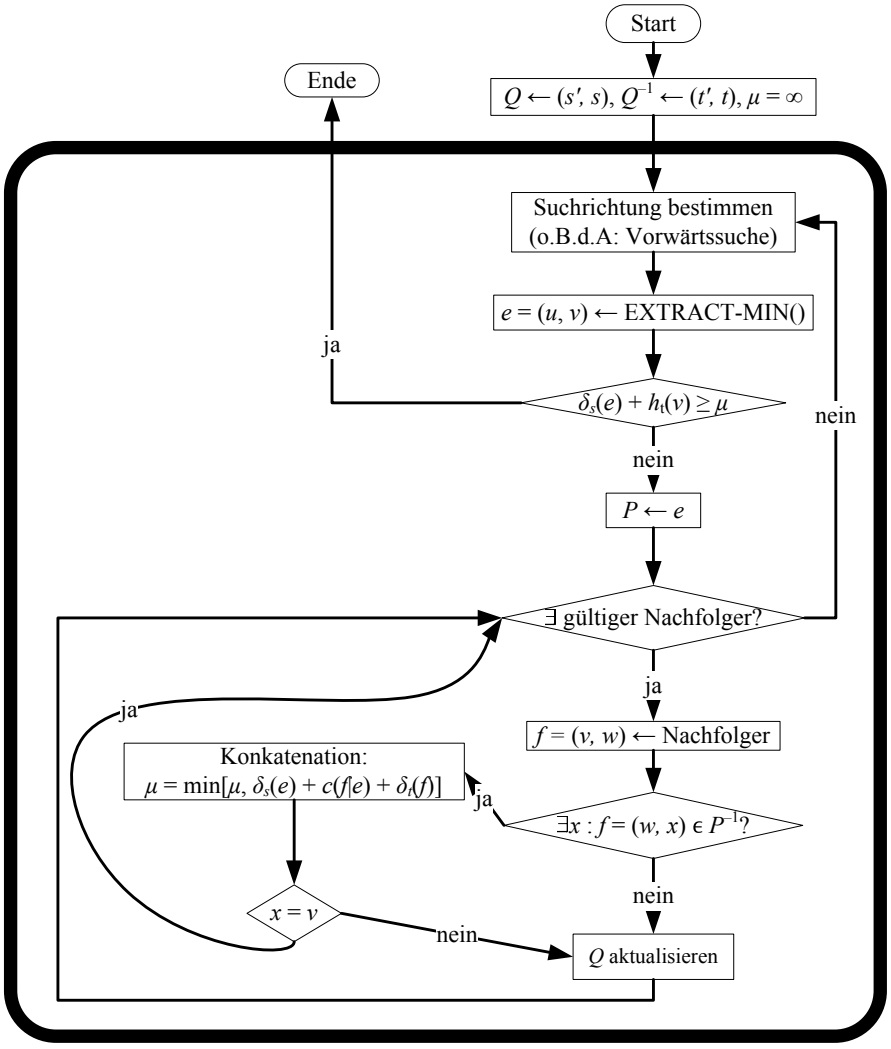


Abbildung 4.9: Ablauf der bidirektionalen A*-Suche bei Verwendung von quasi-affinen Lückenstrafen.

4.5 Zusammenfassung

Wir haben in diesem Kapitel einen graphentheoretischen Ansatz zur Berechnung optimaler multipler Alignments kennen gelernt und verschiedene Kürzeste-Wege-Algorithmen vorgestellt. Mit Hilfe dieser Verfahren kann das Multiple Sequence Alignment im Vergleich zur Dynamischen Programmierung praktisch effizienter gelöst werden. Insbesondere haben wir die bidirektionale A*-Suche behandelt, die bisher noch nicht für das vorliegende Problem verwendet wurde.

Außerdem wurden Probleme, die bei Verwendung von quasi-affinen Lückenstrafen auftreten können aufgezeigt und Lösungsmöglichkeiten entwickelt. Im Folgenden werden wir auf Basis der vorgestellten Methoden einen progressiven Algorithmus entwerfen.

Kapitel 5

Progressive Algorithmen

Die Laufzeit aller bisher vorgestellten Algorithmen wächst exponentiell mit der Anzahl der Sequenzen. Daher werden in der Praxis häufig nur Heuristiken eingesetzt. Bisher ist es nur möglich, für wenige Sequenzen exakte Lösungen zu berechnen.

Einen Kompromiss aus exakten Algorithmen und Heuristiken stellen sog. *progressive Algorithmen* dar. Zentrale Idee dieser Verfahren ist, zunächst nur suboptimale Lösungen zu berechnen, die dann von Iteration zu Iteration weiter verbessert werden und beweisbar gegen das Optimum konvergieren.

Wir werden sehen, dass ein Trade-Off zwischen Quantität der Verbesserung und benötigter Laufzeit vorliegt, da die im Folgenden vorgestellten Algorithmen meistens zu Beginn sehr schnell große Verbesserungen erzielen, der Berechnungsaufwand für weitere Optimierungen jedoch zunehmend größer wird. Außerdem kann es vorkommen, dass der Algorithmus zwar bereits das Optimum berechnet hat, aber noch viel Zeit für den Optimalitätsbeweis benötigt. Da die berechnete Lösung von der verwendeten Modellierung abhängt, kann es zudem passieren, dass eine mathematisch suboptimale Lösung biologisch relevanter ist, als das tatsächliche Optimum bzgl. des verwendeten Modells.

Aus diesen Gründen lohnt es sich häufig nicht, den Algorithmus bis zum Ende laufen zu lassen. Stattdessen erweist es sich als sinnvoll, dem Anwender die Möglichkeit zu geben, die Berechnungen vorzeitig abzubrechen, falls z. B. lange keine Verbesserungen mehr erzielt wurden oder der Benutzer mit der aktuellen Lösung zufrieden ist. Vorgaben bzgl. Laufzeit oder Speicherbedarf können ebenfalls ein Abbruchkriterium sein. Falls der Anwender das Programm vorzeitig abbricht, erhält er dennoch eine zulässige – wenn auch evtl. suboptimale – Lösung.

Ein weiterer Vorteil progressiver Algorithmen ist ihre iterative Arbeitsweise und die Wiederverwendbarkeit von Informationen aus vorhergegangenen Iterationen innerhalb der aktuellen Iteration. Auf diese Weise können die Berechnungen der aktuellen Iteration beschleunigt werden. Ein Beispiel hierfür ist die Verwendung von Zwischenlösungen zur Verstärkung der Pruning-Bedingung (siehe Abschnitt 4.2.2).

Wir beginnen mit der Beschreibung des ersten progressiven Algorithmus für Multiple Sequence Alignment [36]. Im Anschluss daran werden die im Rahmen dieser Arbeit entwickelten progressiven Algorithmen präsentiert.

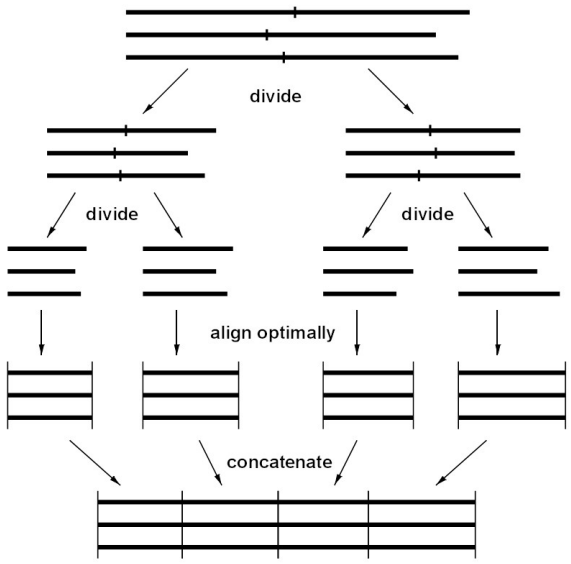


Abbildung 5.1: Schema der Divide-And-Conquer-Methode aus [36].

5.1 Progressiver Divide-And-Conquer-Algorithmus

Reinert et al. haben in [36] den ersten und derzeit schnellsten uns bekannten progressiven Algorithmus für die Berechnung optimaler multipler Alignments, genannt OMA (Abkürzung für „Optimal Multiple Alignment“), vorgestellt. Die zentrale Idee dieses Verfahrens ist, heuristische Divide-And-Conquer-Methoden mit Kürzesten-Wege-Algorithmen zu kombinieren. Die Sequenzen werden rekursiv in Teilsequenzen aufgesplittet, die dann mit Hilfe der unidirektionalen A*-Suche optimal aligniert werden. Durch Konkatenation der berechneten Teillösungen erhält man dann eine heuristische Lösung für das Originalproblem. (siehe Abbildung 5.1)

Die Wahl geeigneter Schnittpositionen während der Divide-Phase ist entscheidend für die Güte der Lösung. Besonders die Wahl einer schlechten Position zu Beginn der Aufteilungen kann die Güte des Ergebnisses negativ beeinflussen. Es existieren effiziente Methoden zur Berechnung guter Schnittpositionen – für Details hierzu sei auf [42] verwiesen.

Außerdem hängt die Qualität der Lösung von der Größe der optimal alignierten Segmente ab. Je größer diese Teilsequenzen sind, desto besser sind die Ergebnisse. Gleichzeitig erhöht sich jedoch die Laufzeit des Algorithmus. Der progressive Algorithmus OMA [36] setzt an dieser Stelle an, um schrittweise bessere Lösungen zu erhalten.

Ein Parameter z bestimmt die maximale Größe der Segmente, die optimal aligniert werden und definiert somit das Rekursionsende der Divide-Phase. Es wird mit einem kleinen Wert z begonnen, um schnell eine erste heuristische Lösung zu erhalten. Der Parameter wird nach jeder Iteration vergrößert. Die berechnete heuristische Gesamtlösung sowie die optimalen Lösungen der Teilsequenzen werden als obere Schranken gespeichert, um die optimale Alignierung längerer Sequenzen mit

Hilfe von Pruning zu beschleunigen. Falls z der Länge der längsten Sequenz entspricht, findet kein Divide-Schritt mehr statt und es wird ein optimales Alignment der Originalsequenzen berechnet. Der Algorithmus konvergiert demnach gegen die optimale Lösung und liefert in jeder Iteration heuristische Zwischenlösungen, die häufig in der Nähe des Optimums liegen.

OMA verwendet quasi-affine Lückenstrafen mit den Parametern $g_{\text{op}} = 8$ und $g_{\text{ext}} = 12$ (siehe Abschnitt 2.2.2). Außerdem wird die PAM-250-Distanzmatrix aus Abbildung B.1 in Anhang B sowie das Sum-Of-Pairs-Maß aus Definition 2.8 benutzt. Die unteren Schranken für die A*-Suche werden mit Hilfe von dreifachen Alignments bestimmt (siehe Abschnitt 4.2.1.2). Die betrachteten Knoten werden in einem *Trie* (siehe Abschnitt 6.2.1.2) verwaltet. Außerdem verwendet OMA den in Abschnitt 6.2.2 beschriebenen *Gray-Code* zur Bestimmung der Nachbarknoten.

5.2 Progressive bidirektionale A*-Suche

Wir stellen zunächst einen sehr einfachen progressiven Algorithmus vor, der später die Basis des bidirektionalen k-Band-Algorithmus bildet. Hierfür betrachten wir die in Abschnitt 4.3.1 vorgestellte bidirektionale A*-Suche mit symmetrischen Schätzfunktionen.

Soweit uns bekannt ist, gibt es bisher keine Untersuchungen zum Verhalten der bidirektionalen A*-Suche auf dem vorliegenden Problem. Wir können sie als progressiven Algorithmus auffassen, da bei jedem aufeinander treffen von Vorwärts- und Rückwärtssuche ein s-t-Weg aus den berechneten Teilwegen zusammengesetzt wird. Dieser Weg stellt eine zulässige, aber nicht notwendigerweise eine optimale Lösung dar. Die bidirektionale A*-Suche konvergiert gegen das Optimum.

Zunächst werden in einem Vorverarbeitungsschritt die unteren Schranken für die A*-Suche berechnet. Wir erinnern an dieser Stelle daran, dass die Schätzfunktionen $h_t : V \to \mathbb{Q}$ und $h_t^3 : V \to \mathbb{Q}$ für die Vorwärtssuche über optimale Alignments von beliebigen Präfixen der Sequenzen definiert sind (siehe (4.8) und (4.11)). Wir berechnen daher mittels Dynamischer Programmierung (siehe Abschnitt 3.1.1) die relevanten paarweisen und / oder dreifachen Rekursionstabellen. Auf Basis dieser Tabellen können später effizient die benötigten unteren Schranken für die Vorwärtssuche ermittelt werden.

Da wir eine bidirektionale Suche mit symmetrischen Schätzfunktionen verwenden, benötigen wir zusätzlich noch untere Schranken für die Rückwärtssuche. Die Schätzfunktionen $h_s : V \to \mathbb{Q}$ und $h_s^3 : V \to \mathbb{Q}$ basieren dabei auf den Suffixen der Sequenzen (siehe Abschnitt 4.3). Da ein Suffix von S_i einem Präfix der inversen Sequenz S_i^{-1} entspricht, können wir einfach die entsprechenden Rekursionstabellen für die inversen Sequenzen $S_1^{-1}, \ldots, S_n^{-1}$ erstellen und daraus später die unteren Schranken für die Rückwärtssuche berechnen.

Außerdem wird das in Abschnitt 4.2.2 beschriebene Pruning verwendet. Sobald Vorwärts- und Rückwärtssuche konkateniert werden, kann der Wert der gefundenen

Lösung als obere Schranke für den Wert des optimalen Alignments eingesetzt werden. Als eine erste triviale obere Schranke kann z. B. der Wert genommen werden, der durch einfaches untereinander schreiben der gegebenen Sequenzen entsteht. Die Verwendung von Zwischenlösungen zur Verstärkung der Pruning-Bedingung stellt einen großen Vorteil progressiver Verfahren dar.

5.3 Progressiver *k*-Band-Algorithmus

Ein Hauptproblem bei der Berechnung optimaler multipler Alignments ist der mit der Anzahl der Sequenzen exponentiell wachsende Suchraum. Aus diesem Grund besteht die Idee des folgenden Algorithmus darin, den Suchraum zunächst auf einen kleinen Bereich einzuschränken und für diesen eine optimale Lösung zu berechnen. Diese Lösung stellt eine obere Schranke für eine Lösung des Gesamtproblems dar. Abhängig von der Teillösung wird der Suchraum dann sukzessive erweitert, bis das aktuelle Ergebnis einer optimalen Lösung für das Gesamtproblem entspricht.

In der Praxis sind gewöhnlich nur Instanzen mit ähnlichen Sequenzen von Bedeutung. Für Sequenzen gleicher Länge mit wenigen Einfügungen und Löschungen befindet sich ein optimales Alignment in der Nähe der Hauptdiagonalen der Rekursionstabelle. Der auf Dynamischer Programmierung basierende *k-Band-Algorithmus* für paarweise Alignments [19] nutzt diese Eigenschaft aus, indem er den Suchraum auf einen Bereich der Breite k – das sog. *k-Band* – um die Hauptdiagonale herum einschränkt.

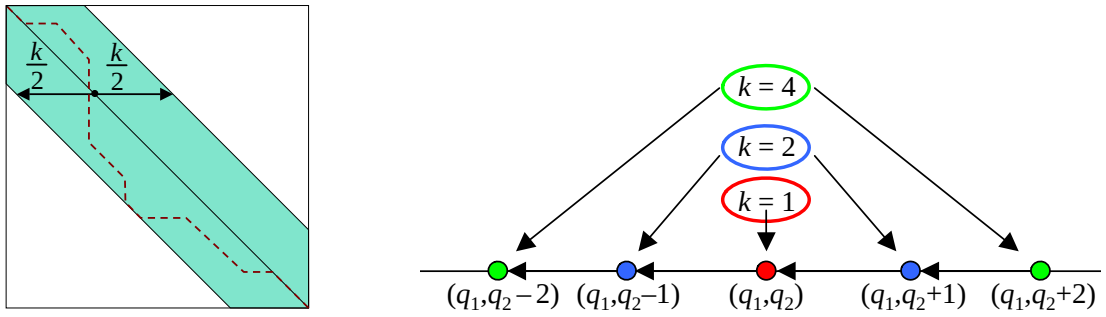
Während einer Iteration werden nur Zellen innerhalb des aktuellen k-Bandes betrachtet. Der Algorithmus startet mit $k = 1$ und berechnet eine Teillösung. Falls diese nicht optimal für die gegebenen Sequenzen ist, wird das k-Band erweitert (z. B. durch Verdoppelung von k) und der Algorithmus erneut gestartet. Dies wird so lange wiederholt, bis ein optimales Alignment gefunden wurde.

Eine zentrale Eigenschaft des k-Band-Algorithmus ist, dass die Breite des aktuell betrachteten k-Bandes einen Hinweis auf die Ähnlichkeit der betrachteten Sequenzen gibt. Aus diesem Grund kann es sinnvoll sein, eine obere Schranke für die Breite des k-Bandes anzugeben, falls man nur an ähnlichen Sequenzen interessiert ist.

Abbildung 5.2(a) visualisiert ein k-Band für zwei Sequenzen. In Abbildung 5.2(b) ist die Schnittgerade durch ein Diagonalelement $q = (q_1, q_2)$ dargestellt. Dabei umfasst ein Band der Breite $k = 1$ nur den rot markierten Knoten (q_1, q_2). Bei Breite $k = 2$ kommen die blauen Knoten $(q_1, q_2 - 1)$ und $(q_1, q_2 + 1)$ hinzu. Ein Band der Breite $k = 4$ enthält sämtliche dargestellte Knoten.

Falls alle Sequenzen dieselbe Länge haben, besteht die Hauptdiagonale aus den Punkten (i, i) mit $i \in \{0, \ell_{\max}\}$, wobei $\ell_{\max}$ die Länge der gegebenen Sequenzen ist. Sind die Sequenzen unterschiedlich lang, berechnen wir die Diagonalelemente $q_i = (q_{i,1}, q_{i,2})$ für $i \in \{0, \ell_{\max}\}$ wie folgt:

$$q_{i,j} = \left\lfloor i \cdot \frac{\ell_j}{\ell_{max}} \right\rfloor \tag{5.1}$$



(a) Das k-Band.

(b) Schnittgerade mit Diagonalelement $q = (q_1, q_2)$.

Abbildung 5.2: 2-dimensionales k-Band.

In jedem Fall gibt es insgesamt $\ell_{\max}$ viele Diagonalelemente.

Wir können den k-Band-Algorithmus als progressive Variante des Needleman-Wunsch-Algorithmus aus Abschnitt 3.1.1 auffassen. Beide Varianten haben dieselbe asymptotische Laufzeit (vgl. Satz 3.1):

Satz 5.1 (Laufzeit des k-Band-Algorithmus für paarweise Alignments)
Der k-Band-Algorithmus berechnet in Zeit $O(\ell^2_{\max})$ ein optimales paarweises Alignment, wenn k nach jedem erfolglosen Durchgang verdoppelt wird.

Beweis: Die Laufzeit für eine Iteration beträgt $O(k \cdot \ell_{\max})$. Spätestens wenn das k-Band den gesamten Suchraum umfasst, erhalten wir eine optimale Lösung. Dies ist der Fall, wenn $k \geq \ell_{\max}$ gilt. Es gibt also höchstens $\log(\ell_{\max})$ viele Iterationen. Daraus ergibt sich insgesamt:

$$\begin{aligned}
\sum_{i=0}^{\log(\ell_{\max})} \left(2^i \cdot \ell_{max}\right) &= \ell_{max} \cdot \sum_{i=0}^{\log(\ell_{\max})} 2^i \\
&= \ell_{\max} \cdot (2^{\log(\ell_{\max})+1} - 1) \\
&= \ell_{\max} \cdot (\ell_{\max} \cdot 2 - 1) \\
&= O(\ell^2_{\max})
\end{aligned}$$

□

Der k-Band-Algorithmus wurde im Rahmen dieser Arbeit auf n Sequenzen erweitert und auf die Berechnung kürzester Wege übertragen. Wir werden im Folgenden darstellen, welche Probleme hierbei auftreten können und Lösungsansätze für diese Probleme entwickeln. Hierzu werden wir zunächst einige grundlegende Modifikationen erläutern und anschließend speziellere Techniken vorstellen. Hierzu gehören verschiedene Distanzmetriken sowie Strategien zur Erweiterung des k-Bandes. Außerdem werden wir zeigen, wie sich durch Neustarts bedingte doppelte Berechnungen minimieren lassen.

5.3.1 Grundlegende Modifikationen für n Sequenzen und Kürzeste-Wege-Algorithmen

Beim k-Band-Algorithmus für kürzeste Wege werden nur noch Knoten betrachtet, die – bzgl. einer gegebenen Distanzmetrik – höchstens die Distanz $\frac{k}{2}$ zur Hauptdiagonalen des Edit-Graphen haben. Die Hauptdiagonale lässt sich dabei auf einem n-dimensionalen Gitter-Graphen analog zu (5.1) berechnen. Wir bezeichnen Knoten mit einer Distanz von genau $\frac{k}{2}$ als *Randknoten*.

Zur Berechnung eines kürzesten Weges kann sowohl die unidirektionale als auch die bidirektionale A*-Suche verwendet werden. Wir beschränken uns im Folgenden auf lineare Lückenstrafen. Die Übertragung auf quasi-affine Lückenstrafen kann, wie in Abschnitt 4.4 beschrieben, durchgeführt werden.

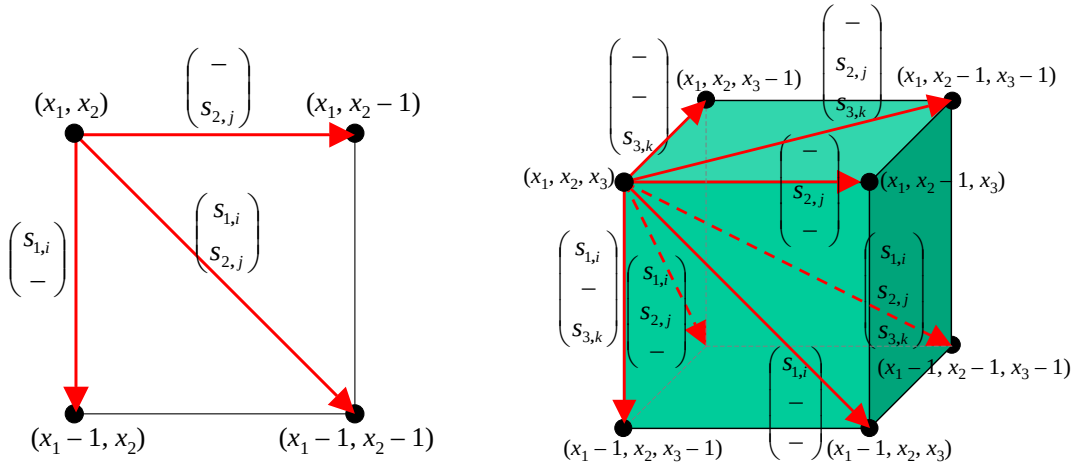
Eine Iteration des k-Band-Algorithmus endet, sobald eine optimale Lösung für den eingeschränkten Suchraum gefunden wurde. Es ist klar, dass die gefundene Teillösung für die betrachteten Sequenzen optimal ist, falls die A*-Suche keinen Randknoten endgültig markiert hat. Ansonsten wird das Band erweitert und die bisher beste Lösung als obere Schranke zur Verbesserung der Pruning-Bedingung verwendet, um so die Berechnungen der nachfolgenden Iteration zu beschleunigen.

Bei der unidirektionalen Suche ist eine gefundene Teillösung optimal, sobald der Zielknoten t extrahiert wird. Wir überprüfen nur bei der Extraktion eines Knotens, ob dieser dem Ziel entspricht, da ein derartiger Test im schlimmsten Fall Zeit von $O(n)$ benötigt und sich somit nicht bei jeder Kanten-Relaxation lohnt. Die optimale Lösung stellt also auch gleichzeitig die erste gefundene zulässige Lösung dar. Bei der bidirektionalen Suche unter Verwendung von symmetrischen Schätzfunktionen erhält man hingegen bei jeder Konkatenation von Vorwärts- und Rückwärtssuche eine neue gültige Lösung (siehe Abschnitt 4.3).

Wir haben in Experimenten (siehe Abschnitt 6.3.1.5) festgestellt, dass meistens eine der ersten Konkatenationen der optimalen Lösung entspricht und danach viele nicht-erfolgreiche Konkatenationen bis zum Optimalitätsbeweis folgen. Dies motiviert einen vorzeitigen Abbruch der bidirektionalen Suche, falls bereits ein Randknoten endgültig markiert wurde und die Erfolgsrate unter einen bestimmten Grenzwert sinkt. Dieser Grenzwert ist u. a. abhängig von der Dimension des Graphen, den Sequenzlängen sowie der aktuellen k-Band-Breite und kann experimentell bestimmt werden. Dabei muss ein Kompromiss aus Anzahl der vorzeitigen Iterationsabbrüche mit suboptimalen Teillösungen und Anzahl unnötiger Berechnungsschritte gefunden werden. Da ein vorzeitiger Abbruch nur durchgeführt wird, falls eine weitere Iteration nötig ist, werden hierdurch Berechnungen verhindert, die keine weitere Verbesserungen bringen und nur dem Optimalitätsbeweis der Teillösung dienen.

5.3.2 Distanzberechnung im n-dimensionalen Edit-Graphen

Während die Distanzberechnung im 2-dimensionalen Fall trivial ist (siehe Abbildung 5.2), nimmt ihre Komplexität mit der Anzahl der Sequenzen zu. Da für jeden betrachteten Punkt der Distanztest durchgeführt werden muss, ist eine effiziente



(a) 2-dimensional (b) 3-dimensional

Abbildung 5.3: Die Gitter-Struktur des Edit-Graphen.

Realisierung dieses Schrittes von entscheidender Bedeutung für das Laufzeitverhalten unseres Algorithmus. Die verwendete Metrik soll dabei den Suchraum möglichst sinnvoll eingrenzen. Wir teilen den Distanztest für einen Punkt $x = (x_1, \ldots, x_n)$ in zwei Schritte auf:

1. Bestimmung der Projektion $q = (q_1, \ldots, q_n)$ von x auf die Hauptdiagonale.
2. Berechnung der Distanz zwischen x und q bzgl. einer gegebenen Distanzmetrik.

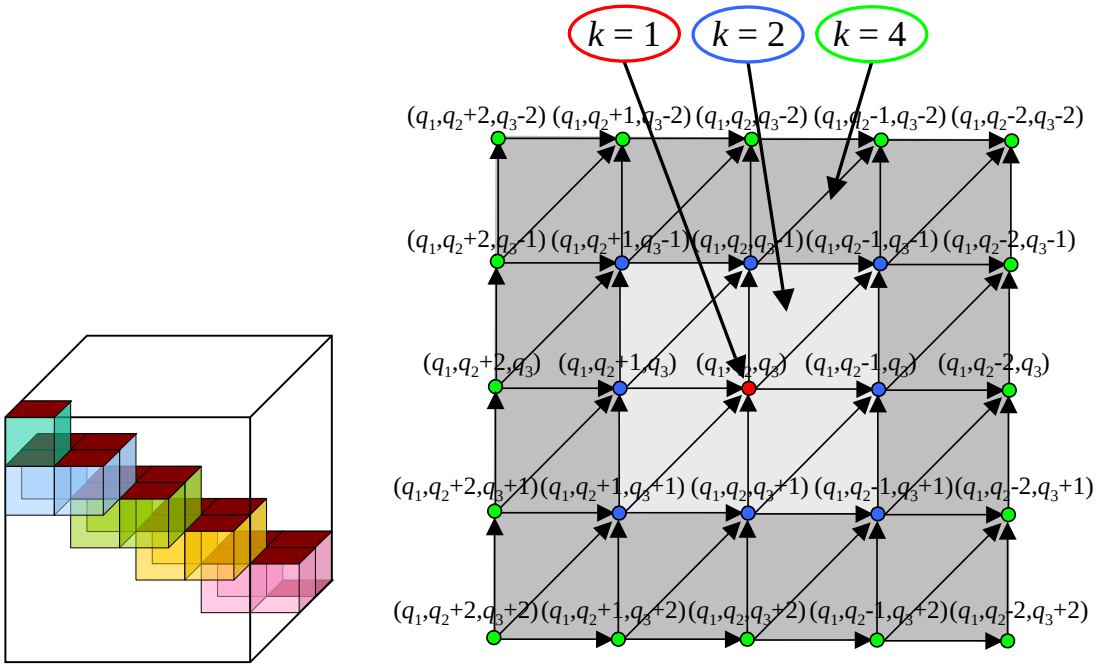
Schritt 1 lässt sich sehr einfach realisieren: Wir nehmen o. B. d. A. an, dass die letzte Sequenz der längsten gegebenen Sequenz entspricht. Damit erhalten wir q als das Diagonalelement mit $x_n = q_n$. Das Distanzproblem wird auf diese Weise um eine Dimension reduziert.

Wir wollen im Folgenden zwei von uns verwendete Metriken vorstellen, mit deren Hilfe Schritt 2 durchgeführt werden kann. Abbildung 5.3 veranschaulicht hierzu noch einmal die Gitter-Struktur eines 2- und 3-dimensionalen Edit-Graphen. Die Metriken werden in Abschnitt 6.3.1.4 experimentell verglichen. Für den 2-dimensionalen Fall liefern beide Metriken dasselbe k-Band.

5.3.2.1 MAX-Metrik

Die MAX-Metrik stellt eine sehr einfache Distanzfunktion dar. Sie ist definiert als das Maximum über die Distanzen in den einzelnen Dimensionen. Das k-Band umfasst alle Punkte, für die folgende Bedingung erfüllt ist:

$$\max_{1 \leq i < n} |x_i - q_i| \leq \frac{k}{2}$$



(a) Das k-Band. (b) Hyperebene mit Diagonalelement $q = (q_1, q_2, q_3)$.

Abbildung 5.4: 3-dimensionales k-Band mit MAX-Metrik.

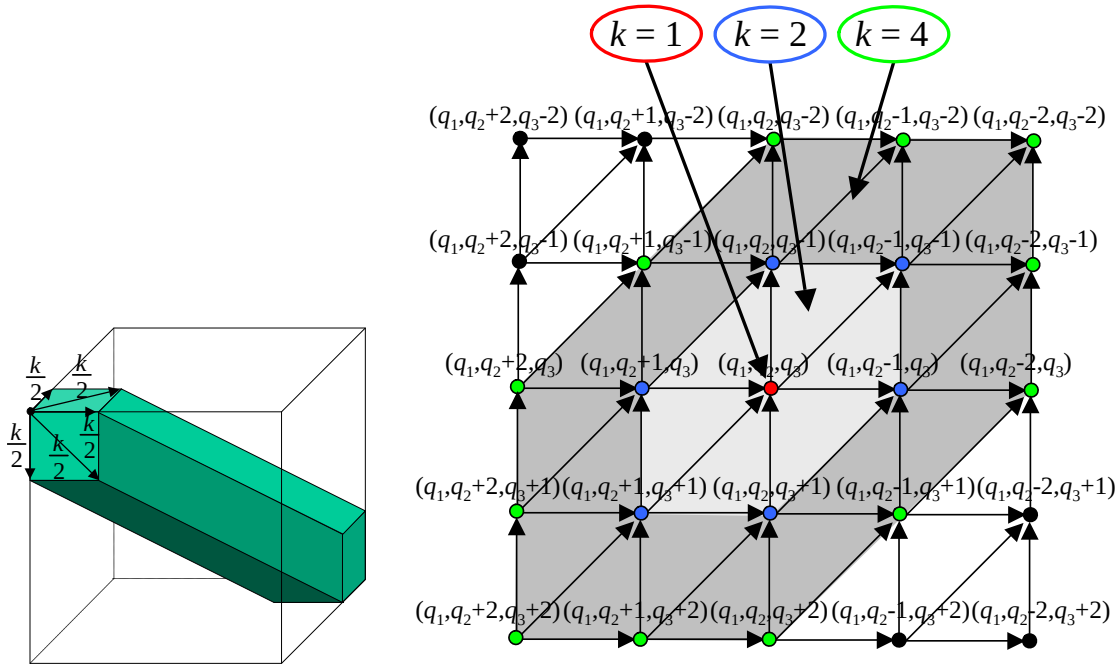
Abbildung 5.4(a) visualisiert ein k-Band für drei Sequenzen gemäß MAX-Metrik. Es entspricht der konvexen Hülle der dunkelrot markierten Hyperebenen und hat für n Sequenzen die Form eines $(2n-2)$-seitigen Prismas.

Zusätzlich ist in Abbildung 5.4(b) die Hyperebene des k-Bandes abgebildet, die das Diagonalelement $q = (q_1, q_2, q_3)$ enthält. Ein Band der Breite $k = 1$ enthält dabei nur das Diagonalelement selbst. Für $k = 2$ umfasst das Band den hellgrau schattierten Bereich. Eine Breite von $k = 4$ entspricht der dunkelgrau schattierten Fläche.

Eine einfache Auswertung der MAX-Metrik benötigt Zeit $O(n)$. Wir werden in Abschnitt 6.2.2 beschreiben, wie mit Hilfe des Gray-Codes [17] eine Reduzierung der Laufzeit auf $O(1)$ erreicht werden kann. Die MAX-Metrik hat also den Vorteil, dass sie sehr effizient berechnet werden kann. Allerdings berücksichtigt sie nicht die Kantenstruktur des Graphen und spiegelt daher nicht die realen Distanzen im Graphen wider.

5.3.2.2 REACH-Metrik

Mit Hilfe der REACH-Metrik wird versucht, die reale Distanz zur Diagonalen genauer zu modellieren. Wir verstehen dabei unter Distanz die Anzahl der Schritte (Kanten), die benötigt werden, um im Graphen von einem Knoten zu einem anderen Knoten zu gelangen. Das k-Band gemäß REACH-Metrik entspricht daher der Menge aller Punkte, die entweder von einem beliebigen Diagonalelement über maximal $\frac{k}{2}$ Kanten erreichbar sind oder von denen aus in $\frac{k}{2}$ Schritten ein beliebiges



(a) Das k-Band. (b) Hyperebene mit Diagonalelement $q = (q_1, q_2, q_3)$.

Abbildung 5.5: 3-dimensionales k-Band mit REACH-Metrik.

Diagonalelement erreicht werden kann. Es hat die Form eines $(2^n - 2)$-seitigen Prismas.

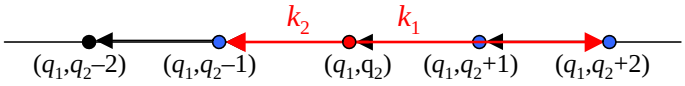
Abbildung 5.5(a) visualisiert ein k-Band mit REACH-Metrik für drei Dimensionen. Wie bei der MAX-Metrik haben wir in Abbildung 5.5(b) die Hyperebene mit Diagonalelement $q = (q_1, q_2, q_3)$ dargestellt. Wiederum enthält ein Band der Breite $k = 1$ nur das Diagonalelement, ein Band der Breite $k = 2$ die hellgraue und ein Band der Breite $k = 4$ die dunkelgraue Fläche.

Wir können nun den k-Band-Test für die REACH-Metrik in Zeit $O(n^2)$ durchführen. Dazu berechnen wir zunächst den Differenzvektor $\Delta = q - x$ und vergleichen danach paarweise die Elemente von Δ. Das k-Band gemäß REACH-Metrik enthält dann alle Punkte für die folgende Bedingung erfüllt ist:

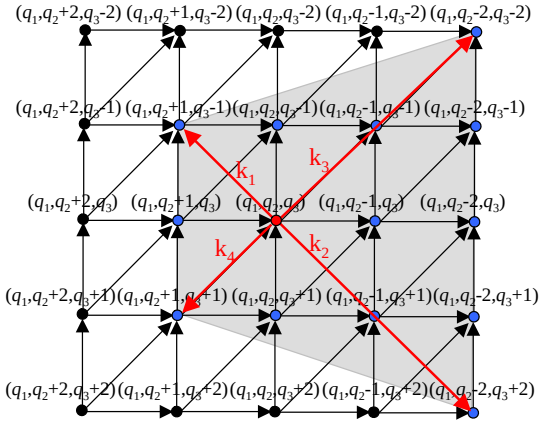
$$\max_{1 \le i < j < n} |\Delta_i - \Delta_j| \le \frac{k}{2}$$

5.3.3 Erweiterung des k-Bandes

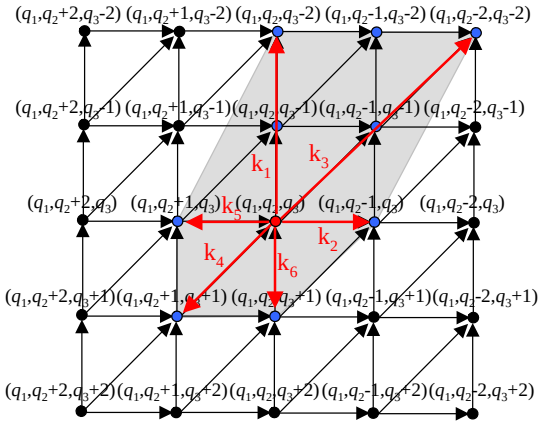
In der einfachen Variante des k-Bandes verwalten wir nur einen einzigen Parameter k und verdoppeln diesen nach jeder Iteration. Dies führt bei höher dimensionalen Eingaben jedoch dazu, dass der Suchraum sehr schnell anwächst und die Einschränkung keine großen Vorteile mehr bringt. Wir haben daher versucht, den Suchraum geschickter zu erweitern. Hierfür wird für jede Seitenfläche des Prismas ein eigener



(a) Schnittgerade für k-Band im 2D.



(b) Hyperebene für MAX-Metrik im 3D.



(c) Hyperebene für REACH-Metrik im 3D.

Abbildung 5.6: Multi-k-Band.

Parameter k_i eingeführt und das entstehende *Multi-k-Band* zunächst in die Richtungen erweitert, die am vielversprechendsten erscheinen.

Prinzipiell kann diese Variante für beide vorgestellten Metriken verwendet werden, wobei wir bei der MAX-Metrik $2n - 2$ und bei der REACH-Metrik $2^n - 2$ Variablen erhalten. Da die Berechnung der REACH-Metrik für $2^n - 2$ Variablen sehr zeitintensiv ist, beschränken wir uns im Folgenden auf eine *Multi-k-Band-Strategie* gemäß MAX-Metrik. Abbildung 5.6 visualisiert das Multi-k-Band für zwei und drei Sequenzen. Dabei gehören jeweils die Punkte innerhalb der grauen Fläche zum k-Band mit den entsprechenden Parametern k_i.

Die Entscheidung, in welche Richtungen das Multi-k-Band erweitert werden soll, ist abhängig von den in der vorhergehenden Iteration markierten Randknoten. Dabei wird angenommen, dass eine Richtung vielversprechend ist, falls die A*-Suche häufig versucht, das Band in diese Richtung zu verlassen. Aus diesem Grund ergibt sich der Erweiterungsfaktor für den Parameter k_i aus dem Anteil der markierten Randknoten, die in der entsprechenden Richtung auf dem Rand des k-Bandes liegen.

5.3.4 Minimierung der Progressivitätskosten

Wie bereits zuvor dargestellt, liegt ein Vorteil des progressiven k-Band-Algorithmus darin, dass auf einem eingeschränkten Suchraum schnell heuristische Lösungen berechnet werden können, die dann durch Verstärkung der Pruning-Bedingung zur

Beschleunigung nachfolgender Iteration eingesetzt werden. Das Neustarten des Algorithmus nach Vergrößerung des k-Bandes führt jedoch zur Mehrfachbetrachtung einiger Knoten. Dies liegt daran, dass sich in der Regel nur Teile eines kürzesten s-t-Weges ändern und somit viele Berechnungen aufeinander folgender Iterationen identisch sind. Aus diesem Grund kann es passieren, dass die Laufzeit trotz besserer Pruning-Bedingungen insgesamt ansteigt. Wir bezeichnen diesen Effekt als *Progressivitätskosten*.

Es ist also wichtig, möglichst viele bereits berechnete Informationen in späteren Iterationen wieder zu verwenden, um die Progressivitätskosten zu minimieren. Hierbei ist zu beachten, dass das mehrfache Extrahieren von Knoten aus der Prioritätswarteschlange nicht vollständig verhindert werden kann, da sich die Distanzlabel $\delta_s(v)$ und $\delta_t(v)$ von Iteration zu Iteration verringern können und damit eine mehrfache Betrachtung des entsprechenden Knotens notwendig ist.

Im Rahmen dieser Arbeit wurden zwei Methoden zur Reduzierung der Progressivitätskosten entwickelt, die wir im Folgenden näher erläutern:

5.3.4.1 Ansatz 1: Snapshot-Methode

Die Snapshot-Methode verfolgt das Ziel, Informationen, die sich in nachfolgenden Iterationen nicht mehr ändern, zu speichern und wieder zu verwenden. Sie nutzt dabei folgende Eigenschaft aus: Sei u der erste Randknoten, der in Iteration i endgültig markiert wird. Dann ist für alle Knoten v, die vor u endgültig markiert wurden, der kürzeste s-v-Weg bekannt. Außerdem wissen wir, dass Iteration $i+1$ bis zu diesem Zeitpunkt die Knoten in derselben Reihenfolge extrahiert, da keine Modifikationen der Kantengewichte und unteren Schranken stattfindet. Es kann höchstens vorkommen, dass in der vorhergehenden Iteration betrachtete Knoten durch eine stärkere Pruning-Bedingung in der nachfolgenden Iteration ausgeschlossen werden.

Diese Eigenschaft ist auf folgende Art und Weise nutzbar: Sobald das erste Mal ein Randknoten endgültig markiert wird, speichern wir eine Momentaufnahme (*Snapshot*) aller Datenstrukturen und setzen die Berechnungen der Iteration normal fort. Die Kopie stellt einen zulässigen Startpunkt für die nachfolgende Iteration dar. Daher beginnen wir die Berechnungen der nächsten Iteration an der gespeicherten Stelle. Auf diese Weise werden unnötige doppelte Berechnungen zu Beginn einer Iteration verhindert. Die Methode benötigt jedoch zusätzliche Zeit zum Kopieren der Datenstrukturen.

5.3.4.2 Ansatz 2: k-Band-Stack-Methode

Im Gegensatz zur Snapshot-Methode verwendet die k-Band-Stack-Methode auch Informationen wieder, die noch nicht als endgültig anzusehen sind. Die Datenstruktur zur Verwaltung bereits markierter Knoten wird dazu vollständig in die nachfolgende Iteration übernommen. Knoten v, für die bereits ein kürzester s-v-Weg bekannt ist, erhalten eine zusätzliche Markierung. Wir bezeichnen diese Knoten als *final*.

Ein Knoten v wird nur dann mehrfach in die Prioritätswarteschlange eingefügt, falls in einer späteren Iteration ein kürzerer s-v-Weg existiert. In diesem Fall benutzt der neue Weg Knoten, die in der vorhergehenden Iteration außerhalb des k-Bandes lagen.

Zusätzlich erfolgt bei der endgültigen Markierung eines Randknotens seine Speicherung auf einem Stack. Zu Beginn der nächsten Iteration werden sämtliche Knoten auf dem Stack in die Prioritätswarteschlange eingefügt. Auf diese Weise kann die Suche beim „besten“ Randknoten fortgesetzt werden.

Bei Verwendung der unidirektionalen A*-Suche ist noch eine weitere Verbesserung möglich: Eine Iteration der unidirektionalen Suche endet mit der Extrahierung des Zielknotens t. Der Schlüssel von t entspricht dann der Länge eines kürzesten s-t-Weges. Da sämtliche in der Prioritätswarteschlange verbleibenden Knoten einen Schlüssel haben, der größer oder gleich dieser Länge ist, können diese zu keiner Lösungsverbesserung führen. Wir entfernen am Ende einer Iteration daher sämtliche Knoten aus der Prioritätswarteschlange und fügen erst danach die gespeicherten Randknoten ein.

Diese Modifikation ist für bidirektionale Suche mit vorzeitigem Iterationsabbruch nicht möglich, da wir die Optimalität der aktuellen Lösung nicht garantieren können. Die Randknoten werden daher nur zusätzlich in die Prioritätswarteschlange eingefügt. Verzichten wir auf den vorzeitigen Iterationsabbruch, ist die Verbesserung auch hier einsetzbar. Dieser Ansatz zeigte in Experimenten jedoch eine Verlangsamung der Berechnungen und wird daher nicht weiter verfolgt.

Die Markierung finaler Knoten sorgt dafür, dass weiterhin das Pruning für Vorwärts- und Rückwärtssuche (siehe Abschnitt 4.3) durchgeführt werden kann. Wir verwenden sämtliche Knoten, die in einer beliebigen Iteration endgültig markiert wurden, für die Konkatenation der beiden Suchrichtungen, d. h. die entsprechenden Datenstrukturen werden aus der vorhergehenden Iteration übernommen. Da jedoch nur für finale Knoten ein kürzester s-v- bzw. t-v-Weg bekannt ist, kann nur bei derartige Knoten auf das Einfügungen in die Prioritätswarteschlange verzichtet werden.

Abbildung 5.7 visualisiert den Ablauf des unidirektionalen k-Band-Algorithmus für lineare Lückenstrafen bei Verwendung der k-Band-Stack-Methode. In Abbildung 5.8 ist der entsprechende Ablauf des bidirektionalen k-Band-Algorithmus (siehe Abbildung 4.5) dargestellt. Dabei bezeichnen wir die Menge der endgültig markierten Knoten mit P bzw. P^{-1}. Des Weiteren heißt ein Nachfolger *gültig*, falls er sich noch innerhalb des Gitters befindet und die Pruning-Bedingung für ihn nicht erfüllt ist. „Q aktualisieren“ bedeutet, dass entweder eine `INSERT`$(v, \delta_s(v))$- oder eine `DECREASE-KEY`$(v, \delta_s(v))$-Operation auf Q durchgeführt wird. Analog ist auch die bidirektionale A*-Suche für affine Lückenstrafen (siehe Abbildung 4.9) verwendbar. Wir verzichten an dieser Stelle jedoch auf die graphische Darstellung.

5.4 Zusammenfassung

Wir haben in diesem Kapitel verschiedene progressive Algorithmen für das Multiple Sequence Alignment kennen gelernt. Diese Algorithmen berechnen zunächst

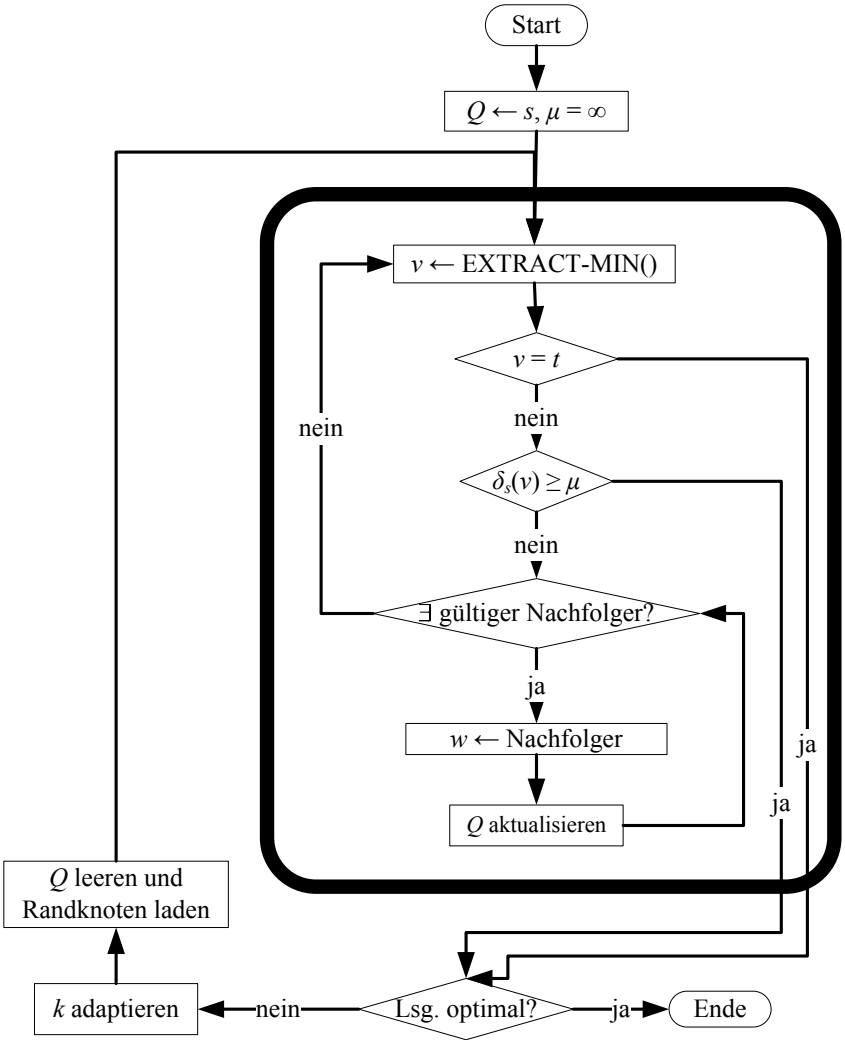


Abbildung 5.7: Ablauf des unidirektionalen k-Band-Algorithmus für lineare Lückenstrafen unter Verwendung der k-Band-Stack-Methode.

suboptimale Lösungen und verbessern diese dann iterativ durch Verwendung von Informationen aus den vorherigen Iterationen. Auf diese Weise hat der Benutzer die Möglichkeit, die Berechnungen mit einem suboptimalen Ergebnis abzubrechen, falls er mit der aktuellen Lösung zufrieden ist oder die zur Verfügung stehenden Ressourcen nicht ausreichen.

Insbesondere wurde in diesem Kapitel der im Rahmen dieser Arbeit entwickelte progressive k-Band-Algorithmus mit verschiedenen möglichen Parametern vorgestellt. Wir wollen diese Parameter im folgenden Kapitel experimentell analysieren und die Ergebnisse mit den anderen progressiven Algorithmen dieses Kapitels vergleichen.

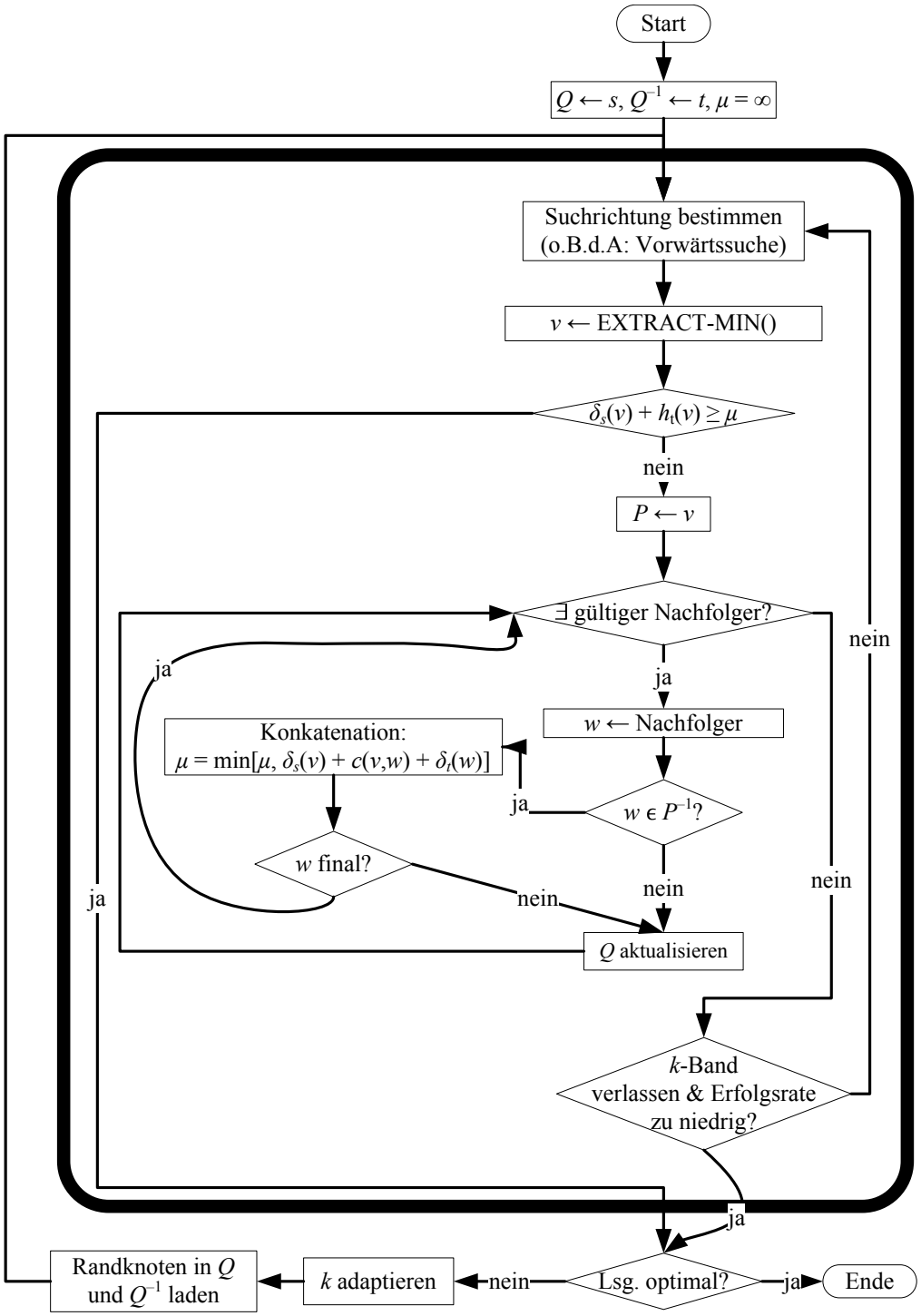


Abbildung 5.8: Ablauf des bidirektionalen k-Band-Algorithmus für lineare Lückenstrafen unter Verwendung der k-Band-Stack-Methode.

Kapitel 6

Experimentelle Analyse

In diesem Kapitel wird abschließend die im Rahmen dieser Arbeit durchgeführte experimentelle Analyse präsentiert. Wir beginnen das Kapitel mit der Vorstellung einiger Implementierungsdetails sowie der Beschreibung verwendeter Datenstrukturen und Testinstanzen. Danach werden verschiedene Parameter des vorgestellten k-Band-Algorithmus untersucht und die Resultate mit denen von OMA [36] verglichen.

6.1 Setup

Die vorgestellten Algorithmen wurden in C++ implementiert. Für den Vergleich mit OMA [36] verwenden wir den von den Autoren veröffentlichten C++-Code. An dieser Stelle sei angemerkt, dass sich die in [36] angegeben Resultate nicht auf die Berechnung einer optimalen Lösung beziehen, da bei den Experimenten zusätzlich die sog. *Face-Bounding-Heuristik* [18, 29] verwendet wurde. Aufgrund der Beobachtung, dass in den letzten Rekursionsleveln meist keine großen Verbesserungen mehr erzielt werden konnten und sich somit weitere Berechnungen nicht lohnten, wurde von den Autoren außerdem eine obere Schranke für den z-Wert, d. h. für die maximale Länge der Teilsequenzen (siehe Abschnitt 5.1), eingeführt. Diese wurde so festgelegt, dass ein Speicherlimit von 2 GB eingehalten werden konnte.

Der Fokus unserer experimentellen Analyse liegt auf dem Optimalitätsaspekt der Algorithmen. Obwohl die Berechnung eines optimalen Alignments gemäß Sum-Of-Pairs-Maß NP-schwer ist [46], kommen exakte Algorithmen z. B. bei der Evaluierung von Heuristiken oder wie bei OMA [36] als Unterprogramm für heuristische Methoden (siehe Abschnitt 5.1) zum Einsatz. Um die Ergebnisse unseres Algorithmus mit denen von OMA vergleichen zu können, verzichten wir in den Testläufen von OMA auf die Verwendung der Face-Bounding-Heuristik und schalten die Beschränkung des z-Wertes aus. Da uns jedoch nur eine 32-Bit-Version von OMA zur Verfügung steht, ist der Speicher bei diesen Testläufen auf 2 GB begrenzt. Für die Analyse des k-Band-Algorithmus gibt es keine derartigen Einschränkungen.

Sämtliche Experimente wurden auf einem AMD Opteron 2.4 GHz durchgeführt. Um die Algorithmen auf praktisch relevanten Daten zu testen, haben wir verschie-

dene Sequenzen aus der BAliBASE (Benchmark Alignment dataBASE, [43]) ausgewählt. Da das Verhalten von Algorithmen zur Berechnung optimaler Alignments u. a. von der Anzahl der Sequenzen, deren Ähnlichkeit und Länge sowie der Anzahl an Einfügungen oder Löschungen abhängt [31], enthält die BAliBASE-Datenbank eine große Bandbreite verschiedener Proteinsequenzen, mit deren Hilfe die Vor- und Nachteile verschiedener Algorithmen analysiert und verglichen werden können.

Sämtliche Sequenzen aus der BAliBASE wurden von Biochemikern ausgewählt, dokumentiert und manuell aligniert. Außerdem wurden sog. *Kernbereiche* identifiziert. Mit Hilfe eines mitgelieferten Bewertungsprogramms namens *bali_score* ist ein Vergleich der Ergebnisse des eigenen Algorithmus mit den (manuell alignierten) Referenzalignments der BAliBASE möglich. Um eine Verfälschung der Evaluierungsergebnisse eines Programms zu verhindern, berücksichtigt *bali_score* nur sog. *Kernbereiche* der Alignments. Diese Bereiche enthalten nur die Sequenzabschnitte, die zuverlässig aligniert werden können. Für Sequenzteile außerhalb dieser Bereiche kann hingegen kein eindeutiges Alignment angegeben werden. Hierbei handelt es sich meist um Bereiche, die keine funktionelle Bedeutung haben und in denen daher häufig Mutationen auftreten können.

Die Referenzsequenzen sind in fünf verschiedene Gruppen unterteilt, die jeweils abhängig von den Längen und der Ähnlichkeit der Sequenzen in weitere Untergruppen aufgeteilt sind. Gruppe 1 enthält sämtliche Sequenzen der BAliBASE. In den Gruppen 2 bis 5 finden sich die Sequenzen in veränderter Form wieder. Mit Hilfe dieser Gruppen ist eine Analyse der Auswirkungen verschiedener Spezialfälle möglich. Im Folgenden werden kurz die Eigenschaften der einzelnen Gruppen erläutert:

Gruppe 1 enthält Instanzen mit bis zu sechs äquidistanten Sequenzen ähnlicher Länge ohne große Blöcke von Einfügungen oder Löschungen. Äquidistant heißt, dass alle Sequenzen evolutionär gleich weit voneinander entfernt sind, die prozentuale Identität aller Sequenzpaare also in einem bestimmten Bereich liegt. Es werden kurze (bis ca. 150 Zeichen), mittlere (bis zu 300 Zeichen) und lange (mehr als 300 Zeichen) Sequenzen unterschieden. Des Weiteren sind die Instanzen nach der durchschnittlichen Identität ($< 25\%$, $20 - 40\%$ und $> 35\%$) ihrer Sequenzen sortiert.

Gruppe 2 enthält Familien mit mindestens 15 sehr ähnlichen Sequenzen, die durch bis zu drei „Waisen“ erweitert werden. Die Identität der Waisen untereinander und zu den Sequenzen aus der Familie darf höchstens 25% betragen. Hiermit werden Auswirkungen einzelner entfernt verwandter Sequenzen auf die Güte eines berechneten Alignments untersucht.

Gruppe 3 enthält Instanzen, die in bis zu vier Untergruppen aufgeteilt sind, wobei Sequenzen aus unterschiedlichen Gruppen weniger als 25% und Sequenzen innerhalb der Gruppen mehr als 25% durchschnittliche Identität besitzen. Es wird untersucht, ob ein Programm in der Lage ist, die Strukturen innerhalb der Familien zu erkennen, obwohl die Sequenzen insgesamt sehr unterschiedlich sind.

(a) BAliBASE 1.0

Set	Instanz	n	Länge	⌀ Id. in %
1.1	1ubi	4	76–94	18
1.1	1wit	5	89–106	17
1.1	3cyr	4	95–109	31
1.1	1pfc	5	108–117	28
1.1	1fmb	4	98–104	49
1.1	1fkj	5	98–110	44
1.2	3grs	4	201–237	14
1.2	1sbp	5	224–263	19
1.2	1ad2	4	203–213	30
1.2	2cba	5	237–259	26
1.2	1zin	4	206–216	42
1.2	1amk	5	242–254	49
1.3	2myr	4	340–474	16
1.3	1pamA	5	435–572	18
1.3	1ac5	4	421–483	29
1.3	2ack	5	452–482	28
1.3	1ad3	4	424–447	47
1.3	1rthA	5	526–541	42

(b) BAliBASE 3.0

Set	Instanz	n	Länge	⌀ Id. in %
1.1	BB11001	4	83–91	< 20
1.1	BB11013	5	51–101	< 20
1.1	BB11025	4	64–103	< 20
1.1	BB11029	4	81–138	< 20
1.1	BB11022	4	63–205	< 20
1.1	BB11021	4	102–139	< 20
1.1	BB11015	4	297–327	< 20
1.1	BB11012	4	320–397	< 20
1.2	BB12021	5	72–86	20–40
1.2	BB12020	4	118–129	20–40
1.2	BB12040	5	113–153	20–40
1.2	BB12025	4	173–215	20–40
1.2	BB12006	4	220–243	20–40

Tabelle 6.1: Überblick über die Eigenschaften der verwendeten Testinstanzen.

Gruppe 4 enthält Instanzen mit bis zu 20 Sequenzen mit sog. *N/C-Terminal Extensionen*, d. h. bei einigen wenigen Sequenzen kommen große Erweiterungen am Anfang oder am Ende der Sequenzen vor. Die Sequenzen bestehen aus bis zu 400 Zeichen, wobei die Längen sehr unterschiedlich sein können.

Gruppe 5 ist ähnlich zu Gruppe 4, jedoch treten die Einfügungen nicht nur an den Enden der Sequenzen auf, sondern sind über die gesamten Sequenzen verteilt. Die Sequenzen haben Längen von bis zu 100 Zeichen.

Da der Fokus unserer Analyse auf der Berechnung optimaler Alignments liegt, müssen wir uns auf Instanzen mit nur wenigen Sequenzen beschränken und daher auf die Betrachtung der Gruppen 2 – 5 verzichten. Es ist jedoch zu erwarten, dass unser progressiver k-Band-Ansatz für Instanzen aus Gruppe 4 und 5 kein gutes Laufzeitverhalten zeigt, da aufgrund der k-Band-Einschränkung lange Lücken erst zum Ende der Berechnungen erlaubt sind.

Aus der Gruppe 1 haben wir Instanzen mit bis zu fünf Sequenzen ausgewählt. Dabei kommen einerseits die in der experimentellen Analyse von Reinert et al. [36] verwendeten Sequenzen aus der älteren BAliBASE 1.0 („BB1“) zum Einsatz. Andererseits haben wir zusätzliche Instanzen aus der neueren BAliBASE 3.0 („BB3“) verwendet. Tabelle 6.1 gibt einen Überblick über die Eigenschaften der ausgewählten Instanzen. Dabei ist jeweils die Anzahl der Sequenzen, deren Länge sowie die durchschnittliche Identität vermerkt.

Zusätzlich zu den Instanzen aus der BAliBASE werden wir – analog zu [36] – die Grenzen der in dieser Arbeit vorgestellten Algorithmen bzgl. der Anzahl der

Sequenzen untersuchen. Hierzu wird eine Familie sehr ähnlicher Sequenzen (Cytochrome C, [1]) ausgewählt. Diese Sequenzen haben eine Länge von 100 – 110 Zeichen und unterscheiden sich nur an wenigen Stellen. Die Bestimmung eines optimalen Alignment stellt daher keine große Herausforderung dar.

6.2 Verwendete Datenstrukturen

Aufgrund der Größe des zugrunde liegenden Edit-Graphen ist eine explizite Graphdarstellung nur mit sehr hohem Aufwand realisierbar. Daher verwenden wir eine implizite Graphdarstellung, d. h. wir erzeugen nur Knoten, die während der Kanten-Relaxation markiert werden. Aufgrund der einfachen Gitterstruktur des Edit-Graphen ist eine effiziente Bestimmung von zueinander adjazenten Knoten möglich. Details hierzu werden in Abschnitt 6.2.2 näher erläutert.

Des Weiteren werden Datenstrukturen benötigt, um effizient nach bereits erzeugten Knoten und Kanten zu suchen. Diese werden auch für die Verwaltung von bereits endgültig markierten Knoten für die bidirektionale Suche verwendet. Abschnitt 6.2.1 beschreibt derartige Datenstrukturen.

Für die Realisierung der Prioritätswarteschlange wird ein einfacher Binärheap verwendet. Abhängig von der verwendeten Lückenstrafe verwaltet dieser Knoten (lineare Kosten) bzw. Kanten (quasi-affine Kosten). Ein Binärheap hat den Vorteil, dass er leicht zu implementieren ist und in der Praxis im Allgemeinen gute Resultate liefert. Dennoch könnte die Verwendung einer fortgeschritteneren Datenstruktur die Laufzeit der vorgestellten Algorithmen weiter verbessern. Wir haben im Rahmen dieser Arbeit darauf verzichtet, da der Schwerpunkt der Untersuchungen auf der Entwicklung des progressiven k-Band-Algorithmus liegen sollte.

6.2.1 Implizite Graphdarstellung

Zur Verwaltung der markierten Knoten wurden im Rahmen dieser Arbeit zwei verschiedene Datenstrukturen verwendet. Hierbei handelt es sich einerseits um eine Listendarstellung, die einfach zu implementieren ist und zudem wenig zusätzlichen Speicher benötigt, dabei jedoch insbesondere bei vielen betrachteten Knoten und / oder vielen Sequenzen sehr langsam sein kann. Daher verwenden wir andererseits eine Baumdatenstruktur, die in den meisten gängigen Verfahren zum Einsatz kommt. Wir werden die beiden Datenstrukturen im Folgenden kurz vorstellen und dann in Abschnitt 6.3.1.2 Laufzeit und Speicherbedarf für beide Datenstrukturen experimentell vergleichen. Außerdem geben wir Erweiterungen für die Verwaltung von Kanten an.

6.2.1.1 Listendarstellung

Eine erste Idee zur Verwaltung der markierten Knoten berücksichtigt die Eigenschaften des progressiven k-Band-Algorithmus. Da das k-Band den Suchraum um

die Diagonale herum einschränkt, besteht insbesondere in den ersten Iterationen die Hoffnung, dass jeweils nur eine geringe Anzahl an Knoten um ein Diagonalelement herum erzeugt wird. Diese Eigenschaft motiviert die Verwendung einer Listendarstellung für spärlich besetzte Matrizen [49].

Wir erinnern an dieser Stelle daran, dass o. B. d. A. angenommen wird, dass die Sequenz S_n der längsten Sequenz entspricht. Wir erzeugen daher ein Feld der Größe $\ell_{\max}$ von Vektoren und ordnen jedem Vektor ein Diagonalelement $q = (q_1, \ldots, q_n)$ zu. Der Vektor zum Element q enthält dann diejenigen Knoten $x = (x_1, \ldots, x_n)$ mit $x_n = q_n$. Durch Verwendung eines Feldes kann der zu einem Punkt gehörige Vektor in konstanter Zeit ermittelt werden. Außerdem gehen wir davon aus, dass der Zugriff auf ein beliebiges Element des Vektors in konstanter Zeit möglich ist. Eine Einfügeoperation kann jedoch im schlimmsten Fall linear in der Anzahl der gespeicherten Elemente sein. Abbildung 6.1(a) zeigt ein Beispiel für diese Datenstruktur. Sie wird im Folgenden als *Listendarstellung* bezeichnet.

Satz 6.1 Der Test, ob ein Knoten bereits markiert wurde, benötigt bei Verwendung der Listendarstellung Zeit $O(n^2 \cdot \log(\ell_{\max}))$.

Beweis: Der zugehörige Vektor kann in konstanter Zeit ermittelt werden. Danach wird mit binärer Suche nach dem gegebenen Element gesucht.

Jedes x_i kann $\ell_i + 1 \leq \ell_{\max} + 1$ verschiedene Werte annehmen. Da zudem x_n fest ist, hat jeder Vektor $O((\ell_{max} + 1)^{n-1})$ Einträge. Der Vergleich des gesuchten Elementes mit einem Element in der Datenstruktur kostet Zeit $O(n)$. Die binäre Suche benötigt folglich insgesamt Zeit

$$O(n \cdot \log((\ell_{\max} + 1)^{n-1})) = O(n^2 \cdot \log(\ell_{\max} + 1)) = O(n^2 \cdot \log(\ell_{\max})),$$

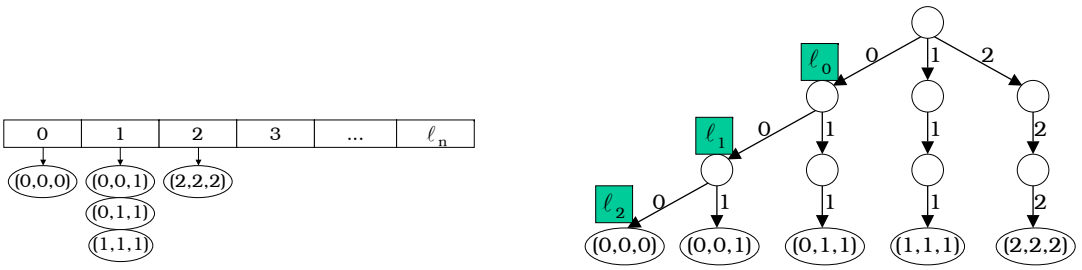
da $\log(\ell_{\max} + 1) \leq \log(2 \cdot \ell_{\max}) = \log(\ell_{\max}) + \log(2)$ für $\ell_{\max} \geq 1$. □

Die tatsächliche Laufzeit hängt hier stark davon ab, wie die markierten Knoten um die Diagonale herum verteilt sind. Die Zeit, um einen Knoten in die Datenstruktur einzufügen, setzt sich aus dem Existenztest und dem Einfügen selbst zusammen. Dabei benötigt das Einfügen im schlimmsten Fall Zeit $O((\ell_{max} + 1)^{n-1})$.

Korollar 6.2 Das Einfügen eines neuen Knoten benötigt bei Verwendung der Listendarstellung Zeit $O(n^2 \cdot \log(\ell_{\max}) + (\ell_{max} + 1)^{n-1}) = O((\ell_{max} + 1)^{n-1})$.

6.2.1.2 Trie

Gängige Verfahren [18, 29, 36] verwenden zur Verwaltung der markierten Knoten einen sog. *Trie* (auch: *Präfixbaum* oder *alphabetischer Suchbaum*). Ein Trie ist ein Suchbaum, mit dessen Hilfe Zeichenketten über einem Alphabet Σ gespeichert werden können. Die Kanten des Baumes sind dabei mit den Zeichen aus Σ beschriftet. Jeder Knoten hat maximal $|\Sigma|$ ausgehende Kanten, wobei alle Kanten unterschiedlich beschriftet sind. Jeder Knoten v des Tries repräsentiert dann die Zeichenkette, die durch die Konkatenation der Kantenbeschriftungen von der Wurzel zu v entsteht.



(a) Listendarstellung. (b) Trie.

Abbildung 6.1: Datenstrukturen für die Verwaltung markierter Knoten.

In der vorliegenden Anwendung wird der Trie auf den Koordinaten der markierten Knoten definiert. Alle Blätter des Baumes haben demnach dieselbe Tiefe n und zeigen auf den zugehörigen Knoten. Jede Ebene repräsentiert eine Dimension des Graphen. Abbildung 6.1(b) zeigt ein Beispiel für drei Sequenzen.

Wichtiger Vorteil eines Tries ist, dass mit seiner Hilfe effizient überprüft werden kann, ob ein Knoten bereits markiert wurde oder nicht. Allerdings wird für den Baum selbst zusätzlicher Speicher benötigt.

Satz 6.3 Der Test, ob ein Knoten bereits markiert wurde, benötigt bei Verwendung eines Tries Zeit $O(n \cdot \log(\ell_{\max}))$.

Beweis: In jeder Ebene des Graphen kann mit binärer Suche auf den vorhandenen Koordinaten überprüft werden, ob der entsprechende Wert bereits existiert oder nicht. Ein Knoten auf Ebene i hat maximal $\ell_i + 1 \leq \ell_{\max} + 1$ ausgehende Kanten. Im schlechtesten Fall ist dieser Test auf jeder der n Ebenen nötig.

Damit ergibt sich mit $\log(\ell_{\max}+1) \leq \log(2 \cdot \ell_{\max}) = \log(\ell_{\max})+\log(2)$ für $\ell_{\max} \geq 1$ eine Gesamtlaufzeit von $O(n \cdot \log(\ell_{\max} + 1)) = O(n \cdot \log(\ell_{\max}))$. □

In der Praxis geht der Test allerdings viel schneller, da nur ein geringer Teil des Graphen erzeugt wird und somit deutlich weniger als ℓ_{max} ausgehende Kanten existieren. Außerdem kann bei einem einfachen Existenztest abgebrochen werden, sobald die gesuchte Koordinate auf einer Ebene nicht vorhanden ist.

Satz 6.4 Das Einfügen eines neuen Knoten benötigt bei Verwendung eines Tries Zeit $O(\ell_{max} + n \cdot \log(\ell_{\max}))$.

Beweis: In jeder Ebene des Graphen wird zunächst ein Existenztest in Zeit $O(n \cdot \log(\ell_{\max}))$ durchgeführt. Falls die entsprechende Koordinate existiert, wird mit der nachfolgenden Ebene analog fortgefahren.

Ist die Koordinate hingegen nicht vorhanden, wird sie in den entsprechenden Knoten des Tries eingefügt. Hierfür wird Zeit $O(\ell_{\max})$ benötigt, da das Einfügen im

schlimmsten Fall linear in der Anzahl der enthaltenen Elemente ist. In den nachfolgenden Ebenen kostet das Einfügen dann nur noch konstante Zeit, da die entsprechenden Knoten noch nicht existieren. Damit ergibt sich insgesamt eine Laufzeit von $O(\ell_{max} + n \cdot \log(\ell_{\max}))$. □

6.2.1.3 Erweiterung für quasi-affine Lückenstrafen

Bei der Verwendung von quasi-affinen Lückenstrafen wird neben einer Datenstruktur für die Knoten auch noch eine Verwaltung für die betrachteten Kanten benötigt. Wir orientieren uns hierbei an der Vorgehensweise von Gupta et al. [18] und speichern für jeden Knoten u eine Liste der von u ausgehenden Kanten (u, v).

Alternativ könnte jeweils eine Kantenliste im Endknoten gespeichert werden. Diese Darstellung hat jedoch einen entscheidenden Nachteil: Sobald die Kante (u, v) aus der Prioritätswarteschlange extrahiert wird, müssen alle von v ausgehenden Kanten (v, w) betrachtet werden. Da jedoch alle Endknoten w verschieden sind und somit unterschiedliche Listen für die Speicherung benötigt werden, ist bei dieser Methode für jede der 2^n-1 ausgehenden Kanten eine Suche in der Knotendatenstruktur notwendig. Bei einer Anordnung am Startknoten genügt hingegen eine einzige Suche nach v. Danach wird lediglich die bei v gespeicherte Kantenliste betrachtet und modifiziert, da alle von v ausgehenden Kanten in dieser Liste verwaltet werden.

6.2.2 Berechnung der Nachbarknoten und Kantengewichte

Für jeden Knoten x bzw. jede Kante $e = (u, v)$, die aus der Prioritätswarteschlange extrahiert wird, müssen $2^n - 1$ Nachfolgeknoten $r_i = (r_{i,1}, \ldots, r_{i,n})$ bzw. Nachfolgekanten (v, r_i) bestimmt werden. Zusätzlich ist für jeden Nachfolger ein Test nötig, ob er sich innerhalb des Gitters und ggf. innerhalb des k-Bandes befindet. Falls diese Tests erfolgreich sind, müssen zudem die Kantenkosten von (x, r_i) bzw. (v, r_i) ermittelt werden. Eine effiziente Realisierung dieser Schritte ist sehr wichtig für das Laufzeitverhalten der betrachteten Algorithmen.

Die triviale Aufzählung aller Nachbarn benötigt Zeit $O(n)$ pro Nachfolger. Wir können diese Zeit deutlich reduzieren, wenn wir anstelle der herkömmlichen Aufzählung in lexikographischer Ordnung den sog. *Gray-Code* [17] verwenden. Beim Gray-Code werden Binärzahlen aus $\{0, 1\}^n - \{(0, \ldots, 0)\}$ so aufgezählt, dass sich aufeinander folgende Zahlen in genau einem Bit unterscheiden (siehe Abbildung 6.2).

Die folgende Darstellung der Vorgehensweise orientiert sich an [29]. Daher gehen wir im Folgenden davon aus, dass die Kanten wie im inversen Edit-Graphen G^{-1} von $t = (0, \ldots, 0)$ nach $s = (\ell_1, \ldots, \ell_n)$ verlaufen. Ein Übertragung auf die Vorwärtssuche ist mit wenigen Modifikationen möglich. Des Weiteren betrachten wir den Fall der linearen Lückenstrafen, d. h. r_i ist ein Knoten.

Jeder Nachfolger r_i von x wird durch den zugehörigen Differenzvektor $r_i - x \in \{0, 1\}^n - \{0, \ldots, 0\}$ repräsentiert. Des Weiteren bezeichne $\ldots i_{[2]} i_{[1]} i_{[0]}$ die

1	0	1
2	1	1
3	1	0

(a) $n = 2$

1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

(b) $n = 3$

1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	1	1
6	0	1	0	1
7	0	1	0	0
8	1	0	0	0

9	1	0	0	1
10	1	0	1	1
11	1	0	1	0
12	1	1	1	0
13	1	1	1	1
14	1	1	0	1
15	1	1	0	0

(c) $n = 4$

Abbildung 6.2: Einige Beispiele für den Gray-Code.

Binärdarstellung der Zahl i. Wir definieren zwei Hilfsvektoren b und d, die in einem Vorverarbeitungsschritt berechnet werden:

$$b[i] = \begin{cases} 0 & \text{falls } i = 0 \\ \min\limits_{i_{[j]}=1} j & \text{falls } 1 \leq i < 2^n \end{cases}$$

$$d[i] = \begin{cases} 0 & \text{falls } i = 0 \\ d[i-1] \text{ XOR } 2^{b[i]-1} & \text{falls } 1 \leq i < 2^n \end{cases}$$

Der Eintrag $b[i]$ gibt für $i = 1, \ldots, 2^n - 1$ an, welches Bit sich von der $(i-1)$-ten zur i-ten Zahl im Gray-Code ändert. Da auf diese Weise die nächste zu verändernde Position bekannt ist, kann Nachfolger r_{i+1} aus Nachfolger r_i in konstanter Zeit bestimmt werden. In der Berechnungsvorschrift für $d[i]$ wird $d[i-1]$ übernommen und genau das in $b[i]$ angegebene Bit invertiert. Deshalb enthält d die Aufzählung der Differenzvektoren in Gray-Code-Reihenfolge. Abbildung 6.3 verdeutlicht die Bestimmung von d an einem Beispiel mit $n = 3$.

Damit ergibt sich der i-te Nachbarn r_i von $x = (x_1, \ldots, x_n)$ durch folgende Berechnung:

$$r_i = x + d_i = (x_1 + d[i]_{[n-1]}, \ldots, x_n + d[i]_{[0]})$$

Mit Hilfe der vorgestellten Technik ist außerdem eine Beschleunigung des Tests auf Lage eines Knotens innerhalb des Gitters möglich. Der triviale Test kostet pro Nachfolger Zeit $O(n)$. Wenn wir jedoch vor der Erzeugung eines Nachfolgers für x jeweils die x_i markieren, die nicht mehr inkrementiert werden dürfen, kommen wir mit Zeit $O(n)$ für alle Nachfolger zusammen aus. Diese Information wird in einem Bitvektor $a \in \{0,1\}^n$ mit

$$a_{[i]} = \begin{cases} 0 & \text{falls } x_i \text{ inkrementiert werden darf} \\ 1 & \text{falls } x_i \text{ nicht mehr inkrementiert werden darf} \end{cases}$$

gespeichert. Der Test entspricht dann für jedes r_i einer bitweisen UND-Verknüpfung von $d[i]$ und a. Falls ($d[i]$ UND a) $= 1$ gilt, befindet sich der Knoten außerhalb des Gitters und kann ignoriert werden.

i	i binär	$b[i]$	$b[i]$ binär	$d[i]$ binär	$d[i]$
0	000	0	000	000	0
1	001	1	001	001	1
2	010	2	010	011	3
3	011	1	001	010	2
4	100	3	100	110	6
5	101	1	001	111	7
6	110	2	010	101	5
7	111	1	001	100	4

Abbildung 6.3: Beispiel für die Berechnung des Gray-Codes für $n = 3$.

Bei Verwendung der MAX-Metrik (siehe Abschnitt 5.3.2.1) kann der Test, ob sich r_i noch innerhalb des aktuellen k-Bandes befindet, analog durchgeführt werden. Es wird wiederum in einem Bitvektor gespeichert, bei welchen Positionen x_i eine Inkrementierung zu einem Verlassen des k-Bandes führen würde und dann eine einfache UND-Verknüpfung durchgeführt. Für die REACH-Metrik (siehe Abschnitt 5.3.2.1) ist diese Verbesserung allerdings nicht möglich, da dabei Abhängigkeiten zwischen den einzelnen Dimensionen bestehen.

Als letzten Schritt beschleunigen wir die Berechnung der Kantengewichte [29]. Da jede Kante eindeutig einer möglichen Spalte eines Alignments zugeordnet werden kann und sich die Spalten zu (x, r_i) und (x, r_{i+1}) nur in einer Position unterscheiden, wird die jeweils aktuelle Spalte in einem String `char[i]` für $i = 0, \ldots, n-1$ gespeichert, der nur mit dem Lückensymbol „–“ initialisiert wird. Dann wird die veränderte Position $j = n - b[i+1] + 1$ wie folgt modifiziert:

$$\texttt{char}[j] = \begin{cases} - & \text{falls } d[i+1]_{[b[i+1]]} = 0 \\ s_{j,\alpha} & \text{falls } d[i+1]_{[b[i+1]]} = 1 \end{cases} \qquad \text{mit } \alpha = r_{i+1,j}$$

Bei der Verwendung von quasi-affinen Lückenstrafen und Kantenkosten $c(f \mid e)$ (siehe Abschnitt 4.4) ist es außerdem möglich, Teile der Berechnung, die nur von der Vorgängerkante e abhängen, in einem Vorverarbeitungsschritt durchzuführen.

Zusätzlich könnte man auch noch die Berechnung des Sum-Of-Pairs-Maßes so modifizieren, dass nur noch die sich ändernden Terme betrachtet werden [29]. Wir bezeichnen die vorausgegangenen Kosten mit z. Da sich nur die Position `char[j]` mit $j = n - b[i+1] + 1$ geändert hat, genügt es, folgende Schritte für z durchzuführen:

- Subtraktion der paarweisen Kosten zwischen dem alten Wert von `char`$[j]$ und allen anderen Zeichen.
- Addition der paarweisen Kosten zwischen dem neuen Wert von `char`$[j]$ und allen anderen Zeichen.

Das Sum-Of-Pairs-Maß lässt sich dann in Zeit $O(n)$ berechnen und man erhält für $n \geq 5$ weniger Terme als bei einer herkömmlichen Methode (siehe Definition 2.8).

In der Praxis zeigt sich jedoch, dass die gewöhnliche Berechnung anhand der Definition effizienter ist. Dies liegt u. a. daran, dass man für viele Knoten nicht alle $O(n^2)$ Terme addieren muss [18]. Häufig kann schon nach den ersten Termen gezeigt werden, dass der Knoten nicht auf einem kürzesten Weg zum Zielknoten liegen kann (siehe Abschnitt 4.2.2) oder dass bereits ein kürzerer Weg zum betrachteten Knoten bekannt ist.

Zudem brauchen wir bei Knoten, die außerhalb des Gitters oder k-Bandes liegen, keine Kantenkosten berechnen, wohingegen die oben beschriebene Modifikation eine Adaption der aktuellen Kantenkosten in jedem Schritt notwendig macht. Dieses Argument ist insbesondere bei dem von uns entwickelten k-Band-Algorithmus (siehe Abschnitt 5.3) ausschlaggebend, da hier sehr viele Knoten aufgrund der Suchraumreduktion nicht betrachtet werden.

6.3 Resultate

In diesem Abschnitt wollen wir abschließend die vorgestellten Algorithmen und deren Parameter experimentell analysieren. Dabei unterscheiden wir jeweils zwischen linearen und quasi-affinen Lückenstrafen. Für alle Testläufe wird eine PAM-250-Distanzmatrix (siehe Abbildung B.1 in Anhang B) eingesetzt. Lineare Lückenstrafen verwenden Kosten in Höhe von $\gamma = 12$, quasi-affine Funktionen Lückenöffnungskosten von $g_{\text{op}} = 8$ und Lückenfortsetzungskosten von $g_{\text{ext}} = 12$.

Wir beginnen diesen Abschnitt mit der Analyse verschiedener Parameter der vorgestellten Algorithmen auf den in Abschnitt 6.1 ausgewählten Instanzen der BAliBASE. Dabei werden in den Abbildungen folgende Abkürzungen für die verschiedenen Varianten der Algorithmen verwendet:

UniDir-A*: einfache unidirektionale A*-Suche ohne k-Band

UniDir-REACH: unidirektionaler k-Band-Algorithmus mit REACH-Metrik

UniDir-MAX: unidirektionaler k-Band-Algorithmus mit MAX-Metrik

UniDir-Multi: unidirektionaler Multi-k-Band-Algorithmus

BiDir-A*: einfache bidirektionale A*-Suche ohne k-Band

BiDir-REACH: bidirektionaler k-Band-Algorithmus mit REACH-Metrik

BiDir-MAX: bidirektionaler k-Band-Algorithmus mit MAX-Metrik

BiDir-Multi: bidirektionaler Multi-k-Band-Algorithmus

OMA: der progressive Algorithmus aus [36]

Die Alternierungsstrategien für die bidirektionale Suche werden wie in Abschnitt 4.3 mit *alt*, *minQ* und *minP* bezeichnet.

Abschließend führen wir analog zu Reinert et al. in [36] Experimente auf einer speziellen Familie von sehr ähnlichen Sequenzen (Cytochrome C) durch.

6.3.1 Experimente mit BAliBASE

Wir beginnen die experimentelle Analyse mit der Gegenüberstellung der in Abschnitt 5.3.4 vorgestellten Methoden zur Minimierung der Progressivitätskosten. Es folgt ein Vergleich von Trie und Listendarstellung zur Verwaltung markierter Knoten und Kanten (siehe Abschnitt 6.2.1). Danach werden paarweise und dreifache Alignments zur Berechnung der unteren Schranken aus Abschnitt 4.2.1 untersucht.

Wir fahren mit einem Vergleich der in Abschnitt 5.3.2 beschriebenen Metriken REACH und MAX sowie der Multi-k-Band-Strategie aus Abschnitt 5.3.3 fort. Anschließend werden der unidirektionale und der bidirektionale k-Band-Algorithmus aus Abschnitt 5.3 analysiert und der entsprechenden Variante ohne k-Band gegenübergestellt.

Auf den vorhergehenden Ergebnissen aufbauend untersuchen wir Eigenschaften von linearen und quasi-affinen Lückenstrafen. Außerdem wird das Konvergenzverhalten der vorgestellten Algorithmen analysiert. Abschließend vergleichen wir die Resultate unserer Untersuchungen mit denen von OMA [36] und fassen die zentralen Ergebnisse der Analyse zusammen.

6.3.1.1 Progressivitätskosten

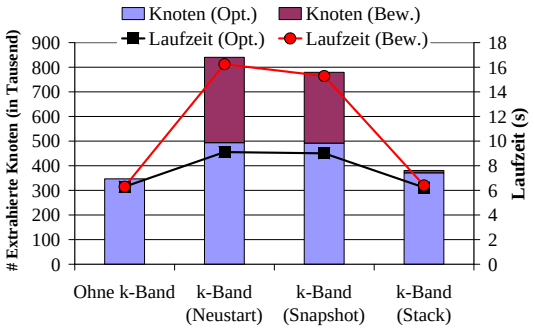
In diesem Abschnitt werden wir zunächst die in Abschnitt 5.3.4 vorgestellten Varianten zur Minimierung der Progressivitätskosten analysieren. Dabei betrachten wir sowohl unidirektionale als auch bidirektionale Testläufe und vergleichen jeweils die Version ohne k-Band, die reine Neustart-Methode, die Snapshot-Methode sowie die k-Band-Stack-Methode miteinander.

Die k-Band-Algorithmen verwenden die REACH-Metrik mit einfacher Erweiterung des k-Bandes, d. h. Verdoppelung von k nach jeder Iteration. Die bidirektionale Suche arbeitet mit der Alternierungsstrategie minQ. Damit in den Testläufen möglichst viele Knoten betrachtet werden, kommen paarweise Alignments zum Einsatz. Außerdem werden Tries zur Verwaltung der Knoten und Kanten benutzt.

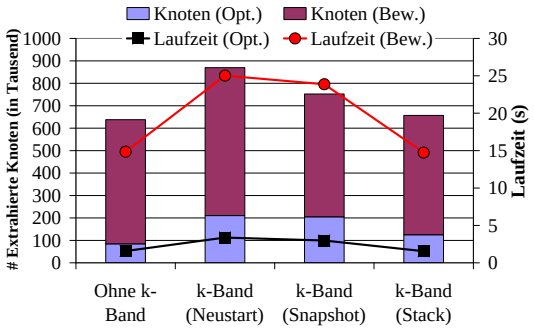
Abbildung 6.4 zeigt Daten für zwei ausgewählte typische Testläufe unter Verwendung linearer Lückenstrafen. Die entsprechenden Daten für quasi-affine Lückenstrafen sind in Abbildung 6.5 dargestellt. Die Anzahl extrahierter Knoten bzw. Kanten wird als Säulendiagramm dargestellt und in Optimumsberechnung und Optimalitätsbeweis unterteilt. Zusätzlich veranschaulicht ein Liniendiagramm die entsprechend benötigte Laufzeit.

Die ausgewählten Instanzen lassen sich für beide Arten von Lückenstrafen unter Verwendung von paarweisen Alignments zur Berechnung der unteren Schranken beweisbar optimal lösen. Dabei handelt es sich einerseits um eine Instanz mit Sequenzlänge < 100 und durchschnittlicher Identität von 18% (`1ubi`). Andererseits wird eine Instanz mit Sequenzlängen von ca. 200 Zeichen und 30% durchschnittlicher Identität verwendet (`1ad2`).

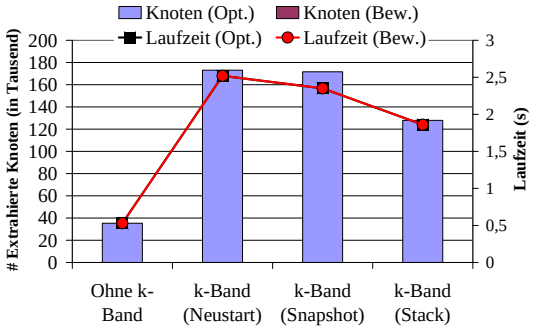
Es fällt auf, dass die Anzahl endgültig markierter Kanten im quasi-affinen Fall deutlich größer ist als die der endgültig markierten Knoten für lineare Lückenstrafen.



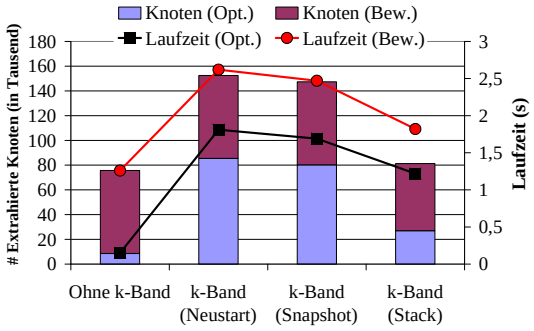
(a) 1ubi (unidirektional)



(b) 1ubi (bidirektional)

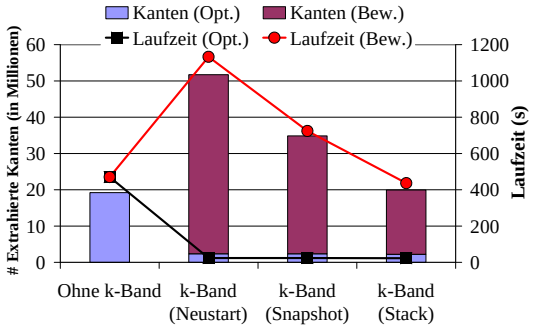


(c) 1ad2 (unidirektional)

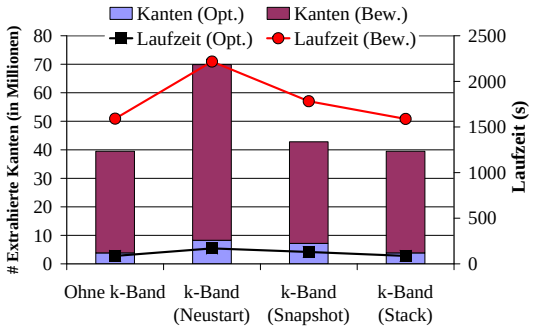


(d) 1ad2 (bidirektional)

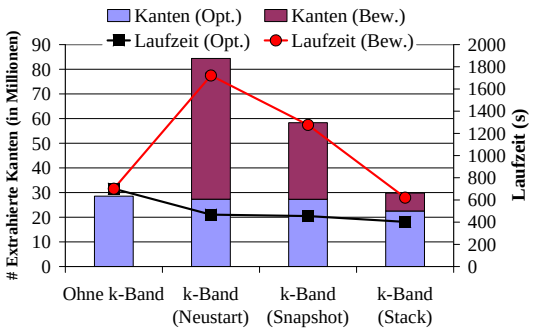
Abbildung 6.4: Progressivitätskosten bei Verwendung von linearen Lückenstrafen.



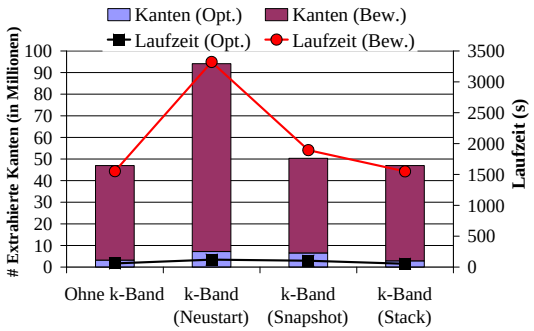
(a) 1ubi (unidirektional)



(b) 1ubi (bidirektional)



(c) 1ad2 (unidirektional)



(d) 1ad2 (bidirektional)

Abbildung 6.5: Progressivitätskosten bei Verwendung von quasi-affinen Lückenstrafen.

Außerdem erkennt man, dass die Laufzeit proportional zur Anzahl der extrahierten Knoten bzw. Kanten ist. Beide Ansätze zur Minimierung der Progressivitätskosten extrahieren weniger Elemente doppelt aus der Prioritätswarteschlange als die Neustart-Methode und reduzieren die benötigte Laufzeit.

Die k-Band-Stack-Methode stellt sich als beste progressive Variante heraus. Es zeigt sich, dass bei der Snapshot-Methode weiterhin viele Knoten doppelt extrahiert werden. Dies liegt u. a. daran, dass nicht notwendigerweise das erste Verlassen des Bandes zu einer besseren Lösung führen muss. Häufig werden mehr als die gespeicherten Knoten von der nachfolgenden Iteration erneut extrahiert und erst durch einen späteren Randknoten Verbesserungen erzielt. Die k-Band-Stack-Methode hingegen betrachtet nur Knoten doppelt, für die in einer späteren Iteration ein kürzerer Weg gefunden wird. Dieser Weg benutzt Knoten, die in der vorhergehenden Iteration noch außerhalb des k-Bandes lagen. Zudem beachtet die k-Band-Stack-Methode sämtliche Randknoten als Kandidaten und führt die Suche beim vielversprechendsten derartigen Knoten fort. Auf diese Weise findet sie tendenziell früher einen besseren Weg.

Ein weiterer Grund für das schlechtere Abschneiden der Snapshot-Variante liegt darin, dass sehr viele Informationen verworfen werden, falls das k-Band schon am Anfang der Iteration verlassen wird. Snapshots liefern erst gute Ergebnisse, wenn viele Knoten kopiert werden können. Dies ist auch daran zu erkennen, dass der Unterschied zwischen Neustart-Methode und Snapshot-Methode bis zur Ermittlung des Optimums nicht so groß ausfällt wie die entsprechende Lücke bei der Beweiszeit. Der Nutzen der Snapshot-Methode wird erst gegen Ende der Berechnungen größer, wohingegen die k-Band-Stack-Methode kontinuierlich Verbesserungen bringt.

Zudem ist die Snapshot-Methode recht speicheraufwändig, da jeweils zusätzlicher Speicherplatz zum Duplizieren der Datenstrukturen benötigt wird. Der Speicherbedarf wächst dabei mit dem Nutzen der Methode. Die k-Band-Stack-Methode hingegen verwaltet lediglich einen Stack für die erzeugten Randknoten. Wir erhalten demnach nicht nur ein besseres Laufzeitverhalten, sondern auch einen deutlich geringeren Speicherbedarf, wenn wir die k-Band-Stack-Methode benutzen.

Man kann jedoch erkennen, dass auch die k-Band-Stack-Methode Knoten bzw. Kanten doppelt extrahiert, da die Vergleichsläufe ohne k-Band mit weniger extrahierten Elementen auskommen. Die Lücke fällt hier unterschiedlich groß aus, wobei in den vorliegenden Tests kein Zusammenhang zu Anzahl, Länge oder Ähnlichkeit der Sequenzen erkennbar war.

Wir können zusammenfassend festhalten, dass sich die k-Band-Stack-Methode als eindeutiger Sieger herausstellt. Daher verwenden wir diese Variante in sämtlichen folgenden Experimenten.

6.3.1.2 Datenstrukturen für markierte Knoten und Kanten

Wir werden nun untersuchen, welche der Datenstrukturen (Trie oder Listendarstellung, siehe Abschnitt 6.2.1) sich in der Praxis zur Verwaltung der betrachteten Knoten und Kanten empfiehlt. Hierbei sollen die Aspekte Laufzeit und Speicherbedarf

untersucht werden. Für die Testläufe wird die REACH-Metrik mit Verdoppelung von k nach jeder Iteration sowie die Alternierungsstrategie minQ verwendet.

Das Laufzeitverhalten der beiden Datenstrukturen hängt von der Anzahl der markierten Knoten und deren Verteilung im Graphen ab. Da dies wiederum von der Wahl der unteren Schranke beeinflusst wird, haben wir sowohl für paarweise als auch für dreifache Alignments Testläufe durchgeführt. Wir konnten in unseren Experimenten jedoch keine Abhängigkeiten erkennen. Wir benutzen daher für die folgenden graphischen Darstellungen die Ergebnisse der dreifachen Alignments.

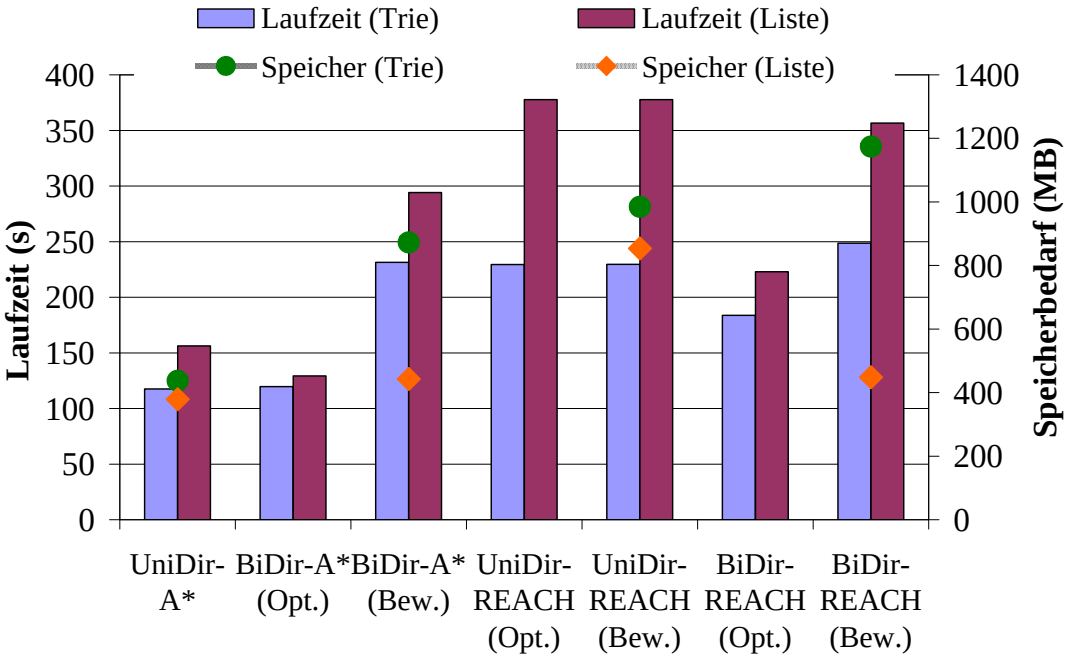
Abbildung 6.6 zeigt jeweils ein repräsentatives Beispiel für Laufzeit und Speicherbedarf bei Verwendung von linearen bzw. quasi-affinen Lückenstrafen. Dabei wird die Laufzeit durch ein Säulendiagramm dargestellt, bei dem zwischen den verschiedenen Algorithmen sowie Optimier- und Beweiszeit unterschieden wird. Punkte stellen den benötigten Speicher der verschiedenen Varianten dar.

Es wurde eine relativ schwere Instanz für lineare Lückenstrafen ausgewählt. Dabei handelt es sich um eine Instanz mit Sequenzlängen zwischen 400 und 500 Zeichen und 29% durchschnittlicher Identität (`1ac5`). Da diese Instanz für quasi-affine Lückenstrafen nicht mehr beweisbar optimal gelöst werden konnte, wurde für quasi-affine Lückenstrafen eine kleinere Instanz mit Sequenzlängen von weniger als 100 Zeichen und 18% durchschnittlicher Identität (`1ubi`) ausgesucht, um auch hier ein Ergebnis für den Optimalitätsbeweis darstellen zu können.

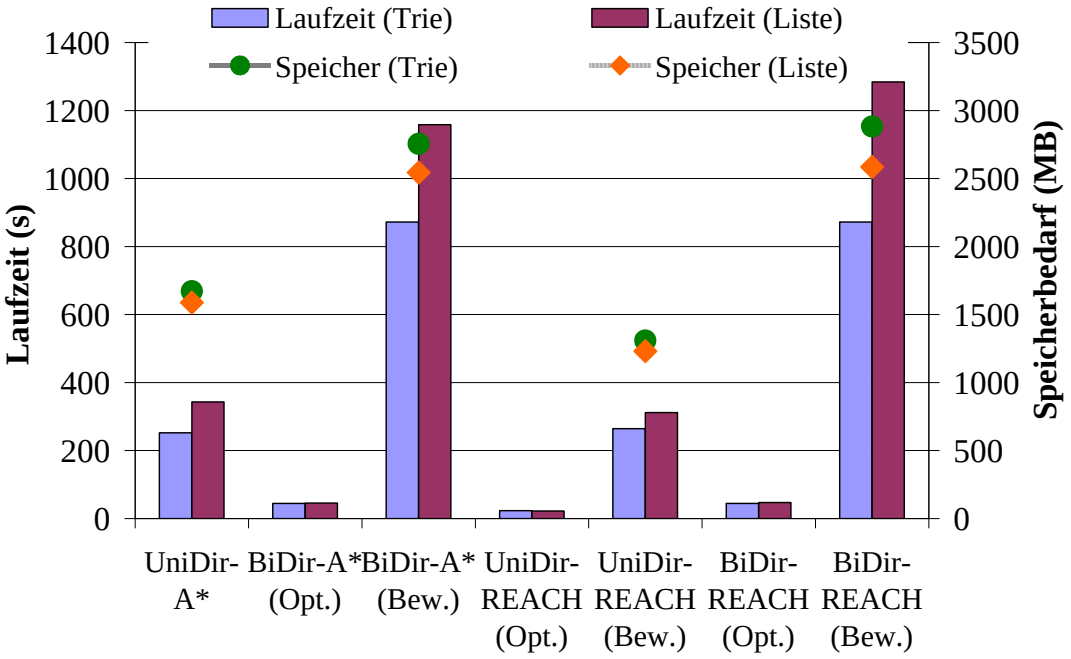
Es ist zu erkennen, dass die Trie-Datenstruktur die Listendarstellung bzgl. der Laufzeit in fast allen Fällen deutlich schlägt. Je höher die Größenordnung der Laufzeit, desto größer fällt die Lücke zwischen den beiden Datenstrukturen aus. Daraus kann gefolgert werden, dass ein Trie für größere Datenmengen deutlich effizienter ist. Für kleine Datenmengen ist der Unterschied zwischen den beiden Datenstrukturen marginal.

Des Weiteren kann man sehen, dass ein Trie deutlich mehr Speicher benötigt. Dies liegt daran, dass für die Konstruktion der Baumstruktur zusätzlicher Speicher erforderlich ist. Der relative Unterschied zwischen den beiden Datenstrukturen ist bei Verwendung von linearen Lückenstrafen höher als bei quasi-affinen Lückenstrafen. Dabei ist der Speicherbedarf eines Tries teilweise doppelt so hoch wie der der Listendarstellung. Ein Grund hierfür ist, dass der Großteil des Speichers bei quasi-affinen Lückenstrafen für die Speicherung der Kanten benötigt wird. Da wir jedoch Kanten jeweils in einer Liste beim entsprechenden Startknoten speichern, wird nicht für jede neue Kante zusätzlicher Speicher im Trie benötigt. Liegen hingegen lineare Lückenstrafen vor, werden nur Knoten verwaltet. Hierbei wird für jeden neuen Knoten auch der Trie entsprechend vergrößert.

Außerdem ist die relative Differenz des Speicherbedarfs der beiden Datenstrukturen bei bidirektionaler Suche größer als bei unidirektionaler Suche. Dies liegt daran, dass bei der bidirektionalen Suche zusätzlich zu den erreichten Knoten sämtliche bereits endgültig markierten Knoten verwaltet werden müssen, um das Pruning für Vorwärts- und Rückwärtssuche zu ermöglichen (siehe Abschnitt 4.3). Da hierfür dieselbe Datenstruktur verwendet wird, macht sich der Unterschied im Speicherbedarf bei bidirektionaler Suche deutlicher bemerkbar als bei unidirektionaler Suche.



(a) Lineare Lückenstrafe (1ac5)



(b) Quasi-affine Lückenstrafe (1ubi)

Abbildung 6.6: Laufzeit und Speicherbedarf von Trie und Listendarstellung.

Bei der Wahl der Datenstruktur sollte demnach zwischen Speicherbedarf und Laufzeit abgewogen werden. Um eine bessere Lösung im Fall von nur wenig Arbeitsspeicher zu erhalten, kann die Listendarstellung verwendet werden. Allerdings wird die Berechnung insbesondere bei quasi-affinen Lückenstrafen sowie bei linearen Lückenstrafen für Instanzen mit Sequenzlängen von mehr als 200 Zeichen und durchschnittlicher Identität $< 40\%$ deutlich verlangsamt.

Wie schon in Abschnitt 6.2.1 vermutet, ist die Listendarstellung bei schmalen k-Bändern effizienter als Tries, da hier nur wenige Knoten um ein Diagonalelement herum betrachtet werden. Dieser Zusammenhang wird in Abbildung 6.7 visualisiert. Es werden dieselben Instanzen verwendet wie in Abbildung 6.6. Die Darstellung bezieht sich auf die unidirektionale Suche, da hier in jeder Iteration garantiert bis zum Beweis der Optimalität gerechnet wird. Die Laufzeit der einzelnen Iterationen wird in Form eines Säulendiagramms, das Verhältnis von Trie zu Listendarstellung als Linie visualisiert.

Wir können festhalten, dass dass die Listendarstellung nur bis zu einer Bandbreite von $k \leq 16$ Vorteile gegenüber einem Trie liefert. Da jedoch das Band in jedem Schritt verdoppelt wird, kann ein Trie diesen Vorsprung sehr schnell wieder ausgleichen. Die Listendarstellung ist also nur für Instanzen geeignet, deren Optimum sich innerhalb eines recht schmalen k-Bandes befindet. Sie stellt aber bzgl. Speicherplatzbedarf und Implementierbarkeit eine gute Alternative zum Trie dar.

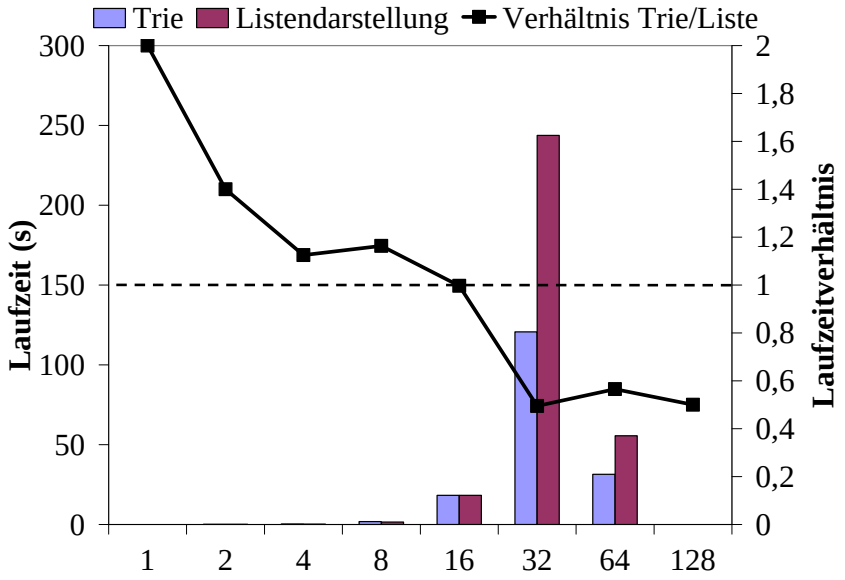
6.3.1.3 Berechnung der unteren Schranke

In diesem Abschnitt vergleichen wir die beiden vorgestellten Methoden zur Bestimmung der unteren Schranken bei der A*-Suche (paarweise und dreifache Alignments, siehe Abschnitt 4.2.1). Dabei werden zwei Aspekte untersucht:

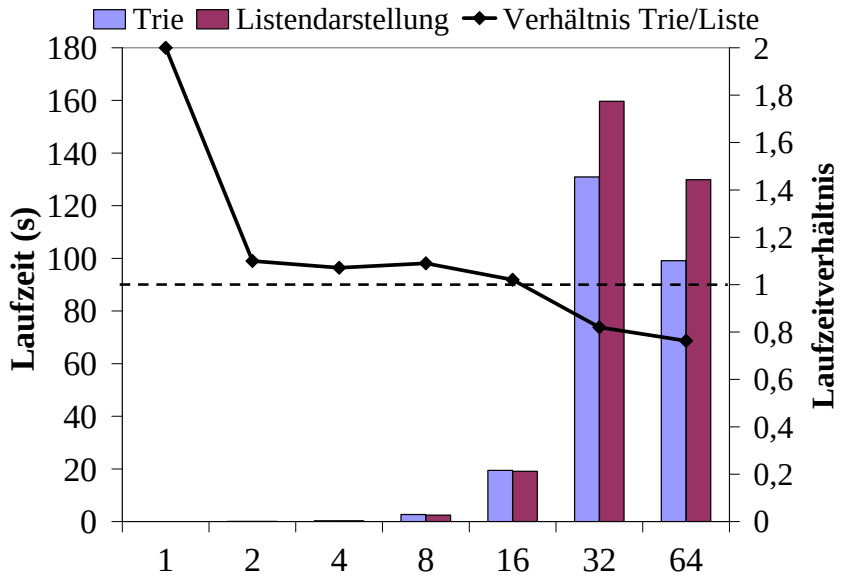
1. Laufzeitverhalten.
2. Einfluss auf die Anzahl der aus der Prioritätswarteschlange extrahierten Knoten bzw. Kanten.

Die Laufzeit, die für die Vorverarbeitung benötigt wird, ist bei dreifachen Alignments höher als bei paarweisen Alignments. Dies ist insbesondere bei vielen und / oder langen Sequenzen der Fall. Da bei der bidirektionalen Suche jeweils Schranken für Vorwärts- und Rückwärtssuche berechnet werden müssen, ist dort der Aufwand für die Vorverarbeitung doppelt so groß wie bei der unidirektionalen Suche.

Aus diesen Gründen ist eine Betrachtung der Laufzeit wichtig. Wir wollen analysieren, für welche Instanzgröße sich der Mehraufwand in der Vorverarbeitung lohnt. Für die k-Band-Algorithmen verwenden wir die REACH-Metrik mit Verdoppelung von k nach jeder Iteration. Zur Verwaltung der betrachteten Knoten und Kanten kommt aufgrund der Ergebnisse des vorherigen Abschnitts ein Trie zum Einsatz. Außerdem wird bei der bidirektionalen Suche die Alternierungsstrategie minQ benutzt.



(a) Lineare Lückenstrafe (1ac5)



(b) Quasi-affine Lückenstrafe (1ubi)

Abbildung 6.7: Laufzeit von Trie und Listendarstellung für verschiedene Breiten des k-Bandes.

An dieser Stelle sei angemerkt, dass die Speicherung der Tabellen für die dreifachen Alignments mehr Speicher in Anspruch nimmt als die für paarweise Alignments. Um zum Zeitpunkt der Kanten-Relaxation möglichst effizient die entsprechenden unteren Schranken berechnen zu können, speichern wir alle verwendeten Tabellen für paarweise und dreifache Alignments.

Bei Verwendung von paarweisen Alignments benötigen wir für n Sequenzen mit maximaler Länge ℓ_{max} insgesamt $\binom{n}{2}$ Integer-Tabellen der Größe $O(\ell^2_{\max})$. Daraus ergibt sich ein Speicherbedarf von $O\left(\binom{n}{2} \cdot \ell^2_{\max}\right) = O\left(n^2 \cdot \ell^2_{\max}\right)$.

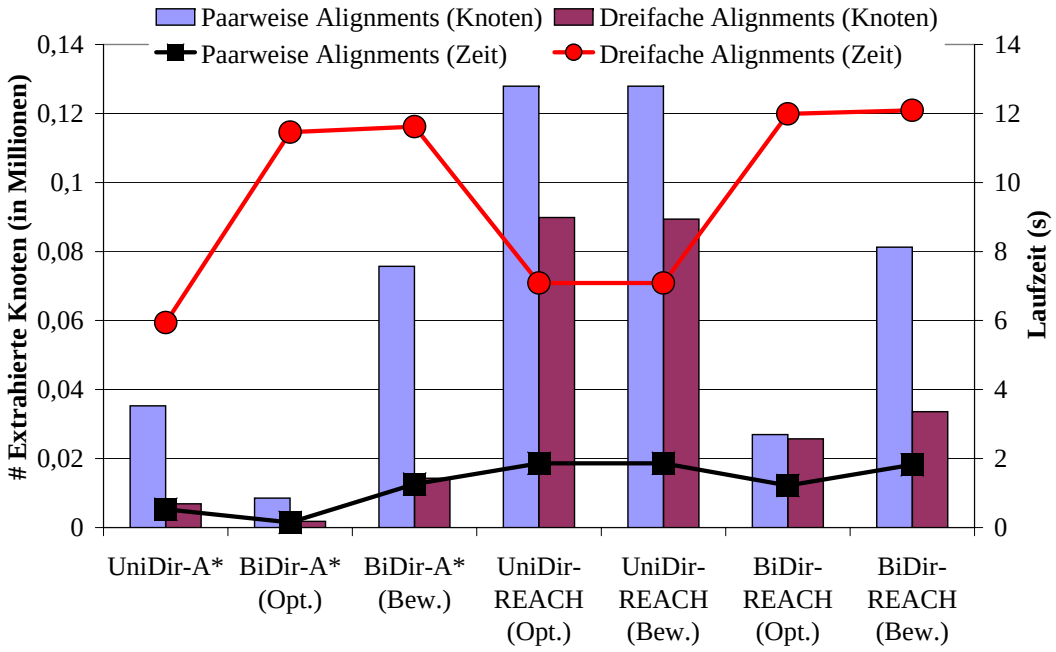
Bei der Konstruktion der dreifachen Alignments werden jeweils drei paarweise Alignments durch ein dreifaches Alignment ersetzt. Damit erzeugt jedes Tripel zusätzlichen Speicher von $O(\ell^3_{\max}) - 3 \cdot O(\ell^2_{\max})$. Diese Zusatzkosten können sich jedoch durch eine effizientere Reihenfolge bei der Knotenauswahl sowie mögliche Verbesserung beim Pruning wieder ausgleichen und sind daher vernachlässigbar.

Wir betrachten zunächst lineare Lückenstrafen. Hierbei hat sich in den Experimenten herausgestellt, dass sich die Verwendung von dreifachen Alignments nur für Instanzen mit Sequenzlängen von mehr als 200 Zeichen und einer durchschnittlichen Ähnlichkeit $< 45\%$ sowie bei Sequenzlängen von weniger als 200 Zeichen und durchschnittlicher Identität $< 20\%$ lohnt. Abbildung 6.8(a) zeigt eine Beispielinstanz aus der BAliBASE 1.0 mit Sequenzlängen von ca. 200 Zeichen und 30% durchschnittliche Identität (`1ad2`). Es ist deutlich zu sehen, dass die Varianten mit dreifachen Alignments weniger Knoten aus der Prioritätswarteschlange extrahieren, aber dennoch eine höhere Laufzeit aufweisen. Wie zu erwarten war, fällt der Unterschied bei den bidirektionalen Algorithmen höher aus als bei den unidirektionalen.

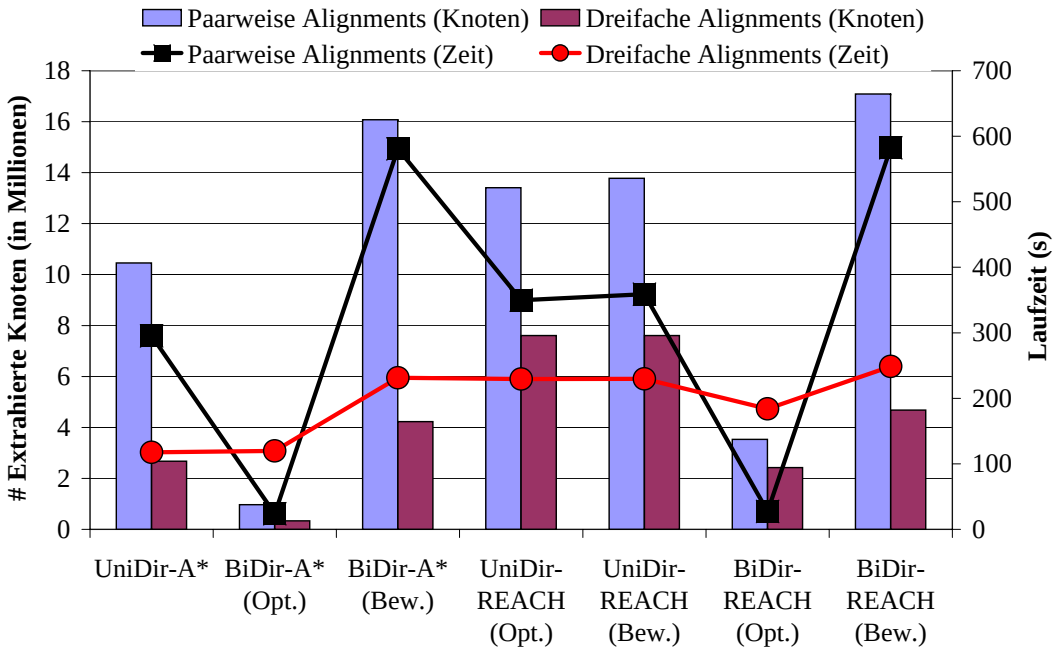
In Abbildung 6.8(b) ist das Resultat für ein Alignment visualisiert, bei dem sich die Verwendung der dreifachen Alignments teilweise auszahlt. Hierbei handelt es sich um Sequenzen mit mehr als 400 Zeichen und 29% durchschnittlicher Identität (`1ac5`). Man kann erkennen, dass in den Fällen, in denen nur wenige Knoten extrahiert werden müssen (Optimierzeit: BiDir-A* und BiDir-REACH), die Verwendung paarweiser Alignments kürzere Laufzeiten aufweist. Im Gegensatz dazu sind die Laufzeiten für die Fälle, in denen mehr Knoten (hier: mehr als 2 Millionen) betrachtet werden, unter Verwendung von dreifachen Alignments deutlich geringer.

Bei der Verwendung von quasi-affinen Lückenstrafen zeigt sich ein anderes Ergebnis. Hierbei ist insbesondere ausschlaggebend, dass der gesamte Speicherbedarf durch Einsatz der dreifachen Alignments deutlich reduziert werden kann. Dies kann dadurch begründet werden, dass der Speicherbedarf von der Reihenfolge, in der Kanten extrahiert werden, sowie der Qualität der oberen Schranke für das Pruning abhängt. Beides wird wiederum von der Güte der unteren Schranken beeinflusst. In den Testläufen konnten viele Instanzen nur unter Verwendung von dreifachen Alignments mit den uns zur Verfügung stehenden Ressourcen optimal gelöst werden.

Außerdem kann bei quasi-affinen Lückenstrafen die Laufzeit durch den Einsatz von dreifachen Alignments deutlich reduziert werden. Die Anwendung von paarweisen Alignments zahlt sich nur noch bei Sequenzen mit weniger als 100 Zeichen oder einer durchschnittlichen Identität $> 40\%$ aus. Abbildung 6.9 zeigt ein Beispiel für quasi-affine Lückenstrafen. Wir haben hierbei die Instanz aus Abbildung 6.8(a)



(a) 1ad2



(b) 1ac5

Abbildung 6.8: Anzahl endgültig markierter Knoten sowie Laufzeit für paarweise und dreifache Alignments bei Verwendung von linearen Lückenstrafen.

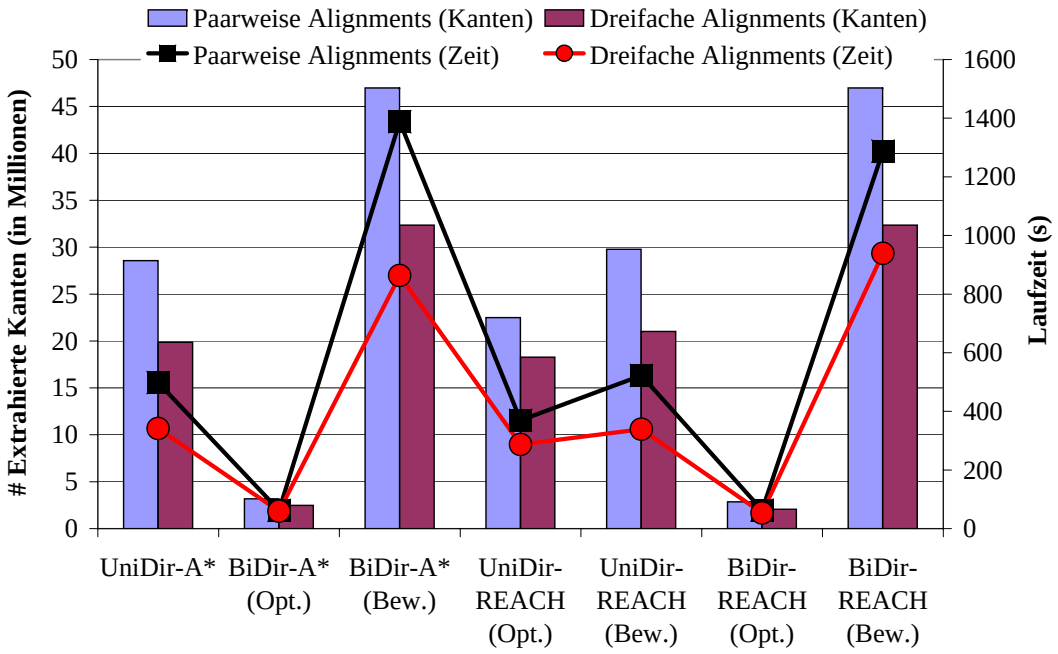


Abbildung 6.9: Anzahl endgültig markierter Knoten sowie Laufzeit für paarweise und dreifache Alignments bei Verwendung von quasi-affinen Lückenstrafen.

ausgewählt, um den Unterschied zwischen den beiden Arten von Lückenstrafen zu verdeutlichen.

Wir können also festhalten, dass bei Verwendung von quasi-affinen Lückenstrafen grundsätzlich der Einsatz von dreifachen Alignments zur Berechnung der unteren Schranken empfehlenswert ist. Je höher die Anzahl zu extrahierender Elemente, desto größer ist der Vorteil einer Nutzung von dreifachen Alignments. Folglich lohnt sich der Einsatz der paarweisen Alignments zur Berechnung der unteren Schranken nur bei einfachen Instanzen. Da die Laufzeit für diese Instanzen jedoch insgesamt sehr niedrig ist, empfehlen wir in der Regel die Benutzung von dreifachen Alignments.

6.3.1.4 Varianten des k-Band-Algorithmus

Im folgenden Abschnitt soll das Verhalten des k-Band-Algorithmus näher untersucht werden. Dabei werden sowohl die unidirektionale als auch die bidirektionale A*-Suche betrachtet. Da wir uns besonders für das Laufzeitverhalten auf schwierigen Instanzen interessieren und dreifache Alignments dort bessere Resultate liefern, verwenden sämtliche Testläufe dreifache Alignments zur Berechnung der unteren Schranken. Außerdem sollen für alle Testläufe dieselben Schranken zum Einsatz kommen. Des Weiteren wird ein Trie zur Verwaltung der markierten Knoten benutzt und zwischen linearen und quasi-affinen Lückenstrafen sowie Optimier- und Beweiszeit unterschieden.

Metriken und Erweiterungsstrategien

Wir werden zunächst die in Abschnitt 5.3.2 vorgestellten Metriken sowie die Multi-k-Band-Strategie aus Abschnitt 5.3.3 experimentell vergleichen.

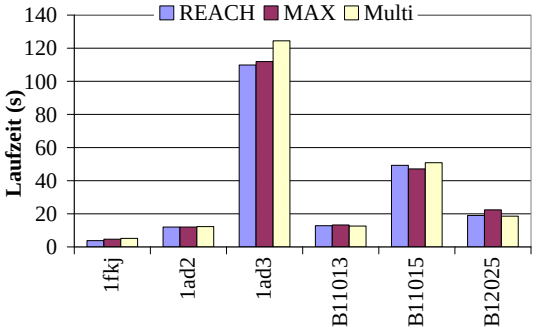
Wir betrachten zunächst den bidirektionalen k-Band-Algorithmus. Es stellt sich heraus, dass es hierbei keinen eindeutigen Sieger gibt. REACH- und MAX-Metrik zeigen in der Regel ein sehr ähnliches Laufzeitverhalten und stellen in dieser Hinsicht robuste Varianten des k-Bandes dar. Die Multi-k-Band-Strategie weist hingegen deutliche Abweichungen sowohl nach oben als auch nach unten auf. Dies führt zu dem Schluss, dass die Multi-k-Band-Strategie stärker von der Struktur der Eingabesequenz abhängt als die einfache Erweiterung durch Verdoppelung des Parameters k.

Abbildung 6.10 visualisiert die Laufzeiten für den bidirektionalen k-Band-Algorithmus an einigen Beispielinstanzen. Hierbei haben wir Sequenzen unterschiedlicher Länge und Ähnlichkeit ausgewählt (vgl. Tabelle 6.1). Um aussagekräftigere Ergebnisse zu erhalten, haben wir für lineare Lückenstrafen einige zusätzliche Instanzen verwendet, die für quasi-affine Lückenstrafen nicht mehr beweisbar optimal gelöst werden können und dafür auf die graphische Darstellung der Instanzen mit sehr kleinen Laufzeiten verzichtet. Ein vollständiger Vergleich von REACH- und MAX-Metrik mit bidirektionaler Suche ist den Tabellen 6.2, 6.3 und 6.4 in Abschnitt 6.3.1.9 zu entnehmen.

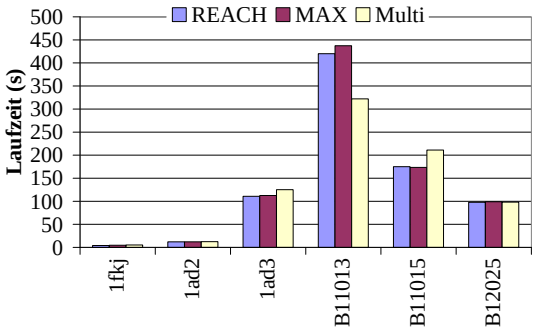
Bei Verwendung des unidirektionalen k-Band-Algorithmus zeigt sich ein anderes Resultat. Hierbei sind die Unterschiede zwischen den verschiedenen Strategien deutlich größer. Insbesondere die Multi-k-Band-Strategie zeigt häufig kein gutes Verhalten und ist nur in sehr wenigen Fällen den anderen Strategien überlegen. Ein Grund hierfür ist, dass die Multi-k-Band-Strategie das Band abhängig von den extrahierten Knoten der vorhergehenden Iteration erweitert. Bei guter Wahl der Erweiterungsrichtungen wird das Optimum schneller gefunden. Bei einer schlechten Wahl kann dies die Berechnungen jedoch deutlich verlangsamen. Schlechte Entscheidungen sind bei der bidirektionalen Suche einfacher zu revidieren, da durch den vorzeitigen Iterationsabbruch erkannt werden kann, dass die ausgewählte Richtung doch nicht so vielversprechend ist wie erwartet. Dies ist bei der unidirektionalen Suche jedoch nicht der Fall, da die Iteration bis zum Ende gerechnet werden muss.

Insgesamt stellt sich die REACH-Metrik als beste Strategie bei der unidirektionalen Suche heraus. Sie ist in den meisten Fällen der MAX-Metrik überlegen und zeigt keine signifikanten Laufzeit-Abweichungen im Vergleich zu den anderen Varianten. Diese Tatsache kann wiederum dadurch erklärt werden, dass die unidirektionale Suche in jeder Iteration bis zur Optimalität weiterrechnen muss. Aufgrund des größeren k-Bandes bei der MAX-Metrik kann es hierdurch zu einer Verlangsamung des Algorithmus kommen. Abbildung 6.11 zeigt die entsprechenden Laufzeiten für die Instanzen, die auch schon für die bidirektionale Suche ausgewählt wurden.

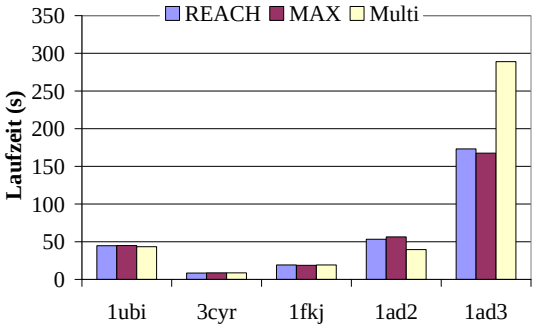
Wir können also zusammenfassen, dass es für die bidirektionale Suche keinen eindeutigen Sieger gibt. Sowohl MAX- als auch REACH-Metrik liefern robuste Ergebnisse. Dahingegen stellt sich die REACH-Metrik bei der unidirektionalen Suche als die beste Strategie heraus.



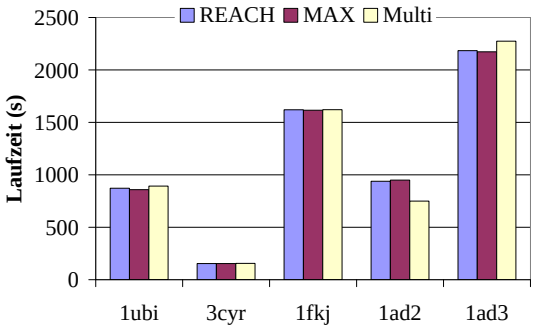
(a) Lineare Lückenstrafen (Optimierzeit).



(b) Lineare Lückenstrafen (Beweiszeit).

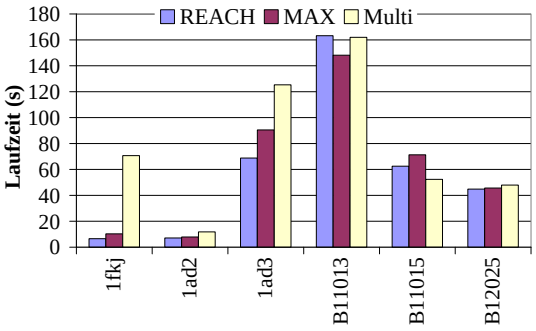


(c) Quasi-affine Lückenstrafe (Optimierzeit).

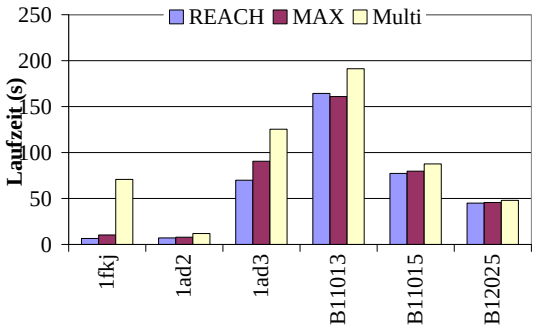


(d) Quasi-affine Lückenstrafe (Beweiszeit).

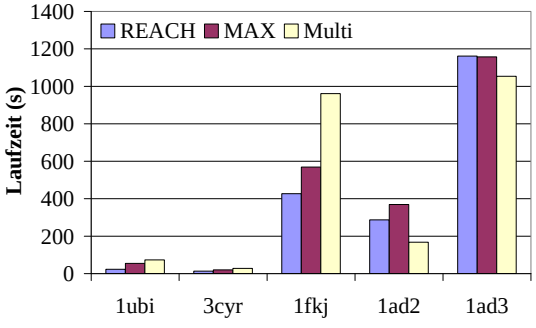
Abbildung 6.10: Laufzeit verschiedener Varianten des k-Bandes (bidirektional).



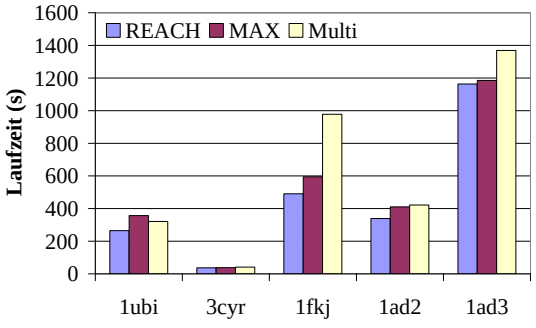
(a) Lineare Lückenstrafen (Optimierzeit).



(b) Lineare Lückenstrafen (Beweiszeit).



(c) Quasi-affine Lückenstrafe (Optimierzeit).



(d) Quasi-affine Lückenstrafe (Beweiszeit).

Abbildung 6.11: Laufzeit verschiedener Varianten des k-Bandes (unidirektional).

Die Laufzeit der verschiedenen Varianten des k-Bandes hängt von den Eigenschaften der jeweiligen Sequenzen ab. Hierbei ist insbesondere die Anzahl aufeinander folgender Lücken im optimalen Alignment von Bedeutung, da das k-Band die maximale Länge solcher Lücken beschränkt und somit die Länge der größten Lücke im optimalen Alignment Einfluss auf die Anzahl benötigter Iterationen hat. Dies ist insbesondere bei der Multi-k-Band-Strategie der Fall. Sie zeigte in den Tests zwar teilweise sehr vielversprechende Ergebnisse, ist aber im Vergleich zu den anderen Strategien weniger robust und besonders bei unidirektionaler Suche nicht konkurrenzfähig. Mögliche Verbesserungen des Verfahrens sehen wir in einer Modifikation der Erweiterungsstrategie sowie in einer problemorientierten Initialisierung des k-Bandes.

Laufzeitverhalten einzelner Iterationen

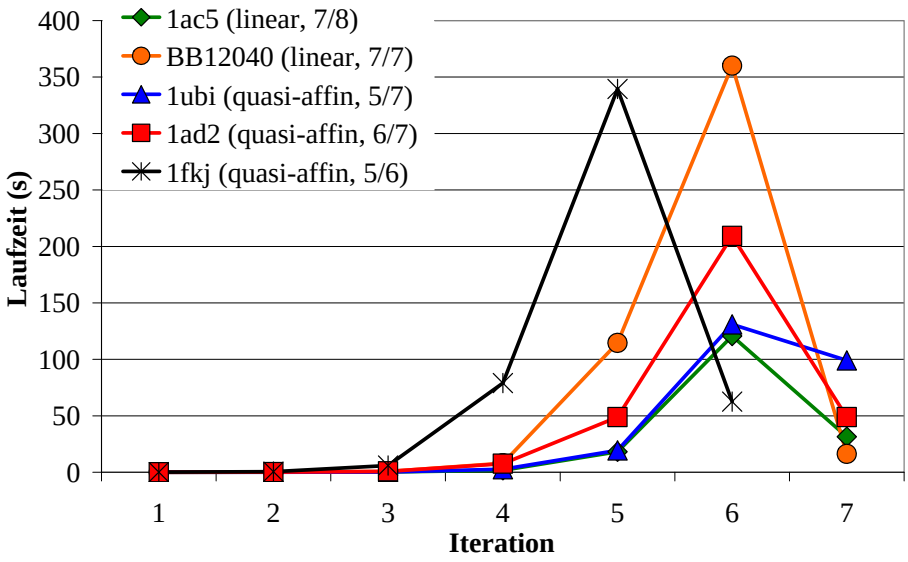
Wir betrachten nun, wie sich die benötigte Laufzeit während der Durchführung der einzelnen Iterationsschritte verändert. Abbildung 6.12 zeigt einige repräsentative Beispiele für unidirektionale und bidirektionale Suche. Dabei haben wir in der Legende jeweils vermerkt, in welcher Iteration das Optimum gefunden wurde und wie viele Iterationen insgesamt notwendig waren.

Es fällt auf, dass sich bei der unidirektionalen Suche die Laufzeit zunächst von Iteration zu Iteration vergrößert und gegen Ende wieder verringert. Die maximale Laufzeit wird normalerweise für die Iteration, in der das Optimum (oder ein Wert nahe am Optimum) gefunden wird, benötigt. Dies ist dadurch zu erklären, dass mehr Knoten durch Pruning ausgeschlossen werden können, wenn eine gute obere Schranke existiert. Das beschriebene Laufzeitverhalten ist selbst bei einigen Instanzen zu beobachten, die erst in der letzten Iteration das Optimum finden (z. B. `B12040`).

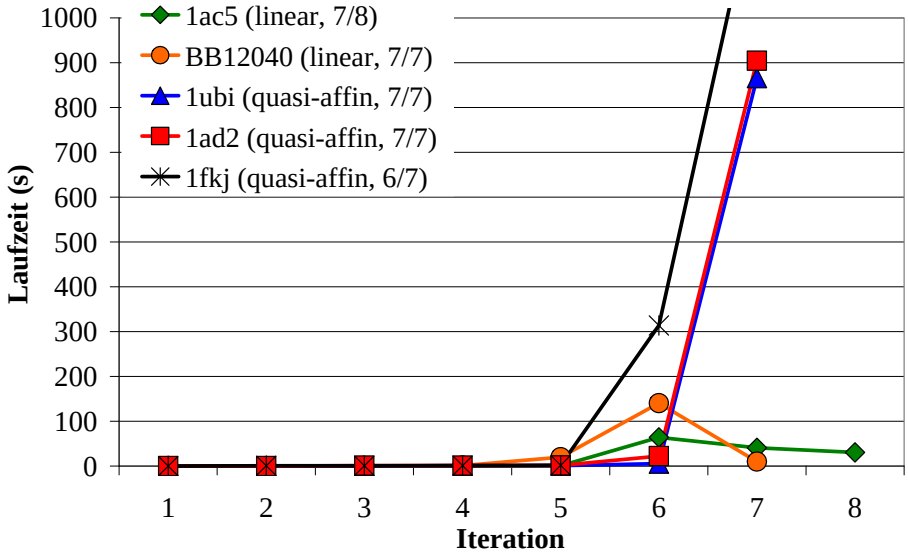
Es existieren nur wenige Instanzen, bei denen die Laufzeit pro Iteration bis zum Ende ansteigt (z. B. `3cyr` mit linearen Lückenstrafen). Solche Fälle kommen in den von uns ausgewählten Testinstanzen nur bei linearen Lückenstrafen und sehr geringen Laufzeiten vor. Daher verzichten wir an dieser Stelle auf die graphische Darstellung.

Bei der bidirektionalen Suche erhält man ein anderes Resultat. Hier ist das Laufzeitverhalten sehr unterschiedlich. In den meisten Fällen steigt die Laufzeit pro Iteration stetig an. Ein Grund hierfür ist, dass bei der bidirektionalen Suche in der Regel das Optimum früher gefunden, aber mehr Zeit für dessen Optimalitätsbeweis benötigt wird. Daher ist für bidirektionale Suchen die Laufzeit der späteren Iterationen höher als die der früheren Iterationen. Wir werden dies im folgenden Abschnitt 6.3.1.5 näher untersuchen.

Es gibt jedoch auch Instanzen, bei denen dasselbe Verhalten wie bei der unidirektionalen Suche zu beobachten ist. Dies ist insbesondere bei einfachen Instanzen und bei linearen Lückenstrafen der Fall.

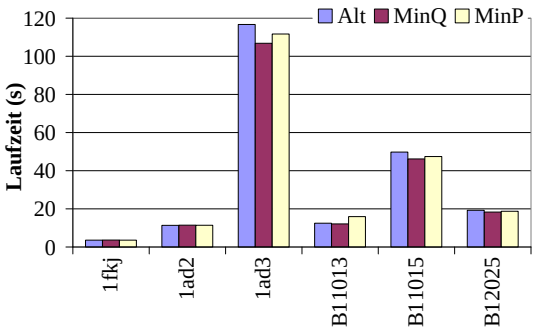


(a) Unidirektionale Suche

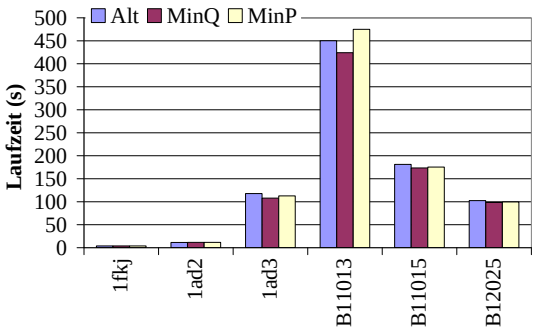


(b) Bidirektionale Suche

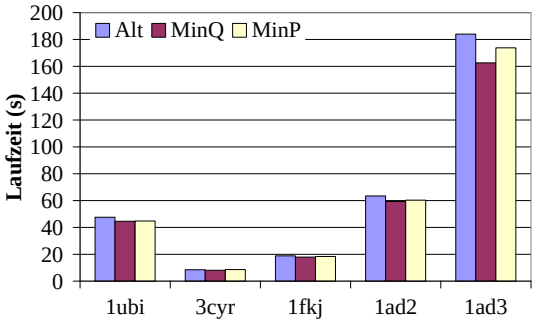
Abbildung 6.12: Laufzeit für BiDir-REACH, aufgesplittet auf einzelne Iterationen.



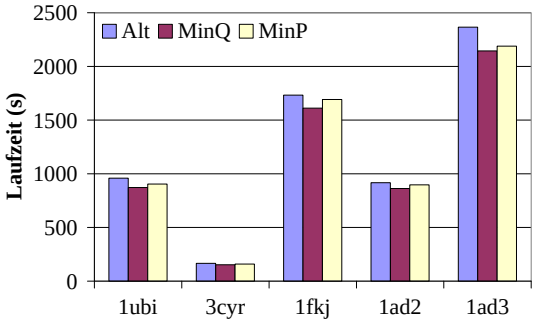
(a) Lineare Lückenstrafen (Optimum).



(b) Lineare Lückenstrafen (Beweis).



(c) Quasi-affine Lückenstrafe (Optimum).



(d) Quasi-affine Lückenstrafe (Beweis).

Abbildung 6.13: Laufzeit von BiDir-A* für verschiedene Alternierungsstrategien.

6.3.1.5 Unidirektionale und bidirektionale Suche

Wir werden nun Vor- und Nachteile von unidirektionaler und bidirektionaler Suche analysieren. Dazu betrachten wir zunächst die unterschiedlichen Alternierungsstrategien (siehe Abschnitt 4.3) für die bidirektionale Suche.

Es stellt sich heraus, dass die Strategie minQ den anderen Varianten deutlich überlegen ist. Es ist also günstig, die Richtung mit der kleineren Prioritätswarteschlange auszuwählen. Dies kann dadurch erklärt werden, dass auf diese Weise die Laufzeiten für die Operationen auf der jeweiligen Prioritätswarteschlange minimiert werden, da diese von der Anzahl der Elemente in der Warteschlange abhängen. Dies ist auch der Grund dafür, dass der Unterschied zwischen den unterschiedlichen Strategien bei größeren Gesamtlaufzeiten deutlicher ausfällt.

Abbildung 6.13 visualisiert die Laufzeiten für die Strategien an einigen Beispielinstanzen. Hierbei wurden dieselben Instanzen wie im vorhergehenden Abschnitt ausgewählt und wiederum zwischen linearen und quasi-affinen Lückenstrafen sowie Optimier- und Beweiszeit unterschieden. Da wir uns wie schon zuvor vor allem für die schwierigen Instanzen interessieren, verwenden die Testläufe einen Trie für Knoten und Kanten sowie dreifache Alignments zur Berechnung der unteren Schranken.

Beim Vergleich von unidirektionaler und bidirektionaler Suche fällt auf, dass die bidirektionale Suche in der Regel zuerst das Optimum findet. Dies liegt daran, dass aufgrund des von uns verwendeten symmetrischen Ansatzes (siehe Abschnitt 4.3.1)

in beiden Richtungen die bestmöglichen unteren Schranken benutzt werden und sich die beiden Suchrichtungen dadurch relativ schnell treffen. Sobald dies geschieht, werden die beiden Teilwege zusammengesetzt und man erhält eine neue Lösung. Bei der unidirektionalen Suche ist es hingegen nötig, die Senke (bzw. eine Kante mit der Senke als Endknoten) aus der Prioritätswarteschlange zu extrahieren, um eine neue Lösung zu erhalten.

Wie schon in Abschnitt 5.3 angesprochen, entspricht meistens eine der ersten Konkatenationen der optimalen Lösung. Die Anzahl der nötigen Konkatenationen hängt von der Anzahl der Sequenzen, der Sequenzlängen und der Breite des k-Bandes ab. Bei der bidirektionalen Suche ohne k-Band reichen bei Sequenzlängen von weniger als 100 Zeichen normalerweise fünf Konkatenationen aus, für Sequenzen mit mehr als 400 Zeichen liegt die Anzahl der benötigten Konkatenationen bei ca. 100. Bis zu einer k-Band-Breite von 64 kommen wir mit $10 - 20$ Konkatenationen aus.

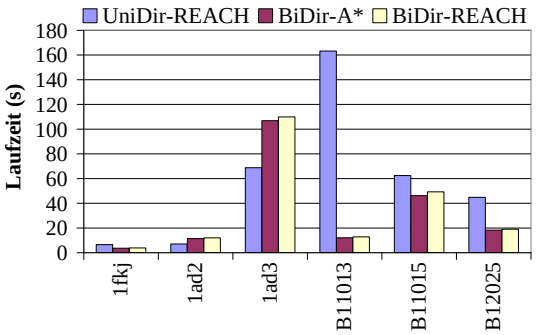
Bei Betrachtung der Beweiszeit der beiden Algorithmen, stellt man fest, dass dort die unidirektionale Suche die bidirektionale Suche bei den meisten Instanzen schlägt. Hierbei ist ausschlaggebend, dass es bei der bidirektionalen Suche viele Möglichkeiten gibt, Vorwärts- und Rückwärtssuche zu konkatenieren. Dies kann damit verglichen werden, bei n gegebenen Sequenzen einen Schnittpunkt zu finden, der einer Spalte eines optimalen Alignments entspricht und damit ein optimales Alignment in zwei Teile teilt. Das Problem, einen solchen Schnittpunkt zu finden, ist NP-schwer [42]. Bei der unidirektionalen Suche ist hingegen nach dem Finden des Optimums keine zusätzliche Beweiszeit mehr nötig, da mit der Extrahierung der Senke ein kürzester s-t-Weg gefunden wurde.

Für die bereits oben ausgewählten Instanzen sind die Unterschiede zwischen unidirektionaler und bidirektionaler Suche in Abbildung 6.14 dargestellt. Für einen Gesamtüberblick über die Optimier- und Beweiszeiten wird auf die Tabellen 6.2, 6.3 und 6.4 in Abschnitt 6.3.1.9 verwiesen. An dieser Stelle sei angemerkt, dass in den beiden Tabellen, die unter Verwendung von dreifachen Alignments zur Berechnung der unteren Schranken erstellt wurden (siehe Tabellen 6.3 und 6.4), einige Instanzen deutliche Abweichungen von den oben beschriebenen Beobachtungen aufweisen. Dies liegt daran, dass bei bidirektionaler Suche der doppelte Aufwand für die Berechnung der unteren Schranke nötig ist (siehe Abschnitt 6.3.1.3). Damit wird der Vorteil der bidirektionalen Suche gegenüber der unidirektionalen Suche in manchen Fällen aufgehoben. Dieser Effekt ist auch in Abbildung 6.14 zu beobachten.

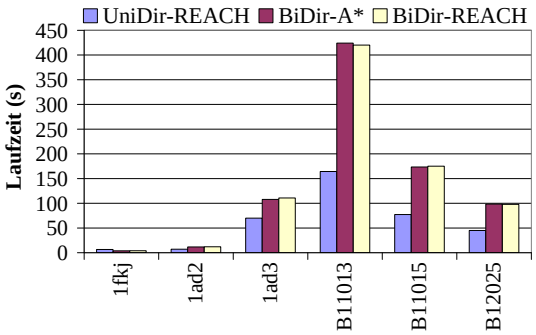
Wir halten fest, dass die Wahl zwischen unidirektionaler und bidirektionaler Suche davon abhängt, ob der Fokus des Anwenders auf der Optimierzeit oder auf der Beweiszeit liegt. Außerdem muss der Effekt der Methode zur Berechnung der unteren Schranken einbezogen werden.

6.3.1.6 Art der Lückenstrafe

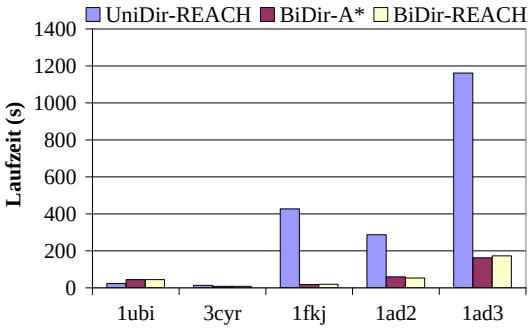
Bei allen vorhergehenden Tests wurde zwischen linearen und quasi-affinen Lückenstrafen unterschieden. Ziel dieses Abschnittes ist es nun, Unterschiede zwischen den beiden Lückenstrafen zu identifizieren.



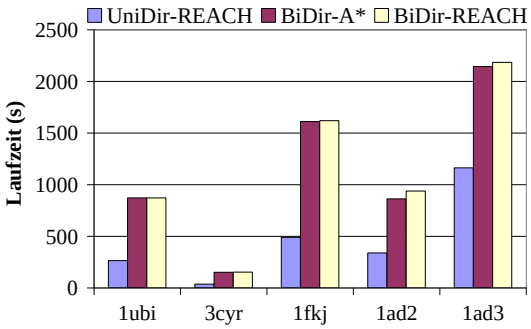
(a) Lineare Lückenstrafen (Optimum).



(b) Lineare Lückenstrafen (Beweis).



(c) Quasi-affine Lückenstrafe (Optimum).



(d) Quasi-affine Lückenstrafe (Beweis).

Abbildung 6.14: Laufzeit für unidirektionale und bidirektionale Suche.

Wir halten zunächst fest, dass bei Verwendung der linearen Lückenstrafen nur Alignments mit geringer biologischer Relevanz berechnet werden. Reinert et al. haben in [36] die Güte der von OMA berechneten Alignments analysiert. Dabei wurde der Durchschnitt über alle Instanzen aus Gruppe 1 der BAliBASE 1.0 (siehe Abschnitt 6.1) gebildet. Wir zitieren folgendes Ergebnis aus [36]:

- Gruppe 1a (durchschnittliche Identität $< 25\%$):
  60% der Kernbereiche werden korrekt aligniert.
- Gruppe 1b (durchschnittliche Identität $20\% - 40\%$):
  92% der Kernbereiche werden korrekt aligniert.
- Gruppe 1c (durchschnittliche Identität $> 35\%$):
  94% der Kernbereiche werden korrekt aligniert.

Laut Reinert et al. sind diese Resultate fast identisch mit denjenigen des besten Programms zur Berechnung optimaler Alignments aus einer ausführlichen Studie von Thompson et al. [44]. Da wir in unserer Implementierung dieselbe Bewertungsfunktion verwenden wie OMA, können wir dieses Resultat für die optimal alignierten Sequenzen übernehmen.

Wir haben in den vorhergehenden Untersuchungen festgestellt, dass sich der Einsatz fortgeschrittener Datenstrukturen und Methoden zur Berechnung der unteren

Schranken nicht für einfache Instanzen und lineare Lückenstrafen lohnt. Dahingegen verhalten sich die unidirektionale und die bidirektionale A*-Suche auf beiden Kostenmodellen identisch.

Bei Betrachtung des k-Band-Algorithmus fällt auf, dass die Verwendung des k-Bandes für quasi-affine Lückenstrafen Vorteile bringen kann. Dies gilt insbesondere für die unidirektionale Suche. Abbildung 6.15 vergleicht die Laufzeiten des k-Band-Algorithmus mit der entsprechenden Variante ohne k-Band. Dabei erkennt man, dass der unidirektionale k-Band-Algorithmus für lineare Lückenstrafen (siehe Abbildung 6.15(a)) in der Regel langsamer ist als die einfache unidirektionale A*-Suche ohne k-Band. Bei quasi-affinen Lückenstrafen ist der unidirektionale k-Band-Algorithmus jedoch meist schneller (siehe Abbildung 6.15(c)).

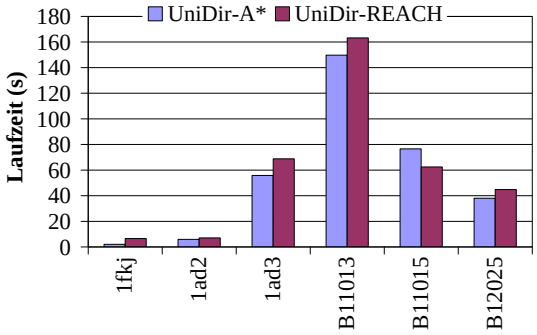
Eine Erklärung hierfür ist, dass die Performance der unidirektionalen A*-Suche ohne k-Band vor allem von der Güte der unteren Schranken abhängt. Bei schlechten unteren Schranken kann es unter Umständen passieren, dass sehr viele Wege zum Ziel „ausprobiert“ werden, bis dieses schließlich aus der Prioritätswarteschlange extrahiert wird. Beim k-Band-Algorithmus haben wir hingegen eine zusätzliche obere Schranke, durch die einige dieser Wege ausgeschlossen werden können. Umgekehrt kann es bei guten unteren Schranken vorkommen, dass die k-Band-Einschränkung den Vorteil der zielgerichteten Suche aufhebt. Hierdurch lassen sich Ausnahmen wie z. B. `BB11015` und `1ad3` erklären.

Wir können in Abbildung 6.15(b) und 6.15(d) eine ähnliche Tendenz für die bidirektionalen Algorithmen erkennen. Diese Beobachtung wird auch in den Abbildungen 6.4, 6.5 und 6.9 bestätigt. Der Unterschied ist hier aber nicht ganz so deutlich, was sich wiederum dadurch erklären lässt, dass bei der bidirektionalen Suche das Optimum in der Regel früher gefunden wird und jeweils verschiedene untere Schranken für Vorwärts- und Rückwärtssuche verwendet werden.

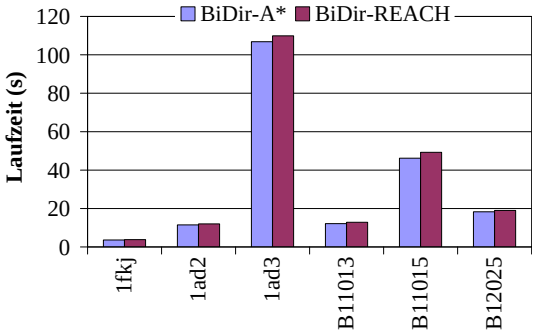
Da bei quasi-affinen Lückenstrafen Kanten anstelle von Knoten in die Prioritätswarteschlange eingefügt werden, wächst die Größe der Eingabeinstanzen exponentiell mit der Anzahl an Sequenzen: Für n Sequenzen mit maximaler Länge $\ell_{\max}$ entsteht ein Graph mit $O(\ell_{max}^n)$ Knoten und $O(2^n \cdot \ell_{max}^n)$ Kanten. Dies hat Auswirkungen auf Laufzeit und Speicherbedarf, wobei der jeweilige Anstieg abhängig von der betrachteten Instanz und Variante des Algorithmus ist. Bei der Laufzeit haben wir z. B. Abweichungen bis zu einem Faktor von 550 (UniDir-A* auf `BB11021`) und 1500 (BiDir-A* Beweis auf `BB11029`) beobachtet. Es gab aber auch Instanzen (z. B. `1fmb`), bei denen nur minimale Unterschiede vorhanden waren.

Dieser Laufzeitunterschied ist vor allem von den Lücken in einem optimalen Alignment abhängig. Die Differenz ist besonders groß, falls sich das Optimum bzgl. linearer Lückenstrafen deutlich von dem bzgl. quasi-affiner Lückenstrafen unterscheidet. Dies ist z. B. dann der Fall, wenn das optimale Alignment für lineare Lückenstrafen viele kleine Lücken beinhaltet. Diese werden bei Verwendung von quasi-affinen Lückenstrafen meist zu einer langen fortlaufenden Lücke zusammengesetzt.

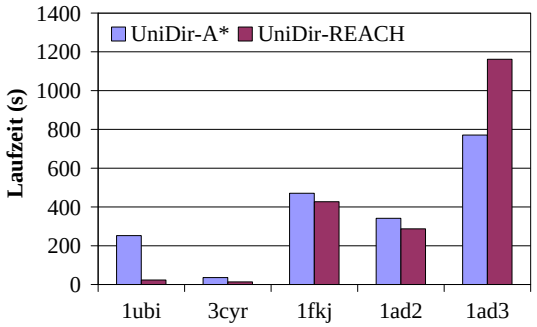
Des Weiteren ist der Laufzeitunterschied zwischen linearen und quasi-affinen Lückenstrafen bei der Beweiszeit der bidirektionalen Suche besonders deutlich. Ein



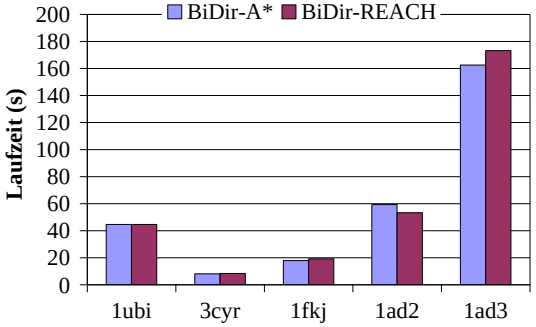
(a) Lineare Lückenstrafen (Unidirektional).



(b) Lineare Lückenstrafen (Bidirektional).



(c) Quasi-affine Lückenstrafe (Unidirektional).



(d) Quasi-affine Lückenstrafe (Bidirektional).

Abbildung 6.15: Optimierzeit für Algorithmen mit und ohne k-Band.

Grund hierfür ist, dass bei Verwendung von quasi-affinen Lückenstrafen mehr Konkatenationen möglich sind als bei linearen Lückenstrafen (vgl. Abschnitt 6.3.1.5).

Abbildung 6.16 visualisiert den Speicherbedarf für einige ausgewählte Instanzen. Der Unterschied wirkt sich hier besonders darauf aus, ob schwierige Instanzen noch lösbar sind oder nicht. Insgesamt haben wir Faktoren von bis zu 500 (UniDir-A* auf `1ad2`) festgestellt. Selbst bei einfacheren Instanzen trat noch ein Faktor von 50 auf.

6.3.1.7 Konvergenzverhalten

Wir untersuchen abschließend das Konvergenzverhalten der betrachteten Algorithmen. Da insbesondere bei Verwendung von quasi-affinen Lückenstrafen einige Instanzen nicht mehr mit den uns zur Verfügung stehenden Ressourcen beweisbar optimal gelöst werden können, interessiert uns, wie schnell die Lösungen gegen das Optimum konvergieren. Alle Testläufe verwenden Tries sowie die Berechnung von unteren Schranken durch dreifache Alignments. Außerdem wird die Alternierungsstrategie minQ für die bidirektionale Suche benutzt.

Wir haben zunächst eine Instanz mit Sequenzlängen von mehr als 400 Zeichen und einer durchschnittlichen Identität von 29% ausgewählt (`1ac5`), die für lineare Lückenstrafen beweisbar optimal gelöst werden kann, bei der für quasi-affine Lückenstrafen jedoch nur eine suboptimale Lösung ermittelbar ist. Die Ergebnisse sind in

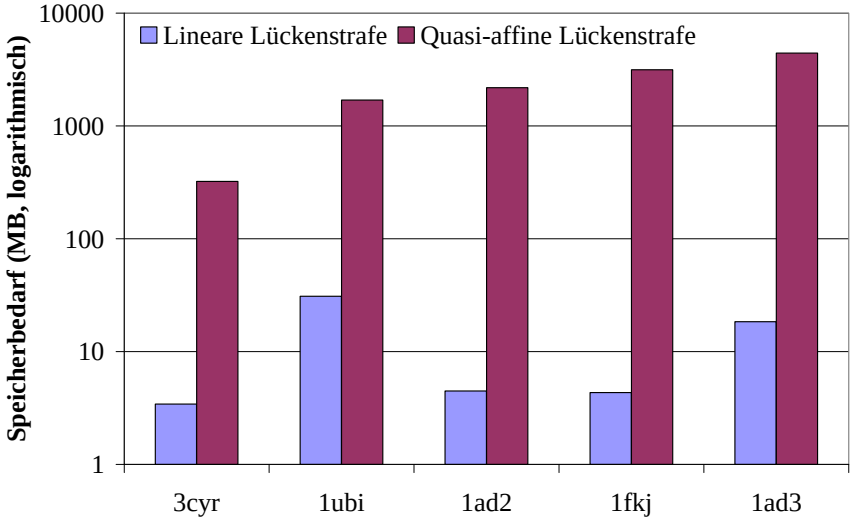


Abbildung 6.16: Speicherbedarf bei Verwendung von linearen und quasi-affinen Lückenstrafen.

Abbildung 6.17 dargestellt. Dabei ist auffällig, dass zu Beginn viele betrachtete Algorithmen das gleiche Verhalten zeigen und sich somit die zugehörigen Laufzeitkurven überschneiden.

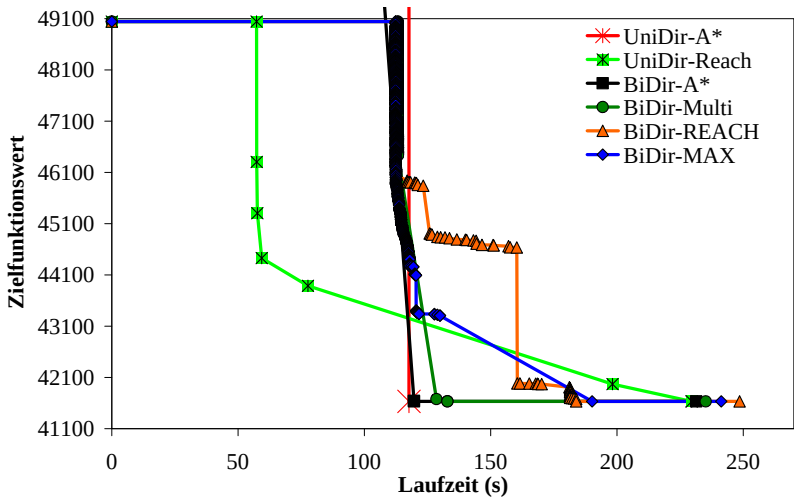
Man erkennt, dass bei beiden Lückenstrafen der unidirektionale k-Band-Algorithmus zu Beginn schneller gegen das Optimum konvergiert, dann aber von den bidirektionalen Varianten „überholt“ wird. Die Laufzeitkurve für die unidirektionale Suche hat eine logarithmische Form, wohingegen die bidirektionalen Varianten insbesondere bei der linearen Lückenstrafe sehr unterschiedlich verlaufen.

Dieses Verhalten kann dadurch erklärt werden, dass bei der unidirektionalen Suche jeweils am Ende einer Iteration eine neue Lösung gefunden wird und die Laufzeit für die Iterationen bis zum Erreichen des Optimums stetig wächst (siehe Abschnitt 6.3.1.5). Dahingegen kann es bei der bidirektionalen k-Band-Variante während einer Iteration zu mehreren Verbesserungen kommen. Hierbei ist die erste gefundene Lösung häufig schon nah am Optimum der aktuellen Iteration (siehe Abschnitt 5.3). Aufgrund dieses Verhaltens ergibt sich ein Kurvenverlauf mit großen sprungartigen Verbesserungen, denen viele kleinere Optimierungen folgen.

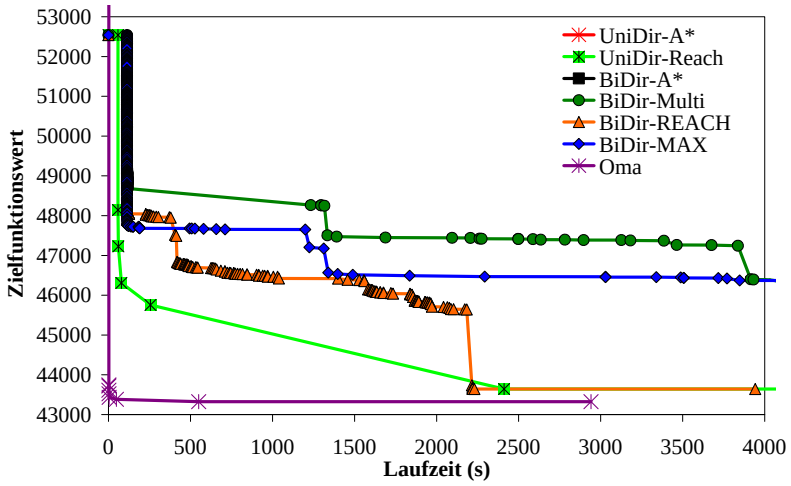
Bei der bidirektionalen Variante ohne k-Band erkennt man, dass die erste gefundene Lösung bei linearen Lückenstrafen nahe am Optimum ist. Bei Verwendung von quasi-affinen Kosten findet sie hingegen keine zulässige Lösung und erscheint daher nicht im Diagramm. Dasselbe gilt für die unidirektionale A*-Suche ohne k-Band.

Für quasi-affine Lückenstrafen wurden in Abbildung 6.17(b) die Laufzeiten für OMA aufgenommen. Es ist zu erkennen, dass OMA deutlich schneller als die k-Band-Algorithmen gegen das Optimum konvergiert. Zudem bestimmt OMA eine optimale Lösung. Die vom unidirektionalen und bidirektionalen k-Band-Algorithmus mit REACH-Metrik berechneten Lösungen sind jedoch nahe am Optimum.

Abbildung 6.18 zeigt zwei weitere repräsentative Beispiele für quasi-affine Lückenstrafen. Dabei terminieren sämtliche Algorithmen mit beweisbar optimalen



(a) 1ac5: lineare Lückenstrafe

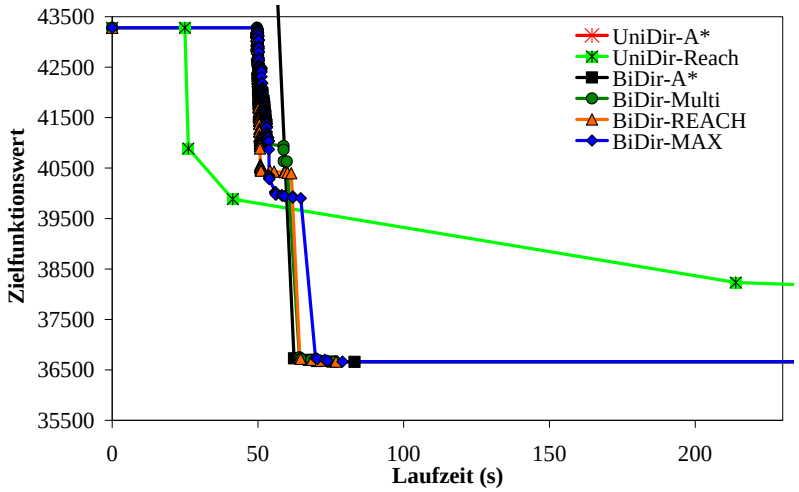


(b) 1ac5: quasi-affine Lückenstrafe

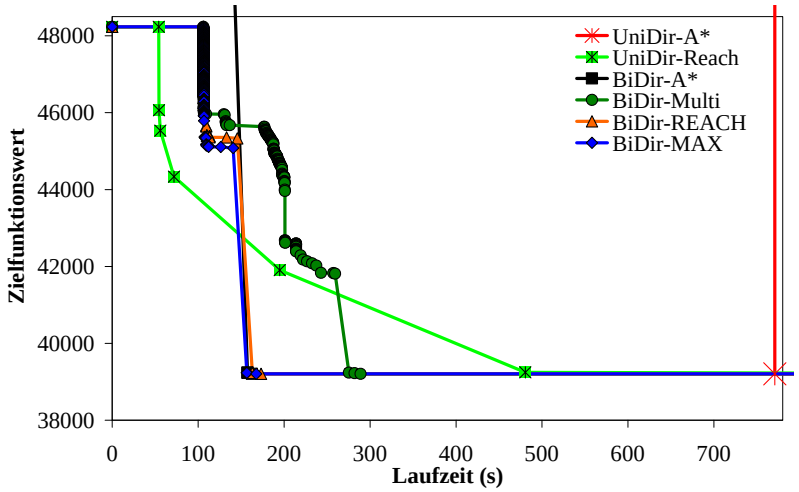
Abbildung 6.17: Konvergenzverhalten 1.

Lösungen. Auch hier ist die logarithmische Form der Kurve für die unidirektionale Suche erkennbar. Die bidirektionale Variante ohne k-Band findet als erstes das Optimum. Die unidirektionale Suche ohne k-Band berechnet zwar in beiden Fällen die optimale Lösung, benötigt dazu aber deutlich mehr Zeit als die anderen Algorithmen.

Bei den Varianten des k-Bandes bestätigen sich die Ergebnisse aus Abschnitt 6.3.1.4. MAX- und REACH-Metrik liefern häufig ähnliche Laufzeiten, wobei Abweichungen in beide Richtungen existieren. Für die Multi-k-Band-Strategie gibt es Instanzen, bei denen die Variante gute Resultate erzielt. In vielen Fällen ist bei der Multi-k-Band-Strategie jedoch mit einer deutlich schlechteren Laufzeit, verglichen mit den anderen Verfahren, zu rechnen (siehe z. B. Abbildung 6.18(b)).



(a) 1amk: quasi-affine Lückenstrafe



(b) 1ad3: quasi-affine Lückenstrafe

Abbildung 6.18: Konvergenzverhalten 2.

Wir können also zusammenfassend sagen, dass der unidirektionale k-Band-Algorithmus zwar zu Beginn schneller große Verbesserungen erzielt als die bidirektionale Variante, insgesamt aber langsamer gegen das Optimum konvergiert. Dabei gibt es keine nennenswerte Unterschiede zwischen der MAX- und der REACH-Metrik. Lediglich die Multi-k-Strategie ist auf einigen Instanzen deutlich langsamer. Die bidirektionale A*-Suche ohne k-Band berechnet zwar häufig am schnellsten das Optimum, ist aber bei schwierigen Instanzen nicht immer in der Lage, überhaupt eine zulässige Lösung auszugeben. Somit zeigen die bidirektionalen k-Band-Algorithmen mit MAX- oder REACH-Metrik in der Regel das beste Konvergenzverhalten.

6.3.1.8 Vergleich mit OMA

Wir wollen nun den entwickelten Algorithmus mit OMA (siehe Abschnitt 5.1, [36]) vergleichen. Dafür haben wir die Laufzeiten von OMA auf den ausgewählten Testinstanzen ermittelt. Hierbei ist anzumerken, dass uns nur eine 32-Bit-Version von OMA zur Verfügung stand und somit diese Testläufe ein Speicherlimit von 2GB hatten. Dennoch war OMA in der Lage, für jede Instanz eine zulässige – evtl. sub-optimale – Lösung zu berechnen.

Leider müssen wir insgesamt feststellen, dass der von uns entwickelte Ansatz die Erwartungen nicht erfüllen konnte. Es gibt nur sehr wenige Instanzen, bei denen unser Algorithmus schneller eine optimale Lösung berechnet als OMA. Die Beweiszeit lag bei allen verwendeten Instanzen über der von OMA. Außerdem gibt es Instanzen, bei denen unser Ansatz nur eine sub-optimale Lösung ausgibt, OMA jedoch eine beweisbar optimale Lösung berechnet. Die Laufzeiten für die verschiedenen Instanzen werden in Tabelle 6.4 den Ergebnissen für unseren Algorithmus gegenübergestellt. Diejenigen Instanzen, die von unserer Implementierung schneller optimiert werden, sind gelb markiert.

Wir können aus diesem Resultat schließen, dass die durch den k-Band-Algorithmus berechneten oberen Schranken schlechter (d. h. weiter entfernt von der optimalen Lösung) als die entsprechenden Schranken von OMA sind. Dies liegt u. a. daran, dass die obere Schranke, die in einer Iteration erreicht werden kann, durch die Struktur des k-Bandes beschränkt wird. Es kann z. B. vorkommen, dass eine große fortlaufende Lücke nötig ist, um ein gutes Alignment zu erhalten. Da dies erst in späteren Iterationen möglich ist, kann es vorkommen, dass eine optimale Lösung für ein schmales k-Band nur eine schlechte obere Schranke für das gesuchte optimale Alignment darstellt. Wir haben dennoch die Hoffnung, dass dieser Nachteil des k-Bandes durch eine andere Initialisierungs- oder Erweiterungsstrategie in weiterführenden Arbeiten verringert werden kann.

6.3.1.9 Zusammenfassung der Ergebnisse

Wir wollen nun die bisherigen Ergebnisse zusammenfassen. Insgesamt konnten bei linearen Lückenstrafen 28 der 31 Instanzen beweisbar optimal gelöst werden, bei quasi-affinen Lückenstrafen immerhin noch 14 von 31. Zusätzlich erhielten wir beim unidirektionalen k-Band-Algorithmus für fünf Instanzen und beim bidirektionalen k-Band-Algorithmus für neun Instanzen eine optimale Lösung, konnten den Optimalitätsbeweis aber nicht mit den uns zur Verfügung stehenden Ressourcen zu Ende führen[1].

Die k-Band-Algorithmen haben für alle Instanzen auf beiden Kostenmodellen eine zulässige (evtl. sub-optimale) Lösung berechnet. Dabei können bei Verwendung von linearen Lückenstrafen bis zu fünf Sequenzen mit einer Länge von höchstens 400 Zeichen und einer durchschnittlichen Identität von mindestens 45% beweisbar

[1] Da beide Programme dasselbe Kostenmodell verwenden, haben wir in diesem Fall die letzte berechnete Lösung mit der beweisbar optimalen Lösung von OMA verglichen.

optimal gelöst werden. Bei bis zu 200 Zeichen reicht eine durchschnittliche Identität von 30% aus.

Bei Verwendung von quasi-affinen Lückenstrafen sind die k-Band-Algorithmen in der Lage, bis zu vier Sequenzen mit einer Länge von bis zu 400 Zeichen und einer durchschnittlichen Identität von mindestens 45% beweisbar optimal zu alignieren. Bei Sequenzlängen von weniger als 200 Zeichen können auch Instanzen mit mindestens 30% durchschnittlicher Identität beweisbar optimal gelöst werden. Instanzen mit fünf Sequenzen können wir nur für Sequenzlängen von bis zu 200 Zeichen optimal alignieren, wobei bei weniger als 100 Zeichen eine durchschnittliche Identität von mindestens 25% und bei längeren Sequenzen eine durchschnittliche Identität von mindestens 40% vorausgesetzt wird.

Die Tabellen 6.2, 6.3 und 6.4 fassen die Laufzeiten der durchgeführten Testläufe zusammen. Wir unterscheiden dabei zwischen linearen (Tabellen 6.2 und 6.3) und quasi-affinen Lückenstrafen (Tabelle 6.4).

Bei den linearen Lückenstrafen gibt es jeweils eine Tabelle für die beiden Möglichkeiten zur Berechnung der unteren Schranken, da hier bei einigen Instanzen deutliche Unterschiede zu sehen sind. Paarweise Alignments sind in Tabelle 6.2, dreifache in Tabelle 6.3 dargestellt. Instanzen, bei denen eine der beiden Methoden eindeutig als Sieger hervorgeht, sind in der entsprechenden Tabelle markiert. Eine rote Markierung gibt dabei an, dass die Art der unteren Schranke weniger geeignet ist. Gelb kennzeichnet die bessere Variante. Hierbei ist u. a. die Instanz `2cba` hervorzuheben, da hier nur bei Verwendung von dreifachen Alignments die Optimalität der Lösung bewiesen werden konnte. Ein ähnliches Verhalten zeigen die Instanzen `2myr` und `1rthA`.

Nicht gekennzeichnete Instanzen zeigen ein unterschiedliches Verhalten. Hierbei kann es einerseits vorkommen, dass dreifache Alignments nur für die unidirektionale Suche bessere Ergebnisse liefern. Andererseits gibt es mehrere Instanzen, bei denen der Vorteil der dreifachen Alignments erst bei der Beweiszeit deutlich wird (z. B. `1pfc`). Letzteres ist insbesondere bei der bidirektionalen Suche häufig der Fall (z. B. `1ac5`, `1rthA`). Es existieren außerdem Instanzen, die weder mit paarweisen noch mit dreifachen Alignments als untere Schranke optimiert werden können (z. B. `1sbp`, `1pamA`).

Zusammenfassend kann man hier sagen, dass sich bei linearen Lückenstrafen der Einsatz von dreifachen Alignments insbesondere bei langen Sequenzen oder bei Sequenzen mit geringer durchschnittlicher Identität lohnt. Bei quasi-affinen Lückenstrafen sind sie grundsätzlich zu empfehlen. Außerdem zeigen sie Vorteile bei Instanzen mit großen Längenunterschieden (z. B. `BB11013`).

Sämtliche Testläufe verwenden einen Trie zu Verwaltung der markierten Knoten. Dies führt insbesondere bei quasi-affinen Lückenstrafen dazu, dass viele Testinstanzen mit dem zur Verfügung stehenden Speicher nicht optimal gelöst bzw. die Optimalität der Lösung nicht bewiesen werden konnte. Wir haben Testläufe, bei denen nur suboptimale Alignments berechnet oder der Optimalitätsbeweis nicht beendet wurde, mit „(*)“ gekennzeichnet. Ein „/“ gibt an, dass der Algorithmus keine zulässige Lösung finden konnte. Die jeweils schnellste Optimierzeit bzw. Beweiszeit

Instanz	UniDir-A*	BiDir-A*		UniDir-REACH			BiDir-REACH			BiDir-MAX		
		Opt	Beweis	Opt	Beweis	k	Opt	Beweis	k	Opt	Beweis	k
1ubi (4)	6,3	1,58	14,85	6,23	6,4	32 /64	1,55	14,73	32 /64	1,57	14,07	32 /64
1wit (5)	88,64	13,21	254,6	50,3	84,04	16 / 64	14,3	271,4	32 / 64	14,86	261,55	32 / 64
3cyr (4)	0,39	0,12	0,9	0,78	0,78	32 / 32	0,3	0,96	32 /32	0,17	0,94	32 /32
1pfc (5)	5,24	0,68	9,15	2,59	5,5	8 / 32	0,73	9,3	16 / 32	1,01	9,38	16 / 32
1fmb (4)	0,01	0	0,01	0,04	0,04	8 / 8	0,03	0,04	8 / 8	0,04	0,04	8 / 8
1fkj (5)	0,52	0,08	0,9	5,37	5,39	16 / 32	0,22	0,96	16 / 32	1,21	1,83	16 / 32
3grs (4)	312,09	35,82	672,31	64,58	310,13	32 / 128	38,71	680,68	64 / 128	32,95	671,73	32 / 128
1sbp (5)	/	/	/	(*)	(*)	(32 / 64)	(*)	(*)	(32 / 32)	(*)	(*)	(64 / 64)
1ad2 (4)	0,53	0,15	1,26	1,86	1,86	32 / 32	1,22	1,82	32 /32	1,27	1,63	32 /32
2eba (5)	4092,38	355,97	(*)	2371,59	4401,55	32 / 128	363,49	(*)	(64 / 128)	363,69	(*)	(64 / 128)
1zin (4)	0,24	0,08	0,62	0,24	0,36	8 / 32	0,17	0,74	8 / 32	0,21	0,77	8 / 32
1amk (5)	7,65	0,97	13,86	20,59	21,2	16 / 32	1,22	13,92	16 / 32	2,15	13,87	32 / 32
2myr (4)	/	(*)	(*)	300,16	(*)	(32 / 128)	2288,88	(*)	(256 / 256)	2423,91	(*)	(256 / 256)
1pamA (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(128 / 128)	(*)	(*)	(16 / 16)
1ac5 (4)	295,23	23,75	580,72	349,77	358,82	64 / 128	27,27	582,39	64 / 128	34,21	592,59	64 / 128
2ack (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(64 / 64)	(*)	(*)	(32 / 32)
1ad3 (4)	3,15	0,73	6,38	17,01	17,81	32 / 64	5,16	9,68	32 / 64	14,06	14,99	64 / 64
1rthA (5)	4084,44	191,01	(*)	323,49	3745,02	16 / 64	130,37	(*)	(16 / 64)	246,87	(*)	(32 / 64)
BB11001 (4)	0,91	0,16	1,67	1,9	1,9	32 / 64	0,42	1,71	32 /32	0,34	1,77	16 / 32
BB11013 (5)	833,9	82,24	2114,79	733,34	860,37	32 / 64	74,74	1900,29	64 / 64	78,72	1770,39	16 / 64
BB11025 (4)	13,51	2,59	31,32	2,49	13,36	16 / 64	2,77	32	32 / 64	2,84	30,85	64 / 64
BB11029 (4)	4,62	1,35	10,19	18,45	18,46	64 / 64	1,78	11,02	64 / 64	1,37	10,24	64 / 64
BB11022 (4)	91,85	25,38	228,06	49,27	95,32	32 / 128	25,98	231,78	64 / 128	25,97	231,48	64 / 128
BB11021 (4)	16,6	4,55	40,35	23,51	23,52	64 / 128	7,16	40,54	64 / 128	7,03	40,21	64 / 128
BB11015 (4)	246,6	36,02	585,05	65	252,97	32 / 128	21,32	596,39	32 / 128	36,63	589,13	64 / 128
BB11012 (4)	104,57	18,67	265,16	157,08	161,32	64 / 128	21,46	266,38	128 / 128	72,14	263,56	128 / 128
BB12021 (5)	23,74	3,62	83,81	60,25	61,87	16 / 32	4,34	83,99	32 / 32	5,65	85,14	32 / 32
BB12020 (4)	0,41	0,09	0,94	0,73	0,74	16 / 32	0,14	0,91	16 / 32	0,17	0,9	32 / 32
BB12040 (5)	163,72	7,82	234,28	892,22	892,57	64 / 64	275,91	353,83	64 / 64	154,71	263,33	64 / 64
BB12025 (4)	103,04	23,16	282,77	105,56	107,08	64 / 128	23,37	272,45	64 / 128	24,15	282,36	64 / 128
BB12006 (4)	3,63	0,72	6,5	5,96	5,97	32 / 64	1,16	6,79	32 / 64	0,91	6,48	32 /64

Tabelle 6.2: Laufzeitübersicht für lineare Lückenstrafen und paarweise Alignments.

Instanz	UniDir-A*	BiDir-A*		UniDir-REACH			BiDir-REACH			BiDir-MAX		
		Opt	Beweis	Opt	Beweis	k	Opt	Beweis	k	Opt	Beweis	k
1ubi (4)	3,07	1,2	6,09	3,3	3,3	32 / 64	1,22	6,01	32 / 64	1,22	6,02	32 / 64
1wit (5)	35,95	8,9	112,03	30,84	34,65	16 / 32	9,64	113,24	32 / 64	9,78	113,5	32 / 64
3cyr (4)	0,86	1,48	1,77	1,17	1,17	32 /32	1,64	1,85	32 /32	1,53	1,83	32 /32
1pfc (5)	3,44	4,73	6,3	3,55	3,99	8 / 32	4,91	6,59	8 / 32	5,2	6,44	16 / 16
1fmb (4)	0,6	1,32	1,32	0,69	0,69	8 / 8	1,33	1,33	8 / 8	1,34	1,35	8 / 8
1fkj (5)	2,02	3,63	3,86	6,55	6,57	16 / 16	3,83	4	16 / 16	4,66	4,74	16 / 16
3grs (4)	72,49	24,33	172,87	46,78	74,45	32 / 128	28,73	171,92	64 / 128	24,08	171,98	64 / 128
1sbp (5)	/	/	/	(*)	(*)	(32 / 64)	(*)	(*)	(64 / 64)	(*)	(*)	(32 / 32)
1ad2 (4)	5,94	11,46	11,62	7,09	7,09	32 / 32	11,99	12,09	32 /32	12	12,1	32 / 32
2cba (5)	1539,27	173,2	5321,76	1430,62	1814,75	32 / 64	173,6	5369,15	32 / 128	174,2	5386,95	64 / 128
1zin (4)	6,43	12,49	12,58	6,53	6,54	8 / 16	12,47	12,58	8 / 32	12,52	12,62	8 / 16
1amk (5)	25,93	49,22	50,31	38,52	38,56	16 / 32	50,12	52,19	32 / 32	50,81	52,99	16 / 32
2myr (4)	4034,86	776,2	(*)	218,47	4122,25	32 / 256	715,91	(*)	(128 / 256)	312,81	(*)	(32 / 256)
1pamA (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(128 / 128)	(*)	(*)	(16 / 16)
1ac5 (4)	117,63	119,62	231,38	229,47	229,64	64 / 128	183,82	248,55	64 / 128	190,15	241,24	64 / 128
2ack (5)	/	(*)	(*)	(*)	(*)	(32 / 64)	(*)	(*)	(64 / 64)	(*)	(*)	(32 / 32)
1ad3 (4)	55,79	106,84	107,93	68,79	69,88	32 / 64	109,87	110,76	32 / 64	111,92	112,52	32 / 64
1rthA (5)	1640,44	560,37	2831,83	540,58	1571,54	16 / 64	559,2	2790,22	16 / 64	604,13	2817,38	32 / 64
BB11001 (4)	0,63	0,93	1,36	1,59	1,59	32 /32	1,23	1,45	32 /32	1,1	1,51	16 / 32
BB11013 (5)	149,7	12,12	424,16	163,21	164,35	32 / 64	12,82	420,05	32 / 64	13,26	437,37	32 / 64
BB11025 (4)	5,49	1,56	11,57	2,39	5,75	16 / 64	1,64	11,96	32 / 64	1,55	11,41	64 / 64
BB11029 (4)	2,01	1,74	3,48	12,14	12,15	64 / 64	4,76	5,5	64 / 64	4,23	5,11	64 / 64
BB11022 (4)	41,46	13,05	106,46	33,75	44,95	32 / 128	12,84	104,71	64 / 128	12,69	104,4	64 / 128
BB11021 (4)	6,15	3,31	10,76	11,85	11,86	64 / 64	4,91	10,95	64 / 64	4,64	10,83	64 / 64
BB11015 (4)	76,59	46,19	173,48	62,48	77,26	32 / 128	49,26	175,09	64 / 128	47,12	173,4	32 / 64
BB11012 (4)	57,54	66,73	117,73	103,26	103,58	64 / 128	71,97	119,03	64 / 128	95,28	124,63	64 / 128
BB12021 (5)	9,08	4,42	24,1	49,19	49,66	16 / 32	4,58	24,82	16 / 32	5,65	25,02	16 / 32
BB12020 (4)	1,39	2,58	2,74	1,73	1,73	16 / 32	2,65	2,81	16 / 32	2,68	2,8	32 / 32
BB12040 (5)	39,92	9,3	49,38	504,07	504,25	64 / 64	169,7	178,98	64 / 64	169,13	183,44	64 / 64
BB12025 (4)	38,04	18,3	98,43	44,84	44,94	64 / 128	18,99	97,8	64 / 128	22,4	98,94	64 / 128
BB12006 (4)	9,44	16,27	17,78	10,9	10,9	32 / 32	16,56	17,81	32 / 32	16,35	17,82	32 / 32

Tabelle 6.3: Laufzeitübersicht für lineare Lückenstrafen und dreifache Alignments.

Instanz	UniDir-A*	BiDir-A*		UniDir-REACH			BiDir-REACH			BiDir-MAX			Oma	
		Opt	Beweis	Opt	Beweis	k	Opt	Beweis	k	Opt	Beweis	k	Opt	Beweis
1ubi (4)	252,27	44,63	872,05	23,17	264,71	16 / 64	44,66	872,13	64 / 64	44,96	858,63	64 / 64	15,71	15,71
1wit (5)	/	1302,54	(*)	1293,72	(*)	(16 / 32)	1176,18	(*)	(32 / 64)	1196,25	(*)	(32 / 64)	39,17	39,17
3cyr (4)	35,55	8,13	152,28	13,59	36,85	16 / 64	8,43	153,57	32 / 64	8,66	153,92	32 / 64	0,67	1,58
1pfc (5)	942,26	89,92	4426,97	638,18	1054,82	16 / 64	94,15	4494,6	32 / 64	109,32	4934,43	32 / 64	6,51	6,51
1fmb (4)	0,82	1,34	1,93	0,98	0,98	8 / 16	1,55	1,99	8 / 16	1,45	1,99	8 / 16	0,3	0,53
1fkj (5)	470,81	17,91	1611,25	427,09	490,52	16 / 32	19,22	1619,8	32 / 64	18,67	1616,01	32 / 64	0,79	0,79
3grs (4)	/	4456,3	(*)	905,51	(*)	(32 / 64)	4713,49	(*)	(128 / 128)	4528,99	(*)	(128 / 128)	1335,41	1335,41
1sbp (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(64 / 64)	(*)	(*)	(32 / 64)	(*)	(*)
1ad2 (4)	341,48	59,32	863,09	287,22	339,27	32 / 64	53,32	938,81	64 / 64	56,44	949,82	64 / 64	1,26	2,94
2cba (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(32 / 32)	(*)	(*)	(32 / 32)	(*)	(*)
1zin (4)	20,92	14,36	81,97	16,28	21,75	16 / 64	15,85	83,94	32 / 64	16,35	83,38	32 / 64	0,7	1,55
1amk (5)	1247,19	83,14	3130,22	981,34	1456,43	16 / 64	76,81	3530,14	32 / 64	79,01	3688,83	32 / 64	1,48	3,02
2myr (4)	/	/	/	(*)	(*)	(32 / 64)	(*)	(*)	(64 / 64)	(*)	(*)	(32 / 64)	(*)	(*)
1pamA (5)	/	/	/	(*)	(*)	(8 / 16)	(*)	(*)	(16 / 16)	(*)	(*)	(16 / 16)	(*)	(*)
1ac5 (4)	/	/	/	(*)	(*)	(32 / 64)	(*)	(*)	(32 / 64)	(*)	(*)	(64 / 64)	550,29	2941,49
2ack (5)	/	/	/	(*)	(*)	(8 / 16)	(*)	(*)	(16 / 16)	(*)	(*)	(8 / 16)	(*)	(*)
1ad3 (4)	770,73	162,55	2144,44	1161,44	1163,46	64 / 128	173,18	2184,41	64 / 128	167,52	2172,54	64 / 128	5,63	5,63
1rthA (5)	/	/	/	7053,51	(*)	(16 / 32)	4632,98	(*)	(64 / 64)	6785,59	(*)	(64 / 64)	1228,84	1228,84
BB11001 (4)	34,26	5,18	99,85	40,97	42,2	32 / 64	6,96	100,58	32 / 64	6,3	101,74	32 / 64	0,43	0,85
BB11013 (5)	/	/	/	(*)	(*)	(16 / 32)	(*)	(*)	(64 / 64)	(*)	(*)	(64 / 64)	(*)	(*)
BB11025 (4)	2157,11	396,51	5361,98	335,23	2477,86	32 / 128	412,82	5367,04	128 / 128	410,86	5466,55	128 / 128	389,52	389,52
BB11029 (4)	1689,5	360,89	4776,76	2050,26	2129,98	64 / 128	337,47	4491,17	128 / 128	332,6	4327,44	128 / 128	83,27	83,27
BB11022 (4)	/	(*)	(*)	(*)	(*)	(64 / 128)	(*)	(*)	(64 / 256)	(*)	(*)	(128 / 128)	(*)	(*)
BB11021 (4)	3290,87	584,94	(*)	455,05	3324,28	32 / 128	592,08	(*)	(128 / 128)	644,38	(*)	(128 / 128)	606,13	606,13
BB11015 (4)	/	2940,84	(*)	1335,47	(*)	(32 / 64)	3121,12	(*)	(64 / 64)	2999,49	(*)	(64 / 64)	345,37	345,37
BB11012 (4)	/	2866,25	(*)	(*)	(*)	(32 / 64)	2675,29	(*)	(64 / 64)	3039	(*)	(128 / 128)	2,56	42,84
BB12021 (5)	/	1535,4	(*)	(*)	(*)	(8 / 16)	1562,07	(*)	(32 / 32)	1727,79	(*)	(32 / 32)	64,66	64,66
BB12020 (4)	210,31	22,66	605,44	32,77	210,72	16 / 64	22,88	607,37	32 / 64	15,09	610,68	32 / 64	0,85	1,83
BB12040 (5)	/	3249,47	(*)	(*)	(*)	(16 / 32)	3179,1	(*)	(64 / 64)	3211,73	(*)	(128 / 128)	355,97	355,97
BB12025 (4)	/	2621,68	(*)	8037,79	(*)	(64 / 128)	2521,3	(*)	(64 / 128)	2797,52	(*)	(128 / 128)	1094,24	1094,24
BB12006 (4)	616,24	67,66	1588,81	440,22	665,79	32 / 64	68,19	1594,21	64 / 64	68,06	1651,42	64 / 64	4,68	4,68

Tabelle 6.4: Laufzeitübersicht für quasi-affine Lückenstrafen.

wurde in hellgrau bzw. dunkelgrau schattiert. Dabei wurde der UniDir-A* nicht berücksichtigt, da es sich hierbei um keinen progressiven Algorithmus handelt.

Des Weiteren wurde bei der bidirektionalen Suche die Alternierungsstrategie minQ verwendet. Aufgrund der Ergebnisse aus Abschnitt 6.3.1.4 haben wir bei der bidirektionalen Variante Testläufe mit MAX- und REACH-Metrik gegenübergestellt. Neben den benötigten Laufzeiten wurde jeweils die Breite des k-Bandes für die Optimumsberechnung und den Optimalitätsbeweis notiert. Für Instanzen, die aufgrund des Speicherbedarfs vorzeitig abgebrochen wurden, haben wir in Klammern angeben, bei welcher k-Band-Breite das bisherige Optimum gefunden wurde und in welcher Iteration die Berechnungen terminierten. Wir können hierbei – unabhängig von der verwendeten Lückenstrafe – festhalten, dass 32 und 64 typische k-Band-Breiten sind. Die maximale Breite, die in unseren Testinstanzen vorkam, war 256. Dabei handelt es sich insbesondere um Sequenzen mit sehr großen Längenunterschieden (z. B. 60 – 205 Zeichen bei `BB11022`) und / oder mehr als 400 Zeichen (z. B. `2myr`). In der Regel wird die maximale k-Band-Breite nicht erreicht.

Es fällt auf, dass der unidirektionale k-Band-Algorithmus bei vielen Instanzen das Optimum in einer früheren Iteration findet als der bidirektionale k-Band-Algorithmus. Eine Ursache hierfür ist sicherlich der vorzeitige Iterationsabbruch bei bidirektionalen Variante. In vielen Fällen kann diese Differenz jedoch auch mit den unterschiedlichen unteren Schranken für Vorwärts- und Rückwärtssuche begründet werden. Es kann passieren, dass nur die Rückwärtssuche das k-Band verlässt und dadurch bei der bidirektionalen Suche mehr Iterationen notwendig sind. Bei den linearen Lückenstrafen ist zusätzlich zu erkennen, dass die Anzahl benötigter Iterationen bei dreifachen Alignments häufig niedriger ausfallen (z. B. `1pfc`, `1fkj`, `2cba`, `BB11013`). Dies verdeutlicht den Vorteil der dreifachen Alignments gegenüber den paarweisen Alignments.

Wir können insgesamt zusammenfassen, dass die bidirektionale A*-Suche in der Regel eher das Optimum findet als die unidirektionale Suche. Bei der Beweiszeit zeigt hingegen die unidirektionale A*-Suche das bessere Verhalten. Beide Beobachtungen gelten sowohl für die Varianten mit als auch für die ohne k-Band. Es gibt allerdings auch vereinzelte Ausnahmen. Besonders auffällig ist hierbei die Instanz `2myr`, bei der die bidirektionale Suche deutlich schlechter abschneidet. Hier liegt die Vermutung nahe, dass die unteren Schranken für Vorwärts- und Rückwärtssuche in diesem Fall sehr unterschiedlich sind und sich somit die beiden Suchrichtungen erst sehr spät treffen. Dies kann auch damit zusammenhängen, dass die Längendifferenz bei dieser Instanz recht hoch ist (340 – 474 Zeichen).

Für lineare Lückenstrafen zeigt die bidirektionale A*-Suche ohne k-Band das beste Laufzeitverhalten bzgl. der Optimierzeit. Ein großer Nachteil dieser Variante ist jedoch, dass sie bei quasi-affinen Lückenstrafen häufig keine zulässige Lösung berechnen konnte. Dahingegen haben die k-Band-Algorithmen in allen Fällen eine Lösung ausgegeben. Zudem ist der Laufzeit-Unterschied zwischen den Algorithmen mit und ohne k-Band bei quasi-affinen Lückenstrafen nicht so deutlich wie bei linearen Lückenstrafen.

Die Art der unteren Schranke für die A*-Suche hat entscheidenden Einfluss auf das Verhalten von unidirektionaler und bidirektionaler Suche. Bei einfachen Instan-

zen haben wir festgestellt, dass sich der höhere Aufwand für die Vorberechnungen beim Einsatz von dreifachen Alignments nicht lohnt. Da dieser Aufwand bei bidirektionaler Suche doppelt so hoch ist wie bei der unidirektionalen, macht sich dieser Effekt bei den bidirektionalen Algorithmen deutlicher bemerkbar. Die Laufzeit ist proportional zur Anzahl der endgültig markierten Knoten.

Bei den verschiedenen k-Band-Metriken gibt es für die bidirektionale Suche keinen eindeutigen Sieger. Beide Metriken zeigen ein ähnliches Laufzeitverhalten. Bei der unidirektionalen Suche zeigt die REACH-Metrik das beste Verhalten. Die Multi-k-Band-Strategie liefert zwar für einige Instanzen vielversprechende Ergebnisse, ist aber insgesamt nicht konkurrenzfähig.

Zusammenfassend ist für lineare Lückenstrafen sowie für einfache Instanzen bei quasi-affinen Lückenstrafen die Verwendung der bidirektionalen A*-Suche ohne k-Band zu empfehlen. Für schwere Instanzen sollte ein bidirektionaler k-Band-Algorithmus mit REACH- oder MAX-Metrik vorgezogen werden. Jedoch liefert der im Rahmen dieser Arbeit entwickelte Algorithmus schlechtere Ergebnisse als OMA [36].

6.3.2 Experimente mit Cytochrome C

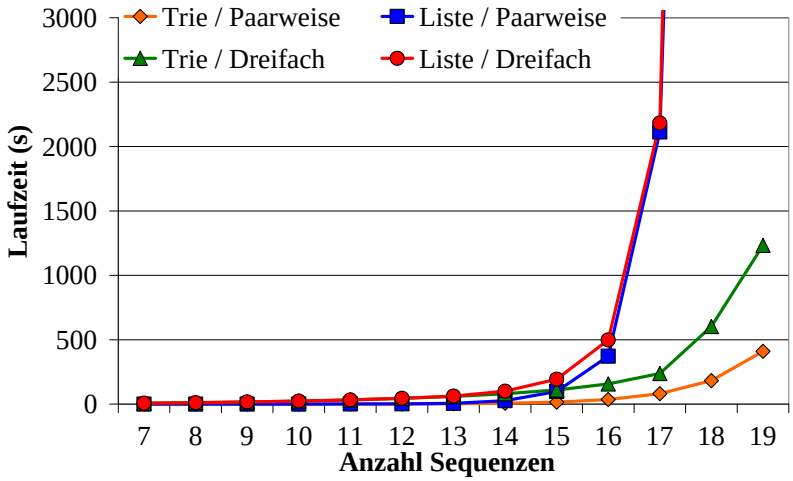
Wir werden nun analog zu Reinert et al. [36] unseren Algorithmus auf einer Familie von sehr ähnlichen Sequenzen (Cytochrome C) testen. Ziel dieser Untersuchungen ist, die Grenzen unseres Ansatzes bzgl. der Anzahl der Sequenzen sowie Abhängigkeiten von der Anzahl der Sequenzen zu ermitteln. Ein optimales Alignment der betrachteten Sequenzen kann aufgrund der großen Ähnlichkeit sehr einfach per Hand erzeugt werden.

Wir unterscheiden erneut zwischen linearen und quasi-affinen Lückenstrafen und analysieren zunächst den Einfluss der Datenstrukturen für markierte Knoten (Trie und Listendarstellung) und die Art der unteren Schranken (paarweise und dreifache Alignments). Anschließend untersuchen wir das Verhalten der verschiedenen Varianten des vorgestellten Algorithmus.

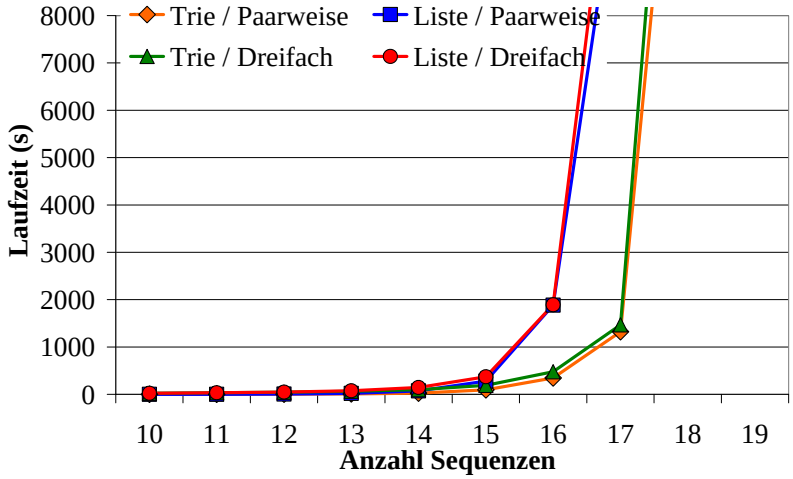
Datenstrukturen und untere Schranke

Wir vergleichen zunächst die vier Kombinationsmöglichkeiten für Datenstrukturen und untere Schranken. Die Abbildungen 6.19 und 6.20 zeigen die entsprechenden Laufzeiten in Abhängigkeit von der Anzahl der zu alignierenden Sequenzen. Dabei wird zwischen unidirektionaler und bidirektionaler Suche unterschieden. Wir haben uns bei der Darstellung jeweils auf die Optimierzeit beschränkt. Die Werte für die Beweiszeit zeigen ein vergleichbares Ergebnis.

Es ist zu erkennen, dass die Verwendung von Trie und paarweisen Alignments für alle Algorithmen zum besten Ergebnis führt. Dies gilt sowohl für lineare als auch für quasi-affine Lückenstrafen. Der Unterschied fällt jedoch bei linearen Lückenstrafen deutlicher aus. Bei quasi-affinen Lückenstrafen zeigt die Kombination von Trie und dreifachen Alignments ein fast identisches Verhalten.



(a) Lineare Lückenstrafe

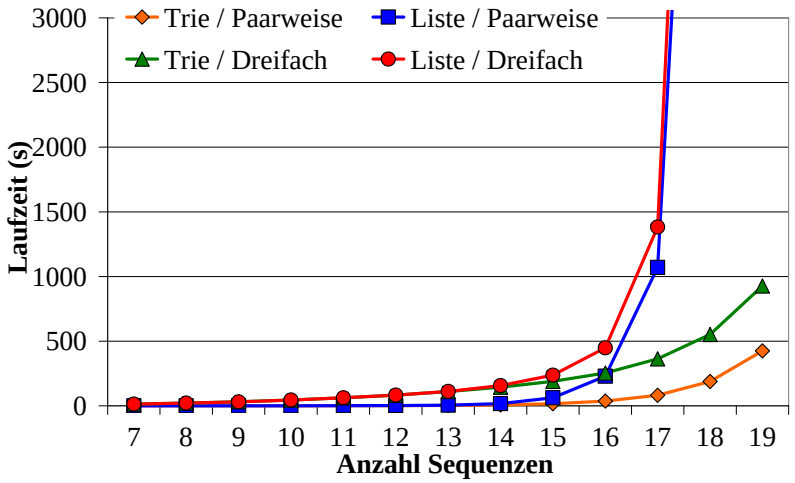


(b) Quasi-affine Lückenstrafe

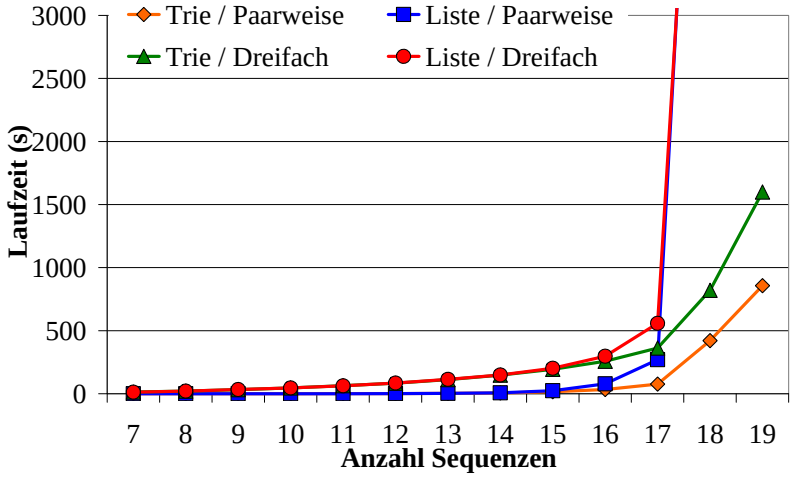
Abbildung 6.19: Laufzeit von UniDir-REACH auf n Cytochrome C Sequenzen.

Wir betrachten zunächst die linearen Lückenstrafen näher. Bis zu einer Anzahl von 15 oder 16 Sequenzen hat die Wahl der Datenstruktur kaum Einfluss auf die Laufzeit. Bei mehr Sequenzen steigt die Laufzeit bei Verwendung der Listendarstellung jedoch rasant an. Wir können also ähnlich wie bei den k-Band-Algorithmen beobachten, dass sich Listendarstellung ab einer gewissen Suchraumgröße nicht mehr lohnt (siehe Abschnitt 6.3.1.2).

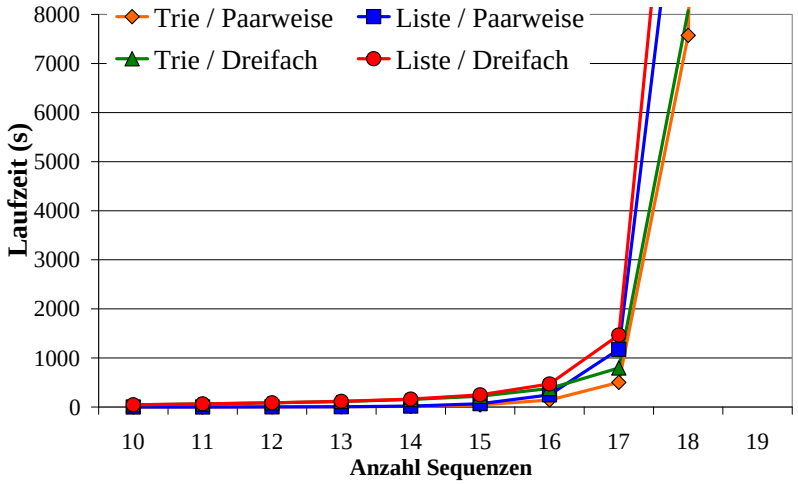
Des Weiteren beobachten wir, dass auf einer Menge von bis zu 15 oder 17 Sequenzen die Laufzeiten bei Verwendung von paarweisen Alignments stets geringer sind als die für dreifache Alignments. Werden mehr Sequenzen aligniert, schlägt die Kombination Trie / dreifache Alignments die Kombination Liste / paarweise Alignments. Die Auswirkungen der Listendarstellung auf die Laufzeit sind also bei steigender Sequenzanzahl höher als die der Vorberechnung. Die Wahl der unteren Schranke hat in diesem Fall kaum Einfluss auf das Laufzeitverhalten.



(a) BiDir-A*: Lineare Lückenstrafe



(b) BiDir-MAX: Lineare Lückenstrafe



(c) BiDir-MAX: Quasi-affine Lückenstrafe

Abbildung 6.20: Laufzeiten für BiDir-A* auf n Cytochrome C Sequenzen.

Bei Verwendung eines Tries hingegen liefern paarweise Alignments ein besseres Ergebnis. Dies liegt daran, dass die betrachteten Sequenzen sehr ähnlich sind und dadurch der Vorteil der dreifachen Alignments geringer ausfällt als bei den BAliBASE-Instanzen. Andererseits hängt die Laufzeit für die Vorberechnungen von der Anzahl der Sequenzen ab und wächst bei dreifachen Alignments deutlich schneller als bei paarweisen Alignments.

Bei quasi-affinen Lückenstrafen zeigt sich ein anderes Ergebnis. Die Laufzeit steigt hier bei 16–17 Sequenzen rasant an, unabhängig davon, welche Kombination verwendet wird. Dennoch ist zu erkennen, dass die Verwendung eines Tries Vorteile gegenüber der Listendarstellung hat. Ebenso liefern paarweise Alignments aus den oben genannten Gründen bessere Ergebnisse. Da hier jedoch insgesamt höhere Laufzeiten vorliegen, ist der Unterschied in den dargestellten Diagrammen nicht so deutlich erkennbar.

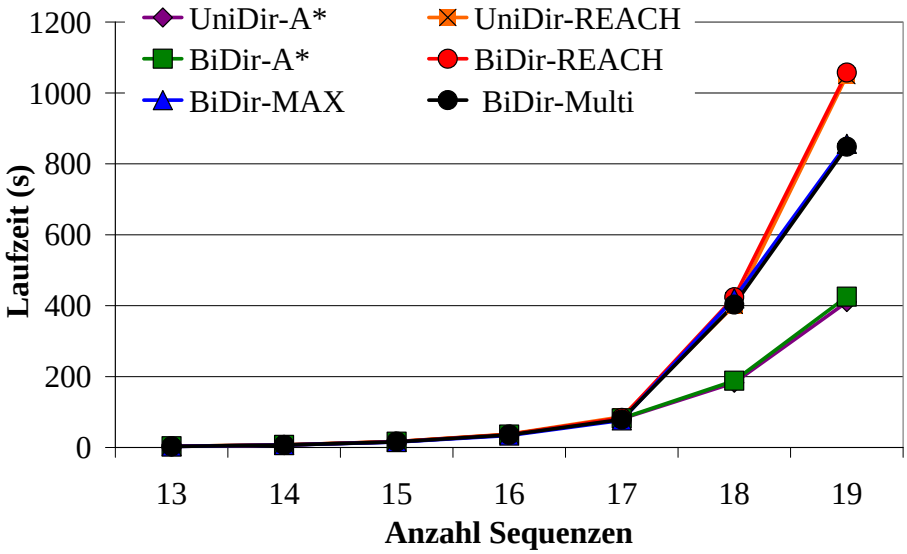
Ergebnisse verschiedener Algorithmen

Aufgrund der vorhergehenden Ergebnisse haben wir im Folgenden die verschiedenen vorgestellten Algorithmen mit der Kombination Trie / paarweise Alignments analysiert.

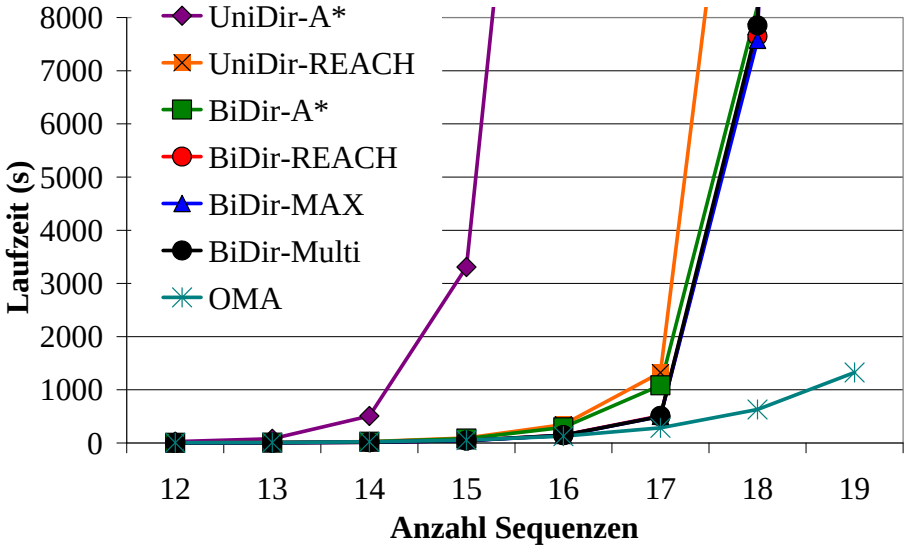
Für lineare Lückenstrafen (siehe Abbildung 6.21(a)) ist die Laufzeit der unterschiedlichen Algorithmen bis zu einer Menge von 17 Sequenzen nahezu identisch. Erst danach macht sich ein Unterschied bemerkbar, wobei die beiden Algorithmen ohne k-Band die besten Ergebnisse liefern. Bei den k-Band-Algorithmen schlagen die MAX-Metrik und die Multi-k-Band-Strategie die anderen Varianten. Das exponentielle Wachstum der Laufzeit ist bei allen Algorithmen deutlich erkennbar.

Bei Verwendung von quasi-affinen Lückenstrafen (siehe Abbildung 6.21(b)) erkennen wir, dass die k-Band-Algorithmen und die bidirektionale Suche ohne k-Band bis zu 18 Sequenzen optimal alignieren können. Dabei steigt die Laufzeit bei mehr als 17 Sequenzen rasant an. Bei der unidirektionalen A*-Suche ohne k-Band ist dies schon bei mehr als 15 Sequenzen der Fall. OMA ist in der Lage, 19 Sequenzen optimal zu alignieren. Insgesamt liefern hier die bidirektionalen k-Band-Algorithmen bessere Ergebnisse als der unidirektionale k-Band-Algorithmus und die Varianten ohne k-Band. Die bidirektionalen k-Band-Algorithmen weisen dabei ein nahezu identisches Laufzeitverhalten auf, so dass sich die entsprechenden Linien im Diagramm überschneiden.

Tabelle 6.5 gibt einen Überblick über die Ergebnisse dieses Abschnitts. Hierbei wurden jeweils Laufzeit und Speicherbedarf für den besten Algorithmus der beiden Lückenstrafen ausgewählt. Diesen Ergebnissen sind die entsprechenden Werte von OMA gegenübergestellt. Da OMA keine Ausgabe des Speicherbedarfs vorsieht, haben wir hierfür die Resultate aus [36] übernommen. Da wir evtl. etwas andere Cytochrome-C-Sequenzen bzw. eine andere Reihenfolge ausgewählt haben, sind die Werte als Schätzung des tatsächlichen Speicherbedarfs zu sehen und sollen lediglich einen Orientierungswert darstellen.



(a) Lineare Lückenstrafe



(b) Quasi-affine Lückenstrafe

Abbildung 6.21: Laufzeit auf n Cytochrome C Sequenzen bei Verwendung von Trie und paarweisen Alignments.

	BiDir-A* (linear)			BiDir-MAX (quasi-affin)			Oma [36]		
n	Opt	Beweis	Speicher	Opt	Beweis	Speicher	Opt	Beweis	Speicher
2	0,01	0,01	0,1	0,01	0,01	3,23	0,24	0,34	6,7
3	0,01	0,01	0,34	0,01	0,01	4,31	0,26	0,42	6,7
4	0,01	0,01	0,6	0,01	0,02	7,94	0,31	0,55	6,7
5	0,01	0,01	1,0	0,02	0,02	14,17	0,33	0,69	7,7
6	0,03	0,03	1,64	0,02	0,02	26,16	0,42	0,94	8,8
7	0,04	0,05	2,73	0,05	0,06	45,42	0,51	1,28	9,1
8	0,07	0,08	4,72	0,08	0,15	84,64	0,67	1,81	10,5
9	0,15	0,18	8,5	0,18	0,38	126,21	0,98	2,81	12,2
10	0,31	0,36	15,93	0,38	1,08	155,30	1,52	4,73	14,6
11	0,68	0,79	30,83	0,86	3,37	187,65	2,58	8,8	19,1
12	1,48	1,69	61,01	2,03	8,38	223,4	4,77	17,93	31,4
13	3,25	3,67	122,58	4,99	21,76	306,42	9,6	38,83	45,8
14	7,3	8,16	248,5	13,63	84,39	483,83	21,77	85,72	84,2
15	16,46	18,02	506,39	43,7	219,21	852,03	53,32	192,93	149,5
16	36,5	40,09	1034,78	143,88	594,45	1608,13	127,61	434,5	298,8
17	82	89,08	2117,29	501,59	1787,77	3239,84	284,47	940,58	1051,1
18	188,27	202,99	4334,41	8293,35	11958,9	9577,44	631,32	2039,39	$<$ 2 GB
19	425,27	452,19	8873,44	$>$ 37300	$>$ 37300	$>$ 30 GB	1324,9	4199,15	$<$ 2 GB

Tabelle 6.5: Laufzeit (in s) und Speicherbedarf (in MB) für n Cytochrome C Sequenzen.

Die Ergebnisse dieses Abschnitts bestätigen die Resultate der BAliBASE-Instanzen. Für einfache Instanzen, d. h. in diesem Fall wenige Sequenzen oder lineare Lückenstrafen, stellt sich die bidirektionale A*-Suche ohne k-Band als bester Algorithmus heraus. Bei schwierigen Instanzen ist diese allerdings nicht immer in der Lage, das optimale Ergebnis zu berechnen. Hier liefern die vorgestellten bidirektionalen Varianten des k-Band-Algorithmus das beste Ergebnis.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neuer progressiver Algorithmus für das Multiple Sequence Alignment vorgestellt. Dieser basiert auf erfolgreichen Ansätzen zur Berechnung optimaler Alignments mit Hilfe der A*-Suche. Wir haben diese Methoden als Ausgangspunkt benutzt und mit einer k-Band-Heuristik zu einem progressiven Algorithmus erweitert. Für den resultierenden Algorithmus wurden verschiedene Varianten und Strategien vorgestellt und in einer umfassenden experimentellen Analyse miteinander verglichen. Abschließend haben wir die erzielten Ergebnisse dem zur Zeit besten progressiven Algorithmus für das Multiple Sequence Alignment (OMA, [36]) gegenübergestellt.

Wir haben festgestellt, dass die bidirektionale A*-Suche unter dem Aspekt, möglichst schnell eine optimale Lösung zu finden, eine Verbesserung gegenüber der unidirektionalen A*-Suche darstellt. Es zeigt sich jedoch auch, dass sowohl mit einer einfachen unidirektionalen A*-Suche als auch mit einer einfachen bidirektionalen A*-Suche nicht für jede der betrachteten Instanzen eine zulässige Lösung berechnet werden kann. Dahingegen liefert der k-Band-Algorithmus für alle betrachteten Instanzen eine zulässige Lösung und zeigt insgesamt ein deutlich besseres Konvergenzverhalten als die einfache A*-Suche.

Während die bidirektionale A*-Suche ohne k-Band für einfache Instanzen das beste Laufzeitverhalten zeigt, lohnt sich der Einsatz der k-Band-Heuristik für schwierige Instanzen und insbesondere für quasi-affine Lückenstrafen. Die Laufzeit der k-Band-Algorithmen hängt dabei neben der Anzahl an Sequenzen, deren Länge und Ähnlichkeit, besonders von der Anzahl aufeinander folgender Lücken im optimalen Alignment ab, da diese durch das k-Band beschränkt wird. Ein großer Vorteil des k-Band-Algorithmus ist die Eigenschaft, dass die Breite des aktuell betrachteten k-Bandes einen Hinweis auf die Ähnlichkeit der betrachteten Sequenzen gibt.

Der vorgestellte Ansatz ist in der momentan vorliegenden Form leider nicht konkurrenzfähig zu OMA, da seine Optimier- und Beweiszeiten in der Regel über den entsprechenden Laufzeiten von OMA liegen. Dennoch sind wir der Meinung, dass der Ansatz vielversprechende Ergebnisse geliefert hat und noch Möglichkeiten für weitere Optimierungen bietet. Hierbei ist sind z. B. dynamische obere Schranken [29] oder die Face-Bounding-Heuristik [18, 29] zu nennen. Weitere Möglichkeiten für

Verbesserungen sind bessere Abbruchkriterien bei der bidirektionalen Suche sowie problemorientierte Initialisierungs- und Erweiterungsstrategien für das k-Band. Beispielsweise kann man zusätzlich zur oberen Schranke für die optimale Lösung eine untere Schranke berechnen und abhängig von der Differenz der beiden Schranken einen geeigneten Erweiterungsfaktor bestimmen.

Des Weiteren bieten Optimierungen bei der Berechnung der unteren Schranken bzw. bessere untere Schranken eine Optimierungsmöglichkeit für den Algorithmus, z. B. durch Verwendung des kubischen Algorithmus von Gotoh [16]. Eine Gewichtung der Sequenzen [36] liefert unter Umständen biologisch relevantere Alignments.

Viele Instanzen konnten aufgrund des hohen Speicherbedarfs von unserem Algorithmus nicht beweisbar optimal gelöst werden. Aus diesem Grund kann es sinnvoll sein, nicht mehr benötigte Knoten und Kanten zu löschen [18]. Ein Nebeneffekt dieser Modifikation wäre eine Beschleunigung von Such- und Einfügeoperationen auf den Datenstrukturen für markierte Knoten.

Für weitere Untersuchungen des vorgestellten Ansatzes ist evtl. ein Vergleich mit GSA [29] hilfreich. GSA ist ein Programm zur Berechnung optimaler multipler Alignments, das auf der A*-Suche basiert und verschiedene Techniken zur Verbesserung von oberen und unteren Schranken benutzt. Dabei unterstützt es sowohl die Verwendung von linearen als auch von quasi-affinen Lückenstrafen. Da uns keine Implementierung dieses Algorithmus vorlag, musste im Rahmen dieser Arbeit jedoch auf diese Analyse verzichtet werden.

Ein anderer Lösungsansatz für das Multiple Sequence Alignment wäre die Konstruktion eines hybriden Algorithmus aus OMA und der einfachen bidirektionalen A*-Suche mit Pruning. Momentan verwendet OMA lediglich eine unidirektionale A*-Suche. Die Untersuchungen im Rahmen dieser Arbeit haben jedoch gezeigt, dass die bidirektionale A*-Suche Vorteile gegenüber der unidirektionalen Suche hat.

Zusammenfassend können wir sagen, dass der k-Band-Algorithmus zwar nicht so gute Ergebnisse erzielt wie OMA, aber dennoch einige vielversprechenden Resultate liefert und somit als Ansatzpunkt für weitere Entwicklungen in Betracht gezogen werden sollte.

Anhang A

Molekularbiologische Grundlagen

Die Molekularbiologie beschäftigt sich als ein Teilgebiet der Biologie mit der Erforschung biochemischer Moleküle und Prozesse. Insbesondere befasst sie sich mit der Struktur, der Funktion und der Interaktion zwischen bestimmten Biomolekülen wie z. B. DNS, RNS und Proteinen.

Um molekularbiologische Fragestellungen mit Hilfe der Informatik bearbeiten zu können, muss ein geeignetes Modell für jedes dieser Biomoleküle erstellt werden. In den beiden folgenden Abschnitten werden zunächst einige biologische Grundlagen erläutert und anschließend geeignete Modelle eingeführt [4, 22]. Abschnitt A.1 beginnt mit einer Einführung in die Nukleinsäuren DNS und RNS, Abschnitt A.2 widmet sich dem Aufbau von Proteinen. Abschließend wird das zentrale Dogma der Molekularbiologie beschrieben. Für ausführlichere biologische Hintergründe wird auf weiterführende Literatur verwiesen, z. B. [2].

A.1 Nukleinsäuren

Nukleinsäuren dienen vor allem als Informationsspeicher der Zelle. Sie spielen aber auch eine Rolle bei der Übertragung von Erbinformationen von einer Generation an die nächste und stellen Baupläne für Proteine (siehe Abschnitt A.2) bereit. Ihre prominentesten Vertreter sind die *Desoxyribonukleinsäure* (DNS, engl.: Deoxyribonucleic Acid bzw. DNA) als Speicher der Erbinformation sowie *Ribonukleinsäure* (RNS, engl.: Ribonucleic Acid bzw. RNA), die diese Erbinformation zur Verarbeitung weitergibt – z. B. bei der Produktion von Proteinen (siehe Abschnitt A.3).

Nukleinsäuren sind aus einer Kette von *Nukleotiden* aufgebaut, wobei ein Nukleotid aus einer Phosphorgruppe (P), einem Zucker (Z) und einer Nukleobase (siehe Tabelle A.1) besteht. Dabei stellen die Nukleobasen ein charakteristisches Unterscheidungsmerkmal für verschiedene Nukleotide und damit für Nukleinsäuren dar.

Es werden fünf verschiedene Nukleobasen unterschieden: Adenin, Guanin, Cytosin, Thymin und Uracil. Um Nukleinsäuren als Strings kodieren zu können, wurde eine entsprechende Kodierung entwickelt. Diese wurde vom Nomenklaturkomitee der International Union of Biochemistry and Molecular Biology vorgeschlagen und

(a) Nukleobasen

Base	Kodierung
Adenin	A
Cytosin	C
Guanin	G
Thymin	T
Uracil	U

(b) Zusätzliche Symbole

Beschreibung	Kodierung
Nicht Adenin	B
Nicht Cytosin	D
Nicht Guanin	H
Nicht Thymin	V
Guanin oder Thymin (Ketogruppe)	K
Adenin oder Cytosin (Aminogruppe)	M
Adenin oder Guanin (Purin)	R
Cytosin oder Guanin (Schwache Wasserstoffbrückenbindungen)	S
Adenin oder Thymin (Starke Wasserstoffbrückenbindungen)	W
Cytosin oder Thymin (Pyrimidin)	Y
Adenin, Cytosin, Guanin oder Thymin	N oder X

Tabelle A.1: Nukleobasen und ihre Kodierung.

durch die IUPAC (International Union of Pure and Applied Chemistry[1]) standardisiert [6]. Sie enthält neben Symbolen für die Nukleobasen zusätzliche Zeichen, mit deren Hilfe Mehrdeutigkeiten kodiert werden können. Die vollständige Kodierung ist in Tabelle A.1 dargestellt.

A.1.1 DNS

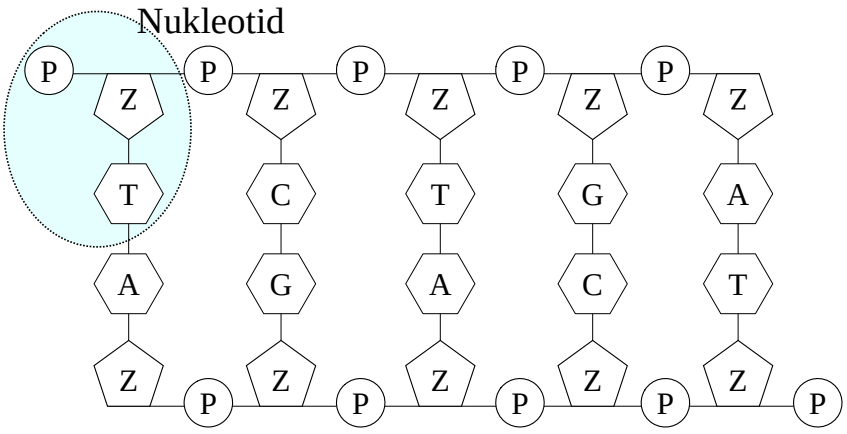
Die DNS besteht aus zwei Nukleotidketten und hat die Form einer rechtsgängigen Doppelhelix. Diese ist in Abbildung A.1 dargestellt. Dabei zeigt Abbildung A.1(a) eine schematische Darstellung der beiden Nukleotidketten und Abbildung A.1(b) die Doppelhelix-Struktur der DNS.

In der DNS kommen vier verschiedene Basen vor: Adenin, Cytosin, Guanin und Thymin. Eine Kette aus Nukleotiden mit diesen Basen wird als DNS-Strang bezeichnet. Bestimmte Basen können mit anderen in Wechselwirkung treten – in der Regel Adenin mit Thymin sowie Cytosin mit Guanin (*Watson-Crick-Paare*). Daher können sich zwei (komplementäre) DNS-Stränge aneinander anlagern. Zum Vergleich verschiedener DNS-Sequenzen genügt es, einen der beiden Einzelstränge zu betrachten, da die Basenreihenfolge des zweiten Stranges direkt aus dem ersten ableitbar ist.

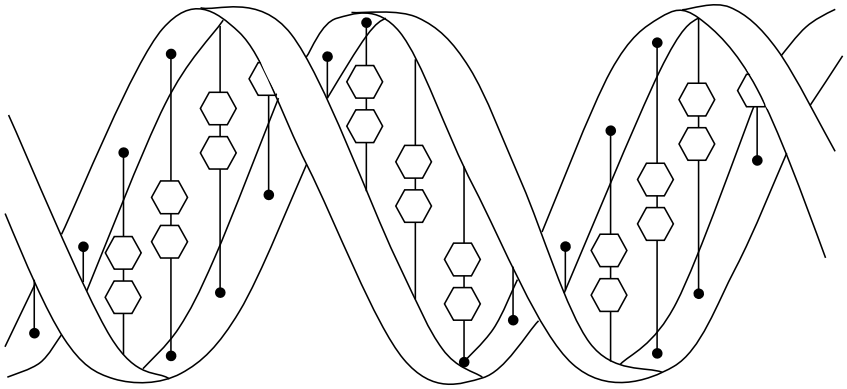
A.1.2 RNS

Im Gegensatz zur DNS tritt bei der RNS die Base Uracil statt Thymin auf. Außerdem enthält sie eine andere Zuckerart und liegt meist als Einzelstrang vor.

[1] Die IUPAC wurde 1919 von Chemikern aus Industrie und Universitäten mit dem Ziel der internationalen Kommunikationsförderung gegründet. Sie ist u. a. für Standardisierungen der Chemie (Nomenklatur, Symbole, Terminologie, Methoden, ...) zuständig.



(a) Schematische Darstellung



(b) Doppelhelix-Struktur

Abbildung A.1: Aufbau der Desoxyribonukleinsäure (DNS) [4].

A.2 Proteine

Proteine sind die wichtigsten Makromoleküle und steuern die meisten Abläufe im Organismus. Sie spielen eine wichtige Rolle beim Stoffwechsel, bestimmen die Zellstruktur und dienen als Baustoff (z. B. für Haare), Transportstoff für körperwichtige Substanzen wie Hämoglobin und Transferrin, zur Infektionsabwehr und als Signalstoff.

Proteine sind aus proteinogenen (proteinerzeugenden) Aminosäuren aufgebaut, die durch Peptidbindungen zu Ketten verbunden werden. Eine solche Aminosäurekette wird auch als *Primärstruktur* des Proteins bezeichnet. Neben dieser werden noch drei weitere Strukturen betrachtet: Die *Sekundärstruktur* beschreibt die räumliche Anordnung einzelner Teilbereiche der Aminosäurekette, die *Tertiärstruktur* deren vollständige räumliche Struktur. Mit *Quartärstruktur* wird schließlich der Zusammenschluss mehrerer Proteine zu einem funktionellen Komplex bezeichnet.

Um verschiedene Proteine zu vergleichen, wird häufig nur die Primärstruktur des Proteins betrachtet. In diesem Fall kann man ein Protein einfach als String kodieren,

(a) Kanonische Aminosäuren

Aminosäure	Kodierung	
Alanin	Ala	A
Arginin	Arg	R
Asparagin	Asn	N
Asparaginsäure	Asp	D
Cystein	Cys	C
Glutamin	Gln	Q
Glutaminsäure	Glu	E
Glycin	Gly	G
Histidin	His	H
Isoleucin	Ile	I
Leucin	Leu	L
Lysin	Lys	K
Methionin	Met	M
Phenylalanin	Phe	F
Prolin	Pro	P
Serin	Ser	S
Threonin	Thr	T
Tryptophan	Trp	W
Tyrosin	Tyr	Y
Valin	Val	V

(b) Nicht-Kanonische Aminosäuren

Aminosäure	Kodierung	
Pyrrolysin	Pyl	O
Selenocystein	Sec	U
Selenomethionin	SeMet	?

(c) Zusätzliche Symbole

Aminosäure	Kodierung	
Asparagin oder Asparaginsäure	Asx	B
Glutamin oder Glutaminsäure	Glx	Z
Beliebige Aminosäure	Unk	X

Tabelle A.2: Proteinogene Aminosäuren und ihre Kodierung.

dessen Zeichen die Aminosäuren der Kette repräsentieren.

Bisher sind 23 proteinogene und ca. 250 nicht-proteinogene Aminosäuren bekannt. Proteinogene Aminosäuren sind diejenigen, aus denen die Proteine von Lebewesen aufgebaut sind. Beim Menschen selbst kommen die 20 seit langem bekannten *Standardaminosäuren* sowie das 1986 entdeckte *Selenocystein* zum Einsatz.

Tabelle A.2 gibt einen Überblick über die verschiedenen proteinogenen Aminosäuren und deren Kodierung. Diese enthält auch zusätzliche Symbole mit deren Hilfe Mehrdeutigkeiten modelliert werden können. So kann es z. B. vorkommen, dass bei der chemischen Analyse nicht eindeutig zwischen Asparagin oder Asparaginsäure unterschieden werden kann. Das Zeichen X wird häufig für Selenocystein verwendet, falls eine Substitutionsmatrix zum Einsatz kommt, die diese Aminosäure noch nicht berücksichtigt. Die Kodierung wurde von Dr. Margaret Oakley Dayhoff entwickelt und durch die IUPAC standardisiert [5].

A.3 Der genetische Informationsfluss

Zum Schluss dieses Kapitels wird beschrieben, auf welche Weise die in der DNS gespeicherte Erbinformation in den Bau von Proteinen umgesetzt werden kann, d. h.

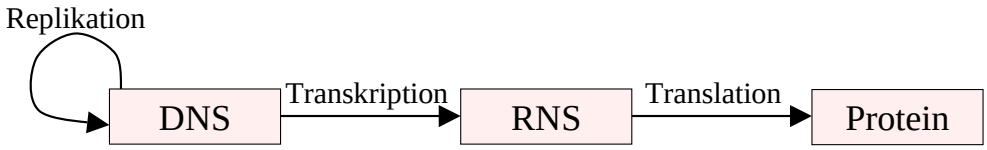


Abbildung A.2: Das zentrale Dogma der Molekularbiologie.

vererbt wird. Hierzu müssen zunächst einige wichtige Begriffe geklärt werden.

Ein *Gen* ist ein DNS-Abschnitt, der biologische Erbinformation enthält und ein Polypeptid[2] kodiert. Dabei dienen nur bestimmte Abschnitte des jeweiligen Gens – die sog. *Exons* – zur Kodierung der Aminosäuresequenz. Die nicht-relevanten Teile des Gens werden als *Introns* bezeichnet.

Auf einem DNS-Molekül befindet sich eine Vielzahl solcher Gene. Die Gesamtheit des genetischen Materials einer Zelle nennt man *Genom*.

Die DNS liegt nur im Zellkern vor und kann diesen auch nicht verlassen. Proteine werden jedoch außerhalb des Zellkerns in den sog. *Ribosomen* hergestellt. Die RNS übernimmt dabei eine Überträgerrolle für die genetische Information, da sie sowohl innerhalb als auch außerhalb des Zellkerns vorkommt.

Bei der Proteinbiosynthese wird zunächst eine Kopie des entsprechenden Gens erzeugt. Dieser Schritt wird als *Transkription* bezeichnet. Dabei werden keine DNS-Nukleotide sondern RNS-Nukleotide verwendet. Die resultierende RNS heißt *messenger RNS* oder kurz *mRNS*. Außerdem werden die Introns des Gens durch einen mit *Splicing* bezeichneten Prozess aus der Kopie entfernt.

Mit Hilfe der mRNS kann bei der sog. *Translation* ein Protein hergestellt werden. Bei diesem Schritt werden jeweils drei aufeinander folgende Basen (*Codon*) als Kodierung für eine Aminosäure aufgefasst.

Zusammenfassend lässt sich der genetische Informationsfluss wie folgt beschreiben:

1. Die DNS speichert die Erbinformation und vervielfacht sie durch Replikation.
2. Durch Transkription wird diese Information ausgelesen und als mRNS weitergegeben.
3. Bei der Translation wird mit Hilfe der mRNS die genetische Information in Proteine übersetzt.

Der genetische Informationsfluss verläuft demnach eindeutig von der DNS über die RNS zu den Proteinen. Es können keine Informationen von einem Protein zu einem anderen Protein oder von einem Protein zurück zu Nukleinsäuren übertragen werden. Dies wird als zentrales Dogma der Molekularbiologie bezeichnet und ist in Abbildung A.2 visualisiert.

[2]Peptide sind Aminosäureketten und unterscheiden sich von Proteinen lediglich in ihrer Größe. Peptide haben in der Regel eine maximale Länge von 100 Aminosäuren.

Anhang B

Substitutionsmatrizen für Proteine

Beim Vergleich von Proteinsequenzen ist es wichtig, die Mutationswahrscheinlichkeiten zwischen den verschiedenen beteiligten Aminosäuren einzubeziehen, da diese sehr unterschiedlich sein können. Dies ist dadurch zu erklären, dass sich Aminosäuren in ihren chemischen Eigenschaften erheblich unterscheiden können. Falls nun eine Aminosäure durch Mutation in eine Aminosäure mit anderen chemischen Eigenschaften umgewandelt wird, kann es sein, dass das entstehende Protein nicht mehr funktionsfähig und damit der entsprechende Organismus nicht mehr lebensfähig ist. Das Protein kann demnach nicht weitervererbt werden und eine derartige Mutation ist sehr unwahrscheinlich. Die Mutation einer einzelnen Aminosäure in einem Protein wird auch Punktmutation genannt.

Substitutionsmatrizen berücksichtigen diesen Umstand, indem für jede mögliche Substitution eine unterschiedliche Bewertung bestimmt wird. Man erhält also eine $|\Sigma| \times |\Sigma|$-Matrix, wobei in der Regel $|\Sigma| = 20$ gilt (siehe Anhang A). Die am häufigsten gebrauchten Formen der Substitutionsmatrizen sind PAM- [10] und BLOSUM-Matrizen [21], die im Folgenden näher erläutert werden.

B.1 PAM-Matrizen

PAM-Matrizen wurden 1978 von Margaret Dayhoff vorgestellt [10]. PAM steht hierbei für „Point Accepted Mutation“ oder „Percent Of Accepted Mutations“. Diese Matrizen werden anhand einer Ähnlichkeitsanalyse von eng verwandten Proteinen erstellt. Die Proteinsequenzen werden aligniert und die Anzahl der erfolgten Mutationen bestimmt. Hieraus lassen sich dann Mutationswahrscheinlichkeiten zwischen den verschiedenen Aminosäuren berechnen.

Basis der PAM-Matrizen ist die sog. *PAM-Einheit*. Zwei Sequenzen S_1 und S_2 sind eine PAM-Einheit voneinander entfernt, wenn S_2 aus S_1 durch eine Serie von akzeptierten Punktmutationen (ohne Einfügungen und Löschungen) entstanden sind und im Schnitt pro 100 Aminosäuren eine Punktmutation aufgetreten ist. „Akzeptiert“ heißt hierbei, dass das resultierende Protein funktionsfähig ist.

An einer Stelle können mehrere Punktmutationen auftreten. Außerdem kann es passieren, dass eine Stelle zu ihrer ursprünglichen Aminosäure zurückmutiert.

Daher ist eine Abweichung von einer PAM-Einheit nicht damit gleichzusetzen, dass sich die beteiligten Proteine in einem Prozent der Aminosäuren unterscheiden. Eine Divergenz von 60 PAM-Einheiten entspricht in etwa einer 60%igen Identität der Sequenzen, eine von 120 PAM-Einheiten noch einer Übereinstimmung von 40%.

Entsprechend der PAM-Einheit können verschiedene Varianten der PAM-Matrizen berechnet werden, die sich jeweils in ihrer PAM-Zahl unterscheiden, z. B. PAM-1, PAM-40, PAM-120, PAM-250. Die PAM-1-Matrix entspricht dabei den Ergebnissen der oben beschriebenen Ähnlichkeitsanalyse. Aufgrund der Annahme, dass wiederholte Mutationen nach demselben Schema erfolgen und an einer Stelle mehrere Mutationen auftreten können, werden alle anderen PAM-Matrizen aus der PAM-1-Matrix abgeleitet. Für Details hierzu sei auf [4, 32] verwiesen.

Die Wahl der am besten geeigneten Matrix ist abhängig von der evolutionären Distanz der gegebenen Sequenzen. Dabei ist eine PAM-n-Matrix optimal für den Vergleich von Sequenzen, die n PAM-Einheiten voneinander entfernt sind. PAM-Matrizen mit einer kleineren PAM-Zahl bieten sich also eher für ähnliche Sequenzen, die mit einer größeren PAM-Zahl für entfernter verwandte Sequenzen an.

In der Praxis ist die Wahl der optimalen PAM-Matrix meist schwierig, da in der Regel nicht bekannt ist, wie weit die gegebenen Sequenzen voneinander entfernt sind. Daher werden meistens Standardmatrizen verwendet. Bei PAM-Matrizen ist dies die PAM-250-Matrix (siehe Abbildung B.1). Sie ist gleichzeitig die höchste noch verwendete PAM-Matrix und entspricht einer Identität von ca. 20%. Bei höheren PAM-Zahlen kann man im Allgemeinen nicht mehr von „ähnlichen“ Sequenzen sprechen.

B.2 BLOSUM-Matrizen

BLOSUM-Matrizen (BLOcks SUbstitution Matrix) wurden 1992 von Steven Henikhoff und Jorja Henikoff als Alternative zu PAM-Matrizen eingeführt [21]. Motivation für die Entwicklung war die Beobachtung, dass PAM-Matrizen für entfernt verwandte Sequenzen keine guten Ergebnisse liefern. Dies liegt u. a. daran, dass das Zusammensetzen von Beobachtungen über einen kurzen Zeitraum (d. h. die Aneinanderreihung von PAM-1-Matrizen), keine gute Approximation für evolutionäre Änderungen über einen längeren Zeitraum liefern.

Um dieses Problem zu beheben, werden BLOSUM-Matrizen anhand einer Analyse entfernt verwandter Sequenzen bestimmt. Basis für die verwendeten Mutationswahrscheinlichkeiten ist hier die sog. *BLOCKS-Datenbank*, die Informationen über ähnliche Regionen in verwandten Proteinen enthält. Sie beinhaltet verschiedene „Blöcke“ aus bereits berechneten multiplen Alignments. Unter einem Block versteht man ein kurzes, zusammenhängendes, nicht durch Lücken unterbrochenes Intervall eines multiplen Alignments. Innerhalb von Blöcken sind die Sequenzen sehr ähnlich und besitzen dieselbe Länge. Für Blöcke in der Datenbank wird eine Mindestlänge vorausgesetzt. Außerdem nimmt man an, dass die Blöcke innerhalb verwandter Proteine von funktioneller Bedeutung sind. Für weiterführende Literatur sei erneut auf [4, 32] verwiesen.

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
A	15	19	17	17	21	16	18	19	18	19	18	17	16	17	19	16	16	17	23	20
C	19	5	22	22	21	20	20	19	22	23	22	21	20	22	21	17	19	19	25	17
D	17	22	13	14	23	16	16	19	17	21	20	15	18	15	18	17	17	19	24	21
E	17	22	14	13	22	17	16	19	17	20	19	16	18	15	18	17	17	19	24	21
F	21	21	23	22	8	22	19	16	22	15	17	21	22	22	21	20	20	18	17	10
G	16	20	16	17	22	12	19	20	19	21	20	17	18	18	20	16	17	18	24	22
H	18	20	16	16	19	19	11	19	17	19	19	15	17	14	15	18	18	19	20	17
I	19	19	19	19	16	20	19	12	19	15	15	19	19	19	19	18	17	13	22	18
K	18	22	17	17	22	19	17	19	12	20	17	16	18	16	14	17	17	19	20	21
L	19	23	21	20	15	21	19	15	20	11	13	20	20	19	20	20	19	15	19	18
M	18	22	20	19	17	20	19	15	17	13	11	19	19	18	17	19	18	15	21	19
N	17	21	15	16	21	17	15	19	16	20	19	15	18	16	17	16	17	19	21	19
P	16	20	18	18	22	18	17	19	18	20	19	18	11	17	17	16	17	18	23	22
Q	17	22	15	15	22	18	14	19	16	19	18	16	17	13	16	18	18	19	22	21
R	19	21	18	18	21	20	15	19	14	20	17	17	17	16	11	17	18	19	15	21
S	16	17	17	17	20	16	18	18	17	20	19	16	16	18	17	15	16	18	19	20
T	16	19	17	17	20	17	18	17	17	19	18	17	17	18	18	16	14	17	22	20
V	17	19	19	19	18	18	19	13	19	15	15	19	18	19	19	18	17	13	23	19
W	23	25	24	24	17	24	20	22	20	19	21	21	23	22	15	19	22	23	0	17
Y	20	17	21	21	10	22	17	18	21	18	19	19	22	21	21	20	20	19	17	7

Abbildung B.1: PAM-250-Matrix in der Distanzvariante.

Wie bei PAM-Matrizen gibt es auch für BLOSUM-Matrizen verschiedene Varianten. Bei BLOSUM-Matrizen entspricht die Kennzahl in etwa der zugehörigen Sequenzidentität. Eine BLOSUM-62-Matrix wird beispielsweise unter Verwendung der Blöcke in der Datenbank berechnet, deren zugehörige Proteine eine Identität von 62% oder weniger aufweisen. Demnach wird für BLOSUM-100-Matrizen die gesamte Datenbank benutzt.

BLOSUM-Matrizen mit hoher Kennzahl sind folglich eher für den Vergleich ähnlicher Sequenzen, die mit niedriger Kennzahl für entfernter verwandte Sequenzen geeignet. Als Standardmatrix wird häufig eine BLOSUM-62-Matrix verwendet.

B.3 Vergleich von PAM- und BLOSUM-Matrizen

Die Wahl zwischen PAM- und BLOSUM-Matrix hängt davon ab, welche Ziele beim Sequenzvergleich verfolgt werden. Der Matrixtyp sollte bei der Interpretation der Ergebnisse berücksichtigt werden.

PAM-Matrizen basieren auf einem expliziten Evolutionsmodell, d. h. sie werden direkt auf Basis eines Vergleichs eng verwandter Proteine bestimmt. Des Weiteren wird angenommen, dass Mutationen unabhängig von vorangehenden Mutationen sind (Markov-Prozess). BLOSUM-Matrizen hingegen basieren auf einem impliziten Evolutionsmodell.

PAM100	$\sim$	BLOSUM90
PAM120	$\sim$	BLOSUM80
PAM160	$\sim$	BLOSUM60
PAM200	$\sim$	BLOSUM52
PAM250	$\sim$	BLOSUM45

Tabelle B.1: Vergleich von PAM-Matrizen und BLOSUM-Matrizen.

Bei der Berechnung von BLOSUM-Matrizen werden nur konservierte Bereiche, d. h. sehr ähnliche Regionen, verwendet, in denen keine Lücken enthalten sein dürfen. PAM-Matrizen hingegen beziehen sowohl ähnliche als auch unterschiedliche Bereiche in die Berechnungen ein. PAM-Matrizen sind also eher geeignet, um evolutionäre Eigenschaften von Proteinen zu untersuchen. BLOSUM-Matrizen wurden für die Identifizierung ähnlicher Bereiche in gegebenen Proteinen entwickelt [32].

Trotz dieser Unterschiede können Äquivalenzen zwischen verschiedenen PAM- und BLOSUM-Matrizen angeben werden (siehe Tabelle B.1). In unserer experimentellen Analyse kommt eine Distanzvariante der PAM-250-Matrix (siehe Abbildung B.1) zum Einsatz, da wir für unsere Berechnungen dasselbe Modell wie Reinert et al. [36] verwenden wollen.

Literaturverzeichnis

[1] *National Center for Biotechnology Information. Cytochrome Database.* `http://www.ncbi.nlm.nih.gov`.

[2] Alberts, Bruce, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts und Peter Walter: *Molecular Biology of the Cell.* Garland Publishing, Inc., New York and London, 4. Auflage, 2002.

[3] Altschul, Stephen F.: *Gap Costs for Multiple Sequence Alignment.* Journal of Theoretical Biology, 138:297–309, 1989.

[4] Böckenhauer, Hans-Joachim und Dirk Bongartz: *Algorithmische Grundlagen der Bioinformatik – Modelle, Methoden und Komplexität.* Teubner, Stuttgart, 2003.

[5] Biochemical Nomenclature (CBN), IUPAC-IUB Commission on: *A One-Letter Notation for Amino Acid Sequences. Tentative Rules.* Pure And Applied Chemistry, 31:641–645, 1972. In vielen weiteren Journals veröffentlicht.

[6] Biochemical Nomenclature (CBN), IUPAC-IUB Commission on: *Abbreviations for and Symbols for Nucleic Acids, Polynucleotides and their Constituents, Recommendations 1970.* Pure And Applied Chemistry, 40:277–290, 1974. In vielen weiteren Journals veröffentlicht.

[7] Carrillo, Humberto und David Lipman: *The multiple sequence alignment problem in biology.* SIAM Journal on Applied Mathematics, 48(5):1073–1082, 1988.

[8] Cherkassky, Boris V., Andrew V. Goldberg und Tomasz Radzik: *Shortest Paths Algorithms: Theory and Experimental Evaluation.* In: *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994.

[9] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms.* The MIT Press, Cambridge, Massachussetts, 2. Auflage, 2001.

[10] Dayhoff, Margaret O., Robert M. Schwartz und Bruce C. Orcutt: *A model of evolutionary change in proteins.* Atlas of Protein Sequence and Structure, 5(2):345–352, 1978.

[11] Dijkstra, Edsger W.: *A note on Two Problems in Connection with Graphs.* Nummerische Mathematik, 1:269–271, 1959.

[12] Goldberg, Andrew und Chris Harrelson: *Computing the Shortest Path: A* Search Meets Graph Theory.* In: *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005).* SIAM, 2005.

[13] Goldberg, Andrew, Haim Kaplan und Renato Werneck: *Reach for A*: Efficient Point-to-Point Shortest Path Algorithms.* In: *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX06).* SIAM, 2006.

[14] Goldberg, Andrew und R. Werneck: *Computing Point-to-Point Shortest Paths from External Memory.* In: *Proceedings Workshop on Algorithm Engineering and Experiments (ALENEX 2005).* SIAM, 2005.

[15] Gotoh, Osamu: *An improved algorithm for matching biological sequences.* Journal of Molecular Biology, 162(3):705–708, 1982.

[16] Gotoh, Osamu: *Alignments of three biological sequences with an efficient trace-back procedure.* Journal of Theoretical Biology, 121:327–337, 1986.

[17] Gray, Frank: *Pulse code communication*, 17. März 1953 (angemeldet im November 1947). U.S. Patent 2,632,058, Claims 1-9.

[18] Gupta, Sandeep K., John D. Kececioglu und Alejandro A. Schäffer: *Improving the Practical Space and Time Efficiency of the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment.* Journal of Computational Biology, 2(3):459–472, 1995.

[19] Gusfield, Dan: *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[20] Hart, Peter E., Nils J. Nilsson und Bertram Raphael: *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions on Systems Science and Cybernetics, SSC-4(2):100–107, 1968.

[21] Henikoff, Steven und Jorja G. Henikoff: *Amino acid substitution matrices from protein blocks.* Proceedings of the National Academy of Sciences of the U.S.A., 89(22):10915–10919, 1992.

[22] Heun, Volker: *Skriptum zur Vorlesung Algorithmische Bioinformatik I/II, SS 2005 und WS 2006/06.* Ludwig Maximilians Universität München, 2006.

[23] Hirschberg, Dan S.: *A linear space algorithm for computing maximal common subsequences.* Communications of the ACM, 18(6):341–343, 1975.

[24] Horton, Paul: *String Algorithms and Machine Learning Applications for Computational Biology.* Doktorarbeit, University of California, Berkeley, CA, 1997.

[25] Ikeda, Takahiro, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku und Kunihiko Mitoh: *A Fast Algorithm for Finding Better Routes by AI Search Techniques.* In: *Proceedings Vehicle Navigation and Information Systems Conference.* IEEE, 1994.

[26] Kruskal, Joseph B.: *An Overview of Sequence Comparison.* In: Sankoff, David und Joseph B. Kruskal (Herausgeber): *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Seiten 1–44. Addison-Wesley, Reading, MA, 1983.

[27] Kwa, James B.H.: *BS*: an admissible bidirectional staged heuristic search algorithm.* Artif. Intell., 38(1):95–109, 1989.

[28] Lengauer, Thomas: *Combinatorial Algorithms for Integrated Circuit Layout.* Teubner, Stuttgart, 1990.

[29] Lermen, Martin und Knut Reinert: *The Practical Use of the A^* Algorithm for Exact Multiple Sequence Alignment.* Journal of Computational Biology, 7(5):655–671, 2000.

[30] Levenshtein, Vladimir I.: *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady, 10(8):707–710, 1966.

[31] McClure, Marcella A., Taha K. Vasi und Walter M. Fitch: *Comparative analysis of multiple protein-sequence alignment methods.* Molecular Biology and Evolution, 4(11):571–592, 1994.

[32] Mount, David W.: *Bioinformatics – Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, Cold Spring Harbor, New York, 2. Auflage, 2004.

[33] Needleman, Saul B. und Christian D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequences of two proteins.* Journal of Molecular Biology, 48:443–453, 1970.

[34] Pevzner, Pavel A.: *Computational Molecular Biology – An Algorithmic Approach.* The MIT Press, Cambridge, Massachussetts, 1. Auflage, 2000.

[35] Pohl, Ira: *Bi-directional Search.* Machine Intelligence, 6:124–140, 1971.

[36] Reinert, Knut, Jens Stoye und Torsten Will: *An iterative Method for faster Sum-of-Pairs Multiple Sequence Alignment.* Bioinformatics, 16(9):808–814, 2000.

[37] Russell, Stuart und Peter Norvig: *Künstliche Intelligenz – Ein moderner Ansatz.* Pearson Education Deutschland, München, 2. Auflage, 2004.

[38] Shibuya, Tetsuo und Hiroshi Imai: *New flexible approaches for multiple sequence alignment.* In: *RECOMB '97: Proceedings of the first annual international conference on Computational molecular biology*, Seiten 267–276, New York, NY, USA, 1997. ACM Press.

[39] Smith, Temple F. und Michael S. Waterman: *Comparison of biosequences.* Advances in Applied Mathematics, 2:482–489, 1981.

[40] Smith, Temple F., Michael S. Waterman und Walter M. Fitch: *Comparative biosequence metrics.* Journal of Molecular Evolution, 18:38–46, 1981.

[41] Stoye, Jens: *Divide-and-Conquer Multiple Sequence Alignment.* Doktorarbeit, Universität Bielefeld, 1997.

[42] Stoye, Jens: *Multiple sequence alignment with the divide-and-conquer method.* Gene, 211:GC45–GC56, 1998.

[43] Thompson, Julie Dawn, Frédéric Plewniak und Olivier Poch: *BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs.* Bioinformatics, 15(1):87–88, 1999.

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

[45] Wagner, Robert A. und Michael J. Fischer: *The String-to-String Correction Problem.* Journal of the ACM, 21(1):168–173, 1974.

[46] Wang, Lusheng und Tao Jiang: *On the Complexity of Multiple Sequence Alignment.* Journal of Computational Biology, 1(4):337–348, 1994.

[47] Waterman, Michael S.: *Introduction to Computational Biology – Maps, sequences and genomes.* Chapman and Hall, 2 – 6 Boundary Row, London SE1 8HN, UK, 1. Auflage, 1995.

[48] Waterman, Michael S., Temple F. Smith und William A. Beyer: *Some biological sequence metrics.* Advances in Mathematics, 20:367–387, 1976.

[49] Wegener, Ingo: *Skriptum zur Vorlesung Datenstrukturen Algorithmen und Programmierung 2.* Universität Dortmund, 2005.

[50] Werners, Brigitte: *Grundlagen des Operations Research.* Springer, Berlin, 1. Auflage, 2006.

[51] Wilbur, W. John und David J. Lipman: *The context-dependent comparison of biological sequences.* SIAM Journal on Applied Mathematics, 44:557–567, 1984.