FACHHOCHSCHULE LAUSITZ

FACHBEREICH INFORMATIK

Studiengang Informatik

DIPLOMARBEIT

Ein Algorithmus zum Vergleich von Proteinsequenzen aufgrund Ihrer Domänenzusammensetzung

Katja Wegner

(geb. am 04.08.1977)

Betreuer: Dr. rer. nat. Ursula Kummer

2. Gutachter: Prof. Dr. rer. nat. habil. Wolfgang Laßner

Senftenberg, August 2001

// Abstract

Sequence alignment methods are important in order to gain information about unknown proteins. They are based upon the idea that similar sequences also have similar functions and structures. The existing algorithms are able to find pointmutations. Nevertheless during the evolutionary process, recombinations (domain or exon shuffling) have occured as well. These cannot be found with the help of the algorithm up to date. Therefore this study focuses on a new algorithm that takes into account both aspects, i.e. on the one hand pointmutations and on the other hand recombinations. This algorithm consists of two parts. First, it produces a list of local alignments by using a modified BLAST algorithm. This part takes into account pointmutations in the sense of substitution, deletion or insertion of one amino acid. In the second part a graphical theoretical approach recognizes possible recombinations. The results record segments that could be recombinations. However the results strongly depend on user entered parameters.

Übersicht

Sequenzvergleiche sind sehr wichtig, um Informationen über unbekannte Proteine zu erhalten. Sie basieren auf dem Ähnlichkeitsdogma, d.h. eine ähnliche Sequenz weist auf eine ähnliche Funktion und Struktur hin. Die existierenden Algorithmen sind in der Lage Punktmutationen zu finden. Jedoch gab es während der Evolution ebenso Rekombinationen. Diese können nicht mit den vorhandenen Algorithmen gefunden werden. Deshalb konzentriert sich diese Diplomarbeit auf die Entwicklung eines neuen Algorithmus, der Punktmutationen und Rekombinationen berücksichtigen kann. Dieser Algorithmus besteht aus zwei Teilen. Als erstes wird eine Liste mit lokalen Alignments mit Hilfe eines modifizierten BLAST-Algorithmus erstellt. Dieser Teil findet Punktmutationen in der Form von Substitutionen, Einfügungen oder Entfernungen einzelner Aminosäuren. Im zweiten Schritt werden mittels eines grafischen Ansatzes mögliche Rekombinationen gesucht. Die Ergebnisse zeigen Sequenzstücke, die Rekombinationen gewesen sein könnten. Die Ergebnisse hängen jedoch stark von den benutzerdefinierten Parametern ab.

Danksagung

Ich möchte mich bei allen, die mich bei der Anfertigung dieser Arbeit unterstützt haben, bedanken. Meiner Familie möchte ich für die Unterstützung und Geduld der letzten Jahre danken. Ein besonderer Dank geht an Stefan Wuchty und meine Betreuerin Dr. Ursula Kummer für ihre ständige Bereitschaft zur Diskussion und Hilfe während der Durchführung der Diplomarbeit. Professor Dr. Laßner danke ich für die Bereitschaft, meine Diplomarbeit als Zweitgutachter zu betreuen. Desweiteren bedanke ich mich bei Dr. Ursula Rost für die Hilfe bei javaspezifischen Problemen und bei Ralph Gauges für das Lösen von systemadminstrativen Problemen. Für das Korrekturlesen meiner Diplomarbeit bedanke ich mich bei Dr. Ursula Kummer, Sabine Portlu, Dr. Ursula Rost und Stefan Wuchty. Bei allen Mitarbeiterinnen und Mitarbeitern des European Media Labratory (EML) möchte ich mich für das angenehme Arbeitsklima und die permanente Hilfsbereitschaft bedanken. Allen meinen Freunden, Bekannten und Verwandten, die mich unterstützt und mir beigestanden haben und aus dieser Aufzählung ausgeblieben sind, danke ich ebenso.

Inhaltsverzeichnis

1. Einleitung 9

2. Grundlagen 12

2.1. Proteine 12

2.1.1. Domänen 15

2.2. Sequenzvergleiche 16

2.2.1. Alignment 16

2.2.2. Substitutionsmatrizen 19

2.2.3. Dynamische Programmierung 20

2.2.4. Heuristische Suchverfahren 24

2.3. Graphen und Dijkstra-Algorithmus 25

3. Entwicklung eines Algorithmus 27

4. Implementierung 32

4.1. Die Programmiersprache Java 32

4.2. Eingabemodus 33

4.3. Eine Anfrage bearbeiten 36

4.4. Ausgabemodus 37

5. Auswertung 40

5.1. Wahl der Parameter 40

5.2. Vergleich mit dem **B**asic **L**ocal **A**lignment **S**earch **T**ool 43

6. Zusammenfassung 48

6.1. Ausblick 49

A. Proteinsubstitutionsmatrizen 51

A.1. BlosumN 51

A.2. PAM500 52

A.3. VT250 52

B. Testergebnisse 53

C. UML - Unified Modelling Language 57

C.1. Zeichenerklärung 58

C.2. Klassendiagramm mit allen implementierten Klassen 59

D. Inhalt der Begleit-CD 60

Abkürzungen

Abb.	Abbildung
BLAST	Basic Local Alignment Search Tool
bzw.	beziehungsweise
d.h.	dass heisst
DNA	Desoxyribonukleinsäure
Kap.	Kapitel
s.	siehe
Tab.	Tabelle
UML	Unified Modelling Language
usw.	und so weiter
z.B.	zum Beispiel

Hinweis:

$$max \left\{ \begin{array}{c} a_1, \\ a_2, \\ .. \\ a_n \end{array} \right\}$$ entspricht der Schreibweise $max\{a_1, a_2, .., a_n\}$

Abbildungsverzeichnis

2.1. Struktur von Aminosäuren [Kni97] 12
2.2. Sekundärstrukturen (α-Helix und β-Blatt) [Kni97] 14
2.3. Räumliche Struktur der PH-Domäne [Pfa01] 15
2.4. Veranschaulichung einer Rekombination 17
2.5. Beispiele für globale und lokale Alignments 18
2.6. Rekursionschema zum Berechnen eines Eintrags in einer Score-Matrix 22
2.7. Beispiel für eine Score-Matrix 23
2.8. Gerichteter Graph 26

3.1. Zusammensetzen der Zielsequenz aus der Quellsequenz 29
3.2. Drei-Zustandsmodell 30
3.3. Beispiel des Algorithmus 31

4.1. Klassendiagramm der grafischen Oberfläche 34
4.2. Startansicht des Programms (Eingabemodus) 35
4.3. Aufruf eines Fehlerfensters 36
4.4. Oberfläche im Textmodus 38
4.5. Oberfläche im Grafikmodus 38
4.6. UML-Diagramm der Klasse *LocalAlignmentFrame* und ein Beispiel . . 39

5.1. Testproteine und ihre Domänenstrukturen 45
5.2. Vergleich von SERINE/THREONINE-PROTEIN KINASE A und SERINE/THREONINE-PROTEIN KINASE SHK2 46

C.1. Darstellung eines Klassendiagramms 57

Tabellenverzeichnis

2.1. Aminosäuren im Überblick 13
2.2. Einteilung der Aminosäuren [ABL+94] 14
2.3. Sonderzeichen 17
2.4. Beispiel eines Sequenz-Alignment 21

4.1. Benutzerdefinierte Parameter 35

5.1. Verschiedene Strafen für Lücken 41
5.2. Beispiele für *threshold 1* 42
5.3. Wirkung von *threshold 3* 44

C.1. Vielfachheit einer Verbindung zwischen zwei Klassen 58

1. Einleitung

Vergleiche von Sequenzen spielen eine zentrale Rolle in der Bioinformatik. Dabei beruht die Vorgehensweise auf der Annahme, daß ähnliche Proteinsequenzen auch ähnliche Strukturen und somit ähnliche Funktionen besitzen. Die enormen Datenmengen, die in den letzten Jahren durch die Sequenzierung ganzer Genome verfügbar geworden sind, wären ohne solche Methoden aus der Bioinformatik nicht auswertbar.
Die ersten Algorithmen entsprechen dem Prinzip der dynamischen Programmierung, zu diesen gehören z.B. der Needleman-Wunsch-Algorithmus [NW70] zur Berechnung von globalen Alignments oder der Smith-Waterman-Algorithmus [SW81] für lokale Alignments. Globale Alignments erstrecken sich über die gesamte Sequenz, wohingegen lokale Alignments nur eine Aussage über die Ähnlichkeit von Teilen der Gesamtsequenz wiedergeben. Diese Algorithmen bestimmen eine optimale Lösung und ihr Aufwand steigt mit der Länge der Sequenzen. Aufgrund der immer weiter wachsenden Datenmengen in den vorhandenen Datenbanken sind heuristische Verfahren, z.B. BLAST [ABGW94, AGM+90, AMS+97] und FASTA [PL88] entwickelt worden. Diese Verfahren sind schneller und benötigen weniger Speicherplatz als die Algorithmen der dynamischen Programmierung, garantieren jedoch keine optimale Lösung. Einen Überblick über die existierenden Algorithmen geben Apostolico [AG97] und Pearson [Pea00].
Bei den genannten Sequenzvergleichen erfolgt das Alignment linear über die ganze Sequenz, d.h. die zu vergleichenden Sequenzen werden an einer Stelle aneinandergelegt und von dort aus (oft mit Lücken) immer in derselben Richtung verglichen. Auf diese Weise können Punktmutationen erkannt werden. Unter Punktmutationen versteht man, dass eine Aminosäure in einem Protein ausgetauscht, eingefügt oder

entfernt wurde. Zur Bewertung solcher Ereignisse stehen Substitutionsmatrizen zur Verfügung. Sie enthalten für jede mögliche Aminosäurenkombination einen Wert für die Wahrscheinlichkeit, dass diese beiden Aminosäuren gegeneinander ausgetauscht worden sind. In diesen findet man einen Wert für jedes mögliche Aminosäurenpaar. Die am meisten verwendeten Substitutionsmatrizen sind Blosum [HH92] und PAM [Alt91]. Diese werden am besten auf Proteine angewendet, die einen konstanten oder geringen Grad der Divergenz haben, d.h. auf Proteine, die eng verwandt sind. Müller und Vingron [MSV00] entwickelten dagegen eine Matrix, die auf einem variablen Verwandtschaftsgrad aufbaut.

Im Laufe der Evolution gab es ebenso vielfach Rekombinationen, die den Einbau und die Umordnung bestimmter Abschnitte einer Sequenz (z.B. Domänen) als Ganzes zuliessen [GL99, Li99]. Diese Ereignisse sind ebenso wichtig wie Punktmutationen, um die Verwandtschaft zwischen Proteinen festzustellen. Allerdings können sie nicht mit den bestehenden Algorithmen erkannt werden, da diese ein optimales Alignment stets aus optimalen Teilalignments zusammensetzen, die sich nicht überschneiden und deren Reihenfolge nicht geändert werden kann. Eine Rekombination hat stattgefunden, wenn zwei Sequenzstücke der einen Sequenz in der anderen Sequenz in umgekehrter Reihenfolge auftreten. Um diese zu erkennen, müsste dagegen die Reihenfolge der Teilalignments verändert werden.

Dieser Problematik der existierenden Algorithmen haben sich schon andere Wissenschaftler gewidmet und haben verschiedene Ansätze entwickelt. Ein Beispiel sind Pseudoknoten in der RNA, die aus sich überschneidenden Teilsequenzen bestehen und deswegen nicht mit den klassischen Methoden behandelt werden können. Rivas und Eddy [RE99] zerlegen die Pseudoknoten in Segmente, die dann entsprechend der dynamischen Programmierung zusammengesetzt werden. Es werden die Stücke, die eine Verletzung der Regeln darstellen, herausgenommen und separat behandelt. Ähnlich gehen Schoeninger und Waterman [SW92, Wat95] vor, um Inversionen in der DNA zu finden.

Heringa und Argos [HA93] haben ein Programm (REPRO) geschrieben, das enfernte Wiederholungen in Proteinsequenzen findet. Zuerst werden diese Wiederholungen

mittels des Smith-Waterman-Algorithmus gesucht und danach werden sie in Cluster eingeteilt. In einem Cluster befinden sich die Stücke, die sich am ähnlichsten sind und eine Wiederholung darstellen könnten.
Mit der Evolution von Proteinen haben sich Morgenstern und Atcheley [MA99] beschäftigt. Sie stellen fest, dass eine Klassifikation aufgrund einzelner Domänen, wie sie mit den klassischen Algorithmen gemacht wird, die wahre evolutionäre Beziehung von Proteinen verdecken könnte; denn wenn eine Proteingruppe (Cluster) nur auf einer gemeinsamen Domäne basiert, wird es schwierig sein, eine genaue Schätzung der Verwandtschaft zu machen. Deshalb entwickelten Enright and Ouzounis [EO00] eine automatische Clustermethode (GeneRAGE) von großen Proteinsequenzdaten, die auch Proteine mit vielen Domänen gruppieren kann. Der Algorithmus stellt alle Ähnlichkeitsbeziehungen zwischen den Proteinen in einer binären Matrix dar. Zufällige Ähnlichkeiten werden entfernt, indem die Matrix mit Hilfe des Smith-Waterman-Algorithmus symmetrisch gemacht wird. Beim iterativen Weiterverarbeiten der Matrixelemente können Proteine mit vielen Domänen und weitere zufällige Ähnlichkeiten erkannt werden. Das Programm bietet eine rasche und bessere Familienrepräsentation jedes betrachteten Proteins.
Für die genannten Beispiele wurden speziellen Lösungen gefunden, um die Nachteile der existierenden Algorithmen zu überwinden. Diese Diplomarbeit beschäftigt sich jedoch mit Proteinen und ihren Domänenzusammensetzungen, die durch Rekombinationen verändert sein können. Es soll ein Algorithmus entwickelt werden, der sowohl Punktmutationen als auch Rekombinationen erkennt und somit die Ähnlichkeit zwischen Proteinen mit gleichen Domänen in verschiedenen Anordnungen feststellen kann, was bisher noch nicht möglich ist. Der Algorithmus sollte ähnlich wie in den genannten Artikeln in zwei Schritten erfolgen. Die Punktmutationen können mit den existierenden Algorithmen berücksichtigt werden und sollten deshalb auch im ersten Schritt damit bestimmt werden. Die Aufgabe des zweiten Teils besteht darin einen Weg zu finden, der mögliche Rekombinationen erkennen und bewerten kann. Diese Teillösungen müssen dann zu einer Gesamtlösung zusammengefasst werden.

2. Grundlagen

Dieses Kapitel vermittelt grundlegende Informationen über Proteine und Domänen, aber ebenso über Algorithmen der dynamischen Programmierung sowie über heuristische Verfahren. Es wird dabei erklärt was ein Alignment ist und welche Rollen die einzelnen Substitutionsmatrizen haben. Im letzten Abschnitt wird die Graphentheorie aufgegriffen, insbesondere dabei der Dijkstra-Algorithmus, der bei der Entwicklung des Algorithmus eine wichtige Rolle spielt.

2.1. Proteine

Proteine (Überblick [LBB+96]) sind Makromoleküle, die aus vielen Einzelbausteinen, den Aminosäuren, bestehen. Es gibt 20 natürliche Aminosäuren (Tab. 2.1). Die Aminosäuren haben eine Säure- oder Carboxyl-Gruppe, eine Amino-Gruppe sowie ein Wasserstoff-Atom und eine Seitengruppe am zentralen C-Atom (Abb. 2.1).

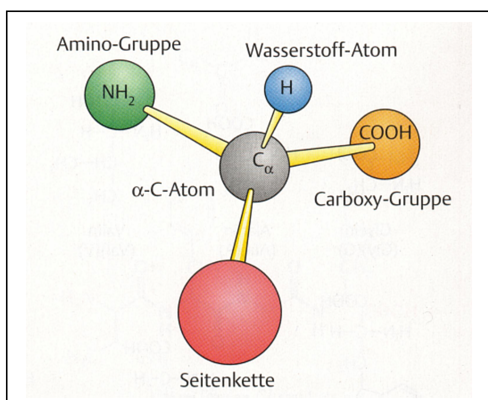


Abbildung 2.1.: Struktur von Aminosäuren [Kni97]

Jede Aminosäure besitzt ein zentrales C-Atom (α-C-Atom), mit dem eine Aminogruppe, ein Wasserstoff-Atom (H), eine Carboxyl-Gruppe sowie eine Seitenkette verbunden sind. Nur durch diese Seitenkette unterscheiden sich die einzelnen Aminosäuren.

Die Aminosäuren werden in dieser Arbeit in vier Gruppen eingeteilt: hydrophob, hydrophil, sauer und basisch (s. Tab. 2.2). Als hydrophil werden wasserlösliche

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

Tabelle 2.1.: Aminosäuren im Überblick

In dieser Tabelle sind alle Aminosäuren mit ihrem Namen und dem Ein-Buchstaben-Code sowie dem Drei-Buchstaben-Code aufgeführt. In dem entwickelten Programm wird nur der Ein-Buchstaben-Code verwendet.

Aminosäuren bezeichnet. Diese besitzen ionisierte oder polare Seitenketten. Die polaren Proteine werden in dieser Einteilung als hydrophob, die ionisierten entweder als basisch oder sauer klassifiziert. Das Gegenteil von hydrophil ist hydrophob. Darunter sind aliphatische Seitenketten zu verstehen, die nicht oder nur sehr wenig wasserlöslich sind. Die hydrophoben Aminosäuren befinden sich meist im Inneren der räumlichen Struktur eines Proteins.

Die Peptidbindung zwischen der Aminogruppe der einen Aminosäure und der Carboxylgruppe der nächsten ermöglicht, dass sich Ketten bilden können. Die Reihenfolge der Aminosäuren bezeichnet man als Sequenz; sie entspricht der Primärstruktur

hydrophob	hydrophil	sauer	basisch
A Alanin	N Asparagin	D Aspartat	H Histidin
C Cystein	Q Glutamin	E Glutamat	K Lysin
F Phenylalanin	S Serin		R Arginin
G Glycin	T Threonin		
I Isoleucin	Y Tyrosin		
L Leucin			
M Methionin			
P Prolin			
V Valin			
W Tryptophan			

Tabelle 2.2.: Einteilung der Aminosäuren [ABL+94]
Diese vier Gruppen werden in der Applikation durch verschiedene Farben dargestellt (hydrophob → grün, hydrophil → blau, sauer → rot und basisch → gelb).

eines Proteins.

Durch Faltung einzelner Abschnitte des Proteins kommt es zur Sekundärstruktur (Abb. 2.2). Dabei können z.B. α-Helices oder β-Faltblätter entstehen. Die Tertiärstruktur entsteht, wenn verschiedene Sekundärstrukturen eine Form bilden. Die Quartiärstruktur tritt bei Proteinen auf, die aus mehreren Untereinheiten bestehen, die eigene Primär-, Sekundär- und Tertiärstrukturen aufweisen. [LP97]

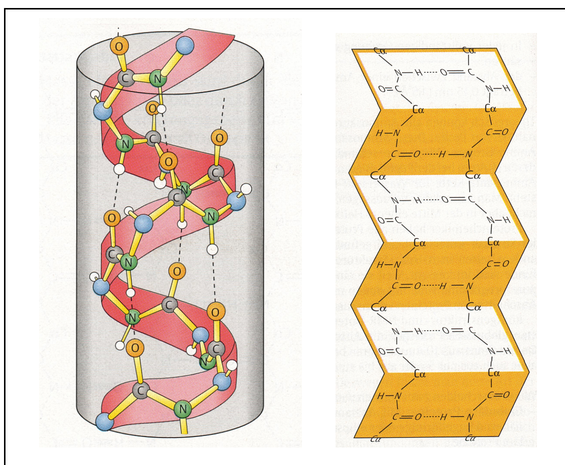


Abbildung 2.2.: Sekundärstrukturen (α-Helix und β-Blatt) [Kni97]
Die α-Helix bildet einen spiralförmig gewundenen Abschnitt (linkes Bild). Innerhalb der Gesamtstruktur eines Proteins ist sie ein stabförmiges Element. Im rechten Bild ist ein β-Faltblatt zu sehen.

2.1.1. Domänen

Domänen [JC85] bilden die kleinsten Protein-Einheiten, die definierte und unabhängig gefaltete Strukturen besitzen (s. Abb. 2.3). Unabhängig vom Rest der Sequenz faltet sich diese Region in eine ausgeprägte Struktur und kennzeichnet die biologische Funktion. Somit entdeckt man in unterschiedlichen Proteinen mit gleichen Domänen ähnliche biologische Aktivitäten. Die meisten Proteine besitzen nur eine einzelne Domäne [Doo95]. Es gibt allerdings auch Proteine, die mehrere Domänen besitzen und unterschiedliche Domänenarchitekturen aufbauen. Im Durchschnitt haben sie zwei oder drei Domänen; dennoch gibt es menschliche Proteine, die bis zu 130 Domänen enthalten können [LGWN01].

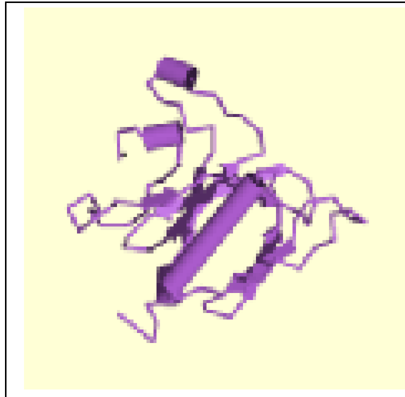


Abbildung 2.3.: Räumliche Struktur der PH-Domäne [Pfa01]

Durch Pfeile werden β-Faltblätter und durch Zylinder α-Helices dargestellt. PH steht für Pleckstrin homology.

Es existieren verschiedene Ansichten bezüglich der Entstehung und der Anzahl der Domänen. Auf der einen Seite geht man davon aus, dass alle Proteine aus einer großen Menge ursprünglicher Proteine durch sogenanntes Shuffling ("Mischen") entstanden sind [DG91].

Auf der anderen Seite geht man davon aus, dass einige wenige Proteine am Anfang existierten. Diese Proteine sind die Vorgänger der meisten heutigen Proteine [Doo95]. Durch Duplikation und anschließende Modifikation entstanden aus dieser kleinen Menge von Molekülen neue Proteine. Diese Veränderungen fanden in der DNA statt; denn die Informationen über die Reihenfolge der Aminosäuren in einem Protein werden den Genen entnommen. Letztere sind aus kodierenden (Exons) und nicht kodierenden Sequenzabschnitten (Introns) aufgebaut. Zeitlich unabhängig von der Einführung der Introns bietet die Rekombination von Introns eine Möglichkeit

Exons zwischen den Genen auszutauschen. Dieses Vorgehen zum Bilden neuer Funktionen von Genen nennt man "exon-shuffling". Diese Shuffling-Ereignisse sind nur von biologischer Signifikanz, wenn in dem Exon eine funktionelle oder strukturelle Domäne enthalten ist. Obwohl viele Beispiele für "exon-shuffling" gefunden wurden, konnten keine wesentliche Korrelationen zwischen den Exons und den entsprechenden Proteinstruktureinheiten erkannt werden [SSZ+94].
Allgemein kann man sagen, dass viele der neu sequenzierten Proteine homolog zu anderen bekannten Proteinen sind. Homolog bedeutet, dass sie sich in großen Teilen der Sequenz sehr ähnlich sind. Folglich könnten diese Proteine von gemeinsamen Vorfahren abstammen. Bei großen Proteinen gibt es oft Zeichen, die darauf hindeuten, dass sie sich aus einer neuen Kombination von existierenden Domänen entwickelt haben. Diese Methode wird als "domain-shuffling" bezeichnet und tritt in zwei Varianten auf [Doo95]:

1. Interne Duplikation mindestens einer Domäne in einem Gen.
2. Einfügen einer Domäne, wenn eine strukturelle oder funktionelle Domäne zwischen zwei Proteine ausgetauscht oder in ein Protein eingefügt wird.

Biologisch wichtiger ist das "domain-shuffling", weil Domänen echte strukturelle und funktionelle Einheiten bilden, aber Exons oft nicht.

2.2. Sequenzvergleiche

2.2.1. Alignment

In vielen Fällen ist die erste Information, die man über ein Protein bekommt, die Aminosäurensequenz. Um die Struktur und die Funktion zu bestimmen, kann man ein Sequenzvergleich mit bekannten Proteinen durchführen. Denn aus einer starken Ähnlichkeit der Sequenz kann man auf eine ähnliche Struktur und Funktion rückschliessen.
Ein paarweises Alignment entsteht, wenn zwei Sequenzen aneinander gelegt und dann in eine Richtung verglichen werden. Es gilt jedoch zu beachten, dass es nicht

mit einem Wortvergleich identisch ist. Denn während der Evolution kam es zu Punktmutationen und Rekombinationen (Shuffling, Abb. 2.4), die berücksichtigt werden müssen. Unter Punktmutationen versteht man das Einfügen, Entfernen oder Ersetzen von einer Aminosäure. Bei einer Rekombination können Sequenzen durch den Ein- und Umbau von ganzen Stücke (z.B. Domänen) verändert worden sein.

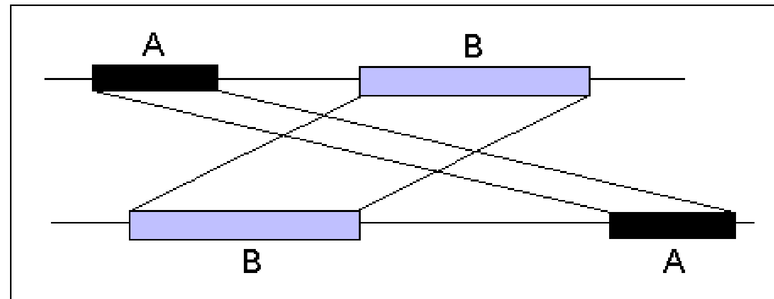


Abbildung 2.4.: Veranschaulichung einer Rekombination

Daraus ergibt sich die Notwendigkeit, dass der Algorithmus in der Lage sein muss, diese Veränderungen zu erkennen und zu behandeln. Mit den existierenden Algorithmen können nur Punktmutationen berücksichtigt werden. Das ist darauf zurückzuführen, dass ein Richtungswechsel, der zur Erkennung einer Rekombination notwendig ist, mit den klassischen Methoden nicht möglich ist.

Es ist ein endliches Alphabet vorhanden, das aus dem Ein-Buchstaben-Code der Aminosäuren, den Buchstaben B, X, Z sowie dem Zeichen '*' (Tab. 2.3) besteht.

B	steht entweder für die Aminosäure Aspartat oder für Arginin
X	steht für eine beliebige Aminosäure
Z	steht entweder für Glutamat oder Glutamin
'*'	unbekannt

Tabelle 2.3.: Sonderzeichen

Durch Sequenzierung werden Art und Position jeder Aminosäure in der Primärstruktur bestimmt. Dabei kann es vorkommen, dass Aminosäuren nicht genau erkannt werden. Aus diesem Grund wurden diese Sonderzeichen eingeführt.

Für jede Buchstabenkombination des Alphabets gibt es einen Wert (Score) für die Wahrscheinlichkeit, dass diese beiden Aminosäuren gegeneinander ausgetauscht wurden. Diese Scores zur Bewertung von Punktmutationen entnimmt man Substitutionsmatrizen. Identitäten oder häufige Substitutionen werden positiv und seltene Substitutionen werden negativ bewertet. Wenn eine Aminosäure eingefügt wurde oder verloren gegangen ist, wird das im Alignment durch ein Lücke (Gap) dargestellt. Ein Gap wird durch das Zeichen '−' symbolisiert. Dadurch rücken die Aminosäuren nach dem Gap um eine Position nach rechts. Der Wert (Score) eines Alignment entspricht der Summe aller lokalen Bewertungen der einzelnen Aminosäuren zueinander (Scores) plus den negativen Werten für Lücken.

Man unterscheidet zwischen globalen und lokalen Alignments (Abb. 2.5). Das globale Alignment erfolgt über die gesamte Sequenz und kann mit dem Needleman-Wunsch-Algorithmus [NW70] bestimmt werden. Dieses Alignment kann große Stücke mit geringer Ähnlichkeit enthalten, d.h. es können viele Lücken auftreten. Lokale Alignments finden die Stücke beider Proteine, an denen sie sich am ähnlichsten sind. So kann man beispielsweise herausfinden, ob sie eine gemeinsame Domäne besitzen. Zum Berechnen dieser Art des Alignment steht der Smith-Waterman-Algorithmus [SW81] zur Verfügung. Einen genaueren Einblick in dieses Thema bietet Durbin [DEKM98].

Globales Alignment

```
1. SQECLSFATW-ALKK-SPVRRMRTDEAL--SHKFLSSDPSMV
2. AQELLVHTAWNELKVVSPVCRPAA-EALLCNHIFQECSPGVV
                 1***1    2*******2
```

Lokale Alignments:

```
1. SPVRR       1. EAL--SHKF
2. SPVCR       2. EALLCNHIF
```

Abbildung 2.5.: Beispiele für globale und lokale Alignments

Zum Bilden der angegebenen Alignments wurden die folgende Sequenzen verwendet:

1. SQECLSFATWALKKSPVRRMRTDEALSHKFLSSDPSMV
2. AQELLVHTAWNELKVVSPVCRPAAEALLCNHIFQECSPGVV

Zusammenfassend kann man sagen, dass das Ziel ein Alignment mit maximalem Score ist. Übereinstimmungen (Matches) werden belohnt und Mismatches oder Lücken bestraft (Gap penalty).

2.2.2. Substitutionsmatrizen

Um Punktmutationen zu berücksichtigen, wurden Substitutionsmatrizen entwickelt. In diesen Matrizen findet man für jede mögliche Kombination des vorhandenen Alphabets, ein Maß für die Wahrscheinlichkeit, dass diese betrachteten Aminosäuren im Laufe der Evolution gegeneinander ausgetauscht wurden. Der Matrixwert ist positiv bei häufigen und negativ bei seltenen Austauschungen. Es handelt sich um symmetrische Matrizen. Auf der Diagonalen findet man die Werte für gleiche Aminosäuren während die anderen Werte ein Maß für die Austauschhäufigkeit von Aminosäuren sind.
In der entstandenen Applikation hat der Nutzer die Möglichkeit, sich die Substitutionsmatrix auszusuchen, die zur Berechnung des Alignments herangezogen werden soll. Die zur Verfügung stehenden Matrizen werden in den folgenden Abschnitten kurz vorgestellt. Die einzelnen Matrizen unterscheiden sich sowohl in der Methode, mit der die einzelnen Werte bestimmt wurden, als auch im Informationsgehalt, der sich daraus ableiten lässt.

Identitätsmatrix Bei dieser Matrix kann der Wert für einen Match selbst festgelegt werden. Im Programm beträgt der Wert 6. Ein Match ist nur dann erfüllt, wenn beide zu vergleichenden Aminosäuren identisch sind. Alle anderen Kombinationen werden als Mismatch betrachtet und bekommen den Wert -6.

Blosum-Matrix [HH92] Um diese Marix zu berechnen, wurde eine Datenbank mit über 2000 Blöcken erstellt. Diese Blöcke bestehen aus alinierten Sequenzabschnitten ohne Lücken, die mehr als 500 Gruppen verwandter Proteine charakterisieren. Jeder Block repräsentiert eine Proteinfamilie. Für jedes neue Mitglied einer Familie wurden die Scores für Matches und Mismatches gesucht, die das beste Alignment zu jedem der anderen Segmente in diesem Block ergeben.

PAM-Matrix [AGM+90, Alt91] PAM ist die Abkürzung für "**P**oint **A**ccepted **M**utation". Es wurde eine Studie zu Punktmutationen zwischen sehr ähnlichen Proteinen (85 % Identität) durchgeführt. Als Punktmutationen werden Substitutionen, Einfügungen und Entfernungen von Aminosäuren bezeichnet. Dabei wurde das stochastische Modell der Proteinevolution nach Dayhoff [DSO78] hinzugezogen. 1 PAM entspricht einer Substitution auf 100 Aminosäuren. Auf der Grundlage dieses Modells wurde die PAM1-Matrix entwickelt. Wenn die PAM1-Matrix verwendet wird, liegt die Anzahl der Substitutionen bei 1 %. Soll die Anzahl der Punktmutationen höher liegen, wird die PAM1-Matrix entsprechend oft mit sich selbst multipliziert. Auf der Basis dieses Modells wurden die Häufigkeiten bestimmt, mit der jedes Aminosäurepaar in einem Alignment von homologen Proteinen auftritt.

VT-Matrix [MSV00] Zur Berechnung dieser Matrix ist man ebenfalls von dem Modell von Dayhoff [DSO78] ausgegangen. Allerdings mit einem Unterschied: der Grad der Verwandschaft in diesem Modell variiert. Man nimmt an, dass sich 1 % der Aminosäuren eines Proteins in einer bestimmten Zeiteinheit ändern. Jede Aminosäure kann während einer bestimmten Zeit in jede andere Aminosäure mutieren. Die Werte für die Matrix wurden mit Hilfe der Datenbank SYSTERS [KV98] erstellt. In dieser befinden sich rund 100.000 Proteine, die in Familien eingeteilt sind. Aus jeder Gruppe wurde zufällig ein aliniertes Sequenzpaar herausgenommen. Es ist ein iteratives Verfahren, das die evolutionäre Distanz zwischen zwei Sequenzen bestimmt und die Werte in der Häufigkeitsmatrix entsprechend anpasst.

Zur Veranschaulichung, wie ein Alignment ohne Lücken mit Hilfe von Substitutionsmatrizen bestimmt wird, dient die Tabelle 2.4.

2.2.3. Dynamische Programmierung

Die dynamische Programmierung (s. z.B. [CLL96]) wird zur Lösung von Optimierungsproblemen verwendet. Das Ziel besteht darin, aus einer Vielzahl von möglichen Lösungen, die optimale Lösung zu finden. Wenn jede Lösung einen numerischen

Sequenz A	A	V	G	A	L	L	A	Score
BlosumN	−2	−4	−5	6	−3	−4	6	−6
Identität	−6	−6	−6	6	−6	−6	6	−18
PAM500	−1	−2	−3	1	−1	−2	1	−8
VT250	−1	−1	−1	2	−1	−1	2	−1
Sequenz B	R	K	C	A	T	S	A	

Tabelle 2.4.: Beispiel eines Sequenz-Alignment
Den schlechtesten Score liefert die Identitätsmatrix, was darauf zurückzuführen ist, dass es nur dann einen positiven Wert gibt, wenn beide Aminosäuren gleich sind. Die anderen Matrizen beruhen auf Austauschhäufigkeiten. Dadurch können die Werte bei einem Mismatch positiv oder negativ ausfallen.

Wert besitzt, dann ist die Lösung optimal, die entweder einem Minimum oder einem Maximum entspricht. Wesentlich ist hierbei, dass sich der Wert für eine Lösung des Gesamtproblems aus den Lösungen der Teilprobleme zusammensetzt. Diese Teilprobleme sind unabhängig voneinander, aber teilen ebenso Teilteilprobleme miteinander. Diese werden einmal gelöst und danach in einer Tabelle gespeichert. Wird die Lösung eines Teilproblems benötigt, muss sie nicht neu berechnet werden, sondern sie wird der Tabelle entnommen. Die dynamische Programmierung kann in vier Schritte gegliedert werden:

1. Charakterisierung der Struktur einer optimalen Lösung.
2. Wert der optimalen Lösung rekursiv definieren.
3. Wert der optimalen Lösung von unten-nach-oben(bottom-up) berechnen.
4. Optimale Lösung aus den berechneten Informationen konstruieren.

Zur Veranschaulichung der dynamischen Programmierung soll hier der Smith-Waterman-Algorithmus [SW81, Wat95] vorgestellt werden, der in abgewandelter Form in die Entwicklung eines eigenen Algorithmus miteinbezogen wird. Dieser Algorithmus findet die Region zweier Sequenzen, innerhalb derer sie sich am ähnlichsten sind. Das bedeutet, dass er die besten lokalen Alignments findet. So können

Übereinstimmungen mit Domänen und Wiederholungen von Domänen in einer Sequenz erkannt werden. Das Ziel besteht darin, den maximalen Ähnlichkeitswert S (Score) für ein Alignment zu finden. Im folgenden werden zwei Sequenzen $A = \{a_1, a_2, ..., a_m\}$ und $B = \{b_1, b_2, ..., b_n\}$ betrachtet.

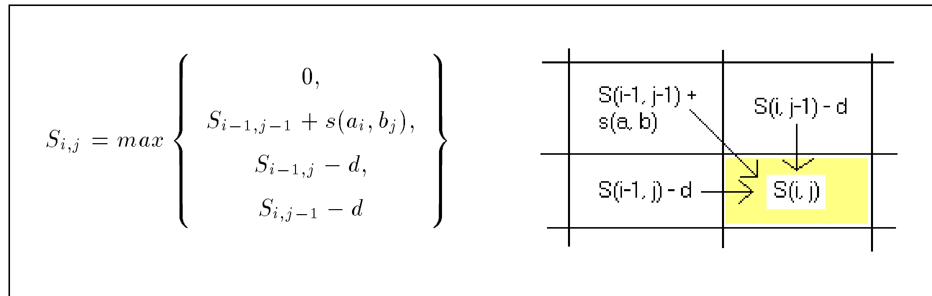


Abbildung 2.6.: Rekursionschema zum Berechnen eines Eintrags in einer Score-Matrix

Als erstes wird für die zu vergleichenden Sequenzen eine Score-Matrix nach dem Rekursionschema aus Abb. 2.6 berechnet. Diese enthält die Teillösungen, die abgerufen werden können, wann immer sie benötigt werden. Jeder Matrixeintrag $S_{i,j}$, der nach Abb. 2.6 berechnet wird, entspricht der Summe aller lokalen Bewertungen der einzelnen Aminosäuren a_1 bis a_i zu b_1 bis b_j. Der kleinste Matrixwert ist 0 und kann den Anfang eines lokalen Alignments bilden. Deswegen werden die erste Zeile ($S_{0,j}$) und die erste Spalte ($S_{i,0}$) auf 0 gesetzt. Der Wert $s(a_i, b_j)$ gibt das Maß der Ähnlichkeit zwischen der i-ten Aminosäure in Sequenz A und j-ten Aminosäure in Sequenz B an. Dieser Wert wird einer Substitutionsmatrix entnommen. Wenn horizontal (obere Sequenz) oder vertikal (untere Sequenz) gegangen wird, d.h. es wird eine Lücke (Gap) eingefügt, gibt es eine Strafe d (Gap penalty). Das entspricht den Fällen 3 und 4 in der Formel in Abb. 2.6.

Diese Entscheidungen müssen für jeden MN Präfix der Sequenzen A und B der Länge M und N gemacht werden. Die letzten drei Fälle aus Abb. 2.6 können unter 0 fallen, wenn die Austauschhäufigkeit dieser beiden Aminosäuren sehr gering ist oder eine Lücke eingefügt wird. Wenn der Score negativ ist, wird der Wert auf 0 zurückgesetzt. Beide Ereignisse würden den Wert eines darauf folgenden Alignments verschlechtern und werden so nicht miteinbezogen. In der Abb. 2.7 ist ein Beispiel für eine Score-Matrix dargestellt.

	A		B		D		D		E	
A	↖									
		1		0		0		0		0
B			↖							
		0		2	←	0		0		0
D				↑	↖		↖			
		0		0		3	←	1		0
E						↑	↖		↖	
		0		0		1		2		2
G							↖	↑	↖	
		0		0		0		0		1

Optimales lokales Alignment:

A B D (obere Sequenz)

A B D (untere Sequenz)

Abbildung 2.7.: Beispiel für eine Score-Matrix

Bei einer Übereinstimmung wird der Score um 1 erhöht und im entgegengesetzten Fall um -2 verringert. Der niedrigste Wert ist 0. Das beste lokale Alignment ist "ABD". [Pea00]

Um das beste lokale Alignment zu finden, wird nach dem größten Wert in der Matrix gesucht. An diesem Punkt wird die Entstehung dieses Wertes zurückverfolgt. Dieser Vorgang wid Backtracking genannt und wird in der Abb. 2.7 durch Pfeile gekennzeichnet.

Es gibt drei Fälle, wie ein Wert $S_{i,j}$ im Beispiel von Abb. 2.6 entstanden sein kann. In diesem Beispiel wurden die Werte willkürlich festgelegt. Wenn zwei Aminosäuren gleich sind, wird $S_{i,j}$ um 1 erhöht. Wenn eine Lücke eingefügt wird, gibt es eine Strafe von -2. [Pea00]

1. Die Aminosäure an der Stelle (i, j) waren gleich und $S_{i-1,j-1}$ wurde um 1 erhöht (diagonal).

2. In der unteren Sequenz wurde ein Lücke eingefügt und $S_{i-1,j}$ wurde um -2 verringert (vertikal).

3. In der oberen Sequenz wurde ein Lücke eingefügt und $S_{i,j-1}$ wurde um -2 verringert (horizontal).

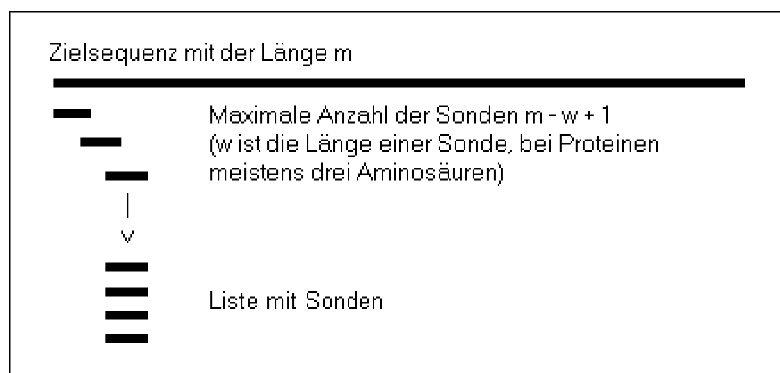
Man geht so lange zurück, bis der Wert 0 erreicht wird; man befindet sich dann am Anfang des lokalen Alignments. In dem Beispiel von Abb. 2.7 ist der höchste Score 3 und das gefundene Alignment ist in beiden Sequenzen "ABD".

2.2.4. Heuristische Suchverfahren

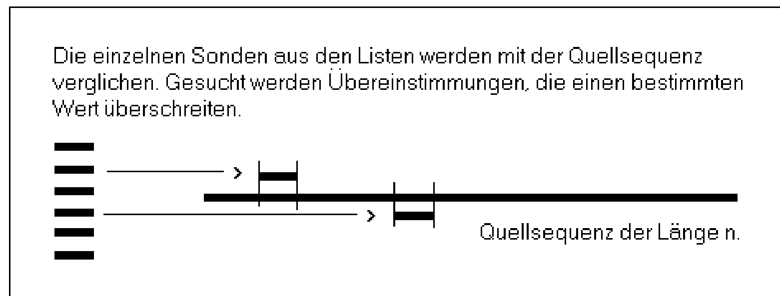
Heuristische Suchverfahren [DEKM98] sind sehr viel schneller und benötigen viel weniger Speicherplatz als die Algorithmen der dynamischen Programmierung, da sie nur eine bestimmte Anzahl der Teilprobleme lösen. Aus diesem Grund kann es sein, dass die optimale Lösung nicht gefunden wird, da nicht alle möglichen Lösungen betrachtet werden.

Ein Vertreter dieser Methode ist das **B**asic **L**ocal **A**lignment **S**earch **T**ool (BLAST) [AGM+90]. Die zugrundeliegende Idee besagt, dass jedes gute Alignment irgendwo ein Stück enthält, das komplett identisch ist oder das einen besonders hohen Score bekommen hat. Dieses Stück wird in der Sequenz A gesucht und als Sonde in der Sequenz B verwendet. Es wird eine Liste mit Sonden erstellt, die eine feste Länge haben (bei Proteinen üblicherweise drei Aminosäuren). Außerdem muss der Score dieser Sonde aliniert mit der Sequenz A, aus der sie stammt, einen vorgegebenen Grenzwert übersteigen. Wenn eine Übereinstimmung in der Sequenz B gefunden wurde, wird die Sonde nach rechts und links in beiden Sequenzen erweitert. Das wird weitergeführt, bis ein Alignment gefunden wurde, das eine bestimmte Länge hat oder der Score nicht weiter steigt. Diese Alignments sind lokal maximal, d.h. sie können nicht durch Erweitern oder Kürzen erhöht werden. Die hier genannten Grenzwerte können durch den Anwender bestimmt werden. Zusammenfassend kann man den Algorithmus in folgende Schritte untergliedern (Abb. nach [Ste97]):

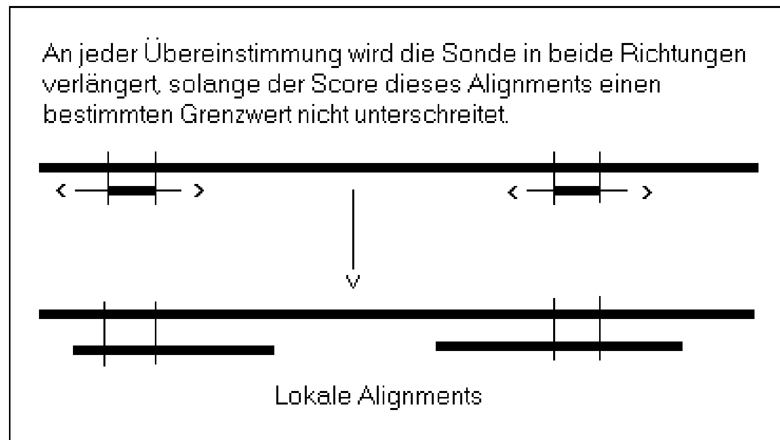
1. Erstellen einer Liste mit den Sonden, die einen bestimmten Grenzwert überschreiten.



2. Durchsuchen der zweiten Sequenz nach Treffern (hits).



3. Erweitern der Treffer nach links und rechts in beiden Sequenzen.



2.3. Graphen und Dijkstra-Algorithmus

An dieser Stelle wird etwas auf die Graphentheorie (s. z.B. [CLL96]) eingegangen, da sie beim Entwickeln des Algorithmus eine wichtige Rolle spielt. Ein Graph (s. Abb. 2.8) besteht aus einer Menge von Knoten V und den dazwischen liegenden Kanten E, die die Beziehung zu den Knoten herstellen. Es gibt gerichtete und ungerichtete Graphen. Eine Kante (u, v) in einem gerichteten Graphen kann durch zwei Ereignisse beschrieben werden. Entweder sie verlässt den Knoten u oder sie trifft in den Knoten v ein. In einem ungerichteten Graphen können die Kanten immer in beide Richtungen begangen werden. Außerdem können die Kanten eines Graphen Gewichte besitzen. Dann bezeichnet man den Graphen als gewichtet. Diese Gewichte werden normalerweise durch eine Gewichtsfunktion angegeben.

Der Weg der Länge k von einem Startknoten zum Zielknoten durch den Graphen entspricht der Reihenfolge der Knoten $(v_0, v_1, .., v_k)$, dabei ist v_0 der Startknoten und v_k der Zielknoten.

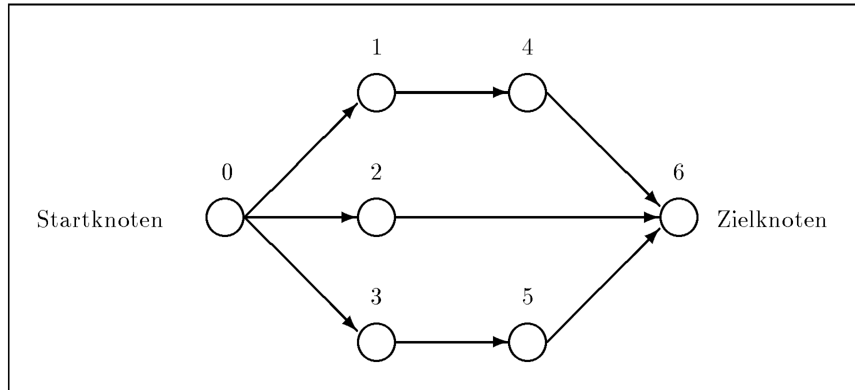


Abbildung 2.8.: Gerichteter Graph

Der Dijkstra-Algorithmus findet den kürzesten Pfad von einem Startknoten zu allen anderen Knoten in einem gewichteten, gerichteten Graphen für den Fall, dass alle Gewichte positiv sind. Der Algorithmus erstellt eine Menge S mit allen Knoten, deren kürzester Pfad bereits berechnet wurde. Aus diesem Grund wird der Menge $V - S$ (V ist die Menge aller Knoten des Graphen) immer wieder der Knoten entnommen, zu dem in diesem Moment der kürzeste Weg führt. Dieser Knoten wird in der Menge S gespeichert und die Wege, die von diesem Knoten wegführen, werden berechnet. Dies geschieht so lange, bis die Menge $V - S$ leer ist. Zu jedem Knoten werden die Distanz vom Startknoten bis zu diesem Knoten und seinem Vorgänger gespeichert. So kann jeder Weg schnell rekonstruiert werden.

3. Entwicklung eines Algorithmus

Um einen Algorithmus zu finden, der sowohl Punktmutationen als auch Rekombinationen erkennt wurden verschiedenen Ansätze ausprobiert. Diese werden im folgenden kurz vorgestellt. Dabei werden auch die Gründe aufgezeigt, warum einzelne verworfen worden sind.

Zunächst wird der Smith-Waterman-Algorithmus [SW81, Wat95] verwendet, um eine vom Benutzer vorgegebene Anzahl der besten lokalen Alignments zu finden. Um aus diesen ein globales Alignment zu erzeugen, wird dann eine Score-Matrix W erstellt. In diese fließt der Score für die gefundenen lokalen Alignments ein; denn an der Stelle, wo ein Alignment anfängt, wird nicht das Ähnlichkeitsmass aus einer Scorematrix eingetragen, sondern der Score für dieses lokale Alignment. Die Idee hierfür basiert auf dem Sequenz-Alignment-Algorithmus von Schoeniger [SW92] und wurde inspiriert von Rivas [RE99].

Um die lokalen Alignments zu berechnen, werden drei verschiedene Matrizen E, F und S benötigt. In der ersten Matrix E wird der Score, den die Aminosäure an der Position i der Sequenz A gegenüber einer Lücke bekommt, gespeichert. Die zweite Matrix F entsprach genau dem entgegengesetzten Fall, d.h. dem Score für die Aminosäure an der Position j der Sequenz B gegenüber einer Lücke.

$$S_{i,j} = max \left\{ \begin{array}{c} 0, \\ S_{i-1,j-1} + s(a,b), \\ E_{i,j}, \\ F_{i,j} \end{array} \right\} \quad E_{i,j} = max \left\{ \begin{array}{c} S_{i-1,j} - \alpha, \\ E_{i-1,j} - \beta \end{array} \right\} \quad F_{i,j} = max \left\{ \begin{array}{c} S_{i,j-1} - \alpha, \\ F_{i,j-1} - \beta \end{array} \right\}$$

Es gibt eine affin-lineare Strafe für eine Lücke. Affin-linear bedeutet, dass die Verlängerung einer Lücke nicht so hart bestraft wird, wie der Anfang (α entspricht der Strafe für die Eröffnung einer Lücke, β der Strafe für die Verlängerung einer Lücke und x der Anzahl der Gaps):

$$\alpha + (x - 1) * \beta \text{ mit } \alpha > \beta$$

Die Score-Matrix S enthielt den bestmöglichen Score von Position *(i, j)* in den Sequenzen A und B. Aus dieser Matrix wird eine Liste mit den k besten lokalen Alignments bestimmt
Danach wird unter Einbeziehung der Werte der gefundenen lokalen Alignments die Score-Matrix W berechnet. Dafür wird wie oben eine Matrix für die Werte der Position i der Sequenz A gegenüber einer Lücke und eine Matrix für den entgegengesetzten Fall erstellt. Im Unterschied zur Matrix S wird jedoch an den Stellen, wo eins der gefundenen lokalen Alignments anfängt, der Score dieses Alignments eingetragen.
Mit Hilfe der erstellten Score-Matrix W wird ein globales Alignment beider Sequenzen ermittelt, dass die lokalen Alignments berücksichtigen sollte. Beim Backtracking von der Position *(i, j)* zu *(0, 0)* wurde festgestellt, dass in manchen Fällen die Stellen, an denen ein lokales Alignment aufhört, nicht getroffen werden. Folglich hat die Verwendung der Scores von den lokalen Alignments keinen Einfluß auf das entstandene globale Alignment. Es wird ein globales Alignment erzeugt wie bisher, jedoch mit einem höheren Score. Das entspricht aber nicht dem gewünschten Ziel, denn es sollten die lokalen Alignments im globalen Alignment erscheinen und die Stellen markieren, die für Domänen oder Rekombinationen in Frage kommen.

Im nächsten Versuch wird daher der zweite Teil des ersten Ansatzes verändert. Die lokalen Alignments werden wieder mit dem oben genannten Smith-Waterman-Algorithmus berechnet. Die Bildung eines Alignments erfolgt jedoch nicht durch Backtracking, sondern die gefundenen lokalen Alignments werden als Knoten eines gerichteten, gewichteten Graphen betrachtet. Eine der eingebenen Sequenzen wird als Zielsequenz und die andere als Quellsequenz betrachtet. Die Kanten des Gra-

phen richteten sich nach den Anfang- und Endpositionen der lokalen Alignments in der Zielsequenz. Eine Kante zwischen zwei Alignments entsteht, wenn das Ende des ersten Alignments vor dem Anfang des zweiten aufhört und kein drittes Alignment dazwischen liegt. Die Gewichte der Kanten ergeben sich aus dem Score des nachfolgenden Alignments, einer Strafe für die Aminosäuren, die zwischen den beiden Alignments liegen, und einer Strafe für gefundene Rekombinationen. Eine Rekombination ist gegeben, wenn das Ende des betrachteten lokalen Alignments vor dem Anfang eines davorliegenden Alignments in der Quellsequenz ist. Die Lösung besteht darin, dass ein Weg durch diesen Graphen gesucht wird, auf dem die Zielsequenz aus den lokalen Alignments zusammengesetzt werden kann. Zusammensetzen bedeutet hierbei, dass die gefundenen lokalen Alignments ohne Überlappen so aneinander gereiht werden, dass der Gesamtscore maximal wird (Abb. 3.1).

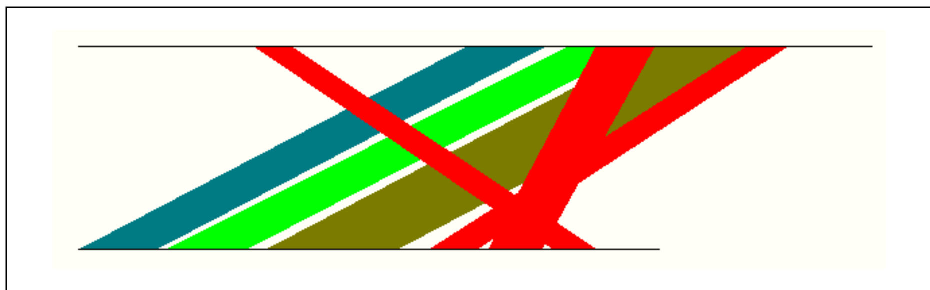


Abbildung 3.1.: Zusammensetzen der Zielsequenz aus der Quellsequenz Die obere Linie repräsentiert die Quellsequenz und die untere die Zielsequenz. Die Alignments, die den maximalen Score ergeben, werden hintereinander ohne Überlappen angeordnet. Ein Viereck in dieser Abbildung entspricht einem lokalen Alignment.

Um den maximalen Weg durch diesen Graphen zu finden, wird ein modifzierter Dijkstra-Algorithmus (Abschnitt 2.3) verwendet. Dieser Algorithmus berechnet den kürzesten Weg von einem Startpunkt zu allen anderen Knoten in einem Graphen. Somit muss der Algorithmus dahingehend verändert werden, dass er anstatt der kürzesten Kante die Kante mit dem größten Gewicht auswählt. Das Ergebnis ist ein Alignment, das sich aus nicht überlappenden lokalen Alignments zusammensetzt.

Bei der Suche nach den lokalen Alignments werden auch Subalignments zugelassen. Subalignments sind lokale Alignments, die sich innerhalb eines anderen optima-

len lokalen Alignments befinden. Diese Variante hat den Vorteil, dass durch diese Subalignments mehr Variabilität beim Zusammensetzen der Zielsequenz entsteht. Allerdings hat sie folgenden Nachteil, der diesen Ansatz scheitern ließ: Weist ein lokales Alignment einen besonders hohen Score auf und liegt der Score des nächsten lokalen Alignments weit darunter, wird vom Algorithmus nur das Alignment mit dem besonders hohen Score und die dazugehörigen Subalignments gefunden. Somit flossen andere lokale Alignments nicht in das Endergebnis ein. Der Score für das Gesamtalignment ist entsprechend schlecht, da nur ein lokales Alignment zum Zusammenbauen verwendet wird.

Aus diesem Grund wird im letzten Ansatz der Smith-Waterman-Algorithmus durch ein Automatenmodell ersetzt, dass auch in BLAST (Abschnitt 2.2.4) verwendet wird.

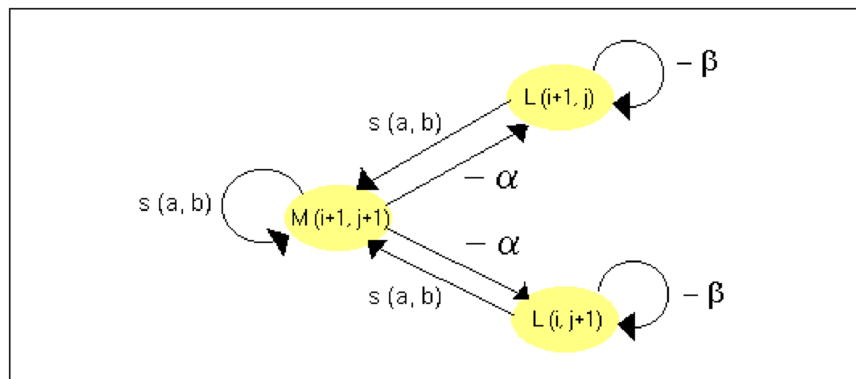


Abbildung 3.2.: Drei-Zustandsmodell

$M_{i+1,j+1}$ beschreibt den Zustand, in dem sich der Automat befindet, wenn es eine Übereinstimmung beider Aminosäuren gibt. Das bedeutet, der Score $s(a,\ b)$ aus einer Substitutionsmatrix ist größer als die Strafe für eine Lücke. Den entgegengesetzten Fall bilden die beiden anderen Zustände. Bei ihnen wird entweder in der Zielseqenz $L_{i,j+1}$ oder in der Quellsequenz $L_{i+1,j}$ eine Lücke eingefügt. In einen dieser Zustände gelangt man, in dem vom Alignmentscore die Strafe α abgezogen wird. Wenn man im Zustand L ist, wird die Lücke erweitert, indem β vom Alignmentscore abgezogen wird oder es wird in den Zustand $M_{i+1,j+1}$ gesprungen, wenn der Score aus der Substitutionsmatrix größer als β ist. [DEKM98]

Es wird eine Liste mit Sonden erstellt, die einen vorgegebenen Grenzwert überschreiten. Mit diesen Sonden wird die Quellsequenz nach Treffern durchsucht. Es werden alle jene Treffer nach links und rechts erweitert, die einen Mindestscore aufweisen. Dieses Erweitern entspricht dem Drei-Zustandsmodell aus Abb. 3.2.

Dieser Algorithmus lässt Lücken zu, denn es wird so lange nach rechts und links erweitert, bis der Score den dritten Grenzwert unterschreitet. Das entstandene Alignment muss eine bestimmte Länge an Aminosäuren haben. Wird dieser erreicht, wird es in die Liste der lokalen Alignments aufgenommen. Sobald alle Alignments gefunden werden, wird mit dem oben beschriebenen Dijkstra-Algorithmus die Zielsequenz aus den lokalen Alignments zusammengesetzt, die den maximalen Score ergeben.

Der letzte Ansatz (s. Abb. 3.3) führte zum Erfolg und wurde in der entstandenen Applikation verwendet.

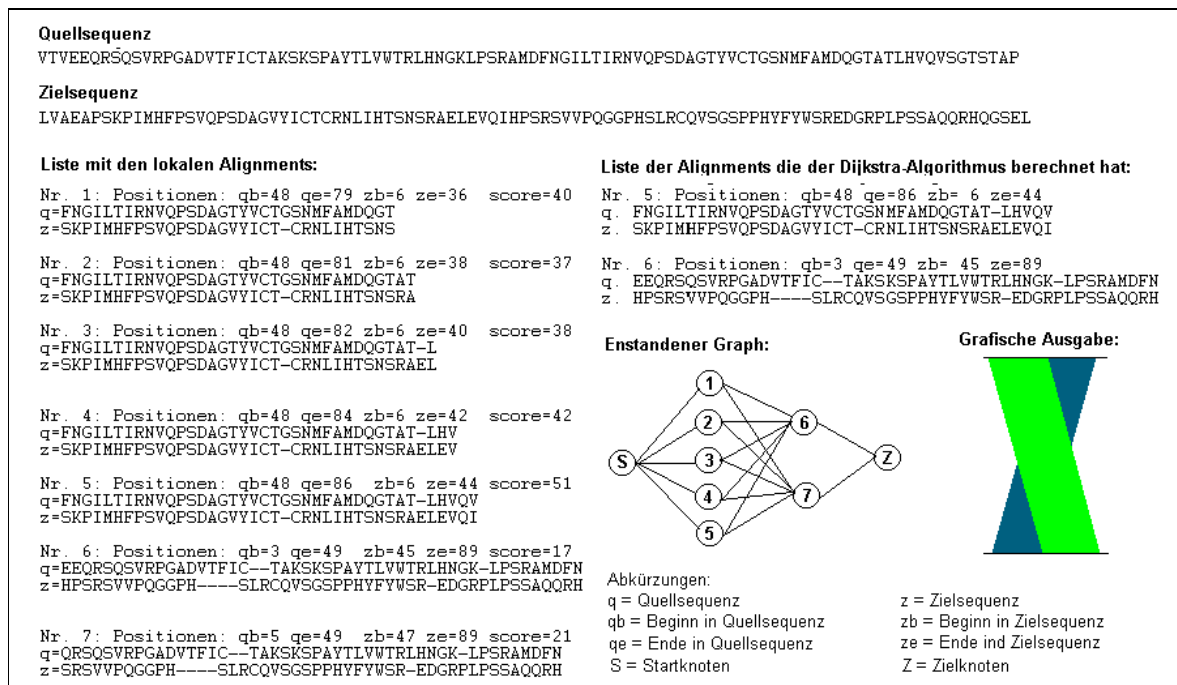


Abbildung 3.3.: Beispiel des Algorithmus

Nachdem der Anwender die Zielsequenz, die Quellsequenz und die benötigten Parameter eingegeben hat, wird eine Liste mit lokalen Alignments erstellt. Diese bilden die Knoten des abgebildeten Graphens. Der abgewandelte Dijkstra-Algorithmus berechnet den maximalen Weg durch den Graphen, der die Knoten 5 und 6 beinhaltet. An der grafischen Ausgabe erkennt man, dass sich beide Alignments kreuzen, was auf eine mögliche Rekombination hinweist.

4. Implementierung

Die Applikation bietet eine grafische Oberfläche. Diese ermöglicht dem Nutzer, zwei zu vergleichende Sequenzen und alle vom Algorithmus verwendeten Parameter zu spezifizieren. Nachdem er die Anfrage abgeschickt hat, werden im Hintergrund mittels des Drei-Zustandsmodells lokale Alignments berechnet. Aus der Liste der lokalen Alignments wird die Zielsequenz zusammengesetzt, so dass der Score maximal wird. Die Reihenfolge wird durch einen veränderten Dijkstra-Algorithmus festgelegt. Das Ergebnis wird dem Nutzer daraufhin textuell sowie grafisch präsentiert. Wie dieser Algorithmus in eine Applikation umgesetzt wurde, wird in diesem Kapitel beschrieben.

4.1. Die Programmiersprache Java

Die Implementierung erfolgte in Java. Die erste Version des **J**ava **D**eveloper **K**it (**JDK** 1.0) wurde 1996 vorgestellt. Der Syntax lehnt sich stark an C und C++ an, aber auf komplexe und fehlerträchtige Elemente dieser Sprachen wurde verzichtet. Neu dagegen sind Multithreading, strukturierte Ausnahmebehandlung und eingebaute grafische Fähigkeiten. Weiterhin verfügt Java über einen Garbage Collector, der automatisch nicht mehr benötigten Speicherplatz frei gibt. Er läuft im Hintergrund und sucht nach Objekten, die nicht mehr referenziert werden. Für das vorliegende Programm ist das besonders wichtig, da es sehr speicherintensiv ist. Der wichtigste Vorteil von Java liegt darin, dass die Programme auf vielen Plattformen funktionieren, ohne neu kompiliert, neu entworfen oder neu programmiert werden zu müssen. Zum Ausführen eines Java-Programms werden lediglich eine

virtuelle **M**aschine (**VM**) und die erforderliche Laufzeitumgebung benötigt, da eine Zwischensprache verwendet wird, die durch die virtuelle Maschine interpretiert wird. [Krü99]

Java ist eine objektorientierte Sprache. Bei der objektorientierten Programmierung werden Datenmengen und eine festgelegte Anzahl zugelassener Operationen auf diese Daten in Objekten zusammengefasst. Objekte sind gekapselte Datenstrukturen, die diese Datenkomponenten und alle zu ihrer Bearbeitung dienenden Funktionen, genannt Methoden, beinhalten. Diese Objekte werden in sogenannten Klassen definiert. Somit ist eine Klasse der Typ des Objekts und beschreibt Typ und Aufbau der Datenkomponenten und Methoden. Klassen können aus bereits definierten Klassen abgeleitet werden, wobei die abgeleitete Klasse automatisch alle Eigenschaften der Oberklasse erbt, d.h. die neue Klasse übernimmt dabei alle Datenkomponenten und Methoden der alten. Darüberhinaus können eigene Komponenten hinzugefügt oder modifiziert werden. [SvG00]

Um die grafische Oberfläche zu erstellen, wurde Java *Swing* verwendet. *Swing* bildet eine Erweiterung des Java **A***bstract* **W***indow* **T***oolkit* (**AWT**). Mit der Hilfe von **AWT** können grafische Komponenten, wie z.B. Fenster, Knöpfe oder Eingabehilfen, definiert werden. Im Gegensatz zum **AWT** zeichnet und verwaltet *Swing* die Benutzerschnittstellen-Elemente in einem Betriebssystem selbst. Somit bleibt die Gestaltung ganz dem Entwickler überlassen. [Mey98]

Die Oberfläche teilt sich in drei Ansichten. Den Rahmen bildet die Klasse *EingabeGuiSwing*, die von der Klasse *javax.swing.JFrame* abgeleitet ist. Diese verwaltet mit Hilfe des Layoutmanagers *java.awt.CardLayout* die Ansichten Eingabemodus (*InputPanel*), Textmodus (*ResultTextPanel*) und Grafikmodus (*ResultGraphicPanel*) (Klassendiagramm Abb. 4.1). Diese werden in den folgenden Abschnitten näher erkärt.

4.2. Eingabemodus

Die Klasse *InputPanel* besitzt zwei Texteingabeflächen (*javax.swing.JTextArea*), eine für die Quellsequenz und die andere für die Zielsequenz. Die Eingabeflächen

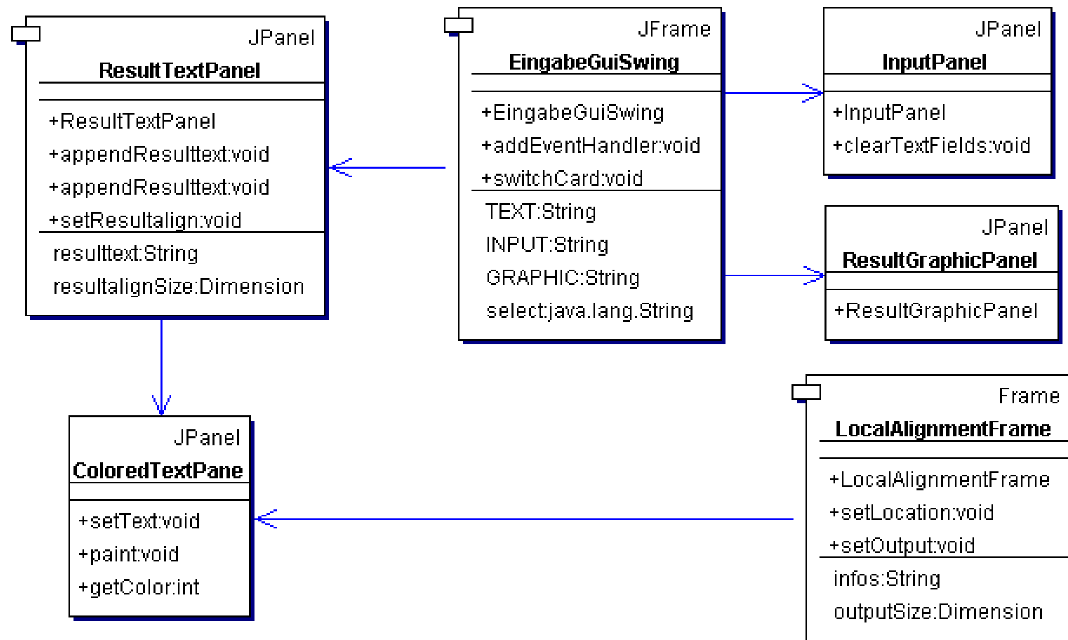


Abbildung 4.1.: Klassendiagramm der grafischen Oberfläche

Eine Einführung in die **U**nified **M**odelling **L**anguage (**UML**) und eine Klassendiagramm, dass alle implementierten Klassen enthält befinden sich im Anhang C.

befinden sich in einem scrollbaren Bereich (*javax.swing.JScrollPane*), der nur zu sehen ist, wenn die Eingabe größer wird als das Sichtfeld. Darüberhinaus ist eine Auswahlbox vorhanden, die von Java durch die Klasse *javax.swing.JComboBox* bereitgestellt wird. Der Anwender hat somit die Möglichkeit, die Substitutionsmatrix (BlosumN, Identität, PAM500, VT250 (Abschnitt 2.2.2)), die verwendet werden soll, auszuwählen. Außerdem kann er verschiedene Parameter (s. Tab. 4.1) vorgeben, die jeweils in einem *javax.swing.JTextField* erfasst werden. Diese Eingabefelder werden mit Objekten der Klasse *javax.swing.JLabel* beschriftet.

In der Abb. 4.2 sind vier Knöpfe zu sehen, denen jeweils ein Listener zugeordnet ist. Wenn ein Knopf gedrückt wird, wird ein Ereignis (Event) ausgelöst, das vom Listener verarbeitet wird. Mit dem Knopf *example* (*javax.swing.JButton*) kann sich der Anwender Beispieldaten anzeigen lassen oder mit *reset* alle Eingaben löschen. Der Knopf *submit* schickt eine Anfrage ab und der letzte Knopf *quit* beendet die Applikation.

Strafe für	
α	das Eröffnen einer Lücke.
β	das Erweitern einer bereits existierenden Lücke.
γ	das Finden einer Rekombination.
Grenzwerte	
threshold 1	Score, den eine Sonde mindestens haben muss.
threshold 2	Das Finden einer Sonde in der Quellsequenz wird als Treffer bezeichnet, wenn der Score für diesen Stelle die Differenz von maximalem Score und diesem Grenzwert nicht unterschreitet. Der maximale Score an dieser Stelle bedeutet, dass alle drei Aminosäuren der Sonde identisch sind.
threshold 3	Der maximale Score, den das lokale Alignment bis dahin erreicht hat, minus diesen Grenzwert ergibt das Abbruchkriterium. Das Alignment wird nur so lange nach beiden Seiten verlängert, bis der Score unter diesen Abbruchwert fällt.

Tabelle 4.1.: Benutzerdefinierte Parameter

Abbildung 4.2.: Startansicht des Programms (Eingabemodus)

4.3. Eine Anfrage bearbeiten

Eine Anfrage wird bearbeitet sobald der Anwender den Knopf *submit* drückt. Bevor zwei Sequenzen verglichen werden können, müssen die Eingaben überprüft werden. Aus den Sequenzen werden zunächst Leerzeichen und Zeilenumbrüche entfernt. Außerdem werden sie nach nicht erlaubten Zeichen, z.B. Sonderzeichen und Zahlen, durchsucht. Auch die Buchstaben *I, O, U* sind unzulässig, da sie keine Aminosäuren darstellen. Wenn die VT-Matrix verwendet wird, sind zusätzlich die Zeichen B, X, Z und '*' auszuschliessen, da diese in der Matrix nicht auftreten. Die Parameterwerte entsprechen ganzen Zahlen. Sollte die Eingabe fehlen oder ein anderes Format verwendet werden, werden die Fehler durch eine Ausnahmebehandlung (Exception) abgefangen. Beim Auslösen einer Exception wird eine entsprechende Fehlermeldung angezeigt. Der Text der Fehlermeldung wird während der Prüfungen dynamisch erstellt. Je nachdem, welche Fehler auftreten, wird der Nutzer durch ein extra Fehlerfenster darauf hingewiesen. Hierbei wird eine von *Swing* vordefinierte Klasse *javax.swing.JOptionPane* verwendet. Diese ermöglicht es, ein Fehlerfenster mit eigenem Text auszugeben (s. Abb. 4.3).

```
JOptionPane.showMessageDialog(null,          → übergeordnetes Fenster
                              text,          → String der anzuzeigende Fehlermeldung
                              "Error",       → Name des Fensters
                              JOptionPane.ERROR_MESSAGE);  → definiert den Stil der Mitteilung
```

Abbildung 4.3.: Aufruf eines Fehlerfensters

Wenn keine Fehler gefunden wurden, wird die Suche nach den lokalen Alignments gestartet. Das übernimmt die Klasse *LocalAlignmentBlast*. Es wird ein Vektor (*java.util.Vector*) mit Sonden der Zielsequenz aufgestellt. Ein Element des Vektors besteht aus einem Objekt der Klasse *WordInfo*. In *WordInfo* werden die String-Repräsentation der Sonde sowie deren Anfang und Ende in der Zielsequenz gespeichert. Mit diesen Sonden wird in der Quellsequenz nach passenden Treffern gesucht. Sobald ein Treffer gefunden wurde, wird das Alignment in beide Richtungen verlängert, bis der Grenzwert unterschritten wird. Dieser Grenzwert bildet sich aus

der Differenz von dem maximalen Score, den das Alignment bis dahin erreicht hat, abzüglich des Wertes für *threshold 3* (s. Tab. 4.1). Das Erweitern geschieht mit Hilfe des Drei-Zustandsmodell aus Abb. 3.2, das in Kapitel 3 beschrieben wurde.

Die Daten, die zu einem Alignment gehören, werden alle in der Klasse *AlignmentInfos* gespeichert. Zu den Daten gehören Anfang und Ende des Alignments in beiden Sequenzen, das Alignment als Textelement, der dazugehörige Score und eine Liste mit möglichen Nachfolgern im Graphen. Die Nachfolger wurden zuerst in einem zweidimensionalen Feld (Adjazensmatrix) gepeichert. Die Alignments, die die Knoten eines Graphen darstellen, werden nummeriert von $1, 2, ..|V|$. Es wurde ein $|V| * |V|$-Feld angelegt und jeder Eintrag $a_{i,j}$ der Adjazenzmatrix ergibt sich aus der folgenden Formel:

$$a_{i,j} = max \begin{cases} \mathit{1} & \mathit{wenn\ es\ eine\ Kante\ zwischen\ i\ und\ j\ gibt} \\ \mathit{0} & \mathit{sonst} \end{cases}$$

Diese Variante hat sich bei einer großen Anzahl von lokalen Alignments als zu speicherintensiv erwiesen, da die Matrix nur wenige 1 enthielt. Deshalb wird jetzt jedem Alignment eine Liste mit den Nummern der möglichen Nachfolger und den dazugehörigen Gewichten zugeordnet.

Nachdem die Liste mit lokalen Alignments erstellt wurde, wird die Methode *bestPath()* der Klasse *DijkstraAlgo* gestartet. Diese findet die Reihenfolge der Alignments, so dass der Score am Ende maximal ist. Danach werden die Ziel- und Quellsequenz zusammen mit den Alignments ausgegeben, die die Methode der Klasse *DijkstraAlgo* ermittelt hat. Das geschieht im Ausgabemodus und wird im nächsten Abschnitt beschrieben.

4.4. Ausgabemodus

Nachdem die Anfrage bearbeitet wurde, wird mittels des CardLayouts auf den Textmodus (Klasse *ResultTextPanel*) umgeschaltet. Dieses Feld zeigt die Ergebnisse textuell an (s. Abb. 4.4). Textuell bedeutet, dass beide Sequenzen übereinander stehen und dass der Anfang und das Ende eines lokalen Alignments durch Nummern und

dazwischen durch Unterstriche gekennzeichnet sind.

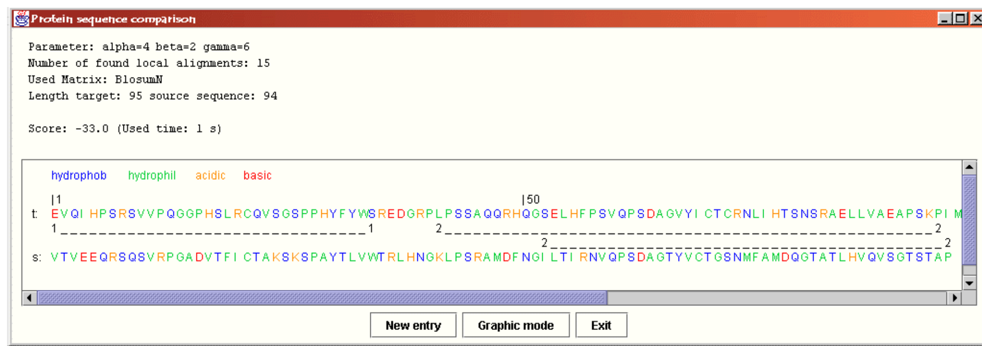


Abbildung 4.4.: Oberfläche im Textmodus

Im oberen Bereich werden die einzelnen Parameter, die ausgewählte Matrix, die Anzahl der gefundenen lokalen Alignments, die Längen beider Sequenzen und der Score für den Vergleich sowie die Bearbeitungszeit in einer *javax.swing.JTextArea* ausgegeben.

Im mittleren Teil des Feldes werden die Sequenzen farbig dargestellt. Das geschieht durch die Klasse *ColoredTextPane*. Diese ist von *javax.swing.JTextPane* abgeleitet. Die Einteilung der Aminosäuren erfolgt entsprechend der Tab. 2.2 nach hydrophil (blau), hydrophob (grün), sauer (rot) und basisch (gelb). Auch hier ist ein scrollbarer Bereich eingebettet, der sich nach der Sequenzlänge richtet.

Im Textmodus hat der Anwender die Möglickeit, mit dem Knopf *New entry* entweder eine komplett neue Eingabe zu machen oder die letzte zu modifizieren. Der Knopf *Graphic mode* schaltet in den Grafikmodus um.

Abbildung 4.5.: Oberfläche im Grafikmodus

Im Grafikmodus (Abb. 4.5) werden die Zielsequenz und die Quellsequenz als Linie dargestellt. Die lokalen Alignments werden durch unterschiedlich farbig gefüllte Polygone repräsentiert. So kann der Anwender die Lage der einzelnen Alignments sehen und mögliche Rekombinationen erkennen. Die Farbe und die Koordinaten eines Polygons sowie das dazugehörige Alignment werden in einem Objekt der Klasse *AlignmentGraphic* gespeichert. So stehen die Informationen zu einem Polygon schnell zur Verfügung. Das wird benötigt, wenn der Anwender mit einem Doppelklick auf ein Polygon drückt. Daraufhin wird ein neues Fenster geöffnet, das das entsprechende lokale Alignment zu diesem Polygon anzeigt: die genauen Positionen in beiden Sequenzen, der Score und das Alignment textuell erscheinen. Für dieses Fenster wurde die Klasse *LocalAlignmentFrame* (Abb. 4.6) entwickelt, die von *javax.swing.JFrame* abgeleitet ist. Sie beinhaltet ein Objekt der Klasse *javax.swing.JTextArea* zum Ausgeben der allgemeinen Informationen, ein Objekt der Klasse *ColoredTextPane* zur textuellen Anzeige des gewünschten lokalen Alignments und einen *javax.swing.JButton* zum Schließen des Fensters.

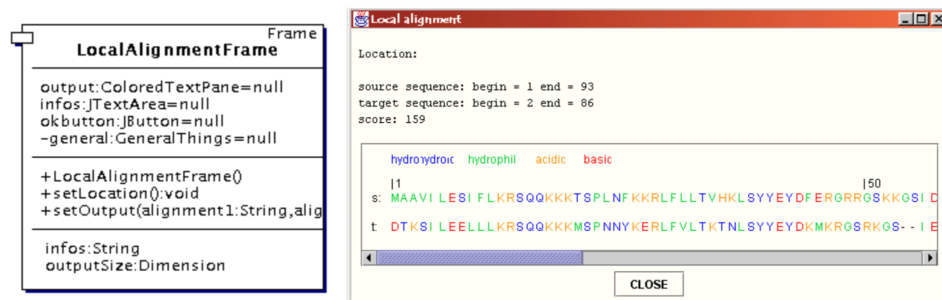


Abbildung 4.6.: UML-Diagramm der Klasse *LocalAlignmentFrame* und ein Beispiel

Die Darstellung der Ergebnisse lehnt sich an das Programm von Varré et al. [VDR99] an, das die Unähnlichkeit zweier Sequenzen aufgrund von segmentbasierenden Ereignissen bestimmt. Diese Programm bestimmt die Distanz zwischen zwei Sequenzen, indem die eine Sequenz in die andere durch Einfügen, Enfernen oder Ersetzen von ganzen Sequenzstücken überführt wird.

5. Auswertung

In den letzten Wochen der Diplomarbeit wurden einige Testdurchläufe mit dem Programm durchgeführt. Aber diese sind bei weitem noch nicht ausreichend, um genaue Aussagen über Schwächen und Stärken dieses Programm zu machen. Es wurde der Einfluss der Parameter untersucht und ein Vergleich mit dem Programm BLAST durchgeführt.

5.1. Wahl der Parameter

Der Anwender kann folgende Parameter festlegen:

1. Eine Strafe für das Eröffnen (α) bzw. für das Erweitern einer Lücke (β).
2. Eine Strafe für eine Rekombination (γ).
3. Drei Grenzwerte zum Finden von lokalen Alignments (*threshold 1-3)*.
4. Die Mindestlänge der lokalen Alignments.

Im Allgemeinen spielt die verwendete Substitutionsmatrix eine große Rolle. Denn die Werte der Parameter sollten auf die entsprechende Matrix abgestimmt sein.
Die Summe der Strafen für das Eröffnen (α) bzw. Erweitern (β) einer Lücke sollte immer größer oder gleich als der niedrigste Wert in der verwendeten Substitutionsmatrix sein [DEKM98]. Wenn der Anwender allerdings nur Alignments ohne Lücken betrachten möchte, sollte der Parameter zum Eröffnen einer Lücke kleiner als der niedrigste Wert der Substitutionsmatrix sein. Dann wird der Zustand, dass eine Lücke aufgemacht wird, nie erreicht, da die Werte in der Substitutionsmatrix im-

mer größer sind. Außerdem darf die Strafe für das Eröffnen einer Lücke nicht zu niedrig sein. Denn bei jeder Eröffnung einer Lücke werden aus einem angefangenen Alignment zwei. In dem einem Alignment wird die Lücke in der Quellsequenz eingefügt und in dem anderen in der Zielsequenz. Wenn zu viele Alignments gebildet werden, besteht die Gefahr, dass der Speicherbereich nicht ausreicht und ein "out of memory"-Fehler auftritt. Beim Starten des Programms werden standardmäßig 64 MB des Arbeitsspeichers benutzt, was besonders bei langen Sequenzen oft nicht ausreichend ist. Wenn dieser Fehler auftritt, kann man über die Option -*XmxGroesse* den Speicher erweitern.

Beispiel für 128 MB: java -Xmx128m eml.seqal.SequenceAlignment

Die Höhe der Strafe für das Erweitern einer Lücke entscheidet über die Länge der Lücke. Wenn es viele Werte in der Substitutionsmatrix gibt, die unter diesem Parameter liegen, kann die Lücke unter Umständen sehr lang werden. Einige Beispiele für diese Parameter sind in der Tab. 5.1 zu finden.

Substitutionsmatrix (niedrigster Wert)	Eröffnen	Erweitern einer Lücke	Ergebnisse
BlosumN (-7)	-5	-2	wenige und nicht zu lange Lücken
	-8	-2	keine Lücken
Pam500 (-9)	-7	-2	wenige und nicht zu lange Lücken
	-10	-2	keine Lücken
VT250 (-3)	-2	-2	wenige und nicht zu lange Lücken
	-4	-2	keine Lücken

Tabelle 5.1.: Verschiedene Strafen für Lücken

Die Parameter wurden immer in Abhängigkeit des niedrigsten Wertes in der Substitutionsmatrix gewählt. Die Sequenzen hatten eine Länge von rund 1400 Aminosäuren. Es ist jeweils ein Beispiel für einen Versuch mit und ohne Lücken aufgeführt. Der zu benutzende Speicher wurde auf 128 MB eingestellt.

Die letzte Strafe, die bei Rekombinationen benutzt wird, wirkt sich auf den Gesamtscore aus. Je höher sie ist, desto schlechter ist der Score, wenn viele Rekombinationen auftreten.

Die letzen drei Parameter beeinflussen das Finden der lokalen Alignments. Der erste Grenzwert (*threshold 1*, Tab. 5.2) sollte in Abhängigkeit der Hauptdiagonalen einer Substitutionsmatrix gewählt werden. Er gibt den Score an, den ein Tripel aus der Quellsequenz mindestens haben muss, um in die Liste der Sonden aufgenommen zu werden. Am wichtigstens sind die Tripel, die den höchstmöglichen Score erreichen, d.h. alle drei Aminosäuren sind identisch. Die Werte für Identitäten befinden sich auf der Hauptdiagonalen in einer Substitutionsmatrix. Deshalb sollte dieser Grenzwert zwischen dem Dreifachen des niedrigsten und des höchsten Wertes der Hauptdiagonalen liegen, je nachdem wie viele lokale Alignments gefunden werden sollen. Je niedriger dieser Wert ist, desto mehr Sonden werden benutzt, um lokale Alignments zu finden und desto mehr Speicherplatz und Zeit wird vom Programm benötigt.

Substitutions-matrix	Intervall der Hauptdiagonale	*threshold 1*	Kommentar
BlosumN	-2 bis 11	18	Wenn man diesen Wert wählt, kann ein Treffer auch einen Mismatch enthalten.
Pam500	0 bis 34	20	34 (W) tritt einmal auf und der nächste Wert ist 22 (C). Die anderen Werte sind viel niedriger. Deshalb sollte der Wert nicht zu hoch sein, da sonst nur die Tripel verwendet werden, die W oder C enthalten.
VT250	2 bis 11	12	Da die meisten Werte zwischen 3 und 5 auf der Hauptdiagonalen liegen, bekommt man so eine gute Trefferanzahl.

Tabelle 5.2.: Beispiele für *threshold 1*

Der Grenzwert ist außerdem abhängig von der Länge der Zielsequenz. Je länger die Zielsequenz ist, um so wahrscheinlicher ist es, dass sie alle möglichen Kombinationen von Aminosäuren enthält. Somit muss der Grenzwert entsprechend dem Problem angepasst werden, da sonst zu viele Sonden gefunden werden.

Ähnlich wirkt der zweite Grenzwert (*threshold 2*). Denn es werden alle die Treffer in der Zielsequenz nach links und rechts erweitert, deren Score die Differenz von

dem höchst möglichen Score (Identität) und diesem Grenzwert nicht unterschreitet. Je größer dieser ist, umso mehr Treffer werden erweitert und somit wird wieder mehr Speicherplatz gebraucht. Er sollte ebenso wie *threshold 1* in Abhängigkeit der Hauptdiagonale gewählt werden. Wenn dieser den Wert 0 bekommt, werden nur die Treffer erweitert, deren Aminosäuren alle identisch sind.
Die Längen der lokalen Alignments verändern sich proportional zu dem dritten Grenzwert (*threshold 3*, Tab. 5.3). Je größer dieser angegeben wird, desto länger wird das lokale Alignment. Denn ein Treffer wird so lange nach links und rechts erweitert bis der Score unter die Differenz von dem maximalen Score und diesem Grenzwert fällt. In diesem Fall entspricht der maximale Score dem höchsten Score, den das Alignment bis dahin durch das Erweitern erreicht hat. Allerdings gibt es einen Nachteil, wenn der Wert zu groß gewählt wird, kann es passieren, dass das Alignment zu viele Lücken enthält und der Gesamtscore sehr schlecht wird. Sobald die lokalen Alignments länger werden, steigt die Anzahl der gefundenen Alignments automatisch mit an, da mehr Alignments die Mindestlänge überschreiten. Folglich wird wieder mehr Speicherplatz und Zeit vom Programm gebraucht.

5.2. Vergleich mit dem Basic Local Alignment Search Tool

BLAST arbeitet mit einem heuristischen Suchverfahren, um die lokalen Alignments zu finden. In dem entwickelten Programm wurde dieser Algorithmus auch verwendet. Allerdings berücksichtigt das enstandene Programm nicht nur Punktmutationen, sondern kann zusätzlich mögliche Rekombinationen erkennen. Folglich sollten beide Programme bei Sequenzen, die die gleiche Domänenstruktur aufweisen, einen hohen Score und damit eine starke Ähnlichkeit feststellen. Bei Sequenzen, die zwar die gleichen Domänen jedoch in verschiedenen Anordnungen besitzen, sollte nur das hier entwickelte Programm die Ähnlichkeit finden. Diese Ähnlichkeit sollte folglich auch größer sein als bei dem Vergleich mit einem Protein, das nur eine der betrachteten Domänen enthält. Das wird im letzten Versuch ausprobiert. Um die Erwartungen zu überprüfen, wurden die vier Proteine aus Abb. 5.1 betrachtet. Eine ausführliche Aufstellung der Ergebnisse befindet sich im Anhang B.

threshold 3	Anzahl der gefundenen Alignments	Anzahl der benutzten Alignments	Zeit min	Score
0	266	5	0:04	2268
10	208	2	0:48	2308
30	368	2	0.58	2238
40	430	2	1:00	2228
60	483	2	1:15	2148
90	457	1	1.49	2101

Tabelle 5.3.: Wirkung von *threshold 3*

Es wurden zwei Proteine mit je einer Länge von rund 500 Aminosäuren betrachtet. Die Parameter wurde wie folgt festgelegt und threshold 3 wurde verändert:

Substitutionsmatrix:	BlosumN	Rekombination:	6
Eröffnen einer Lücke:	5	Erweitern einer Lücke:	2
threshold 1:	10	*threshold 2*:	5
Mindestlänge eines Alignments:	30		

Die betrachteten Proteine sind ähnlich, da sie die gleiche Domänenstruktur aufweisen. Deshalb findet der Algorithmus im letzten Versuch ein Alignment über die gesamte Sequenz.

In allen Versuchen wurde in BLAST [NB01] mit dem Protein SERINE/THREONINE-PROTEIN KINASE A gesucht. Dabei wurde die Matrix
PAM70 verwendet, für das Eröffnen einer Lücke wurde der Wert -6 und für das Erweitern -2 gewählt. Das Protein SERINE/THREONINE-PROTEIN KINASE B hat wie erwartet eine starke Ähnlichkeit von 93 % mit der Suchsequenz. Das gefundene Alignment verlief in beiden Sequenzen von Position 1 bis 431 und somit über die gesamte Sequenz.
Mit der entwickelten Applikation wurde ebenso ein Alignment über die gesamte Sequenz gefunden. Es wurde die Matrix PAM250 mit den Parametern $\alpha = -7$ und $\beta = -2$ benutzt. Damit ist die erste Erwartung eingetroffen, denn beide Algorithmen lieferten bei gleicher Domänenstruktur gleiche Ergebnisse.
Das dritte Protein SERINE/THREONINE-PROTEIN KINASE SHK2 weist laut BLAST eine Ähnlichkeit von 33 % auf. Dieser Wert ist deshalb viel niedriger,

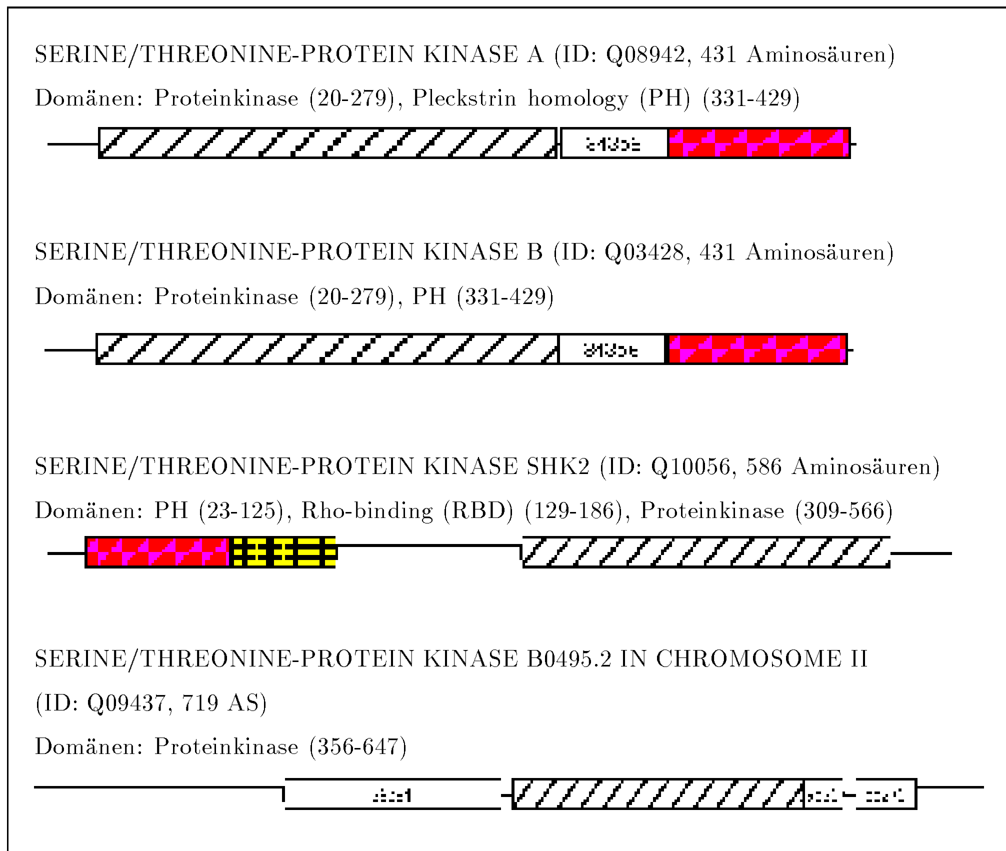


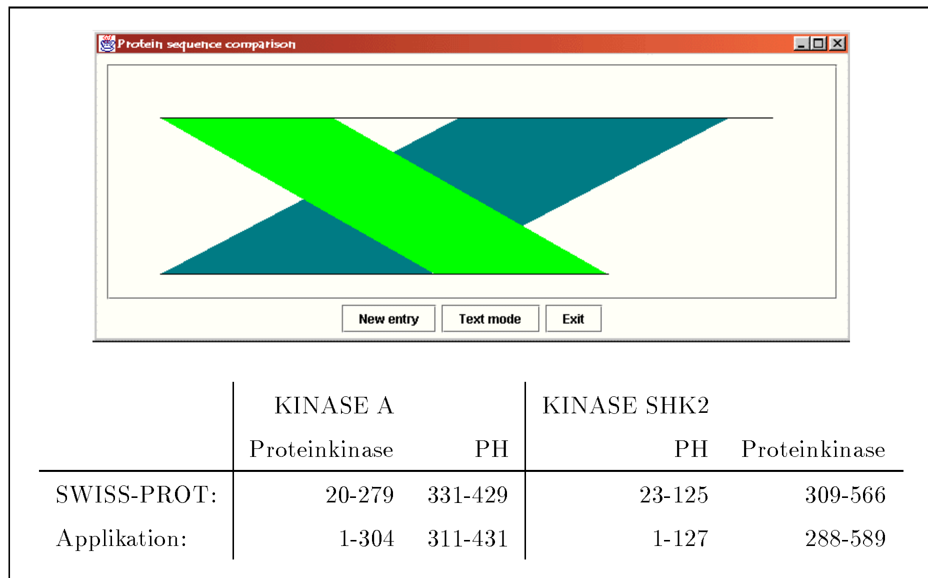
Abbildung 5.1.: Testproteine und ihre Domänenstrukturen

Die ersten beiden Proteine besitzen denselben Domänenaufbau, wohingegen das dritte Protein ebenso diese Domänen beinhaltet, jedoch mit vertauschten Positionen. Das letzte Protein hat nur die Domäne Proteinkinase. Für weitere Informationen über diese Proteine kann man über die ID auf die Datenbank SWISSPROT [SWI01] zugreifen. Die Bilder der Domänenstrukturen sind aus der Datenbank ProDom [Pro01].

da BLAST nur eine der Domänen gefunden hat. Das Alignment beginnt in der Suchsequenz bei 20 und endet bei 189 und in dem anderen Protein bei 308 und bei 474. Die Positionen entsprechen der Domäne Proteinkinase in beiden Proteinen, allerdings ist das Alignment um rund 90 Aminosäuren kürzer.

Im Gegensatz zu BLAST hat das vorliegende Programm zwei Alignments gefunden, die den Domänenpositionen in etwa entsprechen (s. Abb. 5.2).

Um noch deutlicher zu zeigen, dass die entwickelte Applikation zwischen den letzten beiden Proteinen eine stärkere Ähnlichkeit als BLAST festgestellt hat, wurden das Protein SERINE/THREONINE-PROTEIN KINASE B mit dem Protein



	KINASE A		KINASE SHK2	
	Proteinkinase	PH	PH	Proteinkinase
SWISS-PROT:	20-279	331-429	23-125	309-566
Applikation:	1-304	311-431	1-127	288-589

Abbildung 5.2.: Vergleich von SERINE/THREONINE-PROTEIN KINASE A und SERINE/THREONINE-PROTEIN KINASE SHK2

Die Parameter in der Applikation wurden wie folgt gewählt:

$\alpha:$	-6	*threshold 1:*	15	*Matrix:*	PAM250
$\beta:$	-2	*threshold 2:*	10	*Länge:*	30
$\gamma:$	-6	*threshold 3:*	50		

SERINE/THREONINE-PROTEIN KINASE B0495.2 IN CHROMOSOME II verglichen, das nur die Domäne Proteinkinase besitzt. Zwischen diesen beiden zeigte BLAST dieselbe Ähnlichkeit wie zwischen SERINE/THREONINE-PROTEIN KINASE B und SERINE/THREONINE-PROTEIN KINASE SHK2, da BLAST auch hier die Proteinkinase erkannt hat.

Die Ergebnisse der Applikation sehen etwas anders aus. Der Score für den Vergleich von SERINE/THREONINE-PROTEIN KINASE B und SERINE/THREONINE-PROTEIN KINASE B0495.2 IN CHROMOSOME II entsprach der Hälfte des Scores für SERINE/THREONINE-PROTEIN KINASE B und SERINE/THREONINE-PROTEIN KINASE SHK2 (s. Abb. 5.2). Das ist darauf zurückzuführen, dass SERINE/THREONINE-PROTEIN KINASE B und SERINE/THREONINE-PROTEIN KINASE B0495.2 IN CHROMOSOME II nur eine Domäne teilen. Dagegen haben SERINE/THREONINE-PROTEIN KINASE B und SERINE/THREONINE-

PROTEIN KINASE SHK2 zwei gemeinsame Domänen, aber in verschiedenen Reihenfolgen.
Die am Anfang dieses Abschnittes gestellten Erwartungen sind alle eingetroffen. BLAST ermittelte eine starke Ähnlichkeit zwischen den Proteinen mit gleicher Domänenstruktur. Die Applikation hat diese Ähnlichkeit bestätigt und im Gegensatz zu BLAST auch eine stärkere Ähnlichkeit zwischen Proteinen mit gemeinsamen Domänen aber in unterschiedlichen Anordnungen gefunden. Diese unterschiedlichen Anordnungen der Domänen könnten auf Rekombinationsereignisse z.B. in Form von "domain-shuffling" zurückzuführen sein und wurden von dem neuen Algorithmus in den Vergleich miteinbezogen, was mit den existierenden Algorithmen bisher noch nicht möglich gewesen ist.

6. Zusammenfassung

Sequenzvergleiche bieten eine Möglichkeit, Informationen über unbekannte Sequenzen herauszufinden. Denn eine ähnliche Sequenz weist auf eine ähnliche Funktion und Struktur hin. Bei Proteinen entspricht die Sequenz der Reihenfolge der Aminosäuren, aus den die einzelnen Proteine aufgebaut sind. Die kleinste Proteineinheit, die eine unabhängig gefaltete Struktur besitzt, wird als Domäne bezeichnet. Die Domänen kennzeichnen die Funktion eines Proteins und sollten in den Sequenzvergleich einbezogen werden.

Die existierenden Algorithmen vergleichen eine Sequenz allerdings punktuell, d.h. die Sequenzen werden an einer Stelle aneinandergelegt und von dort aus wird immer wieder eine Aminosäure aus der einen Sequenz mit einer Aminosäure aus der anderen verglichen. Damit können Punktmutationen (Einfügen, Entfernen oder Ersetzen von einzelnen Aminosäuren) erkannt und behandelt werden. Zur Bewertung von Punktmutationen stehen Substitutionsmatrizen zur Verfügung. Diese Matrizen besitzen für jede mögliche Kombination von zwei Aminosäuren einen Wert (Score), der die Wahrscheinlichkeit angibt, dass die betrachteten Aminosäuren gegeneinander ausgetauscht worden sind. Aber während der Evolution gab es ebenso Rekombinationen (z.B. "exon-shuffling", "domain-shuffling"), die mit diesen Algorithmen nicht behandelt werden können, da es sich um Ereignisse handelt, die sich auf einen ganzen Abschnitt eines Proteins beziehen und nicht nur auf eine einzelne Aminosäure.

Deshalb ist in dieser Diplomarbeit ein Algorithmus entstanden, der zwei Proteinsequenzen vergleicht und im Gegensatz zu den vorhandenen Algorithmen in der Lage ist, sowohl Punktmutationen als auch mögliche Stellen, an denen Rekombinationen statt gefunden haben könnten, zu zeigen.

Eine der eingegebenen Sequenzen wird als Zielsequenz betrachtet und die andere als Quellsequenz. Das Programm sucht mittels eines an BLAST angelehnten Algorithmus eine Menge von lokalen Alignments. Als erstes wird eine Liste mit Sonden (drei Aminosäuren) aus der Zielsequenz aufgestellt, mit denen in der Quellsequenz nach Treffern gesucht wird, die einen bestimmten Grenzwert überschreiten. Diese Treffer werden nach links und rechts so lange erweitert bis der Score dieses Alignments einen bestimmten Grenzwert unterschreitet. Das Erweitern basiert auf einem Automatenmodell mit drei Zuständen:

1. Score für das alinierte Aminosäurenpaar aus einer Substitutionsmatrix ist größer als die Strafe zum Eröffnen einer Lücke.
2. Eröffnen oder Erweitern einer Lücke in der Zielsequenz.
3. Eröffnen oder Erweitern einer Lücke in der Quellsequenz.

Die gefundenen Alignments werden als Knoten eines Graphen betrachtet. Die Kanten zwischen den Knoten richten sich nach Anfang und Ende der Alignments in der Zielsequenz. Der Anfang eines Alignments muss nach dem Ende des davorliegenden Alignments sein und kein anderes darf dazwischen passen. Das Gewicht der Kanten ergibt sich aus dem Score des nachfolgenden lokalen Alignments, einer Strafe für die Aminosäuren, die sich zwischen den beiden Alignments befinden, und einer Strafe für gefundene Rekombinationen.
Ein abgewandelter Dijkstra-Algorithmus findet den Weg durch den Graphen, auf dem die Summe der Gewichte der Kanten ziwschen den lokalen Alignments maximal ist. Die Alignments, die zu diesem Weg gehören, werden in der Zielsequenz und Quellsequenz textuell und grafisch kenntlich gemacht.
Die Dauer, der Speicherbedarf und die Ergebnisse des Algorithmus hängen stark von den Parametern ab, die durch den Anwender selbst festgelegt werden können.

6.1. Ausblick

Die Applikation kann dahingehend weiterentwickelt werden, dass nicht nur zwei Sequenzen verglichen werden, sondern eine Datenbanksuche bei Eingabe der Zielse-

quenz gestartet wird. Außerdem können sehr leicht weitere Substitutionsmatrizen eingebaut werden. Dazu muss eine neue Klasse mit den entsprechenden Daten angelegt und die Auswahlbox erweitert werden.
Mit diesem Programm können Proteinfamilien aufgrund ihrer Domänenstruktur automatisch eingeteilt werden. Damit steht ein Werkzeug zur Verfügung, das mit der Verwandtschaft von Proteinen, die die gleiche Domänen in unterschiedlichen Anordnungen enthalten, umgehen kann.

A. Proteinsubstitutionsmatrizen

A.1. BlosumN

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	6	-2	-2	-3	-2	-1	-2	-1	-3	-3	-3	-2	-2	-4	-1	1	-1	-4	-4	-1	-3	-2	-1	-7
R	-2	7	-1	-3	-6	0	-2	-4	-1	-5	-4	2	-3	-4	-3	-2	-2	-5	-4	-4	-2	-1	-2	-7
N	-2	-1	7	1	-4	-1	-1	-2	0	-5	-5	-1	-4	-5	-4	0	-1	-6	-4	-4	4	-1	-2	-7
D	-3	-3	1	7	-6	-2	1	-3	-2	-6	-6	-2	-5	-5	-3	-2	-2	-7	-5	-5	4	0	-3	-7
C	-2	-6	-4	-6	9	-5	-7	-5	-6	-2	-3	-5	-3	-3	-5	-2	-2	-5	-4	-2	-5	-6	-4	-7
Q	-1	0	-1	-2	-5	7	1	-4	0	-4	-3	1	-1	-4	-2	-1	-2	-4	-3	-4	-1	4	-2	-7
E	-2	-2	-1	1	-7	1	6	-4	-1	-5	-5	0	-4	-5	-3	-1	-2	-5	-4	-4	0	5	-2	-7
G	-1	-4	-2	-3	-5	-4	-4	6	-4	-6	-6	-3	-5	-5	-4	-1	-3	-5	-6	-5	-2	-4	-3	-7
H	-3	-1	0	-2	-6	0	-1	-4	9	-5	-4	-2	-3	-3	-4	-2	-3	-4	1	-5	-1	-1	-3	-7
I	-3	-5	-5	-6	-2	-4	-5	-6	-5	6	1	-4	1	-1	-5	-4	-2	-4	-3	2	-5	-5	-2	-7
L	-3	-4	-5	-6	-3	-3	-5	-6	-4	1	5	-4	2	0	-5	-4	-3	-4	-3	0	-5	-4	-2	-7
K	-2	2	-1	-2	-5	1	0	-3	-2	-4	-4	6	-2	-4	-2	-1	-2	-6	-4	-4	-1	0	-2	-7
M	-2	-3	-4	-5	-3	-1	-4	-5	-3	1	2	-2	8	-1	-4	-3	-2	-2	-3	0	-5	-3	-2	-7
F	-4	-4	-5	-5	-3	-4	-5	-5	-3	-1	0	-4	-1	7	-5	-4	-3	0	3	-2	-5	-5	-3	-7
P	-1	-3	-4	-3	-5	-2	-3	-4	-4	-5	-5	-2	-4	-5	8	-2	-3	-5	-5	-4	-4	-3	-3	-7
S	1	-2	0	-2	-2	-1	-1	-1	-2	-4	-4	-1	-3	-4	-2	6	1	-4	-3	-3	-1	-1	-1	-7
T	-1	-2	-1	-2	-2	-2	-2	-3	-3	-2	-3	-2	-2	-3	-3	1	6	-5	-3	-1	-2	-2	-1	-7
W	-4	-5	-6	-7	-5	-4	-5	-5	-4	-4	-4	-6	-2	0	-5	-4	-5	11	1	-3	-6	-4	-4	-7
Y	-4	-4	-4	-5	-4	-3	-4	-6	1	-3	-3	-4	-3	3	-5	-3	-3	1	8	-3	-4	-4	-3	-7
V	-1	-4	-4	-5	-2	-4	-4	-5	-5	2	0	-4	0	-2	-4	-3	-1	-3	-3	5	-5	-4	-2	-7
B	-3	-2	4	4	-5	-1	0	-2	-1	-5	-5	-1	-5	-5	-4	-1	-2	-6	-4	-5	4	1	-2	-7
Z	-2	-1	-1	0	-6	4	5	-4	-1	-5	-4	0	-3	-5	-3	-1	-2	-4	-4	-4	1	4	-2	-7
X	-1	-2	-2	-3	-4	-2	-2	-3	-3	-2	-2	-2	-2	-3	-3	-1	-1	-4	-3	-2	-2	-2	-2	-7
*	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	1

Quelle: http://eta.embl-heidelberg.de:8000/misc/mat/

A.2. PAM500

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	1	-1	0	1	-2	0	1	1	0	0	-1	0	-1	-3	1	1	1	-6	-3	0	1	0	0	-9
R	-1	5	1	0	-4	2	0	-1	2	-2	-2	4	0	-4	0	0	0	4	-4	-2	0	1	0	-9
N	0	1	1	2	-3	1	1	1	1	-1	-2	1	-1	-4	0	1	0	-5	-3	-1	1	1	0	-9
D	1	0	2	3	-5	2	3	1	1	-2	-3	1	-2	-5	0	1	0	-7	-5	-1	2	2	0	-9
C	-2	-4	-3	-5	22	-5	-5	-3	-4	-2	-6	-5	-5	-3	-2	0	-2	-9	2	-2	-4	-5	-2	-9
Q	0	2	1	2	-5	2	2	0	2	-1	-2	1	-1	-4	1	0	0	-5	-4	-1	2	2	0	-9
E	1	0	1	3	-5	2	3	1	1	-2	-3	1	-1	-5	0	1	0	-7	-5	-1	2	2	0	-9
G	1	-1	1	1	-3	0	1	4	-1	-2	-3	0	-2	-5	1	1	1	-8	-5	-1	1	1	0	-9
H	0	2	1	1	-4	2	1	-1	4	-2	-2	1	-1	-2	0	0	0	-2	0	-2	1	2	0	-9
I	0	-2	-1	-2	-2	-1	-2	-2	-2	3	4	-2	3	2	-1	-1	0	-5	0	3	-2	-2	0	-9
L	-1	-2	-2	-3	-6	-2	-3	-3	-2	4	7	-2	4	4	-2	-2	-1	-1	1	3	-3	-2	-1	-9
K	0	4	1	1	-5	1	1	0	1	-2	-2	4	0	-5	0	0	0	-3	-5	-2	1	1	0	-9
M	-1	0	-1	-2	-5	-1	-1	-2	-1	3	4	0	4	1	-1	-1	0	-4	-1	2	-1	-1	0	-9
F	-3	-4	-4	-5	-3	-4	-5	-5	-2	2	4	-5	1	13	-4	-3	-3	3	13	0	-4	-5	-2	-9
P	1	0	0	0	-2	1	0	1	0	-1	-2	0	-1	-4	4	1	1	-6	-5	-1	0	1	0	-9
S	1	0	1	1	0	0	1	1	0	-1	-2	0	-1	-3	1	1	1	-3	-3	-1	1	0	0	-9
T	1	0	0	0	-2	0	0	1	0	0	-1	0	0	-3	1	1	1	-6	-3	0	0	0	0	-9
W	-6	4	-5	-7	-9	-5	-7	-8	-2	-5	-1	-3	-4	3	-6	-3	-6	34	2	-6	-6	-6	-4	-9
Y	-3	-4	-3	-5	2	-4	-5	-5	0	0	1	-5	-1	13	-5	-3	-3	2	15	-1	-4	-4	-2	-9
V	0	-2	-1	-1	-2	-1	-1	-1	-2	3	3	-2	2	0	-1	-1	0	-6	-1	3	-1	-1	0	-9
B	1	0	1	2	-4	2	2	1	1	-2	-3	1	-1	-4	0	1	0	-6	-4	-1	2	2	0	-9
Z	0	1	1	2	-5	2	2	1	2	-2	-2	1	-1	-5	1	0	0	-6	-4	-1	2	2	0	-9
X	0	0	0	0	-2	0	0	0	0	0	-1	0	0	-2	0	0	0	-4	-2	0	0	0	0	-9
*	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	-9	1

Quelle: http://eta.embl-heidelberg.de:8000/misc/mat/

A.3. VT250

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	2	-1	0	0	0	0	0	0	-1	0	-1	0	0	-1	0	1	1	-2	-2	0
R	-1	4	0	0	-1	1	0	-1	1	-2	-2	2	-1	-2	0	0	0	-1	-1	-1
N	0	0	3	1	-1	0	1	0	1	-2	-2	1	-1	-2	-1	1	0	-2	-1	-1
D	0	0	1	4	-2	0	2	0	0	-2	-2	0	-2	-3	-1	0	0	-3	-2	-2
C	0	-1	-1	-2	8	-1	-2	-1	-1	-1	0	-1	-1	0	-1	0	0	0	0	0
Q	0	1	0	0	-1	3	1	-1	1	-2	-1	1	-1	-2	0	0	0	-2	-1	-1
E	0	0	1	2	-2	1	3	0	0	-2	-2	1	-1	-3	0	0	0	-2	-2	-1
G	0	-1	0	0	-1	-1	0	5	-1	-2	-2	-1	-2	-3	-1	0	-1	-2	-2	-2
H	-1	1	1	0	-1	1	0	-1	5	-2	-1	0	-1	0	0	0	0	-1	2	-1
I	0	-2	-2	-2	-1	-2	-2	-2	-2	3	2	-2	2	1	-2	-1	0	-1	-1	2
L	-1	-2	-2	-2	0	-1	-2	-2	-1	2	3	-2	2	1	-1	-1	-1	0	0	1
K	0	2	1	0	-1	1	1	-1	0	-2	-2	3	-1	-2	0	0	0	-2	-1	-1
M	0	-1	-1	-2	-1	-1	-1	-2	-1	2	2	-1	4	1	-1	-1	0	-1	-1	1
F	-1	-2	-2	-3	0	-2	-3	-3	0	1	1	-2	1	5	-2	-1	-1	2	3	0
P	0	0	-1	-1	-1	0	0	-1	0	-2	-1	0	-1	-2	5	0	0	-2	-2	-1
S	1	0	1	0	0	0	0	0	0	-1	-1	0	-1	-1	0	2	1	-2	-1	-1
T	1	0	0	0	0	0	0	-1	0	0	-1	0	0	-1	0	1	2	-2	-1	0
W	-2	-1	-2	-3	0	-2	-2	-2	-1	-1	0	-2	-1	2	-2	-2	-2	11	2	-1
Y	-2	-1	-1	-2	0	-1	-2	-2	2	-1	0	-1	-1	3	-2	-1	-1	2	6	-1
V	0	-1	-1	-2	0	-1	-1	-2	-1	2	1	-1	1	0	-1	-1	0	-1	-1	2

Quelle: http://www.dkfz-heidelberg.de/tbi/people/tmueller/model/index.html

B. Testergebnisse

Basis Local Alignment Search Tool (BLAST)

Eingabesequenz (Query): SERINE/THREONINE-PROTEIN KINASE A (Q08942)

Parameter:

Substitutionsmatrix:	PAM90
Eröffnen eineer Lücke:	-6
Erweitern einer Lücke:	-2

Entwickelte Applikation

Zielsequenz: SERINE/THREONINE-PROTEIN KINASE A (Q08942)

Parameter:

Eröffnen einer Lücke:	−6	*threshold 1:*	15	*Matrix:*	PAM250
Erweitern einer Lücke:	−2	*threshold 2:*	10	*Länge:*	30
Rekombination:	−6	*threshold 3:*	50		

SERINE/THREONINE-PROTEIN KINASE B (Q03428)

BLAST: Score = 2210 Expect = 0.0 Identities = 402/431 (93%)

```
Query: 1   MAEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEAYVAESVEDGSLCVAKVMDLSKMSQRD 60
           M+EPFSTILGTDGSGGRCKYLNKGIVGLGSYGE YVAE VEDGSLCVAKVMDLSKMS+RD
Sbjct: 1   MSEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEGYVAERVEDGSLCVAKVMDLSKMSRRD 60

Query: 61  KRYAQSEIKCLANCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKLRGSGDARYFQ 120
           KRYAQSEIK   NCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIK  G+GDARYFQ
Sbjct: 61  KRYAQSEIKYPTNCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKPWGTGDARYFQ 120

Query: 121 EHEXXXXXXXXXXXXDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV 180
           EHE            DYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV
Sbjct: 121 EHEALFLFLQLCLALDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV 180

Query: 181 ASTFCGTPYYLAPELWNNKRYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT 240
           ASTFCGTPYYLAPELWNN RYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT
Sbjct: 181 ASTFCGTPYYLAPELWNNLRYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT 240

Query: 241 YAPLPDSFSSEFKRVVDGILVADPNDRPSVREIFQIPYINKGLKLFVQALKKNERISDSV 300
           YAPLPDSFSSEFKRVVDGILVADPNDRPSVRE FQIPYINKGLKLFVQALKKNERI DSV
Sbjct: 241 YAPLPDSFSSEFKRVVDGILVADPNDRPSVRENFQIPYINKGLKLFVQALKKNERILDSV 300

Query: 301 KEVLVTQVSEILSSEVSPDAHRFLVSQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQ 360
           KEVLV+QVSEILSSEVSPDAHRFL SQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQ
Sbjct: 301 KEVLVSQVSEILSSEVSPDAHRFLESQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQ 360

Query: 361 LILTDDEEGNNPKGLNLEQVQGACPVPHSTAKRDFVFALNTVGGKGMWFQAVSHGDMEMW 420
           LILTDDEEGNNPKGLNLEQVQGACPVP+STAKRDFVFALNTVGG+GMWFQAVSHGDMEMW
Sbjct: 361 LILTDDEEGNNPKGLNLEQVQGACPVPYSTAKRDFVFALNTVGGEGMWFQAVSHGDMEMW 420

Query: 421 VHAIQRGIGVA 431
           VHAIQRGIGVA
Sbjct: 421 VHAIQRGIGVA 431
```

Applikation: Score = 1981

```
Z: 1    MAEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEAYVAESVEDGSLCVAKVMDLSKMSQRD 60
Q: 1    MSEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEGYVAERVEDGSLCVAKVMDLSKMSRRD 60

Z: 61   KRYAQSEIKCLANCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKLRGSGDARYFQ 120
Q: 61   KRYAQSEIKYPTNCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKPWGTGDARYFQ 120

Z: 121  EHEALFLFLQLCLALDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV 180
Q: 121  EHEALFLFLQLCLALDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV 180

Z: 181  ASTFCGTPYYLAPELWNNKRYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT 240
Q: 181  ASTFCGTPYYLAPELWNNLRYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT 240

Z: 241  YAPLPDSFSSEFKRVVDGILVADPNDRPSVREIFQIPYINKGLKLFVQALKKNERISDSV 300
Q: 241  YAPLPDSFSSEFKRVVDGILVADPNDRPSVRENFQIPYINKGLKLFVQALKKNERILDSV 300

Z: 301  KEVLVTQVSEILSSEVSPDAHRFLVSQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQ 360
Q: 301  KEVLVSQVSEILSSEVSPDAHRFLESQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQ 360

Z: 361  LILTDDEEGNNPKGLNLEQVQGACPVPHSTAKRDFVFALNTVGGKGMWFQAVSHGDMEMW 420
Q: 361  LILTDDEEGNNPKGLNLEQVQGACPVPYSTAKRDFVFALNTVGGEGMWFQAVSHGDMEMW 420

Z: 421  VHAIQRGIGVA 431
Q: 421  VHAIQRGIGVA 431
```

SERINE/THREONINE-PROTEIN KINASE SHK2 (Q10056)

BLAST: Score = 216 Expect = 1e-16 Identities = 71/210 (33%)

```
Query: 20  YLN-KGIVGLGSYGEAYVAESVEDGSL----CVA-KVMDLSKMSQ-RDKRYAQSEIKCLA 72
           Y N K  +G G+ G  Y+A+ V    L     VA K +DL    Q R K    +EI  +
Sbjct: 308 YFNVKHKLGQGASGSVYLAKVVGGKQLGIFDSVAIKSIDL--QCQTR-KELILNEITVMR 364

Query: 73  NCNHPNIIRYIEDHEENDR-LLIVMEFADSGNLDEQI---KLRGSGDARYFQEHEXXXXX 128
              HPNI+ Y++     +R L +VME+ ++G+L + I   KL    +A      +
Sbjct: 365 ESIHPNIVTYLDSFLVRERHLWVVMEYMNAGSLTDIIEKSKL---TEA------QIARIC 415

Query: 129 XXXXXXXDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVVASTFCGTP 188
                   ++H+  ++HRDIKS NVLL ++G +K+ DFGF  +  +  +  V  T  GTP
Sbjct: 416 LETCKGIQHLHARNIIHRDIKSDNVLLDNSGNIKITDFGFCARLSNRTNKRV--TMVGTP 473

Query: 189 YYLAPELWNNKRYNKKADVWSLGVLLYEIM 218
           Y++APE+     Y  K D+WSLG+++ E M
Sbjct: 474 YWMAPEVVKQNEYGTKVDIWSLGIMIIE-M 502
```

Applikation: Score = 144

```
Alignment 1:
Z: 1   MAEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEAYVAESVEDGSLCVAKVMDLSKMSQRD 60
Q: 288 TDRQALAMLKDSVTSHDPVEYFNVKHKLGQGASGSVYLAKVVGGKQLGIFDSVAIKSIDL 347

Z: 61  KRYAQSEIKCLANCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKLRGSGDARYFQ 120
Q: 348 QCQTRKELILNEITVMRESIHPNIVTYLDSFLVRERHLWVVMEYMNAGSLTDIIEKSKLT 407

Z: 121 EHEALFLFLQLCLALDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVSGVV 180
Q: 408 EAQIARICLETCKGIQHLHARNIIHRDIKSDNVLLDNSGNIKITDFGFCARLSNRTNKRV 467

Z: 181 ASTFCGTPYYLAPELWNNKRYNKKADVWSLGVLLYEIMGMKKPFSASNLKGLMSKVLAGT 240
Q: 468 TMVGTPYWMAPEVVKQNEYGTKVDI--WSLGIMIIEMIENEPPYLREDPIRALYLIAKNG 527

Z: 241 YAPLPDSFSSEFKRVVDGILVADPNDRPSVREIFQIPYINKGLKLFVQALKKNERISDSV 300
Q: 528 TPTLKKPNLVSKNLKSFLNSCLTIDTIFRATAAELLTHSFLNQACPTEDLKSIIFSRKAN 588

Z: 301 KEVL 304
Q: 589 THIN 589

Alignment 2:
Z: 310 ILSSEVSPDAHRFLVSQINYDVTHRGHVNKLGGGNGKSWKPRFLQIVRGQLILTDDEEGN 359
Q: 1   MLLSVRGVPVEIPSLKDTKKSKGIIRSGWVMLKEDKMKYLPWTKKWLVLSSNSLSIYKGS 60

Z: 370 NPKGLNLEQVQGACPVPHSTAKRDFVFALNTVGGKGMW------FQAVSHGDMEMWVHAI 429
Q: 61  KSESAQVTLLLKDIQKVERSKSRTFCFKLRFKSSTKNFEIQACELSVADNMECYEWMDLI 120

Z: 430 QRGIGVA 430
Q: 121 SSRALAS 126
```

SERINE/THREONINE-PROTEIN KINASE B0495.2 IN CHROMOSOME II (Q09437)

BLAST: Score = 134 Expect = 3e-07 Identities = 32/91 (35%)

```
Query: 139 HSHKM--LHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTV---SGVVASTFCGTPYYLAP 193
           H HK+  LHRD+K++N+L++  G++K+ DFG + +Y D +   + +V      T +Y +P
Sbjct: 474 HMHKLWILHRDLKTSNLLMSHKGILKIADFGLAREYGDPLKKFTSIVV-----TLWYRSP 528

Query: 194 ELWNNKR-YNKKADVWSLGVLLYEIMGMKKP 223
           EL    R Y+   D+WS+G ++ E + + KP
Sbjct: 529 ELLLGTRLYSTPVDMWSVGCIMAEFI-LLKP 558
```

Applikation: Score = 77 (geänderte Parameter: Länge = 200, T3=15)

```
Alignment 1:
Z: 1   MAEPFSTILGTDGSGGRCKYLNKGIVGLGSYGEAYVAESVEDGSLCVAKVMDLSKMSQR 60
Q: 337 IAQLPVFYPGLMGCRNIDEYECVNRVDEGTFGVVYRGKDKRTDEIVALKRLKMEKEKEG 396

Z: 61  DKRVAQSEIKCLANCNHPNIIRYIEDHEENDRLLIVMEFADSGNLDEQIKLRGSGDARY 120
Q: 397 FPITALREINMLLKAGNHPNIVNVKEILLGSNMDKIYMAMEFVEHDMKSLLDTMSRRNK 456

Z: 121 FQEHEALFLFLQLCLALDYIHSHKMLHRDIKSANVLLTSTGLVKLGDFGFSHQYEDTVS 180
Q: 457 RFSIGEQKTLLQQLLSGIEHMHKLWILHRDLKTSNLLMSHKGILKIADFGLAREYGDPL 516

Z: 181 GVVASTFCGTPYYLAPELWNNKRYNKKADVWSLGVLLYEIMGMK 221
Q: 517 KKFTSIVVTLWYRSPELLLGTRLYSTPVDMWSVGCIMAEFILLK 557


Alignment 2:
Z: 229 LKGLMSKVLAGTYAPLPDSFSSEFKRVVDGILVADPNDRPSVREIFQIPYINKGLKLFV 288
Q: 85  RDKERDKKREKDKRDDRRDVRGPDARQKDRDFKGRQERSGRDQKVHEHRHHHHHRKHET 144

Z: 289 QALKKNERISDSVKEVLVTQVSEILSSEVSPDAHRFLVSQINYDVTHRGHVNKLGGGNG 348
Q: 145 DGHRTNRSNRDRSSERDSEKHKRHIDRHKKSSTTSPDNDKSPHKKSKHTDVPADAKLFD 204

Z: 349 KSWKPRFLQIVRGQLILTDDEEGNNPKGLNLEQVQGACPVPHSTAKRDFVFALNTVGGK 408
Q: 205 RILDPNYKKKDDDVLVIEDVEMSPIEILEEKEEKEIVEFTIDSPAGPKKYSKFESDPES 264

Z: 409 GMWFQAVSHGDMEMWVHAIQRGIGVA 431
Q: 265 DH--DDTKPKSPGKAEDDDDVIEVLD 285
```

C. UML - Unified Modelling Language

Die Unified Modelling Language (UML) ist eine Modellierungssprache, die die Vorgehensweise festlegt. Die erste Version ist 1991 enstanden. Sie kann im gesamten Software-Entwicklungsprozeß eingesetzt werden. Zum Erstellen von Sichten und Modellen stehen verschiedene Techniken und Methoden zur Verfügung.

In dieser Diplomarbeit werden nur Klassendiagramme verwendet. Eine Klasse ist der Typ eines Objektes und beschreibt den Aufbau der Datenkomponenten und Methoden, die zu diesem Objekt gehören.

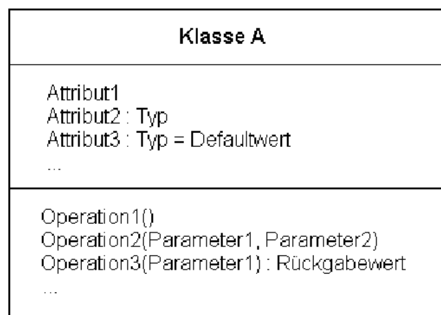


Abbildung C.1.: Darstellung eines Klassendiagramms

Das erste Fach beinhaltet den Namen, das zweite die Attribute und das dritte die Methoden der Klasse. In einem vierten Fach könnten innere Klassen oder Attribute, die jeweils eine öffentliche Methode zum Setzen und Lesen von aussen besitzen, erscheinen.

Ein Rechteck kennzeichnet eine Klasse (Abb. C.1). Nach dem fettgedruckten Klassennamen folgen in einem eigenen Fach die Attribute dieser Klasse. Sie werden mit dem Namen und dem Datentyp angegeben. Aber es werden nicht alle Attribute angegeben, sondern nur die, die zum Verständnis des Modells notwendig sind. Klassenattribute sind kein Merkmal eines Objektes wie die Attribute, sondern ein Merkmal der Klasse. Diese sind für alle Objekte einer Klasse gleich und werden unterstrichen dargestellt.

Das dritte Fach enthält die Methoden dieser Klasse. Bei diesen gilt ebenso wie bei den Attributen, dass nur wichtige angezeigt werden. Es wird der Name der Operation, die Parameterliste und der Rückgabetyp angeben.
Wenn der Aufruf von Operationen durch fremde Objekte geschieht, nennt man diese Beziehung zwischen zwei Klassen Assoziation. Sie wird durch einen Pfeil dargestellt, wenn die Objekte der einen Klasse den Objekten der assozierten Klasse Nachrichten schicken können. Dieser Pfeil wird mit Zahlen oder Intervallen gekenntzeichnet, um zu zeigen wieviele Objekte an dieser Beziehung teilnehmen.

1 : 1	Jede Klasse besitzt ein Objekt.
0 .. 1	Gibt eine optionale Verbindung an.
1 : n	Ein Objekt ist mit einer Menge von Objekten verbunden.
n : m	Auf beiden Seiten gibt es eine Menge von Objekten.

Tabelle C.1.: Vielfachheit einer Verbindung zwischen zwei Klassen

Es gibt weitere Beziehungen und Diagrammarten. Diese wurden in dieser Diplomarbeit nicht verwendet und werden deshalb hier nicht betrachtet. Einen Überblick über die UML bietet [SvG00].

C.1. Zeichenerklärung

-	*private* (Zugriff nur aus der Klasse selbst)
#	*protected* (Zugriff aus der Klasse, aus abgeleiteten Klassen oder innerhalb eines Paketes)
+	*public* (Zugriff aus der Klasse, aus abgeleiteten Klassen und aus jedem Aufruf durch andere Klassen)
name (unterstrichen)	*static* (Klassenvariable)
→	Assoziation

C.2. Klassendiagramm mit allen implementierten Klassen

D. Inhalt der Begleit-CD

- Diplomarbeit (pdf)
- Java Quelldateien
- JavaDoc zu den Quelldateien (HTML)

Literaturverzeichnis

[ABGW94] Stephen F. Altschul, M. S. Boguski, W. Gish, and J. C. Wootton. Issues in searching molecular sequence databases. *Nature Genetics*, 6:119–129, 1994.

[ABL+94] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of THE CELL.* Garland Publishing, 1994.

[AG97] A. Apostolico and R. Giancarlo. Sequence alignment with tandem duplication. *J. Comp. Biol.*, 4(3):351–367, 1997.

[AGM+90] Stephen F. Altschul, Warren Gish, Webb Miller, Eugen M. Myers, and David J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[Alt91] Stephen F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *J. Mol. Biol.*, 219:555–565, 1991.

[AMS+97] Stephen F. Altschul, T.L. Madden, A.A. Schaeffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucl. Acids Res.*, 25(17):3389 – 3402, 1997.

[CLL96] Thomas H. Cormen, Charles E. Leiserson, and Ronald L.Rivest. *Introduction to algorithms.* The MIT Press, 1996.

[DEKM98] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison.

Biological sequence analysis: Probalistic models of proteins and nucleic acids. Cambridge University Press, 1998.

[DG91] R.L. Dorit and W. Gilbert. The limited universe of exons. *Curr. Opin. Genet. Dev.*, 1(4):464–469, 1991.

[Doo95] R.F. Doolittle. The Multiplicity of Domains in Proteins. *Ann. Rev. Biochem.*, 64:287–314, 1995.

[DSO78] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978.

[EO00] Anton J. Enright and Christos A. Ouzounis. Generage: a robust algorithm for sequence clustering and domain detection. *Bioinformatics*, 16(5):451–457, 2000.

[GL99] D. Graur and W. H. Li. *Fundamentals of Molecular Evolution.* Sinauer Associates, 1999.

[HA93] Japp Heringa and P. Argos. A method to recognize distant repeats in protein sequences. *Proteins Struct. Func. Genet.*, 17:391–411, 1993.

[HH92] Steven Henikoff and Joria G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 1992.

[JC85] J. Janin and C. Chothia. Domains in proteins: definitions, location, and structural principles. *Methods Enzym.*, 115:420–430, 1985.

[Kni97] Rolf Knippers. *Molekulare Genetik.* Georg Thieme Verlag, Stuttgart, 1997.

[Krü99] Guido Krüger. *Go to Java 2.* Addison-Wesley, 1999.

[KV98] A. Krause and Martin Vingron. A set-theoretic approach to database searching and clustering. *Bioinformatics*, 14(5):430–438, 1998.

[LBB+96] Harvey Lodish, David Baltimore, Arnold Berk, S. Lawrence, and Zipursky et.al. *Molekulare Zellbiologie.* Walter de Gruyter, 1996.

[LGWN01] W.-H. Li, Z. Gu, H. Wang, and A. Nekrutenko. Evolutionary analyses of the human genome. *Nature*, 409:847–849, 2001.

[Li99] W. H. Li. *Molecular Evolution.* Sinauer Associates, 1999.

[LP97] Georg Löffler and Petro E. Petrides. *Biochemie und Pathobiochemie.* Springer Verlag, 1997.

[MA99] Burkhard Morgenstern and William R. Atchley. Evolution of bhlh transcription factors: Modular evolution by domain shuffling. *Mol. Biol. Evol.*, 16(12):1654–1663, 1999.

[Mey98] André Meyer. *JFC 1.1 mit Java Swing 1.0.* Addison Wesley Longman Verlag GmbH, 1998.

[MSV00] Tobias Müller, Rainer Spang, and Martin Vingron. Modelling amino acid replacements. *J.Comp.Biol.*, 7(6):761–776, 2000.

[NB01] NCBI-BLASTP. http://www.ncbi.nlm.nih.gov/BLAST/, 2001.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[Pea00] William R. Pearson. *Protein Sequence Comparison and Protein Evolution.* University of Virginia, 2000.

[Pfa01] Pfam Version 6.4. `http://www.sanger.ac.uk/Software/Pfam/search.shtml`, 2001.

[PL88] William R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:2444–2448, 1988.

[Pro01] ProDom Version 2001.1. `http://protein.toulouse.inra.fr/prodom/doc/prodom.html`, 2001.

[RE99] E. Rivas and S. R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J. Mol. Biol.*, 285:2053–2068, 1999.

[SSZ+94] A. Stoltzfus, D.F. Spencer, M. Zuker, J.M. Logsdon Jr., and W.F. Doolittle. Testing the Exon Theory of Genes: The Evidence from Protein Structure. *Science*, 265:202–207, 1994.

[Ste97] Michael J. E. Sternberg. *Protein Structure Prediction - a practical approach*. IRL Press at Oxford University Press, 1997.

[SvG00] Jochen Seemann and Jürgen Wolff von Gudenberg. *Software-Entwurf mit UML*. Springer Verlag, 2000.

[SW81] T.F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[SW92] M. Schoeninger and M. S. Waterman. A local algorithm for dna sequence alignment with inversions. *Bull. Math. Biol.*, 54:521–536, 1992.

[SWI01] SWISS-PROT Release 39.22. `http://kr.expasy.org/sprot/`, 2001.

[VDR99] Jean-Stephane Varré, Jean-Paul Delahaye, and Eric Rivals. Transformation distances: a family of dissimilarity measures based on movements of segments. *Bioinformatics*, 15:194–202, 1999.

[Wat95] Michael S. Waterman. *Introduction to computational biology*. Chapman & Hall, 1995.

Index

α, 35, 40
β, 35, 40
γ, 35

affin-linear, 28
Alignment, 16
    globales, 18, 27
    lokales, 18, 27
Alphabet, 17
Aminosäure, 12
Ausgabemodus, 37
Automatenmodell, 30
AWT, 33

basisch, 12
BLAST, 24, 30, 43

Dijkstra-Algorithmus, 26, 29
DNA, 15
Domäne, 15
dynamische Programmierung, 20

Eingabemodus, 33
Exon, 15

Gap, 18
Gen, 15
Grafikmodus, 39
Graph, 25, 28
    gerichteter, 25
    gewichteter, 25

heuristische Suchverfahren, 24
homolog, 16
hydrophil, 12
hydrophob, 12

Intron, 15

Java, 32

Klasse, 33

Lücke, 18
Listener, 34

match, 19
mismatch, 19

Objekte, 33

Proteine, 12
Punktmutationen, 17

Quartiärstruktur, 14
Quellsequenz, 28

Rekombination, 17

sauer, 12

Score-Matrix, 22, 27

Sekundärstruktur, 14

Shuffling, 15

  domain, 16

  exon, 16

Smith-Waterman-Algorithmus, 21, 27

Sonde, 24, 31

Substitutionsmatrix, 19

  Blosum, 19

  Identität, 19

  PAM, 20

  VT, 20

Swing, 33

Tertiärstruktur, 14

Textmodus, 37

threshold 1, 35, 42

threshold 2, 35, 42

threshold 3, 35, 43

Zielsequenz, 28