Fachstudie Bioinformatik:

Werkzeuge zur Durchführung und Visualisierung von Proteinsequenzvergleichen

Abschlussbericht

Søren Brügmann Daniel Minder Jörg Rüdenauer

4. Februar 2003

Inhaltsverzeichnis

1 Einleitung

1.1 Über dieses Dokument

Dieses Dokument ist der Bericht über eine *Fachstudie* im Bereich der Bioinformatik am Institut für technische Biochemie der Universität Stuttgart (ITB) [1]. Es stellt die durchgeführten Arbeiten und die daraus gewonnenen Erkenntnisse zusammen. Primär richtet sich diese Arbeit an die an diesem interdisziplinären Projekt beteiligten Personen, Mitarbeitern vom ITB und von der Abteilung Bildverstehen [2] des Instituts für Parallele und Verteilte Systeme (IPVS) [3] der Stuttgarter Informatik-Fakultät [4].
Durch eine einfache Beschreibung der biologischen Hintergründe, sowie der verwendeten bioinformatischen Basistechnologien erhalten jedoch auch Fachfremde, die sich für dieses Themengebiet und die Durchführung von wissenschaftlichen Arbeiten an dieser Schnittstelle zwischen Biologie und Informatik interessieren einen spannenden Einblick.

In dieser Einleitung wird zunächst kurz beschrieben, was eine Fachstudie im Sinne der Stuttgarter Fakultät für Informatik eigentlich ist. Es folgt eine Darstellung der Thematik dieser Fachstudie und eine Bemerkung zur Wahl dieses eher ungewöhnlichen Themengebietes. Die weiteren Abschnitte behandeln dann zunächst die biologischen Grundlagen und die verwendeten bioinformatischen Software-Werkzeuge, bevor unsere Arbeit und ihre Ergebnisse genauer beleuchtet werden.

1.2 Die Fachstudie im Studiengang Softwaretechnik

Im Diplomstudiengang Softwaretechnik [5] an der Stuttgarter Informatikfakultät ist im Hauptstudium eine sogenannte *Fachstudie* vorgeschrieben. Dabei handelt es sich um eine Arbeit, die üblicherweise von drei Studierenden während eines Semesters durchgeführt wird, wobei der Umfang bei ca. 4 Semesterwochenstunden (SWS) pro Studierendem liegen sollte. Ziel einer Fachstudie ist eine Art *Expertise*. Häufig geht es dabei um die Evaluierung oder den Vergleich von Software-Produkten, -Prozessen oder -Werkzeugen. Die Bearbeiter müssen dazu (in kurzer Zeit) die notwendige Kompetenz erwerben, die richtige Literatur finden, den richtigen Leuten die richtigen Fragen stellen, die richtigen Schlüsse ziehen und ihre Resultate in einem gut lesbaren Bericht dokumentieren.

1.3 Thematik dieser Fachstudie

Das konkrete Thema unserer Fachstudie war die Evaluierung, Erweiterung und Verknüpfung von Werkzeugen, mit denen Vergleiche von Proteinsequenzen durchgeführt und visualisiert werden können.

Eine Proteinsequenz beschreibt den kettenartigen Aufbau eines Proteins aus Aminosäuren. Die zwanzig bekannten Aminosäuren bilden die 'Bausteine' für alle Proteine. Üblicherweise bestehen Proteine aus mehreren hundert Aminosäuren.

Die Sequenz beschreibt das Protein zwar für viele Anwendungen ausreichend genau, jedoch läßt sich aus ihr nur sehr schwierig auf das eigentliche Aussehen und die Wirkungsweise des Proteins schließen, die durch die sogenannte Faltung bestimmt wird. Nichtsdestotrotz hat sich gezeigt, dass Proteine, deren Aminosäurensequenzen große Ähnlichkeiten aufweisen, im Regelfall auch ähnliche Funktionen haben. Deshalb kann man durch einen automatischen Vergleich der Sequenzen mittels geeigneter Algorithmen auf das Maß von Verwandtschaft (*Homologie*) zweier Proteine zueinander schließen.

Aufrund der einzigartigen Fähigkeit des Softwarewerkzeugs *BLAST* zur profilbasierten Suche (*PSI-BLAST*), deren Beschreibung im Weiteren noch folgt, kam für den Vergleich von Anfang an nur dieses Werkzeug in Betracht.

Die Visualisierung genetischer Stammbäume (*phylogenetischer Bäume*) ermöglichen hingegen eine Vielzahl frei verfügbarer Software-Werkzeuge. So war es hier zunächst notwendig eine Recherche durchzuführen, wobei nach Möglichkeit das dem ITB bereits vertraute Software-Werkzeug *Phylip* zum Einsatz kommen sollte.

Die softwaretechnische Grundlage zur Erweiterung und Verknüpfung der Software-Werkzeuge war zu Beginn der Fachstudie offen. Mitarbeiter von ITB empfahlen uns die in der Bioinformatik favorisierte Sprache *Perl* aufgrund ihrer großen Fähigkeiten zum Pattern-Matching.

Das biologische Ziel, zu dem unsere Arbeit beitragen sollte, war nun, Verwandtschaftsbeziehungen zwischen verschiedenen Proteinfamilien zu finden. Dies ist recht kompliziert, weil Vertreter verschiedener Familien nur eine geringe Sequenzähnlichkeit aufweisen.
Erkenntnisse in diesem Bereich würden aber beispielweise ein Licht auf die Entwicklung der Proteine im Zuge der Evolution werfen oder helfen, die Zusammenhänge zwischen Sequenz und Wirkungsweise eines Proteins besser zu verstehen, was bei der Entwicklung neuer Proteine von großer Wichtigkeit ist.

Um dieses Ziel zu erreichen, sollten automatisch eine große Anzahl an Sequenzvergleichen durchgeführt werden, einerseits kreuzweise zwischen typischen Vertretern verschiedener Familien, andererseits mehr 'in die Tiefe' gehend als Suche nach Proteinen, die eine *Brücke* zwischen zwei Familien bilden könnten.

1.4 Eine Fachstudie im Bereich Bioinformatik?

Die Wahl des Themengebietes *Bioinformatik* erfordert wahrscheinlich eine kurze Erläuterung. Fachstudien werden üblicherweise eher in den Kerngebieten der Informatik durchgeführt. Wünschenswert ist dabei ein Kunde aus der Softwareindustrie in enger Kooperation mit einer Abteilung aus der Informatik-Fakultät.

Wir drei Studenten haben jedoch alle ein breites Interesse auch an nicht-informatischen Themengebieten, und einer von uns (*Sören Brügmann*) hatte private Kontakte zu einigen Studierenden des ITB. So entstand die Idee, unsere Fachstudie dort durchzuführen.

In der Abteilung Bildverstehen des IPVS fanden wir schnell den notwendigen Partner auf Seiten der Informatik. Die Abteilung Bildverstehen hatte sich schon einmal in der Vergangenheit mit dem Themengebiet der Bioinformatik beschäftigt.

Namentlich muss man besonders die beiden Doktoren *Michael Schanz* [6] von der Abteilung Bildverstehen und *Jürgen Pleiss* [7] vom ITB lobend erwähnen, ohne deren großen Willen zur Kooperation diese Arbeit niemals zustande gekommen wäre. Alle Beteiligten sind sich einig, dass eine höhere Kooperation zwischen verschiedenen Fakultäten an der Universität - insbesondere in Hinsicht auf den Anwendungsbezug der Informatik - im höchsten Maße wünschenswert wäre, und es ist unsere Hoffnung, dass wir den Anstoß zu einer Erweiterung des (bisher doch sehr beschränkten) Kanons der drei Anwendungsfächer im Studiengang Softwaretechnik um das Fach Biologie bzw. Bioinformatik gegeben haben.

1.5 Personen

Diese Fachstudie wurde von drei Studierenden der Softwaretechnik im 8. Semester durchgeführt:

- Søren Brügmann
  soeren@bruegmann.biz

- Daniel Minder
daniel@minder.de

- Jörg Rüdenauer
joerg@ruedenauer.net

Für Nachfragen stehen wir selbstverständlich alle gerne zur Verfügung. Betreut wurden wir auf Seiten der Biologie von *Markus Fischer*. Auf Seiten der Informatik wurde diese Fachstudie von der Abteilung Bildverstehen im Institut für Parallele und Verteilte Systeme betreut; Ansprechpartner hier ist *Dr. Michael Schanz*, Prüfer ist *Prof. Paul Levi* [8].

2 Die Aufgabenstellung

Unsere Aufgabe ergab sich aus einem Informationsmangel der Lipase-Datenbank LED [9] des ITB, den es zu eliminieren galt. Dieser Mangel bestand in einem Fehlen von Information über Homologiebeziehungen zwischen den 17 Superfamilien. Der Grund für diesen Zustand lag in der grundsätzlichen Schwierigkeit zuverlässig entfernte Verwandschaftsbeziehungen auf Sequenzbasis zu ermitteln ohne dabei sequenzähnliche, aber strukturell völlig unterschiedliche Enzyme fälschlicherweise als verwandt zu klassifizieren.

Übliche Methoden des multiplen Sequenzvergleichs sind hier unbrauchbar, einerseits wegen der großen Anzahl der Sequenzen und andererseits wegen der geringen Ähnlichkeit vieler Sequenzen zueinander.
Auf Strukturbasis wäre die Aufgabe prinzipiell besser lösbar, weil trotz dieses hoch diversen Sequenzraums alle bisher bestimmten Proteinstrukturen der Lipasen die selbe Architektur, nämlich die der *alpha/beta-Hydrolasen* aufweisen [10]. Dies liegt daran, dass die Struktur enger an die Funktion des Proteins gekoppelt ist als die Aminosäuresequenz und so die Proteinstruktur sich deutlich stabiler gegenüber evolutionären Einflüssen verhält.
Aufgrund dessen wird angenommen, dass alle Lipasen, trotz der starken Sequenzunterschiede die Struktur der *alpha/beta Hydrolasen* besitzen. Würde man also die Möglichkeit besitzen, schnell und umfassend die Familien auf Strukturebene zu vergleichen, hätte man es deutlich leichter mit der Ermittlung der Verwandschaftsbeziehungen.
Allerdings ist man zur Zeit noch auf den Vergleich von Sequenzen angewiesen, weil zu zuwenig Proteinen die Stuktur bekannt ist und weil ein Vergleich auf Strukturebene aufgrund der Komplexität zu aufwändig wäre.

Um dennoch Verwandtschaftsbeziehungen zwischen den einzelnen Familien zu erforschen, wurden heuristische Methoden zum Vergleich von evolutionär entfernten Sequenzen entwickelt, wie z.B. der *BLAST* Algorithmus, die nicht mit Strukturmodellen arbeiten, sondern allein auf der Sequenzfolge beruhen. Entsprechend sind die Ergebnisse dieser Verfahren nicht als Tatsachen zu werten, da sie auf stark vereinfachten heuristischen Modellen basieren.

Im Fall einiger Lipasen verschiedener Superfamilien ist die evolutionäre Entfernung aber soweit vorangeschritten, dass auch mit diesen Methoden keine direkte Verbindung zu anderen Lipasen zu finden ist. Zur Betrachtung solche Beziehungen existiert seit einiger Zeit der *PSI-BLAST*-Algorithmus, eine Erweiterung des *BLAST*-Algorithmus.

Das ITB stellte an uns nun die Aufgabe softwaretechnische Hilfsmittel zu evaluieren und damit Methoden auf Basis des *BLAST* und des *PSI-BLAST*-Algorithmus´ zu entwickeln die dazu beitragen können Homologiebeziehungen zwischen Lipasen zu identifizieren um damit die Klassifikation der LED zu überprüfen und ggfs. zu verbessern.

Die Untersuchung sollte dazu aus *zweierlei Sichten* durchgeführt werden. Einmal aus interner Sicht, d.h. in Form einer alleinigen Homologie-Analyse der in der LED gespeicherten Sequenzen durch einen paarweisen automatisieren Vergleich.
Die andere Untersuchung sollte ausgehend von den Sequenzen der LED auch alle anderen bekannten Proteine einschließen, um so mögliche Sequenzen noch unbekannter, sogenannte *Brücken-Proteine* zu entdecken. Solche *Brücken-Proteine* würden zur Zeit noch als völlig unverwandt angesehene Protein-Familien miteinander evolutionär verknüpfen, indem sie möglicherweise einen ersten gemeinsamen Verwandten als Bindeglied zweier Familien identifizieren.

Aus den daraus erhaltenen Beziehungen zwischen den einzelnen Familien kann man dann durch eine Verwandtschaftsanalyse die Gruppierung der bisher isolierten Superfamilien erzielen. Die Brückenele-

mente selbst stellen jeweils einen Vertreter aus einer Sequenzfamilie dar, die nicht zwangsläufig Lipasen umfassen muss, deren Verwendung zum Sequenzvergleich zwischen zwei isolierten Lipasefamilien aber legitim ist, um so Strukturmodelle zu erstellen und neue Einsichten in die Funktionsweise dieser Lipasefamilie zu gewinnen.

3 Biologische Grundlagen

Das Institut für Technische Biochemie (ITB) der Universität Stuttgart beschäftigt sich seit geraumer Zeit mit einer speziellen Art von Proteinen, den *Lipasen* und baute in diesem Zusammenhang die *Lipase Engineering Database* (LED) auf. Dabei handelt es sich um eine internetbasierte Datenbank, die Sequenz- und Struktur-Informationen, sowie insbesondere familiäre Beziehungen zwischen Lipasen zur Verfügung stellt. Zum jetzigen Zeitpunkt strukturiert die Datenbank die Welt der Lipasen in 17 Superfamilien, die sich in 37 homologe, d.h. evolutionär verwandte Familien aufteilen.

Unsere Arbeit im Rahmen der Fachstudie sollte Möglichkeiten aufzeigen, die Datenbasis zu validieren und zu erweitern. Für diese Arbeit war es notwendig sich mit den Grundlagen biologischen vertraut zu machen, primär mit der Frage: *Was sind eigentlich Lipasen?*

Lipasen sind spezielle Proteine mit der Fähigkeit die Spaltung von Fetten (*Lipiden*) zu begünstigen. Sie werden zur Gruppe der Enzyme gerechnet, weil sie eine katalytische Wirkung haben, also die Aktivierungsenergie für eine oder mehrere chemische Reaktionen herabsetzen und damit eine Beschleunigung verursachen.

Wie alle Proteine sind auch Lipasen Aminosäureketten. Das Bild 1 zeigt schematisch den Aufbau eines Proteins in verschienenen, in der Biologie gebräuchlichen Ansichten. Für viele Lipasen ist die Anordnung der Aminosäuren bereits entschlüsselt und in öffentlichen Datenbanken für jedermann zugänglich.

Die Bestimmung der Aminosäuren-Sequenz für ein Protein ist heutzutage einfach durchführbar. Schlechter sieht es dagegen bei der dreidimensionalen Struktur (*Konformation*) aus. Nur zu verhältnismäßig wenigen Enzymen gibt es sichere Strukturinformationen. Um eine sichere Aussage über die Struktur machen zu können, braucht man zum jetzigen Zeitpunkt noch Monate. Man behilft sich entweder mit heuristischen Methoden, die aus der Sequenz mit hoher Wahrscheinlichkeit die zugehörige Struktur durch Analogie-Schluß basierend auf Vergleichen mit ähnlichen Sequenzen (deren Struktur bereits bekannt ist) ermitteln, oder mit mathematischen Modellen. Aufgrund des universellen Charakters ist die Technik zur Strukturvorhersage mit Hilfe mathematischer Modelle besonders interessant. In jüngster Zeit konnten auf diesem Gebiet große Erfolge erzielt werden. Es gelangen Strukturvorhersagen, die sich in Experimenten als bis auf das Angstrom genau herausstellten, im Rahmen der alle zwei Jahre stattfindenden CASP-Konferenz (*Critical Assessment of Strucure Prediction* [11]).
Wie man auf dem Bild unschwer erkennen kann, ist die Struktur eines Enzyms sehr komplex. Die Struktur bestimmt die Fähigkeit(en), die Funktion(en) des Proteins. Bezogen auf Lipasen legt die Struktur also fest, welche Arten von Fetten wie aufgespalten werden. Jedes Enzym und damit auch jede Lipase ist einzigartig hinsichtlich ihrer Eigenschaften. Ihre Arbeit verrichtet sie durch Auslösung chemischer Reaktionen. Dazu muss sie Kontakt mit dem Lipid aufnehmen. Dafür existiert in jeder Lipase ein *aktives (katalytisches) Zentrum*, das aus polaren Resten besteht, die Lipide an sich binden können. Die dreidimensionale Struktur lässt jedoch nur bestimmte Lipide zum aktiven Zentrum vordringen und bestimmt damit die *Substratspezifität* der Lipase. Im aktiven Zentrum wird das Lipid in engen Kontakt mit den umgebenen Aminosäuren gebracht. Die katalytische Wirkung beruht dann auf der kontrollierten gleichzeitigen Einwirkung verschiedener polarer Gruppen auf eine Bindung, Atom oder Atomgruppe.

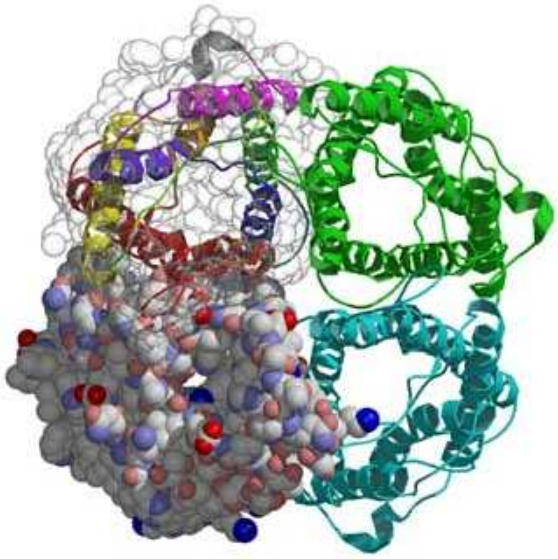

Abbildung 1: Schematische Darstellung eines Proteins

Eventuell unterstützen spezielle Coenzyme den Vorgang, die sich für den Spaltungsprozess an die Lipase anlagern. Die Einbettung des Lipids in die Lipase legt die *Wirkungsspezifität* fest, d.h. die Art und Weise wie das Lipid umgesetzt wird.

Besonders an Lipasen gegenüber anderen ebenfalls die Fettspaltung begünstigenden Enzymen ist, daß das aktive Zentrum nicht zu jedem Moment zugänglich ist. Vielmehr verschließen im Ruhezustand je nach Lipase ein oder zwei helicale Strukturen von jeweils bis zu zwanzig Aminosäuren den Zugang zum aktiven Zentrum. Diese Helices bezeichnet man deshalb auch anschaulich als Deckel der Lipase.

Eine Aktivität der Lipase setzt die Öffnung des Deckels voraus. Chemisch gesehen hat der Deckel eine sehr interessante Eigenschaft. Seine Außenseite ist wasserliebend (*hydrophil*), seine Innenseite hingegen fettliebend (*lipophil*). Eine solche Kombination bezeichnet man als *amphiphil*. Solange sich die Lipase in einer wässrigen Lösung befindet, bleibt aufgrund der Spezifikation des Deckels die Lipase inaktiv. Nähert sich die Lipase einem Lipid, öffnet sich der Deckel und die Lipase 'dockt' mit seinem aktiven Zentrum an das Lipid an.
Aus diesem Verhalten ergibt sich eine weitere Besonderheit der Lipasen. Sie bevorzugen wasserunlösliche Substrate zur Spaltung und lagern sich an der Phasengrenze zwischen Lipid und Wasserlösung an.

Lipasen kommen in allen Zelltypen und Organismen vor und sind essentieller Bestandteil des Fettstoffwechsels. Von Bedeutung sind Lipasen für die biotechnische Industrie, weil es mit ihrer Hilfe möglich ist vielfältige chemische Reaktionen zu katalysieren. Die von Lipasen beschleunigten Reaktionen führen oft zu optisch reinen Produkten. Diese Eigenschaft hat Lipasen insbesondere in der Waschmittelindustrie zu großer Popularität verholfen.

Aber auch in anderen Bereichen stellen Lipasen eine umweltverträgliche und preiswerte Alternative zu Produkten der chemischen Industrie dar. Anwendung finden Lipasen darüber hinaus im medizinischen Bereich. Hier stehen Produkte im Vordergrund, die durch Störungen im Lipidstoffwechsel ausgelöste Krankheiten behandeln sollen.[12]

4 Verfügbare Software

Die LED der Technischen Biochemie Stuttgart enthält Daten, die durch die Kombination vieler verschiedener frei zur Verfügung stehender Software-Werkzeuge und Datenbanken einerseits und Ergebnissen aus Versuchen des eigenen Labors andererseits gewonnen wurden. Die aktuelle Klassifikation beruht auf Vergleichen zwischen Proteinen bzgl. verschiedener Merkmale, wie z.B. der *Sequenz*, der *Struktur*, der *pH-Stabilität und vieler anderer.*

Zur Bearbeitung der Aufgabenstellung war es zunächst unumgänglich sich mit den gängigsten Werkzeugen insbesondere zum Vergleich von Sequenzen und zur Visualisierung von Verwandschaftsbeziehungen vertraut zu machen, um sie bezüglich ihre Brauchbarkeit zur Problemlösung beurteilen zu können.

Glücklicherweise existieren von vielen wissenschaftlichen Einrichtungen weltweit eine riesige Anzahl von Software-Werkzeugen für nahezu alle Bereiche der Bioinformatik. Jedes Software-Werkzeug deckt allerdings nur ein kleines Anwendungsfeld ab. Dafür bieten die Programme meist standardisierte Ein- und Ausgabeformate an, so daß in den meisten Fällen eine Verkettung und Einbettung leicht möglich ist. Viele Autoren stellen zudem den Quellcode zu ihren meist in *Java* oder *Perl* programmierten Paketen frei zur Verfügung.

Die folgenden Kapitel behandeln die Softwarepakete, die wir im Rahmen unser Fachstudie verwendet oder zumindest näher untersucht haben.

4.1 BLAST

Eines der bekanntesten Werkzeuge zu Beginn, die Suchmaschine für Sequenzen: *BLAST* [13]. Der Vergleich von Sequenzen ist in der Bioinformatik von größter Bedeutung. Doch sollte man ihn auch nicht überschätzen. Wie der Erfolg der LED zeigt, erhält man nur ein wirkliches Verständnis von Proteinen, wenn man neben der Sequenz auch noch weitere Eigenschaften des Proteins in seine Betrachtungen mit einbezieht.

Dies soll verdeutlichen, dass ein Vergleich von Proteinen anhand ihrer Sequenz lediglich ein Kriterium für den Vergleich darstellt. Zwar beruhen letztlich alle weiteren Eigentschaften auf ihr, aber in welcher Art ist noch weitgehend unklar, d.h. aus der Sequenz kann man zum jetzigen Zeitpunkt nur grob auf die Eigenschaften des Proteins schließen. Die weiteren Merkmale von Proteinen müssen also zur Zeit noch experimentell ermittelt werden, was bei manchen, wie z.B. der Struktur sehr aufwändig ist.

Ein Sequenzbestimmung (*Sequenzierung*) ist hingegen heutzutage leicht und schnell durchführbar. Folglich gibt es öffentliche täglich aktualisierte Datenbanken in denen die Sequenzen aller bekannten Proteine zu finden sind. Eine Suche in solchen Datenbanken ist Dank schneller Rechensysteme und ausgefeilter Heuristiken trotz der riesigen Menge von Proteinen in Sekunden durchführbar. Für Recherechen in biologischen Datenbanken stehen spezielle Suchmaschinen zur Verfügung die für den Benutzer ähnlich funktionieren wie die jedermann bekannten Suchmaschinen *Google* oder *Altavista* für das Internet.

Den Sequenzsuchmaschinen übergibt man eine Sequenz und bekommt ähnliche in der Datenbank vorhandene Sequenzen zurück. Im Unterschied zu herkömmlichen Suchmaschinen muß man vor einer Suche die Suchmaschine über verschiedene Parameter kalibrieren. Eine weitere Besonderheit liegt in der Kommentierung der Ergebnisse. Man erhält zu jedem Treffer noch statistische Werte für jede gefundene Sequenz, die darüber Auskunft geben, wie gut der jeweilige Treffer ist.

Die bekannteste und verbreiteste Suchmaschine heißt *BLAST*. Genaugenommen bildet *BLAST* den Ober-

begriff für eine ganze Sammlung von Spezialsuchmaschinen für Nucleotid- und Proteinsequenzen.

Auch viele Erkenntisse der LED beruhen auf *BLAST*-Suchen. So war es naheliegend diese Suchmaschine auch für diese Fachstudie einzusetzen, insbesondere deren neueste Entwicklung: die profilbasierte Suche *PSI-BLAST*.

Für unsere Arbeit war es nun ebenso notwendig neben dem biologischen auch bioinformatisches Wissen zu erlangen und dabei insbesondere ein Verständnis von *BLAST* als Grundbaustein unser Fachstudie zu bekommen.
Wie also funktioniert *BLAST* ?

4.1.1 Überblick

BLAST (*Basic Local Alignment Search Tool*) ist ein Software-Werkzeug zur schnellen und effizienten Suche lokaler Alignments in Sequenzdatenbanken. Atschul und Kollegen entwickelten den Algorithmus sowie dessen Implementierung 1990 am *NCBI*.

Alignments sind die Grundlage aller Sequenzanalysen. Solche Analysen basieren entweder auf Protein- oder auf Nucliotidsequenzen. Proteinsequenzen bestehen aus einer Verkettung von Aminosäuren; Nucliotide bilden die Bausteine der Nucliotidsequenzen. Es existieren zwanzig verschiedene Aminosäuren und vier verschiedene Nucliotide.

Zwischen Nucliotid- und Aminosäuresequenzen besteht ein eindeutiger Zusammenhang. Die die Gene bildenden Nucliotidsequenzen definieren Aminosäuresequenzen, wobei drei aufeinanderfolgende Nucliotide (genannt *Codon*) genau eine Aminosäure codieren.

BLAST ist in der Lage beide Sequenzarten zur Ähnlichkeitssuche zu verwenden. Die weitere Beschreibung beschränkt sich allerdings auf Aminosäuresequenzen, weil nur diese Untersuchungsgegenstand der Fachstudie waren.

Der Algorithmus beruht auf der Berechnung der Kenngrößen *Score* und *E-Wert* (Erwartungswert), die auf der Grundlage einer *Substitutionsmatrix* Aussagen zur Ähnlichkeit zweier Sequenzen zueinander machen.

4.1.2 Substitutionsmatrix

Die Auswahl einer geeigneten Substitutionsmatrix ist die kritische Entscheidung bei der Parametrisierung eines Sequenzvergleichs. Keine Scoring-Matrix ist optimal für alle Anwendungen geeignet. Daher sollte eine Matrix ausgewählt werden, die dem biologischen Phänomen, welches durch *BLAST* untersucht werden soll, angepasst ist. Zwei Familien von Substitutionsmatrizen kommen für fast alle Untersuchungen zum Einsatz, die *PAM*- und die *BLOSUM*-Matrizen.

Im Rahmen der Fachstudie fand auf Wunsch des ITB ausschließlich die *BLOSUM*-Matrix Verwendung. Für die Aufgabenstellung war sie nach den bisherigen Erfahrungen des ITB am besten geeignet.

BLOSUM-Matrizen sind folgendermaßen aufgebaut: Ein Eintrag in der Matrix beschreibt die Ähnlichkeit zwischen jeweils zwei Aminosäuren. Bei zwanzig bekannten Aminosäuren besteht die Matrix also aus genau 400 Einträgen. Zur weiteren Vereinfachung und Reduzierung der Datenmenge wurde jedoch der Austausch als symmetrisch definiert, d.h. für einen Austausch von Aminosäure A durch Aminosäure B gilt nach der *BLOSUM*-Matrix die gleiche Wahrscheinlichkeit wie für den Wechsel von B nach A. Damit

ist die Matrix spiegelbar an der Diagonalen.

Für die Ähnlichkeit selbst existiert ein einfaches Modell. Sie ist für zwei Aminosäuren A und B definiert als logarithmierter Quotient aus der Übergangswahrscheinlichkeit von A zu B und dem Produkt der Wahrscheinlichkeiten des Vorkommens von A und B. Die Wahrscheinlichkeiten wurden aus einem umfangreichen Vergleich von Sequenzen gewonnen.
Die Logarithmierung des Quotienten hat vorallem praktische Gründe. Es ist schlicht einfacher Wahrscheinlichkeiten zu addieren, als zu multiplizieren. Man vermeidet so numerische Ungenauigkeiten, die durch sehr kleine Werte entstehen können. Die einzelnen Werte in der Matrix bezeichnet man als *log-odds-ratios*. Sie drücken in etwa aus wieviel wahrscheinlicher ein Austausch der einen Aminosäure durch die andere ist, als man es per Zufall erwarten würde.

Positive Werte in der Matrix bezeichnen also konservative Austausche, die zwischen ähnlichen Aminosäuren möglich sind. Je niedriger die Zahl, um so unwahrscheinlicher ist ein Austausch der Aminosäuren miteinander. Von *BLOSUM*-Matrizen existieren verschiedene Versionen, bezeichnet durch eine Zahl N.

Für eine *BLOSUM-N* Matrix gilt, daß sie aus Sequenzen erzeugt wurde, die zueinander eine Ähnlichkeit von bis zu N Prozent aufweisen. *BLOSUM*-Matrizen mit hohem N eignen sich daher besser zum Vergleich nah verwandter Sequenzen, wohingegen 'niedrige' *BLOSUM*-Matrizen sinnvoller für die Identifizierung entfernter Verwandschaftsbeziehungen sind.[14]

4.1.3 Score

Der Score sagt aus, wie ähnlich zwei (Teil-)Sequenzen auf Basis einer Substitutionsmatrix zueinander sind. Errechnet wird der Score im einfachsten Fall durch Aufsummieren der, in der Matrix angebenen, Ähnlichkeiten zwischen den einzelnen Aminosäurenpaaren, die jeweils direkt untereinander stehen, wenn man die beiden Sequenzen linksbündig untereinander schreibt.

Eine zweitere, erweiterte Score-Berechnung beruht auf der Einführung sogenannter *gaps* (Lücken) an beliebiger Stelle innerhalb einer Sequenz. *Gaps* sollen im Laufe der Evolution durch Rekombination und Mutation verlorengeganene bzw. hinzugekommene Aminosäuren kompensieren. Mit Hilfe von *gaps* gelingt es längere Alignments zu erzeugen und so den Score erhöhen. Man stelle sich beispielsweise die Sequenzen *AABABAB* und *ABABAB* vor. Schreibt man sie direkt untereinander ergibt sich nur ein geringer Score, weil außer dem *A* an erster Position keine Übereinstimmungen vorhanden sind. Fügt man jedoch in der zweiten Sequenz nach dem ersten *A* eine Lücke in der zweiten Sequenz ein, ergeben sich nur Übereinstimmungen und damit ein hoher Score.

Für die Einführung eines gap, sowie dessen Länge, sieht *BLAST* frei definierbare Straf-Werte vor, um Anzahl (*gap-open penalty*) und Länge von *gaps* (*gap-extension-penalty*) zu begrenzen und damit die Aussagekraft des Score in jedem Fall zu gewährleisten. Die genaue Berechnung funktioniert wie folgt:

Summe über alle erfolgreichen Alignments, auch Hits genannt (s.u.) + Gap-open-penalty + gap-extension-penalty * Länge über alle gaps

BLAST verwendet in verschiedenen Schritten sowohl die einfache, als auch die erweiterte Score-Berechnung.[15]

4.1.4 E-Wert

Der Erwartungswert, kurz *E-Wert* genannt, ist ein abstraktes Maß zur Beurteilung der Relevanz eines gefundenen Treffers zu einem gegebenen Score.

Die Berechnung des *E-Wertes* beruht auf dem sogenannten *Z-Score*, einer einfach zu ermittelnden Größe. Den *Z-Score* erhält man, wenn man vom gegebenen Score den im Mittel zu erwartenden Score bei Vergleich aller Sequenzen miteinander in einer Zufallsdatenbank abzieht und das Ergebnis durch die Standardabweichung der Scoreverteilung, die eine Zufallsdatenbank bei Vergleich aller Sequenzen miteinander liefern würde, teilt.

Die *Z-Score*-Berechnung basiert also auf einem Vergleich mit Daten, die eine Datenbank aus mit einem Zufallsgenerator gebildeten Sequenzen liefern würde. Ein *Z-Score* von 0 zu einem Score bedeutet in diese Fall, daß die beobachtete Ähnlichkeit nicht signifikant ist, denn man kann dieses Ergebnis ebenso durch den Zufall erklären.

Der E-Wert gibt letztlich die voraussichtliche Zahl der Sequenzen an, die den gleichen oder einen besseren Z-Wert zu einem gegebenen Score liefern würden, wenn man die Zufallssequenzen-Datenbank durchsucht. Weitere Information zu diesem Thema findet sich bei [16].

Grob gilt: Wenn der E-Wert einer gefundenen Sequenz kleiner ist als $1 * 10^{-50}$, handelt es sich aller Vorraussicht nach um einen engen Verwandten. Ein E-Wert zwischen $1 * 10^{-50}$ und $1 * 10^{-2}$ kann auch noch als Indiz für eine evolutionäre Verbindung interpretiert werden. Für weitere Schlussfolgerungen ist jedoch die Einbeziehung des Score (s. u.) unabdingbar. Handelt es sich um E-Werte im Bereich von 10^{-2} bis zu 1, ist es schwer, selbst mit Hilfe des Scores, Aussagen zur Verwandschaft zu machen. Bei E-Werten größer 1 kann man nicht mehr von einer Verwandtschaftsbeziehung zwischen den Sequenzen ausgehen.

Diese Einteilung ist nur als Daumenregel dienlich, hilft aber recht gut sich schnell einen groben Überblick über Suchergebnisse zu verschaffen.

4.1.5 Der Algorithmus

Der Blast-Algorithmus gliedert sich in drei sequentielle Schritte.

1. Auffinden von Hits (ähnlichen Positionen)
Der erste Schritt beginnt mit der Indexerstellung. Alle Teilsequenzen der über einen Paramter einstellbaren *Wortlänge w*, die sogenannten *w-mers*, der gegebenen Sequenz werden darin aufgenommen. Abhängig von der zugrunde gelegten Substitutionsmatrix kommen für jedes *w-mer* eventuell weitere ähnliche Teilsequenzen der Länge w für den Index hinzu, deren Score über einem konstant definierten Schwellwert liegen. Zu jedem Indexeintrag speichert der Algorithmus zusätzlich die Position des *w-mers* in der Sequenz. Die Indexerstellung dient der Beschleunigung der Suche. Für den sich anschließenden paarweisen lokalen Vergleich der gegebenen Sequenz mit allen Sequenzen aus der Datenbank sind so nur schnelle Zugriffe auf den Index notwendig.
Der paarweise Vergleich der gegebenen Sequenz mit den Sequenzen aus der Datenbank hat zum Ziel sogenannte *Hits* zu finden. Das sind Teilsequenzen, die auf Basis der verwendeten Substitutionsmatrix und dem frei definierbaren *Schwellwert T* als ähnlich zueinander gelten können.

2. Suche nach einem zweiten Hit (two-hit method)
Hat der BLAST-Algorithmus einen Hit entdeckt, so prüft er, ob sich in direkter Nachbarschaft ein zweiter Hit befindet. Der maximale Abstand der beiden Hits wird durch die beliebig definierbare Fensterlänge A

bestimmt, wobei der Abstand definiert ist durch die Positionsdifferenz der Sequenzen. *BLAST* beachtet dabei nur nicht-überlappende Hits. Alle Hits, die den zuletzt gefundenen Hit überlappen werden ignoriert.

3. Ausdehnung der Hits - High Scoring Pairs (HSP)
Gibt es einen zweiten Hit, und liegen die Scores beider Hits über dem angegebenen *Schwellwert T*, wird aus ihnen eine neue Subsequenz für die weitere Suche generiert, die beide Hits, sowie die zwischen ihnen liegende Teilsequenz enthält. Für diese Subsequenz berechnet *BLAST* den Score und dehnt sie weiter bidirektional aus. Kann die Erweiterung den Score erhöhen, merkt sich *BLAST* das neue Maximum. Die Ausdehnung bricht ab, wenn der maximale, bisher gefundene Score um X größer ist, als der aktuelle Score. Liegt der durch Erweiterung gefundenen Score über einem *cutoff score* genannten *Schwellwert S*, definiert *BLAST* die Subsequenz als high scoring pair und damit als relevant. Liegt der Score sogar über einem höheren *Schwellwert Sg*, startet *BLAST* einen zusätzlichen Erweiterungsschritt, bei dem auch gaps erlaubt sind. Eine ausfühliche Beschreibung dieses Schrittes findet sich bei [17].

4.2 PSI-BLAST

Eine Erweiterung der Datenbanksuche ist das *PSI-BLAST* (*Position Specific Iterated BLAST*). Es setzt auf der *BLAST*-Suche auf und beginnt wie sie mit einer vorgegebenen Sequenz der Länge L. Für die Suche müssen entsprechend zum *BLAST*-Suchverfahren ebenfalls alle oben beschriebenen Parameter, wie die zu verwendende Matrix, maximaler *E-Wert*, *Gap-Parameter* usw. spezifiziert werden. Im Gegensatz zur *BLAST*-Suche endet der Algorithmus jedoch nicht mit der sofortigen Ausgabe der Trefferliste, sondern führt basierend auf der Trefferliste iterativ weitere Suchen durch. Die Anzahl der Iterationen ist einstellbar.

PSI-BLAST leitet aus den gefundenen Treffern der initialen BLAST-Suche die sogenannte *Konsensussequenz* ab. Sie ist das Resultat eines Vergleiches der gefundenen Treffer. Die Konsensussequenz bildet bildlich gesprochen sozusagen die Mittelwertsequenz zu den gefundenen Treffer-Sequenzen. Um zu gewährleisten, daß die Konsensussequenz korrekt erzeugt wird, vergleicht *PSI-BLAST* zuvor paarweise alle gefundenen Treffer miteinander. Aus jedem Paar von Treffern mit mehr als 98 Prozent Sequenzidentität verwendet der *PSI-BLAST*-Algorithmus nur eine Sequenz. Diese Art von Vorfilter soll eine zu starke Gewichtung durch sehr ähnliche Sequenzen verhindern. Zudem werden alle Sequenzen die zur Suchsequenz einen größeren E-Wert als 0.01 aufweisen nicht für das multiple Alignment verwendet.

Im Detail funktioniert die Bildung der Konsensussequenz ähnlich wie der oben erklärte *BLAST*-Algorithmus. Zunächst wird nach Hits gesucht, nur statt zwischen zwei zwischen allen Treffer-Sequenzen. Aus den gefundenen Hits leitet *PSI-BLAST* die einzelnen Teil-Konsensussequenzen ab, die anschließend zu einer Gesamt-Konsensussequenz verkettet werden. Die Bildung der Teil-Konsensussequenzen funktioniert denkbar einfach: An jeder Stelle der Teil-Konsensussequenz liegt genau die Aminosäure, die am häufigsten bei den sich ähnelnden oder gleichenden Sequenzstücken zu einem Hit an dieser Stelle zu finden ist.

Die Funktionsweise von *PSI-BLAST* garantiert die Berücksichtigung von gleichen oder ähnlichen Domänen, auch an leicht voneinander differierenden Positionen. Globale Verfahren würden hier versagen.

Die entwickelte Konsensussequenz der Länge L dient anschließend zum Aufbau einer Matrix mit den zwanzig bekannten Aminosäuren, also einer Matrix mit Lx20 Einträgen. Diese Matrix heißt positionsspezifische Matrix (*PSSM*) und enthält die Wahrscheinlichkeiten in Form modifizierter log-odds-ratios für das Auftreten jeder Aminosäure für alle Positionen. Bei einer kleinen Anzahl von Vergleichssequenzen beim multiplen Alignment ist es nicht offensichtlich, wie die Wahrscheinlichkeiten aus den Häufigkeiten der Aminosäuren zu bestimmen sind. Soll etwa eine Aminosäure die an einer bestimmten Position nicht vorkommt mit einer sehr geringen Wahrscheinlichkeit für diese Position eingestuft werden, oder ist dies

als Fehler durch eine zu kleine Stichprobe zu werten, was einen höheren Wert rechtfertigen würde? Zur Lösung dieses Dilemmas korrigiert der *PSI-BLAST*-Algorithmus heuristisch die Wahrscheinlichkeiten mit Hilfe von Pseudocounts auf Basis der verwendeten Substitutionsmatrix in Abhängigkeit vom Stichprobenumfang. Genauere Informationen finden sich bei [17].

Auf der Grundlage der nun vorhandenen *PSSM* startet *PSI-BLAST* eine erneute Ähnlichkeitssuche um nach der *PSSM* ähnlichen Sequenzen zu fahnden. Resultat jeder Iteration dieses Vorgangs ist eine neue *PSSM*. Gefundene Übereinstimmungen führen so zu einer Erhöhung bestimmter Werte in der *PSSM*, wohingegen verschiedene Aminosäuren zu einer Position den Wert in der Matrix verringern. Im Idealfall konvergiert die Matrix nach einigen Iterationen. Dann bildet sie ein Profil für eine ähnliche Menge von Sequenzen. Möglich ist aber auch, daß die Matrix von einem Profil (nach scheinbarer Konvergenz) zum nächsten wandert.

Viele funktionale und evolutionäre Ähnlichkeiten von Proteinen können sicher nur über ihre Struktur erkannt werden. Die Strukturen sind allerdings sehr aufwändig zu bestimmen und liegen deshalb anteilig nur für wenige Proteine vor. Wenn die Strukturinformation fehlt, können profilbasierte Verfahren, wie *PSI-BLAST* helfen dennoch wahrscheinliche strukturelle Ähnlichkeiten zu bestimmen. Strukturell ähnliche Proteine haben auf Sequenzebene nicht unbedingt große Ähnlichkeit miteinander. Diesen Nachteil gleichen Verfahren wie PSI-Blast dadurch aus, indem sie ausgehend von einer Anfangssequenz zunächst sehr ähnliche Sequenzen suchen. Auf Sequenzebene sehr ähnliche Proteine haben mit hoher Wahrscheinlichkeit auch eine ähnliche Struktur. Diese Eigenschaft macht sich der *PSI-BLAST*-Algorithmus zu nutze um daraus ein Profil zu generieren, das strukturell ähnliche Proteine identifiziert.

4.3 Werkzeuge zur Visualisierung der Vergleichsergebnisse

Im Internet sind Dutzende von Werkzeugen frei verfügbar (siehe z.B. [18]), die Abstandsmatrizen visualisieren, wobei sie vorher teilweise noch über einen weiteren Algorithmus biologisch interpretiert werden. Besonders wichtig in diesem Zusammenhang sind Werkzeuge zur Generierung sog. phylogenetischer Bäume [19]. Bei diesen Bäumen sind die Sequenzen der Matrix in den Blättern angeordnet; über diesen wird dann versucht, einen evolutiven Baum aufzubauen, so dass nah verwandte Sequenzen einen direkten gemeinsamen Vorfahr zugeordnet bekommen. Meist wird der angenommene Abstand zum Vorfahr dann über die Länge der Verbindung zwischen den Knoten dargestellt.
Die meisten dieser Werkzeuge sind entstanden, um DNS-Vergleiche zu visualisieren. Sie lassen sich jedoch ebensogut für die Proteinvergleiche verwenden. Ein Problem mit vielen der Werkzeuge ist leider, dass sie – mglw. vor Jahren – durch das Engagement eines einzelnen Diplomanden oder Doktoranden entstanden sind, jetzt aber weder weiterentwickelt noch gewartet werden.
Im Folgenden werden zwei Werkzeuge genauer betrachtet: Phylodendron, der vom Institut derzeit benutzt wird, und Phylographer, der einen ganz anderen Visualisierungsansatz verfolgt.

4.3.1 Phylographer

Phylographer [20] visualisiert Abstandsdaten direkt, ohne daraus vorher einen Baum zu erzeugen. Dazu werden alle Punkte auf einem regelmäßigen n-Eck angeordnet und untereinander mit Linien verbunden. Die Größe des Abstands wird dabei einerseits durch die Farbe, andererseits duch die Stärke der Verbindungslinie zwischen zwei Punkten symbolisiert. Danach hat der Benutzer die Möglichkeit, einzelne Punkte selbst mit der Maus zu verschieben (und so die ursprüngliche Anordnung beliebig zu verändern). Durch seine Eigenschaften eignet sich Phylographer besonders, um stark zusammenhängende Untergruppen ("Cluster") innerhalb einer Distanzmatrix zu finden. Trotzdem erachten wir es für den normalen Einsatz als eher ungeeignet, und zwar aus folgenden Gründen:

- Phylographer erwartet alle Eingabedaten in einem Intervall zwischen 0 und 1. Ausserdem werden automatisch hohe Werte visuell hervorgehoben, während nach einem Sequenzvergleich eher die kleinen Abstände interessieren. Dies macht größere Vorbearbeitungen der ursprünglichen Daten notwendig, wobei stets auch ein Risiko besteht, die eigentliche Aussage der Daten zu verlieren.
- Im Institut ist man an die Darstellung der Daten mittels Bäumen gewöhnt. Eine Visualisierung der Abstände ohne Vorfahren und durch Farbe anstatt durch Länge der Linien würde eine große Umstellung erforden.
- Phylographer ist ein "stand-alone"-Werkzeug. Es kann weder auf einem Server installiert werden, noch kann man aus seinen Bildern auf andere Werkzeuge zugreifen.

4.3.2 Phylodendron

Phylodendron [21] ist ein Werkzeug, um phylogenetische Bäume zu zeichnen. Es berechnet die Bäume dabei nicht selber, sondern erwartet schon entsprechende Daten im Newick-Format [22], die es dann nur visualisiert. Die Bäume können aus einer Distanzmatrix z.B. durch Werkzeuge wie kitsch oder fitch aus dem Phylip-Paket [23] erzeugt werden.
Phylodendron kann eine Reihe verschiedener Bäume erzeugen, insbesondere "unrooted" und "phenogram". Seine großen Vorteile liegen jedoch vor allem darin, dass es als Web-Service eingesetzt werden kann, wobei es GIF-Image Maps erzeugt, durch die man Links aus den Bäumen auf weitere Werkzeuge setzen kann. Von Phylodendron erzeugte Bäume sind u.a. in der Darstellung der biologischen Ergebnisse unserer Fachstudie enthalten.
Wünschenswert von Seiten des Instituts wäre nun eine Erweiterung von Phylodendron (oder ein entsprechender Ersatz) gewesen, mit der man zusätzliche Attribute der Originalsequenzen visualisieren könnte - je nach Art des Attributs beispielsweise durch Farbmarkierungen oder durch weitere Labels.
Da Phylodendron ein Java-Programm ist und im Sourcecode vorliegt, haben wir zunächst untersucht, ob sich das Werkzeug einfach erweitern ließe. Dies mussten wir wegen folgender Schwierigkeiten jedoch leider verneinen:

- Phylodendron ist Teil eines erheblich größeren Frameworks, eine Trennung der Zuständigkeiten ist auf den ersten Blick jedoch nicht klar ersichtlich. Dies erschwert den Überblick.
- Phylodendron benutzt noch die API von Java 1.1. Damit die Visualisierung trotzdem gelingt, wurden große Teile eines Graphikpaketes selbst geschrieben, was das Paket deutlich komplexer macht.
- Zum Sourcecode gibt es keinerlei Dokumentation in Form von Anleitungen, Schnittstellenbeschreibungen oder Klassendiagrammen. Auch im Code selbst sind keine dokumentierende Kommentare enthalten.
- Das Werkzeug wurde mit Hilfe einer IDE (CodeWarrior) geschrieben, die sich nicht vollständig an die üblichen Java-Konventionen hält. So sind u.a. häufig mehrere öffentliche Klassen in einer Datei zusammen gefasst. Dies macht umfangreiche Wartungsarbeiten nötig, bevor der Originalcode überhaupt mit Standardübersetzern funktioniert. Die IDE selbst ist nicht frei verfügbar.
- Durch die Einbindung von Klassen des Apple-JDKs zur Übersetzungszeit ist eine Übersetzung des Codes nur auf Rechnern mit MacOS möglich.

Daher musste eine Erweiterung von Phylodendron verworfen werden.

4.3.3 Andere Werkzeuge

Wie schon erwähnt, gibt es im Internet eine Reihe von weiteren Werkzeugen, die phylogenetische Bäume visualisieren. Trotz intensiver Suche haben wir jedoch keines gefunden, das sich als Ersatz für Phylodendron anbieten würde. Dies liegt hauptsächlich an drei Gründen:

- Fast alle Werkzeuge sind Einzelapplikationen. Sie können nicht auf einem Server installiert werden und remote aufgerufen werden; auch muss man fast immer die Eingabedaten dem Programm von Hand übergeben.
- Abgesehen von Phylodendron bietet keines der untersuchten Werkzeuge die Möglichkeit, das Ergebnis mit Links zu versehen, über die man andere Werkzeuge (oder Datenbanken) ansprechen kann.
- Wir haben ebenfalls kein Werkzeug gefunden, das im Baum zusätzliche Attribute ausser der Distanz zwischen zwei Knoten und dem Namen des Knotens visualisieren kann.

4.3.4 Eigenentwicklung

Aus den oben beschriebenen Gründen denken wir, dass die einzige Möglichkeit, ein Werkzeug zu erlangen, das den gewünschten Kriterien genügt, die eigene neue Entwicklung eines solchen Werkzeuges ist. Wir schlagen als flexibelste Form dafür ein Java-Applet vor, da dies einerseits im Netz verfügbar gemacht werden kann und andererseits große Möglichkeiten der Benutzerinteraktion bietet.
Eine genaue Aufwandschätzung einer solchen Eigenentwicklung haben wir nicht vorgenommen; es ist aber klar, dass sie einerseits im Rahmen dieser Fachstudie nicht gemacht werden konnte, dass aber andererseits der Aufwand zumindest für eine einfache Version nicht besonders hoch ausfallen dürfte. Unserer Meinung nach sollte ein Studienprojekt mit fünf oder sechs Studenten problemlos eine Version erstellen können, die alle oben genannten funktionalen Kriterien erfüllt.

4.4 Frei verfügbare Programmierbibliotheken

Freie Programmierbibliotheken in Verbindung mit standardisierten Datenformaten haben einen großen Stellenwert wir sich auch bei unser Arbeit wieder gezeigt hat. Ohne die im folgenden vorgestellten Bibliotheken wäre unsere Arbeit deutlich umfangreicher gewesen.

4.4.1 BioPerl

BioPerl [24] ist ein Projekt, in dem Perl Tools aus den Bereichen Bioinformatik, Genomforschung und Biowissenschaft gesammelt werden. Voraussetzung ist Perl 5.004 ist höher, welches sowohl für Unix als auch für Windows und Mac erhältlich ist, so dass sich die Programme auch dort einsetzen lassen.
Der Download und die Installation von der Website sind problemlos gemäß der Anleitung in der Datei INSTALL durchzuführen. Da wir nur BLAST ansprechen wollten, war eine Installation der ganzen Zusatzmodule, die nicht mit BioPerl kommen aber von ihm benötigt werden könnten, nicht notwendig.
Um BLAST ansprechen zu können, müssen die Umgebungsvariablen BLASTDIR zum Verzeichnis, in dem sich (PSI-)BLAST befindet, und BLASTDB zum Verzeichnis, in dem sich die BLAST-Datenbank befindet, gesetzt sein.
BioPerl arbeitet BLAST und PSI-BLAST sehr gut zusammen, so dass nur wenige Schritte notwendig sind, um eine Suche durchzuführen und die Ergebnisse zu verwenden.
Auch für einen direkten Zugriff auf die BLAST-Datenbank im FASTA-Format stellt BioPerl Klassen zur Verfügung.
Als nachteilig hat sich die spärliche Dokumentation erwiesen, wodurch oft langwieriges Trial-and-Error notwendig wurde.

4.4.2 Biojava

Wie Bioperl ist auch Biojava [25] eine Bibliothek für die Bioinformatik. Wie der Name schon sagt, ist sie für die Programmiersprache Java [26], wobei ein JDK ab 1.3.1 vorausgesetzt wird.
Bei der Installation von Biojava sollte man nicht ein Binary von der Webseite herunterladen, sondern den anonymen CVS-Zugriff nutzen und es frisch übersetzen. So ist nicht nur der neueste Stand garantiert, sondern man bekommt vor allem auch die Beispiele und Demos, die anders gar nicht erhältlich sind.
Trotz der sehr ähnlichen Zielsetzung bestehen doch einige Unterschiede zwischen Bioperl und Biojava. Während das ältere Bioperl erheblich besser mit externen Programmen wie blast zusammen arbeitet, muss man dafür mit Biojava erst einmal einigen Aufwand betreiben (s. dazu auch [27]). Biojava hat dagegen (neben dem Vorzug, alle Möglichkeiten der Programmiersprache Java nutzen zu können) ein einfacheres Handling von Sequenzen und Annotationen.
Für die Zwecke unserer Fachstudie ist Bioperl besser geeignet gewesen.

5 Interne Analyse der LED

Mit diesem Aufgabenteil begann unsere kleine Programmieraufgabe, die man aber auch als weiterführende Evaluation ansehen kann, denn gerade die Leistungsfähigkeit von Bibliotheken wie BioJava und BioPerl in der Anwendung lässt sich gesichert erst durch praktische Arbeiten mit ihnen überprüfen.

Das ITB hatte den Wunsch ihre ca. 1200 Sequenzen umfassenden Datenbank zu validieren und zu erweitern, ersteres durch einen internen kreuzweisen Vergleich aller Sequenzen, letztderes durch eine Brückensuche im gesamten bekannten Sequenraum der Proteine, wie bereits ausführlich in der Aufgabenstellung geschildert.

Für den internen kreuzweisen Vergleich entwickelten wir die im folgenden vorgestellten JAVA-Klassen mit Hilfe der BioJava Bibliothek.
Trotz der hilfreichen BioJava-Bibliothek mussten leider noch eine Reihe von Hilfklassen geschrieben werden, damit die Benutzung von BLAST komfortabel geschehen kann. Bei allen Klassen wurde auf einen sauberen Entwurf und ausführliche Dokumentation geachtet, so dass sie einfach wiederverwendbar und erweiterbar sind. Ausserdem sind sie alle so konzipiert, dass sie skalieren und damit auch für sehr große Datenmengen geeignet sind (hauptsächlich dadurch, dass zu einem bestimmten Zeitpunkt nur kleine Datenmengen im Hauptspeicher gehalten werden, also z.B. nie eine gesamte Sequenzdatei).
Ein kreuzweiser Vergleich 'von Hand', d.h. über manuelles Eingeben jeder Sequenz in die Suchmaschine BLAST und dem anschießenden Eintragen der Ergebnisse in eine ca. 1200x1200 Zellen große Matrix, wäre so nicht möglich gewesen, wie man sich leicht vorstellen kann.

An ausführbaren Programmen wurden im wesentlichen zwei verschiedene Anwendungen erstellt: die eine, FullMatrixCreator, erstellt eine Matrix aus den E-Werten, die man erhält, wenn man für jede Sequenz in einer FASTA-Datei eine BLAST-Suche auf dieser Datei durchführt. Die andere, MatrixTransformer, bereitet eine solche Matrix für die Weiterverwendung mit anderen Werkzeugen auf. Abbildung 2 zeigt das Zusammenspiel der wichtigsten von uns erstellten Klassen.

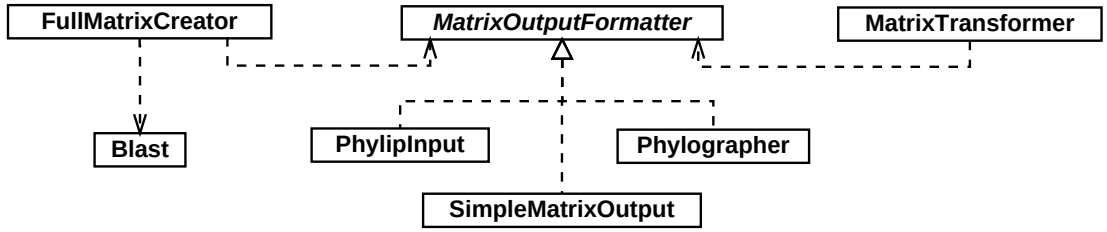

Abbildung 2: Java-Klassen

5.1 FullMatrixCreator

Die wichtigste von uns erstellte Java-Anwendung ist *FullMatrixCreator*. Mit diesem Programm kann eine volle Kreuzabstandsmatrix einer Protein-Datenbank erstellt werden. Das bedeutet: Für jede Sequenz in einer FASTA-Datei wird eine Blast-Suche durchgeführt, und zwar (um diese Suche zu beschleunigen) auf einer Datenbank mit genau den Sequenzen dieser Datei. Für jeden Treffer der Suche wird sodann der E-Value des Treffers in eine Matrix eingetragen, so dass am Ende von jeder Sequenz zu jeder anderen Sequenz der E-Value bekannt ist. Die Matrix ist übrigens *nicht* symmetrisch, d.h. der Wert von einer Sequenz A zu einer Sequenz B kann anders sein als der Wert von B zu A.
Das Ausgabeformat der Daten kann dynamisch festgelegt werden; dafür gibt man dem Programm die

Instanz von MatrixOutputFormatter an, die verwendet werden soll (mehr dazu im nächsten Abschnitt). Die Datenbank zur jeweiligen Datei muss vorher mit dem in der BLAST-Distribution enthaltenen Werkzeug 'formatdb' erstellt werden und ihr Name evtl. noch in Blast.java nachgetragen werden. Die volle Aufrufsyntax von FullMatrixCreator ist:

```
java blastsearch/FullMatrixCreator <FASTA protein file> <OutputFormat class>
```

Wir haben das Programm dazu benutzt, eine Matrix für über 1200 Sequenzen in der LED des ITB zu erstellen. Leider haben wir aber keine Möglichkeit gefunden, diese Datenmenge vernünftig zu visualisieren, sodass die biologische Auswertung der Ergebnisse noch nicht geschehen konnte.

5.2 MatrixTransformer

Um Matrizen so aufzubereiten, dass sie von anderen Werkzeugen weiterverarbeitet werden können (die häufig ein spezielles Format benötigen), haben wir die Anwendung *MatrixTransformer* erstellt. Sie nimmt eine Matrix, die in einem sehr einfachen Format ist, auf der Standardeingabe entgegen und gibt sie transformiert auf der Standardausgabe wieder aus. Dabei wird sie nie komplett, sondern immer nur zeilenweise im Speicher gehalten, was das Programm skalierbar macht. Am einfachsten lassen sich die Möglichkeiten des Programms wahrscheinlich anhand der Aufrufsyntax erklären:

```
java blastsearch/MatrixTransformer limit maxE log formatter [inputfile]
```

Die Parameter haben dabei folgende Bedeutung:

- *limit* ist der maximale E-Value in der Matrix, der direkt übernommen wird, ohne ihn zu ersetzen.
- *maxE* ist der E-Value, der für E-Values oberhalb von *limit* oder Nullwerte in der Matrix eingesetzt wird.
- *log* kann auf 'true' gesetzt werden. In dem Fall wird der Logarithmus der E-Values genommen, die entstehenden Werte nach oben verschoben, so dass keine negativen Werte mehr vorkommen, und schliesslich noch auf das Intervall [0,1] skaliert. Da der Wertebereich a priori nicht bekannt ist, wird am Ende der minimale und maximale Wert in den Daten auf stderr ausgegeben, so dass das Programm schnell auf neue Datenbereiche angepasst werden kann.
- *formatter* ist wie beim FullMatrixCreator die Klasse, die die Formatierung der Matrix vornehmen soll. Im Augenblick gibt es drei verschiedene Formatierungsarten:
  - *SimpleMatrixOutput* gibt die Matrix in einem sehr einfachen Format aus. Dieses Format wird vom MatrixTransformer beim Input erwartet.
  - *PhylipInput* gibt die Matrix in dem Format aus, das die Programme aus dem Phylip-Paket [23] wie z.B. kitsch erwarten.
  - *Phylographer* gibt die Matrix in dem Format aus, das das Programm "Phylographer" [20] benötigt. Dieses Ausgabeformat macht nur Sinn, wenn *log* auf true gesetzt wird. Zusätzlich werden die entstehenden Werte noch "invertiert", d.h. von 1 abgezogen. Phylographer braucht ausserdem noch eine zusätzliche Datei mit den Namen der Sequenzen, die von Hand erstellt werden muss.
- *inputfile* ist die Eingabedatei, die verarbeitet werden soll. Wird keine Datei angegeben, so werden die Daten von der Standardeingabe gelesen.

Die Diagonale der Matrix wird vom MatrixTransformer automatisch auf 0.0 gesetzt (1.0 also beim Phylographer-Format).

6 Identifikation von Brücken

Die Java-Programme ermöglichen dem ITB nun automatisiert schnell einen vollständigen paarweisen Vergleich der Sequenzen der LED mit Hilfe von BLAST. Damit war die interne Analyse für die LED gelöst, wenn man einmal von der bedauerlichen Tatsache der Visualisierungsproblematik absieht.
Nun stand noch die Aufgabe der Suche nach potentiellen 'Brücken'-Proteinen an. Bei dieser Suche brauchten nicht alle in der LED enthaltenen Proteine betrachtet zu werden, weil die Suche ja zum Ziel hatte weit entfernte Sequenzen zu finden. Man kann in diesem Fall annehmen, daß Suchen ausgehend von eng verwandten Proteinen, d.h. Proteinen einer Familie, zu nahezu äquivalenten Ergebnissen führen müssen.

Diesen Sachverhalt nutzten wir um die Komplexität der Untersuchung zu reduzieren und beschlossen so für die Brückensuche stattdessen nur jeweils einen typischen Repräsentanten für jede der 37 homologen Familien zu verwenden, bei deren Auswahl uns das ITB behilflich war.

Wir entwickelten nun einen in folgende Schritte gliederbaren Prozess:

1. PSI-Blast-Suche
Für jede der 37 Sequenzen wird im ersten Schritt eine PSI-Blast-Suche mit einer bestimmten festen Anzahl von Iterationen durchgeführt. Als Ergebnis erhält man eine Liste aller gefundenen Sequenzen. Jeder gefundene Sequenz verweist dabei auf den oder die Startsequenz(en) aus der Menge der 37 Repräsentanten von denen aus sie erreicht wurde.
2. Listen-Erstellung
Ergebnis des zweiten Schrittes ist eine Matrix aus den 37 Familien-Repräsentanten. Jedes Feld enthält dabei die GIs (Global Identifiers) potentieller Brücken-Proteine samt ihrer E-Werte, sofern solche aus dem ersten Schritt ableitet werden konnten.
3. Normierung
Der letzte Schritt befasst sich mit der Normierung der Matrix, um das Ergebnis mit Hilfe des Programmpaketes PHYLIP graphisch in Form eines phylogenetischen Baumes darzustellen.
Zur Umsetzung unserer Lösung verwendeten wir die Programmiersprache Perl in Verbindung mit der frei verfügbaren BioPerl-Bibliothek, die es bequem erlaubt mit BLAST zu kommunizieren. Die nun folgenden Kapitel beschreiben die einzelnen Schritte im Detail.

6.1 psiref.pl

Den ersten Schritt des Prozesses übernimmt das Perl-Programm psiref. Dieses Programm stellt das Kernstück der Fachstudie dar. Es verarbeitet die in der FASTA-Eingabedatei `ref_sequences.dat` gespeicherte Liste von Sequenzen (in unserem Falle den 37 Familienvertretern) und führt für jede dieser Sequenzen eine PSI-BLAST-Suche durch.

Wir haben uns beim Programmieren auf das Wesentliche beschränkt. So ist es zu erklären, daß wir den Namen der Eingabedatei, ihren Ort (aktuelles Verzeichnis), sowie die Parameter für BLAST im Perl-Programm fest codiert haben. Wie man dem im Anhang abgedruckten Prgramm schnell ansieht, läßt es ggfs. leicht anpassen. Trotzdem wäre als Erweiterung eine Steuerung über Aufrufparameter wünschenswert.

In der vorliegenden Form wird eine PSI-BLAST Suche mit fünf Iterationen und einem Schwellwert für den E-Wert von 3 durchgeführt. Da es uns um eine Suche im weiten Umfeld geht, wählten wir als Matrix die BLOSUM45.
Als Ergebnis liefert psiref eine Liste aller gefundenen Sequenzen, wobei ein Eintrag jeweils mit dem GI (Nummer, die die Sequenz eindeutig identifiziert) beginnt, dann ein Doppelpunkt folgt an den sich

durch Semikolon voneinander getrennt alle Namen der Eingabesequenzen anschließen, von denen aus PSI-BLAST die jeweilige Sequenz gefunden hat.
Beispiel:

```
13471376 : ESL2_MYCPN;LIPA_BACSU;Y193_HAEIN;LIP3_MORSP;PRXC_PSEPY
```

Diese Liste schreibt psiref an die Standardausgabe, so dass man sie leicht mit den Möglichkeiten der Shell in eine Datei umgeleiten kann.
Der Quelltext des Programms "psiref.pl" findet sich im Anhang.

6.2 ausw_matrix.pl

Die Ausgabe der Suche dient als Eingabe für das Perl-Programm `ausw_matrix`, das eine quadratische Matrix aus den Eingabesequenzen erzeugt. Im Falle einer Entdeckung stehen in den Zellen der Matrix die GIs der Brücken gefolgt von ihren jeweiligen E-Werten.

Wenn in der Ausgabeliste von `psiref` mehr als ein Name einer homologen Familie hinter einer GI auftaucht, so ist diese Sequenz eine mögliche Brücke zwischen allen diesen homologen Familien.

Nun ist es durchaus möglich, daß die PSI-Blast-Suche für zwei Familien mehr als einen Brücken-Kandidaten findet. Man könnte natürlich alle dieser Kandidaten als gefundene Brücken behandeln und wäre so auf der sicheren Seite keine Beziehung übersehen zu haben.

Schaut man sich die Ergebnisse allerdings näher an, stellt man fest, daß die meisten der Brücken-Kandidaten sehr unsymmetrisch zwischen zwei Familien liegen, d.h. sehr nah an der einen und sehr weit entfernt von der anderen. Unter einer Brücke verstehen wir jedoch ein Element ungefähr in der Mitte.

Unter diesen Gesichtspunkten haben wir die Anzahl von Brückenelementen zwischen jeweils zwei Familien auf maximal eine beschränkt.

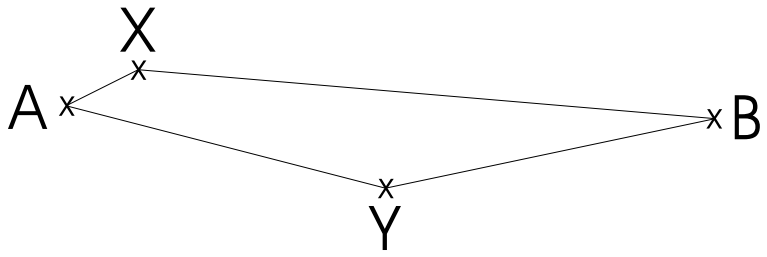

Abbildung 3: Abstand zweier Sequenzen

Bei einem E-Wert handelt es sich, wie im Kapitel über BLAST beschrieben, um eine sehr abstrakte Größe. Dies bringt es mit sich, daß eine Rechnung mit E-Werten nicht trivial ist.
Für die Auswahl einer Brücken-Sequenz aus einer Menge von Kandidaten anhand ihrer E-Werte mussten wir allerdings die E-Werte zur Errechnung des besten Kandidaten heranziehen.
Die Aufgabe der Fachstudie war es nicht mathematische Modelle für dieses Problem zu entwickeln. Deshalb versuchten wir dieses Problem auf möglichst einfache Art und Weise zu lösen.
Zuerst probierten wir es über eine einfache Multiplikation. Da die E-Werte Wahrscheinlichkeiten darstellen, könnten der E-Wert einer Brücke ausgehend von einem Familienvertreter A (siehe Abbildung 3) und

der E-Wert ausgehend von B miteinander multipliziert werden, um so die Gesamtwahrscheinlichkeit für die Verwandtschaft zu berechnen.
Liegt aber eine Brücke sehr nah an der Ausgangssequenz, wie in der Beispielabbildung, so würde das Produkt unverhältnismäßig klein werden, nahe Sequenzen würden also bevorzugt. Das gefiehl uns nicht, denn wir waren ja an Brückensequenzen ungefähr 'in der Mitte' interessiert.
Schließlich fanden wir folgende zufriedenstellende Lösung: Wie man in der Abbildung sieht stellt Y eine bessere Brücke dar, da sie mehr in der Mitte zwischen A und B liegt. Die Abstände von A und B sind ungefähr gleich.
Solche idealen Brücken-Sequenzen lassen sich nährungsweise identifzieren, wenn man jedem Brücken-Kandidaten zunächst das Maximum der beiden E-Werte zu den Familienvertretern zuweist und anschließend über alle Kandidaten minimiert, also den Kandidaten mit dem kleinsten E-Wert auswählt.

Als Ausgabe liefert das Programm dementsprechend eine Matrix mit den eventuell gefundenen Brücken-Sequenzen aus die Standardausgabe. Eine Zeile einer der Ausgabe sieht beispielsweise wie folgt aus:

```
PRXC_PSEPY|()|()|()|()|()|()|15966231(9e-53)|()|()|()|()|()|()|()|()|15966231(9e-53)|
```

Sie beginnt mit dem Namen des Repräsentanten. Danach folgen die einzelnen Zellen getrennt durch einen senkrechten Strich. '()' symbolisiert 'keine Brücke gefunden'. Andernfalls enthält die Zelle die GI gefolgt vom E-Wert in Klammern.

6.3 ausw_phylip.pl

Das Programm `ausw_matrix` gibt die GIs der Brücken und deren E-Wert aus; wurde keine Brücke gefunden, bleibt die Zelle leer. `ausw_phylip` formatiert seine Ausgabe so, dass die Programme des PHYLIP Paketes daraus einen Baum generieren können. Dazu verwendet es den 10er-Logarithmus des E-Werts und skaliert den gesamten Wertebereich automatisch auf das Intervall zwischen 0 und 10. Wurde keine Brücke gefunden, so trägt das Programm als Abstand 11 ein.
Zu benutzen ist diese Programm genau wie `ausw_matrix`, d.h. es liest eine Eingabe von der Standardeingabe lesen und schreibt das Ergebnis an die Standardausgabe.

7 Ergebnisse

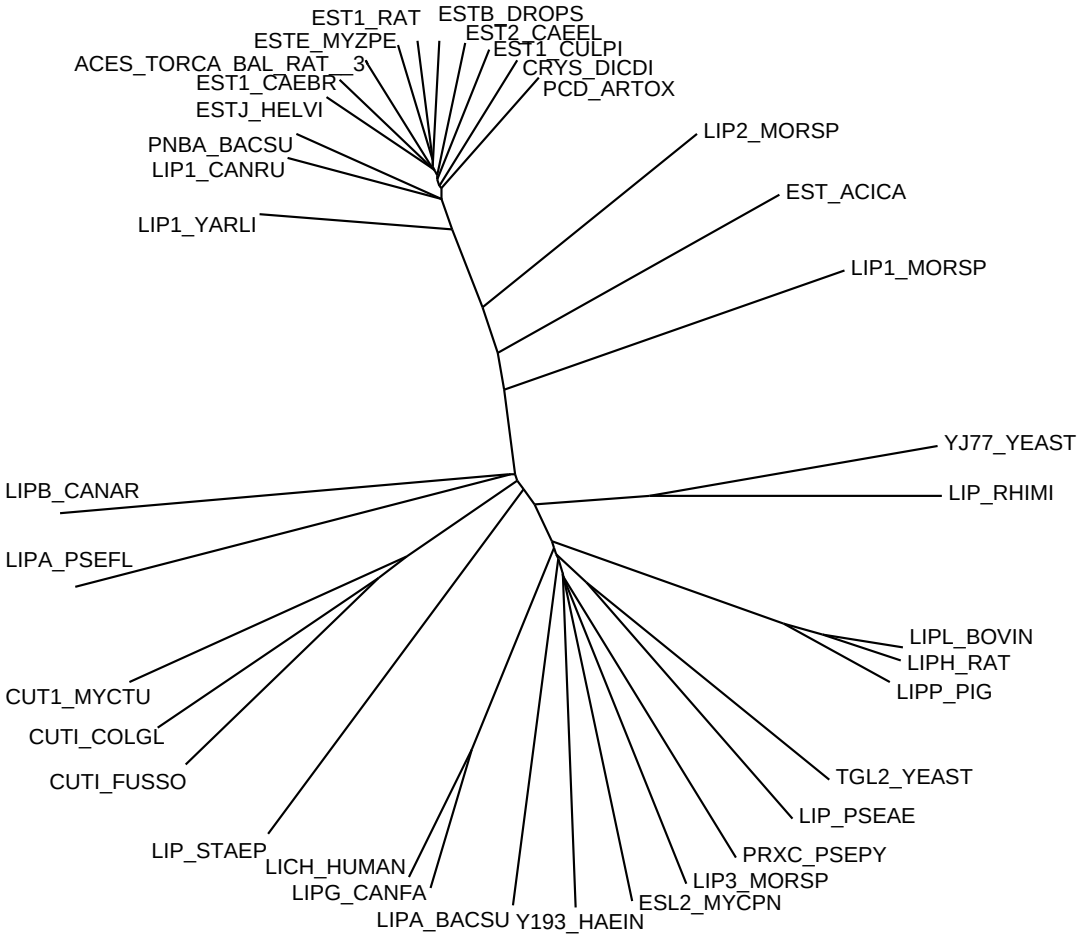

Abbildung 4: Ergebnis-Baum

Aus der Analyse der durch diese Arbeit gewonnenen Ergebnisse lassen sich zwei Klassen von Lipasen definieren, die über keine Brücke im Sequenzraum miteinander verbunden sind.
Die erste Klasse umfasst 12 Superfamilien, die sich in der Funktion und in der Sequenz zum Teil deutlich unterscheiden. Es finden sich überwiegend Lipasen aus Bakterien, Pilzen und Eukaryoten (Organismen ohne Zellkernen), aber auch Enzyme mit verwandter Funktion wie Cutinasen, Phospholipasen und Non-heme Peroxidasen.
Die zweite Klasse umfasst 5 Superfamilien, und zeigt im Vergleich zur ersten Klasse einen engeren Sequenzraum auf. Sie umfasst Esterasen aus Bakterien, Carboxylesterasen aus Eukaryoten, verschiedene Lipasen und auch Zellwandproteine wie Neuroligin, Neurotactin und Glutactin, die keine enzymatische Aktivität besitzen, aber eine wichtige Rolle im Nervensystem spielen.
Bei genauerer Betrachtung der Struktur dieser Klassen, zeigt sich, dass auch diese in zwei Gruppen aufgeteilt werden können. Zwar ist der prinzipielle Aufbau der Proteinstrukturen sehr ähnlich, jedoch unterscheiden sich die beiden Gruppen in der strukturellen Stabilisierung einer funktionell wichtigen Region. Dieser Unterschied spiegelt sich auch auf der Sequenz wieder.

Für die erste Klasse ist für diese funktionelle Region ein Muster bestehend aus der Aminosäure Glycin und einer weiteren Aminosäure X, die für die einzelnen Superfamilien erhalten ist, zu beobachten. Daher der Name GX für diese Klasse.
Im Falle der zweiten Gruppe ist ein Muster der Form Glycin-Glycin-Glycin plus eine weitere, meist hydrophobe Aminosäure zu beobachten, woraus sich der Name GGGX ableitet.
Diese Analyse umfasst noch längst nicht alle Erkenntnisse, die sich aus den Ergebnissen ableiten lassen. Es ist stark zu vermuten, daß eine tiefergehende Analyse der Ergebnisse weitere Erkenntnisse über die Homologiebeziehungen zwischen den Familien liefern.

A Quelltexte der Perl-Programme

A.1 psiref.pl

```
#!/usr/bin/perl

# PSIREF
# Führt für jede Sequenz aus der Datei ref_sequences.dat eine Suche mit
# PSI-BLAST durch und gibt auf der Standardausgabe eine Liste aus, die
# die GIs und alle Sequenznamen enthält, für die diese GI gefunden wurde.
# ref_sequences.dat ist im FASTA-Format, die erste Zeile enthält also
# den Sequenznamen mit einem ">" davor, die zweite Zeile dann die Sequenz.

# Imports
use Bio::Tools::Run::StandAloneBlast;
use Bio::DB::Fasta;

# Funktionsprototypen
sub do_blast($);
sub get_seq($);

# Hash für GIs und zugehörige Sequenznamen
%families = ();

$seqname = "";
open(SEQREF, "ref_sequences.dat");
while($in = <SEQREF>) {
  chomp($in);
  if (substr($in,0,1) eq '>') {
    # Erste Zeile mit Sequenznamen
    $seqname = substr($in, 1);
  } else {
    # Zweite Zeile mit Sequenz
    if ($seqname ne "") {
      # BLAST-Suche für die Sequenz durchführen
      %res = do_blast($in);
      foreach $i (keys %res) {
        # Sequenzname und E-Value bei allen gefundenen GIs speichern
        $families{$i} .= ';' if $families{$i};
        $families{$i} .= $seqname . "=" . $res{$i};
      }
      $seqname = "";
    } else {
      # Falscher Aufbau der Eingabedatei
      print "Sequenz hat keinen Namen!!!\n$in\n";
    }
  }
}
close(SEQREF);
```

```
# Liste geordnet nach den GIs ausgeben
foreach $i (sort keys %families) {
  print $i . " : " . $families{$i} . "\n";
}


# do_blast
# Führt die BLAST-Suche nach der übergebenen Sequenz aus und liefert einen
# Hash zurück mit GI als Key und E-Value als Wert.
sub do_blast($) {
  my $id = shift;

  # Parameter für die BLAST-Suche
  my @params = ('program' => 'blastpgp', # PSI-BLAST
                'database' => 'nr',      # Datenbank
                'I' => 'T',              # GI soll in Ausgabe sein
                'M' => 'BLOSUM45',       # Matrix
                'G' => '15',             # Gap
                'E' => '2',              # Extension
                'a' => '2');             # Zu benutzende Prozessoren
  my $factory = Bio::Tools::Run::StandAloneBlast->new(@params);

  my $input = Bio::Seq->new('-id' => "test query",
                            '-seq' => $id);
  $factory->e(3);  # Schwellwert für E-Value
  $factory->j(5);  # Anzahl Iterationen
  # BLAST aufrufen
  my $psireport = $factory->blastpgp($input);
  # Interessant ist nur das Ergebnis der letzten PSI-BLAST Runde
  my $report = $psireport->round($psireport->number_of_iterations());

  my %result = ();
  while(my $sbjct = $report->nextSbjct) {
    while (my $hsp = $sbjct->nextHSP){
      # GI aus dem zurückgelieferten Sequenznamen extrahieren
      # Einzelne Felder sind durch | getrennt
      @seqn =  split /\|/,$hsp->hit->seqname;
      $result{$seqn[1]} = $hsp->P; # score;
    }
  }

  return %result;
}

# get_seq
# Wird im Moment nicht verwendet.
# Dient dazu, für einen GI die komplette Sequenz aus der Datenbank zu lesen,
# um damit z.B. eine weitere Suche durchzuführen.
sub get_seq($) {
```

```
  my $id = shift;

  my $file = "/opt/blast/data/nr";
  my $db = Bio::DB::Fasta->new($file, -makeid=>\&make_my_id);

  return $db->seq($id);
}

# make_my_id
# Wird in get_seq verwendet, damit für die Generierung des Datenbankindex
# nur die GI verwendet wird.
sub make_my_id {
  my $desc = shift;
  my @id = split(/\|/, $desc);
  return $id[1];
}
```

A.2 ausw_matrix.pl

```
#!/usr/bin/perl

# ausw_matrix.pl
# Erstellt aus den Ausgabelisten von psiref.pl eine Matrix mit den
# Brücken zwischen zwei Sequenzen. Enthält für jede Brücke GI und E-Value.

# Prototypen
sub log10($);
sub max($$);

# Zum Speichern aller angetroffenen Sequenznamen
%names = ();
# Matrix (Hash of Hash) der E-Values für alle Sequenzpaare
%mat = ();
# Matrix (Hash of Hash), die die Brücken GI enthält
%matseq = ();

while ($in = <>) {
  chomp($in);
  # GI und Liste der Sequenzen sind durch : getrennt
  ($gi, $list) = $in =~ /(\d+) : (.*)/;
  # Einzelne Sequenzen sind durch ; getrennt
  @seq = split(/;/, $list);
  for ($i=0; $i<=$#seq; $i++) {
    # Für jeden Eintrag GI und E-Value trennen
    ($family1, $evalue1) = split(/\=/, $seq[$i]);
    # Namen abspeichern
    $names{$family1} = 1;
    # Für alle Kombinationen von Sequenzen, die für diese GI gefunden wurden,
    # evtl. Eintrag in Matrix machen
```

```
    for ($j=0; $j<=$#seq; $j++) {
      # GI und E-Value trennen
      ($family2, $evalue2) = split(/\=/, $seq[$j]);
      # Der Abstand zwischen den beiden Sequenzen ist das Maximum der
      # Abstände der Sequenzen zur Brückensequenz. Der Eintrag erfolgt, wenn
      # dafür ein neues Minimum gefunden wurde oder es der erste Eintrag ist.
      if ($mat{$family1}{$family2} > max($evalue1, $evalue2)
          or !defined $mat{$family1}{$family2}) {
        $mat{$family1}{$family2} = max($evalue1, $evalue2);
        $matseq{$family1}{$family2} = $gi;
      }
    }
  }
}

# Tabellenheader (alle Sequenznamen) ausgeben
print "|";
foreach $i (sort keys %names) {
  print $i . "|";
}
print "\n";

# Tabellenzeilen ausgeben
foreach $i (sort keys %names) {
  print $i . "|";
  foreach $j (sort keys %names) {
    # GI der Brücke und Abstand (E-Value)
    print $matseq{$i}{$j} . "(" . $mat{$i}{$j} . ")";
    print "|";
  }
  print "\n";
}

# max
# Liefert das Maximum der beiden Parameter zurück
sub max($$) {
  my ($a, $b) = @_;
  return $a > $b ? $a : $b;
}
```

A.3 ausw_phylip.pl

```
#!/usr/bin/perl

# ausw_phylip.pl
# Erstellt aus den Ausgabelisten von psiref.pl Eingabedateien für die PhyLip
# Tools kitsch, fitch und neighbor.

# Prototypen
```

```
sub log10($);
sub max($$);
sub min($$);

# Zum Speichern aller angetroffenen Sequenznamen
%names = ();
# Matrix (Hash of Hash) der E-Values für alle Sequenzpaare
%mat = ();

while ($in = <>) {
  chomp($in);
  # GI und Liste der Sequenzen sind durch : getrennt
  ($gi, $list) = $in =~ /(\d+) : (.*)/;
  # Einzelne Sequenzen sind durch ; getrennt
  @seq = split(/;/, $list);
  for ($i=0; $i<=$#seq; $i++) {
    # Für jeden Eintrag GI und E-Value trennen
    ($family1, $evalue1) = split(/\=/, $seq[$i]);
    $evalue1 = log10($evalue1);
    # Namen abspeichern
    $names{$family1} = 1;
    # Für alle Kombinationen von Sequenzen, die für diese GI gefunden wurden,
    # evtl. Eintrag in Matrix machen
    for ($j=0; $j<=$#seq; $j++) {
      # GI und E-Value trennen
      ($family2, $evalue2) = split(/\=/, $seq[$j]);
      $evalue2 = log10($evalue2);
      # Der Abstand zwischen den beiden Sequenzen ist das Maximum der
      # Abstände der Sequenzen zur Brückensequenz. Der Eintrag erfolgt, wenn
      # dafür ein neues Minimum gefunden wurde oder es der erste Eintrag ist.
      $mat{$family1}{$family2} = max ($evalue1, $evalue2)
        if $mat{$family1}{$family2} > max($evalue1, $evalue2)
        or !defined $mat{$family1}{$family2};
    }
  }
}

# Maximal- und Minimalwerte der E-Values bestimmen
$min = 100000;
$max = -100000;
foreach $i (sort keys %names) {
  foreach $j (sort keys %names) {
    $min = $mat{$i}{$j} if $mat{$i}{$j} and $mat{$i}{$j} < $min;
    $max = $mat{$i}{$j} if $mat{$i}{$j} and $mat{$i}{$j} > $max;
  }
}
# So skalieren, dass alle Werte zwischen 0 und 10 liegen
$scale = ($max-$min)/10;

# Anzahl der folgenden Zeilen ausgeben
```

```
print scalar(keys(%names)) . "\n";
# Tabelle ausgeben, beginnend mit Sequenznamen
foreach $i (sort keys %names) {
  print $i . " ";
  foreach $j (sort keys %names) {
    # Wenn keine Brücke gefunden wurde, wird 11 genommen (damit um 1 größer
    # als der Maximalwert der Abstände), ansonsten Abstände auf das Intervall
    # [0,10] skalieren
    print $mat{$i}{$j} ? sprintf(" %0.9f", ($mat{$i}{$j}-$min)/$scale) : " 11";
  }
  print "\n";
}

# log10
# Gibt den Zehnerlogarithmus zurück, bzw. -200 wenn der übergebene Wert 0 ist
sub log10($) {
  my $a = shift;
  return $a == 0 ? -200 : log($a)/log(10);
}

# max
# Liefert das Maximum der beiden Parameter zurück
sub max($$) {
  my ($a, $b) = @_;
  return $a > $b ? $a : $b;
}

# min
# Liefert das Minimum der beiden Parameter zurück
sub min($$) {
  my ($a, $b) = @_;
  return $a < $b ? $a : $b;
}
```

B Quelltexte der Java-Programme

B.1 FullMatrixCreator.java

```
package blastsearch;

import org.biojava.bio.BioException;

import org.biojava.bio.search.SeqSimilaritySearcher;
import org.biojava.bio.search.SeqSimilaritySearchResult;
import org.biojava.bio.search.SeqSimilaritySearchHit;

import org.biojava.bio.symbol.SymbolList;
```

```
import org.biojava.bio.seq.db.SequenceDB;

import java.util.Set;
import java.util.Iterator;
import java.util.HashMap;

import java.io.FileNotFoundException;

/**
 * This class reads a protein file (in FASTA format), and performs BLAST
 * searches with each entry up to a very high e-value. With the results, it
 * builds a Matrix consisting of the e-values of each protein compared to each
 * other.
 * The class can either be used as a standalone program (give the name of the
 * file as a parameter) or by other classes to use the data. In the second
 * case, the proteins need not be read from a file.
 *
 * To-Do: Better Error-Handling.
 *
 * @author Joerg Ruedenauer
 * @version 1.2
 */
public class FullMatrixCreator {

  /**
   * Constructor.
   */
  public FullMatrixCreator() {
  }

  /**
   * Reads all IDs out of a protein file in FASTA format.
   * @param filename the name of the file
   * @return all the IDs of the proteins in the file (the URNs, actually)
   * @throws FileNotFoundException if the file can't be read
   * @throws BioException if another error occurs
   */
  public Set getIDsFromFastaFile(String filename)
  throws FileNotFoundException, BioException {
    SequenceDB db = SearchFactory.fastaProt2DB(filename);
    //System.out.println("IDs: " + db.ids());
    return db.ids();
  }


  /**
   * Creates a matrix of the distances (e-values) of protein sequences. To do
   * this, n BLAST searches are performed. The DB in the second parameter
   * is used for the BLAST search, but should also contain the sequences.
   * @param sequences a Set of the IDs of the sequences
```

```
   * @param db DB that should be used for the search (given to BLAST!)
   * @param printer class that manages the output
   * @throws BioException if something goes wrong
   */
  public void createDistanceMatrix(Set sequences, SequenceDB db,
                                   MatrixOutputFormatter printer)
  throws BioException {
    int size = sequences.size();
    printer.start(sequences);
    // fill a HashMap for fast finding
    Iterator it = sequences.iterator();
    int i = 0;
    HashMap indexes = new HashMap(size);
    while (it.hasNext()) {
      indexes.put(it.next(), new Integer(i));
      i++;
    }
    // perform BLAST searches for the sequences
    SeqSimilaritySearcher blast = SearchFactory.blast();
    HashMap params = new HashMap();
    params.put("p", "blastp");
    params.put("e", "10000");
    params.put("a", "2");
    it = sequences.iterator();
    for (int j = 0; j < size; j++) {
      Double[] line = new Double[size];
      String sourceID = (String) it.next();
      SymbolList sequence = db.getSequence(sourceID);
      System.err.println("BLAST with sequence " + sourceID + "...");
      SeqSimilaritySearchResult res = blast.search(sequence, db, params);
      System.err.println("Done. Got " + res.getHits().size() + " hits.");
      int count = 0;
      Iterator hits = res.getHits().iterator();
      while (hits.hasNext()) {
        SeqSimilaritySearchHit hit = (SeqSimilaritySearchHit) hits.next();
        String targetID = hit.getSequenceID();
        // System.out.print(targetID);
        if (sequences.contains(targetID)) {
          // add Value to the matrix
          int targetIndex = ((Integer)indexes.get(targetID)).intValue();
          line[targetIndex] = new Double(hit.getEValue());
          count++;
        }
      }
      System.err.println(count + " hits were found in DB.");
      printer.print(line);
      // System.out.println();
    }
  }
```

```
/**
 * Main method, taking the proteins out of a file and printing the matrix to
 * STDOUT. Give the filename as the first parameter, and the output class
 * as second parameter.
 */
public static void main(String[] args) {
  if (args.length < 2) {
    System.out.println("Usage: FullMatrixCreator <Fasta Protein file>"
                       + " <OutputFormatClass>");
    System.out.println("Currently known OutputFormats: PhylipInput, "
                       + "SimpleMatrixOutput");
    System.exit(1);
  }
  MatrixOutputFormatter formatter = null;
  if (args[1].equals("PhylipInput")) {
    formatter = new PhylipInput(System.out);
  }
  else if (args[1].equals("SimpleMatrixOutput")) {
    formatter = new SimpleMatrixOutput(System.out);
  }
  else {
    System.out.println("OutputFormatClass not known: " + args[1]);
    System.exit(4);
  }
  FullMatrixCreator fullMatrixCreator1 = new FullMatrixCreator();
  try {
    SeqSimilaritySearcher blast = SearchFactory.blast();
    Iterator it = blast.getSearchableDBs().iterator();
    boolean found = false;
    SequenceDB db = null;
    while (it.hasNext()) {
      SequenceDB testDB = (SequenceDB) it.next();
      if (testDB.getName().equals(args[0])) {
        found = true;
        db = testDB;
        break;
      }
    }
    if (!found) {
      System.out.println("No Database " + args[0]);
      System.exit(2);
    }
    Set idSet = fullMatrixCreator1.getIDsFromFastaFile(args[0]);
    fullMatrixCreator1.createDistanceMatrix(idSet, db, formatter);
    System.exit(0);
  }
  catch (Exception e) {
    e.printStackTrace();
    System.exit(3);
```

```
    }
  }
}
```

B.2 MatrixOutputFormatter.java

```
package blastsearch;

import java.util.Set;
import java.util.Iterator;

import java.io.PrintStream;

/**
 * Interface that defines the function(s) to write a matrix
 * as specified by FullMatrixCreator.createDistanceMatrix in a
 * special format to an output stream.
 *
 * @author Joerg Ruedenauer
 * @version 1.0
 */
public interface MatrixOutputFormatter {

    /**
     * Prepare and begin the output of a matrix with the sequences
     * of the Set.
     * @param sequences the sequences
     */
    void start(Set sequences);

    /**
     * Write a new line of the matrix to the stream
     * @param line the line
     */
    void print(Double[] line);

}
```

B.3 MatrixTransformer.java

```
package blastsearch;

import java.io.*;

import java.util.Set;
import java.util.HashSet;

/**
```

```
 * Reads a matrix of e-values. Then, substitutes each null value and
 * each value higher than e certain value with another certain value.
 * Both of those values must be given as command-line arguments.
 * The matrix must be in the following format:
 * - first line the number of sequences (n)
 * - next n lines the names / ids of the sequences
 * - next n^2 lines the values, from left to right and up to down
 * The transformed matrix is then printed to STDOUT using a
 * MatrixFormatter that is also specified in the command line.
 *
 * @author Joerg Ruedenauer
 * @version 1.1
 */
public class MatrixTransformer {


    /**
     * Constructor. For better reusability, the reader is stored
     * in an attribute.
     * @param reader the reader for the data
     */
    public MatrixTransformer(BufferedReader reader) {
        this.reader = reader;
    }

    /**
     * Transforms the matrix.
     * @param limit the maximum e-value to be kept
     * @param replacement the replacement for the other values
     * @param formatter the class for the output
     * @param log if the decimal logarithm should be taken
     */
    public void transform(double limit, double replacement,
                          MatrixOutputFormatter formatter, boolean log) {
        int lineNr = 0; // for error-output
        String line = "";
        try {
            if (reader.ready()) {
                line = reader.readLine();
                lineNr++;
                int size = Integer.parseInt(line);
                Set names = new HashSet(size);
                for (int i = 0; i < size; i++) {
                    if (reader.ready()) {
                        line = reader.readLine();
                        names.add(line);
                        lineNr++;
                    }
                    else {
                        throw new IOException("Unexpected end of file.");
```

```
            }
        }
        formatter.start(names);
        Double[] values = new Double[size];
        double maxValue = 0.0;
        double minValue = 100000000;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (reader.ready()) {
                    line = reader.readLine();
                    lineNr++;
                    if (line.equals("null")) {
                        values[j] = new Double(replacement);
                    }
                    else {
                        double value = Double.parseDouble(line);
                        if (value > maxValue) maxValue = value;
                        if (value < minValue && value > 0.0)
                          minValue = value;
                        // System.err.println(value);
                        if (log && (value != 0.0)) {
                            // System.err.println(value + " ");
                            value = Math.log(value) / Math.log(10);
                            // System.err.println(value);
                            value = value + 181;
                            value = value / 190;
                        }
                        if (value > limit) {
                            value = replacement;
                        }
                        if (j == i) {
                            value = 0.0;
                        }
                        //System.err.println(value);
                        values[j] = new Double(value);
                    }
                }
                else {
                    throw new IOException("Unexpected end of file.");
                }
            }
            formatter.print(values);
        }
        System.err.println("Info: Maximum value = " + maxValue);
        System.err.println("Info: Minimum value (> 0) = " + minValue);
    }
    else {
        throw new IOException("Unexpected end of file.");
    }
}
```

```
        catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
            System.err.println("At line " + lineNr + "(" + line + ")");
        }
    }

    /**
     * Stores a BufferedReader to get the input data
     */
    private BufferedReader reader;

    public static void main(String[] args) {
        if (args.length < 4) {
            printUsage();
            System.exit(1);
        }
        double limit, maxE;
        boolean log;
        MatrixOutputFormatter formatter = null;
        BufferedReader reader =  null;
        try {
            limit = Double.parseDouble(args[0]);
            maxE = Double.parseDouble(args[1]);
            log = Boolean.valueOf(args[2]).booleanValue();
            if (args[3].equals("SimpleMatrixOutput")) {
                formatter = new SimpleMatrixOutput(System.out);
            }
            else if (args[3].equals("PhylipInput")) {
                formatter = new PhylipInput(System.out);
            }
            else if (args[3].equals("Phylographer")) {
                formatter = new Phylographer(System.out);
            }
            else {
                printUsage();
                System.exit(2);
            }
            if (args.length == 5) {
                reader = new BufferedReader(new FileReader(args[4]));
            }
            else {
                reader = new BufferedReader(new InputStreamReader(System.in));
            }
            MatrixTransformer mt = new MatrixTransformer(reader);
            mt.transform(limit, maxE, formatter, log);
            System.exit(0);
        }
        catch (IOException e) {
            System.out.println("Could not open file " + args[4] + ": ");
            System.out.println(e.getMessage());
```

```
            System.exit(3);
        }
        catch (NumberFormatException e) {
            printUsage();
            System.exit(4);
        }
    }

    public static void printUsage() {
        System.out.println("Usage: java MatrixTransformer limit maxE "
                           + "log formatter [inputfile]");
        System.out.println();
        System.out.println("limit: maximum e-value to be kept");
        System.out.println("maxE: replacement for other e-values");
        System.out.println("formatter: Class that formats the output, "
                           + "instance of MatrixFormatter");
        System.out.println("Currently known formatters: PhylipInput, "
                           + "Phylographer, SimpleMatrixOutput");
        System.out.println("log: if \"true\", the decimal logarithm of "
                           + "the values is taken.");
        System.out.println("inputfile: matrix as written by "
                           + "SimpleMatrixOutput, optional (else stdin)");
    }

}
```

B.4 SimpleMatrixOutput.java

```
package blastsearch;

import java.util.Set;
import java.util.Iterator;


import java.io.PrintStream;

/**
 * Prints the Matrix straightforward: First line the size, next n
 * lines the names (ids), next lines the values from left to right
 * and up to down.
 *
 * @author Joerg Ruedenauer
 * @version 1.0
 */
public class SimpleMatrixOutput implements MatrixOutputFormatter {

    /**
     * Constructor.
     * @param stream the OutputStream
```

```
     */
    public SimpleMatrixOutput(PrintStream stream) {
        this.stream = stream;
    }

    /**
     * Prepare and begin the output of a matrix with the sequences
     * of the Set.
     * @param sequences the sequences
     */
    public void start(Set sequences) {
        stream.println(sequences.size());
        Iterator it = sequences.iterator();
        while(it.hasNext()) {
            stream.println(it.next());
        }
    }


    /**
     * Write a new line to the stream.
     * @param line the line
     */
    public void print(Double[] line) {
        for (int i = 0; i < line.length; i++) {
            stream.println(line[i]);
        }
    }

    /**
     * Stores the stream.
     */
    private PrintStream stream;

}
```

B.5 PylipInput.java

```
package blastsearch;

import java.util.Set;
import java.util.Iterator;
import java.util.Locale;

import java.io.PrintStream;

import java.text.DecimalFormat;
import java.text.NumberFormat;
```

```
/**
 * Writes a matrix to an output stream in the input format needed by the
 * programs in the phylip package.
 * @see MatrixTransformer
 * @author Joerg Ruedenauer
 * @version 1.0
 */
public class PhylipInput implements MatrixOutputFormatter {

    /**
     * Constructor.
     * @param stream the OutputStream
     */
    public PhylipInput(PrintStream stream) {
        this.stream = stream;
        format = NumberFormat.getInstance(Locale.ENGLISH);
        if (format instanceof DecimalFormat) {
            ((DecimalFormat)format).applyPattern(
              "###0.0############;-###0.0############");
        }
    }

    /**
     * Prepare and begin the output of a matrix with the sequences
     * of the Set.
     * @param sequences the sequences
     */
    public void start(Set sequences) {
        this.sequenceIterator = sequences.iterator();
        stream.println(sequences.size());
    }


    /**
     * Write a new line to the stream.
     * @param line the line
     */
    public void print(Double[] line) {
        if (sequenceIterator.hasNext()) {
            String name = sequenceIterator.next().toString();
            while (name.length() < 10) {
                name += " ";
            }
            if (name.length() > 10) {
                System.err.println("Warning: cutting " + name
                                   + " to 10 characters.");
                name = name.substring(0, 9);
            }
            if (name.indexOf(':') != -1) {
                System.err.println("Warning: replacing : with _ in "
```

```
                                + name);
                name = name.replace(':', '_');
            }
            stream.print(name);
            for (int i = 0; i < line.length - 1; i++) {
                stream.print(format.format(line[i]) + " ");
            }
            stream.print(format.format(line[line.length - 1]));
            stream.println();
        }
    }

    /**
     * Stores an iterator for the sequences.
     */
    private Iterator sequenceIterator;

    /**
     * Stores the stream.
     */
    private PrintStream stream;

    /**
     * Stores an instance of NumberFormat to get rid of E-notation.
     */
    private NumberFormat format;

}
```

B.6 Phylographer.java

```
package blastsearch;

import java.util.Set;
import java.util.Iterator;
import java.util.Locale;

import java.io.PrintStream;

import java.text.DecimalFormat;
import java.text.NumberFormat;

/**
 * Prints the matrix in the format needed by the program
 * "Phylographer"
 *
 * @author Joerg Ruedenauer
 * @version 1.0
 */
```

```
public class Phylographer implements MatrixOutputFormatter {

    /**
     * Constructor.
     * @param stream the OutputStream
     */
    public Phylographer(PrintStream stream) {
        this.stream = stream;
        format = NumberFormat.getInstance(Locale.ENGLISH);
        if (format instanceof DecimalFormat) {
            ((DecimalFormat)format).applyPattern(
              "###0.0############;-###0.0############");
        }
    }

    /**
     * Prepare and begin the output of a matrix with the sequences
     * of the Set.
     * @param sequences the sequences
     */
    public void start(Set sequences) {
        this.sequenceNames = new String[sequences.size()];
        Iterator it = sequences.iterator();
        int i = 0;
        while (it.hasNext()) {
            String name = it.next().toString();
            if (name.indexOf(':') != -1) {
                System.err.println("Warning: replacing : with _ in "
                                   + name);
                name = name.replace(':', '_');
            }
            sequenceNames[i++] = name;
        }
        this.lineCount = 0;
    }


    /**
     * Write a new line to the stream.
     * @param line the line
     */
    public void print(Double[] line) {
        if (lineCount < this.sequenceNames.length) {
            stream.print(this.sequenceNames[lineCount] + "|");
            String zeroString = format.format(new Double(0.0));
            for (int i = 0; i < line.length; i++) {
                String value = format.format(line[i]);
                if (!value.equals(zeroString)) {
                    stream.print(this.sequenceNames[i] + ":"
                      + format.format(1.0 - line[i].doubleValue()) + "|");
```

```
                }
            }
            stream.println();
        }
        lineCount++;
    }

    /**
     * Stores the names of the sequences
     */
    private String[] sequenceNames;

    /**
     * Stores the stream.
     */
    private PrintStream stream;

    /**
     * Stores an instance of NumberFormat to get rid of E-notation.
     */
    private NumberFormat format;

    /**
     * The number of the current line.
     */
    private int lineCount;

}
```

C Referenzen

Literatur

[1] Itb. http://www.itb.uni-stuttgart.de/.

[2] Abteilung Bildverstehen. http://www.informatik.uni-stuttgart.de/ipvr/bv/bv_home.html.

[3] Institut für parallele und verteile Systeme. http://www.informatik.uni-stuttgart.de/ipvr/ipvr.html.

[4] Fakultät Informatik der Universität Stuttgart. http://www.informatik.uni-stuttgart.de/.

[5] Informationen über den Studiengang Softwaretechnik. http://www.informatik.uni-stuttgart.de/fakultaet/studienberatung/softwaretechnik/msst.html.

[6] Dr.rer.nat. Michael Schanz, Mitabeiter der Abteilung Bildverstehen. http://www.informatik.uni-stuttgart.de/ipvs/bv/personen/schanz.html.

[7] Dr.rer.nat. Jürgen Pleiss, Leiter der Bioinformatik am ITB. http://www.itb.uni-stuttgart.de/staff_pleiss.html.

[8] Prof. Dr. rer. nat. habil. Paul Levi, Inhaber des Lehrstuhls Bildverstehen. http://www.informatik.uni-stuttgart.de/ipvs/bv/personen/levi.html.

[9] Lipase Engineering Database des ITB. http://www.led.uni-stuttgart.de/.

[10] Marco Bocola. *Strukturelle und kinetische Untersuchung zum Mechanismus und Vorhersage der stereoselektiven Substraterkennung von Lipasen.* PhD thesis, Universität Marburg, 2002. http://archiv.ub.uni-marburg.de/diss/z2002/0104/dmb.pdf.

[11] Critical Assessment of Strucure Prediction. http://predictioncenter.llnl.gov/.

[12] Susanne Wohlfarth Karl-Erich Jaeger. Bakterielle Lipasen: Biochemie, Molekulargenetik und biotechnologische Bedeutung. 1993.

[13] Blast. http://www.ncbi.nlm.nih.gov/BLAST/.

[14] Dr. Jürgen Dippon. Vorlesung: Statistische Methoden der Bioinformatik, 2003. http://www.mathematik.uni-stuttgart.de/mathA/lst3/dippon/.

[15] Andrea Hansen. *Bioinformatik.* Birkhäuser Verlag, 2001.

[16] Arthur M. Lesk. *Bioinformatik.* Spektrum Verlag, 2002.

[17] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.

[18] Phylogeny Programs. http://evolution.genetics.washington.edu/phylip/software.html.

[19] Mona Singh. Phylogeny. http://www.cs.princeton.edu/courses/archive/fall01/cs551/phylogeny.pdf.

[20] Phylographer. http://www.atgc.org/PhyloGrapher/PhyloGrapher_Welcome.html.

[21] Phylodendron. http://iubio.bio.indiana.edu/soft/molbio/java/apps/trees/.

[22] Das Newick Format. http://evolution.genetics.washington.edu/phylip/newicktree.html.

[23] Phylip. http://evolution.genetics.washington.edu/phylip.html.

[24] Bioperl. http://www.bioperl.org/.

[25] Biojava. http://www.biojava.org/.

[26] Java. http://java.sun.com/.

[27] Jörg Rüdenauer. Performing local BLAST searches with Biojava. http://w3studi.informatik.uni-stuttgart.de/~ruedenjg/Software/blast/blast.html.