Analyse der Repetitiviät von Genomen

Thomas M. Hermannsdorfer

DIPLOMARBEIT

eingereicht am
Fachhochschul-Diplomstudiengang

Bioinformatik

in Weihenstephan

im Oktober 2007

Betreuer:

Prof. Dr. Bernhard Haubold
Fachhochschule Weihenstephan
Fachbereich Biotechnologie
Studiengang Bioinformatik

Zweitkorrektor:

Prof. Dr. Frank Lesske
Fachhochschule Weihenstephan
Fachbereich Biotechnologie
Studiengang Bioinformatik

Eidesstattliche Erklärung

Gemäß §23 Abs. 6 der Prüfungsordnung.

Ich erkläre hiermit an Eides statt, dass die vorliegende Arbeit von mir selbst und ohne fremde Hilfe verfaßt und noch nicht anderweitig für Prüfungszwecke vorgelegt wurde. Es wurden keine als die angegebenen Quellen oder Hilfsmittel benutzt. Wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

Freising, den 02. Oktober 2007

Thomas M. Hermannsdorfer

Inhaltsverzeichnis

Danksagung

Die vorliegende Arbeit wurde im Studiengang Bioinformatik an der Fachhochschule Weihenstephan unter Leitung von Herrn Prof. Dr. Bernhard Haubold erstellt. Sie wurde von mehreren Personen auf verschiedene Art und Weise unterstützt. Mein herzlicher Dank gilt:

Herrn Prof. Dr. Haubold für die Überlassung des Themas, sein Interesse am Gelingen der Arbeit, seine Diskussionsbereitschaft in den wöchentlichen Treffen und den vielen Gedankenanregungen, die in die Arbeit mit eingeflossen sind.
Den Administratoren des Studiengangs Bioinformatik für die Einrichtung und Überlassung des Arbeitsplatzes, sowie allen, die sich Zeit genommen haben, die Diplomarbeit vorab durchzulesen, um mir hilfreiche Verbesserungsvorschläge zu liefern.
Mein größter Dank gilt jedoch meiner Familie, meiner Frau Ruth und meinem kleinen Sohn Simon, die mich durch alle Höhen und Tiefen der letzten Monate begleitet und mich gestützt und motiviert haben, wann immer es nötig war.
Nicht zuletzt möchte ich meinen Eltern danken, die immer ein offenes Ohr für mich hatten und allen anderen Personen, die noch in irgendeiner Form zum Gelingen dieser Diplomarbeit beigetragen haben.

Kurzfassung

Sequenz-Alignments werden in der Bioinformatik als fundamentales Werkzeug verwendet, um die Funktionalität von Genen im Organismus zu erforschen, obwohl ihre Berechnungszeit zum Produkt der Länge der analysierten Sequenz ansteigt.

Eine Möglichkeit, Sequenzvergleiche ohne Alignierschritt durchzuführen, besteht in der Messung der Repetititviät von Genomen. Dazu wurde der 'Index of Repetitiveness' (I_r) von *Haubold* und *Wiehe* eingeführt, der auf dem Konzept der 'shortest unique substrings', also der kürzesten sich nicht wiederholenden Teilzeichenfolgen, basiert.

Der Vorteil dieser Methode ist die nur proportional zur Sequenzlänge ansteigende Analysegeschwindigkeit. Diese wird durch die Verwendung von Suffixbäumen erreicht, vorausgesetzt, die entsprechenden Algorithmen konstruieren die Bäume komplett im Hauptspeicher, was wiederum die verarbeitbare Sequenzgröße auf diesen beschränkt. Ziel war es deshalb, festplattenbasierende Konzepte zur Suffixbaumkonstruktion zu untersuchen und ggf. mit der Funktionalität zur Berechnung des I_r auszustatten.

Am effizientesten erschien hierfür der *'Top-Down Disk-Based'*-Algorithmus (*TDD*), von *Tian et al.* Die erwartete Suffixbaumdarstellung des ganzen Humangenoms konnte aber leider damit nicht erreicht werden, sondern maximal die von Chromosom 4 bzw. 14 (einzel- bzw. doppelsträngig). Die Zusammenfassung der vorhandenen Implementierungen von *TDD* und I_r zum Programm *TDD_IR* hatte keine weitere Reduzierung dieser Grenze zur Folge. Ein parallel von *Haubold* und *Wiehe* erstelltes, auf Suffixarrays beruhendes Programm, kann dagegen Größen bis mindestens von Chromosom 1 (doppelsträngig) verarbeiten und wurde deshalb für die abschließende Analyse verwendet.

Grundlage der Auswertung lieferte eine Beobachtung, ebenfalls von *Haubold* und *Wiehe*, nach der sich Regionen mit niedrigem I_r-Wert mit allen Genen des HOXA-Clusters auf Chromosom 7 überschneiden. Diese sind dadurch charakterisiert, dass sie sehr wenig Insertionssequenzen enthalten. Aufgrund dieser Entdeckung wurden entsprechende Regionen im ganzen Humangenom

gesucht und die gefundenen Treffer annotiert. Im nächsten Schritt wurde ein Vergleich der so erhaltenen Gene mit vorhandenen Einträgen in einer Ontologiedatenbank durchgeführt. Damit konnten hochsignifikante Assoziationen zwischen Genfunktion und niedrigem I_r in den Bereichen Nukleinsäurestoffwechsel, Transkriptionsfaktoren, Signalübertragung, sowie weiteren fünf Kategorien erkannt werden.

Eine Unsicherheit in diesem Ergebnis besteht aber u.a. noch darin, dass es sich bei dem benutzten Programm ebenfalls um eine 'in-memory'-Konstruktion mit einer Grenze von $2^{30} \approx 10^9$ Byte handelt, womit nur einzelne Chromosomen und nicht das komplette Humangenom in einem Durchlauf analysiert werden können. Ein Vergleich zweier solcher komplexer Genome ist ebenfalls nicht durchführbar. Mit der in dieser Arbeit erstellten plattenbasierenden Software sollten diese Einschränkungen überwunden werden können, sobald das von den *TDD*-Entwicklern angekündigte Update der Implementierung ihres Algorithmus zur Verfügung steht.

Kapitel 1

Einleitung

1.1 Hintergrund

Nach der erfolgreichen Entschlüsselung der Genome von Mensch und Maus im Jahr 2000 [LLB+01, VAM+01] bzw. 2001, besteht eine der größten Herausforderungen der Molekularbiologie in der Erforschung der Funktionalität der entdeckten Gene im Organismus. Der Aufklärung der Funktionen der Erbanlagen dient die Idee, dass 'ähnliche Sequenzen eine ähnliche Funktion haben' [Gus97]. In der Bioinformatik sind deshalb Sequenz-Alignments das grundlegende Werkzeug, um homologe Bereiche zu finden und damit Rückschlüsse auf funktionelle oder evolutionäre Ähnlichkeiten zu ziehen [NW70, SM81, AGM+90]. Allerdings steigt die Berechnungszeit der Alignments proportional zum Produkt der Längen der verglichenen Sequenzen an [Gus97], weshalb immer wieder nach Algorithmen gesucht wurde, die Sequenzvergleiche unter Umgehung des Alignierschrittes durchführen [Gus97].

Eine dieser Möglichkeiten besteht in der Betrachtung der *shortest unique substrings*. Dies sind *substrings* bzw. Untersequenzen oder Teilzeichenfolgen, die innerhalb der analysierten Sequenz(en) nur einmal vorkommen also *unique* sind und in ihrer Länge, d.h. am rechten bzw. 3'-Ende, nicht weiter verkürzt werden können (*shortest*), ohne ihre Einzigartigkeit zu verlieren [HPMW05]. Mittels dieser sog. 'Shustrings' wird weiter der 'Index of Repetitiveness' (I_r) berechnet, der schließlich als Maßeinheit zur Messung der Repetitivität von Genomen dient [HW06]. Da der I_r von Zufallssequenzen ungefähr gleich *Null* ist ($I_r \approx 0$) [HW06], können Abweichungen und Übereinstimmungen von und mit diesem Wert Aufschluß über Bereiche hoher und niedriger Repetitivität innerhalb eines Genoms geben. Letztere wurden in dieser Arbeit weiter verwendet, um einen Zusammenhang zwischen I_r und Funktion der betroffenen Gene zu finden, wozu die erhaltenen Ergebnisse

abschließend über eine Ontologie-Datenbank ausgewertet wurden.

Gegenüber Alignments liegt der Vorteil dieser Methode in der Geschwindigkeit, die nurmehr proportional zur Länge der Sequenz ansteigt. Zum Erreichen dieser Komplexität werden Suffixbäume zur Berechnung der 'shortest unique substrings' verwendet. Diese haben allerdings den Nachteil, dass es in der Vergangenheit für die praktische Anwendung nur schnelle Algorithmen gab, die die Bäume komplett im Hauptspeicher ablegten (z.B. [Wei73,McC76, E95] und somit in ihrer Größe auf diesen beschränkt waren [TTHP05].

Gerade, um auch verschiedene Organismen vergleichen zu können, benötigt man eine Möglichkeit, die das Hauptspeicherproblem löst, denn Gendatenverwaltungssysteme wie GenBank enthielten schon im Jahre 2005 über 52 Milliarden (56^9) Sequenzdatensätze und wachsen weiterhin exponentiell[1]. Es entstand deshalb die Idee, in jüngerer Zeit entwickelte, plattenbasierende Konzepte, auf ihre Eignung zur 'schnellen' Konstruktion von Suffixbäumen hin zu untersuchen und die effizienteste Variante mit der Funktionalität zur Berechnung des I_r auszustatten.

1.2 Zielsetzung

Ziel dieser Diplomarbeit war es also, ein geeignetes Programm zu erstellen, um den Grad der Repetitivität von großen Genomsequenzen, z.B. den menschlichen Chromosomen, zu untersuchen. Basierend auf dem Konzept der 'shortest unique substrings', also der kürzesten, sich nicht wiederholenden Untersequenzen eines Strings, soll damit der 'Index of Repetitiveness' gemessen werden können. Um 'shortest unique substrings' zu finden, bieten sich als Lösung Indexstrukturen an, die auch als Suffixbäume bekannt sind.

In dieser Arbeit wird deshalb zunächst auf die Terminologie von Strings und Suffixbäumen eingegangen. Es folgt im Rahmen der Darstellung der Zeit- und Speicherkomplexität von Suffixbäumen eine Erläuterung, welche Probleme entstehen, wenn Suffixbäume als reine 'in-memory'-Konstruktionen programmiert werden und welche Verbesserungsansätze es gibt. Getestet wurde nach einer Sichtung verschiedener Alternativen, der 'Top Down Disk-based'-Algorithmus (TDD) bzw. das zugehörige Programm *tdd*, von *Tian et al.* [TTHP05]. In diesem Rahmen wird das Prinzip des 'Top Down Disk-based'-Algorithmus und die Darstellung der Suffixbäume mit dieser Methode vorgestellt. Nach einer Einführung in 'shortest unique substrings' und dem 'Index of Repetitiveness' (I_r) wird gezeigt, wie 'Shustrings' aus den *TDD*-Bäumen ausgelesen werden können, wie die beiden vorhandenen Implemen-

[1] GenBank Statistics, http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html

tierungen zum 'Top-Down Disk-Based'-Algorithmus[2] und 'Index of Repetitiveness'[3] miteinander zum Programm *TDD_IR* verknüpft wurden und welche Möglichkeiten und Grenzen dabei auftraten. Es kann vorweggenommen werden, dass zwar deutlich größere Bereiche als mit den oben erwähnten *in-memory*-Konstruktionen darstellbar waren, aber das versprochene Ziel der Suffixbaumkonstruktion für das ganze Humangenom leider nicht erreicht werden konnte.

Deshalb wurde zur Auswertung ein parallel von *Haubold* und *Wiehe* erstelltes Programm namens *ir* verwendet [HW06], dass auf dem Suffixarraykonstruktionsalgorithmus 'Deep-Shallow' von *Manzini* und *Ferragina* [MF04] und einer Implementierung von *Fischer et al.* [FVK06] beruht und in der Lage ist, alle Sequenzen bis zu Chromosom 1 (doppelsträngig) vollständig zu analysieren.

Zum Schluß werden das Vorgehen und die Ergebnisse der Auswertung anhand der I_r-Werte vorgestellt, die sich mit der Frage beschäftigen, ob ein Zusammenhang zwischen der Funktion von Genen und niedrigen I_r-Bereichen besteht.

Da es sich bei der derzeitigen Implementierung von I_r allerdings auch um eine '*in-memory*'-Konstruktion mit einer Grenze von $2^{30} \approx 10^9$ Byte handelt und eine 64-Bit-Version derzeit noch nicht verfügbar ist[4], kann nicht das komplette Humangenom auf einmal analysiert und auch keine vergleichende Suche, z.B. zwischen zwei solch großen Genomsequenzen in einem Lauf, durchgeführt werden. Dies sollte wiederum mit der plattenbasierenden Version möglich sein, sobald das von den *TDD*-Entwicklern angekündigte Update der Implementierung ihres Algorithmus zur Verfügung steht.

Im Anhang findet man deshalb eine Kopie der Online-Dokumentation zu dem Programm *TDD_IR*, sowie ein ausführliches Klassendiagramm, das den Zusammenhang zwischen den aus *TDD* verwendeten und den selber erstellten Klassen illustriert.

[2] http://www.eecs.umich.edu/tdd/

[3] http://adenine.biz.fh-weihenstephan.de/ir/

[4] lt. Auskunft von G. Manzini

Kapitel 2

Strings und Suffixbäume

2.1 Aufbau und Funktion von Suffixbäumen

Suffixbäume sind vielseitige Datenstrukturen, die effiziente Lösungen zahlreicher Probleme im Bereich der Stringverarbeitung ermöglichen, denn sie beschreiben eine kompakte Repräsentation aller Suffixe eines Strings S in einem Baum. Bevor auf Suffixbäume selber eingegangen wird, soll zum besseren Verständnis vorab die Terminologie von Strings erläutert werden.

2.1.1 String Terminologie

2.1.1.1 Definitionen

Definition 1: Ein String S der Länge n besteht aus einer Sequenz von n Buchstaben aus einem gegebenen Alphabet $\sum$. Die Elemente dieses Strings werden von 1 bis n nummeriert [Jap04].

Definition 2: Der i-te Präfix eines Strings S besteht aus den ersten i Buchstaben von S, $S[1..i]$ [Jap04].

Definition 3: Ein Substring eines Strings S ist eine Sequenz von fortlaufenden Buchstaben aus S, $S[i..j]$ [Jap04].

Anmerkung: Für einen String S der Länge n gilt, dass die Anzahl der Substrings gleich $\binom{n+1}{2}$ ist [HPMW05].

2.1.1.2 Sequenzdaten

Biologische Sequenzdaten können wie eine Sammlung von eindimensionalen Strings von verschiedener Länge behandelt werden [Jap04]. Im Folgenden wird ein solcher String einfach als Sequenz bezeichnet. Die Buchstaben dieser Strings werden aus einem spezifischen Alphabet gezogen, dass mit der chemischen Zusammensetzung der Sequenz korrespondiert [Jap04]. Für diese Arbeit werden DNA-Sequenzen betrachtet, deren Elemente aus einem Vier-Buchstaben-Alphabet $\sum = \{A, G, C, T\}$ bestehen und deren Länge in Basenpaaren (bp) ausgedrückt wird.

2.1.1.3 Was ist ein Suffix?

Definition 4: Der i-te Suffix eines Strings S der Länge n besteht aus den letzten $n - i + 1$ Buchstaben von S, $S[i..n]$ [Jap04].

Anmerkung: Für einen String S der Länge n gilt, dass die Anzahl der Zeichen aller Suffixe von S gleich $\binom{n+1}{2}$ ist [Jap04].

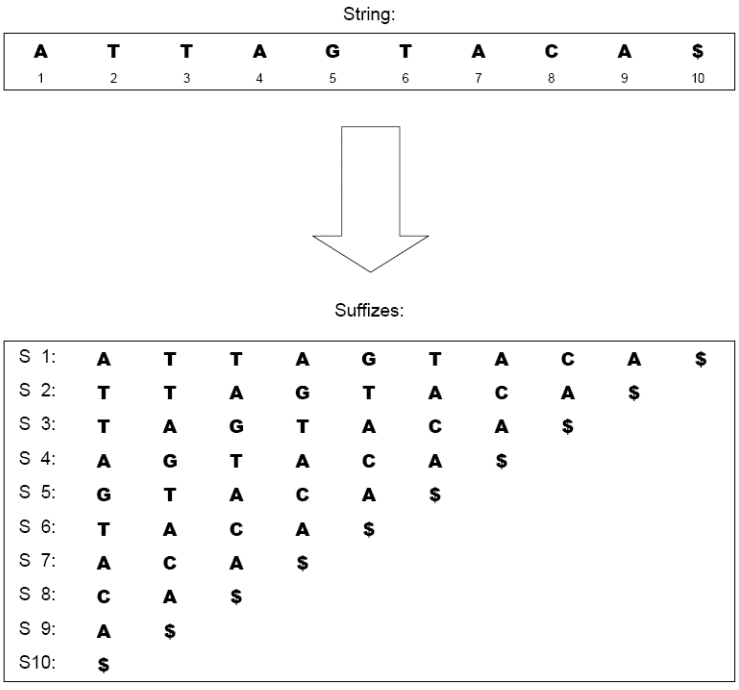


Abbildung 2.1: Die einzelsträngige Beispielsequenz *ATTAGTACA$* und zugehörige Suffixe.

Bezogen auf Gen-Sequenzen ist der Suffix Nummer i, einer Sequenz S, die Teilsequenz, die mit dem i-ten Nukleotid $S[i]$ von S beginnt und bis an das Ende von S reicht. Eine Sequenz hat genau so viele Suffixe wie Nukleotide. Abbildung 2.1 zeigt die nach Position sortierte Liste der Suffizes der Sequenz *S=ATTAGTACA$*. Das Terminierungssymbol '$' wird im nachfolgenden Abschnitt über Suffixbäume näher erklärt.

2.1.2 Was ist ein Suffixbaum?

Alle Suffixe einer n-Buchstaben langen Sequenz S, können in eine Baumstruktur eingetragen werden, die Suffixbaum heißt. Es handelt sich dabei um einen gerichteten Graphen, mit einer Wurzel und genau n Blättern, die je nach verwendeter Konvention von *1* bis n (z.B. [HW06, Gus97]) bzw. von *0* bis $n-1$ (z.B. [TTHP05, PM]) durchnummeriert werden[1].

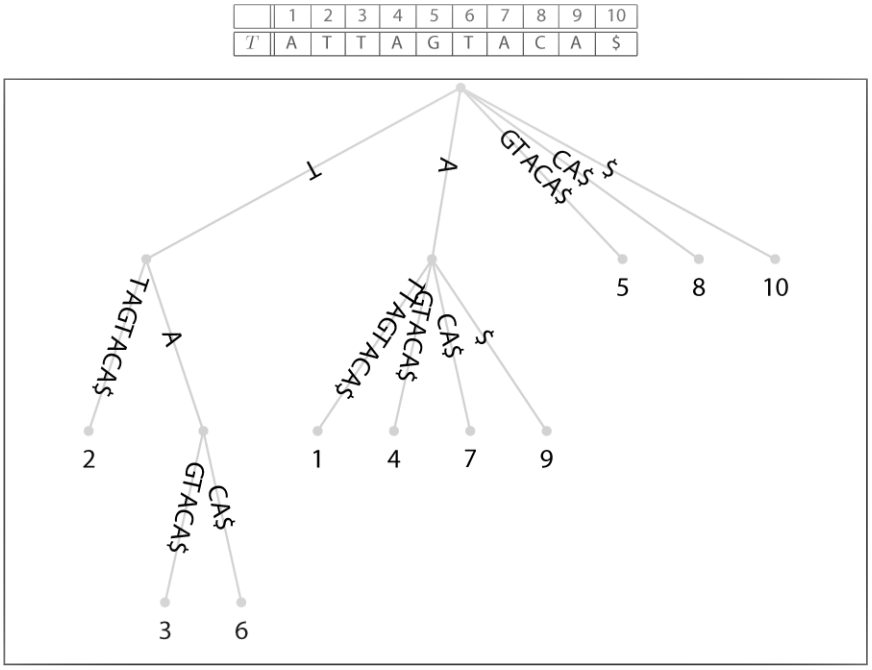


Abbildung 2.2: Suffixbaum für die Sequenz *ATTAGTACA$*. Erstellt mit dem Tool *drawStrees*, von B. Haubold, das unter *http://adenine.biz.fh-weihenstephan.de/drawStrees* erhältlich ist.

[1]In dieser Arbeit werden beide Möglichkeiten verwendet, um dem unterschiedlichen Einsatz in den zentralen *Papers* ([HW06] (*Index of Repetitiveness*) und [TTHP05, THP04] (*Top Down-disk based*-Technik)) jeweils Rechnung zu tragen

Abbildung 2.2 zeigt den Suffixbaum für die Beispielsequenz *S=ATTAGTACA$*. Die Verzweigungen des Baumes heißen (interne) Knoten und die Zweige selber werden als Äste oder Kanten bezeichnet. 'Oben' ist die Wurzel und an den Zweigenden stehen die Blätter. Jeder interne Knoten hat mindestens zwei Kinder und jede Kante erhält als Beschriftung ('Label') einen nichtleeren Substring bzw. eine Untersequenz von S [Gus97]. Keine zwei Kanten die von einem Knoten ausgehen, können am Anfang mit demselben Buchstaben beschriftet sein. Die Schlüsseleigenschaft des Suffixbaumes ist, dass für jedes Blatt i der Suffix von S wiedergegeben wird, der an der Position i beginnt, also $S[i..n]$. Dies geschieht, indem die Kantenbeschriftungen auf dem Pfad von der Wurzel zum jeweiligen Blatt verknüpft werden [Gus97]. Der Suffixbaum hat somit für jeden Suffix genau ein Blatt. Ohne dem Terminierungssymbol '$' könnte es sein, dass kein kompletter Suffixbaum konstruiert wird. Betrachtet man z.B. den verkürzten Strings *S'=ATTAGTA$*, würde nämlich der Suffix *TA* gleichzeitig ein Präfix von *TAGTA* sein und somit statt an einem Blatt, innerhalb der entsprechenden Kante des längeren Suffix enden. Um zu vermeiden, dass also ein Suffixpfad nicht in einem anderen Pfad endet, benützt man ein Terminierungssymbol, dass nicht in dem Alphabet von S enthalten ist. Weit verbreitet ist eben die Verwendung von '$' [Gus97].

Zusammengefaßt kann ein Suffixbaum also folgendermaßen beschrieben werden:

Ein Suffixbaum T für einen String S mit n Symbolen, ist ein Baum mit folgenden Eigenschaften:

- T ist ein Baum mit einer Wurzel (root) und n Blättern, die von $1..n$ markiert sind.
- Jede Kante E wird mit einer nichtleeren Teilzeichenfolge (Substring) von S beschriftet (Label oder Kantenmarkierung) $\Rightarrow |Label(E)| \neq 0$.
- Jeder innere Knoten K hat mindestens zwei Kinder.
- Alle Kanten, die von einem inneren Knoten K oder der *Wurzel* ausgehen, beginnen mit einem unterschiedlichen Zeichen.
- Für jedes Blatt i in T ergeben die aneinandergehängten Beschriftungen der Kanten auf dem Pfad von der *Wurzel* zu i das Suffix von S, das an Index i beginnt ($S[i..n]$).
- Ein String S der Länge n und somit der zugehörige Baum T, enthalten $\binom{n+1}{2}$ Substrings (s.a. 2.1.1.1). Wenn ein Substring α am Punkt p endet, gibt die Anzahl der darunter liegenden Blätter Auskunft über die Häufigkeit von α innerhalb von S wieder und die jeweiligen Blattbeschriftungen entsprechen den Startpositionen von α in S.

2.1.3 Repräsentation einer Beispielsequenz

Anhand von Abbildung 2.2 soll nun der Baum für die Beispielsequenz S dargestellt und beschrieben werden. Um den letzten Gedanken der Zusammenfassung gleich nochmal aufzugreifen, betrachte man den Substring 'T', der am äußersten linken Zweig, am ersten inneren Knoten endet. Darunter liegen drei Blätter, womit der Buchstabe 'T' dreimal in S vorkommt. Analog dazu verhält es sich mit dem Substring 'TA', der an dem nächsten inneren Knoten in diesem Zweig endet und somit nach Anzahl der darunter liegenden Blätter zweimal vorkommt. Diese Substrings beginnen jeweils an den Positionen, die den Bezifferungen der Blätter entsprechen (2,3,6 bzw. 3,6). S hat genausoviele Suffizes wie Nukleotide, also in diesem Beispiel 10 (das Terminierungssymbol '$' wird miteingerechnet), was auch der Anzahl der Baumblätter entspricht. Ein Suffix entspricht der Kantenbeschriftung von der Wurzel bis zu dem entsprechenden Blatt. Z.B. steht an Position/Blatt *4* der Suffix *AGTACA$*.

2.1.4 Verwendung von Suffixbäumen

Ein Suffixbaum ist hervorragend geeignet, um ein Muster (≡Pattern) in einer Sequenz zu finden, zum Beispiel $P{=}AGT$ in der Sequenz S. Man beginnt an der Wurzel und wählt den Zweig, der mit dem Buchstaben 'A' beginnt, gelangt zum nächsten Knoten, wählt dort den Zweig, der mit dem Buchstaben 'G' beginnt und findet schließlich das Muster 'AGT' auf diesem Zweig wieder. Die Suffixnummer am Ende des Zweiges sagt aus, dass P in Position *4* von S auftritt. Die Anzahl der notwendigen Buchstabenvergleiche war *drei*, also gleich der Anzahl der Buchstaben in P. Die Entscheidung an jedem Knoten benötigt maximal $|\sum|$ Schritte ($|\sum|$ = Größe des Alphabets), d.h. für Genomdaten maximal *vier* Schritte. Wenn man den Suffixbaum eines vollständigen Chromosoms aufgebaut hat, braucht man für jede Nukleotidsequenz der Länge n also nur $n * |\sum|$ Entscheidungen und kann somit in zur Länge des Musters linearen Zeit ermitteln, ob und wo sie überall im Chromosom enthalten ist. Dies macht Suffixbäume zu einer geeigneten Datenstruktur z.B. für folgende fundamentale Einsatzmöglichkeiten in der Bioinformatik:

- Sequenzhomologie über exaktes und angenähertes Substringmatching[2]
- Auffinden von Repeats in DNA über längste gemeinsame Substrings
- Erkennen und Auffinden von DNA-Verunreinigungen

Gusfield zählt insgesamt 17 grundlegende Verwendungsmöglichkeiten auf, die damit allerdings noch nicht erschöpft sind [Gus97]. Um diesen Umfang zu erreichen, muß die Konstruktion von Suffixbaumen natürlich in 'annehmbarem' Zeit- und Platzbedarf möglich sein.

2.1.5 Konstruktion eines Suffixbaumes

Es wird zunächst ein einfach zu verstehendes aber zeitaufwändiges Verfahren geschildert, dass sich an dem WOTD-Algorithmus (für *write-only-top-down*) orientiert (Auf Algorithmen mit geringerer Zeitkomplexität wird im anschließenden Abschnitt hingewiesen). Die Methode fügt die einzelnen Suffixe S_i, der Länge $n - i + 1$ von der Wurzel beginnend (daher 'top-down') nacheinander in den Suffixbaum ein. Es sei $S{=}ATTAGTACA\$$ die Sequenz, von der man den Suffixbaum erstellen will. Der Suffix *1* ist die Sequenz selbst. Man trägt ihn entlang des ersten Zweiges ('A'-Zweig) ein. Suffix *2* beginnt mit $TTAGTACA\$$ und sein Eintrag, ausgehend von der Wurzel, liefert den zweiten Zweig ('T'-Zweig). Beim Suffix *3* ($TAGTACA\$$) steigt man zuerst den

[2] effiziente Lösungen hierfür findet man in z.B. in [Hun04] oder [TTHP05]

'T'-Zweig hinab. Nach dem ersten 'T' wird innerhalb dieser Kante der erste Knoten und von dort ausgehend ein weiterer Zweig eingefügt (vgl. Abb. 2.3). Die restlichen Suffixe werden analog eingetragen. Bei noch unbetrachteten Anfangsbuchstaben wird ein neuer Zweig erstellt[3]. Trotz der Einfachheit dieses Verfahrens, lassen die ggf. vielen Vergleiche von einzelnen Buchstaben, z.B. bei einer längeren, repetitiven Sequenz, nun einen hohen Zeitaufwand vermuten, weshalb die Komplexitäten ermittelt werden sollen.

2.2 Probleme und Lösungsansätze zur Suffixbaumdarstellung

2.2.1 Komplexität des naiven Algorithmus

Der Zeit- und Raum-Bedarf des WOTD-Algorithmus soll nun analysiert werden. Die oben beschriebenen Einfügungsschritte müssen $(n-1)$-mal für n Suffixe durchgeführt werden und jeder Schritt von S_i zu S_{i+1} hat eine lineare Zeitkomplexität $O(|S[(i+1)..n]|)$, ist also proportional zur Länge des einzufügenden Suffix. Im schlechtesten Fall braucht diese Methode $O(n^2)$ Zeit, um einen Suffixbaum zu erstellen [Gus97].

Da es auch n Blätter gibt und jeder Pfad zu einem Blatt beschriftet ist, benötigt die Baumkonstruktion und -darstellung auch $O(n^2)$ Raum bzw. Speicher [PM].

2.2.2 Lösungsansätze

Diese Komplexitäten konnten durch verschiedene Algorithmen auf $O(n)$ reduziert werden. Giegerich und Kurtz [GK97] geben hier einen sehr guten Überblick über die richtungsweisenden Ausführungen von Weiner [Wei73], McCreight [McC76] und Ukkonen [E95]. Auf letztere Technik soll etwas näher eingegangen werden, weil sie als 'jüngste' der drei Methoden die Vorteile der anderen beiden Algorithmen in sich vereint und zusätzlich die Möglichkeit bietet, 'online' zu arbeiten, d.h. es werden iterativ die Suffixbäume für wachsende Textlängen konstruiert.

Hilfreich können an dieser Stelle folgende Vorüberlegungen sein. Da jeder interne Knoten eines Suffixbaumes verzweigt wird, können nicht mehr als $n-1$ innere Knoten vorhanden sein. Die maximale Anzahl von Kindern pro Knoten entspricht der Länge des Alphabets (hier: $|\sum| = 4$). Klar wird

[3]Eine ausführlichere Beschreibung findet man in [Gus97]

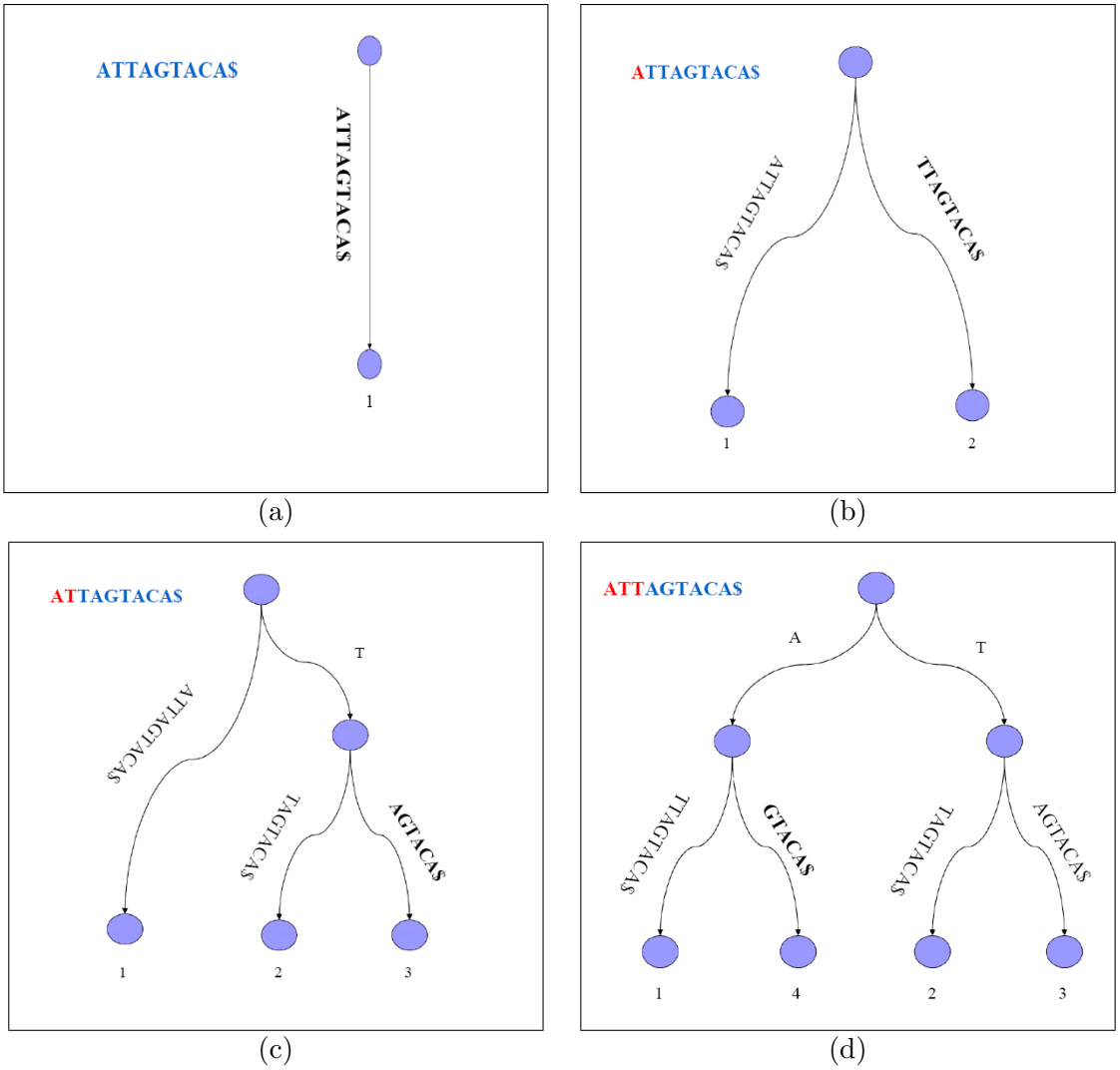


Abbildung 2.3: Suffixbaumkonstruktion der ersten 4 Suffixe der Sequenz *ATTAGTACA$*. Die abgearbeiteten 'Suffixstarts' erscheinen im String rot, die restlichen blau. Man beachte, dass sich die Anordnung der Zweige hier etwas zu Abb. 2.2 unterscheidet, was aber am Wesen des Suffixbaumes nichts ändert. Vgl. [PT05].

dadurch, dass die Anzahl der Kanten um *Eins* weniger ist, als die Summe der Anzahl aus inneren Knoten, Blättern und der Wurzel.

2.2.2.1 Speicherbedarf

In einem Baum muss man im Wesentlichen die Kinder eines Knotens speichern [Heu05]. Indem die Wurzel, Knoten und Blätter bei Suffixbäumen nicht beschriftet werden, benötigt man für jedes dieser Elemente nur konstanten Speicherbedarf. Wenn man die Beschriftung $S[i..j]$ einer Kante nicht durch die jeweiligen Zeichenreihen, sondern durch das Paar der Start- und Stop-Indizes (i, j) darstellt (sog. Referenzen auf das Originalwort), kann jede Kante ebenfalls in konstantem Raum gespeichert werden. Z.B. wird die in der Beispielsequenz S von der Wurzel ausgehende Kante $GTACA\$$ durch (5,10) kodiert. Dadurch ist der erforderliche Platz für den ganzen Baum durch eine lineare ($O(n)$) Komplexität charakterisiert.

2.2.2.2 Zeitbedarf und Suffixlinks

In Ukkonen's Algorithmus wird die im letzten Abschnitt beschriebene Methode der Kantenkodierung als 'edge encoding' bezeichnet. Um ein lineares Zeitverhalten zu erreichen, wurden noch zwei weitere Techniken, nämlich 'skip-and-count' und 'suffix links', eingeführt. Dazu gibt es mehrere ausführlichere Erläuterungen z.B. [E95] und [Gus97], auf die hier verwiesen werden soll. Bezüglich der Verarbeitung großer Sequenzen sind auch primär nur die 'suffix links' oder Suffixzeiger interessant, weshalb diese zum weiteren Verständnis kurz erläutert werden.

Bei der Konstruktion des Suffixbaumes werden zusätzlich zu den regulären Baumkanten Hilfskanten eingefügt. Diese sog. Suffixzeiger werden für jeden inneren Knoten definiert. Sie zeigen von einem Knoten, der das Wort $S[i..j]$ repräsentiert, zu einem Knoten, der das Wort $S[(i + 1)..j]$ repräsentiert. Der Suffixzeiger der Wurzel zeigt auf einen gesonderten Knoten $\perp$, der als Fehlerzustand verstanden wird, wenn man die Bäume als Zustandsgraph eines Automaten betrachtet. Es sei z.B. $x\alpha$ ein beliebiger String, wobei x einen einzelnen Buchstaben bezeichnet und α einen (möglicherweise leeren) Substring. Für einen internen Knoten v mit der Pfadbeschriftung $x\alpha$, gibt es einen anderen Knoten $s(v)$ mit der Pfadbeschriftung α und einen Zeiger von v nach $s(v)$, der *suffix link* genannt wird [Jap04]. Für die Beispielsequenz S ist dies in Abbildung 2.4 aufgezeichnet.

Damit liefert Ukkonen einen O(n), *in-memory-* Konstruktionsalgorithmus, basierend auf der klugen Beobachtung, dass die Konstruktion eines Suffixbaumes durch iteratives Expandieren der Blätter von einem teilweise

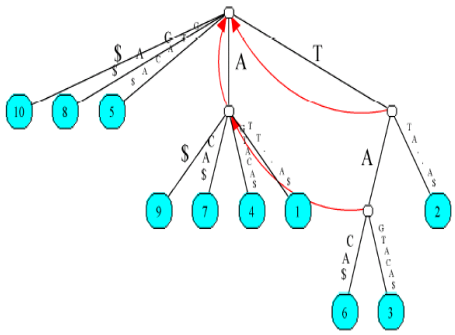


Abbildung 2.4: Suffixbaum mit rot eingezeichneten Suffixzeigern für die Sequenz *ATTAGTACA$*. Erstellt mit einem Tool von I. Ahmetov, das unter *http://is.ifmo.ru/vis/ukkonen/Ukkonen_en.html* erhältlich ist.

konstruierten Suffixbaum ausgeführt werden kann, indem man mittels *Suffixlinks* die Unterbäume schnell durchläuft und dann den $i+1$-sten Buchstaben zu den Blättern des Suffixbaumes hinzuaddiert, der von den vorherigen i Buchstaben aufgebaut worden ist [TTHP05].

Problem der *Suffixlinks* ist, dass sie eine schlechte Lokalität der Speicherreferenzierung haben, wegen der es zu einer schlechten Performance auf gecacheten Architekturen und auf Platte kommt [TTHP05]. Sollte der Baum größer als die verfügbare Hauptspeichermenge werden, werden neben den Kanten und Knoten auch diese Suffixzeiger auf Festplatte gespeichert bzw. ausgelagert ('Swapping'). Durch diesen unkontrollierten Festplattenzugriff nehmen die Zugriffszeiten auf die ausgelagerten Baumelemente eine hohe Komplexität an. Ohnehin ist die 'access time' 10^5 bis 10^6 mal langsamer als die Zeit, um den internen Hauptspeicher des Computers zu nutzen [CF99]. So verlangsamt sich der Algorithmus deutlich oder wird sogar unpraktikabel [TTHP05]. Nicht zuletzt deshalb sind in der Vergangenheit immer wieder Versuche gemacht worden, Algorithmen zu entwickeln, die eine effiziente Festplattennutzung beinhalten. Einige davon werden im nächsten Abschnitt vorgestellt.

2.3 Suffixbäume auf Sekundärspeicher

2.3.1 Hintergrund

Die im letzten Abschnitt genannten Algorithmen sind gut geeignet für kleine *Inputstrings*, für die der Baum komplett im Hauptspeicher konstruiert werden kann. Der Nachteil ist jedoch, dass sie aufgrund ihrer schlechten Speicherreferenzierung für die Anwendung auf Sekundärspeicher ungeeignet sind.

Gleichzeitig können bei biologischen Fragestellungen Suffixbäume häufig so groß werden, dass sie nicht mehr im Hauptspeicher abgelegt werden können [Japp04].

Dass diese Grenze schnell erreicht ist, zeigt z.B. *Kurtz* in [Kur99], wo ein 'in-memory'-Algorithmus vorgestellt wird, der im Durchschnitt 10.1 Bytes pro *input character* und 20 Bytes im schlechtesten Fall benötigt[4]. Er liegt damit mehr als 8 Byte pro *Eingabezeichen* unter den Besten vorhergehenden Techniken. *Kurtz* zeigte in dem gleichen Artikel auch, dass eine Darstellung des Humangenoms mit seiner Methode $1,22 * 10^{10}$ *Integers* oder $4 * 1,22 * 10^{10}/2^{30} = 45,31$ Gigabytes an Hauptspeicher benötigt. Damit würde eine Konstruktion nur auf 64-Bit-Maschinen gelingen, die mit ausreichend RAM ausgestattet sind.

Stehen weniger 'Memory' oder gar nur die derzeit noch weit verbreiteten 32-Bit-Maschinen[5] zur Verfügung, muss man beachten, dass Teile des (wachsenden) Baumes ab und an auf Platte ausgelagert werden müssen, um z.B. das humane oder murine Genom (*6 GB* bzw. *5 GB* doppelsträngig) als Suffixbaum darstellen zu können. Hierzu wurden bereits einige sog. 'disk-based'-Konstruktionen als Variationen zu den herkömmlichen Methoden entwickelt.

2.3.2 Auswahl an Alternativen

Bedathur und *Harits* schufen z.B im Jahr 2004 eine Pufferstrategie, die sie *TOP-Q* nannten. Diese soll die Performance von Ukkonen's Algorithmus (einschließlich Suffixzeiger) verbessern, wenn er auf Platte benutzt wird [BH04]. Außerdem soll ebenfalls eine 'online-Erweiterung' (s. Abschnitt 2.2.2) möglich sein. Die zur Verfügung stehende Implementierung konnte allerdings von mir unter den zur Verfügung stehenden Maschinen nicht fehlerlos kompiliert und auf Testdaten angewandt werden. Auch aus dem zugehörigen *Paper* konnte ich nicht ersehen, ob *TOP-Q* ein geeigneter Mechanismus ist, um große Suffixbäume zu konstruieren. Erstens war das im dortigen Experiment genutzte Datenvolumen deutlich kleiner als z.B. bei Hunt (70 Mbp verglichen mit 286 Mbp), zweitens wurde kein Vergleich zwischen *TOP-Q* und anderen Methoden vorgestellt. Da ich zudem die Kompilierfehler nicht in annehmbarer Zeit beseitigen konnte, sah ich von einer weiteren Verwendung von *TOP-Q* ab.

Dagegen befürworten Giegerich, Kurtz und Stoye in [GKS03] den *WOTD*-Algorithmus und untersuchen die Vorzüge einer 'faulen' ('lazy') Implementierung bzw. Entwicklung von Suffixbäumen. Die Autoren argumentieren hier,

[4]Der Speicherbedarf pro Zeichen ist von der Anzahl der inneren Knoten abhängig.

[5]Die maximal ansprechbare Arbeitsspeichermenge bei 32-Bit-Maschinen beträgt 4 GB [RAM].

dass man die vollen Konstruktionskosten vermeiden kann, wenn man Unterbäume ('subtrees') nur entwickelt, wenn sie das erste Mal geöffnet bzw. benötigt werden. Diese Methode ist nützlich, wenn eine kleine Anzahl an Abfragen oder nur kurze Abfragen gestellt werden [TTHP05]. Für eine große Anzahl bzw. lange Abfragen muss ein Großteil des Baumes erstellt werden, was die Durchführung verzögert und verschlechtert [TTHP05].

Erwähnt werden soll noch der Algorithmus von Hunt, der ebenfalls keine *suffix links* mehr, sondern auch einen $O(n^2)$-Algorithmus benutzt [HAI01]. Die Leistungsfähigkeit wird hier durch eine bessere örtliche Speicherreferenzierung ('locality of memory reference') erreicht [TTHP05], was bedeutet, dass möglichst versucht wird, auf benachbarte Stellen im Hauptspeicher bzw. auf der Festplatte zuzugreifen, um möglichst wenig Sprünge und somit geringere Zugriffszeiten zu erreichen. Dazu wird der Suffixbaum in Partitionen zerlegt und zwar so, dass Suffixe mit gleichem Präfix in einem Teilbaum liegen. Die Präfixe werden so gewählt, dass jeder Teilbaum garantiert in den Hauptspeicher passt. Über eine auf $JAVA^{TM}$ basierende Objektspeicherplattform namens *PJama* [AJ98] wird der *subtree* auf Festplatte gespeichert. Der komplette Baum kann letztlich aus den Teilbäumen zusammengesetzt werden. Dazu werden die Suffixe des Teilbaums von der Festplatte gelesen, im Hauptspeicher zusammengebaut, am Ende geschrieben und nie mehr geladen. Dadurch hat man sehr wenig I/O-Zugriffe und den Hauptspeicher jedesmal wieder neu zur Verfügung. Diese Technik ist erfolgreich benutzt worden, um 286 Mbp an DNA-Sequenzdaten zu indizieren.

Auf die Nachteile dieser Methode weisen *Hunt et al.* bereits selber in [HAI02] hin. Die Konstruktion eines unabhängigen Teilbaumes bzw. einer Partition erfordert jedesmal das Lesen der kompletten Sequenz. Außerdem bringt die verwendete statische Prä-Partitionierungs-Methode mittels Präfixen fester Länge Probleme mit sich. Die Buchstaben in realen DNA Sequenzen sind nämlich nicht gleich verteilt, weshalb einige Partitionen in den Hauptspeicher passen, während es andere möglicherweise nicht tun [PM]. Damit hätte man dann wieder das 'Swapping-Problem' und würde durch die ohnehin quadratische Komplexität der Teilbaumkonstruktion ein sehr ineffizientes Programm erhalten. Außerdem verursacht dieser Algorithmus bei den Durchläufen zur Konstruktion des kompletten Baumes eine große Anzahl von zufälligen Zugriffen, was ebenfalls eine starke Beeinträchtigung darstellt.

Für weitere alternative *in-memory-* und *on-disk*-Algorithmen sei auf die Ausführungen von *Tian et al* [TTHP05] verwiesen.

Auf die zwei Methoden von *Giegerich et al* und *Hunt et al* wurde deshalb kurz eingegangen, weil Teile daraus im 'Top-Down Disk-Based'-Algorithmus ('TDD') von *Tian et al* verwendet worden sind. *TDD* wurde zuerst in [THP04] vorgestellt und für diese Arbeit weiter verwendet, weil er der bis dahin einzige

Algorithmus war, von dem berichtet wurde, dass er auf das gesamte einzelsträngige Humangenom anwendbar sei. Grundlagen hierfür waren eine Partitionierungsstrategie, die ähnlich wie in [SS03] beschrieben, die Strategie von *Hunt et al* verbessert und eben die *WOTD*-Technik von *Giegerich et al*, die in dem Programm *wotdeager* implementiert wurde. Ähnlich wie bei *Hunt et al* verzichtet die *Top-Down Disk-Based*-Methode auch auf die Verwendung von *Suffixlinks* im Austausch für eine bessere 'locality of reference' (siehe oben). Dass *TDD* dabei auch schnell arbeitet, zeigt ein Vergleich in [TTHP05], wo sogar der Algorithmus von Ukkonnen bei der Konstruktion im Hauptspeicher geschlagen wird, wenn Systeme mit *Cache*-Architektur verwendet werden. Im nächsten Kapitel wird das *Top-Down Disk-Based*-Verfahren nun näher beschrieben.

Kapitel 3

Top-Down Disk-Based Algorithm (TDD)

Der *TDD*-Ansatz ist in zwei Phasen unterteilt. Die Kombination aus den im letzten Abschnitt genannten Verfahren dient der Suffixbaumkonstruktion und wird in *TDD* als 'Partition and Write Only Top Down' ('PWOTD') bezeichnet. Der $O(n^2)$-Algorithmus wird dabei durch eine Partitionierungsphase verbessert, die einen schnellen Aufbau von großen, unabhängigen Teilbäumen im Hauptspeicher erlaubt [TTHP05]. Dazu kommt ein ausgeklügeltes Pufferungsverfahren, mit dem die verwendeten Datenstrukturen optimal ersetzt und Puffer mit wahllosem ('random') Zugriff möglichst im Hauptspeicher gehalten werden sollen [Jap04]. Im folgenden wird nun auf beide Phasen näher eingegangen.

3.1 PWOTD

Diese erste Phase ist auch wieder in zwei Schritte unterteilt. Zuerst findet die Partitionierung statt, bei der die Suffixe in Partitionen mit gleichen Präfixen einer bestimmten Länge eingeteilt werden. Für jede Partition wird dann, nachdem sie sortiert worden ist, als Zweites der *wotdeager*-Algorithmus von *Giegerich et al* zur Konstruktion der Teilsuffixbäume durchgeführt. Die Komplexität des *wotdeager*-Algorithmus ist im 'worst-case' $O(n^2)$, was z.B. bei einer aus lauter gleichen Buchstaben bestehenden Sequenz der Fall ist. In dieser Situation berechnet sich der Zeitbedarf entsprechend dem des naiven Algorithmus aus Kapitel 2.2.1. Die Zeitkomplexität im 'expected case' beträgt $O(n * log(n))$ [GKS03]. Wenn so für jede Partition ein Suffixbaum erstellt ist, werden diese zu einem kompletten Suffixbaum zusammengefügt.

3.1.1 Die Partitionierung

Im ersten Schritt werden die Suffixe des *Input-Strings* in $|\sum|^{prefixlen}$ Partitionen eingeteilt, wobei $|\sum|$ die Alphabetgröße des Strings und *prefixlen* die Tiefe der Partitionierung bzw. die gegebene Präfixlänge darstellt. Der Partitionierungsschritt wird folgendermaßen ausgeführt. Der Eingabestring wird von links nach rechts gescannt. An jeder Position i werden die *prefixlen* Buchstaben benutzt, um eine der $|\sum|^{prefixlen}$ Partitionen zu bestimmen. Dieser Index i wird dann in einen dafür berechneten Partitionspuffer (s. Abschnitt 3.1.3) geschrieben. Am Ende des Durchlaufs enthält jede Partition Zeiger[1] auf Suffixe, die alle den gleichen Präfix der Größe *prefixlen* haben. Die Anzahl der Partitionen ist dabei viel kleiner, als die Länge des Strings und die Zeitkomplexität beträgt $O(n \times prefixlen)$ [TTHP05] .

Wenn für jede Partition ein Suffixbaum erstellt ist, werden alle Suffixbäume zusammengefügt. Am Beispiel aus Kapitel 2.1.2 Abbildung 2.2, hat man eine Alphabetgröße von vier (DNA-Code) und wählt eine Präfixlänge von eins. Dadurch ergeben sich vier Partitionen. In jeder Partition stehen also nur Suffixe, die mit dem gleichen Zeichen beginnen. Am Beispiel des im letzten Kapitel verwendeten Beispielstrings *S=ATTAGTACA$* kommen in die erste Partition mit dem Präfix 'T' die Suffixe (1), (2), und (5), wenn man bei *0* zu zählen beginnt[2]. Weil diese die ersten *prefixlen* Zeichen gemeinsam haben, sind die für die Partition 'T' zu konstruierende Suffixe (2), (3) und (6).

Das Zusammenfügen der einzelnen Suffixbäume geschieht, indem man einen Suffixbaum für die ersten *prefixlen* Zeichen erstellt und an den Endknoten dann die jeweiligen Suffixbäume mit dem passenden Präfix anfügt. Abbildung 3.1 zeigt dies für das vorgestellte Beispiel.

Um den Partitionierungsschritt nochmal zu illustrieren, wird wieder der String *ATTAGTACA$* betrachtet, für den das Ganze folgendermaßen aussehen würde (Abb. 3.2). Unter Verwendung einer *prefixlen* von *1* werden *vier* Partitionen von Suffizes erstellt, *eine* für jedes Symbol im Alphabet[3].

Die Suffixpartition für den Buchstaben 'A' würde {0,3,6,8} sein, und die Suffizes {ATTAGTACA$, AGTACA$, ACA$, A$} darstellen. Für 'T' würde das {1,2,5} und {TTAGTACA$, TAGTACA$, TACA$} ergeben, für 'C' {7} und {CA$}, sowie schließlich für 'G' {4} und {GTACA$}.

[1] Beachte: Diese Suffixzeiger sind nicht zu verwechseln mit den *suffix links* von Ukkonen, wie in 2.2.2.2 beschrieben.

[2] Beachte: Für dieses Kapitel wird die Notation verwendet, die die Strings bzw. Suffixe und Blätter ab 0 durchnummeriert, entsprechend der Verwendung in [TTHP05, THP04].

[3] *DNA*-Kodierung, die finale Partition, die nur aus dem *String terminator Symbol* '$' besteht, wird ignoriert.

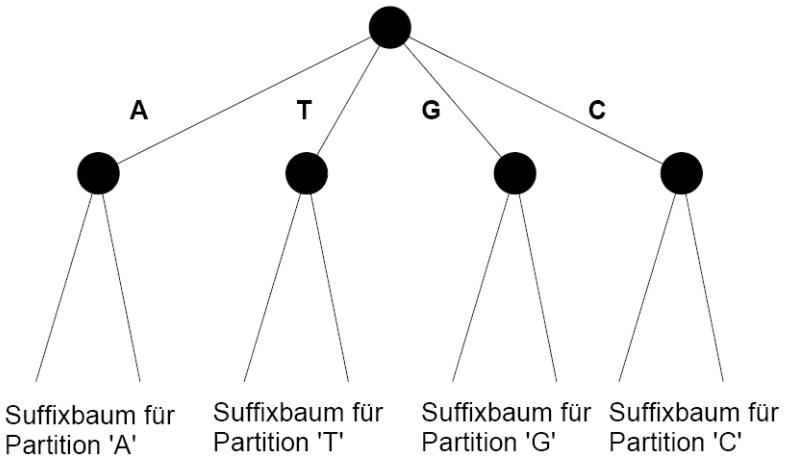


Abbildung 3.1: Schema der Partitionierung für eine DNA-Sequenz.

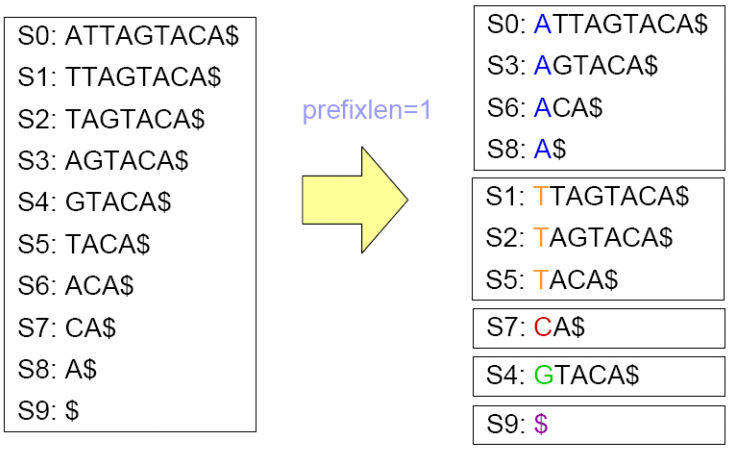


Abbildung 3.2: Partitionierung der Suffixe der Beispielsequnz S in $|A|^{prefixlen}$ (hier: $4^1 = 4$) Partitionen. Aus [PT05]. Es sei hier nochmals darauf hingewiesen, dass im Gegensatz zu Kapitel 2 die Indizierung bei 0 beginnt.

Der Pseudo-Code für den PWOTD-Algorithmus ist in Abb. 3.3 dargestellt. Daran sieht man schon, dass die Partitionierung in Phase *eins* von PWOTD einfach ist, während es für den *wotdeager*-Algorithmus in Phase *zwei* einer weiteren Diskussion bedarf.

Algorithm PWOTD(*String.prefixlen*)
Phase1:
Scan the *String* and partition *Suffixes* based on the first *prefixlen* symbols of each suffix
Phase2: Do for each partition:
1. START BuildSuffixTree
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on first symbol using *Temp*
4. Output branching and leaf nodes to the *Tree*
5. Push the nodes pointing to an unevaluated range onto the *Stack*

While *Stack* is not empty
6. Pop a node
7. Find the Longest Common Prefix(LCP) of all the suffixes in this range by checking the *String*
8. Sort the range in *Suffixes* on the first symbol using *Temp*
9. Write out branching nodes or leaf nodes to *Tree*
10. Push the nodes pointing t an unevaluated range onto the *Stack*

11. END

Abbildung 3.3: Pseudocode für den PWOTD-Algorithmus. Aus [TTHP05].

3.1.2 Der *wotdeager*-Algorithmus

Im zweiten Schritt wird nun der *wotdeager*-Algorithmus benutzt, um für jede Partition einen Suffixbaum unter Verwendung einer *Top-Down*-Konstruktion aufzubauen.

Der *PWOTD*-Algorithmus erfordert *vier* Datenstrukturen, um Suffixbäume zu erstellen: Ein Stringarray für die Eingabe, ein Suffixarray, ein temporäres Array und den Suffixbaum. Für die folgenden Abschnitte werden diese Strukturen vereinfacht *String*, *Suffixes*, *Temp* und *Tree* genannt.

Das Suffixarray wird als Erstes mit den Suffixen einer Partition besetzt, nachdem die ersten *prefixlen* Buchstaben entfernt wurden. Als Beispiel soll

wieder der String *ATTAGTACA$* mit *prefixlen=1* dienen. In Abbildung 3.4 kann man die Konstruktion des Suffixarray für die *T*-Partition betrachten.

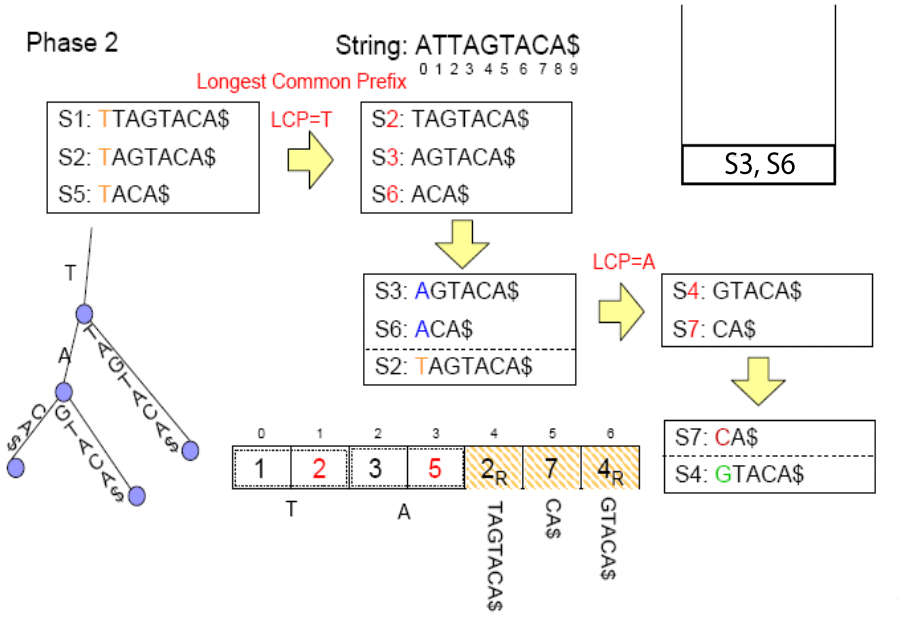


Abbildung 3.4: Aufbau des Suffixarrays für die T-Partition des Beispielstrings. Aus [PT05].

Die Suffixe in dieser Partition stehen an den Positionen *1*, *2* und *5*. Da sich alle diese Suffixe das gleiche Präfix, 'T', teilen, addiert man *eins* zu jedem 'offset', um das neue Suffixarray {2,3,6} zu produzieren. Die Anfangsbuchstaben der Suffixe lauten {T,A,A}. Im nächsten Schritt sortiert man das Suffixarray nach diesen ersten Buchstaben. *TDD* verwendet einen Algorithmus namens 'count-sort', der die Sortierung in linearer Zeit durchführt (für eine konstante Alphabetgröße) [TTHP05]. Dazu muß auf den String zugegriffen werden, um festzustellen, welches Zeichen an der jeweiligen Position steht. Die so produzierten Suffixzeiger werden im *Temp*-Array abgespeichert und alphabetisch sortiert. Man erhält dann das geordnete Suffixarray {3, 6, 2}.

Nun kann in linearer Zeit überprüft werden, ob es sich um einen Verzweigungs- bzw. inneren Knoten oder um ein Blatt handelt. Dazu muss nur getestet werden, ob mehrere Suffixe mit dem selben Zeichen beginnen. Diese müssen sich nach dem Sortieren nebeneinander befinden. Ist dies der Fall, handelt es sich um Knoten, ansonsten um ein Blatt.

Unter Punkt vier des Pseudocodes (Abb. 3.3) werden Blätter sofort in

den *Tree* geschrieben und für die Knoten wird an einer bestimmten Stelle Speicher im *Tree* für sie selber und ihre Kinder reserviert [TTHP05].

Im obigen Beispiel handelt es sich bei {3, 6} um einen Verzweigungsknoten, da die Suffixe an Position *3* und *6* beide mit dem Präfix 'A' beginnen. Unter Punkt fünf werden dann alle Verzweigungsknoten auf einen Stapel oder 'Stack' gepackt, da diese noch weiter bearbeitet werde müssen. Als einziger Knoten wird also {3, 6} auf den *Stack* gegeben. Die Punkte sechs bis zehn müssen nun für jeden Verzweigungsknoten des Suffixbaumes durchgeführt werden. Im Beispiel befindet sich nur {3, 6} auf dem Stack und wird somit entsprechend Punkt sechs vom Stack geholt. Unter Punkt sieben wird nun in linearer Zeit der LCP (longest common prefix) von {3, 6} bestimmt. Da sich im String an Stelle *vier* ein 'G' und an Stelle *sieben* ein 'C' befindet, ist der LCP = 1. Somit ergibt sich für die Kantenbeschriftung 'A', da der LCP die Länge *eins* hat. Es entstehen zwei neue Suffixe, nämlich {4} und {7}. Diese werden unter Schritt acht wieder lexikographisch sortiert, womit man im Beispiel die Suffixe {7, 4} erhält. Im Schritt neun werden wie im Schritt fünf alle Verzweigungsknoten und Blätter in den Suffixbaum eingetragen. Im Beispielfall erhalten wir zwei neue Endknoten. Unter Punkt zehn würden nun alle neuen Verzweigungsknoten wieder auf den Stack gepackt werden und die *while*-Schleife würde erneut durchlaufen werden. Dies entfällt allerdings, da für diese Partition keine weiteren Verzweigungsknoten existieren. D.h., die Berechnung ist solange fortgeschritten, bis alle Knoten erweitert worden sind und der *Stack* leer war. Wenn zu jeder Partition ein Suffixbaum erstellt und schließlich der komplette Suffixbaum zusammengefügt worden ist, ist die *Array*-Darstellung abgeschlossen (s. Abb. 3.5).

Zum Abschluß des POWTD-Algorithmus soll die vollständige graphische Darstellung der Repräsentation des Suffixbaumes zum *String ATTAGTACA$* zusammen mit dem korrespondierenden Array im Speicher erläutert werden. Das Array folgt der kompakten Darstellung wie in *wotdeager* und benötigt somit 8.5 Bytes pro indiziertem Symbol (näheres siehe [GKS03]). Die orange schattierten Kästchen stehen für ein Blatt und die weißen Kästchen für einen internen Knoten. Im Speicher wird dies durch ein 'Flag-Bit' unterschieden, welches den Wert 1 oder 0 erhält und somit angibt, um welche Art Knoten es sich handelt. Ein weiteres *Flag* gibt an, ob es sich bei einem Knoten um den rechtesten Kindknoten handelt. Ist dies der Fall, so hat der aktuelle Knoten keine weiteren Kinder und es folgen die Kinder eines anderen Knotens. In Abbildung 3.5 sind solche Knoten mit einem 'R' in der rechten unteren Ecke gekennzeichnet. Ein Blatt wird durch eine einzige Integerzahl dargestellt, während ein Knoten zwei benötigt. Handelt es sich um ein Blatt, so hat es als Kantenbeschriftung den Teilstring, der von der angegebenen Position bis zum Ende des Strings reicht. Beim internen Knoten beinhaltet der erste Ein-

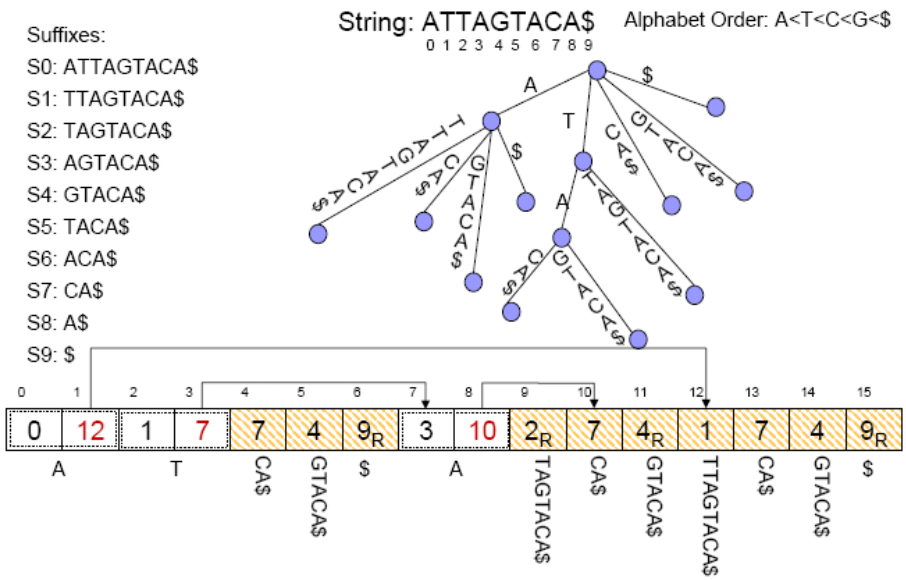


Abbildung 3.5: Der komplette Sufixbaum für den String *ATTAGTACA$* mit der zugehörigen Array-Darstellung. Aus [PT05].

trag einen Index auf den Eingabestring und der entsprechende Buchstabe an dieser Position ist der Beginn des in den Knoten eingehenden Kantenlabels [TTHP05]. Der zweite Eintrag zeigt zum ersten Kind. Somit kann man herausfinden, mit welchem Teilstring man vom Vaterknoten zum Kind kommt. Die Länge des Strings der Kantenbeschriftung entspricht nämlich der Differenz aus Startposition des Vaterknotens und der kleinsten Startposition aller Kindknoten.

In diesem Beispiel sieht man, dass der interne Knoten, der durch die Einträge *0* und *1* im Baum-Array dargestellt wird, vier Blattknoten als Kinder hat, die an den Einträgen *12*, *13*, *14*, und *15* lokalisiert sind. Der Elternsuffix beginnt am Index *0* im String, während die Kind-Suffixe mit den Indizes *1*, *7*, *4* und *9* beginnen. Dadurch kennt man die Länge der Beschriftung der Elternkanten, die nämlich $min\{1, 7, 4, 9\} - 0 = 1$ ist. Die Blätter erfordern nur den Startindex ihrer Kantenbeschriftung (das Ende ist das Terminierungssymbol '$') und benötigen somit weniger Speicherplatz.

Über diese *Label*-Längen kann man schließlich auch die Ausdehnung der in der Einleitung erwähnten *shortest unique substrings* ablesen, wie später in Kapitel 4.2.1 nochmals genauer erläutert wird. Hier befindet sich also die Schnittstelle, um sich *TDD* zur Berechnung des *Index of Repetitiveness* zunutze zu machen.

Bevor allerdings auf *shortest unique substrings* und dem I_r eingegangen wird, soll noch die Pufferungsstrategie näher betrachtet werden, die den *Top-Down Disk-Based*-Algorithmus so effizient macht.

3.1.3 Pufferungsstrategie

Erwähnenswert ist dies deshalb, da ja, im Gegensatz zu dem $O(n)$ Algorithmus von Ukkonen, *TDD* als Basis den theoretisch weniger effizienten $O(n^2)$ Algorithmus von *Giegerich et al* [GKS03] verwendet und trotzdem sogar auf Hauptspeicherebene von *gecacheten* Architekturen schneller ist, als vorhergehende *in-memory*-Techniken [TTHP05]. Wie man gleich sehen wird, besteht der Hauptunterschied darin, das bei *Ukkonen* die *String*-Daten sequentiell geöffnet werden und dann der Suffixbaum durch unwillkürliche Durchläufe aktualisiert wird, während bei *TDD* zufällig auf den Eingabestring zugegriffen und der Baum sequentiell geschrieben wird.

Dies geschieht, indem nach der optimalen Ersetzungsstrategie für die vier Datenstrukturen (*Tree*, *Suffixes*, *Temp*, *String*) gesucht und dann der verfügbare Hauptspeicher möglichst effizient unter den vier Datenstrukturen aufgeteilt wird. Das ist wichtig, weil die Daten mit einer schlechten 'locality of reference', also unwillkürlichem Zugriff, möglichst mit der größten Hauptspeichermenge alloziert werden sollen, während die Daten mit einer guten 'locality of reference' auch ohne Probleme auf der Festplatte abgespeichert werden können.

Die Ersetzungsstrategien lassen sich recht einfach testen, indem man sie auf verschiedenen Datenbanken anwendet und anhand der 'Buffer Cache Misses' auswertet. So sieht man in Abbildung 3.6, dass sich für den *Temp*-Puffer die *MRU*-, also 'Most recently used'-Ersetzungsstrategie sehr gut eignet. Diese Grafik basiert auf Experimenten, die von *Tian et al* mit *Swiss-Prot*-Einträgen durchgeführt wurden. Dadurch fand man ebenso heraus, dass für den *Tree*- und *Suffix*-Puffer, die Ersetzungsstrategie *LRU*, also 'Least recently used' optimal ist und für den *String*-Puffer sowohl *LRU* als *RANDOM* (Random replacement algorithm) vernünftig ist [TTHP05].

Die einzelnen Puffergrößen im Hauptspeicher sind nochmal in der Graphik 3.7 (a) abgebildet. Zudem die Größenverteilung der einzelnen Dateien auf der Festplatte 3.7 (b), zum besseren Verständnis auf den Beispielstring S bezogen. Daran sieht man, dass bei einer Stringgröße von n sowohl das *Suffix*-Array, als auch das *Temp*-Array eine Größe von $4n$ benötigen, da man 4 Byte große Integerzahlen zur Adressierung braucht. Der Suffixbaum beansprucht bei der benutzten Darstellungsform ca. $8 - 12n$ Speicher [TTHP05]. Der *String* besitzt die schlechteste *locality of reference*, da zum Beispiel beim Sortieren der *Suffixe* immer auf ihn zugegriffen werden muss, um herauszu-

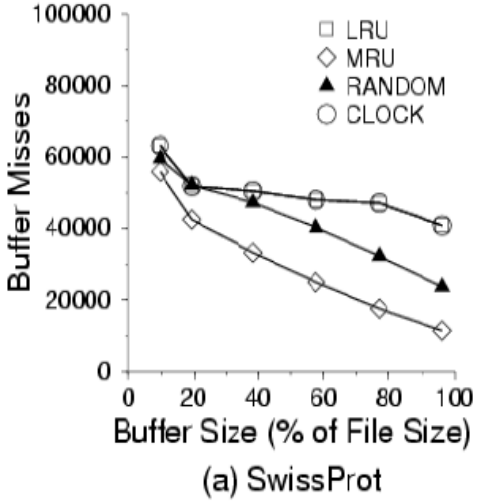


Abbildung 3.6: Kurve der 'Buffer Cache Misses' der *TDD*-Datenstruktur *Temp*, abhängig von ihrer Puffergröße (Buffer Size). Die einzelnen Kurven spiegeln den Verlauf für die vier verschiedenen Ersetzungsstrategien *Least recently used* (LRU), *Most recently used* (MRU), *Random replacement algorithm* (RANDOM) und *Second Chance* (CLOCK - eine Erweiterung von *First in first out*) wieder (näheres siehe Text). Aus [TTHP05].

finden, welches Zeichen an einer bestimmten Stelle im *String* steht. Die zu vergleichenden Zeichen sind dabei relativ zufällig, so dass viel im *String* umhergesprungen wird. Das *Tree*-Array hat dagegen eine sehr gute Speicherreferenzierung, da einmal geschriebene Teile des Baumes nicht noch einmal angefasst werden müssen. Somit sollte der *String* möglichst im Hauptspeicher gehalten werden, während der *Tree* auch auf der Festplatte gespeichert werden kann.

Zuletzt geht es noch darum, den verfügbaren Hauptspeicher zwischen den vier Datenstrukturen aufzuteilen. Aufgrund des verwendeten Sortieralgorithmus (s.a. 3.1.2), sollten nach *Tian et al* sowohl das *Suffix*- als auch das *Temp*-Array mindestens $|\sum|$ Speicherseiten bekommen, da beim Sortieren $|\sum|$ Positionen gespeichert werden müssen. Der *Tree* sollte mindestens zwei Speicherseiten zugewiesen bekommen. Dies erlaubt das direkte Zugreifen auf einen Verzweigungsknoten, der gerade erst auf den *Stack* gepackt wurde. Der Rest des *Trees* kann ohne Probleme auf der Festplatte gehalten werden, da er eine gute Speicherreferenzierung besitzt. Der *String* mit seiner schlechten 'locality of reference' sollte sich hingegen möglichst komplett im RAM befinden, weshalb restlicher verfügbarer Speicherplatz an den *String*

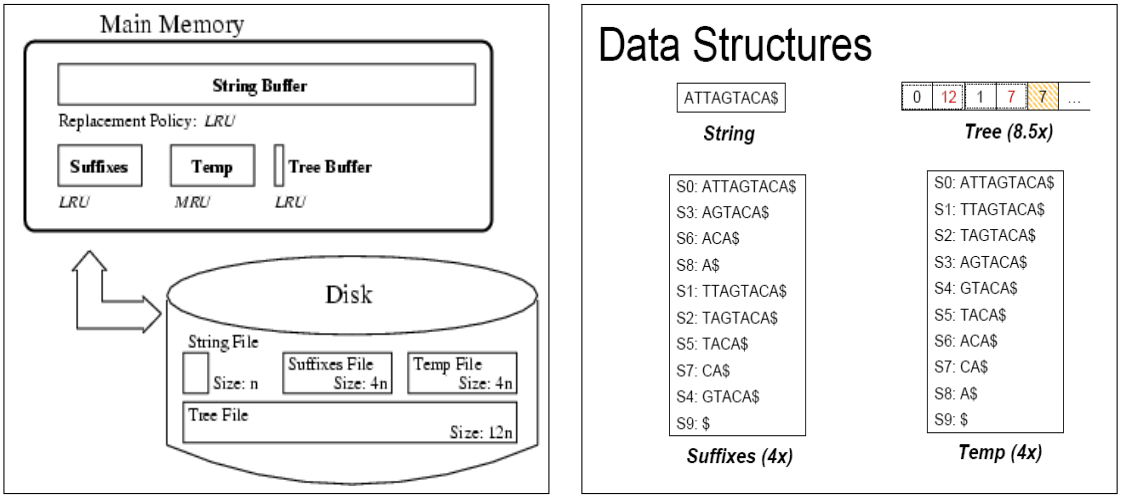


Abbildung 3.7: Darstellung der Pufferungsstrategien im Hauptspeicher (a) und Platzbedarf der *TDD*-Datenstrukturen auf der Festplatte (b). Aus [PT05].

vergeben werden sollte. Wenn danach noch Platz ist, wird dieser nacheinander dem *Suffix*-, dem *Temp*- und zuletzt dem *Tree*-Puffer gewährt. Das Resultat dieser Aufteilung ist in Abb. 3.8 zu sehen. Bei einem kleinen Datensatz (oben) passt alles in den RAM, während bei einem mittleren Datensatz (mitte) hauptsächlich nur der *String* und ein Teil des *Suffix*-Arrays im RAM gehalten wird. Bei großen Datensätzen (unten) hingegen bekommen *Temp*-, *Suffix*- und *Tree*-Puffer lediglich ihren Minimalanteil und der Rest geht an den String.

Tian et al führen in [TTHP05] ihre Pufferstrategie noch viel detailierter aus und gehen auch auf experimentelle Ergebnisse ein. Für *Input-Strings*, die signifikant größer als der verfügbare Hauptspeicher sind, wird eine deutliche Verlangsamung des *TDD*-Algorithmus beschrieben, weshalb zu diesem Zweck ein neuer Algorithmus namens 'ST-Merge' entwickelt wurde. In [TTHP05] wird gezeigt, dass *ST-Merge* in dem beschriebenen Fall viel effizienter ist.

In dieser Arbeit wird aber nicht näher darauf eingegangen, weil es mit dem derzeit verfügbaren Programm-Code nicht möglich war, den Algorithmus zu verwenden. Dieser Umstand wird auch in [PM] beschrieben und ist dort von den *TDD*-Autoren bestätigt worden.

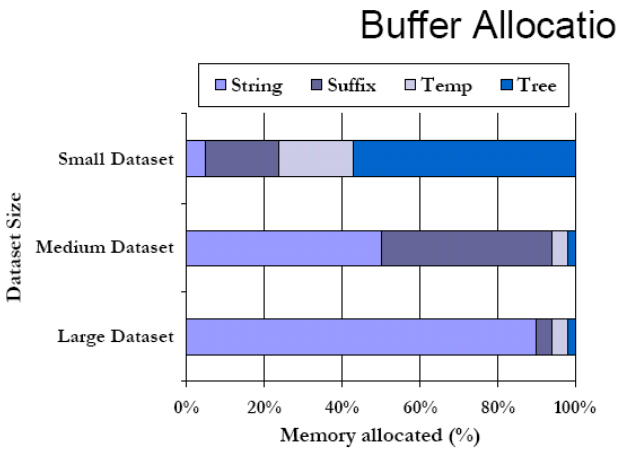


Abbildung 3.8: Pufferalloziierung für die vier Datenstrukturen. Sobald sich die *String*-Größe erhöht, verkleinert sich die Speicherzuweisung für die anderen Strukturen. Aus [PT05].

3.1.4 Zusammenfassung

Auch ohne die Verwendung des im letzten Abschnitt beschriebenen Algorithmus 'ST-Merge' ist *TDD* im Vergleich zum Algorithmus von *Hunt et al* aufgrund seiner Pufferungsstrategie und der Eigenschaft, dass der Baum auf der Festplatte während der Konstruktion nicht durchlaufen wird, deutlich effizienter, was Schnelligkeit und Größe der verarbeitbaren Datensätze angeht.

In [TTHP05] finden sich auch experimentelle Beweise und Grafiken, die belegen, dass *TDD* den Algorithmen von *Ukkonen* und *Hunt* überlegen ist. Es wurde auch beschrieben, dass mit *TDD* als bis dato einzigem Algorithmus das gesamte Humangenom (einzelsträngig, also $3 * 10^9$ Basen) in einem Suffixbaum dargestellt werden konnte. Auch ohne die Funktionalität von *ST-Merge* sollte das auf einem 32-Bit-Rechner noch möglich sein, außerdem wurde in [TTHP05] beschrieben, dass eine Adaption des Programms auf 64-Bit-Architekturen einfach durchführbar sein soll. Dieser Umstand, die Tatsache, dass *TDD* auf den verwendeten Maschinen ohne Probleme kompiliert werden konnte und nicht zuletzt die relativ ausführliche Dokumentation, führten zu der Entscheidung, damit weiterzuarbeiten, um schließlich wie Eingangs erwähnt 'Shustrings' und schließlich den I_r berechnen zu können. Als nächstes erfolgt eine Erklärung dieser zwei Parameter, bevor erläutert wird, wie beide Algorithmen (*TDD* und *IR*) zu dem Programm *TDD_IR* zusammengeführt wurden.

Kapitel 4

Ableitung des Index of Repetitiveness (I_r)

4.1 Hintergrund

Sequenzvergleiche über Alignments spielen eine fundamentale Rolle in der Molekularbiologie bzw. Bioinformatik [HPMW05, NW70, SM81, AGM+90], obwohl sie zu einer Laufzeit tendieren, die dem Produkt der Längen der verglichenen Sequenzen entspricht. Für bestimmte Aufgaben, die mit berechneten Alignments durchgeführt werden können, z.B. dem *Design* von *Oligos* oder der Identifikation von einzigartigen Regionen unter einer Gruppe von nahe verwandten Organismen, kann man sich allerdings auch anderer Methoden bedienen. Dazu zählt u.a. die Quantifizierung und Klassifizierung von sich wiederholenden Elementen ('repeat elements' oder einfach 'Repeats'), was mittlerweile auch eine große Rolle im Bereich der Bioinformatik spielt [HW06, KS99, VHS01]. Hiervon soll der *Index of repetitiveness* (I_r) aus [HW06] vorgestellt werden, als ein Werkzeug mit linearem Zeitverhalten zur Messung repetitiver Elemente innerhalb von Sequenzen. Dieses Verfahren beruht auf der Verwendung von *shortest unique substrings*, also der kürzesten Untersequenzen, die sonst nirgendwo in der analysierten Sequenz gefunden werden, weshalb zunächst noch auf die Herleitung dieser auch so genannten *Shustrings* eingegangen wird.

4.2 Shortest Unique Substrings

4.2.1 Definition und Bestimmung mittels Suffixbäumen

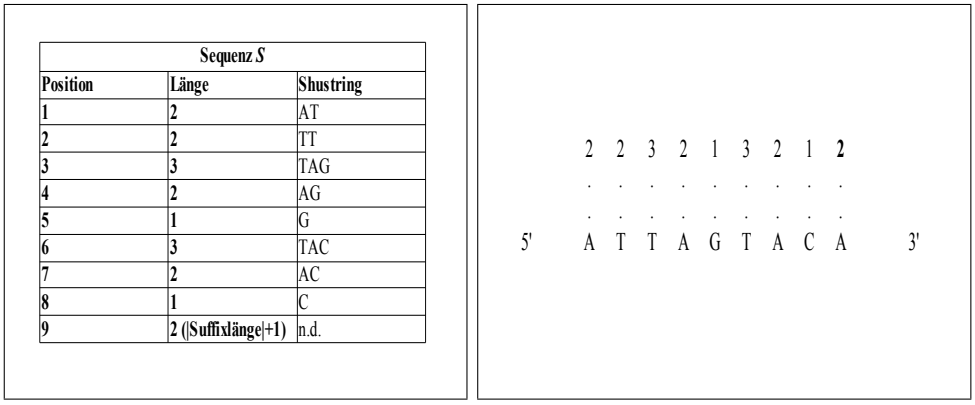
Shortest unique substrings kann man auch als das Komplement von Repeats ansehen. Die Untersuchung dieser *uniquen*/einzigartigen Sequenzen ist also der Startpunkt für die Quantifizierung der Repetitivität in Bezug auf die Genomzusammensetzung, wie man später sehen wird. *Shortest unique substrings* sind einfach zu finden, z.B. ist eine Sequenz immer *unique* in Bezug auf sich selber und sie können nicht in Ihrer Länge reduziert werden, ohne ihre Einzigartigkeit zu verlieren. D.h. mehr formal gesprochen:

Definition 5: Für jede Position i in S gilt, dass es eine Länge x_i für einen *shortest unique substring* gibt, mit der der Substring $S[i..i + x_i - 1]$ in S einzigartig ('unique') ist, während es $S[i..i + x_i - 2]$ nicht mehr ist [HW06].

Man kann somit eine Sequenz oder gar ein ganzes Genom durchlaufen und an jeder Position i die Länge der *shortest unique substrings* bestimmen, die somit an i auch ihren Anfang nehmen. Man beachte, dass im Gegensatz zum letzten Kapitel 3, hier bzw. in [HPMW05, HW06] der Index i wieder mit 1 beginnt, wie es z.B. auch in [Gus97] als Notation verwendet wird.

Am Beispiel der bekannten Sequenz *S=ATTAGTACA$* soll dies anhand von Abbildung 4.1 (a) und (b) veranschaulicht werden. Dieser String hat $\binom{10}{2} = 45$ mögliche Untersequenzen (siehe 2.1.1.1). Zwei davon, nämlich {G, C} haben eine Länge von *Eins* und sind somit *globale shortest unique substrings*. Diese können überall in der Sequenz S auftauchen, hier an den Positionen *fünf* und *acht*. *Lokale shortest unique substrings* sind dagegen an eine spezifische Position in S gebunden (siehe Definition 5). Sie haben hier Längen von *1-3* und lauten der Reihe nach {AT, TT, TAG AG, G, TAC, AC, C}. An Position *9* beginnt kein einzigartiger *Substring* mehr, weil das Nukleotid bzw. der Buchstabe 'A' schon mal vorher in der Sequenz enthalten war. In diesem Fall wird die Länge des *shortest unique substrings* als die Summe aus der entsprechenden Suffixlänge mit *eins* addiert (hier also $1 + 1 = 2$) festgelegt, oder anders ausgedrückt:

Definition 6: Gibt es an einer Position i in einem String S der Länge n keinen einzigartigen Substring, wird als *Shustringlänge* $|S[i..n]| + 1$ verwendet.



Sequenz *S*		
Position	**Länge**	**Shustring**
1	**2**	AT
2	**2**	TT
3	**3**	TAG
4	**2**	AG
5	**1**	G
6	**3**	TAC
7	**2**	AC
8	**1**	C
9	**2 (|Suffixlänge|+1)**	n.d.

(a) (b)

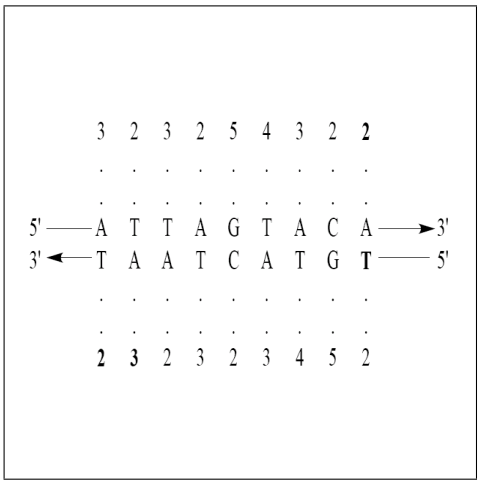
Abbildung 4.1: Länge der *shortest unique substrings* für die Sequenz *ATTAGTACA$* (angelehnt an die Ausgabe des Tools *shustring*, von Haubold und Wiehe, das unter *http://adenine.biz.fh-weihenstephan.de/shustring* erhältlich ist) (a). Darstellung innerhalb der DNA Sequenz. Beginnt man z.B. am ersten Nukleotid, sind in die 3′-Richtung zwei Schritte nötig, um einen *unique substring* zu finden. Über der letzten Position ('A') steht eine fett gedruckte Zahl, die aussagt, dass hier die Suffixlänge mit Eins addiert verwendet wurde (näheres, siehe Text) (Grafik angelehnt an [HW06]) (b).

Aufgrund des doppelsträngigen Aufbaus der DNA bezieht man zur Berechnung der *shortest unique substrings* auch den Rückwärtsstrang mit ein [HPMW05]. Aus Abbildung 4.2 (a) und (b) sind diesesmal für beide Stränge wieder die zugehörigen *Shustrings* und ihre Längen ersichtlich. Man muss natürlich berücksichtigen, dass die Leserichtung und somit die Positionen immer am 5′-Ende beginnen und am 3′-Ende aufhören, wie es auch in der Grafik beschrieben wird. Wie man sieht, unterscheiden sich der Satz an *shortest unique substrings* von doppel- und einzelsträngiger DNA, z.B. gibt es zusätzlich zur neunten Position im Vorwärtsstrang auch an den zwei letzten Stellen im Rückwärtsstrang keine kürzesten *Substrings* mehr, da der Suffix 'AT' bereits im Vorwärtsstrang und 'T' in beiden Strängen enthalten ist. Weil DNA einen komplementären Aufbau hat, ist das Komplement eines *lokalen Shustrings* wieder ein einzigartiger Substring, wenn auch nicht mehr notwendigerweise der Kürzeste (s. Abb. 4.2 (b)). Dagegen ist das Komplement eines *globalen Shustrings* wieder ein kürzester, einzigartiger Substring [HPMW05].

Um sich *shortest unique substrings* als Komplement zu repetitiven Elementen überhaupt biologisch zunutze machen zu können, benötigt man ein Verfahren, mit dem man sie wirkungsvoll auffinden kann, sowie eine Definition ihrer Wahrscheinlichkeitsverteilung. Um die *Shustrings* zu bestimmen,

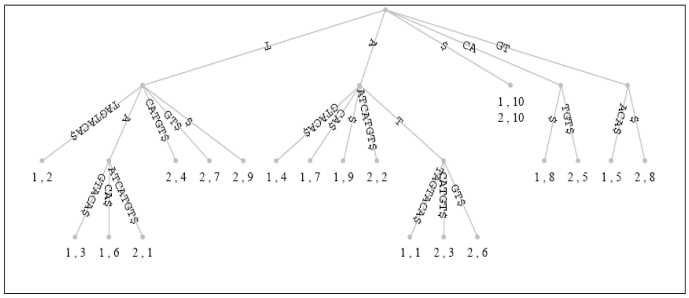
Sequenz *S*		
Position	**Länge**	**Shustring**
1	**3**	AT
2	**2**	TT
3	**3**	TAG
4	**2**	AG
5	**5**	G
6	**4**	TAC
7	**3**	AC
8	**2**	C
9	**2 (|Suffixlänge|+1)**	n.d.
	Sequenz *S revers*	
Position	**Länge**	**Shustring**
1	**2**	TG
2	**5**	TT
3	**4**	TAG
4	**3**	AG
5	**2**	G
6	**3**	TAC
7	**2**	AC
8	**3 (|Suffixlänge|+1)**	n.d.
9	**2 (|Suffixlänge|+1)**	n.d.

(a)



(b)

	1	2	3	4	5	6	7	8	9	10
T_1	A	T	T	A	G	T	A	C	A	$
T_2	T	A	A	T	C	A	T	G	T	$



(c)

Abbildung 4.2: Das Ergebnis für die *Shustringlängen*, unter Einbindung der Rückwärtsstranges *TGTACTAAT* (a). Darstellung des Doppelstranges als Sequenz. Man beachte, dass immer vom 5' nach 3' gelesen wird (b). Suffixbaum für Vorwärts- und Rückwärtsstrang (in der Tabelle T_1 bzw. T_2 genannt und jeweils von $5'$ nach $3'$ gelesen). Im Gegensatz zu den bisherigen Suffixbaumdarstellungen enthalten die Blattbeschriftungen jetzt ein oder mehrere Ziffernpaare. Die erste Ziffer dient der *Stringidentifikation*, die zweite gibt wie bisher die Startposition des Suffix wieder (vgl. Kapitel 2.1.2 und Abb. 2.2) (c).

bedient man sich der in Kapitel 2 erläuterten *Suffixbäume*. Diese eignen sich so gut zum Auffinden der *Shustrings*, weil ja ein an der Wurzel beginnender und an einem Ast endender Teilstring sooft vorkommt, wie es darunter liegende Blätter gibt (s. 2.1.2 und 2.1.3). D.h. wenn eine solche Teilsequenz in einem internen Ast endet, kommt sie mindestens zweimal im Gesamtstring vor. Der *Substring* 'GT' (in Abb. 4.2 (c) ganz rechts außen) kommt z.B. zweimal vor, einmal im Vorwärtsstrang ab Position *5* und einmal im Rückwärtsstrang ab Position *8*. Dagegen beginnt 'GTA' ebenfalls ab Position *5*, endet aber in einem externen Zweig und wird somit nicht nochmal wiederholt. Für einen *shortest unique substrings* bezogen auf die Position i, benötigt man also nur die Länge der Kantenbeschriftung von der Wurzel bis zu dem Elternknoten des entsprechenden Blattes. Diese Länge wird auch als die *Stringtiefe* s ('string depth') des Knotens bezeichnet [HW06]. Addiert man *1* zu s hinzu, erhält man die gewünschte Länge des *Shustrings*, der an i beginnt, weil man sich dann immer in einem externen Zweig befindet. Daraus ergibt sich folgende allgemeine Vorgehensweise zum Auffinden von *Shustrings*:

An einer Position i einer Sequenz S erhält man die Länge x_i eines *shortest unique substrings*, indem man in einem Suffixbaum bis zu den Blättern läuft und anschließend die Stringtiefe s des Elternknotens des durch den Index i markierten Blattes mit eins addiert, d.h. $x_i = s + 1$. Durch die Verwendung des Terminierungssymbols \$ gilt dies auch für Positionen, an denen kein *shortest unique substring* wohldefiniert ist.

4.2.2 Verteilung in Genomen und Zufallssequenzen

Mit dem Auffinden von *shortest unique substrings* mittels Suffixbäumen hat man nun eine Methode, mit der man in linearer Zeit vergleichende Genomik betreiben kann[1]. In vielen solcher Projekte ist das Aufspüren von einzigartigen Genomregionen ein erster Schritt zur funktionellen Annotation [HPMW05]. Vorher muss man aber wissen, wie die Verteilung von *shortest unique substrings* in Zufallssequenzen aussieht, bzw. wie die *Nullhypothese* lautet, damit man eine Aussage treffen kann, wie signifikant die Wahrscheinlichkeit ihres Auftretens ist. Ist dies möglich, besteht zumindest für diese Fragestellungen keine weitere Notwendigkeit von zeitaufwändigen Alingments mehr.

Haubold und *Wiehe* verwendeten in [HW06] einen 2 kb (Kilobasen) langen

[1]Ein Auffinden der *shortest unique substrings* in linearer Zeit setzt natürlich einen ebenfalls O(n)-Baumkonstruktionsalgorithmus voraus.

Ausschnitt aus dem Genom des Krankheitserregers *Mycoplasma genitalium* und berechneten an jeder Position in diesem Bereich die Länge x_i der *shortest unique substrings*, wie in Abbildung 4.3 (a). Daneben durchmischten sie diese Sequenz auch noch und erhielten dann das Ergebnis wie in Abbildung 4.3 (b).

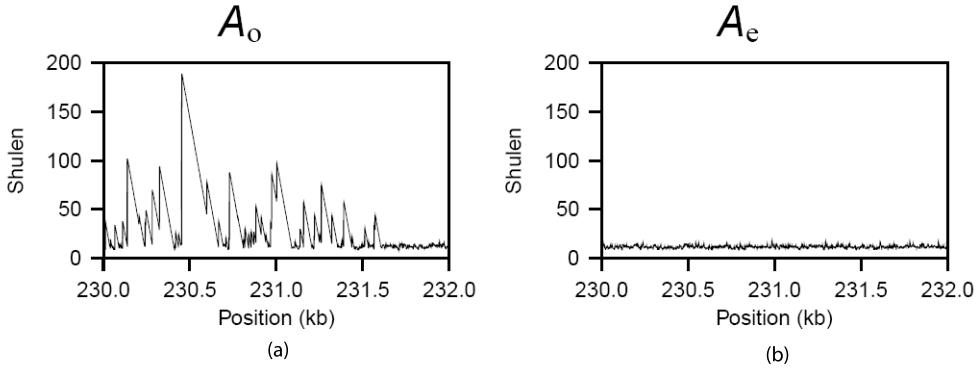


Abbildung 4.3: Die Längen der *shortest unique substrings* entlang von 2 kb Länge des Genoms von *Mycoplasma genitalium*. (a): Originalsequenz; (b): Durchmischte Sequenz. Modifiziert übernommen aus [HW06].

Es fällt hier gleich auf, dass die Originalsequenz deutliche Zacken enthält, die sich mit ungewöhnlich langen *shortest unique substrings* decken. Diese werden durch sich wiederholende Teilsequenzen ('Repeats') verursacht, die länger sind, als die zufällig erwarteten [HW06]. Die beobachtete aggregierte Länge A_0 von *shortest unique substrings* in einer doppelsträngigen Sequenz S mit Gesamtlänge $2n$ wird folgendermaßen definiert:

Definition 7 (aus [HW06]):

$$A_0 = \sum_{i=1}^{2n} x_i.$$

Die Menge A_0 entspricht dabei der Fläche unter der Kurve der miteinander verbundenen *Shustringlängen*, wie z.B. in Abb. 4.3 (a) gezeigt.

In Abbildung 4.3 (b) sind dagegen wie gesagt die Längen der *shortest unique substrings* des durchmischten *Mycoplasma*-Abschnitts zu sehen. Die Zacken, die in der Originalsequenz auf lange repetitive Elemente hinwiesen, sind hier verschwunden und es bleibt nur eine in ihrer Ausdehnung begrenzte Basislinie. Die Fläche unter dieser auch sog. 'baseline' entspricht der Menge A_e, also der erwarteten aggregierten Länge von *shortest unique substrings* und ist wie folgt definiert:

Definition 8 (aus [HW06]):

$$A_e = \sum_x x N_x.$$

Dieser Ausdruck wurde in [HPMW05] ausführlich hergeleitet. Der Term $x * N_x$ gibt dabei die Anzahl der erwarteten *shortest unique substrings* der Länge x wieder, die in einem komplett durchmischten Genom gegebener Länge vorhanden sind. Über dem Term N_x fließt dabei auch der G/C-Gehalt mit ein, wie in [HW06] beschrieben.

Das Verhältnis der beiden Größen A_o und A_e wird im Anschluss benutzt, um den *Index of Repetitiveness* auszurechnen. In [HPMW05] finden sich noch andere nützliche Anwendungen von *shortest unique substrings* wie z.B. die Identifikation von Oligonukleotidsequenzen und außerdem die Größenverteilungen der globalen als auch lokalen *shortest unique substrings* in eukaryoten und prokaryoten Genomen.

4.3 Index of Repetitiveness

4.3.1 Definition und Nullverteilung

Der *Index of Repetitiveness* basiert also auf *shortest unique substrings*. Genauer gesagt definiert er sich als das Verhältnis aus beobachteten und theoretisch erwarteten Längen von *shortest unique substrings*. Wieder formal ausgedrückt:

Definition 9 (aus [HW06]):

$$I_r = log\left(\frac{A_o}{A_e}\right)$$

Für Genome, die keine übermäßige Anzahl an sich wiederholenden Sequenzen haben, ist das Verhältnis $\left(\frac{A_o}{A_e}\right)$ nahezu eins, womit der Logarithmus (bzw. der I_r) daraus ungefähr gleich Null ist, d.h. $I_r \approx 0$. Sequenzen mit einem Überschuß an *Repeats* haben dagegen einen I_r-Wert größer Null ($I_r > 0$).

Anhand des Genoms der Bakteriophage λ wurde dies im Experiment bestätigt. Das Genom wurde wiederholt durchmischt bzw. 'randomisiert' und jeweils der *Index of Repetitiveness* berechnet. In Abbildung 4.4 sieht man das daraus resultierende Histogramm, dass eine symmetrische Verteilung nahe dem erwarteten Mittelwert von Null (Mittelwert: $\mu = 0,0004$, Standardabweichung: $\sigma = 0,0008$) hat.

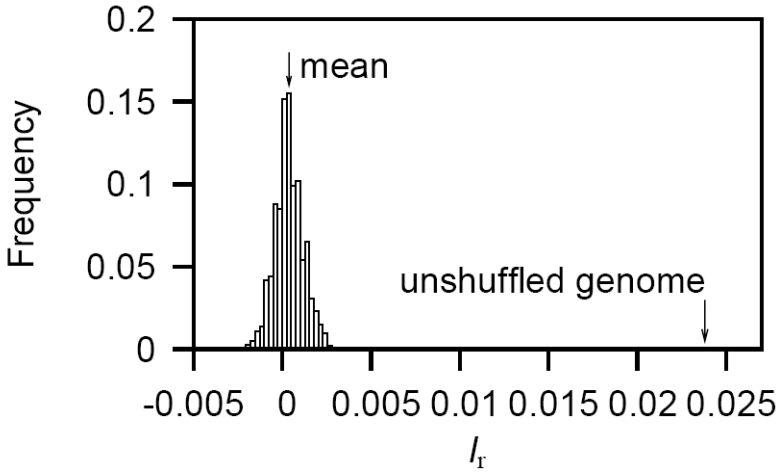


Abbildung 4.4: Die Nullverteilung des I_r anhand des Genoms des Bakteriophagen λ, das 1000 Mal durchmischt und jeweils der I_r berechnet wurde. Aus [HW06].

4.3.2 Anwendung

Der *Index of Repetitivenss* wurde in weiteren Experimenten auf 336 Genome aus allen drei Domänen des Lebens angewandt. Dabei wurde herausgefunden, dass der I_r von Archaen signifikant kleiner ist als von Bakterien und dieser wiederum geringer als der von Eukaryoten ist. Mauschromosomen haben einen signifikant höheren I_r als menschliche Chromosomen und innerhalb dieser beiden Genome hat das Y-Chromosom die höchste Repetitivität [HW06]. In dieser Arbeit wurde der Hauptaugenmerk auf das Humangenom gelegt, weshalb in Abb. 4.5 nur dieses veranschaulicht wird. Darstellungen aus anderen Genomen findet man in [HW06], zudem eine ausführliche Beschreibung der Resultate.

Um innerhalb eines Genoms Bereiche mit hohen, niedrigen oder gar negativen I_r-Werten näher zu untersuchen, wurde noch eine gleitende Fensteranalyse ('sliding window analyses') eingeführt, für die jeweils 'lokal' die Summen aus erwarteten und beobachteten *Shustring*-Längen gebildet und zum I_r umgerechnet wurden. Konkret wurde dies so umgesetzt, dass beginnend an der ersten Position der zu analysierenden Sequenz ein Fenster bzw. Intervall von 1000 bp Länge festgelegt wurde. A_o ist dann die Summe der Längen der *shortest unique substrings*, die innerhalb dieses Bereichs beginnen. Ähnlich dazu wird die Berechnung von A_e durchgeführt. Hierzu werden jetzt nur der lokale G/C-Gehalt berücksichtigt und eben die Fensterlänge von hier 1000. Für eine

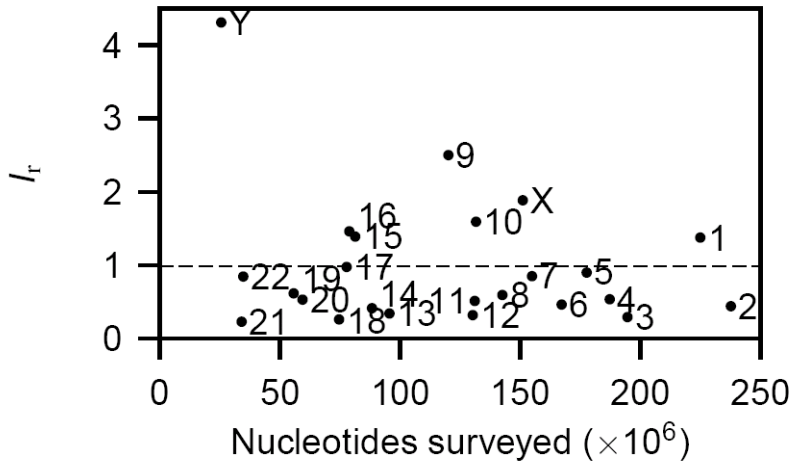


Abbildung 4.5: Die globalen I_r-Werte für die Chromosomen des Humangenoms als Funktion in Abhängigkeit von der jeweiligen Anzahl der Nukleotide. Die gestrichelte Linie gibt den Durchschnittswert von 0,985 wieder. Vgl. Abb. aus [HW06].

grafische Darstellung wird der resultierende I_r an der Position eingetragen, die dem Mittelpunkt des Fensters entspricht (hier: $Startposition + 500,5$). Das Fenster wird dann um ein Zehntel seiner Länge weiterverschoben, also in diesem Fall um 100 bp und der I_r wieder analog zu vorhin berechnet. Ein Beispiel hierfür sieht man an der 'sliding window'-Analyse der *HOX*-Gene auf dem menschlichen Chromosom 7 (Abb. 4.6). Auf eine Interpretation dieser Analyse wird in Kapitel 6 im Rahmen der Auswertung eingegangen.

Anzumerken ist noch, dass unsequenzierte Regionen, für die innerhalb eines Genoms der Buchstabe 'N' steht, entfernt wurden, um eine 'künstliche' Erhöhung sowohl des globalen I_r als auch der Repetitivität bei der Fensteranalyse zu vermeiden.

Ohne homologe Beziehungen zu berücksichtigen, hat man nun eine Maßeinheit zur Messung der Repetitivität von doppelsträngigen DNA-Sequenzen, wobei der Erwartungswert immer gleich Null ist. Weil die Konstruktion des zugrunde liegenden Suffixbaumes theoretisch nur proportional viel Zeit zur Länge der analysierten Sequenz benötigt (s. Kap. 2.2.2), kann der *Index of Repetitivenesss* auch mit zur Länge der Eingabesequenz linearer Zeitkomplexität berechnet werden. Dagegen benötigt die traditionelle Analyse von repetitiven Elementen, wie z.B. im Programm *repeatmasker* [Rep] implementiert, eine Laufzeit proportional zur Länge des Produktes der zu vergleichenden

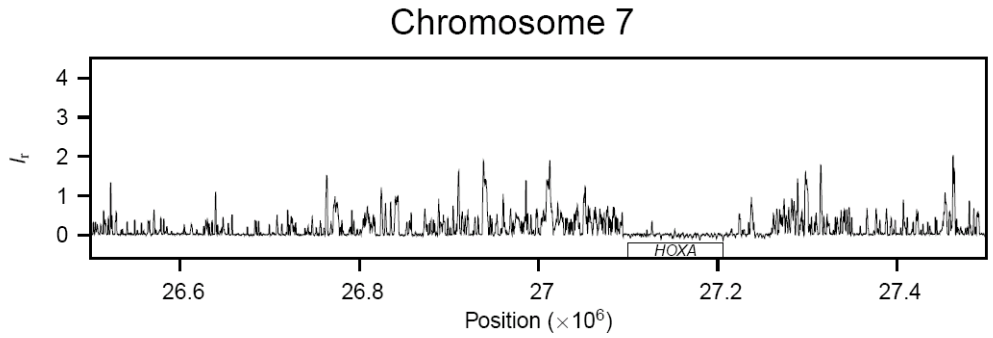


Abbildung 4.6: Eine gleitende Fensteranalyse im Bereich der HOX-Gene auf Chromosom 7. Aus [HW06].

Sequenzen. [HW06].

Wie bei den meisten Suffixbaumimplementierungen, wurde auch der Suffixbaum, auf dem die damalige I_r-Analyse basierte, komplett im Hauptspeicher des Computers konstruiert. Das hatte den Vorteil, dass der Baum relativ einfach implementiert werden konnte, aber den Nachteil, dass diese Methode nur eine durch den verfügbaren *RAM* begrenzte Menge an Sequenzdaten analysieren konnte. Eine Idee, dieses Hindernis zu umgehen, fundierte in der Verwendung von Festplatten basierenden, persistenten Suffixbäumen, wie sie in Kapitel 3 beschrieben wurden. Es musste deshalb herausgefunden werden, wie Suffixbäume aus einer solchen Datenstruktur wieder ausgelesen bzw. *shortest unique substrings* bestimmt werden konnten, bevor man damit die weiteren Methoden zur Bestimmung des I_r verwenden kann.

Kapitel 5

Die Fusion von *TDD* und I_r zu TDD_IR

Im Kapitel 4.3 wurde der *Index of Repetitiveness* vorgestellt, der in [HW06] als ein neues Maß zur Messung der Repetitivität von Genomen eingeführt wurde. Zu dessen Berechnung werden *shortest unique substrings* verwendet, die sich wiederum effektiv über Suffixbäume bestimmen lassen. Da die weit verbreiteten $O(n)$-Suffixbaumkonstruktionsalgorithmen auf die Größe des verfügbaren Hauptspeichers beschränkt sind, bieten sich als Alternative Algorithmen an, die die Suffixbäume persistent auf Festplatte speichern und somit auf viel größere Platzressourcen zurückgreifen können. Am effektivsten erschien hier der *Top-Down Disk-Based*-Algorithmus, der in [TTHP05] und im letzten Kapitel vorgestellt wurde. Im Anschluß wird darauf basierend gezeigt, wie *shortest unique substrings* aus einem so erzeugten Suffixbaum gelesen werden können. Es folgt eine Darstellung, welche Klassen und Methoden aus den Implementierungen zu *TDD* und I_r man dazu und zur Berechnung des *Index of Repetitiveness* benötigt. Schließlich wird auf das neue Programm *TDD_IR* eingegangen und welche Grenzen und Möglichkeiten damit erreicht wurden.

5.1 Algorithmus zum Auslesen von *Shustrings* aus *TDD*-Bäumen

Um *shortest unique substrings* aus der über den *Top-Down Disk-Based*-Algorithmus gewonnen Suffixbaumstruktur auslesen zu können, visualisiert man diese am Besten nochmal anhand der altbekannten Beispielsequenz *S=ATTAGTACA$*. Der Einfachheit halber wird an dieser Stelle nochmal dasselbe Suffixbaumkonstrukt wie im vorletzten Kapitel abgebildet (Abb.

5.1.

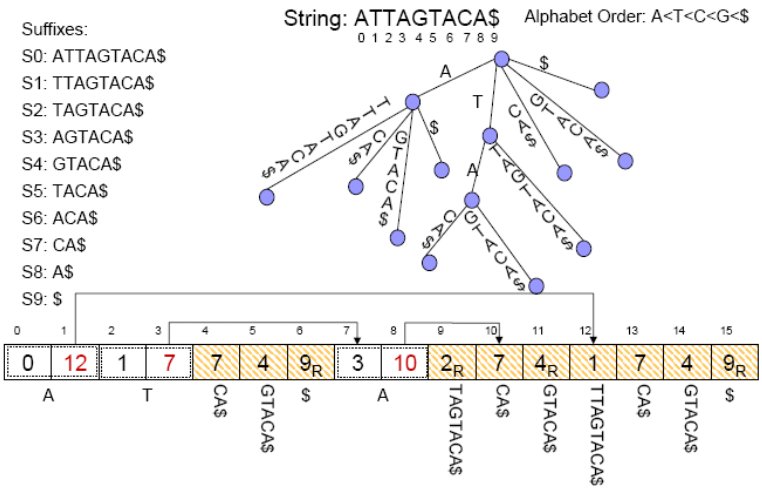


Abbildung 5.1: Der komplette Sufixbaum für den String *ATTAGTACA$* mit der zugehörigen Array-Darstellung. Aus [PT05].

Entsprechend der Gewinnung der *shortest unique substrings* über Suffixbäume (s. 4.2.1), benötigt man die Stringtiefe der Elternknoten zu den jeweiligen Blättern. Man beachte, dass hier im Gegensatz zu den Kapiteln 2 und 4 wieder die von den *TDD*-Autoren favorisierte Notation verwendet wird, bei der die Stringelemente, Suffixe und Blätter mit '0' beginnend durchnummeriert werden. Zur Erinnerung sei nochmal gesagt, dass in dem, den Suffixbaum darstellenden Array, die orange schattierten Kästchen für ein Blatt und die weißen Kästchen für einen Verzweigungs- oder internen Knoten stehen (s. Kapitel 3.1.2). Blätter haben nur einen Eintrag im Array, Knoten dagegen zwei Einträge, die durch eine gestrichelte Linie getrennt sind.

Das Array im Beispiel beginnt also mit einem Verzweigungsknoten als erstem Eintrag. D.h. es handelt sich im Array an Index *0* um einen Elternknoten, der im String ebenfalls an Position *0* beginnt. Der zweite Eintrag eines Knotens zeigt ja immer auf das erste Kind und hier somit auf die Arrayposition *12*. Hier findet man ein Blatt, dessen Suffix an Position *1* im String beginnt. Man geht nun nach rechts weiter, bis man auf ein als 'rightmost' mit dem Buchstaben 'R' markiertes Blatt bzw. Kind trifft. In diesem Beispiel hat man vier Kinder, die gleichzeitig Blätter sind und deren Suffixe an den Positionen *1*, *7*, *4* und *9* beginnen. Dadurch kennt man die Länge der Elternkantenbeschriftung und somit die Stringtiefe des Elternknotens, nämlich, indem man das Minimum aus den Suffixanfängen der Blätter sucht und davon die Startposition des Elternknotens abzieht, d.h. $min\{1, 7, 4, 9\} - 0 = 1$.

Die Suffixstartpositionen im String erhält man, indem man ebenfalls wieder eine Differenz bildet und zwar von den jeweiligen Einträgen im Array und der Länge der Elternkantenbeschriftung, also *1-1=0*, *7-1=6*, *4-1=3* und *9-1=8*. Die *shortest unique substrings* an diesen Positionen (1,6,3,8) entsprechen der Länge der Elternkantenbeschriftung addiert mit eins, also *1+1=2*.

Nachdem man die *Shustrings* für diese Positionen ermittelt hat, springt man im Array wieder zurück und geht einen Schritt weiter, in diesem Fall auf die Position *drei*.

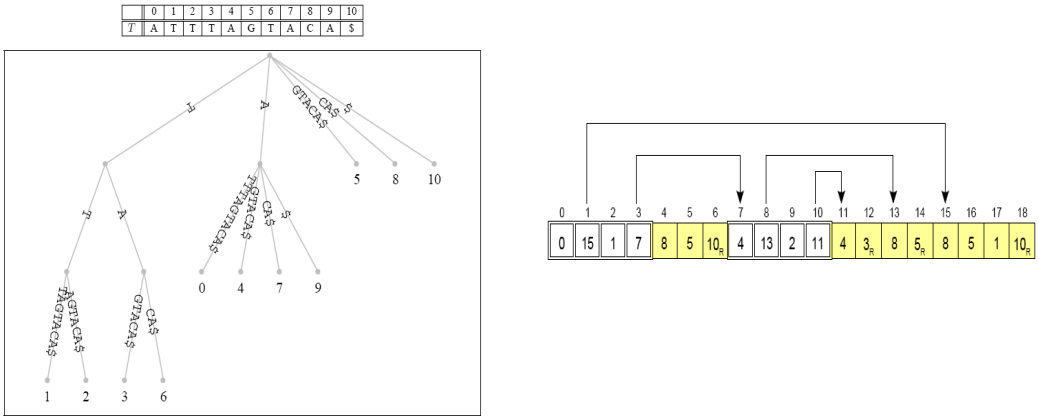
Wieder handelt es sich nicht um ein Blatt, sondern einem weiteren internen Knoten. Man benötigt also die Arrayindizes *zwei* und *drei*. Der Eintrag am Index *zwei* besagt, dass der Elternsuffix an Position *1* im String beginnt und am Index *3* wird auf das erste Kind an Arrayposition *sieben* verwiesen. Hier findet sich allerdings wieder kein Blatt, sondern ein weiterer Elternknoten, dessen Suffix an Stringposition *3* beginnt und der auf sein erstes Kind an Arrayposition *zehn* verweist.

Zunächst merkt man sich aber bereits hier die Länge der Kantenbeschriftung des ersten Elternknotens, indem man alle weiteren direkten Kindknoten im Array durchläuft. Dazu gibt es zwei Möglichkeiten. Entweder ein Kind des ersten Elternknotens ist bereits ein Blatt, dann ist dieses mit 'R' markiert und man läuft bis dahin. Man sieht dies wiederum in Abb. 3.5. Hier muss man sich eigentlich nur die Startpositionen des ersten Elternknotens merken, die von dem verzweigten Knoten und schließlich die von dem 'rechtesten' Blatt. Hieraus bildet man wieder die Differenz wie oben und hat für den ersten Elternknoten eine Kantenlänge von $min\{3,2\}-1=1$. Mit dem zweiten Elternknoten geht es dann erstmal weiter. Der zugehörige zweite Eintrag an Arrayposition *acht* verweist ja auf Position *zehn*. Hier findet man nun zwei hintereinanderfolgende Blätter. Das zweite ist wieder 'rightmost', also folgen auch keine weiteren Kinder mehr. Die beiden Einträge an diesen Blättern sind *7* und *4*. Aus deren Minimum bildet man wieder die Differenz mit ihrem Elternknoten und erhält für dessen 'Label'-Länge den Wert $min\{7,4\}-3=1$. Man beachte, dass hier immer der Eintrag des Suffixbeginns für den direkten Elternknoten verwendet wird. Man weiß nun, dass dieser Elternknoten als Kantenlänge den Wert *eins* hat und sein Vorfahre ebenso. Zusammen ergibt dies nun den Wert *1+1=2* für die *Stringtiefe* des zweiten Elternknotens. D.h. die Länge der *shortest unique substrings* für die darunter liegenden Blätter ist wieder durch die Addition von *1* erhältlich, also *2+1=3*.

Nun springt man im Array wieder zurück auf die Position *acht* und von dort auf die Ausgangsposition *3* und geht einen Schritt weiter. An den drei nachfolgenden Kästchen, Arraypositionen *vier*, *fünf* und *sechs*, findet man jeweils ein Blatt, was bedeutet, dass der einzige Elternknoten die Wurzel ist. In diesem Fall sind die *shortest unique substrings* gleich *1* und gelten gleich

für die im Array eingetragenen Indizes, d.h. man hat am Index *7*, *4* und *9* einen *shortest unique substring* der Länge *1*.

Was aber ist, wenn es kein Blatt, sondern nur weitere Elternknoten der nächsten Generation gibt? Dies soll kurz an dem erweiterten Beispielstring $ATTTAGTACA\$$ demonstriert werden. Hier wurde an der zweiten Stelle ein 'T' eingefügt und der Rest nach rechts verschoben. 5.2 (a) zeigt den zugehörigen Suffixbaum und man sieht, dass im 'T'-Zweig erst nach der zweiten Elterngeneration Blätter erscheinen.



(a) (b)

Abbildung 5.2: Suffixbaum (a) und Suffixbaumarray (b) für die Sequenz $ATTTAGTACA\$$

Man betrachte im Array 5.2 (b) gleich den internen Knoten an Position *vier*, der Eintrag hier ist gleich *1*. Diesen merkt man sich genauso wie oben als Suffixbeginn eines ersten Elternknotens. Der nächste Arrayeintrag verweist auf Position *7*. Hier befindet sich wieder ein Verzweigungsknoten. Auch hier merkt man sich seinen Eintrag (4) und seinen Verweis auf weitere Kinder (13). In der Praxis wird dieser Knoten dann auf einen Stapel ('Stack') gelegt, ebenso seine Kinder, also die beiden Blätter, die an den durch den Verweis festgelegten Arraypositionen lokalisiert sind (Abb. 5.3 (a)).

Man geht dann im Array wieder an die Stelle des Verweises zurück, und geht einen Schritt weiter. Man befindet sich nun an Arrayposition *neun*. Dieser Index wird jetzt mit dem Wert des Verweises des vorherigen Knotens verglichen und muss kleiner als diesem sein, damit man keinen Knoten oder kein Blatt betrachtet, dass gar nicht in diesen Zweig gehört. in diesem Fall hat man $9 < 13$, kann also weiterarbeiten. Man befindet sich weiterhin an Arrayposition *neun* und der Suffixbeginn dieses Knotens wird sich wieder

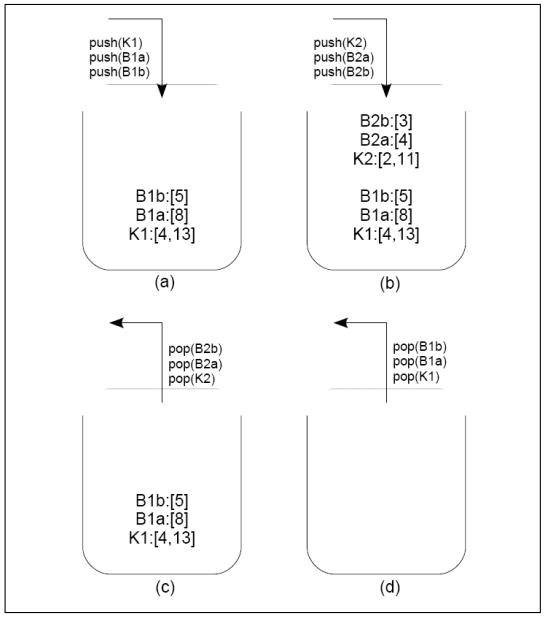


Abbildung 5.3: Der Knoten K1 wird zuerst mit seinen Blättern B1a und B1b auf den Stack gelegt. Es folgt Knoten K2 mit B2a und B2b. Das herunternehmen vom Stapel geschieht in umgekehrter Reihenfolge.

gemerkt (2). Auch dieser Knoten wird mitsamt seinen Kindern an Index *11* und *12* im Array auf den *Stack* gelegt (Abb. 5.3 (b)). Der Verweis zu den Kindern befindet sich im Array an Stelle *10* und hat den Wert *11*. Der Wert dieses Verweises wird mit dem alten (des anderen internen Knotens der gleichen Generation) verglichen und nachdem er kleiner ist, als neue Referenz übernommen. Geht man nun einen Schritt im Array weiter, befindet man sich schon auf Position *11* und nachdem dieser Wert nicht mehr kleiner ist als der niedrigste Verweis, gibt es keine weiteren Kinder mehr.

D.h. man kann nun über die Suffixanfänge der Elternknoten der ersten Generation auf die Länge der Kantenbeschriftung des allerersten Elternknotens schließen. Über $min\{4,2\}-1$ kommt man auf den Wert *1*, den man sich merken muss. Als nächstes werden sukzessive die Knoten bzw. Blätter vom Stapel geholt. Als erstes die beiden Blätter, die an Arrayposition *elf* und *zwölf* eingetragen sind und der Knoten, der durch Index *neun* und *zehn* repräsentiert wird (Abb. 5.3 (c)). Aus den Einträgen für die Suffixanfänge läßt sich für den Knoten eine Kantenlabellänge von $min\{4,3\}-2=1$ berechnen. Zusammen mit der Länge *1* für die Kante des ältesten Vorfahren, ergibt sich als Stringtiefe für diesen Knoten *1+1=2* und eine Shustringlänge von *2+1=3*. Diese sind wieder den Stringpositionen zugeordnet, die sich aus den

Arrayeinträgen und der Stringtiefe ergeben also *4-2=2* und *3-2=1*. Der noch auf dem Stapel befindliche Knoten und dessen Blätter werden analog abgehandelt. Zum Schluß hat man wieder einen leeren Stack, der bei Bedarf für einen anderen Zweig wiederverwendet werden kann.

Zusammenfassend soll nun der Algorithmus allgemein formuliert werden.

Die *shortest unique substrings* für einen mit dem *Top-Down Disk-Based*-Algorithmus erstellten Suffixbaum erhält man über folgenden Auslesevorgang des zugehörigen Suffixbaumarrays:

- Man arbeite die Knoten/Blätter im Array nacheinander ab.
- Trifft man auf einen Verzweigungsknoten, folge man dem Verweis zu seinen Kindern und merke seinen Starteintrag a_1 im Array.
- Man nehme die Starteinträge $a_{11}..a_{1n}$ aller Kinder und berechne die Kantenlänge l_1 des Elternknotens mit der Formel $l_1 = min\{a_{11}..a_{1n}\} - a_1$ und merke sich diese Länge.
- Handelt es sich bei den Kindern wieder um Verzweigungsknoten, gehe man genauso vor, um deren Längen der eingehenden Kanten zu erhalten, z.B. $l_2 = min\{a_{111}..a_{11n}\} - a_{11}$.
- Am Ende eines Astes erreicht man ein oder mehrere Blätter, davon mindestens ein äußerstes rechtes. Für die Stringtiefe des direkten Vorfahren summiere man alle bis zu diesem Knoten berechneten Längen der jeweils eingehenden Kanten: $\mathit{Stringtiefe} = \sum_{k=1}^{n} l_k$.
- Die Shustringlängen erhält man durch die Addition dieser Summe mit 1 ($\mathit{Stringtiefe} + 1$). Die zugehörigen Positionen im String berechnen sich aus der Differenz der Arrayeinträge der Blätter und ebenfalls der Stringtiefe ihrer direkten Vorfahren ($\mathit{Arrayeintrag} - \mathit{Stringtiefe}$).
- Ist ein Zweig abgearbeitet, springe man zurück zum Ausgangspunkt und handele den nächsten Knoten/das nächste Blatt ab.

5.2 Implementierung des Ausleseprozesses

Dieser Algorithmus wurde auch verwendet, um in dem neuen Programm 'TDD_IR' *shortest unique substrings* aus der durch *TDD* erstellten Struktur auszulesen, allerdings in etwas modifizierter Form. Noch bevor die Klassen dieses neuen Programms vorgestellt werden, soll deshalb hier auch gleich der zugehörige Programmcode zur Auslesemethode beschrieben werden.

In einer Klasse, die *IRcalculator* genannt wurde, befinden sich zwei Methoden namens 'find_shustring', und 'collect_all_shustring_leaves'. Darin werden auch noch die zwei Methoden 'nextnode' und 'labelLength' aufgerufen. Letztere wurde von den *TDD*-Autoren übernommen und liefert eine effizient implementierte Funktion, um die in einen Knoten oder ein Blatt eingehende Kantenlänge zu bestimmen.

Es sollen hier erstmal nur vereinfachte Versionen von 'find_shustring' und 'collect_all_shustring_leaves' aufgeführt werden, die die zentralen Punkte enthalten, die zum Auslesen der *Shustrings* dienen. Möchte man nur lokale *shortest unique substrings*, bzw. den I_r von einem einzelnen Strang auffinden, könnten diese zwei Methoden so wie sie hier abgebildet sind, auch im Programm verwendet werden.

Da *find_shustring* die Methode ist, die letztendlich von der Benutzerschnittstelle aus aufgerufen wird, enthält sie noch weitere Übergabeparameter zur Berechnung des *Index of Repetitiveness*, wie z.B. die Fensterlänge zur 'windows analysis', die aber hier zum besseren Verständnis weggelassen wurden. Es wird in dieser Methode im wesentlichen ein Zeiger ('Pointer') auf ein 'Short Integer'-Array namens 'shustrings' übergeben, welches später mit den *shortest unique substrings* gefüllt wird. Als erstes wird mit der Methode 'size_without_padding', die aus der *TDD*-Implementierung stammt, die Größe des Arrays bestimmt, in dem der Suffixbaum dargestellt ist. Die Variable 'start' gibt die Position im Array wieder, an der man sich gerade befindet, zu Beginn an Index Null (s. nächste Seite).

```
//IRcalc::find_shustring
void IRcalc::find_shustring(short int* shustrings{
    int array_size = size_without_padding();
    Tindex start=0;
    while(!rightmost->is_set(start) && start < array_size){
        int ncdepth = labelLength(start);
        collect_all_shustring_leaves(start,ncdepth,shustrings);
        if(leaf->is_set(start)){
            start++;
        }
        else{
            start+=2;
        }
    }
    Tree.close();
    gesamtIR(shustrings);
    if(window==true){
    outputWindowAnalysis(shustrings);
}
//end IRcalc::find_shustring
```

Die *while*-Schleife wird solange durchlaufen, bis die erste Position, die mit 'rightmost' gekennzeichnet ist, oder das Ende des Arrays erreicht ist. Über eine weitere *TDD*-Methode, nämlich 'labelLength' wird hier bereits die Länge der in diesen Knoten eingehenden Kante bestimmt und als 'ncdepth' bezeichnet. Diese wird zusammen mit der momentanen Position im Array und den 'Pointer' auf das Shustring-Array übergeben und zwar an die Methode 'collect_all_shustring_leaves'. Ist die Berechnung in dieser Methode beendet, geht es im Array weiter und zwar im Falle eines Blattes um einen Schritt und bei Verzweigungsknoten um zwei Schritte. Ist die while-Schleife beendet und sind alle Knoten und ihre Kinder abgearbeitet, wird der vor diesem Methodenaufruf initialisierte Baum aus dem Hauptspeicher entladen. Für die Berechnung des globalen I_r oder der Windowanalyse wird ein mit den berechneten *Shustringlängen* erstelltes Array an die entsprechenden Methoden *gesamtIR* und *outputwindowanalysis* übergeben.

Was man aber vorher noch braucht, sind alle Blätter und ihre zugehörigen *Shustringlängen*, die man mit Hilfe der als nächstes aufgeführten Methode 'collect_all_shustring_leaves' findet. Wie der Name schon ahnen lässt, wird hierfür jeder im Array aufgeführte Knoten bis zu seinen Blättern weiterentwickelt und deren *shortest unique substrings* berechnet. Im nachfolgenden Listing steht wieder der auf die Shustringberechnung begrenzte Code. Der Unterschied zur ausführlichen Version besteht hauptsächlich in der Behand-

lung der randständigen *Shustrings*, weil in der 'Vollversion' die Auswertung von mehreren Sequenzen berücksichtigt wurde, die in der 'TDD'-Notation durch das Zeichen ']' getrennt sind. Übergeben wird hier also nur die Position im Suffixbaumarray, die Kantenbeschriftungslänge des aktuellen Knotens und der Verweis auf das Array für die *shortest unique substrings*. Es wird zusätzlich die Länge des Strings über die Klassenvariable 'StringSize' bestimmt, die vorher bei der Rekonstruktion des Baumes initialisiert wurde (s. unten).

```
void IRcalc::collect_all_shustring_leaves(Tindex start, int ncdepth,
short int* shustrings){
    int str_sz = StringSize;
    ND_t a;
    Stack<ND_t>* st;
    st = new Stack<ND_t>::Stack(STACK_SIZE);
    st->push(ND_t(start,ncdepth));
    while(!st->isempty()){
        a = st->pop();
        if(leaf->is_set(a.node)){
            int stringdepthparent = a.depth-labelLength(a.node);
            int Position = (Tree.read(a.node)-stringdepthparent)+1;
            int shustringlength = stringdepthparent+1;

            if(Position<(str_sz-1)){
                if((Position+shustringlength)<=str_sz-1){
                    shustrings[Position-1]=shustringlength;
                }
                else{
                    shustrings[Position-1]=str_sz-Position;
                }
            }
        }
        else{
            a.node = Tree.read(a.node+1);
            Tindex newdepth = 0;
            while(1){
                newdepth = a.depth+labelLength(a.node);
                st->push(ND_t(a.node, newdepth));
                if(rightmost->is_set(a.node)) break;
                else nextnode(a.node);
            }
        }
    }
    delete st;
}
```

Wie im Algorithmus beschrieben, wird nun ein Knoten auf einen neu alloziierten Stack gelegt. Im nächsten Schritt wird dieser Knoten in der *while*-Schleife auch gleich wieder heruntergenommen und geprüft, ob es sich um ein Blatt handelt. Weil das in unserem Beispiel nicht der Fall ist, ist die folgende 'if'-Bedingung nicht erfüllt und man steigt gleich hinab zum 'else'-Zweig. Dort wird der Eintrag im Suffixbaumarray für den ersten Kindknoten über den *TDD*-Befehl *Tree.read(a.node+1)* festgestellt, hier also zwölf. In der folgenden 'kleinen' while-Schleife werden jetzt alle Knoten auf den Stack gelegt, bis ein Blatt folgt, das mit 'rightmost' gekennzeichnet ist. Der *TDD*-Befehl *nextnode* gibt dabei diese Knoten bzw. Blätter zurück. Der Unterschied zu dem beschriebenen theoretischen Algorithmus ist, dass die Knoten und Blätter, gleich mit ihrer über *labelLength* erhaltenen Stringtiefe auf den Stack gelegt werden. Außerdem bewirkt der *nextnode*-Befehl, dass erst alle Verzweigungsknoten auf den Stack gelegt werden und im nächsten Schritt die Blätter des letzten Knotens. Diese und ihr zugehöriger Knoten werden dann vom Stack genommen und dann erst die Blätter für den nächsten Verzweigungsknoten daraufgelegt. In Anlehnung an Abb. 5.3 bedeutet dies nun ein 'push' und 'pop'-Verhalten, wie in Grafik 5.4 gezeigt, zum besseren Verständnis wieder mit den entsprechenden Suffixbaumarrayeinträgen und nicht mit den neuen Stringtiefen eingezeichnet.

Um nun endlich die *shortest unique substrings* auslesen zu können, wird die *while*-Schleife im *else*-Zweig verlassen, sobald ein als 'rightmost' markiertes Blatt auf den Stack gelegt worden ist. Mit dem so befüllten Stapel geht es dann zurück in die übergeordnete *while*-Schleife, wo nun das oberste Element heruntergenommen wird. Nachdem es sich jetzt um ein Blatt handelt, ist die *if*-Bedingung erfüllt. Die Stringtiefe des Blattes wurde ja in den vorherigen Schritten ermittelt und mit auf dem Stack abgelegt. Nun lässt sich wieder über die *labelLength*-Methode die Länge der in das Blatt eingehenden Kantenbeschriftung ermitteln. Nun braucht man nur noch diese *Labellänge* von der Stringtiefe des Blattes abziehen und erhält als Ergebnis die Stringtiefe des Elternknotens. Durch die Addition mit *eins* ergibt sich aus dieser schließlich die Länge des *shortest unique substrings*. Die Position für den *Shustring* erhält man, indem man die Differenz aus ebenfalls dem Eintrag des Blattes im Suffixbaumarray und der Stringtiefe des Elternknotens bildet und das Ganze mit *eins* addiert (im Code: *int Position = (Tree.read(a.node)-stringdepthparent)+1;*). Im Programmcode wird dann noch überprüft, ob es sich um einen entsprechend Kapitel 4.2.1 wohldefinierten *Shustring* handelt und falls nicht, wird die *Shustringlänge* als Summe aus der *Suffixlänge* an dieser Position und *Eins* neu definiert. Der dafür verwendete Term *str_sz-Position* entspricht dieser Definition, weil *TDD* einen *String* intern mit zwei Schrägstrichen am Ende, also '\\' darstellt und somit die reine Buchstaben-

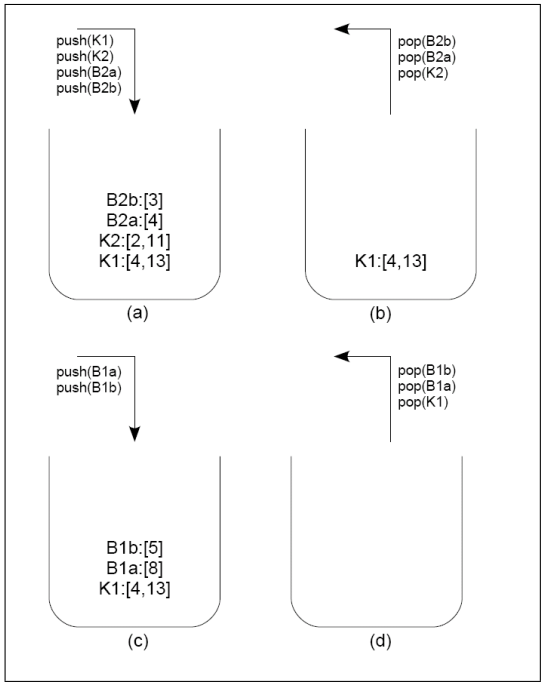


Abbildung 5.4: Es werden zunächst die Verzweigungsknoten K1 und K2 auf den Stack gelegt. Dann folgen die Blätter von K2, also B2a und B2b. Diese drei werden dann auch umgekehrt wieder vom Stapel entfernt. Auf den zurückgebliebenen Knoten K1 werden nun seine Blätter B1a und B1b gelegt. Ebenfalls in umgekehrter Reihenfolge werden diese Drei anschließend wieder heruntergenommen.

länge um zwei erhöht. Gibt es z.B. bei einer *Stringlänge* von 9 an der letzten Position keinen wohldefinierten *Shustring*, wird dieser als $11 - 9 = 2$ definiert. In das Array für die *Shustringlängen* werden diese nun an der jeweils entsprechenden Position eingetragen.

Hat man alle Blätter auf dem Stack auf diese Art und Weise abgearbeitet, liegt entweder noch ein Verzweigungsknoten auf dem Stack, der wieder die *else*-Kondition aufruft, oder der Stapel ist ganz leer. Er wird dann aus dem Hauptspeicher entfernt. Nun erfolgt der Schritt zurück zur Methode *find_shustring*. Dort wird dann der nächste Knoten betrachtet, der die Verarbeitung eines weiteren Zweiges aufruft. Ist in *find_shustring* ein 'rightmost' erreicht, ist das *Shustringarray* fertig gefüllt und bereit, um damit den *Index of Repetitiveness* berechnen zu können.

5.3 Implementierungsstruktur

Neben einem Objekt der Klasse *IRcalculator* benötigt man, wie man an den Methoden zur Bestimmung der *shortest unique substrings* schon sehen kann, noch weitere Datenstrukturen, nicht zuletzt, um überhaupt den auf Festplatte abgespeicherten Suffixbaum zu rekonstruieren. Ein vereinfachtes Klassendiagramm zu dem neuen Programm *TDD_IR* findet sich in Abbildung 5.5. Die ausführliche Version mit allen Attributen und Methoden ist im Anhang A aufgeführt. Entsprechend dieser Grafiken, soll nun erst in tabellarischer Form eine kurze Beschreibung der in *TDD_IR* enthaltenen Dateien und Klassen folgen und im Anschluss daran ein einfacher Programmablauf dargestellt werden.

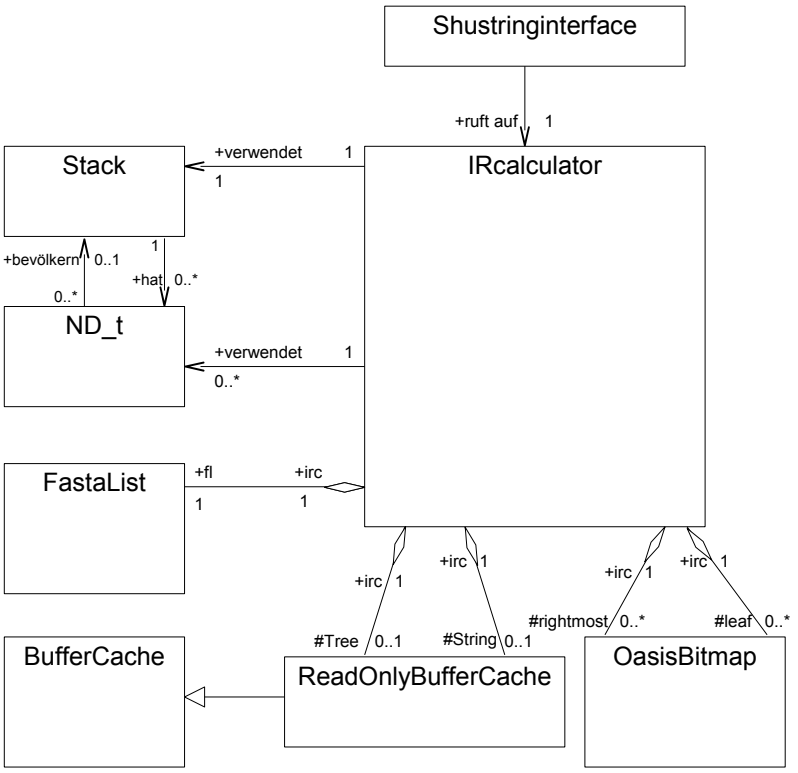


Abbildung 5.5: Klassendiagramm der Implementierung zu TDD_IR.

Die Datei *tdd_ir.cpp* liefert das 'User Interface' auf Kommandozeilenebene unter einem UNIX-System. Über diese Benutzerschnittstelle werden sowohl die zu analysierende Datei und ihr Eingabeformat, als auch mögliche Optionen zur Auswertung eingegeben. Bei 'falschen' Eingaben wird hier auch die

Tabelle 5.1: Kurzbeschreibungen der *TDD_IR*-Dateien.

Datei	*Funktion*
`BufferCache.h`	Definitionen für die BufferCache Klassen.
`FastaList.h`	Liefert Sequenzbeschreibungen zurück.
`ir_calculator.h`	*Shustring*-Detektion und I_r-Berechnung.
`mytypes.h`	Globale Typdeklarationen.
`oasisBM.h`	Eine 'bitmap class'.
`shustringinterface.h`	Schnittstelle für das User-Interface.
`Stack.h`	Liefert eine einfache *Stack*-Klasse.

richtige Syntax zum Programmaufruf erläutert. Bei noch nicht vorhandenen Bäumen wird das Programm *tdd* zu deren Erstellung aufgerufen. Die erhaltenen Parameter werden an die abstrakte Klasse *shustringinterface* übergeben. Die weiteren Dateien sind in Tabelle 5.1 zusammengefaßt.

Etwas ausführlicher sollen die einzelnen Klassen beschrieben werden. Begonnen wird mit der abstrakten Klasse Shustringinterface, der Rest folgt wieder alphabetisch.

Shustringinterface: Schnittstelle für das 'User-Interface'. Ruft die Rekonstruktion eines Baumes über ein $IRcalculator - Objekt$ auf und übergibt an dieses die vom 'User Interface' erhaltenen Optionen. Enthält noch Methoden, um globale und lokale *shortest unique substrings* unabhängig vom I_r auszugeben und bereitet den Eingabestring vor, falls noch der Rückwärtsstrang berechnet, oder mehrere Sequenzen in einem Durchlauf analysiert werden sollen.

BufferCache<T>: Eine sog. 'Template'-Klasse, die ein Interface für gepufferte Datenstrukturen für die Festplattenkonstruktion liefert. Von ihr erben weitere *Template*-Klassen, von denen nur *ReadOnly_BufferCache*$<T>$ verwendet wurde.

FastaList: Werden mehrere Sequenzen verarbeitet, aggregiert sie *TDD* zu einem einzigen durch das Zeichen ']' abgeteilten *String*. Zudem speichert *TDD* die jeweiligen sog. 'Descriptions' in einer Datei ab. Ein Objekt dieser Klasse liefert Methoden, um die Sequenzbeschreibungen anhand der Position im *String* wieder auszulesen.

IRcalculator: Zentrale Klasse, die das Laden und Auslesen der persistent gespeicherten Suffixbäume implementiert, aber auch die Methoden zum Auffinden der *Shustringlängen* und zur Berechnung des I_r enthält. Aus

dem Programm *tdd* wurden im wesentlichen die Methoden zum Auslesen bzw. Laden der Sequenz (String), des Baumes (Tree) und der *Bitmaps*, sowie die oben erwähnten Hilfsmethoden zur *Shustring*-Bestimmung unverändert übernommen. Aus dem Programm *ir* konnten die Methoden zur Berechnung der erwarteten *Shustringlängen* ohne wesentliche Abänderung in den Code eingebunden und verwendet werden.

IRcalc::ND_t: Ein Datentyp für einen Knoten, der zur Shustringbestimmung verwendet wird. Enthält die Startposition eines Knoten- oder Blattsuffix (entsprechend dem Suffixbaumarrayeintrag) und die zugehörige Stringtiefe.

OasisBitmap: Implementierung einer 'bitmap class', deren Objekte zur Markierung von 'normalen' und 'rightmost' Blättern dienen.

ReadOnly_BufferCache<T>: Eine 'Template'-Klasse, die von *BufferCache<T>* erbt. Hier werden keine Modifikationen auf Platte geschrieben, sondern nur vorhandene Strukturen ausgelesen.

Stack<T>: Eine Template-Klasse, mit grundlegenden Stackfunktionen wie push(element E), pop(), isempty() und size().

Für ausführlichere Beschreibungen der TDD-Klassen bzw. Dateien 'FastaList.h', 'mytypes.h', 'oasisBM.h/.cpp, 'Stack.h' und 'BufferCache.h' sei auf die dortige Dokumentation verwiesen (http://www.eecs.umich.edu/tdd/doc/hierarchy.html), da diese unverändert in den *TDD_IR*-Programmcode integriert wurden.

Die Klasse *IRcalculator* enthält wie erwähnt Methoden, um die Suffixbaum-Datenstruktur von der Platte lesen zu können, die nahezu unverändert aus dem Programm *tdd* übernommen werden konnten. Zentrale Struktur war hierbei noch das Klassentemplate *ReadOnlyBufferCache<T>*, das für die Strukturen 'String' und 'Tree' verwendet wurde. Für die Rekonstruktion der Blätter wurden ebenfalls mit *OasisBimap* Objekte aus einer *TDD*-Klasse benützt. Aus dem Programm *ir* wurden mehrere Methoden zur Berechnung des *Index of Repetitiveness* verwendet, wobei z.T. nur die Übergabeparameter angepaßt werden mußten, wie z.B. zur Berechnung der theoretischen Längen der *shortest unique substrings*. Andere Methoden, wie z.B. zur Fensteranalyse mußten weitgehend neu implementiert werden, was hauptsächlich daran lag, dass *TDD* wie nun schon desöfteren erwähnt, bei der Analyse von mehreren Sequenzen diese zu einem String zusammenfasst und die einzelnen Sequenzen durch eine eckige Klammer ']' voneinander trennt. Daraus wird dann der

Suffixbaum aufgebaut. Deshalb musste erst überprüft werden, wo die Klammern im String auftauchen, d.h. wo die einzelnen Sequenzen beginnen bzw. enden, um ihnen die jeweiligen *Shustrings* zuordnen zu können.

Damit war nun ein Programm vorhanden, welches auf die menschlichen Chromosomen angewandt werden konnte.

Über ein User-Interface auf Kommandozeilenebene können nun die zu analysierenden Datensätze übergeben und zusätzliche Optionen ausgewählt werden (siehe Anhang). Ein Aufruf der entsprechenden Methode der abstrakten Klasse *Shustringinterface* leitet dann die Initialisierung des *IRcalculator*-Objektes und das Einlesen von String, Tree und Blattidentifikatoren (*leaf* und *rightmost*) ein. Wenn nötig, wird hier der String noch vorverarbeitet, z.B. indem der Rückwärtsstrang erstellt wird. Während auf den String letztendlich nur noch zur Bestimmung der *GC*-Anteile zugegriffen wird, wird der Baum natürlich intensiv für die Bestimmung der *shortest unique substrings* genutzt, wie oben (Abschnitt 5.2) beschrieben. Dazu benötigt man ebenfalls analog zum Algorithmus noch einen *Stack*, auf dem die berechneten Knoten und Blätter als spezielle Knotenstruktur abgelegt werden können. Der Vergleich mit den *OasisBitmap*-Objekten ist notwendig, um festzustellen, ob es sich bei einem Knoten um eine interne Verzweigung handelt oder um ein externes Blatt. Ebenfalls wird damit ermittelt, ob es sich um ein äußerstes rechtes handelt. Nach der Bestimmung der *shortest unique substrings* können alle diese Objekte wieder aus dem Hauptspeicher gelöscht werden. Was man zurückbekommt, ist ja ein Integerarray, dessen Index die Position in der Sequenz wiedergibt und dessen Wert der Länge des *shortest unique substrings* an dieser Stelle entspricht. Das *Shustringinterface* enthält noch zwei Methoden zur Rückgabe der globalen bzw. lokalen *shortest unique substrings*, die bei Bedarf nun aufgerufen werden. Ansonsten wird über Methoden des *IRCalculator*-Objektes der globale I_r berechnet und bei Bedarf eine Fensteranalyse durchgeführt.

Beispiele zum Programmaufruf, den einzelnen Optionen und den zurückgelieferten Daten findet man in Anhang. Als vorweggenommenes Beispiel soll hier erstmal der Aufruf

```
\$ ./tdd_ir -i 143967.fasta -w 100000
```

dienen. './tdd_ir' ruft das Programm auf. Die Option *-i* bedeutet, dass erst ein Baum erstellt und somit das Programm *tdd* vorab gestartet und abgelaufen sein muss (Alternativoption bei vorhandenem Baum: *-t*). *-w* läutet eine *Window*-Analyse ein und zwar mit einer Fensterlänge der nachfolgenden Zahl (hier: *100000*). Man erhält folgende Ausgabe:

```
# Len       IR
580074    0.142855

>L43967 L43967.1 MYCPLASMA GENITALIUM G37 COMPLETE GENOME.
50000.5   0.07087019410
60000.5   0.07241391039
70000.5   0.07455525539
...
```

Als erstes wird die Länge der analysierten Sequenz, sowie der zugehörige Gesamt-I_r ausgegeben und im Anschluss daran die Mittelwerte der einzelnen Fenster, mit dem für die jeweiligen Intervalle berechneten I_r. Wie man sieht, wird auch der Header der FASTA-Datei mit zurückgeliefert. Dazu braucht man noch ein Objekt der Klasse FastaList. TDD speichert nämlich die Sequenznamen in einer extra Datei und benötigt die FastaList-Struktur, um sie auszulesen und v.a. bei mehreren Sequenzen korrekt zuweisen zu können.

Natürlich lässt sich das Programm auch auf das Humangenom anwenden. Im nächsten Kapitel wird diesbezüglich erläutert, welche Chromosomengrößen damit analysiert werden konnten, worauf man achten und welche Vorarbeit man leisten musste.

5.4 Testumgebung, Möglichkeiten und Grenzen

Das Programm wurde auf folgenden Maschinen getestet:

Maschine A: Betriebssystem: DEBIAN Linux Version 3.1

Kompiler: gcc-version 3.3.5

Prozessor: Intel(R) Pentium(R) 3 CPU 1.00 GHz

Hauptspeicher: 4 GB

Maschine B: Betriebssystem: DEBIAN Linux KNOPPIX-Version 5.0.1

Kompiler: gcc-version 4.0.4

Prozessor: Intel(R) Pentium(R) 4 CPU 3.00 GHz

Hauptspeicher: 1 GB

Ehe auf die Ergebnisse eingegangen wird, soll noch erklärt werden, welche Vorarbeit dafür geleistet werden musste, z.B. die Auswahl der Funktionalitäten. In aller Kürze sind dies die Erstellung und Berücksichtigung des Rückwärtsstranges, die Möglichkeit, mehrere Sequenzen in einer FASTA-Datei

getrennt zu behandeln, Einstellungen für die Fensteranalyse und die Ausgabe aller globalen und lokalen *shortest unique substrings*. Am wichtigsten ist allerdings die Auswahl aus zwei verschiedenen Eingabeformaten. Einmal kann ein bereits vorher erstellter *TDD*-Baum direkt bearbeitet werden, auch wenn die zugehörigen Dateien z.B. von einem anderen System portiert wurden. Die zweite Möglichkeit dient der Verarbeitung von 'FASTA'-Dateien. Dazu muß allerdings vorher das Programm *tdd* installiert sein, da ja erst die persistenten Suffixbäume erstellt werden müssen. Der Konstruktionsaufruf erfolgt dann automatisch über das Programm *TDD_IR*.

Ebenfalls selbsttätig erfolgt die Umwandlung aller Sequenzbuchstaben einschließlich des Headers in Großbuchstaben. Dazu wird *TDD_IR*-intern das UNIX-Tool *tr* aufgerufen ('tr 'a-z' 'A-Z' < inputfile > outputfile). *TDD* kann nämlich nicht mit Dateien umgehen, die Kleinbuchstaben enthalten. Da bei entsprechenden Tests keine Fehlermeldung sondern nur eine 'unendliche' Laufzeit sogar bei kleinen Sequenzen beobachtbar war, konnte dieses Problem auch nicht in annehmbarer Zeit beseitigt werden, weshalb die Substitutionsvariante am einfachsten erschien.

Das nächste Problem war, dass *TDD* sehr hohe Laufzeiten bekommt, bzw. nicht beendet wird, wenn eine hohe Zahl an Wiederholungen in der Sequenz vorhanden sind. Dies ist z.B. der Fall, wenn große, nichtsequenzierte Bereiche eines Genoms mit 'N' gekennzeichnet sind. Dieser Umstand wird auch auf der *TDD*-Homepage unter http://www.eecs.umich.edu/tdd/FAQ.html beschrieben, aber leider dort nicht begründet. Vermutlich liegt die Ursache am quadratischen Laufzeitverhalten des *wotdeager*-Algorithmus aus Kapitel 3.1 und würde deshalb bei allen großen Wiederholungssequenzen, die sich über Millionen von Basen erstrecken, starke Verzögerungen hervorrufen. Innerhalb von Genomen treten solche Fälle praktisch nur bei großen Aneinanderreihungen von N's auf, weshalb aber *TDD* schon nicht mehr auf Chromosom 21 anwendbar war. Ähnlich wie bei dem 'Kleinbuchstabenproblem' kam es zu keiner Fehlermeldung, sondern zu unendlicher Laufzeit. Z.B. wurde die Verarbeitung von *Chromosom 21* nach *30* Stunden per Hand abgebrochen. Da die nichtsequenzierten Bereiche andererseits recht kompakt zusammenliegen, z.B. bei *Chromosom 21* u.a. am Anfang, wurde entschieden, diese Sequenzteile 'auszuschneiden'. Für die weitere Auswertung hat das keine Bedeutung, da man ja unsequenzierten Bereichen auch keine Funktion zuordnen kann. Außerdem würde eine hohe Anzahl an hintereinanderstehenden 'N' wesentlich die Repetitivität erhöhen. Die Entfernung dieser Bereiche wurde für den interessierten Leser auf der UNIX-Konsole mittels des PERL-Befehls *perl -pe 's/N//g'* vollzogen.

Was also getestet wurde, sind in Großbuchstaben dargestellte DNA-Sequenzen ohne nichtsequenzierte Bereiche. Damit konnten folgende menschliche Chro-

mosomen analysiert werden, wenn auf derselben Maschine der Suffixbaum noch konstruiert werden mußte.

Maschine A:

Einzelstrang: *Chromosom 4: 179 MB, 187297063 Basen*

Doppelstrang: *Chromosom 14: 169 MB, 176581170 Basen*

Maschine B:

Einzelstrang: *Chromosom 17: 75 MB, 77800220 Basen*

Doppelstrang: *Chromosom 21: 66 MB, 68340212 Basen*

Die Restriktionen der verarbeitbaren Größen liegen ausschließlich am Programm *tdd*, z.B. wurden die *TDD*-Dateien für Suffixbäume größerer Datensätze, die auf *Maschine A* erstellt wurden, nach *Maschine B* transferiert und konnten dort mit *TDD_IR* und der Option *-t*, die TDD-Baumstrukturen direkt einbindet, verarbeitet werden. Damit war es noch möglich, auf Maschine B die einzelsträngige Sequenz bzw. den dazugehörigen Baum von *Chromosom 14* mit *88290587 Mb*, also ca. *90,06 MB*, zu verarbeiten. Die Größe des persistenten Suffixbaumarrays ist ca. *zehn* Mal so groß wie die Eingabesequenz selber. Deshalb konnten auch keine größeren, auf *Maschine A* erstellten Chromosomenbäume mehr analysiert werden, weil die Begrenzung wiederum die zur Verfügung stehende Hauptspeichermenge darstellt. Die *TDD*-Struktur 'ReadOnlyBufferCache', die zum Lesen der Bäume und des Strings verwendet wurde, legt nämlich diese komplett im Hauptspeicher ab. Den String braucht man eigentlich nur zur Berechnung des *GC*-Gehaltes der Sequenz (siehe Kap. 4), weshalb diese vorab durchgeführt, der *String* dann wieder entladen und der ganze *RAM*[1] dem Baum zur Verfügung gestellt werden kann. Die in [TTHP05] vorgestellte Partitionierung eines Suffixbaumes über die Wahl einer größeren Präfixlänge als *1*, könnte hier eine Lösung bieten, aber leider war es auch nicht möglich diese Funktion, ähnlich wie *ST-Merge* (Kapitel 3.1.3), in der praktischen Implementierung zur Anwendung zu bringen.

Damit hat man nun ein Programm, mit dem zwar deutlich größere Sequenzen als unter Verwendung der in Kapitel 2.2.2 erwähnten linearen 'in-memory'-Suffixbaumkonstruktionsalgorithmen analysiert werden konnten (Chromosom 21 war damit z.B. schon nicht mehr möglich), aber das gewünschte Ziel des kompletten Humangenoms und sogar größerer Chromosome wurde leider verfehlt. Weiterer Nachteil ist, dass praktisch in jedem Chromosom

[1] Random Access Memory

noch nichtsequenzierte Bereiche vorhanden sind und man deshalb auch immer erst diese Regionen entfernen und dann die Positionen zurückrechnen muß. Ein parallel zu *TDD_IR* von *Haubold* und *Wiehe* erstelltes auf Suffixarrays basierendes Programm (erhältlich unter http://adenine.biz.fh-weihenstephan.de/homePage/ir) hat dagegen damit keine Probleme und bietet auch sonst einige Vorteile. Im nächsten Kapitel soll deshalb noch kurz auf Suffixarrays eingegangen und abschließend ein Performance-Test zwischen den beiden Programmen vorgestellt werden.

5.5 Suffixarrays als Alternative zu *TDD*

Obwohl Suffixarrays nahe verwandt zu Suffixbäumen sind und als eine Alternative für viele Stringverarbeitungsaufgaben gelten [AKO04, CCY, CF99, NBYT01], wurde die Verwendung zunächst nicht in Betracht gezogen, weil hier zu Beginn der Diplomarbeit keine Implementierungen zu Disk-basierenden Algorithmen zur Verfügung standen. Eine sehr speichereffiziente 'in-memory'-Methode namens 'deep-shallow' wurde jedoch in [MF04] vorgeschlagen, die nur *5,03n Bytes* Hauptspeicher pro Eingabesequenzzeichen benötigen soll. Aus diesem Grund hatte es einen lohnenswerten Anschein, sich näher mit Suffixarrays auseinanderzusetzen.

5.5.1 Konstruktion eines Suffixarrays

Wie man an der Beschreibung des 'Top-Down Disk-Based'-Algorithmus in Kapitel 3 schon gesehen hat, lassen sich Suffixbäume auch als Arraystruktur darstellen. Benötigt man nicht alle Informationen, die in einem Suffixbaum vorhanden sind, ist es auch möglich eine platzsparende Variante namens 'Suffixarray' zu benutzen. Ein Suffixarray A für einen String S ist ein Integerarray der Länge $|S|$. Dieses Array wird befüllt, indem alle Suffixe von S zuerst nach ihrer lexikographischen Reihenfolge sortiert werden und dann die jeweilige Startposition des i-ten Suffix in $A[i]$ eingetragen wird. Für die bisher verwendete Beispielsequenz S=*ATTAGTACA$* ist dies in Tabelle 5.2 dargestellt. Die Indizizierung kann dabei wie bei Suffixbäumen mit *0* oder *1* beginnen.

Dieses Array kann z.B. einfach aus einem Suffixbaum gewonnen werden, wenn dessen Kanten bereits lexikografisch sortiert sind. Dazu muß man aber vorher erst einen Suffixbaum erstellen, weshalb man weiterhin das Problem der Speicherplatzbegrenzung hat [MF04]. Alternativ dazu kann man auch erst alle Suffixe der Länge nach darstellen, wie in Tabelle 5.2 links und sie dann über einen vergleichsbasierenden Algorithmus sortieren. Effiziente Methoden wurden hierzu von Larsson und Sadakane [LS99], Itoh und Tanaka [IT99],

Tabelle 5.2: Darstellung der Sequenz *ATTAGTACA$* als Suffixarray. Links außen stehen die nach ihrer Länge sortierten Suffixe von S und daneben die lexikographisch sortierten. Ins Suffixarray werden entsprechend der Sortierung schließlich die Startpositionen der Suffixe in S eingetragen.

unsortierte Suffixe	*sortierte Suffixe*	*Arrayindex mit Suffixstartposition*
ATTAGTACA$	A$	A[0] = 8
TTAGTACA$	ACA$	A[1] = 6
TAGTACA$	AGTACA$	A[2] = 3
AGTACA$	ATTAGTACA$	A[3] = 0
GTACA$	CA$	A[4] = 7
TACA$	GTACA$	A[5] = 4
ACA$	TACA$	A[6] = 5
CA$	TAGTACA$	A[7] = 2
A$	TTAGTACA$	A[8] = 1

sowie Seward [Sew00] vorgestellt. Der zuletzt genannte Algorithmus wurde von Manzini und Ferragina weiterentwickelt und das Ergebnis in [MF04] vorgestellt. Dort wird die neuentwickelte Technik nach ihrer Suffixsortiermethode 'deep-shallow' genannt und im Experiment gezeigt, dass sie schneller ist als die anderen getesteten Algorithmen. 'deep-shallow' hat einen linearen $O(n)$ Speicherbedarf, genauer gesagt benötigt dieser Algorithmus $5,03n$ Bytes pro Zeichen [MF04]. Die Laufzeit beträgt durchschnittlich $O(nlogn)$ und im 'worst-case' $O(n^2logn)$. Der letztere Fall tritt nur auf, wenn jedes Suffix ein Präfix eines anderen Suffix ist, d.h. z.B. eine Sequenz, die nur aus gleichen Buchstaben besteht. Die Methode liefert auch noch zusätzlich eine Datenstruktur zurück, die Informationen über das längste gemeinsame Präfix (engl. longest common prefix, kurz: LCP) von zwei aufeinanderfolgenden Suffixen im Array enthält. Mit Suffixarray und LCP-Tabelle hat man nun eine den Suffixbäumen nahezu äquivalente Datenstruktur, die in linearer Zeit in einen Baum umgewandelt werden kann [TTHP05] und mit der sich z.B. auch der tiefste gemeinsame Elternknoten berechnen lässt.

5.5.2 Ermittlung der tiefsten gemeinsamen Vorfahren

Wie dies funktioniert, soll anhand einer Darstellung aus einer Arbeit von Fischer, Heun und Kramer [FVK06] gezeigt werden. Dort wurde auf Basis der *deep-shallow*-Technik ein O(n)-Algorithmus vorgestellt, um z.B. Datenbankabfragen von häufig vorkommenden Mustern effizient zu bewältigen. Das dazu benötigte Suffixarray und seine LCP-Tabelle ist in Abbildung 5.6 dar-

gestellt.

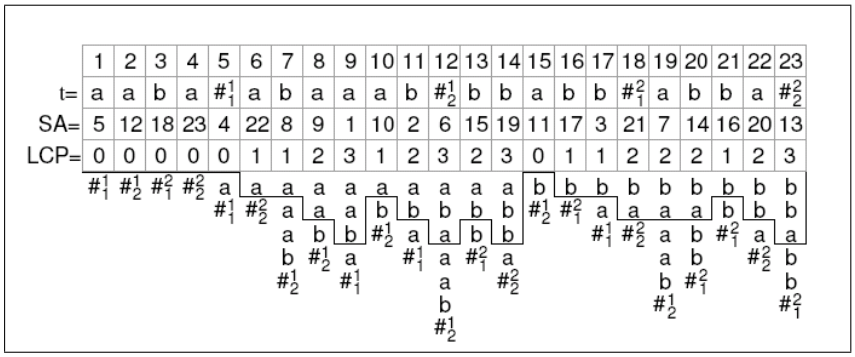


Abbildung 5.6: Das Suffixarray und seine LCP-Tabelle für die vier Strings aaba, abaaab, bbabb und abba.

Hier werden gleich vier Strings verarbeitet, nämlich aaba, abaaab, bbabb und abba. Diese werden ähnlich wie bei *TDD* zu einem einzigen String zusammengefaßt und durch das Zeichen '#' getrennt. Diese Aggregation steht in der zweiten Zeile von oben, gleich unter den Indizes. Darunter stehen die Suffixarray-Einträge und darunter schließlich die LCP-Tabelle. Die Stringtiefe der letzten Elternknoten findet man nun, indem man den maximalen Wert aus zwei benachbarten LCP-Einträgen nimmt und der Position zuweist, die beim ersten Eintrag im Suffixarray steht. Im Beispiel stehen in der LCP-Tabelle an Index 8 und 9 die Einträge 2 und 3. Das heißt, $max\{2,3\} = 3$. Somit ist die Länge des tiefsten Vorfahren an Position 9 (Suffixarray-Eintrag an Index 8) ebenfalls gleich 3. Hat man so für jeden Suffix die Stringtiefe des letzten Elternknotens ermittelt, braucht man nur noch zu jedem Wert eins hinzuaddieren und hat wieder die gewünschte Länge des 'shortest unique substrings' an dieser Stelle. Weitere Erläuterungen zu diesem Beispiel und näheres zu Suffixarrays im Allgemeinen, findet man in den oben genannten Arbeiten. Aus der mittlerweile großen Fülle an weiterer Literatur sollen auch noch die Arbeiten von Manber und Myers [MM93], die als 'Erfinder' der Suffixarrays gelten und nicht zuletzt das Buch von Gusfield [Gus97] genannt werden.

Ein großer Vorteil von Suffixarrays ist, dass die in Kapitel 5.4 erwähnten Probleme mit nichtsequenzierten Bereichen einen deutlich geringeren Einfluß auf die Suffix-Array-Konstruktion haben. Dadurch erhält man sofort die richtigen Positionen, ohne rückrechnen zu müssen. Dass dies nicht der einzige Vorteil ist, soll im nächsten Abschnitt an einem Vergleich der Programme *TDD_IR* und *ir*, die beide der Berechnung des *Index of Repetitiveness* dienen, gezeigt werden.

5.6 Performancevergleich zwischen *tdd_ir* und *ir*

Die neue Version[2] des Programms *ir* beruht nun auf Suffixarrays und dem *deep-shallow*-Algorithmus in Ahnlehnung an die Implementierung von *Fischer et al.*

Im Anschluß soll ein kurzer Performance-Vergleich zeigen, wie sich *tdd_ir* und *ir* bei der Berechnung des Index of Repetitiveness unterscheiden. Verwendet wurden dazu das Genom von *Mycoplasma genitalium* und *Chromosom 21* des Menschen, allerdings mit entfernten nichtsequenzierten Bereichen. Damit handelt es sich um folgende doppelsträngige Testsequenzen:

Mycoplasma Genitalium: *1,2 MB, 1160148 Basen*

Chromosom 21: *66 MB, 68340212 Basen*

Die Tests wurden dieses Mal nur auf *Maschine B* (siehe Kapitel 5.4) durchgeführt, da es sich bei *Maschine A* um einen Server handelt, auf dem natürlich noch weitere Programme laufen und man diesen Einfluss nicht endgültig abschätzen konnte. *Maschine B* ist ein lokaler Rechner, mit dem für alle Tests gleiche Rahmenbedingungen geschaffen werden konnten. Jede der zwei Dateien wurde jeweils fünf Mal analysiert und dann die Mittelwerte für Zeit- und Speicherbedarf ermittelt. Herangezogen wurde als Zeit die 'usertime' die man über den dem Programmaufruf vorangestellten UNIX-Befehl 'time' erhielt. Die Ergebnisse kann man in Abbildung 5.7 sehen.

	Mycoplasma genitalium		Chromosom 21	
	Zeit [s]	Hauptspeicher [MB]	Zeit [s]	Hauptspeicher [MB]
Ir	0,82	< 2	74,04	586
tdd_ir Option -i	3,42	< 2	314,87	658
tdd_ir Option -t	1,21	< 2	75,39	549

Abbildung 5.7: Performance-Vergleich zwischen *tdd_ir* und *ir*. Die Zeit ist in Sekunden, der Hauptspeicherbedarf in Megabyte angegeben.

Es stellte sich heraus, dass das Programm *ir* immer schneller war als *tdd_ir*. Muss der Suffixbaum erst noch über die Software *tdd* erstellt werden, ist *ir* bei beiden Sequenzen um ca. *vier* Mal schneller als *tdd_ir*. Bei bereits auf Platte vorhandenem Suffixbaum reduziert sich dieser Faktor jedoch auf etwa *1,5* bei der kleineren Sequenz und geht gegen *1* für *Chromosom 21*. Diese

[2] http://adenine.biz.fh-weihenstephan.de/homePage/ir

Variante scheint, soweit das mit den Testsequenzen beurteilbar ist, einen nahezu linearen Anstieg des Zeitbedarfs zu haben, während bei zusätzlicher Baumkonstruktion aber auch beim Programm *ir* eine höhere Komplexität vermuten lässt. Dies bestätigt aber nur die Angaben in [TTHP05] und [MF04] für die zugrunde liegenden Algorithmen *TDD* und *deep-shallow*.

Der Speicherbedarf für die Mycoplasma Sequenz ist bei beiden Methoden kaum messbar, d.h. geringer als *2 MB*. Der *tdd_ir*-Wert ist wieder davon abhängig, ob das Programm *tdd* benötigt wird. Falls ja, bewegt sich der Bedarf bei ca. 658 MB. Wenn nicht, werden nur etwa 549 MB benötigt. Der Wert für *ir* liegt mit ungefähr 586 MB dazwischen.

Das Programm *ir* benötigt derzeit ca. *1 GB* Hauptspeicher für ca. *80 Mb* Sequenzdaten[3]. Dies entspricht auch der Beobachtung im Test auf *Maschine 2* mit *1 GB* RAM, wo noch *Chromosom 21*, aber keine größeren mehr analysiert werden konnten. Für *Chromosom 1* (doppelsträngig) werden demzufolge ca. *5 GB* benötigt. Dies wurde im Experiment durch die I_r-Autoren auf einer 64-Bit-Maschine mit 64 MB Hauptspeicher auch bestätigt. Das gesamte Humangenom konnte damit aber auch nicht in einem Schritt analysiert werden, denn schon theoretisch würden dazu hochgerechnet ungefähr *75 GB* Hauptspeicher benötigt (*6 GB/80 MB*). Es wurde aber auch festgestellt, dass bereits das einzelsträngige Humangenom nicht mehr verarbeitet werden konnte, obwohl es ja nur die Hälfte an RAM benötigen würde. Dies liegt nach Auskunft der 'deep-shallow'-Entwickler[4] daran, dass für ihre Methode eine Grenze von 2^{30} Byte also ca. *1 GB* an verarbeitbaren Zeichen besteht.

Das Programm *tdd_ir* konnte leider nicht auf einer 64-Bit-Maschine getestet werden. Solange aber für die Baumkonstruktion unter *tdd* sowieso noch mehr Hauptspeicherplatz benötigt wird als bei *ir* und die Partitionierung der Bäume nicht praktisch angewandt werden kann, ist ein weiterer Test auch nicht sinnvoll; v.a. nicht, solange auch noch das Problem mit den nichtsequenzierten Bereichen besteht. Sollten diese Probleme tatsächlich wie angekündigt[5] in der nächsten Version behoben werden können, wäre noch interessant herauszufinden, ob und wie mit der *ReadOnlyBufferCache*-Struktur von *tdd* auch partitionierte Bäume ausgelesen werden können, so dass zumindest die theoretische zur Verfügung stehende Hauptspeichermenge einer 32-Bit Maschine ($= 4GB$) dafür ausreicht.

Bis dahin ist allerdings das Programm *ir* von *Haubold* und *Wiehe* [HW06] das Mittel der Wahl zur Berechnung des *Index of Repetitivenss*. Die Daten die im nächsten Kapitel ausgewertet werden, stammen auch von dieser Software.

[3] Abschätzung der Entwickler
[4] persönlicher Kontakt
[5] persönlicher Kontakt

Kapitel 6

Zusammenhang von Genfunktion und I_r

Haubold und Wiehe beobachteten im Experiment, dass sich innerhalb des HOXA-Clusters auf Chromosom 7 alle Regionen mit niedrigem I_r-Wert mit Genen überschneiden [HW06]. Dies entspricht genau der Charakteristik der HOX-Gene, die nur sehr wenig repetitive Bereiche enthalten [LLB$^+$01]. Ihre Aufgabe besteht in der Kodierung von Transkriptionsfaktoren, weshalb die Vermutung nahe lag, dass auch weitere Genombereiche mit niedrigem I_r ein Indikator für bestimmte Funktionen sein könnten. Dies führt wiederum zu einem der fundamentalen Ziele der Genomik, nämlich der Charakterisierung von Proteinen bzw. Funktionszuweisung an chromosomale Regionen. Neben den Expressionsdatenbanken gibt es immer mehr Annotationsdatenbanken, in denen die Ontologie der Gene zusammengefasst wird [ADD$^+$01]. Zur Auswertung von Genombereichen mit niedrigem *Index of Repetitiveness* wird zunächst über eine Expressionsdatenbank eine Suche nach annotierten Genen gestartet, die ebenfalls in den betreffenden Regionen liegen. Im Anschluss daran werden die Gen-Identifikatoren in die Annotationsdatenbank PANTHER gespeist und in Bezug auf molekulare Funktionen und biologische Prozesse analysiert. Die Signifikanz der Ergebnisse wird dann noch über eine im Vergleich zu der PANTHER-Methode modifizierten Binomialstatistik berechnet.

Im Folgenden wird nochmals detailierter auf die Beobachtungen eingegangen, die dahinter stecken, dass zur Auswertung Bereiche mit *niedrigem* I_r herangezogen wurden und welche teilweise nicht unerhebliche Vorarbeit geleistet werden musste, um den erhaltenen Daten letztendlich eine Funktion zuordnen zu können.

6.1 Auswertung mittels I_r

Im Kapitel 4.3 wurde der *Index of Repetitiveness* vorgestellt, mit dem man je nach Größe die Repetitivität für ganze Genome oder einzelne Chromosomen berechnen kann. In [HW06] gibt es dazu eine Vielzahl von Ergebnissen, für diese Arbeit wurde aber weiter nur das Humangenom betrachtet.

Nachdem der I_r für alle Chromosomen berechnet wurde, ist auch eine gleitende Fensteranalyse durchgeführt worden, die die Zusammensetzung der repetitiven Elemente innerhalb der einzelnen Chromosomen darstellt. Dabei wurde ein erster Hauptaugenmerk auf HOX-Gene (HOX steht für Homeobox) gelegt, die für Transkriptionsfaktoren kodieren, die wiederum bei allen Tieren die Entwicklung durch Regulation anderer Gene steuern. Die Komplexität dieser Vorgänge lässt möglicherweise Vermutungen zu, warum die vier jeweils ca. 100 kb großen HOX-Cluster (HOXA, HOXB, HOXC und HOXD) die Regionen des Humangenoms sind, die die wenigsten 'Repeats' bzw. die geringste Dichte an Tronsposonen[1] haben [LLB+01].

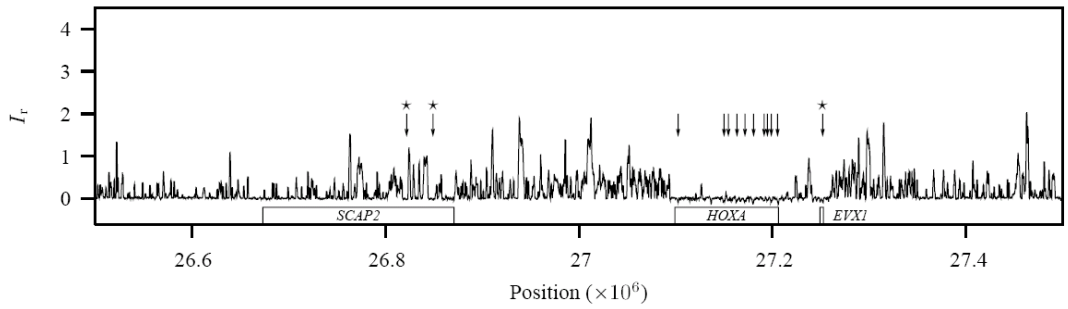


Abbildung 6.1: Eine gleitende Fensteranalyse im Bereich der HOX-Gene auf Chromosom 7. Die Pfeile zeigen Regionen mit niedrigem I_r an, die sich alle mit Genen überschneiden. Aus [HW06].

In Abbildung 6.1 sieht man nun die in [HW06] veröffentlichte 1 MB große Umgebung des HOXA-Clusters auf Chromosom 7. Auffallend sind schon auf dem ersten Blick die niedrigen I_r-Werte, die mit der Lokalisation des HOXA-Clusters übereinstimmen. Um diese Beobachtung zu festigen, wurde dieser Ausschnitt aus Chromomosm 7 nach mindestens 2 kb langen Bereichen untersucht, die einen I_r kleiner 0 habe ($I_r < 0$). Die 13 entdeckten Regionen sind

[1]Transposons (umgangssprachlich 'springende Gene') sind DNA-Abschnitte, die sich replizieren und ihre Lokalisierung im Genom ändern können. Sie sind beiderseits begrenzt durch eine kleinere, gegenläufig-identischen, nicht informativen Nukleotidsequenz, die auch Insertionssequenz genannt wird. Sie kodieren oft für das Enzym Transposase, welches eine wichtige Rolle im Vermehrungsmechanismus des Transposons selber spielt [TRA].

als Pfeile eingezeichnet. Nochmals zur Wiederholung, es entstehen I_r-Werte kleiner gleich 0, wenn weniger *shortest unique substrings* in einem Bereich beobachtet werden, als erwartet worden sind. Zehn der negativen I_r-Bereiche wurden innerhalb des HOXA-Clusters lokalisiert und überschnitten sich jeweils mit dort ansässigen Genen. Die anderen drei durch zusätzliche Sterne markierten Regionen überschnitten sich ebenfalls mit Genen und zwar mit SCAP2 (Signaltransduktionsgen zur T-Zell-Aktivierung) und EVX1 (Gen, das in die Augenentwicklung involviert ist; näheres siehe [HW06]).

Die deutliche Ansammlung von niedrigen I_r-Bereichen führte zu der in [HW06] geäußerten Vermutung, dass in Säugergenomen Regionen von niedrigem I_r durch starke Selektion gegenüber Mutagenese durch Insertionssequenzen stammen könnten. Wenn das der Fall wäre, könnte die Überprüfung von Genomen auf weitere Intervalle, die durch niedrige I_r-Werte gekennzeichnet sind, Gebiete aufdecken, die unter stark reinigender Selektion stehen, d.h. empfindlich auf Transposonen-Mutagenese sind.

Dazu wurden nun als erstes über eine Fensteranalyse Bereiche mit niedrigem I_r identifiziert und dann die Gene festgestellt, die sich mit diesen Intervallen überschneiden. Zum Schluss wurden diese mit Einträgen in einer Annotationsdatenbank verglichen, um herauszufinden, welche molekularen Funktionen diese Regionen haben und welche Rollen sie in biologischen Prozessen spielen. Heraus kamen teilweise hochsignifikante Zusammenhänge zwischen I_r und Funktion der Genprodukte.

Im Anschluß wird Schritt für Schritt gezeigt, wie man von den I_r-Werten der Fensteranalyse zu der Zuweisung von Funktionen in der Ontologie-Datenbank PANTHER kommt und abschließend werden die Ergebnisse diskutiert.

6.2 Vorgehen der Entwicklung vom I_r zur Ontologie

Um diese Entwicklung auch praktisch nachvollziehen zu können, wird für die einzelnen Schritte nach theoretischen Hinführungen auch immer der Aufruf von einzelnen Programmen (AWK- und PERL- Skripte) und die erhaltenen Zwischenergebnisse gezeigt.

Die verwendeten Datensätze mit den *Index of Repetitiveness*-Werten, wurden wie in Kapitel 5.6 erwähnt über das Programm *ir* von *Haubold* und *Wiehe*[2] auf einer 64-Bit-Maschine mit 64 GB Hauptspeicher berechnet. Damit konnte auch Chromosom 1 doppelsträngig analysiert werden.

[2] http://adenine.biz.fh-weihenstephan.de/homePage/ir

6.2.1 Bereiche und Gene mit niedrigem I_r identifizieren

Um Bereiche mit niedrigem I_r aufzufinden, werden die für jedes Fenster erhaltenen I_r-Werte der Reihe nach kumuliert. Als Beispiel wurde ein weiterer Ausschnitt aus Chromosom 7 verwendet und zwar das Fragment um den HOXA-Cluster aus Abb. 6.1.

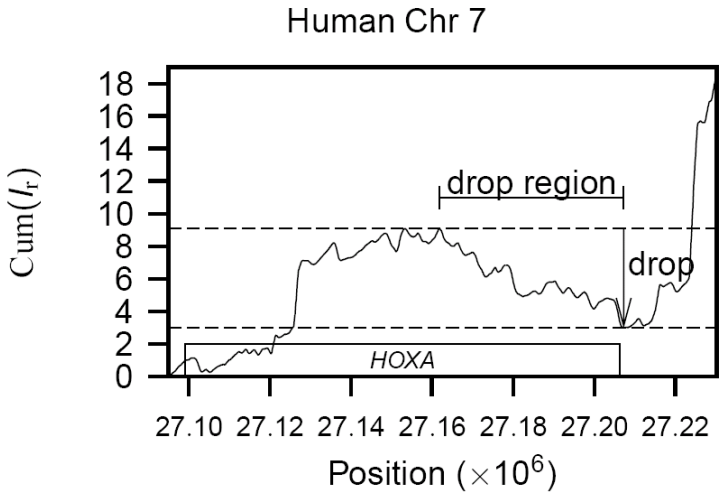


Abbildung 6.2: Niedrige I_r-Bereiche werden über den Abfall des kumulierten I_r-Wertes bestimmt. Von Prof. Dr. Haubold zur Verfügung gestelltes Diagramm.

In Abbildung 6.2 sieht man den resultierenden Graph, der jetzt an den entsprechenden Positionen nicht mehr die jeweiligen Fenster-I_r-Werte, sondern die bis zu diesem Punkt kumulierten I_r-Werte zeigt. Da es für ein Fenster auch vereinzelt negative I_r-Werte geben kann, wird als niedriger I_r-Bereich nur als solcher angesehen, wenn der kumulierte Graph um mehr als 5 Einheiten abfällt. Dazu merkt man sich immer den jeweiligen maximalen kumulierten I_r. Gibt es danach einen Abfall, also erstmal nur noch kleinere kumulierte $Cum(I_r)$-Werte, wird sich das dortige Minimum gemerkt. Dies passiert solange, bis der Graph wieder ansteigt. Wenn der kumulierte Wert den vorher gemerkten erreicht bzw. übersteigt, ist die Prozedur beendet. Dies ist dann der neuen Maximalwert. Nun muss berechnet werden, ob die Differenz aus dem 'alten' Maximum und dem Minimum einen größeren Abfall als 5 Einheiten ergibt. Ist dies der Fall, ist der Bereich zwischen der Position des 'alten' Maximums und der Position des Minimums das Intervall mit negativem I_r,

das weiter verwendet wird. Das neue Maximum ist nun an der Position, ab der das 'alte' wieder erreicht, bzw. überschritten wird. Der Wert für das Minimum wird auf Null gesetzt und die Prozedur beginnt von vorne. Gibt es einen Abfall mit anschließendem Wiederanstieg, bei dem das Minimum nicht um 5 Einheiten kleiner als das Maximum ist, wird dieser Bereich nicht als negativ gewertet, allerdings trotzdem als neues Maximum die Position bezeichnet, an der das 'alte' wieder überschritten wurde.

In Abbildung 6.2 sieht man solche kleinen Täler an den kurzen, nach unten gehenden Zacken. Der richtige negativen I_r-Bereich beginnt kurz nach Position $27,16 * 10^6$ mit einem Maximum von ca. 9 und endet kurz hinter Position $27,20 * 10^6$ bei einem Minimum von ca. 3. Kurz nach Position $27,22 * 10^6$ wird das bisherige Maximum überschritten und der Algorithmus beginnt von Neuem.

Hat man alle Regionen mit einem I_r-Abfall größer als dem Schwellenwert identifiziert, kann mittels der Start- und Endpositionen im Genom nach Genen gesucht werden, die sich damit überschneiden. Hierzu wurde ein Programm erstellt, das die entsprechenden Einträge einer unter PostgreSQL erstellten Datenbank abfragt. Für die Auswertung wurde konkret die NCBI Assembly 36 des Humangenoms verwendet, mit Annotationen, wie sie in der ENSEMBL Datenbank Version 42.36d zusammengefaßt sind.

Ein Eintrag in dieser Datenbank sieht so aus, wie auf der nächsten Seite in gekürzter Form dargestellt (Tabelle 6.1).

Hiervon wurde letztendlich nur die 'gene_stable_id' (ENSG00000184895) und die 'description' (Sex-determining region Y protein (Testis-determining factor)[Source:Uniprot/SWISSPROT;Acc:Q05066]) weiterverwendet, wie man gleich sehen wird. Als allererstes braucht man allerdings die I_r-Werte für die einzelnen Fenster. Diese wurden wie schon erwähnt über das Programm *ir* ermittelt und sind für jedes Chromosom in einer Datei abgespeichert. Für Chromosom 1 sieht das z.B. in einem File namens chr_1.1k.ir folgendermaßen aus:

```
>1 dna:chromosome chromosome:N...
    500.5   0.5698
    600.5   0.4018
    700.5   0.2885
    800.5   0.2081
    ...
```

In der ersten Spalte stehen die Mittelpunkte der Fenster und in der zweiten die zugehörigen I_r-Werte.

Es folgt nun eine Reihe von Programmaufrufen, die alle in einer Datei

Tabelle 6.1: Abgekürzter ENSEMBL-Eintrag der Attribute und Werte für das Gen ENSG00000184895.

Attribut	*Wert*
gene_id_key	51907
gene_stable_id	ENSG00000184895
gene_stable_id_v	ENSG00000184895.4
biotype	protein_coding
source	ensembl
confidence	KNOWN
display_xref_id	24005498
gene_chrom_start	2714896
gene_chrom_end	2715740
chrom_strand	-1
chromosme_id	226031
chr_name	Y
description	Sex-determining region Y protein (Testis-determing factor). [Source:Uniprot/Swissprot;Acc:Q05066]
band	p11.31
known_gene	51907
display_id	SRY
db_name	HUGO
has_5utr_bool	1
ha3_5utr_bool	1
transcript_count	2
aaegypti_homolog_bool	\N
agambiae_homolog_bool	\N
btaurus_homolog_bool	\N
celegans_homolog_bool	1
cfamiliaris_homolog_bool	\N
cintstinalis_homolog_bool	\N
csavignyi_homolog_bool	1
dmelanogaster_homolog_bool	\N
...	...
perc_gc	46
freq	1

namens driver.scr zusammengefaßt wurden:

```
mindrop=5;
for a in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 X Y
do
    cat ../Data36.42/chr_$a.1k.ir | perl -pe 's/\./,/g'
    >../Data36.42/chr_$a.1k.ir.a
    ./scanir.awk $mindrop $a ../Data36.42/chr_$a.1k.ir.a |
    awk -f geneFinder.awk
done
```

Man sieht hier, dass für einen Schwellenwert von 5 jedes Chromosom analysiert wird. Über ein Perl-Skript wird erst Punkt durch Komma ersetzt, wenn das durch die Spracheinstellung der verwendeteten Maschine notwendig ist. Danach dient die neu gewonnene Datei als Input für das Programm scanir.awk. Dieses ist ein AWK-Programm, das den Algorithmus zur Berechnung niedriger I_r-Bereiche nach obiger Definition (Abschnitt 6.2.1) enthält. Dabei kann auch ein Schwellenwert übergeben bzw. festgelegt werden. Im nachfolgenden Beispiel wird als Schwellenwert 5, die Chromosomennummer 21 und die zugehörige Datei mit den I_r-Werten der Fensteranalyse übergeben.

```
./scanir.awk 5 21 ../Data36.42/chr_21.1k.ir.test

    21 36985201 37003501 : 38488,9 38481,5 : 7,3871
    21 46360201 46377301 : 53196,6 53191,3 : 5,3944
    ...
```

Was man als Ausgabe erhält, ist von links nach rechts gelesen wiederum die Chromosomen-ID, Anfang und Ende des I_r-Intervalls, der darin auftauchende höchste und niedrigste kumulierte I_r-Wert, sowie die Differenz aus beiden, was dem Abfall entspricht (hier: 7,3871). Mit diesen Werten geht man nun in die Datenbank und sucht aufgrund der in 4.2.1 erwähnten Komplementarität von *shortest unique substrings*, die dazu führt, dass komplementäre Stränge denselben I_r-Wert haben, nach Genen sowohl im Vorwärts- als auch im Rückwärtsstrang. Für die Überschneidung werden noch 5 kb für die upstream-Regionen zu den entsprechenden Genkoordinaten hinzuaddiert.

Man erhält also für jedes Chromosom die Bereiche berechnet, in denen es einen Abfall des kumulierten I_r um ≥ 5 ($\equiv$ mindrop 5) gibt. Die wie oben für Chromosom 21 dargestellten Daten des gesamten Humangenoms werden dann Zeile für Zeile an das AWK-Programm *geneFinder.awk* übergeben.

Mittels der Start- und Endpositionen der erhaltenen I_r-Intervalle kann in

einer Datenbank nach Genen gesucht werden, die sich mit diesen Bereichen überschneiden. Dazu wurde in dieser Arbeit die ENSEMBL Datenbank Version 42.36d für das humane Genom lokal unter PostgreSQL installiert. Pro gefundenem Gen erhält man folgende Ausgabe:

```
4
10,2981
2029501
2048601
ENSG00000185818
protein_coding
KNOWN
NAT8L N-acetyltransferase 8-like
[Source:RefSeq_peptide;Acc:NP_848652]
```

In der ersten Zeile steht der Chromosomenname. Es folgt der Wert des maximalen Abfalls des kumulierten I_r. Danach Start- und Endposition des I_r-Intervalls, anschließend die ENSEMBL-ID des Gens. In der zweiten Zeile steht der Biotyp, die 'Confidence' (hier: KNOWN), sowie die Beschreibung bzw. auch sogenannte 'Description'. Am Ende ist schließlich die Quellenangabe ([Source:RefSeq_peptide;Acc:NP_848652]) aufgeführt.

Es soll an dieser Stelle erwähnt werden, dass die bisherigen Programme *scanir*, *geneFinder*, die Skripte zur Datenbankkonstruktion, sowie die Dateien mit den I_r-Werten für das Humangenom, von Wiehe und Haubold [HW06] dankenswerterweise für diese Arbeit überlassen wurden. Die im Anschluss beschriebenen Programme entstammen der eigenen Entwicklung.

Die aus der Datenbank in obigem Format erhaltenen Gene werden in einer Datei abgespeichert. Dies geschieht z.B. unter UNIX-Systemen, indem dem Aufruf von driver.scr die Ausgabedatei folgt, z.B. driver.scr > humanScan_mindrop5.txt

6.2.2 Formatierung der erhaltenen Genidentifikatoren

Bisher wurde versucht, die Ontologien für Regionen mit niedrigem I_r über die Datenbank Gene Ontology (GO) herauszufinden. Dazu reichte es, die ENSEMBL-ID's zu extrahieren z.B. über den Befehl

```
cat humanScan_mindrop5.txt | awk '{print $5}'
```

und die Ontologien dann über eine lokal installierbare Version der GO-Datenbank abzufragen. Es war jedoch die Absicht, die weniger detailiertere, dafür über-

sichtlichere Datenbank PANTHER zu verwenden, die die Ergebnisse zudem schneller zurückliefern sollte (siehe auch nachfolgendes Kapitel 6.2.3).

Für PANTHER konnte leider keine lokal installierbare Version gefunden werden. Die Auswertung erfolgte deshalb über ein Tool zur Genexpressionsanalyse, das als Web-Interface unter http://www.pantherdb.org/tools/compareToRefListForm.jsp erhältlich ist.

Dabei mußte festgestellt werden, dass PANTHER über die ENSEMBL-ID's keine Treffer zurücklieferte. Es wurden deshalb die sogenannten ACC-ID's verwendet, die sich in den Quellenangaben innerhalb der Gen-Descriptons befinden (Z.B. [Source:Uniprot/SWISSPROT;Acc:Q05066]).

Diese zu extrahieren benötigt einen etwas größeren Aufwand. Es wurde dazu das PERL-Skript *go4Acc.src* verwendet, das die Daten aus z.B. humanScan_mindrop5.txt nimmt, Zeile für Zeile die Acc-ID's extrahiert und sie in einer weiteren Datei mit dem zusätzlichen Anhang Acc_output (also hier: humanScan_mindrop5.txt.Acc_output) abspeichert. Der Vorgang ist in Abbildung 6.3 als Quellcodelisting dargestellt. Zunächst wird eine übergebene Zeile aufgespalten. Man sucht nach der Spalte, die die 'Source' bzw. Acc-ID enthält. Diese wird dann 'gesplittet', weil man ja nur die ID und nicht die Quellenangabe möchte. Die ID's selber werden dann in einem Hash als 'Key' abgespeichert und ihr 'Value' entspricht der Anzahl ihres Vorkommens.
Was man dadurch erhält sind z.B. folgende Terme (auszugsweise):

```
MI0000478
MI0001150
NM_000683
NM_000741
NP_071938
NP_112236
O00548
O00570
P04085
P06454
Q04756
Q05066
RF00002
RF00026
...
```

Diesen ID's zugrunde liegen verschiedene Datenbanken, die Zusammenhänge sieht man in der Tabelle 6.2.

Da es nicht zu jeder ENSEMBL-ID eine 'Description' bzw. Quellenan-

```
#!/usr/bin/perl
use strict;
use warnings;

my $file = shift;
my @words;
my %id;
open (FILE, "< $file") || die("Couldn't open $file - $!\n");
open(OUTFILE, "> $file.Acc_output") || die("Couldn't open
$file.Acc_output - $!\n");
while (<FILE>){
    @words=split(/\s+/, $_);
    for(@words){
        if(/Acc:/){
            my @uniprot=split(/;/, $_);
            for(@uniprot){
                if(/Acc:/){
                    $_ =~ s/[\]]//gx;
                    my @Acc=split(/:/, $_);
                    my $Accid = $Acc[-1];
                    if(exists($statistic{$Accid})){
                        $statistic{$Accid}++;
                    }
                    else{
                        $statistic{$Accid}=1;
                    }
                }
            }
        }
    }
}

foreach my $key (sort keys %statistic){
    printf OUTFILE "%-70s\n", $key;
} printf "\n";

close (OUTFILE); close (FILE);
```

Abbildung 6.3: Quelltext zum PERL-Skript go4Acc.src.

Tabelle 6.2: Quelldatenbanken der Gen-ID's.

ID-Präfix	*Datenbank*
MI	miRNA_Registry
NM_	Ref_Seq_dna
NP_	Ref_Seq_peptide
O	Uniprot/Swissprot
P	Uniprot/Swissprot
Q	Uniprot/Swissprot
RF	RFAM

gabe gab, kam es durch die Verwendung der *ACC-ID's* zu einem nicht zu vernachlässigenden Informationsverlust. 232 Regionen mit niedrigem I_r wurden gefunden, davon konnten 222 bekannten Genen zugewiesen werden. Allerdings blieben für die weitere Auswertung nur noch 188 übrig, weil nur sie über ID's verfügten, die über die PANTHER-Datenbank ausgewertet werden konnten. Weil im Rahmen der verfügbaren Zeit keine geeignete Alternative gefunden werden konnte, wurde entschieden, den Prozess trotzdem mit diesen Voraussetzungen fortzuführen.

Wie man im Anschluss gleich noch sehen wird, muss man der zu analysierenden Liste einen Typ zuweisen. PANTHER bietet dazu verschiedene Möglichkeiten. Die höchsten Trefferzahlen lieferte eine Auswahl, die sowohl Gene-ID's als auch Transcript-ID's verarbeiten konnte. Protein-ID's wurden dagegen hier nicht berücksichtigt, aber dafür, wenn der entsprechende spezielle Listentyp ausgewählt wurde. Um die Analyse immer nur einmal durchführen und nicht immer mit zwei verschiedenen Listentypen arbeiten zu müssen, wurden deshalb die Protein-ID's, die aus der *Ref_Seq_peptide*-Datenbank stammen, in *Transcript-ID's* umgewandelt, wie sie in *Ref_Seq_dna* annotiert sind.

Über die NCBI-Homepage gibt es dazu ein 'Flatfile' (ftp://ftp.ncbi.nlm.nih.gov/refseq/release/release-catalog/release23.accession2geneid.gz, dass als Datenbanktabelle verwendet werden und in dem zu den Protein-ID's äquivalente Transcript-ID's gesucht werden können.
Die Einträge in dem File haben folgende Struktur:

```
10090   100012  NM_201258.1     NP_957710.1
10090   100017  NM_145554.1     NP_663529.1
10090   100019  NM_001081392.1  NP_001074861.1
...
```

In der 1. Spalte steht die 'TaxonomicID', in der zweiten die 'EntrezGeneID', es folgt die 3. Spalte mit 'Transcript accession.version' und die vierte mit 'Proteinaccession.version'

Es wurde eine auf PostgreSQL basierende Datenbank erstellt, in der diese Tabelle eingelesen wurde. Dies geschah mit folgendem Skript, das auf der DBMS-Ebene[3] aufgerufen wurde:

```
drop table npnm;
CREATE TABLE npnm (
    taxonomy_id int default NULL,
    entrez_gene_id int default NULL,
    transcript_accession_version varchar(128) NOT NULL default '',
    protein_accession_version varchar(140) default NULL
);
\copy np from release22.accession2geneid
```

Mit dem anschließend aufgeführten Skript transcriptFinder.sql wurde nun die Datenabfrage durchgeführt:

```
{
database = "refseq";
start = $1 cmd = "psql -U username -d " database " -c "
cmd = cmd "\"select transcript_accession_version "
cmd = cmd "from np "
cmd = cmd "where protein_accession_version like '" start "__' "
cmd = cmd "\" | tail +3 | grep -v 'Zeile.*)' | sed 's/|//g'";
n = 0;
output = start "\t";
while(cmd | getline){
    if($1){
        printf($1);
        printf("\n");
        n++;
    }
}
close(cmd);

if(n == 0)
    print output
}
```

Mit dem Aufruf

[3]DBMS: Data Base Management System

```
awk -f transcriptFinder.sql humanScan_mindrop5.txt.Acc_output
    > humanScan_mindrop5.Acc_final
```

loggt das Programm an die Datenbank (hier *refseq* genannt) an[4], und sucht in der Datenbank nach den zu den Protein-ID's äquivalenten Transcript-ID's. Mit dieser Methode konnten alle vorhandenen Protein-ID's transformiert werden. Wie man oben gesehen hat, stehen in dem Flatfile für die Datenbanktabelle nach den jeweiligen ID's auch noch die Versionsnummern, z.B. NM_201258.1, statt nur NM_201258. Nachdem PANTHER mit diesem Format ebenfalls nichts anfangen kann, müssen die Versionsnummern nachträglich entfernt werden, was einfach über den UNIX-Befehl 'sed' geschieht:

```
sed 's/\.[0-9]//g' humanScan_mindrop$a.Acc_final
    > humanScan_mindrop$a.Acc_final.a
```

Man hat nun endlich eine Liste mit so formatierten ID's, dass sie zur PANTHER-Datenbank hochgeladen und analysiert werden können. Im nächsten Abschnitt wird das nähere Vorgehen beschrieben, nachdem ein kurze Einführung in die Ontologie-Datenbank erfolgt ist.

6.2.3 Auswertung mittels der Ontologie-Datenbank PANTHER

6.2.3.1 Die Ontologie-Datenbank PANTHER

Das schnelle Wachstum von Proteinsequenzdatenbanken führt gleichzeitig zu einem fortschreitenden Prozess im Verstehen der Beziehungen zwischen Proteinsequenzen und ihrer Funktion [TCK+03]. Freie Textbeschreibungen dieser Zusammenhänge führen jedoch leicht zu einer unübersichtlichen und unstrukturierten Menge an Information. Man benötigt deshalb am besten ein kontrolliertes, beschreibendes Vokabular, um dieses Problem zu vermeiden. Dazu gibt es neben den Expressionsdatenbanken immer mehr Annotationsdatenbanken, in denen diese sogenannten Ontologien zusammengefasst werden [ADD+01]. Die Ontologien sollen ein kontrolliertes Vokabular definieren und zudem eine umfassende computerunterstützte Analyse erlauben [TCK+03]. PANTHER ist eine solche Datenbank. Sie ist über einen Webbrowser darstellbar[5] und enthält Genprodukte, die nach ihrer biologischen Funktion strukturiert sind. Genauer gesagt, werden die Ontologien in zwei

[4]statt *username* muss natürlich der richtige Benutzername angegeben werden

[5]http://www.pantherdb.org

große Gruppen, nämlich molekulare Funktion ('molecular function') und biologischer Prozess ('biological process') eingeteilt. Dazu wurden die einzelnen Ontologie-Terme von als Kuratoren fungierenden Biologen mehr mit Proteinsequenzgruppen als mit individuellen Sequenzen verknüpft [TKC+06].

Aufgrunddessen hat PANTHER einen grundlegend hierarchischen Aufbau. Im Detail handelt es sich um einen azyklischen Graphen, bei dem Kindkategorien unter mehr als einem Elternteil erscheinen können, wenn es biologisch gerechtfertigt ist [TKC+06]. Dieser Graph ist allerdings nicht mehr als drei Ebenen tief und enthält pro Kategorie nicht mehr als ca. 250 Einträge. Damit ist PANTHER etwas anders gestaltet, als die weit verbreitete, umfangreichere 'Gene Ontology (GO)'-Datenbank[6]. Die GO-Struktur beinhaltet über 7000 bzw. ca. 5000 Terme, um molekulare Funktionen und biologische Prozesse zu beschreiben und ist bis zu zwölf Ebenen tief [TCK+03]. Dies liefert natürlich ein größeres Vokabular für die funktionelle Annotation von Genprodukten, aber es gibt wissenschaftliche Anwendungen, die von einer einfacheren Ontologie profitieren würden [TCK+03]. Aus diesem Grund wurde PANTHER etwas anders strukturiert, um die Navigation zu erleichtern und um Analysen von großen Gen- bzw. Protein-Datensätzen durchführen zu können [TKC+06]. Während die GO-Datenbank komplett auf einem lokalen Rechner herunterladbar und installierbar ist[7], bietet PANTHER ein Webinterface, um z.B. Genlisten zur Webseite hochzuladen und sie in Bezug auf molekulare Funktion und biologischen Prozess zu analysieren[8]. Außerdem bietet PANTHER auch eine statistische Signifikanzanalyse der hochgeladenen Gene in Bezug auf eine Referenzliste z.B. 'NCBI: H. sapiens genes' für das Humangenom. Diese und weitere Funktionalitäten der 'Website' sind in [MULK+03] aufgeführt. Eine umfassende Einführung und Beschreibung der PANTHER-Datenbank findet man in den oben zitierten Arbeiten von *Thomas et al* genauso wie einen ausführlicheren Vergleich zwischen GO und PANTHER.

Im nächsten Abschnitt soll anhand von Screenshots erläutert werden, wie man eine durch die Schritte in den vorherigen Abschnitten erhaltene Genliste über das PANTHER-Webinterface analysiert und welche Struktur die zurückgelieferten Daten haben.

Die zu verarbeitenden Genidentifikatoren stehen also als einspaltige Tabelle in einer Datei z.B. namens humanScan_mindrop5.Acc_final.a und können hochgeladen werden. Anhand der Screenshots in Abbildung 6.4 wird das genaue Vorgehen nachvollzogen.

[6] http://www.geneontology.org/

[7] http://www.geneontology.org/GO.downloads.database.shtml

[8] http://www.pantherdb.org/tools/compareToRefListForm.jsp

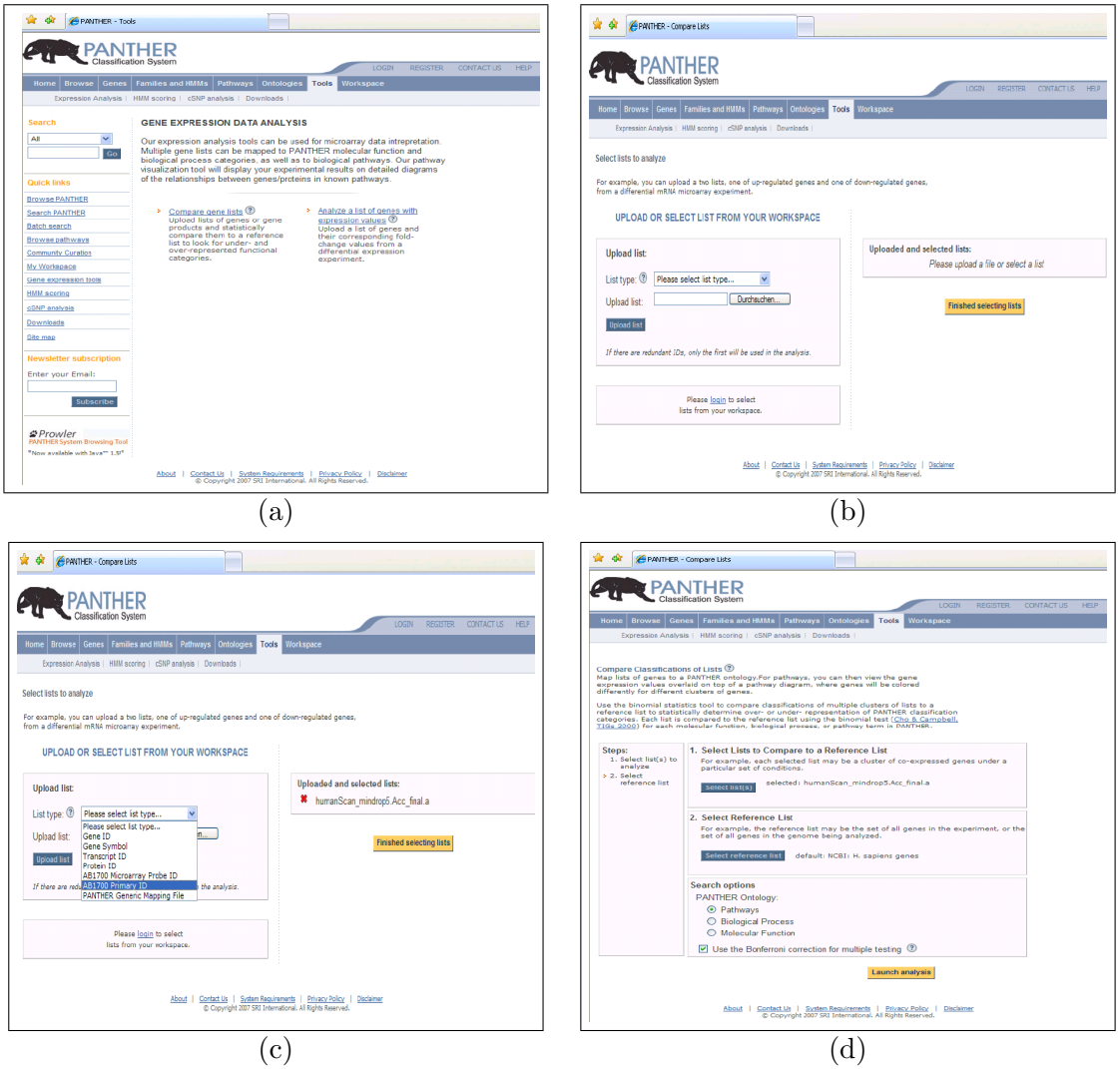
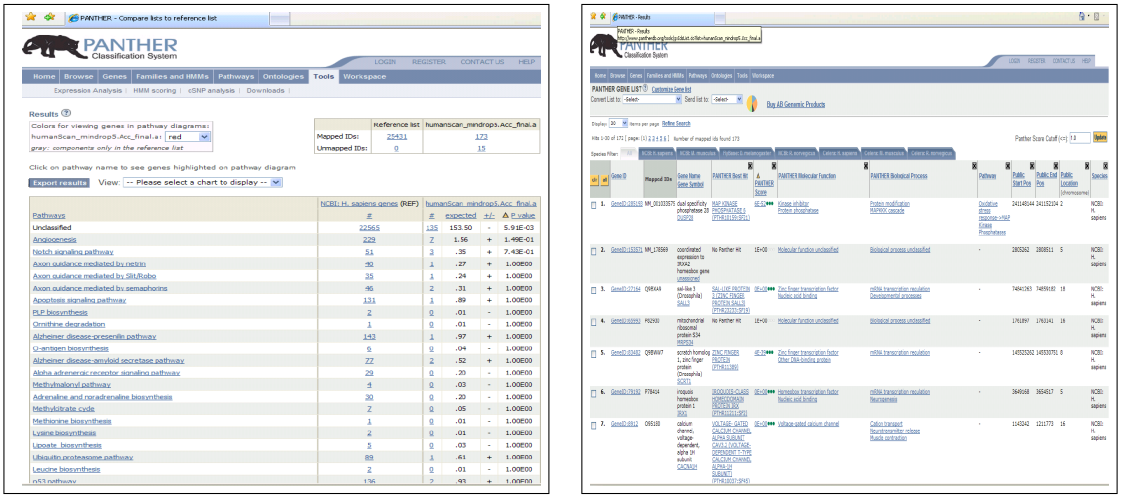


Abbildung 6.4: Vorgehen der Auswertung von Genlisten über das PANTHER-Tool 'Gene Expression Data Analysis'.

Der Aufruf des Webinterfaces erfolgt über http://www.pantherdb.org/tools/. Dort findet man drei sogenannte 'Research Tools', nämlich 'Gene Expression Data Analysis', 'Panther scoring' und 'Evolutionary analysis of coding SNPs'. Für die Zwecke dieser Arbeit wird nur das erste Tool verwendet. Wählt man dieses aus, gelangt man auf eine weiter Seite, die GENE EXPRESSION DATA ANALYSIS heißt (Abb. 6.4 a). Dort muss man 'Compare gene lists' auswählen, weil 'Analyze a list of genes with expression values' noch Werte für die Faltung verlangen würde. Im nächsten Fenster klickt man einfach auf den einzelnen Auswahlbutton 'Select list(s)' und kommt dann zur Ansicht wie in Abbildung 6.4 b. Dort muss man über die 'Combo-Box' 'List type' einen von sieben Typen für die hochzuladende Liste auswählen. Wie vorher schon erwähnt, bringt für die erhaltenen Beispielliste die Auswahl 'AB1700 Primary ID' die meisten Treffer, weil PANTHER damit sowohl Gen-ID's als auch Transcript-ID's erkennt. Über einen Klick auf den 'Durchsuchen'-Button gelangt man in einen Verzeichnisbaum und kann die entsprechende Datei auf seinem Computer auswählen. Als Ergebnis erhält man Bild 6.4 c und kann die Auswahl nun mit 'Finished selecting lists' abschließen. Man wechselt damit in das nächste Fenster (Abb. 6.4 d), wo man unter 'Select reference list' eine andere als die angegebene Referenzliste auswählen kann. Hier wurde die vorgeschlagene NCBI-Liste beibehalten. Ebenso die Bonferroni-Korrektur bei den 'Search options'. Dort muss nur eingestellt werden, welche Ontologie man zurückgeliefert bekommen will. Für die Zwecke dieser Arbeit wurde entweder 'Biological Process' oder 'Molecular Function' ausgewählt. D.h. man erhält pro hochgeladener Datei zwei Datensätze zurück, nämlich eine, die die Ontologien für die molekulare Funktion enthält und eine für den biologischen Prozess. Wie man diese wieder auf den lokalen Rechner bringt, wird in der nächsten Screenshot-Kaskade gezeigt.

Als Ausgabe erhält man eine Darstellung wie in Abbildung 6.5 (a). Oben rechts sieht man die 'Mapped IDs' für die Referenzliste, wo natürlich alle Gene getroffen wurden und für den hochgeladenen Datensatz. Wie man hier sieht, werden von den 188 übergebenen Genidentifikatoren immer noch 15 nicht erkannt. Dies sind die Einträge aus der 'miRNA-Registry'-, der 'RFAM'- und einige der mit 'Q' beginnenden Terme aus der 'Uniport/Swissprot'-Datenbank. Klickt man auf eine der Zahlen für Mapped/Unmapped ID's erhält man entweder die zugehörigen Gene und Ontologien im Detail (Abbildung 6.5 (b)) oder eine Tabelle der nichterkannten ID's. Über einen Klick auf den Button 'Export results' kann man die dargestellte Tabelle als Datei im Textformat herunterladen.



(a) (b)

Abbildung 6.5: Über das PANTHER-Tool 'Gene Expression Data Analysis' erhaltenen Daten zur Auswertung von Genlisten.

Man erhält damit die gleiche Tabelle wie im Webinterface, die nun für die molekulare Funktion so aussieht:

Molecular function unclassified	10934	36	74.38	- 2.01E-08
Transcription factor	2052	39	13.96	+ 1.09E-07
Homeobox transcription factor	249	13	1.69	+ 3.59E-06
Kinase	684	16	4.65	+ 6.04E-04
...				

und nach Auswahl von biologischen Prozessen so:

Neurogenesis	587	26	3.99	+ 1.08E-11
Ectoderm development	692	26	4.71	+ 3.15E-10
Developmental processes	2152	42	14.64	+ 1.12E-08
mRNA transcription regulation	1459	34	9.93	+ 5.73E-08
Biological process unclassified	11321	40	77.01	- 1.22E-07
mRNA transcription	1914	37	13.02	+ 9.68E-07
...				

Die erste Spalte enthält den Namen der PANTHER Klassifikationskategorie. Die zweite Spalte enthält die Anzahl der Gene in der Referenzliste, die zu dieser bestimmten Kategorie gehören. Die dritte Spalte enthält die Anzahl der Gene in der hochgeladenen Liste, die zu dieser PANTHER Klassifikati-

onskategorie gehören.

Die vierte Spalte enthält den Erwartungswert, der die Anzahl der Gene wiedergibt, die aufgrund der Referenzliste für diese Kategorie erwartet worden wären.

Die fünfte Spalte enthält entweder ein plus oder minus '-' und spiegelt wieder, ob mehr oder weniger Gene als erwartet getroffen wurden.

Die sechste Spalte enthält den p-value, der durch Binomialstatistik bestimmt wird und die Wahrscheinlichkeit wiederspiegelt, dass die Anzahl der in einer Kategorie beobachteten Gene per Zufall, basierend auf der Referenzliste, beobachtet werden sein könnte (nach http://www.pantherdb.org/tips/tips_binomial.jsp).

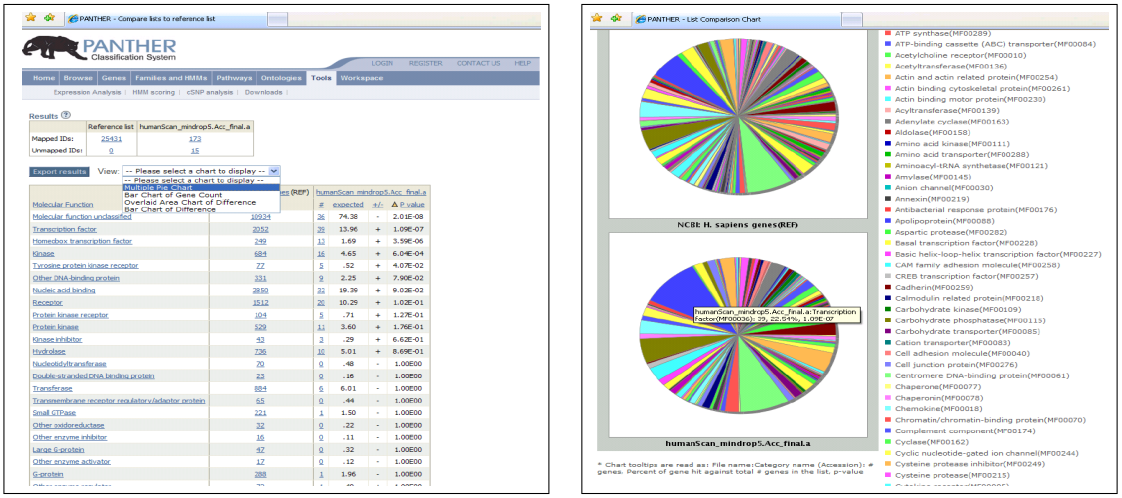
Mit einem Perl-Skript namens *go4panther_2datasets_palue.src* werden die getrennten Dateien für molekulare Funktion und biologischen Prozess zusammengefasst und die Ontologien nach ihrem p-value sortiert aufgelistet. Man erhält folgende Ausgabe:

```
Neurogenesis                          => 1.080000E-11 =>  26 =>   3.99
Ectoderm development                  => 3.150000E-10 =>  26 =>   4.71
Developmental processes               => 1.120000E-08 =>  42 =>  14.64
Molecular function unclassified       => 2.010000E-08 =>  36 =>  74.38
mRNA transcription regulation         => 5.730000E-08 =>  34 =>   9.93
Transcription factor                  => 1.090000E-07 =>  39 =>  13.96
Biological process unclassified       => 1.220000E-07 =>  40 =>  77.01
mRNA transcription                    => 9.680000E-07 =>  37 =>  13.02
Homeobox transcription factor         => 3.590000E-06 =>  13 =>   1.69
Nucleoside, nucleotide and nucleic
acid metabolism                       => 5.650000E-05 =>  46 =>  22.74
...
```

mit den Spalten Ontologie, p-value, absolute Häufigkeit und erwartete Häufigkeit.

PANTHER bietet auch noch eine Funktion, die berechneten Daten für jede Kategorie zu visualisieren. In Abbildung 6.6 (a) sieht man die Option 'View' als Auswahlmöglichkeit.

Dort kann man sich z.B. einen 'Multiple Pie Chart' der Ergebnisse ansehen. Einzelne Daten für Ontologien erhält man, wenn man mit dem Mauszeiger darüberfährt. Die Darstellung wird genauso wie beim 'Bar-chart' über den relativen Anteil der einer Ontologie zugeordneten Gene an der Gesamtzahl der übergebenen Gene bestimmt. D.h. z.B. sind der molekularen Funktion 'Transcription' 39 der 173 in PANTHER getroffenen Gene zugeordnet, also ca. 22,5 Prozent ($39/173 = 0.2254335$). Es wird auch, wie vorhin kurz



(a) (b)

Abbildung 6.6: Die über PANTHER erhaltenen Daten lassen sich über ein weiteres Tool visualisieren.

erwähnt, ein p-value berechnet, der allerdings auf der Nullhypothese begründet ist, dass die Wahrscheinlichkeit, ein Gen der hochgeladenen Liste in einer bestimmten Kategorie C anzutreffen, die gleiche ist, wie die entsprechende Wahrscheinlichkeit der Referenzliste. D.h. p(C)=n(C)/N, wobei n(C) die Anzahl der Gene in Kategorie C und N die Anzahl aller Gene in der Referenzliste ist. (s.a. http://www.pantherdb.org/tips/tips_binomial.jsp#P-Value-calculated)

Unter der obigen Nullhypothese, kann nun unter Annahme einer binomialen Verteilung die Wahrscheinlichkeit berechnet werden, $k(C)$ Gene (oder mehr) in einer hochgeladenen Liste der Größe K zu beobachten. Die Wahrscheinlichkeit, $k(c)$ Gene alleine durch Zufall zu beobachten ist:

$$p\text{-value} = \sum \binom{K}{k} p(c)^k (1 - p(c))^{K-k}$$

Dies ist auch der p-value, der über das Webinterface zurückgeliefert wird und auch in den Files vorhanden ist, die abschließend herunterladbar sind.

Dieser p-value ist aber nicht der Weisheit letzter Schluß. Hier wird nämlich von einer Gleichverteilung der Gene ausgegangen, d.h. man nimmt an, dass alle Gene mit gleicher Wahrscheinlichkeit zufällig getroffen werden können. Diese Annahme trifft aber für unseren Fall nicht zu, weil man ja zufällig Fragmente mit niedrigem I_r auf das Genom wirft und danach fragt, wie wahrscheinlich es ist, dass man x-Mal Gene trifft, die aus einer bestimmten

Klassifikationskategorie stammen. D.h. man muss noch die Zahl der Basenpaare berücksichtigen, die von den Genen der einzelnen Kategorien und den Bereichen mit niedrigem I_r überdeckt werden.

Entsprechend berechnet sich der p-value damit folgendermaßen. Das humane Genom enthielt 232 Regionen mit niedrigem I_r, die insgesamt 2147000 bp überdeckten. 222 davon überschnitten sich mit bekannten Genen. Um die Nullhypothese zu testen, dass die 222 Gene zufällig getroffen werden, betrachte man die Wahrscheinlichkeit p, dass ein einzelnes Gen zufällig getroffen wird. Gegeben ist, dass 31127 bekannte Gene 1127021 bp von 3093120360 bp des Humangenoms bedecken. Dazu addiert man noch 5 kb für die *upstream region* eines jeden Gens. Man erhält folgende Wahrscheinlichkeit:

$$p(c) = \frac{1137021 + 31127 * 5000 + 2147100}{3093120360} \approx 0,42.$$

Die Wahrscheinlichkeit mehr als 222 Gene zu treffen ist:

$$P = \sum_{k=222}^{2} 32 \binom{232}{k} p(c)^k (1 - p(c))^{232-k} \approx 5,13 * 10^{-70}.$$

Für die einzelnen Kategorien müssen die Wahrscheinlichkeiten entsprechend angepaßt werden. Für p(c) nimmt man statt 31127 die Anzahl der Gene der entsprechenden Kategorie her und statt 1137021 die Länge der von diesen Genen überdeckten Sequenz im Genom.

6.2.3.2 Nachbearbeitung der erhaltenen Daten

Man muss also für jede Kategorie die Gene herausfinden und ihre Überdeckung berechnen. Dazu wurde am Ende der Auswertungen entdeckt, dass PANTHER eine Möglichkeit bietet, alle enthaltenen Gene mitsamt ihrer Start- und Endposition, der Chromosomenlokalisation und den zugehörigen Ontologien darzustellen.

Wenn man die Schritte zur PANTHER-Auswertung wie oben beschrieben befolgt, führt dies zur Darstellung der Ontologien, wie in Abbildung 6.6 (a). Man muss nun die Zahl der Mapped-Id's der Referenzliste anklicken und gelangt auf den Bildschirm 6.7 (a). Über ein Auswahlmenü, dass durch einen Klick auf 'Customize Gene list' öffnet, lassen sich die angezeigten Elemente anpassen, lediglich Gene_ID und Gene Name/Gene Symbol müssen erhalten bleiben 6.7 (b). Man kann auch die Reihenfolge ändern, so dass hier gewählt wurde: Gene ID, Gene Name/Gene Symbol, Public Start Pos, Public End Pos, PANTHER Molecular Function und PANTHER Biological Process.

Einzige Unsicherheit ist das Vorkommen von Differenzen zwischen einigen Genpositionen in PANTHER und der referierten ENSEMBL-Sequenz.

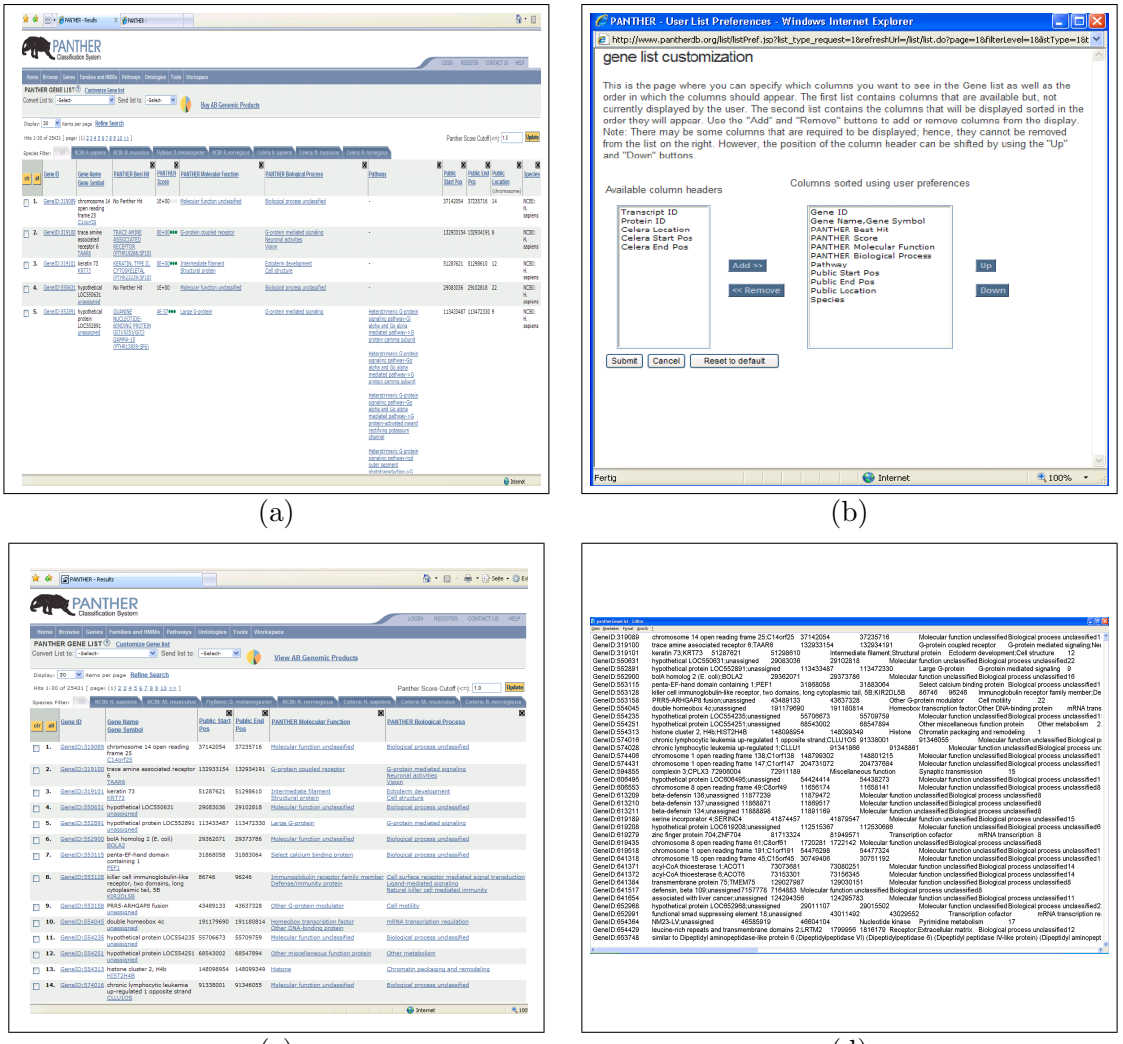


Abbildung 6.7: Über die richtige Auswahl lassen sich die erhaltenen PANTHER-Daten auf die für die weitere Verarbeitung wichtigen Parameter eingrenzen und herunterladen.

D.h. für manche Gene unterscheiden sich die Start- und Endposition, obwohl einem Gen der PANTHER-Datenbank ein ENSEMBL-Eintrag als Quelle diente. Näheres hierzu siehe auch Abschnitt 6.4 am Ende dieses Kapitels.

Dennoch wurde diese Tabelle (Abbildung 6.7 (c)) verwendet, da bei stichprobenartigen Tests nur zwei Gene gefunden wurden, für die diese Beobachtungen zutrafen. Die vorhin erwähnten Unterschiede müssten aber für die Zukunft noch näher untersucht werden. Die Tabelle ist durch Auswahl der Option 'Send list to: File' komplett herunterladbar (Abb. 6.7 (d)).

Mit einem weiteren Perl-Skript wurden nun für jede Ontologie die Längen der dieser zugeordneten Gene aufsummiert, um den Genomanteil zu erhalten, der von einer Ontologie überdeckt wird. Gleichzeitig wurden die Gene gezählt, die zu einer bestimmten Ontologie gehörten.

Damit lassen sich entsprechend der im letzten Abschnitt erläuterten Methode die modifizierten p-values für molekulare Funktion und biologischen Prozess ausrechnen. Nachfolgend ist wieder ein Teil des Ergebnisses aufgeführt. In einer Datei können nun in einer Spalte die Ontologieterme und in einer zweiten die zugehörigen p-values als Tabelle abgelegt werden. Um letztendlich die Signifikanz berechnen zu können, mit der Gene eine bestimmte Ontologie treffen oder nicht, benötigt man nun noch die Anzahl der einer bestimmten Ontologie zugeordneten Gene innerhalb der durch das I_r-Auswerteverfahren erhaltenen Genliste.

Diese hat man auch schon und zwar wie oben aufgeführt durch das Perl-Skript go4panther_2datasets_palue.src erhalten. Jetzt muss nur noch der p-value für eine bestimmte Kategorie mit der Häufigkeit dieser Kategorie bzgl. einer Genliste verknüpft werden, was dadurch erfolgte, dass die p-values und zugehörige Ontologienamen in einer Datenbanktabelle abgelegt wurden und durch eine Abfrage über die Ontologienamen der Genliste die drei gewünschten Werte zurückgeliefert wurden.

In der entsprechenden Ausgabedatei steht in der ersten Spalte die Ontologie, in der zweiten die Häufigkeit dieser Ontologie für die Gene des Testdatensatzes und in der dritten Spalte der p-value für diese Ontologie, bezogen auf das komplette Humangenom.

Natürlich hat man jetzt noch nicht den p-value bzw. die Wahrscheinlichkeit, die im Testdatensatz vorhandene Menge an Genen einer bestimmten Kategorie oder mehr zu finden.

Im nächsten Abschnitt wird nun der abschließende Schritt beschrieben, wie über das Statistikprogramm R dieser p-value berechnet wird und das endgültige Ergebnis vorgestellt.

6.2.3.3 Signifikanzberechnung mit R und Ergebnisse

Das Statistikprogramm R ist ein OpenSource-Projekt und unter http://www.r-project.org/ erhältlich. Mittlerweile gibt es zahlreiche Tutorien, auch zu einzelnen Paketen.

Die im vorherigen Abschnitt zuletzt erhaltene Tabelle mit Ontologiename, Anzahl der dazu gehörigen Gene im Testdatensatz und modifizierten allgemeinen p-value für diese Ontologie, ist z.B. in einer Datei namens *human_mindrop5_ontology.dat* abgespeichert.

In der R-Umgebung läßt sich eine solche Datei mit dem Befehl 'read.table' in einer Variablen speichern. Der Aufruf hierzu sieht folgendermaßen aus:

```
> IR<-read.table("human_mindrop5_IRpvalue",sep="\t",as.is=c(1))
```

as.is=c(1) gibt an, dass in der ersten Spalte der Tabelle Zeichenfolgen und keine Zahlen stehen. In der Variablen IR steht also die Tabelle:

```
> IR
      V1                                V2      V3
    1 Neurogenesis                      26      0.021788
    2 Ectoderm development              26      0.002223
    3 Developmental processes           42      0.012981
    4 Molecular function unclassified   36      0.127908
    5 mRNA transcription regulation     34      0.024524
    6 Transcription factor              39      0.005551
    7 Biological process unclassified   40      0.135986
    ...
```

Nun kann man die Signifikanzen bzw. p-values für den Testdatensatz berechnen. Man verwendet dazu den R-Befehl:

```
> for(i in 1:length(IR[[1]]))
  {IRtemp[i]<-print(sum(dbinom(IR[i,2]:232,232,bidaten2[i,3])))}

    [1] 1.472666e-11
    [1] 1.241211e-35
    [1] 1.556178e-34
    [1] 0.1273316
    [1] 0.068920e-16
    [1] 1.120125e-44
    [1] 0.06727227
    ...
```

In die einspaltige Variable IRtemp sind somit die p-values pro Ontologie enthalten und zwar in der gleichen Reihenfolge, wie diese in der Variablen IR aufgeführt sind.

```
> IRpvalue<-data.frame(IR[[1]], IR[[2]], IRtemp)
> IRpvalue

     IR..1..                           IR..2..     IRtemp
   1 Neurogenesis                      26          1.472666e-11
   2 Ectoderm development              26          1.241211e-35
   3 Developmental processes           42          1.556178e-34
   4 Molecular function unclassified   36          1.273316e-01
   5 mRNA transcription regulation     34          1.068920e-16
   ...
```

In die Variable IRpvalue werden nun die Spalten aus der Variablen IR und die p-values aus IRtemp eingefügt, es entsteht die obige Tabelle, die jetzt die p-values für das entsprechende Ontologievorkommen im Testdatensatz statt der allgemeinen p-values wie am Anfang enthält. Über den Befehl

```
> write.table(IRpvalue, file = "IRpvalue.out", sep = "\t",
  col.names = NA, quote=FALSE)
> q()
```

wird das Ganze noch in eine Datei namens IRpvalue.out geschrieben und über ein weiteres Perl-Skript die Ausgabe so formatiert, dass die folgende Tabelle mit Ontologienamen und p-value für die über den I_r errechnete Genliste entsteht:

```
Ontology                                                        p-value
-----------------------------------------------------------------------

Nucleoside, nucleotide and nucleic acid metabolism => 8.582611E-61
Transcription factor                               => 1.120125E-44
Signal transduction                                => 1.222096E-44
mRNA transcription                                 => 2.679662E-41
Ectoderm development                               => 1.241211E-35
Developmental processes                            => 1.556178E-34
Cell surface receptor mediated signal transduction => 2.424057E-25
Nucleic acid binding                               => 1.119281E-21
Kinase                                             => 9.653730E-21
Protein modification                               => 2.024457E-18
Receptor                                           => 5.535674E-17
mRNA transcription regulation                      => 1.068920E-16
Homeobox transcription factor                      => 1.154772E-12
Intracellular signaling cascade                    => 1.132301E-11
Neurogenesis                                       => 1.472666E-11
Cell structure and motility                        => 1.216454E-09
Hydrolase                                          => 3.290662E-07
Protein kinase receptor                            => 8.936140E-07
Protein kinase                                     => 1.172729E-06
Other DNA-binding protein                          => 1.812553E-06
G-protein mediated signaling                       => 1.352700E-05
Cell proliferation and differentiation             => 1.190085E-04
Protein phosphorylation                            => 2.030136E-04
Tyrosine protein kinase receptor                   => 2.856403E-03
Kinase inhibitor                                   => 3.227470E-03
Nitrogen metabolism                                => 1.433807E-02
Biological process unclassified                    => 6.727227E-02
Molecular function unclassified                    => 1.273316E-01
```

Hiermit kann nun eine Interpretation der Ergebnisse versucht werden, wobei allerdings die im Verlauf der Auswertung aufgetretenen Unsicherheiten berücksichtigt werden müssen.

6.3 Interpretation der Ergebnisse

Im Experiment standen unter Verwendung des 'normalen' p-value Entwicklungs- und Transcriptions-Gene im Vordergrund. Mit dem modifizierten p-value konnte nun eine andere Reihenfolge der Ontologien mit teilweise noch deutlicheren Signifikanzen beobachtet werden. Nachfolgend werden als Ontologie-

namen die englischen Bezeichnungen verwendet, um mit der Darstellung im letzten Abschnitt konform zu gehen. Die Beschreibungen der aufgeführten Ontologien werden von der PANTHER-Webpage zitiert, die alle Terme unter http://www.pantherdb.org/panther/prowler.jsp als eine Art Baumstruktur enthält und noch detailierter ausführt.

Der Stoffwechselprozess *Nucleoside, nucleotide and nucleic acid metabolism* hat die höchste Signifikanz. Eine wesentliche Rolle spielen dabei die dieser Ontologie untergeordneten biologischen Prozesse *mRNA transcription* und *mRNA transcription regulation*, deren Aufgabe die Synthese von mRNA durch RNA-Polymerase unter Benutzung eines DNA-Templates ist.

An zweiter Stelle steht die molekulare Funktion *Transcription factor*. Unter einem Transkriptionsfaktor versteht man ein Protein, das zur Regulation der RNA-Polymerase erforderlich ist. Ein wichtiges Mitglied ist hier der *Homeobox transcription factor* als Teil der in Abschnitt 6.1 erwähnten HOX-Cluster, der die Homeobox-Domäne enthält.

Es folgt der biologische Prozess *Signal transduction* als Ablauf der rezeptorabhängigen Signalereignisse, die für eine Zelle erforderlich sind, um auf ein extrazelluläres Signal zu antworten. Diesem zugeteilt sind auch die *Cell surface receptor mediated signal transduction* als Teil des Signalübertragungsprozesses, der über einen Rezeptor an der Zelloberfläche vermittelt wird und auch *Intracellular signaling cascade* als Teil der Signalübertragung im Zytoplasma der Zelle.

Interessant ist auch die Signifikanz von *Developmental processes* und dessen Untergruppen *Ectoderm development*, sowie *Neurogenesis*. Hier steht der Ablauf von Prozessen im Mittelpunkt, die für die Entwicklung eines multizellulären Organismus aus einer einzelnen Zelle erforderlich sind. *Ectoderm development* ist dabei der Prozess, der die Erzeugung und Differenzierung des Ektoderms, also des äußersten der drei embryonalen Keimblätter, beinhaltet. *Neurogenesis* beschreibt die hieraus erfolgende Erzeugung des Nervensystems.

Zur Funktion *Nucleic acid binding*, werden Nukleinsäure bindende Moleküle gezählt. Diese sind in verschiedene Enzyme und bindende Proteine aufgeteilt, u.a. in *Other DNA-binding protein* als Proteine, die zwar DNA binden, aber keine andere bekannte molekulare Funktion haben.

Die Funktion *Kinase*, beinhaltet Enzyme, die den Phosphatase-Transfer von ATP zu einem anderen Substrat katalysieren. Dazu zählt *Protein kinase* für den Übergang der Phosphatase auf die Hydroxyl-Seitenketten von Proteinen und hierzu wiederum der *Tyrosine protein kinase receptor*, der Tyrosinreste von Proteinen phosphorylisiert.

Protein modification ist der biologische Prozess der post-translationalen Modifikation von Proteinen, die u.a. durch *Protein phosphorylation* geschieht,

durch die eine Phosphatgruppe kovalent an einen bestimmten Aminosäurerest gebunden wird.

Erwähnt werden soll noch die molekulare Funktion *Receptor*. Ein Rezeptor ist eine molekulare Struktur, innerhalb einer Zelle oder an der Zelloberfläche, die durch selektives Binden einer spezifischen Substanz und einem begleitenden spezifischen biologischen Effekt gekennzeichnet ist. Ein *Protein kinase receptor* ist eine solche Struktur und enthält eine extrazelluläre Liganden bindende Domäne eine einzelne Transmembranen Domäne und eine intrazelluläre Kinasedomäne. Ein Mitglied hiervon ist der oben schon einmal erwähnte *Tyrosine protein kinase receptor*. Dieser ist ein Beispiel dafür, dass in PANTHER einzelne Ontologie-Terme mehreren Eltern zugeteilt werden können, nämlich hier den molekularen Funktionen *Receptor* und *Kinase*.

Die weiteren aufgeführten Ontologien stehen für sich alleine und haben keine weiteren Untergruppen, die sich durch eine deutliche Signifikanz hervorheben.

6.4 Kritik am Auswerteverfahren

Um die im letzten Abschnitt erhaltenen Signifikanzen zu bestätigen, müßten v.a. Vergleiche mit Genomen von anderen Säugern gezogen werden. Dabei gab es aber das Problem der Bestimmung von ID's, die mit PANTHER ausgewertet werden konnten. Z.B. konnten ca. 30 Prozent der über die I_r-Auswertung gefundenen Gene bei der Maus nicht verwertet werden, obwohl noch über das Webinterface http://www.ensembl.org/biomart/martview/ eine andere Möglichkeit gefunden wurde, Ensembl-ID's direkt in RefSeq-ID's oder andere Formate umzuwandeln, womit in PANTHER bessere Ergebnisse erzielt worden sind, als über die Auswertung mit ACC-ID's.

Die Möglichkeit, über die Webanwendung die GeneID von Panther, den Gennamen und -symbol, die Start- und Endposition, sowie den chromosomalen Lokalisationsort als Datei abzuspeichern, wurde erst am Ende des Projektes erkannt. Mit diesem Datensatz könnte man theoretisch eine eigene Datenbank aufbauen und dort Gene suchen, die sich mit den kumulierten I_r-Bereichen schneiden. Damit könnte man direkt, also ohne über das Webinterface, die einzelnen Gene und ihre zugehörigen Ontologien herausfinden. Mangels Zeit wurde dieser Ansatz nur noch kurz getestet. Dabei mußten folgende Probleme festgestellt werden. Erstens gab es scheinbare 'Bugs', wie z.B., dass anstatt der Chromosomennummer allein, der Wert 19:NT_113949 stand, wie für GeneID:553128. Mangels Zeit konnte nicht mehr eruiert werden, ob es im gesamten Genom noch mehrere ähnlich abweichende Einträge gibt, die die Übertragung in eine Datenbank stören könnten. Außerdem fiel

beim ersten Testen auf, dass die Start- und Endeinträge der einzelnen Gene z.T. unterschiedlich in PANTHER und ENSEMBL gehandhabt werden. Z.B. wird in PANTHER für das Gen GeneID:729873 ein Startwert von 33358323 und als Ende 33369482 angegeben (Chr17). Der in PANTHER mit dieser ID referenzierte ENSEMBL-Eintrag reicht dagegen nur von 33358353 bis 33369293. Die meisten Enträge sind in beiden Datenbanken natürlich kongruent. Nichtsdestotrotz wurden z.B. auf Chromosom 7 mit 35 zu 17 gut doppelt soviel Gene gefunden, die sich mit niedrigen I_r-Bereichen überschnitten. Für weitere Auswertungen sollte dieser Umstand natürlich erst näher untersucht werden.

Obwohl es sich bei PANTHER um eine schnelle und übersichtliche Methode handelt, Ontologien bestimmten Genen zuzuordnen, wird das Ergebnis durch den Informationsverlust getrübt, den man erhält, weil PANTHER scheinbar nicht mit den verschiedenen Gen-Identifikatoren umgehen kann. Bei anderen Säugergenomen als dem menschlichen, scheint dieses Problem sogar noch größer zu sein. Um mit PANTHER vernünftig Arbeiten zu können, ist es daher notwendig, einen Weg zu finden, wie jedem Gen, dass durch einen niedrigen I_r gekennzeichnet ist, auch einen Identifikator zugewiesen werden kann, mit dem man auch Ergebnisse bei PANTHER-Datenbankabfragen erhält. Gelingt dies, kann die PANTHER-Struktur als effiziente und zunehmend statistisch abgesichertere Grundlage genutzt werden, um Zusammenhänge zwischen Genfunktion und dem *Index of Repetitiveness* zu erhalten.

Noch anwenderfreundlicher wäre es, könnte man über PANTHER eine bereits fertige und bereinigte Datenbank herunterladen, wie das z.B. für die GO-Annotation/Ontologie als mysql-Version möglich ist. Dies würde die Auswerteprozedur natürlich wesentlich vereinfachen. Man könnte direkt über diese Datenbank die Gene herausfinden, die sich mit niedrigen I_r-Bereichen überschneiden und hätte auch gleich die zugehörigen molekularen Funktionen und biologischen Prozesse eruiert. Über deren aufgetreten Häufigkeiten würde sich dann wieder wie oben erklärt die statistische Auswertung durchführen lassen.

Kapitel 7

Schlussbemerkung

In Kapitel 4 wurde der *Index of Repetitiveness* vorgestellt, der in [HW06] als ein neues Maß zur Messung der Repetitivität von Genomen eingeführt wurde. Mit diesem ist es möglich, Sequenzvergleiche nicht mehr über zeitaufwändige Alignments, sondern eben über die Repetitivität der zu analysierenden Genome durchzuführen.

Grundlage hierfür bildet die Verwendung von *shortest unique substrings*, die sich wiederum effektiv über Suffixbäume bestimmen lassen.

Die Verwendung linearer Algorithmen zur Konstruktion solcher Suffixbäume läßt zwar ein insgesamt ebenfalls lineares Verhalten zur Berechnung des I_r zu, ist aber von der Größe der analysierbaren Sequenzen her auf den verfügbaren Hauptspeicher beschränkt.

Als eine mögliche Überwindung dieses Hindernisses wurde in Kapitel 3 der *Top-Down Disk-Based*-Algorithmus vorgestellt, der Suffixbäume für große Sequenzen auf Festplatte konstruieren kann und dabei sehr effizient arbeitet. Grundlage hierfür ist eine spezielle Pufferungsstrategie, die die verwendeten Datenstrukturen je nach Zugriffshäufigkeit im Hauptspeicher behält oder auf die Festplatte auslagert. Damit wird erreicht, dass kleine Sektionen des Baumes im Hauptspeicher konstruiert und dann zur Festplatte transferiert werden. Trotz dieser Plattenzugriffe bleibt das Verfahren durch eine verbesserte Referenzlokalität schnell.

Diese Methode wurde deshalb als Mittel der Wahl angesehen, um als Grundlage für die Berechnung des *Index of Repetitiveness* zu dienen. Dazu musste in Erfahrung gebracht werden, wie die konstruierte und abgespeicherte Suffixbaumstruktur zusammengesetzt ist und wie darin *shortest unique substrings* herausgefunden werden können.

In Kapitel 5 wurde das dafür neu implementierte Programm *tdd_ir* vorgestellt, das zudem Methoden enthält, um den *Index of Repetitiveness* berechnen zu können.

Weil die zugrunde liegende Implementierung des *TDD*-Algorithmus allerdings nicht die erhoffte Performance aufweisen konnte und auch noch Restriktionen bei Sequenzen mit sich häufig wiederholenden Abschnitten hingenommen werden mussten, lag die Suche nach einer Alternative nahe.

Diese fanden die I_r-Autoren wieder in einem 'in-memory'-Algorithmus, der aber dieses Mal auf Suffixarrays beruhte und damit wesentlich sparsamer mit dem Hauptspeicherplatz umging.

Das hierzu implementierte Programm *ir* wurde dann auch verwendet, um für die einzelnen Chromosomen die *Index of Repetitiveness*-Werte zu berechnen.

Die I_r-Autoren beobachteten im Experiment außerdem, dass es evtl. einen Zusammenhang zwischen Bereichen mit niedrigen I_r-Werten und der Funktionalität von sich damit überschneidenden Genen geben könnte. Um dieser Vermutung nachzugehen, deren Grundlage die Ausprägungen der menschlichen HOX-Cluster war, wurde zunächst definiert, ab wann der I_r-Wert für eine Region als 'niedrig' gilt. Danach wurden die Gene ermittelt, die sich mit diesen Bereichen überschnitten.

Diese Gene wurden dann über ihre ID in die Ontologie-Datenbank PANTHER eingespeist, um festzustellen, welche molekularen Funktionen und biologischen Prozesse sie haben. Anhand einer modifizierten Binomialstatistik wurde die Signifikanz der erhaltenen Werte berechnet und teilweise Zusammenhänge herausgefunden, die mit einer hohen Wahrscheinlichkeit auftraten.

Dieses Ergebnis wird etwas dadurch getrübt, dass nicht allen Genen, die über die Expressionsdatenbank herausgefunden wurden, eine ID zuweisbar war, mit der PANTHER umgehen konnte, so dass ein gewisser Informationsverlust entstand. Diese Methode ist deshalb sicher noch verbesserungswürdig, interessant wäre vor allem, wenn PANTHER selber eine Datenbank zur Verfügung stellen würde, in der alle Gene enthalten sind, damit gleich dort nach Überschneidungen gesucht werden könnte.

Nichtsdestotrotz stellten sich hohe Signifikanzen u.a. in den Bereichen Nukleinsäurestoffwechsel, Transkriptionsfaktoren, zu der z.B. auch die HOX-Cluster zählen, Signalübertragung, Entwicklung, Bindung, Transfer und Modifikation, sowie bei Rezeptoren ein.

Eine endgültige verbindliche Aussage zu einem Zusammenhang zwischen Genomregionen mit niedrigem I_r lässt sich aber allein schon aufgrund der vorhin erwähnten nicht analysierbaren Gene noch nicht treffen. Es fehlen letztendlich aus denselben Gründen auch noch Analysen und Vergleiche von und mit anderen Säugergenomen, z.B. dem Mausgenom. Diese sollten auf alle Fälle durchgeführt werden um zu sehen, ob sich ähnliche Ergebnisse ergeben.

Das Abbild der I_r-Werte könnte sich auch noch etwas ändern, wenn sie für das komplette Humangenom zusammenhängend analysiert werden könnten.

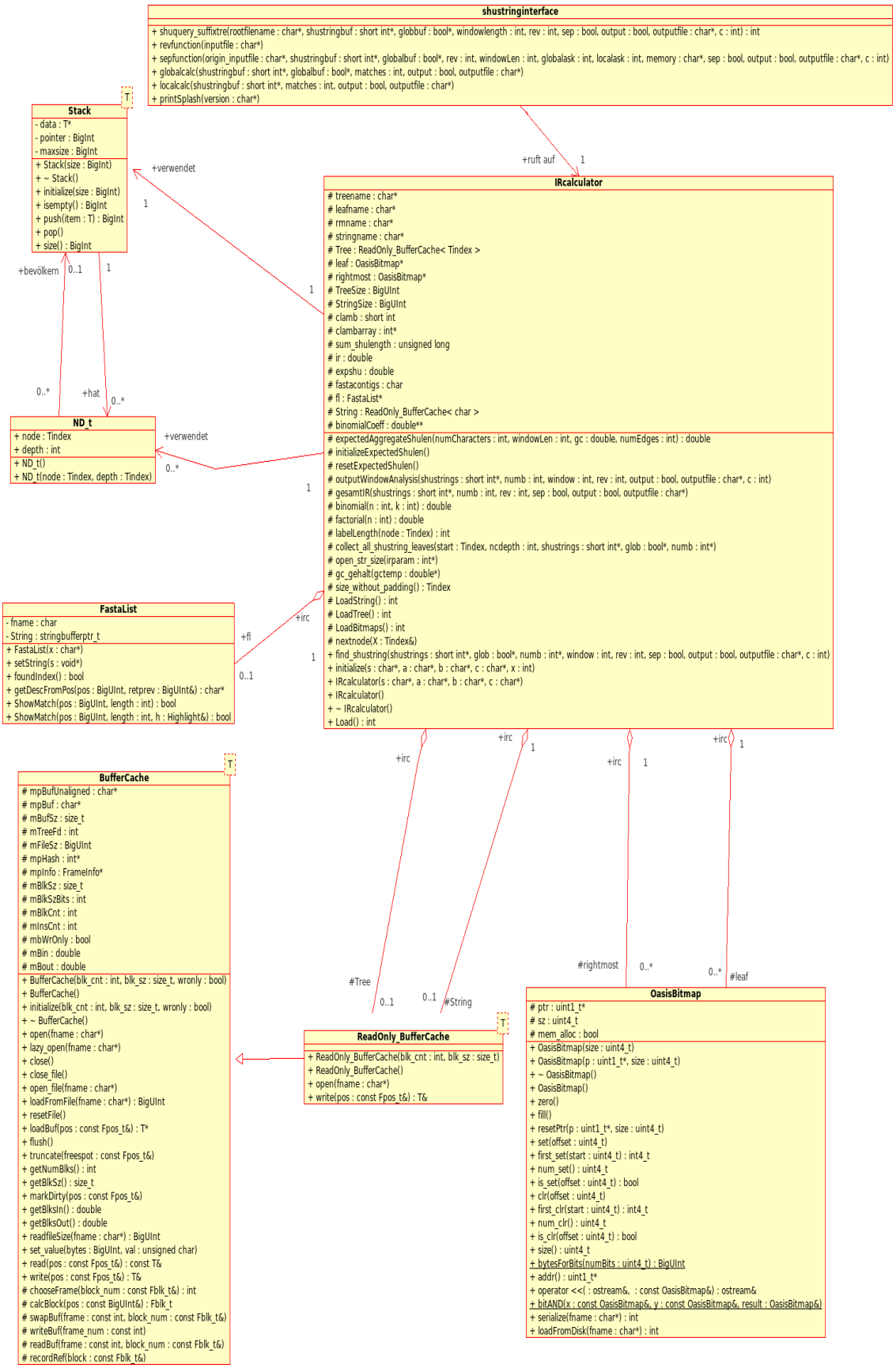
Denn die zugrunde liegenden *shortest unique substrings*, die innerhalb eines Chromosoms einzigartig sind, könnten sich ja nochmal woanders im Genom wiederholen.

Deshalb wird das Programm *TDD_IR* nochmal zur Analyse interessant werden, sobald die angekündigte neue Version der zugrunde liegenden TDD-Implementierung veröffentlicht wird und darin die bisherigen Mängel behoben sind. Eine weitere Alternative wäre, nach persistenten Suffixarraykonstruktionen Ausschau zu halten.

Es wurde gezeigt, dass der *Index of Repetitiveness* ein nützliches Werkzeug sein könnte, um bestimmte funktionelle Regionen in einem Genom aufzuspüren. Bezüglich der Analyse und dem Vergleich von mehreren Genomen, scheint dabei trotz 64-Bit Rechenmaschinen die Verwendung von persistenten Algorithmen zur Suffixbaum- oder Suffixarraykonstruktion das Mittel der Wahl. Es bleibt zu hoffen, dass in Zukunft tatsächlich Methoden entwickelt oder bestehende Techniken ausgebaut werden, die eine Analyse solch großer Datenmengen ermöglichen, damit dieser interessante Ansatz weiter verfolgt werden kann.

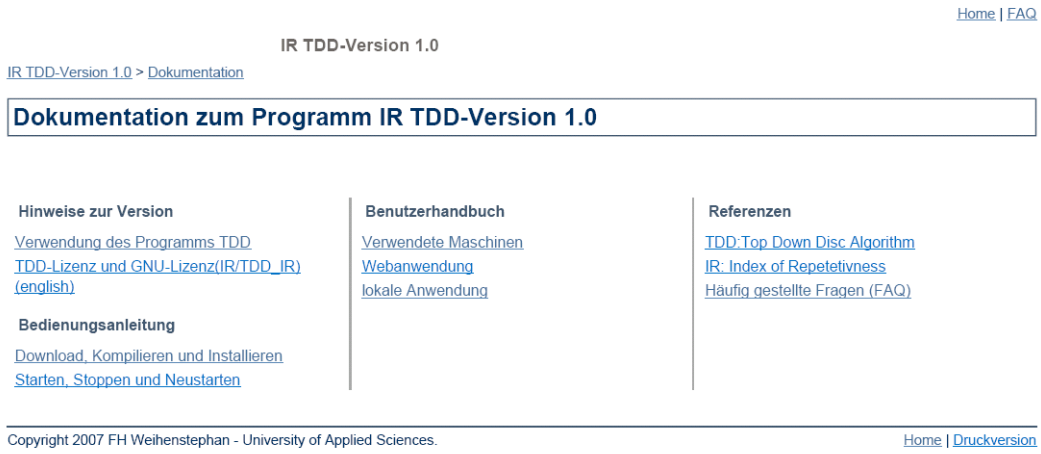
Anhang A

Klassendiagramm zu TDD_IR


shustringinterface
+ shuquery_suffixtre(rootfilename : char*, shustringbuf : short int*, globbuf : bool*, windowlength : int, rev : int, sep : bool, output : bool, outputfile : char*, c : int) : int
+ revfunction(inputfile : char*)
+ sepfunction(origin_inputfile : char*, shustringbuf : short int*, globalbuf : bool*, rev : int, windowLen : int, globalask : int, localask : int, memory : char*, sep : bool, output : bool, outputfile : char*, c : int)
+ globalcalc(shustringbuf : short int*, globalbuf : bool*, matches : int, output : bool, outputfile : char*)
+ localcalc(shustringbuf : short int*, matches : int, output : bool, outputfile : char*)
+ printSplash(version : char*)
+ruft auf
1
Stack
T
- data : T*
- pointer : BigInt
- maxsize : BigInt
+ Stack(size : BigInt)
+ ~ Stack()
+ initialize(size : BigInt)
+ isempty() : BigInt
+ push(item : T) : BigInt
+ pop()
+ size() : BigInt
+verwendet
1
1
IRcalculator
treename : char*
leafname : char*
rmname : char*
stringname : char*
Tree : ReadOnly_BufferCache< Tindex >
leaf : OasisBitmap*
rightmost : OasisBitmap*
TreeSize : BigUInt
StringSize : BigUInt
clamb : short int
clambarray : int*
sum_shulength : unsigned long
ir : double
expshu : double
fastacontigs : char
fl : FastaList*
String : ReadOnly_BufferCache< char >
binomialCoeff : double**
expectedAggregateShulen(numCharacters : int, windowLen : int, gc : double, numEdges : int) : double
initializeExpectedShulen()
resetExpectedShulen()
outputWindowAnalysis(shustrings : short int*, numb : int, window : int, rev : int, output : bool, outputfile : char*, c : int)
gesamtIR(shustrings : short int*, numb : int, rev : int, sep : bool, output : bool, outputfile : char*)
binomial(n : int, k : int) : double
factorial(n : int) : double
labelLength(node : Tindex) : int
collect_all_shustring_leaves(start : Tindex, ncdepth : int, shustrings : short int*, glob : bool*, numb : int*)
open_str_size(irparam : int*)
gc_gehalt(gctemp : double*)
size_without_padding() : Tindex
LoadString() : int
LoadTree() : int
LoadBitmaps() : int
nextnode(X : Tindex&)
+ find_shustring(shustrings : short int*, glob : bool*, numb : int*, window : int, rev : int, sep : bool, output : bool, outputfile : char*, c : int)
+ initialize(s : char*, a : char*, b : char*, c : char*, x : int)
+ IRcalculator(s : char*, a : char*, b : char*, c : char*)
+ IRcalculator()
+ ~ IRcalculator()
+ Load() : int
+bevölkern
0..1
1
0..*
+hat
0..*
ND_t
+ node : Tindex
+ depth : int
+ ND_t()
+ ND_t(node : Tindex, depth : Tindex)
+verwendet
0..*
1
FastaList
- fname : char
- String : stringbufferptr_t
+ FastaList(x : char*)
+ setString(s : void*)
+ foundIndex() : bool
+ getDescFromPos(pos : BigUInt, retprev : BigUInt&) : char*
+ ShowMatch(pos : BigUInt, length : int) : bool
+ ShowMatch(pos : BigUInt, length : int, h : Highlight&) : bool
+fl
0..1
+irc
1
+irc
+irc
1
+irc
1
+irc
1
BufferCache
T
mpBufUnaligned : char*
mpBuf : char*
mBufSz : size_t
mTreeFd : int
mFileSz : BigUInt
mpHash : int*
mpInfo : FrameInfo*
mBlkSz : size_t
mBlkSzBits : int
mBlkCnt : int
mInsCnt : int
mbWrOnly : bool
mBin : double
mBout : double
+ BufferCache(blk_cnt : int, blk_sz : size_t, wronly : bool)
+ BufferCache()
+ initialize(blk_cnt : int, blk_sz : size_t, wronly : bool)
+ ~ BufferCache()
+ open(fname : char*)
+ lazy_open(fname : char*)
+ close()
+ close_file()
+ open_file(fname : char*)
+ loadFromFile(fname : char*) : BigUInt
+ resetFile()
+ loadBuf(pos : const Fpos_t&) : T*
+ flush()
+ truncate(freespot : const Fpos_t&)
+ getNumBlks() : int
+ getBlkSz() : size_t
+ markDirty(pos : const Fpos_t&)
+ getBlksIn() : double
+ getBlksOut() : double
+ readfileSize(fname : char*) : BigUInt
+ set_value(bytes : BigUInt, val : unsigned char)
+ read(pos : const Fpos_t&) : const T&
+ write(pos : const Fpos_t&) : T&
# chooseFrame(block_num : const Fblk_t&) : int
# calcBlock(pos : const BigUInt&) : Fblk_t
# swapBuf(fframe : const int, block_num : const Fblk_t&)
# writeBuf(frame_num : const int)
# readBuf(frame : const int, block_num : const Fblk_t&)
# recordRef(block : const Fblk_t&)
#Tree
0..1
0..1
#String
ReadOnly_BufferCache
T
+ ReadOnly_BufferCache(blk_cnt : int, blk_sz : size_t)
+ ReadOnly_BufferCache()
+ open(fname : char*)
+ write(pos : const Fpos_t&) : T&
#rightmost
0..*
0..*
#leaf
OasisBitmap
ptr : uint1_t*
sz : uint4_t
mem_alloc : bool
+ OasisBitmap(size : uint4_t)
+ OasisBitmap(p : uint1_t*, size : uint4_t)
+ ~ OasisBitmap()
+ OasisBitmap()
+ zero()
+ fill()
+ resetPtr(p : uint1_t*, size : uint4_t)
+ set(offset : uint4_t)
+ first_set(start : uint4_t) : int4_t
+ num_set() : uint4_t
+ is_set(offset : uint4_t) : bool
+ clr(offset : uint4_t)
+ first_clr(start : uint4_t) : int4_t
+ num_clr() : uint4_t
+ is_clr(offset : uint4_t) : bool
+ size() : uint4_t
+ bytesForBits(numBits : uint4_t) : BigUInt
+ addr() : uint1_t*
+ operator <<(: ostream&, : const OasisBitmap&) : ostream&
+ bitAND(x : const OasisBitmap&, y : const OasisBitmap&, result : OasisBitmap&)
+ serialize(fname : char*) : int
+ loadFromDisk(fname : char*) : int

Anhang B

TDD_IR - Installationsanleitung



Home | FAQ

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Dokumentation zum Programm IR TDD-Version 1.0

Hinweise zur Version

Verwendung des Programms TDD
TDD-Lizenz und GNU-Lizenz(IR/TDD_IR) (english)

Bedienungsanleitung

Download, Kompilieren und Installieren
Starten, Stoppen und Neustarten

Benutzerhandbuch

Verwendete Maschinen
Webanwendung
lokale Anwendung

Referenzen

TDD:Top Down Disc Algorithm
IR: Index of Repetetivness
Häufig gestellte Fragen (FAQ)

Copyright 2007 FH Weihenstephan - University of Applied Sciences.

Home | Druckversion

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Verwendung des Programms tdd

Dieses Dokument beschreibt die Verwendung des Programms TDD im Zusammenhang mit TDD_IR.

TDD

Beschreibung:

TDD ist ein Algorithmus zur Konstruktion von Suffixbäumen. Die meisten dafür vorhandenen Algorithmen sind sogenannte 'in-memory'-Algorithmen, die eine schlechte Performance aufweisen, wenn sie auf große Sequenzen angewandt werden.

TDD dagegen benützt eine auf Festplatte basierende 'Top-Down-Annäherung' für eine schnelle Suffix-Baum-Konstruktion (TDD-Homepage).

TDD_IR wiederum verwendet diese auf Festplatte gespeicherten Suffix-Bäume, um daraus den 'Index of Repetitivness' für die zu untersuchenden Sequenzen zu berechnen. TDD_IR ruft dabei das Programm TDD selber auf, kann aber optional auch direkt von TDD erzeugte Datensätze verarbeiten. TDD wird detailiert in dem Paper (Sandeep Tata, Richard A. Hankins, and Jignesh Patel, Practical Suffix Tree Construction, VLDB 2004) beschrieben.

 Home | Druckversion

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Download, Kompilieren und Installieren

Dieses Dokument beschreibt den Download, das Kompilieren und die Installation des in C++ implementierten Programms tdd_ir, unter dem Betriebssystem UNIX bzw. UNIX-ähnlichen Systemen. Getestet wurde es unter SUSE Linux 10.1 (gcc-version 4.1.0), DEBIAN 3.1 (gcc-version 3.3.5) und DEBIAN KNOPPIX 5.0.1 (gcc-version 4.0.4).

Anforderungen

Folgende Anforderungen gelten für das Erstellen und die korrekte Funktion von TDD und TDD_IR:

Disk Space

Die Installation von TDD benötigt ungefähr 14 MB und die von TDD_IR ungefähr 3 MB an freier Festplattenkapazität.
Um Dateien zu verarbeiten, wird durch das plattenbasierende Konzept ebenfalls noch freier Festplattenplatz benötigt. Z.B. braucht Mycoplasma genitalium (Größe: ca. 580 KB) insgesamt 8,2 MB für die TDD-Dateien.
TDD benötigt standardmäßig 256000 KB freien Hauptspeicherplatz. Dieser kann mit der Option '-MEM' entsprechend angepaßt werden. Zur Orientierung können die Werte für die getesteten Maschinen dienen, die unter FAQ einsehbar sind.

ANSI-C Compiler and Build System

Bitte stellen Sie sicher, dass ein ANSI-C Kompiler installiert ist. Empfohlen wird der GNU C compiler (GCC) der Free Software Foundation (FSF). Andere kompatible ANSI-Kompiler wurden nicht getestet Außerdem muß die Variable `PATH` 'basic tools' wie z.B. `make` und `tr` enthalten.

Perl 5

TDD enthält einige Skripten, die einen PERL-Interpreter benötigen. Getestet wurden hier die Versionen 5.8.4 und 5.8.8 unter Linux Debian, sowie 5.8.8 unter SUSE Linux.

Vorabinstallation und Nutzung des Programms TDD

Bei TDD_IR handelt es sich um ein Programm, das Datentypen verarbeitet, die von dem Programm TDD erzeugt werden. Dazu muß erst das Programm TDD installiert sein, welches ebenfalls als Binärdatei oder als vollständiger Quell-Code erhältlich ist.
Das Entpacken der downloadbaren Archivdateien gestaltet sich als unproblematisch und läßt sich einfach mit folgenden Befehlen bewerkstelligen:

```
$ tar xzvf tddbin.tgz
```

bzw.

```
$ tar xzvf tdd.tgz
```

Für die Quellcodeversion muß man nun in das /src-Verzeichnis wechseln, erst 'make' und dann 'make install' eingeben.
Im Anschluß daran muß noch die Variable TDDDIR den Umgebungsvariablen und der Weg zum tdd/bin-Verzeichnis der PATH-Variablen hinzugefügt werden (wie in der zugehörigen README-Datei bzw. unter TDD-Homepage beschrieben), damit TDD ggf. aus tdd_ir heraus aufrufbar ist. Zitiert werden soll hier der Eintrag aus der 'Quellcode-README':

```
* Add TDDDIR to your environment. For instance is you use bash, then add the
following lines to your .bash_profile TDDDIR=/home/username/tdd; export
TDDDIR

* Add the path to tdd/bin to your PATH variable: PATH=$TDDDIR/bin:$PATH;
export PATH
```

Es gibt zwei Arten, wie TDD von der lokalen TDD_IR-Anwendung verwendet wird. Einmal erfolgt der Aufruf aus tdd_ir selber heraus (Option -i) und das andere Mal können mit der Option -t die Dateien, die von TDD erzeugt werden, direkt übernommen werden. Näheres hierzu ist im Benutzerhandbuch unter (lokale Anwendung) einzusehen.
An dieser Stelle soll aber noch kurz darauf eingegangen werden, wie man eine Datei mit TDD separat vorverarbeitet, wobei die ausführliche Dokumentation wieder unter TDD-Homepage einzusehen ist.
Eine vollständig im FASTA-Format vorliegende DNA-Datei wird unter TDD mit dem Befehl *tdd.pl -IF F -DT D* verarbeitet. Sollte es dabei zu Speicherzugriffsfehlern kommen, hilft u.U. noch das Anhängen der Option *-MEM*. Dabei kann der standardmäßig herangezogene Hauptspeicher von 256000 KB je nach

Ausstattung erhöht werden. In FAQ findet man die Dateigrößen, die auf verschiedenen Maschinen verarbeitet werden konnten.

Download

TDD kann unter TDD-Homepage heruntergeladen werden.
TDD_IR ist unter tdd_ir.tgz erhältlich. Es gibt hier ebenfalls neben dieser Quellcodeversion auch noch eine Binär-Version (tdd_irbin.tgz). Beide wurden wie unter 'verwendete Maschinen' beschrieben getestet.

Extract

Die Extrahierung der tdd_ir-Archive sollte ebenfalls unproblematisch sein:

```
$ tar xzvf tdd_irbin.tgz
```

für die Binärdatei bzw.

```
$ tar xzvf tdd_ir.tgz
```

für den Quellcode.

Dies kreirt ein neues Verzeichnis unter dem momentanen Verzeichnis namens TDD_IR, welches entweder den kompletten Sourcecode oder nur den Binärcode, jeweils einschließlich der Dateien *COPYING, README und der Testdatei I43967.fasta* enhält. Mit cd wird am besten in dieses Verzeichnis gewechselt, bevor man dort tdd_ir ausführt bzw. mittels des Befehls 'make' erstellt.

Build

Bei der Quell-Code-Variante besorgt der Befehl:

```
$ make
```

das Erstellen der ausführbaren Datei tdd_ir. In der Binär-Code-Variante ist die ausführbare Datei tdd_ir schon vorhanden. Sollte es zu etwaigen 'Rechteproblemen' auf den verwendeten Maschinen kommen, wird gebeten,

dies in der entsprechenden Distributionskonfiguration anzupassen.
Um tdd_ir überall aufrufen zu können, sollte der Pfad zum tdd_ir-Verzeichnis der PATH-Variablen hinzugefügt werden.

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Starten, Stoppen und Neustarten

Starten

Beschreibung:

Für die lokale Anwendung wird hier auf die Beschreibungen im Kapitel Bedienungsanleitung: Kompilieren und installieren, sowie Benutzerhandbuch: lokale Anwendung verwiesen.
Allerdings soll an dieser Stelle erwähnt werden, dass es möglicherweise zu Verarbeitungsfehlern in Form von Speicherzugriffsfehlern kommen kann, falls zwei Jobs mit jeweils großen Datenmengen gleichzeitig gestartet werden. Abhängig ist dies von den jeweiligen Systemkonfigurationen. Die Empfehlung lautet daher, in solchen Fällen die Berechnungen einzeln hintereinander laufen zu lassen. So wird z.B. auch bei Auswahl der Option *-s* vorgegangen.

Die Webanwendung läßt sich nach dem Upload einer Datei und Auswahl der Optionen durch einen Klick auf den Button *'Compute'* starten.

Stoppen und Neustarten

Beschreibung:

Die lokale Anwendung läßt sich einfach mit der Tastenkombination *CTRL+C* unterbrechen. Wird gerade noch das Programm *TDD* ausgeführt, kann es zu einer kurzen Verzögerung kommen, bis der Prozess beendet wird. Der Prozess kann auch über den *'kill'*-Befehl beendet werden. Eine Wiederaufnahme an dem abgebrochenen Punkt ist allerdings nicht möglich! Das Programm muß komplett neu gestartet werden.

Über das Webinterface kann die Berechnung auf dem zuständigen Server nicht direkt abgebrochen und bei größeren Datensätzen während der Berechnungszeit auch kein zweiter Job gleichzeitig verarbeitet werden.

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Verwendete Maschinen

Maschinenkonfiguration:

DEBIAN Linux: TDD_IR wurde unter dem Betriebssystem DEBIAN Linux auf folgenden zwei Maschinen getestet:

- Betriebssystem: *DEBIAN Linux Version 3.1*
- Kompiler: *gcc-version 3.3.5*
- Prozessor: *Intel(R) Pentium(R) 3 CPU 1.00GHz*
- Hauptspeicher: *4 GB*

sowie auf:

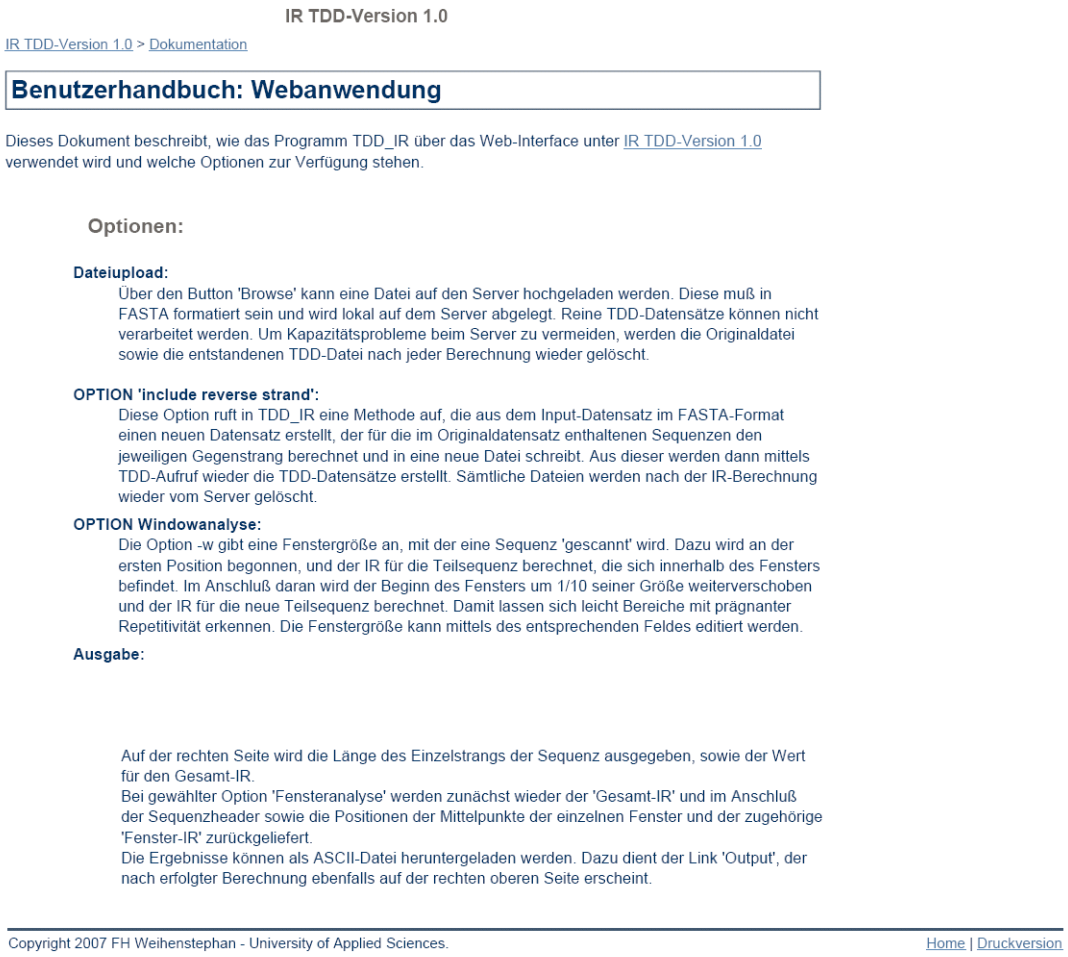
- Betriebssystem: *DEBIAN Linux KNOPPIX-Version 5.0.1*
- Kompiler: *gcc-version 4.0.4*
- Prozessor: *Intel(R) Pentium(R) 4 CPU 3.00GHz*
- Hauptspeicher: *1 GB*

SUSE Linux: Es konnte auch unter SUSE Linux auf folgender Maschine getestet werden. Hierbei kam es allerdings zu Problemen bei der Kompilierung von TDD mit der verwendeten gcc-version 4.1.0, die nicht mehr weiter untersucht wurden. Verwendet wurde deshalb hier die Binärversion von TDD.
TDD_IR selber wurde zwar mit 'warnings' kompiliert, die aber keinen Einfluß auf die Funktionsfähigkeit des Programms hatten:

- Betriebssystem: *SUSE Linux 10.1*
- Kompiler: *gcc-version 4.1.0*
- Prozessor: *Intel(R) Pentium(R) Core(TM) Duo CPU 2.00 GHz*
- Hauptspeicher: *1 GB*

Anhang C

TDD_IR - Benutzerhandbuch



IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Benutzerhandbuch: Webanwendung

Dieses Dokument beschreibt, wie das Programm TDD_IR über das Web-Interface unter IR TDD-Version 1.0 verwendet wird und welche Optionen zur Verfügung stehen.

Optionen:

Dateiupload:
Über den Button 'Browse' kann eine Datei auf den Server hochgeladen werden. Diese muß in FASTA formatiert sein und wird lokal auf dem Server abgelegt. Reine TDD-Datensätze können nicht verarbeitet werden. Um Kapazitätsprobleme beim Server zu vermeiden, werden die Originaldatei sowie die entstandenen TDD-Datei nach jeder Berechnung wieder gelöscht.

OPTION 'include reverse strand':
Diese Option ruft in TDD_IR eine Methode auf, die aus dem Input-Datensatz im FASTA-Format einen neuen Datensatz erstellt, der für die im Originaldatensatz enthaltenen Sequenzen den jeweiligen Gegenstrang berechnet und in eine neue Datei schreibt. Aus dieser werden dann mittels TDD-Aufruf wieder die TDD-Datensätze erstellt. Sämtliche Dateien werden nach der IR-Berechnung wieder vom Server gelöscht.

OPTION Windowanalyse:
Die Option -w gibt eine Fenstergröße an, mit der eine Sequenz 'gescannt' wird. Dazu wird an der ersten Position begonnen, und der IR für die Teilsequenz berechnet, die sich innerhalb des Fensters befindet. Im Anschluß daran wird der Beginn des Fensters um 1/10 seiner Größe weiterverschoben und der IR für die neue Teilsequenz berechnet. Damit lassen sich leicht Bereiche mit prägnanter Repetitivität erkennen. Die Fenstergröße kann mittels des entsprechenden Feldes editiert werden.

Ausgabe:

Auf der rechten Seite wird die Länge des Einzelstrangs der Sequenz ausgegeben, sowie der Wert für den Gesamt-IR.
Bei gewählter Option 'Fensteranalyse' werden zunächst wieder der 'Gesamt-IR' und im Anschluß der Sequenzheader sowie die Positionen der Mittelpunkte der einzelnen Fenster und der zugehörige 'Fenster-IR' zurückgeliefert.
Die Ergebnisse können als ASCII-Datei heruntergeladen werden. Dazu dient der Link 'Output', der nach erfolgter Berechnung ebenfalls auf der rechten oberen Seite erscheint.

Home | Druckversion

IR TDD-Version 1.0

IR TDD-Version 1.0 > Dokumentation

Benutzerhandbuch: lokale Anwendung

Dieses Dokument beschreibt, wie das Programm TDD_IR lokal verwendet wird und welche Optionen zur Verfügung stehen. Bitte beachten Sie auch die Hinweise in Download, Installieren und Kompilieren und FAQ.

Optionen:

Programmaufruf:

Der Programmaufruf erfolgt mittels:

```
tdd_ir -EINGABEFORMAT[t oder i] FILENAME [-OPTIONEN[r, w, s, g, l, c, m, o]]
```

wobei tdd_ir über *./tdd_ir* aufgerufen werden muß, falls es nicht in die Umgebung eingebunden worden ist. Nach Eingabe des Formats wird der Pfad zum Filenamen angegeben. Getestet wurde tdd_ir unter den in verwendete Maschinen aufgeführten Betriebssystemen.

EINGABEFORMAT -i:

Diese Eingabeoption verlangt einen in FASTA formatierten Datensatz. Um Suffixbäume zu erstellen, wird dazu aus tdd_ir das Programm tdd mit der übergebenen Datei aufgerufen. TDD erstellt dann im gleichen Verzeichnis u.a. die 5 Dateien *FILENAME.tdd, FILENAME.tdd.contigs, FILENAME.tdd.leaf, FILENAME.tdd.rmost und FILENAME.tdd.tree*. Mittels diesen wird dann im Programm TDD_IR der IR berechnet. Die Ausgabe-Dateinamen ändern sich, falls die Option -r verwendet wird (s. unten). Die Optionen -r und -s stehen nur bei diesem Eingabeformat zur Verfügung.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta
```

liefert zurück:

```
# Len IR
580074 0.142855
```

EINGABEFORMAT -t:

Diese Eingabeoption verlangt einen mit TDD vorverarbeiteten Datensatz, d.h. als FILENAME muß eine Datei mit der Endung *.tdd* vorliegen, die vorher durch das Programm TDD erstellt wurde. Ebenfalls müssen die anderen 4 Dateien mit den Endungen *.tdd.contigs, .tdd.leaf, .tdd.rmost und .tdd.tree* im selben Verzeichnis liegen. TDD_IR verwendet dann direkt die schon auf Festplatte gespeicherten Suffixbäume. Damit wird natürlich ein Geschwindigkeitsvorteil erreicht, wenn der IR für ein und dieselbe Datei, nur mit unterschiedlichen Optionen (z.B. variierter Fensterlänge) ausgewertet werden soll und dabei die Suffixbäume schon im TDD-Format vorliegen. Den gleichen Effekt erreicht man, wenn man zuerst die Option -i verwendet und dann für weitere Auswertungen die entsprechenden tdd-Dateien mit der Option -t verwendet. Allerdings stehen in der derzeitigen Version die Optionen -r und -s nicht für das Eingabeformat -t zur Verfügung!

Beispiel:

```
$ ./tdd_ir -t l43967.fasta.tdd
```

liefert ebenfalls, nur schneller zurück:

```
# Len IR
580074 0.142855
```

OPTIONEN -r:

Diese Option ruft in TDD_IR eine Methode auf, die aus dem Input-Datensatz im FASTA-Format einen neuen Datensatz erstellt, der für die im Originaldatensatz enthaltenen Sequenzen den jeweiligen Gegenstrang berechnet und in eine neue Datei mit dem Namen *FILENAME.rev* schreibt. Aus dieser werden dann mittels TDD-Aufruf wieder die TDD-Datensätze erstellt. Die TDD-Datei, die dann mit der Option -t wieder neu aufgerufen werden könnte, lautet also *FILENAME.rev.tdd*. Die anderen TDD-Dateien entsprechend. Diese Option ist nur für das Eingabeformat -i verfügbar.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta -r
```

liefert zurück:

```
# Len IR
580074 0.137699
```

OPTIONEN -s:

Diese Option ist ebenfalls nur für das Eingabeformat -i verfügbar. Dabei wird jede in der Datei vorhandene Sequenz erst mittels TDD verarbeitet, bevor dann die Verarbeitung durch TDD_IR ausgeführt wird. In der vorhandenen Version werden die dabei erzeugten TDD-Dateien nur zwischengespeichert, sind also im Nachhinein nicht mehr einzeln aufrufbar.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta.rev -s
```

liefert zurück:

```
>L43967 L43967.1 MYCOPLASMA GENITALIUM G37 COMPLETE GENOME.
# Len IR
580074 0.142855

>L43967 L43967.1 MYCOPLASMA GENITALIUM G37 COMPLETE GENOME. KOMPLEMENT
# Len IR
580074 0.142855
```

OPTIONEN -w :

Die Option -w gibt eine Fenstergröße an, mit der eine Sequenz 'gescannt' wird. Dazu wird an der ersten Position begonnen, und der IR für die Teilsequenz berechnet, die sich innerhalb des Fensters befindet. Im Anschluß daran wird der Beginn des Fensters um 1/10 seiner Größe (bzw. bei Auswahl der Option -c um die entsprechende Länge) weiterverschoben und der IR für die neue Teilsequenz berechnet. Damit lassen sich leicht Bereiche mit prägnanter Repetitivität erkennen. Diese Option ist für beide Eingabeformate verfügbar.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta -w 100000
```

bzw.

```
$ ./tdd_ir -t l43967.fasta.tdd -w 100000
```

liefert zurück:

```
# Len IR
580074 0.142855
>L43967 L43967.1 MYCOPLASMA GENITALIUM G37 COMPLETE GENOME.
50000.5 0.07087019410
60000.5 0.07241391039
70000.5 0.07455525539
```

OPTIONEN -g:

Die Option -g gibt die globalen 'shortest unique substrings' und ihre Positionen zurück. Enthält eine Datei mehrere Sequenzen, wird in der derzeitigen Version einfach durchgezählt, so daß nur innerhalb der ersten Sequenz auch die reellen Positionen zurückgeliefert werden.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta -g
```

bzw.

```
$ ./tdd_ir -t l43967.fasta.tdd -g
```

liefert zurück:

```
# Len IR
580074 0.142855

Globale Shustrings:
Nummer: Position: Laenge:
1 11912 6
2 37970 6
3 60189 6
.
.
```

OPTIONEN -l:

Die Option -l gibt für jede Position die Länge des shortest unique substring zurück. Enthält eine Datei mehrere Sequenzen, wird in der derzeitigen Version einfach durchgezählt, so daß nur innerhalb der ersten Sequenz auch die reellen Positionen zurückgeliefert werden.

Beispiel:

```
$ ./tdd_ir -i l43967.fasta -l
```

bzw.

```
$ ./tdd_ir -t l43967.fasta.tdd -l
```

liefert zurück:

```
# Len IR
580074 0.142855
```

```
Lokale Shustrings:
Position: Shustringlaenge:
1 11
2 12
3 12
.
.
```

OPTIONEN -c :

Die Option -c bietet die Möglichkeit, die durch die Option -w festgelegten Fenster um NUM Positionen weiterzuschieben (statt um 1/10 der Fenstergröße).

Beispiel:

```
$ tdd_ir -i l43967.fasta -w 100000 -c 1000
```

bzw.

```
$ tdd_ir -t l43967.fasta.tdd -w 100000 -c 1000
```

liefert zurück:

```
# Len IR
580074 0.142855
>L43967 L43967.1 MYCOPLASMA GENITALIUM G37 COMPLETE GENOME.
50000.5 0.07087019410
51000.5 0.07162922956
52000.5 0.07173865909
.
.
```

OPTIONEN -m:

Die Option -m erlaubt die Veränderung der Menge an Hauptspeicher, mit der TDD arbeiten soll (s.a. Download, Installieren und Kompilieren). Sie kommt deshalb auch nur im Rahmen des Eingabeformats '-i' zum tragen! Als Default-Wert wurde 400000 KB gewählt. Dies erlaubte z.B. noch eine Verarbeitung von Chromosom 17 des Humangenoms (Einzelstrang/ohne N, näheres s. FAQ) auf einem 1 GB-Rechner. Höhere Werte hatten auf den verwendeten Maschinen nur dort einen Effekt auf TDD, wo mehr als 1 GB (hier: 4 GB) vorhanden war.

Beispiel:

```
> $ tdd_ir -i test.fasta -m 500000
```

Zusammenfassung:

Gegeben sei eine FASTA formatierte Datei, die mehrere Sequenzen und dabei nur die Vorwärtsstränge enthält. Das Ergebnis soll die einzelnen Sequenzen ausgeben, wobei jeweils Vor- und Rückwärtsstrang betrachtet werden soll. Außerdem möchte man die Repetitivität der einzelnen Sequenzen mittels einer Fensteranalyse mit einer Fensterlänge von 1000 aufgeschlüsselt bekommen. Der entsprechende Befehl lautet:

Beispiel:

```
$ tdd_ir -i test.fasta -r -s -w 1000
```

IR TDD-Version 1.0 - FAQ

Dieses Dokument enhält Fragen zu den verwendeten Betriebssystemen und Problemen bei der Auswertung von bestimmten Datensätzen.

Fragen zu TDD_IR:

Welche Betriebssysteme können verwendet werden?

Getestet wurde tdd_ir unter den in verwendeten Maschinen aufgeführten Betriebssystemen.
Ein UNIX-ähnliches System ist auf jeden Fall notwendig, da aus dem C++ Programm heraus auch Systemaufrufe getätigt werden, um z.B. das Programm TDD verwenden zu können. Unter SUSE Linux 10.1 gibt es Probleme beim Kompilieren von TDD mit dem gcc-Kompiler Version 4.1.0, die nicht mehr weiter untersucht wurden. tddbin ist aber lauffähig und TDD_IR selber wird zwar mit einigen Warnmeldungen kompiliert, die aber die Funktionstüchtigkeit des Programms nicht beeinträchtigen.

Welche Datensätze können verwendet und worauf muß dabei geachtet werden?

Wie auf der TDD-Homepage beschrieben, hat TDD Probleme bei der Verwendung von Datensätzen, die z.B. eine große Anzahl von sich wiederholenden N's enthalten. Empfohlen wird daher auch, TDD_IR nur für Datensätze zu verwenden, die vollständig sequenziert wurden. In der derzeitigen Version werden die nichtsequenzierten Bereiche in den IR miteingerechnet, was zu einer Verfälschung des Ergebnisses führen kann. Um zu testen, ob die Sequenz N's enhält, wird die Verwendung eines geeigneten Programmes oder einfach der 'grep'-Befehl empfohlen.
Obwohl TDD lt. TDD-Homepage das ganze Humangenom verarbeiten können müßte (ohne N's), konnte dies auf den verwendeten Maschinen nicht geschafft werden.
Außerdem mußten die Sequenzen in Großbuchstaben vorliegen, damit sie TDD verarbeiten konnte! Wird wie in der Webanwendung die Option '-i' verwendet, wird die Konvertierung der Originaldatei in Großbuchstaben automatisch erledigt (mittels des UNIX-Befehls *tr*). Bitte stellen Sie entsprechende

Schreibrechte zur Verfügung.
Folgende Ergebnisse wurden erreicht, die ausschließlich von der Größe des vorhandenen Hauptspeichers abhängig waren (Humangenomdaten):

- Hauptspeicher: *4 GB*
- Einzelstrang: *Chromosom 4: 179 MB, 187297063 Basen*
- Doppelstrang: *Chromosom 14: 169 MB, 176581170 Basen*
- Maximum -MEM (TDD): *1500000*

sowie auf:

- Hauptspeicher: *1 GB*
- Einzelstrang: *Chromosom 17: 75 MB, 77800220 Basen*
- Doppelstrang: *Chromosom 21: 66 MB, 68340212 Basen*
- Maximum -MEM (TDD): *400000*

Es tritt folgende Fehlermeldung auf:
terminate called after throwing an instance of 'std::bad_alloc'
what(): St9bad_alloc

Es steht nicht genügend Hauptspeicher zur Verfügung. Dies ist zum Einen der Fall, wenn größere als die oben aufgeführten Datensätze verarbeitet werden sollen oder wenn noch andere speicherintensive Anwendungen auf dem Rechner laufen. Diese sollten vor der Verwendung von TDD und TDDIR beendet werden.
TDD liefert ggf. 'partitionierte Suffixbäume' zurück (s. Practical Suffix Tree Construction). Dies ist v.a. vom vorhandenen Hauptspeicher abhängig, aber eine genaue Grenze konnte auch durch mehrere Tests nicht herausgefunden werden. TDDIR kann diese partitionierten Bäume nicht verarbeiten. In diesem Fall wird die Datei *FILENAME.tdd.tree* (s. lokale Anwendung) auch gar nicht von TDD angelegt und deshalb in TDD_IR eine entsprechende Fehlermeldung ausgegeben.

Support:

Bei weiteren Problemen, die ausschließlich mit TDD-IR zusammenhängen, kann der Autor unter bi7719@fh-weihenstephan kontaktiert werden.

Literaturverzeichnis

[ADD+01] Anderle, P, M Duval, S Draghici, A Kuklin, TG Littlejohn, JF Medrano, D Vilanova und MA Roberts: *Gene expression databases and data mining.* Biotechniques., 291(5507):36–44, March 2001.

[AGM+90] Alschul, SF, W Gish, W Miller, EW Myers und Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215:403–410, 1990.

[AJ98] Atkinson, M und M Jordan: *Providing orthogonal persistence for java.* In: *Proceedings of the 20th International Conference on Object-Oriented Programming*, Seiten 383–395, 1998.

[AKO04] Abouelhoda, MI, S Kurtz und E Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2:53–86, 2004.

[BH04] Bedathur, SJ und JR Haritsa: *Engineering a fast online persistent suffix tree construction.* In: *Proceedings of the 20th International Conference on Data Engineering*, Seiten 720–731, 2004.

[CCY] Cheng, LL, D Cheung und SM Yiu: *Approximate string matching in DNA sequences.* In: *In: Proceedings of the 8th International Conference on Database Systems for Advanced Applications.*

[CF99] Crauser, A und P Ferragina: *Research Report: A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory.* Technischer Bericht MPI-I-1999-1-001, Max-Planck Institut für Informatik und Universita di Pisa Dipartimento di Informatica, Saarbrücken and Pisa, I, March 1999.

[E95] E, Ukkonen: *On-line construction of suffix-trees.* Algorithmica, 14(3):249–260, 1995.

[FVK06] Fischer, J, H Volker und S Kramer: *Optimal String Mining Under Frequency Constraints.* In: *PKDD*, Seiten 139–150, 2006.

[GK97] Giegerich, R und S Kurtz: *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction.* Algorithmica, 19(3):331–353, 1997.

[GKS03] Giegerich, R, S Kurtz und J Stoye: *Efficient implementation of lazy suffix trees.* Softw. Pract. Exper., 33(11):1035–1049, 2003.

[Gus97] Gusfield, D: *ALGORITHMS ON STRINGS, TREES, AND SEQUENCES: Computer Science and Computational Biology.* Cambridge University Press, Cambridge, 1997.

[HAI01] Hunt, E, MP Atkinson und RW Irving: *A database index to large biological sequences.* The VLDB Journal, 23(2):139–148, 2001.

[HAI02] Hunt, E, MP Atkinson und RW Irving: *Database indexing for large dna and protein sequence collections.* The VLDB Journal, 11(3):256–271, 2002.

[Heu05] Heun, V: *Skriptum zur Vorlesung Algorithmen auf Sequenzen.* Doktorarbeit, Ludwig-Maximilians-Universität, Institut für Informatik, München, March 2005.

[HPMW05] Haubold, B, N Pierstorff, F Möller und T Wiehe: *Genome comparison without alignment using shortest unique substrings.* BMC Bioinformatics, 6:123, 2005.

[Hun04] Hunt, E: *Indexed Searching on Proteins Using a Suffix Sequoia.* IEEE Data Eng. Bull., 27(3):24–31, 2004.

[HW06] Haubold, B und T Wiehe: *How repetitive are genomes?* BMC Bioinformatics, 7:541, 2006.

[IT99] Itoh, H und H Tanaka: *An efficient method for in memory construction of suffix arrays.* In: *In Proceedings of the sixth Symposium on String Processing and Information Retrieval, SPIRE '99*, Seiten 81–88, 1999.

[Jap04] Japp, R: *The Top-Compressed Suffix Tree. A Disk-Resident Index for Large Sequences.* Technischer Bericht TR-2004-165, Department of Computing Science, University of Glasgow, G12 8QQ, UK, Juli 2004.

[KS99] Kurtz, S und C Schleiermacher: *REPuter - fast computation of maximal repeats in complete genomes.* Bioinformatics, 29:426–427, 1999.

[Kur99] Kurtz, S: *Reducing the Space Requirement of Suffix Trees.* Softw. Pract. Exper., 29(13):1149–1171, 1999.

[LLB+01] Lander, ES, LM Linton, B Birren, C Nusbaum, MC Zody, J Baldwin, K Devon, K Dewar, M Doyle, W FitzHugh et al.: *Initial sequencing and analysis of the human genome.* Nature, 409(6822):860–921, 2001.

[LS99] Larsson, NJ und K Sadakane: *Faster suffix sorting.* Technischer Bericht LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computing Science, Lund University, Sweden, 1999.

[McC76] McCreight, EM: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2):262–272, 1976.

[MF04] Manzini, G und P Ferragina: *Engineering a lightweight suffix array construction algorithm.* Algorithmica, 40(1):33–50, 2004.

[MM93] Manber, U und EW Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5):935–948, 1993.

[MULK+03] Mi, H, R Ulitsky-Lazareva, B andLoo, A Kejariwal, J Vandergriff, S Rabkin, N Guo, A Muruganujan, O Doremieux, MJ Campbell, H Kitano und PD Thomas: *PANTHER: A Library of Protein Families and Subfamilies Indexed by Function.* Genome Res., 13:2129–2141, 2003.

[NBYT01] Navarro, G, R Baeza-Yates und J Tariho: *Indexing methods for approximate string matching.* IEEE Data Eng. Bull., 24(4):19–27, 2001.

[NW70] Needleman, SB und CD Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biologie, 48:443–453, 1970.

[PM] Phoophakdee, B und Zaki MJ: *Genome-scale Disk-based Suffix Tree Indexing.* In: *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data.*

[PT05] Patel, JM und Y Tian: *Practical Methods for Constructing Suffix Trees.* URL, http://www.eecs.umich.edu/~ytian/suffix_tree.pdf, 2005.

[RAM] *Arbeitsspeicher.* URL, http://www.de.wikipedia.org/wiki/Arbeitsspeicher.

[Rep] *RepeatMasker.* URL, http://www.repeatmasker.org.

[Sew00] Seward, J: *On the performance of BWT sorting algorithms.* In: *In DCC: Data Compression Conference*, Seiten 173–182, 2000.

[SM81] Smith, TF und Waterman MS: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147:195–197, 1981.

[SS03] Schurmann, KB und J Stoye: *Suffix-tree construction and storage with limited main memory.* Technischer Bericht, University fo Bielefeld, Germany, 2003.

[TCK+03] Thomas, PD, MJ Campbell, A Kejariwal, H Mi, B Karlak, R Daverman, K Diemer, A Muruganujan und A Narechania: *PANTHER: A Library of Protein Families and Subfamilies Indexed by Function.* Genome Res., 13:2129–2141, 2003.

[THP04] Tata, S, RA Hankins und JM Patel: *Practical suffix tree construction.* In: *Proceedings of 30th International Conference on Very Large Data Bases*, Seiten 36–47, 2004.

[TKC+06] Thomas, PD, A Kejariwal, MJ Campbell, H Mi, K Diemer, N Guo, I Ladunga, B Ulitsky-Lazareva, A Muruganujan, S Rabkin, JA Vandergriff und O Doremieux: *How repetitive are genomes?* BMC Bioinformatics, 7:541, 2006.

[TRA] *Transposon-Mutagenese.* URL, http://www.genetik.uni-bielefeld.de/Genetik/phyto/docs/Aufbaumodul-Theorie1-4.pdf.

[TTHP05] Tian, Y, S Tata, RA Hankins und JM Patel: *Practical methods for constructing sufix trees.* The VLDB Journal, 14:281–299, 2005.

[VAM+01] Venter, JC, MD Adams, EW Myers, PW Li, RJ Mural, GG Sutton, HO Smith, M Yandell, CA Evans, RA Holt et al.: *The sequence of the human genome.* Science, 291(5507):1304–1351, 2001.

[VHS01] Volfovsky, N, BJ Haas und SL Salzberg: *A clustering method for repeat analysis in DNA sequences.* Genome Biology, 2:0027.1–0027.11, 2001.

[Wei73] Weiner, P: *Linear pattern matching algorithms.* In: *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, Seiten 1–11. The University Press of Iowa, 1973.