Algorithmen und Softwarewerkzeuge für vergleichende Genomanalyse

Mohamed Ibrahim Abouelhoda
Fakultät für Informatik, Universität Ulm
mibrahim@informatik.uni-ulm.de

Abstract: Vergleichende Genomanalyse ist ein relativ neues Gebiet der Bioinformatik, das durch die Verfügbarkeit einer immer größer werdender Zahl sequenzierter Genomen an Bedeutung gewinnt. Die vorliegende Dissertation präsentiert Algorithmen und Softwarewerkzeuge, mit denen mehrere Genome effizient verglichen werden können. Die vorgestellten Algorithmen lösen bisher offene Probleme der theoretischen Bioinformatik. In der Praxis reduzierten wir sowohl die Rechenzeit als auch den Platzbedarf für das Vergleichen der großen Genome.

1 Algorithmischer Genomvergleich

DNA ist ein polymeres Molekül, bestehend aus einer Folge chemischer Bausteine, so genannten Nukleotiden vom Typ A, C, G, oder T. Das Wort „Genom" bezieht sich auf den kompletten DNA-Inhalt einer Zelle in einem Organismus; jede Zelle beinhaltet eine Kopie des Genoms. (In der Informatik-Fachsprache ist das Genom ein Zeichenkette „string" über einem Alphabet, das aus vier Zeichen besteht.) Mittels Sequenzierungstechnologie, die in den Siebzigern des letzten Jahrhunderts entwickelt und in den letzten Jahren drastisch verbessert wurde, ist es möglich, die Folge von Nukleotiden eines Genoms zu bestimmen. Das erste vollständig sequenzierte Genom–das Bakteriengenom *H. influenza*–wurde 1995 veröffentlicht. Nach nur wenigen Jahren, 2001, wurde der erste Sequenzierungsentwurf des menschlichen Genoms publiziert. Bislang (März 2006) enthalten die öffentlichen Datenbanken Genome von mehr als 350 (260 im April 2005) Organismen und ungefähr 1100 Viren. Zusätzlich laufen aktuell mehr als 1500 Sequenzierung-Projekte.

Wissenschaftler des Humangenomprojekts konnten viele der unerwarteten Ergebnisse im menschlichen Genom durch den Vergleich mit bekannten Sequenzen anderer Organismen erklären. Durch die stetig steigende Zahl verfügbarer Genomvergleichsdaten kann immer besser nachvollzogen werden, wie Genome sich organisieren, funktionieren, replizieren und entwickeln. Auch die Biotechnologie und Pharmazie erwartet durch den Einsatz von Vergleichsmethoden Aufgaben schneller lösen zu können, wie etwa die Steigerung der Produktivität von Organismen oder die Identifizierung von Protein- oder DNA-Segmenten, auf denen Arzneimittel wirken.

Bevor die Sequenzen gesamter Genome zur Verfügung standen, beschäftigten sich Informatiker in der Biologie hauptsächlich mit der Verwaltung kleiner Teilsequenzen. Auf diesen konnten bekannte Vergleichsverfahren, die quadratische Laufzeit benötigen, in ange-

messener Zeit ausgeführt werden. Die Analyse ganzer Genome, die eine Größenordnung von mehreren Milliarden Basenpaaren haben, erfordert von Informatikern die Entwicklung völlig neuer Algorithmen und Softwarewerkzeuge, die diese enorme Datenmengen vergleichen können. Dies ist zugleich die Motivation für den *Algorithmischen Genomvergleich* als wachsendes Gebiet der Bioinformatik.

2 Die Dissertation: Eine Übersicht

Die vorliegende Dissertation präsentiert Algorithmen und Werkzeuge für die Analyse und den Vergleich großer genomischer Sequenzen [Abo05]. Die dazu entwickelten Algorithmen können in zwei Kategorien eingeteilt werden: Algorithmen, die große genomische Sequenzen anhand einer Indexdatenstruktur analysieren und Algorithmen, welche mehrere Genome vergleichen.

Das Indizieren der genomischen Sequenzen ist ein primärer Schritt, um sie vergleichen zu können. Zu diesem Zweck führten wir eine neue Indexdatenstruktur, genannt *erweitertes Suffix-Array, „Enhanced Suffix Array (ESA)"*, als eine effizientere Datenstruktur als der *Suffix-Baum*, ein. Der Suffix-Baum ist eine bewährte Indexdatenstruktur, die für viele Anwendungen in der Sequenzanalyse verwendet wird [Gus97]. Allerdings stellt der große Platzbedarf des Suffix-Baumes einen gravierenden Engpass der darauf basierenden Anwendungen dar; er benötigt ca. $20n$ Bytes Speicher (20 Bytes pro Zeichen der Eingabesequenz der Länge n). Das Suffix-Array ist eine andere Indexdatenstruktur, die vor ca. 13 Jahren [MM93] vorgestellt wurde und nur $4n$ Bytes beansprucht. Jedoch wurde das Suffix-Array lange nicht beachtet, da die darauf basierenden Algorithmen für nicht so effizient gehalten wurden. Ein Grund dafür war, dass der einzig bekannte Algorithmus auf Suffix-Arrays, der Fragen des Typs „ist die Zeichenkette P der Länge m ein Teilwort des Textes S der Länge n" beantwort, Zeitkomplexität $O(m + \log n)$ hat, während der entsprechende Algorithmus auf dem Suffix-Baum in $O(m)$ Zeit läuft. Gravierender war jedoch, dass man nicht wusste, ob man alle Probleme, die durch einen Suffix-Baum lösbar sind, auch mit einem Suffix-Array lösen kann; und falls dies möglich ist, ob die Effizienz der Algorithmen mit denen des Suffixbaumes vergleichbar sind. Unser erweitertes Suffix-Array lieferte die Antworten auf diese Fragen.

Das erweiterte Suffix-Array einer Sequenz S besteht aus dem bekannten *Suffix-Array* von S und Zusatzinformation, die in weiteren Tabellen abgelegt ist. Mit Hilfe dieser Tabellen kann jeder Algorithmus auf dem Suffix-Baum durch einen gleichwertigen auf dem ESA ohne Verlust der Effizienz systematisch ersetzt werden; d. h. unsere Algorithmen haben die selbe Zeitkomplexität wie jene für den Suffix-Baum. Beispielsweise kann die oben erwähnte Frage „ist P in S?" jetzt in linearer Zeit auf dem ESA beantwortet werden. Außerdem präsentieren wir eine Reihe von Algorithmen, die ausschließlich aus den Eigenschaften des ESA Nutzen ziehen. Abhängig von der Anwendung benötigt das ESA in der Praxis $5n$ bis $7n$ Bytes. Trotz dieser Speicherreduzierung zeigen experimentelle Ergebnisse, dass unsere ESA-Algorithmen nicht nur speichereffizienter, sondern auch schneller als die entsprechenden Suffix-Baum-Algorithmen sind.

Die zweite Kategorie von Algorithmen in dieser Dissertation enthält diejenigen, die große genomische Sequenzen vergleichen. Das Genomvergleichproblem kann man so definieren: Gegeben sind mehrere Genome; finde die homologen (ähnlichen) Gebiete in diesen Genomen. Als *ähnliche* bezeichnen wir jene Gebiete, die, abgesehen von wenigen Zeichen-Änderungen (Mutationen: Ein Nukleotid wurde gelöscht, eingefügt oder ersetzt) identisch sind. Gemäß unserem Überblick in der Dissertation ist die *Anker-basierte Strategie* die erfolgreichste Strategie. Durch sie sind wir im Stande, uns mit großen Sequenzen zu befassen. Sie besteht aus drei Phasen:

1. In den gegebenen Genomen werden Fragmente (kurze ähnliche Gebiete) bestimmt. Wir argumentieren jetzt, dass Gebiete, in denen kein Fragment liegt, keine Ähnlichkeit haben und somit außer Betracht gelassen werden können. Diejenigen, in denen Fragmente liegen, sind Homologie-Kandidaten und werden weiter analysiert.
2. Bestimmte Mengen von nichtüberlappenden Fragmenten, sog. *Anker*, werden bestimmt. Die Nichtankerfragmente entstehen eher zufällig und werden von weiteren Analysen ausgeschlossen.
3. Ein Standard-Alignment-Algorithmus wird auf die Gebiete zwischen den Ankern angewandt. Dieser Schritt liefert ähnliche Gebiete auf Zeichenebene. Damit sollen die wenigen Zeichenänderungen erklärt werden.

In der Dissertation verbesserten und erweiterten wir die Anker-basierte Strategie in mehreren Punkten: Als erstes wurde gezeigt, dass wir in der ersten Phase verschiedene Arten von Fragmenten effizient mit dem Suffix-Array berechnen können. Dann präsentierten wir *Chaining-Algorithmen*, um in Phase zwei die besten Ketten („highest-scoring chains") von kollinearen nichtüberlappenden Fragmenten zu bestimmen. Dabei haben wir Techniken aus der *Algorithmischen Geometrie* benutzt. Somit erzielten wir eine Komplexität, die jene aller bisher verfügbaren Algorithmen übertraf. Tatsächlich lösten unsere Ergebnisse ein 10 Jahre offenes Problem [MM95]. Experimente zeigten, dass unsere Algorithmen so effizient sind, dass sie Millionen von Fragmenten von mehreren Genomen in wenigen Minuten behandeln können. Schließlich präsentierten wir einige Varianten der Chaining-Algorithmen, um verwandte Genomvergleich-Aufgaben zu lösen, wie etwa die Abbildung einer cDNA-Datenbank auf ein Genom und das Vergleichen von Entwurfsgenomen.

Teile unseres Beitrags bezüglich des erweiterten Suffix-Arrays sind in [AKO02, AKO04, AOK02] veröffentlicht worden. Andere Teile bezüglich der Chaining-Algorithmen sind in [AO03b, AO03a, AO05] veröffentlicht worden. Die Implementierung unserer Algorithmen und das System, in dem sie integriert sind, werden in [AO04] beschrieben. Außerdem erschienen zwei Buchkapitel im „Handbook of Computational Molcular Biology" [AKO06, OA06]. Erwähnenswert ist, dass unsere veröffentlichten Datenstrukturen und Algorithmen jetzt in verschiedenen Softwarewerkzeugen, wie PROBEmer [ELD03] und EMAGEN [DYM04] verwendet werden.

Im folgenden Abschnitt werden wir die Idee unseres erweiterten Suffix-Arrays einführen. Abschnitt 4 skizziert unsere Chaining-Algorithmen. Schließlich präsentieren wir in Abschnitt 5 kurz unser Softwaresystem CHAINER und zeigen, wie dort die Algorithmen der Abschnitte 3 und 4 integriert wurden, um Genome zu vergleichen.

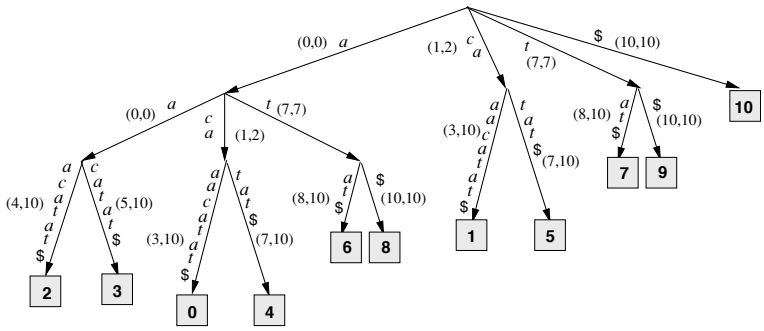

Abbildung 1: Der Suffix-Baum für $S = acaaacatat\$$. Ein Paar (i, j) repräsentiert ein Teilwort $S[i..j]$ entsprechend der jeweiligen Kantenmarkierung.

3 Das erweiterte Suffix-Array

Sei $S[0..n-1]$ eine Zeichenkette der Länge n über einem geordneten Alphabet Σ. Sei $S[i]$ das i-te Zeichen in S, und sei $S[i..j]$ ein *Teilwort*, das an der Position i in S anfängt und an der Position j endet, $0 \leq i \leq j < n$. Das Teilwort $S[i..n-1]$ ist das i-te Suffix von S und wird mit $S(i)$ bezeichnet. Ein Genom ist eine Zeichenkette über $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$.

Ein *Suffix-Baum* für die Zeichenkette $S\$$ ist ein gewurzelter Baum mit $n+1$ Blättern. Das Zeichen $\$$ ist nicht in S enthalten und ist lexikographisch größer als jedes Zeichen in Σ. Jeder innere Knoten hat mindestens zwei Kinder, und jede Kante wird mit dem Teilwort $S[i..j]$ markiert. Die Markierungen zweier Kanten, die zum selben Vaterknoten inzident sind, dürfen nicht mit dem selben Zeichen beginnen. Die Konkatenation der Kantenmarkierungen auf dem Pfad von der Wurzel bis zu einem beliebigen Blatt i stellt genau das Teilwort $S[i..n-1]\$$ dar, das dem i-ten Suffix entspricht, $0 \leq i \leq n$. Die Kantenmarkierungen werden nicht explizit als Zeichenketten gespeichert, sondern durch ein Positionenpaar (i, j), das dem Teilwort $S[i \ldots j]$ entspricht, vertreten. Als Beispiel zeigt Abbildung 1 den Suffix-Baum für die Zeichenkette $S = acaaacatat\$$.

Der Suffix-Baum kann in linearer Zeit und mit linearem Speicherplatzbedarf aufgebaut werden [Gus97]. Alle auf dem Suffix-Baum basierenden Anwendungen können in drei Klassen eingeteilt werden. Anwendungen, die

- eine Bottom-up-Traversierung des Suffix-Baums verwenden,
- eine Top-down-Traversierung des Suffix-Baums durchführen oder
- ein Traversierung des Suffix-Baums durch Suffix-Links (Suffix-Links sind Zeiger, die zwei bestimmte Knoten des Baums verbinden) verwenden.

Das *Suffix-Array* ist eine speichereffizientere Datenstruktur als der Suffix-Baum [MM93]. Das Suffix-Array für $S\$$, das wir mit `suftab` bezeichnen, ist eine Tabelle der Länge $n+1$, die aller Suffixe (repräsentiert durch die Suffixnummer) von $S\$$ in lexikographischer

i	suftab	lcptab	$S(\text{suftab}[i])$
0	2	0	$aaacatat\$$
1	3	2	$aacatat\$$
2	0	1	$acaaacatat\$$
3	4	3	$acatat\$$
4	6	1	$atat\$$
5	8	2	$at\$$
6	1	0	$caaacatat\$$
7	5	2	$catat\$$
8	7	0	$tat\$$
9	9	1	$t\$$
10	10	0	$\$$

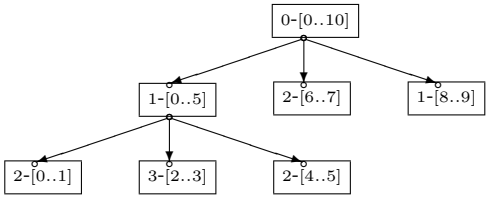

Abbildung 2: Links: Der Suffix-Array der Zeichenkette $S = acaaacatat\$$. Der Suffix-Array wird mit der lcp-Tabelle augmentiert. Rechts: Der lcp-Intervall-Baum von S.

Reihenfolge enthält; siehe dazu Abbildung 2. Das Suffix-Array benötigt nur $4n$ Bytes in seiner Grundform und kann in linearer Zeit konstruiert werden [Gus97, KS03].

Wenn wir für jeden inneren Knoten des Suffix-Baumes die Kanten gemäß ihrer Markierung lexikographisch von links nach rechts sortieren, dann haben die Suffixe an den Blättern dieselbe Anordnung wie die Suffixe im entsprechenden Suffix-Array; siehe Abb. 1 und 2. Allerdings hat das Suffix-Array weniger Strukturinformation als der Suffix-Baum, so dass es nicht klar ist, ob ein Algorithmus, der einen Suffix-Baum verwendet, durch einen auf dem Suffix-Array basierenden Algorithmus ersetzt werden kann. Im Folgenden zeigen wir, dass Probleme, welche mit Suffix-Bäumen gelöst werden können auch ohne Effizienzverlust mit Suffix-Arrays, die wir mit Zusatzinformation erweitern, gelöst werden können. Deshalb nennen wir unsere Datenstruktur *erweitertes Suffix-Array (ESA)*.

Die wesentlichste Erweiterung des Suffix-Arrays ist die *lcp-Tabelle*, die wir als lcptab bezeichnen. Die lcp-Tabelle ist ein Array ganzer Zahlen zwischen 0 und n. $\text{lcptab}[i]$ sei die Länge des längsten gemeinsamen Präfixes von $S(\text{suftab}[i-1])$ und $S(\text{suftab}[i])$ für $1 \leq i \leq n$ ($\text{lcptab}[0] = 0$). Die lcp-Tabelle kann in linearen Zeit berechnet werden und benötigt $4n$ Bytes Speicher. Allerdings haben wir gezeigt, dass die lcp-Tabelle in der Praxis nur $1n$ Bytes benötigt. Unsere Experimente zeigen, dass diese Platzverminderung keinen Effizienzverlust für die betrachteten Algorithmen zur Folge hat.

Definition 3.1 Ein Intervall $[i..j]$ des ESA von S, $0 \leq i < j \leq n$, ist ein *lcp-Intervall des lcp-Werts* ℓ (kurz ein ℓ-Intervall oder ℓ-$[i..j]$) wenn

1. $\text{lcptab}[i] < \ell$,
2. $\text{lcptab}[k] \geq \ell$ für all k mit $i+1 \leq k \leq j$,
3. $\text{lcptab}[k] = \ell$ für mindestens einen k mit $i+1 \leq k \leq j$,
4. $\text{lcptab}[j+1] < \ell$.

Wenn $[i..j]$ ein ℓ-Intervall ist, so dass $\omega = S[\text{suftab}[i]..\ \text{suftab}[i] + \ell - 1]$ das längste gemeinsame Präfix der Suffixe $S(\text{suftab}[i]), S(\text{suftab}[i+1]), .., S(\text{suftab}[j])$ ist, dann wird

$[i..j]$ das ω-Intervall genannt. Jeder Index k, $i+1 \leq k \leq j$, mit $\mathsf{lcptab}[k] = \ell$ wird ℓ-Index genannt.

Ein Beispiel: In Abbildung 2 ist das Intervall $[0..5]$ ein 1-Intervall, weil $\mathsf{lcptab}[0] = 0 < 1$, $\mathsf{lcptab}[5+1] = 0 < 1$, $\mathsf{lcptab}[k] \geq 1$ für alle k mit $1 \leq k \leq 5$ und $\mathsf{lcptab}[2] = 1$ gilt. Außerdem ist 1-$[0..5]$ das a-Intervall (alle Suffixe teilen das Präfix $\omega = a$) und die ℓ-Indizes dieses Intervalls sind 2 und 4, da $\mathsf{lcptab}[2] = \mathsf{lcptab}[4] = 1$.

Definition 3.2 Ein m-Intervall $[l..r]$ ist in einem ℓ-Intervall $[i..j]$ eingebettet, wenn es ein Teilintervall von $[i..j]$ ist und $m > \ell$. Das ℓ-Intervall $[i..j]$ wird dann das *einschließende Intervall* von $[l..r]$ genannt. Wenn es kein Intervall gibt, das in $[i..j]$ eingebettet ist und $[l..r]$ einschließt, dann ist $[l..r]$ das *Kindintervall* von $[i..j]$.

Diese Eltern-Kind-Beziehung bildet einen konzeptionellen (oder virtuellen) Baum, den wir den *lcp-Intervall-Baum* des Suffix-Arrays nennen. Die Wurzel dieses Baums ist das 0-Intervall $[0..n]$; siehe Abbildung 2 (rechts). Der lcp-Intervall-Baum ist im Wesentlichen der Suffix-Baum ohne Blätter (genauer: es gibt eine Bijektion zwischen den Knoten des lcp-Intervall-Baumes und den inneren Knoten des Suffix-Baumes). Diese Blätter sind aber implizit enthalten: Jedes Blatt im Suffix-Baum, das dem Suffix $S(\mathsf{suftab}[l])$ entspricht, kann durch ein *Singleton-Intervall* $[l..l]$ vertreten werden. Das Elternintervall solch eines Singleton-Intervalls ist das kleinste lcp-Intervall $[i..j]$ mit $l \in [i..j]$. In Abbildung 2 sind etwa $[0..1]$, $[2..3]$ und $[4..5]$ Kindintervalle von $[0..5]$. Das Intervall $[0..1]$ hat zwei Singleton- Kindintervalle: $[0..0]$ und $[1..1]$.

Wir haben in der Dissertation gezeigt, dass jede Bottom-up-Traversierung eines Suffix-Baumes durch einen linearen Durchlauf des entsprechenden Suffix-Arrays, das durch suftab und lcptab erweitert wurde, simuliert werden kann. Wir haben auch gezeigt, dass einige Probleme allein durch die Eigenschaften des lcp-Intervall-Baumes gelöst werden können. Für die Top-down-Traversierung über den entsprechenden Suffix-Baum haben wir eine Extratabelle, die sog. *Kinder-Tabelle*, eingeführt. Mittels dieser Tabelle können wir uns von einem Eltern-Intervall bis zu seinen Kindintervallen in konstanter Zeit bewegen und entsprechend die Top-down-Traversierung simulieren. Eine unmittelbare Folge dieses Ergebnisses ist, dass das exakte Mustervergleich-Problem („Ist P in S?") in der selben Komplexität wie der des Suffix-Baumes gelöst werden kann. Um die Traversierung der Suffix-Links zu simulieren, führten wir die *Suffix-Link-Tabelle* ein, die es uns in konstanter Zeit ermöglicht, von einem Intervall im Suffix-Array zu seinem Suffix-Link-Intervall zu springen. Jede der Kinder- und Suffix-Link-Tabellen benötigt in der Theorie $4n$ Bytes Speicher (in der Praxis reichen $1n$ Bytes). Wir zeigten in der Dissertation, wie diese Simulationen für eine große Zahl von Anwendungen funktioniert.

Die Effizienz unseres ESA wollen wir mittels eines Beispiels demonstrieren: das Berechnen von „*maximal unique matches (MUMs)*". Gegeben seien die Sequenzen S_1 und S_2. Ein „*maximal exact match (MEM)*" ist ein Tupel (ℓ, i_1, i_2), das das Teilwort $S_1[i_1..i_1 + \ell - 1] = S_2[i_2..i_2 + \ell - 1]$ repräsentiert, so dass $S_1[i_1 - 1] \neq S_2[i_2 - 1]$ und $S_1[i_1 + \ell] \neq S_2[i_2 + \ell]$ gilt; d. h. die zwei Teilworte sind identisch und können nicht nach links und rechts erweitert werden. Dieses MEM wird MUM genannt, wenn das Teilwort genau einmal in S_1 und einmal in S_2 vorkommt. Für k Sequenzen werden MEMs/MUMs *multi-*

			unique-match		*esamum*	
genome pair	*total size*	#MUMs	*time*	*space*	*time*	*space*
E. coli 2	10,107,957	10817	30.7	472	0.69	62
Yeast 2	24,690,687	2536	118.2	1154	0.66	144
Human 2	67,739,601	217014	430.1	3165	2.34	389

Tabelle 1: Laufzeiten (in Sekunden) und Platzverbrauch (in Megabytes) um MUMs der Länge ≥ 20 zu berechnen. Die Spalte „#MUMs" enthält die Zahl der MUMs.

MEMs/multi-MUMs genannt und analog dazu definiert. Wir haben gezeigt, dass MUMs durch das erweiterte Suffix-Array von $S = S_1 \# S_2$ (S ist die Konkatenation von S_1, einem Trennzeichen, das nicht in S_1 und S_2 vorkommt, und S_2) berechnet werden kann. Tabelle 1 zeigt den Platzverbrauch und die Laufzeit unseres Algorithmus *esamum* und des Algorithmus *unique-match*, der auf dem Suffix-Baum beruht. Für diese Anwendung wird der Speicher um ca. 80% reduziert und die Laufzeit drastisch beschleunigt.

4 Die Chaining-Algorithmen

In der Anker-basierten Strategie werden die eben vorgestellten MUMs/MEMs (multi-MUMs/-MEMs) als Fragmente benutzt. Nun stellt sich die Aufgabe, aus diesen Fragmenten möglichst viele kolineare nichtüberlappende Fragmente –die Anker– zu bestimmen.

Seien k genomische Sequenzen $S_1, .., S_k$ gegeben. Jedes Fragment, das aus den Teilworten (Segmenten) $S_1[l_1 \ldots h_1], .., S_k[l_k \ldots h_k]$ zusammengesetzt ist, wird durch ein Hyper-Rechteck in $\mathbb{R}^k$ repräsentiert. Das Hyper-Rechteck ist durch zwei Eckpunkte $beg(f) = (l_1, \ldots, l_k)$ und $end(f) = (h_1, \ldots, h_k)$ definiert; siehe dazu Abbildung 3. Zwei Fragmente sind kollinear, wenn in allen Genomen die jeweiligen Segmente gleich angeordnet sind; die Fragmente 1 und 4 in Abbildung 3 sind kollinear, 4 und 7 aber nicht. Zwei Fragmente überlappen, wenn ihre Segmente in einem der Genome überlappen; die Fragmente 3 und 4 in Abbildung 3 überlappen, 1 und 4 überlappen nicht. Der *Score* einer Kette von Fragmenten $f_1, f_2, \ldots, f_t$ ist bestimmt durch $\Sigma_{i=1}^{t} f_i.weight - \Sigma_{i=1}^{t-1} g(f_i, f_{i+1})$, wobei $f.weight$ das Fragment-Gewicht (dieses hängt linear von der Länge von f ab) und g eine Lücken-Kostenfunktion ist. Die Lücken-Kosten, die wir verwenden, werden bezüglich der Anfangs- und Endpunkte der Fragmente definiert. In der Dissertation haben wir gezeigt, wie man das Chaining-Problem für Lücken-Kosten in verschiedenen Metriken lösen kann. Etwa in der L_1-Metrik: Die Lücken-Kostenfunktion ist $g(f_i, f_{i+1}) = \Sigma_{j=1}^{k} (f_{i+1}).x_j - (f_i).x_j$, wobei $beg(f_{i+1}).x_j$ die Position ist, an der das Fragment f_{i+1} im Genom G_j anfängt, und $end(f_i).x_j$ die Position ist, an der Fragment f_i in G_j endet. Wir haben ein Sweep-Line-Verfahren vorgestellt, das Range-Maximum-Queries (RMQs) verwendet, um signifikante lokale Ketten, die höchste Score haben, zu finden; Abbildung 3 illustriert ein Beispiel mit lokalen Ketten. Wenn der „range tree" zur Unterstützung der RMQs verwendet wird, benötigt unser Algorithmus $O(m \log^{k-2} m \log\log m)$ Zeit und $O(m \log^{k-2} m)$ Speicher, wobei m die Zahl der k-dimensionalen Fragmente ist. Dieses Ergebnis übertrifft die vorherigen Algorithmen und löst damit ein offenes Problem, das

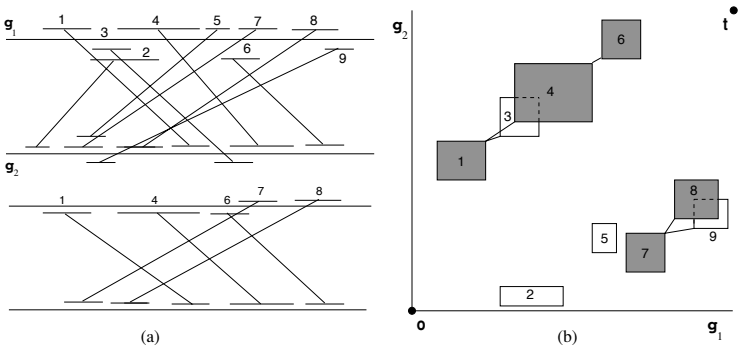

Abbildung 3: (a) Fragmente und (b) ihre geometrische Darstellung. Die x bzw. y-Achse entsprechen dem ersten bzw. zweiten Genom. Es existierten zwei optimale lokale Ketten: $\{1, 4, 6\}$ und $\{7,8\}$.

in [MM95] gestellt wurde. Wenn wir statt des „range tree" einen „*kd*-tree" verwenden, benötigt unser Algorithmus $O((k-1)m^{2-\frac{1}{k-1}})$ Zeit und $O(m)$ Speicher. Unsere Experimente zeigten jedoch, dass der „*kd*-tree" dem „range tree" in der Praxis überlegen ist und Millionen von Fragmenten in wenigen Minuten behandeln kann.

Wir haben auch gezeigt, dass wir durch das Variieren der Lücken-Kostenfunktion und das Eingrenzen der Abfragebereiche der RMQs die Chaining-Algorithmen auf weitere Probleme anwenden können: Etwa die Abbildung von cDNA (Komplementärsequenz einer mRNA) auf Genome und den Vergleich von Draft-Genomen (unvollständige Genome, die Segmente enthalten, deren genaue Anordnung unbekannt ist).

5 Das Softwarewerkzeug CHAINER

Das Diagramm in Abbildung 4 illustriert den Funktionsablauf unseres Systems. Er besteht aus drei Phasen: Als erstes werden die Fragmente erzeugt, dann wenden wir darauf die Chaining–Algorithmen an und schließlich werden die Ergebnisse visualisiert und für weitere Analysen aufbereitet. Jede dieser Phasen besitzt eigene Parameter, die wir durch ein statistisches Modell abschätzen können. Als Beispiel vergleichen wir die Bakteriengenome *M.genitalium* und *M.pneumonia* (580,074 bzw. 816,394 bp). Abbildung 5 (links) zeigt die erzeugte Fragmente (MEMs mit Mindestlänge 12). In derselben Abbildung (rechts) sind die homologen Gebiete identifiziert. Jeder Punkt entspricht einer signifikanten lokalen Kette. Aus der Abbildung kann man leicht erkennen, dass die Anordnung einiger homologen Segmente nicht konserviert wurde. Das Chaining-Verfahren ist schnell: es dauerte ungefähr 8 Sekunden, um 97176 Fragmente zu verketten. Die Erzeugung der Fragmente dauerte ein paar Sekunden (Prozessor Pentium III 933 MHz). Die von uns erzielten Resultate stimmen mit bekannten biologischen Daten überein. Insgesamt können unsere Algorithmen mehrere Genome in wenigen Minuten effizient vergleichen; eine Aufgabe, die Stunden oder sogar Tage mit vorherigen Werkzeugen dauerte.

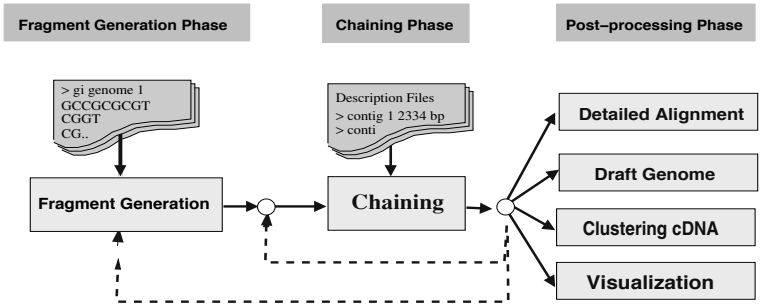

Abbildung 4: Genome sind die Eingabe für das (ESA-basierte) Fragmenterzeugungswerkzeug. Die Feed-Back-Pfeile symbolisieren rekursive Aufrufe: Es ist möglich, kürzere Fragmente zu erzeugen und CHAINER wieder aufzurufen oder die ausgegebenen Ketten weiter zu verketten.

6 Zusammenfassung

Wir haben effiziente Algorithmen und Softwarewerkzeuge zum Genomvergleich präsentiert. Unsere Algorithmen führen diese Vergleiche auf großen Genomen ohne Verwendung großer Rechner-Ressourcen durch. Darüber hinaus wurden diese Algorithmen, vor allem die auf dem Suffix-Array basierenden, so populär, dass sie in anderen Softwarewerkzeugen Verwendung finden. Dies beschränkt sich nicht nur auf den Bereich der Bioinformatik; auch im Gebiet der Text-Verarbeitung und des Text-Mining werden die Algorithmen eingesetzt.

Literatur

[Abo05] M. I. Abouelhoda. Algorithms and A Software System for Comparative Genome Analysis. Ph.d. thesis, University of Ulm, July 2005.

[AKO02] M. I. Abouelhoda, S. Kurtz und E. Ohlebusch. The Enhanced Suffix Array and its Applications to Genome Analysis. In *Proc. of the 2nd Workshop on Algorithms in Bioinformatics*, Jgg. 2452 of *LNCS*, Seiten 449–463. Springer-Verlag, 2002.

[AKO04] M. I. Abouelhoda, S. Kurtz und E. Ohlebusch. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[AKO06] M.I. Abouelhoda, S. Kurtz und E. Ohlebusch. The Enhanced Suffix Arrays. In *Handbook of Computational Molecular Biology*. CRC Press, 2006.

[AO03a] M. I. Abouelhoda und E. Ohlebusch. A Local Chaining Algorithm and its Applications in Comparative Genomics. In *Proc. of the 3rd Workshop on Algorithms in Bioinformatics*, Jgg. 2812 of *LNBI*, Seiten 1–16. Springer Verlag, 2003.

[AO03b] M. I. Abouelhoda und E. Ohlebusch. Multiple Genome Alignment: Chaining Algorithms Revisited. In *Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching, LNCS*, Jgg. 2676 of *LNCS*, Seiten 1–16. Springer Verlag, 2003.

[AO04] M. I. Abouelhoda und E. Ohlebusch. CHAINER: Software for Comparing Genomes. In *Proc. of the 12th International Conference on Intelligent Systems for Molecular Biology/3rd European Conference on Computational Biology*, 2004.

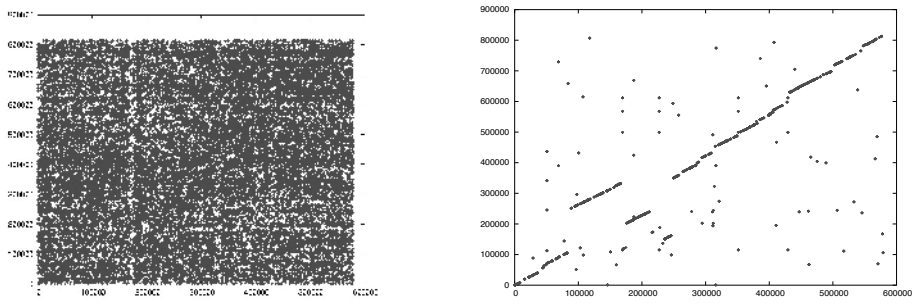

Abbildung 5: Links: Die Fragmente der zwei Genome *M.genitalium* (x-Achse) und *M.pneumonia* (y-Achse). Rechts: Die lokalen Ketten lassen umgeordnete genomische Segmente erkennen.

[AO05] M. I. Abouelhoda und E. Ohlebusch. Chaining Algorithms and applications in comparative genomics. *Journal of Discrete Algorithms*, 3:321–341, 2005.

[AOK02] M.I. Abouelhoda, E. Ohlebusch und S. Kurtz. Optimal Exact String Matching Based on Suffix Arrays. In *Proc. of the 9th International Symposium on String Processing and Information Retrieval*, Jgg. 2476 of *LNCS*, Seiten 31–43. Springer-Verlag, 2002.

[DYM04] J. S. Deogen, J. Yang und F. Ma. EMAGEN: An Efficient Approach to Multiple Genome Alignment. In *Proc. of the 2nd Asia Pacific Bioinformatics Conference (APBC2004)*, Seiten 113–122, 2004.

[ELD03] S. J. Emrich, M. Lowe und A. L. Delcher. PROBEmer: A web-based software tool for selecting optimal DNA oligos. *Nucleic Acids Research*, 31(13):3746–3750, 2003.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[KS03] J. Kärkkäinen und P. Sanders. Simple Linear Work Suffix Array Construction. In *Proc. of the 30th International Colloquium on Automata, Languages and Programming*, Jgg. 2719 of *LNCS*, Seiten 943–955. Springer-Verlag, 2003.

[MM93] U. Manber und E. W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MM95] G. Myers und W. Miller. Chaining Multiple-Alignment Fragments in Sub-Quadratic Time. *Proc. of the 6th annual ACM-SIAM symposium on Discrete algorithms*, Seiten 38–47, 1995.

[OA06] E. Ohlebusch und M. I. Abouelhoda. Chaining Algorithms and applications in comparative genomics. In *Handbook of Computational Molecular Biology*. CRC Press, 2006.

Mohamed I. Abouelhoda wurde 1975 in Kairo (Ägypten) geboren, wo er 1993 das Abitur machte. Ab 1993 studierte er in der Abteilung für biomedizinische Technik der Universität Kairo. 1998 erhielt er den Bachelor und schloss 2001 das Studium mit einem Master ab. 2001 wurde er Doktorand in dem Graduiertenkolleg Bioinformatik der Universität Bielefeld. Er hat sich 2003 der Bioinformatik Arbeitsgruppe der Universität Ulm angeschlossen, wo er 2005 promovierte. Bislang forscht er dort weiter als Postdoc.