Ausarbeitung: Suffix Arrays und Burrows-Wheeler Transformation

Tobias Harrer

19.11.2012

Proseminar, Algorithmen der Bioinformatik

Inhaltsverzeichnis

1 Einleitung und Definitionen

1.1 Einleitung

Die Ausarbeitung zum Thema „Suffix Arrays und Burrows-Wheeler Transformation“ soll einen Überblick über die im Vortrag behandelten Algorithmen zur Suche eines Patterns in Strings mithilfe von Suffix Arrays geben und eine weitere Methode zur Berechnung der BWT mithilfe von Suffix Arrays zeigen.

1.2 Definitionen

- Es seien T ein String der Länge n und p ein Muster der Länge m über dem Alphabet Σ.
- $T_{i,j}$ mit $0 < i \leq j \leq n$ heißt *Substring* von T, der T von Position i bis einschließlich j repräsentiert.
- Der Substring $T_{i,n}$ reicht bis zum einschließlich letzten Zeichen von T und heißt *Suffix*.
- Ein *Suffix Array* ist die Permutation [1, ..., n] der Startindizes i aller, lexikografisch sortierten, Suffixes von T. A[i] ist dann der i-te Eintrag des Suffix Arrays.
- Zusätzlich soll jeder String mit dem Zeichen ”$” enden, wobei für alle $c \in \Sigma$ gilt: $ $< c$.

2 Motivation

Die Vorzüge von Suffix Trees gegenüber der naiven Suchmethode von Pattern in Strings, aber auch gegenüber des Boyer-Moore Algorithmus wurden im vorletzten Vortrag bereits dargestellt. Betrachten wir einen Suffix Tree von T, dessen Kindknoten durch die Kantenbeschriftungen von links nach rechts lexikografisch geordnet sind. Dann entspricht die Ordnung der Blätter von links nach rechts dem Suffix Array A von T [NM07, Abschnitt 3.3]. Da, wie im Folgenden gezeigt, mithilfe von lediglich A (und T) die Suche eines Patterns in T erfolgen kann, erübrigt sich das Abspeichern der gesamten Baumstruktur. Da ein Suffix Array in linearer Zeit erstellt werden kann [NM07, Abschnitt 3.3], ist es nicht praktikabel, dieses aus einem Suffix Tree herzuleiten, da dieser einen relativ großen Speicherplatzbedarf beansprucht. Im Folgenden werden nun Methoden erläutert, wie mit dem String T und dem Suffix Array A, und sogar nur mit A als Selbst-Index von T, ein Pattern p in T gefunden werden kann. Des Weiteren wird die Verwendung von Suffix Arrays in der Burrows-Wheeler Hintransformation besprochen.

3 Effiziente Suche von Pattern in Strings mit Suffix Arrays

3.1 Binäre Suche auf dem Suffix Array

Jedes Pattern p ist, sofern es in T liegt, ein Substring von T, und damit ein Präfix eines Suffixes von T. Da mit T und seinem Suffix Array A alle Suffixes *lexikografisch geordnet* vorliegen und davon jeweils ein Präfix zum Vergleich herangezogen wird, bietet sich die binäre Suche auf A an. Im besten Fall ist das längste gemeinsame Präfix aller Suffixe der leere String, und p hat die Länge 1, dann läuft die binäre Suche von p in $\mathcal{O}(log(n))$ Zeit ab. Nehmen wir aber den ungünstigen Fall T = "bbbbbb", p = "bb", so benötigt die Suche aller Vorkommen von p in T $\mathcal{O}(m \cdot k + m \cdot log(n))$ Zeit, da jeder lexikografische Vergleich $\mathcal{O}(m)$ und das Auffinden der im Suffix Array angrenzenden k Vorkommen von p $\mathcal{O}(m \cdot k)$ Zeit benötigt. Dieser Fall ist vor allem in der Bioinformatik keine Seltenheit, da sich sich das Alphabet bei DNA Sequenzen auf $\Sigma = \{A, T, G, C\}$ beschränkt, n sehr groß sein kann und sich Substrings, also Präfixe von Suffixen, sehr oft wiederholen können. Falls zB alle Vorkommen von Startkodons in einem Genom gesucht werden, erwartet man zumindest bei Gleichverteilung der Nukleobasen $(\frac{1}{4})^3 \cdot n \in \mathcal{O}(n)$ Vorkommen.

3.2 Eine Erweiterung mit Longest Common Prefixes

Endet die binäre Suche im Suffix Array A mit einem Treffer, so müssen die Präfixe der etwa $\frac{1}{64} \cdot n$ erwarteten Suffixe, die ebenfalls mit "ATG" beginnen, nacheinander "links und rechts" vom Treffer mit dem Pattern p verglichen werden. Daher ist es sinnvoll, in dem Intervall [L,R], in dem die binäre Suche mit einem Treffer endete das *longest common prefix* LCP zu bestimmen, so dass $|LCP| = min\{LCP^R(p, T_{A[R],n}), LCP^L(p, T_{A[L],n})\}$ die Länge des längsten gemeinsamen Präfix ist, also das Minimum aus der Länge des LCP^L an der linken und der Länge des LCP^R an der rechten Intervallgrenze. So müssen die Suffixe, die den Treffer in dessen Intervall umgeben, nur mit dem Suffix $p_{|LCP|+1,m}$ von p verglichen werden [Smyth03, Algorithmus 5.3.3]. Dieser Algorithmus kann erweitert werden, indem im Voraus alle möglichen Suchintervalle [L,R] betrachtet werden, und die LCPs aller Suffixes A[L] und A[R] in einem weiteren Array I abgespeichert werden. Es gibt dabei nur 2n-3 mögliche Intervalle und die Größe von I ist auf 3n-8 für n≥5 beschränkt. Dabei kann das Array I bei der Berechnung des Suffix Array A erstellt werden. Da der Zugriff auf I, um das LCP eines bestimmten Intervalls [L,R] zu bestimmen, in konstanter Zeit erfolgt, kann die Suchzeit insgesamt auf $\mathcal{O}(m + log(n))$ reduziert werden [Smyth03, Algorithmus *SAComplex*].

3.3 Backward Search

Da im Normalfall die Patternlänge m erheblich kleiner ist als die Textlänge n, wäre es wünschenswert, eine Suche mit Zeitaufwand proportional zur Länge des Patterns, also in $\mathcal{O}(m)$ durchzuführen. Diese Laufzeit kann mit Suffix Arrays in der sogenannten *Backward*

Search erreicht werden. Ausgehend von Suffix Array A betrachtet man an jeder Stelle A[i] den dem jeweiligen Suffix vorangehenden Buchstaben $T_{A[i]-1}$. Um ein Pattern p = "bcd" zu suchen, beginnt man nicht vorne bei "b", sondern bei am Ende von p bei "d". Um diejenigen Suffixes, die mit "d" beginnen effizient finden zu können, ist es hilfreich, ein zusätzliches Array C der Größe von $|\Sigma|$ zu betrachten, wobei zB C["d"] besagt, wie viele Buchstaben in T lexikografisch kleiner sind als "d", d.h. wie viele Suffixe mit Anfangsbuchstaben kleiner als "d" im Suffix Array "vor" allen mit "d" beginnenden Suffixen stehen. Hat man den Bereich der Suffixes, die mit "d" beginnen, gefunden, stellt sich die Frage, wie in diesem Intervall des Suffix Arrays alle Positionen gefunden werden können, sodass der vorhergehende Buchstabe, also $T_{A[i]-1}$ = "c" ist. Die Antwort ist eine weitere Hilfsfunktion *Occ(c,i)*, die besagt, wie oft "c" in $T_{A[i]-1}$ für alle i' < i vorkommt. *Occ(c,i)* lässt sich berechnen, indem für alle $c \in \Sigma$ ein boolsches Array B erstellt wird, wobei $B^c[i] = true \Leftrightarrow T_{A[i]-1} = c$. Eine Funktion, die in konstanter Zeit die Anzahl von $B^c[i] = true$ für alle i' < i findet, entspricht der Funktion *Occ(c,i)*, die genau wie C[c] in $\mathcal{O}(1)$ eine Zahl zurück gibt. So kann das Intervall [x,y] aller mit "b" beginnenden Suffixes mit zweitem Buchstaben "c" folgendermaßen berechnet werden: x = C["b"] + *Occ("b",i)* und y = C["b"] + *Occ("b",i')*, wobei i die Position ist, an der alle Suffixes mit "c" beginnen und "d" als zweiten Buchstaben haben, und i' die Position ist, an der alle Suffixes mit "c" beginnen und an zweiter Stelle einen Buchstaben lexikografisch größer als "d" haben. Da eine Suche in maximal m Schritten beendet ist und jeder Suchschritt in konstanter Zeit erfolgt ist die Laufzeit in jedem Fall höchstens linear und proportional zur Länge m des Patterns. Die Frage ist, ob im Extremfall T = "aaaaaa", p = "aa", wenn also p annähernd n-mal in T vorkommt, nicht eine Laufzeit von $\mathcal{O}(m \cdot n)$ resultiert. Allerdings liefert die Backward Search in jedem Suchschritt ein Intervall [x,y] zurück, das, sofern $|x - y| > 0$, einen weiteren Suchschritt nach sich zieht. Da aber nach maximal m Schritten p in T gefunden worden ist, und in dem Fall auch nur ein Intervall [x',y'] zurückgegeben wird, ist die absolute Häufigkeit von p in T $|x' - y'| > 0$, was ebenfalls in konstanter Zeit berechenbar ist. Somit ist die Laufzeit der Backward Search von $\mathcal{O}(m)$ garantiert. [NM07, Abschnitt 4.1]. Um den Speicherbedarf der Funktion *Occ("b",i')* noch zu verringern, können auch sogenannte *Wavelet Trees* verwendet werden, wobei die Suchzeit leicht auf $\mathcal{O}(m \cdot |\Sigma|)$ ansteigt [NM07, Abschnitt 4.2]. Nichtsdestotrotz unterbietet die Backward Suche ohne Wavelet Trees die Suchzeit von Suffix Trees $\mathcal{O}(m + k)$ [ucf], insbesondere wenn das Vorkommen k eines Patterns in T sehr groß ist, d.h. $\frac{k}{n} \to 1$.

3.4 Forward Searching

Ein weiterer Ansatz zur Suche eines Patterns p ist die Forward Suche. Sie besitzt zwar mit $\mathcal{O}(m \cdot log(n))$ eine weitaus schlechtere Laufzeit als die Backward Suche, dabei kann aber ein beliebiges p in T *ohne den Zugriff auf T selbst* gefunden werden. Das Suffix Array und eine weitere Permutation Ψ stellen einen *Self Index* von T dar. Während bei der Backward Suche das Intervall [x,y] gesucht war, in dem sich die Suffixes $T_{A[i]-1,n}$ für ein gegebenes Suffix $T_{A[i],n}$ und den vorhergehenden Buchstaben $T_{A[i]-1}$ befinden, nimmt die Forward Suche die entgegengesetzte Richtung. Es wird zusätzlich zum Suffix Array Eintrag A[i] dasjenige $i' = \Psi(i)$ gespeichert, sodass $T_{A[i]} + T_{A[i'],n} = T_{A[i]} + T_{(A[i] mod n)+1,n} = T_{A[i],n}$. Bisher stellt

das jedoch lediglich eine Umkehrung zur Backward Suche dar, wie also könnte man auf T verzichten? Nimmt man der Einfachheit halber an, dass alle $c \in \Sigma$ in T vorkommen, so beschreibt ein Boolsches Array B der Länge $n = |T|$ mit B[i] = true, dass sich an Stelle i ein neuer Buchstabe an erster Stelle des jeweiligen Suffixes befindet. Eine weitere Funktion die Anzahl a von true in B vor Position i bestimmt, weiß man, dass der erste Buchstabe des betrachteten Suffixes der lexikografisch a-te Buchstabe in Σ sein muss, da ja alle $c \in \Sigma$ in T vorkommen sollen. Des Weiteren liefert $\Psi(i)$ die Position des Suffixes $T_{A[i'],n} = T_{(A[i] mod n)+1,n}$. So kann ein Pattern sukzessive in m Schritten konkateniert werden. Da die Grundlage wieder eine binäre Suche ist, liegt der Zeitbedarf bei $\mathcal{O}(m \cdot log(n))$.

4 Suffix Arrays in der BWT

4.1 Die Hintransformation

Die Burrows Wheeler Transformation spielt zwar eine wichtige Rolle in der Textkompression, stellt aber selbst keinen Kompressionsalgorithmus dar, sondern nur eine Vorverarbeitung das Textes T in Form einer Permutation T^{BWT} von T. Dabei kommen in T^{BWT} wahrscheinlicher längere Aneinanderreihungen eines Buchstaben $c \in \Sigma$ vor als in T selbst. Diese sogenannten *Runs* können dann zur effizienten Kompression verwendet werden. Für die Frage, weshalb die Permutation T^{BWT} statistisch die meisten *Runs* enthält, verweise ich auf den Vortrag vom 12.11. Zusätzlich zu in jenem Vortrag vorgestellten Verfahren zur Hin- und Rücktransformation sollen hierbei effizientere Verfahren mit Suffix Arrays vorgestellt werden. (Für ein bildliches Beispiel des Folgenden verweise ich auf meine Folien, da hier Grafiken vermieden werden sollen.) Notiert man alle n Permutationen von T untereinander, beginnend mit T selbst und rotiert T "nach links", so dass jeweils der erste Buchstabe von T an das Ende angehängt wird, so fällt auf, dass das Präfix $T^i_{1,n-i}$ der i-ten Permutation T^i von T dem i-ten Suffix $T_{i,n}$ entspricht. Da in dem Verfahren vom 12.11 anschließend die n Permutatuionen von T alphabetisch sortiert werden, bietet sich eine Verwendung von Suffix Arrays an. Man benötigt also vorerst nur das Suffix Array von T, das in $\mathcal{O}(n)$ Zeit berechnet werden kann [NM07, Abschnitt 3.3] und bereits eine Einsparung gegenüber der lexikographischen Sortierung aller Permutationen, die mindestens $\mathcal{O}(n \cdot log(n))$ Zeit beansprucht. Notiert man nun allerdings zu jedem Eintrag A[i] des Suffix Array A von T das zugehörige Suffix $T_{A[i],n}$, wird deutlich, dass die benötigte letzte Spalte L = T^{BWT} nicht existiert. Dabei ist aber jeder Buchstabe L_i lediglich der Vorgänger $T_{A[i]-1}$ des Buchstaben F_i, wobei die erste Spalte F vollständig bekannt ist. Ist das Suffix Array bekannt, so lässt sich daraus T^{BWT} ganz einfach bestimmen, indem für alle $1 \leq i \leq n$ $T_{A[i]-1}$ konkateniert wird, bis auf den Fall A[i] = 1, wobei das kleinste Zeichen "$" (als nulltes Zeichen von T) an L = T^{BWT} angehängt wird.

5 Zusammenfassung

Im Hauptteil der Ausarbeitung wurden mit der Pattern Suche in Strings durch die binäre Suche mit längsten gemeinsamen Präfixen und die Backward Suche zwei Anwendungen von Suffix Arrays erwähnt, in dem deren Verwendung anstatt der von Suffix Trees von Vorteil ist, sowohl in der Speicher-, als auch in der Laufzeitkomplexität. Des Weiteren wurde auch die Fähigkeit von Suffix Arrays, einen Selbstindex zu darzustellen, besprochen. Schließlich wurde noch die Anwendung in der Burrows-Wheeler Transformation behandelt, wobei die Laufzeit des lexikographischen Sortierens über die Konstruktion von Suffix Arrays in linearer Zeit herabgesetzt wird. Als Ausblick und weitere Anwendung von Suffix Arrays statt Bäumen wird in [Smyth03, S.154 unten] die "Berechnung aller nicht-verlängerbaren sich wiederholenden Substrings in [einem Text] x" erwähnt, was mit Sicherheit ein Thema für einen eigenständigen Vortrag darstellt.

6 Quellen:

- Smyth03: B. Smyth: Computing Pattern in Strings
- NM07: Gonzalo Navarro, Veli Mäkinen: Compressed Full-Text Indexes
- ucf: `http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf` Folie 51, Stand 11.11.12