Proseminar „Algorithmen der Bioinformatik“

Ausarbeitung

Textkompression: Burrows-Wheeler-Transformation

Uli Köhler
12.11.2012

Inhaltsverzeichnis

1 Einleitung

Im Vorliegenden Dokument soll die Bedeutung der Burrows-Wheeler-Transformation (im Folgenden auch BWT genannt) für die Textkompression und deren Anwendung in der Bioinformatik behandelt werden.

Moderne Sequenzierverfahren erzeugen durch Vielfachabdeckung von Genomen große Datenmengen. Um die Rohdaten speichern zu können, müssen spezielle Kompressionsalgorithmen angewandt werden, um den benötigten Speicherplatz auf ein Minimum zu reduzieren.

2 Verlustfreie Kompression

In vielen realen Datensätzen sind starke Redundanzen enthalten. Oft ist die Größe des Alphabets nicht sehr groß (z.B. 36 Alphanumerische Zeichen plus Sonderzeichen in englischen Texten oder 4 Nukleinbasen in DNA- oder RNA-Sequenzen) oder Texte sind stark autokorreliert.

Kompressionsalgorithmen sind Verfahren, die diese Redundanzen eliminieren und dadurch eine platzsparende Speicherung der Daten erlauben, wärend die ursprünglichen unkomprimierten Daten jederzeit vollständig rekonstruiert werden können. Dadurch unterscheiden sich verlustfreie Algorithmen von verlustbehafteten Verfahren, die zwar eine oft deutlich größere Platzeinsparung erzeugen, aber dafür keine vollständige Rekonstruktion erlauben. Verschiedene Kompressionsalgorithmen sind für verschiedene Typen von Datensätzen und Datenbanken geeignet - insbesondere können schnellere Algorithmen meist besser komprimieren als langsamere Algorithmen. Für jeden Anwendungsfall muss so zwischen den Faktoren Kompressionszeit, Dekompressionszeit und Speicherplatzeinsparung abgewogen werden.

3 Die Burrows-Wheeler-Transformation

Die Burrows-Wheeler-Transformation ist selbst kein Kompressionsalgorithmus, sondern ein Verfahren, um einen Text reversibel so zu verändern, dass er sich durch einige Kompressionsalgorithmen deutlich besser komprimieren lässt als der Originaltext (Siehe auch Kapitel 3.6 auf Seite 4)

3.1 Die Burrows-Wheeler-Hintransformation

Die Hintransformation wird detailiert in [Heu03, Abschnitt 6.5] sowie in [BW94, Seite 2 - Algorithmus C] beschrieben. Gegeben sei der zu transformierende Text S. Zuerst wird eine Matrix M der Größe $|S| \times |S|$ initialisiert.

M wird nun mit allen zyklischen Rotationen von S gefüllt - die Reihenfolge (und damit die Richtung der Rotation) spielt hierbei keine Rolle.

Nun werden die Zeilen von M lexikographisch sortiert. In der sortierten Matrix M_S wird nun der Originaltext S gesucht und dessen Index als I gespeichert. Sei L nun der Spaltenvektor der letzten Spalte von M_S. Dann ist das Resultat der Hintransformation das Tupel (L, I).

3.2 Die Burrows-Wheeler-Rücktransformation

Die Rücktransformation wird detailiert in [BW94, Seite 3-5, Algorithmus D] sowie [Heu03, Abschnitt 6.5] beschrieben. Ausgehend von M_S wird die Matrix M'_S als Rotation von M_S um

ein Zeichen nach rechts definiert. Da M_S ausschließlich Rotationen von S enthält, enthält auch M'_S Rotationen. Der Vektor T der Dimension $|S|$ wird nun als Korrelation von M_S zu M'_S wie Folgt berechnet:
Für jede Zeile Z_i von M'_S suche in M_S die Zeile Z_j, die zu Z_i äquivalent ist und setze $T_i = j$
Durch die Konstruktion ist bekannt, dass für alle $0 \leq j \leq |S| - 1$ gilt: L_j ist S der direkte zyklische Nachfolger von L_{T_j}.

Sei nun T^i wie Folgt definiert: ${T^0}_x = x$; ${T^{i+1}}_x = T_{{T^i}_x}$ Dann kann S wie Folgt rekonstrukiert werden: *Für alle $0 \leq i \leq |S|$ gilt:* $S_{N-1-i} = L_{{T^i}_I}$

3.3 Vereinfachtes Verfahren zur Rücktransformation

Das Folgende Verfahren ist eine vereinfachte Version der Rücktransformation, die weniger komplex ist als die in Kapitel 3.2 auf der vorherigen Seite vorgestellte, dafür allerdings mehr Rechenzeit benötigt. Da sie einfacher zu erklären ist, wird sie im Vortrag ausführlicher als die ursprüngliche Rücktransformation dargestellt. Detailiert wird diese Variante der Rücktransformation in [Wik] dargestellt.

Gegeben sei das Tupel (L, I) - das Resultat der Hintransformation. Durch das Kompressionsverfahren ist bekannt, dass der letzte Spaltenvektor von M L ist. Zudem wurde M zuvor lexikographisch sortiert, sodass zudem bekannt ist, dass der erste Spaltenvektor von M - der durch die Rotation dieselben Zeichen mit derselben Häufigkeit wie L enthält - dadurch kann der erste Spaltenvektor von M ausgefüllt werden, indem die Zeichen in L lexikographisch sortiert werden.

Wir definieren nun die Matrix M', indem wir M um ein Zeichen nach links rotieren. Dadurch ist in M' lediglich die letzte und die vorletzte Spalte bekannt. Im weiteren Verlauf werden die Folgenden Schritte auf M' wiederholt, bis M' voll ist:

1. Füllen der freien Spalte mit dem größten Index mit L
2. Lexikographisches Sortieren der gesamten Matrix M'

Wenn M' voll ist, ist M' vollständig äquivalent zu M.

3.4 Komprimierbarkeit der transformierten Textes

Die Gründe, warum der tranformierte Text in vielen Fällen besser komprimierbar ist als der Originaltext, werden in [BW94] detailiert dargestellt. Weitere Beispiele finden sich auch in [Wik].

Betrachten wir Beispielsweise einen englischen Text und die Matrix M' aus der Hintransformation (Siehe auch Kapitel 3.1 auf der vorherigen Seite).

Statistisch lässt sich feststellen, dass in englischen Texten einige Kombinationen aus aufeinanderfolgenden Buchstaben häufiger vorkommen als andere, beispielsweise kommt *th* in *the* deutlich öfter vor als *qy*.

Sofern wir nur diejenigen Zeilen in M' betrachten, die mit dem zweiten Buchstaben eines häufig vorkommenden Buchstabenpaars beginnen, so wird durch die Rotation und lexikographische Sortierung sichergestellt, dass in vielen aufeinanderfolgenden Zeilen der erste Buchstabe des Buchstabenpaars in der letzten Zeile steht. Da nun L durch den letzten Spaltenvektor definiert ist, werden durch solche Buchstabenpaare lange Runs in L erzeugt. Von entsprechenden Kompressionsalgorithmen können diese Runs ausgenutzt werden.

3.5 Blockbasierte Kompression

Gerade bei großen Datensätzen bedeutet die Sortierung einer $n \times n$-Matrix einen enormen Zeitaufwand, der in vielen Anwendungsfällen nicht akzeptabel ist. Um diesem Problem entgegenzuwirken, wird der Originaltext meist in mehrere Blöcke gleicher Größe aufgeteilt - dadurch wird zwar oft nicht die maximale mögliche Kompression erreicht, da Korrelationen zwischen den Blöcken bei der BWT nicht berücksichtigt werden, der Zeitaufwand aber sinkt bei kleinerer Blockgröße enorm.

3.6 Kombination mit anderen Algorithmen

Die im Rahmen dieser Arbeit entwickelte Software *eBWT*[1] führt auf einen Bioinformatik-Datensatz[2] und den *Fictional Text* aus dem Calgary-Kompressionstestkorpus verschiedene Kombinationen von Algorithmen aus und prüft für verschiedene Blockgrößen die Größe der resultierenden Dateien.

eBWT implementiert einen Speichereffizienten Rotierungs- und Sortierungsalgorithmus in C++11, der auf einem modifizierten *Heapsort*-Algorithmus aufbaut. Dieser speichert statt den zu sortierenden Strings nur einen Zeiger und erzielt dadurch eine Speicherkomplexität von $\mathcal{O}(n)$ bezüglich der Blockgröße.

Auf die Repräsentation des Indizes I aus dem Resultat der Hintransformation sowie des Dictionaries des Move-To-Front-Encodings wird verzichtet, da die Wahl der Repräsentation große Einflüsse auf die resultierende Dateigröße haben könnte.

Untersucht wurde insbesondere die Frage, inwiefern die Kombination der BWT mit den in [BW94] vorgeschlagenen Algorithmen *Move-To-Front-Encoding* (MTF) und *Huffman-Kodierung* zur Erreichung einer hohen Speicherplatzeinsparung notwendig ist.

Zum Zeitpunkt der Erstellung dieser Ausarbeitung konnte aufgrund hoher Entwicklungs- und Rechenzeiten nur eine kleine Zahl der möglichen Experimente mit eBWT durchgeführt werden. Die Folgenden Resultate wurden bisher erzielt:

- Mit der BWT transformierte Texte lassen sich mit Dictionary-basierten Algorithmen wie *Deflate* meist schlechter komprimieren als die entsprechenden Originaltexte
- Übereinstimmend mit den Resultaten in [BW94] wurde festgestellt, dass unter jeder möglichen Kombination von BWT, MTF und Huffman-Kodierung unabhängig von der Blöckgröße die Kombination BWT →MTF →Huffman-Codierung die kleinsten Dateien erzeugt
- Die Kombination von MTF und Huffman-Kodierung (ohne BWT) erzeugt beim Bioinformatik-Testdatensatz eine deutlich bessere Kompression als beim englischen Text
- Eine Vergrößerung der Blockgröße über etwa 1e+5 hinaus hat nur einen sehr kleinen Effekt, beim Bioinformatik-Datensatz ist dieser Effekt allerdings größer als beim englischen Text
- Beim Bioinformatik-Datensatz ist der Effekt einer Vergrößerung der Blockgröße auf die resultierende Dateigröße im Allgemeinen deutlich kleiner als beim englischen Text.

[1] extreme BWT

[2] Chromosom III von *C. Elegans* als FASTA-Datei, Quelle: `ftp://hgdownload.cse.ucsc.edu/goldenPath/ce6/chromosomes/`

Insgesamt legen diese Resultate nahe, dass signifikante Unterschiede zwischen dem englischen Text und dem Bioinformatik-Datensatz bestehen. Die Resultate aus nicht-bioinformatikbezogenen Publikationen müssen daher im Bezug auf Bioinformatik-Datesätze mit Vorsicht betrachtet werden.

3.7 Komplexität und Laufzeitverhalten

Die Burrows-Wheeler-Transformation hat die asymptotische Komplexität $\mathcal{O}(n^2)$. Die Sortierung während der Transformation nimmt von den Teilschritten des Verfahrens am meisten Rechenzeit in Anspruch - daher hängt das Laufzeitverhalten insbesondere von der Wahl der Sortierungsalgorithmus ab.

Burrows und Wheeler stellen in [BW94] einen modifizierten Quicksort-Algorithmus vor, der für den von ihnen verwendeten Calgary-Kompressionstestkorpus das beste Laufzeitverhalten zeigte. Das lässt sich allerdings nicht notwendigerweise auf andere Datensätze - insbesondere nicht auf Bioinformatik-Datensätze - verallgemeinern. Die meisten modernen Sortieralgorithmen unterscheiden sich in der Average-Case- und der Worst-Case-Komplexität. Allerdings sind auf Bäumen basierende Algorithmen in vielen Fällen auf konkreter Hardware langsamer als in der Theorie.

Der Grund dafür liegt vor allem darin, dass der Speicher, in dem der Baum und seine Attribute gespeichert sind, aufgrund von Änderungen, z.B. an der Baumstruktur häufig neu alloziert werden muss. Dadurch wird in vielen Fällen die Repräsentation des Baums im Arbeitsspeicher fragmentiert.

CPUs können aber nicht direkt auf den Arbeitsspeicher zugreifen, sondern müssen kleinere Einheiten - sogenannte *Pages*[3] - zuerst in einen CPU-nahen mehrstufigen, aber insgesamt nur wenige Megabyte großen Cache laden. Sofern auf eine Page zugegriffen werden soll, die sich nicht im Cache befindet, wird ein sogenannter *page fault* ausgelöst und die Page aus dem Arbeitsspeicher in den Cache geladen. Dieser Prozess dauert allerdings im Vergleich zu Rechenoperationen sehr lange und verzögert daher den Programmablauf.

Aufgrund der Fragmentierung der Repräsentation von Bäumen im Speicher ist die Wahrscheinlichkeit groß, dass bei einem gegebenen Algorithmus zwei Knoten oder Kanten des Baums, auf die nacheinander zugegriffen wird, in zwei verschiedenen Pages liegen. Unter Umständen befindet sich diese Page aber nicht mehr im Cache und muss daher aus dem Arbeitsspeicher geladen werden.

Aufgrund dieses Effektes sind Baumbasierte Sortieralgorithmen auch für die Burrows-Wheeler-Transformation nicht immer die beste Wahl, selbst wenn ihre theoretische Komplexität für eine definierte Menge an Datensätzen ein besseres Laufzeitverhalten nahelegt. In der Praxis müssen oft verschiedene Algorithmen getestet werden, um den zu finden, der für die zu transformierenden Daten das beste Laufzeitverhalten zeigt.

Einen detailierteren Einblick in diese Thematik bietet [Dre07].

3.8 Resultate von Cox und Bauer

In [CBJR12] wird eine Anwendung einer modifizierten Version der BWT („*BWT-SAP*“) auf eine Datenbank von DNA-Sequenzen beschrieben.

[3] Meist 4096 Bytes groß auf Linux x86_64-Systemen

Insbesondere wird die BWT-SAP benutzt, um die durch die durch die starke Korrelation sich überlappender Sequenzen entstehende Redundanz zu verringern (Siehe auch Kapitel 2 auf Seite 2).

Cox und Bauer benutzen dafür eine auf *Suffix Arrays* basierende Methode, um große Datenmengen effizient komprimieren zu können. Dadurch treffen die in Kapitel 3.7 auf der vorherigen Seite beschriebenen Nachteile von Baumbasierten Algorithmen nciht zu.

4 Zusammenfassung

Im Kontext einer allgemeinen Einführung über die Prinzipien verlustfreier Kompression wurde auf die Burrows-Wheeler-Transformation - ein Algorithmus, um Texte besser komprimierbar zu machen - eingegangen.
Verschiedene Methoden für Hin- und die Rücktransformation wurden vorgestellt.

Anhand der für diesen Zweck entwickelten Software *eBWT* wurde analysiert, welche Algorithmen sich besonders für die Anwendung nach der BWT eignen und wie stark Unterschiede zwischen englischen Texten und Bioinformatikdatensätzen ausgeprägt ist.

Zudem wurde die Wahl des Sortierungsalgorithmus diskutiert, wobei besonders auf die Nachteile von Baumbasierten Algorithmen auf konkreter Hardware eingegangen wurde.

Literatur- und Quellenverzeichnis

[BW94] Burrows, M. ; Wheeler, D.J.: A block-sorting lossless data compression algorithm. (1994)

[CBJR12] Cox, A.J. ; Bauer, M.J. ; Jakobi, T. ; Rosone, G.: Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. In: *Bioinformatics* 28 (2012), Nr. 11, S. 1415–1419

[Dre07] Drepper, Ulrich: What Every Programmer Should Know About Memory. (2007). `http://www.akkadia.org/drepper/cpumemory.pdf`. – Letzter Zugriff: 4.11.2012 - 23:00

[Heu03] Heun, Volker: *Grundlegende Algorithmen*. 2. Vieweg-Verlag, 2003

[Wik] *Burrows–Wheeler transform*. `http://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform`. – Letzter Abruf: 4.11.2012 - 23:00

Open Data

Alle für die Erstellung dieser Ausarbeitung sowie für die Präsentation verwendeten Quelldateien und Rohdaten können dauerhaft unter `https://github.com/ulikoehler/Proseminar` abgerufen werden