Lempel - Ziv - basierter Self-Index

Uwe Baier

09. 10. 2014

Zusammenfassung

Dieser Report beschreibt ein Projekt an der Universität Ulm, in dessen Rahmen ein auf LZ77 - basierender Self - Index implementiert wurde. Neben der Funktionsweise des Index werden zusätzlich experimentelle Resultate aufgeführt, die mithilfe der erstellten Implementierung erzielt wurden.

1 Einleitung

Im Rahmen eines Projekts an der Universität Ulm bestand die Aufgabe darin, einen Self - Index zu implementieren, der auf der LZ77 - Faktorisierung [1] eines Textes S basiert.

Zu den Aufgaben des Index gehören

- Zu gegebenem i und m den Textabschnitt $S[i \dots i+m-1]$ aus der LZ77 - Faktorisierung des Textes zu extrahieren.
- Alle Positionen im Text auszugeben, an denen ein Suchwort (*Pattern*) vorkommt. Genauer gesagt besteht die Aufgabe also darin, für ein gegebenes Pattern P der Länge m die Menge

  $$\{i : S[i \dots i+m-1] = P[1 \dots m]\}$$

  zu berechnen.

Der Index wurde auf der Grundlage eines Artikels von Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich und Simon J. Puglisi [2] erstellt. Auch der Artikel von Sebastian Kreft und Gonzalo Navarro [3] wurde hinzugezogen (der eigentlich grundlegendere Artikel), sofern die Beschreibungen im ersten Artikel zu dünn waren. Dabei wurde der beschriebene Index leicht verändert, um gewisse Operationen einfacher durchführen zu können oder anschaulicher darzustellen.

Abschnitt 2 dieses Reports befasst sich mit der Funktionsweise des Index, sowie diversen Schwierigkeiten, die sich während der Implementierung aufgetan haben.

Im dritten Abschnitt wird dann eine mögliche Anwendung des Index kritisch betrachtet: Gegeben seien „ähnliche“ Texte $S_1, \dots, S_t$, in denen mehrfach nach

einem Suchwort gesucht werden soll. Eine Möglichkeit wäre sicherlich, jeden Text S_i zu indizieren, und dann in jedem Text zu suchen. Da die Texte sich aber ähnlich sind (ohne Ähnlichkeit jetzt genauer beschreiben zu wollen), bietet sich auch folgendes Vorgehen an:
Man konkateniere die Texte zu einem gemeinsamen Text $S = S_1 \dots S_t$, und indiziere diesen mithilfe des hier beschriebenen Index. Durch die Ähnlichkeit der Texte kann man erwarten, dass S stark repetitiv ist, womit die LZ77 - Faktorisierung von S sehr gut komprimiert. Da der Index selbst gerade auf der LZ77 - Faktorisierung basiert, kann man vom Index eine gute Kompressionsrate und effizienten Zugriff erwarten. Diese These wird anhand einiger experimenteller Ergebnisse geprüft.

2 Funktionsweise

Zunächst soll die Funktionsweise des Index erläutert werden. Um diese besser verständlich zu machen, wird der Index beispielhaft für den Text $S =$ *textitexttext* konstruiert. Es wird Großteils auf Angaben zu Speicherverbrauch und Komplexität verzichtet, da diese meist denen aus den originalen Artikeln [2, 3] entsprechen.

2.1 Lempel - Ziv - Faktorisierung

Betrachten wir zunächst die Lempel - Ziv - Faktorisierung unseres Beispieltextes *textitexttext*. Sie wird in der Storer - Szymanski - Variante [4] dargestellt, und kann über ein SA - basiertes Verfahren aus dem Text gewonnen werden [5].

Abbildung 1 zeigt uns eine mögliche LZ - Faktorisierung von S. Die Frage ist nun, wie die Information gespeichert werden soll. Dazu kurz einige Begrifflichkeiten zu einem Abschnitt der LZ - Faktorisierung (einem sogenannten *Phrase*):

- *length(p)* bezeichnet die Länge des Phrases p.
- *start(p)* bezeichnet die Startposition des Phrases p im Text.
- *end(p)* bezeichnet die exklusive Endposition des Phrases p im Text.
- *source(p)* bezeichnet die Position, an der die Wiederholung des Phrases p zu finden ist.

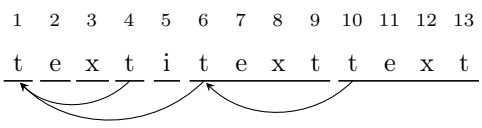


Abbildung 1: LZ - Faktorisierung von S

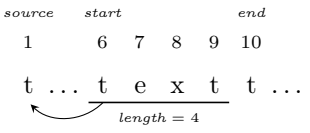


Abbildung 2: Begriffe eines *Phrase*

Üblicherweise wird die Faktorisierung aus Paaren von *source* und *length* gespeichert, wobei $length = 0$ impliziert, dass *source* einen Buchstaben enthält.

Alternativ kann man die Faktorisierung auch aus Paaren von *source* und *end* darstellen, es gilt nämlich folgender Zusammenhang:

- $length(p) = \begin{cases} end(p) - end(p-1) & , p > 1 \\ 1 & , \text{sonst} \end{cases}$
- $start(p) = \begin{cases} end(p-1) & , p > 1 \\ 1 & , \text{sonst} \end{cases}$

Um auch Buchstaben in der Faktorisierung speichern zu können, vereinbart man für einen Phrase $p : length(p) = 1 \Leftrightarrow offset(p)$ enthält einen Buchstaben

```
('t', 0 )
('e', 0 )
('x', 0 )
( 1 , 1 )
('i', 0 )
( 1 , 4 )
( 6 , 4 )
```

(a) Konventionell, bestehend aus (*source*, *length*) - Paaren

```
('t', 2 )
('e', 3 )
('x', 4 )
('t', 5 )
('i', 6 )
( 1 , 10 )
( 6 , 14 )
```

(b) Alternativ, bestehend aus (*source*, *end*) - Paaren

Abbildung 3: LZ - Faktorisierungsrepräsentationen von S

In der alternativen Faktorisierung können Buchstaben nun mehrfach enthalten sein, dies spielt aber Speichertechnisch keine Rolle. Zusätzlich bietet die Alternative einen entscheidenden Vorteil: Länge und Start eines Phrase können in konstanter Laufzeit berechnet werden. Damit bietet die Alternative einen echten Informationsgewinn, der für den umgesetzten Index von entscheidender Bedeutung ist.

2.2 Textabschnitte extrahieren

Nachdem die LZ - Faktorisierung des Textes bekannt ist, stellt sich nun die Frage, wie Textabschnitte effizient extrahiert werden können, also zu gegebenem i und m den Textabschnitt $S[i \dots i+m-1]$ aus der Faktorisierung zu berechnen.

Hierzu wird ein Array benötigt, welches für jeden Phrase angibt, in welchem Phrase sich seine Textquelle (*source*) befindet. Dies kann gewissermaßen als eine Grammatik der LZ - Faktorisierung betrachtet werden.

$$p\_src\_p(p) \coloneqq \begin{cases} 0 & , length(p) = 1 \\ \max\{\tilde{p} : start(\tilde{p}) \leq source(p)\} & , \text{sonst} \end{cases}$$

Um einen gewünschten Textabschnitt zu extrahieren, kann als erstes per binärer Suche der Phrase gesucht werden, in dem der Textabschnitt anfängt.

Nun kann der Grammatik der Faktorisierung gefolgt werden, um den Text herzustellen. Dabei ist zu beachten, dass die ersten Zeichen eines Phrase teilweise übersprungen werden müssen (*skip*), und mit dem nächsten Phrase fortgefahren werden muss, falls der aktuelle Phrase zu wenige Zeichen enthält.

Algorithmus 1 Extraktion eines Textabschnittes. p bezeichnet den Phrase, an dem extrahiert werden soll, *skip*, wie viele Zeichen am Anfang des Phrase übersprungen werden sollen und *count* die Anzahl der zu extrahierenden Zeichen

```
 1: procedure EXTRACT(p, skip, count)
 2:     while length(p) ≤ skip do                          ▷ select correct phrase
 3:         skip ← skip − length(p)
 4:         p ← p + 1
 5:     end while
 6:     repeat                                             ▷ extract count characters
 7:         if length(p) = 1 then                          ▷ phrase contains a character
 8:             output character source(p)
 9:             count ← count − 1
10:         else
11:             pp ← p_src_p(p)
12:             pskip ← skip + source(p) − start(pp)
13:             pcount ← min{length(p) − skip, count}
14:             EXTRACT(pp, pskip, pcount)
15:             count ← count − pcount
16:             skip ← 0
17:         end if
18:         p ← p + 1                                      ▷ process next phrase
19:     until count > 0
20: end procedure
```

Gagie, Gawrychowski, Kärkkäinen, Nekrich und Puglisi erreichen durch Zwischenspeicherung von Nichtterminalsymbolen und Pfaden innerhalb des Grammatikbaumes eine verbesserte Laufzeit bei der Extraktion [2]. Auf diese Modifikation wurde hier verzichtet, um den Projektrahmen nicht zu sprengen.

2.3 Exakte Suche

Nun zu einer elementaren Operation des Index: der exakten Suche. Diese beschreibt die Aufgabe, alle Positionen im Text auszugeben, an denen ein Suchwort (*Pattern*) vorkommt. Formal ausgedrückt muss also die Menge

$$\{i : S[i \dots i + m - 1] = P[1 \dots m]\}$$

berechnet werden, wobei P ein Pattern der Länge m und S der ursprüngliche Text ist. Zunächst unterscheiden wir verschiedene Arten von Vorkommnissen eines Pattern im Text.

- Ein Pattern, welches im Text über mehrere Phrases der Faktorisierung hinweg auftritt, nennen wir *Primary Occurence*.
- Ein Pattern, welches im Text in einem Phrase der Faktorisierung komplett enthalten ist, nennen wir *Secondary Occurence*.

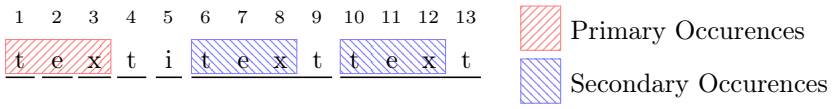


Abbildung 4: Vorkommnisse des Pattern *tex* in S. Phrases der LZ - Faktorisierung sind durch Linien angedeutet.

Wie wir später sehen werden, lassen sich Secondary Occurences leicht mithilfe von Primary Occurences berechnen. Deshalb soll nun als erstes die Aufgabe darin bestehen, Primary Occurences zu finden.

2.3.1 Primary Occurences

Eine Primary Occurence zeichnet sich dadurch aus, dass sie über mehrere Phrases hinweg auftritt. Dies bedeutet, dass eine Primary Occurence durch das Ende des ersten Phrase ihres Auftretens in einen Präfixteil und einen Suffixteil geschnitten wird (siehe Abbildung 5).

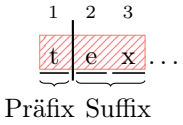


Abbildung 5: Primary Occurence des Pattern *tex* in S. Die Occurence wird durch das Ende des ersten Phrase in zwei Teile zerlegt.

Der Präfixteil des Pattern bildet somit ein Suffix des ersten Phrases der Occurence, während der Suffixteil des Pattern ein Präfix des auf den Phrase folgenden Suffixes ist.

Um nun alle Primary Occurences eines Pattern P zu bestimmen, wird das Pattern an allen möglichen Positionen in einen Präfixteil und einen Suffixteil zerlegt, dies sind $m - 1$ mögliche Zerlegungen[1].
Für jede Zerlegung wird mithilfe zweier Tries bestimmt, welche Phrases mit dem Präfixteil des Pattern enden, und welche auf Phrases folgende Suffixe mit dem Suffixteil des Pattern beginnen. Ein Trie besteht dabei aus den umgedrehten Phrases, der andere aus den Suffixen des Textes, die auf die Phrases folgen.

[1] Für den Fall $m = 1$ können alle Phrases bestimmt werden, die mit dem gegebenen Zeichen $P[1]$ beginnen, und ihre entsprechenden Startpositionen bestimmt werden. Dies ist mithilfe der im Folgenden beschriebenen Datenstrukturen effizient möglich.

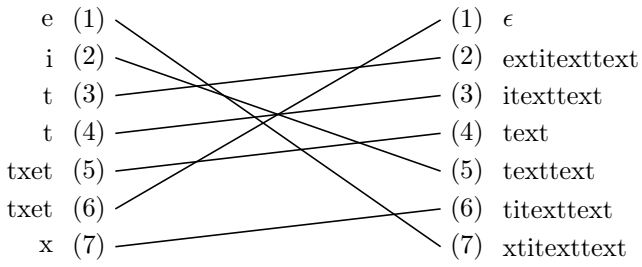


Abbildung 6: Umgedrehte Phrases und Phrasesuffixe des Textes S, lexikographisch sortiert. Assoziationen zwischen Phrases und Suffixen sind durch Linien dargestellt.

Um Speicherplatz zu sparen, werden die Tries als *Patricia Tries* [6] gespeichert. In Ihnen sind die Kanten zwischen den Knoten nicht mit dem gesamten Text, sondern nur mit dem ersten Zeichen des Textes und der Anzahl an Zeichen auf der entsprechenden Kante beschriftet. In den Blättern der Tries befinden sich Identifikationsnummern der Phrases, die den umgedrehten Phrases beziehungsweise deren Suffixen entsprechen. Zusätzlich enthält jeder Knoten eines Tries den Index seines lexikographisch kleinsten und größten Textes.

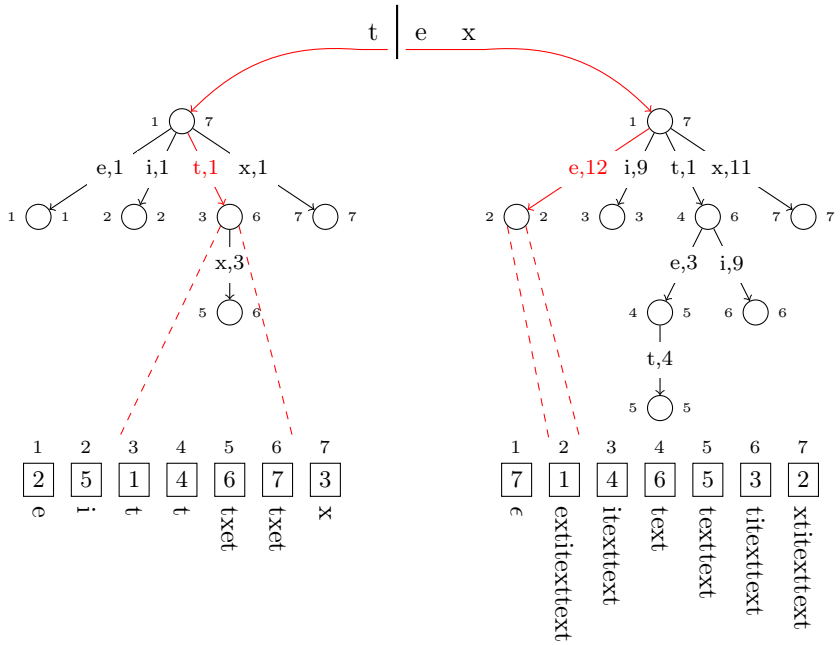


Abbildung 7: Suche nach den lexikographischen Intervallen des Präfixteils t und des Suffixteils ex des Pattern tex in den Patricia Tries der umgedrehten Phrases bzw. den Suffixen der Phrases.

Durch eine entsprechende Suche mit einer Zerlegung des Pattern erhält man also ein lexikographisches Intervall der Phrases, die mit dem Präfixteil der Zerlegung enden, und ein weiteres lexikographisches Intervall an Phrasesuffixen, welche mit dem Suffixteil des Pattern beginnen (ein Beispiel ist in Abbildung 7 zu finden).

Um Phrases zu bestimmen, an denen Primary Occurences auftreten, müssen nun alle Phrases bestimmt werden, welche in beiden Intervallen enthalten sind. Hierzu wird ein Raster angelegt, welches auf den Achsen mit den lexikographischen Ordnungen der umgedrehten Phrases beziehungsweise den Phrasesuffixen beschriftet ist. Für jeden Phrase wird auf dem Raster ein Punkt (i, j) angelegt, wobei i die lexikographische Ordnung des umgedrehten Phrases und j die lexikographische Ordnung des auf den Phrase folgenden Suffixes ist.

Die beiden gefundenen Intervalle aus den Patricia Tries spannen nun einen Bereich auf dem Raster auf. Alle Punkte innerhalb dieses Bereichs entsprechen Phrases, in denen mögliche Primary Occurences auftreten.

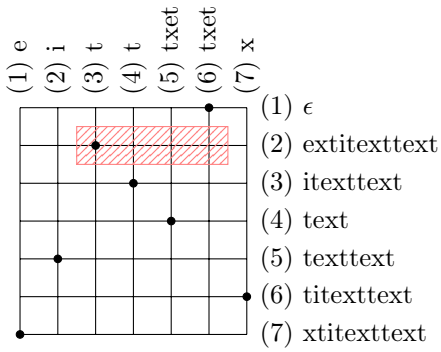


Abbildung 8: Zuordnung der lexikographisch sortierten umgedrehten Phrases mit ihren Suffixen aus Abbildung 6 in einem Raster dargestellt.
Der schraffierte Bereich entspricht den berechneten Intervallen aus Abbildung 7. Alle Punkte innerhalb des Bereichs entsprechen somit einer möglichen Primary Occurence des Pattern *tex*.

Um die tatsächlichen Startpositionen der möglichen Primary Occurences zu bestimmen, wird zuerst mithilfe der x - Koordinaten der gefundenen Punkte in den Blättern des ersten Patricia Tries Phrase Identifier bestimmt[2]. Die Startposition einer möglichen Primary Occurence ergibt sich dann aus der Endposition eines Phrase abzüglich der Anzahl an Zeichen im Präfixteil der Zerlegung.

In unserem aktuellen Beispiel ergibt die Bereichssuche im Raster den Punkt $(3, 2)$. Ein Blick in den Patricia Trie für die umgedrehten Phrases aus Abbildung 7 verrät uns, dass hinter der x - Koordinate 3 der Phrase Nummer 1 steht. Da in der betrachteten Zerlegung der Präfixteil 1 Buchstaben enthält, ergibt sich

[2] Analog könnten auch mit den y - Koordinaten im zweiten Patricia Trie Phrase Identifier bestimmt werden.

die Position der Occurence zu $end(1) - 1 = 2 - 1 = 1$.

Was bisher allerdings nicht beachtet wurde, ist die Tatsache, dass bei der Suche innerhalb der Patricia Tries nicht unbedingt alle Zeichen der Zerlegung mit den tatsächlichen Zeichen im Text verglichen wurden. Um Gewissheit über die Gleichheit des Pattern mit den gefundenen Primary Occurences einer Zerlegung zu haben, muss zuletzt noch der Text an der Anfangspositionen einer Primary Occurence extrahiert und mit dem Pattern verglichen werden.

Eine sehr grobe Abschätzung zeigt, dass das finden aller Primary Occurences eines Pattern mindestens quadratische Laufzeit in der Patternlänge benötigt: Für jede Zerlegung ($m - 1$ viele) müssen im schlimmsten Fall m Zeichen aus der LZ - Faktorisierung extrahiert werden. Man beachte, dass diese Laufzeitabschätzung die Rastersuchzeit und zusätzlichen Overhead durch den Extraktionsalgorithmus nicht beachtet. Genauere Abschätzungen sind in der Arbeit von Kreft und Navarro zu finden.

Nun noch zu einigen Implementierungsdetails:
Das Raster wurde mithilfe eines *Wavelet Tree* [7, 8] repräsentiert, da dieser eine effiziente zweidimensionale Intervallsuche unterstützt. Die Konstruktion der Rasterdaten benötigt ein temporäres Array und erfolgt in zwei Stufen:

1. $temp(p) \coloneqq \mathrm{suff}_{\mathrm{lex}}(p)$ für alle Phrases p
2. $griddata(\mathrm{rp}_{\mathrm{lex}}(p)) \coloneqq temp(p)$ für alle Phrases p

Hierbei bezeichnen $\mathrm{suff}_{\mathrm{lex}}$ bzw. $\mathrm{rp}_{\mathrm{lex}}$ die lexikographischen Ordnungen der Phrasesuffixe bzw. der umgedrehten Phrases. Beide Ordnungen können durch geeignete Iteration der Blätter der jeweiligen Patricia Tries gewonnen werden. Als temporäres Array bietet sich das Suffixarray an, da es nach Konstruktion der LZ - Faktorisierung nicht mehr benötigt wird.

Bei den Patricia Tries gilt es zu beachten, dass die Tries unter Umständen mehrere gleiche Texte aufnehmen müssen, wenn zum Beispiel zwei Phrases identisch sind. Zusätzlich könnten die Tries durch eine *dfuds* - Repräsentation [9] (wie von Kreft und Navarro vorgeschlagen) dargestellt werden, um Speicherplatz zu sparen. Im Rahmen dieses Projekts wurde aber auf eine entsprechende Implementierung verzichtet.

2.3.2 Secondary Occurences

Nachdem bereits alle Primary Occurences eines Pattern bestimmt werden können, soll die Aufmerksamkeit nun den Secondary Occurences gewidmet werden. Dies sind Auftreten eines Pattern P im Text, welche vollständig innerhalb eines Phrase der Faktorisierung enthalten sind.

Zuerst beobachten wir, dass eine Secondary Occurence nur dann auftreten kann, wenn es eine weitere Occurence des Pattern gibt, welche im Text vor der Secondary Occurence auftritt. Begründen lässt sich diese Behauptung durch die Tatsache, dass der Phrase, in dem die Secondary Occurence auftritt, eine Kopie eines vorherigen Textabschnitts ist. Da die Secondary Occurence vollständig im

Phrase enthalten ist, muss sie demzufolge auch in besagtem vorherigem Textabschnitt enthalten sein.

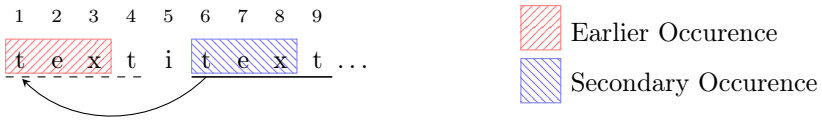


Abbildung 9: Secondary Occurence des Pattern *tex* in S, sowie die Occurence aus der sich die Secondary Occurence ergibt.

Allgemein treten Occurences entweder als Primary Occurences oder Secondary Occurences auf. Mithilfe der eben entdeckten Beobachtung und der Arten eines Auftretens einer Occurence berechnet Algorithmus 2 somit alle Occurences eines Pattern P.

Algorithmus 2 Berechnung aller Occurences eines Pattern P der Länge m.

```
1: let Occ be a List of all Primary Occurences of Pattern P
2: it ← firstelement(Occ)
3: while it ≠ NIL do
4:     i ← value(it)                          ▷ i holds position of occurence
5:     let P be all Phrases p with            ▷ determine phrases whose phrase
         source(p) ≤ i and                    ▷ source contains the occurence
         source(p) + length(p) ≥ i + m
6:     for all p ∈ P do
7:         add start(p) + i − source(p) to the end of list Occ
8:     end for
9:     it ← nextelement(it)
10: end while
11: output Occ
```

Es stellt sich die Frage, wie Zeile 5 des Algorithmus effizient berechnet werden kann. Kreft und Navarro verwenden einen Bitvektor und eine *Range - Maximum - Query* - Datenstruktur [10] um besagte Zeile 5 zu implementieren. Dadurch können alle Secondary Occurences mit Laufzeit $\mathcal{O}(|Occ| \log n')$ berechnet werden, wobei n' die Anzahl der Phrases der LZ - Faktorisierung ist.

In dieser Implementierung wurde ein alternativer Ansatz verwendet, der sich ähnlich zur Zuordnung von umgedrehten Phrases und deren Suffixen aus Abschnitt 2.3.1 verhält. Benötigt werden zwei Arrays *p_ ord_ src* und *p_ ord_ srcend*. Im Array *p_ ord_ src* befinden sich Phrase Identifier, sortiert nach den Quellenpositionen der Phrases.

$$i \leq j \Rightarrow source(p\_ord\_src(i)) \leq source(p\_ord\_src(j))$$

Im Array *p_ ord_ srcend* hingegen befinden sich Phrase Identifier, die nach den

Quellenpositionen der Phrases plus ihrer Länge sortiert sind.

$$i \leq j \Rightarrow \quad source(p\_ord\_srcend(i)) + length(p\_ord\_srcend(i)) \leq source(p\_ord\_srcend(j)) + length(p\_ord\_srcend(j))$$

Man beachte, dass beide Arrays nur Phrase Identifier enthalten dürfen, bei denen der Phrase keinen Buchstaben des Textes enthält (sonst trägt $source(p)$ die Repräsentation des Buchstaben, siehe Abschnitt 2.1). Fortan bezeichne N' die Anzahl solcher Phrases.

Die Arrays werden durch ein Raster komplettiert, welches die Einträge der beiden Arrays zuordnet: Für jeden Phrase p wird ein Punkt (i, j) auf das Raster aufgetragen, wobei i der Index von p im ersten Array und j der Index von p im zweiten Array ist ($p\_ord\_src(i) = p$, $p\_ord\_srcend(j) = p$). Das Raster wird analog zum Raster aus Abschnitt 2.3.1 durch einen Wavelet Tree repräsentiert. Auch die Konstruktion der Rasterdaten erfolgt analog.

Um nun Secondary Occurences einer Occurence an Position i mit Länge m zu bestimmen, wird wie folgt verfahren:

1. Bestimme per binärer Suche den Index s, so dass

   $$s = \max\{x : source(p\_ord\_src(x)) \leq i\}$$

   Da p_ord_src nach Quellenpositionen der Phrases sortiert ist, bedeutet dies, dass alle Phrases des Arrayabschnitts $p\_ord\_src([1 \ldots s])$ eine Quellenposition $\leq i$ besitzen.

2. Bestimme per binärer Suche den Index se, so dass

   $$se = \min\{x : source(p\_ord\_srcend(x)) + length(p\_ord\_srcend(i)) \geq i + m\}$$

   Analog zum vorherigen Index s gibt se nun die Grenzposition im Array p_ord_srcend an, so dass alle Phrases des Arrayabschnitts $p\_ord\_srcend([se \ldots N'])$ eine Quellenendposition $\geq i + m$ besitzen.

3. Um Phrases zu finden, die beide notwendigen Bedingungen erfüllen, werden im Raster alle Punkte im Intervall $[1 \ldots s] \times [se \ldots N']$ bestimmt. Die Phrase Identifier der Punkte können per einsetzen der x - Koordinate der Punkte in p_ord_src gewonnen werden.

Ein vollständiges Beispiel ist in Abbildung 10 zu finden.

Abbildung 10: Suche nach Phrases, in denen das Pattern *tex* vollständig enthalten ist. Das Pattern wurde bereits an Textposition 1 als Primary Occurence ausgemacht.

Phrase	Source	Length	Phrasetext
1	't'	1	t
2	'e'	1	e
3	'x'	1	x
4	't'	1	t
5	'i'	1	i
6	1	4	text
7	6	4	text

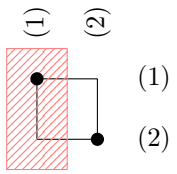
(a) Übersicht wichtiger Kenngrößen der LZ - Faktorisierung des Textes S

i	1	2
$p\_ord\_src(i)$	6	7

(b) Das zu S zugehörige Array p_ord_src. Die binäre Suche nach Phrases mit $source \leq 1$ ergibt die rechte Grenze $s = 1$.

i	1	2
$p\_ord\_srcend(i)$	6	7

(c) Das zu S zugehörige Array p_ord_srcend. Die binäre Suche nach Phrases mit $source + length \geq 4$ ergibt die linke Grenze $se = 1$.



(d) Raster, das p_ord_src und p_ord_srcend verbindet. Die Suche nach Phrases im Intervall $[1 \dots 1] \times [1 \dots 2]$ ergibt den Punkt $(1, 1)$, welcher den Phrase 6 repräsentiert. Dieser enthält auch tatsächlich das Pattern *tex* vollständig, und ist somit eine Secondary Occurence.

Neben der hier angegebenen Implementierungsvariante mit Laufzeit $\mathcal{O}(|Occ| \log N')$ und derer von Kreft und Navarro existieren noch eine ganze Reihe an weiteren Lösungsmöglichkeiten, die mit asymptotisch weniger Speicherverbrauch oder besseren Laufzeiten aufwiegen, siehe zum Beispiel [2] für einige Verbesserungsvorschläge.

Diese Variante stellt jedoch einen „gesunden Mittelweg“ zwischen Laufzeit und Speicherverbrauch dar, ist einfach zu implementieren, und auch recht einfach zu verstehen, was sie für dieses Projekt attraktiv macht.

3 Experimentelle Resultate

In diesem Abschnitt soll der Index anhand einiger Testdaten experimentell getestet werden. Hierzu werden mehrere „ähnliche“ Texte zu einem gemeinsamen Text konkateniert, und daraus ein Index konstruiert. Aufgrund der Ähnlichkeit

der Texte besteht so die Hoffnung, dass die LZ - Kodierung und damit auch der Index selbst wenig Speicherplatz verbrauchen, und die exakte Suche mit schnellen Laufzeiten durchgeführt werden kann.

Aufgrund des veränderten Einsatzzwecks besteht nun eine zusätzliche Aufgabe darin, die Suchtreffer der exakten Suche den ursprünglichen Texten zuzuordnen. Im Rahmen dieses Projekts wurde dies mithilfe eines zusätzlichen Arrays mit den Startpositionen der Texte und einer binären Suche nach dem Text einer Occurence durchgeführt.

Als Testdaten wurden neben verschiedenen Releases von menschlichen Chromosomen [11, chr01, chr09, chr17, jeweils 5 Releases] auch eine Liste mit wichtigen Personen und deren Aufgaben verwendet [12, worldleaders], da diese als stark repetitiv eingestuft wurde.

Alle Benchmarks wurden unter der in Tabelle 1 aufgeführten Rechnerkonfiguration durchgeführt.

CPU	`Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz`
Total Memory	`131970304 KB`
Distribution	`Ubuntu 14.04.1 LTS`

Tabelle 1: verwendete Rechnerkonfiguration

3.1 Speicherverbrauch

Zunächst soll der Speicherverbrauch des Index betrachtet werden. Tabelle 2 zeigt verschiedene Kenngrößen des Speicherverbrauchs.

Testcase	chr01	chr09	chr17	worldleaders
Accumulated Filesize	1072.57 MB	566.52 MB	373.05 MB	44.79 MB
LZ - Coding	106.28 MB	55.41 MB	35.91 MB	1.09 MB
Compression	9.91 %	9.78 %	9.63 %	2.43 %
Index Size	1280.47 MB	670.63 MB	447.11 MB	11.66 MB
$\frac{\text{Index size}}{\text{LZ - Coding}}$	12.05	12.1	12.45	10.7
Index Construction Memory Peak	8916.02 MB	4724.5 MB	3125.14 MB	481.72 MB

Tabelle 2: Kenngrößen des Speicherverbrauchs

Die Daten zeigen, dass der Index sich in der hier angefertigten Implementierung nur für stark repetitive oder sehr ähnliche Texte gut eignet: Der benötigte

Speicherplatz des Index entspricht etwa 12 mal dem der Lempel - Ziv - Kodierung. Dies bedeutet, dass der Index erst ab einer LZ - Kompression von unter 8 Prozent die Daten in komprimierter Weise darstellen kann. Der benötigte Speicher zur Konstruktion des Index wiegt mit etwa 10 Mal der summierten Textgrößen des Testfalls auf.

Betrachtet man den Speicherverbrauch der einzelnen Komponenten des Index in Abbildung 11, stellt man fest, dass vor allem die Repräsentation der Patricia Tries (`pt_revphrases`, `pt_phrasesuffixes`) noch optimiert werden sollte: Die Patricia Tries belegen rund $\frac{2}{3}$ des Gesamtspeicherverbrauchs. Eine alternative Darstellung der Tries (siehe z.B. [9]) könnte hier sicherlich noch einige Verbesserungen mit sich bringen.

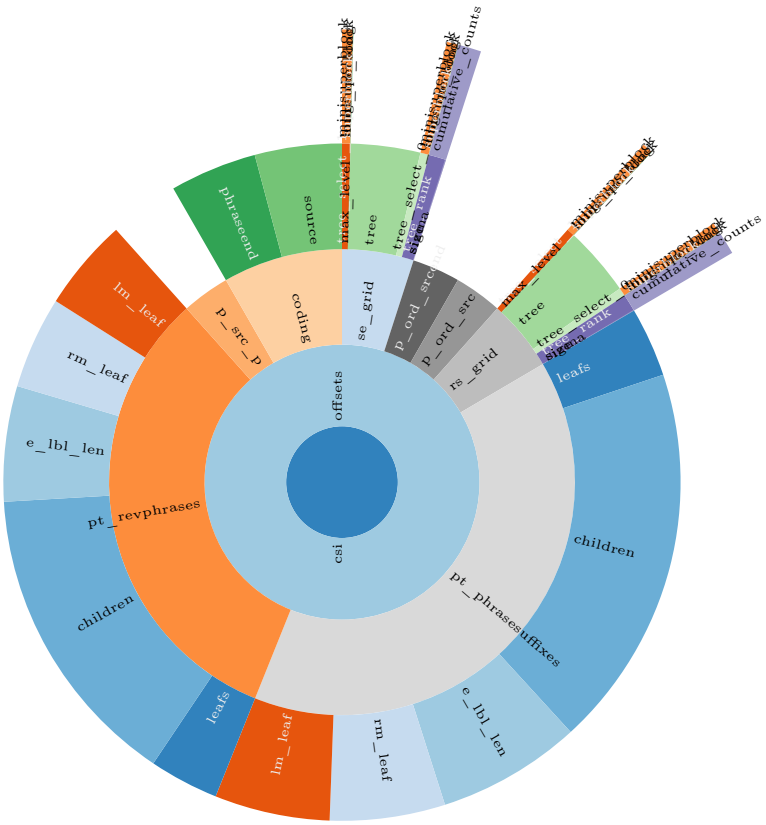


Abbildung 11: Speicherverbrauch einzelner Komponenten des Index, gemessen am Testfall chr01

3.2 Laufzeiten

Nach dem Speicherverbrauch wollen wir uns noch mit Laufzeiten des Index beschäftigen, genauer gesagt mit Konstruktionslaufzeit und Laufzeit für exakte Suche. Hierzu zeigt uns Tabelle 3 Konstruktionszeiten der Testfälle. Im Schnitt werden während der Konstruktion etwa 3.5 $\frac{\text{MB}}{\text{s}}$ verarbeitet.

Testcase	chr01	chr09	chr17	worldleaders
Construction Time	316.61 s	158.23 s	100.53 s	8.05 s
$\frac{\text{Accumulated Filesize}}{\text{Construction Time}}$	3.39 $\frac{\text{MB}}{\text{s}}$	3.58 $\frac{\text{MB}}{\text{s}}$	3.71 $\frac{\text{MB}}{\text{s}}$	5.57 $\frac{\text{MB}}{\text{s}}$

Tabelle 3: Konstruktionszeiten des Index

Nun aber zu der wohl interessantesten Operation des Index: Der exakten Suche. Um im Vorfeld geeignete Pattern zur Laufzeitmessung zu bestimmen, wurde ein kleines Programm angelegt um Repeats verschiedener Länge und Anzahl von Wiederholungen aus den Testdaten zu bestimmen.

Die Laufzeiten der verschiedenen Pattern ist in Abbildung 12 zu finden.

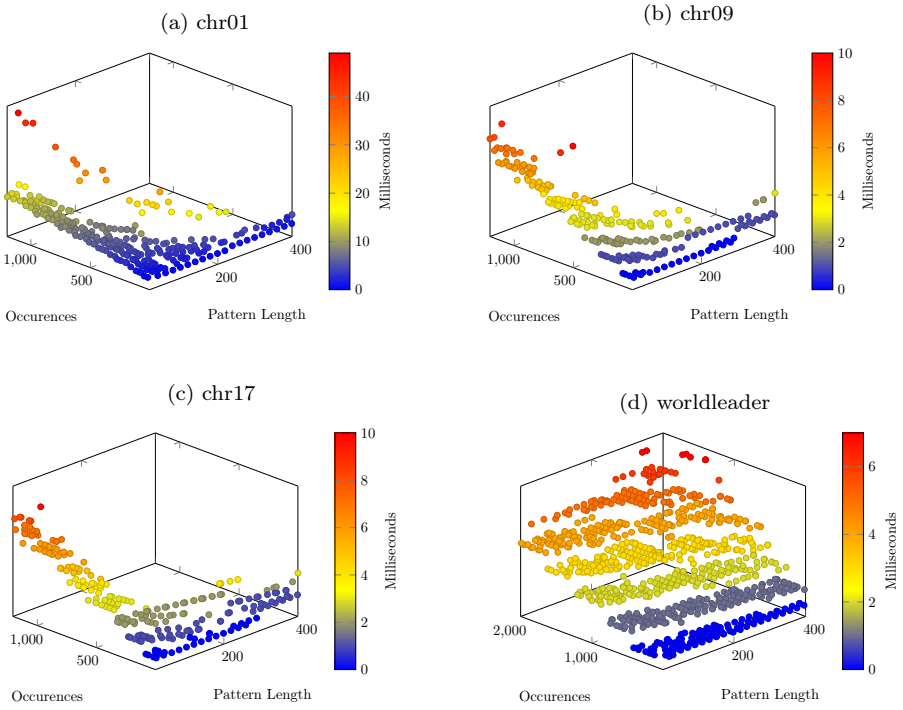


Abbildung 12: Laufzeiten der exakten Suche

Die Laufzeitmessungen zeigen bis auf einige Ausreißer gute Resultate. Von

der quadratischen Abhängigkeit zwischen Patternlänge und Laufzeit ist in den Testdaten bis auf wenige Ausnahmen nichts zu bemerken. Die Anzahl der Auftreten eines Pattern trägt nur mit einem linearen Faktor zur Laufzeit bei.

Auch in diesem Test bestätigt sich der bisherige Trend: Der Index schneidet für stark repetitive Texte am besten ab. Im Übrigen kann man aufgrund der Anzahl an Datenpunkten in Abbildung 12 auch gut einsehen, welche Texte stark repetitiv sind und welche nicht: Die Chromosome besitzen zwar viele Repeats, aber nicht in beliebiger Anzahl an Auftreten. Der Testfall worldleaders hingegen zeigt eine deutlich größere Dichte an Repeats, was sich auch in der Größe seiner LZ - Kodierung widerspiegelt.

3.3 Fazit

Der in diesem Projekt umgesetzte Index eignet sich in seiner bisher implementierten Form aufgrund seines großen Speicherverbrauchs nur für stark repetitive Texte. Durch einige Optimierungen könnte der Index jedoch auch für andere Einsatzzwecke benutzt werden, sofern die Repräsentation seiner Komponenten in noch platzsparenderer Form bewerkstelligt werden kann.

Die Laufzeiten des Index liegen jedoch in einem sehr guten Bereich, was ihn auch jetzt schon für viele Einsatzzwecke attraktiv macht.

Literatur

[1] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[2] Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index, 2012. Preprint from 28.September.

[3] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences, 2011. Preprint from 21. December.

[4] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982.

[5] Enno Ohlebusch. Lempel-ziv factorization: LZ77 without window. `http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Lehre/SS14/Datenkompression/ScriptLZ_140604.pdf` (Last visited 26. July), 2014. Part of Script from Data Compression.

[6] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15*, pages 514 – 534, 1968.

[7] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages

841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[8] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[9] David Benoit, Erik Demaine, Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Representing trees of higher degree. *Algorithmica 43*, pages 275 – 292, 2005.

[10] Johannes Fischer. Optimal succinctness for range minimum queries. In *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 158–169. Springer Berlin Heidelberg, 2010.

[11] Chromosome chr01,chr09,chr17 releases. `ftp://ftp.ensembl.org/pub/` (Last visited 8. October 2014).

[12] World leaders. `http://pizzachili.dcc.uchile.cl/repcorpus.html` (Last visited 6. October 2014).