Universität Paderborn
Institut für Informatik
Prof. Dr. Stefan Böttcher

Proseminar Datenkompression – WS 2016/2017

Linear-Time Suffix-Sorting

Clemens Damke

Matrikelnummer 7011488

Inhaltsverzeichnis

1 Problemstellung

Diese Proseminar-Arbeit beschreibt den GSACA-Algorithmus. Hierbei handelt es sich um den ersten rekursionsfreien Linearzeitalgorithmus zur Konstruktion von Suffix-Arrays.

Im Folgenden wird zunächst erörtert, was Suffix-Arrays sind und wozu sie benutzt werden.

1.1 Was ist ein Suffix-Array?

Das Suffix-Array SA einer Zeichenkette S ist definiert als die lexiographisch aufsteigend sortierte Folge aller Suffixe von S.

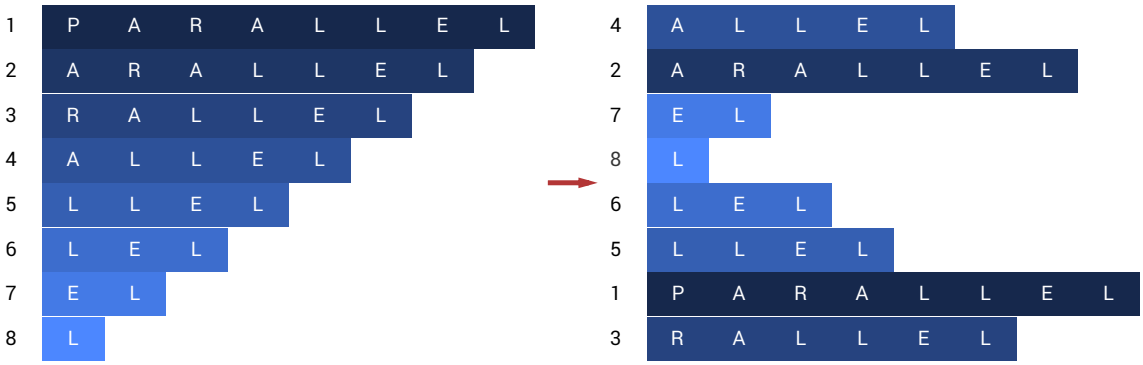

Abbildung 1: Suffixarray für $S =$ ‘parallel’

Um Speicher zu sparen wird SA allerdings nicht als Folge der Suffix-Zeichenketten, sondern als Folge der Startpunkte der Suffixe repräsentiert. Für $S =$ ‘parallel’ wäre also $SA = (4, 2, 7, 8, 6, 5, 1, 3)$. Formal bedeutet dies:

$$
\begin{aligned}
\Sigma &:= \text{streng total geordnetes endliches Alphabet} \\
S &:= \text{Eingabezeichenkette} = (S[1], \ldots, S[n]) \in \Sigma^n, |S| := n \in \mathbb{N} \\
S[i..j+1) := S[i..j] &:= (S[i], \ldots, S[j]) \\
S_i &:= i\text{-ter Suffix von } S = S[i..n] \\
S \sqsubseteq T &:\Leftrightarrow S \text{ ist Präfix von } T \Leftrightarrow S = T[1..|S|] \\
S <_{lex} T &:\Leftrightarrow (\exists\, i : S[i] < T[i] \wedge S[1..i) = T[1..i)) \vee (|S| < |T| \wedge S \sqsubseteq T) \\
SA &:= \text{Permutation von } \{1, \ldots, |S|\}, \text{sodass } \forall\, i < j : S_{SA[i]} <_{lex} S_{SA[j]}
\end{aligned}
$$

1.2 Einsatzgebiete von Suffix-Arrays

Suffix-Arrays finden in vielen Bereichen als Index-Datenstruktur Verwendung. Ein typisches Problem, dessen Lösung durch Suffix-Arrays beschleunigt werden kann, ist z. B. die Substringsuche. Bei dieser soll bestimmt werden, *ob* und wenn ja, *wo* in einem Text T ein Pattern P vorkommt. Ohne einen Index benötigt dieses Problem z. B. mit Knuth-Morris-Pratt $\mathcal{O}(|T| + |P|)$. Mit einem Suffix-Array als Index über T hingegen lassen sich

Matches durch eine binäre Suche in $\mathcal{O}(|P| \log |T|)$ finden. Da i. d. R. $|P| \ll |T|$, ist dies ein deutlicher Speedup, welcher z. B. in Datenbanksystemen für Volltextsuchen und in der Bioinformatik für das Suchen in DNA-Daten nützlich ist.

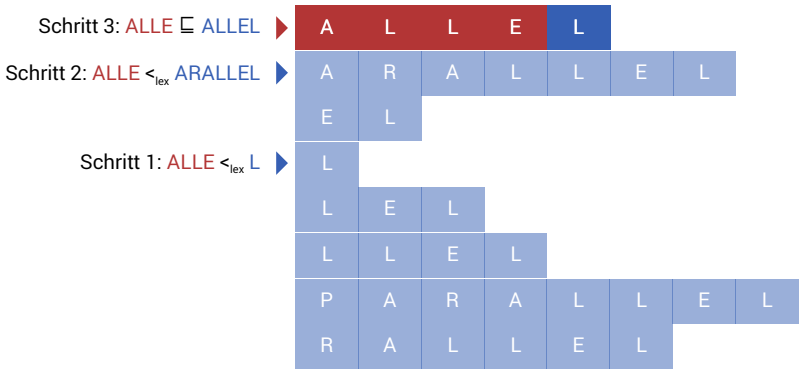

Abbildung 2: Substringsuche mit $P =$ ‘alle’ und $T =$ ‘parallel’

Ein weiteres Einsatzgebiet für Suffix-Arrays ist als Suchstruktur für das Sliding Window in Implementationen des LZ77-Kompressionsalgorithmus.

2 Ansätze zur Suffix-Array-Konstruktion

Nachdem nun der Begriff des Suffix-Arrays definiert wurde, wird im Folgenden betrachtet wie sich dieses prinzipiell algorithmisch berechnen lässt.

2.1 Naiver Ansatz

Da es sich bei der Suffix-Array-Berechnung im Wesentlichen um ein Sortierproblem handelt, liegt die Idee nahe dies mit einem allgemeinen Sortierverfahren zu lösen. Dazu bietet sich z. B. der Quicksort-Algorithmus an. Im average case wären dann $\mathcal{O}(n \log n)$ Vergleiche notwendig. Für den lexiographischen Vergleich zweier Suffixe müssen wiederum bis zu $\mathcal{O}(n)$ Zeichen miteinander verglichen werden. Insgesamt ergibt sich also eine Laufzeit von $\mathcal{O}(n^2 \log n)$. Dies ist weit von der angestrebten $\mathcal{O}(n)$-Laufzeit entfernt. Allgemeine Sortierverfahren sind daher für die Suffix-Array-Konstruktion unbrauchbar.

2.2 Überblick über bisherige Linearzeitansätze

Es sind bereits zahlreiche Linearzeitalgorithmen zur Konstruktion von Suffix-Arrays bekannt. Allerdings sind all diese Verfahren rekursiv. Das bedeutet, dass sie neben der Eingabe und evtl. Hilfsdatenstrukturen zudem mindestens $\mathcal{O}(\log n)$ Speicher für die Stackframes der rekursiven Aufrufe benötigen.

Zwei dieser rekursiven Linearzeitalgorithmen sind der Algorithmus von Skew und der SA-IS-Algorithmus. Skew ist primär wegen seiner Kompaktheit und Eleganz interessant. SA-IS basiert auf dem Konzept der induzierten Sortierung und gehört zu den

schnellsten bekannten SACAs (suffix array construction algorithms). Obwohl es sich bei beiden um Linearzeitalgorithmen handelt, sind die konstanten Faktoren in Implementationen von SA-IS deutlich geringer.

	Skew	SA-IS	**GSACA**
Art	rekursiv	rekursiv	iterativ
Zeit	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Speicher	$\mathcal{O}(\log n) + \max 24n$	$\mathcal{O}(\log n) + \max 2n$	$\mathcal{O}(1) + ?$

Tabelle 1: Vergleich von Skew, SA-IS und GSACA

3 Der GSACA-Algorithmus

Im Rest dieser Arbeit wird es um den GSACA-Algorithmus (greedy suffix array construction algorithm) gehen, welcher der erste bekannte rekursionsfreie Linearzeit-SACA ist. Skew und SA-IS werden dabei als Referenzalgorithmen dienen, mit denen GSACA verglichen wird.

3.1 Grundlegende Konzepte

Um den GSACA-Algorithmus zu verstehen, ist es hilfreich zuerst eine Intuition dafür zu geben, wieso Suffixe überhaupt in $\mathcal{O}(n)$ sortiert werden können. Im Gegensatz zu allgemeinen Sortierverfahren mit $\mathcal{O}(n \log n)$ Vergleichen, lassen sich hier offenbar Vergleiche sparen.

3.1.1 Induziertes Sortieren

Der fundamentale Unterschied zwischen allgemeinen Sortierproblemen und der Suffixsortierung ist, dass aus der Ordnung von bestimmten Suffixen $T(1) <_{lex} \cdots <_{lex} T(n)$ die Ordnung anderer Suffixe $G(1), \ldots, G(n)$ induziert werden kann. Es muss dann also lediglich Zeit in Vergleiche der T-Suffixe investiert werden.

$$
\begin{aligned}
&T := \{T(1), \ldots, T(n)\}, \text{sodass } T(1) <_{lex} \cdots <_{lex} T(n) \\
&G := \{G(1), \ldots, G(n)\}, \text{sodass } \exists\, P \in \Sigma^* \;\forall\, i : G(i) = P\,T(i) \\
&\implies G(1) <_{lex} \cdots <_{lex} G(n)
\end{aligned}
$$

Um diese Implikation nutzen zu können, müssen Suffixe also mit einem geschickt gewählten Präfix P einer Gruppe G zugeordnet werden. GSACA tut genau das.

3.1.2 Gruppenkontext

Für die Gruppierung von Suffixen benutzt GSACA das Konzept des *Gruppenkontextes*, welcher als der gemeinsame Präfix P aller Suffixe in einer Gruppe dient. Dieser Begriff wird im Folgenden näher betrachtet. Es wird dabei angenommen, dass es sich bei der Eingabe S um einen *nullterminierten* String handelt.

$$S \text{ nullterminiert} :\Leftrightarrow S[n] = \$ \wedge S[1..n) \in (\Sigma \setminus \{\$\})^*, \text{mit } \$ \in \Sigma, \forall\, \sigma \in \Sigma \setminus \{\$\} : \$ < \sigma$$

$$\text{zur Erinnerung: } S_i := i\text{-ter Suffix von } S = S[i..n]$$

$$\widehat{i} := \min\{j \in \{i, \ldots, n\} :\ S_j <_{lex} S_i\}$$

$$\text{Gruppenkontext von } S_i := S[i..\widehat{i})$$

$$\text{Gruppe von } S_i := \{S_j :\ \text{Gruppenkontext } S_i = \text{Gruppenkontext } S_j\}$$

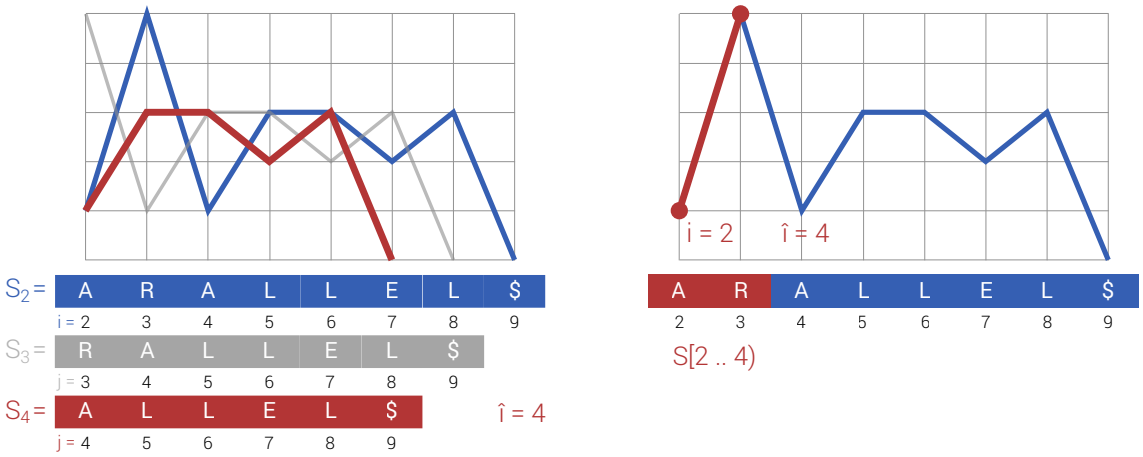

Abbildung 3: Veranschaulichung des Gruppenkontextes von S_2 für $S =$ 'parallel\$'

Der Gruppenkontext eines Suffixes S_i ist also dessen erster Präfix auf den ein lexiographisch kleinerer Suffix von S_i folgt.

3.2 Die zwei Phasen von GSACA

Mit den Konzepten des induzierten Sortierens und des Gruppenkontextes zur Gruppierung von Suffixen, lässt sich GSACA nun als ein in zwei Phasen ablaufender Algorithmus verstehen.

In der *ersten Phase* werden alle Suffixe der Eingabe gemäß ihrer Gruppenkontexte in Gruppen zusammengefasst. Die resultierenden Gruppen werden dabei nach Gruppenkontext aufsteigend sortiert zurückgegeben.

Diese sortierte Folge von Gruppen wird nun als Eingabe der *zweiten Phase* verwendet. Da die Gruppen untereinander bereits nach ihrem Kontext sortiert wurden und die

Kontexte jeweils Präfix der Suffixe einer Gruppe sind, müssen die Suffixe in dieser Phase lediglich innerhalb ihrer jeweiligen Gruppe geordnet werden. Für das Ordnen der Suffixe innerhalb einer Gruppe wird induziertes Sortieren genutzt.

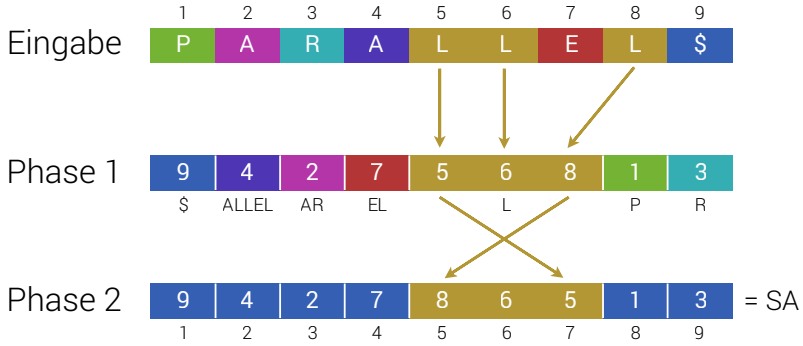

Abbildung 4: Ausgabe beider Phasen für $S =$ ‘parallel$’, Farben kennzeichnen Gruppen

Das Ergebnis der zweiten Phase ist das gesuchte Suffix-Array. Ziel ist es nun beide Phasen ohne Rekursion als $\mathcal{O}(n)$-Algorithmen zu implementieren.

3.2.1 Phase 1

3.2.2 Phase 2

4 Performanceanalyse

test

5 Fazit

test

Literatur