

Suffix Arrays und BWT

Tobias Harrer

19.11.12

Suffix Arrays

Grundlagen

Suffix Arrays aus Suffix Trees

Anwendung von Suffix Arrays

Backward Search

Forward Searching

Burrows-Wheeler Transformation

Suffix Arrays

Definition

- ▶ Sei T ein String der Länge n über Σ und $T_{i,j}$ mit $(0 < i \leq j \leq n)$ der Substring von i bis einschließlich j , dann ist $T_{i,n}$ ein Suffix von T .
- ▶ Das Suffix Array von T ist die Permutation der Startindizes i der lexikografisch geordneten Suffixes $T_{i,n}$ von T .
- ▶ Anmerkung: jeder String T endet mit $\$$, s.d für alle $c \in \Sigma$ gilt $\$ < c$

Suffix Arrays - Beispiel

$T = \text{„abacabra\$“}$

i	$T_{i,n}$	lexikografisch geordnet
1	abacabra\$	\$ 9
2	bacabra\$	a\$ 8
3	acabra\$	abacabra\$ 1
4	cabra\$	abra\$ 5
5	abra\$	acabra\$ 3
6	bra\$	bacabra\$ 2
7	ra\$	bra\$ 6
8	a\$	cabra\$ 4
9	\$	ra\$ 7

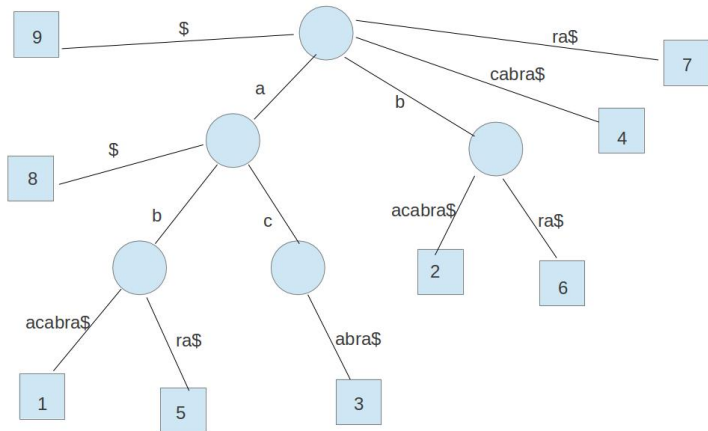
Das Suffix Array von T lautet:

1 2 3 4 5 6 7 8 9 $\rightarrow i$

9 8 1 5 3 2 6 4 7 $\rightarrow S(i)$

Suffix Arrays aus Suffix Trees

- ▶ Sortierung durch z.B. mergeSort in $\mathcal{O}(n \cdot \log(n))$
- ▶ Ein Suffix Array kann aus den Blättern eines Suffix Trees durch Tiefensuche hergeleitet werden: $T = \text{„abacabra\$“}$



Anwendung von Suffix Arrays

- ▶ Finde Pattern p („cab“) in String T („abacabra“)

- ▶ Naiver Ansatz: „schiebe“ p über T :

abacabra

cab→

...

abacabra

. . cab

- ▶ Problem: $\mathcal{O}(n \cdot m)$

Anwendung von Suffix Arrays

- ▶ Vorteil: alle Suffixes lexikografisch sortiert
- ▶ p „abr“ ist Präfix $T_{1,3}$ des Suffix $T_{5,n}$:

$A[i]$	$T_{A[i],n}$
9	\$
8	a\$
1	abacabra\$
5	abra \$
3	acabra\$
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

- ▶ Beste Suchstrategie in sortiertem Array?

Anwendung von Suffix Arrays

- Binäre Suche, Vergleich von „abr“ mit $TA[i], A[i] + 2$:

A[i]	$T_{A[i],n}$
9	\$
8	a\$
1	abacabra\$
5	abra\$
3	aca bra\$ „abr“ < „ aca “ → obere Hälfte
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

Anwendung von Suffix Arrays

- Binäre Suche, Vergleich von „abr“ mit $TA[i], A[i] + 2$:

A[i]	$T_{A[i],n}$
9	\$
8	a\$
1	ab acabra\$ „abr“ > „ ab a“ → untere Hälfte
5	abra\$
3	acabra\$
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

Anwendung von Suffix Arrays

- Binäre Suche, Vergleich von „abr“ mit $TA[i], A[i] + 2$:

$A[i]$	$T_{A[i],n}$
9	\$
8	a\$
1	abacabra\$
5	abr a\$ „abr“ == „ abr “ → Treffer!
3	acabra\$
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

- Suche in $\mathcal{O}(m \cdot \log(n))$, aber: Sind wir hier schon fertig?
- Gibt es weitere Vorkommen von $p = \text{„abr“}$ in T ?

Anwendung von Suffix Arrays

- Diesmal Suche von „ab“
- Binäre Suche, Vergleich von „ab“ mit $TA[i], A[i] + 1$:

A[i]	$TA[i],n$
9	\$
8	a\$
1	abacabra\$
5	abra
3	a cabra\$ „ab“ < „a c “ → obere Hälfte
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

Anwendung von Suffix Arrays

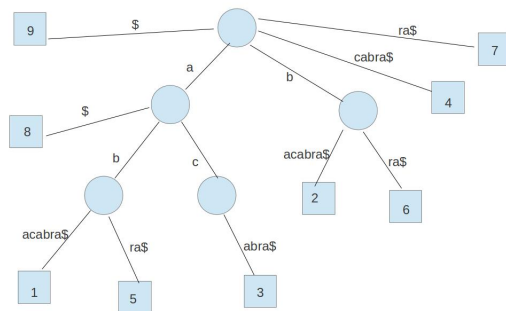
- Binäre Suche, Vergleich von „ab“ mit $TA[i], A[i] + 1$:

A[i]	$T_{A[i],n}$
9	\$
8	a\$
1	ab acabra\$ „ab“== „ ab “ → Treffer!
5	ab ra
3	acabra
2	bacabra\$
6	bra\$
4	cabra\$
7	ra\$

- Treffer $T_{1,2}$ erkannt
- Auffinden aller k weiterer Treffer ($T_{5,6}$) im Such Intervall nur sequenziell möglich $\rightarrow \mathcal{O}(m \cdot k + m \cdot \log(n))$

Cluster Eigenschaft in Suffix Trees

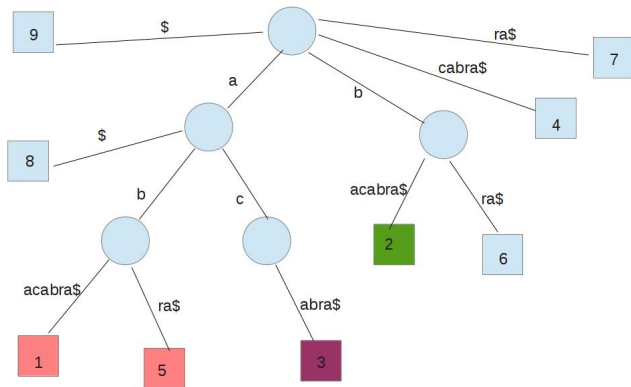
- ▶ Wie kann sequenzielle Suche effizienter werden?
- ▶ Rückblick auf die Suffix Trees: *Cluster Property*



- ▶ Kind-Suffixes haben Pfad von der Wurzel als LCP

Cluster Eigenschaft und Longest Common Prefix

- ▶ Such Intervall entspricht Teilbaum
- ▶ Im Such Intervall *kann* es also LCPs geben: **rosa**: "ab", mit **lila**: "a", mit **grün**: kein LCP



Suffix Array und Longest Common Prefix

- ▶ Sei S ein Such Intervall $[O,U]$, $M = \lfloor \frac{O+U}{2} \rfloor = \text{Treffer}$
- ▶ Wir definieren zwei LCPs: oben = $LCP(T_{A[O],n}, T_{A[M],n})$ und unten = $LCP(T_{A[M],n}, T_{A[U],n})$
- ▶ Bei sequenzieller Suche nur Vergleich von $T_{A[i]+|oben|}, A[i]+m$ nötig

Bsp. $T = \text{"annanas_Anna\$"}$

$A[i]$	$T_{A[i],n}$
13	\$
8	_Anna\$
12	a\$
4	anas_Anna\$
9	Anna\$
1	annanas_Anna\$
6	as_Anna\$
11	na\$
3	nanas_Anna\$
5	nasAnna\$
10	nna\$
2	nnanas_Anna\$
7	s_Anna\$

Zusammenfassung: Binäre Suche in Suffix Arrays

- ▶ Finden der n Suffixes in T : $\mathcal{O}(n)$
- ▶ Lexikografisches Sortieren des Suffix Arrays: $\mathcal{O}(n \cdot \log(n))$
- ▶ Binäre Suche eines Patterns in T : $\mathcal{O}(\log(n))$
- ▶ Insgesamt $\mathcal{O}(n \cdot \log(n))$ (oder $m \cdot \log(n)$?)

Backward Search

- ▶ Grundlage: Suffix Array
- ▶ Neu: keine binäre Suche, finden von zB „b“ mit Array $C[„b“]$
= 5
- ▶ Vorheriger Buchstabe $T_{A[i]-1}$ des Suffix $T_{A[i],n}$

Bsp.: Suche von $p = \text{„abra“}$

$T = \text{„abacabra\$“}$

i	$A[i]$	$T_{A[i]-1}$	$T_{A[i],n}$
1	9	a	\$
2	8	r	a\$
3	1	\$	abacabra\$
4	5	c	abra\$
5	3	b	acabra\$
6	2	a	bacabra\$
7	6	a	bra\$
8	4	a	cabra\$
9	7	b	ra\$

- „abra“ von hinten: alle a, von $C[\text{„a“}] + 1$ bis $C[\text{„b“}]$

Bsp.: Suche von $p = \text{„abra“}$ (Schritt 1)

- Darunter alle „a“ mit Vorgänger „r“

i	$A[i]$	$T_{A[i]-1}$	$T_{A[i],n}$
2	8	r	a\$
3	1	\$	abacabra\$
4	5	c	abra\$
5	3	b	acabra\$

- Weiter zu $i = C[\text{„r“}] + 1 = 9$

Bsp.: Suche von $p = \text{„abra“}$ (Schritt 2)

- Suche „a**bra**“ in r mit Vorgänger b

i	$A[i]$	$T_{A[i]-1}$	$T_{A[i],n}$
\cdot	\cdot	\cdot	\cdot
5	3	b	acabra\$
\cdot	\cdot	\cdot	\cdot
9	7	b	ra\$

- b ist bei $i = 5$ bereits *einmal* in $T_{A[i]-1}$ vorgekommen \rightarrow suche von $C[„b“]+1+1$ bis $C[„c“]$, d.h. bei $i = 7$

Bsp.: Suche von $p = \text{„abra“}$ (Schritt 3)

- Finde alle „**a**bra“ in b mit Vorgänger „a“:

i	$A[i]$	$T_{A[i]-1}$	$T_{A[i],n}$
1	9	a	\$
.	.	.	.
6	2	a	bacabra\$
7	6	a	bra\$

- Da bei $i = 1$ und 6 a bereits *zweimal* in $T_{A[i]-1}$ vorkam
→ suche von $C[„a“]+1+2$ bis $C[„b“]$, d.h. bei $i = 4$

Bsp.: Suche von $p = \text{„abra“}$ (Schritt 4)

i	$A[i]$	$T_{A[i]-1}$	$T_{A[i],n}$
4	5	c	abra\$

- ▶ Nach 4 Schritten (= Länge m von p) ist das Pattern gefunden
 $\rightarrow \mathcal{O}(m)$
- ▶ Problem: z.B. *wie oft ist „b“ in Spalte $T_{A[i]-1}$ vor $i = 5$*
erfordert lineares Durchsuchen von $T_{A[i]-1} \rightarrow \mathcal{O}(m \cdot n)$.
- ▶ Effizientes Vorgehen nötig, sonst Laufzeit wie bei naiver Suche!

Lösung: Funktion $\text{Occ}(c,i)$

- ▶ Die Spalte des Vorgänger-Buchstaben $T_{A[i]-1}$ nennen wir ab jetzt $L_{1,n}$
- ▶ Für alle c aus Σ sei B^c ein Bit-Vektor mit $B^c[i] = 1$ falls $L_i = c$
- ▶ Eine weitere Funktion $\text{rank}_b(B, i)$ liefert die Anzahl von zB $b=1$ in B vor i , s.d. $\text{rank}_1(B^c, i) = \text{Occ}(c, i)$.
- ▶ Dies benötigt linear mehr Speicher, doch der Zugriff durch rank ist konstant, s.d. $\mathcal{O}(m)$ insgesamt garantiert ist.
- ▶ Wavelet Trees?

Forward Searching

- ▶ Vorherige Position: $LF(i) = C[L_i] + Occ(L_i, i)$
- ▶ Während beim Backward Searching ein Suffix auf das vorhergehende abgebildet wird, ist es hier umgekehrt
- ▶ Inverse Funktion $\Psi(i) = i'$, s.d. $A[i'] = (A[i] \bmod n) + 1$ bildet die Pos. eines Suffix auf die seines Nachfolgers ab

Bsp.: Ψ zu $T = \text{„abacabra\$“}$

i	$A[i]$	Ψ	newF	$T_{A[i],n}$
1	9	3	1	\$
2	8	1	1	a\$
3	1	6	0	abacabra\$
4	5	7	0	abra\$
5	3	8	0	acabra\$
6	2	5	1	bacabra\$
7	6	9	0	bra\$
8	4	4	1	cabra\$
9	7	2	1	ra\$

Suche von p in T

- ▶ Falls $\forall c : c \in \Sigma \Rightarrow c \in T$ ex. σ aufsteigende Zahlenfolgen in Ψ : 3; 1,6,7,8; 5,9; 4; 2;
- ▶ Diese zeigen an, wo sich der erste Buchstabe des Suffix ändert. Als Bitvektor $\text{newF} = 110001011$
- ▶ Die Suche von p erfolgt binär, wobei p ein Prefix des jew. Suffix ist, welches durch rekursives Folgen von $\Psi(i)$ *ohne das Suffix Array* gefunden werden kann
- ▶ Der jew. erste Buchstabe $T_{A[i]}$ des Suffixes $T_{A[i],n}$ wird durch $\text{rank}(\text{newF}, i)$ ermittelt. Falls zB $\text{rank}(\text{newF}, i) = 2$ ist $c = \text{„a“}$.

Fazit Forward Searching

- ▶ Ψ ersetzt $A[i]$.
- ▶ Weder $A[i]$ noch T sind zur Suche von p in T nötig
- ▶ Ψ kann durch *gap-encoding* weiter komprimiert werden.

Burrows-Wheeler Transformation

- ▶ Die BWT ist eine Permutation von T , s.d. an jeder Stelle des Suffix Arrays der vorherige Buchstabe angehängt wird:

Definition

Sei $T_{1,n}$ ein String und $A[1,n]$ sein Suffix Array. Dann ist die BWT $T_{1,n}^{bwt}$ von T :

$$T_i^{bwt} = T_{A[i]-1} \quad \forall 1 \leq i \leq n \text{ ausser } A[i] = 1 \Rightarrow T_i^{bwt} = T_n = \$$$

Anschauliches Beispiel, $T = \text{„abacabra\$“}$

Permutationen	alph. geordnet	$F \dots L = T^{bwt}$
abacabra\$	\$abacabra	\$...a
bacabra\$a	a\$abacabr	a...r
acabra\$ab	abacabra\$	a...\$
cabra\$aba	abra\$abac	a...c
abra\$abac	acabra\$ab	a...b
bra\$abaca	bacabra\$a	b...a
ra\$abacab	bra\$abaca	b...a
a\$abacabr	cabra\$aba	c...a
\$abacabra	ra\$abacab	r...b

- ▶ $T^{bwt} = \text{ar$cbaaab}$
- ▶ BWT Permutation besser für weitere Komprimierung von T als T selbst

BWT Rücktransformation

- ▶ Da L_i F_i voransteht, kann T aus T^{bwt} wie folgt wiederhergestellt werden:

F...L	nach links	sortiert	L links angehängt
\$...a	...a\$...\$a	...a\$a
a...r	...ra	...a\$...ra\$
a...\$...\$a	...ab	...\$ab
a...c	...ca	...ab	...cab
a...b	...ba	...ac	...bac
b...a	...ab	...ba	...aba
b...a	...ab	...br	...abr
c...a	...ac	...ca	...aca
r...b	...br	...ra	...bra

- ▶ Reihe $L = T^{bwt}$ bekannt, F wird aus L alph. sortiert
- ▶ Verschieben „nach links“, sortieren
- ▶ $L = T^{bwt}$ links „anhängen, sortieren...

BWT Rücktransformation

Nach n Durchgängen ist die Matrix wiederhergestellt:

\$abacabra

a\$abacabr

abacabra\$

abra\$abac

acabra\$ab

bacabra\$a

bra\$abaca

cabra\$aba

ra\$abacab

T ist derjenige String, der mit „\$“endet.

Zusammenfassung