Suffix Arrays und BWT

Tobias Harrer

19.11.12

Suffix Arrays

Definition

Sei T ein String der Länge n über Σ und $T_{i,j}(i \leq j)$ der Substring von i bis j, dann ist $T_{i,n}$ ein Suffix von T. Das Suffix Array von T ist die Permutation der Startindizes i der alphabetischen geordneten Suffixes $T_{i,n}$ von T.

Anmerkung: jeder String T endet mit \$
$\forall c \in \Sigma : \$ < c$

Beispiel

T = "abacabra$"

i	$T_{i,n}$	alphabetisch 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

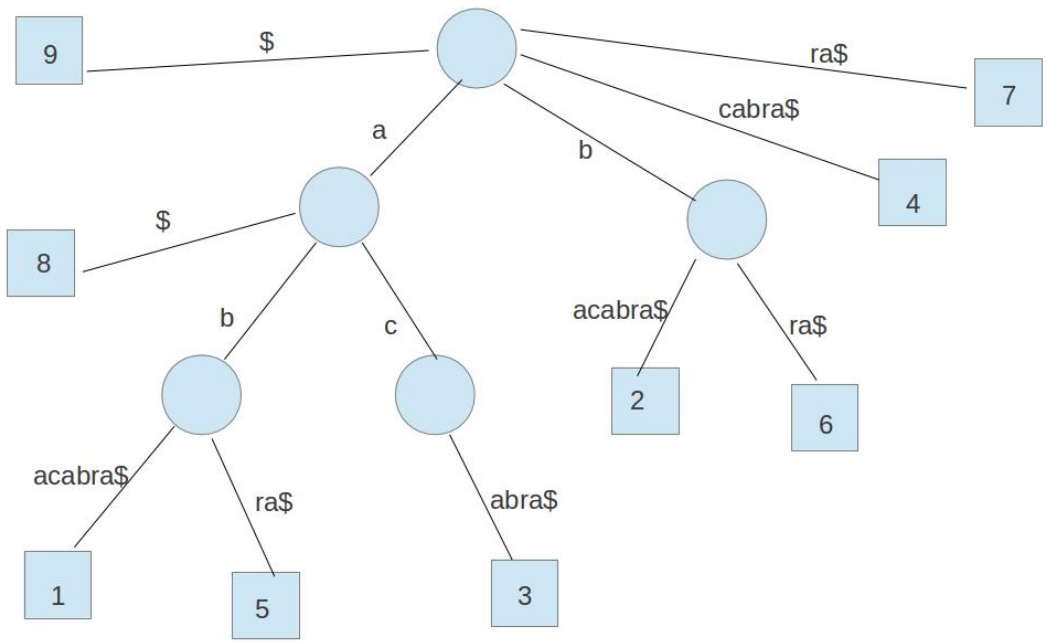
Das Suffix Array von T lautet:

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 $\mathcal{O}(n * logn)$
- Ein Suffix Array kann aus des Blättern eines Suffix Trees durch Tiefensuche hergeleitet werden: T = "abacabra$"



Anwendung von Suffix Arrays

- Finde Pattern p ("cab") in String T ("abacabra")
- Naiver Ansatz: "schiebe" p über T:
  abacabra
  cab→
  ...
  abacabra
  . . cab
- Problem: $\mathcal{O}(n * m)$

Anwendung von Suffix Arrays

- Vorteil: alle Suffixes lexikografisch sortiert
- p "cab" ist Prefix $T_{1,i}$ des Suffix $T_{4,n}$:

  i $T_{i,n}$
  9 $
  8 a$
  1 abacabra$
  5 abra$
  3 acabra$
  2 bacabra$
  6 bra$
  4 **cab**ra$
  7 ra$

- Beste Suchstrategie in sortiertem Array?

Anwendung von Suffix Arrays

- Binäre Suche, Vergleich von "cab" mit $T\,i, i+3$:
  i $T_{i,n}$
  9 $
  8 a$
  1 abacabra$
  5 abra$
  3 **aca**bra$ "cab" $>$ "aca" $\rightarrow$ zweite Hälfte
  2 bacabra$
  6 bra$
  4 cabra$
  7 ra$

Anwendung von Suffix Arrays

i $T_{i,n}$
3 acabra$
2 bacabra$
6 **bra**$ "cab" $>$ "bra" $\rightarrow$ zweite Hälfte
4 cabra$
7 ra$

Anwendung von Suffix Arrays

- i $T_{i,n}$
  6 bra$
  4 **cab**ra$ "cab" = "cab" $\rightarrow$ Pattern ist $T_{4,4+3}$
  7 ra$
- Anschliessend Suche nach weiteren Vorkommen von p.
- Suche in $\mathcal{O}(logn)$

Zusammenfassung: Binäre Suche in Suffix Arrays

- Finden der n Suffixes in T: $\mathcal{O}(n)$
- Alphabetisch Sortieren des Suffix Arrays: $\mathcal{O}(n * logn)$
- Binäre Suche eines Patterns in T: $\mathcal{O}(logn)$
- Insgesamt $\mathcal{O}(nlogn)$ (oder m 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 = "abra"

T = "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$

- "abr**a**" von hinten: alle "a", von C["a"]+1 bis C["b"]

Bsp.: Suche von p = "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["r"]+1 = 9

Bsp.: Suche von p = "abra" (Schritt 2)

- Suche "a**b**ra" in r mit Vorgänger b

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

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

Bsp.: Suche von p = "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 $\rightarrow$ suche von C["a"]+1+*2* bis C["b"], d.h. bei i = 4

Bsp.: Suche von p = "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 $T_{A[i]-1} \rightarrow \mathcal{O}(m * n)$.
- Effizientes Vorgehen nötig, sonst Laufzeit wie bei naiver Suche!

Lösung: Funktion 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 $rank_b(B, i)$ liefert die Anzahl von zB b=1 in B vor i, s.d $rank_1(B^c, i) = Occ(c, i)$.
- 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] mod n)+1 bildet die Pos. eines Suffix auf die seines Nachfolgers ab

Bsp.: Ψ zu T = "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 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 $rank(newF, i)$ ermittelt. Falls zB $rank(newF, i) = 2$ ist c = "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 $T^{bwt}_{1,n}$ von T:

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

Anschauliches Beispiel, T = "abacabra$"

Permutationen	alph. geordnet	$F...L = T^{bwt}$
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} = 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.