Algorithms in Bioinformatics: Lecture 09: Suffix Trees and Variants

Lucia Moura

Fall 2010

Summary: Suffix Trees

Suffix trees can be built in time $O(n)$ (see Farach's algorithm (1997), Sect 3.4).
The space requirement is $O(n|\mathcal{A}|\log n)$.
We saw how to solve the following problems using Suffix trees:

- Exact string matching (is $P[1..m]$ in $T[1..n]$?): after creating the suffix tree once in $O(n)$, each string matching takes $O(m)$
- Longest repeated substring in $T[1..n]$: $O(n)$
- Longest common substring of $S_1, S_2, \ldots, S_k$: $O(n_1 + n_2 + \cdots + n_k)$
- Substring problem: given a database of strings $T_1, \ldots T_k$ known in advance, check if P is a substring of one of the T_i (DNA identification, US army). Preprocessing takes $O(n_1 + \cdots + n_k)$ and each P can be processed in $O(m)$.
- Find maximal repeated pairs of substring in $S[1..n]$: $O(n)$.
- Find all MUMs (Maximal Unique Pairs) in S_1 and S_2: $O(n_1 + n_2)$.
- Find k-mismatches of P in T via LCE: $O(m + nk)$
- Find all maximal palindromes (or complemented palindromes) of S: $O(n)$

Suffix Array

- suffix trees are very useful, but have a large space requirement: $O(n|\mathcal{A}|\log n)$ bits; where $|\mathcal{A}|$ is 4 for DNA and 20 for protein.
- Manber and Myers (1993) proposed suffix arrays which requires only $n \log n$ bits.
- Most applications using suffix trees can be solved using suffix arrays with some time overhead.
- A suffix array can be built in time $O(n)$ using $O(n)$ bits of working space (Hon, Sadakane and Sung, FOCS 2003).

Definition

Let $S[1..n]$ be a string of length n over an alphabet $\mathcal{A}$, with terminator \$ being alphabetically smaller than all characters. A suffix array SA[1..n] is an array of integers such that $S[SA[i]..n]$ is lexicographically the $i-th$ smallest suffix of S.

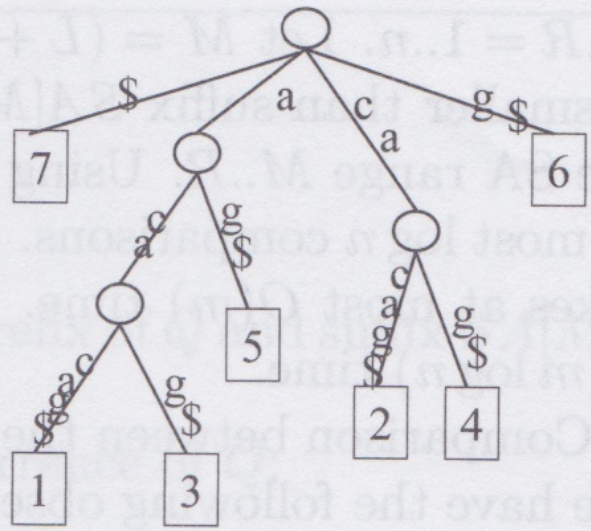
Suffix Array

Suffix	Position
$acacag\$$	1
$cacag\$$	2
$acag\$$	3
$cag\$$	4
$ag\$$	5
$g\$$	6
$\$$	7

(a)

i	$SA[i]$	Suffix
1	7	$\$$
2	1	$acacag\$$
3	3	$acag\$$
4	5	$ag\$$
5	2	$cacag\$$
6	4	$cag\$$
7	6	$g\$$

(b)



(c)

FIGURE 3.13: (a) The set of suffixes of $S = acacag\$$, (b) the corresponding suffix array, and (c) the corresponding suffix tree. Note that the lexicographical order of the leaves in the suffix tree equals the order of the suffixes in the suffix array.

Exact string matching using suffix array

For a query, check whether $Q[1..m]$ exists in $S[1..n]$.

Idea: binary search for Q on the suffix array.

(a)
L=1, l=0
R=7, r=0

$SA[i]$	$SA[i]$	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

(b)
mlr = min(l,r)=0
M=(L+R)/2=4
m=1
Q < suffix $SA[M]$

We set
L=1, l=0
R=M=4, r=m=1

$SA[i]$	$SA[i]$	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

(c)
mlr = min(l,r)=0
M=(L+R)/2=2
m=3
Q>suffix $SA[M]$

We set
L=M=2, l=m=3
R=4, r=1

$SA[i]$	$SA[i]$	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

(d)
mlr = min(l,r)=1
M=(L+R)/2=3
m=4

Q is found in $SA[M]$

$SA[i]$	$SA[i]$	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

FIGURE 3.16: This example demonstrates how to check the existence of $Q = acag$ in the text $S = acacag\$$ by performing binary search on the suffix array of S.

Exact string matching using suffix array: algorithm

SA_binary_search(Q)

```
1: Let L = 1 and R = n;
2: while L ≤ R do
3:    Let M = (L + R)/2.
4:    Find the length m of the longest common prefix of Q and suffix SA[M];
5:    if m = |Q| then
6:       Report suffixes SA[M] contain the occurrence of Q;
7:    else if suffix SA[M] > Q then
8:       set R = M;
9:    else
10:      set L = M;
11:   end if
12: end while
13: if L > R then
14:   Report Q does not exist in S;
15: end if
```

FIGURE 3.14: Checking the existence of a query Q through binary search on the suffix array of S.

Running time:

Binary search does at most $\log n$ comparisons, each of which takes at most m comparisons: $O(m \log n)$

Using suffix trees this could be achieved in $O(m)$ time.

FM-index

When the space requirement of a suffix array is still prohibitive, then FM-index (Ferragini and Manzini, 2000) can be used.

	suffix tree	suffix array	FM-index
Space requirement (in bits):	$O(n\|\mathcal{A}\|\log n)$	$n \log n$	$O(n)$
Human genome requirement:	40 GB	13 GB	1.5 GB

FM-index can be constructed in linear time.

Definition of FM-index

The FM-index consists of the 3 data structures:

1. The Burrows-Wheeler text of S[1..n] is a string $BW[1..n]$ where:

$$\begin{aligned} BW[i] = & \quad S[SA[i]-1], & \text{if } SA[i] \neq 1; \\ = & \quad S[n], & \text{if } SA[i] \neq 1 \end{aligned}$$

   $S = acacag\$$
   $BW = g\$ccaaa$

2. For every $x \in \mathcal{A}$, $C[x]$ stores the total number of occurrences of characters which are lexicographically less than x.
   $C[a] = 1, C[c] = 4, C[g] = 6, C[t] = 7$

3. A data structure that supports $O(1)$ computation of $occ(x, i)$= number of occurrences of $x \in BW[1..i]$. This is stored using $O(n \log\log n / \log n)$. [this is the trickiest part]
   $occ(a, 5) = 1, occ(c, 4) = 2$

Exact string matching using FM-index: Algorithm

Algorithm BW_search($Q[1..m]$)

1: $x = Q[m]$; $st = C[x] + 1$; $ed = C[x+1]$;
2: $i = m - 1$;
3: **while** $st \leq ed$ and $i \geq 1$ **do**
4: $\quad x = Q[i]$;
5: $\quad st = C[x] + occ(x, st - 1) + 1$;
6: $\quad ed = C[x] + occ(x, ed)$;
7: $\quad i = i - 1$;
8: **end while**
9: if $st > ed$, then pattern not found else report $[st..ed]$.

FIGURE 3.19: Given the FM-index of S, the backward search algorithm finds $range(S, Q)$.

Running time: $O(m)$.

Exact string matching using FM-index

(a) First iteration (initial values),
$Q[3..3] = a$
$range(S, Q[3..3]) = [sp..ep]$
$sp = C[a] + 1 = 1 + 1 = 2$
$ep = C[c] = 4$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
→	1	$acacag\$$	\$
	3	$acag\$$	c
→	5	$ag\$$	c
	2	$cacag\$$	a
	4	$cag\$$	a
	6	$g\$$	a

(b) Second iteration,
$Q[2..3] = ca$
$range(S, Q[2..3]) = [sp'..ep']$
$sp' = C[c] + occ(c, sp - 1) + 1 = 4 + 0 + 1 = 5$
$ep' = C[c] + occ(c, ep) = 4 + 2 = 6$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
	1	$acacag\$$	\$
	3	$acag\$$	c
	5	$ag\$$	c
→	2	$cacag\$$	a
→	4	$cag\$$	a
	6	$g\$$	a

(c) Third iteration,
$Q[1..3] = aca$
$range(S, Q[1..3]) = [sp''..ep'']$
$sp'' = C[a] + occ(a, sp' - 1) + 1 = 1 + 0 + 1 = 2$
$ep'' = C[a] + occ(a, ep') = 1 + 2 = 3$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
→	1	$acacag\$$	\$
→	3	$acag\$$	c
	5	$ag\$$	c
	2	$cacag\$$	a
	4	$cag\$$	a
	6	$g\$$	a

FIGURE 3.20: Given the FM-index for the text $S = acacag\$$. This figure shows the three iterations for searching pattern $Q = aca$ using backward search. (a) $range(S, a) = [2..4]$, (b) $range(S, ca) = [5..6]$, and (c) $range(S, aca) = [2..3]$.