

18.417 Introduction to Computational Molecular Biology

Lecture 12: October 19, 2004

Scribe: Tushara C. Karunaratna

Lecturer: Ross Lippert

Editor: Peter Lee

Suffix Arrays and BWTs

Notation

We use Σ to denote the alphabet.

We use S to denote the text string, and n to denote the length of the text.

We use P to denote the pattern, and m to denote the pattern length.

Introduction

Although suffix trees can be constructed in linear space and time, and provide linear time queries, they are not suitable for representing huge sequences such as the whole human genome of three billion base pairs. This unsuitability is due to the unacceptably large constant factors associated with the space requirement in actual implementation.

The implementation described in the previous lecture used the following representation for suffix tree nodes.

```
Node { start, depth, slink, parent, children }
```

Now, we can get rid of the parent pointer – at each insertion the only parent pointer we traverse is of the previously inserted node and thus it suffices to simply remember the parent of the previously inserted node. We can also replace the children list with a pointer to the first child and the right sibling. Furthermore, for leaves, we have to store only the starting position and the right sibling.

Let's analyze the space requirement, taking these optimizations into account. Internal nodes take up 5 machine words, and leaves take up 2 machine words. There are n leaves, and the number of internal nodes could be as large as n . Thus, the space requirement is $5n + 2n = 7n$ words, which is $28n$ bytes assuming a machine word size of four bytes.

It is possible to reduce the space requirement to $20n$ bytes using a technique due to Kurtz.

Suffix Arrays

Suffix arrays are a more space efficient alternative to suffix trees. They were first developed by Manber and Myers, in 1990. The suffix array of a text is defined as the permutation of the sorted suffixes of the text. Note that the suffix array takes up only n integers for storage. Table 12.1 shows the suffix array for the string `acataggagacatacga$`.

suffix	string
17	\$
16	a\$
9	acatacga\$
0	acataggagacatacga\$
13	acga\$
7	agacatacga\$
4	aggagacatacga\$
11	atacga\$
2	ataggagacatacga\$
10	catacga\$
1	cataggagacatacga\$
14	cga\$
15	ga\$
8	gacatacga\$
6	gagacatacga\$
5	ggagacatacga\$
12	tacga\$
3	taggagacatacga\$

Table 12.1: Suffix array for the string `acataggagacatacga$`.

Construction

The suffix array can be thought of as a left-to-right dump of the leaves of the suffix tree. Thus, a naive method of computing the suffix array is by first computing the suffix tree. This naive method takes only $O(n)$ time, but could take a large amount of space during the intermediate step of constructing the suffix tree. Manber and Myers,

in their seminal paper, show how to construct the suffix array *in place* in $O(n \log n)$ time.

Pattern lookup

Figure 12.1 gives pseudocode for $O(m \log n)$ time pattern lookup using the suffix array.

```

find(P):
  i = 0
  lo = 0, hi = length(A)
  for 0 ≤ i < length(P):
    Binary search for x,y where P[i]=S[A[j]+i] for lo ≤ x ≤ j < y ≤ hi
    lo = x, hi = y
  return {A[lo], A[lo+1], ..., A[hi-1]}

```

Figure 12.1: Pattern lookup using the suffix array.

LCP enhancement

Manber and Myers describe how to pre-compute the *longest common prefixes* (LCPs) between adjacent elements of the suffix array, in $O(n \log n)$ time.

Given the LCP information, pattern lookup can be performed in $O(m + \log n)$ time by a modification to the above binary search.

Structure of a suffix array permutation

Suffix arrays are not space efficient. This is because not all permutations are suffix arrays. There are $n!$ permutations on n integers, whereas there are only $|\Sigma|^n$ possible suffix arrays.

Figure 12.2 shows the graph of the suffix array permutation A for the string `acataggagacatacga$`. This graph does not show us any obvious structure.

Figure 12.3 shows the graph of the auxiliary permutation $A^{-1}[A[i] + 1]$. This graph does indeed show us that suffix arrays have structure: the graph consists of $|\Sigma| + 1$ monotonic runs; the runs can break up only at character boundaries.

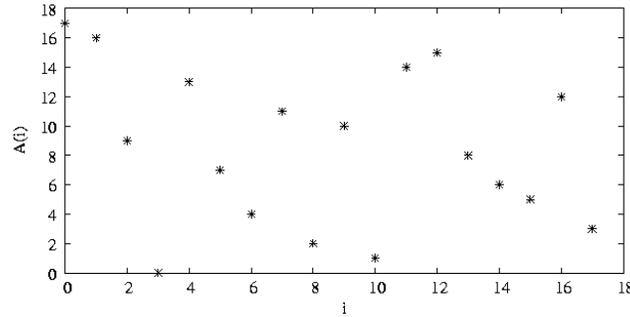


Figure 12.2: Graph of suffix array permutation.

To see why this is the case, consider any i and j such that $i < j$ and $S[A[i]] = S[A[j]]$. Then, we have $S[(A[i] + 1) \dots (n - 1)] \leq S[(A[j] + 1) \dots (n - 1)]$. Hence, we have $A^{-1}[A[i] + 1] \leq A^{-1}[A[j] + 1]$.

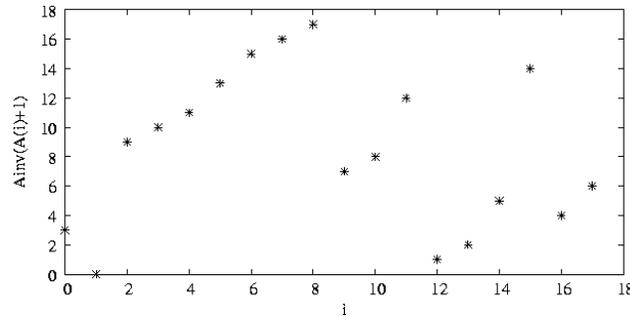


Figure 12.3: Graph of the auxiliary permutation.

Burrows Wheeler Transform

The Burrows Wheeler string B of a string S having suffix array permutation A is given by $B[i] = T[A[i] - 1]$. (Arithmetic on indices is performed *mod n*.) Table 12.2 shows the Burrows Wheeler transform for the string `acataggagacatacga$`.

In order to be able to perform pattern lookup using the Burrows Wheeler string, we require additional $O(|\Sigma|)$ storage for the starting positions of each letter in the suffix array. The following are two important queries done on the Burrows Wheeler string.

i	$S[A[i] - 1]$	$A^{-1}[A[i] + 1]$	$A[i]$	string
0	a	3	17	\$
1	g	0	16	a\$
2	g	9	9	acatacga\$
3	\$	10	0	acataggagacatacga\$
4	t	11	13	acga\$
5	g	13	7	agacatacga\$
6	t	15	4	aggagacatacga\$
7	c	16	11	atacga\$
8	c	17	2	ataggagacatacga\$
9	a	7	10	catacga\$
10	a	8	1	cataggagacatacga\$
11	a	12	14	cga\$
12	c	1	15	ga\$
13	a	2	8	gacatacga\$
14	g	5	6	gagacatacga\$
15	a	14	5	ggagacatacga\$
16	a	4	12	tacga\$
17	a	6	3	taggagacatacga\$

Table 12.2: Burrows Wheeler Transform for the string acataggagacatacga\$.

char	\$	a	c	g	t
block	0	1	9	12	16

Table 12.3: Block starts for each letter.

- $occ(x, i)$: the number of times character x appears before position i in the Burrows Wheeler string.
- $find(x, i)$: the location of the i^{th} occurrence of the character x in the Burrows Wheeler string.

Pattern lookup

Figure 12.4 gives pseudocode for counting the number of occurrences of a pattern P . The approach used is to incrementally determine the interval $[lo, hi)$ in the suffix

```
lookup(P):
  lo = 0, hi = length(B)
  i = length(P)
  while i > 0:
    i = i - 1
    lo = block(P[i]) + occ(P[i], lo)
    hi = block(P[i]) + occ(P[i], hi)
  return hi - lo
```

Figure 12.4: Pattern lookup using the BWT.

array in which the i -suffix of P is a prefix. Thus, pattern lookup comes down to doing $2m$ counts.

Table 12.4 shows the execution trace of the lookup procedure for the pattern `cata`.

		a	t	a	c
lo	0	1	16	7	9
hi	18	9	18	9	11

Table 12.4: Execution trace of the lookup procedure.

Fast Counts

For each letter in the alphabet, and for each integer i that is a multiple of some integer W , we store the number of occurrences of the letter before position i in the Burrows Wheeler string. An example is shown in Table 12.5. This data structure takes $(1 + 4|\Sigma|/W)n$ bytes for storage. Using the data structure, we can perform occ

queries in $O(1 + W)$ time. We can perform *find* queries in $O(W \log n)$ time using a simple binary search. There is a trade-off between space and time. The common

i	# $\$$	#a	#c	#g	#t	$B[i]$
0	0	0	0	0	0	a
1						g
2						g
3	0	1	0	2	0	$\$$
4						t
5						g
6	1	1	0	3	1	t
7						c
8						c
9	1	1	2	3	2	a
10						a
11						a
12	1	4	2	3	2	c
13						a
14						g
15	1	5	3	4	2	a
16						a
17						a

Table 12.5: Counts with $W = 3$.

pattern is to choose W large enough so that the data structure fits into memory.