CS 6968: Data Str & Alg Scalable Comp Spring 2023

Lecture 6 — Jan 30, 2023

Prof. Prashant Pandey *Scribe: Vinutha Raja*

1 Overview

In the last lecture we discussed about Tries, Compact tries and Suffix trees. In Suffix trees, a Text of size T and a pattern of size P, the query for pattern P and the indices at which it appears takes $O(P)$ time complexity. The space complexity i.e the amount of text stored in the tree is approximately bounded by $O(T^2)$ which is not very efficient.

In this lecture, we will be learning about Suffix Arrays, Burrows-Wheeler Transform and FM-index with the goal of achieving better space complexity.

2 Suffix Arrays

This data structure is space efficient with O(T).
Example T: banana, order will be $\$ < a < b ... < z$.

- We find all the suffixes of a string T.

  0 banana\$
  1 anana\$
  2 nana\$
  3 ana\$
  4 na\$
  5 a\$
  6 \$

- Sort the suffixes of T.

  6 \$
  5 a\$
  3 ana\$
  1 anana\$
  0 banana\$
  4 na\$
  2 nana\$

- Store the indices which takes an O(T) space.

Now, we can use binary search to search for a pattern P in T. Hence it takes O(PlogT) time complexity.

3 Space complexity of Human Genome:

A human genome consists of about 3 billion base pairs {A T G C}. Say each base pair takes about 2 bits.
The amount of space required by different data structures is as below:

Suffix Trees : $\sim$ 47 GB
Suffix Arrays : $\sim$ 12 GB
FM Index : $<$ 1.5 GB

4 Burrows-Wheeler Transform:

4.1 Burrows-Wheeler Transform:

BWT is obtained by doing a reversible permutation of the characters of a string which will be used mainly for compression as it brings all the similar characters together. The entropy of this permuted string will be lower compared to the original string.

Here is an example of constructing the Burrows-Wheeler Matrix and BWTransform string of T "abaaba$".

Burrows-Wheeler Matrix by performing Rotations in sorted order is as below:

$	a	b	a	a	b	a
a	$	a	b	a	a	b
a	a	b	a	$	a	b
a	b	a	$	a	b	a
a	b	a	a	b	a	$
b	a	$	a	b	a	a
b	a	a	b	a	$	a

The Burrows-Wheeler Transform is the string obtained from the last column of Burrows-Wheeler Matrix. Burrows-Wheeler Transform(T) is "abba$aa".

Note: Sort order remains the same whether we rotate the rows or the suffixes.

4.2 T-Ranking:

For each character in T, its rank is equal to the number of times the character occurred previously in T. The string with its T -ranking can be represented as $a_0b_0a_1a_2b_2a_3$.

F-L Mapping: The i'th occurrence of a character "c" in L and the i'th occurrence of "c" in F corresponds to the same occurrence in T.
We get the First-Last mapping by considering only the first column and last column in the above BWM. Note that the order of a's and b's are same in both F and L and it forms a bipartite graph.

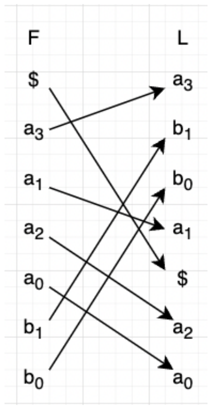

Figure 1: First-Last Mapping

4.3 B-Ranking

In this, a given character's ranks are in ascending order as we look down in the F/L columns. F now has a very simple structure, a \$, a block of a's with ascending ranks, and a block of b's with ascending ranks as shown below.

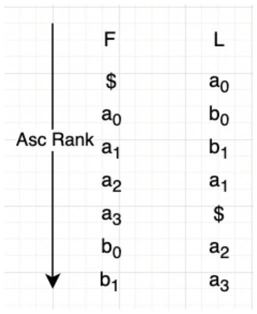

Figure 2: B-Ranking

Problem: Find which BWM row beigns with b_1?
Answer: Skip the row starting wit \$(1 row). Skip the row starting with a (4 rows). Skip the row starting with b(1 row). Hence the row 6 starts with b_1.

4.4 Burrows-Wheeler Transform Reversing:

We can build the reverse BWT(T) starting from the right hand side of T and moving towards left. We start from the first row, F will have a \$ and L contains the character just prior to the \$. From F-L Mapping we know that a_0 is the same occurrence of "a" as first "a" in F. Now, we jump to the row beginning with a_0. L contains the character just prior to a_0 which is b_0. This way we will build the original string as shown below in the graph. The original string will be $a_3b_1a_1c_2b_0a_0\$$.

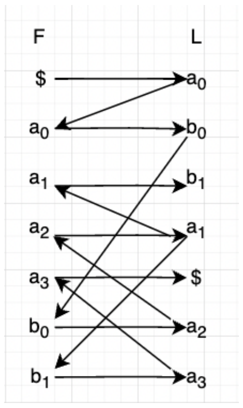

Figure 3: Burrows-Wheeler Transform Reversing

Note : We need not store the F, by linear scanning L, we can identify the number of occurrences of each character and then by using the F-L Mapping we can reconstruct the original string.

5 FM-Index:

Details of the space usage by FM-Index for Human Genome:

- It stores the first column: $\sim |\sum|$ integers which requires about 16 Bytes.
- Stores the last column: m characters which requires about 750 MB.
- Suffix Array sample: $m * a$ integers, a $\sim$ 1/32, this requires about 400MB.
- Check point: $m * b$ integers, b $\sim$ 1/128, requires about 100MB.
- **Hence, the Human Genome requires $< 1.5GB$, which is significantly lesser compared to 12 GB required by Suffix Arrays.**