CS 7280: Data Str & Alg Scalable Comp Spring 2025

Lecture 6

Prof. Prashant Pandey *Scribe: Azhar Abdulla*

1 Overview

In the previous lecture, we focused on tries and suffix trees, which provide efficient pattern matching with query times of $O(P)$ and space complexity of $O(T)$. In this lecture, we will explore how to optimize space further using suffix arrays. Suffix arrays are a compact alternative to suffix trees, offering similar functionality with reduced space requirements.

1.1 Recap: Previous Data Structures

Tries: Tries are rooted trees where edges are labeled with letters from the alphabet Σ, and strings are represented as root-to-leaf paths. To distinguish prefixes from complete strings, a special terminator symbol \$ is appended to each string. While standard tries use $O(T \cdot |\Sigma|)$ space, compressed tries optimize this by merging single-child paths into a single edge, reducing space usage and improving traversal speed.

Suffix Trees: Suffix trees are specialized compressed tries that represent all suffixes of a string. Constructed in $O(T)$ time, they store all substrings of a string in a compact form. Queries, such as substring search or longest repeated substring, can be performed in $O(P)$ time, where P is the length of the query. Suffix trees are particularly useful in applications like genome analysis and text processing.

2 Suffix Arrays

A suffix array is a data structure that stores the sorted indices of all suffixes of a given text T. Instead of explicitly storing the suffixes, it only stores their starting indices, resulting in a space complexity of $O(T)$.

Consider the example of the text $T =$ `banana$`, where \$ is a special end-of-string character. The suffixes of T are:

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

Sorting these suffixes lexicographically gives us the suffix array:

Sorted Index	**Suffix**	**LCP**
6	\$	0
5	a\$	1
3	ana\$	3
1	anana\$	0
0	banana\$	0
4	na\$	2
2	nana\$	0

Here, the LCP (Longest Common Prefix) column represents the length of the shared prefix between consecutive suffixes in the sorted order. The suffix array itself is the list of indices:

$$[6, 5, 3, 1, 0, 4, 2]$$

.

2.1 Searching in a Suffix Array

To search for a pattern P in the text T, we perform a binary search on the suffix array. Since the suffixes are sorted, we can compare P with the suffixes at the indices stored in the suffix array. The query time for this operation is $O(P \log T)$, where P is the length of the pattern and T is the length of the text.

The construction of the suffix array involves sorting the suffixes, which takes $O(T + \text{sort}(\Sigma))$ time, where Σ is the alphabet size. Additionally, the LCP array can be constructed in $O(T)$ time, and together with the suffix array, it can be used to build a Cartesian tree, which is useful for various string processing tasks.

2.2 The Cartesian Tree

The Cartesian Tree is a binary tree representation of the LCP array that enables efficient queries on the suffix array. It is particularly useful for solving problems related to range queries, such as finding the longest common prefix of two suffixes in $O(1)$ time after preprocessing.

2.2.1 Example: Cartesian Tree for $T =$ `banana$`

Consider the suffix array and LCP array for $T =$ `banana$`:

Sorted Index	**Suffix**	**LCP**
6	$	0
5	a$	1
3	ana$	3
1	anana$	0
0	banana$	0
4	na$	2
2	nana$	0

The LCP array is $[0, 1, 3, 0, 0, 2, 0]$. To build the Cartesian Tree:

1. Identify the minimum element of the LCP array. The first minimum is 0 (at multiple positions). The first occurrence, index 0, becomes the root.

2. Recursively divide the array into left and right segments: - Left: None (since the root is at index 0). - Right: $[1, 3, 0, 0, 2, 0]$.

3. Continue recursively: - The next minimum is 0 at index 3. This becomes the right child of the root. - Divide the remaining segments and continue until all nodes are processed.

The resulting Cartesian Tree represents the structure of the LCP array and maps directly to the suffix array indices.

2.2.2 Functionality of the Cartesian Tree

The Cartesian Tree is useful for several string processing tasks:

- **Range Minimum Queries (RMQ):** It enables efficient RMQ operations on the LCP array. For any two suffixes i and j, the LCP value can be computed as the minimum value in the range $[i + 1, j]$ of the LCP array.
- **Substring Analysis:** By combining the Cartesian Tree with the suffix array, we can quickly compute properties like the longest repeated substring, longest palindromic substring, or the longest substring common to two strings.
- **Efficient Preprocessing:** The Cartesian Tree allows preprocessing of the LCP array in $O(T)$ time, enabling $O(1)$ query time for common string operations.

Conclusion: The Cartesian Tree complements the suffix array and LCP array, providing a powerful tool for efficient string analysis. It simplifies range-based queries, making it indispensable for applications like genome analysis and large-scale text processing.

3 Applications in Genome Analysis

The human reference genome consists of approximately 3 billion base pairs. Due to its massive size, only small sequences can be processed at a time. Given a reference genome, we attempt to reconstruct it by aligning sequences. The space efficiency of data structures becomes critical in such applications.

3.1 Space Comparison

Data Structure	**Space Usage**
Suffix Tree	47 GB
Suffix Array (SA)	13 GB
FM-Index	1.5 GB

The FM-Index, based on the Burrows-Wheeler Transform (BWT), is particularly notable for its space efficiency. It compresses the text while still supporting efficient pattern matching queries.

4 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler Transform is a reversible transformation of a text that rearranges its characters to make runs of similar characters longer.

4.1 Example of BWT

Consider the text $T =$ `abaaba$`. To compute its Burrows-Wheeler Transform, we first generate all cyclic rotations of T and then sort them lexicographically. The process is shown below:

Rotation	**Sorted Rotation**
`abaaba$`	`$abaaba`
`baaba$a`	`a$abaab`
`aaba$ab`	`aaba$ab`
`aba$aba`	`aba$aba`
`ba$abaa`	`abaaba$`
`a$abaab`	`ba$abaa`
`$abaaba`	`baaba$a`

The **BWT** of T is the last column of the sorted rotations: `abba$baa`.

4.2 T-Ranking and B-Ranking

To facilitate the reversal of the Burrows-Wheeler Transform (BWT), we introduce two key concepts: T-ranking and B-ranking. These rankings help establish a mapping between the characters in the transformed string and their original positions in the text.

4.2.1 T-Ranking (Text Ranking)

T-ranking represents the position of each character in the sorted order of the original text. More formally, given a character c at position i in the original text T, its T-rank indicates how many occurrences of c precede it in T.

For example, consider the string $T =$ `banana$`. The sorted order of T is:

```
$ a a a b n n
```

The T-ranking of each character is determined based on its position in this sorted order:

Character	**T-Rank**
\$	0
a	1
a	2
a	3
b	4
n	5
n	6

4.2.2 B-Ranking (BWT Ranking)

B-ranking represents the position of each character in the Burrows-Wheeler transformed string. Since BWT sorts rotations of T, characters may appear in a different order than in the original text. The B-rank for each character indicates how many occurrences of that character appear before it in the BWT.

For example, consider the BWT of $T =$ `banana$`, which is:

```
b$aaaba
```

The B-ranking of each character is determined based on its position in the BWT:

Character	**B-Rank**
b	0
\$	0
a	1
a	2
a	3
b	1
n	0

4.2.3 Relation Between T-Ranking and B-Ranking

Since the BWT is constructed by sorting cyclic rotations of T, the T-rank of a character corresponds directly to its B-rank in the transformed string. This one-to-one mapping between the two rankings is crucial for reconstructing the original text.

The LF mapping, which is used for reversing the BWT, exploits this relationship by tracking the relative ordering of characters in both the original text and its transformation. By iteratively applying the LF mapping using T-ranks and B-ranks, we can reconstruct T from its BWT.

This leads naturally to the next step: the reversal of the BWT using LF mapping.

4.3 Reversing the BWT

The Burrows-Wheeler Transform is reversible. To reconstruct the original text, we use the **LF mapping**, which maps characters in the last column (L) to their positions in the first column (F) of the sorted rotations. By iteratively applying the LF mapping, we can recover the original text.

LF Mapping Process:

- Count the frequency of each character in L to determine their order in F.
- Use the relationship between L and F to trace back the original string starting from the unique terminator symbol (\$).

This process is guaranteed to reconstruct the original text in $O(T)$ time.

4.4 Entropy Reduction

The Burrows-Wheeler Transform reduces the entropy of the text by grouping similar characters together. This property makes it highly effective for compression algorithms, as it enables:

- **Run-length encoding:** Longer runs of identical characters can be compressed efficiently.
- **Huffman or Arithmetic coding:** Reduced entropy improves the efficiency of these compression techniques.

5 Summary

This lecture explored advanced string data structures and transformations, focusing on their efficiency, functionality, and practical applications. Suffix arrays were introduced as a space-efficient alternative to suffix trees, storing the sorted indices of all suffixes of a string. With a query time of $O(P \log T)$ and space complexity of $O(T)$, suffix arrays enable efficient pattern matching and substring analysis. The Longest Common Prefix (LCP) array was also discussed, complementing the suffix array by facilitating range minimum queries and substring analysis.

Building on this, Cartesian trees were introduced as a binary tree representation of the LCP array. They allow efficient range minimum queries in $O(1)$ time after preprocessing and enhance the utility of suffix arrays in various string processing tasks. Cartesian trees provide a systematic way to compute properties like the longest repeated substring or the longest common substring in large datasets.

Finally, the lecture covered the Burrows-Wheeler Transform (BWT), a reversible transformation that rearranges text to group similar characters together. This transformation reduces the entropy of the text, making it highly effective for compression. The LF mapping process was discussed to illustrate how the BWT can be reversed to recover the original text.