Bidirectional Search in a String Using Wavelet Matrix and Burrows Wheeler Transform

A Thesis Presented to the Department of Computer Science

African University of Science and Technology

In Partial Fulfilment of the Requirements for the Degree of Master of Science

By

Bello Hannatu Bagudu
(40801)

Abuja, Nigeria

August, 2021.

CERTIFICATION

This is to certify that the thesis titled “Bidirectional Search in a String using Wavelet Matrix and Burrows Wheeler Transform” submitted to the school of postgraduate studies, African University of Science and Technology (AUST), Abuja, Nigeria for the award of the Master's degree is a record of original research carried out by Bello Hannatu Bagudu in the Department of Computer Science.

BIDIRECTIONAL SEARCH IN A STRING USING WAVELET MATRIX AND BURROWS WHEELER TRANSFORM

BY:

Bello Hannatu Bagudu

A THESIS APPROVED BY THE COMPUTER SCIENCE

DEPARTMENT

APPROVAL BY

Supervisor

Surname: Rajesh

First name: Prasad

Signature 

The Head of Department

Surname: Rajesh

First name: Prasad

Signature: 

ABSTRACT

Wavelet tree is a data structure which permits the representation of sequences of symbols over an alphabet of size σ. It recursively partitions a string into two halves until homogeneous data is obtained. An increase in the value of σ has an impact on the wavelet tree's space utilization. To reduce the wavelet tree's space utilization pointer-less wavelet tree was invented, although in theory the complexities stay the same in practice it is slower. An alternative representation of the pointer-less wavelet tree is the wavelet matrix. It concatenates the bitmaps level-wise and also retains all the wavelet tree's features but it is much faster in practice. In this research, we use the wavelet matrix to reconstruct the bidirectional wavelet index. The modification is efficient as our experiment shows that the bidirectional wavelet index constructed using the wavelet matrix decreases the time it takes to execute the queries.

DEDICATION

This work is dedicated foremost to Almighty Allah, who gave the strength, opportunity, and inspiration in the course of this work Alhamdulillah, also to my family for their love and support, and finally to everyone else that contribute towards making my study a successful one.

ACKNOWLEDGEMENT

My profound gratitude goes to Allah, the Most Beneficent the Most Merciful who made it possible for me to reach this far in this program.

I would like to express my great gratitude to my supervisor; Dr. Rajesh Prasad for his guidance, advice, and support, I consider myself fortunate to have worked under your capable guidance, Sir.

My appreciation also goes to all my lecturers for their invaluable knowledge and expertise which they have shared with me throughout my program.

I am grateful to the African University of Science and Technology (AUST) and its staff for providing me with the opportunity to be among the few selected in Africa to study at such a world-class institution; I have gained more knowledge than I had before enrolling for this program.

Further, the acknowledgment would be incomplete if not mention a word of thanks to my beloved parents whose continuous support and encouragement through the program has led me to pursue the degree and confidently complete this work.

To my siblings, thank you for your constant support, encouragement, and contribution toward the success of this work.

Finally, I thank all my friends, course mates and others who supported me in this work or other aspects of my study at AUST, I wish you success in all your endeavors.

TABLE OF CONTENTS

List of Table

List of Figures

CHAPTER 1
Introduction

1.1 Background

String searching is a well-known and existing problem in computer science discipline. In string searching occurrences of a pattern is searched within another string. The string searching's main object is to find a pattern in a sequence S. The goal is to determine whether the pattern occurs in S and, if so, how many times it appears and where each occurrence is located in S. String searching has various applications such as DNA sequence matching, search engines, spell checkers, information retrieval, etc. To allow for fast search, it is essential to represent the sequence S[0,n] over an alphabet ∑[0,σ]. Wavelet tree is a good data structure for solving this problem. It is a versatile data structure that recursively partitions strings into two halves. Grossi, Gupta, and Vitter invented it in 2003 to represent sequence and answer queries [9]. Due to its indexing, searching, and compression abilities, it has become an important tool in full-text indexing. In its simplest form, it is a well-balanced binary tree with nodes that stores bitmaps. The alphabet size σ represents the height of the tree and the length of the sequence n represents the width of the tree, to represent S, you will need $n \lg \sigma + o(n \lg \sigma) + O(\sigma \lg n)$ bits. An increase in the value of σ affects the space utilization of wavelet trees. A pointer-less wavelet tree saves space by concatenating all of the bitmaps level by level removing $O(\sigma \lg n)$ bits from the space. Although it maintains the same complexity as the standard wavelet trees, in practice it is slower.

Thomas, Enno, and Simon introduced a new data structure called bidirectional wavelet index in April 2012, which consists of the wavelet tree of the Burrows-Wheeler transformed string of S (that supports backward search) and the wavelet tree of the Burrows-Wheeler transformed string of S^{rev} (that supports forward search) [19]. This new data structure allows for bidirectional search while using less space. In comparison to affix arrays, the bidirectional wavelet index reduces the space required by a factor of 21, allowing bidirectional search in a very long string. Burrows Wheeler transform (BWT) is an algorithm based on a reversible transformation. It was introduced in 1994 by Burrows and Wheeler, originally it was invented for compression but it is now being used for indexing [12]. The goal is to use a reversible transformation to create a permutation of the characters of an input string S, which is defined over an alphabet Σ so that the string can be compressed more easily. The transformation causes the same characters be grouped

(BWT sorts the characters by their right context) increasing the chances of finding a character close to another instance of the same character. In 2015 Francisco, Gonzalo and Alberto presented the wavelet matrix 2]. It is an alternate version of the balanced pointer-less wavelet tree that reorders the nodes in each level while preserving all the wavelet tree's functionality and the traversals required to complete the operations are simplified and speed up.

1.2 Problem Statement

The wavelet tree uses a tree structure for its representation with a height of lg σ (σ is the number of the alphabet), the wavelet tree space utilization is affected by a large value of σ. Though the pointer-less version concatenates the bitmaps level-wise while retaining the time complexity it is still slower in practice.

1.3 Aim and Objectives

In this research, we use the wavelet matrix to create the Bidirectional wavelet index which will consist of the wavelet matrix of the Burrows Wheeler Transformed of string S (allowing backward search) and the wavelet matrix of the Burrows-Wheeler transformed string of S^{rev} (allowing forward search). Using the wavelet matrix will simplify and speed up the operations.

Following are the objectives to attain the aim:

i. Study and use wavelet Matrix to construct the Bidirectional wavelet index.

ii. Construct a Bidirectional wavelet index that answers the queries faster.

iii. Compare wavelet tree and wavelet matrix.

1.4 Organization of the Dissertation

This dissertation is broken into five chapters. Chapter one introduces the research background, aim, and objective. Chapter two presents a literature review of previous works related to this research work. Chapter three present the materials and methods used for this research. The Chapter is all about the experimental results and in Chapter there is the summary and conclusion of the work.

CHAPTER 2
Literature Review

2.1 Introduction

This chapter contains the review of the previous work done by other researchers. It focuses on the knowledge and ideas established on this topic.

2.2 Background

In 1995, Stoye presented his diploma thesis on affix trees [below17], and Maaß demonstrated that affix trees can be constructed on-line in linear time [11], paving the way for data structures that facilitate bidirectional search in string research. The affix tree of a string S is made up of both the suffix tree of S (which supports forward search) and the suffix tree of the reverse string S^{rev} (which supports backward search). It requires around 45n bytes, where n is the size of S. Affix arrays have the same functionality as affix trees, according to Strothmann, however they take up 18n–20n bytes (depending on the implementation) [18]. The suffix arrays of S and S^{rev} are combined in an affix array, however it is a complex data structure due to the challenging interplay between the two suffix arrays.

2.3 Basic Concepts

Suffix Array

Most Algorithms employ string matching as a basic stage, to reduce the time it takes to perform exact matching we generally do pre-processing of the text. The time complexity becomes linear after pre-processing. This pre-processing can be done with suffix array.

Suffix Array is a sorted array that contains all the suffixes of a string. It's a widely used text index structure with a wide range of applications. Suffix arrays were introduced in 1990 as an alternative to suffix trees (which is another data structure for pattern matching) by Manber and Myers [20]. When compared to the suffix tree, the suffix array takes up less space.

Given a string S, let's S = ATGTGTGGCATT

The suffix array can be computed by first creating the array of the suffixes of S and then sorting them as shown in Figure 2.*1*.

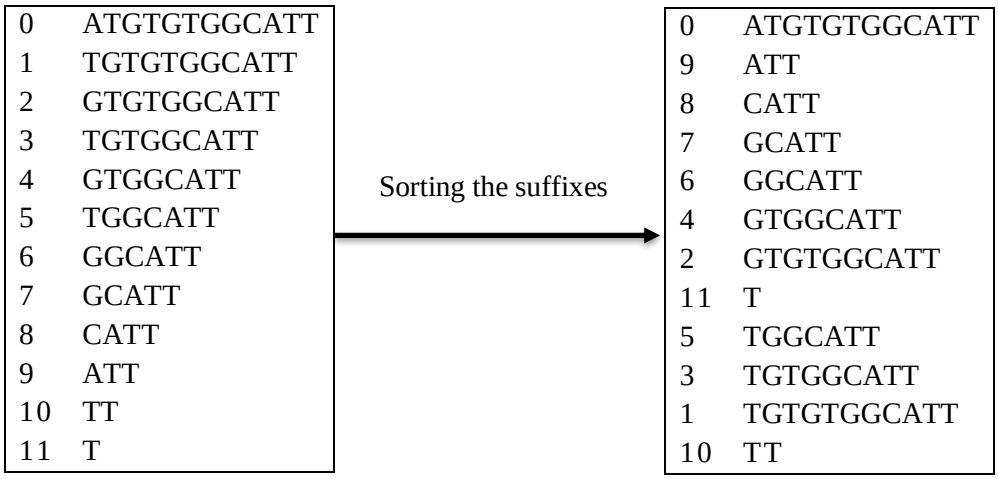


Figure 2.1 Suffix Array of S

BWT

BWT (Burrows Wheeler Transform) is a compression algorithm that rearranges the characters of a string by bringing similar characters to close. It's a text transformation useful for compression and searching. Michael Burrows and David Wheeler created the Burrows Wheeler Transform technique for lossless text compression in 1994. It is based on a transformation discovered by Wheeler in 1983 and published in their work "A Block-sorting Lossless Data Compression Algorithm." [12].

Burrows Wheeler Transform is an algorithm based on a reversible transformation of a string [6]; the transformed string denoted by BWT is simply a permutation of the original string. Given a string S, let $ be a character not in S that is lexicographically less than all the characters of S, the basic idea is to create a (conceptual) matrix whose rows are all the lexicographically ordered cyclic shifts of S and then return the matrix's last column, which tends to group like characters together. The advantage of doing this is that once the same characters have been grouped together, it gave an ordering which makes the string more suitable for compression, searching and other algorithms. For example,

Let S = ATGTGTGGCATT

Append $ to S, S = ATGTGTGGCATT$

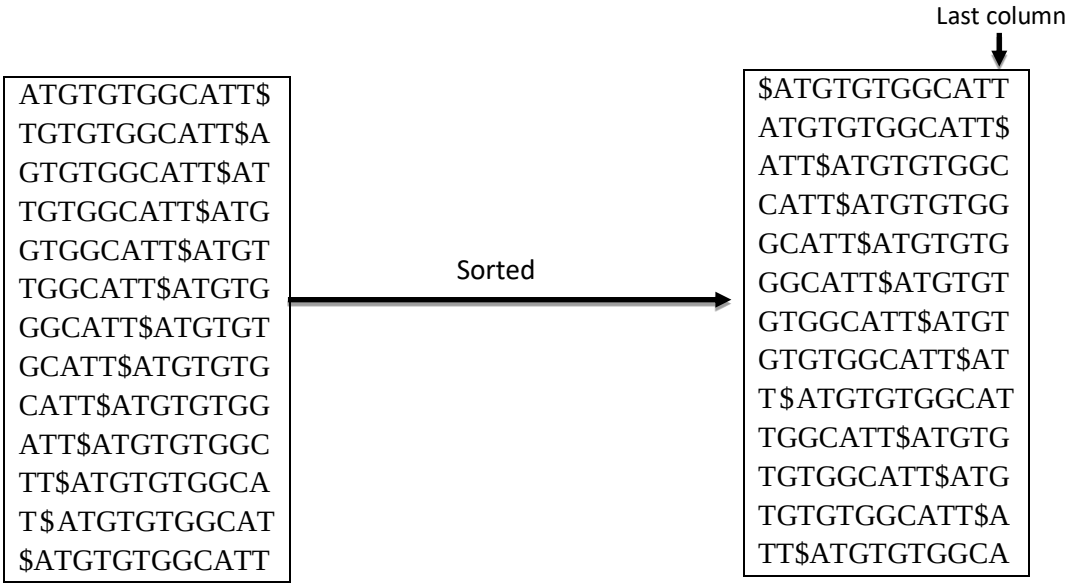


Figure 2.2 Construction of the BWT for S

The last column is what we output as BWT, from Figure 2.*2* the BWT of S = ATGTGTGGCATT$ is BWT = T$CGGTTTTGGAA

BWT with Suffix array

We can construct the BWT from the suffix array:

Let $\sum$ be an ordered alphabet whose smallest element is $ and is in ascending order, Let σ be the size of $\sum$, S is a string of size n over $\sum$ with $ at the end, For $0 \leq i < n$ S[i] denotes the character at position i in S, S[i...j] denotes the substring of S that begins with the character at position i and ends with the character at position j, S_i denotes the i^{th} suffix S[i…n-1] of S

Given the suffix array SA of S the burrows wheeler transform BWT[0...n-1] of S is defined by:

BWT[i] = S[SA[i]-1] for all i with $SA[i] \neq 0$ and BWT[i] = $ otherwise

Example

S = ATGTGTGGCATT$

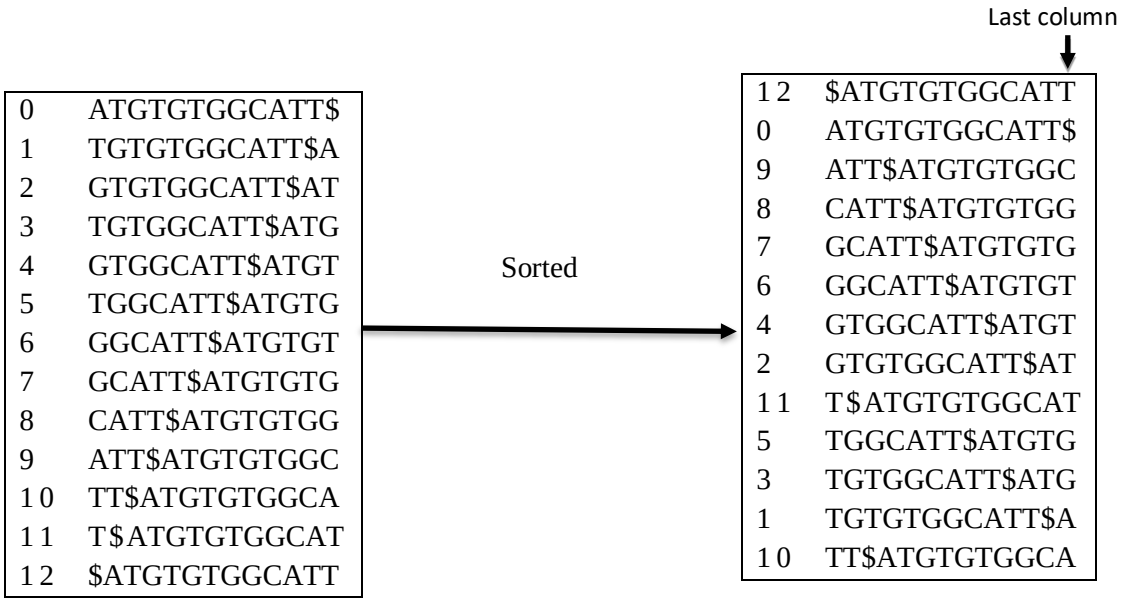


Figure 2.3 BWT with Suffix Array 1

From the Figure 2.*3* SA of S is given as

12	0	9	8	7	6	4	2	11	5	3	1	10

We only store the index of the first character of each suffix. Now BWT can be constructed from the SA as follows:

BWT[0] = S[SA[0] – 1] = S[11] = T

BWT[1] = S[SA[1] – 1] but SA[1] = 0 by definition if SA[i] = 0 BWT[i] = $ therefore

BWT[1] = $

BWT[2] = S[SA[2] – 1] = S[8] = C

.
.
.

BWT[12] = S[SA[12] – 1] = S[9] = A

S can be reconstructed with the BWT of S

LF Mapping

LF Mapping (Last to Front Mapping) is a function mapping from the last column to the first column of BWT. The reversibility of the BWT is obtained via the LF mapping property.

We need to identify the rank of each occurrence of a character in a string because there could be numerous instances of the same character. The LF mapping property states that, in the BWT

matrix the i^{th} occurrence of a character c in the last column corresponds to the i^{th} occurrence of c in the first column. To demonstrate this, let's label each character with its rank, for example the first occurrence of A is labeled as A_1, the second occurrence of A is labeled as A_2, and so on, as depicted in Figure 2.*4*.

Let S = ATGTGTGGCATT$ with rank S = $A_1T_1G_1T_2G_2T_3G_3G_4C_1A_2T_4T_5\$_1$

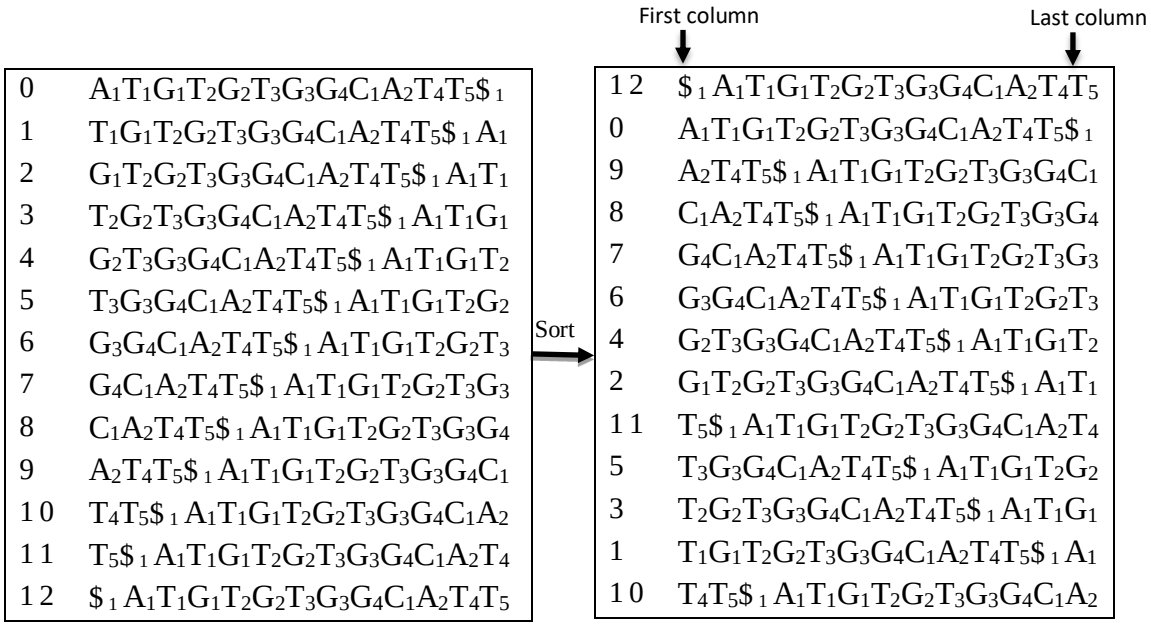


Figure 2.4 Constructing the BWT of S = ATGTGTGGCATT$, each character labeled with its rank

The original string can be reconstructed from right to left, from last to the first character. The first character of the BWT is, by definition, the last character of the string ($ excluded).

We need to know the rank of c (how many characters appear before c in the BWT) in order to determine which character occurs to the left of c (the character before c), according to the mapping property, see Figure 2.*5*.

Because the first character in the BWT indicates where the original string ends, thus we know that the original string ends with T$ from the BWT of the previous example. Because T is the first character of the BWT, it corresponds to the first rotation that starts with T which is row 9, the last character of row 9 is T, therefore the character before T in S is T, and so on, as depicted in Figure 2.*5*.

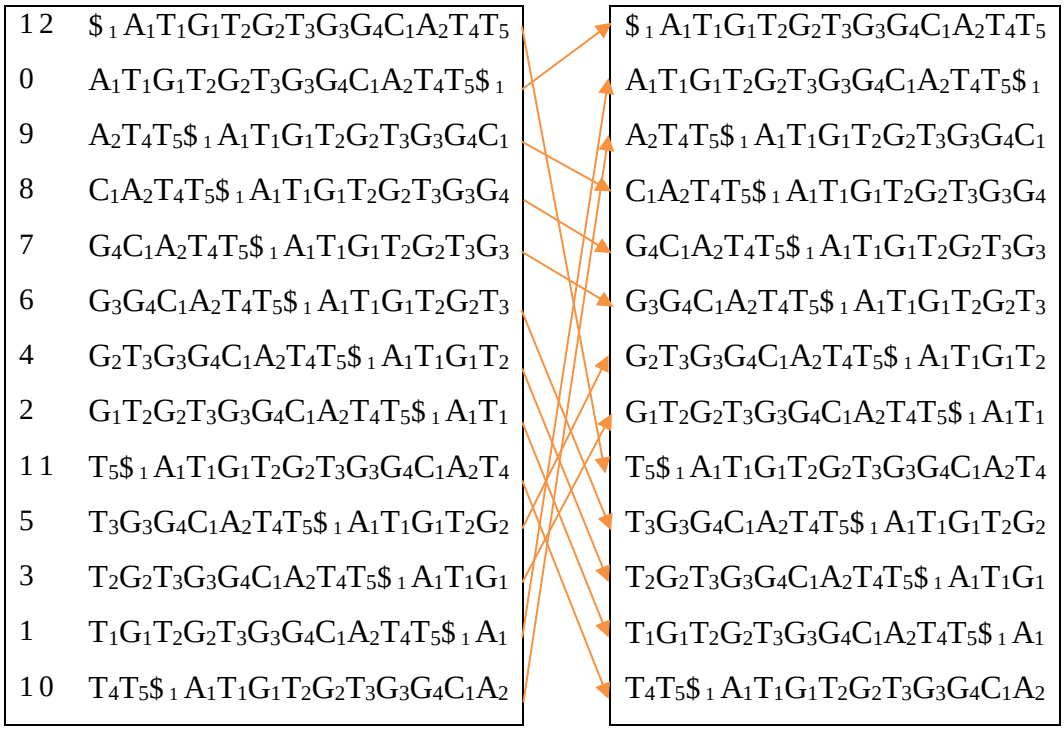


Figure 2.5 LF-Mapping for S = ATGTGTGGCATT$

LF mapping is also defined as:

LF[i] = C[c] + Occ[c,i] - 1 where,

c = BWT [i]

C[c] is the overall number (of occurrences) of characters in S which are strictly smaller than c, and

Occ(c, i) is the number of occurrences of the character c in BWT [0..i]

For example, let BWT = T$CGGTTTTGGAA

LF[0] = C[T] + occ[T,0] = 8 + 1 - 1 = 8

LF[1] = C[$] + occ[$,1] = 0 + 1 - 1 = 0

.

.

.

LF[12] = C[A] + occ[A,12] = 1 + 2 - 1 = 2

FM Index

FM Index (Full-text Index in Minute-space) is a compressed form of the suffix array based on BWT. FM-index was first proposed as a series of compressed full-text indexes to mimic classical suffix arrays by Ferragina and Manzini in 2000 [14, 16, 1]. Because it compresses the input string while permitting quick substring queries, they describe it as an opportunistic data

structure. FM-indexes are comparable to suffix arrays in terms of searching capabilities, however, they take up less space.

This data structure allows you to find all the occurrences of a pattern P in a large string S without having to go through the entire string S. String and pattern are character sequences that span an alphabet Σ of size σ. Sometimes we don't just want to know the value only, that is how many times the pattern appears in the string but also the text positions of those occurrences. FM-index can be used to count and locate the occurrences of the pattern in the original string by only locating at some portion of the compressed (transformed) string. It only takes few milliseconds to perform the operations.

The FM-index creates a compressed suffix array by combining the Burrows Wheeler compression technique with the suffix array data structure. The resulting index is a Burrows Wheeler Transform-based compressed full-text substring index that has some similarities to the suffix array. It's commonly referred to as compressed suffix array if it's based on a compressed ψ-function [7].

The FM-index supports two basic operations:

- count – determines the number of occurrences of a pattern P in a String S. the running time of this operation does not depend on the length of S
- locate – determine all the positions of a pattern P in a string S

FM-Index Querying

Given the BWT of a string S, we wish to count the number of occurrences of a pattern P. We could execute a binary search on the suffix array if we had access to the conceptual matrix. We need a different strategy since we only have access to the First and Last columns of the suffix array as shown in Figure 2.6.

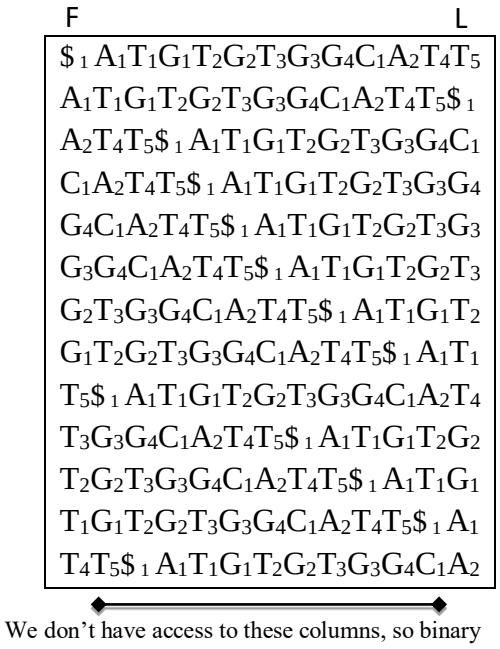


Figure 2.6 Querying FM-Index 1

It is possible to search for a pattern P in S one character at a time, starting with the last character of the pattern and moving backwards (right to left). To begin we set the initial interval to values that corresponds to the range in SA that points to all the occurrences of the last character of P. Each step extends the preceding character in P to the searched string, updates the interval to corresponds to all positions in S that matched the increasing suffix and applies the LF mapping to both sides of the interval. This continues until each character in P has been processed or we fail to find the next character (meaning P does not incur in S). The length of the last interval corresponds to the number the occurrences of P in S. Let's S = ATGTGTGGCATT$, BWT = T$CGGTTTTGGAA and P = TG

To find P, look for range of rows with P as the prefix, we will do this first for the last character of P and then then extend with the preceding character until we exhaust all characters of P or failed to find the next character. Last character of P is 'G', with First column's simple structure it's easy to find all the rows starting with 'G', see Figure 2.7.

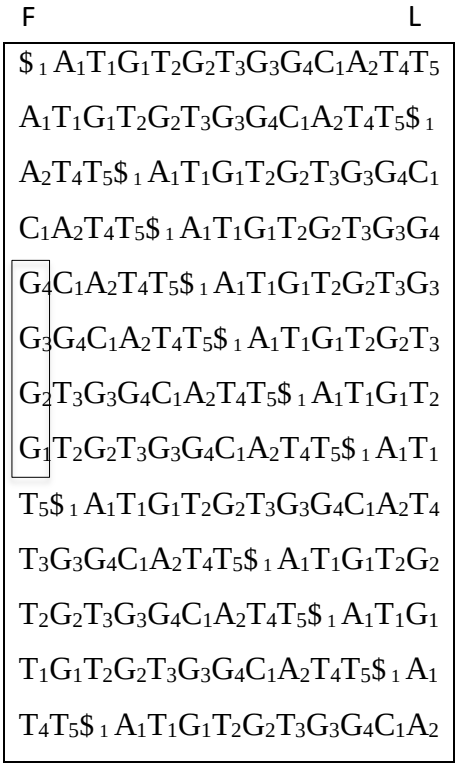


Figure 2.7 Querying FM-Index, looking for rows starting with G

We have rows starting with ‘G’ as shown in Figure 2.7, we now need to find rows starting with TG, we look at the Last column of the rows starting ‘G’ that are equal to ‘T’, see Figure 2.8.

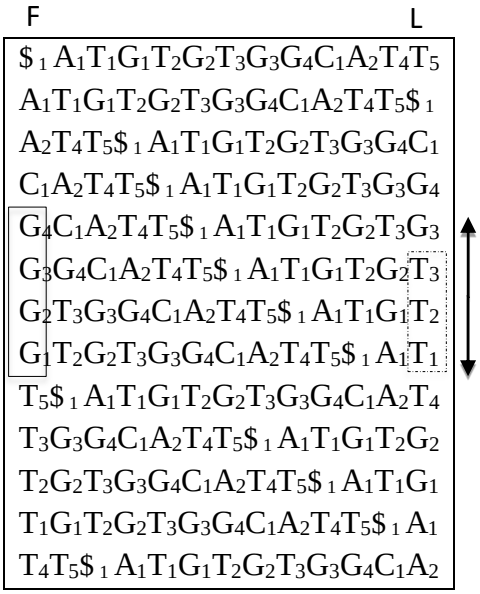


Figure 2.8 Querying FM-Index, rows beginning with TG

Now we can use the LF-Mapping property to get the new interval:

- LF[5] = C[T] + occ[T, 5] - 1 = 8 + 2 - 1 = 9
- LF[6] = C[T] + occ[T, 6] - 1 = 8 + 3 - 1 = 10
- LF[7] = C[T] + occ[T, 7] - 1 = 8 + 4 - 1 = 11

The new interval is [9,11] since we have exhausted all the characters of P we know have rows with prefix P. we have the same interval we would have gotten from querying the suffix array. The length of the last interval corresponds to the number of occurrences of P, length of [9,11] is 3, therefore the number of occurrences of 'TG' in ATGTGTGGCATT is 3. With suffix array, we can locate the position of each match in S as depicted in Figure 2.*9*, which cannot be easily done with FM-Index.

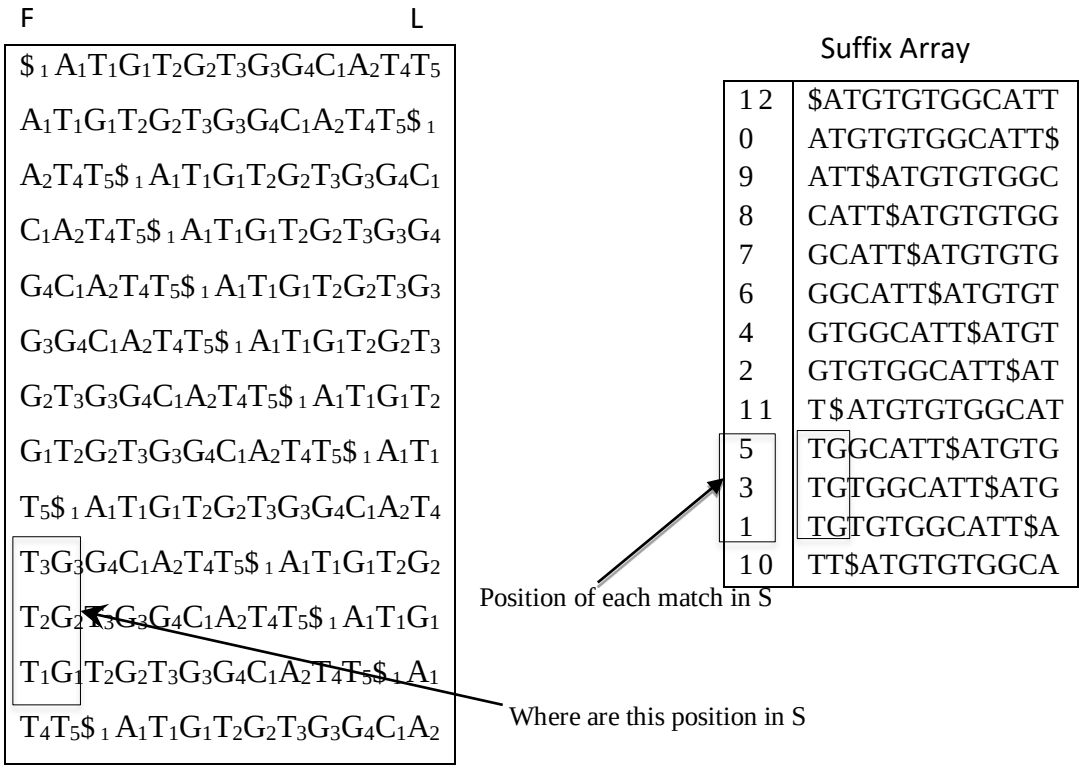


Figure 2.9 Querying FM-Index: Locating position of the matches

Unlike in the suffix array, we don't immediately know where the position of the matches is in S. We need a way to locate the position where the matches occur in S. We could easily checkup the offsets if the suffix array was part of the index as shown in the Figure 2.*10*, but suffix array

requires n integers (where n is the size of the string including the EOF character). Another way is to store not all but some entries of the suffix array as shown in Figure Figure 2.*11*. In this method looking up for the offsets might fail because some of the rows don't have entries. We discard the entries of the suffix array that don't entries and use LF-Mapping to resolve the offsets.

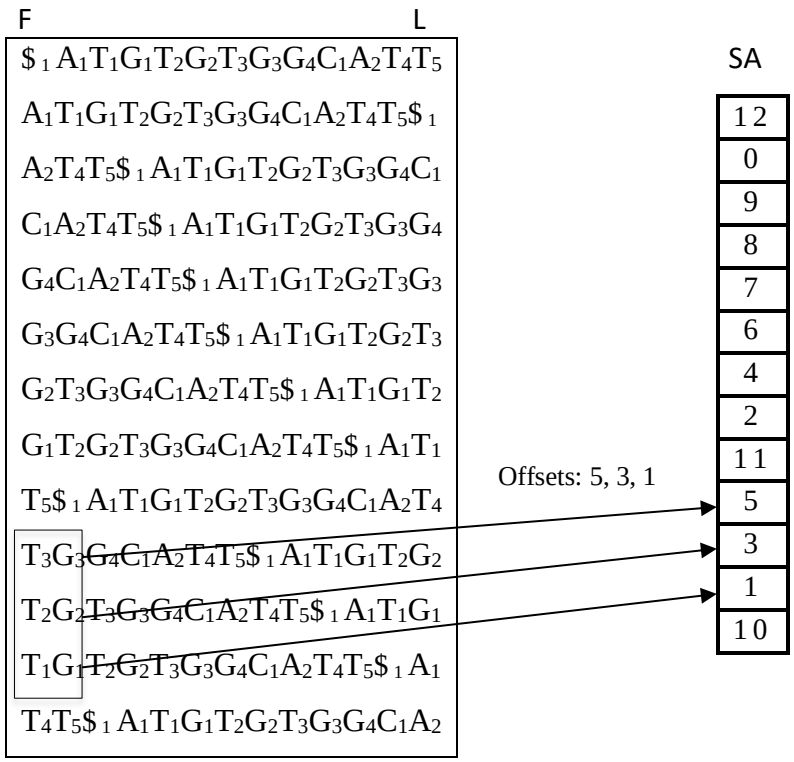


Figure 2.10 Querying FM-Index: Resolving offsets with SA

In the Figure 2.*11* the lookup for row 9 fails we discard the entry of the suffix array and then apply the LF-Mapping property to resolve the offset.

LF[10] = C[G] + occ[G,10] - 1 = 4 + 4 - 1 = 7

With LF-Mapping we found out that the character 'G' at the end of row 10 (row 10's last column) corresponds to the character 'G' at the beginning of row 7 (row 7's first column), see Figure 2.*12*. Row 7 has a suffix array value of 2, so therefore row 10 has suffix array value as follows:

Row 10 suffix array value = 2 (row 7's suffix array value) + 1 (number of steps to row 7) = 3

Resolving offsets takes O(1) time if the stored suffix array values are O(1) positions apart in S.

Now to find the position where the occurrences are in S, see Figure 2.*11*, we can store some of the suffix array values, let's call this suffix array sample (SA sample). With the suffix array sample, we can find the position of the occurrences in S in O(1) time.

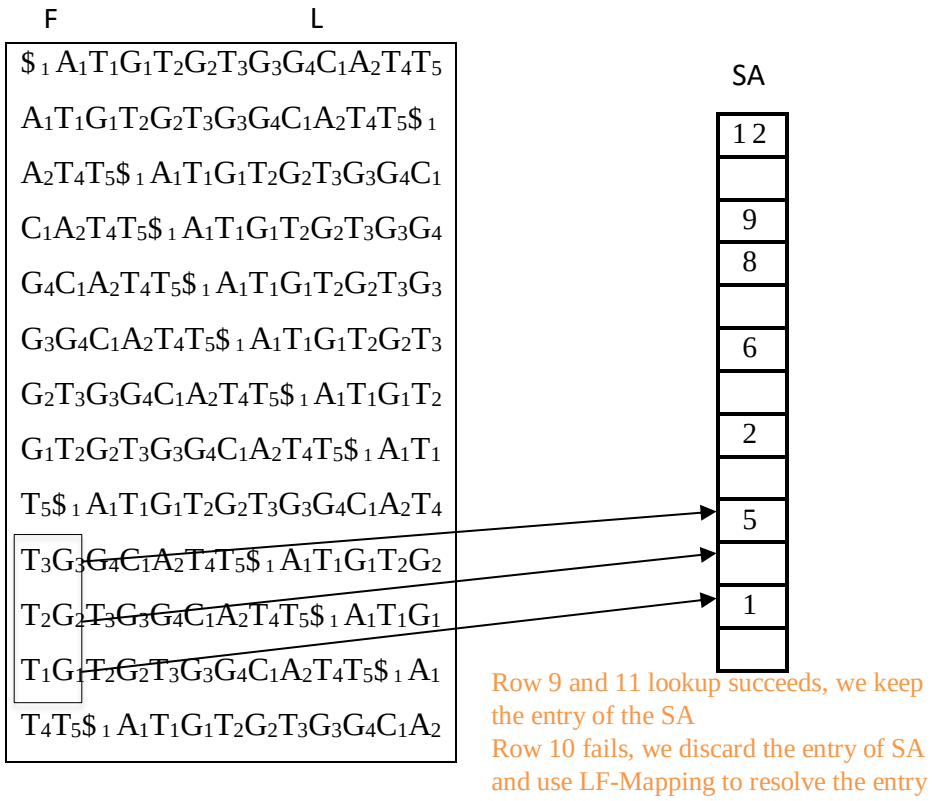


Figure 2.11 Querying FM-Index: Resolving offsets with SA and LF-Mapping

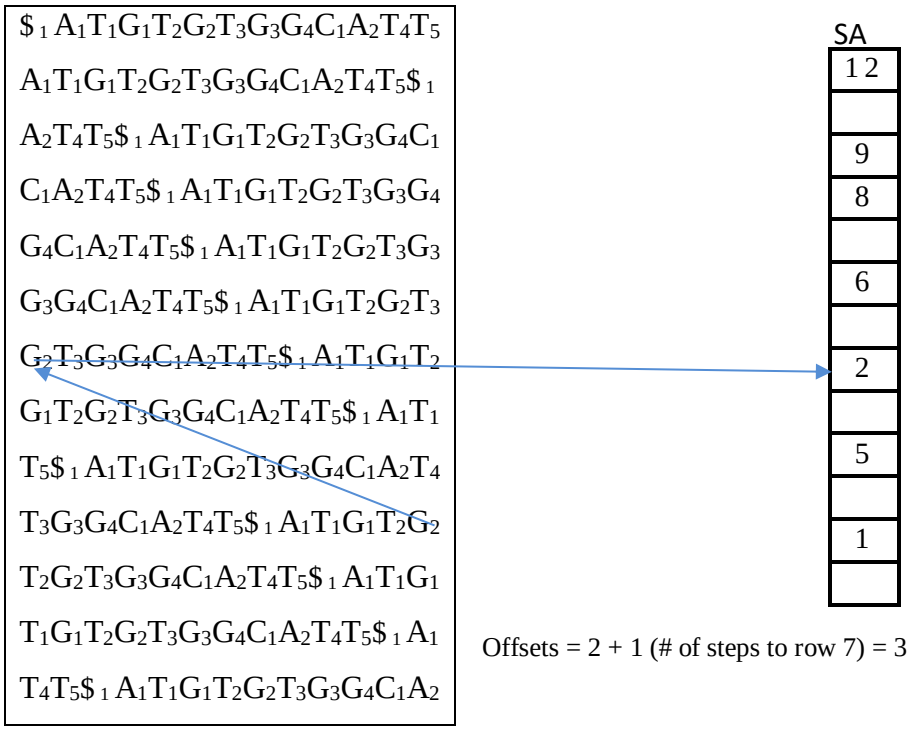


Figure 2.12 Querying FM-Index: Resolving offsets with SA and LF-Mapping

Components of FM-Index:

- First column
- Last column
- SA sample
- Checkpoints

Wavelet Tree

A wavelet tree is a versatile data structure that divides a string into two parts in a recursive manner until homogenous data is obtained. it was introduced in 2003 by Grossi, Gupta, and Vitter as a space-efficient data structure for representing a sequence and answering queries on it [9]. Due to its compression, indexing, and searching capabilities, it has become an important tool in full-text indexing and data compression. In its most basic form, the wavelet is "discovered" within the input signal at various starting points and stretches.

A wavelet of the sequence S[1,..,n] over the alphabet Σ[1..σ] is a balanced binary search tree. It supports the following functionality:

- access(S, i) that returns the character at the position ‘i’ in S (S[i])

- $rank_a(S, i)$ that returns number of the occurrence of 'a' in S from the starting position to 'i' (occurrences of 'a' in S[1,i])
- $select_a(S, i)$ that returns the position of the i^{th} occurrence of 'a' in S

Standard Wavelet Tree Algorithms

Access Query

acc(v; i)
    if $w_v - \alpha_v = 1$ then
        return α_v
    end if
    if $B_v[i] = 0$ then
        $i \leftarrow rank_0(B_v, i)$
        return $acc(v_l, i)$
    else
        $i \leftarrow rank_1(B_v; i)$
        return $acc(v_r; i)$
    end if [1][2]

To access the character at position i in S, we begin from the root, if the if the bit value at position i in the current level bitvector is 0, it means that S[i] belongs to the left child of the root node and S[i] belongs to the right child of the root node otherwise. In the former case, position of $S_v[i_v]$ in S_{vl} is $i_{vl} = rank_0(B_v; i_v)$, whereas in the latter case, the position in S_{vr} is $i_{vr} = rank_1(B_v; i_v)$. we repeat this process extracting $S_v[i_v]$ frome nodes until we reach a leaf representing the character interval [a, a], where we can finally report S[i] = a.

Rank Query

rnk(v, a, i)
    if $w_v - \alpha_v = 1$ then
        return i
    end if
    if $a < 2^{\lceil lg\,(w_v - \alpha_v)\rceil - 1}$ then
        $i \leftarrow rank_0(B_v, i)$
        return $rnk(v_l, a, i)$
    else
        $i \leftarrow rank_1(B_v, i)$
        return $rnk(v_r, a, i)$

end if [1][2]

The procedure of computing $rank_a(S, i)$ is similar to that of computing access. The only difference is that we do not descend based on whether $B_v[i]$ equals 0 or 1, but rather based on the bits of a $\epsilon [0, \sigma)$: the highest bit of a indicates whether we go left or right, and the lower bits are used in the next levels. When move to a child let's say u of v, the number of times the current bit of a appears in $B_v[1, i_v]$ is computed as $i_u = rank_{0/1}=(B_v, i_v)$. The answer to $rank_a(S, i)$ is i_u when we reach the leaf u handling the range [a, a].

Select Query

```
sel(v, a, j)
    if w_v - α_v  = 1 then
        return j
    end if
    if a < 2^⌈lg (w_v −α_v)⌉−1 then
        j ← sel(v_l, a, j)
        return select_0(B_v, j)
    else
        j ← sel(v_r, a, j)
        return select_1(B_v, j)
    end if [1][2]
```

For $select_a(S; j)$, we proceed upwards. We begin with the leaf u handling the character range [a, a]. We would like to keep track of the position of $S_u[j_u]$, $j_u = j$ as it approaches the root. If u is the left child of its parent v, then the corresponding position at the parent is $S_v[j_v]$, where $j_v = select_0(B_v; j_u)$. otherwise, the corresponding position is $j_v = select_1(B_v; j_u)$. When we finally reach the root v, the answer to the query is j_v.

Constructing The Wavelet Tree of The Burrows Wheeler Transform

In the wavelet tree, each node of the tree corresponds to a string $BWT^{[l..r]}$, where [l..r] is an alphabet interval. If an interval [l..r] is a subinterval of [1..σ] where $\sigma = |\Sigma|$, we call it an alphabet interval. By eliminating all the characters in BWT that do not corresponds to the sub alphabet Σ[l..r] of Σ[0..σ], the string $BWT^{[l..r]}$ is obtained, see Figure 2.13. For example, let's consider the BWT = T$CGGTTTTGGAA and an interval [0...2]. The string $BWT^{[0..2]}$ is obtained from the BWT by eliminating the characters T and G ending up with, $BWT^{[0..2]}$ = $CAA.

BWT = $BWT^{[0..\sigma]}$ is the root of the tree. If l = r then the current node have no children, otherwise, it contains two children: the string $BWT^{[l..m]}$ is the left child and its right child is the string $BWT^{[m+1..r]}$, with m = $\lfloor \frac{l+r}{2} \rfloor$. The node holds a bit vector $B^{[l..r]}$ of size r − l + 1 whose i-th entry is 0 if the i-th character in $BWT^{[l..r]}$ belongs to the left subtree and 1 if it belongs to the right subtree. The leaf's of the tree are those nodes that have an alphabet interval of size 1. We do not need to keep the bit vector of a leaf, since it contains only 0. Furthermore, each bit vector in the tree is pre-processed to allow $rank_0(B, i)$ and $rank_1(B, i)$ queries to be answered in constant time, where $rank_b(B, i)$ is the number of occurrences of bit b in B[0..i]. The wavelet tree takes n log bits plus o(n log) bits of space for the data structures that permit rank-queries in constant time [5,3, 8], because only the bit vectors are stored.

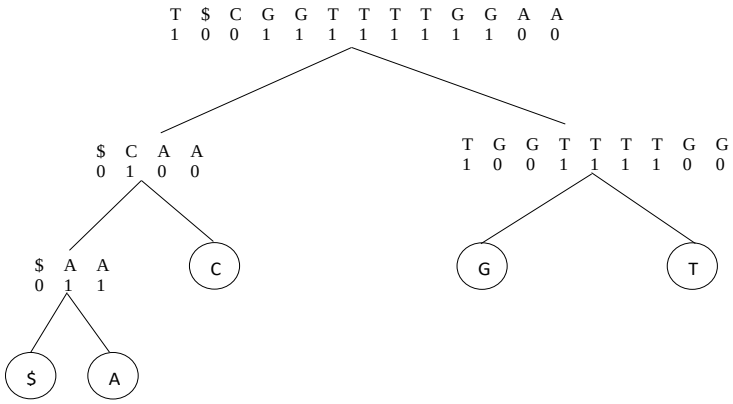


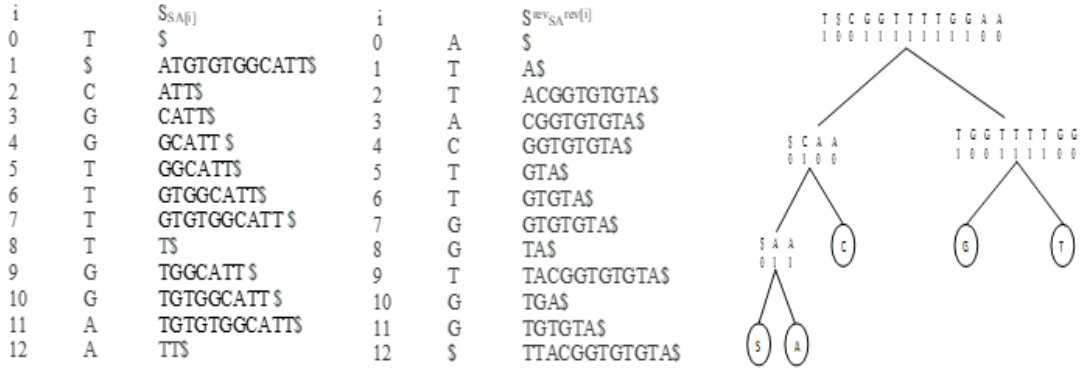
Figure 2.13 Wavelet Tree for the BWT of S = ATGTGTGGCATT$

2.4 RELATED WORK

In April 2012 Thomas, Enno, and Simon proposed a new data structure they called as the bidirectional wavelet index [19], see Figure 2.*14*, which consists of,

- The wavelet tree of the Burrows-Wheeler Transform of S (that supports backward search) and

- The wavelet tree of the Burrows-Wheeler Transform of S^{rev} (that supports forward search).



i		$S_{SA[i]}$
0	T	$
1	$	ATGTGTGGCATT$
2	C	ATT$
3	G	CATT$
4	G	GCATT $
5	T	GGCATT$
6	T	GTGGCATT$
7	T	GTGTGGCATT $
8	T	T$
9	G	TGGCATT $
10	G	TGTGGCATT $
11	A	TGTGTGGCATT$
12	A	TT$

i		$S^{rev}_{SA^{rev}[i]}$
0	A	$
1	T	A$
2	T	ACGGTGTGTA$
3	A	CGGTGTGTA$
4	C	GGTGTGTA$
5	T	GTA$
6	T	GTGTA$
7	G	GTGTGTA$
8	G	TA$
9	T	TACGGTGTGTA$
10	G	TGA$
11	G	TGTGTA$
12	$	TTACGGTGTGTA$

Figure 2.14 Bidirectional wavelet index of S = ATGTGTGGCATT$, wavelet tree of the BWT = T$CGGTTTTGGAA of S

Algorithm 2.1

Given a character c and an alphabet interval [i..j], backwardSearch(c, [i..j]) returns the sub-interval if it exists, and none otherwise

backwardSearch(c, [i..j])

i ← C[c] + Occ(c, i)

j ← C[c] + Occ(c, j) - 1

if i ≤ j then return [i..j]

else return [20][19]

Backward Search

Ferragina and Manzini demonstrated that in the suffix array SA of string S, a pattern can be searched character by character backwards without having to store SA [F]. For example, in Figure 2.14 suffix array, the TG-interval is the interval [9...11]. The pattern TG can be found by searching backwards in the string S = ATGTGTGGCATT$. By definition, backward search for the last character of the pattern that begins with the ε-interval [0...n], where ε represents the empty string. The pseudo-code for one backward search step is shown in Algorithm 1, c = BWT [i], C[c] is the overall occurrences of characters in S which are strictly smaller than c, and Occ(c, i) is the number of occurrences of the character c in BWT[0..i]. In the previous example, backwardSearch(G, [0...12]) will return the G-interval [4...7] because,

- C[G] + Occ(G, 0) = 4 + 0 = 4

- C[G] + Occ(G,12) = 4 + 4 - 1 = 7

In the next step, backwardSearch(T, [4...7]) will return the TG-interval [9..11] because,

- C[T]+Occ(T, 4) = 8 + 1 = 9
- C[T] + Occ(T, 7) = 8 + 4 - 1 = 11

To answer the query Occ(G, 12) we need to traverse through the wavelet tree of the BWT of S. Figure 2.11 shows the wavelet tree of the string BWT = T$CGGTTTTGGAA. The occurrences of G correspond to ones in the bit vector at the root, and they go to the right child, because G belongs to the second half of the ordered alphabet. The number of occurrences of G in $BWT^{[0..4]}$ = T$CGGTTTTGGAA up to position 12 equals the occurrences of G in the string $BWT^{[3,4]}$ = TGGTTTTGG up to position $rank_1(B^{[0..4]}, 12)$. We compute $rank_1(B^{[0..4]}, 12) = 9$. Now since G belongs to the first quarter $\Sigma[3,3]$ of Σ, the occurrences of G correspond to ones in the bit vector, and they go to the left child. The number of occurrences of G in $BWT^{[3,4]}$ = TGGTTTTGG up to position 9 is equal to the number of occurrences of G in $BWT^{[3,3]}$ = GGGG up to position $rank_1(B^{[3,4]}, 4) = 4$. In the last step, Since BWT[3...3] consists only of G's, the occurrences of G in $BWT^{[3..3]}$ up to position 4 is 4.

CHAPTER 3
Materials and Methods

3.1 Introduction

In this chapter we explained the wavelet matrix, how it can be constructed and the algorithm used to construct the wavelet matrix. We also compare the wavelet matrix and wavelet matrix.

3.2 Wavelet Matrix

Wavelet matrix is an efficient data structure that represents a sequence S[0..n] over an alphabet of size σ that is similar to the pointer-less version of the wavelet tree [2, 10]. It preserves all the features of the wavelet tree and it is faster in practice.

In wavelet tree each node stores a bitmap but the wavelet matrix maintains a single bitmap per level, all of the zeros move to the left, and all of the ones move to the right and we keep track of the number of zeros in each level [13]. To obtain a bit vector for a level in the wavelet tree, we concatenate the bit vectors of all the nodes on that level from left to right. In the wavelet matrix, we concatenate the nodes in different order: all left children are moved to the left and all right children are moved to the right. We concatenate the bit vectors of all nodes on every level just like the pointer-less wavelet tree. Example for S = ATGTGTGGCATT the wavelet tree and wavelet matrix of the BWT of S is as shown in Figure 3.*1*.

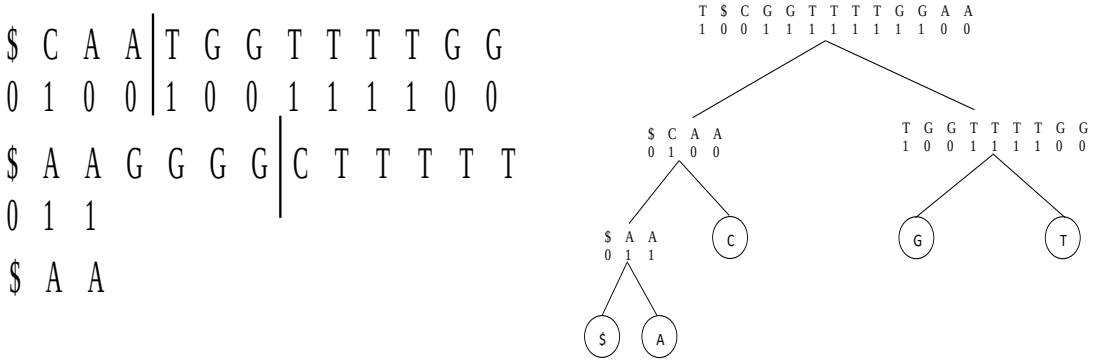


Figure 3.1 Wavelet Matrix and Wavelet Tree for the BWT of S = ATGTGTGGCATT$

Wavelet Matrix Construction

To calculate the bitmap for the first level, we take the whole BWT and then determine the bitmap

value for each character, to do this we consider an integer m = $\lfloor (l + r)/2 \rfloor$ (l is the smallest character and r is the largest) for each character c if c < m then b = 0 (bitmap value), otherwise b = 1. At the next level we split the BWT into two parts and then determine the bitmap value for each character. We do this until r ≤ l

At the first level we keep the bitmap B_0 and then stably sort S by these bitmaps, we keep the next bitmap B_1 then we stably sort S by this bitmap and so on.

Algorithm 3.1

Given S matrix(s) returns the wavelet matrix of S

```
bits(b) :
        If b < = m then
                retrun 0
        else
                return 1

Sort(S ,b) :
        for i in b
                if b[i] = 0 then
                        bitmap[j] ← S[i]
                else
                        append S[i] to bitmap[j]
        return bitmap


matrix(S):
        if S = [] then
                return
        m = ⌈(l + r)/2⌉
        else
                b ← bit(S,m)
                S ← sort(s,b)
```

The wavelet matrix preserves all features of the wavelet tree which include:

- access(l, i) which returns the character at position I in S (S[i])
- rnk(l, a, i, p) returns $rank_a(S, i)$ that is the number of occurrences of the character 'a' in S
- sel(l, a, j, p) returns $select_a(S, j)$ that is the j^{th} occurrence of the character 'a' in S

In this study we are only going to use the rank query to construct the index.

Rank Query

rnk(*l*, a, I, p)

if $w_v - \alpha_v = 1$ then

return i - p

end if

if a < $2^{\lceil lg\,(w_v - \alpha_v)\rceil - 1}$ then

$p \leftarrow rank_0(B_l; p)$

$i \leftarrow rank_0(B_l; i)$

else

$p \leftarrow z_l + rank1(B_l; p)$

$i \leftarrow z_l + rank1(B_l; i)$

end if

return rnk(*l*+1, a, i, p)[2]

To carry out the $rank_a(S, i)$ operation in the wavelet matrix, we must to keep track of the position i, as well as the position before the range, which is initially $p_0 = 0$. At each node v of depth *l*, we go "left" by mapping p_{l+1} to $rank_0(B_l, p_l)$, if a < $2^{\lceil lg\,(w_v - \alpha_v)\rceil - 1}$ and i_{l+1} to $rank_0(B_l, i_l)$. Otherwise, we go "right" by mapping p_{l+1} to $z_l + rank1(B_l, p_l)$ and i_{l+1} to $z_l + rank1(B_l, i_l)$. The solution is $i_l - p_l$ when we reach the leaf.

Constructing the bidirectional wavelet index using wavelet matrix

Let's consider a string S = ATGTGTGGCATT $ to construct the bidirectional wavelet index we need to first determine the burrows wheeler transform of S and the burrows wheeler transform of, S^{rev} = TTACGGTGTGTA$. To get the BWT of S we need to construct a conceptual matrix M which is all the cyclic rotations of S and the sort them in lexicographical order, the last column

of the sorted rotations is the BWT of S, see Figure 3.*2*. We do the same for S^{rev} as shown in Figure 3.*3*.

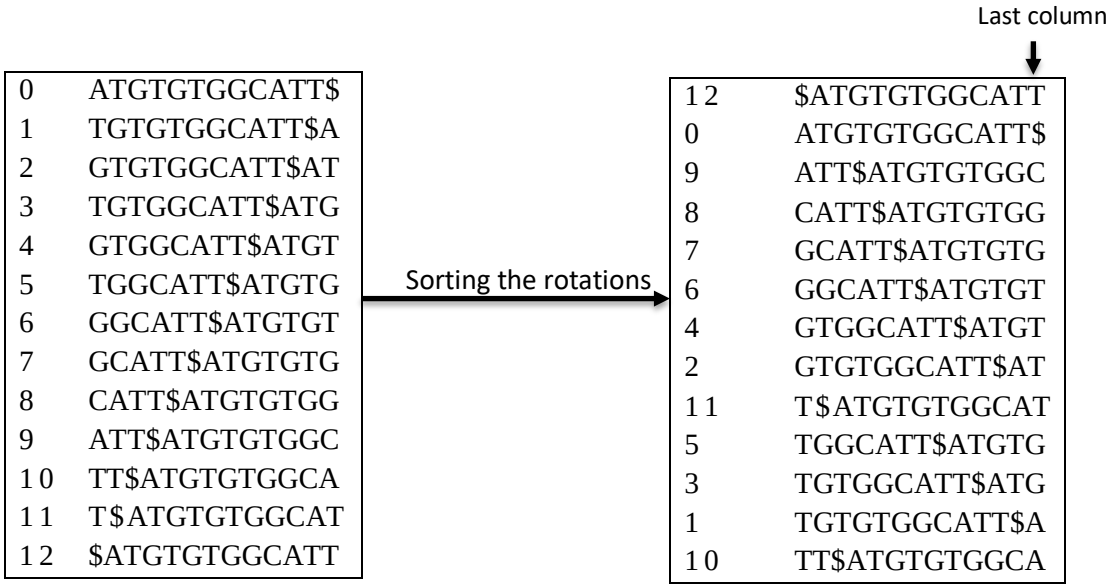


Figure 3.2 Constructing the BWT of S = ATGTGTGGCATT$

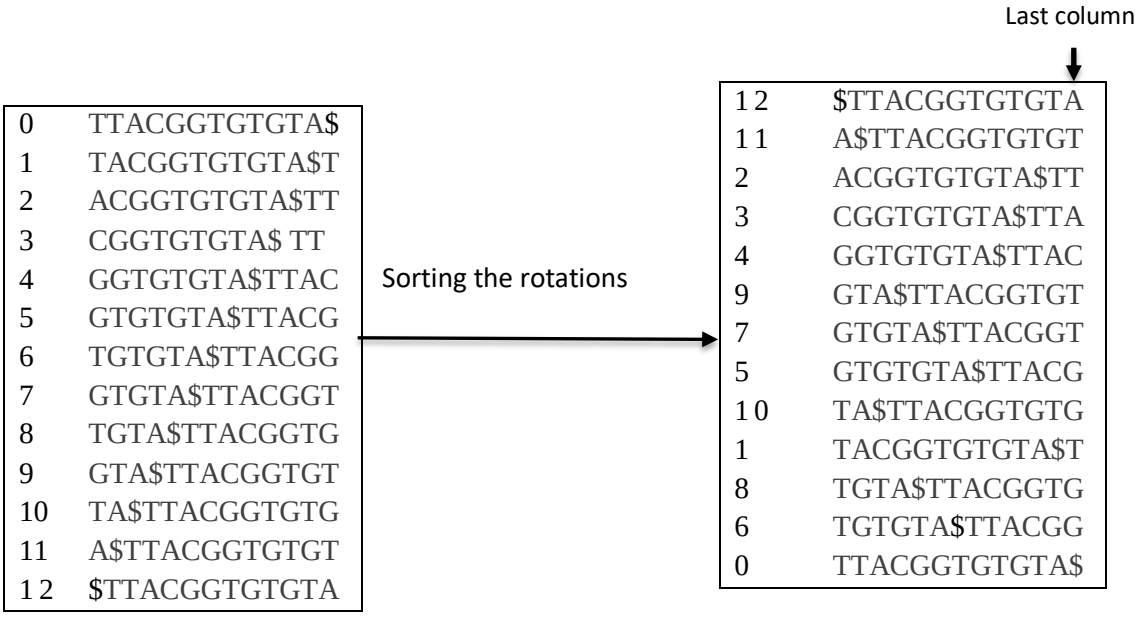


Figure 3.3 Constructing the BWT S^{rev} = TTACGGTGTGTA$

After getting the BWT of S we need construct the wavelet matrix of the BWT of S and S^{rev}. Figure 3.*4* shows the wavelet matrix of S.

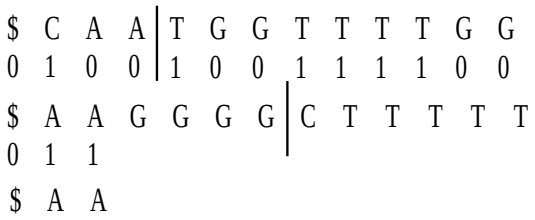


Figure 3.4 wavelet matrix for the BWT of S = ATGTGTGGCATT$

The Bidirectional Wavelet index contains the wavelet matrix of the Burrows Wheeler transform of S and the wavelet matrix of the Burrows Wheeler transform of S^{rev} as shown in Figure 3.5.

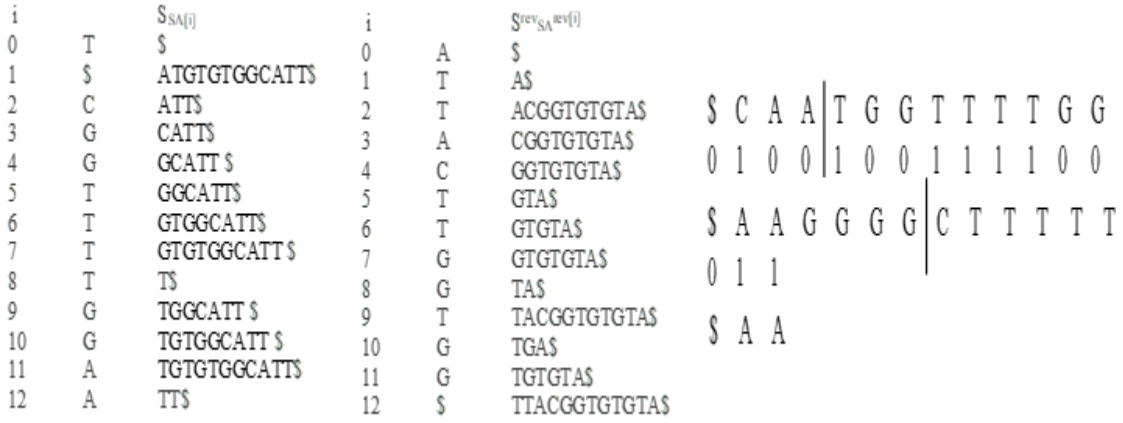


Figure 3.5 Left: Bidirectional wavelet index of S = ATGTGTGGCATT $ and wavelet matrix of the BWT = T$CGGTTTTGGAA of S

Comparison between wavelet tree and wavelet matrix

Wavelet tree has height of $\lceil \lg \sigma \rceil$, it holds n bits per level for the bitmaps. As a result, it stores $n\lceil \lg \sigma \rceil$ bits plus $O(\sigma \lceil \lg n \rceil)$ for storing pointers plus $o(n\lceil \lg \sigma \rceil)$ bits for data structure that support rank query. All the operations are answered in $O(\lg \sigma)$. $O(\sigma \lceil \lg n \rceil)$ becomes dominant in application where the size of the alphabet is large. Makinen and Navarro demonstrated [16] that the bitmaps at each level could be concatenated while still performing the operations. This reduces the space usage of the wavelet tree by removing $O(\sigma \lceil \lg n \rceil)$ bits, while the complexity appears to be the same in theory, in fact, in practice the pointer-less version of the wavelet tree requires three times number of operations carried out in the standard wavelet tree.

To achieve the time complexity of $O(\lg \sigma)$ for the operations both in theory and practice while

preserving the space complexity of the pointer-less wavelet tree, Fransisco and Gonzalo introduce the ‘Wavelet Matrix' [2], a new level-wise organization of the bits. It retains the complexity of the pointer-less wavelet tree and answers all in the operations in $O(\lg \sigma)$ both in theory and practice.

While navigating around the tree, we must keep track of the current node's interval (that is its left and right boundaries) within the relevant level's bit vector, this may be accomplished by using two rank queries on the relevant bit vector. The simpler form of the matrix makes it easy to precompute the left boundary for each level’s right child, which all have been concatenated in each level’s right part bit vector.

rnk(l, a, I, p, e)

if $w_v - \alpha_v = 1$ then

return i

end if

$l \leftarrow rank_0(B_l, p)$

$r \leftarrow rank_0(B_l, e)$

if $a < 2^{\lceil lg\ (w_v - \alpha_v)\rceil - 1}$ then

$z \leftarrow rank_0(Bl, p + i)$

return rnk(l+1, a, z-l, p, p+r-l)

else

$z \leftarrow rank_1(Bl, p + i)$

return rnk(l+1, a, z-(p-l), p+r-l, e)

end if [2]

CHAPTER 4
Experimental Result

4.1 Introduction

In this result we test the performance of the Bidirectional Wavelet Index constructed using the wavelet matrix. We executed some queries and then compare the performance of both the index. We present the result of our findings.

4.2 Data Source

We use human genome data available online. It is available at http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz

4.3 Environment

We use python 3 for the implementation. Following is the system's specification:

- Processor: AMD E1-6015 APU with Radeon (TM) R2 Graphics, 1400 Mhz, 2 Core(s), 2 Logical Processor(s)
- RAM: 4.00 GB
- Disk Drive: Standard disk drives 465 GB
- Adapter: AMD Radeon HD 8200 / R3 Series
- OS Name: Microsoft Windows 10 Pro

Experiments

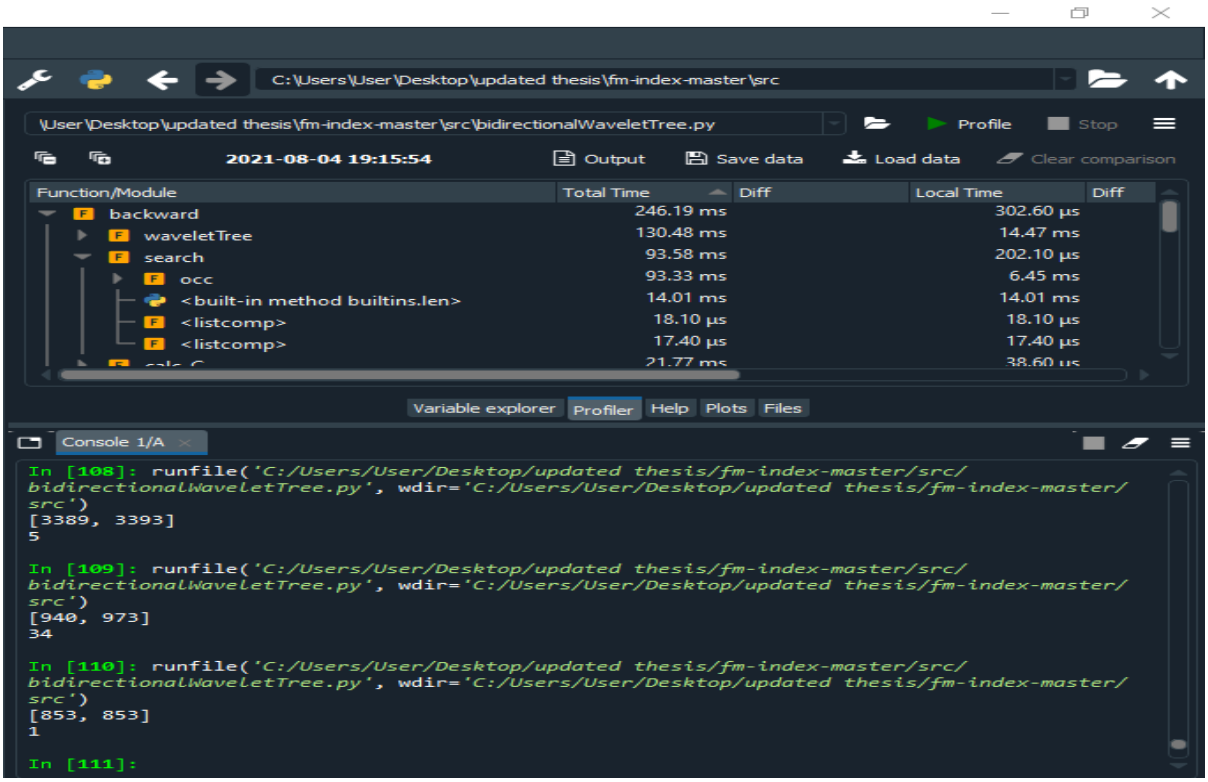
Table 4.*1* shows the running time of the various queries using both the index constructed with wavelet tree and the index constructed with wavelet matrix. The running time is in milliseconds(ms). Index 1 is for the Bidirectional index constructed with the wavelet tree and index 2 is for the Bidirectional index constructed with the wavelet matrix. The second column shows the queries executed and the third column shows the number of matches found. The last row shows the average of both indexes, from this we can see that the queries executed faster with the Bidirectional index constructed with a wavelet matrix than the index constructed with wavelet tree.

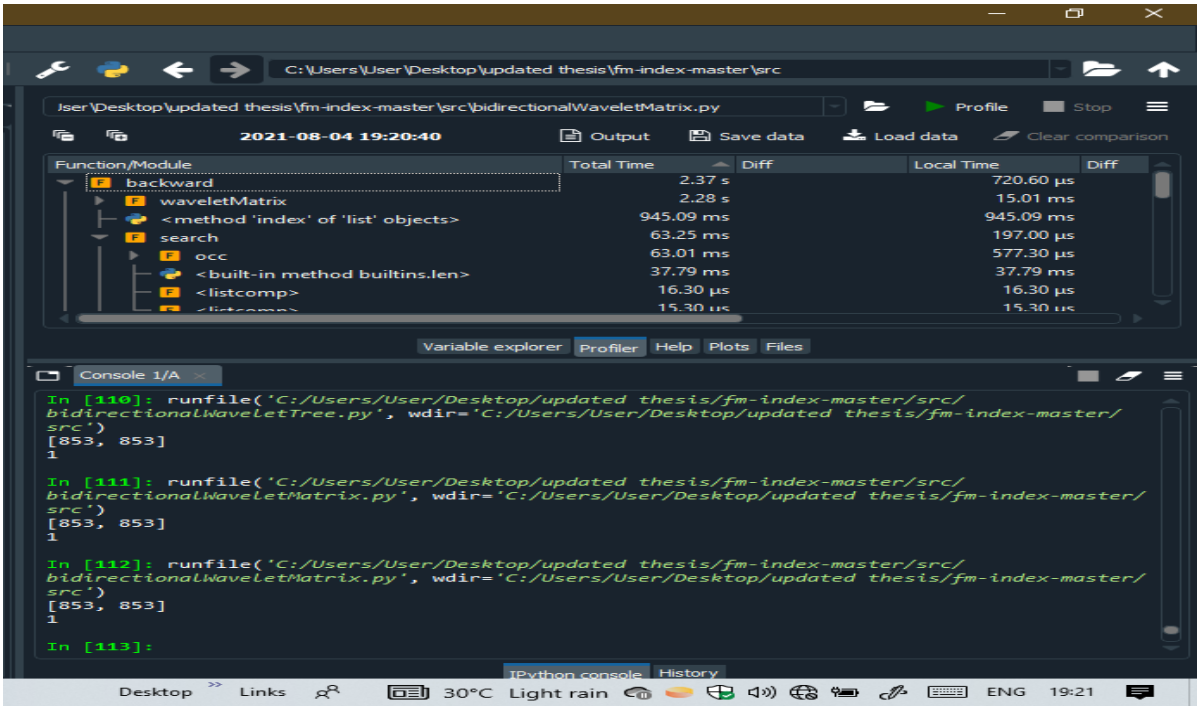
EXPERIMENTS

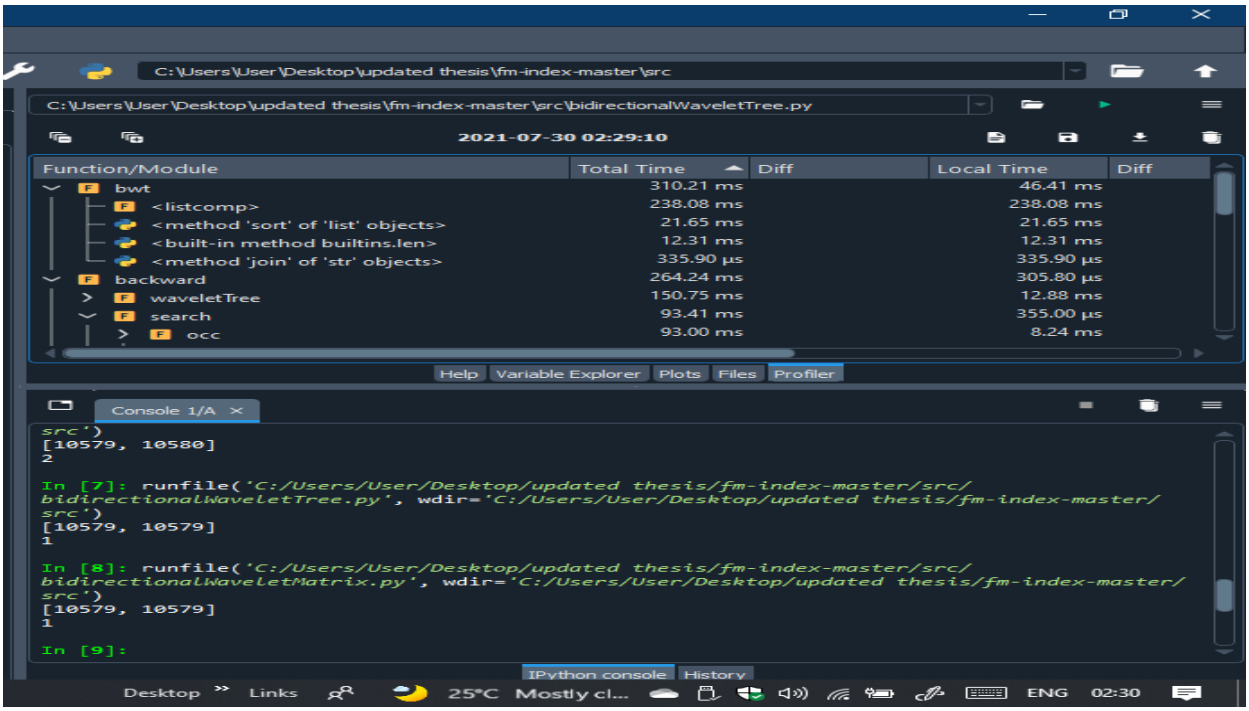
SN	Queries	# Of Matches found	Index 1 (ms)	Index 2 (ms)
1	ATTGT	5	89.00	69.27
2	AATC	33	61.54	38.96
3	TGTC	30	64.25	50.69
4	ATTGG	17	89.42	70.72
5	GTTAA	4	71.52	49.49
6	AAAC	84	45.61	31.76
7	AGGT	37	80.47	58.34
8	AAC	233	38.15	16.60
9	TAAT	24	63.01	32.80
11	GTTAA	4	71.87	49.49
12	AATC	33	65.83	38.96
13	TC	487	27.80	19.27
14	ATGGC	19	84.75	71.37
15	TATG	43	64.81	47.21
16	ATGT	46	77.73	58.04
17	GGAA	130	64.81	47.21
18	GTGG	72	77.64	70.54
19	TGC	128	47.84	36.53
20	TGGC	52	63.22	54.35
22	TTAGTC	1	93.00	86.34
23	AAGTAG	5	93.33	63.01
AVERAGE			68.3619	50.52143

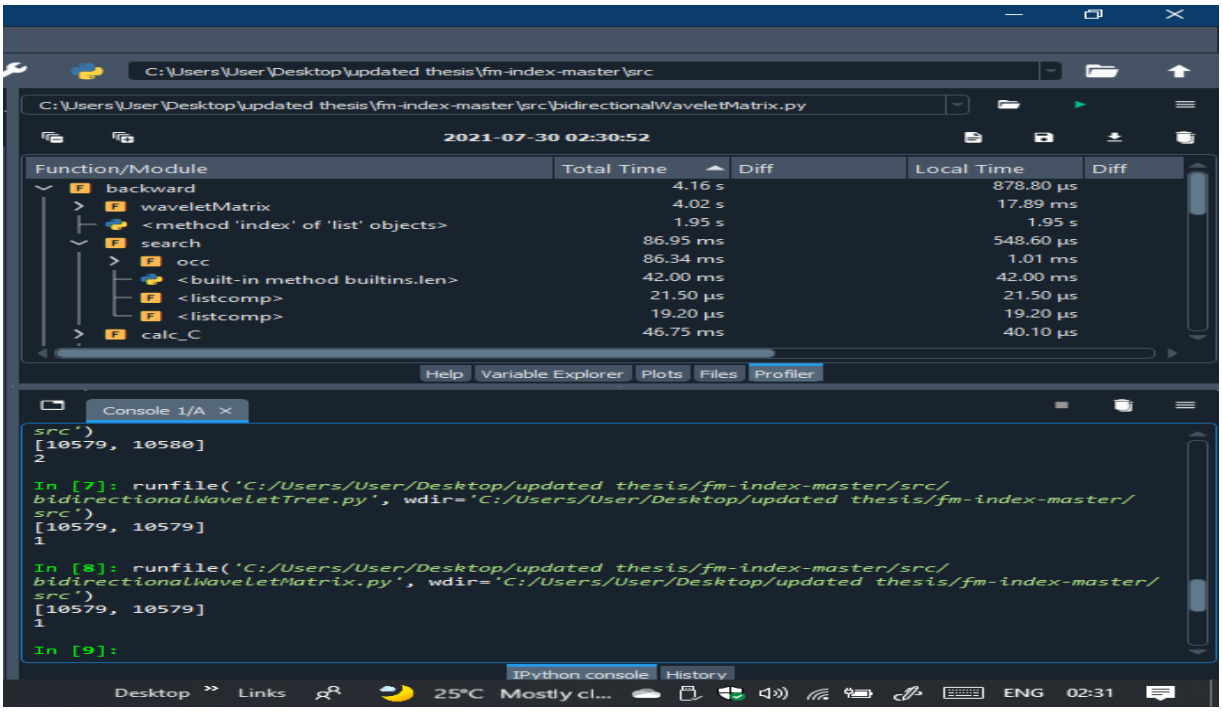
Table 4.1 Experimental results

Snapshots


C:\Users\User\Desktop\updated thesis\fm-index-master\src
\User\Desktop\updated thesis\fm-index-master\src\bidirectionalWaveletTree.py
Profile
Stop
2021-08-04 19:15:54
Output
Save data
Load data
Clear comparison
Function/Module
Total Time
Diff
Local Time
Diff
backward 246.19 ms 302.60 µs
waveletTree 130.48 ms 14.47 ms
search 93.58 ms 202.10 µs
occ 93.33 ms 6.45 ms
<built-in method builtins.len> 14.01 ms 14.01 ms
<listcomp> 18.10 µs 18.10 µs
<listcomp> 17.40 µs 17.40 µs
Variable explorer
Profiler
Help
Plots
Files
Console 1/A
In [108]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[3389, 3393]
5
In [109]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[940, 973]
34
In [110]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[853, 853]
1
In [111]:


C:\Users\User\Desktop\updated thesis\fm-index-master\src
Profile
Stop
2021-08-04 19:20:40
Output
Save data
Load data
Clear comparison
Function/Module
Total Time
Diff
Local Time
Diff
backward 2.37 s 720.60 µs
waveletMatrix 2.28 s 15.01 ms
<method 'index' of 'list' objects> 945.09 ms 945.09 ms
search 63.25 ms 197.00 µs
occ 63.01 ms 577.30 µs
<built-in method builtins.len> 37.79 ms 37.79 ms
<listcomp> 16.30 µs 16.30 µs
Variable explorer
Profiler
Help
Plots
Files
Console 1/A
In [110]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[853, 853]
1
In [111]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletMatrix.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[853, 853]
1
In [112]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletMatrix.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[853, 853]
1
In [113]:
IPython console
History
Desktop
Links
30°C
Light rain
ENG
19:21


C:\Users\User\Desktop\updated thesis\fm-index-master\src
C:\Users\User\Desktop\updated thesis\fm-index-master\src\bidirectionalWaveletTree.py
2021-07-30 02:29:10
Function/Module
Total Time
Diff
Local Time
Diff
bwt 310.21 ms 46.41 ms
<listcomp> 238.08 ms 238.08 ms
<method 'sort' of 'list' objects> 21.65 ms 21.65 ms
<built-in method builtins.len> 12.31 ms 12.31 ms
<method 'join' of 'str' objects> 335.90 µs 335.90 µs
backward 264.24 ms 305.80 µs
waveletTree 150.75 ms 12.88 ms
search 93.41 ms 355.00 µs
occ 93.00 ms 8.24 ms
Help
Variable Explorer
Plots
Files
Profiler
Console 1/A
src')
[10579, 10580]
2
In [7]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[10579, 10579]
1
In [8]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletMatrix.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[10579, 10579]
1
In [9]:
IPython console
History
Desktop
Links
25°C
Mostly cl...
ENG
02:30


C:\Users\User\Desktop\updated thesis\fm-index-master\src
C:\Users\User\Desktop\updated thesis\fm-index-master\src\bidirectionalWaveletMatrix.py
2021-07-30 02:30:52
Function/Module
Total Time
Diff
Local Time
Diff
backward 4.16 s 878.80 µs
waveletMatrix 4.02 s 17.89 ms
<method 'index' of 'list' objects> 1.95 s 1.95 s
search 86.95 ms 548.60 µs
occ 86.34 ms 1.01 ms
<built-in method builtins.len> 42.00 ms 42.00 ms
<listcomp> 21.50 µs 21.50 µs
<listcomp> 19.20 µs 19.20 µs
calc_C 46.75 ms 40.10 µs
Help
Variable Explorer
Plots
Files
Profiler
Console 1/A
src')
[10579, 10580]
2
In [7]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletTree.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[10579, 10579]
1
In [8]: runfile('C:/Users/User/Desktop/updated thesis/fm-index-master/src/bidirectionalWaveletMatrix.py', wdir='C:/Users/User/Desktop/updated thesis/fm-index-master/src')
[10579, 10579]
1
In [9]:
IPython console
History
Desktop
Links
25°C
Mostly cl...
ENG
02:31

CHAPTER 5
Summary and Conclusion

5.1 Summary

Wavelet tree is a good data structure for representing sequence over an alphabet of size σ. the larger the value of σ the space it takes. Pointer-less wavelet tree was created to reduce the space. It concatenates all the bitmaps in each level. By concatenating the bitmaps, it removes the $O(\sigma\lceil \lg n \rceil)$ from the bits. Though the pointer-less wavelet tree retains the time complexity of the standard wavelet tree, in practice it needs three times the number of operations needed in the standard wavelet tree. An alternative to the pointer-less wavelet tree is the wavelet matrix. The wavelet matrix concatenates the bitmaps in each level with a different ordering, all the children of the left node are move left and all children of the right node are move right, and we keep track of the number of zeros in each level. This ordering makes it easier to precompute the left boundary for each level's right child. In this research we use the wavelet matrix to reconstruct the Bidirectional wavelet index. The experimental results shows that the bidirectional index constructed with the wavelet matrix execute the queries faster.

5.2 Conclusion

We have reconstructed the Bidirectional wavelet index using wavelet matrix. We used wavelet matrix to achieve the logarithmic time complexity both in theory and practice while maintaining the space complexity of pointer-less wavelet tree. The wavelet matrix can be built in $O(n\lceil \lg \sigma \rceil)$ time. The experimental results showed that the Bidirectional wavelet index constructed using the wavelet matrix answer the queries faster than the index constructed with the pointer-less wavelet tree.

REFERENCES

1. Alejandro Chacón, Juan Carlos Moure, Antonio Espinosa, Porfidio Hernández. *n-step FM-Index for Faster Pattern Matching*. Procedia Computer Science, Pages 70-79, 2013
2. Francisco Claude, Gonzalo Navarro. *The wavelet matrix: An efficient wavelet tree for large alphabets*. Information Systems, 47:15-32, 2015
3. Francisco Claude, Gonzalo Navarro. *Practical rank/select queries over arbitrary sequences*. In Proc. 15th SPIRE, Pages 176-187, 2008
4. Francisco Claude, Patrik Nicholson, Diego Seco. *Space efficient wavelet tree construction.* In Proc. 18th SPIRE, Pages 185-196, 2011
5. G. Jacobson. *Space-efficient static trees and graphs*. In Proc. 30th Annual Symposium on Foundations of Computer Science, Pages 549–554, 1989
6. Giovanni Manzini. *The Burrows-Wheeler Transform: Theory and practice*. Mathematical Foundations of Computer Science, 24th International Symposium vol. 1672, Pages 34–47, 1999
7. Grossi Roberto, Vitter Jeffrey. *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*. In Proc. ACM Symposium on the Theory of Computing ACM Press, New York, Pages 397-406, 2000
8. Grossi Roberto, Vitter Jeffrey, Xu Bojian. *Wavelet Trees: From Theory to Practice*. Proceedings - 1st International Conference on Data Compression, Communication, and Processing, CCP 2011, Pages 210-221, 2011
9. Grossi Roberto, Gupta Ankur, Vitter Jeffrey. *High-order entropy-compressed text indexes*. In Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Pages 841-850, 2003
10. Hiroki Sudo, Masanobu Jimbo, Koji Nuida, Kana Shimizu. *Secure Wavelet Matrix: Alphabet-Friendly Privacy-Preserving String Search*. In IEEE/ACM Transactions on Computational Biology and Bioinformatics, Pages 1675-1684, 2019
11. Maaß Moritz G. *Linear bidirectional on-line construction of affix trees.* Algorithmica 37, Pages 43–74, 2003
12. Michael Burrows David Wheeler. *A block-sorting lossless data compression algorithm*. Research Report 124, Digital Systems Research Center. 1994
13. Patrick Dinklage. *Translating Between Wavelet Tree and Wavelet Matrix Construction*. 2020
14. Paolo Ferragina, Giovanni Manzini. *Opportunistic Data Structures with Applications*. In: Proc. IEEE Symposium on Foundations of Computer Science, Pages 390–398, 2000
15. Paolo Ferragina, Giovanni Manzini. *Indexing compressed texts*. Journal of the ACM, Pages 552-581, 2005
16. Paolo Ferragina, Giovanni Manzini, Veli Makinen, Gonzalo Navarro. *Compressed representations of sequences and full-text indexes*. ACM Transactions Algorithm, article 20, 2007
17. Stoye Jens. *Affix trees*. Technical report, University of Bielefeld, 2000
18. Strothmann Dirk. *The affix array data structure and its applications to RNA secondary structure analysis*. Theoretical Computer Science 389, 278–294, 2007
19. Thomas Schnattinger, Enno Ohlebusch, Simon Gog. *Bidirectional search in a string with*

wavelet trees and bidirectional matching statistics. Information and Computation, 213:13-22, 2012

20. Udi Manber, Eugene Myers. *Suffix arrays: A new method for on-line string searches*. SIAM Journal on Computing 22(5), 935–948, 1993