UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS

MEMORY-EFFICIENT STORAGE OF MULTIPLE GENOMES OF THE SAME SPECIES

Vinko Kodžoman

Zagreb, March 13, 2015

Acknowledgements

I would like to express my gratitude and special thanks to my mentor Mirjana Domazet-Lošo who has given me a wonderful opportunity to delve deep into a beautiful interdisciplinary field of bioinformatics and for her expert advice, encouragement and guidance through this project.

I wish to thank Marija Kalebota for the help she has given me to improve my English writing skills, for her patient guidance and useful critiques of this work.

I would also like to thank my friend Filip Jurčić for his enthusiastic encouragement and for reminding me more than once that almost anything is possible if we put our minds to it.

Finally, I wish to thank my family for their support and encouragement throughout my study, without whom I would not be here today.

Table of Contents

1. Introduction

Since the introduction of genetics and the discovery of deoxyribonucleic acid (DNA), humankind has put huge efforts into unravelling the mystery of living organisms, especially the human genome. Planned in the 1980s, started in 1990, the Human Genome Project (Sawicki *et al.*, 1993) was founded with efforts to sequence and map all the genes of our species, known as the genome, allowing us to unlock the complete genetic blueprint of a human being for the first time.

In the light of recent improvements in DNA sequencing machines, the amount of data that needs to be stored has been growing. Therefore, today there is a big shift to memory- and time-efficient methods for storing and processing sequenced data. One popular solution is building an index structure over the sequenced data, allowing fast data retrieval. Indexes require time and memory to be built, but are preferred for situations when the queries are long and numerous. One of the most popular is the FM-index (Ferragina and Manzini, 2000), which uses the Burrows-Wheeler transform (BWT) (Burrows and Wheeler, 1994). BWT is a text transformation technique based on the rotations of a given text. So far, there have been many variations of the original FM-index (Ferragina *et al.*, 2009; Navarro, 2003).

The number of sequenced human genomes has grown to more than a thousand in just one decade due to projects like the 1000 Genomes Project (Consortium, 2010). The problem of indexing the growing number of human genomes has arisen, as building an index structure for every genome separately is memory- and time-inefficient. This is caused by high data redundancy, as human genomes usually differ from each other in less than 1% (Consortium, 2010). As the number of sequenced human genomes is growing rapidly, there is a need for indexes that do not need to be rebuilt for every new genome. Recently, a new index called BWBBLE (Huang *et al.*, 2013) has been introduced, encoding all the human genomes into a reference genome. As a result, it is not necessary to build multiple indexes; instead, constructing one index for the reference genome is sufficient.

To address the problem of efficient multiple long genomes storage, I have developed a new reference-genome index inspired by the BWBBLE index. The results are presented in this thesis.

2. Research context

To fully understand the flexible multi-genome index, we need a better understanding of the underlying algorithms and methods used by the index, as well as fundamental terms and concepts. In the next section, an introduction of basic terms is given, followed by a detailed explanation of fundamental algorithms.

2.1. Definitions and notation

A *text T* or, more commonly, a *string*[1] *S,* is a sequence of *n* characters. We write $|T| = n$ to denote the length of any text. Addressing a character inside *T* is denoted by $T[i]$, or more conveniently T_i, the character at the *i*th index in *T*. The whole text is represented as $T[0, n-1]$; $T[0]$ being the first character and $T[n-1]$ the last, meaning the text indexes start from *0*. A *subtext*, or more often called *substring* is represented by $T[i, \dots, j]$ with $0 < i < j < n$. Important types of substrings include *prefixes* $S[0, j]$ and *suffixes* $S[j, n-1]$ [Figure 2.1.1]. A suffix starting at *n*th index in *S* is represented as s_n while the prefix ending at *n*th is represented as p_n. A prefix of a string S is a substring of S located at the beginning of S. A suffix of a string S is a substring located at the end of *S*.

Let $\sum = \{1, \dots, n\}$ be an *alphabet* of a size σ over a text *T*. *Lexicographic order* among two characters is defined as follows; c_i is smaller than c_j if c_i appears before c_j in the alphabet; it is often denoted as $c_i < c_j$. In the same way, we define the lexicographic ordering of strings: so that $S_1 < S_2$ means that the string S_1 is of a lexicographically lower value than S_2. In all the algorithms in this thesis, an additional character $ is added to the original alphabet. It is added to serve as a letter with the lowest lexicographical order in the alphabet. We concatenate $ to every text, thereby ensuring a unique mark to its end.

[1] In computer science, a finite sequence of characters (letters, numerals, symbols and punctuation marks) is called a string.

As the topic of this thesis is focused on genomes, the alphabet will be fairly limited, or, more precisely, $\sum = \{A, C, G, T\}$. The genome is encoded with four nucleotides; adenine (A), guanine (G), thymine (T) and cytosine (C). This information can be used to get better compression rates, as we can store the whole alphabet in only 4 bits of memory. The aforementioned \$ is not used in the compressed genome and is added after the reference genome is loaded, allowing a 4 character alphabet instead of 5. The reference genome needs to encode all the genome variations from the collection of genomes. To encode all the possible variations we need to expand the alphabet with additional characters, because there are more variations than possible code words when using the $\sum = \{A, C, G, T\}$ alphabet.

$$S = AACGCTTG\$ \qquad \sum = \{\$, A, C, G, T\}$$

Examples of prefixes:

$S[0,3] \longrightarrow S = AACGCTTG\$ \qquad p_3 = S[0,3] = AACG$

$S[0,1] \longrightarrow S = AACGCTTG\$ \qquad p_1 = S[0,1] = AA$

$S[0,5] \longrightarrow S = AACGCTTG\$ \qquad p_5 = S[0,5] = AACGCT$

Examples of suffixes:

$S[6,8] \longrightarrow S = AACGCTTG\$ \qquad s_6 = S[6,8] = TG\$$

$S[1,8] \longrightarrow S = AACGCTTG\$ \qquad s_1 = S[1,8] = ACGCTTG\$$

$S[4,8] \longrightarrow S = AACGCTTG\$ \qquad s_4 = S[4,8] = CTTG\$$

Figure 2.1.1 Prefix and suffix examples on string S

I also define the function *rank* as:

$$rank_a(S, n), \tag{1}$$

which returns the number of occurrences of a character *a* up to (and including) the character at the *n*th index in a string S [Figure. 2.1.2].

$S = AACGCTTG\$$

$\sum = \{\$, A, C, G, T\}$

A	A	C	G	C	T	T	G	$	value
0	1	2	3	4	5	6	7	8	index

$rank_C(S, 3) = 1$

A	A	C	G	C	T	T	G	$
0	1	2	3	4	5	6	7	8

$rank_A(S, 5) = 2$

A	A	C	G	C	T	T	G	$
0	1	2	3	4	5	6	7	8

$rank_G(S, 2) = 0$

A	A	C	G	C	T	T	G	$
0	1	2	3	4	5	6	7	8

Figure 2.1.2 Examples of rank query on the string S

2.1.1 Graph data structure (trees)

One of the most popular data structures used for text indexing is the tree structure (Navarro, 2003; Ferragina et al., 2004). A tree is a graph $G = (V, E)$ that consists of a set of nodes V and a set of edges E [Figure 2.1.3]. Every edge $e \in E$ in the graph is used to connect two nodes $(u, v) \in V$, denoted as $e = \{u, v\}$. A graph is *directed* if edge (u, v) is distinct from edge (v, u). The number of incoming edges (u, v) in a node $v \in V$ is defined as indegree $in(v)$ and the outdegree $out(v)$ as the number of outgoing edges (v, w).

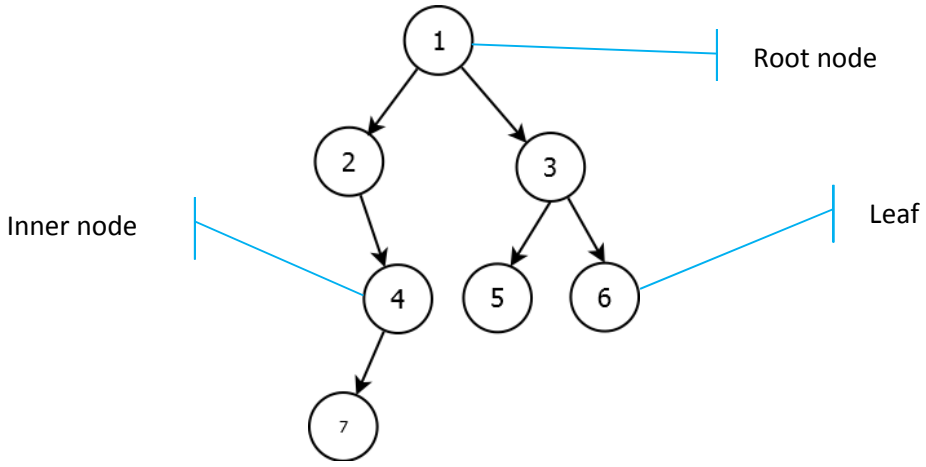

Figure 2.1.3 Directed labeled graph (tree), $ has the lowest lexicographical order in ∑

We attach *labels* $l(v)$ to every node $v \in V$ inside the graph for easier visualization, and this procedure defines a *labeled* graph [Figure 2.1.3]. For example, the root node is labeled with number "1". The top node in a tree is called the root node [Figure 2.1.3]. A *path* is a sequence of nodes. A path $P = v_1 \cdots v_{|P|}$ is an *open walk* if no node is repeated in a sequence of nodes. One path from the root node to the leaf in the tree in Figure 2.1.3 is $P = 1, 2, 4, 7$. A path that starts and ends at the same node, but otherwise has no repeated vertices or edges, is called a *cycle*.

2.1.2 Bit arrays

A *bit array* or a *bit vector* is an array data structure that compactly stores bits. Efficient use of hardware makes bit vectors time- and memory-efficient. A bit array stores $k * m$ bits; $k, m \in \mathbb{N}$, where k is the number of bits in the unit of storage and m is the number of used storage units [Figure 2.1.4]. The smallest allowed storage unit in most programming languages is one byte (8 bits). If we wanted to store 12 bits in such a programming language, we would have to take up 2 units of storage, leaving 4 bits unused and thus wasting them [Figure 2.1.4].

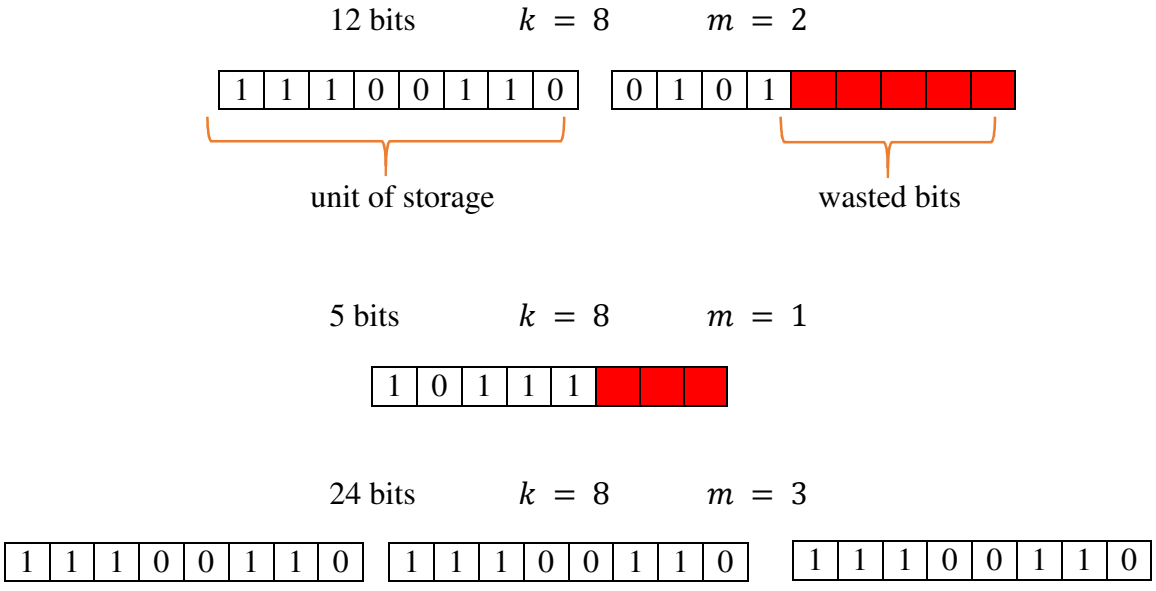

Figure 2.1.4 Example of bit arrays using a 1 byte unit storage

2.2 Suffix based data structures

One of the most difficult tasks in string algorithms is *pattern matching*, i.e. deciding whether a given *pattern* (string) occurs inside a larger string. Most of today's pattern matching algorithms rely on suffixes, as a pattern can be found as a prefix to any suffix of a given text. This simple concept was the basis for many revolutionary data structures and algorithms that we use today (McCreight, 1976; Ukkonen, 1995; Nong *et al.*, 2009; Ferragina and Manzini, 2000).

2.2.1 Suffix trees

A *suffix tree* is a tree structure [Figure 2.2.1] containing all the suffixes of a giving text. The concept was first introduced by Weiner (Weiner, 1973), later named "Algorithm of the Year 1973" by Donald Knuth. The suffix tree allows several quick operations, one of the most popular being finding the location of a substring in a given text.

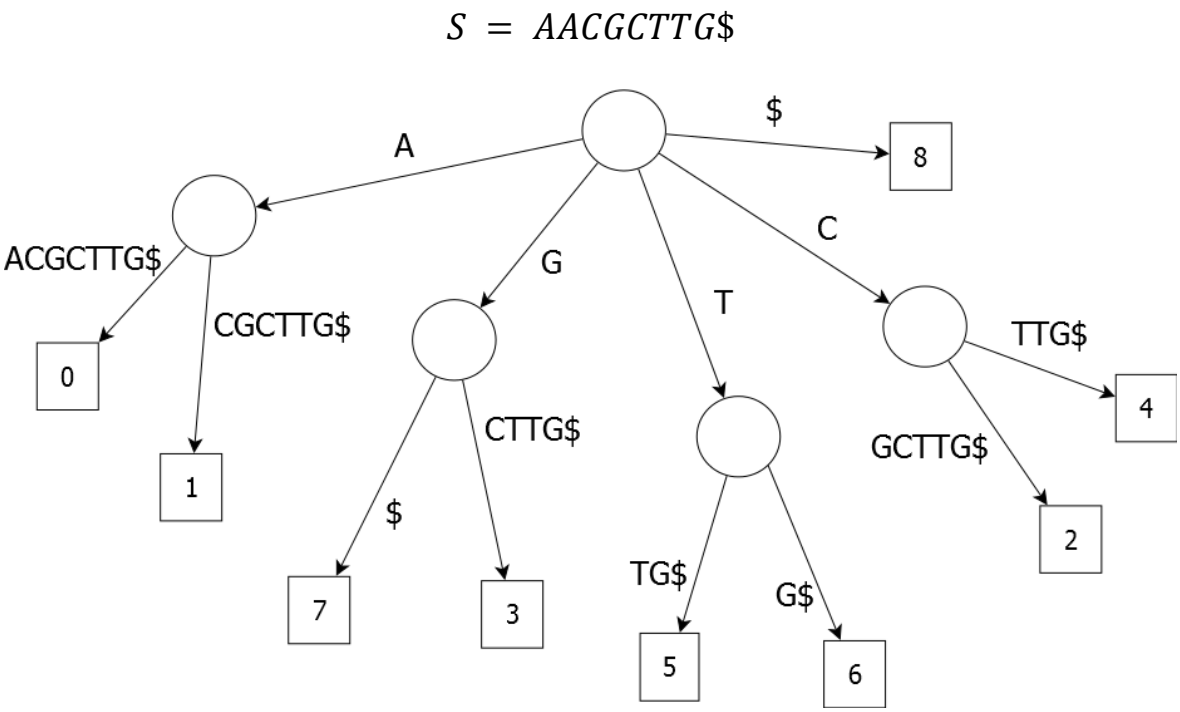

Figure 2.2.1 A suffix tree built on the string S

To build the suffix tree, we first need to find all the suffixes of the string *S* [Figure 2.2.2].

i	$suffix$
0	AACGCTTG$
1	ACGCTTG$
2	CGCTTG$
3	GCTTG$
4	CTTG$
5	TTG$
6	TG$
7	G$
8	$

A	A	C	G	C	T	T	G	$
0	1	2	3	4	5	6	7	8

Figure 2.2.2 All the suffixes of the string S

Starting with the last suffix, $s_8 = \$$, we add a leaf to the root node and write the suffix to the edge that connects them, as well as adding the index of the suffix to the leaf. For this s_8 we write \$ on the edge and the index 8 as a leaf node. Now we take the $s_7 = G\$$ and add it to the edge of a new leaf, in which we write 7 as that is the index of our suffix. When adding a suffix which has the same prefix as another suffix that is already inside the tree, we create a new node with the common prefix and continue doing the same as above. This can be seen in Figures 2.2.1 and 2.2.3 – the s_1 and s_2 have the same prefix "A" which is then extracted as an independent edge. The process is repeated until all the suffixes of the string *S* are inside the suffix tree [Figure 2.2.3]. Now the suffix tree will have one leaf per suffix of the string *S*, and a path from the root to the leaf node in the tree corresponds to one suffix [Figure 2.2.1].

One more important definition is the *suffix interval* – an interval $[L(X), R(X)]$, of all the suffixes starting with the same letter, where:

$$[L(X), R(X)] = [x, y],\ X \in \{A, C, G, T\} \quad (2)$$

$$L(X) - smallest\ index\ with\ the\ nucleotide\ X, equals\ x$$

$$R(X) - heighest\ index\ with\ the\ nucleotide\ X, equals\ y$$

The suffix interval for A is [1,2], as the indexes of suffixes s_1 and s_2 start with A, another example would be [7,8] for the nucleotide T [Figure 2.2.5].

Figure 2.2.3 shows a naive construction algorithm of the suffix tree. The construction algorithm was improved by McCreight (McCreight, 1976) and later also by Ukkonen

(Ukkonen, 1995). Ukkonen's algorithm was optimal for constant-size alphabets, constructing the suffix tree in linear time $O(n)$, where n is the length of the string. Farach (Farach, 1997) was the first to make a suffix tree construction algorithm optimal for any alphabet.

Construction and retrieval times for the suffix tree are efficient but its space complexity is large, which makes it unsuitable for large datasets. For a string of length n, we need at least $10n$ bytes, and for some strings even more than $20n$ bytes to store the whole suffix tree (Kurtz, 1999). As a result, new memory-efficient algorithms were soon introduced.

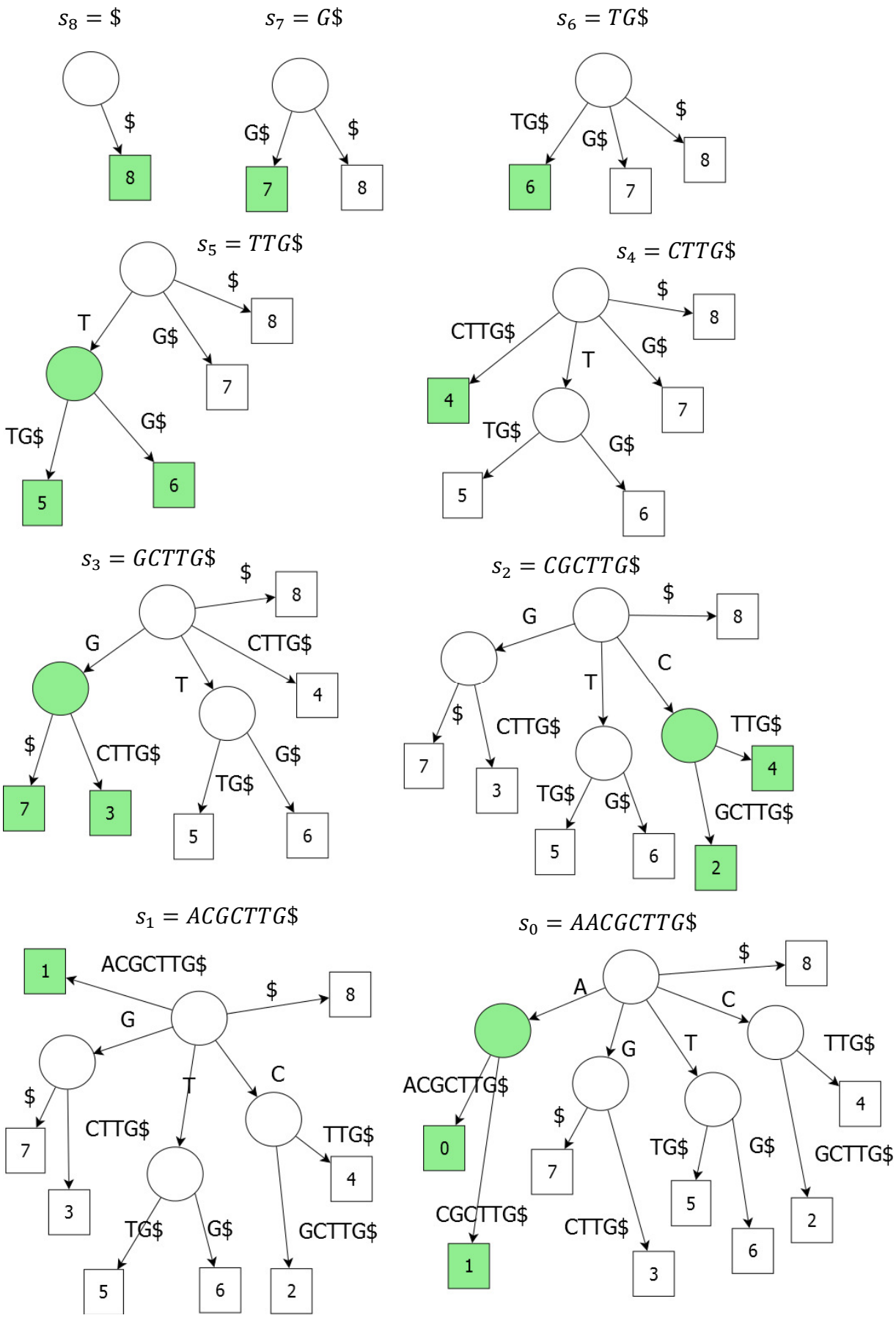

Figure 2.2.3 A naive construction algorithm of the suffix tree for the string S

To get the locations and the number of occurrences of a pattern in the string S using a suffix tree, we need to traverse the tree from its root to a set of leaves. As an example, if we wanted to check the occurrences of a pattern $P_1 = C$ in a string S, we would count the leaf nodes inside the branch which has P_1 as a prefix [Figure 2.2.4]. Reading the number inside the leaf node, we get the indexes of the locations of P_1 in S (2 and 4 for $P_1 = C$). Figure 2.2.4 shows one more example for $P_2 = TG$, which is located at index 6 (number inside the leaf node) inside the string S. It is important to see that P_2 has only one leaf node as a part of the pattern is underneath the green node. In this case, we count the leaf nodes under the last inner node that the pattern includes [Figure 2.2.4].

$$S = AACGCTTG\$$$

$P_1 = C \quad \rightarrow \quad S_2 = S_4 = C$

$P_2 = TG \quad \rightarrow \quad S_{6,7} = TG$

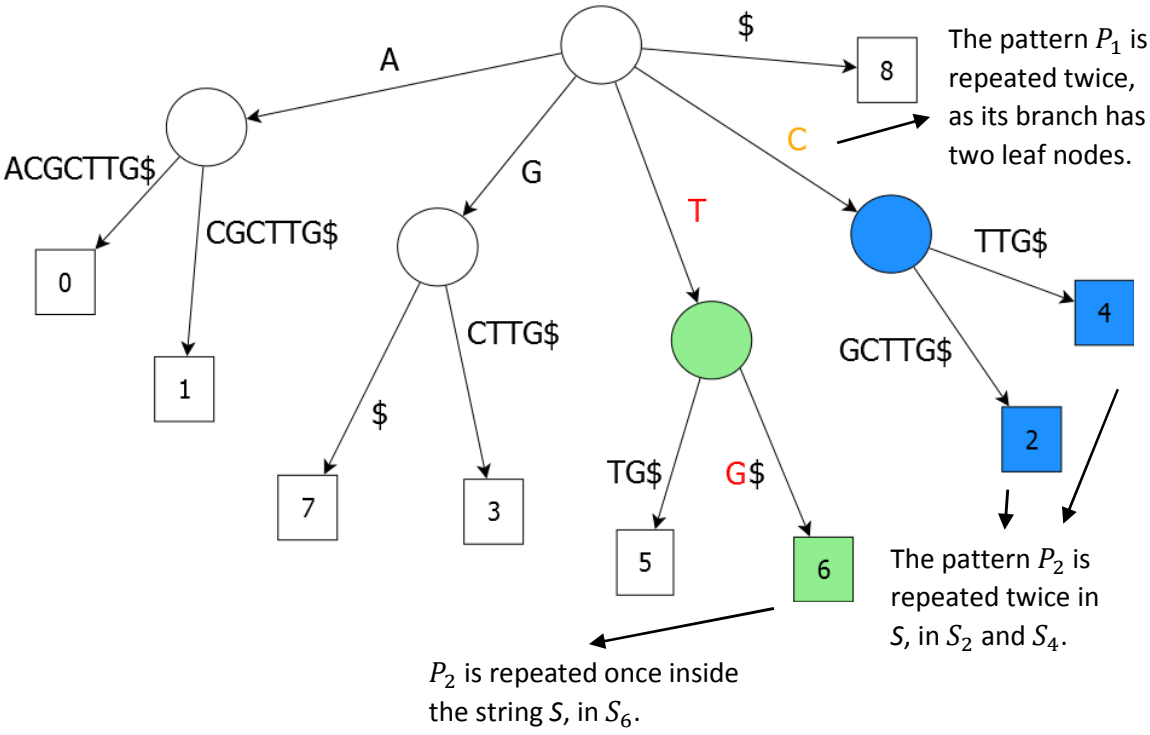

Figure 2.2.4 Patter matching using the suffix tree constructed over the string S, for patters $P = C$ and $P = TG$

2.2.2 Suffix array (SA)

Suffix arrays were introduced by Manber and Myers (Manber and Myers, 1990) in order to improve the space complexity of suffix trees. A suffix array is a lexicographically sorted array of integers built over a string [Figure 2.2.5]. Storing only integers gives them a space complexity of $4n$ bytes, which is better compared to the suffix tree which takes at least $10n$ bytes in the best case scenario (and can go up to $20n$ (Kurtz, 1999)). Furthermore, it has been shown that every suffix tree algorithm can be systematically replaced with an algorithm that uses a suffix array with additional information (Abouelhoda *et al.*, 2004).

$$S = AACGCTTG\$$$

i	$suffix$
0	AACGCTTG$
1	ACGCTTG$
2	CGCTTG$
3	GCTTG$
4	CTTG$
5	TTG$
6	TG$
7	G$
8	$

lexicographically sorted suffixes →

i	$suffix$
8	$
0	AACGCTTG$
1	ACGCTTG$
2	CGCTTG$
4	CTTG$
7	G$
3	GCTTG$
6	TG$
5	TTG$

suffix array →

i	$SA[i]$
0	8
1	0
2	1
3	2
4	4
5	7
6	3
7	6
8	5

Figure 2.2.5 Suffix array built over the string S

Suffix arrays were introduced as a memory-efficient alternative to the suffix tree. The elements of suffix arrays are the leaf nodes of the suffix tree, which can be seen in Figures 2.2.4 and 2.2.5, meaning that the suffix array stores leaf values of the suffix tree. Using this information, it is possible to construct a suffix array in $O(n)$ by constructing the suffix tree first in linear time and obtaining the suffix array by depth-first traversal of the tree. Recently, there has been an introduction of two linear time suffix array construction algorithms (Yuta Mori, 2008; Nong *et al.*, 2009) which directly construct the suffix array, and, as a result, have much lower space complexity.

The suffix array can be used as an index as well. As previously mentioned, an index is a data structure used for fast information retrieval. The goal is to locate every occurrence

of a substring pattern *P* within the string *S*. Having all the suffixes of the text allows for fast pattern matching. Counting the occurrences of *P* is equivalent to finding all the prefixes of suffixes that match the pattern *P*. This approach is shown in Figure 2.2.6.

$$S = AACGCTTG\$ \qquad P = C$$

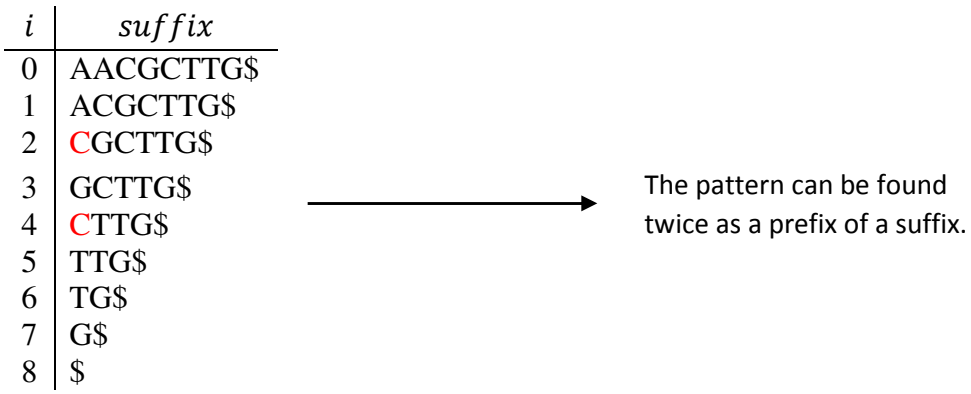

Figure 2.2.6 Finding all the occurrences of a pattern P in a string S using the suffix array

The pattern search algorithm is very similar to the one used in suffix tress – this can be seen in the Figures 2.2.4 and 2.2.6.

2.3 Burrows-Wheeler transform (BWT)

The Burrows-Wheeler transform (Burrows and Wheeler, 1994) or more commonly referred to as BWT, is a text transformation technique based on the rotations of a given text. The transformation rearranges the text in runs of the same characters, making the transformation useful for compression, as it is easier to compress chunks of the same data. The transformation is reversible, without the need for any additional data, making it a great method for improving the efficiency of text compression algorithms.

One way to construct the BWT over a text is to use all the rotations of the text. The algorithm first gets all the rotations of the text and sorts them alphabetically, and then the

last letter in every sorted rotation. The result is the Burrows-Wheeler transform. This approach can be seen in Figure 2.3.1.

$$S = AACGCTTG\$$$

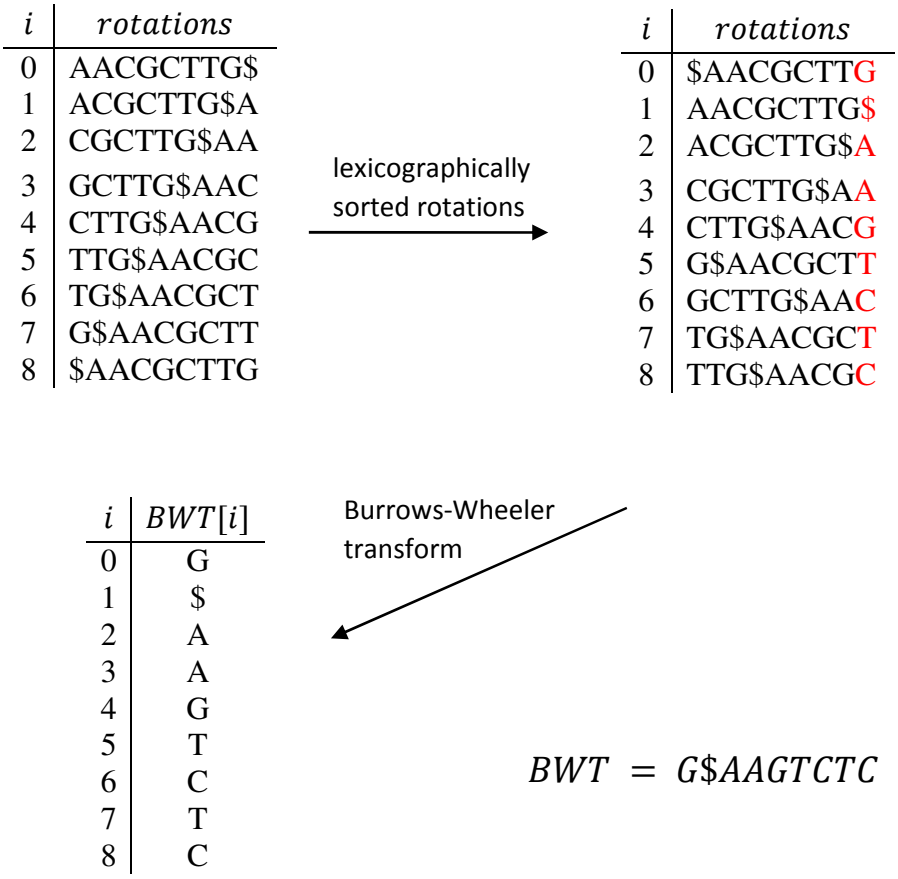

Figure 2.3.1 Constructing the Burrows-Wheeler transform over a string S

The previously mentioned grouping property of the same characters can be seen in Figure 2.3.1. The letter A is grouped on the interval $BWT_{2,3}$ and T on $BWT_{6,7}$. This property is more prominent on longer strings.

The lexicographically sorted rotations in the second step of the BWT construction algorithm in Figure 2.3.1 matches the previously sorted suffixes in the second step of the SA construction algorithm in Figure 2.2.5. The only difference is that the BWT keeps the entire rotation instead of the suffix. The conclusion is that the BWT can be easily constructed from the suffix array, using the following relation:

$$BWT[i] = \$, \qquad when\ SA[i] = 0$$
$$BWT[i] = S[SA[i] - 1], \qquad otherwise \tag{3}$$

$$S = AACGCTTG\$$$

i	$rotations$
0	\$AACGCTTG
1	AACGCTTG\$
2	ACGCTTG\$A
3	CGCTTG\$AA
4	CTTG\$AACG
5	G\$AACGCTT
6	GCTTG\$AAC
7	TG\$AACGCT
8	TTG\$AACGC

i	$suffix$
8	\$
0	AACGCTTG\$
1	ACGCTTG\$
2	CGCTTG\$
4	CTTG\$
7	G\$
3	GCTTG\$
6	TG\$
5	TTG\$

Figure 2.3.2 Comparison between sorted rotations and suffixes

In Figure 2.3.2, the correlation between the sorted rotations used to create the BWT and sorted suffixes to create the SA can been seen easily. Every sorted rotation has a suffix as its prefix – this fact can be used to create the BWT from the SA with previously described equation 14 . For example:

$$BWT[0] = S[SA[0] - 1] = S[8 - 1] = S[7] = G$$

$$BWT[1] = S[SA[1] - 1] = S[0 - 1] = \$ \rightarrow special\ case\ for\ SA[i] = 0$$

$$BWT[2] = S[SA[2] - 1] = S[1 - 1] = S[0] = A$$

$$BWT[3] = S[SA[3] - 1] = S[2 - 1] = S[1] = A$$

$$BWT[4] = S[SA[4] - 1] = S[4 - 1] = S[3] = G$$

$$BWT[5] = S[SA[5] - 1] = S[7 - 1] = S[6] = T$$

…

2.4 Prefix sum table (C)

Collecting all the *prefix sums* into an array we get the *prefix sum table*. The array is precomputed and used to preform backwards search in the reference genome index. The prefix sum table is used to compute the occurrences of all the lexicographically lower characters for each characters in ∑.

$$S = AACGCTTG\$ \qquad \sum = \{\$, A, C, G, T\}$$

$$C_{\$} = 0 \qquad S = AACGCTTG\$$$

$$C_A = C_{\$} + 1 = 1 \qquad S = AACGCTTG\$$$

$$C_C = C_A + 2 = 1 + 0 + 2 = 3 \qquad S = AACGCTTG\$$$

$$C_G = C_C + 1 = 3 + 2 = 5 \qquad S = AACGCTTG\$$$

$$C_T = C_G + 2 = 7 \qquad S = AACGCTTG\$$$

To create the prefix sum table we need to loop through all $x_i \in \sum$ and count the number of occurrences of every character that is lexicographically lower than x_i in S, doing that for every character in ∑ (4) creates the prefix sum table.

Prefix sum of a sequence of numbers $C_0, C_1, \ldots, C_n$ is a sequence of numbers $y_0, y_1, \ldots, y_n$ of the sums of prefixes of the input sequence:

$$y_0 = C_0$$

$$y_1 = C_0 + C_1$$

$$y_2 = C_0 + C_1 + C_2$$

$$\ldots$$

$$y_n = \sum_{i=0}^{n} C_i \qquad (4)$$

2.5 Wavelet tree

A wavelet tree is a succinct data structure that stores strings in compressed space. It is used to answer rank queries over the text it was built. The main advantage of the wavelet tree is its low space complexity. To achieve that complexity, the wavelet tree stores its values as bit vectors. The data structure was originally introduced to represent compressed suffix arrays (Grossi *et al.*, 2003). The tree is defined by recursively partitioning the alphabet into pairs of subsets (half for each subset), one subset for the left child node and one for the right child node. Each node contains a bit vector that represents the text within that node. Every letter is encoded with 0 or 1 depending on whether the text is part of the left or right child node subset.

$$S = AACGCTTG\$ \qquad \sum = \{\$, A, C, G, T\}$$

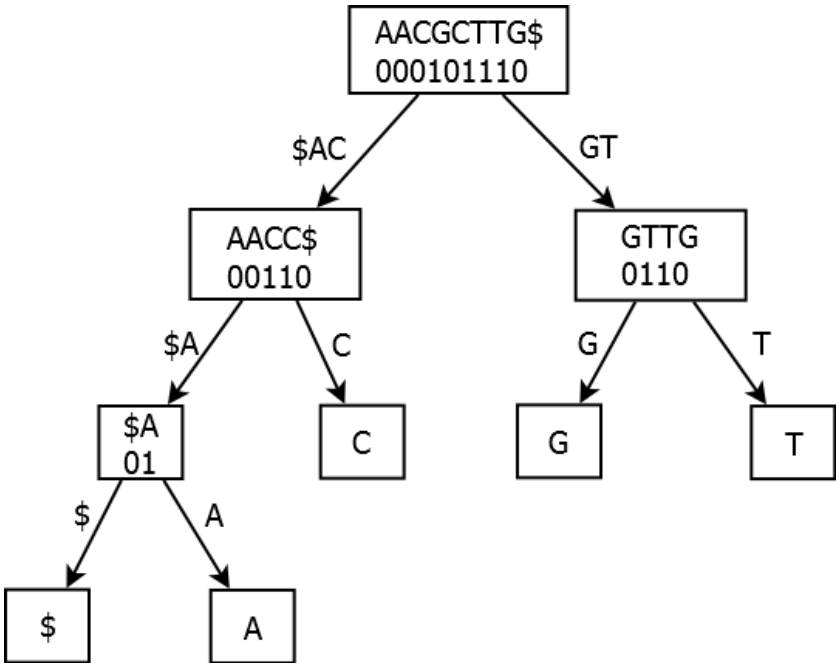

Figure 2.5.1 Wavelet tree constructed over a string S

To create the wavelet tree shown in Figure 2.5.1, we first need to partition the alphabet $\sum = \{\$, A, C, G, T\}$ into two subsets. The first subset $\sum_1 = \{\$, A, C\}$ and the second $\sum_2 = \{G, T\}$. Each subset is given to one child node. Subset $\sum_1$ is for the left child

node while the subset $\sum_2$ is for the right child node. After the alphabet is partitioned, the node values are encoded respectively. Every character that belongs to $\sum_1$ is coded with 0 while the others are coded with 1. Now the root value is split between its child nodes. The left child node gets all the values that belong to $\sum_1$ and the right node gets all the values that belong to $\sum_2$. This is done recursively until the subset becomes only one letter (leaf nodes).

$$S = AACGCTTG\$ \qquad \sum = \{\$, A, C, G, T\} \qquad rank_C(S, 6) = 2$$

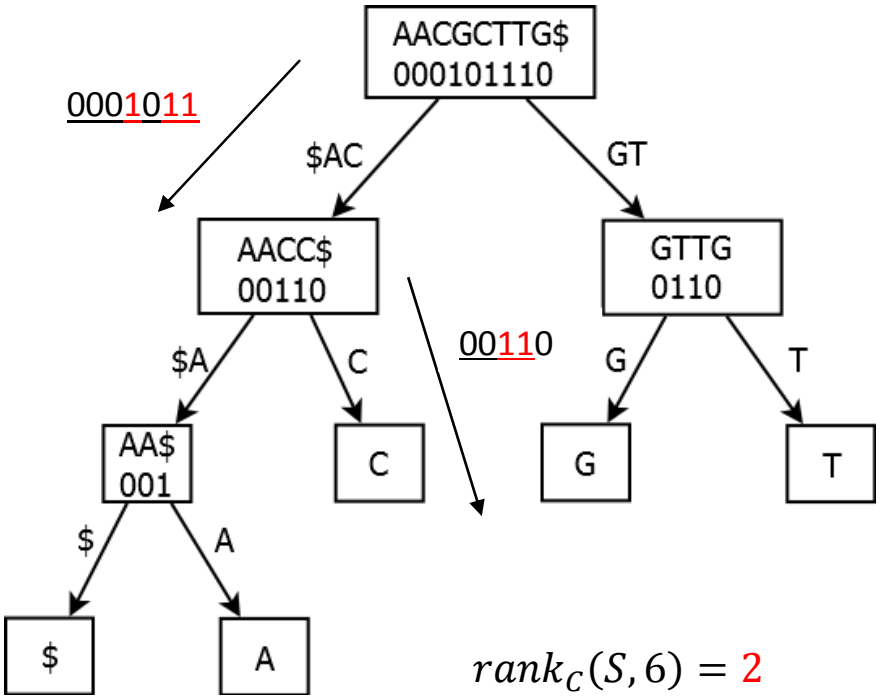

Figure 2.5.2 Rank query using the wavelet tree

To understand the rank query over the wavelet tree, we will give an example for:

$$rank_C(S, 6) = 2 \qquad \text{[Figure 2.5.2]}$$

To find the path of the wavelet tree traversal, we first need to find how the letter C is encoded. As the letter C belongs to the subset $\sum_1 = \{\$, A, C\}$, it is encoded with 0. To keep the notation consistent, the characters of the left child node subset are encoded with 0 ($\sum_1$) and the characters of the right child node subset with 1 ($\sum_2$). In the next step we need to count the number of 0s up to the 6^{th} position. Keeping in mind that S is indexed from 0, there are 4 0s

up to the 6th position. The process is repeated until the node becomes the leaf node. In the next node, C belongs to the subset $\sum_2 = \{C\}$, thus being encoded with 1 (right child node). As there were 4 0s in the root above (first step), we need to count the number of 1s up to the third position (counting from zero). The number of ones is 2 and the node is a leaf node, thus giving us the result of $rank_C(S, 6) = 2$.

The wavelet tree has become popular due to its low memory cost, which is a result of storing the node information in bits.

3. Reference Genome Index

Sequencing machines are becoming more powerful producing more high quality genome data the than ever before. For the first time ever, we have access to thousands of human genomes (Consortium, 2010). With the big increase in genome data, there is a need for efficient representation of a collection of genomes of the same species and an index that can efficiently search all the variations in the collection. The goal of the reference genome index is to create a reference genome from a collection of genomes – population with all of the variations found in the population, thus removing the bias of indexing only one genome in the population and allowing the user the search the whole population with only one index. The representation of a population is important as it is the only way to achieve a concrete representation of all the varieties found within the species. One more important goal is to make the reference genome index as efficient as possible – not hardware heavy, to allow users to use it on their home computers.

3.1 Reference Genome

A genome is all the genetic material of an organism. It consists of DNA or RNA, and of both the genes and the non-coding sequence. DNA is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms. It has two strands coiled around each other forming a double helix. The strands themselves contain smaller units called nucleotides. Each nucleotide has one of four nucleobases: guanine (G), adenine (A), thymine (T) or cytosine (C). Thus, a genome can be stored with the following alphabet $\sum = \{A, C, G, T\}$. As the goal is to encode multiple genomes, the alphabet will get more complex. The reference genome needs to have information from the whole population of genomes, it needs to have every variation even if it is only one nucleotide. A DNA sequence variation occurring within a population in which a single nucleotide differs between members of a species is called Single Nucleotide Polymorphism or SNP (pronounced *snip*) for short.

3.1.1 Encoding

With the importance of genetic sequencing and analysis, there was a need in the 1970s to make a universal notation. The International Union of Pure and Applied Chemistry (IUPAC) in 1970 formalized the nucleic acid notation (IUPAC-IUB Comm. on Biochem. Nomencl, 1970) that is universally accepted today. The notation uses the Roman characters G, C, A and T, to represent four nucleotides commonly found in DNA. The IUPAC notation extends to 16 symbols [Table 3.1.1] to encode all the possible variation among families of related genes.

Table 3.1.1 IUPAC notation

Symbol	Description	Bases represented				
A	**A**denine	A				1
C	**C**ytosine		C			
G	**G**uanine			G		
T	**T**hymine				T	
W	A or T	A			T	2
S	C or G		C	G		
M	A or C	A	C			
K	G or T			G	T	
R	pu**R**ine	A		G		
Y	p**Y**rimidine		C		T	
B	not A (**B** comes after A)		C	G	T	3
D	not C (**D** comes after C)	A		G	T	
H	not G (**H** comes after G)	A	C		T	
V	not T (**V** comes after T)	A	C	G		
N	any **N**ucleotide (not a gap)	A	C	G	T	4
#	no nucleotide (gap)					0

The same notation is used to encode the reference genome. If a reference genome can have adenine or cytosine on a certain index, that index is then encoded with M [Table 3.1.1], see the Figure below for an example.

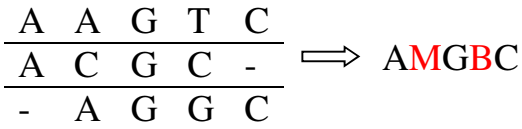

Figure 3.1.1 Example of a reference sequence created from 3 sequences using IUPAC encoding

The reference genome index needs the SA and BWT to function [LF-mapping]. The problem occurs when defining the suffix interval from the reference genome. The goal is to make a lexicographical order of the IUPAC symbols that gives the least amount of suffix intervals, and to do so, every symbol representing the same nucleotide should be near each other. For example, all IUPAC symbols that encoding the nucleotide A should be near each other within the lexicographic order (A, W, M, R, D, H, V and N). As this rule applies to each nucleotide, it creates four sets $\Theta_x, x \in \{A, C, G, T\}$. Each set comprises every symbol a nucleotide can match from the IUPAC notation, so we have: $\Theta_A = \{A, W, M, R, D, H, V, N\}$, $\Theta_C = \{C, S, M, Y, B, H, V, N\}$, $\Theta_G = \{G, S, K, R, B, D, V, N\}$ and $\Theta_T = \{T, W, K, Y, B, D, H, N\}$, these relations can be easily seen from Table 3.1.1. Using these sets the suffix intervals expand, $[L(A), R(A)]$ generates a set of eight suffix intervals:

$$[L(X), R(X)], \quad X \in \Theta_A = \{A, W, M, R, D, H, V, N\}$$

For each nucleotide there are eight suffix intervals (every nucleotide has 8 IUPAC symbols see Table 3.1.1), adjusting the lexicographical order lowers the number greatly [Table 3.1.2]. For example, if every IUPAC symbol that represents A is near each other (lexicographically), one suffix interval will be enough to represent the nucleotide A. It has been shown (Huang *et al.*, 2013) that the optimal solution of the lexicographically ordering the IUPAC code is the *Gray code* (also known as *reflected binary code*) (Gray, 1953). To get the lexicographic ordering of the IUPAC code using the Gray code, we first write the 4 bit Gray code (4 nucleotides). Now we need to arrange the four nucleotides, one example would be $b_A b_C b_G b_T$ (code table), and with each bit corresponding to a single nucleotide, the code word 1011 would match AGT, for our previous example. The first and second nucleotide ($b_A b_C$) will have one suffix interval, while the third will have two (b_G) and the fourth four (b_T). This example can be seen in the table below.

Table 3.1.2 A lexicographical order of IUPAC code with the gray code using the $b_A b_C b_G b_T$ code table

Gray code	Base				IUPAC
0000					#
0001				T	T
0011			G	T	K
0010			G		G
0110		C	G		S
0111		C	G	T	B
0101		C		T	Y
0100		C			C
1100	A	C			M
1101	A	C		T	H
1111	A	C	G	T	N
1110	A	C	G		V
1010	A		G		R
1011	A		G	T	D
1001	A			T	W
1000	A				A

From Table 3.1.2, we can get the number of suffix intervals from the base column. A and C have one suffix interval as they have no gap in the column, while G is grouped twice giving it two intervals and T is grouped four times giving it four suffix intervals. This system is flexible, as we can control the number of suffix intervals for each nucleotide with the code table. The performance of the index can be easily increased by giving one suffix interval to two nucleotides which have the highest occurrence rate within the genome, two suffix intervals to the next highest nucleotide and four intervals to the one with the lowest occurrence rate. For example, A and T occur more frequently in the human genome and using the $b_A b_T b_C b_G$ code table will give a significant performance increase, as the more frequent nucleotides (A and T) will have one suffix interval each, improving the computation speed.

Genome data is stored as bits in the reference genome. The Gray code, explained above, is used to store the genome information, allowing the whole genome to be stored at half its length (storing one char in Gray – code – using 4 bits instead of 8) and adding more genomes without an increase in size (if the added genomes are shorter).

3.2 Program structure

The reference genome index program consists of four smaller programs that can be grouped into two parts: the input/output engine (IO engine) and the index. The IO engine handles all the input data and makes it ready for the index. Programs included in the IO engine are the reference genome adder, maker and decoder. The maker creates the reference genome from all the files located in the make pool folder. The adder can add files to the already existing reference genome, while the decoder decodes the reference genome into IUPAC code (output folder). This can be seen in Figure 3.2.1, where the reference genome maker (codes the input with Gray code) uses two genomes to generate the reference genome in the output folder with the code table ACGT. If we wanted to add more reads to our reference genome located in the output folder, we can use the reference genome adder. The adder will add new reads to the reference genome with the same code table that the reference genome used when it was made. If we wanted to see what the reference genome looks like in the IUPAC code, we can just use the decoder. It will decode the Gray code into IUPAC using the code table that was given to the maker when the reference genome was originally made.

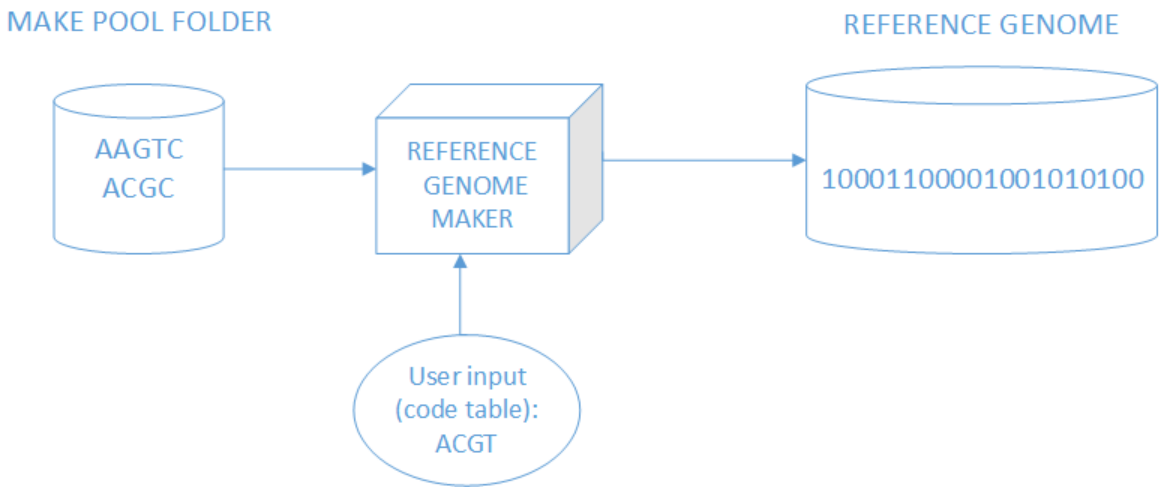

Figure 3.2.1 Program architecture, encoding two reads with the reference genome maker using the ACGT code table

The program structure allows the user to add new genomes freely and to update the reference genome with the reference genome adder [Figure 3.2.2].

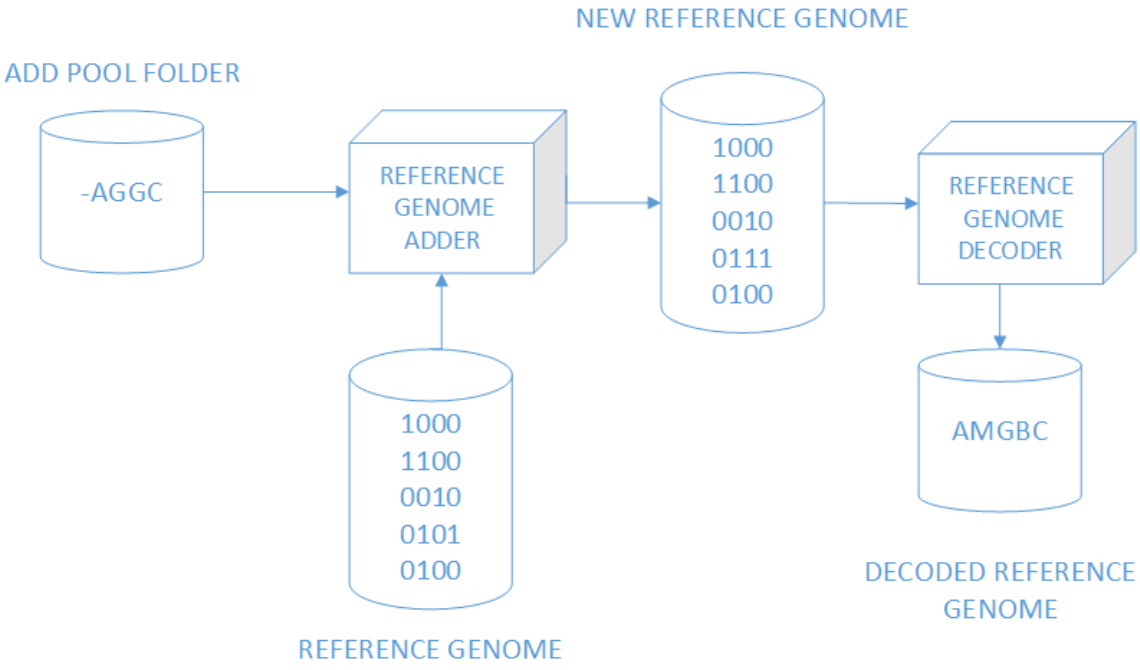

Figure 3.2.2 Adding a new read –AGGC to the reference genome and using the decoder to show the reference genome in the IUPAC CODE

The index program can use the reference genome made by the IO engine to create the index structure, which can then be used to search the reference genome. The index has the functionality to load and store all the data structures used to do the search. Using this function allows the user to save time, as loading the index from a folder is faster than creating one from the output made by the IO engine.

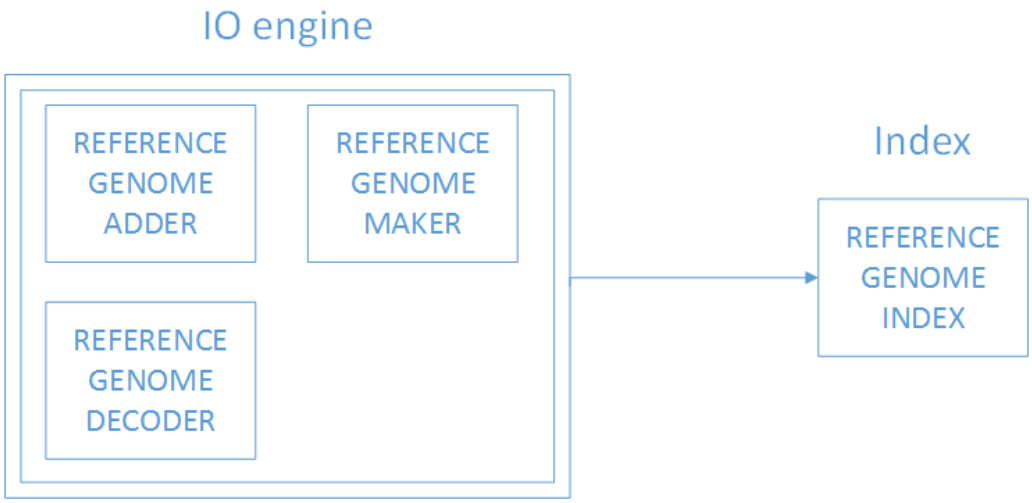

Figure 3.2.3 The program structure containing the IO engine and the index

3.3 The reference genome index

The index uses all the data structures explained before to perform the backwards search – searching from the end of the pattern to the beginning.

3.3.1 LF-mapping

When we were talking about the BWT, we said that it is reversible. The process of reconstructing the original string from the BWT is called *LF-mapping* (*Last-to-First mapping* or *Last-to-Front mapping*). *F* stands for first and *L* for last column in the BWT table [Figure 3.3.1]. The first column, *F*, matches the original string (sorted), while the last column, *L*, matches the BWT. Using the first and the last row plus some additional data structures, we can reconstruct the original string.

$$S = AACGCTTG\$ \qquad BWT = G\$AAGTCTC$$

i	$rotations$
0	$AACGCTTG
1	AACGCTTG$
2	ACGCTTG$A
3	CGCTTG$AA
4	CTTG$AACG
5	G$AACGCTT
6	GCTTG$AAC
7	TG$AACGCT
8	TTG$AACGC

i	**F**irst column	**L**ast column
0	$	G
1	A	$
2	A	A
3	C	A
4	C	G
5	G	T
6	G	C
7	T	T
8	T	C

Figure 3.3.1 LF mapping structure from the BWT

The algorithm starts from the first letter in the *F* column ($), following $ is the BWT_0 or the first letter in the *L* column (G), as it corresponds to the BWT (same index). In the next step we search for the letter G in the first column with the same occurrence of the current letter. In the *L* column the letter G was the first G to occur, so we choose the G that occurs first in the *F* column. We repeat the first step by writing down the letter that is at the same index as G, which is T. The process is repeated until we find $ in the last column (another way is to stop when the length of S' reaches the length of the BWT). The last step is just to inverse S' and the result is the original string S. This algorithm is shown in Figure 3.3.2. The LF-mapping goes backwards, so the name of the index search is backwards searching.

The prefix sum table (C) and the wavelet tree data structures are used to perform LF-mapping. The wavelet tree is used to count the occurrences used in LF-mapping, i.e. mapping the first occurring G in the *L* column with the same occurring G in the *F* column. The prefix sum table is used to skip all the characters that are lexicographically smaller. Using the wavelet tree (rank) and the prefix sum table described in chapters 2.4 and 2.5 allows us to skip the characters and perform the LF-mapping by indexes only. The number of steps in the LF-mapping algorithm matches the length of the BWT, so it is crucial that the wavelet tree and prefix sum table allow fast queries, as the index search is based on the LF-mapping structure of the BWT.

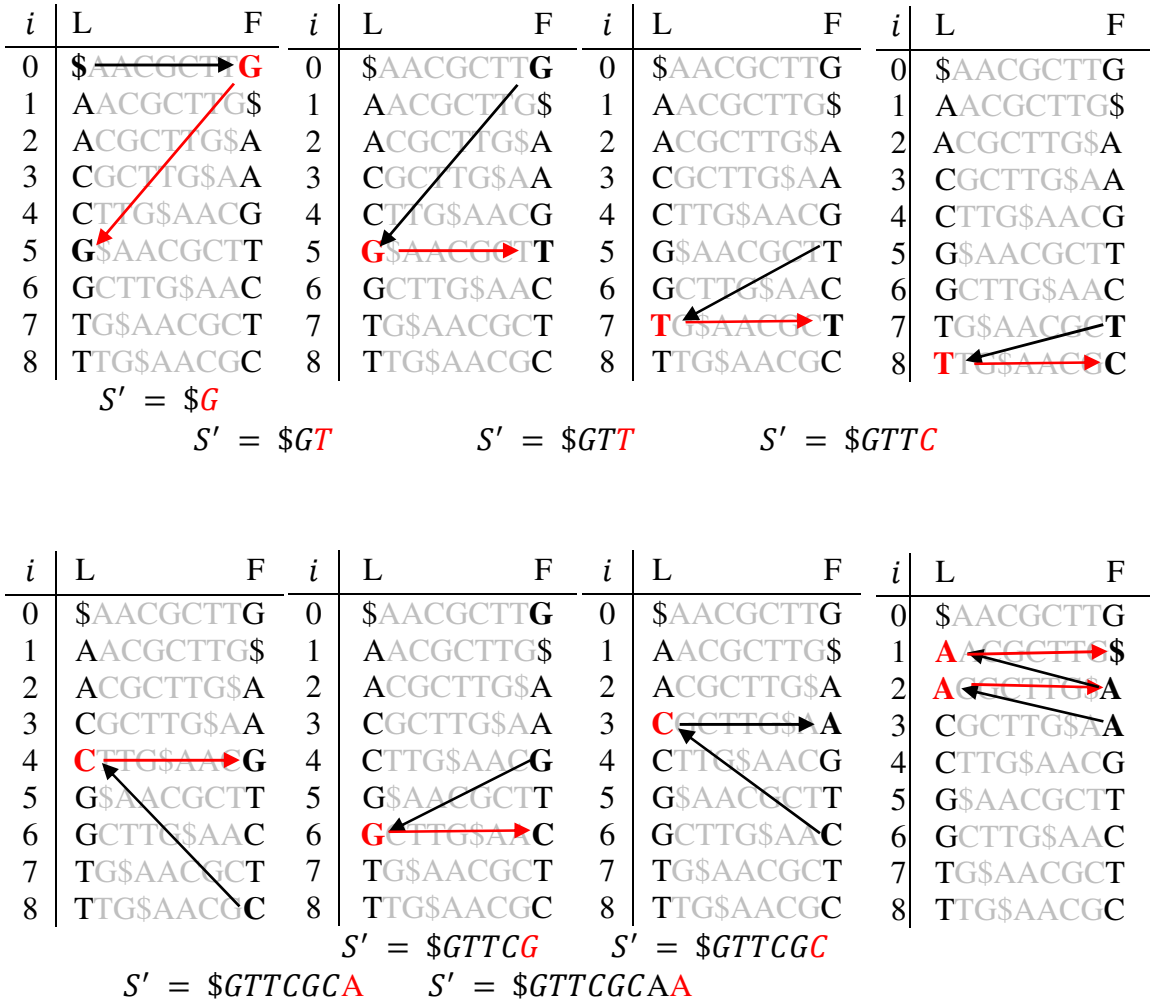

Figure 3.3.2 Last-to-First mapping algorithm over the string S

3.3.2 Backwards search

The reference genome index uses the *backward search* based on the BWT's LF-mapping. The pattern search starts with the last letter of the pattern and is moved forward with each iteration. If a pattern P occurs in a string S, each of the occurrences will appear as a prefix of a certain suffix of S. These suffixes are sorted, and as a result, grouped together into a single suffix array interval (suffix interval) $SA[L(P), R(P)]$. $L(P)$ represents the index of the lexicographically smallest, while $R(P)$ (6) represents the index of the

lexicographically largest suffix starting with *P*. Ferragina and Manzini (Ferragina and Manzini, 2000) showed that if *P* occurs in *S*, then:

$$L(\alpha P) = C(\alpha) + O(\alpha, L(P) - 1) + 1 \quad (5)$$

$$R(\alpha P) = C(\alpha) + O(\alpha, R(P)) \quad (6)$$

where $C(\alpha)$ is the number in the prefix sum table for the character α and $O(\alpha, i)$ is the occurrence of the character α up to the index i. This relation works only if $C(\alpha)$ doesn't count $ as a character, otherwise we just subtract 1 from $L(\alpha P)$ and $R(\alpha P)$. The number of occurrences of *P* in *S* is equal to the length of the suffix interval made by the above relation. This technique allows us to find the occurrences of *P* in *S* in $O(|P|)$ time, keeping in mind that the occurrence and the prefix sum were precomputed.

To use the backwards search on the reference genome (Huang *et al.*, 2013), the relation above needs to expand to match previously mentioned subsets $\Theta_x, x \in \{A, C, G, T\}$. We modify the relation to:

$$\langle L(\alpha P), R(\alpha P) \rangle = \bigcup_{\forall \sigma_x \in \Theta_x} [L(\sigma_x P), R(\sigma_x P)] \quad (7)$$

where as before,

$$L(\sigma_x P) = C(\sigma_x) + O(\sigma_x, L(P) - 1) + 1 \quad (8)$$

$$R(\sigma_x P) = C(\sigma_x) + O(\sigma_x, R(P)) \quad (9)$$

This relation (7) enables the use of iterative backwards search on a reference genome. Adding the nucleotide subsets for every character in *P* (each iteration) increases the time complexity of the backwards search algorithm, so it is important to try to minimize the number of suffix intervals.

3.4 Reference genome index implementation

To use the reference genome index the IO engine needs to create the reference genome. The index loads the references genome and creates all the necessary data structures; S, SA, BWT, alphabet and the wavelet tree (in the order of creation). The suffix array is created with the libdivsufsort library (Yuta Mori, 2008). The BWT is calculated with the SA created in the previous step using the equation (3). To create the alphabet, I used the already created SA to avoid sorting the alphabet as the SA keeps all the suffixes sorted. The

algorithm is simple: we iterate over the suffix array and use the SA values to check the character in the *S,* if the alphabet doesn't contain that character, then it should be added. The result is a sorted alphabet of the string *S* (reference genome). The wavelet tree is constructed as explained in the chapter 2.5 Wavelet tree.

After all the data structures are constructed, we can search the input string *S*. To perform the search, the backwards search relation (7) is used, as previously described (Huang *et al.*, 2013). To increase its flexibility I did not hard-code the code tables used to create suffix intervals (explained in chapter 3.1.1 Encoding), instead I allowed the user to input his own code table creating a custom lexicographical order based on the code table. This subtle difference will allow users to customize the index based on their genome data, i.e. use the code table which gives the best performance.

To increase the performance of the index and the IO engine I used hashmaps (C++ from <map>) to mapped encoded string to its proper value. For example, in the reference genome decoder I needed a way to do fast mapping of Gray to IUPAC code. The problem was that by allowing the users to input a custom code table I did not know the order of nucleotides in the Gray code. One Gray code value would map to multiple IUPAC values (AGT, ATG, GTA, …, TAG would all map to D in the IUPAC notation) – to all permutations. Instead of sorting the decoded Gray code and using the sorted order in the hasmap, I expanded the hashmap to hold all the permutations of the Gray code, thereby increasing the time complexity with a small space complexity increase.

The reference genome index also support data structure storing and loading, allowing the user to store the index when he is finished searching. Loading the index next time will bypass the construction time and save time as the construction time increases greatly with genome length [Figure 4.3.1].

4. Results

4.1 Implementation

The reference genome index and the IO engine are implemented in the programming language C++. The program uses mostly C code for optimization, including some C++ data structures for convenience. The IO engine allows the user to use a custom code table for Gray-code lexicographical order, as well as adding new genomes to the existing reference genome. The index supports pattern searches over the reference genome and saving and loading data structures. The program does not use parallel computing and can be run on a single core processor.

To compute the suffix array, I used a libdivsufsort library made by Yuta Mori (Yuta Mori, 2008). All the results are made on a laptop with the following specifications:

Table 4.1.1 Laptop hardware specifications

Operation system	Arch Linux 64-bit
Processors	Intel Core i7-3632QM 2.2GHz with Turbo Boost up to 3.2GHz
Graphic card	NVIDIA GeForce GT 650M
RAM	8 GB DDR3 memory

4.2 Monkeypox virus

To test the program, I created a reference genome from 5 monkeypox virus strains [Table 4.2.1] (Monkeypox virus strain Gabon-1988, complete genome, 2015, Monkeypox virus strain Cameroon-1990, complete genome, 2015, Monkeypox virus strain Nigeria-SE-1971, complete genome, 2015, Monkeypox virus strain W-Nigeria, complete genome, 2015, Monkeypox virus strain PCH, complete genome, 2015) found on NCBI (Information *et al.*).

Table 4.2.1 All the genomes used to create the monkeypox reference genome found on the NCBI, with their versions and lengths in KB

Definition	**Organism**	**Version**	**Length** [KB]
Monkeypox virus strain Gabon-1988, complete genome.	Monkeypox virus	KJ642619.1 GI: 661921839	196.5
Monkeypox virus strain Cameroon-1990, complete genome.	Monkeypox virus	KJ642618.1 GI: 661921661	194.3
Monkeypox virus strain Nigeria-SE-1971, complete genome.	Monkeypox virus	KJ642617.1 GI: 661921484	197.5
Monkeypox virus strain W-Nigeria, complete genome.	Monkeypox virus	KJ642615.1 GI: 661921128	197.7
Monkeypox virus strain PCH, complete genome.	Monkeypox virus	KJ642616.1 GI: 661921305	198.7

4.2.1 Suffix interval count

The goal of this experiment is to show the number of suffix intervals with each iteration while doing the backwards search over the reference genome. The experiment was done for 10 random substrings of 20 characters length from the reference genome (two from each of the genomes used to make the reference genome). The Figure 4.2.1 plots the average values with the appropriate standard deviation.

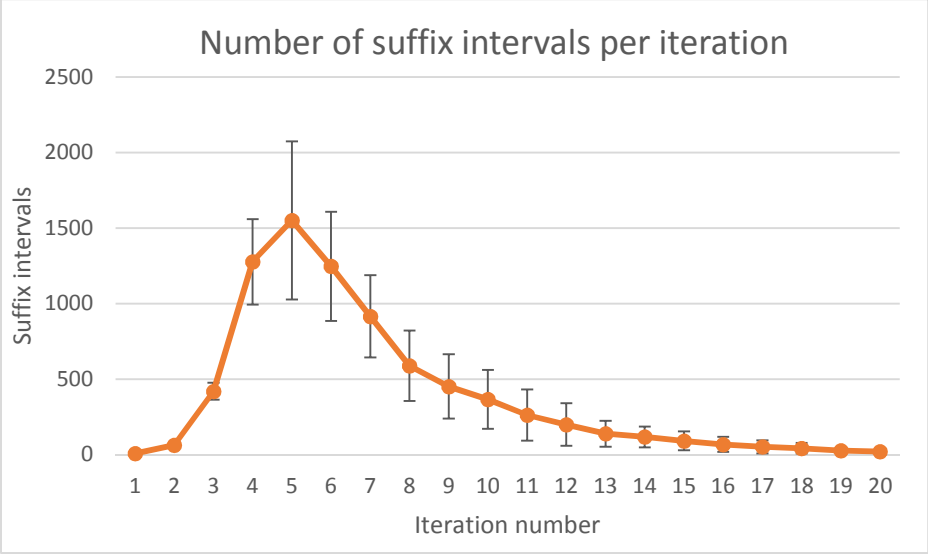

Figure 4.2.1 Number of suffix intervals per iteration of the backwards search algorithm using 8 suffix intervals for each nucleotide (20 character pattern length)

The number of intervals is highest around the beginning and slowly declines towards the end. As the calculations of the suffix intervals per iteration do not depend on each other, there are many possibilities for using multithreading solutions. One interesting solution would be to use graphic cards' high parallel processor power to compute the suffix intervals. Those solutions are not explored in this thesis and are left to be explored in the future.

4.2.2 Construction time

Here I show the construction time of the monkeypox virus reference genome made of the genomes shown in Table 4.2.1 with different code tables. In Figure 4.2.2 we can see that the construction time varies slightly. We can say that construction time is not affected much by using different code tables.

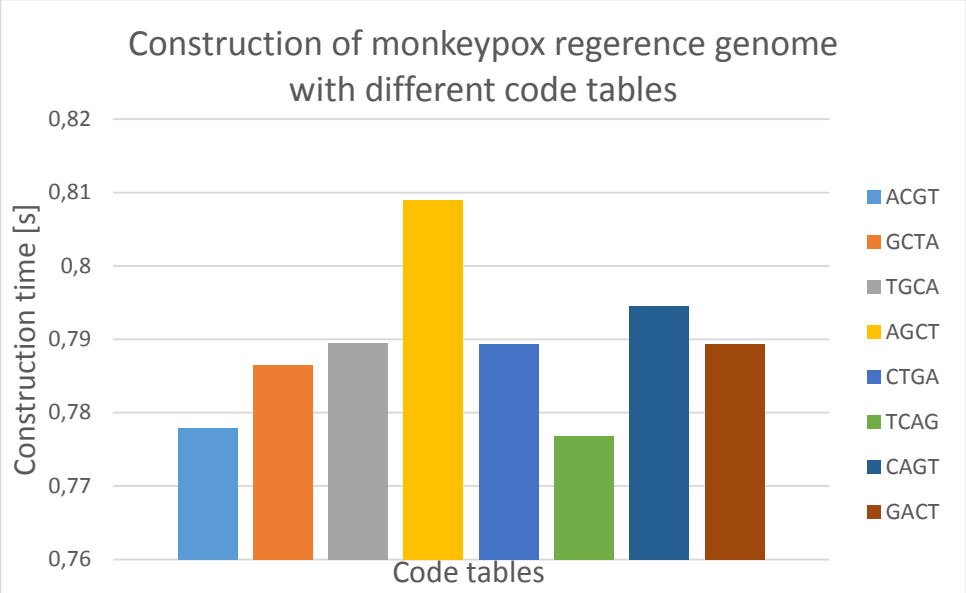

Figure 4.2.2 Construction time of monkeypox virus reference genome with different code tables using the reference genome maker (all the genomes used to create the reference genome are in Table 4.2.1)

Figure 4.2.3 shows the construction time for each data structure that the index is composed of. The index is created from the above mentioned reference genome made of gnomes found in Table 4.2.1. The obvious bottleneck is the wavelet tree used to calculate the occurrences in the backward search algorithm. The results are as expected as the wavelet tree is one of the biggest structures used in the index and there are a lot of possibilities for

optimizing it. One way to optimize it would be to add precomputed buckets in each node to reduce the counting time of the rank operation. Instead of counting different bit values, the algorithm would jump from bucket to another and read the number of zeroes and ones that were precomputed up to that index. With this approach, the algorithm would become a binary search, lowering its time complexity, but as a result we would need to add additional data structures increasing the space complexity.

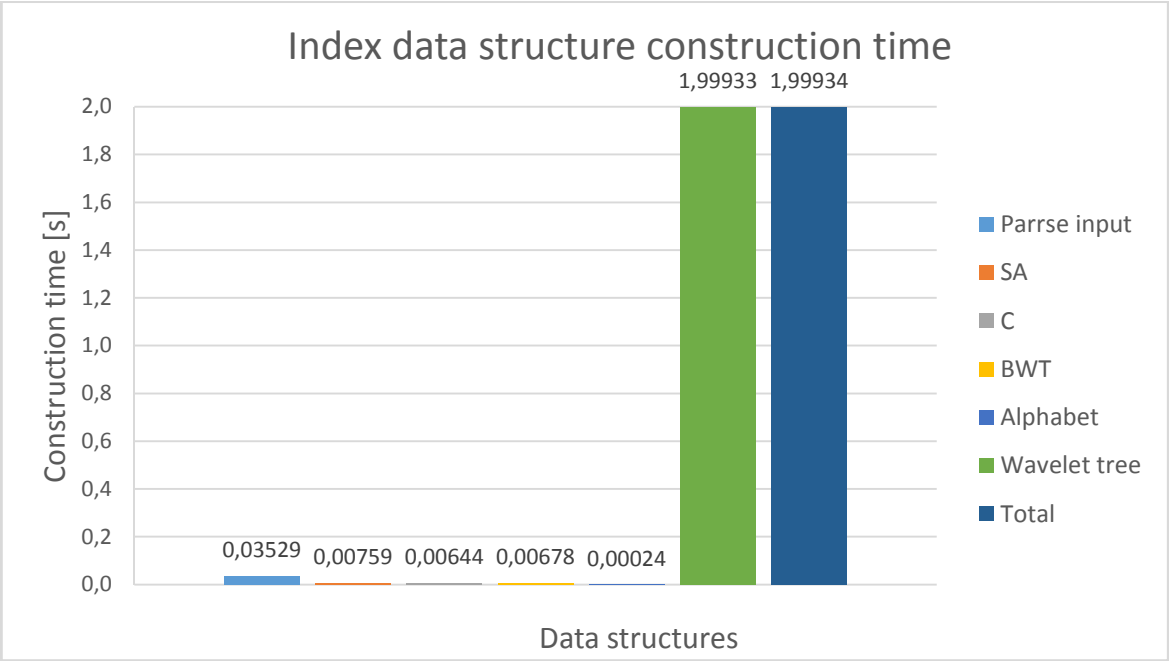

Figure 4.2.3 Index construction time separated into data structures for the monkeypox virus reference genome (all the genomes used to create the reference genome are in Table 4.2.1), the size of the reference genome is 198.7 KB

4.2.3 Space complexity

Figure 4.2.4 shows the program heap usage. As expected, the index having multiple data structures takes up most of the heap memory.

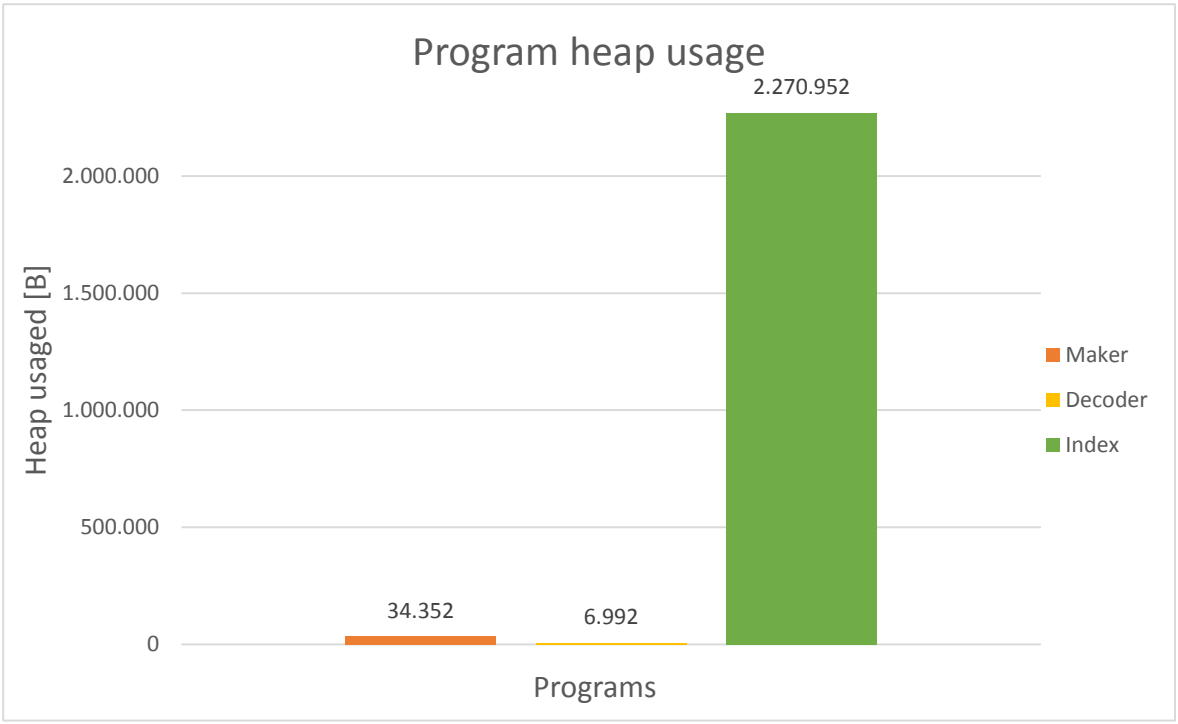

Figure 4.2.4 Reference genome index heap usage, the size of the reference genome is 198.7 KB

To get a detailed view of the space complexity of the index, all data structure construction functions with their real space complexities are shown in *Table 4.2.2*. Legend: n – length of the reference genome (compressed) and k – length of the reference genome (uncompressed).

Table 4.2.2 Real space complexity of functions creating the base data structures in the index

Data structure	Real space complexity
SA	$O(4k) + O(1)$
BWT	$O(k)$
C	$O(16k)$
Alphabet	$O(k)$

4.3 Comparison to BWA

I have compared the reference genome index with the BWA index (Li and Durbin, 2009). I have chosen 4 complete genomes [Table 4.3.1] (Canarypox virus, complete genome, 2012, Monkeypox virus strain PCH, complete genome, 2015, Sulfolobus islandicus LAL14/1, complete genome, 2014, Tanapox virus, complete genome, 2012) from NCBI (Information *et al.*) on which I tested my index construction times against the highly popular BWA index used for mapping low-divergent sequences against a large reference genome. The results are shown in Figure 4.3.1. We can see that BWA index outperforms the reference genome index on every genome selected. The reference genome construction time is competitive to BWA on lower size genomes (viruses), but with the genome size increase the wavelet tree bottleneck in the reference genome index becomes more apparent.

Table 4.3.1 Genomes used to compare the reference genome index to the BWA index, with their versions and lengths in KB

Definition	**Organism**	**Version**	**Length** [KB]
Monkeypox virus strain PCH, complete genome.	Monkeypox virus	KJ642616.1 GI: 661921305	198.7
Canarypox virus, complete genome.	Canarypox virus	NC_005309.1 GI: 40555938	359.8
Tanapox virus, complete genome.	Tanapox virus	NC_009888.1 GI: 157939619	144.5
Sulfolobus islandicus LAL14/1, complete genome.	Sulfolobus islandicus LAL14/1	NC_021058.1 GI: 479324431	2,465.1

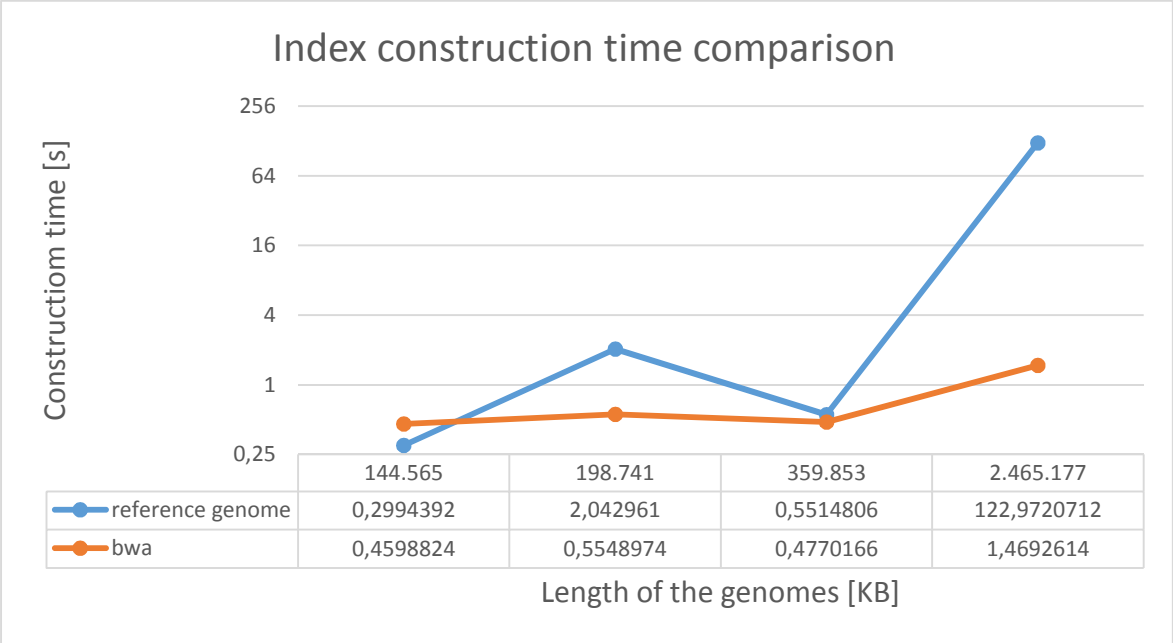

Figure 4.3.1 Index construction time comparison between the BWA and the reference genome index, the genomes used the comparison are found in Table 4.3.1

Because of the wavelet tree bottleneck, the BWA outperforms the reference genome index in construction time, but the reference genome index allows the user to create their own reference genomes from a collection of genomes making it more flexible and useable in more situations. Not only does it allow to make custom reference genomes from a population, it also allows custom code tables, making it more flexible than the BWBBLE (Huang *et al.*, 2013) index. Custom code tables can be used to adjust the memory and time complexities of the index depending on the reference genome used, as previously described.

5. Conclusion

Sequencing machines are becoming more powerful, producing more high quality genome data the than ever before. For the first time ever, we have access to thousands of human genomes (Consortium, 2010). With the big increase in genome data, there is a need for efficient representation of a population of genomes of the same species, and an index that can efficiently search all the variations in the population of genomes. That is why I have developed a reference genome index that can create a reference genome from a population of genomes given by the user, mapping all the variations of a population into a single reference genome. The index can be used to perform searches on the created reference genome, giving a high quality representation of all the variations found in the chosen species.

The reference genome index is in early development and there are many ways to improve it, which are planned as future works. Some of them include adding a flexible component to the index, allowing the user to add one or more genomes to the reference genome without the need to rebuild the entire index. I plan to utilize the parallel power of graphics cards with high numbers of graphics processors by implementing the backwards search algorithm to use the CUDA (Compute Unified Device Architecture) architecture, improving the performance even further.

The current implementation, performance-wise, is not competitive with the more popular reference genome indexes used today but the custom code tables and reference genome maker make the reference genome more flexible than the competitors. My goal is to continue to work on this subject and to improve the index.

6. Bibliography

Huang, Lin, Victoria Popic, and Serafim Batzoglou. "Short Read Alignment with Populations of Genomes." *Bioinformatics* 29, no. 13 (July 1, 2013): i361–70. doi:10.1093/bioinformatics/btt215.

Consortium, The 1000 Genomes Project. "A Map of Human Genome Variation from Population-Scale Sequencing." *Nature* 467, no. 7319 (October 28, 2010): 1061–73. doi:10.1038/nature09534.

Ferragina, Paolo, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. "Compressed Text Indexes: From Theory to Practice." *Journal of Experimental Algorithmics (JEA)* 13 (2009): 12.

Burrows, Michael, and David J. Wheeler. "A Block-Sorting Lossless Data Compression Algorithm," 1994. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.5254.

Sawicki, Mark P., Ghassan Samara, Michael Hurwitz, and Edward Passaro. "Human Genome Project." *The American Journal of Surgery* 165, no. 2 (February 1, 1993): 258–64. doi:10.1016/S0002-9610(05)80522-7.

Ferragina, Paolo, and Giovanni Manzini. "Opportunistic Data Structures with Applications." In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, 390–98. IEEE, 2000. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=892127.

Navarro, Gonzalo. "The LZ-Index: A Text Index Based on the Ziv-Lempel Trie," 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.134.2341.

Ferragina, Paolo, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. "An Alphabet-Friendly FM-Index." In *String Processing and Information Retrieval*, 150–60. Springer, 2004. http://link.springer.com/chapter/10.1007/978-3-540-30213-1_23.

Weiner, Peter. "Linear Pattern Matching Algorithms." In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, 1–11. IEEE, 1973. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4569722.

Weiner, Peter. "Linear Pattern Matching Algorithms." In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, 1–11. IEEE, 1973. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4569722.

Ukkonen, Esko. “On-Line Construction of Suffix Trees.” *Algorithmica* 14, no. 3 (1995): 249–60.

Farach, Martin. “Optimal Suffix Tree Construction with Large Alphabets.” In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, 137–43. IEEE, 1997. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=646102.

Manber, Udi, and Gene Myers. “Suffix Arrays: A New Method for On-Line String Searches.” In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 319–27. SODA ’90. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1990. http://dl.acm.org/citation.cfm?id=320176.320218.

Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch. “Replacing Suffix Trees with Enhanced Suffix Arrays.” *Journal of Discrete Algorithms*, The 9th International Symposium on String Processing and Information Retrieval, 2, no. 1 (March 2004): 53–86. doi:10.1016/S1570-8667(03)00065-0.

Yuta Mori. “Libdivsufsort.” *GitHub*. Accessed June 9, 2015. https://github.com/y-256/libdivsufsort.

Nong, Ge, Sen Zhang, and Wai Hong Chan. “Linear Suffix Array Construction by Almost Pure Induced-Sorting.” In *Data Compression Conference, 2009. DCC’09.*, 193–202. IEEE, 2009. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4976463.

Grossi, Roberto, Ankur Gupta, and Jeffrey Scott Vitter. “High-Order Entropy-Compressed Text Indexes.” In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 841–50. Society for Industrial and Applied Mathematics, 2003. http://dl.acm.org/citation.cfm?id=644250.

IUPAC-IUB Comm. on Biochem. Nomencl. “Abbreviations and Symbols for Nucleic Acids, Polynucleotides, and Their Constituents.” *Biochemistry* 9, no. 20 (September 1970): 4022–27. doi:10.1021/bi00822a023.

Gray, Frank. “Pulse Code Communication,” March 17, 1953. http://www.google.com/patents/US2632058.

Information, National Center for Biotechnology, U. S. National Library of Medicine8600 Rockville Pike, BethesdaMD, and 20894USA. “National Center for Biotechnology Information.” Accessed June 2, 2015. http://www.ncbi.nlm.nih.gov/.

“Monkeypox Virus Strain Cameroon-1990, Complete Genome,” May 11, 2015. http://www.ncbi.nlm.nih.gov/nuccore/KJ642618.1.

“Monkeypox Virus Strain Gabon-1988, Complete Genome,” May 11, 2015. http://www.ncbi.nlm.nih.gov/nuccore/KJ642619.1.

"Monkeypox Virus Strain Nigeria-SE-1971, Complete Genome," May 11, 2015. http://www.ncbi.nlm.nih.gov/nuccore/KJ642617.1.

"Monkeypox Virus Strain PCH, Complete Genome," May 11, 2015. http://www.ncbi.nlm.nih.gov/nuccore/KJ642616.1.

"Monkeypox Virus Strain W-Nigeria, Complete Genome," May 11, 2015. http://www.ncbi.nlm.nih.gov/nuccore/KJ642615.1.

Li, Heng, and Richard Durbin. "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform." *Bioinformatics (Oxford, England)* 25, no. 14 (July 15, 2009): 1754–60. doi:10.1093/bioinformatics/btp324.

"Canarypox Virus, Complete Genome," November 23, 2012. http://www.ncbi.nlm.nih.gov/nuccore/NC_005309.1.

"Sulfolobus Islandicus LAL14/1, Complete Genome," December 18, 2014. http://www.ncbi.nlm.nih.gov/nuccore/NC_021058.1.

"Tanapox Virus, Complete Genome," November 23, 2012. http://www.ncbi.nlm.nih.gov/nuccore/NC_009888.1.

Kurtz, Stefan. "Reducing the Space Requirement of Suffix Trees." *Softw. Pract. Exper.* 29, no. 13 (November 1999): 1149–71. doi:10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.