Summary

This project is an implementation for a fast and minute index.

With the development of the digital storage, the requirement for indexed searching on electronic form of information is growing gradually. In order to achieve better performance on space and time efficiency, classical indexes are combined with compression algorithm. This improvement aims at saving the space consumption and accelerating the time simultaneously. Later, a more powerful index called self-index appeared which is also a compressed index. Its advantage relies on the ability to search the pattern without original text.

In this paper, a self-index called FM-index is experimented. It is constructed mainly on the suffix array and Burrows-Wheeler compression scheme. To verify the performance of FM-index, its space performance will be compared with several common compression schemes and its time performance will be compared with traditional searching algorithm: sequential searching.

The achievements of this paper are listed below:

- I have implemented an algorithm for constructing suffix array. This is called doubling algorithm. The specific introduction can be seen in section 5.1.

- I have completed two searching algorithms on suffix array. They are binary searching algorithm in section 3.1.3 and backward searching algorithm in section 4.2.1. The sequential searching algorithm is also be used to prove the performance of suffix array.

- I have experimented the Burrows-Wheeler compression scheme which is explained in section 3.2. It includes two transformations in two directions and two traditional compression algorithms: Move-to-front coding and Huffman coding.

Acknowledgements

Firstly, I would like to thank Raphaël Clifford who proposed this project and offered me much help during the process of research and implementation.

Secondly, I am really grateful to my parents. They support me unconditionally and encourage me all the time. Moreover, I would also be appreciated for their financial assistance. They gave me this opportunity to conduct this project.

Lastly, I would like to express my deep gratitude to my friends. I can always receive help from them even some of them are not in United Kingdom. The discussion with classmates about the project has also greatly contributed to my final project.

Table of contents

1. Aims and Objectives

The investigation of fast and minute indexes are lighted by the appearance of various kinds of self-indexes. They both aims at achieving high efficiency of time and space cost. This development has great meaning for searching on the digital information. The goal of this paper is to explore the field of self-indexes and take some experiments on a particular self-index called FM-index to verify its behavior. This paper will be with a lot of persuasion if the statistics in experimental stage prove its original intention.

The objectives of this paper can be listed as follows:

a) Build suffix array:
   Construction of suffix array is also an important objective in this paper. It is an important part for FM-index;

b) Implement the suffix array searching:
   Evaluate the performance of suffix array by comparing it with sequential searching. The running time of searching for same pattern on same text will reflect the time consumption difference of them. Ideally, the running time of suffix array should be less than it of sequential searching for large size texts;

c) Implement the Burrows-Wheeler Transformer:
   Burrows-Wheeler Transformer contains two directions. The transformation is the key to following coding scheme which enhanced the performance of compression ratio dramatically;

d) Implement Burrows-Wheeler compression scheme:
   The whole Burrows-Wheeler compression scheme is not only transformation but also Move-to-front coding and Huffman coding. This two parts result in the final compressed text. The expected performance of Burrows-Wheeler compression should be in the range of other compression schemes;

e) Implement FM-index:
The complete FM-index in practice is related to the partition on the text into superbucket and bucket. This technique aims at increasing the speed and fit into internal memory.

2. Introduction

This chapter comprises three parts. The first part introduces the requirements of index according to the conditions of hardware. The second part presents the development of index from classical index to self index. The last part explains the most of the definitions and notations appeared in this paper.

2.1 The issue of index

Digital storage is gradually replacing the traditional form of information recording. Both common natural language and others includes music, program code, biological sequences are now written down in digital form. The application of electronic data is widely extended in various fields. However this trend must rely on sufficient memory to support their gigantic storage space. Fortunately, most electronic devices have expanded their storage ability enormously in recent years and triggered a new kind of daily task: digital data processing.

Text, in such an information explosion era, plays an important role in conveying mass information. Several manipulations over text are subsequently emerged. Information retrieval is the most valuable operation among them which can be used to analyze the occurrences and variants of words or to compare the distinction of sequences in biological field [1].

Definition 1: *Information Retrieval* (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies information needs from within large collections (usually stored on computers) [2].

Take natural language text as the first example, information retrieval aims at discovering the meaning of text paragraphs and their relevance to the information need of a human. For computational biology which works on DNA, protein, or gene sequences, information retrieval helps to extract regulations on evolutionary history, biochemical function, chemical structure, and so on [1]. All the related applications which involve information retrieval demand a basic task on text called string matching. String matching can be classified to two kinds of text searching: sequential text searching and indexed text searching which is the focus of this

paper.

Sequential text searching takes a sequential traversal on text directly to find out all the occurrences of a specified pattern. Instead, indexed text searching processes the text beforehand [1]. The primary advantage of the indexed text searching is the speed acceleration in searching. For example, while an index of 10000 documents can be queried within milliseconds, a sequential scan of every word in the documents could take hours [3]. The conditions on which indexed text searching is preferred can be concluded as follows [1]:

a) Sequential text searching is costly on the consideration of time and storage;

b) Consumption of building and maintaining the index should not affect the efficiency of text searching;

c) Storage space can afford the demand of storing both text and its index.

In practice, the indexed text searching is more widely employed owing to the considerable capacity increment of digital devices and the benefit of time saving. Though the space consumption seems to be the vital aspect for indexed text searching, the major barrier behind it is not storage space but the speed to access them. Except for the simple application on natural language text, such as inverted index, most of the indexes for text searching require from 4 to 20 times the text size [1]. Basically, the original text is stored on the main memory while the corresponding index will be built on the disk.

With respect to the access efficiency, main memory is accessed randomly and access time ranges between 10^{-8} and 10^{-7} seconds. On the other hand, the random access to disk is usually more than 10^{5} times slower than the access to main memory as showed in table 1 [4]. Transferring the data from the disk into main memory is the bottleneck in searching process. As a result, the indexed text searching, to some extent, is confined to the text whose index is lightweight where sequential searching is also suitable on it.

	Processor cache	Random access memory	Hard drives
Access speed	Very fast	Fast	Slow
Storage	Small capacity	Medium capacity	Very large capacity

Table 1. Table of computer memory.

It is now compulsory to fit the index in internal memory and leave as few data as possible onto disk. Text compression technique, in this context, appears to be a fascinating solution to store the text in less space. Besides, some attempts have been devoted to improve the space requirement even more by replacing the text and searching the pattern on the compressed one directly [1].

The main design factors for index include six points [3]:

a) *[Merge factors]*: How data enters the index, or how words or subject features are added to the index during text corpus traversal, and whether multiple indexers can work asynchronously. The indexer must first check whether it is updating old content or adding new content;

b) *[Storage techniques]*: How to store the index data;

c) *[Index size]*: How much computer storage is required to support the index;

d) *[Lookup speed]*: How quickly a word can be found in the inverted index;

e) *[Maintenance]*: How the index is maintained over time;

f) *[Fault tolerance]*: How important it is for the service to be reliable.

Fast and minute index, as the topic in this paper, acts as a method to provide an index algorithm which is fast in speed and tiny in space.

2.2 The development of index

As described above, speed and space are the primary concerns in evaluating an index algorithm. In recent years, many investigations have been taken to improve the performance of index. Four kinds of representative indexes are proposed in previous work:

a) *[Classical index]*: Suffix tree and suffix array are both typical classical indexes. Suffix tree is a type of trie. Suffix array is an alternate representation of suffix tree which is considered to occupy less space and can be combined with data compression scheme such as Burrows-Wheeler Transformer compression.

b) *[Succinct index]*: Succinct index is an index that provides fast searching functionality using a space proportional to that of the text itself (two times the text size) [1].

c) *[Compressed index]*: Compressed index is a strong concept which takes advantage of the regularities of the text and uses a space proportional to that of the compressed text (3 times the zero-order entropy of the text) [1].

d) *[Self index]*: Self index is a more powerful concept which is also a compressed index. In addition to satisfying searching functionality, it holds auxiliary information to efficiently reproduce the original text. In other words, self index can replace the text in some way [1].

The last three advanced indexes all intent to accelerate the searching speed by reducing the index storage. The first succinct index is proposed in 1996 by Kärkkäinen and Ukkonen [5]. Then the first self-index appeared in 2000 which was proposed by Ferragina and Manzini [6]. This breakthrough initially revealed the relationship of text compression and text indexing. Since then on, many other self indexes were suggested which all apply the compression algorithm to optimize the performance of index. So far, an ideal index can be defined as follows:

a) The required space for the index should be similar to the compressed text;

b) This index has the capacity for fast searching and replacing the text at the same time.

2.3 Notations and definitions

This part lists all the correlative definitions and notations mentioned in this paper. For the convenience of reading, all of them are in uniform representations.

- String S: A sequence of characters or symbols;
- Alphabet $\sum$: A finite set of characters or symbols;
- Size σ: The size of alphabet $\sum$, denoted as $|\sum| = \sigma$;
- Substring $S_{i,j} = S_i S_{i+1} \dots S_j$: The prefix of string $S_{1,n}$ is denoted as $S_{1,j}$ and the suffix of string $S_{1,n}$ is denoted as $S_{i,n}$;
- Lexicographical order $' < '$: Defines the lexicographical order between two strings. Let a and b be characters, X and Y be strings. $aX < bY$, if $a < b$ or if $a = b$ and $X < Y$.

In this paper, the search problem is defined as [1]. Given a long text $T_{1,n}$ and a comparatively short pattern $P_{1,m}$ both strings over alphabet $\sum$. Find all the occurrences of $P_{1,m}$ in $T_{1,n}$. Three important operations discussed in this paper are as follows:

a) *[Counting query]*: Count the number of occurrences occ;

b) *[Locating query]*: Locate all the positions of $P_{1,m}$ in $T_{1,n}$;

c) *[Extracting query]*: Display the specified substrings of $T_{1,n}$. For example, return $T_{l,r}$ given l and r.

3. Related Background

3.1 Classical indexes

Two main branches of classical index are suffix tree and suffix array. Suffix array is a space efficient substitute of suffix tree. Both of them permit to answer searching problem. Their shortcomings are the expensive construction process. Besides, the original text must be at hand at query time and the results are not possible to be delivered in text position order [7].

3.1.1 Trie

Before describing the suffix tree, it is necessary to first introduce a data structure called trie. It is essential to the concept of suffix tree.

Trie is initially proposed by Fredkin in 1960 [8] and is widely used in digital searching. It is constructed on a finite set of strings over an alphabet and is presented in a form of traditional tree data structure. In a trie, each path from root to leaf stands for a specific string in the set. Each edge, for convenience, is labeled by a character from the alphabet which corresponds to the string on its path. Assume every string in the set is not a prefix of another and then the number of leaf is exactly the number of strings in the set.

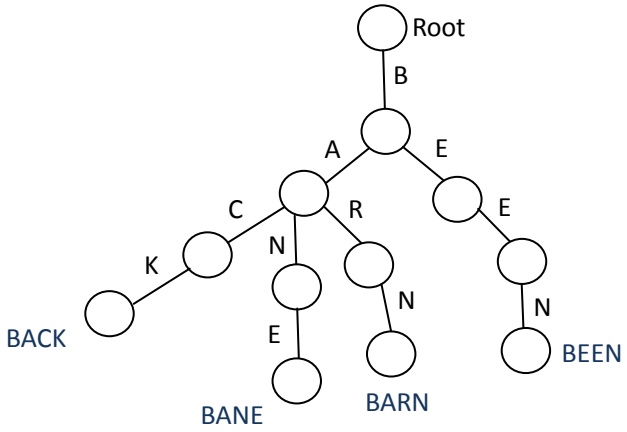


Figure 2. Trie for the set {BACK, BANE, BARN, BEEN}.

Take a string set $\{BACK, BANE, BARN, BEEN\}$ as an example, we can construct its trie as figure 1.

For purpose of searching, we need to follow the edges in the trie under the guidance of characters in the pattern and select the correct branch according to the label on its edge [9]. The process of searching will be terminated in two situations:

a) Searching route arrives at a leaf and the string to be retrieved is found. Searching is successful;

b) Searching route stops at a node and there is no any other subnode to follow. Searching is unsuccessful.

To pursue small amount of storage space and fast access time, considerable discussions about the implementation of trie take place. Two mainstreams among them are tabular implementation and linked list implementation [10].

Tabular implementation is described by Fredkin when he introduced trie in 1960, where each nonleaf node is recorded by an element array with the same size as alphabet. It guarantees a constant time for testing a node but becomes inefficient in storage space.

Linked list implementation then came up in 1963 by Sussenguth [11]. It allows a node to be a linked-list of at most 27 (the size of alphabet) elements. This method saves the space but increases the access time. A relatively new idea is seen in the paper of Jun-Ichi Aoe in 1992 [12]. He presented a double-array implementation for trie. This proposal improves the speed of retrieval operation of linked-list trie.

In tabular implementation case, when the size of alphabet size is σ, the searching for the matched character of each node in its table consumes$O(1)$. However, the storage cost for construction is $O(\sigma)$ for each node.

Trie can, as an exceptional situation, be applicable to prefix searching when string set is a subset of text $T_{1,n}$. This is a particular feature to be exploited in the next data structure called suffix tree.

3.1.2 Suffix tree

Suffix tree is a data structure with extensive applications from longest common

substring of two strings to recognizing DNA contamination. Even so, the most attractive virtue of suffix tree is that it provides solutions to many delicate string problems in linear time. Moreover, it associates the exact matching problems with inexact matching problems [13].

It is Weiner that firstly proposed the algorithm of constructing suffix tree in linear time in 1973 [14]. A few years later, McCreight presented another algorithm to build suffix tree in a more space efficient way [15]. A general suffix tree, in a concise definition, is actually a trie data structure built over all the suffixes of $T_{1,n}$. Given a long text $T_{1,n}$ with n suffixes, the task of building its suffix tree can be converted into building a trie over all its suffixes set $\{T_{n,n}, T_{n-1,n}, T_{n-2,n}, \dots, T_{1,n}\}$. To make all these suffixes prefix-free, a special termination symbol is added to the text as $T_{1,n}\$$. Figure 2 illustrates an example of a general suffix tree established over text $'xabxa'$.

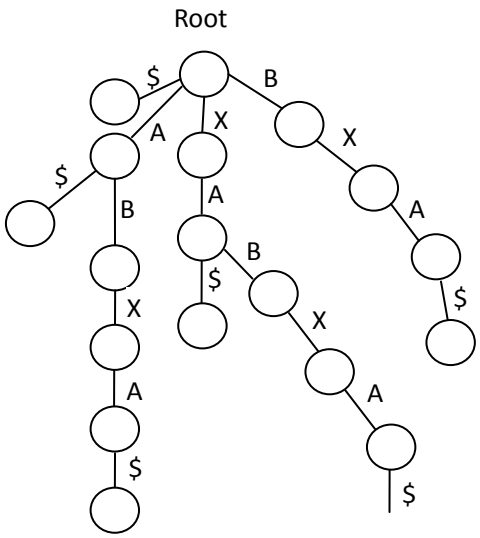


Figure 2. General suffix tree for text 'xabxa'.

Later, several amendments are appended to simplify the general suffix tree on the consideration of time and space cost. Esko Ukkonen thus developed an algorithm to meet the demand of linear-time in suffix tree construction with a laconic explanation. As a result, implicit suffix tree is emerged according to his idea which is more practical [13]. Modifications on the general suffix tree are listed as follows and an example is shown in figure 3:

a) *[Step1]*: Compress the general suffix tree by pruning the node whose trailing path is unary without any branch. Label the edge with pruned symbols in string form. For example, the right branch can be compact to $'BXA\$'$;

b) *[Step2]*: Remove the terminated symbol $ from every edge label;

c) *[Step3]*: Remove every edge with no label at all;

d) *[Step4]*: Remove any node that has only one subnode;

e) *[Step5]*: Label every leaf with the start position of its suffix.

As for searching problem in our paper, the central underlying idea is that the pattern must be treated as a prefix in one or more suffixes of $T_{1,n}$. For general suffix tree, counting the occurrences of pattern $P_{1,m}$ only need to follow the whole tree until the last symbol of pattern is found, then the occurrences is the number of leaves in its subtree. To analyze the time complexity, the counting query needs $O(m)$ time while the task of locating needs several more pointers to indicate the positions of remaining substring in suffix tree. Thus, it may need $O(m + occ)$ time to obtain the positions of every leaf which contains the specific pattern.

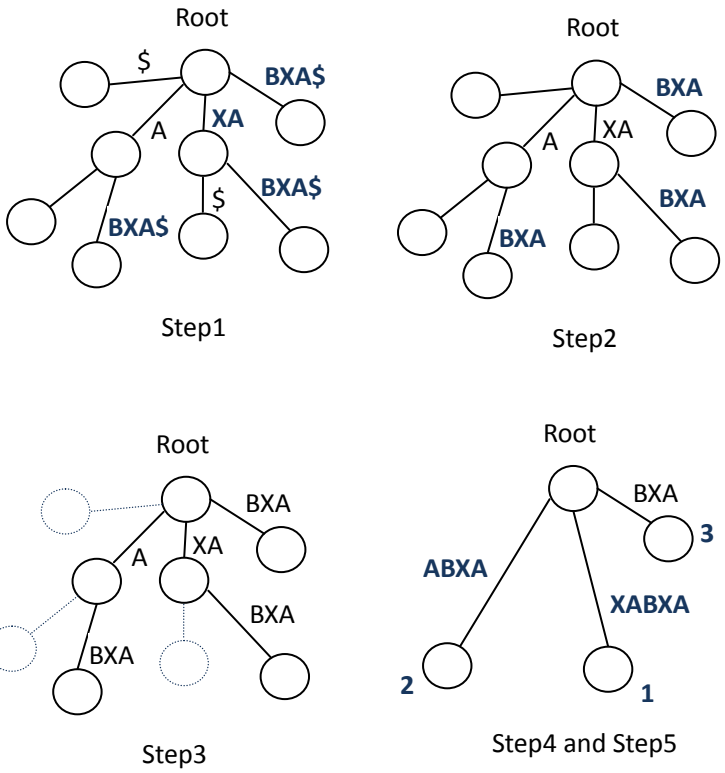


Figure 3. Implicit suffix tree for text ‘xabxa’.

The process of searching problem will finally end with four situations:

a) Searching ends at a leaf with no edge to follow and the symbols of the pattern

have not been found completely. This means the pattern $P_{1,m}$ is not in the text $T_{1,n}$;

b) Searching ends at a node whose edge doesn't match the pattern. This also indicates that the pattern $P_{1,m}$ is not in the text $T_{1,n}$;

c) Searching ends at a leaf and the symbols of the patterns have been found completely. This means that the current path is the only satisfied result;

d) Searching ends at a node and the symbols of the patterns have been found completely. This indicates the whole subtree of this node are all the results.

Suffix tree has showed its advantage in time of searching process. Nonetheless, the space consumption is, on the contrary, relatively greedy. Even the most space efficient previous implementation technique for suffix tree requires $28n$ byte in the worst case where n is the length of the input text [16]. The space consumptions of suffix tree are concluded as follows by previous authors:

- Manber and Myers state that their implementation of suffix tree needs $18.8n$ to $22.4n$ bytes of space for real input strings such as text, code and DNA.

- Kärkkäinen claims that the space requirement of his suffix tree is between $15n$ to $18n$ bytes for DNA sequences.

Take the trend of growing text size into account, expensive space demand is the dominating drawback of suffix tree. An improving approach is desired.

3.1.3 Suffix array

Suffix array is essentially a compact version of suffix tree. It is suggested to make up for the space inefficiency of suffix tree with same functionality. Suffix array is not only a classical index structure. It can also be combined with Burrows Wheeler Transform for compressed index. Manber and Myers first put forth this data structure in 1993 [17]. The suffix array is simply an array containing all the pointers to the text suffixes sorted in lexicographical (alphabetical) order. Since they store one pointer per indexed suffix, the space requirements are almost the same as those for inverted indices (disregarding compression techniques) which is close to 40% overhead over the text size [7].

Given a text $T_{1,n}$, suffix array A is an array with indices 1 to n which implies the lexicographically order of all the suffixes of text $T_{1,n}$. For all the i $(0 \leq i \leq n)$, we

can get $T_{A[i],n} \leq T_{A[i+1],n}$ where $' < '$ represents the lexicographical relationship.

In particular, the special symbol \$ is the smallest symbol. To show an example, the suffix array of text $T = 'mississippi'$ is depicted below:

a) Firstly, all the suffixes of T are permutated in lexicographical order as shown in table 2 with their start positions and lexicographical rank;

Suffix	Start position	Lexicographical rank
i	11	1
pi	10	6
ppi	9	7
ippi	8	2
sippi	7	8
ssippi	6	10
issippi	5	3
sissippi	4	9
ssissippi	3	11
ississippi	2	4
mississippi	1	5

Table 2. Lexicographical order of text 'mississippi'.

b) Secondly, stores the start position of each suffix into array according to the order of lexicographical rank. Then the suffix array is generated as figure 3.

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

$A[0] = 11; ('i')$ $A[1] = 8; ('ippi')$ $A[10] = 3; ('ssissippi')$

Figure 3. Suffix array of text 'mississippi'.

On the whole, suffix array can be built in two ways. One is converting from the suffix tree directly [13] and another way is setting up it directly.

- Method 1: Converting from suffix tree

The suffix array of text $T_{1,n}$ can be obtained by performing a lexicographical depth-first traversal on its corresponding suffix tree. Once the suffix array is constructed, the space of suffix tree can be set free [13].

First of all, the definition of lexicographical order of edges in suffix tree is given. An edge (v, u) is lexicographically less than an edge (v, w) if and only if the first symbol on the (v, u) is lexicographical less than the first symbol on (v, w) [13].

In a suffix tree, there are no two edges outgoing from an identical node v that have the labels with exactly the same beginning symbol. This key characteristic leads to the possibility of lexicographical depth-first searching. Specifically, the smallest suffix of $T_{1,n}$ can be gained by following the smallest edge of each node in suffix tree. Generally, all the leaves can be encountered in lexicographical order by traversing the edges out of each node in identical lexicographical order. Suffix array, consequently, can be built on the result order.

Due to the linear time of suffix tree construction and the linear time of traversal, creating suffix array for a text $T_{1,n}$ from its suffix tree can also be in $O(n)$ time.

- Method 2: Build directly

The most popular algorithms for suffix array construction is doubling algorithm and DC3 algorithm [18]. The first algorithm is selected in the implementation stage of this paper which will be discussed in section 5.1.

Function $DC3(T)$
S:= $[((T[i,i+2]),i) : i \in [0,n), i \bmod 3 \neq 0]$ (1)
sort S by the first component (2)
P:= $name(S)$ (3)
if the names in P are not unique **then**
sort the $(i,r) \in P$ by $(i \bmod 3, i \operatorname{div} 3)$ (4)
SA^{12}:= $DC3([c : (c,i) \in P])$ (5)
P:= $[(j+1, \mathrm{SA}^{12}[j]) : j \in [0, 2n/3)]$ (6)
sort P by the second component (7)
S_0:= $\langle (T[i], T[i+1], c', c'', i) :$ (8)
$i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P \rangle$
S_1:= $\langle (c, T[i], c', i) :$ (9)
$i \bmod 3 = 1, (c,i), (c', i+1) \in P \rangle$
S_2:= $\langle (c, T[i], T[i+1], c'', i) :$ (10)
$i \bmod 3 = 2, (c,i), (c'', i+2) \in P \rangle$
sort S_0 by components 1,3 (11)
sort S_1 and S_2 by component 1 (12)
S:= $merge(S_0, S_1, S_2)$ comparison function: (13)
$(t, t', c', c'', i) \in S_0 \leq (d, u, d', j) \in S_1$
$\Leftrightarrow (t, c') \leq (u, d')$
$(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2$
$\Leftrightarrow (t, t', c'') \leq (u, u', d'')$
$(c, t, c', i) \in S_1 \leq (d, u, u', d'', j) \in S_2$
$\Leftrightarrow c \leq d$
return [last component of $s : s \in S$] (14)

Figure 4. DC3 algorithm. Source: Ref. [19]

The basic idea of the DC3 algorithm is to divide the input into two parts. One is two thirds and another one is one third respectively. The larger part is sorted recursively and the result is utilized to sort the smaller third non-recursively. Lastly, the two parts are then merged to give the final suffix array.

Three steps in DC3 algorithm are explained as follows [19] and the pseudocode of DC3 algorithm is shown in figure4.

a) Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively;

b) Construct the suffix array of the remaining suffixes using the result of the first

step;

c) Combine the two suffix arrays into one.

Take text $'mississippi'$ as an example, its construction process by DC3 is shown in figure 5.

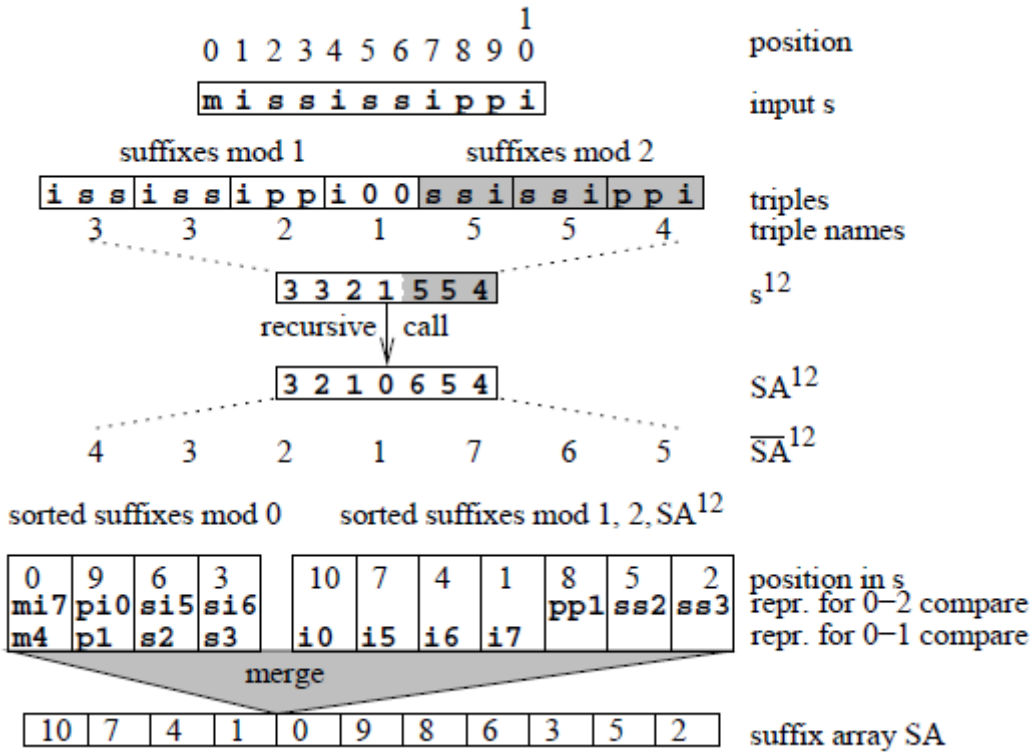


Figure 5. Example of DC3 algorithm. Source: Ref. [18]

Table 3 summarizes the comparison of time and space complexity for doubling algorithm and DC3 algorithm. It is easy to notice that the DC3 algorithm is more efficient. However, the implementation cost is comparatively higher than doubling algorithm. Several experimental tests have been conducted by Roman Dementiev etc. [19]. In their paper, doubling algorithm and DC3 algorithm have both been compared for three kinds of texts with some other variants of doubling algorithms in respect of running time. The outcome graphs are presented in figure 6 [19].

a) *[Random2]*: Two concatenated copies of a random string of length $n/2$;

b) *[Gutenberg]*: Freely available English texts.

c) *[Genome]*: The known pieces of human genome.

	Time complexity	Space complexity
Doubling algorithm	$O(n \log n)$	$O(n)$
DC3 algorithm	$O(n)$	$O(n)$

Table 3. Comparison of doubling algorithm and DC3 algorithm.

The suffix array permits a relatively simple searching algorithm. Due to all the suffixes are lexicographical ordered in array A, searching results of pattern $P_{1,m}$ occurred in text $T_{1,n}$ will lie on a consecutive interval of its suffix array. Take pattern $'issi'$ as instance, it locates in position 2 and 5.The indexes of them are indeed adjacent in its suffix array shown in figure 3.

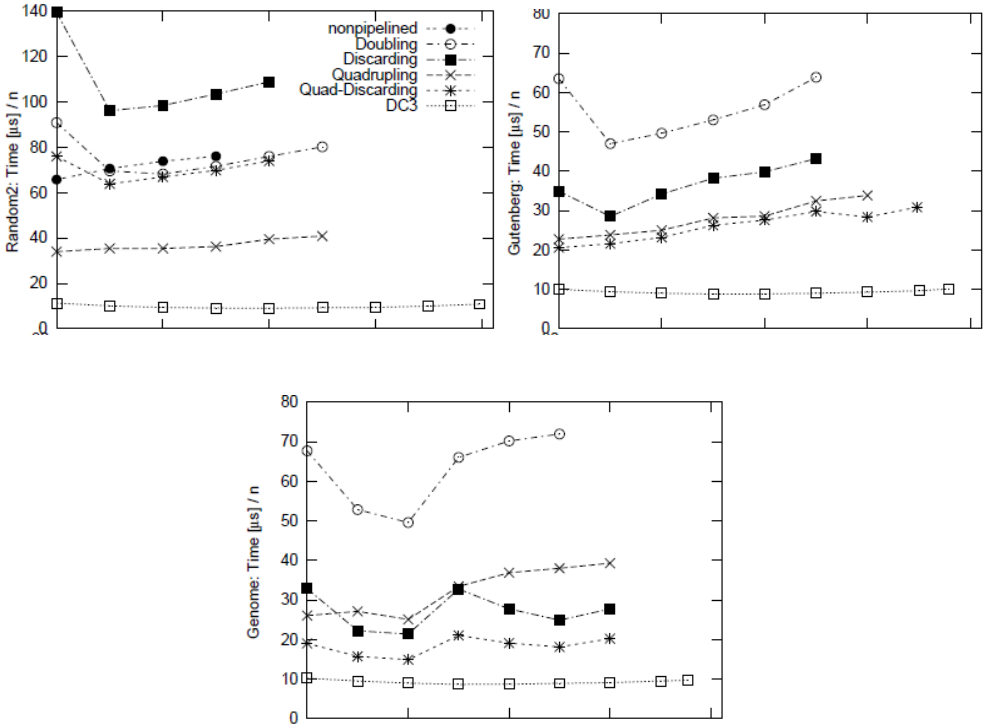


Figure 6. Comparison of doubling algorithm and DC3 algorithm. Source: Ref. [19]

Searching for a pattern $P_{1,m}$ in text $T_{1,n}$ involves two binary searchings on the basis of lexicographical comparison to find the start index sp and end index ep of the interval. Initially, we set $sp = 0$ and $ep = 0$. Every binary searching begins with comparing the pattern $P_{1,m}$ with the suffix in the middle of searching interval $(A[sp]to\ A[ep])$ whose index is denoted as $s = (sp + ep)/2$. If the pattern is less than the suffix, then move sp to $s + 1$, else move ep to $s - 1$. Keep s to indicate the middle index of current searching interval during the iteration. Repeat this process until the smallest index of interval is located so as the largest one

with resemble process [1, 13]. The pseudocode of this twice binary searching algorithm is shown in figure 7. The counting query can be answered by calculating the $occ = ep - sp + 1$. Locating query needs a traversal over the interval from $A[sp]$ to $A[ep]$

The true behavior of the algorithm depends on how many prefixes of $P_{1,m}$ occur in $T_{1,n}$. Two tricks are designed to improve this algorithm. In order to reduce the number of redundant character examinations, the concept of LCP is used to indicate the length of the longest prefix of the suffixes specified in suffix array [13].

Algorithm SASearch($P_{1,m}$,$A[1,n]$,$T_{1,n}$)
(1) $sp \leftarrow 1$; $st \leftarrow n$;
(2) **while** $sp < st$ **do**
(3) $s \leftarrow \lfloor (sp + st)/2 \rfloor$;
(4) **if** $P > T_{A[s],A[s]+m-1}$ **then** $sp \leftarrow s + 1$ **else** $st \leftarrow s$;
(5) $ep \leftarrow sp - 1$; $et \leftarrow n$;
(6) **while** $ep < et$ **do**
(7) $e \leftarrow \lceil (ep + et)/2 \rceil$;
(8) **if** $P = T_{A[e],A[e]+m-1}$ **then** $ep \leftarrow e$ **else** $et \leftarrow e - 1$;
(9) **return** (sp, ep);

Fig. 4. Algorithm to search for P in suffix array A over text T. T is assumed to be terminated by "$", but P is not. Accesses to T outside the range $[1, n]$ are assumed to return "$". From the returned data one can answer the counting query $ep - sp + 1$ or the locating query $A[i]$, for $sp \leq i \leq ep$.

Figure 7. Searching algorithm of suffix array. Source: Ref. [1]

As suffix array is not a self index, the space consumption of it has to involve the original text itself. The time to count, locate and display has already been summed up in following figure 8 [1]. The second part of this table refers to the situation with LCP.

Space in bits	$n \log n + n \log \sigma$
Time to count	$O(m \log n)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$
Space in bits	$2n \log n + n \log \sigma$
Time to count	$O(m + \log n)$
Time to locate	$O(1)$
Time to display ℓ chars	$O(\ell)$

Figure 8. Space and time tradeoff of suffix array. Source: Ref. [1]

3.2 Burrows-Wheeler Transformer

This part explains Burrows-Wheeler Transformer in details. It is an important part for compressed index.

Data compression is the process of reproducing information in a more compact form to save space in memory [20]. Data compression is either lossless or lossy.

Definition 2: *Lossless compression techniques*, as their name implies, involve no loss of information. If data have been losslessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for applications that cannot tolerate any difference between the original and reconstructed data [20].

Definition 3: *Lossy compression techniques* involve some loss of information, and data that have been compressed using lossy techniques generally cannot be recovered or reconstructed exactly. In return for accepting this distortion in the reconstruction, we can generally obtain much higher compression ratios than lossless compression [20].

Burrows-Wheeler Transformer is a lossless one which was published by Burrows and Wheeler in 1994 [21]. Consequently, the integrate compression scheme of Burrows-Wheeler Transformer includes two directions. Figure 9 concludes the major parts in both directions. It plays a core role in designing compressed indexes such as FM-index family. Previously, the most common data compression algorithm is presented by Lempel and Ziv [22]. Unlike their algorithm, the unit of Burrows-Wheeler Transformer procedure is block of text rather than sequential inputs of text. Burrows-Wheeler Transformer is, in nature, a reversible transformation from original strings to new strings that are convenient for compression [21]. It aims at gathering up the symbols followed by the same context so as to increasing the feasibility of searching single similar symbol [21].

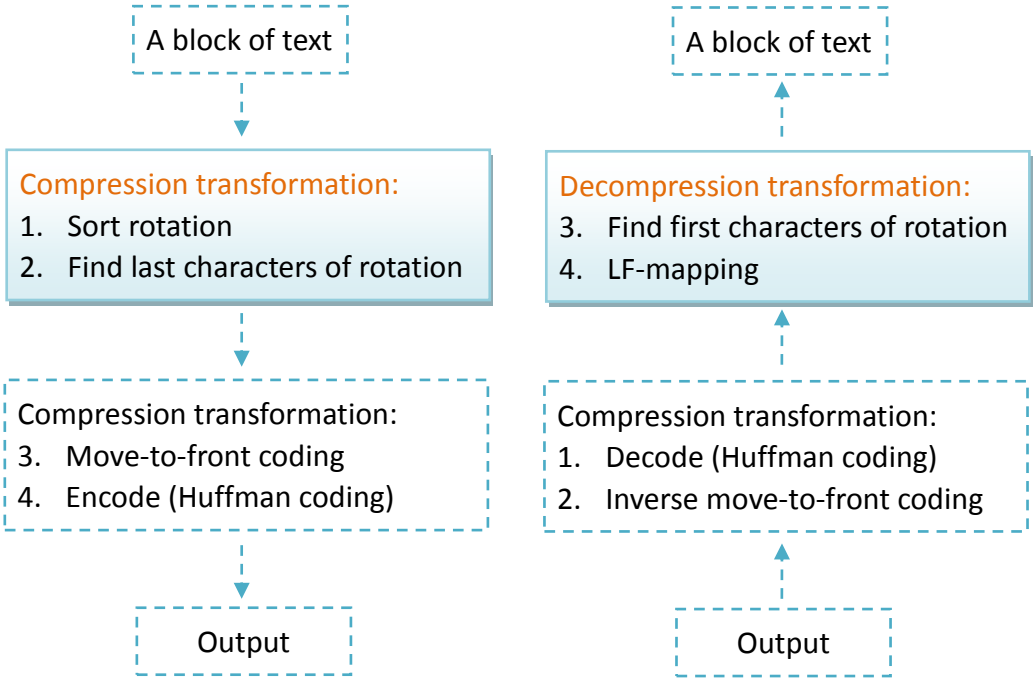


Figure 9. Burrows-Wheeler compression scheme.

The move-to-front coding and Huffman coding in second part are chosen in the implementation stage of this paper. They are both the basic techniques in the field of data compression which will be discussed in section 5.2 and 5.3. In this chapter, the compression transformation and decompression transformation will be the focus.

In the remaining two sections, the example will be the string $'mississippi'$ with the alphabet $X = \{'m', 'i', 's', 'p'\}$.

3.2.1 Compression transformation

Compression transformation consists of three steps as follows [21]:

a) *[Step1. Preprocessing]*: Add special symbol \$ to the end of $T = 'mississippi'$. This symbol is less than any other text character;

b) *[Step2. Sort rotation]*: Form a conceptual matrix $\mathcal{M}$ whose elements are the cyclic shifts of string $T\$$, sorted in lexicographical order. One-round cyclic shift of $T = t_1 t_2 t_3 \dots t_{n-1} t_n$ results in $T' = t_2 t_3 t_4 \dots t_n t_1$. Therefore, n-round cyclic shifts leads to n corresponding texts. See figure 10 step2. In this matrix, the original text T will be placed in the row with index I, numbering from zero. In the example shown in figure 10, the index I is 5;

c) [Step3. Find last character of rotations]: Construct the transformed text L by taking the characters in last column of matrix M and output the pair (L, I). In this example, $L = 'ipssm\$pissii'$ $I = 5$.

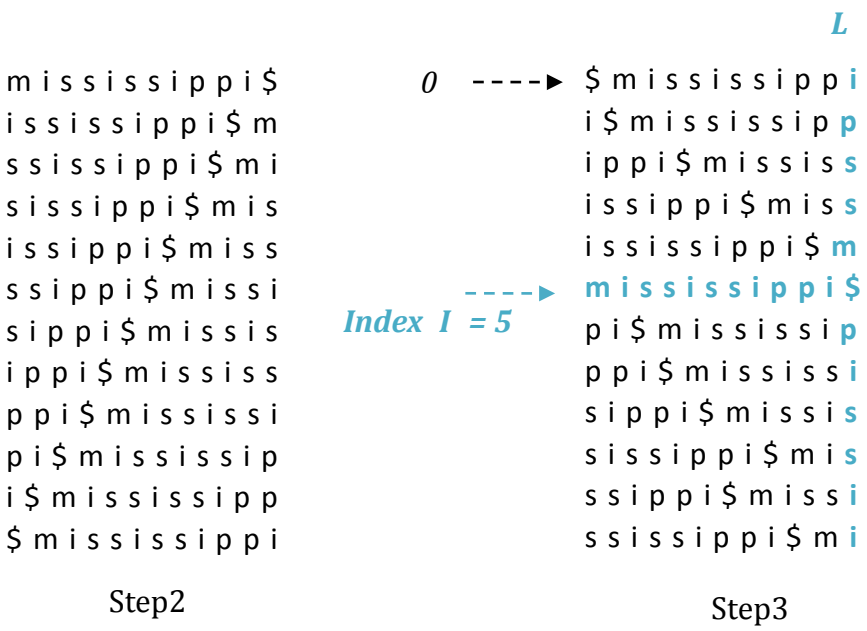


Figure 10. Burrows-Wheeler Transform of T= 'mississippi'.

3.2.2 Decompression transformation

This transformation is the inverse of previous one. It uses the output pair (L, I) to reconstruct its input T.

Decompression transformation consists of three steps as follows [21]:

a) *[Step1. Find first characters of rotations]*: With the input L, the first column F of the matrix in previous transformation can be calculated by sorting the L. It is obvious that the every column of the matrix is a permutation of the original text T. Hence, L and F are both permutations of S, and of each other. In this example, $F = '\$iiiimppssss'$.

b) *[Step2. Build list of predecessor characters (LF-mapping)]*: In this step, the decompressor have three inputs: F, L, I. Burrows and Wheeler defines a new matrix $\mathcal{M}'$ which is formed by rotating each row of previous matrix one character to the right. For each $i = 0,1, \dots, n-1$ and each $j = 0,1, \dots, n-1$, we have:

$$\mathcal{M}'[i,j] = \mathcal{M}[i,(j-1)\ mod\ n] \quad (1)$$

figure 11 illustrates the process.

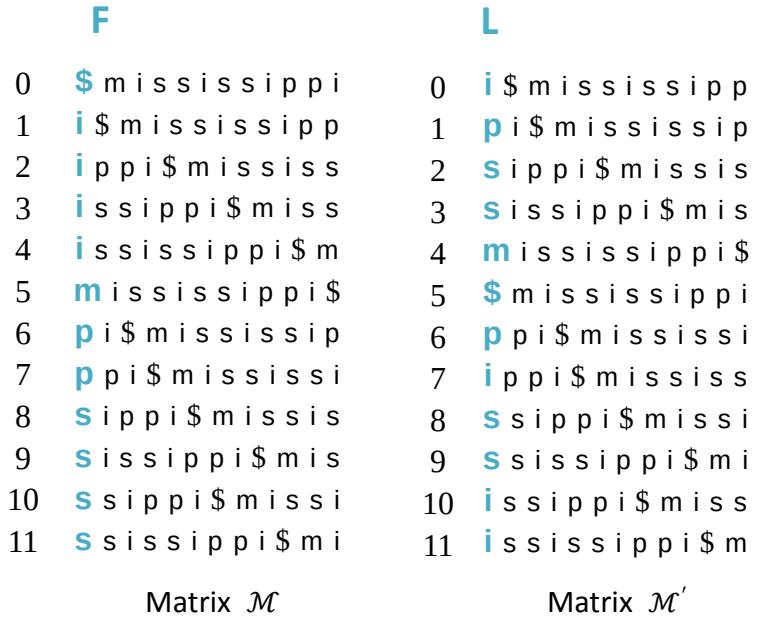


Figure 11. New matrix of Burrows-Wheeler Transformer.

Similar to the previous matrix $\mathcal{M}$, the rows in $\mathcal{M}'$ is also rotation of text T. The first column in $\mathcal{M}$ becomes the second row in $\mathcal{M}'$. As a result, the $\mathcal{M}'$ is sorted lexicographically according to the second character of each row. For any given ch, the rows in $\mathcal{M}$ that begin with ch appear in the same order as the rows in $\mathcal{M}'$ that begin with ch [21].

In this example, the rows begin with character $'s'$ in matrix $\mathcal{M}$ are $'ippi\$missis'$, $'sissippi\$mis'$, $'ssippi\$missi'$, $'ssissippi\$mi'$. It appears exactly the same order in $\mathcal{M}'$.

Using the F and L, the first columns of $\mathcal{M}$ and $\mathcal{M}'$, LF-mapping can be built to indicate the correspondence between the rows of two matrices. For each $i = 0,1,\dots,n-1$, the row i in $\mathcal{M}'$ corresponds to the row $T[i]$ in $\mathcal{M}$ [21].

$$F[T[i]] = L[i] \quad (2)$$

In this example, the T is $(1,6,8,9,5,0,7,2,10,11)$.

The implementation method of LF-mapping is demonstrated in the next section

separately since it is also a central idea for backward searching in chapter 4.

c) *[Step3. Form output S]*: In this step, the decompressor have three inputs: F, L, I, T. For each $i = 0,1, \dots, n-1$, the characters $L[i]$ and $F[i]$ are the last and first characters in the row i of $\mathcal{M}$. Since the matrix $\mathcal{M}'$ is a further rotation of matrix $\mathcal{M}$, the character $L[i]$ precedes the character $F[i]$ in T. With the help of LF-mapping T, it means $L[T[i]]$ cyclically precedes $L[i]$ in T [21].

The index I is refers to the index of row of original text T in matrix $\mathcal{M}$. Thus, the last character of T is $L[i]$. The original text T can be reconstructed by following equation:

$$\text{for each } i = 0,1, \dots, n-1, T[n-1-i] = L[T^i[I]] \quad (3)$$

where $T^0[x] = x$, and $T^{i+1}[x] = T[T^i[x]]$.

In this example, the original text $T = 'mississippi'$.

3.2.3 LF-mapping

LF-mapping can be constructed with two arrays: $C[0, \dots, \sigma]$ and $Occ[0, \dots, n]$. $C[ch]$ is the total number of instances in L of characters preceding character ch in the alphabet. $Occ[i]$ is the number of instances of character $L[i]$ in the prefix $L[0, \dots i-1]$ of L [21].

Given the arrays L, C, Occ, the array T for LF-mapping can be computed by:

$$\text{for each } i = 0,1, \dots, n-1, T[i] = Occ[i] + C[L[i]] \quad (4)$$

Burrows and Wheeler have also presented the implementation of T in two passes over as figure 12. One is a traversal on L and another is a traversal on alphabet.

```
for i := 0 to N-1 do
    P[i] := C[L[i]];
    C[L[i]] := C[L[i]] + 1
end;
```

Now $C[ch]$ is the number of instances in L of character ch. The value $P[i]$ is the number of instances of character $L[i]$ in the prefix $L[0, \ldots, i-1]$ of L.

```
sum := 0;
for ch := FIRST(alphabet) to LAST(alphabet) do
    sum := sum + C[ch];
    C[ch] := sum - C[ch];
end;
```

Now $C[ch]$ is the total number of instances in L of characters preceding ch in the alphabet.

Figure 12. Two traversals in LF-mapping. Source: Ref. [21]

3.2.4 Meaning of compression transformation

In the compression transformation stage, the output pair *(L,I)* is delivered to the next stage for coding.

To demonstrate the meaning of L, the effect on a single letter in a common word in a block of English text will first be explained. Take the word *'the'* as instance which is the most popular word in English article. It must be appeared numerous times in the selected text.

In the matrix M of the text, the beginning word *'he'* will be grouped together according to the lexicographical order. In these rows, the last character of a large proportion of them should be the character *'t'*. This concept can be also applied to all the words. Any localized region of the string L is likely to contain a large number of a few distinct characters. Equally, the probability that given character ch will occur at a given range in L is relatively high if it also appears near that range [21].

It is an impressive attribute for move-to-front coding in next stage. Move-to-front coding encodes an instance of character ch by counting the distinct characters occur since the next previous occurrence of ch. In such a situation, the encoding of L will yields low numbers. These low numbers, like chain reaction, will produces efficient result after Huffman coding [21].

Figure 13 is an example provided by Burrows and Wheeler to enhance the comprehension of this crucial theorem. Each line is the first character and last character of a rotation of a version of English documents. It is worthy to notice that

similar characters are grouped together in a closer region in the column of last characters.

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Figure 13. Example of sorted rotations. Source: Ref. [21]

3.2.5 Performance

Burrows and Wheeler have already concluded their compression scheme in their paper. The most interesting section is the analysis of size of block. They discovered that compression improves with increasing block size, even when the block size is quite large. Nevertheless, when the block size gets across a few tens of millions of characters, it has only a trivial effect. Figure 14 shows the comparison. Book1 in this figure comes from Calgary Compression Corpus and the Hector corpus is a modern English text [21].

Compared with traditional compression algorithm of Lempel and Ziv, Burrows-Wheeler Transformer achieves alike speed. Both of them have faster

speed in compression part than decompression part [21].

Block size (bytes)	bits/character	
	book1	Hector corpus
1k	4.34	4.35
4k	3.86	3.83
16k	3.43	3.39
64k	3.00	2.98
256k	2.68	2.65
750k	2.49	-
1M	-	2.43
4M	-	2.26
16M	-	2.13
64M	-	2.04
103M	-	2.01

Figure 14. The effect of varying block size on compression. Source: Ref. [21]

4. Compressed Indexes

As explained in the introduction, compressed indexes provide a doable alternative to classical indexes. They are parsimonious in space and efficient in query time. In this chapter, the first section will outline the k-th order empirical entropy and then the review of several compressed indexes is followed. Compressed indexes can be divided into three families: Compressed Suffix Array (CSA) family, LZ-indexes family and FM-index family. All of them are self-indexes which get rid of the original text after the construction of indexes. FM-index family is the central one. It is selected in the implementation stage of this paper.

4.1 The k-th order empirical entropy

The empirical entropy resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source) [23]. The pivotal property of empirical entropy is that it is defined point wise for any finite string $T_{1,n}$ and can be used to measure the space performance of compression algorithms without any assumption on the input distribution [23]. For this reason, understanding k-th order empirical is fundamental. In this paper, all the logarithms are taken to base 2.

a) Zero-order empirical entropy:

The zero-order empirical entropy of a text $T_{1,n}$ is defined as:

$$H_0(T) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log(\frac{n_i}{n}) \tag{5}$$

Notation σ is the size of alphabet and n_i is the number of occurrences of symbol i in $T_{1,n}$. Only those symbols that are occurred in $T_{1,n}$ are calculated in the formula. This value represents the output size of an ideal compressor [23].

b) K-th order empirical entropy:

The k-th order empirical entropy of text $T_{1,n}$ is defined as:

$$H_k(T) = \frac{1}{n} \sum_{w \in \sum^k}^{\sigma} |w_s| \, H_0(w_s) \tag{6}$$

Notation $\sum^k$ stands for the set of all sequences of length k over $\sum$. For any string $w \in \sum^k$, w is called a length-k context in T. For a given s, w_s stands for

the string that is formed by concatenating all the symbols following the occurrences of symbol w in T [16]. Take $T = 'mississippi'$ as example, $s = 'si'$ and the $w_s = 'sp'$. The value $|s|\, H_k(T)$ represents a lower bound to the compression it can achieve using codes which depend on the k. A calculation example of $H_1(T)$ where $T = 'mississippi'$ is displayed below:

For $T = 'mississippi', k = 1$ we have all the one-length strings that occur in the text over alphabet $\sum$ are $\{m, i, s, p\}$. Thus we can get $m_s = 'i'$, $i_s = 'ssp'$, $s_s = 'sisi'$, $p_s = 'pi'$. According to formula (5), we can calculate $H_0(i) = 0, H_0(ssp) = 0.918\ldots, H_0(sisi) = 1, H_0(pi) = 1$. Hence, the 1st order empirical entropy is

$$H_1(T) = \frac{1}{11} H_0(i) + \frac{3}{11} H_0(ssp) + \frac{4}{11} H_0(sisi) + \frac{2}{11} H_0(pi) = 0.796\ldots$$

Empirical entropy can be used as a criterion in compression algorithm. It provides a lower bound to the number of bits for a compressor that encodes each character considering only the context of k characters that follow it in T [1]. Several self-indexes claim that the space cost of their indexed text is in the function of the empirical entropy. The significant contribution to compressed index or self-index is that it allows to measure the index size by calculating the best k-th order compressor.

4.2 FM-index family

FM-index family contains five members. They are all based on the backward searching and Burrows Wheeler Transformer mentioned above. Before entering the explanation of specific FM-index, backward searching will first be presented in this chapter.

4.2.1 Backward searching

Backward searching is an alternative approach for searching operation on suffix arrays and has been proposed by Ferragina and Manzizi [24].

Given a pattern $P_{1,m}$ and the suffix array $A[1, n]$ constructed on the text $T_{1,n}$. For any $i = m, m-1, \ldots 1$, the interval $A[sp_i, ep_i]$ will always store all the text suffixes which are prefixed by $P_{i,m}$. Searching process is actually a recursive program shown below [25]:

a) [Initial step]: Set $i = m$ in the first step. Then the interval $A[sp_i, ep_i]$ equals to $A[sp_m, ep_m]$ consequently. It records all the suffixes of $T_{1,n}$ which are

prefixed by the last character of pattern $P_{1,m}$ [25].

b) [Inductive step]: In general case, the interval $A[sp_{i+1}, ep_{i+1}]$ is assumed to be already calculated. The suffixes prefixed by pattern $P_{i+1,m}$ have been located in suffix array. In current step, the task is to determine the next interval $A[sp_i, ep_i]$ for pattern $P_{i,m}$ from the given interval $A[sp_{i+1}, ep_{i+1}]$ and the next pattern symbol P_i [25].

   Loop this step until an empty interval is found or $A[sp_1, ep_1]$ has contained all the occurrences of specific pattern when i is reduced to 1. In the former situation, it indicates that the searching is unsuccessful while the second ending situation means the occurrences of pattern is $ep_1 - sp_1 + 1$ [25].

It goes without saying that the conversion from $A[sp_{i+1}, ep_{i+1}]$ to $A[sp_i, ep_i]$ is the core point in second step above. LF-mapping, described in chapter 3, is an appropriate solution. Figure 15 is the pseudocode for backward searching.

Algorithm FM-search(P, m, n, C, Occ)
(1) $sp \leftarrow 1$; $ep \leftarrow n$;
(2) **for** $i \leftarrow m$ **to** 1
(3) $\quad sp \leftarrow C(P_i) + Occ(P_i, sp - 1) + 1$;
(4) $\quad ep \leftarrow C(P_i) + Occ(P_i, ep)$;
(5) $\quad$ **if** $sp > ep$ **then return** $\emptyset$;
(6) $\quad i \leftarrow i - 1$;
(7) **return** $[sp, ep]$;

Fig. 21. Backward search algorithm to find the interval in $A[1, n]$ of the suffixes that start with $P_{1,m}$.

Figure 15. Backward searching algorithm. Source: Ref. [1]

4.2.2 Ferragina and Mazzini's implementation

The first implementation of backward searching was proposed by Ferragina and Manzini in 2000 [6]. The novelty of their approach resides in the careful combination of Burrows-Wheeler compression algorithm with the suffix array data structure [6].

They devised an indexing tool whose space and query time complexity are both sub linear. The word 'opportunistic' in the title of their paper refers to their space reduction which takes advantage of the compressibility of the input data. The query performance will not be influenced significantly at the same time.

The practical implementation procedure will be described in section 5.4. This section will only introduce two basic searching algorithms in FM-index: counting query and locating query.

Counting query is exactly a backward searching on the suffix array of text $T_{1,n}$. Locating query can be simplified to finding the starting position in $T_{1,n}$ of the suffix on the i-th row in the matrix of Burrows-Wheeler Transformer. For example, given $i = 3$, the third row in matrix of Burrows-Wheeler Transformer is *'ippi$mississ'* in figure 10. The starting position of suffix *'ippi'* in text *'mississippi'* is 8. As a result, all the locations can be found by occ times locating queries for every $i = sp, sp + 1, \dots, ep$.

For the sake of fast locating process, Ferragina and Manzini suggested a marking strategy for locating query. The basic idea is to logically mark a suitable set of rows in matrix $\mathcal{M}$. For these marked rows, their positions in text $T_{1,n}$ are stored explicitly. In each iteration of locating query for i, if i is not marked then scans the text $T_{1,n}$ backwards using the LF-mapping until a marked i' is satisfied. In this case, the result is $A[i'] + t$ where t is the number of backward steps used to find such a i'. Figure 16 is a pseudocode for locating query [25].

Algorithm FM-locate(i)
$i' \leftarrow i, t \leftarrow 0;$
while $A[i']$ is not explicitly stored **do**
    $i' \leftarrow LF(i');$
    $t \leftarrow t + 1;$
return $A[i'] + t;$

Figure 16. Locating algorithm in FM-index. Source: Ref. [25]

4.3 Related work and comparison

There exist other indexing schemes in FM-index family. Apart from FM-index in last section, Grabowski, Mäkinen and Navarro brought forward HUFF-FMI in 2004. Another group of indexes in this family make the use of wavelet trees. The exponent of this idea is WT-FMI by Sadakane in 2002 when the wavelet trees did not appeared yet. Later, Ferragina suggested an AF-FMI in 2004 and the RL-FMI is subsequently proposed by Mäkinen and Navarro in 2005. Table 4 is the conclusion of these indexes in the respect of their entropy term, time to count and time to locate. ϵ is a constant between 0 and 1.

Index	Entropy term	Time to count	Time to locate
FMI	$5nH_k$	$O(m)$	$O(\sigma \log^{1+\epsilon} n)$
HUFF-FMI	$2nH_0$	$O(mH_0)$	$O(\epsilon(H_0+1)\log n)$
WT-FMI	nH_0	$O(m)$	$O(\log\sigma \log^{1+\epsilon} n)$
RL-FMI	$nH_k \log\sigma$	$O(m)$	$O(\log\sigma \log^{1+\epsilon} n)$
AF-FMI	nH_k	$O(m)$	$O(\log\sigma \log^{1+\epsilon} n)$

Table 4. Comparison of compressed suffix array family.

Compressed suffix array family is not initially self indexes. The algorithms in this family make use of classical index, suffix array data structure, in an abstract optimization way. The searching algorithm is as of old as in figure 7. The first two cases in this family are called MAK-CSA of Mäkinen and GV-CSA of Grossi and Vitter. Both of their brainchild came simultaneously and independently during 2000. They utilized the property of self-repetition in suffix array which captures the text regularities [1]. Thereafter, much effort has been devoted into turning these compressed indexes into self-index. The underlying idea of these more advanced algorithms is to replace T by an abstract data type that fives access to any substring of it. Three cases are designed in this stage: Sad-CSA of Sadakane, GGV-CSA of Grossi, Gupta and Vitter and MN-CCSA of Mäkin4en and Navarro [1]. Table 5 is a summary of entropy term, time to count and time to locate for compressed suffix array family [1]. ϵ is a constant between 0 and 1.

Index	Entropy term	Time to count	Time to locate
GV-CSA	nH_0	$O(mlog^2 n)$	$O(log^2 n/(\log\log n)^2)$
MAK-CSA	$2nH_k \log n$	$O(m\log n + poly\log n)$	$O((\log\log n)^2)$
SAD-CSA	nH_0	$O(m\log n)$	$O(\log\epsilon n)$
MN-CCSA	$nH_k \log n$	$O(m\log n)$	$O(\epsilon\log n)$
GGV-CSA	nH_k	$O(m\log\sigma + poly\log n)$	$O(\log\log n + log\sigma)$

Table 5. Comparison of compressed suffix array family.

Compressed suffix array family and FM-index family are both on the basis of suffix array. The second family, LZ-indexes family, is based on a Lempel-Ziv partitioning of the text. The key idea of this compression is identifying repeated text substrings and replacing repetitions by pointers to their former occurrences in T. The members of this family includes KU-LZI of Kärkkäinen and Ukkonen, FM-LZI of

Ferragina and Manzini and NAV-LZI of Navarro [1]. Table 6 is a summary of entropy term, time to count and time to locate for compressed suffix array family [1].

Index	Entropy term	Time to count	Time to locate
KU-LZI	nH_k	$O(mlog^2 n)$	$O(log^2 n/(\log\log n)^2)$
FM-LZI	$nH_k\ log\ \tau\ n$	$O(m\ log\ n + poly\ log\ n)$	$O((\log\log n)^2)$
NAV-LZI	nH_0	$O(m\log n)$	$O(\log \epsilon\ n)$

Table 6. Comparison of LZ-index family.

5. Implementation Methodologies

5.1 Doubling algorithm

Doubling algorithm is implemented in the stage of constructing the suffix array for text $T_{1,n}$. Suffix array can be built by directly sorting all the suffixes of corresponding text $T_{1,n}$. However, it has low efficiency as all the suffixes are treated independently. The relationships between each of them are ignored. Doubling algorithm will not sort the suffixes completely in a single stage. It performs $\lceil log(n+1) \rceil$ stages instead.

5.1.1 Basic idea

Three definitions about suffixes will firstly be presented in this section. For a string u, let u^k be the prefix of u consisting the first k symbols if u contains more than k symbols and u otherwise [17].

$$u^k = \begin{cases} u[1, \dots, k], & k \leq len(u) \\ u, & k > len(u) \end{cases} \tag{7}$$

Second definition is the relation $<_h$. It refers to the lexicographical order of all the u^h of every suffix. Given two suffixes u and v, we have:

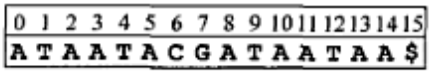
$$\begin{cases} u <_h v & if\ u^h < v^h \\ u =_h v & if\ u^h = v^h \\ u >_h v & if\ u^h > v^h \end{cases} \tag{8}$$

The last definition is called approximate suffix array SA_h. SA_h is an array such that $SA_h[i] = k$ if and only if the i-th suffix in the $<_h$ order starts at the position k in the text $T_{1,n}$ [27]. Figure 17 shows an example of approximate suffix array.

In the first stage of doubling algorithm, the suffixes are sorted according to their first symbol. SA_1 is built as shown in figure 17. In this stage, nine suffixes have the same first symbol $'A'$ and four suffixes shared the same first symbol $'T'$. Manber and Myers called them bucket which contains the group of suffixes with same symbols in each stage of approximate suffix array.

Then, inductively, each stage further partitions the buckets by sorting according to twice the number of symbols. For ease of reading, we denote stages 1, 2, 4, 8,

etc., to indicate the length of sorted prefix for each suffixes [17]. For instance, stage2 in figure 17 sorted the prefix of length 2 for each suffixes. The bucket of $'A'$ in previous stage has been partitioned into four sub-buckets of $'A\$'$, $'AA'$, $'AC'$, $'AT'$. Those four buckets are sorted according to the lexicographical order of the second symbol [27].



SA_1	$S[SA_1[i]]$	SA_2	$S[SA_2[i], SA_2[i]+1]$	SA_4	$S[SA_4[i], SA_4[i]+3]$	SA_8	$S[SA_8[i], SA_8[i]+7]$	SA
0 15	$	0 15		0 15		0 15		0 15
1 0	A	1 14	A $	1 14		1 14		1 14
2 2	A	2 2	A A	2 13	A A $	2 13		2 13
3 3	A	3 10	A A	3 2	A A T A	3 10	A A T A A $	3 10
4 5	A	4 13	A A	4 10	A A T A	4 2	A A T A C G A T	4 2
5 8	A	5 5	A C	5 5		5 5		5 5
6 10	A	6 0	A T	6 0	A T A A	6 11	A T A A $	6 11
7 11	A	7 3	A T	7 8	A T A A	7 8	A T A A T A A $	7 8
8 13	A	8 8	A T	8 11	A T A A	8 0	A T A A T A C G	8 0
9 14	A	9 11	A T	9 3	A T A C	9 3		9 3
10 6	C	10 6		10 6		10 6		10 6
11 7	G	11 7		11 7		11 7		11 7
12 1	T	12 1	T A	12 12	T A A $	12 12		12 12
13 4	T	13 4	T A	13 1	T A A T	13 9	T A A T A A $	13 9
14 9	T	14 9	T A	14 9	T A A T	14 1	T A A T A C G A	14 1
15 12	T	15 12	T A	15 4	T A C G	15 4		15 4

Figure 17. Approximate suffix array. Source: Ref. [27]

The key concept in this algorithm is how to compute SA_{2h} given the SA_h. In general, if SA_h is available, sorting prefixes of length $2h$ for suffixes can be calculated by using the former half of each prefix as the primary key and the latter half as the secondary key [27]. To explain this idea, we define $Suffix(i)$ and $Suffix(j)$ as the i-th and j-th suffix in text $T_{1,n}$. The $2h$-order can be obtained by two h-order comparisons as shown in figure 18. The computation of $2h$-order can be converted into the two comparisons of h-order. One is $Suffix(i)$ and $Suffix(j)$ in red region. Another is $Suffix(i)$ and $Suffix(j)$ in green region.

Final suffix array can be formed during constructions of $SA_{2^0}, SA_{2^1}, SA_{2^2}, \ldots, SA_{2^k}$ until 2^k minimizes$n < 2^k$. In the worst case, the number of approximate suffix arrays constructions is at most $1 + [log\, n]$

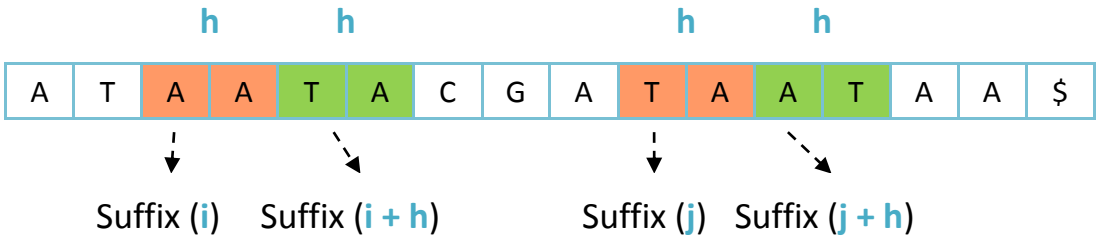


Figure 18. Doubling technique.

5.1.2 Practical implementation

In practice, two arrays are demanded in the process of programming. One is SA_h which will be updated during the interations. Another one is $Rank_h$ which stores the rank of the i-th suffix of text $T_{1,n}$. Essentially, they are the inverse of each other. Namely, we have:

$$Rank_h[SA_h[i]] = i \tag{9}$$

Consequently, $Rank_h$ can be built on the basis of SA_h easily according to the formula above. The time complexity is $O(n)$ as it only need a traversal on the SA_h. Given SA_h and $Rank_h$, SA_{2h} can be obtained by two comparisons on two suffixes using the $Rank_h$ in constant time.

$$\begin{aligned}
&if \quad Rank_h[i] < Rank_h[j] \\
&then \; Suffix(i) <_h Suffix(j) \\
&if \quad Rank_h[i] = Rank_h[j] \; and \; Rank_h[i+h] < Rank_h[j+h] \\
&then \; Suffix(i) <_h Suffix(j) \\
&else \; Suffix(i) <_h Suffix(j)
\end{aligned}$$

The flow graph of this algorithm is illustrated in the figure 19.

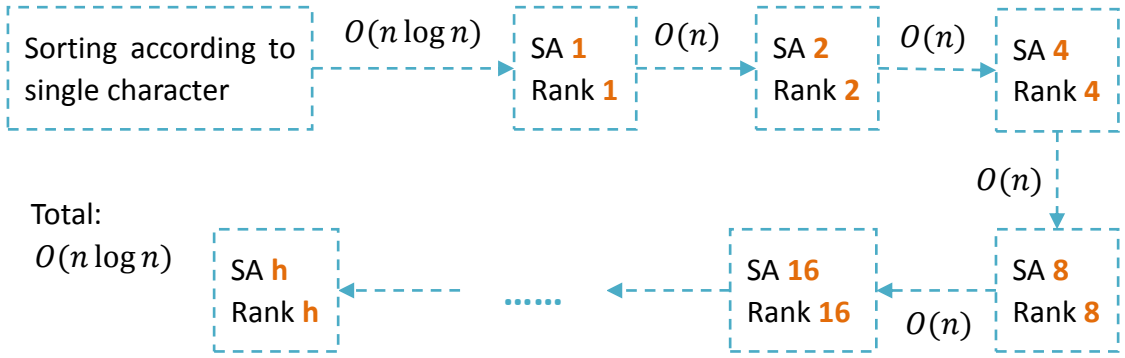


Figure 19. Flow of doubling algorithm.

5.2 Move-to-front coding

Move-to-front coding is an important part in Burrows-Wheeler Transformer scheme which is described in chapter 3.2. It is the third step in compression direction and the second step in the decompression direction as shown in figure 8. In this section, it will be discussed in two aspects: Move-to-front encoding and Move-to-front decoding.

5.2.1 Move-to-front encoding

This algorithm encodes the output pair (L, I) of compression transformation, where L is a string of length n and I is an index.

The idea is invented by Bentley in 1986 [28]. The original invention is to maintain the alphabet A of symbols as a list where frequently occurring symbols are located near the front [29]. Later, some improvements are added. The most basic one is that the encoded symbol will be moved to the front of list A [29]. For example, if the $A = \{'t', 'h', 'e', 's'\}$, then the symbol $'e'$ will be encoded as 2 since there are two symbols $'t'$ and $'h'$ before it. Lastly, the alphabet A will be formed as $\{'e', 't', 'h', 's'\}$ by moving the symbol $'e'$ to the front. This attempts to ensure that frequently used words appear near the front of the list [28]. It is called 'locally adaptive' in the paper of Bentley as it adapts itself to the frequencies of symbols in local areas of the input stream [29].

This method will be appealing if the input stream contains concentrations of identical symbols which are called 'concentration property'. It is worthy to mention that this is just the feature of L after Burrows-Wheeler Transformer. Another fascinating property relies on the small average number of the result after coding. This is a great advantage for Huffman coding in C language [29]. To explain these

two properties, four examples with same alphabet

$$A = \{'a', 'b', 'c', 'd', 'm', 'n', 'o', 'p'\}$$

for two different input streams are given below in figure 20:

a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0	a	abcdmnop	0
b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1	b	abcdmnop	1
c	bacdmnop	2	c	abcdmnop	2	c	bacdmnop	2	c	abcdmnop	2
d	cbadmnop	3	d	abcdmnop	3	d	cbadmnop	3	d	abcdmnop	3
d	dcbamnop	0	d	abcdmnop	3	m	dcbamnop	4	m	abcdmnop	4
c	dcbamnop	1	c	abcdmnop	2	n	mdcbanop	5	n	abcdmnop	5
b	cdbamnop	2	b	abcdmnop	1	o	nmdcbaop	6	o	abcdmnop	6
a	bcdamnop	3	a	abcdmnop	0	p	onmdcbap	7	p	abcdmnop	7
m	abcdmnop	4	m	abcdmnop	4	a	ponmdcba	7	a	abcdmnop	0
n	mabcdnop	5	n	abcdmnop	5	b	aponmdcb	7	b	abcdmnop	1
o	nmabcdop	6	o	abcdmnop	6	c	baponmdc	7	c	abcdmnop	2
p	onmabcdp	7	p	abcdmnop	7	d	cbaponmd	7	d	abcdmnop	3
p	ponmabcd	0	p	abcdmnop	7	m	dcbaponm	7	m	abcdmnop	4
o	ponmabcd	1	o	abcdmnop	6	n	mdcbapon	7	n	abcdmnop	5
n	opnmabcd	2	n	abcdmnop	5	o	nmdcbapo	7	o	abcdmnop	6
m	nopmabcd	3	m	abcdmnop	4	p	onmdcbap	7	p	abcdmnop	7
	mnopabcd						ponmdcba				
	(a)			(b)			(c)			(d)	

Figure 20. Encoding with and without move-to-front. Source: Ref. [29]

In this figure, (a) and (b) are the encoding results with and without move-to-front for input stream $'abcddcbamnopponm'$ which has two concentrations of $'abcd'$ and $'mnop'$. Both two encoding result resides in the range of 0-7. Nevertheless, the average value with move-to-front is 2.5 while the average value without move-to-front is 3.5 [29].

The remaining two examples: (c) and (d) are the encoding results with and without move-to-front for input stream $'abcdmnopabcdmnop'$ which does not have concentrations of identical symbols. As a result, it produces a worst coding. The average value for (c) is 5.25 while the average value for (d) is 3.5. It is obvious that move-to-front performs inefficiently [29].

In practical implementation of Burrows Wheeler scheme, a vector of integers $R[0], R[1], \dots, R[n-1]$ is defined to store the move-to-front codes for the symbols$L[0], L[1], \dots, L[n-1]$. An array $Y[0], Y[1], \dots, Y[\sigma - 1]$ is defined to store the alphabet after each iteration. Two steps are listed as follows [21]:

a) Initialize a list Y of symbols to contain each character in the alphabet exactly once.

b) For each $i = 0, 1, \ldots, n-1$ in turn, set $R[i]$ to the number of characters preceding symbol $L[i]$ in the array Y, then move symbol $L[i]$ to the front of Y.

5.2.2 Move-to-front decoding

Decoding of move-to-front has the same principle with encoding. This section will only present the practical implementation part in Burrows Wheeler compression scheme. The task of decoding is to calculate the string L of n symbols, given the move-to-front codes $R[0], R[1], \ldots, R[n-1]$. Two steps are listed below [21]:

a) Initialize a list Y of symbols to contain each character in the alphabet in the same order as in encoding step;

b) For each $i = 0, 1, \ldots, n-1$ in turn, set $L[i]$ to be the symbol at position $R[i]$ in list Y (numbering from 0).

5.3 Huffman coding

Huffman coding is also a vital part in Burrows-Wheeler Transformer scheme which is described in chapter 3.2. It is the forth step in compression direction and the last step in the decompression direction as shown in figure 8. This section is organized in two parts: Huffman encoding and Huffman decoding.

Huffman coding is a popular algorithm for data compression which brought intensive research into data compression. It has wide applications in various fields. Huffman first proposed this method in 1952 [30]. The idea of Huffman coding is similar to Shannon-Fano coding. They both manipulate the input streaming according to the probability of the symbols. Huffman coding achieves better performance when the probability of the symbols is negative powers of 2. Unlike the up-to-bottom direction of Shannon-Fano coding, Huffman coding constructs the Huffman tree upwards [29].

Commonly, there are two kinds of codes: fixed-length code and variable-length code respectively. A variable-length code can be more efficient than a fixed-length code. It encodes the symbol according to the probability of the symbol. Frequent symbols result in short codewords and infrequent symbols result in long codewords. Huffman coding, as a variable-length coding, act as an optimal coding scheme which saves the space significantly [31].

Now, consider two variable-length codes for a given input stream $'a_1 a_2 a_3 a_4'$ as

shown in figure 21. Code1 becomes relatively weak when facing the decoding process. For example, when the code is '101001', the only symbol start with '1' is a_1 with no doubt. However, it becomes uncertain for the second code '0' as the codes for $a_2\ a_3\ a_4$ all start with '0'. Continue reading to the third code '1' will create ambiguous interpretation. It can be decoded as '1|010|01' which is the code for $'a_1\ a_3\ a_2'$ or '1|010|01' which is the code for $'a_1\ a_2\ a_4'$. Code 2 will not cause this confusion as it has prefix property. This property means that once a code is assigned to a symbol, no other codes should start with that one [29].

Symbol	Prob.	Code1	Code2
a_1	.49	1	1
a_2	.25	01	01
a_3	.25	010	000
a_4	.01	001	001

Figure 21. Variable-size of codes. Source: Ref. [29]

With the definition of prefix property, the requirements for designing the variable-size codes can be concluded as follows:

a) Assign short codes to the more frequent symbols;

b) Obey the prefix property.

The methods like Shannon-Fano coding and Huffman coding which satisfies the prefix property are called prefix codes. It is desirable because they simplify the decoding process.

5.3.1 Huffman encoding

This section will describe the Huffman encoding at an implementation level. The basic algorithm will be listed as follows [29]:

a) Scan the input stream to be compressed and calculate the occurrence of all symbols;
b) Sort the symbols based on number of occurrences in input stream in descending order;
c) Build Huffman tree based on the list;
d) Perform a traversal of tree to determine all code words;
e) Scan input stream again and create new file using the Huffman codes.

The key step in this algorithm is the construction of Huffman tree. It begins with a

symbol at every leaf in the tree. Then, inductively, the two symbols with smallest probabilities in the list are selected to be added to the top of the partial tree. They are also deleted from the list and replaced with an auxiliary symbol representing both of them. It ends when the list has only one auxiliary symbol which refers to the entire alphabet. The pseudocode of this process is shown as below:

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i 1 to n ├ 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN (Q)
6          right[z] ← y ← EXTRACT-MIN (Q)
7          f [z] ← f [x] + f [y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)    ▷Return the root of the tree.
```

Figure 22. Huffman coding algorithm. Source: Ref. [31]

An example is followed below. Given the alphabet $'a_1\ a_2\ a_3\ a_4\ a_5'$ with probabilies of $0.4, 0.2, 0.2, 0.1, 0.1$. The Huffman tree of this example is placed as figure 23. These five symbols are added to the tree in following order [29]:

a) a_4 is combined with a_5. They are both the symbols with smallest probability 0.1. The probability of the new symbol a_{45} is $0.1 + 0.1 = 0.2$.

b) There are now four symbols. Three of them have the probability0.2. Two symbols are selected arbitrarily. Combine the a_{45} and a_3 to form a new symbol a_{345} with probability$0.2 + 0.2 = 0.4$.

c) Similarly, a_2 with probability of 0.2 and a_{345} with probability of 0.4 are selected in the left three symbols. It is also replaced by the new symbola_{2345}.

d) Lastly, the only two symbols are added to the current Huffman tree. The final symbol is a_{12345} is the root indeed with probability of $0.4 + 0.6 = 1.0$

The Huffman tree is complete after four steps. To determine the codes for all the symbols. 0 is assigned to the every upper branch while 1 is assigned to the every bottom branch. The code for each symbol can be obtained by following the Huffman tree from root to corresponding symbol. The concentration of symbols on the branches of this route is actually the code [29].

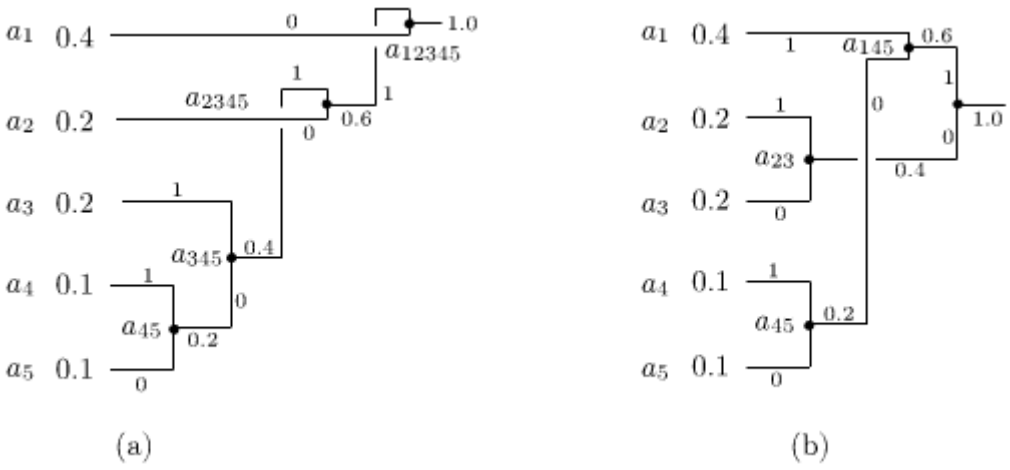


Figure 23. Huffman tree. Source: Ref. [29]

Many variants derived from the pioneer scheme. Moreover, the Huffman coding on the result of move-to-front is finally the output of Burrows Wheeler compression scheme.

5.3.2 Huffman decoding

Decoding is not complicated. Given the compressed stream and the Huffman tree for the alphabet. The process of decoding starts at the first bit of the compressed stream and the root of the Huffman tree. If it is zero, follow bottom edge of the node. Otherwise, follow the upper edge of the node. Read the next bit of the compressed stream in the same way until the leaf is reached. When the decoder gets to a leaf, the corresponding symbol is found. Then the process starts again from the next bit of the compressed stream and the root of the Huffman tree [29].

An example is given below. The alphabet contains five symbols and its Huffman tree is shown in the figure 24.

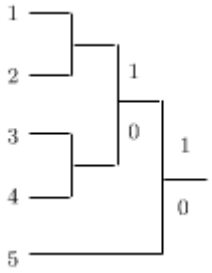


Figure 24. Huffman tree for decoding. Source: Ref. [29]

The compressed stream is $'1001100111'$. Its original input stream is $'a_4a_2a_5a_1'$. The decoder starts at the first bit $'1'$, then follow the upper edge which is labeled in $'1'$. The next bit $'0'$ lead to the bottom edge from the current node. Third bit $'0'$ will follow the same route and the leaf a_4 will be reached. Then it turns to the root again until all the codes are converted to the input stream.

5.4 FM-index

From the section 4.2.2, we can conclude that the efficiency of both counting procedure and locating procedure relies on the computation of value $\mathrm{Occ}(\mathrm{P_i}, \mathrm{q})$ which stands for the number of occurrences of the character $\mathrm{P_i}$ in the prefix $L[1, q]$.

In the [6], the FM-index implementation compressed permutated string L by splitting it into blocks and using independent zero-order compression on each block [25]. String L is broke into superbuckets of size l_{sb}. Each superbucket is also partitioned into buckets of size l_{sb}. For each super bucket, a table stores the number of occurrences of each symbol $c \in \sum$ in string L. Namely, for bucket, store the number of occurrences of each symbol in its super buckets. This auxiliary information, searching for a given symbol became the process of compute the number of occurrences from beginning of L up to the beginning of a bucket. Additionally, it is the buckets that are compressed rather than string L to compute the number of occurrences inside a bucket efficiently [26].

Until now, the structure of the FM-index is described and can be induced as follow:

a) Super buckets section: For each super bucket, it holds the number of occurrences of every symbol in the previous super buckets;

b) Bucket directory: It is a mapping for each compressed bucket to its starting position in the body of the FM-index;

c) Body of the FM-index: main body of the FM-index is the compressed image of each bucket. The compressed image of each bucket includes a header which stores the number of occurrences of each symbol in the super bucket [26].

Computation of $Occ(P_i, q)$ involves four steps [26]:

a) Using the bucket directory to locate the starting position of the bucket B_i that contains the $L[k]$;

b) Decompress the B_i and count the number of the occurrences of P_i from the beginning of the bucket up to $L[k]$;

c) Get the number of occurrences of P_i from the header since the beginning of the superbucket.

d) Get the number of occurrences of P_i from the table since the beginning of the L.

With the value $Occ(P_i, q)$ both counting query and locating query can be answered naturally.

Besides, it is worthy to append a bitmap to the existing structure of FM-index. In practice, it is hard to detect the different symbols in a small portion of the string L. For this reason, Paola suggested a bitmap for each bucket which stores the symbols occurring in it. The contribution of this bitmap is a pre-check for whether the symbol is in this bucket and accelerating the computation of $Occ(P_i, q)$.

6. Experimental Results

6.1 Experimental texts

In implementation stage of this paper, all the text collections come from Pizza&Chili corpus. They have offered various types of texts collections to be indexed and experimented. The requirement for text collections are listed as follows [32]:

a) The text collections should cover the representative applications of full-text indexing to prove its outstanding performance on these fields. In addition, they should be freely available for access;

b) For the sake of time consumption, the size of these text collections is preferred to be small;

c) In order to achieve apparent experimental result, the size of these text collections should be large enough for indexing and compression.

This paper has selected five kinds of text collections from the corpus for experiment. Table 7 is the list and explanation for these text collections.

DNA (gene DNA sequences)	This file is a sequence of newline-separated gene DNA sequences. These bare DNA code comes from Gutenberg project. Each of the 4 bases is coded as an uppercase letter A, G, C, and T and there still exist several special characters [32].
ENGLISH (English texts)	This file is the concatenation of English text files selected from Gutenberg project too. The headers have been deleted and only the real text is left [32].
PROTEIN (protein sequences)	This file is a sequence of newline-separated protein sequences which is similar to the DNA sequences. They come from Swissprot database. Each of the 20 amino acids is coded as one uppercase letter [32].
SOURCES (source program code)	This file is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the Linux-2.6.11.6 and gcc-4.0.0 distribution [32].
XML (structured text)	This file is an XML that provides bibliographic information on major computer science journals and proceedings. They all

come from dblp.uni-trier.de [32].

Table 7. Text collections for experiment.

All the experiments are run on a computer equipped with a 2.27 GHz Intel Core M 43 processor with 256Kb L2 cache, 2 GB RAM and a 200 GB SCSI hard disk. The operating system is Windows 7 professional. The size of alphabet is 99. All the code is written in C programming language.

The goal of the experiment in this paper is verifying the space and time performance of FM-index. The time of searching can be compared with sequential searching. The performance of space consumption can be compared with other common compression schemes.

6.2 Suffix Array analysis

As mentioned in section5.1, the suffix array is constructed by doubling algorithm. Each text collections contain 5 txt files. Their sizes are 1K, 5K, 10K, 20K, and 40K. Totally, there are 25 files to be experimented for doubling algorithm. The running time of them are tested as shown in table 8.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
1K	109	94	93	94	94
5K	2917	2668	2964	2854	2839
10K	12979	11996	12839	12246	11934
20K	53415	54289	54211	50809	51340
40K	232340	234392	234251	207949	218776

Table 8. Running time of suffix array construction (unit: ms).

We can see from the table above that the performance has little difference between various types of files. The behavior varies by the size of input file itself.

A performance comparison of counting query between binary searching on suffix array and sequential searching is conducted. The running time of counting for sequential searching on a 200Mb file is illustrated in table 9. Both the counting query and locating query are executed on the 20K file for five kinds of text collections.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML

Running time	11014	6973	6957	6193	6770
Occ	62278983	10326154	7895450	9112095	14302844
Pattern	ACCAT	teach	IPEAQ	const	title

Table 9. Running time for sequential searching (unit: ms).

A similar trial has been tested on suffix array for binary searching with exactly. The statistics are shown below in table 10.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
Running time	673	609	453	514	580
Occ	62278983	10326154	7895450	9112095	14302844
Pattern	ACCAT	teach	IPEAQ	const	title

Table 10. Running time for binary searching on suffix array (unit: ms).

Backward searching is also tested in this paper. The tested files are same as the previous two searching algorithms.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
Running time	550	548	492	589	573
Occ	62278983	10326154	7895450	9112095	14302844
Pattern	ACCAT	teach	IPEAQ	const	title

Table 11. Running time for backward searching on suffix array (unit: ms).

From the two tables, we can summarize that suffix array is especially suitable for the large size text collections. Its advantages mainly rely on the saving for memeory space.

6.3 Burrows-Wheeler Transformer analysis

To analyze the performance of Burrows-Wheeler compression scheme, compression ratio is the major criteria. Compression ratio can be defined as below:

$$Compression\ ratio = \frac{compressed\ \ size}{uncompressed\ \ size} \tag{10}$$

Three aspects of the performance for Burrows-Wheeler Transformer will be tested in experimental stage. Firstly, the compression ratios for different types of text colletions are presented in table 12 and table 13.

The first table is formed by compressing each type of text collection in same size. The second table, on the contrary, is tested by same type of text collection (DNA) in different sizes.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
Uncompressed size	10K	10K	10K	10K	10K
Compressed size	2.75K	4.25K	4.50K	3.12K	3.25K
Compression ratio	27.5%	42.5%	45.0%	31.2%	32.5%
Average			35.7%		

Table 12. Compression ratio for different types of text collections.

	10K	20K	40K	80K
Compressed size	2.75K	5.5K	11K	22K
Compression ratio	27.5%	27.5%	27.5%	27.5%
Average		27.5%		

Table 13. Compression ratio for DNA in different sizes.

It is easy to figure out that the compression ratio is equal for the same type of text colletions. However, the performance varies from different types of text collections. The average compression ratio for Burrows-Wheeler compression scheme is 35.7%. Apart from the analysis of its own behavior, the comparison with other compression schemes is also an interesting point. Table 14 shows the compression ratio for different text collections by various common compression schemes in 10K size.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML	Average
WinZip	27.2%	38.1%	38.3%	25.4%	24.9%	30.78%
7-Zip	29.3%	43.9%	36.6%	27.6%	27.3%	32.94%
WinRAR	30.1%	44.8%	41.2%	27.9%	27.3%	34.26%
BWT	27.5%	42.5%	45.0%	31.2%	32.5%	35.74%

Table 14. Compress ratio for different compression schemes.

Burrows-Wheeler tranformation has particular advantage for DNA text collection while the compression ratios for other kinds of text collections are in the approximately same level. The running time for its compression and decompression is experimented in table 15 and table 16. The tested files are different text collections in same size 10Kb.

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
10K	3853	5039	4337	4696	3713
20K	8456	6833	6146	8858	7426
40K	19547	17909	24680	20351	21949

Table 15. Burrows-Wheeler compression running time (unit: ms).

	DNA	ENGLISH	PROTEIN	SOURCE PROGRAM	XML
10K	1725	2862	1004	1139	1916
20K	3095	2922	2547	4077	2837
40K	8914	8255	9306	8033	8257

Table 16. Burrows-Wheeler decompression running time (unit: ms).

7. Conclusion and Further Work

In this paper, compressed indexes are discussed. FM-index is even implemented for proving the performance of compressed indexes. Suffix array and Burrows-Wheeler Transformer are the core data structures for FM-index. They have both been experimented. However, the practical structure for FM-index, as mentioned in chapter 1, is not tested in this paper. It is may be an appealing point for testing different sizes of superbucket and bucket.

From the statistics of experimental results, we can conclude that the FM-index has the potential to solve the tradeoff between time and space consumption. It is indeed a good choice for indexed searching.

Most of the developments of self-indexes are still in a theoretical stage. Some improvements on the consideration of algorithm engineering are demanded. Several aspects of FM-index are still immature [1]:

a) Construction time and space;

b) Management of secondary storage;

c) Searching for more complex patterns;

d) Updating upon text changes;

Except from these shortages, the conclusion between different self-indexes is not sufficient. It is still difficult to decide which self-index is suitable for a given text. This confusion is important as the number of existing self-indexes are quite large and the applications of self-indexes have already been expanded.

Bibliography

1. Navarro,G and Mëkinen, V. *Compressed Full-text Indexes*. ACM Computing Surveys (CSUR), 2007, Vol 39, Numb 1, pages 2.

2. Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*, Cambridge University Press, 2008.

3. Hussein Al-Bahadili, Saif Al-Saab, Reyadh Naoum and Shakir M. Hussain. *A Web Search Engine Model Based on Index-query Bit-level Compression.* Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, 2010.

4. Jurgens, M. *Index Structures for Data Warehouses*. Springer-Verlag, 2002, Issu 1859, pages all.

5. Juha Kärkkäinen and Esko Ukkonen. *Lempel-Ziv Parsing and Sublinear-size Index Structures for String Matching*. Proc. 3rd South American Workshop on String Processing (WSP), 1996, pages 141-155.

6. Ferragina Paolo and Manzini Giovanni. *Opportunistic Data Structures with Applications*. In Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS), 2000, Vol 41, pages 390-398.

7. Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

8. Edward Fredkin, Bolt Beranek and Newman, Inc., Cambridge, Mass. *Trie Memory*. Communications of the ACM, 1960, Vol 3, Issue 9.

9. Philippe Flajolet. *On the Performance Evaluation of Extendible Hashing and Trie Searching*. Acta informatica ,1983, Vol 20, Number 4, pages 345-369.

10. Douglas Comer and Ravi Sethi. *Complexity of Trie Index Construction*. 17th Annual Symposium on Foundations of Computer Science (FOCS 1976), 1976, Vol 24, Issue 3, pages 197-207.

11. Edward H. Sussenguth and Jr. *Use of Tree Structures for Processing Files*. Communications of the ACM, 1963, Vol 6, Issue 5, pages 272-279.

12. Jun-Ichi Aoe, Katsushi Morimoto and Takashi Sato. *An Efficient Implementation of Trie Structures*. Software: practice and experience, 1992, Vol 22, Issue 9, pages 695-721.

13. Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

14. P. Weiner. *Linear Pattern Matching Algorithms*. Proc. Of the 14th IEEE Symp. On switching and automata theory, 1973, pages 1-11.

15. E. M. McCreight. *A Space-economical Suffix Tree Construction Algorithm*. Journal of the ACM, 1976, Vol 23, pages 262-272.

16. Stefan Kurtz. *Reducing the Space Requirement of Suffix Trees*. Software practice and experience, 1999, Vol 29, Issue 13, pages 1149-1171.

17. Udi Manber and Gene Myers. *Suffix Arrays: A New Method for On-line String Searches*. Proceeding SODA '90 Proceedings of the first annual ACM-SIAM symposium on discrete algorithms, 1993, Vol 22, Number 5, pages 319-327.

18. J. Kärkkäinen and P. Sanders. *Simple Linear Work Suffix Array Construction*. Lecture notes in computer science, 2003, vol 2719, pages 943-955.

19. R. Dementiev, J. Mehnert, J. Kärkkäinen and P. Sanders. *Better External Memory Suffix Array Construction*. ACM journal of experimental algorithm, 2006.

20. K. Sayood, *Introduction to Data Compression*. New York: Morgan Kaufmann, 1996.

21. M. Burrows, D. J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*. Research report 124, Digital Equipment Corporation, 1994.

22. J. Ziv and A. Lempel. *A Universal Algorithm for Sequential Data Compression.* Information theory, 1997, Vol 23, Issue 3, pages 337-343.

23. Giovanni Manzini. *An Analysis of the Burrows-Wheeler Transforms*. Proceeding SODA '99 Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms, 1999.

24. Paolo Ferragina and Giovanni Manzini. *Indexing Compressed Text,* Journal of the ACM, 2005, Vol 52, Issue 4.

25. Paolo Ferragina, Rodrigo González, Gonzalo Navarro and Rossano Venturini. *Compressed Text Indexes: From Theory to Practice.* Journal of experimental algorithmic (JEA), 2009, Volume 13.

26. Paolo Ferragina and Giovanni Manzini. *An Experimental Study of a Compressed Index*. Information sciences, 2001, Vol 135, Issue1-2, pages 13-28.

27. Masahiro Kasahara, Shinichi Morishita. *Large-scale Genome Sequence Processing.* Imperial College Press, 2006, pages 68.

28. Bentley, J.L. et al. *A Locally Adaptive Compression Scheme*. Communications of the ACM, 1986, Vol 24, Issue 9, pages 320-330.

29. David Salomon. *Data Compression*. Springer, 1997.

30. Huffman, David. *A Method for the Construction of Minimum Redundancy Codes.* Proceedings of the IRE, 1952, Vol 40, Issue 9, pages 1098-1101.

31. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001, ISBN 0-262-03293-7. Section 16.3, pages 385–392.

32. Pizza&chili corpus. Available at: http://pizzachili.dcc.uchile.cl/index.html.

Appendix: source code

a) Suffix array construction: doubling algorithm

```
//Build suffix array using doubling algorithm
int* GetSA(char* S
{
    int* SA;
    int len=strlen(S);
    int* RANK_k;
    bool* Flag;
    char* Tmp;
    int i,j;
    int k=0;
    int R=0;
    /***Initialisation***/
    SA=new int[len];
    for(i=0;i<len;i++)
        SA[i]=0;
    RANK_k=new int[len];
    Flag=new bool[len];
    Tmp=new char[len];
    memset(RANK_k,0,len);
    memset(Flag,false,len);
    memset(Tmp,0,len);
    //strcpy_s(Tmp,S);
    for(i=0;i<len;i++)
        Tmp[i]=S[i];
    /***Build SA_1***/
    char MinChar='~';  //'~' is larger than any char in alphabet
    int MinNum=-1;
    for(i=0;i<len;i++)
    {
        for(j=0;j<len;j++)
        {
            if(Tmp[j]<MinChar)
            {
                MinChar=S[j];
                MinNum=j;
            }
```

```
        }
        SA[i]=MinNum;
        Tmp[MinNum]='~';
        MinChar='~';
    }
    /***Build SA_2k and RANK_2k ...***/
    MinNum=0;
    while((int)pow(2.0,k)<=len)
    {
        /***Build RANK_k, RANK_2k, RANK_4k...***/
        R=0;
        RANK_k[SA[0]]=R;
        for(i=1;i<len;i++)
        {
            if(Cmp(S,SA[i],SA[i-1],(int)pow(2.0,k),len)==true)
            //S[SA[i]] =k S[SA[i-1]]
            {
                RANK_k[SA[i]]=R;
            }
            else
            {
                R++;
                RANK_k[SA[i]]=R;
            }
        }
        k++;
        MinNum=-1;
        /***Build SA_2k, SA_4k...***/
        for(i=0;i<len;i++)
        {
            MinNum=-1;
            for(j=0;j<len;j++)
            {
                if(Flag[j]==false)
                {
                    if(MinNum==-1)  MinNum=j;
                    if(RANK_k[j]<RANK_k[MinNum]||
                       (RANK_k[j]==RANK_k[MinNum] &&
                       (RANK_k[j+k]<RANK_k[MinNum+k]))
                    {
                        MinNum=j;
```

```
}
                }
            }
            Flag[MinNum]=true;
            SA[i]=MinNum;
        }
        memset(Flag,false,len);
    }
    free(RANK_k);
    free(Flag);
    free(Tmp);
    return SA;
}
//Compare strings
//if equal then return true
bool Cmp(char* S,int m,int n,int k,int len)
{
    int i;
    for(i=0;i<k && m+i<len && n+i<len;i++)
    {
        if(S[m+i]!=S[n+i])
        {
            return false;
        }
    }
    return true;
}
```

b) Suffix array searching: binary searching

```
typedef struct position
{
    int sp;  //start position
    int ep;  //end position
}position;

position BinarySearch(char* S, int* SA, char* P)
{
    position ps;
    int st,et;
    int s,e;
```

```
    int len=strlen(S);
    ps.sp=0;
    st=len-1;
    while(ps.sp<st)
    {
        s=(int)floor((double)(ps.sp+st)/2);
        if(PatternCmp(S,SA,P,s)>0)
        {
            ps.sp=s+1;
        }
        else
        {
            st=s;
        }
    }
    ps.ep=ps.sp-1;
    et=len-1;
    while(ps.ep<et)
    {
        e=(int)ceil((double)(ps.ep+et)/2);
        if(PatternCmp(S,SA,P,e)==0)
        {
            ps.ep=e;
        }
        else
        {
            et=e-1;
        }
    }
    return ps;
}
```

c) Burrows-Wheeler Tran formation: decompression transformation

```
void Decompress(char* Text, int* SA, char* L, int I, char* Alphabet,
char* S)
{
    char* F;
    //the total number of instances in L of characters preceding
    character C[i] in alphabet
    int* C;
    int* P;  //the number of instances of character L[i] in the prefix
```

```
L[0,...,i-1] of L.
int* T;  //F[T[j]]=L[j]
int length=strlen(Text);
int i,j;
int n;  //the length of the alphabet
int count;
int tmp;
/***D1 find first characters of rotations (F)***/
F=new char[length];
memset(F,0,length);
for(i=0;i<length;i++)
    F[i]=Text[SA[i]];
/***D2 build list of predecessor characters (T)***/
n=strlen(Alphabet);
C=new int[n];
memset(C,0,n);
for(i=0;i<n;i++)
{
    count=0;
    for(j=0;j<length;j++)
    {
        if(L[j]<Alphabet[i])
        {
            count++;
        }
    }
    C[i]=count;
}
P=new int[length];
memset(P,0,length);
for(i=0;i<length;i++)
{
    count=0;
    for(j=0;j<i;j++)
    {
        if(L[j]==L[i])
        {
            count++;
        }
    }
    P[i]=count;
```

```
}
/***Print P***/
printf("\nP in BWT:\n");
for(i=0;i<length;i++)
   printf("%d  ",P[i]);
printf("\n");
T=new int[length];
memset(T,0,length);
for(i=0;i<length;i++)
{
   for(j=0;j<n;j++)
   {
      if(L[i]==Alphabet[j])
      {
         T[i]=P[i]+C[j];
      }
   }
}
/***Print T***/
printf("\nT in BWT:\n");
for(i=0;i<length;i++)
   printf("%d  ",T[i]);
printf("\n");
/***D3 form output S***/
for(i=0;i<length;i++)
{
   if(i==0)
   {
      j=I;
      tmp=j;
   }
   else
   {
      j=T[tmp];
      tmp=j;
   }
   S[length-1-i]=L[j];
}
/***Print S***/
printf("\nS in BWT:\n");
for(i=0;i<length;i++)
```

```
        printf("%c  ",S[i]);
    printf("\n");
}
```

d) Burrows-Wheeler Transformation: Huffman coding

```
HuffmanTree HT;
HuffmanCode HC;
void HuffmanCoding(unsigned short* R, int length)
{
    unsigned short* v;
    double* w;
    int i,j,k;
    int n=1;
    int m;  //number of leaf
    int s1=0;
    int s2=0;
    int start;
    int c;
    int f;
    char* cd;
    bool flag=true;
    HuffmanTree p;
    FILE* fp;
    /***Count the number of different characters in R***/
    for(i=1;i<length;i++)
    {
        flag=true;
        for(j=0;j<i;j++)
        {
            if(R[j]==R[i])
            {
                flag=false;
            }
        }
        if(flag==true)
        {
            n++;
        }
    }
    printf("Here");
```

```
/***Initialisation***/
v=new unsigned short[n];
memset(v,0,n);
w=new double[n];
memset(w,0,n);
/***Build v,w,T***/
k=1;
v[0]=R[0];
for(i=1;i<length;i++)
{
    flag=true; //If R[i] is a new character, then flag=true
    for(j=0;j<i;j++)
    {
        if(R[j]==R[i])
        {
            flag=false;
        }
    }
    if(flag==true)
    {
        v[k]=R[i];
        k++;
    }
}
for(i=0;i<n;i++)
{
    k=0;
    for(j=0;j<length;j++)
    {
        if(R[j]==v[i])
        {
            k++;
        }
    }
    w[i]=(double)k/(double)length;
}
/***HuffmanTree initialisation***/
if(n<=1) return;
m=2*n-1;
HT=(HuffmanTree)malloc((m)*sizeof(HTNode));
p=HT;
```

```
    for(i=0;i<n;i++)
    {
        p->weight=w[i];
        p->parent=0;
        p->lchild=0;
        p->rchild=0;
        p++;
    }
    for(;i<m;i++)
    {
        p->weight=(double)0.0;
        p->parent=0;
        p->lchild=0;
        p->rchild=0;
        p++;
    }
    p=HT;
    /***HuffmanTree building***/
    for(i=n;i<m;i++)
    {
        Select(HT,i,&s1,&s2);
        HT[s1].parent=i;
        HT[s2].parent=i;
        HT[i].lchild=s1;
        HT[i].rchild=s2;
        HT[i].weight=HT[s1].weight+HT[s2].weight;
    }
    printf("\n");
    p=HT;
    for(j=0;j<m;j++)
{
    printf("%d:  %f  %d  %d  %d\n",j,p->weight,p->parent,p->lchild,
p->rchild);
        p++;
    }
    /***Huffman code building***/
    HC=(HuffmanCode)malloc((n)*sizeof(char *));
    cd=(char*)malloc(n*sizeof(char));
    cd[n-1]='\0';
    for(i=0;i<n;i++)
    {
```

```
        start=n-1;
        for(c=i,f=HT[i].parent;f!=0;c=f,f=HT[f].parent)
        {
            if(HT[f].lchild==c)
            {
                cd[--start]='0';
            }
            else
            {
                cd[--start]='1';
            }
        }
        HC[i]=(char*)malloc((n-start)*sizeof(char));
        for(j=0;j<n-start;j++)
        {
            HC[i][j]=cd[start+j];
        }
    }
    free(cd);
    free(HT);
    /***Write file***/
    fp=fopen("result.txt","w");
    if(fp==NULL)
    {
        printf("can't open file!\n");
        exit(0);
    }
    for(i=0;i<length;i++)
    {
        for(j=0;v[j]!=R[i];j++);
        fprintf(fp,"%s",HC[j]);
    }
    fclose(fp);
}
void Select(HuffmanTree &HT,int i,int* s1,int* s2)
{
    int j;
    double min1,min2;
    HuffmanTree p;
    min1=1.0;  //value of smallest node
    min2=1.0;  //value of smallest node
```

```
    p=HT;
    for(j=0;j<=i-1;j++)
    {
        if(p->weight<min1 && p->parent==0)
        {
            *s1=j;
            min1=p->weight;
        }
        p++;
    }
    p=HT;
    for(j=0;j<=i-1;j++)
    {
        if(p->weight<=min2 && j!=*s1 && p->parent==0)
        {
            *s2=j;
            min2=p->weight;

        }
        p++;
    }
}
```