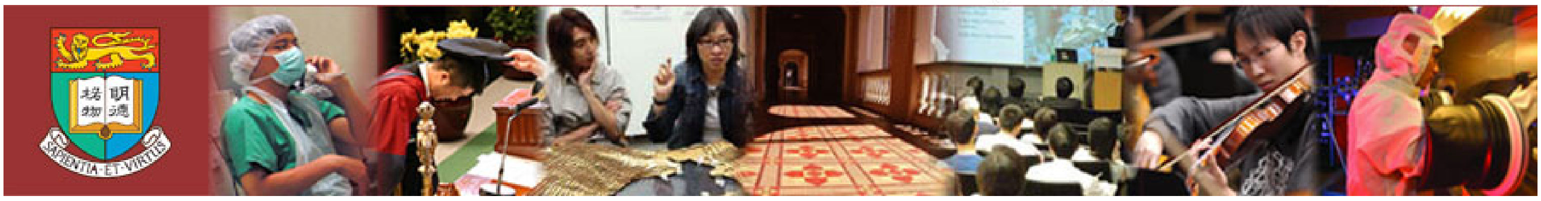


Bibliographic Notes

Chapter 3 to Chapter 8 consist of the main technical part of this thesis, whose materials are respectively extracted and revised from the following papers:

- Chapter 3 – "Constructing Compressed Suffix Arrays with Large Alphabets", in *Proceedings of the 14th International Conference on Algorithms and Computation (ISAAC'03)*, pp. 240–249, 2003. (Co-authored with T. W. Lam, K. Sadakane and W. K. Sung.)
- Chapter 4 – "Breaking a Time-and-Space Barrier in Constructing Full-Text Indices", in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, pp. 251–260, 2003. (Co-authored with K. Sadakane and W. K. Sung.)
- Chapter 5 – "Space-Economical Algorithms for Finding Maximal Unique Matches", in *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, pp. 144–152, 2002. (Co-authored with K. Sadakane.)
- Chapter 6 – "Compressed Index for a Dynamic Collection of Texts", in *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04)*, pp. 445–456, 2004. (Co-authored with H. L. Chan and T. W. Lam.)
- Chapter 7 – "Compressed Index for Dynamic Text", in *Proceedings of the 14th IEEE Data Compression Conference (DCC'04)*, pp. 102–111, 2004. (Co-authored with T. W. Lam, K. Sadakane, W. K. Sung and S. M. Yiu.)
- Chapter 8 – "Practical Aspects of Compressed Suffix Arrays and FM-index in Searching DNA Sequences", in *Proceedings of the 5th Workshop on Algorithm Engineering and Experimentations (ALENEX'04)*, pp. 31–38, 2004. (Co-authored with T. W. Lam, W. K. Sung, W. L. Tse, C. K. Wong and S. M. Yiu.)

Chapter 1

Introduction

This thesis studies the theoretical and practical aspects of maintaining indexes for long texts, especially the DNA sequences which find most applications nowadays, that are minute in space and exhibit competent text searching ability when compared to the uncompressed counterparts. Particularly, two recent indexes, namely the CSA and the FM-index, are studied. Our results include time-and-space efficient algorithms that construct these indexes, experimentation for the practical performance of these indexes, and extension of these indexes in solving more complicated text queries. A brief introduction is shown as follows.

1.1 Survey on Traditional Text Indexes

Text searching is a classical problem in computer science. Let T be a text of length n, and P be a pattern of length p. The basic problem is to locate all occurrences of the pattern P in the text T efficiently. Using Knuth-Morris-Pratt [49] or Boyer-Moore [12] algorithms, this can be solved in the optimal $O(n + p)$ time. In most applications, the text is often given in advance, and will be searched against different patterns. It thus pays some extra memory to pre-process the text and create an *index* on it, so as to facilitate the subsequent searching process.

Text indexes can be classified into two categories, namely, the *word-based* indexes and the *full-text* indexes. Word-based indexes, such as *inverted lists* [40] and *signature files* [21], are used for texts with word boundaries. They are constructed based on the distinct words that appear in the texts. For instance, the inverted lists of a text store, for each distinct word, the list of positions that the word occurs in the text; and, the set of distinct words are stored via a trie or hashing to allow efficient access to a particular list. Subsequent searching for a word can be done efficiently by locating the corresponding list and enumerating all the positions of occurrence. For signature files, it is built by first partitioning the text into *pages*; then, each word is hashed into a fixed length s-bit value, and a page is represented by a signature that forms by 'OR'ing all the hash values of the words appearing in that page. Subsequent searching for a word can often be confined to a small number of pages whose signature 'agrees' with the hash value of the word. In general, word-based indexes are able to support fast word queries; in addition, they are small in size, occupying about 20-30% of the original text size [82].

Unfortunately, word-based indexes are not suitable to handle texts without clear-cut word boundaries like DNA sequences, Chinese texts and Japanese texts. In these circumstances, full-text indexes, which are indexes that make no assumption on the word boundary, are relied upon, at the cost of increasing the space occupancy. Basically, these indexes are constructed by storing information on all the substrings occurring in the texts. *Suffix trees* [57, 79] and *suffix arrays* [56] are two fundamental full-text indexes in the literature, which find numerous applications in areas including biological research (e.g., gene hunting, promoter consensus identification, and motif finding) [38], data mining [41] and text compression [28]. Suffix tree is a compact version of the *trie* that stores all suffixes of the given text. Each suffix is represented by a unique leaf storing its starting position. Based on the suffix tree, any substring of the text can be found by following some path descending from the root. The importance of the suffix tree is underlined by the fact that it has been rediscovered many times in the scientific literature, disguised under different names [34]. Some examples include the compact bi-tree [79], the prefix tree [15], the PAT tree [30], the position tree [3, 47, 53], the repetition finder [67], and the subword tree [8, 15]. On the other hand, suffix array is a reduced form of a suffix tree, which is obtained by visiting the leaves of the corresponding suffix tree from left to right. More precisely, it is an array of positions, sorted in the lexicographic order of the corresponding suffixes.

Both suffix tree and suffix array exhibit superb searching performance. Given the suffix tree of a text T whose characters are from an alphabet Σ, we can search for a pattern P within T using $O(p \log |\Sigma| + occ)$ time, * where occ denotes the

*We use the notation $\log_b^c n$ to denote $(\log n / \log b)^c$, which is the c-th power of the base-b logarithm of n. Unless specified, we use $b = 2$.

number of occurrences of P in T. Note that the time is independent of the text size. For suffix array, its searching time is $O(p + \log n + occ)$, which is only a bit slower. For the space concern, both of them require $O(n \log n)$ bits; suffix array is associated with a smaller constant, though.

In the literature, there is another full-text index called *directed acyclic word graph* (DAWG) [10], which is the smallest finite state automaton that recognizes all the substrings appearing in the given text [17]. Thus, based on DAWG, the existence of a pattern P in T can be determined in $O(p)$ time. By compacting the edges of the DAWG and augmenting additional information in the nodes, we obtain the *labeled compact DAWG* (CDAWG) of [11], which is equivalently obtained from the suffix tree of the text by merging its edge-isomorphic subtrees and deleting part of the resulting structure [34]. CDAWG provides significant reductions of the memory space required by suffix trees and DAWGs [18]; nevertheless, in order to support locating all the occurrences of a pattern in the text, the space requirement is still $O(n \log n)$ bits.

With the advance in bio-technology, the complete DNA sequences for a number of living organisms have been known. Some examples of these DNA sequences are depicted in Table 1.1. The size of the DNA sequences can be much longer than the traditional texts. For instance, the human DNA comprises about 3.3 billion characters. For this DNA sequence, the best known implementation of suffix tree and suffix array require 40 Gigabytes [29, 50] and 14 Gigabytes, respectively. Such memory requirement far exceeds the capacity of ordinary computers. Existing approaches for indexing human DNA include (1) using supercomputers with large main memory [75] and (2) storing the indexing data structure in the secondary storage [16, 41]. The first approach is expensive and inflexible, while the second one is slow. As more and more DNA are decoded, it is vital that individual biologists can eventually analyze different DNA sequences efficiently with their ordinary PCs. In the next section, we show the recent trend in tackling the problem—by compressing the index.

1.2 Survey on Compressed Text Indexes

To overcome the memory requirement, we need to construct indexes that require considerably less space. Perhaps the most effective way to start with is to reduce the space of the existing indexes, while maintaining their searching powers. This idea has stimulated many work in the last decade. Firstly, Kärkäinen (1995) [43]

Table 1.1: The size of some known DNA sequences. [48]

Sequence Name	Size (bases)	Notes
Phi-X 174	5386	virus of E. coli
Human mitochondrion	16,569	
Mycoplasma genitalium	580,073	three of the smallest true organisms
Ureaplasma urealyticum	751,719	
Mycoplasma pneumoniae	816,394	
Campylobacter jejuni	1,641,481	frequent cause of food poisoning
Helicobacter pylori	1,667,867	chief cause of stomach ulcers (not stress and diet)
E. coli	4,639,221	
Saccharomyces cerevisiae	12,495,682	the budding yeast
Drosophila melanogaster	122,653,977	the fruit fly
Fugu rubripes	3.65×10^8	the pufferfish
Rice	4.3×10^8	
Human	3.3×10^9	
Amphibians	10^9 - 10^{11}	
Psilotum nudum	2.5×10^{11}	the whisk fern

proposed a new suffix-tree-like index called *suffix cactus*, whose structure can be seen as either a compact version of the suffix tree or as an augmented suffix array. Its size lies between those of suffix tree and suffix array, and pattern matching requires $O(p|\Sigma| + occ)$ time. Later, Kärkäinen and Ukkonen (1996) [45] initiated the study of *sparse suffix tree* (SST), which is a suffix tree that represents only a subset of suffixes of the text. In particular, they gave construction and search algorithms for the special case called *evenly spaced SST*, which is an SST that represents every k-th suffix of the text for some fixed k. By increasing k, the SST can be made arbitrarily small; unfortunately, the pattern matching time could be huge, requiring $O(kn)$ time in the worst case. Afterwards, Munro, Raman and Rao (1998) [62] devised the *space efficient suffix tree*, which requires $n \log n + O(n)$ bits space. Under the standard unit-cost RAM model,[†] their index supports pattern matching in $O(p + occ)$ time. Recently, Abouelhoda, Ohlebusch and Kurtz (2002) [1] enhanced the suffix array to support $O(p+occ)$-time pattern matching query; their data structure requires two additional $n\lceil \log n \rceil$-bit arrays

[†]The standard unit-cost RAM model assumes that all the standard arithmetic and boolean operations on $\log n$ bit words and reading and writing $\log n$ bit strings can be performed in constant time.

and makes no assumption on the computational model. On the other hand, Mäkinen (2000) [54] proposed the *compact suffix array*, in an attempt to further reduce the size of the suffix array. The idea is analogous to compacting the suffix tree to become a CDAWG, where the compact suffix array is obtained by replacing some areas of a suffix array with links of the other areas. In practice, the compact suffix array achieves a 25% to 60% reduction in space; for searching, it is comparable – ranging from 2 to 3 times – with the suffix array, though there is no worst-case guarantee. Another interesting index is the *Lempel-Ziv index* by Kärkkäinen and Sutinen (1996) [44], which requires $O(n)$ bits and takes $O(p+occ)$ time to search patterns of length shorter than $\log n$; for longer patterns, the index may occupy $\Theta(n \log n)$ bits.

Despite the remarkable improvements, all of the above indexes still require $O(n \log n)$ bits in order to maintain competitive searching performance. Innovative elements are quested to answer the space requirement problem. Recent breakthrough results shed light in this direction. The first achievement of a new trend started with Grossi and Vitter (2000) [35]. They transformed the suffix array for binary texts into a more compressible function, called *compressed suffix arrays* (CSA), which needed $O(n)$ bits. Under the unit-cost RAM model, accessing any entry of the suffix array can be done in $O(\log^\epsilon n)$ time for any $\epsilon > 0$. Based on the CSA, they have proposed an index which occupies $O(n)$ bits and supports pattern matching in $O(p/\log n + occ \log^\epsilon n)$ time. Later, Sadakane (2000) [74] extended the study of CSA for general texts, giving an index that requires $O(n(H_0+1))$ bits, where H_0 denotes the zero-th order entropy of the text. This index supports searching in $O(p \log n + occ \log^\epsilon n)$ time. Moreover, it does not require to store the original text explicitly, while any substring of the text of length ℓ can be retrieved in $O(\ell + \log^\epsilon n)$ time. On the other hand, Rao (2002) [69] targeted at the space-and-time tradeoff of the CSA, and proposed a family of CSA for binary texts that takes $O(nt(\log n)^{1/t})$ bits and each entry of the suffix array can be accessed in $O(t)$ time, for any parameter t with $1 \leq t \leq \log \log n$. When $t = O(1/\epsilon)$, this tradeoff gives the most space-efficient representation of a suffix array with constant access time; furthermore, when coupled with the idea in the original paper of Grossi and Vitter, Rao obtained an index that requires $O(n \log^\epsilon n)$ bits and supports pattern matching in $O(p/\log n + occ)$ time.

At the same time as the debut of CSA, Ferragina and Manzini (2000) [25] made use of another approach in compressing the suffix arrays. Their index, called *FM-index*,[‡] is based on Burrows-Wheeler transform [13] and Move-to-Front encoding [9]. For texts with constant-size alphabets, the index requires at most $5nH_k + o(n)$ bits, and can answer pattern matching query in $O(p + occ \log^{1+\epsilon} n)$ time, where H_k is the k-th order entropy of the text; for general texts, a $2^{O(|\Sigma| \log |\Sigma|)}$-bit decoding table is required to maintain the query time.

An interesting point about CSA and FM-index is that, both indexes do not require storing the original texts explicitly for searching, and we are able to reproduce the original text without loss based on them. Thus, the original text is included implicitly, and the two indexes are also referred as *self-index*. For human DNA, the CSA or the FM-index occupy about 2 Gigabytes. Nowadays a PC can have up to 4 Gigabytes of main memory and can easily accommodate such a data structure.

Recently, Sadakane (2002) [73] enhanced the CSA to include the longest common prefix (LCP) information, with which we can count the occurrences of a pattern P in $O(p)$ time. The index requires $nH_1/\epsilon + O(n)$ bits. Later, Sadakane (2003) [74] made further use of the LCP information, and obtained the *compressed suffix tree* (CST) which requires $O(n(H_0 + 1))$ bits and supports every navigation operation on the suffix tree in at most $O(\log^\epsilon n)$ time.

Also, two new indexes are proposed recently that are based on the Ziv-Lempel compression algorithms [83, 80]. The first one is by Ferragina and Manzini (2002) [27] which extends their FM-index and achieves $O(p + occ)$ time for the pattern matching query. The index is suitable for texts over constant-size alphabets, and it requires $O(nH_k \log^\epsilon n) + o(n)$ bits of storage, which is a factor of $\log^\epsilon n$ more than the original FM-index. The second one is the *LZ-index* by Navarro (2002) [63], which, on the other hand, is suitable for general texts. Like CSA or FM-index, this index is also a self-index. For the performance, it supports pattern matching in $O(p^3 \log |\Sigma| + (p+occ) \log n)$ time, which could be slow when pattern is long; on contrary, the space occupancy is little, requiring only $nH_k(4 + o(1))$ bits of storage.

There are yet two more indexes that are noteworthy. The first one is by Grossi, Gupta and Vitter (2003) [32], which provides another extension of CSA for general texts, while aiming at storage space to be as close to the empirical entropy of the text as possible. Basically, their idea is to organize the CSA using wavelet trees and succinct static dictionaries [66, 68]. The space requirement is $nH_k + o(n \log |\Sigma| \log \log n / \log n)$ bits, and pattern matching can be supported

[‡]According to the authors, FM-index stands for *full-text* index in *minute* space.

in $O(p \log |\Sigma| + \text{polylog}(n) + occ \log^2 n / \log \log n)$ time. The second one is the *compressed compact suffix arrays* introduced by Mäkinen and Navarro (2004) [55], which is an improvement over the compact suffix arrays of Mäkinen [54]. The first improvement is that it becomes a self-index. In addition, the space is reduced from $O(n \log n)$ bits to $O(nH_k \log n) + O(n)$ bits, while pattern matching has a worst-case guarantee, needing $O((p + occ) \log n)$ time.

1.3 Our Contribution

The exciting development of the compressed text indexes has offered us the fertile grounds for studies in this thesis. Theoretically, we have come up with two space-and-time efficient construction algorithms for CSA and FM-index, and we have designed compressed indexes that solve other text searching problem. Empirically, we have examined the behavior of CSA and FM-index in practical use. These results are presented as follows.

1.3.1 Space-and-Time Efficient Construction Algorithms for CSA and FM-index

Theoretically speaking, the compressed suffix arrays, namely the CSA and the FM-index, mentioned in the previous section can be constructed using $O(n)$ time; however, the construction process requires much more than $O(n)$ bits of memory. Among others, the original suffix array has to be built first, taking up at least $n \log n$ bits. In the context of human DNA, the memory for constructing a compressed suffix array is at least 25 Gigabytes [75], far exceeding the capacity of ordinary PCs. This motivates us to investigate whether we can construct a compressed suffix array in $O(n \log n)$ time while using $O(n)$ bits of memory.

Assuming the standard unit-cost word RAM,[§] this thesis provides the first two algorithms of such a kind, showing that CSA [35, 72] can be built in a space-and-time efficient manner. These results also imply a space-and-time efficient algorithm for constructing the FM-index, as FM-index can be conveniently converted from CSA with $O(n \log |\Sigma|)$-bit space and $O(n \log^\epsilon n)$ time for any fixed and positive ϵ. In general, our construction algorithms also work well for indexing

[§]In this computation model, standard arithmetic and bitwise boolean operations on word-sized operands take constant time.

other kinds of texts like Chinese or Japanese, whose alphabet consists of at least a few thousand characters.

Our first algorithm does not require much space other than that for storing the compressed suffix arrays. Precisely, for a text with an alphabet Σ, our algorithm requires $O(n(H_0 + 1))$ bits of memory, where $H_0 \le \log |\Sigma|$ denotes the entropy of the text; for the time complexity, it is $O(n \log n)$ which is independent of $|\Sigma|$. Technically speaking, the efficiency in space is based on an observation that the compressed suffix arrays of two consecutive suffixes are very similar. Thus, we can build the entire compressed suffix array directly from the text in an incremental 'character by character' manner. Exploiting this observation further, we can speed up the construction by processing more characters each time, yielding a 'segment by segment' algorithm that runs in $O(n \log n)$ time.

Our second algorithm, on the other hand, requires considerably less time, while using slightly more space than the first one. Precisely, for a text with an alphabet Σ, our algorithm requires $O(n \log \log |\Sigma|)$ time and $O(n \log |\Sigma|)$ bits of memory. Note that when $|\Sigma| = O(1)$, both time and space requirements become $O(n)$, which is optimal. Another significant implication from this algorithm is that, suffix tree and suffix array can be constructed in $o(n \log n)$ time and $o(n \log n)$-bit *working space*. Here, we assume that the output is written directly to a 'write-only' secondary storage[¶] without occupying the main memory, so that the working space does not include the space for the output. This solves a question that has been open for a long time.

The second algorithm is technically more interesting. We have borrowed the framework of Farach's linear time suffix tree construction algorithm [22] to construct our compressed suffix arrays. Basically, Farach's algorithm first constructs the suffix tree for the even-position suffixes by recursion, which is then used to induce the one for the odd-position suffixes. Afterwards, these two suffix trees are merged together to produce the required suffix tree. Here, we have make use of CSA and the Burrows-Wheeler text [13] alternately to represent a suffix tree so as to avoid storing the suffix pointers explicitly. Immediately, this gives an $O(n \log n)$ time algorithm that constructs the compressed suffix arrays in $O(n \log |\Sigma|)$-bit space. A further improvement is related to the backward search algorithm, which is used to find a pattern within the text based on the CSA. If

[¶]The 'write-only' condition separates our algorithm from the external memory algorithms [78], which treat secondary storage as an extension of the main memory and the focus is to reduce the number of I/O operations instead of CPU time.

we apply a known method [75], each step of the algorithm requires $O(\log n)$ time. This thesis presents a new $O(n \log |\Sigma|)$-bit auxiliary data structure which speeds up each backward search step into $O(\log \log |\Sigma|)$ time. Consequently, the time complexity of the construction algorithm is improved to $O(n \log \log |\Sigma|)$.

A summary of our results and the existing methods for constructing the CSA is shown in Table 1.2.

Table 1.2: The space and time complexities for constructing CSA.

	Description	Space (bits)	Time
existing	via suffix tree [50, 57]	$5n \log n$	$O(n)$
	via suffix array [52, 56]	$2n \log n$	$O(n \log n)$
this thesis	optimal space	$O(n(H_0 + 1))$	$O(n \log n)$
	optimal (when $\|\Sigma\| = O(1)$)	$O(n \log \|\Sigma\|)$	$O(n \log \log \|\Sigma\|)$

1.3.2 Design of Compressed Indexes for Advanced Text Searching Problems

In previous sections, we have seen how a compressed version of a suffix array can be constructed and applied to solve a simple text searching problem, under the standard unit-cost word RAM. The next question is to ask if a compressed version of a suffix tree exists, as suffix tree is the heart of many data structures for solving more complicated queries. Sadakane [71] has answered this affirmatively, giving a data structure that supports all the functionalities of a suffix tree in $O(n \log |\Sigma|)$-bit space, with each operation slowing down by at most a factor of $O(\log^\epsilon n)$, for any fixed $\epsilon > 0$.

This thesis extends the above work, where we show that this compressed version of a suffix tree, or CST, can be constructed from the CSA efficiently in $O(n \log^\epsilon n)$ time and $O(n \log |\Sigma|)$-bit space. In addition, we show that CST can be coupled with CSA and FM-index to solve three advanced text problems: finding *maximal unique matches* (MUM), managing a *dynamic library* and managing a *dynamic dictionary*. Then, our solutions to the latter two problems immediately imply a compressed index for the *dynamic text* problem. Again, we have assumed the standard unit-cost word RAM as our computation model. In the following, we shall elaborate each of our results one by one.

Finding Maximal Unique Matches

Aligning two DNA sequences has been a long-standing topic in computational biology. Important early work includes Needleman and Wunsch [64], and Smith and Waterman [77]. When two DNA sequences are aligned, many interesting biological features may be identified. For example, exact matches may suggest an occurrence of a gene, whereas SNPs (single nucleotide polymorphism), tandem repeats, large inserts or reversals can be found around the mismatches.

However, traditional alignment algorithms require $O(mn)$ time, where m and n denote the length of the two input sequences. This is impractical for aligning two whole DNA sequences where m and n are of the order of 10^6 or even more. To circumvent the timing problem, Delcher *et al.* [19] designed an efficient heuristic algorithm, which exploits a reasonable intuition that, if a long and identical sequence occurs exactly once in each DNA sequence (which they called a maximal unique match, or MUM), such a sequence is almost certain to be part of the alignment. The alignment problem is now reduced to finding the MUM's, aligning the MUM's and finally aligning the local gaps between successive MUM's. When the input DNA sequences are short enough, the first step can be solved in $O(m+n)$ time by constructing a suffix tree for the concatenation of the input sequences, while the latter two steps in practice can be performed with little space and little time.

When the input sequences are long, the above method is no longer applicable because of the huge space required by the suffix tree in finding the MUM's. Nevertheless, in this thesis, we show that when given the CST for the concatenation of the input sequences, this first step can still be done in optimal $O(m+n)$ time, while requiring only $O((m+n)\log|\Sigma|)$-bit space. Table 1.3 gives a summary of these results.

Table 1.3: Finding MUM's for two texts with length m and n.

	Description	Space (bits)	Time
existing	via suffix tree [19]	$O((m+n)\log(m+n))$	$O(m+n)$
this thesis	when CST is given	$O((m+n)\log\vert\Sigma\vert)$	$O(m+n)$
	only texts are given	$O((m+n)\log\vert\Sigma\vert)$	$O((m+n)\log^{\epsilon}(m+n))$

Managing Dynamic Library

Given a collection $\mathcal{L}$ of texts of total length n, we want to maintain an index for $\mathcal{L}$ such that when a pattern P is given later, all occurrences of P in $\mathcal{L}$ can be reported efficiently. This extension of the simple text searching query is called the *library matching* query, which appears naturally in homepage searching [31], classification of genes or proteins [60], and many other real-life applications.

In the static case where the texts in $\mathcal{L}$ never change, we can concatenate all the texts in $\mathcal{L}$ to form one text, and then build a suffix tree for this single text. The space occupancy is $O(n \log n)$ bits, and the library matching query is reduced to the simple text searching problem, which can be solved optimally in $O(p + occ)$ time, where p is the length of P and occ denotes the number of occurrences. We can reduce the space to $O(n \log |\Sigma|)$ bits, making use of CSA or FM-index. The searching times become $O(p \log \log |\Sigma| + occ \log^{\epsilon} n)$ and $O(p + occ \log^{\epsilon} n)$ respectively, for any fixed $\epsilon > 0$.

In the dynamic case where texts can be inserted or deleted from $\mathcal{L}$, we need to update the index properly so that the library matching query can be performed correctly when $\mathcal{L}$ is changed. In addition, such an update should be done efficiently. We call this the *dynamic library management* problem. If $O(n \log n)$ bits of space is allowed, one can build a generalized suffix tree, which is a single compact trie containing the suffixes of each text in $\mathcal{L}$. Then, to insert or delete a text of length t in $\mathcal{L}$, we update the generalized suffix tree by adding or removing all suffixes of this text, which can be done in $O(t)$ time. For searching a pattern P, the time remains $O(p + occ)$.

To reduce space, one may attempt to 'dynamize' a compressed index such as CSA or FM-index. Indeed, Ferragina and Manzini have demonstrated in [25] how to maintain multiple FM-indexes for the dynamic library management. Their solution requires $O(n + m \log n)$ bits, where m is the number of texts in the collection. Pattern matching is only slowed down slightly, using $O(p \log^3 n + occ \log n)$ time. However, insertion and deletion have only amortized performance guarantee; precisely, insertion and deletion of a text of length t takes $O(t \log n)$ and $O(t \log^2 n)$ amortized time, respectively. In the worst case, a single insertion or deletion may require re-constructing many of the FM-indexes, using $\Theta(n/ \log^2 n)$ time even if t is very small.

In this thesis, we introduce a compressed index for the dynamic library management problem for texts with constant-size alphabet, which requires only $O(n)$

bits. Inserting or deleting a text of length t takes $O(t \log n)$ time, while searching for a pattern takes $O(p \log n + occ \log^2 n)$ time. Note that the time complexities of all operations are measured in the worst case (instead of the amortized case). To our knowledge, this is the first result that requires only $O(n)$ bits, yet supporting both update and searching efficiently, i.e., in $O(t \log^{O(1)} n)$ and $O((p + occ) \log^{O(1)} n)$ time, respectively.

Technically speaking, our compressed index is based on CSA and FM-index. Yet a few more techniques are needed in order to achieve the optimal space requirement and efficient updating. Firstly, recall that the index proposed in [25] requires $O(n + m \log n)$ bits. We have a simple but useful trick in organizing the texts to avoid using a lot of space when the collection involves a lot of very short strings, thus eliminating the $m \log n$ term. Secondly, the original representations of CSA and FM-index do not support updates efficiently. For instance, the index proposed in [25], essentially requires re-building one or more FM-index whenever a text is inserted. Inspired by a dynamic representation of CSA in [51], we manage to dynamize the CSA and the FM-index to support efficient updates to a collection of texts. With either of them, we can immediately obtain an $O(n)$-bit index that supports updates in $O(t \log^2 n)$ time. For pattern matching, using FM-index alone can achieve $O(p \log n + occ \log^2 n)$ time, and using CSA alone takes $O(p \log^2 n + occ \log^2 n)$ time.

Last but not the least, we find that FM-index and CSA can complement each other nicely to further improve the update time. Roughly speaking, in the process of updating such suffix-array based compressed indexes, we need two pieces of crucial information; we observe that one of them can be provided quickly by FM-index, and the other can be provided quickly by CSA. Thus, by maintaining both CSA and FM-index together, we can perform the update in a straightforward manner, improving the update time to $O(t \log n)$.

A summary of the results is shown in Table 1.4.

Remarks. In the above discussion, we assume that the alphabet Σ has a constant size. For a variable size alphabet, our compressed index occupies $O(n|\Sigma|)$ bits, which may become a problem if $|\Sigma|$ is huge. Nevertheless, our compressed index based on CSA alone achieves a space complexity of $O(n \log |\Sigma|)$ bits, but with update and pattern searching suffering a slowdown by a logarithmic factor.

Table 1.4: Managing a collection of texts for library matching query.

	Description	Space (bits)	Library Matching Time	Insertion/Deletion Time
stat.	via suffix tree	$O(n \log n)$	$O(p + occ)$	
	via CSA	$O(n \log \lvert\Sigma\rvert)$	$O(p \log\log \lvert\Sigma\rvert + occ \log^{\epsilon} n)$	
	via FM-index ($\lvert\Sigma\rvert = O(1)$)	$O(nH_k)$	$O(p + occ \log^{\epsilon} n)$	
dyn.	via suffix tree	$O(n \log n)$	$O(p + occ)$	$O(t)$
	via FM-index ($\lvert\Sigma\rvert = O(1)$)	$O(nH_k)$	$O(p \log^3 n + occ \log n)$	amor. $O(t \log n)/O(t \log^2 n)$
	this thesis ($\lvert\Sigma\rvert = O(1)$)	$O(n)$	$O(p \log n + occ \log^2 n)$	$O(t \log n)$
	this thesis	$O(n \log \lvert\Sigma\rvert)$	$O(p \log^2 n + occ \log^2 n)$	$O(t \log^2 n)$

Managing a Dynamic Dictionary

The *dynamic dictionary management* problem forms a dual with the dynamic library management problem, where the former one deals with indexing a collection $\mathcal{D}$ of patterns $\{P_1, P_2, \ldots, P_k\}$ with total length d, so as to answer efficiently the occurrences of all P_i in any given text T (which is called a *dictionary matching* query), and allow efficient insertion and deletion of patterns. This problem is well studied in the literature [2, 4, 7, 5, 6, 76], and most of the previous solutions are based on suffix trees. In particular, Amir *et al.* [6] showed that updating a pattern of length p can be done in $O(p \log d / \log\log d)$ time and a dictionary matching query for a text T of length t takes $O((t + occ) \log d / \log\log d)$ time. Later, Sahinalp and Vishkin [76] devised a new data structure called *fat-tree*, and improved the update time to $O(p)$, and query time to $O(t + occ)$. The space required by both data structures is $O(d \log d)$ bits, which becomes impractical when d is large.

As far as we know, no one has considered the setting under the compressed storage requirement of $O(d \log \lvert\Sigma\rvert)$ bits. In this thesis, we present a compressed solution for a special case of this problem, where we assume that no pattern in $\mathcal{D}$ is a proper substring of the other. Our solution is based on the CST, and it supports dictionary matching in $O((t \log^2 d)(\log^{\epsilon} d + \log \lvert\Sigma\rvert) + occ \log^{2+\epsilon} n)$ time for any fixed $\epsilon > 0$. For updating of a pattern, both insertion and deletion can be done in $O(p \log^{2+\epsilon} d)$ amortized time. See Table 1.5 for a summary of the results.

Table 1.5: Managing a dynamic dictionary.

	Description	Space (bits)	Dictionary Matching Time	Insertion/Deletion Time
existing	via suffix tree [6]	$O(d \log d)$	$O((t + occ) \log d / \log \log d)$	$O(p \log d / \log \log d)$
	via fat-tree [76]	$O(d \log d)$	$O(t + occ)$	$O(p)$
this thesis	via CST (special case)	$O(d \log \lvert\Sigma\rvert)$	$O((t + occ) \log^3 d)$	amor. $O(p \log^{2+\epsilon} d)$

Managing a Dynamic Text

In the above discussion, we have discussed the simple text searching problem, in which we need to maintain a single piece of static text, and the library management problem where we need to maintain a dynamic collection of texts. Another related problem is to maintain a single piece of text which is subject to update over the times. This problem is useful in managing DNA texts, as they are frequently updated due to errors in sequencing process.

Ferragina and Grossi [24] proposed an *interval partitioning* scheme to exploit the *generalized suffix tree* to give an index that occupies $O(n \log n)$ bits of space where n is the length of the text. It supports searching of a pattern P of length p in $O(p+occ)$ time. In addition, it supports insertion (and deletion) of a substring of length y at an arbitrary position in T in $O(y + \sqrt{n})$ time. Later, Sahinalp and Vishkin [76] proposed the fat-tree and further improved the insertion and deletion time to $O(y + \log^3 n)$.

It was open whether there is a compressed index (i.e., using $O(n \log |\Sigma|)$ bits) that can manage a dynamic text efficiently. In this thesis, we report the progress of this dynamic problem. Precisely, we propose an index that occupies $O(n \log |\Sigma|)$ bits of space for any fixed $\epsilon > 0$, while supporting pattern searching in $O((p \log^2 n)(\log^\epsilon n + \log |\Sigma|) + occ \log^{2+\epsilon} n)$ time, and insertion/deletion of a substring of length y in $O((y + \sqrt{n}) \log^{2+\epsilon} n)$ amortized time.

Briefly speaking, we make use of the interval partitioning technique in [24] to reduce the dynamic text problem into the dynamic dictionary management and the dynamic library management problems. Then, applying the compressed solutions to the latter two problems, we produce the required compressed index. A summary of the results are shown in Table 1.6.

Table 1.6: Managing a dynamic text.

	Description	Space (bits)	Searching Time	Insertion/Deletion Time
existing	via suffix tree [24]	$O(n \log n)$	$O(p + occ)$	$O(y + \sqrt{n})$
	via fat-tree [76]	$O(n \log n)$	$O(p + occ)$	$O(y + \log^3 n)$
this thesis	via CSA and CST	$O(n \log \lvert\Sigma\rvert)$	$O((p + occ) \log^3 n)$	amor. $O((y + \sqrt{n}) \log^{2+\epsilon} n)$

1.3.3 Experimental Results on the Practical Aspects of CSA and FM-index

While theoretical bounds are important, the success of a data structure is often measured in terms of its performance in practice. Indeed, both CSA and FM-index have demonstrated their practicality for text indexing in the literature. For instance, for the DNA sequence *E. coli*, Ferragina and Manzini have shown in their experimental paper [26] that the corresponding FM-index occupies $2.689n$ bits, while its performance is comparable to that of the suffix arrays when searching a pattern whose length is short (8-15 chars). On the other hand, Grossi, Gupta and Vitter [33] have shown that the most space-efficient variant of the CSA [32] can be implemented in $2.392n$ bits while supporting fast searching queries. Moreover, the performance of these indexes are also tested extensively across various texts, whose lengths vary from 4 million to 70 million characters. Nevertheless, these lengths cannot yet cover some of the popular DNA sequences such as fruit fly, human, fugu, and rice. One possible reason may be due to the lack of a space-efficient construction algorithm for these indexes as proposed in the previous section. This, together with some other issues to be explained later, motivate us to study the practical aspects of the CSA and FM-index from a different point of view. In this thesis, we attempt to find out the answers to the following questions:

1. What is the largest DNA sequence whose CSA and FM-index can be constructed in main memory? Will the construction time be acceptable?
2. Previous studies on searching focus on short patterns. In real life, DNA sequences are often searched against genes whose lengths are much longer. Will the searching performance in this case be consistent with that for searching short patterns? Also, will the length of the DNA sequences affect the performance?

3. In the literature, there are two types of searching methodology for CSA or FM-index. One of them is called *forward search*, which is the classical approach for suffix arrays. The other one is called *backward search*, which is the method tailored for CSA or FM-index and is better than forward search in theory. However, in practice, will backward search always beat forward search?

4. Finally, can we conclude which one of the two indexes is best-suited for indexing DNA sequences?

We conducted experiments on construction and searching performances with an ordinary PC equipped with a 1.7 GHz Pentium IV processor with 256 Kbytes of L2 cache, and 4 Gbytes of RAM. The operating system was Solaris 9. Note that this modest configuration can easily be acquired by most research laboratories nowadays. Our results can be briefly summarized as follows. For construction limits, we have successfully construct the CSA and FM-index for DNA sequences of length up to 3 Gbases. The construction times are 24 and 28 hours, respectively. For the searching performance, we have constructed the CSA and the FM-index for E. coli (4.6 Mbases), Fly (98 Mbases) and Human (2.88 Gbases). In each setting, we tested the searching times (for both forward search and backward search) using patterns of length from 10 to 10,000, where the patterns are extracted from random positions in the corresponding DNA sequence to boost the worst-case performance. From our experiments, we find that backward search is sensitive to the length of the pattern, while forward search is not. On the other hand, searching different DNAs against patterns of similar length shows similar timing, indicating that the length of the DNA has little effect on the searching performance.

For the comparison between forward search and backward search, we observe that using backward search, FM-index is consistently faster than CSA. However, using forward search, CSA is faster than FM-index. The most surprising result is that, for long patterns, forward search is more efficient than backward search. Roughly speaking, for patterns of length less than 2000, FM-index with backward search is most efficient; otherwise, CSA with forward search is fastest, while FM-index with forward search is comparable. See Figure 1.1 for the timing of the experiments on the CSA and FM-index of Human, with each index occupying about 2.2 Gbytes ($6n$ bits) of space.

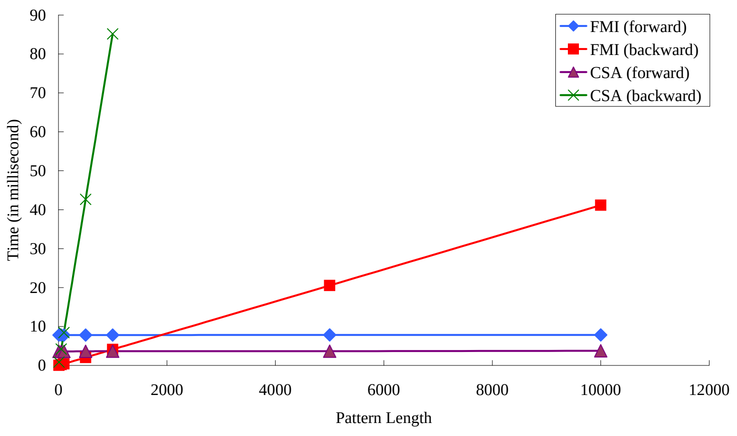


Figure 1.1: Searching performance of CSA and FM-index of Human. The space occupancy for each index is about 2.2 Gbytes.

In summary, our experiments suggested that FM-index is faster than CSA for searching short patterns, while for long patterns, CSA is better.

1.4 Organization of this Thesis

This thesis is organized as follows. In the next chapter, we describe the data structures of suffix trees, suffix arrays, CSA and FM-index, and define some basic notations. Chapters 3 and 4 respectively discuss the two construction algorithms for CSA. Chapter 5 gives the definition of CST, shows the conversion from CSA to CST, and describes how to apply CST to find MUM's efficiently. In Chapter 6, we present the compressed solution for maintaining a dynamic library; then, in Chapter 7, we present the compressed solutions for maintaining a dynamic dictionary and a dynamic text. Chapter 8 shows our experimental studies of the CSA and FM-index. Finally, concluding remarks are given in Chapter 9.

Chapter 2

Preliminaries

In this chapter, we review the definition of the suffix tree, the suffix arrays, the compressed suffix arrays (CSA), and the FM-index, and introduce some notations to be used throughout the thesis. In addition, some simple observations on these data structures are presented.

Let T be a text over an alphabet Σ. We assume that T is ended with a special symbol `$` not appearing anywhere else inside the text, and `$` is lexicographically smaller than the other characters in Σ. Let n be the number of characters (including `$`) in T. We assume that T is stored in an array $T[0, n-1]$, where $T[n-1] = \$$. For any integer i in $[0, n-1]$, we denote

- $T[i]$ as the $(i+1)$-th character of T from the left; and
- T_i as the suffix of T starting from position i; that is, $T_i = T[i, n-1] = T[i]T[i+1]...T[n-1]$.

Furthermore, let $\mathcal{S}(T)$ denote the set of all suffixes of T, $\{T_0, T_1, \ldots, T_{n-1}\}$.

For instance, if $T = \mathsf{acaaccg}\$$, then $\mathcal{S}(T) = \{\mathsf{acaaccg}\$, \mathsf{caaccg}\$, \mathsf{aaccg}\$, \mathsf{accg}\$, \mathsf{ccg}\$, \mathsf{cg}\$, \mathsf{g}\$, \$\}$.

2.1 Suffix Tree

The *suffix trie* of T is a trie built on $\mathcal{S}(T)$. Formally, it is a rooted, directed and ordered tree with exactly n leaves, with each edge labeled by a character in Σ; in addition, no two edges out of a node can have the same label, and the children of a node are ordered by the lexicographical order of the edge label of their incident edges; finally, by concatenating the characters on the edges along the path from the root to any leaf, we obtain a distinct suffix of T. Figure 2.1 shows the suffix trie of the text `acaaccg$`.

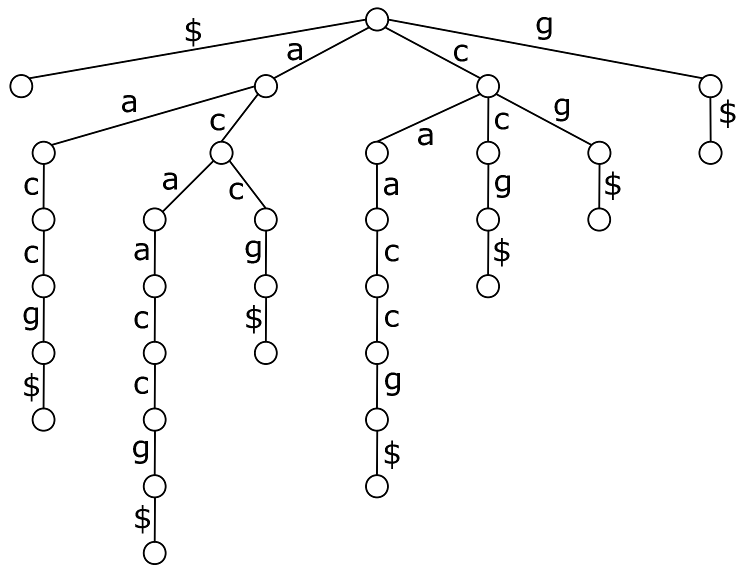


Figure 2.1: The suffix trie of the text $T =$ `acaaccg$`.

It is easy to check that a pattern P of length p occurs in the text T at position i (that is, exactly matching the substring $T[i, i+p-1]$) if and only if P is the prefix of the suffix T_i. Exploiting this fact, we can traverse the suffix trie based on P to determine whether P occurs in T or not. The idea is that we start from the root, and set the current character to the first character of P. During the traversal where we are at a particular node, we try to find the outgoing edge whose label matches the current character of P. If such an edge is found, we follow the edge to the child node, and advance the current character to the next character of P; otherwise, the traversal stops. In the end, if the traversal succeeds in matching all the characters in P, this indicates that P is the prefix of some suffix of T, thus implying P occurs in T; otherwise, P is not a prefix of any suffix of T, then by the fact, we can conclude that P does not occur in T.

A suffix trie usually requires large amount of storage, as it contains $\Omega(n^2)$ nodes and edges in the worst case. This leads to the definition of the *suffix tree*, where we contract all the degree-1 internal nodes (except the root) in the suffix trie, making the tree to contain $O(n)$ nodes and edges only. Note that each edge of the suffix tree is now labeled with a non-empty substring of T instead of a single character, and the labels of different edges out of the same node are starting with different characters. Also, the children of a node is now ordered by the lexicographical order of the first character of the edge label of their incident edges. In addition, we label each leaf by the starting position of the corresponding suffix. See Figure 2.2 for the suffix tree of the text `acaaccg$`.

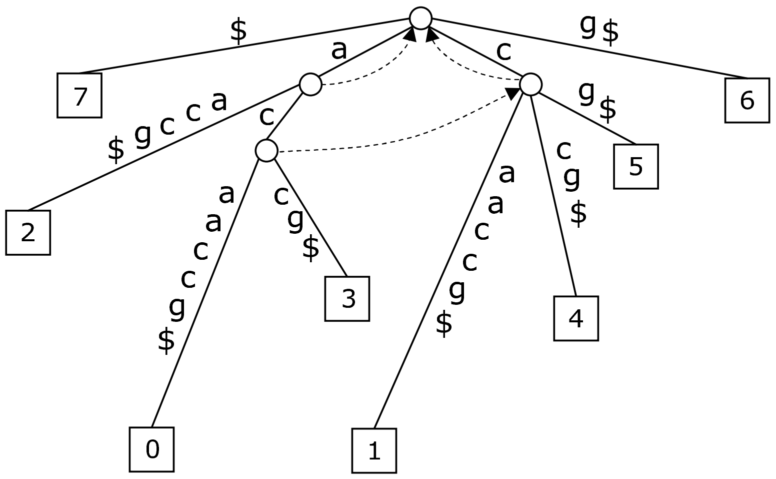


Figure 2.2: The suffix tree of the text T = `acaaccg$`. The suffix link of an internal node is shown by its outgoing dashed arrow.

For pattern matching, we proceed as if we are using a suffix trie, and determine whether P occurs in T or not first. If so, suppose that the traversal last visits the edge (u, v), where u is the parent of v in the suffix tree. Then, it is easy to verify that the subtree rooted at v will contain the leaves of all suffixes having P as the prefix. Now, by visiting these leaves and retrieving their labels, we can obtain all positions where P occurs in T. Thus, pattern matching can be done using $O(p)$ character comparisons, $O(p)$ retrieval of edge labels, $O(p + occ)$ traversal operations, and $O(occ)$ retrieval of leaf labels, where occ denotes the number of occurrences. Under an $O(|\Sigma| n \log n)$-bit implementation of a suffix tree, the overall time complexity is $O(p + occ)$, as each basic operation can be performed in constant time.

A suffix tree is usually augmented with some short-cuts between internal nodes, called *suffix links*, which are key ingredients in all existing efficient construction algorithms of a suffix tree. Suffix links also find usage in solving complicated text queries. Formally, its definition is as follows. For an internal node u, let *path label* of u be the concatenation of edge labels along the path from the root to u. (The path label of the root is defined to be the empty string.) The suffix link of an internal node u (excluding the root) is a link from u to some node v, such that the path label of v is equal to the path label of u but with the first character removed. For example in Figure 2.2, the suffix link of the node with path label `ac`, is the link from that node to the one with path label `c`. It is well-known that the suffix link of any internal node is well-defined. In Chapter 5, we will show how suffix links can enhance a suffix tree to solve some complicated text queries.

2.2 Suffix Arrays

The *suffix array* of T, denoted $\mathrm{SA}[0, n-1]$, is a representation for the sorted sequence of the suffixes of T. Formally, $\mathrm{SA}[i]$ stores the starting position of the $(i+1)$-th smallest suffix of T. In other words, according to the lexicographical order, $T_{\mathrm{SA}[0]} < T_{\mathrm{SA}[1]} < \ldots < T_{\mathrm{SA}[n-1]}$. See Figure 2.3 for an example. Note that $\mathrm{SA}[0] = n-1$, and the suffix array can be obtained by visiting the leaves in the suffix tree from left to right.

i	T_i	rank	$\mathrm{SA}^{-1}[i]$	i	$\mathrm{SA}[i]$	$T_{\mathrm{SA}[i]}$
0	acaaccg\$	2	2	0	7	\$
1	caaccg\$	4	4	1	2	aaccg\$
2	aaccg\$	1	1	2	0	acaaccg\$
3	accg\$	3	3	3	3	accg\$
4	ccg\$	5	5	4	1	caaccg\$
5	cg\$	6	6	5	4	ccg\$
6	g\$	7	7	6	5	cg\$
7	\$	0	0	7	6	g\$

Figure 2.3: The suffix array of `acaaccg$`, its inverse and the rank of a suffix.

Given a text T together with the suffix array $\mathrm{SA}[0, n-1]$, the occurrences of any pattern P in T can be found without scanning T again. The idea is to perform p (the length of P) binary search steps on the suffix array. Recall that suffixes are lexicographically sorted in the suffix array, so that suffixes sharing the same prefix will be placed at consecutive region in SA. We then find the maximal region $[\ell_0, r_0]$ in SA such that the corresponding suffixes shares the same prefix $P[0]$. This can be achieved by binary search on SA. Afterwards, we proceed similarly as follows. At the beginning of j-th binary search step, we have already determined the maximal region $[\ell_{j-1}, r_{j-1}]$ in SA such that the corresponding suffixes share the same prefix $P[0, j-1]$. Then, by a binary search on SA in the region $[\ell_{j-1}, r_{j-1}]$, we can find the maximal region where the $(j+1)$-th character of these suffixes is equal to $P[j]$. That is, after this binary search step, we obtain the maximal region $[\ell_j, r_j]$ in SA such that the corresponding suffixes share the same prefix $P[0, j]$. (Note that if no such region exists, we can declare P does not occur in T.) Thus, after p binary search steps, we obtain the region $[\ell_{p-1}, r_{p-1}]$, and all occurrences of P can be found by reading $\mathrm{SA}[i]$ for all i in this region.

The above pattern matching process requires $O(p \log n + occ)$ retrieval of entries in the suffix array, and $O(p \log n)$ retrieval of characters in T. Under an $O(n \log n)$-bit implementation of suffix array, each of the retrieval can be done in constant time, so that the overall time complexity is $O(p \log n + occ)$.

If the suffix array is coupled with an *LCP* array that stores the length of the longest common prefix between two suffixes that are adjacent in the *binary search* order, all the $p \log n$ terms can be improved to $p + \log n$. For more details, see Manber and Myers [56].

For every i in $[0, n-1]$, define $\mathrm{SA}^{-1}[i]$ to be the integer j such that $\mathrm{SA}[j] = i$. Intuitively, $\mathrm{SA}^{-1}[i]$ denotes the rank of T_i among the suffixes of T, which is the number of suffixes of T lexicographically smaller than T_i. We use the notation $\mathrm{Rank}(X, \mathcal{S})$ to denote the rank of X among a set of strings $\mathcal{S}$. Thus, $\mathrm{SA}^{-1}[i] = \mathrm{Rank}(T_i, \mathcal{S}(T))$. See Figure 2.3 for an example.

2.3 Compressed Suffix Arrays (CSA)

Based on SA and SA^{-1}, the compressed suffix arrays (CSA) of a text T is an array $\Psi[0, n-1]$ where $\Psi[i] = \mathrm{SA}^{-1}[\mathrm{SA}[i] + 1]$ for $i = 1, 2, \ldots, n-1$, whereas $\Psi[0]$ is defined as $\mathrm{SA}^{-1}[0]$. In other words, if T_k is the suffix with rank i, $\Psi[i]$ is the rank of the suffix T_{k+1}. See Figure 2.4 for an example. It is worth-mentioning that Ψ can be used to recover SA^{-1} iteratively: $\mathrm{SA}^{-1}[1] = \Psi[\Psi[0]]$, $\mathrm{SA}^{-1}[2] = \Psi[\Psi[\Psi[0]]]$, ..., etc.

Note that $\Psi[0, n-1]$ contains n integers. A trivial way to store the array requires $n \log n$ bits, which is the same space as SA. Nevertheless, the Ψ array can be decomposed into $|\Sigma|$ strictly increasing sequences, which allows it to be stored succinctly. This increasing property is based on the following lemmas.

Lemma 2.1 *For every $i < j$, if $T[\mathrm{SA}[i]] = T[\mathrm{SA}[j]]$, then $\Psi[i] < \Psi[j]$.*

i	SA[i]	SA^{-1}[i]	$\Psi[i]$	T[SA[i]]
0	7	2	2	`$`
1	2	4	3	`a`
2	0	1	4	`a`
3	3	3	5	`a`
4	1	5	1	`c`
5	4	6	6	`c`
6	5	7	7	`c`
7	6	0	0	`g`

Figure 2.4: The CSA of `acaaccg$`. Note that the Ψ array can be partitioned into $|\Sigma|$ $(= 4)$ increasing sequences based on T[SA[i]], namely, $\Psi[0, 0]$, $\Psi[1, 3]$, $\Psi[4, 6]$, and $\Psi[7, 7]$.

Proof: Note that $i < j$ if and only if $T_{\mathrm{SA}[i]} < T_{\mathrm{SA}[j]}$. This implies that if $i < j$ and $T[\mathrm{SA}[i]] = T[\mathrm{SA}[j]]$, $T_{\mathrm{SA}[i]+1} < T_{\mathrm{SA}[j]+1}$. Equivalently, we have $T_{\mathrm{SA}[\Psi[i]]} < T_{\mathrm{SA}[\Psi[j]]}$. Thus, $\Psi[i] < \Psi[j]$ and the lemma follows. □

For each character c, let $\alpha(c)$ denote Rank$(c, \mathcal{S}(T))$, which is the number of suffixes starting with a character lexicographically smaller than c. Also, let $\#(c)$ be the number of suffixes starting with c.

Corollary 2.2 *For each character c, $\Psi[\alpha(c), \alpha(c) + \#(c) - 1]$ gives a strictly increasing sequence.*

Proof: For any character c, $T[\mathrm{SA}[\alpha(c)]] = T[\mathrm{SA}[\alpha(c)+1]] = \cdots = T[\mathrm{SA}[\alpha(c) + \#(c)-1]] = c$. By Lemma 2.1, Ψ is strictly increasing in $\Psi[\alpha(c), \alpha(c)+\#(c)-1]$. □

See Figure 2.4 for an illustration.

Based on the above increasing property, Grossi and Vitter [36] devised the following scheme to store Ψ in $n(H_0 + 2)$ bits, where $H_0 \le \log |\Sigma|$ is the entropy of the text T. For each character c, the sequence $\Psi[\alpha(c), \alpha(c) + \#(c) - 1]$ is represented using Rice code [70]. That is, each $\Psi[i]$ in the sequence is divided into two parts q_i and r_i, where q_i is the first (or most significant) $\log \#(c)$ bits, and r_i is the remaining $\log(n/\#(c))$ bits. The r_i's are stored explicitly in an array of size $\#(c) \log(n/\#(c))$ bits. For the q_i's, since they form a monotonic increasing sequence bounded by 0 and $\#(c) - 1$, we store $q_{\alpha(c)}$, and the difference values $q_{i+1} - q_i$ for i in $[\alpha(c), \alpha(c) + \#(c) - 2]$ using unary codes,* which requires $2\#(c)$ bits. In total, the space required is $\sum_{c \in \Sigma} \#(c)(\log(n/\#(c)) + 2)$, which is $n(H_0 + 2)$ bits. Together with an auxiliary data structure of $o(n)$ bits, each Ψ value can be retrieved in $O(1)$ time [42, 61]. In addition, another $O(n(H_0+1))$-bit auxiliary data structure allows any SA$[i]$ to be retrieved in $O(\log^\epsilon n)$ time, for any $0 < \epsilon \leq 1$. We summarize the above discussion in the following lemma.

Lemma 2.3 *Ψ can be represented using $n(H_0+2+o(1))$ bits, while allowing any Ψ value to be retrieved in $O(1)$ time. With an additional auxiliary data structure of size $O(n(H_0 + 1))$ bits, any* SA$[i]$ *value can be retrieved in $O(\log^\epsilon n)$ time, for any $0 < \epsilon \leq 1$. If we can enumerate the values of $\Psi[i]$ sequentially, this representation can be constructed incrementally using $O(n)$ time without extra space.*

There are also some other ways for storing the Ψ array succinctly. In later chapters, we will specify clearly whenever a different scheme is used.

Note that Lemma 2.3 immediately implies an $O(p \log^{1+\epsilon} n + occ \log^\epsilon n)$-time algorithm for finding all occurrences of P in T. Sadakane [72] observed that using an extra $\min\{n, |\Sigma| \log n\}$ bits of space, the $p \log^{1+\epsilon} n$ term can be improved to $p \log n$ instead, based on the pattern matching algorithm of FM-index. We defer the details to the next section, where we review the FM-index. Furthermore, in Chapter 4, we show that if we are allowed to use yet another $O(n)$ bits of extra space, the $p \log n$ term can further be improved to $p \log \log |\Sigma|$.

2.4 FM-index

The Burrows-Wheeler transformation transforms a text T into another text W, such that $W[i] = T[\text{SA}[i] - 1]$ if $\text{SA}[i] > 0$, and $W[i] = \$$ if $\text{SA}[i] = 0$. The FM-index basically consists of $|\Sigma|$ functions, called *count*, that are defined based on the transformed text W. For each character c in Σ and $i = 0, \ldots, n-1$, the function *count*(c, i) is the number of character c appearing in $W[0, i-1]$. See Figure 2.5 for an example.

Now, we state a lemma that demonstrates the pattern searching ability of the *count* functions.

*The unary code for an integer $x \geq 0$ is encoded as x `0`'s followed by a `1`.

i	SA[i]	$T_{\mathrm{SA}[i]}$	W	*count*('\$',i)	*count*('a',i)	*count*('c',i)	*count*('g',i)
0	7	\$	g	0	0	0	1
1	2	aaccg\$	c	0	0	1	1
2	0	acaaccg\$	\$	1	0	1	1
3	3	accg\$	a	1	1	1	1
4	1	caaccg\$	a	1	2	1	1
5	4	ccg\$	a	1	3	1	1
6	5	cg\$	c	1	3	2	1
7	6	g\$	c	1	3	3	1

Figure 2.5: The Burrows-Wheeler text W, and the *count* functions.

Lemma 2.4 ([25]) *Let P be any pattern and let c be any character over Σ. Denote the rank of P among all suffixes of T as i. Then, the rank of cP among all suffixes of T is equal to $count(c, i) + \alpha(c)$, where $\alpha(c)$ denotes the number of suffixes of T starting with a character less than c.*

We refer to an execution of the above lemma a *backward search step*. Suppose that $\alpha(c)$ can be reported in constant time. Applying backward search steps repeatedly, we can find the rank of any pattern P among the suffixes of T using $O(p)$ queries to the *count* function. This method can readily be extended to find the maximal region $[\ell, r]$ in the suffix array such that the corresponding suffixes share P as their prefix, and it is known as the *backward search* algorithm in the literature.

When the alphabet size is small, precisely, $|\Sigma| \log |\Sigma| = O(\log n)$, Ferragina and Manzini [25] gave a $5nH_k + o(n)$-bit implementation of FM-index that can support any *count* query in constant time. In addition, if FM-index is augmented with an $o(n)$-bit auxiliary data structure, each entry of the suffix array can be reported in $O(\log^{1+\epsilon} n)$ time, for $0 < \epsilon \leq 1$. This gives the follow lemma.

Lemma 2.5 ([25]) *Suppose that $|\Sigma| \log |\Sigma| = O(\log n)$. We can store the FM-index of T in $5nH_k + o(n)$ bits, which supports searching of any pattern P in T in $O(p + occ \log^{1+\epsilon} n)$ time, where $0 < \epsilon \leq 1$ and occ denotes the number of occurrences.*

2.4.1 Relation between CSA and FM-index

The CSA and the FM-index are in fact closely related. In this section, we give two lemmas that relates the Ψ function of CSA and the *count* function of FM-index.

Firstly, recall the following notation from Section 2.3. For each character c, let $\alpha(c)$ denote $\text{Rank}(c, \mathcal{S}(T))$, which is the number of suffixes starting with a character lexicographically smaller than c. Also, let $\#(c)$ be the number of suffixes starting with c.

Note that $\alpha(c)$ and $\#(c)$ can be stored in $O(n)$ bits using the succinct indexable dictionary of Raman, Raman and Rao [68], while maintaining constant time retrieval. In addition, with this indexable dictionary, $T[\text{SA}[i]]$ can be retrieved in constant time.

Suppose that we are given this indexable dictionary. Then, we have the following lemmas.

Lemma 2.6 *For any $i = 0, 1, \ldots, n-1$, we can compute $\Psi[i]$ using $O(\log n)$ queries to the count function.*

Proof: Let $c = T[\text{SA}[i]]$ and $y = i - \alpha(c)$. Both c and y can be computed in constant time based on the indexable dictionary. Now, suppose that the following claim is correct: $W[\Psi[i]]$ is the $(y+1)$-th c in the transformed text W. Then, $\Psi[i]$ is the smallest k such that $\mathit{count}(c, k) = y + 1$. As $\mathit{count}(c, \cdot)$ is monotonic increasing, the value of k (and thus $\Psi[i]$) can be found based on binary search, using $O(\log n)$ queries to the *count* function.

To prove the claim, we first show that $W[\Psi[i]] = c$. This is true since $W[\Psi[i]] = T[\text{SA}[i]]$. Next, we observe that for each suffix of T that begins with c, say, of the form $cT_{\text{SA}[j]}$, it is lexicographically smaller than or equal to $T_{\text{SA}[i]}$ if and only if $j \leq \Psi[i]$ and $W[j] = c$. In other words, if r denotes the rank of the suffix $T_{\text{SA}[i]}$ among all suffixes of T that begin with c, $W[\Psi[i]]$ is the $(r+1)$-th c in W.

Clearly, $r = i - \alpha(c) = y$. This completes the proof of the claim, and the lemma follows. □

Lemma 2.7 *For any c in Σ and $i = 0, 1, \ldots, n-1$, we can compute count(c, i) using $O(\log n)$ queries to Ψ.*

Proof: Observe that $T[\text{SA}[j]] = c$ if and only if $W[\Psi[j]] = c$. This implies that $\mathit{count}(c, i)$ is the number of j satisfying $T[\text{SA}[j]] = c$ and $\Psi[j] \leq i$.

However, for $T[\mathrm{SA}[j]] = c$, j must be in the region $[\alpha(c), \alpha(c) + \#(c) - 1]$. Thus, *count*$(c, i)$ is equal to the number of j in $[\alpha(c), \alpha(c) + \#(c) - 1]$ satisfying $\Psi[j] \leq i$.

By Corollary 2.2, $\Psi[\alpha(c), \alpha(c) + \#(c) - 1]$ is an increasing sequence. Thus, *count*(c, i), can be found by a binary search on the sequence $\Psi[\alpha(c), \alpha(c) + \#(c) - 1]$, using $O(\log n)$ queries to Ψ. $\square$

Combining Lemma 2.3, Lemma 2.4, Lemma 2.7 and the succinct indexable dictionary of [68], we get the following corollary.

Corollary 2.8 *We can store the CSA of T in $O(n(H_0 + 1))$ bits, which supports searching of any pattern P in T in $O(p \log n + occ \log^{\epsilon} n)$ time, where $0 < \epsilon \leq 1$ and occ denotes the number of occurrences.*

Chapter 3

Constructing CSA in Optimal Space

In this chapter, we present an $O(n \log n)$-time construction algorithm for the CSA defined in [36]. The space requirement is $O(n(H_0+1))$ bits, which is optimal since the CSA alone takes $n(H_0+2)$ bits. The chapter is divided into two parts. In the first part, we investigate the relationship between the CSA's of two consecutive suffixes, which provides insight for constructing CSA incrementally by processing one character at a time. Then, in the second part, we extend this relationship to give the main algorithm, which runs faster by allowing the incremental algorithm to process more characters at a time.

3.1 CSA of Two Consecutive Suffixes

This section serves as a warm up to the main algorithm presented in the next section. In particular, we investigate the relationship between the CSA of two consecutive suffixes. Then, based on this relationship, we demonstrate an algorithm that constructs the CSA for a text T, in an incremental manner. Since this algorithm is not the main result of this thesis, we only give the high-level description. One can refer to [51] for the implementation details.

Let T be a string with n characters. We assume that T is represented by an array $T[0, n-1]$ and $T[n-1] = \$$. Let SA_T and Ψ_T be the suffix array and CSA of T, respectively.

Suppose that we are given the CSA of T, and we want to construct the CSA for a longer text $T' = cT$, where c is a character. Let $\mathrm{SA}_{T'}$ and $\Psi_{T'}[0, n]$ denote the suffix array and the CSA of T', respectively. To see the relationship between the CSA of T and T', we first show that the suffix array of T' can be easily obtained from that of T.

Recall that SA_T is a sequence of the starting positions of the suffixes of T, sorted according to their ranks. Except T' itself, T' shares all its suffixes with T; thus, $\mathrm{SA}_{T'}$ has exactly one more entry than SA_T, which is due to the suffix T'. Intuitively, to obtain $\mathrm{SA}_{T'}$, we can insert the suffix T' (which is represented by the starting position 0) into T. Let $x = \mathrm{Rank}(T', \mathcal{S}(T))$ be the rank of T' among the set of suffixes $\mathcal{S}(T)$. T' should be inserted between $\mathrm{SA}_T[x-1]$ and $\mathrm{SA}_T[x]$. Also, since a character is added to the beginning of T, we increment every entry of SA_T by 1 to reflect the change in their starting position. Thus, we have the following lemma.

Lemma 3.1 *Let* $x = \mathrm{Rank}(T', \mathcal{S}(T))$. *Then,*

$$\mathrm{SA}_{T'}[i] = \begin{cases} \mathrm{SA}_T[i] + 1 & \textit{if } 0 \le i \le x-1 \\ 0 & \textit{if } i = x \\ \mathrm{SA}_T[i-1] + 1 & \textit{if } i \ge x+1 \end{cases}$$

Based on Lemma 3.1, we observe the relationship between the CSA of T and T' as follows.

Lemma 3.2 *Let* $x = \mathrm{Rank}(T', \mathcal{S}(T))$. *Then,*

- $\Psi_{T'}[0] = x$;
- *for* $1 \le i < x$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[i] & \textit{if } \Psi_T[i] < x \\ \Psi_T[i] + 1 & \textit{if } \Psi_T[i] \ge x \end{cases}$;
- *for* $i = x$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[0] & \textit{if } \Psi_T[0] < x \\ \Psi_T[0] + 1 & \textit{if } \Psi_T[0] \ge x \end{cases}$;
- *for* $x < i \le n$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[i-1] & \textit{if } \Psi_T[i-1] < x \\ \Psi_T[i-1] + 1 & \textit{if } \Psi_T[i-1] \ge x \end{cases}$.

The above lemma suggests that we can compute $\Psi_{T'}$ from Ψ_T as follows.

```
1. Compute $x$ = the rank of $T'$ among all suffixes of $T$.
2. Set $\Psi_{T'}[0] = x$.
3. For $1 \le i \le n$, set $\Psi_{T'}[i] = \begin{cases} \Psi_T[i] & \text{if } i < x \\ \Psi_T[0] & \text{if } i = x \\ \Psi_T[i-1] & \text{if } i > x \end{cases}$
4. For each $1 \le i \le n$, if $\Psi_{T'}[i] \ge x$, increment $\Psi_{T'}[i]$ by one.
```

To build the CSA for a text T of length n starting from scratch, we can execute the above algorithm repeatedly, constructing the CSA for the suffixes $T_{n-1}, T_{n-2}, \cdots, T_0$ incrementally. Each such execution can be implemented in $O(n)$ time. Thus, we can construct the CSA Ψ_T for $T[0, n-1]$ using $O(n^2)$ time. In the next section, we will present the details of an $O(n \log n)$-time algorithm for constructing the CSA. The idea is that, instead of updating Ψ every time a character is added, we collectively perform the update for every 'segment'. This gives an incremental algorithm which processes the text in a 'segment by segment' manner.

3.2 Incremental Construction Algorithm

In this section, we show how to compute $\Psi[0, n-1]$ for the text T incrementally, in a 'segment by segment' manner. To do so, we first partition the text into $\lceil n/\ell \rceil$ consecutive segments $T^1, T^2, \ldots, T^{\lceil n/\ell \rceil}$, where $\ell = \Theta(n/\log n)$ will be specified later; each segment, except the last one, contains ℓ characters, i.e., T^i refers to the string represented by $T[(i-1)\ell, i\ell - 1]$. The algorithm builds the Ψ array of T incrementally, starting with that of $T^{\lceil n/\ell \rceil}$, and then constructs the Ψ array of $T^{\lceil n/\ell \rceil - 1} T^{\lceil n/\ell \rceil}$ and so on. Eventually the Ψ array of $T^1 T^2 \ldots T^{\lceil n/\ell \rceil} = T$ is constructed. The construction time required for each segment is $O(\ell \log n + n) = O(n)$ time, and the overall time is $O(n \log n)$, which is independent of $|\Sigma|$. The space required is $O(n(H_0 + 1))$ bits (precisely, $n(H_0 + 3 + \epsilon + o(1))$ bits for any $0 < \epsilon < 1$).

Recall from the last section that, when we construct the CSA character by character, the key point is to compute the rank of the newly added suffix among the existing ones, and alter the existing Ψ array accordingly. Indeed, when we construct the CSA segment by segment, the idea is similar. To cater for a new segment, we again compute the rank of all newly added suffixes among the existing ones. Intuitively, this is to find all positions where the existing Ψ array needs to be 'expanded' in order to accommodate the new suffixes. However, knowing such rank is not sufficient. We also need the rank of the new suffixes among themselves. Details are as follows.

Consider any i in $[1, \lceil n/\ell \rceil - 1]$. Let B denote the string $T^{i+1}T^{i+2}\cdots T^{\lceil n/\ell \rceil}$. Suppose that we have built Ψ_B, the CSA of B. Let $A = T^iB$. Adding T^i to B introduces ℓ new suffixes; we call them the *long suffixes* of A. The set of the long suffixes are referred to as $\mathcal{LS}(A)$. Other suffixes of A are also suffixes of B, we call them the *short suffixes*. Note that $\mathcal{S}(A) = \mathcal{S}(B) \cup \mathcal{LS}(A)$. To determine the rank of a long suffix x among $\mathcal{S}(A)$, we can compute the rank of x among $\mathcal{S}(B)$ and the rank of x among $\mathcal{LS}(A)$, and then sum them up.

Fact 3.3 *Let x be a long suffix of A (i.e., $x = A_k$ for some $k \in [0, \ell - 1]$). Then* $\text{Rank}(x, \mathcal{S}(A)) = \text{Rank}(x, \mathcal{LS}(A)) + \text{Rank}(x, \mathcal{S}(B))$.

Once the rank of the long suffixes among among $\mathcal{S}(A)$ is known, we can also compute the rank of each short suffix among $\mathcal{S}(A)$ by simply adjusting the rank of a short suffix among $\mathcal{S}(B)$ according to distribution of the long suffixes. To speed up the computation, we exploits a data structure that supports in $O(1)$ time the rank and select operations.

In Sections 3.2.1 and 3.2.2, we show how to compute the values $\text{Rank}(x, \mathcal{LS}(A))$ and $\text{Rank}(x, \mathcal{S}(B))$ for every long suffix x, respectively. In addition, we describe how to store them in a space efficient way to allow fast retrieval. In Section 3.2.3, we give the details of constructing Ψ_A from Ψ_B, and show that the CSA of T can be constructed in $O(n \log n)$ time using $O(n(H_0 + 1))$ bits.

Before going into the details of the incremental construction, we give the details for building the first Ψ array (i.e., the Ψ for $T^{\lceil n/\ell \rceil}$). Note that $T^{\lceil n/\ell \rceil}$ contains at most ℓ characters and a brute force approach for constructing Ψ does not use too much space. Precisely, this Ψ can be obtained easily in $O(\ell \log \ell)$ time using $3\ell \log n$ bits of space as follows. We use three arrays of $\ell \log n$ bits for storing the SA, SA^{-1} and Ψ of $T^{\lceil n/\ell \rceil}$ explicitly. First, we compute the SA for $T^{\lceil n/\ell \rceil}$ by suffix sorting, which takes $O(\ell \log \ell)$ time using $\ell \log n$ bits in addition to that for storing SA [52]. Afterwards, the SA^{-1} can be computed in $O(\ell)$ time. When both SA and SA^{-1} are available, we can construct the Ψ array in $O(\ell)$ time.

3.2.1 Rank of Long Suffixes among Themselves

This section describes how to compute the rank of the ℓ long suffixes of A among themselves (i.e., suffixes in $\mathcal{LS}(A)$). A straightforward method is to sort the suffixes of A and then determine the rank of every suffix of A among themselves. However, this requires $O(n \log n)$ time when $|A| = O(n)$ [52, 56]. In fact, when given Ψ_B, a simple observation shows that it suffices to perform suffix sorting on the prefix $A[0, 2\ell - 1]$ only, and the time is reduced to $O(\ell \log \ell)$. The idea is as follows: if the first ℓ characters of two suffixes (say, A_i and A_j) in $\mathcal{LS}(A)$ are different, their relative order can be decided immediately; otherwise we resolve their relative order by comparing their suffixes starting at the $(\ell+1)$-th character, which are exactly the suffixes of B starting at position i and j (i.e., B_i and B_j). Note that the relative order of B_i and B_j can be deduced from Ψ_B. More precisely, define P and Q to be two arrays of ℓ integers such that for all k in $[0, \ell - 1]$,

- $P[k]$ is the rank of A_k among $\mathcal{LS}(A)$ when only the first ℓ characters are considered;
- $Q[k]$ is the rank of B_k among $\mathcal{S}(B)$.

Fact 3.4 *Consider the tuples $(P[k], Q[k])$ for all $k \in [0, \ell - 1]$. For any long suffix A_h, $\text{Rank}(A_h, \mathcal{LS}(A))$ is equal to the rank of $(P[h], Q[h])$ among these ℓ tuples.*

Suppose that Ψ_B is given. Below we give the details of computing the arrays P and Q. Then, we make use of the above fact to compute the rank of the long suffixes of A among themselves. The results are stored in an array called M. Details are as follows:

Step 1: Computing P. To sort the ℓ long suffixes of A according to their first ℓ characters, we focus on the substring $A[0, 2\ell - 1]$ and apply the suffix sorting algorithm of Larsson and Sadakane [52] for $\log \ell$ rounds, which can figure out the order of the suffixes according to the first $2^{\log \ell} = \ell$ characters. Then, for each k in $[0, \ell - 1]$, we extract the rank of A_k and store it into $P[k]$. The time required is $O(\ell \log \ell)$.

Step 2: Computing Q. For any k in $[0, \ell - 1]$, $Q[k] = \text{Rank}(B_k, \mathcal{S}(B))$, which is equal to $\text{SA}_B^{-1}[k]$. By definition, $\text{SA}_B^{-1}[0] = \Psi_B[0]$, $\text{SA}_B^{-1}[1] = \Psi_B[\Psi_B[0]]$, and in general, $\text{SA}_B^{-1}[k] = \Psi_B^{k+1}[0]$. Thus, we can compute Q by evaluating $\Psi^k[0]$ iteratively for $k = 1, \ldots, \ell$. The time required is $O(\ell)$.

Step 3: Sorting. Consider the tuples $(P[k], Q[k])$ for all k in $[0, \ell{-}1]$. Perform the sorting on these tuples in $O(\ell \log \ell)$ time, Then, for each k in $[0, \ell - 1]$, $M[k]$ is the order of $\text{Rank}(A_k, \mathcal{LS}(A))$.

Time and space requirement: Steps 1-3 altogether require $O(\ell \log \ell)$ time. As to be shown later, we will also need the inverse of M, denoted M^{-1}, which can be computed from M in $O(\ell)$ time. Note that M and M^{-1} each require $\ell \log n$ bits, and the above steps require an additional working space of $2\ell \log n$ bits (for storing P and Q). The total space requirement is $4\ell \log n$ bits.

3.2.2 Rank of Long Suffixes among $\mathcal{S}(B)$

This section shows that if Ψ_B is given, then the rank of the ℓ long suffixes of A among all suffixes of B can be computed in $O(\ell \log n + n)$ time. Apart from Ψ_B, the space required is $\ell \log n$ bits, which is essentially needed for storing the output.

For any character c, let $\#_B(c)$ denote the number suffixes of B starting with c, and let $\alpha_B(c)$ denote the number of suffixes of B whose starting character is lexicographically smaller than c. Note that $O(n)$ time suffices to compute $\#_B(c)$ and $\alpha_B(c)$ for all c. All suffixes of B starting with c have a rank in the range $[\alpha_B(c), \alpha_B(c) + \#_B(c) - 1]$, which is denoted $R_B(c)$ below. The following lemma shows how to determine rank incrementally, i.e., how to derive $\text{Rank}(cX, \mathcal{S}(B))$ from $\text{Rank}(X, \mathcal{S}(B))$ for any string X and character c.

Lemma 3.5 *Consider any string X and any character c. Let $\mathcal{H}$ denote the set $\{r \in R_B(c) \mid \Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))\}$. Then,*

$$\text{Rank}(cX, \mathcal{S}(B)) = \begin{cases} \alpha_B(c) & \text{if } \mathcal{H} \text{ is empty} \\ \max\{r \mid r \in \mathcal{H}\} & \text{otherwise} \end{cases}$$

Proof: First, we claim that $\mathcal{H}$ stores the rank of all those suffixes of B which have c as the first character, and which are lexicographically smaller than cX. The reason is as follows: Consider any suffix of B_i whose first character is c. Let r be its rank among $S(B)$. Note that r is within $R_B(c)$. If $B_i < cX$, then $B_{i+1} < X$. Denote the rank of B_{i+1} as r'. Then $r' < \text{Rank}(X, \mathcal{S}(B))$. On the other hand, by definition, $\Psi_B[r] = \text{SA}_B^{-1}[\text{SA}_B[r]{+}1] = r'$ (where SA_B and SA_B^{-1} denotes the suffix array of B and its inverse). Therefore, $\Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))$. Reversing the argument, we can show that for every r in $R_B(c)$ with $\Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))$, the suffix of B with rank r (i.e., $B_{\mathrm{SA}_B[r]}$) is lexicographically smaller than cX. Thus, the claim follows.

We are now ready to prove the lemma. If $\mathcal{H}$ is empty, any suffix of B starting with character c is lexicographically larger than or equal to cX. Then, $\mathrm{Rank}(cX, \mathcal{S}(B))$ is equal to the rank of the single character c among $\mathcal{S}(B)$, which is $\alpha_B(c)$. If $\mathcal{H}$ is not empty, $\mathrm{Rank}(cX, \mathcal{S}(B)) = \alpha_B(c) + |\mathcal{H}|$. By Corollary 2.2, $\Psi_B[r]$ is strictly increasing for r in $R_B(c)$, and $\mathcal{H}$ is equal to $\{\alpha_B(c), \alpha_B(c) + 1, \ldots, \alpha_B(c) + |\mathcal{H}| - 1\}$. Thus, $\max\{r \mid r \in \mathcal{H}\} = \alpha_B(c) + |\mathcal{H}| - 1$, and the lemma follows. $\square$

Based on the above lemma, we can compute the required rank in a backward manner as follows. The result is stored in an array $L[0, \ell - 1]$ such that $L[k] = \mathrm{Rank}(A_k, \mathcal{S}(B))$ for all k in $[0, \ell - 1]$.

> For $k = \ell - 1$ down to 0, compute $L[k]$ as follows: Let $c = A[k]$. The suffix A_k can be expressed as cA_{k+1}. Note that $\mathrm{Rank}(A_{k+1}, \mathcal{S}(B))$ has been computed and stored in $L[k+1]$.[*] To compute $L[k]$, we find the maximum r in $R_B(c)$ satisfying $\Psi_B[r] \le L[k+1]$. Since Ψ_{B} is strictly increasing in the range $R_B(c)$, we can use a binary search to find the maximum r; this requires $O(\log n)$ time. If r exists, we set $L[k]$ to be r; otherwise, we set $L[k]$ to be $\alpha_B(c)$.

Time and space requirement: The time required for computing $\#_B$, α_B, and L is $O(\ell \log n + n)$. Note that L occupies $\ell \log n$ bits. The array α_B requires $|\Sigma| \log n$ bits. We do not store $\#_B$ explicitly as any of its entries can be computed from α_B in $O(1)$ time. In most applications, we can assume that $|\Sigma| \le n/\log n$ and α_B requires at most n bits.[†]

[*]When $k = \ell - 1$, we assume that $L[\ell]$ has been set to the value of $\Psi_B[0]$. Note that $L[\ell]$ is the rank of A_ℓ (or equivalently B_0) among $\mathcal{S}(B)$, which is equal to $\mathrm{SA}_B^{-1}[0] = \Psi_B[0]$.

[†]For the unusual case where $|\Sigma| > n/\log n$, we can still reduce the space requirement to $2n + o(n)$ bits by representing α_B as a bit vector Z plus an indexing data structure that supports $O(1)$-time `rank` and `select` operation [42, 68]. Z is defined as $\#_B(a_1)$ 0's, a 1, $\#_B(a_2)$ 0's, a 1, $\#_B(a_3)$ 0's, a 1, and so on, where a_i denotes the i-th smallest character in Σ. Z occupies at most $2n$ bits, and it takes $o(n)$ bits for the extra data structure [68]. Any entry $\alpha_B[i]$ can then be computed in $O(1)$ time by a select-one operation (to obtain the position u of the $(i-1)$-th 1 in Z), followed by a rank-zero operation (to count the number of zeroes before position u).

3.2.3 Computing Ψ_A

This section shows how to make use of the results of Sections 3.2.1 and 3.2.2 to compute Ψ_A in $O(\ell \log n + n)$ time. For the space requirement, it takes $4\ell \log n + o(n)$ bits in addition to that for maintaining Ψ_A and Ψ_B. Recall that the following three arrays are available.

1. An array M such that $M[i]$ stores $\text{Rank}(A_i, \mathcal{LS}(A))$.
2. An array M^{-1}, which is the inverse of M, such that $M^{-1}[i]$ stores the position of the suffix among $\mathcal{LS}(A)$ whose rank is i.
3. An array L such that $L[i]$ stores $\text{Rank}(A_i, \mathcal{S}(B))$.

By Fact 3.3, we can compute the rank of each long suffix A_k (where k in $[0, \ell-1]$) among $\mathcal{S}(A)$ by summing $M[k]$ and $L[k]$. For the short suffixes of A, their rank among $\mathcal{S}(A)$ can be figured out by adjusting their rank among $\mathcal{S}(B)$ according to distribution of the long suffixes. Precisely, let $m = |A|$, and define $V[0, m-1]$ to be a bit vector such that $V[i] = 1$ if the suffix of A with rank i is a long suffix, and $V[i] = 0$ otherwise. We need V to support two types of efficient queries:

- $\texttt{rank}_0(V, i)$ and $\texttt{rank}_1(V, i)$ returns the number of 0's and 1's preceding $V[i]$, respectively.
- $\texttt{select}_0(V, j)$ returns the position of the j-th 0 in V.

Before showing how to construct V, we present a simple way to make use of V to calculate the rank of a short suffix among $\mathcal{S}(A)$ from its rank among $\mathcal{S}(B)$, and vice versa.

Lemma 3.6 *For any short suffix x of A, let $r = \text{Rank}(x, \mathcal{S}(A))$ and $r' = \text{Rank}(x, \mathcal{S}(B))$. Then, $r = \texttt{select}_0(V, r'+1)$ and $r' = \texttt{rank}_0(V, r)$.*

Proof: By definition, $V[r] = 0$. In the subarray $V[0, r-1]$, the number of 0's is equal to the number of short suffixes lexicographically smaller than x, which is equal to r'. Furthermore, $V[r]$ contains the $(r'+1)$-th 0. □

Next, we give the details of constructing V. Note that the number of bits in V depends on the size of A, which can be as big as n.

Lemma 3.7 *The bit vector V can be constructed from the array L in $O(n)$ time.*

Proof: We assume that $|A|$ bits are allocated for storing V explicitly. We compute V from L as follows: Recall that L stores the ranks of the long suffixes among $\mathcal{S}(B)$. These ranks can solely determine which entries in V store the 1's. We sort the ranks in L in ascending order, denoted as $r_0, r_1, \cdots, r_{\ell-1}$. Then we fill V with the following bits: r_0 0's, a 1, $(r_1 - r_0)$ 0's, a 1, $\cdots$, and finally $(r_{\ell-1} - r_{\ell-2})$ 0's, a 1, followed by all zeroes. □

There are several data structures in the literature that support the rank and select operations on a bit vector in constant time [42, 68]. In particular, we can make use of the recent result by Raman, *et al.* [68]; precisely, we can build a fully indexable dictionary for V (Lemma 2.3 in [68]) directly from L and we do not need to store the vector V explicitly. The size of this data structure and the construction space are both $\log \binom{n}{\ell} + O(\frac{n \log\log n}{\log n}) = o(n)$ bits, and the construction time is $O(n)$. With this data structure, the retrieval of $V[i]$ and the queries $\texttt{rank}_0(V, i)$, $\texttt{rank}_1(V, i)$, and $\texttt{select}_0(V, j)$ are performed in $O(1)$ time.

Finally, we are ready to show how to compute $\Psi_A[r]$ for all r in $[0, m-1]$. Recall that $\Psi_A[r]$ is defined as $\mathrm{SA}_A^{-1}[\mathrm{SA}_A[r]+1]$, or equivalently, if A_k is the suffix such that $\mathrm{Rank}(A_k, \mathcal{S}(A)) = r$, then $\Psi_A[r] = \mathrm{Rank}(A_{k+1}, \mathcal{S}(A))$. The following two lemmas show how to make use of V to figure out $\Psi_A[r]$ from $\Psi_B[r]$.

Lemma 3.8 *Consider any short suffix A_k whose rank among $\mathcal{S}(A)$ is r. Then*

- $\mathrm{Rank}(A_{k+1}, \mathcal{S}(B)) = \Psi_B[\texttt{rank}_0(V, r)]$; *and*
- $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = \texttt{select}_0(V, \Psi_B[\texttt{rank}_0(V, r)] + 1)$.

Proof: Since A_k is a short suffix whose rank among all suffixes of A is r, its rank among all suffixes of B is $r' = \texttt{rank}_0(V, r)$. The rank of A_{k+1} among all suffixes of B is $p = \Psi_B[r']$. By Lemma 3.6, $\Psi_A[r]$, the rank of A_{k+1} among all suffixes of A, is $\texttt{select}_0(V, p+1)$. □

Lemma 3.9 *Consider any long suffix A_k whose rank among $\mathcal{S}(A)$ is r. Then*

- $k = M^{-1}[\texttt{rank}_1(V, r)]$; *and*
- *if* $k < \ell - 1$ *then* $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = M[k+1] + L[k+1]$; *otherwise,* $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = \texttt{select}_0(V, \Psi_B[0] + 1)$.

Proof: Since x is a long suffix, its rank among all long suffixes is $r' = \texttt{rank}_1(V, r)$. By the definition of M, $k = M^{-1}[r']$. Note that k is in the range $[0, \ell - 1]$. If $k < \ell - 1$, then $\Psi_A[r]$, which is the rank of A_{k+1} among all suffixes of A, is equal to $M[k+1] + L[k+1]$ (by Fact 3.3).

For the special case where k is equal to $\ell - 1$, $\Psi_A[r]$ is equal to the rank of $A_\ell = B_0$ among all suffixes of A. We can find this rank as follows: Compute the rank p of B_0 among all suffixes of B, which is equal to $\mathrm{SA}_B^{-1}[0] = \Psi_B[0]$. Then, by Lemma 3.6, the rank of B_0 among all suffixes of A is $\mathtt{select}_0(V, p+1)$. □

Based on the above two lemmas, we can compute $\Psi_A[r]$ sequentially for $r = 0, 1, \ldots, m-1$. For the base case when $r = 0$, we note that $\Psi_A[0]$, which is defined as $SA^{-1}[0]$ or the rank of A_0 among all suffixes of A, is exactly $M[0] + L[0]$ (by Fact 3.3). The details are depicted in Figure 3.1.

```
$\Psi_A[0] \leftarrow M[0] + L[0]$;
for $r \leftarrow 1$ to $m - 1$
   if $V[r] = 0$ {    % The suffix with rank $r$ is a short suffix.
         $r' \leftarrow \mathtt{rank}_0(V, r)$;
         $p \leftarrow \Psi_B[r']$;
         $\Psi_A[r] \leftarrow \mathtt{select}_0(V, p+1)$;
   }
   else {      % The suffix with rank $r$ is a long suffix.
         $r' \leftarrow \mathtt{rank}_1(V, r)$;
         $k \leftarrow M^{-1}[r']$;
         if $k < \ell - 1$
               $\Psi_A[r] \leftarrow M[k+1] + L[k+1]$;
         else {
               $p \leftarrow \Psi_B[0]$;
               $\Psi_A[r] \leftarrow \mathtt{select}_0(V, p+1)$;
         }
   }
```

Figure 3.1: Computing $\Psi_A[r]$ sequentially.

Calculating each $\Psi_A[r]$ involves a constant number of $O(1)$ time operations, and the whole procedure takes $O(m) = O(n)$ time. Combining the results of Sections 3.2.1 and 3.2.2, we have the following lemma.

Lemma 3.10 *Suppose that Ψ_B is given. Computing all the auxiliary data structures (M, M^{-1}, L, and V) and then enumerating the values of Ψ_A can be done in $O(\ell \log n + n)$ time. Excluding the space for representing Ψ_A and Ψ_B, the working space required is $4\ell \log n + n + o(n)$ bits.*

As mentioned in Lemma 2.3 of Section 2.3, Ψ_A admits a compact representation using $n(H_0 + 2 + o(1))$ bits. Together with Lemma 3.10, we conclude this section with the following result.

Theorem 3.11 *Given a string T of length n, the CSA of T can be computed in $O(n \log n)$ time using $O(n(H_0 + 1))$ bits.*

Proof: The construction is divided into $\lceil n/\ell \rceil = \Theta O(\log n)$ phases since $\ell = \Theta(n/\log n)$. Each phase takes $O(\ell \log n + n) = O(n)$ time, and the overall time is $O(n \log n)$.

For the space requirement, it takes $4\ell \log n + n + o(n)$ bits in addition to that for two Ψ arrays. The total space is thus $n(2H_0+5+o(1))+4\ell \log n$ bits. Choosing $\ell = \frac{\epsilon n}{4 \log n}$ for any $0 < \epsilon < 1$, the space requirement becomes $n(2H_0+5+\epsilon+o(1)) = O(n(H_0 + 1))$ bits. Furthermore, by a careful memory management, we can actually maintain the two Ψ arrays together using $n(H_0 + 2 + o(1))$ bits only. That is, we overwrite the old Ψ array when we store the new Ψ. Then the total space requirement is reduced to $n(H_0 + 3 + \epsilon + o(1))$ bits. $\square$

Chapter 4

Constructing CSA and FM-index in $O(n \log \log |\Sigma|)$ time

In Chapter 2, we have described a representation of the Ψ function of the CSA that takes $O(n(H_0+1))$ bits of storage, and show that this representation can be stored easily when the Ψ values are output in sequential order. In this chapter, we introduce another representation of Ψ, which takes slightly more space of $O(n \log |\Sigma|)$ bits. In compensation, this representation can be produced easily even if the Ψ values are output in an arbitrary order.

Based on this representation, we show that CSA and FM-index can be constructed in $O(n \log n)$ time and $O(n \log |\Sigma|)$-bit working space, following the framework of Farach's linear-time suffix tree construction algorithm [22]. Intuitively, Farach's idea is to first construct the suffix tree for even-position suffixes using recursion, and then from which we obtain the suffix tree for odd-position suffixes, and finally we merge the two suffix trees together to produce the desired suffix tree. Here, we make use of the Ψ function of CSA and the Burrows-Wheeler text alternately as an implicit representation of these suffix trees.

To achieve further speed up, we observe that the bottleneck of our algorithm is the frequent execution of backward search steps. Using the known implementation by Sadakane and Shibuya [75], each step requires $O(\log n)$ time. In this chapter, we introduce an auxiliary data structure of $O(n \log |\Sigma|)$ bits which can improve each step to $O(\log \log |\Sigma|)$ time instead. As a result, the total time for our construction algorithm is improved to $O(n \log \log |\Sigma|)$ time.

The organization of this chapter is as follows. In Section 4.1, we introduce the new representation of Ψ and some preliminary lemmas. In Section 4.2, we introduce the auxiliary data structure that leads to the improvement in backward search algorithm. Section 4.3 describes the framework for the construction algorithm, while Section 4.4 and 4.5 details the main steps of the algorithm. Finally, we discuss a further improvement when the alphabet size is small in Section 4.6.

4.1 Preliminaries

This section is divided into two parts. The first part describes the new representation of the Ψ function. In the second part, we discuss the duality between the Ψ function and the Burrows-Wheeler text W.

Firstly, we review some of the basic notations and assumptions. For a text T of length n over an alphabet Σ, it is denoted by $T[0, n-1]$. Each character of Σ is uniquely encoded by an integer in $[0, |\Sigma|-1]$ which occupies $\log|\Sigma|$ bits. In addition, a character c is alphabetically larger than a character c' if and only if the encoding of c is larger than the encoding of c'.

Finally, we assume that $T[n-1]$ is a special character that does not appear elsewhere in the text. Note that we have removed a previous assumption that this special character is the smallest character in the alphabet. According to this change, we generalize the definition of the Ψ function as follows:

- $\Psi[i] = \mathrm{SA}^{-1}[\mathrm{SA}[i]+1]$ if $\mathrm{SA}[i] \neq n-1$;
- $\Psi[i] = \mathrm{SA}^{-1}[0]$ otherwise.

4.1.1 Representation of Ψ

Recall from Corollary 2.2 that the Ψ function is piece-wise increasing. In addition, each Ψ value is less than n. Therefore, we can make use of a function $\rho(c, x) = enc(c) \cdot n + x$ and obtain a total increasing function $\Psi'[i] = \rho(T[i], \Psi[i])$, where $enc(c)$ denotes the encoding of the character c. Note that value of Ψ' is less than $n|\Sigma|$.

Based on the total increasing property, Ψ' can be stored as follows [36]. We divide each $\Psi'[i]$, which takes $\log n + \log|\Sigma|$ bits, into two parts q_i and r_i, where q_i is the first (or most significant) $\log n$ bits, and r_i is the remaining $\log|\Sigma|$ bits. We encode the values $q_0, q_1 - q_0, \ldots, q_{n-1} - q_{n-2}$ in a bit-vector B_1 using unary codes. (Recall that the unary code for an integer $x \geq 0$ is encoded as x `0`'s followed by a `1`.) Note that the encoding has exactly n `1`'s where the $(i+1)$-th

1, which corresponds to $\Psi[i]$, is at position $i+q_i$. Also, the total number of 0's is q_{n-1}, which is at most n. Thus, B_1 uses $2n$ bits. The r_i's are stored explicitly in an array $B_2[0, n-1]$ where each entry occupies $\log|\Sigma|$ bits. Thus, B_2 occupies $n\log|\Sigma|$ bits. Moreover, an auxiliary data structure of $O(n/\log\log n)$ bits is constructed in $O(n)$ time to enable constant time *rank* and *select* queries, and thus supporting the retrieval of any q_i in constant time [42, 61]. Then, the total size is $n(\log|\Sigma|+2)+o(n)$ bits. Since q_i and r_i can be retrieved in constant time, so can $\Psi'[i] = |\Sigma|q_i + r_i$. This gives the following lemma.

Lemma 4.1 *The Ψ' function can be encoded in $O(n\log|\Sigma|)$ bits, so that each $\Psi'[i]$ can be retrieved in constant time.*

Corollary 4.2 *The Ψ function can be encoded as Ψ' in $O(n\log|\Sigma|)$ bits, so that each $\Psi[i]$ can be retrieved in constant time.*

Proof: The retrieval time follows since $\Psi[i] = \Psi'[i] \bmod n$. □

4.1.2 Duality between Ψ and W

Recall that the Burrows-Wheeler text W is a transformation on T such that $W[i] = T[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] > 0$, and $W[i] = \$$ otherwise. It is known that Ψ and W are one-to-one corresponding. In the section, we show that the transformation between them can be done in linear time and in $O(n\log|\Sigma|)$-bit space.

We first give a property relating W and Ψ.

Definition 4.3 *Given an array of characters $x[0, n-1]$, we define the* stable sorting order *of $x[i]$ in x to be the number of characters in x which is alphabetically smaller than $x[i]$, plus the number of characters $x[j]$ with $j < i$ which is equal to $x[i]$. This is in fact the position of $x[i]$ after stable sorting.*

Lemma 4.4 ([13]) *Let k be the stable sorting order of $W[i]$ in W. Then, $\Psi[k] = i$.*

Proof: Let Y_i denote the suffix $T_{\mathrm{SA}[i]-1}$ when $\mathrm{SA}[i] > 0$, and the suffix $T[n-1]$ otherwise. Note that when $i < j$, if Y_i and Y_j are starting with the same character, the lexicographical order of Y_i will be smaller than Y_j. The reason is that, the remaining part of Y_i is $T_{\mathrm{SA}[i]}$, and the remaining part of Y_j is $T_{\mathrm{SA}[j]}$, and since $i < j$, we have $T_{\mathrm{SA}[i]} < T_{\mathrm{SA}[j]}$ and thus $Y_i < Y_j$. Also, observe that the first character of Y_i is equal to $W[i]$. Then, it follows that the stable sorting order of $W[i]$ in W is equal to the rank of Y_i among the set of all Y_i's, which is the set of all suffixes $\mathcal{S}(T)$.

Thus, we have $k = \mathrm{SA}^{-1}[\mathrm{SA}[i]-1]$ when $\mathrm{SA}[i] > 0$, and $k = \mathrm{SA}^{-1}[n-1]$ otherwise. In the former case, $\mathrm{SA}[k] = \mathrm{SA}[i]-1 < n-1$, so $\Psi[k] = \mathrm{SA}^{-1}[\mathrm{SA}[k]+1] = \mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$. For the latter case, we have $i = \mathrm{SA}^{-1}[0]$ and $\mathrm{SA}[k] = n-1$. Thus we have $\Psi[k] = \mathrm{SA}^{-1}[0] = i$. In summary, $\Psi[k] = i$ for all cases, and the lemma follows. □

The next two lemmas show the linear time conversion between W and Ψ.

Lemma 4.5 *Given W, we can store Ψ in $O(n)$ time and in $O(n \log|\Sigma|)$ bits. The working space is $O(n \log|\Sigma|)$ bits.*

Proof: We construct Ψ' in Section 4.1.1 from W as an encoding of Ψ (Corollary 4.2). To construct Ψ', we create a bit-vector $B_1[0, 2n-1]$ and initialize all bits to `0`. We also create an array $B_2[0, n-1]$ where each entry occupies $\log|\Sigma|$ bits.

Now, we show how to compute the stable sorting order of $W[i]$ in W, for $i = 0, 1, 2, \ldots$. To do so, we use three arrays for our help. The first array is L_1 such that $L_1[c]$ stores the number of occurrences of the character c in W. This array can be initialized by scanning W once. The second array is L_2 such that $L_2[c]$ stores the number of occurrences of a character that is smaller than c in W. This array can be initialized by scanning L_1 once. Finally, the third array is L_3 such that $L_3[c]$ stores the number of occurrences of c seen so far. Initially, all entries of L_3 are initialized to 0.

Now, we proceed to read $W[0]$, $W[1]$, and so on. Note that during the process, when we read a character c, we maintain the correctness of L_3 by incrementing $L_3[c]$ just before the next character is read. Thus, at the beginning of step i, the counter $L_3[W[i]]$ will be storing the number of occurrences of $W[i]$ in $W[0, i-1]$, and the stable sorting order of $W[i]$ can be computed at by $L_2[W[i]] + L_3[W[i]]$.

Let k be the stable sorting order of $W[i]$ that is computed at step i in the above algorithm. By Lemma 4.4, $\Psi[k] = i$. Thus, we have $\Psi'[k]$ equals $x = \rho(T[\mathrm{SA}[k]], \Psi[k]) = \rho(W[i], i)$. By our scheme, x is divided into two parts q and r, where $q = x \text{ div } |\Sigma|$ is the first $\log n$ bits, and $r = x \bmod |\Sigma|$ is the remaining bits. For q, a `1` is stored at $B_1[k+q]$. For r, it is stored at $B_2[k]$.

As the stable sorting order of each $W[i]$ is different, all possible $\Psi[k]$ will be computed and stored eventually. A summary of the overall algorithm is shown

```
1. Compute $L_1[c]$ to store the number of occurrences of each
   $c$ in $\Sigma$.
2. Compute $L_2[c]$ to store the number of occurrences of a char-
   acter that is smaller than $c$. That is, $L_2[c] = \sum_{d<c} L_1[c]$ for
   $c$ in $\Sigma$.
3. Let $L_3[c]$ be an array that stores the number of occurrences
   of the character $c$ seen so far.
4. Initialize all entries of $L_3$ to be 0.
5. For $i = 1, 2, \ldots, n$
      Let $c = W[i]$, $k = L_2[c] + L_3[c]$
      Compute $x = \rho(c, i)$
      Let $q = x$ div $|\Sigma|$, $r = x$ mod $|\Sigma|$
      Set $B_1[k+q] = 1$ and $B_2[k] = r$
      Increment $L_3[c]$ by one
6. Compute the $O(n/\log\log n)$-bit auxiliary data structure
   for $B_1$.
```

Figure 4.1: Computing Ψ from W.

in Figure 4.1. It is easy to see that the overall time is $O(n + |\Sigma|)$. For the space complexity, note that L_1, L_2 and L_3 each occupies $|\Sigma| \log n$ bits, which is at most $n \log|\Sigma|$ bits because $|\Sigma| \le n$. Thus, we use $O(n \log|\Sigma|)$-bit working space. □

Lemma 4.6 *Given Ψ and T, we can store W in $O(n)$ time and in $O(n \log|\Sigma|)$ bits. The working space is $O(n \log|\Sigma|)$ bits.*

Proof: Let $t = \mathrm{SA}^{-1}[0]$. Then, we have $\Psi[t] = \mathrm{SA}^{-1}[1]$. In general, $\Psi^k[t] = \mathrm{SA}^{-1}[k]$.

Hence, we have $W[\Psi^k[t]] = T[k-1]$. To construct W, we can first compute t. Recall that $T[n-1]$ is a unique character in T. By scanning T, we can find the value $x = \mathrm{SA}^{-1}[n-1]$, which is equal to the number of occurrences of a character in T that is smaller than $T[n-1]$. Then, by definition, $\Psi[x] = \mathrm{SA}^{-1}[0]$, which is equal to t. Thus, t can be found in $O(n)$ time. Afterwards, we iteratively compute $\Psi^i[t]$ and set $W[\Psi^i[t]] = T[i-1]$, for $i = 1$ to n. As $\Psi^i[t]$ corresponds to the rank of a different suffix of T for different i, all the characters of W will eventually be computed and stored by the above algorithm. The total time of the algorithm is $O(n)$, and the space for W,T, and Ψ are all $O(n \log |\Sigma|)$ bits. The lemma thus follows. $\square$

4.2 Improving the Backward Search Algorithm

Let S be a text of length m over an alphabet Δ. In this section, we present an $O(m+|\Delta|)$-bit auxiliary data structure for the the Ψ function of S that improves each backward search step in the backward search algorithm from $O(\log m)$ time to $O(\log \log |\Delta|)$ time.

Formally, a *backward search step* is defined as follows.

Definition 4.7 *For any pattern P, suppose that the rank of P among all suffixes of S is known. A* backward search step *then computes the rank of cP among the suffixes of S for any character $c \in \Delta$.*

Let Ψ' denote the total increasing function such that $\Psi'[i] = \rho(S[\mathrm{SA}[i]], \Psi[i])$ and $\rho(c, x) = enc(c) \cdot m + x$. Then, we have the following lemma.

Lemma 4.8 *Let r be the rank of P among all suffixes of S. Then, the rank of cP among all suffixes of S is equal to $j \in [0, m]$ such that $\Psi'[j-1] < \rho(c, r) \leq \Psi'[j]$. (As a sentinel, we let $\Psi'[-1] = -1$ and $\Psi'[m] = m|\Delta|$.)*

Proof: It is easy to check that for all $i = 0, 1, \ldots, j-1$, the rank-i suffix of S must either be starting with a character smaller than c, or starting with c but the remaining part is lexicographically smaller than P. Thus, for all $i = 0, 1, \ldots, j-1$, the rank-i suffix of S is lexicographically smaller than cP. On the other hand, for all $i \geq j$, the rank-i suffix of S is lexicographically not smaller than cP. Thus, the rank of cP is j. $\square$

Let Q be a set of numbers. For any integer x, the *rank* of x in Q is the number of elements in Q which is smaller than x. Essentially, a backward search step that computes the rank of cP in the above lemma, is equivalent to finding the rank of $\rho(c, r)$ in the set of all Ψ' values. The following theorem shows a data structure that is useful for finding such rank. Note that in contrast to the previous data structures for the rank query [42, 68], our data structure requires either less space for storage, or less time in the construction; the drawback is a blow-up in query time. The proof of this theorem will be deferred to Section 4.2.1.

Theorem 4.9 *Let Q be a set of n numbers, each of length $\Theta(\log n)$ bits. Then, a data structure of size $O(n \log n)$ bits supporting $O(\log\log n)$-time* `rank` *query in Q can be constructed in $O(n)$ time and $O(n \log n)$-bit working space.*

Based on Theorem 4.9, we can use some extra space to achieve a more generalized result, as shown in the following corollary.

Corollary 4.10 *Let Q' be a set of n values, each of length $\log \ell + \Theta(\log n)$ bits for any ℓ. Then, a data structure of size $O(n \log n + \ell)$ bits supporting $O(\log\log n)$-time rank query in Q' can be constructed in $O(n + \ell)$ time and $O(n \log n + \ell)$-bit working space.*

Proof: The idea is to apply Theorem 4.9 by transforming the set Q' into another set such that each value takes only $\Theta(\log n)$ bits. Firstly, we scan Q' and create a bit-vector $B[0, \ell - 1]$ such that $B[i] = \mathtt{1}$ if there is some number in Q' whose first $\log \ell$ bits represents a value i, and $B[i] = \mathtt{0}$ otherwise. Afterwards, we construct an auxiliary data structure for B of size $o(\ell)$ bits to support constant time `rank` and `select` queries.

Now, we transform each number in Q' as follows: if the first $\log \ell$ bit of the number represents the value i, these bits are replaced by the binary bit-sequence for the rank of i in B. Note that the rank of i is less than n, as there are only n numbers. Thus, after the transformation, each value takes $\Theta(\log n)$ bits, and in addition, the transformation preserves the ordering among the elements in Q'.

Let the set of the transformed values be Q. We create the data structure of Theorem 4.9 on Q. To perform a rank query for x in Q' (we assume that x has the same length as any number in Q'), we first obtain the first $\log \ell$ bits of x. Suppose that these bits represent the value i_x. Then there are two cases:

(Case 1) If $B[i_x] = \mathtt{1}$, we replace the first $\log \ell$ bits of x by the $\log n$-bits that represents the rank of i_x in B, and obtain a new value y. Then, it is easy to see that the rank of x in Q' is equal to the rank of y in Q.

(Case 2) Otherwise, we replace the first $\log \ell$ bits of x by the $\log n$-bits that represents the rank of i_x in B, while setting the remaining $\Theta(\log n)$ bits to zeroes, and obtain a new value z. Then, it is easy to see that the rank of x in Q' is equal to the rank of z in Q.

Finally, for the time and space complexity, B and its auxiliary data structures can be created in $O(\ell)$ time and stored in $\ell + o(\ell)$ bits, while the data structure for rank query in Q can be created in $O(n)$ time and stored in $O(n \log n)$ bits (By Theorem 4.9). The lemma thus follows. $\square$

Now, we are ready to describe the main lemma of this section.

Lemma 4.11 *Let S be a text of length m over an alphabet Δ. Suppose that the Ψ function of S is given, which is stored as Ψ' using the scheme in Section 4.1.1. Then, an auxiliary data structure for the Ψ function of S can be constructed in $O(m+|\Delta|)$ time, which supports each backward search step in $O(\log\log|\Delta|)$ time. The space requirement is $O(m+|\Delta|)$ bits.*

Proof: Let V denote the set of all Ψ' values. To prove the lemma, it suffices to show a data structure of $O(m+|\Delta|)$ bits that supports `rank` query for any x in V in $O(\log\log|\Delta|)$ time.

Firstly, recall that in our encoding of Ψ', each value in V is stored in two parts, where the first $\log m$ bits are encoded by unary codes in a bit-vector B_1, and the remaining $\log|\Delta|$ bits are encoded in an array B_2 as it is. In addition, there is an auxiliary data structure supporting constant time `rank` and `select` queries.

Let G_i be the set of Ψ' values whose first $\log m$ bits represents the value i. Among the sets of G_i's, we are concerned with those sets whose size is greater than $\log|\Delta|$. Let $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ be such sets, and $i_1 < i_2 < \cdots < i_k$.

Note that the groups $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ each has size between $\log|\Delta|$ and $|\Delta|$. Now, we combine the groups, from left to right, into super-groups of size $\Theta(|\Delta|)$. More precisely, we start from G_{i_1}, merge it with G_{i_2}, G_{i_3} and so on, until the size exceeds $|\Delta|$. Then, we merge the next unmerged group with its succeeding group and so on, until the size exceeds $|\Delta|$. The process is repeated until all groups are within a super-group. (To ensure that each super-groups would have size $\Theta(|\Delta|)$, we add a dummy group $G_m = \{m|\Delta|, m|\Delta|+1, \ldots, (m+1)|\Delta|-1\}$ as a sentinel.)

For each super-group $\mathcal{G}$, let $v_0, v_1, \ldots, v_p$ be its $\Theta(|\Delta|)$ elements. Now, we pick every $\log|\Delta|$ elements (i.e., $v_0, v_{\log|\Delta|}, v_{2\log|\Delta|}, \ldots$), subtract each of them by v_0, and make them the representatives of this super-group. Then, we construct the data structure for `rank` query of Corollary 4.10 over these representatives.

With the above data structure, `rank` query for any x in $\mathcal{G}$ can be supported as follows. We first check if $x \le v_0$. If so, the `rank` of x is 0. Otherwise, we find the `rank` of $x - v_0$ among the representatives, which takes $O(\log\log|\Delta|)$ time.

Suppose the rank is r. Then, the rank of x in $\mathcal{G}$ must lie between $r\log|\Delta|$ and $(r+1)\log|\Delta|-1$, and this can be found by a binary search in the elements $\{v_{r\log|\Delta|},\ldots,v_{(r+1)\log|\Delta|-1}\}$ which takes $O(\log\log|\Delta|)$ time. In summary, the time required is $O(\log\log|\Delta|)$.

Now, let us complete the whole picture to show how to perform the `rank` query for x in V. Firstly, we extract the first $\log m$ bits of x by dividing it with $|\Delta|$. Let $i' = x \text{ div } |\Delta|$ be its value. Next, we determine the size of $G_{i'}$, which can be done in constant-time using `rank` and `select` queries on B_1. If the size is 0 (i.e., $G_{i'}$ is empty), the rank of x in V can be computed immediately (precisely, the required rank is equal to the number of `1`'s in $B_1[0,i'-1]$, which can be computed in constant time using B_1 and its auxiliary data structure). If the size is smaller than $\log|\Delta|$, the rank of x can be found by performing a binary search with the elements in $G_{i'}$, which takes $O(\log\log|\Delta|)$ time. Finally, if the size is greater than $\log|\Delta|$, we locate the super-group $\mathcal{G}$ that contains the elements of $G_{i'}$, and retrieve the rank r of its smallest element v_0 in V. Then, the required rank is r plus the rank of x in $\mathcal{G}$. We now claim that locating the super-group and retrieval of r can be done in constant time (to be proved shortly), so that the total time is $O(\log\log|\Delta|)$.

We prove the above claim as follows. To support finding the smallest element in each super-group, and retrieval of its rank in V, we use a bit-vector B_1' of $O(m)$ bits, obtained from B_1 by keeping only those `1`'s whose corresponding Ψ' value is a smallest element in some super-group. Also, we augment B_1' with constant-time `rank` and `select` date structures. Then, the smallest value of the $(i+1)$-th super-group, and its rank in V, can be found by consulting B_1 and B_1' in constant time. In addition, for any G_i (with size greater than $\log|\Delta|$), the rank of its super-group among the other super-groups can be found by consulting B_1' in constant time.

On the other hand, to support locating the `rank` data structure of the super-group, we first analyze the space requirement of these data structures. For a particular super-group $\mathcal{G} = \{v_0, v_1, \ldots, v_p\}$, the data structure is built for $p/\log\Delta = \Theta(\Delta/\log\Delta)$ elements, each of which has value in $[0, v_p - v_0]$, so that the space is $O(v_p - v_0 + \frac{p}{\log|\Delta|}\cdot\log|\Delta|)$ bits (by Corollary 4.10), which is $O(v_p - v_0)$ bits since $p \le v_p - v_0$. Thus, the total space requirement is $O(m + |\Delta|)$ bits,[*] and we assume that the data structures of the super-groups are stored consecutively according to the rank of its smallest element. Then, we create a bit-vector B_3

[*]The additional $O(|\Delta|)$ bits are due to the dummy group G_m.

whose length is identical to the above data structures, which is used to mark the starting position of each data structure. Also, we augment the bit-vector with an $o(m+|\Delta|)$-bit auxiliary data structure to support constant time `rank` and `select` query. Thereafter, when we want to locate a super-group for G_i, we find its rank r among the other super-groups using B'_1, and then this rank-r super-group can be accessed in constant time using B_3.

In summary, our data structure takes a total space of $O(m + |\Delta|)$ bits and supports each backward search step in $O(\log\log|\Delta|)$ time. For the construction, it takes at most $O(m + |\Delta|)$ time. The lemma thus follows. $\square$

4.2.1 Data Structure for Efficient Rank Query

This section is devoted to proving Theorem 4.9. We begin with two supporting lemmas. The first one is on perfect hash function, which is obtained by rephrasing the result of Section 4 of [39] as follows.

Lemma 4.12 *Given x b-bit numbers, where $b = \Theta(\log x)$, a data structure of size $O(xb)$ bits supporting $O(1)$-time existential query can be constructed in $O(x \log x)$ time and $O(xb)$-bit working space.*

The second one is derived from adapting a result in [58, 81] based on Lemma 4.12.

Lemma 4.13 *Given z w-bit numbers, where $w = \Theta(\log z)$, a data structure of size $O(zw^2)$ bits supporting $O(\log w)$-time* `rank` *queries can be constructed in $O(zw \log(zw))$ time and $O(zw^2)$-bit working space.*

Proof: It is shown that `rank` queries can be solved in $O(\log w)$ time, if existential query for all prefixes of the z numbers can be answered in $O(1)$ time [58, 81].[†] The idea is that, given a w-bit number k, its longest common prefix with the z numbers can be found by binary search (on the length) using $O(\log w)$ existential queries, and such a prefix uniquely determines the `rank` of k.

[†]In the original papers, the results are for another query called `predecessor`, which finds the largest element in the z numbers that is smaller than the input w-bit number k. However, such a result can be modified easily for the `rank` query as follows. For each number i in the z numbers, it is replaced by the number $iz +$ the rank of i (so that the number now has $w + \log z$ bits), and we construct the `predecessor` data structure for these modified numbers. For the intended `rank` query, we first try to find the predecessor for kz in the modified numbers, and if no predecessor is found, the rank of k in the z numbers is 0. Otherwise, let this predecessor be p. It is easy to see that the required result is equal to $(p \bmod z) + 1$.

Notice that only $O(zw)$ strings can be a prefix of the z numbers and each can be represented in $\Theta(w)$ bits. Applying Lemma 4.12 on this set of strings (with $x = O(zw)$ and $b = \Theta(w)$), we have the required data structure. The lemma thus follows. □

Now, we prove Theorem 4.9 with the data structure constructed as follows.

1. Let $k_0 < k_1 < \cdots < k_{n-1}$ be n numbers stored in ascending order by an array.
2. Partition the n numbers into n/w^2 lists, each containing w^2 numbers. Precisely, the lists are in the form $\{k_i, k_{i+1}, \ldots, k_{i+w^2-1}\}$, where $i \equiv 0 \pmod{w^2}$.
3. Let the smallest element in each list be its representative. Construct a data structure for `rank` query for these representatives based on Lemma 4.13.

The above data structure occupies $O(nw)$ bits, and can be constructed in $O(n)$ time and $O(nw)$-bit working space. With such a data structure, the `rank` of x among the n numbers can be found in $O(\log w)$ time as follows.

1. Find the `rank` of x among the n/w^2 representatives of the lists. Let this be r.
2. Then, the `rank` of x among the n numbers must now lie in $[rw^2, (r+1)w^2-1]$. Binary search on the w^2 elements $\{k_{rw^2}, k_{rw^2+1}, \ldots, k_{(r+1)w^2-1}\}$ to find the `rank` of x.

Both steps thus take $O(\log w)$ time. This completes the proof of Theorem 4.9.

4.3 The Framework of Constructing CSA and FM-index

Recall that $T[0, n-1]$ is a text of length n over an alphabet Σ, and we assume that $T[n-1]$ is a special character that does not appear elsewhere in T. This section describes how to construct the Ψ function and the Burrows-Wheeler text W of T in $O(n \log\log|\Sigma|)$ time. Our idea is based on Farach's framework for linear-time construction of the suffix tree [22], which first constructs the suffix tree for even-position suffixes by recursion, based on which induces the suffix tree for odd-position suffixes, and then merge the two suffix trees to obtain the required one.

For our case, we first assume that the length of T is a multiple of $2^{\lceil \log\log_{|\Sigma|} n \rceil + 1}$. (Otherwise, we add enough \$ and a \$$'$ at the end of T, where \$$'$ is a character alphabetically smaller than the other characters in T, and proceed with the algorithm. The Ψ of this modified string can be converted into the Ψ of T in $O(n)$ time.) Let h be $\lceil \log\log_{|\Sigma|} n \rceil$. For $0 \le k \le h$, we define T^k to be the string over the alphabet Σ^{2^k}, which is formed by concatenating every 2^k characters in T to make one character. That is, $T^k[i] = T[i \cdot 2^k 1, (i+1) \cdot 2^k - 1]$, for $1 \le i \le n/2^k$. By definition, $T^0 = T$.

In addition, we introduce two notations associating with a string. For any string $S[0, m-1]$ with even number of characters, denote S_e and S_o to be the strings of length $m/2$ formed by merging every 2 characters in $S[0, m-1]$ and $S[1, m-1]S[0]$, respectively. More precisely, $S_e[i] = S[2i]S[2i+1]$ and $S_o[i] = S[2i+1]S[2i+2]$, where we set $S[m] = S[0]$. Intuitively, the suffixes of S_e and S_o corresponds to the even-position and odd-position suffixes of S, respectively. We have the following observation.

Observation 4.14 $T^i_e = T^{i+1}$.

Also, note that the last characters of T^i_o and T^i_e are unique among the corresponding string. This makes the results in Sections 4.1 and 4.2 applicable for both texts.

Our basic framework is to use a bottom-up approach to construct Ψ of T^i, or Ψ_{T^i}, for $i = \lceil \log\log_{|\Sigma|} n \rceil$ down to 0, thereby obtaining Ψ of T in the end. Precisely,

- For Step $i = \lceil \log\log_{|\Sigma|} n \rceil$, Ψ_{T^i} is constructed by first building the suffix tree for T^i using Farach's algorithm [22], and then converting it back to the Ψ function.
- For the remaining steps, we construct the Ψ_{T^i} based on the $\Psi_{T^{i+1}}$, the latter of which is in fact $\Psi_{T^i_e}$ by Observation 4.14. We first obtain $\Psi_{T^i_o}$ based on T^i and $\Psi_{T^i_e}$. Afterwards, we merge $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$ to give Ψ_{T^i}. The complete algorithm is shown in Figure 4.2.

Sections 4.4 and 4.5 describe in details how to obtain $\Psi_{T^i_o}$ from $\Psi_{T^i_e}$ and T^i, and how to merge $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$ to obtain Ψ_{T^i}, respectively. This gives the main theorem of this section.

```
1. For $i = \lceil \log\log_{|\Sigma|} n \rceil$
    (a) Construct suffix tree for $T^i$.
    (b) Construct $\Psi_{T^i}$ from the suffix tree.
2. For $i = \lceil \log\log_{|\Sigma|} n \rceil - 1$ to 0
    (a) Construct $\Psi_{T^i_o}$ based on $\Psi_{T^i_e}$. (Note: $\Psi_{T^i_e} = \Psi_{T^{i+1}}$.)
    (b) Construct $\Psi_{T^i}$ based on the $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$.
```

Figure 4.2: The construction algorithm of Ψ function of T.

Theorem 4.15 *The Ψ function and the Burrows-Wheeler text W of T can be constructed in $O(n \log\log|\Sigma|)$ time and $O(n \log|\Sigma|)$-bit working space.*

Proof: We refer to the algorithm in Figure 4.2, which has two phases. For Phase 1, we have $i = \lceil \log\log_{|\Sigma|} n \rceil$. We first construct the suffix tree for T^i whose size is $n/2^{\lceil \log\log_{|\Sigma|} n \rceil} \leq n \log|\Sigma| / \log n$. This requires $O(n \log|\Sigma| / \log n)$ time and $O(n \log|\Sigma|)$-bit space by using Farach's suffix tree construction algorithm [22]. Then, Ψ_{T^i} can be constructed in $O(n \log|\Sigma| / \log n)$ time and $O(n \log|\Sigma|)$-bit working space. Thus, Phase 1 in total takes $O(n)$ time and $O(n \log|\Sigma|)$-bit space.

For every Step i in Phase 2, we construct Ψ_{T^i}. Let Δ_i be the alphabet of T^i. Then, Part (a) takes $O(|T^i| + |\Delta_i|)$ time (Lemma 4.21), and Part (b) takes $O(|T^i| \log\log|\Delta_i| + |\Delta_i|)$ time (Lemma 4.23). For the space, both require $O(|T^i| \log|\Delta_i| + |\Delta_i|)$ bits. Note that $|T^i| = n/2^i$ and $|\Delta_i| \leq |\Sigma|^{2^i} \leq n$, so the space used by Step i is $O(|T^i| \log|\Delta_i| + |\Delta_i|) = O(n \log|\Sigma|)$ bits, and the time is $O(|T^i| \log\log|\Delta_i| + |\Delta_i|) = O((n/2^i) \cdot (i + \log\log|\Sigma|) + |\Sigma|^{2^i})$. In total, the space for Phase 2 is $O(n \log|\Sigma|)$ bits and the time is:

$$\begin{aligned} & \sum_{i=1}^{\lceil \log\log_{|\Sigma|} n \rceil - 1} O\left(\frac{n}{2^i}(i + \log\log|\Sigma|) + |\Sigma|^{2^i}\right) \\ = \; & O(n \log\log|\Sigma|). \end{aligned}$$

The whole algorithm for constructing Ψ of T therefore takes $O(n \log\log|\Sigma|)$ time and $O(n \log|\Sigma|)$-bit space. Finally, the Burrows-Wheeler text W can be constructed from Ψ using Lemma 4.6 in $O(n)$ time and $O(n \log|\Sigma|)$-bit space. This completes the proof. □

Once the Burrows-Wheeler transformation is completed, FM-index can be created by encoding the transformed text W using Move-to-Front encoding and Run-Length encoding [25]. When the alphabet size is small, precisely, when $|\Sigma| \log |\Sigma| = O(\log n)$, Move-to-Front encoding and Run-Length encoding can be done in $O(n)$ time based on a pre-computed table of $o(n)$ bits. In summary, this encoding procedure takes $O(n)$ time using $o(n)$-bit space in addition to the output index. Thus, we have the following result.

Theorem 4.16 *The FM-index of T can be constructed in $O(n \log\log |\Sigma|)$ time and $O(n \log |\Sigma|)$-bit working space, when $|\Sigma| \log |\Sigma| = O(\log n)$.*

4.4 Constructing Ψ_{S_o}

Given $S[0, m-1]$ and Ψ_{S_e}, this section describes how to construct Ψ_{S_o}. Our approach is indirect, as prior to obtaining Ψ_{S_o}, we need to construct the Burrows-Wheeler text C_o of S_o.

Let Δ be the alphabet of S. Define $x[0, m/2-1]$ to be an array of characters such that $x[i] = S[2SA_e[i] - 1]$ where $2SA_e[i] - 1$ is computed in modulo-m arithmetic. Let X_i be the string $x[i]S_e[SA_e[i], m/2-1]S[0]$.

Observation 4.17 *X_i is a suffix of S_o if $SA_e[i] \neq 0$. Otherwise, the first character of X_i is $S[m-1]$, which is unique among other characters in S.*

Let X be the set $\{X_k \,|\, 0 \leq k \leq m/2-1\}$. Intuitively, X is the same as the set of suffixes of S_o. See Figure 4.3(a) for an example of X_i.

Lemma 4.18 *The stable sorting order of $x[i]$ in x equals the rank of X_i in X.*

Proof: By omitting the first characters of every X_i's, they are of the form $S_e[SA_e[i], m/2-1]S[0]$, which are already sorted. Thus, the rank of X_i is equal to the stable sorting order of $x[i]$ in x. □

Lemma 4.19 *Given Ψ_{S_e} and S, we can construct C_o in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: Let $y[0, m/2-1]$ be an array such that $y[i]$ stores the two characters that immediately precedes $x[i]$ in S (i.e., $S[2SA_e[i]-3]S[2SA_e[i]-2]$). In fact, $y[i]$ is

		X_i		
i	$y[i]$	$x[i]$	$S_e[SA_e[i], m/2-1]$	$S[0]$
0	`$a`	`c`	`aa cc g$`	`a`
1	`cg`	`$`	`ac aa cc g$`	`a`
2	`ca`	`a`	`cc g$`	`a`
3	`ac`	`c`	`g$`	`a`

(a)

i	$y \to C_o$	$x \to$ sorted x
0	`cg`	`$`
1	`ca`	`a`
2	`$a`	`c`
3	`ac`	`c`

(b)

Figure 4.3: Consider $S =$ `acaaccg$`. (a) The relationship between $x[i]$, $y[i]$ and X_i. Note that X_i corresponds to a suffix of S_o. (b) After stable sorting on the array x, the array y becomes C_o.

the preceding character of X_i in S_o. Using similar approach as in Lemma 4.6, x and y can be computed in $O(m)$ time, and both arrays occupy $O(m \log |\Delta|)$ bits.

To construct C_o, we perform a stable sort on x as in Lemma 4.5, and iteratively compute the stable sorting order k of $x[i]$, which is equal to the rank of X_i by Lemma 4.18. During the process, we set $C_o[k] = y[i]$. The total time is $O(m+|\Delta|)$ and the total space is $O(m \log |\Delta| + |\Delta|)$ bits. See Figure 4.3 for an example. □

Lemma 4.20 *Ψ_{S_o} can be constructed from C_o in $O(m+|\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: The proof is similar to Lemma 4.5. □

Thus, we conclude this section with the following lemma.

Lemma 4.21 *Given Ψ_{S_e} and S, we can construct Ψ_{S_o} in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: The construction is as follows. First, C_o is constructed from Ψ_{S_e} and S by Lemma 4.19. Then, Ψ_{S_o} is constructed from C_o by Lemma 4.20. The lemma thus follows. □

4.5 Merging Ψ_{S_o} and Ψ_{S_e}

In this section, we construct Ψ_S from Ψ_{S_o} and Ψ_{S_e}. The idea is to determine the rank of any suffix of S among all suffixes of S, and based on this information, we construct the Burrows-Wheeler text C of S. Finally, we convert C to Ψ_S by Lemma 4.5.

Let s be any suffix of S. Observe that the rank of s among the suffixes of S, is equivalent to the sum of the rank of s among the odd-position suffixes and that among the even-position suffixes of S. Based on this observation, we can construct the C array (the Burrows-Wheeler transformation of S) as follows.

Firstly, we construct the auxiliary data structures of Lemma 4.11 for Ψ_{S_o} and for Ψ_{S_e}. Next, we perform backward searches for S_e on Ψ_{S_o} and Ψ_{S_e} simultaneously by Lemma 4.8, so that at step i, we obtain the ranks of $S_e[m/2-i, m/2-1]$ among the odd-position suffixes and even-position suffixes of S, respectively. By summing these two ranks, we get the rank k of $S_e[m/2-i, m/2-1]$ among all suffixes of of S. Then, we set $C[k]$ to be $S[m-2i-1]$, which is the preceding character of the suffix $S[m-2i, m-1] = S_e[m/2-i, m/2-1]$.

Similarly, we perform a simultaneous backward search for S_o on Ψ_{S_o} and Ψ_{S_e} to complete the remaining entries of C. Thus, we obtain C by $O(m)$ backward search steps. The algorithm is depicted as `MergeCSA` in Figure 4.4.

```
MergeCSA

  1. Construct the auxiliary data structures for $\Psi_{S_o}$ and for $\Psi_{S_e}$ to support efficient backward search.

  2. Backward search for $S_e$ on $\Psi_{S_o}$ and $\Psi_{S_e}$ simultaneously, and

      (a) at Step $i$, we obtain the rank of $S_e[m/2-i, m/2-1]$ among the odd-position suffixes and that among the even-position suffixes of $S$. Let the sum of the ranks be $k$.

      (b) Set $C[k] = S[m-2i-1]$.

  3. Backward search for $S_o$ on $\Psi_{S_o}$ and $\Psi_{S_e}$ simultaneously, and fill in $C[k]$ accordingly.
```

Figure 4.4: Merging Ψ_{S_o} and Ψ_{S_e}.

The following lemma shows the correctness of our algorithm.

Lemma 4.22 *The algorithm* `MergeCSA` *in Figure 4.4 correctly constructs* $C[0, m-1]$.

Proof: Recall that for every suffix $S[i, m-1]$, $C[SA^{-1}[i]]$ equals the preceding character of $S[i, m-1]$. For every even-position suffix $S[i, m-1] = S_e[i/2, m/2-1]$, Step 2 computes its rank k among all odd-position and even-position suffixes. By definition, $k = SA^{-1}[i]$. Therefore, Step 2 correctly assigns $C[k]$ to be the preceding character of $S[i, m-1]$. By the same argument, Step 3 handles the odd-position suffixes and correctly assigns $C[SA^{-1}[i]]$ to be the preceding character of $S[i, m-1]$.

Therefore, after Steps 2 and 3, `MergeCSA` completely constructs $C[0, m-1]$. The lemma thus follows. $\square$

By Lemma 4.11, the auxiliary data structures can be constructed in $O(m + |\Delta|)$ time and $O(m+|\Delta|)$-bit space, and then each backward search step is done in $O(\log\log|\Delta|)$ time. On the other hand, the Ψ functions occupies $O(m\log|\Delta|)$-bit space. Thus, we have the following lemma.

Lemma 4.23 *Given Ψ_{S_o} and Ψ_{S_e}, we can construct Ψ_S in $O(m\log\log|\Delta|+|\Delta|)$ time and $O(m\log|\Delta| + |\Delta|)$-bit space.*

4.6 Improvement when $\log|\Sigma| = O((\log\log n)^{1-\epsilon})$

In case when the alphabet size is small, precisely, when $\log|\Sigma| = O((\log\log n)^{1-\epsilon})$, we can improve the construction time of CSA and FM-index to $O(n)$, which is optimal. The improvement is based on the following data structure of Pagh for supporting constant-time `rank` queries [66].

Theorem 4.24 [66] *Given n distinct numbers in $[0, m-1]$ such that $m = n\log^{O(1)} n$, a data structure of size $B + O(\frac{n(\log\log n)^2}{\log n})$ bits supporting constant-time `rank` queries can be constructed in $O(n)$ time and $O(B)$-bit space where $B = \lceil\log\binom{m}{n}\rceil = n\log\frac{m}{n} + O(n)$.*

We apply the same algorithm as in Section 4.3 for the construction of CSA, but we make changes only in the encodings of Ψ_{T^i}, for $i < \log\log\log|\Sigma|$. For those values of i, we have $|T^i| = n/2^i$ and the alphabet size of T^i is $|\Sigma|^{2^i}$. When $\log|\Sigma| = O((\log\log n)^{1-\epsilon})$, we have $|\Sigma|^{2^i} = \log^{O(1)}|T^i|$.[‡] Thus, the total increasing sequence of such Ψ'_{T^i}'s can be encoded by Theorem 4.24, and each backward

[‡]This can be seen by considering the boundary case of $i = \log\log\log|\Sigma|$, so that $|T^i|$ becomes smallest while the alphabet size becomes largest.

search step on these Ψ functions can be done in constant time. This gives the following theorem.

Theorem 4.25 *The CSA, FM-index and the Burrows-Wheeler text W of T can be constructed in $O(n)$ time and $O(n \log|\Sigma|)$-bit working space if* $\log|\mathcal{A}| = O((\log\log n)^{1-\epsilon})$.

Proof: We refer to the algorithm in Figure 4.2. After the change in the encodings of Ψ_{T^i} for $i < \log\log\log|\Sigma|$, the time required by each phase is as follows.

- Phase 1 takes $O(n)$ time;
- For $i \geq \log\log\log|\Sigma|$, Step i in Phase 2 takes $O((n/2^i)\cdot(i+\log\log|\Sigma|)+|\Sigma|^{2^i})$ time;
- For $i < \log\log\log|\Sigma|$, Step i in Phase 2 takes $O(n/2^i + |\Sigma|^{2^i})$ time.

It follows that the total time required is $O(n)$. For the space complexity, it remains $O(n\log|\Sigma|)$ bits. Thus, the CSA and the Burrows-Wheeler text W of T can be constructed in the stated time and space, while the FM-index can be constructed in $O(n)$ time and $O(n\log|\Sigma|)$-bit space once W is obtained. This completes the proof. $\square$

Chapter 5

Construction of Compressed Suffix Trees and its Application for Finding Maximal Unique Matches

In this chapter, we first give the definition of Compressed Suffix Tree (CST) of Sadakane [71], and show that CST can be converted from CSA in $O(n \log^{\epsilon} n)$ time. Afterwards, we introduce the problem of finding the *maximal unique matches* (MUMs) between two texts, and propose efficient algorithm based on CST.

Before we give the definition of the CST, we introduce its basic components as follows.

Definition 5.1 *The parentheses encoding of an ordered tree is defined by at most $2n$ nested open and close parentheses, which can be constructed by a preorder traversal of the tree, where we write an open parenthesis when a node is first visited, then the parentheses encodings for the subtrees of the child node according to their order, and finally write a close parenthesis when it is last visited.*

The parentheses encoding is used to store the topology of the suffix tree. However, it does not store the information of edge lengths or edge labels. Such information is stored using the *Hgt* array as follows.

Definition 5.2 *Let $LCP(T_1, T_2)$ denote the length of the longest common prefix between T_1 and T_2. Then, the array $Hgt[0, n-2]$ is defined such that $Hgt[i] = LCP(T_{SA[i]}, T_{SA[i+1]})$.*

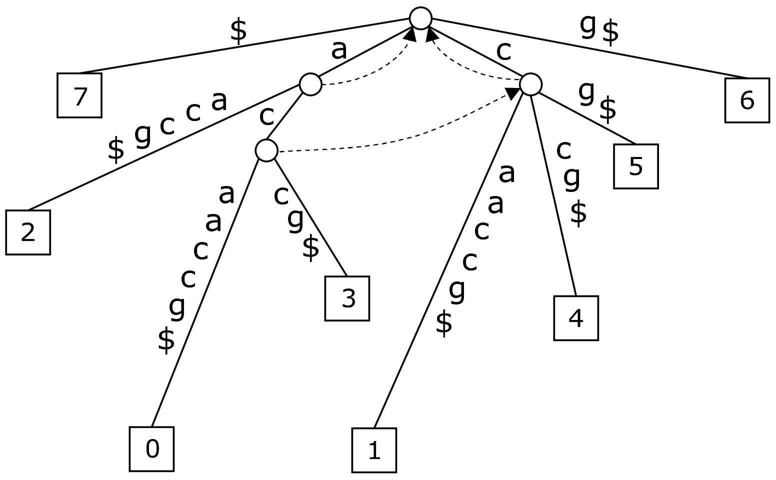
The *Hgt* array occupies $n \log n$ bits without compression. However it can be represented in linear size by using the suffix array or the compressed suffix array.

Lemma 5.3 ([73]) *Given i and $SA[i]$, the value $Hgt[i]$ can be computed in constant time using a data structure of size $2n + o(n)$ bits.*

For example, the parentheses encoding, and the *Hgt* array for the suffix tree of `acaaccg`$ are shown in Figure 5.1.

Formally, the CST for a text T is defined as follows.

Definition 5.4 *The compressed suffix tree of a string T consists of the CSA of T, the Hgt array, and the parentheses encoding of the topology of the suffix tree of T.*



i	0	1	2	3	4	5	6
$Hgt[i]$	0	1	2	0	1	1	0

Parentheses Encoding: (()(()(()()))(()()())())

Figure 5.1: The suffix tree of the text T = `acaaccg$`, the *Hgt* array, and the parentheses encoding. Note that each () in the parentheses encoding corresponds to a leaf in the suffix tree.

We have the following theorem concerning the performance of the CST.

Theorem 5.5 ([74]) *For a text T over a constant-size alphabet, the CST of T occupies $O(n)$ bits, which can simulate each navigation operation in the suffix tree of T in $O(\log^{\epsilon} n)$ time, and any leaf label can be retrieved in $O(\log^{\epsilon} n)$ time.*

Remark. In fact, by modifying the definition of the parentheses encoding slightly, we obtain the following result based on the previous theorem.

Theorem 5.6 *For a text T over an alphabet Σ, the CST occupies $O(n \log |\Sigma|)$ bits, which can simulate each navigation operation in the suffix tree of T in $O(\log^\epsilon n + \log |\Sigma|)$ time, and any leaf label can be retrieved in $O(\log^\epsilon n)$ time.*

5.1 Conversion from CSA to CST

In this section, we describe a space-efficient construction algorithm for CST of T.

Firstly, assume that the CSA of T is given. Then, we augment the CSA with an auxiliary data structure of $O(n \log |\Sigma|)$ bits, which can support the retrieval of any entry of SA^{-1} in $O(\log^\epsilon n)$ time. The above auxiliary data structure is similar to that for the retrieval of SA, and can be constructed in $O(n \log |\Sigma|)$ time [72].

In [46], it is shown that the $Hgt[i]$ can be enumerated iteratively with $O(n)$ queries to the text, SA and SA^{-1}. Based on this algorithm, the *Hgt* array can easily be encoded in $2n+o(n)$ bits without extra working space (See [73] for more details on the *Hgt* array). The time required is $O(n \log^\epsilon n)$ time since each of the query can be answered in $O(\log^\epsilon n)$ time given the augmented CSA.

Next, we construct the parentheses representation for the suffix tree topology. Basically, with the provision of the *Hgt* array and the CSA, we can simulate a bottom-up (left to right) traversal of the suffix tree using the algorithm in [46]. Let L be a list of $2n$ bits. During the traversal, we append () to the end of L if we visit a leaf, and we append) to the end of L if we last visit an internal node. Essentially, after the traversal, L is very close to the parentheses encoding of the CST, only with all the ('s corresponding to the internal nodes removed. Similarly, by performing a bottom-up (right to left) traversal starting from the rightmost leaf, we can obtain a list of parentheses L' which is equal to the parentheses encoding of the CST, but with all the)'s corresponding to the internal node removed. The above algorithm requires retrieval of $O(n)$ queries to the *Hgt* array and the CSA, so that it takes $O(n \log^\epsilon n)$ time in total. Afterwards, it is easy to combine L and L' in $O(n)$ time to obtain the parentheses encoding of the CST.

For the space complexity, the original algorithm of [46] requires the maintenance of a stack, which takes $O(n \log n)$ bits without compression. Fortunately, the values in the stack are always monotone, so that we can encode them with the difference encoding. Then, the space required to store the numbers is bounded by $O(n)$ bits if the differences are encoded by γ code or δ code [20].

Thus, we have the following theorem.

Theorem 5.7 *Given the text T, the CST of $T[0, n-1]$ can be constructed in $O(n \log^{\epsilon} n)$ time and in $O(n \log |\Sigma|)$-bit working space.*[*]

5.2 Finding Maximal Unique Matches

In this section, we show how to find the MUMs between two texts A and B efficiently. Formally, it is defined as follows.

Definition 5.8 *A pattern P is said to be a* unique match *(UM) of two strings A and B if P appears exactly once in both A and B. A pattern P is said to be a* maximal unique match *(MUM) if P is a UM and it is not contained in any other UMs.*

Note that unique match and maximal unique match are similar to maximal repeat and super-maximal repeat [38], respectively.

Lemma 5.9 *If an internal node v of a suffix tree for $A\$_1B\$_2$ has an internal node as its child, or has more than three children, then the path label of v is not an MUM of A and B.*

Proof: Let the internal node be v. In both cases, the path lablel of v is a common prefix of at least three suffixes of $A\$_1B\$_2$. This implies that the path label of v appears in A or B at least twice, so that it cannot be an MUM. □

On the other hand, it is easy to verify that an MUM of A and B must be the path label of some internal node in the suffix tree for $A\$_1B\$_2$. Therefore, an MUM of A and B must be a path label of an internal node with exactly two leaves, one for the suffix in A and the other for the suffix in B. (Note that the converse is not true, because the path label may not be maximal.)

Based on the suffix tree of $A\$_1B\$_2$, we can find the MUMs as follows:

[*]Note that this time and space bound also hold for the CST in Theorem 5.6 that is based on another parentheses encoding.

1. Mark each internal node v with exactly two leaves, one from a suffix in A and the other from a suffix in B.
2. For each internal v, unmark the suffix link of v
3. Report the path labels of all marked nodes

Note that in Step 3, we unmark the suffix link of v even if v is not marked, which implies that we can actually unmark the nodes in any order. This observation is crucial to this algorithm, and our later algorithm as well. The correctness is shown in the following lemma. Note also that the total length of MUMs can be much larger than $O(n)$, so that we report each MUM implicitly, by a tuple (j, ℓ) for the path label $A[j, j+\ell-1]$, to save the time.

Lemma 5.10 *Suppose the length of the string $A\$_1B\$_2$ is n. The above algorithm reports all the MUMs of A and B in $O(n)$ time.*

Proof: Step 1 of the algorithm marks all nodes that correspond to an MUM. Step 2 correctly unmarks all the internal nodes that cannot correspond to an MUM for the following reason: Let u be the suffix link of v. If u is an UM, then v must also be an UM. This implies that u can never be an MUM. Step 3 outputs all the MUMs, assuming that each node in the suffix tree has stored the tuple representing its path label. As each step can be done in $O(n)$ time, the lemma follows. □

Now, we are ready to show our compressed solution for finding MUMs.

5.2.1 Finding MUMs with CSA

Firstly, we show a trivial algorithm for the case when we are given the CSA for the string $A\$_1B\$_2$. The algorithm is as follows.

```
1. Construct the $Hgt$ array of $A\$_1B\$_2$.
2. **For** $i = 0, 1, \ldots, n-1$
       **if** $Hgt[i] > Hgt[i-1]$
           and $Hgt[i] < Hgt[i+1]$
           and $leaf(i)$ and $leaf(i+1)$ come from different strings
           and $T[\mathrm{SA}[i]-1]$ and $T[\mathrm{SA}[i+1]-1]$ differ
       **then** Output $(\mathrm{SA}[i], Hgt[i])$ to represent the substring
           $T[\mathrm{SA}[i], \mathrm{SA}[i]+Hgt[i]-1]$.
```

The correctness of the above algorithm is shown below.

Theorem 5.11 *The above algorithm reports all MUMs of A and B in $O(n \log^{\epsilon} n)$ time and requires $O(n \log |\Sigma|)$-bit working space.*

Proof: The first two conditions in the **if** statement guarantee that we are dealing with a node which has exactly two leaf children, one corresponding to a suffix of A and the other corresponding to a suffix of B. Thus, the node corresponds to a UM. The third condition in the **if** statement guarantees the maximality of UM. Thus, any node passing all the three conditions would correspond to an MUM. On the other hand, we can see that the converse is also true. Thus, the above algorithm correctly reports all MUMs of A and B.

For the time complexity, since $Hgt[i]$ and $T[\mathrm{SA}[i]-1]$ can be computed in $O(\log^{\epsilon} n)$ time, the algorithm runs in $O(n \log^{\epsilon} n)$ time. The space required consists of the CSA and the Hgt array, which is $O(n \log |\Sigma|)$ bits. □

5.2.2 Finding MUMs with CST

The algorithm with CSA in Section 5.2.1 takes $O(n \log^{\epsilon} n)$ time because some basic operations of the CSA requires $O(\log^{\epsilon} n)$ time. Interestingly, we can solve the problem in $O(n)$ time when the CST is given, even if CSA is a core part of the CST. Note that the CST does not have suffix links. However, we can simulate suffix links by using CSA.

A high-level description of the algorithm is shown below, which is followed the details of each step.

1. Compute a bit-vector $D[0, n-1]$ such that $D[i] = \mathtt{1}$ if *leaf*(i) is a suffix from A, and $D[i] = \mathtt{0}$ otherwise.
2. Mark nodes which have exactly two leaves, one from A and the other from B.
3. Unmark non-maximal nodes.
4. Report all MUMs.

In Step 1, to compute the bit-vector, we compute $i = p, \Psi[p], \Psi^2[p], \Psi^3[p], \ldots$, where $p = \mathrm{SA}^{-1}[0]$. We set $D[i] = \mathtt{1}$ for the first $|A|+1$ iterations, and set $D[i] = \mathtt{0}$ for the remaining $|B|+1$ iterations. This takes $O(n)$ time because computing each Ψ takes constant time.

In Step 2, we scan the parentheses encoding of the suffix tree from left to right to find a pattern '(()())' which corresponds to an internal node with exactly two leaves. Let i be the rank of the left leaf, which can be reported in constant time while we scan the parentheses encoding. We mark this i if leaf i and leaf $i+1$ in the suffix tree correspond to suffixes from different strings. To do so, we use another bit-vector $V[0, n-2]$, and set $V[i] = \texttt{1}$ if $D[i] \neq D[i+1]$.

Then, Step 3 and Step 4 can be done as follows.

```
3.1. Define a temporary bit-vector $W[0, n-2]$.
3.2. **For** $i = 0, 1, \ldots, n-2$, set $W[i] = V[i]$.
3.3. **For** $i = 0, 1, \ldots, n-2$
         **if** $\Psi[i] + 1 = \Psi[i+1]$
             Set $W[\Psi[i]] = \texttt{0}$.
4.1. Set $i = \mathrm{SA}^{-1}[0]$.
4.2 **For** $j = 0, 1, \ldots, n-2$
         **if** $V[i] = \texttt{1}$ and $W[i] = \texttt{1}$
             Output $(j, Hgt[i])$ that represents the substring
             $T[j, j + Hgt[i] - 1]$.
         Set $i = \Psi[i]$.
```

Then, we have the following theorem which gives the timing and correctness of the algorithm with CST.

Theorem 5.12 *The above algorithm reports all MUMs of A and B in $O(n)$ time and $O(n \log |\Sigma|)$-bit working space.*

Proof: After Step 1 and Step 2, the bit-vector V stores all the nodes whose path label is a candidate of MUM. All the internal nodes which has two leaf children, and is pointed by some other node with a suffix link are unmarked by Step 3.3, while the internal nodes for an MUM are unaffected. Thus, it follows that if $V[i] = \texttt{1}$ and $W[i] = \texttt{1}$ after Step 3, the path label represented by $(\mathrm{SA}[i], Hgt[i])$ is the MUM of A and B. This completes the proof of the correctness.

For the time complexity, Step 1 and Step 3 are done in $O(n)$ time because computing a value of Ψ takes constant time. Step 2 takes $O(n)$ time for the scanning. For Step 4, although it involves queries to the *Hgt* array, it takes $O(n)$ time because we know $\mathrm{SA}[i] = j$ in each iteration, so that we can compute each $Hgt[i]$ in constant time. Thus, the total time is $O(n)$. For the space complexity, it requires $O(n \log |\Sigma|)$ bits for the CST, and $O(n)$ bits for the bit-vectors. The theorem thus follows. $\square$

Chapter 6

Managing a Dynamic Library

The problem of managing a dynamic library is defined as follows: We need to maintain a collection $\mathcal{L}$ of texts of total length n, with characters drawn from an alphabet Σ; from time to time, a text may be inserted or deleted from $\mathcal{L}$, and a pattern P may be given and its occurrences in $\mathcal{L}$ are to be reported.

In this chapter, we introduce a compressed index for the dynamic library management problem, which requires only $O(n)$ bits for constant-size alphabet. Inserting or deleting a text of length t takes $O(t \log n)$ time, while searching for a pattern takes $O(p \log n + \mathit{occ} \log^2 n)$ time. Note that the time complexities of all operations are measured in the worst case. To our knowledge, this is the first result that requires only $O(n)$ bits, yet supporting both update and searching efficiently, i.e., in $O(t \log^{O(1)} n)$ and $O((p + \mathit{occ}) \log^{O(1)} n)$ time, respectively.

Technically speaking, our compressed index is based on CSA and FM-index to support efficient pattern searching. However, the CSA and the FM-index we employ are designed so as to allow efficient updates. This is achieved through a dynamic representation of them. In addition, we have a simple trick in organizing the texts to avoid the use of pointers, thus eliminating the $O(m \log n)$ bits generally required, where m denotes the number of texts in $\mathcal{L}$.

The remaining of this chapter is organized as follows. In the first part, we describe the high level organization of our index, where we introduce the dynamic representations of CSA and FM-index for a collection of texts, and show how to use them to achieve our stated result. For the implementation details, they are deferred to the second part of this chapter.

Remarks. In case of a variable-size alphabet, the FM-index in our compressed index occupies $O(n|\Sigma|)$ bits, which may become a problem if $|\Sigma|$ is huge. Nevertheless, we can reduce the space complexity to $O(n \log |\Sigma|)$ bits by keeping the

CSA alone, while the use of FM-index is simulated by the CSA when needed (See Lemma 2.7). Afterwards, inserting or deleting a text of length t takes $O(t \log^2 n)$ time, while searching for a pattern takes $O(p \log^2 n + occ \log^2 n)$ time.

6.1 High Level Organization

This section gives the high level description of our compressed index. In particular, we introduce three data structures, namely, *COUNT*, *MARK*, and *PSI*, that correspond to the dynamic representations of the *count* functions of FM-index, the auxiliary data structure to retrieve suffix array entries, and the Ψ function of CSA, respectively.

Below we first introduce *COUNT*, which is the core data structure that already supports counting the occurrences of a pattern P in $\mathcal{L}$ efficiently, and fast insertion or deletion of texts. Afterwards, we discuss how to exploit *MARK* and *PSI* to support efficient enumeration of the positions where a pattern P occurs, and further speed up the updating process.

Consider a set of texts $\mathcal{L} = \{T_1, T_2, \ldots, T_m\}$ over a constant-size alphabet Σ. We assume that the texts are distinct, and each text T *starts* with a special character `$` in Σ, which is alphabetically smaller than all other characters in Σ and `$` does not appear in any other part of a text.

Denote the total length of texts as n. In case the contents of the text collection is changed, we always label the existing texts in $\mathcal{L}$ in such a way that T_j refers to the rank-j texts currently in $\mathcal{L}$.

Conceptually, we want to construct a suffix array SA for the texts by listing out all suffixes of all texts in lexicographical order. For $i = 0, 1, 2, \ldots, n-1$,

$$\mathrm{SA}[i] = (j, k)$$

if the suffix $T_j[k, |T_j| - 1]$ is the rank-i suffix among all suffixes. Then, to insert a text T to $\mathcal{L}$, we insert all suffixes of T into the SA. Similarly, to delete a text from $\mathcal{L}$, we delete all suffixes of T from SA. Searching for a pattern P is done by determining the interval $[x, y]$ such that each suffix from $\mathrm{SA}[x]$ up to $\mathrm{SA}[y]$ has P as a prefix. $\mathrm{SA}[x], \mathrm{SA}[x+1], \ldots, \mathrm{SA}[y]$ gives all locations where P occurs in $\mathcal{L}$.

6.1.1 The *COUNT* Data Structure

Due to the space restriction, we cannot directly store the SA table. Instead, we use the FM-index, which requires only $O(n)$ bits, to represent the SA table implicitly. The FM-index for $\mathcal{L}$ consists of the functions *count*(c, i) for each c in Σ, which returns the total number of occurrences of character c in $W[0, i]$, where $W[i]$ is defined as the character $T_j[k-1]$ if $\mathrm{SA}[i] = (j, k)$.* Note that this definition is analogous to the original definition of FM-index for a single text, so that we refer W as the Burrows-Wheeler transformation of $\mathcal{L}$.

We implement the *count*(c, i) function with a dynamic data structure *COUNT*, whose performance is summarized in the lemma below. The proof (i.e., the detailed construction of *COUNT*) will be in Section 6.2.1.

Lemma 6.1 *We can maintain the COUNT data structure using $O(n)$ bits space such that each of the following operations is supported in $O(\log n)$ time.*

- *Report(c, i): Returns the value of count(c, i).*
- *Insert(c, i): Updates all count functions due to a character c inserted to the position i of W.*
- *Delete(i): Updates all count functions due to a character deleted from the position i of W.*

Note that *COUNT* allows efficient updates which is needed when the W array is changed due to insertion or deletion of texts. Our implementation is different from the original one in [25], where the *count* functions are stored in a data structure which is difficult to update (but allows constant-time query).

Pattern matching. We define $\#(c)$ for each character c to be the number of suffixes whose first character is c, and $\alpha(c)$ to be the rank of the character c among all the suffixes. We maintain the $\#(c)$ values explicitly. Then, since the alphabet size is constant, $\alpha(c)$ or $T[\mathrm{SA}[i]]$ can be computed in constant time.† Together with the *COUNT* data structure, we can support counting the occurrences of $P[0, p-1]$ in $\mathcal{L}$ in $O(p \log n)$ time, using the traditional backward search as follows: Firstly, the rank of $P[p-1]$ among all suffixes can be computed by $\alpha(P[p-1])$. Then, by Lemma 2.4, we can find the rank of $P[p-2, p-1]$ using one query to the *count* functions. The process is repeated, so that eventually we can find the rank (say, x) of $P[0, p-1]$. Similarly, we can find the rank (say, y)

*Precisely, the index $k-1$ in $T_j[k-1]$ is defined under modulo-$|T_j|$ arithmetic.

†For general alphabet, we can organize the $\#(c)$ values in a 'differential' red-black tree [51], so that $\#(c)$, $\alpha(c)$ or $T[\mathrm{SA}[i]]$ can be retrieved in $O(\log |\Sigma|)$ time. Updating takes $O(\log |\Sigma|)$ time as well.

of $P[0, p-1]\mathtt{z}$, where z is assumed to be an arbitrary string of rank n among all suffixes. Then, the number of occurrences of P is $y - x$. As the whole process requires $O(p)$ queries to the *count* functions and each query takes $O(\log n)$ time, the total time follows.

Text insertion. To insert a text $T[0, t-1]$, we conceptually insert all the suffixes of T to SA, starting from the shortest one. The rank of $T[t-1]$ among the suffixes stored in SA, denoted as i, is $\alpha(T[t-1])$. Conceptually, we want to insert $T[t-1]$ at the $(i+1)$-th row in SA. However, as the SA table is not stored explicitly, we reflect the change in SA by the corresponding change in W instead, where we insert the character $T[t-2]$ to the $(i+1)$-th position of the W array. This is done by performing *Insert*$(T[t-1], i)$ provided by *COUNT*. Next, to insert (conceptually) the suffix $T[t-2, t-1]$ to SA, let i' be the rank of $T[t-2, t-1]$ among the suffixes stored in the updated SA, which can be found easily by one backward search step in the updated *COUNT* data structure. The required change in SA is reflected by inserting $T[t-3]$ to the $(i'+1)$-th position of W. The process continues until the longest suffix $T[0, t-1]$ is inserted to SA, which is reflected by inserting $T[t-1]$ to W. The whole process takes $O(t \log n)$ time.

Text Deletion. Deleting a text $T[0, t-1]$ from the collection of texts is more troublesome because among all those single-character suffixes that equals to $T[t-1]$, we do not know which one belongs to T.[‡] To handle the problem, we first perform a backward search for $T[0, t-1]$ and let $[x, y]$ be the interval such that for any i in $[x, y]$, $T[0, t-1]$ is a prefix of SA$[i]$. Recall that all texts in the collection are distinct and each of them starts with a special character `$` which is alphabetically smaller than all other characters. Thus, we can conclude that SA$[x]$ corresponds to the text $T[0, t-1]$ to be deleted because no other text can be lexicographically less than $T[0, t-1]$ and have $T[0, t-1]$ as a prefix.

Then, performing *Delete*(x) provided by *COUNT*, we can (conceptually) delete the suffix $T[0, t-1]$ and update the SA accordingly. Afterwards, we repeat the process to delete the remaining suffixes $T[i, t-1]$ for $i = 1, 2, ..., t-1$, i.e., from the longest one to the shortest one. This is done by first computing the rank x' of $T[i, t-1]$ among the suffixes stored in the updated SA, and then performing

[‡]We assume that the suffixes of all texts in $\mathcal{L}$ each has a distinct rank, even if they are the same in appearance. As can be seen from the above discussion, the relative rank among equal suffixes is fixed according to the order of insertion.

Delete(x').

Note that if the Ψ function of CSA is given,[§] we can compute the rank of $T[i, t-1]$ easily from the rank of $T[i-1, t-1]$. However, as the Ψ function is not available, we need to simulate each query to Ψ by $O(\log n)$ queries to the *count* functions. Since a query to *count* takes $O(\log n)$ time, deleting each suffix of $T[0, t-1]$ takes $O(\log^2 n)$ time, and the whole process takes $O(t \log^2 n)$ time.

Summarizing the discussion, we have the following theorem.

Theorem 6.2 *Let $\mathcal{L} = \{T_1, T_2, \cdots, T_m\}$ be a set of m distinct strings over a constant-size alphabet Σ. Let n be the total length of all strings in $\mathcal{L}$. We can maintain $\mathcal{L}$ in $O(n)$-bit space such that counting the occurrences of a pattern $P[0, p-1]$ takes $O(p \log n)$ time, inserting a text $T[0, t-1]$ takes $O(t \log n)$ time, and deleting a text $T[0, t-1]$ takes $O(t \log^2 n)$ time.*

6.1.2 The *MARK* Data Structure

The *COUNT* data structure in the previous discussion does not support retrieving SA$[x]$ and thus cannot report the positions where a pattern occurs. In the following, we give an additional data structure called *MARK* for the retrieval of SA values.

Recall that all texts in $\mathcal{L}$ start with the character \$ which is lexicographically smaller than any other character in Σ. As a result, for the set of m texts in $\mathcal{L}$, the first m entries of SA corresponds to the m texts sorted in lexicographical order.

Consider the entries SA$[i] = (j, k)$ where k is a positive integral multiple of $\log n$. There are at most $n/\log n$ such entries and our *MARK* data structure stores a tuple $(i, (j, k))$ for each of them. Now, suppose that given a certain x, we want to find the value (j, k) with SA$[x] = (j, k)$. We first check whether SA$[x]$ is stored in *MARK*. If so, we obtain the value of SA$[x]$ immediately. Otherwise, we check whether $x \leq m$, which would imply that SA$[x] = (x, 0)$. If both cases are false, we can determine the rank of the suffix $T_j[k-1, |T_j|-1]$, denoted as x', easily using backward search with the *COUNT* data structure. We check whether the entry SA$[x']$ is stored in *MARK* or $x' \leq m$. The process continues and after $\log n$ steps, we must either meet a suffix $T_j[k-r, |T_j|-1]$ such that $k-r$ is a multiple of $\log n$, or $k-r=0$. In both cases, the value of (j, k) can be found accordingly.

[§]If SA$[i] = (j, k)$, then $\Psi[i]$ is the rank of $T_j[k+1, |T_j|-1]$ among the suffixes of all texts, where $k+1$ is computed under modulo-$|T_j|$ arithmetic.

As to be shown in Lemma 6.3, for any value x, *MARK* determines whether the tuple SA[x] is stored (and if so, reports its value) in $O(\log n)$ time. Thus, it takes $O(\log^2 n)$ time to find the value of SA[x] for any value x.

When a suffix is inserted to or deleted from SA, some of the originally stored tuples may require updates. For example, when a new suffix with rank u among the existing suffixes is inserted, a tuple (j, k), corresponding to SA[v] originally, becomes a tuple corresponding to SA[$v+1$] if $v \geq u$. That is, we may need to update the i-value of a stored tuple whenever a suffix is inserted to or deleted from SA. Also, the rank of an original text T_j in $\mathcal{L}$ may change due to text insertion or deletion, so that we may need to update the j-value of a stored tuple.

Bearing the above concern in mind, *MARK* must allow a set of operations for handling the updates carefully. We summarize the performance of *MARK* in the lemma below. Note that *MARK* stores at most $n/\log n$ entries of SA, and the i-values of the stored tuples are distinct. We defer the proof in Section 6.2.2.

Lemma 6.3 *Consider the entries* SA[i] $= (j, k)$ *where k is a positive integral multiple of* $\log n$. *We can maintain a data structure MARK in $O(n)$ bits for storing the tuples $(i, (j, k))$ for each of these entries, such that each of the following operations is supported in $O(\log n)$ time.*

- *Report(i): Returns the $(i, (j, k))$ if this tuple is stored. Else, return false.*
- *Insert(i, j, k): Inserts the tuple $(i, (j, k))$ to MARK.*
- *Delete(i): Deletes the tuple $(i, (j, k))$ from MARK.*
- *Increment_lexico(ℓ): For each tuple stored, the j-value is incremented by one if the original j-value is at least ℓ. This function allows us to update the rank of the original texts after a new text with lexicographical order ℓ is inserted.*
- *Decrement_lexico(ℓ): For each tuple stored, the j-value is decrement by one if the original j-value is greater than ℓ.*
- *Shift_up(ℓ): For each tuple stored, the i-value is incremented by one if the original i-value is at least ℓ. This function allows us to update the correspondence between tuples and* SA *after a new suffix is inserted to position ℓ of* SA.

- *Shift_down(ℓ): For each tuple stored, the i-value is decremented by one if the original i-value is greater than ℓ.*

With *COUNT* and *MARK*, we can find the positions where a pattern $P[0, p-1]$ occurs in the collection of texts in $O(p \log n + occ \log^2 n)$ time.

6.1.3 The *PSI* Data Structure

Recall that to delete a text $T[0, t-1]$ in *COUNT*, we first determine the location of $T[0, t-1]$ in SA. Then, we delete all the suffixes of T starting from the longest one. The bottleneck for the deletion operation is determining the rank of $T[i, t-1]$ after the deletion of the suffix $T[i-1, t-1]$. We observe that CSA provides a good solution for it. In fact, the Ψ function of CSA stores exactly the information we needed.

However, we cannot use the original implementation of Ψ as we need to update Ψ efficiently. We dynamize Ψ with the data structure *PSI* whose performance is summarized in the lemma below. The proof of the lemma is presented in Section 6.2.3.

Recall that Ψ is a list of n integers such that if $\text{SA}[i] = (j, k)$, then $\Psi[i]$ is the rank of $T_j[k+1, |T_j|-1]$ among the suffixes of all texts (where $k+1$ is computed under modulo-$|T_j|$ arithmetic).

Lemma 6.4 *We can maintain the PSI data structure in $O(n)$ bits such that each of the following operations is supported in $O(\log n)$ time.*

- *Report(i): Returns $\Psi[i]$.*
- *Insert(i, x): Inserts the integer x to position i of the list. This function is needed when we insert a suffix to* SA.
- *Delete(i): Deletes the integer from position i of the list.*
- *Shift_up(ℓ): Each integer in the list with value at least ℓ is incremented by one. This function is needed when we insert a suffix to position ℓ of* SA.
- *Shift_down(ℓ): Each integer in the list with value greater than ℓ is decremented by one.*

With the *PSI* data structure, insertion and deletion of a text of length t can both be improved to $O(t \log n)$ time.

6.1.4 All in a Nutshell

We summarize how the search, insert and delete operations are performed with *COUNT*, *MARK*, and *PSI*.

Searching for a pattern $P[0, p-1]$. We perform backward search to determine the interval $[x, y]$ such that for each i in $[x, y]$, SA$[i]$ corresponds to an occurrence of P. This can be done in $O(p \log n)$ time using the *COUNT* data structure. Then, for each i in $[x, y]$, the value of SA$[i]$ is obtained by at most $\log n$ backward search steps, with one query to *MARK* in each step. Thus, the time is $O(p \log n + occ \log^2 n)$.

Inserting a text $T[0, t-1]$. Intuitively, we insert each suffix of T to SA starting from the shortest one. For $x = t-1, t-2, \ldots, 0$, we first determine the rank (say, r) of $T[x, t-1]$ among the existing suffixes. Then, to simulate the effect of inserting $T[x, t-1]$ into position r of SA, we update *COUNT* by inserting $T[x-1]$ to position r of W. Then, we update *PSI* by incrementing all integers in the stored list whose value at least r, insert the rank of $T[x+1, t-1]$ to position r of *PSI*, and increment $\#(T[x])$ by one.

The *MARK* data structure is updated as follows. We first determine the rank r' of T among all texts in $\mathcal{L}$ using backward search algorithm. This takes $O(t \log n)$ time. Then, we update the j-value of all tuples in *MARK* such that for each tuple with j-value at least r', we increment its j-value by one. Then, when each suffix of T is inserted to SA (whose rank is r among all existing suffixes), we increment the i-value for any tuples whose i-value is at least r. Finally, we insert tuples corresponding to T to *MARK*. The total time is $O(t \log n)$.

Deleting a text $T[0, t-1]$. Intuitively, we delete each suffix of T starting from the longest one. We first determine the rank of T among all suffixes of all texts. Afterwards, the rank of the other suffixes of T can be found using the *PSI*. Updating of *COUNT*, *MARK*, and *PSI* are done similarly to that of inserting a text, except that we are decrementing the values this time. The total time required is $O(t \log n)$ as well.

Adjustment due to huge updates. Note that in the above discussion, our data structures require the value of $\lceil \log n \rceil$ as a parameter, and we have assumed that this value is fixed over the time. This is not true in general as texts can be inserted or deleted in the collection. Thus, when the value of $\lceil \log n \rceil$ changes, our data structures should be changed basing on a different parameter. A simple way to handle this is to reconstruct everything when necessary, but this would imply huge update time, say, $O(n)$ time, on the single update operation that induces the change. To avoid this, we use a standard technique for global rebuilding [65], where we maintain three copies of each data structure, one based on the current parameter x, and the other two partially constructed based on the parameters $x-1$ and $x+1$, respectively, and distribute the reconstruction process over each update operation. In this way, we can bound the update time to be $O(t \log n)$, while having a new data structure ready when $\lceil \log n \rceil$ is changed.

Summarizing the results, we have the following theorem.

Theorem 6.5 *Let $\mathcal{L} = \{T_1, T_2, \cdots, T_m\}$ be a set of m distinct strings over a constant-size alphabet Σ. Let n be the total length of all strings in $\mathcal{L}$. We can maintain $\mathcal{L}$ in $O(n)$-bit space such that inserting or deleting a text $T[0, t-1]$ takes $O(t \log n)$ time, and searching for a pattern $P[0, p-1]$ takes $O(p \log n + occ \log^2 n)$ time, where occ is the total of occurrences.*

6.2 Implementation Details

In this section, we explain how each of data structures *COUNT*, *MARK*, and *PSI* is implemented.

6.2.1 Implementation of *COUNT*

Recall that the *COUNT* data structure maintains the functions *count*(c, i), which returns the total number of occurrences of the character c in $W[0, i-1]$, where W is the Burrows-Wheeler transformation of $\mathcal{L}$. To implement *COUNT*, we store $|\Sigma|$ lists of bits, denoted as $COUNT_c$ for each c in Σ. Each list is n-bit long, with $COUNT_c[i] = \mathtt{1}$ if $W[i] = c$ and $COUNT_c[i] = \mathtt{0}$ otherwise.

To support updates easily, for each list $COUNT_c$, we partition it into segments of $\log n$ bits to $2 \log n$ bits. The segments are stored in the nodes in a red-black tree, so that a left to right traversal of the tree gives the list $COUNT_c$. Precisely, each node u in the tree contains the following fields.

- A color bit (red or black), a pointer to parent, a pointer to the left child and a pointer to the right child.

- A segment of bits, with length $\log n$ to $2 \log n$.

- An integer *size* indicating the total number of bits contained in the subtree rooted at u.
- An integer *sum* indicating the total number of `1`'s contained in the subtree rooted at u.

To support the function $\mathit{count}(c, i)$, we use the *size* value in each node to traverse the tree of $COUNT_c$ and locate the node u that contains the bit $COUNT_c[i]$. We record the number of `1`'s in the segment of u up to this bit,¶ and also the *sum* in the left child of u. Then, we traverse from u to the root. For every left parent v on the path, we record the number of `1`'s in the segment of v and also the *sum* in the left child of v. We can see that summing up all these recorded values gives the number of `1`'s in the list of $COUNT_c$ up to the bit $COUNT_c[i-1]$, which equals $\mathit{count}(c, i)$. The whole process takes $O(\log n)$ time.

To update the $COUNT$ data structure when a character c is inserted to position i of W, we insert a bit `1` to position i of $COUNT_c$ and insert a bit `0` to position i of $COUNT_{c'}$ for each $c' \neq c$. The time required is $O(\log n)$. Deletion of a character from W can be done in the opposite way in $O(\log n)$ time.

For the space requirement, we note that each node takes $O(\log n)$ bits and there are $O(n/\log n)$ nodes. Thus, the space requirement is $O(n)$ bits.

6.2.2 Implementation of *MARK*

Recall that $MARK$ is a set of at most $n/\log n$ tuples, each in the format of $(i, (j, k))$. Note that no two tuples have the same i-value, but there may be more than one tuple having the same j-value.

To support efficient update, we maintain two red-black trees, one for the i-values and the other for the (j, k)-values as follows.

For all the i-values of the tuples, they are stored in a red-black tree R_i, such that the left to right traversal of the tree gives the i-values of the tuples in sorted order.

For all the j-values of the tuples (allowing duplication), they are stored in a red-black tree, denoted as R_j, such that the left to right traversal of the tree gives the j-values of all the tuples in sorted order.

Let $i(u)$ be the i value stored in the node u in R_i. Let $j(v)$ be the j value stored in the node v in R_j. To represent the tuples, we store a pointer at each

¶This can be done in constant time in RAM with a universal decoding table of $o(n)$ bits [42].

node u in R_i, pointing to the node v in R_j if $i(u)$ and $j(v)$ belongs to the same tuple. Furthermore, the k-value of the corresponding tuple $(i(u),(j(v),k))$ is stored in the node v in R_j.

More precisely, each node in R_i has the following fields.

- A color bit (red or black), a pointer to the left child and a pointer to the right child.
- An integer $\mathit{diff}(u) = i(u) - i(lp(u))$, where $lp(u)$ denotes the left parent of u. $i(lp(u)) = 0$ if $lp(u)$ does not exist.
- A pointer to a node v in R_j.

Each node in R_j has the following fields.

- A color bit (red or black), a pointer to the left child and a pointer to the right child.
- An integer $\mathit{diff}(v) = j(v) - j(lp(v))$, where $lp(v)$ denotes the left parent of u. $j(lp(v)) = 0$ if $lp(v)$ does not exist.
- A pointer to a node u in R_i.
- An integer k.

Although we do not store the value $i(u)$ explicitly for every node u in R_i, its value can be recovered when we traverse down the tree R_i starting from the root. The idea is that, when we traverse down the tree, for every node x we meet on the path, we can compute the values $lp(x)$ and $i(x)$ in constant time along the way as follows. Let x' be the parent of x and assume inductively that $lp(x')$ and $i(lp(x'))$ are known. If x is the left child of x', $lp(x) = lp(x')$. Else, $lp(x) = x'$. In both cases, $i(x) = i(lp(x)) + \mathit{diff}(x)$.

Note that R_i and R_j are very similar to a red-black tree and they inherit the advantages of a balanced binary search tree. Searching, inserting and deleting a tuple can be done easily in $O(\log n)$ time. For any integer ℓ, let $X_\ell = \{u \mid u \in R_i \text{ and } i(u) \geq k\}$. To support the function *Shift_up*(ℓ), we need to increment $i(u)$ by one for each u in X_ℓ. Recall that the actual value of $i(u)$ is not stored in the node u of R_i. Instead, we store $\mathit{diff}(u) = i(u) - i(lp(u))$. Thus, if the value $i(lp(u))$ is increased by one, the value $i(u)$ is also increased by one automatically. Precisely speaking, to increment $i(u)$ by one for all node u in X_ℓ, we search R_i

for the node x with smallest $i(x)$ such that $i(x) \geq \ell$. Then, for any node w not equal to x but on the path from from root to x, we increment the value *diff*(w) by one if w is a right parent of some other node on the path. We also increment *diff*(x) by one. It is easy to see that $i(u)$ is incremented by one for each node u in X_ℓ. Thus, operation *Shift_up*(ℓ) can be done in $O(\log n)$ time.

The other operations, *Shift_down*, *Increment_lexico* and *Decrement_lexico* are supported similarly in $O(\log n)$ time. This completes the discussion for *MARK*.

6.2.3 Implementation of *PSI*

The *PSI* data structure has been used in the space-efficient CSA construction algorithm by Lam *et al.* [51]. In their paper, *PSI* was used as a dynamic representation of Ψ, allowing the CSA of a text $T[0, t-1]$ to be constructed incrementally in t phases where phase i modifies the Ψ of $T[t-i, t-1]$ slightly to become Ψ of $T[t-i-1, t-1]$.

The main idea is to maintain Ψ as $|\Sigma|$ increasing sequences, one for each c in Σ. Each sequence, say $v_1, v_2, ..., v_k$, is represented by a sequence of difference values $v_i - v_{i-1}$, encoded by γ code [20] to save space. The γ-coded sequence is partitioned into chunks of $O(\log n)$ bits, which are stored as nodes in a red-black tree (in a way similar to R_i in Section 6.2.2). The height of the red-black tree is $O(\log n)$. With the aid of an $o(n)$-bit decoding table, reporting a Ψ value, or updating with *Insert* or *Delete* can be done in $O(\log n)$ time in the RAM. For *Shift_up* or *Shift_down*, they can also be performed in $O(\log n)$ time due to the 'differential' nature of how the sequence is stored. The total space is $O(n \log |\Sigma|)$ bits, which is $O(n)$ bits for constant-size alphabet.

Chapter 7

Managing Dynamic Dictionary and Dynamic Text

In Chapter 6, we have discussed the problem of managing a dynamic library. In this chapter, we first study a dual problem which deals with indexing a collection patterns, so as to answer efficiently the occurrences of all patterns in any given text T. This query is referred as the *dictionary matching* query.

Our solution for managing the dynamic dictionary is based on its static counter-part, which can be 'dynamized' using a technique applied in [25]. As a side result, we also give an alternative index for managing a dynamic library. The difference with the index in Chapter 6 is that, the library matching time becomes faster, but the time to handle updates of the library becomes amortized, and could be much slower in the worst case.

Afterwards, we study the problem of managing a dynamic text, where we want to build an index for a text T to support efficient pattern matching, but T itself is subjected to insertion or deletion of substring at arbitrary position. In Section 7.2, we describe a technique called *interval partitioning*, which has been used previously to index a dynamic text in $\Theta(n \log n)$ bits. In Section 7.3, we propose a compressed solution for the dynamic text, by combining the interval partitioning technique with our dynamic dictionary and dynamic library.

7.1 Managing a Dynamic Dictionary

In this section, we show how to apply the CST and the CSA to manage a static dictionary and a static library, respectively. Then, we apply the technique used in [25] to turn our indexes into those for the dynamic cases.

We start by demonstrating the power of CST for answering the dictionary matching query for a *static* set of patterns $\{S_1, S_2, \ldots, S_k\}$ of total length n, such that no S_i is a proper substring of the other.* This is done by adapting the algorithm of Chang and Lawler [14] as follows.

Lemma 7.1 *Let* $S = S_1\$S_2\$\ldots S_k\$$, *where* \$ *is a special character not appearing in any* S_i. *Suppose that no* S_i *is a proper substring of the other. Given the CST of* S, *denoted by* $CST(S)$, *we can find all occurrences of each of the* S_i*'s in a text* $T[0, |T|-1]$ *in* $O(|T|(\log^\epsilon n + \log|\Sigma|) + occ\log^\epsilon n)$ *time, where* $0 < \epsilon \leq 1$.

Proof: To report the occurrences, we traverse $CST(S)$ by matching characters of T, starting from the first character. This is similar to finding an occurrence of T in S using a suffix tree; but whenever we encounter a mismatch, or the previous step revealed an occurrence of S_i in T, we restrict the search by omitting the first character of the substring of T that is currently matched. For instance, suppose that we are at the position of the tree that matches the substring $T[q, r]$ (i.e., the position reached by matching $T[q, r]$ starting from the root of $CST(S)$). Then, we locate the position of the tree that matches the substring $T[q+1, r]$. This can be done by following a suffix link (which is a tree navigation operation) and then moving downwards a couple of nodes in the tree. Afterwards, we continue matching the next character $T[r+1]$ there.

Note that each S_i corresponds to a distinct leaf edge in $CST(S)$. At the $(r+1)$-th step where we match $T[r]$, we can check whether there is an occurrence of some S_i in T ending at $T[r]$, by examining whether matching an extra \$ would bring us to a leaf edge of some S_i.

Assuming that no S_i is a proper substring of the other, it is easy to verify that the above algorithm locates all occurrences of each S_i in T. For the running time, it follows since we can show that the number of tree navigation operations required (based on the arguments in [14]) is $O(|T|)$, and each operation requires at most $O(\log^\epsilon|S| + \log|\Sigma|)$ time by Lemma 5.6. This completes the proof of the lemma. □

On the other hand, CSA can be directly applied to answer the library matching query as follows.

*As to be shown in the later section, the reduction from the compressed dynamic indexing problem involves a collection of strings satisfying this restriction.

Lemma 7.2 *Let $S = S_1\$S_2\$\ldots S_k\$$. Given the CSA of S, denoted by $CSA(S)$, we can find all occurrences of a pattern P in each of the S_i's in $O(|P|\log\log|\Sigma| + occ\log^{\epsilon} n)$ time.*

Proof: We use the definition in Chapter 4 for the CSA of S, which occupies $O(n\log|\Sigma|)$ and supports each backward search step in $O(\log\log|\Sigma|)$ time. Also, the CSA is augmented with an auxiliary data structure of $O(n\log|\Sigma|)$ bits to support retrieval of an SA entry in $O(\log^{\epsilon} n)$ time. Then, by performing a backward search of P with $|P|$ backward search step by Lemma 2.4, each occurrence of P can be reported in $O(\log^{\epsilon} n)$ time. The lemma thus follows. $\square$

Next, we describe how to manage a dynamic dictionary and a dynamic library based on our results for the static cases. The idea follows closely to how Ferragina and Manzini 'dynamize' the FM-index, which is shown in [25]. Basically, we divide the texts in the collection into $O(\log n)$ groups. Groups are labeled by consecutive integers starting from 1, so that group i is either empty, or contains texts with total length in the range $[2^i, 2^{i+1})$. Note that each group is in fact a (sub-)collection of texts, and therefore we can support dictionary matching and library matching queries in each group using Lemmas 7.1 and 7.2. In other words, the total time for both queries will be increased by a factor of $O(\log n)$ in order to search all groups.

To insert a text X into the collection, we let $x = \lceil\log|X|\rceil$ and insert X into group x. However, there can be overflow during the insertion, since we require that the total length of texts in group x is less than 2^{x+1}. In this case, we merge group x and group $x+1$ (and possibly more groups) to form a new group, and construct a new CSA and CST accordingly. In the worst case, an insertion of a text can require a construction of CSA and CST for texts of total length $O(n)$; however, such a worst case would not be frequent, by considering that a particular text can be merged with the others at most $O(\log n)$ times. Thus, the amortized insertion time can be bounded by $O(\log n \cdot (|X|\log^{1+\epsilon} n))$.

For deletion, we apply a lazy update scheme so that the CST and the CSA of a group is re-constructed only when a fraction of $O(1/\log n)$ of the total length is marked deleted. To do so, we keep a record on which texts are deleted, and a balanced search tree of height $O(\log n)$ for marking the intervals in the CSA and CST of the deleted texts. Then, deletion of S_i from the collection can be done in $O(\log n \cdot (|S_i|\log^{1+\epsilon} n))$ amortized time, but the time of the the dictionary or library matching queries is further increased by a factor of $O(\log n)$.

In summary, we have the following theorem.

Theorem 7.3 *Consider all texts over an alphabet Σ. Let $\Delta = \{S_1, S_2, \ldots, S_k\}$ be a collection of strings with total length n. For any fixed ϵ with $0 < \epsilon \leq 1$, we can maintain a data structure CDL(S) of size $O(n \log |\Sigma|)$ bits that supports*

- *finding all occurrences of S_i in a text T for all i in $O(|T| \log^2 n(\log^\epsilon n + \log |\Sigma|) + occ \log^{2+\epsilon} n)$ if no S_i is a proper substring of the other, where occ denotes the total number of occurrences of all the S_i's;*
- *finding all occurrences of a pattern P in the texts in Δ in $O(|P| \log^2 n \log\log |\Sigma| + occ' \log^{2+\epsilon} n)$ time, where occ' denotes the total number of occurrences of P;*
- *inserting a text X into Δ in $O(|X| \log^{2+\epsilon} n)$ amortized time; and,*
- *deleting a text S_i from Δ in $O(|S_i| \log^{2+\epsilon} n)$ amortized time.*

The data structure in Theorem 7.3 can also be used to solve the *prefix-suffix* matching problem, which finds the longest suffix of a pattern P which is a prefix of any S_i in Δ. Basically, this is done by performing a backward search of P for s_{max} steps, where $s_{max} = \max |S_i|$. Then, we can partition the SA into $O(s_{max})$ ranges, such that for a particular range, the longest suffix of P matching the prefix of any suffix of in this range has the same length. Afterwards, by checking those SA entries that correspond to the whole string of S_i's, we obtain the desired lengths. The result is stated in the following lemma.

Lemma 7.4 *The data structure in Theorem 7.3 supports finding the longest suffix of a pattern P which is a prefix of S_i for each i in $O(s_{max} \log^2 n \log\log |\Sigma| + k \log^{2+\epsilon} n)$, where s_{max} denotes the length of the longest text in $\{S_1, S_2, \ldots, S_k\}$.*

7.2 Interval Partitioning

After studying the dictionary and library problem, we are ready to discuss the indexing of a dynamic text to allow substring insertion and deletion at arbitrary positions. In this section we review a basic technique called interval partitioning, which was commonly used in previous work on indexing a dynamic text with $\Theta(n \log n)$ bits (e.g., [23], [24]). The idea is that, given a text T, we divide it into "short", possibly overlapping *substrings*, so that a change in T at some position

p is limited to changes in a few intervals around p. The drawback is that pattern searching would become a more involved operation.

The following interval partition scheme is proposed in [24]. Let ℓ be an integer (which will be chosen as $\sqrt{n}$). Suppose that T is partitioned into $k = \Theta(n/\ell)$ intervals $I_1, I_2, \ldots, I_k$ with each I_i is of length between ℓ to 2ℓ. Let $c_1 > c_2$ be some suitable constants. Define two boundary intervals I_0 and I_{k+1} containing $c_1\ell$ \$'s. To allow efficient searching, each interval I_i is to be represented by five "short" strings, which are of length $O(\ell)$:

- X_i to be the string $I_i \cdots I_{i+r}$, where r is the smallest integer such that $|I_i| + \cdots + |I_{i+r}| \geq c_1\ell$;
- C_i and C'_i to be I_i's prefix and suffix of length ℓ, respectively;
- L_i to be the suffix of $I_0 I_i \cdots I_i$ of length $c_2\ell$ and R_i to be the prefix of $I_i \cdots I_k I_{k+1}$ of length $c_2\ell$.

In other words, T is represented by several collections of short strings (namely, X_i, C_i and C'_i, L_i, and R_i). Notice that for a pattern P of length at most $(c_1-2)\ell$, if it occurs in T, it must occur in some X_i. Thus, we only need to search the collection of X_i's.

Lemma 7.5 *Let* `Query-X(`P`)` *denote the query of finding all occurrences of a pattern P in each $X \in \{X_1, \cdots, X_k\}$. Suppose that* `Query-X(`$P$`)` *can be answered in t_X time. Then, if $|P| \leq (c_1 - 2)\ell$, all occurrences of P in T can be found in $O(t_X)$ time.*

Searching for longer patterns is non-trivial, yet Ferragina and Grossi [24] were able to devise an efficient algorithm based on the following batch queries to the four collections. The time complexity is stated in the following lemma.

- `Query-C(`P`)`: For each $X \in \{C_1, \cdots, C_k, C'_1, \cdots, C'_k\}$), find an occurrence of P.
- `Query-L(`P`)`: For each $X \in \{L_1, \cdots, L_k\}$, find P's *longest prefix* that is a suffix of X.
- `Query-R(`P`)`: For each $X \in \{R_1, \cdots, R_k\}$, find P's *longest suffix* that is a prefix of X.

Lemma 7.6 [24] *Suppose that* `Query-C(`P`)`*,* `Query-L(`P`)` *and* `Query-R(`P`)` *can be answered in* t_C*,* t_L *and* t_R *time, respectively. Then, if* $|P| > (c_1 - 2)\ell$*, all occurrences of* P *in* T *can be found in* $O(t_C + t_L + t_R + occ)$ *time, where occ denotes the number of occurrences of* P *in* T*.*

If $O(n \log n)$-bit space is given, one can use generalized suffix trees to represent the X_i's, C_i's, L_i's, and R_i's. Then `Query-X` can then be answered in $O(|P|+occ)$, where *occ* denotes the total number of occurrences of P in each X_i's, and the other three queries can be supported in $O(|P| + k + \ell)$ time.

In the next section, we propose a compressed data structure for storing the short strings in $O(n \log |\Sigma|)$ bits, and show how to support the queries efficiently. Then based on Lemma 7.5 and 7.6, any pattern can be searched efficiently.

7.3 Managing a Dynamic Text

In this section, we describe the compressed indexing data structure for the *dynamic text* problem using $O(n \log |\Sigma|)$ bits of space.

Recall from the previous section that a text T is represented by several collections of short strings (namely, X_i's, C_i's and C_i''s, L_i's and R_i's), each of length $O(\ell)$. We apply the compressed index discussed in Section 7.1 to represent each collection and support the update of strings in the collection, dictionary matching and library matching. Denote as CDL_X the resulting index $CDL(\{X_1, X_2, \ldots, X_k\})$, and similarly CDL_C, CDL_R, and CDL_L for respectively the indexes $CDL(\{C_1^l, \ldots, C_k^l, C_1^r, \ldots, C_k^r\})$, $CDL(\{R_1, \ldots, R_k\})$ and $CDL(\{L_1', \ldots, L_k'\})$, where L_i' denotes the reverse of L_i.

Note that all C_i's and C_i''s have equal length, so that none of them can be a proper substring of the other. This is also true for L_i's and R_i's. Thus, the time complexity stated in Theorem 7.3 for the dictionary matching query holds for CDL_C, CDL_R, and CDL_L. Also, recall that finding all occurrences of a pattern P in T can be reduced to answering some batch queries `Query-X`, `Query-C`, `Query-L` and `Query-R`. It is easy to see that these queries are readily supported by CDL_X, CDL_C, CDL_L and CDL_R based on Theorem 7.3 and Lemma 7.4. Precisely, `Query-X` corresponds to a library management query on CDL_X, `Query-C` corresponds to a dictionary matching query on CDL_C, while `Query-L` and `Query-R` correspond to prefix-suffix queries on CDL_L and CDL_R, respectively. The time complexity is stated in the following lemmas.

Lemma 7.7 *Given CDL_X,* `Query-X(`P`)` *can be done in $O(|P|\log^2 n \log\log|\Sigma| + occ_X \log^{2+\epsilon} n)$ time, where occ_X denotes the total number of occurrence of P in all X_i's.*

Lemma 7.8 *Given CDL_C,* `Query-C(`P`)` *can be answered in $O(|P|\log^2 n(\log^\epsilon n + \log|\Sigma|) + k\log^{2+\epsilon} n)$ time.*

Proof: It follows from a minor modification to Lemma 7.1 which outputs only one occurrence for each of the S_i's in P. □

Lemma 7.9 *Given CDL_L and CDL_R,* `Query-L(`P`)` *and* `Query-R(`P`)` *can both be answered in $O(\ell\log^2 n \log\log|\Sigma| + k\log^{2+\epsilon} n)$ time.*

By combining the above three lemmas with Lemma 7.5 and Lemma 7.6, we get the following result.

Corollary 7.10 *Let occ be the number of occurrences of P in T.*[†]

1. *If $|P| \le (c_1 - 2)\ell$, all occurrences of P in T can be found in $O(|P|\log^2 n \log\log|\Sigma| + occ\log^{2+\epsilon} n)$ time;*
2. *Otherwise, all occurrences can be found in $O(|P|\log^2 n(\log^\epsilon n + \log|\Sigma|) + (k + occ)\log^{2+\epsilon} n + \ell\log^2 n\log\log|\Sigma|)$ time.*

Now, let us consider how to update the above *CDL*'s. Our method is based on the one used by [24] (where they are updating the generalized suffix trees instead). Firstly, suppose an arbitrary substring Y of length y is deleted from T. we observe that only those intervals $I_i, I_{i+1}, \ldots, I_j$ that overlap with Y is affected. (Note that the total length of these overlap intervals is at most $y + 4\ell$.) Consequently, we can update each of the *CDL*'s by removing those strings that depend on the overlap intervals from the corresponding collection. This in turn corresponds to the deletion of texts in CDL_X, CDL_C, CDL_L, and CDL_R of total length at most $c_1(y + 4\ell)$, $2(y + 4\ell)$, $c_2(y + 4\ell)$ and $c_2(y + 4\ell)$, respectively. To complete the discussion, note that there may be characters remaining in I_i and I_j after the deletion of Y. These characters can be merged together with I_{i-1} to form a new interval in T (and splitting may be required if the resulting length exceeds 2ℓ). Afterwards, we create new strings that depend on I_{i-1} and insert

[†]Note that $occ_X \le (c_1 - 2)occ$, where occ_X is the number of occurrences of P in all X_i's.

them into the corresponding *CDL*'s. Together, this corresponds to deletions and insertions of texts of total length $O(\ell)$ in each of the *CDL*'s.

On the other hand, when a text Y of length y is inserted into T, where the insertion point is either in the middle of some interval I_i, or between some interval I_i and I_{i+1}, we see that those strings in the *CDL*'s that depend on I_i (and I_{i+1} as well for the latter case) are needed to be removed. This corresponds to the deletions of texts of total length $O(\ell)$ in each of the *CDL*'s. Afterwards, we create intervals in T to accommodate Y. The total length of these intervals is at most $y + 2\ell$ (in case Y is inserted in the middle of I_i). This in turn corresponds to insertions of texts of total length $O(y + \ell)$ in each of the *CDL*'s.

In summary, an insertion or deletion of an arbitrary text of length y in T would require insertions and deletions of texts of total length at most $O(y + \ell)$ in CDL_X, CDL_C, CDL_L, and CDL_R. Thus, we have the following lemma.

Lemma 7.11 *We can update the data structures CDL_X, CDL_C, CDL_L and CDL_R in $O((y + \ell)\log^{2+\epsilon} n)$ amortized time, when an arbitrary text of length y is inserted or deleted in T.*

By setting $\ell = \sqrt{n}$, we have $k = O(\sqrt{n})$. This immediately leads to the following theorem.

Theorem 7.12 *For any fixed ϵ with $0 < \epsilon \leq 1$, we can maintain a compressed index for $T[0, n-1]$ over an alphabet Σ in $O(n \log |\Sigma|)$ bits that supports*

- *searching a pattern P in T, using $O(|P| \log^2 n(\log^\epsilon n + \log |\Sigma|) + occ \log^{2+\epsilon} n)$ time; and,*
- *insertion/deletion of an arbitrary text Y, using $O((|Y| + \sqrt{n}) \log^{2+\epsilon} n)$ amortized time.*

Proof: The insertion and deletion time follows from Lemma 7.11. If $|P| \leq (c_1 - 2)\sqrt{n}$, the searching time follows, as it is $O(|P| \log^2 n \log\log |\Sigma| + occ \log^{2+\epsilon} n)$ by Corollary 7.10(1). Otherwise, we have $k = O(|P|)$, and the searching time follows from Corollary 7.10(2). $\square$

Chapter 8

Experimental Studies of CSA and FM-index for DNA Sequences

With more and more DNA sequences being decoded, searching DNA sequences has become a frequent and mundane part of most biological research nowadays. This has posted a real challenge in traditional indexes, for the length of DNA sequences can be very long which dictates the use of huge amount of computer memory. Though the above problem can usually be solved by using a super-computer, this expensive solution is seldom a convenient choice for a biology laboratory of moderate scale.

The invention of CSA and FM-index advanced a big step towards indexing with an ordinary PC. For DNA sequence, which is basically a text over a four-letter alphabet $\{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$, the bound for the basic Ψ function of the CSA requires only $5n$ bits in theory, while for FM-index, it occupies $3n$ bits in practice [26]. These figures immediately imply that we can store the CSA and FM-index of DNA of length up to a few Gigabases, which covers almost every known DNA sequences nowadays.

However, if CSA and FM-index are to be workable in practice, we have to consider yet another important issue: the memory requirement for their construction. In Chapters 3 and 4, we have shown that CSA can be directly constructed from the DNA sequence in $O(n)$ bits space, while FM-index can be converted from the CSA in negligible space. We would like to determine the maximum length of a genome whose CSA and FM-index can be constructed, and the time required for constructing these indexes.

On the other hand, the searching performance of CSA and FM-index is asymptotically of the same order as (or even better than) the traditional suffix arrays, despite the compactness of their sizes. Given this promising theoretical bound, it is natural to ask how fast CSA and FM-index can search in practice? Moreover, which one is better?

For FM-index, previous experiments [26] have shown that its performance is comparable to suffix arrays, when searching a pattern whose length is short (8-15 bases). However, for searching long patterns (say, a few hundred or even a few thousand bases), it is not known whether the result will be consistent.

In addition, as we may recall, there are two types of searching methodology for CSA or FM-index. The first one is the *backward search* mentioned in Chapter 2, which is the method tailored for CSA or FM-index. The second one is to apply the traditional pattern searching algorithm with suffix arrays, while using CSA or FM-index for the retrieval of suffix array entries when needed. Theoretically speaking, backward search is faster than forward search. Table 8.1 gives a summary of the worst-case searching performance of CSA and FM-index. Note that in the table, the bounds depend on a parameter ϵ, where $0 < \epsilon \leq 1$. This ϵ is in fact a tradeoff parameter for the searching time and index space. In order to achieve the stated timing bounds, the corresponding CSA or FM-index would require $\frac{1}{\epsilon}$ times the size of the basic structure. In this paper, we consider ϵ to be 1.[*]

Although the worst-case behavior of backward search (in both indexes) is better than that of the forward search, in practice, however, forward search may surpass backward search for two reasons: Firstly, based on Manber and Myers [56], the average time of forward search is expected to be $O(|P| \log n + occ \log n)$ and $O(|P| + occ \log n)$ for CSA and FM-index, respectively, which matches to the worst-case time of backward search. Secondly, each operation of the backward search is usually more computationally involved, so that the hidden constant in the worst-case bound could be very high. Thus, it is interesting to find out in practice, which searching methodology prevails?

This chapter evaluates the performance of the CSA and FM-index when they are used to index DNA sequences, and attempts to answer all the above questions. We ran all the experiments on a machine equipped with a 1.7 GHz Pentium IV processor with 256 Kbytes of L2 cache, and 4 Gbytes of RAM. The operating

[*]Readers may also note that we are referring a simpler implementation of CSA, so that pattern searching by backward search takes $O(|P| \log n)$ time, instead of the better result of $O(|P| \log \log |\Sigma|)$ time mentioned in Chapter 4.

Table 8.1: Searching performance of CSA and FM-index.

Index	Forward Search	Backward Search
CSA	$O(\lvert P\rvert \log^{1+\epsilon} n + occ \log^{\epsilon} n)$	$O(\lvert P\rvert \log n + occ \log^{\epsilon} n)$
CSA ($\epsilon = 1$)	$O(\lvert P\rvert \log^2 n + occ \log n)$	$O(\lvert P\rvert \log n + occ \log n)$
FM-index	$O(\lvert P\rvert \log^{\epsilon} n + occ \log^{\epsilon} n)$	$O(\lvert P\rvert + occ \log^{\epsilon} n)$
FM-index ($\epsilon = 1$)	$O(\lvert P\rvert \log n + occ \log n)$	$O(\lvert P\rvert + occ \log n)$

system was Solaris 9. Over this platform, we implement programs to construct and to test the CSA and the FM-index for three genomes, of size from different scales: E. coli (4.6 Mbases), Fly (98 Mbases) and Human (2.88 Gbases).[†] Our programs are implemented in `C`, which is compiled by `gcc` with the option `-O3`.

The remaining of the chapter is organized as follows. In Section 7.1, we explain our implementation of CSA and FM-index, where we have made slight changes over the original definitions. In Section 7.2, we report the time and space requirements for our implementation of the construction algorithm. Finally, Section 7.3 presents our preliminary findings concerning the searching performance of CSA and FM-index,

8.1 Implementation Details

Our encoding for the Ψ function of CSA in this chapter is based on Lam *et al.* [51], which is different from those discussed in the previous chapters.[‡] Basically, our encoding makes use of Corollary 2.2 and store Ψ as four increasing sequences, with values between 0 and n, as follows: for each sequence $s_1, s_2, \ldots$, we store the difference values (that is, $s_1, s_2 - s_1, s_3 - s_2$, and so on) instead of the original values. Compression is achieved by encoding each of these difference values separately using variable-length prefix-free codes, such as γ code or δ code [20].

The original paper suggests to use δ code, which has better worst-case size bound for general alphabets; but for our case, we use γ code instead, for it achieves smaller size than δ code when encoding the Ψ function of DNA sequence

[†]The genomes are obtained from the Genome Bioinformatics Group of University of California, Santa Cruz [37], which was the latest version by the time our experiments are conducted.

[‡]We have implemented and tested three other variations of encodings. The performance for searching DNA sequence is quite similar; the variation reported in this chapter is consistently the best in our studies.

in practice.

Note that the use of variable-length code has a disadvantage: to decode any region of the sequence, we must start the decoding process from the beginning of the encoded sequence. In order to speed up the decoding process, we adopt a simple scheme by storing the value of $\Psi[i]$ and the starting position of its encoding explicitly whenever $i \bmod \ell = 0$ for some $\ell = O(n/\log n)$. Then, to compute $\Psi[k]$, we first find $\Psi[c\ell]$ with $c\ell \leq k < (c+1)\ell$ in constant time. Afterwards, we look at the encoded sequence for $\Psi[c\ell+1] - \Psi[c\ell], \Psi[c\ell+2] - \Psi[c\ell+1], \ldots, \Psi[k] - \Psi[k-1]$ to compute $\Psi[k] - \Psi[c\ell]$. The latter part can be done by $O(\log n)$ table-lookups in the worst-case; but on average, the number of table-lookups is constant. Thus, the time to compute Ψ takes $O(1)$ time on average. The space requirement for this auxiliary data structure is $O(n)$ bits. Note that the more values of Ψ we store explicitly, the more space we need, but the faster the decoding process.

To support retrieval of the suffix array entries, we need to store another $O(n)$-bit auxiliary data structure. Precisely, we store SA$[i]$ explicitly whenever $i \bmod t = 0$ for some $t = O(n/\log n)$. To compute SA$[i]$, we compute $\Psi[i], \Psi^2[i], \Psi^3[i]$ and so on, until $\Psi^k[i] \bmod t = 0$. Then, we obtain the value $s = \text{SA}[\Psi^k[i]]$, and it is easy to verify that $\text{SA}[i] = s - k$. However, this auxiliary data structure does not guarantee the number of Ψ calls for retrieving the SA values, where in the worst case, $O(n)$ calls are required. Despite of this, it has a good average case performance, where an SA value can be retrieved in expected $O(\log n)$ Ψ calls. This equals to an average $O(\log n)$ time in practice. In theory, we can guarantee the worst number of Ψ calls to be $O(\log n)$ by sacrificing some space for constant-time `rank` and `select` data structures of [42]; however, the hidden constant accompanied in the $O(\log n)$ term is high in practice, which means more space for the index *and* more time for getting the SA.

For the same reason, our implementation of FM-index adopts an auxiliary data structure that supports the retrieval of SA in average $O(\log n)$ time instead of worst-case. For the other parts, it follows closely to that in [25], which includes the Burrows-Wheeler transform [13], move-to-front encoding [9] and run-length encoding for the compression. (See the original paper for more details.)

8.2 Construction Requirements

In Chapters 3 and 4, we have shown that CSA can be directly constructed from the DNA sequence. The space requirement is $(5 + r)n$ bits space, for any $r > 0$. Afterwards, FM-index can be converted from the CSA in negligible extra space. Note that the larger the r, the faster the algorithm is.

For the actual construction, we apply the algorithm of Lam *et al.* [51], which is the precursor of our construction algorithm in Chapter 3. Based on this, we have implemented space-efficient programs for constructing CSA and FM-index on a PC, and tested it with E. coli, Fly and Human as the input genomes.

We set $r = 5$ to make the maximal use of the main memory. Table 8.2 shows the corresponding construction space and time for each genome.

Table 8.2: Construction time and space for CSA and FM-index.

DNA	Construction Space	Construction Time	
		CSA	FM-index
E. coli	5.8M byte (10n bits)	60 sec	72 sec
Fly	125M byte (10n bits)	30 min	36 min
Human	3.6G byte (10n bits)	24 hour	28 hour

From the indexes constructed, we found that if we discount the space for all the auxiliary data structures, the size of CSA is close to $4n$ bits, while the size of FM-index is close to $3n$ bits. To conclude this section, Table 8.3 shows the limitations on the index that can be constructed and resided in an ordinary PC nowadays, assuming a RAM of size 4 Gbytes.[§]

8.3 Searching a Pattern with CSA and FM-index

This section investigates the practical searching behavior of CSA and FM-index. In the first part, we compare the two searching methodologies for CSA and FM-index, where our experimental results show that forward search in practice performs better than backward search when the pattern length is long. In addition, other interesting findings are also observed. In the second part, we perform a

[§]Technically speaking, we cannot make full use of all the RAM as some space must be reserved for the operating system. In our case, only a maximum of 3.6 Gbytes out of 4 Gbytes are available.

Table 8.3: Limitations on various indexes, assuming a RAM of 4 Gbytes.

Index	Construction Algorithm	Maximum Genome Constructed	Maximum Genome Able To Reside
Suffix Tree	Kurtz [50]	180 Mbases	180 Mbases
Suffix Array	Larsson-Sadakane [52]	450 Mbases	900 Mbases
CSA	Lam *et al.* [51]	5000 Mbases	7200 Mbases
FM-index	this thesis	5000 Mbases	9600 Mbases

case study to compare the performance of CSA and FM-index with the suffix tree and suffix arrays.

8.3.1 Forward Search and Backward Search

We have constructed the CSA and the FM-index for the following genomes: E. coli (4.6 Mbases), Fly (98 Mbases), Human Genome (2.88 Gbases). As mentioned, the amount of auxiliary information affects the performance of the compressed index greatly. For our experiments, we have investigated three implementations of each index, which respectively requires a total of $4.5n$, $6n$ and $8n$ bits of space, which are refer to as the small, medium and large implementations.¶ In case where we conduct forward search, an additional $2n$ bits of memory is used for storing the DNA text.

For each genome, we have tested the searching times using patterns of lengths 10, 50, 100, 500, 1000, 5000, and 10000, where patterns are selected from the corresponding genome at random positions, so as to get a more accurate account of the worst-case behavior.‖ For each test case, it is repeated for 1000 times to obtain an average timing. Finally, only searching times for existence (that is, to determine whether the pattern P occurs in the DNA sequence T) are reported, as the time for enumeration (that is, reporting the occurrence of P in T) is independent of the searching methodology. From our experiments, we observe

¶We have also investigated another setting in which both data structures have the same amount of auxiliary storage (precisely, $2n$ bits). Note that this setting is not fair to FM-index as we allow CSA together with its auxiliary storage to use more memory; nevertheless, the finding is similar to those to be reported below.

‖As DNA is a very biased string, a random pattern is unlikely to be found there; thus, searching for a random string is often very fast as a few comparisons could confirm the non-existence. To test the worst behavior of the indexing data structures, we use substrings or modified substrings of the DNA.

that if for a fixed type of index, if we are provided with the same amount of space (in terms of n), the searching performance *does not vary* a lot with respect to the length of the underlying genome. For instance, see Figures 8.1, 8.2 and 8.3 for the searching performance of CSA and FM-index for different genomes under the medium implementation, and observe the similarity across the three figures.

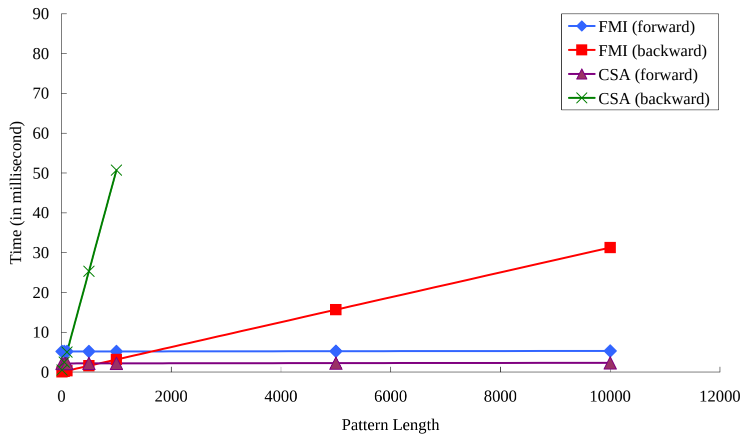


Figure 8.1: Searching performance of the meidum implementations of CSA and FM-index of E. coli.

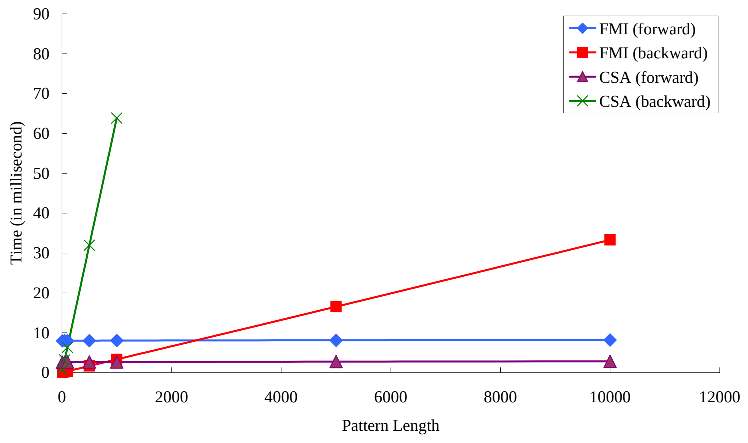


Figure 8.2: Searching performance of the meidum implementations of CSA and FM-index of Fly.

In this section, we shall focus on the experimental result of Human (Table 8.4), and use it to present the general observations that we have made.

Some interesting findings can be summarized as follows.

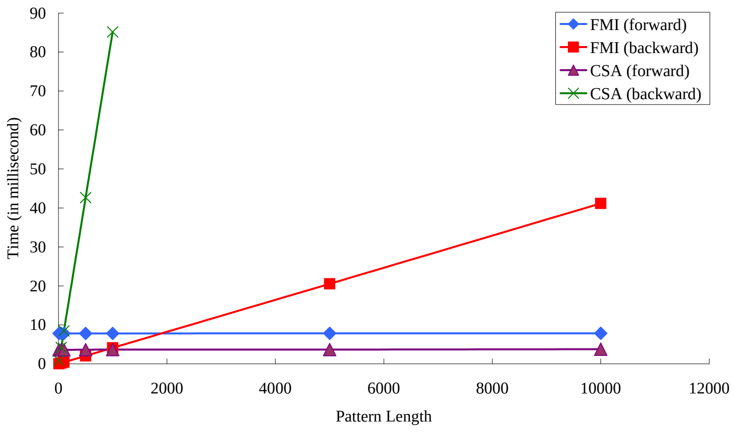


Figure 8.3: Searching performance of the meidum implementations of CSA and FM-index of Human.

- Using backward search, FM-index is at least ten times faster than CSA in all testing cases. Both CSA and FM-index have searching time increasing linearly on pattern length.

- Using forward search, CSA is, however, two times faster than FM-index in the medium or the large implementation, and slightly slower than FM-index in the small implementation.

  Unlike backward search, forward search is not sensitive to pattern length. We believe that in practice, forward searching with CSA and FM-index requires $\alpha|P| + \beta \log n$ time, for some constants $\alpha \ll \beta$. In other words, the time is determined by the $\log n$ factor instead of the pattern length.

- Theoretically, backward search is better than forward search for both indexes. Most surprisingly, experiments show that for long patterns, forward search is more efficient. For CSA, forward search outperforms backward search for patterns of length 200 or more; for FM-index, this occurs for patterns of length around 3000 or more.

  Roughly speaking, for patterns of length less than 1500, FM-index with backward search is the best; otherwise, CSA with forward search is fastest, while FM-index with forward search is comparable.

Table 8.4: Searching performance (in msec) of CSA and FM-index for Human.

Index	Index Size	Searching Method	Pattern Length						
			10	50	100	500	1000	5000	10000
CSA	small	forward	38.96	38.96	38.98	38.98	39.01	39.10	39.20
		backward	2.51	12.37	24.74	123.7	247.6	1235	2474
	medium	forward	3.603	3.604	3.607	3.618	3.656	3.658	3.757
		backward	0.852	4.200	8.443	42.66	85.15	425.2	851.2
	large	forward	1.359	1.359	1.360	1.362	1.370	1.454	1.545
		backward	0.584	2.910	5.802	29.17	58.45	291.5	583.2
FMI	small	forward	20.01	20.02	20.04	20.05	20.08	20.13	20.22
		backward	0.074	0.352	0.710	3.554	7.120	35.62	71.31
	medium	forward	7.805	7.805	7.806	7.808	7.811	7.819	7.829
		backward	0.044	0.208	0.412	2.051	4.108	20.55	41.15
	large	forward	2.638	2.639	2.640	2.644	2.672	2.735	2.821
		backward	0.032	0.155	0.312	1.536	3.084	15.37	30.72

8.3.2 Comparison with Suffix Trees and Suffix Arrays

As one can expect, the searching performance for suffix tree or suffix arrays should be better than CSA and FM-index, as there are no compression involved in the former indexes. In this section, we try to give a quantitative comparison between these four indexes in practice.

We have constructed the four indexes for the E. coli (4.6 Mbases) genome. For CSA and FM-index, we just consider the medium implementations, which each occupies $6n$ bits. We have tested the searching times using patterns of lengths 10, 50, 100, 500, 1000, 5000, and 10000. Forward search are conducted for all the indexes, and backward search are conducted for CSA and FM-index. In case where we conduct forward search, an additional $2n$ bits of memory is used for storing the DNA text.

Patterns are selected from the E. coli genome at random positions. For each test case, it is repeated for 1000 times to obtain an average timing. The searching times are separated into two parts: the time for reporting whether the pattern exists in the text, and the time for enumerating the location of each occurrence. Tables 8.5(a) and 8.5(b) show the best time obtained by each index.

From Table 8.5, we observe that both CSA and FM-index are much slower than suffix trees and suffix arrays for both existential query or enumerating occurrences. Nevertheless, in terms of absolute time, each existential query using

Table 8.5: Searching performance of different indexes. (a) Average time (in msec) for one existential query for different pattern length. (b) Average time (in μsec) for reporting the location of one occurrence

Index	Pattern Length						
	10	50	100	500	1000	5000	10000
Suffix Tree	0.003	0.003	0.004	0.006	0.010	0.035	0.067
Suffix Array	0.010	0.010	0.011	0.015	0.021	0.060	0.111
CSA	0.512	2.145	2.150	2.166	2.173	2.232	2.318
FM-index	0.035	0.162	0.320	1.573	3.152	5.225	5.309

(a)

Index	Time
Suffix Tree	11.5
Suffix Array	0.5
CSA	49.0
FM-index	114.0

(b)

CSA or FM-index can be answered within a few milliseconds, and the enumeration of one occurrence is in the order of *microseconds*, which is acceptable in most applications.

Chapter 9

Concluding Remarks and Further Work

With the advance in bio-technology and information technology, the needs for indexing very long texts while supporting efficient query are becoming more and more important. The most promising solution lies in the compressed text indexes. In this thesis, we have studied the construction and the application, and the design of some of these indexes. Our results, and possibly some directions for further study, are summarized as follows.

1. In Chapter 3, we have given an $O(n \log n)$-time algorithm for constructing CSA in the optimal space of $O(nH_0 + n)$ bits. Then, in Chapter 4, an $O(n \log \log |\Sigma|)$-time algorithm for constructing CSA and FM-index in $O(n \log |\Sigma|)$ bits of working space is described. One would naturally ask: Is there a construction algorithm for CSA and/or FM-index that requires optimal space, and runs in $o(n \log n)$ time?

2. In Chapter 5, we have shown the conversion algorithm from CSA to CST. In addition, we have shown how to apply CSA or CST to solve the *maximal unique matches* problem.

3. In Chapter 6 and Chapter 7, we have proposed the first compressed indexes of $O(n \log |\Sigma|)$-bits for managing a dynamic library, a dynamic dictionary, and a dynamic texts.

   Many grounds are still open for further research.

   - Though our solution for the dynamic library supports query and update times within a polylog(n) factor from the best results achieved by the uncompressed index in the literature, it may be possible to further improve it to be within $O(\log n)$ factor.

- In our solution for the dynamic dictionary, we are in fact assuming in the text collection we manage, no one is a substring of the other. Is it possible to remove this requirement?
- In our solution for the dynamic dictionary and dynamic texts, only query time admits a worst-case performance. Can we improve the update time from amortized-case to worst-case? Also, can both the time be improved to be closer to from the best results achieved by uncompressed index in the literature?

4. In Chapter 8, we have given preliminary experimental studies for the CSA and FM-index. For further work, there are many other compressed indexes, such as the Compressed Compact Suffix Arrays (CCSA) [55] of Mäkinen and Navarro, and other versions of the CSA and FM-index, should be included for a thorough comparison. Also, our input texts have been only the DNA sequences, but texts with large alphabets, such as most texts in Asian languages, should also be investigated.

   On the other hand, during the implementation of our construction algorithms, we feel that it would be convenient if the implementations of the existing compressed data structures, such as `rank` and `select` data structures of Jacobson [42], are available through a simple library call. The quest for such a library, like LEDA [59], will be a most meaningful piece of research in the era of overwhelming information.