COMPRESSED INDEXING DATA STRUCTURES FOR BIOLOGICAL SEQUENCES

DO HUY HOANG

(*B.C.S. (Hons), NUS*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Do Huy Hoang

November 25, 2012

Acknowledgement

I would like to express my special thanks of gratitude to my supervisor Professor Sung Wing-Kin for valuable lessons and supports throughout my research. I am also grateful to Jesper Jansson, Kunihiko Sadakane, Franco P. Preparata, Kwok Pui Choi, Louxin Zhang for their great discussions and collaborations. Last but not least, I would like to thank my family and friends for their caring before and during my research.

Contents

List of Figures

Summary

A compressed text index is a data structure that stores a text in the compressed form while efficiently supports pattern searching queries. This thesis investigates three compressed text indexes and their applications in bioinformatics.

Suffix tree, suffix array, and directed acyclic word graph (DAWG) are the pioneers text indexing structures developed during the 70's and 80's. Recently, the development of compressed data-structure research has created many structures that use surprisingly small space while being able to simulate all operations of the original structures. Many of them are compressed versions of suffix arrays and suffix trees, however, there is still no compressed structure for DAWG with full functionality. Our first work introduces an $nH_k(\overline{S}) + 2nH_0^*(\mathcal{T}_{\overline{S}}) + o(n)$-bit compressed data-structure for simulating DAWG where $H_k(\overline{S})$ and $H_0^*(\mathcal{T}_{\overline{S}})$ are the empirical entropy of the reversed input sequence and the suffix tree topology of the reversed sequence, respectively. Besides, we also proposed an application of DAWG that improves the time complexity of local alignment problem. In this application, using DAWG, the problem can be solved in $O(n^{0.628}m)$ average case time and $O(nm)$ worst case time where n and m are the lengths of the database and the query, respectively.

In the second work, we focus on text indexes for a set of similar sequences. In the context of genomic, these sequences are DNA of related species which are highly similar, but hard to compress individually. One of the effective compression schemes for this data (called delta compression) is to store the first sequence and the changes in term of insertions and deletions between each pair of sequences. However, using this scheme, many types of queries on the sequences cannot be supported effectively. In the first part of this work, we design a data structure to support the rank and select queries in the delta compressed sequences. The data structure is called multi-version rank/select. It answers the rank and select queries in any sequence in $O(\log\log\sigma + \log m/\log\log m)$ time where m is the number of changes between input sequences. Based on this result, we propose an indexing data structure for similar sequences called multi-version FM-index which can find a pattern P in $O(|P|(\log m + \log\log\sigma))$ average time for any sequence S_i.

Our third work is a different approach for similar sequences. The sequences are

compressed by a scheme called relative Lempel-Ziv. Given a (large) set $\mathcal{S}$ of strings, the scheme represents each string in $\mathcal{S}$ as a concatenation of substrings from a constructed or given reference string R. This basic scheme gives a good compression ratio when every string in $\mathcal{S}$ is similar to R, but does not provide any pattern searching functionality. Our indexing data structure offers two trade-offs between the index space and the query time. The smaller structure stores the index in asymptotically optimal space, while the pattern searching query takes logarithmic time in term of the reference length. The faster structure blows up the space by a small factor and pattern query takes sub-logarithmic time.

Apart from the three main indexing data structures, some additional novel structures and improvements to existing structures may be useful for other tasks. Some examples include the bi-directional FM-index in the RLZ index, the multi-version rank/select, and the k-th line cut in the multi-version FM index.

Chapter 1

Background

1.1 Introduction

As more and more information is generated in the text format from sources like biological research, the internet, XML database and library archive, the problem of storing and searching within text collections becomes more and more important and challenging. A *text index* is a data structure that pre-processes the text to facilitate efficient pattern searching queries. Once a text is indexed, many string related problems can be solved efficiently. For example, computing the number of occurrences of a string, finding the longest repeated substring, finding repetitions in a text, searching for a square, computing the longest common substring of a finite set of strings, on-line substring matching, and approximate string matching [3, 56, 86, 108]. The solutions for these problems find applications in many research areas. However, the two most popular practical applications of text indexes are, perhaps, in DNA sequence database and in natural language search engines where the data volume is enormous and the performance is critical.

In this thesis, we focus on indexes that work for biological sequences. In contrast to natural language text, these sequences do not have syntactical structure like word or phrase. Thus, it makes word based structures such as inverted indexes [116] which are popular in natural language search engines less suitable. Instead, we focus on the most general type of text indexes called *full-text* index [88] where it is possible to search for any substring of the text.

The early researches on full-text indexing data structures e.g. suffix tree [112], directed acyclic word graph [14], suffix array [48, 80] were more focused on construction algorithms

[82, 110, 31] and query algorithms[80]. The space was measured by the big-Oh notations in terms of memory words which hides all constant factors. However, as indexing data structures usually need to hold a massive amount of data, the constant factors cannot be neglected. The recent trend of data structure research has been paying more attention on the space usage. Two important types of space measurement concepts emerged. A *succinct* data structure requires the main order of space equals the theoretical optimal of its inputs data. A *compressed* data structure exploits regularity in some subset of the possible inputs to store them in less than the average requirement. In text data, compression is often measure in terms of the k-order empirical entropy of the input text denoted H_k. It is the lower bound for any algorithm that encodes each character based on a context of length k.

Consider a text of length n over an alphabet of size σ, the theoretical information for this text is $n \log \sigma$ bits, while the most compact classical index, the suffix array, stores a permutation of $[1..n]$ which costs $O(n \log n)$ bits. When the text is long and the alphabet is small in case of DNA sequences (where $\log \sigma$ is 2 and $\log n$ is at least 32), there is a huge difference between the succinct measurement and the classical index storage.

Initiated by the work of Jacobson [61], data structures in general and text indexes in particular have been designed using succinct and compressed measurements. Several succinct and compressed versions of the suffix array and the suffix tree with various space-time trade-offs were introduced. For suffix array, after observing some self repetitions in the array, Grossi and Vitter [54] have created the first succinct suffix array that is close to $n \log \sigma$ bit-space with the expense that the query time of every operation is increased by a factor of $\log n$. The result was further refined and developed into some fully compressed forms [101, 75, 52], with the latest structure uses $(1 + \frac{1}{\epsilon})nH_k + o(n \log \sigma)$ bits, where $\epsilon \leq 1$. Simultaneously, Ferragina and Manzini introduced a new type of indexing scheme [36] called FM-index which is related to suffix array, but has novel representation and searching algorithm. This family of indexes stores a permutation of the input text (called Burrows-Wheeler transform [17]), and uses a variety of text compression techniques [36, 39, 77, 106] to achieve the space of $nH_k + o(n \log \sigma)$ while theoretically having faster pattern searching compared to suffix array of the same size. Suffix tree is a more complex structure, therefore, the compressed suffix trees only appeared after the maturity of the suffix array and structures for succinct tree representations. The first compressed suffix

tree proposed by Sadakane[102] uses $(1+\epsilon)nH_k + 6n + o(n)$ bits while slowing down some tree operations by $\log n$ factor. Further developments [99] have reduced the space to $nH_k + o(n)$ bits while the query time of every operation is increased by another factor of $\log \log n$.

Another trend in compressed index data structure is building text indexes based on Lempel-Ziv and grammar based compression. For example, some indexes based on Lempel-Ziv compression are LZ78[7], LZ77[65], RLZ[27]. Indexes based on grammar compression are SLP[22, 46], CFG[23]. Unlike the previous approach where succinct and compression techniques are applied to existing indexing data structure to reduce the space, this approach starts with some known text compression method, then builds an index base on the compression. The performance of these indexes are quite diverse, and highly depend on the details of the base compression methods. However, compared to compressed suffix tree and compressed suffix array, searching for pattern in these indexes are usually more complex and slower [7], however, decompressing substrings from these indexes are often faster.

Some other research directions in the full-text indexing data structure field includes: indexes in external memory (for suffix array[35, 105], for suffix tree[10], for FM-index[51], and in general [57]), parallel and distributed indexes[97], more complex queries[59], dynamic index[96], better construction algorithms (for suffix array[93], for suffix tree in external memory[9], for FM-index in external memory [33], for LZ78 index[5]). This list is far from complete, but it helps to show the great activity in the field of indexing data structure.

Although many text indexes have been proposed so far, in bioinformatics, the demand for innovations does not decline. The general full-text data structures like suffix tree, suffix array are designed without assumption about the underlying sequences. In bioinformatics, we still know very little about the details of nature sequences; however, some important characteristics of biological sequences have been noticed. First of all, the underlying process governing all the biological sequences is evolution. The traces of evolution are shown in the similarity and the gradual changes between related biological sequences. For example, the genome similarity between human beings are 99.5–99.9%, between human and chimpanzees are 96%–98% and between human and mouse are 75–90%, depending on how "similarity" is measured. Secondly, although the similarity between

related sequences is high, their fragments seem to be purely random. Many compression schemas that look for local regularity cannot perform well. For example, when using gzip to compress the human genome, the size of the result is not significant better than storing the sequence compactly using 2 bits per DNA character. (Note that DNA has 4 characters in total.)

As more knowledge of the biological sequence accumulated, our motivation for this thesis is to design specialized compressed indexing data structures for biological data and applications. First, Chapter 2 describes a compressed version of directed acyclic word graph (DAWG). It can be seen as a member of the suffix array and suffix tree family. Apart from being the first compressed full-functional version of its type, we also explore its application in local alignment, a popular sequence similarity measurement in bioinformatics. In this application, DAWG can have good the average time and have better worst case guarantee. The second index in Chapter 3 also belongs to suffix tree and suffix array family. However, the text targeted are similar sequences with gradual changes. In this work, we record the changes by marking the insertions and deletions between the sequences. Then, the indexes and its auxiliary data structures are designed to handle the delta compressed sequences, and answer the necessary queries. The last index in Chapter 4 is also for similar sequences, but based on RLZ compression, a member of the Lempel-Ziv family. In this approach, the sequences are compressed relatively to a reference sequence. This approach can avoid some of the shortcoming of the delta compression method, where large chunks of DNA change locations in the genome.

1.2 Preliminaries

This section introduces notations and definitions that are used through out the thesis.

1.2.1 Strings

An *alphabet* is a finite total ordered set whose elements are called *characters*. The conventional notation for an alphabet is Σ, and for its size is σ. An *array* (a.k.a. *vector*) $A[1..n]$ is a collection of n elements such that each element $A[i]$ can be accessed in constant time. A *string* (a.k.a. *sequence*) over an alphabet Σ is a array where elements are member of the alphabet.

Consider a string S, let $S[i..j]$ denote a substring from i to j of S. A *prefix* of a string S is a substring $S[1..i]$ for some index i. A *suffix* of a string S is substring $S[i..|S|]$ for some index i.

Consider a set of strings $\{s_1, \ldots s_n\}$ share the same alphabet Σ, the *lexicographical order* on $\{s_1, \ldots s_n\}$ is an total order such that $s_i < s_j$ if there is an index k such that $s_i[1..k] = s_j[1..k]$ and $s_i[k+1] < s_j[k+1]$.

Consider a string $S[1..n]$, S can be stored using $\lceil n \log \sigma \rceil$ bits. However, when the string S has some regularities, it can be stored in less space. One of the popular measurement for text regularity is the *empirical entropy* in [81]. The zero order empirical entropy of string S is defined as

$$H_0(S) = - \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n_c}{n}$$

where n_c is the number of occurrences of character c in S.

Then, the k-th order empirical entropy of S is defined as

$$H_k(S) = \sum_{w \in \Sigma^k} \frac{|w_S|}{n} H_0(w_S)$$

where Σ^k is a set of length k strings, and w_S is the string of characters that $w_S[i]$ is the character that follows the i-th occurrence of w in S.

Note that $nH_k(S)$ is a lower bound for the number of bits needed to compress S using any algorithm that encodes each character regarding only the context of k characters before it in S (See [81]). We have $H_k(S) \leq H_{k-1}(S) \leq \ldots \leq H_0(S) \leq \log \sigma$.

1.2.2 *rank* and *select* data structures

Let $B[1..n]$ be a bit vector of length n with k ones and $n-k$ zeros. The *rank* and *select* data structure of B supports two operations: $\text{rank}_B(i)$ returns the number of ones in $B[1..i]$; and $\text{select}_B(i)$ returns the position of the i-th one in B.

Proposition 1.1. *(Pătraşcu [92]) There exists a data structure that presents bit vector B in $\log \binom{n}{k} + o(n)$ bits and supports operations $\text{rank}_B(i)$ and $\text{select}_B(i)$ in $O(1)$ time.*

A generalized rank/select data structure for a string is defined as follows. Consider a string $S[1..n]$ over an alphabet of size σ, rank/select data structure for string S supports

two similar queries. The query $rank(S, c, i)$ counts the number of occurrences of character c in $S[1..i]$. The query $select(S, c, i)$ finds the i-th position of the character c in S.

Proposition 1.2. *(Belazzougui and Navarro [12]) There exists a structure that requires* $nH_k(S) + o(n \log \sigma)$ *bits and answers the rank and select queries in* $O(\log \frac{\log \sigma}{\log \log n})$ *time.*

1.2.3 Some integer data structures

Given an array $A[1..n]$ of non-negative integers, where each element is at most m, we are interested in the following operations: $\text{max\_index}_A(i, j)$ returns $\arg\max_{k \in i..j} A[k]$, and $\text{range\_query}_A(i, j, v)$ returns the set $\{k \in i..j : A[k] \geq v\}$. In case that $A[1..n]$ is sorted in non-decreasing order, operation $\text{successor\_index}_A(v)$ returns the smallest index i such that $A[i] \geq v$. The data structure for this operation is called the *y-fast trie* [113]. The complexities of some existing data structures supporting the above operations are listed in the table in Fig 1.1.

Operation	Extra space	Time	Reference	Remark
$\text{rank}_B(i)$, $\text{select}_B(i)$	$\log \binom{n}{k} + o(n)$	$O(1)$	[92]	
$\text{max\_index}_A(i, j)$	$2n + o(n)$	$O(1)$	[43]	
$\text{range\_query}_A(i, j, v)$	$O(n \log m)$	$O(1 + occ)$	[85], p. 660	
$\text{successor\_index}_A(v)$	$O(n \log m)$	$O(\log \log m)$	[113]	A is sorted

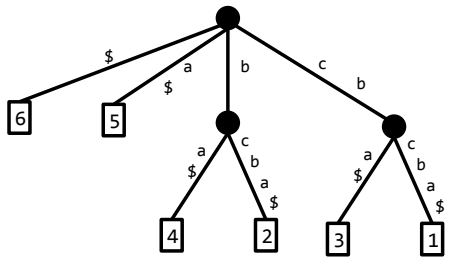
Figure 1.1: The time and space complexities to support the operations defined above.

1.2.4 Suffix data structures

Suffix tree and suffix array are classical data structure for text indexing, numerous books and surveys [56, 88, 111] have thoroughly covered them. Therefore, this section only introduces the three core definitions that are essential for our works. They are structures of suffix tree, suffix array and Burrows-Wheeler transform.

Index	Start pos.	Suffix	BW_S
1	6	\$	a
2	5	a\$	b
3	4	ba\$	c
4	2	bcba\$	c
5	3	cba\$	b
6	1	cbcba\$	\$

(a)



(b)

Figure 1.2: Suffix array and suffix tree of "cbcba". The suffix ranges for "b" and "cb" are (3,4) and (5,6), respectively.

Consider any string S with a special terminating character \$ which is lexicographically smaller than all the other characters. The *suffix tree* $\mathcal{T}_S$ of the string S is a tree whose edges are labelled with strings such that every suffix of S corresponds to exactly one path from the tree's root to a leaf. Figure 1.2(b) shows an example suffix tree for *cbcba*\$. Searching for a pattern P in the string S is equivalent to finding a path from the root of the suffix tree $\mathcal{T}_S$ to a node of $\mathcal{T}_S$ or a point in the edge in which the labels of the travelled edges equals P.

For a string S with the special terminating character \$, the *suffix array* SA_S is the array of integers specifying the starting positions of all suffixes of S sorted lexicographically. For any string P, let st and ed be the smallest and the biggest, respectively, indexes such that P is the prefix of suffix $SA_S[i]$ for all $st \leq i \leq ed$. Then, (st, ed) is called a *suffix range* or SA_S*-range* of P. i.e. P occurs at positions $SA_S[st], SA_S[st+1], \ldots, SA_S[ed]$ in S. See Fig. 1.2(a) for example. Pattern searching of P can be done using binary searches in suffix array SA_S to find the suffix range of P (as in [80]).

The Burrows-Wheeler transform [17] of S is a sequence which can be specified as follows:

$$BW_S[i] = \begin{cases} S[SA_S[i]-1]] & \text{if } SA_S[i] \neq 1 \\ S[n] & \text{if } SA_S[i] = 1 \end{cases}$$

For any given string P specified by its suffix range (st, ed) in SA_S, operation $backward\_search_S(c, (st, ed))$ returns the suffix range in SA_S of the string $P' = cP$, where c is any character and (st, ed) is the suffix range of P. The operation $backward\_search_S$ can be implemented as follows [36].

```
1 function backward_search_S(c, (st, ed))
2     Let l_c be the total number of characters in S that is alphabetically less than c
3     st' = l_c + rank(BW_S, c, st − 1) + 1
4     ed' = l_c + rank(BW_S, c, ed)
5     return (st', ed')
```

Using *backward_search*, the pattern searching for a string P can be done by extending one character at a time.

1.2.5 Compressed suffix data structures

For a text of length n, storing its suffix array or suffix tree explicitly requires $O(n \log n)$ bits, which is space inefficient. Several compressed variations of suffix array and suffix tree have been proposed to address the space problem. In this section, we discuss about three important sub-families of compressed suffix structures: compress suffix arrays, FM-indexes and compressed suffix trees. Note that, the actual boundaries between the sub-families are quite blur, since the typical operations of structures from one sub-family can usually be simulated by structures from other sub-family with some time penalty. We try to group the structures by their design influences.

First, most of the compressed suffix arrays represent data using the following framework. They store a compressible function called Ψ_S and a sample of the original array. The $\Psi_S(i)$ is a function that returns the index j such that $SA_S[j] = SA_S[i] + 1$, if $SA_S[i] + 1 \leq n$, and $SA_S[j] = 1$ if $SA_S[i] = n$. For any i, entry $SA_S[i]$ can be computed by $SA_S[i] = SA_S[\Psi^k(i)] - k$ where $\Psi^k(i)$ is $\Psi(\Psi(\ldots\Psi(i)\ldots))$ k-time. An algorithm using function Ψ_S to recover the original suffix array from its samples is to iteratively apply Ψ_S until it finds a sampled entry. The data structures in compressed suffix array family are different by the details of how Ψ_S is compressed and how the array is sampled. Fig. 1.3 summarized recent compressed suffix arrays with different time-space trade-offs.

Reference	Space	Ψ_S time	$SA_S[i]$ time
Sadakene[101]	$(1+\frac{1}{\epsilon})nH_0(S) + O(n \log\log\sigma) + \sigma\log\sigma$	$O(1)$	$O(\log^{\epsilon} n)$
Grossi et al.[52]	$(1+\frac{1}{\epsilon})nH_k(S) + 2(\log e + 1)n + o(n)$	$O(\frac{\log\sigma}{\log\log n})$	$O(\frac{\log\sigma \log^{\epsilon} n}{\log\log n})$
Grossi et al.[52]	$(1+\frac{1}{\epsilon})nH_k(S) + O\left(\frac{n\log\log n}{\log_{\sigma}^{\epsilon/(1+\epsilon)} n}\right)$	$O(1)$	$O(\log_{\sigma}^{\epsilon} n + \log\sigma)$
Ferragina et al.[40]	$nH_k(S) + O(\frac{n\log\sigma\log\log n}{\log n}) + O(\frac{n}{\log^{\epsilon} n})$	$O(\frac{\log\sigma}{\log\log n})$	$O(\frac{\log^{1+\epsilon} n \log\sigma}{\log\log n})$

Figure 1.3: Some compressed suffix array data structures with different time-space trade-offs. Note that structure in [40] is also an FM-index.

Second sub-family of the compressed suffix structures is the FM-index sub-family. These indexes based on the compression of the Burrows-Wheeler transform sequence while allowing rank and select operations. The first proposal [36] uses move-to-front transform, then run-length compression, and a variable-length prefix code to compress the sequence. Their index uses $5nH_k(S) + o(n \log\sigma)$ bits for any alphabet of size σ which is less than $\log n / \log\log n$. Subsequently, they developed techniques focused on scaling the index for larger alphabet [39, 76], improving the space bounds[40, 77], refining the technique for practical purpose [34], and speeding up the location extraction operations [49]. For

	Sadakene[102]	Fischer et al.[44]	Russo et al.[99]
Space	$(1+\frac{1}{\epsilon})nH_k(S)+6n+o(n)$	$(1+\frac{1}{\epsilon})nH_k(S)+o(n)$	$nH_k(S)+o(n)$
Child	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n(\log\log n)^2)$
Edge label letter	$O(\log^\epsilon n)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
Suffix link	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$
Other tree nav.	$O(1)$	$O(\log^\epsilon n)$	$O(\log n \log\log n)$

Figure 1.4: Some compressed suffix tree data structures with different time-space trade-offs. Note that we only list the operation time of some important operations.

theoretical purposes, the result from [40] supersedes all the previous implementations, therefore, we use it as a general reference for FM-index. The index uses $nH_k(S)+o(n\log\sigma)$ bits, while supports the backward search operation in $O(\log\sigma/\log\log n)$ time.

The third sub-family of compressed suffix structures is compressed suffix tree. The operations of the structures in this sub-family are usually emulated by using suffix array or FM-index plus two other components called tree topology and LCP array. The tree topology records the shape of the suffix tree. For any index $i > 1$, the entry $LCP[i]$ stores the length of the longest common prefix of $S[SA_S[i]..n]$ and $S[SA_S[i-1]..n]$, and $LCP[1] = 0$. The LCP array can be used to deduce the lengths of the suffix tree branches. The first fully functional suffix tree proposed by Sadakane [102] stores the LCP array in $2n + o(n)$ bits, the tree topology in $4n + o(n)$ bits and an compressed suffix array. Further works [99, 44] on auxiliary data structures reduces the space requirement for the tree topology and the LCP array to $o(n)$. Fig. 1.4 shows some interesting space-time trade-offs for compressed suffix trees.

Chapter 2

Directed Acyclic Word Graph

2.1 Introduction

Among all text indexing data-structures, suffix tree [112] and suffix array [80] are the most popular structures. Both suffix tree and suffix array index all possible suffixes of the text. Another variant is directed acyclic word graph (DAWG) [14]. This data-structure uses a directed acyclic graph to model all possible substrings of the text.

However, all above data-structures require $O(n \log n)$-bit space, where n is the length of the text. When the text is long (e.g. human genome whose length is 3 billions basepairs), those data-structures become impractical since they consume too much memory. Recently, due to the advance in compression methods, both suffix tree and suffix array can be stored in only $O(nH_k(S))$ bits [102, 62]. Nevertheless, previous works on DAWG data structures [14, 24, 60] focus on explicit construction of DAWG and its variants. They not only require much memory but also cannot return the locations of the indexed sub-string. Recently, Li et al. [73] also independently presented a DAWG by mapping its nodes to ranges of the reversed suffix array. However, their version can only perform forward enumerate of the nodes of the DAWG. A practical, full functional and small data structure for DAWG is still needed.

In this chapter, we propose a compressed data-structure for DAWG which requires only $O(nH_k(S))$ bits. More precisely, it takes $n(H_k(\overline{S}) + 2H_0^*(\mathcal{T}_{\overline{S}})) + o(n)$ bit-space, where $H_k(\overline{S})$ and $H_0^*(\mathcal{T}_{\overline{S}})$ is the empirical entropy of the reversed input sequence and the suffix tree topology of the reversed sequence. Our data-structure supports navigation of the DAWG in constant time and decodes each of the locations of the substrings

represented in some node in $O(\log n)$ time.

In addition, this chapter also describes one problem which can be solved more efficiently by using the DAWG than suffix tree. This application is called local alignment; the input is a database S of total length n and a query sequence P of length m. Our aim is to find the best local alignment between the pattern P and the database S which maximizes the number of matches. This problem can be solved in $\Theta(nm)$ time by the Smith-Waterman algorithm [107]. However, when the database S is known in advance, we can improve the running time. There are two groups of methods (see [108] for a detailed survey of the methods). One group is heuristics like Oasis[83] and CPS-tree[114] which do not provide any bound. Second group includes Navarro et al. method[87] and Lam et al. method[70] which can guarantee some average time bound. Specifically, the previously proposed solution in [70] built suffix tree or FM-index data-structures for S. The best local alignment between P and S can be computed in $O(nm^2)$ worst case time and $O(n^{0.628}m)$ expected time in random input for the edit distance function or a scoring function similar to BLAST [2]. We showed that, by building the compressed DAWG for S instead of suffix tree, the worst case time can be improved to $O(nm)$ while the expected time and space remain the same. Note that, the worst case of [70] happens when the query is long and occurs inside the database. That means their algorithm runs much slower when there are many positive matches. However, the alignment is a precise and expensive process; people usually only run it after having some hints that the pattern has potential matches to exhaustively confirm the positive results. Thus, our worst case improvement means the algorithm will be faster in the more meaningful scenarios.

The rest of this chapter is organized as follows. In Section 2, we review existing data-structures. Section 3 describes how to simulate the DAWG. Section 4 shows the application of the DAWG in the local alignment problem.

2.2 Basic concepts and definitions

Let Σ be a finite alphabet and Σ^* be the set of all strings over Σ. The empty string is denoted by ε. If $S = xyz$ for strings $x, y, z \in \Sigma^*$, then x, y, and z are denoted as prefix, substring, and suffix, respectively, of S. For any $S \in \Sigma^*$, let $|S|$ be the length of S.

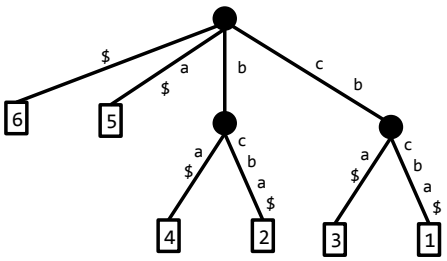


Figure 2.1: suffix tree of "cbcba"

2.2.1 Suffix tree and suffix array operations

Recall some definitions about suffix tree and suffix array from Section 1.2.4, let $\mathcal{A}_S$ and $\mathcal{T}_S$ denote the suffix array and suffix tree of string S, respectively. Any substring x of S can be represented by a pair of indexes (st, ed), called suffix range. The operation lookup(i) returns $\mathcal{A}_S[i]$. Consider a suffix range (st, ed) in $\mathcal{A}_S$ for some string $P[1..m]$, the operation backward-search(st, ed, c) returns another suffix range (st', ed') for $cP[1..m]$.

For every node u in the suffix tree $\mathcal{T}_S$, the string on the path from the root to u is called the *path label* of the node u, denoted as label(u).

In this work, we require the following operations on the suffix tree:

- parent(u): return the parent node of node u.
- leaf-rank(u): returns the number of leaves less than or equal to u in preorder sequence.
- leaf-select(i): returns the leaf of the suffix tree which has rank i.
- leftmost-child(u): returns the leftmost child of the subtree rooted at u.
- rightmost-child(u): returns the rightmost child of the subtree rooted at u.
- lca(u, v): returns the lowest common ancestor of two leaves u and v.
- depth(u): returns the depth of u. (i.e. the number of nodes from u to the root minus one).
- level-ancestor(u, d): returns the ancestor of u with depth d.
- suffix-link(u) returns a node v such that label(v) equals the string label(u) with the first character removed.

Suffix tree and suffix array are closely related. If the children of each node in the suffix tree $\mathcal{T}_S$ are ordered lexically according to the labels of the edges, the suffixes corresponding to the leaves of $\mathcal{T}_S$ are ordered exactly the same as that of the suffix array $\mathcal{A}_S$. Therefore, the rank-i leaf of $\mathcal{T}_S$ is one-to-one mapped to $\mathcal{A}_S[i]$. For any node w in the suffix tree $\mathcal{T}_S$, let u and v be the leftmost and the rightmost leaves, respectively. The suffix range of $label(w)$ is $(\text{leaf-rank}(u), \text{leaf-rank}(v))$.

In the suffix tree, some leaves hang on the tree by edges whose labels are just the single terminal character \$. These are called trivial leaves; all remaining nodes in the tree are called *non-trivial nodes*. In Fig. 2.1, leaf number 6 is a trivial leaf.

2.2.2 Compressed data-structures for suffix array and suffix tree

For a text of length n, storing its suffix array or suffix tree explicitly requires $O(n \log n)$ bits, which is space inefficient. Several compressed variations of suffix array and suffix tree, whose sizes are in $O(nH_k(S))$ bits, have been proposed to address the space problem.

For the compressed data structure on suffix array, Ferragina and Manzini introduced a variant called FM-index [36] which can be stored in $O(nH_k(S))$ bits and supports backward-search(st, ed, c) in constant time. This result was further improved by Mäkinen and Navarro [77] to $nH_k(n) + o(n)$ bits.

For the data structures on suffix tree, using the idea of Grossi et al. [52], Sadakane [102], and Jansson et al. [62], we can construct an $O(n)$-bit data-structure which supports suffix-link(u) and all the tree operations in constant time. Given a tree T, the tree degree entropy is defined as $H_0^*(T) = \sum_i \frac{n_i}{n} \log \frac{n}{n_i}$ where n is the number of nodes in T, n_i is the number of nodes with i children. Below three lemmas summarize the space and the operations supported by these data-structures.

Lemma 2.1. *(Jansson et al. [62]) Given a tree $\mathcal{T}$ of size n, there is an $nH_0^*(\mathcal{T}) + o(n)$ bits data structure that supports the following operations in constant time:* parent(u), leaf_rank(u), leaf_select(i), leftmost_child(u), rightmost_child(u) *and* lca(u, v), depth(u) *and* level-ancestor(u, d).

Lemma 2.2. *(Sadakane [102]) Given a sequence S of length n, the suffix tree $\mathcal{T}_S$ can be stored using $4n + nH_k(S) + o(n)$ bits and supports the operation* suffix-link(u) *in constant time.*

Lemma 2.3. *(Mäkinen and Navarro [77]) Given the $nH_k(n) + o(n)$ bit FM-index of the sequence S, for every suffix range (st, ed) of the suffix array and every character c, the operation* backward-search(st, ed, c) *runs in constant time; and the operation* lookup(i) *runs in $O(\log n)$ time.*

Corollary 2.4. *Given a sequence S of length n, let $\mathcal{T}_S$ be the suffix tree of S. There is a data structure that supports all the tree operations in Lemma 2.1, the* suffix-link(u) *operation in Lemma 2.2, and the* backward-search(st, ed, c) *operation in Lemma 2.3 using $n(H_k(S) + 2H_0^*(\mathcal{T}_S)) + o(n)$ bits.*

Proof. We recombine and refine the data structures from Lemma 2.1, 2.2 and 2.3 to obtain a data structure that supports the necessary operations. The data structure consists of two components: (i) the suffix tree topology from Lemma 2.1 detailed in [62], (ii) the FM-index detailed in [77]. Since the operations on tree and backward-search were already supported by these Lemma, we will only show how to simulate suffix-link operation using these two components.

In [102], $\Psi[i]$ is defined as an array such that $\Psi[i] = i'$ if $\mathcal{A}_S[i'] = \mathcal{A}_S[i] + 1$ and $\Psi[i] = 0$ otherwise. suffix-link(u) can be computed following this procedure: Let $x =$ leaf-rank(leftmost-child(u)) and $y =$ leaf-rank(rightmost-child(u)). Let $x' = \Psi[x]$ and $y' = \Psi[y]$. It is proved that suffix-link$(u) =$ lca(leaf-select(x'), leaf-select(y')). Since all the tree operations are available, we need to simulate the $\Psi[i]$ using the FM-index. This result was actually proven in [47] (Section 3.2). Therefore, all the operations can be supported using the suffix tree topology and the FM-index.

For the space complexity, the FM-index takes $nH_k(n) + o(n)$ bit-space. The suffix tree topology takes $2nH_0^*(\mathcal{T}_S) + o(n)$, since the suffix tree of a sequence of length n can have up to $2n$ nodes. The space bound therefore is $nH_k(n) + 2nH_0^*(\mathcal{T}_S) + o(n)$. □

2.2.3 Directed Acyclic Word Graph

Apart from suffix tree, we can index a text S using a directed acyclic word graph (DAWG). Prior to define the DAWG, we first define the end-set equivalence relation. Let $S = a_1 a_2 \dots a_n$ $(a_i \in \Sigma)$ be a string in Σ^*. For any non-empty string $y \in \Sigma^*$, its end-set in S is defined as end-set$_S(y) = \{i \mid y = a_{i-|y|+1} \dots a_i\}$. In particular, end-set$_S(\varepsilon) = \{0, 1, 2, \dots, n\}$. An end-set equivalence class is a set of substrings of S which have the

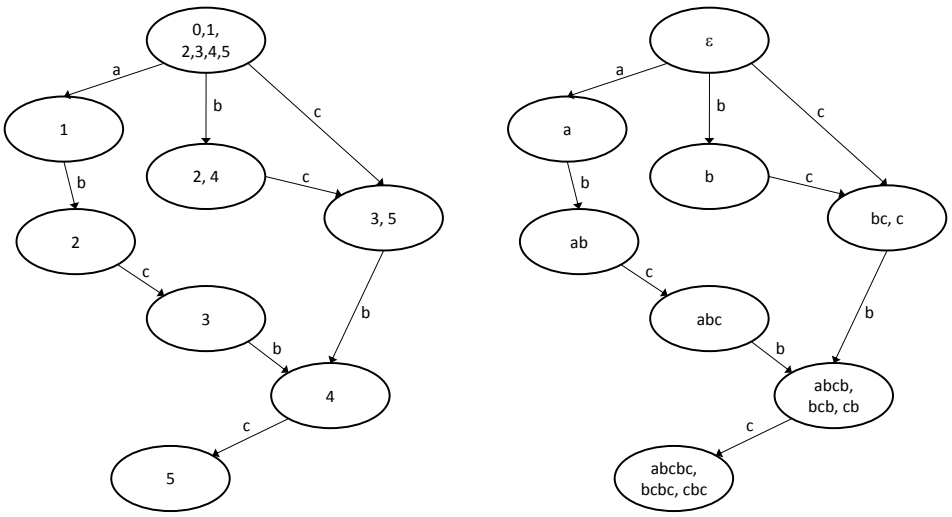


Figure 2.2: DAWG of string "abcbc" (left: with end-set, right: with set path labels).

same end-set. For any substring x of S, we denote $[x]_S$ as the end-set equivalence class containing the string x, i.e., $[x]_S = \{y \mid y \in \Sigma^*, \text{end-set}_S(x) = \text{end-set}_S(y)\}$. Note that $[x]_S = [y]_S$ if and only if $\text{end-set}_S(x) = \text{end-set}_S(y)$ for any strings x and y. Moreover, the set of all end-set equivalence classes of S forms a partition of all substrings of S.

The DAWG $\mathcal{D}_S$ for a string S is defined as a directed acyclic graph (V, E) such that V is the set of all end-set equivalence classes of S and $E = \{([x]_S, [xa]_S) \mid x \text{ and } xa \text{ are substrings of } S, \text{end-set}_S(x) \neq \text{end-set}_S(xa)\}$. Furthermore, every edge $([x]_S, [xa]_S)$ is labeled by the character a. Denote $c_{(u,v)}$ as the edge label of an edge (u, v).

In the DAWG $\mathcal{D}_S$, $[\varepsilon]_S = \{0, 1, \ldots, n\}$ is the only node with in-degree zero. Hence, $[\varepsilon]_S$ is called the source node. For every path P in $\mathcal{D}_S$ starting from the source node, let its path label be the string obtained by concatenating all labels of the edges on P. A DAWG $\mathcal{D}_S$ has an important property: For every node u in $\mathcal{D}_S$, the set of path labels of all paths between the source node and u equals the end-set equivalence class of u.

For example, Fig. 2.2 shows the DAWG for $S = \texttt{abcbc}$. We have $\text{end-set}_S(\texttt{bc}) = \text{end-set}_S(\texttt{c}) = \{3, 5\}$. Hence, $\{\texttt{bc}, \texttt{c}\}$ forms an end-set equivalence class.

The following theorem obtained from [14] states the size bound of a DAWG. Note that the size bound is tight. The upper bounds for the number of nodes and edges are achieved when $S = ab^n$ and $S = ab^nc$ respectively, for some distinct letters $a, b, c \in \Sigma$.

Theorem 2.5. *(Blumer et al. [14]) Consider any string S of length at least 3 (i.e. $n \geq 3$). The Directed Acyclic Word Graph $\mathcal{D}_S$ for S has at most $2n - 1$ states, and $3n - 4$ transition edges (regardless of the size of Σ).*

For any string x, we denote $\overline{x}$ as the reverse sequence of x. Consider a string S, let $\mathcal{D}_S$ be the DAWG of S and $\mathcal{T}_{\overline{S}}$ be the suffix tree of $\overline{S}$. For every non-trivial node u in $\mathcal{T}_{\overline{S}}$, let $\gamma(u)$ be $[\overline{\text{label}(u)}]_S$. (Please refer to the end of Section 2.1 for the definition of non-trivial.) The following lemma states the relationship between a DAWG and a suffix tree.

Lemma 2.6. *(Blumer et al. [14]) The function γ is a one-to-one correspondence mapping from the non-trivial nodes of $\mathcal{T}_{\overline{S}}$ to the nodes of $\mathcal{D}_S$.*

For example, for the suffix tree in Fig. 1.2(b) and the DAWG in Fig. 2.2, the internal node of the suffix tree with path label "cb" maps to node $[\overline{cb}]_S = [bc]_S = \{bc, c\}$ in the DAWG. In fact, every non-trivial node in the suffix tree maps to a node in the DAWG, and vice versa. Precisely, the root of the suffix tree maps to the source node of the DAWG, the internal node with path label "b" maps to node {"b"}, the internal node with path label "cb" maps to node {"bc", "c"}, leaf 5 maps to node {"a"}, leaf 4 maps to node {"ab"}, leaf 2 maps to node {"abcb","bcb","cb"}, leaf 3 maps to node {"abc"}, and leaf 1 maps to node {"abcbc","bcbc","cbc"}.

2.3 Simulating DAWG

Consider a sequence S of length n, this section describes an $O(n)$-bit data-structure for the DAWG $\mathcal{D}_S$ which supports the following four operations to navigate in the graph in constant time:

- Get-Source(): returns the source node of $\mathcal{D}_S$;
- Find-Child(u, c): returns the child v of u in $\mathcal{D}_S$ s.t. (u, v) is labeled by c.
- Parent-Count(u): returns the number of parents of u in $\mathcal{D}_S$.
- Extract-Parent(u, i): returns the i-th parent where $1 \leq i \leq$ Parent-Count(u).

We also support two operations which help to extract the substring information of each node. The first operation, denoted End-Set-Count(u), returns the number of members of the end-set at node u in constant time. The second operation, denoted Extract-End-Set(u, i), returns the i-th end point in the set in $O(\log n)$ time.

To support the operations, we can store the nodes and the edges of $\mathcal{D}_S$ directly. However, such a data-structure requires $O(n \log n)$-bit space. Instead, this section shows that, given the FM-index of $\overline{S}$ and the compressed topology of the suffix tree of $\overline{S}$ (summarized in Corollary 2.4), we can simulate the DAWG $\mathcal{D}_S$ and support all operations efficiently with $O(n)$ bits space.

First, we analyse the space complexity. Both the FM-index of $\overline{S}$ and the compressed suffix tree $\mathcal{T}_{\overline{S}}$ can be stored in $n(H_k(\overline{S}) + H_0^*(\mathcal{T}_{\overline{S}}) + o(n)$ bits.

Next, we describe how to represent the nodes in the DAWG $\mathcal{D}_S$. Lemma 2.6 implies that each non-trivial node u in $\mathcal{T}_{\overline{S}}$ is one-to-one corresponding to a node $\gamma(u)$ in $\mathcal{D}_S$. Hence, in our simulation, the non-trivial node u in $\mathcal{T}_{\overline{S}}$ represents the node $\gamma(u)$ in $\mathcal{D}_S$.

Below four subsections describe how can we support the following operations: Get-Source(), Find-Child(u, c), Parent-Count(u), Extract-Parent(u, i), End-Set-Count(u) and Extract-End-Point(u, i). The implementation details is shown in Listings 2.1, 2.2, 2.3 and 2.4.

```
1 function Get−Source() { return the root node of T_S̄; }
```

Listing 2.1: Operation Get-source: returns the source node of $\mathcal{D}_S$

```
1 function Find−Child(u, c)
2     st, ed = leftmost-child(u), rightmost-child(u);
3     st′, ed′ = backward-search(st, ed, c);
4     if ((st′, ed′) is a valid range)
5         l, r = leaf-select(st′), leaf-select(ed′);
6         return lca(l, r);
7     else return nil;
```

Listing 2.2: Operation Find-Child: finds the child node v of u such that the edge label of $(u\, v)$ is c

```
1 function Parent−Count(u)
2     if (u is the root node) return 0; /* no parent for source node */
3     v = parent(u);
4     b = suffix-link(u);
5     if (v is the root node) /* The list is [b, p_2, .., p_{k−1}, v], where p_i is parent*/
6         return depth(b) − depth(v) + 1;/* of p_{i−1}, p_2 is parent of b, v is parent of p_{k−1}*/
```

```
7     else
8         e = suffix-link(v); /* The list is [b, p_2, .., p_{k-1}, e) */
9         return depth(e) − depth(b); /* (exluding e)                    */
10
11 function Extract−Parent(u, i)
12     b = suffix-link(u);
13     return level-ancestor(b, depth(b) + i − 1)
```

Listing 2.3: Operation Parent-Count and Extract-Parent: use to list parents of the node u in $\mathcal{D}_S$

```
1  function End−Set−Count(u)
2      st, ed = leftmost-child(u), rightmost-child(u);
3      return ed − st + 1;
4
5  function Extract−End−Point(u, i)
6      st = leftmost-child(u);
7      return n + 1 − lookup(i + st − 1);
```

Listing 2.4: Operations End-Set-Count and Extract-End-Point

2.3.1 Get-Source operation

The source node in $\mathcal{D}_S$ is $[\varepsilon]_S$, which is represented by the root in $\mathcal{T}_{\overline{S}}$. Hence, the operation Get-Source() just returns the root in $\mathcal{T}_{\overline{S}}$, which takes constant time.

2.3.2 End-Set operations

Each node in the DAWG $\mathcal{D}_S$ is represented directly by a node in the suffix tree $\mathcal{T}_{\overline{S}}$. Consider a non-trivial node u in $\mathcal{T}_{\overline{S}}$, operations End-Set-Count(u) and Extract-End-Point(u, i) can be used to list the ending locations of $\overline{\text{label}(u)}$ in string S. In fact, these ending locations can be derived from the starting location of label(u) in $\overline{S}$.

By definition, the starting locations of label(u) in $\overline{S}$ are $\{\mathcal{A}_{\overline{S}}[i] \mid i = st, \ldots ed\}$ where st = leftmost_child(u) and ed = rightmost_child(u). Hence, the ending locations of $\overline{\text{label}(u)}$ in S are $\{n+1-\mathcal{A}_{\overline{S}}[i] \mid i = st, \ldots, ed\}$. Line 2 in Listings 2.4 captures st and ed. The size of the end-set is thus $ed - st + 1$. To extract each ending location, we can use operation Extract-End-Point(u, i). Line 7 computes $\mathcal{A}_{\overline{S}}[i + st - 1]$ by calling the lookup

operation of the FM-index of $\overline{S}$ and reports the locations. Since the lookup operation in FM-index takes $O(\log n)$ time, the cost of extracting each end point is $O(\log n)$ time.

2.3.3 Child operation

Consider a non-trivial node u in $\mathcal{T}_{\overline{S}}$ which represents the node $\gamma(u)$ in $\mathcal{D}_S$. This section describes the operation Find-Child$_S(u, c)$ which returns a non-trivial node v in $\mathcal{T}_{\overline{S}}$ such that $\gamma(v)$ is the child of $\gamma(u)$ with edge label c. Our solution is based on the following two lemmas:

Lemma 2.1. *Consider a string S, the DAWG $\mathcal{D}_S$, and the suffix tree $\mathcal{T}_{\overline{S}}$. For any non-trivial node u in $\mathcal{T}_{\overline{S}}$, if $v =$ Find-Child(u, c) is not nil in $\mathcal{T}_{\overline{S}}$, then $(\gamma(u), \gamma(v))$ is an edge in $\mathcal{D}_S$ with edge label c.*

Proof. Suppose x is the path label of u in $\mathcal{T}_{\overline{S}}$. Line 2 in Listing 2.2 converts the node u to the suffix range (st, ed) in $\mathcal{A}_{\overline{S}}$ which represents the same substring x. By the definition of backward-search(st, ed, c), line 3 finds the suffix range (st', ed') in $\mathcal{A}_{\overline{S}}$ which represents cx. Since v is not nil, (st', ed') is a valid range. After the computation in line 5, st' and ed' are mapped back to two leaves l and r, respectively, of $\mathcal{T}_{\overline{S}}$. Note that label$(l)$ and label(r) both share cx as the prefix. Hence, cx should be a prefix of the path label of $v = \text{lca}(l, r)$. In addition, since cx does not contain the terminal character \$, v should be a non-trivial node. As label(v) is at least longer than $x =$ label(u), u and v are different nodes in $\mathcal{T}_{\overline{S}}$. By Lemma 2.6, $\gamma(u) = [\overline{x}]_S$ and $\gamma(v) = [\overline{x}c]_S$ are different. By the definition of $\mathcal{D}_S$, $(\gamma(u), \gamma(v)) = ([\overline{x}]_S, [\overline{x}c]_S)$ is an edge in $\mathcal{D}_S$ with edge label c. □

Lemma 2.2. *For any node u in $\mathcal{T}_{\overline{S}}$, if* Find-child$(u, c)$ *is nil, then $\gamma(u)$ will not have any child with edge label c in $\mathcal{D}_S$.*

Proof. By contrary, assume that there is a node $\gamma(v)$ in $\mathcal{D}_S$ such that $(\gamma(u), \gamma(v))$ is an edge in $\mathcal{D}_S$ with label c. Let $x =$ label(u) in $\mathcal{T}_{\overline{S}}$. By definition, $\overline{x}$ is one of the path labels from the source node to $\gamma(u)$ in $\mathcal{D}_S$. Since $\gamma(v)$ is a child of $\gamma(u)$ with edge label c, $\overline{x}c$ is a substring of S. However, since backward-search(st, ed, c) does not return a valid range, cx is not a substring of $\overline{S}$, i.e. $\overline{x}c$ is not a substring of S, which is a contradiction. □

Based on the above lemmas, given a non-trivial node u in $\mathcal{T}_{\overline{S}}$ which represents the node $\gamma(u)$ in $\mathcal{D}_S$, the algorithm Find-child$_S(u, c)$ in Listing 2.2 returns another non-trivial node v in $\mathcal{T}_{\overline{S}}$ such that $\gamma(v)$ is the child of $\gamma(u)$ with edge label c.

Since backward-search(st, ed, c), lefmost-child(u), rightmost-child(u), leaf-select(i), lca(u, v) each take $O(1)$ time, Find-child$_S(u, c)$ can be computed in $O(1)$ time.

2.3.4 Parent operations

Consider a non-trivial node u in $\mathcal{T}_{\overline{S}}$ which represents the node $\gamma(u)$ in $\mathcal{D}_S$. This section describes the operation Parent-Count(u) and Extract-Parent(u, i) which can be used to list all parents of $\gamma(u)$. Precisely, we present a constant time algorithm which finds two non-trivial nodes b and e in $\mathcal{T}_{\overline{S}}$ where e is the ancestor of b in $\mathcal{T}_{\overline{S}}$. We show that $\gamma(p)$ is a parent of $\gamma(u)$ in $\mathcal{D}_S$ if and only if node p is in the path between b and e in $\mathcal{T}_{\overline{S}}$. Our solution is based on the following lemmas.

Lemma 2.3. *Consider a non-trivial node u such that u is not the root of $\mathcal{T}_{\overline{S}}$, let v be u's parent and $x =$ label(v) and $xy =$ label(u). For any non-empty prefix z of y, we have $\gamma(u) = [\overline{(xy)}]_S = [\overline{(xz)}]_S$. In fact, $\gamma(u) = \{\overline{(xz)} \mid z$ is a non-empty prefix of $y\}$.*

Proof. Let $\{o_i\}$ be the set of starting positions where xy occurs in $\overline{S}$. By definition, end-set$_S(\overline{(xy)}) = \{n - o_i\}$. Consider a string xz where z is some non-empty prefix of y. Since there is no branch between u and v in $\mathcal{T}_{\overline{S}}$, xz is the prefix of all suffixes represented by the leaves under the subtree at u. Hence, the set of starting locations of xz in $\overline{S}$ and that of xy are exactly the same, which is $\{o_i\}$. By definition, end-set$_S(\overline{(xz)}) = \{n - o_i + 1\}$. Hence, $\gamma(u) = [\overline{(xy)}]_S = [\overline{(xz)}]_S$.

Note that only xz can occur at $\{o_i\}$ in $\overline{S}$ for all non-empty prefix z of y. Thus, $\gamma(u) = \{\overline{(xz)} \mid z$ is a non-empty prefix of $y\}$. □

For any non-trivial node u in $\mathcal{T}_{\overline{S}}$, below two lemmas states how to find the parents of $\gamma(u)$ in $\mathcal{D}_S$. Lemma 2.4 covers the case when u's parent is not a root node of $\mathcal{T}_{\overline{S}}$; and Lemma 2.5 covers the other case.

Lemma 2.4. *Consider a non-trivial node u whose parent, v, is not the root node in $\mathcal{T}_{\overline{S}}$. Suppose* suffix-link$(u) = b$ *and* suffix-link$(v) = e$. *For every node p in the path from b to e (excluding e) in $\mathcal{T}_{\overline{S}}$, $\gamma(p)$ is a parent of $\gamma(u)$ in $\mathcal{D}_S$.*

Proof. Since v is not the root node, let ax and axy be the path labels of v and u, respectively, in $\mathcal{T}_{\overline{S}}$ where $a \in \Sigma$ and $x, y \in \Sigma^*$. By the definition of suffix link, we have $x =$ label(e) and $xy =$ label(b). Note that a suffix link from a non-trivial node points to another non-trivial node.

(Necessary condition) For any node p on the path from b to e in $\mathcal{T}_{\overline{S}}$, the path label of p is label$(p) = xz$ where z is some non-empty prefix of y. Since p and u are two different nodes in $\mathcal{T}_{\overline{S}}$, $\gamma(p)$ and $\gamma(u)$ are two different nodes in $\mathcal{D}_S$ (see Lemma 2.6). From Lemma 2.3, $\gamma(u) = [\overline{(axy)}]_S = [\overline{(axz)}]_S$. By definition of DAWG, $(\gamma(p), \gamma(u)) = ([\overline{(xz)}]_S, [\overline{(axz)}]_S)$ is an edge in $\mathcal{D}_S$ with edge label a. This implies that $\gamma(p)$ is a parent of $\gamma(u)$.

(Sufficient condition) Note that label$(v) = ax$ and label$(u) = axy$ in $\mathcal{T}_{\overline{S}}$. By Lemma 2.3, $\gamma(u) = \{\overline{(axz)} \mid z \text{ is non-empty prefix of } y\}$. Suppose $\gamma(p)$ is parent of $\gamma(u)$ in $\mathcal{D}_S$. By definition of DAWG, $\gamma(p)$ must be $[\overline{(xz)}]_S$ for some z is non-empty prefix of y. This implies that the path label of p in $\mathcal{T}_{\overline{S}}$ is xz. Thus, p is a node on the path from b to e excluding e. □

Lemma 2.5. *Consider a non-trivial node u whose parent is the root node of $\mathcal{T}_{\overline{S}}$. Suppose* suffix-link$(u) = b$. *The set of parents of $\gamma(u)$ in $\mathcal{D}_S$ is $\{\gamma(p) \mid p$ is any node on the path from b to the root in $\mathcal{T}_{\overline{S}}\}$.*

Proof. Let v be the root node of $\mathcal{T}_{\overline{S}}$. Let ax be the path label of u. We have label$(b) = x$. From Lemma 2.3, $\gamma(u) = [z]_S$ where z is any non-empty prefix of x. Since every node p on the path from the root to b (excluding the root) has a path label which is a non-empty prefix of x. Similar to the argument in Lemma 2.4, we can show that $\gamma(p)$ is a parent of $\gamma(u)$. In addition, the source node of $\mathcal{D}_S$, $\gamma(v) = [\varepsilon]_S$, is also a parent of $\gamma(u)$ since $\gamma(v) = [\varepsilon]_S$ and $\gamma(u) = [z]_S = [a]_S$. □

Based on the above lemmas, the algorithms in Listing 2.3 can list all parents of u in $\mathcal{D}_S$. In the operation Parent-Count(u), line 6 corresponds to the case in Lemma 2.5, and line 8-9 corresponds to the case in Lemma 2.4. In the operation Extract-Parent(u, i), since the last node in the list is always an ancestor of the first node $b =$ suffix-link(u), the interested node is the i-th parent of b in $\mathcal{T}_S$. The operation level-ancestor (in Lemma 2.1) is used to compute the answer.

In summary, we have the following theorem:

Theorem 2.6. *Given a sequence S, there is a data structure to simulate the DAWG $\mathcal{D}_S$ that uses $n(H_k(\overline{S}) + 2H_0^*(\mathcal{T}_{\overline{S}})) + o(n)$. It supports* Get-Source(), Find-Child(u, c), Parent-Count(u), Extract-Parent(u, i), End-Set-Count(u) *in $O(1)$ time and support* Extract-End-Set(u, i) *in $O(\log n)$ time.*

2.4 Application of DAWG in Local alignment

This section studies the local alignment problem. Consider a database S of length n. For any string P of length m, our aim is to compute the best local alignment between P and S. By indexing the database S using an $O(n)$-bit FM-index data-structure, Lam et al. [70] showed that under a scoring function similar to BLAST, the best local alignment between any query pattern P and S can be computed using $O(n^{0.628}m)$ expected time in random input and $O(nm^2)$ worst case time. Their worst case time happens when P is long and occurs inside S.

In this work, we show that, by replacing the FM-index data-structure by the $O(n)$-bit compressed DAWG, we can narrow down the gap between the worst case and the expected case. Thus, improve the running time when there are many positive matches. Specifically, the worst case time can be improved from $O(nm^2)$ to $O(mn)$ while the expected running time in random input remains the same.

2.4.1 Definitions of global, local, and meaningful alignments

Let X and Y be two strings in Σ^*. A space "$-$" is a special character that is not in these two strings. An alignment A of X and Y are two equal length strings X' and Y' that may contain spaces, such that (i) removing spaces from X' and Y' will get back X and Y, respectively; and (ii) for any i, $X'[i]$ and $Y'[i]$ cannot be both spaces.

For every i, the pair of characters $X'[i]$ and $Y'[i]$ is called an indel if one of them is the space character, a match if they are the same, and a mismatch otherwise. The alignment score of an alignment A equals $\sum_i \delta(X'[i], Y'[i])$, where δ is a scoring scheme defined over the character pairs.

Let S be a string of n characters and P be a pattern of m characters. Below, we define the global alignment problem and the local alignment problem.

- The *global alignment* problem is to find an alignment A between S and P which maximizes A's alignment score with respect to a scoring scheme δ. Such score is denoted as global-score(S, P).

- The *local alignment* problem is to find an alignment A between any substring of S and any substring of P which maximizes A's alignment score. Such score is denoted

as local-score(S, P). Precisely, local-score$(S, P) = \max\{$global-score$(S[h..i], P[k..j]) \mid 1 \leq h \leq i \leq n, 1 \leq k \leq j \leq m\}$.

In practical situations, people use alignment to find string similarity; therefore, they are only interested in alignment which has enough matches (e.g. more than 50% of the positions are matches). In [70], the meaningful alignment is defined as follows:

- Consider a scoring scheme δ where mismatches and indels have negative score. Let $A = (X', Y')$ be an alignment of two strings X and Y. A is called a meaningful alignment if and only if the alignment scores of all the non-empty prefixes of the aligned strings X' and Y' is greater than zero, i.e., global-score$(X'[1..i], Y'[1..i]) > 0$ for all $i = 1, \ldots, |X'|$. Otherwise, A is said to be meaningless.

Note that from this point, we only consider scoring scheme where mismatch and indel have negative scores. And, we only consider local alignment score which is greater than or equal to zero.

Consider two strings S and P, we define meaningful-score(S, P) as the best meaningful alignment score between S and P if one exists. If it does not exist, meaningful-score(S, P) is $-\infty$. Authors in [70] showed the following relationship between local alignment and meaningful alignment:

Lemma 2.1. *(Lam et al. [70]) We have*

$$\text{local-score}(S, P) = \max_{1 \leq h \leq i \leq n, 1 \leq k \leq j \leq m} \text{meaningful-score}(S[h..i], P[k..j])$$

2.4.2 Local alignment using DAWG

Consider a database S and a pattern P. Let $\mathcal{D}_S = (V, E)$ be the DAWG of S (i.e. the DAWG of the concatenation of all strings in S separated by \$). This section derives a dynamic programming solution to compute local-score(P, S).

Recall that each node $u \in V$ represents the set of path labels of all possible paths from the source node to u. We say a string $x \in u$, if x is a path label of a path from the source node to u. Note that these sets form a partition of all substrings in S.

First, we define a recursive formula. For every $j \leq |P|$, for every node $u \in \mathcal{D}_S$, we denote $N_j[u] = \max_{k \leq j, y \in u}$ meaningful-score$(P[k..j], y)$. Below lemma states the recursive formula for computing $N_j[v]$.

Lemma 2.2. *The meaningful alignment score $N_j[u]$ defined above satisfies the following recursive formula:*

$$N_j[\varepsilon] = 0 \qquad \forall j = 0..m$$
$$N_0[u] = -\infty \qquad \forall u \in V - \{\varepsilon\}$$

$$N_j[u] = \text{filter}\left(\max_{(v,u)\in E}\begin{cases} N_{j-1}[v] + \delta(P[j], c_{(v,u)}) & (Case\ A) \\ N_{j-1}[u] + \delta(P[j], -) & (Case\ B) \\ N_j[v] + \delta(-, c_{(v,u)}) & (Case\ C) \end{cases}\right) \tag{2.1}$$

where $\text{filter}(x) = x$ *if* $x > 0$*; and* $-\infty$*, otherwise.*

Proof. Let $score(x, y)$ be the short name for meaningful-score(x, y).

Proof by induction: The base case where $u = \varepsilon$ or $j = 0$ is obviously hold. Given any topological order $\pi = \pi_1\pi_2 \ldots \pi_k$ of the nodes of $\mathcal{D}_S$ (note that $\pi_1 = [\varepsilon]_S$), assume $N_j[u]$ satisfies the recursive relation for all $j \le l$ and $u = \pi_1, \ldots, \pi_{i-1}, \pi_i$ except $N_l[\pi_i]$. Below, we show that the following equation is correct for $j = l$ and $u = \pi_i$, that is:

$$\text{filter}\left(\max_{(v,\pi_i)\in E}\begin{cases} N_{l-1}[v] + \delta(P[l], c_{(v,\pi_i)}) \\ N_{l-1}[\pi_i] + \delta(P[l], -) \\ N_l[v] + \delta(-, c_{(v,\pi_i)}) \end{cases}\right) = \max_{x=P[k..l], y\in\pi_i, k\le l} score(x, y) \tag{2.2}$$

where $\text{filter}(x) = x$ if $x > 0$; and $-\infty$, otherwise

We will prove both $LHS \le RHS$ and $LHS \ge RHS$.

($LHS \le RHS$) Let $A = N_{l-1}[v] + \delta(P[l], c_{(v,\pi_i)})$, $B = N_{l-1}[\pi_i] + \delta(P[l], -)$ and $C = N_l[v] + \delta(-, c_{(v,\pi_i)})$. Note that $\text{filter}(\max_{(v,\pi_i)\in E}\{A, B, C\}) = \max_{(v,\pi_i)\in E}\{\text{filter}(A), \text{filter}(B), \text{filter}(C)\}$. If any of A, B or C is not positive, after applying filter it becomes $-\infty$. Then, we do not need to care about that term any more.

Consider $A = N_{l-1}[v] + \delta(P[l], c_{(v,\pi_i)})$. If A is positive, base on the inductive assumption, we have $N_{l-1}[v] = \max_{x=P[k..l-1], y\in v, k\le l-1} score(x, y)$. Let $(X_1, Y_1) = \arg\max_{x=P[k..l-1], y\in v, k\le l-1} score(x, y)$. Consider a string $X_a = X_1 \cdot P[l]$ and $Y_a =$

$Y_1 \cdot c_{(v,\pi_i)}$. One of the alignment of X_a and Y_a can be found by taking the alignment of X_1 and Y_1 and respectively adding $P[l]$ and $c_{(v,\pi_i)}$ at each end of the string. Therefore, $A \leq score(X_a, Y_a)$. (In fact, we can prove that $A = score(X_a, Y_a)$, but it is not necessary.) As X_a is a substring of P ending at l and Y_a is a string in π_i, this means $\text{filter}(N_{l-1}[v] + \delta(P[l], c_{(v,\pi_i)})) \leq score(X_a, Y_a) \leq RHS$.

Consider $B = N_{l-1}[v] + \delta(P[l], -)$, similar to the previous case for A, let $(X_2, Y_2) = \arg\max_{x=P[k..l-1], y\in\pi_i, k\leq l-1} score(x, y)$, then choose $X_b = X_2 \cdot P[l]$ and $Y_b = Y_2$. For $C = N_l[v] + \delta(-, c_{(v,\pi_i)})$, let $(X_3, Y_3) = \arg\max_{x=P[k..l], y\in v, k\leq l} score(x, y)$, choose $X_c = X_3$ and $Y_c = Y_3 \cdot c_{(v,\pi_i)}$. We both have $\text{filter}(B) \leq score(X_b, Y_b)$ and $\text{filter}(C) \leq score(X_c, Y_c)$. Therefore, we have $\max\{\text{filter}(A), \text{filter}(B), \text{filter}(C)\} \leq \max\{score(X_a, Y_a), score(X_b, Y_b), score(X_c, Y_c)\} \leq RHS$. That implies $LHS \leq RHS$.

($LHS \geq RHS$) By definition, meaningful score is either a positive number or $-\infty$.

If RHS is $-\infty$, this implies no meaningful alignment exists between any substring of P ends at j and any substring of S represented by a node π_i. Obviously, $LHS \geq RHS$ is still correct.

If RHS is a positive number, let $(X, Y) = \arg\max_{x=P[k..l], y\in\pi_i, k\leq l} score(x, y)$. X should equal to a substring of P which ends at l, and Y should equal to a substring of S represented by a node u in $\mathcal{D}_S$. Let (X', Y') be the best alignment of (X, Y). Let a, b be the last character of X' and Y', respectively. There are three cases for a and b: (i) $a, b \in \Sigma$, (ii) $a \in \Sigma$ and $b = -$, (iii) $a = -$ and $b \in \Sigma$.

In case (i), the last characters of X, Y are respectively a and b . Let X_m and Y_m be the strings obtained by removing the last character from X and Y, respectively. X_m should equal to a substring ends at l; and Y_m should equal to a path label of a parent node of π_i. In this case, we have $score(X_m, Y_m) \geq score(X, Y) - \delta(a, b)$. As, $N_{l-1}[v] = \max_{x=P[k..l-1], y\in v} score(x, y)$, $N_{l-1}[v] \geq score(X_m, Y_m)$. Hence, $LHS \geq score(X_m, Y_m) + \delta(a, b) \geq score(X, Y)$. Similarly, we can also prove $LHS \geq score(X, Y)$ in cases (ii) and (iii). □

By Lemma 2.1, we have $\text{local-score}(P, S) = \max_{j=1..|P|, u\in\mathcal{D}_S} N_j[u]$. Using the recursive equation in Lemma 2.2, we obtain the dynamic programming algorithm in Listing 2.5. Below two lemmas analyse the time and space complexity of the algorithm.

Lemma 2.3. *Let $m = |P|$ and $n = |S|$.* $\text{local-score}(P, S)$ *can be computed in $O(mn)$*

```
1 Initialize N_0[u] for all u
2 for (j = 1 to m)
3     /* using formula from Lemma 2.2 */
4     foreach (positive entry N_{j-1}[v] and edge (v,u))
5         Update N_j[u] = max{N_j[u], N_{j-1}[v] + δ(P[j], c_(v,u))} (Case A)
6         Update N_j[v] = max{N_j[v], N_{j-1}[v] + δ(P[j], −)} (Case B)
7     foreach (positive entry N_j[v] in any topo. order of v and edge (v,u))
8         Update N_j[u] = max{N_j[u], N_j[v] + δ(−, c_(v,u))} (Case C)
```

Listing 2.5: Complete algorithm

worst case time using $O(n \log n)$ worst case bits memory.

Proof. The number of entries in the array $N_j[u]$ is $O(mn)$. Note that in the recursive formula, for each j, each edge (v, u) of the graph $\mathcal{D}_S$ is visited once. Since there are only $O(n)$ nodes and edges in $\mathcal{D}_S$ (Theorem 2.5), the worst case running time is $O(mn)$.

For every node u, the entries $N_j[u]$ only depend on $N_{j-1}[u]$. Therefore, after $N_j[u]$ has been computed, the memory for $N_{j-2}[u]$ down to $N_0[u]$ can be freed. Thus, the maximal required memory is $O(n \log n)$ bits. □

The following lemma gives some analysis on the average case behaviour of the algorithm to compute local alignment using the formula in Lemma 2.2.

Lemma 2.4. *The expected running time and memory to find the meaningful alignment using DAWG is bounded by the expected number of distinct substrings in S and substrings in P in which meaningful alignment score is greater than zero.*

Proof. Each entry $N_j[u]$ is computed from positive entries among $(N_{j-1}[v_1],\ldots,N_{j-1}[v_k])$, $(N_j[v_1],\ldots, N_j[v_k])$ and $N_{j-1}[u]$ where $(v_1, u), \ldots, (v_k, u)$ are edges in $\mathcal{D}_S$. Therefore, the expected running time and memory is in the order of the number of positive entries in N and the number of visited edges (v, u). Since, any node v in $\mathcal{D}_S$ has at most $|\Sigma|$ out-going edges (one for each character in Σ). The number of visited edges is proportional to the number of positive entries.

Consider a positive positive entry $N_j[u] = \max_{k \le j, y \in u}$ meaningful-score$(P[k..j], y)$. It is obviously that each positive entry corresponds to distinct substring y in S and a substring x in P in which meaningful alignment score is greater than zero. □

From the above lemma, the problem of estimating the average running time becomes the problem of estimating the number of substring pairs which have positive meaningful score. We do not notice any direct result on this bound; however, there are a few results

on measuring the average number of pairs of strings which have Hamming distance within certain bound.

For example, Baeza-Yates [8] analysed the all-against-all alignment problem (set of strings against themselves) on suffix tree. The core of the analysis is to measure the average number of comparisons for searching a random string over a trie allowing errors. This yields an $O(n^{\alpha} m \log n)$ bound on our problem where α is a constant which is less than one. Maaß [74] analysed the time for searching a pattern on a trie of n random strings allowing at most D Hamming's errors. In the case where D is less than $(\sigma - 1)/\sigma \log_{\sigma} n$ where $\sigma = |\Sigma|$, the average number of comparison is sub-linear ($o(n)$). We can use this result to obtain a bound of sub-quadratic $o(nm)$ on the average case where match is 1 and mismatch is less than or equal to -1. Lam et al. [70] studied a specific case of allowing Hamming errors where match is 1 and mismatch is -3. This score roughly approximates the score used by BLAST. They proved that the running time is bound by $O(n^{0.628} m)$. Their experiments also suggested that in scoring model with gap penalty (gap score is -3), the expected running time is also roughly $O(n^{0.628} m)$.

Lemma 2.5. *The expected running time to find the meaningful alignment using DAWG is at least as good as the expected running time of BWT-SW [70]. (i.e. $O(n^{0.628} m)$ for their alignment score.)*

Proof. In the algorithm BWT-SW, the string S is organized in a suffix tree $\mathcal{T}_S$. The alignment process computes and keeps the meaningful alignment scores between path label of nodes of $\mathcal{T}_S$ and substrings of the pattern string P. Note that each node of the DAWG $\mathcal{D}_S$ can be seen as the combination of multiple nodes of the suffix tree $\mathcal{T}_S$. Therefore, each entry computed in BWT-SW can be mapped to an entry of $N_j[u]$. (Multiple entries in BWT-SW can be mapped to the same entry $N_j[u]$ in our algorithm.) The expected asymptotic running time of our algorithm is thus bounded by that of BWT-SW. □

For simplicity, the above discussion only focuses on computing the maximum alignment score, i.e., the entry $N_j[u]$ which is the maximum. In real-life, we may also want to recover the regions in S containing the alignments represented by $N_j[u]$. In this case, the value of $N_j[u]$ is not enough. We need to compute two more numbers $I_{j,u}$ and $L_{j,u}$ such that meaningful-score$(P[I_{j,u}..j], S') = N_j[u]$ where S' is a length-$L_{j,u}$ substring belongs

to u. Then, using the operations End-Set-Count(u) and Extract-End-Point(u, i), we can enumerate all alignments represented by $N_j[u]$, i.e., $\{(P[I_{j,u}..j], S[q - L_{j,u}..q]) \mid q \in \text{end-set}_S(u)\}$.

$I_{j,u}$ and $L_{j,u}$ can be computed by dynamic programming along with $N_j[u]$. For the base cases, we have $I_{j,\varepsilon}$ equals j and $L_{j,\varepsilon}$ equals -1. Then, depend on the outcome of Equation 2.1, $I_{j,u}$ and $L_{j,u}$ can be updated using the following equations:

$$I_{j,u} = \begin{cases} I_{j-1,v} & \text{if (A) happens} \\ I_{j-1,u} & \text{if (B) happens} \\ I_{j,v} & \text{if (C) happens} \end{cases} \qquad L_{j,u} = \begin{cases} L_{j-1,v} + 1 & \text{if (A) happens} \\ L_{j-1,u} & \text{if (B) happens} \\ I_{j,v} + 1 & \text{if (C) happens} \end{cases}$$

For time and space complexities, note that $L_{j,u}$ and $I_{j,u}$ can be computed using the same time and space complexities as $N_j[u]$. After that, all alignments represented by $N_j[u]$ can be reported using $O(occ \log n)$ time, where occ is the number of such alignments.

2.5 Experiments on local alignment

This section reports experimental results on four local alignment algorithms discussed in the previous sections.

We implemented four local alignment algorithms, namely SeqSeq, TreeSeq, GraphSeq, GraphGraph. The algorithm SeqSeq is derived from Smith-Waterman local alignment. Each dynamic programming cell of this algorithm represents a pair (i, j) where i and j are positions in the text and the pattern sequences, respectively. The TreeSeq algorithm is similar to that implemented in [70] where the text is represented by a suffix trie; each dynamic programming cell represents a pair (v, j) where v is a node in the suffix trie and j is a position in the pattern sequence. The GraphSeq algorithm is based on the algorithm in Section 2.4.2 with each dynamic programming cell represents a pair (v, j) where v is a node in the DAWG and j is a position in the pattern sequence. We also implement an extension GraphGraph where the alignment is computed from a dynamic programming formula in the cross product of the DAWGs of both the text and the pattern. Each dynamic programming cell represents a pair (u, v) where u and v are nodes in the two DAWGs.

The first set of experiments shows the performance of the four algorithms when the

sequences are randomly generated. We keep the pattern length fixed and change the text length. Figure 2.3(a) shows the running time of each algorithm. Figure 2.3(b) shows the number of dynamic programming cells that are created and accessed by each algorithm. From these two figures, the GraphGraph algorithm is clearly the slowest, although the GraphGraph algorithm created and accessed the least number of dynamic programming cells. This should be due the high overhead of evaluating each dynamic programming cell. Similar phenomenon occurs the all other experiment, therefore, the subsequence results we will exclude the GraphGraph algorithm. Figure 2.3(c) ignores the running time of the GraphGraph algorithm. It shows that the GraphSeq algorithm is slightly slower than the TreeSeq algorithm, although it used slightly less number of cells.

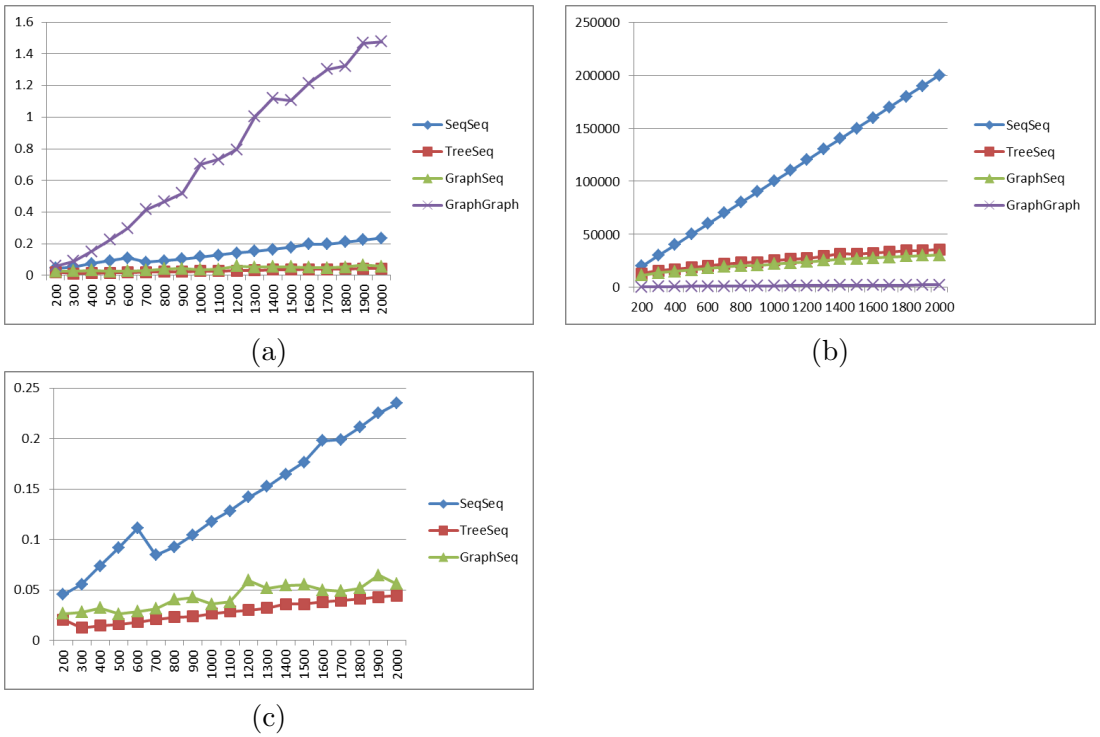
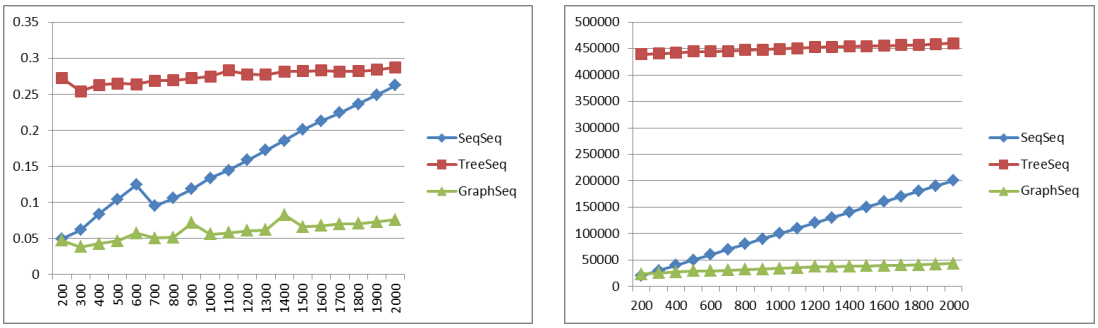


Figure 2.3: The performance of four local alignment algorithms. The pattern length is fixed at 100 and the text length changes from 200 to 2000 in the X-axis. In (a) and (c), the Y-axis measures the running time. In (b) and (d), the Y-axis counts the number of dynamic programming cells created and accessed.

The next set of experiments (shown in Figure 2.4) are similar to the first one, except the pattern is a substring of the text. From the running time and number of dynamic programming cells accessed. All the algorithms take more time to run in this setting. However, the TreeSeq algorithm becomes significant slower. This observation agrees with the worst case analysis of the TreeSeq algorithm.

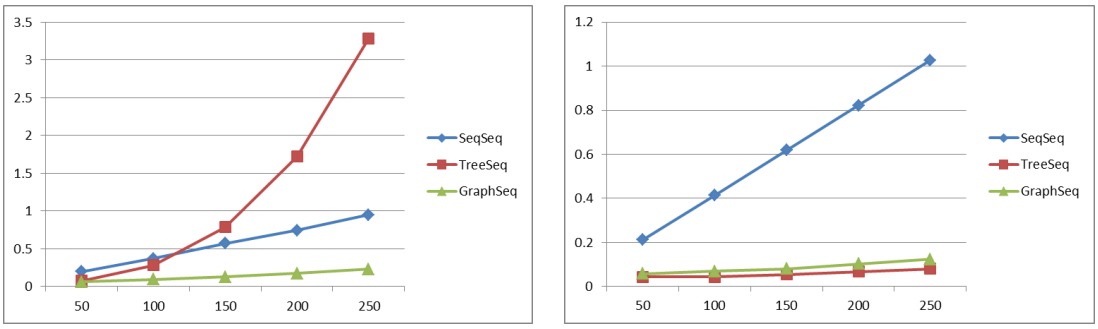
To illustrate the worst case of TreeSeq, we perform another experiment in Figure 2.5.



(a) (b)

Figure 2.4: The performance of three local alignment algorithms when the pattern is a substring of the text. (a) the running time (b) the number of dynamic programming cells.

In this experiment, the text is fixed and the pattern length is varied. Figure 2.5(a) shows the running times when the pattern is part of the text. Figure 2.5(b) shows the running time when two sequences are independent. The experiment shows that the algorithm TreeSeq is in quadratic time to the pattern length when it similar to the part of the text as predicted. The GraphSeq algorithm does not affected significantly in both cases.



(a) (b)

Figure 2.5: Measure running time of 3 algorithms when text length is fixed at 2000. The X-axis shows the pattern length. (a) The pattern is a substring of the text. (b) Two sequences are totally random.

In conclusion, the number of cells in the dynamic programming which used to bound the running time in the theoretical work does correlated with the running time. However, the constant factor also plays significant factor when compare between different algorithms. The running time of TreeSeq algorithm shows quadratic behaviour when the pattern is a substring of the text. The GraphSeq algorithm shows more consistent running time.

Chapter 3

Multi-version FM-index

3.1 Introduction

The amazing advances in biology, chemistry and engineering have made the DNA sequencing become cheaper and faster. It promises many more exciting discovery in living mechanism as well as personal medicine. However, the huge amount of data produced by sequencing technology also poses new challenges to bioinformatic research to find efficient way to to compress the generated sequence data while still providing basic function like substring search (a.k.a indexing).

Due to evolution, the DNA sequences of individuals in the same species and DNA sequences of related species are highly similar. Therefore, their storage can be greatly reduced by clever compression. However, well-established compression methods developed during the 80's and 90's designed for text and media file do not solve the storing problem for genomic data. First, well known compression programs like zip, bzip2 often compress data locally by dividing the data into blocks, or searching for repetitions within some window around the current encoding point. In genomic data, this strategy is often not very effective. At local level (e.g. a few thousand bases) the DNA sequence is very close to uniformly random, however, at larger scale, (e.g. between two human genomes of billions bases) the difference may be as low as 1%. Increasing the block or windows size to a few gigabytes may help to achieve better the compression, but it makes searching and accessing the sequences more difficult.

Recent research in indexing data structures has started to devote more attention to this type of similar sequences, and proposed a few compression structures [79, 27] to index

the sequences. In this work, we also set out to find some solution for problem of indexing similar sequences, but using different compression approach. An obvious compression scheme that can effectively compress the type of sequences is delta compression. This scheme stores the first sequence and the changes between each pair of sequences. However, using this scheme, many types of queries cannot be supported easily.

We first model the changes in the similar sequences by keeping only the inserted and deleted characters between each pair of sequences. Then, we design a data structure called multi-version rank and select to support the rank and select queries in the delta compressed sequences. Based on this result, we propose an index data structure for similar sequences called multi-version FM-index which can find occurrences of a pattern P for any sequence S_i.

Related methods

Our approach was inspired by the persistent data structures which are popular in logical and functional programming[90]. In this approach, data structure always preserves the previous version of itself when it is changed. A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The data structure is fully persistent, if every version can be both accessed and modified. Applying this idea into compression, we store a set of versions, but do not allow changes to them to improve space requirement and query time. Therefore, we called our data structure multi-version rather persistent.

There are two main branches in persistent data structure research [63]. The first branch focuses on transforming any dynamic data structure into persistent data structure with low space and time overhead. The second direction focuses on designing persistent data structure for specific representation e.g. lists, search tree.

In the transformation direction, a persistent data structure for rank and select can be constructed by taking a dynamic rank and select structure, and converting it into persistent version. The state of the art dynamic rank and select structure [50] uses $nH_0(S) + o(n \log \sigma)$ bits, and serves the queries and updates in $t = O(\log n(1 + \log \sigma / \log \log n))$ time where S is a string of length n, and H_0 is the zero-order entropy. Let m be the total number of changes in the inputs, using the transformation in [26], the data structure will slow down by a factor of $\log \log(mt) \approx \log \log(m \log n)$ time, and

blow up the space by adding $mt \log n \approx m \log^2 n$ bits. This transformation is not time and space efficient compared to our proposal.

Our work on multi-version rank and select is built upon the works in specific persistent data structure for set proposed by Dobkin and Munro[28], later improved by Overmars[91]. Section 3.3 can be seen as a generalized and improved version of Overmars' structure[91]. The previous data structure uses $O(N \log^2 N)$ bits and answers query in $O(\log N)$ time, while ours uses $O(m \log N)$ bits and answers queries in $O(\log m / \log \log m)$ time where $N = n + m$.

3.2 Multi-version rank and select problem

Given a sequence $S[1..n]$ over an alphabet Σ and $i \leq n+1$, we define two edit operations *insert* and *delete* as follows: (i) $insert(S, c, i)$ returns a string S' such that $S' = S[1..i-1]cS[i..n]$ where c is a character in Σ; (ii) $delete(S, i)$ returns S' such that $S' = S[1..i-1]S[i+1..n]$. Note that another common edit operation called replacement which changes one character of a sequence by another character can be simulated by one *delete* and one *insert* operation.

Consider a set $\mathcal{S}$ of m sequences $\{S_1, S_2, \ldots S_m\}$, these sequences are called *multi-version sequences*, if, for $i = 1, \ldots, m-1$, the sequence S_{i+1} can be obtained from the sequence S_i by either an insertion or a deletion operation ed_i. Multi-version rank and select data structure supports two queries $rank(S_i, c, j)$ and $select(S_i, c, k)$ for all $S_i \in \mathcal{S}$. Query $rank(S_i, c, j)$ counts the number of occurrences of character c in the substring $S_i[1..j]$. Query $select(S_i, c, k)$ returns the position of the k-th occurrence of character c in S_i. Our result is summarized as follows:

Let S be a sequence that equals S_1 concatenates with all the inserted characters from the editing operations. Let n be the length of S_1, and m be the number of edit operations; and σ be the size of the alphabet. Let $w = \Omega(\log n)$ be the word size of the machine.

Theorem 3.1. *There exists a multi-version rank and select data structures that uses* $|S|H_k(S) + 2m \log(m+n) + o(|S| \log \sigma + m \log(m+n))$ *bits and answers the queries in* $O(\log \frac{\log \sigma}{\log w} + \log m / \log \log m)$ *time.*

From the definition, one straightforward scheme to represent the sequences is to store

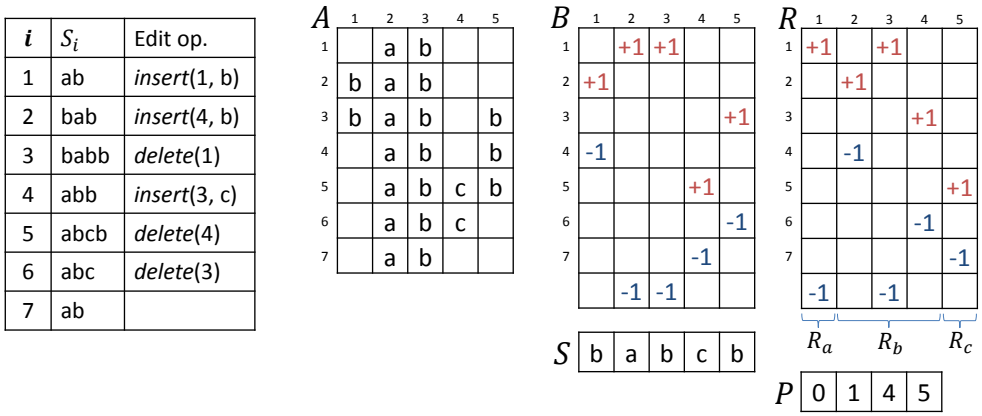


Figure 3.1: (a) Sequences and edit operations (b) Alignment (c) Balance matrices

$\mathcal{S}$ explicitly. In this way, queries *rank* and *select* can be supported efficiently. However, this representation wastes a lot of space. For example, if the length of the sequence S_1 is n, this scheme requires about $\Omega(n \times m \log |\Sigma|)$ bits. One space economical scheme is to store S_1 and all editing operations. However, if the edit operations are stored in a trivial form like $insert(i, j, c)$ and $delete(i, j)$, it is hard to answer the query efficiently.

In our approach, the data is stored in an intermediate form called alignment. The next sub-section defines the alignment and the transformation from the input of sequences and edit operations into the alignment form. The rest of the section describes the data structure and the query algorithms based on the alignment.

3.2.1 Alignment

Consider an alphabet Σ and a space character $-$ which does not appear in Σ. Given a set of multi-version sequences $\mathcal{S} = \{S_1, \ldots, S_m\}$ over the alphabet Σ, an alignment of these sequences is a matrix A that has m rows, and contains characters in $\Sigma \cup \{-\}$. The alignment A also needs to satisfy the following conditions: (1) S_i equals row i of A after removing spaces. (2) each column of the matrix A has exactly one type of non-space characters (3) the non-space characters are consecutive in each column. (See Fig. 3.1 for an example.)

From the definition, an alignment can be represented as a set of columns $\{(c_j, s_j, e_j)\}$ where c_j is the non-space character in column j, s_j and e_j are, respectively, the starting and ending rows of character c_i in the alignment. In the alignment, we call columns that does not have space as *trivial* columns. An alignment for multi-version sequences $\mathcal{S}$ in the column representation can be found as following:

Lemma 3.2. *Consider a set of multi-version sequences $\mathcal{S} = \{S_1, S_2, \ldots S_m\}$ with the edit operations $\{ed_i\}$. Let n be $|S_1|$. Let m_i and m_d be the number of insertions and deletions respectively (i.e. $m = m_i + m_d$). There exists an alignment A for $\mathcal{S}$ that contains m rows and $n + m_i$ columns. Furthermore, the matrix A contains between $n - m_d$ and n trivial columns. We can find the alignment A for $\mathcal{S}$ in $O(n + m \log(m + n))$ time.*

Proof. The algorithm will incrementally build the alignment A_k of $\{S_1, \ldots, S_k\}$ for $k = 1, \ldots, m$. We represent A_k as follows. Suppose A_k has n' columns and the i-th column is represented as (c_i, s_i, e_i) where s_i is the insertion time and e_i is the deletion time. Note that, in the implementation, we set $e_i = \infty$ if it has not deleted. We represents A_i as a modified B-tree with the i-th leaf equals (c_i, s_i, e_i). Furthermore, for each internal node u, it maintains two sets of keys: (1) the number of leaves of every subtrees attached to u, (2) the size of the set $\{(c_i, s_i, e_i) \in S' \mid e_i = \infty\}$ for every subtree S' attached to u. Given the two sets of keys per internal node, we can find the leaf for the i-th column of A_k and for the column represents $S_k[i]$ in $O(\log n')$ time. In addition, insertion into the B-tree also take $O(\log n')$ time.

Initially, A_1 is represented as a B-tree where the i-th leaf equals $(S_1[i], 1, \infty)$ for $i = 1, \ldots, |S_1|$. It can be build in $O(|S_1|) = O(n)$ time.

Then, we incrementally construct A_k from A_{k-1} for $k = 2, \ldots, m$. There are two cases depending whether it is insertion or deletion.

Case 1: ed_k is an insertion of a character c at position j. We first identify the leaf x in the B-tree for A_{k-1} that represents $S_{k-1}[j]$; then, we insert (c, k, ∞) just before the leaf x.

Case 2: ed_k is a deletion of $S_{k-1}[j]$. We first identify the leaf $x = (c_j, s_j, e_j)$ in the B-tree for A_k that represents $S_{k-1}[j]$; then, we replace x by (c_j, s_j, k).

For both cases, it takes $O(\log n')$ time. □

Alignment representation

Storing an alignment of multi-version sequences $\mathcal{S} = \{S_1, \ldots, S_m\}$ in the column representation of $\{(c_i, s_i, e_i)\}$ takes $2N \log m + N \log \sigma$ bits where N is the number of columns in the alignment, m is the number of changes and σ is the size of the alphabet. However, supporting query in the plain column representation is not efficient. Instead, we represent an alignment in the following matrix form.

Let M be a matrix of size $m \times n$. Denote $M[i..i', j..j']$ as a sub-matrix of M taken the region $[i..i'] \times [j..j']$ from M. Denote $M[i, j..j']$ as sub-column-vector of M, and $M[i..i', j]$ as a sub-row-vector of M.

Given an alignment $A[1..m, 1..N]$ which is represented by a set of columns $\{(c_j, s_j, e_j)\}$, let S be the concatenation of all characters c_j. Let $B[1..m+1, 1..N]$ be a matrix such that $B[s_j, j]$ is set to 1 and $B[e_j + 1, j]$ is set to -1, the other values of B are zero. (See Fig. 3.1) Storing the alignment A is equivalent to storing the sequence S and matrix B.

We are interested in two operation in B: operation $sum(B, i, j)$ returns the value of $\sum_{i'=1}^{i} \sum_{j'=1}^{j} B[i', j']$; operation $select\_sum(B, i, k)$ finds the smallest value j such that $sum(B, i, j) \geq k$. The relation between the alignment A and the matrix B is shown in the following lemma:

Lemma 3.3. *The operation $sum(B, i, j)$ returns the number of non-space characters in the prefix j of row i in A (i.e. $A[i, 1..j]$). The operations $select\_sum(B, i, k)$ finds the column of the k-th non-space character in row i of the alignment A.*

Proof. For the first property, we proof it by induction on each column. Assume the property holds for $A[i, 1..j-1]$. We have $sum(B, i, j) = sum(B, i, j-1) + \sum_{i'=1}^{i} B[i', j]$. Consider only column j, if $A[i, j]$ is a non-space character, then $s_j \leq i$ and $e_j \leq i$. By the definition, we have $B[s_j, j] = 1$, therefore $\sum_{i'=1}^{i} B[i', j] = 1$. If $A[i, j]$ is a space character, we have either $s_j \leq e_j < i$ or $i < s_i \leq e_j$. In both cases, $\sum_{i'=1}^{i} B[i', j] = 0$. Therefore, $sum(B, i, j) = sum(B, i, j-1) + 1$ if $A[i, j]$ is non-space, and $sum(B, i, j) = sum(B, i, j-1)$, otherwise. That concludes the induction.

The second property follows directly from the first property and the definition of *select_sum*. Let $j = sum(B, i, k)$. By the definition, we have $sum(B, i, j) = k$ and $A[i, j]$ is a non-space character. □

Matrix B is a sparse matrix, therefore, it can be stored succinctly. It satisfies the following properties: (1) every row except the first and the last row has only one non-zero number, (2) each column has two non-zero numbers where number 1 is placed before number -1. The columns of B that correspond to trivial columns of A are also called *trivial*. (i.e. these columns start with 1 and end with -1.) We call these type of matrices and its generalized form (defined later) as *balance matrix*. Section 3.3 details our strategy to store balance matrix and to support queries *sum* and *sum* and *select_sum* efficiently.

The result is summarized as follows:

Theorem 3.4. *Consider a matrix B that satisfies the two conditions above, has n trivial columns and m other columns, there exists a data structure that uses $m\log(m+n) + o(m\log(m+n) + n + m)$ bits, supports the queries sum and select_sum on B using $O(\log m/\log\log m)$ time.*

Proof. See Section 3.3. □

3.2.2 Data structure for multi-version rank and select

Using the alignment representation in the previous section. We can construct data structure for multi-version rank and select as follows:

Consider the set of multi-version sequences $\mathcal{S}$, let matrix B and sequence S be the representation of the alignment of $\mathcal{S}$. Assume each character $c \in \Sigma$ is presented as a number in the range from 1 to σ. Let $P[0..\sigma]$ be an array such that $P[0] = 0$ and $P[c]$ stores the number of characters in S which is smaller than or equal to character c. For each character $c \in \Sigma$, let R_c be a matrix such that R_c is obtained from B by removing all columns j such that $S[j] \neq c$. Let R be the concatenation of the columns of R_c in alphabetical order of character c. (i.e. columns $P[c-1]+1$ to $P[c]$ of R equals R_c.) Note that, R can also be viewed as a column permutation of B. Therefore, R has the same properties as B, and both the operation *sum* and *select_sum* are applicable to R. (See an example in Fig. 3.1)

The data structure for multi-version sequence rank and select problem consists of:

- General static rank and select data structure for S, using $|S|H_k(S) + o(n\log\sigma)$ bits.
- Prefix sum data structure for array P, using $\sigma\log|S|$ bits which is $o(|S|H_0(S))$ bits.
- Data structures for matrices B and R to support $sum(A)$, $select\_sum(A)$, $sum(R)$ and $select\_sum(R)$ (detailed in Section 3.3). These data structure uses $2m\log(m+n) + o(m\log(m+n))$ bits.

Summing up the space for all components of the data structure, we can get the space claim in Theorem 3.1. The next subsection shows how to implement query $rank(S_i, i, j)$

and $select(S_i, i, k)$ using constant number of operations on S, P, R and A. Since the operations in R and A requires $O(\log m / \log\log m)$ time (detailed Section 3.3). The operations on S requires $O(\log \frac{\log \sigma}{\log\log n})$ time (see Section 1.2.2). The running time claim in Theorem 3.1 follows.

3.2.3 Query algorithms

Note that in the data structure specification, we store only R, although, the following proofs only uses the concept R_c. Because storing the concatenation of R_c uses less space than storing multiple separate data structure R_c for each character c. (e.g. save space for pointers, save space for vertical coordinate scaling). The following lemma shows how to simulate operations on R_c using matrix R and array P.

Lemma 3.5. *We can simulate operations on R_c by using R and P as follows:*

$$sum(R_c, i, j) = sum(R, i, j + P[c-1]) - sum(R, i, P[c-1])$$
$$select\_sum(R_c, i, k) = select\_sum(R, i, k + sum(R, i, P[c-1])) - P[c-1]$$

Proof. Since R_c is concatenated in alphabetical order, therefore, the first column of R_c is the $(P[c-1]+1)$ column inside R. The j-th column of R_c is $(P[c-1]+j)$ column in R. The number of non-space characters of row i in alignment A from column $P[c-1]+1$ to column $P[c-1]+j$ equals the RHS of the first equation. Since these columns are mapped to the columns 1 to j of R_c, we have the first equation holds.

For the second equation, let k' be the number of non-space characters of row $R[i]$ from column 1 to column $P[c-1]$. The k-th non-space character of $R_c[i]$ is the $k+k'$ non-space character of R. Thus, we can use *select_cut* on R to find the k-th non-space character on $R_c[i]$. However, we need to map back the result to R_c. Thus, we arrive at the second equation. □

Recall that, the operation $rank(S_i, c, j)$ counts the number of character c in $S_i[1..j]$. Our computation of $rank(S_i, c, j)$ is based on the transformation from sequence S_i to the row i of the alignment A, and then to matrix R_c. Intuitively, it is easier to do the *rank* counting in matrix R_c, since R_c contains only columns of character c. In this approach, the problem becomes how to keep track of the parameter of position j during the transformation. Precisely, the operation $rank(S_i, c, j)$ can be computed as follows:

Lemma 3.6. *For any $i = 1..m$, $j = 1..|S_i|$ and $c \in \Sigma$:*

$$rank(S_i, c, j) = sum(R_c, i, rank(S, c, select\_sum(B, i, j))) \tag{3.1}$$

Proof. Denote A_i as the row i of alignment A. We break down the equation into two parts:

$$rank(S_i, c, j) = rank(A_i, c, select\_sum(B, i, j))$$
$$rank(A_i, c, j) = sum(R_c, i, rank(S, c, j))$$

For the first equation, S_i equals A_i after removing all spaces. Let $k = select\_sum(B, i, j)$. By definition, the k-th non-space character in A_i is the j-th character in S_i. Hence, $rank(S_i, c, j) = rank(A_i, c, k)$. The first equation follows.

For the second equation, We also have R_c is B after removing all the columns that do not have character c. Let $k = rank(S, c, j)$. By definition, the k-th column in R_c is mapped to the j-th column of B. Moreover, since R_c only corresponds to non-space character c, counting character c in $R_c[i, 1..k]$ can be done by $sum(R_c, i, k)$. Thus, $rank(A_i, c, j) = sum(R_c, i, k)$. The second equation follows. Combine the two equations we get the lemma. □

Recall that query $select(S_i, c, k)$ returns the position of the k-th occurrence of character c in S_i. Intuitively, the computational process of *select* operation is reversed compared to that of *rank* (in Eq. 3.1). We first find the column of the k-th non-space character in matrix R_c, trace back the position of that column in matrix A, and then trace back the position in S_i. The formula to compute *select* is:

Lemma 3.7. *For any $i = 1..m$, and $c \in \Sigma$:*

$$select(S_i, c, k) = sum(B, i, select(S, c, select\_sum(R_c, i, k)))$$

Proof. We have the following observations:

1. The position of the k-th occurrence of c in S_i is mapped to column j of R_c where $j = select\_sum(R_c, i, k)$.
2. The j column in R_c is column $select(S, c, j)$ in alignment A.

3. The j' column in alignment A is position $sum(B, i, j')$ in S_i.

For the first observation, since in the alignment step only space characters are inserted to get A, the k-th occurrences of c in S_i is still the k-th occurrences of c in $A_i[i]$. Then, R_c is obtained by isolating all c column. The k-th occurrences of c in S_i become the k-th occurrence of c in R_c. Since R_c only contains c column, the column can be found by operation $select\_sum(R_c, i, k)$.

For the second observation, since the columns in R_c keep the same relative order as those in A, the j-th column in R_c is the j-th column that has character c in A. Thus, it is column $select(S, c, j)$ in alignment A.

For the third observation, since A_i removing space becomes S_i, and by the definition of sum, the formula to convert is $sum(B, i, j')$. □

3.3 Data structure for balance matrix

This section develops data structure to support operations sum and $select\_sum$ on the balance matrices defined in the previous section. Recall that, they are matrices of $\{-1, 0, 1\}$ where (1) every row except the first and the last row has one non-zero number, (2) each column has two non-zero numbers in which number 1 is placed before number -1. The columns that start with 1 and end with -1 are called trivial columns. Operation $sum(B, i, j)$ returns $\sum_{i'=1}^{i} \sum_{j'=1}^{j} B[i', j']$. Operation $select\_sum(B, i, k)$ finds the smallest value j such that $sum(B, i, j) \geq k$.

The data structure for these matrices and $select\_sum$ operation is equivalent to the k-th cut line shooting problem which was considered by Overmars [91]. In the shooting problem, given a set of vertical lines, the queries is to find the k-th cut between the lines and a horizontal ray shot from some point from the left. The previous result uses $O(N \log^2 N)$ bits and requires $O(\log N)$ time for query where N is the total number of columns.

Compact matrix representation

To reduce the space and query time when there are many trivial columns, we transform the original balance matrix into a compact form. Given a balance matrix B with the properties above, create a matrix C such that consecutive trivial columns of B are merged

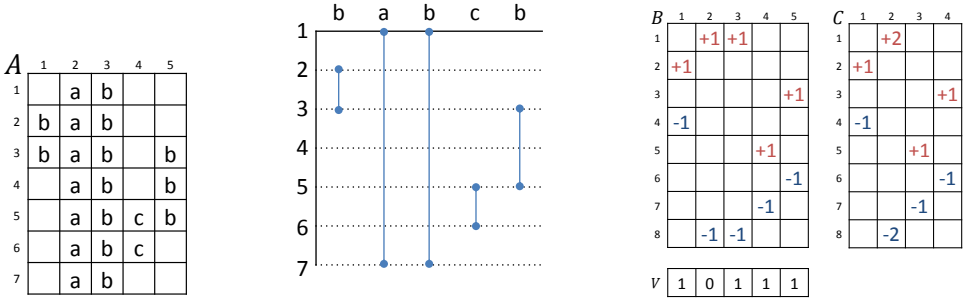


Figure 3.2: (a) Alignment (b) Geometrical form (c) Balance matrix (d) Compact balance matrix

and represented by their sum; and other columns of B are kept (See Fig. 3.2(c) for the original matrix and Fig 3.2(d) for the compact form.)

The reduction from an original balance matrix to its compact form can be summarized in the following lemma.

Lemma 3.1. *Given an original balance matrix B of size $m \times (n+m)$ where n is the number of trivial columns. It can be compacted into a matrix C with of size of size $m \times cm$ where $1/2 \leq c \leq 2$. The operations sum and select_sum on the original matrix B can be simulated by the same operations on C using additional $m \log((n+m)/m) + o(n+m)$ bits.*

Proof. Given a balance matrix B, we use a bit vector V to mark the consecutive trivial columns in M. Set $V[j]$ to 0 if column j and column $j+1$ of B are a trivial columns.

Let $r = rank(V, 1, j)$, we have $sum(B, i, j) = sum(C, i, r) + j - select(V, 1, r)$

Let $j = select\_sum(C, i, k)$, $select\_sum(B, i, k) = select(V, 1, j) + k - sum(C, i, j)$ □

Note that, in a balance matrix, we call the submatrix consists of the rows except the first and the last row as the *body* of the matrix. Since the properties of the first row and the body of a balance matrix are quite different, the body and the first row are usually stored in separated components. However, the operations are still defined in the whole matrix.

The next sub-section gives more details on storing a balance matrix with m rows and the sum of the first row is n in $m \log(m+n) + o(m \log(m+n) + n + m)$ bits, while two operations can be supported in $O(\log m / \log \log m)$ time.

3.3.1 Data structure for balance matrix

In this subsection, we discuss the implementation of operations *sum* and *select_sum* on a compact balance matrix C. Our approach based on two observations. First, when the width of the matrix is small enough, the queries can be answered in constant time. Second, large balance matrix can be hierarchically decomposed into small width matrices to answer the queries. This subsection focuses on the decomposition of the large matrix into small matrices. The results for small width matrix is summarized next, but details are presented in Section 3.4.

Narrow balance matrix

Consider a matrix T, let m be the height of the matrix. T is called *narrow matrix* if and only if the width of the matrix is less than $\log^\epsilon m$ where ϵ is a constant less than $1/2$. Given a narrow balance matrix T, beside the two operations $sum(T, i, j)$ and $select\_sum(T, i, k)$ are defined earilier, we also need an operation called $count\_nzero(T, i, j)$ which counts the number of non-zero numbers in $T[1..i, 1...j]$.

Given access to a narrow balance matrix in certain format, we build auxiliary data structures that supports the queries *sum*, *select_sum* and *count_nzero* in constant time using only the small order number of bits of the original matrix. The format requirement for each part of the balance matrix are:

The body of matrix T which consists of rows 2 to $m-1$ is represented by an array $Y[1..m-2]$ where each entry $Y[r] = T[r+1, c] \times c$ if $T[r+1, c]$ is the only non-empty entry in row $r+1$. We call this representation as row position format. Note that $Y[r]$ is in the range $[-\log^\epsilon m .. \log^\epsilon m]$ and hence, each element $Y[r]$ takes only $\lceil 2\epsilon \log\log m \rceil$ bits. Each chunk of length ρ of Y (i.e. $Y[r..r+\rho-1]$) can be accessed in constant time where $\rho = O(\log m / \log\log m)$.

Let F be the first row of the matrix i.e. $F = T[1, 1.. \log^\epsilon m]$. Let P be the prefix sum array of F i.e. $P[i] = \sum_{i'=1}^{i} F[i']$.

Lemma 3.2. *Given access to array Y by chunk, and access to the prefix sum array P in constant time, there is an auxiliary data structure to support the operations sum, select_sum and count_nzero on the a balance matrix T in constant time that uses extra $o(m \log\log m) + O(\log^\epsilon m \log\log n)$ bits.*

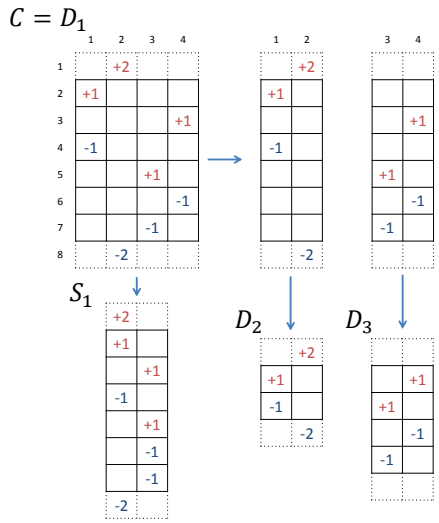


Figure 3.3: Example of the construction steps for $p = 2$. The root node is 1 and two children nodes are 2 and 3. Matrices S_1, D_2, and D_3 are constructed from D_1 as indicated by the arrows.

For a set of same size balance matrices, we have:

Corollary 3.3. *Given q balance matrices $T_1, \ldots, T_q$ with the same size $[1..\ell] \times [1..\log^\epsilon m]$ and $m = q\ell$. Let n be the sum of the values in the first rows of the matrices. Given access to the matrix bodies Y_i and the prefix sum arrays P_i of each matrix T_i, there exists a data structure that supports the queries in each matrix $sum(T_t, i, j)$, $select\_sum(T_t, i, k)$ and $count\_nzero(T_t, i, j)$ in constant time; and uses $o(m \log\log m) + O(q \log^\epsilon m \log\log n)$ bits.*

Data structure construction

Next, the decomposition of a large balance matrix to narrow balance matrices while supporting *sum* and *select_sum* is as follows.

Consider a compact balance matrix C of size $m \times cm$ where c is some constant and $1 \leq c \leq 2$, let $p = \log^\epsilon m$ for some constant $\epsilon < 1/2$. The skeleton of our data structure is a complete p-ary tree T that contains $\lceil cm/p \rceil$ leaves. Intuitively, each node of the tree hierarchically handles a submatrix of C. Each submatrix is vertically partitioned into p smaller roughly equal width sub-matrices, each is managed by one of its children nodes. For each node u of the tree, we store a narrow balance matrix called S_u. Each column of S_u store a summary information for each child of u. (See Fig. 3.3)

To define S_u precisely, we needs two intermediate concepts. First, consider a node u

in T which is the j-th node from left to right at depth d of the tree, the range of node u is $[s_u..e_u]$ where $s_u = (j-1)p^{h-d+1} + 1$, $e_u = \min(jp^{h-d+1}, cm)$ and h is the height of the tree. Conceptually, node u manages the submatrix consisting of columns from s_u to e_u (i.e. $C[1..m, s_u..e_u]$). Second, let's call a row in a matrix that has zero values as empty row. Denote D_u as a balance matrix obtained by removing empty rows from $C[1..m, s_u..e_u]$ except for the first and the last row.

The definition of S_u is as follows. For each leave node l, matrix S_l equals D_l. For each internal node u, the width of matrix S_u is the number of children of node u (which is usually p). Let node v be the j-th child of node u. Entry $S_u[i, j]$ stores the sum of the i-row of D_v.

Space analysis

We store the set of narrow balance matrices $\{S_u\}$ in three separate components: (1) The first rows of the matrices $\{S_u\}$ (2) the bodies of the matrices (the rows from 2 to $m-1$) and (3) the auxiliary data structures for the three operations.

For each matrix S_u, the body matrices are stored explicitly. The size of matrix C is $m \times (cm)$ where $1/2 \leq c \leq 2$. Since the branch out of the tree is p, the height is $O(\log m / \log p)$ which equals $O(\log m / \log \log m)$. Level d of the tree has about $q_i = p^d$ matrices. The sum of the heights of these matrices in each level is m. Therefore, the total size for storing the body matrices of S_u is $m \log m$ bits.

For the first rows, we have the following property: $S_u[1, j] = \sum_{j'=s_v}^{e_v} C[1, j']$ where v is the j-th child of node u. Therefore, given the first row of C, we can represent all the first rows of S_u implicitly by storing only the first row of C. Therefore, size for storing the first rows is $m \log((n+m)/m) + o(n+m)$.

According to Corollary 3.3, all the auxiliary data structures at level d, stored in $o(m \log \log m) + O(p^d \log^{\epsilon} m \log \log n)$ bits. Since there are $\log m / \log p$ levels. The total space for all levels is $o(m \log m) + O(m \log \log n)$ bits. Summing up all the space requirement, we have $m \log(n+m) + o(n + m + m \log(n+m))$ bits.

Query algorithms

Lemma 3.4 shows a recursive equation for $sum(D_u, i, j)$. Since at the root of the tree T, we have $D_u = C$. The operation $sum(C, i, j)$ equals $sum(D_u, i, j)$, thus $sum(C, i, j)$ can

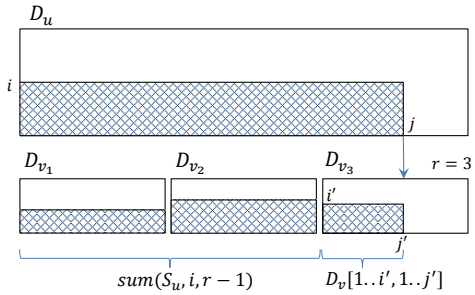


Figure 3.4: Illustration for sum query. The sum for the region $[1..i, 1..j]$ in D_u equals the sums in the three regions in D_{v_1}, D_{v_2} and D_{v_3} respectively.

can be compute recursively using S_u.

Intuitively, the computation process is carried as follows. Recall that, the matrix handled by each node is divided into smaller sub-matrices; each is managed by one of child node. Any summation from the start to a submatrix division boundary can be accessed immediately through S_u. The summation from a division boundary to another column can be computed by recursively call to a child node. (See an illustration in Fig. 3.4)

Lemma 3.4. *For any node u, we have the following recursive relation:*

$$sum(D_u, i, j) = \begin{cases} sum(S_u, i, j) & \text{if } u \text{ is a leaf} \\ sum(S_u, i, r-1) + sum(D_v, i', j') & \text{otherwise} \end{cases}$$

where w is the width of $D_{child(u,1)}$, $r = \lfloor j/w \rfloor$, $j' = j - (r-1)w$, v is r-th child of u, and $i' = count\_nzero(S_u, i, r) - count\_nzero(S_u, i, r-1)$.

Proof. The basis of the equation is correct, since at the leaf, there is no more matrix division. There is no zero row in D_u, therefore, $D_u = S_u$.

Assume the equation is correct up nodes in level $d+1$ of the tree, we show that it is correct in node u in level d. The widths of the submatrices of the children u except that of the last child are all equal. That is w in the equation. Therefore, the submatrix contains column j belongs to the $\lfloor j/w \rfloor$-th children of u. Let v denote that child. The column j in D_u is $j - (r-1)w$ in D_v. Since D_v is obtained by removing zero-rows in the submatrix of C. Row i in D_u becomes row i' in D_v where i' is the number of non-zero elements in $S_u[1..i, \lfloor j/w \rfloor]$. Thus, sum of $D_v[1..i', 1..j']$ matches that of $D_u[1..i, (r-1)w+1..j]$. Since the recursive equation holds for level $l+1$, $D_u[1..i, (r-1)w+1..j]$ can be computed

by the recursive term in the equation.

By definition of S_u, $sum(S_u, i, r-1)$ equals sum of $D_u[1..i, 1..(r-1)w]$. Therefore, the sum of the two terms equals sum of $D_u[1..i, 1..j]$ for node u. □

The query algorithm for $select\_sum(D_u, i, k)$ is shown in Lemma 3.5. This algorithm shares a similar intuition to that in Lemma 3.4. It finds the submatrix division boundary that approximates value k using matrix S_u, then recursive into the child submatrix to find exact column.

Lemma 3.5. *Let $r = select\_sum(S_u, i, k)$, we have*

$$select\_sum(D_u, i, k) = \begin{cases} r & \text{if } u \text{ is a leaf} \\ w(r-1) + select\_sum(D_v, i', k - sum(S_u, i, r-1)) & \text{otherwise} \end{cases}$$

where w is the width of $C_{child(u,1)}$, v is r-th child of u, and $i' = count\_nzero(S_u, i, r) - count\_nzero(S_u, i, r-1)$.

Proof. The basis of the equation is similar to that of Lemma 3.4, since $D_u = S_u$ for all leaves u.

Assume the equation holds up to level $d+1$. Consider node u in level d. We can find the child submatrix r whose sum in row i is greater than or equals to k, using $select\_sum(S_u, i, k)$. Let k' be the sum of $D_u[1..i, 1..w(r-1)]$. Since the k-th sum lie within the r-th submatrix, it is equals the start of the first column of D_v which is $w(r-1)$ plus the index of $(k-k')$ sum in D_v. Since the equation holds for level $d+1$, the index of $(k-k')$ in D_v can be found by $select\_sum(D_v, i', k-k')$. Therefore, the inductive equation holds. □

3.4 Narrow balance matrix

In this section, we present the implementations of the operations *sum*, *select_sum* and *count_nzero* in narrow matrices which defined in Section 3.3.1. First, recall some definitions. A narrow balance matrix $T[1..m, 1..\log^{\epsilon} m]$ consists of two parts: the matrix body and the first/last row. The matrix body is the sub-matrix $T[2..m-1, 1..\log^{\epsilon} m]$. Each row of the matrix body has only one non-zero number; and this number is either 1 or -1. The first row of T can contain any non-negative numbers. The sum of these

numbers in this row is n. The last row of T mirrors the values of the first row with negative values.

In this section, we show that given access to a narrow balance matrix in row position format, auxiliary data structures that supports the queries *sum*, *select_sum* and *count_nzero* in constant time can be built using only the small order number of bits of the original matrix.

Recal that, the row position format is specified as: The body of matrix T is represented by an array $Y[1..m-2]$ where each entry $Y[r] = T[r+1, c] \times c$ if $T[r+1, c]$ is the only non-empty entry in row $r+1$. Each chunk of length ρ of Y (i.e. $Y[r..r+\rho-1]$) can be accessed in constant time where $\rho = O(\log m / \log\log m)$. Let F be the first row of the matrix i.e. $F = T[1, 1..\log^{\epsilon} m]$. Let P be the prefix sum array of F i.e. $P[i] = \sum_{i'=1}^{i} F[i']$.

Our data structure is divided into two cases based on the relation between m the height of the matrix, and n the sum of the values in the first row. The first case is when $n = O(m^a)$, the second case is $n = \Omega(m^a)$. In the first case, n and m are in the same order. The values that bounded by n or m can be both stored in $O(\log m) = O(\log n)$ bits. The data structure does not need for distinction between the first row and other rows. Section 3.4.3 solves the first case using word RAM operations. In the second case, n is significant larger than m. We use some additional structures to reduce this case to the first case. The details for this case is presented in Section 3.4.4.

This section is organized as follow. Subsection 3.4.1 introduces some constant time operations when the inputs are fit in some RAM words. Subsection 3.4.2 gives an variant of rank/select data structure that we use in the construction. Subsection 3.4.3 and Subsection 3.4.4 solves the first case and the second case, respectively.

3.4.1 Sub-word operations in word RAM machine

Word RAM machine is a simplified model of our modern computer. In this model, computer has a memory of size n words. Each word contains $\mathtt{w}$ bits and $\mathtt{w} = \Omega(\log n)$. Any word can be accessed in constant time; and arithmetic operations (e.g. plus, minus, multiple) between two words also take constant time. Using the power of RAM model, operations whose input and output are in $O(\mathtt{w})$ bits can be computed in constant time using pre-computed tables or arithmetic operations.

In our work, we requires a few constant time operations for small input and output.

The first group of operations handles small arrays. Consider two array A and B of length $\log^\epsilon m$ that contain integers smaller than $O(\log \log m)$ bits.

- Operation $madd\_array(A, B)$ returns a array C of length $\log^\epsilon m$ such that $C[i] = A[i] + B[i]$.
- Operation $msuccessor\_rank(A, v)$ returns the minimum index i such that $A[i] \geq v$ where $v < \log^2 n$.

The second group of operations compute *sum* and *count_nzero* on some sub-matrix of a narrow balance matrix. Consider a narrow balance matrix $T[1..m, 1..\log^\epsilon m]$. Let $\rho = \log m / \log^4 \log m$. Let $T' = T[r..r + \rho - 1, 1..\log^\epsilon]$ for some $r > 1$. Let Y' be the row position format for submatrix T'. Since Y' only takes $O(\log m / \log^3 \log m)$ bits, we defines the following operations:

- Operation $mcount(Y', i, k)$ counts the non-zero numbers in sub-matrix $T'[1..i, 1..k]$.
- Operation $msum(Y', i)$ returns an array $E[1..\log^\epsilon m]$ such that $E[k]$ is $sum(T', i, k)$. For convenience, we also define variant of this operation called $msum(Y', i, k)$ which returns only $sum(T', i, k)$.

Lemma 3.1. *All the above operations can be computed in $O(1)$ time given some lookup tables of size $o(m)$ bits.*

Proof. The first operation $madd(A, B)$ can be implemented only arithmetic operations. Since the bit length of each A and B is $O(\log^\epsilon m \log \log m)$, which is asymptotically smaller than $\log m \leq \mathtt{w}$. Therefore, the whole array can be stored in a constant number of words. Each element is stored in two's complement binary format with two bits guard for overflow. Operation *add* then can be perform by a normal integer addition operation.

The second operation $msuccessor\_rank(A, v)$ can also be implemented without table look-up, but it requires multiplication and some bitwise operations [45, 109]. If these operations are not available, we can always use table look-up of $o(m)$ bits.

Consider operation $mcount(Y', i, k)$, since sub-array Y' takes $O(\log m / \log^3 \log m)$; and each index takes $O(\log \log m)$ bits. The input takes $O(\log m / \log \log m)$ bits. Therefore, the table of this operation takes $O(2^{\log m / \log \log m} \log \log m) = o(m)$ bits. Similarly, the table for $msum$ takes $O(2^{\log m / \log^2 \log m} \log m) = o(m)$ bits. □

3.4.2 Predecessor data structures

Consider a set Y of m integers in a range from 1 to n, we are interested in few operations on this set:

- Predecessor operation $pred(Y, x)$ returns the smallest number smaller than or equal to the query value x i.e. $pred(Y, x) = \min\{y \mid y \in Y, y \leq x\}$.
- Number rank operation $rank(Y, x)$ counts the numbers of number in Y that is smaller than or equals to x.
- Number select operation $select(Y, i)$ returns the i-th smallest number in Y.

Although all of these operations can be implemented in constant time using the operations on a bit vector of length n with m one, mentioned previously, this approach requires the $o(n)$ term which can be dominant when $m \ll n$. This subsection discusses solutions with less space for small m.

There are a few ways to store the set Y to support *select* in $O(1)$ time. For example, the values of Y can be stored in an sorted array of $n \log n$ bits. The more advanced methods includes Elias-Fano representation [29, 30] which uses $m \log(n/m) + O(m)$ bits.

For the *pred* operation, Grossi et al. [53] showed that a data structure for this operation can be implemented using the *select* operation with a small space overhead.

Lemma 3.2. *(Grossi et al. [53]) Given an data structure that can support select(Y, i) in $S(n)$ time, there exists a representation that uses additional $O(m \log \log n)$ bits and supports pred(Y, x) in $O(S(n) \log m / \log \log n)$ time.*

The value of this result is the set Y does not need to be explicitly stored to support predecessor query. Since $pred(Y, x)$ and $rank(Y, x)$ are closely related, the structure can be simply extended to add the *rank* operation.

Corollary 3.3. *Given an data structure that can support select(Y, i) in $S(n)$ time, there exists a representation that uses additional $O(m \log \log n)$ bits and supports rank(Y, x) in $O(S(n) \log m / \log \log n)$ time.*

Proof. Although the data structure in [53] can be directly changed to support $rank(Y, x)$ operation, we use it as a black box. Given a predecessor data structure for Y, we use a monotone minimal perfect hashing [11] which costs an extra $O(m \log \log n)$ bits to map each element of Y to its rank with constant time lookup. □

In later sections, we use a variation of the rank operation of predecessor search called $successor\_rank(Y, x)$. It returns the rank of the smallest number that is larger than or equals to x. This operations can be easily implemented using the traditional $rank$ and $select$ operation.

3.4.3 Balance matrix for case 1

This subsection shows how to implement the operations sum, $select\_sum$ and $count\_nzero$ in narrow balance matrix T when n the sum of the first row is the same order as the height of the matrix.

We need to conceptually divide the matrix $T[1..m, 1..\log^\epsilon m]$ into $\frac{m}{\log^2 m}$ buckets; each is a submatrix of size $\log^2 m \times \log^\epsilon m$. Let $\tau = \log^2 m$ denote the height of each bucket. Bucket i contains rows $(i-1)\tau + 1$ to $i\tau$ of T.

Each bucket is further subdivided into $\log m \log^4 \log m$ sub-buckets, each is a submatrix of size $(\log m / \log^4 \log m) \times \log^\epsilon m$. Denote $\rho = \log m / \log^4 \log m$ as the height of each sub-bucket. Let $r_{i,j}$ denote the start row of sub-bucket j in T. i.e. $r_{i,j} = (i-1)\tau + (j-1)\rho + 1$ Fig. 3.5 demonstrates how the matrix T is partitioned into buckets and sub-buckets.

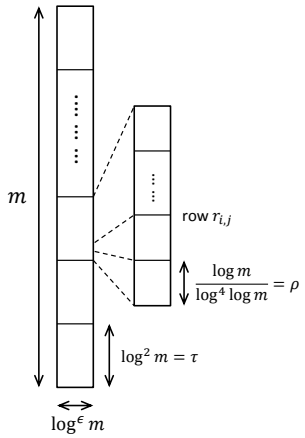


Figure 3.5: Bucket illustration

Auxiliary structures for sum and $count\_nzero$

Our data structure stores some arrays for each bucket and each sub-bucket. For each bucket $i = 1, \cdots, m/\tau$, we store two arrays $P_i[1..\log^\epsilon m]$ and $Q_i[1..\log^\epsilon m]$ where $P_i[k]$ equals $sum(T, r_{i,1} - 1, k)$ and $Q_i[k]$ equals the number of non-zero entries in $T[1..r_{i,1} - 1, 1..k]$ for each $k = 1..\log^\epsilon m$.

For each sub-bucket j of the bucket i, we store two arrays $R_{ij}[1..\log^{\epsilon} m]$ and $S_{ij}[1..\log^{\epsilon} m]$ where $R_{ij}[k]$ equals $sum(T, r_{i,j} - 1, k) - P_i[k]$. $S_{ij}[k]$ equals the difference between the number of non-zero entries in $T[1..r_{i,j} - 1, 1..k]$ and $Q_i[k]$. Note that the row position representation of the sub-bucket j of the bucket i is $Y[r_{i,j}..r_{i,j+1} - 1]$, which takes $O(\epsilon \log m)$ bits.

Now, we describe how to compute $sum(T, i, k)$ and $count\_nzero(T, i, k)$ for narrow balance matrix T. Observe that, for any i,

$$i = (i_1 - 1)\tau + (i_2 - 1)\rho + i_3 \tag{3.2}$$

where $i_1 = \lceil \frac{i}{\tau} \rceil$, $i_2 = \lceil \frac{i \mod \tau}{\rho} \rceil$, $i_3 = i \mod \rho$.

Intuitively, i_1 and i_2 are the bucket and sub-bucket that row i is in; i_3 is relative the index of i inside the sub-bucket. Based on the data structure, the total sum in $T[1..i - i_3, 1..k]$ is $P_{i_1}[k] + R_{i_1,i_2}[k]$. The sum in $T[i - i_3 + 1..i, 1..k]$ equals $msum(Y[r_{i_1,i_2}..r_{i_1,i_2+1} - 1], i_3, k)$. Hence, we can compute $sum(T, i, k)$ in $O(1)$ time:

$$sum(T, i, j) = P_{i_1}[k] + R_{i_1,i_2}[k] + msum(Y[r_{i_1,i_2}..r_{i_1,i_2+1} - 1], i_3, k) \tag{3.3}$$

Similarly, we can show that $count\_nzero(T, i, k)$ can be computed in $O(1)$ time using Q_i and S_{ij} and $mcount$.

Auxiliary structures for *select_sum*

For each bucket i, we store an array $D_i[1..\log^{\epsilon} m - 1]$ and predecessor data structure F_i (Section 3.4.2). Recall that $P_i[k] = sum(T, r_{i,1} - 1, k)$. F_i is the predecessor data structure of set of values of $P_i[1..\log^{\epsilon} m]$. The array D_i is such: $D_i[1] = \min\{P_i[1], 2\tau\}$, and for $k > 1$, $D_i[k] = \sum_{k'=1}^{k} \min\{P_i[k] - P_i[k-1], 2\tau\}$. Intuitively, D_i is a scaled down version of P_i. If all the values of P_i are less than 2τ, D_i equals P_i. Note that, each value $D_i[k]$ is less than $\tau \log^{\epsilon} n = \log^{2+\epsilon} m$, therefore, the whole array D_i is fitted in one word of RAM.

To compute $select\_sum(T, i, v)$, first, we also compute the relative bucket and sub-bucket indexes i_1, i_2, i_3 using Equation 3.2. Next, we have the following observation:

Lemma 3.4. *Given a narrow balance matrix T, for any $i > r$ and $k > r$, we have:*

$$select\_sum(T, i-r, v-r) \leq select\_sum(T, i, v) \leq select\_sum(T, i-r, v+r)$$

Proof. From the definition of balance matrix, we have two properties of *sum*: (1) $sum(T, i, j) \leq sum(T, i, j+1)$, and(2) $|sum(T', i, j) - sum(T', i-r, j)| \leq r$ for any positive integer $r < i$.

Let $j_1 = select\_sum(T, i-r, v-r)$ and $j_2 = select\_sum(T, i-r, v+r)$. By the definition of *select_sum*, we have $sum(T, i-r, j_1) \geq v-r$ and $sum(T, i-r, j_2) \geq v+r$. From the properties of *sum*, $sum(T, i, j_1) \leq sum(T, i-r, j_1) + r \leq v$. Similarly,we have $sum(T, i, j_2) \geq v$. Therefore, $sum(T, i, j_1) \leq v \leq sum(T, i, j_2)$. Since, $j_1 \leq j_2$. Thus, $j_1 \leq select\_sum(T, i, v) \leq j_2$. The lemma follows. □

The lemma shows that the value of $select\_sum(T, i, v)$ can be approximated by searching in another row. We find the lower bound l for $select\_sum(T, i, v)$ using the row $r_{i_1,1}-1$ (i.e. the row before the start row of bucket i_1). Thus, l equals $select\_sum(T, r_{i_1,1}-1, v-(i-r_{i,1}))$. By the structure definitions, the value of l can be computed by using the predecessor data structure F_{i_1}. i.e. $l = successor\_rank(F_{i_1}, v-i+r_{i_1,1})$.

Next, we need to compute two intermediate arrays $E[1..\log^\epsilon m]$ and $C[1..\log^\epsilon m]$ to account for the sum changes from row $r_{i_1,1}$ to row i. First, $E[k] = sum(T[r_{i_1,i_2}..i], i_3, k)$. It is the changes in the sub-bucket i_2 in bucket i_1. Second, $C[k] = D_{i_1}[k] + R_{i_1,i_2}[k] + E[k]$ where R_{i_1,i_2} was defined previously as the array keeps sums between the start of the bucket and the start of the sub-bucket. Both arrays are computed using table look-up. Note that, if $P_i[k] < 2\tau$ for all k, then $C[k] = sum(T, i, k)$; a successor search for v in C can return the values of $select\_sum(T, i, v)$. To handle the case when there are some $P_i[k] > 2\tau$, we scale the value of v to its relative value in array C with the guide from the lower bound position l. i.e. $v' = v - sum(T, i, l) + C[l]$.

The whole computational process is summarized in the next algorithm.

```
0 function select_sum(T, i, v)
1   Compute the indexes i1, i2, i3 as in Equation 3.2
2   l = successor_rank(F_{i1}, v − i + r_{i,1}) /* Lower bound value of the answer */
3   E = msum(Y[r_{i1,i2}..r_{i1,i2+1}], i3)
4   C = madd_array(D_{i1}, R_{i1,i2}, E) /* The sum changes from row r_{i,1} to row i */
```

```
5   v' = v − sum(T, i, l) + C[l]
6   return msuccessor_rank(C, v')
```

Listing 3.1: select sum in narrow matrix

Space analysis

The total space for the auxiliary structure is accounted as follows. The lookup tables for operations *msum*, *mcount*, *madd_array* and *msuccessor_rank*, can be stored in $o(m)$ bits.

For each bucket, the auxiliary data structure are arrays P_i, Q_i for *sum*/*count_nzero* and, F_i, D_i for *select_sum*. Each of these structure uses $O(\log^\epsilon m \times \log m)$ bits. We have total $m/\tau = m/\log^2 m$ buckets. Therefore, the total size of these structures is $O(m/\log^{1-\epsilon} m) = o(m)$ bits.

For each sub-bucket, the auxiliary structures are $R_{i,j}$ and $S_{i,j}$. Each of these structures uses $O(\log^\epsilon m \times \log\log m)$ bits. There are $m \log^4 \log m/\log m$ sub-buckets. The total size of all the sub-buckets is $O(m \log^5 \log m/\log^{1-\epsilon} m) = o(m)$ bits.

Thus, the additional space (without the original matrix) is $o(m)$.

3.4.4 Data structure case 2

This subsection solves the case 2 of the data structure when the sum of the first row is much larger than the height of the matrix. Given a balance matrix $T[1..m, 1..\log^\epsilon m]$, let T' be the body of the matrix i.e. $T' = T[2..m-1, 1..\log^\epsilon m]$. Let F be the first row of the matrix $F = T[1, 1..\log^\epsilon m]$. We have matrix T can be represented by two components (T', F).

Query *sum* in T can be simulated by *sum* queries in T' and F. i.e. $sum(T, i, k) = sum(T', i-1, k) + \sum_{k'=1}^{k} F[k']$. We use the auxiliary data structure in the previous subsection for T', and a prefix sum data structure for F. Similarly, query $count\_nzero(T, i, k)$ can be supported using T' and F. Next, we show how to support query *select_sum*.

Supporting query *select_sum*

To reduce this case to the previous case for *select_sum*, we store the following data structures:

- Predecessor data structure P (Section 3.4.2) for the prefix sums of the row F. (i.e. P stores the set $\{\sum_{k'=1}^{k} F[k'] \mid k = 1..\log^{\epsilon} m\}$.)

- Auxiliary structures for narrow balance matrix data structure in case (1) for balance matrix represented by (T', F') where $R'[1..\log^{\epsilon} m]$ is an array such that $F'[k] = \max(F[k], m)$.

The algorithm for case (2) is shown next. Conceptually, we divide columns of the matrix T into groups of base on the values of F. Two consecutive columns k and $k+1$ are in the same group if $F[k] \leq m$. Let $k = \mathit{select\_sum}(T, i, v)$ be the answer that needed to be found. Based on Lemma 3.4, we can find the group that column k falls in by searching in the first row (shown in Line 2 of the algorithm). The query, then, is reduced to searching for k inside the group (shown the rest of the algorithm).

```
1 function select_sum(T, i, v)
2   g = successor_rank(P, v)
3   v' = v − sum(T, i, g) + sum((T', F'), i, g)
4   return select_sum((T', F'), i, v') /* reduced to case (1) */
```

The size of the predecessor data structure for P is $O(\log^{\epsilon} m \log\log n)$. The sizes of the auxiliary structures for (T', R') and G are both $o(m \log\log m)$.

Storing a set of balance matrices

In Corollary 3.3, the inputs are q matrices with the same size $[1..\ell] \times [1..\log^{\epsilon} m]$ where $m = q\ell$. Our modification for this type of input is very simple.

Both the data structure for bucket and sub-bucket are arrays and data structures that linearly scale with input matrix height. Therefore, we can concatenate these block data structure together for multiple matrix input. The table lookup for the word RAM operation can be shared between all the matrices since they have the same width.

3.5 Application on multi-version FM-index

A FM-index [36] is a very popular data structure for text indexing. The FM-index uses as much space as the compressed text of the original sequence, and can search for the occurrences of a pattern in linear time. The core of the implementation of a FM-index is a rank and select data structure. Therefore, in this section, we apply the multi-version

rank and select data structure in Section 3.2 to have a multi-version FM-index data structure.

To construct an FM-index, we need two concepts: suffix array and Burrows-Wheeler sequence as follows. Consider any sequence S with a special terminating character \$ which is lexicographically smaller than all the other characters. The *suffix array* SA_S is the array of integers specifying the starting positions of all suffixes of S sorted lexicographically. Formally, $SA_S[1..n]$ is an array of integers such that $S[SA_S[i]..n]$ is lexicographically the i-th smallest suffix of S.

Let SA_S be the suffix array of S. The Burrows-Wheeler (BW) sequence [17] of S is a sequence which can be specified as follows:

$$BW_S[i] = \begin{cases} S[SA_S[i]-1]] & \text{if } SA_S[i] \neq 1 \\ S[n] & \text{if } SA_S[i] = 1 \end{cases}$$

Let $C_S[x]$ be an array of integers such that $C_S[x]$ stores the total number of characters in BW_S which lexicographically less than character x.

A *FM-index* of S is a data structure that stores array $C_S[x]$ and the rank and select data structure for BW_S. According to [36], the time required for searching a pattern P in S using the FM-index of S is $O(|P|r)$ where r is the time to compute $rank(BW_S, x, i)$.

Consider a multi-version sequences $\mathcal{S} = \{S_1, \ldots, S_m\}$ where S_{i+1} is different with S_i by one edit operation, a multi-version FM-index is a set of FM-indexes for each sequence S_i for $i = 1..m$. In short, let BW_i and C_i denote the Burrows-Wheeler sequence of S_i and the array C_{S_i}, respectively. The data structure for multi-version FM-index consists of:

- Multi-version rank and select data structure for sequences $\{BW_i\}$.
- A simple data structure for the set of arrays $\{C_i\}$

Instead of storing the rank and select data structure for all sequences $\{BW_i\}$, we can first compute the minimum number of insertions and deletions which converts BW_i to BW_{i+1}. Second, using multi-version rank and select in Section 3.2 to record all the edit operations from BW_i to BW_{i+1} for every i. Let's call this data structure W. Third, store an array $Z[1..m]$ where $Z[i]$ counts the number of edit operations from BW_1 to BW_i. Thus, we have $rank(BW_i, c, j) = rank(W_{Z[i]}, c, j)$.

For the set of arrays $\{C_i\}$, we use the following data structure: Since BW_{S_i} is a permutation of the characters of S_i; and the difference between S_i and S_{i+1} is one edit operation, thus, the count of characters in S_i and S_{i+1} differs by only one. Let σ be the size of the alphabet. Construct a balance matrix $Q[1..2(m+1), 1..\sigma]$ to record the change in $\{C_i\}$. Row 1 of Q is exactly the same as row C_1. Set $Q[i+1, x]$ to 1 if the number of character x in BW_i is one more than that in BW_{i+1}. Set $Q[i+1, x]$ to -1 if the number of character x in BW_i is one less than that in BW_{i+1}. The upper half of Q (i.e. $Q[m+1..2m, 1..\sigma]$) is set to mirror the lower half so that we have a balance matrix. We have $C_i[x]$ equals $sum(Q, i, x)$ (See Section 3.3.1 for details on balance matrix).

To estimate the complexity of this multi-version FM-index data structure, we need to count the number of edit operations between each pair of BW_{i+1} and BW_i. This has been studied in [72]:

Lemma 3.1. *(Leonard et al. [72]) The average number of elements in BW_{i+1} reordered compared to BW_i is at most equal to L_{avg}, the average length of the longest common prefix between $S[SA_S[i]..n]$ and $S[SA_S[i+1]..n]$ for $i = 1..n-1$.*

Although in the worse case L_{avg} can be in $O(n)$, when string S is random generated by a Markov model of order one, Fayolle and Ward [32] proved that L_{avg} is about $\log n/H_1(S) + c$ where c is a constant and $H_1(S)$ is the first order empirical entropy of S. In addition, by experiments [72] suggested that L_{avg} is roughly $\log n$ in random generated texts, natural-language texts, and biological sequences.

From here, we can summarize our result for multi-version FM-index as follows. Given the multi-version sequences $\mathcal{S}$, let S be a sequence that equals S_1 concatenated with all the inserted characters from the editing operations of $\mathcal{S}$. Let $n = |S|$. Let z be the total number of changes between BW_{i+1} and BW_i for $i = 1..m-1$.s

Lemma 3.2. *There exists a data structure for multi-version FM-index that uses $|S|H_k(S) + O(z\log(z+n)) + o(|S|\log\sigma)$ bits. The data structure can search for the occurrences of a pattern P in $O(|P|\log z/\log\log z)$ time. On the average case, z equals $O(m\log n)$, the data structures uses $|S|H_k(S) + O(m\log^2(m+n)) + o(|S|\log\sigma)$ bits.*

Proof. From Theorem 3.1 the size of the multi-version rank and select is $(nH_k(S) + 2z(\log z + \log n))(1 + o(1)) + o(n\log\sigma)$ bits. From Section 3.3.1, the data structure for set of arrays $\{C_i\}$ uses $(\sigma\log n + m\log\sigma)(1 + o(1))$ bits. □

3.6 Experiments

We implemented two methods for multi-version *rank*/*select* problem. The first method saves space but has slower query speed. The second method is more efficient but requires more space. Both methods implement the the *rank*/*select* data structure and operations in Section 3.2. The differences are in the details of the *sum*/*select_sum* operations. The first method implement the *sum* operation using a 2D-sum data structure based on wavelet tree [89]. The underlying bit vectors of the wavelet tree are compressed using RRR compression scheme [94]. The size of the wavelet tree is therefore $nH_0(V)$ where $H_0(V)$ is the zero-th order entropy of the version numbers. To support the *select_sum* operation, we perform a binary search on each row of the matrix. The time for the *select_sum* operation is therefore $O(\log^2 n)$. The second method implements a simplified version of the *sum* and *select_sum* operations in Section 3.3. In our implementation, we have $k = 2$ and use RRR compression scheme of the bit-vectors. The sum and *select_sum* operations take $O(\log n)$ time.

We use two datasets to test these implementations. The first one is simulated dataset. We use it to compare the performance of the methods in pure uniformly random model. The second data set is the DNA sequences of 36 wild yeast genomes. We measure the size of each method, and the query speed for each operation access, rank and select. In this dataset, we also measure the performances of the methods in the BWT-transformed sequences of the genomes. The purpose of this data set is to explore the feasibility of using the multi-version rank/select for the multi-version FM-index application.

3.6.1 Simulated dataset

We start with an empty sequence and generate 4 million linear insertions/deletions to this sequence to create 4 million versions. If all the sequence versions are stored explicitly the total length is about 2665 billion characters. Using our data structures, the sequences can be stored in about 23 MB and 73 MB using the space-efficient approach and the time-efficient approach, respectively. The space-efficient data structure can answer about one thousand rank or select queries per second on our Intel Xeon X5680 CPU machine. The time-efficient method can answer 18 thousands rank queries and 32 thousands select queries per second.

	Yeast genomes	BWT-transformed yeast genomes
Number of sequences	36	
Total length	412,279,635	
Average sequence length	11,452,212	
Number of changes	19,671,844	42,504,388
Consecutive changes	16,183,025	6,614,441

Figure 3.6: Summary of the real dataset of wild yeast (S. paradoxus) from `http://www.sanger.ac.uk/research/projects/genomeinformatics/sgrp.html`

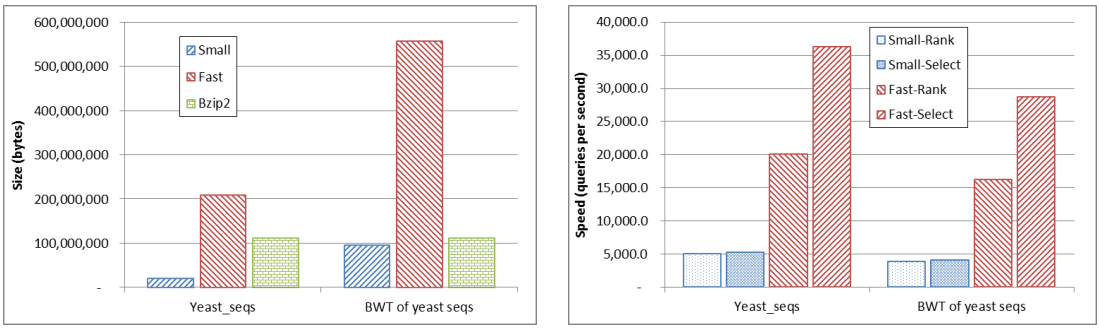
3.6.2 Real datasets

In the yeast genomes data sets, we assume that the DNA sequence of each yeast strain represents a linear versioning of a changing sequence. The order of the versioning is computed heuristically. We first build a phylogenetic tree of the sequences, the ambiguous order the sequences in each sub-tree based on their names, and then the leaves of the tree are numbered from left to right to make a linear version order. The changes (insertions/deletions) between each pair of versions are computed by LAGAN[16], a genome pairwise alignment program. The overall changes between all versions are based on the algorithm in Section 3.2.1.

In the wild yeast data set, there are 36 DNA sequences of total 412 mega-bases. Each sequence length is about 11 mega-bases (See the table in Figure 3.6 for the summary of the dataset.) G-zip and Bzip2 programs take about 2 bits per base to compress these sequences. After the alignment, we found about 19 million changes between the sequences. Most of the changes (16 millions) occur consecutively. In the BWT-transformed dataset, the number of changes is doubled; however the number of consecutive changes is only about a third.

The experimental results are shown in Figure 3.7 where the space of the methods are measured in bytes and the speed is measured by number of queries per second. In the original sequences, the size of the space-efficient method is 10 times smaller than the fast-method, while the speed of queries of the space-efficient are about 5-7 times slower. In the BWT-transformed sequences, the space-efficient method is about 5 times smaller, while the queries are about 5-8 times slower. For the space-efficient method, there is no different in the speed of the rank and select queries. For the time-efficient method, the query speed for select is almost doubled that of rank.

Based on the experimental results, the space-efficient method is more suitable for



(a) (b)

Figure 3.7: Data structure performance. (a) Space usage (b) Query speed. The space-efficient method is named "Small". The time-efficient method is named "Fast".

real-world application since the number of versions is usually much smaller than the number of changes, and the changes are usually consecutive. The zero-th order entropy compression in this method reduces the space substantially.

Chapter 4

RLZ index for similar sequences

4.1 Introduction

There is an increasing need for indexing methods that can store collections of *similar* strings (or repetitive text) compactly while supporting fast pattern searching queries. For example, in genomic applications, the sequencing of individual genomes is becoming a feasible task. The "1000 Genomes Project" [1], aimed at characterizing common human genetic variations, has already sequenced the partial genomes of a large number of persons from various populations. Aligning a read from a sample to multiple human genomes has been proven to be useful for identifying polymorphisms [104]. In the near future, researchers will face the problem of storing those individual (and highly similar) genomic sequences compactly and indexing them efficiently. As another example, Wikipedia documents are modified and snapshots are taken every day to remember older versions of the data. Typically, changes between versions are small. Hence, fast indexing methods for compressed similar texts may allow people to search archived versions of Wikipedia documents quickly. The above applications motivate the following general task:

The string set self-indexing problem: Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_t\}$, construct a data structure that can subsequently report all exact occurrences in $\mathcal{S}$ of any query pattern without using $\mathcal{S}$.

This chapter is concerned with the case where the given strings are similar. Before stating our new results, we survey some existing data compression methods and compressed indexes that are suitable for sets of similar sequences in the next two subsections. Throughout this chapter, we use the terms "string" and "sequence" synonymously.

4.1.1 Similar text compression methods

To compress a single string S of length n, methods that are guaranteed to achieve the empirical k-order entropy $nH_k(S)$ are often used. However, this entropy measurement may not be a good bound for repetitive texts whose repeats are longer than k. For example, the storage based on entropy bound of the text SS (where $|S| \gg k$) is $2nH_k(S)$ bits. On the other hand, one can easily encode the text in $nH_k(S) + O(\log n)$ bits. Thus, there are methods that achieve the empirical k-order entropy, yet perform poorly for repetitive texts [106]. As a consequence, compression methods have been designed for specific types of repetitive texts in biology. For example, GenCompress [20] compresses a text considering approximate repeats. Christley *et al.* [21] and Kuruppu *et al.* [67] compressed DNA sequences with respect to a reference sequence. BioCompress [55], XM [18], and COMRAD [66] are other repetitive compressors designed specifically for DNA. Alternative approaches include methods based on grammar compression (for example, Re-pair [71] was one of the first effective grammar-based compression methods) and LZ77 compression [115] for general repetitive texts. Cfact [95] and Offlines [4] greedily replace duplicate text with shorter codes.

The compression methods above can store repetitive texts compactly, but do not allow random access to the compressed text directly. Previous work has addressed this issue. Kreft and Navarro [64] provided the first efficient random access operations for the LZ77 method. Bille *et al.* [13] built additional data structures on top of an existing grammar-based compression scheme to allow random access of any region with only logarithmic extra time per query.

4.1.2 Compressed indexes for similar text

Although the above compression schemes can compress similar sequences, they do not allow us to search for the occurrences of an arbitrary pattern quickly. Below, we survey some specialized data structures for indexing repetitive texts. In a pioneering paper of Mäkinen *et al.* [79], a repetitive text is defined as a collection of strings of total length N, where the strings are assumed to be highly similar, each string length is approximately n, and the strings share an alphabet of size σ. They employed run-length encoding to reduce the redundancy of a suffix array structure. Their approach shrinks the total index size greatly, but the space of the index is still proportional to

Base compression method	Popular in	Effective$^{(*)}$	Search time$^{(**)}$	Reference
LZ78	GIF image	No	linear	[6, 38, 98]
BWT-transform	bzip2	No	linear	[79]
LZ77	zip	Yes	quadratic	[65]
Grammar based		Yes	quadratic	[22, 46]
Restricted structure		Yes	quadratic	[58]
RLZ		Yes	linear	this result

Figure 4.1: Summary of the compressed indexing structures. $^{(*)}$: Effective for similar sequences. $^{(**)}$: The search time is expressed in terms of the pattern length.

the number of strings. In another paper, Huang *et al.* [58] assumed that every string contains at most m' point mutations with respect to a reference string. They designed a space-efficient data structure of size $O(n \log \sigma + m' \log m')$ bits to encode all such strings. Although the resulting data structure is small, their approach cannot index certain other types of similar strings such as *genome rearrangements*, formed by swapping substrings in genomic sequences, efficiently. (When only a few such rearrangements have occurred, long substrings of the genomic sequences will be preserved; they just occur in a different order.) Kreft and Navarro [65] built a self-index based on LZ77 compression. If the text of length N can be compressed using m LZ77 phrases, their data structure is of size $2m \log N + m \log m + 5m \log \sigma + O(m) + o(N)$ bits, but the query time is $O(\ell^2 h + (\ell + occ) \log N)$, i.e., quadratic in the pattern length ℓ, and also dependent on h the maximal number of layers of overlapped phrases which is only bounded by m. In another line of research, Claude and Navarro [22] proposed a self-index for grammar-based compression methods. It uses $O(r \log r) + r \log N$ bits, where r is the number of rules generated by their grammar compression, and the resulting query time is quadratic ($O((\ell^2 + h(\ell + occ)) \log r)$). Using another technique for constructing a grammar from the LZ77 phrasing, Gagie *et al.* [46] obtained a data structure of size $2r \log r + O(m(\log n + \log m \log \log m))$ and query time $O(\ell^2 + (\ell + occ) \log \log N)$. Some results for LZ78 compression and FM-index were given in [6, 38, 98]. They have good query time but require $O(NH_k)$ bit-space in the worst case. They may not be good enough to index a repetitive text in practice [106] or in theory [100]. In summary, existing indexes for a set of similar strings either require: (1) a lot of space, (2) the text to have some special structure, or (3) quadratic query time (for a summary, see the table in Fig. 4.1).

4.1.3 Our results

Our main contribution is a compressed static indexing data structure with two alternative space-time trade-offs. The smaller alternative can store a set of strings $\mathcal{S}$ relatively to a reference string R in asymptotically optimal space. The larger alternative improves the query time at the expense of using more space. The results are summarized as follows:

Theorem 4.1. *Given a reference string R of length n over an alphabet Σ of size $\sigma = O(\log^a n)$ for some constant a and a set of strings $\mathcal{S} = \{S_1, \ldots, S_t\}$ over Σ, let m be the smallest possible number of substrings of R (a.k.a. factors) to represent $\mathcal{S}$. All exact occurrences of any query pattern P of length ℓ can be reported using either of the following alternatives. Their complexity specifications are:*

(a) $\left(2 + \frac{1}{\epsilon}\right) nH_k(R) + O(n) + O(m \log n)$ *bits and* $O(\ell \log^{\epsilon} n + occ \cdot (\log_\sigma^{\epsilon} n + \frac{\log m}{\log n}))$ *query time;*

(b) $\left(2 + \frac{1}{\epsilon}\right) nH_k(R) + O(n) + O(m \log n \log\log n)$ *bits and* $O(\ell \log\log n + occ \cdot (\log_\sigma^{\epsilon} n + \frac{\log m}{\log n}))$ *query time,*

where occ is the number of occurrences of P, k is any positive integer less than $\log_\sigma n$, and $\epsilon \leq 1$ is a constant. For both alternatives, the data structure can be constructed in $O(\sum_{i=1}^{t} |S_i| + (n+m)\log(n+m))$ time.

Our compression scheme is based on a variant of the relative Lempel-Ziv (RLZ) compression scheme from [67]. It represents each $S_i \in \mathcal{S}$ as a concatenation of substrings of R (referred to as *factors*) obtained from the LZ77-like factorization of R. See Fig. 4.2 for an example. (Note that In Lemma 4.1, we proved that the scheme uses the smallest possible number of factors from the reference.) Experiments on large scale genomic data in [67] have shown that this method yields good compression ratios for repetitive texts even when parts of the sequence are rearranged.

In this result, we assume that the reference R is given. In case where R is not available, we can apply the method of Kuruppu *et al.* [68] to find a suitable one. We also assume the alphabet size σ is in polylogarithmic of the word length (i.e. $\sigma = O(\log^a n)$ for some constant a). For larger alphabets, e.g. $\sigma = \Omega(n^\alpha)$, the query time needs an additional term of $O(\ell \log\sigma / \log\log n + occ \cdot \log\sigma)$ and the space needs an additional term of $O(n \log\sigma \log\log n / \log n) = o(n \log\sigma)$.

Both alternatives in Theorem 4.1 use the same pattern searching algorithm. The algorithm considers two cases: Case 1, where the pattern P is a substring of a single factor; and Case 2, where P crosses at least one boundary between two factors. (See Fig. 4.4.) For case 2, the pattern is partitioned into two parts: left and right. The left part ends at the end of the first factor, while the right part begins at the start of the second factor. For each possible partition of the pattern, the left part and right part are searched independently and then joined together by an appropriate 2D range query data structure. However, to avoid the quadratic pattern search time, we use multiple tricks to reuse results between the searches in each partition.

We remark that recently, Gagie *et al.* [46] independently proposed a similar method to index a set of sequences. Their space complexity is $O(nH_k(R) + n + m(\log n + \log m \log\log m))$ bits, and the query time is $O((\ell + occ)\log^{\epsilon} n)$, where $\epsilon > 0$. Thus, compared to the method in our Theorem 4.1 (a), their method always uses more space while having similar time. Compared to that in Theorem 4.1 (b), theirs is slower while having asymptotically comparable space. Also note that in their method, the reference sequence is restricted. It must be one of the sequences in $\mathcal{S}$ (otherwise false occurrences may be reported).

The chapter is organized as follows. Section 4.2 defines the notation used throughout the paper and outlines the framework of our new data structures. Section 4.3 describes some auxiliary data structures used in our construction. These data structures are known in the literature; however, we also present some improvements which may be of independent interest. Section 4.3.3 presents a new data structure for answering a restricted type of 2D range queries. Sections 4.4 – 4.7 describe further technical details of our main data structure.

4.2 Data structure framework

4.2.1 The relative Lempel-Ziv (RLZ) compression scheme

Let R be a reference sequence of length n over an alphabet Σ and let $\mathcal{S} = \{S_1, \ldots, S_t\}$ be a given set of strings over Σ. Each sequence $S_i \in \mathcal{S}$ is compressed based on R by relative Lempel-Ziv (RLZ) compression [67]. Precisely, given two strings S and R, where R contains all the symbols in S, the *Lempel-Ziv factorization* (or *parsing*) of S relative

to R, denoted by $LZ(S|R)$, is a way to express S as a concatenation of substrings of the form $S = w_0 w_1 w_2 \dots w_z$ such that: (1) w_0 is an empty string; and (2) w_i for $i > 0$ is a non-empty substring of S and w_i is the longest prefix of $S[(|w_0..w_{i-1}| + 1)..|S|]$ that occurs in R. Each substring w_i is called a *factor* (or *phrase*), and can be represented by a pair of numbers (p_i, l_i), where p_i is a starting position of w_i in R and l_i denotes the length of w_i.

$LZ(S|R)$ was suggested in [67]. The algorithm, which runs in linear time, is summarized in Fig. 4.3. By definition, the decomposition guarantees that no factor can be expanded any further to the right. Furthermore, the RLZ compression scheme has the following property:

Lemma 4.1. *$LZ(S|R)$ represents S using the smallest possible number of factors.*

Proof. Consider the algorithm to decompose a string into RLZ factors in Fig. 4.3. Let $dist_R(S)$ denote the minimal number of factors of R to represent S. We prove the property by induction. First, any string S of length 1 has a decomposition using $dist_R(S) = 1$ factor of R.

Next, by induction, for any string X of length less than ℓ, we assume X can be

R = ACGTGATAG

S_1 = TGATAGACG = TGATAG, ACG = 8 2
S_2 = GAGTACTA = GA, GT, AC, TA = 5 6 1 7
S_3 = GTACGT = GT, ACGT = 6 3
S_4 = AGGA = AG, GA = 4 5

(a)

$T[..]$	Factor	Pos. in R
1	AC	1..2
2	ACG	1..3
3	ACGT	1..4
4	AG	8..9
5	GA	5..6
6	GT	3..4
7	TA	7..8
8	TGATAG	4..9

(b)

$\overline{T}[..]$	Factor (rev.)	Pos. in R
1	GA	5..6
2	TA	7..8
3	AC	1..2
4	AG	8..9
5	TGATAG	4..9
6	ACG	1..3
7	GT	3..4
8	ACGT	1..4

(c)

Figure 4.2: (a) A reference string R and a set of strings $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ decomposed into the smallest possible number of factors from R. (b) The array $T[1..8]$ (to be defined in Section 4.2) consists of the distinct factors sorted in lexicographical order. (c) The array $\overline{T}[1..8]$.

Input: A string S and the BWT of $\overline{R}$
Output: A decomposition of S, i.e., $S_1 \dots S_k$

1: $i = 1$; $k = 1$:
2: **while** $i \leq |S|$ **do**
3: By backward search on $\overline{R}$, identify the longest prefix $S_k = S[i..j]$ of $S[i..|S|]$ such that S_k is a substring of R.
4: $k = k + 1$; $i = j + 1$;
5: **end while**
6: Report $S_1 \dots S_k$

Figure 4.3: Algorithm to decompose a string into RLZ factors

constructed using $dist_R(X)$ factors of R. Now, consider a string S of length ℓ and assume the algorithm reports $S = S_1 \dots S_k$. To obtain a contradiction, suppose the optimal decomposition is $S'_1 \dots S'_{k'}$ where $k' < k$. Since the algorithm always finds the longest string, we know that $|S_1| \geq |S'_1|$. Note that for $S[|S_1| + 1..\ell]$, the algorithm will decompose it into $S_2 \dots S_k$, which consists of $k - 1$ factors. The induction hypothesis states that $dist_R(S[|S_1| + 1..\ell]) = k - 1$. As $S[|S'_1| + 1..\ell]$ is longer than $S[|S_1| + 1..\ell])$, we have $dist_R(S[|S'_1| + 1..\ell]) \geq dist_R(S[|S_1| + 1..\ell])$. Hence, $k' - 1 = dist_R(S[|S'_1| + 1..\ell]) \geq dist_R(S[|S_1| + 1..\ell]) = k - 1$. Contradiction. □

For every $S_i \in \mathcal{S}$, denote the Lempel-Ziv factorization of each S_i relative to R by $S_i = S_{i1}S_{i2} \dots S_{ic_i}$. Define $m = \sum_{i=1}^{t} c_i$. By Lemma 1, m is in fact the smallest possible number of factors to represent $\mathcal{S}$. Next, take all the s *distinct factors* that appear in the factorizations for $\mathcal{S}$ and let $T[1..s]$ be an array containing these factors sorted in lexicographical order (see Fig. 4.2 (b)). Note that $s \leq \min\{n^2, m\}$. Our data structure stores $T[1..s]$ in $O(s \log n)$ bits by encoding each $T[j]$ by its starting and ending positions in the reference string R, and the set $\mathcal{S}$ in $O(m \log s) = O(m \log n)$ bits by representing each $S_i \in \mathcal{S}$ as a list of indices from $T[1..s]$ (see Fig. 4.2 (a)).

Let $F[1..m]$ be the lexicographically sorted array of all non-empty suffixes in $\mathcal{S}$ that start with a factor; i.e., each element $F[y]$ is of the form $S_{ip}S_{i(p+1)} \dots S_{ic_i}$, and is called a *factor suffix* from here on. See Fig. 4.11 (a) for an example. Importantly, our data structure does not store $F[1..m]$ explicitly. For any string x, $\overline{x}$ denotes its reverse. Let $\overline{T}[1..s]$ be an array of all reversed distinct factors $\overline{S}_{ij}$ sorted lexicographically. By using the relative Lempel-Ziv decomposition, each sequence S_i can be viewed as a new sequence S'_i based on the alphabet of all the distinct factors in $T[1..s]$ (see Fig. 4.2).

4.2.2 Pattern searching

To find the occurrences of a query pattern P in $\mathcal{S}$, we follow the basic strategy outlined in Section 4.1.3. Suppose P is a query pattern of length ℓ. Each occurrence of P in $S_1, \ldots, S_t$ belongs to one of the following two main cases; see Fig. 4.4:

- **Case 1:** P lies completely inside one factor, denoted by S_{ip}.
- **Case 2:** P is not a substring of a single factor, i.e., $P = XS_{ip} \ldots S_{iq}Y$, where X is a suffix of $S_{i(p-1)}$ and Y is a prefix of $S_{i(q+1)}$.

(Observe that the case $P = XY$ is an instance of case 2.) To locate all occurrences of P, our data structure uses a number of auxiliary data structures (explained in subsection 4.2.3), to report all occurrences of P in $\mathcal{S}$ according to case 1 and case 2 separately. Let occ_1 and occ_2 be the number of occurrences of P as in case 1 and case 2, respectively.

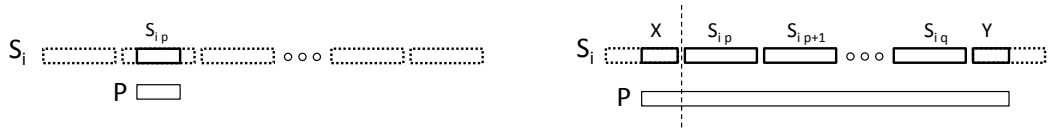


Figure 4.4: When P occurs in string S_i, there are two possibilities, referred to as case 1 and case 2. In case 1 (shown on the left), P is contained inside a single factor S_{ip}. In case 2 (shown on the right), P stretches across two or more factors $S_{i(p-1)}, S_{ip}, \ldots, S_{i(q+1)}$.

Case 1: [P occurs inside a factor] Since all the factors are substrings of the reference R, the pattern is first searched for in the reference. Then, the factors that cover an occurrence of the pattern in the previous step are reported as the result. This case takes $O(\ell + occ_1 \log^{\epsilon} n)$ time, as discussed in detail in Section 4.4.

Case 2: [P is not a substring of a single factor] As illustrated in Fig. 4.4, in this case, every occurrence of P can be divided into two parts: the *left part* ($P[1..j]$ matches some suffix of a factor), and the *right part* ($P[j+1..\ell]$ matches a factor suffix). To find the occurrences of this case, we try to match all the $(\ell + 1)$ possible partitions $(P[1..j], P[j+1..\ell])$ of the pattern. For each partition, the left parts are matched against the set of reversed factors in $\overline{T}$. The successful matches are represented by a range in $\overline{T}$. The right parts are matched against the set of factor suffixes in F. The results are also represented by a range in F. Then, the successful matches of the left part $P[1..j]$ and

right part $P[j+1..\ell]$ are combined and validated using a 2D-range query data structure. (See Fig. 4.5 for an example.)

4.2.3 Overview of our main data structure

The data structure for case 1 is called $\mathcal{I}(T)$ and defined in Section 4.4 to find all occurrences of P in $O(\ell + occ_1 \log^{\epsilon} n)$ time; and uses $2n + o(n) + O(s \log n)$ bits (Theorem 4.1).

The data structures that facilitate the searching for case 2 are more complicated and consist of three components: (i) $\mathcal{X}(\overline{T})$ to match the left parts; (ii) $\mathcal{Y}(F, T)$ to match the right parts; and (iii) $\mathcal{M}$ to report the correct combinations of the left parts and right parts. Further technical details of $\mathcal{X}(\overline{T})$, $\mathcal{Y}(F, T)$, and $\mathcal{M}$ are given in Sections 4.5, 4.6, and 4.3.3, respectively. Note that each of the two alternatives in Theorem 4.1 uses the same components for (i) and (ii). Their space and time trade-off result from using different versions of (iii). The usage of each component is summarized as follows:

(i) First, $\mathcal{X}(\overline{T})$ in Section 4.5 uses $O(s \log n) + o(n)$ bits space. It finds all occurrences of prefixes of P that are equal to a suffix of a factor $S_{i(p-1)}$ in $O(\ell \log \log n)$ time. More precisely, $\mathcal{X}(\overline{T})$ returns, for every j, the maximal range $st_j..ed_j$ in $\overline{T}$ such that $\overline{P[1..j]}$ is a prefix of every element in $\overline{T}[st_j], \ldots, \overline{T}[ed_j]$.

(ii) Second, $\mathcal{Y}(F, T)$ in Section 4.6 uses $(2+1/\epsilon)nH_k(R)+2.55n+o(n \log \sigma)+O(m \log n)$ bits space. It finds all occurrences of suffixes of P that are equal to a prefix of a factor suffix in F, i.e., $S_{ip} \ldots S_{iq}Y$, where Y is a prefix of $S_{i(q+1)}$, in $O(\ell(\log \sigma / \log \log n + \log \log n))$ time. More precisely, $\mathcal{Y}(F, T)$ returns, for every j, the maximal range $st'_j..ed'_j$ such that $P[(j+1)..\ell]$ is a prefix of every element in $F[st'_j], \ldots, F[ed'_j]$.

(iii) Third, we encode all combinations of $S_{i(p-1)}$ and $S_{ip} \ldots S_{iq}Y$ as follows: Define M to be a binary $(s \times m)$-matrix where $M[x, y] = 1$ if and only if $\overline{T}[x]$ is the preceding factor of the suffix $F[y]$, i.e., $F[y] = S_{ip}S_{i(p+1)} \ldots S_{ic_i}$ and $\overline{S}_{i(p-1)} = \overline{T}[x]$ is the x-th lexicographically smallest in $\overline{T}$. Note that each column of the matrix M contains exactly one 1. (See Fig. 4.5)

Lemma 4.2. *All case 2 occurrences of P can be found by listing the entries equal to* 1 *in the rectangles $[st_j, ed_j] \times [st'_j, ed'_j]$ in M, for all j.*

Proof. ($\rightarrow$) Consider an occurrence of case 2 of P in S_i, that is, $S_i[s..e] = P$ and $S_i[a..t]$ is a factor and $a \leq s < t < e$. Let $\overline{T}[p]$ be the entry in $\overline{T}$ that represents the factor $S_i[a..t]$. We have $S_i[t+1..|S_i|]$ is a factor suffix. Let $F[p']$ be the entry in F that represents $S_i[t+1..|S_i|]$. By the definition of the matrix M, there is an entry 1 in $M[p, p']$. Consider $j = (t - s + 1)$, we have $P[1..j] = S_i[a..s]$ is a suffix of $\overline{T}[p]$, and therefore, $st_j \leq p \leq ed_j$. Similarly, $P[j..|P|]$ is prefix of $F[p']$, and therefore, $st'_j \leq p' \leq ed'_j$. Therefore, the occurrence $S_i[s..e] = P$ implies a number 1 in the specified rectangle.

($\leftarrow$) Consider a number 1 in the region $[st_j, ed_j] \times [st'_j, ed'_j]$. The position of the occurrence can be found as follows. Let (i', j') be the position of the number 1. Let $S_i[p..|S_i|]$ be the factor suffix of $F[j']$. We have $P[j+1..|P|] = S_i[p..p+|P|-j]$ Since $\overline{T}[i']$ is the previous factor of $F[j']$, $S_i[p-j..p-1] = P[1..j]$. Therefore, P occurs in S_i from position $p - j$ to position $|P|$. □

For each pattern P of length ℓ, we may need up to ℓ queries in the matrix M to find all the results. (See Fig. 4.11 (b) for an example.) Section 4.3 gives two alternative 2D range query data structures $\mathcal{M}$ that support the operation query_2d$(M, [st, ed], [st', ed'])$ on M for finding these entries: If $\mathcal{M}$ is of size $O(m \log s \log\log s)$ bits, all entries equal to 1 can be found in $O((1 + occ) \log\log s)$ time for each query, and if $\mathcal{M}$ is of size $O(m \log s)$ bits, the query takes $O((1 + occ) \cdot \log^{\epsilon} s)$ time.

	AC-TA	ACG	ACGT	AG-GA	GA-GT-AC-TA	GA	GT-AC-TA	GT-ACGT	TA	TGATAG-ACG
$				1	1			1		1
GA							1			
TA										
AC									1	
AG						1				
TGATAG		1								
ACG										
GT	1		1							
ACGT										

Figure 4.5: Each row represents the string $\overline{T}[i]$ in reverse; each column corresponds to a factor suffix $F[i]$ (with dashes to mark factor boundaries). The locations of the number "1" in the matrix mark the factor in the row preceding the suffix in the column. Consider an example pattern "AGTA". There are 5 possible partitions of the pattern: "-AGTA", "A-GTA", "AG-TA", "AGT-A" and "AGTA-". Using the index of the sequences in Fig. 4.2, the big shaded box is a 2D query for "A-GTA" and the small shaded box is a 2D query for "AG-TA".

As a final step, we need a data structure to decode all occurrences of case 1 and 2

to find their actual locations in $\mathcal{S}$. This simple data structure, called $\mathcal{D}$, is used to convert indices of F to their exact locations in $\mathcal{S}$. It requires $O(m \log n)$ bits, and $O(\log m / \log n + \log \log n)$ time for decoding each occurrence. The details are presented in Section 4.7.

Note that the data structure is designed as a static database. Once the reference sequence is given, the input strings can be factorized in linear time using the algorithm in Section 4.2.1 i.e., $O(N)$ where N is the total length of the input. The construction algorithms for $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{I}$ are not complicated. The internal components can be directly constructed based on their definitions using only constant number of sort and scan operations on some arrays. All the external components (Section 4.3) can be built in $O(m \log m)$ time. The whole data structure can be constructed in $O(N + (n+m) \log(n+m))$ time.

The total space equals the sum of the spaces of all components, namely: arrays T and $\overline{T}$; data structures: $\mathcal{I}(T)$, $\mathcal{X}(\overline{T})$, $\mathcal{Y}(F, T)$, $\mathcal{M}$ and $\mathcal{D}$. (Note that the FM-indexes of R although counted only inside $\mathcal{Y}(F, T)$, it is shared with $\mathcal{I}(T)$ and $\mathcal{X}(\overline{T})$ for looking up values in suffix array.) Putting everything together, the total space requirement is $(2 + 1/\epsilon)\, nH_k(R) + 5.55n + O(m \log n)$ bits, while all occurrences of P in S can be found in $O(\ell(\log \sigma / \log \log n + \log^{\epsilon} n) + occ \cdot (\log_{\sigma}^{\epsilon} n + \frac{\log m}{\log n}))$ time; or $(2 + 1/\epsilon)\, nH_k(R) + 5.55(n) + O(m \log n \log \log n)$ bits and $O(\ell(\log \sigma / \log \log n + \log \log n) + occ \cdot (\log_{\sigma}^{\epsilon} n + \frac{\log m}{\log n}))$ time. When σ is in $\Omega(\log^{O(1)} n)$, the term $\log \sigma / \log \log n$ becomes $O(1)$. We thus obtain Theorem 4.1 above.

4.3 Some useful auxiliary data structures

This section introduces some of the useful auxiliary data structures that we modified from the literature for our construction.

4.3.1 Combined suffix array and FM-index

Consider any string R with a special terminating character \$ which is lexicographically smaller than all the other characters. The *suffix array* SA_R is the array of integers specifying the starting positions of all suffixes of R sorted lexicographically. For any string P, let *st* and *ed* be the smallest and the biggest, respectively, indexes such that P

is the prefix of suffix $SA_R[i]$ for all $st \leq i \leq ed$. Then, (st, ed) is called a *suffix range* or *SA_R-range* of P. i.e. P occurs at $SA_R[st+1], \ldots, SA_R[ed]$ in R. For any given string P specified by its suffix range (st, ed) in SA_R, an *FM-index of R* supports the following operations: $\text{lookup}_R(i)$ returns the value of $SA_R[i]$; $\Psi_R(i)$ returns the index j such that $SA_R[j] = SA_R[i] + 1$; and $\text{backward\_search}_R(c, (st, ed))$ returns the suffix range in SA_R of the string cP, where c is any character and (st, ed) is the suffix range of P.

Lemma 4.1. *Given any string R of length n over an alphabet of size σ, the FM-index of R uses $nH_k(R) + O(n \log \sigma \log\log n / \log n)$ bits and supports* backward_search_R *in $O(\log \sigma / \log\log n)$ time and Ψ_R in $O(1)$ time (where $k < \log_\sigma n$). Given additional $(1/\epsilon)nH_k(R) + 2(\log e + 1)n + o(n)$ bits,* lookup_R *can be supported in $O(\log^\epsilon_\sigma n + \log \sigma)$ time.*

Proof. The FM-index, that uses $nH_k(R)+O(n \log \sigma \log\log n / \log n)$ bits and supports the operations backward_search_R and Ψ_R within the specified time, is described in [37, 78, 88]. [52] showed how to support Ψ_R and lookup_R using $(1+1/\epsilon)nH_k(R)+2(\log e+1)n+o(n)$ bits. If we store the FM-index and the data structure in [52] separately, it takes $(2+1/\epsilon)nH_k(R) + O(n)$ bits. Since these two data structures have some overlap, we can reduce the space when storing both of them by $nH_k(R)$ bits as follows:

Let Σ^i be an alphabet, such that for any character $c \in \Sigma^i$, $c = a_1 a_2 \ldots a_{2^i}$, where $a_j \in \Sigma$. Let R^i be the sequence using the alphabet Σ^i, such that $R^i[j] = R[2^i \times j]R[2^i \times j + 1] \ldots R[2^i \times j + 2^i - 1]$. Let Ψ^i be the Ψ function for sequence R^i.

In [52], a recursive data structure of $(1 + 1/\epsilon)$ level is stored to compute the value of SA_R. Let $h' = \log\log_\sigma n$ and $h = \log\log n$, they store Ψ^i for $i = 0, h'\epsilon, 2h'\epsilon, \ldots h'$ and SA_{R^h}. However, since $\Psi_R = \Psi^0$, and Ψ_R is provided by the normal FM-index, we don't need additional space for Ψ^0. That saves $nH_k(R)$ bits. □

In the FM-index above, the compression technique is only effective for moderate size alphabet (i.e. $\sigma = O(\log^a n)$ for some constant a). When the alphabet is larger (e.g. $\sigma = O(n^\alpha)$), the sequence becomes more like a permutation of distinct numbers. The second term in the space complexity can surpass the main $nH_k(R)$ term; and $H_k(R)$ grows to its $\log n$ upper bound. Moreover, the running time with $\log \sigma$ is no longer a small number. A *general FM-index* is a FM-index extended to alphabets of unbounded size. It is not compressed, but its query time is only $\log\log \sigma$. The next lemma is our

simple extension of the normal FM-index to the general FM-index case, obtained by applying the result from [47] and some additional arrays:

Lemma 4.2. *Given any string S of length m over an alphabet of size s, there exists a general FM-index of S that uses $m \log s + o(m \log s)$ bits and supports* backward_search$_S$ *in $O(\log \log s)$ time and Ψ_S in $O(1)$ time. Using an additional $m \log s + o(m \log s)$ bits,* lookup$_S$ *can be supported in $O(\log m / \log s)$ time.*

Proof. The original data structure for the FM-index uses $m \log s + o(m \log s)$ bits space and support backward_search$_S$ and Ψ_S as described in Sections 3.1 and 3.2 of [47]. We now explain how to support lookup$_S$ using the stated space and time complexity.

Define $\Psi_S^1(i) = \Psi_S(i)$ and $\Psi_S^k(i) = \Psi_S(\Psi_S^{k-1}(i))$ for $k > 1$. Then, we have, $SA_S[\Psi_S^k(i)] = SA_S[i] + k$. Let $t = \log m / \log s$. Use the additional bits to store:

- A succinct bit vector $B[1..m]$ such that $B[i] = 1$ if and only if $SA_S[i] \mod t \equiv 0$.
- An array of integers $V[1..m/t]$ such that $V[i] = SA_S[\text{select}_B(i)]$.

Since $SA_S[1..m]$ contains all values from 1 to m, there exists a value i' such that $SA_S[i'] \mod t \equiv 0$ and $SA_S[i] - SA_S[i'] < t$. From the definitions of B and Ψ_S, we have $B[i'] = 1$ and $i' = \Psi_S^k(i)$ and $k < t$. To find i', iteratively compute $\Psi_S^k(i)$ until $B[\Psi_S^k(i)]$ is 1; this enumeration takes at most $O(t)$ time. The value of $SA_S[i']$ can be looked up from V. Then, $SA_S[i] = SA_S[i'] - k$. Therefore, the value of $SA_S[i]$ takes $O(\log m / \log s)$ time to compute.

For the extra space complexity, the bit vector B uses $O(m)$ bits. The array V uses $m \cdot \log s / (\log m / \log s) \leq m \log s$ bits. In total, we use $2m \log s + o(m \log s)$ additional bits. □

4.3.2 Bi-directional FM-index

Recall that, given a suffix array SA_R, a pattern P can always be represented by an SA-range (st, ed). The traditional FM-index can only extend the search pattern to the left by one character using backward search (i.e. given SA-range of P, *backward_search* returns the SA-range of cP). However, computing the value of array $A[1..|P|]$ in Section 4.6 requires us to modify the search pattern at both the left end and the right end. A trivial solution is to use a heavier data structure called the suffix tree. However, even using the

existing compressed suffix tree [41, 102], the modification at one end of the pattern will take $O(\log n)$ time, which is too much for our requirement in Theorem 4.1 (b).

Therefore, we use a data structure called *bi-directional FM-index* which allows us to extend and delete one character at either end of the pattern in $O(\log \sigma / \log \log n)$ time (where σ is the size of the alphabet).

Consider a sequence R over an alphabet Σ of size σ, the suffix array SA_R, and the suffix array of the reversed sequence $SA_{\overline{R}}$. Given a string X, we let $st(X)$ and $ed(X)$ be the start and the end of the suffix range of X in SA_R, respectively. Similarly, $\overline{st}(X)$ and $\overline{ed}(X)$ denote the start and the end of the suffix range of $\overline{X}$ in $SA_{\overline{R}}$, respectively. Given a pattern $P[1..\ell]$, let r_R be the suffix range of P in SA_R, i.e., $r_R = (st(P), ed(P))$. Let $r_{\overline{R}} = (\overline{st}(P), \overline{ed}(P))$. Let c be any character in Σ. The bi-directional FM-index is a data structure supporting the following four operations:

- $forward\_search(r_R, r_{\overline{R}}, c)$: returns the new suffix range of pattern Pc in SA_R, and the suffix range of $\overline{Pc}$ in $SA_{\overline{R}}$.
- $backward\_search(r_R, r_{\overline{R}}, c)$: returns the suffix ranges of cP and $\overline{cP}$ in SA_R and $SA_{\overline{R}}$, respectively.
- $delete\_back(r_R, r_{\overline{R}})$: returns the suffix range of $P[2..\ell]$ in SA_R, and the suffix range of $\overline{P[2..\ell]}$ in $SA_{\overline{R}}$.
- $delete\_front(r_R, r_{\overline{R}})$: returns the suffix ranges of $P[1..\ell-1]$ and $\overline{P[1..\ell-1]}$ in SA_R and $SA_{\overline{R}}$, respectively.

In other words, the operation $forward\_search$ extends the searching pattern by one character to the right, while the operation *backward_search* extends it one character to the left. The operation *delete_back* deletes the leftmost character from the searching pattern, the $delete\_front$ deletes the rightmost one.

To implement the bi-directional FM-index, we use: the BWT of R, the BWT of $\overline{R}$, the topology of the suffix tree of R, the topology of the suffix tree of $\overline{R}$. (Note that the topology of each tree can be stored in $2.55n + o(n)$ bits each according to [42].) The operations $forward\_search$ and *backward_search* have been considered before by Lam *et al.* [69] and Schnattinger *et al.* [103]:

Lemma 4.3. *(Lam* et al. *[69]) Using the FM-index of R and the FM-index of $\overline{R}$, we can compute operation* forward_search *and operation* backward_search*.*

We will now present how to implement the operation *delete_back*. For $delete\_front$, since R and $\overline{R}$ are symmetric by a string reversal operation, we can implement $delete\_front$ by a similar procedure but swapping the roles of R and $\overline{R}$.

Formally, the problem of computing *delete_back* is: given $st(cX)$, $ed(cX)$ $\overline{st}(cX)$ and $\overline{ed}(cX)$, compute the values of: $st(X)$, $ed(X)$, $\overline{st}(X)$ and $\overline{ed}(X)$. First, $st(X)$ and $ed(X)$ can be computed using the following lemma:

Lemma 4.4. *Given $st(cX)$ and $ed(cX)$, the values of $st(X)$ and $ed(X)$ can be computed using the suffix link of R in $O(1)$ time.*

Proof. Computing the suffix range of $P[2..\ell]$ in SA_R from that of $P[1..\ell]$ is identical to computing the suffix link in the compressed suffix tree [42, 102]. The operation can be done in constant time if the topology of the suffix tree and the function Ψ_R are given. Here Ψ_R is the inverse of the backward search of the FM-index, and can be computed in constant time (See Lemma 4.2). □

Then, the values of $\overline{st}(X)$ and $\overline{ed}(X)$ can be computed using the following Lemma:

Lemma 4.5. *$\overline{st}(X)$ and $\overline{ed}(X)$ can be computed by using the following equations:*

$$\overline{st}(X) = \overline{st}(cX) - \textstyle\sum_{a<c} [ed(aX) - st(aX) + 1]$$
$$\overline{ed}(X) = \overline{ed}(cX) + \textstyle\sum_{z>c} [ed(zX) - st(zX) + 1]$$

Proof. As $\overline{cX} = \overline{X}c$, $\overline{X}$ is prefix of $\overline{cX}$. Note that $(\overline{st}(cX), \overline{ed}(cX))$ is the suffix range of $\overline{cX}$ in $SA_{\overline{R}}$. Therefore, $\overline{st}(X) \leq \overline{st}(cX) \leq \overline{ed}(cX) \leq \overline{ed}(X)$.

Let $\Delta_{\overline{st}} = \overline{st}(cX) - \overline{st}(X)$. Because c is the last character of $\overline{X}c$, hence, $\Delta_{\overline{st}}$ equals the number of occurrences of substrings $\overline{X}a$ in $\overline{R}$ for all character a in Σ and $a < c$. Thus, $\Delta_{\overline{st}}$ equals the number of occurrences of aX in R for all $a < c$. Note that the number of occurrences of aX in R is $[ed(aX) - st(aX) + 1]$. That is, $\overline{st}(X) - \overline{st}(cX) = \sum_{a<c} [ed(aX) - st(aX) + 1]$.

Similarly, let $\Delta_{\overline{ed}} = \overline{ed}(X) - \overline{ed}(cX)$. $\Delta_{\overline{ed}}$ equals the number of occurrences of zX in R for all character z and $z > c$. We obtain $\overline{ed}(X) - \overline{ed}(cX) = \sum_{c<z} [ed(zX) - st(zX) + 1]$.

□

From Lemma 4.4 and Lemma 4.5, we can compute *delete_front*. *delete_back* follows similarly. To summarize, we have the following theorem:

Theorem 4.6. *The bi-directional FM-index can be implemented using the BWT of R, the BWT of $\overline{R}$, and the topologies of the two suffix trees of R and $\overline{R}$. It supports all of the four operations in $O(\log\sigma/\log\log n+1)$ time, and uses $(2+1/\epsilon)nH_k(S)+O(n)$ bits.*

Proof. From Lemma 4.3, we can do forward_search and backward_search. From Lemma 4.4, we can compute $st(X)$ and $ed(X)$. From the equations in Lemma 4.5, we can compute $\overline{st}(X)$ and $\overline{ed}(X)$ using the following procedure. For each character a, we compute $(st(aX), ed(aX)) = \text{backward\_search}((st(X), ed(X)), a)$. Then, we substitute the values on the right-hand side of the equations. Since each operation backward_search takes $O(\log\sigma/\log\log n+1)$ time, we can complete the operation *delete_back* in $O(\frac{\sigma\log\sigma}{\log\log n}+1)$ time.

Note that if we use the wavelet tree in [15] as a component of the BWT (as in [78]), the whole term $\sum_{a<c}[ed(aX)-st(aX)+1]$ and $\sum_{c<z}[ed(zX)-st(zX)+1]$ can be computed in $O(\log\sigma/\log\log n+1)$ time, because characters in the alphabet are stored in leaves of the wavelet tree in alphabetic order. By a single traversal from the root of the wavelet tree to the leaf for c, we can compute the total frequency of characters smaller than c. This improvement also applies to the *forward_search* and *backward_search* operations by Lam *et al.* [69] in Lemma 4.3.

The space requirement can be proven by adding up all the requirement of each component. □

4.3.3 A new data structure for a special case of 2D range queries

We now describe the *2D range query* data structure mentioned in Section 4.2 for case 2. This data structure, called $\mathcal{M}$, helps to combine the results of $\mathcal{X}(\overline{T})$ and $\mathcal{Y}(F,T)$ to form the final answers for case 2. Let M be a binary $(s\times m)$-matrix. We define $M[x,y]=1$ if $\overline{T}[x]$ is the preceding factor of the factor suffix $F[y]$. The operation query_2d$(M,[a_1,a_2],[b_1,b_2])$ reports all points in the rectangle $[a_1,a_2]\times[b_1,b_2]$ in M whose values are 1. Here, $[a_1,a_2]$ and $[b_1,b_2]$ specify consecutive rows and consecutive columns of M, respectively. The next lemma summarizes known results for general binary matrices:

Lemma 4.7 (Chan *et al.*[19])**.** *Let M be a given binary matrix of size $m \times m$ with n 1s. M can be stored while supporting* query_2d$(M, [a_1, a_2], [b_1, b_2])$ *as follows:*

1. $O(n \log^{1+\epsilon} m)$ *bits and* $O(\log \log m + occ)$ *query time.*

2. $O(n \log m)$ *bits and* $O((1 + occ) \log^{\epsilon} m)$ *query time.*

where $\epsilon > 0$ is a constant and occ is the number of 1s inside the specified rectangle.

A proof of Lemma 4.7 was given by [19]. In this section, we improve the time for 2D range queries when M has a special form, namely when every column of $M[1..s, 1..m]$ contains exactly one 1. The corollary is as follows.

Corollary 4.8. *Let M be a given binary matrix of size $s \times m$, where $s \leq m$ and every column contains exactly one entry equal to 1. We can store M while supporting* query_2d$(M, [a_1, a_2], [b_1, b_2])$ *within the following space and time complexities:*

1. $O(m \log s \log \log s)$ *bits and* $O((1 + occ) \log \log s)$ *query time; or, alternatively,*

2. $O(m \log s)$ *bits and* $O((1 + occ) \log^{\epsilon} s)$ *query time,*

where $\epsilon > 0$ is a constant and occ is the number of 1s in the specified rectangle.

Proof. Suppose we have access to any data structure for storing general binary matrices of size $(s \times m)$ that uses $O(m \cdot \alpha(m))$ bits space and supports query_2d$(M, [a_1, a_2], [b_1, b_2])$ in $O(\beta(m) + \gamma(m)occ))$ time, where α, β, and γ are polylogarithmic functions (e.g. $\log^k m$), and $\alpha(m)$ is in $\Omega(\log m)$, and $s \leq m$. Then we can construct another data structure for the special case in which each column has exactly one 1 that uses $O(m \cdot \alpha(s))$ bits and with $O(\beta(s) + \gamma(s)occ)$ query time.

Let M be a binary matrix of size $(s \times m)$ in which each column has exactly one 1 and $s \leq m$. We partition M into $\kappa = \frac{m}{s^2}$ vertical blocks of size $s \times s^2$ arranged from left to right. For $\ell = 1, 2, \ldots, \kappa$, define $st_\ell = 1 + s^2(\ell - 1)$ and $ed_\ell = s^2\ell$, and let block ℓ be the submatrix $M[1..s, s_\ell..e_\ell]$. Any query_2d$(P, [a_1, a_2], [b_1, b_2])$ can be classified into one of two types: (i) $st_\ell \leq b_1 \leq b_2 \leq ed_\ell$ for some $\ell = 1, \ldots, \kappa$; and (ii) otherwise.

For (i), the query rectangle lies within a single block and we just use either data structure from Lemma 4.7 for $M[1..s, st_\ell..ed_\ell]$ for $\ell = 1, \ldots, \kappa$. Since every block has s^2 ones, the total space needed to support queries of type (i) in $O(\beta(s) + \gamma(s)occ)$ time is $O(\kappa s^2 \alpha(s^2)) = O(m\alpha(s))$ bits.

To handle queries of type (ii), we store $\binom{s}{2}$ *rank* and *select* data structures for $\binom{s}{2}$ bit vectors $B_{ij}[1..\kappa]$, defined as follows. For $1 \leq i \leq j \leq s$ and $1 \leq \ell \leq \kappa$, let $B_{ij}[\ell] = 1$ if and only if there exists some $M[p,q] = 1$ where $i \leq p \leq j$ and $1 + \frac{s^2}{m}(\ell - 1) \leq q \leq \frac{s^2}{m}\ell$. By Section 1.2.3, the total space to store the *rank* and *select* data structures is $O\left(\binom{s}{2}\kappa\right) = O(m)$ bits. Furthermore, for each $1 \leq \ell \leq \kappa$, we store a list L_ℓ of the s^2 ones in $M[1..s, st_\ell..ed_\ell]$ in sorted order according to their column numbers. We also store s pointers $Ptr_\ell[1..s]$, where $Ptr_\ell[i]$ points to the first entry in the list L_ℓ whose column number is at least i. All lists L_ℓ and Ptr_ℓ can be stored in $O(s^2\kappa \log s) = O(m \log s)$ bits.

Using $B_{ij}[\ell]$, L_ℓ, and Ptr_ℓ, we answer query_2d$(M, [a_1, a_2], [b_1, b_2])$ as follows. Let $st_{i_{min}}$ and $ed_{i_{max}}$ be the smallest st_i and the biggest ed_i such that both of them lie in the interval $b_1..b_2$. Then the answer to the query equals the union of:

(1) query_2d$(M, [a_1, a_2], [b_1, st_{i_{min}} - 1])$;

(2) query_2d$(M, [a_1, a_2], [st_{i_{min}}, ed_{i_{max}}])$; and

(3) query_2d$(M, [a_1, a_2], [ed_{i_{max}}, b_2])$.

Now, (1) and (3) can be computed in $O(\beta(s) + \gamma(s)occ)$ time by querying in inside blocks (case (i) using data structure in Lemma 4.7). For (2), we use the *rank* and *select* data structure for $B_{a_1a_2}$ to find all entries $B_{a_1a_2}[j] = 1$ for $i_{min} \leq j \leq i_{max}$, and for each such j, we report all points in L_j within $Ptr_j[a_1]$ and $Ptr_j[a_2 + 1]$. The running time is $O(1 + occ)$ time.

In conclusion, we can build a data structure of size $O(m\alpha(s))$ bits that supports the operation query_2d$(M, [a_1, a_2], [b_1, b_2])$ on M in $O(\beta(s) + \gamma(s)occ))$ time. Combining this result and Lemma 4.7, we have Corollary 4.8. □

4.4 The data structure $\mathcal{I}(T)$ for case 1

Recall from Section 4.2 that the array $T[1..s]$ stores the s distinct factors of R that occur in the factorizations of $\mathcal{S}$ in lexicographical order. Here, we define a data structure named $\mathcal{I}(T)$ and apply it to locate all occurrences of a query pattern P that lie entirely inside single factors in $T[1..s]$ (case 1 in Section 4.2). The main result of this section is summarized in the following theorem:

Theorem 4.1. *The data structure $\mathcal{I}(T)$ uses $2n+o(n)+O(s\log n)$ bits. Given the suffix range st..ed of a query pattern P in SA_R, it reports all occurrences of P inside factors stored in $T[1..s]$ using $O(occ_1(\log^{\epsilon} n+\log\sigma))$ time, where occ_1 is the number of answers.*

A naive solution is to concatenate all the factors in $T[1..s]$ and then build a suffix tree or an FM-index, but the space used by such an approach would be proportional to the total size of $\mathcal{S}$. Instead, we formulate the problem as a variant of an interval cover problem. Each factor will be represented as an interval on the reference sequence R. The pattern P is first searched in the reference R, then the factors that cover the locations that P occurs at are reported.

Note that we cannot simply enumerate all the occurrences of P in R to find the covering factors. This is because we assume that the reference R may be independent of the sequences $\mathcal{S}$, and there may be occurrences of P in R but not in $\mathcal{S}$ (we call these occurrences *false positives*). The number of false positives of P in R can be $O(n)$. If we enumerate them, the search time cannot be bounded by ℓ and occ_1.

To avoid checking the false positive occurrences of the pattern, we impose an order on the occurrences of P in R. Each location in R is implicitly annotated with the length of the longest factor that covers over it. The searching algorithm prioritizes the occurrences of P in R with longer covers. It stops when the longest possible cover factor is shorter than the pattern length. In this way, we can ensure that the false positive occurrences are not enumerated.

We need the following definitions: for each $i\in\{1,2,\ldots,s\}$, define sp_i and ep_i as the starting and ending positions of the factor $T[i]$ inside the reference string R, i.e., $T[i]=R[sp_i..ep_i]$. We say that any factor $T[i]$ *covers a position* p if $sp_i\le p\le ep_i$. Also, factor $T[i]$ is to the *left of* factor $T[j]$ if either: (1) $sp_i<sp_j$; or (2) $sp_i=sp_j$ and $ep_i<ep_j$. Let $G[1..s]$ be an array of indices such that $G[i]=j$ if $T[j]$ is the i-th leftmost factor. To be able to convert between indices, we define $I_s[j]=sp_{G[i]}$ and $I_e[j]=ep_{G[i]}$. Note that $I_s[1]$ is the starting position of the leftmost factor and that the values of $I_s[1..s]$ are non-decreasing.

Next, for every $p\in\{1,2,\ldots,n\}$, define $D[p]=\max_{j=1..s}\{I_e[j]-p+1 : I_s[j]\le p\}$. Intuitively, $D[p]$ measures the distance from position p to the rightmost ending position of all factors that cover p. We have the following observation. Let $\{p_i\}$ be the set of positions of the occurrences of pattern P in R. For any such p_j, if $D[p_j]>\ell$ (where

ℓ is the pattern length), the occurrence p_j of P is covered by at least one factor. (In other words, position p_j is a true positive occurrence of pattern P.) Therefore, if the $\{p_i\}$ is sorted by the values of $D[p_i]$, the true positive and false positive occurrences can be separated easily.

However, since the occurrences of P in R are already sorted by the order in the suffix array SA_R, we need some additional conceptual structures to remember the $D[p_i]$-order. Let $D'[1..n]$ be an array such that $D'[p] = D[SA_R[p]]$. (For an example, see Fig. 4.6 (a).) $D'[p]$ tells us the length of the longest interval whose starting position equals $SA_R[p]$. Hence, D and D' can be used to filter all false positive occurrences according to the next lemma:

Lemma 4.2. *For any index p and length a, there exists a factor $T[j]$ that covers all positions from $SA_R[p]$ to $(SA_R[p] + \ell - 1)$ in R if and only if $D'[p] \geq \ell$.*

Proof. (Necessary condition) If $T[j]$ covers $SA_R[p]$, then $D[SA_R[p]] \geq |T[j]|$. We also have that $T[j]$ covers $(SA_R[p] + a - 1)$; therefore, $|T[j]| \geq a$. That means $D[SA_R[p]] \geq a$. By the definition of D', we have $D'[p] \geq a$.
(Sufficient condition) Follows directly from the definitions of D and D'. □

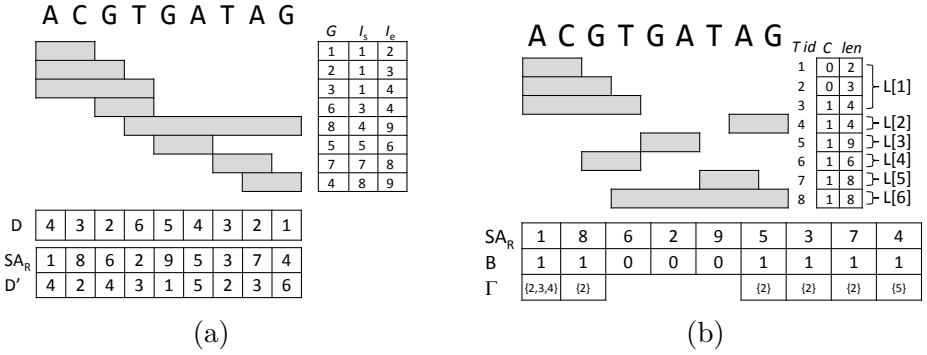
Now, we describe the new data structure $\mathcal{I}(T)$. It consists of:

- The array $G[1..s]$, using $s \log n$ bits;
- A successor data structure (see Section 4.3) for I_s, using $s \log n + o(n)$ bits;
- A range maximum data structure (see Section 4.3) for I_e, using $2s + o(s)$ bits; and
- A range maximum data structure for D', using $2n + o(n)$ bits.

Note that we do not explicitly store the arrays $D[1..n]$, $D'[1..n]$, $I_s[1..s]$, and $I_e[1..s]$. Lemma 4.3 shows how to recover the values of $D[p]$ and $D'[p]$ for any position $p \in \{1, 2, \ldots, n\}$ from the data structure $\mathcal{I}(T)$. Also, $I_s[i]$ and $I_e[i]$ can be computed in $O(1)$ time given $G[i]$ and T.

Lemma 4.3. *Given the data structures $\mathcal{I}(T)$ and the FM-index of R, for any positions p and q in R, we can:*

(i) Compute $D[p]$ in $O(1)$ time;



(a) (b)

Figure 4.6: (a) The factors (displayed as grey bars) from the example in Fig. 4.2 listed in left-to-right order, and the arrays G, I_s, I_e, D, and D' that define the data structure $\mathcal{I}(T)$ in Section 4.4. (b) The same factors ordered lexicographically from top to bottom, and the arrays B, C, and Γ that define the data structure $\mathcal{X}(\overline{T})$ in Section 4.5.

(ii) Compute $D'[p]$ in $O(\log^{\epsilon}_{\sigma} n + \log \sigma)$ time; and

(ii) Report all factors that cover positions $p..q$ in $O(1 + occ)$ time.

Proof. For (i), using the successor data structure for I_s, we can identify the maximum y such that $I_s[y] \leq p$ in $O(1)$ time. Using the range maximum data structure for I_e, we can identify an index v such that $I_e[v] = \max_{j \leq y} I_e[j]$ in $O(1)$ time. Then, $D[p] = I_e[v] - p + 1$. For (ii), $SA_R[p]$ can be computed in $O(\log^{\epsilon} n + \log \sigma)$ time by Lemma 4.1 in Section 4.3, so $D'[p] = D[SA_R[p]]$ can be computed in the same time.

For (iii), it is obvious that all the factors that cover both p and q need to start at a position less than or equal to p. Among them, the factors ending at q or to the right of q are those that need to be reported. Formally, the set of answers is $\{T[G[i]] : I_s[i] \leq p$ and $q \leq I_e[i]\}$.

This is the 2-sided range query problem. We first find the maximum index y such that $I_s[y] \leq p$. Since I_s is non-decreasing, the problem becomes reporting every value i such that $i \leq y$ and $q \leq I_e[i]$. This problem can be handled by the maximum data structure (Section 1.2.3) for I_e . □

Based on $\mathcal{I}(T)$ and the suffix range for the query pattern P, Algorithm Search_Pattern in Fig. 4.7 finds all occurrences of P in factors from $T[1..s]$. Basically, it checks the occurrences $\{p_i\}$ of P in R based on the order of $D[p_i]$ from bigger to smaller, and stops when $D[p_i] < \ell$.

In Fig. 4.7, the value p_i is implicitly represented by $SA_R[q]$ i.e. $p_i = SA_R[q]$. Let $st..ed$ be the suffix range of P in SA_R. In line 1, the algorithm finds an index q from

Algorithm Search_Pattern(st, ed)
Input: The data structure $\mathcal{I}(T)$, the FM-index of R and the suffix range $st..ed$ of the pattern P in SA_R.
Output: Every factor $T[j]$ in which P occurs.

1: Compute $q = \text{max\_index}_{D'}(st, ed)$
2: **if** $D'[q] \geq \ell$ **then**
3: Report all factors that cover $SA_R[q]..(SA_R[q] + \ell - 1)$ using Lemma 4.3
4: Search_Pattern($st, q - 1$)
5: Search_Pattern($q + 1, ed$)
6: **end if**

Figure 4.7: Algorithm for computing all occurrences of P in $T[1..s]$.

the range $st \leq q \leq ed$, such that $D[SA_R[q]]$ has the biggest value. The condition $D'[q] \geq |P|$, in line 2, guarantees that $SA_R[q]$ and $SA_R[q] + \ell - 1$ are covered by at least one factor (where ℓ is the length of the pattern). Since $st \leq q \leq ed$, it holds that $R[SA_R[q]..(SA_R[q] + \ell - 1)]$ is an occurrence of P in R. Then, the line 3 of the algorithm reports every $T[j]$ that contains P by using Lemma 4.3. Finally, the algorithm recursively finds smaller values from its sub-ranges in line 4 and line 5.

4.5 The data structure $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$ for case 2

We now turn our attention to case 2 in Section 4.2 (see Fig. 4.4 (b)). This section gives the details of two symmetric data structures $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$. For any given pattern P, $\mathcal{X}(T)$ ($\mathcal{X}(\overline{T})$) locates every occurrence of a suffix (prefix) of P that equals a prefix (suffix) of a factor of $\mathcal{S}$. (See Fig. 4.8.). Data structure $\mathcal{X}(\overline{T})$ is used to find the left part of the pattern in our searching algorithm outlined in Section 4.2.2. Data structure $\mathcal{X}(T)$ is used as a component in the data structure $\mathcal{Y}(F, T)$ to find the right part in Section 4.6. To simplify the presentation, we only describe $\mathcal{X}(T)$ below.

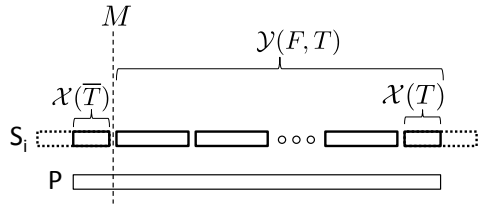


Figure 4.8: Data structures used in case 2

Our solution considers every non-empty suffix of P as a separate query pattern

for $\mathcal{X}(T)$. For each suffix Q, we assume that Q is specified by the corresponding suffix range $st_Q..ed_Q$ in the suffix array SA_R for the reference string R, along with the length of Q. Since $T[1..s]$ stores all distinct factors S_{ij} in lexicographically sorted order, all occurrences of Q in $\mathcal{S}$ can be represented as a range $p..q$ in T such that Q is a prefix of every element in $T[p], \ldots, T[q]$. The theorem and corollary below summarize the data structures $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$.

Theorem 4.1. *The data structure $\mathcal{X}(T)$ uses $O(s \log n) + o(n)$ bits. For any suffix range $st..ed$ in SA_R of a query pattern P, it can report the maximal range $p..q$ such that P is a prefix of all $T[j]$, where $p \leq j \leq q$, in $O(\log \log n)$ time.*

Corollary 4.2. *The data structure $\mathcal{X}(\overline{T})$ uses $O(s \log n) + o(n)$ bits. For any suffix range $st..ed$ in $SA_{\overline{R}}$ of a query pattern P, it can report the maximal range $p..q$ such that $\overline{P}$ is a prefix of all $\overline{T}[j]$, where $p \leq j \leq q$, in $O(\log \log n)$ time.*

A simple solution for this problem is to build a trie of all the factors of $\mathcal{S}$. However, such a data structure requires too much space. In this section, we observe a mapping between the lexicographically sorted order of the factors (stored in the array $T[1..s]$) and the suffix array of reference sequence in Lemma 4.3. Based on this mapping, to find if one pattern is a prefix of any factor, we search for the pattern in the reference suffix array SA_R, and then calculate the mapping using Lemma 4.4 in $O(\log \log n)$ time to extract all factors.

To start, we need some efficient way to check if the query pattern P is a prefix of any specified factor $T[j]$. Since the factor $T[j]$ is a substring of R, let $st_j..ed_j$ denote the corresponding suffix range of $T[j]$ in SA_R. The next lemma says how their suffix ranges are related:

Lemma 4.3. *Suppose $st_P..ed_P$ is the suffix range of P in SA_R. P is a prefix of $T[j]$ if and only if either: (1) $st_P < st_j \leq ed_P$; or (2) $st_P = st_j$ and $|T[j]| \geq |P|$.*

Proof. We use the following property of the suffix array: Given a suffix array SA_R, consider two strings x and y such that $|x| < |y|$. Let st_x and ed_x be the suffix range of x in SA_R. Let st_y and ed_y be the suffix range of y. If x is prefix of y, then $st_x \leq st_y \leq ed_y \leq ed_x$. Otherwise, (st_x, ed_x) and (st_y, ed_y) are disjoint.

($\rightarrow$) By the property of the suffix array, if P is prefix of $T[j]$, then the suffix range of

$T[j]$ is inside the suffix range of P in SA_R. That is $st_P \leq st_j \leq ed_j \leq ed_P$. In addition, since P is prefix of $T[j]$, we have $|P| \leq |T[j]|$. That means condition (1) or (2) is correct.

($\leftarrow$) If condition (1) is true, i.e. $st_P < st_j \leq ed_P$, then $ed_j \leq ed_P$. (Otherwise it will violate the property of the suffix array.) Since P is equals the share prefixes of all $R[SA_R[st_P]..n] \ldots R[SA_R[ed_P]..n]$ and $T[j]$ is the share prefix of $R[SA_R[st_j]..n] \ldots R[SA_R[ed_j]..n]$. Since the range of $T[j]$ is strictly inside the range of $T[j]$, the length of P is strictly less than the length of $T[j]$. Therefore, P is a proper prefix of $T[j]$.

If condition (2) is true, we have $st_P = st_j$. Thus, either P is prefix of $T[j]$ or $T[j]$ is prefix of P. However, we also have $|T[j]| \geq P$; therefore, P is a prefix of $T[j]$. □

For every $i = 1, \ldots, n$, define $\Gamma(i) = \{|T[j]| : st_j = i$ and $st_j..ed_j$ is the suffix range of $T[j]$ in $SA_R\}$. In other words, $\Gamma(i)$ is the set of lengths of factors whose suffix ranges start at i in SA_R. We use $\Gamma(i)$ to map a suffix range in SA_R to a range of factors in T according to:

Lemma 4.4. *Suppose $st_P..ed_P$ is the suffix range of P in SA_R. Then, $p..q$ is the range in $T[1..s]$ such that P is a prefix of all $T[j]$ where $p \leq j \leq q$, where $p = 1 + \sum_{i=1}^{st_P - 1} |\Gamma(i)| + |\{x \in \Gamma(st_P) : x < |P|\}|$ and $q = \sum_{i=1}^{ed_P} |\Gamma(i)|$.*

Proof. By the definition of $\Gamma(i)$, for every $T[j]$ such that $1 + \sum_{i=1}^{st_P - 1} |\Gamma(i)| \leq j \leq \sum_{i=1}^{ed_P} |\Gamma(i)|$, we have $st_P \leq st_j \leq ed_P$. If $st_P < st_j \leq ed_P$, condition (1) in Lemma 4.3 holds. Otherwise, $st_P = st_j$. However, $p = 1 + \sum_{i=1}^{st_P - 1} |\Gamma(i)| + |\{x \in \Gamma(st_P) : x < |P|\}|$. All the $T[j]$'s with length less than $|P|$ are not included; therefore, condition (2) in Lemma 4.3 holds. □

Now, we present the data-structure $\mathcal{X}(T)$ based on Lemma 4.4. First, let $B[1..n]$ be a bit vector such that $B[i] = 1$ if $\Gamma(i)$ is non-empty, and $B[i] = 0$ otherwise. Next, suppose $\Gamma(i)$ is the r-th non-empty set, and let $L[r]$ be a y-fast trie [113] for $\Gamma(i)$ (see Section 4.3). Let $C[1..s]$ be a bit vector such that $C\left[\sum_{i=1}^{r} |\Gamma(i)|\right] = 1$, and 0 otherwise. See Fig. 4.6 (b). The data structure $\mathcal{X}(T)$ consists of three parts: (i) The *rank* data structure for the bit vector $B[1..n]$ ($s \log n + o(n)$ bits); (ii) The *select* data structure for the bit vector $C[1..s]$ ($s \log n + o(n)$ bits); and (iii) The y-fast trie data structure $L[r]$ for $\Gamma(i)$ if $\Gamma(i)$ is the r-th non-empty set ($O(s \log n)$ bits). In total, $\mathcal{X}(T)$ requires $O(s \log n) + o(n)$ bits.

Note that, for any ℓ, we have

$$\sum_{i=1}^{\ell} |\Gamma(i)| = \text{select}_C(\text{rank}_B(\ell))$$

$$|\{x \in \Gamma(\ell) : x < c\}| = \text{successor\_index}(L[rank_B(\ell)], c)$$

Using $\mathcal{X}(T)$, they can be computed in $O(\log \log n)$ time. Hence, the values of p and q in Lemma 4.4 can be computed in $O(\log \log n)$ time. Theorem 4.1 follows.

4.6 The data structure $\mathcal{Y}(F, T)$ for case 2

This section outlines our solution for finding the occurrences for the right part of the pattern. For each suffix $P[i..\ell]$ for $1 \leq i \leq \ell$ of the pattern P, we need to check if $P[i..\ell]$ is the prefix of some factor suffix in $\mathcal{S}$ (see Fig. 4.2.2 in Section 4.2.2).

Solving this problem is not too complicated. Since any factor suffix has a unique RLZ factorization; given one pattern P', we can factorize the pattern using the reference, i.e., $RLZ(P'|R)$; then, match all the factorizations generated from the pattern with those sequences of $\mathcal{S}$. However, in this problem, all the suffixes $P[i..\ell]$ needs to be matched against $\mathcal{S}$. If we treat each suffix as an independent query pattern, it would take $O(\ell^2)$ time to answer all the queries. In this section, we present our approach to reuse the factorization and matching information between the suffixes to speed up the whole process. Briefly, each suffix of P is represented by a suffix range in the factor suffix array F. We factorize the suffixes of the pattern and match them with the database sequences in one run from right to left using dynamic programming.

To be precise, we build a data structure $\mathcal{Y}(F, T)$ which for any pattern P of length ℓ can compute the range of $P[i..\ell]$ in F for all $1 \leq i \leq \ell$, i.e., the range $st..ed$ in F where $P[i..\ell]$ is a prefix of $F[st], \ldots, F[ed]$. Let $Q[i]$ denote the range for each i. The following theorem summarizes the main result:

Theorem 4.1. *The data structure $\mathcal{Y}(F, T)$ uses $O(n) + (2 + 1/\epsilon)nH_k(R) + o(n \log \sigma) + O(m \log n)$ bits. It can find all suffix ranges of F that match some suffix of a query pattern P of length ℓ in $O(\ell(\log \sigma / \log \log n + \log \log n))$ time.*

First, there are two sub-cases for our pattern in this section (see Fig. 4.9): (1) the whole suffix $P[i..\ell]$ is a prefix of some factors; and (2) suffix $P[i..\ell]$ contains at least

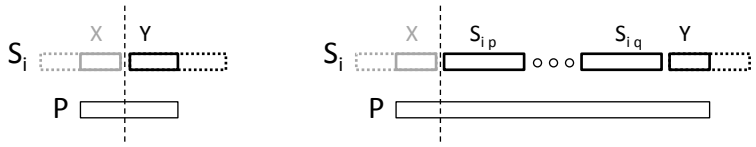


Figure 4.9: Two sub-cases

one factor inside and a tail which is a prefix of some factors. Solving the first sub-case is straightforward (since we can use data structure $\mathcal{X}(T)$ in Section 4.5). The second sub-case can be simplified based on the observation that the matched factors are unique. The properties of the two sub-cases are summarized in the following lemma:

Lemma 4.2. *For any $F[i']$, define the* head of $F[i']$ *to be the first factor of $F[i']$. For any $1 \leq i \leq \ell$, if $P[i..\ell]$ is prefix of $F[st], \ldots, F[ed]$, then P and $st..ed$ satisfy either one of the following properties:*

- *(1) $P[i..\ell]$ is the prefix of the heads of all factor suffixes $F[st], \ldots, F[ed]$.*
- *(2) The heads of all $F[st], \ldots, F[ed]$ are prefix of $P[i..\ell]$. In fact, these heads are the same; and equal $P[i..j]$ where j is the biggest index such that $P[i..j]$ is a factor of $\mathcal{S}$.*

Proof. Denote $P[i..\ell]$ by P_i for short. Assume P_i is the prefix of two factor suffix $F[x]$ and $F[y]$. Let $X = T[x']$ and $Y = T[y']$ are the head of $F[x]$ and $F[y]$ respectively. Because x and y are symmetrical, without loss of generality, we just consider two cases $|X| = |Y|$ and $|X| < |Y|$. Assume $|X| < |Y|$, we have the following sub-cases: (a) X is prefix of Y and Y is prefix of P_i. (b) X is prefix of P_i and P_i is prefix of Y. (c) P_i is prefix of both X and Y.

We will prove that sub-cases (a) and (b) cannot be true. Therefore, only sub-case (c) or $|X| = |Y|$ happens which leads to only cases (1) and (2) of the Lemma.

In both sub-cases (a) and (b), factor X is a prefix of Y. Let c be the character in position $(|X| + 1)$ of $F[x]$. Due to the maximal property of the encoding in Section 4.2.1, $X \cdot c$ does not occurs in the reference R. But, Y must have some occurrence in R; therefore, the character at position $(|X| + 1)$ of $F[x]$ is different from the character at the same position of $F[y]$. However, P_i matches the position $(|X| + 1)$ of both $F[x]$ and $F[y]$ in these sub-cases, and therefore, it is a contradiction. □

Lemma 4.2 gives an important property of maximally factored suffixes. Namely, if a

pattern matches the prefixes of two factor suffixes, the list of factors in the two prefixes are identical, except for the last factor whose prefix matches a suffix of the pattern. In other words, the factorization of $P[i..\ell]$ only depends on the factorization of one other suffix $P[i+a..\ell]$ (for some value a that can be computed). From this observation, we obtain the following.

Let $\mathbb{S}$ be the concatenation of the factorizations of all strings in $\mathcal{S}$, and let $\mathcal{B}$ be a general FM-index of $\mathbb{S}$ (Section 4.3) that supports $\mathit{backward\_search}_{\mathbb{S}}(T[i], (st, ed))$. The array $Q[i]$ can be computed as follows. Define $A[i] = P[i..j]$, where j is the largest index such that $P[i..j]$ is a factor of $\mathcal{S}$, if one exists, and *nil* otherwise. Let $Y[i]$ be the range $st..ed$ in F such that $P[i..\ell]$ is the prefix of all the heads of factor suffixes $F[st]..F[ed]$, if one exists, and *nil* otherwise.

Informally, each entry $Y[i]$ stores the result of the sub-case (1). Each entry $A[i]$ of array A stores the trace of a possible factorization for suffix $P[i..\ell]$. Then, array $Q[1..\ell]$ can be computed by dynamic programming based on the following equation:

$$Q[i] = \begin{cases} Y[i] & \text{if } Y[i] \neq \mathit{nil} \\ \text{backward\_search}_{\mathbb{S}}(A[i], Q[i + |A[i]|]) & \text{if } Y[i] = \mathit{nil} \ \&\ A[i] \neq \mathit{nil} \\ \mathit{nil} & \text{otherwise} \end{cases} \tag{4.1}$$

By Equation (4.1), $Q[1..\ell]$ can be computed in three steps:

(a) Compute $A[i]$ for $i = 1$ to ℓ;

(b) Compute $Y[i]$ for $i = \ell$ to 1; and

(c) Compute $Q[i]$ for $i = \ell$ to 1.

Next, we present the data structure $\mathcal{Y}(F, T)$ and discuss steps (a)–(c). The data structure $\mathcal{Y}(F, T)$ consists of:

- The bi-directional BWT (see Section 4.3.2).
- The data structure $\mathcal{X}(T)$ (see Section 4.5).
- The *select* data structure for a bit-vector $V[1..m]$, defined by $V[i] = 1$ if the head of $F[i]$ differs from the head of $F[i+1]$, and $V[i] = 0$ otherwise.
- The general FM-index $\mathcal{B}$ of $\mathbb{S}$.

First, we discuss step (a). Fig. 4.10 gives the algorithm to compute $A[1..\ell]$. Lemma 4.3 presents the correctness of the time complexity of the algorithm.

Lemma 4.3. *We can compute all* $A[1..\ell]$ *in* $O(\ell(\log\sigma/\log\log n + \log\log n))$ *time.*

Proof. We apply the bi-directional FM-index (see Section 4.3.2) to compute $A[1..\ell]$, as shown in Fig. 4.10:

The inner loop (lines 4–7) of the algorithm extends the search sequence to the maximal length. The outer loop (lines 3–11) assigns a value to $A[i]$ and deletes the first character to move to the next position. To check any factor in $\mathcal{X}(T)$ takes $O(\log\log n)$ time. The alphabet is of constant size, so the time for every forward_search and delete_back operation is $O(\log\sigma/\log\log n)$. Thus, each $A[i]$ is obtained in $O(\log\sigma/\log\log n + \log\log n)$ time. □

```
 1: Let r_R and r_R̄ be suffix ranges of the empty string ε in SA_R and SA_R̄.
 2: j = 1
 3: for i = 1 to |P| do
 4:    while j ≤ |P| and the last forward search succeeded do
 5:       r_R, r_R̄ = forward_search(r_R, r_R̄, P[j])
 6:       j = j + 1
 7:    end while
 8:    if r_R is a factor according to 𝒳(T) then let A[i] = the factor found by 𝒳(T)
 9:    else let A[i] = nil
10:    r_R, r_R̄ = delete_back(r_R, r_R̄)
11: end for
```

Figure 4.10: Algorithm to fill in the array $A[1..|P|]$.

In step (b), we compute $Y[1..\ell]$ in two phases. The first phase computes another array $Y'[1..\ell]$, defined as follows: $Y'[i]$ is the range $st'..ed'$ in T such that $P[i..\ell]$ is the prefix of $T[st'], \ldots, T[ed']$. By using the $\mathcal{X}(T)$ data structure from Section 4.5, we can obtain $Y'[1..\ell]$. Then, given $Y'[1..\ell]$, the second phase computes $Y[1..\ell]$ with the *select* data structure for V as follows: $Y[i] = (\text{select}_V(st-1)+1, \text{select}_V(ed))$, where $(st, ed) = Y'[i]$. Finally, in step (c), we apply Equation (4.1) to compute $Q[1..\ell]$. The total running time is therefore $O(\ell(\log\sigma/\log\log n + \log\log n))$.

The data structure $\mathcal{X}(T)$ uses $O(s\log n) = O(m\log n)$ bits. The bi-directional BWT uses $(2+1/\epsilon)nH_k(R) + O(n)$ bits. The general FM-index $\mathcal{B}$ requires $O(m\log s) = O(m\log n)$ bits. The *select* data structure on bit-vector V is implemented using $O(m)$ bits. Thus, Theorem 4.1 follows.

Suf. id	Seg. suffix	V	BWT
$F[1]$	1 7	1	6
$F[2]$	2	1	8
$F[3]$	3	1	6
$F[4]$	4 5	1	$
$F[5]$	5 6 1 7	1	$
$F[6]$	5	0	4
$F[7]$	6 1 7	1	5
$F[8]$	6 3	0	$
$F[9]$	7	1	1
$F[10]$	8 2	1	$

(a)

		$F[1]$	$F[2]$	$F[3]$	$F[4]$	$F[5]$	$F[6]$	$F[7]$	$F[8]$	$F[9]$	$F[10]$
$	$				1	1			1		1
$\overline{T}[1]$	$T[5]$							1			
$\overline{T}[2]$	$T[7]$										
$\overline{T}[3]$	$T[1]$									1	
$\overline{T}[4]$	$T[4]$						1				
$\overline{T}[5]$	$T[8]$		1								
$\overline{T}[6]$	$T[2]$										
$\overline{T}[7]$	$T[6]$	1		1							
$\overline{T}[8]$	$T[3]$										

(b)

Figure 4.11: (a) The array $F[1..m]$ consists of the factor suffixes $S_{ip}S_{i(p+1)}\dots S_{ic_i}$, encoded as indices of $T[1..s]$. Also shown in the table is a bit vector V and BWT-values, defined in Section 4.6. (b) For each factor suffix $F[j]$, column j in M indicates which of the factors that precede $F[j]$ in $\mathcal{S}$. To search for the pattern P = AGTA, we need to do two 2D range queries in M: one with $st = 1$, $ed = 2$, $st' = 7$, $ed' = 8$ since A is a suffix of $T[5]$ and $T[7]$ (i.e., a prefix in $\overline{T}[1..2]$) and GTA is a prefix in $F[7..8]$, and another one with $st = 4$, $ed = 4$, $st' = 9$, $ed' = 9$ since AG is a suffix of $T[4]$ (i.e., a prefix in $\overline{T}[4]$) and TA is a prefix in $F[9]$.

4.7 Decoding the occurrence locations

Recall that given strings $\mathcal{S} = \{S_1, S_2, \dots S_t\}$, we decompose each S_i into factors. The substring from the start of a factor to the end of the string is called factor suffix. One factor may occur at multiple locations of the set of strings $\mathcal{S}$, but every factor suffix has a unique location in $\mathcal{S}$. All the distinct factors are represented in the array $T[1..s]$. The sorted order of the factor suffixes is represented in the array $F[1..m]$.

The result of case 1 of our algorithm is a set of factors such that P is a substring of them. Since each factor in this set can have multiple locations in $\mathcal{S}$, the first problem reports, for an index p of T, all the locations in $\mathcal{S}$ that factor $T[p]$ occurs at.

The result of case 2 is a set of factor suffixes represented in F such that a suffix of P is the prefix of these factor suffixes. The second problem reports, for an index p of F, the unique location in $\mathcal{S}$ that the factor suffix $F[p]$ occurs at. We design a pipeline with 3 phases to resolve cases 1 and 2.

- Phase (I): Given an index p of T, return a set of indices $\{p'\}$ such that $T[p]$ equals the first factor of each $F[p']$.

- Phase (II) computes relative locations in $\mathcal{S}$ for a factor suffix in F:

  Given an index p of F, return i, j such that $F[p]$ starts at S_{ij} in $\mathcal{S}$.

- Phase (III) converts the relative locations in $\mathcal{S}$ to the exact location in $\mathcal{S}$:

  Given i, j, return $1 + \sum_{q=1}^{j-1} |S_{iq}|$, i.e., the starting location of S_{ij} in the input string

S_i.

To obtain the results for case 1, we apply all 3 phases. For case 2, we only apply phases (II) and (III).

Phase (I) can be done using the $\mathcal{Y}(F,T)$ data structure in $O(1+occ)$ time. Phase (II) can be done by decoding the general FM-index with $\mathcal{Y}(F,T)$ in $O(1+occ\cdot\log m/\log s)$ time.

Phase (III) is described next. The idea is to compute the position of S_{ij} in the string that is the concatenation of $S_1,\ldots,S_t$ and then convert it to the position in S_i. Let $L[1..s]$ be an array storing the lengths of all factors in the order of occurrences in the concatenated string, that is, the length of factor S_{ij} is stored in entry $L[\sum_{i'=1}^{i-1} c_{i'}+j]$. Let $C[0..s]$ be a bit vector where $C[0]$ is set to 1, and $C\left[\sum_{i'=1}^{i} c_{i'}\right]$ are set to 1 for all $i=1,\ldots,N$ where $c_{i'}$ is the number of factors in $S_{i'}$. (Thus, C encodes the indices in L of heads of factors.)

To implement phase (III), we store: the prefix sum data structure for L and the *select* data structure for C. The location of S_{ij} in S_i is obtained as follows. First, compute $s=\mathrm{select}_C(i)$. Then, the value of $1+\sum_{q=1}^{j-1}|S_{iq}|$ is given by $1+\mathrm{prefix\_sum}_L(s+j-1)-\mathrm{prefix\_sum}_L(s)$.

Lemma 4.1. *Phase (III) runs in $O(occ\cdot\log\log n)$ time and uses $O(m\log n)$ bits.*

Proof. The array L has m elements and the sum of all of them is at most mn. Based on [25], the space for the prefix sum data structure of L is $O(m\log(mn/m))+O(m)=O(m\log n)$ bits. Because the length of C is at most m, the *select* data structure for C uses at most $O(m)$ bits. Therefore, the total size of this data structure is $O(m\log n)$ bits.

The $\mathrm{prefix\_sum}_L$ operation in L takes $O(\log\log n)$ time, and the select_C operation in C takes $O(1)$ time. □

Chapter 5

Conclusions

Due to recent improvements in sequencing throughput, indexing data structures are becoming an essential tool for DNA sequence analysis. In this thesis, we study a few compressed indexing data structures in regard to sequence similarity in biological sequences. The first work is a data structure with application in sequence alignment. The successive works explore compressed structures for storing similar sequences with fast pattern searching. The detail technical contributions are summarized as follows.

Our first contribution is to introduce the first full-functional compressed version of directed acyclic word graph (DAWG). In this work, by observing a close relationship between DAWG and existing compressed data structures namely suffix tree and FM-index, we developed algorithms to emulate operations on DAWG using components of the existing structures. The structure uses $nH_k(\overline{S}) + 2nH_0^*(\mathcal{T}_{\overline{S}}) + o(n)$ bits and supports the DAWG operations in at most $O(\log n)$ time. In addition, we also applied our DAWG data structure to speed up the computation of local alignment, a key biological sequence similarity measurement method. Precisely, we develop an algorithm to compute the meaningful alignment between a query and a database sequence indexed by DAWG. Compared to previous works, this method improves the running time when the query has many matches with the database sequence. That leads to an improvement in the worst case bound while keeping the good average case bound in the random input case.

Our second contribution is the introduction of two new data structures for a set of similar sequences called multi-version rank/select and multi-version FM-index. These data structures model the changes between the sequences by storing only the inserted and deleted characters between each pair of sequences. This scheme gives an effective

compression when the sequences are long, and each sequence is hard to compress. The multi-version rank/select data structure requires $|S|H_k(S) + 2m(\log m + \log n) + o(n \log \sigma + m(\log m + \log n))$ bits, and answers the rank/select queries in $O(\log \log \sigma + \log m / \log \log m)$ time where m is the number of changes, σ is the size of the alphabet, and S is a sequence that consists of the characters from the first sequence and the inserted characters. The multi-version FM-index uses $|S|H_k(S) + O(m \log^2(m+n)) + o(n \log \sigma)$ bits, and finds pattern P in $O(|P|(\log \log \sigma + \log m))$ time.

Our third contribution is a novel indexing data structure for RLZ compression scheme for a set of similar sequences. Consider a set of similar sequences $\mathcal{S}$, and a reference sequence R of length n over a moderate alphabet of size σ. Let m be the smallest possible number of substrings of R to represent $\mathcal{S}$. The data structure takes $\left(2 + \frac{1}{\epsilon}\right) nH_k(R) + O(n) + O(m \log n)$ bits. All exact occurrences of any query pattern P of length ℓ can be reported within $O(\ell \log^{\epsilon} n + occ \cdot (\log_{\sigma}^{\epsilon} n + \frac{\log m}{\log n}))$ time where occ is the number of occurrences of P, and $\epsilon \leq 1$ is a constant. Using additional $O(m \log n \log \log n)$ bits, the query time can be reduced to $O(\ell \log \log n + occ \cdot (\log_{\sigma}^{\epsilon} n + \frac{\log m}{\log n}))$.

Besides the specific contributions mentioned above, we also improve some existing data structures and propose new supporting structures for the design of the main indexes. In section 3.3, we present a succinct version of the k-th line cut data structure. This data structure is used to store a set of vertical lines and supports a query that finds the k-th cut of these lines with a horizontal ray. For the regular case, we improve the space by a factor of $\log n$ and query time by a factor of $\log \log n$; and for certain inputs, the bound can be further reduced. In Section 4.3, we improve the bi-directional FM-index which is used in DNA short read mapping [69] and RNA structure patterns searching [84]. We add new operations and improve query time of existing operations from $O(\sigma \log \sigma / \log \log n)$ to $O(\log \sigma / \log \log n)$. Section 4.3 also provides an improvement for a restricted type of 2D range query data structure when the input is asymmetry.

For future directions, there are a number of interesting questions regarding similarity measurement and indexes for this type of data. First, as discussed in Chapter 4, the empirical entropy measurement H_k which often uses for benchmarking traditional compressed structures cannot reflect accurately the amount of redundancy in similar sequences. Currently, each model of similarity gives rise to a different measurement and representation method. For example, in this thesis, we work on delta compression for

insertions/deletions and RLZ compression. However, the indexes based on different compressions are hard to compare. Therefore, more research is needed to better understand and unify the concept of sequence similarity. Secondly, future works are required to explore the space-time trade-off of the indexing data structure and new operations and of the current indexes. For example, the current bounds for pattern searching in RLZ index is very close to linear of the pattern length. However, we still do not know whether it is possible to reduce the searching time without scarifying too much space. Besides, multi-version FM-index and multi-version rank/select can be extend to handle sequences with relationship that forms a evolutionary tree. Last but not least, this thesis consists of mostly theoretical results; we wish to further work on some simplified but practical implementations of these results.

Bibliography

[1] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] A. Apostolico. The myriad virtues of subword trees. 1985.

[4] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *DCC*, pages 143–152, 2000.

[5] D. Arroyuelo and G. Navarro. Space-efficient construction of lempel-ziv compressed text indexes. *Information and Computation*, 209(7):1070–1102, 2011.

[6] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *CPM*, volume 4009 of *LNCS*, pages 318–329, 2006.

[7] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger lempel-ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101, 2012.

[8] R.A. Baeza-Yates and G.H. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *In Proc. of SPIRE*, 1999.

[9] M. Barsky, U. Stege, and A. Thomo. A survey of practical algorithms for suffix tree construction in external memory. *Software: Practice and Experience*, 40(11):965–988, 2010.

[10] M. Barsky, U. Stege, and A. Thomo. Suffix trees for inputs larger than main memory. *Information Systems*, 36(3):644–654, 2011.

[11] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 785–794, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[12] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *ESA*, pages 181–192, 2012.

[13] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *SODA*, pages 373–389, 2011.

[14] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, MT Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[15] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *WADS*, volume 5664 of *LNCS*, pages 98–109, 2009.

[16] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, and E. Davydov. Lagan and multilagan: efficient tools for large-scale multiple alignment of genomic dna. *Genome research*, 13(4):721–731, 2003.

[17] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[18] M. D. Cao, T. I. Dix, L. Allison, and C. Mears. A simple statistical algorithm for biological sequence compression. In *DCC*, pages 43–52, 2007.

[19] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *SoCG*, pages 1–10, 2011.

[20] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. In *RECOMB*, page 107, 2000.

[21] S. Christley, Y. Lu, C. Li, and X. Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 2009.

[22] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *MFCS*, volume 5734 of *LNCS*, pages 235–246, 2009.

[23] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *SPIRE*, pages 180–192. Springer, 2012.

[24] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. *LNCS*, 1261:192–211, 1997.

[25] O. Delpratt, N. Rahman, and R. Raman. Compressed prefix sums. In *SOFSEM*, 2007.

[26] Paul F. Dietz. Fully persistent arrays. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, pages 67–74, London, UK, UK, 1989. Springer-Verlag.

[27] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative Lempel-Ziv self-index for similar sequences. In *FAW-AAIM*, pages 291–302, 2012.

[28] D.P. Dobkin and J.I. Munro. Efficient uses of the past. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 200–206. IEEE, 1980.

[29] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, April 1974.

[30] R.M. Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[31] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000.

[32] J. Fayolle and M.D. Ward. Analysis of the average depth in a suffix tree under a markov model. In *International Conference on Analysis of Algorithms DMTCS proc. AD*, volume 95, page 104, 2005.

[33] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *LATIN 2010: Theoretical Informatics*, 6034:697–710, 2010.

[34] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13:12, 2009.

[35] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM (JACM)*, 46(2):236–280, 1999.

[36] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, page 390, 2000.

[37] P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler Transform. In *SODA*, pages 655–663, 2004.

[38] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, July 2005.

[39] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *SPIRE*, pages 150–160, 2004.

[40] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, May 2007.

[41] J. Fischer. Wee LCP. *Information Processing Letters*, 110:317–320, 2010.

[42] J. Fischer. Combined data structure for previous-and next-smaller-values. *Theoretical Computer Science*, 412:2451–2456, 2011.

[43] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE*, volume 4614 of *LNCS*, pages 459–470, 2007.

[44] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[45] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 1–7. ACM, 1990.

[46] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. Puglisi. A faster grammar-based self-index. *Language and Automata Theory and Applications*, 7183:240–251, 2012.

[47] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.

[48] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. *Information retrieval: data structures and algorithms*, pages 66–82, 1992.

[49] R. González and G. Navarro. Compressed text indexes with fast locate. In *CPM*, pages 216–227. Springer, 2007.

[50] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. *LATIN 2008: Theoretical Informatics*, 374-386:374–386, 2008.

[51] R. González and G. Navarro. A compressed text index on secondary memory. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 71:127, 2009.

[52] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850. SIAM, 2003.

[53] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *STACS*, pages 517–528, 2009.

[54] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406. ACM, 2000.

[55] S. Grumbach and F. Tahi. Compression of DNA sequences. In *DCC*, pages 340–350, 1993.

[56] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[57] W.K. Hon, R. Shah, and J.S. Vitter. Compression, indexing, and retrieval for massive string data. In *Proceedings of the 21st annual conference on Combinatorial pattern matching*, CPM'10, pages 260–274. Springer-Verlag, 2010.

[58] S. Huang, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu. Indexing similar DNA sequences. In *AAIM*, volume 6124 of *LNCS*, pages 180–190, 2010.

[59] Trinh N. D. Huynh, Hon W.K., Lam T.W., and Sung W.K. Approximate string matching using compressed suffix arrays. In *In Proceedings of Symposium on Combinatorial Pattern Matching*, pages 434–444, 2004.

[60] S. Inenaga and M. Takeda. Sparse compact directed acyclic word graphs. In *Proc. Prague Stringology Conf*, pages 197–211, 2006.

[61] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554. IEEE Computer Society, 1989.

[62] J. Jansson, K. Sadakane, and W.K. Sung. Ultra-succinct representation of ordered trees. In *ACM-SIAM*, 2007.

[63] Haim Kaplan. Persistent data structures. In *In Handbook on Data Structures and Applications, Crc Press 2001, Dinesh Mehta and Sartaj Sahni (Editors) Boroujerdi, A., And Moret, B.M.E., "Persistency In Computational Geometry" Proc. 7th Canadian Conf. Comp. Geometry, Quebec*, pages 241–246, 1995.

[64] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *DCC*, pages 239–248, 2010.

[65] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *CPM*, volume 6661, pages 41–54, 2011.

[66] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel. Repetition-based compression of large DNA datasets. Poster at RECOMB, 2009.

[67] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *SPIRE*, volume 6393 of *LNCS*, pages 201–206, 2010.

[68] S. Kuruppu, S. J. Puglisi, and J. Zobel. Reference sequence construction for relative compression of genomes. In *SPIRE*, volume 7024 of *LNCS*, pages 420–425, 2011.

[69] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S. M. Yiu. High throughput short read alignment via bi-directional BWT. In *BIBM*, pages 31–36. IEEE, 2009.

[70] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, Mar 2008.

[71] N.J. Larsson and A. Moffat. Offline dictionary-based compression. In *DCC*, pages 296–305, 1999.

[72] M. Léonard, L. Mouchard, and M. Salson. On the number of elements to reorder when updating a suffix array. *Journal of Discrete Algorithms*, 11:87–99, 2011.

[73] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589, 2010.

[74] M.G. Maaß. Average-case analysis of approximate trie search. *Algorithmica*, 46(3):469–491, 2006.

[75] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *CPM*, pages 420–433, 2004.

[76] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching*, pages 121–137. Springer, 2005.

[77] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *String Processing and Information Retrieval*, pages 229–241. Springer, 2007.

[78] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE*, volume 4726 of *LNCS*, pages 229–241, 2007.

[79] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[80] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. of ACM-SIAM*. SIAM, 1990.

[81] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, May 2001.

[82] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[83] C. Meek, J.M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proc. of the 29th Intl. VLDB conference-Volume 29*, page 921, 2003.

[84] F. Meyer, S. Kurtz, R. Backofen, S. Will, and M. Beckstette. Structator: fast index-based search for rna sequence-structure patterns. *BMC bioinformatics*, 12(1):214, 2011.

[85] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.

[86] G. Navarro. A guided tour to approximate string matching. *CSUR*, 33:88, 2001.

[87] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1:205–239, 2000.

[88] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[89] Gonzalo Navarro. Wavelet trees for all. In *Combinatorial Pattern Matching*, pages 2–26. Springer, 2012.

[90] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, Princeton University, 1996.

[91] M. H. Overmars. Searching in the past, I, II. Technical report, University of Utrecht Technical Reports, 1981.

[92] M. Pătraşcu. Succincter. In *FOCS*, pages 305–313, 2008.

[93] S.J. Puglisi, W.F. Smyth, and A.H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4, 2007.

[94] R. Raman, V. Raman, and S.R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43, 2007.

[95] E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange. A guaranteed compression scheme for repetitive DNA sequences. In *DCC*, page 453, 1996.

[96] L. Russo, G. Navarro, and A. Oliveira. Dynamic fully-compressed suffix trees. In Paolo Ferragina and Gad Landau, editors, *Combinatorial Pattern Matching*, volume 5029 of *Lecture Notes in Computer Science*, pages 191–203. Springer, 2008.

[97] L. Russo, G. Navarro, and A. Oliveira. Parallel and distributed compressed indexes. In *Combinatorial Pattern Matching*, pages 348–360. Springer, 2010.

[98] L. M. S. Russo and A. L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. In *SPIRE*, volume 4209 of *LNCS*, pages 163–180, 2006.

[99] Luís M. S. Russo, G. Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.

[100] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302:211–222, 2003.

[101] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[102] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41:589–607, 2007.

[103] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213(0):13 – 22, 2012.

[104] K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, and D. Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10:1–12, 2009.

[105] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin. Improving suffix array locality for fast pattern matching on disk. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 661–672. ACM, 2008.

[106] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *SPIRE*, volume 5280 of *LNCS*, pages 164–175, 2008.

[107] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147:195–197, 1981.

[108] Wing-Kin Sung. Indexed approximate string matching. In *Encyclopedia of Algorithms*. Springer, 2008.

[109] M. Thorup. On AC0 implementations of fusion trees and atomic heaps. In *Proc. of the 14th ACM-SIAM sym. on Discrete algorithms*, SODA '03, pages 699–707. SIAM, 2003.

[110] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[111] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. Prospects and limitations of full-text index structures in genome analysis. *Nucleic acids research*, 40:6993–7015, 2012.

[112] P. Weiner. Linear pattern matching algorithms. In *IEEE SWAT*, 1973.

[113] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[114] S.S. Wong, W.K. Sung, and L. Wong. CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences. In *ICDE*, 2007.

[115] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[116] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.