Space Efficient Algorithms for String Processing

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

Jasbir Kaur Dhaliwal B.Com.Sci. (Hons.), M.App.Sci.,
School of Computer Science and Information Technology,
College of Science, Engineering and Health,
RMIT University,
Melbourne, Victoria, Australia.

February 2013

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Jasbir Kaur Dhaliwal
School of Computer Science and Information Technology
RMIT University
February 2013

Acknowledgments

This thesis would not have been possible without the support and guidance of various persons. First and foremost, I am truly indebted and thankful to my supervisors: Simon J. Puglisi, Andrew Turpin and Falk Scholer for their encouragement and guidance. To Juha Kärkkäinen with whom I have the honour and privilege in working on inverting the Burrows-Wheeler transform on disk.

Thanks to the numerous authors for making their code and data available. To William F. Smyth, for the loan of an experimental machine. I would also like to acknowledge the administrative support provided by Anton Demidov and Dora Drakopoulos. Last but not least, my thanks go to the university for providing the means by way of RMIT PhD International Scholarship to pursue my doctorate studies.

While working on the thesis, there have been many light moments resulting in laughter and joy. There have also been occasions which resulted in tears and frustration. However, it was the persistence coupled with the encouragement from my supervisors and my parents throughout my candidature which helped in completing the thesis.

I would always cherish the wonderful memories of the staff and students whom I met while working on my PhD in RMIT University, Melbourne, Australia.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- **Trends in Suffix Sorting: A Survey of Low Memory Algorithms**
  Jasbir Dhaliwal, Simon J. Puglisi and Andrew Turpin. In Proceedings of the 35th Australasian Computer Science Conference (ACSC), 2012.

- **Fast Semi-external Suffix Sorting**
  Jasbir Dhaliwal and Simon J. Puglisi. Technical Report TR-11-1, RMIT University, School of Computer Science and Information Technology, 2011 (submitted for publication at Information Processing Letters).

- **Practical Efficient String Mining**
  Jasbir Dhaliwal, Simon J. Puglisi and Andrew Turpin. IEEE Transactions on Knowledge and Data Engineering (TKDE), Accepted with minor modifications (2010).

This work was supported by the RMIT PhD International Scholarship.
The thesis was written in the `Vim` editor and typeset using the LaTeX 2_ε document preparation system.

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.

Contents

List of Figures

List of Tables

Abstract

The *suffix array (SA)*, which is an array containing the suffixes of a string sorted into lexicographical order, was introduced in the late eighties as a space efficient alternative to the suffix tree. It has since emerged as a useful data structure in string processing problems such as pattern matching, pattern discovery, and data compression.

The SA is often coupled with the *longest-common-prefix (LCP) array* that contains the length of the longest common prefixes between consecutive suffixes in the SA. When enhanced with the LCP array, the SA can provide efficient solutions to the above applications including a problem called pattern mining. To date, all the mining algorithms lie at either extreme of the efficiency spectrum: they are either fast and use enormous amounts of space, or they are compact and orders of magnitude slower. We present a mining algorithm that achieves the best of both these extremes, having runtime comparable to the fastest published algorithms while using less space than the most space efficient.

In all these applications, the construction of the SA — also known as suffix sorting — is one of the main computational bottlenecks. Most papers describing the SA assume the SA fits in RAM memory, limiting their applications. Even if the SA itself fits in memory, many algorithms constructing such an array require more space than the final result. The fastest algorithms in this large memory suffix sorting category use powerful pointer copying heuristics to expedite suffix sorting. Several space efficient algorithms have emerged in the last five years, where the trend is to use as little RAM as possible. They do so by finding a clever way to trade runtime, or by using slow compressed data structures, or by using external memory (disk), or some combination of these techniques. In this thesis, we focus on improving the runtime of a space efficient algorithm due to Kärkkäinen by adapting the heuristics from large memory suffix sorting to a semi-external setting.

Also, pointer copying has been heavily used to speed up the construction of the SA, but not the LCP array. We also discuss our attempts of combining the pointer copying heuristics

to an efficient LCP construction algorithm due to Kärkkäinen, Manzini and Puglisi.

The *Burrows-Wheeler transform (BWT)* was discovered independently of the SA, but it is now known that the two data structures are deeply linked. The BWT is central to practical compression tools such as szip and bzip2. Many papers have been published on constructing the BWT either in RAM or in external memory but few on inverting the BWT to obtain the original string — in fact none in external memory. For larger datasets, the existing traditional approaches cannot be used to invert the BWT. In such cases, we have to use disk. We close the gap between theory and practice by examining the problem of inverting the BWT efficiently on disk. We provide a practical implementation of the only theoretical proposal for the problem by Ferragina, Gagie and Manzini. We also provide new, faster solutions to the problem based on simple scanning and compression techniques.

Chapter 1

Introduction

The International Data Corporation (IDC) recent white paper sponsored by EMC Corporation,[1] reported that 486 billion gigabytes of data was generated in 2008 [Gantz et al., 2008]. The IDC analysts have forecasted data to grow massively in the next few years and with the decrease of hard disk costs per gigabyte, this does not seem implausible [Kirk, 2007]. It is clear that the amount of data is growing, and there is a pressing need to find useful information from it. The sources of datasets could vary from natural language such as the World Wide Web, to categorical sequences including genomic databases such as Genbank, or even large customer databases held by many organizations.

Strings are one of the most basic and useful data representations, and algorithms for their efficient processing pervade computer science with applications, perhaps too numerous to be listed completely. Some of the string processing applications are in search engines, the World Wide Web and operating systems. Other common, yet vital applications are in word processors such as Microsoft Word, utilities such as grep on UNIX, telephone directories, online dictionaries, thesauri and the list goes on [Gusfield, 1997].

The *suffix tree* [Wiener, 1973] is a fundamental data structure for string processing, but it has large memory requirements, which hampers its use in practical situations. Reducing suffix tree memory usage has been the focus of intense research in the algorithms community and has given rise to the *suffix array (SA)* [Gonnet, 1987; Manber and Myers, 1990; 1993] data structure. It provides efficient — often optimal — solutions for pattern matching (counting or finding all the occurrences of a specific pattern), pattern discovery and mining (counting or finding generic, previously unknown, repeated patterns in data), and related

[1] EMC are the acronyms for the founders of the company where E is for Egan, M is for Marino and C is for the third partner that left before the corporation was formed

problems, such as data compression. The SA is widely used in bioinformatics and computational biology [Gusfield, 1997; Abouelhoda et al., 2004; Flicek and Birney, 2009], and as a tool for compression in database systems [Chen et al., 2008; Ferragina and Manzini, 2010]. More recently, it is beginning to move from theory to practice as an index in information retrieval [Culpepper et al., 2010; Patil et al., 2011].

In all these applications, the construction of the SA — a process also known as suffix sorting — is one of the main computational bottlenecks. *Suffix array construction algorithms (SACAs)* have been the focus of intense research effort in the last 20 years or so. The survey due to Puglisi, Smyth and Turpin [Puglisi et al., 2007] counts 19 different *large (internal) memory* SACAs. These algorithms have assumed that the SA fits in RAM memory and so, limiting their applications to smaller datasets. Even if the SA itself fits in memory, many algorithms constructing such an array require more space than the final result. Moreover, this often makes the array construction infeasible. The fastest existing internal memory algorithms [Itoh and Tanaka, 1999; Seward, 2000; Ko and Aluru, 2005; Maniscalco and Puglisi, 2007] require at least $5n$ bytes (n is the length of the string) for suffix sorting, and use a broad class of heuristic methods called *pointer copying* to expedite sorting. In this method, a complete sort of a selected subset of suffixes is used to derive the order of the remaining suffixes as described in Chapter 3.

In the last five years, even more algorithms have emerged. The trend in these recent SACAs, has been to use as little memory as possible, either by finding a clever way to trade runtime, or by using slow compressed data structures, or by using disk, or some combination of these techniques. It is these recent *low memory* (*space efficient*) SACAs that are the focus of this thesis, in particular, a semi-external suffix sorting algorithm due to Kärkkäinen [2007]. At present, the algorithm sorts suffixes in blocks. A pass is made over the text to collect the first block of suffixes. The block is sorted and written to disk to form a contiguous section of SA. The memory is then reused to collect the second block where a second pass is made over the text and so on. This process is repeated until all the suffixes are written to disk. Observe that the number of passes made over the text is proportional to the sorting time which is the bottleneck of this algorithm. We therefore ask:

- *Can we implicitly sort the suffixes in a manner that will reduce the number of passes to be made over text without increasing the memory requirement of the Kärkkäinen's suffix sorting algorithm?*

In Chapter 5, we improve the runtime of this suffix sorting algorithm. The main con-

tribution of this chapter is a method for implementing the pointer copying heuristics from internal memory suffix array construction in a semi-external setting. It is not so straightforward as this method assumes the text and SA to be held in RAM memory throughout suffix sorting. However, Kärkkäinen's algorithm has a more restrictive setting. At any given time, only the text, a block of SA and some small data structures can reside in the RAM memory. We achieve a speed up of 2-4 times without increasing the memory usage of the algorithm. When compared to the recent algorithm by Ferragina, Gagie and Manzini [Ferragina et al., 2012], we are twice as fast when working memory is equated. To this end we ask:

- *Using the new algorithm, can we suffix sort strings with large alphabets?*

When compared to the best published algorithm for such strings, that is the algorithm by Larsson and Sadakane [2007], we are 2-3 times slower but we use less memory.

The SA is often coupled with the *longest-common-prefix (LCP) array* that contains the length of the longest common prefixes of adjacent suffixes in the SA. When enhanced with the LCP array, the SA can provide efficient solutions to *bottom-up* and *top-town* traversals within the same time bounds of the suffix tree, but less space overhead [Abouelhoda et al., 2004]. These traversals are the heart for many string problems discussed above, including a problem called *pattern mining*. In applications such as computational biology, pattern mining could be used to find potential indicators of genetic disorders that are being analyzed. For example, say a genetic disease is caused by a disorder of Chromosome-21 and the cause of this failure is unknown. Rather than analyzing every single pattern of the chromosome, the scope is narrowed down by collecting genetic sequences of Chromosome-21 of 1000 ill patients which are then stored in the positive database, and likewise, another 1000 healthy patients in the negative database. The patterns that are frequent (or always) in the positive database and infrequent (or never) in the negative database are the potential indicators of genetic disorder under consideration [Fischer, 2007].

In recent years, several algorithms for mining patterns from databases of string data (such as proteins and natural language texts) have been discovered, all of which traverse the enhanced SA data structure. All of these algorithms lie at either extreme of the efficiency spectrum: they are either fast and use enormous amounts of space, or they are compact and orders of magnitude slower. In particular, the two mining algorithms for string data that are feasible on genome datasets are: optimal linear time (fastest) by Fischer, Heun and Kramer [Fischer et al., 2006] and space efficient by Fischer, Mäkinen and Välimäki [Fischer et al., 2008] algorithms. Assume $\mathcal{M}$ to be a database of strings that has σ alphabets.

The optimal linear time algorithm solves mining problems in optimally linear time with the drawback of having space requirement of $O(n \log n)$ bits, which is not optimal for $\sigma << n$ (especially for constant σ) as the datasets themselves consume $n \log \sigma$ bits. On the other hand, the space efficient algorithm uses $O(n \log \sigma + |\mathcal{M}| \log n)$ bits with the drawback of having a time requirement of $O(n \log n)$. To this end, we ask:

- *Can we solve mining problems for string data closer time to the fastest published algorithm and use the same amount of space or less than the most space efficient algorithm?*

In Chapter 4, we present an algorithm that achieves the best of both these extremes, having runtime comparable to the fastest published algorithms [Fischer et al., 2006] while using less space than the most space efficient ones [Fischer et al., 2008]. This excellent practical performance is underpinned by theoretical guarantees. Our main mechanism for keeping memory usage low is to build the enhanced suffix array incrementally, in blocks. Once built, a block is traversed to output patterns with required support before its space is reclaimed to be used for the next block.

Pointer copying heuristics have been used to speed up SACAs but not *LCP construction algorithms (LCPCAs)* [Kasai et al., 2001; Manzini, 2004; Puglisi and Turpin, 2008; Kärkkäinen et al., 2009; Gog and Ohlebusch, 2011]. We therefore ask:

- *Can we reduce the construction time of the LCP array via pointer copying?*

Chapter 6 presents our attempts to combine pointer copying heuristics with an efficient LCPCA due to Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009]. Unfortunately, at the same time to us, Fischer [2011] had the same idea independently, and he published it before us. He used less space than us as he saw a clever way to use unused space in the LCP array which we did not see.

The *Burrows-Wheeler transform (BWT)* [Burrows and Wheeler, 1994] was discovered independently of the SA, but it is now known that the two data structures are essentially equivalent. The construction of the BWT — directly from the string or via the SA or via some other method — is known as the *forward BWT*. In the last decade, the relationship between the BWT and the SA has been heavily investigated, and has led to the very active field of compressed full-text indexing: the study of data structures that allow fast pattern matching, but require space close to that of the compressed text (see Navarro and Mäkinen [2007] and references therein). The BWT itself has been the focus of much research in text compression [Sadakane, 1998; Balkenhol et al., 1999; Manzini, 1999; Deorowicz, 2002; Abel,

2005]. It performs no compression, but transforms the input string in such a way that makes compression easy using other methods [Manzini, 2001a; Kärkkäinen and Puglisi, 2010]. The BWT is used in compression tools such as szip [Schindler, 2002] and bzip2 [Seward, 2004]. These compression tools divide the BWT into blocks and then compress each block separately, meaning that only local redundancy can be detected. Compression is significantly better if the entire string is processed (see Ferragina and Manzini [2010]).

Much research has been carried out on the forward BWT algorithms but very little on *inverse (reverse) BWT (IBWT)* where we obtain the input string from the BWT. For a very large collection, there might not be enough RAM to invert the BWT using traditional approaches [Kärkkäinen and Puglisi, 2010]. In such cases, we have to use external memory (disk) and, invert the BWT on disk. There are forward BWT algorithms that work on disk, but no inverse BWT algorithms. Only a theoretical proposal by Ferragina, Gagie, Manzini [Ferragina et al., 2010] exists. We therefore ask:

- *Can we provide an implementation of the only theoretical proposal of Ferragina, Gagie and Manzini's algorithm?*

In Chapter 7, we close the gap between theory and practice by examining the problem of inverting the BWT efficiently on disk. As there were open questions related to this proposal, several possibilities were explored and the best approach in terms of time and space was implemented. To this end, we ask:

- *Their algorithm is complex, so can we discover simple, more practical inversion algorithms for disk?*

Several new inversion algorithms of our own that use simple scanning and compression techniques were implemented. Our best approach is up to 14 times faster than the implementation of Ferragina, Gagie and Manzini's algorithm.

The significance and novelty of this thesis can be summarized as follows.

- We introduce a new approach to space efficient string mining in which the suffix and LCP arrays are constructed one block at a time, with each block traversed before the memory is reclaimed for the next block. Two algorithmic parameters can be tuned so that this new approach requires $O(n \log \sigma + |\mathcal{M}| \log n)$ bits of space and $O(n \log^2 n)$ time in the worst case.

- The empirical performance of the new approach on large collections of DNA and Protein data is examined. The new approach is as space efficient as Fischer et al. [2008] but

is more than an order of magnitude faster, running only slightly slower in practice than Fischer et al. [2006]. We stress, as Fischer et al. [2008] do, that for large scale string mining problems space is often more of a concern than time: without adequate memory, mining becomes impossible, though one is often willing to wait a little longer, provided results eventually arrive. Our approach has two parameters which allow for a space-time tradeoff, which we explore.

- We provide a practical implementation of a semi-external suffix sorting due to Kärkkäinen [2007].
- We improve the practical performance of Kärkkäinen's suffix sorting by showing how to adapt powerful heuristics from large memory suffix algorithms to the semi-external setting.
- We show the performance of the new algorithm has relevant space-time tradeoffs in practice, relative to existing approaches.
- We demonstrate the efficacy of the algorithms for suffix sorting of strings on large alphabets.
- We improve the practical performance of the LCP construction algorithm by Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009] using the powerful heuristics from large memory suffix algorithms.
- We provide a practical implementation to invert the BWT of the only theoretical proposal by Ferragina, Gagie and Manzini [Ferragina et al., 2010] .
- We provide faster solutions to the inversion problem, based on simple scanning and compression techniques.

This thesis proceeds in the following manner.

Chapter 2 describes the background and notation used for the rest of the thesis. This overview can be safely skimmed by readers already familiar with suffix sorting, but may serve as a useful tutorial for those new to the problem.

Chapter 3 provides high level descriptions of several SACAs that have emerged since the survey due to Puglisi et al. [2007], avoiding implementation details as much as possible.

These algorithms have tended to strive for small working space (RAM), often at the cost of runtime, and make use of compressed data structures or disk to achieve this goal.

Chapter 4 presents an efficient pattern mining algorithm that has runtime comparable to the fastest published algorithms with space less than most space efficient algorithms.

Chapter 5 describes how pointer copying heuristics from internal memory suffix array construction are adapted to work in a semi-external setting. Several algorithmic optimization techniques are also used to speed up the construction time of the semi-external SACA by Kärkkäinen [2007]. The runtime of our new algorithm is faster than the original algorithm and alternative approaches when the amount of working memory is equated.

Chapter 6 describes our work on combining pointer copying heuristics with the LCPCA due to Kärkkäinen et al. [2009]. The construction time of the LCP array is improved.

Chapter 7 considers the problem of inverting the BWT efficiently on disk. We describe an implementation of Ferragina et al. [2010], a variant of their algorithm and several new inversion algorithms. The runtime of our best approach is faster than the implementation of Ferragina et al. [2010].

Chapter 8 concludes and outlines some directions that could benefit from further research.

Chapter 2

Preliminaries

This chapter sets out notation and introduces basic concepts used throughout this thesis.

We consider a string x of $n+1$ characters (or symbols), $x = x[0,n] = x[0]x[1]\ldots x[n]$, drawn from a fixed, ordered alphabet Σ of size σ. The final character, $x[n]$, is a special end of string character, '\$', which occurs nowhere else in x and is lexicographically (alphabetically) smaller than any other character in Σ. The string x requires $(n+1)\log\sigma$ bits of storage without compression.

We write $x[i,j]$ to represent the substring of x starting at position i and ending at position j. There are two special types of substrings we are interested in: a substring $x[0,i]$, $1 \leq i \leq n$, that begins at the first symbol of x is called a *prefix* of x; and a substring $x[i,n]$, $0 \leq i \leq n$, that ends at the last letter of x is called a *suffix* of x. For brevity, we will often refer to suffix $x[i,n]$ as "suffix i".

The *suffix array* $\mathrm{SA}[0,n]$ of string x (or just SA, when the context is clear) is a permutation of the integers 0 to n such that

$$x[\mathrm{SA}[0],n] < x[\mathrm{SA}[1],n] < \ldots < x[\mathrm{SA}[n],n].$$

In other words, SA lists the suffixes of x in ascending lexicographical order. As SA stores $n+1$ integers in the range $[0,n]$, it requires $(n+1)\log(n+1)$ bits of storage. Figure 2.1 shows an example SA for the string x, florreencee\$, where $x[11,11] =$ \$ is the lexicographically least suffix, $x[8,11] =$ cee\$ is the second least and so on.

A *suffix tree* is a tree representation of the above suffixes and its string. To see the relationship of the SA to the suffix tree, observe that the values in the SA are merely the leaves of the tree as they would be visited in the depth first order, as we see in Figure 2.2.

i	Suffixes	SA	Suffixes
0	florreencee$	11	$
1	lorreencee$	8	cee$
2	orreencee$	10	e$
3	rreencee$	9	ee$
4	reencee$	5	eencee$
5	eencee$	6	encee$
6	encee$	0	florreencee$
7	ncee$	1	lorreencee$
8	cee$	7	ncee$
9	ee$	2	orreencee$
10	e$	4	reencee$
11	$	3	rreencee$

Figure 2.1: Suffixes of string florreencee$ are sorted lexicographically forming the SA. On the left hand side, the suffixes are listed in string order and the on the right hand side they are listed lexicographically.

Occasionally it will be useful for us to refer to the *inverse suffix array* SA^{-1}, which is defined such that $\mathrm{SA}^{-1}[i] = j$ if and only if $\mathrm{SA}[j] = i$. In other words, SA^{-1} refers to the position of a given suffix in SA.

The *longest-common-prefix (LCP) array*, $\mathrm{LCP}[0, n]$ of x is an array defined relative to SA. In particular, let $\mathrm{lcp}(y, z)$ denote the length of the longest common prefix of strings y and z. Then for every $j \in [1, n]$,

$$\mathrm{LCP}[j] = \mathrm{lcp}(x[\mathrm{SA}[j-1], n], x[\mathrm{SA}[j], n]),$$

that is, $\mathrm{LCP}[j]$ is the length of the longest common prefix of suffixes $\mathrm{SA}[j-1]$ and $\mathrm{SA}[j]$. $\mathrm{LCP}[0]$ is undefined. For an overview of the state-of-the-art of LCP array construction algorithms, see Kärkkäinen et al. [2009] and Gog and Ohlebusch [2011].

For strings with low amounts of repetition, string sorting algorithms such as multikey quicksort (MKQS) [Bentley and Sedgewick, 1997] are effective for building the SA. However, it is usually faster to employ one of the many specialized suffix sorting algorithms, which are surveyed in Chapter 3. Some of these algorithms build a *Difference Cover Sample* [Burkhardt and Kärkkäinen, 2003; Kärkkäinen, 2007] of size v, $\mathrm{DCS}_v(x)$ for string x, which allows the lexicographical order of suffixes to be determined efficiently. $\mathrm{DCS}_v(x)$ is a sorted subset of all suffixes that are chosen so that for any two positions in SA, say i and j, there exists a $k < v$ such that both $\mathrm{SA}[i + k]$ and $\mathrm{SA}[j + k]$ are in the sample, where v is a parameter

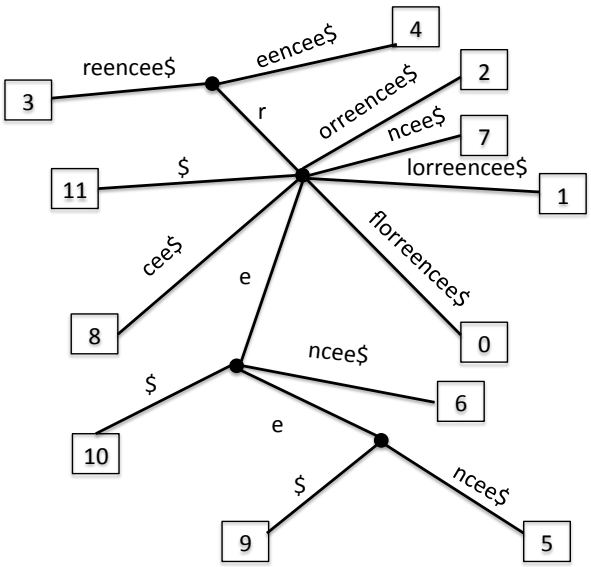

Figure 2.2: The suffix tree for string florreencee$ where the suffixes are stored as indices into the original string at the leaves of the tree instead of the suffixes themselves.

that controls the size of the sample. For example, if $v = 3$, then the DCS can be set to all suffixes whose index is 1 or 2 (mod 3). The set $\{1, 2\}$ is a *difference cover* of the range 1 to 3 [Colbourn and Ling, 2000]. Now for any pair of positive integers, i and j, there will exist $k < 3$ such that $i + k$ (mod 3) and $j + k$ (mod 3) will be in the set $\{1, 2\}$. Generally, there exists a difference cover with $\Theta(\sqrt{v})$ elements, given v. The usefulness of this type of sampling is summarized in the following lemma.

Lemma 1 ([Burkhardt and Kärkkäinen, 2003; Kärkkäinen, 2007])**.** *The Difference Cover Sample* $\mathrm{DCS}_v(x)$ *of text* $x[1, n]$ *and period* $v \in [3, n]$ *can be constructed in* $O(|\mathrm{S}| \log |\mathrm{S}| + v|\mathrm{S}|)$ *time and in* $O(v+|\mathrm{S}|)$ *space (excluding the space for* x*), where* S *is a set of* $\Theta(n/\sqrt{v})$ *suffixes. Let* $x[i, n]$ *and* $x[j, n]$ *be two suffixes such that* $\mathrm{lcp}(x[i, n], x[j, n]) \geq v-1$*, then given* $\mathrm{DCS}_v(x)$ *the lexicographical order of* $x[i, n]$ *and* $x[j, n]$ *can be determined in constant time.*

Thus, given any two positions in the text, the suffixes beginning at those positions can be lexicographically ordered by comparing up to $k < v$ individual characters from their prefixes.

If any of these characters differ, then the order is resolved, but if all are the same, the order can be resolved by the suffixes at positions $i + k$ and $j + k$ which are already ordered in the sample. Determining the relative order of two suffixes thus requires $O(v)$ time.

Recently, Puglisi and Turpin [2008] showed how to extend the above $\mathrm{DCS}_v(x)$ data structure so that $\mathrm{lcp}(x[i, n], x[j, n])$ for any suffixes i and j can also be computed in $O(v)$ time. We summarize their result with the following lemma.

Lemma 2 ([Puglisi and Turpin, 2008])**.** *The extended Difference Cover Sample can be constructed from* $\mathrm{DCS}_v(x)$ *and* x *in* $O(v|S|)$ *time and in* $O(v + |S|)$ *space (including* $\mathrm{DCS}_v(x)$ *and excluding the space for* x*), where* S *is the same set of* $\Theta(n/\sqrt{v})$ *suffixes used in* $\mathrm{DCS}_v(x)$*. For any two suffixes that share a prefix of* $v - 1$ *characters, given the extended Difference Cover Sample,* $\mathrm{lcp}(x[i, n], x[j, n])$ *can be determined in constant time.*

Suffix sorting, the process of constructing SA, is also the computational bottleneck for performing the *Burrows-Wheeler transform (BWT)* [Burrows and Wheeler, 1994]. The BWT was discovered independently of the suffix array, but it is now known that the two data structures are deeply linked.

The BWT transforms the string x by sorting its $n + 1$ cyclic rotations as shown in Figure 2.3b. To see the relationship between the SA and BWT: in the matrix of sorted rotations, the prefixes of each rotation up to the \$ are precisely the suffixes of x in the same order in which they appear in SA. BWT is a permutation of x defined by SA: $\mathrm{BWT}[i] = x[\mathrm{SA}[i] - 1]$, unless $\mathrm{SA}[i] = 0$, in which case $\mathrm{BWT}[i] = \$$. To see the relationship between SA, BWT and LCP, see Figure 2.4.

For the full properties of the BWT matrix, we refer the reader to Burrows and Wheeler [1994], Manzini [2001a] and Ferragina and Manzini [2005]. We only discuss properties that will aid in understanding the algorithms in this thesis. The two main columns of the matrix are: F, the first column which is obtained by lexicographically sorting the characters in x; and L, the last column that represents the BWT. Although simple, these cyclic rotations are a powerful tool that can be used to invert the BWT.

The elements critical for inverting the BWT are as follows [Ferragina and Manzini, 2005].

- $\mathrm{C}[c]$ where $c \in \Sigma$ denotes the starting position of the group that begins with character c in F. Sometimes, these positions in C are referred as *group counters*.
- $\mathrm{rank}(\mathrm{L}[q], \mathrm{L}, q)$ denotes the number of occurrences of c at position q in prefix $\mathrm{L}[0, q]$.

i	Suffixes	SA	Suffixes
0	florreencee$	11	$
1	lorreencee$	8	cee$
2	orreencee$	10	e$
3	rreencee$	9	ee$
4	reencee$	5	eencee$
5	eencee$	6	encee$
6	encee$	0	florreencee$
7	ncee$	1	lorreencee$
8	cee$	7	ncee$
9	ee$	2	orreencee$
10	e$	4	reencee$
11	$	3	rreencee$

(a) Suffixes of string florreencee$ are sorted lexicographically forming the SA.

All Rotations	F		L	rank(L[i], L, i)	i
florreencee$	$	florreence	e	0	0
lorreencee$f	c	ee$florree	n	0	1
orreencee$fl	e	$florreenc	e	1	2
rreencee$flo	e	e$florreen	c	0	3
reencee$flor	e	encee$flor	r	0	4
eencee$florr	e	ncee$florr	e	2	5
encee$florre	f	lorreencee	$	0	6
ncee$florree	l	orreencee$	f	0	7
cee$florreen	n	cee$florre	e	3	8
ee$florreenc	o	rreencee$f	l	0	9
e$florreence	r	eencee$flo	r	1	10
$florreencee	r	reencee$fl	o	0	11

(b) The left column shows all rotations of the string *florreencee$*. When sorted (right column) this gives the BWT as column L. The F column contains characters of the input sorted lexicographically. Position 6 in L is the `starting` point for inversion.

Figure 2.3: The relationship between suffix sorting, Burrows-Wheeler transform and Inverse Burrows-Wheeler transform.

C is combined with rank to obtain the *Last to First mapping*, that is the LF function of Equation 2.1.

$$\mathrm{LF} = \mathrm{C}[\mathrm{L}[i]] + \mathrm{rank}(\mathrm{L}[i], \mathrm{L}, i) \tag{2.1}$$

To compute C, we count the occurrences of each character in L as shown in Figure 2.5a. By replacing C[j] with the cumulative sum of the counts in C[$j-1$] for $j > 0$, we can obtain the starting position of each character group in F (see Figure 2.5b). For example, C[$] is 0 as the starting position of '$' in F is 0.

To invert the BWT, the symbol in the BWT which is the last symbol of the input string

	0	1	2	3	4	5	6	7	8	9	10	11
x	f	l	o	r	r	e	e	n	c	e	e	$
SA	11	8	10	9	5	6	0	1	7	2	4	3
BWT	e	n	e	c	r	e	$	f	e	l	r	o
LCP	0	0	0	1	2	1	0	0	0	0	0	1

Figure 2.4: The relationship between SA, BWT and LCP of string florreencee$.

is output. The position of this symbol is recorded when the forward transform is performed. The LF function is then used to find the next character (of the input string) as shown in Figure 2.3b. Position 6 is the starting point for inversion and so, $ is output. To compute rank($, L, 6), we count the occurrences of symbol '$' in prefix L[0,6] which is 0 as there are no occurrences in L[0,6]. 0 is then added to C[$] which gives the next character to be inverted, that is the character at position 0 in L.

In the above example, the BWT is inverted in reverse order. The implementation of reverse order operations are faster in practice [Kärkkäinen and Puglisi, 2010] and so, implemented in algorithms described in Chapter 7. However, to invert the BWT in forward order using the reverse operations, the input string can be reversed before a forward transform is performed.

	$	c	e	f	l	n	o	r
j	0	1	2	3	4	5	6	7
C	1	1	4	1	1	1	1	2

(a) Round 1: Occurrences of each character are stored in C.

	$	c	e	f	l	n	o	r
j	0	1	2	3	4	5	6	7
C	0	1	2	6	7	8	9	10

(b) Round 2: Cumulative count for each distinct character. This corresponds to the starting position of each character group in F.

Figure 2.5: The two rounds to compute the starting positions of each group in F.

Chapter 3

Survey of Recent Suffix Sorting Algorithms

In this chapter, we survey several suffix array construction algorithms (SACAs) that have been discovered since the survey due to Puglisi, Smyth and Turpin [Puglisi et al., 2007]. These algorithms strive for small working space (RAM), at the cost of runtime, and use compressed data structures or secondary memory (disk) to achieve this goal. Previous work and applications of suffix arrays (SAs) are briefly reviewed in Section 3.1. Then, in Section 3.2, a high level description of each new algorithm is given. Experimental comparisons of some of the recent algorithms that we have efficient implementations are given in Section 3.3.

3.1 Suffix arrays: History and Applications

In the early seventies, Knuth, Morris and Pratt described pattern matching algorithms that compute a failure-function. This failure-function indicates the number of character comparisons that can be skipped based on the previously performed comparisons. Despite having linear time complexity, the `KMP` algorithm is inefficient on very large input strings as the search is bounded by n, the length of the input string [Knuth et al., 1977].

Wiener [1973] introduced the suffix tree (or position tree or patricia (PAT) tree) data structure that reduces pattern search time to be proportional to the length of the pattern. The drawback of Wiener's approach was that although the suffix tree requires $O(n)$ space, it has high constant factors, and occupies about $30n$ bytes in practice. Moreover, the suffix tree is unsuitable for applications such as document listing [Muthukrishnan, 2002].

McCreight [1976] improved the space requirements of Weiner's approach by about 25% by adding the suffixes in the decreasing order of their length, in contrast to Wiener who adds them in increasing order, without affecting its linear construction time. Ukkonen [1995] then produced a simpler and easier variant of McCreight's algorithm that runs in the same time bounds but processes the string from left-to-right, a property that is required in some applications.

The above suffix tree construction algorithms assume a constant alphabet size. Farach [1997] described the first recursive algorithm that is linear for all alphabets including integers. Several improvements have since been made to this recursive algorithm (see Farach-Colton et al. [2000]). The sorting routing of this algorithm has inspired several recursive SACAs, such as `KS` [Kärkkäinen and Sanders, 2003], `KSPP` [Kim et al., 2003] and `KA` [Ko and Aluru, 2005]. The suffix tree has been superseded, at least in practice, by the SA, mostly due to the large space requirements to construct the tree even with the improvements by Kurtz [1999].

The SA is a fundamental data structure for string processing. It provides efficient solutions for pattern matching (counting or finding all the occurrences of a specific pattern), pattern discovery and mining (counting or finding generic, previously unknown, repeated patterns in data), and related problems, such as data compression. SA is also widely used in bioinformatics and computational biology [Gusfield, 1997; Abouelhoda et al., 2004; Flicek and Birney, 2009], and as a tool for compression in database systems [Chen et al., 2008; Ferragina and Manzini, 2010]. More recently, it is beginning to move from theory to practice as an index in information retrieval [Culpepper et al., 2010; Patil et al., 2011].

3.2 Suffix array construction algorithms (SACAs)

The suffix array (or patricia (PAT) array) was discovered independently with different intentions. Gonnet [1987] implemented a suffix array to speed pattern search in the Oxford English Dictionary project, and Manber and Myers [1990; 1993] for genome databases. Since then, many SACAs have emerged. The survey by Puglisi, Smyth and Turpin [Puglisi et al., 2007] counts 19 different SACAs and categorizes them as follows:

- Prefix-Doubling
  Suffixes in this category of algorithms are not sorted in a single iteration but semi-sorted in several iterations (at most $\log_2 n$). The overall runtime is $O(n \log n)$. The two main algorithms in this category are the `MM` [Manber and Myers, 1990; 1993] (or alternatively `G` [Gonnet, 1987]) algorithm and the `LS` [Larsson and Sadakane, 2007] algorithm.

- Induced Copying

  Induced Copying (or pointer copying) is a heuristic method for suffix sorting. A complete sort of a selected subset of suffixes is used to derive the order the remaining suffixes. The basic principle of this method was first introduced by Burrows and Wheeler [1994] and since then, many different styles have developed: `IT` [Itoh and Tanaka, 1999], `S` [Seward, 2000], `BK` [Burkhardt and Kärkkäinen, 2003], `MF` [Manzini and Ferragina, 2004], `SS` [Schürmann and Stoye, 2005], `BB` [Baron and Bresler, 2005], `M` [Maniscalco, 2005] and `MP` [Maniscalco and Puglisi, 2007]. The selected subset of suffixes can be sorted with any efficient string sorting algorithms such as ternary-split quicksort (TSQS) [Bentley and McIlroy, 1993], multikey quicksort (MKQS) [Bentley and Sedgewick, 1997] and burst sort [Sinha and Zobel, 2004]. These string sorting algorithms must not be used to construct the entire suffix array as they are computationally expensive in runtime, in particular for large strings that have many repetitions.

  With an exception of the `BK` algorithm which is $O(n \log n)$, the algorithms in this category generally have a super linear worst case runtime.

- Recursive

  The general idea of the algorithms in this category is similar to induced copying except that the selected subsets are sorted recursively. The algorithms in this category are usually linear and include: `KS` [Kärkkäinen and Sanders, 2003], `KSPP` [Kim et al., 2003], `KJP` [Kim et al., 2004], `N` [Na, 2005], `KA` [Ko and Aluru, 2005], `NZC` variants [Nong et al., 2009a;b; 2011; Nong, 2011] and `OS` [Okanohara and Sadakane, 2009].

For more details about the above SACAs, we refer the reader to the survey by Puglisi et al. [2007]. In this section, we describe several SACAs that have emerged since the survey. We categorize them as "low memory" in Figure 3.1. These recent low memory algorithms (`S` [Sirén, 2009], `FGM` [Ferragina et al., 2010] and `BCR` [Bauer et al., 2011]) are covered in this chapter, with the exception of the `K` algorithm that is described in more detail in Chapter 5, as it is vital for other parts of the thesis. We also describe two SACAs that are "lightweight" (they use space to hold the input string and the resulting SA or BWT): `NZC` variants [Nong et al., 2009a;b; 2011; Nong, 2011] and `OS` [Okanohara and Sadakane, 2009]. The `NZC` variants are the most space efficient algorithms when compared to the 2007 survey, but the most space consuming algorithms in this section.

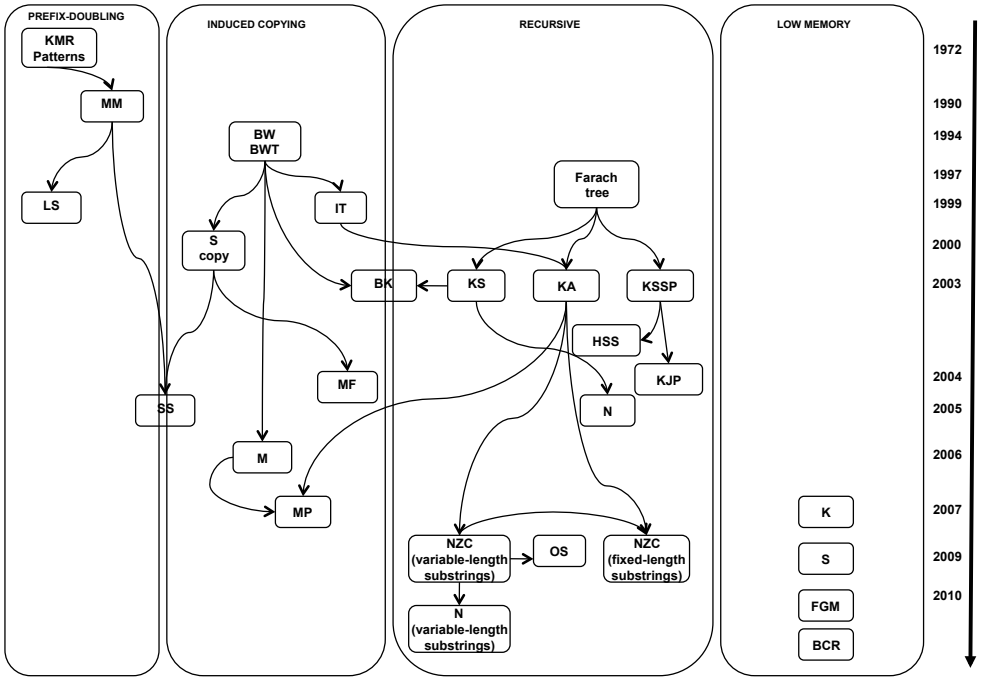

Figure 3.1: The three main categories of the SACAs: Prefix-Doubling, Induced Copying and Recursive that are identified by Puglisi et al. [2007]. The algorithms that have emerged since the survey are categorized on the right of the figure as Low Memory (space efficient).

3.2.1 Algorithm NZC [Nong et al., 2009a;b; 2011; Nong, 2011]

Several variants of Nong, Zhang and Chan algorithm exist. The variants are as follows.

Algorithm NZC (variable-length substrings)

The algorithm due to Nong, Zhang and Chan [Nong et al., 2009b; 2011] begins by selecting and sorting a subset of suffixes, and then using the order of those suffixes to induce the sort of the remaining suffixes. It uses RAM to hold the input string and the resulting suffix array. In this sense, it is "lightweight" in the terminology of Manzini and Ferragina [2004]. It also runs in linear time in the length of the string, settling an open problem posed by Puglisi et al. [2007], by showing that it is possible to be simultaneously lightweight in space usage and linear in runtime.

The way the suffixes are differentiated into subsets is similar to the algorithm by Ko and Aluru [2005] and Maniscalco and Puglisi [2007]. Suffixes are split into *Larger* and *Smaller* types (L and S respectively) depending on their lexicographic rank relative to their right hand neighbouring suffix. Then, a group of leftmost S suffixes (LMS) can be used to derive the sort of the L suffixes, which in turn is used to induce the sort of the S suffixes.

Input: x.
1: Label each position S or L.
2: Identify LMS substrings.
3: Induce sort the LMS substrings.
4: Name each LMS substring by rank to obtain a shorter string, x_1.
5: **if** each character in x_1 is unique **then**
6: Compute SA_1 from x_1.
7: **else**
8: Recursively call this pseudocode, that is SA-IS(x_1,SA_1)
9: Induce SA from SA_1.
Output: SA.

*Figure 3.2: Pseudocode SA-IS(x,*SA*) that uses LMS due to Nong et al. [2009b; 2011].*

Pseudocode of this algorithm is presented in Figure 3.2. The algorithm begins by making a pass over the string, $x[0,n]$ assigning a type of either L or S to the suffix (of the input string) depending if it is *Larger* or *Smaller* than its right hand neighbouring suffix. Thus, a suffix $x[i,n]$ is type S if $x[i,n] < x[i+1,n]$, or type L if $x[i,n] > x[i+1,n]$. These definitions are then used to define the leftmost S-type (LMS) character and substrings respectively. A character in the string is defined as an LMS character if $x[i]$ is a type S and $x[i-1]$ is a type L if $x > 0$. An LMS substring is a substring $x[i,j]$ with both $x[i]$ and $x[j]$ being LMS characters, and there is no other LMS character in the substring, for $i \neq j$. The end of string symbol, $x[n] = \$$, is defined to be S, and hence is also an LMS suffix (as its left hand neighbour must be larger than it). Types are shown below for the example input string x, ababaacaa$.

	0	1	2	3	4	5	6	7	8	9
x	a	b	a	b	a	a	c	a	a	$
type	*S*	*L*	*S*	*L*	*S*	*S*	*L*	*L*	*L*	*S*
LMS			*		*					*

Suffix 8 (a$) is of type L as it is lexicographically larger than suffix 9 ($). Suffix 7 (aa$) is of type L as it is lexicographically larger than suffix 8. This process is repeated until

the types are assigned to all the suffixes of the string. During the scan, the LMS types are identified and are indicated above by '$*$'.

After identifying the types, the indices of the LMS suffixes are placed in their character groups in the SA. Each suffix beginning with an LMS character is placed at the rightmost end of its character group in SA, and the group counter decremented. Let C[j] be the cumulative count of all characters in the string that are lexicographically less than or equal to j. Thus, for our input string:

	\$	a	b	c
j	0	1	2	3
C[j]	0	6	8	9

The first LMS character encountered is 'a', the beginning of suffix 2, which is placed at position 6, as C[a] = 6, and C[a] is decremented. Next we encounter 'a' at the beginning of suffix 4, and so it is inserted at C[a] = 5. Finally, suffix 9 beginning with '\$' is inserted at C[\$] = 0. The end result is shown here, with the boundaries of each group denoted by brackets and the empty positions denoted by '$\perp$'.

	0	1	2	3	4	5	6	7	8	9
SA	9	($\perp$	$\perp$	$\perp$	$\perp$	4	2)	($\perp$	$\perp$)	$\perp$
x_1	1	1	0							
groups	\$	(					a)	(	b)	c

Ranks are then assigned to the LMS suffixes to obtain a shorter x_1 string while preserving the order of the string. So, in our example, x_1 is 110. As there is a single LMS \$ suffix in the \$ group, LMS \$ suffix is given the rank 0. The remaining LMS aba suffixes are given the rank 1 as they cannot be differentiated with just LMS aba. Induced sorting is applied until the ranks become unique. Unique ranks indicate that the LMS suffixes have been sorted. It is a coincidence that the order of the LMS indices in this example do not change as the indices are in their correct positions.

Once the LMS suffixes are sorted, a left-to-right ($j = 0..n$) scan is made over SA to derive the order of the L suffixes. Within a group, type L suffixes always come before type S suffixes as the latter is lexicographically larger than the former. If, while scanning, we find a suffix i ($i = \text{SA}[j]$) such that suffix $i - 1$ is of type L and has not been placed on SA (has not been sorted), we place suffix $i - 1$ at the empty position of its group. The starting position of each group in the SA is stored in C and so, C = $\{0, 1, 7, 9\}$. Continuing with the example, when $j = 0$, $i = \text{SA}[0] = 9$, suffix $9 - 1 = 8$ is a suffix of type L, beginning with 'a' and has

not been sorted. It is thus placed at position 1 as C[a] = 1 before incrementing its group counter. Likewise, when $j = 2$, $i = \text{SA}[2] = 8$, suffix $8 - 1 = 7$ is a type L suffix that has not been placed on the SA and so, placed at position 2 as C[a] = 2. This process is repeated until all the type L suffixes have been placed on the SA.

	0	1	2	3	4	5	6	7	8	9
SA	9	(8	7	⊥	⊥	4	2)	(3	1)	6
groups	$	(					a)	(	b)	c

After placing the type L suffixes, the type S suffixes are collected in a single scan over the SA. A right-to-left scan is made over SA and for each i ($i = \text{SA}[j]$), we inspect suffix $i - 1$ and if it is a type S suffix that has not been placed on the SA, it is placed at the end of its group. The group counters in C are reset so that the ending position of each group is stored, $\text{C} = \{0, 6, 8, 9\}$. Should there be a LMS suffix at the said position, it is overwritten with the S suffix. Continuing with the example, when $j = n$, $i = \text{SA}[j] = 6$, suffix $6 - 1 = 5$ is a type S suffix, beginning with 'a' and so, it is placed at position 6 as C[a] = 6. C[a] is then decremented. Likewise, when $j = 8$, $i = 1$, suffix $1 - 1 = 0$ is a suffix of type S, beginning with 'a', and so, it is placed at position 5 as C[a] = 5. This process is repeated until all the type S have been derived. An exceptional case occurs when $j = 5$, $i = 0$ as there is no suffix $0 - 1$. When this occurs, we simply ignore the suffix and scan to the next position, which is $j = 6$ in this example.

	0	1	2	3	4	5	6	7	8	9
SA	9	(8	7	4	2	0	5)	(3	1)	6
groups	$	(					a)	(	b)	c

Algorithm NZC (fixed-length substrings)

Nong, Zhang and Chan [Nong et al., 2009a; 2011] proposed an algorithm that offers space-time tradeoff by using fixed length substrings known as d-critical substrings. A character in the string is defined as a d-critical character ($d \geq 2$) if it is an LMS character; or $x[i - d]$ is a d-critical character and $x[i + 1]$ is not an LMS character with no d-critical character in $x[i - d + 1, i - 1]$. An exception occurs as the first character of the string is not a d-critical character. That is, it is neither an LMS character nor does $x[0 - d]$ exists. A substring is defined to be a d-critical substring $x[i, i + d + 1]$ if $x[i]$ is a d-critical character.

Say d is 2. The 2-critical substrings are indicated below by '$*$' for the example string, ababaacaa$. The 2-critical substrings are: abaa, aaca, caa$, $$$$. '$' is added to suffix 9 to ensure the length of the final 2-critical substring is four.

	0	1	2	3	4	5	6	7	8	9
x	a	b	a	b	a	a	c	a	a	\$
type	*S*	*L*	*S*	*L*	*S*	*S*	*L*	*L*	*L*	*S*
2-critical			∗		∗		∗			∗

As noted by the authors, the LMS substrings form a subset of d-critical substrings. The number of d-critical substrings is equal to the number of LMS substrings in the best case. However, the former is higher than the latter in the worst case.

Algorithm N (variable-length substrings)

A bit vector of $O(n)$ bits of space (RAM) is required to differentiate type L and S suffixes. Nong [2011] improves the space efficiency of Algorithm `NZC` (variable-length substrings) by using unused space in the SA for the bit vector. The C array is removed during recursion. This algorithm runs in linear time and uses $O(1)$ working space excluding the space for the text and SA.

3.2.2 Algorithm OS [Okanohara and Sadakane, 2009]

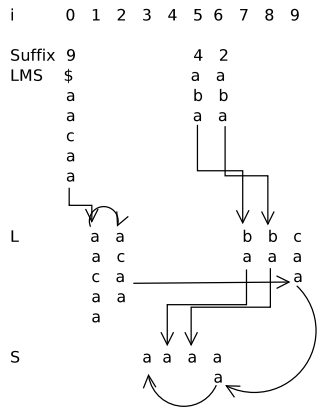

Figure 3.3: An example of the modified algorithm that induces the sort of the string ababaacaa\$ using LMS substrings that are moved in queues.

The algorithm of Okanohara and Sadakane [2009] uses the same framework as the algorithm by Nong et al. [2009b], described in the previous section, but derives the BWT string

instead of the SA. This algorithm runs in linear time via careful implementation of data structures that reduce memory overheads to $O(n\log\sigma\log\log_\sigma n)$ bits.

Okanohara and Sadakane explicitly store and move LMS substrings in queues, namely, LMS_c, L_c and S_c queues. The LMS_c queue, as the name suggests, stores LMS substrings where c represents the last character of the substring. L_c stores substrings that are of type L and ends with character c. S_c stores substrings that are of type S and ends with character c. As the LMS substrings are moved between L_c and S_c, the character c which is the BWT character is removed and stored in its character group data structure, namely, $\mathrm{BWT}_{\mathrm{L}_c}$ and $\mathrm{BWT}_{\mathrm{S}_c}$. The character c of type L is stored in $\mathrm{BWT}_{\mathrm{L}_c}$ and the character c of type S is stored in $\mathrm{BWT}_{\mathrm{S}_c}$. These character group based data structures are then combined to form the BWT once all the characters have been removed from the LMS substrings.

Figure 3.3 shows an example for string string ababaacaa\$. Once the LMS substrings are stored in LMS_c queues, they are moved between the L_c and S_c queues. LMS \$aacaa, stored in $\mathrm{LMS}_\$$, LMS aba, stored in LMS_a and LMS aba, stored in LMS_a. '\$' is removed from $\mathrm{LMS}_\$$ and stored in $\mathrm{BWT}_{\mathrm{S}_\$}$. Substring aacaa is moved to L_a. Likewise, 'a' is removed from L_a and stored in $\mathrm{BWT}_{\mathrm{L}_\mathrm{a}}$.

3.2.3 Algorithm S [Sirén, 2009]

Sirén [2009] describes a method for directly building the *compressed* SA (CSA) of a collection of strings, such as a collection of documents or genomes. In the worst case, this method requires $O(n \log n)$ time and $O(n)$ bits of extra space in addition to the space for the CSA.

We use the collection $\mathcal{M} = \{x_1, x_2\} = \{\mathrm{taa\$}_1, \mathrm{aat\$}_2\}$ as an example. The end of string symbol '$\$_1$' has a smaller lexicographical rank than the end of string symbol '$\$_2$'. The BWT is computed for the collection. The positions of the characters from string x_2 in the BWT are marked in an bit vector called I. As the second symbol 't' in BWT at position 1 is from x_2, position 1 in I is marked. The compact representation of this collection is built based on the compressed representation of two bit vectors. The bit vectors are based on character groups, namely, ψ_a and ψ_t in our example. ψ_a as the name suggests, marks each position of symbol 'a' in the BWT and ψ_t marks each position of symbol 't' in BWT.

	0	1	2	3	4	5	6	7
$\mathcal{M}$	t	a	a	$\$_1$	a	a	t	$\$_2$
SA	3	7	2	1	4	5	6	0
I	0	1	0	0	0	1	1	1
BWT	a	t	a	t	$\$_1$	a	a	$\$_2$
ψ_a	1	0	1	0	0	1	1	0
ψ_t	0	1	0	1	0	0	0	0

Each pair of distinct characters that are represented by the bit vectors is merged by making a scan over I. This method is thus inefficient for collections that have a large alphabet, as several scans are made over I to merge the character group based bit vectors. Sirén overcomes this inefficiency by combining the BWT of each string in the collection using the "backward search algorithm" (see Navarro and Mäkinen [2007]). For more details, we refer the reader to Sirén [2009].

3.2.4 Algorithm FGM [Ferragina et al., 2010; 2012]

Ferragina, Gagie and Manzini [Ferragina et al., 2010; 2012] describe an external memory algorithm to compute the BWT. Their algorithm computes the BWT for the first block of a certain size in RAM and stores it on disk. For the subsequent blocks, the BWT of the block is computed in RAM, and denoted as BWT_{int}. It is then merged with the BWT that is on disk, which we refer to as BWT_{ext}. This process is repeated until the entire BWT for the input string has been computed. Pseudocode is shown in Figure 3.4.

Input: x.
1: Compute BWT for the rightmost block.
2: Store BWT on external disk, BWT_{ext}.
3: **while** BWT not fully computed **do**
4: Compute BWT_{int} for the next block.
5: Merge BWT_{ext} and BWT_{int}.
Output: BWT.

Figure 3.4: Pseudocode to compute the BWT due to Ferragina et al. [2010; 2012].

Computing BWT for the first block

The algorithm of Ferragina et al. begins by dividing the input string into blocks of size m. In the example below we set $m = 3$. The blocks are numbered from right-to-left.

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	$
B		3			2			1		

The first block, $B_1 = x[6, 8]$, is brought into RAM and its BWT is derived. The BWT for the block is then stored on disk as BWT_{ext}.

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{int}										
BWT_{ext}							b	a	a	
SA							6	7	8	
B		3			2			1		

Computing BWT for the subsequent blocks

The BWT computation for subsequent blocks differs from the first block. The input string for the blocks $B_1 = x[6, 8]$ and $B_2 = x[3, 5]$ are brought into RAM. Suffixes are compared näively, character by character. For example, while comparing suffix 3 (aaaaab$) to suffix 4 (aaaab$), the order is resolved when a mismatch occurred at character 5, position 8 in the string. Having ordered the suffixes (that is, built the SA) for block B_2, the BWT is derived and stored in RAM as BWT_{int}.

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{int}				a	a	a				
BWT_{ext}							b	a	a	
SA				3	4	5	6	7	8	
B		3			2			1		

The next step is to find the number of suffixes of the already processed string (covered by previous blocks) that fall between the suffixes of the current block. Let $B_i = x[j, j + m - 1]$ be the current block, and let SA_B be it's SA. We compute an array G such that $G[i]$ is the number of suffixes of $x[j + m - 1, n]$ that are lexicographically smaller than $SA_B[i]$. G can be computed efficiently using the "backward search algorithm" (see Navarro and Mäkinen [2007]) on a suitably preprocessed BWT_{int}. In our example, the current block is B_2.

k	$x[k]$	G			
		0	1	2	3
	SA	3	4	5	
9	\$	1			
8	b				1
7	a				1
6	a				1

Using the G array for the SA of the first block, BWT_{ext} and BWT_{int} are merged. For instance, G[0] indicates there is one suffix that is lexicographically smaller than suffix 3 (in fact suffix 9 (\$)). As the BWT for this suffix is never computed, the entry for G[0] is ignored. $\text{G}[0,2] = 0$ and so, $\text{BWT}_{\text{int}}[0,2]$ is copied to disk. Then, the $\text{BWT}_{\text{ext}}[0,2]$ is copied (as indicated by G[3]) which shows that there are three suffixes that are lexicographically larger than suffix 5.

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	\$
BWT_{int}										
BWT_{ext}				b	a	a	a	a	a	
SA				3	4	5	6	7	8	
B		3			2			1		

Lastly, the BWT for B = 3 is computed. Similar to the previous round, $x[0,5]$ is brought into RAM and the SA for it is computed näively, via character comparisons, and in the case of a tie, a data structure that orders the suffixes in $O(m)$ time is used (see Ferragina et al. [2010; 2012] for implementation details).

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	\$
BWT_{int}	b	a	#							
BWT_{ext}				b	a	a	a	a	a	
SA	1	2	0	3	4	5	6	7	8	
B		3			2			1		

k	$x[k]$	G			
		0	1	2	3
	SA	1	2	0	
9	\$	1			
8	b			1	
7	a			1	
6	a			1	
5	a			1	
4	a			1	
3	a			1	

G[2] = 6 shows there are six suffixes that exist between SA[1] and SA[2]. Therefore, the BWT_{ext} is merged as below.

	0	1	2	3	4	5	6	7	8	9
x	(b	a	a)	(a	a	a)	(a	a	b)	\$
BWT_{int}										
BWT_{ext}	b	a	b	a	a	a	a	a	#	
SA	1	2	0	3	4	5	6	7	8	
B		3			2			1		

3.2.5 Algorithm BCR [Bauer et al., 2011]

Sirén [2009] describes a method for directly building the compressed suffix array of a collection of strings, such as documents or genomes. Bauer, Cox and Rosone [Bauer et al., 2011] consider a similar problem of suffix sorting on a collection of strings that is examined by Sirén, but each string in the collection is relatively short, in particular, a sequenced DNA fragment. Bauer et al. compute the BWT for a collection $\mathcal{M}$ consisting of m strings of k length. Their algorithm requires $O(m \log m)$ bits of memory and it runs in either $O(\text{sort } (m))$ or $O(km)$ time (depending on the implementation).

We use the collection $\mathcal{M} = \{x_1, x_2, x_3\} = \{\text{taa}\$_1, \text{cct}\$_2, \text{aaa}\$_3\}$ as an example. This collection consists of three strings where the lexicographical ranks of end of string symbols are as follows: $\$_1 < \$_2 < \$_3$. Let j-suffix of x_i be the last non-\$ symbol of the string whereas 0-suffix is the \$ symbol. The symbol that precedes j-suffix is a BWT symbol and is written into the h file that we call $\text{B}_j(h)$. $\text{B}_j(h)$ contains BWT symbols of j-suffixes that are of length j or less, and begins with $c_0 = \$$ and $c_h \in \Sigma$ for $h = 1, \ldots, \sigma$.

Computing BWT for the first round

For the first round, the partial BWT is computed by extracting characters preceding the 0-suffixes, that is, symbols that precede the end of string symbols. For example, the 0-suffix for suffix a$\$_1$ is $\$_1$ and the BWT symbol is 'a', which is written into $B_1(0)$. Likewise, symbol 't' precedes 0-suffix $\$_2$ and is inserted into $B_1(0)$.

$B_1(0)$	BWT	0-suffixes	suffixes
0	a	$\$_1$	a$\$_1$
1	t	$\$_2$	t$\$_2$
2	a	$\$_3$	a$\$_3$

Computing BWT for the subsequent rounds

For the subsequent rounds, the BWT symbols are inserted into the $B_2(h)$ files using LF mapping (see Ferragina and Manzini [2005] for details). The observation of their algorithm is similar to Ferragina et al. [2010; 2012]. The BWT symbols inserted in each round do not affect the relative order of the existing symbols in the partial BWT.

$B_2(0)$	BWT	0-suffixes	suffixes
0	a	$\$_1$	a$\$_1$
1	t	$\$_2$	t$\$_2$
2	a	$\$_3$	a$\$_3$
$B_2(1)$	BWT	1-suffixes	suffixes
0	a	a$\$_1$	aa$\$_1$
1	a	a$\$_3$	aa$\$_3$
$B_2(2)$	BWT	1-suffixes	suffixes
0	c	t$\$_2$	ct$\$_2$

The manner in which the BWT symbols are inserted for the third round is shown below using the LF function (Equation 2.1 on page 14). Say, to find the position to insert the BWT symbol, 'a' of suffix aa$\$_3$, $\text{rank}(\text{a}, \text{BWT}, 4) = 3$ is added to $\text{C[a]} = 3$ which gives 6. Therefore, 'a' must be inserted at position 6 in BWT but corresponds to position 3 in $B_3(1)$.

$B_3(0)$	BWT	0-suffixes	suffixes
0	a	$\$_1$	a$\$_1$
1	t	$\$_2$	t$\$_2$
2	a	$\$_3$	a$\$_3$
$B_3(1)$	BWT	2-suffixes	suffixes
0	a	a$\$_1$	aa$\$_1$
1	a	a$\$_3$	aa$\$_3$
2	t	aa$\$_1$	taa$\$_1$
3	a	aa$\$_3$	aaa$\$_3$
$B_3(2)$	BWT	2-suffixes	suffixes
0	c	ct$\$_2$	cct$\$_2$
$B_3(3)$	BWT	1-suffixes	suffixes
0	c	t$\$_2$	ct$\$_2$

3.3 Experiments

In this section, we provide experimental comparison on runtime and peak memory usage of the recent SACAs that we have efficient implementations for. These implementations are obtained from the websites of the authors themselves, except for Algorithm `K` that we implemented ourselves and Algorithm `NZC` (variable-length substrings) that is implemented by Mori [2008]. The performance of Algorithms `K`, `NZC` (variable-length substrings), `NZC` (fixed-length substrings), `OS` and `FGM` is measured.

These algorithms either construct the SA or BWT for a single string. `S` and `BCR` compute SA for a collection of strings. The latter benefits from fast disk access made possible by Solid State Disk (SSD), as reported in their paper. As our experimental machine does not have SSD, the performance of `BCR` is not measured relative to `S`.

3.3.1 Data and Setup

Table 3.1 shows the test data: 200MB of `ENGLISH` (english texts from *Gutenberg Project*, May 2005 download), `DNA` (bare DNA sequences from *Gutenberg Project*, June 2005 download) and `SOURCES` (source code from the selected LINUX and gcc distributions, June 2005 download), from the Pizza & Chilli corpus.[1] This well-known corpus contains benchmark datasets for testing algorithms.

All code was written in C/C++, compiled using gcc/g++ version 4.4.3 and the -O3

[1] http://pizzachili.dcc.uchile.cl/

optimization flag. The code was executed on an otherwise idle 3.16 GHz Intel® Core™ 2 test machine of 4 GB of RAM and 6144 KB of cache. The operating system was Ubuntu 10.04.3. Times reported are the minimum of three runs, measured with the C time function. We also report peak memory usage, measured using the memusage tool.

3.3.2 Results and Analysis

The runtimes are reported in Table 3.2 and the peak memory usage is reported in Table 3.3. RAM of 481 MB is allocated to the `K` and `FGM` algorithms. `NZC` variants (`NZC` (variable-length substrings) and `NZC` (fixed-length substrings)) require both the SA and input string to be resident in RAM and so, consume 1,000 MB of memory.

`NZC` (variable-length substrings) is clearly the fastest to build the SA. It uses an induced copying heuristic, and such methods are used in all fast algorithms. `NZC` variants also use twice the memory of other algorithms. `OS` computes the BWT directly and so, a small amount of RAM is used: each BWT character takes a single byte and an integer in SA takes 4 bytes (up to 8 bytes for larger datasets). The amount of RAM used by `OS` depends on the length of the LMS substrings. `K` requires only the input string to be in memory and for FGM, all the data including the input string resides in external memory (disk). The size of the available RAM of 4 GB on the test machine is sufficient to hold the input string and SA in memory, and so, `K` and `FGM` may get faster if they are tuned for RAM.

3.4 Summary

This chapter surveyed several SACAs that have emerged since the survey due to Puglisi, Smyth and Turpin [Puglisi et al., 2007]. The trend in these more recent algorithms is to use as little memory as possible by trading runtime. `NZC` variants are the most space efficient algorithms when compared to the 2007 survey, but the most space consuming algorithms in this chapter. `OS`, which is based on `NZC`, uses less space than `NZC` as it computes the BWT (and not the SA). `K` requires the text, small data structures (for efficient sorting) and a block

Files	σ	Mean LCP	Max LCP
`ENGLISH`	226	9,390	987,770
`DNA`	17	59	97,979
`SOURCES`	231	373	307,871

Table 3.1: LCP is the longest common prefix between adjacent suffixes in the SA. A higher Mean LCP generally increases the cost of suffix sorting.

Algorithms	ENGLISH	DNA	SOURCES
K	244	247	293
NZC (variable-length substrings)	**72**	72	**53**
NZC (fixed-length substrings)	169	166	116
OS	76	**65**	62
FGM	560	577	479

Table 3.2: Total wall clock time taken in seconds by the algorithms to run on the 200 MB dataset. The minimum time is shown in bold.

Algorithms	ENGLISH	DNA	SOURCES
K	481	481	481
NZC (variable-length substrings)	1000	1000	1000
NZC (fixed-length substrings)	1102	1119	1086
OS	**446**	**383**	**434**
FGM	481	481	481

Table 3.3: The peak memory usage (MB) by the algorithms for the 200 MB dataset. The minimum usage is shown in bold. OS *computes the BWT directly and so a small amount of RAM is used in contrast to the other algorithms that compute the SA. Each BWT character takes a single byte and an integer in SA takes 4 bytes.*

of SA to reside in RAM. In contrast, FGM requires the text to reside on disk, and small amount of RAM is used to sort a block of suffixes.

Our experiment results show that NZC (variable-length substrings) is the fastest as it uses a induced copying heuristic that is used in all fast algorithms. As the size of the RAM on the test machine is sufficient to hold the text and SA, K and FGM might have shown faster runtimes if they had been tuned for RAM. S and BCR are used to sort a collection of strings where the latter requires SSD for fast disk access. Since our test machine does not have SSD, the performance of BCR relative to S was not measured.

Chapter 4

Practical Efficient String Mining

The growth of genomic databases and other large volumes of textual data has led to the need for efficient algorithms for *string mining* problems. For example, given a multi-set (database) of strings $\mathcal{M} = \{s_1, s_2, \ldots, s_{|\mathcal{M}|}\}$ the *frequent string mining problem* is to report all patterns or substrings in $\mathcal{M}$ that occur in at least ω different strings, where ω is called the support of the pattern. This and other such string mining problems are analogs of classical data mining problems over itemsets.

In recent years, several algorithms for mining frequent and emerging substring patterns from databases of string data (such as proteins and natural language texts) have been discovered. Fischer, Huen and Kramer [Fischer et al., 2006], building on earlier work by Hui [1992], showed how the suffix array, combined with auxiliary information, could be used to solve frequency constrained string mining problems – that is, problems where our interest in a pattern is based solely on its frequency in the underlying database(s) – in optimal $O(n)$ time, using $O(n \log n)$ bits of space, where $n = \sum_{i=1}^{\mathcal{M}} |s_i|$ is the number of characters in the database. We refer to their approach as Algorithm `FHK`. Experimental results show this approach is fast in practice, but has difficulty scaling to large databases due to its unwieldy space requirements [Fischer et al., 2006]. To address this later problem, Fischer, Mäkinen and Välimäki [Fischer et al., 2008] applied recently developed techniques for compressing suffix arrays to the data structures of Fischer et al. [2006] and reduced space requirements to $O(n \log \sigma + |\mathcal{M}| \log n)$ bits. The price for these space savings in Algorithm `FMV` is an increase in asymptotic runtime to $O(n \log n)$ time, however in practice an even greater penalty is incurred: the compressed data structure is around two orders of magnitude slower than the `FHK` algorithm [Fischer et al., 2006]. We observe that these and the other recent algorithms

Table 4.1: Summary of theoretical and practical performance of previous string mining algorithms, and this chapter's contribution in the final row. n is the length of the input database and σ is its alphabet size. Theoretical space is in bits, practical space is in bytes. Practical time is given relative the original frequent linear algorithm in the first row [Fischer et al., 2006].

Reference	Theory		Practice	
	Space	Time	Space	Time
`FHK` [Fischer et al., 2006]	$O(n \log n)$	$O(n)$	22n	1
[Kügel and Ohlebusch, 2008]	$O(n \log n)$	$O(n)$	11n	0.5
[Weese and Schulz, 2008]	$O(n \log n)$	$O(n)$	11n	0.5
`FMV` [Fischer et al., 2008]	$O(n \log \sigma)$	$O(n \log n)$	3n	90
This chapter	$O(n \log \sigma)$	$O(n \log^2 n)$	2n	3

for string mining [Kügel and Ohlebusch, 2008; Weese and Schulz, 2008] lie at either extreme of the efficiency spectrum: they are either fast and use enormous amounts of space, or they are compact and orders of magnitude slower. Table 4.1 presents a summary of the asymptotic resource usage for existing approaches, with the result from this chapter listed in the final row.

In this chapter we present an algorithm that aims to achieve the best of both these extremes; having runtime near the fastest published algorithms, while using less space than the most space efficient ones. In the next section we set notation and define the data structures we will refer to throughout. Section 4.2 gives an overview of the basic `FHK` algorithm [Fischer et al., 2006]. Section 4.3 describes our new algorithm and Section 4.4 gives an empirical comparison to previous incarnations of Fischer et al.'s algorithms.

4.1 Preliminaries

A *string database* is simply a multiset of d strings, $\mathcal{M} = \{s_1, s_2, \ldots, s_d\}$, $s_i \in \Sigma^*$.

The suffix array for a string database $\mathcal{M} = \{s_1, s_2, \ldots, s_{|\mathcal{M}|}\}$ is the suffix array built from the string:

$$x_{\mathcal{M}} = s_1 \#_1 s_2 \#_2 \ldots \#_{|\mathcal{M}|-1} s_{|\mathcal{M}|},$$

formed by concatenating the database's contents, placing symbols $\#_i$ between each string, where these symbols do not occur in the database.[1] The algorithms we describe in later

[1] We build the SA for multiple databases in a similar way by concatenating the string for each database.

i	$x_{\mathcal{M}}$	SA	LCP	D	C	$x_{\mathcal{M}}$[SA[i]...]
0	G	24	-	0	0	\$
1	A	5	0	1	0	$\$\#_1$TAGA...
2	G	11	1	2	0	$\#_2$CTAG...
3	A	17	1	3	0	$\#_3$AGTA...
4	G	23	0	4	3	A\$
5	#	16	1	3	1	A$\#_3$AGT...
6	T	3	1	1	1	AG$\#_1$TA...
7	A	9	2	2	1	AG$\#_2$CT...
8	G	21	2	4	1	AGA\$
9	A	14	3	3	0	AGA$\#_3$A...
10	G	1	3	1	0	AGAG$\#_1$...
11	#	7	4	2	0	AGAG$\#_2$...
12	C	18	2	4	1	AGTAG...
13	T	12	0	3	4	CTAGA...
14	A	4	0	1	1	G$\#_1$TAG...
15	G	10	1	2	1	G$\#_2$CTA...
16	A	22	1	4	1	GA\$
17	#	15	2	3	0	GA$\#_3$AG...
18	A	2	2	1	0	GAG$\#_1$T...
19	G	8	3	2	1	GAG$\#_2$C...
20	T	0	3	1	0	GAGAG...
21	A	19	1	4	1	GTAGA
22	G	20	0	4	3	TAGA\$
23	A	13	4	3	0	TAGA$\#_3$...
24	\$	6	4	2	0	TAGAG...

*Figure 4.1: The suffix (*SA*), LCP,* D *and* C *arrays for database* $\mathcal{M} = \{GAGAG, TAGAG, CTAGA, AGTAGA\}$*. For clarity, the final column shows the (at most) first 5 characters of the suffix of* $x_{\mathcal{M}}$ *beginning at* SA[i]*, but is never actually stored.*

sections all require a data structure to map suffixes of $x_{\mathcal{M}}$ back to the strings s_j of $\mathcal{M}$. We use D to denote this suffix-to-string mapping, with D(SA[i]) = j if and only if suffix SA[i] of $x_{\mathcal{M}}$ begins in string $s_j \in \mathcal{M}$ (including the added #).

Figure 4.1 gives an example of these arrays for a small database containing four strings. The array C is explained in Section 4.2.1. For example, the final entry in the suffix array, SA[24] = 6, indicates that the suffix of $x_{\mathcal{M}}$ beginning at position 6 (TAGAG...) is the lexicographically largest of all suffixes of the string. The entry LCP[24] = 4 indicates that suffix 13 shares the first 4 characters with suffix 6 (TAGA). Note that every possible substring of the original database appears as a prefix of some suffix that is listed in SA, which facilitates us solving substring-mining problems.

The LCP array gives the length of the longest common prefix of adjacent suffixes in the

suffix array. Consider two arbitrary suffixes SA[i] and SA[j], $0 \leq i < j \leq n$. A well-known and useful result (see, for example, Manber and Myers [1993]; Kasai et al. [2001]) is that the longest common prefix of these two suffixes, lcp(x[SA[i], n], x[SA[j], n]), is the minimum value in LCP[$i + 1, j$]. For example, in Figure 4.1, the longest common prefix of SA[16] through SA[20] is 2 (GA), which is the minimum value in LCP[17, 20]. For this reason, we will often be interested in answering so-called *range minimum queries (RMQs)* on the LCP array. Formally,

$$\text{RMQ}(i, j) = \text{argmin}_{i<k\leq j}\text{LCP}[k].$$

There are several methods for preprocessing any array of n integers in $O(n)$ time to build a data structure of $O(n)$ bits such that future RMQs can be answered in constant time [Fischer, 2007].

4.2 Existing Algorithms

Algorithm `FHK` of Fischer et al. [Fischer et al., 2006] makes two passes from start to finish of the LCP array to calculate the support of all substrings in the database. The first pass computes and stores how often substrings repeat within a single string of the database. We call these *correction factors*, and store them in an array C. The second pass then computes how often all substrings occur within the whole database. By subtracting the correction factor from this number, you get the support for each substring. That is,

Support for substring s	=	Number of times s occurs in $\mathcal{M}$	−	Number of times s repeats within strings in $\mathcal{M}$.

4.2.1 Computing Correction Factors

The aim of the first pass over the LCP array is to calculate the number of times that substrings in the database repeat within individual strings. This information is stored in the C array, where if C[i] = k, then the substring beginning in position SA[i] of $x_{\mathcal{M}}$ and containing LCP[i] characters, repeats at least k times in some strings in $x_{\mathcal{M}}$. For example, in Figure 4.1, C[7] = 1 reflects that "AG" repeats once (occurs twice) in some string (s_2), as does C[8] and C[12]. Hence the total correction factor for the string "AG" is three. A second example: C[19] = 1 indicates that "GAG" repeats once in some string (s_1). Note that when

$\text{LCP}[i] = 0$, $\text{C}[i]$ has no ready interpretation, but can be set during the construction process (for example, $\text{C}[4] = 3$).

Input: LCP and D.
1: $\text{P}[0, |\mathcal{M}|] \leftarrow -1$
2: $\text{C}[0, n] \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** n **do**
4: $p \leftarrow \text{P}[\text{D}[i]]$
5: **if** $p > -1$ **then**
6: $j \leftarrow \text{RMQ}(p + 1, i)$
7: $\text{C}[j] \leftarrow \text{C}[j] + 1$
8: $\text{P}[\text{D}[i]] \leftarrow i$
Output: C.

(a) Pass 1: deriving the C array.

Input: LCP and C.
1: **for** $i \leftarrow 2$ **to** n **do**
2: **if** $\text{LCP}[i] < \text{LCP}[i - 1]$ **then**
2: **for** $\ell \leftarrow \text{LCP}[i - 1]$ **to** $\text{LCP}[i] - 1$ **do**
4: Find the largest $j < i$ and $\text{LCP}[j] < \ell$
5: Support is $i - j + 1 - \sum_{k=i}^{j} \text{C}[k]$
Output: Support values.

(b) Pass 2: computing support.

Figure 4.2: The two passes of unoptimized Algorithm `FHK`.

Fischer et al. give an efficient algorithm for calculating C, shown in Figure 4.2a, where the value of $\text{C}[\text{RMQ}(\text{P}[\text{D}[i]] + 1, i)]$ is incremented for all $1 \leq i \leq n$, and $\text{P}[i]$ stores the index of the last occurrence of $\text{D}[i]$ in $\text{D}[1, i - 1]$. The key observation, proved in [Fischer, 2007], allowing the algorithm to work is that if a substring of length ℓ repeats within string s_k, then there are at least two suffixes of $x_{\mathcal{M}}$ beginning with that substring, say at positions p and i, with $\text{D}[p] = \text{D}[i] = k$; now the minimum LCP value in the range $\text{LCP}[p + 1, i]$ must be ℓ. For example, in Figure 4.1, "A" repeats in s_4, and the minimum LCP value in the range $\text{LCP}[4 + 1, 8]$ is 1, occurring at $\text{LCP}[5]$, and this is recorded by increasing $\text{C}[5]$ by one.

In order to get the total number of times a substring s repeats in all of the strings in $x_{\mathcal{M}}$, one sums all of the C values for suffixes beginning with s whose LCP value is at least $|s|$. For example, "G" repeats 4 times (twice in s_1, once in s_2 and once in s_4), and the sum of all C entries in the range $[14, 21]$ with an LCP value greater than zero is 4. Similarly, "AG" repeats 3 times within strings in $x_{\mathcal{M}}$, and the sum of $\text{C}[6, 12]$ ignoring $\text{C}[6]$ is 3.

4.2.2 Computing Support

Once the correction factors are computed in C, another pass is made over the LCP array. When processing the LCP array from start to end, we know that when $\text{LCP}[i] \geq \text{LCP}[i+1]$, then the prefix of $\text{SA}[i]$ of length $\text{LCP}[i]$ characters repeats at least once in the database. For example, at position $i+1 = 21$ in Figure 4.1, the LCP value has fallen from $\text{LCP}[20] = 3$ to 1. This indicates that the 3 character prefix of the suffix beginning at $\text{SA}[20]$ (GAG) occurs at least twice in the database. By searching backwards from $\text{LCP}[i]$ towards $\text{LCP}[0]$, looking for the largest $j < i$ such that $\text{LCP}[j] < \text{LCP}[i]$, we can locate the range of suffixes in the SA whose prefix is equal to the first $\text{LCP}[i]$ characters of $\text{SA}[i]$. In this example, when $i = 20$, $j = 18$, and so the substring "GAG" must be the prefix of suffixes in the range $\text{SA}[18, 20]$.

This fact is also true for all prefixes of $\text{SA}[i]$ of length up to $\text{LCP}[i+1]+1$. For example, when $i = 20$, $\text{LCP}[i+1]+1 = 2$, and so the range in SA where substring "GA" is a prefix of a suffix can be found by locating the maximum $j < i$ where $\text{LCP}[j] < 2$. In this case $j = 16$, and so all suffixes in $\text{SA}[16, 20]$ begin with "GA".

The difference between j and i plus one gives the total count of the number of times the substring occurs in the database, and subtracting the sum of $\text{C}[j+1, i]$ gives the support. For example, "GA" has a support of (20-16+1)-1=4, and "GAG" has a support of (20-18+1)-1=2.

To avoid a linear-time scan backwards through the LCP array every time the condition $\text{LCP}[i] > \text{LCP}[i+1]$ is met when increasing i, Algorithm `FHK` keeps a stack of previous LCP values which grows (push) as LCP values increase during processing, and shrinks (pop) when LCP values decrease. This general notion has been exploited by many authors for different string processing problems on the SA [Abouelhoda et al., 2004; H.Bannai et al., 2004; Kasai et al., 2001]. Using the stack values allows constant time computation of j whenever a decrease in LCP values is encountered. Similarly, cumulative sums of C values can be computed as part of this pass, rather than in a separate pass, and stored on the stack.

In summary, `FHK` makes an initial pass of the LCP to compute C using RMQ, and then the second pass moves from left-to-right through the LCP array, pushing/popping information onto a stack of occurrence counts and correction factors which are used to compute support. In order to print out the actual strings that are identified with high support values, the algorithm also needs access to SA and the original text.

4.2.3 Memory Reductions

Fischer et al. published a modification to Algorithm `FHK` that removed the need for the C array to be explicitly computed and stored in a first pass over the LCP array [Fischer et al., 2008].

The key observation allowing C to be removed and computed as part of the support calculation pass was that the stack of LCP values that was accumulated to compute occurrence counts can also hold the C values required as correction factors. In this algorithm, when processing LCP[i], $p = \text{RMQ}(\text{P}[\text{D}[i]] + 1, i)$ is computed, but the result is stored in the topmost stack entry that has an LCP value of LCP[p], rather than being stored in C in pre-processing stage. Hence Algorithm `FMV` only requires one pass over LCP, but must maintain the P array of previous string occurrences throughout the pass, and must also search the stack at each step for the correct position for the correction factor.

Algorithm `FMV` also makes heavy use of compressed data structures, which further reduces the memory requirements from Algorithm `FHK`, but also significantly increases the time required to mine repeating strings.

4.3 New Algorithm

By observing that almost all accesses to SA, LCP, and D in `FMV` are left-to-right, allows us to propose several non-trivial modifications to the algorithm that result in large space savings, but which do not lead to an increase in running time over the original `FHK` algorithm. To exploit this observation we draw on a recent suffix sorting algorithm due to Kärkkäinen [2007] and its relationship to an LCP algorithm due to Puglisi and Turpin [2008]. We combine these two algorithms to efficiently obtain the SA and LCP arrays left-to-right one block at a time. After it is produced, each completed SA/LCP block is "fed" into the `FMV` algorithm. The memory for that block is then reused for subsequent blocks, enabling us to keep overall memory usage low — we never hold the entire SA or LCP array in memory.

The only complication introduced by this approach is answering the range minimum queries over LCP required to compute correction factors. It may be the case that the range of the query extends back into a block we have already discarded. We show how to service the necessary RMQs on-demand using a separate, small data structure, which we describe in Section 4.3.3.

We begin by describing how the SA and LCP can be efficiently obtained in blocks from left-to-right.

Input: A strict lower and loose upper bound on suffixes, s_ℓ and s_u, and $b_{\max}$.
1: B $\leftarrow$ {}
2: **for** $j \leftarrow 0$ **to** n **do**
3: **if** $s_\ell < x_{\mathcal{M}}[j, n] \leq s_u$ **then**
4: **if** $|\mathrm{B}| = b_{\max}$ **then**
5: **sort** the suffixes beginning at positions in B
6: discard B$[b_{\max}/2..b_{\max}]$
7: $s_u \leftarrow \mathrm{B}[b_{\max}/2 - 1]$
8: B $\leftarrow$ B $\cup\, j$
9: **sort** the suffixes beginning at positions in B and construct the LCP on the result
Output: B (including LCP), and s_u.

Figure 4.3: Algorithm GetBlock *used to extract a block of* SA *and* LCP *where all suffixes are greater than s_ℓ, and less than s_u, without exceeding an imposed block size of b_{max} suffixes.*

4.3.1 Difference Cover Sample

The first step in the new algorithm is to build the *Difference Cover Sample* of the input string [Burkhardt and Kärkkäinen, 2003; Kärkkäinen, 2007], $\mathrm{DCS}_v(x)$ (as explained on page 12).

4.3.2 Constructing SA and LCP in Blocks

Having built $\mathrm{DCS}_v(x)$ we proceed to (conceptually) divide the SA lexicographically into contiguous, non-overlapping *blocks* and make multiple passes over $x_{\mathcal{M}}$ populating each block in turn. In each pass we gather all the suffixes that belong in the current block. At the end of each pass we sort the suffixes we have collected and so form a contiguous section (block) of the SA. In order to allow for memory restrictions, particularly in the case where main memory is not large enough to hold $x_{\mathcal{M}}$, $\mathrm{DCS}_v(x_{\mathcal{M}})$, SA, LCP, and the required stack and P arrays, we impose a limit of $b_{\max}$ suffixes per block. Figure 4.3 shows Algorithm GetBlock, which extracts a single block of SA and LCP strictly bounded below by s_ℓ, and containing no suffix greater than s_u.

The approach is to simply collect suffixes as they are encountered in a left-to-right sweep of the text. There is no guarantee that the number of suffixes in the range $[s_\ell, s_u]$ is equal to $b_{\max}$, and so Step 4 checks if a block gets too full; if so, then the top half is discarded. This strategy guarantees that we always gather at least $b_{\max}/2$ suffixes in every pass, hence a total of $O(n/b_{\max})$ passes over the string are required to construct the full SA and LCP.

Input: String $x[0, n]$, $\mathrm{DCS}_v(x)$, and $b_{\max}$.
1: *stack* $\leftarrow \{\}$ and $\mathrm{P}[0, |M|] \leftarrow 0$
2: Populate Q, a queue of m splitters, from $\mathrm{DCS}_v(x)$
3: $s_\ell \leftarrow$ **dequeue**(Q)
4: *remaining* $\leftarrow n$
5: **while** *remaining* > 0 **do**
6: $\quad s_u \leftarrow$ **dequeue**(Q)
7: $\quad (\mathrm{B}, s_\ell) \leftarrow$ GetBlock $(s_\ell, s_u, b_{\max})$
8: $\quad$ Process B, maintaining *stack* and P
9: $\quad$ *remaining* $\leftarrow$ *remaining* $- |\mathrm{B}|$
Output: The incremental results from Step 8.

Figure 4.4: The algorithm used to apply `FMV` *to contiguous blocks of* SA *and* LCP*. This algorithm does not include the necessary detail to implement the RMQ over portions of the* LCP *into previous blocks, which is given in Section 4.3.3.*

Using the $\mathrm{DCS}_v(x)$ to aid comparisons, Step 3 can be achieved in $O(v)$ time, so the running time for GetBlock is $O(nv)$ in addition to the cost of any sort-and-throw-away executions in Steps 5 to 7. However, a further optimization is possible to speed up each pass, so that the total time for each pass used in collecting suffixes into blocks is $O(n)$ [Kärkkäinen, 2007]. We do not reproduce the algorithm here, simply observe that this faster approach is contingent on the skipping characters based on the upper splitter, s_u, in the spirit of the classic Boyer-Moore string searching algorithm.

Ideally, in order to gain maximum advantage of this speed up, each s_u should be the last element of its respective block. Of course this is not known, a priori, so the s_u values, or *splitters*, are estimated. It is conceivable that a distribution count could be undertaken as a pre-processing phase, choosing splitters that are lexiographically close, and then discarding splitters that lead to underful blocks. In this chapter we do not explore this option, but rather sample (in a periodic fashion) m splitters from the already sorted suffixes in the $\mathrm{DCS}_v(x)$. We will conveniently assume throughout that this (sorted) queue of splitters, Q, contains both the lexicographically smallest and largest suffixes in the collection. Removing this assumption at implementation time is not difficult. For each splitter $s_i \in \mathrm{Q}$ we use Algorithm GetBlock, altering s_ℓ and s_u as appropriate. Figure 4.4 outlines the process.

At the end of a scan in Step 7, the block B contains pointers to all the suffixes that fall lexicographically in the range $[s_\ell, s_u)$ in the suffix array, but they are not yet in any particular order and must be sorted to produce $\mathrm{SA}[\mathrm{SA}^{-1}[s_i], \mathrm{SA}^{-1}[s_{i+1}]]$. The sort is performed in two

phases. First, the suffixes are sorted to depth v using multikey quicksort (MKQS) [Bentley and Sedgewick, 1997]. Second, for any groups of suffixes that remain tied (that is, share common prefix $\geq v$ with at least one other suffix in the block) the sort is completed using the $\mathrm{DCS}_v(x_{\mathcal{M}})$, exploiting Lemma 1 (on page 12). The time required to sort a block of b suffixes this way is $O(vb + b \log b)$ [Kärkkäinen, 2007].

With a completed block $\mathrm{SA}[\mathrm{SA}^{-1}[s_i], \mathrm{SA}^{-1}[s_{i+1}]]$ in hand, obtaining the corresponding block of the LCP array — $\mathrm{LCP}[\mathrm{SA}^{-1}[s_i], \mathrm{SA}^{-1}[s_{i+1}]]$ — is straightforward. We scan the SA block and to compute each $\mathrm{LCP}[j]$ we compare the first v symbols of suffixes $\mathrm{SA}[j]$ and $\mathrm{SA}[j-1]$. If the suffixes mismatch in this v length prefix then we know $\mathrm{LCP}[j]$: it is simply the offset of the first mismatching symbol. If instead the suffixes share the same v length prefix, we determine their LCP using the extended Difference Cover Sample (Lemma 2 on page 13) in constant extra time. Thus, the time required to compute a block of b LCP values is $O(vb)$ in the worst case [Puglisi and Turpin, 2008].

Once a block of SA and LCP is completed, we can process that block to find repeating substrings and their support, using the stack and P arrays of the `FMV` method. Without reinitializing the stack nor the P array, we continue using the value returned from GetBlock as the s_ℓ for the next block. Note that while the stack and P array are easy to maintain as they are updated left-to-right in the original `FMV` algorithm, as written the algorithm does not allow for the RMQs over LCP values that may extend back into previous blocks. That is, P may contain indexes that are in previous blocks. We add an extra data structure to cope with this problem in the next section.

Using the above algorithm, the number of passes required is $O(n/b_{\max})$, and each pass requires $O(n)$ time, making $O(n^2/b_{\max})$ total for collecting. We collect $O(b_{\max})$ suffixes per pass and then we sort these in $O(vb_{\max} + b_{\max} \log b_{\max})$ time and then take a further $O(vb_{\max})$ time to compute LCP values. If we set $b_{\max} = n/\sqrt{v}$ the overall time simplifies to $O(n \log n + vn)$. The space required on top of the input string and DCS_v is $O(b_{\max}) = O(n/\sqrt{v})$ words for the SA and LCP blocks, and $O(n/b_{\max}) = O(\sqrt{v})$ words for the splitters, which is $O(n \log n/\sqrt{v} + \sqrt{v} \log n)$ bits.

4.3.3 Computing RMQs On Demand

Recall $\mathrm{P}[0, d]$ is an array of integers such that $\mathrm{P}[\mathrm{D}(\mathrm{SA}[i])] = j$ where $j < i$ is the largest j such that $\mathrm{D}(\mathrm{SA}[j]) = \mathrm{D}(\mathrm{SA}[i])$ or is -1 if no such j exists. In other words, $\mathrm{P}[j]$ is maintained to hold the position in D of the last occurrence of string s_j (initially -1).

We seek a data structure of a reasonable size that will quickly tell us the minimum value in LCP[P[D(SA[i])] + 1, i]. Fischer et al. [2006] preprocess the LCP array for constant time Range Minimum Queries for this purpose. Such a solution will not do for us however, as we never have the entire LCP array available to preprocess. We now describe a data structure to provide RMQ(P[D(SA[i])] + 1, i) on-the-fly, as needed, in $O(\log d)$ time, and using just $O(d)$ words of memory, where d is the number of strings in the database.

Our data structure, S, is essentially a stack which keeps track of at most f of the most recent potential minima in the LCP array and their positions; that is, the stack contains pairs (m_k, p_k) where m_k is the value of the k^{th} most recent possible minima (m_0 is the most recent, and is atop the stack) and p_k is where it occurred in LCP. The stack is maintained as follows. At a generic position i in the LCP array, if LCP[i] > LCP[$i-1$] we push the pair (LCP[i],i) onto the stack. Otherwise, if LCP[i] $\leq$ LCP[$i-1$] we pop elements from the stack while LCP[i] $< m_0$ and finally push the pair (LCP[i],p_{-1}) on, where p_{-1} is the position value of the last item popped. In many cases, the effect will be to just update value of m_0 to be LCP[i]. Observe that the elements popped in this action could not be the answer to any RMQ we may subsequently issue between the current position i and some other position $< i$ as they are all greater than the newly added item, which is in that range.

Note that when maintained this way the elements on the stack are always in descending order of m_k *and* p_k. Also, any given element on the stack indicates m_k is the minimum value in LCP[p_k, i]. These two observations mean that at any point we can output RMQ(P[D(SA[i])]+ 1, i) by binary searching the elements on the stack for the smallest $p_k >$ P[D(SA[i])] + 1, in $O(\log |\text{S}|)$ (this requires the stack to be implemented as an array).

We will now show how to bound the size of the stack to $O(d)$ items. This in turn bounds the time to answer minimum queries by binary searching the stack elements to $O(\log d)$. First, assuming the LCP can be an arbitrary sequence of n non-negative integers, observe that, if left unfettered, the stack can grow to size n with the sequence $1, 2, 3, 4, 5, ...$ The sequence has a potential minima at every position and each one is greater than the last so will be added to the stack. Secondly, observe that we should be able to get away with only d elements on the stack, because we are only ever interested in the range minimum in LCP between the current position i and the position of the last previous occurrence of D[i].

We let S grow to at most $2d$ items, at which point we perform a "clean up" step, which will always reduce S to at most d items. The clean up involves, for each P[j], searching for the element in S having the smallest $p_k >$ P[j] and marking these elements. We search for at most d items and so at most d items get marked. We remove all unmarked items, leaving at

most d items on the stack. In total, the clean up process requires $O(d \log d)$ time, which we can afford as the stack only requires a clean up at most every d elements. Our maintenance of the stack adds $O(\frac{n}{d} \log d)$ to the overall runtime, which is dominated by the time to sort each block. As the size of S is kept to $O(d)$ words the total extra space is $O(d \log n)$ bits.

4.3.4 Analysis

We now summarize the complexity of our approach.

Space Complexity. Our stack for on-demand RMQs, S, and the P array both require $O(d \log n)$ bits. We represent D, the string mapping, in $n + o(n)$ bits by using a bitarray preprocessed for rank and select queries [Clark, 1996] as in Fischer et al. [2008]. As the LCP array is still processed using the `FHK` approach (albeit one block at a time) we still require their stack (R). Using the compressed representation of stack R introduced for `FMV` [Fischer et al., 2008], the size of R is limited to $O(n)$ bits in total. [2] Apart from the input string, which is always held in memory, and requires $O(n \log \sigma)$ bits, the space complexities discussed in the preceding sections are dominated by the size of DCS_v and the SA and LCP blocks, all of which require $O(n \log n / \sqrt{v})$ bits. If we set $v = \log_\sigma^2 n$ the total space complexity becomes $O(n \log \sigma)$ bits.

Time Complexity. Runtime is dominated by the building of SA blocks, which overall requires $O(n \log n + vn)$ time. With $v = \log_\sigma^2 n$ as above, this is $O(n \log n + n \log^2 n) = O(n \log^2 n)$ time.

4.4 Experiments

This section reports the performance of our implementation, referred to as `NEW`, relative to `FMV` and `FHK` using the same experiments as reported by Fischer et al. [2008]. The purpose of the experiments is twofold. Firstly, they aim to assess the performance of the new algorithm relative to the previously published algorithms; and secondly aim to elucidate the effect on performance of the parameters v, the size of the difference cover, and b, the size of the blocks processed in each pass.

All code tested was written in C++ and compiled using the GNU g++ compiler with -O3 optimization level.

[2] In our prototype implementation used for the experiments in the next section we use an explicit, rather than compressed, stack; it remains relatively small in practice.

Table 4.2: Running time (seconds) for mining frequent patterns using FMV *($s = 16$),* FHK *and* NEW *($b = 5mill$, $v = 128$). ρ is the minimum support threshold for $\mathcal{M}_1$, and the maximum support threshold $\tau = 0.95$. The final column is the number of bytes output. The final row gives the memory usage in MB for the three methods.*

ρ	FHK	FMV	NEW	Out
0.1	135	12,132	337	3,559
0.2	135	12,121	336	1,211
0.3	136	12,150	335	953
0.4	136	12,127	337	694
0.5	136	12,170	337	436
0.6	136	12,118	335	196
0.7	136	12,121	335	49
0.8	136	12,341	336	7
0.9	136	12,132	335	2
MB	1,267	245	161	–

We used two different datasets. The first, PROTEIN, was obtained from the authors of Fischer et al. [2008], and contains $d = 133,984$ strings, $n \approx 53$ MB characters, and has $\sigma = 23$ different characters. The whole set contains 71,622 protein strings from a human database, and 62,362 from a mouse database. The second, GENOME, was the entire human genome, downloaded from the NCBI.[3] This contains $d = 24$ strings, $n = 2.8$ GB characters, with $\sigma = 5$.

4.4.1 Protein Data

Experiments on this dataset were conducted on an otherwise idle 3GHz Intel Xeon with 4 GB main memory and 1024 KB L2 Cache. The operating system was Fedora Linux running Kernel 2.6.9. Memory use was measured using the memusage utility, and times reported are the minimum of three runs.

Table 4.2 shows the times for FHK, FMV and NEW to extract frequent patterns with threshold ρ, replicating Table 1 from Fischer et al. [2008]. As expected, the times for FHK and FMV are comparable to those previously reported: an average of 3 hours 22 minutes and 2 minutes 16 seconds respectively. Our approach required 5 minutes 36 seconds to report the same, frequent strings. The memory use of the techniques is also remarkable. Even though NEW is

[3] ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/

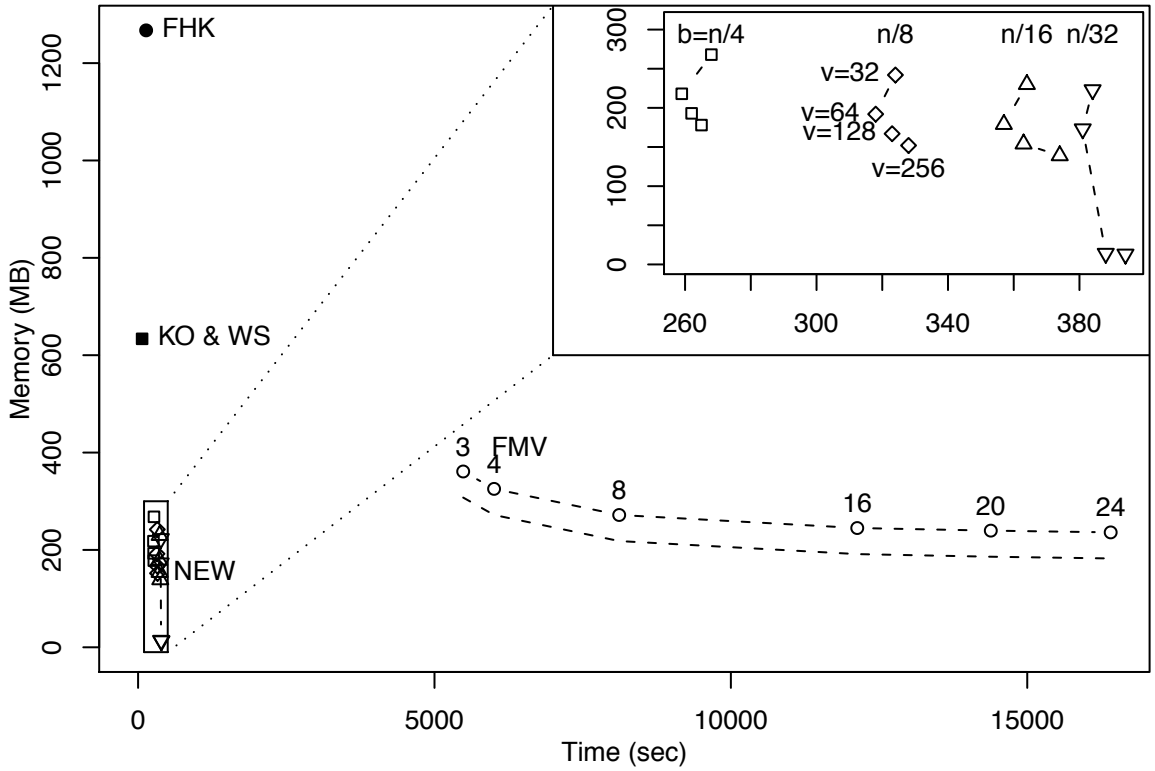

Figure 4.5: Time and memory required to mine PROTEIN *for frequent strings with* $\rho = 0.1$ *and* $\tau = 0.95$ *by the various algorithms. Note the point for KO [Kügel and Ohlebusch, 2008] and WS [Weese and Schulz, 2008] is indicative only: we did not implement and run their methods on this data. For the expanded area showing* `NEW` *in the top right, the blocksize for each group of 4 points is given above the points (*$n = 53.6$ *MB), and for each group of 4, the* v *parameter is as labelled for the* $n/8$ *case. Numbers above the open circles for* `FMV` *show the sample rate used. The plain dotted line is explained in the text.*

approaching the speed of `FHK`, it uses only 161 MB of memory: less than the space-efficient `FMV` algorithm that requires over 3 hours for the task!

Figure 4.5 summarizes this tradeoff for the frequent mining problem, and also includes the performance for `NEW` and `FMV` as their algorithmic parameters vary. `FMV` uses sample rate s as a parameter, which is shown above the open circles in the figure. As the sample rate increases, the space required by `FMV` decreases, but time increases accordingly. Note that our accounting of the space used by `FMV` seems to be about n bytes higher than that reported in Fischer et al. [2008]. We assume that the implementation we are using (obtained from Fischer himself) is slightly different to that used in their original paper. The dotted

Table 4.3: Running time (seconds) for mining emerging patterns using FMV ($s = 16$), `FHK` *and* `NEW` *($b = 5mill$, $v = 128$). ρ_s is the minimum support threshold, and ρ_g the growth rate.*

ρ_s	ρ_g	FHK	FMV	NEW
.1	1.33	135	12,121	335
.05	1.33	135	12,149	335
.01	1.33	135	12,191	335
.005	1.33	136	12,220	325
.001	1.33	141	12,354	336
.1	2.00	136	12,134	335
.05	2.00	135	12,138	335
.01	2.00	135	12,170	335
.005	2.00	136	12,177	335
.001	2.00	142	12,289	337

curve underneath the `FMV` curve in the figure shows the performance with n bytes subtracted, consistent with the original paper [Fischer et al., 2008].

For various parameter choices of blocksize b and difference cover size v, algorithm `NEW` is about as fast as `FHK`, but uses less space than `FMV`.

The magnified portion of the graph in the top right shows the affect of altering b and v on the resource usage of `NEW`. As blocksize is decreased from 13MB ($b = n/4$) down to 1.6 MB ($n/32$), running time increases, but memory use decreases slightly as less data is held in memory, regardless of v. For a fixed block size, increasing v decreases the amount of memory used, as $\mathrm{DCS}_v(x_{\mathcal{M}})$ reduces in size. Generally, as v increases, running times increase as the MKQS employed must do more operations per string comparison, as the strings can be up to v in length. Interestingly, as v is increased from 32 to 64, running times drop slightly, but this is more likely a quirk of the data, and we will investigate this further in future work.

Continuing our comparison with Fischer et al. [2008], Table 4.3 presents running times for mining emerging patterns (see Fischer et al. [2008] for definition). In this instance, the collection of human protein strings in PROTEIN is the positive database, and the collection of mouse protein strings in PROTEIN is the negative database. Again, our algorithm is slightly slower than `FHK`, but still much faster than `FMV` and the memory requirements remain as in Table 4.2.

Figure 4.6 confirms that the running time and memory usage of our algorithms scale in a similar way to Fischer et al. [2008].

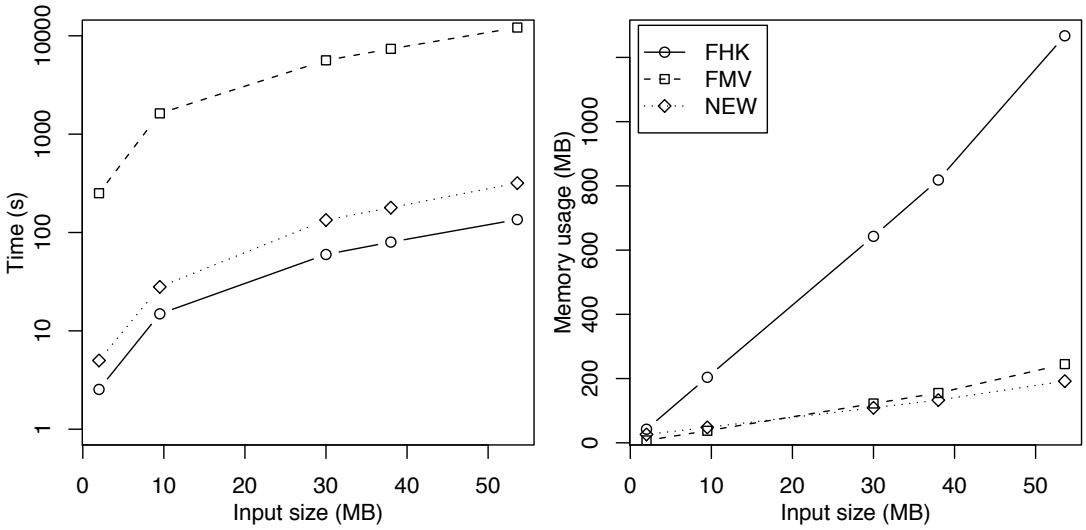

Figure 4.6: Time and memory required to mine frequent patterns from differing size prefixes of PROTEIN *using the three approaches ($\rho = 0.1$ and $\tau = 0.95$).* `NEW` *has $b = n/8$ and $v = 64$, while* `FMV` *is using $s = 16$.*

4.4.2 Genome Scale Data

Tests on this dataset were conducted on an otherwise idle Quad-Core AMD Opteron(tm) 2.3 GHz processor with cache size of 1 MB and 32 GB of RAM running Red Hat 4.1.2-44, and compiled with g++ version is 4.1.2. Memory use was measured using the `memusage` utility, and times reported are for three runs.

Table 4.4 shows the resources required by `FMV` and `NEW` to mine frequent strings from the GENOME database. The `FMV` algorithm required nearly 72 hours, and used 10 GB of memory,

Table 4.4: Resources required to mine the whole human genome for support $\rho = 0.9$, and $\tau = 1.0$. `NEW` *used $v = 128$.* † *Note that* `FHK` *was not actually run, but resource usage is as estimated by Fischer et al. [2008].*

Algorithm	Time	Memory (GB)
`FHK`	1h†	50.0†
`FMV`, $s = 16$	72h 12m	10.0
`NEW`, $b = n/2$	3h 4m	17.7
`NEW`, $b = n/4$	4h 27m	12.1
`NEW`, $b = n/8$	5h 55m	9.3
`NEW`, $b = n/16$	6h 4m	7.9

using a sample rate of $s = 16$. This is slightly slower than that report by Fischer et al. [2008], because their CPU was faster. `NEW` performed as expected: an order of magnitude faster than `FMV`, and using comparable memory.

4.5 Summary

In this chapter, we have presented an algorithm for mining substrings from a database of strings that is, in practice, is either much faster with comparable use of space, or uses much less space with comparable running time. Our main mechanism for keeping memory usage low is to build the enhanced suffix array incrementally, in blocks. Once built, a block is traversed to output patterns with required support before its space is reclaimed to be used for the next block. Asymptotically we require $O(n \log \sigma)$ bits of memory and $O(n \log^2 n)$ space to mine a database of total length n symbols drawn from an alphabet of σ possible symbols.

Chapter 5

Faster Semi-external Suffix Sorting

The suffix array (SA) provides efficient solutions to many problems of pattern matching and pattern discovery in string data. SA construction is a time and memory bottleneck in many applications and, as the size of strings requiring processing grows, more efficient construction algorithms are required. To date SA construction algorithms fall into two sets: large memory and small memory. Algorithms in the first set assume the SA fits in memory and are thus limited to smaller datasets. The fastest existing algorithms in this set [Itoh and Tanaka, 1999; Ko and Aluru, 2005; Maniscalco and Puglisi, 2007] require at least $5n$ bytes. The second, more recent set of algorithms use much less memory via compression or external memory [Ferragina et al., 2012; Kärkkäinen, 2007], at the price of a slower runtime.

In this chapter, we improve the runtime of a semi-external SA construction algorithm by Kärkkäinen [2007]. Section 5.1 describes the Kärkkäinen's algorithm and the algorithmic optimization techniques used. The main contribution of this chapter is a method for implementing the "pointer copying" heuristic from internal memory suffix array construction in a semi-external setting and is described in Section 5.2. A comprehensive experimental comparison is given in Section 5.3.

5.1 Kärkkäinen's Algorithm

The essence of Kärkkäinen's suffix sorting algorithm is similar to samplesort for integers. In samplesort, elements are randomly selected and sorted to become alternating lower bound, s_ℓ, and upper bound, s_u, splitters of buckets (delimited by them) with the remaining elements distributed into those buckets.

Kärkkäinen applies this idea to suffix sorting. Rather than distributing the elements

(suffixes) into all the buckets at once (which would consume too much space), Kärkkäinen builds and processes each bucket separately; first the leftmost bucket, is collected, sorted and written to disk to form a contiguous section of SA. The memory is then reused to process the next bucket.

Once we have built the *Difference Cover Sample* of the input string, $\mathrm{DCS}_v(x)$ (as explained on page 12), we choose a random set of suffixes from x and sort them. We call these suffixes *splitters*. Like in samplesort, they delimit the lower and upper bounds buckets. Although not mentioned in [Kärkkäinen, 2007], we found in practice it makes sense to choose the splitters from the already sorted $\mathrm{DCS}_v(x)$.

We now describe how a block of SA that we call B, lower-bounded by splitter, s_ℓ and upper-bounded by s_u, that is $[s_\ell, s_u)$ range is computed. Let $b_{\max}$ be the maximum number of suffixes we can collect in one round in B, determined by the amount of available RAM memory. The algorithm begins by making a left-to-right pass over string x, collecting and storing pointers of suffixes that have the same or higher lexicographical rank (lexrank) than s_ℓ but smaller than s_u. To determine if a suffix falls between the splitters efficiently, the splitters are preprocessed using a *Knuth-Morris-Pratt (KMP)-like failure function* before falling back to $\mathrm{DCS}_v(x)$ so that at most v character comparisons are required and the time for each scan is $O(n)$. Pseudocode for the KMP-like failure function is presented in Figure 5.1.

At the end of a generic round, B contains pointers to the suffixes that fall between the range $[s_\ell, s_u)$, but not necessarily in lexrank. The suffixes in B are then sorted using multikey quicksort (MKQS) [Bentley and Sedgewick, 1997] until the depth of v where a mismatch obviously orders the suffixes. But, if there is a tie after v comparisons, $\mathrm{DCS}_v(x)$ is used to order them. The time required to sort b suffixes this way is $O(vb + b \log b)$ [Kärkkäinen, 2007]. B is then written to disk as a contiguous section of the SA.

We demonstrate Kärkkäinen's suffix sorting algorithm using an example string x, znefniinzznefr\$. Assume $b_{\max}$ is 4 and the splitters are stored in an array $\mathrm{Q} = \{14, 6, 13\}$. So, for the first round, s_ℓ is $x[14..14] = \$$ and s_u is $x[6..14] =$ inzznefr\$. When a left-to-right pass is made over x, the first suffix, suffix 0 (znefniinzznefr\$) is ignored as it has a higher lexrank than s_u but the pointer for suffix 2 (efniinzznefr\$) is stored in B as it fits in the range $[s_\ell, s_u)$. This process continues until the suffixes in the range are collected.

However, there is no guarantee that the splitters in Q divide the SA evenly because they are randomly chosen (from a lexicographic point of view). That is, the number of suffixes placed in B may reach $b_{\max}$ before the scan completes. In our example, suffix 14 (\$) cannot be stored in the current $\mathrm{B} = \{2, 3, 5, 11\}$ as it is full. Kärkkäinen provides a clever method for

Input: x, s_u, and LCP$[0, v-1]$.
1 : $i \leftarrow 0$
2 : $j \leftarrow -1$
3 : $k \leftarrow -1$
4 : **while** $i \leq n$ **do**
5 : **if** $i > k$ **then**
6 : $k \leftarrow i$
7 : $l \leftarrow 0$
8 : **else**
9 : $l \leftarrow$ LCP$[i-j]$
10: **if** $i + l \equiv k$ **then**
11: **while** $l < m$ **and** $k < n$ **and** $s_u[l] \leftarrow x[k]$ **and** $l \leq v$ **do**
12: $k \leftarrow k + 1$
13: $l \leftarrow l + 1$
14: **if** $l \equiv v$ **then**
15: Use DCS$_v(x)$ to resolve the order between $s_u[l]$ and $x[k]$.
16: **break** from loop.
17: $j \leftarrow i$
18: **else if** $i + l > k$ **then**
19: $l \leftarrow k - i$
20: $j \leftarrow i$
21: **if** $l \neq m$ **and** $(i + l \equiv n$ **or** $x[i + l] < s_u[l])$ **then**
22: Found the suffix that is smaller than s_u, that is suffix i.
23: $i \leftarrow i + 1$
Output: Suffixes that are lexicographically smaller than s_u.

Figure 5.1: Pseudocode for KMP-like failure function used in Kärkkäinen's suffix sorting algorithm [Kärkkäinen, 2007]. This same function can easily be modified to preprocess the v-length prefix of suffix splitters which is stored in the LCP array.

dealing with this problem. If, while collecting suffixes for the current block, $b_{\max}$ is reached, the scan is halted and the content of the current block is sorted. The lexicographically larger half of the block is then discarded, the median suffix in the block becomes the new s_u, and the scan resumes. When this occurs, we say the block has a *split*. This method does not asymptotically increase the number of scans, although practical execution time increases, as more rounds are required to compute the entire SA.

Thus, B is sorted and the median in B $= \{2, 11, 3, 5\}$ which is suffix 3 is chosen to be the new s_u. Suffix 5 is discarded as it is lexicographically larger than the current s_u and the scan resumes. At the end of the round, B $= \{2, 11, 14\}$ is sorted in the usual way and written to

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA_1	14	2	11												
SA_2	14	2	11	3	12	5									
SA_3	14	2	11	3	12	5	6	1	10						
SA_4	14	2	11	3	12	5	6	1	10	4	7				
SA_5	14	2	11	3	12	5	6	1	10	4	7	13	0	9	8

Figure 5.2: SA_i is the portion of the SA constructed after round i when processing string x, znefniinzznefr\$, $b_{max} = 4$, and using initial splitters $\mathrm{Q} = \{14, 6, 13\}$.

disk as SA_i where i represents the round the portion of SA is constructed (see Figure 5.2). The memory that was used to hold them (during collection) is reclaimed for the subsequent round. The previous s_u becomes the new s_ℓ and s_u is assigned suffix 6, that is Q[1] as we never collected the suffixes within the said range, since there was a split.

The algorithm requires $O(n \log n + vn)$ time and $O(n \log n/\sqrt{v})$ bits of space in addition to the text. Via v and $b_{\max}$ the algorithm also allows for different space-time tradeoffs, depending on the amount of available memory.

We now describe our first optimization. Using suffixes as splitters as Kärkkäinen described is a good general approach to dividing lexicographic space, which in turn allows the SA to be built one block at a time. In practice however, we found a significant boost to runtime is possible if one simply uses symbol frequencies (or bigram frequencies) to divide the suffixes into lexicographic blocks. More precisely, we make a scan of x and count the frequency of each symbol. Let $\mathrm{C}[0..\sigma]$ be an array containing these symbol frequencies. We then (conceptually) partition the suffix array by determining k symbols, $t_0 < t_1 < \ldots < t_k$, such that, for $i < k$, $\sum_{j=t_i}^{t_{i+1}} \mathrm{C}[t_i] < b_{\max}$ and $\sum_{j=t_i}^{t_{i+2}} \mathrm{C}[t_i] > b_{\max}$. Consecutive selected symbols t_i and $t_i + 1$ are later used as (respectively) upper- and lower-bound splitters. This approach is faster because suffix inclusion in a block is determined with a single symbol comparison, not several, as can be the case with the use of the KMP-like failure function. Moreover, preprocessing of the v-length prefix of suffix splitters is avoided. Of course there can be inputs for which the frequency of one symbol exceeds $b_{\max}$, but these are rare in practice. In such cases it is easy to have the algorithm fall back to using suffix splitters, as in Kärkkäinen's original approach. In order to enable this when counting symbols, we also collect a single suffix that starts with the consecutive selected symbols (t_i and $t_i + 1$).

5.2 Pointer Copying

Pointer Copying is the name given to a broad class of heuristic methods for suffix sorting that derives the order of some suffixes from the order of other suffixes whose order is already known. The method was first introduced by Itoh and Tanaka [1999] and now many different styles are known [Seward, 2000; Ko and Aluru, 2005; Maniscalco and Puglisi, 2007]. All pointer copying methods are designed for large memory use. We adapted these to work in a semi-external setting. We describe how we adapted the Itoh and Tanaka's method; other methods work similarly.

Itoh and Tanaka distinguish each suffix as type U or type V. If $(x[i] \leq x[i+1])$ then suffix $x[i..n]$ is type U, otherwise $(x[i] > x[i+1])$ it is type V. We illustrate these types below on our example string x, znefniinzznefr\$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x	z	n	e	f	n	i	i	n	z	z	n	e	f	r	\$
type	*V*	*V*	*U*	*U*	*V*	*U*	*U*	*U*	*U*	*V*	*V*	*U*	*U*	*V*	*U*

	\$	e	f	i	n	r	z
j	0	1	2	3	4	5	6
C	0	2	4	6	10	11	14

Importantly, for suffixes with equal first letter, type V suffixes always precede type U suffixes in SA. Let $\mathrm{C}[0, \sigma]$ be an array of integers such that $\mathrm{C}[c]$ is the total number of symbols less than c in the input string x. Thus $\mathrm{SA}[\mathrm{C}[c]..\mathrm{C}[c+1]]$ is the area of the suffix array where the group of suffixes that start with symbol c belong. Itoh and Tanaka first scan x and calculate C, then they scan x again to place the U suffixes at the end of the appropriate groups, as determined by C. They then sort the U suffixes with MKQS. To complete the computation of SA, it remains to sort the V suffixes. To do this, Itoh and Tanaka scan the partially complete SA, which to begin with has only U suffixes in it, from left-to-right. For each suffix $j = \mathrm{SA}[i]$ encountered in the scan, if $j-1$ is a V suffix, it belongs at position $\mathrm{C}[x[j-1]]$ (i.e. the start of its group). We place it there, setting $\mathrm{SA}[\mathrm{C}[x[j-1]]] = j-1$, and then increment $\mathrm{C}[x[j-1]]$.

We now describe how we adapted Itoh and Tanaka's approach to work in external memory. First, we initialize a file of n integers, which will eventually contain SA, to contain n sentinel ('$\perp$') values. There are two hurdles to using Itoh and Tanaka's approach in a semi-external setting. The first is that we cannot necessarily afford to use MKQS to sort the U suffixes,

as the U suffixes for a character group may exceed the amount of available memory. We can jump this hurdle by using Kärkkäinen's algorithm, and restricting it to only collect and sort the U suffixes. This is easy to do because the type (U or V) of a suffix is determined in constant time. The U suffixes are placed in the correct (and final) positions on disk with the help of the C array, from which we can tell the end of each group of suffixes with the same first letter.

The second problem is to mitigate the non-local memory accesses to SA that occur in Itoh and Tanaka's algorithm when moving the V suffixes into place in the final phase of the algorithm. Our approach here is to delay writing a V suffix to its final position in SA as soon as it is known, but instead to buffer V suffixes in memory, buffering as many as available memory will allow us to hold. Because we must buffer both the V suffix pointer and its position in SA, we can buffer at most $b_{\max}/2$ suffixes. When memory becomes full, we halt the scan of SA and write all the buffered V suffixes to their final positions. This requires at most σ non-sequential accesses to disk each time, and $n\sigma/b_{\max}$ overall.

5.3 Experiments

We implemented our improvements to Kärkkäinen's algorithm and measured practical performance on real data. These experiments aim to show the effect of the parameters v, and b. Ferragina, Gagie and Manzini's external memory approach [Ferragina et al., 2012] was used as a baseline. We say these experiments are at the *character* level as SA or BWT is computed for each character of data.

But, in some types of data, particularly natural language, pattern matching is done at a *word* rather than character level. We say a word is a maximal sequence of alpha-numeric characters. A second set of experiments gauges the efficacy of Kärkkäinen's algorithm for suffix sorting at the word level. The performance of another SA construction algorithm suitable for integer strings by Larsson and Sadakane [2007] serves as a baseline. We use the abbreviations: `K` for Kärkkäinen's algorithm; `D` for `K` optimized as described at the end of Section 5.1; `D-XX` for `D` with pointer copying method `XX`; `FGM` for Ferragina et al.'s external memory algorithm; `LS` for Larsson and Sadakane's SA algorithm.

Table 5.1: Files used for testing. A higher LCP generally increases the cost of suffix sorting. The last four columns give the percentage of suffixes that require explicit sorting for various pointer copying methods.

Dataset	Size	Mean LCP	Max LCP	σ	IT	KA	MP	S
PROT	1.1	713	343,628	27	54	50	16	54
LAW	5.0	3,977	692,921	256	53	49	14	53
WIKI	8.8	6,012	556,673	128	50	48	15	50
GUTEN	3.1	37,006	2,475,053	256	50	48	16	50
HUMAN	2.9	214,002	20,289,999	7	64	45	13	64

Table 5.2: Description for the words data corpus which consists of LAW *and* WIKI *files from Table 5.1. The second column is the size of the original file in GB. The number of integer word tokens is in the third column.*

Dataset	Size	Total words	Distinct words
LAW	5.0	762,754,420	3,507,436
WIKI	5.0	861,471,533	2,608,830

5.3.1 Data and Setup

For testing we used the files in Table 5.1. The files are used in other papers and are available in public repositories.[1][2][3] PROT (December 2006 download) is a collection of protein base sequences from the *Swissprot database* and GUTEN (September 2005 download) is a concatenation of English text files from *Gutenberg Project*. WIKI is a crawl of English Wikipedia articles (early 2009 version). In contrast, LAW is a crawl of HTML pages from the *UK* domain (2006-2007). HUMAN (March 2008 version) is the entire human genome in bases.

In character level experiments, the first 1 GB prefix from each file in the corpus is used. In word level experiments, maximal sequences of alpha-numeric characters were converted to integers for different prefixes of LAW and WIKI, see Table 5.2.

All code was written in C++, compiled with g++ -O3. Peak memory was measured with memusage. Times are the minimum of three runs, measured with C time. The experimental machine was a 3.4 GHz Intel Core i7-2600, with Ubuntu 12.04, 4 GB RAM (for character level experiments)/8 GB RAM (for word level experiments), 8192 KB cache, and a Seagate Momentus SATA 500GB 7200 RPM disk.

[1] ftp://ftp.ncbi.nih.gov/genomes/
[2] http://pizzachili.dcc.uchile.cl/
[3] http://boston.lti.cs.cmu.edu/Data/clueweb09/

Table 5.3: Character-level experiment: $v = 128$ and b varies. Running time in minutes (in brackets, peak memory usage in GB) on 1 GB prefixes of files.

Dataset	$b_{\max}$	K	D	D-IT	D-KA	D-MP	D-S	FGM
PROT	$n/4$	29 (3.4)	14 (3.4)	12 (3.4)	27 (3.5)	21 (3.5)	16 (3.5)	21 (3.5)
	$n/8$	28 (2.4)	14 (2.4)	11 (2.4)	19 (2.5)	17 (2.5)	14 (2.5)	26 (2.5)
	$n/16$	34 (2.2)	14 (2.2)	12 (2.2)	19 (2.3)	17 (2.3)	13 (2.2)	26 (2.4)
	$n/32$	41 (2.2)	15 (2.2)	13 (2.2)	20 (2.3)	17 (2.3)	13 (2.2)	26 (2.3)
	$n/64$	53 (2.2)	17 (2.2)	14 (2.2)	21 (2.3)	18 (2.3)	14 (2.2)	26 (2.2)
LAW	$n/4$	50 (3.4)	23 (3.4)	16 (3.4)	28 (3.5)	24 (3.5)	24 (3.5)	20 (3.5)
	$n/8$	41 (2.4)	23 (2.4)	16 (2.4)	23 (2.5)	19 (2.5)	21 (2.5)	24 (2.5)
	$n/16$	46 (2.2)	23 (2.2)	16 (2.2)	22 (2.3)	19 (2.3)	20 (2.2)	24 (2.4)
	$n/32$	86 (2.2)	27 (2.2)	19 (2.2)	23 (2.3)	21 (2.3)	22 (2.2)	25 (2.3)
	$n/64$	109 (2.2)	31 (2.2)	21 (2.2)	26 (2.3)	20 (2.3)	23 (2.2)	25 (2.2)
WIKI	$n/4$	52 (3.4)	27 (3.4)	18 (3.4)	21 (3.5)	26 (3.5)	26 (3.5)	20 (3.5)
	$n/8$	52 (2.4)	26 (2.4)	17 (2.4)	17 (2.5)	21 (2.5)	22 (2.5)	25 (2.5)
	$n/16$	62 (2.2)	26 (2.2)	18 (2.2)	17 (2.3)	21 (2.3)	21 (2.2)	25 (2.4)
	$n/32$	72 (2.2)	27 (2.2)	18 (2.2)	18 (2.3)	21 (2.3)	20 (2.2)	25 (2.3)
	$n/64$	106 (2.2)	31 (2.2)	19 (2.2)	19 (2.3)	22 (2.3)	21 (2.2)	24 (2.2)
GUTEN	$n/4$	14 (3.4)	13 (3.4)	10 (3.4)	12 (3.5)	20 (3.5)	16 (3.5)	21 (3.5)
	$n/8$	16 (2.4)	12 (2.4)	10 (2.4)	10 (2.5)	17 (2.5)	13 (2.5)	29 (2.5)
	$n/16$	27 (2.2)	13 (2.2)	10 (2.2)	11 (2.3)	17 (2.3)	12 (2.2)	29 (2.4)
	$n/32$	36 (2.2)	13 (2.2)	11 (2.2)	11 (2.3)	18 (2.3)	12 (2.2)	29 (2.3)
	$n/64$	45 (2.2)	20 (2.2)	14 (2.2)	14 (2.3)	18 (2.3)	15 (2.2)	29 (2.2)
HUMAN	$n/4$	15 (3.4)	11 (3.4)	11 (3.4)	11 (3.5)	18 (3.5)	23 (3.5)	22 (3.5)
	$n/8$	24 (2.4)	11 (2.4)	10 (2.4)	10 (2.5)	17 (2.5)	11 (2.5)	27 (2.5)
	$n/16$	29 (2.2)	17 (2.2)	15 (2.2)	14 (2.3)	19 (2.3)	14 (2.2)	27 (2.4)
	$n/32$	34 (2.2)	24 (2.2)	19 (2.2)	17 (2.3)	20 (2.3)	19 (2.2)	27 (2.3)
	$n/64$	47 (2.2)	36 (2.2)	27 (2.2)	22 (2.3)	22 (2.3)	26 (2.2)	27 (2.2)

Table 5.4: The running time in minutes (in brackets peak memory usage in GB). The algorithm by Larsson and Sadakane [2007] takes 6 minutes with 5.7 GB on LAW *and 10 minutes with 6.4 GB on* WIKI*.*

	LAW				WIKI			
$b_{\max}$	$v = 128$	$v = 256$	$v = 512$	$v = 1024$	$v = 128$	$v = 256$	$v = 512$	$v = 1024$
$n/4$	14 (4.6)	17 (4.5)	20 (4.5)	24 (4.4)	19 (5.2)	21 (5.1)	25 (5.0)	27 (5.0)
$n/8$	15 (3.9)	17 (3.8)	21 (3.8)	25 (3.7)	20 (4.4)	23 (4.3)	25 (4.2)	29 (4.2)
$n/16$	16 (3.7)	19 (3.5)	22 (3.4)	28 (3.4)	21 (4.2)	25 (4.0)	30 (3.8)	30 (3.8)
$n/32$	17 (3.7)	19 (3.5)	26 (3.3)	31 (3.2)	22 (4.2)	25 (4.0)	31 (3.7)	31 (3.6)
$n/64$	20 (3.7)	26 (3.5)	31 (3.3)	38 (3.2)	26 (4.2)	30 (4.0)	32 (3.7)	39 (3.6)

5.3.2 Results and Analysis

Results of character level experiments for varying b and fixed $v = 128$, are shown in Table 5.3. Note that from $b = n/16$ onwards, peak memory usage is the same regardless of the block size because memory is dominated by $\mathrm{DCS}_v(x)$. The peak memory of Kärkkäinen's suffix sorting algorithm at any given time is $n + 2|\mathrm{DCS}_v(x)|$. There is a slight difference in the reported peak memory usage for `D-KA`, `D-MP` and `D-S`. For `D-KA` and `D-MP`, a bit vector of size n is needed to differentiate the type U and V suffixes in $O(1)$ time.

Times for `K` are similar when $b = n/4, n/8, n/16$ due to underfilled blocks. If splitters are not carefully chosen, blocks are underfull at the end of each scan, and more scans are required. Likewise, overfull blocks result in splits (part way through a scan), again leading to more scans overall. Both cases increase runtime. `D` outperformed `K` as blocks are relatively full due to our splitter selection approach, and as expected, splitter comparisons are faster than the failure-function used in `K`.

When pointer copying methods are added to `D`, we expected the method that explicitly sorts the least suffixes to be fastest, as with large memory SA algorithms. That is, we expected $|\texttt{MP}| \leq |\texttt{KA}| \leq |\texttt{IT}| \leq |\texttt{S}|$, where $|\texttt{XX}|$ is the time for method `XX`. However, we discovered that right-to-left scans to induce the order of suffixes are expensive when the SA is on disk. Both `KA` and `MP` use right-to-left scans. `D-IT`, which uses a left-to-right scan, speeds up `D` by about 30%.

Compared to the external memory `FGM` algorithm, we are generally faster when available memory is higher (i.e. smaller v) and always at least as fast. Our memory usage is $2.4n$ bytes of memory, which was used to set `FGM`'s memory.

The difference covers period, v affects overall runtime and peak memory usage. Up to v comparisons are made via MKQS before employing $\mathrm{DCS}_v(x)$ to resolve the order. There is a space-time tradeoff here. The larger the value of v, the fewer suffixes are chosen to become sample suffixes and so, less memory is required. However, for a dataset with high Mean LCP, the sorting time increases with v.

Table 5.4 shows the performance of the `D-IT` and `LS` algorithms for the large alphabet data. While `LS` is faster than `D-IT` at all parameter settings, `LS` does not allow a space-time tradeoff, and so `D-IT` can operate at lower memory levels. Memory can often be the limiting factor in many tasks (more so than time).

5.4 Summary

SA construction is a time and memory bottleneck in many application and, as the size of strings requiring processing grows, more efficient construction algorithms are required. In this chapter, we improve the runtime of a semi-external SA construction algorithm by Kärkkäinen [2007]. We achieve a speed up of 2-4 times without increasing the memory usage of the algorithm. Four pointer copying methods ([Itoh and Tanaka, 1999; Seward, 2000; Ko and Aluru, 2005; Maniscalco and Puglisi, 2007]) were adapted to work in a semi-external setting. We are up to twice as fast as the next fastest algorithm by Ferragina, Gagie and Manzini [Ferragina et al., 2012] when working memory is equated. That is, the same amount of memory is made available to the algorithms so that their running time can be compared.

The performance of the improved algorithm is also measured on strings that have large datasets. We are 2-3 times slower on such strings than Larsson and Sadakane [2007] (the best published algorithm for strings with large alphabets), but we use less memory. In many applications (for instance bioinformatics experiments) we may be willing to wait a long time (days or weeks say) to compute a result, but if our algorithm uses more memory than is available we will not be able to compute the result at all. In such applications memory (not time) is the limiting factor.

Chapter 6

Pointer Copying for Longest-Common-Prefix

The longest-common-prefix (LCP) array is often used with the SA to simulate bottom-up and top-down traversals of the suffix tree for string processing problems, within the same time bounds but with less space overhead [Abouelhoda et al., 2004; Puglisi and Turpin, 2008]. Pointer copying is a technique which has been heavily used to speed up the construction of SA, but not LCP array.

In this chapter, we discuss our attempts in combining a fast, space efficient LCP construction algorithm due to Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009] with three pointer copying techniques (Itoh and Tanaka [1999], Seward [2000] and Ko and Aluru [2005]). Section 6.1 describes the LCP array construction algorithm by Kärkkäinen et al. [2009], and the way we combined the algorithm with pointer copying is described in Section 6.2. Related work is in Section 6.3. A comprehensive experimental comparison is given in Section 6.4.

6.1 Kärkkäinen et al. [2009] Algorithm

Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009] reduce the practical time and space cost of LCP construction by storing the values in position order in the *permuted LCP (PLCP) array*, instead of the classic lexicographical rank. The PLCP$[0, n]$ is the same as LCP with an exception on the order of its content. In particular, for every $j \in [0, n]$,

$$\text{PLCP}[\text{SA}[j]] = \text{LCP}[j].$$

The LCP values can then be simulated from the PLCP. The *sparse PLCP array*, called PLCP_q, stores every q^{th} entry of the PLCP such that $\text{PLCP}_q[i] = \text{PLCP}[iq]$. The remaining entries are computed using the following lemma.

Lemma 3 ([Kärkkäinen et al., 2009])**.** *For any $i \in [0, n]$, let $a = \lfloor i/q \rfloor$ and $b = i \bmod q$, i.e., $i = aq + b$. If $(a + 1)q \leq n - 1$, then $\text{PLCP}_q[a] - b \leq \text{PLCP}[i] \leq \text{PLCP}_q[a + 1] + q - b$. If $(a + 1)q > n - 1$, then $\text{PLCP}_q[a] - b \leq \text{PLCP}[i] \leq n - i \leq q$.*

Thus, at most $q + \text{PLCP}_q[a + 1] - \text{PLCP}_q[a]$ comparisons (or at most q comparisons if $((a + 1)q > n - 1)$) are made, and this number can be close to n for some i. For strings that have a high Mean LCP, obtaining the missing values via comparisons will affect the running time of the algorithm. The amortized number of comparisons over all i is at most q as shown in the following lemma.

Lemma 4 ([Kärkkäinen et al., 2009])**.** *Assuming the text and the SA are available, the sparse PLCP array PLCP_q supports random access to LCP values in $O(q)$ amortized time.*

Sequential accesses are made to the SA and the LCP array is built sequentially with some random accesses made to the PLCP_q array. This is fast in practice although it has a time complexity of $O(nq)$ and $O(n/q)$ words of space are required, excluding the space required for the text and SA.

6.2 Pointer Copying

We next describe how we combined the Itoh and Tanaka [1999] pointer copying method to the LCP construction algorithm of Kärkkäinen, Manzini and Puglisi using our example input string $x =$ cbabcdbabcbab\$. Other pointer copying methods (Seward [2000] and Ko and Aluru [2005]) are combined in a similar way.

6.2.1 Collecting Type U LCPs

As explained in the previous chapter, Itoh and Tanaka classify each suffix as being type *U* or type *V*. A suffix is type *U* if the first character of the suffix has a smaller or equal lexicographical rank (lexrank) than its rightmost suffix ($x[i] \leq x[i + 1]$). Otherwise, the suffix has a higher lexrank than its rightmost suffix ($x[i] > x[i + 1]$) and is type *V*. We illustrate these types below on our example string.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x	c	b	a	b	c	d	b	a	b	c	b	a	b	$
type	*V*	*V*	*U*	*U*	*U*	*V*	*V*	*U*	*U*	*V*	*V*	*U*	*V*	*U*

Suffix 0 (cbabcdbabcbab$) is of type *V* as it is lexicographically larger than suffix 1 (babcdbabcbab$). If a suffix is a type *V* suffix, the LCP for the suffix is of type *V* as well. Recall for suffixes with equal first letter, type *V* suffixes always precede type *U* suffixes in SA, and similarly for the LCP. The algorithm of Kärkkäinen et al. is used to compute the type *U* LCPs. They are then placed in the LCP array using the C array that stores the starting positions of the type *U* LCPs as shown below.

	$	a	b	c
C	0	1	8	12

LCP	Suffixes	SA	j
0	$	13	0
0	ab$	11	1
2	abcbab$	7	2
3	abcdbabcbab$	2	3
⊥	b$	12	4
⊥	bab$	10	5
⊥	babcbab$	6	6
⊥	babcdbabcbab$	1	7
1	bcbab$	8	8
2	bcdbabcbab$	3	9
⊥	cbab$	9	10
⊥	cbabcdbabcbab$	0	11
1	cdbabcbab$	4	12
⊥	dbabcbab$	5	13

In the above example, position 0 in the LCP is undefined as there is no previous suffix to be compared, and is indicated with 0. The LCP between SA[1] and SA[0], that is suffix 11 (ab$) and suffix 13 ($) is 0, as no common prefix is shared. 0 is then placed at position 1 in the LCP as suffix 11 begins with the letter 'a', and C[a] is 1. C[a] is then incremented so that the LCP for the next suffix that begins with the letter 'a' can be placed correctly. This process is repeated until all type *U* LCPs are placed in the LCP array. Only for illustration purposes, type *V* LCPs are indicated with '⊥'.

6.2.2 Deriving Order of Type V LCPs

```
Input:    x, SA and LCP.
1 : for j ← 0  to n do
2 :     maintain the S stack.
3 :     i ← SA[j]
3 :     if i > 0 then
4 :         if x[i] > x[i − 1] then
5 :             if LCP not been derived for letter group x[i − 1] then
6 :                 P[x[i − 1]] ← j
7 :                 LCP[C[x[i − 1]]] ← 0
8 :                 C[x[i − 1]] ← C[x[i − 1]] + 1
9 :             else
10:                 m_k ← RMQ(P[x[i − 1]] + 1,j)
11:                 P[x[i − 1]] ← j
12:                 LCP[C[x[i − 1]]] ← 1+S[m_k]
13:                 C[x[i − 1]] ← C[x[i − 1]] + 1
Output:   LCP.
```

Figure 6.1: Pseudocode for deriving the LCP. The S *refers to the stack used in the RMQ data structure (defined on page 42).*

Pseudocode for deriving the type V LCPs from the type U LCPs is presented in Figure 6.1. Once the type U LCPs are placed in the LCP, the type V LCPs are derived in a single scan over the SA. A left-to-right ($j = 0 \ldots n$) scan is made over SA, and for each i ($i = \text{SA}[j]$), we inspect suffix $i - 1$ and if it is a type V suffix, the length of the common prefix within its single letter group until position j is computed. The value is then written (with some adjustments which we explain later for clarity) to the position indicated by $\text{C}[x[i-1]]$ array (see Steps 3 to 13) as shown below.

	b	c	d
C	4	10	13

The length of the common prefix within the its single letter group is found efficiently using the RMQ data structure (defined on page 42). Recall S stack stores the the most recent potential minima values in LCP and their positions; that is, the stack contains pairs (m_k, p_k) where m_k is the value of the k^{th} most recent possible minima (m_0 is the most recent, and is atop the stack) and p_k is where it occurred in LCP. The stack is maintained as follows. At a generic position j in the LCP array, if $\text{LCP}[j] > \text{LCP}[j-1]$ we push the pair (LCP[j],j)

onto the stack. Otherwise, if LCP[j] $\leq$ LCP[$j-1$] we pop elements from the stack while LCP[j] $< m_0$ and finally push the pair (LCP[j],p_{-1}) on, where p_{-1} is the position value of the last item popped. In many cases, the effect will be to just update value of m_0 to be LCP[j]. Observe that the elements popped in this action could not be the answer to any RMQ we may subsequently issue between the current position j and the other position $< j$ stored in P[$x[i-1]$] as they are all greater than the newly added item, which is in that range. P[$x[i-1]$] stores the current position in the LCP (see Steps 6 and 11).

LCP	Suffixes	SA	j
0	\$	13	0
0	ab\$	11	1
2	abcbab\$	7	2
3	abcdbabcbab\$	2	3
0	b\$	12	4
0+1	bab\$	10	5
2+1	babcbab\$	6	6
$\perp$	babcdbabcbab\$	1	7
1	bcbab\$	8	8
2	bcdbabcbab\$	3	9
$\perp$	cbab\$	9	10
$\perp$	cbabcdbabcbab\$	0	11
1	cdbabcbab\$	4	12
$\perp$	dbabcbab\$	5	13

Continuing with the example, when $j = 0$, i = SA[0] = 13, is a suffix beginning with '\$' whose LCP is undefined, that is 0 as it is the first suffix in the LCP array. In contrast, suffix $i - 1 = 13 - 1 = 12$ is a type *V* suffix beginning with 'b' whose LCP has not been derived. LCP[0] = 0 is thus copied to position 4 as C[b] = 4, before incrementing its group counter. P[$x[i-1]$] is set to 0 as j is 0. Likewise, when $j = 1$, i = SA[1] = 11, is a suffix beginning with 'a' whose LCP is 0. This value is then copied to position 5, indicated by C[b] as suffix $11 - 1 = 10$ is a type *V* suffix whose LCP has not been derived. But, it is the second entry in the in the $x[i-1]$ = b group and so, an additional 1 is added to the derived LCP as it obviously shares the prefix 'b' with the previous consecutive entry. P[$x[i-1]$] is set to 1 as j is 1. For i = SA[2] = 7, suffix $7 - 1 = 6$ is a type *V* suffix beginning with 'b' whose LCP has not been derived. The RMQ data structure is used to find the length of the common prefix

between SA[2] and SA[1]. That is, RMQ(P[$x[i-1]]+1, 2$) is 2 as these consecutive suffixes share the prefix "ab". This value is then copied to position 6 as C[b] = 6. 1 is then added to LCP[6] as this entry obviously shares the prefix 'b' with the previous consecutive entry and so, LCP[6] = 2 + 1 = 3. This process is repeated until the all type V LCPs have been derived.

The Kärkkäinen et al. algorithm, combined with pointer copying has the same time complexity of the original algorithm as left-to-right scans (or sometimes, right-to-left scans as for the Ko and Aluru [2005] pointer copying technique) are made, with the RMQ data structure taking $O(\log |\mathrm{S}|)$ time per query. Space is mainly used to store the LCP array ($4n$ bytes) as the original algorithm by Kärkkäinen et al. overwrites SA with LCP, but SA is required here for pointer copying.

6.3 Related Work

Recently, Fischer [2011] published an approach similar to ours. The pointer copying technique due to Nong, Zhang and Chan [Nong et al., 2009b] is combined with Kärkkäinen et al.'s algorithm. The RMQ is found efficiently using the stack due to Gog and Ohlebusch [2011]. One subtle but important difference of Fischer's approach to ours is that he saw a clever way to use unused space in the LCP which we did not see. He stored the values of the sparse PLCP array, PLCP_q in the LCP and later, overwrote them with the correct LCP values. Thus, his total memory usage is $9n$ bytes.

6.4 Experiments

We implemented several different versions of our improvements to Kärkkäinen, Manzini and Puglisi algorithm and measured its practical performance using the data of Section 6.4.1. These experiments aim to elucidate the effect on performance of the q. The performance of Fischer [2011] is also measured as a second baseline. We use the abbreviations: `KMP` for Kärkkäinen et al.'s algorithm; `KMP-XX` for `KMP` combined with the pointer copying `XX`; `F` for Fischer's algorithm.

6.4.1 Data and Setup

For testing, we used the files in Table 6.1. The files are available in public repositories.[1] The files can be categorized based on the source generated, that is, Artificial, Pseudo-Real and Real texts.

As the name suggests, the Artificial texts are artificially generated via some mathematical definitions (i.e. `FIBONACCI`, `RUN RICH` and `THUE MORSE`). In contrast, Pseudo-Real texts are generated by repeating real texts (i.e. `XML`, `ENGLISH` and `PROTEINS`) from the Pizza & Chilli corpus. Real texts are DNA sequences from three species. That is, Saccharomyces Paradoxus (`PARA`), Saccharomyces Cerevisiae (`CERE`) and Escherichia Coli (`ECOLI`).

All code was written in C/C++, compiled using g++ version 4.6.3 with the -O3 optimization flag. Experiments were run on an otherwise idle 3.40GHz Intel® Core™ i7-2600 of 4 GB of RAM and 8192 KB of cache. The operating system was Ubuntu 12.04 running Kernel 3.2.0-25-generic. Times reported are the minimum of three runs, measured with the C getrusage function. Peak memory usage is the sum of data structures reported by the C sizeof function.

Table 6.1: The LCP, size (in MB) and σ is for our dataset that is terminated with the special end of string character (see text). A higher LCP generally increases the cost of suffix sorting. The last four columns give the percentage of LCPs that require explicit sorting for various pointer copying methods.

Dataset	Size	Mean LCP	Max LCP	σ	IT	KA	S	NZC
FIBONACCI	255.5	70,711,161	165,580,139	2	62	38	62	38
RUN RICH	206.7	44,038,468	114,413,063	2	62	50	62	38
THUE MORSE	256.0	32,156,331	67,108,864	2	67	50	67	33
XML	100.0	95,781	510,561	89	50	49	50	28
ENGLISH	100.0	987	11,222	106	51	49	51	31
PROTEINS	100.0	991	11,091	21	54	50	54	32
CERE	439.9	7,080	303,204	5	66	48	66	26
ECOLI	107.5	11,322	698,433	15	62	48	62	28
PARA	409.4	3,275	104,177	5	65	48	65	27

6.4.2 Results and Analysis

Tables 6.2, 6.3 and 6.4 show the runtimes in seconds and space usage in MB. As all methods need to build the SA, these times are not included.

[1] http://pizzachili.dcc.uchile.cl/repcorpus.html

Recall parameter q controls the memory used, which further affects the runtimes. If q is large, less memory is used by the sparse PLCP array, which further reduces the speed of the algorithm as more values are computed via comparisons. As expected, these comparisons become a bottleneck for `KMP` in files that have a high Mean LCP (repetitive text), in contrast to the pointer copying variants that use RMQs to compute the value. It is then copied to the position indicated by the group counters stored in C.

However, Fischer who independently developed the same approach to us reported less space as he used the unused space in the LCP array to store the sparse PLCP array and later, overwrote it with the correct values. This clever way ensured his total memory usage to be $9n$.

In theory, we expect the pointer copying technique that requires the least number of LCP to be explicitly sorted to be fastest $|\texttt{NZC}| \leq |\texttt{KA}| \leq |\texttt{IT}| \leq |\texttt{S}|$ (see Table 6.1). That is, `NZC` has the least number of LCP to be explicitly sorted and should therefore be the fastest and so on. However, in practice, there is only a small difference in runtimes reported between these technique. This could perhaps be due to the fact that these experiments were run in RAM memory. Although the pointer copying techniques require a lot of random accesses to text, these accesses are fast when done in RAM in contrast to disk.

6.5 Summary

This chapter describes our attempts to introduce pointer copying methods to a fast, space efficient LCP construction algorithm due to Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009]. We experimented with three pointer copying techniques (Itoh and Tanaka [1999], Ko and Aluru [2005] and Seward [2000]) and showed that pointer copying, an approach that is used to speed up SA construction, does speed up LCP array construction as well, especially on repetitive texts when the q parameter is large. However, Fischer [2011] who came up with the same idea to us independently, used less space as he used the unused space in the LCP array to store the sparse PLCP array, PLCP_q. This clever way ensured his total memory usage to be $9n$ regardless of the q value used.

Table 6.2: Running time in minutes (in brackets, peak memory usage in MB) on `Artifical` *texts.*

Dataset	q	KMP	KMP-IT	KMP-KA	KMP-S	F
FIBONACCI	4	8 (2555.0)	21 (2555.2)	31 (2555.2)	21 (2555.2)	29 (2299.5)
	8	9 (2427.3)	21 (2427.4)	31 (2427.4)	21 (2427.4)	
	16	9 (2363.4)	21 (2363.5)	31 (2363.5)	20 (2363.5)	
	32	11 (2331.5)	22 (2331.6)	32 (2331.6)	22 (2331.6)	
	64	17 (2315.5)	26 (2315.6)	34 (2315.6)	25 (2315.6)	
	128	16 (2307.5)	25 (2307.6)	34 (2307.6)	25 (2307.6)	
	256	13 (2303.5)	24 (2303.7)	32 (2303.7)	23 (2303.7)	
	512	59 (2301.5)	52 (2301.7)	50 (2301.7)	51 (2301.7)	
	1024	59 (2300.5)	52 (2300.7)	50 (2300.7)	51 (2300.7)	
	2048	59 (2300.0)	52 (2300.2)	50 (2300.2)	51 (2300.2)	
RUN RICH	4	7 (2067.1)	17 (2067.2)	22 (2067.2)	16 (2067.2)	24 (1860.4)
	8	7 (1963.7)	17 (1963.8)	22 (1963.8)	17 (1963.8)	
	16	8 (1912.0)	17 (1912.2)	22 (1912.2)	17 (1912.2)	
	32	9 (1886.2)	18 (1886.3)	22 (1886.3)	17 (1886.3)	
	64	10 (1873.3)	19 (1873.4)	23 (1873.4)	18 (1873.4)	
	128	10 (1866.8)	19 (1866.9)	23 (1866.9)	19 (1866.9)	
	256	16 (1863.6)	22 (1863.7)	26 (1863.7)	22 (1863.7)	
	512	37 (1862.0)	35 (1862.1)	37 (1862.1)	35 (1862.1)	
	1024	55 (1861.2)	46 (1861.3)	45 (1861.3)	46 (1861.3)	
	2048	164 (1860.8)	113 (1860.9)	100 (1860.9)	113 (1860.9)	
THUE MORSE	4	11 (2560.0)	23 (2560.1)	27 (2560.1)	23 (2560.1)	27 (2304.0)
	8	10 (2432.0)	22 (2432.1)	26 (2432.1)	22 (2432.1)	
	16	10 (2368.0)	22 (2368.1)	25 (2368.1)	21 (2368.1)	
	32	9 (2336.0)	21 (2336.1)	25 (2336.1)	21 (2336.1)	
	64	9 (2320.0)	21 (2320.1)	25 (2320.1)	21 (2320.1)	
	128	8 (2312.0)	20 (2312.1)	24 (2312.1)	20 (2312.1)	
	256	7 (2308.0)	19 (2308.1)	24 (2308.1)	19 (2308.1)	
	512	5 (2306.0)	18 (2306.1)	23 (2306.1)	18 (2306.1)	
	1024	5 (2305.0)	18 (2305.1)	23 (2305.1)	18 (2305.1)	
	2048	5 (2304.5)	18 (2304.6)	23 (2304.6)	18 (2304.6)	

Table 6.3: As of Table 6.2, but on `Pseudo-Real` *texts.*

Dataset	q	KMP	KMP-IT	KMP-KA	KMP-S	F
XML	4	4 (1000.0)	9 (1000.1)	10 (1000.1)	9 (1000.1)	9 (900.0)
	8	4 (950.0)	9 (950.1)	10 (950.1)	9 (950.1)	
	16	5 (925.0)	9 (925.1)	10 (925.1)	9 (925.1)	
	32	5 (912.5)	9 (912.6)	10 (912.6)	9 (912.6)	
	64	7 (906.3)	10 (906.4)	11 (906.4)	10 (906.4)	
	128	10 (903.1)	12 (903.3)	13 (903.3)	11 (903.3)	
	256	17 (901.6)	15 (901.7)	16 (901.7)	15 (901.7)	
	512	33 (900.8)	23 (900.9)	24 (900.9)	23 (900.9)	
	1024	62 (900.4)	38 (900.5)	38 (900.5)	37 (900.5)	
	2048	116 (900.2)	65 (900.3)	64 (900.3)	64 (900.3)	
ENGLISH	4	5 (1000.0)	9 (1000.1)	11 (1000.1)	9 (1000.1)	10 (900.0)
	8	5 (950.0)	9 (950.1)	11 (950.1)	9 (950.1)	
	16	5 (925.0)	9 (925.1)	11 (925.1)	9 (925.1)	
	32	6 (912.5)	10 (912.6)	11 (912.6)	10 (912.6)	
	64	8 (906.3)	11 (906.4)	12 (906.4)	10 (906.4)	
	128	12 (903.1)	13 (903.3)	14 (903.3)	12 (903.3)	
	256	18 (901.6)	16 (901.7)	17 (901.7)	16 (901.7)	
	512	31 (900.8)	23 (900.9)	23 (900.9)	22 (900.9)	
	1024	51 (900.4)	33 (900.5)	33 (900.5)	32 (900.5)	
	2048	75 (900.2)	46 (900.3)	45 (900.3)	45 (900.3)	
PROTEINS	4	5 (1000.0)	9 (1000.1)	11 (1000.1)	9 (1000.1)	10 (900.0)
	8	5 (950.0)	9 (950.1)	10 (950.1)	9 (950.1)	
	16	5 (925.0)	9 (925.1)	11 (925.1)	9 (925.1)	
	32	6 (912.5)	10 (912.6)	11 (912.6)	9 (912.6)	
	64	8 (906.3)	11 (906.4)	12 (906.4)	10 (906.4)	
	128	11 (903.1)	12 (903.3)	14 (903.3)	12 (903.3)	
	256	18 (901.6)	16 (901.7)	17 (901.7)	16 (901.7)	
	512	30 (900.8)	23 (900.9)	23 (900.9)	23 (900.9)	
	1024	50 (900.4)	34 (900.5)	33 (900.5)	33 (900.5)	
	2048	74 (900.2)	47 (900.3)	45 (900.3)	46 (900.3)	

Table 6.4: As of Table 6.2, but on Real texts.

Dataset	q	KMP	KMP-IT	KMP-KA	KMP-S	F
CERE	4	26 (4399.2)	44 (4399.3)	52 (4399.3)	43 (4399.3)	47 (3959.3)
	8	26 (4179.2)	44 (4179.3)	51 (4179.3)	43 (4179.3)	
	16	28 (4069.2)	46 (4069.4)	52 (4069.4)	44 (4069.4)	
	32	32 (4014.2)	48 (4014.4)	54 (4014.4)	46 (4014.4)	
	64	41 (3986.7)	54 (3986.9)	58 (3986.9)	52 (3986.9)	
	128	57 (3973.0)	65 (3973.1)	66 (3973.1)	62 (3973.1)	
	256	89 (3966.1)	86 (3966.3)	81 (3966.3)	84 (3966.3)	
	512	148 (3962.7)	127 (3962.8)	109 (3962.8)	123 (3962.8)	
	1024	253 (3961.0)	199 (3961.1)	159 (3961.1)	193 (3961.1)	
	2048	430 (3960.1)	320 (3960.2)	243 (3960.2)	311 (3960.2)	
ECOLI	4	6 (1074.7)	10 (1074.8)	12 (1074.8)	10 (1074.8)	10 (967.2)
	8	6 (1021.0)	10 (1021.1)	12 (1021.1)	10 (1021.1)	
	16	7 (994.1)	11 (994.2)	12 (994.2)	10 (994.2)	
	32	8 (980.7)	11 (980.8)	12 (980.8)	11 (980.8)	
	64	10 (973.9)	12 (974.1)	13 (974.1)	12 (974.1)	
	128	13 (970.6)	14 (970.7)	15 (970.7)	14 (970.7)	
	256	18 (968.9)	18 (969.0)	17 (969.0)	17 (969.0)	
	512	28 (968.1)	24 (968.2)	22 (968.2)	23 (968.2)	
	1024	42 (967.6)	33 (967.8)	29 (967.8)	32 (967.8)	
	2048	64 (967.4)	47 (967.6)	39 (967.6)	45 (967.6)	
PARA	4	25 (4093.8)	42 (4093.9)	49 (4093.9)	40 (4093.9)	45 (3684.4)
	8	25 (3889.1)	42 (3889.2)	48 (3889.2)	40 (3889.2)	
	16	27 (3786.8)	43 (3786.9)	49 (3786.9)	41 (3786.9)	
	32	31 (3735.6)	45 (3735.7)	51 (3735.7)	44 (3735.7)	
	64	38 (3710.0)	50 (3710.1)	54 (3710.1)	49 (3710.1)	
	128	53 (3697.2)	60 (3697.3)	62 (3697.3)	58 (3697.3)	
	256	82 (3690.8)	80 (3690.9)	75 (3690.9)	77 (3690.9)	
	512	135 (3687.6)	115 (3687.7)	101 (3687.7)	111 (3687.7)	
	1024	231 (3686.0)	180 (3686.1)	147 (3686.1)	174 (3686.1)	
	2048	396 (3685.2)	290 (3685.3)	226 (3685.3)	281 (3685.3)	

Chapter 7

Scalable Inverse Burrows-Wheeler transform

In this chapter, we consider the problem of inverting the Burrows-Wheeler transform (BWT) efficiently in external memory. The only previous work on the problem is due to Ferragina, Gagie and Manzini [Ferragina et al., 2010]. However, no implementation of their algorithm exists. Section 7.2 describes an implementation of Ferragina et al.'s approach and we explore a variant in Section 7.3. Then, in Section 7.4 we describe several new inversion algorithms of our own which utilize scanning and compression techniques. A comprehensive experimental comparison on a wide range of large datasets is given in Section 7.5.

7.1 Basic BWT Algorithm

To invert the BWT in RAM, the basic BWT inversion algorithm begins at the symbol in the BWT which is the last symbol of the input string. The inverse algorithm recovers the symbol of the original string in reverse order. The position of this symbol is recorded when the forward transform is performed. The character at that position is output, and then the LF function (Equation 2.1 on page 14) is used to find the `next` character (of the input string). We illustrate this process in Figure 7.1 using the BWT of input string $x =$ flooorrreeeencccee\$.

In the example, position 10 is the `starting` point for inversion. The '\$' symbol is output as the first character in array x^r. Using the LF function, rank$(\$, \mathrm{L}, 10)$ is added to C[\$] which gives the `next` inversion point as 0. The BWT character at position 0 is output as $x^r[1] = \mathrm{e}$. Continuing the example, rank$(\mathrm{e}, \mathrm{L}, 0)$ is 0 as there are no occurrences of 'e' in prefix $\mathrm{L}[0, 0]$ and so, 0 is added to C[e] which gives the `next` decoded character $x^r[2] = \mathrm{e}$.

	$	c	e	f	l	n	o	r
j	0	1	2	3	4	5	6	7
C	0	1	4	10	11	12	13	16

(a) The C array, which is derived in a single pass over L.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	e	n	c	c	e	c	r	e	e	e	$	f	e	l	o	o	r	r	o
rank(L[i], L, i)	0	0	0	1	1	2	0	2	3	4	0	0	5	0	0	1	1	2	2
`next`	4	12	1	2	5	3	16	6	7	8	0	10	9	11	13	14	17	18	15
x^r	$	e	e	c	c	c	n	e	e	e	e	r	r	r	o	o	o	l	f

(b) The position 10 in the L array is the `starting` point for inversion. When C is combined with rank as in the LF function, the `next` character of the input string is output.

Figure 7.1: The two arrays: C *and* L *that are required by the LF function to invert the BWT.*

The movement through the L array in the above example can be likened to a *linked list*, a set of nodes where each node is chained to the next node with an exception of the last node [Sedgewick, 1999]. The order in which these nodes are traversed is very important as the predecessor must be visited before the successor. This problem of determining the order for a given set of nodes is known as *"list ranking"* [Chiang et al., 1995; Sibeyn et al., 1999; Ferragina et al., 2010], and is similar to our problem of inverting the BWT where each position can be represented as a node that is chained to the `next` position as shown in Figure 7.2.

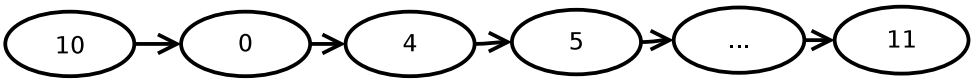

Figure 7.2: When a pointer is set from one position to the `next` *using the LF function, a chain is formed where the* `head` *of the chain is the first node. A substring is produced when the characters indicated by these nodes are output in x^r.*

In the example, the `head` of the chain is the first node and has the recorded `starting` point of 10. This in turn is chained to the `next` of 0 and that, in turn is chained to another node whose `next` is 4 and so on. This chain can be implemented by setting a pointer to 10 and then moving to the `next` position computed by the LF function. This process of following the pointer is fast, provided that L and x^r fit in RAM. It is slow in the case that they do not and must reside on disk. Should L and x^r reside on the same disk, the time taken in seeking between these two data structures (files henceforth) is excessive, as a single disk pointer is shared. These two files are thus stored on two separate disks.

```
Input:    L on disk, and C in RAM.
1 : F[0, σ] ← 0
2 : starting ← position of the last symbol of the string
3 : current ← starting
4 : c ← L[current]
5 : insert c into x^r
6 : next ← C[c] + F[c]
7 : F[c] ← F[c] + 1
8 : while not all BWT inverted do
9 :       c ← L[current]
10:       if next > current then
11:             current ← current + 1
12:             F[c] ← F[c] + 1
13:       else if current < next then
14:             current ← start of the file
15:             reset F
16:       else
17:             append c into x^r
18:             next ← C[c] + F[c]
19:             F[c] ← F[c] + 1
Output:   x^r on a second disk.
```

Figure 7.3: Pseudocode to invert BWT on two disks. F *stores the ranks of characters in* L *and is reused as long as* `next` *is to the right of* `current`*. These ranks are thrown when* `next` *is to the left of* `current`*.* F *is then reset and the scan begins from the start of the file.*

Pseudocode for inverting the BWT using two disks is presented in Figure 7.3. In order to ensure that as many ranks as possible are processed during the left-to-right scan over L, the frequency of each character is recorded in an array called F (see Steps 7, 12 and 19) so that the ranks can be reused during the move as long as the `next` positions are to the right of the `current` position. However, should a `next` fall to the left of the `current` position, the ranks in F are reset and built from the start of the file (see Steps 13 to 15).

In addition to the disk pointer, a disk has a cache [Haas, 2012] that acts as an intermediate storage between the RAM memory and disk. Should a read be required from a file, a seek is made into the file and that portion of data is stored in the disk cache. Access time is reduced if the same data is required again, provided it exists in cache. This scenario is called a *cache hit*. The probability of cache hits occurring is high if left-to-right accesses are made to the file. Therefore, an external memory algorithm will benefit in runtime if it accesses a file in a left-to-right manner.

A *cache miss* is the opposite of cache hit. That is, the requested data does not exist in cache and so, it must be buffered. The cache is emptied in the event that there is no

space for the buffered data. Similar operations occur when writing data into a file. These operations are controlled by the operating system. In summary, when a disk seek is made, it is not necessarily the case that a seek is made to the file on disk, as the data might have been accessed from the cache.

7.2 An implementation of Ferragina et al. [2010]

With the Basic BWT algorithm, as long as `next` is to the right of the `current` position, as shown in Figure 7.3, we continue moving right while keeping and reusing F. However, as soon as `next` is to the left of `current`, the left move erases F and the scan begins at the start of the file. Ferragina, Gagie and Manzini recommend only moving right by keeping track of several `starting` points simultaneously, where each produces a separate chain. Each chain in turn produces a separate substring of the original text. These chains become adjacent to each other and overlap as inversion progresses. Sufficient information to manage these chains is stored in a data structure called *header*.

Header

The positions of the first m symbols in the BWT are used as the `starting` points. The BWT characters at the `starting` points are copied into a buffer before writing them into x^r. At the same time, the algorithm records the length and location of each substring to `len` and `loc` respectively. As x^r and L reside on the same disk, the buffer prevents seeking between the two files. The positions of the BWT characters are marked simultaneously in a bit vector called `USED` that resides on the second disk. The `next`s are then computed using the LF function before sorting them numerically. The sorted `next` values allow the respective BWT symbols to be assigned to the `char`s in a single left-to-right pass over L. In parallel, the `next` positions are marked in `USED`. These positions are marked in advance to avoid accessing the disk when the `char`s are being output to x^r. Illustrated below is the header using the BWT of input string x = flooorrreeeencccee$.

header

`starting`	2	3	0	1
`len`	1	1	1	1
`loc`	2	3	0	1
`next`	1	2	4	12
`char`	n	c	e	e

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	e	n	c	c	e	c	r	e	e	e	$	f	e	l	o	o	r	r	o
`USED`	1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
x^r	e|	n|	c|	c|															

In the above example, the `starting` points are positions of the first four symbols in the BWT, that is, positions 0,1,2 and 3. The symbols at these positions are output into x^r indicated by `loc`. x^r is empty as inversion is yet to begin, and so the first symbol 'e' is written at the start of the file, that is, position 0. Likewise, the second symbol 'n' is written to the consecutive position, which is 1 in our case and so on. For clarity, the substrings are separated here with '|'. The `len` is set to 1 as each substring consists of a single character. The `next` values (4,12,1,2) are then sorted to 1,2,4,12, so that the BWT symbols in `next`s are collected in a single left-to-right pass. In parallel, the unmarked positions of the `next` values, 4 and 12, are marked in `USED`.

Chains

The `starting` point for the Basic BWT algorithm is the recorded position of the last symbol of the input string. This produces a single chain. In contrast, Ferragina et. al's algorithm has several `starting` points that produce several chains, and these chains will overlap at some point in time. Overlapping chains can be identified by recursively binary searching each `starting` point in the `next` values. If the `starting` point cannot be found, then that is the `head` of the chain and is marked as shown below.

header				
`starting`	2	3	0	1
`len`	1	1	1	1
`loc`	2	3	0	1
`next`	1	2	4	12
`char`	n	c	e	e
`head`	0	1	1	0

Continuing with the example, `starting` point of 3 is not found in the `next` values and so, is the `head` of a chain. A mark is thus placed in its `head` field. Recall this chain can be represented using linked lists if a pointer is set from its `starting` point to its `next`, as shown in Figure 7.4. Therefore, the `head` of each chain, implied by a mark, must be identified to ensure substrings are chained correctly.

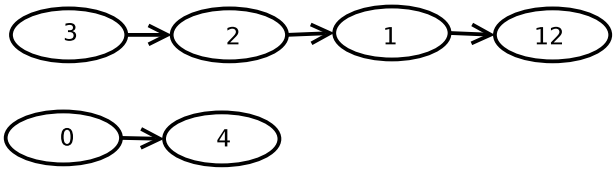

Figure 7.4: When a pointer is set from each `starting` *point (that has a mark in the* `head` *field) to its* `next`*, two chains are formed. The* `head` *of each chain is the first node.*

Once the `head` of chains have been identified, a new copy of substrings is made. The copy must begin with the `starting` point marked in `head` as it represents the first node of a chain that corresponds to the first character of a substring. There are several options for writing these substrings. The options are as follows.

a. **Storage in a single file**

Substrings are stored in one file. Pairs of `loc` and `len` are created to manage the scattered substrings. Otherwise, nm space would be required as n positions are reserved for each substring should the chains become adjacent (as the length of each substring is not known in advance). The disk pointer needs to skip n positions for each substring in order to store the `next` decoded character. This increases the access time and could be reduced by using a fast disk such as Solid State Disk; however, our intention is to make our implementation disk independent.

b. **Storage in multiple files**

Storing each substring in an individual file removes the need of having `loc` and `len`. There are m files as there are m substrings for m `starting` points. In theory, an operating system can open an infinite number of files, but in practice, this number is limited by the available RAM to handle and manage each file. Should there be sufficient memory to handle these files, seeks are still unavoidable between these files, as they are stored on the same (or several) disks. It is not viable to store m files on m disks as m can be very large.

c. **Storage in a second file**

Substrings are copied from the source file (that resides on one disk) to the destination file on a second disk. This method is used in our implementation of the algorithm. The `next` values are recursively binary searched in `starting` points while copying the substrings from source x^r (indicated by `loc` and `len`) to destination y^r. Should a `next` value not be found, the `char` is added to its substring (on disk, as described in the original Ferragina, Gagie and Manzini algorithm) in y^r without the need for the `USED` bit vector, as the failed

search implies that the subsequent character (of the input string) has not been previously inverted. Should a chain become adjacent to or overlap another chain, an empty header in created for the corresponding `starting` point as we illustrate below.

	header			
`starting`	0	1	2	3
`len`	1	1	1	1
`loc`	0	1	2	3
`next`	4	12	1	2
`char`	e	e	n	c
`head`	1	0	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	e	n	c	c	e	c	r	e	e	e	$	f	e	l	o	o	r	r	o
`USED`	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0
x^r	e|	n|	c|	c|															

In the above example, `starting` point of the first header is a `head` of a chain. So, the single character substring "e" is copied from position 0 (indicated by `loc`) in source x^r, to the beginning of destination y^r. Its `loc` is updated to the new position in destination, which is also 0. Its `next` of 4 is then binary searched in the `starting` points, and as it is not found (a failed search), we know that the subsequent character, 'e' (indicated by `char`) has not been previously inverted. It is thus appended to y^r. `len` is then updated to 2 as the present substring consist of two characters, that is, "ee" as shown below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
y^r	e	e|																	

Once the substrings are copied into y^r and if there is an empty header, it is replaced with the next unmarked position in the BWT, to ensure m characters are inverted in each round. `USED` is scanned for the first unmarked position and the BWT symbol is copied into a buffer. The buffer prevents seeking between two files that reside on the same disk. It is then copied into y^r as shown below.

	header			
`starting`	0	5	6	3
`len`	2	1	1	4
`loc`	0	6	7	2
`next`	4			12
`char`				
`head`	1	0	0	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	e	n	c	c	e	c	r	e	e	e	$	f	e	l	o	o	r	r	o
`USED`	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0
y^r	e	e|	c	c	n	e|	c|	r|										

The overlapped substrings have created space for two inversion points, that is, two empty headers. The next unmarked position in `USED` is position 5 and is assigned to the second `starting` point, followed by position 6 that is assigned to the third `starting` point. At the same time, the corresponding BWT symbols are output into y^r while recording the respective length and location. A round of inversion is completed when $|x^r|+m$ characters are written into y^r. This process is repeated until the entire BWT is inverted.

7.2.1 Implementation details

Our implementation of Ferragina et al.'s algorithm differs from that described in the original paper [Ferragina et al., 2010]. The differences are as follows.

a. **Header**

In the original paper, a header consists of the following fields: `starting` point, `next` and `char`. Each substring is prefaced by this header and is stored in a file called S that resides on disk. In contrast, the header in our implementation has three additional fields: `len`, `loc` and `head`, and resides in RAM. Only the substrings reside on disk.

We explored the possibility of storing the headers on disk via the use of an efficient external memory library, that is the Standard Template Library for XXL (STXXL) [Dementiev et al., 2005]. However, this variant was very slow due to the time taken to search the `starting` point in `next`s (and vice versa) on disk. As a result, we decided to store the

headers in RAM to have a better runtime where the number of headers, m is restricted by the available RAM in contrast to the original paper that sets m to $n/\log n$.

We note that our implementation of Ferragina et al.'s algorithm can easily be modified to use the STXXL should there be insufficient RAM.

b. **Chains**

In the original paper, the headers are extracted from S, which resides on one disk, and written to another file, called S′, which resides on a second disk. The list ranking algorithm of Chiang et al. [1995] is then applied (after each round) to the `starting` points and `next`s in S′ to find the chains that will become adjacent to each other and overlap. Should overlap occur, the merge order is stored in the `char` field. These headers are then reinserted into S while merging adjacent substrings into a single substring. Then, the constituent substrings are deleted using the merge order stored in `char`s.

The `char`s are then reused to hold new BWT characters. Once the `char`s are assigned, the headers are reinserted into S. The `char` is only appended to its substring if the `next` is unmarked in the bit vector (indicating that the character has not yet been inverted). The bit vector is accessed m times as there are m `char`s. This process is repeated until the entire BWT is inverted.

There is an open question as to how the substrings are stored and merged in S. Several possibilities of storing the substrings were explored, and we chose to alternate substrings between two files. That is, we placed source x^r on one disk and destination y^r on the second disk. `loc` and `len` were then introduced to store the location and the length of each substring in the file. When the BWT characters are assigned to `char`s, the bit vector is marked in parallel even though these characters have not been added to their substrings. This effect can be simulated during inversion without even accessing the said bit vector. A failed search of the `next` in `starting` points implies that the BWT character has not been previously inverted.

We also replaced the list ranking algorithm with binary search as it gave us similar asymptotic complexity. The `head` field was introduced so that the head of each chain can be marked (identified).

c. **Others**

Sorting `next`s enables one to collect the `char`s in a single left-to-right pass over the BWT file (as described in the original paper). However, random accesses are made to the bit

vector as it is checked each time the `char` is appended to see if it has been inverted before. Thus, we explored the possibility of replicating the bit information to the header. However, we later found that replicating the bit information was not required as a failed search would imply that the character has not been previously inverted.

We also explored the possibility of interleaving the BWT and the `USED` bit vector in the same file that resides on a disk. Each bit used 1 byte as there is no bit data type. To cache as many BWT characters as possible in the disk cache in a single pass over the BWT, we chose to store the BWT and `USED` in two separate files on two separate disks.

Note that a journal version of their paper was recently published in [Ferragina et al., 2012]. The implementation in this chapter was based on the earlier conference paper.

7.3 Variant of Ferragina et al. [2010] Algorithm

We explored a variant, Algorithm `FGM-V` where the `starting` points begin inside the longest character run of the BWT string, followed by the second longest run and so on, until there are m inversion points. The `starting` points from a run are in increasing order (with the values only differing by one), and so are the values in `next`s. These `next`s might land in the same character run or even split across two or three runs. In either case, should we begin in a run, there is a great possibility that the `next`s remain clumped together as shown below.

header

`starting`	2	7	8	9
`len`	1	1	1	1
`loc`	3	0	1	2
`next`	1	6	7	8
`char`	n	r	e	e

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	e	n	c	c	e	c	r	e	e	e	$	f	e	l	o	o	r	r	o
`USED`	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
x^r	e|	e|	e|	c|															

In this example, the last three `starting` points (7,8,9) are from the longest character run, that is run *e*, and the first `starting` point (2) is from the second longest character run, namely run *c*. Note that the values in the `starting` points are in increasing order

(only differing by one) and so is the order of the `next`s. In other words, the `next` values are clumped together. A single seek is thus sufficient to process these `next`s.

Negligibly small arrays are used to manage these character runs in constant time. These runs are used during the initialization and replacement stage of empty headers. Should the longest runs become exhausted, the algorithm retreats to the original approach of choosing the `starting` points, that is, the positions of the first m unmarked symbols in the BWT.

7.4 New Inversion Algorithms

Our new inversion algorithms use scanning and simple compression methods to invert the BWT. Like the Ferragina, Gagie and Manzini algorithm, our new algorithm inverts the BWT from multiple `starting` points. However, unlike Ferragina, Gagie and Manzini we assume the `starting` points are evenly spread in the input file. Thus, a `starting` point is chosen at every interval p of the input string. The position of the first character (of each substring) in the input string is available together with the `starting` point in a file called `START`. We note that no additional time is required in computing these positions if they are recorded when the BWT is produced as we illustrate below using the BWT of input string $x =$ flooorrreeeencccee\$.

$$\texttt{START} \rightarrow \langle 1, 12\rangle \ , \langle 10, 18\rangle \ , \langle 16, 6\rangle$$

In the above example, assume p is 6. Each angle bracket shows the position of the `starting` point in the BWT and in the input string. For example, L[1] = x[12] = n. That is, the character n has the index of 1 in L and of 12 in x. It is trivial to modify any BWT algorithm to record these multiple `starting` points.

Header

The `starting` points and the respective positions in the input string are read from `START` into the `next` and `loc` arrays (which we define later for clarity). The `next` values are then inserted into a heap, where each `next` is linked to its respective `loc` via the index (of the `loc`), called `i`. Thus, each entry in the heap consists of `next` and `i`. A min-heap based on the `next` values is created as shown below, where the first entry of the heap is the root.

min-heap			
`next`	1	10	16
`i`	1	0	2

```
Input:      L.
1 : create min-heap of next values
2 : while min-heap is not empty do
3 :       current ← next[0] (root of the heap)
4 :       output L[current] into the respective substrings
5 :       use LF function to compute next
6 :       if next is on the right of current then
7 :             add to min-heap
8 :       else
9 :             add to temporary list
10:copy temporary list into min-heap
Output:     x^r.
```

Figure 7.5: Pseudocode of our inversion algorithms that uses a min-heap and a list to ensure more than m characters are inverted in each round.

Pseudocode for inverting the BWT using a min-heap is presented in Figure 7.5. The character indicated by the `next` value at root of the heap is output, and the LF function is used to compute the new `next` for that inversion point. Should the new `next` be to the right of the root, the value is inserted into the heap. Otherwise, it is stored in a temporary list. The items in the list are then emptied into the heap once all the characters indicated by the `next` values have been output into their corresponding substrings (see Steps 1 to 10).

Unlike Ferragina et. al's algorithm that inverts m BWT characters in each round, our method of using a min-heap and temporary list guarantees *at least* m characters are inverted in each round, although there are m `starting` points.

Chains

The substrings in our new algorithms never overlap, as their positions in the input string (computed when the BWT was produced) have been recorded in the `loc` array. No memory is allocated for these substrings during the initialization stage. If there is a character waiting to be output, 1 byte of RAM is allocated for the respective substring. This amount is doubled whenever the substring runs out of space to append the output character. We also ensure the amount of the new memory allocated never exceeds the number of remaining uninverted characters. We note that the initial growing size of each substring can be set to any size, within the limits of available RAM.

The memory blocks for these substrings are managed using `cap` (which stores the amount of allocated RAM for each substring) and `len` (which stores the current length of each

substring) as shown below.

`i`	0	1	2
`loc`	0	6	12
`substr`	⊥	⊥	⊥
`len`	0	0	0
`cap`	0	0	0

The length and capacity of each substring is set to 0, and ⊥ is indicated for the (empty) substrings as the decoding is yet to begin. The index, `i` links the corresponding arrays to the respective entries in the min-heap. For example, the second column of the arrays stores the substring's information for the root as they share a similar `i`, which is 1 in this case. The `len` values avoid use of a bit vector to track inverted characters. When the sum of lengths of all the substrings equals n, we know the entire BWT has been inverted.

If at any point the substrings exhaust the available RAM memory, the inversion is paused and the substrings are written into x^r on positions indicated by `loc` values. The memory allocated for these substrings is released. Then, inversion resumes as illustrated below.

`i`	0	1	2
`loc`	0	6	12
`substr`	$e	ne	rro
`len`	2	2	3
`cap`	2	2	4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
x^r	$	e					n	e					r	r	o				

In the above example, the first substring ($e) consist of two characters (indicated by `len`) and 2 bytes have been allocated for it (indicated by `cap`). The BWT character in L[`current`] cannot be outputted as the substrings have exhausted the allocated RAM, that is 8 bytes. The inversion thus comes to a halt and the substrings are written into x^r at positions indicated by `loc`s, in a single left-to-right pass. For instance, the first substring is written at position 0 as `loc` is 0. This `loc` is then updated to 2 so that the substring is written from position 2 onwards for the next write. Note that each substring requires another four characters for the inversion to be completed. So, when the memory is released for the substrings, `cap` is set to 0, but not `len` as it keeps track on the number of inverted characters as shown below.

i	0	1	2
loc	2	8	15
substr	$\perp$	$\perp$	$\perp$
len	2	2	3
cap	0	0	0

Releasing this memory is important as each substring grows at a different rate. This process is repeated until the entire BWT is inverted. This algorithm, which we call `SCAN` is the basis of several variants that are described in the following sections.

7.4.1 Scan based approach

We explored a variant, Algorithm `SCAN-PR` that stored partial information about the ranks in a two dimensional R array [Lauther and Lukovszki, 2005; Kärkkäinen and Puglisi, 2010]. The idea is that storing this information will improve the speed of the basic algorithm as time is not wasted recounting the BWT symbols during each round of inversion.

More precisely, for each symbol we store rank values at every d position in the BWT string. That is, there is a reference point at positions $0, d, 2d, \ldots$. These reference points divide the BWT into $b = n/d$ blocks where each reference point correlates to the beginning of a block. Each position j in the block is associated to a reference point ref(j) and for all $c \in \Sigma$

$$\text{rank}(c, \text{L}, j) = \text{rank}(c, \text{L}, \text{ref}(j)) + \text{F}[c]$$

and

$$\text{rank}(c, \text{L}, \text{ref}(j)) = \text{R}[\text{ref}(j)][c],$$

as shown below. This method counts the symbols between the reference point and the current position j, and R is stored in memory.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	(e	n	c	c	e	c	r)	(e	e	e	$	f	e	l)	(o	o	r	r	o)
rank(c, L, ref(j))	0	0	0	1	1	2	0	0	1	2	0	0	3	0	0	1	0	1	2
ref(j)				0							1						2		

In this example, assume d is 7. The reference points at positions 0, 7 and 14 correlate to the start of the respective blocks. The boundaries of the block are indicated by the brackets.

For clarity, the associated block for each position j is shown in ref(j). The partial ranks are stored in the R array as illustrated below.

ref(j)	0	1	2
\$	0	0	1
c	0	3	3
e	0	2	6
f	0	0	1
l	0	0	1
n	0	1	1
o	0	0	0
r	0	1	1

The second column indicates the ranks for the first reference point, that is position 0. Likewise, the third column indicates the ranks for the second reference point, that is position 7. For instance, to compute rank(e, L, 12), we must find the block associated with position 12. It is the second block as ref(12) is 1. This index is then used to access the partial ranks in R in constant time. R[1][e] is 2 and is added to the occurrences of character 'e' in prefix L[7, 12], which is 3 here. So, rank(e, L, ref(12)) is 5.

Unlike the previous variant, that always uses the reference point at the start of the block, our second variant, Algorithm `SCAN-NR` uses the nearest reference point approach [Lauther and Lukovszki, 2005; Kärkkäinen and Puglisi, 2010]. That is, the reference point at the end of block is sometimes closer than the reference point at the start of the block. Should a position j fall in the first half of the block,

$$\text{rank}(c, \text{L}, j) = \text{rank}(c, \text{L}, \text{ref}(j)) + \text{F}[c]$$

and should it fall in the second half of the block,

$$\text{rank}(c, \text{L}, j) = \text{rank}(c, \text{L}, \text{ref}(j)) + \text{F}[c] - 1.$$

The partial ranks are computed and stored in a file. These ranks are interleaved with the BWT characters. Each BWT character and rank pair uses 3 bytes (1 byte for the character, and the other 2 bytes for the partial rank). The maximum value that can be stored for this partial rank is thus 65536.

7.4.2 Block based approach

We explored a simple variant of `SCAN-PR`, Algorithm `BLOCK-PR` that prioritizes each block based on the number of `next` values it contains. That is, the block processed always has the most number of `next` values. A max-heap keyed on the number of `next` values in each block is created. Each entry in the heap contains the unique ref(j) value (to differentiate blocks from each other) and a separate heap (to store the `next` values contained in the block).

The `next` values are processed with slight modifications to the pseudocode in Figure 7.5. That is, a `next` is only stored in the temporary list should it belong to the existing block in the RAM. Otherwise, the `next` is stored in its respective min-heap. These individual heaps are accessed in constant time, by recording the position of each block (after the heapifying process) in an array called I. The existing block (in RAM) is reprocessed only while the number of `next` values exceed a threshold. This is to take advantage of the block already residing in RAM in contrast to having to make a seek to access a new block on disk.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L	(e	n	c	c	e	c	r	e	e	e	$	f	e	l)	(o	o	r	r	o)
rank(c, L, ref(j))	0	0	0	1	1	2	0	0	1	2	0	0	3	0	0	1	0	1	2
ref(j)							0										1		

	max-heap	
min-heap	{0,10},{1,1}	{2,16}
ref(j)	0	1

In the above example, assume d is 14. The reference points at positions 0 and 14 correlate to the start of the respective blocks. Similar to the previous variants, the associated block for each position j is indicated in ref(j), which is used to access the partial ranks in the R array. The max-heap consist of two entries. The first entry which is the root contains the inversion points for the first block, that is 2 here. Likewise, the second entry contains a single inversion point for the second block. Therefore, the first block is brought into RAM to be processed as it has the most number of inversion points in contrast to the other block.

7.4.3 Compression approach

Our final algorithm, Algorithm `COMP-PR` is a modification of `BLOCK-PR` that boosts the effective block size using Run Length Encoding (RLE) [Salomon and Motta, 2010]. As the name

suggests, in this encoding (compression) technique, each character run (consecutive repeated symbols) is replaced with the character and the run length, as shown below.

aaabbcd → a3b2c1d1

In this example, the encoded string has 4 runs. This type of compression is efficient on BWT data if the encoded string contains repetitions. Encoding can be effective for this algorithm for two main reasons: more characters are brought into RAM, and time is not spent counting occurrences of each character (from the respective reference point onwards) as this value has been indicated in the run length. Moreover, strings that compress well with the BWT are known to contain many runs [Manzini, 2001b].

The encoding is done in linear time using two disks. The BWT is stored on one disk and the encoded BWT on the other disk. We fix each (BWT character , run length) pair in the encoded BWT to use 2 bytes: 1 byte for the BWT character, and the other byte for the run length. The maximum value that can be stored for length is thus 255. However, by considering the existence of the BWT character, each encoded pair can be represented by the run length of 256 as shown below. The encoded BWT is stored in a file called `RLE`.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
`RLE`	(e	0	n	0	c	1)	(e	0	c	0	r	0)	(e	2	$	0	f	0)	(e	0	l	0	o	1)	(r	1	o	0)
ref(j)			0						1						2						3					4		

In the above encoded BWT string, the length for single character runs is indicated with a 0. We note that the 0 can be removed but this is not done here due to the intensive checking required during inversion. The third pair (c 1) indicates that the character 'c' has a run length of 2.

Unlike the previous variants that store partial information about the ranks at every d position in the BWT string, this variant stores them at every r run, as the number of characters varies in each run. In the example, assume r is 3. That is, each block consist of 3 runs. In other words, we divide the string into $b = n/r$ variable size blocks. The total number of characters in each block is stored in an array called S. This array is then binary searched to find the block associated to the `next` so that the partial ranks in R can be accessed in constant time, as shown below.

ref(j)	0	1	2	3	4	5
S	0	4	7	12	16	19

Continuing with the same example, the second column in S represents the number of characters prior to the first reference point, which is 0. The number of characters prior to the second reference point is indicated in the third column. For instance, to find rank(e, L, 12), we must find the block associated with position 12. This can easily be obtained by binary searching S for 12 and if it does not exist, the first value that is smaller than 12 is searched. In our case, it is 3 as ref(12) = 3. This index is then used to access R in constant time.

We also explored a simple variant to `COMP-PR`, that is `COMP-SCAN-PR`. This variant scans each block sequentially without giving any priority to the the block that contains the most number of `next` values. The purpose of implementing this variant is to take advantage of compression and cache hits due to the left-to-right accesses over the smaller encoded BWT.

7.5 Experiments

In this section, we report the practical performance of the algorithms on real data. The experiments aim to measure the effect on performance of the number of `starting` points m (chosen evenly at every $p = n/m$ position in the new inversion algorithms), and reference points (chosen at every d character in the BWT and every r run in the encoded BWT).

7.5.1 Data and Setup

For testing, we used the files listed in Table 7.1. These files are typical datasets from a number of different public domains.[1][2][3] `DNA` is from the *Saccharomyces Genome Resequencing Project*[4] that provides 36 sequences of *Saccharomyces Paradoxus* and 37 sequences of *Saccharomyces Cerevisiae* species. `WIKI` is a crawl of English Wikipedia articles (early 2009), and `GOV2` is a crawl of HTML pages from *.gov domain* in the US (early 2004). Each of the test files is 512 MB as it is enough data to significantly exceed the available memory to test the scalability of the algorithms.

All code was written in C/C++, compiled using gcc/g++ version 4.4.5 and the -O3 optimization flag. Experiments were run on an otherwise idle 2.80 GHz Intel® Pentium® 4 of 371 MB of RAM, 512 KB of cache and two Seagate Barracuda SATA II 1TB 7200 RPM disks. The operating system is Red Hat Enterprise Linux Server release 6.1 running Kernel 2.6.32-131.2.1.el6.i686. Times reported are the minimum of three runs, measured with the C

[1] http://pizzachili.dcc.uchile.cl/repcorpus.html
[2] http://boston.lti.cs.cmu.edu/Data/clueweb09/
[3] Text REtrieval Conference (TREC)
[4] http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp

Table 7.1: All test files are 512 MB, and terminated with the special end of string character.

Dataset	Mean LCP	Max LCP	σ	No. of runs
DNA	3,044	104,177	5	23,381,469
WIKI	7,495	556,673	128	41,336,845
GOV2	96	94,066	195	149,228,428

time function. Our inversion algorithms make many scans of the BWT and try to work on disk. There is a possibility that some of the BWT characters are cached in memory. Thus, memory was flushed after each run. Note that this was not done in the previous chapters as we explicitly held the text in RAM, and streamed the SA to disk for suffix sorting. We further eliminate the caching effect as much as possible by ensuring the I/O files (BWT and inverted BWT) are larger than the available memory. The I/O time for reading and writing to disk is also included. Peak memory usage is the sum of data structures reported by the C sizeof function.

7.5.2 Results and Analysis

The effect of parameters p, d and r on the performance is given in Tables 7.3, 7.4 and 7.5. The size of the substrings is set to 64 MB. Upon reaching this limit, the substrings are written to disk. Table 7.6 summarizes the results between the new inversion algorithms and Ferragina et. al algorithms.

In general, the reported times are not always relative to the space used as they are influenced by various factors. These factors include the number of headers that require sorting, random accesses made to files, the use of fast compressed data structures and others. A large number of headers does not indicate the inversion speed.

`FGM` and `FGM-V` require two sorts. The headers are first sorted based on the `next`s to find the `head` of the chains, before resorting them based on the `starting` points to find the `next`s. The second sort can be removed, provided the indexes of the `next`s are stored in the headers during the first sort. We explored this approach (removal of the second sort) but the reported times were almost twice of `FGM` in Table 7.2, as fewer characters were being inverted in each round since $4m$ bytes were being used to store each index.

The likelihood of random accesses occurring in `FGM` algorithm is reduced in `FGM-V` as the `starting` points are chosen from the longest character runs so that the inversion points are clumped together in the subsequent round. This increases the possibility of the substrings

Table 7.2: The time measured in minutes, the peak memory in MB (in brakets).

Dataset	m	FGM	FGM-V
DNA	9,942,053	253 (248)	305 (248)
	5,779,785	184 (144)	190 (144)
	5,137,299	187 (128)	188 (128)
	4,971,026	189 (124)	192 (124)
	3,853,222	195 (96)	199 (96)
	3,314,017	208 (83)	211 (83)
	3,211,000	210 (80)	214 (80)
	2,889,900	220 (72)	221 (72)
	2,809,500	223 (70)	227 (70)
WIKI	9,942,053	257 (248)	301 (248)
	5,779,785	180 (144)	181 (144)
	5,137,299	181 (128)	178 (128)
	4,971,026	186 (124)	183 (124)
	3,853,222	203 (96)	189 (96)
	3,314,017	217 (83)	211 (83)
	3,211,000	218 (80)	208 (80)
	2,889,900	236 (72)	221 (72)
	2,809,500	237 (70)	225 (70)
GOV2	9,942,053	933 (248)	973 (248)
	5,779,785	535 (144)	549 (144)
	5,137,299	508 (128)	518 (128)
	4,971,026	497 (124)	506 (124)
	3,853,222	477 (96)	467 (96)
	3,314,017	459 (83)	463 (83)
	3,211,000	460 (80)	466 (80)
	2,889,900	454 (72)	463 (72)
	2,809,500	463 (70)	468 (70)

being clumped together in the files as well. However, the character runs were used up in the first round of inversion itself.

In the new inversion algorithms, competitive times are observed between the scan based algorithms and the block based algorithm. The efficiency of the block based algorithm, BLOCK-PR depends on the number of next values contained in the block. There is a tradeoff between this number and with the random access made to access the block. The number of nexts characters that are inverted may outweigh the time taken to seek and read the particular block from disk into RAM. The reference points, as expected, increased the inverting speed due to the storing of partial information about the ranks.

By far the best approaches were compression based algorithms, which outperformed all other algorithms by boosting the effective block size using RLE.

Table 7.3: The running time in minutes (in brackets, peak memory usage in MB) on various p for 512 MB prefix of DNA *dataset. d is shared between* SCAN-PR *and* SCAN*. r is shared between* COMP-PR *and* COMP-SCAN-PR*.*

p	SCAN	d	SCAN-PR	BLOCK-PR	SCAN-NR	r	COMP-PR	COMP-SCAN-PR
256	47 (128)	$n/4$	48 (128)	45 (240)	76 (128)	$n/32$	19 (144)	18 (144)
		$n/16$	56 (128)	40 (144)		$n/64$	19 (120)	17 (120)
		$n/64$	60 (128)	41 (120)		$n/128$	18 (114)	17 (114)
		$n/256$	57 (128)	42 (114)		$n/256$	17 (113)	17 (113)
		$n/1024$	49 (128)	51 (113)		$n/512$	17 (112)	16 (112)
512	70 (96)	$n/4$	73 (96)	64 (216)	206 (96)	$n/32$	17 (120)	17 (120)
		$n/16$	86 (96)	60 (120)		$n/64$	17 (96)	16 (96)
		$n/64$	88 (96)	62 (96)		$n/128$	16 (90)	16 (90)
		$n/256$	85 (96)	65 (90)		$n/256$	15 (89)	15 (89)
		$n/1024$	66 (96)	76 (89)		$n/512$	15 (88)	15 (88)
1024	120 (80)	$n/4$	124 (80)	110 (204)	357 (80)	$n/32$	17 (108)	17 (108)
		$n/16$	151 (80)	97 (108)		$n/64$	16 (84)	16 (84)
		$n/64$	153 (80)	112 (84)		$n/128$	15 (78)	15 (78)
		$n/256$	131 (80)	109 (78)		$n/256$	14 (77)	15 (77)
		$n/1024$	105 (80)	121 (77)		$n/512$	14 (76)	14 (76)
2048	214 (72)	$n/4$	231 (72)	222 (198)	343 (72)	$n/32$	19 (102)	18 (102)
		$n/16$	284 (72)	144 (102)		$n/64$	18 (78)	18 (78)
		$n/64$	281 (72)	203 (78)		$n/128$	17 (72)	18 (72)
		$n/256$	237 (72)	200 (72)		$n/256$	16 (71)	17 (71)
		$n/1024$	190 (72)	213 (71)		$n/512$	15 (70)	16 (70)

7.6 Summary

Previously, there were no BWT inversion algorithms that scale to disk. This chapter has closed this gap by providing the first implementation of the only previous work due to Ferragina, Gagie and Manzini [Ferragina et al., 2010]. As there were open questions related to this algorithm, especially as to how partially decoded substrings should be stored and managed, several implementation possibilities were explored. The best approach in terms of time and space was then implemented. We also removed the frequent disk accesses to the USED bit vector for each inverted BWT character by simulating its effect during binary search. A failed searched implies that the sought character has not yet been decoded.

Several new inversion algorithms of our own that use scanning, blocking, and compression techniques were also described. A comprehensive experimental comparison on a wide range of datasets shows that our compression approach is up to 14 times faster than the implementation of Ferragina et al., and that the inversion of the BWT in external memory is practical.

Table 7.4: The running time in minutes (in brackets, peak memory usage in MB) on various p for 512 MB prefix of WIKI *dataset. d is shared between* SCAN-PR *and* SCAN*. r is shared between* COMP-PR *and* COMP-SCAN-PR*.*

p	SCAN	d	SCAN-PR	BLOCK-PR	SCAN-NR	r	COMP-PR	COMP-SCAN-PR
256	38 (128)	$n/4$	37 (128)	49 (240)	71 (130)	$n/32$	19 (144)	20 (144)
		$n/16$	38 (128)	38 (144)		$n/64$	18 (120)	18 (120)
		$n/64$	38 (128)	35 (120)		$n/128$	18 (114)	17 (114)
		$n/256$	38 (128)	35 (114)		$n/256$	17 (113)	17 (113)
		$n/1024$	39 (129)	43 (113)		$n/512$	17 (112)	17 (112)
512	53 (96)	$n/4$	53 (96)	69 (216)	187 (98)	$n/32$	17 (120)	17 (120)
		$n/16$	51 (96)	55 (120)		$n/64$	17 (96)	17 (96)
		$n/64$	50 (96)	50 (96)		$n/128$	16 (90)	16 (90)
		$n/256$	50 (96)	50 (90)		$n/256$	16 (89)	16 (89)
		$n/1024$	49 (97)	64 (89)		$n/512$	15 (88)	15 (88)
1024	86 (80)	$n/4$	83 (80)	117 (204)	307 (82)	$n/32$	19 (108)	18 (108)
		$n/16$	84 (80)	93 (108)		$n/64$	18 (84)	17 (84)
		$n/64$	81 (80)	83 (84)		$n/128$	18 (78)	17 (78)
		$n/256$	80 (80)	83 (78)		$n/256$	16 (77)	16 (77)
		$n/1024$	78 (81)	108 (77)		$n/512$	16 (76)	16 (76)
2048	146 (72)	$n/4$	146 (72)	225 (198)	276 (74)	$n/32$	23 (102)	22 (102)
		$n/16$	147 (72)	171 (102)		$n/64$	21 (78)	20 (78)
		$n/64$	143 (72)	151 (78)		$n/128$	21 (72)	20 (72)
		$n/256$	138 (72)	149 (72)		$n/256$	20 (71)	19 (71)
		$n/1024$	138 (73)	194 (71)		$n/512$	19 (70)	18 (70)

Table 7.5: The running time in minutes (in brackets, peak memory usage in MB) on various p for 512 MB prefix of GOV2 *dataset. d is shared between* SCAN-PR *and* SCAN*. r is shared between* COMP-PR *and* COMP-SCAN-PR*.*

p	SCAN	d	SCAN-PR	BLOCK-PR	SCAN-NR	r	COMP-PR	COMP-SCAN-PR
256	48 (128)	$n/4$	48 (128)	50 (240)	87 (131)	$n/32$	32 (144)	33 (144)
		$n/16$	54 (128)	44 (144)		$n/64$	27 (120)	32 (120)
		$n/64$	56 (128)	42 (120)		$n/128$	27 (114)	30 (114)
		$n/256$	54 (128)	44 (114)		$n/256$	31 (113)	28 (113)
		$n/1024$	48 (129)	53 (113)		$n/1024$	53 (114)	26 (114)
512	68 (96)	$n/4$	70 (96)	81 (216)	183 (99)	$n/32$	33 (120)	43 (120)
		$n/16$	82 (96)	65 (120)		$n/64$	29 (96)	42 (96)
		$n/64$	81 (96)	63 (96)		$n/128$	32 (90)	40 (90)
		$n/256$	71 (96)	66 (90)		$n/256$	33 (89)	37 (89)
		$n/1024$	60 (97)	81 (89)		$n/1024$	50 (90)	32 (90)
1024	115 (80)	$n/4$	118 (80)	144 (204)	324 (83)	$n/32$	43 (108)	69 (108)
		$n/16$	128 (80)	110 (108)		$n/64$	37 (84)	67 (84)
		$n/64$	116 (80)	110 (84)		$n/128$	36 (78)	63 (78)
		$n/256$	100 (80)	114 (78)		$n/256$	39 (77)	58 (77)
		$n/1024$	87 (81)	136 (77)		$n/1024$	48 (78)	49 (78)
2048	194 (72)	$n/4$	202 (72)	227 (198)	284 (75)	$n/32$	67 (102)	118 (102)
		$n/16$	212 (72)	195 (102)		$n/64$	60 (78)	113 (78)
		$n/64$	185 (72)	200 (78)		$n/128$	64 (72)	108 (72)
		$n/256$	158 (72)	203 (72)		$n/256$	58 (71)	99 (71)
		$n/1024$	144 (73)	245 (71)		$n/1024$	68 (72)	83 (72)

Table 7.6: The running time in minutes (in brackets, peak memory usage in MB) on 512 MB prefix of files.

Dataset	SCAN	SCAN-PR	BLOCK-PR	SCAN-NR	COMP-PR	COMP-SCAN-PR	FGM	FGM-V
DNA	47 (128)	48 (128)	41 (120)	76 (128)	19 (120)	17 (120)	187 (128)	188 (128)
	70 (96)	73 (96)	62 (96)	206 (96)	17 (96)	16 (96)	195 (96)	199 (96)
	120 (80)	105 (80)	109 (78)	357 (80)	15 (78)	15 (78)	210 (80)	214 (80)
	214 (72)	190 (72)	200 (72)	343 (72)	17 (72)	18 (72)	220 (72)	221 (72)
WIKI	38 (128)	37 (128)	35 (120)	71 (130)	18 (120)	18 (120)	181 (128)	178 (128)
	53 (96)	50 (96)	50 (96)	187 (98)	17 (96)	17 (96)	203 (96)	189 (96)
	86 (80)	80 (80)	83 (78)	307 (82)	18 (78)	17 (78)	218 (80)	208 (80)
	146 (72)	138 (72)	149 (72)	276 (74)	21 (72)	20 (72)	236 (72)	221 (72)
GOV2	48 (128)	48 (128)	42 (120)	87 (131)	27 (120)	32 (120)	508 (128)	518 (128)
	68 (96)	70 (96)	63 (96)	183 (99)	29 (96)	42 (96)	477 (96)	467 (96)
	115 (80)	100 (80)	114 (78)	324 (83)	36 (78)	49 (78)	460 (80)	466 (80)
	194 (72)	158 (72)	203 (72)	284 (75)	64 (72)	83 (72)	454 (72)	463 (72)

Chapter 8

Conclusions and Future Work

This thesis has presented several improved methods for processing strings in restricted memory settings. In this chapter, we reaffirm our research questions, summarize the main results, and outline some directions for future work.

8.1 Conclusions

The current *suffix array construction algorithms (SACAs)* lie at either extreme of the efficiency spectrum: they are either fast and use at least $5n$ of space, or they trade runtime by using slow compressed data structures or disk or some combination techniques that use less memory. A space-efficient, semi-external suffix sorting algorithm by Kärkkäinen [2007] lies in the middle of the spectrum. Kärkkäinen builds the *suffix array (SA)* in blocks. He builds and processes each block separately; first the leftmost block is collected, sorted and written to disk to form a contiguous section of SA. The memory is then reused to process the next block. For each block that is processed, a pass is made over the text. The number of passes made over the text is proportional to the sorting time which is the bottleneck of this algorithm. In Chapter 5, we aimed to reduce the sorting time of the algorithm. In particular, we aimed to address the following research question:

- *Can we implicitly sort the suffixes in a manner that will reduce the number of passes to be made over text without increasing the memory requirement of the Kärkkäinen's suffix sorting algorithm?*

The main contribution of this chapter is a method for implementing the pointer copying heuristics from internal memory suffix array construction in a semi-external setting besides

using algorithmic optimization techniques. We achieve a speed up of 2-4 times without increasing memory usage of the algorithm. We are up to twice as fast as the next fastest algorithm by Ferragina, Gagie and Manzini [Ferragina et al., 2012] when working memory is equated. We then set out to answer the following research question:

- *Using the new algorithm, can we suffix sort strings with large alphabets?*

We are 2-3 times slower than Larsson and Sadakane [2007] (the best published algorithm for strings with large alphabets), but we use less memory. This new algorithm is thus of use when memory is particularly tight.

Often used with the SA, is the *longest-common-prefix (LCP) array*. When enhanced with the LCP array, the SA can provide efficient solutions to many string processing applications including a problem called *pattern mining*. The existing mining algorithms lie at either extreme of the efficiency spectrum: they are either fast and use enormous amounts of space, or they are compact and orders of magnitude slower. In Chapter 4, we aimed to address the following research question:

- *Can we solve mining problems for string data closer time to the fastest published algorithm and use the same amount of space or less than the most space efficient algorithm?*

We presented an algorithm for mining substrings from a database of strings that is, in practice, as fast as the fastest existing approaches, and uses less memory that the most space efficient approaches. Our main mechanism for keeping memory usage low is to build the enhanced SA incrementally, in blocks. Once built, a block is traversed to output patterns with required support before its space is reclaimed to be used for the next block. Asymptotically, we require $O(n \log \sigma)$ bits of memory and $O(n \log^2 n)$ space to mine a database of total length n symbols drawn from an alphabet of σ possible symbols.

Using our approach one can find all frequent patterns in the human genome with a given support in under 6 hours using 9 GB of RAM on a standard work station. It should be noted that our algorithm is capable of handling all string mining problems where the reporting (or not) of a pattern is based on the frequency of the pattern in the database, similar to [Fischer et al., 2006; 2008; Kügel and Ohlebusch, 2008; Weese and Schulz, 2008]. We have considered frequent patterns and emerging patterns here, but we can also restrict patterns to, for example, pass the χ^2-test.

Pointer copying heuristics have been used to speed SACAs, but not LCP construction algorithms. We then set out to ascertain the following research question:

- *Can we reduce the construction time of the LCP array via pointer copying?*

In Chapter 6, we investigated new approaches to combine the pointer copying heuristics to an efficient LCP construction algorithm due to Kärkkäinen, Manzini and Puglisi [Kärkkäinen et al., 2009]. Improvements in construction time are observed.

The *Burrows-Wheeler transform (BWT)* was discovered independently of the SA, but it is now known that the two data structures are essentially equivalent. In the last 20 years or so, hundreds of papers have been published on SACAs, forward BWT algorithms and their variants, but very few papers on inverting the BWT. For very large collections, there may not be enough RAM memory to invert the BWT using traditional approaches [Kärkkäinen and Puglisi, 2010]. There are forward BWT algorithms that work on disk but none for inversion. Only a theoretical proposal due to Ferragina, Gagie and Manzini [Ferragina et al., 2010] exists. We then set out to establish the following research question:

- *Can we provide an implementation of the only theoretical proposal of Ferragina, Gagie and Manzini's algorithm?*

Chapter 7 has closed the gap between the theory and practice by examining the problem of inverting BWT efficiently on disk. Several possibilities of implementing the only theoretical proposal due to Ferragina, Gagie and Manzini [Ferragina et al., 2010] were explored. The best approach in terms of time and space was implemented. To this end, we aimed to address the following research question:

- *Their algorithm is complex, so can we discover simple, more practical inversion algorithms for disk?*

Some of our new inversion techniques include scanning, prioritizing the block based on ranks, and boosting the effective block size with Run Length Encoding. These techniques are further aided with previous and nearest reference points that store precomputed ranks. We have shown that our best approach is up to 14 times faster than the implementation of Ferragina et al., and that inversion of the BWT in external memory is practical.

8.2 Future Work

We now outline some directions that could benefit from future research.

8.2.1 Suffix Sorting

Some potential areas for future research in suffix sorting are as follows.

Suffix selection. A problem where one seeks the suffix of a given rank, without resorting to sorting all suffixes, has received some attention recently [Franceschini and Muthukrishnan, 2007a;b; Franceschini et al., 2009; Franceschini and Hagerup, 2011]. Efficient suffix selection algorithms can aid suffix sorting algorithms. For example, they could be used to choose good splitters in Kärkkäinen's algorithm. This removes the situation of underfull/overfull blocks as each pass will always collect $b_{\max}$ number of suffixes. Recall the algorithm currently falls back to using suffix splitters when the frequency of one symbol in the input string exceeds $b_{\max}$, although this is rare in practice. Should this occur, a good selection of splitters can reduce the number of passes over the text which further reduces the overall running time. However, suffix selection algorithms discovered to date have high constant factors (both on time and space bounds) and do not seem practical. Developing more practical lightweight suffix selection algorithms would be of interest.

Free of alphabet size. A somewhat neglected aspect of many of the space-efficient suffix sorting algorithms is the alphabet size. In particular, the external memory algorithm of Ferragina et al. [2012] assumes a small, constant alphabet. We have shown Kärkkäinen's algorithm works for larger alphabets, but memory use is still quite high. The development of an efficient external memory algorithm free of this assumption is an important open problem. Some theoretical progress has been made [Hon et al., 2003; Na, 2005], but we are aware of no practical approaches. The algorithm due to Hon et al. [2003] was implemented a long time ago but it was slow and used too much space [Kärkkäinen, 2007].

Improving average-case runtime. For inputs where the length of the maximum (or even the average) longest-common-prefix is relatively small ($O(v)$ say), the *Difference Cover Sample of size* v, $\mathrm{DCS}_v(x)$ will not be greatly utilized, with most (or all) of the SA/LCP block determined by multikey quicksort (MKQS) and brute force symbol inspection. For such inputs the memory used by $\mathrm{DCS}_v(x)$ would be better spent on a larger block size, so that more suffixes could be collected each pass, speeding overall processing. We observe that the sampled LCP values contained in $\mathrm{DCS}_v(x)$ and used for Lemma 2 (on page 12) tell us within v the maximum overall LCP value [Puglisi and Turpin, 2008], allowing us to make the decision to discard it in favour of a larger block size before traversal begins. A prototype of this approach has been completed and we plan to integrate it into the suffix

sorting code of Chapter 5.

8.2.2 String Mining

Some potential areas for future research in string mining are as follows.

Further reducing space. While we use dramatically less memory than the algorithm of Fischer, Heun and Kramer [Fischer et al., 2006], the random accesses we make to the input string of concatenated databases during block sorting necessitates they be held in memory, and for very large inputs (hundreds of gigabytes or more) this will be onerous. It may be worth investigating ways to make only sequential accesses to the input string so that it can reside on secondary memory throughout the sorting/mining process. One idea is to explicitly copy v length prefixes of suffixes into the block instead of simply storing a pointer into the text for each suffix. This requires us to make (at most) v times the number of passes, but may still lead to acceptable runtimes.

Parallelization. The block-oriented nature of our algorithm in Chapter 4 means it parallelizes naturally: each processing element can be given a different lexicographically contiguous subset of the splitters and then processes its own part of the suffix array, computing the frequent items therein. Each element would need (read-only) access to $x_{\mathcal{M}}$ and $\mathrm{DCS}_v(x_{\mathcal{M}})$, which is trivial if the elements share a common memory. If processing elements do not share memory, then $x_{\mathcal{M}}$ and $\mathrm{DCS}_v(x_{\mathcal{M}})$ can be computed centrally and then replicated for each node. Care is needed in accumulating counts for short patterns which may span blocks. We sketch a possible approach: choose σ^λ splitters, $\lambda \geq 1$ such that each splitter has a different λ length prefix. Substring patterns of length $\geq \lambda$ are mined in parallel by different cores using the new algorithm. Provided λ is not too large, short patterns with length $< \lambda$ can be gathered separately in a single scan of x using a table of size $O(\sigma^\lambda)$ to accumulate counts.

Suffix tree traversals. Here we have focussed on substring mining, but the bottom-up suffix tree traversal we simulate has many other interesting and useful applications, for example computing the LZ77-factorization of the input [Chen et al., 2008; Gusfield, 1997].

Bibliography

J. Abel. A fast and efficient post BWT-stage for the Burrows-Wheeler Compression Algorithm. In *Proc. of the Data Compression Conference (DCC)*, page 449. IEEE Computer Society, March 2005.

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the Burrows and Wheeler data compression algorithm. In *Proc. of the Data Compression Conference (DCC)*, pages 188–197. IEEE Computer Society, March 1999.

D. Baron and Y. Bresler. Antisequential suffix sorting for BWT-based data compression. *IEEE Transactions on Computers*, 54(4):385–397, 2005.

M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In R. Giancarlo and G. Manzini, editors, *Proc. of the 22nd Annual Combinatorial Pattern Matching (CPM) Symposium*, volume 6661 of *LNCS*, pages 219–231. Springer, June 2011.

J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software Practice Experience*, 23(11), 1993.

J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In M. E. Saks, editor, *Proc. of the 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, January 1997.

S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proc. of the 14th Annual*

Symposium Combinatorial Pattern Matching (CPM), volume 2676 of *LNCS*, pages 55–69. Springer, June 2003.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital Equipment Corporation, 1994.

G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time and space. *Mathematics in Computer Science*, 1(4):605–623, 2008.

Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In K. L. Clarkson, editor, *Proc. of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149, January 1995.

D. Clark. *Compact PAT trees.* PhD thesis, Waterloo University, Canada, 1996.

C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Information Processing Letters*, 75(1-2):9–12, 2000.

J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In U. Meyer and M. de Berg, editors, *Proc. of the 18th Annual European Symposium on Algorithms (ESA)*, volume 6347 of *LNCS*, pages 194–205. Springer, September 2010.

R. Dementiev, L. Kettner, and P. Sanders. Standard template library for XXL data sets. In G. S. Brodal and S. Leonardi, editors, *Proc. of the 13th Annual European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 640 –651. Springer, October 2005.

S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software Practice and Experience*, 32:99–111, 2002.

M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE Computer Society, October 1997.

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

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

P. Ferragina and G. Manzini. On compressing the textual web. In B. D. Davison, T. Suel, N. Craswell, and B. Liu, editors, *Proc. of the 3rd ACM International Conference on Web Search and Web Data Mining (WSDM)*, pages 391–400. ACM, February 2010.

P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In A. Lopéz-Ortiz, editor, *Proc. of the 9th Latin American Theoretical Informatics Symposium (LATIN)*, volume 6034 of *LNCS*, pages 697–710. Springer, April 2010.

P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

J. Fischer. *Data Structures for Efficient String Algorithms*. Dissertation, Mathematik, Informatik und Statistik, Ludwig-Maximilians-Universitat Munchen, 2007.

J. Fischer. Inducing the LCP-array. In F. Dehne, J. Iacono, and J.-R. Sack, editors, *Proc. of the 12th International Symposium on Algorithms and Data Structures (WADS)*, volume 6844 of *LNCS*, pages 374–385. Springer, August 2011.

J. Fischer, V. Huen, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 4213 of *LNCS*, pages 139–150. Springer, September 2006.

J. Fischer, V. Mäkinen, and N. Välimäki. Space efficient string mining under frequency constraints. In *Proc. IEEE International Conference on Data Mining (ICDM)*, pages 193–202. IEEE Computer Society, December 2008.

P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods (Suppl)*, 6(11):S6–S12, 2009.

G. Franceschini and T. Hagerup. Finding the maximum suffix with fewer comparisons. *Journal of Discrete Algorithms*, 9(3):279–286, 2011.

G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *34th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4596 of *LNCS*, pages 533–545. Springer, July 2007a.

G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In D. S. Johnson and U. Feige, editors, *Proc. of the 39th ACM Symposium on Theory of Computing (STOC)*, pages 328–337. ACM, June 2007b.

G. Franceschini, R. Grossi, and S. Muthukrishnan. Optimal cache-aware suffix selection. In *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, February 2009.

J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe - an updated forecast of worldwide information growth through 2011. White Paper, 2008.

S. Gog and E. Ohlebusch. Fast and lightweight LCP-array construction algorithms. In M. Müller-Hannemann and R. F. F. Werneck, editors, *Proc. of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 25–34. SIAM, January 2011.

G. H. Gonnet. Pat 3.1: An efficient text searching system, user's manual. 1987.

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

J. Haas. Linux system administrator's guide. http://linux.about.com/od/lsa_guide/a/gdelsa44.htm, 2012. [Online: accessed May 2012].

H.Bannai, H.Hyyro, A.Shinohara, M.Takeda, K.Nakai, and S.Miyano. An $o(n^2)$ algorithm for discovering optimal boolean pattern pairs. *IEEE-ACM Transactions on Computational Biology and Bioinformatics*, 1(4):159–170, 2004.

W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:251, 2003. ISSN 0272-5428.

L. C. K. Hui. Color set size problem with application to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 644 of *LNCS*, pages 230–243. Springer, April 1992.

H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. of the String Processing and Information Retrieval Symposium and International Workshop on GroupWare (SPIRE/CRIWG)*, pages 81–88. IEEE Computer Society, 1999.

J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

J. Kärkkäinen and S. J. Puglisi. Medium-space algorithms for inverse BWT. In M. de Berg and U. Meyer, editors, *Proc. of the 18th Annual European Symposium on Algorithms (ESA)*, volume 6346 of *LNCS*, pages 451–462. Springer, September 2010.

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *LNCS*, pages 943–955. Springer, June 2003.

J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted Longest-Common-Prefix array. In G. Kucherov and E. Ukkonen, editors, *Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5577 of *LNCS*, pages 181–192. Springer, June 2009.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix computation in suffix arrays and its applications. In A. Amir and G. M. Landau, editors, *Proc. of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2089 of *LNCS*, pages 181–192. Springer, July 2001.

D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 186–199. Springer, June 2003.

D. K. Kim, J. Jo, and H. Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In C. C. Ribeiro and S. L. Martins, editors, *Proc. of the 3rd International Workshop on Experimental and Efficient Algorithms (WEA)*, volume 3059 of *LNCS*, pages 301–314. Springer, October 2004.

J. Kirk. Pcworld:data explosion shakes up it, 2007. URL http://www.computerworld.com/s/article/9036378/Data_explosion_shakes_up_IT. [Online: accessed Jan 2013].

D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3:143–156, 2005.

A. Kügel and E. Ohlebusch. A space efficient solution to the frequent string mining problem for many databases. *Data Mining and Knowledge Discovery*, 17:24–38, 2008.

S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

N. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3): 258–272, 2007.

U. Lauther and T. Lukovszki. Space efficient algorithms for the Burrows-Wheeler backtransformation. In G. S. Brodal and S. Leonardi, editors, *Proc. of the 13th Annual European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 293–304. Springer, October 2005.

U. Manber and E. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327. SIAM, January 1990.

M. A. Maniscalco. Msufsort, 2005. URL http://www.michael-maniscalco.com/msufsort.htm. [Online: accessed March 2011].

M. A. Maniscalco and S. J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics*, 12, 2007.

G. Manzini. Invited lecture: The Burrows-Wheeler transform: Theory and practice. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proc. of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1672 of *LNCS*, pages 34–47. Springer, September 1999.

G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3): 407–430, 2001a.

G. Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001b.

G. Manzini. Two space saving tricks for linear time LCP array computation. In T. Hagerup and J. Katajainen, editors, *Proc. of the 9th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *LNCS*, pages 372–383. Springer, July 2004.

G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

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

Y. Mori. sais: An implementation of the induced sorting algorithm, 2008. URL http://sites.google.com/site/yuta256/sais. [Online: accessed Aug 2011].

S. Muthukrishnan. Efficient algorithms for document retrieval problems. In D. Eppstein, editor, *Proc. of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 657–666. Society for Industrial and Applied Mathematics, January 2002.

J. C. Na. Linear-time construction of compressed suffix arrays using o(n log n)-bit working space for large alphabets. In A. Apostolico, M. Crochemore, and K. Park, editors, *Proc. of the 16th Annual Combinatorial Pattern Matching (CPM) Symposium*, volume 3537 of *LNCS*, pages 57–67. Springer, June 2005.

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

G. Nong. An optimal suffix array construction algorithm. Technical report, Department of Computer Science Sun Yat-sen University, Guangzhou 510275, PRC, 2011.

G. Nong, S. Zhang, and W. H. Chan. Linear time suffix array construction using d-critical substrings. In G. Kucherov and E. Ukkonen, editors, *Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5577 of *LNCS*, pages 54–67. Springer, June 2009a.

G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In J. A. Storer and M. W. Marcellin, editors, *Proc. of the Data Compression Conference (DCC)*, pages 193–202. IEEE Computer Society, March 2009b.

G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In J. Karlgren, , J. Tarhio, and H. Hyyrö, editors, *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *LNCS*, pages 90–101. Springer, August 2009.

M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In W.-Y. Ma, J.-Y. Nie, R. A. Baeza-Yates, T.-S. Chua, and W. B. Croft, editors, *Proc. of the 34th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 555–564. ACM, July 2011.

S. J. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proc. of the 19th International Symposium on Algorithms and Computation (ISAAC)*, volume 5369 of *LNCS*, pages 124–135. Springer, Gold Coast, Australia, December 2008.

S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

K. Sadakane. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *Proc. of the Data Compression Conference (DCC)*, pages 129–138. IEEE Computer Society, April 1998.

D. Salomon and G. Motta. *Handbook of Data Compression (5. ed.)*. Springer, 2010. ISBN 978-1-84882-902-2.

M. Schindler. The szip home page, 2002. URL http://www.compressconsult.com/szip/. [Online: accessed 1-June-2008].

K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *Proc. of the 7th Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics (ALENEX /ANALCO)*, pages 78–85. SIAM, January 2005.

R. Sedgewick. *Algorithms in C++ - Parts 1 - 4: Fundamentals, Data Structures, Sorting, Searching (3. ed.)*. Addison-Wesley-Longman, 1999. ISBN 978-0-201-18537-9.

J. Seward. On the performance of BWT sorting algorithms. In *Proc. of the Data Compression Conference (DCC)*, pages 173 – 182. IEEE Computer Society, March 2000.

J. Seward. The bzip2 and libbzip2 home page, 2004. URL http://www.bzip.org/. [Online: accessed 1-June-2008].

J. F. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56(2):156–180, 1999.

R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithmics*, 9, 2004.

J. Sirén. Compressed suffix arrays for massive data. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *LNCS*, pages 63–74. Springer, August 2009.

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

D. Weese and M. H. Schulz. Efficient string mining under constraints via the deferred frequency index. In P. Perner, editor, *Proc. of the 8th Industrial Conference on Data Mining (ICDM)*, volume 5077 of *LNCS*, pages 374–388. Springer, July 2008.

P. Wiener. Linear pattern matching algorithms. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, October 1973.