Trends in Suffix Sorting: A Survey of Low Memory Algorithms

Jasbir Dhaliwal[1] **Simon J. Puglisi**[1] **Andrew Turpin**[2]

[1] School of Computer Science and Information Technology
RMIT University,
Melbourne, Australia,
Email: {jasbir.dhaliwal,simon.puglisi}@rmit.edu.au

[2] Department of Computing and Information Systems
University of Melbourne,
Melbourne, Australia,
Email: aturpin@unimelb.edu.au

Abstract

The suffix array is a sorted array of all the suffixes in a string. This remarkably simple data structure is fundamental for string processing and lies at the heart of efficient algorithms for pattern matching, pattern mining, and data compression. In many applications suffix array construction, or equivalently suffix sorting, is a computational bottleneck and so has been the focus of intense research in the last 20 years. This paper outlines several suffix array construction algorithms that have emerged since the survey due to Puglisi, Smyth and Turpin [ACM Computing Surveys 39, 2007]. 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 secondary memory (disk) to achieve this goal. We provide a high-level description of each algorithm, avoiding implementation details as much as possible, and outline directions that could benefit from further research.

Keywords: suffix array, suffix sorting, Burrows-Wheeler transform, suffix tree, data compression

1 Introduction

Strings are one of the most basic and useful data representations, and algorithms for their efficient processing pervade computer science. A fundamental data structure for string processing is the suffix array [Manber and Myers, 1993]. 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 problems, such as data compression. The suffix array 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 a 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 suffix array — a process also known as suffix sorting — is one of the main computational bottlenecks. Suffix array construction algorithms (SACAs) therefore have been the focus of intense research effort in the last 15 years or so. The survey by Puglisi et al. [2007] counts 19 different SACAs, and in the last five years even more methods have emerged. The trend in these more recent algorithms has been to use as little memory as possible, either by finding a clever way to trade runtime, or by using compressed data structures, or by using disk, or some combination of these techniques. It is these recent "low memory" SACAs which are our focus in this paper.

In the next section we set notation and introduce basic concepts to be used throughout. 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. Section 3 describes the new algorithms in turn, illustrating each with a worked example. A snapshot experimental comparison is offered in Section 4. We then outline some directions future work might take.

2 Background

Throughout we consider a string x of n characters (or symbols), $x = x[1..n] = x[1]x[2]...x[n]$, 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 $n \log \sigma$ bits of storage without compression.

For $i = 1, ..., n$ we write $x[i..j]$ to represent the *substring* $x[i]x[i+1]...x[j]$ of x that starts at position i and ends at position j. We write $x[i..n]$ to denote the *suffix* of x of length $n-i$, that is $x[i..n] = x[i]x[i+1]...x[n]$, which we will frequently refer to as 'suffix i' for simplicity. Similarly a *prefix* is a substring of the form $x[1..i]$.

The *suffix array* of a string x, which we write as SA, is an array containing all the suffixes of x sorted into lexicographical order. Suffixes are represented as indices into the original string, and thus, the suffix array requires only space sufficient to store n integers, or $n \log n$ bits. More formally, SA is an array SA[1..n] that contains the permutation of the integers 1..n such that x[SA[1]..n] $<$ x[SA[2]..n] $<$ ... $<$ x[SA[n]..n]. Figure 1 shows an example SA for the string *werribbe\$*, where $x[9..9] = \$$ is the lexicographically least suffix, x[6..9]=*bbe\$* is the second least and so on.

Some of the algorithms we describe do not produce the SA directly, but instead produce the *Burrows-*

	1	2	3	4	5	6	7	8	9
x	w	e	r	r	i	b	b	e	\$
SA	9	6	7	8	2	5	4	3	1
BWT	e	i	b	b	w	r	r	e	\$

Figure 1: The Suffix Array and Burrows-Wheeler transform for the string *werribbe\$*.

All Rotations		F	L	i
werribbe\$		\$ werribb	e	1
erribbe\$w		b be\$werr	i	2
rribbe\$we		b e\$werri	b	3
ribbe\$wer		e \$werrib	b	4
ibbe\$werr	Sorted	e rribbe\$	w	5
bbe\$werri	$\longrightarrow$	i bbe\$wer	r	6
be\$werrib		r ibbe\$we	r	7
e\$werribb		r ribbe\$w	e	8
\$werribbe		w erribbe	\$	9

Figure 2: The left column shows all rotations of the string *werribbe\$*. When sorted (right column) this gives the BWT as column L. The F column contains characters of the input sorted lexicographically.

Wheeler transform (BWT) of the input string [Burrows and Wheeler, 1994]. The BWT is a reversible transformation of a string that allows the string to be easily and efficiently compressed [Manzini, 2001]. The BWT was discovered independently of the suffix array, but it is now known that the two data structures are equivalent. 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). As we shall see, suffix sorting is the computational bottleneck for performing the BWT.

The Burrows-Wheeler transform (BWT) transforms the string x by sorting it's n cyclic rotations as in Figure 2. For the full properties of the BWT matrix, we refer the reader to Burrows and Wheeler [1994], Manzini [2001] and Ferragina and Manzini [2005]. We only discuss properties of the matrix that will aid in the explanation of the algorithms. 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. Observe 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. Formally, the BWT of x is an array BWT[1..n] such that:

$$\mathrm{BWT}[i] = \begin{cases} x[\mathrm{SA}[i]-1] \Longleftrightarrow \mathrm{SA}[i] \neq 1 \\ \$ \Longleftrightarrow \mathrm{SA}[i] = 1 \end{cases} \quad (1)$$

The notation used to describe the three main properties of the BWT is as follows [Ferragina and Manzini, 2005].

- C[c] where $c \in \Sigma$ denotes the number of characters that are smaller than c in the BWT.
- Occ(c, q) denotes the number of occurrences of character c in prefix L[1..q].

Combining C and Occ gives the *Last to First mapping*, LF. This function allows one to locate the character $c = \mathrm{L}[i]$ in F.

$$\mathrm{LF}(i) = \mathrm{C}[c] + \mathrm{Occ}(c, i) \quad (2)$$

Observe how the last character in L[i] precedes the character in F[i] at any given row in the string (as each row is a cyclic rotation). For example, character F[6] = 'i' comes after L[6] = 'r' in x. The LF function allows one to locate the character L[i] in F[i]. Continuing with our example, we want to find the character L[6] = 'r' in F. So, LF(6) = C[L[6]] + Occ(L[6], 6) = C['r'] + Occ('r', 6) where C['r'] = 6, as there are six characters that are lexicographically smaller than the character 'r', and Occ('r', 6) = 1, as it occurs once in L[1..6]. Thus, LF(6) = 6 + 1 = 7, indicating that the character 'r' is at position 7 in F, F[LF(6)] = F[7].

3 The Algorithms

3.1 Algorithm K [Kärkkäinen, 2007]

Figure 3 shows a high level description of Kärkkäinen's algorithm. The algorithm begins by computing the difference cover sample [Burkhardt and Kärkkäinen, 2003], a special set of suffixes, the order of which allows the relative order of two arbitrary suffixes to be determined easily. In particular the difference cover sample allows us to determine the relative order of two suffixes, i and j, by first comparing their v character prefixes. If these prefixes are not equal, the order of the suffixes has been determined. However, if a tie occurs, the order of the suffixes is given by the order of suffixes $i+v$ and $j+v$,which are guaranteed to be in the difference cover sample. Here, v is a parameter, and controls a space-time tradeoff. For more information on difference covers, we refer the reader to Burkhardt and Kärkkäinen [2003].

Input: x
1: Compute difference covers.
2: Select suffixes to become splitters.
3: **while** SA not fully computed
4: Collect suffixes based on splitters.
5: Sort suffixes.
6: Write suffixes to disk.
Output: SA.

Figure 3: Pseudocode for Algorithm K, the suffix sorting algorithm due to Kärkkäinen [2007].

The next step in this algorithm is to select and sort a set of *splitters*. For instance, using the example string *ababaacaa\$*, we could select splitters at positions 1, 4, and 8 as indicated by the arrows shown below. The last suffix, suffix 10, is also taken as it is the smallest suffix in the string.

	1	2	3	4	5	6	7	8	9	10
x	a	b	a	b	a	a	c	a	a	\$
	↑			↑				↑		↑

j	S	Suffixes
1	10	\$
2	8	aa\$
3	1	ababaacaa\$
$m = 4$	4	baacaa\$

These m splitters, S[1..m] are then sorted and stored in memory. Notice that adjacent pairs of splitters divide the suffix array into blocks. In the next phase of the algorithm these splitters will be used to compute the suffix array one block at a time, where a block of suffixes can fit in RAM.

	1	2	3	4	5	6	7	8	9	10
SA_1	10	9								
SA_2			8	5	3					
SA_3						1	6			
SA_4								4	2	7

Figure 4: SA_i represents the portion of the SA constructed using a lower bound splitter $j = i$ and and upper bound splitter $j = i + 1$. Note the final pass using splitter m does not have an upper bound.

With the splitters sorted, Algorithm K proceeds by taking the first two splitters S[1] = 10(*$*) and S[2] = 8 (*aa$*) and using them them as lower bound and upper bound values. A left to right scan of the string is then made. During the scan, any suffix larger than or equal to the lower bound and smaller than the upper bound is collected. In our example, this means on the first scan suffix 1 is ignored as it lexicographically larger than the upper bound value. However, suffix 9 (*a$*) is picked as it is lexicographically larger than suffix S[1] but lexicographically smaller than suffix S[2]. In order to determine efficiently whether a suffix falls between to splitters we make use of the difference cover sample, so at most v character comparisons are required. At the end of the pass, there are two suffixes in the first block: suffix 9 and suffix 10. These suffixes are ordered to depth v using multikey quicksort [Bentley and Sedgewick, 1997] and if there is a tie, the difference covers sample is used again to order them. The suffixes are then written to disk, and the memory that was used to hold them (during collection) is reclaimed.

Subsequent passes work similarly. The upper bound splitter from the last round becomes the lower bound, and the next splitter in the set becomes the new upper bound. In our example, S[2] = 8 becomes the lower bound and S[3] = 1 becomes the upper bound. Continuing with our example, suffix 1 is compared with suffix 8 and is lexicographically larger than suffix 8 but it is not lexicographically smaller than itself, thus it is ignored. On the other hand, suffix 3 does fall between the splitters and so it is collected. This process is continues until all the suffixes in the range $[x[8..n], x[1..n])$ have been collected. The suffixes are then sorted and written to disk. The process continues, deriving the SA as shown in Figure 4.

As splitters are randomly chosen (from a lexicographic point of view), there is no guarantee they divide the SA evenly, and so some blocks my be much larger than others. In particular, a block may exceed RAM limits. Kärkkäinen provides a clever method for dealing with this problem. If, while collecting suffixes for the current block, the amount of available memory is reached, the scan is halted and the contents 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 upper bound splitter, and the scan resumes. This trick does not (asymptotically) increase the number of scans.

Algorithm K requires $O(n \log n + vn)$ time and $O(n \log n/\sqrt{v})$ bits of space in addition to the text, where v is the period of the difference cover used. The algorithm also allows for different space-time tradeoffs, depending on the amount of available memory. Setting $v = \log^2 n$ gives a runtime of $O(n \log^2 n)$ and requires $O(n)$ bits of working space.

3.2 Algorithm NZC [Nong et al., 2009]

The algorithm of Nong et al. 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. The algorithm uses only space to hold the input string and the resulting suffix array. In this sense it is "lightweight" [Manzini and Ferragina, 2004], and ranks among the most space efficient algorithms at the time of the 2007 survey, however it is the most space consuming algorithm we discuss in the survey. Algorithm NZC also runs in linear time in the length of the string, settling an open problem posed by [Puglisi et al., 2007] by showing it possible to be simultaneously lightweight in space usage and linear in runtime.

The idea of differentiating the suffixes into subsets is similar to Ko and Aluru [2005]. Suffixes are split into *Larger* and *Smaller* types (L and S respectively) depending on their lexicographic order relative to their righthand 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 sorted order of the S suffixes.

Input: x
1: Label each position S or L.
2: Identify LMS substrings and place in SA_1.
3: Scan SA_1 left to right: move type Ls to get SA_2.
4: Scan SA_2 right to left: move type Ss to get SA.
Output: SA.

Figure 5: Pseudocode for the copying algorithm due to Nong et al. [2009].

The algorithm begins by making a pass over the string, $x[1..n]$ assigning a type of either L or S to the suffix beginning at each position depending if it is Larger or Smaller than its righthand neighbour 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 Left Most S-type (LMS) positions, which are the positions i such that suffix i is of type S and suffix $i - 1$ is of type L for $x > 1$.

The end-of-string symbol, $x[n] =$ \$, is defined to be S, and hence is also an LMS suffix (as its left neighbour must be larger than it). For example, types are assigned when a right to left scan is made over our example string, *ababaacaa$* as follows.

	1	2	3	4	5	6	7	8	9	10
x	a	b	a	b	a	a	c	a	a	\$
type	S	L	S	L	S	S	L	L	L	S
LMS			*		*					*

Suffix 1 is categorised as type S (*ababaacaa$*) as it is lexicographically smaller than suffix 2 (*babaacaa$*). Suffix 2 is of type L as it is lexicographically larger than suffix 3 (*abaacaa$*). This process is repeated until the types are assigned to all the suffixes of the string. While making the scan, we also identify the LMS types, which are marked with an $*$.

After identifying the types, the indices of the LMS positions are placed in buckets in the SA, processing the LMS left to right, to get SA_1. Each suffix beginning with an LMS character is placed at the rightmost end of its character's bucket in SA_1, and the bucket 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 example string:

	$	a	b	c
j	0	1	2	3
C[j]	1	7	9	10

The first LMS character encountered is 'a', the beginning of suffix 3, which is placed in position 7, as C[a] = 7, and C[a] is decremented. Next we encounter 'a' at the beginning of suffix 5, and so it is inserted at C[a] = 6. Finally, suffix 10 beginning with $ is inserted at C[$] = 1. The end result is shown here, with the boundaries of each bucket denoted by brackets.

	$	(			a			)(	b		)(	c	)
	1	(	2	3	4	5	6	7	)(	8	9	)(	10	)
SA_1	10	(					5	3	)(			)(		)
SA_2	10	(	9	8			5	3	)(	4	2	)(	7	)
SA	10	(	9	8	5	3	1	6	)(	4	2	)(	7	)

For the next pass, we make a left to right ($j = 1..n$) scan over SA_1 to derive the order of the L suffixes. Within a bucket, type L suffixes always come before type S suffixes since the latter is lexicographically larger than the former. During the scan, if we find a suffix i ($i = SA_1[j]$), we look for suffix $i - 1$ and if that suffix is of type L and has not been placed in SA_1, we place this suffix in the first empty position of the group. Hence our bucket counters start off as $C = \{1, 2, 8, 10\}$, one more than the number of characters in the string less than the index. In our case, when $j = 1$ and $i = SA_1[1] = 10$, we find suffix 10-1 = 9 which is a suffix of type L that is yet to be sorted. Therefore, we place it at the start of its 'a' bucket (C[a] = 2) and increment the bucket count. Continuing the scan, $i = SA_1[2] = 9$, suffix 9-1=8 is a suffix of type L. So, it is placed at the next available position in bucket 'a', which is C[a] = 3. This process repeats itself until all type L suffixes have been placed in the array to get SA_2.

After placing the type L suffixes, we now collect the type S suffixes. We make a right to left ($j = n..1$) scan over SA_2 and for each i ($i = SA_2[j]$), we look for suffix $i - 1$ and if the suffix of type S and has yet to be sorted, we place this suffix at the end of its bucket indicated by its first character. (The bucket counters have again been reset.) For example, when $j = n$, $i = 7$, we find a suffix 7-1=6 and it is a type S suffix from the group 'a'. So, we overwrite $SA_2[7]$ with 6, and decrement the bucket counter for 'a'. Continuing with our example, when $j = 9$, $i = 2$, we find a suffix 2-1=1 and it is a suffix of type S. So, we overwrite the suffix 5 in $SA_2[6]$ with suffix 1. This process is repeated until all the type S have derived (as indicated by SA).

3.3 Algorithm OS [Okanohara and Sadakane, 2009]

The algorithm of Okanohara and Sadakane uses the same framework as the algorithm by Nong et al. [2009], described in the previous section, but derives the BWT string, rather than the SA. Like NZC, OS runs in linear time, but via a careful implementation of the required data structures, it is able to reduce memory overheads to $O(n\log\sigma\log\log_\sigma n)$ bits.

Like NZC, suffixes are first classified into types L and S. Then, the LMS substrings are identified. The difference comes in that OS explicitly stores the LMS substrings in queue-like data structures.

Figure 6 illustrates the operation of OS on the example string *ababaacaa$*. The LMS substrings are stored in queues and their actual positions in x are stored in F. When these LMS substrings are moved among queues the L suffixes are induced. Their movement is indicated by the arrows and captured in P.

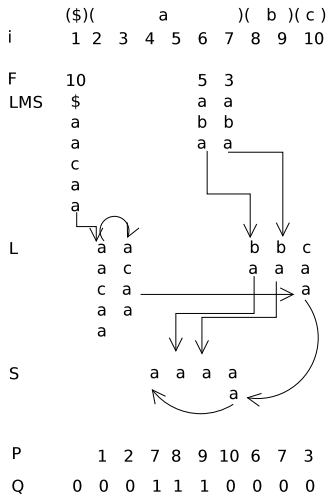

Figure 6: An example of the modified algorithm that induces the sort of the string *ababaacaa$* using LMS substrings that are moved in queue-like data structures.

For example, since the LMS substring, *$aacaa* was moved from position $i = 1$ to $i = 2$, P[2] = 1. Furthermore, inducing suffix 9 of type L and so on.

Lastly, Q[i] shows the location of the LMS substrings in the queues that are further indicated with '1's. Moreover, the order of two adjacent LMS substrings can be determined by tracing this Q data structure together with P where $i = P[i]$ is computed repeatedly when Q[i] = 1 until the head of the substring is reached (stored in F). For example, when $i = 4$ and Q[i] = 1, $i = \{P[4] = 7, P[7] = 10, P[10] = 3, P[3] = 2, P[2] = 1\}$. In fact, the order of two adjacent LMS substrings can be determined in time proportional to their lengths.

3.4 Algorithm FGM [Ferragina et al., 2010]

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

Figure 7: Pseudocode for computing the BWT by Ferragina et al. [2010].

Figure 7 shows the high level description of Ferragina et al.'s algorithm. In a nutshell, the BWT is computed for the first block of certain size and stored on disk. For the next subsequent blocks, the BWT is computed in internal memory which we will call as BWT_{int} and merged with the external BWT that is on disk which we will refer as BWT_{ext}. This process is repeated until the entire BWT for the text has been computed.

3.4.1 Compute BWT for the first block

The algorithm begins by dividing the text into blocks of size m. For example, with $m = 3$:

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
B		3			2			1		

The first block, $B_1 = x[7..9]$, is brought into memory, its BWT is derived and stored on disk as, BWT_{ext}.

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{ext}							b	a	a	
SA							7	8	9	
B		3			2			1		

3.4.2 Compute BWT for subsequent blocks

The BWT computation for the next blocks differs from the first block. In our example, text for the blocks $B_1 = x[7..9]$ and $B_2 = x[4..6]$ are brought into memory. Suffixes are compared näively, character by character. For example, when comparing suffix 4 (*aaaaab$*) with suffix 5 (*aaaab$*), the suffix was ordered when a mismatch occurred at character 5, position 9 in the string. Having ordered the suffixes (built the SA) for block B_2, the BWT is computed and stored in memory as BWT_{int}.

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{int}				a	a	a				
BWT_{ext}							b	a	a	
SA				4	5	6	7	8	9	
B		3			2			1		

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

In our case, the current block is B_2.

k	$x[k]$	G				
		0	1	2	3	4
10	$	1				
9	b				1	
8	a					1
7	a					1

Using the G array for the SA of the first block, BWT_{ext} and BWT_{int} are merged. For instance, G[0] means there is one suffix that is smaller than suffix 4 (that's suffix 10 *'$'*). However, since the BWT for this suffix was never computed, it can safely be ignored. Since G[0..2]=0, we can copy $BWT_{int}[0..2]$ to disk. Then, the $BWT_{ext}[0..2]$ is copied (as indicated by G[3] and G[4]) which shows that there are three suffixes that are larger than the suffix 6.

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{ext}				b	a	a	a	a	a	
SA				4	5	6	7	8	9	
B		3			2			1		

Lastly, we compute the BWT for B = 3. Similar to the previous pass, $x[1..6]$ is brought into memory and the SA for it is computed näively, via character comparisons and having a tie, a data structure that orders the suffixes in $O(m)$ time is used (see Ferragina et al. [2010] for implementation details).

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{int}	b	a	#							
BWT_{ext}				b	a	a	a	a	a	
SA	2	3	1	4	5	6	7	8	9	
B		3			2			1		

k	$x[k]$	G				
		0	1	2	3	4
10	$	1				
9	b			1		
8	a			1		
7	a			1		
6	a			1		
5	a			1		
4	a			1		

$G[2] = 6$ shows there are six old suffixes that exist between SA[2] and SA[3]. Therefore, the new BWT_{ext} is merged as below.

	1	2	3	4	5	6	7	8	9	10
x	(b	a	a)	(a	a	a)	(a	a	b)	$
BWT_{int}										
BWT_{ext}	b	a	b	a	a	a	a	a	#	
SA	2	3	1	4	5	6	7	8	9	
B		3			2			1		

4 Experiments

In this section we provide a brief experimental comparison of the recent algorithms for which we have efficient implementations. The experiments were run on an otherwise idle 3.16GHz Intel Core (TM) 2 Duo CPU E8500 of 4 GB of RAM and a cache size of 6144 KB. The operating system is Ubuntu 10.04.3. All the code was compiled with gcc/g++ version of 4.4.3 and the `-O3` optimisation flag. The memory usage was measured using the memusage tool and times reported are the minimum of three runs, measured with the C `time` function.

We measured implementations of Algorithms K, S, NZC and FGM. From personal communication with Okanohara [2010], we understand that there is no publicly available code for Algorithm OS. For Algorithm NZC, we downloaded a publicly available implementation by Yuta Mori [SAIS, 2011]. Likewise, we downloaded a publicly available implementation for Algorithm S by the author himself. We implemented Algorithm K ourselves. Of these algorithms, Algorithm NZC is an in-memory algorithm; Algorithm FGM is an external memory algorithm (using disk as working space) and Algorithm K is a semi-external algorithm (writing only the BWT to disk).

Table 1 shows our test data: 200MB of ENGLISH, DNA and SOURCES from the Pizza and Chili [2009] Corpus. We equated the amount of memory allocated by Algorithm K and Algorithm FGM (to 481MB). Algorithm NZC requires both the SA and input string to be resident in RAM and so consumes 1,000MB of memory.

Running times are reported in Table 2. Algorithm NZC is clearly fastest, but uses twice the memory of the other two algorithms. This result is expected as NZC uses an induced copying heuristic that is used in all fast algorithms [Puglisi et al., 2007]. Also, NZC is an in-memory algorithm, whereas Algorithm K requires only the text to be in memory and for FGM, all the data including the text resides in external memory. The size of the available RAM on the test machine is big enough to hold both the input string and

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

Table 1: LCP is the longest common prefix between adjacent suffixes in the suffix array. A higher LCP generally increases the cost of suffix sorting.

Algorithms	ENGLISH	DNA	SOURCES
K	244	246	293
NZC	**58**	**59**	**41**
FGM	405	421	324

Table 2: Total wall clock time taken in seconds by the algorithms to run on the 200MB dataset. The minimum time is shown in bold.

Algorithms	Peak Memory
K	**481**
NZC	1000
FGM	**481**

Table 3: The peak memory usage (MB) by the algorithms for the 200MB dataset. The minimum is shown in bold.

the SA in memory, and so Algorithm FGM and Algorithm K might get faster if they were tuned for RAM.

5 Concluding Remarks

Algorithms for suffix sorting continue to mature. In this paper we have surveyed a number of recent techniques, all of which aim to reduce the amount of main memory required during sorting. Algorithms also continue to emerge for related problems; for example Sirén [2009] describes a method for directly building the compressed suffix array of a collection of strings, such as documents or genomes. Bauer et al. [2011] consider a similar problem, but where each string in the collection is relatively short (in particular the length of a sequenced DNA fragment).

A somewhat neglected aspect of many of the new algorithms is the alphabet size. In particular the external memory algorithm of Ferragina et al. [2010] assumes a small, constant alphabet. 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.

Another area to explore, at least for obtaining practical algorithms, is the blending of older suffix sorting heuristics, surveyed in Puglisi et al. [2007], with the low memory algorithms examined here. Some initial attempts to introduce pointer copying heuristics to Kärkkäinen's algorithm are described by Dhaliwal and Puglisi [2011].

Finally, a related problem called *suffix selection*, 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 have the potential to aid suffix sorting algorithms – for example they could be used to choose good splitters in Kärkkäinen's algorithm – however suffix selection algorithms discovered to date have high constant factors (both on time and space bounds) and do not seem practical.

6 Acknowledgements

Our thanks goes to the authors that made their code available to us. This work was supported by the Australian Research Council.

References

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

M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *22nd Annual Combinatorial Pattern Matching (CPM) symposium*, volume 6661 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2011.

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

S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 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.

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. 18th Annual European Symposium on Algorithms (ESA)*, volume 6347 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2010.

J. Dhaliwal and S. J. Puglisi. Fast semi-external suffix sorting. Technical Report TR-11-1, RMIT University, School of Computer Science and Information Technology, 2011.

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

P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM '10: Proceedings of the third ACM international conference on Web search and data mining*, pages 391–400, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-889-6. doi: http://doi.acm.org/10.1145/1718487.1718536.

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 *Lecture Notes in Computer Science*, pages 697–710. Springer, 2010.

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(JDA)*, 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 *Lecture Notes in Computer Science*, pages 533–545. Springer, 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, 2007b.

G. Franceschini, R. Grossi, and S. Muthukrishnan. Optimal cache-aware suffix selection. In *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, Germany, 2009.

D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

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.

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

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

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

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

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

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, *16th Annual Combinatorial Pattern Matching (CPM) Symposium*, volume 3537 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2005.

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

G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference (DCC)*, pages 193–202, Snowbird, UT, USA, 2009. IEEE Computer Society.

D. Okanohara. Personal communication, 2010.

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 *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009.

M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information*, SIGIR '11, pages 555–564, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0757-4. doi: http://doi.acm.org/10.1145/2009916.2009992.

Pizza and Chili. Pizza and chili corpus, compressed indexes and their testbeds. May 2009. URL `http://pizzachili.dcc.uchile.cl/`.

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

SAIS. Sais: An implementation of the induced sorting algorithm. Aug. 2011. URL `http://sites.google.com/site/yuta256/sais`.

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 *Lecture Notes in Computer Science*, pages 63–74. Springer, 2009.