A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation

Kunihiko Sadakane
Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, JAPAN
sada@is.s.u-tokyo.ac.jp

Abstract

We propose a fast and memory efficient algorithm for sorting suffixes of a text in lexicographic order. It is important to sort suffixes because an array of indexes of suffixes is called *suffix array* and it is a memory efficient alternative of the suffix tree. Sorting suffixes is also used for the Burrows-Wheeler transformation in the Block Sorting text compression, therefore fast sorting algorithms are desired.

We compare algorithms for making suffix arrays of Bentley-Sedgewick, Andersson-Nilsson and Karp-Miller-Rosenberg and making suffix trees of Larsson on speed and required memory and propose a new algorithm which is fast and memory efficient by combining them. We also define a measure of difficulty of sorting suffixes: *average match length*. Our algorithm is effective when the average match length of a text is large, especially for large databases.

1 Introduction

1.1 Suffix array

Today large databases become available, such as full text of newspapers or Web pages, and Genome sequences, therefore it is important to store them on memory for quick queries. The suffix array [10] is a compact data structure for on-line string searches. For such purposes, suffix tree is used because it enables us to find the longest substring of a large text that matches the query string in linear time. However, the suffix tree requires huge memory and therefore it is impossible to use it for large text.

Though the suffix tree can be constructed in linear time [11, 13, 9], the time complexity depends on alphabet size. Searching time also depends on the alphabet size, and it becomes slow if the alphabet size is large. On the other hand, construction and searching time of the suffix array does not depend on the size of alphabet. Therefore searching time of the suffix array may be comparable with the suffix tree though it has superlinear time complexity of $O(P \log n)$ where P is the length of a query string and n is the length of a large text.

1.2 Block Sorting data compression

The Block Sorting is a lossless data compression scheme [4]. Though its compression ratio is comparable with variants of the PPM [6], its compression speed is faster than them and decompression speed is much faster than compression. Moreover, the required memory is smaller than the PPM, therefore the block sorting is a promising scheme.

The Burrows-Wheeler transformation used in the Block Sorting is a permutation of a text to be compressed and it requires sorting of all suffixes of the text. Though the Block Sorting is faster than the PPM, it is slower than the `gzip` because sorting of long text is required. Therefore the Block Sorting generally divides a long text into many *blocks* and performs the Burrows-Wheeler transformation to each block and that makes the compression ratio worse. Therefore speeding-up of sorting suffixes for large texts is important.

1.3 Our results

We propose a fast and memory efficient algorithm for sorting all suffixes of a text. We use the doubling technique of Karp-Miller-Rosenberg [8], ternary partitioning of Bentley-Sedgewick [3], and numbering method of Andersson-Nilsson [1] and Manber-Myers [10]. It requires only $9n$ bytes for sorting a text of length n by using a clever ordering of sorting and an integer encoding. Note that we assume the size of integers is 32bits. We define a measure of difficulty of sorting suffixes: average match length (AML). Our algorithm requires much more memory than the Bentley-Sedgewick, which requires $5n$ bytes, but our algorithm is faster than other sorting algorithms when AML is large. It is faster than the suffix tree construction algorithm of Larsson [9] if the alphabet size is large and it requires less than half of memory for the suffix tree. When a text becomes long, many pairs of words appear repeatedly and the AML becomes large, therefore our algorithm is practical for large texts.

1.4 Definitions

Assume that we make a suffix array of a text $X = x[0..n-1] = x_0x_1\dots x_{n-1}$. The size of alphabet Σ is constant. We add a special symbol '\$' at the tail of the text X, that is, $x_n = \$$. It is not in the alphabet and smaller than any other symbols in the alphabet. Suffixes of the text are represented by $S_i = x[i..n]$. The suffix array $I[0..n]$ is an array of indexes of suffixes S_i. All suffixes in the array are sorted in lexicographic order. A suffix S_i is lexicographically less than a suffix S_j if $\exists l \geq 0\ x[i..i+l-1] = x[j..j+l-1]$ and $x[i+l] < x[j+l]$.

We use another ordering $<_k$ defined by comparison of prefixes of length k as follows.

- $S_i <_k S_j \iff 0 \leq \exists l < k\ x[i..i+l-1] = x[j..j+l-1]$ and $x[i+l] < x[j+l]$
- $S_i =_k S_j \iff x[i..i+k-1] = x[j..j+k-1]$
- $S_i >_k S_j \iff 0 \leq \exists l < k\ x[i..i+l-1] = x[j..j+l-1]$ and $x[i+l] > x[j+l]$

2 Related works

It is necessary to sort all suffixes of a string in lexicographic order for making a suffix array. Several algorithms below are available to sort suffixes. First two algorithms are general algorithms for sorting strings and last two algorithms are for sorting suffixes of a string.

2.1 Bentley-Sedgewick algorithm

This is a practical algorithm for sorting strings [3]. It is similar to the quick sort, but it recursively partitions pointers of strings into three parts. A pivot used to partition is the first symbol in a string and all strings are partitioned into less than, equal to and greater than the symbol according to their first symbols. A key idea of this algorithm is to use the equal part. Strings in it have same first symbols, therefore we can sort the strings without comparing the first symbols.

The Bentley-Sedgewick algorithm is used in a free software `bzip2` [12], an implementation of the Block Sorting.

2.2 Andersson-Nilsson algorithm (Forward Radix Sort)

This is a general algorithm for sorting strings using radix sort [1]. First all strings are sorted and split into groups according to their first symbols by using a radix sort. Next all strings are moved to buckets according to second symbols. The buckets are traversed in alphabetical order and strings in them are returned to their own groups. Now all strings are sorted according to first two symbols, therefore we split the groups and iterate these operations until all strings are sorted. This algorithm is simple, but it theoretically has good time complexity.

2.3 Karp-Miller-Rosenberg (KMR) algorithm

This is an algorithm for finding repeated patterns in a string [8] and it can be used for sorting suffixes. First all suffixes S_i of a string are split into groups according to their first symbols and given numbers $V[i]$. All suffixes in a group have the same $v = V[i]$ and the group is represented by the number v. All groups have different numbers. Next suffixes of each group are split into subgroups according to the $V[i]$ of suffixes. As a result, all suffixes are split according to their first two symbols. This operation continues until all suffixes belong to groups of size one. If groups are split and ordered in alphabetical order, all suffixes can be sorted in lexicographic order.

The key of the algorithm is so-called *doubling technique*. Because all suffixes in a group have the same first k symbols $x[i..i+k-1]$ after an iteration, we can split the groups according to $x[i+k..n]$. Since the numbers $V[i+k]$ are already calculated in the last iteration, we can split the groups according to $x[i+k..i+2k-1]$ in the next iteration. Consequently, all suffixes can be sorted in lexicographic order within $\lceil \log n \rceil$ iterations.

2.4 Manber-Myers algorithm

This algorithm [10] uses the doubling technique in the KMR algorithm. First we sort and group all suffixes by their first symbols using bucket sort. Suffixes are given $V[i]$ like the KMR algorithm. Next all groups are traversed in order of $V[i]$ to sort suffixes in groups by the second symbols. If $I[0] = i$, S_{i-1} is the smallest suffix among the group $V[i-1]$. We can move suffixes to their correct position according to first two symbols. The number of iterations is less than $\lceil \log n \rceil$ and each iteration can be done in $O(n)$ time, therefore this algorithms works in $O(n \log n)$ time.

2.5 Larsson's suffix tree construction algorithm

This algorithm can maintain a sliding window of a text in linear time and it can be used for PPM-style data compression [9]. Though the suffix tree requires more memory than the suffix array, it enables us to search substrings in linear time.

3 Proposed algorithm

3.1 Idea

We propose an algorithm for sorting suffixes using the KMR algorithm and the Bentley-Sedgewick algorithm. Though the original KMR algorithm uses bucket sort for each group, it is not practical because the number of different values of V_i becomes large and the number of unsorted elements becomes small as iteration of the KMR algorithm proceeds and cost of initializing the bucket becomes large. The algorithm of Manber and Myers solved this problem of the KMR algorithm, but it traverses all suffixes in each iteration even if almost all suffixes were already sorted. We use a comparison-based algorithm for sorting each group and sort only unsorted strings in each iteration.

Our algorithm maintains $<_k$ order of all suffixes in an array $V[0..n]$ and iterates until all $V[i]$ have different values. After iteration i, all suffixes are sorted according to their fist $k = 2^i$ symbols. Suffixes whose first k symbols are equal form a group. Groups are called *unsorted* if their size is more than one and they are called *sorted* if the size is one. We want to update $V[i]$ and $I[i]$ for only unsorted groups, therefore $V[i]$ and $I[i]$ of sorted groups must be consistent in all iterations, that is, if a suffix S_i is in a sorted group, $V[i]$ and $I[j] = i$ must be equal to the values in case of all suffixes are sorted.

Definition 1 *The array $V[0..n]$ is consistent with $<_k$ order if*

- $S_i < S_j \to V[i] \leq V[j]$,
- $S_i > S_j \to V[i] \geq V[j]$, *and*
- $S_i \neq_k S_j \to V[i] \neq V[j]$.

Note that we can assign different values to $V[i]$, $V[j]$ even if $S_i =_k S_j$. This enables us to sort suffixes without temporary arrays.

Our algorithm continues maintaining $V[0..n]$ is always consistent. If $V[i]$ represents the number of suffixes which are less than a suffix S_i according to $<_k$ order, the value is consistent and $V[I[i]] = i$ if a suffix S_i is in a sorted group. This numbering comes from [10, 1].

Our algorithm proceeds as follows:

1. sort S_i by their first symbols using bucket sort, initialize $V[i]$ $(1 \leq i \leq n)$, let $k = 1$

2. sort unsorted groups according to $<_k$ order using a comparison-based sorting algorithm

3. split groups and update V

4. combine consecutive sorted groups into one sorted group

5. if the number of groups is n, exit

6. $k = k \times 2$ and goto 2.

Sorting and splitting a group is done as follows. Assume that suffixes in a group $G = I[s..e]$ are equal according to $<_k$ order and now we calculate $<_{2k}$ order and $V[i]$ for $i \in I[s..e]$. First we sort the suffixes according to $V[I[i] + k]$ $(s \leq i \leq e)$ by using a comparison-based algorithm, then we split the group and update $V[i]$. The group is split if $V[I[i] + k] \neq V[I[i + 1] + k]$

3.2 Implementation

Groups are represented by three arrays: $I[0..n]$, $V[0..n]$ and $S[0..n]$. After iteration k, they represents

- $I[i]$: the index of i-th suffix in $<_k$ order,
- $V[s]$: the group of suffix S_s,
- $S[g]$: the size of the group g.

To implement the algorithm without temporary arrays, we must update groups carefully. All groups are traversed in $<_k$ order in iteration k. The first group begins at index 0 and its $size$ is $S[0]$. We sort suffixes $S_s \in I[0..size - 1]$ according to $V[s + k]$. The next group begins at index $g = size$, therefore we sort $S_s \in I[g..g + S[g] - 1]$. After all groups are traversed, they will be split to form $<_{2k}$ order. If consecutive suffixes $I[i]$ and $I[i+1]$ in group g have different group $V[I[i]+k]$ and $V[I[i+1]+k]$, the group g is split between index i and $i+1$, that is, the size of the group is updated. Next we update the array V for having different value according to $<_{2k}$ order.

If the size of a group g becomes one in iteration k, the group is sorted and it is unnecessary to sort the group after the iteration. Therefore we want to skip such groups. To do so, we combine consecutive sorted groups into one group. We use sign flag of the size of groups to indicate whether a group is sorted or not. The array $S[g]$ is a memory efficient alternative of a linked list which contains unsorted groups.

Figure 1 shows an example of the proposed algorithm for sorting suffixes of "tobeornottobe\$." In the figure, second row shows the example string and third, fourth and fifth rows show the result of sorting suffixes by their first symbols. Negative numbers in $S[0]$, $S[5]$ and $S[10]$ show that suffixes $S_{I[0]}$, $S_{I[5]}$ and $S_{I[10]}$ are already sorted. In the first iteration ($k = 1$), we sort suffixes in groups 1 ($I[1..2]$), 3 ($I[3..4]$), 6 ($I[6..9]$) and 11 ($I[11..13]$) separately by the second symbols. The second symbol of a suffix S_s is the first symbol of the suffix S_{s+1}, that is the key idea of the KMR algorithm, and is represented by $V[s + 1]$, therefore we sort $I[i]$ by $V[I[i] + 1]$. After the sorting, we split the groups, that is, update $S[i]$, and then update $V[i]$.

In the beginning of the second iteration ($k = 2$), all suffixes have been sorted in $<_2$ order. Therefore we sort suffixes in a group by third and later symbols. The element $V[s + 2]$ represents the $<_2$ order of $x_{s+2}x_{s+3}$, therefore after sorting suffixes by $V[I[i] + 2]$, we obtain $<_4$ order of suffixes.

Figure 1: An example of our algorithm

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
k	x_i	t	o	b	e	o	r	n	o	t	t	o	b	e	\$
	$I[i]$	13	2	11	3	12	6	1	4	7	10	5	0	8	9
	$S[i]$	-1	2		2		-1	4				-1	3		
	$V[I[i]]$	0	1	1	3	3	5	6	6	6	6	10	11	11	11
1	$V[I[i]+k]$		3	3	6	0		1	10	11	1		6	11	6
	$I[i]$		2	11	12	3		1	10	4	7		0	9	8
	$S[i]$	-1	2		-3			2		-3			2		-1
	$V[I[i]]$		1	1	3	4		6	6	8	9		11	11	13
2	$V[I[i]+k]$		8	0				4	3				1	1	
	$I[i]$		11	2				10	1				0	9	
	$S[i]$	-11											2		-1
	$V[I[i]]$		1	2				6	7				11	11	
4	$V[I[i]+k]$												8	0	
	$I[i]$												9	0	
	$S[i]$	-14													
	$V[I[i]]$												11	12	
	$I[i]$	13	11	2	12	3	6	10	1	4	7	5	9	0	8

3.3 Further improvements

3.3.1 One pass implementation

The above implementation requires three passes: sorting, updating S, and updating V. Note that combining groups can be done in sorting pass of the next iteration.

However, we can change the algorithm to work in one pass. To update V in the sorting pass, we must guarantee values of V to be consistent. If an element $V[i]$ is updated, the value is always incremented and it may become larger than other values which are greater than S_i in $<_k$ order. However, if we update the array V right to left, it is always consistent. Therefore we use quick sort or Bentley-Sedgewick algorithm and sort recursively from greater part to less part.

3.3.2 Initial bucket sort

Because we use comparison-based sorting algorithms for each group, the first iteration requires $O(n \log n)$ time. We can accelerate the iteration using bucket sort by first two symbols of suffixes. We use an array of size $|\Sigma|^2$ and count the number of all patterns of two symbols. All suffixes are sorted according to first two symbols in $O(n)$ time.

3.3.3 Word size of the array S

Because elements of the array S is the size of groups, range of the values varies from 1 to n. However, only one byte is enough for each element.

If the size of a group g is s, $S[g+1..g+s-1]$ are not used. Therefore we can store the size of the group to the unused array if the size cannot be represented by one byte.

4 Experimental results

We have experimented on sorting time of various algorithms. We used SparcStation20 with 128MB memory. Algorithms we experimented are Bentley-Sedgewick, Manber-Myers, Larsson, and our algorithm. All algorithms except the Larsson use initial bucket sorting. Larsson is not a sorting algorithm but a suffix tree construction algorithm. Children of nodes in a suffix tree are represented by a linked list and elements of the list is rearranged by move-to-front rule. Our algorithm uses the doubling technique of the KMR algorithm and ternary partitioning of the Bentley-Sedgewick.

Table 1 shows memory requirements of the algorithms. Bentley-Sedgewick uses only one array $I[0..n]$ and a text buffer. Larsson uses five arrays of size n and a text buffer. Our algorithm uses two arrays I and V of $4n$ bytes and an array S of n bytes. Both Bentley-Sedgewick and our algorithm use a stack. However, the depth of the stack of our algorithm is smaller than that of Bentley-Sedgewick because Bentley-Sedgewick is depth-first algorithm and ours is breadth-first algorithm. Though Larsson's suffix tree requires $22n$ bytes in the worst case, we found that generally the number of internal nodes is about half of leaves and the tree requires about $15n$ bytes by experiments.

Table 2 shows time for sorting suffixes or making suffix tree. We use files in Calgary corpus [5] and Canterbury corpus [2] for benchmark. In the table, first,

Table 1: memory requirements

algorithm	memory
Bentley-Sedgewick	$5n$ + stack
Manber-Myers	$13n$
Larsson	$22n$
our algorithm	$9n$ + stack

second and third columns show filename, size and average match length (AML) of the files respectively. The AML is defined as follows.

$$AML = \frac{1}{n-1}\sum_{i=1}^{n-1}(\text{match length of } S_{I[i]} \text{ and } S_{I[i+1]})$$

Note that the AML is average number of symbols per suffix to verify that all suffixes are sorted correctly. Fourth to seventh columns show sorting time of Bentley-Sedgewick, Manber-Myers, Larsson, and our algorithm respectively. Files are sorted

Table 2: sorting time and average match length

files			sorting time (s)			
name	size (byte)	AML	BS	MM	Lar	ours
geo	102400	3.5	**0.6**	0.9	3.3	0.7
book1	768771	7.3	**8.2**	18.0	14.3	9.5
progc	39611	8.2	0.5	0.3	0.3	**0.2**
book2	610856	9.6	7.6	14.1	8.9	**6.9**
bible.txt	4047392	13.9	83	136	**65**	70
E.coli	4638690	17.3	142	177	**84**	101
news	377109	18.1	12.9	10.1	6.4	**3.5**
world192.txt	2473400	22.9	70	80	**38**	**38**
progl	71646	24.6	3.0	1.0	0.6	**0.4**

in order of AML. Bentley-Sedgewick is fast if AML is small (geo, book1). Larsson and our algorithm are fast if AML is large. Larsson is much faster than our algorithm for E.coli because E.coli is a DNA sequence of a *colon bacillus* and linked lists in the tree is efficient for the small alphabet (ATCG). Our algorithm is fast if AML is large because it uses the doubling technique of the KMR algorithm. Roughly speaking, its sorting time is proportional to logarithm of the AML. On the other hand, sorting time of the Bentley-Sedgewick is proportional to the AML.

Table 3 shows experimental results for large files. We used Sun Ultra30 with 512MB memory. In the table, *yeast* is a DNA sequence, *KIJI13M* is a part of the text of *Nihon Keizai Shimbun*, which is a Japanese newspaper including headlines and bodies, and *gcc-2.7.2.3.tar* is an archive of gcc sources. Our algorithm is 2.3

to 8.7 times faster than the Bentley-Sedgewick and 2.1 to 3.0 times faster than the Manber-Myers. The Larsson is fast for small alphabets. If we use binary trees or hash tables for finding children of nodes in a suffix tree, the Larsson will become faster. However, it requires more memory.

The last row in the table shows the result for a very large file which is a part of bodies of the newspaper. The Manber-Myers and Larsson cannot be available due to memory limitation. Our algorithm is 1.79 times faster than the Bentley-Sedgewick.

Table 3: sorting time and average match length

files			sorting time (s)			
name	size (byte)	AML	BS	MM	Lar	ours
yeast	8720211	30.9	135	122	**52**	58
KIJI13M	13174078	32.8	193	199	104	**65**
gcc-2.7.2.3.tar	28231680	76.0	1390	478	253	**159**
KIJI50M	51200000	19.3	580	—	—	**323**

5 Concluding remarks

We examined speed and required memory of several algorithms for sorting suffixes and making a suffix tree and we proposed a practical algorithm for sorting suffixes by combining other algorithms. Our algorithm is faster than other sorting algorithms when average match length of suffixes is large. Its speed is compared with a suffix tree construction algorithm and it requires less than half of memory for the suffix tree. The average match length becomes large if a text is long, therefore our algorithm is practical for large texts.

Our algorithm is slower than the suffix tree construction algorithm if the size of alphabet is small. The reason is that the suffix tree uses linked lists for representing children of nodes. If we use binary trees or hash tables for storing children, the suffix tree construction algorithm will become faster. However, the tree requires huge memory. Our algorithm requires only $9n$ bytes for sorting a text of length n.

Though our algorithm is practically fast, the worst-case time complexity is $O(n \log^2 n)$ and it is worse than $O(n \log n)$ of Manber-Myers because our algorithm uses comparison-based sorting. As future works, we will therefore try to remove the comparison-based sorting like the algorithm of Manber and Myers.

The Block Sorting compression uses sorting of suffixes. However, sorting is not exactly a necessary condition. For example, when all preceding symbols x_{i-1} of suffixes S_i in a group are the same, sorting of the group is not required. We may accelerate compression speed of the Block Sorting by using the property.

Very recently, Hongo and Yokoo [7] proposed an algorithm for the Burrows-Wheeler transformation. Their algorithm uses the KMR algorithm and sorts only unsorted groups like ours. For sorting a group, it compares two numbers at a time.

Thus the number of iteration is less than $\lceil \log_3 n \rceil$ rather than $\lceil \log_2 n \rceil$. However, practically our algorithm is faster than theirs.

Acknowledgements

The author would like to thank Prof. Hiroshi Imai for his helpful comments.

References

[1] A. Andersson and S. Nilsson. A New Efficient Radix Sort. In *35th Symp. on Foundations of Computer Science*, pages 714–721, 1994.

[2] R. Arnold and T. Bell. A Corpus for the Evaluation of Lossless Compression Algorithms. In *Data Compression Conference*, pages 201–210, March 1997. http://corpus.canterbury.ac.nz/.

[3] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997. http://www.cs.princeton.edu/~rs/strings/.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithms. Technical Report 124, Digital SRC Research Report, 1994.

[5] Calgary Text Compression Corpus. ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/.

[6] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Commun.*, COM-32(4):396–402, April 1984.

[7] F. Hongo and H. Yokoo. Block-Sorting Data Compression and KMR Algorithm. In *20th Symposium on Information Theory and Its Applications*, pages 673–676. SITA, December 1997. (in Japanese).

[8] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. In *4th ACM Symposium on Theory of Computing*, pages 125–136, 1972.

[9] N. J. Larsson. Extended application of suffix trees to data compression. In *Data Compression Conference*, pages 190–199, April 1996.

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

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(12):262–272, 1976.

[12] J. Seward. bzip2, 1996. http://www.muraroa.demon.co.uk/.

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