International Journal of Pattern Recognition
and Artificial Intelligence
Vol. 30, No. 6 (2016) 1659018 (16 pages)

DOI: 10.1142/S0218001416590187

An Efficient Algorithm for Suffix Sorting

Zhan Peng*, Yuping Wang*,‡, Xingsi Xue† and
Jingxuan Wei*

*School of Computer Science and Technology
Xidian University
Xi'an, Shaanxi 710071, P. R. China

†School of Information Science and Engineering
Fujian University of Technology
Fuzhou, Fujian 350118, P. R. China

‡ywang@xidian.edu.cn

Received 2 September 2015
Accepted 12 February 2016
Published 14 April 2016

The Suffix Array (SA) is a fundamental data structure which is widely used in the applications such as string matching, text index and computation biology, etc. How to sort the suffixes of a string in lexicographical order is a primary problem in constructing SAs, and one of the widely used suffix sorting algorithms is *qsufsort*. However, *qsufsort* suffers one critical limitation that the order of suffixes starting with the same 2^k characters cannot be determined in the kth round. To this point, in our paper, an efficient suffix sorting algorithm called *dsufsort* is proposed by overcoming the drawback of the *qsufsort* algorithm. In particular, our proposal maintains the *depth* of each unsorted portion of SA, and sorts the suffixes based on the *depth* in each round. By this means, some suffixes that cannot be sorted by *qsufsort* in each round can be sorted now, as a result, more sorting results in current round can be utilized by the latter rounds and the total number of sorting rounds will be reduced, which means *dsufsort* is more efficient than *qsufsort*. The experimental results show the effectiveness of the proposed algorithm, especially for the text with high repetitions.

Keywords: Suffix sorting; suffix array; text index; computation biology.

1. Introduction

How to sort the suffixes of a given string in lexicographical order is a primary problem in constructing SAs. Currently, one of the widely used algorithms for this problem is the *qsufsort*[13] algorithm, which sorts the suffixes round by round until all the suffixes are in lexicographical order. In particular, the suffixes are initially sorted based on their first characters, then after each round, the checked prefix of each suffix is doubled and the suffixes are sorted according to a doubled number of characters. As a result, after the kth round, the suffixes are sorted according to their first 2^k

characters. It means that the order of suffixes whose first 2^k are the same cannot be determined after the kth round. Therefore, for suffixes with large LCP (*length* of the Longest Common Prefix), *qsufsort* needs several rounds to determine their order, which leads to a considerable time cost.

To overcome the drawback of *qsufsort*, in this paper, we present an efficient suffix sorting algorithm called *dsufsort* which is based on the *qsufsort* algorithm. Specifically, for each unsorted portion of SA (called an unsorted *bucket*), the *dsufsort* maintains the max known LCP of the suffixes in that bucket (called *depth* of the bucket), and update that depth timely during processing. Then, in each round, the suffixes in each unsorted bucket are sorted based on their own bucket depth. By this means, some suffixes can be sorted according to more than 2^k characters in the kth round, which means the suffixes starting with the same 2^k characters which cannot be sorted by the *qsufsort* may be sorted now. Since more suffixes can be sorted now in each round, more sorting results in current round can be utilized by the latter rounds, and hence the total number of sorting rounds will be reduced. Therefore, *dsufsort* is more efficient than *qsufsort*, especially for suffixes with large LCP.

The rest of this paper is organized as follows. In Sec. 2, we introduce the related work. In Sec. 3, we list some notations and terminologies used in this paper. The details of *qsufsort* and *dsufsort* are described in Sec. 4. In Sec. 5, we discuss some efficient implementation techniques for the proposed algorithm. Experimental results are shown in Sec. 6. Section 7 concludes the paper and proposes some open problems.

2. Related Work

During the past two decades, a large number of suffix array construction algorithms with different time and space complexities have been proposed. Next, a brief introduction is given on some of them, for more details, please see the surveys.[3,20]

Computing the SA from suffix trees is one of the most simplest methods. The limitation of this method is its high space and time requirement. Manber and Myers[14] present the first efficient algorithm with time complexity $O(n\log^n)$ to directly construct the suffix arrays. The algorithm uses a technique called *prefix doubling* which originates from Karp[10]. It sorts the suffixes initially by their first characters and then doubles the sorted prefix in each of the following rounds. Larsson and Sadakane[13] present an algorithm called *qsufsort* to improve Manber's algorithm. Unlike the Manber's algorithm that checks every part of SA (called a *bucket*) in each round, the *qsufsort* marks the buckets of SA that have been completely sorted in previous passes. Then in each pass, it skips the sorted buckets and sorts the unsorted buckets only. Although the *qsufsort* algorithm has the same asymptotic time complexity as that of Manber and Myers in theory, it is much faster in practice. Schurmann and Stoye[22] present an $O(n^2)$ method called *bpr*. Unlike the Manber's algorithm and *qsufsort* which use a *broad-first* strategy to sort the buckets round by round, the *bpr* algorithm adopts a *depth-first* sorting strategy: for each unsorted bucket, it recursively sorts the bucket untill all the suffixes in that bucket have been completely

sorted. Rajasekaran and Nicolae[21] proposed a new algorithm called *RadixSA* with time complexity $O(n\log^n)$. Different from the three algorithms introduced above which simply sort the buckets from left to right, the *RadixSA* sorts the buckets in a special order: suppose the ith suffix S_i resides in bucket B_i, and the algorithm sorts bucket B_n first, then storts $B_{n-1}, \ldots, B_1$ in order. This order ensures that after sorting B_i, S_i will be in its final location in SA.

Seward[23] presents another two suffixes sorting algorithms: *Copy* and *Cache*. They initially sort the suffixes according to their first two characters, then they sort the unsorted buckets from small to large, once a bucket is completely sorted, the sorting result can be used by future processing. However, the *Copy* and *Cache* sort all buckets with the same sort routine, and this is inefficient for suffixes that share a very long common prefix. To address this inefficiency, Manzini and Ferragina[15] presents an algorithm called *deep-shallow*. Whenever a bucket is being sorted, *deep-shallow* uses a *shallow* sorter for the suffixes with a short common prefix, and a *deep* sorter for the suffixes with a long common prefix. Although the *deep-shallow* has a good performance, its complicated framework limits its application in practice.

All the algorithms introduced above have a *super-linear* time complexity. However, there exist some algorithms that have a linear time complexity, among which *KA*,[12] *KS*[9] and *KSP*[11] are three notable ones. The *KSP* seems adopting a similar idea as Farach's algorithm[4] on suffix trees by using a very similar and complex merging step. The *KS* algorithm uses a *divide-and-conquer* approach which includes three steps: (1) recursively construct the SA for suffixes starting at positions i where $i \bmod 3 \neq 0$; (2) construct the SA of the remaining suffixes using the result of the first step; (3) merge the two SAs into one. The *KA* is an improvement of the *two-stage* algorithm.[7] It classifies all the suffixes in the string into two classes: L-type and S-type. Then it recursively sorts all the L-type suffixes, after that, the order of the S-type suffixes can be induced by the order of L-type suffixes. Nong[19] presents two algorithms *SA-IS* and *SA-DS* to improve *KA* by exploiting the variable-length leftmost S-type substrings and the fixed-length d-critical substrings for problem reduction, and the simple algorithms for sorting these sampled substrings. Recently, Nong[16] further presents another linear time algorithm called *SACA-K* for constant alphabets which only uses $O(1)$ workspace. Although the linear algorithms have good theoretical time complexity, but in practice they are usually not as fast as finely tuned super-linear algorithms for real world data.[21]

Currently, some external-memory algorithms[8,17,18] have been proposed for constructing large SAs, where the space needed by external memory algorithms is mainly supplied by low-cost massive disks. With the external-memory algorithms, the SAs which are too large to be accommodated in memory can be constructed now.

3. Preliminaries

Let Σ be an *alphabet* consists of a finite number of character symbols. (In this paper, we will mainly focus on the case that Σ is the ASCII character set, where $|\Sigma| = 256$

and each character requires only one byte of memory.) Given the alphabet Σ, a *string* as well as its *substring* over Σ can be defined as:

Definition 1. *A string T over Σ is a sequence consisting of a finite number of characters from Σ. Particularly, a string of length n over Σ is denoted by $T = t_0 t_1 .. t_{n-1}$, where $T[i] = t_i \in \Sigma$ $(0 \le i \le n-1)$. A substring of T is a sequence consisting of any consecutive characters of T. A substring of T which starts at position i and ends at position j is denoted by $T[i, j]$.*

The basic concepts in suffixes sorting are the *suffixes* and *prefixes*, which are special substrings of a given string:

Definition 2. *For a string $T = t_0 t_1 .. t_{n-1}$, the suffix of T that starts at position $i (0 \le i \le n-1)$ is $T[i, n-1]$, which is denoted by $S_i(T)$. The length-h prefix of T is $T[0, h-1]$, and it is denoted by $P_h(T)$.*

In this paper, whenever we refer to some suffixes, they are related to the *same* string, so the notation $S_i(T)$ can be abbreviated to S_i without ambiguity. To ensure that no suffix is a prefix of another suffix, a special terminal character '$' is always appended to the rear of T, and '$' is defined to be smaller than any character in Σ.

We use the notation $S_i \prec S_j$ to denote that S_i is *lexicographically smaller* than S_j. And our ultimate goal is to sort all the suffixes of a string in ascending lexicographical order to form a *suffix array*.

Definition 3. *For a string $T = t_0 t_1 \dots t_{n-1}\$$, the suffix array of T is an array $SA[0 \dots n]$ consisting of a permutation of the integers $0, 1, \dots, n$ such that for all $0 \le i < j \le n$, $S_{SA[i]} \prec S_{SA[j]}$, where $SA[i]$ is the ith element of the array SA.*

Since a suffix can be uniquely determined by its starting position, only the starting position numbers of the suffixes are stored in SA.

For simplicity, the term *order* (of suffixes) is used to indicate the *lexicographical order* unless explicitly stated. Based on the *lexicographical order*, we can further define the *h-order* of suffixes by just comparing their first h characters: S_i is said to be *h-smaller* than S_j if and only if $P_h(S_i) \prec P_h(S_j)$, and this relation is denoted by $S_i \prec_h S_j$. The notations $=_h$ and $\preceq_h$ can be defined in a similar way. Obviously, there is: $S_i \prec_h S_j \Rightarrow S_i \prec S_j$. If all the suffixes are sorted according to their first h characters, they are called *in h-order*.

Next we introduce the concept of *bucket* which is a key concept for construction of both *qsufsort* and *dsufsort*.

Definition 4. *A depth-h bucket of SA is a sub-array $SA[l \dots r]$ $(l \le r)$ such that: $S_{SA[l]} =_h S_{SA[l+1]} \cdots =_h S_{SA[r]}$, $S_{SA[l-1]} \neq_h S_{SA[l]}$ and $S_{SA[r]} \neq_h S_{SA[r+1]}$. The bucket number of a bucket $SA[l \dots r]$ is defined to be l and the bucket is denoted by B_l.*

Note that, h is the length of the common prefix of suffixes in B_l that we have *currently known*. In order to record the bucket of a given suffix, an array B is used: if $B[i] = j$, then S_i is currently in B_j. Note that, B_i is the bucket whose number is

i, while $B[i]$ is the number of the bucket in which S_i currently resides, and these two similar notations can be distinguished from the context without ambiguity.

For suffixes S_i and S_j, $\mathrm{LCP}(S_i, S_j)$ is defined to be the length of the longest common prefix of S_i and S_j. And based on $\mathrm{LCP}(S_i, S_j)$, the *average* LCP of a given string can be defined as follows:

Definition 5. *Given a string T of length $n+1$ and its array SA, the average* LCP *of T is defined to be*:

$$\frac{1}{n}\sum_{i=0}^{n-1}\mathrm{LCP}(S_{SA[i]}, S_{SA[i+1]}). \tag{1}$$

The *average* LCP is a rough measure of the difficulty of sorting the suffixes: if the *average* LCP is large, we need — in principle — to examine *many* characters to determine the order of two suffixes.

4. The *dsufsort* Algorithm

In this section, we describe our *dsufsort* algorithm, which is an improvement of the *qsufsort*, for sorting all the suffixes of a given string. The *dsufsort* improves the *qsufsort* by maintaining the *depth* for each unsorted bucket during processing. Once an unsorted bucket is being sorted, the suffixes are sorted based on the bucket depth. By this way, more suffixes can be completely sorted by *dsufsort* than by *qsufsort* in each round, in other words, the *dsufsort* needs less rounds to completely sort all the suffixes, which improves the performance. In the following subsections, a brief introduction is given to the original *qsufsort* algorithm first, and then the improvement –*dsufsort* is discussed in detail.

4.1. *The qsufsort algorithm*

Larsson and Sadakane's *qsufsort* algorithm uses a technique called *prefix doubling* to sort the suffixes. It works round by round. In round 0, all the suffixes of a given string will be sorted according to their first characters. The result is that the suffixes starting with the same character will be arranged together to form a *depth*-1 bucket in SA, and the whole SA is conceptually partitioned into a sequence of *depth*-1 buckets: the first bucket contains the suffixes starting with the smallest character, the second bucket contains the suffixes starting with the second-smallest character, and so on.

Note that, if a *depth*-1 bucket contains only a single suffix, the bucket as well as the single suffix are called *completely sorted* ones. Because that suffix can be uniquely distinguished from all others by its first character, and thus, it is already in its final location in SA and will not be sorted in the future. However, if a *depth*-1 bucket contains more than one suffix, the bucket as well as its suffixes are called *unsorted*

ones, and it is needed to check more than one character to determine the order of the suffixes in future rounds.

After round 0, all the suffixes are sorted in 1-order. And the 1-order of any two suffixes S_i and S_j can be determined by their bucket numbers: $S_i \preceq_1 S_j \Leftrightarrow B[i] \leq B[j]$, without loss of generality.

Using the prefix doubling technique, the suffixes are sorted according to a doubled number of the most-left characters after each round. As a result, after the $(k-1)$th round, all the suffixes are sorted in 2^{k-1} order, in other words, all the unsorted buckets left have the same depth of 2^{k-1}. And these unsorted buckets, will be sorted one by one, from left to right, in the kth round.

Now, assuming an unsorted bucket B_p is being sorted in the kth round. For any S_i and S_j in B_p, since $S_i =_{2^{k-1}} S_j$, the order of S_i and S_j will depend on the order of $S_{i+2^{k-1}}$ and $S_{j+2^{k-1}}$, that is $S_i \prec S_j \Leftrightarrow S_{i+2^{k-1}} \prec S_{j+2^{k-1}}$, without loss of generality. Here, $S_{i+2^{k-1}}(S_{j+2^{k-1}})$ is called the *anchor suffix* of $S_i(S_j)$. However, since only the 2^{k-1} order of $S_{i+2^{k-1}}$ and $S_{j+2^{k-1}}$ is known now, only the 2^k order of S_i and S_j can be determined: $S_i \preceq_{2^k} S_j \Leftrightarrow S_{i+2^{k-1}} \preceq_{2^{k-1}} S_{j+2^{k-1}} \Leftrightarrow B[i+2^{k-1}] \leq B[j+2^{k-1}]$. This derivation gives a way to sort the suffixes of B_p in 2^k order: for any S_i in the B_p, the value $B[i+2^{k-1}]$, which is the bucket number of S_i's anchor suffix, can be used as the key of S_i:key$(S_i) = B[i+2^{k-1}]$, and then all the keys of the suffixes are sorted first by a common integer sorting routine, after that the suffixes in B_p can be sorted in 2^k order by being simply rearranged according to the arithmetic order of their keys.

Once an unsorted bucket is sorted, new buckets are formed according to the following two cases: (1) If a suffix has a unique key, it will be in its final location in SA to form a singleton bucket which is completely sorted. (2) If some suffixes have the same key, they will be rearranged together and form a new unsorted bucket which will be sorted in future rounds. And the B array needs to be updated according to these new buckets.

The *qsufsort* algorithm runs round by round untill there is no unsorted bucket left. Note that, using the prefix doubling technique, the length of the checked prefix of each suffix is doubled after each round, so for any two suffixes of the input string, their order can be determined in at most $\log^n$ rounds, where n is the length of the input string. Therefore the *qsufsort* algorithm performs up to $\log^n$ rounds to completely sort all the suffixes.

4.2. *The dsufsort algorithm: maintaining the depth for each unsorted bucket*

As stated above, in the kth round, the *qsufsort* algorithm checks only the first 2^k characters of each suffix to determine their order, which means the order of the suffixes starting with the same 2^k characters cannot be determined in the kth round. Therefore for suffixes with a long common prefix, *qsufsort* needs several rounds to determine their order, which will lead to a great computation cost.

It is necessary to reduce the computation cost by checking as many characters of each suffix as possible to determine their order in each round. To achieve this purpose, for each unsorted bucket, the *dsufsort* algorithm maintains its *bucket depth* (length of the common prefix of its suffixes that we have currently know), and updates the bucket depth once an unsorted bucket is sorted. In each round, when sorting the suffixes in an unsorted bucket, the *dsufsort* uses the depth of the bucket to compute the keys of the suffixes and then sort them. Using the keys which are computed from the bucket depth, the suffixes of some unsorted buckets can be sorted according to more than the most-left 2^k characters in the kth round, as a result, some of the suffixes starting with the same 2^k characters can be completely sorted by *dsufsort* in the k-round. Therefore, the *dsufsort* can sort more suffixes than the *qsufsort* in each round, and it usually uses fewer sorting rounds.

In order to store the depth information, an array D is created. The depth of a bucket can be indexed by its bucket number: for a bucket B_p, its depth is recorded in $D[p]$. After round 0, we set $D[p] = 1$ for each unsorted bucket B_p.

Similar to *qsufsort*, the *dsufsort* algorithm works round by round until there is no unsorted bucket left. In each round, the *dsufsort* algorithm adopts a two phases *sorting–updating* strategy to sort each unsorted bucket and update their depths. As an example, suppose an unsorted bucket B_p is being sorted in the kth round.

1. **Sorting:** for any S_i and S_j in B_p, since $S_i =_{D[p]} S_j$, the order of S_i and S_j depends on the order of $S_{i+D[p]}$ and $S_{j+D[p]}$: $S_i \prec S_j \Leftrightarrow S_{i+D[p]} \prec S_{j+D[p]} \Leftrightarrow B[i + D[p]] < B[j + D[p]]$, without loss of generality. Here $S_{i+D[p]}(S_{j+D[p]})$ is the *anchor suffix* of $S_i(S_j)$. According to this relation, for any S_i in B_p, the *dsufsort* algorithm takes $B[i + D[p]]$ (rather than $B[i + 2^{k-1}]$ which is used in *qsufsort*) as its key, and then sorts the keys of suffixes with a common integer sorting algorithm. After that, the suffixes in B_p are rearranged according to their keys.

   After rearranging, for each suffix S_i in B_p, it is sorted according to the first $D[p] + D[B[i + D[p]]$ rather than 2^k characters. We will soon see that in the kth round(after the (k-1)th round), there must be $D[p] \geq 2^{k-1}$ and $D[B[i + D[p]] \geq 2^{k-1}$, which means the suffixes are sorted according to *at least* the first 2^k characters, thus the *dsufsort* can perform at least as good as *qsufsort*. And we will also see that in some case, there holds $D[p] > 2^{k-1}$, which means all the suffixes in B_p are sorted according to more than 2^k characters, and in this case the *dsufsort* algorithm performs better than *qsufsort*.
2. **Updating:** the D and B array will be updated immediately once B_p is sorted. Suppose the suffixes are rearranged as $S_{i_1}, S_{i_2}, \ldots, S_{i_s}$ according to their keys $\text{key}(S_{i_1}) \leq \text{key}(S_{i_2}) \leq \ldots \leq \text{key}(S_{i_s})$. Depending on whether or not the key of a suffix is unique, there are two kinds of suffixes:

   - For any S_{i_j} such that $\text{key}(S_{i_{j-1}}) \neq \text{key}(S_{i_j})$ and $\text{key}(S_{i_j}) \neq \text{key}(S_{i_{j+1}})$, S_{i_j} will be in its final location in SA and forms a completely sorted singleton bucket: B_{p+j-1}. The bucket of S_{i_j} is updated accordingly: $B[i_j]$ is updated to $p + j - 1$.

However, there is no need to update the depth (the D array) for completely sorted buckets.

- For each group of suffixes $\{S_{i_l}, S_{i_{l+1}}, \ldots, S_{i_r}\}$ such that: $\text{key}(S_{i_l}) = \text{key}(S_{i_{l+1}}) = \ldots = \text{key}(S_{i_r}) = m$, $\text{key}(S_{i_{l-1}}) \neq m$ and $\text{key}(S_{i_{r+1}}) \neq m$, the group $\{S_{i_l}, S_{i_{l+1}}, \ldots, S_{i_r}\}$ will form a new unsorted bucket: B_{p+l-1}. Since every suffix in B_{p+l-1} has its anchor suffix in B_m, B_m is called the *anchor bucket* of B_{p+l-1}. The bucket of each suffix in the group is updated accordingly: all of $B[i_l], B[i_{l+1}], \ldots, B[i_r]$ are updated to $p + l - 1$. Since we have already known that $S_{i_l} =_{D[p]} S_{i_{l+1}} =_{D[p]} \cdots =_{D[p]} S_{i_r}$ and $S_{i_l+D[p]} =_{D[m]} S_{i_{l+1}+D[p]} =_{D[m]} \cdots =_{D[m]} S_{r+D[p]}$, there holds $S_{i_l} = D[p] + D[m] S_{i_{l+1}} =_{D[p]+D[m]} \cdots =_{D[p]+D[m]} S_{i_r}$. Therefore, the depth of the newly created bucket B_{p+l-1} is $D[p + l - 1] = D[p] + D[m]$, and we will use this formula to update the depth for each newly created unsorted bucket.

Based on the two phases *sorting–updating* strategy, the framework of *dsufsort* can be summarized as below:

Step 1. Initialization (round 0): sort all the suffixes of a given string according to their first characters. For each unsorted bucket B_p, set $D[p] = 1$, and for any S_i in B_p, set $B[i] = p$.

Step 2. For each unsorted bucket (say B_p) left in SA, perform the following *sorting–updating* procedure:

(i) *Sorting*: for each suffix S_i in B_p, taking $B[i + D[p]]$ as its key, and sort the keys of suffixes by an integer sorting algorithm. Rearrange the suffixes in B_p according to the arithmetic order of their keys.

(ii) *Updating*: suppose the suffixes in B_p are rearranged as: $S_{i_1}, S_{i_2}, \ldots, S_{i_s}$. For any S_{i_j} such that: $\text{key}(S_{i_{j-1}}) \neq \text{key}(S_{i_j})$ and $\text{key}(S_{i_j}) \neq \text{key}(S_{i_{j+1}})$, set $\text{B}[i_j]$ to $p + j - 1$. For each group of suffixes $\{S_{i_l}, S_{i_{l+1}}, \ldots, S_{i_r}\}$ such that $\text{key}(S_{i_l}) = \text{key}(S_{i_{l+1}}) = \ldots = \text{key}(S_{i_r}) = m$, $\text{key}(S_{i_{l-1}}) \neq m$ and $\text{key}(S_{i_{r+1}}) \neq m$, create a new unsorted bucket B_{p+l-1} for that group. Set $D[p + l - 1]$ to $D[p] + D[m]$, and set all of $B[i_l], B[i_{l+1}], \ldots, B[i_r]$ to $p + l - 1$.

Step 3. If there are no unsorted buckets left in SA, stop and quit; otherwise, go to step 2 and start a new sorting round.

As stated before, the *dsufsort* algorithm has the following important feature:

Lemma 1. *For any unsorted bucket B_q created in the kth round $(k = 0, 1, 2, \ldots)$, there holds*: $D[q] \geq 2^k$.

Proof. This can be proved by induction. Basis: for any unsorted bucket B_q created in round 0, there holds $D[q] = 1 \geq 2^0$. Induction: suppose B_q is an unsorted bucket created from B_p in the kth round, and B_m is the anchor bucket of B_q. Since B_p and B_m are unsorted buckets created in the $(k-1)$th round, by induction, there holds $D[p] \geq 2^{k-1}$ and $D[m] \geq 2^{k-1}$, therefore $D[q] = D[p] + D[m] \geq 2^{k-1} + 2^{k-1} = 2^k$. □

This feature ensures that the *dsufsort* algorithm can perform at least as good as *qsufsort*. However it is necessary to explore in which case the *dsufsort* performs better, specially, in which case there is $D[q] > 2^k$. Suppose we are in the $(k-1)$th round, and sort an unsorted bucket B_p which is created in the $(k-2)$th round with $D[p] = 2^{k-2}$. If a new unsorted bucket B_q is created from B_p and its anchor bucket is B_m, there is $D[q] = D[p] + D[m]$.

The key point is that, if B_m, which is the anchor bucket of B_p, is also a new unsorted bucket that created previously in the $(k-1)$th round, due to the feature we have just proved, there must be $D[m] \geq 2^{k-1}$. So we have $D[q] = D[p] + D[m] \geq 2^{k-2} + 2^{k-1} > 2^{k-1}$, which means the depth of B_q is strictly larger than 2^{k-1}. Consequently, when B_q is processed in the kth round, any S_i in B_p is sorted according to its first $D[p] + D[B[i + D[p]]$ characters as stated before, since $D[q] > 2^{k-1}$ and $D[B[i + D[p]] \geq 2^{k-1}$, S_i is sorted according to more than 2^k characters in the kth round, which means the *dsufsort* algorithm performs better than *qsufsort*. Furthermore, if B_m is in the same case as B_q, that is, the anchor bucket of B_m is also a bucket created in the $(k-1)$-round, then there is $D[m] > 2^{k-1}$. Due to the *accumulation of the depths*, $D[q]$ can be much larger than 2^{k-1}, as a result, the suffixes in B_q can be sorted according to much more than 2^k characters in the kth round.

Example. Figure 1 shows the procedure of the *dsufsort* algorithm with the input string '*tobeornottobe*'. To show clearly, we put S_i in SA instead of its starting position i. The depth of a bucket is shown in the parentheses following the bucket. After round 0, all the suffixes are sorted according to their first characters. Since each of B_0, B_5 and B_{10} contains a single suffix, these three buckets are completely sorted.

In round 1, the four unsorted buckets B_1, B_3, B_6 and B_{11} left from round 0 are sorted in order. As an example, we sort B_6, and the other three buckets can be sorted

After round 0:

Buckets:	B_0	$B_1(1)$		$B_3(1)$		B_5	$B_6(1)$				B_{10}	$B_{11}(1)$		
SA:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	S_{13}	S_2	S_{11}	S_3	S_{12}	S_6	S_1	S_4	S_7	S_{10}	S_5	S_0	S_8	S_9
Keys:		3	3	6	0		1	10	11	1		6	11	6

After round 1:

Buckets:	B_0	$B_1(2)$		B_3	B_4	B_5	$B_6(3)$		B_8	B_9	B_{10}	$B_{11}(4)$		B_{13}
SA:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	S_{13}	S_2	S_{11}	S_{12}	S_3	S_6	S_1	S_{10}	S_4	S_7	S_5	S_0	S_9	S_8
Keys:		8	0				8	0				8	0	

After round 2:

SA:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	S_{13}	S_{11}	S_2	S_{12}	S_3	S_6	S_{10}	S_1	S_4	S_7	S_5	S_9	S_0	S_8

Fig. 1. An example run of the *dsufsort* algorithm with the input string '*tobeornottobe*'.

in the same way. For any S_i in B_6, its key can be computed from the bucket depth by $\text{key}(S_i) = B[i + D[6]]$. So the keys of suffixes of B_6 are: $\text{key}(S_1) = B[1 + D[6]] = 1$, $\text{key}(S_4) = B[4 + D[6]] = 10$, $\text{key}(S_7) = B[7 + D[6]] = 11$, $\text{key}(S_{10}) = B[10 + D[6]] = 1$. Then these keys are sorted by a regular integer sorting algorithm to have the order: $\text{key}(S_1) = \text{key}(S_{10}) < \text{key}(S_4) < \text{key}(S_7)$. According to the arithmetic order of their keys, the 1-order of the suffixes in B_6 can be determined: $S_1 =_1 S_{10} \prec_1 S_4 \prec_1 S_7$, and the suffixes are rearranged accordingly. Since S_1 and S_{10} have the same key, they will be grouped together in a newly created unsorted bucket B_6. Note that, compared with the *old* B_6 (denoted by B_6^{old}) which is being processed now, the newly created B_6 (denoted by B_6^{new}) has the same bucket number (which means the bucket numbers of S_1 and S_{10} need not to be updated), but different size and depth. The depth of B_6^{new} is $D[6]^{\text{new}} = D[6]^{\text{old}} + D[1] = 3$, because B_1 is the anchor bucket of B_6^{new} and $D[1] = 2$. On the other hand, S_4 and S_7, each of which has a unique key, will be in completely sorted buckets B_8 and B_9, respectively, and only get their bucket numbers updated: $B[4] = 8$, $B[7] = 9$. After round 1, there left three unsorted buckets: B_1^{new}, B_6^{new}, B_{11}^{new}.

In round 2, B_1^{new}, B_6^{new} and B_{11}^{new} will be sorted in order. Using $D[1] = 2$, $D[6] = 3$ and $D[11] = 4$ respectively to compute the keys of suffixes and sort them in the corresponding unsorted buckets, all these three buckets will be completely sorted in this round, which ends the whole algorithm.

To illustrate the advantage of *dsufsort* over *qsufsort*, we examine the sorting process of B_{11}^{old} in round 1. Since the keys of the suffixes in B_{11}^{old} are: $\text{key}(S_0) = 6$, $\text{key}(S_9) = 6$ and $\text{key}(S_8) = 11$, S_8 is completed sorted in B_{13}, while S_0 and S_9 will be in unsorted bucket B_{11}^{new}. Since B_{11}^{new} is created from B_{11}^{old}, and its anchor bucket is B_6^{new}, the depth of B_{11}^{new} is $D[11]^{\text{new}} = D[11]^{\text{old}} + D[6]^{\text{new}}$. As we have seen, B_6^{new} is also created in round 1 with $D[6]^{\text{new}} = 3$, so $D[11]^{\text{new}} = 1 + 3 = 4$. Therefore, the depth of B_{11}^{new} is larger than 2 which is the "depth" of all buckets computed by *qsufsort* in round 1. Consequently in round 2, the suffixes contained in B_{11}^{new} are sorted according to at least their first $4 + 2$ characters instead of $2 + 2$ characters as in the *qsufsort*. The keys of suffixes are: $\text{key}(S_0) = B[0 + D[11]^{\text{new}}] = 8$ and $\text{key}(S_9) = B[9 + D[11]^{\text{new}}] = 13$, which means B_{11} can be completely sorted in round 2. By contrast, in the *qsufsort*, the keys of suffixes in B_{11}^{old} are $\text{key}(S_0) = B[0 + 2] = 1$ and $\text{key}(S_9) = B[9 + 2] = 1$ which means B_{11}^{old} cannot be completely sorted after the second round. The result is that *dsufsort* algorithm only needs 3 rounds to completely sort all the suffixes, while *qsufsort* needs 4 rounds.

For the input string with a large *average* LCP, the feature of the *depth accumulations* of *dsufsort* can be fully used, and the order of the suffixes with very long common prefix can be determined much faster.

5. Efficient Implementation

In this section, we describe some techniques used to derive efficient implementation of the proposed algorithm.

5.1. *Input transformation*

In round 0, the suffixes are sorted only according to their first characters. Actually, they can be sorted according to the first *few* characters by using a technique called *input transformation* to the input string in advance. The *input transformation* includes the following two phases:

5.1.1. *Alphabet compaction*

Given a text string $T = t_0 t_1 \dots t_{n-1}\$$ over Σ, suppose the characters that actually appear in T form an ordered set $C = \{c_0, c_1, \dots, c_{m-1}\}$, in which $c_i < c_j \Leftrightarrow i < j$. Note that, the smallest terminal character '\$' must be c_0. Then, for each character in T, we can replace that character by its corresponding ordinal number in C, that is: $t_i \mapsto j \Leftrightarrow t_i = c_j$. By using this mapping, each character of T is encoded as an integer, and the order of suffixes is preserved. With the alphabet compaction, Σ is transformed into a smaller integer alphabet: $\{0, 1, \dots, m-1\}$.

5.1.2. *Characters aggregation*

After applying the alphabet compaction to T, we get a new alphabet of size m: $\{0, 1, \dots, m-1\}$. Let k be the largest integer such that $m^k - 1$ can be held in a typical machine integer. Then, for each suffix of T, we can aggregate its first k characters into one using the following formula:

$$T[i] = \sum_{j=1}^{k} t_{i+j-1} \cdot m^{k-j} \quad (0 \le i \le n), \tag{2}$$

where we define $t_s = 0$, for $s \ge n$. In round 0, $T[i]$ is used as the key for S_i, thus, sorting is based on not only the first character of each suffix, but also the first k characters. The subsequent rounds of sorting can start with bucket depth k instead of 1, and the number of rounds will be reduced. Formula (2) can be computed in linear time through an alternate form:

$$T[i] = \begin{cases} \displaystyle\sum_{j=1}^{k} (t_{j-1} \cdot m^{k-j}), & i = 0, \\ (T[i-1] \bmod m^{k-1}) \cdot m + t_{i-1+k}, & 0 < i \le n, \end{cases} \tag{3}$$

where $t_s = 0$, for $s \ge n$. The multiplication and modulo operations can be replaced by faster bitwise operations *shift* and *and*.

Note that, since the alphabet of T may not be consecutive after characters aggregation, we need to use alphabet compaction again to the new aggregated string T.

5.2. *Initial bucket sorting*

The round 0 (initialization) of the algorithm is quite independent to the rest of the algorithm and is not required to use the same sorting method as the following rounds.

Since this step must process all of the suffixes in a single round, a substantial improvement can be gained by using a linear time bucket sorting algorithm instead of a comparison-based algorithm that requires $O(n\log^n)$ time.

A reasonable improvement can be gained by combining bucket sorting with input transformation as described above. Given a string $T = t_0 t_1 \dots t_{n-1}\$$, suppose the new alphabet after applying input transformation to T is $I = \{0, 1, \dots, m-1\}$. For each integer i in I, its number of occurrences in T is counted and an array F of size m is used to record that number. In particular, if integer i occurs j times in T, then $F[i] = j$, and based on this definition, there is $\sum_{i=0}^{m-1} F[i] = n+1$. To conclude, the round 0 of the *dsufsort* algorithm using bucket sorting includes the following four steps:

1. Initialize F: $\forall i \in I$, set $F[i] = 0$.
2. Scan T forwards to compute the occurrence frequency of its symbols: for $i = 0, \dots, n$, increment $F[T[i]]$ by 1.
3. Traverse F forwards, and sum adjacent elements so as to form cumulative frequency counts: for $i = 1, \dots, m-1$, set $F[i] = F[i] + F[i-1]$.
4. Scan T backwards to put each of its suffix in the proper bucket: for $i = n, n-1, \dots, 0$, decrement $F[T[i]]$ by 1, and set $SA[F[T[i]]] = i$.

After round 0, SA is partitioned into m (completely sorted or unsorted) buckets, and all suffixes in an unsorted bucket start with the same k characters. In essence, the term *bucket* in the *bucket sort* here is exactly the depth-k bucket defined by Definition 5 in the Preliminaries.

5.3. *Choosing an integer sorting subroutine*

Both of the *qsufsort* and the *dsufsort* algorithm need an integer sorting subroutine to sort the keys, thus, this subsidiary subroutine may have a great effect on the performance of the whole algorithms. The sorting subroutine used in this paper is called *split-end partitioning* which is proposed by Bentley and Mcllroy.[1] It is a variant of the well-known *Quicksort*,[5] but uses a *ternary-split partition* strategy.

The classical *Quicksort* which uses a *binary-split partition* strategy recursively partitions an array into *two* parts, one with smaller elements than a pivot element and one with larger elements. Then the parts are processed recursively until the whole array is sorted. The *Quicksort* mixes the elements being equal to the pivot into one or both of the parts depending on the implementation. However, the *split-end partitioning* algorithm which uses a ternary-split partition strategy generates *three* parts: one with elements smaller than the pivot, one with elements equal to the pivot, and one with elements larger than the pivot. The smaller and larger parts are then processed recursively while the equal part is remained, since its elements are already correctly placed.

The implementation of *split-end partitioning* in our algorithm is based on Program 7 of Bentley and Mcllroy[1] with one exception: for fast handling of small

buckets, a variant of selection sort which is nonrecursive is used to sort buckets with less than seven elements.

6. Experimental Results

We compared the *dsufsort* algorithm to three well-known algorithms, i.e. the original *qsufsort* algorithm,[13] the *DC32*[2] algorithm, and the *KS* algorithm.[9] Our *dsufsort* algorithm is an improvement of *qsufsort* algorithm. The *DC32* is the *difference-cover* algorithm with difference-cover modulo 32. The *KS* algorithm is one of the fastest suffixes sorting algorithms of a worst-case linear time complexity. We evaluated the performance of these algorithms for real world data and for degenerated data: artificial strings with very high *average* LCP.

The experiments were performed on a notebook running Ubuntu 14.04-64bit operating system with the following configuration: Intel Core i7-2630QM 2.00GHz processor, 8GB 1333Mhz DDR3 SDRAM, 500GB SATA disk. All the programs were implemented in C/C++, and compiled by *gcc* 4.8.2 with flags"–O3". To ensure the correctness of the testing algorithms, all the sorting results are checked by a *suffix array checker* which is described in Burkhardt.[2]

The real world data set we used in the experiment is the *Pizza Chili Corpus*[6] which contains six kinds of data: source code, pitch values, protein sequence, DNA sequence, English text and XML files. The characteristic of the data set is shown in Table 1 which shows the size, the average and maximum LCP lengthes for each file. The average/maximum LCP for a string is the average/maximum ones for all pairs of adjacent suffixes in the suffix array. The maximum LCP length is equivalent to the length of the longest repeated substring. These values give a good estimate of the repetitiveness of the data.

Table 2 shows the mean sorting time of each testing algorithm based on 10 independent runs, in which the best run times among all compared algorithms are shown in bold. We also use artificial repetitive files to test the robustness of the testing algorithms. The first two files contain solely the letter a and the pair ab, respectively, and the following three files that have the form *'rand-k-rep'* are generated by a single random *seed* string of length k which is repeated until 100MB characters are reached.

Table 1. Data sets used in the experiment.

Files	Description	Size(bytes)	$\|\Sigma\|$	Average LCP	Max LCP
Proteins	Protein sequence	66,804,271	24	33.46	6380
XML	XML files	294,724,056	97	44.91	1084
Pitches	MIDI pitch values	55,832,855	133	262.00	25,178
Sources	C/Java source code	210,866,607	230	371.80	307,871
DNA	DNA sequence	403,927,746	16	2420.73	1,378,596
English	English text	2,210,395,553	235	6675.35	987,770

Table 2. Sorting times in seconds.

Files	Size	Dsufsort	Qsufsort	DC32	KS
Proteins	100MB	25.34	**24.11**	35.43	98.91
XML	100MB	27.59	**26.75**	49.54	67.39
Pitches	50MB	**9.27**	10.52	12.42	32.83
Sources	100MB	**23.81**	25.64	33.03	83.03
DNA	100MB	**26.02**	28.90	38.15	85.44
English	100MB	**41.72**	44.35	48.20	97.12
aaa…	100MB	**9.14**	10.65	73.32	11.87
abab…	100MB	**8.82**	11.55	30.23	9.56
rand-5-rep	100MB	**10.36**	16.32	35.60	12.77
rand-10-rep	100MB	**16.73**	24.28	33.57	17.53
rand-20-rep	100MB	23.11	39.03	35.92	**22.85**

From the results, we can see that, for real world data, the *dsufsort* and the *qsufsort* perform best, and the *dsufsort* performs better than *qsufsort* on most real world data. Only for Proteins and XML files which have very low *average* LCP, *dsufsort* performs a little bit worse than *qsufsort*. The reason is that, compared with *qsufsort*, the *dsufsort* needs to read and update the D array during the processing, and these operations require additional random memory accesses which will lead to additional cache misses. So the time cost will be over the one we saved from the *depth accumulation* which is not very effective for strings with very low *average* LCP.

For artificial data whose *average* LCP is relatively high, the *dsufsort* can dramatically reduce the time for sorting suffixes with long LCP. The results in Table 2 also show that *dsufsort* dominates others except for *KS* on one file (rand-20-rep) where the liner time algorithm *KS* is faster. This is because the performance of linear time algorithm is not severely affected by the characteristic of input data. However, the *KS* algorithm performs not well for ordinary data, and this limits its application in practice.

In summary, we can draw the conclusion that *dsufsort* algorithm outperforms the others in most cases. Only for case with very low *average* LCP, *dsufsort* algorithm does not outperform *qsufsort*, and for case with very high *average* LCP, it does not outperform *KS* algorithm sometimes.

7. Conclusions and Future Work

In this paper, an efficient suffixes sorting algorithm, i.e. *dsufsort*, is proposed, which is based on the framework of *qsufsort*. Unlike the *qsufsort* sort suffixes according to a fixed number of characters in each round, *dsufsort* maintains the depth for each unsorted bucket, and sort the bucket based its depth, furthermore, the *accumulation of depths* during processing makes *dsufsort* very efficient, especially for strings with large *average* LCP.

There are two problems left for further research: Firstly, in our implementation, the unsorted buckets are simply processed one by one from left to right. However, the

order in which the buckets are processed can have an effect on the performance of algorithm, so what is the best order from which the buckets should be processed? Secondly, the D array used to record the depth information may be very sparse, which leads to unnecessary excessive memory consumption. How to compact the D array to reduce the space complexity?

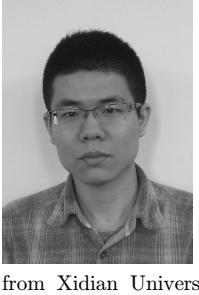
Acknowledgments

This work is supported by National Natural Science Foundation of China (No. 61472297, No. 61203372 and No. U1404622).

References

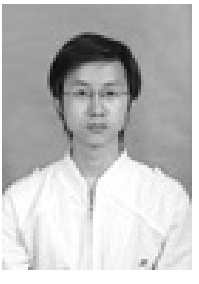
1. J. L. Bentley and M. D. Mcllroy, Engineering a sort function, *Softw. Pract. Exp.* **23** (1993) 1249–1265.
2. S. Burkhardt and J. Karkkainen, Fast lightweight suffix array construction and checking, in *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, Morelia, Michoacan, Mexico, June 2003, pp. 55–69.
3. J. Dhaliwal, S. J. Puglisi and A. Turpin, Trends in suffix sorting: A survey of low memory algorithms, in *Proc. 12nd. Australasian Computer Science Conf.*, Darlinghurst, Australia (2012), pp. 91–98.
4. M. Farach, Optimal suffix tree construction with large alphabets, in *Proc. 38th Ann. Symp. Foundations of Computer Science*, Miami Beach, FL, October 1997, pp. 137–143.
5. C. A. R. Hoare, Quickfort, *Comput. J.* **5** (1962) 10–15.
6. http://pizzachil.dcc.uchile.cl/.
7. H. Itoh and H. Tanaka, An efficient method for in memory construction of suffix arrays, in *Proc. 2nd. Ann. Symp. String Processing and Information Retrieval Sym Int. Workshop on Groupware* Cancun, Mexico, September 1999, pp. 34–42.
8. J. Karkkainen and D. Kempa, Engineering a lightweight external memory suffix array construction algorithm, in *Proc. 2nd. Int. Conf. Algorithms for Big Data*, Palermo, Italy, April 2014, pp. 7–9.
9. J. Karkkainen, P. Sanders and S. Burkhardt, Linear work suffix array construction, *J. ACM* **53** (2006) 918–936.
10. R. M. Karp, R. E. Miller and A. L. Rosenberg, Rapid identification of repeated patterns in strings, trees and arrays, in *Proc. 4th Ann. Theory of Computing* (ACM Press, New York, 1972), pp. 125–136.
11. D. K. Kim, I. S. Sim, H. Park and K. Park, Consructing suffix arrays in linear time, *J. Discrete Algorithms* **3** (2005) 126–142.
12. P. Ko and S. Aluru, Space efficient linear time construction of suffix arrays, *J. Discrete Algoritms* **3** (2005) 143–156.
13. N. Larsson and K. Sadakane, Faster suffix sorting, *Theor. Comput. Sci.* **387** (2007) 258–272.
14. U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Comput.* **22** (1993) 935–948.
15. G. Manzini and P. Ferragina, Engineering a lightweight suffix array construction algorithm, *Algorithmica* **40** (2004) 33–50.
16. G. Nong, Practical linear-time O(1)-workspace suffix sorting for constant alphabets, *ACM Trans. Inf Syst* **31** (2013) 15:1–15:15.

17. G. Nong, W. H. Chan, S. Q. Hu and Y. Wu, Induced sorting suffixes in external memory, *ACM Trans. Inf. Syst.* **33** (2015) 12:1–12:15.
18. G. Nong, W. H. Chan, S. Zhang and X. F. Guan, Suffix array construction in external memory using D-critical substrings, *ACM Trans. Inf. Syst.* **32** (2014) 1:1–1:15.
19. G. Nong, S. Zhang and W. H. Chan, Two efficient algorithms for linear time suffix array construction, *IEEE Trans. Comput.* **60** (2011) 1471–1484.
20. S. J. Puglisi, W. F. Smyth and A. H. Turpin, A taxonomy of suffix array construction algorithms, *ACM Comput. Surv.* **39** (2007) 1–31.
21. S. Rajasekaran and M. Nicolae, An elegant algorithm for the construction of suffix arrays, *J. Discrete Algorithms* **27** (2014) 21–28.
22. K.-B. Schurmann and J. Stoye, An incomplex algorithm for fast suffix array construction, *Softw. Pract. Exp.* **37** (2007) 309–329.
23. J. Seward, On the performance of BWT sorting algorithms, *DCC: Data Compression Conf.* (IEEE Computer Society Press, Los Alamitos, CA, 2000), pp. 173–182.



Zhan Peng is currently a Ph.D. candidate at Xidian University, majoring in Computer Science and Technology. He received his B.S. degree in Software Engineering from Xidian University in 2008 and his M.S. degree in Computer Science and Technology from Xidian University in 2011. His research interests include string pattern matching and text indexing.

Yuping Wang is a Professor with the School of Computer Science and Technology, Xidian University, Xi'an, China. He received his Ph.D. from the Department of Mathematics, Xi'an Jiaotong University, China in 1993. He is a Senior member of IEEE, and visited Chinese University of Hong Kong, City University of Hong Kong, and Hong Kong Baptist University as a research fellow many times from 1997 to 2010. He has authored or co-authored over 100 research papers in journals and conferences. His current research interests include evolutionary computation, optimization methods, data mining and scheduling, etc.



Xingsi Xue is a Lecturer with the School of Information Science and Engineering, Fujian University of Technology, Fuzhou, Fujian, China. Currently, his research interests include intelligent computation, ontology matching technology and intelligent decision supporting system.

Jingxuan Wei is an Associate Professor with the School of Computer Science and Technology, Xidian University, Xi'an, China. She received her B.S. degree in Applied Mathematics from Shanxi Normal University in 2003. She received her M.S. and Ph.D. degrees in Applied Mathematics from Xidian University in 2006 and 2009, respectively. Her research interests include computational intelligence, evolutionary computation and particle swarm optimization, etc.