Linear Time Suffix Array Construction Using D-Critical Substrings

Ge Nong[1]*, Sen Zhang[2], and Wai Hong Chan[3]**

[1] Computer Science Department, Sun Yat-Sen University, P.R.C., `issng@mail.sysu.edu.cn`

[2] Dept. of Math., Comp. Sci. and Stat., SUNY College at Oneonta, U.S.A., `zhangs@oneonta.edu`

[3] Department of Mathematics, Hong Kong Baptist University, Hong Kong, `dchan@hkbu.edu.hk`

Abstract. In this paper we present in detail a new efficient linear time and space suffix array construction algorithm(SACA), called the D-Critical-Substring algorithm. The algorithm is built upon a novel concept called fixed-size D-Critical-Substrings, which allow us to compute suffix arrays through a balanced combination of the bucket-sort and the induction sort. The D-Critical-Substring algorithm is very simple, a fully-functioning sample implementation of which in C++ is embodied in only about 100 effective lines. The results of the experiment that we conducted on the data from the Canterbury and Manzini-Ferragina corpora indicate that our algorithm outperforms the two previously best-known linear time algorithms: the Kärkkäinen-Sanders (KS) and the Ko-Aluru (KA) algorithms.

1 Introduction

Background

Suffix arrays were first proposed by Manber and Myers in their seminal SODA'90 paper [1] as a memory efficient alternative data structure to suffix trees. While a suffix tree physically contains all the suffixes of the text, a suffix array is only an integer array consisting of specially arranged indexes which references all the suffixes of the text in their lexicographically ascending order. Due to their light-weighted nature, suffix arrays have been used in large-scale indexing, compressing and pattern matching applications, e.g., internet-scale searching and genome sequence databases, where the magnitudes of data are measured often in billions of characters [2, 3]. The original suffix array construction algorithm introduced in [1] is superlinear, it, however, has well stimulated several attempts in designing linear time suffix array construction algorithms during the past two decades. Among them are three well-known linear SACAs contemporarily reported in

* Nong was partially supported by the National Science Foundation of P.R.C. (Grant No. 60873056).

** Chan was partially supported by the Faculty Research Grant (FRG/07-08/II-30).

2003. They are attributed to Kim, Sim, Park and Park (KSP) [4], Kärkkäinen and Sanders (KS) [5], and Ko and Aluru (KA) [6] respectively. While KSP is an array version of Farach's work [7] on suffix trees, the KA and KS algorithms share a same recursive framework but employ two different block compressing schemes and two different sorting methods: the KA algorithm compresses the text using varying length substrings at induction sorting friendly positions, but the KS algorithm compresses the text using fixed size substrings without utilizing induction sort at all. Yuta Mori has independently evaluated the KS and KA algorithms [8] to demonstrate that the KA is over one time faster than the KS. However, the implementation of the KS algorithm is dramatically simpler than that of the KA. This is well evidenced by the sample implementations of the two algorithms from their respective original inventors (The KS can be coded using about 100 lines of C code while the KA uses much more than 1000 lines). The reason is that the fixed size substrings used in the KS algorithm can be easily sorted by the bucket-sort, while the KA has to use very involved logics to maintain S-list in its crucial substring renaming phase.

Our Contributions

We proposed a new algorithm to combine the benefits of both fixed size substrings and the induction sort. The algorithm is called the D-Critical Substring algorithm and has been outlined in our poster at DCC'08 [9]. In this paper, we further extend it by including a detailed analysis and evaluation of the D-critical substring algorithm. The key idea of our D-Critical algorithm is to compress the original string using fixed D-Critical substrings (hence the name of the algorithm), which in turn are defined based on D-Critical characters. As will be seen in the following sections, D-critical characters are induction sort friendly and D-critical-substrings are fixed size. As a result, this new algorithm outperforms both the KA and the KS algorithms.

2 Preliminary

Unless explicitly specified otherwise, the following notations will be used throughout the paper.

- S: a string with n characters consecutively arranged as an array with the index starting from 0.
- $S[i..j]$: the consecutive substring starting at ith position and ending at jth position of S, where $i \leq j$.
- \$: The size-n input string S is terminated by a sentinel \$, which is the unique lexicographically smallest character in S.
- $suf(S, i)$: the suffix in S starting at $S[i]$ and running into the last character.
- $\Sigma(S)$ and $SA(S)$: the alphabet and the suffix array of S, respectively.
- Type-S or type-L suffix: a suffix $suf(S, i)$ is of type-S (type-L respectively) if $suf(S, i) < suf(S, i+1)$ ($suf(S, i) > suf(S, i+1)$ respectively). The last suffix $suf(S, n-1)$ consisting of only the single character \$ (the sentinel) is specifically defined as type-S.

- Type-S or type-L character: a character $S[i]$ is of type-S or type-L if $suf(S,i)$ is type-S or type-L, respectively.
- Leftmost type-S (LMS) character: $S[i]$ is a LMS character if $S[i]$ is type-S and $S[i-1]$ is type-L, where $i \in (0, n-1]$.
- Leftmost type-S (LMS) suffix: $suf(S,i)$ is a LMS suffix if $S[i]$ is a LMS character.
- t: let $t(S,i)$ be the LS-type function of $S[i]$, defined as $t(S,i) = 1$ for $S[i]$ is type-S or else $t(S,i) = 0$. For simplicity, $t(S,i)$ is also denoted as $t[i]$.
- ω: let $\omega(S,i)$ be the ω-weighting function of $S[i]$, defined as $\omega(S,i) = 2S[i] + t[i]$.
- S_ω: the ω-weighted string of S, where $S_\omega[i] = \omega(S,i)$.

3 D-Critical Substring algorithm

3.1 Algorithm Framework

Our proposed D-Critical substring algorithm, which we simply refer to as the DCS algorithm, is outlined in Fig. 1. Lines 1-3 first produce the reduced problem, which is then solved recursively by Lines 4-8. From the solution of the reduced problem, Line 9 induces the final solution for the original problem. The time and space bottleneck of this algorithm resides at reducing the problem in Lines 1-3, which is $O(n)$, achieved by using D-Critical substrings to re-represent the original text.

3.2 Critical Characters

Definition 1. *(Critical Character) A character $S[i]$ is D-Critical, where $d \geq 2$, iif (1) $S[i]$ is a LMS character; or else (2) $S[i-d]$ is a D-Critical character, $S[i+1]$ is not a LMS character and no character in $S[i-d+1..i-1]$ is D-Critical.*

Definition 2. *(Neighboring Critical Characters) A pair of D-Critical characters $S[i]$ and $S[j]$ are said to be two neighboring D-Critical characters in S, if there is no other D-Critical characters between them.*

Definition 3. *(Critical Substring) The substring $S[i..i+d+1]$ is a D-Critical substring for the D-Critical character $S[i]$ in S. For $i \geq n-(d+1)$, $S[i..i+d+1] = S[i..n-2]\{S[n-1]\}^{d+1-(n-2-i)}$, where $\{S[n-1]\}^x$ denotes that $S[n-1]$ is repeated x times.*

From the above definitions, we have the following immediate observations.

Proposition 1. *In S, (1) all LMS characters are D-Critical characters, (2) the last character must be a D-Critical character, and (3) the first character must not be a D-Critical character.*

Proposition 2. *If $S[i]$ is a D-Critical character, neither $S[i-1]$ nor $S[i+1]$ is a D-Critical character.*

Lemma 1. *The distance between any two neighboring D-Critical characters $S[i]$ and $S[j]$ in S must be in $[2, d+1]$, i.e., $j - i \in [2, d+1]$, where $d \geq 2$ and $i < j$.*

Proof. From Proposition 2, given $S[i]$ is a D-Critical character, $S[i+1]$ must not be a D-Critical character. In other words, the first d-character on the right hand of $S[i]$ may be any in $S[i+2, i+d+1]$, but must not be $S[i+1]$.

```
DCS-Sort($S, SA$)
      ▷ $S$ is the input string;
      ▷ $SA$ is the output suffix array of $S$;
      $P_1$, $S_1$: array $[0..n_1]$ of integer;
      $bkt$: array $[0..\|\Sigma(S)\| - 1]$ of integer;
 1    Find the sample pointer array $P_1$ for all the fixed-size D-Critical substrings in $S$;
 2    Bucket sort all the fixed-size D-Critical substrings in $P_1$;
 3    Name each D-Critical substring in $S$ by its bucket index to get
      a new shortened string $S_1$;
 4    if $\|S_1\|$ = Number of Buckets
 5        then
 6                Directly compute $SA_1$ from $S_1$;
 7        else
 8                DCS-Sort($S_1, SA_1$);
 9    Induce $SA$ from $SA_1$;
10    return
```

Fig. 1. Our DCS algorithm.

3.3 Reducing the Problem

To simplify the discussion, we use $\Psi_{C-d}(S)$ to denote the D-Critical substring array for S, which contains all the D-Critical substrings in S, one substring per item, consecutively arranged according to their original positional order in S.

Definition 4. *(Sample Pointer Array) The array P_1 contains the sample pointers for all the D-Critical substrings in S preserving their original positional order, i.e. $S[P_1[i]..P_1[i] + d + 1]$ is a D-Critical substring.*

From the definitions of P_1 and Ψ_{C-d}, immediately we can conclude $\Psi_{C-d} = \{S[P_1[i]..P_1[i] + d + 1] | i \in [0, n_1)\}$, where n_1 denotes the size (or cardinality) of Ψ_{C-d}. Hereafter, we simply consider P_1 at pointer level, but the underneath comparisons for its items lie in the substrings in Ψ_{C-d}. To compute P_1 from S, we need to know the LS-type of each character in S. This can be done by scanning S once from right to left in $O(n)$ time, by utilizing these properties: (i) $S[i]$ is type-S if (i.1) $S[i] < S[i+1]$ or (i.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-S; and (ii) $S[i]$ is type-L if (ii.1) $S[i] > S[i+1]$ or (ii.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-L. Provided with the LS-type of each character is known, we can traverse S once from right to left to compute P_1 in $O(n)$ time.

Definition 5. *(Siblings) $P_1[i]$ and $S[P_1[i]..P_1[i]+d+1]$ are said as a pair of siblings.*

Let's bucket sort all the items of P_1 by their ω-weighted siblings (i.e. $S_\omega[P_1[i]..P_1[i]+d+1]$ for $P_1[i]$) in increasing order, where all the buckets are indexed from 0. Then we can replace each item of P_1 with the index of its bucket to obtain a new string S_1. Notice this process is equivalent to a 1-to-1 mapping relationship between a D-Critical string and a unique integer. It is the so-called renaming (or simply naming) step. Here, we have the following observations on S_1.

Lemma 2. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From Proposition 1, we know that $S[n-1]$ must be a D-Critical character and the D-Critical substring starting at $S[n-1]$ must be the unique smallest among all sampled by P_1.

Lemma 3. *(1/2 Reduction Ratio) $\|S_1\|$ is at most half of $\|S\|$, i.e. $n_1 \leq \lfloor n/2 \rfloor$.*

Proof. From Proposition 1, $S[0]$ must not be a D-Critical character. We know from Lemma 1 the distance between any two neighboring D-Critical character is at least 2, which immediately completes the proof.

The above two lemmas state that, S_1 is at least half smaller than S and terminated by an unique smallest sentinel too.

Theorem 1. *(Coverage) For any two characters $S_1[i] = S_1[j]$, there must be $P_1[i+1] - P_1[i] = P_1[j+1] - P[j] \in [2, d+1]$, where $d \geq 2$, i.e., $P_1[i+1]$ and $P_1[i]$ are at distance no more than $d+1$, and so do $P_1[j+1]$ and $P_1[j]$.*

Proof. Given $S_1[i] = S_1[j]$, from the definition of S_1, there must be (1) $S[P_1[i]..P_1[i+1]] = S[P_1[j]..P_1[j+1]]$ and (2) $t[P_1[i]..P_1[i+1]] = t[P_1[j]..P_1[j+1]]$. Given (1) and (2) are satisfied, let $i' = P_1[i]+1$ and $j' = P_1[j]+1$, we have the below observations:

- Any in $S[i'..i'+d+1]$ is a LMS character. In this case, given $S_1[i] = S_1[j]$, we must have $P_1[i+1] = P_1[j+1]$.
- None in $S[i'..i'+d+1]$ is a LMS character. In this case, both $i'+d$ and $j'+d$ must be in P_1.

In either case, we have $P_1[i+1] - P_1[i] = P_1[j+1] - P[j]$.

Theorem 2. *(Order Preservation) The relative order of any two suffixes $suf(S_1, i)$ and $suf(S_1, j)$ in S_1 is the same as that of $suf(S, P_1[i])$ and $suf(S, P_1[j])$ in S.*

Proof. The proof is due to the following consideration on two cases:

- Case 1: $S_1[i] \neq S_1[j]$. This case can be further split into two cases in respect to whether the two critical substrings in S starting at $S[P_1[i]]$ and $S[P_1[j]]$ are equal or not. If they are different, the statement is obviously correct. If they are identical, we must have $t[P_1[i]+d+1] \neq t[P_1[j]+d+1]$ (or else we must have $S_1[i] = S_1[j]$), which implies that the statement is correct too.
- Case 2: $S_1[i] = S_1[j]$. In this case, the order of $suf(S_1, i)$ and $suf(S_1, j)$ is determined by the order of $suf(S_1, i+1)$ and $suf(S_1, j+1)$. The same argument can be recursively conducted on $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2]$,...$S_1[i+k-1] = S_1[j+k-1]$ until a k is reached that makes $S_1[i+k] \neq S_1[j+k]$. Because that $S_1[i..i+k-1] = S_1[j..j+k-1]$, from Theorem 1, we must have $P_1[i+k] - P_1[i] = P_1[j+k] - P_1[j]$, i.e., the substrings $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$ are of the same length. This suggests that sorting $S_1[i..i+k]$ and $S_1[j..j+k]$ is equal to sorting $S[P_1[i]..P_1[i+k]+d+1]$ and $S[P_1[j]..P_1[j+k]+d+1]$. Hence, the statement is correct in this case, too.

This theorem suggests that in order to find the orders for all D-Critical suffixes in S, we can sort S_1 instead. Because S_1 is at least 1/2 smaller than S, the computation on S_1 can be done within about one half the complexity for S. In the following subsections, we show how to bucket sort and name the items of P_1, i.e. the two crucial subtasks of computing S_1.

3.4 Sorting and Naming P_1

To bucket sort and name all the items of P_1, intuitively, we need at least three integer arrays of at most $2n_1 + n$ integers in total: two size-n_1 used as the alternating buffers for bucket sorting P_1, and another size-n for the bucket pointers, where $2n_1 \leq n$. The array of bucket pointers needs a size of n because each character of P_1 is in the range $[0, n-1]$. The space needed for sorting P_1 constitutes the space bottleneck for our algorithm. To further improve the space efficiency, we can use the following γ-weighting scheme for bucket sorting P_1 instead.

Definition 6. *(γ-Weighted Substring) The γ-weighted substring $S_\gamma[i..j]$ in S is defined as $S_\gamma[i..j] = S[i..j-1]S_\omega[j]$.*

For any two γ-weighted substrings, we immediately have the below result.

Lemma 4. *Given $S_\gamma[i..i+k] < S_\gamma[j..j+k]$ and $S[i..i+k] = S[j..j+k]$, we must have $t(S, i+x) \leq t(S, j+x)$ for any $x \in [0, k]$.*

By replacing $S_\omega[i..j]$ with $S_\gamma[i..j]$ as the weight of $P_1[i]$ for bucket sorting P_1 to produce S_1, we have the following result.

Theorem 3. *(γ-Order Equivalence) (1) Given $S_\gamma[P_1[i]..P_1[i]+d+1] = S_\gamma[P_1[j]..P_1[j]+d+1]$, there must be $S_\omega[P_1[i]..P_1[i]+d+1] = S_\omega[P_1[j]..P_1[j]+d+1]$; and (2) Given $S_\gamma[P_1[i]..P_1[i]+d+1] < S_\gamma[P_1[j]..P_1[j]+d+1]$, there must be $S_\omega[P_1[i]..P_1[i]+d+1] < S_\omega[P_1[j]..P_1[j]+d+1]$.*

Proof. Let $i' = P_1[i]$ and $j' = P_1[j]$. If $S_\gamma[i'..i'+d+1] = S_\gamma[j'..j'+d+1]$, we must have $S[i'..i'+d+1] = S[j'..j'+d+1]$ and $t(S, i'+d+1) = t(S, j'+d+1)$, i.e., $S_\omega[i'..i'+d+1] = S_\omega[j'..j'+d+1]$. Further, if $S_\omega[i'+d+1] = S_\omega[j'+d+1]$ and $S[i'+d] = S[j'+d]$, we must have $t(S, i'+d) = t(S, j'+d)$ as well as $S_\omega(i'+d) = S_\omega(j'+d)$, and so on for the other characters in the two substrings. Therefore, we must have $S_\omega[i'..i'+d+1] = S_\omega[j'..j'+d+1]$. When $S_\gamma[i'..i'+d+1] < S_\gamma[j'..j'+d+1]$, we consider these two cases:

– If $S[i'..i'+d+1] \neq S[j'..j'+d+1]$, given $S_\gamma[i'..i'+d+1] < S_\gamma[j'..j'+d+1]$, there must be $S[i'..i'+d+1] < S[j'..j'+d+1]$ from the definition of γ-weighted substring (Definition 6), which immediately yields $S_\omega[i'..i'+d+1] < S_\omega[j'..j'+d+1]$ from the definition of S_ω.
– If $S[i'..i'+d+1] = S[j'..j'+d+1]$, we must have $t(S, i'+d+1) = 0$ and $t(S, j'+d+1) = 1$. Further, from Lemma 4, we have $t(S, i'+x) \leq t(S, j'+x)$ for any $x \in [0, d+1]$, resulting in $S_\omega[i'..i'+d+1] < S_\omega[j'..j'+d+1]$.

Hence, we complete the proof.

Theorem 3 suggests that, to determine the order of two ω-weighted D-Critical substrings, we can use their γ-weighted counterparts instead. As a result, we need to compare the characters' types only for the last characters of the D-Critical substrings. Therefore, sorting all the items of P_1 according to the last characters of their γ-weighted siblings can be decomposed into two passes sequentially: (1) bucket sort according to the types of these characters; and (2) bucket sort according to these characters themselves. Using this method, we only need an array of $\Sigma(S)$ or n_1 integers for maintaining the bucket information at the 1st or 2nd iterations, respectively.

Now, provided with P_1, t and S, we can compute S_1, i.e. the reduced problem, using the two-step algorithm described below.

– Step 1: Bucket sort all the elements of P_1 into another array P_1' by their corresponding siblings (i.e. fixed-size D-Critical substrings) in S, with $\Sigma(S)$ buckets. The sorting is done through $d+2$ passes, in a manner of least-significant-character-first. This step requires a time complexity of $O(dn_1) = O(n_1)$, for $d = O(1)$.
– Step 2: Compute the names for all elements in P_1' (as well as P_1). This job can be done by a simple algorithm described as following: (i) allocate a size-n array *tmp*, where each item is an integer in $[0, n-1]$; (ii) initialize all items of *tmp* to be -1; (iii) scan P_1' once from left to right to compute all the names for the items of P_1', by setting $tmp[P_1'[i]]$ with the index of bucket that $P_1'[i]$ belonging to; (iv) pack all non-negative elements in *tmp* into the buffer of P_1', by traversing *tmp* once. Now, the buffer of P_1' stores the string of S_1.

One problem with Step 2 in the above algorithm is that in addition to P_1' and S_1, it uses a large space of n integers (each integer is of $\lceil \log n \rceil$ bits) for *tmp*. Alternatively, we can use another space-efficient algorithm for this by reusing *tmp* for P_1' and S_1, described as following. Let's define a logical array $\widetilde{tmp}_e =$

$\{tmp[i]|i\%2 = 0\}$ for the first n_1 even items of *tmp*, where $\widetilde{tmp_e}$ is said to be a logical array for its physical buffer is distributed into the first n_1 even items of *tmp*, i.e., its physical buffer is not spatially continuous.

Suppose that P'_1 is initially stored in the first n_1 items of *tmp*, we first copy P'_1 into $\widetilde{tmp_e}$ and set $tmp[j] = -1$ for any $tmp[j] \notin \widetilde{tmp_e}$, i.e., distribute P'_1 into the first even items of *tmp*. Next, we scan $\widetilde{tmp_e}$ from left to right to compute the names for all the the items of $\widetilde{tmp_e}$. For each $\widetilde{tmp_e}[i]$, we record its name as following: (1) if $\widetilde{tmp_e}[i]$ is even, set $tmp[\widetilde{tmp_e}[i] - 1]$ with the name; or else set $tmp[\widetilde{tmp_e}[i]]$ with the name. Now, all the items of S_1 are stored in the non-negative odd items of *tmp* in their correct relative positional orders. Last, we traverse *tmp* once to compact all the non-negative odd items into S_1. Using this method for Step 2, *tmp* is reused for accommodating both P'_1 and S_1, resulting in that only one n-integer array is required for all of them.

3.5 Inducing SA from $SA(S_1)$

For denotation simplicity, let $SA_0 = SA(S)$ and $SA_1 = SA(S_1)$. Furthermore, let $SA_{lms} = \{SA_0[i]|\mathrm{S[SA_0[i]]\ is\ a\ LMS\ character}\}$ and similarly $SA_l = \{SA_0[i]|\mathrm{S[SA_0[i]]\ is\ a\ type-L\ character}\}$. From SA_1, we can derive SA_{lms} and then induce SA_l from SA_{lms} and SA_0 from SA_l, as described below. The algorithm for inducing SA_0 from SA_1 consists of four sequential stages in $O(n)$ time/space:

1. Initialization: (1) Set all the items of SA_0 to be negative. (2) Scan S at most twice to find the buckets in SA_0 for all the suffixes in S according to their first characters.
2. Deriving SA_{lms} from SA_1: (1) Initialize all the buckets in SA_0 as empty by setting the start of each bucket as its end. (2) Scan SA_1 once from right to left, if $S[P_1[SA_1[i]]]$ is a LMS character then put $P_1[SA_1[i]]$ to the current start of its bucket in SA_0 and move the bucket start one item to the left.
3. Inducing SA_l from SA_{lms}: (1) For each bucket in SA_0, set the end as its start. (2) Scan SA_0 from left to right, for each non-negative item $SA_0[i]$, if $S[SA_0[i] - 1]$ is type-L then put $SA_0[i] - 1$ to the current end of its bucket and move the bucket end one item to the right.
4. Inducing SA from $SA_l(S)$: (1) For each bucket in SA_0, set the start as its end. (2) Scan SA_0 from right to left, for each item $SA_0[i]$, if $S[SA_0[i] - 1]$ is type-S then put $SA_0[i] - 1$ to the current start of its bucket and move the bucket start one item to the left.

In the above algorithm, in addition to SA_0, we need another array *bkt* for maintaining the start/end of each bucket on-the-fly in each stage, where *bkt* has $\|\Sigma(S)\|$ items and each item is of $\lceil \log n \rceil$ bits.

We now consider the correctness of the inducing algorithm for the stages 2-4. The correctness of stage 4, i.e. inducing SA from $SA_l(S)$, has been proven in [6] (Lemma 3), which is quoted and re-stated in this paper's terms as below.

Lemma 5. *[6] Given all the type-L (or type-S) suffixes of S sorted, all the suffixes of S can be sorted in $O(n)$ time.*

From the above lemma, it is trivial for us to have the below corollary for our previous work [10]. This corollary supports the correctness of stage 3 and re-stated here.

Corollary 1. *Given all the LMS suffixes of S sorted, all the type-L suffixes of S can be sorted in $O(n)$ time.*

Proof. From Lemma 5, we know that given SA_s, we can induce SA as well as SA_l in $O(n)$ time, by traversing SA once from left to right. Notice that not every type-S suffix is needed for sorting SA_l (here we use SA_l to denote all type-L suffixes); instead a type-S suffix is needed only when it is also a LMS suffix. In order words, knowing the order of all the LMS suffixes, we can traverse once from left to right to populate the order of SA_l in $O(n)$ time.

Given SA_1, the correctness of stage 2 is obvious, for we just simply copy all LMS items of SA_1 into the ends of their corresponding buckets (notice that in a bucket, a type-L suffix is less than a type-S suffix), keeping their relative orders unchanged.

4 Practical Strategies

We propose some techniques to further improve the time/space efficiencies of our algorithm in practice. Without loss of generality, we assume a 32-bit machine and each integer consumes 4 bytes.

General Strategy: Reusing the Buffer for $SA(S)$

From our algorithm framework in Fig.1, we see that the algorithm consists of three steps in sequence: (1) sorting P_1; (2) naming P_1 to obtain S_1; and (3) inducing $SA(S)$ from $SA(S_1)$. Notice that $SA(S)$ is an array of n integers, and both P_1 and S_1 have n_1 integers, where $2n_1 \leq n$, we can re-use the buffer for $SA(S)$ for the steps (1) and (2) too. For more details, the reader is referred to the sample codes in the appendix.

Strategy 1: Storing the LS-Type Array

Each element of the LS-type array for S is one-bit and a total of at most $n(1 + 1/2 + 1/4 + ... + \log^{-1} n) < 2n$ bits are required by the LS-arrays for all recursions. Hence, we can use the two most-significant-bits (MSBs) of $SA(S)[i]$ for storing the LS-type of $S[i]$. Recalling that the space for each integer is allocated in units of 4-byte instead of bits, the two MSBs of an integer is always available for us in this case. This is because that to compute $SA(S)$, our algorithm running on a 32-bit machine requires at least $5n$ bytes, where $4n$ for the items (each is a 4-byte integer) in $SA(S)$ and n for the input string (usually one byte per character). Therefore, the maximum size n_{max} of the input string must satisfy $5n_{max} < 2^{32}$, resulting in $n_{max} < 2^{32}/5$ and $\log n_{max} < 30$. In order words, 30 bits are enough for each item of $SA(S)$. However, for implementation convenience, we can simply store the LS-type arrays using bit arrays of maximum $2n$ bits in total, i.e. $0.25n$ bytes.

Strategy 2: Bucket Sorting P_1

Given the buffers for P_1 and S_1, to bucket sort P_1, we can use another array *bkt* in Fig. 1 for maintaining the buckets, where the size of *bkt* is determined by the alphabet size of the input string S. Even the original input string S is of a constant alphabet, after the first iteration, we will have S_1 as the input S for the next iteration. Since S_1 has an integer alphabet that can be as large as n_1 in the worst case, *bkt* may require a maximum space up to $n_1 \leq \lfloor n/2 \rfloor$ integers. To prevent *bkt* from growing with n_1, instead of sorting characters—each character is of 4 bytes—in each pass of bucket sorting the d-critical substrings, we simply sort each character with two passes, i.e., the bucket sorting is performed on units of 2-byte. The time complexity for bucket sorting all the fixed-size d-critical substrings at each iteration is linear proportional to the total number of characters for these substrings. Since each d-critical substring is of fixed-size $d+2$ characters and the number of substrings decreases at least half per iteration, the total number of characters sorted at each iteration is upper bounded by $O((d+2)(1/2+1/4+...+\log^{-1} n)) = O(dn)$, which is $O(n)$ given $d = O(1)$. Hence, the time complexity for bucket sorting in this way remains linear $O(n)$. For $n \leq 2^{32}$, the entire bucket sorting process will be half slowed down. However, the space for *bkt* can be fixed to 65536 integers, i.e. $O(1)$.

Strategy 3: Inducing the Final Result

In the inducing algorithm we described before, a buffer *bkt* is needed for dynamically recording the current start/end of each bucket. However, in order to save more space, we can use an alternative inducing algorithm which requires only the buffer for $SA(S_1)$ and needs no *bkt* when inducing $SA(S_1)$. This idea is to name the elements of P_1 in a different way: once all the items of P_1 have been sorted into their buckets, we can name each item of P_1 by the end of of its bucket to produce S_1. To be more precise, this is because the MSB of each item in SA_1 and S_1 is unused (when the strategy 1 is not applied). Given that each item of S_1 points to the end of its bucket in SA_1, the inducing can be done in this way: when an empty bucket in SA_1 is inserted the first item $S_1[i]$ at $SA_1[j]$, we set $SA_1[j] = i$ and $S_1[i] = j$, and mark the MSBs of $SA_1[j]$ and $S_1[i]$ by 1 to indicate they are borrowed for maintaining the bucket end. At the end of each inducing stage, we can restore the items in S_1 and SA_1 to their correct values in this way: scan SA_1 from left to right, for each $SA_1[i]$ with its MSB as 1, let $S_1[SA[i]] = i$ and and reset the MSBs of $SA_1[i]$ and $S_1[SA_1[i]]$ as 0.

5 Main Results

Theorem 4. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and the space complexities for the algorithm DCS in Fig. 1 to compute $SA(S)$ are $O(n)$ and $O(n\lceil \log n \rceil)$ bits, respectively.*

Proof. Because the problem is reduced at least 1/2 at each recursion, we have the time complexity governed by Eq. 1, where the reduced problem is of size at most $\lfloor n/2 \rfloor$. The first $O(n)$ in the equation accounts for reducing the problem and inducing the final solution from the reduced problem.

$$\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n) \tag{1}$$

The space complexity is obvious $O(n\lceil \log n \rceil)$ bits, for the size of each array used at the first iteration is upper bounded by $n\lceil \log n \rceil$ bits, and decreases at least a half for each iteration thereafter.

Theorem 5. *The DCS algorithm can construct the suffix array for a size-n string with a constant or integer alphabet using $O(n)$ time and a working space of only $0.25n + O(1)$ bytes.*

Proof. (Sketch) The key technique is to design the algorithm DCS with the general strategy and the strategies 2-3 in Section 4, the details of which are omitted here due to the limited space. We have coded in C++ a sample implementation for this (i.e. the DCS2 algorithm in the experiment section). Interested readers are welcome to contact us for the details/codes.

In [2], a working space of $0.03n$ bytes was reported for the algorithm proposed by Manzini and Ferragina, as a statistical result from experiments. Compared to this, our worst-case result of $0.25n + O(1)$ is approaching the existing best result for working space required by suffix array construction algorithms.

6 Experiments

For comparison convenience, we adopt some data sets from the Canterbury and the Manzini-Ferragina [2] corpora that are widely used for performance evaluations of various suffix array algorithms. All the input strings are of constant alphabets smaller than 256, and one byte is consumed by each character. The experiments were performed on a machine with AMD Athlon(tm) 64x2 Dual Core Processor 4200+ 2.20GHz and 2.00GB RAM, and the operating system is Linux (Sabayon Linux distribution).

The algorithms investigated in our experiments are limited to *linear* algorithms only, i.e. DCS1, DCS2, KS and KA, where DCS1 and DCS2 are the DCS algorithm with $d = 3$ and enhanced by the practical strategies proposed in Section 4. Specifically, the algorithms DCS1 and DCS2 use different settings of strategies: DCS1 uses the general strategy only, while DCS2 uses the strategies 2-3 in addition to the general strategy. The KS algorithm was downloaded

Table 1. Data Used in the Experiments

Data	$\|\Sigma\|$	Characters	Description
world192.txt	94	2 473 400	CIA world fact book
bible.txt	63	4 047 392	King James Bible
chr22.dna	4	34 553 758	Human chromosome 22
E.coli	4	4 638 690	Escherichia coli genome
sprot34.dat	66	109 617 186	Swissprot V34 protein database
etext99	146	105 277 340	Texts from Gutenberg project
howto	197	39 422 105	Linux Howto files

from Sanders's website [11], and the KA algorithm was downloaded from Ko's website [12]. All the programs were compiled by *g++* with the -O3 option. The performance measurements to be investigated are: the time/space complexities, the recursion depth and the mean reduction ratio. The time for each algorithm is the mean of 3 runs; and the space is the heap peak measured by using the *memusage* command to fire the running of each program. The total time (in seconds) and space (in million bytes, MBytes) for each algorithm are the sums of the times and spaces consumed by running the algorithm for all the input data, respectively. The mean time (measured in seconds per MBytes) and space (in bytes per character of the input string) for each algorithm are the total time and space divided by the total number of characters in all input data. The recursion depth is defined as the number of iterations, and the mean reduction ratio is the sum of reduction ratios for all iterations divided by the recursion depth.

We show in Table 2 the statistic results collected from the experiments, where the best results are styled in the italic fonts. For comparison convenience, we normalize all the results by that from the DCS1 algorithm. In the program for the KS algorithm, each character of the input string S is stored as a 4-byte integer, and the buffer for $SA(S)$ is not reused for the others. To be fair, we subtract $7n$ bytes from the space results we measured for the KS algorithm in the experiments, for we are sure $7n$ space can be trivially saved using some engineering tricks. From these results, we see that the best time and space performances are achieved by our algorithms DCS1 and DCS2, respectively. Specifically, the DCS1 algorithm is the fastest, which in average is more than twice (232%) faster than the KS, and 9% faster than the KA. The best space performance is achieved by the DCS2, which is 23% slower than the DCS1, however, still more than 1.5 times ((3.32-1.23)/1.23=169%) faster than the KS algorithm. The mean space of $24.34n$ for the KS algorithm in our experiments is about twice of the 10-13n for another space efficient implementation of the KS algorithm by Puglisi [13]. Even suppose the better 10-13n space, the KS algorithm still uses a space more than twice of that for our algorithms DCS1 and DCS2. Similar to the observations from the others [13, 14], the KA algorithm in our experiments is more space efficient than the KS algorithm, however, which uses about 60-70% more space than our algorithms DCS1 and DCS2, respectively.

The results for recursion depths and reduction ratios are machine-independent and deterministic for each given input string. Obviously, the smaller the reduction ratio is, the faster the algorithm is. For an overall comparison, we also give the total for the recursion depth and the reduction ratio for each algorithm and the mean for both, where the former is the sum of all corresponding results and the later is the former divided by the number of individual input data, i.e. 7 for the DCS and the KS algorithms and 5 for the KA algorithm in this case, respectively in this case. In this table, clearly, the DCS algorithm achieves the best reduction ratio for all the input data. The DCS algorithm has a mean reduction ratio about a half ((0.67-0.366)/0.67=45%) smaller than that of the KS algorithm. This well coincides with their time results. where the DCS1 algorithm is more than twice faster than the KS algorithm.

Table 2. Time, Space, Recursion Depth and Reduction Ratio

Data	Time (Seconds)				Space (MBytes)				Recursion Depth			Reduction Ratio		
	DCS1	DCS2	KS	KA	DCS1	DCS2	KS	KA	DCS	KS	KA	DCS	KS	KA
world	*1.61*	2	4.8	1.9	12.91	*12.50*	55.24	21.24	7	*6*	*6*	*.37*	.67	.42
bible	*3.11*	3.9	8.9	3.51	21.50	*20.30*	90.40	34.45	*6*	*6*	*6*	*.37*	.67	.45
chr22	*31.5*	39.6	92.8	33.41	184.44	*171.41*	819.25	289.97	10	12	*8*	*.36*	.67	.43
E.coli	*3.53*	4.3	10	3.98	25.15	*23.23*	105.93	40.01	8	*7*	8	*.36*	.67	.42
sprot	*111.59*	139.6	356	132.89	560.44	*543.26*	2591.62	930.06	*8*	9	9	*.37*	.67	.45
etext	*123.2*	150.4	428.1	149.67	559.55	*521.85*	2369.92	907.34	*12*	*12*	*12*	*.37*	.67	.45
howto	*36.3*	44.05	130.4	42.85	208.08	*195.55*	932.07	331.54	*10*	11	13	*.36*	.67	.45
Total	*310.84*	383.85	1031	368.21	1572.07	*1488.08*	6964.44	2554.61	*61*	63	62	-	-	-
Mean	*1.09*	1.34	3.60	1.29	5.49	*5.20*	24.34	8.93	*8.71*	9.0	8.85	*0.366*	0.67	0.438
Norm.	*1*	1.23	3.32	1.18	1	*0.95*	4.43	1.63	*1*	1.03	1.02	*1*	1.83	1.19

7 Closing Remarks

In this paper, we have analyzed the D-Critical substring algorithm [9], a new linear time/space SACA which uses a novel concept called D-Critical Substrings to encode the original text. The implementation of our algorithm is as intuitive as that of the KS algorithm and our algorithm runs even faster than the KA algorithm. The complete C++ source code (around 100+ effective lines) of the sample implementation of the proposed algorithms used in our experiments is available upon readers' request.

Recently, after the development of the D-Critical-Substring algorithm, we proposed another SA algorithm using the almost pure induced sorting technique (i.e. the SA-IS algorithm in [15]), and both algorithms are running in linear time. Currently, the two algorithms have been investigated under the assumption that the whole input string is completely stored in the main memory. However, as we have noticed that, there are increasing needs for building the SAs of huge corpora, e.g., the ever growing genome databases. When the input string demands more space than the main memory can provide, we have to seek for help from using the slower but far more larger external memory such as flash memory or harddisk. Our recent study on using external memory for our proposed linear SA algorithms indicated that, compared with the SA-IS algorithm, the D-Critical-Substring algorithm has greater potentials to be improved for using external memory, due to its distinct advantage of sorting fixed-size d-critical substrings, in contrast to sorting variable-size substrings in the SA-IS algorithm.

References

1. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms. (1990) 319–327
2. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica **40**(1) (September 2004) 33–50
3. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: The 32nd Annual ACM Symposium on Theory of Computing (STOC'00). (2000) 397–406
4. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proceedings 14th Annual Symp. Combinatorial Pattern Matching, LNCS 2676, Springer-Verlag. (2003) 186–199
5. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: The 30th International Colloquium on Automata, Languages and Programming (ICALP '03). (2003) 943–955
6. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proceedings 14th Annual Symp. Combinatorial Pattern Matching, LNCS 2676, Springer-Verlag. (2003) 200–210
7. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97). (1997) 137
8. Mori, Y.: divsufsort. [Online] Available: http://homepage3.nifty.com/wpage/software/libdivsufsort.html (2007)
9. Zhang, S., Nong, G.: Fast and space efficient linear suffix array construction. In: IEEE Data Compression Conference 2008 (DCC '08). (2008) 553–553
10. Nong, G., Zhang, S.: Optimal lightweight construction of suffix arrays. In: Proceedings WADS '07, 10th Workshop on Algorithms and Data Structures, Halifax, Canada (August 2007)
11. Sanders, P.: A driver program for the KS algorithm. [Online] Available: http://www.mpi-inf.mpg.de/ sanders/programs/suffix/. (2007)
12. Ko, P.: Source codes for the KA algorithm. http://kopang.public.iastate.edu/homepage.php?page=source (2007)
13. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2) (2007) 1–31
14. Lee, S., Park, K.: Efficient implementations of suffix array construction algorithms. In: Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms. (2004) 64–72
15. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: IEEE Data Compression Conference 2009 (DCC '09). (To Appear)