Suffix Arrays: A New Method for On-Line String Searches

Udi Manber*
Gene Myers#

Abstract

A new and conceptually simple data structure, called a *suffix array*, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that they are three to five times more space efficient. Suffix arrays permit on-line string searches of the type, "Is W a substring of A?" to be answered in time $O(P + \log N)$, where P is the length of W and N is the length of A, which is competitive with (and in some cases slightly better than) suffix trees. The only drawback is that in those instances where the underlying alphabet is finite and small, suffix trees can be constructed in $O(N)$ time in the worst-case versus $O(N \log N)$ time for suffix arrays. We show, however, that suffix arrays can be constructed in $O(N)$ expected time, regardless of the alphabet size. We believe that suffix arrays will prove to be better in practice than suffix trees for many applications.

1. Introduction

Finding all instances of a string W in a large text A is an important pattern matching problem. There are many applications in which a fixed text is queried many times. In these cases, it is worthwhile to construct a data structure to allow fast queries. *Suffix trees* are data structures that admit efficient on-line string searches. A suffix tree for a text A of length N over an alphabet Σ can be built in $O(N \log |\Sigma|)$ time and $O(N)$ space [Wei73, McC76]. Suffix trees permit on-line string searches of the type, "Is W a substring of A?" to be answered in $O(P \log |\Sigma|)$ time, where P is the length of W. We explicitly consider the dependence of the complexity of the algorithms on $|\Sigma|$, rather than assume that it is a fixed constant, because Σ can be quite large for many applications. Suffix trees can also be constructed in time $O(N)$ with $O(P)$ time for a query, but this requires $O(N |\Sigma|)$ space, which renders this method impractical in many applications.

Suffix trees have been studied and used extensively. A survey paper by Apostolico [Apo85] cites over forty references. Suffix trees have been refined from tries to minimum state finite automaton [BBE85], generalized to on-line construction [MR80, BB86], and real-time construction [Sli80], and parallelized [AI86]. Suffix trees have been applied to fundamental string problems such as finding the longest repeated substring [Wei73], finding all squares or repetitions in a string [AP83], computing substring statistics [AP85], approximate string matching [LV86, Mye88], and string comparison [EH86]. They have also been used to address other types of problems

* Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T.
# Supported in part by the NIH (grant R01 LM04960-01).

such as text compression [RPE81], compressing assembly code [FWM84], inverted indices [Car75], and analyzing genetic sequences [CHM86]. Galil [Ga85] lists a number of open problems concerning suffix trees and on-line string searching.

In this paper, we present a new data structure, called a *suffix array*, that is basically a sorted list of all the suffixes of A. When coupled with information about the *longest common prefixes (lcps)* of adjacent elements in the suffix array, string searches can be answered in $O(P + \log N)$ time with a simple augmentation to a classic binary search. The suffix array and associated *lcp* information occupy a mere $2N$ integers, and searches are shown to require at most $P + \lceil \log_2 (N-1) \rceil$ single-symbol comparisons. The construction of the suffix array and *lcp* information require $O(N \log N)$ time in the worst case. Under the assumption that all strings of N symbols are equally likely, the expected length of the *longest* repeated substring is $O(\log N / \log |\Sigma|)$ [KGO83]. By further refining our algorithms to take advantage of this fact, we can construct a suffix array and its *lcp* information in $O(N)$ expected time.

Our approach distills the nature of a suffix tree to its barest essence: A sorted array coupled with another to accelerate the search. Suffix arrays may be used in lieu of suffix trees in the many applications of this ubiquitous structure. Our search and sort approach is distinctly different and, in theory, provides superior querying time at the expense of somewhat slower construction. Galil [Ga85, Problem 9] poses the problem of designing algorithms that are not dependent on $|\Sigma|$ and our algorithms meet this criterion, i.e., $O(P + \log N)$ search time with an $O(N)$ space structure, independent of Σ. In practice, an implementation based on a blend of the ideas in this paper compares favorably with an implementation based on suffix trees. Our suffix array structure requires only $5N$ bytes on a VAX, which is three to five times more space efficient than any reasonable suffix tree encoding. Search times are competitive, but suffix arrays do require three to ten times longer to build. For these reasons, we believe that suffix arrays will become the data structure of choice for the many applications where the text is very large. In fact, we recently found that the basic concept of suffix arrays (sans the *lcp* and a provable efficient algorithm) has been used in the Oxford English Dictionary (OED) project at the university of Waterloo [Go89]. Suffix arrays have also been used as a basis for a sublinear approximate matching algorithm [ME89].

The paper is organized as follows. In Section 2, we present the search algorithm under the assumption that the suffix array and the *lcp* information have been computed. In Section 3, we show how to construct the sorted suffix array. In Section 4, we give the algorithm for computing the *lcp* information. In Section 5, we modify the algorithms to achieve better expected running times. We end with empirical results and comments about practice in Section 6.

2. Searching

Let $A = a_0 a_1 \cdots a_{N-1}$ be a large text of length N. Denote by $A_i = a_i a_{i+1} \cdots a_{N-1}$ the suffix of A that starts at position i. The basis of our data structure is a lexicographically sorted array, *Pos*, of the suffixes of A; namely, $Pos[k]$ is the start position of the kth smallest suffix in the set $\{A_0, A_1, \ldots A_{N-1}\}$. The sort that produces the array *Pos* is described in the next Section. For now we assume that *Pos* is given; namely, $A_{Pos[0]} < A_{Pos[1]} < \ldots < A_{Pos[N-1]}$, where "<" denotes the lexicographical order.

For a string u, let u^p be the prefix consisting of the first p symbols of u if u contains more than p symbols, and u otherwise. We define the relation $<_p$ to be the lexicographical order of p-symbol prefixes; that is, $u <_p v$ iff $u^p < v^p$. We define the relations $\le_p$, $=_p$, $_p$, $>_p$, and $\ge_p$ in a similar way. Note that, for any choice of p, the *Pos* array is also ordered according to $\le_p$, because $u < v$ implies $u \le_p v$. All suffixes that have equal p-prefixes, for some $p < N$, must appear in consecutive positions in the *Pos* array, because the *Pos* array is sorted lexicographically. These facts are central to our search algorithm.

Suppose that we wish to find all instances of a string $W = w_0 w_1 \cdots w_{P-1}$ of length $P \le N$ in A. Let $L_W = \min(k : W \le_P A_{Pos[k]} \text{ or } k = N)$ and $R_W = \max(k : A_{Pos[k]} \le_P W \text{ or } k = -1)$. Since *Pos* is in $\le_P$-order, it follows that W matches $a_i a_{i+1} \cdots a_{i+P-1}$ if and only if $i = Pos[k]$ for some $k \in [L_W, R_W]$. Thus, if L_W and R_W can be found quickly, then the number of matches is $R_W - L_W + 1$ and their left endpoints are given by $Pos[L_W], Pos[L_W+1], \ldots Pos[R_W]$. But *Pos* is in $\le_P$-order, hence a simple binary search can find L_W and R_W using $O(\log N)$ comparisons of strings of size at most P; each such comparison requires $O(P)$ single-symbol comparisons. Thus, the *Pos* array allows us to find all instances of a string in A in time $O(P \log N)$. The algorithm is given in Fig. 1.

```
1.   if W ≤_P A_Pos[0] then
2.       L_W ← 0
3.   else if W >_P A_Pos[N−1] then
4.       L_W ← N
5.   else
6.     { (L, R) ← (0, N−1)
7.       while R−L > 1 do
8.         { M ← (L+R)/2
9.           if W ≤_P A_Pos[M] then
10.              R ← M
11.          else
12.              L ← M
           }
13.      L_W ← R
     }
```

Figure 1: An $O(P\log N)$ search for L_W.

The algorithm in Fig. 1 is very simple, but its running time can be improved. We show next that the $\le_p$-comparisons involved in the binary search need not be started from scratch in each iteration of the while loop. We can use information obtained from one comparison to speedup the ensuing comparisons. When this strategy is coupled with some additional precomputed information, the search is improved to $P+\lceil\log_2(N-1)\rceil$ single-symbol comparisons in the worst case, which is a substantial improvement.

Let $lcp(v, w)$ be the length of the longest common prefix of v and w. When we lexicographically compare v and w in a left-to-right scan that ends at the first unequal symbol we obtain $lcp(v, w)$ as a byproduct. We can modify the binary search in Fig. 1 by maintaining two variables, l and r, such that $l = lcp(A_{Pos[L]}, W)$, and $r = lcp(W, A_{Pos[R]})$. Initially, l is set by the comparison of W and $A_{Pos[0]}$ in line 1, and r is set in the comparison against $A_{Pos[N-1]}$ in line 3. Thereafter, each comparison of W against $A_{Pos[M]}$ in line 9, permits l or r to be appropriately updated in line 10 or 12, respectively. By so maintaining l and r, $h = min(l, r)$ single-symbol comparisons can be saved when comparing $A_{Pos[M]}$ to W, because $A_{Pos[L]} =_l W =_r A_{Pos[R]}$ implies $A_{Pos[k]} =_h W$ for all k in $[L, R]$ including M. While this reduces the number of single-symbol comparisons needed to determine the $\le_P$-order of a midpoint with respect to W, it turns out that the worst case running time is still $O(P\log N)$.

To reduce the number of single-symbol comparisons to $P+\lceil\log_2(N-1)\rceil$ in the worst case, we use precomputed information about the $lcps$ of $A_{Pos[M]}$ with each of $A_{Pos[L]}$ and $A_{Pos[R]}$. Consider the set of all triples (L, M, R) that can arise in the inner loop of the binary search of Fig. 1. There are exactly $N-2$ such triples, each with a unique midpoint $M \in [1, N-2]$, and for each triple $0 \le L < M < R \le N-1$. Suppose that (L_M, M, R_M) is the unique triple containing midpoint M. Let $Llcp$ be an array of size $N-2$ such that $Llcp[M] = lcp(A_{Pos[L_M]}, A_{Pos[M]})$, and Let $Rlcp$ be another array of size $N-2$ such that $Rlcp[M] = lcp(A_{Pos[M]}, A_{Pos[R_M]})$. The construction of the two $(N-2)$-element arrays, $Llcp$ and $Rlcp$, can be interwoven with the sort producing Pos and will be shown in Section 4. For now, we assume that the $Llcp$ and $Rlcp$ arrays have been precomputed.

Consider an iteration of the search loop for triple (L, M, R), and, without loss of generality, assume that $l \ge r$. Let $h = max(l, r)$ and let Δh be the difference between the value of h at the beginning and at the end of the iteration. There are three cases to consider[1], based on whether $Llcp[M]$ is greater than, equal to, or less than l. The cases are illustrated in Fig. 2(a), 2(b), and 2(c), respectively. The vertical bars denote the $lcps$ between W and the suffixes in the Pos array (except for l and r, these $lcps$ are not known at the time we consider M). The shaded areas illustrate $Llcp[M]$. For each case, we must determine whether L_W is in the right half or the left half (the binary search step) and we must update the value of

[1] The first two cases can be combined in the program. We use three cases only for description purposes.

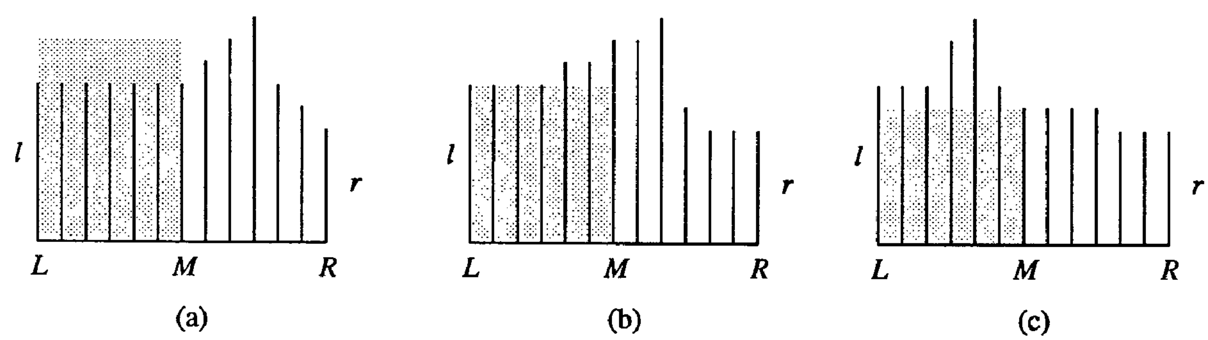

Figure 2: The three cases of the $O(P+\log N)$ search.

either l or r. It turns out that both these steps are easy to make:

Case 1: $Llcp[M] > l$ (Fig. 2(a)): in this case, $A_{Pos[M]} =_{l+1} A_{Pos[L]} \neq_{l+1} W$, and so W must be in the right half and l is unchanged.

Case 2: $Llcp[M] = l$ (Fig. 2(b)): in this case, we know that the first l symbols of $Pos[M]$ and W are equal; thus, we need to compare only the $l+1$th symbol, $l+2$th symbol, and so on, until we find one, say $l+j$, such that $W_{l+j} Pos[M]$. The $l+j$th symbol determines whether L_W is in the right or left side. In either case, we also know the new value of r or l — it is $l+j$. Since $l = h$ at the beginning of the loop, this step takes $\Delta h+1$ single-symbol comparisons.

Case 3: $Llcp[M] < l$ (Fig. 2(c)): in this case, since W matched l symbols of L and $< l$ symbols of M, it is clear that L_W is in the left side and than the new value of r is $Llcp[M]$.

Hence, the use of the arrays *Llcp* and *Rlcp* (the *Rlcp* array is used when $l < r$) reduces the number of single-symbol comparisons to no more than $\Delta h+1$ for each iteration. Summing over all iterations and observing that $\Sigma \Delta h \leq P$, the total number of single-symbol comparisons made in an on-line string search is at most $P + \lceil \log_2(N-1) \rceil$, and $O(P + \log N)$ time is taken in the worst-case. The precise search algorithm is given in Fig. 3.

$$
\begin{array}{l}
l \leftarrow lcp(A_{Pos[0]}, W) \\
r \leftarrow lcp(A_{Pos[N-1]}, W) \\
\textbf{if } l = P \textbf{ or } w_l \leq a_{Pos[0]+l} \textbf{ then} \\
\quad L_W \leftarrow 0 \\
\textbf{else if } r < P \textbf{ or } w_r \leq a_{Pos[N-1]+r} \textbf{ then} \\
\quad L_W \leftarrow N \\
\textbf{else} \\
\quad \{\ (L, R) \leftarrow (0, N-1) \\
\qquad \textbf{while } R - L > 1 \textbf{ do} \\
\qquad \{\ M \leftarrow (L+R)/2 \\
\qquad\quad \textbf{if } l \geq r \textbf{ then} \\
\qquad\qquad \textbf{if } Lcp[M] \geq l \textbf{ then} \\
\qquad\qquad\quad m \leftarrow l + lcp(A_{Pos[M]+l}, W_l) \\
\qquad\qquad \textbf{else } m \leftarrow Lcp[M] \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{if } Rcp[M] \geq r \textbf{ then} \\
\qquad\qquad\quad m \leftarrow r + lcp(A_{Pos[M]+r}, W_r) \\
\qquad\qquad \textbf{else } m \leftarrow Rcp[M] \\
\qquad\quad \textbf{if } m = P \textbf{ or } w_m \leq a_{Pos[M]+m} \textbf{ then} \\
\qquad\qquad (R, r) \leftarrow (M, m) \\
\qquad\quad \textbf{else } (L, l) \leftarrow (M, m) \\
\qquad \} \\
\qquad L_W \leftarrow R \\
\quad \}
\end{array}
$$

Figure 3: An $O(P + \log N)$ search for L_W.

3. Sorting

The sorting is done in $\lceil \log_2(N+1) \rceil$ stages. In the first stage, the suffixes are put in buckets according to their first symbol. Then, inductively, each stage further partitions the buckets by sorting according to twice the number of symbols. For simplicity of notation, we number the stages 1, 2, 4, 8, etc., to indicate the number of affected symbols. Thus, in the H^{th} stage, the suffixes are sorted according to the $\leq_H$-order. For simplicity, we pad the suffixes by adding blank symbols, such that the lengths of all of them become $N+1$. (This padding is not necessary, but it simplifies the discussion.) The first stage consists of a bucket sort according to the first symbol of each suffix. The suffixes are divided into m_1 buckets ($m_1 \leq |\Sigma|$), each holding the suffixes with the same first symbol. Assume that after the H^{th} stage the suffixes are partitioned into m_H buckets, each holding suffixes with the same H first symbols, and that these buckets are sorted according to the $\leq_H$-relation. We will show how to sort the elements in each H-bucket to produce the $\leq_{2H}$-order in $O(N)$ time. Our sorting algorithm uses similar ideas to those in [KMR72].

Let A_i and A_j be two suffixes belonging to the same bucket after the H^{th} step; that is, $A_i =_H A_j$. We need to compare A_i and A_j according to the next H symbols. But, the next H symbols of A_i (A_j) are exactly the first H symbols of A_{i+H} (A_{j+H}). By the assumption, we already know the relative order, according to the $\leq_H$-relation, of A_{i+H} and A_{j+H}. It remains to see how we can use that knowledge to complete the stage efficiently. We first describe the main idea, and then show how to implement it efficiently.

We start with the first bucket, which must contain the smallest suffixes according to the $\leq_H$-relation. Let A_i be the first suffix in the first bucket (i.e., $Pos[1] = i$), and consider A_{i-H} (if $i - H < 0$, then we ignore A_i and take the suffix of $Pos[2]$, and so on). Since A_i starts with the smallest H-symbol string, A_{i-H} should be the first in its $2H$-bucket. Thus, we move A_{i-H} to the beginning of its bucket and mark this fact. For every bucket, we need to know the number of suffixes in that bucket that have already been moved and thus placed in $\leq_{2H}$-order. The algorithm basically scans the suffixes as they appear in the $\leq_H$-order, and for each A_i it moves A_{i-H} (if it exists) to the next available place in its H-bucket. While this basic idea is simple, its efficient implementation (in terms of both space and time) is not trivial. We describe it below.

We maintain three integers arrays, *Pos*, *Prm*, and *Count*, and two boolean arrays, *BH* and *B2H*, all with N

elements[2]. At the start of stage H, $Pos[i]$ contains the start position of the i^{th} smallest suffix, $Prm[i]$ is the inverse of Pos, namely, $Prm[Pos[i]]=i$, and $BH[i]$ is 1 iff $Pos[i]$ contains the leftmost suffix of an H-bucket (i.e., $A_{Pos[i]} \neq_H A_{Pos[i-1]}$). *Count* and *B2H* are temporary arrays; their use will become apparent in the description of a stage of the sort. A radix sort on the first symbol of each suffix is easily tailored to produce *Pos*, *Prm*, and *BH* for stage 1 in $O(N)$ time. Assume that *Pos*, *Prm*, and *BH* have the correct values after stage H, and consider stage $2H$.

We first reset $Prm[i]$ to point to the leftmost cell of the H-bucket containing the i^{th} suffix rather than to suffix's precise place in the bucket. We also initialize $Count[i]$ to 0 for all i. All operations above can be done in $O(N)$ time. We then scan the *Pos* array in increasing order, one bucket at a time. Let l and r ($l \leq r$) mark the left and right boundary of the H-bucket currently being scanned. Let T_i (the H-tail of i) denote $Pos[i]-H$. For every i, $l \leq i \leq r$, we increment $Count[Prm[T_i]]$, set $Prm[T_i] = Prm[T_i] + Count[Prm[T_i]] - 1$, and set $B2H[Prm[T_i]]$ to 1. In effect, all the suffixes whose $H+1^{th}$ through $2H^{th}$ symbols equal the unique H-prefix of the current H-bucket are moved to the top of their H-buckets. The *B2H* field is used to mark those prefixes that were moved. Before the next H-bucket is considered, we make another pass, find all the moved suffixes, and reset the *B2H* fields such that only the leftmost of them in each $2H$-bucket is set to 1, and the rest are reset to 0. This way, the *B2H* fields correctly mark the beginning of the $2H$-buckets. Thus the scan updates *Prm* and sets *B2H* so that they are consistent with the $\leq_{2H}$-order of the suffixes. In the final step, we update the *Pos* array (which is the inverse of *Prm*), and set *BH* to *B2H*. All the steps above can clearly be done in $O(N)$ time, and, since there are at most $\lceil \log_2(N+1) \rceil$ stages, the sorting requires $O(N \log N)$ time in the worst case. Average-case analysis is presented in Section 5.

4. Finding Longest Common Prefixes

The $O(P + \log N)$ search algorithm requires precomputed information about the *lcps* between the suffixes starting at each midpoint M and its left and right boundaries L_M and R_M. We first show how to compute the *lcps* between suffixes that are consecutive in the sorted *Pos* array. We will see later how to compute all the necessary *lcps*. The key idea is the following. Assume that after stage H we know the *lcps* between suffixes in adjacent buckets (after the first stage, the *lcps* between suffixes in adjacent buckets are 0). At stage $2H$ the buckets are partitioned according to $2H$ symbols. Thus, the *lcps* between suffixes in newly adjacent buckets must be at least H and at most $2H-1$. Furthermore, if A_p and A_q are in the same H-bucket but are in distinct $2H$-buckets, then

$$lcp(A_p, A_q) = H + lcp(A_{p+H}, A_{q+H}). \tag{1}$$

If we can maintain, after stage H, information about all *lcps* whose values are less than H, then computing *lcps* in stage $2H$ will be straightforward from (1). But, this is too much information. We are interested only in computing *lcps* between adjacent suffixes in the final order, not between every pair of prefixes. Instead of computing all *lcps*, we maintain an $O(N)$-space data structure that enables us to compute any *lcp* whose value is less than H in $O(\log N)$ time. We will describe this data structure, which we call an *interval tree*, after we establish our basic approach.

We define $height(i) = lcp(A_{Pos[i-1]}, A_{Pos[i]})$, $1 \leq i \leq N-1$, where *Pos* is the final sorted order of the suffixes. These $N-1$ *height* values are computed in an array $Hgt[i]$. The computation is performed inductively, together with the sort, such that $Hgt[i]$ achieves its correct value at stage H iff $height(i) < H$, and it is undefined (specifically, $N+1$) otherwise. Notice that, if $height(i) < H$, then $A_{Pos[i-1]}$ and $A_{Pos[i]}$ must be in different H-buckets since H-buckets contain suffixes with the same H-symbol prefix.

Let Pos^H, Hgt^H, and Prm^H be the values of the given arrays at the end of stage H. In stage $2H$ of the sort, the $\leq_{2H}$-ordered list Pos^{2H} is produced by sorting the suffixes in each H-bucket of the $\leq_H$-ordered list Pos^H. The following lemma captures the essence of how we compute Hgt^{2H} from Hgt^H given Pos^{2H} and Prm^{2H}.

Lemma 1: If $H \leq height(i) < 2H$ then $height(i) = H + \min(Hgt^H[k] : k \in [\min(a, b)+1, \max(a, b)])$, where $a = Prm^{2H}[Pos^{2H}[i-1]+H]$, and $b = Prm^{2H}[Pos^{2H}[i]+H]$.

Proof: Let $p = Pos^{2H}[i-1]$ and $q = Pos^{2H}[i]$. As we have observed, $height(i) < 2H$ implies $height(i) = H + lcp(A_{p+H}, A_{q+H})$. Next observe that $Pos^{2H}[a] = p+H$ and $Pos^{2H}[b] = q+H$ by the choice of a and b. Without loss of generality, assume that $a < b$. We now know that $height(i) = H + lcp(u, v)$ where $u = A_{Pos^{2H}[a]}$, $v = A_{Pos^{2H}[b]}$, $lcp(u, v) < H$, and $u <_H v$. Observe that $x <_H z$ and $x \leq_H y \leq_H z$ imply $lcp(x, z) = \min(lcp(x, y), lcp(y, z))$. It follows, by induction, that if $x_0 <_H x_n$ and $x_0 \leq_H x_1 \leq_H \cdots \leq_H x_n$ then $lcp(x_0, x_n) = \min(lcp(x_{k-1}, x_k) : k \in [1, n])$. Thus, $lcp(u, v) =$

[2] In fact, two integers are sufficient, and since these integers are always positive we can use their sign bit for the boolean values. Thus, the space requirement is only two integers per symbol. We present a slightly simplified version in this paper.

$min(lcp(A_{Pos^{2H}[k-1]}, A_{Pos^{2H}[k]}) : k \in [a+1, b])$. Now $lcp(u, v) < H$ implies that at least one term in the minimum is less than H. For those terms less than H, $lcp(A_{Pos^{2H}[k-1]}, A_{Pos^{2H}[k]}) = height(k) = Hgt^H[k]$. This, combined with the fact that $Hgt^H[k] = N+1 \geq H$ for all other terms, gives the result. ■

We are now ready to describe the algorithm. In the first stage, we set $Hgt[i]$ to 0 if $a_{Pos^1[i-1]} \neq a_{Pos^1[i]}$, and $N+1$ otherwise. This correctly establishes Hgt^1. At the end of stage $2H > 1$, we have computed Pos^{2H}, Prm^{2H}, and BH^{2H} (marks the $2H$-buckets). Thus, by Lemma 1, the following code correctly establishes Hgt^{2H} from Hgt^H when placed at the end of a sorting stage.

```
for i ∈ [1, N−1] such that BH[i] and Hgt[i] > N do
{  a ← Prm[Pos[i−1]+H]
   b ← Prm[Pos[i]+H]
   Set(i, H + Min_Height(min(a, b)+1, max(a, b)))
   { these routines are defined below }
}
```

The routine $Set(i, h)$ sets $Hgt[i]$ to h in our interval tree, and $Min\_Height(i, j)$ determines $\min(Hgt[k] : k \in [i, j])$ using the interval tree. We now show how to implement each routine in time $O(\log N)$ in the worst case. Consider a balanced and full binary tree with $N-1$ leaves which, in left-to-right order, correspond to the elements of the array Hgt. The tree has height $O(\log N)$ and $N-2$ interior vertices. Assume that a value $Hgt[v]$ is also kept at each interior vertex v. We say that the tree is *current* if for every interior vertex v, $Hgt[v] = min(Hgt[left(v)], Hgt[right(v)])$, where $left(v)$ and $right(v)$ are the left and right children of v.

Let T be a current tree. We need to perform two operations on the tree, a query $Min\_Height(i, j)$, and a dynamic operation $Set(i, h)$. The query operation $Min\_Height(i, j)$ computes $\min(Hgt[k] : k \in [i, j])$. It can be answered in $O(\log N)$ time as follows. Let $lca(i, j)$ be the lowest common ancestor of leaves i and j. Since the tree is fixed, $lca(i, j)$ can be found in constant time with simple arithmetics (see, for example, [HT84]). Let P be the set of vertices on the path from i to $lca(i, j)$ excluding $lca(i, j)$, and let Q be the similar path for leaf j. $Min\_Height(i, j)$ is the minimum of the following values: (1) $Hgt[i]$, (2) $Hgt[w]$ such that $right(v) = w$ and $w \notin P$ for some $v \in P$, (3) $Hgt[w]$ such that $left(v) = w$ and $w \notin Q$ for some $v \in Q$, and (4) $Hgt[j]$. These $O(\log N)$ vertices can be found and their minimum computed in $O(\log N)$ time. The operation $Set(i, h)$ sets $Hgt[i]$ to h and then makes T current again by updating the Hgt values of the interior vertices on the path from i to the root. This takes $O(\log N)$ time.

Overall, the time taken to compute the *height* values in stage H is $O(N + Set_H \log N)$ where Set_H is the number of indices i for which $height(i) \in [H, 2H-1]$. Since $\Sigma Set_H = N$ over all stages, the total additional time required to compute Hgt during the sort is $O(N \log N)$.

The Hgt array gives the lcps of suffixes that are consecutive in the Pos array. We now show that the arrays $Llcp$ and $Rlcp$ can be computed similarly. We are free to choose any full and balanced tree for this the scheme. Using the tree based on the binary search of Figure 1 gives us the arrays $Llcp$ and $Rlcp$ needed for the search in a direct fashion. The tree consists of $2N-3$ vertices each labeled with one of the $2N-3$ pairs, (L, R), that can arise at entry and exit from the while loop of the binary search. The root of the tree is labeled $(0, N-1)$ and the remaining vertices are labeled either (L_M, M) or (M, R_M) for some midpoint $M \in [1, N-2]$. Alternately, the tree's $N-2$ interior vertices are (L_M, R_M) for each midpoint M, and its $N-1$ leaves are $(i-1, i)$ for $i \in [1, N-1]$ in left to right order. For each interior vertex, $left((L_M, R_M)) = (L_M, M)$ and $right((L_M, R_M)) = (M, R_M)$. Since the tree is full and balanced, it is appropriate for realizing Set and Min_Height if we let leaf $(i-1, i)$ hold the value of $Hgt[i]$. Moreover, at the end of the sort, $Hgt[(L, R)] = \min(height(k) : k \in [L+1, R]) = lcp(A_{Pos[L]}, A_{Pos[R]})$. Thus, $Llcp[M] = Hgt[(L_M, M)]$ and $Rlcp[M] = Hgt[(M, R_M)]$. So with this tree, the arrays $Llcp$ and $Rlcp$ are directly available upon completion of the sort.[3]

5. Linear Time Expected-case Variations

We now consider the expected time complexity of constructing and searching suffix arrays under the distributional model where all N-symbol strings are equally likely[4]. Under this input distribution, the expected length of the longest repeated substring has been shown to be $2\log_{|\Sigma|} N + O(1)$ [KGO83]. This fact provides the central leverage for all the results that follow. Note that it immediately imples that, in the expected case, Pos will be completely sorted after $O(\log \log N)$ stages, and the sorting algorithm of Section 3 thus takes $O(N \log \log N)$ expected time.

[3] The interval tree requires $2N-3$ positive integers. However, the observation that one child of each interior vertex has the same value as its father, permits interval trees (and thus the $Llcp$ and $Rlcp$ arrays) to be encoded and manipulated as $N-1$ signed integers.

[4] The ensuing results also hold under the more general model where each text is assumed to be the result of N independent Bernoulli trials of a $|\Sigma|$-sided coin toss, which is not necessarily uniform.

The expected sorting time can be reduced to $O(N)$ by modifying the radix sort of the first stage as follows. Let $T=\lfloor \log_{|\Sigma|} N \rfloor$ and consider mapping each string of T symbols over Σ to the integer obtained when the string is viewed as a T-digit, radix-$|\Sigma|$ number. This oft-used encoding is an isomorphism onto the range $[0, |\Sigma|^T-1] \subseteq [0, N-1]$, and the $\leq$-relation on the integers is identical with the $\leq_T$-relation on the corresponding strings. Let $Int_T(A_p)$ be the integer encoding of the T-symbol prefix of suffix A_p. It is easy to compute $Int_T(A_p)$ for all p in a single $O(N)$ sweep of the text by employing the observation that $Int_T(A_p) = a_p |\Sigma|^{T-1} + \lfloor Int_T(A_{p+1}) / |\Sigma| \rfloor$. Instead of performing the initial radix sort on the first symbol of each suffix, perform it on the integer encoding of the first T symbols of each suffix. This radix sort still takes just $O(N)$ time and space because the choice of T guarantees that the integer encodings are all less than N. Moreover, it sorts the suffixes according to the $\leq_T$-relation. Effectively, the base case of the sort has been extended from $H = 1$ to $H = T$ with no loss of asymptotic efficiency. Since the expected length of the longest repeated substring is $T(2+O(T^{-1}))$, at most 2 subsequent stages are needed to complete the sort in the expected case. Thus this slight variation gives an $O(N)$ expected time algorithm for sorting the suffixes.

Corresponding expected-case improvements for computing the *lcp* information, in addition to the sorted suffix array, are harder to come by. We can still achieve $O(N)$ expected-case time, but by employing an approach to computing *height*(i) that uses identity (4.1) recursively to obtain the desired *lcp*s. Let the *sort history* of a particular sort be the tree that models the successive refinement of buckets during the sort. There is a vertex for each H-bucket except those H-buckets that are identical to the $(H/2)$-buckets containing them. The sort history thus has $O(N)$ vertices, as each leaf corresponds to a suffix and each interior vertex has at least two children. Each vertex contains a pointer to its parent and each interior vertex also contains the stage number at which its bucket was split. The leaves of the tree are assumed to be arranged in an N element array, so that the singleton bucket for suffix A_p can be indexed by p. It is a straightforward exercise to build the sort history in $O(N)$ time overhead during the sort. Notice that we determine the values *height*(i) only after the sort is finished.

Given the sort history produced by the sort, we determine the *lcp* of A_p and A_q as follows. First we find the nearest common ancestor (*nca*) of suffixes A_p and A_q in the sort history using an $O(1)$ time *nca* algorithm [HT84, SV88]. The stage number H associated with this ancestor tells us that $lcp(A_p, A_q) = H + lcp(A_{p+H}, A_{q+H}) \in [H, 2H-1]$. We then recursively find the *lcp* of A_{p+H} and A_{q+H} by finding the *nca* of suffixes A_{p+H} and A_{q+H} in the history, and so on, until an *nca* is discovered to be the root of the history. At each successive level of the recursion, the number of the *nca* is at least halved, and so the number of levels performed is $O(\log L)$, where L is the *lcp* of A_p and A_q. Because the longest repeated substring has expected length $O(\log_{|\Sigma|} N)$, the $N-1$ *lcp* values of adjacent sorted suffixes are found in $O(N \log\log N)$ expected time.

The scheme above can be improved to $O(N)$ expected time by strengthening the induction basis as was done for the sort. Suppose that we stop the recursion above when the stage number of an *nca* becomes less than $T' = \lfloor ½ \log_{|\Sigma|} N \rfloor$. Our knowledge of the expected maximum *lcp* length implies that, on average, only three or four levels are performed before this condition is met. Each level takes $O(1)$ time, and we are left having to determine the *lcp* of two suffixes, say A_p and A_q, that is known to be less than T'. To answer this final *lcp* query in constant time, we build a $|\Sigma|^{T'}$-by-$|\Sigma|^{T'}$ array *Lookup*, where $Lookup[Int_{T'}(x), Int_{T'}(y)] = lcp(x, y)$ for all T'-symbol strings x and y. By the choice of T' there are no more than N entries in the array and they can be computed incrementally in an $O(N)$ preprocessing step along with the integer encodings $Int_{T'}(A_p)$ for all p. So for the final level of the recursion, $lcp(A_p, A_q) = Lookup[Int_{T'}(A_p), Int_{T'}(A_q)]$ may be computed in $O(1)$ time via table lookup. In summary, we can compute the lcp between any two suffixes in $O(1)$ expected time, and so can produce the *lcp* array in $O(N)$ expected time.

The technique of using integer encodings of $O(\log N)$-symbol strings to speedup the expected preprocessing times, also provides a *pragmatic* speedup for searching. For any $K \leq T$, let $Buck[k] = \min\{i : Int_K(A_{Pos[k]}) = i\}$. This bucket array contains $|\Sigma|^K$ non-decreasing entries and can be computed from the ordered suffix array in $O(N)$ additional time. Given a word W, we know immediately that L_W and R_W are between $Buck[k]$ and $Buck[k+1]-1$ for $k = Int_K(W)$. Thus in $O(K)$ time we can limit the interval to which we apply the search algorithm proper, to one whose average size is $N/|\Sigma|^K$. Choosing K to be T or very close to T, implies that the search proper is applied to an $O(1)$ expected-size interval and thus consumes $O(P)$ time in expectation *regardless* of whether the algorithm of Figure 1 or 2 is used. While the use of bucketing does not asymptotically improve either worst-case or expected-case times, we found this speedup very important in practice.

6. Practice

A primary motivation for this paper was to be able to efficiently answer on-line string queries for very long genetic sequences (on the order of one million or more symbols long). In practice, it is the space overhead of the query data structure that limits the largest text that may be handled. Suffix trees are quite space expensive, requiring roughly 16 bytes of overhead per text character. Utilizing an appropriate blend of the suffix array algorithms given in this paper, we developed an implementation requiring 5 bytes of overhead per text character whose construction and search speeds are competitive with suffix trees.

There are three distinct ways to implement a data structure for suffix trees, depending on how the outedges of an interior vertex are encoded. Using a $|\Sigma|$-element vector gives a structure requiring $8N+4(|\Sigma|+2)\cdot I$ bytes where I is the number of interior nodes in the suffix tree. Encoding each set of outedges with a binary search tree requires $8N+20I$ bytes. Finally, encoding each outedge set as a linked list requires $8N+16I$ bytes. The parameter $I<N$ varies as a function of the text. The first four columns of Table 1 illustrates the value of I/N and the per-text-symbol space consumption of each of the three coding schemes. These results suggest that the linked scheme is the most space parsimonious. We developed a tightly coded implementation of this scheme for the timing comparisons with our suffix array software.

For our practical implementation, we chose to build just a suffix array and use the radix-N initial bucket sort described in Section 5 to build it in $O(N)$ expected time. Without the *lcp* array the search must take $O(P\log N)$ worst-case time. However, keeping variables l and r as suggested in arriving at the $O(P+\log N)$ search, significantly improves search speed in practice. We further accelerate the search to $O(P)$ expected time by using a bucket table with $K=\log_{|\Sigma|}N/4$ as described in Section 5. Our search structure thus consists of an N integer suffix array and a $N/4$ integer bucket array, and so consumes only 5 bytes per text symbol (assuming an integer is 4 bytes).

Table 1 summarizes a number of timing experiments on texts of length 100,000. All times are in seconds and were obtained on a VAX 8650 running UNIX. Columns 6 and 7 give the times for constructing the suffix tree and suffix array, respectively. Columns 8 and 9 give the time to perform 100,000 successful queries of length 20 for the suffix tree and array, respectively. In synopsis, suffix arrays are 3–10 times more expensive to build, 2–5 times more space efficient, and can be queried at speeds comparable to suffix trees.

References

[Apo85] Apostolico, A., "The myriad virtues of subword trees," *Combinatorial Algorithms on Words* (A. Apostolico & Z. Galil, eds.), NATO ASI Series F: Computer and System Sciences, Vol. 12, Springer-Verlag (1985), 85–96.

[AI86] Apostolico, A., and C. Iliopoulos, "Parallel log-time construction of suffix trees," CSD TR 632, Dept. of Computer Science, Purdue University (1986).

[AP83] Apostolico, A. and F.P. Preparata, "Optimal off-line detection of repetitions in a string," *Theoretical Computer Science* 22 (1983), 297–315.

[AP85] Apostolico, A. and F.P. Preparata, "Structural properties of the string statistics problem," *Journal of Computer and System Science* 31 (1985), 394–411.

	Space (Bytes/text symbol)					Construction Time		Search Time	
		S.Trees							
	I/N	Link	Tree	Vector	*S.Arrays*	*S.Trees*	*S.Arrays*	*S.Trees*	*S.Arrays*
Random ($\|\Sigma\|$=2)	.99	23.8	27.8	19.8	5.0	2.6	7.1	6.0	5.8
Random ($\|\Sigma\|$=4)	.62	17.9	20.4	18.9	5.0	3.1	11.7	5.2	5.6
Random ($\|\Sigma\|$=8)	.45	15.2	17.0	20.8	5.0	4.6	11.4	5.8	6.6
Random ($\|\Sigma\|$=16)	.37	13.9	15.4	30.6	5.0	6.9	11.6	9.2	6.8
Random ($\|\Sigma\|$=32)	.31	13.0	14.2	46.2	5.0	10.9	11.7	10.2	7.0
Text ($\|\Sigma\|$=96)	.54	16.6	18.8	220.0	5.0	5.3	28.3	22.4	9.5
Code ($\|\Sigma\|$=96)	.63	18.1	20.6	255.0	5.0	4.2	35.9	29.3	9.0
DNA ($\|\Sigma\|$=4)	.72	19.5	22.4	25.2	5.0	2.9	18.7	14.6	9.2

Table 1: Empirical results for texts of length 100,000.

[BB86] Blumer, J. and A. Blumer, "On-line construction of a complete inverted file," UCSC-CRL-86-11, Dept. of Computer Science, University of Colorado (1986).

[BBE85] Blumer, J., Blumer, A., Ehrenfeucht, E., Haussler, D., Chen, M.T., and J. Seiferas, "The smallest automaton recognizing the subwords of a text," *Theoretical Computer Science*, (1985).

[Car75] Cardenas, A.F., "Analysis and performance of inverted data base structures," *Comm. of the ACM* 18, 5 (1975), 253–263.

[CHM86] Clift, B., Haussler, D., McConnell, R., Schneider, T.D., and G.D. Stormo, "Sequence landscapes," *Nucleic Acids Research* 4, 1 (1986), 141–158.

[EH86] Ehrenfeucht, A. and D. Haussler, "A new distance metric on strings computable in linear time," UCSC-CRL-86-27, Dept. of Computer Science, University of Colorado (1986).

[FWM84] Fraser, C., Wendt, A., and E.W. Myers, "Analyzing and compressing assembly code," *Proceedings of the SIGPLAN Symposium on Compiler Construction* (1984), 117–121.

[Gal85] Galil, Z., "Open problems in stringology," *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), NATO ASI Series F: Computer and System Sciences, Vol. 12, Springer-Verlag (1985), 1–8.

[Go89] Gonnet G., Private communication.

[HT84] Harel, D. and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing* 13 (1984), 338–355.

[KGO83] Karlin S., Ghandour G., Ost F., Tavare S., and L. J. Korn, "New approaches for computer analysis of nucleic acid sequences," *Proc. Natl. Acad. Sci. USA*, **80**, (September 1983), 5660–5664.

[KMR72] Karp R. M., R. E. Miller, and A. L. Rosenberg, "Rapid identification of repeated patterns in strings, trees and arrays," *Fourth Annual ACM Symposium on Theory of Computing*, (May 1972), pp. 125–136.

[LV86] Landau, G.M. and U. Vishkin, "Introducing efficient parallelism into approximate string matching," *Proc. 18th ACM Symposium on Theory of Computing* (1986), 220–230.

[McC76] McCreight, E.M., "A space-economical suffix tree construction algorithm," *Journal of the ACM* 23 (1976), 262–272.

[MR80] Majster, M.E., and A. Reiser, "Efficient on-line construction and correction of position trees, *SIAM Journal on Computing* 9, 4 (1980), 785–807.

[Mye88] Myers, E.W., "Incremental alignment algorithms and their applications," *SIAM Journal on Computing*, accepted for publication.

[ME89] Myers, E.W., and A. Ehrenfeucht, *A sub-linear algorithm for similarity searching*, in preparation.

[RPE81] Rodeh, M., Pratt, V.R., and S. Even, "Linear algorithm for data compression via string matching," *Journal of the ACM* 28, 1 (1981), 16–24.

[Sli80] Slisenko, A.O., "Detection of periodicities and string-matching in real time," *Journal of Soviet Mathematics* 22, 3 (1983), 1316–1387; translated from Zpiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V.A. Steklova AN SSSR, 105 (1980), 62–173.

[SV88] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," *SIAM Journal on Computing*, **17** (December 1988), pp. 1253–1262.

[Wei73] Weiner, P., "Linear pattern matching algorithm," *Proc. 14th IEEE Symposium on Switching and Automata Theory* (1973), 1–11.