Optimal Logarithmic Time Randomized Suffix Tree Construction

Martin Farach*
Rutgers University

S. Muthukrishnan†
Univ. of Warwick

November 10, 1995

Abstract

The suffix tree of a string, the fundamental data structure in the area of combinatorial pattern matching, has many elegant applications. In this paper, we present a novel, simple sequential algorithm for the construction of suffix trees. We are also able to parallelize our algorithm so that we settle the main open problem in the construction of suffix trees: we give a Las Vegas CRCW PRAM algorithm that constructs the suffix tree of a *binary string* of length n in $O(\log n)$ time and $O(n)$ work with high probability. In contrast, the previously known work-optimal algorithms, while deterministic, take $\Omega(\log^2 n)$ time.

We also give a work-optimal randomized comparison-based algorithm to convert any string over an unbounded alphabet to an equivalent string over a binary alphabet. As a result, we obtain the *first work-optimal algorithm* for suffix tree construction under the unbounded alphabet assumption.

*Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA. (*farach@cs.rutgers.edu*, *http://www.cs.rutgers.edu/∼farach*) Supported by NSF Career Development Award CCR-9501942. Work done while this author was a Visitor Member of the Courant Institute of NYU.

†muthu@dcs.warwick.ac.uk; Partly supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648. Partly supported by Alcom IT.

1 Introduction

Given a string s of length n, the *suffix tree* T_s of s is the compacted trie of all the suffixes of s. For many reasons, this is the fundamental data structure in combinatorial pattern matching. It has a compact $O(n)$ space representation which has several elegant uses [Apo84]. Furthermore, Weiner [Wei73], who introduced this powerful data structure, showed that T_s can be constructed in $O(n)$ time. This sequential construction and its analysis are nontrivial. Considerable effort has been put into producing simplified $O(n)$ time suffix tree constructions, however, to date, though innovative, all such algorithms are variants of one another and continue to rely on the classical approaches of Weiner [Wei73] and McCreight [McC76]. Efficient parallel construction of suffix trees has also proven to be a challenge. In fact, a classical open problem in stringology is to design a fast, work-optimal parallel algorithm for constructing this data structure. While some recent progress has been made in designing work-optimal algorithms (at the expense of the running time), in this paper, we settle the following open problem: our *main result* is a novel simplified suffix tree contruction which when parallelized yields an $O(\log n)$ time, $O(n)$ space and $O(n)$ work Las Vegas type suffix tree construction algorithm. We believe that our algorithm, which departs completely from the Weiner/McCreight approach, is interesting in both its sequential and parallel versions.

In what follows, we will explain our results in more detail before giving a technical overview of our algorithms.

1.1 Our Results.

Given the versatility of suffix trees, it is not surprising that they have been extensively studied. Since Weiner presented his classical construction, many simplified constructions have been developed [CS85, McC76]. Sequential construction of suffix trees continues to be an active area of research [DK95, Kos94].

All these algorithm are inherently sequential since they scan the string in monotonic order, updating the partial data structure as each symbol is scanned. In [AIL$^+$88], it is shown how to build a suffix tree in $O(\log n)$ time and $O(n \log n)$ work. This algorithm is not optimal; moreover, it uses polynomial space.[1] Designing work-optimal algorithms for this problem remained a challenge. Recent results have shown that one can achieve work optimality at the expense of the running time. In [SV94], a work optimal $O(\log^2 n)$ time algorithm was presented using polynomial space. In [Har94] a different work-optimal algorithm is presented which uses linear space; this algorithm takes $O(\log^4 n)$ time[2].

In this paper, we settle the following double challenge: our main result is a novel suffix tree algorithm whose sequential version is quite simple and whose (considerably non-trivial) parallelization yields an $O(\log n)$ time and $O(n)$ work parallel algorithm for suffix tree construction that uses $O(n)$ space. Our algorithm is randomized of the Las Vegas type.

An additional issue of string processing interest is that of *alphabet dependence*. String processing algorithms that work under the *unbounded alphabet* assumption access the string using only order comparisons. Sequential suffix tree construction algorithms that work under this model take

[1] Throughout, we will use "polynomial space" to mean $\Theta(n^{1+\epsilon})$ space for some $\epsilon > 0$.

[2] This algorithm works on the CREW PRAM while all other algorithms mentioned in this paper, including ours, use the stronger Arbitrary CRCW PRAM.

$\Theta(n \log \sigma)$ time if the number of distinct symbols in s is σ; in contrast, the best known parallel algorithms in this model do $\Theta(n \log n)$ work and are therefore sub-optimal. In this paper, we give a simple randomized work/time optimal reduction from the case when alphabet set is unbounded to the case when it is binary. Our reduction works in the ordered-comparison model. Consequently we derive the first known work-optimal algorithms for suffix tree construction under the unbounded alphabet assumption. Our reduction above can be used with any of the work optimal algorithms of [SV94, Har94] for binary strings, however using it with our algorithm in this paper gives the fastest work optimal suffix tree construction algorithm under the unbounded alphabet assumption.

1.2 Technical Overview.

Suffix tree construction for binary strings. Let $s \in \{0,1\}^*$. Then, the suffix tree of s is the compacted trie[3] of all the suffixes of $s\$$ where $\$ \neq 0, 1$. At a high level, our algorithm works as follows. First we recursively determine the *odd tree*, T_o, which is the compacted trie of all the suffixes beginning at the odd positions. From the odd tree, we then derive the *even tree*, T_e, the compacted trie of all the suffixes at the even positions. Finally, we merge T_o and T_e to obtain the suffix tree of s. The technical crux of this approach in both sequential and parallel setting lies in merging the trees.

The approach of constructing the suffix tree by merging the odd and even trees, one of which is recursively constructed, is not new. It was used in a modified form in [SV94] to obtain an $O(\log^2 n)$ time work-optimal suffix tree construction. There too merging two partial suffix trees (such as T_o and T_e) formed the technical crux. There the authors developed a novel symmetry-breaking naming scheme to solve that problem; however their algorithm for merging the two trees takes $\Omega(n \log^* n)$ work[4]. That was sufficient for them to obtain an work optimal algorithm for suffix tree construction (taking $O(\log^2 n)$ time) by relying on the Four Russian's technique of constructing certain exhaustive tables for "short" strings.

In this paper, we directly solve the problem of merging the two partial suffix trees T_o and T_e by using several structural properties of these trees which we identify and prove here. Our algorithm for this takes $O(\log n)$ time and $O(n)$ work and is therefore work-optimal. Using this algorithm as such to construct suffix trees in parallel we can get an algorithm that takes $O(\log n \log\log n)$ time and $O(n)$ work. Note that this is already the fastest work-optimal algorithm known for this problem. However to get our main result, which is a $O(\log n)$ time work-optimal algorithm for this problem, we use additional ideas which are explained in detail in Section 3. We merely note that in order to achieve this result we had to reüse the structural properties of the partial trees we identified to design an algorithm for merging them in only $O(\frac{\log n}{\log\log n})$ time (rather than the $O(\log n)$ time quoted above), again in $O(n)$ work, provided the Euler tours of the two trees are given.

Finally, as a curiosity, we mention that our parallel algorithm makes use of a number of tools that have been developed in the parallel algorithms community recently – see the list of lemmas in Section A. However, our sequential algorithm does not need these tools and is very simple.

[3] A compacted trie differs from a trie in that degree 2 paths are replaced by edges labeled by the appropriate substring.

[4] $\log^* n = \min\{i | \log^{(i)} n \leq 2\}$.

Suffix tree construction for strings over unbounded alphabet. A standard approach to cope with the strings drawn from an unbounded alphabet is to map the distinct symbols of the string to a small alphabet and then use the suffix tree construction algorithm that works for strings over small alphabet. We can thus formalize the following *renaming* problem. Given an array of size n with σ distinct elements, map distinct elements in the array to range $1 \cdots \sigma$ in the (ordered) comparison model; here σ is unknown at the beginning. Renaming is a generic reduction from string problems over unbounded alphabet to those over (integer) alphabet $1 \cdots \sigma$.

Sequentially renaming can be done optimally using a balanced binary tree such as a 2–3 tree. Known parallel algorithms for 2–3 trees [PVW83], however, do not give optimal parallel bounds for renaming. The best known algorithms for renaming use parallel sorting and therefore take $O(n \log n)$ work and $O(\log n)$ time. In this paper, we present the first known simultaneously time and work-optimal algorithm for renaming. Our algorithm is randomized and of Las Vegas type; it takes $O(\log \sigma)$ expected time and $O(n \log \sigma)$ expected work.

Our algorithm is obtained by random sampling reminiscent of randomized parallel sorting algorithms [Rei81]. Since σ is unknown and we seek an algorithm with time dependent on only σ, considerable technical difficulties arise in our setting which we overcome.

Map. We describe the sequential version of our algorithm in Section 2. We describe our parallel suffix tree construction in Section 3. We have only sketched the details in most of the steps, and the reader is refered to the appendices for these.

2 The Sequential Algorithm

We will use one basic tool: KR naming (Lemma A.1). With this tool, we can check the equality of any two substrings in constant time. Now our algorithm has three main steps:

Compute T_o: To construct the odd tree T_o recursively, we make the following observation. Consider a string s' generated from s in which $s'[i] = f(s[2i-1], s[2i])$ where f is the randomized fingerprint function of Lemma A.1. Let T' be the suffix tree of s'. Then, T_o is closely related to T'. The difference is the following. Consider any two odd suffixes. The least common ancestor (`lca`) of their corresponding leaves in T_o gives their longest common prefix, while the `lca` of their corresponding leaves in T' is the longest common prefix of *even* length. Therefore, T_o can be derived from T' by local patching in linear time. Since $|s'| = |s|/2$, this recursive procedure does not increase the complexity of our algorithm.

Compute T_e: To construct the even tree T_e, we note that every even suffix is a single character followed by an odd suffix. From T_o, we know the longest common prefix of any two odd suffixes, so we also now know the longest common prefix of any two even suffixes. To construct a tree, we only need to know an in-order traversal of the leaves and the relative depth for the `lca` of adjacent leaves. The latter we know already, and we can get the in-order traversal of the leaves by a stable integer sorting of the in-order leaves of T_o using the character preceding an odd suffix as its sort key, once again, in linear time.

Merging T_o and T_e: Merging T_o and T_e will give the suffix tree of s, so this step is the heart of the algorithm. We will merge the two trees by a coupled DFS, so at each recursive step, we will be merging a subtree of T_o with one from T_e. For purposes of illustration, we will consider subtrees rooted at edges as well as those rooted at nodes. Thus, we must describe what our algorithm does when merging pairs rooted at node/node, node/edge and edge/edge. Since we start by merging at the roots of both trees, we start with a node/node pair.

NODE/NODE: Suppose we are merging v_o and v_e. The only relevant information is the first character on the edges leading to their children. For any children c_i of v_o and c_j of v_e which begin with the same character, merge their subtrees via an Edge/Edge merge. Any child whose edge begins with a unique character is not merged. Its subtree is simply included in the final tree unchanged.

NODE/EDGE: Suppose we are merging node v and edge e, and that the first character on e is c. If v does not have an edge to a child which begins in c, then root e's subtree at v. Otherwise, let d be the child of v whose edge starts with a c. Merge the edge to d with e and root the resulting tree at v. Leave all other subtrees below v unchanged.

EDGE/EDGE: In both of the above cases, we recursively merge two trees rooted at edges. Let e_o and e_e be the edges we are merging. We have two cases. First, suppose the string of one edge is a proper prefix of the string of the other edge. WLOG, we will assume e_o is a prefix of e_e. We can check this in constant time via fingerprints. Then we add a node b in e_e after $|e_o|$ characters. Now, let d be the node at the bottom of e_o. We know that we can proceed by merging b and d, as above.

The other case is much more interesting. The edges e_o and e_e share a common prefix but neither is a prefix of the other. We cannot afford to find out how long the common prefix is for these two edges, since it would take $\Omega(\log n)$ to do so by direct methods, and there could be $\Omega(n)$ such pairs of edges to merge. Instead of directly determining this longest string, we will introduce a *refinement node* into e_o and e_e which will act as a place-holder. The placement of the refinement nodes within the edge pair gives us enough information to complete the structural merging of the even and odd trees. After placing the refinement node, we have bottomed out the recursion.

REFINEMENT: However, we have yet to determine exactly where the edges need to be "broken" to introduce the refinement node; in other words, the refinement nodes can be thought of as sliding nodes whose exact placement we defer until after merging the trees.

Consider some refinement node r. By construction, it has out-degree 2, and one of its descendant trees, t_o comes from the odd tree, while the other, t_e, comes from the even tree. Let l_i be an arbitrary leaf in t_o and l_j be an arbitrary leaf in t_e, where, for any k, l_k represents the kth suffix of s. We can *restate* the problem of placing the refinement node within the edge pair as that of determining the longest common prefix of l_i and l_j. Now, we can conclude that the length of r (which is all we are after) is 1 more than the length of $\texttt{lca}(l_{i+1}, l_{j+1}) = v$. But we may not know the length of v if v is itself a refinement node. So we see that $|r| = 1 + |v|$, and conclude that we have a dependency tree (more generally, a forest) between the length of various nodes in the tree. We define $d(r) = v$ where r and v are as above. The length of r depends on its depth in the tree defined by $d(\cdot)$. Finally, we

determine, by DFS on the d tree, the depth of all refinement nodes in their trees and thus deduce their lengths.

The above coupled-DFS merging procedure, along with the final DFS refinement procedure take $O(n)$ time, thus completing our construction. The output of this algorithm is correct with high probability. In Appendix D, we show an algorithm that checks the correctness of the output in $O(\log n)$ time and $O(n)$ work on a PRAM, and hence is sufficient to check the output of this algorithm, as well as the following parallel algorithm.

3 The Parallel Algorithm

The parallel algorithm has the same three basic steps as the sequential algorithm in Section 2. However, we now have the added constraint that we want an $O(\log n)$ time and $O(n)$ work algorithm. The sequential algorithm naturally decomposes into $\log n$ recursive phases, and each phase seems to take $\Omega(\log n)$ time, since basically any tree manipulation has list ranking as bottle-neck. In addition, we will need a radically different approach to merging the two trees since a DFS based approach is not well suited to parallelization.

Given this scheme, we make the following three contributions.

1. We provide a work-optimal algorithm for merging the odd and even trees. This step is the crux of our technical contribution and it relies on several structural properties of the odd and even suffix trees that we isolate and prove (See Section 3.1). Our algorithm works in $O(\log n)$ time w.h.p. Combined with a straightforward $O(\log n)$ time algorithm for deriving the even tree from the odd tree, this gives a work-optimal $O(\log^2 n)$ time suffix tree algorithm. Note that this already provides an alternate algorithm matching the running time of the fastest known optimal algorithm for the suffix tree construction problem [SV94], while taking only linear space.

2. Consider the following lemma:

Lemma 3.1 *There exists an algorithm to compute the suffix tree of a string of length n in $O(\log n)$ time, $O(n \log n)$ work w.h.p. and $O(n \log^2 n)$ space.*

Proof: The algorithm in [AIL$^+$88] uses a bulletin board which can be replaced by the constant time hashing scheme of Lemma A.3. ∎

We may now bottom out the recursion after $O(\log \log n)$ levels. We use this lemma to derive a $O(\log n \log \log n)$ time suffix tree construction algorithm, while still preserving the linear work and space. We note that this "bottoming out" cannot be done in the known work-optimal constructions of either [Har94] or [SV94] to speed up their algorithms.

3. Each step of our recursion takes $\Theta(\log n)$ time. This is not surprising since we perform several tree manipulation operations at each stage, each of which relies on computing Euler tours. Computing the Euler tour in turn has the well-known bottleneck of list ranking, which takes $\Theta(\log n)$ time as noted above. We overcome this bottleneck in our algorithm by *introducing the key idea of maintaining the Euler tour* (rather than the tree itself) through the levels of recursion.

This, on the one hand, implies that Euler tours need not be computed at each level and therefore the $\Theta(\log n)$ bottleneck is averted. On the other hand, maintaining Euler tours and manipulating them give rise to new problems, which can be illustrated as follows. Suppose we have a tree and its Euler tour and we wish to reörder the children at each node (as we are required to do in our

algorithm). In a tree representation, such a rearrangement is a purely local computation at the nodes. In the Euler tour, however, such a change involves global reörganization. We demonstrate how to perform this reörganization as well as the other tasks at each level in $O(\log n/\log\log n)$ time with optimal work. This in turn gives our main result, namely:

Theorem 3.2 *Given a binary string s of length n, its suffix tree can be computed in $O(\log n)$ time and $O(n)$ work w.h.p. using $O(n)$ space.*

In what follows, we describe how each of the three steps is implemented so as to derive the Euler tour of the suffix tree of s. In fact, *all trees will be assumed to be in their Euler tour representation.* Note that it is straightforward to generate the pointer version of the suffix tree given its Euler tour in $O(\log\log n)$ time and $O(n)$ work.

Compute T_o: Derive s' from s as in the sequential version, and recurse to produce T'. Deriving T_o from T' is technically quite difficult since T' is in an Euler Tour representation. We make this transformation via the procedure UNNAME. We will show the following in Appendix B.

Lemma 3.3 *Generating s' can be done in $O(1)$ time and $O(n)$ work. Procedure* UNNAME *can be implemented in $O(\log n/\log\log n)$ time and $O(n)$ work w.h.p. if s is drawn from an alphabet of size $O(\text{polylog } n)$ and in $O(\log n/\log\log n)$ time and $O(n\log\log n)$ work w.h.p. if it is drawn from an alphabet of size $O(\text{poly } n)$; in both cases, the space is linear.*

By recursing for only $2\log\log n$ levels, we get:

Lemma 3.4 *All calls to procedure* UNNAME *over all recursive levels take $O(\log n)$ time and $O(n)$ work w.h.p. and use $O(n)$ space.*

Proof: Note that at every recursive level s' is at most half the length of s. Also, if the alphabet size of s is σ, then the alphabet size of s' is at most σ^2. At each stage, we hash the tuples of s' to ensure that in fact the alphabet is in the range $1\ldots\sigma^2$. For the first $\log\log\log n$ levels, the alphabet size of the strings remains polylogarithmic and therefore, by Lemma 3.3, the work performed during this step is linear. At this stage, the string under consideration is of length at most $O(n/\log\log n)$. Therefore, we can afford to use the suboptimal large alphabet version of the procedure UNNAME up to $2\log\log n$ levels while still performing only linear work. Note that our alphabet never grows bigger than n. ∎

Compute T_e: The sequential version is easy to parallelize. We need only be careful about parallel stable integer sorting.

Lemma 3.5 *Procedure* ODD-EVEN *can be implemented in $O(\log n/\log\log n)$ time and $O(n)$ work w.h.p. if s is drawn from an alphabet of size $O(\text{polylog } n)$ and in $O(\log n/\log\log n)$ time and $O(n\log\log n)$ work w.h.p. if it is drawn from an alphabet of size $O(\text{poly } n)$; in both cases, the space is linear. All calls to procedure* ODD-EVEN *over all recursive levels take $O(\log n)$ time and $O(n)$ work w.h.p. and use $O(n)$ space.*

Merging T_o and T_e: We change the focus of our algorithm from the sequential version. In the sequential algorithm, we considered merging subtrees rooted at nodes or edges. In the parallel version, we will consider the set of strings which can be obtained by tracing down from the root from a trie. We can partition this set into *node strings*, which are strings that end at nodes, and *edge strings*, which are strings that end between two nodes. Our parallel algorithm will rely on finding strings which are shared by both tries and then performing the actual structural merging.

The node string/edge string bipartition means that strings shared by the two tries come in three flavors: *node-node*, *node-edge*, and *edge-edge*. As it turns out, common strings of the node-node flavor are easy to find, and node-edge strings require only a little more work. As in the sequential case, the true challenge is to find common strings of the edge-edge flavor. We will end up once again with refinement nodes, but will need to avoid computing the depth in the d tree, since this would take $O(\log n)$ time. We will present an algorithm for refinement node computation which takes only $O(\alpha(n))$ time[5] while still performing $O(n)$ work (For details, see Section 3.1).

Summarizing the discussion above, we need three procedures: NODE-NODE, NODE-EDGE and EDGE-EDGE. In addition, we need a procedure to generate the merged tree once we have found common strings. We call this last procedure MERGE-PATHS and use these four together to implement MERGE-TREES. See Section 3.1.

Lemma 3.6 *Procedure* MERGE-TREES *can be implemented in $O(\log n/\log\log n)$ time and $O(n)$ work w.h.p. if s is drawn from an alphabet of size $O(\text{polylog } n)$ and in $O(\log n/\log\log n)$ time and $O(n\log\log n)$ work w.h.p. if it is drawn from an alphabet of size $O(\text{poly } n)$; in both cases, the space is linear. All calls to procedure* MERGE-TREES *over all recursive levels take $O(\log n)$ time and $O(n)$ work w.h.p. and use $O(n)$ space.*

From the preceding sequence of lemmas, we can conclude Theorem 3.2. In what follows, we will describe one of the important procedures, namely that for tree merging.

3.1 Tree Merging

Recall that procedure MERGE-TREE is implemented by first finding common strings represented by the odd and even trees (using procedures NODE-NODE, NODE-EDGE, and EDGE-EDGE) and finally building the merged tree around this core of common strings (using procedure MERGE-PATHS). Note that any prefix of a common string is also a common string, thus, to represent all common strings, it suffices to determine common strings which are *maximal*. In the discussion that follows, such maximal strings will be referred to as *explicit* strings and their prefixes will be referred to as *implicit*. Procedure MERGE-TREE relies on the following analysis of strings common to both trees.

The simplest of common strings is a Node-Node string. Define an *anchor pair* to be a pair (u, v), $u \in \mathtt{V}(T_o)$, $v \in \mathtt{V}(T_e)$ such that their strings are the same; each such node is called an *anchor*.

Property 1 *For any two anchor pairs (u_1, v_1) and (u_2, v_2), $(\mathtt{lca}(u_1, u_2), \mathtt{lca}(v_1, v_2))$ is also an anchor pair.*

[5]Here $\alpha(\cdot)$ is the inverse Ackerman function.

Consider any node in one of the trees which has an anchor descendant. Clearly the string that corresponds to this node is represented in the other tree as well. Therefore, the set of all anchor pairs implicitly defines all such common strings. So in searching for Node-Edge strings, we need only consider nodes with no anchor descendants.

Let a *side tree* be a maximal component such that no node in the side tree is an anchor node or has an anchor descendant. Define a *side tree pair* to be a pair of side trees such that the parents of their roots form an anchor pair, and the first character on the edge for the anchor pair to the roots is the same. Node-Edge strings which occur in side trees must occur in a side tree pair. Within a side tree pair, define a node to be *active* if its string is a Node-Edge string, that is, if it occurs in an edge of the paired side tree. It can be easily shown that the least common ancestor of two active nodes is an anchor unless one is an ancestor of the other. Since there are no anchors in a side tree, this implies the following property.

Property 2 *Active nodes within a single side tree are linearly ordered; in fact, they form a path from the root of the side tree.*

Fix a side tree pair and consider all its active nodes. Let u be the active node with the longest string in either of the trees. Let $\overline{vw}$ be the edge in the other tree such that the string of v is a prefix of the string of u which is a prefix of the string of w. Suppose we insert a node u' into the edge $\overline{vw}$ such that the string of u' equals the string of u. We will call such a pair (u, u') a *pseudo-anchor pair* and call each node a *pseudo-anchor*. Observe that once we have created a pseudo-anchor pair, it implicitly represents *all* Node-Edge strings in its side tree pair.

It remains to consider Edge-Edge strings. In fact, we need only consider those which are not implicitly represented by anchors or pseudo-anchors. These Edge-Edge strings occur within edge pairs that have the following property: either the edge pairs come off pseudo-anchor pairs, or they are the edges connecting the roots of a side tree pair to their anchor pair parents, in the case where there are no active nodes in that side tree pair. Recall that the side tree pairs were limited to hang off of anchor nodes; the stated property for edge pairs gives a similarly useful restriction to the location of relevant Edge-Edge pairs.

Finally, we must find the longest shared Edge-Edge string within an edge pair, since such a string implicitly represents *all* Edge-Edge strings within an edge pair. As before, we will introduce a refinement node into each edge. The placement of the refinement nodes within the edge pair gives us enough information to complete the structural merging of the even and odd trees, and as before, we finish merging the trees before resolving the lengths of the strings at refinement nodes. Just as in the sequential algorithm, we can define and construct $d(r)$ for any refinement node r, and as before, the length of r depends on its depth in the tree defined by $d(\cdot)$. It would appear that such a depth computation takes $O(\log n)$ time by tree contraction. We circumvent this by the following observation. Define a refinement node r to be *deep* if $d(r)$ is a refinement node; otherwise, it is *shallow*.

Property 3 *If deep node r has leaf descendant l_i, then the refinement node ancestor of l_{i+1} is $d(r)$.*

Thus, finding the depth of r in the $d(\cdot)$ tree is reduced to finding the smallest $j > i$ such that the refinement node ancestor of l_j is shallow. The linear ordering of the suffixes allows us to find such a

j for all i in $o(\log n)$ time with optimal work. See procedure REFINE in the implementation details below.

Implementation Details. To implement MERGE-TREE we need only implement procedures NODE-NODE, NODE-EDGE, EDGE-EDGE, MERGE-PATHS, and REFINE.

• NODE-NODE: We hash the fingerprints of the node strings in one tree and look up the fingerprints of the node strings of the other tree. This gives the Node-Node strings. Time: $O(\log^* n)$.
• NODE-EDGE: Since the active nodes are linearly ordered, searching for the pseudo-anchor is reminiscent of finding the leftmost 1 in a 0/1 array. The tree structure adds complications which we overcome by random sampling of the leaves. Time: $O(1)$.
• EDGE-EDGE: Once we have found the pseudo-anchors, inserting the refinement nodes is trivial. Time: $O(1)$.
• MERGE-PATHS: We resort to techniques which we use to implement UNNAME and which will be outlined in Appendix B. The crucial lemma is Lemma B.1. This step requires integer sorting and thus is both the time and work bottleneck (Lemma A.2).
• REFINE: We easily compute the d pointers and have each refinement node determine if it is shallow or deep. Each leaf l_i determines if it has a shallow ancestor, and if so, marks $A[i] = 1$ in some work array A. All other positions of A get marked with 0. Now an all-nearest-ones (Lemma A.5) computation gives the depth of each refinement node in the d tree. Time: $O(\alpha(n))$.

Taken together we achieve the bounds stated in Lemma 3.6. Proofs of the properties mentioned in this section as well as technical details of the implementation are omitted here.

3.2 Unbounded Alphabet Suffix Tree Construction

As mentioned earlier, we solve the *renaming problem* which is a generic reduction from strings over unbounded alphabet to binary strings. Specifically, we will show the following result. Given an array A of size n consisting of $\sigma \leq n$ elements from an ordered universe, we provide a randomized algorithm RENAME which outputs a sorted array containing only the *distinct* elements of A; this algorithm takes $O(\log \sigma)$ time and $O(n \log \sigma)$ work w.h.p. An n character string can then be represented as an $n \log \sigma$ bit string, and the suffix tree construction algorithm for binary strings can then be applied; we omit the details for accomplishing this.

The renaming problem can be recast into a broader setting by viewing it as a form of input-sensitive sorting. In regular sorting each element is assigned its overall rank while in input-sensitive sorting, each element is assigned its rank from amongst the distinct elements. This difference is crucial since the former problem has a lower bound of $\Omega(\log n / \log \log n)$ time even when $\sigma = 2$ [BH87]; in contrast, our algorithm for renaming takes $O(1)$ time for any constant σ.

In the sequential comparison model, renaming takes $O(n \log \sigma)$ time, even though σ is unknown. In the PRAM model, not knowing σ implies that no more than $O(n)$ processors can be allocated for a problem instance of size n, and a lower bound of $\Omega(\log \sigma)$ on the running time for any work-optimal algorithm for this problem follows. Thus the algorithm we present here is in fact *both time and work optimal* for the renaming problem.

Now we present a sketch of our algorithm.

Lemma 3.7 *Given an array A of size $\sqrt{n}$ with k distinct elements, we can* RENAME *A in $O(\log k)$ time and $O(n \log k)$ work.*

Lemma 3.8 *Given an array A of size n, a sample S in which each element from A is chosen independently and randomly with probability p, with K being the set of distinct elements from S, let $k = |K|$ and let $A - K$ be the compacted array in which all elements from K are removed from array A. Then, $A - K$ is of size $O(\frac{k \log n}{p})$ w.h.p.*

If we choose a (quadratically) small sample from our input array, we can rename it within our target time bounds using Lemma 3.7. As a byproduct of such a procedure, we can get a lower bound on k, the number of distinct elements. Further, if we find that this lower bound is large (say $\Omega(n^{1/4})$), then we can sort the array directly, via Cole's Mergesort [Col86], and determine the output by directly computing the rank of the distinct elements via prefix sums. Otherwise, the lower bound is small and since we prove in Lemma 3.8 that almost all elements of the input array are represented in our sample, we can recurse on the (proportionally) smaller surviving set by eliminating those represented in the sample from further consideration. We will show that *we terminate after only a constant number of rounds* of this procedure. Full details are in the Appendix E. Note finally that we are always compacting polynomially small subsets of array A, so this can be done in $O(1)$ time w.h.p. and linear work by Lemma A.4.

4 Discussion

In this paper, we settle a long standing open problem in stringology by giving a log time, linear work suffix tree construction algorithm. Just as importantly, we have given an algorithm with a very simple sequential version, one which differs considerably from the amortization based algorithm known heretofore. We leave as an open problem to give a linear work logarithmic time deterministic algorithm for this problem.

Although several sequential uses of suffix trees are known, optimal parallel algorithms for these applications of suffix trees are open. Only recently optimal use of suffix trees was developed for some applications, namely in dictionary matching and compression [FM95]. Using our suffix tree construction algorithms to feed the applications in [FM95] gives the fastest work optimal algorithms for the problems there. But optimal parallel usability of suffix trees for other applications remains to be explored.

References

[AIL+88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Scieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[Apo84] Alberto Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlag, Berlin, 1984.

[BBG+89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[BDH+91] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Raznik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, pages 29–47, 1991.

[BDH92] Bast, Dietzfelbinger, and T. Hagerup. A perfect parallel dictionary. *MFCS, LNCS, Vol. 629*, pages 133–141, 1992.

[BH87] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[BV89] Omer Berkman and Uzi Vishkin. Recursive *-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989.

[Col86] Richard Cole. Parallel merge sort. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 511–516, 1986.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[CV89] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[DK95] A. Delcher and S. Kosaraju. Large-scale assembly of dna strings and space-efficient construction of suffix trees. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*, 1995.

[FM95] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[FRW84] Faith E. Fich, Prabhakar L. Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation (preliminary version). In *Proceedings 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 179–189, 1984.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 698–710, 1991.

[Goo91] M. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 711–722, 1991.

[Hag91] T. Hagerup. Constant time parallel integer sorting. *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, pages 299–306, 1991.

[Har94] R. Hariharan. Optimal parallel suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, 1994.

[Kos94] S. Kosaraju. Real-time suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, 1994.

[KR87] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

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

[MV91] Y. Matias and U. Vishkin. Converting high probability into nearly constant time–with ap plications to parallel hashing. *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991.

[PVW83] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2-3 trees. Technical Report TR-70, Courant Institue, NYU, 1983.

[Rag90] P. Ragde. The parallel simplicity of compaction and chaining. *Proc. of 17th International Colloquium on Automata Languages and Programming, Springer LNCS 443*, pages 744–751, 1990.

[Ram90] R. Raman. The power of collision: Randomized parallel algorithms for chaining an d integer sorting. *Proc. of the 10th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, pages 161–175, 1990.

[Rei81] Rüdiger Reischuk. A fast probabilistic parallel sorting algorithm. In *Proc. of the 22nd IEEE Annual Symp. on Foundation of Computer Science*, pages 212–219, 1981.

[RR89] S. Rajasekaran and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18, 1989.

[SV94] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, 1994.

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

Appendix

A Tools

In this section we state a menu of results used throughout the paper.

Lemma A.1 ([KR87]) *Given a string s of length n, there exists a function $f(i,j)$ which can be computed in linear work and $O(\log n)$ time such that for any i,j,k, if $f(i,i+k) \neq f(j,j+k)$ then, $s[i \ldots i+k] \neq s[j \ldots j+k]$, and if $f(i,i+k) = f(j,j+k)$ then, $s[i \ldots i+k] = s[j \ldots j+k]$ with probability at least $1 - 1/n^3$.*

Lemma A.2 *Given n integers in the range $1 \ldots n \log^{O(1)} n$ there is a randomized algorithm to sort them in $O(\log n / \log\log n)$ time and $O(n)$ work [RR89, Hag91, MV91, Ram90].*

Given n integers in the range $1 \ldots n^{O(1)}$ there is a randomized algorithm to sort them in $O(\log n / \log\log n)$ time and $O(n \log\log n)$ work [BDH+91].

Given n items, there is an algorithm to comparison-sort them in $O(\log n)$ time and $O(n \log n)$ work [Col86].

Lemma A.3 ([GMV91]) *Given n integers in the range $1 \ldots n^{O(1)}$ there is a randomized algorithm to insert them into a $O(n)$ space hash table in $O(\log^* n)$ time and $O(n)$ work. Items can then be looked up in constant time using a processor each.*

Given n integers in the range $1 \ldots n^{O(1)}$ there is a randomized algorithm to insert them into a $O(n \log n)$ space hash table in $O(1)$ time and $O(n)$ work. Items can then be looked up in constant time using a processor each.

Lemma A.4 ([BDH92, Goo91]) *Given an array of size n with n^ϵ selected items for fixed positive $\epsilon \leq 1/2$, there is a randomized algorithm to compact the selected items into an array linear in the number of selected items in $O(1)$ time and $O(n)$ work.*

Lemma A.5 ([Rag90]) *Given a 0/1 array of size n, there is an algorithm to find the nearest one to the right from each position which works in $O(\alpha(n))$ time and $O(n)$ work.*

Lemma A.6 ([CV89]) *Given an array of n numbers each of $O(\log n)$ bits, their prefix sums can be computed in $O(\log n / \log\log n)$ time and $O(n)$ work.*

Lemma A.7 ([BV89]) *Given the Euler tour of a tree, there is an algorithm to preprocess the tour so that Least Common Ancestor (LCA) queries can be answered in $O(1)$ time with one processor. The preprocessing time is $O(\log n / \log\log n)$ and the work is $O(n)$.*

B Unnaming

We show how to implement procedure UNNAME so that $T_o = \text{UNNAME}(T')$. UNNAME can be implemented trivially in a tree representation of T' in $O(\log^* n)$ time, as follows. We work independently on each node and consider the first character labeling each edge. This character in T' is a pair of

characters in T_o. Say T'_o is derived from T' by substituting the appropriate tuple of characters for the first symbol on each edge of T'. Then we may note that the edges descending from each node of T'_o need not differ in the first character, so T'_o is not a well formed trie. For example, suppose that node v in T' has descendant edges whose first character are α and β. Now, to derive T'_o, we substitute for α and β their respective character tuples, which, for the sake of illustration, we take to be aa and ab. Now note that v in T'_o has two children whose edges are labeled with an a, thus violating the definition of a trie. In a pointer tree, we can just hash all edges by $\langle p, c\rangle$ pair, where p is the parent and c the first character on the edge in T'_o. All collisions in the hashing represent new nodes that must be introduced to group a set of edges. The pointers can then be reset in constant time, and the only bottleneck is hashing.

However, as noted above, we cannot afford to work in the pointer representation since other operations in this representation take $O(\log n)$ time. We instead sketch a method for unnaming in the Euler tour representation which performs within the bounds promised in Lemma 3.3.

The main difficulty in manipulating Euler tours can be seen as follows. Consider relabeling the edges in T' to get T'_o as mentioned earlier. Clearly, the children of each node which share the first symbol along the edge to each of them get grouped together. Therefore, the Euler tours of the subtrees under the children in a group occur together in T_o. However, the task is much harder than merely removing portions of T corresponding to the subtrees under the nodes in each group, and putting them together. This is because for the subtree under a given node in a group, such a rearrangement of Euler tours needs to be performed at each node within the subtree. Thus T_o is actually an appropriate rearrangement of the Euler tours of the children of T.

We rely on the following simple lemma to construct the desired Euler tour T_o.

Lemma B.1 *If, for each node v in a tree T we know d_v and l_v, where d_v is the number of descendant leaves of v and l_v is the number of leaves which are visited before v in a preörder traversal of T, then we can build the Euler tour of T work-optimally in $O(\log n/\log\log n)$ time and linear space.*

Proof: Each node v with parent p will fill in two pieces of information in the Euler tour: it will write down a v before in an array such that it comes before the Euler tour of the subtree rooted at v, and it will write a p just after the subtour of v is complete. It is easy to see that such a tour is a correct Euler tour of T. But v can estimate the locations where it must write by d_v and l_v by noting that an Euler tour of an m leaf tree is of size no more than $4m$. So, node v writes a v in position $4l_v + 1$ and writes a p in position $4d_v + 4l_v + 1$. To finish the computation, the resulting tour, which may have gaps, is compacted into a contiguous array. ∎

Therefore, to compute T_o, we need to get d_v and l_v for each node in T_o. Very briefly, we can compute d_v for all nodes in T'. Then, we sort the nodes in T'_o according to $\langle p, c\rangle$, as before. We can now add up the d_v's for all nodes with the same parent p and initial character c. This gives the l_v for all "new" nodes which we must introduce into T_o. Now, we must compute the l_v. But if each node computes how may leaves there are to the left of it *amongst its sibling*, and such a value is called ll_v, say, by prefix sums, then the l_v value of each node is the sum of the ll values for the path from the root to v.

Note that all the operations are prefix sums except for the integer sorting of the $\langle p, c\rangle$. In fact, the sorting is the bottleneck, but we resort to Lemma A.2 to obtain the desired overall bounds.

C Odd to Even

Let Σ_e denote all those symbols in s which appear at even positions. Given a set I of odd indices in s, the *skeleton tree* for I is the compacted trie of the i-suffixes where $i \in I$. Given $a \in \Sigma_e$, *a-class* comprises all indices $2i$ in s, for integral $i > 0$, such that $s[2i-1] = a$.

Notice that the tree T_e is a root node with an *a-child* for each distinct symbol $a \in \Sigma_e$. At the a-child, the skeleton tree for the a-class hangs. Therefore, the Euler tour T_e is derived from the Euler tours of each of the skeleton trees. In what follows, we describe how to generate the Euler tours for the skeleton tree for a-class, contiguously, in left-to-right increasing order of $a \in \Sigma$ in array E.

We sort the leaves into a-classes by assigning each leaf l_i the tuple $\prec a, L \succ$ where L is its location in T_o, and i is in the a-class. Within each a-class, consider two consecutive leaves l_i and l_j. Let v be their lca. We note that the lca algorithm of [BBG$^+$89] actually returns the occurrence of v in the Euler tour T_o which comes between l_i and l_j. If we compute lca's of all adjacent leaves, we get all nodes in the skeleton tree for the a-class. In fact, we get all nodes in the Euler tour of the skeleton tree except the first and last occurrence of each internal node. Finally we note that the first and last occurrence of any node in T_o is in the same relative order as in the a-class. Now the algorithm consists of sorting leaves by a-class, computing the lca's of adjacent leaves and having each internal node find its first and last occurrence in T_o. Finally, we sort all leaves and the internal nodes according to the tuple of their a-class along with their index from T_o. The bottleneck in this procedure is integer sorting. We achieve the bounds claimed in Lemma 3.5 by applying Lemma A.2.

D Checking the tree

We are given a suffix tree whose correctness we want to determine. Each node in a suffix tree represents a string, and, when the context is clear, we will refer to a node and its string interchangeably. Recall that we denote the ith suffix as l_i. Suppose that $a\alpha$ is a node in the tree, where a is a character and α is a string. Then, it is easily shown that α is also a node in the tree. Set $sl(a\alpha) = \alpha$ to be the *suffix link* of $a\alpha$. Note that $sl(l_i) = l_{i+1}$. The function sl can be easily computed by hashing the fingerprints of the nodes. For each edge $\overline{xy}$ let $substr(\overline{xy})$ be the suffix of y of length $|y| - |x|$. The function $substr$ can be computed directly from x and y.

Lemma D.1 *The input tree is correct if and only if the following hold:*
- *If y is a child of x, then $sl(y)$ is a descendant of $sl(x)$ and x and y begin with the same character.*
- *For each internal node x with children $c_1, \ldots, c_k$, then each of $substr(\overline{xc_i})$ begins with a different character.*
- *If r is the root, then $|r| = 0$. If y is a child of x, then $|y| > |x|$.*
- *For each node x, $|sl(x)| = |x| - 1$.*

Proof: Consider 2 suffixes of the given strings; let x and y be the leaves representing them. Let z be their LCA. Let u be the longest common prefix of the two suffixes. Suppose for a contradiction that $|str(z)| \neq |u|$. Let z_i (resp. x_i, y_i) be the nodes suffix of z (resp x, y) of length i. There are two cases.

First, consider the case when the $|u| < |str(z)|$. Consider $z_{|u|}, x_{|u|}, y_{|u|}$. Note that by check 4, $|str(z_{|u|})| = |str(z)| - |u| > 0$, and therefore, by check 3, $z_{|u|}$ is not the root. By check 1, $x_{|u|}, y_{|u|}$ are both descendant leaves of $z_{|u|}$. Further, $str(x_{|u|})$ and $str(y_{|u|})$ differ in the first character. It follows that for some edge (a, b) on the path from $z_{|u|}$ to either $x_{|u|}$ or $y_{|u|}$, b a child of a, $str(a)$ and $str(b)$ differ in the first character, which contradicts check 1.

Next, consider the case when $|u| > |str(z)|$. Let x', y' be the children of z which are ancestors of x, y, respectively. By check 2, the $(|str(z)| + 1)$th characters of $str(x')$ and $str(y')$ are different. Since $|u| > |str(z)|$, the $(|str(z)| + 1)$th characters of $str(x)$ and $str(y)$ are identical. Let a, b be the leftmost descendant leaves of x', y', respectively. By definition, $str(x'), str(y')$ are prefixes of $str(a), str(b)$, respectively; therefore the $(|str(z)|+1)$th characters of $str(a)$ and $str(b)$ are different. It follows that the $(|str(z)| + 1)$th characters of either $str(x), str(a)$ or $str(y), str(b)$ are different. Without loss of generality, assume that the $(|str(z)|+1)$th characters of $str(x), str(a)$ are different. Let c be the LCA of x, a. Then c is a descendant of x'. Further the longest common prefix of x, a has length at most $|str(z)|$ which is less than $|str(c)|$ by check 3. The case in the previous paragraph holds with x, a, c replacing x, y, z. Repeating the argument in the previous paragraph gives the necessary contradiction. ∎

E Renaming

We solve a related problem called DistinctSort in which we will output the sorted list of only the distinct items from the input array. Once this list has been computed, each element of A can find its rank in $O(\log \sigma)$ time with a single processor via binary search, thus completing the computation for renaming.

Given quadratically many processors DistinctSort can be performed as follows.

Lemma E.1 *Given an array A of size $\sqrt{n}$ with k distinct elements, we can sort the distinct elements of A in $O(\log k)$ time and $O(n \log k)$ work.*

Proof:

1. Each element $A[i]$ is assigned $\sqrt{n}$ processors, with which it checks in constant time if it is the leftmost occurrence of its value in the array. If not, $A[i]$ becomes inactive.

2. Each active element $A[i]$ find its nearest preceeding active neighbor in constant time via the rightmost-ones algorithm of [FRW84] using a total of n processors.

3. A list ranking is performed by pointer doubling on the active elements of A. They are thus placed by rank in a new array B. This takes $O(\log k)$ time and $\sqrt{n}$ processors.

4. The elements of B are sorted via Parallel Mergesort [Col86], in $O(\log k)$ time with k processors.

∎

The algorithm is then as follows. Below we use $|A|$ to denote the size of an array A.

DISTINCTSORT($C[1,m]$): Procedure to find the distinct elements of array C and sort them, using n processors.

1. If $m \leq \sqrt{n}$, exit from the recursion and apply Lemma E.1.
2. Set $\mu = \frac{\sqrt{n}}{m \log n}$. Independently select each element of C with probability μ. (Therefore, we expect to have selected $\sqrt{n}/\log n$ elements.)
3. Compact the selected items from C into an array D; See Lemma A.4.
4. Sort the unique elements of D into array E via Lemma E.1.
5. If $|E| \geq n^{1/4}$, exit from the recursion and sort C directly using Mergesort [Col86] and determine the output by directly computing the rank of the distinct elements via prefix sums.
6. Each element $C[i]$ searches for its value in E via binary search with a single processor. If its value is found $C[i]$ becomes inactive. Otherwise it remains active.
7. The active elements of C along with the elements in E are compacted into F. Return the list output by DISTINCTSORT(F).

To solve our problem, we need to invoke DISTINCTSORT($A[1,n]$).

E.1 Analysis

We first prove some general lemmas.

Lemma E.2 *Given an array A of size n, a sample S in which each element from A is chosen independently and randomly with probability p, with K being the set of distinct elements from S, let $k = |K|$ and let $A - K$ be the compacted array in which all elements from K are removed from array A. Then, $A - K$ is of size $O(k \log n/p)$ w.h.p.*

Proof: Let B be the sorted array of the elements in A. Notice that selecting elements independently at random in A is equivalent to doing so in B, so our argument will proceed by considering B. For our proof, we consider B. The items in S divide B into disjoint intervals. Consider an element $A[i]$. Suppose it lies in the interval between two elements of S in B and these two elements are both equal to, say e. Then it follows that $A[i] = e$ since B is sorted. In that case $A[i]$ cannot be in $A - K$. Therefore the size of $A - K$ is at most the total number of elements in B that lie in any interval between two elements of S that are distinct. Note that there are at most k such intervals. In what follows we will prove that w.h.p no interval has $\Omega(\log n/p)$ elements; that would prove $|A - K| = O(k \log n/p)$ w.h.p. We have

$$\begin{aligned} Pr(\text{Some interval has} \geq l \text{ elements}) &= \text{No. of intervals} \cdot Pr(\text{An interval has} \geq l \text{ elements}) \\ &\leq n(1-p)^l \\ &\leq ne^{-pl} \quad \text{since } (1-x) \leq e^{-x} \text{ for } x < 1 \\ &\leq 1/n^c \text{ for any constant } c \text{ setting } l = O(\log n/p) \end{aligned}$$

It follows that w.h.p. no interval has $\Omega(\log n/p)$ elements. That proves the lemma. ■

Lemma E.3 DISTINCTSORT *performs only* $O(1)$ *recursive rounds.*

Proof: We will simply unravel the first few rounds of recursion and determine the sizes of various arrays. Let the subscript i denote the ith round of recursion with any parameter in the algorithm. For the top level, $i = 1$; thus $m_1 = 1$ and $\mu_1 = \frac{1}{\sqrt{n}\log n}$. Since $|E_1| < n^{1/4}$ for round 2 to begin, we have $m_2 = O(\frac{n^{1/4}\log n}{\mu_1}) = O(n^{3/4}\log^2 n)$ by applying Lemma E.2. Then $\mu_2 = \Omega(\frac{1}{n^{1/4}\log^2 n})$ and $m_3 = O(n^{1/2}\log^3 n)$ if round 3 commences. Continuing similarly, $\mu_3 = \Omega(1/\log^3 n)$ and $m_4 = O(n^{1/4}\log^4 n)$ if round 4 commences. But $m_4 = O(n^{1/4}\log^4 n) \leq n^{1/2}$ for sufficiently large n. Thus the recursion cannot proceed beyond that round (that is, round 4) in Step 1. That proves the lemma. ■

Now we can complete the argument to conclude

Theorem E.4 DISTINCTSORT *takes* $O(\log\sigma)$ *time and* $O(n\log\sigma)$ *work on an arbitrary CRCW PRAM w.h.p.*

Finally as remarked earlier, the renaming problem can be solved in $O(\log\sigma)$ time and $O(n\log\sigma)$ work once DISTINCTSORT is performed. This proves our result.