Optimal Parallel Dictionary Matching and Compression

(Extended Abstract)

Martin Farach*
Rutgers University

S. Muthukrishnan†
DIMACS

April 26, 1995

Abstract

Emerging applications in multi-media and the Human Genome Project require storage and searching of large databases of strings – a task for which parallelism seems the only hope. In this paper, we consider the parallelism in some of the fundamental problems in compressing strings and in matching large dictionaries of patterns against texts. We present the *first work-optimal algorithms* for these well-studied problems including the classical dictionary matching problem, optimal compression with a static dictionary and the universal data compression with dynamic dictionary of Lempel and Ziv. All our algorithms are randomized and they are of the Las Vegas type. Furthermore, they are fast, working in time logarithmic in the input size. Additionally, our algorithms seem suitable for a distributed implementation.

1 Introduction

Large data bases of strings from multi-media applications and the Human Genome Project are now available on-line. The size of such information makes compression essential. Furthermore, one must be able to search through such data bases quickly. As the size of data bases increases and users demand quicker turnaround times, parallelism offers the only hope of meeting both challenges. In this paper, we consider parallelism in some of the fundamental problems in compressing strings, and in matching large dictionaries of patterns against texts. These two areas of study have an intimately linked history and are amongst the most intensively studied problems in Computer Science. (For compression see e.g. [25] and for dictionary matching see e.g. [3, 18, 22, 5, 4]). In this paper, we present the *first work-optimal algorithms* for these problems in a parallel setting. Furthermore, all of our algorithms are fast, working in time logarithmic in the input size.

Compression Schemes A wide variety of compression schemes exist in the literature [25]. Amongst the most powerful, and the ones that will be addressed in this paper, are the so-called *dictionary schemes*. These schemes can generally be described as follows. Suppose the prefix $S[1,i]$ of string S has been compressed. Then replace a prefix of $S[i+1,n]$ with a reference to some word from a "dictionary". If this word is of length k, then we will have compressed the prefix $S[1,i+k]$. Two issues arise: what is the dictionary, and how do we pick the best word from this dictionary so as to minimize the number of references in such a parsing.

The well-known LZ1 compression scheme of Lempel and Ziv [20] makes the following choices. The dictionary is all substrings $S[x,y]$ of S such that $x \leq i$. In this case, since the dictionary is always changing, that is, new strings are added to the dictionary as longer prefixes get compressed, this scheme is known as a *dynamic dictionary compression* scheme. As far as how a match is chosen from the dictionary, it is greedily taken to be the longest match. Since matches are substrings of strings, rather than say prefixes, this greedy heuristic is provably optimal. By *optimal* we mean that it gives the fewest dictionary references.

By contrast, *static compression schemes* are those in which the dictionary of words is fixed. Then the goal is to minimize the number of dictionary references needed to parse the string. A typical assumption, and one we

*farach@cs.rutgers.edu; Work done while this author was a Visitor Member of the Courant Institute of NYU.

†muthu@dimacs.rutgers.edu; Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

make in this paper as well, is that the dictionary consists of the input words and their prefixes. Such dictionaries are said to have the *prefix property*. The question of how a match is selected now becomes problematical. The greedy heuristic of always choosing the longest match need not give optimal compression. A variety of other sub-optimal heuristics (Longest Fragment First, etc.) have been proposed [25]. In this paper, we will only consider optimal parsing.

One of the crucial subtask for static dictionary compression is to find matches from a dictionary of patterns at some locations in the text. Of course this is simply the *dictionary matching* problem which is defined below. This problem has a rich independent history in the string matching literature.

The Dictionary Matching Problem. Given a set of patterns called the *dictionary* that can be preprocessed, the problem is to find, for each position in a text string given on-line, the longest pattern that occurs at that position. The natural parameters for dictionary $D = \{P_1, \ldots, P_k\}$ are $d = \sum_{i=1}^{k} |P_i|$ and $m = \max_i\{|P_i|\}$. The goal is to make the matching work independent of the dictionary size.

1.1 Our Results

Our results are obtained on the arbitrary CRCW PRAM. All our algorithms are randomized and they are of the Las Vegas type.

Dictionary Matching: First consider the case when the strings are drawn from a constant-sized alphabet. We present a randomized algorithm that preprocesses the dictionary in $O(\log d)$ time and $O(d)$ work. Following that, it matches a text string of length n against this dictionary in $O(\log d)$ time and $O(n)$ work. The preprocessing and matching work bounds are clearly optimal.

The previously known work bound for dictionary processing and text matching in this case are $O(d\sqrt{\log m})$ and $O(n\sqrt{\log m})$, respectively [22].

A natural model in string matching is the comparison model, commonly referred to as the case when the alphabet size is unbounded. We derive an algorithm for this case that has an additional $\log |\Sigma|$ multiplicative factor over the bounds above in both the time and the work; here, Σ is the alphabet set. On this model too, the work bounds of our algorithm are optimal.

For the special case when the alphabet size is polynomial in the input size, the classical algorithm of [3] for dictionary matching can be implemented with randomization in $O(n)$ sequential time and space. For this case, we obtain a suboptimal algorithm: our algorithm has an $O(\log\log d)$ extra factor in the dictionary and text processing work (with no penalties on time). An important task in our algorithm is the construction of a suffix tree – this has the well-known bottleneck of parallel integer sorting for which the best known algorithm is suboptimal by an $O(\log\log d)$ factor in this case [8]. The work bound of our algorithm improves on the previously best algorithm [22] unless the alphabet size is superexponential in m.

In what follows, we assume that the alphabet size is constant, as is standard in the literature on compression techniques [25]. However, the bounds we quote below can be easily modified to include other cases in a manner analogous to the dictionary matching case above.

Optimal Static Compression: We give a work optimal algorithm for static dictionary compression. Our algorithm pre-processes the dictionary of size d in $O(\log d)$ time and $O(d)$ work. Given a string of length n, our algorithm generates its *optimally* compressed string in $O(\log d + \log n)$ time and $O(n)$ work[1].

The previously known best algorithm for this problem takes time $O(\log^2 n)$ and $O(n^3 \log^2 n)$ work, or alternately, takes time $O(\log n)$ and $O(n^4 \log n)$ work [2] after $\Omega(d \log d)$ work for preprocessing.

Dynamic Dictionary Compression: We present the first known optimal algorithm for LZ1 compression as well for its uncompression. Our algorithm constructs the LZ1 compression of a given string of length n in $O(\log n)$ time and $O(n)$ work. Also, given the LZ1 compressed version of a string of length n, our algorithm reconstructs the string in $O(\log n)$ time and $O(n)$ work. Here we make the standard assumption [23] that n is known.

The previously known best algorithms perform $O(n \log n)$ work for compression [23, 10] as well as for uncompression [23].

1.2 Technical Contribution

The suffix tree of a string (defined in Section 2) is a versatile data structure in string processing. All our algorithms crucially rely on a recently discovered algorithm for constructing the suffix tree of a string [11]. While this work-optimal algorithm paved way for our work optimal algorithms for dictionary matching and compression, we stress that the suffix tree construction was not the inherent bottleneck in the previously best

[1] Recall that the optimally compressed string is one with the fewest possible dictionary references.

results we have quoted for these problems. For instance, the dictionary matching algorithm in [22] does not rely on suffix trees at all. The bottleneck in the previosuly known best bounds for static optimal compression [2] lies in computing shortest paths in graphs.

Dictionary Matching. Within two years of the discovery of the classical linear time string matching algorithm due to Knuth, Morris and Pratt [19], Aho and Corasick [3] designed a linear time (hence, optimal) algorithm for dictionary matching by generalizing the finite automaton construction in [19] to a set of strings. In the mid-eighties, Galil [12] and Vishkin [27] designed the first work-optimal string matching algorithms, which have since been extended significantly [28, 13, 9]. However, a work-optimal algorithm for dictionary matching has remained elusive. As in the case of [19], the finite automaton based approach of [3] is inherently sequential. Recent progress on parallel dictionary matching [4, 5, 18, 22] based on alternate techniques has only yielded suboptimal bounds.

Our work-optimal results for dictionary matching are obtained by judiciously combining ideas from [5] and [22] to solve a generalization of dictionary matching, called, *dictionary substring matching* (see Section 3). We crucially employ a novel data structural primitive that we call the *nearest colored ancestors* problem on trees. The sequential version of our solution for this problem has already been applied successfully in compilers for Object Oriented Programming Languages [21]. Also, interestingly, we have devised a very fast procedure that checks the output of the basic Monte Carlo dictionary matching algorithm: therefore, our dictionary matching algorithm is of the Las Vegas type.

Static Dictionary Compression. We obtain our work-optimal algorithm for this problem by discovering and exploiting the structural properties of optimal parsings. In particular, we show that it suffices to consider certain *dominating* dictionary references to determine the optimal parsing (See Section 4). In contrast, previous approaches to this problem have relied on applying a general purpose shortest-paths routine [2]; this step is work-inefficient due to the well-known transitive closure bottleneck [16].

Dynamic Dictionary Compression. LZ1 [20] and LZ2 [30] are two well-known dynamic compression schemes. LZ1 is known to give better compressions in practice; for example, see Unix `compress` and `gnuzip`. Nonetheless, LZ2 is implemented in practice because of the simplicity of its sequential implementation. Curiously, while we provide optimal work RNC algorithm for LZ1 compression, LZ2 is P-Complete [1] (hence unlikely to have (R)NC algorithms).

Versions of our algorithms seem suitable for distributed implementation on a network of workstations [24]. In fact, in this setting, we can conclude from Communication Complexity that even checking equality of strings requires randomization for efficiency [29]. Thus the randomization in our algorithms seems well justified.

Section 2 contains a review of results we use. Our algorithms for dictionary matching, dynamic compression and static compression are described in Sections 3, 4, and 5 respectively.

2 Models, Tools and Definitions

For a string s, its suffix tree T is a compacted trie of its suffixes. Each edge in T is labeled with a substring of s and the substrings marking the edges out of a node all have distinct first characters. The child of u that has the edge marked by a substring with leading symbol a is called the *a-child* of u; any node in the subtree of the a-child of u is called an *a-descendant* of u. We denote the concatenation of the labels on the path from the root to a node u as $\sigma(u)$. T has $|s|$ leaves which are in one-to-one correspondence with the suffixes of s. For a leaf u, $\sigma(u)$ is the suffix that corresponds to u. For each node u for which $\sigma(u) = a\alpha$ for some alphabet a and string α, a *suffix link* is a pointer from node u to node v in the suffix tree where $\sigma(v) = \alpha$. we call this as the *a-link*. ¿From the properties of suffix trees, it follows that the suffix links are well-defined. In referring to strings, we write $s_1 - s_2$ to denote the suffix of s_1 after removing the prefix s_2.

All our algorithms are derived on the arbitrary CRCW PRAM. All the expected time and work bounds of our algorithms hold with high probability, that is, the failure probability is polynomially small is the input size. We use the following results in parallel algorithms.

Lemma 2.1 ([11]) *Given a string of length n, its suffix tree can be constructed in $O(\log n)$ expected time and $O(n)$ expected work if the string has constant-sized alphabet, and in $O(\log n)$ expected time and $O(n \log\log n)$ expected work if it has a polynomial-sized alphabet.*

Lemma 2.2 ([14]) *Given a graph on n vertices and m edges, its connected components can be determined in $O(\log n)$ time and $O(m)$ work.*

Lemma 2.3 ([7]) *Given an array A of n numbers, it can be pre-processed in $O(\log n)$ time and $O(n)$ work so any range-maxima query (that is, given $[i, j]$, return the maximum value in $A[i], A[i+1], \ldots, A[j]$) can be processed in $O(1)$ time and work.*

Lemma 2.4 ([6]) *Given an array, A, of n numbers, we can compute, for each location i, the nearest position j such that $j < i$ and $A[j] < A[i]$ in $O(\log\log n)$ time with $O(n)$ work.*

Lemma 2.5 ([26]) *A subset of numbers from the universe $1 \ldots N$ can be maintained under insert, delete, extract maximum or minimum and find predecessor or successor queries in $O(\log\log N)$ time using $O(s)$ space where s is the size of the subset.*

Lemma 2.6 ([7]) *Given a suffix tree of a string T, it can be pre-processed in $O(\log n)$ time and $O(n)$ work for string equality queries (that is, is $T[i \cdots (i+l)] = T[j \cdots j+l]$ for some i, j, l), and more generally, longest common prefix queries (that is, what is the longest common prefix between $T[u \cdots u+l_1]$ and $T[x \cdots x+l_2]$ for some nonnegative integers u, x, l_1, l_2) can be answered in $O(1)$ time and work per query.*

Lemma 2.7 ([11]) *Given a tree of n nodes with some marked nodes, the nearest marked ancestor of each node in the tree can be computed in $O(\log n)$ time and $O(n)$ work.*

3 Dictionary Matching

3.1 Algorithmic Outline

In this section, we provide optimal parallel algorithms for dictionary matching. Formally, the dictionary matching problem is as follows. A dictionary $D = \{P_1, \ldots, P_k\}$ such that $\sum_{i=1}^{k} |P_i| = d$ is given for preprocessing. Queries of the form $T[1, n]$ are presented and the output is $M[1, n]$ such that $M[i] = j$ if $T[i, i + |P_j| - 1] = P_j$, that is, if P_j matches at $T[i]$, and no longer pattern from D also matches at $T[i]$[2] In what follows, let $\hat{D}$ be the concatenation of the patterns in the dictionary. We solve this problem in two steps.

Step 1. We first solve the *dictionary substring problem.* That is, for each text position $T[i]$, we determine $S[i]$, which is the longest substring in $\hat{D}$ (not necessarily a pattern in the dictionary) that occurs there. This substring $S[i]$ is specified as a pointer into the suffix tree of $\hat{D}$, that is, as a pointer to the edge (u, v) and a length l, such that $S[i]$ is a prefix of $\sigma(v)$ of length $|\sigma(u)| + l$.

Step 2. For each text position $T[i]$, we determine $M[i]$ from $S[i]$.

[2] We can take $M[i]$ to be some length 0 "pattern" if no P_j matches at $T[i]$.

Outline of Step 1. We perform this in two substeps. Fix $\mathcal{L}$, an integer to be determined later.

Step 1A. We determine $S[i]$, for each location $i = k\mathcal{L}$, for integer $k \in [1, n/\mathcal{L}]$. This substring is specified as an edge (u, v) and a length l. This is done as in [5]. We briefly sketch the approach there. We first construct a separator decomposition of the suffix tree of $\hat{D}$. Then we trace down from the root starting from each of the desired text locations independently. The key is that string comparison along the edges and separators are done using fingerprints [17].

Step 1B. We now compute $S[i]$ for all positions which are not a multiple of $\mathcal{L}$. We process each window $T[k\mathcal{L} - (\mathcal{L} - 1), \ldots, k\mathcal{L}]$ independently, for integer $k \in [1, n/\mathcal{L}]$. Given $S[i]$, we show how to compute $S[i-1]$ via a procedure called ExtendLeft procedure. Starting with $S[k\mathcal{L}]$ and applying ExtendLeft repeatedly, we can compute $S[i]$ for all i within the window under consideration. To implement ExtendLeft we make the following observation.

Observation 1 *$S[i-1]$ is the longest prefix of $T[i-1] \circ S[i]$ that is a substring in $\hat{D}$, where $\circ$ denotes string concatenation.*

We also use the following structural property of the suffix links. Let $p(w)$ denote the nodes in order on the path from the root to node w. Let $S[i]$ fall on edge (u, v) with length l.

Observation 2 *The possible substrings $S[i-1]$ form a path in $T_{\hat{D}}$. Furthermore, each node on this path has a $T[i-1]$-link to the nodes in $p(u)$.*

We do ExtendLeft in two substeps.

Step 1B.1. We determine the deepest node in $T_{\hat{D}}$, say w, that has a $T[i-1]$-link to an ancestor, say u_a, of u. Furthermore, let r be the node in $T_{\hat{D}}$, if any, that is the q-child of w if u is the q-descendant of u_a. Then $\sigma(w)$ is a prefix of $S[i-1]$. In order to implement Step 1B.1, that is, determine w, r and u_a, we abstract a problem on trees that we call the *nearest colored ancestors* problem. In Section 3.2, we state the problem and present our algorithm.

Step 1B.2. In Step 1B.1, we have determined a prefix of $S[i-1]$, namely, $\sigma(w)$. We now determine the suffix of $S[i-1]$, namely, $S[i-1] - \sigma(w)$. We claim without proof that this suffix is the longest prefix of $\sigma(u) - \sigma(u_a)$ and $\sigma(r) - \sigma(w)$. This can determined by Lemma 2.6.

Details of Step 2. We perform Step 2 in two substeps. Both the substeps require precomputation on the dictionary, as described below. In what follows, consider only a single text location $T[i]$; the other text locations are considered similarly.

Step 2A. In this step, we determine $B[i]$, that is, the longest prefix of $S[i]$ that is a prefix in D. For this, we perform the following precomputation. We mark each node in $T_{\hat{D}}$ that is a prefix in the dictionary; this is performed by a table look-up using the fingerprints. Then, we determine, for each node v, its nearest marked ancestor, $A[v]$. Furthermore, for each leaf l_j, we determine $L[j]$, its *legal length*, defined as follows. $L[j] = l$ if l is the minimum number such that a new pattern starts at $j + l - 1$ in $\hat{D}$. For each node v, we precompute $L[v]$ to be the maximum legal length of the leaves in the subtree rooted at that node. This is performed by Lemma 2.3.

Given L and A, Step 2A proceeds as follows. Recall that the substring $S[i]$ is provided as an edge (u, v) and a length l. We determine the longest prefix of $\sigma(v) - \sigma(u)$ that leads to a prefix in the dictionary while tracing down starting from $A[u]$ along an edge. We now claim, without proof, that this is of length exactly $L[A[u]]$. That completes the description of Step 2A.

Step 2B. We will now compute $M[i]$. For this, we precompute, for each pattern prefix, its longest prefix which is a pattern. This is done by modifying a similar step in [22]. Following that, we can look up $M[i]$ directly.

3.2 Data Structures

The Nearest Colored Ancestor Problem. Consider a rooted tree with n nodes, each node being marked with several colors such that the number of distinct colors used over the entire tree is $\mathcal{C}$ and the total number of colors used is C. The problem is to preprocess this information so as to answer $\textsc{Find}(p, c)$ queries quickly. The $\textsc{Find}(p, c)$ operation returns the nearest ancestor of the node pointed to by p (possibly the node itself) which is marked with color $c \in \mathcal{C}$.

The Algorithm Let T be the initial tree. Let $\hat{T}$ be the *augmented tree* derived from T by adding two children to each leaf of T, and an extra leaf to each internal node with only a single child. The leaves of $\hat{T}$ are called the *out-leaves*. Note that the number of out-leaves is $O(n)$. The cth *naive skeleton tree*, denoted T_c, is the tree obtained by restricting $\hat{T}$ to only those nodes which are colored c; in addition, all out-leaves are retained in each T_c and the root of $\hat{T}$ is retained regardless of whether it is colored c. The cth *real skeleton tree*, denoted R_c, is obtained by deleting the out-leaves from T_c. In what follows, we describe the algorithm as if the tree T and the query are presented at the same time. It is easy to modify it to the case when the tree is available for preprocessing and the query is presented later. First we describe the algorithm using naive skeleton trees. Later we will show how to use the real skeleton trees instead to improve the efficiency.

Step 1. Generate $\hat{T}$. For each internal node v in $\hat{T}$, determine two out-leaves l_v and r_v whose least common ancestor (LCA) is v; clearly two such leaves exist, by construction.

Step 2. Generate the naive skeleton tree T_c for each $c \in \mathcal{C}$. Process each T_c so LCA queries can be answered efficiently.

Step 3. To implement $\textsc{Find}(p, c)$, determine the LCA of l_v and r_v in T_c. This is the answer to the query.

That completes the algorithm. The preprocessing takes time $O(\log d)$ and work

$$O(\textstyle\sum_{\text{all naive skeleton trees } T_c} |T_c|) = O(n|\mathcal{C}|).$$

Each query takes time $O(1)$. Both these bounds follow from Lemma 2.6.

In what follows, we show how the preprocessing work can be reduced to

$$O(\textstyle\sum_{\text{all real skeleton trees } R_c} |R_c|) = O(n + C)$$

while taking $O(\log \log n)$ time for each query. To achieve this, we use the van Emde Boas' result [26].

For any $c \in \mathcal{C}$, associate with each node in the naive skeleton tree T_c, the group of out-leaves which are its descendant but which are not the descendant of any other internal node in T_c. The key observation is that *these groups form a partition of the out-leaves of $\hat{T}$ into sets of disjoint ranges* in their Euler tour numbering. Moreover, the sets of disjoint ranges associated with an internal node u can be assigned to distinct instances of u in the Euler tour unambiguously (as described below). As a result, we can look upon the out-leaves as being just partitioned into disjoint ranges associated with distinct instances of nodes. Using this observation, the algorithm can be changed as follows.

Step 1. Generate $\hat{T}$ and assign Euler tour numbering to the vertices. For each node v in T, determine two out-leaves l_v and r_v whose least common ancestor (LCA) is v; clearly two such leaves exist, by construction.

Step 2. For each c, generate the real skeleton trees R_c and also generate its nodes in the Euler tour order with the same numbering as that in $\hat{T}$. Note that a node which has two descendant nodes marked c can appear several times in the Euler tour on $\hat{T}$ between the last and the first occurrence of the left and the right descendant respectively; in the Euler ordering for R_c, picking its number for any one such occurrence (say the leftmost) will do. For each node v in the real skeleton tree R_c, define G_v to be the set of out-leaves that are direct children of node v in the corresponding naive skeleton tree T_c.

Step 3. We implement FIND(p, c) as follows. Say p points to v and l_v and r_v are as defined above. By determining the smallest number greater than l_v and the largest number smaller than l_v in the Euler ordering for R_c, we can easily determine the node u such that G_u contains l_v. We do a similar computation for r_v. Now, we determine the LCA of the node(s) which contain l_v and r_v. This is the answer to the `Find` query.

That completes the algorithm. All operations can be done in $O(n+C)$ work and $O(\log n)$ time. For each R_c, predecessor and successor queries can be determined using a Van Emde-Boas structure where $N = n$ and $s \leq n$. Each query takes $O(\log\log n)$ time.

We remark that the entire algorithm has two main ideas. One is to show how finding the nearest colored ancestor can be implemented using LCA queries by adding extra leaves. The second is the structural observation that the extra leaves can be considered in groups of ranges, which reduces our problem to the Van Emde-Boas structure. This algorithm has already been used for dynamic method look-up in object oriented programming languages like the smalltalk [21].

3.3 Results

In this section, we state the complexity bounds for each step in our dictionary matching algorithm. All time bounds stated below are expected time bounds that hold with high probability. Also, the bound below do not include that for constructing the suffix tree for T_D from Lemma 2.1 [11].

Step 1. From the bounds in [5] it follows that Step 1A can be done in $O(\log d)$ time and $O(n \log d/\mathcal{L})$ work. The precomputation required can be done in $O(\log d)$ time and $O(d)$ work. Using the complexity bounds in Section 3.2, Step 1B can be done in $O(\mathcal{L}\log\log d)$ time and $O(n\log\log d)$ work if the string has polynomial-sized alphabet, and $O(\mathcal{L})$ time and $O(n)$ work if it has binary alphabet; the required precomputation for this can be done in $O(\log d)$ time and $O(d)$ work.

Step 2. Step 2A can be done in $O(1)$ time and $O(1)$ work per text location. The preprocessing can be done in $O(\log d)$ time and $O(d)$ work. ¿From the results in [22], the preprocessing for Step 2B can be done in $O(\log d)$ time and $O(d)$ work. The text processing in Step 2B can be done in $O(\log^* d)$ time and $O(d)$ work.

To sum up, the text processing takes time $O(\log d + \mathcal{L})$ and work $O(n + n\log d/\mathcal{L})$ if D has binary sized alphabet and it takes time $O(\log d + \mathcal{L}\log\log d)$ and work $O(n + n\log d/\mathcal{L})$ if D has a polynomial sized alphabet. Therefore,

Theorem 3.1 *The dictionary matching problem over constant-sized alphabet can be solved in $O(\log d)$ time and $O(n)$ work following $O(\log d)$ time and $O(d)$ work preprocessing. The work bounds are optimal.*

Proof: By encoding each symbol in the alphabet in binary, any string over a constant-sized alphabet can be replaced by another of same asymptotic size over a binary-sized alphabet. Then, we set $\mathcal{L} = \log d$ to get the theorem. ∎

Setting $\mathcal{L} = \log d/\log\log d$ we get,

Theorem 3.2 *The dictionary matching problem for strings over a polynomial sized alphabet can be solved in $O(\log d)$ time and $O(n\log\log d)$ work following $O(\log d)$ time and $O(d\log\log d)$ work preprocessing.*

Theorem 3.3 *The dictionary matching problem for strings over an unbounded alphabet (that is, in the comparison model) can be solved in $O(\log d\log|\Sigma|)$ time and $O(n\log|\Sigma|)$ work following $O(\log d\log|\Sigma|)$ time and $O(d\log|\Sigma|)$ work preprocessing. Here Σ is the size of the alphabet set. These work bounds are optimal.*

Proof: First we use the randomized renaming procedure in [11] that remaps the symbols into the range $1\cdots|\Sigma|$ in $O(\log n\log|\Sigma|)$ time and $O(n\log|\Sigma|)$ work in the comparison model. Following that, we can replace the n length text by a string of length $O(n\log|\Sigma|)$ and apply Theorem 3.1. ∎

3.4 Checking the Pattern Matches

We are given, for each $i \in [1, n]$, a pointer $M[i]$ to suffix tree T_D of dictionary D which is claimed to represent the longest pattern prefix from D that matches at $T[i]$.

The problem is to check the correctness of the claimed matches.

First, we can check in constant time if any of the matches can be extended by a character, by reference to T_D. For technical reasons, we make the following small change. If at some location i, $M[i]$ is undefined, that is, there is no match at i, then we treat $M[i]$ to be a special pointer to the singleton character $T[i]$. Now, we can check in constant time if the first character of $M[i]$ matches text position $T[i]$. If neither test fails, the we proceed as follows. Let $L[i]$ be the match length at i, that is, the string length of $M[i]$. We say that i *dominates* j if $i < j$ and $i + L[i] \geq j + L[j]$. If a position j has some dominating position i, then we say that j is *dominated*, and if no such i exists, we say that j is *dominating*. We can find, for each position j, a dominating edge, by techniques described in Lemma 5.2. Now, for each j, let $A[j]$ be a dominating match of j, if such a match exists. If so, we can check if the match at j is consistent with the overlapping substring given by $M[A[i]]$ over the same positions. For example, suppose $M[i] = P_{k_1}[1, L[i]], M[j] = P_{k_2}[1, L[j]]$ and $i = A[j]$. Then we check to make sure that $P_{k_2}[1, L[j]] = P_{k_1}[j - i, j - i + L[i]]$. Such a check takes constant time, by Lemma 2.6.

Now we consider only dominating matches, since, by the previous step, if the dominating matches are correct, so are the dominated matches. Furthermore, we need only check that the dominating matches are all pairwise consistent, since we have already checked to make sure that each text position is consistent with some pattern. So if the patterns are all pairwise consistent, then they are all correct. For each dominating match i, we find the next dominating match $N[i]$ via all nearest ones (See Lemma 2.4). For each dominating i, we do the following. Let $j = N[i], M[i] = P_{k_1}[1, L[i]]$ and $M[j] = P_{k_2}[1, L[j]]$. Then we check to make sure that $P_{k_2}[1, L[i] - j + i - 1] = P_{k_1}[j - i, L[i]]$. Once again, such a check takes constant time, by Lemma 2.6.

Lemma 3.4 *If an array M passes the above tests, then it is correct.*

Proof: We need to show only that for dominating matches $u < v < w$, if u is consistent with v and v is consistent with w, then v is consistent with w. Let $M[u] = P_{k_u}[1, L[u]], M[v] = P_{k_v}[1, L[v]], M[w] = P_{k_w}[1, L[w]]$. Then $P_{k_w}[1, L[u] - w + u - 1] = P_{k_v}[w - v, L[u] - v + u - 1]P_{k_u}[w - u, L[u]]$. ∎

4 Dynamic Compression

4.1 Compression

We wish to compute the LZ1 parsing of a string $S[1, n]$. Recall that the LZ1 parsing is defined as follows: If we have parsed $S[1, i]$, then the next phrase to be parsed is the longest substring of S which starts at both $S[i + 1]$, and at some $S[j]$, for $j < i + 1$, that is, we must find the pair (j, k) such that $S[i + 1, i + k + 1] = S[j, j + k]$, $j < i + 1$, and k maximized. Given such a (j, k), we can replace $S[i + 1, s + k + 1]$ with (j, k), and we will have parsed up until $S[i + k + 1]$. If we ever encounter a new character α, e.g. when we parse the first character in the string, then we output $(\alpha, 0)$ and advance one character[3].

Suppose that for each $i \in [1, n]$, we have a pair $M[i] = (j, k)$ such that $j < i$, $S[i, i + k] = S[j, j + k]$ and k is maximized. Notice that this is a superset of the information needed for the LZ1 parse, since we need the $M[i]$ for only a (hopefully small) subset of the positions of a string. We can easily find those i which participate in the LZ1 parse in $O(\log n)$ and $O(n)$ work as follows. Let $G = ([1, n], E)$ be a directed graph such that $(i, i + k) \in E$ if $M[i] = (j, k)$ and $k > 1$. For $M[i] = (\alpha, 0)$, $(i, i + 1) \in E$. Notice that such a graph is a tree, since it is acyclic and each node i has a parent $i + k$. Also, the node 1 is a leaf, and n is the root. Then the positions that participate in an LZ1 parsing of S are exactly those on the path from 1 to n in G. These nodes can be determined in $O(\log n)$ time and $O(n)$ work by many methods, e.g. tree contraction, level ancestors, Euler tour techniques, etc.

So we must show how to compute the $M[i]$ within the desired bounds. As in any string matching problem, we begin by building a suffix tree T_S. Let l_i be the leaf representing the suffix $S[i, n]$. For each interval node v, let $L[v] = \min\{i | l_i \text{ is a descendant of } v\}$. Notice that if $L[v] = i$, then some child w of v will have $L[w] = i$. Conversely, for each l_i, there will be a (possibly empty) chain of ancestors v such that $L[v] = i$. For each i, let $A[i] = v$ if v is the deepest ancestor of l_i such that $L[v] \neq i$. Notice that $L[A[i]] < i$. The importance of these functions is summarized as follows:

Lemma 4.1 $\forall i \in [1, n], M[i] = (L[A[i]], |A[i]|)$.

Proof: First, by the observations above, $L[A[i]] < i$, and so the $L[A[i]]$th suffix is to the left of the ith suffix. Now suppose that there is a (j, k) such that $j < i$, $k > |A[i]|$ and $S[i, i + k] = S[j, j + k]$. Let v be the least common ancestor of l_i and l_j.

[3] There are several variants on how new characters are handled, but they are easily convertible, and the algorithms in this section serve to compress and uncompress according to any of the standard LZ1 variants.

For L, mark a node v if $A[v] > A[p(v)]$, where $p(v)$ is the parent of v. Then, for each leaf $l(i)$, $L[i]$ is the parent of the nearest marked ancestors of l_i. By Lemma 2.7, we can therefore compute L within the desired time bounds. ■

Theorem 4.2 *A string of length n can be compressed via LZ1 in $O(\log n)$ expected time and with $O(n)$ expected work.*

4.2 Uncompression

Given $C[1,m]$, and LZ1 parsing of a string of length n, we we will construct the uncompressed string $T[1,n]$. First, we do a prefix sum on the lengths of the blocks in C -taking the special blocks of the form $(\alpha, 0)$ to have length 1. We place a 1 at each i in an array $B[i]$ if i is the beginning of a block in T. $B[i] = 0$ otherwise. Now, for each $i \in [1,n]$ we compute which block it is in, that is, if $B[i] = 0$, we set $B[i] = j$ such that $j < i$, $B[j] = 1$ and j is maximized. This is done by Lemma 2.4. And for each $B[i] = 1$, we set $P[i] = j$ if the block starting at $T[i]$ is represented by $C[j]$. Now we set up the directed graph $G = ([1,n], E)$ as follows: $(i,j) \in E$ if $(x,y) = P[B[i]]$ and $j = x + i - B[i]$. We get a forest in which the root of each tree is a node of the form $(\alpha, 0)$ and all nodes in the tree are αs in the text. Thus, to uncompress, we need only find out which tree each node is in, which we do by Lemma 2.2.

Theorem 4.3 *Given the LZ1 compressed representation of an n character string, the uncompressed representation can be generated in $O(\log n)$ time and $O(n)$ work.*

5 Static Compression

For each position, i, let $M[i]$ be the length of the longest pattern prefix from D that occurs at $T[i]$, that is, if $M[i] = j$, then there is some k such that $P_k \in D$ and $T[i, i+j] = P_k[1,j]$. Similarly, $M[i]$ is the length of the longest matching pattern in a dictionary with the prefix property. Let $G = ([1,n], E)$ be a directed graph such that $(i, i+k) \in E$ if $k \leq M[i] + 1$. Now those positions, i, that participate in an optimal parse of T are those i along a shortest path from 1 to n in G. However, it would be grossly inefficient to apply a general shortest paths routine to this problem. Instead, we take advantage of the structure of G, as follows.

Let edge (i,j) *dominate* edge (k,l) if $i < k$ and $j \geq l$. Let $G^d = ([1,n], E^d)$ be a directed graph such that $(i, i+k) \in E^d$ if $(i, i+k) \in E$ and there is no $e \in E$ which dominates $(i, i+k)$. Then:

Lemma 5.1 *The shortest path from 1 to n in G^d is the same length as the shortest path from 1 to n in G.*

Proof: Let $S_G[i]$ be the length of the shortest path in G from 1 to i.

Claim 5.1A $1 \leq i < j \leq n \Rightarrow S_G[i] \leq S_G[j]$

Proof: Let $p_1, \ldots, p_{S_G[j]}$ be a shortest path from 1 to j in G. Let k be such that $p_k < i \leq p_{k+1}$. Then $p_1, \ldots, p_k, i$ is path from 1 to i of length $k+1 \leq S_G[j]$. Hence, $S_G[i] \leq k+1 \leq S_G[j]$. ■

Claim 5.1B *There is a path of length $S_G[i]$ from 1 to i in G which uses only dominating edges.*

Proof: By induction: For $i = 2$, $(1,2)$ is a dominating edge.

Assume the claim holds for all $i' < i$. Suppose it fails for i and for some path, $P = p_1, \ldots, p_{S_G[i]}$, $p_1 = 1, p_{S_G[i]} = i$, let (p_j, p_{j+1}) be the last dominated edge used. If $p_{j+1} < i$, the we can replace $p_1, \ldots, p_{j+1}$ with a path of length $S_G[p_{j+1}]$ using only dominating edges, and hence get a shortest path to i using only dominating edges.

Now consider the case where $p_{j+1} = i$. Let (l,i) be the dominating edge ending at i –the existence of such an edge follows directly from the construction of Lemma 5.2. Now Claim 5.1A implies that $S_G[l] \leq S_G[p_j]$ and so the dominating path to l followed by edge (l,i) is a shortest path to i, hence establishing our claim. ■

This also finishes the proof of the lemma. ■

The questions is how to find the shortest 1 to n path in G^d, and how to construct G^d directly. Notice, however, that each node in G^d has a single incoming edge, so G^d is a rooted tree in which the edges are directed away from the root. In this case, the 1 to n path is unique and can be computed in $O(\log n)$ time and $O(n)$ work. As for the construction:

Lemma 5.2 *G^d can be constructed in $O(\log\log n)$ time and $O(n)$ work from array M.*

Proof: Let $M'[i]$ be the prefix maximum (see Lemma 2.3) of array $M[i]$. For each i, let $L[i] = \min\{j | M'[j] > i\}$. Then, $(x,y) \in E^d$ if $L[y] = x$. So we must find the rank of each $i \in [1,n]$ in M'. If $r(i)$ is the rank of i, then $L[i] = r(i) + 1$. The rank of each i can be computed in constant time and linear work. All steps are therefore executable within the desired time bounds, thus finishing the proof. ■

We conclude with

Theorem 5.3 *Given a dictionary, D, of size d, and a string, T, of size n, then Static Dictionary with Prefix Property Optimal Parse of S with respect to D can be computed in $O(\log d + \log n)$ time and $O(n)$ work, after $O(\log d)$ time, $O(d)$ work preprocessing of D.*

References

[1] S. De Agostino. P-complete problems in data compression. *Theoretical Computer Science*, 127:181–186, 1984.

[2] S. De Agostino and J. Storer. Parallel algorithms for optimal compression using dictionaries with the prefix property. *Proc. of the 2nd IEEE Data Compression Conference*, pages 52–61, 1992.

[3] A.V. Aho and M.J. Corasick. Efficient string matching. *Communications of the ACM*, 18(6):333–340, 1975.

[4] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 760–766, 1991.

[5] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. *Proc. of 3rd Combinatorial Pattern Matching Conference*, pages 259–272, 1992. Tucson, Arizona.

[6] 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.

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

[8] 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.

[9] R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, K. Park, S. Muthukrishnan, H. Ramesh, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. of the 34th IEEE Annual Symp. on Foundation of Computer Science*, pages 248–258, 1993.

[10] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Information Processing Letters*, 38:57–60, 1991.

[11] M. Farach and S. Muthukrishnan. An optimal, logarithmic time, randomized parallel suffix tree contruction algorithm. Technical report, DIMACS, 1995.

[12] Z. Galil. Optimal parallel algorithms for string matching. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 240–248, 1984.

[13] Z. Galil. A constant-time optimal parallel string-matching algorithm. *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, pages 69–76, May 1992.

[14] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 492–501, 1986.

[15] M. E. Gonzalez-Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, pages 344–373, 1985.

[16] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. North-Holland, Amsterdam, 1990.

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

[18] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 6:388–398, 1989.

[19] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[20] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22:75–81, 1976.

[21] S. Muthukrishnan. A time and space efficient algorithm for dynamic method look-up in object oriented programming languages. *manuscript*, 1995.

[22] S. Muthukrishnan and K. Palem. Highly efficient parallel dictionary matching. *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[23] M. Naor. String matching with preprocessing of text and pattern. *Proc. of 18th International Colloquium on Automata Languages and Programming*, pages 739–750, 1991.

[24] M. Papadipoyli. A distributed dictionary matching implementation. 1994.

[25] J. Storer. *Data compression: methods and theory*. Computer Science Press, Rockville, Maryland, 1988.

[26] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[27] U. Vishkin. Optimal parallel pattern matching in strings. *Proc. of 12th International Colloquium on Automata Languages and Programming, Springer LNCS*, pages 91–113, 1985.

[28] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM Journal on Computing*, 20:303–314, 1991.

[29] A. C. C. Yao. Some complexity questions related to distributed computing. *Proc. of the 11th Ann. ACM Symp. on Theory of Computing*, pages 209–213, 1979.

[30] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.