Adaptive Dictionary Matching

Amihood Amir* Martin Farach†
Georgia Tech University of Maryland

Abstract

Most traditional pattern matching algorithms solve the problem of finding all occurrences of a given pattern string P in a given text T. Another important paradigm is the *dictionary matching* problem defined as follows. Let $D = \{P_1, ..., P_k\}$ be the *dictionary*. We are seeking all locations of dictionary patterns that appear in a given text T.

Aho and Corasick gave a *semi-adaptive* algorithm. The dictionary is preprocessed in time $O(|D| \log |\Sigma|)$, where Σ is the alphabet. Subsequently any given text T is analyzed in time $O(|T| \log |\Sigma|)$. However, inserting a new pattern into the dictionary or deleting a dictionary pattern results in a need for reprocessing the entire dictionary.

We present new semi-adaptive and fully-adaptive dictionary matching algorithms. In the fully adaptive algorithm, the dictionary is processed in time $O(|D| \log |D|)$. Inserting a new pattern P_{k+1} into the dictionary can be done in time $O(|P_{k+1}| \log |D|)$. A dictionary pattern can be deleted in time $O(\log |D|)$. Text scanning is accomplished in time $O(|T| \log |D|)$. We also present a parallel version of the algorithm with optimal speedup for the dictionary construction and pattern addition phase and a logarithmic overhead in the text scan phase.

Our method incorporates a new way of using suffix trees as well as a new data structure in which the suffix tree is embedded for the sequential algorithm. In the parallel algorithm, we introduce a data structure which supports rapid binary-like search through the substrings of the text.

*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 894-3152; amir@cc.gatech.edu; Partially supported by NSF grant IRI-9013055.

†Department of Computer Science, University of Maryland, College Park, MD 20742, mpf@cs.umd.edu

1 Introduction

Traditional Pattern Matching has dealt with the problem of finding all occurrences of a single pattern in a text (under some definition of the word "occurrence"). The most basic instance of this problem is the *Exact String Matching Problem*, i.e. the problem of finding all exact occurrences of a pattern in a text. This problem has been extensively studied. The earliest linear time algorithms include [KMP77] and [BM77]. For more recent work see e.g. [GS83, Vis89]. A somewhat more general problem arises when the notion of approximate matching is introduced. In this case, a pattern matches a piece of text if their distance under a given metric is below some threshold. The choice of metric strongly influences the resultant algorithm (see e.g. [AF91, ALV90, GG88, Ukk85]). Note that exact string matching is a subcase of approximate matching in which the distance metric is the Hamming distance and the threshold is 1.

While the case of a pattern/text pair is of fundamental importance, the single pattern model is not always appropriate. One would often like to find all occurrences of a *set* of patterns in a text. We call such a set of patterns a *dictionary*. In addition to its theoretical importance, Dictionary Matching has many practical applications. For example, in molecular biology, one is often concerned with determining the sequence of a piece of DNA. Having found the sequence (which is simply of string of symbols) the next step is to compare the string against all know strings to find ones that are related. Clearly one would like an algorithm which is minimally dependent on the size of the database of known sequences.

Any pattern matching algorithm can be trivially extended to a set of patterns by matching for each pattern separately. If a given algorithm runs on a text T and a pattern P in time $M(|T|, |P|)$ then the trivial scheme runs on a text T and a dictionary $D = \{P_1, P_2, \ldots, P_k\}$ in time $\sum_{i=1}^{k} M(|T|, |P_i|)$.

In the search for dictionary matching algorithms that are more efficient than the above mentioned

"brute force" method we distinguish between different types of dictionary matching algorithms:

Multiple Matching: An algorithm that requires time $\Omega(|D|+|T|)$ to find all occurrences in T of the dictionary patterns. For example, an optimal exact multiple matching algorithm will run in time $O(|D|+|T|)$ for every text T, even if the dictionary remains unchanged. Parallel versions of exact multiple pattern matching appear in [KLP89] and a more general result appears in [Apo91].

Semi-adaptive Dictionary Matching: An algorithm that preprocesses the dictionary D and subsequently finds all dictionary occurrences in any give text T in time dependent on $|T|$ and $o(|D|)$.

Aho and Corasick [AC75] solved the Exact Dictionary Matching problem. Given a dictionary D whose characters are taken from the alphabet Σ, they preprocess the dictionary in time $O(|D|\log|\Sigma|)$ and then process text T in time $O(|T|\log|\Sigma|)$. This result is perhaps surprising because the text scanning time is *independent* of the dictionary size (for finite alphabet). Aho and Corasick reported their complexity in terms of output size. This is because more than one pattern might match at any given location. They chose to represent their output as a list at each location of all patterns that match at that location. However, when dealing with exact matching, one needs simply to output the *longest* pattern since all information about shorter patterns is contained implicitly in this representation and can be retrieved in constant time. We choose this representation since it is computationally equivalent but the output size is linear.

Adaptive Dictionary Matching: The most general dictionary matching paradigm. An algorithm is adaptive if it is semi-adaptive and if pattern P can be added to or removed from the dictionary in time dependent on $|P|$ and $o(|D|)$.

The Aho and Corasick algorithm provides no mechanism for changing the dictionary. It is easy to see that modifying their dictionary data structure may require work proportional to the size of the entire dictionary. To our knowledge there is no know adaptive dictionary matching algorithm.

Having introduced the notion of adaptive dictionary matching, we will show both sequential and parallel algorithms to solve this problem. In the sequential algorithm, we introduce a novel concept in pattern matching - an adaptive automaton. Our method incorporates a new way of using suffix trees as well as a new data structure in which the suffix tree is embedded. The time complexity of our algorithm is:

Constructing Dictionary:	$O(\|D\|\log\|D\|)$
Adding Pattern P:	$O(\|P\|\log(\|D\|+\|P\|))$
Deleting Pattern P:	$O(\log\|D\|)$
Scanning Text T:	$O(\|T\|\log\|D\|)$.

Note that for unbounded alphabets our algorithm is no worse than the Aho-Corasick semi-adaptive algorithm. This work is presented in section 2.

In section 3 we introduce a new data structure, the *binary label tree* which supports rapid binary-like searches through the substrings of the text. This data structure allow the following bounds to be achieved:

	Time/Processors:
Constructing D:	$O(\log m\log\|D\|)/O(\|D\|/\log m)$
Adding P:	$O(\log m\log\|D\|)/O(\|P\|/\log m)$
Deleting P:	not supported
Scanning T:	$O(\log m\log\|D\|)/O(\|T\|)$.

In section 4 we discuss some open problems suggested by this research.

2 Sequential Adaptive Algorithm

In this section, we present novel algorithms for both the semi-adaptive and fully adaptive exact dictionary matching. Our contribution is to use suffix trees in a new way. Suffix trees have often been used in conjunction with Least Common Ancestor Queries because this combination provides constant time queries about the longest common prefix of two strings (see e.g. [AF91, ALV90, HT84]). We take a different approach and use the suffix tree as a pattern matching machine in the tradition of automaton based pattern matching machines.

2.1 Suffix Tree: Definition and Construction

Definition: Let $C = c_1, ..., c_n$ be a string, where c_n = \$ and the symbol \$ does not appear elsewhere in the string C. We define the *suffix tree* T_C of C as follows:

1. T_C is a tree in which all the edges of the tree are directed away from the root. The out-degree of each node of the tree is either zero (if the node is a leaf) or no less than 2.

2. *Leaves of the suffix tree*: Each suffix $C_i = c_i, ..., c_n$ of the string C defines a *leaf* of the tree. (The tree has n leaves.)

3. *Edges of the suffix tree*: Let E be a contiguous substring of the string C. Let B be a proper prefix of E. Suppose also that both E and B define nodes of the tree. Then there is an *edge* connecting the nodes of E and B if there is no contiguous substring F of the string C such that the following three conditions hold at once: F is a proper prefix of E, B is a proper prefix of F and F defines a node of the tree.

Weiner [Wei73] (and McCreight [McC76]) showed a construction for the suffix tree in $O(n)$ time when the size of the alphabet is fixed. If the alphabet of the pattern is Σ then it is easy to adapt the algorithm of [Wei73] to run in time $O(n \log |\Sigma|)$. The details of the Weiner algorithm are far too involved for inclusion in this abstract. However, certain details are important for the algorithm presented in the next section. We therefore give a high level overview of the Weiner algorithm.

Let T_X be the suffix tree of the string $X = x_1, \ldots, x_n$. Further, let $T_{\geq i}$ be the suffix tree of the substring $x_i, \ldots, x_n$. The Weiner algorithm proceeds by taking $T_{\geq i}$ and updating it to create $T_{\geq i-1}$. This step takes amortized constant time (for finite alphabet) and therefore the construction takes linear time.

While the amortized cost is constant, the maximum possible cost for inserting a new suffix is of the order of the depth of the last inserted node in a tree. For notational ease, we denote the depth of the last inserted suffix in T_X as $d(T_X)$. The worst case cost of converting $T_{\geq i}$ to $T_{\geq i-1}$ is $O(d(T_{\geq i}))$. Therefore the cost of transforming T_X to T_{YX} for strings X and Y is $O(|Y| + d(T_X))$. Let \$ be a symbol that does not appear in X (\$ was used in the definition of the suffix tree as a unique element, however, we can always generate a new unique element when needed therefore this \$ might be more precisely called \$'). Then building $T_{\$X}$ takes $O(|T|)$. But since the suffix $\$x_1x_2 \ldots x_n$ shares no initial characters with any other suffix of $\$X$, then $d(T_{\$X}) = O(1)$. Therefore changing $T_{\$X}$ to $T_{Y\$X}$ takes time $O(|Y| \log |\Sigma|)$. Finally we denote T_D as the suffix tree of the concatenation of the patterns in D. $T_{\$D}$ is defined similarly.

2.2 New Semi-adaptive Algorithm

Before showing the fully adaptive dictionary matching algorithm, we present a new semi-adaptive algorithm with the same complexity as the Aho and Corasick algorithm.

Suppose we have built $T_{\$D}$ for some dictionary D in time $O(|D| \log |\Sigma|)$. Now consider any node v in $T_{\$D}$. Each node v has some string s_v associated with it (s_v is simply the concatenation of all the edge labels from the root of the tree to v). Each node v may have prefixes which are patterns in D. Let $L(v)$ be the number of the longest pattern which is a prefix of s_v. Suppose that both P_i and P_j are prefixes of s_v and no other pattern is a prefix of s_v. Then if $|P_i| > |P_j|$, $L(v) = i$. If no pattern is a prefix of x_v, then $L(v) = 0$. We can assign these labels in linear time using a depth first search traversal of $T_{\$D}$ (the implementation details are left to the journal paper).

Now suppose we convert $T_{\$D}$ to $T_{T\$D}$ for some text T. Then there is a leaf in $T_{T\$D}$ corresponding to each suffix in the string $T\$D$. In particular, there is a leaf l_i corresponding to the string $t_i, \ldots, t_n, \$, D$. We now consider the label $L(p(l_i))$ where $p(v)$ is the parent of v in $T_{T\$D}$. Then $P_{L(p(l_i))}$ is a prefix of string $s_{p(l_i)}$. But $s_{p(l_i)}$ is a prefix of s_{l_i} and therefore $P_{L(p(l_i))}$ is a prefix of s_{l_i}. We conclude that pattern $L(p(l_i))$ matches at location i. Since this check can be accomplished in constant time, the time for this algorithm is simply the time required to convert $T_{\$D}$ to $T_{T\$D}$ which is $O(|T| \log |\Sigma|)$. This complexity bound matches the Aho and Corasick bound.

However, the introduction of a new pattern into $T_{\$D}$ may require the reassignment of many node labels. That is, the new pattern P_{k+1} may be the longest pattern prefix of a large number of nodes in the tree. In the next subsection, we present a new data structure which allows us to rapidly obtain and update the L labels.

2.3 Fully-adaptive Algorithm

To allow for efficient searching and modification of the the L labels, we will impose a higher order structure on the suffix tree. First, we define the notion of a *marked* node.

Definition:

1. Let v be a node such that s_v has some pattern P_i in D as a prefix. Further let $s_{p(v)}$ be a *proper prefix* of P_i. Then v is *marked*.
2. Every leaf is *marked*.
3. No other nodes are marked.

Finding the L value of a node will now require the following steps:

Suppose we wish to find the value $L(v)$ of some node v.

1. Jump down in the tree to some descendent u of v such that u is marked and no node on the path from v to u is marked.

2. Find $L(u)$

Then we know that $L(v) = L(u)$. Therefore we need to know how to find a marked descendant and how to find the L label of a marked node.

To find a marked descendant, we will keep a pointer $N(v)$ at each node v which points to some marked descendant. When we insert a new pattern P_{k+1} into D, we may have to update some N pointers. However, note that this only happens when $s_v \leq P_{k+1} \leq s_{N(v)}$ (were $\leq$ is the "is a prefix of" relation). Furthermore, since P_{k+1} has $|P_{k+1}|$ prefixes, there can be at most $|P_{k+1}|$ nodes which need their pointers modified. Once we locate the node v such that $s_{p(v)} < P_{k+1} \leq s_v$, then we mark v and for each node u on the path from the root to v, if $|s_{N(u)}| > |P_{k+1}|$ then we set $N(u) = v$.

Finding v in the suffix tree takes $O(|P_{k+1}| \log |\Sigma|)$ and marking all the nodes on the path takes $O(|P_{k+1}|)$, therefore, this step takes $O(|P_{k+1}| \log |\Sigma|)$.

It remains to be shown how we update the L values of marked nodes. We impose a secondary tree on the marked nodes to supplement their relationship within the suffix tree. We will call this secondary tree the L-tree since we will use the tree to find the L labels of the nodes. The L-tree is a rooted tree of degree bounded by $|D|$. Finding the L value in the suffix tree is equivalent to finding the parent in the L-tree. However, the L-tree must also support the reparenting operation in time sublinear in the number of nodes reparented. Otherwise, inserting a pattern would take up to $O(|D|)$. Note that when we reparent some of the nodes of a tree, we always reparent a *continuous subsection* of the children of a node.

To achieve the desired complexity, we keep the list of children of a node in a 2-3 tree. Finding the parent in the L-tree becomes equivalent to finding the root of the 2-3 tree, which takes $O(\log |D|)$ time. Furthermore, suppose v is the root of some 2-3 tree within the L-tree. Suppose also that node u is a non-marked descendant of v which we mark upon insertion of some new pattern into the dictionary. Now there exist some marked nodes w and y such that all nodes between w and y inclusive must be made children of u instead of v. We exploit the *concatenate* and *split* operations of 2-3 trees (see e.g. [AHU74]) which can be implemented each in time $O(\log |D|)$. The steps required are as follows:

1. Let T_1 be the 2-3 tree associated with v
2. *split* T_1 into $T_{<w}$ and $T_{\geq w}$
3. *split* $T_{\geq w}$ into $T_{>y}$ and $T_{\geq w, \leq y}$
4. Set u to be the root of $T_{\geq w, \leq y}$
5. *concatenate* $T_{<w}$ and $T_{>y}$. Call this tree T_2
6. *insert* u into T_2. Set v to be the root of T_2

Each of these steps requires at most $O(\log |D|)$ time.

Finally, when scanning a text, we can find the L label of any node in $O(\log |D|)$ time and therefore text scanning takes $O(|T| \log |D|)$ time. When inserting a pattern P_{k+1}, we create $|P_{k+1}|$ new leaves in the suffix tree and one new marked node. Each leaf must be inserted into a 2-3 tree within the L-tree and the new marked internal node must be inserted by the procedure outlined above. Each such step takes $O(\log |D|)$ and so P_{k+1} can be inserted in time $O(|P_{k+1}| \log |D|)$. Finally, to remove a pattern, we need only reverse the steps outlined for inserting a new marked node. Note that the pattern is not actually removed from the suffix tree but it no longer has a labelled node. We can remove the pattern in amortized $O(|P| \log |D|)$ by reprocessing the dictionary once we have a fixed constant percent "junk" patterns [1]. Our worst case time for removing the pattern labels is then $O(\log |D|)$.

To summarize, the combination of the L-tree with the suffix tree provides a data structure which supports the desired operations in time:

Constructing Dictionary:	$O(\|D\| \log \|D\|)$
Adding Pattern P:	$O(\|P\| \log(\|D\| + \|P\|))$
Deleting Pattern P:	$O(\log \|D\|)$
Scanning Text T:	$O(\|T\| \log \|D\|)$.

3 Parallel Algorithm

The previous section described an augmentation of the suffix tree which allowed the Wiener [Wei73] algorithm to be used as the state transition apparatus of a pattern matching machine. In [AIL+88], a $O(\log n)$ time, $O(n \log n)$ work CRCW PRAM algorithm was shown for constructing the suffix tree of a string of length n, however, this algorithm requires quadratic space. If a limitation of linear space is imposed, then the computation bounds increase to $O(\log^2 n)$ time, $O(n \log^2 n)$ work. This algorithm is based on the *label doubling* technique introduced in [KMR72]. We also used a modified version the label doubling technique. However, rather than using the labels to build a suffix tree as in [AIL+88], we construct a new type of tree,

[1] Note that Galil and Park have observed that with a small modification, this the data structure can support pattern removal in worst case $O(|P| \log |D|)$

the *binary label tree* on the dictionary which supports parallel adaptive dictionary matching.

3.1 Label Doubling

Given a string $S = s_1, s_2, \ldots, s_n$, the "standard" label doubling of S proceeds as follows.

For each position x we assign a "label", $l_0(x)$ as follows. We use the location of some occurrence of the letter at each location as its label. For example, suppose that we have the string $S = \langle a, a, b, c, a, c, b \rangle$. A possible labeling of S would be $\langle 5, 5, 3, 4, 5, 4, 3 \rangle$. (Note that for an unbounded alphabet, this step has a lower bound of $\Omega(n \log n)$ work.)

We proceed by "doubling" the labels so that at stage i, a label represents a string of length 2^i. The rule can be written concisely as $l_i(x) = f_i(l_{i-1}(x), l_{i-1}(x+2^{i-1}))$ were f_i is a one-to-one function to be specified below. The upshot is that $l_i(x)$ is a unique identifier of the string $s_x, \ldots, s_{x+2^i}$.

To produce an one-to-one function f, we proceed as follows. Suppose we wish to compute $f_i(\alpha, \beta)$. Further, suppose that these labels occur exactly at positions $(x_1, x_1+2^i), (x_2, x_2+2^i), \ldots, (x_k, x_k+2^i)$. Then $f_i(\alpha, \beta) = x_j$ for some $1 \leq j \leq k$. In words, f_i returns the value of some occurrence of the string represented by the pair $\langle \alpha, \beta \rangle$.

Ignoring boundary cases, this is the basic idea of the label doubling technique. In [AIL+88], it is shown how to achieve such a labeling in logarithmic time using a quadratic space bulletin board, or in logarithm squared time using linear space. The following sections will use various modifications of this basic theme.

3.2 Building the Dictionary

In the generic doubling scheme, the input consists of a single string S. This algorithm is changed in two ways for dictionary matching. The first modification is to allow doubling on a set of strings. Given strings $S_1, S_2, \ldots, S_k$ such that $m = \max\{|S_i|\}_{i=1}^k$, we compute $f_1, f_2, \ldots, f_{\lfloor \log m \rfloor}$ in much the same way as it was computed for the single string case. The labels now take the form $l_i(j, k)$ where j represents the string number and k is the position within string S_j that is being labeled.

A more important difference is the following. In the generic scheme, each position j of string i receives labels $l_1(i, j), l_2(i, j), \ldots, l_{\lfloor \log m \rfloor}(i, j)$. We introduce a new data structure, the *binary label tree* which depends only on labels $l_k(i, j)$ where 2^k divides j but 2^{k+1} does not, for $1 \leq \lfloor \log m \rfloor$. In other words, we compute l_0 for all j, l_1 for all even j, l_2 for all j divisible by 4, etc. Clearly this reduces the amount of work by a factor of $\lfloor \log m \rfloor$. The standard doubling technique needs little modification to compute the reduced set of labels.

The actual method to compute the f_i's is as follows. We take an array of all the label pairs and sort them, also eliminating redundant entries in this process. Then we assign to each pair a new label corresponding to whichever pair survived the redundancy elimination (i.e. an arbitrary occurrence). Finally, we put the new triplet of labels in a 2-3 tree. This step can be accomplished in $O(\log \frac{|D|}{2^i})$ time with $O(\frac{|D|}{2^i})$ processors for f_i (see [PVW83] for details of parallel 2-3 insertion, etc.). Therefore, the labeling step takes $O(\log m \log |D|)$ time and $O(\frac{|D|}{\log m})$ processors.

A *binary label tree* is a tree defined as follows. Each node n in the tree has associated $\mathit{length}(n) \in \{1, \ldots, m\}$ and $\mathit{strings}(n) \subseteq \{1, \ldots, k\}$ such that $\mathit{strings}(n)$ is an equivalence class of the relation $=_d$ such that $i =_d j$ iff $S_{i,1}S_{i,2}\ldots S_{i,d} = S_{j,1}S_{j,2}\ldots S_{j,d}$. The out edges of n are organized as follows: suppose that 2^k divides $\mathit{length}(n) + 1$ but 2^{k+1} does not. Then the out edges are organized into clusters $\mathit{out}_0(n), \mathit{out}_2(n), \ldots, \mathit{out}_k(n)$. Each $\mathit{out}_i(n)$ is a 2-3 tree whose leaves are pointers to other nodes. Furthermore, if there is a pointer p to node m in $\mathit{out}_i(n)$, then $\mathit{length}(m) = \mathit{length}(n) + 2^i$ and $\mathit{strings}(m) \subseteq \mathit{strings}(n)$. Clearly the equivalence classes within some given $\mathit{out}_i(n)$ cluster are also the equivalence classes of the $l_i(j, \mathit{length}(n))$ for $j \in \mathit{strings}(n)$. Thus, this part of the binary label tree can be computed in the same resource bounds as the label computation (full details will appear in final version).

The second part of the binary label tree keeps information about which patterns are prefixes of other patterns. To motivate the need for this part of the data structure (as well as the first part of the data structure), we outline the text scanning algorithm (see subsection 3.4 for more details).

During text scanning, the labels $l_0(T, i), l_1(T, i), \ldots, l_{\lfloor \log m \rfloor}(T, i)$ are generated for $1 \leq i \leq |T|$. Then, for each location i of the pattern a "binary search" on the substrings of T is performed using the labels. We start at the root of the tree and find the pointer in $\mathit{out}_{\lfloor \log m \rfloor}(\mathit{root})$ which corresponds with $l_{\lfloor \log m \rfloor}(T, i)$. If we find $l_{\lfloor \log m \rfloor}(T, i)$ in $\mathit{out}_{\lfloor \log m \rfloor}(\mathit{root})$ then we proceed to that node (call it n) and find $l_{\lfloor \log m \rfloor - 1}(T, i+2^{\lfloor \log m \rfloor})$ in $\mathit{out}_{\lfloor \log m \rfloor - 1}(n)$, etc. If we fail to find, at some node m $l_j(T, i+k)$ in $\mathit{out}_j(m)$ for all $j < l$ (for appropriate value of i,j,k,l), that is, if we can no longer extend the match, then

we must find the longest pattern that is a prefix of the string represented by m. Therefore, we keep the following information about pattern prefixes.

Recall that each node of the tree has an associated *length*. For each *length*, we keep information about which patterns are prefixes of which nodes. If we were to keep the information about pattern prefixes associated with each node, then, upon introduction of a new pattern, $O(|D|)$ nodes could require updating, thus violating the requirement for adaptivity. Therefore we borrow an idea of from the sequential algorithm and keep the nodes in a set of 2-3 trees. To find which pattern is the longest prefix of a given node, we find the root of its 2-3 tree. We apply the same techniques as in the case of the sequential algorithm to split and reglue the 2-3 trees. Space considerations do not allow inclusion of all the details but by analogy to binary search, for any given pattern, we can update $O(\log m)$ such 2-3 trees rather than all the trees.

The overal complexity to build the binary label tree with associated pattern prefix information is $O(\log m \log |D|)$ time and $O(\frac{|D|}{\log m})$ processors.

3.3 Adding a Pattern

To insert a new pattern P in the binary label tree, we first compute the labels of substrings of P. Once again we need only compute a subset of the labels. To compute $f_i(\alpha, \beta)$, we search for $\langle \alpha, \beta, \gamma \rangle$ in the 2-3 tree representing f_i. If such a triplet exists, then γ is the desired label. For those $\langle \alpha, \beta \rangle$ pairs which do not occur, assign a new label as we did before (by sorting and removing redundancies) and then insert the new triples into the 2-3 tree (using the [PVW83] algorithm). The overall bounds are $O(\log |P| \log m)$ time and $O(\frac{|P|}{\log |P|})$ processors.

3.4 Scanning a text

The basic idea for the text scanning algorithm was described subsection 3.2. The label doubling phase requires $O(\log m)$ doubling phases and each phase takes $O(|D|)$ to determine the value of f_i at each location. Thus this part takes $O(\log m \log |D|)$ time and $O(n)$ processors. The binary search phase requires $O(n)$ processors and the search through the binary label tree takes $O(\log m)$ stages which require $O(\log |D|)$ time each. Therefore the overall resource bound for the text scan in $O(\log m \log |D|)$ time and $O(n)$ work.

4 Open Problems

Our serial adaptive algorithm has a $\log |D|$ factor even for a finite alphabet. Can this factor be done away with? The parallel algorithm we presented was not optimal for the text scanning phase. It would be of interest to come up with an optimal speedup algorithm. Finally, we are also interested in various extensions. No results are known for efficient approximate dictionary matching (such algorithms would be enormously important in molecular biological applications). Also, extensions to two or more dimensions are of interest.

5 Acknowledgments

The authors warmly thank Alberto Apostolico for hours of fruitful discussions.

References

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching. *C. ACM*, 18(6):333–340, 1975.

[AF91] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of 2nd Symoposium on Descrete Algorithms, San Fransisco, CA*, Jan 1991.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[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.

[ALV90] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Proceedings of First Symposium on Discrete Algorithms, San Fransisco, CA*, 1990.

[Apo91] A. Apostolico. Efficient parallel multiple matching. Personal Comunication, 1991.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[GS83] Z. Galil and J.I. Seiferas. Time-space-optimal string matching. *J. Computer and System Science*, 26:280–294, 1983.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[KLP89] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 6:388–398, 1989.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[KMR72] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *Symposium on the Theory of Computing*, 4:125–136, 1972.

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

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

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Vis89] U. Vishkin. Deterministic sampling for fast pattern matching. Manuscript, 1989.

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