Text Indexing and Dictionary Matching with One Error*

Amihood Amir†‡ Bar-Ilan University and Georgia Tech

Dmitry Keselman§ Simons Technologies

Gad M. Landau¶ Haifa University and Polytechnic University

Moshe Lewenstein† ‖ Bar-Ilan University

Noa Lewenstein† ** Bar-Ilan University

Michael Rodeh †† IBM Research

Abstract

The *indexing problem* is the one where a text is preprocessed and subsequent queries of the form: "Find all occurrences of pattern P in the text" are answered in time proportional to the length of the query and the number of occurrences. In the *dictionary matching problem* a set of patterns is preprocessed and subsequent queries of the form: "Find all occurrences of dictionary patterns in text T" are answered in time proportional to the length of the text and the number of occurrences.

There exist efficient worst-case solutions for the indexing problem and the dictionary matching problem, but none that find *approximate* occurrences of the patterns, i.e. where the pattern is within a bound edit (or Hamming) distance from the appropriate text location.

In this paper we present a uniform deterministic solution to both the indexing and the general dictionary matching problem with one error. We preprocess the data in time $O(n \log^2 n)$, where n is the text size in the indexing problem and the dictionary size in the dictionary matching problem. Our query time for the indexing problem is $O(m \log n \log\log n + tocc)$, where m is the query string size and $tocc$ is the number of occurrences.

Our query time for the dictionary matching problem is $O(n \log^3 d \log\log d + tocc)$, where n is the text size and d the dictionary size.

The time bounds above apply to both bounded and unbounded alphabets.

*A preliminary version of this paper appeared in WADS'99 [5]

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; {amir,moshe,noa}@cs.biu.ac.il

‡Partially supported by NSF grant CCR-96-10170 and BSF grant 96-00509.

§Simons Technologies, One West Court Square, Decatur, GA 30030, (404)370-7010; dkeselman@hasimons.com.

¶Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, phone: (718) 260-3154, FAX: (718) 260-3906; Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; email: landau@poly.edu; partially supported by NSF grant CCR-9610238, and by the Israel Science Foundation founded by the Israeli Academy of Sciences and Humanities.

‖Supported by an Eshkol Fellowship from the Israel Ministry of Science and the Arts.

**Partially supported by the Israel Ministry of Science and the Arts grant 8560.

††IBM Research Lab in Haifa, MATAM – Advanced Technology Center, Haifa, 31905. email: rodeh@il.ibm.com.

1 Introduction

The well known string matching problem that appears in all algorithms textbooks has as its input a text T of length n and pattern P of length m over a given alphabet Σ. The output is all text locations where there is an exact match of the pattern. This problem has received much attention, and many algorithms were developed to solve it (e.g. [20, 15, 22, 7, 21]). A detailed modern view of stringology can be found in a number of recently published books [6, 10, 18].

Two important more general models have been identified quite early, *indexing* and *dictionary matching*. These models have attained an even greater importance with the explosive growth of multimedia, digital libraries, and the Internet.

1.1 The Indexing Problem

The *indexing* problem assumes a (usually very large) text that is to be preprocessed in a fashion that will allow efficient future queries of the following type. A query is a (significantly shorter) pattern. One wants to find all text locations that match the pattern in time proportional to the *pattern length and number of occurrences.*

Weiner [27] invented the *suffix tree* data structure whereby the text is preprocessed in linear time, and subsequent queries of length m get answered in time $O(m + tocc)$, where $tocc$ is the number of pattern occurrences in the text. The times above are to be multiplied by $\log m$ in case the alphabet size is unbounded.

Weiner's suffix tree in effect solved the indexing problem for exact matching of fixed texts. Succeedingly improved algorithms for the indexing problem in *dynamic* texts were suggested, for example by [16, 12, 13, 26]. No algorithm is currently known for *approximate indexing*, i.e. the indexing problem where up to a given number of errors is allowed in a match. The problem is formally defined as follows.

- *Input:* Text T of length n over alphabet Σ and integer k.
- *Query:* Pattern P of length m over alphabet Σ.
- *Goal:* Preprocess T as fast as possible, and answer a length-m query in time as close to $O(m + tocc)$ as possible, where $tocc$ is the number of pattern occurrences in the text with at most k mismatches.

We note that there are several definitions of errors. The *edit distance* allows for mismatches, insertions and deletions [23], the *Hamming distance* allows for mismatches only. Throughout the rest of this paper we will discuss a mismatch error, but a similar treatment will handle insertions and deletions.

1.2 The Dictionary Matching Problem

The *dictionary matching* problem is, in some sense, the "inverse" of the indexing problem. The large body that needs to be preprocessed is a set of patterns, called the *dictionary.* The queries are texts whose length is typically significantly smaller than the total dictionary size. It is desired to find all (exact) occurrences of dictionary patterns in the text in time proportional to the *text length and number of occurrences*.

Aho and Corasick [1] gave an automaton-based algorithm that preprocesses the dictionary in time $O(d)$ and answers a query in time $O(n + tocc)$. A logarithmic multiple is present for alphabets of unbounded size. Efficient algorithms for a dynamic dictionary appear in [2, 3, 19, 4, 26]. As in the indexing case, the *approximate dictionary matching problem*, where all pattern occurrences with at most a given number of errors, has proven elusive. The problem is formally defined as follows.

- *Input:*
  1. A dictionary $D = \{P_1, ...P_s\}$, where P_i, $i = 1, ..., s$, are patterns over alphabet Σ, and $d = \sum_{i=1}^{s} |P_i|$, is the sum of the lengths of all the dictionary patterns.
  2. An integer k.
- *Query:* Text T of length n over alphabet Σ.
- *Goal:* Preprocess D in time as close to linear as possible, and answer a length-n query in time as close to $O(n + tocc)$ as possible, where $tocc$ is the number of occurrences of dictionary patterns that appear in the text with at most k mismatches.

The *Nearest Neighbor* Problem with the Hamming metric is a special case of this problem. The difference is that in the nearest neighbor problem all dictionary elements are of the same size as well as the text. Moreover, in the dynamic dictionary matching problem one is interested in the nearest dictionary element's *position in the text*, in addition to the nearest neighbor itself.

In a recent paper [28], Yao and Yao give a data structure for deterministically solving the nearest neighbor problem for the case of *one error* in the Hamming metric. They assume that all dictionary patterns are of exactly the same length, and that the query text is also of the same length. The preprocessing time and space of their algorithm is $O(d \log m)$ and the query time is $O(m \log \log s)$. The above result was improved by Brodal and Gąsieniec [8]. They shaved off the log factors of all the results in [28]. It should be noted that both above papers assume a bounded finite alphabet.

As remarked above, both these algorithms can easily be extended for longer query texts by multiplying the query time by the text length n (i.e. the query time for the Yao and Yao algorithm becomes $O(nm \log \log s)$, and the Brodal and Gąsieniec algorithm becomes $O(nm)$). However, their methods rely very heavily on all patterns having the same length.

1.3 Unified Results

In this paper we present a new approach that solves the single error version of both the indexing and dictionary matching problems. Unlike the previous deterministic results, our dictionary may have patterns of different lengths. Our matching algorithm looks for *any* pattern that appears in the text, within error distance one, not just for dictionary patterns whose Hamming distance from the query pattern is one. Throughout the paper we a assume a bounded fixed alphabet Σ for ease of exposition. As is well known, the suffix tree construction and navigation times have a multiplicative $O(\log \sigma)$ factor, where $\sigma = \min(m, |\Sigma|)$, for unbounded alphabet Σ. In fact, the time bounds of this paper's algorithms are dominated by the range query algorithms. The state-of-the-art in range query is such that the time bounds of our algorithms remain the same for unbounded alphabets as well.

We achieve the following time bounds. For the indexing problem, our preprocessing time is $O(n + qp(n)$, where $qp(n)$ is the preprocessing necessary for two dimensional range-query on a grid (see section 5). The current fastest such algorithm is due to Overmars [25] and its preprocessing time and space is $O(n \log^2 n)$. Note that for unbounded alphabets our time would be $O(n \log \sigma + qp(n))$ which is still dominated by $qp(n)$ to be $O(n \log^2 n)$. The query time is $O(mq(n) + tocc)$ ($O(m(\log \sigma + q(n)) + tocc)$ for unbounded alphabets). Again, using Overmars algorithm we achieve $q(n) = \log n \log \log n$ for a total query time of $O(m \log n \log \log n + tocc)$ for both bounded and unbounded alphabets.

For dictionary matching our preprocessing time is $O(d + tp(d))$, where $tp(d)$ is the preprocessing necessary for the tree path intersection problem (see section 9). We show that the tree path intersection time can be solved with a preprocessing time and space of $tp(d) = O(d \log^2 d)$. Note that, again, for unbounded alphabet the time is $O(d \log \sigma + tp(d)) = O(d \log^2 d)$. Our query time is $O(nt(d))$ ($O(n(\log \sigma + t(d))$ for unbounded alphabets), where $t(d)$ is the time for a tree path intersection query. We show that a solution where $t(d) = \log^3 d \log \log d + tocc$, where $tocc$ is the number of occurrences found, is possible. Thus our query time is $O(n \log^3 d \log \log d + tocc)$, for both bounded and unbounded alphabets.

Both of our algorithms combine a bidirectional construction of suffix trees, similarly to the data structure in [8], with range queries for efficient set intersections. The latter is an interesting problem in its own right.

Belween the submission of this paper and its seeing print, Ferragina, Muthukrishnan and deBerg [14] described a geometric data structure that can be applied to approximate dictionary matching. Although their text query time is faster than ours ($O(n \log \log d + tocc)$), their data structure uses more space ($O(d^{1+\epsilon})$ for any given ϵ). Our approaches to the problem are also different in that they are concerned with a geometric data structure and its applications, and we seek a simple unified approach to approximate text indexing and dictionary matching.

Paper organization. This paper is organized in the following way. In Section 2 we describe suffix trees and their use for indexing. In Section 3 we describe our idea for bidirectional suffix tree construction in order to solve the one error indexing problem. Section 5 is self contained, and describes how to efficiently compute set intersection using range queries. This is used as a subroutine in our algorithm. Section 7 describes the Amir-Farach [2] idea of using suffix trees for dictionary matching, and Section 8 describes our use of suffix trees for dictionary matching with one error. Section 9 is also self contained. It describes an efficient method of computing set intersections where the sets are labels on tree paths. We conclude with a brief discussion and open problems.

2 Suffix Trees and the Indexing Problem

Definition 2.1 *A trie T for a set of strings $\{S_1, \cdots, S_r\}$ is a rooted directed tree satisfying:*

1. *Each edge is labeled with a character, and no two edges emanating from the same node have the same label.*

2. *Each node v is associated with a string, denoted by $L(v)$, obtained by concatenating the labels on the path from the root to v, $L(root)$ is the empty string.*

3. *There is a node v in T if and only if $L(v)$ is a prefix of some string S_j in the set.*

A *compacted trie* T' is obtained from T by collapsing paths of internal nodes with only one child into a single edge and by concatenating the labels of the edges along the path to form the label of the new edge. The label of an edge in T' is a nonempty substring of some S_j, and it can be succinctly encoded by the starting and ending positions of an occurrence of the substring. The number of nodes of a compacted trie is $O(r)$.

Let $S[1, m] = s_1 s_2 \cdots s_{m-1}\$$ be a string, where the special character \$ is not in Σ. The *suffix tree* T_S of S is a compacted trie for all suffixes of S. Since \$ is not in the alphabet, all suffixes of S are distinct and each suffix is associated with a leaf of T_S. There are several papers that describe linear time algorithms for building suffix trees, e.g. [27, 24, 9, 11].

Fact 2.2 *Let S_T be the suffix tree of text T. Let $P = p_1 \cdots p_m$ be a pattern. Start at the suffix tree root and follow the labels on the tree as long as they match $p_1 \cdots p_m$. If at some point there is no matching label, then P does not appear in T. Otherwise, let v be the closest node (from below) to the label where we stopped. The starting location of the suffixes that are leaves in the subtree rooted at v are precisely all text locations where the pattern appears. These locations can be located and listed, for a fixed bounded alphabet, in time $O(m + tocc)$.*

3 Indexing with One Error – Bidirectional Use of Suffix Trees

For simplicity's sake we make the following assumption. *Assume that there are no* **exact** *matches of the pattern in the text.* We will relax this assumption later. In Section 6 we will handle the case where there may be exact matches of the pattern in the text and we are interested in all occurrences with *exactly* one error.

The main idea: Assume there is a pattern occurrence at text location i with a single mismatch in location $i + j - 1$. This means that $p_1 \cdots p_{j-1}$ has an *exact match* at location i and $p_{j+1} \cdots p_m$ has an exact match at location $i + j$.

The distance between location i and location $i + j$ is dependent on the mismatch location, and that is somewhat problematic. We therefore choose to "wrap" the pattern around the mismatch. In other words, if we stand exactly at location $i + j - 1$ of the text and look left, we see $p_{j-1} \cdots p_1$. If we look right we see $p_{j+1} \cdots p_m$.

This leads to the following algorithm.

Algorithm Outline

Preprocessing:

P.1 Construct a suffix tree S_T of text string T and suffix tree S_{T^R} of the string T^R, where T^R is the reversed text $T^R = t_n \cdots t_1$.

P.2 For each of the suffix trees, link all leaves of the suffix tree in a left-to-right order.

P.3 For each of the suffix trees, set pointers from each tree node v to its leftmost leaf v_l and rightmost leaf v_r in the linked list.

P.4 Designate each leaf in S_T by the starting location of its suffix. Designate each leaf in S_{T^R} by $n - i + 3$, where i is the starting position of the leaf's suffix in T^R.
{ The leaf designation was made to coincide for a left-segment and right segment of the same error location. }

Query Processing:

For $j = 1, ..., m$ do

1. Find node v, the location of $p_{j+1} \cdots p_m$ in S_T, if such a node exists.
2. Find node w, the location of $p_{j-1} \cdots p_1$ in S_{T^R}, if such a node exists.
3. If v and w exist, find intersection of leaves in the subtrees rooted at v and w.

end Algorithm Outline

In Section 4 we will show how to process Steps 1 and 2 of the query processing for the j's in *overall* linear time, and in Section 5 we will see an efficient implementation of Step 3 of the query processing.

4 Navigating on the Suffix Trees

Assume that we have found the node in S_T where $p_j \cdots p_m$ resides. We would like to move to the node where $p_{j+1} \cdots p_m$ appears. Similarly, if we have the node in S_{T^R} where $p_i \cdots p_1$ ends, we would like to arrive at the node where $p_{i+1} \cdots p_1$ ends.

In order to achieve this we review the traits of one of the algorithms for linear time suffix tree constructions – Weiner's algorithm [27]. There is no need to understand the details of the algorithm. The only necessary information is the following.

Let $S = s_1 \cdots s_n\$$ be a string. The Weiner construction of suffix tree starts with the suffix \$, then adds the suffixes $s_n\$$, $s_{n-1}s_n\$$, ... , $s_1 s_2 \cdots s_n\$$. The total construction time is linear for finite fixed alphabets.

Consider the string $p_m p_{m-1} \cdots p_1 \% T^R$, where $\% \notin T^R$. The Weiner construction will at some point have the suffix tree for T^R, then add $p_1\%T^R$, $p_2p_1\%T^R$, ... ,$p_m p_{m-1} \cdots p_1 \% T^R$. As pointed out in [2], because $\% \notin T^R$, the total time for Weiner's algorithm to add all the suffixes that start at the pattern is $O(m)$.

The suffix tree part for T^R is precisely S_{T^R}, and this part is constructed during the preprocessing phase. For every query pattern, we simply continue the Weiner construction. This, in effect, finds for us the locations we desire in total linear time. When the query is over, we retrace our steps and remove the pattern parts from S_{TR}.

The case for the tree S_T is similar. Consider the string $p_1 p_2 \cdots p_m \% T$, where $\% \notin T$. The Weiner construction will at some point have the suffix tree for T, which is the S_T part done during preprocessing. Then the Weiner construction adds $p_m\%T$, $p_{m-1}p_m\%T$, ... ,$p_1 p_2 \cdots p_m\%T$. This, in reality, also finds all locations we are interested in, but the order they are encountered is reversed. This fact can be simply circumvented by keeping an array of pointers to all the necessary nodes, and following that array backwards in lockstep with the forward movement on tree S_{TR}.

One last necessary detail would be to have a special tag on the new nodes (added during the query phase) in order to tell if the final nodes mean an occurrence in the original tree or not.

This use of Weiner's construction clearly indicates that it is indeed possible to navigate the trees in linear time in the size of the pattern. In the next section we show how to extract the intersection of two sets in time proportional to the intersection.

5 Efficient Set Intersection via Range Queries

In Step 3, given nodes v and w, we want to find the leaves that appear both in intervals $[v_l ... v_r]$ and $[w_l ... w_r]$, where the four endpoints of the two intervals are defined in Step P.3 of the preprocessing. Thus, we are seeking a solution to the following problem.

Definition 5.1 *The subarray intersection problem is defined as follows. Let $V[1..n]$ and $W[1..n]$ be two permutations of $\{1, ..., n\}$. Preprocess the arrays in efficient time in a fashion that allows fast solution to the following query.*

Query: Find the intersection of elements of $V[i..i+k]$ and $W[j..j+\ell]$.

We now show that this problem is just a different formulation of the well-studied *range searching on a grid* problem. In the range searching on a grid problem, one is given n points in 2-dimensional space. These points all lie on the grid $[0, 1, ..., U] \times [0, 1, ..., U]$. Queries are of the form $[a, b] \times [c, d]$ and the result are all the points in that matrix.

Set $U = n$. Since the arrays are permutations, every number between 1 and n appears precisely once in each array. The coordinates of every number i are $[x, y]$, where $V[x] = W[y] = i$. It is clear that the range search gives precisely the intersection.

Overmars [25] shows an algorithm that preprocesses the points in time and space $O(n \log n)$ and the query time is $O(k + \sqrt{\log U})$, where k is the number of points in the range $[a, b] \times [c, d]$. Therefore we have the following.

Theorem 5.2 *Let $T = t_1 \cdots t_n$ and $P = p_1 \cdots p_m$. Indexing with one error can be solved with $O(n \log n)$ preprocessing time and $O(tocc + m\sqrt{\log n})$ query time, where tocc is the number of occurrences of the pattern in the text with one error, when no exact match exists.*

Proof: Since Steps P.1 to P.4 can be implemented in $O(n)$ time the total preprocessing time is dominated by the $O(n \log n)$ preprocessing for the subarray intersection.

Step 3 of the query processing can be done in time $O(tocc + \sqrt{\log n})$ by combining Overmar's result with the observation above. We have seen in Section 4 that Steps 1 and 2 of the query processing can be done in total time $O(m)$ for $j = 1, ..., m$. This will bring our total query time to $O(m\sqrt{\log n} + tocc)$. □

6 Indexing with One Error when Exact Matches Exist

Our algorithm assumed that there was no exact pattern occurrence in the text. In fact, the algorithm would also work for the case where there are exact pattern matches in the text, but its time complexity would suffer. Our algorithm's main idea was checking, for every text location, for exact matches of all subpatterns of all lengths to the left and to the right of that location. If there are exact pattern matches, this means that every exact occurrence is reported m times. The worst case could end up being as bad as $O(nm)$ (for example if the text is A^n and the pattern is A^m).

We propose to handle the case of exact occurrences using the following idea. Our navigation down the suffix trees allows us to position ourselves at all text locations that have an exact match to the left and simultaneously all locations that have an exact match to the right. In Section 5 we saw how to efficiently compute the intersection of those two sets. What we currently need is really a third dimension to the range. We actually need the intersection of all suffix labels such that the symbol *preceding* the suffix is *different from* the symbol at that respective pattern location.

We therefore need to make the following additions to the algorithm.

Preprocessing:

{ Recall that each leaf in S_T is designated by the starting location of its suffix, and each leaf in S_{T^R} is designated by $n - i + 3$ where i is the starting location of the suffix in T^R. }

P.5 Add the symbol $T[i-1]$ to the leaf designated by i, in both S_T and S_{T^R}.

Range Query Preprocessing: Preprocess for a 3-dimensional range queries problem on the matrix $[1, ..., n] \times [1, ..., n] \times \Sigma$. If Σ is unbounded, then use only the $O(n)$ symbols in T. As in Section 5, the coordinates of number i with symbol a are $[x, y, a]$, where $V[x] = W[y]$.

The only necessary modification in the query processing part of the algorithm is in Step 3 which becomes:

3. If v and w exist, find intersection of leaves in the subtrees rooted at v and w where the attached alphabet symbol is not the respective pattern mismatch symbol.

The above step can be implemented by two half infinite range queries on the three dimensional range $[1, a-1] \times [v_\ell, v_r] \times [w_\ell, w_r]$ and $[a+1, |\Sigma|] \times [v_\ell, v_r] \times [w_\ell, w_r]$, where we assume that the alphabet symbols are numbered $1, ..., |\Sigma|$.

Theorem 6.1 *Let $T = t_1 \cdots t_n$ and $P = p_1 \cdots p_m$. Indexing with one error can be solved with $O(n \log^2 n)$ preprocessing time and $O(tocc + m \log n \log\log n)$ query time, where tocc is the number of occurences of the pattern in the text with at most one error.*

Proof: The time of the preprocessing stage is affected only by the range query preprocessing. By Theorem 7.2 of [25], half infinite range queries on the three dimensional range can be preprocessed in time and space $O(n \log^2 n)$. Therefore $O(n \log^2 n)$ is the total preprocessing time.

If we use the method suggested in [25] and give the highest level to one of the index coordinates and use the alphabet symbol as the second coordinate, such a query can be implemented in time $O(tocc + \log n \log\log n)$. $\Box$

7 Suffix Trees and Dictionary Matching

We use a similar technique to the one used by Amir and Farach [2] for dictionary matching and by Gusfiled, Landau and Schieber [17] in their generalized suffix trees. The idea is the following.

Consider the dictionary D. To simplify notation we also denote with D the concatenation of all dictionary patterns, with a separator, \$, not in Σ, at the end of each pattern. Construct suffix tree S_D of D. Mark the leaves that start in a dictionary pattern.

When a query text arrives, add it to the suffix tree in the Weiner fashion described in Section 4. Every node touched by a text suffix that has a marked child designates an occurrence of that dictionary pattern.

8 Dictionary Matching with One Error

Our aim is to use the bidirectional suffix tree idea of Section 3 combined with dictionary matching via suffix trees as described in Section 7.

Construct suffix trees S_D and S_{D^R}, where D is a string of the concatenated dictionary patterns, separated by the separating symbol \$. Upon the arrival of a text string T, insert T into S_D and T^R into S_{D^R}. Suppose that for location i we have $t_{i+1} \cdots t_n$ ending in node v of S_D and $t_{i-1} \cdots t_1$ ending in node w of S_{D^R}. Consider the paths from the root to v in S_D and from the root to w in S_{D^R}. Any node in these paths that has a leaf as a direct child whose label begins with a separator is an indication of a pattern suffix that starts at the start of substring $t_{i+1} \cdots t_n$ and matches the appropriate locations in $t_{i+1} \cdots t_n$ until it concludes (or a pattern prefix that starts at t_{i-1} and ends somewhere inside $t_{i-1} \cdots t_1$).

As in the indexing case (Section 3) label pattern substrings in the different trees such that a suffix in S_D has the same label as a prefix in S_{D^R} that starts one location away from the suffix beginning. If that is the case, then our problem is reduced to finding the intersection of the set of labels that are direct children of nodes on the path from the root to v and from the root to w. As in the indexing case, let us assume for now that there is no exact match of any pattern in the text. This assumption can be relaxed in precisely the same manner as in the indexing case.

The above idea suggests an algorithm outline that we will shortly describe. To clarify terminology we say that a specific appearance of the separator symbol in the suffix tree is *edgefirst* if it appears as the first character on an edge. We say that it is *treefirst* if the associated string appearing on the path from the root till this separator does not contain another separator.

Algorithm Outline

Preprocessing:

P.1 Construct a suffix tree S_D of string D and suffix tree S_{D^R} of the string D^R, where D is the concatenation of all dictionary patterns, with a separator at the end of each pattern, and where D^R is the reversal of string D.

P.2 Modify suffix tree S_D, and S_{D^R} respectively, as follows. For each separator which is treefirst but not edgefirst, i.e. it appears on an edge (u, v) labelled $\sigma\$\sigma''$, where $\sigma \neq \epsilon$, break (u, v) into (u, w) and (w, v). Label (u, w) with σ and (w, v) with $\$\sigma'$.
Note: After step P.2 every suffix of a pattern in S_D, and every prefix of a pattern in S_{D^R} respectively, ends in a vertex (with an edge exiting beginning with \$). *So, to consider all the desired suffixes it is sufficient to consider the vertices of the trees.*

P.3 Scan suffix tree S_D, respectively S_{D^R}, and modify as follows. For each vertex v consider the associated string $L(v)$, i.e. the string from the root to v. Label v with all the locations of the pattern suffixes, resp. prefixes, that are equal to $L(v)$. To implement this note that all the relevant suffixes share a prefix of $L(v)\$$. So, go to edge (v, w) with label beginning with \$, assuming such exists, and scan the subtree rooted at w to find all relevant suffixes.

Query Processing:

For $j = 1, ..., n$ do

1. Find node v, the location of the longest prefix of $t_{j+1} \cdots t_n$ in S_D.
2. Find node w, the location of the longest prefix of $t_{j-1} \cdots t_1$ in S_{D^R}.
3. Find intersection of markings of nodes on the path from the root to v in S_D and on the path from the root to w in S_{D^R}.

end Algorithm Outline

Note that in step P.3 of the preprocessing we are simply labeling the two halves of pattern j in both trees with the same label, in a similar fashion to the way it was done in the indexing case.

The navigation on the suffix trees S_D and S_{D^R} is identical to the navigation on trees S_T and S_{T^R} as described in Section 4. We only need to show how to efficiently find the label intersection. This will be seen in the next section.

9 Efficient Set Intersection on Tree Paths

We are seeking a solution to the *tree path intersection problem* defined as follows.

Definition 9.1 *The tree path intersection problem is defined on two trees T_1 and T_2 each of size t with some nodes having labels from set $\{1, ..., t\}$. No two different nodes (in the same tree) have the same label. We would like to preprocess the trees in efficient time in a manner that allows fast solution to the following query.*

Query: For given nodes $v \in T_1$, $w \in T_2$, find the common labels to the path from the root to v in T_1 and to the path from the root to w in T_2.

We will present a method whose preprocessing time is $O(t \log t)$, and where the query time is $O(\log^{2.5} t + int)$, where int is the size of the intersection.

We first need the following lemma that allows tree decomposition into *vertical* paths.

Definition 9.2 *A vertical path of a tree is a tree path (possibly consisting of a single node) where no two nodes on the path are of the same height.*

Let D be a fixed decomposition of the tree nodes into a set of edge-disjoint vertical paths, where every vertex is in some path of D. Each vertex x has a unique path to the tree root. Denote by $LD(x)$ the number of *decomposition paths* that have a non-empty intersection with the unique path from x to the root. Let $LD(T) = \max(LD(x))$ over all tree nodes x.

Lemma 9.3 *Let T be a tree with t vertices. There exists a decomposition D of the tree such that $LD(T) \leq \log(t+1)$.*

Proof: By induction on t. For $t = 1$ it is clear.

Assume now that the lemma is true for all $k \leq t$, prove for a tree of $t+1$ nodes. If the root has a single child it is obvious, simply extend the path that reaches the root's child in the subtree rooted there. Otherwise the root has at least two children.

Let $x_1, ..., x_p$ be the children of the root v, and T_i be the subtree of T, rooted at node x_i, $i = 1, ..., p$. Let t_i be the number of nodes in T_i. Obviously, $t = t_1 + ... + t_p + 1$. Let D_i be an optimal decomposition of T_i, i.e. $LD_i(T_i)$ is smallest. We can assume without loss of generality that $LD_i(T_i) \geq LD_{i+1}(T_{i+1}))$, $i = 1, ..., p-1$. Let D be the union of D_i with v added after x_1 to path of D_1.

We can estimate $LD(T)$. By the inductive assumption, $LD_i(T_i) \leq \log(t_i + 1)$. For any node x in T_i, $LD(x) = LD_i(x) + 1$ for $i > 1$, and $LD(x) = LD_1(x)$ for $x \in T_1$. Then $LD(T) = \max\{LD_1(T_1), LD_i(T_i) + 1,\ i \geq 2\}$.

If $LD_2(T_2) + 1 \leq LD_1(T_1)$, then the inductive step is clear, since then $LD(T) = LD_1(T_1)$.

Let us assume that $LD_2(T_2) + 1 > LD_1(T_1)$. Then $LD(T) = LD_2(T_2) + 1$. If $t_1 < t_2$, then $LD(T) = LD_2(T_2)+1 \leq LD_1(T_1)+1 \leq \log(t_1+1)+1 = \log(t_1+t_1+2) \leq \log(t_1+t_2+1) < \log(t+1)$. If $t_1 \geq t_2$, then $LD(T) = LD_2(T_2)+1 \leq \log(t_2+1)+1 = \log(t_2+t_2+2) \leq \log(t_1+t_2+2) \leq \log(t+1)$.

This completes the inductive step and concludes the proof of the lemma. □

The above lemma is constructive. It is clear that the decomposition can be constructed in linear time. It may be seen from the proof that the decomposition of D is constructed from the bottom to the top in a fashion where $LD(T)$ is smallest possible. The bound in the lemma is tight and is reached on the complete binary tree.

Returning to our tree path intersection problem we can show the following.

Lemma 9.4 *The tree path intersection problem can be solved with $O(t \log t)$ preprocessing time and $O(int + \log^{2.5} t)$ query time.*

Proof: Decompose both T_1 and T_2 in the manner described in the proof of Lemma 9.3.

The labels on the trees can be preprocessed for range queries on a grid using the decomposition as follows. The labels on every path on the decomposition are ordered by their heights. Concatenate the lists of labels thus ordered for all the paths in the decomposition of T_1 and similarly concatenate the labels of T_2. The order of the paths does not matter. (Delete from the lists any label that does not appear in both trees.)

Now preprocess the two lists of labels for range queries as follows. The coordinates of label i are $[x, y]$ where i appears in location x of T_1's list and location y of T_2's list.

Using the method from [25] the grid can be preprocessed in $O(t \log t)$ time such that a grid-query can be computed in time $O(int + \sqrt{\log t})$. For every $v \in T_1$ and $w \in T_2$ the paths to the roots in T_1 and T_2 pass through no more than $\log t$ path segments in the decomposition of T_1 and T_2. Intersecting every segment in T_1 and every segment in T_2 gives a reply to the tree path intersection query. Since there are at most $\log^2 t$ such intersections to consider, the total time is $O(int + \log^{2.5} n)$. $\square$

The bottleneck of dictionary matching with one error is tree path intersection, therefore we have the following result.

Theorem 9.5 *Let $D = \{P_1, ..., P_s\}$ be a dictionary of size d, we can preprocess D in $O(d \log d)$ time such that a text-query $t_1 \cdots t_n$ can be answered in $O(n \log^{2.5} d + tocc)$, where tocc is the number of occurrences of matches with exactly one mismatch. We can also preprocess D in $O(d \log^2 d)$ time such that a query is answered in $O(n \log^3 d \log \log d + tocc)$, where tocc counts matches with at most one mismatch.*

Proof: The first part follows directly from Lemma 9.4 and the fact that the intersection of the paths is the bottleneck of the algorithm.

The second part of the theorem similarly follows from the path intersection bottleneck and using the same reasoning as in Theorem 6.1. $\square$

10 Discussion and Open Problems

We have seen a single unified approach for efficient deterministic worst-case solutions of both the dictionary and indexing problems with one error. Our ideas easily handle insertion and deletion errors as well as mismatches. In these aspects our idea is superior to the Yao and Yao data structure. However, like the Yao and Yao as well as the Brodal and Gąsieniec ideas, our method does not seem to be easily extendable to a higher (albeit small) number of errors.

References

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

[2] A. Amir and M. Farach. Adaptive dictionary matching. *Proc. 32nd IEEE FOCS*, pages 760–766, 1991.

[3] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[4] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[5] A. Amir, D. Keselman, G.M. Landau, N. Lewenstein, M. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. In *Proc. 1999 Workshop on Algorithms and Data Structures (WADS)*, pages 181–192, 1999.

[6] A. Apostolico and Z. Galil (editors). *Pattern Matching Algorithms.* Oxford University Press, 1997.

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

[8] G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, pages 65–74. LNCS 1075, Springer, 1996.

[9] 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.

[10] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[11] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[12] P. Ferragina and R. Grossi. Fast incremental text editing. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 1995.

[13] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM J. Computing*, 27(3):713–736, 1998.

[14] P. Ferragina, S. Muthukrishnan, and M. deBerg. Multi-method dispatching: a geometric approach with applications to string matching. In *Proc. 31st Annual Symposium on the Theory of Computing (STOC)*, pages 483–491, 1999.

[15] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[16] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 697–704, 1994.

[17] D. Gusfield, G.M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41:181–185, 1992.

[18] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[19] R.M. Idury and A.A Schäffer. Dynamic dictionary matching with failure functions. *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 273–284, 1992.

[20] 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.

[21] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. and Dev.*, pages 249–260, 1987.

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

[23] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

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

[25] M. H. Overmars. Efficient data structures for range searching on a grid. *J. of Algorithms*, 9:254–275, 1988.

[26] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. 37th FOCS*, pages 320–328, 1996.

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

[28] A. C.-C. Yao and F. F. Yao. Dictionary lookup with one error. *J. of Algorithms*, 25(1):194–202, 1997.