Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm

(extended abstract)

Süleyman Cenk Şahinalp
University of Maryland at College Park and Bell Laboratories, Murray Hill

Uzi Vishkin *
University of Maryland at College Park and Tel Aviv University, Israel

Abstract

A key approach in string processing algorithmics has been the labeling paradigm [KMR72], which is based on assigning labels to some of the substrings of a given string. If these labels are chosen consistently, they can enable fast comparisons of substrings. Until the first optimal parallel algorithm for suffix tree construction was given in [SV94], the labeling paradigm was considered not to be competitive with other approaches. In this paper we show that, this general method is also useful for several central problems in the area of string processing:

- *Approximate String Matching,*
- *Dynamic Dictionary Matching,*
- *Dynamic Text Indexing.*

The approximate string matching problem deals with finding all substrings of a text which match a pattern "approximately", i.e., with at most m differences. The differences can be in the form of inserted, deleted, or replaced characters.

The text indexing problem deals with finding all occurrences of a pattern in a text, after the text is preprocessed. In the dynamic text indexing problem, updates to the text in the form of insertions and deletions of substrings are permitted.

The dictionary matching problem deals with finding all occurrences of each pattern out of a set of patterns in a text, after the pattern set is preprocessed. In the dynamic dictionary matching problem, insertions and deletions of patterns to the pattern set are permitted.

*partially supported by NSF grant CCR-9416890

1. Introduction

As the size of electronically stored information grows rapidly, efficient methods for string processing are becoming critical. In this abstract and its full version [SV96], we extend our work on parallel construction of a suffix tree [SV94] in a nontrivial manner, to obtain efficient algorithms for three fundamentally important problems: (i) approximate string matching, (ii) dynamic dictionary matching, (iii) dynamic text indexing. We describe each of these problems, discuss the relevant literature, and summarize our contributions below.

Approximate String Matching This problem deals with finding all substrings of a text T that match a pattern P, with the exception of at most m differences, for some given integer m. The differences can be in the form of inserted, deleted or replaced characters. Several deterministic algorithms for solving this problem are given in [Se80], [LV85], [LV86], [GG88], [LV88], [LV89], [GP90], and numerous others. All of these have worst case running time of $\Omega(tm)$.

A probabilistic algorithm, given in [CL90] runs in $O(t)$ *expected time* if $m = O(p/\log p)$.

In this paper, we provide a deterministic algorithm for the approximate string matching problem, which runs in $O(poly(m)t\log p/p + t)$, hence achieving linear time for $poly(m) < O(p/\log p)$.

Dictionary Matching This problem deals with preprocessing a set of patterns $P(1), P(2), \ldots, P(n)$ of sizes $p(1), p(2), \ldots, p(n)$ respectively, to find all occurrences of each of the patterns once a text T is given . The first linear algorithm for this problem is given in [AC75]. This algorithm preprocesses the patterns in $O(d = p(1) + p(2) +, \ldots, +p(n))$ time, and runs in $O(t + tocc)$ time. Here $tocc$ is the total number of occurrences of all patterns in the text.

In *dynamic* dictionary matching, updates to the patterns occurring in the form of deletions and insertions of patterns are allowed. Algorithms for this problem have been provided in [AF91], [AFM92], [AFGGP94] and many others. Among known results, the best time is achieved in [AFILS93]. This algorithm preprocesses the patterns in $O(d)$ time, performs an update (i.e., insertion or deletion) of a whole pattern of size p in $O(p \log d / \log\log d)$ time, and runs in $O((t + tocc) \log d / \log\log d)$ time.

In this paper, we provide a deterministic algorithm for dynamic dictionary matching problem, which preprocesses the patterns in $O(d)$ time, performs search in $O(t + tocc)$ time, and updates in $O(p)$ time.

Text Indexing This problem deals with preprocessing a text T to find all occurrences of a given pattern P. The suffix tree data structure solves this problem in $O(p + tocc)$ time, where $tocc$ is the number of all occurrences of P in T. Algorithms for constructing suffix trees were first provided in in [KMR72], and then in [We73], and [Mc76]. The latter two algorithms build the suffix tree in $O(t)$ time. This running time is also achieved by the serial execution of three new parallel algorithms given in [SV94], [Ha94], [FM96]. Suffix trees have proven their use in several domains [Ko94], [Ja90]. Other data structures, including the suffix arrays [MM90], are considered not as efficient as the suffix trees.

In the *dynamic* text indexing problem, updates to the text in the form of insertions and deletions of substrings are permitted. Such updates can take $O(t)$ time in a suffix tree; hence suffix trees are not suitable for this problem. The border tree [GFB94] is the first alternative to the suffix tree in this domain. It is constructed in $O(t)$ time, and performs updates in terms of insertions and deletions of single characters in $O(\log t)$ time. It can perform searching in $O(p + tocc \log i + i \log p)$ time, where i is the number of updated characters. A recent result, given in [FG95], enables $O(p + tocc)$ time searching, and is constructed in $O(t)$ time. It performs updates in the form of deletion and insertion of a substring of size u in $O(\sqrt{t} + u)$ time.

In this paper, we provide a deterministic algorithm for dynamic text indexing problem, which preprocesses the text in $O(t)$ time, performs searching in $O(p + tocc)$ time, and performs updates of substrings in $O(\log^3 t + u)$ time.

Organization of the Paper We start our presentation by giving some preliminaries. Then, in section 2, we describe a new linear time string matching algorithm, which demonstrates some of the main ideas behind our approach. We proceed to give our algorithms for approximate string matching, dynamic dictionary matching and dynamic text indexing in Sections 3, 4 and 5 respectively. We provide more details on our algorithms and their properties in the appendices.

1.1. Some Preliminaries

All the strings we deal with consist of characters from a fixed alphabet A of size a. Any finite string of characters are denoted by capital letters like P, R, S, T. The size of a string R is denoted by r, the characters of R are denoted by $r_1, r_2, \ldots, r_r$, and its substrings $r_i, r_{i+1}, \ldots, r_{i+j}$ $(1 \leq i \leq j \leq r)$ are denoted by $R_{i,j}$. We define the *reverse* of the string R to be $r_r, r_{r-1}, \ldots, r_1$.

Unless otherwise stated, logarithms are base 2, and where non-integer values are referred to in integer context, their ceiling is taken.

Comment to the Reader For simplicity, most of our informal outlines of algorithmic ideas suppress the case of periodicity in *any substring* of the input. We provide the general versions of our algorithms which involve periodicities in the appendices and in [SV96].

2. A New String Matching Algorithm

In this section, we present a new linear time algorithm for the classical string matching problem. The algorithm runs in four stages.

In the first stage, we identify $O(t)$ possibly overlapping substrings of the text T which we call *cores*, in $O(t)$ time. This is done consistently, i.e., if among two identical substrings, we identify one of them as a core, then we identify the second one as a core as well. Each core is assigned a label. The labels are also chosen consistently, i.e., two cores get identical labels if and only if they are identical. We give the actual construction of the cores in Section 2.1.

In the second stage, we identify the cores of P consistently with the cores of T, i.e., if a substring of P is identical to a core of T, then it is identified as a core as well. To label the cores of P, we use the labels of cores of T. This stage takes $O(p)$ time.

The cores play a crucial part in all of our algorithms. Due to the following properties, the cores enable to compare substrings of T and/or P very efficiently. These properties are formally given in Section 2.2.

Compact Representation Property: Any substring of size l can be uniquely represented as a concatenation of (possibly overlapping) only $O(\log l)$ of its cores.

Consistent Representation Property: Identical substrings are composed of identical cores.

In the third stage, we obtain the *compact representation* of the pattern. We describe this stage in detail in Section 2.1. This stage takes $O(\log p)$ time.

In the fourth stage, we find which of the $O(t)$ size p substrings of T match P in $O(t)$ time. By using the compact representations, we first show how to eliminate the possibility of a match between P and all but $O(t/p)$ such substrings, and then show how to verify if each of the remaining substrings actually matches P in Section 2.2.

Complexity The total running time of all the stages is $O(t+p)$.

2.1. The Core System and the Compact Representation of a String

We show how to identify $O(s)$ cores of a string S. For this purpose we build a ($\log s$ level) *fat-tree* of degree $\log^* s$, as explained next.

A fat-tree is a data structure which is very similar to a rooted tree, with the following difference: a node of a fat tree can have more than one parent, there might be more than one root and some non-root node may have no parents at all.

Definition 1 (Fat-Tree) *A fat-tree of degree d, with h levels is a DAG in which: (1) each node of level-i ($1 \leq i \leq h$) has at most d child nodes, all at level-$(i-1)$; (2) each node of level-$(i-1)$ has at most $d/2$ parent nodes; (3) there is at least one node at level-h; (4) the nodes at level-h are called roots and they have no parent nodes. Figure 1 illustrates how a fat-tree looks like.*

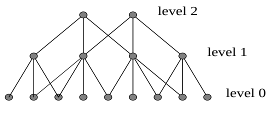

Figure 1. A fat-tree of degree 4, and its levels

We derive all the $O(s)$ cores of S in $O(\log s)$ iterations as follows. In iteration 0, we designate all single character substrings of S as *level-0 cores*. These cores constitute the leaves (level-0 nodes) of our fat-tree. The labels of level-0 cores are their respective characters themselves. In iteration i, we designate the cores of level-i as follows:

1. Consider the sequence of level-$(i-1)$ labels (of nodes). Using a variant of the "locally consistent parsing" (LCP) method of [SV94], which we describe in Appendix A, we get (possibly overlapping) subsequences of $O(\log^* s)$ labels each. Each label may appear in at most $d/2 = O(\log^* s)$ subsequences and the number of subsequences will not exceed $s/2^i$.

2. For each subsequence, we create a parent node of level-i. We connect this parent node to all the nodes of the subsequence. Each such node represents a core of level i. The core of a node is defined as the concatenation of the cores of its children.

3. We give consistent labels to all nodes of level-i and their respective cores, i.e., two nodes get identical labels if and only if their children are identical.

In the end of the last iteration, we obtain $O(1)$ cores of level-$\log s$. Among them, we call the leftmost one the *main core* of S.

The cores of S and their labels are illustrated in Figure 2.

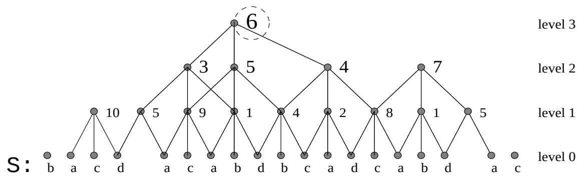

Figure 2. The cores of an example string $S = bacdacabdbcadcabdac$ and their labels. LCP procedure is not demonstrated. The main core is marked with a surrounding circle. We represent the labels of increasing levels with larger characters.

The *compact representation* of S is the sequence, from left to right, of the labels of the nodes of the fat-tree which do not have parents. The *compact representation* of a given substring R of S is defined as follows. The part of the fat-tree if S which pertains to R consists of all cores which fully fall with the boundaries of R. Removing all other nodes from the fat-tree of S will provide a reduced fat-tree for R which gives the compact representation of R.

Given the fat-tree of S, the computation of the compact representation of any substring R takes $O(\log r)$ time. We demonstrate how the compact representation of the string in Figure 2 is computed in Figure 3.

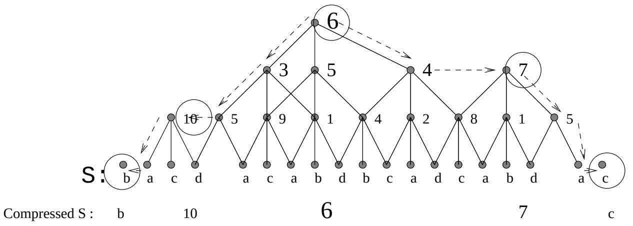

Figure 3. The compact representation of the whole S obtained via the traversal of the fat-tree of S. Notice that, among all the traversed nodes, only those whose labels are surrounded by a circle are of interest to us.

2.2. Properties of Cores and the Compact Representations of Strings

Once the cores of P and T are computed consistently, they satisfy the following important properties that we give without proofs. We leave the proofs of these properties to the full version of the paper [SV96].

Property 1 (Consistency of the Cores) *Suppose P is identical to a substring $T_{k,k+p}$. If $P_{i,j}$ is a core at P of level-g, then $T_{k+i-1,k+j-1}$ is also a core at level-g, and has the same label as $P_{i,j}$.*

Property 2 (Compaction of the Cores) *Any pattern P can be uniquely represented as a concatenation of $O(1)$ cores of level-0, $O(1)$ cores of level-1, ..., $O(1)$ cores of level-$\log l - 1$, $O(1)$ cores of level-$\log l$, $O(1)$ cores of level-$\log l - 1$, ..., $O(1)$ cores of level-1, ..., and $O(1)$ cores of level-0.*

These two properties are of fundamental importance to the correctness of our new string matching algorithm. Property 1 enables us to immediately eliminate matches between P and all but $O(t/p)$ of the size p substrings of T. It implies that we only need to consider these substrings as *candidates* for a match to P. Property 2 ensures that the compact representation (as computed in Section 2.1) of identical substrings of P or T are identical.

Below we give some more properties of cores which will be used in our approximate string matching algorithm of section 3.

The property below states that the overlap between two neighbor cores can not be too long.

Property 3 (Maximum Overlap of Cores) *Given a core C of level-$\log k$, any of the neighbor cores of C can include all but at least k of the characters of C.*

A corollary to property 3 states that the exact matches of a core of P in T, can not be too close.

Property 4 (Maximum Overlap of Matches) *Suppose that a core C of P of level-$\log k$ (for any $k < p$) matches exactly a substring D (which is necessarily a core) in T, then it can not exactly match any of the substrings of T whose starting (or ending) character is less than k apart from the starting (or ending) character of D.*

3. Approximate String Matching

Given a text T, and an integer m, our approximate string matching algorithm finds all occurrences of a pattern P with at most m differences. The substring $T_{i,j}$ is said to match P with at most m differences, if we can obtain P from $T_{i,j}$ by applying at most m changes in one of the following forms each:

Replacement: Replace a character of $T_{i,j}$ with another character.

Insertion: Insert a character in any location of $T_{i,j}$.

Deletion: Delete a character from any location of $T_{i,j}$.

A restricted version of the problem is the *string matching with mismatches*, in which only the differences be in the form of replacements are allowed.

The approximate string matching algorithm is a generalization of the exact string matching algorithm we described above. The generalization is possible because of the following observations.

Suppose $T_{i,j}$ match P with m mismatches. Then:

(1) We can write P as $P = S(1), p(1), S(2), p(2), \ldots, p(m), S(m+1)$, and $T_{i,j}$ as $T_{i,j} = S(1), t(1), S(2), t(2), \ldots, t(m), S(m+1)$ where each $S(l)$ is a possibly empty substring (of both P and T), and $p(l)$ and $t(l)$ are single characters. If $T_{i,j}$ matches P with m differences, we can still parse P and $T_{i,j}$ similarly, with one difference: for any given l, at most one of $p(l)$ or $t(l)$ might be a NULL character.

(2) There is at least one substring $S(l)$ whose size is at least $p/m - 1$.

(3) Consider all substrings $S(l)$ whose size is at least $p/m - 1$. Each of these substrings must have a main core of level-$O(\log p/m)$ or higher. Hence, among all level-$\log(p/m)$ cores of P, there should be at least one which match the corresponding core in $T_{i,j}$.

(4) The compact representations of each of the $S(i)$ $(1 \geq i \geq (m+1))$ should be identical in both P and $T_{i,j}$.

Following these observations, we obtain an algorithm for string matching with mismatches below. We then give the description of the algorithm for string matching with differences in section 3.2.

3.1. Algorithm for String Matching with m Mismatches:

Observation (1) states that if P matches $T_{i,j}$ with m mismatches, P and $T_{i,j}$ should have at least one matching pair of cores of level-log(p/m). Hence, in step 1, we use each level-log p/m core in P as an "anchor" core (i.e., a core which should have an exact match), and we find all level-log p/m cores in T that match it. There are at most m such cores in P and at most tm/p in T which need to be compared for a total of tm^2/p comparisons. This step eliminates all but at most tm^2/p candidate substrings that could match P.

Now, we only need to check if each of these candidates $T_{i,j}$, matches P (with at most m mismatches). This can be done in $O(p)$ time, by comparing the characters of P and $T_{i,j}$. Since this gives an $O(tm^2)$ time, we show how to finish this task faster by utilizing observation (4) below.

Starting from each initial match found in step 1, we attempt to extend to the left and to the right with the smallest possible number of mismatches. In general, we first try to match the core to the immediate left of the initial match. In case of a success we continue to the next core to the left. In case of a failure we may need to climb down the fat tree till we either find a match or reach the leaf level and find a single character of mismatch. Following a mismatch we may either find additional mismatches or start climbing up the fat tree to find growing size matches. A similar procedure is used for extending the initial match to the right.

Overall it is not difficult to see that each of the climb up or climb down operation in the fat tree can be charged to a mismatch. No more than $2 \log p/m$ operations could be charged to a mismatch and since we can abort after finding $m+1$ mismatches, these charges would total $O(m \log p/m)$ operations. In addition, there will be at most m matches at level $\log p/m$, hence the total number of operations per each of the tm^2/p initial matches is $O(m \log p/m)$ and the overall time complexity is $O(t/p \times m^3 \log p/m)$.

3.2. Algorithm for Approximate String Matching

Our approximate string matching algorithm is based on the algorithm for string matching with m mismatches; however the techniques we use are much more involved and we give the algorithm without all its details.

First it is important to understand why the approximate string matching is more involved than string matching with m mismatches: Intuitively, if a substring $T_{i,j}$ matches P with m mismatches, this, for "most cases", implies that P does not match $T_{i+1,j+1}, T_{i+2,j+2}$ and so on. Hence, the candidate determination is much easier in this problem.

In the approximate string matching problem, if $T_{i,j}$ matches P with, say, $l < m$ differences, then obviously the strings $T_{i-1,j}, T_{i+1,j}$ and so on would match P.

This difficulty could be overcome by the following property, whose proof is provided in [SV96]. This property states that, if in a "neighborhood" of T, a long substring R matches a substring S of P exactly, then there exists a match of P in the neighborhood, in which R is matched to S in full that gives the smallest number of differences for any match in the neighborhood.

Property 5 (Best Match) *Let S be a substring of P, and R be a substring of T which exactly matches S. Let S (and hence R) have at least m level $-\log m$ cores. Consider all matches of P in T in which S is fully matched with R. Among these matches, let $T_{i,j}$ be the one with smallest number ($= n$) of differences. Then, for any k in the range $i-m, \ldots, i+k$, and for any l in the range $j-m, \ldots, j+m$, the match between $T_{k,l}$ and P should have at least n differences, provided that $n \leq m$.*

This property provides the following intuition. If P and T have exactly matching substrings which are long enough, then in their immediate neighborhood (i.e. small shifts of at most m characters to left or right) a match with the smallest number of differences can be obtained by taking their largest possible exact match and extending it to a full match with differences.

In our algorithm, the anchor cores as described in the previous section will provide such initial long exact matches. Extending them to left and right will be done as follows. We will find additional nearest long enough exact matches by using the fat trees of P and T. We then will use the standard dynamic programing algorithms for approximate matching (e.g. [LV89]) to connect with them using the smallest possible number of differences.

A key observation is that if no additional large exactly matching substrings exist and a big fraction of the pattern is not yet matched (possibly with few differences) then for small enough m, a match with m differences is impossible.

Now we are ready to describe our string matching with m differences algorithm. Our algorithm starts with the computation of the fat-trees and hence the cores of T and P. This is illustrated in Figure 4.

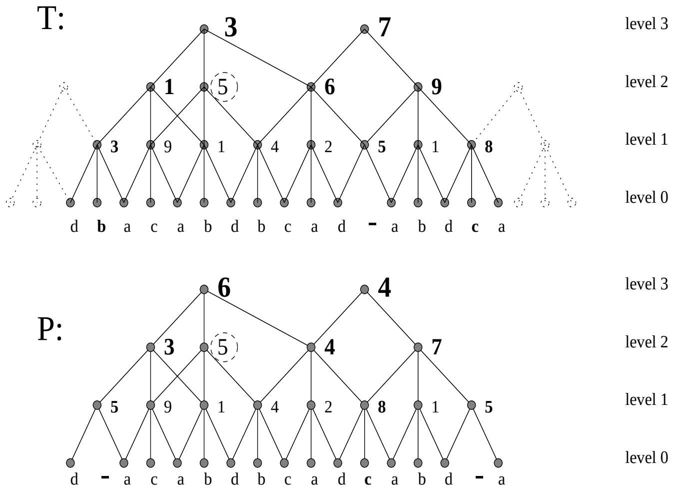

Figure 4. The fat-trees of P and the matching substring of T. The differences in each level are denoted with bold letters. Note that there are at most 3 differences in the labels of any level. The highest level core label which is identical in both P and T is marked with a circle

.

Our algorithm has two major stages. In stage-1, similar to the string matching with m mismatches algorithm, we use each level-$\log m$ core of P as the anchor core, and try to find all exact matches of each such core in T. In stage-2, we take each such match, and try to extend the match towards both right and left of the anchor core. This is done very similar to the climbing up and climbing down processes of the string matching with m mismatches algorithm, with the following difference: After applying the climb down process towards the right (or left) of the anchor core, and finding the first character which does not have a match, we can not immediately proceed with the climb up process (unlike in the mismatches algorithm).

Rather, we do the following. We consider the suffix of P, which starts at the end of the first character which does not have a match. In this suffix, we search for the first core of level-$\log(m^2)$, which exactly matches its corresponding core (there exists only once such core) in T. If there exists no such core among the leftmost (rightmost) m cores of level-$\log(m^2)$ in this suffix, then we terminate search. If, however, there exists such a core C, we apply the standard dynamic programming techniques to find the best match of substrings lying between the anchor and C. Following this, we take C as the next anchor and further try to extend the match towards right (left) of C by subsequent applications of a climb up, followed by a climb down procedure until we reach the right (left) end of P.

The climb up and climb down procedures, each of which is followed by the application of the dynamic programming procedure can each be charged for a difference in the match. Each climb up and climb down procedure takes $O(\log p)$ time and each approximate string matching procedure for a substring of size $O(m^3)$ takes $O(m^4)$ time. The total time for all climb up and climb down procedures is $O(m \log p)$. It is a simple exercise to reduce the total number of operations for the application of dynamic programming from $O(m^5)$ to $O(m^4)$ for each match of each of the cores of P chosen as an anchor. As the total number of matches between all level-$\log p/m$ cores of P and T are $O(m^2)$, the algorithm runs in $O(t/p(m^3 \log p + m^6))$. Again by using property 5, this can be reduced to $O(t/p(m^2 \log p + m^5))$.

4. Dynamic Dictionary Matching Algorithm

The *input* of our dictionary matching algorithm is a set of patterns $P(1), \ldots, P(n)$. The *output* of our algorithm is the data structure D, which supports the following operations:

Searching: Given a text T, find all occurrences of each of $P(1), \ldots, P(n)$ in T.

Insertion: Given a new pattern $R = R_1, \ldots, R_r$, insert R into D.

Deletion: Given a pattern $R = P(i)$ in D, delete $P(i)$ from D.

The simple string matching algorithm we gave above provides the basic intuition behind the dictionary matching algorithm. We know that if any of the patterns match a substring of the text, then the main core of the pattern and the substring should necessarily be aligned.

To exploit this property, we first classify the patterns into groups according to the level of their main core. We build an independent data structure for each group.

Some Definitions For each pattern $P(i)$ $(1 \leq i \leq n)$, define its *main prefix* to be the prefix of $P(i)$ pattern which ends where its main core ends. Similarly define its *main suffix* to be the suffix which begins where its main core begins.

4.1. A Restricted Version of The Data Structure

For obtaining a restricted version of our data structure, we *assume* that no main suffix of a pattern is a prefix of the main suffix of another pattern. Similarly we assume that no main prefix of any pattern is a suffix of the main prefix of another pattern. We also assume that the fat-tree of the patterns and the input text have been constructed. We generalize this algorithm for any set of patterns in [SV96].

Definition 2 (Compact Trie) *The "compact trie" of a set of strings, $P(1), P(2), \ldots, P(i)$ is the tree in which:*
each edge is labeled by a substring,
each string $P(j)$ is represented as a leaf $L(j)$ such that the concatenation of edge labels from the root to $L(j)$ gives the string $P(j)$,
no prefixes of two sibling edges are identical.

See Figure 5 for an illustration of a compact trie.

Description of the Data Structure Consider the group of patterns whose main core is at level-l. Our data structure for this group consists of two "compact trie"s. The first compact trie, denoted by *Suf(l)*, is the compact trie of the compact representations of the main suffixes of the patterns in this group. The second compact trie, denoted by *Pre(l)*, is the compact trie of the compact representations of the reverses of the main prefixes of the patterns in this group. The construction of a compact trie takes $O(p(1) + \ldots + p(i))$ time [AFM92].

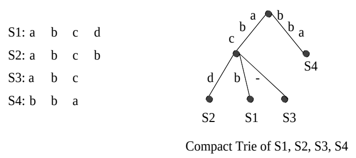

Figure 5. The compact trie of a set of strings.

Searching For each level of the fat-tree, for each core in that level:
(1) Match it to the compact suffix trie of that level. This will result in a few match of suffixes of the pattern.
(2) Finish by checking whether those matches extend to the left using the compact prefix trie. The restricted version of the problem, this can be done efficiently with no additional effort.

Insertion We first compute the compact representation pattern. Inserting the compact version of the pattern to any of the tries in $O(r)$ is straightforward.

Deletion One simply removes the leaves representing the pattern in the tries. This takes only $O(1)$ time. One also needs to update the core tables, which can be done in $O(r)$ time.

5. Dynamic Text Indexing

The *input* of our text indexing algorithm is the text string T. The *output* of our algorithm is the $DS(T)$ data structure, which supports the following operations:
Searching: Given a pattern P, find all occurrences of P in T.
Insertion: Given a string $S = s_1, s_2, \ldots, s_s$, and a location i in T, insert S in T between its $i - 1^{th}$ and i^{th} characters, to obtain an updated text $T' = T_{1,i-1}, S, T_{i,t}$. Update the data structure $DS(T)$ to obtain $DS(T')$.
Deletion: Given a substring of T, $T_{i,j}$, delete $T_{i,j}$ from T to obtain $T'' = T_{1,i-1}, T_{j+1,t}$. Update $DS(T)$ to obtain $DS(T'')$.

Our dynamic text indexing data structure extends the ideas we used for the basic pattern matching algorithm and the dynamic dictionary data structure. However, to achieve the desired complexity, we employ more involved techniques.

As a first step we again compute the fat-tree of the cores of T. Then we build two separate data structures for searching patterns of size smaller than $\log_a \sqrt{t}$, (where a is the size of the alphabet A) and larger than $\log_a \sqrt{t}$. We employ a standard lookup table for patterns whose size is smaller than $\log_a \sqrt{t}$, and use a trie based approach, as used in the dynamic dictionary matching algorithm, for patterns whose size is larger than $\log_a \sqrt{t}$. Please see [SV96] for a description of how these data structures are built and how the search and update operations are performed.

Acknowledgements We would like to thank A.Funda Ergun (Cornell U. and MIT), Martin Farach (Rutgers U.), and Yossi Matias (Bell Laboratories), for their suggestions and help.

References

[AC75] A. Aho and M. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Communications of the ACM (CACM)*, 1975

[AF91] A. Amir and M. Farach, Adaptive Dictionary Matching, *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991

[AFGGP94] A. Amir, M. Farach, Z. Galil, R. Giancarlo, K. Park, Dynamic Dictionary Matching, *Journal of Computer and System Sciences (JCSS)*, 1994

[AFILS93] A. Amir, M. Farach, R. Idury, A. La Poutre and A. Schaffer, Improved Dynamic Dictionary Matching, *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993

[AFM92] A. Amir, M. Farach and Y. Matias, Efficient Randomized Dictionary Matching Algorithms, *Symposium on Combinatorial Pattern Matching (CPM)*, 1992

[AILSV88] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber and U. Vishkin, Parallel Construction of a Suffix Tree with Applications, *Algorithmica*, 1988

[BM77] R. Boyer and J. Moore, A Fast String Searching Algorithm, *Communications of the ACM (CACM)*, 1977

[CL90] W. Chang and E. Lawler, Approximate String Matching in Sublinear Expected Time, *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1990

[CV86] R. Cole and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades, Micro and Macro Techniques for Designing Parallel Algorithms, *ACM Symposium on Theory of Computing (STOC)*, 1986

[FG95] P. Ferragina and R. Grossi, Optimal On-Line Search and Sublinear Time Update in String Matching, *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1995

[FM96] M. Farach and M. Muthukrishnan, An optimal logarithmic time, randomized parallel string matching algorithm, *International Colloquium on Automata, Languages and Programming (ICALP)*, 1996

[GG88] Z. Galil and R. Giancarlo, Data Structures and Algorithms for Approximate String Matching, *Journal of Complexity (JComp)*, 1988

[GP90] Z. Galil and K. Park, An Improved Algorithm for Approximate String Matching, *SIAM Journal on Computing (SIAMJC)*, 1990

[GFB94] M. Gu, M. Farach and R. Beigel, An Efficient Algorithm for Dynamic Text Indexing, *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1994

[Ha94] R. Hariharan, Optimal Parallel Suffix Tree Construction, *ACM Symposium on Theory of Computing (STOC)*, 1994

[Ja90] J. Ja'Ja', Introduction to Parallel Algorithms, *Addison-Wesley*, 1990

[KMR72] R. Karp, R. Miller and A. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, *ACM Symposium on Theory of Computing (STOC)*, 1972

[KMP77] D. Knuth, J. Morris and V. Pratt, Fast Pattern Matching in Strings, *SIAM Journal on Computing (SIAMJC)*, 1977

[Ko94] S. Kosaraju, Real Time Pattern Matching and Quasi Real Time Construction of Suffix Trees, *ACM Symposium on Theory of Computing (STOC)*, 1994

[LV85] G. Landau and U. Vishkin, Efficient String Matching in the Presence of Errors, *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1985

[LV86] G. Landau and U. Vishkin, Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm, *ACM Symposium on Theory of Computing (STOC)*, 1986

[LV88] G. Landau and U. Vishkin, Fast String Matching with k Differences, *Journal of Computer and System Sciences (JCSS)*, 1988

[LV89] G. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms (JAlg)*, 1989

[MM90] U. Manber and G. Myers, Suffix Arrays: A New Method for On-Line String Searches, *SIAM Journal on Computing (SIAMJC)*, 1993

[Mc76] E. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *Journal of the ACM (JACM)*, 1976

[Me83] N. Meggido, Applying Parallel Computation Algorithms in the Design of Serial Algorithms *Journal of the ACM (JACM)*, 1983

[SV94] S. Sahinalp and U. Vishkin, Symmetry Breaking for Suffix Tree Construction, *ACM Symposium on Theory of Computing (STOC)*, 1994

[SV96] S. Sahinalp and U. Vishkin, Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm, *(Technical Report)*, http://www.umiacs.umd.edu/ jenk/Research/ds.tr.ps

[Se80] P. Sellers, The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms (JAlg)*, 1980

[We73] P. Weiner, Linear Pattern Matching Algorithms, *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1973

A. Description of LCP

We give the description of the locally consistent parsing procedure (LCP) [SV94] below.

Given a string of level-i labels in the Fat Tree, LCP determines the nodes and labels of level-$(i+1)$. More concretely, LCP derives "blocks" (or intervals) of size $O(\log^* l)$, where l is an upper bound on the number of possible labels. These blocks define nodes. The labels are set so that two blocks which consist of identical level-i labels get the same level-$(i+1)$ label. As a first step, LCP handles each label in the string LCP whose neighbors are not identical to it.

For every such label x, LCP determines if it is the last label of a block in $O(\log^* l)$ iterations, as follows.
(1) For each label in the string, LCP computes its tag. The tag of a label is the index of the least significant bit in which the binary representations of the label and its left neighbor differ.
(2) Then, each label is replaced by a new number which is obtained by the concatenation of two binary numbers:
(i) the tag of the label and
(ii) the value of the bit of the label whose index is the tag.
As the initial range of labels are $0, \ldots, l$, they will be replaced by numbers selected from $0, \ldots, 2 \log l$.
(3) LCP applies the same procedure for $\log^* l - 1$ iterations. In the first iteration the range of labels would be reduced from $0, \ldots, l-1$ to $2 \log l - 1$, in the second iteration, to $0, \ldots, 2 \log \log l - 1 \ +2$, and in the last iteration to $0, \ldots, 5$.
(4) LCP sets a label x as the end of a block, if the left neighbor of x is a local minimum; i.e., x is less than both its right (which is x itself) and left neighbors.

Notice that there can be at most $O(1)$ $level - i$ labels between two neighbor level-$(i+1)$ block ends. This is since as a result of the $\log^* l$ iterations the range for the resulting labels is 0...5.

Once a symbol x is identified as the end of a block, the block is set to include the end label and its preceeding $\log^* l$ labels.

As a second step, LCP handles substrings which consist of a single repeating label, as follows:

(1) LCP finds the beginning and end of each substring that consists of a single label.

(2) Starting from the first label in the substring, LCP divides the substring into blocks of size 2. If the substring is of odd size, then LCP sets the last block to be of size three. Notice that there are no overlaps between the blocks obtained by this procedure.

One obvious thing to note is that during the labeling of these blocks, the ones whose size is two get different labels than those whose size is three. In subsequent iterations, the labels would actually indicate whether the number of labels that a block represents is two or three.