26. R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Comput.*, 14:862–874, 1985.

This article was processed using the LaTeX macro package with LLNCS style

2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.
3. A. Amir and M. Farach. Adaptive dictionary matching. *FOCS '91*, 1991.
4. A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. Manuscript, 1991.
5. A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree. *Algorithmica*, 3:347–365, 1988.
6. H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *SPAA '91*, pages 50–61, July 1991.
7. R. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20:762–772, 1977.
8. M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *ICALP '92*, July 1992.
9. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *FOCS '88*, pages 524–531, Oct. 1988. Also, Revised Version: Tech. Report, University of Paderborn, FB 17 Mathematik/Informatik, 1991.
10. M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *SPAA '89*, pages 360–368, 1989.
11. M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hahshing in real time. In *ICALP '90*, pages 6–19, 1990.
12. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.
13. Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. of Complexity*, 4:33–72, 1988.
14. J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *SODA '91*, pages 271–280, Jan. 1991.
15. J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS '91*, pages 698–710, Oct. 1991.
16. R. Idury and A. Schäffer. Dynamic dictionary matching with failure functions. In *Combinatorial Pattern Matching*, 1992.
17. C. Jordan. Sur le assemblages des lignes. *J. Reine und Ang. Math.*, 70:185–190, 1869.
18. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31:249–260, 1987.
19. D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.
20. Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *STOC '91*, pages 307–316, 1991. Also in UMIACS-TR-91-65, Inst. for Advanced Computer Studies, Univ. of Maryland, April 1991.
21. Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *J. of Alg.*, 12(4):573–606, 1991.
22. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 28:852–865, 1983.
23. K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin Heidelberg, 1984.
24. M. Naor. String matching with preprocessing of text and pattern. In *ICALP '91*, pages 739–750, 1991.
25. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

must resort to indirect methods. In [23], Mehlhorn described a scheme for introducing some dynamization into static data structures, at the expense of amortization and a logarithmic slowdown. Full dynamization is not possible by simply using Mehlhorn's scheme. Only pattern insertion is allows. We discuss pattern deletions separately below. We do not describe the details of Mehlhorn's scheme in this abstract. We only note that we can achieve the following complexities by using the above static algorithm, that is, both the suffix tree construction and the trie balancing, as a "black box."

Inserting P**:** $O(\log p \log^* p)$ time, $O(p \log p \log k)$ work.

Text Scan: $O(\log d + \log t)$ time, $O(t \log d \log k)$ work.

Allowing for pattern deletions To make the algorithm fully dynamic, that is, to allow dictionary patterns to be deleted, we must take into account pattern deletions. As noted above, Mehlhorn's dynamization scheme does not allow for efficient deletions. The problem of pattern deletions can be neatly solved by an additional data structure as follows.

Let m_j be the marked node associated with pattern P_j. Notice that the suffix tree induces a forest partial order amongst the marked nodes as follows: $m_j \leq m_k$ if m_j is an ancestor of m_k. Let $\mathcal{P}(\Updownarrow_j) = \Updownarrow_i$ if m_j is the parent of m_i under this forest partial order. If m_i has no parent, the we set $\mathcal{P}(\Updownarrow_j) = \Lambda$, the root. If we start with each m_j disconnected from the other m_ks and we `join` m_j to $P(m_j)$ whenever m_j is deleted, then finding the nearest marked ancestor for a node v in the suffix tree becomes a two part process. First, the static pointer is followed to get v's nearest marked ancestor m_i. However, since P_i may have been deleted, we check the supplemental data structure for the tree in which m_i currently finds itself. The root of this tree will be the appropriate marked ancestor for n.

Implementation The above additional data structure can be viewed as implementing a UNION/FIND on the set of m_j in the suffix tree. As noted in section 4, many elegant and efficient amortized time algorithms exist for UNION/FIND (see e.g. [25]). However, since it takes $O(\log d)$ to traverse the suffix tree, we can use an $O(\log d)$ worst case time algorithm for UNION/FIND (see [2]).

Comment The general scheme of using the general dynamization paradigm of Mehlhorn for insertions and the union/find operations for deletions, can be used to adapt the Aho-Corasick (static) algorithm into a dynamic one; the preprocessing and updates then become $O(p \log k)$, and the text scan takes $O(t \log k)$.

Overall time bounds

Inserting P**:** $O(\log p \log^* p)$ time, $O(p \log p \log k)$ work.

Removing P**:** $O(\log d)$ sequential time initially for the `union` operation followed by an amortized $O(\log d \log^* d)$ time,$O(p \log d)$ work to "clean up" the remnants of the patterns which were not actually deleted from the Mehlhorn dynamic data structure.

Text Scan: $O(\log d + \log t)$ time, $O(t \log d \log k)$ work.

References

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

trie of the dictionary patterns as a sub-tree, with possibly some paths compressed. Hence, we can use the suffix tree in the same way that we use compressed tries in Sections 3 and 4 for dictionary matching.

Apostolico et al [5] showed how to build a suffix tree for a string of length n in $O(n \log n)$ operations, $O(\log n)$ time, and $n^{1+\epsilon}$ space, for any constant $\epsilon > 0$ (see also [13]). Using a parallel hashing algorithm (Section 2.2), the algorithm can be implemented in linear space; the time then increases by a factor of $O(\log^* n)$ with high probability, but there is no change in the number of operations.

Building the balanced trie In this paper, we will be satisfied with construction which takes $O(\log n)$ time, using n processors for an n node tree T, since we will have other procedure with this complexity. For this performance a simple implementation can be described. We use the Euler-tour technique of Tarjan and Vishkin [26] to compute for each node the size of the subtree of T rooted at this node; this step takes $O(\log t)$ time and $O(t)$ operations. The idea in this technique is to hold a linked list along an Euler circuit that "surrounds" the tree T, starting at the root. The circuit is defined over a graph defined by replacing each edge in T with two anti-symmetric edges. Then, by computing the rank of each edge in the list, a node v can easily find out the size of its subtree by computing the difference in rank between the edge $p(v) \rightarrow v$ and $v \rightarrow p(v)$, where $p(v)$ is the parent of v in T. Now a separator v can be recognized in constant time and t processors as follows: each of its neighboring edges notify it that the subtree rooted at each of its children is of size at most $\frac{2}{3}n$ and that the subtree rooted at v is of size at least $\frac{1}{3}n$ (equivalently, each connected component induced by removing v has at most $\frac{2}{3}n$ nodes). To construct a balanced trie, we employ the procedure above $O(\log n)$ times. The main observation is that after a separator is removed from a tree, the list of each new sub-tree can be updated in constant time.

Traversing the trie Since the sequential time is $O(\log d)$, we simply assign one processor to each text location and traverse the trie sequentially for each location.

Finding the nearest marked ancestor The final part, that of assigning a pointer from each node to its nearest marked neighbor, can be accomplished in $O(\log d)$ time with d processors using pointer doubling. The overall bounds for preprocessing are $O(\log d \log^* d)$ time and $O(d \log d)$ work.

Scanning the text The text scanning phase has two parts. Scanning through the trie, for each text location, takes $O(\log d)$ sequential time. We can therefore accomplish this phase in $O(\log d)$ with n processors. The second part, that of finding the marked ancestor, takes constant sequential time. The overall bounds are therefore $O(\log d)$ time with n processors.

Overall time bounds For preprocessing, we have time bounds of $O(\log d \log^* d)$ time, $O(d \log d)$ work. To scan the text, it takes $O(\log d)$ time and $O(n \log d)$ work.

5.2 Dynamic Matching

Parallel Trie Update Unlike the sequential suffix tree construction, the parallel construction is not amenable to insertions or deletions of patterns. Therefore, we

per update (since this will meet the complexity bounds of other procedures in our algorithms). Indeed this is done as follows.

We start with a weakly balanced trie that is also a balanced trie. Each node keeps track on the size of its subtree; thus each inserted node updates all nodes along its path from the root in the trie. When a sub-tree becomes too large, let T' be the largest sub-tree that becomes unbalanced. We construct a new complete separator decomposition for T'. While this construction costs $O(|T'|)$ work, we note that T' can become unbalanced only after $O(|T'|)$ insertions (due to the relaxation in the definition of the WBT). Since each node belongs to $O(\log n)$ different sub-trees in the WBT, the update cost of each insertion is $O(\log n)$, as desired.

Finding Marked Ancestor Consider an Euler-tour for the trie CT_D. In such a tour, each edge appears twice, and between those two occurrences are the edges of the subtree rooted at that edge. Similarly, the edges at the subtree rooted at a node v occur between the edges $(parent(v), v)$ and $(v, parent(v))$. Now consider the effect of marking a node v in the tree. As noted above, the subtree rooted at v form a consecutive subsequence in the Euler-tour. It is relatively straightforward to cast the operation of marking nodes, unmarking nodes, and finding the nearest marked ancestor as the sequence operations of `double split`, `union`, and `find`. The analogy is clear since marking a node v means that the descendant edges of v can be `split` away from the rest of the tree. Now finding the nearest marked ancestor involves a `find` operation to find which set the edge currently finds itself in. Finally, unmarking a node reverses the effect of a `double split`, therefore it is implemented by a `join`.

The implementation of such a scheme is straightforward since the induced an ordering on the edges which can be used to maintain a 2-3 tree (see e.g. [2]). 2-3 trees on n nodes support such operations as `split`, `find`, `min`, and `concatenate` in $O(\log n)$ time. The details of the use of `split`, `min`, and `concatenate` to implement the `double split`, `find` and `join` operations on the Euler-tour, are left to the final paper.

Inserting or Deleting P: $O(p \log d)$
Scanning T: $O(t \log d)$

5 Parallel Algorithms

The parallel algorithms described below are based on the algorithms of Section 4. Both the static and dynamic algorithms had the following four components. In the preprocessing we build a trie for the dictionary. We then construct a balanced trie from it. During the text scanning phase, we must traverse the trie and then find the nearest marked ancestor.

5.1 Parallel Static Dictionary Matching

Parallel Trie Construction A suffix tree of a given string is a compressed trie of a dictionary which consists of all the suffixes of the string. Thus, if we take a string to be the concatenation of the dictionary patterns, its suffix tree will contain the

As with the trie searching procedures described in Section 3, the output of the Balanced Trie search is, for each text position, the longest pattern prefix that matches at that text location. In a straightforward traversal of a trie, this is sufficient to determine which patterns match at that text location, since we traverse all nodes representing prefixes of the longest match. As we descend through the tree, we can check in constant time if there is an outgoing edge labeled with a \$. We know that any such edge represents a matching pattern.

However, in a balanced trie, we may jump over such nodes if we traverse the distinguished edge of some node. We can reduce the problem of finding all prefixes of a pattern which are themselves patterns to a tree problem as follows. We call any node in the trie CT_D a *marked* node if it has an outgoing edge labeled with a \$. The nodes in BT_D are clearly in a one-to-one correspondence with the nodes of CT_D. Therefore, if s is the longest pattern prefix that matches at text location i, we can find the matching patterns at i by finding all the marked ancestors of the node representing i in CT_D.

Finding Marked Ancestors The algorithm for finding the nearest marked ancestor given an unchanging trie is quite straightforward. By simply traversing the trie in depth first search order, we keep a stack of marked nodes, pushing each marked node as we encounter it and popping it as we exit that node for the last time, i.e. when we finish processing the subtree rooted at that node. For each node, we associate a pointer which we set to be the top node on the marked node stack. Clearly, each node will end up with a pointer to the nearest marked ancestor.

The total time for static dictionary matching is then:

Preprocessing: $O(d)$

Text Scanning: $O(t \log k)$.

4.3 Dynamic Dictionary Matching

The dynamic algorithm is a modification of the static case. Rather than using a balanced trie based on the separator decomposition tree of the dictionary trie, we use a dynamic balanced trie based on the balanced decomposition tree described below. In addition, we can no longer afford to maintain back pointers from each node to their nearest marked ancestor, since new marked nodes may be introduced into the dictionary trie at any time. Instead, we maintain a secondary data structure which allows us to update insertion and deletion information directly, a variation of which appears in [3].

Dynamic Balanced Tries We would like to maintain the balanced trie data structure, when inserting new nodes. To enable efficient computation, we replace this data structure with an alternative one which has more relaxed requirements. A node v in T is called a *pseudo-separator* if each of the subtrees induced by removing v from T is of size at most $\frac{3}{4}n$. We call a balanced trie in which pseudo-separators replace separators a *weakly balanced trie (WBT)*. It is easy to see that the height of a WBT is $O(\log n)$ as well. Such weakly balanced tries can therefore replace balanced tries in our applications. Our objective is to support insertions with an $O(\log n)$ cost

Definition and construction: In a rooted tree T, $|T| = n$, a node v is called a *separator* if each of the connected components induced by removing v from T is of size at most $\frac{2}{3}n$. It is well known that every tree contains a separator. A *complete separator decomposition tree* $SD(T)$ of T is defined as follows. $SD(T)$ is a rooted tree whose root is a separator v of T. By removing the node v from T we get rooted trees whose roots are the children of v in T as well as the subtree rooted at the root of T (assuming that the root of T is not a separator of T). The recursively defined separators of these rooted subtrees are the children of v in the complete separator decomposition tree $SD(T)$.

It is easy to see that the height of the separator decomposition tree $SD(T)$ is $O(\log n)$. It is known that a complete separator decomposition tree can be constructed in linear time [24].

4.2 Algorithm

We are now ready for the dictionary matching algorithm based on balanced tries.

Suppose that dictionary $D = \{P_1, \ldots, P_k\}$ has compressed trie CT_D and that tree CT_D has separator decomposition $SD(CT_D)$. We will construct a *balanced trie* BT_D as follows. Let $ST(v)$ be the subtree rooted at *node*. Let $ST(v) - ST(v')$ be the subtree rooted at v with the subtree rooted at v' deleted, if v' is a descendant of v, and $ST(v) - ST(v') = ST(v)$ otherwise.

To define BT_D, we take CT_D and perform the following operations. Let v be a separator of CT_D and let $s(v)$ be the concatenation of the labels from the root to v. Let $c_1, c_2, \ldots, c_q$ be the children of the root and let $l(c_i)$ be the label on edge $(root, c_i)$. Then the root of BT_D will have $q+1$ children $v', c'_1, \ldots, c'_q$ where $l(v') = s(v)$ and $l(c'_i) = l(c_i)$. Furthermore, the subtree rooted at v' will be the recursively defined balanced trie of $ST(v)$ and the subtrees rooted at each c'_i will be the balanced trie of $ST(c_i) - ST(v)$.

Now the algorithm proceeds as in the case for the compressed trie with one modification. We have a *distinguished* edge which points to the separator u' of the subtree rooted at the current node u. Similarly, we call a child v *distinguished* if the edge between v and $parent(v)$ is the distinguished edge of $parent(v)$. The distinguished edge is labeled by the fingerprint of the string associated with the path between u and u' in the trie. We test this edge first. Whether this edge matches or not, this test eliminates a constant fraction of the remaining trie, thus making the worst case search time logarithmic in k.

To summarize, once a balanced trie has been built for a dictionary, we set $cur \leftarrow root$ and repeat the following:

1. Let v be the distinguished child of cur.
2. If $l(v)$ = next characters in text, that is, their fingerprints match, then
   - set $cur \leftarrow v$
   - advance the pointer in text by $|l(v)|$

   else
   - check which of the remaining edges starts with the next text character.
   - Proceed down that edge if possible.

hash table. Then query takes constant time and adding a new edge is implemented by inserting a new element into the hash table (see § 2.2). Therefore the updated and traversal operations can be performed in linear time.

3.2 Compressed Tries

It will sometimes be the case that many nodes in a trie will have out degree one. In such a case, we can use a compressed trie to significantly reduce the space requirements for storing the trie.

In a compressed trie, edges are labeled with substrings of the input patterns rather than simply with single characters. Whenever we have a chain of degree one nodes, we compress the chain into a single edge labeled with the substring obtained by concatenating the deleted edge labels. Clearly the new label will be a substring of some input pattern and so it can be represented by a pair which consists of a pointer into that input pattern and a length. Now every internal node has degree at least two. The size of such a trie on k patterns is therefore $O(k)$ rather than $O(d)$ for the uncompressed trie.

Furthermore, a compressed trie, in conjunction with fingerprints (see § 2.1) can be used to speed up the search through the trie. Let CT_D be the compressed trie for some set D of strings. Let T_D be the corresponding uncompressed trie. If $depth(CT_D) \ll depth(T_D)$ then we can apply the following algorithm.

Proceed as before independently from each text location to search through CT_D. When an edge is labeled with a character, simply compare the character and proceed appropriately. If an edge is labeled with a pattern substring, then compare the fingerprint of that substring with the appropriate fingerprint from the text. Traverse the edge if the two fingerprints are equal.

We still take constant time at each edge to determine if we jump down the tree or if we are done. Therefore, the complexity of this algorithm is $O(t \times depth(CT_D))$.

In general, $depth(CT_D)$ need not differ significantly (or at all) from $depth(T_D)$. In the following section, we present a scheme for producing a shallow search tree based on CT_D, thus speeding of the search algorithm.

4 Trie Based Dictionary Matching Algorithm

Many techniques have been used to balance trees in order to accelerate searching through them. Our problem, as described in Section 3 requires just such a balancing technique. We will consider the separator decomposition of a tree and show how this technique can be used in conjunction with the trie dictionary matching algorithm described above.

4.1 A Separator Decomposition Tree

The notion of the complete separator decomposition of a rooted tree is well known and has been extensively applied (see [17] or [22]). It has been used in the context of string matching by Naor [24] (the problem considered is on-line string matching after the text and the pattern are pre-processed separately).

Then the algorithm can be made to run in time $O(kt\sqrt{m})$ by simply checking, for each long prefix, if one of its $O(k)$ short suffixes matched. More accurately, let k_i be the number of dictionary patterns sharing the same long prefix as the pattern P_i. Then for a given text the running time is $O(t\sqrt{m}\sum_{i=1}^{d} k_i)$ where the sum is only over the dictionary patterns P_i for which there is a match between the text and their long prefix $L(P_i)$. While the worst case running time is the same as the basic algorithm above, $O(tm)$, it is clear that for many input instances this algorithm will perform much better.

Two dimensional dictionary matching The string matching algorithm of Karp and Rabin extends also for two-dimensional data. Our fingerprints based dictionary matching algorithms can be extended in a similar manner. We will provide more details in the full paper.

3 Tries

Consider the following simple algorithm for dictionary matching. For each text location, try to match each pattern by brute force. This algorithm clearly runs in $O(td)$. We will improve on this algorithm in several ways to improve the complexity. First, we use the well known data structure, the trie. A trie is a data structure that allows a set of strings to be stored and updated, and allows membership queries. A straightforward use of a trie for the dictionary problem leads to an algorithm that runs in time $O(tm)$ were m is the length of the longest pattern.

In the following subsections, we present some background needed to manipulate tries. In section 4, we will show how to modify a trie to speed up the search.

3.1 Definition and Construction

Tries are a common data structure for representing sets of strings [2].

Definition: Let $D = \{P_1, \ldots, P_k\}$ where $P_i \in \Sigma^*$ and $\$ \notin \Sigma$. A *trie* T_D of D is a tree such that each edge is labeled with a character from $\Sigma \cup \{\$\}$, and each pattern $P_i \in D$ is represented by a leaf l_i such that the concatenation of the edge labels from the root to l_i is the string $P_i\$$. Furthermore, the only nodes in a trie are exactly those so induced by the k leaves $l_1, \ldots, l_k$.

Constructing and updating a trie is straightforward. Suppose that we want to add string $P[1, \ldots, m]$ to D and to T_D. Then we simply start with the root of the tree as the current node, and at the beginning of P. We check if there is an edge labeled with the next character in P. If such a edge exists then we traverse it, resetting the current node, and advancing one step in the pattern P. When we can no longer traverse the tree, we create a linear subtree for the rest of the pattern and insert it below the current node. Finally, we add a single node at the bottom of the resulting tree with an edge labeled with a \$.

To remove a pattern P_i, simply start at leaf l_i and remove it. Proceed to the parent, and, if it now has outdegree zero, remove it. The process continues until a node is reached which still has a child left.

Time: For a bounded alphabet, the outgoing edge labels of each node can be checked in constant time. For unbounded alphabets, the edge labels can be held in a

Parallel algorithm The fingerprints of the text substrings can be computed using a prefix sum computation, as described in [18]. The dictionary patterns' fingerprints can be independently for each dictionary pattern. By using an appropriate parallel hashing algorithm (see Section 2.2 we get the following bounds with high probability:

Preprocessing: $O(\log m)$ time, $O(d)$ work.

Text Processing: $O(\log t)$ time, $O(tm)$ work.

Dynamic dictionary matching As before, we hold the fingerprints of the dictionary patterns in a hash table. Now the set of dictionary patterns is changing. Accordingly, we maintain the hash table by a dynamic-hashing algorithm. By using the real-time dictionary (see Section 2.2) each instruction can be processed in constant time with high probability.

We note that when the size of the dictionary changes significantly, we need to change the fingerprint function accordingly. While this may require $O(d)$ time, it is easy to see that it does not add more than an amortized constant factor per insertion/deletion. By using "masking" techniques (see, e.g., [15]) the amortized cost can be made non-amortized (with high probability). These considerations imply the following bounds:

Pattern Addition: $O(p)$

Pattern Deletion: $O(1)$

Text Processing: $O(tm)$.

Parallel algorithms Recall that in a parallel dynamic-hashing algorithm (Section 2.2) a batch of n insertions can be supported in $O(\log^* n)$ time with high probability. Membership queries and deletions take constant (worst case) time. The bottleneck for the parallel computation remains the computation of the fingerprint function, except for pattern deletion which takes constant time, i.e. the same time as hash table deletion. We get, with high probability,

Pattern Addition: $O(\log p)$ time, $O(p)$ work.

Pattern Deletion: $O(1)$ time, $O(1)$ work.

Text Processing: $O(\log t)$ time, $O(tm)$ work.

An input sensitive algorithm In the above algorithm the text scan takes $O(tm)$ time. We briefly present here an input-sensitive algorithm which improves on this bound for certain, rather "natural", cases.

Each string P_i in the dictionary D can be decomposed into a "long" prefix $L(P_i)$ and "short" suffix $S(P_i)$ by setting $L(P_i) = P_i[1], \ldots, P_i[c\sqrt{m}]$, where $c\sqrt{m} \leq p_i < (c+1)\sqrt{m}$, and setting $S(P_i) = P_i[c\sqrt{m}+1], \ldots, P_i[p_i]$. Thus, $P_i = L(P_i)S(P_i)$. Note that short strings come in at most $\sqrt{m}$ different sizes, as do long strings. We can therefore use the above $O(tm)$ algorithm to find all short and long strings of a dictionary in $O(t\sqrt{m})$ time. The remaining issue is how to reconstruct the dictionary strings from the short and long strings that match at each text location.

A naive approach would be to store all pairs $\langle l, s \rangle$ such that long string L_l and short string S_s form a dictionary string $L_l S_s$ in the hash table. Then we could check, for each text location, which of the $O(\sqrt{m})$ long strings and $O(\sqrt{m})$ short strings that matched at that location form a valid pair. This clearly takes $O(tm)$ time.

However, one may expect that very few strings would have the same long prefix. Suppose that no more than k strings in the dictionary have identical $L(\cdot)$ prefixes.

Parallel algorithms A first optimal[7] parallel dynamic-hashing algorithm was introduced in [10]; its running time is $O(n^\epsilon)$ for any fixed $\epsilon > 0$. An optimal parallel (static) hashing algorithm in $O(\log n)$ expected time was given in [21], followed by an $O(\log \log n)$ expected time algorithm presented in [14]. An $O(\log^* n \log \log^* n)$ expected time[8] algorithm was given in [20]; its time complexity was later shown to actually be $O(\log^* n)$ with high probability [15]. A similar improvement (from $O(\log^* n \log \log^* n)$ to $O(\log^* n)$) was also given by [6]. A real-time parallel dynamic-hashing algorithm was given by [15]; the algorithm supports a batch of n membership-query or delete instructions in constant time, and a batch of n insert instructions in $O(\log^* n)$ time with high probability, using an optimal number of processors.

Applications As noted in [21], hashing can replace sorting-based naming assignment procedures which are used to map potentially large alphabets into a linear size alphabet. In the algorithm given in [1], the $\log \sigma$ factors in the $O(d \log \sigma)$ preprocessing time and in the $O(t \log \sigma)$ text processing time are the result of a binary search through characters which occur in the dictionary patterns. This binary search can be replaced by lookups into a hash table, thus improving the complexities to $O(d)$ for preprocessing and $O(t)$ for text processing.

2.3 Fingerprint based Dictionary Matching

We are ready to present a first dictionary matching algorithm. The algorithm is extremely simple, and its parallel complexity is superior to the previously known algorithm. Next we extend it to dynamic dictionary matching.

We extend the Karp-Rabin algorithm for dictionaries as follows. Assume, for the time being, that all patterns are of the same length m. We will use a fingerprint function of size $O(\log t + \log d)$. This will guarantee that any false match between a substring and a pattern in the dictionary still occurs with sufficiently small probability.

The dictionary patterns are kept in a hash table. The Karp-Rabin algorithm is employed with a slight modification. First, the fingerprint $f(S)$ is computed (as in the string matching algorithm) for each substring S of size p in the text. Now, for each substring S, we perform a membership query into the hash table. In constant time we find out if there is a match. We note that the word size is $O(\log t + \log d)$ and thus the fingerprint can be processed in constant time. The hash table is built using an appropriate hashing algorithm from Section 2.2.

In general, the patterns need not be of the same size. Let the length of the longest pattern be m. Then the patterns can be partitioned into at most m equivalence classes based on length. Using the single length algorithm as a subroutine, we can match for each class independently. The resulting complexities are:

Preprocessing: $O(d)$

Text Processing: $O(tm)$

---

[7] An *optimal* parallel algorithm is an algorithm whose time-processor product is the same, up to a constant factor, as the best known sequential time.

[8] Let $\log^{(i)} x \equiv \log(\log^{(i-1)} x)$ for $i > 1$, and $\log^{(1)} x \equiv \log x$; $\log^* x \equiv \min\{i : \log^{(i)} x \leq 2\}$. The function $\log^*(\cdot)$ is extremely slow increasing and for instance $\log^* 2^{65536} = 5$.

and $f(S_i)$. Karp and Rabin showed that one can select a fingerprint function f with the following properties:

1. The function $f(\cdot)$ can be computed for all substrings in T, in $O(t)$ time, and in parallel in $O(\log t)$ time and optimal number of processors.
2. The function f gives a "reliable" fingerprint: if $|f(s)| = O(\log p + \log t)$ then the probability for a false match among t pairs of strings, each of length p, can be bounded by $O(t^{-c})$, for any constant $c > 0$.

The Karp-Rabin algorithm can be described in two steps. First, a fingerprint of length $O(\log t)$ is computed for the pattern P and for all substrings of lengths p. Then, the fingerprint $f(P)$ is checked against each position in the text T; each comparison is assumed to take constant time since the word size is $O(\log t)$. In $O(t)$ sequential time, or in $O(\log t)$ parallel time and optimal work, the algorithm finds all positions i in the text such that $f(P) = f(S_i)$. Each of these positions is a candidate, with high probability, for a match. We can check a candidate match in $O(p)$ steps in the straightforward manner. Since a mismatch of a candidate match only occurs with a polynomially small probability, the number of steps required for finding one occurrence of P in the text or deciding that P does not appear is $O(t)$, with high probability. However, if one wishes to list all occurrences of P without an error, the number of steps would be, with high probability, $O(t + lp)$, where l is the number of times P appears in T. Note that in this case lp is the size of the output.

2.2 Hashing

Given a set S of n keys from a finite universe $U = \{0, \ldots, q-1\}$, i.e. $S \subset U$ and $|S| = n$, the *hashing* problem is to find a one-to-one function $h : S \mapsto [1, dn]$, for some constant d, such that h is represented in $O(n)$ space and for any $x \in U$, $h(x)$ can be evaluated in constant time. The function h is called a *hash function* and the induced data structure is called a *hash table*. The requirement implies that set-membership queries can be processed in constant time.

The *dynamic hashing* problem is to maintain a data structure which supports the *insert*, *delete*, and *membership query* instructions. That is, the set of keys S is changing dynamically. It is required that at all times, the size of data structure be linear in the number of keys stored in it.

Sequential algorithms In a seminal paper, Fredman, Komlós, and Szemerédi [12] gave a simple algorithm for the hashing problem, whose expected running time is $O(n)$. The general framework introduced in [12] was used in all subsequent papers mentioned here. A dynamic-hashing algorithm with $O(1)$ expected amortized cost per insertion was introduced by [9]. A real-time dynamic-hashing, in which every insertion can be completed in constant time with high probability was introduced in [11]. In a recent paper, [8] show how to modify the FKS hashing algorithm so that its running time would be $O(n)$ with high probability, and so that the number of random bits used by the algorithm is substantially reduced. They also give a simplified real-time dynamic-hashing algorithm that requires substantially fewer random bits than in [11].

		Static		Dynamic	
prep.& update	time			$\log m \log d$ †	[3]
	work			$p \log d$	
text scan	time			$\log m \log d$	
	work			$t \log m \log d$	
prep.& update	time	$\log d \log^* d$	§5	$\log p \log^* p$	§5
	work	$d \log d$		$p \log p \log k$	
text scan	time	$\log d$		$\log d$	
	work	$t \log d$		$t \log d$	
prep.& update	time			$\log p$	§2.3
	work			p	
text scan	time			$\log t$	
	work			tm	

Table 2. Parallel algorithms. (The preprocessing and update complexities are per pattern.) † Only semi-dynamic: no deletion.

The rest of the paper is organized as follows. In Section 2, we introduce fingerprints and perfect hashing and show how to combine them into a simple dictionary matching algorithm. In Section 3, we discuss the trie data structure and describe simple trie-based dictionary-matching algorithms. These algorithms serve as a warm-up towards the main algorithm. In Section 4, we present the main algorithm of the paper, which is based on tries and on a tree decomposition technique. This algorithm shares some features with the algorithm presented in [24]; it is however based on a simpler data structure and in fact solves a different problem. Parallel algorithms are presented in Section 5. Our algorithms are more efficient than previous ones, and are the first to support efficiently parallel dynamic dictionary-matching with the delete operation.

2 Fingerprints and Hashing

In this section, we discuss fingerprints, a basic tool for randomized string matching. The fingerprints technique will be an essential tool in the algorithms given in the following sections. We show here how they can be used, together with perfect hashing, to get extremely simple algorithms for dictionary matching.

2.1 Fingerprints

Karp and Rabin [18] introduced a simple, yet powerful, technique for randomized string matching. In their algorithm, a given string P is replaced by a fingerprint function $f(P)$: a short string that represents the string. Similarly, each candidate substring S_i in the text T is substituted by a fingerprint $f(S_i)$, where S_i is the substring of length p starting at position i in the text, $i = 1, \ldots t - p + 1$. The idea is to test for a match between P and S_i, by first testing for a match between $f(P)$

against all known strings to find ones that are related. Clearly one would like an algorithm which is fast, even given some huge dictionary.

Aho and Corasick [1] introduced and 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 constant size alphabet). However, if the dictionary changes, that is, if patterns are inserted or deleted, then the work needed to update the dictionary data structure is proportional to the dictionary size, since the entire dictionary needs to be reconstructed. The issue of *dynamic* dictionary matching was introduced in [3]. Related work on dynamic dictionary matching appears in [4, 16].

In this paper we introduce several randomized algorithms that use combinations of useful techniques, such as fingerprints, hash tables, the trie data structure, and tree decomposition. Our algorithms take different approaches than previous algorithms, and they are either simpler or have better performance.

In [18], Karp and Rabin introduced the notion of fingerprints. These are short representative strings which can be used to distinguish, with high probability, between different strings in constant time. Karp and Rabin used the fingerprints to give an elegant algorithm for string matching which runs in linear time, with high probability. Throughout the paper, we follow Rabin and Karp [18] in assuming that the operations allowed in constant time include comparisons, arithmetic operations, and indirect addressing, where the word size is $O(\log |D| + \log |T|)$.

Our results are summarized in Tables 1-2, along with the previously known results. All stated results are asymptotic (the "O" is omitted) and are randomized in the sense that within the stated time the algorithms terminate with one sided error (i.e., the "Monte-Carlo" type). All our algorithms can be easily converted into "Las-Vegas" type algorithms, where there is no error and the randomization is on the running-time only. Let $d = \sum_{i=1}^{k} |P_i|$ be the size of the dictionary, σ be the effective alphabet, that is, the number of distinct characters that occur in D, $t = |T|$ be the text size, $p = |P|$ be the size of the pattern to be inserted or deleted, and m be the size of the largest pattern in D.

	Static		Dynamic	
prep.&updates	$d \log \sigma$	[1]	$p \log d$	[3]
text scan	$t \log \sigma$		$t \log d$	
prep.&updates	d	§2.2	$p \log k$	§5.2
Text Scan	t		$t \log k$	
prep.&updates	d	§4	p	§2.3
text scan	$t \log d$		mt	

Table 1. Serial algorithms

Efficient Randomized Dictionary Matching Algorithms

(Extended Abstract)

Amihood Amir[1] Martin Farach[2] Yossi Matias[3]

[1] Georgia Tech[†]
[2] DIMACS[‡]
[3] Univ. of Maryland and Tel Aviv Univ.[§]

Abstract. The standard string matching problem involves finding all occurrences of a single pattern in a single text. While this approach works well in many application areas, there are some domains in which it is more appropriate to deal with *dictionaries* of patterns. A dictionary is a set of patterns; the goal of dictionary matching is to find all dictionary patterns in a given text, simultaneously.
In string matching, randomized algorithms have primarily made use of randomized hashing functions which convert strings into "signatures" or "finger prints". We explore the use of finger prints in conjunction with other randomized and deterministic techniques and data structures. We present several new algorithms for dictionary matching, along with parallel algorithms which are simpler of more efficient than previously known algorithms.

1 Introduction

Traditional *pattern matching* has dealt with the problem of finding all occurrences of a single pattern in a text. A basic instance of this problem is the *exact string matching* problem, the problem of finding all exact occurrences of a pattern string in a text. This problem has been extensively studied. The earliest linear time algorithms include [19] and [7].

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 in a given text of patterns from a given set of patterns, called a *dictionary*. This is the *dictionary matching* problem. In addition to its theoretical importance, dictionary matching has many 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 a string of symbols) the next step is to compare the string

[†] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; (404) 853-0083; amir@cc.gatech.edu; partially supported by NSF grant IRI-90-13055.

[‡] DIMACS, Box 1179, Rutgers University, Piscataway, NJ 08855; (908) 932-5928; farach@dimacs.rutgers.edu; supported by DIMACS under NSF contract STC-88-09648.

[§] Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; matias@umiacs.umd.edu; partially supported by NSF grants CCR-9111348 and CCR-8906949.