Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

The Dynamic Dictionary Matching Problem Revisited

T.W. Lam and K.K. To*

Department of Computer Science
University of Hong Kong
Pokfulam, Hong Kong
{twlam,kkto}@cs.hku.hk

Abstract

The dynamic dictionary matching problem asks for a data structure to represent a set of pattern strings $\Delta = \{x_1, x_2, \cdots, x_k\}$, which can be updated efficiently when a pattern is inserted into or deleted from Δ, and which can support efficient searching of a text string to report all occurrences of the patterns of Δ. The work of Amir et al. [3, 2] has produced an elegant solution supporting the update of Δ for a pattern x in $O(|x| \log n)$ time and the search of a text t in $O((|t| + \texttt{tocc}) \log n)$ time, where n denotes the total length of the patterns in Δ and tocc *denotes the total number of occurrences of the patterns in t. Recently, the time complexity of both operations has been improved, with the factor of $\log n$ replaced by $\log n / \log\log n$ [4].*

This paper presents a new solution in which an update takes $O(|x| + \log k)$ time and a search $O((|t| + \texttt{tocc}) \log k)$ time. Moreover, we can further improve the search time to $O((|t| + \texttt{tocc})\frac{\log k}{\log\log k})$, but the update time increases to $O(|x| \log k / \log\log k)$.

1 Introduction

This paper is concerned with a string matching problem, called the dynamic dictionary matching problem. We are given a set of pattern strings $\Delta = \{x_1, x_2, \cdots, x_k\}$, also referred to as the *dictionary*, which can change over time with new patterns added in and old ones removed. We assume that characters in these strings are chosen from a fixed alphabet Σ of bounded size. The dictionary matching problem asks for a data structure to represent Δ such that any changes of Δ can be processed efficiently and, given any text string t, the occurrences of the patterns of Δ in t can be reported efficiently.

For the static case in which no insertion or deletion is supported, optimal solutions already exist [1, 5]. These solutions each can build a data structure representing Δ in $O(n)$ time and search a text t in $O(|t| + \texttt{tocc})$ time, where n denotes the total length of patterns in Δ and tocc is the total number of occurrences of the patterns in the given text.

Amir, Farach, Galil, Giancarlo, and Park [2, 3] were the first to obtain non-trivial solutions to the dynamic dictionary matching problem. Assume the alphabet is of bounded size. The solution of [3] supports an insertion or deletion of a pattern x in $O(|x| \log n)$ time and a search of a text t in $O((|t| + \texttt{tocc}) \log n)$ time. Idury and Schäffer [9] later obtained a solution which can trade the update time for the search time; for any constant c, their scheme achieves $O((|t| + \texttt{tocc})c)$ search time with update time of $O(c|x|n^{1/c})$. Recently, Amir, Farach, Idury, La Poutré, and Schäffer [4] improved the time bounds in [3], giving a solution with update time $O(|x|\frac{\log n}{\log\log n})$ and search time $O((|t| + \texttt{tocc})\frac{\log n}{\log\log n})$.

At present, it is still open whether the factor of $\log n$ can be removed completely from the time complexity of both update and search, producing a solution with the same time complexity as the static case. Perhaps this is too ambitious in view of the dynamic nature of the problem. A less ambitious open problem is whether there is a solution with time complexity depending on the total number of patterns instead of the total length of the patterns. This is a realistic direction as the problem do not require any query about the substrings inside the patterns, and we always consider each individual pattern as a single entity. This paper presents a positive result along these directions. More specifically, we improve the update time to $O(|x| + \log k)$ and search time to $O((|t| + \texttt{tocc}) \log k)$, where k denotes the number of patterns currently in the dictionary. It is noteworthy that the update time complexity involves a sum rather than a product of $|x|$ and $\log k$. Also, the dictionary can be built in $O(n + k \log k)$ time by inserting the patterns one by one.

Like the previous work [2, 3, 4], the solution pre-

*Research supported in part by Hong Kong RGC Grant 338/065/0027

sented in this paper is based on the suffix tree [10]. Our improvement roots at very efficient data structures to manage the locations of the patterns on the suffix tree. Using other recent results on integer searching [7], we can further improve the search time to $O((|t|+\mathtt{tocc})\log k/\log\log k)$, but the update time increases to $O(|x|\log k/\log\log k)$.

The remainder of this paper is organized as follows: Section 2 describes some off-the-shelf data structures our solution makes use of. Section 3 gives a brief review of the suffix tree and the way it is used to solve the dynamic dictionary matching problem. Section 4 discusses the new data structures which can augment the searching process on the suffix tree, and which can be updated efficiently. Sections 5 and 6 give the details of the searching and updating algorithms. In the last section, we highlight the techniques involved in trading the update time for the search time.

2 Tools from the literature

Our new solution to the dynamic dictionary matching problem makes use of two off-the-shelf data structures. The first one is Dietz and Sleator's data structure [6] for maintaining the order in a list. We call this data structure a DS-list. It requires only linear space, supporting in constant time each of the following operations:

1. $\mathtt{Order}(p,q)$: Given two elements p,q of the list (more precisely, p,q are pointers to elements), determine whether p precedes q;
2. $\mathtt{Insert}(p,q)$: Insert a new element p into the list at the position immediately before an existing element q;
3. $\mathtt{Delete}(q)$: Delete the element q from the list.

The second data structure is for maintaining the nesting structure of a sequence of balanced parentheses φ. It is intended to support the following operations:

1. $\mathtt{Nep}(p)$: Given a parenthesis p of φ (more precisely, p is a pointer to a parenthesis), find the nearest pair of parentheses enclosing p;
2. $\mathtt{Insert}(p,q)$: Insert a matching pair of parentheses into φ such that the left one is just after an existing parenthesis p and the right one is just before another existing parenthesis q. It is required that the resultant sequence must be balanced. (To insert a left parenthesis at the beginning of φ, we set $p=-\infty$; similarly, to insert a right parenthesis at the end of φ, we set $q=+\infty$.)
3. $\mathtt{Delete}(p,q)$: Given a matching pair of parentheses p and q, remove them from φ.

It is indeed not difficult to figure out a data structure that can support each of the above operation in $O(\log l)$ time, where l denotes the current length of the sequence. Güting and Wood [8] suggested an implementation that requires linear space. To achieve better time complexity is possible but not trivial. Amir *et al.* [4] have given a fairly complicated solution improving the time complexity to $O(\log l/\log\log l)$.

3 Background

In this section we review Amir *et. al.*'s suffix-tree-based solution [3] to the dynamic dictionary matching problem. Our new solution basically follows a similar approach.

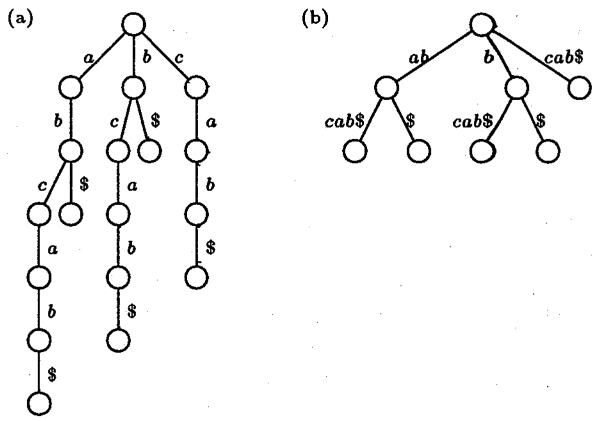

Figure 1: (a) A trie for a string "*abcab*"; (b) The corresponding suffix tree.

Let x be a string of n characters over a bounded alphabet Σ. Suppose \$ is a special symbol not matching any character in Σ, including itself. Let I be a trie comprising all the suffixes of x appended with \$, i.e. $x[1:n]\$$, $x[2:n]\$$, $\cdots$, $x[n:n]\$$. Every edge of I is labeled with a character in x or \$. I has exactly n leaves, each corresponding to a unique suffix of x. A suffix tree R for x is a compacted version of I, in which the out-degree of an internal node (except the root) is at least two and every edge is labeled with a nonempty substring of $x\$$. The number of nodes in R is at most $2n$. Figure 1 depicts an example.

For each node u, the path label of u (also denoted by $\mathtt{path\text{-}label}(u)$) is defined to be the concatenation of the labels along the path from the root to u. The path label of a leaf is equal to a unique suffix of x appended with \$. We denote by w_0 the leaf with path label equal to $x\$$. Using McCreight's algorithm [10], it takes only $O(n)$ time to build the suffix tree R, as well as storing a useful pointer called the *suffix link* in each internal node v. If $\mathtt{path\text{-}label}(v)=ax$ for some $a\in\Sigma$ and $x\in\Sigma^*$, there is always another internal node v' with $\mathtt{path\text{-}label}(v')=x$. The suffix link of v is defined to be a pointer to v'.

Using R, we can find the occurrences of x in any text string t of length m in time $O(m)$. The idea is to search R to match every suffix of t as many characters as possible; if a suffix $t[i:m]$ can match the path-label of w_0 (i.e. the leaf representing $x\$$) except the last \$ symbol then x appears in t starting from the position i. It is obviously too slow to match each suffix of t character by character starting from the root of R. Fortunately, based on the suffix links, we can actually determine in constant amortized time how far a suffix of t can match R [10, 3], thus giving an $O(m)$ time algorithm for reporting the occurrences of x in t.

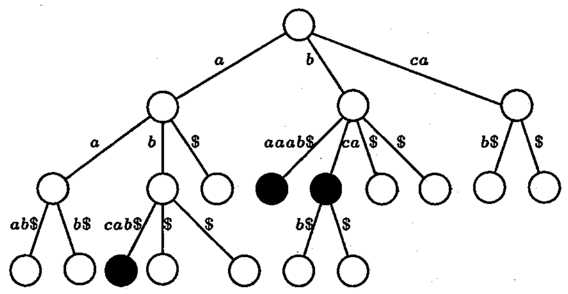

Figure 2: A suffix tree for the strings "*abcab*", '*baaab*" and "*bca*". Marked nodes are filled.

To represent a set of strings $\Delta = \{x_1, x_2, \cdots, x_k\}$, we extend McCregiht's algorithm [10, 3] to construct a suffix tree R comprising all the suffixes of strings in Δ appended with \$. The construction requires $O(n)$ time, where n denotes $\sum_{i=1}^{k} |x_i|$. Note that R has exactly n leaves. To ease the job of text searching, we put a mark on R for each string x_i of Δ: If there is an internal node v whose path label is exactly x_i, v is marked; otherwise, the mark is put on a leaf w with path label equal to $x_i\$$ (see Figure 2). To efficiently report the occurrences of strings of Δ in a text t, we again make use of the suffix links. For each suffix $t[i:m]$, we first locate the endpoint of the longest path in R that $t[i:m]$ matches. If the longest possible match of $t[i:m]$ ends at exactly a marked internal node or is one \$ symbol from a marked leaf, we report the occurrence of the corresponding string. Moreover, every marked node on the path that $t[i:m]$ matches also defines an occurrence of a string of Δ. Thus, we would like to traverse backward to the root of R to recover all the marked nodes. Yet brute-force traversal is very time consuming (though locating the endpoint of the path requires only $O(1)$ amortized time). We need more efficient methods to report those marked nodes.

Suppose the dictionary Δ is fixed. We can simply store a special pointer in every node of R keeping track of the nearest marked ancestor. This allows us to report an occurrence of a string in constant time. Thus, finding all occurrences of strings of Δ in a text t takes $O(|t| + \texttt{tocc})$ time.

The above technique does not work for the dynamic dictionary matching problem. When a string x is inserted into or deleted from Δ, we can modify McCreight's algorithm [10] to update the suffix tree R in $O(|x|)$ time, but excluding the special pointers. In fact, deleting even a short string may possibly cause the special pointers of many nodes of R to get updated. Amir *et. al.* [3] abandoned the idea of special pointers and used a dynamic data structure called dynamic trees [11] to organize the marked nodes. The dynamic trees can be updated using $O(|x| \log n)$ time when a string x is inserted or deleted, where n is the total length of the strings currently in Δ; however, the time for searching the nearest marked ancestor increases to $O(\log n)$, and searching a text t requires $O((|t| + \texttt{tocc}) \log n)$ time.

In the rest of this paper, we present new data structures to organize the marked nodes such that a query of the nearest marked ancestor can be answered in $O(\log k)$ time, where k is the number of patterns currently in Δ. The time for text searching can thus be reduced to $O((|t| + \texttt{tocc}) \log k)$. Surprisingly, we do not sacrifice the time for processing an insertion or deletion. In fact, the update time is reduced even more drastically to $O(|x| + \log k)$.

4 Bookkeeping of marked nodes

In this section we show new data structures to organize the marked nodes of a suffix tree, which can speed up the updating and searching process. The space required is in the same order as that of the suffix tree. The data structures are based on the following linear representation of a suffix tree. (Amir *et. al.* [4] have studied another linear representation of a suffix tree, based on which they derived a different approach to solve the dynamic dictionary matching problem, improving the time bound in [3] by a factor of $\log \log n$.)

Let R be a suffix tree with n nodes. Suppose $k < n$ nodes of R are marked. We define a sequence of balanced parentheses, called φ, to capture the structure of R. If R contains a single node, the sequence is simply "()". This pair of parentheses is associated with the only node of R. If the root of R has one or more children, the sequence of R is obtained by concatenating the sequences of the subtrees rooted at the children of the root of R, and then enclosing the resultant sequence by a pair of left and right parentheses. The outermost pair of parentheses is associated with the root.

The sequence φ contains exactly n pairs of parentheses, each associated with a node of R. Note that the parentheses associated with a node u is enclosed by the pair of parentheses associated with any ancestor of u. That means, to find the nearest marked ancestor of a node u, we can examine the left (or right) parenthesis associated with u and find the nearest enclosing pair of parentheses which are associated with a marked node. Below, we define three data structures for representing the parentheses in φ, which can also be regarded as an implicit representation of the marked nodes. These data structures, other than allowing efficient updating, are aimed at supporting the operation of finding, for any parenthesis in φ, the nearest enclosing pair of parentheses which are associated with a marked node of R as fast as in $O(\log k)$ time.

1. a DS-list L: We represent φ as a DS-list L, which contains exactly $2n$ elements. We assume that each node in R stores two pointers to its left and right parentheses on L, and vice versa. The DS-list L allows us to compare in constant time, for any two nodes α, β of R, whether the left (or right) parenthesis of α precedes that of β.

2. a parentheses-maintenance data structure M: The parentheses associated with the marked nodes of R define a subsequence of φ. This subsequence is represented by a parentheses-maintenance data structure M. If R has k marked nodes, M contains $2k$ parentheses. Also, we assume each marked node and its associated parentheses in M are linked by two-way pointers.

3. an AVL tree A: We build an AVL tree A consisting of $2k$ nodes, each corresponding to a parenthesis associated with one of the marked nodes of R. The ordering among the nodes of A is determined by the precedence relationship given by L.

Let us look at the details of finding, for any parenthesis α in φ, the nearest pair of enclosing parentheses which are associated with a marked node. W.l.o.g., assume α is a left parenthesis. If α itself is associated with a marked node, then a simple query to M suffices. Otherwise, we search A to find the rightmost parenthesis preceding α. Denote this parenthesis by β. By definition of A, β is associated with a marked node. If β is a left parenthesis then β and its right counterpart enclose α and are the answer for α. If β is a right parenthesis, we consult M again to find, among the parentheses associated with the marked nodes, the nearest pair enclosing β, which are also the nearest pair enclosing α.

5 Text searching

In Section 3 we observed that the dynamic dictionary matching problem can be solved using a suffix tree R augmented with data structures for organizing the marked nodes. Thus, based on the data structures L, M, and A described in the previous section, we can find the occurrences of the patterns represented by R in a text t as follows:

```
Input: A text $t$ of length $m > 0$
1: for $i = 1$ to $m$ do
2:   $p \leftarrow$ the longest prefix of $t[i:m]$ matching a prefix of any suffixes stored in $R$
3:   $w \leftarrow$ the highest node in $R$ which path label includes $p$ as a prefix
4:   if $w$ is marked and path-label($w$) is either $p$ or $p\$$ then
5:     report occurence
6:   $w \leftarrow$ the nearest marked ancester of $w$ {See Section 4}
7:   while $w \neq$ not-found do
8:     report occurence
9:     $w \leftarrow$ the nearest marked ancester of $w$
```

Time complexity: In the above algorithm, lines 2 and 3 are done concurrently using the standard suffix tree algorithm [3, 10]. Using suffix links they can be done in amortized constant time over the outermost for-loop. Lines 4 and 5 can be done in constant time. In lines 6 to 9, we invoke the algorithm in Section 4 for occ + 1 times, where occ denotes the number of occurences we report in the current iteration. Each such invocation needs $O(\log k)$ time. Summing over the $|t|$ iterations, we spend $O(|t|)$ time for matching the suffixes of t in R and $O((|t| + \texttt{tocc}) \log k)$ time for finding nearest marked enclosing parentheses in φ. The latter clearly dominates.

6 Dictionary update

We complete our algorithms by explaining in detail how the data structures are modified upon insertions and deletions of patterns. The time complexity is $O(|x| + \log k)$ where x denotes the pattern to be inserted or deleted.

The suffix tree R: We update the suffix tree R using the algorithm of Amir *et al.* [3] with slight modification. According to their algorithm, when a new pattern x is inserted into the dictionary Δ, we insert all suffixes of x into R in a decreasing order of their length. Each insertion causes a new leaf to be created under a possibly freshly created internal node. The deletion of pattern is processed similarly, with all suffixes of x deleted from R. Each deletion causes a leaf and possibly its parent to be deleted. By making use of the previously installed suffix links, inserting or deleting a suffix can be done in amortized constant time over the insertion or deletion of a pattern. Thus, the update of R due to a pattern x costs only $O(|x|)$ time.

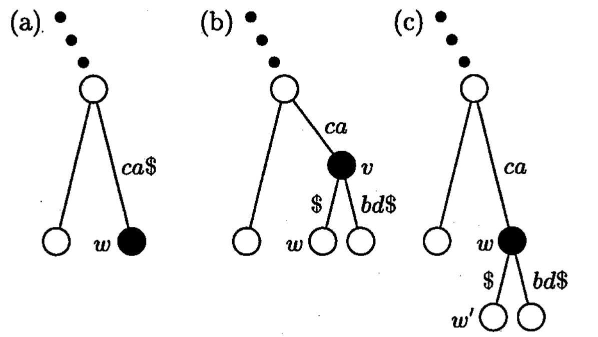

Figure 3: (a) w is a marked leaf representing a pattern in Δ, say, "*abca*". (b) Inserting a suffix "*abcabd$*" would create an internal node v. (c) A leaf w' is created instead of an internal node v.

Interestingly, in the course of inserting a suffix, a straightforward implementation of the algorithm of Amir *et al.* [3] may cause a mark put previously on R (representing an existing pattern in Δ) to move from a leaf to an internal node. More specifically, suppose Δ contains a pattern x_i with the associated mark currently on a leaf w of R, and we want to insert a suffix σ into R, which contains x_i as a prefix. The insertion would then create an internal node v with path label equal to x_i, which have two children, namely w and a new leaf representing σ (see Figure 3 *a* and *b*). By our definition of marked nodes, the mark for x_i should be moved from w to v. Of course, moving a mark on R takes only constant time, yet this induces an additional update on the data structures M and A, which would require $O(\log k)$ time. Similarly, mark movement can also happen during the deletion of a suffix.

To obviate the need to move a mark, we modify the algorithm slightly. Whenever an internal node v is to be inserted as the new parent of a marked leaf w in such a way that the mark should be moved to v afterwards, we avoid this insertion. Instead we insert a new leaf w' under w, thereby making w an internal node, which will play the role of v. The labels of the edges pointing towards w and w' are modified so as to emulate the effect of inserting v (Figure 3c). As a result, the association between a previously inserted pattern and its marked node on R is left intact, and we avoid updating the data structures M and A. Similarly, we can modify the algorithm for deleting a suffix to avoid moving the marks on the nodes representing other existing patterns.

The DS-list L: As mentioned before, inserting a suffix into the suffix tree R creates a leaf and possibly an internal node. The conceptual linear representation φ changes as follows: Firstly, suppose a new internal node v is inserted as the parent of an existing node u. A new pair of parentheses, associated with v, should be inserted immediately outside the parentheses associated with u. Secondly, when a new leaf w is inserted below an internal node v already in R, a pair of parentheses, associated with w, emerges within the parentheses associated with v. The new pair encloses no other parentheses, and is located just to the left of the closing parenthesis associated with v. In both cases, L can be modified to reflect the change of φ using only constant time. Similarly, when a suffix is deleted from R, a leaf and possibly an internal node are deleted. The associated pairs of parentheses in φ should disappear, which can easily be reflected in L by deleting the corresponding parentheses in L. Again, this requires only constant time.

The data structures M and A: After all the suffixes of a new pattern x are inserted into R, we put on R a mark associated with x. By the definition of mark, it should be put on a node u where `path-label`$(u) = x$ if such a node exists, or `path-label`$(u) = x\$$ otherwise. Since the marking status of the nodes is represented implicitly in the parentheses maintenance structure M and the AVL tree A, we must insert a pair the parentheses associated with u in both data structures. Insertion in A is trivial. Afterwards, we find the predecessor and successor in A of the left and right parentheses just inserted, which give the correct location in M for inserting the pair of parentheses associated with u. When a pattern is deleted, we unmark a node in R. Thus, we delete the parentheses in A, as well as in M, associated with the deleted nodes in R.

Time complexity: In summary, to insert or delete a pattern x, $|x|$ suffixes are inserted into or deleted from the suffix tree R, each taking $O(1)$ time amortized over the insertion or deletion of the pattern. With respect to the DS-list L, at most $|x|$ pairs of parentheses are inserted into or deleted, using a total of $O(|x|)$ time. On the other hand, the update of x requires us to mark or unmark only one node of R, incurring $O(\log k)$ time for updating A and M. In total, an update of Δ due to x costs $O(|x| + \log k)$ time.

7 Trading for search efficiency

We have mentioned in Section 3 that Amir *et al.* [3] devised a parentheses maintenance data structure so that the update and query operations in $O(\log l/\log\log l)$ time, where l denotes the number of pairs of parentheses in the structure. Obviously, incorporating this data structure into our algorithm would improve the search time to $O(|t|\log k + \texttt{tocc}\frac{\log k}{\log\log k})$.

Now the bottleneck of our algorithm lies on the AVL tree A. If we have a faster data structure which can support the insert, delete, search, predecessor and successor operations as an AVL tree, the search time of our algorithm improves. Intuitively, we would like to replace the AVL tree with a fusion tree [7], which can perform the AVL-tree operations in $O(\log l/\log\log l)$ time, where l denotes the number of elements stored. However, a fusion tree cannot be used in our case in a straightforward manner since it operates only on integers. Recall that the elements of A are elements of the DS-list L instead of integers. In fact, every time we compare two elements of A we have to issue an `Order` query to the DS-list. Our new solution integrates the DS-list structure with the fusion tree to replace both L and A. That is, the combined data structure is still conceptually in the form of a list keeping track of all elements stored in L, as well as their marking status captured by A. It supports each of the following operation in $O(\log l/\log\log l)$ time, where l denotes the number of marked elements (or equivalently, the number of elements that would have been stored in A).

1. `Insert`(p, q): Insert a new element p into the list at the position immediately before an existing element q; p is assumed to be unmarked;
2. `Delete`(q): Delete an unmarked element q from the list.
3. `Mark`(q): Mark an element q in the list.
4. `Unmark`(q): Unmark an element q in the list.
5. `FindMark`(q): Find the marked predecessor nearest to q in the list.

As a result, the operation of searching for the nearest enclosing parentheses associated with a marked node in the suffix tree (as describe in Section 4) requires only $O(\log k/\log\log k)$ time rather than $O(\log k)$ time. However, the update of the combined data structure due to an insertion and deletion of a suffix in the suffix tree R will require the same amount of time instead of $O(1)$ time. Thus, the update complexity becomes $O(|x|\log k/\log\log k)$ while the search complexity improves to $O((|t| + \texttt{tocc})\frac{\log k}{\log\log k})$.

The details of the combined data structure is left to the full paper.

References

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

[2] Amihood Amir and Martin Farach. Adaptive dictionary matching. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 760–766, San Juan, Puerto Rico, 1–4 October 1991.

[3] Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, October 1994.

[4] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, June 1995.

[5] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the Sixth International Colloquium on Automata Languages and Programming*, pages 118–132, 1979.

[6] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, 25–27 May 1987.

[7] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

[8] R. H. Güting and D. Wood. The parenthesis tree. *Information Sciences*, 27:151–162, 1982.

[9] Ramana M. Idury and Alejandro A. Schäffer. Dynamic dictionary matching with failure functions. *Theoretical Computer Science*, 131(2):295–310, 12 September 1994.

[10] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[11] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.