later than when its ending position is scanned. However, we can change our algorithm so that it finds patterns according to their ending positions, thus making the search on-line. Notice that the internal nodes of the suffix tree T_{D_i} and their suffix links make a tree $\widetilde{T}$. We partition the tree $\widetilde{T}$ into a forest of trees by deleting $(v, SL(v))$ for all marked nodes v, and maintain the forest by using the dynamic trees. Then when an occurrence of a pattern p is found in the text, all patterns that are suffixes of p can be found by going up the tree $\widetilde{T}$ as in the procedure *FINDALL*. The time complexities of insertion, deletion, and search remain the same.

References

[AC75] A.V. Aho and M.J. Corasick, Efficient string matching: An aid to bibliographic search, *Comm. of the ACM* **18** (1975), 333–340.

[AF91] A. Amir and M. Farach, Adaptive dictionary matching, *Proc. 32nd IEEE Symp. Found. Computer Science* (1991), 760–766.

[ASU86] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986.

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

[CS84] M.T. Chen and J. Seiferas, Efficient and elegant subword-tree construction, In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pp. 97–107, Berlin, 1984. NATO ASI Series , Vol. F12, Springer-Verlag.

[C79] B. Commentz-Walter, A string matching algorithm fast on the average, *Proc. 6th ICALP* (1979), 118–132.

[D88] C. DeLisi, Computers in molecular biology: Current applications and emerging trends, *Science* **24** (1988), 47–52.

[GGP91] Z. Galil, R. Giancarlo, and K. Park, Fully dynamic dictionary matching. Technical Report, Department of Computer Science, Columbia University, 1991.

[H90] A. Hume, Keyword matching, Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1990.

[K73] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[KMP77] D.E. Knuth, J.H. Morris, and V.B. Pratt, Fast pattern matching in strings, *SIAM J. on Computing* **6** (1977), 323–350.

[L88] A. Lesk, editor, *Computational Molecular Biology: Sources and Methods for Sequence Analysis*, Oxford University Press, 1988.

[Mc76] E.M. McCreight, A space economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.* **23** (1976), 262–272.

[ST83] D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees. *J. of Computer and System Sciences* **26** (1983), 362–391.

[S88] J. Storer, *Data Compression: Methods and Theory*, Addison-Wesley, 1988.

[W88] M.S. Waterman, Editor, *Mathematical Methods for DNA Sequences*, CRC Press, Inc., Los Angeles, CA, 1988.

[W73] P. Weiner, Linear pattern matching algorithms, *Proc. 14th Symposium on Switching and Automata Theory* (1973), 1–11.

As for the transformation of $F_{D_{i-1}}$ into F_{D_i}, we use the procedure *FTD* which deletes a node from the forest as soon as it is passed to it by *STD*. *FTD* also takes appropriate action when *STD* unmarks a node in the suffix tree, not necessarily to be deleted. Assume that u has just been passed to *FTD*. Let $v = parent(u)$.

procedure *FTD*(suffix tree node u)

A. Case u is a leaf: $cut(\hat{u})$. Since u is a leaf in the suffix tree, $\hat{u}$ is a leaf in the forest of trees. Cutting the edge $(\hat{v}, \hat{u})$ removes $\hat{u}$ from the forest.

B. Case u is an internal node, unmarked but not deleted: $link(\hat{v}, \hat{u})$. u has been unmarked because the pattern $L(u)$ is not anymore in the dictionary. Therefore, $L(w)$ such that $\hat{w} = root(\hat{v})$ is the longest pattern that is a prefix of $L(y)$ for all $\hat{y}$ in the tree rooted at $\hat{u}$, implying that $\hat{u}$ and all of its descendants in the forest must be descendants of $root(\hat{v})$.

C. Case u is an internal node, which has been deleted. Let q be the only child of u in the suffix tree (a leaf is deleted before its parent): $cut(\hat{u})$; $cut(\hat{q})$; $link(\hat{v}, \hat{q})$. In the suffix tree, the edges (v, u) and (u, q) have been transformed into (v, q). The *link* and *cut* operations involving $\hat{u}$, $\hat{v}$ and $\hat{q}$ perform the same change in the forest of trees.

Lemma 8. Each node passed to *FTD* is correctly removed from the forest in $O(\log |D_{i-1}|)$ time.

Proof. The correctness comes from the discussion in the presentation of *FTD*. As for the time analysis, we have a constant number of cases, each consisting of a constant number of *link*, and *cut* operations. Each of those operations takes the claimed time bound since we are using Sleator and Tarjan's dynamic trees [ST83] to maintain our forest and the number of nodes in the forest is at most $O(|D_{i-1}|)$ (as many as the nodes in the suffix tree when the deletion operation is started). □

Theorem 4. The deletion of a pattern of length m in the dictionary at time i requires $O(m \log |D_{i-1}|)$ time in the worst case.

Proof. By Lemma 7 the deletion of a pattern of length m in the suffix tree can be done in $O(m)$ time. Moreover, at most $2m$ new nodes are deleted, and at most one node is unmarked and not deleted. Each of those nodes is placed into the forest in $O(\log |D_{i-1}|)$ time by Lemma 7. □

7. Conclusion

The time complexity of the search is slightly better than that stated in Theorem 2. Assume that the procedure *FINDALL* performed the $root(\hat{u})$ operation k times. Since the trees in which $root(\hat{u})$ is performed are disjoint, the time for *FINDALL* is $O(\log d_1 + \cdots + \log d_k)$, where d_i's are tree sizes. Since $d_1 + \cdots + d_k \leq |D_i|$, the time is $O(k \log(|D_i|/k))$ by the concavity of the log function. For the whole text, the sum of tree sizes involved in $root(\hat{u})$ is bounded by $n|D_i|$. Thus if $tocc \geq n$, the time for the search is bounded by $O(tocc \log(n|D_i|/tocc))$. Therefore, the search takes $O(\max(n, tocc) \log \min(|D_i|, n|D_i|/tocc))$ time.

Our search discovers patterns according to their starting positions in the text (longest pattern first). Thus the search is not on-line, since a pattern that is a prefix of another will be reported

modifying $F_{D_{i-1}}$ accordingly. Again we use two interleaved procedures, STD and FTD, that delete nodes from the suffix tree and the forest, respectively.

We delete the suffixes of $p_j\$_j$ from $T_{D_{i-1}}$ one at a time from longest to shortest. When the k-th suffix is deleted, we first find the locus u of the suffix and use the procedure STD to actually remove nodes from the suffix tree. Let $v = parent(u)$.

procedure STD(suffix tree node v)

A. Case v is the root: delete u. If there remains any pattern in the dictionary after deleting the current pattern, there are at least two children of the root. If u is the only leaf in the suffix tree, this operation will leave the root only, which is the initial state of the suffix tree.
B. Case v is not the root. There are two subcases.
   B1. v has more than two children: delete u. After deleting u, v still has at least two children.
   B2. v has exactly two children: (let w be the sibling of u) delete u; delete v; make w be the child of $parent(v)$ and assign to this edge as label the concatenation of labels on the edges $(parent(v), v), (v, w)$.

The discussion in STD shows that the right nodes are removed from the suffix tree when the k-th suffix of $p_j\$_j$ is deleted. However, in order to prove that the resulting tree is still a suffix tree, we need to show that none of the nodes removed from the suffix tree has a suffix link pointing to it. Let $\alpha\$_j$ be the k-th suffix of $p_j\$_j$. We maintain an invariant: At the beginning of the k-th step, the tree is the suffix tree for the string $S = \cdots p_{j-1}\$_{j-1}\alpha\$_j p_{j+1}\$_{j+1}\cdots$. The invariant holds initially. At the k-th step we delete the suffix $\alpha\$_j$ from the suffix tree. We delete its locus u_k and in case B2 also its parent v_k. Let $\beta = L(v_k)$.

Lemma 6. If we delete v_k, then there is no suffix link pointing to it.

Proof. Assume that an internal node z has its suffix link pointing to v_k. Since v_k has exactly two children, β appears in the string S as a prefix of βa_1 or βa_2 for some characters a_1, a_2. Since u_k, a child of v_k, is a leaf, one of βa_1 and βa_2 (say βa_1) has only one occurrence; i.e., βa_1 is a prefix of $\alpha\$_j$.

By the definition of suffix links, the internal node z is the locus of $b\beta$ for $b \in \Sigma$. z must have exactly two children, otherwise v_k would have more than two children. Since the only occurrence of βa_1 in S is $\cdots\$_{j-1}\beta a_1\cdots$, it cannot be an occurrence of $b\beta$. Therefore, $b\beta$ can appear only as a prefix of $b\beta a_2$, which implies that z cannot be an internal node. □

Lemma 7. $T_{D_{i-1}}$ is transformed into T_{D_i} in $O(|p_j|)$ time, and at most $O(|p_j|)$ nodes are deleted from the suffix tree.

Proof. The correctness comes from the discussion in STD and Lemma 6. Since each call to STD takes constant time, $|p_j| + 1$ calls to STD amount to $O(|p_j|)$ time. The unmarking of suffix tree nodes is simply the reverse of the marking. We need to find the loci of the suffixes of $p_j\$_j$, which can be done by searching the suffix tree with the text $p_j\$_j$. Recall, however, that searching takes $O(|p_j|)$ time. □

A. Case u is a leaf in the suffix tree: $newnode(\hat{u})$; $link(\hat{v}, \hat{u})$. $\hat{v}$ is already in the forest since the parent of a leaf is created before the leaf in the suffix tree.

B. Case u is marked, but not a new node: $cut(\hat{u})$. $\hat{u}$ was not the root of any tree in the forest since u was not marked in the suffix tree. After the marking of u, $\hat{u}$ must become the root of the tree consisting of all its descendants in the forest because $L(u)$ is now a pattern, and for each node w such that $\hat{w}$ is descendant of $\hat{u}$, $L(u)$ is the longest pattern that is prefix of $L(w)$.

C. u is a new internal node: $newnode(\hat{u})$. An edge (v, w) in the suffix tree has been transformed into two edges (v, u) and (u, w). Notice that $\hat{v}$ and $\hat{w}$ and the corresponding edge, if w is not marked, are already in the forest. We have a few subcases.

   C1. Subcase u not marked and $root(\hat{v}) = root(\hat{w})$: $cut(\hat{w})$; $link(\hat{v}, \hat{u})$; $link(\hat{u}, \hat{w})$. If $\hat{v}$ and $\hat{w}$ are in the same tree in the forest, then $\hat{u}$ must also be in that tree ($L(q)$ such that $\hat{q} = root(\hat{v})$ is the longest pattern that is a prefix of $L(u)$). Thus we must transform $(\hat{v}, \hat{w})$ into $(\hat{v}, \hat{u})$ and $(\hat{u}, \hat{w})$.

   C2. Subcase u not marked and $root(\hat{v}) \neq root(\hat{w})$: $link(\hat{v}, \hat{u})$. Since $\hat{v}$ and $\hat{w}$ are not in the same tree in the forest, $root(\hat{w}) = \hat{w}$ (there is no other path in the suffix tree between v and w, except (v, w)) and, by definition, w is marked since it is not the root of the suffix tree. Since u is not marked, $\hat{u}$ cannot be the root of a tree, therefore it must be in the same tree as $\hat{v}$.

   C3. Subcase u marked and $root(\hat{v}) = root(\hat{w})$: $cut(\hat{w})$; $link(\hat{u}, \hat{w})$. The correctness of this operation is analogous to Case B, the only difference being that we have to disconnect w, rather than u, from v.

   C4. Subcase u marked and $root(\hat{v}) \neq root(\hat{w})$: Do nothing. Again, w is marked and since u is also marked, $\hat{u}$ cannot be connected to either $\hat{v}$ or $\hat{w}$, so it forms a tree of its own.

Lemma 5. Each node passed to *FTI* is correctly placed into the forest in $O(\log |D_i|)$ time.

Proof. The correctness comes from the discussion in the presentation of *FTI*. As for the time analysis, we have a constant number of cases, each consisting of a constant number of *link*, *cut*, *root* and *newnode* operations. Each of those operations takes the claimed time bound since we are using Sleator and Tarjan's dynamic trees [ST83] to maintain our forest. The number of nodes in the forest is $O(|D_i|)$ (as many as the nodes in the suffix tree when the insertion is completed). □

Theorem 3. The insertion of a pattern of length m into the dictionary at time i requires $O(m \log |D_i|)$ time in the worst case.

Proof. By Lemma 4 the insertion of a pattern of length m into the suffix tree can be done in $O(m)$ time. At most $2m$ new nodes are created, and there is at most one node that is marked, but not a new node. Each of those nodes is placed into the forest in $O(\log |D_i|)$ time by Lemma 5. □

6. Deletion Algorithm

We delete p_j from $D_{i-1} = \{p_1, \cdots, p_s\}$. By Lemma 2 this can be done by transforming $T_{D_{i-1}}$ into T_{D_i} for $DS_i = p_1\$_1 \cdots p_{j-1}\$_{j-1}p_{j+1}\$_{j+1} \cdots p_s\$_s$ (i.e., by deleting all suffixes of $p_j\$_j$), and by

Now we find all occurrences of the patterns in the text by calling *SEARCH* and *FINDALL* for each position of the text.

Theorem 2. All occurrences of the patterns of dictionary D_i in a text of length n are found in $O((n + tocc)\log|D_i|)$ time, where *tocc* is the total number of such occurrences.

Proof. We make n calls to *SEARCH* and *FINDALL*. During each call, *SEARCH* computes h_j. The correctness of *SEARCH* is analogous to that of *STI* in Section 2.1. By Lemma 3 *FINDALL* correctly reports all occurrences of patterns that are prefixes of h_j.

The time analysis of *SEARCH* is again analogous to that of *STI*. Thus *SEARCH* takes a total, over all calls, of $O(n)$ time. By Lemma 3, each call to *FINDALL* takes $O((occ_j + 1)\log|D_i|)$ time. The sum of such times, over all calls, is bounded by $O((n + tocc)\log|D_i|)$. □

5. Insertion Algorithm

We insert p_j into $D_{i-1} = \{p_1, \cdots, p_s\}$. By Lemma 2, this can be done by transforming $T_{D_{i-1}}$ into T_{D_i} for $DS_i = DS_{i-1}p_j\$_j$ (i.e., by inserting all suffixes of $p_j\$_j$), and by modifying $F_{D_{i-1}}$ accordingly. We use two interleaved procedures that carry out the modifications of the suffix tree (procedure *STI* slightly modified) and of the forest (procedure *FTI*), respectively. *STI* is modified so that it passes a newly created node immediately to *FTI*. It also marks nodes as described below and passes such nodes to *FTI* immediately after its marking. We analyze both procedures independently, since the slowest of the two gives the time bound of the insertion algorithm.

We insert the suffixes of $p_j\$_j$ into $T_{D_{i-1}}$ one at a time from longest to shortest. For each insertion of the suffixes we use *STI*. Since the insertion takes place after the last character $\$_s$ in DS_{i-1}, we must provide the locus of $head_{|DS_{i-1}|}$ in $T_{D_{i-1}}$. Since $\$_s$ does not match anything, the locus of $head_{|DS_{i-1}|}$ is the root of the suffix tree. Notice also that $T_{D_{i-1}}$ trivially satisfies the invariant of *STI*.

We mark suffix tree nodes as follows.

1. After the suffix $p_j\$_j$ is inserted: If there is a node v (not necessarily new) such that $L(v) = p_j$, mark v.
2. After any other suffix $\alpha\$_j$ is inserted: If there is a new node v such that $L(v) = \alpha$ and α is a pattern, mark v.

Since each marking takes constant time, we have the following lemma.

Lemma 4. $T_{D_{i-1}}$ is transformed into T_{D_i} in $O(|p_j|)$ time, and at most $O(|p_j|)$ new nodes are created.

As for the transformation of $F_{D_{i-1}}$ into F_{D_i}, we use the procedure *FTI*, which inserts a node into the forest as soon as it is passed to it by *STI*. Assume that u has just been passed to *FTI*. Let $v = parent(u)$. We have several cases to consider depending on the relationship between u and the other nodes in the suffix tree. We will give along with each case a short proof of its correctness.

procedure *FTI*(suffix tree node u)

C1. If $|\alpha| > |\hat{\beta}|$, then $h_j = L(x)\hat{\beta}$, since the first character of $\alpha - \hat{\beta}$ mismatches the current character of the text as it did in step $j-1$. Stop and return $(x, \hat{\beta})$.

C2. If $|\alpha| = |\hat{\beta}|$, set $y \leftarrow f$. Go to Step D.

D. From the node y, the procedure searches down the suffix tree by scanning the text characters one by one. When the search falls out of the tree (as it must), the last node visited is the contracted locus of h_j. Return the contracted locus v and $\beta = h_j - L(v)$.

SEARCH finds h_j, the longest prefix of $text[j,n]$ that appears as a substring in D_i. We find all patterns, if any, that are prefixes of h_j as follows.

1. If β is not empty and $h_j = p_r$ for some r (this happens only when the extended locus of h_j is the leaf w such that $L(w) = p_r\$_r$), report the occurrence of p_r.
2. Now any pattern that is a prefix of h_j must have its locus v in the suffix tree T_{D_i}, and v is on the path from *clocus* to the root of T_{D_i}. Furthermore, the corresponding node $\hat{v}$ is a root in a tree of F_{D_i} by the definition of marked nodes.

The procedure *FINDALL* in Fig. 1 takes as input $(clocus, \beta)$ of h_j and reports all patterns that are prefixes of h_j.

```
procedure FINDALL(clocus, β)
begin
  report occurrence in Case 1, if any;
  u ← clocus;
  while u ≠ root of T_{D_i} do
  begin
    v̂ ← root(û);
    /* v is a marked node in the suffix tree */
    /* if different from the root */
    if v ≠ root of T_{D_i} then report occurrence;
    u ← parent(v);
  end
end
```

Fig. 1. Procedure *FINDALL*

Lemma 3. Procedure *FINDALL* correctly finds all patterns in D_i that are prefixes of h_j in $O((occ_j + 1)\log|D_i|)$ time, where occ_j is the number of such patterns.

Proof. Since all patterns in the dictionary are distinct, the number of marked nodes on the path from *clocus* to the root of T_{D_i} is either occ_j or $occ_j - 1$ depending on Case 1. Each marked node corresponds to the root of a tree $\mathcal{T}_1$ in the forest, and the parent of a marked node corresponds to a node in a tree $\mathcal{T}_2 \neq \mathcal{T}_1$ in the forest. The transition from $\mathcal{T}_1$ to $\mathcal{T}_2$ is performed via the instruction $v \leftarrow parent(u)$. None of the marked nodes are missed because they are found in the order of decreasing depth.

As for the time analysis, Case 1 takes constant time. The most expensive operation is $root(\hat{v})$ which has a time bound of $\log|D_i|$. We execute it at most $occ_j + 1$ times. □

$ does not match itself (i.e., $\$ \neq \$$). And for each leaf we store the number of different strings of the node (the same string with different $\$_i$'s).

We define the second data structure in terms of T_{D_i}. A node v in T_{D_i} is *marked* if it is the locus of $p_j \in D_i$ for some j. Notice that a pattern p_j is a substring of another pattern if and only if $p_j = L(v)$ for some internal node v. Thus there is a one to one correspondence between marked nodes and patterns that are substrings of other patterns in the dictionary. A pattern p_l that is not a substring of other patterns appears only in the leaf w such that $L(w) = p_l\$_l$.

We partition the suffix tree T_{D_i} into a forest of trees F_{D_i} by deleting edges $(parent(v), v)$ for all marked node v. The following properties hold for F_{D_i}:

1. For each node v in T_{D_i} there is a corresponding node $\hat{v}$ in F_{D_i}.
2. v is marked in T_{D_i} if and only if $\hat{v}$ is the root of some tree in F_{D_i}. (The root of T_{D_i} is an exception.)
3. $\hat{v}$ is in the tree with root $\hat{r}$ in F_{D_i} if and only if, among the marked nodes, $L(r)$ is the longest pattern that is a prefix of $L(v)$.

We implement F_{D_i} by means of the dynamic trees in [ST83] and we assume that such an implementation is available at time i. Notice that F_{D_i} keeps track of the fact that a given pattern may be a substring of other patterns (this information will be useful during the search phase). In the following sections, we show how to efficiently maintain the suffix tree and the forest under insertion and deletion of patterns and how to use them for string matching.

4. Search Algorithm

Let $t[1, n]$ be a text. We want to find all occurrences of patterns of D_i that appear in the text. We can solve this problem by finding the longest prefix, denoted by h_j, of $t[j, n]\$$ that appears as a substring in D_i, for $1 \leq j \leq n$. Then, we can check which patterns are prefixes of h_j.

We find h_j in the order of increasing j. In the suffix tree T_{D_i}, the string h_j is represented by a pair $(clocus, \beta)$:

1. *clocus* is the contracted locus of h_j.
2. $\beta = h_j - L(clocus)$.

Given $(clocus, \beta)$ of h_{j-1}, the procedure *SEARCH* finds $(clocus, \beta)$ of h_j (This procedure is quite similar to procedure *STI*.). *SEARCH* also maintains the current position k of the text (implicit in the procedure below). Initially, *clocus* is the root, β is empty, and $k = 1$.

procedure SEARCH($clocus, \beta$)

A. Case *clocus* is the root: $y \leftarrow root$. Go to Step D.
B. Case *clocus* is not the root: $x \leftarrow SL(clocus)$. Go to Step C.
C. By Lemma 1, starting from node x, there is a path that has β as prefix. That path is traversed as follows. Set $\hat{\beta} \leftarrow \beta$. Let α be the label of the edge from x to its child f such that the first characters of α and $\hat{\beta}$ are equal. If $|\alpha| < |\hat{\beta}|$, set $\hat{\beta} \leftarrow \hat{\beta} - \alpha$ and $x \leftarrow f$ and repeat the label selection with the new values of $\hat{\beta}$ and x until $|\alpha| \geq |\hat{\beta}|$.

- $root(v)$: Return the root of the tree containing v.
- $link(w, v)$: Combine the trees containing v and w by adding the edge (w, v). This operation assumes that v and w are in different trees and that v is a tree root.
- $cut(v)$: Divide the tree containing v into two trees by deleting the edge $(parent(v), v)$. This operation assumes that v is not a tree root.

Sleator and Tarjan [ST83] designed a data structure (called dynamic trees) that supports, among others, the four operations just defined. Each operation can be implemented to take $O(\log k)$ time in the worst case, where k is the number of nodes in the tree involved in the operation.

3. Our Data Structure at Time i

Let $D_i = \{p_1, \ldots, p_s\}$ be the dictionary at time i. We assume that all patterns in the dictionary are distinct. We organize D_i in two data structures: a suffix tree and a forest of trees. Since the suffix tree is defined for a string of symbols, we need to represent our dictionary by means of a single string. Assume that $p_1, \cdots, p_s$ were inserted into the dictionary in that order. We could conceivably just build a suffix tree of the string $p_s p_{s-1} \cdots p_1\$$. Indeed, Weiner's or Chen and Seiferas's algorithm can construct the suffix tree for such a string, even if the patterns must be processed on-line. Since the procedure *STI* builds the suffix tree from longest suffix to shortest, we cannot construct a suffix tree until all the patterns are in the dictionary, and thus the suffix tree would not be available for text scanning until the dictionary was complete. We overcome this difficulty by introducing more special characters. That is, we construct the suffix tree for the string $p_1\$_1 \cdots p_s\$_s$, which we denote by DS_i.

Lemma 2. The suffix tree T for $DS_i = p_1\$_1 \cdots p_s\$_s$ is isomorphic to the compacted trie T' for all suffixes of $p_1\$_1$, all suffixes of $p_2\$_2$,..., and all suffixes of $p_s\$_s$. Furthermore, the two trees are identical except for the labels of the edges incident to leaves.

Proof. Consider an internal node u in T. $L(u)$ does not contain any special character $\$_j$, since it is a unique character in DS_i. Thus $L(u)$ is a substring of DS_i if and only if it is a substring of p_j for some j. This implies that the two trees are isomorphic and the labels of the edges whose endpoints are both internal nodes are the same. It is easy to see that T has a leaf v such that the label of edge $(parent(v), v)$ is $\alpha\$_j p_{j+1}\$_{j+1} \cdots p_s\$_s$ for $\alpha \in \Sigma^*$ if and only if T' has a leaf v' such that the label of $(parent(v'), v')$ is $\alpha\$_j$. □

Lemma 2 will allow us to insert a pattern p_j by inserting all suffixes of $p_j\$_j$ to the suffix tree, and to delete a pattern $p_l\$_l$ by deleting all suffixes of $p_l\$_l$. By Lemma 2 the suffix tree depends on the set of patterns, and not on the order in which patterns were inserted. From now on we refer to the suffix tree for the string DS_i as T_{D_i}.

One disadvantage of using $p_1\$_1 \cdots p_s\$_s$ is that the alphabet grows as we insert more patterns into the dictionary. This is a problem since the linear time construction for suffix trees assumes a finite alphabet, and the $\$_i$s would blow up the alphabet to be arbitrarily large. We can avoid this problem by simulating the special characters by a single special character \$: (1) \$ is not in Σ, (2)

C. In this step the procedure constructs the suffix link of $head_{i-1}$. By Lemma 1, starting from node x, there is a path that has β as prefix. That path is traversed as follows. Set $\hat{\beta} \leftarrow \beta$. Let α be the label of the edge from x to its child f such that the first characters of α and $\hat{\beta}$ are equal. If $|\alpha| < |\hat{\beta}|$, set $\hat{\beta} \leftarrow \hat{\beta} - \alpha$ and $x \leftarrow f$ and repeat the label selection with the new values of $\hat{\beta}$ and x until $|\alpha| \geq |\hat{\beta}|$. This step takes time linear in the number of nodes traversed.

C1. If $|\alpha| > |\hat{\beta}|$, f is the extended locus of $head_{i-1} - S[i-1]$. Create an internal node d such that $L(d) = head_{i-1} - S[i-1]$. Set $SL(v) \leftarrow d$. Create a leaf w such that $L(w) = S[i,m]$, as a child of d. Stop and return d as the locus of $head_i$.

C2. If $|\alpha| = |\hat{\beta}|$, f is the locus of $head_{i-1} - S[i-1]$. Set $SL(v) \leftarrow f$; $y \leftarrow f$. Go to Step D.

D. In this step, the procedure constructs the locus of $head_i$ (notice that $head_i$ is not known yet). By Lemma 1, $head_i = L(y) \cdot \gamma$, for some possibly empty string γ. Therefore, we can start the search from y. The search is guided by the characters of $S[i,p] - L(y)$ (of which γ is prefix) which are scanned one by one from left to right. When the search falls out of the tree (as it must, since \$ is not in the alphabet), the last node visited is the contracted locus of $head_i$. Create an internal node v such that $L(v) = head_i$, if one does not exist. Create a leaf w such that $L(w) = S[i,m]$, as a child of v. Return v as the locus of $head_i$.

Note that during step i a new leaf and at most one new internal node are created.

Theorem 1. [Mc76] Given a string $S[1,m] = a_1 a_2 \cdots a_{m-1}\$$, the suffix tree for S can be correctly built in $O(m)$ time.

Proof. We call the procedure *STI* m times to insert suffixes from longest to shortest. At step 1 the invariant is trivially satisfied. It is also satisfied at the end of each step by Step C of *STI*, which also provides a proof that each suffix is correctly inserted into the tree.

As for the time analysis, we have m distinct calls to *STI*. Each operation in the procedure takes constant time except for Steps C and D. We now show that the time for Steps C and D takes also constant time, amortized over all calls.

As for Step D, the number of characters that must be scanned during step i to locate $head_i$ is given by $|head_i| - |head_{i-1}| + 1$. The sum of such terms, taken over all steps is bounded by m, since $head_1 = head_m$ is empty.

Let res_i be $S[i,m] - L(x)$, the suffix of $S[i,m]$ starting from node x. Notice that for every node f encountered during Step C, there is a nonempty string α which is contained in res_i but not in res_{i+1}. Therefore, the number of nodes visited during Step C of step i is at most $|res_i| - |res_{i+1}|$. The total time over all steps is bounded again by m, since $res_1 = m$ and $res_m = 0$. □

We remark that Weiner [W73] and Chen and Seiferas [CS84] construct slightly different suffix trees from shortest suffix of S to longest.

2.2. Dynamic trees

Let F be a forest of rooted trees, with edges directed away from the root. We are interested in performing the following operations on nodes and edges of F:

- *newnode*(v): Create a new node v, which is also the root of a new tree.

3. There is a node v in T if and only if $L(v)$ is a prefix of s_j for some j.

A *compacted trie* T' is obtained from T by removing internal nodes that have a single child and by concatenating the labels. Now the label of an edge in T' is a nonempty substring of s_j for some j, and it is represented by the starting and ending positions of an occurrence of the substring. Notice that the size of the trie T is $O(|s_1| + \cdots + |s_r|)$, while the size of the compacted trie T' is $O(r)$, since there are at most r leaves and each internal node has degree at least two.

Let $S[1,m] = a_1 a_2 \cdots a_{m-1}\$$ be a string, where the special character \$ is not in the alphabet Σ. The *suffix tree* T_S for the string S is a compacted trie for all suffixes of S. The suffix tree defined by McCreight [Mc76] has one more piece of information:

4. Each internal node u such that $L(u) = a\alpha$, a a character and α a string, has a *suffix link* $SL(u)$ pointing to the node w such that $L(w) = \alpha$ (if α is empty, w is the root of T_S); i.e., $SL(u) = w$.

Notice that since \$ is not in the alphabet, all suffixes of S are distinct and each of them is associated with a leaf of T_S. The suffix tree was proposed by McCreight [Mc76] as a space efficient alternative to Weiner's position tree [W73]. McCreight also gives a very elegant linear-time algorithm for the construction of the suffix tree, of which we present a simplified version. For a unified treatment of position, suffix trees and related data structures, the reader is referred to Chen and Seiferas [CS84].

We need some definitions and notations. Given two strings α and β such that α is a prefix of β, we denote by $\beta - \alpha$ the string obtained by deleting α from β. The *locus* of a string α in the suffix tree T_S is the node associated with α, if any. The *contracted locus* of α is the locus of the longest prefix of α whose locus exists. The *extended locus* of α is the locus of the shortest string that has α as prefix. We define $head_i$ to be the longest prefix of $S[i,m]$ (a suffix of S) which is also a prefix of $S[j,m]$, for some $j < i$. Notice that the locus of $head_i$ always exists. We need the following lemma:

Lemma 1. [Mc76] If $head_{i-1} = a\alpha$ for some character a and some (possibly empty) string α, then α is a prefix of $head_i$.

The algorithm for the construction of the suffix tree for S consists of m steps. At the beginning of step i, each suffix $S[j,m]$, $j < i$, is in the tree and the algorithm inserts $S[i,m]$ at step i. We denote by T_i the tree at the end of step i. Initially (step zero), there is only the root node. The procedure for the insertion of a suffix (referred to as STI) takes as input i (we want to insert $S[i,m]$), and it returns the locus of $head_i$. It maintains an invariant: The locus of $head_i$ in T_i is the only node that could fail to have a suffix link. Let v be the locus of $head_{i-1}$.

procedure $STI(v,i)$

A. Case v is the root (i.e., $head_{i-1}$ is empty): $y \leftarrow root$. Go to Step D.

B1. Case $parent(v)$ is not the root: $x \leftarrow SL(parent(u))$. Let β be the label of edge $(parent(v), v)$. Go to Step C.

B2. Case $parent(v)$ is the root: $x \leftarrow root$; $\beta \leftarrow head_{i-1} - S[i-1]$ (label of edge $(parent(v), v)$ minus its first character). Go to Step C.

to efficiently represent and update the dictionary and as a finite automaton in the style of the automata designed by Knuth, Morris and Pratt [KMP77] and Aho and Corasick [AC75] for fast string matching.

In particular, we show that we can define the suffix tree of a set of patterns so that the tree is independent of the order of the words in the dictionary. We can use such a tree, which can be dynamically updated by a modification of McCreight's algorithm [Mc76], to search a text. However, such a searching scheme requires the off-the-shelf use of Sleator-Tarjan dynamic trees to maintain the prefix relation of the prefixes of dictionary words.

Let \$ be a special character not in the input alphabet Σ and that does not match itself. Assume that $D_i = \{p_1, \cdots, p_s\}$ is the dictionary after the i-th insertion/deletion operation. Since the suffix tree is defined for one string while the dictionary is a set of strings, we have to find a linear representation of the latter: We use the string $x = p_1\$p_2\$\cdots p_s\$$ and build the suffix tree for x. The main advantage of this representation is that the insertion and deletion of p_j from the suffix tree can be done by means of very simple algorithms, dual of each other, in $O(|p_j|)$ time. We also partition the suffix tree into a set of trees to maintain the containment information about the patterns; i.e., which pattern is a substring of another. Such a collection of trees is dynamically changed by means of the data structure and algorithm of Sleator and Tarjan [ST83]. We briefly describe our algorithm and its time performance:

- $insert(p, D_{i-1})$: p is inserted into the suffix tree and the partition is updated in $O(m \log |D_i|)$ worst case time. The size of our data structure is $O(|D_i|)$.
- $delete(p, D_{i-1})$: p is deleted from the suffix tree and the partition is updated in $O(m \log |D_{i-1}|)$ worst case time. The size of our data structure is $O(|D_i|)$.
- $search(t, D_i)$: We show how to use the suffix tree as a finite automaton to process the text. Moreover, through our partition of it into subtrees, we can find all occurrences of patterns in the text in $O((n + tocc) \log |D_i|)$ worst case time.

The paper is organized as follows. In Section 2 we review McCreight's algorithm for the construction of the suffix tree [Mc76] and the dynamic tree operations of Sleator and Tarjan [ST83]. In the following two sections, we show how to suitably combine those two tools to obtain an elegant and efficient algorithm for dynamic dictionary matching.

2. Basic Data Structures

We give a detailed outline of suffix tree constructions since we will be modifying this algorithm. We will use dynamic trees directly and will therefore only outline their properties.

2.1. Suffix tree

A *trie* T [K73] for a set of strings $\{s_1, \cdots, s_r\}$ is a rooted tree that satisfies the following conditions:

1. Each edge is labeled with a character, and no two sibling edges have the same label (character).
2. Each node v is associated with a string, denoted by $L(v)$, the one obtained by concatenating the labels on the path from the root to v.

1. Introduction

String processing algorithms have been an active area of research in computer science for quite some time. Much of this study has been motivated and has found applications in many diverse fields ranging from storage and transmission of information [S88], compiler construction technology [ASU86] to molecular biology [W88]. In recent years, interest in this area has grown even further due to the computational needs of molecular biology [D88,L88].

We consider the *dynamic dictionary matching problem*. We are given a set of pattern strings $D = \{p_1, \cdots, p_s\}$ (the dictionary) that can change over time; i.e., we can insert a new pattern into D or delete a pattern from D. Moreover, given a text string $t[1,n]$, we must be able to find all occurrences of any pattern of the dictionary in the text. More precisely, let D_0 be the empty dictionary. We are interested in performing any sequence of the following operations:

(1) $insert(p, D_{i-1})$: Insert pattern $p[1,m]$ into the dictionary D_{i-1}. D_i is the dictionary after the operation.
(2) $delete(p, D_{i-1})$: Delete pattern $p[1,m]$ from the dictionary D_{i-1}. D_i is the dictionary after the operation.
(3) $search(t, D_i)$: Search text $t[1,n]$ for all occurrences of the patterns of dictionary D_i.

Efficient algorithms for this problem have applications to bibliographic database searches and to molecular biology, as discussed in [AC75,H90].

In its *static* version (i.e., $D_0 = D$ is a non-empty set of strings, and no insertion or deletion of patterns from the dictionary is allowed) this problem is a generalization of the well known *string matching problem*: Given a pattern string and a text string, find all occurrences of the pattern in the text. For the static dictionary matching problem, two algorithms are known: one due to Aho and Corasick [AC75] (AC for short), which can be seen as a generalization of the Knuth-Morris-Pratt string matching algorithm [KMP77], and the other one due to Commentz-Walter [C79] (CW for short), which can be seen as a generalization of the Boyer-Moore algorithm [BM77]. Both AC and CW have preprocessing phases in which graphs are built from the dictionary D for later use, and search phases in which text positions are checked in increasing order for occurrences of patterns. The time complexity of the preprocessing of both algorithms is $O(|D|)$, where $|D|$ denotes the sum of the lengths of the patterns in D. The time complexity of the AC search algorithm is $O(n+tocc)$, where n is the length of the text and $tocc$ is the total number of occurrences of patterns in the text, while that of the CW search algorithm is $O(n|D|)$ in the worst case. For both algorithms, once the preprocessing is done, we can use the search phase for many texts at no extra penalty in running time. Unfortunately, if we want to insert or delete a pattern, it seems that there is no better way than doing the preprocessing all over again, implying that the cost of an insertion/deletion operation would be linear in the size of the dictionary. Thus neither of the static algorithms can be extended to deal efficiently with the dynamic version of the problem.

We present an algorithm for dynamic dictionary matching whose time performance per operation compares well with a dynamization of any of the static algorithms. This work is an extension of the algorithm and data structures presented in [AF91] and [GGP91]. Moreover, we show how to use the suffix tree (see [Mc76] for a definition of this data structure) both as a data structure

Abstract

We consider the dynamic dictionary matching problem. We are given a set of pattern strings (the dictionary) that can change over time; that is, we can insert a new pattern into the dictionary or delete a pattern from it. Moreover, given a text string, we must be able to find all occurrences of any pattern of the dictionary in the text.

Let D_0 be the empty dictionary. We present an algorithm that performs any sequence of the following operations in the given time bounds:

(1) $insert(p, D_{i-1})$: Insert pattern $p[1,m]$ into the dictionary D_{i-1}. D_i is the dictionary after the operation. The time complexity is $O(m \log |D_i|)$.

(2) $delete(p, D_{i-1})$: Delete pattern $p[1,m]$ from the dictionary D_{i-1}. D_i is the dictionary after the operation. The time complexity is $O(m \log |D_{i-1}|)$.

(3) $search(t, D_i)$: Search text $t[1,n]$ for all occurrences of the patterns of dictionary D_i. The time complexity is $O((n + tocc) \log |D_i|)$, where $tocc$ is the total number of occurrences of patterns in the text.

Dynamic Dictionary Matching

Martin Farach

DIMACS
Box 1179
Rutgers University
Piscataway, NJ 08855

farach@dimacs.rutgers.edu

Dynamic Dictionary Matching

June 1993

Amihood Amir[1]

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA

Martin Farach[2]

DIMACS Box 1179
Rutgers University
Piscataway, NJ 08855, USA

Zvi Galil[3]

Department of Computer Science
Columbia University, NY 10027, USA and
Tel-Aviv University, Tel-Aviv, Israel

Raffaele Giancarlo[4]

AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

Kunsoo Park

Department of Computing
King's College London
Strand, London WC2R 2LS, UK

[1] Partially supported by NSF Grant IRI-9013055

[2] Supported by DIMACS under NSF Contract STC-8809648

[3] Partially supported by NSF Grant CCR-90-14605

[4] On leave from University of Palermo, Italy