

Dynamic Pattern Matching: The World of Tries and Range Queries?

6.854 Final Project

Alexandr Andoni
andoni@mit.edu

Cristian Cadar
cristic@mit.edu

December 10, 2003

Abstract

In this paper, we consider the problem of finding all the occurrences of a pattern P in a text T , where search queries can be intertwined with insertions and deletions of characters in T . Our approach first finds the occurrences of P which are not affected by the modifications performed on T , and then finds those occurrences which are introduced by the respective modifications. A restriction imposed on input is that any search query must be of size at most the minimum distance between two consecutive modifications. We propose a solution that spends $O(P \log k \log \log k + \log T + occ)$ for a search query, and $O(\log k \log \log k + P^{max} \cdot \log T)$ for an update of T , where k is the total number of modifications performed on the original text, and P^{max} is the length of the longest search pattern P so far. This paper is an extension of our previous work in [3].

1 Introduction

The static variant of the pattern matching problem can be formulated as follows. Given a text T and a pattern P , we need to find all the occurrences of P in T , if any such occurrences exist. The alphabet of T and P is Σ and is of constant size.

The static problem is a well-studied problem, for which an optimal solution has been known since 1973 [1]. This solution is based on suffix trees and uses $O(T)$ time to preprocess the text $|T|$ and $O(|P| + occ)$ to perform search, where occ represents the number of occurrences of P in T .

In the dynamic case, searches can be intercalated with online modifications (updates) of the text T . Two basic types of modifications of T are allowed:

1. Delete a character at position j ;
2. Insert a character c after position j .

Note that a replacement of a character at position j with character c can be modeled as a deletion and an insertion, and therefore we do not consider it as a stand-alone modification.

There are several proposed solutions to dynamic pattern matching (also known as dynamic text indexing) in the literature. A solution to the dynamic pattern problem depends on two main parameters: the search time and the update time, which in turn are expressed in terms of the length of the text $|T|$, the length of the pattern $|P|$, the total number of modifications performed on the original text k , and the number of occurrences occ .

Gu, Farach, and Beigel proposed a solution with $O(|P| + occ \cdot \log k + k \log |P|)$ search time and $O(\log |T|)$ update time [6]. Amir, Lewenstein, and Schäffer proposed a solution that performs k updates and l search queries in $O(k|P^{max}| \log^2 n + l \cdot |P^{max}| + occ)$ time, where P^{max} is the longest query pattern [7]. In 1998, Ferragina and Grossi presented the first solution that achieves the optimal search time of $O(|P| + occ)$, with an update time of $O(\sqrt{|T|})$ [8]. In 1999, Ferragina and Grossi proposed an alternative solution whose running times for the search and the update stages are $O(|P| + occ + k \log |P| + \log |T|)$ and $O(\log |T|)$ respectively [9]. Sahinalp and Vishkin proposed another solution with optimal $O(|P| + occ)$ search time and an update time of $O(\log^3 |T|)$ [10]. Finally, Alstrup, Brodal, and Rauhe presented a solution that achieves $O(|P| + occ + \log |T| \log \log |T|)$ search time and $O(\log^2 |T| \log \log |T| \log^* |T|)$ update time [11].

In this paper, we achieve a search time of $O(|P| \log k \log \log k + \log |T| + occ)$ and an update time of $O(\log k \log \log k + |P^{max}| \cdot \log |T|)$, where P^{max} is the longest search query so far, under a restriction. The restriction is that the modifications are “far apart”, meaning that any two modifications are at a distance of at least the size of the search query. Reconstructing all the data structures whenever k reaches $\Theta(|T|)$, we obtain the bounds of $O(|P| \log |T| \log \log |T| + occ)$ and $O(\log |T| \log \log |T| + |P^{max}| \cdot \log |T|)$.

Our solution is based on ideas from a previous class project submitted in May 2003 [3]. The algorithm is divided into two steps and [3] discussed one of them. To one of the two steps, [3] proposed three solutions, under the same restriction of having any two modifications at distance at least $|P|$. The first solution proposed in that paper has $O(k|P|)$ search time and $O(\log k)$ update time, the second solution has $O(|P|^2)$ search time and $O(|T|^{2/3})$ update time, and the third solution has $O(|P| \log k)$ search time, while maintaining $O(|T|^{2/3})$ update time. In this paper, we focus on improving the third solution proposed in [3]. We propose a solution with a slightly worse search time, $O(|P| \log k \log \log k)$ instead of $O(|P| \log k)$, but with a much better update time, $O(\log |T| \log \log |T|)$ instead of $O(|T|^{2/3})$ time. This new result is achieved by reducing the problem of intersecting the set of leafs covered by two nodes in a trie to the problem of dynamic 2D range queries [4]. We also solve the second step of the algorithm, the one that is not solved in [3].

Further, to ease the presentation we consider only deletions – the insertions are treated similarly (specifically, one can treat the insertion pretty similarly to a deletion in our solution).

Let k be the number of deletions performed on the original text T , and let j_1, j_2, \dots, j_k be the positions of these modifications in increasing order, relative to the original text T . Let's denote by T' the current (modified) text.

Our approach divides the search algorithm in two steps.

The first step is to find the occurrences of P that strand one of the positions j_1, j_2, \dots, j_k . Below, we show how to perform this stage in $O(|P| \log k \log \log k + occ_1)$, where occ_1 is the number of reported matches. The update stage takes $O(\log k \log \log k)$ amortized time.

The second step uses the suffix tree as well as some 1D range query data structures to find all the occurrences of P that do not strand any of the positions j_1, j_2, \dots, j_k in T . The complexity of this step is $O(|P| + \log |T| + occ)$ for search and $O(|P^{max}| \cdot \log |T|)$ for update.

2 Notations

This sections presents the notations used in the rest of the paper.

For a string α , we have:

- $|\alpha|$ is the length of a string α .
- $\alpha[i]$ is the i^{th} character of α .
- $\alpha[i : j]$ is the substring $\alpha[i]\alpha[i + 1]\dots\alpha[j]$ of α ; $\alpha[: j] = \alpha[1]\alpha[2] \dots \alpha[j]$; $\alpha[i :] = \alpha[i]\alpha[i + 1] \dots \alpha[|\alpha|]$.
- $\alpha \preceq \beta$ if α is a prefix of β or $\alpha = \beta$.
- $I(\alpha)$ is the inverse string of α , i.e., $I(\alpha) = \alpha[|\alpha|] \dots \alpha[1]$.

For a node N in a trie, we have:

- $str(N)$ is the string obtained by traversing the trie from the root to the node N .
- $dist(N) = |str(N)|$.
- $leafs(N)$ represents the set of the leafs in the subtree rooted at N . If the trie contains suffixes of some text, we will usually mean by $leafs(N)$ the start indexes of the suffixes that are in N 's subtree.

Finally, we keep the following fields in a node N of a suffix tree:

- $N.dist$ represents $dist(N)$.
- $N.link$ represents the link pointer of the node N – a link of the node N is defined to be the node M such that $str(N) = c \circ str(M)$, where c is a character and \circ is the concatenation operation.
- $N.child[c]$ is the pointer to the child of N such that the string corresponding to the edge $(N, N.child[c])$ starts with the character c ; $N.child[c] = nil$ if no such child exists.

3 Occurrences of P that strand exactly one modification

In this section, we show how to find all the occurrences of the pattern P in the text T' that strand exactly one deletion. More precisely, in this part of the algorithm, we find all positions p (in the old text T), such that: 1) exactly one $j_i \in \{p + 1 \dots, p + |P| - 2\}$; and 2) $T_p T_{p+1} \dots T_{j_i-1} T_{j_i+1} \dots T_{p+|P|} = P$. Further, when we refer to occurrences of P , we mean these kind of occurrences only. Also, in this section, we assume that any two modifications are at distance at least $|P|$ from each other, that is $j_i - j_{i-1} - 1 \geq |P|$ for all $i = 2 \dots k$.

Note that in this step of the algorithm, as well as in the second, reported occurrences are indexed in the old text T , while we would need the indices in the new (modified) text T' . We discuss this issue in the section 5.3.

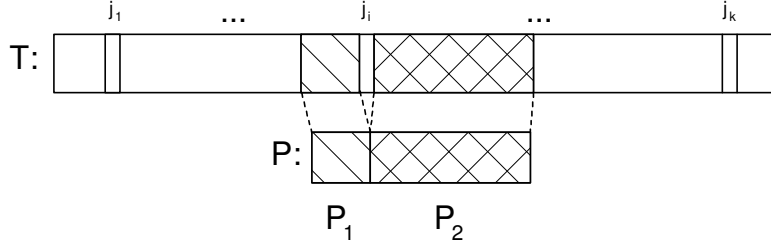


Figure 1: Decomposing P into $P = P_1P_2$, and matching P_1 and P_2 with the corresponding portions of T .

3.1 Overview of the algorithm for search query

The main idea is to try all possible decompositions of P into two parts P_1 and P_2 ($|P_1|, |P_2| \geq 1$) and then to try to match these parts with the portions in T before and after each modification, as shown in Figure 1.

The occurrences of P can be found in the following way. Let $Suf(P_2)$ be the set of indexes j_i such that $T[j_i + 1 : j_i + |P_2|] = P_2$. Let $Pre(P_1)$ be the set of all indexes j_i such that $T[j_i - |P_1| : j_i - 1] = P_1$. Then the occurrences that we look for are $\cup_{P=P_1P_2} \{x - |P_1| \mid x \in Suf(P_2) \cap Pre(P_1)\}$.

To compute $Suf(P_2)$ and $Pre(P_1)$ efficiently, we construct two additional data structures, a trie of suffixes TS and a trie of prefixes TP . For each modification at position j_i , we define the suffix s^i of that modification as the string containing all the characters of T starting from position $j_i + 1$, and we define the prefix p^i of that modification as the string containing all characters of T from the beginning of T until position $j_i - 1$. Figure 2 shows graphically the prefix p^i and the suffix s^i for each modification.

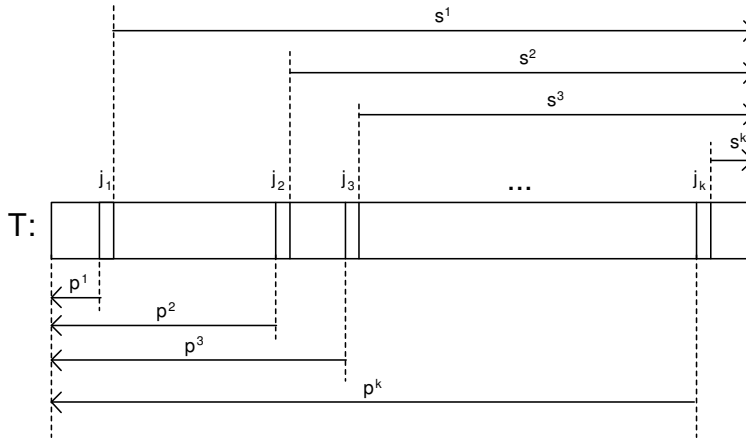


Figure 2: The prefixes and the suffixes of the k modifications.

The trie of suffixes TS contains the suffixes of all modifications (in compacted form), and the trie of prefixes TP contains the prefixes of all modifications, in reverse order (in compacted form). Note that a search of a string s in any of these structures takes $O(|s|)$ time.

Intuitively, the algorithm works as follows. For each $i = 1 \dots |P| - 1$, let $P = P_1^i P_2^i$, where $|P_1^i| = i$. Let N_1^i be a node of TP such that $I(P_1^i) \preceq str(N_1^i)$ and N_2^i be a node of TS such that $P_2^i \preceq str(N_2^i)$. Then, the subtree rooted at N_1^i contains $Pre(P_1^i)$ and the subtree rooted at

N_2^i contains $Suf(P_2^i)$. Thus, once we know $N_1^i, N_2^i, i = 1 \dots |P| - 1$, we only need to compute $Pre(P_1) \cap Suf(P_2) = leafs(N_1^i) \cap leafs(N_2^i)$ to find all the occurrences.

Further, we show how to compute nodes N_1^i and $N_2^i, i = 1 \dots |P| - 1$ in $O(P)$, and then how to compute $Pre(P_1^i) \cap Suf(P_2^i)$ efficiently.

3.2 Computing N_1^i and N_2^i

We show how to compute all $N_2^i, i = 1 \dots |P| - 1$, N_1^i being computed in a similar way. To find the node N_2^i , we first find the node corresponding to N_2^i in the suffix tree *SuffixTree*, which we call the *mirror node* of N_2^i . More precisely, the mirror node of a node N in TS is the node M in *SuffixTree* such that $str(M) = str(N)$. Note that each node N in TS has a mirror node because the suffixes contained by TS are just a subset of the suffixes contained by *SuffixTree*. For a node N in TS , we denote by $Mirror(N)$ the mirror node M in *SuffixTree*. Also, for a node M in *SuffixTree*, we denote by $Mirror^{-1}(M)$ the node N in TS whose mirror node is M (if one such exists). For simplicity, we refer to $Mirror^{-1}(M)$ as the *anti-mirror* of M .

For each node in TS we keep a pointer to its mirror node in *SuffixTree*. Also, for each node in *SuffixTree* we remember whether it is a mirror node, and, if it is, we keep a pointer to its anti-mirror.

To compute N_2^i , we first compute the node M_2^i in *SuffixTree*, where M_2^i is the node in *SuffixTree* with the minimum $dist(M_2^i)$ such that $str(P_2^i) \preceq str(M_2^i)$. Then, N_2^i is the anti-mirror node of the closest descendant of M_2^i that is also a mirror node (if there is one).

We compute M_2^i as follows. First, we find M_2^1 and then, using link pointers in the suffix tree, compute successively $M_2^i, i = 2 \dots |P| - 1$. One problem occurs when no M_2^i exist (no suffixes start with $P[2 :]$). However, note that we can still keep at each step the node M in the suffix tree with the maximum $dist(M_2^i)$ such that $str(P_2^i) \preceq str(M_2^i)$.

The exact algorithm is described in figure 3. M is always the node in the suffix tree whose $str(M)$ matches the most of the current P_2^i ; l is the last position such that $P[i + 1 : l]$ still matches to (the prefix of) some suffix in the *SuffixTree*. Since $i + dist(M)$ increases from 1 to at most $|P|$ and l from 0 to at most $|P|$, the total running time for the algorithm is $O(|P|)$.

1. $M :=$ node in *SuffixTree* such that $str(M) \preceq P_2^1$ and $dist(M)$ is maximum.
 $l := 1 + \#$ of characters of P_2^1 that match some suffix in *SuffixTree* ($M.dist \leq l - 1 \leq |P|$).
2. if $M.dist = |P| - 1$, then $M_2^1 = M$.
3. if $M.dist < |P| - 1$, then $M_2^1 = M.child[P[1 + M.dist + 1]]$.
4. for $i = 2 \dots |P| - 1$:
5. $M = M.link$
6. if $i + M.dist = |P|$, then $M_2^i := M$ and continue.
7. while $i + M.dist < |P|$ and $M.child[P[i + M.dist + 1]] \neq nil$ and $M.child[P[i + M.dist + 1]].dist \geq l - i$
 $M := M.child[P[i + M.dist + 1]]$.
8. if $i + M.dist = l$ and $l < |P|$ and $M.child[P[l + 1]] \neq nil$, then:
9. Continue matching string $P[l + 1 :]$ beginning with the node M in the suffix tree (at this moment, $P[i + 1 : l] = str(M)$).
 Let $M :=$ node in *SuffixTree* such that $str(M) \preceq P_2^i$ and $dist(M)$ is maximum.
 Let l be the last position in P such that $P_2^i[i + 1 : l]$ still matches in the suffix tree. l will be $M.dist \leq l \leq |P|$.
10. if $i + M.dist = |P|$, then $M_2^i = M$.
11. if $i + M.dist < |P|$, then $M_2^i = M.child[P[i + M.dist + 1]]$.

Figure 3: Algorithm for computing $M_2^i, i = 1 \dots |P| - 1$.

Finally, we show how to compute N_2^i from M_2^i (suppose $M_2^i \neq nil$). First, we compute the mirror node of the parent of N_2^i , which is in fact the closest ancestor of M_2^i that is also a mirror node (the ancestor must be strictly above M_2^i). Call this node A_2^i . We can reformulate

the problem of finding A_2^i from M_2^i in the following way: given a tree (*SuffixTree*) in which some nodes are marked (mirror nodes), and given a query node M , we need to find the closest ancestor of M that is marked. This problem is known in the literature as the marked ancestor problem [2]. In the update stage, we need to mark some of the nodes as mirrors, meaning that we need to maintain *mark* operations (but not *unmark* operations). Thus, we can do query in $O(1)$ time, and update (mark a node as mirror) in $O(1)$ amortized time [2]. This means we can find A_2^i in $O(1)$ time from M_2^i . Knowing A_2^i , we find the node $B_2^i = \text{Mirror}^{-1}(A_2^i)$. If $B_2^i.\text{child}[P[i + A_2^i.\text{dist} + 1]] \neq \text{nil}$ and $\text{Mirror}(B_2^i.\text{child}[P[i + A_2^i.\text{dist} + 1]])$ is in M_2^i 's subtree (or is equal to M_2^i), then $N_2^i = B_2^i.\text{child}[P[i + A_2^i.\text{dist} + 1]]$. Note that we can compute whether a node is the subtree of some other node in constant time, using, for example, precalculated DFS numbers or static LCAs.

3.3 Computing $\text{Pre}(P_1) \cap \text{Suf}(P_2)$

Note that, at this moment, if we use the same approach as that described in [3] in section 3, we obtain search time of $O(|P| + \text{occ})$ (by having a precomputed table $\text{table}[N_1^i, N_2^i]$ that hold the intersection $\text{table}[N_1^i, N_2^i] = \text{leafs}(N_1^i) \cap \text{leafs}(N_2^i) = \text{Pre}(P_1) \cap \text{Suf}(P_2)$ precomputed). Unfortunately, the update then takes $O(k^2)$ time.

In this section, we reduce the time for update to $O(\log k \log \log k)$ at the expense of increasing the search time to $O(|P| \cdot \log k \log \log k + \text{occ})$. Further, we show to reduce the problem of computing $\text{leafs}(N_1^i) \cap \text{leafs}(N_2^i)$ to dynamic 2D range queries.

We define the function $x(\cdot) : \{1 \dots |T|\} \rightarrow \{1 \dots |T|\}$ as follows. Suppose we traverse the *SuffixTree* in inorder, and $s_1 s_2 \dots s_{|T|}$ is the order in which we visit the leafs: s_1 is the index of the suffix that is visited first, and so forth. Note that $s_1 \dots s_{|T|}$ corresponds also to the order of suffixes when the suffixes are sorted in the lexicographical order. Then, for a suffix s_i (suffix $T[s_i :]$), $x(s_i) = i$. Intuitively, for a suffix s , we define its x function to be the position of s in the lexicographical ordering of all suffixes.

The function $y(\cdot) : \{1 \dots |T|\} \rightarrow \{1 \dots |T|\}$ is defined similarly. If $p_1 \dots p_{|T|}$ is the order of leafs when traversing *PrefixTree* in inorder, then define $x(p_i) = i$ (prefix p_i is the prefix $T[: p_i]$). Intuitively, for a prefix p , we define its y function to be the position of p in the lexicographical ordering of all prefixes.

Suppose, we want to compute $\text{leafs}(N_1^i) \cap \text{leafs}(N_2^i)$, $N_1^i \neq \text{nil}$ and $N_2^i \neq \text{nil}$ (if one of the pointers is *nil*, then clearly $\text{Pre}(P_1) \cap \text{Suf}(P_2) = \emptyset$). Let Y^i be the mirror node of N_1^i (in *PrefixTree*) and X^i be the mirror node of N_2^i (in *SuffixTree*). Let $x_1 = \min_{l \in \text{leafs}(X^i)}\{x(l)\}$ and $x_2 = \min_{l \in \text{leafs}(X^i)}\{x(l)\}$; in other words X^i comprises in its subtree leafs with indeces from x_1 to x_2 . Define similarly y_1, y_2 : $y_1 = \min_{l \in \text{leafs}(Y^i)}\{y(l)\}$ and $y_2 = \min_{l \in \text{leafs}(Y^i)}\{y(l)\}$. Note that x_1, x_2, y_1, y_2 can be precomputed in each node X^i, Y^i during preprocessing in linear time.

Further, consider we have a data structure *2DRQ* that implements dynamic 2D range queries (permitted operations are *insert*(x, y) and *range-query*(x_1, x_2, y_1, y_2)). Suppose *2DRQ* contains the point $(x(j+1), y(j-1))$ for each deletion at position j . Then, *range-query*(x_1, x_2, y_1, y_2) (with x_1, x_2, y_1, y_2 defined above) returns all points $(x(j+1), y(j-1))$ such that $x_1 \leq x(j+1) \leq x_2$ and $y_1 \leq y(j-1) \leq y_2$ and j is the position of a deletion. Let J be the set of j 's such that the above *range-query* returns the point $(x(j+1), y(j-1))$. J represents exactly all deletions j such that the suffix $j+1$ is in the subtree of X^i and prefix $j-1$ is in the subtree of Y^i .

Now, for all suffixes $j+1$ such that there is a deletion at position j , suffix $j+1$ is in X^i 's subtree if and only if suffix $j+1$ starts with P_2 (because all suffixes $j+1$, where j is a deletion position, that start with P_2 are in N_2^i 's subtree and, subsequently, in X^i 's subtree). Similarly, with all prefixes $j-1$. Thus, we can conclude that $J = \text{Pre}(P_1) \cap \text{Suf}(P_2)$.

Resuming, if we have the *2DRQ* data structure, we can issue a range query on (x_1, x_2, y_1, y_2) , and for all points $x(j+1), y(j-1)$ find the corresponding j 's. Computing j from $(x(j+1), y(j-1))$ can be done easily if we precompute x^{-1} in the precomputation stage (it is easily done in linear time). Once we have js , we can compute the starting position of the match of each of the occurrences $(j - |P_1^i|)$.

To maintain the $2DRQ$ data structure, we need to add the point $(x(j+1), y(j-1))$ to $2DRQ$ whenever we do a delete at position j .

There is a $2DRQ$ data structure that does range queries and inserts both in $O(\log k \log \log k)$ time [4] (where k is the number of points in the data structure, the number of deletions so far in our case). This gives total search time of $O(|P| \log k \log \log k + occ)$ and a term of $O(\log k \log \log k)$ in update operation for inserting of $(x(j+1), y(j-1))$.

3.4 Update operation

In the update operation, we need to do mainly two things: 1) update the $2DRQ$ data structure as described in the section 3.3; 2) update the trie of suffixes TS and the trie of prefixes TP (updating, as well, the pointers to mirror nodes and vice-versa).

Suppose we do a deletion at position j_{k+1} . Further we describe how to update TS and TP accordingly in $O(1)$ time. We need to insert the suffix $s^{k+1} = T[j_{k+1} + 1 :]$ into TS , and the prefix $p^{k+1} = I(T[: j_{k+1} - 1])$ into TP .

We first describe the insertion of the suffix s^{k+1} into TS , the insertion of the prefix p^{k+1} into TP being similar. The insertion of the string s^{k+1} into TS implies adding one or two new nodes in the trie TS . Let $\alpha = LCP(s^{k+1}, s^{i^*})$ such that $|LCP(s^{k+1}, s^{i^*})|$ is maximum, $i^* = 1 \dots k$ (s^{i^*} is the “closest” suffix to s^{k+1} already in TS). If there is a node P in TS such that $str(P) = \alpha$, then only one node, S^{k+1} , needs to be added to TS as a leaf of P . Otherwise, there are nodes P and Q , Q is a child of P , such that $str(P) \preceq \alpha$, $\alpha \preceq str(Q)$ and $dist(P) < |\alpha|$, $dist(Q) > |\alpha|$; in this case, the edge (P, Q) needs to be split to insert a new node \tilde{S}^{k+1} ; leaf S^{k+1} is the child of \tilde{S}^{k+1} . See a graphical illustration of the last case in figure 4. There is also a third case, when $|\alpha| = 0$; this case is easy to detect and process in $O(1)$ time.

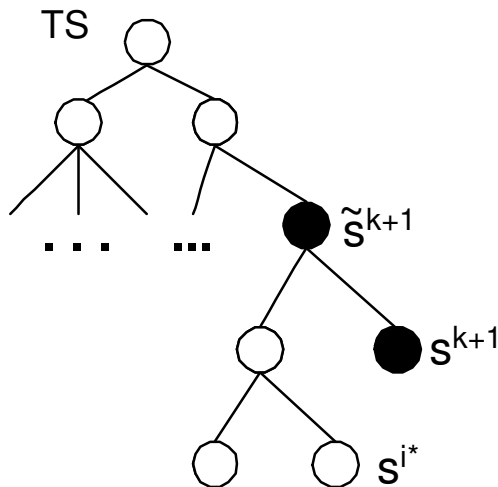


Figure 4: Inserting nodes \tilde{S}^{k+1} and S^{k+1} into TS .

We find the node P in TS using the same technique as the one used to find N_2^i described in section 3.2. Specifically, let R be the leaf node in $SuffixTree$ that corresponds to the suffix $s^{k+1} = T[j_{k+1} + 1 :]$. Using the same data structure for solving the marked ancestor problem as in section 3.2, we can find the closest mirror node P' in ST that is an ancestor of R . Next, we can observe that $P = Mirror^{-1}(P')$. Thus, we find the node P . We check whether S^{k+1} should be added as a child of P or an edge (P, Q) must be split to first add \tilde{S}^{k+1} and then S^{k+1} as a child of S^{k+1} . All these operations take $O(1)$.

Next, we need to setup the links to mirror nodes and viceversa for nodes S^{k+1} and \tilde{S}^{k+1} . Finding the mirror node of S^{k+1} is trivial – it’s just the node R above. Finding the mirror node of \tilde{S}^{k+1} is a little trickier: $Mirror(\tilde{S}^{k+1}) = LCA(T, Q)$. We can compute LCA in $SuffixTree$ in $O(1)$ (for example, [5]). Setting pointers $Mirror^{-1}$ is trivial after we found the mirror nodes. Note that, here, we also need to do *mark* operation of the new mirror nodes in the data structure supporting marked ancestor problem. As mentioned earlier, *mark* operation takes $O(1)$ time amortized time and $O\left(\frac{\log k}{\log \log k}\right)$ worst case time [2].

Thus, the time required to update the tries TS and TP is $O(1)$, and the total time for update operation is $O(\log k \log \log k)$.

4 Occurrences of P that strand zero modifications

In this section, we show how to find the occurrences of P in the text T' that do not strand any of the deletions j_i . More precisely, in this part of the algorithm, we find all positions p in the text T , such that $T[p : p + |P| - 1] = P$ and $j_i \notin \{p \dots p + |P| - 1\}$ for all $i = 1 \dots k$. Further, in this section, when we refer to occurrences, we mean this kind of occurrences only (unless otherwise noted). We still assume that $j_i - j_{i-1} - 1 \geq |P|$ for all $i = 2 \dots k$.

A straight-forward solution could do the following. Match the pattern P in the suffix tree: let N be the node such that $P \preceq str(N)$ and $dist(N)$ is minimal. Then the occurrences we need to report are in $leafs(N)$. Unfortunately, there could be many “false” occurrences – occurrences of P that were present in T , but are not present anymore in T' due to the modifications. There could potentially be as many as $O(k|P|)$ of them, therefore, just skimming through $leafs(N)$ and dropping the invalidated occurrences is not acceptable.

Below, we propose a solution that achieves $O(|P| + \log |P^{max}| + \log |T| + occ)$ amortized search time, where P^{max} is the longest search query seen so far. The update takes $O(|P^{max}| \cdot \log |T|)$ amortized time.

To achieve these bounds, we have special data structures that hold the indexes $[j_i - y_1^\delta, j_i - y_2^\delta]$, $i = 1 \dots k$, for each pair (y_1^δ, y_2^δ) , where $[y_1^\delta, y_2^\delta]$ is one of the diadic intervals covering $[0, P^{max} - 1]$. Each of the data structures supports 1D range queries; the range-queries are according to function $x(\cdot)$ defined in section 3.3. We assume we have a data structure that can: 1) hold a set of points $\subseteq \{1 \dots |T|\}$; 2) support insertions of a point in $O(1)$; 3) support range-query of the form $[x_1, x_2]$ in $O(|output|)$; 4) has preprocessing time $O(1)$. We call such a data structure $1DRQ - I$ and describe how to construct a $1DRQ - I$ data structure later. We also have a $1DRQ$ data structure that is another 1D dynamic range-query data structure, but that supports deletions in addition to insertions; all operations are permitted to take $O(\log |T|)$ time; processing time may take $O(n)$ time. Note that $1DRQ$ may be implemented as a balanced binary search tree.

4.1 Data structures used

Let J be the set of all occurrences of the pattern P in the old text T . Then, all occurrences from the set $FALSE = J \cap (\cup_{i=1 \dots k} [j_i - |P| + 1, j_i])$ are “false” occurrences in the new text T' and should not be reported; call $j \in FALSE$ *invalid*. Call all other j ’s *valid*; the valid occurrence positions are exactly the ones that should be reported in this stage of the algorithm. Below, we will also identify a suffix that starts at position j with the position j itself.

Let $L = \min \{P^{max}, \min_{i=1 \dots k} j_i - j_{i-1} - 1\}$, that is L is the minimum between the longest search pattern seen so far and the length of the a search pattern that can be issued at the current moment or later. Note that for any $j \in [1, |T|]$, we know for sure that j is valid if $j_i - j \geq L$ or $j_i - j < 0$ for all $i = 1 \dots k$. All other j ’s can be invalid for some values of P (of length $[1, L]$).

In our approach, we hold a $1DRQ$ data structure V that contains all the valid suffixes. All the other suffixes (that could be invalid) are stored in $O(L)$ $1DRQ - I$ data structures.

Further, we describe how we store invalid suffixes in the $1DRQ - I$ data structures. Suppose intervals $[y_1^\delta, y_2^\delta]$ are diadic intervals that cover the interval $[0, L - 1]$; for $\delta = (\lambda, \chi)$, $0 \leq \lambda \leq$

$\lfloor \log_2 L \rfloor, 0 \leq \chi \leq \lfloor L/2^\lambda \rfloor - 1$, we have $[y_1^\delta, y_2^\delta] = [\chi 2^\lambda, (\chi + 1)2^\lambda]$. Then, for each interval $[y_1^\delta, y_2^\delta]$, we have a $1DRQ - I$ data structure INV^δ that holds the suffixes $\cup_{i=1\dots k} [j_i - y_2^\delta, j_i - y_1^\delta]$.

When we say that we store a suffix j in a data structure $1DRQ$ or $1DRQ - I$, we actually mean to store $x(j)$. The function $x(\cdot)$ is described in the section 3.3: $x(j)$ is the index of the leaf corresponding to the suffix j in the inorder traversal order of the leaves of the suffix tree (or, equivalently, $x(j)$ is the order of suffix j in the lexicographical ordering of the suffixes).

Because we store $x(j)$ in the 1D range query data structures, we are able to issue queries of the type “give me all the suffixes starting with P in this 1D range-query data structure.” For each pattern P , if X is the node in the suffix tree such that $P \preceq str(X)$ and $dist(X)$ is minimum, then $leaves(X)$ is precisely the set of occurrence positions of P , and the x values of positions $leaves(X)$ are precisely some interval $[x_1, x_2]$ (see more explanations in the section 3.3). Thus, for a 1D range-query data structure τ , if we issue a range-query $[x_1, x_2]$, we obtain the x values of the suffixes from τ that also start with P .

4.2 Search

With the above data structures, the search operation is as follows. Suppose the length of the query pattern P is $|P| \leq L$ (we explain the case when $|P| < L$ in section 4.4). Let X be the node in the suffix tree such that $P \preceq str(X)$ and $dist(X)$ is minimum; and let x_1, x_2 be the such that the range $[x_1, x_2]$ is equal to the set of x values of $J = leaves(X)$ (in section 3.3 we explain that x_1, x_2 can be found in $O(1)$ time once we computed X).

First, we extract all occurrences that are surely valid (occurrences $J \cap \{j \in [1, |T|] \mid j_i - j \geq L \text{ or } j_i - j < 0, \forall i = 1 \dots k\}$). These occurrences are returned by a *range - query*(x_1, x_2) in the V data structure (formally, the query returns x values of the occurrences).

Next, we need to extract all occurrences that are valid, but are still stored in the “invalid” $1DRQ - I$ data structures IND^δ . Let Δ be the minimum set of δ 's such that diadic intervals $[y_1^\delta, y_2^\delta]$ cover the interval $[|P|, L - 1]$; $|\Delta| = O(\log L)$. Then, all the remaining occurrences of P in T' (those that are not returned by the range-query in V) are the suffixes returned by range queries *range - query*(x_1, x_2) in the data structures $INV^\delta, \delta \in \Delta$. Thus, observe that the queries *range - query*(x_1, x_2) will return x values of positions $J \cap \{j \in [1, |T|] \mid \exists i \in [1, k] : |P| \leq j_i - j < L\}$.

Occurrences returned by *range - query*(x_1, x_2) in V and $INV^\delta, \delta \in \Delta$, are the x values of precisely all occurrences of P in the new text T' . We can compute $x^{-1}(\cdot)$ of the outputs of the range queries to obtain the actual positions of occurrences. $x^{-1}(\cdot)$ can be computed in $O(1)$ as noted in 3.3.

The search time is $O(|P| + \log |T| + \log L + occ)$ – the range-query in V takes $O(\log |T| + |output|)$, each of the $O(\log L)$ range-queries in $INV^\delta, \delta \in \Delta$, takes $O(|output|)$ time.

4.3 Update

Suppose we perform a deletion at position j_i . This new deletion could potentially decrease L , in which case we have to destruct several of the structures INV^δ . First, however, we will assume that the deletion at position j_i does not decrease the value L (which happens when $j_i - j_{i-1} - 1 \geq L$ and $j_{i+1} - j_i - 1 \geq L$).

If the deletion does not change the current value of L , then the suffixes starting in the range $[j_i - L + 1, j_i]$ are in the V data structure. We need to evict suffixes $[j_i - L + 1, j_i]$ from V and add them to corresponding INV^δ data structures. Thus, for all $j \in [j_i - L + 1, j_i]$, we perform a *delete*($x(j)$) operation on V . Then, for each $j \in [j_i - L + 1, j_i]$, we insert $x(j)$ in INV^δ for all δ such that $j_i - j \in [y_1^\delta, y_2^\delta]$.

This step takes $O(L \cdot \log |T|)$ steps since it takes $O(L \cdot \log |T|)$ to delete suffixes $[j_i - L + 1, j_i]$ from V , and it takes $O(L \log L)$ to insert suffixes $[j_i - L + 1, j_i]$ into INV^δ (each suffix is inserted in $O(\log L)$ INV^δ structures).

Now, consider that j_i decreases the value of L . Let L' be the new value of L . We need to destruct the diadic intervals that cover the interval $[L', L - 1]$, and put the suffixes contains this

interval back into the data structure V . Destruction itself does not take any time (we do not need to actually destroy the data structures), however we do need to spend time putting back the suffixes from the destructed diadic intervals into V . To put back the suffixes from the destructed diadic intervals into V , we perform *range-query*(1, $|T|$) operation on all destructed structures INV^δ that correspond to intervals of length 1 (all structures INV^δ where $\delta = (0, \chi), \chi \in [L', L - 1]$). Once we obtain the suffixes, we insert them back into the structure V . Note that all these operations take $O(d \cdot \log |T|)$ time, where d is the number of suffixes that are inserted back into V . However, we charge the time of putting back suffixes to the corresponding operations of deleting the d suffixes from V . Thus, the amortized time of these operations (putting suffixes back into V) is zero.

Further, we do the same operations as in the case when j_i does not change L .

Note that over the entire lifespan of our data structures (from the start moment to the point when all data structures are reconstructed), the value of L initially increases (due to P^{max} term) and then decreases (due to shrinking distance between consecutive deletions). During the lifespan of our data structures, we evict at most P^{max} suffixes from V for each deletion j_i (and at most P^{max} suffixes are inserted back into V). Thus, amortized, at most P^{max} suffixes are deleted from V during an update operation (there are k update operations). For each suffix that is deleted from V , we have $O(\log |T|)$ time for deletion from V ; $O(\log P^{max})$ time for inserting into at most $O(\log |P^{max}|)$ INV^δ data structures; and $O(\log |T|)$ time for inserting back into V . Thus, total amortized time for an update stage is $O(|P^{max}| \cdot \log |T|)$.

4.4 More on search: treating $|P| > L$

It is possible that $|P| > L$ when the current search pattern is longer than any previous search patterns, $|P| > |P^{max}|$, but still $|P|$ is shorter than the distance between any two consecutive deletions: $|P| \leq j_i - j_{i-1}, i \in [2, k]$. Whenever, the length search pattern P is greater than the current L , we first update the value L , and then construct new data structures INV^δ , where δ describes the new diadic intervals. Note that we do not have actually to construct the new structures INV^δ (they are preconstructed in the precomputation stage), however, we do need to delete some suffixes from V and add them into the newly created INV^δ structures. We charge all this time to the update stage (and the analysis of the update stage presented above includes these times). Therefore, the amortized time for search does not change.

4.5 1D range query data structures 1DRQ and 1DRQ - I

As mentioned before, 1DRQ data structure could be a classical balanced binary search tree that has $O(\log |T|)$ time for insertion/deletion operation and $O(\log |T| + |output|)$ for range-query.

1DRQ - I with $O(1)$ for insertion/preprocessing and $O(|output|)$ for range-query is trickier. We construct 1DRQ - I data structure in the following manner. First, we show how to construct a 1DRQ - I structure that has $O(1)$ for insertion and $O(|output|)$ for range-query, but takes $O(n)$ time to preprocess. Note that our 1DRQ - I structures are required to work only for integers in the range $[1, n]$ (here and later, we denote by n the value $|T|$).

We construct 1DRQ - I using marked ancestor problem [2]. In the preprocessing stage, we construct a tree that is a chain: node $i + 1$ is the child of node i , where $i = 1 \dots n$. Using marked ancestor problem, the tree supports the following operations: *mark*(i) - marks the node i ; *firstmarked*(i) - returns the greatest $j, j \leq i$, such that j is marked. In [2], it is shown how to support such operations in $O(1)$ time.

The insertion of a point x into 1DRQ - I is just marking of node $x, mark(x)$. The insertion takes $O(1)$ time because *mark*(x) takes $O(1)$ time.

Range-query in the interval $[x_1, x_2]$ is implemented as follows. Find the first ancestor of $x_2, v_1 = findmarked(x_2)$. If $v_1 \geq x_1$, output v_1 and find $v_2 = findmarked(v_1 - 1)$. If $v_2 \geq x_1$, then output v_2 , and find $v_3 = findmarked(v_2 - 1)$. Continue finding v_i 's until some $v_i < x_1$ - at this moment we found all the values $v_1 \dots v_{i-1}$ that are in the range $[x_1, x_2]$. Since *findmarked*(x) takes $O(1)$ time, we spend in total $O(|output|)$ to reply to a range-query.

The deficiency of the described $1DRQ - I$ data structure is that it requires $O(n)$ preprocessing time and $O(n)$ space. To bypass this deficiency, we use the following trick. Suppose we have a data structure that takes $O(1)$ time per operation. After h operations, the data structure touched at most $O(h)$ memory locations.

Using the above observation, adopt the following strategy. Keep an initialized $1DRQ - I$ data structure Υ in a separate place (Υ uses $O(n)$ space and $O(n)$ preprocessing time); Υ is never modified. A new $1DRQ - I$ data structure τ will have a special hash table of memory locations τ modified so far; together with the memory locations we store also the values stored in the corresponding locations (the hash table is a kind of cache). Thus, when the data structure τ reads a memory location l that has not been accessed by τ before (is not in the hash table), then read the memory location l from Υ . If τ reads a memory location that is in the hash table, then get the value associated to l in the hash table. If τ writes to a memory location l , then: 1) if l is in the hash table, update the value kept in the hash table; 2) if l is not in the hash table, add l to the hash table (together with the new value).

If we use dynamic perfect hashing for the hash table [12], we obtain $O(1)$ amortized expected time per operation. Also, the space used is $O(h)$, where h is the number of operations performed on τ so far. Note that once $h \geq |T|$, we could use normal $1DRQ - I$ structure (without the hash tables).

Thus, during the preprocessing time of a new $1DRQ - I$ data structure τ , we need only to initialize an empty dynamic perfect hash table, which takes $O(1)$ time. Thus we obtain the desired $1DRQ - I$ data structure.

5 Remarks

5.1 Preprocessing time

The processing time is $O(|T| \log |T|)$.

Constructing the suffix tree and the prefix tree takes $O(|T|)$ time. Initializing the $2DRQ$ data structure used to obtain the one-stranded occurrences (sections 3.3, 3.4) takes $O(|T| \log |T|)$ time [4]. All other helping data structures for the first part of the algorithm (one-stranded occurrences) take $O(|T|)$ or $O(|T| \log |T|)$ time to preprocess.

For the second part of the algorithm (zero-stranded occurrences), the main time is to construct the $1DRQ/1DRQ - I$ data structures. We need to construct Υ structure (preprocess a $1DRQ - I$ data structure), and this takes $O(n)$ time. Next, we need to initialize $O(|T|)$ INV^δ data structures (we construct all of them at the beginning so that we do not need to actually initialize any of INV^δ data structures when we add new diadic intervals). Their initialization also takes $O(|T|) \cdot O(1) = O(|T|)$ time. Initialization of the $1DRQ$ structure takes $O(|T| \log |T|)$ time.

Thus, total preprocessing time is $O(|T| \log |T|)$.

5.2 Space requirement

Space requirement for $2DRQ$ data structure is $O(|T| \log |T|)$. The suffix tree, prefix tree, TS , TP , and other helper data structures in the first part of the algorithm take $O(|T|)$ space.

In the second part of the algorithm, we need $O(|T|)$ to maintain the $1DRQ$ data structure V , and $O(|T|)$ to maintain the Υ data structure. Further, we have $O(|T|)$ $1DRQ - I$ data structures, and $|P^{max}|$ of them take at most $O(|T|)$ space. Therefore, the total space requirement for $1DRQ - I$ data structures is $O(|P^{max}| \cdot |T|)$. We believe this space requirement is actually smaller ($O(|T|)$), and this could be shown by a more attentive analysis of the memory used by $1DRQ - I$ structures (in particular, the analysis of the memory locations the structure writes to during a query).

5.3 Technicalities

To perform some of the operations described above, we need to store the positions of deletions in, say, a balanced binary search tree. Note that this does not change time bounds.

As noted earlier, we need to transform the occurrences indexed in the old text T into indexes in the new text T' . For this, we would need to be able to find for any $j \in [1, |T|]$ the number of deletions that precede j in $O(1)$. To do this, we, yet again, use a marked ancestor problem so that we could find the closest deletion to the left of j (deleted positions are the marked positions). Each of the deletion positions could also hold the number of deletions preceding it.

6 Conclusions

In this paper, we presented a data structure for dynamic pattern matching problem. We divided our algorithm into two separate independent steps. In the first step, the algorithm reports all the occurrences that are stranded by a modification. In the second step, the algorithm reports all the occurrences that are not stranded by any of the modifications.

The first step requires $O(|P| \log k \log \log k + occ)$ time for search and $O(\log k \log \log k)$ time for update. The second step of the algorithm takes $O(|P| + \log |T| + occ)$ time for search and $O(|P^{max}| \log |T|)$ time for update. P^{max} is the longest search pattern so far.

This paper is based on a previous work, [3]. Compared to [3], this paper improves significantly the update time of the first step, as well as provides a solution to the second step of the algorithm ([3] did not discuss the second step of the algorithm at all).

7 Acknowledgments

We thank Erik Demaine for his comments on the previous version of this work, [3], as well as for many insightful hints. We also thank Mihai Pătraşcu for pointing out marked ancestor problem, as well as for suggesting the neat idea of using cache-like hash tables for reducing preprocessing time of $1DRQ - I$ data structures.

References

- [1] P. Weiner, Linear Pattern Matching Algorithm, *Proceedings of 14th IEEE Symposium on Switching and Automata Theory*, pp. 1-11, 1973.
- [2] S. Alstrup, T. Husfeldt, T. Rauhe. Marked Ancestor Problems. *IEEE Symposium on Foundations of Computer Science*, pp. 534-544. 1998.
- [3] A. Andoni, C. Cadar. Dynamic Pattern Matching. *Final Project for 6.897*, Spring 2003. Available at <http://web.mit.edu/andoni/www/papers/dynamic.pdf>.
- [4] P. K. Agarwal. Range Searching. *Handbook of Discrete and Computational Geometry*, CRC Press, 1997.
- [5] H. N. Gabow, J. L. Bentley, R. E. Tarjan. Scaling and related techniques for geometry problems. *In Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 135-143, 1984.
- [6] M. Gu, M. Farach and R. Beigel. An efficient algorithm for dynamic text indexing. *In Proceedings of the Fifth Annual ACM/SIAM Symposium on Discrete Algorithms*, Arlington, VA, pp. 697-704, 1994.
- [7] A. Amir, N. Lewenstein, A. A. Schäffer, Practical Amortized Dynamic Indexing. Available at <http://citeseer.nj.nec.com/363888.html>. Last time accessed on December 10, 2003.
- [8] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM J. Comp.*, 27(3):713-736, 1998.
- [9] P. Ferragina, R. Grossi. Improved Dynamic Text Indexing. *J. Algorithms.*, 31(2):291-319, 1999.
- [10] S. C. Sahinalp, U. Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. *IEEE Symposium on Foundations of Computer Science*, pp. 320-328, 1996.
- [11] S. Alstrup, G. S. Brodal, T. Rauhe. Pattern matching in dynamic texts. *Symposium on Discrete Algorithms*, pp. 819-828, 2000.
- [12] M. Dietzfelbinger, A R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H Rohnert, R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *IEEE Symposium on Foundations of Computer Science*, pp. 524-531, 1988.