Improved Approximate String Matching using Compressed Suffix Data Structures

Tak-Wah Lam[2], Wing-Kin Sung[1], and Swee-Seong Wong[1*]

[1] School of Computing, National University of Singapore, Singapore,
{ksung,wongss}@comp.nus.edu.sg

[2] Department of Computer Science, The University of Hong Kong, Hong Kong
twlam@cs.hku.hk

Abstract. Approximate string matching is about finding a given string pattern in a text by allowing some degree of errors. In this paper we present a space efficient data structure to solve the 1-mismatch and 1-difference problems. Given a text T of length n over a fixed alphabet A, we can preprocess T and give an $O(n\sqrt{\log n})$-bit space data structure so that, for any query pattern P of length m, we can find all 1-mismatch (or 1-difference) occurrences of P in $O(m \log\log n + occ)$ time, where occ is the number of occurrences. This is the fastest known query time given that the space of the data structure is $o(n \log^2 n)$ bits.

The space of our data structure can be further reduced to $O(n)$ if we can afford a slow down factor of $\log^\epsilon n$, for $0 < \epsilon \leq 1$. Furthermore, our solution can be generalized to solve the k-mismatch (and the k-difference) problem in $O(|A|^k m^k (k + \log\log n) + occ)$ and $O(\log^\epsilon n(|A|^k m^k (k + \log\log n) + occ))$ query time using an $O(n\sqrt{\log n})$-bit and an $O(n)$-bit indexing data structures, respectively.

1 Introduction

Consider a text T of length n and a pattern P of length m, both strings over a constant size alphabet A. The approximate string matching problem is to find all approximate occurrences of P in T. Depending on the definition of "error", this problem has two variations: (1) The k-difference problem is to find all occurrences of P in T that have edit distance at most k from P (edit distance is the minimum number of character insertions, deletions and replacements to convert one string to another); and (2) The k-mismatch problem is to find all occurrences of P in T that have Hamming distance at most k from P (Hamming distance is the minimum number of character replacements to convert one string to another). Both k-difference and k-mismatch problems are well-studied and they found applications in many areas including computational biology, text retrieval, multi-media data retrieval, pattern recognition, signal processing, handwriting recognition, etc.

Recently, people are interested in the offline approximate matching problem, in which, we can preprocess the text T and build some indexing data structure so that any pattern query can be answered in a shorter time. Jokinen and Ukkonen [6] were the

* Supported by Institute for Infocomm Research, a member of A*STAR, Singapore.

first to treat the approximate offline matching problem. Since then, many different approaches have been proposed. Some techniques are fast on the average [8, 9]. However, they incur a query time complexity depending on n, i.e., in the worst case, they are inefficient even if the pattern is very short and k is as small as one. The first solution with query time complexity independent of n is proposed by Ukkonen [13]. When $k = 1$ (that is, 1-mismatch or 1-difference problem), Cobbs [3] gave an indexing data structure using $O(n \log n)$ bits space and having $O(m^2 + occ)$ query time. Later, Amir et al [1] proposed an $O(n \log^3 n)$-bit indexing data structure with $O(m \log n \log \log n + occ)$ query time. Then, Buchsbaum et al [2] proposed another indexing data structure which uses $O(n \log^2 n)$ bits space so that every query can be solved in $O(m \log \log n + occ)$ time. Cole *et. al.* [4] further improved the query time. They gave an $O(n \log^2 n)$-bit data structure so that both the 1-mismatch and the 1-difference problems can be solved in $O(m + \log n \log \log n + occ)$ time, respectively. Recently, in order to target large text indexing like in genomic sequences, Trinh *et. al.* [12] improves upon the space-efficiency. They proposed two data structures of size $O(n \log n)$ bits and $O(n)$ bits with query time $O(m \log n + occ)$ and $O(m \log^2 n + occ \log n)$, respectively.

Some of the above results can be generalized for $k > 1$. Cobbs's $O(n \log n)$-bit indexing data structure can answer both k-mismatch and k-difference queries in $O(m^{k+2}|A|^k + occ)$ time [3]. Cole *et. al.* [4] proposed an $O(n \frac{(c_3 \log n)^k}{k!} \log n)$-bit indexing data structure with query times of $O(\frac{(c_1 \log n)^k \log \log n}{k!} + m + occ)$ and $O(\frac{(c_2 \log n)^k \log \log n}{k!} + m + 3^k \cdot occ)$ for the k-mismatch and k-difference problems, respectively, where c_1, c_2, c_3 are constants with $c_2 > c_1$. Trinh *et. al.* [12] gave $O(n \log n)$-bit and $O(n)$-bit data structures that can answer a k-mismatch (or a k-difference) query in $O(|A|^k m^k \log n + occ)$ time and $O(|A|^k m^k \log^2 n + occ \log n)$ time, respectively.

All previous data structures for supporting the 1-mismatch (or 1-difference) query either require a space of $\Omega(n \log^2 n)$ bits or $\Omega(m \log n + occ)$ time. It is an open problem whether there exists an $O(n \log n)$ or even $o(n \log n)$-bit data structure so that every 1-mismatch (or 1-difference) query can be answered in $o(m \log n + occ)$ time. In this paper, we resolve this open problem in the affirmative by presenting a data structure which uses $O(n\sqrt{\log n})$ bit space while every 1-mismatch (or 1-difference) query can be answered in $O(m \log \log n + occ)$ time.

Our result can be further extended in two ways. First, we show that the space of the data structure can be reduced to $O(n)$ bits if we accept a slow down factor of $\log^\epsilon n$ for the query time where $0 < \epsilon \leq 1$. Second, the data structure can be extended to solve the k-mismatch (or the k-difference) problem for $k \geq 1$. Our solution can solve the k-mismatch (or the k-difference) problem in $O(|A|^k m^k (k + \log \log n) + occ)$ or $O(\log^\epsilon n\ (|A|^k m^k (k + \log \log n) + occ))$ query time, when the text is encoded using $O(n\sqrt{\log n})$-bit or $O(n)$-bit indexing data structures, respectively.

2 Preliminaries

2.1 Suffix array, inverse suffix array, and Ψ function

Let $T[0..n] = t_0 t_1 \cdots t_{n-1}$ be a text of length n over an alphabet A, appended with a special symbol $t_n =$ '$\$$' that is not in A and is smaller than any other symbol in A. The j-th suffix of T is defined as $T[j..n] = t_j \cdots t_n$ and is denoted by T_j.

The *suffix array* $SA[0..n]$ of T is an array of integers so that $T_{SA[i]}$ is lexicographically smaller than $T_{SA[j]}$ if and only if $i < j$. Note that $SA[0] = n$. The *inverse suffix array* of T is denoted as $SA^{-1}[0..n]$, that is, $SA^{-1}[i]$ equals the number of suffixes which are lexicographically smaller than T_i.

In this paper, an interval $[st..ed]$ is called the range of the suffix array of T corresponding to a string P if $[st..ed]$ is the largest interval such that P is a prefix of every suffix T_j for $j = SA[st], SA[st+1], \ldots, SA[ed]$. We write $[st..ed] = range(T, P)$.

A concept related to the suffix array is the $\Psi[0..n]$ [5], which is defined as follows:

$$\Psi[i] = SA^{-1}[SA[i] + 1].$$

Let t_{SA} and t_{Ψ} be the access time of each entry on SA and Ψ respectively. In this paper, we need a data structure $\mathcal{D}$ which supports, for any i, the following operations.

- reports $SA[i]$ in t_{SA} time,
- reports $SA^{-1}[i]$ in t_{SA} time,
- reports $\Psi[i]$ in t_{Ψ} time, and
- reports *substring(i,l)* $= T[SA[i]..SA[i] + l - 1]$ in $O(lt_{\Psi})$ time for some length l.

Lemmas 1 and 3 give two implementations of the data structure $\mathcal{D}$.

Lemma 1. *[5] The data structure $\mathcal{D}$ can be implemented in $O(n \log |A|)$ bits so that $t_{SA} = O(\log^{\epsilon} n)$ and $t_{\Psi} = O(1)$, where $1 \geq \epsilon > 0$.*

Below lemma is needed for the second implementation of the data structure $\mathcal{D}$.

Lemma 2. *Let $X_1, ..., X_\ell$ be ℓ subsets of $\{0, ..., n-1\}$ such that $|X_j| = n/\ell$, $1 \leq j \leq \ell$. Then $\{\Psi^j[z] | z \in X_j\}$ for all j, can be stored in an $O(n\ell \log |A| + |A|^\ell \log n)$-bit data structure such that $\Psi^j[z]$, for any $z \in X_j$, can be accessed in $O(1)$ time.*

Proof. The result follows from data structures described in Rao's paper [10]. $\square$

Lemma 3. *The data structure $\mathcal{D}$ can be implemented in $O(n\sqrt{\log n} \log |A|)$ bits so that $t_{SA} = O(1)$ and $t_{\Psi} = O(1)$.*

Proof. Building the $O(n \log |A|)$-bit data structure in Lemma 1, the Ψ function can be accessed in $O(1)$ time. Below, we describe $O(n\sqrt{\log n} \log |A|)$-bit data structures so that both SA and SA^{-1} can be computed in $O(1)$ time.

For the access of SA value, recall that Rao [10] gives an implementation of the compressed suffix array that reports $SA[i]$ in $O(1)$ time using $O(n\sqrt{\log n})$ bits for binary text string (refer to Theorem 4 in [10]). For text on a fixed finite alphabet A, Rao's idea can be generalized so that $SA[i]$ can be accessed in constant time using an $O(n\sqrt{\log n} \log |A|)$-bit data structure.

For the access of SA^{-1} value, we need the following data structure. Let $\ell = \sqrt{\log n}$. First, we store $SA^{-1}[x\ell]$ for all $0 \leq x \leq \lfloor n/\ell \rfloor$, which requires $O(n\sqrt{\log n})$ bits. Then, we need a data structure so that $\Psi^j[z]$ can be accessed in $O(1)$ time for any

$0 \le x \le \lfloor n/\ell \rfloor$ and any $1 \le j \le \ell$. By Lemma 2, such data structure can be stored in $O(n\sqrt{\log n}\log|A| + |A|^{\sqrt{\log n}}\log n) = O(n\sqrt{\log n}\log|A|)$ bits.

Now we show how to access $SA^{-1}[i]$ given i in constant time. Let $y = \lfloor i/\ell \rfloor$, $k' = i - y\ell$, and $z' = SA^{-1}[y\ell]$. We claim that $SA^{-1}[i] = \Psi^{k'}[z']$ and $k' \le \ell$. Then, using the data structures above, $SA^{-1}[i]$ can be computed in $O(1)$ time.

Note that $y\ell \le i \le (y+1)\ell$ and thus, $k' = i - y\ell \le \ell$. It is then easy to verify that $SA[\Psi^{k'}[z']] = SA[z'] + k'$. Since $SA[z'] = y\ell$, we have $SA[\Psi^{k'}[z']] = y\ell + k' = i$. Thus, the claim follows. □

2.2 Suffix tree

A suffix tree for the text T is an edge-labeled rooted directed tree with exactly $n+1$ leaves numbered 0 to n. Each edge is labeled with a non-empty substring of T such that no two outgoing edges from a node have labels with the same first character. For every node v, its path label $plabel(v)$ is constructed by concatenating the edge labels, in order, from the root to the node. Note that the path label of every leaf i, is a suffix of T that starts at position i.

We assumed that the suffixes of the leaves in the suffix tree are lexicographically ordered so that the collection of leaf nodes from left to right will form the suffix array denoted by $SA[0..n]$. For our approach, we require a suffix tree that support the following operations:

$label(u, v)$: returns the label on the edge joining node u to v in $O(xt_{SA})$ time where x is the length of the edge label of (u, v).

$plen(v)$: returns the length of the path label $plabel(v)$ in $O(t_{SA})$ time .

$leftmost(v)$: returns the SA index of the leftmost leaf in the subtree rooted at node v in $O(1)$ time.

$rightmost(v)$: returns the SA index of the rightmost leaf in the subtree rooted at node v in $O(1)$ time.

$slink(v)$: returns a node u if there is a suffix link from node v to node u in $O(t_\Psi)$ time.

$child(v, c)$: returns a child w of the node v if c is a prefix character to string $label(v, w)$ in $O(|A|t_{SA})$ time.

Lemma 4. *A suffix tree with the above properties can be implemented using (1) $O(n\log|A|)$ bits for $t_{SA} = O(\log^\epsilon n)$ and $t_\Psi = O(1)$, or (2) $O(n\sqrt{\log n}\log|A|)$ bits for $t_{SA} = O(1)$ and $t_\Psi = O(1)$.*

Proof. We refer to Sadakane's paper [11] on compressed suffix tree (CST) implementation that uses data structure $\mathcal{D}$ and $O(n)$ bits for the balanced parentheses representation of the suffix tree [7]. The space complexities follow from Lemmas 1 and 3. □

The following result on LCP query is also available.

Lemma 5. *[11] Given SA indexes i and j, the length of the longest common prefix (LCP) between suffixes at positions $SA[i]$ and $SA[j]$, denoted by $|lcp(i, j)|$, can be computed in $O(t_{SA})$ time using additional $O(n)$ bits data structure. The lowest common ancestor (LCA) node between any two nodes in the suffix tree can also be computed in $O(t_{SA})$ time.*

2.3 Other data structures

Given a suffix tree ST built from the text T, and a query pattern P of length m, we define the following terminologies and data structures:

Definition 1. *Given a node x in ST, let x_{le} and x_{ri} denote indexes of SA corresponding to the leftmost and rightmost leaf nodes in the subtree spanned by x.*

Based on the above definition, for any node x in ST, we have $[x_{le}..x_{ri}] = range(T, plabel(x))$.

Definition 2. *Arrays $F_{st}[1..m]$ and $F_{ed}[1..m]$ are such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$ for $1 \leq i \leq m$. We also define $F_{st}[j] = 0$ and $F_{ed}[j] = n$ for $j > m$.*

Lemma 6. *$F_{st}[1..m]$ and $F_{ed}[1..m]$ can be constructed in $O(mt_{\Psi} + m|A|t_{SA})$ time.*

Proof. This can be done using the suffix links in ST in $O(mt_{\Psi})$ time, given that $range(T, P[1..m])$ can be obtained by traversing the suffix tree in $O(m|A|t_{SA})$ time. □

Furthermore, the following lemma is needed to support exact pattern search over a subtree in the suffix tree.

Lemma 7. *Given a pattern P, let x be a node such that $[x_{le}..x_{ri}] = range(T, P)$. For any position i in T, P is a prefix of $T[i..n]$ if and only if $x_{le} \leq SA^{-1}[i] \leq x_{ri}$. Also, $SA^{-1}[SA[x_{le}] + |P|] < SA^{-1}[SA[x_{le} + 1] + |P|] < \ldots < SA^{-1}[SA[x_{ri}] + |P|]$.*

2.4 Heavy path decomposition

We introduce a standard technique to partition $O(n)$ nodes of a tree into $O(\log n)$ levels. The heavy path decomposition scheme is as such: Given a suffix tree ST, we assign a level to every node in ST. The root is assigned $level$ 1. If a node v has $level\ i$, we assign $level\ i$ to the single child node of v, that has the largest subtree (in terms of number of nodes) among all the other child nodes of v. The other child nodes of v are assigned $level\ i+1$. Edges joining 2 nodes with the same $level$ are denoted as *core edges* and the rest of edges that join nodes at $level\ i$ to nodes at $level\ i+1$ are denoted as *side edges*. An internal node will have exactly one outgoing *core edge* and the rest of the outgoing edges are *side edges*. We also denote a node with an incoming core edge as a *core node* and otherwise, a *side node*. The root, is by default, a *side node*. There are $O(\log n)$ levels. The followings are also observed:

Lemma 8. *There are $O(\log n)$ side edges on the path from the root to any node in the suffix tree.*

Lemma 9. *Consider any two distinct side edges e_1 and e_2 with end nodes v_1 and v_2 respectively. If both v_1 and v_2 have $level\ i$, then the two subtrees rooted at v_1 and v_2 are disjoint. In other words, the subtrees rooted at any two distinct side nodes of the same level are disjoint.*

Lemma 10. *Given any side node v, that begins a core path (where all edges in the path are core edges), we can find the leaf node u that terminates the core path in $O(1)$ time using additional $O(n)$ bits data structure.*

Proof. Let u_i be the i-th node in the suffix tree according to the prefix order, and $t = O(n)$ be the number of nodes in the suffix tree. Let $X[1..t]$ be an array such that X[i]='(' if u_i is an internal side node and $X[j] =')'$ if u_j is a core leaf node. It can easily check that the parentheses in $X[1..t]$ is balance. More importantly, for every pair of parentheses (i, j) in X, u_i and u_j form the start and the end of some core path. By the data structure for balanced parentheses [7], the lemma follows. □

3 1-Approximate string matching problem

3.1 The data structure for 1-approximate matching

Our 1-approximate matching data structure is basically the suffix tree ST of the text T (see Section 2.2), together with two other data structures. First, for every side node v (see Section 2.4), let u be the parent node of v, we maintain a set $\Gamma_v = \{SA^{-1}[SA[i] + plen(u) + 1] \mid i = 1(\text{mod } \log^2 n)$ and $v_{le} \leq i \leq v_{ri}\}$.

Second, for every core leaf node u (whose SA index is k), let v be the start of the corresponding core path, we maintain 2 lists of SA indexes, $H^l_u = \{i \mid i = 1(\text{mod } \log^2 n), i \leq k$ and $|lcp(k,i)| \geq plen(v)\}$ and $H^r_u = \{i \mid i = 1(\text{mod } \log^2 n), i > k$ and $|lcp(k,i)| \geq plen(v)\}$. The values in H^l_u and H^r_u are ordered by increasing longest common prefix length $|lcp(k,i)|$.

Lemma 11. *We can store Γ_v for all side nodes v using $O(n)$ bit space. In addition, we can answer any range query in Γ_v using $O(\log\log n)$ time.*

Proof. By Lemma 9, all subtrees rooted at different side nodes, of the same level-ℓ, are disjoint. Hence, the total size of Γ_v for all level-ℓ side nodes v is at most $n/\log^2 n$. Since there are $O(\log n)$ levels, the total size of Γ_v for all side nodes v is $O(n/\log n)$. We store Γ_v, for every side node v, using the *y-fast trie* [14] data structure. The size of the data structure is $O(|\Gamma_v| \log n)$ bits and it allows efficient range query in $O(\log\log n)$ time. Since the total size of all Γ_v is $O(n/\log n)$, the lemma follows. □

Lemma 12. *We can store H^l_u and H^r_u for all core leaf nodes u (whose SA index is k) using $O(n)$ bit space. In addition, for any range $[x..y]$, we can report the values $i^l \in H^l_u$ and $i^r \in H^r_u$ such that $x \leq lcp(i^l, k) \leq y$ and $x \leq lcp(i^r, k) \leq y$ using $O(\log\log n)$ time.*

Proof. There are at most $n/\log^2 n$ selected leaf nodes and each leaf node is reachable from at most $\log n$ different core paths. Hence, the total size of H^l_u and H^r_u for all core paths is $O(n/\log n)$. Each H^l_u and H^r_u is stored using the *y-fast trie* [14] data structure. The size of the data structures is $O((|H^l_u| + |H^r_u|) \log n)$ bits. Since the total size of all H^l_u and H^r_u is $O(n/\log n)$, the lemma follows. □

By Lemmas 4, 11 and 12,

Lemma 13. *The 1-approximate matching data structure can be stored in* $O(n \log |A|)$ *and* $O(n\sqrt{\log n} \log |A|)$ *bits for* $t_{SA} = O(\log^{\epsilon} n)$ *and* $t_{SA} = O(1)$ *respectively. In both cases,* $t_{\psi} = O(1)$.

Below is the key lemma for our algorithm.

Lemma 14. *Consider (1) a node* u *in* ST *such that* $P_1 = plabel(u)$*, (2) a character c, and (3) another string* P_2 *with* $[st..ed] = range(T, P_2)$*. Let* $v = child(u, c)$*. Then, all occurrences of* $P_1 c P_2$ *can be computed in* $O(t_{SA}(\log\log n + occ))$ *time where occ is the total number of occurrences of* $P_1 c P_2$ *in* T.

Proof. Note that $[v_{le}..v_{ri}] = range(T, P_1 c)$. If $P = P_1 c P_2$ occurs in T, as $P_1 c$ is a prefix of P, P must occur at position $SA[i]$ for some $v_{le} \leq i \leq v_{ri}$. By Lemma 7, we can verify if P occurs at position $SA[i]$ by checking if $st \leq SA^{-1}[SA[i] + |P_1| + 1] \leq ed$. Hence, the occurrence of P can be found by performing the above checking for all $i \in [v_{le}..v_{ri}]$. Moreover, by Lemma 7, $SA^{-1}[SA[i] + |P_1| + 1]$ is increasing for $i \in [v_{le}..v_{ri}]$. Note that, for any i, $SA^{-1}[SA[i] + |P_1| + 1]$ can be retrieved in $O(t_{SA})$ time. Hence, one occurrence of P can be found in $O(t_{SA} \log(v_{ri} - v_{le}))$ time by binary search.

If v is a side node, recall that we associate a set Γ_v to it, where $\Gamma_v = \{SA^{-1}[SA[i] + |P_1| + 1] \mid i = 1 (\text{mod } \log^2 n) \text{ and } i \in [v_{le}..v_{ri}]\}$. There are 3 cases.

- Case 1: Γ_v is empty. This means that the number of leaves in the subtree of v $(v_{ri} - v_{le} + 1)$ is $< \log^2 n$. Using the method we just discussed, one occurrence of P can be found in $O(t_{SA} \log(v_{ri} - v_{le})) = O(t_{SA} \log\log n)$ time.
- Case 2: Γ_v is non-empty and, by Lemma 11, we find some i such that $st \leq SA^{-1}[SA[i]+|P_1|+1] \leq ed$. Since any range query of *y-fast trie* takes $O(\log\log n)$ time, the second case follows.
- Case 3: Γ_v is non-empty and, by Lemma 11, we cannot find any i such that $st \leq SA^{-1}[SA[i]+|P_1|+1] \leq ed$. In this case, using $O(\log\log n)$ time, we apply *y-fast trie* to find a and b such that $SA^{-1}[SA[a] + |P_1| + 1] \in \Gamma_v$ is just smaller than st and $SA^{-1}[SA[b] + |P_1| + 1] \in \Gamma_v$ is just bigger than ed. Note that $b - a \leq \log^2 n$. Then, using the method described at the beginning of the proof, we can found one occurrence of P in $O(t_{SA} \log(b - a)) = O(t_{SA} \log\log n)$ time.

If v is a core node, let CP be the core path containing v. Since the side node lies on the path from the root node to v and would have been uncovered from traversing the suffix tree to obtain P_1, here we assume that the side node that begins the core path CP is known. We obtain the terminating leaf node x (whose SA index is k) of CP by Lemma 10 using $O(1)$ time. Next, we search for the node $r \in CP$ whose path label is of length $|P_1| + q + 1$ where $q = |lcp(SA^{-1}[SA[k] + |P_1| + 1], st)|$. By Lemma 5, q is computed in $O(t_{SA})$ time. There are 3 cases.

- Case 1: H_x is empty. This means that the number of leaves hanging from the core path is $< \log^2 n$. We can find one occurrence of P in $O(t_{SA} \log(v_{ri} - v_{le})) = O(t_{SA} \log\log n)$ time.
- Case 2: $q \geq |P_2|$. This means that leaf node x corresponds to a suffix with P as its prefix. We have recovered one occurrence of P.

– Case 3: $q < |P_2|$. First, we would like to find $j^l \in H_x^l$ and $j^r \in H_x^r$ such that $|lcp(j^l - \log^2 n, k)| \leq |P_1| + q + 1 \leq |lcp(j^l, k)|$ and $|lcp(j^r, k)| \leq |P_1| + q + 1 \leq |lcp(j^r + \log^2 n, k)|$ respectively. This can be computed in $O(\log \log n)$ time given Lemma 12. Next, using binary search, we locate i^l and i^r within the range of $j^l - \log^2 n...j^l$ and $j^r...j^r + \log^2 n$ such that $|lcp(i^l, k)| = |P_1| + q + 1$ and $|lcp(i^r, k)| = |P_1| + q + 1$ respectively. If both j^l and j^r are not found, the binary search is performed for $k - \log^2 n \leq i^l \leq k + \log^2 n$. The binary search takes $O(t_{SA} \log \log n)$ time. Given i^l or i^r whichever one is found, we can recover node r by performing a LCA on the leaf node at i^l or i^r with x in $O(t_{SA})$ time. If P_2 is not completely matched after node r, we can continue to search the outgoing side edges from node r as described above (case where v is a side node) using additional $O(t_{SA} \log \log n)$ time. Overall, the time taken is still $O(t_{SA} \log \log n)$.

Once we confirm that P occurs in position $SA[i]$, the remaining occurrences of P can be found by performing, for entries i' to the left and to the right of i, the above checking (that is, $st \leq SA^{-1}[SA[i'] + |P_1| + 1] \leq ed$), until we reach the boundary of $[v_{le}..v_{ri}]$ or a false case occurs. The time required is $O(t_{SA}(occ + 2))$. □

Here, we define the procedure $TreeSearch(u, c, [st..ed])$ to be the routine which finds all the occurrences of $P_1 c P_2$ where $P_1 = plabel(u)$ and $[st..ed] = range(T, P_2)$. By Lemma 14, this procedure runs in $O(t_{SA}(\log \log n + occ))$ time.

3.2 The 1-approximate matching algorithm

The algorithm traverses the suffix tree from the root to find the pattern P character by character. Then, for every position i, it introduces an "error" at that position and checks for occurrences by calling $TreeSearch$. The details of the algorithm is stated in Figure 1.

Lemma 15. *Given the indexing data structure in Section 3.1, we can locate all* 1-*mismatch (or* 1-*difference) occurrences of a length-m pattern P in T, using $O(t_\Psi m + t_{SA}(|A| m \log \log n + occ))$ time.*

Proof. By Lemma 6, Step 1 takes $O(mt_\Psi + m|A|t_{SA})$ time. Step 2 takes $O(1)$ time.

When we traverse down the suffix tree to a node u (with $plabel(u) = P[1..i-1]$), we will execute Steps 3(a-c) for the node u. By Lemma 14, Steps 3(a-c) in total takes $O(t_{SA}(|A| m \log \log n + occ))$ time.

For Step 3(d), it takes $O(t_{SA}|A|m)$ time. The lemma follows. □

By Lemmas 13 and 15, we get the following 2 theorems:

Theorem 1. *Given an $O(n\sqrt{\log n} \log |A|)$-bit indexing data structure, the 1-mismatch or 1-difference problem can locate all* 1-*approximate occurrences of a length-m pattern P in T, using $O(|A| m \log \log n + occ)$ time.*

Theorem 2. *Given an $O(n \log |A|)$-bit indexing data structure, the 1-mismatch or 1-difference problem can locate all* 1-*approximate occurrences of a length-m pattern P in T, using $O(\log^\epsilon n(|A| m \log \log n + occ))$ time, where $0 < \epsilon \leq 1$.*

Algorithm *1-approximate match*

1. Construct $F_{st}[1..m]$ and $F_{ed}[1..m]$ such that $[F_{st}[i]..F_{ed}[i]] = range(T, P[i..m])$.
2. u = root node, $i = 1$.
3. Repeat
  /* Note: we maintain the invariant that $plabel(u) = P[1..i-1]$. */
  (a) Deletion at i (find occurrences of $P[1..i-1]P[i+1..m]$)
    If $P[i] \neq P[i+1]$
      report the occurrences found by $TreeSearch(u, P[i+1], [F_{st}[i+2]..F_{ed}[i+2]])$.
  (b) Substitution at i (find occurrences of $P[1..i-1]cP[i+1..m]$ for all $c \in A - \{P[i]\}$)
    For $c \in A - \{P[i]\}$,
      report the occurrences found by $TreeSearch(u, c, [F_{st}[i+1]..F_{ed}[i+1]])$.
  (c) Insertion at i (find occurrences of $P[1..i-1]cP[i..m]$ for all $c \in A - \{P[i]\}$)
    For $c \in A - \{P[i]\}$,
      report the occurrences found by $TreeSearch(u, c, [F_{st}[i]..F_{ed}[i]])$.
  (d) No insertion, deletion, and substitution at i
    Let $v = child(u, P[i])$, $E = label(u, v)$.
    If $P[i..i+|E|-1] = E$
      $u = v$, $i = i + |E|$
    Else
      Find the smallest $j > i$ such that $P[j] \neq E[j-i+1]$.
      Report all the occurrences of P so that the error is at j.
      Terminate and return.

Fig. 1. Algorithm for 1-mismatch and 1-difference.

3.3 The k-approximate matching problem with $k \geq 1$

Theorem 3. *After preprocessing the text T of length n and obtain an $O(n\sqrt{\log n}\log|A|)$ or $O(n \log|A|)$ bits data structure, the k-mismatch or k-difference problem can locate all approximate occurrences of a length-m pattern P in T, using $O(|A|^k m^k(k + \log\log n) + occ)$ or $O(\log^{\epsilon} n(|A|^k m^k(k + \log\log n) + occ))$ time respectively, where $0 < \epsilon \leq 1$, $|A|$ is the fixed alphabet size and occ is the number of approximate occurrences of P in T.*

Proof. We give a sketch of the proof here. The k-approximate matching problem can be solved using dynamic programming on the suffix tree in $O(|A|^k m^k mk)$ time. The factor $|A|^k m^k$ is the number of different paths down the suffix tree that are to be traversed, which is bounded by the number of edit traces [13]. By merging our data structure in Section 3.1 to compute only for the kth error, with dynamic programming over the suffix tree, we can obtain the stated complexities. □

4 Conclusion

In this paper, we give an $O(n\sqrt{\log n})$-bit data structure which can answer 1-mismatch (and 1-difference) query in $O(m \log\log n + occ)$ time. If we can afford a slow down factor of $\log^{\epsilon} n$ for the query time, the size of the data structure can be further reduced to $O(n)$ bits. We also generalize our solution to solve k-mismatch and k-difference problems.

5 Acknowledgments

The authors would like to thank Limsoon Wong for his helpful comments. This work is supported in part by the NUS Academic Research Grant R-252-000-119-112.

References

1. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000.
2. A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *Proceedings of the 8th Annual European Symposium on Algorithms*, pages 120–131, 2000.
3. A. L. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 41–54, July 1995.
4. R. Cole, L-A. Gottlieb, and M. Lewenstein. Dictionary matcing and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004.
5. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.
6. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science*, pages 240–248, September 1991.
7. J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39:205–222, 2001.
8. G. Navarro and R.Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.
9. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 350–365, 2000.
10. S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82:307–311, 2002.
11. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, Accepted.
12. H. N. D. Trinh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*, pages 434–444, 2004.
13. E. Ukkonen. Approximate string-matching over suffix trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pages 228–242, 1993.
14. D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17:81–84, August 1983.