Fast index for approximate string matching

Dekel Tsur*

Abstract

We present an index that stores a text of length n such that given a pattern of length m, all the substrings of the text that are within Hamming distance (or edit distance) at most k from the pattern are reported in $O(m + \log\log n + \#\text{matches})$ time (for constant k). The space complexity of the index is $O(n^{1+\epsilon})$ for any constant $\epsilon > 0$.

1 Introduction

One of the fundamental problems in pattern matching is indexing a text t such that given a query pattern p, all the occurrences of p in t can be reported efficiently. This can be solved optimally using suffix trees [12]: The construction time and space complexity of the index is $O(n)$, and the query time is $O(m + \#\text{matches})$, where n is the length of t, m is the length of p, and #matches is the number of times p appears in t. For simplicity, we shall assume throughout the paper that the size of the alphabet is constant.

A natural extension of text indexing is to allow approximate search in the index. Formally, given a text t and an integer k, the goal is to build an index for t such that given a query string p, all the substrings of t with Hamming distance (or edit distance) at most k from p can be reported efficiently. Again for simplicity, we assume throughout that k is constant. Building an approximate index with almost linear space and query time was a major open problem. The first efficient approximate index was obtained for the case $k = 1$ by Amir et al. [1]. The index of Amir et al. uses $O(n \log^2 n)$ space, and answer queries in time $O(m \log n \log\log n + \#\text{matches})$. A faster query time is obtained using the data-structure of [2]. Linear space indices that support one error were given in [7, 8].

A big breakthrough was obtained by Cole et al. [4] which presented an index that supports an arbitrary number of errors. The index of Cole et al. uses $O(n \log^k n)$ space and answers queries in time $O(m + \log^k n \cdot \log\log n + \#\text{matches})$. Chan et al. [3] gave an $O(n)$-space index that answers queries in time $O(m + (\log n)^{k(k+1)} \log\log n + \#\text{matches})$.

Most of the results above work for both Hamming distance or edit distance. We note that the query time complexity of the edit distance index in [4] is $O(m + \log^k n \cdot$

*Department of Computer Science, Ben-Gurion University of the Negev. Email: dekelts@cs.bgu.ac.il

$\log\log n + 3^k \cdot \#\text{matches})$. However, as we assume here that k is constant, the time complexity becomes $O(m + \log^k n \cdot \log\log n + \#\text{matches})$.

The indices mentioned above have worst case performance guarantees. Indices with good performance on average were given in [5, 6, 9–11].

In this paper, we show how to speed-up the query time in the index of Cole et al. This comes at a cost of increasing the space complexity of the index. More precisely, we show that for every integer α with $2 \leq \alpha \leq n/2$, there is an $O(n(\alpha \log \alpha \log n)^k)$-space index (for Hamming distance or edit distance) that answers queries in time $O(m + (\log_\alpha n)^k \log\log n + \#\text{matches})$. In particular, for every fixed $\epsilon > 0$, one can take $\alpha = \log^{\epsilon/2k} n$ and get an index with space complexity $O(n \log^{k+\epsilon} n)$ and query time $O(m + \log^k n/(\log\log n)^{k-1} + \#\text{matches})$ (recall that k is assumed to be constant, so $\log_\alpha n = \Theta(\log / \log\log n)$). To get faster query time, one can take $\alpha = n^{\epsilon/2k}$ for some $\epsilon > 0$ and get an index with space complexity $O(n^{1+\epsilon})$ and query time $O(m + \log\log n + \#\text{matches})$.

2 Preliminaries

Let $s_1, \ldots, s_n$ be a collection of strings, where each string ends with the character '\$', and '\$' does not appear elsewhere in $s_1, \ldots, s_n$. A *compressed trie* for $s_1, \ldots, s_n$ is a rooted tree T that has n leaves and each internal vertex has at least two children. Every edge of T is labeled by a string. Every string s_i corresponds to a distinct leaf v_i of T such that the concatenation of the labels of the edges on the path from the root of T to v_i is exactly s_i.

A *location* l on a compressed trie T is pair (v, s) where v is a vertex of T and s is an empty string or a proper prefix of the label of some edge between v and a child of v. We will sometimes refer to a vertex v as a location (v, ϵ) and vice versa.

For a vertex v in a compressed trie T, the string that corresponds to v is the concatenation of the labels on the path from the root of T to v. For a location $l = (v, s)$, the string that corresponds to l, denoted $\text{str}(l)$, is the concatenation of the string that corresponds to v and s.

The *weight* of a vertex v in a tree T is the number of descendent leaves of v. A path $[v_1, \ldots, v_d]$ in a tree T is a *heavy path* if (1) v_1 is the root of T, (2) v_d is a leaf, and (3) for every $i < d$, there is no child of v_i with weight greater than the weight of v_{i+1}. A *heavy path decomposition* of a tree T is a set $\mathcal{C}$ of paths in T such that (1) $\mathcal{C}$ contains a heavy path C of T, and (2) for every connected component T' in $T - C$, $\mathcal{C}$ contains the paths in a heavy path decomposition of T' ($T - C$ is the graph obtained from T by removing the vertices of C). For a heavy path decomposition $\mathcal{C}$ define $T_\mathcal{C}$ to be a rooted tree whose set of vertices is $\mathcal{C}$, and there is an edge from C to C' in $T_\mathcal{C}$ if there is a vertex $v \in C$ such that the topmost vertex in C' is a child of v in T.

Given a heavy path decomposition $\mathcal{C}$ of a compressed trie T and a location $l = (v, s)$ in T, $\text{nextloc}(l)$ is the location reached when moving from l one character along the path $C \in \mathcal{C}$ that contains v. Formally, $\text{nextloc}(l)$ is the location $l' = (v', s')$ such that the string $\text{str}(l')$ is the prefix of length $|\text{str}(l)| + 1$ of the string that corresponds to the bottommost vertex of C. If there is no such location l' then

nextloc(l) is undefined. We also define next(l) to be the last character of the string that corresponds to nextloc(l).

For a vertex v in a compressed trie T, nextchars(v) is the set of all first characters in the labels of the edges between v and its children. For a character $a \in$ nextchars(v), let w be the child of v such that the first character of the label of the edge (v, w) is a. We define $\textsc{Sub}(T, v, a)$ to be the tree obtained by first taking the subtree of T induced by v, w, and all the descendents of w. Furthermore, if the label of (v, w) contains only one character then the vertex v and the edge (v, w) are removed from $\textsc{Sub}(T, v, a)$. Otherwise, the first character of the label of (v, w) is erased.

Let $T_1, \ldots, T_d$ be compressed tries. The *merge* of $T_1, \ldots, T_d$ is a compressed trie whose strings set is the union of the strings sets of $T_1, \ldots, T_d$.

3 k-mismatches index

The following problem is a generalization of the indexing problem that was discussed in the introduction.

Input A compressed trie T over strings $s_1, \ldots, s_n$.

Query A string p, and a location l on T.

Output All the strings s_i such that str(l) is a prefix of s_i and the Hamming distance between p and $s_i[|\text{str}(l)| + 1..|\text{str}(l)| + m]$ is exactly k, where m is the length of p.

A data-structure that solves the problem above is called an *unrooted k-mismatches index*. A data-structure that solves a simpler variant of the problem in which str(l) is always empty is called a *rooted k-mismatches index*. To solve the indexing problem mentioned in the introduction, one can construct a rooted k'-mismatches index on all the suffixes of the input string t for all $k' \leq k$. We note that we use Hamming distance to simplify the presentation. The same techniques can also be used for edit distance.

We first describe the k-mismatches index of Cole et al. [4]. The main idea is to define new compressed tries called *group trees*, and recursively build a rooted $(k-1)$-mismatches index on each group tree (the recursion stops when k is equal to 0). A k-mismatches query on T is answered by making $(k-1)$-mismatches queries on $O(\log n)$ group trees.

Let T be a compressed trie of the strings $s_1, \ldots, s_n$, and let $\mathcal{C}$ be a heavy path decomposition of T. Consider some heavy path $C \in \mathcal{C}$, and let $v_1, \ldots, v_d$ be the vertices along the path C (where v_1 is the topmost vertex in the path). We define *error trees* as follows: For every vertex v_i and every $a \in$ nextchars$(v_i) \setminus \{\text{next}(v_i)\}$, the error tree $\textsc{Err}(T, v_i, a)$ is equal to $\textsc{Sub}(T, v_i, a)$. The error tree $\textsc{Err}(T, v_i)$ is the tree obtained by merging the trees $\textsc{Sub}(T, v_i, a)$ for every $a \in$ nextchars$(v_i) \setminus \{\text{next}(v_i)\}$. Then, if the root u of the resulting tree has more than one child we add a new root u' and an edge (u', u) with label s, where s is the string obtained by concatenating the labels of the edges on the path from v_1 to v_i, and the character

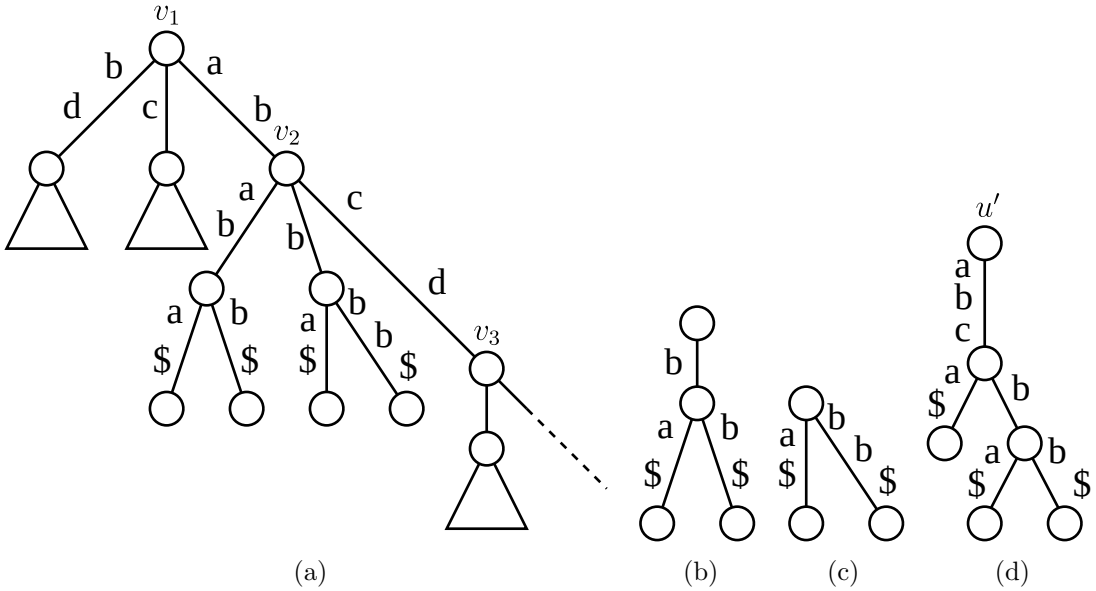

Figure 1: Example of error trees. Figure (a) shows a heavy path $v_1, v_2, \ldots$ and the vertices hanging from this path. The error trees $\textsc{Err}(T, v_2, a)$ and $\textsc{Err}(T, v_2, b)$ are shown in Figures (b) and (c), respectively. Figure (d) shows the error tree $\textsc{Err}(T, v_2)$, which is obtained by merging $\textsc{Err}(T, v_2, a)$ and $\textsc{Err}(T, v_2, b)$, and adding a new root u'.

next(v_i). If u has only one child we prepend the string s to label of the edge between u and its child. See Figure 1 for examples of the definitions above.

The next step is to construct *group trees* from the error trees. Let w_i be the number of leaves in the tree $\textsc{Err}(T, v_i)$. For each vertex v_i we assign an interval $I_i = [\sum_{j<i} w_j, \sum_{j \le i} w_j)$. For an interval $I = [a, b)$, we will denote left$(I) = a$ and right$(I) = b$. The merge of $\textsc{Err}(T, v_i), \ldots, \textsc{Err}(T, v_j)$ will be denoted $\textsc{Group}_1(T, v_i, v_j)$ and will be called *type 1 group tree*. We do not create $\textsc{Group}_1(T, v_i, v_j)$ for all i and j (as this would take too much space). Instead, the type 1 group trees are constructed by the following procedure (an example is given in Figure 2).

For every $C \in \mathcal{C}$ which is not a leaf in $T_{\mathcal{C}}$ **do**
  Let $v_1, \ldots, v_d$ be the vertices of C with intervals $I_1, \ldots, I_d$.
  $L_1 \leftarrow \{(1, d)\}$.
  $t \leftarrow 1$.
  While $L_t \neq \emptyset$ **do**
    $L_{t+1} \leftarrow \emptyset$.
    For every $(i, i') \in L_t$ **do**
      $a \leftarrow$ left(I_i), $b \leftarrow$ right$(I_{i'})$.
      Let j be the index such that $\frac{a+b}{2} \in I_j$.
      If $j \geq i + 1$ **then** build the group tree $\textsc{Group}_1(T, v_i, v_{j-1})$
      Build the group tree $\textsc{Group}_1(T, v_j, v_j)$.
      If $j \leq i' - 1$ **then** build the group tree $\textsc{Group}_1(T, v_{j+1}, v_{i'})$
      If $j > i + 1$ **then** add $(i, j - 1)$ to L_{t+1}.
      If $j < i' - 1$ **then** add $(j + 1, i')$ to L_{t+1}.

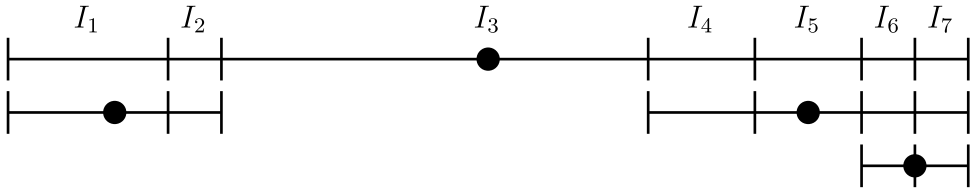

Figure 2: An example of type 1 group tree construction. The top line shows intervals $I_1, \ldots, I_7$ and the point $\frac{a+b}{2} \in I_3$. Thus, the first iteration creates the group trees $\textsc{Group}_1(T, v_1, v_2)$, $\textsc{Group}_1(T, v_3, v_3)$, and $\textsc{Group}_1(T, v_4, v_7)$. In the next iteration, the following trees are created: $\textsc{Group}_1(T, v_1, v_1)$, $\textsc{Group}_1(T, v_2, v_2)$, $\textsc{Group}_1(T, v_4, v_4)$, $\textsc{Group}_1(T, v_5, v_5)$, and $\textsc{Group}_1(T, v_6, v_7)$. In the final iteration, the group trees $\textsc{Group}_1(T, v_6, v_6)$ and $\textsc{Group}_1(T, v_7, v_7)$ are created.

15: $t \leftarrow t + 1$.

For every vertex v in T we create group trees from the error trees $\textsc{Err}(T, v, a)$ in a similar way. These trees will be called *type 2 group trees*. On every group tree (of type 1 or 2) we build a rooted $(k-1)$-mismatches index. Also, we build an unrooted $(k-1)$-mismatches index on T.

We now describe how to answer a rooted query p. This is done by performing $(k-1)$-mismatches queries on some group trees or on T. Let l be the location in T such that $\text{str}(l)$ is a prefix of p, and $|\text{str}(l)|$ is maximal. The *path that corresponds to* p is the path from the root of T to l. Let $C_1, \ldots, C_r$ be the paths of $\mathcal{C}$ through which the path that corresponds to p passes, in order from top to bottom. For $t = 1, \ldots, r$, let l_t be the last location on C_t through which the path that corresponds to p passes. Note that for $t < r$, l_t must be a vertex.

For every path C_t, let $v_1, \ldots, v_d$ be the vertices of the path, and let j be the minimum index such that $|\text{str}(v_j)| \geq |\text{str}(l_t)|$. The following queries are performed:

1. If l_t is not a leaf, do an unrooted $(k-1)$-mismatches query on T with query string $p[|\text{str}(l_t)| + 2..m]$ and start position $\text{nextloc}(l_t)$.

2. Identify the type 1 group trees whose merge includes precisely the error trees $\textsc{Err}(T, v_1), \ldots, \textsc{Err}(T, v_{j-1})$. On each group tree, do a $(k-1)$-mismatches query with query string $p[|\text{str}(v_1)| + 1..m]$.

3. If $l_t = v_j$ and l_t is not a leaf, identify the type 2 group trees whose merge includes precisely the error trees $\textsc{Err}(T, v_j, a)$ for all $a \neq p[|\text{str}(v_j)| + 1]$. On each group tree, do a $(k-1)$-mismatches query with query string $p[|\text{str}(v_j)| + 2..m]$.

Handling an unrooted query is done similarly: In this case the path that corresponds to p starts at the query location l instead of the starting at the root. Handling the paths $C_2, \ldots, C_r$ is the same as before. For the path C_1, the type 1 group trees that are queried are the trees whose merge includes precisely the error trees $\textsc{Err}(T, v_i)$, $\ldots, \textsc{Err}(T, v_{j-1})$, where i is the minimum index such that $|\text{str}(v_i)| \geq |\text{str}(l)|$ and j is defined as before.

4 New index

Our construction is similar to the construction of Cole et al. We build more group trees in order to reduce the number of group trees that are searched when answering a query. In particular, while in the construction of Cole et al. a group tree consists of error trees that come from one heavy path, in our construction some group trees (called type 3 group trees) consist of error trees from several heavy paths.

Let α be some integer with $2 \leq \alpha \leq n/2$. The type 1 group trees are built using procedure Build described below.

1: **For** every $C \in \mathcal{C}$ which is not a leaf in $T_{\mathcal{C}}$ **do**
2: Let $v_1, \ldots, v_d$ be the vertices of C with intervals $I_1, \ldots, I_d$.
3: $L_1 \leftarrow \{(1, d)\}$.
4: $t \leftarrow 1$.
5: **While** $L_t \neq \emptyset$ **do**
6: $L_{t+1} \leftarrow \emptyset$.
7: **For** every $(i, i') \in L_t$ **do**
8: $a \leftarrow \text{left}(I_i)$, $b \leftarrow \text{right}(I_{i'})$.
9: $i_0 \leftarrow i - 1$.
10: **For** $j = 1, \ldots, \alpha - 1$ **do**
11: Let i_j be the index such that $a + \frac{j}{\alpha}(b - a) \in I_{i_j}$.
12: **If** $i_j > i_{j-1}$ **then**
13: **If** $i_j \geq i + 1$ **then** build the group tree $\textsc{Group}_1(T, v_i, v_{i_j - 1})$.
14: Build the group tree $\textsc{Group}_1(T, v_{i_j}, v_{i_j})$.
15: **If** $i_j \leq i' - 1$ **then** build the group tree $\textsc{Group}_1(T, v_{i_j + 1}, v_{i'})$.
16: **If** $i_j > i_{j-1} + 2$ **then** add $(i_{j-1} + 1, i_j - 1)$ to L_{t+1}.
17: **If** $i_{\alpha-1} < i' - 1$ **then** add $(i_{\alpha-1} + 1, i')$ to L_{t+1}.
18: $t \leftarrow t + 1$.

The type 2 group trees are built similarly. We also define *type 3 group trees* as follows. The *weight* of a path $C \in \mathcal{C}$ is the weight of the topmost vertex in C. A path $C' \in \mathcal{C}$ is called *bad* if $\text{weight}(C') > \frac{1}{\alpha}\text{weight}(C)$, where C is the parent of C' in $T_{\mathcal{C}}$. We scan the vertices of the tree $T_{\mathcal{C}}$ in a preorder. When we reach a vertex C that has at least one bad child, we built a set $B(C)$ containing the path C and all paths $C' \in \mathcal{C}$ such that C' is a descendent of C in $T_{\mathcal{C}}$ and $\text{weight}(C') > \frac{1}{\alpha}\text{weight}(C)$. Note that every $C' \in B(C) \setminus \{C\}$ is a bad path.

For every $C', C'' \in B(C)$ such that C'' is a descendent of C' we create a type 3 group tree, denoted $\textsc{Group}_3(T, C', C'')$, in the following way. Let $C' = C_1, C_2, \ldots, C_{r-1}, C_r = C''$ be the path from C' to C'' in $T_{\mathcal{C}}$. Let u_i be the first vertex in the path C_i, and for $i < r$ let v_i be the parent of u_{i+1} in T (note that $v_i \in C_i$). Let c_i be the first character of the label of the edge (v_i, u_{i+1}). Let s_i be the concatenation of the labels of the edges on the path from u_1 to u_i, and let s'_i be the concatenation of the labels of the edges on the path from u_1 to v_i, and the character c_i. The group tree $\textsc{Group}_3(T, C', C'')$ is the merge of the following trees.

1. For every $i < r$ and every $v \in C_i$ which is an ancestor of v_i, the tree obtained by taking $\textsc{Err}(T, v)$ and prepending the string s_i to the label of the edge

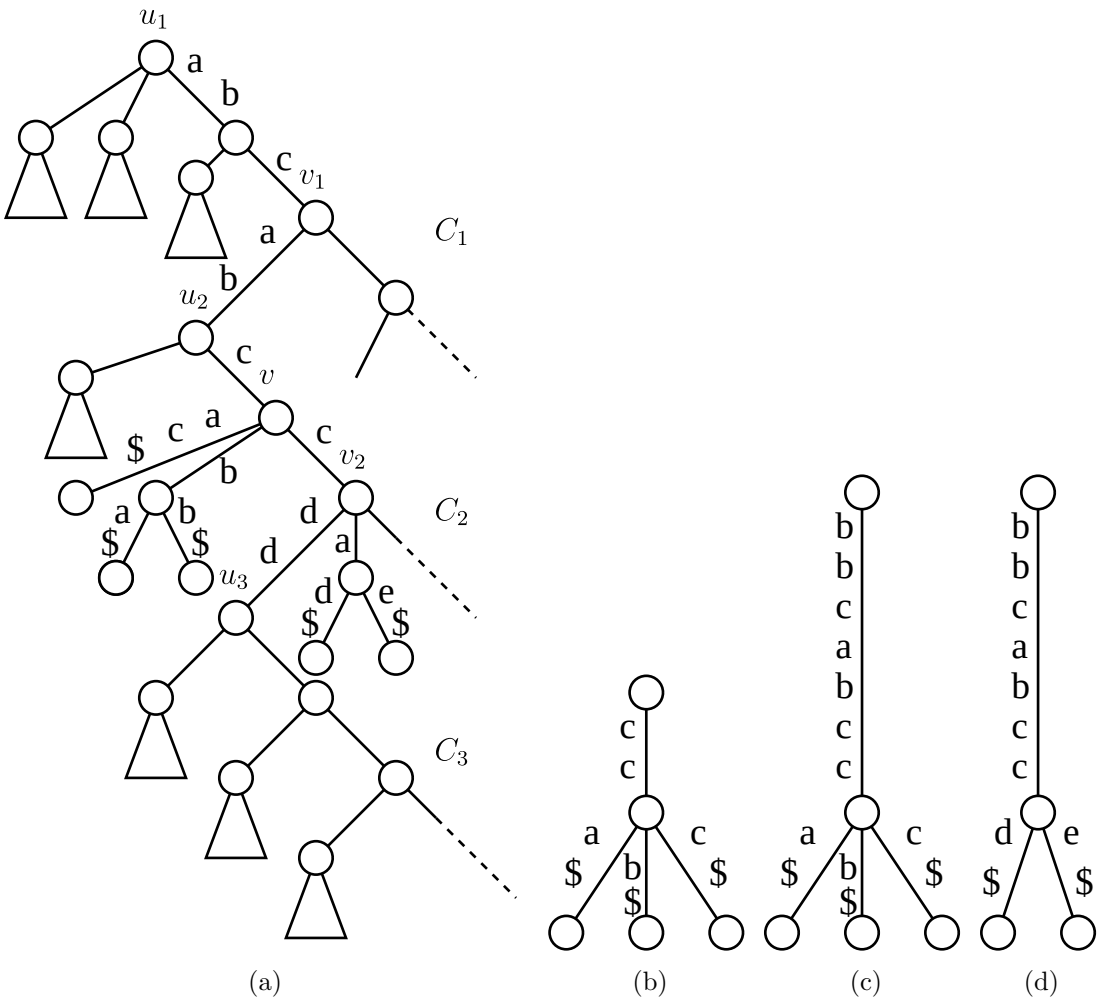

Figure 3: Example of type 3 group trees. The paths $C' = C_1$, C_2, and $C'' = C_3$ are shown in Figure (a). Two of the trees that are merged when creating $\textsc{Group}_3(T, C', C'')$ are shown in (c) and (d). The tree in (c) is obtained from $\textsc{Err}(T, v)$ (shown in (b)) by adding the string $s_2 = \text{abcab}$ to the label of the edge between the root and its child. The tree in (d) is obtained from $\textsc{Sub}(T, v_2, a)$ by adding a new root, where the label of the new edge is $s'_2 = \text{bbcabbc}$.

between the root of $\textsc{Err}(T, v)$ and its only child.

2. For every $i < r$ and every $a \in \text{nextchars}(v_i) \setminus \{c_i\}$ (note that this includes $a = \text{next}(v_i)$), the tree obtained by taking $\textsc{Sub}(T, v_i, a)$ and if the root of this tree has only one child, prepending the string s'_i to the edge between the root and its child. Otherwise, a new root is added and connected to the old root by an edge, where the label of the edge is s'_i.

An example is given in Figure 3.

Answering an unrooted query p is performed as follows. Let $C_1, \ldots, C_r$ be the paths of $\mathcal{C}$ through which the path that corresponds to p in T passes. Start with $t = 1$. At each iteration, if $t = r$ or C_{t+1} is not a bad path, perform queries for C_t as described in the previous section, and increase t by 1. Otherwise, do a rooted

$(k-1)$-mismatches query on $\textsc{Group}_3(T, C_t, C_{t'})$ and set t to t', where $t' > t$ is the maximum index such that $C_{t'} \in B(C_t)$. In more details, the algorithm is as follows (we omit the queries on type 2 group trees which are handled similarly to the queries on type 1 group trees).

Let $C_1, \ldots, C_r$ be the paths of $\mathcal{C}$ through which the path that corresponds to p in T passes.
$t \leftarrow 1$.
While $t \leq r$ **do**
  Let $v_1, \ldots, v_d$ be the vertices of C_t, with intervals $I_1, \ldots, I_d$.
  If $t < r$ and C_{t+1} is a bad path
    Let $t' > t$ be the maximum index such that $C_{t'} \in B(C_t)$.
    Do a rooted $(k-1)$-mismatches query on $\textsc{Group}_3(T, C_t, C_{t'})$ with query string $p[|\text{str}(v_1)| + 1..m]$.
    $t \leftarrow t'$.
  Else
    Let l_t be the last location on C_t through which the path that corresponds to p passes.
    If l_t is not a leaf **then** do an unrooted $(k-1)$-mismatches query on T with query string $p[|\text{str}(l_t)| + 2..m]$ and start position $\text{nextloc}(l_t)$.
    Let j be the minimum index such that $|\text{str}(v_j)| \geq |\text{str}(l_t)|$.
    $p' \leftarrow p[|\text{str}(v_j)| + 1..m]$.
    $i \leftarrow 1$, $i' \leftarrow d$.
    While $i < j$ **do**
      $a \leftarrow \text{left}(I_i)$, $b \leftarrow \text{right}(I_{i'})$.
      Let β be the maximum integer such that $a + \frac{\beta}{\alpha}(b-a) < \text{right}(I_j)$.
      If $\beta > 0$ **then** let j_1 be the index such that $a + \frac{\beta}{\alpha}(b-a) \in I_{j_1}$ **else** $j_1 \leftarrow i - 1$.
      If $\beta < \alpha - 1$ **then** let j_2 be the index such that $a + \frac{\beta+1}{\alpha}(b-a) \in I_{j_2}$ **else** $j_2 \leftarrow i' + 1$.
      If $j_1 \geq i+1$ **then** do a rooted $(k-1)$-mismatches query on $\textsc{Group}_1(T, v_i, v_{j_1-1})$ with query string p'.
      If $i \leq j_1 < j$ **then** do a rooted $(k-1)$-mismatches query on $\textsc{Group}_1(T, v_{j_1}, v_{j_1})$ with query string p'.
      $i \leftarrow j_1 + 1$, $i' \leftarrow j_2 - 1$.
    $t \leftarrow t + 1$

For an unrooted query, the path C_1 is handled as in the handling of unrooted queries described in the previous section. Then, $C_2, \ldots, C_r$ are handled using the algorithm above.

Theorem 1. *The time for answering a query is* $O(m + (\log_\alpha n)^k \log\log n + \#matches)$.

Proof. Let $t_1, \ldots, t_{r'}$ be the different values of t during the run of the algorithm. We first give a bound on r'. We claim that for every $i \leq r' - 2$, $\text{weight}(C_{t_{i+2}}) \leq \frac{1}{\alpha}\text{weight}(C_{t_i})$: If C_{t_i+1} is not a bad path then $t_{i+1} = t_i + 1$ and $\text{weight}(C_{t_{i+1}}) \leq \frac{1}{\alpha}\text{weight}(C_{t_i})$. Since $\text{weight}(C_1) > \text{weight}(C_2) > \cdots > \text{weight}(C_t)$ and $t_{i+2} \geq t_{i+1}$,

we obtain that $\text{weight}(C_{t_{i+2}}) \leq \frac{1}{\alpha}\text{weight}(C_{t_i})$. If C_{t_i+1} is a bad path then $C_{t_{i+1}+1}$ is not in $B(C_t)$. Therefore, $\text{weight}(C_{t_{i+2}}) \leq \text{weight}(C_{t_{i+1}+1}) \leq \frac{1}{\alpha}\text{weight}(C_{t_i})$.

Since $\text{weight}(C_1) = n$ and $\text{weight}(C_t) \geq 1$, we conclude that $r' \leq 2 + 2\log_\alpha n$. Therefore, the number of $(k-1)$-mismatches queries performed at lines 7 and 11 is at most $r' \leq 2 + 2\log_\alpha n$.

We next bound the number of queries performed on type 1 group trees. During the execution of lines 15–22, we say that the *current interval* is the interval $I_i \cup I_{i+1} \cup \cdots \cup I_{i'}$. The sequence of current intervals during the execution of the algorithm (for all t) is decreasing in lengths. If for some C_t, lines 15–22 are executed s times, then the length of the current interval decreases by a factor of at least $\alpha^{\max(1,s-1)}$. Thus, lines 15–22 are executed at most $2 + 2\log_\alpha n$ times, and the number queries performed on type 1 group trees is at most $4 + 4\log_\alpha n$. Using similar analysis, the number of queries on type 2 group trees is at most $8 + 8\log_\alpha n$ (in each iteration of the search in the type 2 group trees, up to 4 queries can be made).

Combining the bounds above, we have that the total number of $(k-1)$-mismatches queries performed when answering a rooted queries is at most $14 + 14\log_\alpha n$. When answering an unrooted query, at most $18 + 18\log_\alpha n$ $(k-1)$-mismatches queries are made (the additional $4 + 4\log_\alpha n$ queries are due to the special handling of the path C_1). Using induction, the total number of 0-mismatches queries performed for a rooted or unrooted query is at most $(18 + 18\log_\alpha n)^k = O((\log_\alpha n)^k)$.

Using the LCP data-structures of Cole et al. [4] we have that after a preprocessing stage that takes $O(m)$ time, the i-th 0-mismatches query takes $O(\log\log n + \#\text{matches}_i)$ time, where $\#\text{matches}_i$ is the number of matches returned by the query. Since each approximate match of p in t is reported exactly once, $\sum_i \#\text{matches}_i = \#\text{matches}$. Therefore, the total time complexity of a k-mismatches query is $O(m + (\log_\alpha n)^k \log\log n + \#\text{matches})$. ∎

Theorem 2. *The space complexity of the index is $O(n(\alpha \log\alpha \log n)^k)$.*

Proof. First, we bound the total number of leaves in all type 1 group trees (the analysis is similar to the analysis of Cole et al.). Define $S_k(n) = (5\alpha \log\alpha \log n)^k$. We will show that the total number of leaves in all group trees that are built for a k-mismatches index over a compressed trie T with n leaves is at most $S_k(n) \cdot n$. The claim is proved using induction on k. The base $k = 0$ is trivial.

Suppose we proved the claim for $k-1$, and consider some k-mismatches index over a compressed trie T with n leaves. Let $T_1, \ldots, T_d$ be all the type 1 group trees that are built for T by procedure Build, and denote by x_i the number of leaves in T_i. By induction, we have that the $(k-1)$-mismatches indices constructed on the trees $T_1, \ldots, T_d$ have at most $\sum_{i=1}^{d} S_{k-1}(x_i) \cdot x_i$ leaves.

For a leaf v of T, let $i(v,1), \ldots, i(v,d_v)$ denote the indices of group trees in which v appears. Clearly, $\sum_{i=1}^{d} S_{k-1}(x_i) \cdot x_i = \sum_v \sum_{j=1}^{d_v} S_{k-1}(x_{i(v,j)})$. The function $S_{k-1}(x)$ is an increasing function of x. Therefore, $\sum_{i=1}^{d} S_{k-1}(x_i) \cdot x_i \leq \sum_v \sum_{j=1}^{d_v} S_{k-1}(n) = S_{k-1}(n) \sum_v d_v$.

We now give a bound on d_v. Fix some leaf v of T. We partition the group trees that contain v into sets, where each set consists of all the trees that are generated during one execution of lines 10–16 of procedure Build. In each set the number of trees that contain v is at most $\alpha - 1$. Similarly to the proof of Theorem 1, the

number of sets is at most $\log n + \log_\alpha n \leq 2 \log n$. It follows that the number of leaves in the $(k-1)$-mismatches indices built on the type 1 group trees is at most $(\alpha - 1) \cdot 2 \log n \cdot S_{k-1}(n)$. Similarly, the number of leaves in the indices built on the type 2 group trees is at most $(\alpha - 1) \cdot 2 \log n \cdot S_{k-1}(n)$.

It remains to bound the number of leaves in the indices built on the type 3 group trees. We begin by bounding the size of $B(C)$ for some path C. Consider the subtree T' of $T_{\mathcal{C}}$ that is induced by the vertices of $B(C)$. For every two leaves C_1 and C_2 in T', the set of vertices of T that are descendents of the topmost vertex in C_1 is disjoint with the set of vertices of T that are descendents of the topmost vertex in C_2. It follows that the sum of weights of the leaves of T' is less than or equal to weight(C). Since each leaf in T' has weight greater than $\frac{1}{\alpha}$weight(C), we conclude that T' has at most α leaves. By the definition of heavy path decomposition, we have that if C_1 is a child of C_2 in T' then the weight of C_1 is less than half the weight of C_2. Therefore, for every leaf C' in T', the number of ancestors of C' in T' is at most $\log \alpha$. Thus, $|B(C)| \leq \alpha \log \alpha$.

Using the same arguments as above, the number of leaves in the $(k-1)$-mismatches indices built on the type 3 group trees is at most $S_{k-1}(n) \sum_v d'_v$, where d'_v is the number of type 3 group trees that contain the leaf v. A type 3 group tree that contains v must be of the form $\textsc{Group}_3(T, C', C'')$ where C' is a path through which the path from the root of T to v passes. The number of such paths is at most $\log n$. Moreover, for fixed C', there are at most $\alpha \log \alpha$ ways to choose C''. Therefore, $d'_v \leq \alpha \log \alpha \log n$.

We conclude that the total number of leaves in the indices built on all group trees is at most

$$(2 \cdot 2(\alpha - 1) \log n + \alpha \log \alpha \log n) \cdot S_{k-1}(n) \leq 5\alpha \log \alpha \log n \cdot S_{k-1}(n) = S_k(n). \quad \blacksquare$$

References

[1] A. Amir, D. Keselman, G. M. Landau, N. Lewenstein, M. Lewenstein, and M. Rodeh. Dictionary matching with one error. *J. 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 *Proc. 8th European Symposium on Algorithms (ESA)*, pages 120–131, 2000.

[3] H. Chan, T. W. Lam, W. Sung, S. Tam, and S. Wong. A linear size index for approximate pattern matching. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 49–59, 2006.

[4] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on Theory Of Computing (STOC)*, pages 91–100, 2004.

[5] C. Epifanio, A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Languages with mismatches. *Theoretical Computer Science*, 385(1-3):152–166, 2007.

[6] A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Indexing structures for approximate string matching. In *Proc. 5th Italian Conference on Algorithms and Complexity (CIAC)*, pages 140–151, 2003.

[7] T. N. D. Huynh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proc. 15th Symposium on Combinatorial Pattern Matching (CPM)*, pages 434–444, 2004.

[8] T. W. Lam, W. K. Sung, and S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proc. 16th International Symposium on Algorithms and Computation (ISAAC)*, pages 339–348, 2005.

[9] M. G. Maaß and J.Nowak. Text indexing with errors. In *Proc. 16th Symposium on Combinatorial Pattern Matching (CPM)*, pages 21–32, 2005.

[10] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000.

[11] G. Navarro and E. Chávez. A metric index for approximate string matching. *Theoretical Computer Science*, 352(1–3):266–279, 2006.

[12] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.