Improved space-time tradeoffs for approximate full-text indexing with one edit error

Djamal Belazzougui

LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France, dbelaz@liafa.jussieu.fr

January 14, 2013

Abstract

In this paper we are interested in indexing texts for substring matching queries with one edit error. That is given a text \( T \) of \( n \) characters over an alphabet of size \( \sigma \), we are asked to build a data structure that answers the following query: find all the \( \text{occ} \) substrings of the text which are at edit distance at most 1 from a string \( q \) of length \( m \). In this paper we show two new results for this problem. The first result suitable for arbitrary alphabet size \( \sigma \) uses \( O(n(\log \varepsilon n + \log \sigma)) \) words of space and answers to queries in time \( O(m + \text{occ}) \). This improves simultaneously in space and time over the result of Cole et Al \cite{8}. The second result suitable only for constant alphabet relies on compressed indexes and comes in two variants: the first variant uses \( O(n \log \varepsilon n) \) bits of space (where \( \varepsilon \) is any constant such that \( 0 < \varepsilon < 1 \)) and answers to queries in time \( O(m + \text{occ}) \) while the second variant uses \( O(n \log \log n) \) bits of space and answers to queries in time \( O((m + \text{occ}) \log \log n) \). This second result improves on the previously best results for constant alphabets achieved in Lam et Al \cite{16} and Chan et Al \cite{6}.

1 Introduction

The problem of approximate string matching with over texts was intensively studied. The problem consists in given a pattern \( q \), a text \( T \) (the characters of \( T \) and \( q \) are drawn from the same alphabet of size \( \sigma \)) and a parameter \( k \), to find all the substrings of \( T \) which are at distance at most \( k \) from \( q \). There exists many different distances which can be used for this problem. In this paper, we are interested in the edit distance in which the distance between two strings \( x \) and \( y \) is defined as the minimal number of edit operations needed to transform \( x \) into \( y \) where the considered edit operations are deletion of a character, substitution of a character by another and finally insertion of a character at some position in the string. Generally two flavors of the problem are considered: the online variant and the indexed variant. In the online variant, we assume that we know

\[\text{occ} \] This work is partially supported by the french ANR project MAPPI (project number ANR-2010-COSI-004).
the pattern in advance and the text arrives character by character. In our case we are interested in the offline version in which we can preprocess the text in advance so that we can efficiently answer to queries which consist only in the pattern and a parameter $k$. Further, we restrict our interest to the case $k = 1$.

### 1.1 Related work

We now mention the best results from the literature we are aware of for our problem. We only consider the results with worst case space and time bounds. We thus do not consider results like the result in [17] in which either the query time or the space usage only hold on average on the assumption that the text and/or the patterns are drawn from some random distribution. For general integer alphabet a result by Amir et Al [1] further improved by Buchsbaum et Al [4] has led to $O(n \log^2 n)$ space with query time $O(m \log \log m + occ)$. Later Cole et Al [8] described an index for arbitrary number of errors $k$ but which for the case $k = 1$ uses $O(n \log^2 n)$ bits of space and answers to queries in $O(m + \log n \log \log n + occ)$ time. For the special case of constant sized alphabets a series or results culminated with the results of Lam et Al [16], Chan et Al [5] and Chan et Al [6] with various tradeoffs between the occupied space and query time. By adapting some ideas of [2] and combining with two indices described in [5] we are able to essentially remove the additive polylogarithmic term from the query times associated with some of the best previously known results while using the same space (or even less in some cases). The reader can refer to tables 1 and 2 for a full comparison between our new results and the old ones.

As can be seen both our indexes improve on the state of the art. We should mention that the results in the table attributed to [16] are not stated in that form, but can be easily deduced from the main result in [16] by using different...
compressed text index implementations. The results in the first table are all unsuitable for large alphabets as their query times all have a hidden linear dependence on $\sigma$. That means that for very large alphabets of size $\sigma = \Theta(\sqrt{n})$ or of size $\sigma = 2^{\Theta(\log n)}$ for example, the query time of those algorithms will be unreasonable $\Omega(\sqrt{n})$ or $\Omega(2^{\sqrt{\log n}})$. By contrast the query times of the algorithms in the second table do not have any dependence on the alphabet size. Our result in the second table always dominates the Cole et Al result for both space and time. Only in case of very large alphabet (e.g $\sigma = \Theta(n)$ or $\sigma = \Theta(\sqrt{n})$) and long pattern length $m = \Omega(\log n \log \log n)$ will both have the same space and time. In the other cases, our data structure will dominate. For example in case $\sigma = 2^{\Theta(\log n)}$, our data structure will use space $O(n \log^{1.5} n)$ bits while the data structure of Cole et Al still uses $O(n \log^2 n)$ bits. Another example is the case $m \leq \log n$ where the query time of Cole et Al will be $\Omega(m \log \log n + \text{occ})$ while our query time will still be just $O(m + \text{occ})$.

2 Preliminaries and outline of the results

At the core of our paper is a result for indexing all substrings of a text $T$ of $n$ characters bounded by some given length $b$. In particular, we show the following two theorems:

**Theorem 1** For any text $T$ of length $n$ characters over an alphabet of fixed size, given a parameter $b$, we can build an index of size $O(n(b^\varepsilon + \log^2 n))$ bits (where $\varepsilon$ is any constant such that $0 < \varepsilon < 1$) so that for any given string $q$ of length $m < b$ we can report all of the occ substrings of the text which are at edit distance 1 from $q$ in time $O(m + \text{occ})$. Alternatively we can build a data structure which occupies $O(n \log b + \log \log n)$ bits of space and which answers to queries in time $O((m + \text{occ})(\log b + \log \log n))$.

The second theorem is based on Cole et Al approach [8] combined with some ideas used in the first theorem and with two recent results of [3] and [14].

**Theorem 2** For any text $T$ of length $n$ characters over an alphabet of size $\sigma$, given a parameter $b$, we can build an index of size $O(n \log \sigma (b^\varepsilon + \log \sigma))$ bits (where $\varepsilon$ is any constant such that $0 < \varepsilon < 1$) such that for any given string $q$ of length $m < b$ we can report all of the occ substrings of the text which are at edit distance 1 from $q$ in time $O(m + \text{occ})$.

The first theorem gives an immediate improvement for constant sized alphabets when combined with a result appearing in [5]:

**Theorem 3** For any text $T$ of length $n$ over an alphabet of size $\sigma$ we can build the following indices which are able to return for any query string $q$ the occ occurrences of substrings of $T$ which are at edit distance 1 from $q$:
• An index which assumes that $\sigma = O(1)$ and which occupies $O(n \log^\varepsilon n)$ bits of space and answers to queries in time $O(m + occ)$ where $\varepsilon$ is any constant such that $0 < \varepsilon < 1$.

• An index which assumes that $\sigma = O(1)$ and which occupies $O(n \log \log n)$ bits of space and answers to queries in time $O((m + occ) \log \log m)$.

The second theorem can also be combined with another result which has appeared in [5] to get the following result suitable for arbitrary alphabet sizes:

**Theorem 4** For any text $T$ of length $n$ over an alphabet of size $\sigma$ we can build an index which occupies using only $O(n (\log \varepsilon n + \log \sigma))$ bits of space and able to return for any query string $q$ the $occ$ occurrences of substrings of $T$ which are at edit distance 1 from $q$ in time $O(m + occ)$.

The second theorem can be used for any alphabet size while the first one holds only for fixed alphabet. Our two theorems can be used only for matching strings of bounded length but can provide an improvement when used in combination with previous result which are efficient only for long strings.

Our new methods for proving theorems [1] and [2] makes use of some ideas introduced in [2] combined with tools which recently proposed in [3] and [14]. In [2] a new dictionary for approximate queries with one error was proposed. For a string of length $m$ it achieved $O(m + occ)$ query time while at the same time using optimal space (up to constant factors). A naive use of that dictionary to the problem of full-text indexing was also proposed in that paper. However while this leads to the same $O(m + occ)$ query time achieved in this paper, the space usage was too large namely $O(n (\log n \log \log n)^2 \log \sigma)$ bits of space for alphabet of size $\sigma$. Nonetheless we will borrow some ideas from that paper and use them to prove our main results.

The paper is organized as follows: we first begin with the data structure suitable for constant sized alphabets (theorems [1] and [3]) in section 3 before showing the data structure for large alphabets (theorems [2] and [4]) in section 4. We finally conclude the paper by mentioning some open problems in section 5.

### 2.1 Model and notation

In the remaining we note by $\overline{x}$ the reverse of string $x$. That is $\overline{x}$ is the string $x$ written in reverse order. For a given string $s$, we note by $s[i, j]$ or by $s[i..j]$ the substring of $s$ spanning the characters $i$ through $j$. We assume that the reader is familiar with the trie concept and with classical text indexing data structures like suffix trees and suffix arrays. The model assumed in this paper is the word RAM model with word length $w = \Theta(\log n)$ where $n$ is the size of the considered problem. We further assume that standard arithmetic operations including multiplications can be computed in constant time. We assume that our text $T$ to indexed is of length $n$ and its alphabet is of size $\sigma < n$.

\footnote{Throughout the paper we always assume that $\sigma < n$, as otherwise we can store an auxiliary hash based dictionary with space usage $O(n \log \sigma)$ and which stores all characters appearing
2.2 Basic definitions

We now briefly recall some basic text indexing data structures.

**Suffix array** A suffix array \([18]\) (denoted \(SA[1..n]\)) built on a text \(T\) of length \(n\) just stores the pointers to the suffixes of \(T\) in sorted order (where the order is the usual lexicographic order). Clearly a suffix array occupies \(n \log n\) bits.

**Suffix tree** A suffix tree \([20]\) is a specially built tree on a text \(T\) where:

- Every suffix of \(T\) is associated with a leaf in the tree.
- A factor \(p\) of \(T\) is associated with an internal node in the tree iff there exists two characters \(a\) and \(b\) such that \(pa\) and \(pb\) is also a factor of \(T\).
- The subtree rooted at any internal node associated with a prefix \(p\) contains all the suffixes of \(T\) which have \(p\) as a prefix.
- An edge connecting an internal node associated with a factor \(p\) to a node (which could be a leaf or an internal node) associated with a string \(s = ps'\) (which could either be a suffix or another factor) will be labeled with the string \(s'\).

For more detailed description of the suffix array or the suffix tree, the reader can refer to any book on text indexing algorithms. The essential property of a suffix tree is that it can be implemented in such a way to occupies \(O(n)\) pointers (that is \(O(n \log n)\) bits) in addition to the text and that given any factor \(p\) of \(T\) it is possible to find all the suffixes of \(T\) which are prefixed by \(p\) in \(O(|p|)\) time. A suffix tree can also be augmented in several ways so as to support many other operations, but in this paper we will use very few of them.

3 Data structure for constant sized alphabets

In this section we give a proof of theorems 1 and 3. For that we first begin by describing the data structures used in theorem 1 in section 3.1, then describe how the queries are executed on those data structures in section 3.1 which concludes demonstration of theorem 1. Finally theorem 3 is proved in section 3.3.

3.1 Data structure for short patterns

Our data structure for short patterns relies on a central idea used in [2]. The idea was that of using a hash based dictionary, finding all the strings in the dictionary which are at distance 1 from a pattern \(q\) of length \(m\) can be done in \(O(1)\) time. This stems from two facts:

in \(T\) then at query time if we meet that character at some position \(i\) in the pattern we know that the error occurs at that position and we can easily find the matching of the pattern using the indexes of this paper, by considering substitution or deletion of that character.
1. If the dictionary use some suitable hash function $H$ then after we have done a preprocessing step in $O(m)$ time, the computation of $H(p)$ takes constant time for any string $p$ at distance 1 from $q$.

2. Using a trie and reverse trie, we can verify any matching in constant time (this idea was used many times before).

In our case we will use different techniques. As we are searching in a text rather than a dictionary we will be looking for suffixes prefixed by some string $p$ instead of finding exact matching entries in a dictionary. For that reason we will use a weak prefix search data structure which while still using reasonable space, will permit us to look for matching suffixes in very fast time.

We now describe more in detail the data structure we use for the matching of patterns of bounded length over small alphabets (theorem 1). This data structure uses the following components:

1. A suffix array $SA$ built on the text $T$.

2. A suffix tree $S$ built on the text $T$. In each node of the suffix tree representing a factor $p$ of $T$ we store the range of suffixes which start with $p$. That is we store a range $[i, j]$ such that any suffix starts with $p$ iff its rank $k$ in lexicographic order is included in $[i, j]$.

3. A reverse suffix tree $S$ built on the text $T$ the reverse of the text $T$ (we could call $S$ a prefix tree as it actually stores prefixes of $T$). In each node of $S$ representing a factor $p$ we store the range of suffixes of $T$ which start with $p$. That is we store a range $[i, j]$ such that any suffix of $T$ starts with $p$ iff its rank $k$ in lexicographic order relatively to all other suffixes of $T$ is included in $[i, j]$. This is equivalent to say that any prefix of $T$ ends with $p$ iff its rank $k$ in lexicographic order relatively to all other prefixes of $T$ is included in $[i, j]$.

4. A table $SA^{-1}[1..n]$. This table stores for each suffix $T[i..n]$ for all $1 \leq i \leq n$ the rank of the suffix $T[i..n]$ in lexicographic order relatively to all other suffixes of $T$.

5. A table $PA^{-1}[1..n]$. This table stores for each prefix $T[1..i]$ the rank of the prefix $T[1..i]$ in lexicographic order relatively to all the other prefixes of $T$.

6. A polynomial hash function $H$ parametrized with a prime $P > n^3 \sigma$ and an integer $r$ (a seed). For a string $x$ we have $H(x) = x[1] \cdot r + x[2] \cdot r^2 ... x[|x|] \cdot r^{|x|}$. The details of the construction are described below. The hash function essentially uses just $O(\log n)$ bits of space to store the numbers $P$ and $r$.

7. A constant time weak prefix search data structure (which we denote by $W_0$) built on the set $U$, the set of substrings (factors) of $T$ of fixed length $b$ characters to which we add a set of $b - 1$ artificial strings of length $b$).

6
Note that $|U| \leq n$. This data structure which is described in \[3\] comes in two versions one of which uses $O(n(b^\varepsilon + \log \log \sigma))$ bits of space for any constant $0 < \varepsilon < 1$ and answers to queries in $O(1)$ time. We note that query time by $t_{W_0}$. Details are described below.

8. Finally a prefix sum data structure $V_0$ which stores for every $p \in U$ sorted in lexicographic order, the number of suffixes of $T$ prefixed by $p$ (for each of the artificial strings this number is set to one). This prefix sum data structure uses $O(|U|) = O(n)$ bits of space and can be queried in constant time.

Note that the total space usage is dominated by the indexing data structures $(SA, SA^{-1}, PA, \text{the suffix and prefix trees})$ which occupy $O(n \log n)$ bits of space.

**Text indexing data structures** The only operation we need to do on the prefix tree is for a given pattern $q$, to determine for each prefix $p$ of $q$ the range of all prefixes of $T$ which have $p$ as a suffix. Similarly for the suffix tree we only need to know for each suffix $s$ of $q$ the range of suffixes which are prefixed by $s$.

The classical representation for our text indexing data structures $SA, SA^{-1}, PA$, the suffix and prefix tree all occupy $O(n \log n)$ bits of space. However in our case for representing the text indexing data structures, we need to go below $O(n \log n)$ bits used by the classical representations. We thus will make use of compressed representations of the text indexing data structures [11, 12]. In particular we need only to have the following results:

- For every prefix $p_i = q[i|q| - i + 1, |q|]$ of length $i$ determine the range $[pl_i, pr_i]$ of prefixes of $T$ which are suffixed by $p_i$. This can be accomplished incrementally in $O(m)$ time by following the suffix links in the prefix tree $S$. That is deducing the range corresponding to the prefix of $q$ of length $i$ from the range of the prefix of $q$ of length $i+1$ in $O(1)$ time (following a suffix link at each step takes $O(1)$ time). In the context of compressed data structures, this can be accomplished using the backward search on the compressed representation of the prefix array $PA$ still in time $O(m)$ and representing $PA$ in $O(n)$ bits only (assuming constant alphabet). In this case the range corresponding to the prefix of $p$ of length $i + 1$ is deduced from the range corresponding to the prefix of $p$ of length $i$.

- For every suffix $s_i = q[i|q| - i + 1, |q|]$ of length $i$ determine the range $[sl_i, sr_i]$ of suffixes of $T$ which are prefixed by $s_i$. This can be done in a similar way in total $O(m)$ time by either following suffix links in standard representation of the suffix tree $S$ or by backward search in a compressed representation of the suffix array $SA$. The compressed representation occupies $O(n)$ bits only.

- For any $i$ we need to have a fast access to $SA[i], SA^{-1}[i], PA[i], PA^{-1}[i]$. In case those four tables are represented explicitly in $O(n \log n)$ bits of
space the access time is trivially $O(1)$. However in the context of compressed representation, we need to use less than $O(n \log n)$ bits of space and still be able to have fast access to the arrays.

The first two results can be summarized with the following lemma:

**Lemma 1** [11] Given a text $T$ of length $n$ over constant sized alphabet we can build a data structure with $O(n)$ bits of space such that given a pattern $q$ of length $m$:

- We can in $O(m)$ time determine the range of suffixes of $T$ prefixed by $s_i$ for all suffixes $s_i$ of length $i \in [1..m]$.
- We can in $O(m)$ time determine the range of prefixes of $T$ suffixed by $p_i$ for all prefixes $p_i$ of length $i \in [1..m]$.

If the alphabet is non constant, then we can obtain the same results using $O(n \log n)$ bits of space.

The third needed text indexing result is summarized with the following lemma:

**Lemma 2** [12, 19, 15] Assuming a constant alphabet size, we can compress the array $PA, SA, PA^{-1}$ and $SA^{-1}$ with the following tradeoffs:

- A representation in $O(n \log \log n)$ bits with access time $t_{SA} = O(\log \log n)$ time.
- A representation in $O(n \log^2 n)$ bits with access time $t_{SA} = O(1)$ time.

**Weak prefix search data structure** A weak prefix search data structure built on a set of strings $U$ permits given a prefix $p$ of any element in $U$ to return the range of elements of $U$ prefixed by $p$ in lexicographic order. If given an element which is not prefix of any element in $U$, it returns an arbitrary range. This weak prefix search data structure needs to use a hash function $H$ and assumes that after preprocessing a query string $p$, the computation of $H(p[1, |i|])$ for any $i$ takes constant time. It also assumes that every $H(p[1, |i|])$ is distinct for all $p \in U$ and all $i \in [1, b]$. In our case we will use the following time/space tradeoffs described in [3]:

**Lemma 3** Given a set of $n$ strings of fixed length $b$ each over alphabet $\sigma$, we can have a weak prefix search data structure with the following tradeoffs:

- Query time $t_{W} = O(c)$ with a data structure which uses $O(n(b^{1/c} \log b + \log \log \sigma))$ bits of space for any constant $c$.

---

2Actually the result in [3] states a space usage $O(n b^{1/c} \log b)$ but assumes a constant alphabet size. However, it is easy to see that the same data structure just works for arbitrary $\sigma$ in which case it uses $O(n(b^{1/c} \log b + \log \log \sigma))$ bits of space.
• Query time \( t_W = O(\log b) \) with a data structure which uses \( O(n(\log b + \log \log \sigma)) \) bits of space.

The query time assumes that the computation of \( H(p') \) for any prefix \( p' \) of \( p \) takes \( O(1) \) time, where \( p \) is the query string and \( H \) is the hash function used by the weak prefix search data structure.

In our case, the weak prefix search data structure will be built on the set \( U \), the set of factors of \( T' = T \#^{b-1} \) (that is the text \( T \) concatenated with \( b-1 \) times character \( \# \) where \( \# \) is a special character absent from initial alphabet and which is lexicographically smaller than all other characters in the alphabet) of fixed length \( b \) (or \( b \log \sigma \) bits). Note that the set \( U \) contains exactly the set of factors of \( T \) of length \( b \) to which we add \( b-1 \) artificial strings which are obtained by appending \( \#^{b-i} \) to every suffix of \( T \) of length \( i < b \).

**Hash function** In the hash function \( H \) the parameter \( P \) is fixed but the seed \( r \) is chosen randomly. The goal is to build a hash function \( H \) such that all the hash values of the substrings of \( T \) used by the weak prefix search are all distinct. If not the case we randomly choose a new \( r \) and repeat the construction until all the needed substrings of \( T \) are mapped to distinct hash values.

**Prefix-sum data structure** A prefix-sum data structure which permits to succinctly encode an array \( A[1..n] \) of integers of total sum \( D \) in space \( n(2 + \lceil \log(D/n) \rceil) \) bits, so that the sum \( \sum_{1 \leq j \leq i} A[j] \) for any \( i \) can be computed in constant time. This can be obtained by combining fast indexed bitvector implementations [13, 7] with Elias-Fano coding [9, 10].

### 3.2 Queries

**Preprocessing** To make a query on our full-text index for a string \( q \) of length \( m \), we will proceed in a preprocessing step which takes \( O(m) \) time. The preprocessing consists in the following phases:

1. Compute the arrays \( L[0..|q|] \) and \( R[1..|q|+1] \) by traversing the suffix tree \( S \) for the string \( q \) and the prefix tree \( \overline{S} \) for the string \( \overline{q} \). Initially \( L[0] = R[|q| + 1] = [1, n] \) and then \( L[i] \) stores the range of prefixes suffixed by \( q[1..i] \) and \( R[i] \) stores the range of suffixes prefixed by \( q[i, m] \) (the range \([1, n]\) is naturally associated with the empty string as the empty strings \( q[1..0] \) or \( q[m+1..m] \) is suffix and prefix of any other string). This step takes always \( O(m) \) whatever the suffix array implementation we use.

2. We precompute an array which stores all the values of \( r^i \) for all \( 0 \leq i \leq m \).

3. We precompute all the values \( H(q[1, i]) \) for all \( 1 \leq i \leq m \). That is all the hash values for all the prefixes of \( q \). This can easily be done incrementally as we have \( H(q[1, i]) = q[1] \cdot r^i \) and then \( H(q[1, i+1]) = H(q[1, i]) + q[i+1] \cdot r^i+1 \) for all \( 1 \leq i < m \).
4. We precompute all the values \( H(q[m - i + 1, m]) \) for all \( 1 \leq i \leq m \). That is all the hash values for all the suffixes of \( q \). This can also easily be done incrementally as we have \( H(q[m, m]) = q[m] \cdot r^m \) and then \( H(q[i, m]) = (H(q[i + 1, m]) + q[i]) \cdot r \) for all \( 1 \leq i < m \).

**Hash function computation** We now describe some useful properties of the hash function \( H \) which will be of interest for queries. An interesting property of the hash function \( H \) is that after the precomputation phase, computing \( H(p) \) for any \( p \) at edit distance 1 from \( q \) takes constant time:

1. Deletion at position \( i \): computing the hash value of \( p = q[1, i-1]q[i+1, m] \) (\( q[1, i] \) is defined as the empty string when \( i = 0 \)) is done by the formulae \( H(p) = H(q[1, i-1]) + H(q[i+1, m]) \cdot r^{i-1} \) (\( H(q[1, i-1]) = 0 \) if \( i = 1 \)).

2. Substitution at position \( i \): computing the hash value of \( p = q[1, i-1]cq[i+1, m] \) is done by the formulae \( H(p) = H(q[1, i-1]) + (c + H(q[i+1, m])) \cdot r^i \).

3. Insertion at position \( i \): computing the hash value of \( p = q[1, i-1]cq[i, m] \) is done by the formulae \( H(p) = H(q[1, i-1]) + (c + H(q[i, m])) \cdot r^i \).

Moreover, computing \( H(p') \) for any prefix \( p' \) of a string \( p \) at edit distance 1 from \( q \) also takes constant time:

- The hash value for a prefix \( p' \) of length \( j \) of a string \( p \) obtained by deletion at position \( i \) in \( q \) can be obtained by \( H(p') = H(p) - H(q[j+2, m]) \cdot r^{j+1} \) (\( H(q[j+2, m]) = 0 \) if \( j + 2 > m \)) whenever \( j \geq i \) or \( H(p') = H(q[1, j]) \) otherwise.

- The hash value for a prefix \( p' \) of length \( j \) of a string \( p \) obtained by substitution at position \( i \) in \( q \) can be obtained by \( H(p') = H(p) - H(q[j+1, m]) \cdot r^j \) (\( H(q[j+1, m]) = 0 \) if \( j + 1 > m \)) whenever \( j \geq i \) or \( H(p') = H(q[1, j]) \) otherwise.

- The hash value for a prefix \( p' \) of length \( j \) of a string \( p \) obtained by insertion at position \( i \) in \( q \) can be obtained by \( H(p') = H(p) - H(q[j, m]) \cdot r^{j-1} \) (\( H(q[j, m]) = 0 \) if \( j > m \)) whenever \( j > i \) or \( H(p') = H(q[1, j]) \) otherwise.

**Checking occurrences** Suppose that we have found a potential occurrence of a matching substring of the text obtained by one deletion, one insertion or one substitution. There exists a standard way to check for the validity of the matching, using the arrays \( PA^{-1} \) and \( SA^{-1} \). Suppose that we have located a potential occurrence of a string \( p \) obtainable by deletion of the character at position \( i \) in the query string \( q \). In this case we have \( p = q[1, i-1]q[i+1, m] \). Moreover, Suppose that we have found for \( p \) a potentially matching location \( j \) in the text. Then checking whether this matching location is correct is a matter of just checking that \( PA^{-1}[j + i - 2] \in L[i - 1] \) and that \( SA^{-1}[j + i - 1] \in R[i + 1] \). That is just checking whether \( T[j..j + m - 1] = p = q[1, i-1]q[i+1, m] \) is just a matter of checking that
Lemma 4 Given any pattern $q$ for which the arrays $L$ and $R$ have been pre-computed, we can for any string $p$ at distance 1 from $q$ (where the string $p$ is described with $O(1)$ words of information needed to described the edit operation which transforms $q$ into $p$) and a location $l$ check whether $p$ occurs at location $l$ in the text by probing the text and the arrays $SA^{-1}$ $PA^{-1}$ a constant number of time and thus in time $O(t_{SA})$.

Query algorithm We now describe how queries for a given string $q$ of length $m$ are implemented. Recall that we are dealing with an alphabet of fixed size. That means that the number of strings at distance 1 from a given $q$ is $O(mnσ) = O(m)$. Our algorithm will simply check exhaustively for matching in the text of every string $p$ which can be obtained by one insertion, one deletion or one substitution in the string $q$. Each time we check for a string $p$ we also report the location of all occurrences in which it matches. The matching for a given string $p$ proceeds in the following way:

- Do a weak prefix search on $W_0$ for the string $p$ which takes either constant time or $O(\log b)$ time depending on the implementation used. The result of this weak prefix search is a range $[l_0, r_0]$ of elements in $U$ which are potentially prefixed by $p$.

- Using the prefix-sum data structure $V_0$, compute the range $[l_1, r_1]$ of suffixes of $T$ potentially prefixed by $p$. This range is given by $l_1 = \sum_{1 \leq t < l_0} V_0[t]$ and $r_1 = \sum_{1 \leq t \leq r_0} V_0[t]$ and its computation takes constant time.

- Do a lookup for $j = SA[r_1]$ in the suffix array. This takes either time $O(\log \log n)$ or $O(1)$ depending on the suffix array implementation.

- Finally check that there is a match in the location $j$ in the text (this is done differently depending on whether we are dealing with an insertion, a suppression or a deletion). This checking which is done with the help of lemma 4 needs to do one access to $PA^{-1}$ and one access to $SA^{-1}$ and
thus takes either $O(\log \log n)$ or $O(1)$ times depending on the implementation. If the match is correct we report the position $j$ and additionally report all the remaining matching locations which are at positions $SA[r_1 + 1], SA[r_1 + 2], \ldots, SA[l_1 - 1], SA[l_1]$ (by querying the compressed suffix array using lemma 3). Otherwise we return an empty set.

For proving the correctness of the query, we first prove the following lemma:

**Lemma 5** Given a string $p$ of length at most $b$ such that $p$ is a prefix of at least one suffix of $T$, with the help of $W_0$ and $V_0$, we can find the interval of suffixes prefixed by $p$ in time $O(t_{W_0})$. Further given any $p$ we can check whether it prefixes some suffix of $T$ in time $O(t_{W_0} + t_{SA})$ and if not return an empty set.

**Proof.** We start with the first assertion. If $p$ is prefix of some suffix $s \in T$ then it will also be some prefix of some element in $U$. This is trivially the case if $|s| \geq b$ and this is also the case if $|s| < b$ as we are storing in $U$ the string $b\#^{b-|s|}$ which is necessarily prefixed by $p$ as well. Now we prove that the returned interval $[l_1, r_1]$ is the right interval of suffixes prefixed by $p$. First notice that the weak prefix search by definition returns the interval $[l_0, r_0]$ of elements of $U$ which are prefixed by $p$. Now, we can easily prove that $l_1 = \sum_{1 \leq t < l_0} V_0[t]$ is exactly the number of suffixes which are lexicographically smaller than $p$. This is the case as we know that the sum $\sum_{1 \leq t < l_0} V_0[t]$ includes exactly the following:

1. All suffixes of length less than $b$ which are lexicographically smaller than $p$ and for which an artificial element was inserted in $U$.

2. All the suffixes of length at least $b$ whose prefixes of length $b$ are lexicographically smaller than $p$.

On the other hand we can prove that $r_1 - l_1$ gives exactly the number of suffixes prefixed by $p$. That is $r_1 - l_1 = \sum_{l_0 \leq t < r_0} V_0[t]$ which gives the number of suffixes of length at least $b$ prefixed by elements of $U$ prefixed by $p$ in addition to suffixes of length less than $b$ prefixed by $p$ and for which a corresponding artificial element has been stored in $U$. Now that the first assertion of the lemma has been proved we turn our attention to the second assertion. This second assertion is immediate: give a string $p$ which does not prefix any suffix we know by lemma 4 that the checking will fail for any suffix of $T$ and thus fail for the suffix at position $j$ in the last step which thus returns an empty set.

The following lemma summarizes the query for a prefix $p$ at distance one from $q$:

**Lemma 6** Given any pattern $q$ for which the arrays $L$ and $R$ have been computed, we can for any string $p$ at distance $1$ from $q$ (where the string $p$ is described with $O(1)$ words of information needed to describe the edit operation which transforms $q$ into $p$) search for all the $occ$ suffixes prefixed by $p$ in time $O(t_{W_0} + (occ + 1)t_{SA})$. 

12
Proof. We first prove the correctness of the operations as described above an then prove the time bound. For that we examine the two possibilities:

- $p$ is not prefix to any suffix in $T$ in which case the query should return an empty set. It is easy to prove the equivalent implication: if the data structure returns a non empty set, then there exists at least some suffix of $T$ prefixed by $p$. For the data structure to return a non empty set the checking using lemma 4 must return true for the location $j$ in the text and for this checking to be true $q$ must be prefix of the suffix starting at position $j$ in the text.

- $p$ is prefix of some suffix of $T$ in which case the query must return all those suffixes. Note that by definition the weak prefix search $W_0$ will return the right range of elements of $U$ which are prefixed by $p$. The justification for this is that a single match implies that there exists at least one suffix prefixed by $p$ which implies the weak prefix search $W_0$ returns the correct range of suffixes $SA[r_1], SA[r_1 + 1], SA[r_1 + 2], \ldots, SA[l_1 - 1], SA[l_1]$ which must also have $p$ as a prefix.

We now prove the time bound. In the case that $p$ is not prefix of any suffix in $T$, the query time is clearly $O(t_{W_0} + t_{SA})$ as we are doing one query to weak prefix search $W_0$ (which takes $O(t_{W_0})$ time), one query of the prefix-sum data structure $V_0$ in constant time and finally the checking using lemma 4 which takes $O(t_{SA})$ time. In case $p$ is prefix of some suffixes, then the first step for checking that the set is non empty also takes constant time and reporting each occurrence takes additional $t_{SA}$ time per occurrence.

3.3 Solution for arbitrary pattern length

We use the following lemma proved in [5, section 3.2].

Lemma 7 For any text $T$ of length $n$ characters over an alphabet of constant size we can build an index of size $O(n)$ bits so that we can report all of the occ substrings of the text which are at edit distance 1 from any pattern $q$ of length $m \geq \log^4 n \log \log n$ in time $O(m + \text{occ})$.

The solution for theorem 6 is easily obtained by combining theorem 1 with lemma 7 in the following way: we first build the index of 6 whose query time is upper bounded by $O(m + \text{occ})$ whenever $m \geq \log^4 n \log \log n$ and whose space usage is $O(n)$ bits for some constant $c$. Then we build the data structure of theorem 1 in which we set $b = \log n^4 \log \log n$ and $\varepsilon = \delta/5$ where $\delta$ is any constant which satisfies $0 < \delta < 1$. In the case we have a string of length less than $b$ we use the index of theorem 6 to answer the query in time $O(m + \text{occ})$ when using the first variant or in time $O((m + \text{occ}) \log \log n)$ when using the second variant. In the case where we have a string of length at least $b = \log^4 n \log \log n$, we use the index of 6 answering to queries in time $O(m + \log^4 n \log \log n + \text{occ})$.
occ) = O(m + occ). The space is thus dominated by our index which uses either
O(n(log^4 n log log n)^ε) = O(n log^4 n) or O(n log(log^4 n log log n)) = O(log log n)
bits of space.

4 Data structure for large alphabets

Actually the time bound of theorem 1 have linear dependence on the alphabet
size as the query time is actually O(σm + occ). This query time is not reasonable
in case σ is non constant. In this section we show a solution which has no
dependence on alphabet size. In order to get this solution we combine the
result of [5] with an improved version of [6]. This is shown in section 4.5.
Before that we first show how we improve the solution in [6] but only for query
strings of length bounded by a parameter b. This is proved in sections 4.2, 4.3
and 4.4. This improvement is an extension of the data structures described in
the previous section, but with two major differences: it does not use compressed
variants of the text indexing data structures and its query time is independent
of the alphabet size. Before describing the details of the used data structures,
we first recall in section 4.1 a few definitions and data structures which will be
used in our construction.

4.1 Tools

For proving our results for arbitrary alphabet sizes we will make use of the
following additional tools:

Centroid path decomposition A centroid path decomposition of a tree is a
special decomposition of a tree of n nodes into sets of disjoint paths (sequences
of consecutive parent-child edges) called centroid paths. The main property of
this decomposition is that any root to leaf path in the original tree contains at
most O(log n) centroid paths. A centroid path decomposition relies on the heavy
child notion where the only heavy child of a node n is defined as the child with
the largest subtree size among all the children of n (when more than one child
share the largest subtree size, then choose any one of them). All other children
of a node are defined as light children. A heavy edge is an edge connecting a
parent to its heavy child. A light edge is an edge connecting a node to a light
child. The centroid paths are built in the following way: An initial centroid path
is constituted by the only root-to leaf path which consists in only heavy edges.
Each of the other centroid paths consist in a single initial light edge followed by
a maximal sequence of heavy edges which terminate at a leaf of the tree. In any
centroid path, the labels of the heavy edges are called branching characters. We
say that a centroid path x hangs from a centroid path y at a node p if and only
if there exists some node n_x in x which has a (light) child n_y in y (The light
edge connecting n_x to n_y is the first edge in centroid path y).
Centroid path traversal  For a given pattern of length \( m \) the traversal of a 
suffix tree decomposed according to a centroid path decomposition will traverse 
at most \( t = \min(m, \log n) \) centroid paths. We note those traversed centroid 
paths by \( C_1, C_2, \ldots, C_t \), where each centroid path \( C_i \) hangs from the centroid 
path \( C_{i-1} \).

1D range color reporting data structure  A 1D range color reporting data 
structure solves the following problem: given an array \( A[1..n] \) of colors each 
chosen from the same alphabet of size \( \sigma \), we want to answer to the following 
queries: given an interval \([i, j]\) return all the distinct \( occ \) colors which occur in 
the array elements \( A[i], A[i+1], \ldots, A[j] \). The recent solution devised in [14] uses 
optimal space \( O(n \log \sigma) \) and permits to answer to queries in optimal \( O(occ) \) 
time. Moreover that solution can also report the distinct colors in a given 
interval, one by one in \( O(1) \) time per color.

Lemma 8  [14] Given an array \( A[1..n] \) of colors from the set \( \{1, 2, \ldots, \sigma\} \), we 
can build a data structure of size \( O(n \log \sigma) \) so that given any range \([i..j]\) we 
can return all the \( occ \) distinct colors which appear in \( A[i]...A[j] \) in time \( O(occ) \). 
Moreover the colors can be returned one by one in \( O(1) \) time per color.

4.2 High level description

The solution of theorem 1 has a too strong alphabet dependence in its query 
time. In particular testing all insertion and substitution candidate strings takes 
\( O(m\sigma) \) time as we have \( O(m\sigma) \) candidates and spend \( O(1) \) time for testing 
every candidate. By contrast, deletion candidates are at most \( m \) and thus can 
be tested in \( O(m) \) time. We now give a high level description of our solution. 
Our solution is based on a slightly simplified version of the solution described 
in [6] which itself is a simplification of the solution initially described in [8]. The 
main idea for doing approximate matching for insertions and substitutions is to 
build correction trees (deletion and substitution trees) for each centroid path. 
In total there will be exactly \( O(n) \) trees which store \( O(n \log n) \) elements in total 
(see [8] for the details).

Type-1 substitution tree  A type-1 substitution tree is built in the following 
way. Consider all the centroid paths which hang from a given centroid path \( C \). 
Consider the set of suffixes which are stored in those centroid paths. Each such 
suffix will be modified by substitution on exactly one position in the suffix and 
then stored in the type-1 substitution tree. More in detail, consider the edges 
of a centroid path \( C \) in the top to bottom order with nodes \( n_1, n_2, \ldots, n_t \). Each 
edge \((n_i, n_{i+1})\) for \( 1 < i < t \) is a heavy edge \( \text{5} \) labeled with some character \( c_i \). 
Additionally for each node \( n_i \) for \( 1 < i < t \) there will be one or more centroid 
paths hanging from \( C_i \) at the node \( n_i \). The substitution tree is built in the

\text{5} Note that for the initial centroid path \((n_1, n_2)\) is also a heavy edge
\text{4} the same holds for \( n_1 \) in case \( C \) is the initial centroid path (actually \( n_1 \) in this case is the 
root of the tree)
following way, for each light child $n'$ of a node $n_i$ where the edge $(n_i, n')$ is labeled with the character $c'$ we store all the suffixes stored in the subtree of $n'$ but in which the character $c'$ is replaced with the character $c_i$.

**Type-1 deletion tree** A type-1 deletion tree will be built in a similar way to the type-1 substitution tree. For a given light child $n'$ of a node $n_i$ where $(n_i, n')$ is labeled with the character $c'$, we store only the modified suffixes which are obtained from the suffixes stored in the subtree of $n'$ in which the character $c_i$ immediately follows the character $c'$. This time the modification consists in just removing the character $c'$ from the suffixes instead of replacing the character $c'$ with the branching character $c_i$.

**Type-2 substitution tree** A type-2 substitution tree will be built exactly as the type-1 substitution tree but in which a modified suffix is built by substituting a wildcard character $\alpha$ (this character is a new character added to the alphabet) instead of the branching character $c_i$.

### 4.3 Data structure Implementation

In addition to the data structures used for theorem 1 in which we use non compressed variants of the text indexing data structures and implement a general strategy for searching in the substitution and deletion trees. This search is only useful when used for patterns of length at most $b$. As a consequence before building our data structures we shrink the deletion and substitution tree by retaining only the suffixes modified at one of the $b$ first positions. A correction tree of size $n'$ is implemented through the use of the following data structures:

- We build a weak prefix search data with constant query time on $U'$, the set of all the prefixes of length $b$ of the modified suffixes stored in the correction tree. This data structure uses $O(n' \log n'(b^e + \log \log \sigma))$ bits of space. Similarly to theorem 1 modified suffixes of length $i < b$ are appended with $\#^{b-1}$ and stored in the weak prefix search data structure as artificial elements.

- We store in a prefix-sum data structure the number of modified suffixes prefixed by each element in $U'$ (once again the entry of an artificial element is considered as a one). The entries in the prefix-sum data structure follow the lexicographic order of the prefixes. This prefix sum data structure will use $O(n')$ bits of space.

- We finally store a vector $R$ of $O(n')$ elements in which for each modified suffix in lexicographic order we store the modification on the initial suffix which was used to obtain this modified suffix. The encoding of the modification differs according to type of the tree we are encoding:
  - In case of a type-1 deletion tree, we store a pair which consists in the deleted character $c'$ plus the position of deletion. This takes $\log \sigma$ bits and $\log b$ bits respectively.
In case of a type-1 substitution tree, we store a pair which consists in the substituted character $c'$ plus the position of substitution. This also takes $\log \sigma$ bits and $\log b$ bits respectively.

In case of a type-2 substitution tree, we only store the substituted character $c'$ using only $\log \sigma$ bits. The position of substitution is not necessary in this case as it is known to be at the position of the wildcard character $\alpha$ in the modified suffix.

This vector will thus occupy $O(n'(\log b + \log \sigma))$ bits. Additionally we augment the vector with a 1D range color reporting data structure of lemma 8. This data structure will be able to return all the distinct elements in any range $R[r, l]$ for any $r$ and any $l$ in time $O(1)$ per element. This data structure does not augment the space usage by more than a constant factor.

Summing up the space used by all the components of a correction tree we get a total space usage $O(n'(\log b + \log \sigma))$. Summing up over all the correction trees we get a total $O(n \log n (b' + \log \sigma))$ bits of space usage. An important point to emphasize is that we use the same hash function $H$ for building all the weak prefix search data structures of all the correction trees. In this context, the value $P > n^3\sigma$ used for computing the hash function is large enough to ensure that with high probability all the hash values used in any of the weak prefix search data structures will be distinct (recall that we are using one prefix search per correction tree and we have three correction per centroid path, which implies that we are storing in total $O(n)$ weak prefix search data structures).

### 4.4 Query algorithm

The query algorithm works for exact matches and matches with one deletion, one insertion or one substitution. Exact matching is trivial. Deletion matching works exactly in the same way as in theorem 4. We thus mostly concentrate on the matching for substitutions and insertions. Matching for substitutions is done with the help of the type-1 and type-2 substitutions trees while the matching for insertions is done with the help of the type-1 Deletion tree. More in detail recall that at the matching for a pattern $q_i$, we encounter $t \leq \log n$ centroid paths named $C_1, \ldots, C_t$. Notice that the mismatch with the centroid path $C_i$ for $i < t$ always happens at a branching character $c_i$ in which the corresponding pattern character is $b_i \neq c_i$. For the last step $C_t$ we do not necessarily have a mismatch with $q$ and even if we have one it does not necessarily happen at a branching character. In this case we also note the two mismatching characters in the pattern and in the query by $b_t$ and $c_t$ respectively. The strategy we use is the same as the one used in 8 and 6 except that the implementation differs. More precisely for matching substitutions we do the following for each centroid path $C_i$ for each $i < t$:

- We match all the suffixes obtained by substitution above the character $c_i$ in the centroid path. Note that this substitution can only happen at a
branching character above the character $c_i$. The suffixes which could potentially match above character $c_i$ (more precisely suffixes which could be obtained by insertion at branching characters before character $c_i$) are thus included in the set of suffixes stored in the centroid paths hanging above $c_i$. For matching those suffixes it suffices to query the type-1 substitution tree for the pattern $q$. Details on the implementation are below.

- We match all the suffixes obtained by substitution at the branching character $c_i$. For that, we use the type-2 substitution tree however this time the query is done on the pattern $q$ in which the character $b_i$ is replaced with the character $\alpha$.

For the last centroid path $C_t$ the second step may differ. In fact at the last step the mismatch could well happen at a non branching character or even there could be no mismatch at all. In the latter case we do not do the second step at all. In the former case the second step consists in just querying directly the suffix tree for the pattern $q$ in which the mismatching character $b_t$ is changed to the character $c_t$ found in the centroid path (this is done using lemma 6). Otherwise if the mismatch happens at a branching character, then we just do the second step as it was done for the other centroid paths.

For matching insertions a similar strategy is used:

- We match all the suffixes obtained by insertion above the character $c_i$ in the centroid path. Once again an insertion can only happen at a branching character above the character $c_i$. The suffixes which could potentially match with an insertion above character $c_i$ (more precisely insertions right before the branching characters above character $c_i$) are thus included in the set of suffixes stored in the centroid paths hanging above $c_i$. For matching those suffixes it suffices to query the type-1 deletion for the pattern $q$. Details on the implementation are below.

- We match all the suffixes obtained by insertion in the pattern right before character $c_i$. For that, we also use the type-2 substitution tree however this time the query is done on the pattern $q$ in which the character $\alpha$ is inserted right before the character $b_i$.

As in the case of substitution, the second step of the insertion matching will also differ for last centroid path $C_t$. That is in case the mismatch happens at a non branching character, we will just query the suffix tree for the pattern $q$ in which the character $c_t$ found in the centroid path is inserted right before the mismatching character $b_t$. In case we did not have a mismatch, then the second step is just omitted.

---

\[\text{\footnote{It is easy to see that a substitution at a non branching character do not match any suffix as all the suffixes in the subtrees which hang below a non branching character all contain that non branching character.}}\]

\[\text{\footnote{It is easy to see that any match which can be obtained by an insertion anywhere between two branching characters can also be obtained by insertion right before a branching character.}}\]
Query implementation The queries on the substitution and deletion trees are similar and follow the same strategy. Each query for a string \( q \) on a correction tree follows a preliminary phase which consists in following preliminary steps:

- First query the weak prefix search corresponding to the tree. This returns an interval \([i_0, j_0]\) and takes \(O(1)\) time.

- Query the prefix-sum data structure for the indexes \(i_0\) and \(j_0\) which returns two values \(i_1\) and \(j_1\). From there, we have an interval \([i_1, j_1]\). This also takes \(O(1)\) time.

- Query the 1D range coloring data structure for the interval \([i_0, j_1]\) and only retain the first color. The query time is also \(O(1)\).

- Using this first color, we can check whether the result of the weak-prefix search is right (whether the prefix is a prefix of a modified suffix). To that end, we apply the change to the modified suffix. This change depends on the type of the query and the type of the tree:
  - In case the query is for a type-1 substitution tree, the color is a pair \((\text{char}, \text{pos})\) and we just substitute the character \text{char} at position \text{pos} in \(q\) getting a string \(p\) at the condition that \text{pos} is above \(b_i\). Otherwise we immediately conclude that the query result is invalid and that we do not have any match.
  - In case the query is for a type-1 deletion tree, the color is also a pair \((\text{char}, \text{pos})\) and we just insert the character \text{char} at position \text{pos} in \(q\) getting a string \(p\) at the condition that \text{pos} is above \(b_i\). Otherwise we immediately conclude that the query result is invalid and that we do not have any match.
  - In case the query is for a type-2 substitution tree, the color is just a character \text{char} and we just substitute the character \(\alpha\) (which was at the position of character \(p_i\)) with the character \text{char} getting a string \(p\).

Then we query the suffix tree for the suffixes prefixed by \(p\) with the help of lemma \([4]\). This returns a range of suffixes and we just check for the validity of the matching of the first suffix in that range using lemma \([4]\).

If the matching at the last step of the preliminary phase fails, we deduce the non existence of any modified suffix prefixed by \(q\) and the query terminates. Otherwise, we deduce that there is indeed at least one modified suffix prefixed by \(q\) and will thus return all the matching suffixes which correspond to the modified suffixes prefixed by \(q\). For that, we requery the 1D range coloring data structure again for all the colors in interval \([j_0, j_1]\) but this time we list all the reported colors. Then depending on the type of the tree and the query, we do the following:
For a type-1 substitution or deletion tree we first check whether the reported color \((\text{char}, \text{pos})\) is such that \(q[\text{pos}] \neq \text{char}\). If this is not the case, then we should not report any string as the reported color is wrong meaning that all returned colors are also wrong. Otherwise for each color \((\text{char}, \text{pos})\) we do the following: as in the preliminary phase, we first apply the modification on \(q\) as indicated by the color getting a new string \(p\). That is for a color consisting in a pair \((\text{char}, \text{pos})\) substitute or insert the character \(\text{char}\) at position \(\text{pos}\) in \(q\) getting a string \(q'\), and finally query the suffix tree for the string \(p\) using lemma 6.

For a type-2 substitution tree, for each color \(\text{char}\) where \(\text{char} \neq b_i\) and \(\text{char} \neq c_i\), we do the following: as in the preliminary phase, we first apply the modification on \(q\) as indicated by the color getting a new string \(p\). That is we substitute \(\alpha\) with \(\text{char}\). Then finally query the suffix tree for the prefix \(p\) using lemma 6.

We thus have the following lemma:

**Lemma 9** Querying a correction tree takes \(O(\text{occ} + 1)\) time where \(\text{occ}\) is the number of matching suffixes.

**Proof.** The proof is immediate. Suppose that no matching suffix exists. Then clearly the last step in the preliminary phase can not succeed as the checking can not return true unless there existed a matching suffix. Moreover the time for the preliminary phase is \(O(1)\). Now suppose that we have exactly \(\text{occ}\) matching suffixes, then the preliminary phase will succeed in \(O(1)\) time and the following phase will return the matching suffixes one by one in \(O(1)\) time by matching suffix for a total \(O(\text{occ})\) time.

Finally theorem 4 follows immediately from lemma 9. That is we know that we traverse exactly at most \(\min(m, \log n)\) centroid paths and that for each centroid path \(i\) we need to do at most three queries to correction trees which take in total \(O(1 + \text{occ}_i)\) where \(\text{occ}_i\) is the number of matching suffixes which hang from centroid path \(C_i\). As the sets of matching suffixes which hang from different centroid paths are disjoint, we deduce that the total query time is \(O(m + \text{occ})\).

### 4.5 Solution for arbitrary pattern length

In order to prove theorem 4 we will make use of the following lemma also proved in [5]:

**Lemma 10** [5, section 2.3,Theorem 1] For any text \(T\) of length \(n\) characters over an alphabet of size \(\sigma\) we can build an index of size \(O(n \log n)\) bits so that given any pattern \(p\) of length \(m \geq \log^3 n \log \log n\) we can report all of the \(\text{occ}\) substrings of the text which are at edit distance 1 from \(p\) in time \(O(m + \text{occ})\).

In order to get theorem 4 we combine this lemma with theorem 2. The combination is also straightforward. That is we build both indexes where the index
of theorem 2 is built using the parameter \( b = \log^3 n \log \log n \). Then if given a pattern of short length \( m < \log^3 n \log \log n \), we use the index of theorem 2 to answer in time \( O(m + \text{occ}) \), otherwise given a pattern of length \( m \geq \log^3 n \log \log n \) we use the index of lemma 10 to answer in time \( O(m + \log^3 n \log \log n + \text{occ}) = O(m + \text{occ}) \).

5 Open problems

An obvious open problem is whether the time-space tradeoffs achieved in this paper can be further improved. For constant sized alphabets the space-time tradeoff is not too far from what is achieved for exact matching. However for larger alphabets the space usage seems a bit high and an improvement seems plausible.

References

[1] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.

[2] D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *CPM*, pages 154–167, 2009.

[3] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *ESA* (1), pages 427–438, 2010.

[4] A. L. Buchsbaum, M. T. Goodrich, and J. Westbrook. Range searching over tree cross products. In *ESA*, pages 120–131, 2000.

[5] H.-L. Chan, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. In *CPM*, pages 49–59, 2006.

[6] H.-L. Chan, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. Compressed indexes for approximate string matching. *Algorithmica*, 58(2):263–281, 2010.

[7] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391, 1996.

[8] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don’t cares. In *STOC*, pages 91–100, 2004.

[9] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[10] R. M. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d., 1971.

[11] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[12] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.
[13] G. Jacobson. Space-efficient static trees and graphs. In FOCS, pages 549–554, 1989.

[14] M. Karpinski and Y. Nekrich. Top-k color queries for document retrieval. 2010.

[15] T. W. Lam, W.-K. Sung, and S.-S. Wong. Improved approximate string matching using compressed suffix data structures. In ISAAC, pages 339–348, 2005.

[16] T. W. Lam, W.-K. Sung, and S.-S. Wong. Improved approximate string matching using compressed suffix data structures. Algorithmica, 51(3):298–314, 2008.

[17] M. G. Maass and J. Nowak. Text indexing with errors. In CPM, 2005.

[18] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. SIAM J. Comput., 22(5):935–948, 1993.

[19] S. S. Rao. Time-space trade-offs for compressed suffix arrays. Inf. Process. Lett., 82(6):307–311, 2002.

[20] P. Weiner. Linear pattern matching algorithms. In FOCS, pages 1–11, 1973.