Compressed Indexes for Approximate String Matching

Ho-Leung Chan[1] Tak-Wah Lam[1,*] Wing-Kin Sung[2]
Siu-Lung Tam[1] Swee-Seong Wong[2]

[1] Department of Computer Science, University of Hong Kong.
{hlchan,twlam,sltam}@cs.hku.hk
[2] Department of Computer Science, National University of Singapore.
{ksung,wongss}@comp.nus.edu.sg

Abstract. We revisit the problem of indexing a string $S[1..n]$ to support searching all substrings in S that match a given pattern $P[1..m]$ with at most k errors. Previous solutions either require an index of size exponential in k or need $\Omega(m^k)$ time for searching. Motivated by the indexing of DNA sequences, we investigate space efficient indexes that occupy only $O(n)$ space. For $k = 1$, we give an index to support matching in $O(m + occ + \log n \log\log n)$ time. The previously best solution achieving this time complexity requires an index of size $O(n \log n)$. This new index can be used to improve existing indexes for $k \geq 2$ errors. Among others, it can support matching with $k = 2$ errors in $O(m \log n \log\log n + occ)$ time.

1 Introduction

Given a string $S[1..n]$ over a constant-size alphabet Σ and an integer $k \geq 0$, we want to build an index for S, such that for any subsequent query pattern $P[1..m]$, we can report efficiently all substrings in S that match P with at most k errors. The primary concern is how to achieve efficient pattern matching given limited space for indexing. We consider two kinds of errors: In the Hamming distance case, an error is a character substitution; in the edit distance case, an error can be a character substitution, insertion or deletion.

For exact string matching (i.e., $k = 0$), simple and efficient solutions have been known since the 1970s. Suffix trees [14,20] use $O(n)$ space[3] and achieve the optimal matching time, i.e. $O(m + occ)$, where occ is the number occurrences of P in S. Suffix arrays [13], also using $O(n)$ space but with a smaller constant, give an $O(m+occ+\log n)$ matching time. Recently, two compressed solutions, namely, CSA [8] and FM-index [7], have been proposed; they require only $O(n)$-bit space and support matching in $O(m + occ \log^{\epsilon} n)$ time, for any constant $\epsilon > 0$.

Approximate matching is a more challenging problem even if only one error is allowed. The simplest solution is to search the suffix tree of S for every 1-error

* This research was supported by Hong Kong RGC Grant 7139/04E.

[3] Unless otherwise stated, the space complexity is measured in terms of the number of words, where a word can store $O(\log n)$ bits.

Space	$k = 1$		$k = 2$	
$O(n \log^2 n)$ words	$O(m \log n \log \log n + occ)$	[1]	$O(m + \log^2 n \log \log n + occ)$	[5]
$O(n \log n)$ words	$O(m \log \log n + occ)$	[2]	$O(m + \log^2 n \log \log n + occ)$	†
	$O(m + \log n \log \log n + occ)$	[5]		
$O(n)$ words	$O(\min\{n, m^2\} + occ)$	[4]	$O(\min\{n, m^3\} + occ)$	[4]
	$O(m \log n + occ)$	[10]	$O(m^2 \log n + occ)$	[10]
	$O(m \log \log n + occ)$	[11]	$O(m^2 \log \log n + occ)$	[11]
	$O(m + \log^3 n \log \log n + occ)$	[6]	$O(m + \log^6 n \log \log n + occ)$	[6]
	$O(m + \log n \log \log n + occ)$	†	$O(m \log n \log \log n + occ)$	†

Table 1. A summary of results. Results given in this paper are marked with †.

modification of the query pattern, this requires $O(m^2 + occ)$ time[4] [4]. The first non-trivial improvement was due to Amir et al. [1], who showed that the matching time can be improved to $O(m \log n \log \log n + occ)$ using an index occupying $O(n \log^2 n)$ space. Later Buchshaum et al. [2] further improved the matching time to $O(m \log \log n + occ)$, as well as reducing the index space to $O(n \log n)$. Huynh et al. [10] and Lam et al. [11] further compressed the index to $O(n)$ space, while achieving the time complexity of the indexes reported in [1] and [2], respectively. It has been an open problem whether a time complexity linear in m and occ can be achieved. Recently, Cole et al. [5] resolved in the affirmative with an $O(n \log n)$-space index that supports one-error matching in $O(m + \log n \log \log n + occ)$ time. And more recently, Chan et al. [6] found that Cole et al.'s index admits a time-space tradeoff, i.e., the space can be reduced to $O(n)$ space, yet the time complexity increases to $O(m + \log^3 n \log \log n + occ)$. In this paper, we give new techniques for compressing Cole et al.'s index to $O(n)$ space, while retaining the same time complexity.

To cater for $k = O(1)$ errors, one can perform a brute-force search on a one-error index (i.e., repeatedly modify the pattern at different $k - 1$ positions and search for one-error matches); the matching becomes very slow, involving a factor of m^k in the time complexity. A breakthrough result was given by Cole et al. [5], who devised a recursive solution to build an index that occupies $O(n \log^k n)$ space and supports k-error matching in $O(m + \log^k n \log \log n + occ)$ time for Hamming distance. The term occ is replaced with $occ \cdot 3^k$ for edit distance. Our new 1-error index is essentially a compressed version of the Cole et al.'s 1-error index and can replace it as the base case in their recursive solution. This gives an $O(n \log^{k-1} n)$-space index for k-error matching with the same time complexity.

For indexing long sequences like DNA (which contains a few million to a few billion characters), it is not desirable to have an index whose space complexity grows exponentially as k increases. Like the case of 1-error, the k-error index of Cole et al.'s also admits a time-space tradeoff; in particular, Chan et al. [6] showed that the tree cross product technique by Buchshaum et al. [2] can be

[4] Unless otherwise stated, all matching time mentioned applies to both Hamming and edit distance.

used to trade time for space in the k-error index by Cole et al., the space can be reduced to $O(n)$ while the time for k-error matching increases to $O(m + \log^{k(k+1)} n \log\log n + occ)$. Note that this result is of theoretical interest only as the time complexity is far from practical. For $k = 2$, the time complexity already involves a term $\log^6 n \log\log n$, which is likely to be much bigger than m in most applications. In this paper, we devise a more practical solution for 2-error matching. Specifically, we show that our new $O(n)$-space index for 1-error matching can readily support 2-error matching in $O(m \log n \log\log n + occ)$ time. Furthermore, this index can also handle $k \geq 3$ errors using a brute force manner, and the matching time is $O(m^{k-1} \log n \log\log n + occ)$.

In this paper, we assume the alphabet size $|\Sigma|$ is a constant and hence does not affect the asymptotic analysis. If $|\Sigma|$ is huge or unbounded, we remark that our data structures takes $O(n)$ space (i.e., does not involve $|\Sigma|$); the 1-error matching time is $O(m + |\Sigma| \log n \log\log n + occ)$, the 2-error matching time is $O(|\Sigma|^2 m \log n \log\log n + occ)$, and the k-error matching time, $k > 2$, is $O(|\Sigma|^k m^{k-1} \log n \log\log n + occ)$.

On the technical side, our result is based on a new technique to replace the tree-like data structure of Cole et al. [5] with simple arrays of integers, which are basically some kind of lexicographical information about a suffix tree. We show how approximate string matching can be done by simple range queries over these arrays, instead of the more complicated tree traversals as in [5]. Furthermore, we show how to compress these arrays by storing the lexicographical information imprecisely. This simple approximation can save space and can be verified efficiently. Using the known results on concise representation of increasing sequences and range searching, we reduce the space requirement of Cole et al. by a factor of $O(\log n)$, without increasing the matching time.

We extend our data structure for 1-error matching to support a lazy preprocessing of the input pattern P, which takes $O(m)$ time. Then, for any P' formed by modifying P at one position, we can find the 1-error matches of P' in S in $O(\log n \log\log n + occ')$ time, where occ' is the number of 1-error matches for P'. There are $O(m)$ possible P', so all the 2-error matches of P can be found in $O(m \log n \log\log n + occ)$ time.

Due to the page limit, we omitted the details about our results on k-error matching, which will be given in the full paper. We remark that our paper concerns only worst-case performance. The literature also contains several interesting results on average-case performance, see, e.g., [3, 12, 17].

2 Preliminaries

Let $S[1..n]$ be a string over a constant-size alphabet Σ. The *suffix tree* T of S is a compact trie comprising all suffixes of S. Throughout this paper, we assume that the suffixes are ordered from left to right in increasing lexicographical order. The *suffix array* $SA[1..n]$ is an array of integers such that $SA[i] = j$ if $S[j..n]$ is the lexicographically i-th suffix of S. Note that the *inverse suffix array* $SA^{-1}[1..n]$ satisfies that $SA^{-1}[j]$ gives the lexicographical order of the suffix $S[j..n]$. We

always store T, $SA[1..n]$ and $SA^{-1}[1..n]$, which take $O(n)$ words, or equivalently, $O(n \log n)$ bits.

2.1 Centroid path decomposition

For a suffix tree T, the centroid path decomposition [5] of T is defined as follows. For every internal node u, let v be the child of u with the most number of leaves (ties broken arbitrarily). The edge uv is called a *core edge*. Edges other than core edges are called *side edges*. A *centroid path* $\mathcal{C}$ is a maximal path connecting consecutive core edges. The *root* of $\mathcal{C}$, denoted $r(\mathcal{C})$, is the top-most node on $\mathcal{C}$. The *level* of $\mathcal{C}$ is the number of side edges on the path from the root of T to $r(\mathcal{C})$. We denote $\Delta(T)$ the set of all centroid paths in T.

Denote T_u the subtree of T rooted at a node u and $|T_u|$ be the number of leaves in T_u. Let $T_{\mathcal{C}}$ be $T_{r(\mathcal{C})}$. A leaf (i.e., a suffix) x of T *belongs to* a centroid path $\mathcal{C}$ if x is in $T_{\mathcal{C}}$. A node u *hangs from* $\mathcal{C}$ if its parent edge is a side edge connecting to a node on $\mathcal{C}$, and T_u is called a *side tree* of $\mathcal{C}$. For any node u hanging from $\mathcal{C}$, we note that $|T_u| \leq |T_{\mathcal{C}}|/2$. We highlight some properties of the decomposition.

Fact 1 **(i)** *For any leaf x in T, the path from root of T to x has at most $\log n$ side edges, and x belongs to at most $\log n$ centroid paths.* **(ii)** $\Sigma_{\mathcal{C}\in\Delta(T)}|T_{\mathcal{C}}| \leq n \log n$. **(iii)** *For any two centroid paths $\mathcal{C}_1$ and $\mathcal{C}_2$ of the same level, $T_{\mathcal{C}_1}$ and $T_{\mathcal{C}_2}$ are disjoint, i.e., they do not have common leaves.*

2.2 The side-tree rank of a leaf

Consider a centroid path C. The leaves in $T_{\mathcal{C}}$ are partitioned among the side trees, and our compressed index needs the association between the leaves and the side trees. To save space, we rank the side trees hanging from $\mathcal{C}$ in descending order of their size (i.e., the number of leaves), and we store, for each leaf x, the rank of the side tree containing x, which is denoted *st-rank*$_{\mathcal{C}}(x)$. To store such side-tree ranks for all centroid paths, we need $\sum_{\mathcal{C}\in\Delta(T)} \sum_{x\in T_{\mathcal{C}}} \lceil \log \textit{st-rank}_{\mathcal{C}}(x) \rceil$ bits, which is naively at most $n \log^2 n$ bits (because $\Sigma_{\mathcal{C}\in\Delta(T)}|T_{\mathcal{C}}| \leq n \log n$ and *st-rank*$_{\mathcal{C}}(x) \leq n$). Our compressed index actually takes advantage of a better upper bound.

Lemma 1. **(i)** *Let x be a leaf in T, belonging to centroid paths $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_\alpha$. Then, $\sum_{1\leq i\leq\alpha} \log \textit{st-rank}_{\mathcal{C}_i}(x) \leq \log n$.* **(ii)** $\sum_{\mathcal{C}\in\Delta(T)} \sum_{x\in T_{\mathcal{C}}} \lceil \log \textit{st-rank}_{\mathcal{C}}(x) \rceil \leq 2n \log n$.

Proof. Let x be any leaf in T. **(i)** We assume that the $\alpha \geq 1$ centroid paths to which x belongs are labeled in such a way that $r(\mathcal{C}_{i+1})$ hangs from $\mathcal{C}_i$, for $i = 1, \ldots, \alpha - 1$. Let $r_i = \textit{st-rank}_{\mathcal{C}_i}(x)$. We note that $|T_{\mathcal{C}_i}| \geq |T_{\mathcal{C}_{i+1}}| \times r_i$, because the r_i-th largest side tree has at most $\frac{1}{r_i}$ of all leaves belonging to $\mathcal{C}_i$. Thus, we have $\sum_{i=1}^{\alpha} \log r_i \leq \sum_{i=1}^{\alpha-1} \log(\frac{|T_{\mathcal{C}_i}|}{|T_{\mathcal{C}_{i+1}}|}) + \log r_\alpha \leq \log \frac{|T_{\mathcal{C}_1}|}{|T_{\mathcal{C}_\alpha}|} + \log |T_{\mathcal{C}_\alpha}| = \log |T_{\mathcal{C}_1}| \leq \log n$. **(ii)** It follows directly from (i). □

2.3 The y-fast trie

Let $A[1..\ell]$ be a sorted array of integers. Given any integer j, the predecessor query reports the smallest i such that $j < A[i]$. Willard [21] gave the y-fast trie data structure with the following performance.

Lemma 2 ([21]). *Let $A[1..\ell]$ be a sort array of integers in $[1, n]$, we can build a y-fast trie for A using $O(\ell \log n)$ bits to support the predecessor query in $O(\log \log n)$ time.*

2.4 The LCP data structure

Let T be the suffix tree for $S[1..n]$. Let T' be a trie for only a subset of suffixes in T. A *location* in T' is a node in T' or a point on an edge some characters below a node. Given a pattern $P[1..m]$, an integer $i \leq m$ and a location u in T', the query $\text{LCP}(P, i, u)$ asks for the location at which the suffix $P[i..m]$ diverges from T', when $P[i..m]$ is aligned to T' starting from u. We are allowed to preprocess P in $O(m)$ time, and the concern is to efficiently answer subsequent LCP queries for different suffix $P[i..m]$ and location u.

Let ℓ be the number of leaves in T'. Cole et al. [5] proposed an $O(\ell \log^2 n)$-bit LCP data structure to answer each subsequent LCP query in $O(\log \log n)$ time. In this paper, we use a simple observation to reduce the space requirement to $O(\ell \log n)$ bits.

We first outline the LCP data structure in [5]. Essentially, [5] performs a centroid path decomposition on T'. For any $\mathcal{C} \in \Delta(T')$, let $T'_{\mathcal{C}}$ be the subtree of T' rooted at $r(\mathcal{C})$ and let $\ell_{\mathcal{C}}$ be the number of leaves in $T'_{\mathcal{C}}$. The path from $r(\mathcal{C})$ to a leaf in T' is a suffix of S. In [5], an array $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is used to store the lexicographical order of each such suffix, among all suffixes of S. Note that $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is strictly increasing. A y-fast trie [21] of size $O(\ell_{\mathcal{C}} \log n)$ bits is built to answer in $O(\log \log n)$ time the predecessor query. These A arrays and y-fast tries over all centroid paths in T' take totally $O(\ell \log^2 n)$-bit space. The remaining part of the LCP data structure takes only $O(\ell \log n)$-bit space. We use the following observation to reduce the space requirement.

Lemma 3. *For any centroid path $\mathcal{C}$ in T', let $h_{\mathcal{C}}$ be the total length of edge label from root of T' to $r(\mathcal{C})$.* **(i)** *For $i = 1, \ldots, \ell_{\mathcal{C}}$, $A_{\mathcal{C}}[i]$ can be computed in $O(1)$ time using $h_{\mathcal{C}}$ and the inverse suffix array of S.* **(ii)** *The predecessor query can be supported in $O(\log \log n)$ time using an $O(\ell_{\mathcal{C}})$-bit data structure.*

Proof. **(i)** Consider the i-th leaf in $T'_{\mathcal{C}}$ and let $S[j..n]$ be its leaf label, i.e., $S[j..n]$ is the suffix corresponding to the path from root of T' to this leaf. By definition, $A_{\mathcal{C}}[i]$ is the lexicographical order of $S[j + h_{\mathcal{C}}..n]$, so $A_{\mathcal{C}}[i] = SA^{-1}[j + h_{\mathcal{C}}]$.

(ii) Instead of building a y-fast trie on the complete $A_{\mathcal{C}}$ array, we only build a y-fast trie for $A_{\mathcal{C}}[\log n], A_{\mathcal{C}}[2 \log n], \ldots$ using $O(\ell_{\mathcal{C}})$ bits space. The predecessor query can be done by first querying y-fast trie, then performing a binary search in $A_{\mathcal{C}}$ within an interval of length $\log n$. It takes $O(\log \log n)$ time. □

Thus, for each centroid path $\mathcal{C}$, we store an integer $h_{\mathcal{C}}$ and an $O(\ell_{\mathcal{C}})$-bit predecessor data structure. It takes totally $O(\ell \log n)$ bits over all centroid paths in T'. Together with the remaining part of the LCP data structure of [5], we have the following lemma.

Lemma 4. *Let T' be a compact trie comprising ℓ suffixes of $S[1..n]$. We can build an $O(\ell \log n)$-bit LCP data structure for T'. Given any pattern $P[1..m]$, we can preprocess P in $O(m)$ time. Each subsequent LCP query can be answered in $O(\log \log n)$ time.*

3 An $O(n \log n)$-bit index for 1-error matching

This section explains how to compress the data structure of Cole et al. [5] in order to obtain an $O(n \log n)$-bit index for $S[1..n]$, such that for any pattern $P[1..m]$, it finds the 1-error matches of P in $O(m + occ + \log n \log \log n)$ time. We consider Hamming distance first. Extension to edit distance is given at the end of the section.

Let us first consider the suffix tree T of S. We perform a centroid path decomposition on T and let $\Delta(T)$ be the set of all centroid paths. For any centroid path $\mathcal{C}$, we define a set of *modified suffixes* as follows. Let s be a suffix in T passing through the root of $\mathcal{C}$, and diverging from $\mathcal{C}$ at a node u on $\mathcal{C}$. We create a modified suffix s' by modifying s at the first character after u, replacing it with the first character on the core edge out of u. We say that s *generates* s' *according to* $\mathcal{C}$. We assume the suffix corresponding to $\mathcal{C}$ itself is also a modified suffix generated according to $\mathcal{C}$.

To find the 1-error matches of P in S, the core of our algorithm is solving the following prefix matching problem.

Definition 1 (Prefix matching query for modified suffixes). Let T be the suffix tree of S and P be a pattern. For any centroid path $\mathcal{C}$ of T, let $\phi_{\mathcal{C}}$ be the set of modified suffixes generated according to $\mathcal{C}$ with P as a prefix. The *prefix matching query*, denoted *prefix*$(\mathcal{C})$, reports the set $\Phi_{\mathcal{C}}$ of suffixes in T that generate the modified suffixes in $\phi_{\mathcal{C}}$.

Lemma 5. *Let T be the suffix tree of $S[1..n]$. We can build an $O(n \log n)$-bit index for T. For any pattern $P[1..m]$, we can preprocess P in $O(m)$ time; then prefix$(\mathcal{C})$, for any centroid path $\mathcal{C}$ in T, can be answered in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, where $e_{\mathcal{C}}$ is non-negative and the sum of $e_{\mathcal{C}}$ over all centroid paths in T is at most $2 \log n$.*

The following subsections are devoted to the proof of Lemma 5. Then by using the framework of Cole et al. [5], we can exploit our indexes for the prefix matching queries and LCP queries to obtain an $O(n \log n)$-bit index for the 1-error matching problem, and Theorem 1 follows. We leave the details to the full paper.

Theorem 1. *We can build an $O(n \log n)$-bit index for $S[1..n]$ that finds the 1-error matches of any $P[1..m]$ in $O(m + occ + \log n \log \log n)$ time, where occ is the number of matches found.*

3.1 The prefix matching query for modified suffixes

Cole et al. [5] used the error-tree data structure to support the prefix matching query in $O(m)$ preprocessing time and $O(\log\log n + |\Phi_{\mathcal{C}}|)$ query time. Their solution takes $O(n\log^2 n)$-bits and requires sophisticated tree operations. In this paper, we uses interesting techniques to replace their tree-like data structure with simple arrays of integers.

A simple $O(n\log^2 n)$-bit solution. Let T be the suffix tree of S. Let U be the set of all the $O(n\log n)$ modified suffixes generated according to all the centroid paths. For each centroid path $\mathcal{C}$, we simply store two arrays of integers. Let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated according to $\mathcal{C}$, in increasing lexicographical order. We store (1) *lex-order*$_{\mathcal{C}}$, where *lex-order*$_{\mathcal{C}}[i]$ is the lexicographical order of s'_i among all modified suffixes in U. (2) *label*$_{\mathcal{C}}$, where *label*$_{\mathcal{C}}[i]$ $= j$ if s'_i is generated by the suffix $S[j..n]$.

In addition, we store a compact trie M for U. Given any pattern P, we preprocess P by aligning P with M starting from the root. It determines the range $[d, e]$ such that all modified suffixes with lexicographical order in $[d, e]$ (w.r.t. U) have P as a prefix. Given any centroid path $\mathcal{C}$ of T, the prefix matching query is done by a range search query on *lex-order*$_{\mathcal{C}}$. For each i such that $d \leq$ *lex-order*$_{\mathcal{C}}[i] \leq e$, we report *label*$_{\mathcal{C}}[i]$. The range search on *lex-order*$_{\mathcal{C}}$ takes $O(\log\log n)$ time by storing a y-fast trie [21] on *lex-order*$_{\mathcal{C}}$. Thus, finding the $\Phi_{\mathcal{C}}$ takes $O(\log\log n + |\Phi_{\mathcal{C}}|)$ time. The total space requirement is $O(n\log^2 n)$ bits.

An $O(n\log n)$-bit solution. We exploit sophisticated techniques to reduce the space requirement of the above solution to $O(n\log n)$ bits.

1. *Sampling.* Instead of M, we store a compact trie containing only one in every $\log n$ leaves of M. With this approximation, answering the prefix matching query on a centroid path $\mathcal{C}$ requires extra verification on the suffixes before reporting it as $\Phi_{\mathcal{C}}$. Let $e_{\mathcal{C}}$ be the number of suffixes that require verification. We show that the sum of $e_{\mathcal{C}}$ is at most $2\log n$ over all centroid paths.
2. *Constant time verification.* Given the pattern P, a centroid path $\mathcal{C}$ and a suffix $s = S[j..n]$, we need to verify whether s generates a modified suffix s' according to $\mathcal{C}$ with P as a prefix. We show how to perform the verification in $O(1)$ time using the suffix tree, suffix array and the LCP data structure.
3. *Concise representation.* The *lex-order*$_{\mathcal{C}}$ and *label*$_{\mathcal{C}}$ arrays take totally $O(n\log^2 n)$ bits if stored directly. We replace their entries with integers of smaller values, by exploiting the properties of the centroid path decomposition. Then, we use variable size encoding to represent the arrays in $O(n\log n)$ bits.

Precisely, our $O(n\log n)$-bit solution stores a compact trie N comprising $O(n)$ modified suffixes in U, namely, the lexicographically $(\log n)$-th, $(2\log n)$-th, $(3\log n)$-th, ... modified suffixes. For a centroid path $\mathcal{C}$, let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated for $\mathcal{C}$. We store two length-ℓ arrays for $\mathcal{C}$.

– *lex-order*$_{\mathcal{C}}[1..\ell]$: *lex-order*$_{\mathcal{C}}[i]$ is the lexicographical order of s'_i among all the $O(n)$ modified suffixes in N.

– $label_{\mathcal{C}}[1..\ell]$: Define $label_{\mathcal{C}}[i] = j$ if s'_i is generated by $S[j..n]$.

A naive way to store the *rank* and *label* arrays still takes $O(n \log^2 n)$ bits. In Section 3.3, we give non-trivial techniques to compress them into $O(n \log n)$ bits. We first proceed to show how to use these two arrays to find $\Phi_{\mathcal{C}}$ efficiently.

3.2 Answering a prefix matching query

Given a pattern P, we show how to preprocess P in $O(m)$ time such that for any centroid path $\mathcal{C}$, the prefix matching query *prefix*($\mathcal{C}$) can be answered in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, and the sum of $e_{\mathcal{C}}$ over all centroid paths $\mathcal{C}$ is at most $2 \log n$.

Error-bounded candidate generation. We align P with N starting from the root in $O(m)$ time to find the range $[d, e]$ corresponding to leaves in N with P as a prefix. Then, for any centroid path $\mathcal{C}$, we can find $\Phi_{\mathcal{C}}$ as follows.

1. Find the maximal range $[p..q]$ such that $d-1 \leq \textit{lex-order}_{\mathcal{C}}[p] \leq \textit{lex-order}_{\mathcal{C}}[q] \leq e+1$ by a *range_search* query on the $\textit{lex-order}_{\mathcal{C}}$ array.
2. For each i in $[p..q]$, let $j = label_{\mathcal{C}}[i]$. If $\textit{lex-order}_{\mathcal{C}}[i]$ is not $d-1$ or $e+1$, report $S[j..n]$ in $\Phi_{\mathcal{C}}$; otherwise, call $S[j..n]$ a *candidate* and verify whether $S[j..n]$ is in $\Phi_{\mathcal{C}}$.

We want the $\textit{lex-order}_{\mathcal{C}}$ array to support the operation *range_search*(d, e): given integers d and e, $d \leq e$, return p, q such that $\textit{lex-order}_{\mathcal{C}}[p..q]$ is the largest interval satisfying $d-1 \leq \textit{lex-order}_{\mathcal{C}}[p] \leq \textit{lex-order}_{\mathcal{C}}[q] \leq e+1$. We can build a y-fast trie [21] on one per $\log n$ entries in $\textit{lex-order}_{\mathcal{C}}$. Then *range_search* can be done in $O(\log \log n)$ time by a query to the y-fast trie and then a binary search in an interval of length $\log n$. It uses $O(n \log n)$ bits over all centroid paths.

Lemma 6. *For any centroid path $\mathcal{C}$, let $e_{\mathcal{C}}$ be the number of candidates generated for verification. The sum of $e_{\mathcal{C}}$ over all centroid paths is at most $2 \log n$.*

Proof. For any integer i, at most $\log n$ entries over all *lex-order* arrays equal i, and we verify a suffix only if its *lex-order* value is $d-1$ or $e+1$. □

Constant time verification. We need to verify whether a candidate is in $\Phi_{\mathcal{C}}$.

Lemma 7. *We can preprocess P in $O(m)$ time. Then, for any centroid path $\mathcal{C}$ and candidate $S[j..n]$, we can verify in $O(1)$ time whether $S[j..n]$ is in $\Phi_{\mathcal{C}}$, i.e., $S[j..n]$ generates a modified suffix according to $\mathcal{C}$ with P as a prefix.*

Proof. We preprocess P with the suffix tree T, which takes $O(m)$ time: For each suffix $P[r..m]$, we store the range $[d_r, e_r]$ such that all leaves with lexicographical order (w.r.t. T) in $[d_r, e_r]$ have $P[r..m]$ as a prefix.

To verify a suffix $S[j..n]$, let v be the node in T that $S[j..n]$ diverges from the path of P. v can be found in constant time using an $O(n \log n)$-bit LCA data structure [9] for T. Let $P[1..r]$ (or equivalently, $S[j..j+r-1]$) be the path label from the root to v. (For each node v in T, we store the path length from the

root to v so that $P[1..r]$ is known in $O(1)$ time.) $S[j..n]$ is in $\Phi_{\mathcal{C}}$ if (1) v is on $\mathcal{C}$, (2) the first character on the core edge out of v is $P[r+1]$, and (3) $S[j+r+1..n]$ has a prefix matching $P[r+2..m]$. The last condition can be checked in constant time by comparing $SA^{-1}[j+r+1]$ with the range $[d_{r+2}, e_{r+2}]$ obtained during the preprocessing. □

In conclusion, Lemmas 6 and 7 show that we can build $O(n \log n)$-bit data structures on top of *lex-order*$_{\mathcal{C}}$ and *label*$_{\mathcal{C}}$. Then, we can preprocess P in $O(m)$ time; for any centroid path $\mathcal{C}$, we can answer the prefix matching query in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, where $e_{\mathcal{C}}$ is the number of verification performed and the sum of $e_{\mathcal{C}}$ over all centroid path is at most $2 \log n$.

3.3 Compressed representation of the lexicographical information

We show how to store the *lex-order* and *label* arrays in $O(n \log n)$-bit space.

Compressing the *lex-order* arrays. For any centroid path $\mathcal{C}$, entries in *lex-order*$_{\mathcal{C}}$ are monotonic increasing, so efficient compression is possible by Lemma 8. Proof of Lemma 8 will be given in the full paper.

Lemma 8. *Let $c_1 \le c_2 \le \ldots \le c_\ell$ be a sequence of positive integers. We can store the sequence in $O(\log c_1 + \ell \cdot \max\{\log(\frac{c_\ell - c_1}{\ell}), 1\})$ bits and support $O(1)$ retrieval time for each c_i.*

Lemma 9. *We can store the lex-order arrays of all centroid paths in $O(n \log n)$-bit space and support $O(1)$ retrieval time to each entry.*

Proof. For any centroid path $\mathcal{C}$, let $\ell_{\mathcal{C}}$ be the number of modified suffixes generated according to $\mathcal{C}$. Let $h_{\mathcal{C}} =$ *lex-order*$_{\mathcal{C}}[\ell_{\mathcal{C}}] -$ *lex-order*$_{\mathcal{C}}[1]$. Consider all $\mathcal{C} \in \Delta(T)$. By Lemma 8, the total space required for the *lex-order*$_{\mathcal{C}}$ arrays is $\sum O(\log$ *lex-order*$_{\mathcal{C}}[1] + \ell_{\mathcal{C}} \cdot \max\{\log(\frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}}), 1\}) \le \sum O(\log$ *lex-order*$_{\mathcal{C}}[1] + \ell_{\mathcal{C}} \cdot \log(2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})) = O(n \log n) + O(\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}})$ bits. By the AM-GM inequality on $\sum \ell_{\mathcal{C}}$ numbers, we have $\prod \left(2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}}\right)^{\ell_{\mathcal{C}}} \le \left(\frac{1}{\sum \ell_{\mathcal{C}}} \sum \ell_{\mathcal{C}} (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})\right)^{\sum \ell_{\mathcal{C}}} = \left(\frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}\right)^{\sum \ell_{\mathcal{C}}}$. Note that $x^{1/x} \le 2$ for $x \ge 2$. Let $x = \frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}$, we have $\left(\frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}\right)^{\sum \ell_{\mathcal{C}}} = x^{\frac{1}{x}(2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}})} \le 2^{(2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}})}$. Thus, $\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}} \le 2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}$. Let L_j be the set of all centroid paths with level j, $j \le \log n$. For any two centroid paths in L_j, their *lex-order* arrays are disjoint, so $\sum_{\mathcal{C} \in L_j} h_{\mathcal{C}} \le n$. There are at most $\log n$ levels, so $\sum_{\mathcal{C} \in \Delta(T)} h_{\mathcal{C}} \le n \log n$. □

Compressing the *label* arrays. Unlike *lex-order*, the *label* array is not an increasing sequence. To compress *label*, we simulate it by other "simpler" arrays. For a centroid path $\mathcal{C}$, let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated according to $\mathcal{C}$, in increasing lexicographical order. Assume that t side trees hang from $\mathcal{C}$. We store the following information.

- *st-rank*$_{\mathcal{C}}[1..\ell]$: Suppose s'_i is generated by the suffix s in T. Then *st-rank*$_{\mathcal{C}}[i]$ stores the side-tree-rank of s, i.e., the rank of the side tree containing s.

– *tree_pointer*$_{\mathcal{C}}[1..t]$: *tree_pointer*$_{\mathcal{C}}[j]$ points to the j-th largest side tree of $\mathcal{C}$ in T, ties are broken arbitrarily.
– *modified_rank*$_{\mathcal{C},v}[1..|T_v|]$, for each side-tree T_v of $\mathcal{C}$ in T: *modified_rank*$_{\mathcal{C},v}[j] = i$ if the j-th suffix in T_v generates s'_i.

To find *label*$_{\mathcal{C}}[i]$, note that *tree_pointer*$_{\mathcal{C}}$[*st-rank*$_{\mathcal{C}}[i]$] returns the side tree T_v hanging from $\mathcal{C}$ which contains the suffix that generates s'_i. We perform a *rank*(i) query on *unmodifed_rank*$_{\mathcal{C},v}$, where *rank*(i) returns j if *modified_rank*$_{\mathcal{C},v}[j] = i$. Thus, *label*$_{\mathcal{C}}[i]$ is the j-th suffix in T_v. Let w_v be the number suffixes on the left of v in T. Then *label*$_{\mathcal{C}}[i] = SA[w_v + j]$.

By Lemma 1, the *st-rank*$_{\mathcal{C}}$ arrays for all centroid paths $\mathcal{C}$ can be represented in $O(n \log n)$ bits using variable size encoding. We can build a *select* data structure [15] on the arrays, which uses $O(n \log n)$ bit, to support $O(1)$ time access to each entry. The *tree_pointer* arrays contain only n pointers in total and take $O(n \log n)$ bits over all centroid paths.

Lemma 10. *We can store modified_rank*$_{\mathcal{C},v}$ *array to support the rank*(i) *query in* $O(1)$ *time: given any integer* i*, return* j *if modified_rank*$_{\mathcal{C},v}[j] = i$*; return null otherwise. Total space required over all* $\mathcal{C} \in \Delta(T)$ *and all side trees* T_v *hanging from* $\mathcal{C}$ *is* $O(n \log n)$ *bits.*

Proof. Consider any centroid path $\mathcal{C}$ and a side tree T_v hanging from $\mathcal{C}$. The sequence *modified_rank*$_{\mathcal{C},v}$ is strictly increasing and ranges from 1 to $|T_{\mathcal{C}}|$, hence it can be stored in $O(|T_v| \log \frac{|T_{\mathcal{C}}|}{|T_v|})$ bits while supporting the *rank* query [18]. Let $f(T_{\mathcal{C}})$ denote the total space required to store the *modified_rank* arrays for all centroid paths with root in $T_{\mathcal{C}}$, including $\mathcal{C}$. Let $T_{v_1}, T_{v_2}, \ldots, T_{v_t}$ be side trees hanging from $\mathcal{C}$. Note that $f(T_{\mathcal{C}}) \leq \sum_{i=1}^{t} \left(O(|T_{v_i}| \log \frac{|T_{\mathcal{C}}|}{|T_{v_i}|}) + f(T_{v_i})\right)$. Resolving this recurrence, we have $f(T_{\mathcal{C}}) = O(|T_{\mathcal{C}}| \log |T_{\mathcal{C}}|)$ for any $\mathcal{C}$. Therefore, all *modified_rank* arrays in T can be stored in $O(n \log n)$ bits. □

In conclusion, Lemma 9 and 10 show that the *lex-order* and *label* arrays can be represented in $O(n \log n)$ bits and support $O(1)$ time retrieval. Together with the matching algorithm of Section 3.2, Lemma 5 follows.

Extending to edit distance. We handle each type of edit operations separately. Substitution is handled by the above data structure. To find substrings of S that matches P with one insertion (to the substrings), we generate another type of modified suffixes, which we called the *insertion suffixes*. Precisely, let $\mathcal{C}$ be a centroid path in the suffix tree T. Let s be a suffix in T passing through the root of $\mathcal{C}$, and diverging from $\mathcal{C}$ at a node u on $\mathcal{C}$. We create an insertion suffix s' by inserting a character c to s after u, where c is the first character on the core edge out of u. We say that s generates an insertion suffix s' according to $\mathcal{C}$. Given a pattern P, finding the 1-error matches can be reduced to a number of prefix matching queries on the insertion suffixes and LCP queries. By handing the prefix matching queries using the same techniques as shown, we find all 1-error matches for insertion in the $O(m + occ' + \log n \log \log n)$ time, where occ' is the number of matches found. Handling deletion is identical. The total space for the data structures is $O(n \log n)$ bits.

4 An $O(n \log n)$-bit index for 2-error

Given a pattern $P[1..m]$, we can find its 2-error matches in S as follows: First modify P at each position $P[i]$, substituting it with a character $c \neq P[i]$. Denote the modified pattern as $P_{i,c}[1..m]$. Then, find all 1-error matches of $P_{i,c}$ with the error in $P_{i,c}[1..i-1]$. By trying all $i = 1, \ldots, m$ and each possible $c \in \Sigma$, each 2-error match of P is found exactly once.

To support the above operations, we store the $O(n \log n)$-bit index for 1-error matching, as well as some $O(n \log n)$-bit auxiliary data structures (to be defined). Then, to find all 2-error matches of a pattern $P[1..m]$, we perform the followings for every $i = 1, \ldots, m$ and for every $c \in \Sigma$.

1. Search $P_{i,c}$ in T to identify the centroid paths and side edges $P_{i,c}$ overlaps.
2. Search $P_{i,c}$ in the sampled 1-error tree N to identify an interval $[d, e]$ corresponding to modified suffixes in N with $P_{i,c}$ as a prefix.
3. Find the 1-error matches of $P_{i,c}$ where the error is in $P[1..i-1]$ and is on a side edge. This is done by performing an LCP query in T for each side edge $P_{i,c}$ overlaps.
4. Find the 1-error matches of $P_{i,c}$ where the error is in $P[1..i-1]$ and is on a centroid path. We follow the approach in Section 3.2, generating candidates and verifying them for correct matches.

By preprocessing P (but not $P_{i,c}$) in $O(m)$ time, we can perform Step 1 in $O(\log \log n + w)$ time, where $w \leq 2 \log n$ is the number of centroid paths and side edges $P_{i,c}$ overlaps. We can also build an $O(n \log n)$-bit LCP data structure for N so that Step 2 takes $O(\log \log n)$ time. Step 3 can be done in $O(\log n \log \log n + \#output)$ time.

The main challenge is Step 4. Generating candidates involves *range_search* queries on the $rank_{\mathcal{C}}$ arrays, and the candidates generated may include an unbounded number of modified suffixes having $P_{i,c}$ as a prefix but their modified position is not in $P_{i,c}[1..i-1]$. Thus, for each centroid path $\mathcal{C}$, we store a $modified_{\mathcal{C}}$ array storing the location of the modified character of each modified suffix generated according to $\mathcal{C}$. We then use a Bounded Value Range Query (BVRQ) data structure [16] to ensure generating candidates with the required error positions, plus at most $2 \log n$ counterfeits. Finally, we verify each candidate in $O(1)$ time. We can show that the $modified_{\mathcal{C}}$ arrays and the BVRQ data structures take totally $O(n \log n)$ bits and Step 4 takes $O(\log n \log \log n + \#output)$ time. So, we have the following lemma.

Lemma 11. *We can build an $O(n \log n)$-bit index for $S[1..n]$. Given a pattern $P[1..m]$, we can preprocess P in $O(m)$ time. For any modified pattern $P_{i,c}$, we can find all 1-error matches of $P_{i,c}$ with the error in $P_{i,c}[1..i-1]$ in $O(\log n \log \log n + occ_{i,c})$ time, where $occ_{i,c}$ is the number of matches found.*

By repeating the search for each $P_{i,c}$, $i \leq m$ and $c \in \Sigma$, we can find all 2-error matches of P in $O(m \log n \log \log n + occ)$ time, where occ is the number of 2-error matches.

References

1. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. In *Proceedings of Workshop on Algorithms and Data Structures*, 1999, pages 181–192.
2. A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *European Symposium on Algorithms*, 2000, pages 120–131.
3. E. Chavez and G. Navarro. A metric index for approximate string matching. In *Proceedings of Latin American Theoretical Informatics*, 2002, pages 181-195.
4. A. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 1995, pages 41–54.
5. R. Cole, L. A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of Symposium on Theory of Computing*, 2004, pages 91–100.
6. H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. A Linear-Size Index for Approximate Pattern Matching. To appear in *CPM*, 2006.
7. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Symposium on Foundations of Computer Science*, 2000, pages 390–398.
8. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, 2000, pages 397–406.
9. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-355, 1984.
10. T. N. D. Huynh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 2004, pages 434–444.
11. T. W. Lam, W. K. Sung, S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proceedings of International Symposium on Algorithms and Computation*, 2005.
12. M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-10503, Fakultät für Informatik, TU München, Mar. 2005.
13. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
14. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
15. J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Computer Science*, 1996, pages 37–42.
16. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of 13th Symposium on Discrete Algorithms*, 2002, pages 657–666.
17. G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *J. Discrete Algorithms*, 1(1):205-209, 2000.
18. R. Raman, V. Raman, and S. S. Rao Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002, pages 233–242.
19. K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002, pages 225–232.
20. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, 1973, pages 1–11.
21. D. E. Willard. Log-Logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.