Algorithmica (2010) 58: 263–281
DOI 10.1007/s00453-008-9263-2

Compressed Indexes for Approximate String Matching

Ho-Leung Chan · Tak-Wah Lam · Wing-Kin Sung · Siu-Lung Tam · Swee-Seong Wong

Received: 6 September 2007 / Accepted: 2 December 2008 / Published online: 17 December 2008

Abstract We revisit the problem of indexing a string $S[1..n]$ to support finding 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, 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 $O(n \log n)$ space. This new index can also be used to improve existing indexes for $k \geq 2$ errors. Among others, it can support 2-error matching in $O(m \log n \log\log n + occ)$ time, and k-error matching, for any $k > 2$, in $O(m^{k-1} \log n \log\log n + occ)$ time.

Keywords Compressed index · Approximate string matching

Result in this paper has appeared in a preliminary form in the Proceedings of the 14th Annual European Symposium (ESA 2006).
Current address of H.-L. Chan: Department of Computer Science, University of Pittsburgh, USA.
The research T.-W. Lam was supported by the Hong Kong RGC Grant HKU 7140/06E.

H.-L. Chan · T.-W. Lam · S.-L. Tam
Department of Computer Science, University of Hong Kong, Hong Kong, Hong Kong

H.-L. Chan
e-mail: hlchan@cs.hku.hk

T.-W. Lam
e-mail: twlam@cs.hku.hk

S.-L. Tam
e-mail: sltam@cs.hku.hk

W.-K. Sung (✉) · S.-S. Wong
Department of Computer Science, National University of Singapore, Singapore, Singapore
e-mail: ksung@comp.nus.edu.sg

S.-S. Wong
e-mail: wongss@comp.nus.edu.sg

1 Introduction

Given a string $S[1..n]$ over a finite 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 locations 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 in the 1970s. Suffix trees [16, 23] use $O(n)$ space[1] and achieve the optimal matching time, i.e. $O(m + occ)$, where occ is the number occurrences of P in S. Suffix arrays [15], 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 [10] and FM-index [9], have been proposed, they require only $O(n)$-bit space and they can support matching in $O(m + occ \log^{\epsilon} n)$ time, for any constant $\epsilon > 0$.

Approximate matching is a challenging problem even if only one error is allowed. The simplest solution is to search the suffix tree of S for every 1-error modification of the query pattern, this requires $O(m^2 + occ)$ time[2] [7]. 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. [4] 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. [12] and Lam et al. [13] further compressed the index to $O(n)$ space, while achieving the time complexity reported in [1] and [4], respectively. It has been an open problem whether a time complexity linear in m and occ can be achieved. Recently, Cole et al. [8] 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. [5] 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 to compress 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 has been given by Cole et al. [8], who devised a recursive solution to build an index that occupies $O(n \log^k n)$ space and takes $O(m + \log^k n \log\log n + occ)$ time to perform a k-error matching. 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.

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

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

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

Space	$k=1$		$k=2$	
$O(n\log^2 n)$ words	$O(m\log n\log\log n+occ)$	[1]	$O(m+\log^2 n\log\log n+occ)$	[8]
$O(n\log n)$ words	$O(m\log\log n+occ)$	[4]	$O(m+\log^2 n\log\log n+occ)$	†
	$O(m+\log n\log\log n+occ)$	[8]		
$O(n)$ words	$O(\min\{n,m^2\}+occ)$	[7]	$O(\min\{n,m^3\}+occ)$	[7]
	$O(m\log n+occ)$	[12]	$O(m^2\log n+occ)$	[12]
	$O(m\log\log n+occ)$	[13]	$O(m^2\log\log n+occ)$	[13]
	$O(m+\log^3 n\log\log n+occ)$	[5]	$O(m+\log^6 n\log\log n+occ)$	[5]
	$O(m+\log n\log\log n+occ)$	†	$O(m\log n\log\log n+occ)$	†

For indexing long sequences like DNA (which often contains millions to billions of 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. [5] showed that the tree cross product technique by Buchshaum et al. [4] can be used to trade time for space in the k-error index by Cole et al., and the space can be reduced to $O(n)$ while the time for k-error matching increases to $O(m+\log^{k(k+1)}\log\log n+occ)$. 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 $O(|\Sigma|^{k-1}m^{k-1}\log n\log\log n+occ)$ for Hamming distance and $O((2|\Sigma|)^{k-1}m^{k-1}\log n\log\log n+occ)$ for edit distance. We believe that this time complexity is more practical.

On the technical side, our result is based on a new technique to replace the tree-like data structure of Cole et al. [8] 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 [8]. 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 a given pattern $P[1..m]$, which takes $O(m)$ time. Then, for any P' formed by modifying P at one of the m positions, 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'. And all the 2-error matches of P can be found in $O(m\log n\log\log n+occ)$ time.

Table 1 summarizes the existing results and the new results in this 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., [6, 14, 20].

2 Preliminaries

We first review several basic data structures including suffix tree, centroid path decomposition and y-fast trie, as well as some related notations. Then, we present two new observations on side-tree rank and LCP data structure, which are essential to the $O(n)$-space index.

2.1 Suffix Tree, Centroid Path Decomposition and y-Fast Trie

Let $S[1..n]$ be a string over a finite alphabet Σ. The *suffix tree* $\mathcal{T}$ of S is a compact trie comprising all suffixes of S. Each edge is labeled with a substring 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 $\mathcal{T}$, $SA[1..n]$ and $SA^{-1}[1..n]$, which take $O(n)$ words, or equivalently, $O(n \log n)$ bits.

With respect to a suffix tree $\mathcal{T}$, the centroid path decomposition [8] of $\mathcal{T}$ 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 $\mathcal{C}$. We denote $\Delta(\mathcal{T})$ the set of all centroid paths in $\mathcal{T}$. It is useful to define the level of a centroid path. Intuitively, all centroid paths attached to the same higher centroid path via a side edges are considered to be of the same level; specifically, we define the *level* of $\mathcal{C}$ is the number of side edges on the path from the root of $\mathcal{T}$ to $r(\mathcal{C})$.

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

Fact 1 (i) *For any leaf x in $\mathcal{T}$, the path from root of $\mathcal{T}$ to x has at most* $\log n$ *side edges, and x belongs to at most* $\log n$ *centroid paths.*

(ii) $\sum_{\mathcal{C} \in \Delta(\mathcal{T})} |\mathcal{T}_{\mathcal{C}}| \leq n \log n$.

(iii) *For any two centroid paths $\mathcal{C}_1$ and $\mathcal{C}_2$ of the same level, $\mathcal{T}_{\mathcal{C}_1}$ and $\mathcal{T}_{\mathcal{C}_2}$ are disjoint, i.e., they do not have common leaves.*

Let $A[1..\ell]$ be a array of integers in increasing order. Given any integer j, the predecessor query reports the smallest i such that $A[i] > j$. Assuming the range of the integers is fixed, Willard [24] gave an efficient data structure called the y-fast trie to support the predecessor query.

Lemma 1 ([24]) *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.*

2.2 The Side-Tree Rank of a Leaf

Let $\mathcal{T}$ be the suffix tree for $S[1..n]$. Consider a centroid path $\mathcal{C}$ of $\mathcal{T}$. The leaves in $\mathcal{T}_{\mathcal{C}}$ are partitioned among the side trees of $\mathcal{C}$. We want to store 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 $\textit{st-rank}_{\mathcal{C}}(x)$. To store such side-tree ranks for all centroid paths, we need $\sum_{\mathcal{C}\in\Delta(\mathcal{T})}\sum_{x\in\mathcal{T}_{\mathcal{C}}}\lceil\log \textit{st-rank}_{\mathcal{C}}(x)\rceil$ bits, which is naively at most $n\log^2 n$ bits (because $\sum_{\mathcal{C}\in\Delta(\mathcal{T})}|\mathcal{T}_{\mathcal{C}}| \le n\log n$ and $\textit{st-rank}_{\mathcal{C}}(x) \le n$). Our compressed index takes advantage of a better upper bound as follows.

Lemma 2 (i) *Let x be a leaf in $\mathcal{T}$. Suppose that x belongs to $\alpha \ge 1$ centroid paths $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\alpha$. Then $\sum_{1\le i\le\alpha}\log \textit{st-rank}_{\mathcal{C}_i}(x) \le \log n$.*

(ii) $\sum_{\mathcal{C}\in\Delta(\mathcal{T})}\sum_{x\in\mathcal{T}_{\mathcal{C}}}\lceil\log \textit{st-rank}_{\mathcal{C}}(x)\rceil \le 2n\log n$.

Proof Let x be any leaf in $\mathcal{T}$.

(i) We assume that the $\alpha \ge 1$ 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 $|\mathcal{T}_{\mathcal{C}_i}| \ge |\mathcal{T}_{\mathcal{C}_{i+1}}| \times r_i$, 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 $\sum_{i=1}^{\alpha}\log r_i \le \sum_{i=1}^{\alpha-1}\log(\frac{|\mathcal{T}_{\mathcal{C}_i}|}{|\mathcal{T}_{\mathcal{C}_{i+1}}|}) + \log r_\alpha \le \log\frac{|\mathcal{T}_{\mathcal{C}_1}|}{|\mathcal{T}_{\mathcal{C}_\alpha}|} + \log|\mathcal{T}_{\mathcal{C}_\alpha}| = \log|\mathcal{T}_{\mathcal{C}_1}| \le \log n$.

(ii) It follows directly from (i). □

2.3 The LCP Data Structure

The LCP query can be defined with respect to the suffix tree $\mathcal{T}$ of a text $S[1..n]$, or, in general, any compact trie $\mathcal{T}'$ of a subset of suffixes of S. Below a *location* in $\mathcal{T}'$ refers to a node in $\mathcal{T}'$ or an edge; the latter is further characterized by a position in the edge label, which represents a proper prefix of the edge label.

Definition 1 (LCP query) Given a pattern $P[1..m]$, an integer $i \le m$ and a location u in $\mathcal{T}'$, we attempt to match the suffix $P[i..m]$ character by character with the edge labels of $\mathcal{T}'$ starting from u. Note that this process may terminate without matching all characters of $P[i..m]$. The query $\mathrm{LCP}(P,i,u)$ asks for the location in $\mathcal{T}'$ at which the matching ends.

We are allowed to preprocess P in $O(m)$ time, and the concern is to efficiently answer subsequent LCP queries for different suffixes $P[i..m]$ and different locations u.

Let ℓ be the number of leaves in $\mathcal{T}'$. Cole et al. [8] proposed an $O(\ell\log^2 n)$-bit LCP data structure to answer an LCP query in $O(\log\log n)$ time. In this paper, we give a simple observation to reduce the space requirement to $O(\ell\log n)$ bits.

First of all, we review the LCP data structure in [8]. A centroid path decomposition is performed on $\mathcal{T}'$. Consider a particular centroid path $\mathcal{C}$ in $\mathcal{T}'$ and all the leaves in the subtree $\mathcal{T}'_{\mathcal{C}}$ (which is rooted at $r(\mathcal{C})$). Every leaf represents a suffix of S. We want to store the rank of each leaf (suffix) in $\mathcal{T}'_{\mathcal{C}}$ with respect to all suffixes of S. Let $\ell_{\mathcal{C}}$ be the number of leaves in $\mathcal{T}'_{\mathcal{C}}$. In [8], an array $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is used to store the rank of the leaves (suffixes) in $\mathcal{T}'_{\mathcal{C}}$. Precisely, $A_{\mathcal{C}}[i]$ stores the rank of π_i among all suffixes of S, where π_i denotes the concatenation of edge labels on the path from $r(\mathcal{C})$ to the i-th leaf in $\mathcal{T}'_{\mathcal{C}}$. Note that $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is strictly increasing. A y-fast trie [24] of size $O(\ell_{\mathcal{C}} \log n)$ bits is built to answer in $O(\log \log n)$ time the predecessor query. The A arrays and y-fast tries over all centroid paths in $\mathcal{T}'$ occupy $O(\ell \log^2 n)$-bit space. The remaining part of Cole et al.'s 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 $\mathcal{T}'$, let $h_{\mathcal{C}}$ be the total length of edge label from $r(\mathcal{T}')$ 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 $\mathcal{T}'_{\mathcal{C}}$ and let $S[j..n]$ be its leaf label, i.e., $S[j..n]$ is the suffix corresponding to the path from root of $\mathcal{T}'$ to this leaf. Then, $A_{\mathcal{C}}[i]$ is the lexicographical order of $S[j + h_{\mathcal{C}}..n]$, which is $SA^{-1}[j + h_{\mathcal{C}}]$.

(ii) Instead of building a y-fast trie on the complete $A_{\mathcal{C}}$ array, we only build a y-fast trie for $A_{\mathcal{C}}[\log n]$, $A_{\mathcal{C}}[2 \log n]$, ... 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 $\mathcal{T}'$. Together with the remaining part of the LCP data structure of [8], we have the following lemma.

Lemma 4 *Let $\mathcal{T}'$ be a compact trie comprising ℓ suffixes of $S[1..n]$. We can build an $O(\ell \log n)$-bit LCP data structure for $\mathcal{T}'$. 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. [8] into an $O(n \log n)$-bit index for $S[1..n]$, such that for any pattern $P[1..m]$, its 1-error matches in S can be found in $O(m + occ + \log n \log \log n)$ time. We consider the Hamming distance first. Extension to the edit distance and $k > 1$ errors are given at the end of the section.

First of all, it is useful to observe a simple fact about where a 1-error match of a pattern P appears in the suffix tree $\mathcal{T}$ of S.

Consider any prefix $P[1..i]$ of P that has an exact match in S. Suppose that $P[1..i]$ also has a 1-error match in S, in which $P[j]$, where $1 \leq j \leq i$, is the mismatch character. Then, with respect to $\mathcal{T}$, the path from $r(\mathcal{T})$ labeled with $P[1..j-1]$ always ends at a node u, and $P[j]$ is the first character labeling an outgoing edge of u.

With respect to the suffix tree $\mathcal{T}$ of S, we perform a centroid path decomposition on $\mathcal{T}$. For each centroid path $\mathcal{C}$, we define a set of *modified suffixes* of $\mathcal{C}$ as follows. Let s be a suffix of S. Note that s corresponds to a path from $r(\mathcal{T})$ to a leaf ℓ in $\mathcal{T}$. Suppose that this path passes through $\mathcal{C}$, from $r(\mathcal{C})$ down to a node u on $\mathcal{C}$. We create a modified suffix s' by modifying s at the first character after u (if exists), replacing it with the first character on the core edge out of u. We say that s *generates* s' with respect to $\mathcal{C}$. Let $\phi_{\mathcal{C}}$ be the set of all modified suffixes with respect to $\mathcal{C}$. To ease our discussion, we assume that $\phi_{\mathcal{C}}$ includes the suffix ending at the leaf of $\mathcal{C}$. See Fig. 1 for an example.

The core of our solution to the 1-error matching problem is an index that answers the following query about modified suffixes.

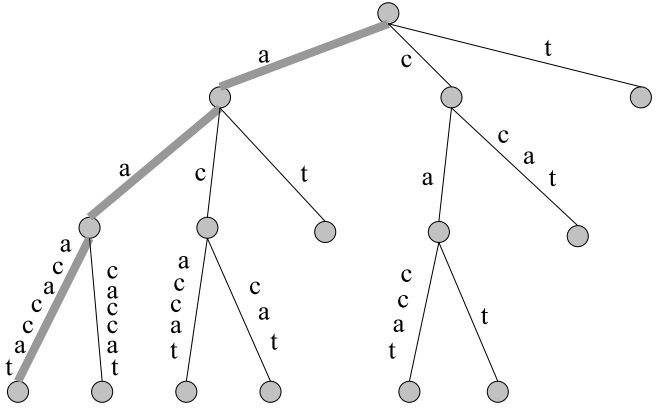
Definition 2 (Prefix matching of modified suffixes (PMMS) query) For any pattern P and any centroid path $\mathcal{C}$ of $\mathcal{T}$, let $\phi_{\mathcal{C}}(P)$ be the subset of modified suffixes in $\phi_{\mathcal{C}}$ that contain P as a prefix. The query $\text{PMMS}_{\mathcal{C}}(P)$ asks for the set of suffixes in $\mathcal{T}$ that generate the modified suffixes in $\phi_{\mathcal{C}}(P)$.

Lemma 5 *Let $\mathcal{T}$ be the suffix tree of $S[1..n]$. We can build an $O(n \log n)$-bit index for $\mathcal{T}$. For any pattern $P[1..m]$, we can preprocess P in $O(m)$ time; then $\text{PMMS}_{\mathcal{C}}(P)$, for any centroid path $\mathcal{C}$ in $\mathcal{T}$, can be answered in $O(\log\log n + |\text{PMMS}_{\mathcal{C}}(P)| + e_{\mathcal{C}})$ time, where $e_{\mathcal{C}} \geq 0$ and the sum of $e_{\mathcal{C}}$ over all centroid paths in $\mathcal{T}$ is at most $2 \log n$.*

The rest of this section (Sects. 3.1 to 3.3) is devoted to proving Lemma 5. Before proving Lemma 5, we show that 1-error matching problem can be solved by using three $O(n \log n)$-bit indexes, namely, a suffix tree, an index for LCP queries and an index for PMMS queries.

Theorem 6 *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.*

Proof Given any pattern P, we match P with $\mathcal{T}$ starting from the root. We first assume that P can be matched entirely. (The other case that P cannot be matched entirely is similar and will be discussed later.) Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_h$ be the centroid paths visited during the matching. We order the centroid paths so that $r(\mathcal{C}_i)$ hangs from $\mathcal{C}_{i-1}$ for $i = 2, \ldots, h$. Let v_i be the last node visited on $\mathcal{C}_i$, and let $P[x_i..y_i]$ be the portion of P matched with $\mathcal{C}_i$, i.e., $P[x_i..y_i]$ is matched with the edge labels from $r(\mathcal{C}_i)$ to v_i. Note that for each $\mathcal{C}_i$, it is possible that the root $r(\mathcal{C}_i)$ is the only node



Modified suffixes	Generated by
a	S[9..9]
aa	S[8..9]
aaaaccat	S[2..9]
aaacaccat	S[1..9]
aaaccat	S[3..9]
aacat	S[5..9]
aaccat	S[4..9]
aat	S[7..9]
acat	S[6..9]

Fig. 1 The suffix tree for $S[1..9] =$ "aaacaccat" is shown on the *left*. The thick edges correspond to a centroid path $\mathcal{C}$. The table on the right shows the modified suffixes s' generated according to $\mathcal{C}$, and the respective suffixes s that generate s'. As an example, if the query pattern P is "aac", then $\text{PMMS}_{\mathcal{C}}(P)$ should report $S[5..9]$ and $S[4..9]$

visited, i.e., $v_i = r(\mathcal{C}_i)$, and $P[x_i..y_i]$ is a null string in this case. For simplicity, we let $x_{h+1} = m + 1$. We find all 1-error matches of P in S in two steps.

1. We first find all 1-error matches with the mismatch in $P[x_i..y_i]$ for some $i = 1, 2, \ldots, h$. To do it, for each $\mathcal{C}_i$, we use an PMMS query (precisely, $\text{PMMS}_{C_i}(P)$) to find the set of suffixes in $\mathcal{T}$ that generates the modified suffixes in $\phi_{\mathcal{C}_i}(P)$. Each of these suffixes corresponds to a one-error match of P with the mismatch occurring in a certain character in $P[x_i..y_i]$.
2. We then find all 1-error matches with the mismatch in $P[y_i + 1..x_{i+1} - 1]$ for some $i = 1, 2, \ldots, h$. Note that each such 1-error match X must have the mismatch at the position $P[y_i + 1]$ (otherwise, X can be matched entirely with $P[y_i + 1..x_{i+1} - 1]$, which is a contradiction). For each v_i, $P[1..y_i]$ is exactly the label of the path from the root of $\mathcal{T}$ to v_i. If $y_i < m$, then for every outgoing edge e of v_i with the first character $\neq P[y_i + 1]$, we consider the location w on e that is one character below v_i. We want to know how much $P[y_i + 2..m]$ can match $\mathcal{T}$ starting from w; in particular, we use an LCP query ($\text{LCP}(\mathcal{T}, y_i + 2, w)$) to determine the location w' where the matching ends. If $P[y_i + 2..m]$ can be matched completely, all the suffixes in $\mathcal{T}$ under the location w' are one-error matches of P with the mismatch at $P[y_i + 1]$; otherwise no one-mismatch is to be reported.

The 1-error matches reported in Step 1 and Step 2 are disjoint, and they together include all possible 1-error matches of P in S. Note that if $y_h = m$, then $\text{PMMS}_{\mathcal{C}_h}(P)$ also returns all the 0-error matches of P.

Notice that $h \leq \log n$. Step 1 issues at most $\log n$ PMMS queries, and Step 2 invokes at most $O(\log n)$ LCP queries (recall that the alphabet is assumed to be of constant size). By Lemmas 4 and 5, the total time required to find all the 1-error matches of P is $O(m + occ + \log n \log\log n)$.

If P cannot be matched entirely, let $P[1..m']$ be the longest prefix of P matched with $\mathcal{T}$ and it ends at a location v in $\mathcal{T}$. We first perform the above two steps as before to find all 1-error matches with the mismatch in $P[1..m']$. Then, if v locates on the edge, we find all 1-error matches with the mismatch at $P[m' + 1]$ by using $O(1)$ LCP queries on the location that is one character below v. We don't need to perform any thing if v is a node, because v would be equal to v_h in this case and the case 2 above finds the required 1-error matches as well. □

3.1 The Prefix Matching Query of Modified Suffixes

Cole et al. [8] used the error-tree data structure to support the PMMS query in $O(m)$ preprocessing time and $O(\log\log n + |\text{PMMS}_{\mathcal{C}}(P)|)$ query time. Their solution takes $O(n \log^2 n)$-bits and requires sophisticated tree operations. In this paper, we use interesting techniques to replace their tree-like data structure with arrays of integers.

A Simple $O(n \log^2 n)$-bit Solution Let $\mathcal{T}$ 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) $\textit{lex-order}_{\mathcal{C}}$, where $\textit{lex-order}_{\mathcal{C}}[i]$ 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 $\mathcal{T}$, the PMMS query can be done by a range search query on $lex\text{-}order_{\mathcal{C}}$. For each i such that $d \leq lex\text{-}order_{\mathcal{C}}[i] \leq e$, we report $label_{\mathcal{C}}[i]$. The range search on $lex\text{-}order_{\mathcal{C}}$ can be done in $O(\log\log n)$ time storing a y-fast trie [24] on $lex\text{-}order_{\mathcal{C}}$. Thus, finding $\text{PMMS}_{\mathcal{C}}(P)$ takes $O(\log\log n + |\text{PMMS}_{\mathcal{C}}(P)|)$ time. The total space required 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 PMMS query requires extra verification. Let $e_{\mathcal{C}}$ be the number of suffixes that require verification. We will 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 $lex\text{-}order_{\mathcal{C}}$ 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 $\mathcal{C}$.

- $lex\text{-}order_{\mathcal{C}}[1..\ell]$: $lex\text{-}order_{\mathcal{C}}[i]$ 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 *lex-order* and *label* arrays still takes $O(n\log^2 n)$ bits. In Sect. 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 $\text{PMMS}_{\mathcal{C}}(P)$ efficiently.

3.2 Answering a PMMS Query

Given a pattern P, we show how to preprocess P in $O(m)$ time such that for any centroid path $\mathcal{C}$, $\text{PMMS}_{\mathcal{C}}(P)$ can be answered in $O(\log\log n + |\text{PMMS}_{\mathcal{C}}(P)| + e_{\mathcal{C}})$ 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 $[d, e]$ that corresponds to all the leaves in N with P as a prefix. Then, for any centroid path $\mathcal{C}$, we can find $\text{PMMS}_{\mathcal{C}}(P)$ as follows.

1. Find the maximal range $[p..q]$ such that $d-1 \le \textit{lex-order}_{\mathcal{C}}[p] \le \textit{lex-order}_{\mathcal{C}}[q] \le e+1$ by a *range_search* query on the $\textit{lex-order}_{\mathcal{C}}$ array.
2. For each i in $[p..q]$, let $j = \textit{label}_{\mathcal{C}}[i]$. 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 $\textit{range\_search}(d, e)$: given integers d and e where $d \le e$, return p, q such that $\textit{lex-order}_{\mathcal{C}}[p..q]$ is the largest interval satisfying $d-1 \le \textit{lex-order}_{\mathcal{C}}[p] \le \textit{lex-order}_{\mathcal{C}}[q] \le e+1$. We can build a y-fast trie [24] on one per $\log n$ entries in $\textit{lex-order}_{\mathcal{C}}$. Then a *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 $O(n \log n)$-bit space over all centroid paths.

Lemma 7 *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. We verify a suffix only if its *lex-order* value is $d-1$ or $e+1$, so the lemma follows. □

Constant Time Verification We can verify whether a candidate is in $\text{PMMS}_{\mathcal{C}}(P)$ in $O(1)$ time.

Lemma 8 *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 $\textit{PMMS}_{\mathcal{C}}(P)$, 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 $\mathcal{T}$, which takes $O(m)$ time: For each suffix $P[r..m]$, we compute and store the range $[d_r, e_r]$ such that all leaves with lexicographical order (w.r.t. $\mathcal{T}$) 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 $\mathcal{T}$ that $S[j..n]$ diverges from the path of P. It takes only constant time to find v using an $O(n \log n)$-bit LCA data structure [11] for $\mathcal{T}$. Let $P[1..r]$ $(= S[j..j+r-1])$ be the path label from the root to v. (For each node v in $\mathcal{T}$, we store the path length from the root to v, then r can be determined in $O(1)$ time.) $S[j..n]$ is in $\text{PMMS}_{\mathcal{C}}(P)$ if

- v is on $\mathcal{C}$;
- the first character on the core edge out of v is $P[r+1]$; and
- $S[j+r+1..n]$ 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 $[d_{r+2}, e_{r+2}]$, the latter has been obtained during the preprocessing. □

In conclusion, Lemmas 7 and 8 show that we can build $O(n \log n)$-bit index on top of $\textit{lex-order}_{\mathcal{C}}$ and $\textit{label}_{\mathcal{C}}$. Then, we can preprocess any given pattern P in $O(m)$ time, for any centroid path $\mathcal{C}$, we can answer the query $\text{PMMS}_{\mathcal{C}}(P)$ in $O(\log\log n + |\text{PMMS}_{\mathcal{C}}(P)| + e_{\mathcal{C}})$ 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 now explain 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.

Lemma 9 *Let $c_1 \le c_2 \le \cdots \le c_\ell$ 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.*

Proof We store c_1 explicitly in $O(\log c_1)$ bits, and we store $d_i = c_i - c_1$, for $i = 1, \ldots, \ell$ as follows. Note that d_i is monotonic increasing and $d_\ell = c_\ell - c_1$.

If $d_\ell \le \ell$, we store a bit sequence with ℓ `1`'s, and there are $d_i - d_{i-1}$ `0`'s between the i-th and the $(i-1)$-th `1`, for $i = 2, \ldots, \ell$. We build a select data structure [17] on the bit sequence to support the *select*(i) query, i.e., reporting the location of the i-th `1`, in $O(1)$ time. Finding d_i is done by *select*$(i) - i$, in $O(1)$ time. The bit sequence and the select data structure takes $O(\ell)$ bits.

If $d_\ell > \ell$, we can use the data structure of [22] for storing sorted integers, which stores d_i's in $O(\ell \log \frac{d_\ell}{\ell})$ bits and supports $O(1)$ retrieval time. □

Lemma 10 *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 $\mathcal{C} \in \Delta(\mathcal{T})$. By Lemma 9, the total space (bits) 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*$_{\mathcal{C}}[1] + \ell_{\mathcal{C}} \cdot \log(2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})) = O(n \log n) + O(\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}})$. Since $1 + x \le e^x$ for any x, $\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}}) \le \log \prod (e^{(1 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})\ell_{\mathcal{C}}}) = O(\sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}})$. 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 $\mathcal{T}$. 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 $\mathcal{T}$, ties are broken arbitrarily.
- *modified_rank*$_{\mathcal{C},v}[1..|\mathcal{T}_v|]$, for each side-tree $\mathcal{T}_v$ of $\mathcal{C}$ in $\mathcal{T}$: *modified_rank*$_{\mathcal{C},v}[j] = i$ if the j-th suffix in $\mathcal{T}_v$ generates s'_i.

To find $label_{\mathcal{C}}[i]$, we note that $tree\_pointer_{\mathcal{C}}[st\text{-}rank_{\mathcal{C}}[i]]$ returns the side tree $\mathcal{T}_v$ hanging from $\mathcal{C}$ that contains the suffix generating s'_i. We perform a $rank(i)$ query on $modified\_rank_{\mathcal{C},v}$, 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 of suffixes on the left of v in $\mathcal{T}$. Then $label_{\mathcal{C}}[i] = SA[w_v + j]$.

By Lemma 2, the $st\text{-}rank_{\mathcal{C}}$ 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 [17] 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 11 *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 $\mathcal{C} \in \Delta(\mathcal{T})$ and all side trees $\mathcal{T}_v$ hanging from $\mathcal{C}$ is $O(n \log n)$ bits.*

Proof Consider any centroid path $\mathcal{C}$ and a side tree $\mathcal{T}_v$ hanging from $\mathcal{C}$. The sequence $modified\_rank_{\mathcal{C},v}$ is strictly increasing and ranges from 1 to $|\mathcal{T}_{\mathcal{C}}|$, hence it can be stored in $O(|\mathcal{T}_v| \log \frac{|\mathcal{T}_{\mathcal{C}}|}{|\mathcal{T}_v|})$ bits while supporting the *rank* query [21]. Let $f(\mathcal{T}_{\mathcal{C}})$ denote the total space required to store the *modified_rank* arrays for all centroid paths with root in $\mathcal{T}_{\mathcal{C}}$, including $\mathcal{C}$. Let $\mathcal{T}_{v_1}, \mathcal{T}_{v_2}, \ldots, \mathcal{T}_{v_t}$ be side trees hanging from $\mathcal{C}$. Note that $f(\mathcal{T}_{\mathcal{C}}) \leq \sum_{i=1}^{t} (O(|\mathcal{T}_{v_i}| \log \frac{|\mathcal{T}_{\mathcal{C}}|}{|\mathcal{T}_{v_i}|}) + f(\mathcal{T}_{v_i}))$. Resolving this recurrence, we have $f(\mathcal{T}_{\mathcal{C}}) = O(|\mathcal{T}_{\mathcal{C}}| \log |\mathcal{T}_{\mathcal{C}}|)$ for any $\mathcal{C}$. Therefore, all *modified_rank* arrays in $\mathcal{T}$ can be stored in $O(n \log n)$ bits. □

In conclusion, Lemma 10 and 11 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 Sect. 3.2, Lemma 5 follows.

3.4 Remarks

Extension 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 $\mathcal{T}$. Let s be a suffix in $\mathcal{T}$ 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 PMMS queries on the insertion suffixes and LCP queries. By handling the PMMS 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. Deletion can be handled in an identical way. The total space for the data structures is $O(n \log n)$ bits.

Extension to k Errors To support 2-error matching, Cole et al. [8] build compact tries on the modified suffixes, calling them the 1-error trees. Then, 2-*modified suffixes* are generated by performing centroid path decompositions on the 1-error trees, and generating modified suffixes according to the new centroid paths. In general, k-error matching requires the ℓ-error trees for all $\ell = 1, \ldots, k-1$ and the k-modified suffixes. The space requirement increases by a $\log n$ factor when k increases by one. That is, the $(k-1)$-error trees occupy $O(n \log^k n)$ bits and the k-modified suffixes requires $O(n \log^{k+1} n)$-bits. Using our technique, we can support k-error matching by storing the ℓ-error trees for all $\ell < k$, and replacing the k-modified suffixes with the *lex-order* and *label* arrays (which are based on the $(k-1)$-error trees instead of the suffix tree). The replacement saves the space requirement by a factor of $\log n$, and the total space of all data structures is $O(n \log^k n)$ bits. The matching time remains the same as in [8].

4 An $O(n \log n)$-bit Index for 2-Error and k-Error Matching

This section gives an $O(n \log n)$-bit index for $S[1..n]$ such that for any pattern $P[1..m]$, it finds all 2-error matches of P in $O(m \log n \log \log n + occ)$ time. We first consider the Hamming distance. Extensions to the edit distance and $k > 2$ errors are explained afterwards.

To build the 2-error matching index for a text $S[1..n]$, we need the $O(n \log n)$-bit index for 1-error matching plus some $O(n \log n)$-bit auxiliary data structures (to be defined). Then, given a pattern $P[1..m]$, we find the 2-error matches in S as follows: We first modify P at each position $P[i]$, substituting it with a character $c \neq P[i]$. Denote the modified pattern as $P_{i,c}[1..m]$. Next, we 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.

We find the required 1-error matches for $P_{i,c}$ by the following steps.

1. Search $P_{i,c}$ in $\mathcal{T}$ to identify the centroid paths and side edges that $P_{i,c}$ 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 $P_{i,c}[1..i-1]$ and is on a side edge. This is done by performing an LCP query in $\mathcal{T}$ for each side edge $P_{i,c}[1..i-1]$ overlaps.
4. Find the 1-error matches of $P_{i,c}$ where the error is in $P_{i,c}[1..i-1]$ and is on a centroid path. We follow the approach in Sect. 3.2. I.e., for each centroid path that $P_{i,c}[1..i-1]$ overlaps, generate the candidates and verify them for correct matches.

We will show in Sect. 4.1 that by using an enhanced LCP data structure, we can preprocess P (but not $P_{i,c}$) in $O(m)$ time, afterwards it takes $O(\log \log n + w)$ time to execute Step 1 for each $P_{i,c}$, where w is the number of centroid paths and side edges $P_{i,c}$ overlaps. Similarly, we will show that Step 2 takes $O(\log \log n)$ time, and Step 3 takes $O(\log n \log \log n + occ')$ time, where occ' is the number of matches found. For Step 4, we will show in Sects. 4.2 and 4.3 that generating candidates and

verifying them takes $O(\log n \log\log n + occ'')$ time, where occ'' is the number of matches reported. So we have the following lemma.

Lemma 12 *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 have the following theorem.

Theorem 13 *We can build an $O(n \log n)$-bit index for $S[1..n]$ that finds the 2-error matches of any $P[1..m]$ in $O(m \log n \log\log n + occ)$-time, where occ is the number of matches found.*

4.1 Enhanced LCP Data Structure

We will see that Steps 1, 2 and 3 are closely related to the following LCP problem: Consider a trie $\mathcal{T}'$ for a subset of ℓ suffixes in $\mathcal{T}$. Recall that the LCP query $\mathrm{LCP}(P, x, u)$ asks for the location at which the suffix $P[x..m]$ diverges from $\mathcal{T}'$, when $P[x..m]$ is aligned to $\mathcal{T}'$ starting from a location u on $\mathcal{T}'$. We extend the query to $P_{i,c}$ such that $\mathrm{LCP}(P_{i,c}, x, u)$ asks for the location at which $P_{i,c}[x..m]$ diverges from $\mathcal{T}'$ when aligning from u. The challenge is that we can only process P in $O(m)$ time, and we want to answer each LCP query for different $P_{i,c}$ in $O(\log\log n)$ time.

We show that we can achieve it as follows. We augment the original LCP data structure with an *measured ancestor data structure* [2] which supports the following query in $O(\log\log n)$ time: given a leaf y in $\mathcal{T}'$ and a length h, report the location at which the prefix of y of length $|y| - h$ ends, i.e., the location with length h above the leaf. The measured ancestor data structure takes $O(\ell \log n)$ bits. In each node u, we also store the length from the root of $\mathcal{T}'$ to u, and a pointer to the leftmost leaf of u. It takes $O(\ell \log n)$ bits.

Lemma 14 *Let $\mathcal{T}'$ be a compact trie comprising ℓ suffixes of $S[1..n]$. We can build an LCP data structure for $\mathcal{T}'$ that occupies $O(\ell \log n)$ bits. Then, given any pattern P, we can preprocess P in $O(m)$ time, and every subsequent LCP query $\mathrm{LCP}(P_{i,c}, x, u)$ can be answered in $O(\log\log n)$ time.*

Proof The case for $x \geq i$ is straightforward. Assume $x < i$. We first perform an LCP query $\mathrm{LCP}(P, x, u)$. Let v be the location reported, which may be a node or a location on an edge. Let a be the length from u to v. Note that a can be found in $O(1)$ time using the length information stored in each node. Now, if $a \leq i - x$, $P_{i,c}$ diverges at the same point as P, so $\mathrm{LCP}(P_{i,c}, x, u) = v$. Otherwise, we go to the position $i - x$ characters below u, along the path from u to v. It can be done in $O(\log\log n)$ time using the leftmost leaf information and the measured ancestor data structure. Then, we align $P_{i,c}[i]$ with $\mathcal{T}'$ for 1 character and let w' be the location 1 character below w. Finally, we align $P_{i,c}[i+1..m]$, which is equivalent to $P[i+1..m]$, with $\mathcal{T}'$ using an LCP query. The whole process takes $O(\log\log n)$ time. □

With the enhanced LCP data structure, we can perform the three steps as follows.

Step 1. We build the enhanced LCP data structure for the suffix tree $\mathcal{T}$. At each node u in $\mathcal{T}$, we also store a pointer to the root of the centroid path passing through u. It takes $O(n \log n)$ bits. Then, Step 1 can be done by an LCP query with $P_{i,c}[1..m]$ and the root of $\mathcal{T}$. By following the pointer that points to the roots of the centroid paths, Step 1 takes $O(\log \log n + w)$ time, where w is the number of centroid paths found.

Step 2. Recall that $[d, e]$ is the interval such that every modified suffixes with *lex_order* between $[d, e]$ have $P_{i,c}$ as a prefix. Finding $[d, e]$ can be done by aligning $P_{i,c}$ with N starting from the root of N, or equivalently, an LCP query for $P_{i,c}$ and N. To support it efficiently, we build a compact trie $\mathcal{T}''$ consisting of all suffixes of $\mathcal{T}$ and N. Note that $\mathcal{T}''$ has some subtrees that are not found in $\mathcal{T}$. Consider one such subtree T. T is formed by suffixes in N that diverge from $\mathcal{T}$ at the same location u and have the same character just after location u. Let u' be the location one character below u. We observe that each suffix from u' to a leaf is a suffix of $\mathcal{T}$. So we can build the above LCP data structure for this subtree rooted at u'. We build an LCP data structure for each other subtree that forms by suffixes from N. We also build a measured ancestor data structure for $\mathcal{T}''$. It takes totally $O(n \log n)$ bits.

To perform the LCP query for $P_{i,c}$ and $\mathcal{T}''$, we first perform an LCP query for $P_{i,c}$ and $\mathcal{T}$. Let u on $\mathcal{T}$ be the location reported. We can find the corresponding location in $\mathcal{T}''$ by first locating the corresponding leftmost leaf of u, and then use the measured ancestor data structure on $\mathcal{T}''$. If there is a subtree leaving v on $\mathcal{T}''$ such that the first character on the outgoing edge matches that on $P_{i,c}$, we can align one character, followed by an LCP query on that subtree to find the location u' where $P_{i,c}$ diverges from $\mathcal{T}''$. The interval $[d, e]$ is simply the *lex_order* of the leaves below u'. The whole process takes $O(\log \log n)$ time.

Step 3. We perform an LCP query for each side edges that $P_{i,c}[1..i-1]$ overlaps. Since there at most $\log n$ such side edges, it takes $O(\log n \log \log n + occ')$ time, where occ' is the number of matches found.

4.2 Candidate Generation with Specific Error Location

Step 4 is challenging because generating candidates involves *range_search* queries on the *lex-order*$_{\mathcal{C}}$ 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 $P_{i,c}[1..i-1]$. This causes problem, e.g., an exact match of P in S will be reported for each $P_{i,c}$, leading to a term of $m \cdot occ$, instead of occ, in the searching time.

To generate only the candidates with modified position in the required range, we store a *modified*$_{\mathcal{C}}$ array for each centroid path $\mathcal{C}$. It records 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 [19] to ensure generating candidates with the modified positions within $P_{i,c}[1..i-1]$. Details are as follows.

Consider a 1-error tree $E_{\mathcal{C}}$ with ℓ leaves. The array *modified*$_{\mathcal{C}}[1..\ell]$ is defined such that *modified*$_{\mathcal{C}}[j] = e$ if the j-th modified suffix in $E_{\mathcal{C}}$ is formed by modifying the e-th character of a suffix in $\mathcal{T}$. We do not need to store *modified*$_{\mathcal{C}}[j]$ explicitly. Instead, it can be calculated in $O(1)$ time as follows: For the j-th modified suffix, recall that

we can find in $O(1)$ time the side tree that contains the original suffix, by using the *st-rank*$_{\mathcal{C}}$ and *tree_pointer*$_{\mathcal{C}}$ arrays. Hence, for each side tree T in the suffix tree $\mathcal{T}$, we store the total length x from the root of $\mathcal{T}$ to the node u to which T is hanging. Then, *modified*$_{\mathcal{C}}[j] = x + 1$. Since each side tree stores the length information only once, it takes $O(n \log n)$ bits.

Given an interval $[p, q]$ in the *lex-order*$_{\mathcal{C}}$ array that corresponds to modified suffixes with a prefix $P_{i,c}$, we only consider the j-th modified suffix as a candidate if $p \le j \le q$ and *modified*$_{\mathcal{C}}[j] < i$. To locate the candidates efficiently, we build the following Bounded Value Range Query (BVRQ) data structure for *modified*$_{\mathcal{C}}[1..\ell]$, using $O(\ell)$-bit space. The total space for all BVRQ data structures over all centroid paths is $O(n \log n)$ bits.

Lemma 15 *Given any integer array $M[1..\ell]$, we can build an $O(\ell)$-bit data structure such that for any integers p, q and z, it answer in $O(occ)$ time the following* BVRQ(p, q, z) *query*: *report all indices j satisfying $p \le j \le q$ and $M[j] \le z$, where occ is the number of indices reported.*

Proof As shown in [19], we can reduce the BVRQ(p, q, z) query to the range minimum query (RMQ), where RMQ(a, b), for any integer a, b, returns any i such that $M[i] = \min_{j=a}^{b} M[j]$. BVRQ$(p, q, z)$ takes $O(occ)$ time if RMQ(a, b) can be answered in $O(1)$ time.

An $O(\ell \log n)$-bit data structure supporting the RMQ query in $O(1)$ time is given in [3]. It constructs a binary tree in T with ℓ nodes recursively as follows: Each entry in $M[j]$ corresponds to a node u_j in T. Let $M[x]$ be the minimum entry in M and u_x be the corresponding node. The root of T is u_x, and two trees T_1 and T_2 are built recursively for $M[1..x-1]$ and $M[x+1..\ell]$. The root of T_1 (resp. T_2) becomes the left child (resp. right child) of u_x. Note that for any j, the node u_j corresponding to $M[j]$ has in-order j in the tree T, and RMQ(a, b) corresponds to the lowest common ancestor of u_a and u_b, denoted as $lca(u_a, u_b)$.

We note that the space requirement can be reduced to $O(\ell)$ bits, as follows. We use parenthesis encoding [18] to represent the tree T in $O(\ell)$ bits, which supports finding the node u_j with in-order j and calculating the in-order of a node u in $O(1)$ time. Then, we build an $O(\ell)$-bit LCA data structure [22] for the parenthesis encoding of the tree, so that finding $w = lca(u_a, u_b)$ for any node u_a and u_b can be done in $O(1)$ time.

Then, RMQ(a, b) can be done in $O(1)$ time as follows: We first find the nodes u_a and u_b corresponding to $M[a]$ and $M[b]$. We calculate $w = lca(u_a, u_b)$ and find the in-order of w. It takes $O(1)$ time. □

In conclusion, given a range $[p, q]$ in the *lex-order*$_{\mathcal{C}}$ array, we can report only the modified suffixes with modified position in $P_{i,c}[1..i-1]$ in $O(occ')$ time, where occ' is the number of modified suffixes reported.

4.3 Constant Time Verification

Given a suffix x in $\mathcal{T}$, we want to verify in $O(1)$ time whether x is a 1-error match for $P_{i,c}$. It is non-trivial because only P, but not $P_{i,c}$, is preprocessed. To support

$O(1)$ verification time, we exploit the properties of suffix trees and the inverse suffix arrays, as follows.

Lemma 16 *We can build an $O(n)$-bit data structure for $\mathcal{T}$. For any $P[1..m]$, we can preprocess P in $O(m)$ time. Then for any suffix x in $\mathcal{T}$, we can verify in $O(1)$-time whether x has a prefix that is a 1-error match of $P_{i,c}$ with the error in $P_{i,c}[1..i-1]$.*

Proof We build an $O(n)$-bit lowest common ancestor data structure [22] for $\mathcal{T}$. Given any two nodes u and v in $\mathcal{T}$, it returns in $O(1)$ time the lowest common ancestor, denoted $lca(u, v)$, of u and v. Given P, we preprocess P by finding, for each suffix $P[j..m]$, a suffix y_j in $\mathcal{T}$ that has the longest common prefix (or lcp for short) with $P[j..m]$.

To verify a suffix x in $\mathcal{T}$, we let $S[a_x..n]$ be the corresponding suffix. We first find the lcp between $S[a_x..n]$ and $P_{i,c}[1..m]$, by finding the lcp between $S[a_x..n]$ and $P[1..m]$ using the query $lca(x, y_1)$. It locates the first error between $S[a_x..n]$ and $P_{i,c}[1..m]$. Let $P_{i,c}[h]$ be error position. We return false if $h > i-1$. To check whether $S[a_x + h..n]$ has $P_{i,c}[h+1..m]$ as a prefix, we note that the leaf x' in $\mathcal{T}$ corresponding to the suffix $S[a_x + h..n]$ can be found by $SA^{-1}[a_x + h]$. So the lcp of $S[a_x + h..n]$ and $P_{i,c}[h+1..m]$ can be found by performing $lca(x', y_{h+1})$. It should locate the second error at the i-th position. We finally find the lcp of $S[a_x + i..n]$ and $P_{i,c}[i+1..m]$ to confirm the 1-error match. The whole process takes $O(1)$ time. □

4.4 Remarks

Extension to Edit Distance We handle deletion and insertion separately. For deletion, for each $i \le m$, we delete $P[i]$ to form a new pattern P'_i. Then we follow the above framework to find all 1-error matches of P'_i with the error in $P'_i[1..i-1]$. Handling insertion is similar. The search time remains $O(m \log n \log\log n + occ)$.

Extension to k Errors We first discuss the case for Hamming distance. To find the k-error matches for a pattern $P[1..m]$, we first generate a modified pattern P' by modifying $k-2$ characters of P. Let $i_3 < \cdots < i_k$ be the modified positions. We preprocess P' in $O(m)$ time. Let $P'_{i_2,c}$, where $i_2 < i_3$, be a pattern obtained by modifying the i_2-th character of P', substituting it with a character c. Following the above approach, we can find the 1-error matches of $P'_{i_2,c}$ with the error in $P'_{i_2,c}[1..i_2-1]$ in $O(\log n \log\log n + occ'_{i_2,c})$ time, where $occ'_{i_2,c}$ is the number of matches found. Note that these are distinct k-error matches of P. There are $O(|\Sigma|^{k-2}m^{k-2})$ choices of P' and $O(|\Sigma|^{k-1}m^{k-1})$ possible $P'_{i_2,c}$, so the total time complexity is $O(|\Sigma|^{k-1}m^{k-1} \log n \log\log n + occ)$, where occ is the number of k-error matches of P.

Handling edit distance is similar. We generate all possible patterns P' that have $k-2$ errors with P. Note that at each position of P, we can modify it by inserting a character, substituting a character or deleting a character, so there are $O((2|\Sigma|)^{k-1}m^{k-1})$ possible P'. For each P', we find the 2-error matches separately in $O(2|\Sigma|m \log n \log\log n + occ')$ time. Therefore, the total searching time is $O((2|\Sigma|)^{k-1}m^{k-1} \log n \log\log n + occ)$ time.

References

1. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Indexing and dictionary matching with one error. In: Proceedings of Workshop on Algorithms and Data Structures, pp. 181–192 (1999)
2. Amir, A., Landau, G., Lewenstein, M., Sokol, D.: Dynamic pattern, static text matching. In: Proceedings of Workshop on Algorithms and Data Structures, pp. 340–352 (2003)
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proceedings of Theoretical Informatics, 4th Latin American Symposium, pp. 88–94 (2000)
4. Buchsbaum, A.L., Goodrich, M.T., Westbrook, J.R.: Range searching over tree cross products. In: Proceedings of European Symposium on Algorithms, pp. 120–131 (2000)
5. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: A linear-size index for approximate pattern matching. In: Proceedings of Combinatorial Pattern Matching, pp. 49–59 (2006)
6. Chavez, E., Navarro, G.: A metric index for approximate string matching. In: Proceedings of Latin American Theoretical Informatics, pp. 181–195 (2002)
7. Cobbs, A.: Fast approximate matching using suffix trees. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 41–54 (1995)
8. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of Symposium on Theory of Computing, pp. 91–100 (2004)
9. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of Symposium on Foundations of Computer Science, pp. 390–398 (2000)
10. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of Symposium on Theory of Computing, pp. 397–406 (2000)
11. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
12. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 434–444 (2004)
13. Lam, T.W., Sung, W.K., Wong, S.S.: Improved approximate string matching using compressed suffix data structures. Algorithmica **51**, 298–314 (2008)
14. Maaß, M.G., Nowak, J.: Text indexing with errors. Technical Report TUM-10503, Fakultät für Informatik, TU München (Mar. 2005)
15. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
16. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976)
17. Munro, J.I.: Tables. In: Proceedings of Conference on Foundations of Software Technology and Computer Science, pp. 37–42 (1996)
18. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. J. Comput. **31**(3), 762–776
19. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proceedings of 13th Symposium on Discrete Algorithms, pp. 657–666 (2002)
20. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. J. Discrete Algorithms **1**(1), 205–209 (2000). Special issue on Matching Patterns
21. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms, pp. 233–242 (2002)
22. Sadakane, K.: Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms, pp. 225–232 (2002)
23. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
24. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. Inf. Process. Lett. **17**(2), 81–84 (1983)