A Linear Size Index for Approximate Pattern Matching

Ho-Leung Chan[1], Tak-Wah Lam[1], Wing-Kin Sung[2],
Siu-Lung Tam[1], and 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. This paper revisits the problem of indexing a text $S[1..n]$ to support searching substrings in S that match a given pattern $P[1..m]$ with at most k errors. A naive solution either has a worst-case matching time complexity of $\Omega(m^k)$ or requires $\Omega(n^k)$ space. Devising a solution with better performance has been a challenge until Cole et al. [5] showed an $O(n \log^k n)$-space index that can support k-error matching in $O(m+occ+\log^k n \log \log n)$ time, where occ is the number of occurrences. Motivated by the indexing of DNA, we investigate in this paper the feasibility of devising a linear-size index that still has a time complexity linear in m. In particular, we give an $O(n)$-space index that supports k-error matching in $O(m + occ + (\log n)^{k(k+1)} \log \log n)$ worst-case time. Furthermore, the index can be compressed from $O(n)$ words into $O(n)$ bits with a slight increase in the time complexity.

1 Introduction

In this paper, we consider the indexing problem for k-approximate matching: given an integer $k \geq 0$ and a text $S[1..n]$ over a finite alphabet Σ, we want to build an index for S such that for any query pattern $P[1..m]$, we can report efficiently all locations in S that match P with at most k errors. The number of errors is measured in terms of either the Hamming distance (number of character substitutions) or the edit distance (number of character substitutions, insertions or deletions). The major concern is how to achieve efficient matching without using a large amount of space for indexing. Typical applications include the indexing of DNA or protein sequences for biological research.

To support exact matching (i.e., $k = 0$), suffix trees and suffix arrays are the most well-known indexes. Suffix trees [15,12] occupy $O(n)$ space and achieve the optimal matching time, i.e., $O(m + occ)$, where occ is the number occurrences of P in S.[1] For suffix arrays [11], the space requirement is also $O(n)$ space (but with a smaller constant), and the matching time is $O(m + occ + \log n)$. Recently, two

---

[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.

compressed solutions, namely, compressed suffix arrays [7] and FM-index [6], have been proposed; they requires $O(n)$ bits only and the matching time is $O(m + occ \log^\epsilon n)$, where $\epsilon > 0$.

Indexing a string for approximate matching is a challenging problem. Even the special case where only one error is allowed (i.e., $k = 1$) has attracted a lot of attention. A simple solution is to use the suffix tree of S and repeatedly search for every 1-error modification of the query pattern; this solution uses $O(n)$ space and the matching time is $O(m^2 + occ)$ [4]. With a bigger index of size $O(n \log n)$, the matching time complexity has been improved tremendously by a chain of results to $O(m \log n \log \log n + occ)$ [1], $O(m \log \log n + occ)$ [2], and finally $O(m + occ + \log n \log \log n)$ [5]. It is also known that indexes using $O(n)$ space takes $O(m \log n + occ)$ time [8] and $O(m \log \log n + occ)$ time [9] for 1-error matching. These two indexes can also be compressed to $O(n)$ bits, and the 1-error matching time is $O(m \log^2 n + occ \log n)$ and $O((m \log \log n + occ) \log^\epsilon n)$, respectively, where $\epsilon < 1$.

To cater for $k = O(1)$ errors, one can perform a brute-force search on an one-error index (i.e., repeatedly modify the pattern at different $k - 1$ positions and search for one-error matches); the matching becomes very inefficient, involving a factor of m^k in the time complexity. Alternatively, one can improve the matching time by including all possible erroneous substrings into the index; yet this seems to require $\Omega(n^k)$ space. It has been open whether there exists an index with performance better than a navie solution. The breakthrough is due to Cole et al. [5], who are able to avoid brute-force matching of a pattern with a moderate increase in the index size. Precisely, their index occupies $O(\frac{d^k}{k!} n \log^k n)$ space and supports k-error matching in $O(m + occ + \frac{c^k}{k!} \log^k n \log \log n)$ time for Hamming distance, where d and c are some constants. The term occ is replaced with $occ \cdot 3^k$ for edit distance. This solution gives an obvious improvement to the matching efficiency. The space requirement is acceptable for many applications, but it may be too demanding for indexing DNA sequences or webpages. [2]

In this paper, we focus on indexes that use only $O(n)$ words or $O(n)$ bits for k-error matching, and we hope that the time complexity can be better than $O(m^k)$. Prior to our work, indexes using $O(n)$ words to answer a k-error query takes $O((cm)^k \log n + occ))$ time [8] or takes $O((cm)^k \log \log n + occ))$ time [9]. Indexes using $O(n)$ bits have a slightly worse time complexity [8,9]. See Table 1 for a summary of results. The main results of this paper are as follows.

(i) We give an $O(n)$-word index that supports k-error matching in $O(m + occ + (c \log n)^{k(k+1)} \log \log n)$ time, where c is a constant. Furthermore, if the pattern is known to be long (precisely, $\Omega(\log^{k+1} n)$), the matching time can be improved to $O(m + occ + (c \log n)^{2k+1} \log \log n)$. The term occ becomes $occ \cdot k^3 3^k$ if edit distance is in concern.

---

[2] For example, consider $k = 2$, the index requires $O(n \log^2 n)$ words, which means tens of gigabytes of memory for a text of a few million characters. Indexing a human chromosome or genome (typically a few hundred million to a few billion characters) is not feasible.

Table 1. Known results for k-error matching. Results given in this paper are marked with †. c and ϵ are positive constants.

Space	$k = 1$	
$O(n \log^2 n)$ words	$O(m \log n \log \log n + occ)$	[1]
$O(n \log n)$ words	$O(m \log \log n + occ)$	[2]
	$O(m + occ + \log n \log \log n)$	[5]
$O(n)$ words	$O(\min\{n, m^2\} + occ)$	[4]
	$O(m \log n + occ)$	[8]
	$O(m \log \log n + occ)$	[9]
	$O(m + occ + \log^3 n \log \log n)$	†
$O(n)$ bits	$O(m \log^2 n + occ \log n)$	[8]
	$O((m \log \log n + occ) \log^\epsilon n)$	[9]
	$O((m + occ + \log^4 n \log \log n) \log^\epsilon n)$	†

Space	$k \geq 2$	
$O(n \log^k n)$ words	$O(m + occ + \frac{1}{k!}(c \log n)^k \log \log n)$	[5]
$O(n)$ words	$O(\min\{n, m^{k+1}\} + occ)$	[4]
	$O((cm)^k \log n + occ)$	[8]
	$O((cm)^k \log \log n + occ)$	[9]
	$O(m + occ + (c \log n)^{k(k+1)} \log \log n)$	†
$O(n)$ bits	$O((cm)^k \log^2 n + occ \log n)$	[8]
	$O(((cm)^k \log \log n + occ) \log^\epsilon n)$	[9]
	$O((m + occ + (c \log n)^{k(k+2)} \log \log n) \log^\epsilon n)$	†

This index also admits a simple tradeoff between space and time. I.e., the matching can be speeded up if more space is used. Roughly speaking, for any $h \leq k$, if $O(n \log^{k-h+1} n)$ space is used, then a k-error query can be answered in $O(m + occ + c^{k^2} \log^{\max\{kh, k+h\}} n \log \log n)$ time. For example, choosing $h = 3$ gives an $O(n \log^{k-2} n)$-word index with matching time $O(m + occ + c^k \log^{3k} n \log \log n)$.

(ii) The $O(n)$-word index can be compressed to occupy $O(n)$ bits only, with k-error matching time increasing to $O((m + occ + (c \log n)^{k(k+2)} \log \log n) \log^\epsilon n)$, where $\epsilon < 1$. In particular, when $k = 1$, the $O(n)$-bit index achieves matching in $O((m + occ + \log^4 n \log \log n) \log^\epsilon n)$ time.

Other related results. Note that the above results concern worst-case performance. The literature also contains several interesting results on average-case performance (see, e.g., [13, 10, 3]).

2 An $O(n)$-Word Index for k-Error Matching

This section considers Hamming distance only and presents an $O(n)$-word index for a text $S[1..n]$. Given any pattern $P[1..m]$, the index finds all substrings of S matching P within k errors, in $O(m + occ + \text{polylog } n)$ time. We call these substrings the k-error matches of P.

The index handles long patterns and short patterns separately. Intuitively, short patterns can be handled easily. For example, a pattern of length $\log n$ can be handled in polylog n time even with the naive $\Omega(m^k)$ time methods. The main novelty of our index is a check-point technique for handling long patterns: we define some locations of S to be *check-points*. Special indexing are done for suffixes and prefixes of S terminating at these check-points. For long patterns, their k-error matches in S will certainly contain some check-points, so the special indexing at the check-points suffices for finding the matches efficiently.

We now describe how to handle long patterns. Consider a text $S[1..n]$. Let β be a positive integer, which will be fixed later to $k3^k \log^{k+1} n$. Intuitively, a pattern is long if its length is at least β. For each $a = \beta, 2\beta, 3\beta, \ldots$, we call $S[a]$ a check-point.

Observation 1. *Let $P[1..m]$ be a pattern with $m \geq \beta$. For any k-error match $S[j_1..j_2]$ of P, there exists an integer a, $j_1 \leq a \leq j_2$ such that $S[a]$ is a check-point and $0 \leq a - j_1 \leq \beta - 1$.*

Furthermore, let $i = a - j_1 + 1$. There exist integers $k_1, k_2 \geq 0$, such that (1) $S[a..n]$ has a prefix matching $P[i..m]$ with k_1 errors, (2) $S[1..a-1]$ has a suffix matching $P[1..i-1]$ with k_2 errors, and (3) $k_1 + k_2 \leq k$.

Let TAIL be the set of suffixes of S beginning at a check-point, i.e., TAIL $= \{S[a..n] \mid a = \beta, 2\beta, \ldots\}$. Similarly, let HEAD be the set of prefixes of S ending just before a check-point, i.e., HEAD $= \{S[1..a-1] \mid a = \beta, 2\beta, \ldots\}$. Observation 1 suggests finding the k-error matches of P as follows.

Algorithm 1. k-MATCH(P): finds all k-error matches of P in S, for $|P| \geq \beta$.

For each $i = 1, \ldots, \beta$, cut P into $P[1..i-1]$ and $P[i..m]$. Try all possible $k_1, k_2 \geq 0$ such that $k_1 + k_2 \leq k$, and perform the following.

Step 1. Find all $S[a..n] \in$ TAIL that have a prefix matching $P[i..m]$ with exactly k_1 errors. Let tail_{i,k_1} be the set of these suffixes.

Step 2. Find all $S[1..b] \in$ HEAD that have a suffix matching $P[1..i-1]$ with exactly k_2 errors. Let head_{i,k_2} be the set of these prefixes.

Step 3. For each $S[a..n] \in \text{tail}_{i,k_1}$ and $S[1..b] \in \text{head}_{i,k_2}$, we call them a *connecting pair* if $a = b + 1$. For each connecting pair, we report a k-error match of P starting at $S[a - i + 1]$.

We first prove the correctness of the algorithm. Details of the implementation are given in the coming subsections.

Lemma 1. *Let $P[1..m]$ be a pattern with $m \geq \beta$. k-MATCH(S, P) finds all k-error matches of P in S.*

Proof. For each k-error match $S[j_1..j_2]$ of P, Observation 1 states that there is a check-point $S[a]$ contained in $S[j_1..j_2]$ and $0 \leq a - j_1 \leq \beta - 1$.

Consider aligning $P[1..m]$ with $S[j_1..j_2]$. The suffix $S[a..n]$ has a prefix matching $P[i'..m]$ with k_1' errors, where $i' = a - j_1 + 1$ and k_1' is some integer between

0 and k. Thus, $S[a..n]$ will be included in $\text{tail}_{i',k_1'}$. Similarly, $S[1..a-1]$ will be included in $\text{head}_{i',k_2'}$, where k_2' is some integer between 0 and k, and $k_1'+k_2' \le k$. They form a connecting pair, so $S[j_1..j_2]$ will be reported.

There are only n/β suffixes and prefixes in TAIL and HEAD, respectively, so we can build more complicated data structures to support the above steps efficiently, while maintaining a small space requirement. In the following subsections, we present the actual data structures. Then, we will give analysis for the total space and time complexity of the index.

2.1   Indexes for Finding Tail_{i,k_1} and Head_{i,k_2}

We want to find tail_{i,k_1} efficiently for any pattern $P[1..m]$, $i = 1, \ldots, \beta$ and $k_1 = 0, \ldots, k$. We do it by storing an ℓ-error-tree [5] for TAIL, for each $\ell = 0, \ldots, k$. The performance guarantee provided by an ℓ-error tree is stated in the following lemma.

Lemma 2.   [5] Let Z be any collection of suffixes of a text $S[1..n]$. For any integer $\ell \ge 0$, an ℓ-error-tree for Z has the following properties.

1. The ℓ-error tree is a collection of trees with totally $O(|Z|3^\ell \log^\ell n)$ nodes. Each leaf represents a suffix in Z and at most $O(3^\ell \log^\ell n)$ leaves represent the same suffix.
2. The ℓ-error tree takes $O(|Z|3^\ell \log^\ell n)$-word space.
3. For any pattern $Q[1..m']$, there exist $O(6^\ell \log^\ell n)$ nodes in the ℓ-error-tree, such that each leaf under the nodes represents a distinct suffix in Z that has a prefix matching Q with exactly ℓ errors. It takes $O(6^\ell \log^\ell n \log \log n)$ time to find these nodes, after preprocessing all suffixes of Q with the suffix tree of S in totally $O(m')$ time.

For each $\ell = 0, 1, \ldots, k$, We store an ℓ-error tree for TAIL, calling them T-error-tree$_0$, T-error-tree$_1$, …, T-error-tree$_k$. Furthermore, we store a suffix tree for S.

For any i and k_1, the above lemma implies that there exist $O(6^{k_1} \log^{k_1} n)$ nodes in T-error-tree$_{k_1}$ such that the leaves under them represent the distinct suffixes in tail_{i,k_1}. We called these nodes the *covering nodes* for tail_{i,k_1}. For time efficiency, we will not find tail_{i,k_1} explicitly, instead we only find the covering nodes to represent tail_{i,k_1} implicitly. Using the error-tree data structures, we have the following performance on finding the covering nodes.

Lemma 3. *We can build an $O(n + n/\beta \times 3^k \log^k n)$-word data structure for TAIL. For any pattern $P[1..m]$, we preprocess P in $O(m)$ time. Then, for any $i = 1, \ldots, \beta$ and $k_1 = 0, \ldots, k$, we can find $O(6^{k_1} \log^{k_1} n)$ covering nodes for tail_{i,k_1} in T-error-tree$_{k_1}$ in $O(6^{k_1} \log^{k_1} n \log \log n)$ time.*

Proof. The suffix tree of S takes $O(n)$ words and T-error-tree$_0$, T-error-tree$_1$, …, T-error-tree$_k$ take totally $\sum_{\ell=0}^{k} O(n/\beta \times 3^\ell \log^\ell n) = O(n/\beta \times 3^k \log^k n)$ words.

Given any $P[1..m]$, we preprocess all suffixes of P with the suffix tree of S in totally $O(m)$ time. It implies preprocessing all suffixes of $P[i..m]$ with the suffix tree. Thus, finding the covering nodes for tail_{i,k_1} can be done in $O(6^{k_1} \log^{k_1} n \log \log n)$ time using T-error-tree$_{k_1}$.

Note that there can be more than one set of covering nodes for tail_{i,k_1}, and any set of covering nodes is sufficient for our algorithm to find the k-error matches of P.

The case for finding head_{i,k_2} is symmetric. For each $\ell = 0, 1, \ldots, k$, we store an ℓ-error-tree for HEAD, calling them H-error-tree$_0$, ..., H-error-tree$_k$. We also store the suffix tree for the reverse of S. Finding covering nodes for head_{i,k_2}, for any i and k_2 takes $O(6^{k_2} \log^{k_2} n \log \log n)$ time, after an $O(m)$ time preprocessing of P with the suffix tree for the reverse of S.

2.2   Indexes for Finding Connecting Pairs

Consider certain i, k_1 and k_2 where $k_1 + k_2 \leq k$. Assume that tail_{i,k_1} is found implicitly, represented by a set of covering nodes U in T-error-tree$_{k_1}$. Similarly, assume that head_{i,k_2} is represented by a set of covering nodes W in H-error-tree$_{k_2}$. To find the k-error matches of P, we want to find all suffixes $S[a..n] \in \text{tail}_{i,k_1}$ and prefixes $S[1..b] \in \text{head}_{i,k_2}$ that are connecting pairs, i.e., $a = b + 1$.

We observe that this can be done as follows. We preprocess T-error-tree$_{k_1}$ with H-error-tree$_{k_2}$. For each leaf in T-error-tree$_{k_1}$ representing a suffix $S[a..n]$ and for each leaf in H-error-tree$_{k_2}$ representing a prefix $S[1..b]$, we draw an imaginary edge between them if $a = b + 1$. Then, to find the connecting pairs between tail_{i,k_1} and head_{i,k_2}, we try each pair of $u \in U$ and $w \in W$ and perform the following EdgeReport(u, w) query: Given $u \in U$ and $w \in W$, find all leaf pairs (x, y) such that x and y are descendents of u and w, respectively, and x, y are connected by an imaginary edge.

While T-error-tree$_{k_1}$ is a collection of trees, we can always convert it into a single tree by linking all trees to a new root. Similarly, we convert H-error-tree$_{k_2}$ into a single tree. Then, we store a tree-cross-product data structure [2] for T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$ to support the EdgeReport(u, w) query efficiently, which has the following performance.

Lemma 4. *[2] Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two trees. Let $V = V_1 \cup V_2$ and let $I \subseteq V_1 \times V_2$ be a set of imaginary edges connecting some nodes in V_1 and V_2. We can build an $O(|I| \log |V|)$-word index for T_1 and T_2 such that for any $u \in V_1$ and $w \in V_2$, the EdgeReport(u, w) query takes $O(\log \log |V| + occ')$ time, where occ' is the number of imaginary edges reported.*

For each pair of error-trees T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$, where $k_1 + k_2 \leq k$, we create the imaginary edges and build the tree-cross-product data structure. It allows us to find the connecting pairs efficiently. We assume that tail_{i,k_1} and head_{i,k_2} are represented by $O(6^{k_1} \log^{k_1} n)$ and $O(6^{k_2} \log^{k_2} n)$ covering nodes in the corresponding error-trees, respectively, which is the case during the execution of the Algorithm k-MATCH.

Lemma 5. *We can store an $O(k \times n/\beta \times 3^k \log^{k+1} n)$-word data structure for the error-trees. Then, for any i, k_1 and k_2 with $k_1+k_2 \le k$, we can find all connecting pairs between $tail_{i,k_1}$ and $head_{i,k_2}$ in $O(6^{k_1+k_2} \log^{k_1+k_2} n \log \log n + occ')$ time, where occ' is the number of connecting pairs found.*

Proof. Consider T-error-tree$_{k_1}$ and H-error-tree$_{k_2}$, where $k_1+k_2 = c$ for some $c \le k$. There are $O(n/\beta \times 3^{k_1} \log^{k_1} n)$ leaves in T-error-tree$_{k_1}$. For each leaf representing a suffix $S[a..n]$, the prefix $S[1..a-1]$ is represented by at most $O(3^{k_2} \log^{k_2} n)$ leaves in H-error-tree$_{k_2}$. So, the number of imaginary edges between the two error-trees is $O(n/\beta \times 3^{k_1+k_2} \log^{k_1+k_2} n)$, and the tree-cross-product data structure takes $O(n/\beta \times 3^c \log^{c+1} n)$ words. For any c, there are at most $k + 1$ pairs of possible (k_1, k_2), and we store tree-cross product data structures for $c = 0, 1, \ldots, k$, so the total space needed is $\sum_{c=0}^{k} O(k \times n/\beta \times 3^c \log^{c+1} n) = O(k \times n/\beta \times 3^k \log^{k+1} n)$ words.

For any $tail_{i,k_1}$ and $head_{i,k_2}$, where $k_1 + k_2 \le k$, let U and W be the corresponding set of covering nodes. Finding the connecting pairs is done by performing an EdgeReport(u, v) query for each $u \in U$ and $w \in W$. There are $O(6^{k_1} \log^{k_1} n \times 6^{k_2} \log^{k_2} n)$ queries, and the total query time is $O(6^{k_1+k_2} \times \log^{k_1+k_2} n \log \log n + occ')$ time.

2.3   Total Time and Space Complexity

With Lemma 3 and 5, we can analyse the space and time complexity of our data structure.

Theorem 1. *We can build an $O(n + k \times n/\beta \times 3^k \log^{k+1} n)$-word index for $S[1..n]$. For any pattern $P[1..m]$, $m \ge \beta$, we can find all k-error matches of P in S in $O(m + occ + \beta k 6^k \log^k n \log \log n)$ time, where occ is the number of matches.*

Proof. We only need to store the data structures specified in Lemma 3 and 5, so the total space is $O(n + k \times n/\beta \times 3^k \log^{k+1} n)$ words.

To find the k-error matches of P, we perform an $O(m)$ time preprocessing of P, as required by Lemma 3. Then, we iterate for $i = 1, 2, \ldots, \beta$ and $c = 0, 1, \ldots, k$. For each i and c, there are at most $k + 1$ pairs of $k_1, k_2 \ge 0$ such that $k_1 + k_2 = c$. Finding the covering nodes for $tail_{i,k_1}$ and $head_{i,k_2}$ takes $O(6^{k_1} \log^{k_1} n \log \log n + 6^{k_2} \log^{k_2} n \log \log n)$. Finding the connecting pairs between $tail_{i,k_1}$ and $head_{i,k_2}$ takes $O(6^{k_1+k_2} \log^{k_1+k_2} n \log \log n + occ')$ time, where occ' is the number of connecting pairs found. Thus, for any fixed i and c, the runtime is $O(k \times 6^c \log^c n \log \log n + occ')$ time.

We try i from 1 to β and c from 0 to k, so the total time complexity is $O(m + \beta \times \sum_{c=0}^{k} k \times 6^c \log^c n \log \log n + occ)$ $O(m + \beta k 6^k \log^k n \log \log n + occ)$.

By putting $\beta = k 3^k \log^{k+1} n$, we obtain an $O(n)$-word index for handling long patterns. For short patterns, we can use the $O(n)$-word data structure of Lam et al. [9] which find the k-error matches of a pattern $P[1..m]$ in $O(|\Sigma|^k m^k \log \log n + occ)$ time, where $|\Sigma|$ is the size of the alphabet.

Corollary 1. *For any constant k, we can build an $O(n)$-word index for $S[1..n]$. For any pattern $P[1..m]$, finding the k-error matches of P in S takes $O(m + occ + (c \log n)^{\max\{k(k+1), 2k+1\}} \log \log n)$ time.*

Proof. We put $\beta = k3^k \log^{k+1} n$ to Theorem 1 to obtain an $O(n)$-word index. We also store the $O(n)$-word data structure of Lam et al. [9].

For pattern of length at least $k3^k \log^{k+1} n$, finding the k-error matches takes $O(m + occ + k^2 18^k \log^{2k+1} n \log \log n)$ time. For pattern of length less than $k3^k \log^{k+1} n$, finding the k-error matches takes $O(occ + |\Sigma|^k m^k \log \log n) = O(occ + |\Sigma|^k k^k 3^{k^2} \log^{k(k+1)} n \log \log n)$ time.

Reducing the polylog n term in matching time. The polylog n term in the matching time is biggest for patterns with length slightly less than $k3^k \log^{k+1} n$, in which we use the brute-force method to obtain a runtime of $O(occ + |\Sigma|^k k^k 3^{k^2} \times \log^{k^2+k} n \log \log n)$. We can reduce the polylog n term by a small trick. To ease the discussion, we remove the constant factors $|\Sigma|$ and k from the asymptotic analysis.

We improve the matching time for patterns of length between $O(\log^k n)$ and $O(\log^{k+1} n)$ by choosing a smaller value of β. In particular, we choose β to be $O(\log^k n)$, but we only build an data structure for finding $(k - 1)$-error matches. By Theorem 1, the index takes only $O(n)$ words. To find the k-error matches, we explicitly try different positions on the pattern and modify that position with a different character. Then, we search for $(k - 1)$-error matches for each of the modified patterns, which will be the k-error matches of the pattern. This gives a runtime of $O(m \times (m + \beta \log^{k-1} n \log \log n + occ)) = O(\log^{2k+2} n + \log^{3k} n \log \log n + occ \times \log^{k+1} n)$. The multiplicative term for occ can be removed by careful book-keeping to avoid reporting the same occurrence for multiple times. It reduces the matching time from $O(occ + \log^{k^2+k} n \log \log n)$ to $O(occ + \max\{\log^{2k+2} n, \log^{3k} n \log \log n\})$, for patterns of length $O(\log^k n)$ to $O(\log^{k+1} n)$.

We can continue to apply this technique for other range of pattern length, and it can reduce the polylog n term in the matching time to $\log^{k^2/2+O(1)k} n$ in the worst case.

3   Tradeoff Between Space and Time

Our data structure allows a tradeoff between space and time. We notice that the value β controls the number of check-points in S, which is equivalent to the number of suffixes of S on which special indexes are built. Choosing a smaller β generates more check-points and increases the index size, but it allows patterns of shorter length to be handled and reduces the matching time. On the other hand, choosing a bigger β reduces the number of check-points such that we can even obtain an $O(n)$-bit data structure for k-error matching, at the cost of increasing the matching time. This section presents the results for this tradeoff.

3.1   Improved Searching with More Space

We choose $\beta = k3^k \log^h n$, where h is any integer, $0 \le h \le k$. Note that a smaller h generates more check-points and bigger index size. By Theorem 1, it gives an $O(n \log^{k-h+1} n)$-word index, which finds the k-error matches of $P[1..m]$, $m \ge k3^k \log^h n$, in $O(m + occ + k^2 18^k \log^{h+k} n \log \log n)$ time.

For patterns of length less than $k3^k \log^h n$, we use the $O(n)$-word data structure of Lam et al. [9], which gives a matching time of $O(|\Sigma|^k k^k 3^{k^2} \log^{hk} n \log \log n + occ)$.

Theorem 2. *For any constant h and k such that $0 \le h \le k$, we can build an index for $S[1..n]$ using $O(n \log^{k-h+1} n)$ space. For any pattern $P[1..m]$, we can find all k-error matches of P in S in $O(m + occ + c^{k^2} (\log n)^{\max\{hk, h+k\}} \log \log n)$ time where occ is the number of occurrences found and c is some constant.*

3.2   Reducing to $O(n)$-Bit Space

We can choose $\beta = k3^k \log^{k+2} n$. Then, the error-trees and the tree-cross-product data structures takes $O(n)$-bit space. We can replace the suffix tree of S by a compressed suffix tree [14], which supports each of the suffix tree operations in $O(\log^\epsilon n)$ time. Thus, the preprocessing of P takes $O(m \log^\epsilon n)$ time. The matching time for pattern of length at least $k3^k \log^{k+2} n$ is $O(m \log^\epsilon n + occ + k^2 18^k \log^{2k+2} n \log \log n)$.

For patterns of length less than $k3^k \log^{k+2} n$, we use the $O(n)$-bit data structure of [9], which gives a matching time of $O((|\Sigma|^k k^k 3^{k^2} \log^{k^2+2k} n \log \log n + occ) \log^\epsilon n)$.

Theorem 3. *For any constant k, we can build an index for $S[1..n]$ using $O(n)$-bit space. For any pattern $P[1..m]$, we can find all k-error matches of P in S in $O((m + occ + (c \log n)^{\max\{k^2+2k, 2k+2\}} \log \log n) \log^\epsilon n)$ time where occ is the number of occurrences reported, c is some constant, and $\epsilon > 0$.*

4 k-Error Matching in Edit Distance

This section considers edit distance, and an error is an insertion, deletion or substitution. We give an $O(n)$-word data structure for $S[1..n]$ which supports finding the k-error matches of P in S. Precisely, given $P[1..m]$, it finds all starting positions j such that $S[j..n]$ has a prefix matching P with at most k errors, in $O(m + k^3 3^k occ + \text{polylog } n)$ time, where occ is the number of starting positions found.

Similar to the case of Hamming distance, we handle long patterns by the check-point technique, while short patterns are handled by simple brute force methods. We define $S[a]$ to be a check-point for $a = \beta, 2\beta, \ldots$, where β will be set later to $k5^k \log^{k+1} n$.

Observation 2. *Let $P[1..m]$ be a pattern with $m \geq \beta + k$. For any k-error match $S[j_1..j_2]$ of P, there exists an integer a, $j_1 \leq a \leq j_2$ such that $S[a]$ is a check-point and $0 \leq a - j_1 \leq \beta - 1$.*

Furthermore, there exist integers i, $1 \leq i \leq \beta + k$ and $k_1, k_2 \geq 0$, such that (1) $S[a..n]$ has a prefix matching $P[i..m]$ with k_1 errors, (2) $S[1..a-1]$ has a suffix matching $P[1..i-1]$ with k_2 errors, and (3) $k_1 + k_2 \leq k$.

Define HEAD and TAIL as before. Observation 2 suggests the following algorithm.

Algorithm 2. k-EDIT(P), find starting positions of k-error matches of P in S, $|P| \geq \beta + k$.

For each $i = 1, \ldots, \beta + k$, cut P into $P[1..i-1]$ and $P[i..m]$. Try all possible $k_1, k_2 \geq 0$ such that $k_1 + k_2 \leq k$, and perform the following.

Step 1. Find all $S[a..n] \in$ TAIL that have a prefix matching $P[i..m]$ with exactly k_1 errors. Let tail_{i,k_1} be the set of these suffixes.

Step 2. Find all $S[1..b] \in$ HEAD that have a suffix matching $P[1..i-1]$ with exactly k_2 errors. Let head_{i,k_2} be the set of these prefixes.

Step 3. For each $S[a..n] \in \text{tail}_{i,k_1}$ and $S[1..b] \in \text{head}_{i,k_2}$, we call them a *connecting pair* if $a = b + 1$. For each connecting pair, we find all j_1 such that $S[j_1..a-1]$ matches $P[1..i-1]$ with exactly k_2 errors, and we report each j_1 as an answer.

To find tail_{i,k_1} and head_{i,k_2} efficiently for different i, k_1 and k_2, we store another type of error-trees by Cole et al. [5] for TAIL and HEAD, which work for edit distance. We call them edit-trees to avoid confusion. Basically, an edit-tree is similar to an error-trees, which is also built for a collection Z of suffixes of S. Given a pattern $Q[1..m']$, an ℓ-edit tree returns the nodes such that the leaves under the nodes represent all suffixes in Z that has a prefix matching Q with exactly ℓ errors (edit distance). However, an edit-tree may give duplicated answers, i.e., there may be different leaves under these nodes representing the same suffix in Z.

We build T-edit-tree$_0$, ..., T-edit-tree$_k$ for TAIL and H-edit-tree$_0$, ..., H-edit-tree$_k$ for HEAD. We also store the suffix trees for S and the reverse of S. Finally, we build the tree-cross-product data structures for the pair T-edit-tree$_{k_1}$ and H-edit-tree$_{k_2}$, for every k_1, k_2. These data structures can support the Algorithm k-EDIT efficiently.

We can analyse the space and time complexity of the data structures similar to that in Section 2 and we obtain the following theorem. There is a $k^3 3^k$ factor for *occ* because when we find tail_{i,k_1} for some i, k_1, the edit-trees may return the same suffix for multiple times, leading to duplication in the output.

Theorem 4. *We can build an $O(n + k \times n/\beta \times 5^k \log^{k+1} n)$-word index for $S[1..n]$. For any pattern $P[1..m]$, $m \geq \beta$, we can find all j such that $S[j..n]$ has a prefix matching P with at most k errors (in edit distance), in $O(m + k^3 3^k occ + \beta k 6^k \log^k n \log \log n)$ time, where occ is the number of answers found.*

By putting $\beta = k5^k \log^{k+1} n$, and handling short patterns by Lam et al. [9] we obtain an $O(n)$-word index which finds the k-error matches in $O(m + k^3 3^k occ + \text{polylog } n)$ time.

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 *Proceedings of 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. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.
7. 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*, pages 397–406, 2000.
8. 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.
9. 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, pages 339–348.
10. M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-10503, Fakultät für Informatik, TU München, Mar. 2005.
11. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
12. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
13. G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *J. Discrete Algorithms*, 1(1):205-209, 2000. Special issue on Matching Patterns.
14. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, accepted.
15. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, 1973, pages 1–11.