Cache-Oblivious Index for Approximate String Matching*

Wing-Kai Hon[1], Tak-Wah Lam[2], Rahul Shah[3], Siu-Lung Tam[2], and Jeffrey Scott Vitter[3]

[1] Department of Computer Science, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Department of Computer Science, The University of Hong Kong, Hong Kong
{twlam,sltam}@cs.hku.hk
[3] Department of Computer Sciences, Purdue University, Indiana, USA
{rahul,jsv}@cs.purdue.edu

Abstract. This paper revisits the problem of indexing a text for approximate string matching. Specifically, given a text T of length n and a positive integer k, we want to construct an index of T such that for any input pattern P, we can find all its k-error matches in T efficiently. This problem is well-studied in the internal-memory setting. Here, we extend some of these recent results to external-memory solutions, which are also cache-oblivious. Our first index occupies $O((n \log^k n)/B)$ disk pages and finds all k-error matches with $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os, where B denotes the number of words in a disk page. To the best of our knowledge, this index is the first external-memory data structure that does not require $\Omega(|P| + occ + \text{poly}(\log n))$ I/Os. The second index reduces the space to $O((n \log n)/B)$ disk pages, and the I/O complexity is $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$.

1 Introduction

Recent years have witnessed a huge growth in the amount of data produced in various disciplines. Well-known examples include DNA sequences, financial time-series, sensor data, and web files. Due to the limited capacity of main memory, traditional data structures and algorithms that perform optimally in main memory become inadequate in many applications. For example, the suffix tree [19,25] is an efficient data structure for indexing a text T for exact pattern matching; given a pattern P, it takes $O(|P| + occ)$ time to report all occurrences of P in T, where occ denotes the number of occurrences. However, if we apply a suffix tree to index DNA, for example, the human genome which has 3 billion characters, at least 64G bytes of main memory would be needed.

* Research of T.W. Lam is supported by the Hong Kong RGC Grant 7140/06E. Research of R. Shah and J.S. Vitter is supported by NSF Grants IIS–0415097 and CCF–0621457, and ARO Grant DAAD 20–03–1–0321. Part of the work was done while W.K. Hon was at Purdue University.

B. Ma and K. Zhang (Eds.): CPM 2007, LNCS 4580, pp. 40–51, 2007.

To solve the problem in dealing with these massive data sets, a natural way is to exploit the external memory as an extension of main memory. In this paradigm of computation, data can be transferred in and out of main memory through an I/O operation. In practice, an I/O operation takes much more time than an operation in main memory. Therefore, it is more important to minimize the number of I/Os.

Aggarwal and Vitter [2] proposed a widely accepted *two-level I/O-model* for analyzing the I/O complexity. In their model, the memory hierarchy consists of a main memory of M words and an external memory. Data reside in external memory initially (as they exceed the capacity of main memory), and computation can be performed only when the required data are present in main memory. With one I/O operation, a disk page with B contiguous words can be read from external memory to main memory, or B words from main memory can be written to a disk page in external memory; the I/O complexity of an algorithm counts only the number of I/O operations involved. To reduce the I/O complexity, an algorithm must be able to exploit the locality of data in external memory. For instance, under this model, sorting a set of n numbers can be done in $O\left(\left(\frac{n}{B}\log\frac{n}{B}\right)/\log\left(\frac{M}{B}\right)\right)$ I/Os, and this bound is proven to be optimal. (See [24] for more algorithms and data structures in the two-level I/O model.)

Later, Frigo et al. [15] introduced the notion of *cache-obliviousness*, in which we do not have advance knowledge of M or B in designing data structures and algorithms for external memory; instead, we require the data structures and algorithms to work for any given M and B. Furthermore, we would like to match the I/O complexity when M and B are known in advance. Among others, cache-obliviousness implies that the algorithms and data structures will readily work well under different machines, without the need of fine tuning the algorithm (or recompilation) or rebuilding the data structures. Many optimal cache-oblivious algorithms and data structures are proposed over the recent years, including algorithms for sorting [20] and matrix transposition [20], and data structures like priority queues [8] and B-trees [7].

For string matching, the recent data structure proposed by Brodal and Fagerberg [9] can index a text T in $O(n/B)$ disk pages[1] and find all occurrences of a given pattern P in T in $O((|P| + occ)/B + \log_B n)$ I/Os. This index works in a cache-oblivious manner, improving the String-B tree, which is an earlier work by Ferragina and Grossi [14] that achieves the same space and I/O bounds but requires the knowledge of B to operate.[2] In this paper, we consider the *approximate string matching* problem defined as follows:

> Given a text T of length n and a fixed positive integer k, construct an index on T such that for any input pattern P, we can find all *k-error matches* of P in T, where a k-error match of P is a string that can be transformed to P using at most k character insertions, deletions, or replacements.

[1] Under the cache-oblivious model, the index occupies $O(n)$ contiguous words in the external memory. The value of B is arbitrary, which is considered only in the analysis.

[2] Recently Bender et al. [6] have devised the cache-oblivious string B-tree, which is for other pattern matching queries such as prefix matching and range searching.

The above problem is well-studied in the internal-memory setting [21,12,3,10,17,11]. Recently, Cole et al. [13] proposed an index that occupies $O(n \log^k n)$ words of space, and can find all k-error matches of a pattern P in $O(|P| + \log^k n \log\log n + occ)$ time. This is the first solution with time complexity linear to $|P|$; in contrast, the time complexity of other existing solutions depends on $|P|^k$. Chan et al. [11] later gave another index that requires only $O(n)$ space, and the time complexity increases to $O(|P| + \log^{k(k+1)} n \log\log n + occ)$. In this paper, we extend these two results to the external-memory setting. In addition, our solution is cache-oblivious.

The main difficulty in extending Cole et al.'s index to the external-memory setting lies in how to answer the longest common prefix (LCP) query for an arbitrary suffix of a pattern P using a few I/Os. More specifically, given a suffix P_i, we want to find a substring of T that is the longest prefix of P_i. In the internal memory setting, we can compute all possible LCP values in advance in $O(|P|)$ time (there are $|P|$ such values) by exploiting the suffix links in the suffix tree of T. Then each LCP query can be answered in $O(1)$ time. In the external memory setting, a naive implementation would require $\Omega(\min\{|P|^2/B, |P|\})$ I/Os to compute all LCP values. To circumvent this bottleneck, we target to compute only some "useful" LCP values in advance (using $O(|P|/B + k \log_B n)$ I/Os), so that each subsequent LCP query can still be answered efficiently (in $O(\log\log_B n)$ I/Os). Yet this target is very difficult to achieve for general patterns. Instead, we take advantage of a new notion called *k-partitionable* and show that if P is k-partitionable, we can achieve the above target; otherwise, T contains no k-error match of P. To support this idea, we devise an I/O-efficient *screening test* that checks whether P is k-partitionable; if P is k-partitionable, the screening test would also compute some useful LCP values as a by-product, which can then be utilized to answer the LCP query for an arbitrary P_i in $O(\log\log_B n)$ I/Os.

Together with other cache oblivious data structures (for supporting LCA, Y-Fast Trie and WLA), we are able to construct an index to find all k-error matches using $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os. The space of the index is $O((n \log^k n)/B)$ disk pages. To the best of our knowledge, this is the first external-memory data structure that does not require $\Omega(|P| + occ + \text{poly}(\log n))$ I/Os. Note that both Cole et al.'s index and our index can work even if the alphabet size is unbounded.

Recall that the internal-memory index by Chan et al. [11] occupies only $O(n)$ space. The reduction of space demands a more involved searching algorithm. In particular, they need the data structure of [10] to support a special query called Tree-Cross-Product. Again, we can 'externalize' this index. Here, the difficulties come in two parts: (i) computing the LCP values, and (ii) answering the Tree-Cross-Product queries. For (i), we will use the same approach as we externalize Cole et al.'s index. For (ii), there is no external memory counter-part for the data structure of [10]; instead, we reduce the Tree-Cross-Product query to a two-dimensional orthogonal range search query, the latter can be answered efficiently using an external-memory index based on the work in [1]. In this way, for any fixed $k \geq 2$, we can construct an index using $O((n \log n)/B)$ disk pages, which

can find all k-error matches of P in $O((|P|+occ)/B+\log^{k(k+1)} n \log\log n)$ I/Os. Following [11], our second result assumes alphabet size is constant.

In Section 2, we give a survey of a few interesting queries that have efficient cache-oblivious solutions. In particular, the index for WLA (weighted level ancestor) is not known in the literature. Section 3 reviews Cole et al.'s internal memory index for k-error matching and discusses how to turn it into an external memory index. Section 4 defines the k-partitionable property, describes the screening test, and show how to compute LCP queries efficiently. Finally, Section 5 states our result obtained by externalizing the index of Chan et al.

2 Preliminaries

2.1 Suffix Tree, Suffix Array, and Inverse Suffix Array

Given a text $T[1..n]$, the substring $T[i..n]$ for any $i \in [1,n]$ is called a suffix of T. We assume that characters in T are drawn from an ordered alphabet which is of constant size, and $T[n] = \$$ is a distinct character that does not appear elsewhere in T. The *suffix tree* of T [19,25] is a compact trie storing all suffixes of T. Each edge corresponds to a substring of T, which is called the *edge label*. For any node u, the concatenation of edge labels along the path from root to u is called the *path label* of u. There are n leaves in the suffix tree, with each leaf corresponding to a suffix of T. Each leaf stores the starting position of its corresponding suffix, which is called the *leaf label*. The children of an internal node are ordered by the lexicographical order of their edge labels.

The *suffix array* of T [18], denoted by SA, is an array of integers such that $SA[i]$ stores the starting position of the ith smallest suffix in the lexicographical order. It is worth-mentioning that SA can also be obtained by traversing the suffix tree in a left-to-right order and recording the leaf labels. Furthermore, the descendant leaves of each internal node u in the suffix tree correspond to a contiguous range in the suffix array, and we call this the SA range of u.

The *inverse suffix array*, denoted by SA^{-1}, is defined such that $SA^{-1}[i] = j$ if and only if $i = SA[j]$. When stored in the external memory, the space of both arrays take $O(n/B)$ disk pages, and each entry can be reported in one I/O.

Suppose that we are given a pattern P, which appears at position i of T. That is, $T[i..i+|P|-1] = P$. Then, we observe that P is a prefix of the suffix $T[i..n]$. Furthermore, each other occurrence of P in T corresponds to a distinct suffix of T sharing P as a prefix. Based on this observation, the following lemma from [18] shows a nice property about the suffix array.

Lemma 1. *Suppose P is a pattern appearing in T. Then there exists $i \le j$ such that $SA[i]$, $SA[i+1], \ldots, SA[j]$ are the starting positions of all suffixes sharing P as a prefix. In other words, $SA[i..j]$ lists all occurrences of P in T.* □

2.2 Cache-Oblivious String Dictionaries

Recently, Brodal and Fagerberg proposed an external-memory index for a text T of length n that supports efficient pattern matching query [9]. Their index

takes $O(n/B)$ disk pages of storage; also, it does not require the knowledge of M or B to operate and is therefore cache-oblivious. For the pattern matching query, given any input pattern P, we can find all occurrences of P in T using $O((|P| + occ)/B + \log_B n)$ I/Os.

In this paper, we are interested in answering a slightly more general query. Given a pattern P, let ℓ be the length of the longest prefix of P that appears in T. We want to find all suffixes of T that has $P[1..\ell]$ as a prefix (that is, all suffixes of T whose common prefix with P is the longest among the others). We denote Q to be the set of starting positions of all such suffixes. Note that Q occupies a contiguous region in SA, say $SA[i..j]$. Now we define the *LCP query* of P with respect to T, denoted by $LCP(P, T)$, which reports (i) the *SA range*, $[i, j]$, that corresponds to the SA region occupied by Q, and (ii) the *LCP length*, ℓ.

With very minor adaptation, the index in [9] can readily be used to support efficient LCP query, as stated in the following lemma.

Lemma 2. *We can construct a cache-oblivious index for a text T of length n, such that given a pattern P, we can find $LCP(P, T)$, its SA range, and its length in $O(|P|/B + \log_B n)$ I/Os. The space of the index is $O(n/B)$ disk pages.* □

2.3 LCA Index on Rooted Tree

For any two nodes u and v in a rooted tree, a *common ancestor* of u and v is a node that appears in both the path from u to the root and the path from v to the root; among all common ancestors of u and v, the one that is closest to u and v is called the *lowest common ancestor* of u and v, denoted by $LCA(u, v)$. The lemma below states the performance of an external-memory index for LCA queries, which follows directly from the results in [16,5].

Lemma 3. *Given a rooted tree with n nodes, we can construct a cache-oblivious index of size $O(n/B)$ disk pages such that for any nodes u and v in the tree, $LCA(u, v)$ can be reported in $O(1)$ I/Os.* □

2.4 Cache-Oblivious Y-Fast Trie

Given a set X of x integers, the predecessor of r in X, denoted by $Pred(r, X)$, is the largest integer in X which is smaller than r. If the integers in X are chosen from $[1, n]$, the *Y-fast trie* on X [26] can find the predecessor of any input r in $O(\log \log n)$ time under the word RAM model; [3] the space occupancy is $O(x)$ words. In the external-memory setting, we can store the Y-fast trie easily using the van Emde Boas layout [7,22,23,20], giving the following lemma.

Lemma 4. *Given a set X of x integers chosen from $[1, n]$, we can construct a cache-oblivious Y-fast trie such that $Pred(r, X)$ for any integer r can be answered using $O(\log \log_B n)$ I/Os. The space of the Y-fast trie is $O(x/B)$ disk pages.* □

[3] A word RAM supports standard arithmetic and bitwise boolean operations on word-sized operands in $O(1)$ time.

2.5 Cache-Oblivious WLA Index

Let R be an edge-weighted rooted tree with n nodes, where the weight on each edge is an integer in $[1, W]$. We want to construct an index on R so that given any node u and any integer w, we can find the unique node v (if exists) with the following properties: (1) v is an ancestor u, (2) sum of weights on the edge from the root of R to v is at least w, and (3) no ancestor of v satisfies the above two properties. We call v the weighted level ancestor of u at depth w, and denote it by $WLA(u, w)$.

Assume that $\log W = O(\log n)$. In the internal-memory setting, we can construct an index that requires $O(n)$ words of space and finds $WLA(u, w)$ in $O(\log \log n)$ time [4]. In the following, we describe the result of a new WLA index that works cache-obliviously, which may be of independent interest. This result is based on a recursive structure with careful space management, whose proof is deferred to the full paper.

Lemma 5. *We can construct a cache-oblivious index on R such that for any node u and any integer w, $WLA(u, w)$ can be reported in $O(\log \log_B n)$ I/Os. The total space of the index is $O(n/B)$ disk pages.* □

2.6 Cache-Oblivious Index for Join Operation

Let T be a text of length n. For any two strings Q_1 and Q_2, suppose that $LCP(Q_1, T)$ and $LCP(Q_2, T)$ are known. The *join* operation for Q_1 and Q_2 is to compute $LCP(Q_1Q_2, T)$, where Q_1Q_2 is the concatenation of Q_1 and Q_2.

Cole et al. (Section 5 of [13]) had developed an index of $O(n \log n)$ words that performs the *join* operation in $O(\log \log n)$ time in the internal-memory setting. Their index assumes the internal-memory results of LCA index, Y-fast trie, and WLA index. In the following lemma, we give an index that supports efficient *join* operations in the cache-oblivious setting.

Lemma 6. *We can construct a cache-oblivious index on T of $O((n \log n)/B)$ disk pages and supports the join operation in $O(\log \log_B n)$ I/Os.*

Proof. Using Lemmas 3, 4, and 5, we can directly extend Cole et al.'s index into a cache-oblivious index. □

3 A Review of Cole et al.'s *k*-Error Matching

In this section, we review the internal-memory index for k-error matching proposed by Cole et al. [13], and explain the challenge in turning it into a cache-oblivious index.

To index a text T of length n, Cole et al.'s index includes two data structures: (1) the suffix tree of T that occupies $O(n)$ words, and (2) a special tree structure, called k-error tree, that occupies a total of $O(n \log^k n)$ words in space. The k-error tree connects a number of $(k-1)$-error trees, each of which in turn connects

to a number of $(k-2)$-error trees, and so on. The bottom of this recursive structure are 0-error trees.

Given a pattern P, Cole et al.'s matching algorithm considers different ways of making k edit operations on P in order to obtain an exact match in T. Intuitively, the matching algorithm first considers all possible locations of the leftmost error on P to obtain a match; then for each location i that has an error, we can focus on searching the remaining suffix, $P[i+1..|P|]$, for subsequent errors. The searches are efficiently supported by the recursive tree structure. More precisely, at the top level, the k-error tree will immediately identify all matches of P in T with no errors; in addition, for those matches of P with at least one error, the k-error tree classifies the different ways that the leftmost edit operation on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-1)$-error tree. Subsequently, each $(k-1)$-error tree being searched will immediately identify all matches of P with one error, while for those matches of P with at least two errors, the $(k-1)$-error tree further classifies the different ways that the second-leftmost edit operation on P into $O(\log n)$ groups, and then each group creates a search in a dedicated $(k-2)$-error tree. The process continues until we are searching a 0-error tree, in which all matches of P with exactly k errors are reported.

The classification step in each k'-error tree is cleverly done to avoid repeatedly accessing characters in P. It does so by means of a constant number of LCA, LCP, Pred, and WLA queries; then, we are able to create enough information (such as the starting position of the remaining suffix of P to be matched) to guide the subsequent $O(\log n)$ searches in the $(k'-1)$-error trees. Reporting matches in each error tree can also be done by a constant number of LCA, LCP, Pred, and WLA queries. In total, it can be done by $O(\log^k n)$ of these queries. See Figure 1 for the framework of Cole et al.'s algorithm.

Each LCA, Pred, or WLA query can be answered in $O(\log\log n)$ time. For the LCP queries, they are all in the form of $LCP(P_i, T)$, where P_i denotes the suffix $P[i..|P|]$. Instead of computing these values on demand, Cole et al. computes all these LCP values at the beginning of the algorithm. There are $|P|$ such LCP values, which can be computed in $O(|P|)$ time by exploiting the *suffix links* of the suffix tree of T (the algorithm is essentially McCreight's suffix tree construction algorithm [19]). Consequently, each LCP query is returned in $O(1)$ time when needed. Then, Cole et al.'s index supports k-error matching in $O(|P|+\log^k n \log\log n + occ)$ time, where occ denotes the number of occurrences.

3.1 Externalization of Cole et al.'s Index

One may now think of turning Cole et al.'s index directly into a cache-oblivious index, based on the existing techniques. While each LCA, Pred, or WLA query can be answered in $O(\log\log_B n)$ I/Os by storing suitable data structures (See Lemmas 3, 4, and 5), the bottleneck lies in answering the $O(\log^k n)$ LCP queries. In the external memory setting, though we can replace the suffix tree with Brodal and Fagerberg's cache-oblivious string dictionary (Lemma 2), we can no longer exploit the suffix links as efficiently as before. That means, if we compute $LCP(P_i, T)$ for all i in advance, we will need $\Omega(|P|^2/B)$ I/Os. Alternatively, if

```
Algorithm APPROXIMATE_MATCH(P)
Input: A pattern P
Output: All occurrences of k-error match of P in T
1.  R ← k-error tree of T;
2.  SEARCH_ERROR_TREE(P, R, nil);
3.  return;

Subroutine SEARCH_ERROR_TREE(P, R, I)
Input: A pattern P, an error tree R, information I to guide the search
    of P in R
1.  if R is a 0-error tree
2.     then Output all matches of P with k errors based on R and I;
3.          return;
4.     else (* R is a k'-error tree for some k' > 0 *)
5.          Output all matches of P with k − k' errors based on R
            and I;
6.          Classify potential error positions into O(log n) groups
            based on P, R, and I;
7.          for each group i
8.                Identify the (k' − 1)-error tree R_i corresponding
                  to group i;
9.                Compute information I_i to guide the search of P
                  in R_i;
10.               SEARCH_ERROR_TREE(P, R_i, I_i);
11. return;
```

Fig. 1. Cole et al.'s algorithm for k-error matching

we compute each LCP query on demand without doing anything at the beginning, we will need a total of $\Omega((\log^k n)|P|/B)$ I/Os to answer all LCP queries during the search process. In summary, a direct translation of Cole et al.'s index into an external memory index will need $\Omega((\min\{|P|^2, |P|\log^k n\} + occ)/B + \log^k n \log\log_B n)$ I/Os for k-error matching.

In the next section, we propose another approach, where we compute *some* useful LCP values using $O(|P|/B + k\log_B n)$ I/Os at the beginning, and each subsequent query of $LCP(P_i, T)$ can be answered efficiently in $O(\log\log_B n)$ I/Os (see Lemma 9 in Section 4). This result leads us to the following theorem.

Theorem 1. *For a fixed integer k, we can construct a cache-oblivious index on T of size $O((n\log^k n)/B)$ disk pages such that, given any pattern P, the k-error matches of P can be found in $O((|P| + occ)/B + \log^k n \log\log_B n)$ I/Os.* □

4 Cache-Oblivious k-Error Matching

Let P be a pattern, and let $P_i = P[i..|P|]$ be a suffix of P. In this section, our target is to perform some preprocessing on P in $O(|P|/B + k\log_B n)$ I/Os to

obtain some useful $LCP(P_i, T)$ values, such that subsequent query of $LCP(P_j, T)$ for any j can be answered in $O(\log \log_B n)$ I/Os.

We observe that for a general pattern P, the above target may be difficult to achieve. Instead, we take advantage by concerning only those P that potentially has k-error matches. We formulate a notion called *k-partitionable* and show that

- if P is k-partitionable, we can achieve the above target;
- if P is not k-partitionable, there must be no k-error match of P in T.

In Section 4.1, we first define the k-partitionable property, and describe an efficient *screening test* that checks whether P is k-partitionable; in case P is k-partitionable, the screening test would have computed $LCP(P_i, T)$ values for some i as a by-product. In Section 4.2, we show how to utilize these precomputed LCP values to answer $LCP(P_j, T)$ for any j in $O(\log \log_B n)$ I/Os.

In the following, we assume that we have maintained the suffix array and inverse suffix array of T. Each entry of these two arrays will be accessed one at a time, at the cost of one I/O per access.

4.1 k-Partitionable and Screening Test

Consider the following partitioning process on P. In Step 1, we delete the first ℓ characters of P where ℓ is the LCP length reported by $LCP(P, T)$. While P is not empty, Step 2 removes further the first character from P. Then, we repeatedly apply Step 1 and Step 2 until P is empty. In this way, P is partitioned into $\pi_1, c_1, \pi_2, c_2, \ldots, \pi_d, c_d, \pi_{d+1}$ such that π_i is a string that appears in T, and c_i is called a *cut-character* such that $\pi_i c_i$ is a string that does not appear in T. (Note that π_{d+1} is an empty string if P becomes empty after some Step 2.) Note that this partitioning is unique, and we call this the *greedy partitioning* of P.

Definition 1. *P is called k-partitionable if the greedy partitioning of P consists of at most k cut-characters.* □

The following lemma states that k-partitionable property is a necessary condition for the existence of k-error match.

Lemma 7. *If P has a k-error match, P is k-partitionable.* □

The *screening test* on P performs the greedy partitioning of P to check if P is k-partitionable. If not, we can immediately conclude that P does not have a k-error match in T. One way to perform the screening test is to apply Lemma 2 repeatedly, so that we discover π_1 and c_1 in $O(|P|/B + \log_B n)$ I/Os, then discover π_2 and c_2 in $O((|P| - |\pi_1| - 1)/B + \log_B n)$ I/Os, and so on. However, in the worst case, this procedure will require $O(k(|P|/B + \log_B n))$ I/Os. In the following lemma, we make a better use of Lemma 2 with the standard doubling technique and show how to use $O(|P|/B + k \log_B n)$ I/Os to determine whether P passes the screening test or not.

Lemma 8. *The screening test on P can be done cache-obliviously in $O(|P|/B + k \log_B n)$ I/Os.*

Proof. Let $r = \lceil |P|/k \rceil$. In Round 1, we perform the following steps.

- We apply Lemma 2 on $P[1..r]$ to see if it appears in T. If so, we double the value of r and check if $P[1..r]$ appears in T. The doubling continues until we obtain some $P[1..r]$ which does not appear in T, and in which case, we have also obtained π_1 and $LCP(\pi_1, T)$.
- Next, we remove the prefix π_1 from P. The first character of P will then become the cut-character c_1, and we apply Lemma 2 to get $LCP(c_1, T)$. After that, remove c_1 from P.

In each subsequent round, say Round i, we reset the value of r to be $\lceil |P|/k \rceil$, and apply the same steps to find π_i and c_i (as well as $LCP(\pi_i, T)$ and $LCP(c_i, T)$). The algorithm stops when P is empty, or when we get c_{k+1}.

It is easy to check that the above process correctly outputs the greedy partitioning of P (or, up to the cut-character c_{k+1} if P does not become empty) and thus checks if P is k-partitionable. The number of I/Os of the above process can be bounded as follows. Let a_i denote the number of times we apply Lemma 2 in Round i, and b_i denote the total number of characters compared in Round i. Then, the total I/O cost is at most $O((\sum_i b_i)/B + (\sum_i a_i) \log_B n)$ by Lemma 2. The term $\sum_i b_i$ is bounded by $O(|P| + k)$ because Round i compares $O(|\pi_i| + \lceil |P|/k \rceil)$ characters, and there are only $O(k)$ rounds. For a_i, it is bounded by $O(\log(k|\pi_i|/|P|) + 1)$, so that by Jensen's inequality, the term $\sum_i a_i$ is bounded by $O(k)$. □

4.2 Computing LCP for k-Partitionable Pattern

In case P is k-partitionable, the screening test in Section 4.1 would also have computed the answers for $LCP(\pi_i, T)$ and $LCP(c_i, T)$. To answer $LCP(P_j, T)$, we will make use of the *join* operation (Lemma 6) as follows. Firstly, we determine which π_i or c_i covers the jth position of P.[4] Then, there are two cases:

- **Case 1:** If the jth position of P is covered by π_i, we notice that the LCP length of $LCP(P_j, T)$ cannot be too long since $\pi_{i+1}c_{i+1}$ does not appear in T. Denote $\pi_i(j)$ to be the suffix of π_i that overlaps with P_j. Indeed, we have:

  Fact 1. $LCP(P_j, T) = LCP(\pi_i(j)c_i\pi_{i+1}, T)$.

  This shows that $LCP(P_j, T)$ can be found by the *join* operations in Lemma 6 repeatedly on $\pi_i(j)$, c_i and π_{i+1}. The SA range of $\pi_i(j)$ can be found easily using SA, SA^{-1} and WLA as follows. Let $[p, q]$ be the SA range of π_i. The pth smallest suffix is the string $T[SA[p]..n]$, which has π_i as a prefix. We can compute $p' = SA^{-1}[SA[p] + j]$, such that the p'th smallest suffix has $\pi_i(j)$ as a prefix. Using the WLA index, we can locate the node (or edge) in the suffix tree of T corresponding to $\pi_i(j)$. Then, we can retrieve the required SA range from this node. The LCP query on P_j can be answered in $O(\log \log_B n)$ I/Os.

[4] This is in fact a predecessor query and can be answered in $O(\log \log_B n)$ I/Os by maintaining a Y-fast trie for the starting positions of each π_i and c_i.

– **Case 2:** If c_i is the jth character of P, the LCP query on P_j can be answered by the *join* operation on c_i and π_{i+1} in $O(\log\log_B n)$ I/Os, using similar arguments as in Case 1.

Thus, we can conclude the section with the following lemma.

Lemma 9. *Let T be a text of length n, and k be a fixed integer. Given any pattern P, we can perform a screening test in $O(|P|/B + k\log_B n)$ I/Os such that if P does not pass the test, it implies there is no k-error match of P in T. In case P passes the test, $LCP(P[j..|P|], T)$ for any j can be returned in $O(\log\log_B n)$ I/Os.* □

5 $O(n \log n)$ Space Cache-Oblivious Index

To obtain an $O(n\log n)$-space index, we externalize Chan et al.'s internal-memory index, so that for patterns longer than $\log^{k+1} n$, they can be searched in $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$ I/Os. Roughly speaking, this index consists of a 'simplified' version of the index in Theorem 1, together with the range-searching index by Arge et al. [1] to answer the Tree-Cross-Product queries. To handle short patterns, we find that the internal-memory index of Lam et al. [17] can be used directly without modification, so that short patterns can be searched in $O(\log^{k(k+1)} n \log\log n + occ/B)$ I/Os.

Due to space limitation, we only state our result obtained by the above schemes. Details are deferred to the full paper.

Theorem 2. *For a constant $k \geq 2$, we can construct a cache-oblivious index on T of size $O(n\log n/B)$ pages such that on given any pattern P, the k-error matches of P can be found in $O((|P| + occ)/B + \log^{k(k+1)} n \log\log n)$ I/Os. For $k = 1$, searching takes $O((|P| + occ)/B + \log^3 n \log_B n)$ I/Os.* □

Acknowledgement

The authors would like to thank Gerth Stølting Brodal and Rolf Fagerberg for discussion on their results in [9], and the anonymous referees for their comments.

References

1. Arge, L., Brodal, G.S., Fagerberg, R., Laustsen, M.: Cache-Oblivious Planar Orthogonal Range Searching and Counting. In: Proc. of Annual Symposium on Computational Geometry, pp. 160–169 (2005)
2. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1988)
3. Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Indexing and Dictionary Matching with One Error. In: Proc. of Workshop on Algorithms and Data Structures, pp. 181–192 (1999)

4. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic Text and Static Pattern Matching. In: Proc. of Workshop on Algorithms and Data Structures, pp. 340–352 (2003)
5. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Proc. of Latin American Symposium on Theoretical Informatics, pp. 88–94 (2000)
6. Bender, M.A., Farach-Colton, M., Kuszmaul, B.C.: Cache-Oblivious String B-trees. In: Proc. of Principles of Database Systems, pp. 233–242 (2006)
7. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-Oblivious B-trees. In: Proc. of Foundations of Computer Science, pp. 399–409 (2000)
8. Brodal, G.S., Fagerberg, R.: Funnel Heap—A Cache Oblivious Priority Queue. In: Proc. of Int. Symposium on Algorithms and Computation, pp. 219–228 (2002)
9. Brodal, G.S., Fagerberg, R.: Cache-Oblivious String Dictionaries. In: Proc. of Symposium on Discrete Algorithms, pp. 581–590 (2006)
10. Buchsbaum, A.L., Goodrich, M.T., Westbrook, J.: Range Searching Over Tree Cross Products. In: Proc. of European Symposium on Algorithms, pp. 120–131 (2000)
11. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: A Linear Size Index for Approximate Pattern Matching. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 49–59 (2006)
12. Cobbs, A.: Fast Approximate Matching using Suffix Trees. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 41–54 (1995)
13. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary Matching and Indexing with Errors and Don't Cares. In: Proc. of Symposium on Theory of Computing, pp. 91–100 (2004)
14. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. JACM 46(2), 236–280 (1999)
15. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-Oblivious Algorithms. In: Proc. of Foundations of Computer Science, pp. 285–298 (1999)
16. Harel, D., Tarjan, R.: Fast Algorithms for Finding Nearest Common Ancestor. SIAM Journal on Computing 13, 338–355 (1984)
17. Lam, T.W., Sung, W.K., Wong, S.S.: Improved Approximate String Matching Using Compressed Suffix Data Structures. In: Proc. of International Symposium on Algorithms and Computation, pp. 339–348 (2005)
18. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
19. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. JACM 23(2), 262–272 (1976)
20. Prokop, H.: Cache-Oblivious Algorithms, Master's thesis, MIT (1999)
21. Ukkonen, E.: Approximate Matching Over Suffix Trees. In: Proc. of Symposium on Combinatorial Pattern Matching, pp. 228–242 (1993)
22. van Emde Boas, P.: Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. Information Processing Letters 6(3), 80–82 (1977)
23. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory 10, 99–127 (1977)
24. Vitter, J.S.: External Memory Algorithms and Data Structures: Dealing with Massive Data, 2007. Revision to the article that appeared in ACM Computing Surveys 33(2), 209–271 (2001)
25. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
26. Willard, D.E.: Log-Logarithmic Worst-Case Range Queries are Possible in Space$\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)