On Entropy-Compressed Text Indexing in External Memory*

Wing-Kai Hon[1], Rahul Shah[2], Sharma V. Thankachan[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, Louisiana State University, LA, US
{rahul,svt}@csc.lsu.edu
[3] Department of Computer Science, Texas A & M University, TX, USA
jsv@tamu.edu

Abstract. A new trend in the field of pattern matching is to design indexing data structures which take space very close to that required by the indexed text (in entropy-compressed form) and also simultaneously achieve good query performance. Two popular indexes, namely the FM-index [Ferragina and Manzini, 2005] and the CSA [Grossi and Vitter 2005], achieve this goal by exploiting the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994]. However, due to the intricate permutation structure of BWT, no locality of reference can be guaranteed when we perform pattern matching with these indexes. Chien *et al.* [2008] gave an alternative text index which is based on sparsifying the traditional suffix tree and maintaining an auxiliary 2-D range query structure. Given a text T of length n drawn from a σ-sized alphabet set, they achieved $O(n \log \sigma)$-bit index for T and showed that this index can preserve locality in pattern matching and hence is amenable to be used in external-memory settings. We improve upon this index and show how to apply entropy compression to reduce index space. Our index takes $O(n(H_k + 1)) + o(n \log \sigma)$ bits of space where H_k is the kth-order empirical entropy of the text. This is achieved by creating variable length blocks of text using arithmetic coding.

1 Introduction

Given a text T and a pattern P, finding all occurrences of P in T is the most fundamental problem in the field of pattern matching. In the data-structural sense, an index is built over T, and later some pattern P comes as a query; our target is to solve the above problem more quickly with the help of the index. Suffix trees [20,16] and suffix arrays [15] are the most popular indexes which can answer the query in $O(p + occ)$ time and $O(p + \log n + occ)$ time respectively, where $n = |T|$, $p = |P|$, and occ is the number of places where P occurs in T.

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082-MY3 (W. Hon) and US NSF Grant CCF–0621457 (R. Shah and J. S. Vitter).

J. Karlgren, J. Tarhio, and H. Hyyrö (Eds.): SPIRE 2009, LNCS 5721, pp. 75–89, 2009.

Historically, these two data structures are considered to consume "linear" space. However, the notion of space measure here was in terms of memory words. When measured in terms of bits, these indexes take $O(n \log n)$ bits which is asymptotically higher than the $n\lceil \log \sigma \rceil$ bits required to store the text in plain form; here, σ denotes the size of the common alphabet set Σ from which characters of T and P are drawn. Practically when we are indexing DNA texts (with $\sigma = 4$), these indexes are reported to take 15 to 50 times more space than the original data. Furthermore, the text T can often be compressed into nH_k bits by entropy-compression methods like gzip or bzip, where $H_k \leq \log \sigma$ denotes the kth-order empirical entropy of the text. Thus, the actual gap between the indexing space and the storage space is even larger.

A longstanding open question was to develop a text index which takes "truly" linear space. Grossi and Vitter [10] presented the first text index taking $O(n \log \sigma)$ bits. Simultaneously, Ferragina and Manzini [6] presented an index based on Burrows-Wheeler transform (BWT) [3] which took $O(nH_k)$ bits. Both indexing schemes were further refined [18,9,7] to take $nH_k + o(n \log \sigma)$ bits, and various space-time trade-offs are also obtained (see [17] for an excellent survey). One of the main approach in designing all these indexes is to permute the text according to the BWT. However, a short-coming of this approach is that BWT permutation completely shatters the locality of text characters. Each next character of the pattern being matched can occur at a random location in the BWT. Hence, no efficient external memory results were possible with such an approach. Chien *et al.* [4] took a different approach of *sparsifying* the suffix tree to achieve space reduction. The main idea was to combine a few contiguous characters from the text to create a block, where each block in turn is treated like a new alphabet symbol (or a meta-character). The index structures then includes the suffix tree of this blocked text as a component, which is effectively a miniature of the suffix tree of the original text but with fewer suffixes. This leads to an alternative $O(n \log \sigma)$-bit index when we set each block to contain roughly $d = 0.5 \log_\sigma n$ characters.

In this paper, we show the first entropy-compressed index in external memory which can effectively exploit locality in pattern matching. Our technique is to improve the blocking technique of Chien *et al.* [4]. We first introduce a variable-length blocking technique which is combined with arithmetic coding scheme. Using this we improve the space from $O(n \log \sigma)$ bits to $O(nH_k) + o(n \log \sigma)$ bits when $k = o(\log_\sigma n)$ and $\sigma = O(n^{1-\epsilon})$ for any fixed $\epsilon > 0$. We first present an index that works efficiently in the RAM model. Then, we show how to convert it to work in the external-memory model, and show that by maintaining an $O(n^\epsilon)$-bit table in RAM, pattern matching queries can be answered in $O((p \log n)/B + \log^3 n/(\log \sigma \log B) + occ \log_B n)$ I/Os; here, B denotes I/O block size in terms of memory words. This result is further improved to $O(p/(B \log_\sigma n) + \log^4 n/ \log \log n + occ \log_B n)$ I/Os by using $O(n)$-bit extra space.

On a related note, there were several attempts at designing compressed indexes in secondary memory based on LZ-indexes. In [2], Arroyuelo and Navarro

proposed an index whose space is $O(nH_k) + o(n \log \sigma)$, but the I/O bounds for pattern searching were not given. Their work is practical in nature and claims to answer pattern matching queries in about 20–60 disk accesses. In [8], González and Navarro provided an index which achieves $O(p + occ/B)$ I/Os for answering pattern matching query. However, their space usage is $O((n \log n) \times H_k \log(1/H_k))$ bits, which is an $O(\log n)$ factor more in terms of the optimal space complexity. Our techniques of blocking text and encoding blocks (meta-characters) using arithmetic coding are similar to the ones used in the above LZ-index line of work [2,8]. The key difference is in the way how the size of the blocks is controlled to achieve the desired theoretical bounds.

2 Preliminaries

This section introduces a few existing data structures for text indexing and orthogonal range searching which form the building blocks of our compressed text indexes. We will briefly explain their roles in our indexes, while a more detailed description is deferred in later sections. We also give a brief summary of the external-memory model of [1].

Throughout the paper, we use T to denote the text to be indexed, and $n = |T|$ to denote its length. We use P to denote the pattern which comes as an online pattern matching query, and $p = |P|$ to denote its length. Further, we assume the characters of T and P are both drawn from the same alphabet set Σ whose size is σ.

2.1 Suffix Trees, Suffix Arrays, and Burrows-Wheeler Transform

Suffix trees [20,16] and suffix arrays [15] are two well-known and popular text indexes that support online pattern matching queries in optimal (or nearly optimal) time. For text $T[1...n]$ to be indexed, each substring $T[i..n]$, with $i \in [1, n]$, is called a *suffix* of T. The suffix tree for T is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge e in the suffix tree is labeled by a series of characters, such that if we examine each root-to-leaf path, the concatenation of the edge labels along the path is exactly equal to the corresponding suffix represented by the leaf.

Suffix array $SA[1...n]$ is an array of length n, where $SA[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T. An important property of SA is that the starting positions of all suffixes with the same prefix are always stored in a contiguous region in SA. Based on this property, we define the *suffix range* of a pattern P in SA to be the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, $SA[j]$ is the starting point of a suffix of T with P as a prefix. Note that SA can be obtained by traversing the leaves of suffix tree in a left-to-right order, and outputting the starting position of each leaf (i.e., a suffix of T) along this traversal. In particular, we have the following technical lemma about suffix trees, suffix arrays, and suffix ranges.

Lemma 1. *Given a text T of length n, we can index T using suffix tree and suffix array in $\Theta(n \log n)$ bits such that the suffix range of any input pattern P can be obtained in $O(p)$ time.*

Suffix trees or suffix arrays maintain relevant information of all n suffixes of T such that on given any input pattern P, we can easily search for the occurrences of P *simultaneously* in each position of T. However, a major drawback is the blowup in space requirement, from the original $\Theta(n \log \sigma)$ bits of storing the text in plain form to the $\Theta(n \log n)$ bits of maintaining the indexes. In our compressed text indexes, we apply a natural and very simple idea to achieve space reduction, as suggested in [4] by maintaining only a fraction of these suffixes. The consequence is that we can no longer search all positions of T in a single pass. Instead, we need multiple passes, thus causing some inefficiency in the query time. On the other hand, we gain much space reduction by storing fewer suffixes.

The Burrows-Wheeler transform of a text T is an array BWT of characters such that $BWT[i]$ is the character preceding the ith lexicographically smallest suffix of T. That is, $BWT[i] = T[SA[i] - 1]$.

2.2 External-Memory Model

The external-memory model [1] or I/O model was introduced by Aggarwal and Vitter in 1988. In this model, the CPU is connected directly to an internal memory of size M, which is then connected to a much larger and slower disk. The disk is divided into blocks of B words (i.e., $B \log n$ bits). The CPU can only operate on data inside the internal memory. So, we need to transfer data between internal memory and disk through I/O operations, where each I/O may transfer a block from the disk to the memory (or vice versa). Since internal memory (RAM) is much faster, operations on data inside this memory are considered free. Performance of an algorithm in the external-memory model is measured by the number of I/O operations used.

2.3 String B-Tree

String B-tree (SBT) [5] is an index for a text T that supports efficient online pattern matching queries in the external-memory setting. Basically, it is a B-tree over the suffix array SA of T but with extra information stored in each B-tree node to facilitate the matching. The performance of SBT is summarized as follows.

Lemma 2. *Given a text T of length n characters , we can index T using a string B-tree in $\Theta(n/B)$ blocks or $\Theta(n \log n)$ bits such that the suffix range of any input pattern P of length p can be obtained in $O(p/(B \log_\sigma n) + \log_B n)$ I/Os.* □

In our compressed text index for the external-memory setting, we again achieve space reduction by maintaining fewer suffixes. Thus, our index includes a sparsified version of the SBT as the main component.

2.4 Orthogonal Range Searching in 2D Grid Using Wavelet Tree

In our compressed text index, in addition to the suffix trees or SBT, another key component is a data structure to represent some integer array $A[1...m]$, with each integer drawn from $[1, n]$, which can efficiently support online 4-sided queries of the following form:

Input: A position range $[\ell, r]$ and a value bound $[y, y']$
Output: All those z's in $[\ell, r]$ such that $y \leq A[z] \leq y'$

The above problem can easily be modeled as a geometric problem as follows. First, for each $i \in [1, m]$, generate a point $(i, A[i])$ in the 2-dimensional grid $[1, m] \times [1, n]$. This forms the representation of the array A. Then, for any input query with position range $[\ell, r]$ and value bound $[y, y']$, the desired output corresponds to all points in the grid that are lying inside the rectangle $[\ell, r] \times [y, y']$.

Such a query is called an *orthogonal range query* in the literature, and many indexing schemes are devised that have different tradeoffs between index space and query time. In our compressed text indexes, we will require an index for A which takes $O(m \log n)$ bits of space, so we select the wavelet tree [14,12,21] as our choice, whose results are summarized in the following lemma.

Lemma 3. *Given an integer array A of length m with values drawn from $[1, n]$, we can index A in $O(m \log n)$ bits such that the 4-sided query of any position range $[\ell, r]$ and any value bound $[y, y']$ can be answered in $O((occ + 1) \log n / \log \log n)$ time in the RAM model and $O((occ + 1) \log_B n)$ I/Os in the external-memory model.* □

3 The Framework of Our Indexing Scheme

This section first describes the general framework of our index design, which consists of a combination of the building block data structures mentioned in Section 2. Afterwards, we will look at the general approach to perform pattern matching based on our index. The following two sections details with the design and the analysis of the index performance.

3.1 The Framework of the Index Design

To obtain our compressed index, we perform the following three key steps:

Step 1: Given a text T, we first transform T into an equivalent text T' such that T' consists of at most $O((nH_k + o(n \log \sigma))/ \log n)$ meta-characters, where each meta-character represents at most d consecutive characters in the original text for some threshold d. In addition, we also require that each meta-character can be described in $O(\log n)$ bits, so that T' can be described in $O(nH_k) + o(n \log \sigma)$ bits.

Step 2: We maintain the suffix tree or String B-Tree for T', where we consider each meta-character of T' as a single character from a new alphabet.

Step 3: We perform the Burrows-Wheeler transform on T' to obtain an array A. Then we maintain the wavelet tree for A.

3.2 The Framework of the Pattern Matching Algorithm

The suffix tree or SBT in our index will maintain only the suffixes of T', which correspond to only a fraction of the original suffixes. Then, when a pattern P occurs in T, it will in general match the corresponding meta-characters of T' in the following way:

> The first part of P, say $P[1..i]$, matches the suffix of a meta-character $T'[j]$ and the remainder of P, say $P[i+1..p]$, matches the prefix of $T'[j+1..|T'|]$. We shall call such an occurrence of P an *offset-i occurrence* of P in T.

Our pattern matching algorithm is to find the offset-i occurrences of P separately for each relevant i. In our design, each meta-character of T' represents at most d original characters of T. It is therefore sufficient to consider only those i in $[0, d-1]$. This leads to the following pattern matching algorithm, which consists of two major steps:

Step 1: Compute the suffix range of $P[i+1..n]$ in the suffix array SA' of T' for each $i \in [0, d-1]$ using the suffix tree (ST') or String B-Tree (SBT') of T'.

Step 2: For each $i \in [0, d-1]$, use the suffix range of $P[i+1..n]$ to issue a 4-sided query in the wavelet tree of A to find all offset-i occurrences of P. (Details of how to issue the corresponding 4-sided query are given in the next section.)

4 Index for Internal Memory Model

In this section, we show a simple index based on variable length meta-character blocking and sparse suffix tree in the internal memory model. Later, in section 5, we shall show how to extend our results to the external memory model.

4.1 Index Design

In the index given by Chien *et al.* [4], the given text T is converted to an equivalent text T' by blocking every $d = 0.5 \log_\sigma n$ characters. Each block, called a meta-character, contains fixed number of characters. The transformed text T' consists of $O(n/\log_\sigma n)$ meta-characters. Hence, the suffix tree of T' takes $O(n \log \sigma)$ bits space.[1] The new index we propose in this paper improves the space complexity to $O(nH_k) + o(n \log \sigma)$ bits. Here, instead of having each meta-character contain a fixed number of characters, we allow a variable number of characters. Each meta-character is encoded in such a way that, its first k characters are written explicitly (using fixed length encoding) and the rest using kth-order arithmetic coding. The number of characters within a meta-character is restricted by the following two conditions.

[1] Assuming each integer and each pointer is at most $\log n$ bits long.

– The number of characters should not exceed a threshold $d = \log^2 n / \log \sigma$.
– After encoding, the total length should not exceed $0.5 \log n$ bits.[2]

In our new index, the transformation of T into T' can be performed as follows. Start encoding T from $T[1]$ and get its longest prefix $T[1...j]$, which satisfies the conditions of a meta-character. Hence, $T[1...j]$ in its encoded form is our first meta-character. After that the remainder of T is encoded recursively. (Note that the strings corresponding to distinct meta-characters are not required to be prefix-free.) The starting position of each meta-character is stored in an array M such that $M[i]$ corresponds to the starting position of ith meta-character in T. In other words, the substring $T[M(i)...(M[i+1]-1)]$ corresponds to the ith meta-character. For instance, $M[1] = 1$ and $M[2] = j + 1$. By concatenating all these meta-characters (in the order in which the corresponding block appears in T), we obtain the desired string T'.

Since each meta-character corresponds to a maximal substring of T without violating the two conditions, a meta-character corresponds either to (i) exactly d characters of T, or (ii) its encoding is just below $0.5 \log n$ in which case the encoding is of $\Theta(\log n)$ bits and corresponds to $\Theta(\log_\sigma n)$ characters of T.[3] Note that in both cases each meta-character corresponds to $\Omega(\log_\sigma n)$ characters.

Direct entropy compression of T would have resulted in $nH_k + o(n \log \sigma)$-bit space for T'. But in our scheme, the first k characters are written explicitly in each block. This results in an overhead of $O((n/\log_\sigma n) \times k \log \sigma) = o(n \log \sigma)$ bits to encode T', assuming $k = o(\log_\sigma n)$.[4] Thus, the number of meta-characters from (i) cannot exceed $n/d = o(n \log \sigma / \log n)$, while the number of meta-characters from (ii) is bounded by $O((nH_k + o(n \log \sigma))/\log n)$. In summary, the length of $T' = nH_k + o(n \log \sigma)$ bits, and there is a total of $O((nH_k + o(n \log \sigma))/\log n)$ meta-characters in T'.

By considering each meta-character as a single character from the new alphabet set, we construct the suffix tree ST' of T'. As the length of T' is given by $O((nH_k + o(n \log \sigma))/\log n)$, so is the number of nodes in ST'. Thus, ST' takes $O((nH_k + o(n \log \sigma))/\log n \times \log n) = O(nH_k) + o(n \log \sigma)$ bits of space.

Lemma 4. *The total number of distinct meta-characters is* $O(\sqrt{n})$.

Proof. Each meta-character has an encoding between 1 and $0.5 \log n$ bits. Thus, the number of distinct meta-character is at most $\sum_{r=1}^{0.5 \log n} 2^r = O(\sqrt{n})$. □

[2] Without loss of generality, we assume here that $\sigma < n^{1/4}$. The parameters can be appropriately adjusted for the more general case when $\sigma = O(n^{1-\epsilon})$ for any fixed $\epsilon > 0$.

[3] Here, we make a slight modification that one extra bit is spent for each meta-character, such that if our kth-order encoding of the next $o(\log_\sigma n)$ characters already exceeds $0.5 \log n$, we shall instead encode the next $0.5 \log_\sigma n$ characters (i.e., more characters) in its plain form. The extra bit is used to indicate whether we use the plain encoding or the kth-order encoding.

[4] As mentioned, there is also an extra bit overhead per meta-character; however, we will soon see that the number of meta-characters $= O((nH_k + o(n \log \sigma))/\log n)$ so that this overhead is negligible.

We also construct an auxiliary trie-structure Π which can be used to rank each of the meta-characters among all the meta-characters that are constructed from the text. Let B be a block in T which corresponds to a meta-character C in T', and let $\overleftarrow{B}$ denote the string obtained by reversing the characters of B. We maintain a string L which is the concatenation of all distinct $\overleftarrow{B}$'s in the uncompressed form and we construct a compact trie Π storing all distinct $\overleftarrow{B}$'s. The edges of Π are represented using two pointers, which are the starting and ending points of the corresponding substring in L. String L takes $O(\sqrt{n} \times (\log^2 n/\log\sigma) \times \log\sigma) = o(n)$ bits and Π takes $O(\sqrt{n} \times \log n) = o(n)$ bits of space.

Let $\Pi(i)$ represent the ith leftmost leaf of Π. Now we shall show how to obtain an array A from which we construct the wavelet tree. For this, we first compute BWT of T'. Let $BWT[i] = C$, where C is a meta-character and B is its corresponding character block. Now, search for $\overleftarrow{B}$ in Π and reach a leaf node $\Pi(j)$; then we set $A[i] = j$. That is, $A[i]$ is the leaf-rank of $\overleftarrow{B}$ in Π. Finally, we maintain a wavelet tree of A based on Lemmas 3 and 4, whose space takes $O((nH_k + o(n\log\sigma))/\log n) \times \log(O(\sqrt{n})) = O(nH_k) + o(n\log\sigma)$ bits. The total space requirement for our index is $O(nH_k) + o(n\log\sigma)$ bits.

4.2 Pattern Matching Algorithm

The suffix tree ST' maintains only the suffixes of T'. Therefore navigating through ST' can only report those occurrences of the query pattern P which start at a meta-character boundary. But in general, P can start anywhere inside T, where $P[1...i]$ matches to the suffix of a meta-character $T'[j]$ and the remaining of P, $P[i+1...p]$ matches the prefix of $T'[j+1...|T'|]$. We call such an occurrence of P an offset-i occurrence of P in T. We need to check for all possible offset occurrences. Since the number of characters inside a meta-character is at most d, it is sufficient to check for those offsets i where $i = 0, 1, 2, \ldots, d-1$.

To find offset-i occurrences, we let P_{pre} represent the prefix $P[1...i]$ and P_{suf} represent the suffix $P[i+1...p]$ of the pattern P. We first convert P_{suf} into P'_{suf} by blocking this into meta-characters. Following our convention, we use $\overleftarrow{P_{pre}}$ to denote the reverse of P_{pre}. Next, we search for $\overleftarrow{P_{pre}}$ in the compact trie Π to reach a position u^* (if exists); note that u^* may be an internal node, or within an edge, rather than a leaf. In any case, we use $\Pi(i_{left})$ and $\Pi(i_{right})$ to denote, the leftmost and rightmost leaves in the subtree of u^*.

We are now ready to show how to search for the desired offset-i occurrences of P:

1. Search for P'_{suf} in ST' and obtain its suffix range $SA'[\ell...r]$. Here P'_{suf} is of length at most $p\log\sigma$, hence by assuming standard word length of $O(\log n)$ bits, this matching step can be performed in $O(p/\log_\sigma n)$ time. But for matching an ending portion of a pattern, which may be smaller than the length of a meta-character, we need to perform a "predecessor search" in order to get the range. Therefore, in general the suffix range can be obtained in $O(p/\log_\sigma n + \log n)$ time.[5]

[5] More precisely, we maintain the SBT data structure for short patterns as suggested by Hon *et al.* [11] to accomplish the task. We defer the details in the full paper.

2. We need to find out those text positions in $SA'[\ell...r]$, such that P_{pre} occurs before those positions. This is equivalent to finding all z's in $[\ell, r]$, such that $i_{left} \leq A[z] \leq i_{right}$.
3. Now the search for offset-i occurrences is reduced to an orthogonal range searching problem in 2D grid. We use the wavelet tree structure of A to solve this query. According to Lemma 3, this will take $O((occ(i)+1) \log n / \log\log n)$ time, where $occ(i)$ represents the number of offset-i occurrences.

Lemma 5. *Based on ST' and the wavelet tree of A, all the offset-i occurrences of a pattern P in T, which cross at least one meta-character boundary, can be reported in $O(p/\log_\sigma n + \log n + occ(i) \log n / \log\log n)$ time, where $occ(i)$ is the number of offset-i occurrences of P in T.* □

The above steps need to be performed for all possible offsets i, where $i = 0, 1, \ldots, d-1$. For each offset i we need to convert P_{suf} into P'_{suf}. Assuming the conversion is done independently for each offset, it will in total take $O(p \log n + d \log n)$ time. This gives the following lemma.

Lemma 6. *A given text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits such that all the occurrences of a pattern P in T, which crosses at least one meta-character boundary in T, can be reported in $O(p \log n + \log^3 n / \log \sigma + occ \log n / \log\log n)$ time.* □

4.3 Index for Short Patterns

The methods described before will work only for those occurrences of a pattern that cross a meta-character boundary. To find those short patterns which start and end inside the same meta-character, we rely on an auxiliary data structure which is a generalized suffix tree Δ of all the *distinct* meta-characters that appear. Considering Lemma 4, the space for Δ can easily be bounded by $o(n)$.

The search begins by matching the pattern P in Δ to obtain the list L of all the distinct meta-characters in which P occurs (along with the relative positions of pattern occurrences inside a given meta-character. Now, on top of this, for each distinct meta-character C appearing in the text, we maintain the list H_C of all the positions in T' where the meta-character C occurs. These lists overall take $\log n$ bits per meta-character and hence the total space for the H structure is bounded by $O(nH_k) + o(n \log \sigma)$ bits. Once the list L of meta-characters (along with the internal positions) is obtained from Δ we use H as the de-referencing structure to obtain the final set of positions.

Lemma 7. *A given text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits such that all the occurrences of pattern P in T, which starts and ends inside the same meta-character in T, can be reported in $O(p + occ)$ time.* □

The following theorem concludes our result.

Theorem 1. *A text T can be indexed in $O(nH_k) + o(n \log \sigma)$ bits space, such that all the occurrences of a pattern P in T can be reported in $O(p \log n + \log^3 n / \log \sigma + occ \log n / \log\log n)$ time.* □

5 Extension to External Memory Model

In this section, we extend our results in the RAM model to the external memory model.[6] For this, we replace each data structure in internal memory model with its external memory counterpart. The sparse suffix tree ST' will be replaced by a sparse string B-tree SBT' of T'. The wavelet tree of array A will be replaced with its external memory version [12,14]. By performing a similar analysis, and setting the threshold d to be $\log^2 n/\log\sigma$, the searching for a pattern P in T will take $\sum_{i=0}^{d-1} O(p/(B\log_\sigma n) + \log_B n + occ(i)\log_B n) = O((p\log n)/B + (\log_B n)(\log^2 n/\log\sigma) + occ\log_B n)$ I/Os, where $occ(i)$ represents the number of offset-i occurrences that cross at least one meta-character boundary and occ represents the total number of such occurrences. The generalized suffix tree for short patterns will be replaced by string B-tree, which can perform pattern matching in $O(p/B + \log_B n + occ)$ I/Os. Immediately, we have the following theorem.

Theorem 2. *A text T can be indexed in $O(nH_k)+o(n\log\sigma)$ bits in the external memory, such that all occurrences of pattern P can be reported in $O((p\log n)/B+ \log^3 n/(\log\sigma\log B) + occ\log_B n)$ I/Os.*

Indeed, we can reduce the $O((p\log n)/B)$ term to $O(p/(B\log_\sigma n))$, if we allow slightly more index space. This is done by combining our index with Sadakane's Compressed Suffix Tree (CST) [19]. Our goal is to avoid repeated pattern matching for various offsets, which is done by using the "suffix link" functionality provided by CST. The main idea is that if some part of the pattern is matched during the offset-k search then we avoid re-matching it for offset-$(k+1)$ search and onwards; instead we rely on the suffix link to provide information for the subsequent search.

In the remainder of this section, we sketch how the pattern matching algorithm can be sped up by storing the CST. Firstly, for any internal node u inside the suffix tree, let $path(u)$ denote the string obtained by concatenation of edge labels from root to u. The *suffix link* of u is defined to be the (unique) internal node v such that the removal of the first character of $path(u)$ is exactly the same as $path(v)$. However, suffix link with respect to the original suffix tree may not exist in the sparse suffix tree or the sparse string B-tree (simply because some suffixes are missing).

In our algorithm, the full (non-sparse) suffix tree on T must be used, so that we can follow the original suffix links. To stay within our space bounds of $O(nH_k)$ we cannot afford to use the regular suffix tree. This explains why we choose the CST of [19], which provides *all* suffix tree functionalities in compressed space.

[6] Recall that the block size parameter B is measured in terms of memory words while the pattern length p is measured in terms of characters. Here, we further assume that the decoding table for arithmetic coding fits in the internal memory. By choosing appropriate parameters and with the condition that $k = o(\log_\sigma n)$, we can ensure that the decoding table size is $O(n^\epsilon)$ bits.

5.1 Compressed Suffix Tree

Let us assume we have stored Compressed Suffix Tree CST of the text T. In addition, all the nodes in CST which are also in the sparse suffix tree ST' are marked. For this marking, a bit-vector is maintained in addition to CST. The nodes in CST are considered in pre-order fashion and whenever a marked node is visited we write "1" or else we write "0". Thus, this bit-vector $\mathcal{B}$ stores marking information on the top of CST.

We shall need the following functionalities provided by the recent CST of [19] together with our bit-vector $\mathcal{B}$:

Suffix link: Given a node u (by its pre-order rank) in CST, return the suffix link node v (by its pre-order rank). This function can be done in $O(\log \sigma)$ I/Os.

Highest marked descendant: Given a node u in CST, its highest marked descendant is defined to be the node v such that v is in the subtree of u, v is marked, and no nodes between u and v is marked. Such a node v (if exists) is unique. This is due to the fact that the least common ancestor of two marked nodes (i.e., the least common ancestor of two sparse suffix tree nodes) is also marked. Note that this functionality is not directly provided by CST of [19] but can easily be implemented in $O(1)$ I/Os by storing a rank/select data structure over the bit-vector $\mathcal{B}$ along with the parentheses encoding of CST.

Lowest marked ancestor: Given a node u in CST, report its lowest marked ancestor (if exists). This can be done in $O(1)$ I/Os based on $\mathcal{B}$ and its the rank/select data structure.

Leftmost leaf: Given a node u in CST, locate its leftmost (rightmost) leaf node in its subtree. This can be done in $O(1)$ I/Os.

String-depth: Given a node u, report the length of $path(u)$. This can be done in $O(\log^2 n/\log\log n)$ I/Os.

Weighted level ancestor: Given a leaf ℓ and string-depth w, report the (unique) node u such that u is the first node on the path from root to ℓ with string-depth $\geq w$. This node u must be a lowest common ancestor between ℓ and some other leaf ℓ', so that we can find u if ℓ' is determined. Such ℓ' can be found by binary searching all leaves to the right of ℓ, and examine the string-depth of lowest common ancestor of ℓ and the leaf. The process can be done in $O(\log^3 n/\log\log n)$ I/Os.

5.2 Sparse String B-Tree

Our explanation below shall refer to both the sparse suffix tree and the sparse string B-tree. However, the sparse suffix tree is never stored and is just for the sake of notation and the identification of nodes. Firstly, the following two functionalities of the sparse string-B tree SBT' will be used. The I/O complexity for both functions follows directly from the searching strategy of SBT in the original paper [5].

1. Given a pattern P, let $lcp(P, ST')$ be the length of the longest common prefix of P with any suffix stored in SBT': we can use $O(lcp(P, ST')/B + \log_B n)$ I/Os to find the node u (by its pre-order ranking in the suffix tree ST') such that u is the node with smallest string-depth in ST' and $lcp(P, ST') = lcp(P, path(u))$.
2. If we are given a node u in ST' such that the pattern P is guaranteed to match up to some length x on $path(u)$, then the above lcp search can be done in $O((lcp(P, ST') - x)/B + \log_B n)$ I/Os.

5.3 Pattern Matching Algorithm

Now, we are ready to show how we match a pattern P in this combination of sparse string B-tree and CST. First we start with finding offset-0 occurrences, then we find offset-1 occurrences, then offset-2 occurrences and so on. Let P_i denote the pattern P with the first i characters deleted. Thus we have to match $P_0, P_1, P_2, \ldots, P_{d-1}$ in the string B-tree. Corresponding to each offset i we find the range $[\ell_i, r_i]$ in the sparse string B-tree.

We start matching the pattern $P = P_0$ in SBT'; this allows us to find the node u in ST', such that u is the closest node from root such that $lcp(path(u), P) = lcp(P, ST')$. If the pattern is matched entirely, then we call this offset a success and output its range. In this case we set $lcp = p$, and also obtain the range $[\ell_0, r_0]$. If not, we set $lcp = lcp(P, ST')$ and follow the "suffix link". Let's first define the notion of suffix link in the sparse suffix tree ST' (or SBT').

Definition 1. *Given the pair (u, lcp), let pair (v, lcp') be such that (1) $lcp' = lcp - t$, (2) $path(u)[t+1..lcp] = path(v)[1..lcp']$ and (3) t is the smallest integer ≥ 1 for which such a node v exists in ST'. If more than one v exists in ST', we set v to be the highest node among them. Then (v, lcp') as is called t-suffix link of (u, lcp).*

Now, we show how to compute t-suffix link for pair (u, lcp) in $O(t \log^3 n / \log\log n)$ I/Os. This is done by using the suffix link functionality provided by CST. First, we use the pre-order rank of u to find the corresponding node in CST. Then, inside CST, we can find u's ancestor y such that string-depth of y is just more than lcp. This can be done by the weighted level ancestor query in $O(\log^3 n / \log\log n)$ I/Os. The node y represents the location where P stops in the CST if P were matched with the CST instead. To proceed for the next offset, we follow the suffix link from y and reach node w (and increment t by 1). Now, we first find the lowest marked ancestor m of w in $O(1)$ I/Os and check if its string-depth is at least $lcp - t$. If so, we come back to its corresponding node v in ST' and set $lcp' = lcp - t$. Note that (v, lcp') is the desired t-suffix link of (u, lcp), so that we can proceed with the pattern matching in SBT'.[7] Otherwise, if m does not exist or its string-depth is too small, we find in the subtree of w and try the highest marked descendant m' of w in $O(1)$ I/Os. If m' exists, we come

[7] Note that when we switch back to a node in SBT', we choose the top-most node in SBT' corresponding to the node v.

back to its corresponding node v' in ST' and set $lcp' = lcp - t$, while it follows that (v', lcp') is the desired t-suffix link of (u, lcp) so that we can again proceed with the pattern matching in SBT'. If there is no such marked descendant m', we follow further the suffix link from w (and increment t), and keep following suffix links until we reach either a node m or m' using the above procedure. In this case, we can be sure that none of the offsets between 1 and $t-1$ would produce any results. Consequently the corresponding (v, lcp') or (v', lcp') will be the desired t-suffix link and we can directly jump to offset-t match. This procedure gives us all the ranges $[\ell_i, r_i]$ for all the possible offsets (up to at most d of them).

5.4 Analysis

The space taken by both CST and string B-tree is $O(nH_k + n) + o(n \log \sigma)$ bits. For matching the pattern P, there are d phases. In each phase, we match some *distinct* part of P and then spend $O(\log^3 n / \log\log n)$ I/Os in CST plus an extra $O(\log_B n)$ I/Os (apart from matching characters of P) in SBT'. Thus, in total, we spend $O(d \log^3 n / \log\log n)$ in addition to the I/O in which the pattern is matched with the actual text inside the SBT'. On the other hand, since the characters of P are accessed once and are accessed sequentially, the total I/Os for matching characters of P can be bounded by $O(p/(B \log_\sigma n) + d \log_B n)$. For the conversion of the characters in P into the corresponding meta-characters, we assume that it is done in RAM so that it does not incur additional I/Os. Overall, this gives us $O(p/(B \log_\sigma n) + d \log^3 n / \log\log n)$ I/Os for finding out all the ranges $[\ell_0, r_0], [\ell_1, r_1], ..., [\ell_{d-1}, r_{d-1}]$.

Once these ranges are ready, we can use the external memory wavelet tree to find out the actual occurrences (which cross a meta-character boundary). The short patterns are handled as before using the generalized suffix tree approach (except we are using a SBT instead). Since the space of CST is $O(nH_k + n)$ bits which is the bottleneck, we may reduce the blocking factor to be $d = 0.5 \log n$ (thus having the effect of more meta-characters in T' but faster query) without affecting the space. The following theorem captures our new result.

Theorem 3. *A text T can be indexed in $O(nH_k + n) + o(n \log \sigma)$ bits in external memory, such that all occurrences of a pattern P in T can be reported in $O(p/(B \log_\sigma n) + \log^4 n / \log\log n + occ \log_B n)$ I/Os.*

6 Conclusion

We show the first entropy compressed text index in external memory. Our index is based on the paradigm of using sampled suffixes [13], and achieves locality while matching pattern which was lacking in other BWT based indexes. The main idea here is to partition the text into variable length block according to their compressibility and then compress each block using arithmetic coding. We show how this idea can be combined with the notion of suffix links by using CST of Sadakane[19].

We achieve optimal query I/O performance with respect to the length p of the input query pattern, taking $O(p/(B \log_\sigma n))$ I/Os. As noted by Chien *et al.* [4], the lower bounds in range searching data structures suggest that the last term $O(occ \log_B n)$ cannot be improved to $O(occ/B)$. But, it may be possible to improve the middle term of polylog(n). Another possible improvement could be in reducing space term from $O(nH_k)$ to strictly nH_k.

Acknowledgments

We would like to thank the anonymous reviewers for their careful reading and constructive comments, and for pointing out a potential flaw in the paper. We would also like to express our gratitude to Kunihiko Sadakane for clarifying the functionalities of the CST in his recent paper [19].

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM 31(9), 1116–1127 (1998)
2. Arroyuelo, D., Navarro, G.: A Lempel-Ziv Text Index on Secondary Storage. In: Proceedings of Symposium on Combinatorial Pattern Matching, pp. 83–94 (2007)
3. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA (1994)
4. Chien, Y.-F., Hon, W.-K., Shah, R., Vitter, J.S.: Geometric Burrows-Wheeler Transform: Linking Range Searching and Text Indexing. In: Proceedings of Data Compression Conference, pp. 252–261 (2008)
5. Ferragina, P., Grossi, R.: The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. Journal of the ACM 46(2), 236–280 (1999)
6. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2005); A preliminary version appears in FOCS 2000
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed Representations of Sequences and Full-Text Indexes. ACM Transactions on Algorithms 3(2) (2007)
8. González, R., Navarro, G.: A Compressed Text Index on Secondary Memory. In: Proceedings of IWOCA, pp. 80–91 (2007)
9. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: Proceedings of Symposium on Discrete Algorithms, pp. 841–850 (2003)
10. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005); A preliminary version appears in STOC 2000
11. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed Index for Dictionary Matching. In: Proceedings of Data Compression Conference, pp. 23–32 (2008)
12. Hon, W.K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Technical Report TR-06-008, Department of CS, Purdue University (2006)
13. Kärkkäinen, J., Ukkonen, E.: Sparse Suffix Trees. In: Cai, J.-Y., Wong, C.K. (eds.) COCOON 1996. LNCS, vol. 1090, pp. 219–230. Springer, Heidelberg (1996)

14. Mäkinen, V., Navarro, G.: Position-Restricted Substring Searching. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 703–714. Springer, Heidelberg (2006)
15. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing 22(5), 935–948 (1993)
16. McCreight, E.M.: A Space-economical Suffix Tree Construction Algorithm. Journal of the ACM 23(2), 262–272 (1976)
17. Navarro, G., Mäkinen, V.: Compressed Full-Text Indexes. ACM Computing Surveys 39(1) (2007)
18. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2), 294–313 (2003); A preliminary version appears in ISAAC 2000
19. Sadakane, K.: Compressed Suffix Trees with Full Functionality. Theory of Computing Systems, 589–607 (2007)
20. Weiner, P.: Linear Pattern Matching Algorithms. In: Proceedings of Symposium on Switching and Automata Theory, pp. 1–11 (1973)
21. Yu, C.C., Hon, W.K., Wang, B.F.: Efficient Data Structures for Orthogonal Range Successor Problem. In: Ngo, H.Q. (ed.) COCOON 2009. LNCS, vol. 5609, pp. 97–106. Springer, Heidelberg (2009)