Efficient Maximal Repeat Finding Using the Burrows-Wheeler Transform and Wavelet Tree

M. Oğuzhan Külekci, Jeffrey Scott Vitter, *Fellow, IEEE,* and Bojian Xu*, *Member, IEEE*

Abstract—Finding repetitive structures in genomes and proteins is important to understand their biological functions. Many data compressors for modern genomic sequences rely heavily on finding repeats in the sequences. Small-scale and local repetitive structures are better understood than large and complex interspersed ones. The notion of maximal repeats captures all the repeats in the data in a space-efficient way. Prior work on maximal repeat finding used either a suffix tree or a suffix array along with other auxiliary data structures. Their space usage is 19–50 times the text size with the best engineering efforts, prohibiting their usability on massive data such as the whole human genome. We focus on finding all the maximal repeats from massive texts in a time- and space-efficient manner. Our technique uses the Burrows-Wheeler Transform and wavelet trees. For data sets consisting of natural language texts and protein data, the space usage of our method is no more than three times the text size. For genomic sequences stored using one byte per base, the space usage of our method is less than double the sequence size. Our space-efficient method keeps the timing performance fast. In fact, our method is orders of magnitude faster than the prior methods for processing massive texts such as the whole human genome, since the prior methods must use external memory. For the first time, our method enables a desktop computer with 8GB internal memory (actual internal memory usage is less than 6GB) to find all the maximal repeats in the whole human genome in less than 17 hours. We have implemented our method as general-purpose open-source software for public use.

Index Terms—repeats, maximal repeats, Burrows-Wheeler Transform, wavelet trees.

1 Introduction

FINDING repetitive structures in genomes and proteins is important in understanding their biological functions [12]. One of the well-known features of DNA is its repetitive structures, especially in the genomes of eukaryotes. Examples are that overall about one-third of the whole human genome consists of repeated subsequences [29]; about 10–25% of all known proteins have some form of repetitive structures [24]. In addition, a number of significant problems in molecular sequence analysis can be reduced to repeat finding [28]. It is of great interests for biologists to find such repeats in order to understand their biological functions and solve other problems. Another motivation for finding repeats is to compress the DNA sequences, which is known as one of the most challenging tasks in the data compression field. DNA sequences consist only of symbols from {ACGT} and therefore can be represented by two bits per character. Standard compressors such as `gzip` and `bzip` usually use more than two bits per character and therefore cannot reach good compression. Many modern genomic sequence data compression techniques highly rely on the repeat finding in the sequences [27], [2]. For all these purposes, repeat finding is the first step which is critical and needs to be conducted efficiently.

There can be many repeats of various lengths in a text consuming much space for their storage. We need a notion that captures all the repeats in a space-efficient way, which is served by *maximal repeats* [12]. Maximal repeats are repeats whose extensions occur fewer times than the maximal repeats in the text. However, current techniques for finding maximal repeats are either based on suffix trees [12] or suffix arrays [1], both requiring enormous space usage caused by the large space cost of the suffix trees or suffix arrays and their auxiliary data structures. In fact, their space usage is 19–50 times the text size with the best engineering efforts. Such enormous space requirements limit the usage of the current techniques, making them only useful for texts of no more than hundreds of millions characters and prohibiting their usage in the setting of billions of characters such as the whole human genome, unless expensive supercomputers with large internal memory are used.

The field of compressed data structures and compressed full-text indexing [31] involves the design of indexes that support fast full-text pattern matching using limited amount of space. In particular, the goal is to have an index whose size is roughly equal to the size of the text in the compressed format, with search performance comparable to the one achieved by uncompressed methods such as suffix trees and suffix arrays. Many of the compressed indexing techniques such as the compressed suffix array [11], [10] and FM-index [8], [7] developed in the last decade make use of *wavelet trees* [10] and the text's Ψ decomposition [11] or the the *Burrows-Wheeler transform* (BWT) [4].

Since the suffix array-based method for maximal repeat

- *M. O. Külekci is with the National Research Institute of Electronics and Cryptology, 41470, Gebze, Kocaeli, Turkey. E-mail: kulekci@uekae.tubitak.gov.tr*
- *J. S. Vitter and B. Xu are with the Department of Electrical Engineering & Computer Science of the University of Kansas, KS 66045, USA. E-mail: {jsv, bojianxu}@ku.edu*
- **Corresponding author*

- *A preliminary version [17] of this work appeared in the proceedings of IEEE International Conference on Bioinformatics & Biomedicine (BIBM), December 18-21, Hong Kong, China. Part of this work was done while all the authors were with Texas A&M University.*

finding [1] also uses other large auxiliary data structures (inverse suffix array, longest common prefix (LCP) array, and an non-decreasing permutation of the LCP array), we cannot directly get a smaller-space solution by simply replacing the suffix array with a compressed suffix array.

Compressed suffix tree (CST) is also not a practical indexing technique for maximal repeat finding in massive data like the whole human genome if we only use normal desktop computers. The main problem of using CST is the expensive space and time cost of the CST construction. For the whole human genome which has about three billion bases, *"the construction takes about four days, the final index (CST) occupies about 8.5GB and the peak memory usage is 24GB"* [39]. It is worth noting that this 4-day construction time happens with a 32GB-memory machine [39]. The construction will take even much longer if only 8GB memory (as in our setting) is available. Note that we have not counted the time cost for the maximal repeat finding process yet.

Our method uses the Burrow-Wheeler Transform (BWT) [4] and wavelet trees [10] with provable time and space efficiency and good usability in practice. Overall, for data sets consisting of natural language text and protein data, our method uses space no more than three times the text size. For genomic sequences stored using one byte per base, the space usage of our method is less than double the sequence size. Our space-efficient method also keeps the timing performance fast. In fact, our method is orders of magnitude faster than the prior methods for processing massive texts such as the whole human genome when the internal memory capacity is limited, because the prior methods have to use the external memory [40]. To the best of our knowledge, this is the first work that enables a desktop computer with 8GB internal memory (actual internal memory usage is less than 6GB) to find all the maximal repeats of the whole human genome in less than 17 hours.

1.1 Problem

Let $T = T[1...n] = t_1 t_2 \dots t_n$ be a text of size n, where each character t_i, $1 \leq i \leq n-1$, is drawn from a finite ordered alphabet Σ of size σ, while $t_n = \$$ is a special character that does not appear in Σ and is lexicographically smaller than any character in Σ. Character t_n is used only to ease the text processing. Our goal is to find all the *maximal repeats* of T.

Definition 1 (Maximal Repeat, Section 7.12 of [12]): A *maximal repeat* of text T is a subtext of T that occurs in T at least twice such that any extension of the subtext occurs in T fewer times.

Example 1: The maximal repeats of $T =$ `mississippi$` are {`i(4)`, `p(2)`, `s(4)`, `issi(2)`}. In the parenthesis are the numbers of occurrences of the corresponding repeats in T. {`$(1)`, `m(1)`} are not maximal repeats because each of them occurs only once in T. {`is(2)`, `si(2)`, `ss(2)`, `ssi(2)`} are not maximal repeats because their extension `issi` also occurs twice in T. Note that maximal repeats can be nested (`s` and `issi`) and overlapped (`i` and `issi`).

Reporting all the repeats of a text requires enormous space to store the output. Maximal repeats efficiently capture all the repeats of a text. The number of maximal repeats of a given text is bounded by the text size.

Fact 1 ([12]): There can be at most n maximal repeats in any text of size n.

1.2 Our Contribution

The main contribution of this work is that we provide an option for people to use normal computers with limited internal memory to find maximal repeats in massive text data such as the whole human genome within an acceptable amount of time. We designed, analyzed, and implemented an algorithm to serve this purpose without any assumption on the alphabet size. Our algorithm not only has provable time and space efficiency but also has been empirically shown to work very well in practice as we find that, for the first time, it enables a normal computer with 8GB internal memory (the actual internal memory usage is less than 6GB) to find all the maximal repeats in the whole human genome, which consists of about three billion bases, in less than 17 hours. The best prior work, which is suffix array based, needs workspace at least 19 times the text size and becomes orders of magnitude slower than our method when it cannot fit into the internal memory. We fully implemented our algorithm as a general-purpose open-source software for public use.

1.3 Comparison with Related Work

In this section, we survey the suffix tree- and the suffix array-based methods for the maximal repeat finding, as well as other methods and systems that deal with text repeat finding.

The state-of-the-art method for maximal repeat finding uses suffix array[1]. It first finds the candidate maximal repeats with the aid of the longest common prefix (LCP) array, and then verifies whether each candidate maximal repeat can be extended to the right and/or to the left. Those inextensible candidates are actual maximal repeats. The algorithm also uses the inverse suffix array, the LCP array and a permutation of the non-decreasing LCP array, each of which is of the suffix array size. If we use fixed 32-bit data type to store integers from $[1, n]$, which is necessary and being used in the method of [1] for processing large data sets like the human genome sequences, the total space cost becomes at least $16n$ bytes—16 times the text size, not yet including the storage space for the text itself and other auxiliary data structures. Our experimental study shows that the actual space cost of their algorithm is more than 19 times the text size.

The suffix tree-based method (Section7.12.1, [12]) uses even more space than the suffix array-based method does—the suffix tree alone consumes space at least 20 times in the worst case and 10.1 times on the average the text size with the best engineering efforts [18]. The REPuter tool [20],

[19] is an example system using the suffix tree, where the input size is limited to the RAM size divided by 45 in the worst case.

Other methods for repeat finding include [22], which however only finds fixed k-mers. The toolkits by [33] only searches for n-grams, while we find all the maximal repeats. Similarly, the method for frequent pattern mining in [9] can output all the repeats of a string by an appropriate setting for parameter and input, but it is not known how their method can be customized to output maximal repeats. We refer readers to the recent survey by [36], [37] for methods that are based on heuristics [21] and search for specific biological identifications [3], [5].

Paper organization. In Section 2, we introduce the notations used in the entire paper and prepare the building blocks for our maximal repeat finding algorithm. We provide an overview of our algorithm in Section 3 and describe its details in Section 4. Implementation details and experimental results are given in Section 5. The paper is concluded with future work in Section 6 .

2 Notations and Building Blocks

Recall $T = t_1 t_2 \dots t_n$, where $t_i \in \Sigma$ for $1 \le i \le n-1$ and $t_n = \$$. Without loss of generality, we assume $\Sigma[1] < \Sigma[2] < \cdots < \Sigma[\sigma]$. Let $T[i \dots j]$ denote the subtext $t_i t_{i+1} \dots t_j$. For each $i \in [1, n]$, $T[1 \dots i]$ is T's *prefix* of size i, and $T[i \dots n]$ is T's *suffix* of size $n - i + 1$ *locating* at text position i. For any $i \neq j$, $T[i \dots n]$ is lexicographically smaller than $T[j \dots n]$, iff: (1) $T[i] < T[j]$ or (2) $T[i] = T[j]$ and $T[i+1 \dots n]$ is lexicographically smaller than $T[j+1 \dots n]$. Because of the special character \$, all the suffixes of T can be unambiguously sorted in a lexicographic order. The *suffix array* $SA[1 \dots n]$ of T is a permutation of $[1, n]$, such that $T[SA[1] \dots n], \dots, T[SA[n] \dots n]$ are in lexicographically ascending order. For any $SA[i] = j$, we call i the *suffix array index* and j the *suffix array pointer* of the suffix $T[j \dots n]$. The *inverse suffix array* $SA^{-1}[1 \dots n]$ of T is a permutation of $[1, n]$, such that $SA^{-1}[i] = j$ iff $SA[j] = i$. For ease of presentation, we assume $SA[i] - 1 = n$ if $SA[i] = 1$ and $SA[j] + 1 = 1$ if $SA[j] = n$.

Definition 2 ([7]): $LF(i) = SA^{-1}\big[SA[i]-1\big], i \in [1, n]$.

Definition 3 ([11]): $\Phi(i) = SA^{-1}\big[SA[i]+1\big], i \in [1, n]$.

$LF(i)$ (resp. $\Phi(i)$) returns the suffix array index of the suffix that locates right before (resp. after) $T\big[SA[i] \dots n\big]$ in the text T.

Definition 4: $C[i] = \big|\{c \in T \mid c \le \Sigma[i]\}\big|, 1 \le i \le \sigma$.

Let $\{1/\epsilon, 2/\epsilon, \dots, \epsilon n/\epsilon = n\}$ be ϵn sampled text positions, where $0 < \epsilon < 1$. For ease of presentation, we assume ϵn is an integer and i/ϵ for all $i \in [1, \epsilon n]$ are also integers; otherwise ceiling or flooring can be used to round the numbers into integers. The bit array $\mathcal{B}$ marks T's suffixes whose text locations are sampled.

Definition 5: $\mathcal{B} = \mathcal{B}[1 \dots n]$: $\mathcal{B}[i] = 1$ iff there exists some $j \in \{1/\epsilon, 2/\epsilon, \dots, n\}$, such that $SA[i] = j$; otherwise, $\mathcal{B}[i] = 0$.

The integer array $\mathcal{I}$ stores the sampled text positions in the sorted order of the suffixes that locate at those sampled text positions.

Definition 6: $\mathcal{I} = \mathcal{I}[1 \dots \epsilon n]$, such that $\mathcal{I}[i] = j$ iff there exists some $k \in [1, n]$ such that $\mathcal{B}[k] = 1$ and $Rank_1(\mathcal{B}, k) = i$ and $SA[k] = j$.

2.1 Succinct Bit Array Indexing

Succinct bit array is often a key component in designing compressed data structures. It is also used in our method for maximal repeat finding.

Lemma 1 ([34]): Any arbitrary bit array $B[1 \dots n]$ can be represented in $nH_0(B) + o(n)$ bits, where $H_0(B)$ is the 0-order empirical entropy of B, so that for any i and b, $1 \le i \le n$, $b \in \{0, 1\}$, the following queries can be answered in constant time.

- $Member(B, i)$: $B[i]$.
- $Rank_b(B, i)$: the number of bit b in $B[1 \dots i]$.
- $Select_b(B, i)$: the smallest $j \in [1, n]$ such that $Rank_b(B, j) = i$, if j exists; otherwise, return null.

The succinct representation of B can be constructed in $O(n)$ time using $O(n)$ bits of space.

2.2 Wavelet Trees

Wavelet tree [10] is an elegant data structure for coding sequences of characters from a multicharacter alphabet. It extends the support for member, rank and select queries from bit arrays to general multicharacter texts.

Lemma 2 ([10]): The wavelet tree of a text of size n drawn from an alphabet Σ of size σ uses $nH_0(T) + O(n \log\log n / \log_\sigma n)$ bits of space, where $H_0(T)$ is the 0-order empirical entropy of T. For any $i \in [1, n]$ and $c \in \Sigma$, the wavelet tree can answer the following queries in $O(\log \sigma)$ time:

- $Member(T, i)$: $T[i]$.
- $Rank_c(T, i)$: the number of character c in $T[1 \dots i]$.
- $Select_c(T, i)$: the smallest $j \in [1, n]$, such that $Rank_c(T, j) = i$, if j exists; otherwise, return null.

The wavelet tree can be constructed in $O(n \log \sigma)$ time using $O(n \log \sigma)$ bits of space.

2.3 Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) [4] of a text T, denoted as T_{bwt}, is a permutation of T, such that $T_{\text{bwt}}[i] = T\big[SA[i] - 1\big]$. Recall that we denote $SA[i] - 1 = n$, if $SA[i] = 1$. T_{bwt} can be viewed as the array of characters, each of which precedes each of the sorted suffixes of T. Space-efficient BWT construction directly from the text already exist [15], [23], [13], [14], [30], [32].

2.4 Succinct Computation of $LF(\cdot)$, $\Phi(\cdot)$, and $SA[\cdot]$

Lemma 3 ([11], [7]): (1) Given C, Σ, and the wavelet tree of T_{bwt}, the succinct representation of $\mathcal{B}$ can be constructed using $O(n \log \sigma)$ time and $O\big(nH_0(T) + \sigma \log n\big)$ bits of space; (2) Given C, Σ, and the wavelet tree of

Algorithm 1: Find Maximal Repeats–a High-level View

Input: T, T_{bwt}, LCP of T, SA of T
Output: The text and locations of T's maximal repeats, each of which has length at least ml and occurs at least mo times.

```
for m ← ml ... σ' do   /* σ' is #distinct values in LCP */
    Compute R_m                                  /* Definition 7 */
    for j ← m_1 ... m_{k_m} do
        if r_j − l_j + 1 < mo then Continue
        if T_bwt[l_j ... r_j] are NOT identical then
            Output T[SA[l_j] ... SA[l_j] + m]        /* Repeat */
            for k ← l_j ... r_j do                   /* Text locations */
                Output SA[k]
            end
        end
    end
end
```

T_{bwt}, and the succinct representation of $\mathcal{B}$, the integer array $\mathcal{I}$ can be constructed using $O(n \log \sigma)$ time and $O\big(nH_0(T)+(\sigma+\epsilon n)\log n\big)$ bits of space; (3) Given C, Σ, and the wavelet tree of T_{bwt}, for any $i \in [1,n]$, we can compute $LF(i)$ and $\Phi(i)$ using space of $O\big(\sigma \log n + nH_0(T)\big)$ bits and $O(\log \sigma)$ time; (4) Given C, Σ, the wavelet tree of T_{bwt}, $\mathcal{I}$, and the succinct representation of $\mathcal{B}$, for any $i \in [1,n]$, we can compute $SA[i]$ in $O\big((1/\epsilon)\log \sigma\big)$ time using $O\big(nH_0(T) + (\sigma+\epsilon n)\log n\big)$ bits of space.

2.5 Succinct Longest Common Prefix Array

The *longest common prefix* (LCP) array of text T, denoted as $LCP[1 \ldots n]$, stores the lengths of the longest common prefix of every two neighboring suffixes that are in the lexicographical order: $LCP[i] = \max\big\{t \geq 0 \mid \forall j \in [0,t], T\big[SA[i-1]+j\big] = T\big[SA[i]+j\big]\big\}$, if $i > 1$; we define $LCP[1] = 0$. The LCP array can be succinctly stored in a $2n$-bit array [35], which we call *succinct LCP* array (SLCP).

Lemma 4: Given T, C, Σ, the wavelet tree of T_{bwt}, $SA^{-1}[1]$, $\mathcal{B}$, and $\mathcal{I}$, algorithm 3 constructs the $2n$-bit SLCP using $O\big(n \log \sigma + (\sigma+\epsilon n)\log n\big)$ bits of space and $O\big((1/\epsilon) n \log \sigma\big)$ time.

Lemma 5: Given C, Σ, the wavelet tree of T_{bwt}, $\mathcal{B}$, $\mathcal{I}$, and the succinct representation of the $2n$-bit SLCP bit array, for any $i \in [1,n]$, we can retrieve $LCP[i]$ using $O\big(n \log \sigma + (\sigma+\epsilon n)\log n\big)$ bits of space and $O\big((1/\epsilon)\log \sigma\big)$ time.

We refer the reader to Appendix A for our space-efficient method for the SLCP array construction and the proofs for Lemma 4 and 5.

3 A High-level View of Our Method

By the definition of maximal repeat in Section 1.1, we know the length of maximal repeats ranges from 1 to $n-1$. Our strategy is to find all the maximal repeats in the order of their lengths from the shortest to the longest. For a particular maximal repeat length, we first find a set of candidate maximal repeats of that particular length, and then find the actual maximal repeats from the candidate set.

In particular, for a given repeat length, we first find a set of largest suffix array intervals such that, for each suffix array interval, the length of the longest common prefix of the suffixes in that suffix array interval is equal to the given length. Those longest common prefixes are candidate maximal repeats, which only need to be verified whether they can be extended to the left. The verification for the left extension of one particular candidate maximal repeats can be done by checking whether the characters preceding the multiple copies of that particular candidate maximal repeat are identical.

Definition 7: For a given integer m, $1 \leq m \leq n-1$, the *suffix array intervals of candidate maximal repeats* of size m is a sequence of non-overlapped suffix array intervals $R_m = \big\langle [l_{m_1}, r_{m_1}], [l_{m_2}, r_{m_2}], \ldots, [l_{m_{k_m}}, r_{m_{k_m}}] \big\rangle$, for some integer k_m, such that for all $i \in [1, k_m]$:

- $l_{m_i} = \min\{j \in [r_{m_{i-1}}+1, n-1] \mid LCP[j+1] \geq m\}$ $(r_{m_0} \equiv 0)$
- $r_{m_i} = \max\{j \in [l_{m_i}+1, n] \mid LCP[\alpha] \geq m, \forall \alpha \in [l_{m_i}+1, j]\}$
- $\min\big\{LCP[j] \mid j \in [l_{m_i}+1, r_{m_i}]\big\} = m$
- if $r_{m_{k_m}} < n$, then for all $j \in [r_{m_{k_m}}+1, n]$, $LCP[j] < m$

Intuitively, R_m is the set of largest suffix array intervals, such that for each suffix array interval in R_m, the length of the longest common prefix of the suffixes belonging to that suffix array interval is exactly m. Note that R_m can be empty.

For any $m \in [1, n-1]$, if $R_m \neq \emptyset$, let P_{m_i}, $1 \leq i \leq k_m$, denote the longest common prefix of the suffixes in the suffix array interval $[l_{m_i}, r_{m_i}] \in R_m$. By the definition of R_m, we know that for each $i \in [1, k_m]$, P_{m_i} is a candidate maximal repeat and $|P_{m_i}| = m$. For a particular m, the number of candidate maximal repeats is no more than n, because $m_{k_m} \leq n$. The next lemma shows that P_{m_i} is a maximal repeat if its left extension occurs fewer times than P_{m_i} does.

Lemma 6: For any $m \in [1, n-1]$ such that $R_m \neq \emptyset$ and any $i \in [1, k_m]$, if the symbols in $T_{\text{bwt}}[l_{m_i} \ldots r_{m_i}]$ are not the same, then P_{m_i} is a maximal repeat of size m in T. (Proof in Appendix B.)

Lemma 7: Any maximal repeat must occur as P_{m_i} for some $m \in [1, n-1]$ and some $i \in [1, k_m]$. (Proof in Appendix B.)

Therefore, we can find the maximal repeats of T by finding the P_{m_i} for all $m \in [1, n-1]$ and all $i \in [1, k_m]$ where $R_m \neq \emptyset$. Then for each P_{m_i}, we can verify whether its one-character left extension occurs fewer times in T than P_{m_i} using T_{bwt}. This idea serves as the basis of our algorithm for maximal repeat finding. Algorithm 1 gives a high-level description of our algorithm.

It is necessary to note that the high-level idea of our method is similar to the one in [1], but our algorithm uses the notion of suffix array interval which helps avoid the checking of the right extension of the candidate maximal repeats, and therefore simplifies the algorithm. Our algorithm also uses the BWT of the text to verify the left extension of the candidate maximal repeat instead of the complicated judging condition in [1], making our algorithm easier to understand.

Algorithm 2: Find Maximal Repeats–Final Algorithm

Input: $\mathcal{W}_{\text{lcp}}$, B_{lcp}, B_{bwt}, $\mathcal{W}$, T, $\mathcal{B}$, $\mathcal{I}$, ml, mo
/* $\mathcal{W}_{\text{lcp}}$, B_{lcp} and B_{bwt} are defined in Section 4. $\mathcal{W}$ is wavelet tree of T_{bwt}. */
Output: Maximal repeats of T and their text locations. Each returned maximal repeat has length at least ml and occurs at least mo times in T.

1 **for** $i \leftarrow 1 \dots \sigma'$ **do** /* σ' is #distinct values in LCP */
2 **for** $j \leftarrow 1 \dots v_i$ **do** /* v_i is #occurrences of i in LCP */
3 $pos_{i,j} \leftarrow Select_i(LCP, j)$ /* $\mathcal{W}_{\text{lcp}}$'s $Select()$ query */
4 **if** $i < ml$ **then** $\{B_{\text{lcp}}[pos_{i,j}] \leftarrow 1$; continue$\}$
5 $l \leftarrow \max\{k \mid k < pos_{i,j}$ and $B_{\text{lcp}}[k] = 1\}$
6 $r \leftarrow \min\{k \mid k > pos_{i,j}$ and $B_{\text{lcp}}[k] = 1\} - 1$
7 $B_{\text{lcp}}[pos_{i,j}] \leftarrow 1$
8 **if** $r - l + 1 < mo$ **then** continue
9 **if** $l > 0$ *and* $Member(LCP, l) = i$ **then** continue
/* $[l, r]$:SA interval of a candidate max repeat */
10 **if** $(Rank_1(B_{\text{bwt}}, r) - Rank_1(B_{\text{bwt}}, l) > 0)$ **then**
/* Use Lemma 3 to compute $SA[\cdot]$ */
11 Output $T[SA[l] \dots SA[l] + i]$ /* Repeat */
12 **for** $k \leftarrow l \dots r$ **do** Output $SA[k]$ /* Text locations */
13 **end**
14 **end**
15 **end**

	C/Java	Proteins	English	dblp xml	Ch. 1
MAX_{lcp}	71,651	25,822	109,394	1,005	67,631
AVG_{lcp}	168	166	2,221	42	38
$H_0(LCP)$	6.34	5.05	7.73	6.71	3.94

TABLE 1
The maximum value, average value and the 0-order empirical entropy of the LCP array of some example texts. Ch. 1 is the first human chromosome with all the masked 'N' characters removed and is obtained from ftp://ftp.ncbi.nlm.nih.gov. Other texts are obtained from the *Pizza&Chili Corpus* and each has 50MB characters.

4 Final Algorithm

Now we reduce the space cost in the high-level idea in order to enable normal computers to find maximal repeats in massive data such as the whole human genome, while still maintain a good search performance. In particular, we can use any existing space-efficient Burrows-Wheeler Transform construction algorithms [15], [23], [13], [14], [30], [32] to construct the BWT of the text. Then we create and use the wavelet tree of the BWT as the input for Algorithm 3 (in the Appendix) to compute the $2n$-bit SLCP. We then create and use another wavelet tree built over the LCP array to retrieve the repeat lengths from the shortest one to the longest one and find the corresponding candidate maximal repeats. The space cost for the wavelet tree of the LCP array is only the entropy size of the LCP array, which is usually very small due to the skewness in the LCP array values (see the statistics of some example texts in Table 1). Candidate maximal repeats can be further verified by checking if the BWT entries that precede the multiple copies of a candidate repeat are identical. This can be efficiently done via succinct bit array rank queries. Our method is so space-efficient that it can find the maximal repeats of the whole human genome using less than $6GB$ memory in less than one day. Before we proceed to our final algorithm, we prepare the following data structures.

1) $\mathcal{W}_{\text{lcp}}$ is the wavelet tree of the LCP array. Using the SLCP bit array, $\mathcal{W}_{\text{lcp}}$ can be constructed in $O\big((1/\epsilon)n \log \sigma + n \log \sigma'\big)$ time, where $\sigma' \leq n - 1$ is the number of distinct values in the LCP array. Retrieving all the LCP values from SLCP over the course of the wavelet tree construction takes $O((1/\epsilon)n \log \sigma)$ time (Lemma 5). The construction of $\mathcal{W}_{\text{lcp}}$ takes another $O(n \log \sigma')$ time (Lemma 2), so the total time cost is $O\big((1/\epsilon)n \log \sigma + n \log \sigma'\big)$. The space cost of constructing $\mathcal{W}_{\text{lcp}}$ is $O(n \log \sigma')$ (Lemma 2).
2) $B_{\text{lcp}}[0 \dots n + 1]$ is a bit array of size $n + 2$. B_{lcp} is initialized as all 0 except $B_{\text{lcp}}[0]$ and $B_{\text{lcp}}[n + 1]$. Those positions with 0-bits will be turned on one by one by our algorithm for a purpose that will be clarified later. By using a $2n$-bit binary bit tree structure designed in [1], which can be constructed in $O(n)$ time, given an integer $i \in [1, n]$ such that $B_{\text{lcp}}[i] = 0$, we can get $\max\{k \mid k < i$ and $B_{\text{lcp}}[k] = 1\}$ and $\min\{k \mid k > i$ and $B_{\text{lcp}}[k] = 1\}$ and turn on $B_{\text{lcp}}[i]$ in $O(\log n)$ time.
3) $B_{\text{bwt}}[1 \dots n]$ is a bit array of size n. $B_{\text{bwt}}[i] = 1$ iff $i = 1$ or $T_{\text{bwt}}[i] \neq T_{\text{bwt}}[i - 1]$, so that for any $1 \leq j < k \leq n$, all the characters in $T_{\text{bwt}}[j \dots k]$ are the same iff $B_{\text{bwt}}[j + 1 \dots k]$ are all 0-bits. Clearly, B_{bwt} can be constructed in $O(n)$ time.

Algorithm 2 shows the pseudocode of our maximal repeat finding algorithm. We traverse the lengths of the maximal repeat from the shortest to the longest by traversing all the LCP values from the smallest to the largest using the space-saving data structure $\mathcal{W}_{\text{lcp}}$. For each particular repeat length, we find the corresponding suffix array intervals of candidate maximal repeats. Note that each wavelet tree node represents a distinct character in the alphabet (a distinct value in the LCP array here). So the ith leftmost leaf node of $\mathcal{W}_{\text{lcp}}$ represents the LCP value i. Let v_i denote the number of occurrences of i in the LCP array. For $j = 1 \dots v_i$, let $pos_{i,j}$ denote the position of the jth leftmost occurrence of the LCP value i in the LCP array. Each $LCP[pos_{i,j}]$ can be retrieved via $Select$ operation on $\mathcal{W}_{\text{lcp}}$ using $O(\log \sigma')$ time—steps 1–3. We ignore all the LCP array values that are smaller than the user input minimum repeat length threshold—step 4. Otherwise, we find a suffix array interval $[l, r]$ at steps 5–6 using the bit array B_{lcp}. Because we traverse all the LCP values from the smallest to the largest and all of $LCP[l + 1 \dots r]$ have not been traversed yet, we know $LCP[k] \geq i$ for all $k \in [l + 1, r]$ and $\min\{LCP[k] \mid l + 1 \leq k \leq r\} = i$. Since the number of occurrences of the longest common prefix of the suffixes in the suffix interval $[l, r]$ is $r - l + 1$, we ignore the suffix array interval $[l, r]$ if $r - l + 1$ is smaller than the user input minimum threshold of the number of the occurrences of the repeats—step 8. If $LCP[l] = i$ (step 9), meaning that the longest common prefix of the suffixes belonging to $[l - 1, r]$ is also i, then $[l, r]$ is not

a suffix array interval of a candidate maximal repeat of size i. Any suffix array interval $[l, r]$ of candidate maximal repeats of size i will be detected by the algorithm when $pos_{i,j} = \min\{k \in [l+1, r] \mid LCP[k] = i\}$ is traversed. Steps 10–13 verifies whether the candidate maximal repeat can be extended to the left by using the B_{bwt} bit array and report the maximal repeats.

Theorem 1: Given a text T of size n drawn from an alphabet of size σ, Algorithm 2 finds the maximal repeats of T that have size of at least ml characters and occur at least mo times, using $O\big(n\log\sigma + (\sigma+\epsilon n)\log n + n\log\sigma'\big)$ bits of space and $O\big(n\log n + (1/\epsilon)n\log\sigma\big)$ time. Reporting the text of a particular maximal repeat P of size p takes additional time of $O\big((1/\epsilon)\log\sigma + p\big)$. Reporting the text locations of P takes additional time of $O\big(occ\cdot(1/\epsilon)\log\sigma\big)$, where occ is the number of occurrences of P in T. (Proof in Appendix B.)

Comments: It's worth noting that the space cost in the above theorem is the peak space usage over the course of the construction of the relevant data structures, not the size of the resulting data structures. Recall that ϵ (defined in Section 2.4) is a user-input parameter which determines the percentage of the sampled text positions. By reasonably setting $\epsilon = 1/32$ which is often smaller than or comparable to $1/\log n$ even for large texts, the space usage in Theorem 1 becomes $O\big(n\log(\sigma\sigma')\big)$, where σ' is the number of distinct values in the LCP array and is often a small number. The resulting data structures used for the maximal repeat finding uses space of $O\big(n(\log\sigma + H_0')\big)$ bits, where H_0', the 0-order empirical entropy of the LCP array, is often much smaller than $\log\sigma'$ due to the skewness in the LCP array values (Table 1). The time complexity of our method $O\big(n\log n + (1/\epsilon)n\log\sigma\big) = O(n\log n)$ matches the time complexity of the state-of-the-art SA-based method [1].

5 Implementation and Experiments

We fully implemented our algorithm in C++[1]. We implemented all the parts of the algorithm except the BWT construction. We use Kärkkäinen's C++ code [15] and Lippert *et al.*'s C code [23] to build BWTs for non-genomic and genomic texts, respectively. Our implementation is full and generic in that it supports maximal repeat finding in texts of any alphabet size and is ready for public use.

Experimental set-up and environment. We used g++ 4.4.1 to build the executables of all the source code in our experiments. The experiments were conducted on a Dell Vostro 430 with a 2.8GHz four-core Intel@CoreTM i7-860 chip with 8MB L3 Cache, but no parallelism was used. The machine runs 64-bit Ubuntu 9.10 operating system and has 8GB internal memory, 24GB swap space, and one 1TB Serial ATA Hard Drive (7200RPM) with DataBurst CacheTM. We used the following real-world biological and nonbiological data to test the efficiency and usability of our method:

1. Source code can be downloaded at: http://www.ittc.ku.edu/~bojianxu/publications/findmaxrep.zip

	Text Size	SU1	SU1 / TS	SU2	SU2 / TS
Ch. 1	215.47	4,100	19.03	360	1.67
Ch. 1–2	442.64	8,618	19.47	683	1.54
Ch. 1–3	628.41	12,232	19.46	986	1.57
Ch. 1–4	808.31	15,732	19.46	1,271	1.57
Ch. 1–5	977.77	19,030	19.46	1,524	1.56
Ch. 1–8	1,448.48	≈ 28,245	≈ 19.50	2,232	1.54
W.H.G.	2,759.57	≈ 53,811	≈ 19.50	5,494	1.99
Prot. 1	100	1,906	19.06	280	2.80
Prot. 2	200	3,806	19.03	549	2.75
Prot. 3	400	7,789	19.47	1,180	2.77
Prot. 4	600	11,680	19.47	1,674	2.79
Prot. 5	650	12,652	19.46	1,807	2.78
Prot. 6	800	≈ 15,600	≈ 19.50	2,235	2.79
Prot. 7	1,000	≈ 19,500	≈ 19.50	2,768	2.77
Eng. 1	100	1,952	19.52	290	2.90
Eng. 2	200	3,898	19.49	568	2.84
Eng. 3	400	7,788	19.47	1,124	2.81
Eng. 4	600	11,680	19.47	1,682	2.80
Eng. 5	800	15,571	19.46	2,238	2.80
Eng. 6	1,500	≈ 29,250	≈ 19.50	4,198	2.80
Eng. 7	2,000	≈ 39,000	≈ 19.50	5,594	2.80

TABLE 2
Space usage comparison between the SA-based method [1] and our method. Space size is measured in megabytes. Genomic data are stored using one byte per base. SU1 = space usage of the SA-based method; SU2 = space usage of our method; TS = text size. For the Chromosome 1–8, the whole human genome, Protein 6 and 7, and English 6 and 7, the SA-base method did not terminate in ten days and the space usages are estimates.

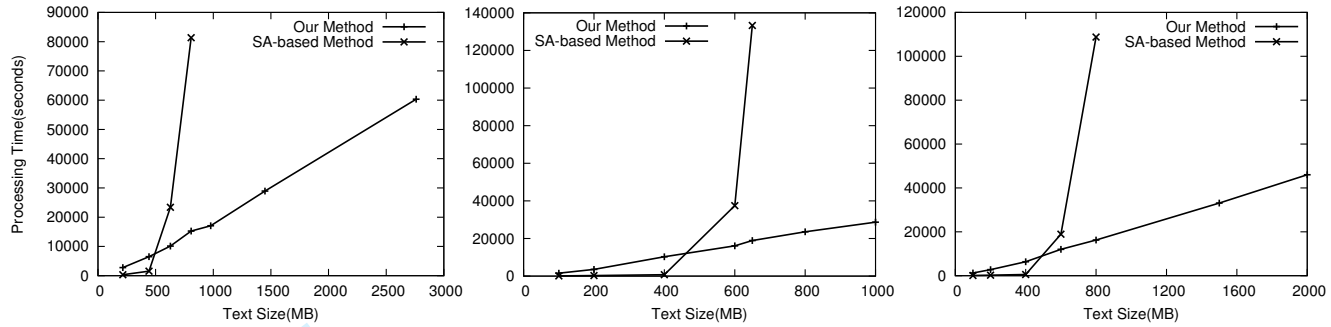
1. The human genome sequences from NCBI[2]. We removed all the masked `'N'` symbols, so the sequences only contain symbols from {`ACGT`}.

2. Protein data from the Pizza&Chili Corpus[3].

3. English texts from the Wikipedia dump. English 1–5 are from the dump on 2006–03–03 [4]; English 6–7 are from the dump on 2010–07–30 [5].

We set the user input parameter ml, the minimum threshold for repeat size, to be the nearest whole number of $\log_2 n$. This is a reasonable setting, because repeats of smaller sizes are usually meaningless as they can even occur in a randomly generated text as long as the text size is of the order of the power of the repeat size. We set the user input parameter $mo = 2$, the minimum threshold for frequencies of repeats. Thus, all the maximal repeats whose sizes are larger than or equal to ml will be reported. We set the parameter $\epsilon = 1/32$ for our algorithm. All experiments output the maximal repeats onto local hard disk files, including the text of the repeats and their frequencies and text locations. We used the system time to measure the programs' time cost. We used the `VmPeak` entry in the `/proc/<pid>/status` file created by the OS to measure the space cost, which is the peak of the total amount of virtual memory used by the program, including

2. ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/Assembled_chromosomes
3. http://pizzachili.dcc.uchile.cl/texts.html
4. http://cs.fit.edu/~mmahoney/compression/enwik9.zip
5. The original data file was from: http://download.wikimedia.org/enwiki/20100730/enwiki-20100730-pages-articles.xml.bz2, which is now temporary unavailable due to the hardware problem at Wikipedia. We put a copy of the data at our own machine at: http://faculty.cse.tamu.edu/bojianxu/enwiki-20100730-pages-articles.xml.gz



(a) Human genome (b) Protein (c) English

	Text size (MB)	Construction time for SA	SA-based total time	SA construction time percentage	Our method's total time
Ch. 1	215.47	250	329	76.00%	2,784
Ch. 1–2	442.64	624	1,543	40.44%	6,486
Ch. 1–3	628.41	1,162	23,370	4.97%	10,119
Ch. 1–4	808.31	1,657	81,345	2.04%	15,258
Ch. 1–5	977.77	18,446	490,016	3.76%	17,069
Ch. 1–8	1,448.48	n/a	> 864,000	n/a	28,945
W.H.G.	2,759.57	n/a	> 864,000	n/a	60,344
Prot. 1	100	97	135	71.82%	1,545
Prot. 2	200	211	309	68.17%	3,582
Prot. 3	400	489	767	63.75%	10,284
Prot. 4	600	825	37,491	2.20%	16,080
Prot. 5	650	900	133,247	0.68%	18,924
Prot. 6	800	n/a	> 864,000	n/a	23,591
Prot. 7	1,000	n/a	> 864,000	n/a	28,685
Eng. 1	100	81	100	80.43%	1,242
Eng. 2	200	185	226	81.76%	2,783
Eng. 3	400	423	521	81.16%	6,362
Eng. 4	600	713	18,903	3.77%	12,041
Eng. 5	800	1,027	108,746	0.94%	16,270
Eng. 6	1,500	n/a	> 864,000	n/a	33,104
Eng. 7	2,000	n/a	> 864,000	n/a	46,009

Fig. 1. Timing performance comparison between the SA-based method and our method. Time is measured in seconds. Genomic data are stored using one byte per base. The SA-based method becomes unacceptably slow when the input size becomes larger than 600MB, since the program's workspace becomes larger than 11GB (Table 2), exceeding the 8GB internal memory capacity. The timing of the SA-based method for Chromosome 1-5 is too large and is not shown on the curve in order to get clear plots of other points on the curve (please see the data in the table). Data for the SA-based method regarding other larger data sets are not available as the program did not terminate in ten days. The SA construction time percentage shows that when its workspace exceeds the internal memory capacity, the performance bottleneck of the SA-based method is the maximal repeat finding process after the SA is constructed.

code, data, and shared libraries plus the pages that have been swapped out.

Main observations. We compared the performance of our algorithm with the performance of the state-of-the-art suffix array-based algorithm [1]. The experimental study led to the following main observations:

1. The SA-based method consistently consumes more than 19 times the text size for all types of texts, while our method uses space less than three times the text size for the English texts and protein data, and no more than double the text size for the human genomic sequences stored using one byte per base. Our method can therefore fit into a normal computer with 6GB internal memory to find the maximal repeats of the whole human genome (Table 2).

2. When its input size exceeds 600MB and thus its workspace becomes larger than 11GB (Table 2), exceeding the 8GB internal memory limit, the SA-based method becomes unacceptably slow because of the page faults and swapping (Figure 1). The SA-based method spent so long in processing Chromosome 1–5 ($490,016$ seconds ≈ 5.7 days) that data are not shown in Figure 1(a) in order to get clear plots for other points on the curves. The SA-based method even did not terminate in ten days in the processing of Chromosome 1–8, the whole human genome, protein 6–7, and English 6–7.

3. When its workspace exceeds the internal memory capacity, the SA-based method's performance bottleneck is not the SA construction but the maximal repeat finding process after the SA construction (the table in Figure 1). For example, for the Chromosome 1–3, the SA construction takes 1,162 seconds, which is about 4.97% of the total $23,370$ seconds. Similar results regarding other data sets can be found in the table of Figure 1. The SA construction time is negligible when the input size is large, meaning that using external memory-efficient SA construction algorithm [6] cannot significantly improve the performance of the SA-based method.

4. Our method can find maximal repeats in massive texts using a normal computer with a time cost orders of magnitude less than the time cost of the SA-based method. In particular, our method can find all the maximal repeats of the whole human genome, which is about three billion bases (ACGT), using a normal computer with 8GB internal memory (actual internal memory used is less than 6GB, Table 2) in less than 17 hours (Figure 1).

6 Conclusion and Future Work

This paper allows finding maximal repeats from massive text data using normal computers. To the best of our knowledge, this is the first work that enables a normal desktop with 8GB internal memory (actual memory usage is only 6GB) to quickly find the maximal repeats from the whole human genome. We fully implemented our algorithm as a generic tool for public use.

Our method trades processing time for space savings. The LCP array computation is one of the performance bottlenecks. Finding a faster LCP array construction algorithm using the BWT and wavelet trees can significantly improve the timing performance of our method. Also, our method (as well as the suffix tree and suffix array-based methods) does not support online queries—the algorithm returns all the maximal repeats that satisfy the user-given parameters in one run and then quits. It would be more useful in practice if the algorithm can run as a service. That is, the algorithm runs continuously and can receive and answer queries for maximal repeats of interest online. Another interesting improvement is to parallelize our method to take the full advantages of multicore processors.

APPENDIX A
SLCP ARRAY CONSTRUCTION

Recently there have been a few space-efficient LCP array construction algorithms using normal data structures such as the suffix array [26] or compressed data structures such as the compressed suffix array [38]. Our space-efficient SLCP construction uses the BWT and wavelet tree. We start from the following fact from [16].

Fact 2 ([16]): If $LCP\big[SA^{-1}[i]\big] = h > 0$, then $LCP\big[SA^{-1}[i+1]\big] \geq h-1$, for any $i \in [1, n-1]$.

Linear-time LCP construction. Based on Fact 2, an elegant linear-time algorithm for constructing the LCP array in the order of text positions was proposed by [16]. Their algorithm takes $O(n)$ time in the worst case but requires space of $(3n\log n + n\log\sigma)$ bits for storing the LCP array, the suffix array, the inverse suffix array, and the text. The space cost was later improved by [25] and further reduced by [26] to be $(2n\log n + n\log\sigma)$ bits by reusing the inverse suffix array to store the LCP array.

Succinct representation of LCP. From Fact 2, it is observed in [35] that $LCP\big[SA^{-1}[1]\big] + 2\cdot 1 < LCP\big[SA^{-1}[2]\big] + 2\cdot 2 < \ldots < LCP\big[SA^{-1}[n]\big] + 2\cdot n = 2n$, and therefore the LCP array can be succinctly represented by a bit array of $2n$ bits, called SLCP (succinct LCP): $SLCP[j] \leftarrow 1$ if and only if $LCP\big[SA^{-1}[i]\big] + 2i = j$ for some $i \in [1, n]$. Upon receiving a query for $LCP[k]$, $1 \leq k \leq n$, we return $LCP[k] = Select_1\big(SLCP, SA[k]\big) - 2\,SA[k]$.

Our space-efficient SLCP construction. We compute the SLCP bit array using the wavelet tree of T_{bwt} for space efficiency (Algorithm 3). The algorithm uses a similar high-level structure as the one in [16], computing the LCP array values in the order of text positions, but does not use the suffix array, inverse suffix array and the LCP array, and therefore achieves space efficiency. We also use Fact 3 to further speed up our algorithm (steps 6–7, Algorithm 3).

Fact 3 ([26]): If $T_{\text{bwt}}[i] = T_{\text{bwt}}[i-1]$, then $LCP[i] = LCP\big[LF(i)\big] - 1$.

Proof for Lemma 4.

Proof: The space cost is derived from the following space usages: $n\log\sigma$ bits for T, $nH_0(T) + O(n\log\log n/\log_\sigma n) \leq n\log\sigma$ bits for $\mathcal{W}$ (Lemma 2), $\sigma\log\sigma$ bits for Σ, $\sigma\log n$ bits for array C, $nH_0(\mathcal{B}) + o(n) < n + o(n)$ bits for $\mathcal{B}$ (Lemma 3), $\epsilon n\log n$ bits for $\mathcal{I}$ (Lemma 3), and $2n$ bits for SLCP. The total space cost is $O\big(n\log\sigma + (\sigma+\epsilon n)\log n\big)$ bits. Regarding the time cost, let us look at one iteration of the `while` loop (step 5): it takes $O\big((1/\epsilon)\log\sigma\big)$ time for computing the $SA[i-1]$ at step 9 (Lemma 3); it takes $O(\log\sigma)$ time for computing $\Phi(i)$ at step 13 (Lemma 3); the amortized time cost for step 10 is $O(1)$; All other steps take constant time. The `while` loop (step 5) has $n-1$ iterations, so the total time cost is $O\big((1/\epsilon)n\log\sigma\big)$. □

Algorithm 3: SLCP construction using the BWT and wavelet trees

Input: $T, \mathcal{W}, \Sigma, C, SA^{-1}[1], \mathcal{B}, I$ /* $\mathcal{W}$:wavelet tree of T_{bwt} */
Output: SLCP /* $2n$-bit succinct representation of LCP */

1 $SLCP[2n] \leftarrow 1$ /* $LCP[1] \leftarrow 0$ */
2 $i \leftarrow SA^{-1}[1]$ /* suffix index of text pos being processed. */
3 $j \leftarrow 1$ /* $j = SA[i]$ */
4 $h \leftarrow 0$
5 **while** $i > 1$ **do** /* $SA^{-1}[n] = 1; T[n] =$' $', smallest suffix. */
6 **if** $T_{\text{bwt}}[i] = T_{\text{bwt}}[i-1]$ **then** /* $T_{\text{bwt}}[]$:$\mathcal{W}$'s member query */
7 $h \leftarrow h - 1$
8 **else**
9 $k \leftarrow SA[i-1]$ /* Use Lemma 3 to compute $SA[\cdot]$ */
10 **while** $T[j+h] = T[k+h]$ **do** $h \leftarrow h+1$
11 **end**
12 $SLCP[h+2j] \leftarrow 1$
13 $i \leftarrow \Phi(i)$ /* Use Lemma 3 to compute $\Phi(\cdot)$ */
14 $j \leftarrow j+1$
15 **if** $h > 0$ **then** $h \leftarrow h-1$
16 **end**

Proof for Lemma 5.

Proof: Recall that we retrieve $LCP[i] = Select_1\big(SLCP, SA[i]\big) - 2\,SA[i]$. Using C, Σ, the wavelet tree of T_{bwt}, $\mathcal{B}$, and $\mathcal{I}$, we can retrieve $SA[i]$ in $O\big((1/\epsilon)\log\sigma\big)$ time (Lemma 3). Then by using $SA[i]$ and the succinct representation of SLCP, we can finish the $Select$ operation in constant time (Lemma 1). Thus, the total time cost for retrieving $LCP[i]$ is $O\big((1/\epsilon)\log\sigma\big)$. The space cost is the same as that for constructing SLCP (Lemma 4) plus an additional $o(n)$ bits for the succinct representation of SLCP and minus the space cost for T. □

APPENDIX B

Proof for Lemma 6.

Proof: By definition, we know P_{m_i} occurs $r_{m_i} - l_{m_i} + 1 \geq 2$ times in the text T as the m-character prefix of the suffixes that belong to the suffix array interval $[l_{m_i}, r_{m_i}]$.

All the one-character right extensions of P_{m_i} (if they exist) appear as the $(m+1)$-character prefixes of the suffixes in the suffix array interval $[l_{m_i}, r_{m_i}]$. Since the length of the longest common prefix of the suffixes in the suffix array interval $[l_{m_i}, r_{m_i}]$ is m, any one-character right extension of P_{m_i} occurs less than $r_{m_i} - l_{m_i} + 1$ times in the text T. All the one-character left extensions of P_{m_i} (if they exist) are the $r_{m_i} - l_{m_i} + 1$ subtexts of $(m+1)$ characters in T,

which are the m-character longest common prefix of the suffixes in the suffix array interval $[l_{m_i}, r_{m_i}]$ prepended by each of the characters in $T_{\text{bwt}}[l_{m_i}, r_{m_i}]$. If the characters in $T_{\text{bwt}}[l_{m_i}, r_{m_i}]$ are not the same, any one-character left extension of P_{m_i} appears less than $r_{m_i} - l_{m_i} + 1$ times in T. Since any one-character left or right extension of P_{m_i} occurs less than $r_{m_i} - l_{m_i} + 1$ times, any extension of P_{m_i} in T occurs less than $r_{m_i} - l_{m_i} + 1$ times, which finishes the proof. □

The next lemma shows that searching P_{m_i}'s is sufficient for finding maximal repeats.

Proof for Lemma 7.

Proof: Let P be a maximal repeat and occur occ times in the text. Let $|P| = p$. Since P is a repeat, it must occur as a common prefix of suffixes of a suffix array interval, say $[l, r]$, then $r - l + 1 = occ$. Because P is a maximal repeat, any right extension of P occurs less than occ times, meaning that any other longer prefixes (if they exist) of the suffixes in the suffix array interval $[l, r]$ occur less than occ times in the text. So, $\min\{LCP[i] \mid l + 1 \le i \le r\} = m$. Because P is not a prefix of the suffixes that are out of the suffix array interval $[l, r]$, we have $LCP[l] < m$ and $LCP[r + 1] < m$ if $r < n$, which implies that $[l, r]$ is one of suffix array intervals of candidate maximal repeats of size p. So P must occur as P_{m_i} for $m = p$ and some $i \in [1, k_p]$. □

Proof for Theorem 1.

Proof: By Lemma 6 and 7, we know that in order to find all the maximal repeats, it suffices to first find all P_{m_j} for all $m \in [1, n-1]$ and $j \in [1, k_m]$ where $R_m \neq \emptyset$, then those P_{m_j} whose all occurrences in the text do not have a unique preceding character are the maximal repeats of size m. For any particular $m \in [1, n-1]$ and $j \in [1, k_m]$ where $R_m \neq \emptyset$, the suffix array interval $[l_{m_j}, r_{m_j}]$ is checked by the algorithm when $pos = \min\{k \in [l_{m_j} + 1, r_{m_j}] \mid LCP[k] = m\}$ is traversed by step 3, because 1) steps 1– 2 guarantee pos will be traversed at some point; 2) steps 5–6 return the suffix array interval $[l, r]$; and 3) the condition checking at step 9 guarantee $[l_{m_j}, r_{m_j}]$ cannot be extended. Note that $LCP[r + 1]$ is always smaller than i, because we traverse the LCP array values from the smallest to the largest, and for a particular LCP value, we traverse its different occurrences from the left to the right. P_{m_j} corresponding to the suffix array $[l, r]$ is further verified by step 10 whether it can be left extended. Step 4 filters out all the suffix array intervals of maximal repeats of size smaller than ml. Step 8 filters out all the suffix array intervals of maximal repeats whose occ is less than mo.

The time and space complexity bounds are derived from the cost for constructing the building blocks and the body of Algorithm 2. The proof is simply adding them up. Suppose we use the method in [14]. Table 3 shows the construction cost of the building blocks. The time cost for steps 1–10 of Algorithm 2 (finding the maximal repeats without reporting them) takes time $O(n \log n)$, because we have total $n - 1$ LCP array values to traverse at steps 1–2, and steps 5–6 take $O(\log n)$ time dominating other steps in the loop from steps 3–10. So finding the maximal repeats takes $O(n \log n + (1/\epsilon)n \log \sigma)$ of time and use $O\big(n \log \sigma + (\sigma + \epsilon n) \log n + n \log \sigma'\big)$ bits of space, where σ' is the number of distinct values in the LCP array.

Reporting the text of a maximal repeat P of size p at step 11 takes $O\big((1/\epsilon) \log \sigma\big)$ for computing $SA[l]$ (Lemma 3) and additional $O(p)$ of time to list the characters of P. Reporting the text locations of P takes $O\big(occ \cdot (1/\epsilon) \log \sigma\big)$ time because each $SA[k]$ computation at step 12 takes $O\big((1/\epsilon) \log \sigma\big)$ time, where occ is the number of occurrences of P in T. □

	Time Cost	Space Cost (bits)	Input	Notes
T	0	$n \log \sigma$	$\emptyset$	T is given
T_{bwt}	$O(n \log \log \sigma)$	$O(n \log \sigma)$	T	[14]
B_{bwt}	$O(n)$	$O(n \log \sigma)$	T_{bwt}	Section 4 Bulletin 3
$\mathcal{W}$	$O(n \log \sigma)$	$O(n \log \sigma)$	T_{bwt}	Lemma 2
$\mathcal{B}$	$O(n \log \sigma)$	$O(nH_0(T) + \sigma \log n)$	$C, \Sigma, \mathcal{W}$	Lemma 3 Bulletin 3
$\mathcal{I}$	$O(n \log \sigma)$	$O(nH_0(T) + (\sigma + \epsilon n) \log n)$	$C, \Sigma, \mathcal{W}, \mathcal{B}$	Lemma 3 Bulletin 4
B_{lcp}	O(n)	2n	$\emptyset$	Section 4 Bulletin 2
SLCP	$O((1/\epsilon)n \log \sigma)$	$O(n \log \sigma + (\sigma + \epsilon n) \log n)$	$T, C, \Sigma, \mathcal{W}, SA^{-1}[1], \mathcal{B}, \mathcal{I}$	Lemma 4
$\mathcal{W}_{\text{lcp}}$	$O((1/\epsilon)n \log \sigma + n \log \sigma')$	$O((\sigma + \epsilon n) \log n + n \log \sigma')$	$C, \Sigma, \mathcal{W}, \mathcal{B}, \mathcal{I}$, SLCP	Section 4 Bulletin 1

TABLE 3
Construction time and space cost of the building blocks. $\mathcal{W}$ is the wavelet tree of T_{bwt}

ACKNOWLEDGMENTS

We are grateful to the authors of [1], [23], and [15] for providing their source code. We thank the reviewers especially one of them for the helpful comments to improve our presentation. This research was supported in part by the U.S. National Science Foundation grant CCF–0621457.

REFERENCES

[1] V. Becher, A. Deymonnaz, and P. A. Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–1753, 2009.

[2] B. Behzadi and F. L. Fessant. Dna compression challenge revisited: A dynamic programming approach. In *Annual Symposium on Combinatorial Pattern Matching*, 2005.

[3] G. Benson. Tandem repeats finder: a program to analyze dna sequences. *Nucleic Acids Research*, 27(2):573–580, 1999.

[4] M. Burrows and D.J. Wheeler. A block sorting data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[5] A. T. Castelo, W. Martins, and G. R. Gao. Troll–tandem repeat occurrence locator. *Bioinformatics*, 18(4):634–636, 2002.

[6] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12:1–24, 2008.

[7] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of ACM*, 52(4):552–581, 2005.

[8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *International Symposium on String Processing and Information Retrieval(SPIRE)*, pages 150–160, 2004.

[9] J. Fischer, V. Mäkinen, and N. Välimäki. Space efficient string mining under frequency constraints. In *Proceedings of IEEE International Conference on Data Mining*, pages 193–202, 2008.

[10] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[11] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(32):378–407, 2005.

[12] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[13] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

[14] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[15] J. Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[16] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Symposium on Combinatorial Pattern Matching*, pages 181–192, 2001.

[17] M. O. Külekci, J. S. Vitter, and B. Xu. Time- and space-efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet trees. In *the Proceedings of IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 622–625, 2010.

[18] S. Kurtz. Reducing the space requirements of suffix trees. *Softw. Pract. Exp.*, 29(13):1149–1171, 1999.

[19] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Reputer: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.

[20] S. Kurtz and C. Schleiermacher. Reputer: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[21] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

[22] R. Lippert. Space-efficient whole genome comparisons with Burrows-Wheeler transforms. *Journal of Computational Biology*, 12(4):407–415, 2005.

[23] R. A. Lippert, C. M. Mobarry, and B. Walenz. A space-efficient construction of the Burrows-Wheeler transform for genomic data. *J. of Comp. Bio.*, 12(7):943–951, 2005.

[24] X. Liu and L. Wang. Finding the region of pseudo-periodic tandem repeats in biological sequences. *Algorithms for Molecular Biology*, 1(1):2, 2006.

[25] V. Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundamenta Informaticae*, 56:191–210, October 2002.

[26] G. Manzini. Two space saving tricks for linear time lcp array computation. In *Scandinavian Workshop on Algorithm Theory*, pages 372–383, 2004.

[27] G. Manzini and M. Rastero. A simple and fast dna compressor. *Software – Practice and Experience*, 34:1397–1411, 2004.

[28] H. M. Martinez. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Res.*, 11(13):4629–4634, 1983.

[29] E. H. McConkey. *Human Genetics: The Molecular Revolution*. Jones and Bartlett, Boston, MA, 1993.

[30] J. C. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. *Theor. Comp. Sci.*, 385(1-3):127–136, 2007.

[31] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[32] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *String Processing and Information Retrieval Symposium*, pages 90–101, 2009.

[33] A. Poddar, N. Chandra, M. Ganapathiraju, K. Sekar, J. Klein-Seetharaman, R. Judith, R. Reddy, and N. Balakrishnan. Evolutionary insights from suffix array-based genome sequence analysis. *Journal of Biosciences*, 32(5):871–881, 2007.

[34] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. on Algorithms*, 3(4):43, 2007.

[35] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[36] S. Saha, S. Bridges, Z. V. Magbanua, and D. G. Peterson. Computational approaches and tools used in identification of dispersed repetitive dna sequences. *Tropical Plant Biology*, 1(1):85–96, 2008.

[37] S. Saha, S. Bridges, Z. V. Magbanua, and D. G. Peterson. Empirical comparison of ab initio repeat finding programs. *Nucleic Acids Research*, 36(7):2284–2294, 2008.

[38] J. Sirén. Sampled longest common prefix array. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 227–237, 2010.

[39] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *Journal of Experimental Algorithmics (JEA)*, 14:2:4.2–2:4.23, 2010.

[40] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Foundations and Trends in Theoretical Computer Science. Now Publishers, Hanover, MA, 2008.

M. Oğuzhan Külekci received his B.Sc. in Computer Engineering from Bogazici University in 1998 and Ph.D. in Computer Science and Engineering from Sabancı University in 2006. Since 1999, he has been with the Turkey National Research Institute of Electronics and Cryptology (TÜBİTAK-UEKAE), where he currently is a chief researcher. In Fall 2009 and Spring 2010, he was with the Department of Computer Science and Engineering of Texas A&M University as an Assistant Research Professor. His main research area is algorithms and computation, especially focusing on text algorithmics, both from theoretical and practical perspectives.

Jeffrey Scott Vitter received the B.S. degree in mathematics with highest honors from the University of Notre Dame in 1977, the Ph.D. degree in computer science from Stanford University in 1980, and the MBA degree from Duke University in 2002. He is the provost and executive vice chancellor and the Roy A. Roberts Distinguished Professor at the University of Kansas. His academic home is the Department of Electrical Engineering and Computer Science, and he is a member of the Information and Telecommunication Technology Center. From 2008 to 2010, he was on the faculty at Texas A&M University, where he served from 2008 to 2009 as provost and executive vice president for academics. From 2002 to 2008, he was the Frederick L. Hovde Dean of the College of Science and Professor of Computer Science at Purdue University. From 1993 to 2002, he was the Gilbert, Louis, and Edward Lehrman Professor of Computer Science at Duke University. He served from 1993 to 2001 as chair of the Department of Computer Science and from 1997 to 2002 as co-director of Duke's Center for Geometric and Biological Computing. From 1980 to 1992, he advanced through the faculty ranks at Brown University.

Dr. Vitter is a Guggenheim fellow, ACM fellow, IEEE fellow, AAAS fellow, NSF Presidential Young Investigator, and Fulbright Scholar. He has received the IBM Faculty Development award, ACM Recognition of Service award (twice), and 2009 ACM SIGMOD Test of Time award. He sits on the board of advisors of the School of Science and Engineering at Tulane University. From 2000 to 2009, he served on the board of directors of the Computing Research Association (CRA), where he continues to co-chair the Government Affairs Committee. He has served as chair, vice-chair, and member-at-large of ACM SIGACT and has served on the EATCS executive committee.

Dr. Vitter is author of the book *Algorithms and Data Structures for External Memory*, coauthor of the books *Design and Analysis of Coalesced Hashing* and *Efficient Algorithms for MPEG Video Compression*, coeditor of the collections *External Memory Algorithms* and *Algorithm Engineering*, and coholder of patents in the areas of external sorting, prediction, and approximate data structures. His research interests span the design and analysis of algorithms, external memory algorithms, data compression, databases, compressed data structures, parallel algorithms, machine learning, random variate generation, and sampling. He serves or has served on the editorial boards of *Algorithmica*, *Communications of the ACM*, *IEEE Transactions on Computers*, *Theory of Computing Systems*, and *SIAM Journal on Computing*, and has edited several special issues. He proposed the concept and participated in the design of what has become the Purdue University Research Expertise database (PURE) and the Indiana Database for University Research Expertise (INDURE), www.indure.org.

Bojian Xu received his B.E. in Computer Science and Engineering from Zhejiang University, Hangzhou, China in 2000. He worked as an engineer in the telecommunication industry from 2000 to 2004. He received his Ph.D. in Computer Engineering from Iowa State University in 2009 before he joined the Department of Computer Science and Engineering of Texas A&M University as an Assistant Research Professor. He is currently an Assistant Research Professor at the Information and Telecommunication Technology Center (ITTC) of the University of Kansas. His research interests are in designing algorithms and building systems for managing large data sets.