

Time- and Space-efficient Maximal Repeat Finding Using the Burrows-Wheeler Transform and Wavelet Trees

M. Oğuzhan Külekci*, Jeffrey Scott Vitter† and Bojian Xu†

* National Research Institute of Electronics & Cryptology, Turkey. kulekci@uekae.tubitak.gov.tr

† Information and Telecommunication Technology Center, University of Kansas, USA. {jsv, bojianxu}@ku.edu

Abstract—Finding repetitive structures in genomes is important to understand their biological functions. Many modern genomic sequence data compressors also highly rely on finding the repeats over the sequences. The notion of maximal repeats captures all the repeats in a space-efficient way. Prior works 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 as large as the text size with the best engineering efforts, prohibiting their usability on massive data such as the whole human genome.

Our technique is based on the Burrows-Wheeler Transform and wavelet trees. For genomic sequences stored using one byte per base, the space usage of our method is less than double of 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 normal 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.

Keywords—repeats; maximal repeats; Burrows-Wheeler transform; wavelet trees;

I. INTRODUCTION

Finding repetitive structures in genomes and proteins is important in understanding their biological functions [1]. 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. In addition, a number of significant problems in molecular sequence analysis can be reduced to repeat finding. Finding repeats is also useful in DNA sequence compression, which is known as one of the most challenging tasks in the data compression field. DNA sequences consist only of symbols in {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 finding the repeats in the sequences [2], [3]. For all these purposes, finding

repeats is the first and critical step and needs to be conducted efficiently.

Maximal repeats [1] that are repeats whose extensions occur fewer times in the text captures all the repeats in a space-efficient way. However, current techniques for finding maximal repeats are either based on suffix trees [1] or suffix arrays [4], both requiring enormously space usage caused by the large space cost of the suffix trees and suffix arrays and their auxiliary data structures. In fact, their space usage is 19–50 times of the text size with the best engineering efforts, 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 huge internal memory are used. It is also worth noting that since the suffix array based method also uses other large auxiliary data structures such as 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 [5]. It is also not clear how a compressed suffix tree can be used for finding maximal repeats with good practical performance [6].

Our contribution: We provide an option for people to use normal computers with limited internal memory capacity to find maximal repeats in massive text data such as the whole human genome using an acceptable amount of time. Our method is based on the Burrow-Wheeler Transform (BWT) [7] and wavelet trees [8] with provable time and space efficiency and good usability in practice without any assumption on the alphabet size. Overall, for genomic sequences stored using one byte per base, the space usage of our method is less than double of 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 when the internal memory capacity is limited, since the prior methods have to use external memory [9]. To the best of our knowledge, this is the first work that enables a normal 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. We fully implemented our algorithm

This research was supported in part by US NSF grant CCF-0621457. Part of the work was done while authors were with Texas A&M University. Full version of this paper is available at authors' webpage.

as a general-purpose open-source software for public use.

A. Problem

Let $T = T[1..n] = t_1 t_2 \dots t_n$ be a text of size n . 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 . Without loss of generality, we assume $\Sigma[1] < \Sigma[2] < \dots < \Sigma[\sigma]$. t_n is added only for ease of text processing. We want to find all the *maximal repeats* of T .

Definition 1 (Maximal Repeat [1]): 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.

Maximal repeats capture all the repeats of a text in a space-efficient way.

Theorem 1 ([1]): Any text of size n has at most n maximal repeats.

II. FINDING THE MAXIMAL REPEATS

We refer the readers to the full paper for an intuitive description of our method. The full paper also has a description of important concepts and techniques such as suffix array (SA), inverse suffix array, succinct longest common prefix (slcp) array, succinct bit arrays, wavelet trees, and the Burrows-Wheeler transform (BWT). Some of these techniques serve as the building blocks in our method.

By the definition of maximal repeat, we know the lengths of maximal repeats of a text of size n are in the range $[1, n-1]$. 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, then find the actual maximal repeats from the candidate set.

Definition 2: 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 $R_m = \langle [l_{m_1}, r_{m_1}], [l_{m_2}, r_{m_2}], \dots, [l_{m_{k_m}}, r_{m_{k_m}}] \rangle$, for some integer k_m , such that for all $i \in [1, k_m]$:
 $- l_{m_i} = \min\{j \in [r_{m_{i-1}} + 1, n-1] \mid lcp[j+1] \geq m\}$ ($r_{m_0} \equiv 0$)
 $- r_{m_i} = \max\{j \in [l_{m_i} + 1, n] \mid lcp[\alpha] \geq m, \forall \alpha \in [l_{m_i} + 1, j]\}$
 $- \min\{lcp[j] \mid j \in [l_{m_i} + 1, r_{m_i}]\} = m$
 $-$ 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.

Definition 3: 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 $|P_{m_i}| = m$ for all $i \in [1, k_m]$. The next lemma shows that P_{m_i} is a maximal repeat if its left extension occurs fewer times than P_{m_i} .

Lemma 1: For any $m \in [1, n-1]$ such that $R_m \neq \emptyset$ and any $i \in [1, k_m]$, if the characters in $T_{\text{bwt}}[l_{m_i} \dots r_{m_i}]$ are

Algorithm 1: Finding the maximal repeats of T

```

Input:  $\mathcal{W}_{\text{lcp}}, B_{\text{lcp}}, B_{\text{bwt}}, \mathcal{W}, T, B, \mathcal{I}, ml, mo$ 
/* Parameters are defined in the full paper. */
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
2   for  $j \leftarrow 1 \dots v_i$  do
3      $pos_{i,j} \leftarrow \text{Select}_{\text{leaves}[i]}(lcp, j)$  /*  $\mathcal{W}_{\text{lcp}}$ 's  $\text{Select}()$  */
4     if  $\text{leaves}[i] < ml$  then  $\{B_{\text{lcp}}[pos_{i,j}] \leftarrow 1; \text{continue}\}$ 
/* Use the binary bit tree in [4] over  $B_{\text{LCP}}$ 
to compute the min and max */
5      $l \leftarrow \max\{k \mid k < pos_{i,j} \text{ and } B_{\text{lcp}}[k] = 1\}$ 
6      $r \leftarrow \min\{k \mid k > pos_{i,j} \text{ and } B_{\text{lcp}}[k] = 1\} - 1$ 
7      $B_{\text{lcp}}[pos_{i,j}] \leftarrow 1$ 
8     if  $r - l + 1 < mo$  then continue
9     if  $l > 0$  and  $\text{Member}(lcp, l) = \text{leaves}[i]$  then continue
/*  $[l, r]$ : SA interval of a candidate max repeat
*/
10    if ( $\text{Member}(B_{\text{bwt}}, l) = 1$  and
( $\text{Rank}_1(B_{\text{bwt}}, r) - \text{Rank}_1(B_{\text{bwt}}, l) = 1$ )) or
( $\text{Rank}_1(B_{\text{bwt}}, r) - \text{Rank}_1(B_{\text{bwt}}, l) = 0$ )) then
/* Use  $T_{\text{BWT}}$  to compute  $SA[l]$  */
11    Output  $T[SA[l] \dots SA[l] + \text{leaves}[i]]$  /* Repeat */
12    for  $k \leftarrow l \dots r$  do Output  $SA[k]$  /* Text locations
*/
13    end
14  end
15 end

```

not the same, then P_{m_i} is a maximal repeat of size m in T . (*proof in the full paper.*)

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

Lemma 2: 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 the full paper.*)

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 the maximal repeat finding.

A. Algorithm

We first prepare the following data structures (details in the full paper).

- 1) \mathcal{W}_{lcp} is the wavelet tree of the lcp array. Using the $slcp$ bit array, \mathcal{W}_{lcp} can be constructed in $O((1/\epsilon)n \log \sigma + n \log \sigma')$ time, where σ' 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. The construction of \mathcal{W}_{lcp} takes another $O(n \log \sigma')$ time, so the total time cost is $O((1/\epsilon)n \log \sigma + n \log \sigma')$. The space cost of constructing \mathcal{W}_{lcp} is $O(n \log \sigma')$.
- 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 some purpose that will be clear later. By using a $2n$ -bit binary bit tree structure designed by [4], which can be constructed in $O(n)$

time, given an integer $i \in [1, n]$ such that $B_{lcp}[i] = 0$, we can get $\max\{k \mid k < i \text{ and } B_{lcp}[k] = 1\}$ and $\min\{k \mid k > i \text{ and } B_{lcp}[k] = 1\}$ and turn on $B_{lcp}[i]$ in $O(\log n)$ time.

- 3) $B_{bwt}[1 \dots n]$ is a bit array of size n . $B_{bwt}[i] = 1$ iff $i = 1$ or $T_{bwt}[i] \neq T_{bwt}[i - 1]$, so that for any $1 \leq j < k \leq n$, all the characters in $T_{bwt}[j \dots k]$ are the same iff $B_{bwt}[j + 1 \dots k]$ are all 0-bits. Clearly, B_{bwt} can be constructed in $O(n)$ time.

Algorithm 1 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}_{lcp} . For each particular repeat length, we find the corresponding suffix array intervals of candidate maximal repeats. Let $leaves[i]$ denote the i th smallest lcp value represented by the i th leftmost leaf node of \mathcal{W}_{lcp} . Let v_i denote the number of occurrences of $leaves[i]$ in the lcp array. For $j = 1 \dots v_i$, let $pos_{i,j}$ denote the position of the j th leftmost occurrence of the lcp value $leaves[i]$ in the lcp array. Each $lcp[pos_{i,j}]$ can be retrieved via *Select* operation on \mathcal{W}_{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 $lcp[k] \geq leaves[i]$ for all $k \in [l + 1, r]$ and $\min\{lcp[k] \mid l + 1 \leq k \leq r\} = leaves[i]$. 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 $lcp[l] = leaves[i]$ (step 9), meaning that the longest common prefix of the suffixes belonging to $[l - 1, r]$ is also $leaves[i]$, then $[l, r]$ is not a suffix array interval of a candidate maximal repeat of size $leaves[i]$. Any suffix array interval $[l, r]$ of candidate maximal repeats of size $leaves[i]$ will be detected by the algorithm when $pos_{i,j} = \min\{k \in [l + 1, r] \mid lcp[k] = leaves[i]\}$ 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 2: Given a text T of size n drawn from an alphabet of size σ , Algorithm 1 can find all the maximal repeats of T by setting $ml = 1$ and $mo = 2$ and using $O(n \log \sigma + (\sigma + \epsilon n) \log n + n \log \sigma')$ bits of space and $O(n \log n + (1/\epsilon)n \log \sigma)$ time. Reporting the text of a particular maximal repeat P of size p takes additional time of $O((1/\epsilon) \log \sigma + p)$. Reporting the text locations of P takes additional time of $O(occ \cdot (1/\epsilon) \log \sigma)$, where occ is the number of occurrences of P in T . (*proof in the full paper.*)

	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	\approx 28, 245	\approx 19.50	2, 232	1.54
W.H.G.	2, 759.57	\approx 53, 811	\approx 19.50	5, 494	1.99

Figure 1. Space usage comparison between the SA-based method [4] 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.

Comments: The space usage in the above theorem is used for the data structure construction. By setting $\epsilon = 1/32$, which is often smaller than or comparable to $1/\log n$ even for large texts, it becomes $O(n \log(\sigma \sigma'))$, where σ' , the number of distinct values in the lcp array, is often a small number. The final data structures used for the maximal repeat finding uses space of $O(n(\log \sigma + H'_0))$ bits, where H'_0 , the order-0 empirical entropy of the lcp array, is often much smaller than $\log \sigma'$ due to the skewness in the lcp array values. The time complexity of our method $O(n \log n + (1/\epsilon)n \log \sigma) = O(n \log n)$ matches the time complexity of the SA-based method [4].

III. IMPLEMENTATION AND EXPERIMENTS

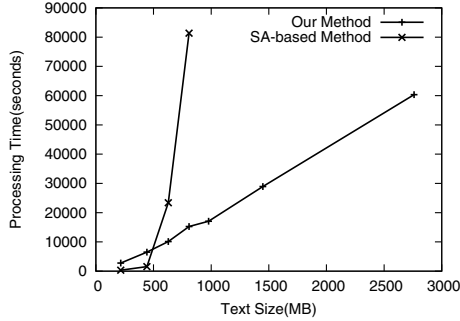
We fully implemented our algorithm in C++¹. We developed our own code for all the parts of the algorithm except the BWT construction. We use Lippert *et. al.*'s C code [11] to build BWT for genome sequences. *Our implementation is full and generic in that it supports maximal repeat finding in texts of any alphabet size.*

Experimental set-up and environment. We used g++ 4.4.1 as the compiler 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, and only one core is 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 human genome sequences from NCBI² to test the efficiency and generability of our method. We removed all the masked 'N' symbols from the genome sequences, so they only contain symbols from {ACGT}.

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 maximal repeats of size larger than or equal to ml will be reported. We set the parameter

¹Source code: <http://www.itc.ku.edu/~bojianxu/publications>

²ftp://ftp.ncbi.nlm.nih.gov/genomes/H_sapiens/Assembled_chromosomes



	Text size (MB)	Construction time for SA	SA-based total time	SA construction time percentage	Our 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

Figure 2. Timing performance comparison between the SA-based method and our method. Time is measured in seconds.

$\epsilon = 1/32$ 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 use the system time to measure the programs' time cost. We use 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 code, data, and shared libraries plus pages that have been swapped out.

Main observations. We compare the performance of our algorithm with the performance of the suffix array-based method [4]. The experimental study led to the following main observations:

1) The SA-based method consistently consumes more than 19 times of the text size, while our method uses space no more than double of 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 (Figure 1).

2) When its input size exceeds 600MB thus its workspace becomes larger than 11GB (Figure 1), exceeding the 8GB internal memory limit, the SA-based method becomes unacceptably slow because of the page faults and swapping (Figure 2). The SA-based method spent so long in processing Chromosome 1-5 (490,016 seconds \approx 5.7 days) and data are not plotted in Figure 2 in order to get clear plots for other points on the curves. The SA-based method even did not terminate in ten days for other larger inputs.

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 (table in Figure 2). 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 of other data sets can be found in the table of Figure 2. The SA construction time is negligible when the input size is large, meaning that using external memory-efficient SA construction algorithm [9] 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 comprises about three billion bases (ACGT), using a normal computer with 8GB internal memory (actual internal memory used is less than 6GB, Figure 1) in less than 17 hours (Figure 2).

REFERENCES

- [1] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.
- [2] G. Manzini and M. Rastero, "A simple and fast dna compressor," *Software – Practice and Experience*, vol. 34, pp. 1397–1411, 2004.
- [3] B. Behzadi and F. L. Fessant, "Dna compression challenge revisited: A dynamic programming approach," in *Annual Symposium on Combinatorial Pattern Matching*, 2005.
- [4] 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*, vol. 25, no. 14, pp. 1746–1753, 2009.
- [5] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 32, pp. 378–407, 2005.
- [6] J. Fischer, V. Mäkinen, and G. Navarro, "Faster entropy-bounded compressed suffix trees," *Theoretical Computer Science*, vol. 410, no. 51, pp. 5354–5364, 2009.
- [7] M. Burrows and D. Wheeler, "A block sorting data compression algorithm," Digital Systems Research Center, Tech. Rep., 1994.
- [8] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *ACM-SIAM Symposium on Discrete Algorithms*, 2003, pp. 841–850.
- [9] J. S. Vitter, *Algorithms and Data Structures for External Memory*, ser. Foundations and Trends in Theoretical Computer Science. Hanover, MA: now Publishers, 2008.
- [10] J. Kärkkäinen, "Fast bwt in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 249–257, 2007.
- [11] 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.*, vol. 12, no. 7, pp. 943–951, 2005.