ACSIJ Advances in Computer Science: an International Journal, Vol. 4, Issue 5, No.17 , September 2015
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Repeat Finding Techniques, Data Structures and Algorithms in DNA sequences: A Survey

Freeson Kaniwa[1], Heiko Schroeder[2] and Otlhapile Dinakenyane[3]

[1] Department of Computer Science, Botswana International University of Science and Technology
Private Bag 16, Palapye, Botswana
fkaniwa@gmail.com

[2] Department of Computer Science, Botswana International University of Science and Technology
Private Bag 16, Palapye, Botswana
schroederh@biust.ac.bw

[3] Department of Computer Science, Botswana International University of Science and Technology
Private Bag 16, Palapye, Botswana
dinakenyaneo@biust.ac.bw

Abstract

DNA sequencing technologies keep getting faster and cheaper leading to massive availability of entire human genomes. This massive availability calls for better analysis tools with a potential to realize a shift from reactive to predictive medicine. The challenge remains, since the entire human genomes need more space and processing power than that can be offered by a standard Desktop PC for their analysis. A background of key concepts surrounding the area of DNA analysis is given and a review of selected prominent algorithms used in this area. The significance of this paper would be to survey the concepts surrounding DNA analysis so as to provide a deep rooted understanding and knowledge transfer regarding existing approaches for DNA analysis using Burrows-Wheeler transform, Wavelet tree and their respective strengths and weaknesses. Consequent to this survey, the paper attempts to provide some directions for future research.

Keywords: *DNA sequences, Repeats, Burrows-Wheeler transform, Wavelet trees.*

1. Introduction

DNA sequences carry genetic information for each human being. A fast and accurate DNA analysis in DNA sequences is one of the basic algorithm design problems. Generating DNA sequences has become easy and cheaper due to current advanced technology which have suddenly made DNA sequences easily available [1-4]. DNA sequences consists of repetitive structures usually called DNA repeats. DNA repeats are broadly categorized into two categories thus Interspersed and Tandem repeats. Biologists further categorizes between three types of Tandem Repeats (TRs) namely microsatellites, minisatellites and satellites. These three TRs differ in terms of the length of their consensus motif. Microsatellites have a motif length of ($2 \leq |motif| \leq 5$), minisatellites have a motif length of ($5 < |motif| \leq 100$) and the motif length of satellites is ($|motif| > 100$) [5].

Interspersed Repeats (IRs) are distributed throughout genomes in a non-tandem, albeit non-random, manner. The majority of IRs are transposons, which are sequences that can directly or indirectly jump or move from one position to another. Classification of transposons is usually difficult since new types of these repeats are discovered at a rapid rate and their evolutionary relationships among repeat groups are unclear. Transposons can be broadly divided into two classes namely retrotransposons and DNA transposons. Retrotransposons are replicated and mobilized through an RNA intermediate via a *copy-and-paste* mechanism involving the enzyme reverse transcriptase. In contrast, DNA transposons utilize *cut-and-paste* or *copy-and-paste* methods of transposition that do not involve an RNA intermediate [6]. The effects of transposons in maize are shown in Figure 1.

Figure 1 Effects of Transposons in maize [7]

The remainder of the paper is laid out as follows, in Section II we discuss types of repeats and their characteristics. Section III provide a discussion of key data structures and their limitations. Section IV we provide

different techniques employed in DNA analysis. Section V we provide an overview of the algorithmic details of repeat finding algorithms.

2. Repeats

A DNA sequence can be viewed as a sequence of an alphabet consisting of four letters of A, C, G and T extracted from the molecules of the DNA sequencing process (Σ = {A, C, G, T}). DNA sequencing is a process of determining the precise order of nucleotides within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases which are adenine, guanine, cytosine, and thymine found in a DNA strand. For instance, a DNA sequence can be in the following sequence S_1 = {G G G G A C G T A C G T A C G T A A G T A C G T}. The sequence S go up to lengths of approximately 3 billion base pairs (bp) which makes it challenging to process due to high space and processing requirements. Period and exponent are frequently used terms in repeat searching algorithms. Period refers to the length or size of a repeat while exponent refers to the frequency of the repeat occurrence in a sequence. An example of a TR, using the sequence S_1 is A C G T and for IR is T A C G all of period 4.

Another interesting type of a repeat is a Maximal Repeat (MR). A maximal repeat can be mathematically defined as follows, let *S* be a string of length *n*, $S = S[1...n] = s1s2...sn$, where each character si, $1 \leq i \leq n-1$, is from a finite ordered alphabet Σ of size σ, while sn = $, a special character that does not appear in Σ. Basically a maximal repeat of string S is a substring of S that occurs in S at least twice such that any extension of the substring occurs in S fewer times [8]. The notion of maximal repeats captures all types of repeats in the sequence in a space-efficient way [8]. For instance, given the following sequence S_2 = { G A T A C G T A G A T A G A T G T A C G T A C G }. G A T (as highlighted) is a maximal repeat since it occurs 3 times and its extensions (G A T A) occurs fewer times (2 times). T A C is not a maximal repeat since it can be extended to T A C G and those extensions do not occur fewer times than the repeat itself.

Another criterion for classifying repeats are exact and approximate. Using the sequence S_2 an exact repeat would be A C G T and with an exponent of 3 and an approximate repeat would be G T A C G T with period of 6 and exponent 2 with only variation on index 1 of the repeat. Approximate repeat finding involves the use of distance based definitions. In information theory distance based definitions are usually based on two distance metrics (or more) which are hamming distance, and Levenshtein distance. An approximate repeat can be defined as a repeating unit where the repeating units are similar according to some distance metric. Consider the two strings in Figure 2.

A	C	G	T	T	T	C	G	G	A
–	|	|	|	–	–	–	|	–	|
C	C	G	T	A	A	A	G	T	A
M	H	H	H	M	M	M	H	M	H

Figure 2 Two repeats aligned, M is a Miss (mismatch) and H is a Hit (match)

The second string in Figure 2 can be considered to be an approximate repeat with Levenshtein distance of 4. Levenshtein distance is defined as the minimum number of editing operations (insertion, deletion and substitution) required to make the strings equal. In most cases the number of operations is bound to a certain given number. Depending on the parameter set to determine the fuzziness of the repeats, the approximate repeats can be extracted. This technique has been successfully applied in word processors for spell check and in dynamic programing.
Hamming distance is only valid when the two strings in question have the same lengths, it is the number of mismatches when two strings are aligned character by character. From Figure 2, the hamming distance is 4 (number of operations needed to transform the second string into the first string as shown by the M and H). It is therefore possible to find repeats where each unit differs by k-mismatches. The limitation of this measure is that it does not work with repeated units which involves deletions and insertions but rather only substitutions which are countered by edit distance and alignment scores.

3. Data structures

A data structure specifies how data is organized together with access methods. In this section we present interesting key data structures which have being commonly used in the area of DNA analysis specifically in repeat finding algorithms. These data structures are important since they affect the overall performance of an algorithm. They are used for indexing text to improve the search process. Two issues emerge when dealing with data structures. The first is the construction space requirements needed for indexing and secondly the speed of the search process once the data structure have been generated.

3.1 Suffix Tree

A suffix tree is a data structure mostly used in string matching algorithms. It is constructed as a pre-processing step to improve searching for repeats in a string. This data

structure has been first introduced by Weiner [9] and was improved further by Ukkonen to have a with linear time construction of *O(n)* [10]. The suffix tree is defined and constructed as follows: Let *T = T [1..n]* be a text of length *n* over a finite alphabet . A suffix tree for *T* is a tree with *n* leaves that have different initial letters at every edge and a concatenation of labels from the root node to the leaf node. For instance, the string *S = abab*, first append the special terminal character $, so *S = abab$*, a suffix tree of *S* is a compressed trie of all suffixes of *S=abab$* and can be graphically represented as in Figure 3.

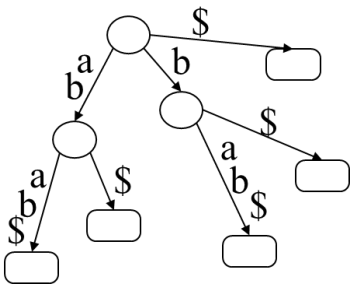

Figure 3 The Suffix Tree

The major limitation with this data structure is that of high memory requirements when processing huge sequences such as the entire human genome however it performs well when processing smaller sequences such as the X, Y chromosomes or other chromosomes. The suffix tree is a theoretically attractive option which does not perform well in practice (especially for the complete human genomes) due to a hidden huge constant factor in its complexity. For instance 100MB of genome sequence would require 5GB of memory in its construction [11].

3.2 Suffix Array (SA)

A suffix array is a more space efficient data structure developed from a suffix tree [12]. The suffix array was first introduced by Manber and Myers [12]. It is mathematically defined as follows, given a text *T* of length *n*, the suffix array for *T*, is array of integers of range *0* to *n-1* specifying the lexicographic ordering of the *n* suffixes of the string *T$*.
Table 1 shows the suffix array for a text *T = mississippi$*, with Σ = [*i,m, p, s*].

Table 1: The Suffix Array and The LCP Array

i	T_i	$T_{suf\_array[i]}$	*Suf_array[i]*	*LCP*
0	mississippi$	$	12	0
1	ississippi$	i$	11	0
2	ssissippi$	ippi$	8	1
3	sissippi$	issippi$	5	0
4	issippi$	ississippi$	2	4
5	ssippi$	mississippi$	1	0
6	sippi$	pi$	10	0
7	ippi$	ppi$	9	1
8	ppi$	sippi$	7	0
9	pi$	sissippi$	4	1
10	i$	ssippi$	6	0
11	$	ssissippi$	3	3

Assuming that *$* < Σ, The LCP column is discussed in Section 3.3. A suffix array can be best constructed in O(*n* log *n*) time and searching can be done in O(*m log n*) for a pattern of length *m* and text of length *n*.

3.3 FM-index

The name stands for Full-text index in Minute space which is a compressed full-text substring index based on Burrows-Wheeler transform (BWT) [13]. This data structure has a lot of similarities to the suffix array. It is built from Burrows-Wheeler Matrix (BWM) and the index itself consists of the L and F columns from the Burrows-Wheeler Matrix (BWM) [14]. F can be simply represented (1 integer per alphabet character) and L is highly compressible. When querying, binary search is not possible since we won't be having all the information of the BWM just like the case for the suffix array. The data structure allows compression of the input sequence but still allows fast queries on substrings. The FM-index allows finding number of occurrences of a pattern within the compressed sequence together with the position of each of the occurrences. It is also important to note that both the storage space requirements and are sub-linear as function of *n* (size of the input data). The count operation can be completed in linear time since the occurrences of pattern *P* will be next to each other in a single continuous range. The operation works by iterating backwards over the pattern. The locate operation gives the positions of the pattern. To find the occurrence of a pattern, first, the range of character is found whose suffix is the pattern in the same way the count operation works out the range. The position of every character in the range can be located and

this can be completed in $O(p + occ\ log^{\varepsilon} u)$ time with $O\left(H_k(T) + \frac{log\ logu}{log^{\varepsilon} u}\right)$ bits per input symbol for any $k \geq 0$ [13]. Where *occ* represents occurrence of a pattern *p*[*1..p*] in a text T[*1..u*].

The original authors did a further improvement to the FM-index and dubbed it 'alphabet-friendly FM-index'. It uses compression boosting techniques and the wavelet trees [15].This new FM-index supports large alphabets like the complete human genome have been shown to significantly reduce the space requirements.

3.4 Wavelet Tree

A wavelet tree is a succinct data structure which is constructed like a balanced binary tree [16]. The wavelet tree is perhaps the current most space efficient tree data structure. The tree supports 3 operations which are rank, select and access queries. The rank query, $rank_c(S,i)$ gives you the number of char c at or before position *i* in *S*. The select query, $select_c(S,j)$ returns the position of the *j*th occurrence of *c* in *S*. Rank and select queries are inverses of each other since $rank_c(S, select_c(S, j)) = j$. The access query, *S*[*i*] returns the character at index *i*. Therefore, rank, select, access queries take O($log\ |\Sigma|$) time to run. When constructing the wavelet tree, the string has to be first converted into binary search tree of bit vectors, where a 0 replaces first half of the symbols, and a 1 replaces the second half (Hence, 0: letter $\in$ first half of alphabet and 1: letter $\in$ second half of alphabet).This creates ambiguity, however at every level the alphabet is filtered and re-encoded, so the ambiguity continuously reduces until there is no ambiguity at all. The tree is defined recursively as follows:

1. Take the alphabet of the sequence, and encode the first half as 0, the second half as 1: {*w, x, y, z*} would become {0, 0, 1, 1};
2. Group each 0-encoded symbol, {w,x}, as a sub-tree;
3. Group each 1-encoded symbol, {y,z}, as a sub-tree;
4. Reapply this to each subtree recursively until there is only one or two symbols left (that is when a 0 or 1 can only mean one thing).

For instance, given a string *S* = {*a, b, r, a, c, a, d, a, b, r, a*}, the alphabet is defined as lexicographic order of the characters in the sequence, which gives us Σ = {*a, b, c, d, r*}, however the final alphabet set should include the special character, $, as discussed before, hence Σ = {*a, b, c, d, r, $*}. So the alphabet {*a, b, c, d, r, $*} will be mapped to {0,0,0,1,1,1} which means for example, *a* will map to 0, and *d* will map to 1.The left sub-tree is created by taking just the *0-encoded* symbols {*a, b, c*} and then re-encoding them by dividing this new alphabet: {0,0,0,0,1,1,1,1}.This process goes on in a recursive manner. Hence the complete wavelet tree is shown in Figure 4.

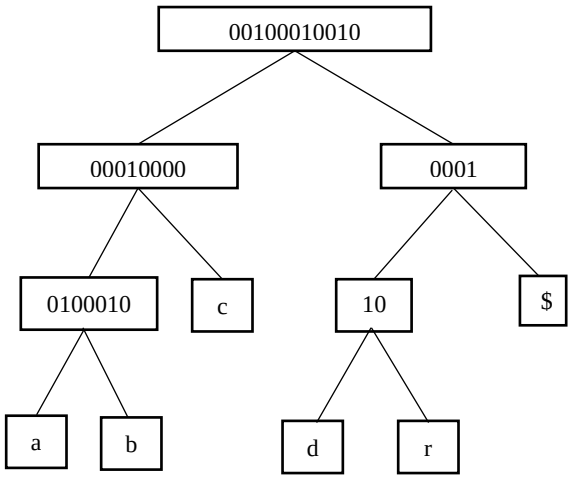

Figure 4 The Wavelet Tree

This data structure seem to be very promising given the recent results of the repeat finding algorithms which makes use the wavelet tree, hence this shows research directions for efficient data structures used in repeat finding algorithms. The wavelet tree can be constructed in O(n log σ) time using O(n log σ) bits of space, where σ = $|\sum|$. It is important to note that algorithms which make use of the wavelet tree have been shown to perform better than the algorithms which makes use the suffix array as shown in Figure 5.

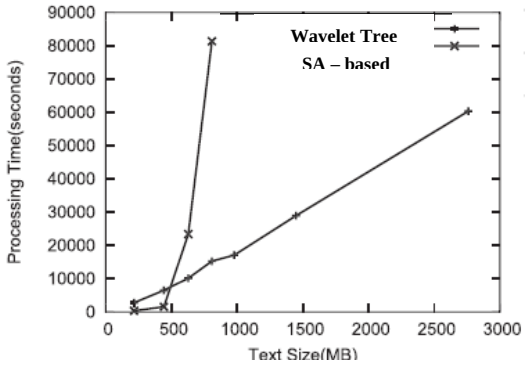

Figure 5. Results showing the SA-based versus the wavelet tree-based algorithm [8]

4. Techniques

This section discusses different techniques used in repeat finding algorithms. These techniques perform and assist the process of identification of repeats in a data structure.

4.1 Lempel-Ziv (LZ) Factorization

LZ factorization is an algorithm technique used in compression algorithms. It works by parsing the given sequence into distinct phases in a greedy manner [17]. For instance, given the string {*A A B A B B B A B*}. The LZ factorisation proceed by taking the first character *A* which is the shortest phrase we have ever seen then *AB* then *ABB* and so on in that manner {*A*| *AB*| *ABB*| *BAB*}. However there are many variations of this technique.

4.2 Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform is a compression algorithm which arranges characters in groups of identical characters into similar runs [14]. The powerful property of the Burrows-Wheeler Transform lies in the fact that it is reversible and produces a compressible transformed string. This is crucial given the major problem of space in DNA analysis when processing entire genomes. The BWT is defined as follows, given a string *S*, perform all the cyclic permutations of *S* and sort them in lexicographic order forming what is called the Burrows-Wheeler Matrix (BWM). The first columns are labelled as *F* and the last column as *L*, at this point only the *L* column is stored together with the index containing the full string. An example of the Burrows Wheeler Transformed for string *S* = *panama$* shown in Figure 6.

p	**a**	**n**	**a**	**m**	**a**	**$**
a	n	a	m	a	$	p
n	a	m	a	$	p	a
a	m	a	$	p	a	n
m	a	$	p	a	n	a
a	$	p	a	n	a	m
$	p	a	n	a	m	a

(a) Cyclic permutations of *S*

i	**F**						**L**
0	a	m	a	$	p	a	n
1	a	n	a	m	a	$	p
2	a	$	p	a	n	a	m
3	m	a	$	p	a	n	a
4	n	a	m	a	$	p	a
5	p	a	n	a	m	a	$
6	$	p	a	n	a	m	a

(b) Burrows-Wheeler Matrix of *S*

Figure 6 Burrows-Wheeler Matrix (b) computed from (a)

Finally, the *BWT(S)* = [*n,p,m,a,a,$,a*] and index 5 are the only values to be stored

In table 1, column $T_{suf\_array}[i]$, we can observe the appearance of *$*s in each row which leads us to another way of defining BWT(T) via the suffix array *SA(T)* as shown in equation (3). Let *BWT*[*i*] denote the character at 0-based offset *i* in BWT*(T)* and let *SA*[*i*] denote the suffix at 0-based offset *i* in *SA(T)*.The BWT is mathematically defined in Eq. (1).

$$BWT[i] = \begin{cases} T[SA[i]-1] & if\ SA[i] > 0 \\ \$ & if\ SA[i\} = 0 \end{cases} \quad (1)$$

Huffman coding or any other coding method can then be used to compress the Burrows-Wheeler transformed string by utilizing the BWT property which arranges characters in similar runs.

4.3 Backward Search

Backward search uses the BWT in a series of paired rank queries (which can be answered with a Wavelet Tree, for example). This search technique is used in FM-index to support fast pattern matching operations [13]. Backward search issues *p* pairs of rank queries, where *p* denotes the length of the pattern *P*. The paired rank queries are given in Eq. (2) and Eq. (3).

$$s = C\big[P[i]\big] + rank(s-1, P[i]+1 \quad (2)$$

$$e = C\big[P[i]\big] + rank(e, P[i]) \quad (3)$$

Where *s* denotes the start of the range and *e* is the end of the range. Initially *s*=1 and *e*=*N*. If at any stage *e* < *s*, then *P* does not exist in *S*. *C* is a lookup table containing the count of all symbols in our alphabet which are sorted lexicographically before *P*[*i*].

4.4 Longest Common Prefix (LCP) Array

An LCP array is an auxiliary array, usually generated from a suffix array or Burrows-Wheeler Transform [12]. The purpose of this array is to provide a starting point for candidate maximal repeats and to improve the search process. The array consists of lengths of the longest common prefixes for consecutive pairs of suffixes. When a suffix array is used with the LCP array, it can improve its search time to O(*m* + log *n*). Basically each LCP is a candidate maximal repeat ready to be confirmed by extending to both left and right preserving the repeat pattern. For instance, the LCP array provided in Table 1, the last column shows the lengths of the common prefixes

from *Tsuf_array* column. The process can be summarized in Figure 7.

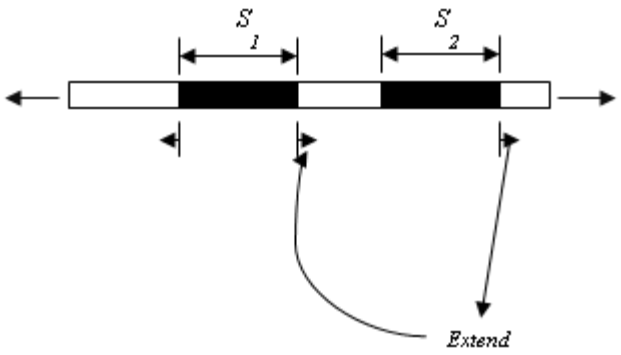

Figure 7 Ascertaining maximal repeats

Substrings S_1 and S_2 are repeats which are then re-compared by extending to the left to see if they can still match whilst observing the maximal repeat definition. The next step is to extend to the right if the left no longer matches and repeat the same process. This technique is commonly implemented in the current techniques to improve the search process in the identification of maximal repeats [8]. Manzini et al. [18] specified a measurement for indicating the difficulty of the suffix sorting by calculating the average LCP defined as in Eq. (4).

$$Average\ LCP = \left(\frac{1}{n-1}\right) \sum_{i=1}^{n-1} LCP(T[SA[i], n], T[SA[i+1, n]) \quad (4)$$

Which means if the average LCP is large then many characters would have to be analyzed to determine the relative order between the two suffixes.

5. Algorithms

Repeat finding algorithms can be broadly classified as *library-based* and *ab initio* [6]. Library based techniques works by comparing an input pattern to a set of known repeats in a database. An example of such tool would be RepeatMasker [19]. *Ab initio* based techniques find the repetitive sequences without using known references. Examples of *ab initio* based algorithms include: Reputer [20],[21], TRF[22], Mreps [23], ATRHunter [24]. The focus on the following algorithms is on *ab initio* tools.

Mreps is an implementation of an algorithm developed by Kolpakov et al. [23] for detecting all maximal repeats in a sequence *S* of length *n* in O(*n*) time. The algorithm is based on combinatorics and heuristics. Mreps algorithm runs in two stages: the first stage identifies all repetitions using the combinatorial algorithm and the second stage applies heuristics for filtering to get biologically relevant repeats. The strength of this program lies in its relative speed on low resolution, support for searching of fuzzy repeats and provides a range of parameters to control the execution. This control of execution is possible through the use of the resolution parameter which can be used to control the fuzziness of the repeats. The major limitations of this algorithm is that it runs very slowly on a complete human genome since it has high memory requirements. This is because it uses an index like structure based on the suffix tree on the first stage of the algorithm.

Stoye and Gusfield [25], proposed an algorithm which runs in O (*n* log *n*) time to produce all occurrences of tandem repeats and in O(*n*) space. The major advantage of this algorithm is that it runs relatively faster as the Mreps however it does not scale well if applied to large genomic sequences. This drawback on the algorithm comes from the fact that it is still based on the idea of suffix trees as its data structure like the Mreps. A suffix tree generally requires a lot of memory in its construction, it is therefore impractical to meet the memory requirements of this algorithm if the sequence is very long.

Abouelhoda et al. [26], improved the Stoye and Gusfield algorithm limitations of the suffix tree data structure by incorporating a new data structure called the suffix array. They replaced the bottom-up traversal technique on a suffix tree by their algorithm basing it on the enhanced suffix array. The algorithm only supports searching for tandem repeats leaving out interspersed repeats. It is interesting to note that reduction of the space consumption was done in the same time complexity as the one for suffix tree construction. The algorithm was implemented in a program called Vmatch. The identification of all tandem repeats is done in O(*n* log *n*) time and in 9*n* bytes of memory.

Sputnik program uses combinatorial approach and searches only for microsatellites [27]. The algorithm uses recursion and a sliding window approach by scanning through the sequence. The major limitation of this algorithm is that it only searches for smaller motif size, which is 1-5 base pairs (bp) but it is commonly used in many research projects because of its fast performance and less memory requirements however this is only possible because it searches for shorter repeats (1-5bp).

Since the introduction of the relatively successful suffix array, researchers switched focus towards improving the construction time of the suffix array which led to Ferragina and Manzini [18] proposing an algorithm for building a suffix array. Their new approach called “deep - shallow sorting” used a "deep" sorter for long common suffixes and "shallow" sorter for short suffixes. Most proposed

algorithms for constructing suffix arrays were inefficient when a sequence consists of a lot of repeats. Their algorithm managed to overcome this dichotomy. The algorithm runs in O(n log n) time in the worst case and uses O($n/\sqrt{}$log n) space in addition to the input text and the suffix array.

Another attempt for the SA (Suffix Array) construction algorithm was from Valimaki et al. [11]. This algorithm made use of a data structure called the Compressed Suffix Tree (CST) indexing technique for maximal repeat finding in the whole human genome. The algorithm runs in O(n log n log $|\sum|$) time and $nH_0 + 10n$ + O(n log $|\sum|$) bits. The main problem of the CST is that of high memory requirements and long running times in its construction. For instance, the whole human genome construction takes about four days, the final index (CST) occupies about 8.5 GB and the peak memory usage is 24 GB on a 32 GB-memory machine without including the time cost for the maximal repeat finding process. This completely shows that this indexing technique is not a practical solution to be considered or explored when considering repeat finding with entire human genomes.

Fast BWT algorithm works by block wise suffix sorting technique and was introduced by Karkkainen [28]. The algorithm is based on suffix arrays and BWT. Usually the BWT is constructed from a suffix array and the problem with this is that the suffix array takes a lot of memory space. In this algorithm they got rid of the full suffix array and decided to compute the BWT by going a small piece or block at a time on the suffix array (*BWT*[*i*] then compute *SA*[*i*] and so forth). The computation of BWT for a text of length n takes O(n log n+ vn) time using O(n log $n/\sqrt{}v$) space in addition to the text and the BWT. This method of SA construction was found to be 2-3 times faster with a construction rate of 1GB/hour better than the 300‑400MB/hour by Dementiev et. al.[29].

Another repeat finding algorithm improvement was done by Pokrzywa and Polanski called BWtrs algorithm [30]. The BWtrs is based on the indexing structure by Ferragina and Manzini. The algorithm has a limitation that it only operates on exact tandem maximal repeats. The BWtrs runs in O(n log n) time and the algorithm uses the FM-Index for its data structure. The FM index is a very space efficient data structure, it is a compressed full text index based on the BWT with some similarities to the SA. The major strength for this algorithm is that it uses an efficient index which requires less memory unlike the Mreps algorithm and other algorithms already discussed.

The algorithm by Fischer et al. [31] for pattern mining is based on the Compressed Suffix Array and LCP array. This algorithm can output all the repeats of a sequence by specifying an appropriate setting for parameters and input. The algorithm was implemented and tested on a 3.0 GHz CPU and 128 GB main memory. This algorithm was able to mine the whole *homo sapiens* genome in at least 39.8 hours (excluding the time for outputting the substrings) and a peak memory usage of 9.3 GB. The author claim that their method can be modified to output maximal repeats, however a clear algorithmic description and a full implementation for this customization is still being expected. The details of customization are not published yet hence it is not easy to benchmark with other maximal repeat finding algorithms however the running time was fairly good comparing with the ones at the time.

Recently Kulekci et al. [8] algorithm is among one of the fastest to the best of our knowledge. This algorithm is based on the new succinct and space efficient data structure called the wavelet tree. It also uses the suffix array to compute an auxiliary data structure, the LCP array which stores the longest common prefixes. This algorithm has been tested on a standard PC with 8GB internal memory and 2.8 GHz four-core Intel@CoreTM i7-860 chip with a running time of approximately 17 hours to complete searching for all the maximal repeats in the whole human genome. The results are shown in Table 2, where *ch.* represents chromosome and *W.H.G* represents Whole Human Genome. The text size column represents the size in megabytes for the chromosomes used and the whole human genome.

Table 2: Kulekci et al. [8]algorithm running times

	Text size(MB)	Construction time for SA (s)	Algorithm total time (s)
Ch. 1	215.47	250	2,784
Ch. 1 - 2	442.64	624	6,486
Ch. 1 - 3	628.41	1,162	10,119
Ch. 1 - 4	808.31	1,657	15,258
Ch. 1 - 5	977.77	18,446	17,069
Ch. 1 - 8	1,448.48	n/a	28,945
W.H.G	2,759.57	n/a	60,344

Its limitation is that it is still not acceptably fast to be usable and widely adopted for a “standard PC” use hence the need for continuous improvement to repeat finding algorithms in general.
In conclusion, all algorithms discussed above still leave a gap for producing a reasonable running time on a standard PC and this calls for continuous research to improve repeat finding algorithms.

6. Conclusions

In conclusion, we have shown the various key data structures to be considered for repeat finding algorithms however the most space efficient ones are the Wavelet tree and the FM-index. We have discussed classical indexing data structures like suffix trees and suffix arrays and their limitations for indexing complete human genomes due to high memory requirements. We have also presented several techniques to improve the search operations on the discussed data structures. Lastly we reviewed some prominent repeat finding algorithms and their limitations. It is also important to note that, even though parallel algorithms for BWT construction, Wavelet Construction exists, there is no one complete repeat finding algorithm which utilizes these on standard PC. The idea of parallelism can be explored to improve the performance issue of repeat finding algorithms in entire human genomes, given that todays' computers are well suited for parallel programs. As more and larger genomes are sequenced, efficiency and scalability will continue to become increasingly important.

References

[1] W. J. Ansorge, "Next-generation DNA sequencing techniques," *New biotechnology*, vol. 25, pp. 195-203, 2009.

[2] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, and B. Schmidt, "SHREC: a short-read error correction method," *Bioinformatics*, vol. 25, pp. 2157-2163, 2009.

[3] Y. Kodama, M. Shumway, and R. Leinonen, "The Sequence Read Archive: explosive growth of sequencing data," *Nucleic acids research*, vol. 40, pp. D54-D56, 2012.

[4] S. Wandelt, M. Bux, and U. Leser, "Trends in genome compression," *Current Bioinformatics*, vol. 9, pp. 315-326, 2014.

[5] T. Masombuka, C. de Ridder, and D. Kourie, "An investigation of software for minisatellite detection," in *Proceedings on the twenty-first PRASA symposium, F. Nicolls, Ed*, 2010.

[6] 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*, vol. 1, pp. 85-96, 2008.

[7] A. Fontana, "A hypothesis on the role of transposons," *Biosystems*, vol. 101, pp. 187-193, 2010.

[8] M. O. Kulekci, J. S. Vitter, and B. Xu, "Efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet tree," *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, vol. 9, pp. 421-429, 2012.

[9] P. Weiner, "Linear pattern matching algorithms," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, 1973, pp. 1-11.

[10] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, pp. 249-260, 1995.

[11] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit, "Engineering a compressed suffix tree implementation," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 2, 2009.

[12] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, pp. 935-948, 1993.

[13] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, 2000, pp. 390-398.

[14] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "An alphabet-friendly FM-index," in *String Processing and Information Retrieval*, 2004, pp. 150-160.

[16] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003, pp. 841-850.

[17] M. Crochemore, L. Ilie, and W. F. Smyth, "A simple algorithm for computing the Lempel Ziv factorization," in *Data Compression Conference*, 2008, pp. 482-488.

[18] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, pp. 33-50, 2004.

[19] A. Smit and P. Green, "RepeatMasker documentation," *Online:http://www.repeatmasker.org/webrepeatmaskerhelp. html*, 2003.

[20] S. Kurtz and C. Schleiermacher, "REPuter: fast computation of maximal repeats in complete genomes," *Bioinformatics*, vol. 15, pp. 426-427, 1999.

[21] 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*, vol. 29, pp. 4633-4642, 2001.

[22] G. Benson, "Tandem repeats finder: a program to analyze DNA sequences," *Nucleic acids research*, vol. 27, p. 573, 1999.

[23] R. Kolpakov, G. Bana, and G. Kucherov, "mreps: efficient and flexible detection of tandem repeats in DNA," *Nucleic acids research*, vol. 31, pp. 3672-3678, 2003.

[24] Y. Wexler, Z. Yakhini, Y. Kashi, and D. Geiger, "Finding approximate tandem repeats in genomic sequences," *Journal of Computational Biology*, vol. 12, pp. 928-942, 2005.

[25] J. Stoye and D. Gusfield, "Simple and flexible detection of contiguous repeats using a suffix tree," *Theoretical Computer Science*, vol. 270, pp. 843-856, 2002.

[26] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "The enhanced suffix array and its applications to genome analysis," in *Algorithms in Bioinformatics*, ed: Springer, 2002, pp. 449-463.

[27] C. Abajian, "Sputnik: DNA microsatellite repeat search utility," *Program available at: http://epressoftware. com/pages/sputnik. jsp*, 1994.

[28] J. Kärkkäinen, "Fast BWT in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, pp. 249-257, 2007.
[29] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders, "Better external memory suffix array construction," *Journal of Experimental Algorithmics (JEA)*, vol. 12, p. 3.4, 2008.
[30] R. Pokrzywa and A. Polanski, "BWtrs: a tool for searching for tandem repeats in DNA sequences based on the Burrows–Wheeler transform," *Genomics*, vol. 96, pp. 316-321, 2010.
[31] J. Fischer, V. Makinen, and N. Valimaki, "Space efficient string mining under frequency constraints," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, 2008, pp. 193-202.

First Author MSc. Computer Science. Research Assistant, Teaching Assistant and PhD student at Botswana International University of Science and Technology.

Second Author PhD. Computer Science, Dean for College of ICT at Botswana International University of Science and Technology.

Third Author PhD. Computer Science, Lecturer at Botswana International University of Science and Technology.