2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing

An adaptive suffix tree based algorithm for repeats recognition in a DNA sequence

Hongwei Huo, Xiaowu Wang
School of Computer Science and Technique
Xidian University
Xi'an, Shaanxi 710071, China
e-mail: hwhuo@mail.xidian.edu.cn

Vojislav Stojkovic
Department of Computer Science
Morgan State University
Baltimore, Maryland 21251, USA
e-mail: vojislav.stojkovic@morgan.edu

Abstract—**Many methods for repeats recognition are based on alignments. Their speed and time significantly limit their applications. This paper presents the fast Rep(eats)Seeker algorithm for repeats recognition based on the adaptive Ukkonen algorithm for a suffix tree construction. The RepSeeker algorithm uses the lowest frequency limit to maximize the extension of repeats. The adaptive improvements to the Ukkonen suffix tree construction are made to increase the efficiency of the RepSeeker algorithm. The node information required by the RepSeeker algorithm is added during the suffix tree construction. Because information in leaves and branch nodes are different, the RepSeeker algorithm directly obtains the needed information from nodes to find out the frequency and locate the positions of the substring. The improvement is noticeable for the repeats recognition. Comparisons between before and after improvements of the suffix tree construction show that improvements greatly reduce the running time of the RepSeeker algorithm without losing the accuracy.**

Keywords- Repeats recognition; adaptive suffix tree; Ukkonen algorithm; RepSeeker algorithm.

I. INTRODUCTION

A genome is an organism's complete set of DNA. Genomes vary widely in size: the smallest known genome for a free-living organism (a bacterium) contains about 600,000 DNA base pairs (bp), while mouse and human genomes have about 3 billions base pairs. A genome has a lot of repeats. For example, more than 50% of a human genome is various kinds of repeats [1]. Repeats recognition is a critical part of any analysis of a new sequence genome, because repeats drive genome evolution in various ways and pragmatic needs for thorough repeat mask prior to perform homology searches[9]. There are various kinds of repeats. Unfortunately, the functions of the most of repeats are not yet well understood and defined [2]. Current studies show that some repeats play an important role in the gene expression and transcription regulations [3]. The base composition of regions with repeats may lead to genome recombination which results in great changes of the genome. These repeats fall into different types, which contain from a few to several hundreds of base pairs, some up to ten thousands of base pairs. Some human genetic diseases such as the fragile-*X* chromosome syndrome, Huntington's disease, and Friedreich's ataxia are all related to irregularities of the length of repeats [4].

An important bioinformatics problem is how to recognize fast and represent efficiently repeats in a genome. There are two major approaches to solve the repeats recognition problems. RepeatMasker [5] uses an annotated library of repeats, which largely depends on similarity to consensus sequences for known repeat classes. This approach is not capable of treating new genomes whose libraries of regions are not available, because RepeatMasker does not build such libraries for new sequenced genomes. Moreover, the repeat libraries have to be manually compiled for any new genome because they are genome-specific. Do novo compilation of the RepeatMasker libraries remains a challenging bioinformatics problem [6]. The other approach, such as REPuter, is to extract all pairs of similar repeated regions with maximal length [7, 8]. It defines the repeats as a list of pairs of similar strings of maximal length. Both approaches tend to ignore the importance of repeat frequency/occurrence. In general, repeats appear more than twice times in the real biological sequences. For example, *Alu* repeats typically occur 10^6 in human genome. Transposons typically occur hundreds of thousands of times in complex genomes. Thus, it is more reasonable to incorporate frequencies of repeats.

This paper presents, according to the definition of elementary repeats [11], the fast RepSeeker algorithm for repeats identification based on the adaptive Ukkonen algorithm for a suffix tree construction[14]. The RepSeeker algorithm uses the lowest frequency limit to maximize the extension of repeats. In the same time, in order to increase the efficiency of the RepSeeker algorithm, the adaptive improvements to the Ukkonen suffix tree construction[10,12] are made. During the suffix tree construction, the leaf nodes are numbering and leaf's information is added to branch nodes. The information in leaves and branch nodes are different, thus the RepSeeker algorithm can obtain directly needed information from nodes to find out the frequency and locate the positions of a substring. The improvement is distinct for the repeats recognition. The running time is O(nNf) with a little extra cost in space, where n is the length of the sequence, N is the number of different repeats, and f is the average occurring frequency of a substring in the sliding windows. The RepSeeker algorithm uses nine sequences from NCBI to test the performance. The comparisons between before and after improvements to the suffix tree constriction have been made. The results show that the improvements greatly reduce the running time of the RepSeeker algorithm without losing the accuracy. The experimental results coincide with the theoretical expectations.

II. THE REPSEEKER ALGORITHM

A. *Definitions*

Definition 1 Let A be a repeat of sequence S, and let (A_1, A_2, …, A_m) be the sorted list of occurrences of A in S, where A_i is the ith occurrence of A in S, and m is the number of occurrences of A in S, and $m \geq 2$. Let B be a substring of A and let (B_1, B_2, …, B_k) be the sorted list of occurrences of B in S, where k is the number of occurrences of B in S.

If B starts at position s in A, then $B = A[s, s\text{-}1+|B|]$, $0 \leq s \leq |A|\text{-}|B|$.

B is a subrepeat of A, if $k = m$ and each B_i occurs with the same shift s in A_i for $i = 1, 2, .., m$.

Definition 2 A is a nontrivial substring of S if and only if A is a nonempty, proper, L-substring of S, $0<|A|<|S|$, where L is the minimal length of repeats.

Definition 3 A is an elementary repeat of S, if A is a nontrivial substring of S with the maximal length, A occurs in S at least F_m times, and every nontrivial substring of A is a subrepeat of A. F_m is a specified lowest frequency of occurrences of repeats.

Property 1 If A is an elementary repeat of S with the occurrences f, then each of all L-substrings of A has the occurrences f.

Proof. Immediately from definition 3. ■

The above property allows for removing most of the non-candidate repeats by computing the occurrences. The suffix tree is used to count the occurrences of its L-substring with a given threshold.

B. *Description of the RepSeeker algorithm*

The RepSeeker algorithm first finds all elementary repeats of the input sequence S and then outputs the sorted list of repeats. An element of the list of repeats is a pair, which represents the starting position and the ending position of a repeat copy in the input sequence. Hence, the problem of identifying elementary repeats can be converted into the problem of finding boundaries of repeats, the task of the RepSeeker algorithm is to check each position in the input sequence S and determine whether the position is a boundary of a repeat or not.

It is impractical to use the exhaustive algorithm to find elementary repeats because there are $O(n^2)$ substrings in S, and each substring of length m contains $O(m^2)$ substrings to be checked. So a suffix tree is constructed from the input sequence S to help count the frequencies. According to property 1, we know that if A is an elementary repeat of S, then all the L-substrings of A have the same frequency and at least F_m is a necessary condition. Therefore, the RepSeeker algorithm first computes the frequencies for all L-substrings and groups the same frequency at least F_m in a block in terms of the frequency array. Then, owning to the necessary condition, the RepSeeker algorithm checks these blocks and splits the blocks containing non-subrepeats. Then, the RepSeeker algorithm extends the obtained repeats by maximizing their length. Finally, the RepSeeker algorithm classifies them.

The RepSeeker algorithm consists of the following five steps.

Step 1：Compute the frequency of a substring in a sliding window.

We regard the lowest length L of a repeat as the width W of a sliding window and compute the occurrences of the substring of length W within the window. Every time the sliding window shifts for a position right. Let the frequency array be f and f[i] denotes the number of occurrences of the string of length L starting at position i, that is, the frequency of S[i, i+L-1]. Fig. 1 shows the values of frequency array f for the input sequence S.

Step 2：Finding the blocks with the same frequency

We can compute the starting position and the ending position of an L-substring with the occurrences at least F_m according to the frequency array f obtained in step 1. In the RepSeeker algorithm, we use l to keep track of the left boundary (the starting position) and r to keep track of the right boundary (the ending position). If two elements from l and r arrays, respectively, have the same position i in the input sequence S, then they denote a candidate repeat block with the same frequency, which is (l[i], r[i]). We can partition the sequence into same frequency blocks based on the values of the frequency array f. Specifically, for any position i, if $f[i] \neq f[i-1]$, then the position i is the starting position of the same frequency block; if $f[i] \neq f[i+1]$, then it illustrates that the last L-substring of the same frequency block starts at position i, that is, i+L-1 is the ending position of the block. Every time we have a starting position or an ending position, we insert it into the l or r array in order. After finishing the computation of l and r arrays, next is to pair the elements in l and r arrays. Thus, (l[i], r[j]) is the i-th same frequency block we have to find it.

Step 3：Checking subrepeats

The goal of this step is to check whether the blocks derived from the first two steps contain non-subrepeats L-substrings or not. If a block contains a substring which is not a subrepeat, it means that the block is not a repeat and we need to split it. Assume that a block D_i = S[starting position, ending position] with a frequency k. And all substrings with length L in the block D_i have the frequency m. If $k < m$, then it means that the block D_i contains a substring which is not a subrepeat. We scan the block D_i from left to right in turn to find the positions of the occurrences in S for all L-substrings in the block D_i and then compare the occurring position sequences for two L-substrings with the contiguous starting positions. If there is a corresponding pair from the two position sequences whose element is not contiguous, then we split D_i.

Step 4：Extending repeats

In order to maximize the length of a repeat, we merge the overlapped repeats if the resulting repeats have a frequency at least F_m, the threshold of frequency. The repeats with the higher frequency are still left and the repeats with the lower frequency are extended.

Let repeats A and B have an overlapping block, called C. The restrictions for merging are defined as follows:

$$merop(A, B) = \frac{\mathrm{C}}{\mathrm{A} \cup \mathrm{B}} \times 100\% \geq OP$$

and

$$fre\ (A \cup B) \geq F_m$$

where *merop* is the overlapping rate, *OP* is the specified minimal overlapping rate, $A \cup B$ is the result after merging A and B, and *fre* is the occurrences of the resulting after merging A and B.

Step 5：Classifying the repeats

In this step, we classify the repeats obtained in the step 4 and find all the copies of each of repeats.

C. The RepSeeker algorithm

The RepSeeker algorithm is as follows.

```
RepSeeker($S$, $L$, $F_m$)
Input: string $S$ of length $n$, minimum length $L$ of repeat, $F_m$
        minimum frequency
Output: classification list of all repeats in $S$ that appears at
        least $F_m$ times
1. Building a suffix tree for the string $S$
2. Computing frequency $f$ of the ith $L$-substring in $S$
3. Creating an array $D$ of repeat blocks in $S$
4. if $f[0] \geq F_m$ then add 0 in $l$ array
5. for $i \leftarrow 1$ to $n$-$L$-1 do
6.     if $f[i] \geq F_m$ then
7.         if $f[i] \neq f[i-1]$ then add $i$ in $l$ array
8.         if $f[i] \neq f[i+1]$ then add $i+L-1$ in $r$ array
9. for $i \leftarrow 0$ to $|l|$-1 do
10.    $D[i] \leftarrow (l[i], r[i])$
11. $D \leftarrow \{D[0], D[1], \ldots, D[k-1]\}$, sum $\leftarrow |D|$
12. Check($D$)
13. Extend($D$)
14. Classify($D$)
```

```
Check($D$: an array of blocks of equal frequency)
1. for $i \leftarrow 0$ to $sum$ -1 do
2.     if $f[l[i]] \neq f(D[i])$ then
3.         for $j \leftarrow 0$ to $|D[i]|$-$L$ do
4.             $P[j] \leftarrow$ sorted list of positions of occurrences of
                   j-th $L$-substring in $D[i]$
5.         for $k \leftarrow 0$ to $|D[i]$-$L$-1 do
6.             for $m \leftarrow 0$ to $f[l[i]]$-1 do
7.                 if $P[k+1, m] \neq P[k, m]+1$
8.                     then insert $l[i]+k+1$ into $l$
9.                         insert $l[i]+k+L-1$ into $r$
10.                        $sum \leftarrow sum+1$
```

```
Extend($D$: an array of elementary repeats)
1. for $i \leftarrow 1$ to $sum$-1 do
2.     if $merop(i, i+1) \geq OP$ and $fre \geq F_m$
3.         then $D[i] \leftarrow (l[i], r[i+1])$
4.             $i \leftarrow i$-1, $sum \leftarrow sum$-1
```

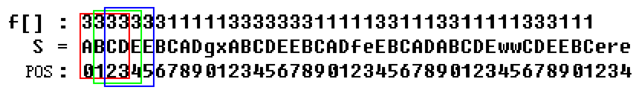
```
Classify($D$: an array of extended repeats)
1. for $i \leftarrow 0$ to $sum$-1 do
2.     $class[i] \leftarrow$ repeats equal to $D[i]$
```

The RepSeeker algorithm works as follows. Lines 1-2 compute the frequencies of all the L-substrings of S using the built suffix tree. Lines 4~8 find the starting and ending positions of all repetitive blocks with frequency at least F_m, which l and r record the starting and ending positions, respectively. Lines 9-10 pair the contiguous starting and ending positions from l and r arrays. The subroutine Check checks whether the block D_i contains a substring which is not a subrepeat. The subroutine Extend merges the repeats under some restrictions. The subroutine Classify sorts out the repeats. Section II-C gives a detailed description of the RepSeeker algorithm.

III. TIME AND SPACE COMPLEXITY ANALYSIS

Using the adaptive suffix tree, the lines 1-2 in the RepSeeker algorithm compute the frequency array in O(n) time, because a search for a substring of length L on a suffix tree is performed on a hash table. Lines 4~8 find the starting position and the ending position of each repeats with a frequency at least F_m in the sequence S in time O(n). Thus, the time complexity of the RepSeeker algorithm is largely determined by the time complexity of the Check procedure. The Check procedure contains three nested loops. The outmost loop of the RepSeeker algorithm scans all the blocks with frequency at least F_m. The intermediate loop of the RepSeeker algorithm can be done in at most O($\max_i|Di|$) times. The number of iterations in the innermost loop is the number of positions of the occurrences of the substrings in the current sliding window. The Extend procedure and the Classify procedure both have a linear running time. So - the RepSeeker algorithm runs in time O(nNf), where N is the number of different repeats and f is the occurring frequency of a string of the window on the average. Under normal circumstances, N and f are far less than n. So the RepSeeker algorithm can run very fast.

The RepSeeker algorithm maintains a hash table of size about $2n$ that holds the edge information to support the fast



```
f[] : 333333311111333333311111331113311111333111
  S = ABCDEEBCADgxABCDEEBCADfeEBCADABCDEwwCDEEBCere
 POS: 012345678901234567890123456789012345678901234
```

Figure 1. Frequency array f with $L = 4$, $F_m = 3$

search. Moreover, the RepSeeker algorithm use an array of size about n to store a node and extra array LL to store the extra information (the number of leaf nodes and their positions in S) associated with each node. Assume that the average extra space for a node is $|X|$ and the space for a leaf node is $|Y|$, the total space for the algorithm RepSeeker is about O(n(|X|+|Y|)). In order to obtain the gain in speed, the RepSeeker algorithm increases a little extra cost in space.

IV. Experimental results and analysis

A. Parameter setting

We coded the RepSeeker algorithm in the C++ programming language. The running time of the RepSeeker algorithm for each sequence is given in Table 1. We executed the RepSeeker program at an Intel 3GHz / 1GB memory desktop computer. The width L for the sliding window is 20, the least frequency restriction F_m is 3, and the overlapping rate OP is 25%.

B. Experimental results

We test the performance of the RepSeeker algorithm using some DNA sequences recommended by the National Center for Biotechnology Information (NCBI). The comparisons between the results include the number of repeats and the length of the maximal repeats identified by the RepSeeker algorithm under the parameter settings. The time T1 - before the adaption of the suffix tree and the time T2 - after the adaption of the suffix tree for the RepSeeker algorithm are given in Table 1.

C. Analysis of the experimental results

We analyzed the experimental results. For the running time, the adaption to the construction of a suffix tree provides a strong support to the performance of the RepSeeker algorithm. The improvement to the running time for the small sequences, for example, the sequence AC008583, is not significant and it is within 100 seconds whether the adaption has been made or not. However, for the larger sequence, such as HUMAN3M, the improvement cuts about five hours of the running time. The RepSeeker algorithm based on the adaptive suffix tree outperforms the algorithm before the adaption, as we can see in TABLE 1. The experimental results coincide with the theoretical expectations.

TABLE I. COMPARISONS ON THE PERFORMANCE OF THE REPSEEKER ALGORITHM

Name	Length	T1 (s)	T2 (s)	# reps	Maxlen
X14112	152261	125.473	10.297	45	299
AL593853	223276	231.506	20.172	739	147
AC008583	122493	72.323	6.937	78	41
CU210914	31433	16.528	0.921	8	37
NC_007410	366354	201.937	71.578	26	1547
DOG	1308479	10214.687	2880.469	3999	207
HUMAN3M	2900010	35007.182	15609.39	32759	513
NC_007436	949627	791.134	352.657	568	95
NC_008578	2443553	27323.901	9696.594	232	257

V. Conclusion

This paper presents the fast RepSeeker algorithm for repeats recognition based on the adaptive Ukkonen suffix tree construction algorithm. Adaptations to the construction of a suffix tree provide a strong support to the performance of the RepSeeker algorithm. The RepSeeker algorithm can recognize the repeats fast and locate starting and ending positions of repeats and then divide the repeats into categories. The RepSeeker algorithm makes full use of the adaptive suffix tree to compute the frequency, check subrepeats, merge repeats and classify repeats. The RepSeeker algorithm improves the running time greatly. The experimental results show that the RepSeeker algorithm is an efficient algorithm.

Acknowledgment

The work on this paper was supported by the National Natural Science Foundation of China, Grant No.69601003 and the National Young Natural Science Foundation of China, Grant No.60705004.

References

[1] Lander E.S, Linton L.M, Birren B, et al., “Initial Sequencing and Analysis of the Human Genome,” Nature, vol. 409, Feb. 2001, pp. 860-921.

[2] Lefebvre A, Lecroq T, Dauchel H. and Alexandre J., “FORRepeats: Detects Repeats on Entire Chromosomes and between Genomes,” Bioinformatics, vol. 19, no. 3, 2003, pp. 319-326, doi: 10.1093/bioinformatics/btf843

[3] Neil C. Jones and Pavel A. Pevzner, Introduction to Bioinformatics Algorithms, 1st ed., MIT Press: Cambridge, 2004, pp.311-337

[4] Huntington's Disease Collaborative Research Group, “A Novel Gene Containing a Trinucleotide Repeat That Is Expanded an Unstable on Huntington's Disease Chromosomes,” Cell, vol. 72, Mar. 1993, pp. 971-983.

[5] Smit A.F.A. and Green, http://repeatmasker.genome.washington.edu/

[6] Pevzner P. A, Tang H, and Tesler G, “De Novo Repeat Classification and Fragment Assembly,” Genome Res., 2004, vol. 14, pp. 1786-1796, doi:10.1101/gr.2395204

[7] Kurtz S. and Schleiermacher C, “REPuter: Fast Computation of Maximal Repeats in Complete Genomes,” Bioinformatics, vol. 15, no. 5, 1999, pp. 426–427.

[8] Kurtz S, Choudhuri J. V, Ohlebusch E, Schleiermacher C, Stoye J, and Giegerich R, “REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale,” Nucleic Acids Res., vol. 29, no.22, 2001, pp. 4633–4642.

[9] Alkes. L. Price, Neil C. Jones, and Pavel A. Pevzner, “De Novo Identification of Repeat Families in Large Genomes,” Bioinformatics, 2005, vol. 21, Suppl. i351-i358, doi:10.1093/bioinformatics/bti1018

[10] Esko Ukkonen, “On-line construction of suffix tree,” Algorithmica, vol. 14, no.3, 1995, pp.249-260

[11] Dan He, “Using Suffix Tree to Discover Complex Repetitive Patterns in DNA Sequences,” IEEE International Conference of the Engineering in Medicine and Biology Society (EMBS 06), IEEE Press, Aug. 2006, pp. 3474-3477

[12] R. Giegerich and S. Kurtz, “From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction,” Algorithmica, vol. 19, no.22, 1997, pp. 331-353.

[13] Gusfield D, Algorithms on Strings, Trees, and Sequences:Computer Science and Computational Biology, 1st ed, Cambridge University Press:Cambridge, 1997, pp87-168.

[14] Hongwei Huo and Vojislav Stojkovic, “A Suffix Tree Construction Algorithm for DNA Sequences,”Proceedings of the Seventh IEEE Symposium on Bioinformatics and Bioengineering (BIBE 07), IEEE Press, Oct. 2007, pp. 1178-1182, doi: 10.1109/BIBE.2007.4375711