International Journal of Computer Applications (0975 – 8887)
Volume 128 – No.16, October 2015

Efficient Algorithm for Extracting Complete Repeats from Biological Sequences

Munina Yusufu
School of Computer Science and Technology
Xinjiang Normal University
Urumqi, China

Gulina Yusufu
School of Education Science
Xinjiang Normal University
Urumqi, China

ABSTRACT
In this paper, an approach for efficiently extracting the repeating patterns in a biological sequence is proposed. A repeating pattern is a subsequence which appears more than once in a sequence, which is one of the most important features that can be used for revealing functional or evolutionary relationships in biological sequences. The algorithm does a rapid scan of the string to find repeating regions where the repeating substring has been marked using length, occurrence positions, and occurrence frequency. The algorithm execute in linear time and space independent of alphabet size. The algorithm also has the capability to restrict output complete repeats in which length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum. The algorithm outputs complete repeats, and can be extended or applied to other situations, for example computing maximal repeats, or finding common motifs in a set of biological sequences.

General Terms
Algorithms, Data Mining.

Keywords
Complete repeats, Biological sequence, Suffix array, Motif finding

1. INTRODUCTION
A large portion of DNA consists of repeating patterns of various sizes, from very small to very large. It has been estimated that repetitive DNA sequences comprise approximately 50% of the human genome [1, 2]. It is assumed that frequent or rare patterns may correspond to signals in biological sequences that imply functional or evolutionary relationships. For example, repetitive patterns in the promoter regions of functionally related genes may correspond to regulatory elements (e.g. transcription factor binding sites). Another example can be found in [3] where the authors applied repeat analysis to five distinct areas of computational biology: checking fragment assemblies, searching for low copy repeats related to human malformations, finding unique sequences, comparing gene structures and mapping of cDNA/EST.

The researchers point out that repeats in the evolutionary process can be used to help forming a new gene [4]. In some cases, repeating patterns have been implicated in human disease [5]. A repeating three nucleotide pattern on the human X chromosome is sometimes replicated incorrectly, causing the number of repeats to balloon from 50 to hundreds or thousands [6]. Individual with this defect suffer from Fragile-X mental retardation [6]. Several other diseases are also known to have association with the increases in the number of trinucleotide repeats, including Huntington's disease [7] and Friedreich's ataxia [8]. Therefore, in-depth research on biological sequence repeats will help to reveal the pathogenesis of certain genetic diseases, and provide an effective method for the diagnosis and treatment of these genetic diseases. Finding common substrings in a set of strings is also important. For example, motifs or short strings common to protein sequences are assumed to represent a specific property of the sequences [9, 10].

Given the importance of repeating patterns and the massive amount of exponentially growing DNA sequence data, it is a daunting challenge to develop efficient methods and algorithms for finding repeats. In this paper, we investigate mathematical and algorithmic aspects of repeats in biological sequences. Considering the importance of the definition of repeats, we introduce a definition of repeats that considers both length and frequency of occurrences. We then propose an efficient approach for the extraction of the repeating patterns in biological objects, which does a rapid scan of the string, to find repeating regions where the repeating substring has been marked using length, occurrence positions, and occurrence frequency. The algorithm execute in linear time and space independent of alphabet size. The algorithm also has the capability to restrict output complete repeats in which length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum.

In Section 2 we describe previous related work. In Section 3 we introduce some definitions and preprocessing, we then analyze various features of repeating patterns in biological sequences. We use an example to show the basic idea of the proposed method for finding complete repeats and propose a linear time and space algorithm. Section 4 summarizes the results and outlines the future work.

2. PREVIOUS WORK
The first step of designing efficient algorithm for locating repeats is giving the accurate definition of the repeats. *Repetitive patterns*, or *repeats* for short, usually refer to the sequences that occur repeatedly in biological sequences.

Based on the reassociation rate, DNA sequences are divided into three classes:

1. **Highly repetitive:** About 10-15% of mammalian DNA reassociates very rapidly. This class includes *tandem repeats*. The copies lie adjacent to each other, either directly or inverted. There are three kinds of tandem repeats, which include: satellites, minisatellites (variable number tandem repeat), and microsatellites (short tandem repeat). Tandem repeat is also called *repetition* in some literatures.
2. Moderately repetitive: Roughly 25-40% of mammalian DNA reassociates at an intermediate rate. This class includes

interspersed repeats.

3. Single copy genes (or very low copy number genes, also called Genomic island): This class accounts for 50-60% of mammalian DNA.

The automated detection of such repeats in biological sequences is no trivial task. The first step in the identification may give the proper definition of the repeats. Biologically meaningful definition of repeats must consider the length and frequency of repetitive substrings. Some studies suggest that accurately define repeats is not easy. Some methods can only find short repeats or tandem repeats. However, they are unable to find long and interspersed repeats.

Some methods require an annotated library of repeats for repeat identification, for example RepeatMasker [11]. This library is largely dependent on the similarity of homologous sequences, so the method is limiting. In [12] an algorithm is proposed to find all the maximal repeated pairs in a string. The main data structure is suffix tree, and the time complexity is $O(\alpha n+q)$, where q is the number of pairs output.

REPuter program [3,13] overcomes the limitation of input sequence size and is the suffix tree based algorithm to identify repeated sequences, while the output is again a list of pairs of similar strings of maximal length. A limitation to this method is the size of the genomic target due to the workload. MUMmer program [14] is an alignment and comparison program for a long DNA sequence, and it can also be used to identify repeat patterns. TRF (Tandem Repeats Finder) [15] designed by Beason is the most influential method for tandem repeats discovery. But there is a restriction on the size of the tandem repeats period.

Algorithms in [16] uses either the suffix trees of both a string and its reversed string, or alternatively the suffix arrays of both, to compute all the complete NE repeats (Nonextendible repeats) in a string in $\theta(n)$ time. Algorithm in [17] describes a suffix array-based linear-time algorithm to compute all substring equivalence classes in a string --- that is, complete NE repeats together with all substrings that are unique in a string in $O(n)$. In [18], several fast algorithms for computing different kinds of maximal repeats under some restrictions were proposed, which are also suffix array-based linear-time algorithms. In practice algorithms in [18] uses substantially less time and space than either of [16,17].

3. OUR ALGORITHM

3.1 Definitions

We use standard concepts and notation about strings. The set Σ denotes a nonempty alphabet of symbols, and the alphabet size $\sigma = |\Sigma|$. A *string* S is an ordered sequence of elements drawn from Σ. In this paper, we represent S as an array $S[0..n-1]$ of $n \geq 0$ letters, where $n = |S|$ is called the length of the string, while the empty string is denoted by ε. We say that S has n elements $S[0], S[2],..., S[n-1]$, and has n *positions* while position 0 is at *leftmost* side of S and position n-1 is at *rightmost* side of S. Corresponding to any pair of integers i and j that satisfy $0 \leq i \leq j \leq n-1$, we define a *substring* $S[i..j]$ of S as follows: $S[i..j] = S[i]S[i+1]...S[j]$. We say that $S[i..j]$ occurs at position i of S and that it has length j-i+1.

In particular, $S[0, i]$ is called a prefix of S that ends at position i and $S[i, n-1]$ is called a suffix of S that begins at position i. Let $\mathrm{pref}_i(S) = S[0, i]$ and $\mathrm{suff}_i(S) = S[i, n-1]$ denote the prefix and suffix of S, respectively. Omitting the subscripts, we let pref(S) and suff(S) denote the set of all non-empty prefixes and suffixes of S, respectively.

Intuitively, a repeat is a collection of repeating substrings, not necessarily adjacent. In a simple way, a repeat can be described as $R = ((i_1, j_1), (i_2, j_2))$ as in Figure 1, where (i_1, j_1) is the first starting and ending positions of repeating substring ATGC, and (i_2, j_2) is the second one.

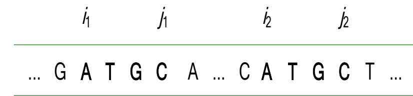

Fig. 1 Repeat R = $((i_1, j_1), (i_2, j_2))$ = ATGC

If repeating substring appears many times, above definition is inefficient; we also discussed that meaningful definition of repeats must consider the length and occurrence frequency. We notice that the repeating substrings of a repeat are the same length. Therefore, more formally, a repeat in S can be defined as a tuple

$$R_{S,u} = (p;\ i_1,\ i_2,...,i_e),$$

where $e \geq 2,\ 0 \leq i_1 < i_2 < ... < i_e \leq n-1$; the repeating substring

$$u = S[i_1..i_1+p-1] = S[i_2..i_2+p-1] = ... = S[i_e..i_e+p-1].$$

We call u the *generator*, p the *period* (the *length*), e the *exponent*, and $i_1, i_2, ..., i_e$ the occurrence positions of $R_{S,u}$. Note that it may happen, for some $j \in 1..e-1$, that $i_{j+1}-i_j = p$ or that $i_{j+1}-i_j < p$ -- that is, the substrings of a repeat may be *adjacent* or even *overlap*.

We say that $R_{S,u}$ is *complete* if for every

$$i \in 0..n-1 \text{ and } i \notin (p;\ i_1,\ i_2,...,i_e),$$

we are assured that $S[p;\ i..i+p-1] \neq u$. We say that $R_{S,u}$ is *left-extendible* (LE) if

$$(p+1;\ i_1-1,\ i_2-1,...,i_e-1)$$

is a repeat; in this case, $(p+1;\ i_1-1,\ i_2-1,...,i_e-1)$ is a repeat whose suffixes of length p are specified by $R_{S,u}$. Similarly, $R_{S,u}$ is *right-extendible* (RE) if

$$(p+1;\ i_1,\ i_2,...,i_e)$$

is a repeat; in this case, $(p+1;\ i_1,\ i_2,...,i_e)$ is a repeat whose prefixes of length p are specified by $R_{S,u}$. If $R_{S,u}$ is neither LE nor RE, we say that it is *nonextendible* (NE).

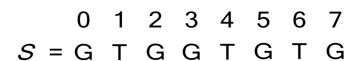

Fig. 2 DNA Sequence S = GTGGTGTG

In Figure 2, there are several repeats in the DNA sequence S; for example, $R_{S,GTG} = (3;\ 0,\ 3,\ 5)$ is NE repeat; while $R_{S,TG} = (2;\ 1,\ 4,\ 6)$ is LE repeat , and $R_{S,GT} = (2;\ 0,\ 3,\ 5)$ is RE repeat. All of them are complete repeats.

3.2 Biological Sequence

Since a major application of the problem is computational molecular biology, we briefly introduce the biological sequence here. DNA sequence is a string containing characters A, C, G and T, which means that Σ = {A, T, C, G} for DNA; Σ = {A,C,G,U} for RNA, and for the protein sequence, Σ = { A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y}. As stated in [19]: *The development of fast methods for sequencing genes and proteins is one of the most significant technological achievements of recent times. This has enabled the creation of large databases which can be processed by abstracting sequences of nucleic acids (DNA,*

RNA) and amino acids (proteins) into strings of characters. Finding all repeats in a biological sequence is equivalent to the problem of finding all repeats in an arbitrary string containing those characters.

3.3 Preprocessing

Now we introduce certain preprocessing in our algorithm, namely the suffix array and longest common prefix array, which are usually used together, become a central data structure in computational molecular biology. The *suffix array* (SA) is an array SA[0..n-1] in which SA[j] = i iff suffix i is the jth in lexicographical order among all the suffixes of S. The suffix array of a string of length n over an integer alphabet can be computed in O(n) time [20].

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	A	T	G	C	A	A	T	G	C	C	V	G	G	C	A	T	T	G	C	A	T	V
SA	4	0	5	14	19	3	13	18	8	9	2	12	17	7	11	1	16	6	15	20	10	21
LCP	-1	1	4	2	2	0	2	3	1	1	0	3	4	2	1	0	4	3	1	1	0	0

Fig 3: SA and LCP arrays of string *S*

Let us denote the length of the *longest common prefix* of suffixes i and j by $lcp(i, j)$. Then, the LCP array contains the lengths of the longest common prefixes between successive suffixes of SA. That is, LCP[i] = lcp(SA[i-1], SA[i]) for $0 < i \leq n-1$. Given S and SA, LCP can also be computed in $\theta(n)$ time [21]. An example of protein sequence as string S together with SA and LCP arrays are shown in Figure 3.

3.4 Design of the Algorithm

As we discussed in Section 2, the automated detection of repeats in biological sequences is no trivial task. The first step of designing efficient algorithm for locating repeats is giving the accurate definition of the repeats. In Section 3.1, we give the definition of a repeat as $R_{S,u} = (p;\ i_1,i_2,...,i_e)$, which consider the *generator u*, the *period p*, the *exponent e*, and the occurrence positions i_1, i_2, ..., i_e of $R_{S,u}$. Since biologically meaningful definition of repeats must consider the length and frequency of repetitive substrings, here p is the length of the repeat, and the frequency can be derived from the occurrence positions $i_1,i_2,...,i_e$. Since all these information are unknown, we need to design an effective approach to locate the positions of the repeating substrings and the boundaries.

In this section we introduce the basic methodology and design process of our algorithms, illustrated with the example S = ATGCAATGCCVGGCATTGCATV.

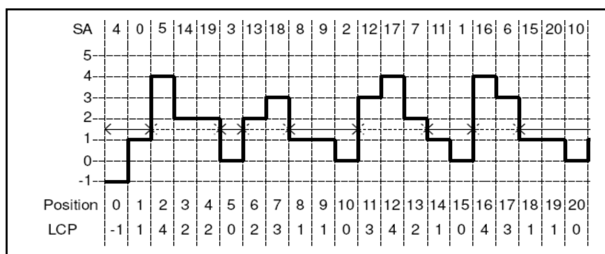

Fig 4: graphical representation of the SA and LCP values of S

Figure 4 gives a graphical representation of the SA and LCP values of S. For simplifying the algorithm, we suppose that LCP[0]=-1, LCP[n]=-1.

- At each position i = 0, 1,…, n-1, the corresponding SA[i] values and LCP[i] values are shown.
- The vertical lines in the figure identify increases and decreases in the LCP values p as i ranges from 0 to n-1.
- The LCP values allow the detection of repeats:
- When LCP[i] < LCP[i+1] : open a potential repeat occurring at the positions of SA[i], SA[i+1];
- When LCP[i] = LCP[i+1] : sustain a potential repeat occurring at the positions of SA[i], SA[i+1];
- When LCP[i] > LCP[i+1]: close the repeat previously created.
- For example, when LCP[1] < LCP[2], open a potential repeat occurring at the positions of SA[1], SA[2]; that is, the repeat substring is ATGC occurring at positions 0 and 5.
- When LCP[3] = LCP[4], sustain a potential repeat occurring at the positions of SA[3], SA[4]; that is, sustain a potential repeat with repeating substring of AT at positions 14,19.
- When LCP[4] > LCP[5], close the repeat. That means there is no repeat at the positions SA[4] = 19, SA[5] = 3.
- Each horizontal line can be specified by a tuple (p; i, j) where i is the left endpoint of the line, j is the right endpoint of the line at height p; then (p; i, j) specifies a repeat in S, moreover, a complete repeat. For example, there are 3 repeats of length 4, that is, (4; 1, 2), (4; 11, 12), and (4; 15, 16), corresponding to (4; 0, 5) = ATGC, (4; 12, 17) = GCAT, and (4; 1, 16) = TGCA, respectively.
- The notation (p; i, j) used here, that identifies a range $i..j$ in SA provides a mechanism for compressing the reporting of repeats; in terms of positions in S, for example, the repeat (p; i, j) = (4; 1, 2) would need to be reported as (4; 0, 5). See below for the explanation.
- For example, according to definition in Section 3.1, repeat $R_{S,ATGC}$ will be (4; 0, 5), but now it can be specified as (4;1,2), since (4; SA[1], SA[2]) = (4; 0, 5) = ATGC; same as $R_{S,AT}$ = (4;0,5,14,19), can be specified as (2;1,4), since (2;1,4) = (2; SA[1], SA[2], SA[3], SA[4]) = (4;0,5,14,19) = AT.
- Some horizontal lines occur in a specific *segment* (p; i, j), $p = p'$, p'+1,..,q; that is, with the same range $i..j$, but with different heights p. In such cases, the peak tuple (q; i, j) represents a longest repeat for positions i and j:

  S[SA[i] .. SA[i] + q-1]

  S[SA[i+1] .. SA[i+1] + q-1]

  ……………….

  S[SA[j] .. SA[j] + q-1]

For example, the peak tuple (4;1,2) identifies the longest repeat (4; SA[1],SA[2]) = (4;0,5) = ATGC; the other tuples in this segment is (3; SA[1],SA[2]) = (3;0,5) = ATG, so (3;1,2) corresponding to ATG. We observe that ATG is RE repeats, but ATGC is NRE, because it is at the peak (it can not be right extended).

The following lemma express more formally the observations made above.

Lemma 1 (Completeness) Suppose there is a tuple (p; i, j) as

defined above. Let u = S[SA[i]..SA[i] + p-1]. Then (p; i, j) identifies a complete repeat.

Proof Since p is the longest common prefix of the suffixes SA[i], SA[i + 1], .., SA[j], and $i < j$, therefore the prefixes of length p of these suffixes certainly identify a repeat (p; SA[i], SA[i + 1], .., SA[j]) of S. If (p; i, j) is not a complete repeat, then there must exist k, such that $S[k..k + p-1] = u$, for $k \in \{1, 2,\ldots, n\} \wedge k \notin \{SA[i], SA[i + 1],..,SA[j]\}$. But since for some t, SA[t] = k, it follows that t, $k \in \{i, i + 1, .., j\}$; that is, $k \in \{SA[i], SA[i + 1], .., SA[j]\}$, so $R_{S,u}$ = (p; SA[i], SA[i + 1], .., SA[j]) must be a complete repeat of S.

Occurrence frequency of the repeating patterns plays the important roles in bioinformatics studies. We now analyze some features relating with the occurrence frequency of the repeating substring.

In the form of $R_{S,u} = (p; i_1, i_2,...,i_e)$, we could easily determine the occurrence frequency of the repeating substring u in the repeat $R_{S,u}$, which equals to the e, the number of the repeating substring u appeared. For example, for $R_{S,AT}$ = (4; 0, 5, 14, 19), the occurrence frequency of the repeating substring u = AT is 4. The occurrence frequency of the repeating substring u is simply called the occurrence frequency of the repeat $R_{S,u}$ and written in Fre($R_{S,u}$). For the form of $R_{S,u}$ = (p; i, j), then the occurrence frequency of the repeat Fre($R_{S,u}$) equals to j-i+1. If Fre($R_{S,u}$) is equal to or higher than predefined threshold F_{min}, $R_{S,u}$ is called high-frequency repeat, while $F_{min} = 2$ is a minimal frequency for a repeat. Then we have the following lemmas that express more features of the repeats.

Lemma 2 If $Fre(R_{S,u}) \geq \lambda$, where $\lambda \geq 2$, then $Fre(pref(u)) \geq \lambda$ and $Fre(suff(u)) \geq \lambda$, where pref($u$) and suff($u$) denote the set of all non-empty prefixes and suffixes of u as defined in Section 3.1.

Lemma 3 For a repeat $R_{S,u}$ and pref(u), the following inequation holds.

$$Fre(R_{S,u}) \leq \min(Fre(pref(u)).$$

An example can be drawn from Figure 2. According to Lemma 1, there are 4 complete repeats in the third segment of the figure, which are:

$$Fre(R_{S,GCAT}) = 2, \text{ since } R_{S,GCAT} = (p; i, j) = (4; 11, 12),$$

$$Fre(R_{S,GCA}) = 3, \text{ since } R_{S,GCA} = (p; i, j) = (3; 10, 12),$$

$$Fre(R_{S,GC}) = 4, \text{ since } R_{S,GC} = (p; i, j) = (2; 10, 13),$$

$$Fre(R_{S,G}) = 5, \text{ since } R_{S,GCA} = (p; i, j) = (1; 10, 14);$$

According to Lemma 2 and 3, we have:

$$Fre(R_{S,GCAT}) \geq \lambda \ (\lambda \geq 2) \Rightarrow$$

$$Fre(R_{S,G}) \geq \lambda, Fre(R_{S,GC}) \geq \lambda, Fre(R_{S,GCA}) \geq \lambda;$$

$$\text{And } Fre(R_{S,GCAT}) \leq \min(Fre(R_{S,GCA}), Fre(R_{S,GC}), Fre(R_{S,G}))$$

For a repeat $R_{S,u}$ and pref(u), according to Lemma 2, all pref(u) are repeats too, so we have Lemma 4 as follow:

Lemma 4 The peak tuple (q; i, j) represents a longest repeat for positions i and j, and also the NRE repeat.

Lemma 1,2,3 and 4, together with the graphical presentation exemplified in Figure 4, motivates representation of repeats by (p; i, j) where $i..j$ is a range in SA. This is the approach adopted by our algorithm.

If we want to locate all complete repeats (p; i, j) where $i..j$ is a range in SA, we need to scan LCP several times. In order to compute all complete repeats by only scanning LCP once, we design a stack STALOCH where every stack element has the form: (*location, height*). Here *location* represent the corresponding initial position (or left boundary), *height* represent the corresponding LCP values.

Lemma 1 tells us that in order to specify a complete repeat in S, it is necessary only to output a triple (p; i, j) -- provided the suffix array of S is available. We add frequency Fre for the applications that require the frequency Fre $>=F_{min}$, here F_{min} is a user defined minimal frequency for a repeat. We as well give the restriction to the period of repeats as P_{min}, so the trivial repeats will not be output.

The algorithm is presented in Figure 5.

Algorithm Complete Repeat Finding algorithm (CRFinder)

```
CRFinder(S, P_min, F_min)
Input: string S of length n, requested minimum repeat length
 threshold P_min and minimum frequency F_min
Output: all complete repeats (p; i, j) in S that appear at least
 F_min times and period p ≥ P_min
Preprocessing: Computer SA[i] and LCP[i] (0≤i≤n-1) of
string S; let LCP[0] = -1, LCP[n] = -1
1. k=0; push (STALOCH; 0, 0)
 //(location, height) are stored in stack STALOCH, initial values set to 0
2. while (k < n-1)
3.     while (LCP[k] <= LCP[k+1]) do
         //when LCP[k]<LCP[k+1]: open a potential repeat occurring
4.         if (STALOCH[top].height < LCP[k+1]) then
              push (STALOCH; k, LCP[k+1])
5.         k++
6.     while (LCP[k] > LCP[k+1]) do
        //when LCP[k] > LCP[k+1]: close the repeat previously created
7.         j=k
8.         k++
9.         pop(STALOCH) to (i,h)
10.        while (STALOCH[top].height > LCP[k]) do
11.            for p = h down to STALOCH[top].height +1 do
12.                Check(p, i, j)
            //check period and occurrence frequency satisfy the requirements
13.            pop(STALOCH) to (i,h)
14.        for p = h downto LCP[k] +1 do
15.            Check(p, i, j)
16.        if (STALOCH[top].height < LCP[k]) then
               push (STALOCH; i, LCP[k])
17.        while (LCP[k] = LCP[k+1] ) do
            //when LCP[k] = LCP[k+1]: sustain a potential repeat occurring
18.            k++

Function Check(p, i, j)
1. f = j-i+1
2.     if f >=F_min and p >= P_min then
3.         output (p; i, j; f)
```

Fig 5: Complete Repeat Finding algorithm (CRFinder)

To locate all complete repeats in S, we build a suffix array and LCP array for S in linear time (see e.g. [20] and [21] and references therein for details about suffix array and LCP array and their linear time construction), and then our algorithm CRFinder perform a single left-to-right scan of LCP, so the total time complexity is linear independent of alphabet size. Each complete repeat can be specified in constant space (about 9 bytes), so the space complexity is also linear. A few execution steps of algorithm CRFinder are showed by using a

part of string *S* (see Figure 3) in Figure 6.Although our algorithm CRFinder aimed to locate all complete repeats in a biological sequence, it is trivial to extend CRFinder to find all the NRE repeats as Lemma 4 pointed "The peak tuple (q; i, j) represents a longest repeat for positions i and j, and also the NRE repeat", it only need to push into the stack with peak tuple (q; i, j) rather than all the tuples.

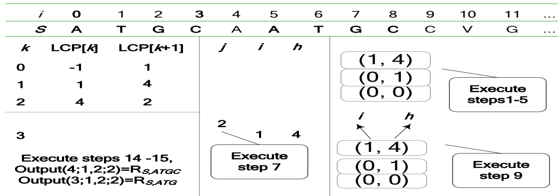

Fig 6: A few execution steps of algorithm CRFinder are showed by using a part of string *S* (see Figure 3); we let F_{min} =2 and P_{min} =2. The repeats $R_{S,ATGC}$ and $R_{S,ATG}$ are output.

4. CONCLUSION

Repetitive patterns have a great importance in a variety of applications not only in computational molecular biology (including tandem repeats analysis, motif finding, etc.), but also in data mining [22], and data compression [23].

In this paper we discussed repetitive patterns problem. A novel approach of finding complete repeats in biological sequence is proposed, which leads to a natural definition of repeat length and boundaries, also the frequency can be derived from the boundaries. The algorithm allows for listing all occurrences of complete repeats in a given string of length n in O(n) time. The algorithm also has the capability to restrict output by using the user-specified minimum and can be extended or applied to other situations.

5. ACKNOWLEDGMENTS

This work is funded by the Natural Science Foundation of Xinjiang Uyghur Autonomous Region (No.2012211A056).

6. REFERENCES

[1] Lander E.S, Linton L.M, Birren B, et al.. Initial Sequencing and Analysis of the Human Genome, Nature, 2001, 409(6822): 860-921.

[2] Treangen TJ, Salzberg SL. Repetitive DNA and next-generation sequencing: Computational challenges and solutions, Nature Reviews Genetics, 2011, 13 (1): 36-46.

[3] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, Robert Giegerich. REPuter: The manifold applications of repeat analysis on a genomic scale, Nucleic Acids Research, 2001, 29(22):4633-4642.

[4] Makalowski W. Not junk after all, Science, 2003, 300(5623): 1246-1247.

[5] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. Nature, 2004, 431: 931-945.

[6] Verkerk A., Pieretti M., Sutcliffe J., Fu Y., Kul D., Pizzuti A., Refiner O., et al.. Identification of gene (FMR-1) containing a CGG repeat coincident with a breakpoint cluster region exhibiting length variation in fragile X syndrome, Cell, 1991, 65: 905-914.

[7] Huntington's Disease Collaborative Research Group. A novel gene containing a trinucleotide repeat that is expanded an unstable on Huntington's disease chromosomes, Cell, 1993, 72: 971-983.

[8] Campuzano V., Montermini L., Molto M.D., Pianese L. Cossee M, et al.. Friedreich's ataxia: autosomal recessive disease caused by an intronic GAA triplet repeat expansion, Science, 1996, 271(5254):1423-1427.

[9] E. Eskin, P. A. Pevzner. Finding Composite Regulatory Patterns in DNA Sequences. Bioinformatics, 2002, 18 Suppl 1:S354-363.

[10] Sagot, MF. Spelling Approximate Repeated or Common Motifs Using a Suffix Tree. Lecture Notes in Computer Science, 1998, 1380:111-127.

[11] A.F.A. Smit and P. Green. REPEATMASKER. Available at http://www.repeatmasker.org/

[12] Dan Gusfield. Algorithms on Strings, Trees and Sequences, Cambridge University Press, 1997.

[13] Stefan Kurtz, Chris Schleiermacher. REPuter: Fast computation of maximal repeats in complete genomes, Bioinformatics, 1999, 15(5):426-427.

[14] A. L. Delcher et al. Alignment of whole genomes, Nucleic Acids Research, 1999, 27:2369-2376.

[15] Gary Benson. Tandem repeats finder: A program to analyze DNA sequences, Nucleic Acids Research, 1999, 27(2):573-580.

[16] Frantisek Franek, William F. Smyth, Yudong Tang. Computing all repeats using suffix arrays, Journal of Automata, Languages and Combinatorics, 2003, 8(4): 579-591.

[17] Kaziyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Efficient computation of substring equivalence classes with suffix arrays, Proc. of 18th CPM, 2007, 340-351.

[18] Simon J. Puglisi, W. F. Smyth, Munina Yusufu. Fast, Practical Algorithms for Computing All the Repeats in a String, Mathematics in Computer Science, 2010, 3(4):371-496.

[19] Albert A. Conti, Tom Van Court, Martin C. Herbordt. Processing Repetitive Sequence Structures with Mismatches at Streaming Rate. Lecture Notes in Computer Science, 2004, 3203:1080-1083.

[20] Juha Karkkainen, Peter Sanders. Simple linear work suffix array construction, Proc. of 30th ICALP, LNCS 2719, 2003, 943-955.

[21] Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications, Proc. of 12th CPM, LNCS 2089, 2001, 181-192.

[22] Nizar R. Mabroukeh, C. I. Ezeife. A Taxonomy of Sequential Pattern Mining Algorithms. ACM Computing Surveys, 2010, 43(1):3:1-3:41.

[23] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, W.F. Smyth, German Tischler, Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. ACM Computing Surveys, 2012, 45(1):5:1-5:17.