A Multiple Genome Sequence Matching Based on Skipping Tree

Zihuan Xu, Kewei Cheng, Yi Ding, Ziqiang Tian, and Hui Zhao

***Abstract*—In this paper, a new algorithm, skipping suffix algorithm based on a new encoded mode for genome sequence aimed at accelerating multiple genome sequence matching are proposed. By introducing binary coding, the efficiency of gene sequence alignment gets improved obviously. Besides, we decide the maximal bits to skip by constructing skipping tree. A contrastive evaluation of the computational efficiency of KMP algorithm, suffix array and skipping suffix algorithm shows that preprocess of skipping suffix algorithm is more than 12 times speedup than that of suffix array. Moreover, multiple genome sequence matching based on suffix array is more than 50 times speedup than that of KMP. In a word, skipping suffix algorithm strike balance between preprocess and search successfully which better help it fit into large-scale genetic data matching.**

***Index Terms*—Bioinformatics, skipping tree, bit manipulation, binary search.**

I. Introduction

Thanks to Human Genome Project, DNA sequencing technology has developed at unprecedented speed within the latest decade. As more and more biological data are being created every day, there is a pressing need to extract useful information from a sea of biological data efficiently [1], which usually begins with identifying particular DNA sequence which encoding specific protein in order to determine the biological property of certain DNA sequence. Thus, it raises a significant problem, comparison of similarity between two DNA sequences.

Traditionally, DNA sequences will be abstracted as a string which incorporates only a collection of four letters, A, G, C, T. Also there are many algorithms related to genome sequence matching problem [2], [3]. Meanwhile, the suffix tree [4], as a significant data structures in the field of string matching, has been used widely in gene sequence alignment. Although suffix tree has numerous advantages, like powerful indexing capability, taking hugeness of genes on a chromosome into consideration, even there is a new structure named suffix array [5] based on it, it still consumes intolerable time in constructing [6] the suffix tree while doing preprocess on reference sequences.

Also, there are some methods which do some preprocess on query sequences like KMP Algorithm [8], Boyer and Moore Algorithm [9], and Sunday Algorithm [10]. Nowadays, there are many algorithms based on the same thought that they collect the information of query sequences and then use it to determine how to move the window of query sequence under specific condition [11]. However, the strategies that they used to determine how to move the windows are still not effective enough.

In addition, there are some new algorithms based on the thought of making a memo of matching condition to make the decision dynamically like bit-parallel algorithm [12].

In this paper we put forward a solution to gene sequence alignment problem by applying a new coding scheme in gene sequence encoding which is used in Gang Liao's paper [7]. In the new encoding, we do not compare two DNA sequences in the form of text string but in the form of binary string. In this way, we can introduce bit manipulation into comparison process which has higher efficiency compared with string comparison. At the same time, we can compress the size of data as well.

Enlightened by suffix tree, we design a new algorithm to construct tree-type data structure, which is named skipping tree, to improve searching efficiency when process binary streams.

Considering the hugeness of genes on a chromosome, we hope to find a method to avoid construct skipping tree based on those huge data. Since one of the major problems in bioinformatics is to find out the biological properties of certain DNA sequence by searching in the DNA or protein sequence database, we are considering construct skipping tree based on the pattern string in the DNA or protein sequence database. In this way, we can reduce the time consuming in preprocessing data of the whole chromosome. Besides, the pattern string in data-base is reusable.

Since exact repeats usually form the core blocks of approximate repeats, the algorithm we designed will serve exact matching of DNA.

II. A New Encoded Mode

In traditional method of gene sequence alignment, DNA sequences will be always abstracted as a string. Initially, for the four nucleic acid bases include adenine, guanine, thymine, and cytosine which make up DNA, define $\Sigma = \{A, C, G, T\}$. The collection Σ only incorporates four letters, A, G, C, T. In other words, only 4 different numbers are needed to encode gene sequence, which are just a small subset of text string. Thus, as shown in the Table I, we decide to propose a new encoding scheme, in which gene sequences will be represented as binary stream.

TABLE I: The Encoded Mode Table for Four Nucleic Acid Bases

A(adenine)	00
T(thymine)	11
C(cytosine)	10
G(guanine)	01

Manuscript received July 19, 2014; revised October 29, 2014.

The authors are with School of Software Engineering, Sichuan University, 610225 Chengdu, China (e-mail: xzhflying@163.com, viviancheng1993@gmail.com, dingyidy163@163.com, imtianziqiang@gmail.com).

We proposed such encoding scheme based on the following facts.

- We can cut the storage cost.
- We increase efficiency of comparison by bringing bit manipulation into comparison process.

Due to the double helix of DNA, there exists some complementary matching relationship between different bases, for instance A is complementary base of T and C is complementary base of G. Hence, Gene sequences of equal relations can be divided into two cases. One is that two DNA sequences are the same. The other is that two DNA sequences are complementary. In traditional way, there is no way to manifest the complementary relation-ship between two DNA sequences. Therefore, it has no efficient way to determine the equal relationship between the complementary DNA sequences. However, using our encoding scheme, complementary DNA sequences of target gene sequence can be obtained easily by inverting target gene sequences bitwise. For example, A is encoded as 00, whose complement is 11 which is used to represent T. In the same way, it also works for the complementary base pairs C and G.

III. Bit Manipulation in Gene Sequence Alignment

Based on the new coding scheme above, DNA sequences are abstracted as binary streams, we need to introduce bit manipulation into comparison process.

For convenience, we define the reference sequence as *T* and query sequence as *P*. *n* is indicated as the number of nucleic acid bases in reference sequence and m is denoted as the number of nucleic acid bases in query sequence. Thus, after use the coding scheme illustrated before, $T = t_1 t_2 t_3 \ldots t_{2n}$ and $P = p_1 p_2 p_3 \ldots p_{2m}$, t_i and p_i represent a bit in reference sequence and query sequence respectively. For convenience, we assume that $T(i, j) = t_i t_{i+1} t_{i+2} \ldots t_j$.

Exclusive or is a logical operation that outputs true whenever both inputs differ (one is true, the other is false).Taking advantage of it, to determine whether two binary strings are equal, we xor reference sequence with query sequence. Only if the result is 0 can two binary strings are equal. For convenience, we express the exclusive or operation as function

$$E(S_1, S_2) = S_1 \wedge S_2.$$

Besides, mask can help us to take out the data on the specific location.

After introduce the basic bit manipulation, we will then describe the algorithm of gene sequence alignment.

We regard the query sequence as a window. Searching the exact repeat of P in T actually is the window sliding from left to right one bit at a time.

There are 7 steps in gene sequence alignment.

1. Put P to the most left side of T
2. Take out the data in *T* within the window scope, that is $T(i, i+2m-1), i = 1, 2, 3 \ldots 2n-2m+1$, then compute bitwise exclusion-OR of P and it.
3. If the result is equal to 0.Then record the position of the first bit we took out, that is i.
4. Else, we perform left shift of the bits twice in the reference sequence in order to slide the window.
5. Repeat the steps 2-5 until the window is slide to the most right side of T
6. Count all positions recorded.
7. Finish.

As show above, in this way we can replace the string manipulation functions with bit manipulation, which have higher execution efficiency.

IV. Two-Step Comparison

As illustrate above, a base is represented by two bits. Only if the first bit of each base in T matches the first bit of each base in P can they possibly match each other. Hence, we divide the process of comparing into two steps in order to compare two bits of each base respectively. For convenience, we define the binary stream constituted by the first bit of each base in T as $T1$. Similarly, we define the binary stream constituted by the second bit of each base in T as $T2$. In the same way, we get $P1$ and $P2$.

There are 9 steps in gene sequence alignment.

1. Preprocess T and P.

We separate two bits of each base in T and P respectively in order to get $T1, T2$ and $P1, P2$.

2. Put $P1$ to the most left side of $T1$
3. Determine the exact repeat of $P1$ contained in $T1$. Then record the position of the first bit of the exact repeat of $P1$ (we defined it as pos).
4. Slide the window to pos of $T2$, check if there is an exact repeat of $P2$ in $T2$
5. If there exist an exact repeat of $P2$ in $T2$, record the position pos.
6. Else, slide the window to pos+1 of $T1$.
7. Repeat the steps 3-5 until he window is slide to the most right side of $T1$
8. Count all positions recorded.
9. Finish.

Now, we will analyze how it effects on improving matching efficiency by probability and statistics method.

Only if we can find a sub-string in $T1$ which can match with $P1$, then we will get to the next step to check whether $T2$ can match with $P2$ at the same position. For convenience, we assume that the length of T is $2n$, and the length of P is $2m$. Thus, the length of $T1$ and $T2$ is n, and the length of $P1$ and $P2$ is m. In each comparison, we can define the probability of the success match between $T1$ and $P1$ as $P(T1-P1)$. For every bit within the scope of window, we assign a label $T1_i$ for it, $i = 1, 2, \ldots m$.Similarly, we assign a label $P1_i$ for every bits of $P1$, $i = 1, 2, \ldots m$. We define the success match between $T1_i$ and $P1_i$ as event A_i. Event A_i are independent with each other. We assume that the pro-

bability of occurrence of each nucleic acid base is the same. The probability of occurrence of event A_i is 0.5. Based on mutually independent event probability multi-plication formula, we get $P(A_1A_2\ldots A_m)=(\frac{1}{2})^m$.

Hence, we can get $P(T1-P1)$

$$P(T1-P1)=(\frac{1}{2})^m$$

From the statistical data, the number of nucleic acid bases in query sequence is between 1000 and 10000.Here we use the average 5000 as the value of m.

Each time, when window slide, the probability of success match between $P1$ and $T1$ is $(\frac{1}{2})^{5000}$, approximately 7.080×10^{-1506}.

In other word, the chance for the second comparison is quite low. There is no need to make second comparison every time. In this way, we will nearly double operational efficiency.

V. Skipping Tree

Two-step comparison method only separates comparison into two steps. Still, we need to search exact repeat of query sequence in reference sequence bit by bit. In order to determine the maximal bits to skip within the scope of a window, skipping tree is used for searching query sequence $P1$ in $T1$. Given a query binary sequence $P1=p_1p_2p_3\ldots p_m$. For $i=1,2,\ldots m$, every $P1(1,i)$ is a prefix of $P1$. We shall label the suffixes according to the location of the starting character, that is, $P1_i=P(1,i)$. As illustrate in chapter 3, the comparison between query sequence and reference sequence can be thought as window sliding. We may put $P1$ to the left side of $T1$ and slide $P1$ from left to right one bit at a time. However, the message a comparison can convey is not only whether the sequences are equal.

If we can take advantage the characteristics of the internal arrangement of $P1$, we can determine the maxi-mal bits to skip within the scope of a window. In order to achieve this goal, we construct a series of sequences based on the query binary sequence $P1$ at first. Let $P1=010011$, $m=6$.

For convenience, we call it the preprocessing for construction of skipping tree.

1 Create an two-dimensional array at the size of $(m-1)\times(m-1)$ in order to store sequences from $P1_{m-1}$ to $P1_1$, as show in the Fig. 1. For convenience, we call it array $P1_1$.

P1	0	1	0	0	1	1
$P1_5$		0	1	0	0	1
$P1_4$			0	1	0	0
$P1_3$				0	1	0
$P1_2$					0	1
$P1_1$						0

Fig. 1. The string derived from *P*1 = 010011 by right shift operator.

2 We xor $P1_i$ with $P1$ then we get its corresponding result array, as show in the Fig. 2. For convenience, we call it array R in which R_i is corresponding result of $E(P1,P1_i)$.

R5	1	1	0	1	0
R4		0	1	1	1
R3			0	0	1
R2				1	0
R1					1

Fig. 2. Result R for P1 = 010011.

The pseudo code for creating array *R* is depicted in Alg. 1.

Algorithm 1: Create the result matrix

```
1 for i = 1; i < SIZE_OF_PATTERN ; i++
2     Temp_P1 = P1;
3     P1=P1<<1;
4     Result_Matrix [i]= (Temp_P1 ) ^ (P1);
5 end for
```

The preprocessing for construction of skipping tree help us to record the characteristics of the internal arrangement of $P1$. To avoid search the pattern sequence bit by bit in target sequence, we will construct skipping tree based on the array above to decide the maximal bits to skip within the scope of a window.

A skipping tree of $P1$, which is in the length of m, is a tree with the following properties:

1) Each tree edge is labeled 0 or 1.

2) Each internal node has at most two children.

2) Each leaf node has stored the maximal bits to skip or NULL.

3) For $1\le i<m$, each R_i has its corresponding labeled path from root to a leaf or an internal node.

The skipping tree can be constructed from $P1$ in $O(m^2)$ linear time.

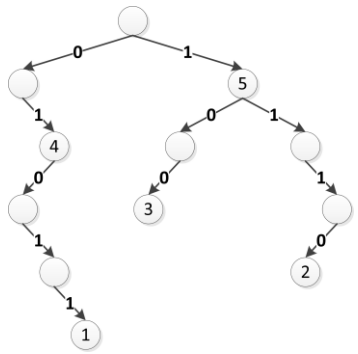


Fig. 3. The skipping tree for query sequence *P*1 = 010011.

As illustrates in Fig. 3, after get the array *R*, there are 2 steps to construct skipping tree.

1) From R_1 to R_{m-1}, for R_i, form a path from root to a node with the maximal bits to skip recorded inside, that is $m-i$.

2) For R_i, traversal data from the back forward until the first bit of the sequence and insert the nodes to form a path from root to a node with the maximal bits to skip recorded inside. For convenience, we assume that $R_i=a_1a_2\ldots a_i$, a_i represents a bit in R_i.

The rules for inserting node are as follows:

Start from the root, check if there is a edge of root labeled a_i .If there is not, add an edge labeled a_i to the root and add a new node to the edge.

Repeat the same process until a_2.

Follow the current node, add an edge labeled a_1 to the node and add a new node with the maximal bits to skip, $m-i$, recorded inside to the edge.

Initialize both of its children as NULL.

The pseudo code for Skipping tree construction is depicted in Alg. 2.

Algorithm 2: Build the skipping tree based on result matrix

```
1  for  i = SIZE_OF_PATTERN - 1; i > 0; i--
2     for  j = SIZE_OF_PATTERN - 1; j >= i; j--
3        if  Result_Matrix[i][j] == 0  Then
4           if  root->L_Leaf == NULL  Then
5              root->L_Leaf = new Node();
6           end if
7           root = root->L_Leaf;
8        else
9           if  root->R_Leaf == NULL  Then
10             root->R_Leaf = new Node();
11          end if
12          root = root->R_Leaf;
13       end if
14    end for
15    root->jump = i;
16 end for
```

After the skipping tree is constructed, we discuss how to use it for searching query sequence $P1$.

In each comparison, the algorithm steps are as follows:

1) We xor the $T1$ within the window scope with $P1$, we define the result sequence as S , $S = s_1 s_2 \ldots s_m$, s_i represents a bit in sequence S .

2) Traversal from s_m forward. Begin from the root, search the edge with the same label as s_i until a NULL node.

3) The maximal bits to skip is recorded in the parent node of the NULL node.

The pseudo code for searching skipping tree is depicted in Alg. 3.

Algorithm 3: Search the max jump number based on the skipping tree

```
1  int temp_jump = SIZE_OF_PATTERN ;
2  for  i = SIZE_OF_PATTERN - 1; i >= 0; i--
3     if (result[i] == 0)
4        if  root.L_Leaf == NULL          Then
5           return temp_jump;
6        end if
7        root = root.L_Leaf;
8        if  root.jump > 0        Then
9           temp_jump = root.jump;
10       end if
11    else
12       if  root.R_Leaf == NULL          Then
13          return temp_jump;
14       end if
15       root = root.R_Leaf;
16       if  root.jump > 0        Then
17          temp_jump = root.jump;
18       end if
19    end if
20 end for
```

To illustrate the mathematical principle of skipping tree, we will use boolean expressions.

In boolean algebra, $P1_i$ can be expressed as $P1 >> i$, thus R_i can be expressed as $(P1 \wedge (P1 >> i))$ .When we search the query sequence $P1$ in reference sequence $T1$ by the method of skipping tree, the search process can be abstracted as $(P1 \wedge T1) \wedge (P1 \wedge (P1 >> i))$. $(P1 \wedge T1) \wedge (P1 \wedge (P1 >> i))$ is equal to $T1 \wedge (P1 >> i)$. In other word, the search process can be described in mathematical formula $T1 \wedge (P1 >> i)$. It is exactly the same as process of window sliding illustrated above.

VI. Skipping Tree

We can see from the above chapters, the algorithm efficiency can be improved in three aspects, bit manipulation, two-step comparison and skipping tree.

Bit manipulation is very simple, but its running time can be improved. All data in computer memory is stored in a binary form. Bit manipulation, is essentially directly manipulate integer in memory operations in the form of binary stream. At the same time, every bit of binary stream won't affect each other. Due to bit manipulation directly manipulate memory data, the processing speed is very fast.

When it comes to two-step comparison, as analyze in chapter, it will greatly reduce the need for second match because in the actual situation, the chance of successful match in first comparison is quite low. In macro level, we only need to deal with half of the amount of original data, which will double the operational efficiency.

We now consider the expected time complexity of constructing and searching skipping tree. We assume that the length of the reference sequence T is $2n$ and the length of the query sequence P is $2m$, Different from other algorithm which construct tree based on reference sequence, we construct skipping tree based on the query sequence. The skipping tree can be constructed from $P1$ in $O(m^2)$ linear time. Space that is required to store this skipping tree structure is $(m-1)*(m-1)$ .In the actual situation, the data size of reference sequence is much larger than the data size of query sequence. Thus, we greatly reduce the time of skipping tree construction. Meanwhile, we greatly reduce the amount of space that is required to store this skipping tree structure.

From chapter 5, we know that the depth of the tree (including the root node) is equal to the length of the query sequence m . For convenience, we use variable d to represent the search depth, $d \le m$ .The sum of search depth and the maximal bits to skip is $m+1$. In worst cases, that is when the data in current window is exactly same as R_{m-1}, the maximal bits to skip is one bit , but we need to traverse the whole tree in O(m) linear time. In best cases, that is there is no suffix of the data in current window equal to any suffix of R array, the maximal bits to skip is m bit and search stop at the first layer of the tree.

For convenience, we emphasize some obvious properties of the skipping tree.

There is one, and only one node with maximal bits to skip recorded inside in each layer.

There is one, and only one path from root to each node.

To calculate the average search depth $\bar{d}$, we need to know the conditions for the end of the search.

We label the path from root to the node in layer i with maximal bits to skip recorded inside $path_i$.

Search stop at layer i of the tree, $i < m$ only if

1) It can go through the $path_i$.

2) It can't go through $path_{i+1}, path_{i+2}, \ldots, path_m$.

If $i = m$, since it is the last layer of the skipping tree, there is no layer below it. Search stop only if

1) It can go through the $path_m$.

Since the sum of search depth and the maximal bits to skip is $m+1$, for better generality, we deduce the ave-rage search depth $\bar{d}$ by the average maximal bits to skip.

We assume that random variable X represent the maximal bits to skip, $X \in \{1, 2, \cdots m\}$. As analyze before, we discuss the distribution of random variable X res-pectively when $X = 1$ and $X = x$, $x > 1$. Since the probability of going through the $path_i$ is $(\frac{1}{2})^{i-1}$.

Since there is only one path from root to the node in layer i with maximal bits to skip recorded inside, we can easily deduce that the probability of going through the $path_i$ is $(\frac{1}{2})^{i-1}$.

Thus, the probability of not going through the $path_i$ is $1-(\frac{1}{2})^{i-1}$.

Therefore,

$$P(X=1) = (\frac{1}{2})^{m-1}$$

Similarly, when $X = x$, $x > 1$

$$P(X=x) = \frac{1}{2^{m-x}} \prod_{i=1}^{x-1} (1 - \frac{1}{2^{m-i}})$$

Hence, we get the formula for distribution of random variable *X*

$$P(X=x) = \begin{cases} \frac{1}{2^{m-1}}, x=1 \\ \frac{1}{2^{m-x}} \prod_{i=1}^{x-1} (1-\frac{1}{2^{m-i}}), x>1 \end{cases}$$

Based on the random variable mathematical expectation formula, we get

$$EX = \frac{1}{2^{m-1}} + \sum_{j=2}^{m} \left[\frac{j}{2^{m-j}} \cdot \prod_{i=1}^{j-1} (1 - \frac{1}{2^{m-i}}) \right]$$

We use EX to estimate the average maximal bits to skip. Thus, the average search depth is $(m+1) - EX$.

As show in Fig. 4, it describes function curve with changes of *m* in mathematical expectation of random variable *X*. *m* range from 1000 to 10000, and the step size is 400.We can easily see from the Fig. 4 that changes of m in mathematical expectation of random variable *X* is nearly linear growth process. From the points in the Fig. 4, we know that the average search depth is between 2 and 3. Thus, the average search depth is quite low. In each comparison, window is almost skipping the whole window length.

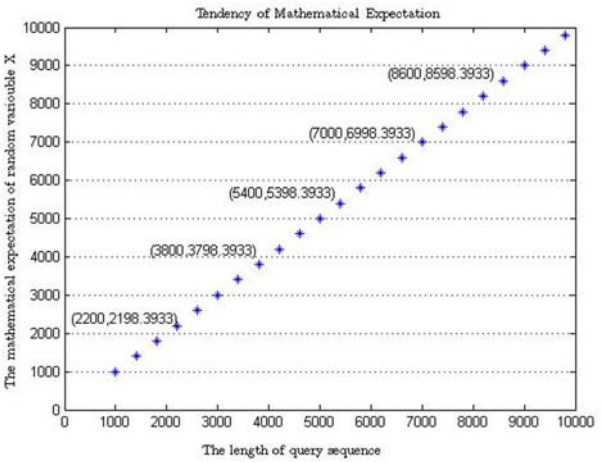


Fig. 4. Tendency of EX.

VII. Result

The skipping tree algorithm was evaluated in a computer composed of an Intel Core i5 3210M quad-core processor, running at 2.5GHz, with 6GB DRAM.

We extracted DNA sequence from GEN BANK database to do the contrast experiment to evaluate the performance of skipping tree algorithm, KMP and suffix array. The reference sequence are extracted from the chromosomes of animal Monodelphis domestica, Equus caballus isolate Twilight breed thoroughbred, Sus scrofa breed mixed and Theobroma cacao cultivar Matina. We used difference sizes of reference sequence in our experiment, ranging from 1M to 32M. All the query sequences we used are extracted from the chromosomes of corresponding animal which provides the reference sequence for them. Several sizes of query sequences were used in the experimental test, ranging from 8 to 4096 nucleotides long.

Most methods of gene sequence alignment can't strike balance between preprocess and search. That is one of the reason why those algorithm can't achieve better performance overall. KMP and suffix array is classical example. KPM represents the algorithm which has fast preprocess but intolerable search efficiency, while suffix array represents the algorithm which has excellent search efficiency but intolerable preprocess time. Thus, we choose these two algorithms to make a contrastive evaluation with

skipping tree algorithm to illustrate how skipping tree strike balance between preprocess and search.

Table II and Table III give the specific data of the contrastive experiments. We draw Fig. 5-Fig. 11 based on these two tables.

In the case of different length of reference sequence and difference length of query sequence, we test the KMP algorithm. As shown in the Fig. 5, the total processing time increases obviously with the increase of the length of reference sequence. Specially, in the case of the same length of reference sequence, with the increase of query sequence, the total processing time drops down.

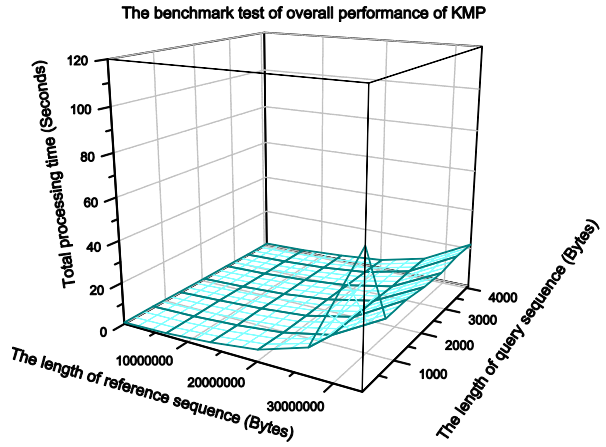


Fig. 5. The benchmark test of overall performance of KMP.

Similarly, we test the suffix tree. As shown in the Fig. 6, the total processing time increases sharply with the increase of the length of reference sequence. However, the length of query doesn't affect the total processing time a lot.

Here, we simply calculate total processing time by adding the preprocess time and search time up. Besides, in the case of the same size of reference sequence, with the increase of the size of query sequence, performance of skipping tree drop down in the preprocess stage but go up in the searching stage.

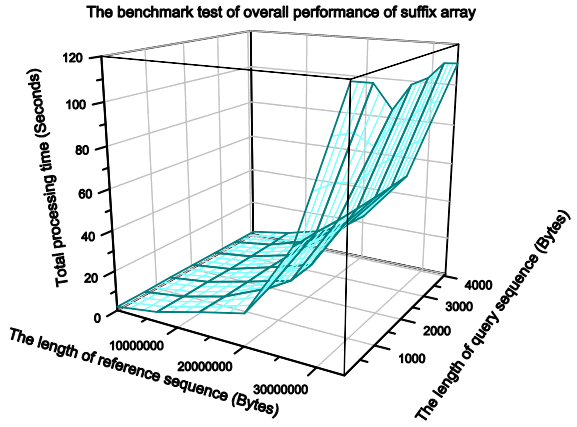


Fig. 6. The benchmark test of overall performance of suffix array.

As shown in the Fig. 7, the total processing time hardly increases both with the increase of the length of reference sequence and query sequence. We can conclude that the total processing time of skipping tree is the shortest. Specially, skipping tree has close performance as KMP in preprocess stage and as suffix array in the searching stage when the size of query sequence is same.

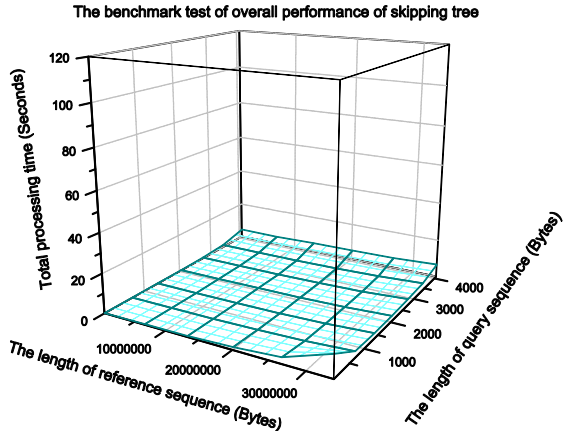


Fig. 7. The benchmark test of overall performance of skipping tree.

TABLE II: A CONTRASTIVE EVALUATION OF PREPROCESS TIEM OF KMP, SUFFIX ARRAY AND SKIPPING TREE ALGORITHM

The same size of query sequence(4096 nucleotides)				The same size of reference sequence(16777216 nucleotides)			
Reference Sequences/nucleotide	Skipping Tree	KMP	Suffix Array	Query Sequences/nucleotide	Skipping Tree	KMP	Suffix Array
1048576 (1M)	4.165	0.003	1.669	8	0	0	50.709
2097152 (2M)	4.001	0.003	5.22	32	0	0	50.534
4194304 (4M)	4.447	0.002	11.012	128	0.005	0	49.666
8388608 (8M)	4.442	0.002	24.023	512	0.065	0	49.398
16777216 (16M)	4.236	0.002	50.022	1024	0.278	0	49.374
33554432 (32M)	4.166	0.003	110.668	2048	1.176	0.001	49.528
				4096	4.236	0.002	50.022

TABLE III: A CONTRASTIVE EVALUATION OF SEARCHING TIEM OF KMP, SUFFIX ARRAY AND SKIPPING SUFFIX ALGORITHM

The same size of query sequence(4096 nucleotides)				The same size of reference sequence(16777216 nucleotides)			
Reference Sequences/nucleotide	Skipping Tree	KMP	Suffix Array	Query Sequences/nucleotide	Skipping Tree	KMP	Suffix Array
1048576 (1M)	0.007	0.607	0	8	1.58	12.012	0.001
2097152 (2M)	0.013	1.15	0.001	32	0.321	9.027	0.001
4194304 (4M)	0.025	2.436	0	128	0.153	9.113	0.001
8388608 (8M)	0.049	4.362	0	512	0.107	9.084	0
16777216 (16M)	0.101	9.41	0	1024	0.122	8.75	0
33554432 (32M)	0.197	20.895	0	2048	0.095	9.442	0
				4096	0.101	9.41	0

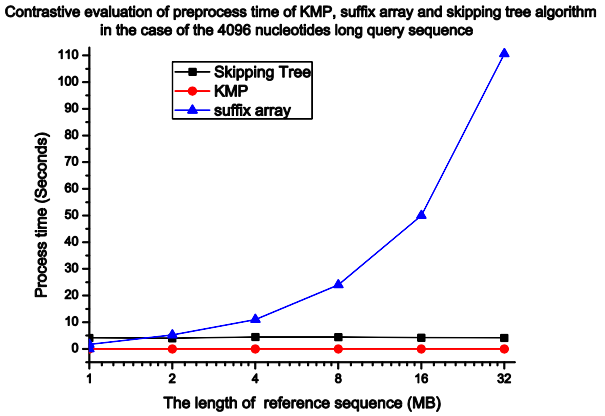


Fig. 8. Contrastive evaluation of preprocess time of KMP, suffix array and skipping tree algorithm in the case of the 4096 nucleotides long query sequence.

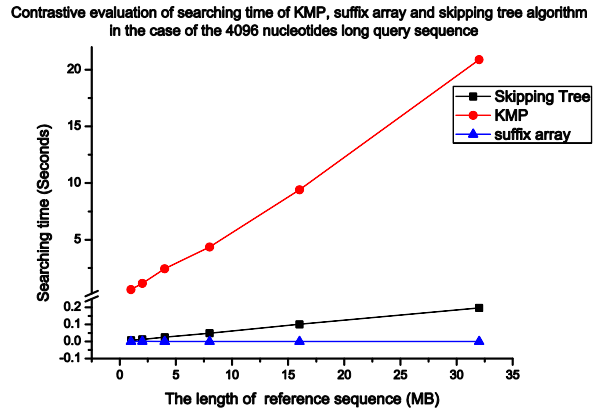


Fig. 9. Contrastive evaluation of searching time of KMP, suffix array and skipping tree algorithm in the case of the 4096 nucleotides long query sequence.

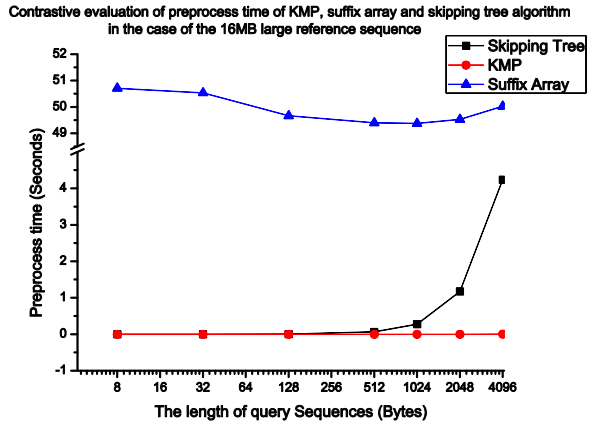


Fig. 10. Contrastive evaluation of preprocess time of KMP, suffix array and skipping tree algorithm in the case of the 16MB large reference sequence.

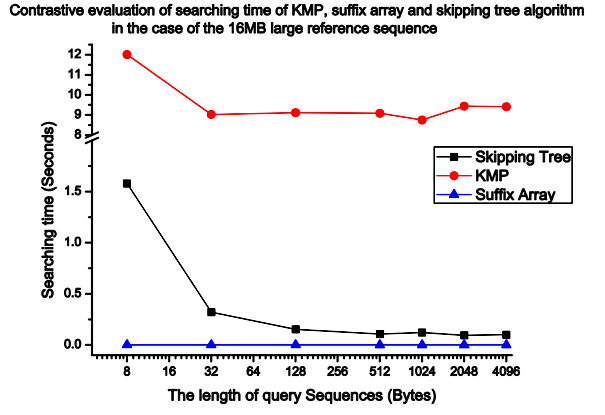


Fig. 11. Contrastive evaluation of searching time of KMP, suffix array and skipping tree algorithm in the case of the 16MB large reference sequence.

We compare the preprocess time of KMP, suffix array, skipping tree in the case of different reference sequence size when the query sequence is 4096 nucleotides long. The result is illustrated in Fig. 8. Because skipping tree is built based on query sequence, there is scarcely any influence when the size of reference changes. We can see from the Fig. 8 that it has close efficiency with KMP.

In Fig. 9, the searching performance of these three algorithms with the increase of reference sequence is portrayed. Though with the increase of reference sequence skipping tree cannot perform as better as suffix tree do, the result is still satisfactory.

As illustrate in Fig. 10, in the condition of the same size of reference sequence, with the increase of query sequence, the preprocess time of skipping tree increase sharply since the scale of the skipping tree depend on the length of query sequence. However, the longer query sequence is, the faster searching time will be, because in probability, the longer query sequence is, the lower search depth will be. Thus, skipping tree performs better in searching stage if size of query sequence increases. The more details are observed in Fig. 11.

VIII. Conclusion

This paper proposed a new algorithm skipping tree to gene sequence alignment problem by applying a new encoded mode for DNA sequence and a new data structure, skipping tree, which are extra applicable for accelerating DNA sequence matching. We compare the computation efficiency of our algorithm, skipping tree,

KMP and suffix array both in preprocess stage and multiple genome sequence matching stage.

These observation reveal that skipping tree algorithm far more efficient than suffix array in the preprocess stage, which can be constructed from reference sequence in $O(m^2)$ linear time. Besides, it is also better than KMP in multiple genome sequence matching. When the length of the reference sequence is between 1000 and 10000, the average search depth is between 2 and 3.Thus, it strikes the balance between preprocess time and multiple genome sequence matching time successfully. According to the results, there are convincing reasons to believe that multiple genome sequence matching based on skipping tree is an efficient approach to high performance bio-informatics applications.

Authors' Contribution

Kewei Cheng and Zihuan Xu contributed equally to this paper.

Acknowledgment

This work was completely supported and funded by Sichuan University College. All authors read and approved the final manuscript.

References

[1] J. Shendure and H. Ji, "Next-generation DNA sequencing," *Nature Biotechnology*, vol. 26, pp. 1135-1145, Oct. 2008.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., The MIT Press, 2009.

[3] A. H. F. Laender and A. L. Oliveira, "Lecture notes in computer science," in *Proc. 9th International Symposium on String Processing and Information Retrieval Conf.*, Lisbon, 2002, pp. 31-43.
[4] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, pp. 262-272, April 1976.
[5] U. Manber and G. Myers, "Suffix arrays: A new method for online string searches," *SIAM Journal on Computing*, vol. 22, pp. 935-948, 1993.
[6] 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, pp. 378-407, 2005.
[7] G. Liao, Q. Sun, L. Ma, S. Ding, and W. Xie, "Ultra-fast multiple genome sequence matching using GPU," *arXiv Preprint arXiv*: 1303.3692, 2013.
[8] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, pp. 323-350, 1977.
[9] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762-772, October 1977.
[10] D. M. Sunday, "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, pp. 132-142, 1990.
[11] K. Nałęcz-Charkiewicz and R. Nowak, "Algorithm to search for genomic rearrangements," in *Proc. SPIE 8903, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013*, Wilga, 2013.
[12] K. H. Chen, G. S. Huang, and R. C. T. Lee, "Bit-parallel algorithms for exact circular string matching," *The Computer Journal*, vol. 57, pp. 731-743, August 2014.

Zihuan Xu is currently an undergraduate in the School of Software Engineering, Sichuan University, China.

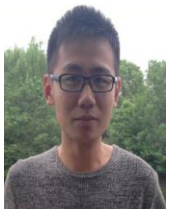
His research concerns bioinformatics and high performance computing. And now he focuses on multiple genome sequence matching and proteomics. He is also interested in parallel programming.

Kewei Cheng is currently an undergraduate in the School of Software Engineering, Sichuan University, China.

She is engaged in bioinformatics and data mining research. More specifically, her research concerns multiple genome sequence matching and proteomics.

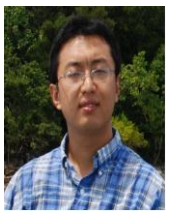
Yi Ding is currently an undergraduate student in the School of Software Engineering, Sichuan University, China.

Her research concerns bioinformatics and high performance computing. Meanwhile, she is interested in parallel computing and parallel architecture.



Ziqiang Tian is currently an undergraduate student in the school of Software Engineering, Sichuan University, China.

He is engaged in algorithms and big data.



Hui Zhao is currently a Faculty member of the School of Software Engineering, Sichuan University, China. He got his BS, MS, and PhD degrees from Sichuan University.

His research fields include information security, artificial intelligence, and machine learning.