Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB 2007)

An ACGT-Words Tree for Efficient Data Access in Genomic Databases

Ye-In Chang, Wei-Horng Yeh, Jiun-Rung Chen, and Jen-Wei Hu
Dept. of Computer Science and Engineering, National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C
Telephone: (886) 7–525–2000 (ext. 4334)
Fax: (886) 7–525–4301

***Abstract*— Genomic sequence databases, like GenBank [2], EMBL, are widely used by molecular biologists for homology searching. Because of the increase of the size of genomic sequence databases, the importance of indexing the sequences for fast queries grows. In this paper, we propose a new index structure, ACGT-Words tree[1], for efficiently support query processing in genomic databases. We define the concept of words which is different from the word definition given in the word suffix tree, and separate the DNA sequences stored in the database and in the query sequence into distinct words. Our approach does not store all of the suffixes in the database sequences. Therefore, we need less space than the suffix tree approach. We also propose an efficient search algorithm to do the sequence match based on the ACGT-Words tree index structure. Therefore, we could take less time to finish the search than the suffix array approach. Moreover, our approach avoids the missing cases occurring in the word suffix tree. The simulation results show that our ACGT-Words tree outperforms the suffix tree and the suffix array in terms of storage and processing time, respectively.**

I. INTRODUCTION

Bioinformatics has received increased publicity over the past few years, in large part due to its importance to the Human Genome Project. With the increase of genomic sequences, there are numerous databases holding these DNA and protein sequences [5]. Basically, the primary structure of **DNA (Deoxyribonucleic acid)** is represented as strings, or linear sequences, which is composed of four basic molecules called *nucleotides* with different nitrogen bases: *Adenine* (A), *Guanine* (G), *Cytosine* (C), and *Thymine* (T) [9]. One characteristic of the sequences in those public nucleotide databases is its long length. The average sequence length is around 1,000 bases, with sequences ranging from 10 to 700,000 bases in length. The DNA sequences can not break into words, because each character in sequences is important. Due to the long length of sequences, the search in databases becomes more difficulty.

A general method for reducing searching costs is to store an abstraction or index that can be used to assess broad similarity to a query [9]. Given a set of database sequences, the well-known Wilbur-Lipman approach [8] is first preprocess, through hashing each *interval* in the query sequence. An interval in this context is a fixed-length overlapping subsequence from a sequence. In [9], they used an inverted index to select a subset of sequences in a *coarse search*. The inverted index uses hashing to store the intervals and search the query sequence. However, it will have some missing cases. Weiner proposed the *suffix tree* [7] that is a compact version of the *suffix tries*. A suffix tree of a string with length n will have n leaf nodes, one per suffix. Therefore, the disadvantage of the suffix tree is its large storage space. To reduce the storage space, Manber and Myers [6] proposed the *suffix array*. This data structure is basically a sorted list of all the suffixes of strings. They only store the sorted list. Although the suffix array reduces the storage space for indexing DNA sequences, the construction and search of the suffix array waste too much time. In [4], Irving and Love proposed the *suffix binary search tree* to efficiently construct a suffix array.

For most approaches mentioned above, they store all suffixes of the sequences. They do not consider the concept of words. Andersson and Nilsson designed the *word suffix trees* [1], which breaks a sequence into words. Although the word suffix tree stores less suffixes than traditional suffix trees, it will lose information in the searching process. Take Figure 1 as an example. In Figure 1-(a), the sequence $TGAGC$ occurs at position 4. However, in Figure 1-(b), we can not find this sequence in the tree structure.

In this paper, we design a new data structure to index the DNA sequences. We use the concept of words, which is different from the word definition given in the word suffix tree [1], to construct our index structure. The definition of an ACGT-Word is as follows: for a sequence which starts with character k, $k \in \{A, C, G, T\}$, an ACGT-Word consists of the successive characters from character k to the character which is the previous one before the same character k appearing again in the sequence. We will generate all ACGT-Words from all suffixes of the given sequence according to the definition. For example, for DNA sequence $AGAGACT\$$, the related ACGT-Words are $\{AG, GA, AG, GACT, ACT, CT, T\}$. Figure 2 shows the ACGT-Words tree index structure constructed by the same DNA sequence shown in Figure 1-(a). Although the number of words generated by the word suffix tree is less than that of our index structure, the word suffix tree loses some information as we mentioned previously. For any word which occurs at many positions of the sequence, we only construct the word once in our tree structure. For example, in Figure 2, word $ACGCTG$ is constructed only once in the tree structure,

[1]This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-101.

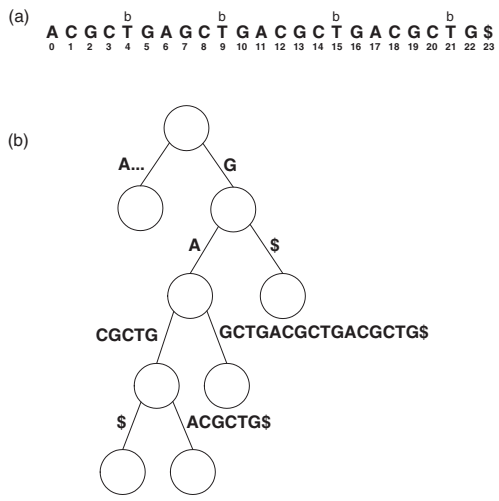

Fig. 1. An example sequence where $\flat = T$: (a) the input sequence and the position of the delimiter; (b) the word suffix tree.

even if this word occurs three times in the DNA sequence, *i.e.*, positions 0, 11, and 17. Hence, we can reduce the storage space as compared to the suffix tree. Moreover, the ACGT-Words tree has better performance than the suffix array in terms of the construction time and the query processing time. Table I shows the comparison of the complexity between the suffix array and our ACGT-Words tree structure. From our performance analysis and experiments on real data, we show that the ACGT-Words tree outperforms the suffix tree and the suffix array in terms of the storage and processing time, respectively.

The rest of this paper is organized as follows. Section 2 describes the word suffix trees. Section 3 presents the proposed ACGT-Words tree index structure. In Section 4, we study the performance. Finally, Section 5 gives the conclusion.

II. The Word Suffix Trees

Traditional suffix tree construction algorithms rely heavily on the fact of that all suffixes are inserted. Andersson *et al.* [1] proposed *word suffix trees*. These trees store, for a string of length *n* in an arbitrary alphabet, only *m* suffixes that start at word boundaries.

A word suffix tree is constructed by an input string consisting of *n* characters from an alphabet of size *k*, including two special characters, \$ and ♭. The \$ character is an end marker which must be the last character of the input string and can not appear elsewhere, while ♭ represents some *delimiting character* which appears at *m* - 1 places of the input string. Andersson *et al.* regarded the input string as a series of words — the *m* non-overlapping substrings ending either with \$ or ♭. There may of course exist multiple occurrences of the same word in the input string. They want to create a tree structure containing *m* strings, namely the suffixes of the input string that start at the beginning of words [1]. Figure 1-(b) shows an example of the word suffix tree.

TABLE I

A Comparison of the Complexity between the Suffix Array and the ACGT-Words Tree

	construction time	search time
Suffix Array	$O(nlog(n))$	$O(m + logn)$
ACGT-Words tree	$O(n)$	$O(m)$

n: the length of the database sequence; m: the length of a query sequence

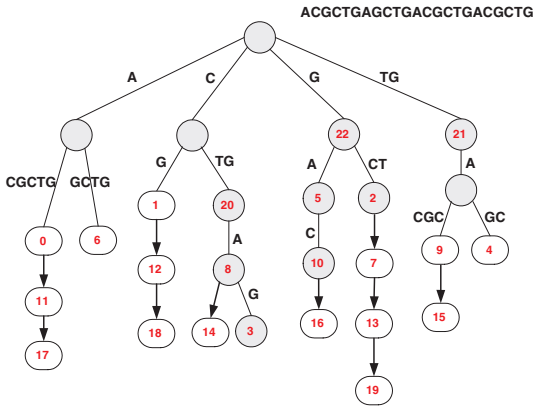

Fig. 2. The ACGT-Words tree for the sequence ACGCTGAGCTGACGCTGACGCTG

III. The ACGT-Words Tree

Although the word suffix tree uses the concept of words and reduces the storage space, they lose information in the searching process. In this section, we formally define a ACGT-Word in the DNA sequence, which is different from the word definition given in the word suffix tree, and present a new data structure called the *ACGT-Word* tree to index the DNA sequence.

A. The Definition

Let $S = s_0 s_1 ... s_{n-1}$ be a sequence of n characters over alphabet $\sum = \{A, C, G, T, \$\}$. A *substring* of S is a sequence $S_i^j = s_i s_{i+1} ... s_j$ for some $0 \leq i \leq j \leq n-1$. The sequence $W_i = S_i^j$ is a *ACGT-Word* for the given sequence S, where $s_{j+1} = s_i$ or $s_{j+1} = \$$ or $j = n-1$.

B. Tree Construction

In this subsection, we illustrate how to construct the ACGT-Words tree for the given sequence S. We have three procedures, $Construct(S)$, $Split$, and $Insert$, in the process of tree construction, where procedure $Construct(S)$ is the main procedure for constructing the ACGT-Words tree for input sequence S ; procedure $Split$ splits one ACGT-Word from S; procedure $Insert$ inserts ACGT-Words into the tree structure. First, in Procedure $Construct$ shown in Figure 3, according to the current processing character of S, we apply procedure $Split$ to split sequence S into words. For example, for the given input sequence $ATACACGAT\$$ (S) as shown in Figure 4-(a), Figure 4-(b) shows the process of generating the ACGT-Words.

```
Procedure Construct(S);
/* Generate the words of input sequence S. */
/* P is an index of the input sequence S. */
    begin
        P := 0;
        while ( S[P] ≠ '$') do
        begin /* Create a new Node and a new Edge */
            if ( S[P] = 'A' ) then
                Split(A_num, S, P);
            else if ( S[P] = 'C' ) then
                Split(C_num, S, P);
            else if ( S[P] = 'G' ) then
                Split(G_num, S, P);
            else if ( S[P] = 'T' ) then
                Split(T_num, S, P);
            P := P + 1;
        end;
        InsertRemainingWord();
    end;
```

Fig. 3. Procedure $Construct(S)$

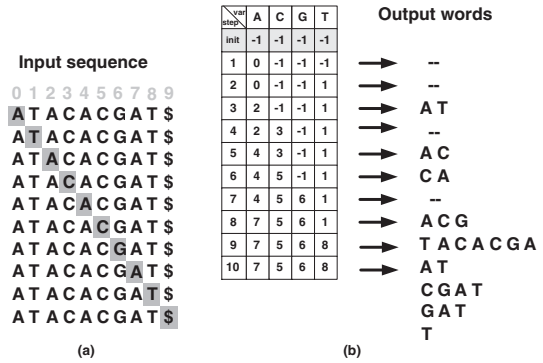

Fig. 4. The steps for splitting the ACGT-Words: (a) scan input sequence; (b) change the variables and generate the words.

Assume that P is a position of the sequence S. There are four variables, A_num, C_num, G_num, and T_num, which store the starting position for words beginning with A, C, G, and T, respectively. Initially, let P be pointed at the starting position of the sequence S, and variables A_num, C_num, G_num, and T_num be set to -1, as shown in Figure 4-(b). First, we find that the first character of the sequence S is A. Then, we check whether the variable A_num is -1 in Procedure $Split$. If A_num is -1, we change the variable A_num to the value of P; otherwise, a word will be generated. Since A_num is -1 now, we set $A\_num = P = 0$. After changing the value of A_num, position P points to the next character of the sequence S. Because the pointed character is T and the value of T_num is -1, we change the value of T_num to 1, and P will point to the next character. Now, P is at position 2, and the pointed character is A. Since variable A_num is 0, not -1, which implies that we find a word which locates from A_num to (P - 1). The first ACGT-Word, $W_0 = AT$, of the sequence S is generated. The similar steps are processed until the ending symbol \$ is reached. When we reach the ending symbol \$, the values of variables A_num, C_num, G_num, T_num may not equal to -1. It means that some ACGT-Words are not generated yet.

TABLE II

VARIABLES USED IN PROCEDURE $SearchQS(QS)$

QS	the query sequence
SW	an array which records all the searched words of QS
$SW[i]$	the i-th searched word in SW
$\|SW[i]\|$	the length of $SW[i]$
num	the number of searched words
X	an array which stores the result of function $SearchSW$
Y	an array which stores the result of the searching process
$flag$	true if QS is in the database
$lastword$	true if the searched word is the last word of QS
E	one edge in the ACGT-Words tree
$E.label$	characters stored in edge E
$E.endnode$	the connected node under edge E
k	the length of the same prefix between the searched word and $E.label$
$suffix\_num$	occurring positions stored in one node of the ACGT-Words tree

Therefore, as shown in Figure 4-(b), we generate the remaining ACGT-Words AT, $CGAT$, GAT, and T by calling Procedure $InsertRemainingWord$.

In procedures $Split$ or $InsertRemainingWord$, if one ACGT-Word is generated, we call procedure $Insert$ to insert the generated word into the ACGT-Words tree. Procedure $Insert$ is similar to the method used in the word suffix trees [1], which inserts words into the word suffix tree. The only difference is that in the word suffix trees method, they record "in-order numbers" in nodes of the tree structure. However, in our method, for each ACGT-Word, we record its occurring positions of input sequence S in a node, where this node is at the end of the path for this word in our tree structure. Take the ACGT-words shown in Figure 4-(b) as an example. Figures 5 shows the process of inserting all the ACGT-Words. For example, in Figure 5-(a), we insert word AT into the ACGT-Words tree. Since word AT occurs at position 0 of the given sequence $ATACACGAT\$$, we create one edge with label equal to AT and one node which records position 0. Then, in Figure 5-(b), we insert word AC into the ACGT-Words tree. Because the first character of AC, $i.e.$, A, is the same as the first character of label AT in the edge, we split this edge into three edges, *i.e.*, edges with labels "A", "C", and "T". That is, words AC and AT will share the same edge with label A, but have the different edges with label C and T. We also create a new node to store the occurring position of word AC, *i.e.*, position 2. Therefore, from Figure 5-(b), we could know that word AC appears at position 2 and word AT appears at position 0.

C. Search

Given a query sequence, QS, we apply procedure $SearchQS(QS)$ to do the searching process. Table II shows variables used in procedure $SearchQS$, and Figure 6 shows this procedure. This procedure cuts the query sequence, QS, into searched words. Moreover, procedure $SearchQS$ will set flag $lastword$ to true if the searched word is the last word of QS. After generating the searched words, we call function $SearchSW(R, SW[i], lastword)$ to search one word from our ACGT-Words tree, where R is the node that we are

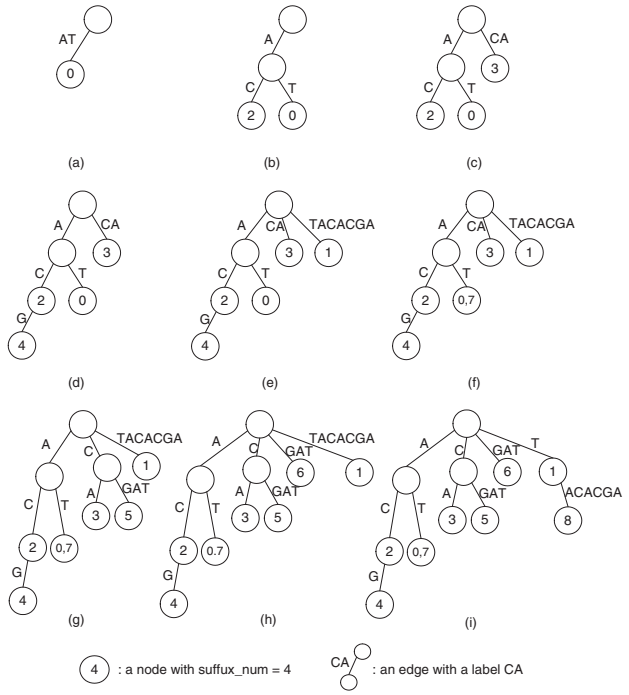

Fig. 5. An example of insertion: (a) AT; (b) AC; (c) CA; (d) ACG; (e) TACACGA; (f) AT; (g) CGAT; (h) GAT; (i) T.

TABLE III
EIGHT CASES IN SEARCH

	$k < \|SW\|, \|E.label\|, k \geq 0$	$\|SW\| = k < \|E.label\|, k > 0$	
$lastword = true$ $lastword = false$	Case 1	Case 2 Case 3	
	$\|SW\| > k = \|E.label\|, k > 0$	$\|SW\| = \|E.label\| = k, k > 0$	
		no $suffix\_num$	has $suffix\_num$
$lastword = true$ $lastword = false$	Case 4	Case 5 Case 6	Case 7 Case 8

traversing now. (Note that in function $SearchSW$, we denote $SW[i]$ as SW.) There are eight different cases in function $SearchSW$. Table III shows the conditions of these eight cases, and the related semantics are illustrated in Figure 7. These cases are described as follows:

1) Case 1 (as shown in Figure 7-(a)): This case occurs if there is no common prefix sequence between word SW and $E.label$. It means that we can not find an edge whose label matches the searched word. The output for this case will be that no sequence matches the query sequence. For example, in Figure 8, assume that $QS = GACGT$ and $SW = GAC$. When we traverse this tree from the root R, there is no edge whose label starts with character G. Therefore, we conclude that no sequence matches word GAC in the database, and $flag$ is set to false.

2) Case 2 (as shown in Figure 7-(b)): This case occurs if word SW is the same as the prefix of $E.label$, and SW is

Procedure $SearchQS(QS)$;
/* The main procedure of the searching process in the ACGT-Words tree. */
/* $Split(QS)$ is a function that returns an array which contains the searched words of QS. */
/* $TotalNum(SW)$ is a function that returns the number of searched words in the query sequence QS. */
/* $Sort(X)$ is a function that returns the resulting array after sorting array X. */
/* $Combine(Y, X)$ is a function that returns an array which contains the matching positions in the database. */

```
begin
    SW := Split(QS);
    num := TotalNum(SW);
    X := ∅
    Y := ∅;
    flag := true;
    lastword := false;
    i := 0;
    while ( flag = true and i < (num − 1) ) do
    begin
        X := SearchSW(R, SW[i], lastword);
        if X = ∅ then
            flag := false;
        else
        begin
            Y := Combine(Y, X, |SW[i − 1]|);
            i := i + 1;
        end;
    end;
    lastword := true;
    if ( flag ≠ false ) then
    begin
        X := SearchSW(R, SW[i], lastword);
        X := Sort(X);
        Y := Combine(Y, X, |SW[i − 1]|);
    end;
    if ( X = ∅ or Y = ∅) then
        writeln(" Not found ! ");
    else
        writeln(Y);
end;
```

Fig. 6. Procedure $SearchQS(QS)$

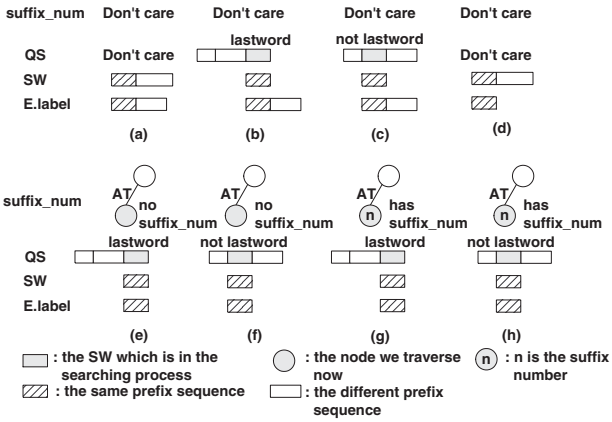

Fig. 7. Eight cases in the searching process: (a) Case 1; (b) Case 2; (c) Case 3; (d) Case 4; (e) Case 5; (f) Case 6; (g) Case7; (h) Case 8.

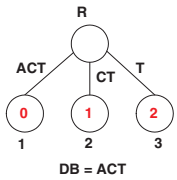

Fig. 8. An example for Case 1 in the searching process ($QS =$$\underline{GAC}GT$, $SW = GAC$)

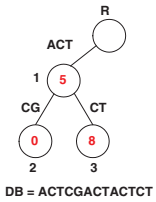

Fig. 9. An example for Case 2 in the searching process ($QS = AGT\underline{AC}$, $SW = AC$)

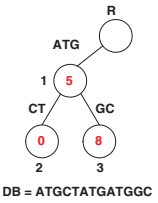

Fig. 10. An example for Case 3 in the searching process ($QS =$$\underline{AT}ACG$, $SW = AT$)

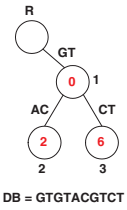

Fig. 11. An example for Case 4 in the searching process ($QS =$$\underline{GTCT}GA$, $SW = GTCT$)

the last searched word of the query sequence. In this case, we apply function $OutputAll()$. This function traverses the subtree from the last edge which the searched word SW meets, and returns an array of $suffix\_num$'s which are stored in those nodes that function $OutputAll()$ traverses. In Figure 9, assume that $QS = AGTAC$ and $SW = AC$ (*i.e.*, the last searched word). We traverse the tree from the root node, and find that AC is the same as the prefix of $E.label = ACT$ ($|E.label| > |SW|$). Therefore, $E.label = ACT$ is one output that matches the condition $SW = AC$. Due to SW is the last word of QS ($lastword = true$), it may have other outputs, *e.g.*, $ACTCG$ and $ACTCT$ in this example, which also match the searched word. Function $OutputAll()$ will also traverse those nodes. Therefore, the output of this example is {0, 5, 8}. It means that positions 0, 5, and 8 match the last searched word $SW = AC$.

3) Case 3 (as shown in Figure 7-(c)): This case occurs if SW is the prefix of $E.label$, but is not the last searched word of the query sequence. This means that no sequence matches the query sequence. For example, in Figure 10, assume that $QS = ATACG$ and $SW = AT$. Since word AT is not the last word of QS, word AT must exactly match the label of edge E, i.e., $SW = E.label$. However, in this example, we have $SW = AT \neq E.label = ATG$. Therefore, there is no sequence matching query sequence QS.

4) Case 4 (as shown in Figure 7-(d)): This case occurs if $E.label$ is the prefix sequence of word SW. First, we call function $Shift(E.label, SW)$ to cut the prefix sequence between $E.label$ and SW. Second, we change the searched node R to the end node of E. Finally, we call function $SearchSW$ again to search the new searched word. For example, in Figure 11, assume that $QS = GTCTGA$ and $SW = GTCT$. First, we traverse the tree from the root node. We find that we will reach node 1 after searching GT (the common prefix between $E.label = GT$ and $SW = GTCT$). Next, we change the searched node to node 1, and the new search word becomes CT (due to $|SW| > |E.label|$). Then, we call function $SearchSW$ again to traverse the tree deeply (from node 1 with $SW = CT$). Finally, the searched word $GTCT$ is found in the database (*i.e.*, position 6 stored in node 3).

5) Case 5 (as shown in Figure 7-(e)): This case occurs if (1) word SW is exactly equal to $E.label$; (2) word SW is the last searched word of the query sequence; (3) $E.endnode$ has no $suffix\_num$. It means that this node is created by two or more edges that have the same prefix sequence. For example, In Figure 12, sequence $ATGATC$ constructs the tree. Let's consider the branch with the starting character A. It has two ACGT-Words, ATG and ATC. These two ACGT-Words have shared characters, AT. Hence, after constructing the ACGT-Words tree, the tree contains one edge $E.label = AT$ and $E.endnode$ (*i.e.*, node 1) has no $suffix\_num$. Therefore, in this case, we only call function $OutputAll()$. It traverses the tree from the edge which the searching word meets. For example, assume that $SW = AT$. In Figure 12, we traverse the tree from the root node. Then, edge E with label AT ($E.label = AT$) matches the word SW, and $E.endnode$, node 1, stores no $suffix\_num$. Therefore, we call function $OutputAll()$. The result after calling function $OutputAll()$ is {0, 3}. It means that positions 0 and 3 of the database sequence match the searched word $SW = AT$.

6) Case 6 (as shown in Figure 7-(f)): This case occurs if (1) word SW is exactly equal to $E.label$; (2) word SW is not the last searched word of the query sequence; (3) $E.endnode$ has no $suffix\_num$. In this case, there is one difference with Case 5: word SW is not the last searched word of the query sequence. This means that no sequence matches the query sequence. For example, in Figure 13, assume that $QS = CACTG$ and $SW = CA$ ($lastword = false$). We traverse the tree from the root node, and SW matches edge

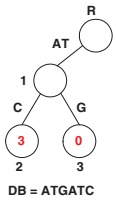

Fig. 12. An example for Case 5 in the searching process ($QS = ACG\underline{AT}$, $SW = AT$)

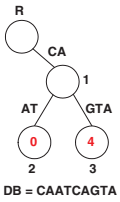

Fig. 13. An example for Case 6 in the searching process ($QS = \underline{CA}CTG$, $SW = CA$)

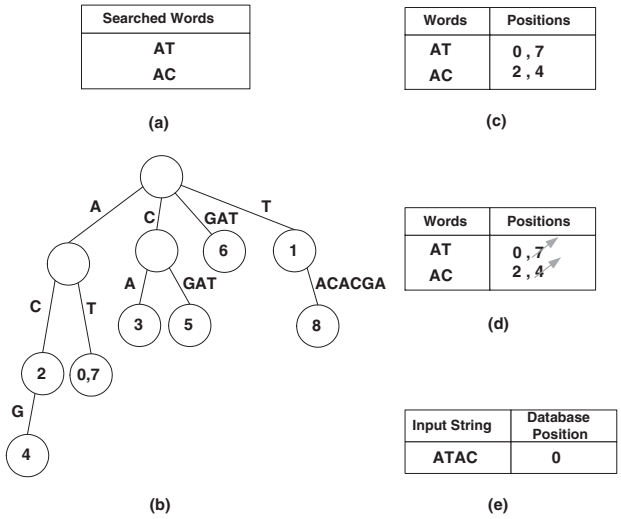

Fig. 15. The searching process for the query sequence $ATAC$ (AT in Case 8, AC in Case 7) : (a) the ACGT-Words; (b) the tree structure; (c) the initial result; (d) combining; (e) the final result.

E with a label CA. However, $E.endnode$, node 1, stores no $suffix\_num$. Therefore, we output that no sequence matches the query sequence $QS = CACTG$ in the database.

7) Case 7 (as shown in Figure 7-(g)): This case is similar to Case 5. The only difference is that $E.endnode$ has $suffix\_num$. Therefore, first, we output those $suffix\_num$'s stored in $E.endnode$. Next, we apply function $OutputAll()$ to traverse the subtree under $E.endnode$.

8) Case 8 (as shown in Figure 7-(h)): This case is similar to Case 6. The only difference is that $E.endnode$ has $suffix\_num$. In this case, word SW exactly matches the $E.label$ in the tree. For example, in Figure 14, assume that $QS = CTCGACA$ and $SW = CT$. We traverse the tree from the root node. We find that edge E with label CT exactly matches $SW = CT$, and $E.endnode$, node 1, stores positions 0 and 5 ($E.endnode.suffix\_num \neq null$). Therefore, the result of this example is {0, 5}. It means that these two positions of the database sequence match the searched word $SW = CT$.

Up to this point, we have discussed all cases considered by function $SearchSW$. Next, in procedure $SearchQS$, after applying function $SearchSW$ for a searched word $SW[i]$, we will combine the result from function $SearchSW$, *i.e.*, X, and the previous found result, *i.e.*, Y, by applying function $Combine(Y,\ X,\ |SW[i]|)$. In function $Combine$, we prune those positions in Y that can not occur in the query sequence. After we process all searched words, array Y, which is returned from function $Combine$, stores the positions that query sequence QS occurs in the database.

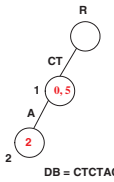

Fig. 14. An example for Case 8 in the searching process ($QS = \underline{CT}CGACA$, $SW = CT$)

Figure 15 shows an example of the searching process in the ACGT-Words tree. Given database sequence $ATACACGAT$ and query sequence $QS = ATAC$, in our searching process, procedure $SearchQS$ will split query sequence $ATAC$ into two searched words, $\{AT,\ AC\}$. When a searched word, SW, is generated, we search this searched word by function $SearchSW$ immediately. In this example, we search $SW = AT$ first. Then, positions 0 and 7, which satisfy the condition $SW = AT$, are returned by function $SearchSW$. Finally, function $Combine(Y,\ X,\ len)$ combines X = {0, 7} with $Y = \emptyset$. Therefore, {0, 7} is stored in Y. Now, we search the next word $SW = AC$. In procedure $SearchQS$, we know that AC is the last searched word of the query sequence QS. We traverse the tree structure from the root, and find that $SW = AC$ occurs at position 2. Then, position 2 is added into X. However, since $SW = AC$ is the last searched word of the query sequence QS in this example, we have to traverse all nodes under the node which we are traversing now. Therefore, position 4 will be added into X. After applying function $SearchSW$ for $SW = AC$, we get the resulting array X = {2, 4}. Finally, we combine $X = \{2, 4\}$ and the previous result $Y = \{0, 7\}$ by applying function $Combine(Y,\ X,\ |SW[i-1]|)$. Since the previous searched word, AT, occurs at positions 0 and 7 of the database sequence, the current searched word, AC, should occur at positions 2 ($= 0+2$) and 9 ($= 7+2$). Therefore, position 2 in X could be combined with position 0 in Y. That is, position 0 of the database sequence matches the query sequence $ATAC$.

TABLE IV

A COMPARISON OF THE NUMBER OF NODES BETWEEN THE SUFFIX TREE AND THE ACGT-WORDS TREE FOR THE REAL DNA SEQUENCES

	Drosophila	Part of human Chromosome 19
DNA length	14405	17010
ACGT-Words Tree	14992	17704
Suffix Tree	23509	27898

TABLE V

PARAMETERS USED IN SIMULATION MODEL

Parameter	Meaning	Value
PN	the number of distinct patterns	6..10
$PL(i)$	the length of the ith synthetic pattern	
Min_PL	the minimum length of a synthetic pattern	5
Max_PL	the maximum length of a synthetic pattern	10
$Mean\_PL$	the mean length of a synthetic pattern	7
Min_TL	the minimum length of the database sequence	10000
Max_TL	the maximum length of the database sequence	20000
$Mean\_TL$	the mean length of the database sequence	15000
QN	the number of query sequences	200
QL	the length of query sequences	
Min_QL	the minimum length of a query sequence	50
Max_QL	the maximum length of a query sequence	200
$Mean\_QL$	the mean length of a query sequence	80
θ	the probability for containing a pattern in the DNA sequence	0.1..1.0

IV. PERFORMANCE STUDY

In this section, we study the performance of the proposed ACGT-Words tree data structure, and make a comparison with the suffix tree and the suffix array data structures by simulation. First, we study the performance by using the real DNA sequences in GenBank. Then, in order to evaluate the performance of the algorithms over a large range of data characteristics, we generate the synthetic data and study the performance by using the synthetic data.

A. Experiment Results for Real Genomic Databases

In this subsection, we study the performance of the ACGT-Words tree by using the real DNA sequences in GenBank (http://www.ncbi.nlm.nih.gov). We use Drosophila and Human Chromosome 19 as our database sequences. Table IV shows the comparison of the number of nodes between the suffix tree and the ACGT-Words tree. Obviously, the number of nodes used in the ACGT-Words tree is less than that used in the suffix tree.

B. Generation of Synthetic Data

In this subsection, we generate synthetic sequences to evaluate the performance of the algorithms. The synthetic data is used to simulate the occurrence of a DNA query sequence in a genomic database. The parameters used in our performance model are shown in Table V. The details of our performance model are described in [3]. The simulation results is the average of 2000 experiments for different parameters which we consider.

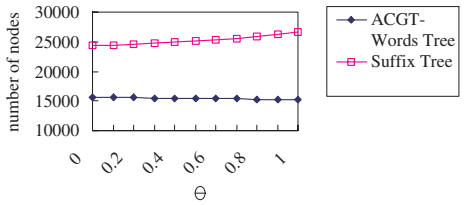

Fig. 16. A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different probabilities of patterns (TL = 15000, PN = 6, PL = 7)

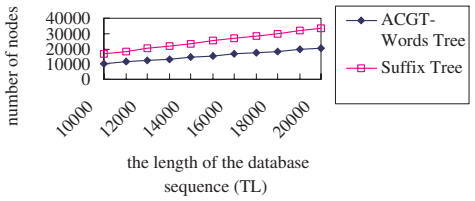

Fig. 17. A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different length of the database sequence (θ = 0.5, PN = 6, PL = 7).

C. Simulation Result

In this subsection, we show the simulation results of the ACGT-Words tree.

1) The Suffix Tree vs. the ACGT-Words Tree: A comparison of the number of nodes under the different occurring probabilities of the same patterns is shown in Figure 16. From this figure, we show that the ACGT-Words tree always requires less number of nodes than the suffix tree no matter what value of the probability θ is. That is, the ACGT-Words tree needs less storage space than the suffix tree. Moreover, as the occurring probability of the same patterns increases, the number of nodes increases in the suffix tree, but decreases in the ACGT-Words tree. This is because as the occurring probability of the same patterns increases, the number of ACGT-Words is less than the number of suffixes.

A comparison of the number of nodes under the different length of database sequences is shown in Figure 17. From this figure, we show that the ACGT-Words tree always requires less number of nodes than the suffix tree no matter what value the length of the database sequence (TL) is. As TL increases, some patterns may occur for many times. Therefore, the increasing rate of the number of nodes of the ACGT-Words tree is slower than that of the suffix tree.

A comparison of the number of nodes under the different number of patterns is shown in Figure 18. From this figure, we show that the ACGT-Words tree always requires less number of nodes than the suffix tree no matter what value the number of database patterns (PN) is. The simulation result does not contain any obvious regulation. That is, the value of PN is not the main factor for affecting the storage spaces of the suffix tree and the ACGT-Words tree.

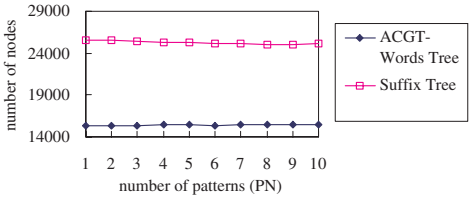

Fig. 18. A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different number of patterns (θ = 0.5, TL = 15000, PL = 7).

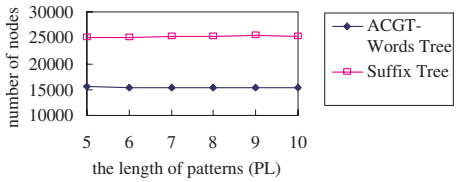

Fig. 19. A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different length of patterns (θ = 0.5, PN = 6, TL = 15000).

A comparison of the number of nodes under the different length of patterns is shown in Figure 19. From this figure, we show that the ACGT-Words tree always requires less number of nodes than the suffix tree no matter what value the length of patterns (PL) is. Similar to the previous result, this result also contains no obvious regulation.

2) The Suffix Array vs. the ACGT-Words Tree: A comparison of the constructing time based on the different length of the database sequences is shown in Figure 20. In this figure, we show that the ACGT-Words tree always takes less time than the suffix array to construct the index structure no matter what length the database sequence is. When the suffix array is constructed, it needs to sort all suffixes in the database sequence first. Therefore, it will take much time and require more space to construct its data structure.

A comparison of the searching time under the different length of query sequences is shown in Table VI. In this table, we observe that the ACGT-Words tree always takes less time for the searching process than the suffix array. The searching process of the suffix array uses a binary search. In the ACGT-Words tree, we search only one path for the query sequence. Although we need to combine the results after searching ACGT-Words, we still have a better performance than the suffix array.

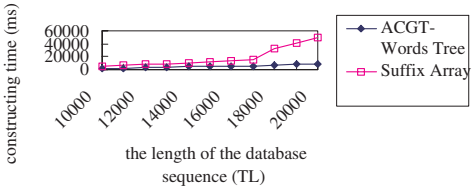

Fig. 20. A comparison of the constructing time between the ACGT-Words tree and the suffix array under the different length of database sequence

TABLE VI
A COMPARISON OF THE SEARCHING TIME BETWEEN THE SUFFIX ARRAY AND THE ACGT-WORDS TREE UNDER THE DIFFERENT LENGTH OF QUERY SEQUENCES

Query Length	ACGT-Words Tree (ms)	Suffix Array (ms)
50	36	89
60	65	76
70	91	92
80	55	81
90	76	98
100	76	110
150	88	179
200	101	187

V. CONCLUSION

Genomic sequence databases are widely used by molecular biologists for searching. In this paper, we proposed a new data structure, ACGT-Words, for indexing the genomic sequences. In the proposed ACGT-Words tree, we insert the ACGT-Words generated, instead of all suffixes, into the tree structure. The size of the ACGT-Words tree is small, and the ACGT-Words tree has no missing case occurred in the word suffix trees. From our simulation results, we have shown that the storage space of the ACGT-Words tree is less than that of the suffix tree. Moreover, we have shown that the construction time and the search time of the ACGT-Words tree are shorter than those of the suffix array. How to extend our index structure to support the *approximate sequence matching* in database sequences is the possible research direction.

REFERENCES

[1] A. Andersson, N. J. Larsson, and K. Swanson, "Suffix Trees on Words," *Algorithmica,* vol. 23, no. 3, pp. 246–260, Jan. 1999.
[2] *http://www.ncbi.nlm.nih.gov/Genbank/index.html*
[3] J. W. Hu, "An ACGT-Words Tree for Efficient Data Access in Genomic Databases," *Master Thesis, Dept. of Computer Science and Eng., National Sun Yat-Sen University,* 2003.
[4] R. W. Irving and L. Love, "Suffix Binary Search Trees and Suffix Arrays," *Technical Report no. TR-2001-82 of the Computing Science Department of Glasgow University,* March 2001.
[5] W. C. Kim, S. Park, J. I. Won, S. W. Kim, and J. H. Yoon, "An Efficient DNA Sequence Searching Method Using Position Specific Weighting Scheme, " *Journal of Information Science,* vol. 32, no. 2, pp. 176–190, April 2006.
[6] U. Manber and E. W. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing,* vol. 22, no. 5, pp. 935–948, Oct. 1993.
[7] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. of the 14th IEEE Annual Symp. on Switching and Automata Theory,* pp. 1–11, 1973.
[8] W. J. Wilbur and D. J. Lipman, "The Context Dependent Comparison of Biological Sequences," *SIAM Journal of Applied Mathematics,* vol. 44, no. 3, pp. 557–567, 1984.
[9] H. E. Williams and J. Zobel, "Indexing and Retrieval for Genomic Databases," *IEEE Trans. on Knowledge and Data Eng.,* vol. 14, no. 1, pp. 63–78, Jan./Feb. 2002.