AN ACGT-WORDS TREE FOR EFFICIENT DATA ACCESS IN GENOMIC DATABASES

A Thesis

Submitted to the Faculty

of

National Sun Yat-sen University

by

Jen-Wei Hu

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

June 2003

ABSTRACT

Genomic sequence databases, like GenBank, 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. The DNA sequences are composed of 4 base pairs, and these genomic sequences can be regarded as the text strings. Similar to conventional databases, there are some approaches use indexes to provide efficient access to the data. The inverted-list indexing approach uses hashing to store the database sequences. However, the perfect hashing function is difficult to construct, and the collision in a hash table may occur frequently. Different from the inverted-list approach, there are other data structures, such as the suffix tree, the suffix array, and the suffix binary search tree, to index the genomic sequences. One characteristic of those suffix-tree-like data structures is that they store all suffixes of the sequences. They do not break the sequences into words. The advantage of the suffix tree is simple. However, the storage space of the suffix tree is too large. The suffix array and the suffix binary search tree reduce more storage space than the suffix tree. But since they use the binary searching technique to find the query sequence, they waste too much time to do the search. Another data structure, the word suffix tree, uses the concept of words and stores partial suffixes to index the DNA sequence. Although the word suffix tree reduces the storage space, it will lose information in the search process. In this thesis, we propose a new index structure, ACGT-Words tree, 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 can take less time to finish the search than the suffix array approach. Our approach also avoids the missing cases in the word suffix tree. Then, based on the ACGT-Words tree, we propose one improved operation for data insertion and two improved operations for the searching process. In the improved operation for insertion, we sort the ACGT-Words generated and then preprocess them before constructing the tree structure. In the two improved operations, we can provide better performance when the query sequence satisfies some conditions. The simulation results show that the ACGT-Words tree outperforms the suffix tree and the suffix array in terms of storage and processing time, respectively. Moreover, we show that the improved operations in the ACGT-Words tree also require shorter time to construct or search than the original processes or the suffix array.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

Introduction

Bioinformatics has received increased publicity over the past few years, in large part due to its importance to the Human Genome Project. The first draft of the human genome was published in February 2001. With the increase of genomic sequences, numerous and large databases holding these DNA and protein sequences. These databases are now readily available over the WEB and are quickly becoming the "lifeblood of molecular biology". This is due to the combination of the power of biomolecular sequence comparison with the ease-of-use of tools for searching such databases for sequences exhibiting similarity to a given query sequence. These searches are nowadays routine and vital for molecular biology because they serve to "generate new knowledge". There are many computational issues in biology needed to solve, such as the gene finding, the genome rearrangement, the protein folding, and the sequence finding and alignment. Due to the increase of computing power, computer scientists can assist molecular biologists in reducing the time to solve those issues [4].

1.1 Genomics

To realize the mystery of gene, we briefly describe the background of molecular biology.

Deoxyribonucleic acid (DNA), stores complete instructions for all the cellar functions of an organism. DNA sequences hold the code of life for every living organism. The primary structure of DNA is represented as strings, or linear sequences. Figure 1.1 is an example of this structure. DNA is composed of four basic molecules

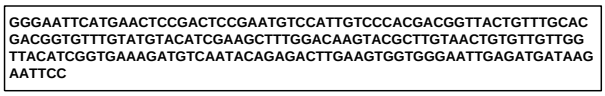


Figure 1.1 Nucleotide structure of a synthesized human

called *nucleotides*, which are identical except that each contains a different nitrogen base. Each nucleotide contains phosphate, sugar, and one of the four bases: *Adenine* (A), *Guanine* (G), *Cytosine* (C), and *Thymine* (T) [12, 18, 24, 29].

The structure of DNA is described as a *double helix*, as shown in Figure 1.2, which looks rather like two interlocked bedsprings. The two helices are held together by hydrogen bonds. Each base pairs consists of one *purine* base (A or G) and one *pyrimidine* base (C or T), paired according to the following rule: $G \equiv C, A = T$ (each '-' symbolizes a hydrogen bond), as shown in Figure 1.3. That is, whenever there is an A in one strand, there will be a corresponding T in the other strand. DNA sequences will replicate itself at the certain time. The replication reaction is catalyzed by the enzyme *DNA polymerase*. This enzyme can extend a chain, then a short nucleotide chain generates a segment of duplex DNA that is turned into a new strand by the replication process.

Each DNA sequence exists in a separate *chromosome*, and the total genetic information stored in the chromosomes of an organism is said to constitute its *genome*. The human genome contains about 3×10^9 base pairs, organized as 46 chromosomes. The 24 different chromosomes from 50×10^6 to 250×10^6 bp. A gene is a region of DNA that controls a discrete hereditary characteristic. That is, gene is a functional DNA. Most genes have their coding sequences called *exons*, interrupted by non-coding sequences called *introns*. In humans genes constitute approximately 2-3% of the DNA, leaving 97-98% of non-genic *junk DNA*. The role of introns is as yet unknown; however, experiments involving removal of these parts proved to be lethal. This means that all regions of DNA sequence are important [24].

Figure 1.2 DNA double helix structure

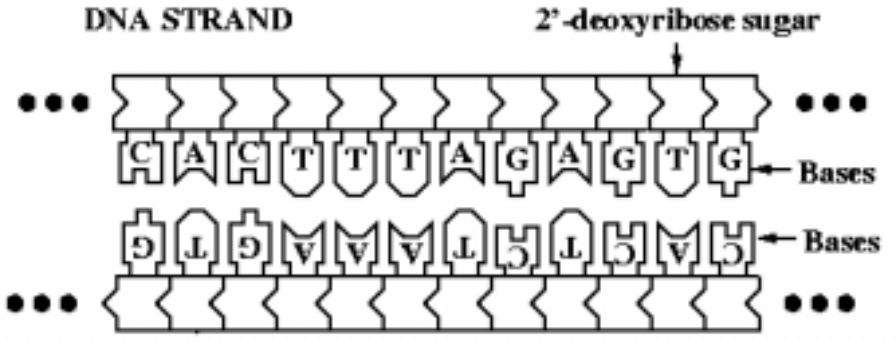


Figure 1.3 The rule of pairs

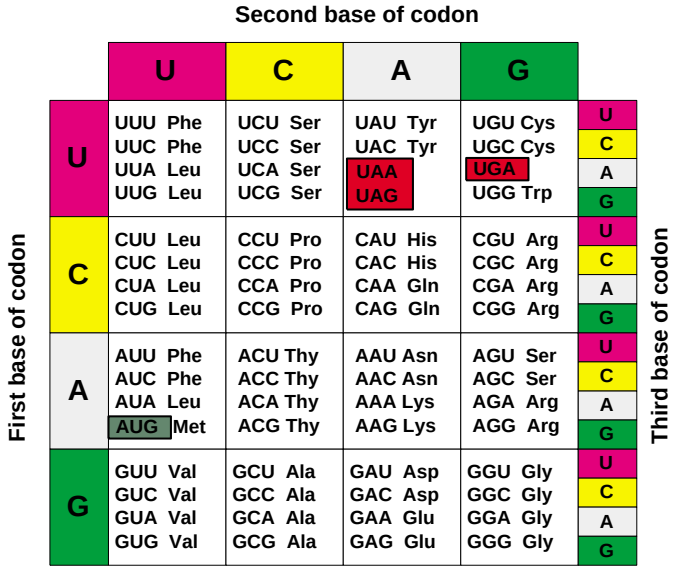


Figure 1.4 The genetic code: the three terminators are boxed in the upper part of this figure; the AUG initiator codon is shown in the left part of this figure.

Ribonucleic acid (RNA) is a long chain of nucleic acids, and it has many different properties: (1) In RNA, there is no Thymine (T) which exists in DNA. Instead, *Uracil* (U) is found in RNA. In other words, RNA is constructed out A, G, C, and U. (2) DNA has a double helix structure while RNA has only one strand.

RNA plays an important role in the production of the particular protein which a cell needs. A complex of RNA processing enzymes removes all the intron sequence, thereby producing a much shorter RNA molecule, *mRNA*. The pattern of the splicing can vary depending on the tissue in which the transcription occurs. After producing mRNA, the sequence of nucleotides will be translated into the amino acid sequence of the corresponding protein. Each triplet of nucleotides, called a *codon*, specifies one amino acid. Since RNA is a linear polymer of four different nucleotides, there are $4^3 = 64$ possible codon triplets, as shown in Figure 1.4. However, only 20 different amino acids are commonly found in proteins, so that most amino acids are specified by several codons. In addition, 3 codons specify the end of translation, called *stop codon*. The codon AUG is the initiator codon in the translation [18, 24].

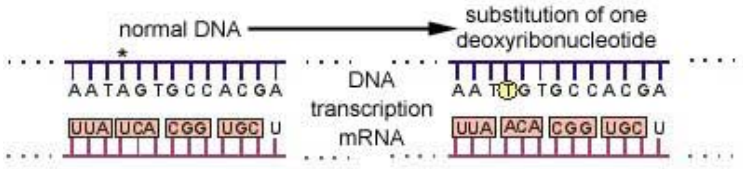


Figure 1.5 Substitution of a nucleotide

Proteins are the structural components of living cells and tissue, and thus an important building block in all living organisms. Humans can synthesize about 80,000 different kinds of proteins. Proteins determine the functions of cells. For instance, proteins make up most hairs, fingernails, and skins. On the other hand, many proteins are enzymes, chemical materials that speed up chemical processes which are necessary for living organisms to function as living things. The building blocks of proteins are the amino acids. Only 20 different amino acids make up the diverse array of proteins found in living things. The average protein size is around 200 amino acids long, while large proteins can reach over a thousand amino acids [18].

A **mutation** is defined as a heritable change in the nucleotide sequence in the DNA, caused by a faulty replication process. These errors in replication occur often due to exposure to ultra violet radiation or the other environment condition. There are several kinds of point mutations: (1) Substitution: a change of one nucleotide in the DNA sequence, as shown in Figure 1.5; (2) Insertion: an addition of one or more nucleotides to the DNA sequence, as shown in Figure 1.6; (3) Deletion: a removal of one or more nucleotides from th DNA sequence, as shown in Figure 1.7.

Due to the mutation, it comes the problem of *alignment* and increases the difficulty of the search in genomic databases. The sequence alignment problem can be easily seen that we have to compare two sequences. For instance, it may well happen that we are given a DNA sequence and there is another DNA sequence. They are not exactly the same. Yet, we like to adjust these two sequences in some way such that

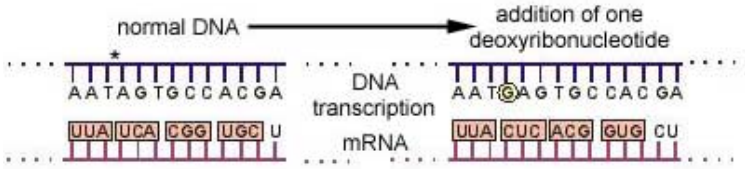


Figure 1.6 Insertion of a nucleotide

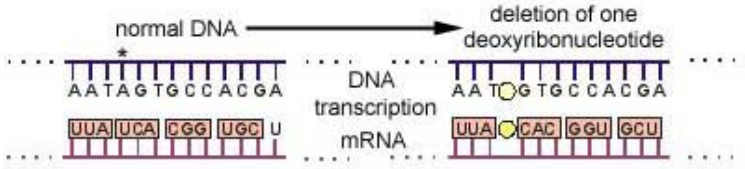


Figure 1.7 Deletion of a nucleotide

information can be extracted from them. Consider, for example, the following two sequences:

ATTCATTACAACCGCTATG

ACCCATCAACAACCGCTATG

These two sequences do not look similar to each other. Sequence alignment lets us measure the similarity of two sequences. The operations of alignment have substitution, insert blanks (gaps) or characters, and deletion. Suppose we perform a sequence alignment operation on these sequences, the result will be as follows:

ATTCATTA-CAACCGCTATG

ACCCATCAACAACCGCTATG

1.2 Query Types of Genomic Databases

There are several public nucleotide databases. Three of the larger repositories are GenBank, the DNA Databank of Japan, and the European Molecular Biology Laboratory database. Additionally, there are several single-species databases, such as the Portable Mouse Genome database.

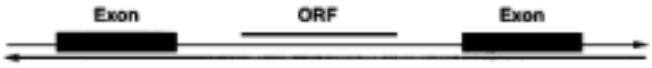
GenBank stores sequence data generated through the human genome initiative, which not only focuses on the human genome, but also on organisms, as shown in Figure 1.8. Currently, GenBank contains around 300 million nucleotide bases and is doubling in size every 21 months. Data within GenBank is, in some cases, duplicated through the submission of identical and, rather more frequently, overlapping sequences. Additionally, there is a small but a significant error rate, both as a result of sequence determination errors and of data entry errors. Nucleotide databases store strings representing sequences of nucleotide bases, and each string annotated by the natural language description of the function of the sequence. One characteristic of the sequences in these 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. It will miss some information if one or more characters are deleted from the sequence. Due to the long length of sequences, the search in databases becomes more difficulty.

In [8], they proposed the query language for genomic databases. They presented a set of built-in operations on DNA sequences that are used in composing queries presented. A query is a sequence of nucleotide bases whose properties are not fully known; answers are nucleotide sequences that exhibit local similarity to the query. A list of built-in operations is summarized in Table 1.1. Some examples of queries are as follows: (1) Figure 1.9 retrieves the name, location, and sequence of the open reading frames that occur within introns. (2) Figure 1.10 locates the open reading frames that are overlapping on the same strand.

```
LOCUS       CC144564                489 bp    DNA     linear   GSS 23-APR-2003
DEFINITION  CSU-K20-3A10.T7 CSU-K20 Aedes aegypti genomic clone CSU-K20-3A10,
            DNA sequence.
ACCESSION   CC144564
VERSION     CC144564.1  GI:30065760
KEYWORDS    GSS.
SOURCE      Aedes aegypti (yellow fever mosquito)
  ORGANISM  Aedes aegypti
            Eukaryota; Metazoa; Arthropoda; Hexapoda; Insecta; Pterygota;
            Neoptera; Endopterygota; Diptera; Nematocera; Culicoidea; Aedes.
REFERENCE   1  (bases 1 to 489)
  AUTHORS   Severson,D.W., deBruyn,B., Lovin,D.D., Brown,S.E., Knudson,D.L. and
            Morlais,I.
  TITLE     Comparative genome analysis of the yellow fever mosquito Aedes
            aegypti with the malaria vector mosquito Anopheles gambiae and
            Drosophila melanogaster
  JOURNAL   Unpublished (2003)

  BASE COUNT      167 a     86 c     91 g    145 t
  ORIGIN
        1 tcgacctgca ggcatgcaag cttgtttact ctttggaaaa ataatccaca cgctttatat
       61 ctgaaatcgg tactggatgt tgtttaaatt ccgcaaatct tataatgttt gacatgcatc
      121 tttgtacgaa aaatgaattc ttgagaatcc agggacattg gtgatattac aaaagattat
      181 taaaccgtta gaaactgtcg aacacaataa gaaatcataa ttcagaagtt acctatcgac
      241 cccactagta accgagtcaa gttgagtaac aactgttgaa aaactcgaat gatacagacg
      301 tgccgagcaa atttttgga cgacagtatt gttcgttttg taacgaatag gttttgtgca
      361 aatcttaaat gtggggaaat aaataggtac tcacacgcaa atttgtctgg cagttcagca
      421 aaagcacaaa ccttacattc aatgcacaat ccaaaagtgt acgttatatg tgcgatacat
      481 gtagatttt
```

Figure 1.8 Sample GenBank entry




```
SELECT    gene.name, gene.INTRON.{location, seq},
          gene.ORF.{location, seq}
FROM      gene
WHERE     gene.INTRON.seq.contains gene.ORG.seq
```

Figure 1.9 Containing features and its corresponding query language

Table 1.1 Table of built-in operations

$contains(s, t)$	returns true if sequence s contains sequence t and false otherwise.
$overlaps(s, t)$	returns true if a prefix of sequence s is identical to a suffix of sequence t and returns false otherwise.
$precedes(s, t, u)$	this function returns true if there an occurrence of t within s precedes some occurrence of u within s.
$before(s, t, n)$	retrieves n characters of the string s prior to an occurrence of t within the string s.


```
SELECT    H.{location, seq}, J.{location, seq}
FROM      gene
WHERE     H is gene.ORF and
          J is gene.ORF and
          H != J and
          H.seq overlaps J.seq
```

Figure 1.10 Overlapping features and its corresponding query language

Basically, the query types of genomic databases contain the *exactly* and *partially* matches. The exactly match means that the query sequence has to be the same as the database sequence. The first example mentioned above shows the exactly match. The partial match means that the query sequence matches the database sequences up to k errors, and the second example shows the partial match. The regions derive from a common ancestor sequence, so that identification of the existence of similarity, through sequence comparison of these regions can shed light on the evolutionary history, biochemical function, and chemical structure of these molecules. A primary use of GenBank is for querying for such similarities [28, 29].

These genomic databases are used to assist molecular biologists to determine the biochemical function and the chemical structure of query strings, and to investigate the evolutionary history of organisms. A query is a sequence of nucleotide bases whose properties are not fully known; answers are nucleotide sequences that exhibit local similarity to the query [28]. That is, database search algorithms are used to compute pairwise comparisons between a candidate query sequence and each of the sequences stored within a database in order to find all the pairs of sequences that have a similarity above a defined threshold. There are three principal database search algorithms: (1) *Smith-Waterman algorithm*, (2) *FASTA*, and (3) *BLAST*.

The Smith-Waterman algorithm [25] uses dynamic programming to compute the most sensitive pairwise similarity alignments. However, these optimal computations require execution in order quadratic time.

The FASTA algorithm is an approximate heuristic algorithm used to compute suboptimal pairwise similarity comparisons. It was developed by Lipman and Pearson [19]. Although not as optimal as the Smith-Waterman algorithm, the FASTA algorithm nevertheless executes in more rapid time and thus provides a trade-off between comparison accuracy versus execution time.

The BLAST (basic local alignment search tool) algorithm, proposed by Altschul *et al.* [1], is another approximate heuristic algorithm used to compute suboptimal pairwise similarity comparisons. The BLAST algorithm is an improvement over the similar FASTA algorithm by offering three advantages. First, it provides more rapid

execution time. Second, its output includes a range of solutions. Finally, each reported match is accompanied by an estimate of statistical significance. Thus, the BLAST algorithm has become the dominant search engine for biological sequence databases [10]. Although BLAST does not allow alignments with gaps, it is possible to obtain most of the correct alignments while saving much of the computation time compared to the standard dynamic programming method.

1.3 Indexing Methods

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. The cost is the need for the time to build an index and for the space to store the index on disk. The potential saving is that based on the index, we can fetch a small number of sequences as likely answers from the database, thus it reduces both the number of disk I/O and the computation time required to resolve a query [29].

Given a set of database sequences, the well-known Wilbur-Lipman approach [27] is first preprocess, though hashing, each *interval* in the query sequence. An interval in this context is a fixed-length overlapping subsequence from a sequence; there are *(l - n + 1)* intervals for a sequence of length l and interval length n. In [29], they used an inverted index to select a subset of sequences in a *coarse search*. Figure 1.11 shows an example of the inverted index with $l = 5$ and $n = 23$.

The inverted index is a simple approach, and it uses hashing to store the intervals and search the query sequence. However, the inverted index may lose some information when the length of the query sequence is less than l. Moreover, the perfect hashing function is difficult to construct, and the collision in a hash table may occur frequently. Based on the idea that using the tree structure can provide better accuracy than hashing, Weiner proposed the *suffix tree* [26] that is a compact version of the *suffix tries*. It was well established in string processing and a good introduction to their use in biology is provided by Gusfield [9]. A suffix tree indexing a string of length n has n leaf nodes, one per suffix. Each edge is labelled with a non-empty

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	C	G	C	T	G	A	G	C	T	G	A	C	G	C	T	G	A	C	G	C	T	G

ACGCT (3: 0, 11, 17)
AGCTG (1: 6)
CGCTG (3: 1, 12, 18)
CTGAC (2: 8, 14)
CTGAG (1: 3)
GACGC (2: 10, 16)
GAGCT (1: 5)
GCTGA (3: 2, 11, 13)
TGACG (2: 9, 15)
TGAGC (1: 4)

Figure 1.11 An example of the inverted index with $l = 5$ and $n = 23$

substring, and at each branching node, the starting letters of the outgoing edges are different, so that each path from the root to a leaf spells the suffix that starts at the sequence position held in the leaf. Figure 1.12 shows an example of this data structure.

Since the size of genome sequence databases is increasing, the storage space of a suffix tree is also increasing. To reduce the storage space, Manber and Myers [20] proposed a new data structure, called the *suffix array*. Figure 1.13 shows an example of the suffix array. This data structure is basically a sorted list of all the suffixes of strings. They only store the sorted list. The main advantage of the suffix array over the suffix tree is that the suffix array uses three to five times less space.

To efficiently construct a suffix array is not an easy task. In [13, 14], Irving and Love proposed another new data structure, the *suffix binary search tree* (SBST for short), to index DNA sequences. It is an alternative construction method for suffix arrays. A suffix array can be efficiently constructed from a suffix binary search tree. The suffix binary search tree is also more space efficient than the suffix tree. Figure 1.14 shows an example of the suffix binary search tree. This data structure uses likely

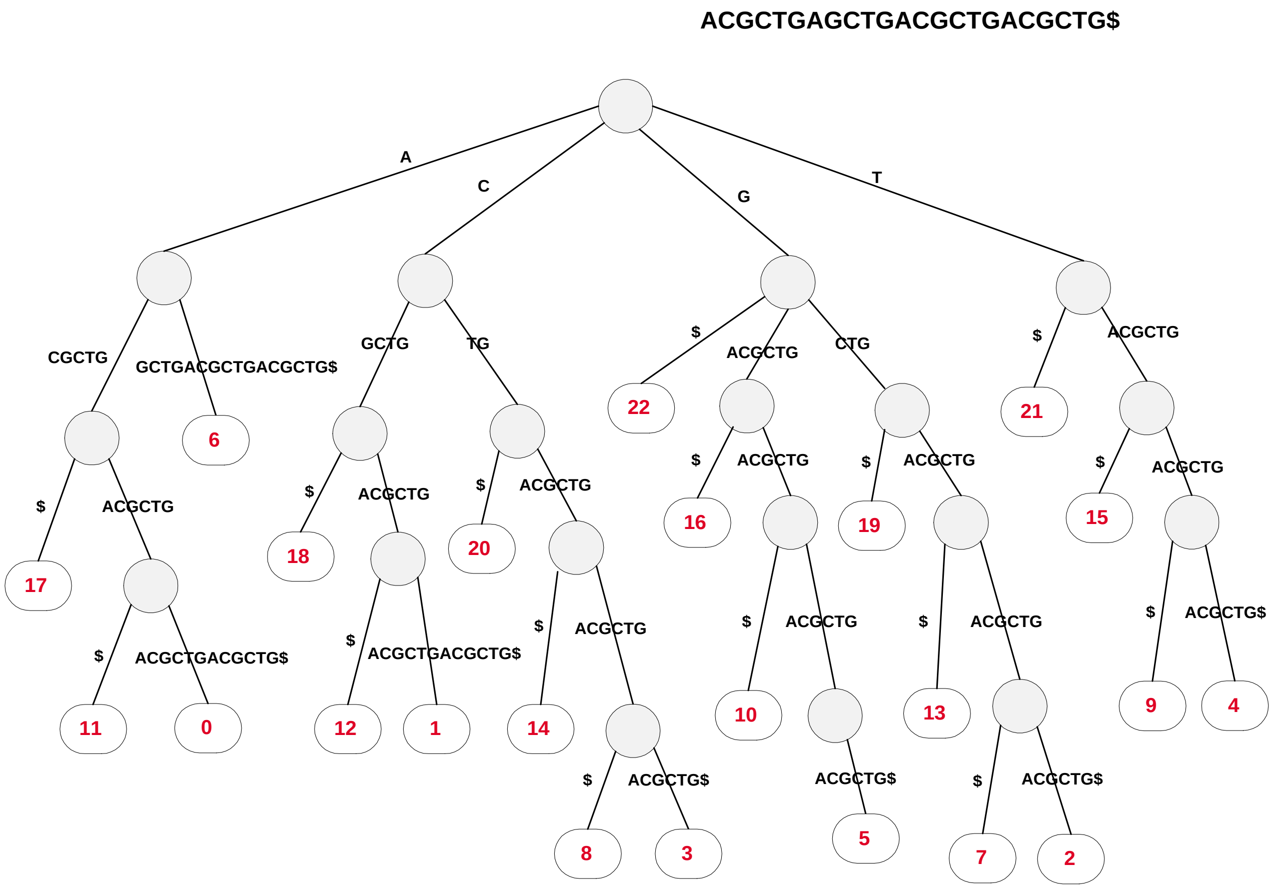


Figure 1.12 The suffix tree for the sequence ACGCTGAGCTGACGCTGACGCTG$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Position	17	11	0	6	18	12	1	20	14	8	3	22	16	10	5	19	13	7	2	21	15	9	4

Figure 1.13 The suffix array for the sequence ACGCTGAGCTGACGCTGACGCTG$

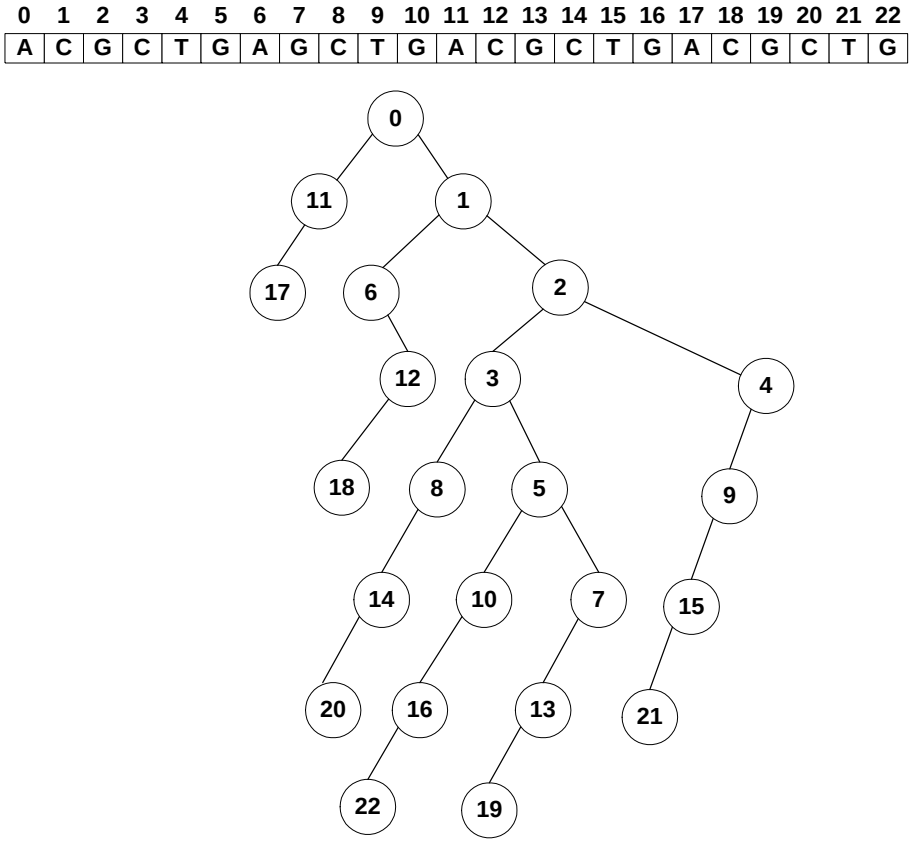


Figure 1.14 An example of the suffix binary search tree

the binary tree to construct and search the tree. The problem of the suffix binary search tree is its balance of the tree. If it is not balance, it will be affected the time for search.

For those approaches mentioned above, they store all suffixes of the sequences. They do not consider the concept of words. Andersson and Nilsson designed a data structure, the *word suffix trees* [3], to index a string of length n which has a natural partitioning into m multi-character substrings or words. This word suffix tree represented only the m suffixes that start at word boundaries. These boundaries are determined by delimiters $\flat$. Since traditional suffix tree construction algorithms rely heavily on the fact that all suffixes were inserted, construction of a word suffix

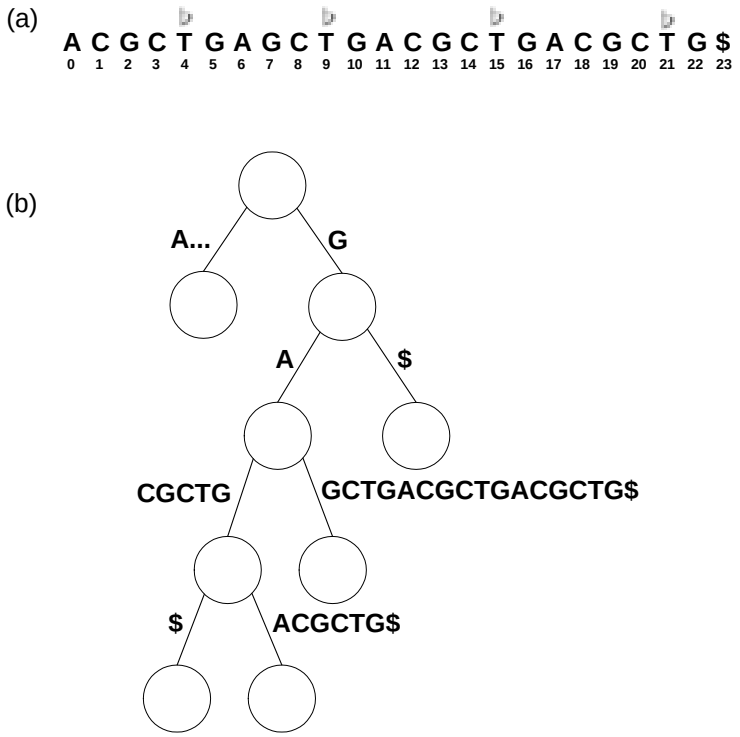


Figure 1.15 A sample sequence where $\flat = T$: (a) the input sequence and the position of the delimiter; (b) the word suffix tree.

tree stores only m' suffixes, where m' is the number of distinct words. Figure 1.15 shows an example of the word suffix tree, where $\flat = T$ and the words are $\{ACGCT$, $GAGCT, GACGCT, GACGCT, G\}$. (Note that, however, in the ACGT-Words tree, they still store the whole suffix word. For example, for the word $GAGCT$, they store $GAGCTGACGCTGACGCTG$ in Figure 1.15.)

1.4 Motivations

Because of the increase in the size of genome sequence databases, the search in genomic databases is an important problem. Although there have been several heuristic approaches [1, 19, 25] proposed to compare the query sequence to database sequences one base pair by one base pair, these methods still waste too much time to

do the search. Conventional databases use indexes to provide efficient access to the data. Therefore, recently, several indexing approaches [2, 3, 11, 14, 20, 26, 29] have been proposed for fast query processing for the genomic databases. In [26], Weiner proposed the suffix tree approach to index the DNA sequences. We can follow the tree structure to search the query sequence in the genomic databases. The suffix tree is a simple indexing structure for constructing and searching the DNA sequence. However, the disadvantage of the suffix tree is its large storage space. Another data structure to index the sequence is the suffix array. It is a lexicographically ordered array of the suffixes of the sequence. It reduces the storage space for indexing the DNA sequence. However, the construction and search of the suffix array wastes too much time, because it uses the binary search to construct the tree and search the query sequence.

For those approaches mentioned above, they insert all suffixes of the sequence. They do not break the sequence into words. Andersson *et al.* [3] proposed the word suffix tree. These trees store, for a sequence of length n in an arbitrary alphabet, only m suffixes that start at word boundaries. Although the word suffix tree stores only m suffixes, it will lose information in the search process. In Figure 1.15-(a), the sequence $TGAGC$ occurs at position 4. But in Figure 1.15-(b), we can not find this sequence in the tree structure.

In this thesis, 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 [3] to construct our index structure. The definition for an ACGT-Word is as follows: for a sequence beginning with a character k, $k \epsilon \{A,\ C,\ G,\ T\}$, the successive characters y_i after it until the same character k appearing again or the end symbol \$ appearing, the sequence constructed by k and y_i is an ACGT-Word. For example the DNA sequence $AGAGACT\$$, the related words are $\{AG,\ GA,\ AG,\ GACT,\ ACT,\ CT,\ T\}$. Given the same input DNA sequence as used in the previous examples, the words constructed from our approach are shown at the right part of Table 1.2. Figure 1.16 shows the ACGT-Words tree index structure constructed by the related words. Although a word may occur at many positions in the sequence, we only construct a

word once in the tree structure. In Figure 1.16, the word *ACGCTG* occurs only once in the tree structure, although this word occurs three times, positions 0, 11, and 17, in the sequence. Hence, we can reduce the storage space needed by the suffix tree. In Table 1.3, the number of nodes is 28 in the ACGT-Words tree as compared to 44 in the suffix tree. The number of edges is 26 in the ACGT-Words tree as compared to 42 in the suffix tree. The number of levels is 6 in the ACGT-Words tree as compared to 5 in the suffix tree. We can see that the ACGT-Word tree needs less storage space than the suffix tree. On the other hand, the suffix array takes long time in construction and search. That is, the ACGT-Words tree has better performance than the suffix array in terms of the construction time and the query processing time. Table 1.4 shows a comparison of the suffix array and our ACGT-Words tree structure. Although the number of words generated by the word suffix tree is less than that of our index structure, as shown in Table 1.2, the word suffix tree loses some information when we search in this structure. Consequently, in this thesis, we propose a new index structure to provide the same query results of the suffix-tree-like index structures, while it avoids the disadvantages of those approaches.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives a survey of several well-known data structures for indexing the DNA sequence, including *inverted indexes*, *suffix trees*, *suffix arrays*, *suffix binary search trees*, and *word suffix trees*. Chapter 3 presents the proposed ACGT-Words tree index structure. In Chapter 4, we study the performance and make a comparison of our approach with the suffix tree and the suffix array. Finally, Chapter 5 gives the conclusion.

Table 1.2 A comparison of the words that constructed from the word suffix tree and the ACGT-Words tree

	Word suffix tree	ACGT-Words tree
1	ACGCT	ACGCTG
2	GAGCT	AGCTG
3	GACGCT	CG
4	G$	CTG
5		CTGA
6		CTGAG
7		G
8		GA
9		GAC
10		GCT
11		TG
12		TGACGC
13		TGAGC

Table 1.3 A comparison of the suffix tree and the ACGT-Words tree

	node	edge	level
Suffix Tree	44	42	6
ACGT-Words tree	28	26	5

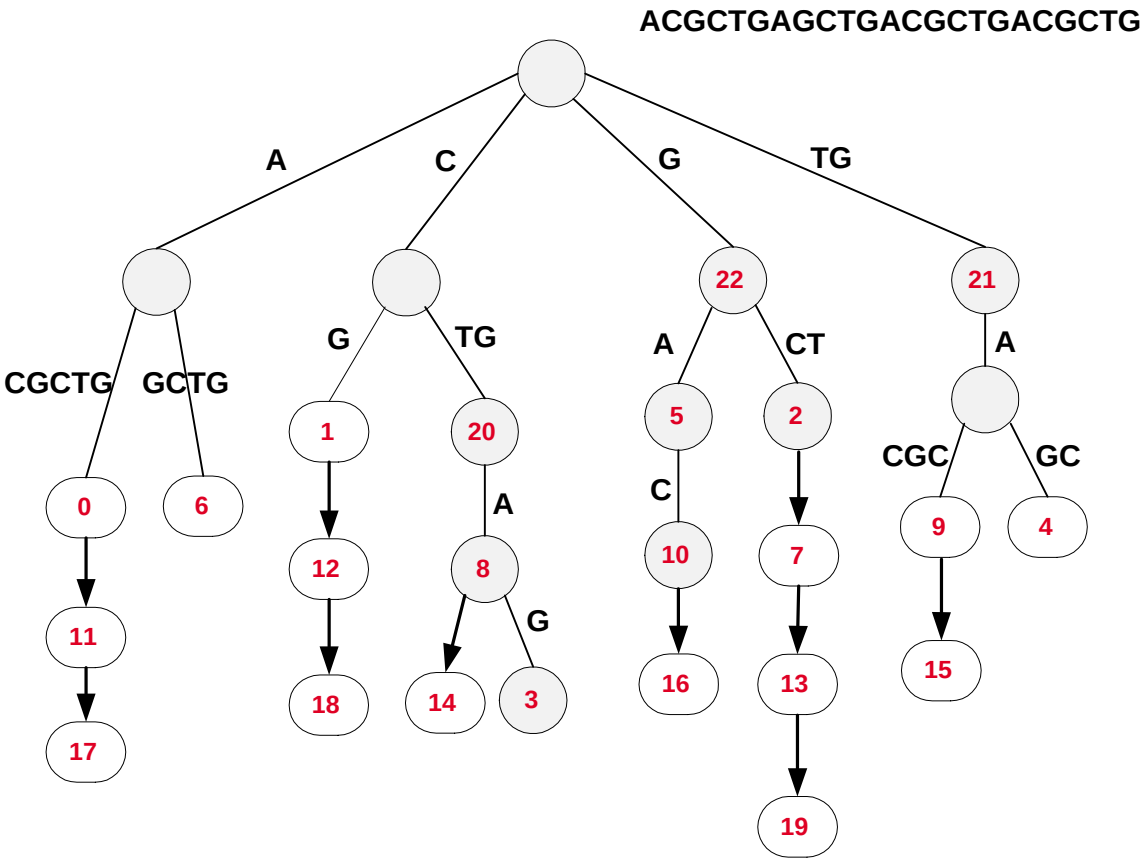


Figure 1.16 The ACGT-Words tree for the sequence ACGCTGAGCTGACGCT-GACGCTG

Table 1.4 A comparison of 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

CHAPTER II

A Survey

In this Chapter, we will give a survey of some well-known indexing techniques for supporting genomic databases, including the inverted indices [1, 19, 29], suffix tries [26], suffix trees [26], suffix arrays [20], suffix binary search trees [13, 14], and word suffix trees [3].

2.1 Inverted Indices

An inverted index is a word-oriented mechanism for indexing a text collection in order to speed up the searching task. The inverted index structure is composed of two elements: a search structure and postings lists. The search structure is the set of all different words in the text. For each such a word, a list of all the text positions where the word appears is stored. The set of all those lists is called the posting lists. Figure 2.1 shows an example.

Inverted files have been shown to be a successful tool for large text database retrieval. But in DNA sequences, we can not break sequences into words. In [29], they suggest that an appropriate choice of an index term is the intervals occurring in each sequence, where the intervals are overlapping substrings of some fixed-length n. Figure 2.2 shows an example of fixed-length 3. Fixed-length overlapping intervals have been shown to be practical in exhaustive search systems [1, 19].

An inverted index also has two components: a search structure and postings lists. Figure 2.3 shows an example. The search structure consists of the set of unique searchable terms, in this case, the set of intervals; while associated with each term in the search structure is a postings list. The postings list contains the ordinal numbers

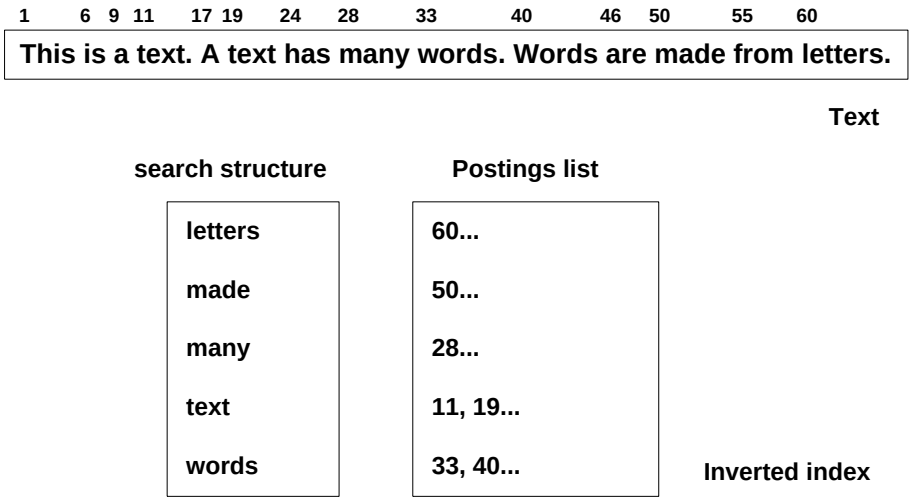


Figure 2.1 A sample text and the related inverted index

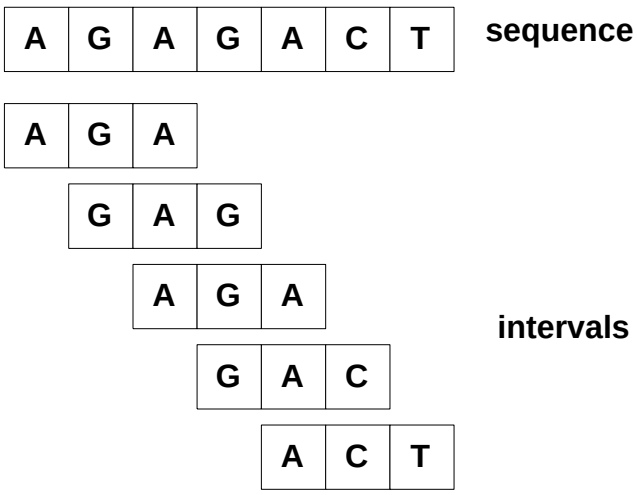


Figure 2.2 $T = AGAGACT$ and the related 3-gram intervals

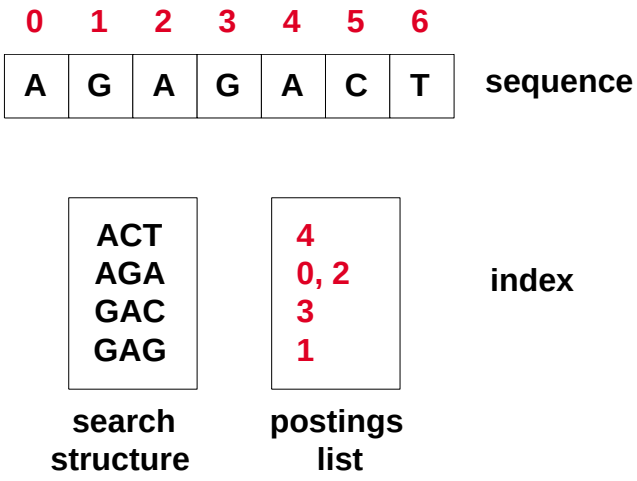


Figure 2.3 $T = AGAGACT$ and its related inverted index

of the documents containing the associated search term. Each postings list of the inverted index approach is stored not only the ordinal sequence number that contains the interval, but also offset information. For example, consider the following postings list:

$$ACCC\ 12, (3 : 144, 154, 962),\ 38, (2 : 47, 1045)...$$

in which the indexed sequences, the 12th and 38th, contain the interval $ACCC$. The interval occurs three times in the 12th sequence, at offsets 144, 154, and 962, and twice in the 38th sequence, at offsets 47 and 1,045 [29].

2.2 Suffix Tries

Let $T = t_1 t_2 ... t_n$ be a string over alphabet Σ. A *substring* of T is a string $T_i^j = t_i t_{i+1} ... t_j$ for some $1 \leq i \leq j \leq n$. The string $T_i = T_i^n = t_i ... t_n$ is a *suffix* of string T and the string $T^j = T_1^j = t_1 ... t_j$ is a *prefix* of the string T.

A *trie* is a rooted tree with the following properties:

1. Each node, except the root, *contains* a symbol of the alphabet.

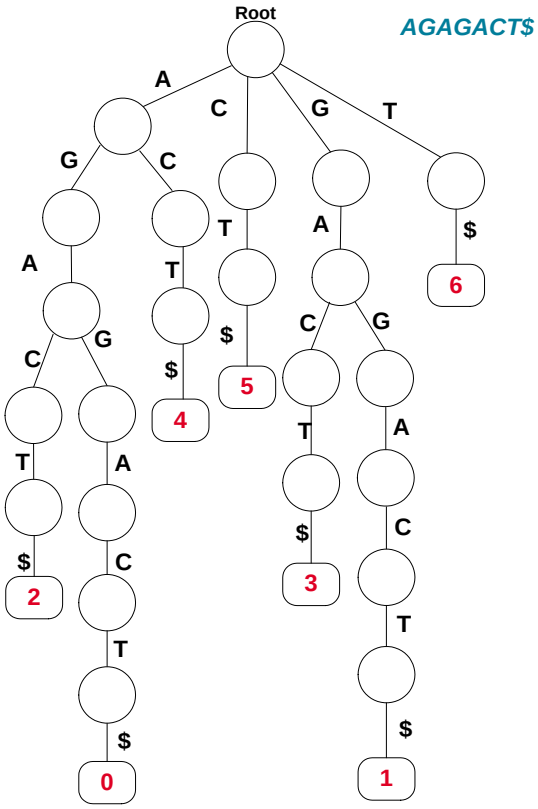


Figure 2.4 The suffix tries for $AGAGACT\$$

2. No two children of the same node contain the same symbol.

The *suffix trie* $ST_r(T)$ of text T is a trie whose leaves represent the suffixes of T. The nodes of $ST_r(T)$ represent exactly the set of substrings of T, because every substring of the text is a prefix of some suffix, i.e. $T_i^j = (T_i)^j$. An example suffix trie, for the string $AGAGACT\$$, is shown in Figure 2.4.

When we look for the string ACT, we start at the root and we follow the A edge, followed by the C edge and finally the T edge. We can find position 4 of the string has ACT. The above structure is rather wasteful because each edge represents a single symbol. For huge texts, this is an enormous waste of space.

2.3 Suffix Trees

The *suffix tree* proposed by Weiner [26] is a compact version of the suffix tries. It is formed by catenating each unary node (a node with exactly one child) with its child.

2.3.1 The Definition

A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 0 to $m-1$. Each internal node, other than the root, has at least two children and each edge is labelled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i [9].

2.3.2 Construction

Let T be a string of m characters. Let T_i denote the suffix $T[i..m-1]$ of T for $0 \leq i \leq m-1$. We present a straightforward algorithm to build a suffix tree for string T. The first step inserts a single edge for suffix T_0 into the tree; then it successively enters suffix T_i into the growing tree, for i from 1 to $m-1$.

In detail, we let T_i denote the suffix $T[i..m-1]$. Table 2.1 shows the all suffixes of the sequence T. T_0 consists of a single edge between the root of the tree and a leaf labelled 0. The edge is labelled with the string $T\$$. Starting at the root of T_i find the longest path from the root whose label matches a prefix of T_{i+1}. This path is found by successively comparing and matching characters in T_{i+1} to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character [9, 18, 26].

A suffix tree of T of length m is a tree with the following properties:

Table 2.1 Suffixes for T ="AGAGACT$"

AGAGACT$	T_0
GAGACT$	T_1
AGACT$	T_2
GACT$	T_3
ACT$	T_4
CT$	T_5
T$	T_6

1. Each tree edge is labelled by a substring of S.
2. Each internal node has at least 2 children.
3. The number of leaves is m.
4. Each suffix has its unique leaf.

2.3.3 Searching

Given a pattern P of length n and a text T of length m, we match the characters of P along the unique path in $\mathcal{T}$ until either P is exhausted or no more matches are possible. In the latter case, P does not appear anywhere in T. In the former case, every leaf in the subtree below the point of the last match is numbered with a starting location of P in T, and every starting location of P in T numbers such a leaf [9].

For example, Figure 2.5 shows the suffix tree for string $T = AGAGACT\$$. Pattern $P = AGA$ appears two times in T starting at locations 0 and 2.

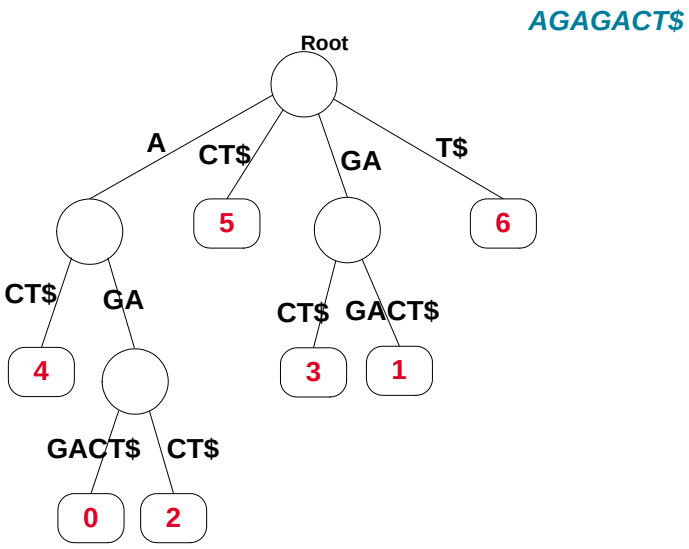


Figure 2.5 The suffix tree of $T =$ "$AGAGACT\$$"

2.4 Suffix Arrays

A *suffix array* [20] is an alternative data structure to the suffix tree. This new data structure is very space efficient and can be used to solve the exact matching problem almost as efficiently as with a suffix tree.

2.4.1 The Definition

Let $T = a_0a_1...a_{m-1}$ be a large text of length m, a *suffix array* for T, called Pos, is an array of the integers in the range 0 to $m-1$, specifying the lexicographic order of the m suffixes of string T. Figure 2.6 is an example.

The basic idea of the suffix array data structure is a lexicographically sorted array, Pos, of the suffixes of T. That is, the suffix starting at position $Pos[0]$ of T is the lexically smallest suffix, and in general, suffix $Pos[k]$ of T is lexically smaller than $Pos[k+1]$. As usual, we will affix a terminal symbol \$ to the end of T, we interpret it to be lexically less than any other character in the alphabet. For example, if T is

$AGAGACT\$$, then the suffix array Pos is 2, 0, 4, 5, 3, 1, 6. Figure 2.7 shows this suffix array of T [9, 14, 18, 20].

2.4.2 Construction

A suffix array for T can be constructed by the lexical depth first searching in the suffix tree for T. We start the search from the root node x. Select an unvisited node y branched from x such that the label of edge linking x and y has the smallest alphabetical ordering among edges linking unvisited nodes and x. When a node z is reached, we back up to the last vertex visited and continue the depth first search until all nodes of the suffix tree are visited. Suffix array Pos is the ordered list of suffix numbers encountered at the leaves of a suffix tree during the lexical depth first search.

2.4.3 Searching

The suffix array for string T allows a very simple algorithm to find all occurrences of any pattern P in T. The key idea is that if P occurs in T, then all the locations of those occurrences will be grouped consecutively in Pos. For example, $P = AGA$ occurs in $AGAGACT\$$ starting at locations 0 and 4, which are indeed adjacent in Pos. So, to search for occurrences of P in T, simply, we can do the binary search over the suffix array [9]. In more detail, suppose that P is lexically less than the suffix in the middle position of $Pos[m/2]$. In that case, the first place in Pos that contains a position where P occurs in T must be in the first halt of Pos. Similarly, if P is lexically greater than suffix $Pos[m/2]$, then the places where P occurs in T must be in the second half of Pos. Using the binary search, one can therefore find the smallest index i in Pos such that P exactly matches the first n characters of suffix $Pos[i]$. Similarly, one can find the largest index i' with that property. Then, pattern P occurs in T starting at every location given by $Pos[i]$ through $Pos[i']$.

i	0	1	2	3	4	5	6
Pos	2	0	4	5	3	1	6

Figure 2.6 The suffix array A for $T =$ "$AGAGACT\$$"

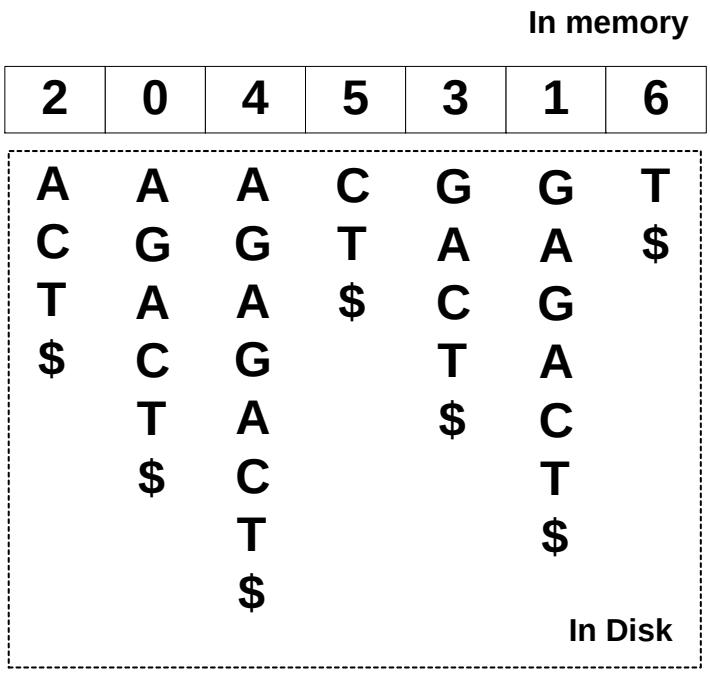


Figure 2.7 Sort the suffixes and store all the indices in an array

2.5 Suffix Binary Search Trees

The suffix binary search tree (SBST) is a new data structure [13, 14] with potential for string indexing. It is more space efficient than the suffix tree. It can be built in $O(nh)$ time, where h is the height of the tree and n the string length, and can be searched in $O(m+l)$ time, where m is the length of the query and l the length of the path traversed in the tree.

Given a string $\sigma = \sigma_1\sigma_2...\sigma_n$ of length n, a suffix binary search tree for σ is a binary tree containing n nodes, each labelled by a unique integer in the range $1...n$, the integer i representing the ith suffix $\sigma^i = \sigma_i\sigma_{i+1}...\sigma_n$ of σ. They refer to the node representing suffix σ^i simply as node i of the tree. Furthermore, the tree is structured so that, for each node i, σ^i is lexicographically greater than σ^j for every node j in its left subtree, and lexicographically less than σ^k for every node k in its right subtree.

2.6 Word Suffix Trees

Since traditional suffix tree construction algorithms rely heavily on the fact that all suffixes are inserted. Andersson *et al.* [3] presented *word suffix tree*. These trees store, for a string of length n in an arbitrary alphabet, only m suffixes that start at word boundaries.

2.6.1 The Definition

The definition of a word suffix tree is given an input string consisting of n characters from an alphabet of size k, including two, possible implicit, special characters \$ and ♭. The \$ character is an end marker which must be the last character of the input string and may not appear elsewhere, while ♭ represents some *delimiting character* which appears in m - 1 places in the input string. Andersson *et al.* regarded the input string as a series of words–the m nonoverlapping substrings ending either with \$ or ♭. There may of course exist multiple occurrences of the same word in the input

string, They denote the number of *distinct* words by m'. There are no empty words in their algorithm; the shortest possible word is a single \$ or $\flat$. They want to create a trie structure containing m strings, namely the suffixes of the input string that start at the beginning of words [3].

2.6.2 Construction

They present three different algorithms to construct the word suffix tree, the results of these algorithms are same. We introduce one of these algorithms–Algorithm B. The stages of the algorithm are outlined as follow:

1. Building the word trie.
2. Assigning in-order numbers.
3. Generating a number string.
4. Constructing the number-based suffix tree.
5. Expanding the number-based suffix tree.

Figure 2.8 and Figure 2.9 show the steps of constructing the word suffix tree. Figure 2.10 shows another example of the word suffix tree.

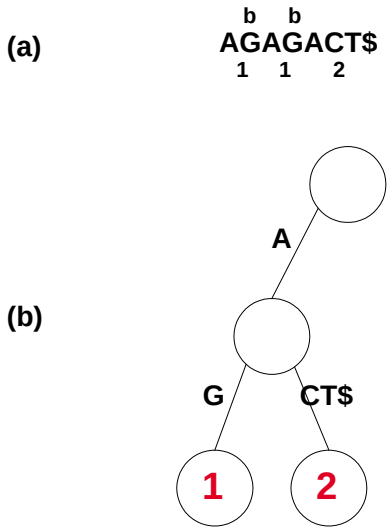


Figure 2.8 A sample string where $\flat = G$: (a) its number string; (b) its corresponding word trie.

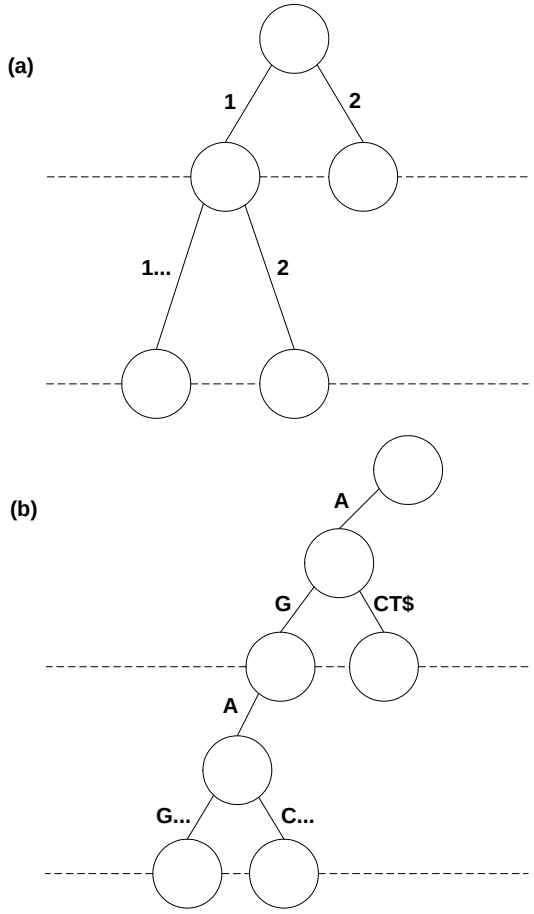


Figure 2.9 Two steps of constructing data structure: (a)The number suffix tree; (b) its corresponding word trie. (Dotted lines denote corresponding levels.)

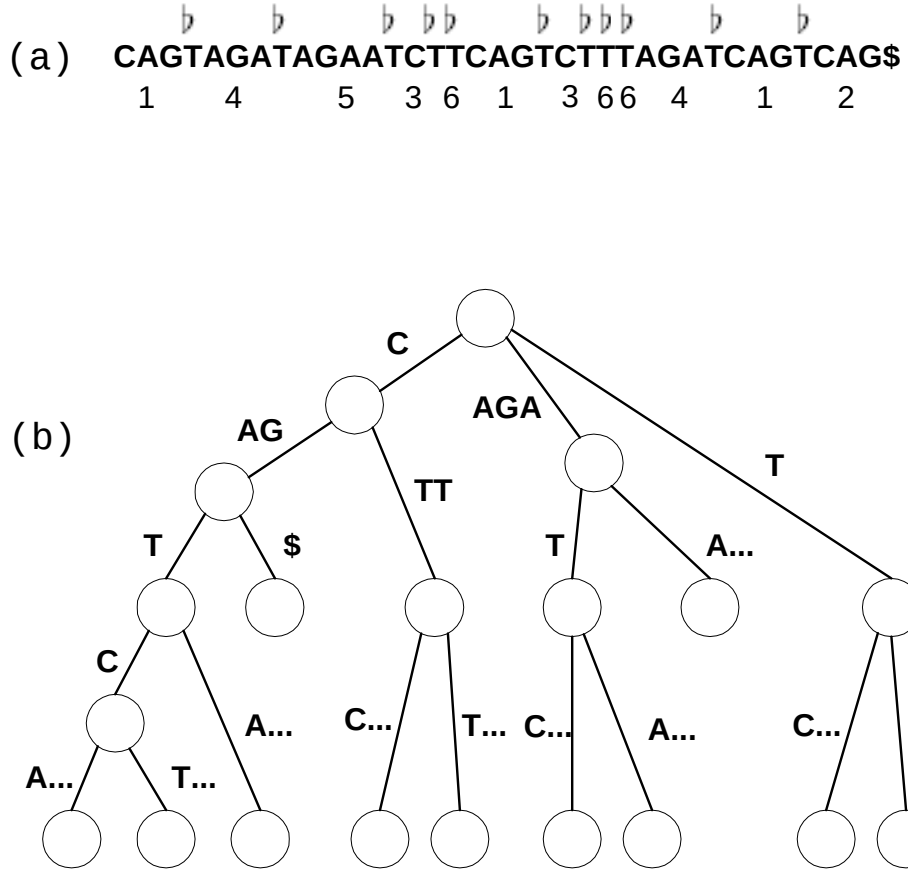


Figure 2.10 A sample string where $\flat = T$: (a) its number string; (b) the word suffix tree.

CHAPTER III

The ACGT-Words Tree

In the genomic databases, the suffix tree index structure is used widely. Based on the suffix tree, there are another data structures proposed to index the genomic databases, such as the suffix array, the suffix binary search tree, and so on. However, these approaches insert all suffixes of the sequence. They do not have the concept of words. Although the word suffix tree uses the concept of words and reduces the storage space, they lose information in the search process. In this Chapter, we define the 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.

3.1 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 $j = n-1$. For example, given the DNA sequence $AGAGACT\$$, the related words are $\{AG, GA, AG, GACT, ACT, CT, T\}$.

3.2 Tree Construction

In this section, we illustrate how to construct the ACGT-Words tree for the given sequence S. We have two procedures, $Split(S)$ and $Insert(R, W, index)$, in the process of tree construction, where S is the input sequence for constructing the tree; W

is the ACGT-Word; R is the node which the word W is inserted; $index$ denotes the starting position of the ACGT-Word W. First, we split the sequence S into words, though Procedure $Split(S)$. Procedure $Split(S)$ is shown in Figure 3.1. Figure 3.2 shows the flowchart for splitting the sequence S into words. For example, for the given input sequence $ATACACGAT\$$ (S) as shown in Figure 3.3-(a), Figure 3.3-(b) shows the process of generating the ACGT-Words.

Assume that P is an index of the sequence S and $Pos[P]$ denotes the position of the index. 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 3.3-(b). We find the first character of the sequence S is A, then we check whether the variable A_num is -1 or not. If A_num is -1, we change the variable A_num to the value of $Pos[P]$. That is, the value of A_num is 0 now. Otherwise, a word will be generated. After changing the value of A_num, the index P points to the next character of the sequence S. Because the character pointed is T and the value of T_num is -1, we change the value of T_num to 1, and then P points to the next character. Now, P is at position 2, and the character is A. Since variable A_num is 0, not -1, which implies that we find a word which locates from A_num to ($Pos[P]$ - 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. Hence, we generate the remaining ACGT-Words W_5, W_6, W_7, W_8, as shown in Table 3.1.

During the processing of words generation, words are generated in procedure $Split(S)$. After generating words, we call procedure $Insert(R, W, index)$ to construct the ACGT-Words tree. Next, we describe how to construct the ACGT-Words tree. The node data structure and edge data structure of the ACGT-Words tree are shown as follows:

Table 3.1 The ACGT-Words for the given sequence $ATACACGAT\$$

	ACGT-Words
W_0	AT
W_1	AC
W_2	CA
W_3	ACG
W_4	TACACGA
W_5	AT
W_6	CGAT
W_7	GAT
W_8	T

```
struct Node
{
  int id;
  int suffix_num;
}
struct Edge
{
  String label;
  int first_index;
  int last_index;
  Node parentnode;
  Node endnode;
}
```

Formally, procedure $Insert(R, W, index)$ in the ACGT-Word tree is shown in Figure 3.4 and Table 3.2 shows the variables used in procedure $Insert(R, W, index)$.

```
Procedure Split(S);
/* Generate the words of input sequence S. */
/* P is an index of the input sequence S. */
/* W stores an ACGT-Word. */
/* Substring(S, Ns, Ne) is a function which returns subsequence of S which starts from Ns and ends at Ne. */
        begin
            P := 0;
            while ( S[P] ≠ '$') do
            begin /* Create a new Node and a new Edge */
                if ( S[P] = 'A' ) then
                begin
                    if ( A_num ≠ −1 ) then
                    begin
                        W := Substring(S, A_num, (P − 1));
                        Insert(R, W, A_num);
                    end;
                    A_num := P;
                end
                else if ( S[P] = 'C' ) then
                begin
                    if ( C_num ≠ −1 ) then
                    begin
                        W := Substring(S, C_num, (P − 1));
                        Insert(R, W, C_num);
                    end;
                    C_num := P;
                end
                else if ( S[P] = 'G' ) then
                begin
                    if ( G_num ≠ −1 ) then
                    begin
                        W := Substring(S, G_num, (P − 1));
                        Insert(R, W, G_num);
                    end;
                    G_num := P;
                end
                else ( S[P] = 'T' ) then
                begin
                    if ( T_num ≠ −1 ) then
                    begin
                        W := Substring(S, T_num, (P − 1));
                        Insert(R, W, T_num);
                    end;
                    T_num := P;
                end;
                P := P + 1;
            end;
            InsertRemainingWord();
        end;
```

Figure 3.1 Procedure $Split(S)$

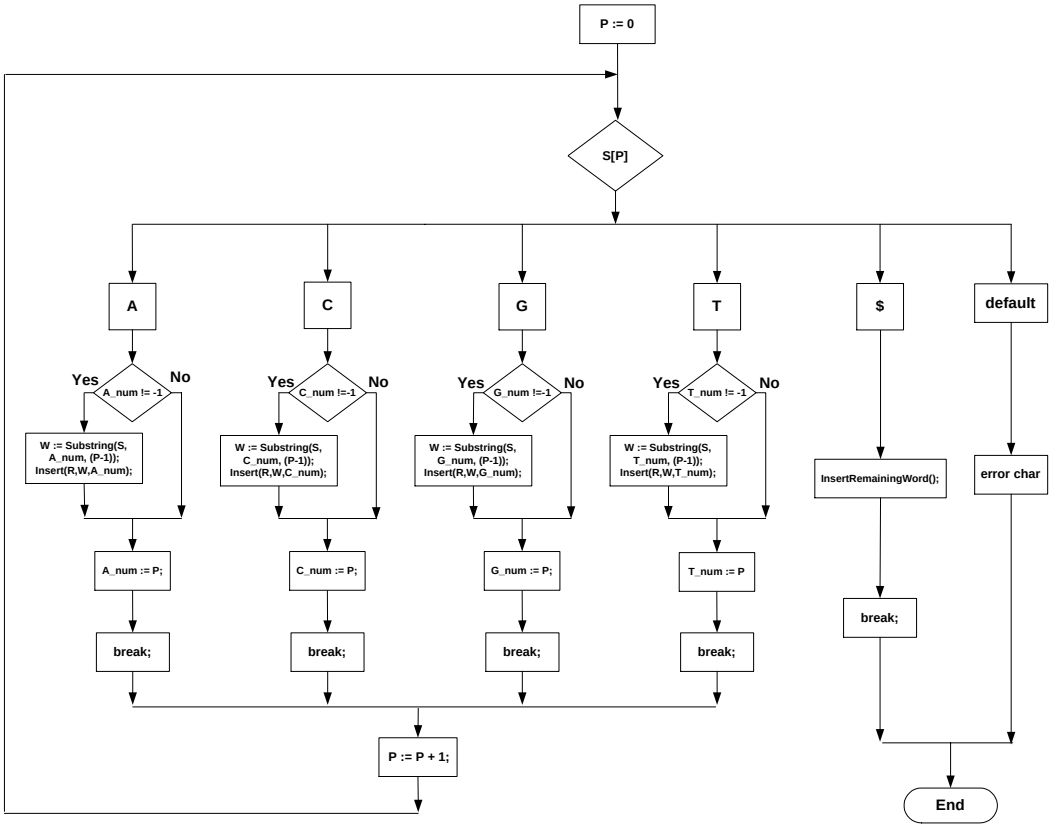


Figure 3.2 The flowchart for splitting the given sequence into the ACGT-Words

Table 3.2 Variables used in procedure $Insert(R, W, index)$

i	a character in $\{A, C, G, T, s\}$
N	a node in the ACGT-Words tree
R	a root which we traverse now
W	the input word for inserting
k	an integer stores the result of the $CompareSameEdge$ function
$index$	the position of the word that occurs in the sequence
$R.Edge_i$	the edge with a label in which the starting character is i

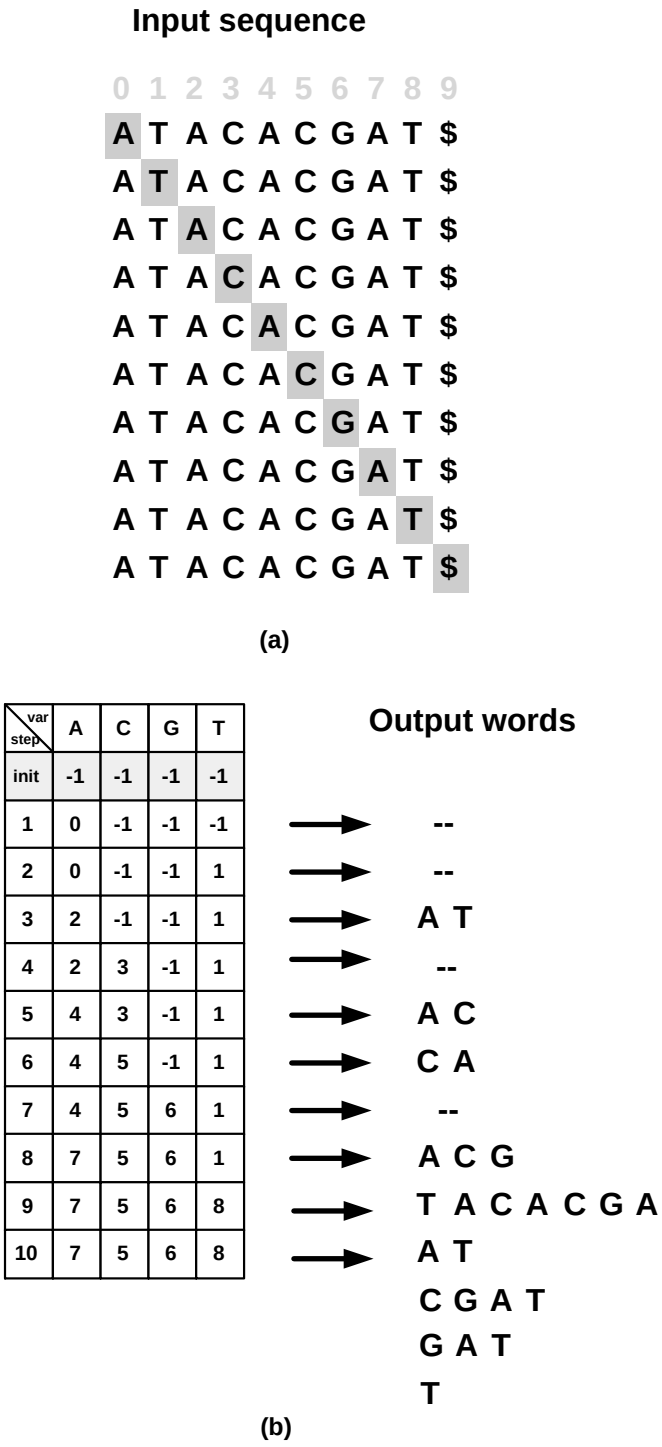


Figure 3.3 The steps for splitting the ACGT-Words: (a) scan input sequence; (b) change the variables and generate the words.

Table 3.3 Five cases in insertion

$k = 0$	$k > 0$			
	$k < \|W\|, \|E.label\|$	$\|W\| = k < \|E.label\|$	$\|W\| > k = \|E.label\|$	$\|W\| = \|E.label\| = k$
Case 1	Case 2	Case 3	Case 4	Case 5

Figure 3.5 shows the flowchart for inserting. According to different words constructed from splitting process, there are different cases in the constructing process. Before we describe those cases in detail, let's see some definitions. Let $e_0e_1...e_{n-1}$ be the label in an edge E. Let $|E.label|$ be the length of $E.label$. Let $W = w_0w_1...w_{m-1}$ be an input word. Let $|W|$ be the length of word W. Let k be the length of the same prefix characters between word W and $E.label$. Table 3.3 shows the conditions of five cases in procedure $Insert(R, W, index)$ and the related semantics is illustrated in Figure 3.6. The five different cases of procedure $Insert(R, W, index)$ are described as follows:

Case 1: This case occurs when $k = 0$. That is, there is no common prefix sequence between W and $E.label$. It means that the tree structure has no edge with a label that the starting character is w_0. Hence, we create a new node N by calling function $CreateNode(index)$. Then, we create a new edge E with a label W from node R to node N. Figure 3.7-(a), Figure 3.7-(c), and Figure 3.7-(e) illustrate this case. For example, in Figure 3.7-(a), an edge labelled with AT $(= W)$ is created between the new node 0 $(= N)$ and the root R.

Case 2: This case occurs when $k < |W|$, and $k < |E.label|$, where $k > 0$. That is, the common prefix sequence between W and $E.label$ is not exactly equal to W and $E.label$. According to the value of k and edge E, the $SplitEdge(E, j)$ function splits edge E into two edges, E_0, E_1, and creates a new node N to connect edges E_0 and E_1, such that $E_0.label = e_0e_1...e_{k-1}$ and $E_1.label = e_ke_{k+1}...e_{n-1}$. For example, Figure 3.7-(a) shows the tree before splitting. The new word $W = AC$ is inserted, it sets $k = 1$ and E with a label AT $(E.label = AT)$

```
Procedure Insert(R, W, index);
/* Insert the words into the ACGT-Words tree. */
/* CompareSameEdge(W1, W2) is a function that returns the common length between W1 and W2. */
/* First(W) is a function that returns the first character of W. */
/* Shift(W, com_len) is a function that returns the resulting W from com_len to the ending position of the word. */
/* SplitEdge(Edge, length) is a function that returns a new node after splitting. */
/* Sibling() is a function that returns the character s which labels the edge between two same words. */
/* CreateNode(index) is a function that returns a new node whose suffix_num is index. */
        begin
            i := First(W) ;
            if ( R.Edge_i.label = NULL ) then
            /* Case 1 */
            begin
                N := CreateNode(index);
                R.Edge_i.label := W;
                R.Edge_i.endnode := N;
            end
            else
            begin
                k := CompareSameEdge(R.Edge_i.label, W);
                if ( k < length(R.Edge_i.label) ) then
                begin
                    R := SplitEdge(R.Edge_i, k);
                    if ( k < length(W) ) then
                    /* Case 2 */
                    begin
                        W := Shift(W, k);
                        Insert(R, W, index);
                    end
                    else
                    /* Case 3 */
                        R.suffix_node := index;
                end
                else if ( k = length(R.Edge_i.label) and length(R.Edge_i.label) < length(W) )
                /* Case 4 */
                begin
                    R := R.Edge_i.endnode;
                    W := Shift(W, com_len);
                    Insert(R, W, index);
                end
                else
                /* Case 5 */
                begin
                    i := Sibling();
                    while ( R.Edge_i.label = NULL ) do
                        R := R.Edge_i.endnode;
                    N := CreateNode(index);
                    R.Edge_i.ednnode := N;
                end;
            end;
        end;
```

Figure 3.4 Procedure $Insert(R, W, index)$

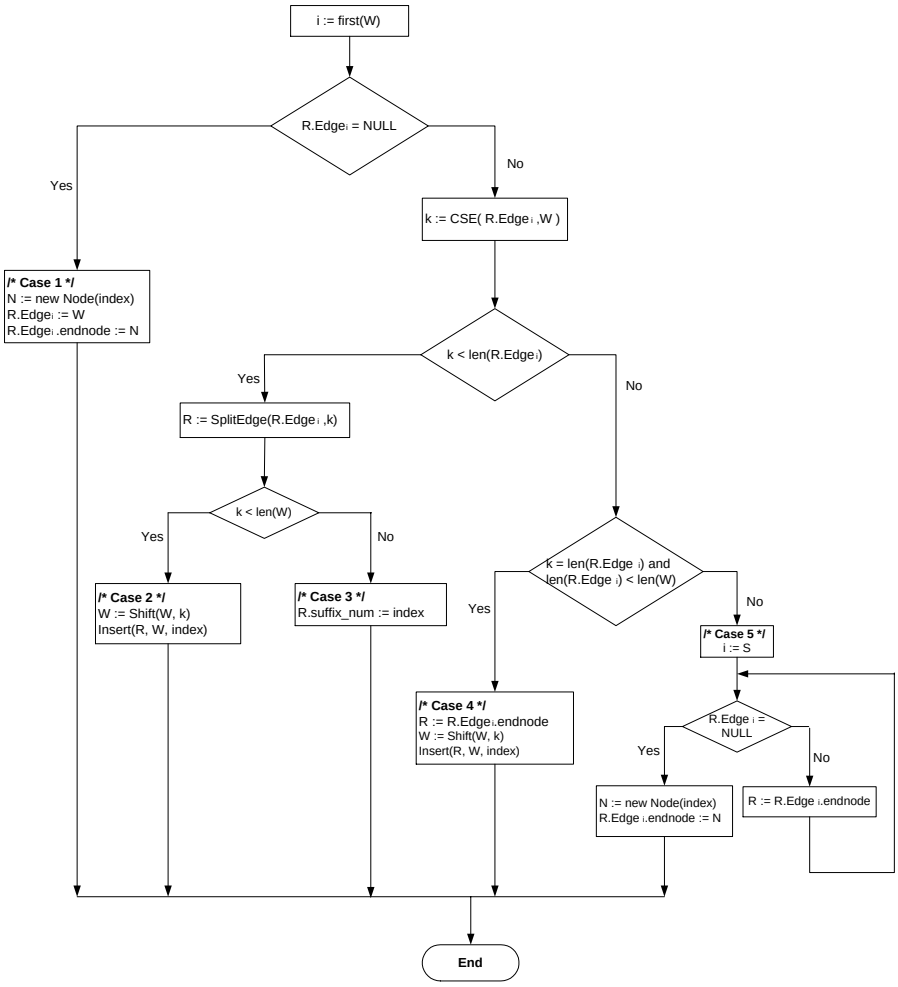


Figure 3.5 The flowchart of procedure $Insert(R, W, index)$

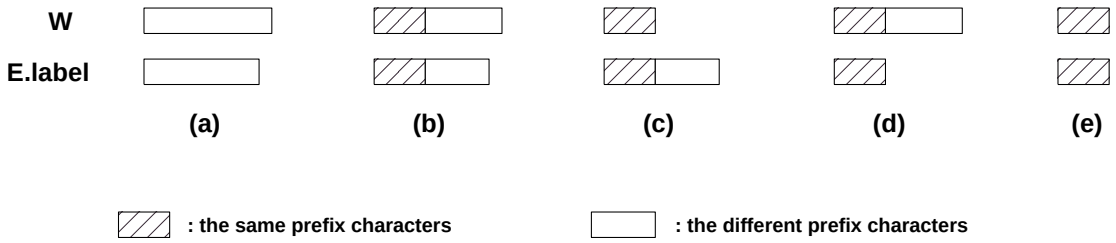


Figure 3.6 Fives cases in insertion: (a) Case 1; (b) Case 2; (c) Case 3; (d) Case 4; (e) Case5.

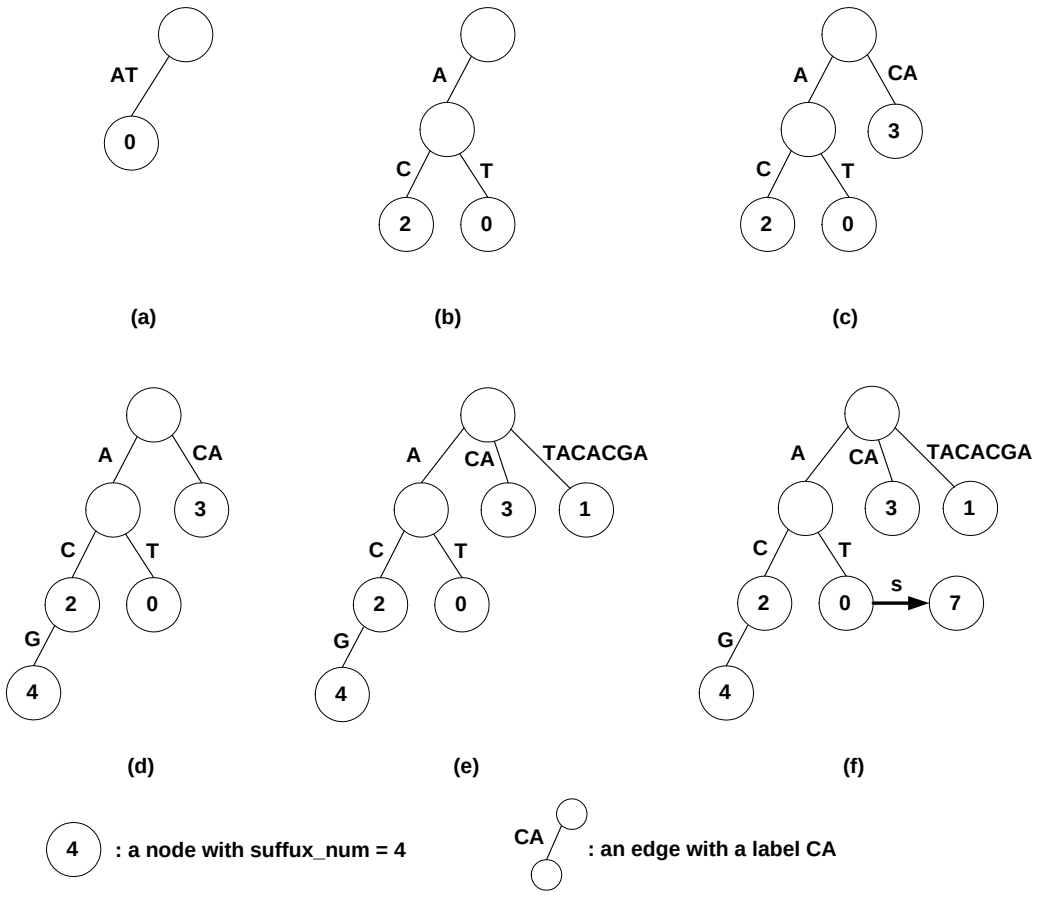


Figure 3.7 An example of insertion: (a) AT; (b) AC; (c) CA; (d) ACG; (e) TACACGA; (f) AT.

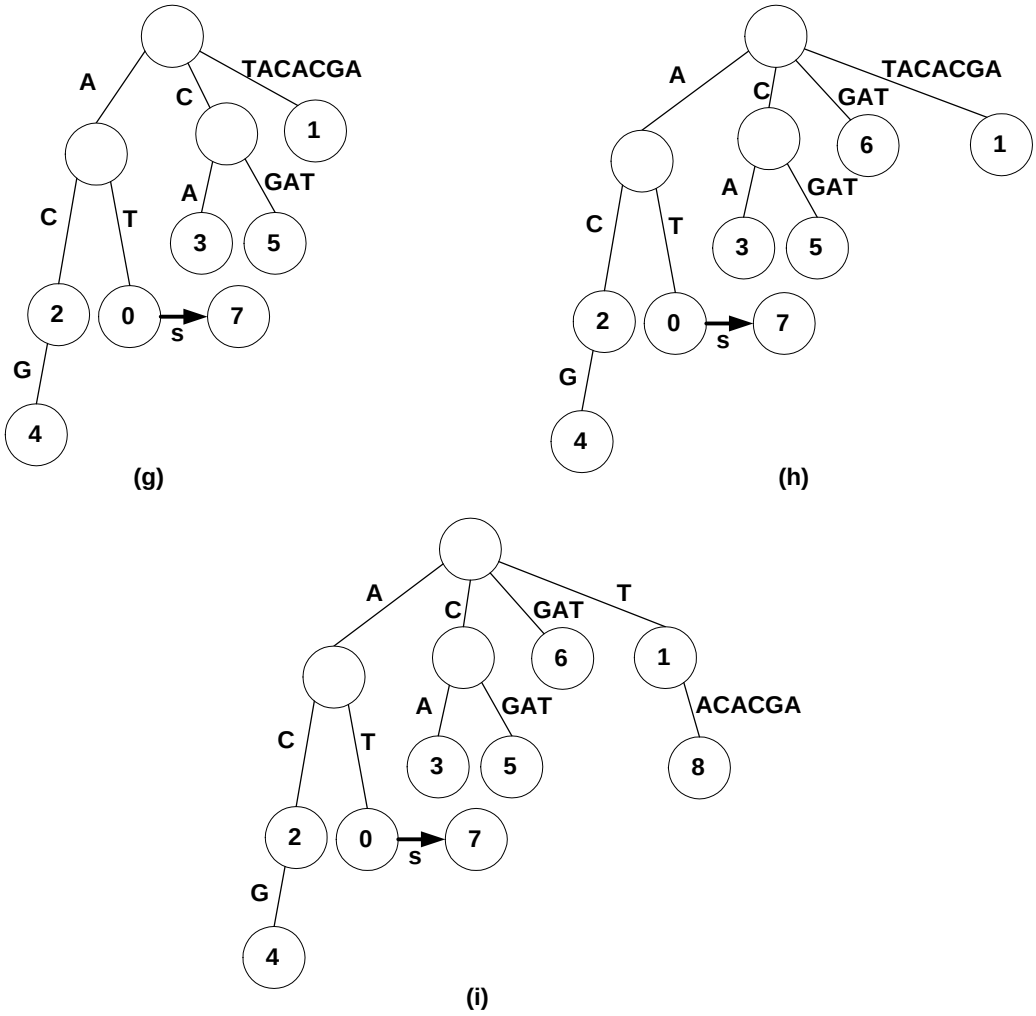


Figure 3.8 An example of insertion (continued): (g) CGAT; (h) GAT; (i) T.

is split into E_0 with a label A ($E_0.label = A$) and edge E_1 with a label T ($E_1.label = T$). The $SplitEdge(E, j)$ function returns a new node N between edge E_0 and edge E_1, i.e., $E_0.endnode = E_1.parentnode = N$. We use the $Shift(S)$ function to cut the same characters, i.e., $w_0w_1...w_{k-1}$ and the resulting $W = w_kw_{k+1}...w_{m-1}$. Hence, the $Shift(S)$ function cuts the character A from the word AC and the resulting $W = C$ is inserted at the new node N constructed from the $SplitEdge(E, j)$ function. Figure 3.7-(b) shows the result after the word $W = AC$ is inserted.

Case 3: This case occurs when $|W| = k < |E.label|$, where $k > 0$. That is, word W is the prefix sequence of $E.label$ (i.e., $E.label = We_ke_{k+1}...e_{n-1}$). Edge E in the tree is split into edge E_0 with a label $e_0e_1...e_{k-1}$ and edge E_1 with a label $e_ke_{k+1}...e_{n-1}$. The new node N is created by the $SplitEdge(E, j)$ function such that $E_0.endnode = E_1.parentnode = N$. Then, we set $N.suffix\_num = index$. For example, in Figure 3.8-(i), when an input word $W = T$ is inserted into the tree, an edge E with $E.label = TACACGA$ will be split into edge E_0 with $E_0.label = T$ and edge E_1 with $E_1.label = ACACGA$. And the new node N is created by $SplitEdge(E, 1)$. Then, the $suffix\_num$ of the new node N is set to 1.

Case 4: This case occurs when $|W| > k = |E.label|$, where $k > 0$. That is, $E.label$ is the prefix sequence of word W but not equal to word W (i.e. $W = E.labelw_kw_{k+1}...w_{m-1}$). First, we call the $Shift(S)$ function to cut the same prefix characters between the edge E and the input word W. Then, the new input word W becomes $w_kw_{k+1}...w_{m-1}$ and the root R becomes $E.endnode$. We call procedure $Insert(R, W, index)$ recursively to construct the tree. Figure 3.7-(c) shows the tree before the input word $W = ACG$ is inserted and $E.label = AC$. We traverse the tree structure from the root, then the edge with a label A, and end at the edge with a label C. Now, the input word $W = ACG$ is cut to be G. We change the root R to the node with a value 2 (i.e., the suffix number of character C), then call procedure $Insert(R, W, index)$ again to insert this input word W. At the last step, the input word $W = G$ will be inserted at

Table 3.4 Variables used in procedure $SearchQS(QS)$

SW	a searched word in the query sequence
num	the number of searched words
X	an array which stores the result of function $SearchQW(R, SW, lastword)$
Y	an array which stores the result of the searching process
$flag$	a flag which determines whether the query sequence is in the database or not
$lastword$	a flag which determines whether SW is the last searched word in the query sequence or not

node 2. And an edge with a label G will be created between node 2 and node 4. Figure 3.7-(d) shows the result of this case.

Case 5: This case occurs when $|W| = |E.label| = k$, where $k > 0$. That is, the tree already has an edge E with a label which is the same as the input word W. First, We follow the tree structure to find the node. Next, we will traverse the sibling edges (if any) to reach the end node N of the path. Then, we create a new node K by $CreateNode(index)$ and a new edge with a label s which connects the new node K and node N in the tree. For example, in Figure 3.7-(e), the tree has an edge E labelled with AT. In Figure 3.7-(f), the input word $W = AT$ is the same as the edge $E.label$. Hence, we create an edge labelled with s between the new node 7 and node 0 in the tree.

Figure 3.7 and Figure 3.8 show the process of inserting all ACGT-Words generated by the given sequence S.

3.3 Search

In the ACGT-Words tree, the concept of the search process is similar to the inserting process. However, the searching process is more complicated than the inserting process. Let QS be a query sequence. Let $SW = sw_0sw_1...sw_{m-1}$ be a word for

Table 3.5 Eight cases in search

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

searching and its length is $|SW|$. Let *lastword* be a flag used to determine whether SW is the last searching word in the given query sequence or not. And the definitions of k, $E.label$, and $|E.label|$ are same as the cases of the insertion operation. Table 3.4 shows the variables used in procedure $SearchQS(QS)$. Figure 3.9 shows procedure $SearchQS(QS)$. This procedure cuts the query sequence QS into searched words. For example, given the query sequence $QS = GAGAGCT$, after procedure $SearchQS(QS)$ is executed, it will generate searched words { GA, GA, GCT }. (Note that different from the case of insertion, in the search process, we only consider those disjoint sequences of words.) Moreover, procedure $SearchQS(QS)$ will set flag *lastword* on if SW is the last searched word in the query sequence. For the same example, GCT is the last searched word in the query sequence and the flag *lastword* will be turn on in this case. After generating a searched word in procedure $SearchQS(QS)$, we call function $SearchQW(R, SW, lastword)$ immediately, where R is the node which we traverse now, SW is the searched word, and *lastword* is the flag which indicates whether SW is the last searched word. Figure 3.10 and Figure 3.11 show function $Search(R, SW, lastword)$ and the flowchart for searching, respectively. There are eight different cases in function $SearchQW(R, SW, lastword)$. Table 3.5 shows the conditions of these eight cases and the related semantics are illustrated in Figure 3.12. The conditions and an example of these different cases are described as follows:

```
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 := SearchQW(R, SW[i], lastword);
                if X = ∅ then
                    flag := false;
                else
                begin
                    Y := Combine(Y, X, length(SW[i]));
                    i := i + 1;
                end;
            end;
            lastword := true;
            if ( flag ≠ false ) then
            begin
                X := SearchQW(R, SW[i], lastword);
                X := Sort(X);
                Y := Combine(Y, X, length(SW[i]));
            end;
            if ( X = ∅ or Y = ∅) then
                writeln(" Not find ! ");
            else
                writeln(Y);
        end;
```

Figure 3.9 Procedure $SearchQS(QS)$

```
Function SearchQW(R, SW, lastword): array;
/* The searching process of the ACGT-Words tree for word SW*/
    begin
        i := First(SW);
        k := CompareSameEdge(R.Edge_i, SW);
        if ( k < length(R.Edge_i) ) then
        begin /* Case 1 */
            if ( k < length(SW) ) then
                return ∅;
            else
            begin
                if ( lastword = true ) then /* Case 2 */
                begin
                    X := OutputAll();
                    return X;
                end
                else /* Case 3 */
                    return ∅;
        end
        else
        begin
            if ( k < length(SW) ) then /* Case 4 */
            begin
                R := R.Edge_i.endnode;
                SW := Shift(SW, k);
                X := SearchQW(R, SW, lastword);
                return X;
            end
            else
            begin
                if ( suffix_num_R = NULL ) then /* Case 5 */
                begin
                    if (lastword = true ) then
                    begin
                        X := OutputAll();
                        return X;
                    end
                    else /* Case 6 */
                        return ∅;
                end
                else
                begin
                    i := s;
                    while ( R.Edge_i ≠ NULL )
                    begin
                        X := X ∪ R.Edge_i;
                        R := R.Edge_i.endnode;
                    end;
                    if (lastword = true ) then /* Case 7 */
                        X := X ∪ OutputAll();
                    /* Case 8 */
                    return X;
                end;
            end;
        end;
    end;
```

Figure 3.10 Function $SearchQW(R, SW, lastword)$

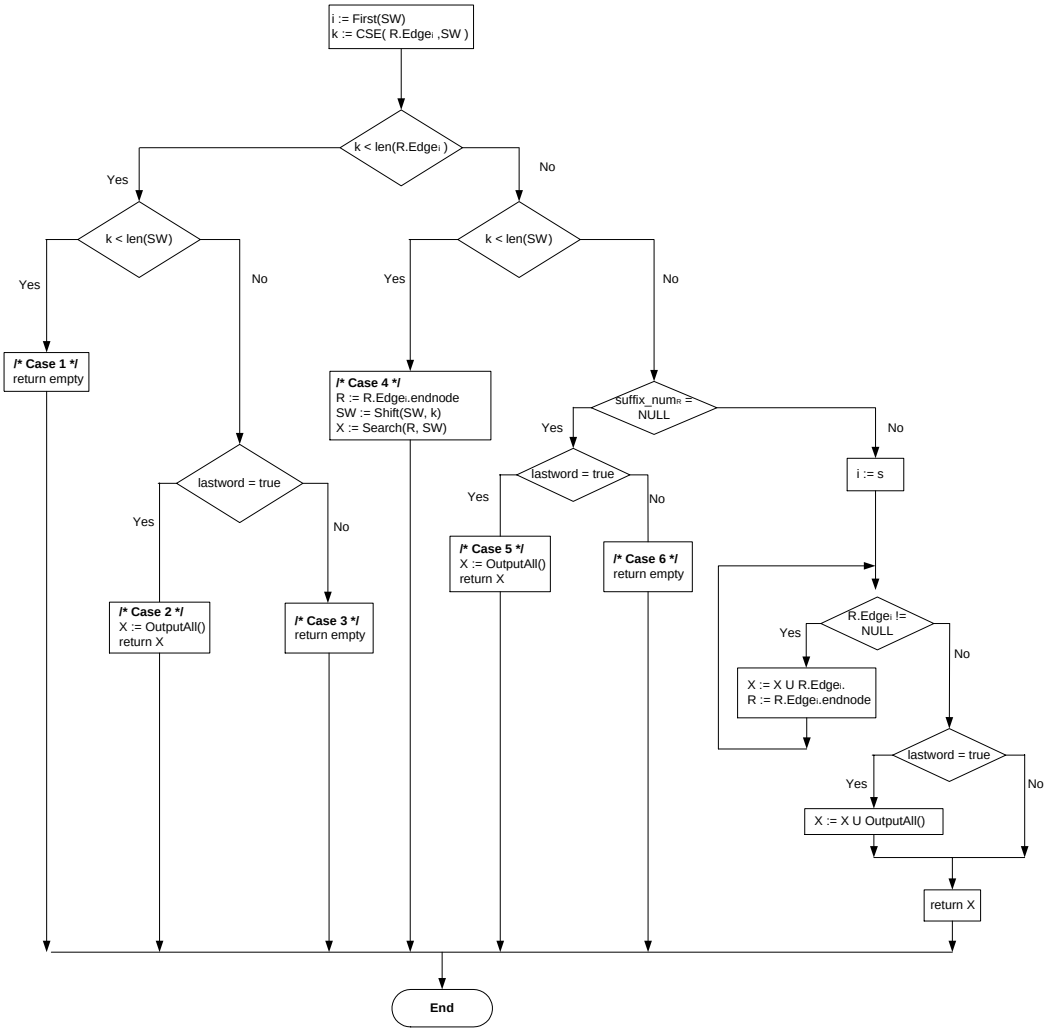


Figure 3.11 The flowchart of function $SearchQW(R,\ SW,\ lastword)$

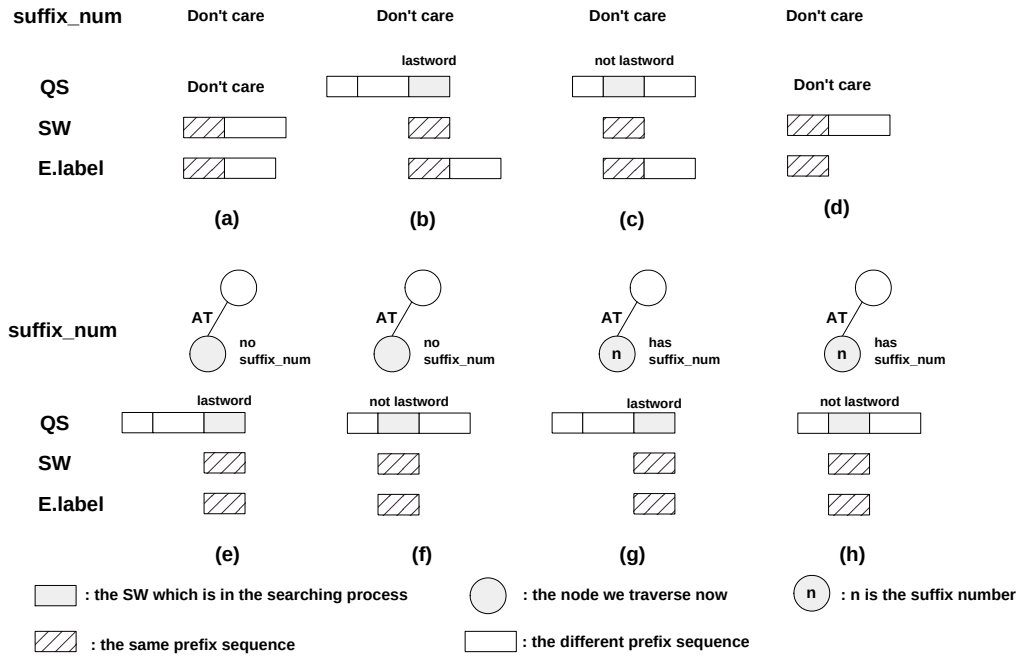


Figure 3.12 Eight cases in the searching process: (a) Case 1; (b) Case 2; (c) Case 3; (d) Case 4; (e) Case 5; (f) Case 6; (g) Case7; (h) Case 8.

Case 1: This case will occur if the following condition is satisfied: $k < |SW|$ and $K < |E.label|$, where $k \geq 0$. That is, there is no common prefix sequence between word SW and $E.label$. In this case, the tree has no edge labelled with the starting character sw_0. It means that we can not find an edge that its label matches the searched word SW. The output of this case prints the information that no sequence matches the query sequence. For example, we assume the query sequence $QS = GACGT$. There are two searched words $\{GAC, GT\}$ in the query sequence QS. We consider the searched word searched word $SW = GAC$. In Figure 3.13, we traverse the tree from the root R, there is no edge with the staring character G. Hence, we conclude that no sequence matches the searched word $SW = GAC$ in the database.

Case 2: This case will occur if the following two conditions are satisfied: (1) $|SW| = k < |E.label|$, where $k > 0$; (2) $lastword = 1$. That is, word SW is the prefix

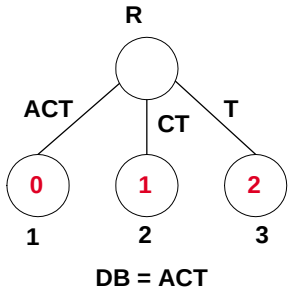


Figure 3.13 An example for Case 1 in the searching process (QS =$\underline{GAC}GT$, $SW = GAC$)

sequence of $E.label$ and SW is the last searched word of the query sequence. In this case, we call function $OutputAll()$. This function traverses the tree from the last edge which the searched word SW meets, and returns an array which stores $suffix\_num$ of nodes that function $OutputAll()$ traverses. In Figure 3.14, we assume the query sequence $QS = AGTAC$. There are two searched words in this query sequence QS. We consider the last searched word $SW = AC$. We traverse the tree from node R, we find $SW = sw_0sw_1 = AC = e_0e_1$ and $|E.label| > |SW|$. Hence, $E.label = ACT$ is one output that matches the condition $SW = AC$. Due to that word SW is the last word of the query sequence QS ($lastword = 1$), it may have other outputs, in this example, like $ACTCG$, $ACTCT$, which also match the searched word SW. Function $OutputAll()$ does reach this goal. Hence, the output of this example is {0, 5, 8}. It means that positions 0, 5, and 8 match the last searched word $SW = AC$.

Case 3: This case will occur if the following two conditions are satisfied: (1) $|SW| = k < |E.label|$, where $k > 0$; (2) $lastword = 0$. That is, SW is the prefix sequence of $E.label$ but SW is not the last searched word of the query sequence. The output of this case returns that the database has no sequence which matches the query sequence. In Figure 3.15, we assume the query sequence $QS = ATACG$.

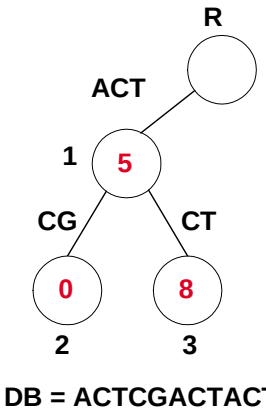


Figure 3.14 An example for Case 2 in the searching process ($QS = AGT\underline{AC}$, $SW = AC$)

After the query sequence QS is split into words in procedure $SearchQS(QS)$, the searched words generated by this procedure are $\{AT, ACG\}$. First, we find whether word $SW = AT$ is in the database. Because word SW is not the last word of the query sequence QS, word $SW = AT$ must exactly match the label of edge E, i.e., $SW = E.label$. But, in this example, we have $SW = AT \neq E = ATG$. It means that no sequence matches the query sequence QS.

Case 4: This case will occur if the following condition is satisfied: $|SW| > k = |E.label|$, where $k > 0$. That is, $E.label$ is the prefix sequence of word SW. First, we call the $Shift(S)$ function to cut the prefix sequence between $E.label$ and the searched word SW. Second, we change the root R to $endnode$ of E. Finally, we call function $SearchQW(R, SW, lastword)$ again to search the new searched word. For example, in Figure 3.16, we assume the query sequence $QS = GTCTGA$. There are two searched words $\{GTC, GA\}$ in the query sequence. We consider the searched word $SW = GTGC$. We traverse the tree from node R. Then, we find $e_0e1 = GT = sw_0sw_1$, and $sw_2sw_3 = CT$ is not an empty sequence. Hence, we change the root R to node 1, and $SW = sw_2sw_3 = CT$ (due to $|SW| > |E.label|$). Then, we call function

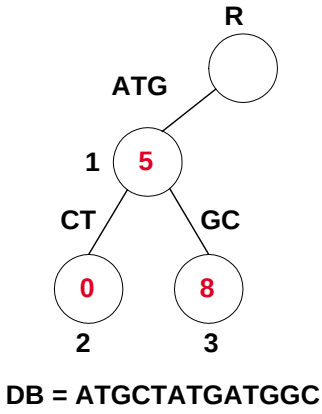


Figure 3.15 An example for Case 3 in the searching process (QS =$\underline{AT}ACG$, $SW = AT$)

$SearchQW(R, SW, lastword)$ again to traverse the tree deeply. Finally, the searched word $GTCT$ is found in the database.

Case 5: This case occurs when the following three conditions are satisfied: (1) $|SW| = |E.label| = k$, where $k > 0$; (2) $E.endnode.suffix\_num = null$; (3) $lastword = 1$. That is, word SW is exactly equal to $E.label$ and word SW is the last searched word of the query sequence. But, $E.endnode$ has no $suffix\_num$. It means that this node is created by two or more edges that have prefix sequence. For example, In Figure 3.17, the 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 an edge $E = AT$ and its $endnode$ with no $suffix\_num$. Hence, in this case, we only call function $OutputAll()$. It traverses the tree from the edge which the searching word meets. For example, we assume the query sequence $QS = ACGAT$. That is, there are two searched words in the query sequence QS. We consider the searched sequence $SW = AT$ and SW is the last searched word of the query sequence. In Figure 3.17, we traverse the tree from the root

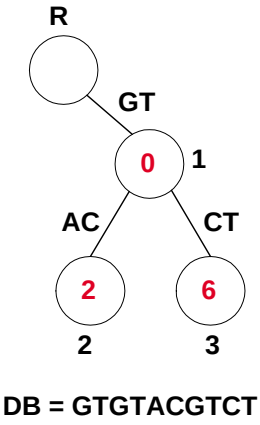


Figure 3.16 An example for Case 4 in the searching process (QS =$\underline{GTCT}GA$, $SW = GTCT$)

R. Then, the edge E with a label AT ($E.label = AT$) matches the word SW and $E.endnode$ stores no $suffix\_num$. Hence, we call function $OutputAll()$ to traverse the tree and output the result. In this example, the result after calling function $OutputAll()$ is $\{0, 3\}$. It means that positions 0 and 3 of the database sequence match the searched word $SW = AT$.

Case 6: This case occurs when the following three conditions are satisfied: (1) $|SW| = |E.label| = k$, where $k > 0$; (2) $E.endnode.suffix\_num = null$; (3) $lastword = 0$. That is, word SW is exactly equal to $E.label$ but word SW is not the last searched word of the query sequence and $E.endnode$ 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. In Figure 3.18, we assume the query sequence $QS = CACTG$. After splitting the query sequence QS in procedure $SearchQS(QS)$, the searched words generated by this procedure are $\{CA, CTG\}$. First, we consider $SW = CA$ and word SW is not the last word of query sequence QS. Hence, we have the flag $lastword = 0$. We traverse the tree from the root R, and the word SW matches the edge E with a label CA ($E.label = CA$). But the $endnode$ of E, node 1, stores no $suffix\_num$ and

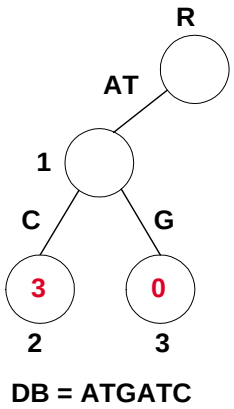


Figure 3.17 An example for Case 5 in the searching process ($QS = ACG\underline{AT}$, $SW = AT$)

SW is not the last searched word of the query sequence. Hence, we output that no sequence matches the query sequence $QS = CACTG$ in the database.

Case 7: This case occurs when the following three conditions are satisfied: (1) $|SW| = |E.label| = k$, where $k > 0$; (2) $E.endnode.suffix\_num \neq null$; (3) $lastword = 1$. That is, SW is exactly equal to $E.label$, SW is the last searched word of the query sequence, and $E.endnode$ has $suffix\_num$. In Figure 3.19, we assume the query sequence $QS = ACATG$. That is, there are two searched words $\{AC, ATG\}$ in the query sequence QS. We consider the searched word $SW = ATG$, and this searched word is the last word of query sequence QS. We traverse the tree structure from root R. We find edge E with a label ATG exactly matches $SW = ATG$ and the $endnode$ of edge E has value 0 ($E.endnode.suffix\_num \neq null$). Next, we find whether there exists other edges with a label s. We find that node 1 has an edge with a label s. We traverse node 3 and add its $suffix\_num$ into our result. We repeat this step until no edge with a label s is found. Finally, we traverse all the nodes which are under node 1 and add the $suffix\_num$ of nodes to our result until no node is found. Function $OutputAll()$ does reach this goal.

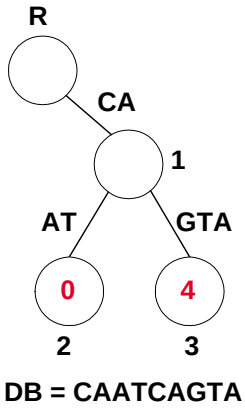


Figure 3.18 An example for Case 6 in the searching process (QS =$\underline{CA}CTG$, $SW = CA$)

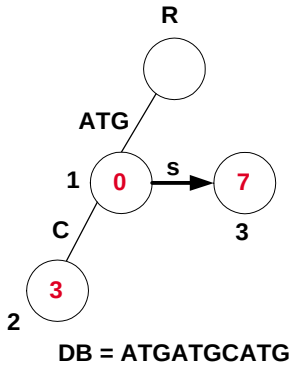


Figure 3.19 An example for Case 7 in the searching process ($QS = AC\underline{ATG}$, $SW = ATG$)

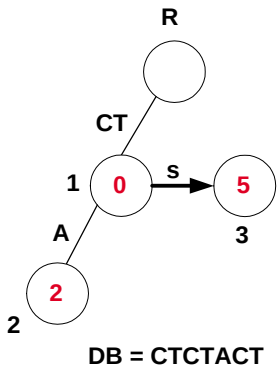


Figure 3.20 An example for Case 8 in the searching process ($QS = \underline{CT}CGACA$, $SW = CT$)

Case 8: This case occurs when the following three conditions are satisfied: (1) $|SW| = |E.label| = k$, where $k > 0$; (2) $E.endnode.suffix\_num \neq null$; (3) $lastword = 0$. That is, word SW is exactly equal to $E.label$ and $E.endnode$ has $suffix\_num$. But, word SW is not the last searched word of the query sequence. In this case, the searched word SW exactly matches the $E.label$ in the tree. For example, in Figure 3.20, we assume the query sequence $QS = CTCGACA$. After splitting this query sequence QS, the searched words are $\{CT, CGA\ CA\}$. First, we consider the searched word $SW = CT$. We traverse the tree from the root R. Then, we find the edge E with a label CT exactly matches $SW = CT$ and the $endnode$ of edge E, node 1, has value 0 ($E.endnode.suffix\_num \neq null$). Next, we check whether there exists other edges with a label s. It means that edge $E.label = CT$ occurs more than once in the database. In this example, we find that node 1 has an edge with a label s. We follows this edge with a label s to its $endnode$, node 3. Then, we add its $suffix\_num$ to our result. We repeat this step until no edge with a label s is found. Hence, the result of this example is $\{0, 5\}$. It means that these two positions of the database sequence match the first searched word $SW = CT$.

```
Function Combine(Y, X, len): array;
/* The combining process after executing the searching process. */
        begin
            index := 0;
            i := 0;
            j := 0;
            Temp := Y;
            Y := ∅;
            while ( i ≠ length(Temp) and j ≠ length(X) ) do
            begin
                if ( (Temp[i] + len) = X[j] ) then
                begin
                    Y[index] := Temp[i];
                    i := i + 1;
                    j := j + 1;
                    index := index + 1;
                end
                else if ( (Temp[i] + len) > X[j] ) then
                    j := j + 1
                else
                    i := i + 1;
            end;
            return Y;
        end;
```

Figure 3.21 Function $Combine(Y, X, len)$

Next, in procedure $SearchQS(R, SW, lastword)$, after calling function $SearchQW$ $(R, SW, lastword)$ for a searched word SW, we will combine the result from function $SearchQW(R, SW, lastword)$ and the previous result by calling function $Combine(Y, X, len)$ until all searched words are found. Figure 3.21 shows function $Combine(Y, X, len)$. In function $Combine(Y, X, len)$, we prune the positions that can not occur in the query sequence. After we find all searched words, the array Y, which is returned from function $Combine(Y, X, len)$, is the positions that query sequence QS occur in the database.

Given the database sequence $ATACACGAT$ and the query sequence $QS = ATAC$, in our searching process, procedure $SearchQS(QS)$ will split the query sequence $ATAC$ into two searched words $\{AT, AC\}$. When a searched word SW is generated, we search this searched word SW by function $SearchQW(R, SW, lastword)$ immediately. In this example, we search $SW = AT$ first. Then, the positions 0 and 7, which satisfy the condition $SW = AT$, are returned from function $SearchQW(R, SW, lastword)$. Finally, function $Combine(Y, X, len)$ combines $X = \{0, 7\}$ with $Y = \emptyset$ and the result is stored in Y. Now, we search the next searched

word $SW = AC$. In procedure $SearchQS(QS)$, we know AC is the last searched word of the query sequence QS. We traverse the tree structure from the root, and find $SW = AC$ occurring at position 2. Then, position 2 is added into X. But $SW = AC$ is the last searched word of the query sequence QS in this example, we have to traverse all the nodes under the node which we are traversing now. Hence, position 4 will be added into X. After function $SearchQW(R, SW, lastword)$ is executed for $SW = AC$, we get the resulting array $X = \{2, 4\}$. Finally, we combine X and the previous result Y by function $Combine(Y, X, len)$ and get the final answer, i.e., position 0 in the database containing the query sequence $ATAC$. Figure 3.22 shows the searching process in the ACGT-Words tree.

Let's see another example. Given the query sequence $QS = CACG$ and the database sequence is same as the previous example, procedure $SearchQS(QS)$ splits the query sequence into $\{CA, CG\}$. First, we follow the tree structure to search $SW = CA$ by function $SearchQW(R, SW, lastword)$. The result of function $SearchQW(R, SW, lastword)$ is position 3. This is different from the previous example. Because in this case, the searched word $SW = CA$ is not the last searched word of the query sequence. Then, we add this result into X and combine X and Y. Similarly, we search the next searched word $SW = CG$ by function $SearchQW(R, SW, lastword)$ and add the result into $X = \{5\}$. Then, function $Combine(Y, X, len)$ will combine X and Y. Finally, we get the resulting array $Y = \{3\}$ which stores the position of the occurrence of the query sequence $QS = CACG$ in the database. Figure 3.23 shows the searching process in the ACGT-Words tree.

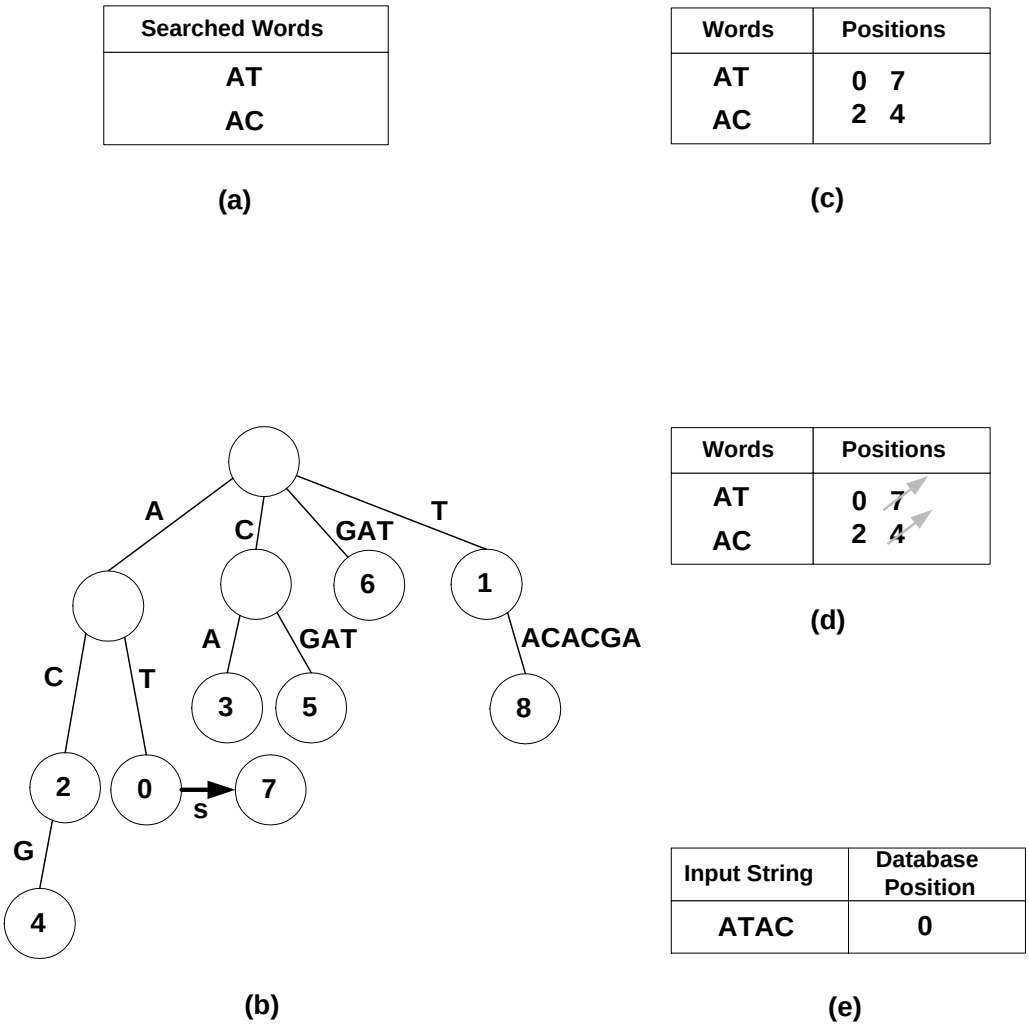


Figure 3.22 The searching process for the query sequence $ATAC$ (AT in Case 8, AC in Case 7) : (a) after parsing; (b) scanning database; (c) the initial result; (d) combining; (e) the final result.

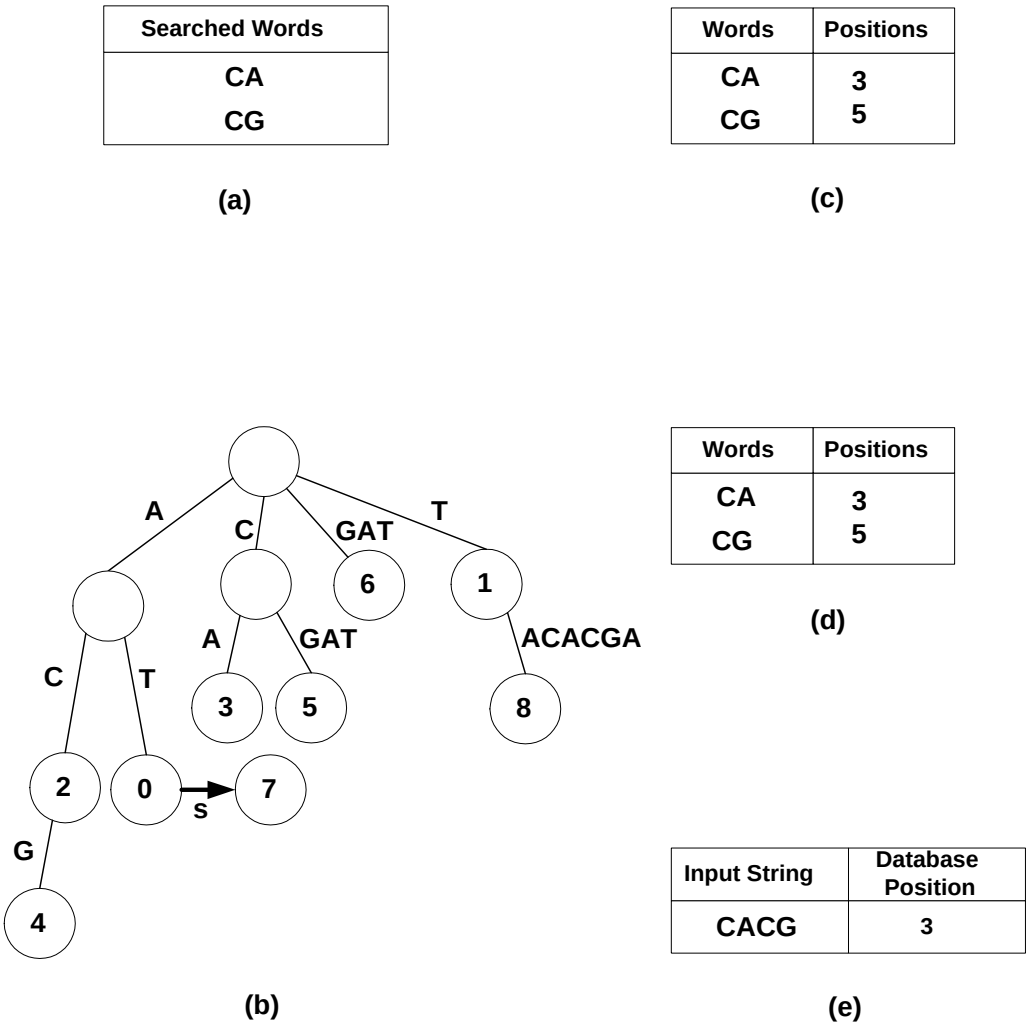


Figure 3.23 The searching process for the query sequence $CACG$ (CA in Case 8, CG in Case 2) : (a) after parsing; (b) scanning database; (c) the initial result; (d) combining; (e) the final result.

CHAPTER IV

Improvements of Operations in an ACGT-Words Tree

In this Chapter, we present some improving methods for the process of insertion and search in our ACGT-Words tree.

4.1 Improvement of the Insertion Operation

In our process of data insertion presented in Chapter 3.2, when the DNA sequence is split into an ACGT-Word, we insert the word into the ACGT-Words tree immediately. Whenever the same ACGT-Word occurs again, we have to traverse the tree structure from the root again, which takes a lot of time. Hence, we propose a new method for improving the insertion process.

Table 4.1 shows the ACGT-Words after the sequence $ACAGTAC$ is split by procedure $Split(S)$. We insert these ACGT-Words one after one in the order of generation. In this example, we can find the ACGT-Word AC which occurs twice. But the ACGT-Word AC does not occur successively, we have to traverse the same path again to insert this ACGT-Word AC. In the improvement of the insertion process, after all the ACGT-Words are generated, we sort those ACGT-Words by the lexicographic order. Hence, we make some change from procedure $Split(S)$ to procedure $Split^*(S)$. After an ACGT-Word W is inserted into the tree, we will check whether the remaining ACGT-Words contain the same word W. We only use a string comparison to reach this goal. Hence, we do not waste time to traverse the same path in the tree structure again. Table 4.2 shows the resulting ACGT-Words after those words are sorted by procedure $Split^*(S)$.

Table 4.1 The ACGT-Words constructed by $Split(S)$ from $ACAGTAC$

ACGT-Words	Position
AC	0
AGT	2
$CAGTC$	1
AC	5
C	6
GTA	3
TAC	4

Table 4.3 shows the variables used in procedure $Split^*(S)$ and Figure 4.1 shows the process of procedure $Split^*(S)$, where /**/ denotes the change. In procedure $Split^*(S)$, when a new ACGT-Word is generated, we do not call procedure $Insert(R,$ $W,$ $index)$ immediately. Instead, we store the word into an array BS. After all the words are split and added into array BS, we sort this array BS by the lexicographic order and store the result in another array AS. Therefore, the same ACGT-Words will be put in the successive elements of array AS.

According to the size of array AS, we determine how many times to call procedure $Insert^*(R,$ $index)$. In the final step of procedure $Split^*(S)$, we construct the tree structure by procedure $Insert^*(R,$ $index)$. Figure 4.2 shows procedure $Insert^*(R,$ $index)$. This procedure is different from procedure $Insert(R,$ $W,$ $index)$. After an ACGT-Word X is inserted, we check whether the next element Y in array BS is the same as word X in procedure $Split^*(S)$. If $Y = X$, we will call procedure $Insert1(R,$ $index)$ to add word Y into the sibling node of the node containing word X which we traverse now. We do not traverse the tree from the root again for inserting the same words. Therefore, it can improve the performance of the inserting process.

Table 4.2 The ACGT-Words constructed by $Split^*(S)$ from $ACAGTAC$

ACGT-Words	Position
AC	0
AC	5
AGT	2
C	6
$CAGTC$	1
GTA	3
TAC	4

Table 4.3 The variables in procedure $Split^*(S)$

S	the input sequence
P	an index of the input sequence S
W	an ACGT-Word
BS	an array which stores data structure *item* before sorting
AS	an array which stores data structure *item* after sorting BS

```
Procedure Split*(S);
/* Generate the words of input sequence S.*/
/* Substring(S, Ns, Ne) is a function which returns subsequence of S which starts from Ns and ends at Ne. */
/* Item(W, pos) is a function which returns an item that contains an ACGT-Word and its position in sequence S. */
/* AddRemainingWord(BS) is a function that returns an array which stores all ACGT-Words. */
/* RemoveNotSearchedWord(BS, z) is a function that returns an array BS which stores the searched words z. */
        begin
            P := 0;
            PreWord := null;
            while ( P < len(S) ) do
            begin
                if ( S[P] = 'A' ) then
                begin
                    if ( A_num ≠ −1 ) then
                    begin
                        W := Substring(S, A_num, (P − 1));
                        BS[P] := Item(W, A_num); /**/
                    end;
                    A_num := P;
                end
                else if ( S[P] = 'C' ) then
                begin
                    if ( C_num ≠ −1 ) then
                    begin
                        W := Substring(S, C_num, (P − 1));
                        BS[P] := Item(W, C_num); /**/
                    end;
                    C_num := P;
                end
                else if ( S[P] = 'G' ) then
                begin
                    if ( G_num ≠ −1 ) then
                    begin
                        W := Substring(S, G_num, (P − 1));
                        BS[P] := Item(W, G_num); /**/
                    end;
                    G_num := P;
                end
                else ( S[P] = 'T' ) then
                begin
                    if ( T_num ≠ −1 ) then
                    begin
                        W := Substring(S, T_num, (P − 1));
                        BS[P] := Item(W, T_num); /**/
                    end;
                    T_num := P;
                end;
                P := P + 1;
            end;
            /**/
            BS := AddRemainingWord(BS);
            BS := RemoveNotSearchedWord(BS, S[0]);
            AS := Sort(BS);
            for ( i = 0; i < length(BS); i + + )
            begin
                index := GetIndex(BS[i]);
                W := GetWord(BS[i]);
                if ( W = PreWord )
                    Insert1(R, index);
                else
                    Insert(R, W, index);
                R := GetNewNode();
                PreWord := W;
            end;
            /**/
        end;
```

Figure 4.1 Procedure $Split^*(S)$

```
Procedure Insert*(R, index);
/* The improved insertion operation in the ACGT-Words tree. */
/* Sibling() is a function that returns the character s which labels the edge between two same words. */
/* CreateNode(index) is a function that returns a new node whose suffix_num is index. */
        begin
            i := Sibling();
            N := CreateNode(index);
            R.Edge_i.ednnode := N;
        end;
```

Figure 4.2 Procedure $Insert^*(R, index)$

4.2 Improvements of the Search Operation

In this section, we improve the searching process for two cases in query sequences. We describe these two cases in detail.

4.2.1 Case 1

First, we consider the query sequences which have the condition that it is the last searched word of the query sequence and with size = 1 (i.e., a single character). (Note that this case will also be considered in original Cases 2, 5, and 7 discussed in Chapter 3.3.) For example, if the query sequence $QS = ACA$, the searched words generated by procedure $SearchQS(QS)$ are $\{AC, A\}$. We consider the last searched word A. It is the last searched word and it is the first character of the query sequence $QS = ACA$. If the query sequence satisfies this condition, we can handle it in another way to provide better performance for the searching process.

In the original search process (presented in Chapter 3.3), for this searched word A, we have to traverse all the nodes with the starting character A of the query sequence in the tree structure. However, if we can record down the position of the last occurrence of ACGT-Words starting with A in the database, we can immediately answer the query without doing such a traverse of the tree. For example, given the sequence $S = ACAGTAC\$$ stored in the database, the position of the last occurrence of the character A is 5 (recorded in Max_A). Before we start to search the searched word $SW = A$, we have already known the result for the searched word $SW = AC$, which

is $\{0,\ 5\}$. Since the length of AC is 2, the first possible resulting position for the searched word $SW = A$ is $0 + 2 = 2$, and the second possible resulting position for $SW = A$ is $5 + 2 = 7$. However, since the possible resulting position $7 > Max_A(= 5)$ which records the last occurrence of the ACGT-Word starting with A, the possible resulting position 7 can not be the final answer. While the possible resulting position $0 + 2 = 2 \leq Max_A(= 5)$ can be the final result. Therefore, by making use of the variable Max_A to record the last occurrence of ACGT-Words starting with A, we can answer the query for such a case immediately.

Based on this concept, we can use four variables Max_A, Max_C, Max_G, and Max_T to record the positions of the last occurrence of A, C, G, T, respectively, to achieve this goal. There are some changes in our algorithm. We use procedure $Split1(S)$, instead of procedure $Split(S)$. Figure 4.3 shows this procedure. At the end of procedure $Split1(S)$, we use procedure $SetFourVariables()$ to set the value of these four variables. This procedure records the positions of the last occurrence of ACGT-Words starting with A, C, G, and T, respectively. Then, we set the value of four variables, Max_A, Max_C, Max_G, and Max_T, to the positions of four ACGT-Words, respectively. For example, given the database sequence $S = ACAGTAC$, the last ACGT-Words starting with A, C, G, T are AC, C, $GTAC$, and TAC, respectively. The positions of these four ACGT-Words starting with A, C, G, T are 5, 6, 4, 3, respectively. Hence, the value of Max_A, Max_C, Max_G, and Max_T are 5, 6, 4, and 3, respectively.

There are some differences between procedure $SearchQS1(QS)$ and the original procedure $SearchQS(QS)$. Figure 4.4 shows this procedure, where /**/ denotes the change. In procedure $SearchQS1(QS)$, if the last searched word matches the condition of this improved operation, we will call function $Combine1(Y,\ max,\ SW)$ to check each of the elements whether its value is less than or equal to max in Y, which is the result for the previous searched words. For the previous example, the query sequence $QS = ACA$, the two searched words are $\{AC,\ A\}$ in the query sequence QS. Before we start to search the searched word $SW = A$, we have already known the result for the searched word $SW = AC$, which is $Y = \{0, 5\}$. Then, we call

```
Procedure Split1(S);
/* Generate the words of input sequence S. */
/* P is an index of the input sequence S. */
/* W stores an ACGT-Word. */
/* Substring(S, Ns, Ne) is a function which returns subsequence of S which starts from Ns and ends at Ne. */
        begin
            P := 0;
            while ( S[P] ≠ '$') do
            begin /* Create a new Node and a new Edge */
                if ( S[P] = 'A' ) then
                begin
                    if ( A_num ≠ −1 ) then
                    begin
                        W := Substring(S, A_num, (P − 1));
                        Insert(R, W, A_num);
                    end;
                    A_num := P;
                    Max_A = A_nem;
                end
                else if ( S[P] = 'C' ) then
                begin
                    if ( C_num ≠ −1 ) then
                    begin
                        W := Substring(S, C_num, (P − 1));
                        Insert(R, W, C_num);
                    end;
                    C_num := P;
                    Max_C = C_nem;
                end
                else if ( S[P] = 'G' ) then
                begin
                    if ( G_num ≠ −1 ) then
                    begin
                        W := Substring(S, G_num, (P − 1));
                        Insert(R, W, G_num);
                    end;
                    G_num := P;
                    Max_G = G_nem;
                end
                else ( S[P] = 'T' ) then
                begin
                    if ( T_num ≠ −1 ) then
                    begin
                        W := Substring(S, T_num, (P − 1));
                        Insert(R, W, T_num);
                    end;
                    T_num := P;
                    Max_T = T_nem;
                end;
                P := P + 1;
            end;
            InsertRemainingWord();
            SetFourVariables(); /**/
        end;
```

Figure 4.3 Procedure $Split1(S)$

function $Combine1(Y,\ max,\ SW)$ to check all elements in Y. Since the length of AC, the previous searched word, is 2, we check the first element, 0, in Y by function $Combine1(Y, max, SW)$. Since $0+2=2<Max_A(=5)$, the first element in Y is the final answer. Then, we check next element, 5, in Y. Since $5+2=7>Max_A(=5)$, the second, also the last element is not the final answer. Therefore, by making use of these four variables (Max_A, Max_C, Max_G, and Max_T), we do not traverse the tree structure and can answer the query for such a case immediately.

4.2.2 Case 2

In this case, we consider the query sequences which have successive same characters (i.e, A, C, G, and T). For example, the query sequence $QS = ACAAAATG$. There is a subsequence $AAAA$ which has successive same character A. This case may occur in the query sequence because there are only four characters in DNA sequences.

Although we can answer the query sequences which contain these conditions based on the method presented in Chapter 3, we are trying to improve the performance of this case in our tree structure. Given the previous example, the query sequence $QS = ACAAATG$, in our previous algorithm, the searched words are $\{AC,\ A,\ A,\ A,\ ATG\}$. We have to search the searched word $SW = A$ three times in this example. It may waste a lot of time to traverse the tree structure when this case occurs. Hence, we plan to make some changes in the process of splitting query sequence.

For achieving this goal, we use function $Split2(S)$, instead of function $Split(S)$. Figure 4.6 shows this function. In function $Split2(S)$, we use another function, $CombineSSW(BS)$, to combine the successive same searched words with only one character to be a searched word after the searched words are generated. Figure 4.7 shows function $CombineSSW(BS)$. For the previous example, the query sequence $QS = ACAAAATG$, the searched words are $\{AC,\ A,\ A,\ A,\ ATG\}$. Now, we will combine three successive searched words with the same character A to be a searched word AAA. Hence, the searched words in the query sequence $QS = ACAAAATG$ will be changed to $\{AC,\ AAA,\ ATG\}$ by calling function $CombineSSW(BS)$.

```
Procedure SearchQS1(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. */
/* TotalLength(SW) is a function that returns the total length of all searched words. */
/* Sort(X) is a function that returns the resulting an array after sorting array X. */
/* Combine1(Y, max) is a function that returns an array which contains the matching positions in the database. */
    begin
        SW := Split(QS);
        num := TotalNum(SW);
        TL_SW := TotalLength(SW);
        X := ∅
        Y := ∅;
        flag := true;
        lastword := false;
        i := 0;
        while ( flag = true and i < (num − 1) )
        do
        begin
            X := SearchQW(R, SW[i], lastword);
            if X = ∅ then
                flag := false
            else
            begin
                Y := Combine(Y, X, length(SW[i]));
                i := i + 1;
            end;
        end;
        lastword := true;
        if ( flag ≠ false ) then
        begin
            /**/
            if ( length(SW[i]) = 1 )
            begin
                max := GetMax(SW[i]);
                Y := Combine1(Y, max, SW[i − 1]);
            end
            /**/
            else
            begin
                X := SearchQW(R, SW[i], lastword);
                X := Sort(X);
                Y := Combine(Y, X, length(SW[i]));
            end;
        end;
        if ( X = ∅ or Y = ∅) then
            writeln(" Not find ! ")
        else
            writeln(Y − TL_SW);
    end;
```

Figure 4.4 Procedure $SearchQS1(QS)$

```
Function Combine1(Y, max, SW): array;
/* The combining process after executing the searching process. */
        begin
            index := 0;
            i := 0;
            Temp := Y;
            Y := ∅;
            while ( i ≤ length(Temp) ) then
            begin
                if ( Temp[i] + length(SW) ≤ max ) then
                begin
                    Y[index] := Temp[i] + length(SW);
                    index := index + 1;
                end
                i := i + 1;
            return Y;
        end;
```

Figure 4.5 Function $Combine1(Y, max, SW)$

However, there is one special case which we must concern, that is, when the sequence containing successive same character occurs at the tail of the query sequence. In this case, we do not combine the last searched word with the successive same character in the query sequence. For example, given the query sequence $QS = ACATGAAA$, the searched words should be changed to $\{AC, ATG, AA, A\}$ in the query sequence.

After the searched words are generated, we call procedure $SearchQS2(QS)$ to find these searched words. Figure 4.8 shows procedure $SearchQS2(QS)$. This procedure has some difference with the original procedure $SearchQS(QS)$, where /**/ denotes the changes. In procedure $SearchQS2(QS)$, we have a function $IsSuccessive(SW)$ which checks whether this searched word SW is a successive same character or not. If SW is the searched word with the successive same character, we will call function $SearchQW2(R, Startchar, n, lastword)$ to find the searched word SW. Figure 4.9 shows function $SearchQW2(R, Startchar, n, lastword)$. When searching the successive same searched word in function $SearchQW2(R, Startchar, n, lastword)$, we traverse the tree structure to find the first searched word of the successive same searched words. For example, when searching AAA, we traverse the tree structure to find the first searched word A first. After finding the node with the first searched word, we traverse the siblings of this node. According to the parameter n, which

```
Function Split2(S): array;
/* Generate the words of input sequence S. */
/* Substring(S, Ns, Ne) is a function which returns subsequence of S which starts from Ns and ends at Ne. */
/* Item(W, pos) is a function which returns an item that contains a ACGT-Word and its position in sequence S. */
/* AddRemainingWord(BS) is a function that returns an array which stores all ACGT-Words. */
/* RemoveNotSearchedWord(BS, z) is a function that returns an array BS which stores the searched words z. */
        begin
            P := 0;
            PreWord := null;
            while ( S[P] ≠ '$') do
            begin
                if ( S[P] = 'A' ) then
                begin
                    if ( A_num ≠ −1 ) then
                    begin
                        W := Substring(S, A_num, (P − 1));
                        BS[P] := Item(W, A_num);
                    end;
                    A_num := P;
                end
                else if ( S[P] = 'C' ) then
                begin
                    if ( C_num ≠ −1 ) then
                    begin
                        W := Substring(S, C_num, (P − 1));
                        BS[P] := Item(W, C_num);
                    end;
                    C_num := P;
                end
                else if ( S[P] = 'G' ) then
                begin
                    if ( G_num ≠ −1 ) then
                    begin
                        W := Substring(S, G_num, (P − 1));
                        BS[P] := Item(W, G_num);
                    end;
                    G_num := P;
                end
                else ( S[P] = 'T' ) then
                begin
                    if ( T_num ≠ −1 ) then
                    begin
                        W := Substring(S, T_num, (P − 1));
                        BS[P] := Item(W, T_num);
                    end;
                    T_num := P;
                end;
                P := P + 1;
            end;
            BS := AddRemainingWord(BS);
            BS := RemoveNotSearchedWord(BS, S[0]);
            BS := CombineSSW(BS); /**/
            return BS;
        end;
```

Figure 4.6 Function $Split2(S)$

```
Function CombineSSW(BS): array;
/* Combine the same searched word which has only one character in array BS. */
/* Item(W, pos) is a function which returns an item that contains an ACGT-Word and its position in sequence S. */
/* BS is an array which stores the result after combining the same searched words. */
        begin
            i := 0;
            W := null;
            flag := false;
            size := 0;
            count := 0;
            index := 0;
            TempBS := BS;
            while ( i < length(TempBS) ) do
            begin
                if ( length(TempBS) = 1 and i ≠ (length(TempBS) − 1)) then
                begin
                    count := count + 1;
                    flag := true;
                end
                else
                begin
                    if ( flag = true )
                    begin
                        index := i − count;
                        W := CreateWord(TempBS[i], count);
                        BS[size] := Item(W, index);
                        flag := false;
                        size := size + 1;
                    end
                    BS[size] := TempBS[i];
                    size := size + 1;
                end;
                i := i + 1;
            end;
            return BS;
        end;
```

Figure 4.7 Function $CombineSSW(BS)$

```
Function SearchQS2(QS): array;
/* The main procedure of the searching process in the ACGT-Words tree. */
/* Split2(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. */
/* TotalLength(SW) is a function that returns the total length of all searched words. */
/* 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. */
/* GetStartchar(SW) is a function that returns the starting character of the searched word SW. */
/* IsSuccessive(SW) is a function that returns whether the searched word SW contains the successive same character.*/
        begin
            SW := Split2(QS);
            num := TotalNum(SW);
            TL_SW := TotalLength(SW);
            X := ∅
            Y := ∅;
            i := 0;
            count := 0;
            flag := true;
            lastword := false;
            Startchar := 0;
            while ( flag = true and i < (num − 1) )
            do
            begin
                /**/
                if ( IsSuccessive(SW[i]) = true )
                begin
                    Startchar = GetStartChar(SW[i]);
                    count = length(SW[i]);
                    X := SearchQW2(R, Startchar, count, lastword);
                    Y := Combine(Y, X, length(SW[i]));
                    i := i + 1;
                end
                /**/
                else
                begin
                    X := SearchQW(R, SW[i], lastword);
                    if X = ∅ then
                        flag := false;
                    else
                    begin
                        Y := Combine(Y, X, length(SW[i]));
                        i := i + 1;
                    end;
                end;
            end;
            lastword := true;
            if ( flag ≠ false ) then
            begin
                X := SearchQW(R, SW[i], lastword);
                X := Sort(X);
                Y := Combine(Y, X, length(SW[i]));
            end;
            if ( X = ∅ or Y = ∅) then
                writeln(" Not find ! ");
            else
                writeln(Y − TL_SW);
        end;
```

Figure 4.8 Function $SearchQS2(QS)$

```
Function SearchQW2(R, Startchar, n, lastword): array;
/* The searching process for the searched word which contains the successive character. */
/* OccurStartchar is an array that stores the the positions of the searched words Startchar. */
/* OccurPos is an array that stores the result of searching the successive same character. */
        begin
            i := Startchar;
            index := 0;
            size_OS := 0;
            size_OP := 0;
            while ( R.Edge_i ≠ NULL )
            begin
                R := R.Edge_i.endnode;
                OccurStartchar[size_OS] := R.suffix_num;
                size_OS := size_OS + 1;
            end;
            while ( (index + n) ≤ size_OS )
            begin
                if (OccurStartchar[index + n] = OccurStartchar[index] )
                begin
                    OccurPos[size_OP] := OccurStartchar[index];
                    size_OP := size_OP + 1;
                end;
                index := index + 1;
            end;
            return OccurPos;
        end;
```

Figure 4.9 Function $SearchQW2(R, Startchar, n, lastword)$

records the number of the successive same searched words, of function $SearchQW2(R, Startchar, n, lastword)$, we check whether this node contains n successive siblings whose $suffix\_num$ is successive. For example, when we search AA in Figure 4.10. First, we find the searched word A in the node with $suffix\_num$ 3. Next, we find the siblings of the node with $suffix\_num$ 3. That is, the siblings are the node with $suffix\_num$ 4 and the node with $suffix\_num$ 7. Since the length of searched word AA is 2, the value of n is 2. Therefore, we check whether those 2 successive nodes are with successive $suffix\_num$ by function $SearchQW2(R, Startchar, n, lastword)$. In Figure 4.10, the node with $suffix\_num$ 3 and the node with $suffix\_num$ 4 are an example, so is the case of the node with $suffix\_num$ 7 and the node with $suffix\_num$ 8. Hence, in this example, the result after calling function $SearchQW2(R, Startchar, n, lastword)$ is $\{3, 7\}$. It means that the positions 3 and 7 of database sequence contain this searched word AA. Therefore, if the query sequence contains the successive same searched words, we only traverse the tree structure once and can answer the query for such a case immediately.

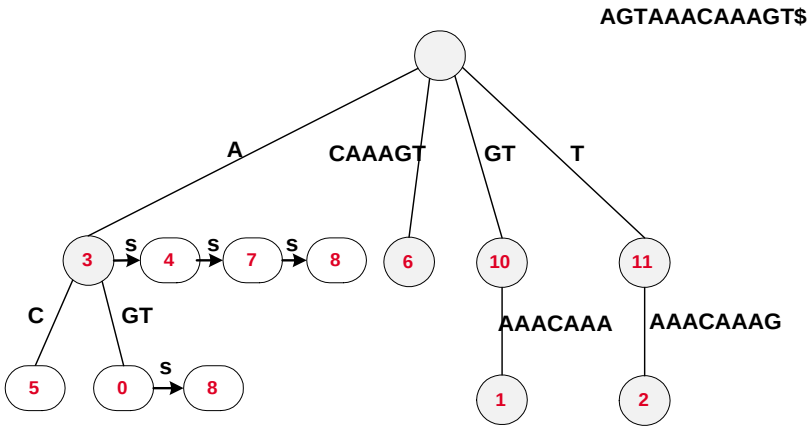


Figure 4.10 An example of improved search operation (Case 2)

CHAPTER V

Performance

In this Chapter, 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. Our experiments were performed on a Celeron machine with one CPU clock rate of 1 GHz, 384 MB of main memory, running Windows 2000 Professional version, and coded in Java.

5.1 Generation of Synthetic Data

We generate synthetic sequences to evaluate the performance of the algorithms over a large range of data characteristics. 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 5.1. PN is the number of distinct patterns in the DNA sequence, and $PL(i)$ is the length of the ith synthetic pattern, $1 \leq i \leq PN$. $MinPL$ and $MaxPL$ are the minimum and maximum length of a pattern, respectively. That is, $MinPL \leq PL(i) \leq MaxPL$ and $MeanPL$ is the mean of the uniform distribution between $MinPL$ and $MaxPL$. $MinTL$ and $MaxTL$ are the minimum and maximum length of the DNA sequence constructed, respectively. $MeanTL$ is the mean of a uniform distribution between $MinTL$ and $MaxTL$. QN is the number of query sequences, and $QL(j)$ is the length of the jth query sequence, where $1 \leq j \leq QN$. $MinQL$ and $MaxQL$ are the minimum and maximum length of the query sequence, respectively. $MeanQL$ is the mean of the uniform distribution between $MinQL$ and $MaxQL$.

Table 5.1 Parameters

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

To model the phenomenon that DNA sequences often have repeating patterns, we generate different patterns first. Given PN, we generate PN distinct patterns. These patterns will be used when we generate the DNA sequence. For each of them, they may have different length. When generating a pattern, we randomly choose a number between $MinPL$ and $MaxPL$. This number is the value of $PL(i)$. That is, the length of the ith pattern. The pattern is composed of four characters, A, C, G, and T, the probability of the occurrence of these four characters is the same.

Next, we generate a DNA sequence which is stored in our data structure. We randomly choose a number between $MinTL$ and $MaxTL$. This number is the length of the DNA sequence. After deciding the length of the DNA sequence stored, we need to decide the contents of this sequence. The contents of the DNA sequence have five possible cases, A, C, G, T, and some patterns which we have generated. The parameter θ is to decide whether we need to use a pattern or not. We randomly choose a probability w between 0 and 1. If the probability w is less than parameter θ, we will randomly choose a pattern from those PN patterns and add it into the DNA sequence; otherwise, we randomly choose only one of these four characters A, C, G, and T. We repeat this process until the length of the DNA sequence is equal to the length which we have decided previously.

Finally, we generate query sequences after a database sequence S is generated. The length of each query sequence is determined by QL (from 50, 60, 70, 80, 90, 100, 150, and 200). To consider the successful exact match, we decide the starting position of the database sequence, which is a random number between 0 and $TL - QL$ (TL is the length of the database sequence S). (Note that a successful query means that the query sequence will match some subsequences stored in the database.) We repeat the process, which chooses a number, 200 times to generate 200 successful queries for each database sequence.

In this simulation, the value of PN, $PL(i)$, and θ is variable. We set $MinPL = 5$, $MaxPL = 10$, $MinTL = 10000$, $MaxTL = 12000$, $QN = 200$, $MaxQL = 50$, and $MaxQL = 200$ in our simulation. The simulation results is the average of 2000 experiments for different parameters which we consider.

5.2 Simulation Result

In the section, we show the simulation results. First, we show the comparison of the performance of the suffix tree and the ACGT-Words tree. Next, we show the comparison of the performance of the suffix array and the ACGT-Words tree. Finally, we show the comparison of the performance of the improved operations of the ACGT-Words tree and the ACGT-Words tree.

5.2.1 The Suffix Tree vs. the ACGT-Words Tree

A comparison of the number of nodes under the different probabilities of patterns occurrence is shown in Figure 5.1. 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. The detailed information about this comparison is shown in Table 5.2. Let NN be the number of nodes. The ratio in Table 5.2 is the percentage of $\frac{NN(suffix\ tree)-NN(ACGT-Words\ tree)}{NN(suffix\ tree)} = \alpha$. Given $\theta = 0.5$, we observe that our ACGT-Words tree can reduce 39% the number of nodes in the suffix tree. Moreover, as the occurring probability of patterns is increased, the number of nodes in the suffix tree is increased. But under the same consideration, the number of nodes is decreased in the ACGT-Words tree.

The ACGT-Words tree is a word-based data structure. The ACGT-Words tree only store one path for each different ACGT-Word (pattern). In the suffix tree, it stores one path for one suffix. As the probability of patterns is increased, the number of ACGT-Words is less than the number of suffixes. That is, as the probability of the repeat occurrence of patterns is increased, the ACGT-Words tree reduces more storage space than the suffix tree.

A comparison of the number of nodes under the different length of the database sequence is shown in Figure 5.2. 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

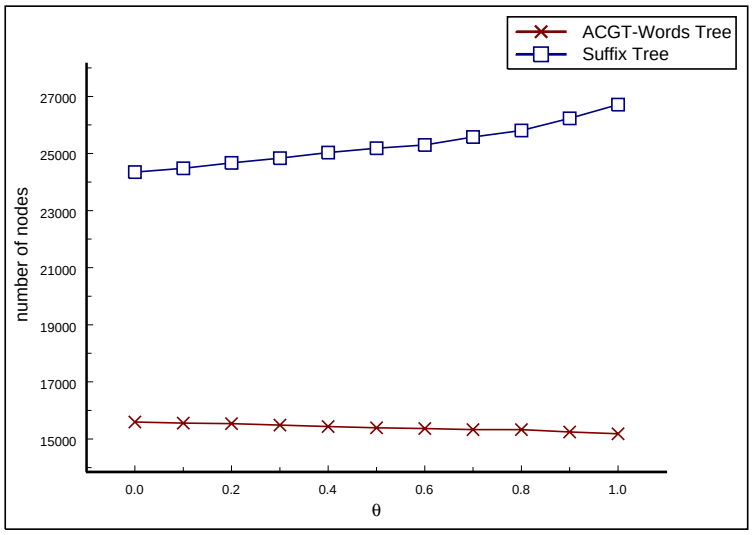


Figure 5.1 A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different probabilities of patterns ($TL = 15000$, $PN = 6$, $PL = 7$)

length of database sequence (TL) is. The detailed information about this comparison is shown in Table 5.3. From Table 5.3, given $TL = 15000$ (the mean length of the database sequence), we observe that the number of nodes in the ACGT-Words tree is 15381, while the number of nodes in the suffix tree is 25192. The ACGT-Word tree can reduce 39% the number of nodes in the suffix tree. Obviously, as TL is increased, the number of nodes in both data structures is increased. Moreover, as TL is increased, the rate of the increment of the suffix tree is faster than that of the ACGT-Words tree.

When the length of the database sequence (TL) is increased, the storage spaces of both the suffix tree and the ACGT-Words tree are increased. The reason is that they must use large storage space to store the large database sequences. As the database sequence is increased, some patterns may occur frequently. Therefore, in this case,

Table 5.2 A comparison of the number of nodes under the different probabilities of patterns ($TL = 15000$, $PN = 6$, $PL = 7$)

θ	ACGT-Words Tree	Suffix Tree	Ratio (α)
0.0	15592	24345	(37%)
0.1	15554	24482	(36%)
0.2	15534	24671	(37%)
0.3	15486	24833	(38%)
0.4	15434	25033	(38%)
0.5	15389	25184	(39%)
0.6	15363	25298	(39%)
0.7	15330	25577	(40%)
0.8	15330	25803	(41%)
0.9	15247	26230	(42%)
1.0	15181	26710	(43%)

the rate of the increment of the number of nodes of the ACGT-Words tree is slower than the suffix tree.

A comparison of the number of nodes under the different number of patterns is shown in Figure 5.3. 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 number of database patterns (PN) is. The detailed information about the comparison of number of patterns is shown in Table 5.4. From Table 5.4, given $PN = 6$ (the mean number of patterns), we observe that the number of nodes in the ACGT-Words tree is 15393, while the number of nodes in the suffix tree is 25195. The ACGT-Words tree can reduce 39% the number of nodes in the suffix tree.

The result of the number of patterns does not contain obvious regulation. Although we use different values of PN, these patterns which will occur or not are

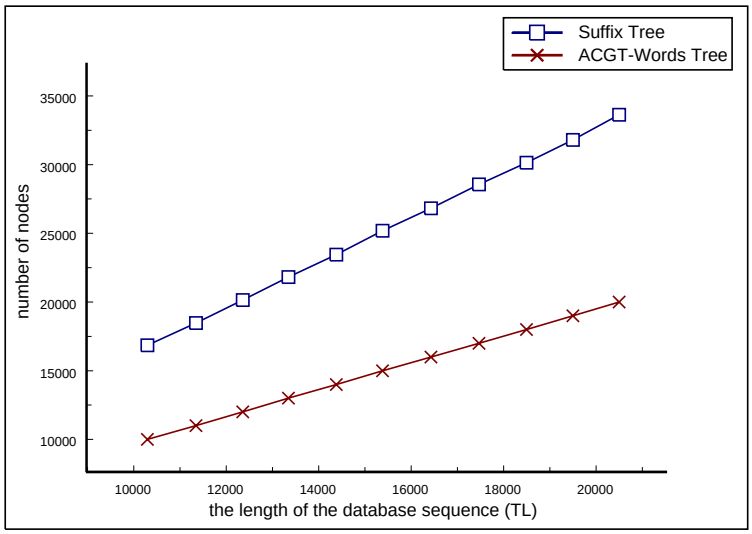


Figure 5.2 A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different length of the database sequence ($\theta = 0.5$, $PN = 6$, $PL = 7$).

determined by θ. Hence, the value of PN is not the main factor to affect the storage spaces of the suffix tree and the ACGT-Words tree.

A comparison of the number of nodes under the different length of patterns is shown in Figure 5.4. 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 length of patterns (PL) is. The detailed information about the comparison of the length of patterns is shown in Table 5.5. From Table 5.5, given $PL = 7$ (the mean length of patterns), we observe that the number of nodes in the ACGT-Words tree is 15411, while the number of nodes in the suffix tree is 25195. The ACGT-Words tree can reduce 39% the number of nodes in the suffix tree.

Similar to the result of the number of patterns, the result of the length of patterns does not contain obvious regulation. Although we use different values of PL, these

Table 5.3 A comparison of the number of nodes under the different length of the database sequence ($\theta = 0.5$, $PN = 6$, $PL = 7$)

TL	ACGT-Words Tree	Suffix Tree	Ratio (α)
10000	10294	16846	(39%)
11000	11346	18473	(39%)
12000	12356	20146	(39%)
13000	13343	21813	(39%)
14000	14378	23447	(39%)
15000	15381	25192	(39%)
16000	16426	26816	(39%)
17000	17465	28557	(39%)
18000	18491	30146	(39%)
19000	19498	31795	(39%)
20000	20496	33624	(39%)

patterns which will occur or not are determined by θ. Hence, the value of PL is not the main factor to affect the storage spaces of the suffix tree and the ACGT-Words tree.

5.2.2 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 5.5. 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 value of the length of database sequences is. The detailed information about constructing time shown in Table 5.6. From Table 5.6, we observe that the suffix array is 2-5 times more expensive to build to the ACGT-Words tree. Obviously, as TL is increased, the constructing time in both data structures is increased. Moreover,

Table 5.4 A comparison of the number of nodes under the different number of patterns ($\theta = 0.5$, $TL = 15000$, $PL = 7$)

PN	ACGT-Words Tree	Suffix Tree	Ratio (α)
1	15319	25593	(40%)
2	15342	25509	(40%)
3	15335	25454	(40%)
4	15423	25230	(39%)
5	15396	25277	(39%)
6	15393	25195	(39%)
7	15434	25134	(39%)
8	15443	25021	(38%)
9	15411	25053	(38%)
10	15427	25110	(39%)

Table 5.5 A comparison of the number of nodes under the different length of patterns ($\theta = 0.5$, $PN = 6$, $TL = 15000$)

PL	ACGT-Words Tree	Suffix Tree	Ratio (α)
5	15493	25011	(38%)
6	15444	25114	(39%)
7	15411	25195	(39%)
8	15389	25251	(39%)
9	15361	25462	(40%)
10	15318	25319	(39%)

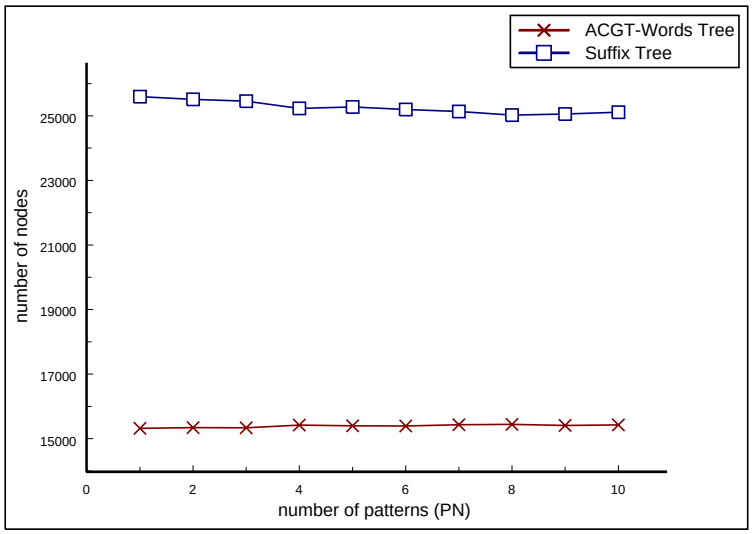


Figure 5.3 A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different number of patterns ($\theta = 0.5$, $TL = 15000$, $PL = 7$).

as TL is increased, the rate of the increment of the suffix array is faster than that of the ACGT-Words tree.

When the suffix array is constructed, it needs to sort all suffixes in the database sequence first. Hence, it will take much time and require more space to construct its data structure. In the ACGT-Words tree, we traverse the tree structure to insert all ACGT-Words. We do not take time to sort the suffixes. Therefore, the ACGT-Words tree has better performance than the suffix array in the constructing process.

A comparison of the searching time under the different length of query sequences is shown in Table 5.7. 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 traverse the tree structure to find the query sequence. We search

Table 5.6 A comparison of the constructing time under the different length of database sequence ($\theta = 0.5$, $PN = 6$, $PL = 7$)

TL	ACGT-Words Tree (ms)	Suffix Array (ms)
10000	1885	5453
11000	2190	6657
12000	2905	7906
13000	4070	9281
14000	4315	10844
15000	5082	12390
16000	5308	14204
17000	5717	15891
18000	7347	32093
19000	8494	41515
20000	8767	49359

Table 5.7 A comparison of the searching time between the suffix array and the ACGT-Words tree under the different length of query sequences

QL	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

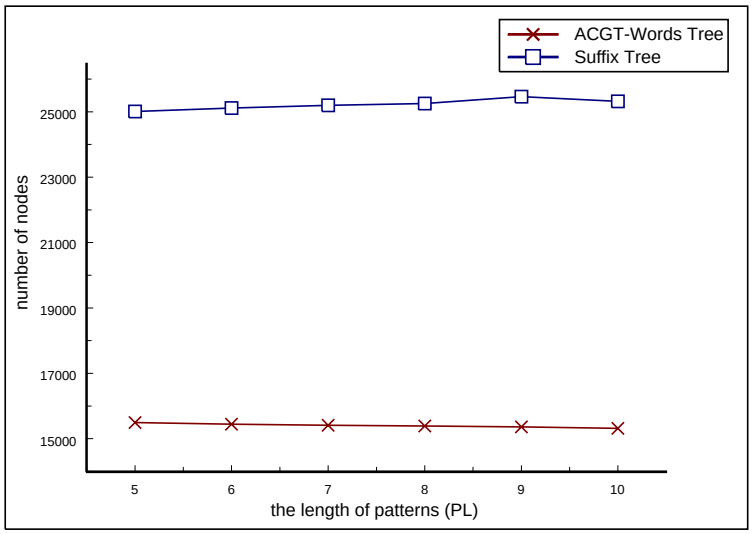


Figure 5.4 A comparison of the number of nodes between the ACGT-Words tree and the suffix tree under the different length of patterns ($\theta = 0.5$, $PN = 6$, $TL = 15000$).

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.

5.2.3 The ACGT-Words Tree vs. the Improved the Insertion Operation

In this section, we study the performance of the improved operation for insertion, and make a comparison of the constructing time with the original insertion operation of the ACGT-Words tree. A comparison of the constructing time based on the different length of database sequences is shown in Table 5.8. Obviously, this operation for improved insertion operation takes less time for the constructing process than the original insertion operation. The reason is that we sort the ACGT-Words by the

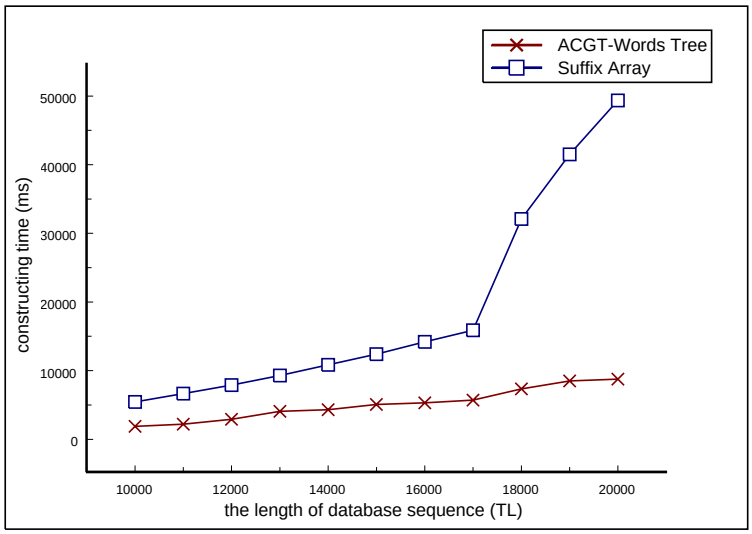


Figure 5.5 A comparison of the constructing time between the ACGT-Words tree and the suffix array under the different length of database sequence

lexicographic order before constructing the tree structure in the improved insertion operation. After an ACGT-Word W is inserted into the tree, we will check whether the remaining ACGT-Words contain the same word W. We only use a string comparison to reach this goal. Hence, we do not take time to traverse the same path in the tree structure again. Therefore, the constructing time of the improved insertion operation is shorter than that of the original insertion operation.

5.2.4 The ACGT-Words Tree vs. the Improved the Search Operations

In Chapter 4, we consider two cases for improving the searching process. First, we consider the improved operation (Case 1), the query sequence may occur that the

Table 5.8 A comparison of the constructing time between the ACGT-Words tree and the improved insertion operation of ACGT-Words tree under the different length of database sequences

TL	ACGT-Words Tree (ms)	Improved the Insertion (ms)
10000	1885	264
11000	2190	242
12000	2905	336
13000	4070	356
14000	4315	359
15000	5082	304
16000	5308	403
17000	5717	415
18000	7347	412
19000	8494	419
20000	8767	442

last character is same as the first character in the query sequence. A comparison of the searching time based on the different length of query sequences is shown in Table 5.9. Obviously, this improved operation for the searching process takes less time for searching than the original searching process.

Let's consider another improved operation in Chapter 4 (Case 2), the query sequence may contain successive same characters. A comparison of the searching time under the different length of query sequences is shown in Table 5.10. Obviously, this improved operation for the searching process always take less time for searching than the original searching process no matter what value of the length of query sequences.

Finally, let's consider the storage space, the construct time, the search time, and update time. A comparison of the suffix tree, the suffix array, and the ACGT-Words tree are summarized in Table 5.11. From this table, we conclude that our indexing

Table 5.9 A comparison of the searching time between the original searching process and the improved searching process (Case 1) under the different length of query sequences

QL	ACGT-Words Tree (ms)	Case 1 (ms)
50	36	28
60	65	38
70	91	46
80	55	29
90	76	56
100	76	53
150	88	68
200	101	77

Table 5.10 A comparison of the searching time between the original searching process and the improved searching process (Case 2) under the different length of query sequences

QL	ACGT-Words Tree (ms)	Case 2 (ms)
50	36	35
60	65	46
70	91	55
80	55	36
90	76	52
100	76	62
150	88	69
200	101	88

Table 5.11 A comparison

	Suffix tree	Suffix array	ACGT-Words tree
Storage space	3rd	1st	2nd
Construct time	1st	3rd	2nd
Search time	1st	3rd	2nd
Update time	Faster than the suffix array	Slower than others	Faster than others

Table 5.12 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

structure, the ACGT-Words tree, needs less storage space than the suffix tree. Moreover, the ACGT-Words tree outperforms the suffix array in the construct time and the search time. Furthermore, when the incremental update operation occurs, we need to update parts of the tree structure. Hence, we have the update time of the ACGT-Words tree which is shorter than others indexing structure.

5.3 Performance Result for the Input from Genomic Databases

In this section, we study the performance of using the real DNA sequences in GenBank (http://www.ncbi.nlm.nih.gov). We use Drosophila and human Chromosome 19 to be our database sequence. Table 5.12 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.

CHAPTER VI

Conclusion

In this thesis, we have proposed the ACGT-Words tree for indexing the genomic database sequences. In this Chapter, we give a summary of this thesis and point out some future research directions.

6.1 Summary

Genomic sequence databases are widely used by molecular biologists for searching. These databases assist molecular biologists in understanding the biochemical function, chemical structure of organisms. Popular systems for searching genomic databases match queries to answer by comparing a query to each of the sequences in the database [29]. However, with the increasing number of users and the rapid increase in the number of stored sequences in genomic databases, existing algorithms for finding answers use exhaustive search are becoming prohibitively expensive. Similar to conventional databases, there are some approaches use indexing to provide fast access to genomic sequences, such as the suffix tree [26], the suffix array [20], the suffix binary search tree [13, 14], the word suffix tree [3], and so on. The storage space and the search time is a tradeoff in these indexing algorithms.

In Chapter 2, we have given a survey of some indexing data structures, including the suffix tree [26], the suffix array [20], the suffix binary search tree [13, 14], the word suffix tree [3], respectively.

In Chapter 3, we have given the definition of ACGT-Words and proposed a new data structure for indexing the genomic sequences. A sequence beginning with a character k, $k \in \{A, C, G, T\}$, the successive characters y_i after it until the same

character k appearing again or the end symbol \$ appearing, the sequence constructed by k and y_i is an ACGT-Word. In the proposed ACGT-Words tree, we insert the ACGT-Words generated, instead of all suffixes, into the tree structure. We have proposed a different way to construct and search in the ACGT-Words tree. We can compact the index structure and also provide the same results with other index structures.

In Chapter 4, we have proposed an improved operation for the insertion process. We sort the ACGT-Words generated before constructing the tree structure. We do not the tree structure repeatedly when the database sequence contains same ACGT-Words. Moreover, we have discovered two improved operations (Case 1 and Case 2) to efficiently search in the ACGT-Words tree when the query sequences match the conditions of these two operations.

In Chapter 5, we have studied the performance of the proposed ACGT-Words tree and the improved operations in the ACGT-Words tree. Moreover, we have made a comparison with the suffix tree and the suffix array by simulation. We have shown that the storage space of the ACGT-Words tree is less than that of the suffix tree. Moreover, from our simulation results, we have shown that the construct time and the search time of the ACGT-Words tree are shorter than those of the suffix array. Furthermore, we also have shown that the improved operation for insertion outperforms the original insertion process and the improved operations for search process take less time than the original search process.

6.2 Future Work

In this thesis, we have designed an efficient data structure for indexing the genomic database. How to efficiently construct the index structure and search the query sequences are the major works. In the future, we plan to extend our index structure to support the *approximate sequence matching* in database sequences. The problem of approximate sequence matching is formally stated as follows: given a long sequence $S = s_0 s_1 ... s_{n-1}$ of length n and a comparatively short pattern $P = p_0 p_1 ... p_{m-1}$ of

length m, both sequences over an alphabet $\sum$ of size σ, find the sequence positions that match the pattern with at most k "errors" [23]. For example, the query sequence $P = ACAGT$ matches one subsequence of the database sequence $S = \underline{AC}AT\underline{AGT}$ with at most 2 errors. (Note that the query sequence is the pattern.) Moreover, we plan to provide the incremental update operation, which does not reconstruct the tree structure when the database sequence are changed. Furthermore, we plan to provide efficiently constructing and searching processes in the ACGT-Words tree, such as other improved operations and parallel processing to construct and search the query sequences.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology,* Vol. 215, pp. 403–410, Oct. 1990.

[2] A. Andersson and S. Nilsson, "Efficient Implementation of Suffix Trees," *Software Practice and Experience,* Vol. 25, No. 2, pp. 129–141, Feb. 1995.

[3] A. Andersson, N. J. Larsson, and K. Swanson, "Suffix Trees on Words," *Algorithmica,* Vol. 23, No. 3, pp. 246–260, Jan. 1999.

[4] S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, and M. Vingron, " q-gram Based Database Searching Using a Suffix Array (QUASAR)," *Annual Conf. on Research in Computational Molecular Biology,* pp. 77–83, 1999.

[5] W. Chen and K. Aberer, "Efficient Querying on Genomic Databases by Using Metric Space Indexing Techniques," *Proc. of 8th Int. Conf. and Workshop on Database and Expert Systems Application,* pp. 148–152, 1997.

[6] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of Whole Genomes," *Nucleic Acid Research,* Vol. 27, No. 11, pp. 2369–2376, June 1999.

[7] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," *The 38nd Annual Symposium on Foundations of Computer Science,* pp. 137–143, 1997.

[8] S. Ganguly and M. Noordewier, "Proximal: A Database System for the Efficient Retrieval of Genetic Information," *Computer in Biology and Medicine,* Vol. 26, No. 3, pp. 199–207, May 1996.

[9] D. Gusfield, *"Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology,"* Cambridge University Press, 1997.

[10] J. L. Houle, W. Cadigan, S. Henry, A. Pinnamanenib, and S. Lundahlc, "Database Mining in the Human Genome Initiative,"

http://www.biodatabases.com/whitepaper01.html.

[11] E. Hunt, R. W. Irving, and M. Atkinson, "Persistent Suffix Trees and Suffix Binary Search Trees as DNA Sequence Indexes," *Technical Report no. TR-2000-63 of the Computing Science Department of Glasgow University, October 2000.*

[12] E. Hunt, M. P. Atkinson, and R. W. Irving, "A Database Index to Large Biological Sequences," *Proc. of the 27th VLDB Conf.,* pp. 139–148, 2001.

[13] R. W. Irving and L. Love, "The Suffix Binary Search Tree and Suffix AVL Tree," *Technical Report no. TR-2000-54 of the Computing Science Department of Glasgow University, July 2000.*

[14] 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.*

[15] T. Kahveci and A. K. Singh, "An Efficient Index Structure for String Databases," *Proc. of the 27th VLDB Conf.,* pp. 351–160, 2001.

[16] J. Karkkainen and E. Sutinen, "Lempel-Ziv Index for q-Grams," *Algorithmica,* Vol. 21, No. 1, pp. 137–154, May 1998.

[17] T. Kasai, H. Arimura, and S. Arikawa, "Efficient Substring Traversal with Suffix Arrays," *DOI Technical Report of the Informatics Department of Kyushu University, February 2001.*

[18] C. T. Lee, "Computational Biology,"
http://www.csie.ncnu.edu.tw/∼rctlee/biology.html.

[19] D. J. Lipman and W. R. Pearson, "Rapid and Sensitive Protein Similarity Searches," *Science,* Vol. 227, pp. 1435–1441, March 1985.

[20] 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.

[21] G. Navarro and R. Baeza-Yates, "A Practical q-Gram Index for Text Retrieval Allowing Errors," *CLEI,* Vol. 1, No. 2, pp. 273–282, Dec. 1998.

[22] G. Navarro, *"Modern Information Retrieval,"* Addison Wesley/ACM Press, Reading, MA, pp. 191–228, 1999.

[23] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio, "Indexing Methods for Approximate String Matching," *IEEE Data Eng. Bulletin,* Vol. 24, No. 4, pp. 19–27, Dec. 2001.

[24] R. Shamir, "Algorithms for Molecular Biology,"
http://www.math.tau.ac.il/∼rshamir/algmb/01/algmb01.html.

[25] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology,* Vol. 147, pp. 195–197, March 1981

[26] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory,* pp. 1–11, 1973.

[27] 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.

[28] H. Williams and J. Zobel, "Indexing Nucleotide Databases for Fast Query Evaluation," *Int. Conf. on Extending Database Technology,* pp. 275–288, 1996.

[29] 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.

[30] M. Zhou and F. W. Tompa, "The Suffix-Signature Method for Searching for Phrases in Text," *Information Systems,* Vol. 23, No. 8, pp. 567–588, Dec. 1998.

Appendixes