A LOCALIZED CLUSTERING ALGORITHM AND ITS APPLICATION TO DNA STRING PROCESSING

By

JUN HUAN

Bachelor of Science

Peking University

Beijing, China

1997

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2000

A LOCALIZED CLUSTERING ALGORITHM AND ITS APPLICATION TO DNA STRING PROCESSING

Thesis Approved:

[illegible]
Thesis Advisor

[illegible]

[illegible]

[illegible]
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. K. M. George for his intelligent supervision, constructive guidance, inspiration and friendship. My sincere appreciation extends to my other committee members Dr. G. E. Hedrick and Dr. H. K. Dai, whose guidance, assistance, encouragement and friendship are invaluable.

More Over, I wish to express my sincere gratitude to Dr. Ross Overbeek and Dr. Natalie Maltsev in Mathematics and Computer Science Division, Argonne National Laboratory and Dr. R. Burnap in Microbiology Department, Oklahoma State University for suggestion and assistance for this study.

I would also like to give my special appreciation to my parents Shuyu Huan and Chuanhui Gu as well as my sister Zhen Gu for their love, support and encouragement.

I would like to thank the Department of Computer Science for supporting during these two years of study.

Finally, I want to say "Thanks" to all my friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Ω	Ordered set of n objects {1, 2, .., n}.
κ	A positive integer denotes the selected frequency threshold.
ρ	Constant denoting a pattern's length.
D	D is a distance matrix of an ordered set of clusters (objects).
DNA	A n-character string composed of four characters: A, T, G, C.
d(i, j)	Distance between two objects i and j.
$df(CL_1, CL_2)$	Distance function used to calculate the distance between two clusters CL_1 and CL_2.
HF	A mapping of a pattern to an integer called index.
Htr	A mapping of a single character to an integer.
IT	A table of integers where IT[i] is the number of patterns mapping to index i.
Sd	Shift distance of two patterns.
Str	A pattern from a DNA.
ST	A string hash table of patterns from a DNA.
T	An n-tree obtained from objects-set Ω.
T_1	A set of clusters obtained from Ω.

CHAPTER I

INTRODUCTION

Clustering is concerned with the investigation of a set of objects in order to establish whether or not they fall naturally into groups of objects with the property that objects in the same group are similar to one another and different from objects in other groups ([4], [24]).

Hierarchical clustering is the most popular clustering algorithm. The hierarchical algorithm builds an n-tree based on an $n \times n$ distance matrix ([3], [4]).

In certain applications such as the DNA string analysis, a set of objects usually contains hundreds of thousands of objects ([17]). The limited memory becomes a restriction for applying the hierarchical clustering to these applications because of the $n \times n$ distance matrix. In this research, the author proposes a localized clustering algorithm. This algorithm is designed for clustering patterns from a DNA sequence. Compared to the classical hierarchical clustering algorithm ([4]), the proposed algorithm is both memory-saving and time-efficient.

The organization of this thesis is the following:

In chapter II, the biological background is presented.

In chapter III, the literature review is presented.

In chapter IV, the related research is presented and discussed.

The design of the localized clustering algorithm is discussed in chapter V.

Implementation and result analyses are presented in chapter VI.

Chapter VII gives a summary of the work and discusses future work.

CHAPTER II

BIOLOGICAL BACKGROUND

Through molecular biology research, it is clear that most of the biological functions are carried by proteins ([2], [22]). A Protein is synthesized using the information stored in its DNA template. DNA is a long string composed of 4 elements. The sequence of the DNA specifies the protein sequence ([2]). Protein synthesis is initialized by a class of proteins called *transcription factors*. The transcription factor recognizes a short DNA sequence and binds to it. After binding using the transcription factor, more proteins come to form a protein complex to start the transcription ([2], [22]). The short DNA sequence to which the transcription factor binds to is known as *regulatory element*. Regulatory Elements are considered as the key to understand the regulation of the protein synthesis ([2]).

Mutations (DNA sequence's change) are phenomena that occur in living organisms. Mutation accumulates during evolution ([2]). By accumulations of mutations, several regulatory elements were derived from a common regulatory element in the evolution. Clustering algorithm, which is concerned with the investigation of a set of objects in order to establish whether or not they fall naturally into groups, becomes a very useful tool to analyze the set of regulatory elements. This helps greatly in understanding the evolutionary process.

CHAPTER III

LITERATURE REVIEW

3.1 Introduction

A *cluster* is a set of objects that are grouped together by certain rules ([24]). The goal of clustering is to find groups containing objects that are most homogeneous to each other within the groups while these groups are most heterogeneous to each other. The term *homogeneous* and *heterogeneous* refer to the common and distinguishing properties of the objects according which clustering is done on the given set of objects ([4], [24]).

The set of objects that are to be investigated in a clustering study are generally ordered and described by a profile matrix ([4], [24]). In this thesis, we use the term *objects-set* to denote a set of objects (with an ordering).

A *profile matrix* of an objects-set is a n $\times$ p matrix $H \equiv (h_{ik})$, where h_{ik} denotes the value taken by the kth variable describing the ith object ($i = 1, \ldots, n; k = 1, .., p$).

An objects-set with four objects distributed in a two-dimensional space is shown in figure 1b. The objects-set's profile matrix is shown in figure 1c.

Let Ω be the ordered set of n objects $\{1, 2, \ldots, n\}$, an *n-tree* from Ω is a set T of subsets of Ω satisfying the conditions ([1], [14]):

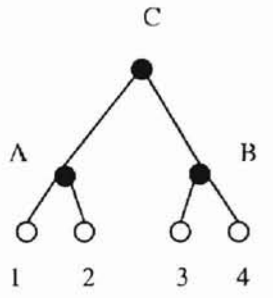
1 $\Omega \in T$

2 $\phi \notin T$

3 $\{i\} \in T$ for all $i \in \Omega$

4 if $A, B \in T$, then $A \cap B \in \{A, B, \phi\}$

Condition 4 ensures that the subsets are hierarchically-nested. This is illustrated in the rooted tree diagram shown in figure 1a, in which each leaf (depicted by an open circle) represents an object, and each internal node (depicted by a filled circle) represents a non-singleton subset of objects. In addition to the complete set $\Omega = \{1, 2, 3, 4\}$ and the singleton subsets $\{i\}$ $(i = 1, 2, 3, 4)$, which are present in all n-trees, this diagram comprises the subsets $\{1, 2\}$ and $\{3, 4\}$, that are represented by the internal nodes A and B respectively. In other words, this n-tree T is $T = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{3, 4\}, \{1, 2, 3, 4\}\}$. The corresponding distribution of the objects is presented in figure 1b.



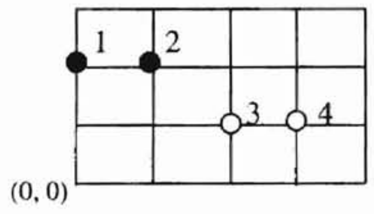


Figure 1a. An n-tree from the objects-set $\Omega = \{1, 2, 3, 4\}$ Figure 1b. Two-dimensional objects distribution

Object	X_{i1}	X_{i2}
1	0	2
2	1	2
3	2	1
4	3	1

Figure 1c. A profile matrix of objects-set {1, 2, 3, 4}

The *distance* between two objects (clusters) denotes any meaningful measurement of the differences between those two objects (clusters, respectively), given that the measurement satisfies the following conditions ([24]):

- $d_{i,j} >= 0$
- $d_{i,i} = 0$
- $d_{i,j} = d_{j,i}$

where $d_{i,j}$ denotes the distance between the ith and jth objects (clusters) in the objects-set (clusters) (i, j = 1, 2, .., n).

One example of the distance between two objects is the *Minkowski distance* $\mathbf{d_m}$:

$$d_{mij} = [\sum_{k=1}^{p}(x_{ik} - x_{jk})^{m}]^{(1/m)} \quad (i, j = 1, 2, \ldots, n, x_{ik} \text{ is the kth variable describing the ith object in the objects-set})$$

If m = 1, the *Manhattan distance* is obtained, and if m= 2, the *Euclidian distance* is obtained ([24]). We use **d(i, j)** to denote the distance between two objects i and j in Ω (i, j = 1, 2, …, n).

Examples of the distance between two clusters such as single-link, complete-link and Ward's method are discussed in sections 3.3, 3.4 and 3.7 respectively. We call the function which is used to calculate the distance between two clusters as *distance function* ([24]). We use **df** to denote a distance function and use $\mathbf{df(CL_1, CL_2)}$ to denote the distance between two clusters CL_1 and CL_2.

The distance between an object i and a cluster CL_p is defined to be the distance between the singleton cluster {i} and CL_p.

The distance between two singleton clusters is defined to be the distance between the two objects in the clusters. That is:

df ({i}, {j}) = d(i, j) where i, j = 1, 2, …, n.

Let S be an ordered set of n clusters (or objects), a *distance matrix* D calculated for S is an $n \times n$ matrix $D \equiv (d_{ij})$, where d_{ij} denotes the distance between the ith and jth clusters (or objects) in S.

An example of the distance matrix for an ordered set of clusters S = {{1}, {2}, {3}, {4}} is shown in figure 2. The corresponding data distribution is shown in figure 1b.

Cluster	{1}	{2}	{3}	{4}
{1}	0	1	2.24	3.16
{2}	1	0	1.41	2.24
{3}	2.24	1.41	0	1
{4}	3.16	2.24	1	0

Figure 2. Distance matrix for an ordered set of clusters {{1}, {2}, {3}, {4}}

3.2 Hierarchical Clustering Algorithm Model

The entire group of hierarchical clustering algorithm can be presented as a four-step procedure described below ([4], [24]):

1. Calculate the $n \times n$ distance matrix $D = (d_{ij})$ from the ordered set of clusters $T_1 = \{\{1\}, \{2\}, .., \{n\}\}$ on objects-set $\Omega = \{1, 2, \ldots, n\}$. Initialize clusters set T_2 to empty set ϕ.

2. Find the minimal positive distance in the distance matrix and let I and J be the corresponding elements. Form a new cluster $CL = I \cup J$. Copy the remaining elements in T_1 to T_2. Add CL to T_2.

3. Replace T_1 by T_2. Calculate a new distance matrix for T_1.

4. If cardinality of T_1 is greater than 1, go to step2; else stop.

The following is an example of applying this hierarchical clustering algorithm to the objects-set $\Omega = \{1, 2, 3, 4\}$ shown in figure 1b. There are three iterations in this example.

For the first iteration, the distance matrix is presented in figure 2. The minimal positive distance in this matrix is given by df({1},{2}) and df({3}, {4}), where df({1}, {2}) denotes the distance between the singleton clusters {1} and {2} and df({3}, {4}) denotes the distance between the singleton clusters {3} and {4}. Both have value 1. Arbitrary decision has to be made here ([24]). If we chose to form a cluster $CL_{12} = \{1, 2\}$, then the distance matrix is re-calculated and presented in figure 3.

The single-link algorithm is used in this calculation. This algorithm is discussed in section 3.3. Using this algorithm, the distance between the cluster CL_{12} and singleton cluster {3} is defined as the smaller value of d(1, 3) and d(2, 3), where d(1, 3) denotes the distance between the objects 1 and 3 and d(2, 3) denotes the distance between the objects 2 and 3. Because d(2, 3) $= 1.41 < d(1, 3) = 2.24$, the distance between the cluster CL_{12} and singleton cluster {3} is 1.41.

Cluster	CL_{12}	{3}	{4}
CL_{12}	0	1.41	2.24
{3}	1.41	0	1
{4}	2.24	1	0

Figure 3. Distance matrix for $T_1 = \{CL_{12}, \{3\}, \{4\}\}$

For the second iteration, the minimal positive distance in the distance matrix is df({3}, {4}). New cluster CL_{34} ({3, 4}) is formed and the distance matrix is updated. This updated distance matrix is presented in figure 4.

Cluster	CL_{12}	CL_{34}
CL_{12}	0	1.41
CL_{34}	1.41	0

Figure 4. Distance matrix for $T_1 = \{CL_{12}, CL_{34}\}$

In the third iteration, the two clusters CL_{12} and CL_{34} are joined and thus form a cluster $CL_{1234} = \{1, 2, 3, 4\}$. After the third iteration, the ordered set of clusters T_1, contains only one element, CL_{1234}. Therefore, the hierarchical clustering stops. The resulting n-tree is shown in figure 1a.

We discuss several classical clustering algorithms which follow the procedure presented above. The differences are in the distance functions they use. These clustering algorithms are described in section 3.3 - 3.7.

3.3 Single-Link

The single-link algorithm ([7], [9], [19]) is the first clustering algorithm proposed in the literature. The distance between two clusters CL_p and CL_q is defined as the shortest distance between two objects, one from each cluster. The distance function is defined as:

$$df(CL_p, CL_q) = \min\{ d(i,j) \}, \quad \text{where } i \in CL_p \;\&\; j \in CL_q$$

For example, consider the objects-set shown in figure 5. Assume there are two clusters CL_{12} ({1, 2}) and CL_{34} ({3, 4}). The distance between CL_{12} and CL_{34} is d(2, 3), which denotes the distance between the objects 2 and 3, and that is 1.41.

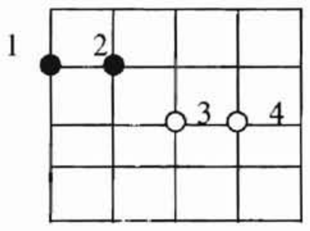


Figure 5. An example of clusters
$CL_{12} = \{1, 2\}$ and $CL_{34} = \{3, 4\}$
When single-linkage is used: $df(CL_{12}, CL_{34}) = d(2, 3)$
$= 1.41$
When complete-linkage is used: $df(CL_{12}, CL_{34}) = d(1, 4)$
$= 3.16$

3.4 Complete-Link

Complete-link is a modification of the single-link algorithm ([7], [9]). The distance between two clusters is defined as the largest distance between two objects, one from each cluster. The distance function is:

$$df(CL_p, CL_q) = \max\{ d(i,j) \}, \quad \text{where } i \in CL_p \;\&\; j \in CL_q$$

In the distribution shown in figure 5, the distance between the cluster CL_{12} and CL_{34}, using complete-link, is d(1, 4) and that is 3.16.

3.5 Unweighted Group Average Method (UWGAM)

The distance function is the average of the distances between the two clusters. The distance between two clusters is given by the formula ([10], [15], [20]):

df (CL_p, CL_q) = mean { d(i, j) }, where $i \in CL_p$ & $j \in CL_q$ and

mean represents the average of all d(i, j)

3.6 Weighted Group Average Method (WGAM)

In WGAM, each cluster is given equal weight. In UWGAM, each object is given equal weight ([20]).

To describe the distance function, we need the following definition:

Cluster A is a *maximal-subset* of cluster B if and only if $A \subset B$ and there is no other cluster C satisfying the relation $A \subset C \subset B$. For example, consider the n-tree defined on objects-set $\Omega = \{1, 2, 3, 4, 5, 6, 7\}$ presented in figure 6. Considering cluster E ({1, 2, 3, 4, 5}), cluster A ({1, 2, 3}) and cluster C ({4, 5}), A and C are E's maximal-subsets while cluster B ({2, 3}) is not a maximal-subset of E because we have the relation $B \subset A \subset E$.

The distance between two clusters is defined recursively as:

(1) df ({i}, {j}) = d(i, j) (i, j = 1, 2,..., n) where d(i, j) denotes the distance between the ith and jth objects in the original objects-set.

(2) df (CL_p, CL_q) = mean { df (CL_1, CL_2) }, for every pair (CL_1, CL_2), CL_1 is a maximal-subset of CL_p and CL_2 is a maximal-subset of CL_q.

The following example shows the difference between WGRAM and UWGRAM in the calculation of distance between clusters D and E in figure 6.

$$\begin{aligned}
\text{By UWGAM, } df(D, E) &= 1/10\,(d(1, 6) + d(2, 6) + d(3, 6) + d(4, 6) + \\
&\qquad d(5, 6) + d(1, 7) + d(2, 7) + d(3, 7) + \\
&\qquad d(4, 7) + d(5, 7)) \\
&= 1/10\, d(1, 6) + 1/10\, d(2, 6) + 1/10\, d(3, 6) + 1/10\, d(4, 6) \\
&\quad + 1/10\, d(5, 6) + 1/10\, d(1, 7) + 1/10\, d(2, 7) + 1/10\, d(3, 7) \\
&\quad + 1/10\, d(4, 7) + 1/10\, d(5, 7)
\end{aligned}$$

$$\begin{aligned}
\text{By WGAM } \quad df(D, E) &= 1/4\,(\, df(A, \{6\}) + df(A, \{7\}) + df(C, \{6\}) + df(C, \{7\})\,) \\
&= 1/4\,(\, 1/2\,(d(1, 6) + df(B, \{6\})) + 1/2\,(d(1, 7) + df(B, \{7\})) + \\
&\quad 1/2\,(d(4, 6) + d(5, 6)) + 1/2\,(d(4, 7) + d(5, 7))\,) \\
&= 1/8\, d(1, 6) + 1/16\, d(2, 6) + 1/16\, d(3, 6) + 1/8\, d(4, 6) + \\
&\quad 1/8\, d(5, 6) + 1/8\, d(1, 7) + 1/16\, d(2, 7) + 1/16\, d(3, 7) + \\
&\quad 1/8\, d(4, 7) + 1/8\, d(5, 7)
\end{aligned}$$

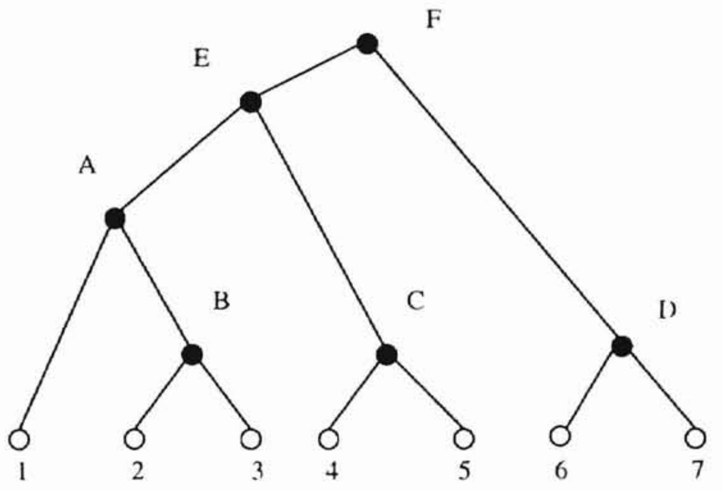


Figure 6. An n-tree defined on objects-set {1, 2, 3, 4, 5, 6}

3.7 Ward's Method

Ward's method is based on statistical minimization of clustering's "expansion". At each step, a central point is calculated for any possible combination of two clusters and then the sum of squared distances from this point to any objects in those two clusters are evaluated. This value is defined as the distance between those two clusters. If we use CP to represent the central point of that two clusters CL_p, CL_q, then the distance between the two clusters is defined as ([23]):

$$df(CL_p, CL_q) = \sum_{i} [d(i, CP)]^2, \quad i \in CL_p \cup CL_q$$

CHAPTER IV

RELATED RESEARCH

4.1 Introduction

The research undertaken in this thesis is based on previous unpublished research [17]. In [17], an algorithm to obtain all high frequency patterns from a DNA sequence is described. Because of its closeness to the current research, the algorithm is described in detail in this section. For other algorithms designed for obtaining the "repeated" substrings, the reader is referred to [21].

A DNA molecule can be described by an n-character string **DNAS**, which is composed of characters 'A', 'C', 'G' and 'T'. We call the string DNAS as the *DNA sequence*. For simplicity, we also call the DNAS as a *DNA*. We use DNA[i, j] to denotes the substring that starts at position i and ends at position j in the DNA ($i, j = 1, 2, \ldots, n; i<=j$). We use |DNA| to denote the length of a DNA.

A *pattern* in a DNA is a substring in the DNA with a fixed-length ρ ($\rho>0$). The *frequency* of a string, Str, is defined as its number of distinct occurrences in the DNA. The frequency of substring Str is denoted by $\mathbf{f_{Str}}$.

We use the term *high frequency pattern* (**HFP**) to denote any pattern that has at least two distinct occurrences in the DNA. An *n-frequency pattern* is any pattern that has at least n distinct occurrences in the DNA (n >1). In the following discussion, only n-frequency pattern is used. Therefore, for simplicity, we use the term "high frequency pattern" to refer to an n-frequency pattern.

A *high frequency pattern list* (**HFPL**) of a DNA is the set of all high frequency patterns of the DNA sorted in descending order by the patterns' frequencies. We use HFPL[i] to denote the ith high frequency pattern in the HFPL. The size of the list HFPL is denoted by size(HFPL).

Each pattern in a DNA can be mapped to an integer called *index*. The mapping is discussed below. Before we introduce the mapping of a pattern to its index, we first introduce the mapping of a single character to an integer. We call this mapping as the *translation function* (**Htr**). **Htr**: $A \rightarrow 0$, $C \rightarrow 1$, $G \rightarrow 2$, $T \rightarrow 3$ is used in this work.

HF is the mapping of a pattern to its index. We define HF by an algorithm. HF receives as input a pattern, Str, and returns the index of the pattern. In the following discussion, variable **Ch** is a single character variable and φ is a positive integer constant. The specific value of φ depends on applications. Generally, φ should be a primer number in the following hash function. We use |Str| to denote the pattern Str's length. The function HF is defined below:

```
Function HF(Str)
    $i \leftarrow 1$ ; $Ch \leftarrow Str[i]$; $index \leftarrow$ Htr(Ch);
    loop while ($i <= |Str|$ )
```

```
        i ← i + 1; Ch ← Str[i];
    index ← ( index × 4 + **Htr**(Ch) ) mod φ;
    **end loop**;
    **return** index;
**end;**
```

We consider the pattern "TTAC" and calculate its index using the above algorithm as an example. Suppose the constant φ is 7. The first character is "T" and it is mapped to integer 3 by translation function Htr $A \rightarrow 0$, $C \rightarrow 1$, $G \rightarrow 2$, $T \rightarrow 3$. The variable index is initialized to 3. The second character is still "T" and it is mapped to 3. The index is $(3 \times 4 + 3) \bmod 7 = 1$. The third character is "A" and mapped to 0. The index is $(1 \times 4 + 0) \bmod 7 = 4$. The last character is "C" and mapped to 1. The index is $(4 \times 4 + 1) \bmod 7 = 3$. The calculation stops and three is the index of pattern "TTAC"; i.e. HF("TTAC") = 3.

4.2 High Frequency Pattern Algorithm

Identifying all the high frequency patterns in a DNA sequence with a given frequency threshold is our research interest. We call this problem as "High Frequency Pattern Problem" ([17]) and the problem may be formulated as follows ($|s|$ denotes the length of the string s):

Given a DNA with length $|DNA| = n$ ($n > 0$), and an integer κ ($\kappa > 1$), identify all patterns p with $|p| = \rho$ ($\rho > 0$ and $\rho < n$) in the DNA such that p's occurrences $f_p >= \kappa$.

We designed a three-step algorithm called *high frequency pattern algorithm* (**HFPA**) to solve the problem. In HFPA, function **get_index_frequency** is used to calculate the table **IT**. IT is defined as follows:

IT[i] = γ_{i}, where γ_i denotes the number of patterns which are mapped to index i. These patterns are located at distinct positions in a DNA sequence with distinct or same sequences.

Let i to be a pattern Str's index, Str is a *candidate high frequency pattern* (**CHFP**) if and only if IT[i] >= κ, where κ denotes the frequency threshold stated in the high frequency pattern problem.

Function **get_candidate_patterns** is used to obtain the set of candidate high frequency patterns. String hash table **ST** is calculated by get_candidate_patterns. ST is indexed by patterns and is defined as follows:

ST[Str] = f_{Str} where Str is a distinct pattern in a DNA and f_{Str} denotes the frequency of pattern Str.

Function **get_HFPL** is used to obtain the sorted list of high frequency patterns with frequency threshold κ.

The pseudo code of HFPA is discussed below. In HFPA, the input parameters are a string DNA, a positive frequency threshold κ and a positive constant (length) ρ. HFPA returns the high frequency patterns list HFPL. P is the set of all candidate high frequency patterns. The high frequency pattern algorithm is given below:

```
Function HFPA(DNA, $\kappa$, $\rho$)
/* the inputs are: string DNA, frequency threshold $\kappa$ and length constant $\rho$*/

    P $\leftarrow \Phi$;      /* the set of candidate high frequency patterns*/

    IT $\leftarrow$ get_index_frequency(DNA, $\kappa$, $\rho$);

    ST $\leftarrow$ get_candidate_patterns(DNA, IT, P, $\kappa$, $\rho$);
    /*P is updated in this function*/

    HFPL <- get_HFPL(P, ST, $\kappa$);

return HFPL;

end;


Function get_index_frequency (DNA, $\kappa$, $\rho$)

    i $\leftarrow$ 1; Create table IT;

    loop while ( i <= |DNA| $-$ $\rho$ + 1)

        Str $\leftarrow$ DNA[i, i + $\rho$ $-$1];

        index $\leftarrow$ HF(Str);

        IT[index] $\leftarrow$ IT[index] + 1 ;      /* number of patterns mapped to this index */

        i $\leftarrow$ i +1;

    end loop

return IT;

end;
```

```
Function get_candidate_patterns(DNA, IT, P, $\kappa$, $\rho$)
/* the set of candidate high frequency patterns, P, is updated in this function */

  i $\leftarrow$ 1;     Create string hash table ST;        /*ST is indexed by patterns*/

   loop while ( i<= |DNA| – $\rho$ + 1)

     Str $\leftarrow$ DNA[i, i +$\rho$ –1];

     index $\leftarrow$ HF(Str);

     $\gamma_{index}$ = IT[index];        /*the number of patterns mapped to this index*/

     if  $\gamma_{index}$ >= $\kappa$  then        /*pattern is a candidate high frequency pattern */

          if   Str $\notin$ P  then

               add Str to P;

          end if;

          ST[Str] $\leftarrow$ ST[Str] +1 ;     /*calculate pattern Str's frequency $f_{str}$*/

     else;                         /*pattern is not a candidate high frequency pattern */

     end if;

     i $\leftarrow$ i+1;

   end loop;

return ST;

end;
```

```
Function get_HFPL (P, ST, κ)
    i ← 1; HFPL ← Φ;
    for each pattern Str ∈ P do
        if ST[Str] >= κ then            /*Str's number of distinct occurrence >=κ*/
            add Str to HFPL;
        else;
        end if;
    end do;
    sort HFPL in descending order by patterns' frequencies;
return HFPL;
end;
```

4.3 Redundancy

Before we introduce the concept of redundancy, we first introduce several relationships between two patterns.

String sStr is a *maximal overlapping string* (**MOS**) of two distinct patterns Str1 and Str2 (of same length) if and only if:

Str1 = sStr || s1 and Str2 = s2 || sStr

or Str1 = s1 || sStr and Str2 = sStr || s2

where the length of sStr is maximal among all such strings satisfying the relation presented above. s1 and s2 are strings of length >0.

Two patterns Str1 and Str2 is a *shift* to each other if and only if the maximal overlapping string sStr of these two patterns has length |sStr| >0.

The *signed shift distance* $\mathbf{Sd}_{\mathbf{Str1, Str2}}$ of a pattern Str1 and its shift Str2 is defined as:

$$\mathrm{Sd}_{\mathrm{Str1, Str2}} = \lambda \times (|\mathrm{Str1}| - |\mathrm{sStr}|)$$

Where sStr is the maximal overlapping string of Str1 and Str2.

λ is 1 if sStr is Str1's prefix and otherwise –1.

One example of string shift is shown in figure 7. Pattern Str1 = "ATTTTTCTGGG GACTCCT" is a shift of pattern Str2 = "TTTTCTGGGGACTCCTGG" with signed shift distance $\mathrm{Sd}_{\mathrm{Str1, Str2}} = -2$ and Str2 is a shift of Str1 with signed shift distance $\mathrm{Sd}_{\mathrm{Str2, Str1}} = 2$.

We define *shift distance*, **Sd**, of two patterns Str1 and Str2 to be $|\mathrm{Sd}_{\mathrm{Str1, Str2}}|$. For two patterns that do not have a shift relationship the Sd is defined to be infinity.

```
AT|TTTTCTGGGGACTCCT|
  |TTTTCTGGGGACTCCT|GG
```

Figure 7. An example of pattern shift patterns "AT TTTTCTGGGGACTCCT" and "TTTTCTGGGGACTCCT GG" have a shift relationship and shift distance Sd = 2

One pattern Str1 *overlaps* Str2 if and only if at least one occurrence of Str1 overlaps an occurrence of Str2. One example of pattern overlapping is shown in figure 8. Pattern "ATTGCT" overlaps "TTGCTA" because its first occurrence overlaps the first occurrence of "TTGCTA".

Pattern Str1 is Str2's *partner* if and only if every occurrence of Str1 overlaps with an occurrence of Str2.

Patterns Str1 and Str2 are *couple* if and only if Str1 is Str2's partner and Str2 is Str1's partner.

Pattern Str1 is a *ghost* of Str2 if and only if Str1 is Str2's partner while Str2 is not Str1's partner. A ghost is defined to be a *redundancy*.

"GA **TTGCTA** TACGCAG **TTGCTA** GGGCGACT **TTGCTA** GTACGAC **TTGC TA** CCCAGTCCT **TCAGGC** TTCGATCA **TCAGGC** GGGCTTACA **TTGCTA**".

Figure 8. An example of pattern overlapping

The function **RR** (defined below) is used to remove the redundancies from the sorted high frequency pattern list HFPL.

```
Function RR (HFPL)
  i ← 1;
  loop while (i<= Size(HFPL) )
      Str1 ← HFPL[i];
      remove Str2 from the HFPL where Str2 is Str1's ghost;
      i ← i + 1;
  end loop;
return HFPL;
end;
```

Based on the algorithm described above, a clustering algorithm is developed in this thesis. The algorithm is called Localized Clustering Algorithm (LCA). The design of LCA is presented in the next chapter.

CHAPTER V

THE LOCALIZED CLUSTERING ALGORITHM DESIGN

5.1 Introduction

In the localized clustering algorithm, each object in the objects-set $\Omega = \{1, 2, \ldots, n\}$ is described by two variables v and f.

v is a $1 \times p$ matrix $v \equiv (M_k)$, where M_k is the value taken by the kth variable describing this object ($k = 1, 2, \ldots, p$). We call this variable v as the related object's *value*. We use $\mathbf{v_i}$ to denote the value of object i in Ω ($i=1, 2, \ldots, n$).

Another variable f is a positive integer. We call the variable f as the related object's *frequen*cy. We use $\mathbf{f_i}$ to denote the frequency of the object i in Ω ($i=1, 2,\ldots, n$).

Correspondingly, each cluster in the localized clustering algorithm is associated with two variables V and F.

V is a $1 \times p$ matrix $V = (N_k)$, where N_k is the value taken by the kth variable describing this cluster ($k = 1, 2, \ldots, p$). We call the variable V as the related cluster's *value* and use $\mathbf{V_Y}$ to denote the value of cluster Y.

If Y is a singleton cluster {i}, Y's value is the same as the value of the object i in Ω. For clusters which contain more than one object, the definition of the value can be found in section 5.2 along with the discussion of LCA.

Another variable F is a positive integer and is called as the cluster's *frequency*. We use F_Y to denote cluster Y's frequency. F_Y is defined to be the sum of the frequencies of objects that belong to Y. That is:

$$F_Y = \sum f_i \quad \text{for every object } i \in Y$$

The distance between two clusters is defined to be the distance between the two clusters' values. That is:

$df(Y, Z) = d(V_Y, V_Z)$ where Y and Z denote two clusters. V_Y and V_Z denote the values of Y and Z, respectively.

Let S be an ordered set of clusters, the *weight* of a cluster Y, $\mathbf{W_Y}$, with given distance variable **Dis** is defined as:

$$W_Y = \sum_Z \delta_{YZ} \times F_Z \text{ where } Z \in S$$

$$\text{and } \delta_{YZ} = 1 \text{ if } df(Y, Z) <= Dis$$
$$= 0 \text{ otherwise}$$

The LCA is described in the next section. The algorithm description is at the abstract level without reference to any specific distance function.

5.2 Localized Clustering Algorithm

In the description of the localized clustering algorithm, *Dis* denotes a distance variable. *MinD* denotes the minimal distance found in the objects-set Ω. *DisInc* is a positive number denoting the distance increment. T represents a set of clusters representing the n-tree obtained from Ω by LCA. T_1 and T_2 represent ordered sets of clusters.

The localized clustering algorithm is a six-step algorithm described below:

```
1. Dis ← MinD; $T = T_1 = \{ \{1\}, \{2\}, \ldots, \{n\} \}$ where $\Omega = \{1, 2, \ldots, n\}$;

2. For each cluster $Y \in T_1$, calculate cluster Y's weight $W_Y$:

   $W_Y = \sum_X F_X$  where $X \in T_1$ and $df(Y, X) <= Dis$;

3. Sort $T_1$ in descending order of weights; initialize ordered set of clusters $T_2$ to $\phi$;

4. Let cluster Z be the first element of $T_1$;   $CL \leftarrow \phi$;

   for all clusters Q in $T_1$ do

      if $df(Z, Q) <= Dis$ then

            $T_1 \leftarrow T_1 - \{Q\}$;   $CL \leftarrow CL \cup Q$;

      end if;

   end do;

   $V_{CL} \leftarrow V_Z$; $T_2 \leftarrow T_2 \cup \{CL\}$;

5. If $T_1$ is empty, replace $T_1$ by $T_2$ and go to step 6, otherwise go to step 4;

6. $T \leftarrow T \cup T_1$; stop if $T_1$ contains only one element, otherwise Dis ← Dis + DisInc and
   go to step 2.
```

The algorithm can be illustrated by the following example in which the algorithm is applied to the distribution shown in figure 9. Intermediate results are shown in figure 10 and among the distances between the objects, the minimal positive distance is 1. Therefore the variable Dis has an initial value 1 each. In this example, the DisInc is 1. Initializing T and T_1, we get $T = T_1 = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \{9\} \}$.

There are three iterations in the example. The first iteration's result is shown in figure 10. For the first iteration, after calculating all clusters' weights, the weight of the cluster CL_1 ($\{1\}$) is $W_{CL1} = 9$. For simplicity, we refer to the cluster CL_1 as "cluster 1". Therefore, the above fact can be expressed as cluster 1 has weight $W_1 = 9$. We trace the updating process as follows:

The distance between cluster 1 and itself is 0 (by distance definition). The distances between cluster 1 and clusters 2, 3, 4 and 5 are all equal to 1 each. Therefore cluster 1's frequency 1, cluster 2's frequency 2, cluster 3's frequency 1, cluster 4's frequency 3 and cluster 5' frequency 2 are added up. The result 9 is assigned as cluster 1's weight.

Similar calculations are performed to the rest of the clusters in T_1 and their weights are listed below: $W_2 = 3$, $W_3 = 2$, $W_4 = 4$, $W_5 = 3$, $W_6 =4$, $W_7 =3$, $W_8 = 3$ and $W_9 = 1$.

After sorting elements in T_1, we get the order: 1(9), 4(4), 6(4), 2(3), 5(3), 7(3), 8(3), 3(2) and 9(1). The numbers before the parentheses indicate the clusters and those within parentheses are the calculated weights. Considering cluster 1, the first component in T_1, the cluster A $\{1, 2, 3, 4, 5\}$ is formed by joining clusters 2, 3, 4 and 5 to cluster 1. A's value is the same as the value of cluster 1. It is the vector (3, 2). A's frequency is 9. After this operation, T_1 is updated from $\{\{1\}, \{4\}, \{6\}, \{2\}, \{5\}, \{7\}, \{8\}, \{3\}, \{9\}\}$ (after sort) to $\{\{6\}, \{7\}, \{8\}, \{9\}\}\}$ and $T_2 = \{A\}$. Cluster B is formed by joining clusters 6, 7 and 8 together. B's value is the same as that of cluster 6 (the vector (6, 2)) and B's frequency, f_B, is 4. After this operation, T_1 is {{9}} and T_2 is {A, B}. Finally, {9} is moved from T_1 to T_2. Therefore, T_1 is empty and T_2 is {A, B, {9}}. Control is transferred to step 6 in the procedure we presented above.

At the end of the first iteration, T = {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {1, 2, 3, 4, 5}, {6, 7, 8}} and T_1 (replaced by T_2) is {{1, 2, 3, 4, 5}, {6, 7, 8}, {9}}. The result of the first iteration is shown in figure 10.

For the second iteration, the Dis's value is 2. No variable changes because every possible distance in figure 10 is greater than 2.

At the beginning of the third iteration, T_1 contains three components: clusters A ({1, 2, 3, 4, 5}), B ({6, 7, 8}) and {9}. The frequencies of the clusters are indicated within the parentheses in figure 10. The weight of A is, using the similar calculation as shown in the first iteration, 9 + 4 + 1 = 14. The weight of B is 9 + 4 = 13 and that of cluster {9} is 9 + 1 = 10. Cluster C is formed by joining A, B and {9}. This cluster has value (3, 2) and frequency 14.
At the end of third iteration, T = {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {1, 2, 3, 4, 5}, {6, 7, 8}, {1, 2, 3, 4, 5, 6, 7, 8, 9}} and T_1 = {{1, 2, 3, 4, 5, 6, 7, 8, 9}}. Since T_1 only contains a single element now, the clustering process stops.

The n-tree of the objects-set using localized clustering algorithm is shown in figure 12.

An outline of an implementation of LCA is given in the next chapter.

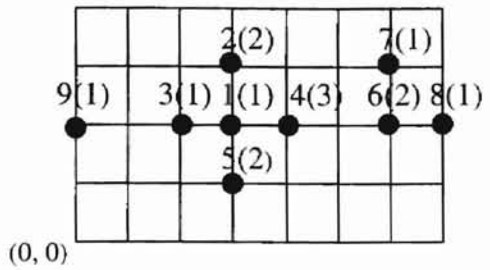


Figure 9. Distribution of 9 objects

Frequencies are: $f_1 = 1$, $f_2 = 2$, $f_3 = 1$, $f_4 = 3$,

$f_5 = 2$, $f_6 = 2$, $f_7 = 1$, $f_8 = 1$ and $f_9 = 1$

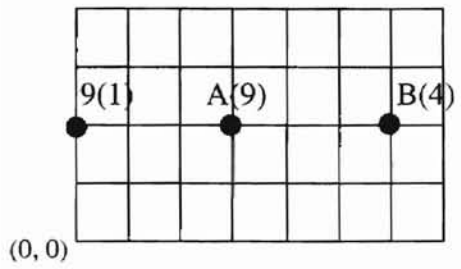


Figure 10. The result of the first iteration of the LCA

Cluster A = {1, 2, 3, 4, 5} frequency $f_A = 9$

Cluster B = {6, 7, 8} frequency $f_B = 4$

T = {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8},

{9}, {1, 2, 3, 4, 5}, {6, 7, 8}}.

T_1 = {{1, 2, 3, 4, 5}, {6, 7, 8}, {9}}.

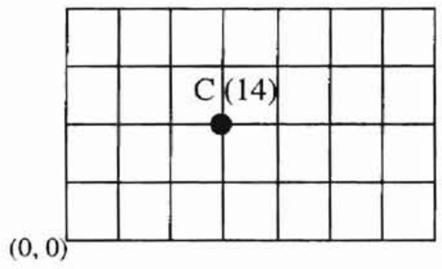


Figure 11. The result of the third iteration of the LCA

C = {1, 2, 3, 4, 5, 6, 7, 8, 9}

T = {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8},

{9}, {1, 2, 3, 4, 5}, {6, 7, 8},

{1, 2, 3, 4, 5, 6, 7, 8, 9}}

T_1 = {{1, 2, 3, 4, 5, 6, 7, 8, 9}}

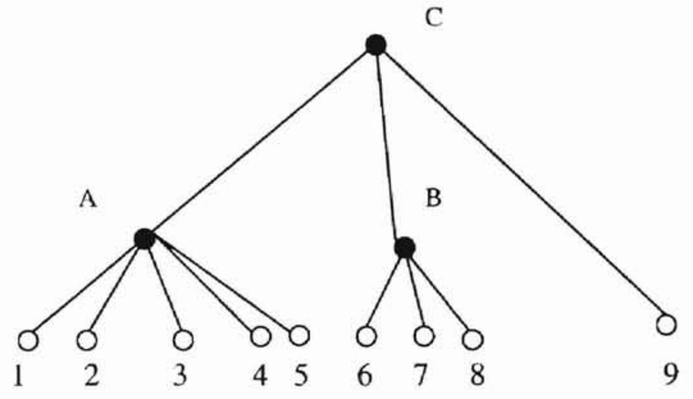


Figure 12. The n-tree constructed by LCA

CHAPTER VI

ALGORITHM IMPLEMENTATION AND ANALYSIS

6.1 Introduction

In this chapter, we first discuss the implementation of the localized clustering algorithm. Theoretical analysis of the time complexity and space complexity is performed. The experimental results using the human genome sequence verifies the result of the theoretical analysis of time complexity.

6.2 The Localized Clustering Algorithm Implementation

In this section, we give an implementation of the localized clustering algorithm. In the description of the algorithm, let L denote an ordered set, then size(L) denotes the cardinality of L and L[i] denotes the ith element of L (i= 1, 2, .., size(L)).

The localized clustering algorithm receives an input objects-set Ω. It returns the n-tree T from Ω. Sets T_1 and T_2 are ordered sets of clusters. Distance variable Dis, minimal distance MinD in Ω, the distance increment DisInc, object i's value v_i and frequency f_i are defined in section 5.1 and 5.2. Unless stated otherwise, we assume they have the same meaning as defined in section 5.1 and 5.2.

Each cluster in the LCA is implemented as a structure with four fields. The first three fields in the structure are the value of the clusters, the frequency of the cluster and the weight of the cluster. These variables are defined in section 5.1 and we use Y.v, Y.f and Y.w to denote a cluster Y's value, frequency and weight, respectively. The fourth field is the set of objects that belong to Y. We use Y.s to denote the set of objects that belong to Y.

The six-step procedure LCA is given in pseudo code form by the functions LCA. Initialize_set, Weight, Sort and Absorb. They are shown below:

```
Function LCA(Ω)
        Dis ← MinD;    /*MinD is calculated from Ω*/
        T ← $T_1$ ← Initialize_set(Ω);
        Loop while (size($T_1$)>1 )
                $T_1$ ← Weight(Dis, $T_1$);
                Sort($T_1$);
                $T_1$ ← Absorb(Dis, $T_1$);
                T ← $T \cup T_1$; Dis ← Dis + DisInc;
        end loop;
return T;
end;
```

The function Initialize_set, Weight, Sort and Absorb are described as follows:

```
Function Initialize_set ($\Omega$)
        P $\leftarrow \phi$;
        for each i $\in \Omega$ do
                create new cluster CL;
                CL.v $\leftarrow v_i$;   CL.f $\leftarrow f_i$;
                CL.w $\leftarrow$ 0;   CL.s $\leftarrow$ {i};
                Add CL to P;
        end do;
return P;
end;

Function Weight(Dis, $T_1$)
/* weights of clusters in $T_1$ are calculated in this function*/
        i $\leftarrow$ 1;
        loop while ( i < = size($T_1$) )
                CL$\leftarrow$ $T_1$[i];
                CL.w $\leftarrow$ CL.w + Q.f for every Q $\in T_1$ and df(CL, Q) <= Dis;
                i $\leftarrow$ i+1;
        end loop
return $T_1$;
end;
```

```
Function Sort(S)
        sort the set S of clusters in descending order by elements' weights using any
        efficient sorting algorithms;
end;

Function Absorb (Dis, $T_1$)
        $T_2 \leftarrow \phi$;
        loop while (size($T_1$) >0)
            cluster Z $\leftarrow T_1[1]$;   create new cluster CL;
            CL.v $\leftarrow$ Z.v;   CL.f $\leftarrow$ 0;                    /*initialization*/
            CL.w $\leftarrow$ 0;      CL.s $\leftarrow \phi$;
            for all clusters Q in $T_1$ do
                if df(Z, Q) <= Dis then
                        $T_1 \leftarrow T_1 - \{Q\}$;   CL.s $\leftarrow$ CL.s $\cup$ Q.s;
                end if;
            end do;
            CL.f $\leftarrow \Sigma f_i$, for every i $\in$ CL    /*calculate cluster CL's frequency*/
            $T_2 \leftarrow T_2 \cup \{CL\}$;
        end loop
return $T_2$;
end;
```

The function LCA was implemented as a Perl script and the source code is presented in appendix A. The functions Initialize_set, Weight and Absorb were implemented in C programming language and the source codes are presented in appendix B. The function Sort was implemented by UNIX command sort.

6.3 The Theoretical Analysis of the Localized Clustering Algorithm

We compare the localized clustering algorithm to the hierarchical clustering algorithms presented in [4] and [24].

Let Ω to be the objects-set with n objects, the hierarchical clustering algorithm must calculate and maintain an $n \times n$ two-dimensional distance matrix ([4], [24]). In the localized clustering algorithm, each cluster is implemented by a structure with four fields: the cluster's value, the cluster's frequency, the cluster's weight and the set of objects that belong to this cluster. Using this implementation, considering a set of n clusters, P, the total space required by P is:

$3 \times n + \sum_{Y} |Y.s|$ where $Y \in P$ and $|Y.s|$ denotes the number of objects belonging to Y

Because there are n objects in Ω, the sum of all clusters' size (number of objects belonging to the cluster) is n. Therefore, the total size required by P is $O(3n + n) = O(4n)$. This memory-saving property is a great advantage of our algorithm.

Most widely-used hierarchical clustering algorithms can be implemented in the Lance-Williams-Jambu general recurrence relation, which would have $O(n^3)$ time complexity and $O(n^2)$ space complexity ([11], [12]). However, there are some algorithms that do not fit within this framework, for example those using criteria based on information content

([11]). Further, the authors in [3] pointed out that the time complexity can be reduced to $O(n^2 \log n)$ for special cases.

There are at most n iterations in the Weight function in the localized clustering algorithm. Within one such iteration, there are at most n operations. Therefore, the Weight function takes time $O(n \times n) = O(n^2)$. The Sort function, considering a heap implementation, could take time $O(n \log n)$. In the Absorb function, there are at most n iterations and within one iteration, there are at most n operations. Therefore, Absorb takes another $O(n^2)$ time. Therefore one loop iteration of the localized clustering algorithm takes time $O(n^2)$. There are k iterations and the total execution time complexity is $k \times O(n^2) = O(kn^2)$. The variable k's value depends on the application. However, in most cases k can be assumed to be a constant. In our substring processing application, let MinD, (the minimal distance found in the objects-set Ω) and DisInc (the distance increment) both have value 1. Then the worst case of the iteration number, k, is ρ -1, where ρ is the length constant of the pattern. To prove this conclusion, we notice that the maximal possible distance between two substrings is ρ. In the localized clustering algorithm, when the distance variable, Dis, has the value equal to the maximal distance found in Ω, all clusters will be joined to form one cluster. The LCA will stop after that. To finish our proof, we notice that after ρ-1 iterations, the distance variable Dis has the value ρ. Therefore in our substring processing application, we have the time complexity $O(\rho n^2)$.

6.4 The Experimental Analysis of the Localized Clustering Algorithm

The High Frequency Substring Algorithm was applied to human genome sequence (downloadable from web site: ftp://ftp.ebi.ac.uk/pub/databases/embl/release/). A list of

126K patterns was obtained with frequency threshold $\kappa = 50$. After removing the redundancy, a list of 102K patterns was obtained. This pattern list is used to test our localized clustering algorithm.

The execution results are listed in table 1. The objects-set's size is the number of patterns used. The execution time was collected using UNIX command "time" and then transformed to seconds. For comparison, one loop iteration is performed for each objects-set.

Table1. Relationship of execution time and objects-set size

Objects-set Size	Execution Time(S)
1019667	9475
57013	2785
35513	1409
31103	1012
24805	565
14425	172
5507	24
1566	2

The graph presented in figure 13 compares the execution time with the function n^2. It verifies that the performance of the localized clustering algorithm is $O(n^2)$.

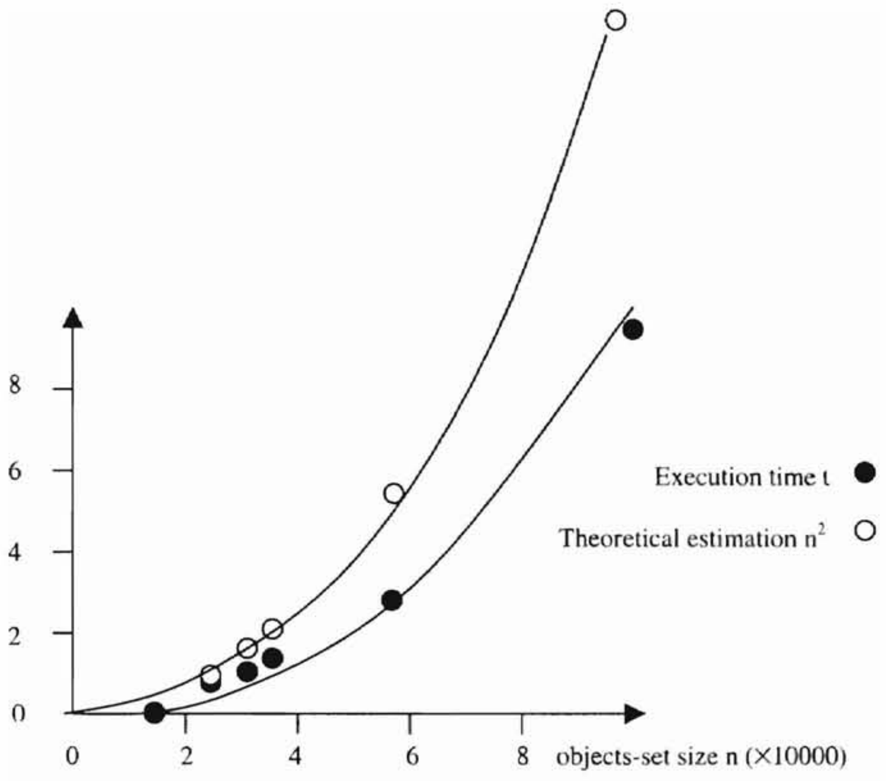


Figure 13. Comparison of execution time and function n^2

CHAPTER VII

SUMMARY AND FUTURE WORK

In this thesis, a time and space efficient clustering algorithm for DNA string clustering is designed and implemented. This algorithm is named localized clustering algorithm. The localized clustering algorithm runs on space complexity O(n) and time complexity $O(kn^2)$ (k's value dependents on applications and in most cases k can be assumed to be a constant).

Currently, the distance between two substrings is defined as the hamming distance. Biologists have established that a DNA can change its sequence by insertion (adding a DNA sequence to an existing DNA sequence), deletion (removing a piece of DNA sequence from existing DNA sequence), or reversing (changing the direction of a piece of DNA sequences) ([2]). Modification of the distance function to reflect these changes and evaluation of the new distance function are proposed as future work.

REFERENCES

[1] H.M. Bobisud & L.E. Bobisud, "A Metric for Classification," *Taxon*, Vol. 21, 1972, pp. 607-613.

[2] M.K. Campell, *Biochemistry (2nd edition)*, Saunders College Publishing, Orlando, Florida, 1995.

[3] W.H. Day & H. Edelsbrunner, "Efficient Algorithms for Agglomerative Hierarchical Clustering Methods," *Journal of Classification*, Vol. 1, 1984, pp. 7-24.

[4] A.D. Gordon, *Clustering and Classification*, World Scientific Publ., Singapore, 1996.

[5] J.C. Gower, "A Comparison of Some Methods of Cluster Analysis," *Biometrics*, Vol. 23, 1967, pp. 623-638.

[6] M.C. Harrison, "Implementation of the Substring test by Hashing," *Communications of the ACM*, Vol. 14, No.1, Dec. 1971, pp. 777-779.

[7] S. C. Johnson, "Hierarchical Clustering Schemes," *Psyhometrika*, Vol. 32, 1967, pp. 241-254.

[8] R.M. Karp & M.O. Rabin, "Efficient Randomized Pattern-matching Algorithms," *IBM journal of Research and Development*, Vol. 31, No. 2, March, 1987, pp. 249-260.

[9] G.N. Lance & W.T. Williams, "A General Theory of Classificatory Sorting Strategies I. Hierarchical Systems," *Computer Journal*, Vol. 9, Feb. 1967, pp. 373 – 380.

[10] G.N. Lance & W. T. Williams, "Note on the Classification of Multi-level Data," *Computer Journal*, Vol. 9, Feb. 1967, pp. 380-380.

[11] G.N. Lance & W. T. Williams, "A General Sorting Strategy for Computer Classifications," *Nature*, Vol. 212, Oct. 1966, pp. 218-218.

[12] G.N. Lance & W. T. Williams, "Computer programs for Hierarchical Polythetic Classification (Similarity Analyses)," *Computer Journal*, Vol. 9, May 1966, pp. 60 – 64.

[13] F.J. Lapointe & P. Legendre, "The Generation of Random Ultrametric Matrices Representing Dendrograms," *Journal of Classification*, Vol. 8, 1991, pp.177-200.

[14] F. R. McMorris, D. B. Mronk & D.A. Neumann, *Numerical taxonomy*, Springer-Verlag, Berlin, Germany, 1983.

[15] L.L. McQuitty, "Similarity Analysis by Reciprocal Pairs for Discrete and Continuous Data," *Educational and Psychological Measurement*, Vol. 26, 1966, pp. 825 – 831.

[16] L.L. McQuitty, "Hierarcial Linkage Analysis for the Isolation of Types," *Educational and Psychological Measurement*, Vol. 20, 1960, pp. 55- 67.

[17] R. Overbeek & J. Huan, "A High Frequency Substring Algorithm," unpublished, 1999.

[18] N. Shubin, C. Tabin & S. Carroll, "Fossils, Genes and the Evolution of Animal Limbs," *Nature*, Vol. 388 (6643), Aug. 1997, pp. 639-648.

[19] P.H.A. Sneath, "The Application of Computers in Taxonomy," *J. Gen. Microbiol.*, Vol. 17, 1957, pp. 201 - 226.

[20] R.R. Sokal & C.D. Michener, "A Statistical Method for Evaluating Systematic Relationships," *University of Kansas Science Bulletin*, Vol. 38, 1958, pp. 1409 – 1438.

[21] G.A. Stephen, *String Searching Algorithms*, World Scientific, Singapore, 1994.

[22] H. Turner & J.P. Kinet, "Signaling Through the High-affinity IgE Receptor Fc EpsilonRI," *Nature*, Vol. 402 (6760 Suppl B), Nov. 1999, pp. 24-30.

[23] J. H. Ward, "Hierarchical Grouping to Optimize an Objective Function," *J. Am. Statist. Assoc.*, Vol. 58, 1963, pp.236-244.

[24] J. Zupan, *Clustering of Large Data Sets*, Research Studies Press, Herts, England, 1982.

APPENDIX A — PERL SCRIPT OF THE LOCALIZED CLUSTERING ALGORITHM

```
#!/usr/local/bin/perl5 -w
( $]>=5.004) || die "version is $] -- need perl 5.004 or greater";

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# This software is being made freely available without conditions on its
# use or distribution. The software was developed by cooperation of
# Oklahoma State University and Mathematics and Computer Science Division
# of Argonne National Laboratory. No liability is assumed for any flaws
# in the software or for failure of the software to work as expected.
# If you find this software useful in your research, we would appreciate
# your adding the following acknowledgment:
#
#     ''We gratefully acknowledge use of the Localized Clustering
#      software developed by researchers in Oklahoma State University
#      and the Mathematics and Computer Science Division of Argonne
#      National Laboratory. "
#
#-----------------------------------------------------------------------

#check the command line parameter
$file = shift @ARGV;
open ( FILE_IN,  "<$file") || die "Usage: cluster.pl <input file> <begin distance> <end
distance> <distance increase> \n";
close(FILE_IN);

#set default value
$being_dis = 2;
$end_dis = 2;
$dis_inc =1;

$wei_command ="time wei ";
$abs_command ="time abs ";
$round_number =0;
```

```
#clean data directory clusters
if( opendir( "DIR", "clusters") )
{
    close(DIR);
    system("rm -r clusters");
}

system( "mkdir clusters");

#run the clean, weight and absorb
$clean_command = "clean ".$file." clusters/cluster_dis0";
print("$clean_command");
system("$clean_command");

#run weight and absorb using loop
for( $tar_dis = $being_dis; $tar_dis <=$end_dis; $tar_dis += $dis_inc)
{
   if( $tar_dis == $being_dis){ $tar_dis1 = 0 ; }
   else{   $tar_dis1 = $tar_dis - $dis_inc; }

   $wei_command = $wei_command." clusters/cluster_dis".$tar_dis1." ".$tar_dis;
   $abs_command = $abs_command." clusters/cluster_dis".$tar_dis." ".$tar_dis;

   #execute the command
   print("\nwei operation\n");
   print("$wei_command\n");
   system("$wei_command");

   #sort by unix command
   system("rm .cluster_temp");
   system("sort -r -n +1 .cluster_tempo > .cluster_temp");
   system("wc -l .cluster_temp");

   #do the abs command
   print("abs operation\n");
   print("$abs_command\n");
   system("$abs_command");

   $wei_command ="time wei ";
   $abs_command ="time abs ";
   $round_number++;

}
```

APPENDIX B — IMPLEMENTATION OF FUNCTIONS INITIALIZE_SET, WEIGHT AND ABSORB

Weight Function:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

/*define macros*/
#define UNIT_ENTRY_NUM 13        /*how many memory block*/
#define MEMORY_UNIT_SIZE 10000    /*each block's size*/
#define STR_LENGTH 18  /*string length*/
#define CLEAN_CONSTANT 1.1  /*clean constant*/
#define SHIFT_DIS 3  /*the shift distance*/
#define TRUE 1  /*boolean constant*/
#define FALSE 0

/*data definition*/
typedef struct data_def {
 int frequency;
 int weight;
 char str[STR_LENGTH+1];
 char flag;
} Data;

/*Global Variable*/
Data * MEM_BLOCK[UNIT_ENTRY_NUM];/*store the memory block's location*/
int TOTAL_NUM;/*total data number*/

/*functions*/
void block_ini(void);/*INITIALIZE THE SET*/
Data * memory_allocate(void);/*allocate the memory for block*/
int memory_deallocate(void);  /*deallocate the memory for block*/
Data * get_data(int data_num);    /*interface of the memory management and data*/
void read_data(char * file_name);/*read data from file*/
void write_data(char * file_name);/*wrote cleaned data to file*/
int find_distance(Data * data1_pt, Data * data2_pt); /*find the distance of two data*/
void weight(int);  /*perform the weight operation */
void main(int argc, char * argv[])
{

 char* input_file = argv[1];
 char output_file[] = ".cluster_tempo";
 int dis_p;
 FILE * system = fopen("log_file","a");

 if( argc <3 ){
    fprintf(system,"Usage: wei <distance parameter>\n");
    printf("Usage: wei <distance parameter>\n");
 }
 else{

  printf("Wei begin\n");
  sscanf(argv[2], "%d", &dis_p);
  read_data(input_file);
  weight(dis_p);

  write_data(output_file);
  memory_deallocate();
 }

 fclose(system);

 printf("wei finished\n");

}

/*
  Block initialize will initialize the block entry
  */
void block_ini(void)
{
  int i;
  for( i=0; i< UNIT_ENTRY_NUM; i++){
      MEM_BLOCK[i] = NULL;
  }

}


/*
  Allocate the memory for MEMORY_UNIT_SIZE's data
  and return the point to the calling process
  */
Data * memory_allocate(void)
{
  Data * pt = NULL;
  pt = malloc( MEMORY_UNIT_SIZE * sizeof(Data) );


  if( !pt ) { printf("Out of Memory!"); return NULL; }
  else return pt;

}

/*
  free memory and let the os know the free event

  */
int memory_deallocate(void)
{

 int i;
 Data * curr_pt;

 /*free the block */
 for( i=0; i<UNIT_ENTRY_NUM; i++){

   if( (curr_pt = MEM_BLOCK[i] ) != NULL){
     free( curr_pt );
   }

 }

 return EXIT_SUCCESS;

}


/*
   Get data will get the data from the memory
   */
Data * get_data(int data_num)
{
   Data * data_pt = NULL;
   Data * block_pt;


   /*out of array boundary*/
  if( data_num < 0 || ( (data_num / MEMORY_UNIT_SIZE) > UNIT_ENTRY_NUM) )
    return NULL;

  block_pt = MEM_BLOCK[data_num /MEMORY_UNIT_SIZE];


  /*get the block base address */
  if(block_pt != NULL )
    data_pt = &(block_pt[data_num % MEMORY_UNIT_SIZE]);

  return data_pt;

}

/*
  read the data from the input file and put the
  frequency, weight, the string, the flag into the
  structure
  input parameter: input file name
  */
void read_data(char * file_name)
{
 FILE * in_file = fopen(file_name, "r");
 int line_num =0;
 char line_buff[81] ="\n" ;
 char string[STR_LENGTH] ="\n";
 int frequency =0;
 Data * curr_m_pt;
 int curr_index;
 FILE * system = fopen("log_file", "a");

 /*inilizing block array*/
 block_ini();


 /*allocate the first block*/
 curr_m_pt =memory_allocate();
 if( curr_m_pt == NULL ) exit( 0 );    /*out of memory*/



 fprintf(system, "Begin to load data\n");
 fprintf(system, ".");

 /*return*/;
 MEM_BLOCK[0] = curr_m_pt;
 /*read data into memory*/
 while( fgets(line_buff, 80, in_file)  != NULL )
  {
   line_num++;


   /*printf("%d-- %s", line_num, line_buff);*/

   /*read the string and frequency*/
   sscanf(line_buff, "%s%d", string, &frequency);

   curr_index  = (line_num -1) % MEMORY_UNIT_SIZE;
   curr_m_pt[curr_index].frequency = frequency;
   curr_m_pt[curr_index].weight = frequency;
   strncpy(curr_m_pt[curr_index].str, string, STR_LENGTH);
   (curr_m_pt[curr_index].str)[STR_LENGTH] = '\0';
   curr_m_pt[curr_index].flag = 't';


   /*if the line size come to block size, then
         allocate the memeory again*/
   if( line_num % MEMORY_UNIT_SIZE == 0)
        {
         curr_m_pt = memory_allocate();
         if( curr_m_pt == NULL ) exit(0);
         MEM_BLOCK[line_num /MEMORY_UNIT_SIZE] = curr_m_pt;
         fprintf(system, ".");
        }

  }

 TOTAL_NUM = line_num;
 fprintf(system, "\nFinished loading\n");
 close(system);
}

/*
  If two objecs distance is within certain distance, then
  update their weight
*/
void weight(int target_dis)
{
 int line_num, line_num2;
 Data * target_pt, *curr_data_pt;
 int ham_dis = 0;
 int freq1, freq2;
 FILE * system  = fopen("log_file", "a");

 fprintf(system, "Begin to do weight: %d\n", TOTAL_NUM);

 for( line_num =0; line_num < TOTAL_NUM; line_num++)
   {
    target_pt = get_data(line_num);
    freq1 = target_pt->frequency;

    /*report      */
    if(( line_num +1)  % 1000 == 0){
          fprintf(system, ".");
               if(( line_num+1) % 10000 == 0){
           fprintf(system, "\n");
               }
    }

    /*compare the target and current data*/
    for( line_num2 =line_num+1; line_num2 < TOTAL_NUM; line_num2++)
      {
       curr_data_pt = get_data(line_num2);

       ham_dis = find_distance(target_pt, curr_data_pt);

       if( ham_dis <= target_dis){
        freq2 = curr_data_pt->frequency;
        target_pt->weight += freq2;
        curr_data_pt->weight += freq1;
       }
        }
   }

 /*after weight*/
 fprintf(system, "Finished weight\n");
}

/*
  Find the hamming distance for the two data object

*/
int find_distance(Data * data1_pt, Data * data2_pt)
{
  char * str1 = data1_pt->str;
  char * str2 = data2_pt->str;
  int haming_dis =0;
  int i;

  for(i=0;i<STR_LENGTH;i++){
    if( str1[i] != str2[i] ) haming_dis++;
  }

  return haming_dis;
}

/*
 write the data back to a file only if the flag is  t

 */

void write_data(char * file_name)
{
 FILE * file_out = fopen(file_name, "w");
 int i;
 Data * data_pt;

 if( file_out == NULL ){
   printf("Error in open file %s for writing\n");
   return;
 }


 for( i=0; i<TOTAL_NUM; i++)
   {
    data_pt = get_data(i);

    if(data_pt != NULL)
         fprintf(file_out, "%s %d %d\n", data_pt->str,
                              data_pt->weight, data_pt->frequency);
   }


}
```

Absorb Function:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
```

```
/*define macros*/
#define UNIT_ENTRY_NUM 13        /*how many memory block*/
#define MEMORY_UNIT_SIZE 10000   /*each block's size*/
#define STR_LENGTH 18  /*string length*/
#define CLEAN_CONSTANT 1.1  /*clean constant*/
#define SHIFT_DIS 3  /*the shift distance*/
#define TRUE 1  /*boolean constant*/
#define FALSE 0

/*data definition*/
typedef struct data_def {
 int frequency;
 int weight;
 char str[STR_LENGTH+1];
 char flag;
} Data;

/*Global Variable*/
Data * MEM_BLOCK[UNIT_ENTRY_NUM];/*store the memory block's location*/
int TOTAL_NUM;/*total data number*/

/*functions*/
void block_ini(void);/*inilize the block entry*/
Data * memory_allocate(void);/*allocate the memory for block*/
int memory_deallocate(void);  /*deallocate the memory for block*/
Data * get_data(int data_num);    /*interface of the memory management and data*/
void read_data(char * file_name);/*read data from file*/
void write_data(char * file_name);/*wrote cleaned data to file*/
int is_shift(Data * data1_pt, Data * data2_pt);/*whether the two data is shift*/
int find_distance(Data * data1_pt, Data * data2_pt); /*find the distance of two data*/
void absorb(int, char*);   /*perform the absorb operatoin */

void main(int argc, char * argv[])
{
 char input_file[] = ".cluster_temp";
 char *output_file;
 int distance_p;

 /*check error message*/
 if( argc < 3 ) { printf("abs <output file> <distance parameter>"); }
 else{
   sscanf(argv[2], "%d", &distance_p);
   output_file = argv[1];

   read_data(input_file);
   absorb(distance_p, output_file);
```

```
    write_data(output_file);
    memory_deallocate();
  }
}

/*
  Block initilize will initilize the block entry
  */
void block_ini(void)
{
  int i;
  for( i=0; i< UNIT_ENTRY_NUM; i++){
      MEM_BLOCK[i] = NULL;
  }

}


/*
  Allocate the memory for MEMORY_UNIT_SIZE's data
  and return the point to the calling process
  */
Data * memory_allocate(void)
{
  Data * pt = NULL;
  pt = malloc( MEMORY_UNIT_SIZE * sizeof(Data) );


  if( !pt ) { printf("Out of Memory!"); return NULL; }
  else return pt;

}

/*
  free memory and let the os know the free event

  */
int memory_deallocate(void)
{

 int i;
 Data * curr_pt;

 /*free the block */
 for( i=0; i<UNIT_ENTRY_NUM; i++){
    if( (curr_pt = MEM_BLOCK[i] ) != NULL){
      free( curr_pt );
    }

  }

  return EXIT_SUCCESS;

}


/*
   Get data will get the data from the memory
   */
Data * get_data(int data_num)
{
   Data * data_pt = NULL;
   Data * block_pt;


   /*out of array boundary*/
   if( data_num < 0 || ( (data_num / MEMORY_UNIT_SIZE) > UNIT_ENTRY_NUM)  )
     return NULL;

   block_pt = MEM_BLOCK[data_num /MEMORY_UNIT_SIZE];


   /*get the block base address */
   if(block_pt != NULL )
     data_pt = &(block_pt[data_num % MEMORY_UNIT_SIZE]);

   return data_pt;

}

/*
  read the data from the input file and put the
  frequency, weight, the string, the flag into the
  structure
  input parameter: input file name
  */
void read_data(char * file_name)
{
 FILE * in_file = fopen(file_name, "r");
 int line_num =0;
 char line_buff[81] ="\n" ;
char string[STR_LENGTH] ="\n";
int frequency =0, weight=0;
Data * curr_m_pt;
int curr_index;
FILE * system = fopen("log_file", "a");

/*inilizing block array*/
block_ini();


/*allocate the first block*/
curr_m_pt =memory_allocate();
if( curr_m_pt == NULL ) exit( 0 );     /*out of memory*/


fprintf(system, "Begin to load data\n");
fprintf(system, ".");

/*return*/;
MEM_BLOCK[0] = curr_m_pt;


/*read data into memory*/
while( fgets(line_buff, 80, in_file)  != NULL )
 {
  line_num++;


  /*printf("%d-- %s", line_num, line_buff);*/

  /*read the string and frequency*/
  sscanf(line_buff, "%s%d%d", string, &weight, &frequency);

  curr_index  = (line_num -1) % MEMORY_UNIT_SIZE;
  curr_m_pt[curr_index].frequency = frequency;
  curr_m_pt[curr_index].weight = frequency;
  strncpy(curr_m_pt[curr_index].str, string, STR_LENGTH);
  (curr_m_pt[curr_index].str)[STR_LENGTH] = '\0';
  curr_m_pt[curr_index].flag = 't';


  /*if the line size come to block size, then
        allocate the memeory again*/
  if( line_num % MEMORY_UNIT_SIZE == 0)
      {
       curr_m_pt = memory_allocate();
```

```
          if( curr_m_pt == NULL ) exit(0);
          MEM_BLOCK[line_num /MEMORY_UNIT_SIZE] = curr_m_pt;
          fprintf(system, ".");
        }
    }

  TOTAL_NUM = line_num;
  fprintf(system, "\nFinished loading.\n");
  close(system);
}

/*
   data 1 will abost data 2 only when both of them have not been absorted by
   other data and data 2 is within certain distance of data1. The data1'weight
   is higher than data2' weight
*/
void absorb(int target_dis, char* file_name)
{
  int line_num, line_num2;
  Data * target_pt, *curr_data_pt;
  int ham_dis = 0;
  int freq1, freq2;
  FILE * system;
  char lfile[30];
  FILE * cluster_log;

  system = fopen("log_file", "a");


  /*log file for cluster*/
  strcpy( lfile, file_name);
  strcat(lfile, "_log");

  cluster_log  = fopen(lfile,"w");
  if( cluster_log == NULL )
    {
      printf("Error in wtirting the log file\n");
      return;
    }

  printf("%s", lfile);

  fprintf(system, "Begin to do absorb: %d\n", TOTAL_NUM);

  for( line_num =0; line_num < TOTAL_NUM; line_num++)
    {
```

```
    target_pt = get_data(line_num);
    freq1 = target_pt->frequency;

    /*report      */
    if(( line_num +1)  % 1000 == 0){
           fprintf(system, ".");
           if(( line_num+1) % 10000 == 0){
            fprintf(system, "\n");
           }
    }



    /*compare the target and current data*/
    if( target_pt->flag == 't' ){    /*only the one has not been absorbed could absorb
others*/

        fprintf(cluster_log, "%s %d\n", target_pt->str, target_pt->frequency);
        for( line_num2 =line_num+1; line_num2 < TOTAL_NUM; line_num2++)
         {
          curr_data_pt = get_data(line_num2);

          ham_dis = find_distance(target_pt, curr_data_pt);

          if( curr_data_pt->flag == 't' &&  ham_dis <= target_dis){
                 curr_data_pt->flag = 'f';
                 target_pt->weight += curr_data_pt->frequency;
           fprintf(cluster_log, "\t\t%s %d\n", curr_data_pt->str, curr_data_pt-
>frequency);
          }

         }
    }
   }

  /*finished absorb*/
  fprintf(system, "Finished doing absort\n");
  fclose(system);
  fclose(cluster_log);

}


/*
  Find the hamming distance for the two data object
*/
```

```
int find_distance(Data * data1_pt, Data * data2_pt)
{
   char * str1 = data1_pt->str;
   char * str2 = data2_pt->str;
   int haming_dis =0;
   int i;

   for(i=0;i<STR_LENGTH;i++){
    if( str1[i] != str2[i] ) haming_dis++;
   }

   return haming_dis;
}

/*
  write the data back to a file only if the flag is  t

  */

void write_data(char * file_name)
{
  FILE * file_out = fopen(file_name, "w");
  int i;
  Data * data_pt;

  if( file_out == NULL ){
   printf("Error in open file %s for writing\n");
   return;
  }


  for( i=0; i<TOTAL_NUM; i++)
   {
    data_pt = get_data(i);

    if(data_pt != NULL  && data_pt->flag == 't')
         fprintf(file_out, "%s %d\n", data_pt->str, data_pt->weight);

   }


}
```

2

VITA

Jun Huan

Candidate for the Degree of

Master of Science

Thesis: A LOCALIZED CLUSTERING ALGORITHM AND ITS APPLICATION TO DNA STRING PROCESSING

Major Field: Computer Science

Biographical:

Personal Data: Born in Suzhou, Jiangsu, China, P. R. On March 6, 1975, the second son of Shuyu Huan and Chuanhui Gu.

Education: Graduated in July, 1993 from No. 1 Middle School, Changsha, Hunan. Received a Bachelor of Science degree in Biochemistry and Molecular Biology from Peking University in July, 1997; Finished one-year graduate study in University of Illinois, Chicago, Illinois from August, 1997 to August, 1998. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July 2000.

Professional Experience: Employed as a summer research aid from May 1999 to August 1999 by Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois; Employed as a graduate research assistant from September, 1998 to May 1999 by Department of Microbiology, Oklahoma State University, Stillwater, Oklahoma.