Implementation of Planted Motif Search
Algorithms PMS1 and PMS2

Clifford Locke

BioGrid Research Experience for Undergraduates, Summer 2008
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT

Abstract. Large amounts of data from biological sequencing projects have necessitated the development of efficient algorithms for multiple sequence comparison problems. One of these problems, the planted motif search problem, has proven useful in the study of conserved sequences. Many algorithms for motif search have been presented in the literature. This paper outlines two exact, pattern-based algorithms for motif search, algorithms PMS1 and PMS2, and analyzes implementations of both.

1 Introduction

Voluminous data have evolved from genome and protein sequencing projects and many biologists are in the process of extracting important information from the data. Though useful on their own, biological sequences from many organisms provide a wealth of information when examined for similarities. As a result, efficient computational methods for multiple sequence comparison are becoming imperative for advancement in biological research. This paper outlines and describes the implementation of two algorithms, PMS1 and PMS2, for a specific multiple sequence comparison problem, planted motif search.

1.1 Importance of Multiple Sequence Comparison

Most multiple sequence comparison problems seek to extract commonalities from a given set of sequences and have become especially important in DNA sequence analysis. As outlined in [1], because DNA codes for the specific characteristics of all living organisms, similarities in multiple organisms’ DNA are useful in deducing biological information. Commonalities in different species’ DNA, for example, may expose key evolutionary relationships and resolve missing links in phylogenetic trees. Since the structure of biological molecules usually determines their function, and DNA sequences code for the amino acid sequences of proteins, the discovery of preserved DNA sequences facilitates protein function studies, protein categorization, and gene characterization.
Two universal approaches to multiple sequence comparison problems are outlined in [1]. For some problems, biological relationships are deduced once common subsequences are found computationally. Other problems seek to decipher the common molecular subsequences responsible for a known biological relationship. Algorithms PMS1 and PMS2 focus on the latter approach.

1.2 Planted Motif Search

Motif search involves finding common functional subsequences, known as motifs, in a given set of biological sequences. Various models exist for motif search; algorithms PMS1 and PMS2 are derived from a model outlined in [2] and [3] based on Hamming distance. Hamming distance is defined as the number of mismatches between two strings of equal length. A formal definition of the planted motif search problem based on this model is given:

**Planted (l,d)-Motif Search Problem (PMSP):** Input are \( n \) sequences of average length \( m \) and two integers, \( l \) and \( d \). Find the length-\( l \) motif(s) \( M \) given that each input sequence contains at least one variant of each element of \( M \). A variant of any element \( x \) of \( M \) may be defined as any \( l \)-mer \( y \) at a Hamming distance of at most \( d \) from \( x \). A variant can also be termed an occurrence.

Algorithms PMS1 and PMS2 are exact algorithms for motif search, meaning that they always output the correct planted motif(s) [2]. Heuristic algorithms do not always output the correct motifs(s) [2].

Researchers have applied planted motif search algorithms to many biological problems. Applications reported in the literature include phylogenetic footprinting [4], finding transcription factor binding sites [2], and drug target identification [5].

2 Survey of Other Planted Motif Search Algorithms

According to [5], pattern-driven algorithms, such as PMS1 and PMS2, for motif search were first promoted by Staden in [6]. Early pattern-driven algorithms generated all \( \sum^l \) possible \( l \)-mers, where \( \sum \) refers to the alphabet used and \( |\sum| \) is the size of the alphabet, and output those qualifying as motifs [5]. In [7], Tompa successfully applies such an algorithm to the ribosome binding site problem. However, for large values of \( l \), such algorithms become impractical due to memory limitations and runtime considerations.

PROJECTION [8] is a pattern-based, heuristic algorithm that attempts to improve the results of algorithms such as CONSENSUS [10] and GibbsDNA [11], which rely on local alignment and probability scores to find the correct planted motifs. Such scoring-based algorithms perform poorly for challenging cases such as (14,4), (15,4), (16,5), and (18,6) [8]. Such cases are difficult because any two occurrences of a
motif in the input set may differ in up to $2d$ places and relatively large values of $d$, therefore, introduce spurious motifs that can mask the planted motif [5]. In the (18,6) case, for example, occurrences may differ in up to 12 places, which constitutes two thirds of any given occurrence. Pattern-based algorithms offer an advantage over scoring-based algorithms because the probability matrices of most scoring-based algorithms favor motif occurrences with concentrated mismatches [8]. Pattern-based approaches do not differentiate between concentrated and scattered mismatches [8]. This reduces the probability that spurious motifs mask the correct ones.

PROJECTION improves scoring-based algorithms by initially partitioning the $l$-mers of every input sequence, collectively, into groups based on random projections, then applying statistical analysis to find the correct motif.

WINNER [5] employs a graphical procedure that places the $l$-mers of each input sequence at the vertices of a graph. Any two $l$-mers from different sequences at a Hamming distance of up to $2d$ are connected by an edge, and the algorithm finds cliques in the graph to generate the motif. As stated in [5], WINNER requires copious time and memory and is slow for large input sets. MITRA [9] improves WINNER by using a mismatch tree to split the collection of input $l$-mers into subspaces of sequences with a common prefix. Cliques are first found in the subspaces and “weak” subspaces are eliminated. Remaining subspaces are further divided into more subspaces using longer prefixes, and the process is repeated. Once a prefix of length $l$ is reached, the subspace is given a score, and the correct motifs are derived from the best-scoring subspaces. MITRA is the best-performing algorithm of those aforementioned [2].

Hamming distance is a popular measure of the difference between two sequences for PMSP. Most of the aforementioned algorithms utilize Hamming distance or otherwise only consider substitutions in calculating the distance between two sequences. Other algorithms for motif search may employ edit distance, a dynamic programming algorithm which accounts for substitutions, insertions, and deletions. This becomes useful when the alignment of two strings of different length is desirable or necessary. Refer to [1] for a detailed description of edit distance. Thota, Balla, and Rajasekaran outline algorithms EDMS1 and EDMS2 in [12] to solve the Edit Distance Motif Problem (EDMP).

3 Implementation of Planted Motif Search Algorithms

Algorithms PMS1 and PMS2 were coded using Microsoft Visual C++ and tested on a 1.66 GHz, 1 GB RAM machine. A data file containing twenty ($n = 20$) randomly generated DNA sequences of six hundred nucleotide characters ($m = 600$) was input to both implementations for each planted $(l,d)$-motif problem solved. For each test case, a variant of a randomly generated motif $M$ (of corresponding length $l$) was inserted into each input sequence at a random position. Algorithmic
performance was analyzed based on program runtime and whether or not the correct motif \( M \) was output (amongst possible other spurious motifs). Section 4 outlines and discusses the results of experimentation. The rest of Section 3 is devoted to explanations of each algorithm and the corresponding code.

3.1 Overview of Algorithm PMS1

Rajasekaran, Balla, and Huang (2005) give a detailed description of algorithm PMS1 in [2]. From each of the \( n \) input sequences \( (S_1, S_2, \ldots, S_n) \) all possible \( l \)-mers are generated; let \( C_i \) contain every \( l \)-mer of \( S_i \). For every \( l \)-mer \( u \in C_i \), all possible \( l \)-mers \( v \) are generated such that \( v \) is at a Hamming distance of at most \( d \) from \( u \). Such \( l \)-mers \( v \) are individually termed neighbors of \( u \) and are members of the neighborhood of \( u \). Let \( L_i \) contain the collection of neighborhoods of all \( l \)-mers \( u \) from \( C_i \). All sets \( L_i \) are alphabetically sorted and duplicates are eliminated. Any \( l \)-mer that is an element of every set \( L_i \) is added to the solution set \( M \) of motifs. Using algorithm PMS1, any given PMSP may be solved in \( O(n l \sum |\Sigma| - 1)^d \frac{l}{w} \) time, where \( w \) is the word length of the computer [3].

3.2 Implementation of Algorithm PMS1

Source code for algorithm PMS1 may be found in Appendix A. Code for every function used in the PMS1 and PMS2 implementations is given in Appendix C. To start, the user is prompted to input values for Hamming distance \( (d) \) and motif length \( (l) \), represented by “hamming_distance” and “motif_length,” respectively, in the code. If the Hamming distance exceeds the motif length, an error message appears and new values are requested. The program then requests the total number of DNA sequences \( (n) \) and the filename of the data file containing the DNA sequences. A vector of strings (“dna”) stores the sequences, which are input from the data file as such:

\[
\text{for } i := 0 \text{ to } n - 1 \text{ do}
\]

\[
\text{Line } i + 1 \text{ of the data file is input and stored in element } i \text{ of “dna.”}
\]

Next, each neighborhood \( L_i \) is created by passing each \( l \)-mer of the corresponding input sequence \( S_i \) into the function “create_neighborhood.” In addition to an \( l \)-mer, this function accepts the Hamming distance and a counter \( i \),

---

1 Names of program variables are placed inside of quotations.
2 C++ counts positions in strings, arrays, and vectors starting from zero (as opposed to one).
entered as (-1) when the function is called in the main program, as input parameters. An explanation of the function "create_neighborhood" is given:

```plaintext
for j := i + 1 to k - d do
    Note that k is equal to the length of the input string. If the input Hamming distance is equal to zero, output the input string. Otherwise, change the character at position j of the input k-mer. If the input Hamming distance is greater than one, call the function again with the modified k-mer as the input string, the original Hamming distance minus one as the input Hamming distance, and the positional integer j as the counter. Thus, the string is modified at the next position to the right as the “for” loop is repeated. Once the input Hamming distance is equal to one, the number of changed characters is equal to the original, user-input Hamming distance (d), and the continuation of the for loops allows for all permutations of d positions in k characters to be modified to all 3^d possible combinations. This function creates the neighborhood of all strings at a Hamming distance of exactly d from the input k-mer. To allow for the creation of all neighbors within a Hamming distance d, the function calls itself in the “return” statement to create the neighborhood of the input k-mer at a Hamming distance of the original distance minus one and concatenates the result.
```

Neighborhoods of all the l-mers of input sequence S_i are concatenated to generate the neighborhood L_i for the entire input sequence. As neighborhoods are created, they are sorted using Radix sort and passed into the function “eliminate_duplicates” to eliminate duplicate l-mers. Shown below is a detailed description of the “eliminate_duplicates” function.

```plaintext
for i := 0 to m - 2*k do
    Passed into the function are a set of (concatenated) sorted k-mers and the length k of each element. Add the k-mer beginning at position i of the input string to the output string only if it is not equal to the k-mer beginning at position i + k. The k-mer at i = m - 2*k is always appended to the output string. This procedure ensures that duplicates are added only once to the output string. Specifically, l is passed into the function as k for algorithm PMS1.
```

To determine which l-mers occur in every neighborhood, an iterative intersection procedure is performed on the neighborhoods as they are generated. A string variable "intersection_all" is declared and initially set equal to the neighborhood of the first input sequence, L_1 ("neighborhood1"). As each of the other neighborhoods is generated, “intersection_all” is set equal to the

---

3 With the exception of the input strings, all sets of strings (e.g., neighborhoods and motif sets) are stored as individual string values as opposed to arrays or vectors of string values.
intersection of itself with the new neighborhood using the function “intersect_neighborhoods.” Once all of the neighborhoods have been generated and intersected, “intersection_all” contains the set $M$ of $(l,d)$-motifs. As described in [2], such a procedure conserves memory because only one neighborhood and the merged set are stored in memory at any given point. Below is an explanation of the function “intersect_neighborhoods.”

\[
i := 0, \quad j := 0
\]

while $i <= m - k$ or $j <= m - k$

Two sorted sets of concatenated $k$-mers and the length $k$ of the sets’ elements are passed into the function. Consider the $k$-mer beginning at position $i$ of one input string and position $j$ of the other. If both $k$-mers are equal, append that $k$-mer to the output string and add $k$ to both $i$ and $j$. If they are not equal, add $k$ to the position counter $i$ or $j$ corresponding to the $k$-mer which comes first in alphabetical order. Once either $i$ or $j$ exceeds $m - k$, return the output string. For algorithm PMS1, $l$ is passed into the function as $k$.

The following is a representation of the general code for the program.

for $i := 0$ to $m - l$

Pass the $l$-mer beginning at position $i$ of the first input sequence into the “create_neighborhood” function and concatenate the result to the “neighborhood1” string. Once the entire neighborhood is generated it is passed into the Radix sort and “eliminate_duplicates” functions, in that order, and the resulting string is stored in “intersection_all.”

for $j := 1$ to $n - 1$

for $k := 0$ to $m - l$

For the input sequence stored in element $j$ of “dna” ($S_{j+1}$), repeat the process performed on the first input sequence to generate, sort, and eliminate duplicates from neighborhood $L_{j+1}$. Set the string “intersection_all” equal to the intersection of itself and the neighborhood using the “intersect_neighborhoods” function.

Newline characters are inserted every $l$ positions along “intersection_all” for formatting purposes once all neighborhoods are generated and intersected. At the end of the program, all motifs are output to the screen with a count of how many motifs were found.

3.3 Overview of Algorithm PMS2
Algorithm PMS2, introduced in [2] and [3], is an improvement of algorithm PMS1 that exploits the following observations. For each \((l,d)\)-motif \(M\), at least \(l - k + 1\) length-\(k\) substrings of \(M\) must occur in every input sequence. There must also be at least one position \(i_j\) in every input sequence such that each successive \(k\)-mer of \(M\) occurs at every position from \(i_j\) to \(i_j + l - k\). Algorithm PMS2 utilizes such \(k\)-mers to find the \((l,d)\)-motifs.

A modified PMS1 algorithm is first employed to solve the planted \((d+c,d)\)-motif problem with an appropriate value for \(c\). Let \(R\) be the set of all \((d+c)\)-motifs. Every occurrence of each motif in \(R\) is found in an arbitrary input sequence \(S_j\), noting the location in \(S_j\) of each occurrence. Define \(L_i\), for \(1 \leq i \leq m\), as the set of all \((d+c)\)-motifs with a variant starting at position \(i\) of \(S_j\). If \(M_1\) and \(M_2\) are members of \(L_i\) and \(L_{i + (d+c)}\), respectively, such that the last \(2(d+c) - l\) characters of \(M_1\) are equal to the first \(2(d+c) - l\) characters of \(M_2\), an \(l\)-mer is formed by appending the last \(l - (d+c)\) characters of \(M_2\) onto \(M_1\). If such an \(l\)-mer is at a Hamming distance of \(d\) or less from the \(l\)-mer of \(S_j\) starting at position \(i\), then the \(l\)-mer is added to a list \(C\) of candidate \((l,d)\)-motifs. Each candidate is checked to determine whether or not it is a motif in \(O(znm)\) time, where \(z\) is the number of candidates in \(C\). Algorithm PMS2 runs in time

\[ O(nm \sum_{j=0}^{d} (d+c)(\lceil \Sigma | -1 \rceil) \frac{d + c}{w} + znm + \sum_{i=1}^{m-l+1} |L_i| \parallel |l-(d+c)| l) \],

where \(w\) is the word length of the computer [3].

### 3.4 Implementation of Algorithm PMS2

Appendix B shows the source code for algorithm PMS2. An implementation of algorithm PMS1 is used to solve the planted \((d+c,d)\)-motif problem. To accomplish this, an extra user-input integer variable “motif_sublength” is used to store the value of \(d+c\) and appears in the program wherever “motif_length” appeared in the PMS1 implementation (Appendix A). If the input value for \(d+c\) is greater than \(l\), an error message is generated and new values are requested for \(d\), \(l\), and \(d+c\).

Once the set of \((d+c)\)-motifs \(R\) is generated, the location \(i\) of every occurrence of \(R\) in the first input sequence is determined, and the corresponding members of the solution set are stored in element \(i\) of a vector (“L”) of length \(m - (d+c)\); such a vector corresponds to the sets \(L_i\) outlined in Section 3.3. This is accomplished by the following method:

for \(j := 0 \rightarrow |R| - (d+c)\) do
  for \(i := 0 \rightarrow m - (d+c)\) do
    Note that, for each iteration, \(j\) is incremented by \((d+c)\) because \(R\) is the concatenation of every \((d+c)\)-motif of the input set. Integer \(i\), the starting position of every \(l\)-mer in the first input sequence, is incremented by one. If the Hamming distance between the \((d+c)\)-motif beginning at position \(j\) of \(R\)
and the \((d+c)\)-mer beginning at position \(i\) of the first input sequence is less than or equal to \(d\), concatenate the \((d+c)\)-motif to \(L_i\) ("L\([i]\)").

Candidate list \(C\) ("candidates") is next constructed according to the requirements outlined in Section 3.3. Once the candidates are determined, the list is sorted and duplicates are eliminated.

\[
\text{for } i := 0 \text{ to } m - l \text{ do }
\text{for } j := 0 \text{ to } |L_i| - (d+c) \text{ do }
\text{for } k := 0 \text{ to } |L_{i+l, (d+c)}| - (d+c) \text{ do }
\]

All sets \(L_i\) are stored as concatenations of \((d+c)\)-mers, such that \(j\) and \(k\) must be incremented by \((d+c)\) at the end of each iteration; \(i\) is the starting position of each \(l\)-mer of the first input sequence and is incremented by one. Let \(M_1\) ("M1") be equal to the \((d+c)\)-mer starting at position \(j\) of \(L_i\) ("L\([i]\)") and let \(M_2\) ("M2") be equal to the \((d+c)\)-mer starting at position \(k\) of \(L_{i+l,(d+c)}\) ("L\([i+motif\_length-motif\_sublength]\)"). If the substring containing the last \(2(d+c) - l\) characters of \(M_1\) is equal to that of the first \(2(d+c) - l\) characters of \(M_2\), an \(l\)-mer \(B\) is formed by concatenating the last \(l - (d+c)\) characters of \(M_2\) to \(M_1\). If the Hamming distance between the \(l\)-mer at position \(i\) of the first input sequence and \(B\) is less than or equal to \(d\), concatenate the latter to the list of candidates \(C\) ("candidates").

As a final step, each candidate stored in \(C\) is checked whether it is a motif. To do this, each candidate is compared to each \(l\)-mer of each input sequence other than the first. Once a candidate is found not to occur in an input sequence, the candidate is erased. Any remaining candidates after the checking procedure is completed are motifs and are output in the same manner as the motifs in the PMS1 implementation.

\[
\text{for } i := 1 \text{ to } n - 1 \text{ do }
\text{for } j := 0 \text{ to } |C| - l \text{ do }
\text{for } k := 0 \text{ to } m - l \text{ do }
\]

At the end of each iteration, \(j\) is incremented by \(l\) because \(C\) is the concatenation of length-\(l\) candidate motifs; \(i\) and \(k\) are incremented by one. Each candidate beginning at position \(j\) of "candidates" is compared to the \(l\)-mer beginning at position \(k\) of the input sequence stored in element \(i\) of "dna" (\(S_{n+1}\)). If the Hamming distance between the candidate and the \(l\)-mer of \(S_{n+1}\) is less than or equal to \(d\), the next candidate is immediately considered starting from \(k = 0\). Otherwise, the candidate is compared to the next \(l\)-mer of the input sequence. If the candidate is found to not occur in any given input sequence, it is erased from "candidates."
4 Results and Discussion

For each planted \((l,d)\)-motif problem tested, both algorithms output the correct planted motif, an expected result for any exact algorithm. Table 1 shows the measured runtimes of algorithms PMS1 and PMS2 for the indicated planted \((l,d)\)-motif problems.

<table>
<thead>
<tr>
<th>((l,d))</th>
<th>PMS1 Runtime (Sec.)</th>
<th>PMS2 Runtime (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((9,2))</td>
<td>53.343</td>
<td>(d+c = 5: 305.672)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 6: 342.657)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 7: 394.234)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 8: 72.609)</td>
</tr>
<tr>
<td>((10,2))</td>
<td>73.203</td>
<td>(d+c = 7: 547.25)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 8: 72.344)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 9: 53.828)</td>
</tr>
<tr>
<td>((11,2))</td>
<td>89.704</td>
<td>(d+c = 7: 705.75)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 8: 70.25)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 9: 54.046)</td>
</tr>
<tr>
<td>((12,2))</td>
<td>118.266</td>
<td>(d+c = 8: 76.468)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 9: 54.312)</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>(d+c = 10: 71.531)</td>
</tr>
<tr>
<td>((11,3))</td>
<td>1076.23</td>
<td>(d+c = 10: 1105.03)</td>
</tr>
<tr>
<td>((12,3))</td>
<td>1552.47</td>
<td>(d+c = 10: 1059.83)</td>
</tr>
</tbody>
</table>

Table 1. Performance of PMS1 and PMS2 implementations.

Similar data obtained by Rajasekaran, Balla, and Huang in [2] show much shorter runtimes for both algorithms. For instance, their implementation of PMS2 solved \((9,2)\) in 1.44 seconds, \((10,2)\) in .84 seconds, and \((11,2)\) in .84 seconds. A faster machine was used to test their implementations (2.4 GHz as opposed to 1.66 GHz), but that alone does not account for such large differences in runtimes. Notable are multiple optimization measures in [2] that were not taken in this report’s implementation. Each sequence was represented by a set of integers in [2], and each nucleotide character corresponded to two bits of memory storage. If an integer used two bytes of memory, for instance, such an integer would represent eight residues. Implementations for this paper expressed DNA sequences as strings of eight-bit characters, such that memory requirements were four times as large. Also, arrays of strings may have provided a more efficient procedure than the concatenation of substrings to store sets.

Despite slower-than-expected runtimes, experimentation shed light on the appropriate uses of both algorithms. Table 1 shows that algorithm PMS1 was faster than PMS2 for both \((9,2)\) and \((11,3)\), which are challenging instances (see Section 2
As \( l \) increases (with constant \( d \)), however, PMS2 may or may not run faster than PMS1 depending on the value chosen for \( c \). Larger values of \( c \), in most cases, resulted in shorter runtimes for algorithm PMS2.

Notable in Table 1 are the large runtimes of algorithm PMS2 for \( d = 2 \) and \( d + c \leq 7 \) relative to \( d = 2 \) and \( d + c > 7 \). A probable cause is the high number of motifs generated for \((l,2)\) instances where \( l \leq 7 \). Such instances were tested; solutions to \((5,2)\) and \((6,2)\) gave all \(4^l\) possible \( l \)-mers in \( M \), and \((7,2)\) gave close to \(4^7\) motifs. A high number of candidates must have resulted from the large \((d+c)\)-motif solution sets when \( d + c \leq 7 \), thereby extending runtimes. This result confirms the statement in [2] that \( c \) should be chosen as the largest integer that allows for practical use of algorithm PMS1. If \( c \) is too small, large \((d+c, d)\)-motif sets hinder the performance of algorithm PMS2.

These results imply that algorithm PMS2 is likely impractical for challenging instances of the planted \((l,d)\)-motif problem. Any value for \( d + c \), which must be lower than \( l \), would give a high number of spurious \((d+c, d)\)-motifs if \((l,d)\) corresponded to a challenging instance. Therefore, algorithm PMS1 is better suited for challenge problems than algorithm PMS2, but PMS2 is preferred for planted \((l,d)\)-motif problems where \( l \) is higher than it would be for a challenging case.

5 Conclusions

Algorithms PMS1 and PMS2 were first introduced in [2] to provide exact, pattern-based approaches to the planted \((l,d)\)-motif problem that were built on ideas different from those of past algorithms (e.g., WINNOWER and MITRA). Implementations of both algorithms were presented in this paper to explore and better understand prior work. Trends in the experimental results reflected past results while providing a useful comparison of the two algorithms.

Acknowledgements

A special thanks to my supervisor, Dr. Sanguthevar Rajasekaran, and the University of Connecticut Department of Computer Science and Engineering for their support and to the National Science Foundation for its sponsorship.

References


Appendix A – Algorithm PMS1 Source Code

/**********************************************************************************
* Clifford Locke                                                           *
* BioGrid Research Experience for Undergraduates, Summer 2008.            *
* University of Connecticut, Storrs, CT.                                 *
* Supervisor: Prof. Sanguthevar Rajasekaran, Department of Computer Science and *
* Engineering                                                          *
* Sponsor: National Science Foundation                                    *
* This program implements algorithm PMS1, intended to solve the planted   *
* (l,d)- motif search problem.                                           *
***********************************************************************************/

#include<iostream>
#include<cmath>
#include<string>
#include<time.h>
#include<cstdlib>
#include<iomanip>
#include<assert.h>
#include<vector>
#include<fstream>
#include<algorithm>
using namespace std;

int main()
{
    /*Declare variables for file i/o.*/
    string filename;
    ifstream dna_file;

    /*Input variables are declared.*/
    int hamming_distance, motif_length, file_number, motif_count;

    /*Declare string variables used later.*/
    string neighborhood1, *intersection_all;

    /*Functions are initialized*/
    string create_neighborhood(string base_string, int distance, int i);
    void radix_sort(string &input, int subsequence_length);
    string eliminate_duplicates(string* input, int subsequence_length);
    string intersect_neighborhoods(string *string1, string *string2, int
    subsequence_length);

    /*Prompts user to enter the Hamming distance and motif length.*/
    cout<<"Enter the Hamming distance (d): ";
    cin>>hamming_distance;

    cout<<"Enter the motif length (l): ";
    cin>>motif_length;

    /*Ouputs error message if the Hamming distance is longer than the motif
    and prompts the user to enter new values.*/
    if(hamming_distance>motif_length)
    {
        cout<<"Error. The Hamming distance must be less than the motif
        length.\n";
        cout<<"Enter a new value for the Hamming distance: ";
        cin>>hamming_distance;
cout<<"Enter a new value for motif length: ";
cin>>motif_length;
}

//Prompts the user to enter the total number of input sequences.
cout<<"Enter the total number of DNA sequences: ";
cin>>file_number;

/*Declare a vector to store all input strings.*/
vector<string> dna(file_number);

/*Prompts the user to enter the name of the file
containing the DNA sequence data.*/
cout<<"Enter the filename: ";
cin>>filename;

//Opens the data file.
dna_file.open(filename.data());

/*Prompts the user to enter a valid filename if the file
does not open properly.*/
while( dna_file.fail() )
{

dna_file.clear();
cout<<"Invalid filename. Enter a valid filename: ";
cin >> filename;
dna_file.open(filename.data());
}

/*Each input sequence is contained on a different line of
the input file. This loop stores each input sequence in
a vector for use later.*/
int file_counter = 0;
while(file_counter<file_number)
{

dna_file>>dna[file_counter];

file_counter++;
}

//Closes the input file.
dna_file.close();

//Declares variables used to measure runtime.
clock_t start, stop;

//Starts timing.
double t = 0.0;
assert( (start = clock()) != -1);

/*The first neighborhood is created.*/
for(int i = 0; i<=(dna[0]).length()-motif_length; i++)
{
    neighborhood1 += create_neighborhood((dna[0]).substr(i, motif_length),
hamming_distance, -1);
}

/*The neighborhood is sorted.*/
radix_sort(neighborhood1, motif_length);

/*Duplicates are eliminated.*/
neighborhood1 = eliminate_duplicates(&neighborhood1, motif_length);
/*Sets the motif string equal to neighborhood 1 for recursive intersecting later*/
intersection_all = &neighborhood1;

/*Generates the motif set.*/
for(int j = 1; j<file_number; ++j)
{
    /*A string to store the neighborhood of each input string is declared.*/
    string neighborhood;

    /*Creates the neighborhoods for each input sequence*/
    for(int k = 0; k<=(dna[j]).length() - motif_length; ++k)
        neighborhood += create_neighborhood((dna[j]).substr(k,
            motif_length), hamming_distance, -1);

    /*Sorts and eliminates duplicates.*/
    radix_sort(neighborhood, motif_length);
    neighborhood = eliminate_duplicates(&neighborhood, motif_length);

    /*Intersection_all is the intersection of itself and the new neighborhood. When done recursively, intersection_all contains all of the motifs after all neighborhoods have bee generated and intersected.*/
    *intersection_all = intersect_neighborhoods(&(*intersection_all),
        &neighborhood, motif_length);

    /*If, at any point, intersection_all is an empty set, the program is aborted because no motifs were found.*/
    if((*intersection_all).length() == 0)
    {
        cout<< "No motifs were found.\n\n";
        return 0;
    }
}

/*Calculates the total number of motifs found.*/
motif_count = (*intersection_all).length()/motif_length;

/*Insert newline characters to separate the subsequences of length motif_length*/
for( int y = motif_length; y<(*intersection_all).length()-(motif_length-1); y += (motif_length+1))
{
    int z = 0;
    (*intersection_all).insert(y + z, "\n");
    z++;
}

cout<<endl<<"Motifs found:\n\n"<<*intersection_all<<endl;
cout<<endl<<motif_count<<" motifs were found.\n\n";

//Records stop time and outputs the calculated runtime.
stop = clock();
t = (double) (stop-start)/CLOCKS_PER_SEC;
cout << "Runtime: " << t << " seconds.\n\n";
return 0;
}
Appendix B – Algorithm PMS2 Source Code

#include<iostream>
#include<cmath>
#include<string>
#include<cstdlib>
#include<iomanip>
#include<assert.h>
#include<vector>
#include<fstream>
#include<time.h>
using namespace std;

int main()
{
    string neighborhood1, *intersection_all, filename;
    ifstream dna_file;
    int hamming_distance, motif_length, motif_sublength, file_number, motif_count;
    string create_neighborhood(string base_string, int distance, int i);
    void radix_sort(string &input, int subsequence_length);
    string eliminate_duplicates(string* input, int subsequence_length);
    string intersect_neighborhoods(string* string1, string* string2, int subsequence_length);

    cout<< "Enter the Hamming distance (d): ";
    cin>>hamming_distance;
    cout<< "Enter the motif length (l): ";
    cin>>motif_length;
    cout<< "Enter the desired motif substring length (d+c): ";
    cin>>motif_sublength;

    if((hamming_distance>motif_length)||(motif_sublength>motif_length))
    {
        cout<<"Error.  Invalid values were entered.\nEnter a new value for the Hamming distance.\n";
        cin>>hamming_distance;
        cout<<"Enter a new value for motif length.\n";
        cin>>motif_length;
        cout<<"Enter a new value for the motif substring length.\n";
        cin>>motif_sublength;
    
/*Prompts user to enter the Hamming distance, motif length, and the motif substring length.*/
    cout<<"Enter the Hamming distance (d): ";
    cin>>hamming_distance;
    cout<<"Enter the motif length (l): ";
    cin>>motif_length;
    cout<<"Enter the desired motif substring length (d+c): ";
    cin>>motif_sublength;

    /*Ouputs an error message if the Hamming distance is longer than the motif length or if the motif substring length is higher than the motif length.  New values are requested.*/
    if((hamming_distance>motif_length)||(motif_sublength>motif_length))
    {
        cout<<"Error.  Invalid values were entered.\nEnter a new value for the Hamming distance.\n";
        cin>>hamming_distance;
        cout<<"Enter a new value for motif length.\n";
        cin>>motif_length;
        cout<<"Enter a new value for the motif substring length.\n";
        cin>>motif_sublength;
    
/*Prompts user to enter the Hamming distance, motif length, and the motif substring length.*/
    cout<<"Enter the Hamming distance (d): ";
    cin>>hamming_distance;
    cout<<"Enter the motif length (l): ";
    cin>>motif_length;
    cout<<"Enter the desired motif substring length (d+c): ";
    cin>>motif_sublength;

    /*Ouputs an error message if the Hamming distance is longer than the motif length or if the motif substring length is higher than the motif length.  New values are requested.*/
    if((hamming_distance>motif_length)||(motif_sublength>motif_length))
    {
        cout<<"Error.  Invalid values were entered.\nEnter a new value for the Hamming distance.\n";
        cin>>hamming_distance;
        cout<<"Enter a new value for motif length.\n";
        cin>>motif_length;
        cout<<"Enter a new value for the motif substring length.\n";
        cin>>motif_sublength;

    /*Prompts user to enter the Hamming distance, motif length, and the motif substring length.*/
    cout<<"Enter the Hamming distance (d): ";
    cin>>hamming_distance;
    cout<<"Enter the motif length (l): ";
    cin>>motif_length;
    cout<<"Enter the desired motif substring length (d+c): ";
    cin>>motif_sublength;

    /*Ouputs an error message if the Hamming distance is longer than the motif length or if the motif substring length is higher than the motif length.  New values are requested.*/
    if((hamming_distance>motif_length)||(motif_sublength>motif_length))
    {
        cout<<"Error.  Invalid values were entered.\nEnter a new value for the Hamming distance.\n";
        cin>>hamming_distance;
        cout<<"Enter a new value for motif length.\n";
        cin>>motif_length;
        cout<<"Enter a new value for the motif substring length.\n";
        cin>>motif_sublength;

    /*Prompts user to enter the Hamming distance, motif length, and the motif substring length.*/
    cout<<"Enter the Hamming distance (d): ";
    cin>>hamming_distance;
    cout<<"Enter the motif length (l): ";
    cin>>motif_length;
    cout<<"Enter the desired motif substring length (d+c): ";
    cin>>motif_sublength;

    /*Ouputs an error message if the Hamming distance is longer than the motif length or if the motif substring length is higher than the motif length.  New values are requested.*/
    if((hamming_distance>motif_length)||(motif_sublength>motif_length))
    {
        cout<<"Error.  Invalid values were entered.\nEnter a new value for the Hamming distance.\n";
        cin>>hamming_distance;
        cout<<"Enter a new value for motif length.\n";
        cin>>motif_length;
        cout<<"Enter a new value for the motif substring length.\n";
        cin>>motif_sublength;
Prompts the user to enter the total number of input sequences.
cout<<"Enter the total number of DNA sequences: ";
cin>>file_number;

/*This program solves the (d+c,d)-motif problem in the same manner that the
PMS1 algorithm in Appendix A solves the (l,d)-motif problem. See Appendix A
or Section 3 if clarification is necessary.*/

vector<string> dna(file_number);
cout<<"Enter the filename containing all DNA sequences: ";
cin>>filename;
dna_file.open(filename.data());

while( dna_file.fail() )
{
    dna_file.clear();
cout<<"Invalid filename. Enter a valid filename: ";
cin >> filename;
dna_file.open(filename.data());
}

int file_counter = 0;
while(file_counter < file_number)
{
    dna_file >> dna[file_counter];
    file_counter++;
}
dna_file.close();
clock_t start, stop;
double t = 0.0;
assert( (start = clock()) != -1);
for(int i = 0; i<=(dna[0]).length()-motif_sublength; i++)
{
    neighborhood1 += create_neighborhood((dna[0]).substr(i, motif_sublength), hamming_distance, -1);
}
radix_sort(neighborhood1, motif_sublength);
neighborhood1 = eliminate_duplicates(&neighborhood1, motif_sublength);
intersection_all = &neighborhood1;

for(int j = 1; j<file_number; ++j)
{
    string neighborhood;
    for(int k = 0; k<=(dna[j]).length() - motif_sublength; ++k)
        neighborhood += create_neighborhood((dna[j]).substr(k, motif_sublength), hamming_distance, -1);
    cout<<endl<<"Neighborhood "+j+" created.\n\n";
    radix_sort(neighborhood, motif_sublength);
    neighborhood = eliminate_duplicates(&neighborhood, motif_sublength);
    *intersection_all = intersect_neighborhoods(&(*intersection_all), &neighborhood, motif_sublength);
if((*intersection_all).length() == 0)
{
    cout<< "No motifs were found.\n\n";
    return 0;
}

cout<<(*intersection_all).length()/motif_sublength<<endl<<endl;

/*when determining the occurrences of each element of R in the first input
sequence (S1), variable "element_R" will store a (d+c)-motif and "element_S"
will store a (d+c)-mer from S1.*/
string element_R, element_S;

/*Declares a vector "L" that will store all of the occurrences of the (d+c)-
motif set R at each position of S1. The subscript of the vector will correspond
to the position of the occurrences in S1.*/
vector<string> L ((dna[0]).length() - motif_sublength + 1);

/*Integer used to calculate the Hamming distance between two strings.*/
int h_distance;

/*Determines the location of each occurrence of each (d+c,d) motif in S1*/
for(int k = 0; k<=(*intersection_all).length()-motif_sublength; k+=
    motif_sublength)
{
    for(int l = 0; l <= (dna[0]).length() - motif_sublength; ++l)
    {
        /*Compares each element of the motif string R to each l-mer of S1.
If the Hamming distance between an element of R and any l-mer is
less than or equal to d, the element of R is stored in the vector
element of L whose index corresponds to the position of the
 corresponding l-mer in S1.*/
        element_R = (*intersection_all).substr(k, motif_sublength);
        element_S = (dna[0]).substr(l, motif_sublength);
        h_distance = 0;
        for(int f = 0; f < motif_sublength; f++)
        {
            if(element_R[f] != element_S[f])
                h_distance++;
            if(h_distance > hamming_distance)
                break;
            else
                continue;
        }
        if(h_distance <= hamming_distance)
            L[l] += element_R;
        else
            continue;
    }
}

cout<<"Occurrences at all positions of sequence 1 determined.\n\n";

/*Selects motif candidates using vector L and DNA S1*/

/*Defines a string to hold the candidates (candidates), a string to
hold the l-mer at the corresponding position of S1 (l_mer), and strings
to hold the elements of sets L[i] (M1) and L[i + l - (d+c)] (M2).*/
string candidates, l_mer, M1, M2;
/*These nested loops determine the (l, d)-motif candidates.*/

/*Integer "j" will represent the starting position of each l-mer of S1.*/
for(int j = 0; j <= (dna[0]).length()-motif_length; j++)
{
    if( ( (L[j]).length() == 0 ) || ( (L[j + motif_length -
        motif_sublength]).length() == 0 ) )
        continue;

    /*Integers "n" and "m" will correspond to the starting positions
    of the elements of L[i] and L[i+1-(d+c)], respectively.*/
    for(int n = 0; n <= (L[j]).length()-motif_sublength; n +=
        motif_sublength)
    {
        for(int m = 0; m <= (L[j + motif_length -
            motif_sublength]).length()-motif_sublength; m +=
            motif_sublength)
        {
            //Stores the l-mer from the corresponding input sequence.
            l_mer = (dna[0]).substr(j, motif_length);

            //Stores the appropriate elements from the sets L[i] and
            L[i+1-(d+c)]
            M1 = (L[j]).substr(n, motif_sublength);
            M2 = (L[j + motif_length - motif_sublength]).substr(m,
                motif_sublength);

            /*Appends the last l-(d+c) characters of M2 onto M1 if the
            last 2(d+c)-l characters
            of M1 are equal to the first 2(d+c)-l characters of M2.*/
            if(M1.substr(motif_length -
                motif_sublength, 2*motif_sublength -
                motif_length) == M2.substr(0,
                2*motif_sublength -
                motif_length))
            {
                M1 += M2.substr(2*motif_sublength -
                    motif_length,
                    motif_length-motif_sublength);

                /*This loop calculates the Hamming distance between
                the potential candidate and the corresponding l_mer
                from S1.*/
                h_distance = 0;
                for(int k = 0; k < motif_length; k++)
                {
                    if(M1[k] != l_mer[k])
                        ++h_distance;
                    /*If the calculated Hamming distance exceeds
                    the user-input maximum Hamming distance, the
                    loop is exited.*/
                    if(h_distance>hamming_distance)
                        break;
                    else
                        continue;
                }

                /*The potential candidate is added to the list of
                candidates if the calculated Hamming distance is less
                than or equal to the user-input maximum Hamming
                distance.*/
                if(h_distance<=hamming_distance)
                {
                    candidates += M1;
                }
            }
        }
    }
}
/*The candidates list is sorted and any duplicates are eliminated.*/
radix_sort(candidates, motif_length);
candidates = eliminate_duplicates(&candidates, motif_length);

/*Checks if each candidate is a motif. Candidates are deleted once they are
found to not occur in any given string*/

/*Integer p is used to call the elements of the vector "dna," which stores the
input sequences.*/
for(int p = 1; p < file_number; ++p)
{
    /*These strings will hold an l-mer from an input sequence and
    a candidate motif.*/
    string sequence_l_mer, candidate;

    /*Integer q corresponds to the starting point of each candidate l-mer in
    the string "candidates."*/
    for(int q = 0; q <= candidates.length() - motif_length; q +=
motif_length)
    {
        /*Integer r corresponds to the starting point of each l-mer
        in each input sequence.*/
        for(int r = 0; r <= (dna[p]).length() - motif_length; ++r)
        {
            /*Defines an l-mer from input sequence (p+1) and a
            candidate.*/
            sequence_l_mer = (dna[p]).substr(r, motif_length);
            candidate = candidates.substr(q, motif_length);

            /*Computes the Hamming distance between the candidate and
            the l-mer.*/
            h_distance = 0;
            for(int k = 0; k < motif_length; ++k)
            {
                /*The calculated Hamming distance is incremented when
                a mismatch is found.*/
                if(sequence_l_mer[k] != candidate[k])
                    ++h_distance;
                /*If "h_distance" exceeds "hamming_distance," no
                occurrence has been found and the next input sequence
                l-mer is considered.*/
                if(h_distance > hamming_distance)
                    break;
            }

            /*If "h_distance" does not exceed "hamming_distance," an
            occurrence has been found and the next candidate is
            considered.*/
            if(h_distance <= hamming_distance)
                break;
            else
                continue;
        }
    }
}

/*If this point is reached and "h_distance" exceeds
"hamming_distance," an occurrence was not found and the candidate
is erased.*/
if(h_distance > hamming_distance)
{
    candidates.erase(q, motif_length);
    q = q - motif_length;
}
/*If the candidates list is empty, no motifs were found.*/
if(candidates.length() == 0)
{
    cout << "No motifs were found.\n\n";
    return 0;
}
}
}

motif_count = candidates.length()/motif_length;
for( int y = motif_length; y<candidates.length()-(motif_length-1); y += (motif_length+1))
{
    int z = 0;
    candidates.insert(y + z, "\n");
    z++;
}
cout<<endl<<endl<<"Motifs found:\n\n"<< candidates << endl;
cout<<endl<<motif_count<<" motifs were found.\n\n";
stop = clock();
t = (double) (stop-start)/CLOCKS_PER_SEC;
cout << "Runtime: " << t << " seconds.\n\n";
return 0;
}
Appendix C- Source Code for all Implemented Functions

/* This function will be used to create the neighborhood of all DNA sequences within a
Hamming distance d of any given input sequence*/
string create_neighborhood(string base_string, int distance, int i)
{
    string string1, string2;

    /*If the Hamming distance is equal to zero, there are no neighbors*/
    if(distance == 0)
        return base_string;

    /*The for loop begins the modification of the string. Counters are such that,
when the function is called recursively, the next character is modified*/
    for(int j=i+1; j<=base_string.length()-distance; j++)
    {
        /*To help prevent duplicates, a separate code is constructed for each
character case.*/
        if(base_string[j]=='a')
        {
            string1=base_string;
            string1[j]='c';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            string2+=string1;
            string1=base_string;
            string1[j]='g';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            string2+=string1;
            string1=base_string;
            string1[j]='t';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            continue;
        }
        if(base_string[j]=='c')
        {
            string1=base_string;
            string1[j]='a';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            string2+=string1;
            string1=base_string;
            string1[j]='g';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            string2+=string1;
            string1=base_string;
            string1[j]='t';
            if(distance>1)
                string1=create_neighborhood(string1, distance-1, j);
            continue;
        }
    }
}
string1=create_neighborhood(string1, distance-1, j);
string2+=string1;
continue;
}
if(base_string[j]=='g')
{
    string1=base_string;
    string1[j]='c';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;

    string1=base_string;
    string1[j]='a';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;

    string1=base_string;
    string1[j]='t';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;
    continue;
}
if(base_string[j]=='t')
{
    string1=base_string;
    string1[j]='c';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;

    string1=base_string;
    string1[j]='g';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;

    string1=base_string;
    string1[j]='a';
    if(distance>1)
        string1=create_neighborhood(string1, distance-1, j);
    string2+=string1;
    continue;
}
/*The function on its own produces all neighbor strings at a Hamming distance exactly equal to the input Hamming distance. Thus, the function must be called for all distances 0<d<Hamming distance.*/
return(string2 + create_neighborhood(base_string, distance-1, -1));
*/

/*Function to sort the neighborhood string using Radix Sort.*/
void radix_sort(string &input, int subsequence_length)
{
    /*This string will hold the substrings for sorting.*/
    string hold;

    /*Integer to store the length of the input string.*/
    int length = input.length();
}
/*The Radix sort algorithm begins sorting a list by looking at the last
character of each list element first, then the second to last character, and so
on to the beginning of each element.*/
for(int i = subsequence_length - 1; i >= 0; --i)
{
    /*An array of 4 elements is declared; each element corresponds
to a different letter of the alphabet a,c,g,t.*/
    string Radix[4];
    for(int j = 0; j <= length - subsequence_length; j += subsequence_length)
    {
        /*Hold contains the element of the input set to be categorized.*/
        hold = input.substr(j, subsequence_length);
        /*Places the list element into the corresponding sublist based
        on the element's value at position i.*/
        if(hold[i] == 'a')
            Radix[0] += hold;
        if(hold[i] == 'c')
            Radix[1] += hold;
        if(hold[i] == 'g')
            Radix[2] += hold;
        if(hold[i] == 't')
            Radix[3] += hold;
    }
    /*Erases the input string.*/
    input.erase(0,length);
    /*Input is refilled in increasing character order. At the end of
    "subsequence_length" number of iterations, the list is sorted
    alphabetically.*/
    for(int k = 0; k < 4; k++)
        input += Radix[k];
}
return;
} /*Function to eliminate duplicates. This function requires that
the input string is sorted alphabetically.*/
string eliminate_duplicates(string* input, int subsequence_length)
{
    /*Declares the string to be returned to the main program.*/
    string output;
    /*Declares positional counter "a."*/
    int a;
    //If the list has only one element, return it.
    if( (*input).length()/subsequence_length == 1 )
        return (*input);
    /*Any element of the list is only appended to the output if it is not equal
to the next element, thus ensuring that duplicates are only added once.*/
    for( a = 0; a <=(*input).length()-2*subsequence_length; a += subsequence_length)
    {
        if((input).substr(a, subsequence_length)!=(input).substr(a +
        subsequence_length, subsequence_length))
            {output += (*input).substr(a, subsequence_length);
                continue;
            }
        else
            continue;
/*The last element of the list is always added to
the output string.*/
output += (*input).substr(a, subsequence_length);
return output;
}

/*Function to intersect two alphabetically sorted neighborhoods.*/
string intersect_neighborhoods(string *string1, string *string2, int
subsequence_length)
{
    /*Declare the intersection string*/
    string neighborhood_intersection;

    /*Declares positional counters i and j.*/
    int i=0, j=0;

    /*Consider the list elements in alphabetical order and append only those
    elements that appear in both sets
to the output string.*/
    while((i <= ((*string1).length() - subsequence_length)) || (j<=
    ((*string2).length() - subsequence_length))
    {
        /*If either i or j (indices) are over the maximum, return the
        intersection string*/
        if( i > ((*string1).length() - subsequence_length))
        {
            return neighborhood_intersection;
        }
        if( j > ((*string2).length() - subsequence_length))
        {
            return neighborhood_intersection;
        }

        /*The following if statements compare the substrings of string1 to
        string2.*/
        /*If the list elements are not equal, continue to the next element in the
        list corresponding to the element that comes first alphabetically.*/
        if((*string1).substr(i, subsequence_length) < (*string2).substr(j,
subsequence_length))
        {
            i += subsequence_length;
            continue;
        }
        if((*string2).substr(j, subsequence_length) < (*string1).substr(i,
        subsequence_length))
        {
            j += subsequence_length;
            continue;
        }

        /*If two substrings in each neighborhood are equal, append the substring
to the intersection string.*/
        if((*string2).substr(j, subsequence_length) == (*string1).substr(i,
        subsequence_length))
        {
            neighborhood_intersection += (*string1).substr(i,
            subsequence_length);
            i += subsequence_length;
            j += subsequence_length;
        }
    }
}

}
continue;
}
}

return (neighborhood_intersection);