GENETIC ALGORITHM USING RESTRICTED SEQUENCE ALIGNMENTS

A Thesis Presented To

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

Evgeni Liakhovitch
August, 2000
THIS THESIS ENTITLED
"GENETIC ALGORITHM USING RESTRICTED SEQUENCE ALIGNMENTS"

by Evgueni Liakhovitch

has been approved

for the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology

David W. Juedes
Associate Professor of Electrical Engineering and Computer Science

Jerrel R. Mitchell, Dean
Fritz J. and Dolores H. Russ
College of Engineering and Technology
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Sequence Alignments</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Real World Examples</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Complexity</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Algorithms for Multiple Sequence Alignment</td>
<td>5</td>
</tr>
<tr>
<td>1.6 Restricted Versions of Sequence Alignment</td>
<td>5</td>
</tr>
<tr>
<td>1.7 New Paradigm - Genetic Approach</td>
<td>7</td>
</tr>
<tr>
<td>2. GUSFIELD'S ALGORITHM</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Bounded-error Approximation Algorithms</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Algorithm Outline</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Algorithm Implementation</td>
<td>10</td>
</tr>
<tr>
<td>3. GAP 0-1 ALIGNMENT PROBLEM</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Description</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Using the Gap 0-1 For Improving Existing Alignments</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Fixed Parameter Tractable Algorithm for Gap 0-1</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Semidefinite Programming Algorithm for Gap 0-1</td>
<td>17</td>
</tr>
<tr>
<td>3.4.1 Constructing Quadratic Integer Programming Problem</td>
<td>18</td>
</tr>
<tr>
<td>3.4.2 Constructing Semidefinite Programming Problem</td>
<td>19</td>
</tr>
<tr>
<td>3.4.3 Randomized Approximation Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>3.4.4 Implementation Using CSDP Library</td>
<td>21</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.</td>
<td>GENETIC ALGORITHM</td>
</tr>
<tr>
<td>4.1</td>
<td>Proposed Approach</td>
</tr>
<tr>
<td>4.2</td>
<td>The Genetic Algorithm Description</td>
</tr>
<tr>
<td>5.</td>
<td>EXPERIMENTAL RESULTS</td>
</tr>
<tr>
<td>5.1</td>
<td>Test Suite</td>
</tr>
<tr>
<td>5.1.1</td>
<td>COG Database</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Matrices Used For The Tests</td>
</tr>
<tr>
<td>5.2</td>
<td>Test Results</td>
</tr>
<tr>
<td>5.3</td>
<td>The Resulting Alignment for COG2157</td>
</tr>
<tr>
<td>6.</td>
<td>CONCLUSION</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td></td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
</tr>
<tr>
<td>A.</td>
<td>WEB INTERFACE</td>
</tr>
<tr>
<td>B.</td>
<td>FRAGMENTS OF THE PROGRAM CODE</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Test Suite.</td>
<td>30</td>
</tr>
<tr>
<td>5.2 COG Sizes.</td>
<td>31</td>
</tr>
<tr>
<td>5.3 Parameter settings and the sum of SP-Scores for 10 COGs ($n = 5$)</td>
<td>34</td>
</tr>
<tr>
<td>5.4 Parameter settings and the sum of SP-Scores for 10 COGs ($n = 10$)</td>
<td>35</td>
</tr>
<tr>
<td>5.5 Parameter settings and the sum of SP-Scores for 10 COGs ($n = 15$)</td>
<td>35</td>
</tr>
<tr>
<td>5.6 SP-Scores using the BLOSUM 62 matrix when $n = 15$, $m = 16$, $p_1 = 0.75$, $p_2 = 0.1$</td>
<td>37</td>
</tr>
<tr>
<td>5.7 Timing data</td>
<td>38</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A simple example of an alignment of two strings. The alignment shown here is optimal because all matching characters are aligned.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Fragments of several proteins aligned using MSA</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>a) Multiple alignment of four strings. b) String 4 aligned with the new string 5. c) String 5 is appended to the multiple alignment. Since the alignment of 4 and 5 inserted a new space into 4, a column of spaces is inserted into the rest of strings.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>A center star for five strings with AGCGT as the center string (x^\ast).</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Given two strings (X) and (Y), there are four possible Gap 0-1 alignments. Note that the first and second alignments have identical SP-Scores.</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Step 1 - initial alignment created with Gusfield’s center star algorithm. Steps 2-5 - iterative improvement of the initial alignment using Gap 0-1 algorithm.</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of a multiple alignment of long sequences where a small window of size 3 is chosen to apply a local Gap 0-1 improvement.</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Local improvement using Gap 0-1 alignment. a) The characters (V, K, Q) and (A) are about to be moved. b) When the pass is finished, all characters are in the right columns.</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Sexual reproduction. Each &quot;child&quot; individual inherits approximately half of its genes from the &quot;mother&quot; individual and half from the &quot;father&quot; individual.</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>BLOSUM62 matrix</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Pam250 matrix</td>
<td>33</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>Simulation results ( (n = 12, m = 8, p_1 = 2/3, p_2 = 1/4) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>Screenshot of the web interface for the program</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Computational biology is a relatively new science that has emerged as a result of cooperative work of computer scientists and biologists in solving biological problems using computational techniques. Much of the work in computational biology involves molecular sequences such as DNA and proteins. A comprehensive overview of recent work in computational biology can be found in any number of current textbooks on the subject such as Gusfield [15], Waterman [31], and Durbin et. al. [9].

It is well known that DNA is made up of four nucleotides (adenine, cytosine, guanine, thymine) and typically has a linear, one dimensional structure. Thus, DNA can easily be encoded as a string by assigning a single letter to each nucleotide (A, C, G, T) and listing them in the linear order. Proteins are also linear in structure and consist of 20 amino acids which can be represented by an alphabet of 20 characters. For both DNA and proteins, there exist standard alphabets made up of designated English letters which are widely used.

1.1 Sequence Alignments

Several fundamental problems in computational biology concern sequence alignments. Sequence alignment is the task of aligning characters of two (or more) strings in the best possible way (Figure 1.1). For biologists, the best possible way may have several different interpretations. One approach is to simply maximize the number of matches of pairs of opposing characters. In this case, alignments are produced by inserting spaces in the strings so that matching characters are aligned.
Figure 1.1 A simple example of an alignment of two strings. The alignment shown here is optimal because all matching characters are aligned.

To determine the fitness of an alignment between two strings, a scoring must be performed. To score an alignment of two strings, a scoring matrix is often used. For an alphabet of $n$ characters, a scoring matrix is an $n$ by $n$ symmetric matrix where entry $(i, j)$ contains a score, or a cost of aligning a character $i$ with a character $j$. The most common method of scoring an alignment of two strings is to compute the sum of all the scores of pairs of characters.

0) LAK IFRITGIMSK KGKDP LYFRKEYKALKPEALEILYSEFGGGRKVKSRIKILNIEE
1) MKMKTIFRVKGKFLM GDK LQPFTKELNAIREEEIEELYERLYSEFGSKHRVPRSKVKIEEIEE
2) MEVK VFRVSGYFEK DGRK FK FTKEYRALKEEHVKEVSDLGSRHKVRRKIFIKEIRE
3) M K FEVRGAFKTLLEGWQ K FTKVVEANNERYALEKVSLSLIGSNHKVRN L IK IEE

Figure 1.2 Fragments of several proteins aligned using MSA

Sequence alignment among several strings is the multiple sequence alignment (MSA) problem (Figure 1.2), and it is of great interest to biologists. To score a multiple sequence alignment, a common approach is to compute the sum of the individual scores of all pairs of sequences. This is known as the $SP$-score (SP stands for sum of pairs) [15]. This is the most widely used of the scoring schemes for multiple sequence alignments.
1.2 Motivation

The importance of multiple sequence alignment lies in its ability to help biologists form reasonable hypotheses about the molecular changes that occurred during evolutionary history. Given a collection of related molecular sequences, there are many possible sequences of evolutionary events (insertions, deletions, and genetic mutations) leading to final sequences. Each sequence of evolutionary events provides a possible hypothesis. To determine which of these possible hypotheses is most reasonable, biologists employ the principle of Occam’s Razor [24] which states that “Entities should not be multiplied beyond necessity.” This principle is often restated [29] as follows:

"When you have two competing theories which make exactly the same predictions, the one that is simpler is the better."

This principle can be applied to the problem of revealing the evolutionary relationships between protein sequences and to sequence alignment problems in general. Consider the problem of determining how closely related several protein sequences are to each other. In attempting to determine the biological distance between these two protein sequences, it must be assumed that during the process of evolution one protein sequence can be transformed into another through internal structure changes. Intuitively, those changes include insertions, deletions and replacements (mutations) of certain amino acids. Sequence alignments attempt to measure the degree of these changes, or, in other words, the biological distance between sequences. The shorter this distance is, the simpler the transformation and, according to Occam’s Razor, the greater the probability that such measurement is correct. Thus, sequence alignments attempt to find "the simplest explanation" of biological and evolutionary relationships among the sequences. While exact sequence alignment algorithms find optimal solutions, approximation algorithms do their best to find solutions that are as good (and as simple) as possible.
1.3 Real World Examples

As mentioned previously, sequence alignment is a useful tool for biologists. It has several important applications. Sequence alignments can be used to predict the secondary structure of proteins, or they can help biologists to infer phylogenetic relationships among the organisms. They can also be used to search for similar entries in a database. One good example is the case of the Cystic Fibrosis Gene. Cystic fibrosis is one of the most common genetic diseases in the Caucasian population. For many years scientists were unable to treat cystic fibrosis effectively. Multiple sequence alignment made it possible to classify the protein and to make useful hypotheses that could lead to the discovery of an effective cure [31].

1.4 Complexity

It is well known that the multiple sequence alignment problem can be solved in $\Theta(2^k n^k)$ steps on $k$ sequences of length at most $n$ using dynamic programming [15]. While it is easy to implement this algorithm for any fixed $k$, this approach is not practical even for a small number of sequences. It is also known that the multiple sequence alignment problem is NP-complete [30]. Thus, the optimal solution to the MSA problem cannot be found in polynomial time, unless $P = NP$.

Multiple sequence alignment problem is NP-hard, and hence is intractable. Moreover, it was recently proven that the multiple sequence alignment problem with SP-score is MAX-SNP hard and, hence, does not have a PTAS (polynomial time approximation scheme) unless $P = NP$ [21]. In contrast, it is known that there exist bounded-error approximation algorithms for the multiple sequence alignment problem. These are discussed in the next section.

\footnote{The algorithm was implemented by the author for $k = 4$, and the running time was significant even for very short sequences.}
1.5 Algorithms for Multiple Sequence Alignment

Certain algorithms exist that produce near optimal alignments in polynomial time. These algorithms, called bounded-error approximation algorithms, always find a solution not worse than some known factor. Other algorithms, called heuristics, simply try their best to find a good solution.

One bounded-error approximation algorithm for the multiple sequence alignment problem is Gusfield's Center Star algorithm [14]. Gusfield’s approach guarantees an alignment within $2 - 2/k$ of optimal and, hence, is a bounded-error approximation algorithm. Although this algorithm is reasonably fast and simple, an alignment created using this algorithm can deviate significantly from the optimal alignment.

One well-known heuristic for multiple sequence alignment is called progressive alignment. It works by first aligning two sequences. A third sequence is then selected and aligned to the first alignment. This process continues until all sequences have been aligned [10]. One widely used implementation of a progressive alignment algorithm is the package called CLUSTALW [28], which succeeded an earlier popular program, CLUSTALV [17]. Another well-known approach is the use of probabilistic modeling. An example of this approach is hidden Markov modeling for multiple sequence alignments [9].

1.6 Restricted Versions of Sequence Alignment

Very recent research conducted in the Algorithms and Complexity research group at Ohio University has concentrated on restricted versions of the multiple sequence alignment problem. Jiang [19] recently proposed the investigation of restricted version of the multiple sequence alignment problem where spaces (gaps) are allowed only at the beginning (or end) of a sequence. The two problems that have received the most attention are called the Gap-0 and Gap 0-1 alignment problems. A Gap-0 alignment is an alignment where spaces can be inserted only at the beginning or end
of a sequence. A Gap 0-1 alignment is an alignment where only a single space can be inserted at the beginning or end of a sequence.

Just [22] demonstrated that both the Gap 0 and Gap 0-1 alignment problems are NP-complete and MAX-SNP hard. Moreover, Juedes [20] proved that the Gap 0-1 alignment problem cannot be approximated to within 6.25% of optimal, unless $P = NP$. While both problems are intractable, there is some hope that approximate algorithms for one (or both) of these problems can be used in software for sequence alignment. Since it appears that these restricted versions of the multiple sequence alignment problem are simpler than the unrestricted version, algorithms for these problems may also be useful in solving the unrestricted version.

There is evidence that these sequence alignment problems are easier than the unrestricted version. First, the Gap 0-1 alignment problem is fixed parameter tractable [5]. In contrast, it appears that the unrestricted version of sequence alignment is not fixed parameter tractable [8]. Second, very good bounded error approximation algorithms exist for the Gap 0-1 alignment problem. Juedes [20] demonstrated that there is a bounded error approximation algorithm for the Gap 0-1 alignment problem that always produces an alignment that is within 25% of optimal. This approximation algorithm uses semidefinite programming.

Restricted versions of sequence alignment can be used to produce good multiple sequence alignments. The key to this is that restricted alignments can be used to locally improve precomputed alignments. The power of this approach can be demonstrated by employing a genetic search technique along with local improvement to find near optimal sequence alignments.
1.7 New Paradigm - Genetic Approach

A powerful technique for creating near optimal alignments is the so called *genetic*, or *evolutionary approach* [18, 11]. The principal idea behind this approach is to apply the laws of nature to the sequence alignment problem. In nature, species grow, reproduce, fight for resources, mutate and improve from generation to generation. These evolutionary processes can be simplified and described in a very restricted algorithmic form.

Multiple sequence alignment is a natural candidate for evolutionary algorithms. Evolutionary search techniques for multiple sequence alignments have been studied extensively and implemented by many scientists and research groups, including Notredame and Higgins [25], Zhang and Wong [32], Anabarasu, Narayanasamy, and Sundarararjan [2], Gonzalez, Izquerdo, and Seijas [13], Notredame, Holm and Higgins [26], and Chellapilla and Fogel [6].

In contrast to this previous work, our work employed bounded-error approximation algorithm as part of the genetic algorithm. In this research, a collection of individuals (alignments) is simulated through many generations. Gusfield's center star algorithm is employed to create an initial population of alignments. The Gap 0-1 multiple sequence alignment algorithm is then used to locally improve the individuals.
2. GUSFIELD'S ALGORITHM

2.1 Bounded-error Approximation Algorithms

Because determining the exact solution for a multiple sequence alignment problem is an infeasible task even for a small number of strings, many practical (heuristic) methods have been developed that provide good, but not optimal solutions. Most of these methods produce solutions of acceptable quality, yet, little is usually known about the relationship between the SP-scores of the alignments produced by these heuristic algorithms and the optimal SP-score. There are, however, several methods where such analysis is possible. These methods, called bounded-error approximation methods, produce alignments whose SP-score is guaranteed to be within some constant of optimal and complete their task in polynomial worst-case time. The earliest known bounded-error approximation algorithm for the multiple sequence alignment is Gusfield’s center star algorithm[14]. This sequence alignment algorithm is employed to produce a good initial alignment in the genetic framework.

2.2 Algorithm Outline

The following is the brief description of the Gusfield’s algorithm:

1. Given \( k \) sequences \( x_1, x_2, \ldots, x_k \), find the sequence \( x^* \) (called star center), that optimizes the sum \( \sum_{i=1}^{k} D(x^*, x_k) \), where \( D(x_i, x_j) \) is the score of the optimal pairwise alignment of the sequences \( x_i \) and \( x_j \). A pairwise alignment between \( x_i \) and \( x_j \) is scored by summing the costs of aligning their characters from
the scoring matrix $M$. The optimal pairwise alignment can be computed via
dynamic programming in $O(|x_i| \cdot |x_j|)$ steps.

2. Once the star center is found, the $(k - 1)$ remaining sequences are aligned
optimally to the $x^*$ one at a time (using the dynamic programming approach).
More specifically, $x^*$ is first aligned with, say $x_1$. Then $x^*$ is aligned with $x_2$.
If any new spaces are introduced in $x^*$, these spaces are also introduced into
the current version of $x_1$. This process continues until all sequences are aligned.
For an example, see Figure 2.1.

As mentioned by Gusfield [14], the center star algorithm produces an alignment
that is consistent with a tree (see Figure 2.2). In this case, the tree is a depth one
tree with a root $x^*$.

The center star algorithm achieves the approximation ratio of $2 - 2/k$. If the
number of strings is 3, then the guaranteed upper bound is 4/3. The upper bound
increases with the number of strings, but always stays below 2. It must be noted
that there are two conditions that must hold in order to achieve this approximation
ratio. First, the cost of aligning a space with a space must be zero. This is necessary
to preserve the current SP-score when new spaces are introduced into the alignment.
Second, the scoring matrix must satisfy the triangle inequality. Unfortunately, not all matrices used in computational biology satisfy the triangle inequality. Two such matrices are used in our genetic algorithm, as will later be shown.

2.3 Algorithm Implementation

Gusfield’s algorithm is used extensively in our genetic framework. Here we discuss an implementation of Gusfield’s algorithm. For a detailed analysis of Gusfield’s algorithm, see Gusfield’s recent book on computational biology [15].

Our simple implementation of Gusfield’s algorithm consists of the following C function. This function is called after the input sequences and the scoring matrix are read from the input files and placed into dynamic arrays. Several global variables are used in the function. The global array char **str contains input sequences. The global variable int k contains the number of input sequences. In addition, the external function Align() implements the dynamic programming algorithm for optimal alignment of the two sequences. The listing of the Align() function has not been included here. See Appendix B for the listings of this and other functions. The comments in the following code fragment explain our implementation of Gusfield’s center star algorithm.
/*   Gusfield’s algorithm implementation.  
****************************************/  
void gusfield()  
{  
  int **d,*data,x,y,z,sum,maxsum,sc,xk,yk,xx;  
  char *temp;  
  
  /* allocate memory and create a dynamic 2-dimensional array */  
data = malloc(k * k * sizeof(int));  
d = malloc(k * sizeof(int *));  
for (z=0; z<k; z++)  
d[z] = data + (z*k);  
  
  /* optimally align all possible pairs of strings */  
  /* put resulting scores in a matrix */  
for (y=0; y<k; y++)  
  for (x=0; x<k; x++)  
    if (x != y)  
      d[x][y] = align(x,y);  
    else  
      d[x][y] = 0;  
  
  /* find x* - star center */  
  /* for each string y, the sum of scores of aligning y with */  
  /* other strings is computed, the maximum sum is recorded */  
maxsum = -BIG;  
for (y=0; y<k; y++) {  
  sum = 0;  
  for (x=0; x<k; x++)  
    sum += d[x][y];  
  if (sum > maxsum) {  
    maxsum = sum;  
    sc = y;  /* sc is a star center */  
  }  
}  
}
/* Create an alignment: */
/* for all k sequences.. */
for (x=0; x<k; x++) {
    /* ..built optimal alignment with the x* */
    if (x != sc) {
        align(sc, x);
        /* if any new spaces introduced in x*.. */
        if (strlen(str[sc]) != strlen(sx)) { xx = 0;
            /* insert spaces into all sequences */
            /* that are already in the alignment */
            for (xk=0; xk<strlen(str[sc]); xk++, xx++)
                if (*(str[sc]+xk) != *(sx+xx)) {
                    for (yk=0; yk<x; yk++)
                        if (yk != sc) {
                            /* insert space into current string */
                            temp = malloc(strlen(str[yk])+1);
                            strcpy(temp, str[yk]);
                            *(str[yk]+xk) = ' ';
                            *(str[yk]+xk+1) = 0;
                            strcat(str[yk], temp+xk);
                            free(temp);
                        }
                    xx++;
                }
        }
        strcpy(str[sc], sx);
    }
}

/* output the alignment on a screen */
cleanup();
output();
}
3. GAP 0-1 ALIGNMENT PROBLEM

3.1 Description

As mentioned in the introduction, Gap 0-1 alignment problem is a very restricted version of the multiple sequence alignment problem where only single gaps are allowed at the beginnings or endings of the strings. For an example, see Figure 3.1.

\[
\begin{align*}
X_- \text{ or } X \quad \text{or} \quad X_- \text{ or } X \\
Y_- \quad \text{or} \quad Y_- \quad \text{or} \quad Y_-
\end{align*}
\]

Figure 3.1 Given two strings \(X\) and \(Y\), there are four possible Gap 0-1 alignments. Note that the first and second alignments have identical SP-Scores.

Given an alphabet \(\Sigma\), and \(k\) strings \(x_1, x_2, ..., x_k\) of length \(n\) over \(\Sigma\) and a \(|\Sigma| + 1 \times |\Sigma| + 1\) symmetric scoring matrix \(M\) (includes the "gap" character), the Gap 0-1 alignment problem is to find a Gap 0-1 alignment \(x'_1, x'_2, ..., x'_k\) of \(k\) strings that minimizes (or maximizes) the SP-score

\[
\sum_{i<j} d_M(x'_i, x'_j),
\]

where

\[
d_M(x'_i, x'_j) = \sum_{k=1}^{[x'_i]} M_{x'_i[k], x'_j[k]}.
\]
The distance $d_M(x'_i, x'_j)$ is the sum of the costs of the pairwise alignment of all characters of $x'_i$ with $x'_j$. The individual alignment score of aligning one character with another is given by the scoring matrix $M$.

Here both maximization and minimization versions of Gap 0-1 alignment problem are examined. Although the Gap 0-1 alignment problem is a greatly simplified version of the multiple sequence alignment problem, it is still intractable. As shown by Just[22], the Gap 0-1 alignment problem is NP-hard, and even MAX-SNP-hard. That is, unless $P = NP$, Gap 0-1 alignment problem cannot be approximated to the ratio of 1.0624 in polynomial time [20].

3.2 Using the Gap 0-1 For Improving Existing Alignments

Gap 0-1 alignment is described in detail here because it is used as a subproblem in the genetic algorithm to iteratively improve alignments. As shown here, optimal Gap 0-1 alignments can be used to improve alignments created using Gusfield’s algorithm. For example, see Figure 3.2.

Consider the minimization problem of the multiple sequence alignment problem. Let the scoring matrix $M$ contain all ones except for zeroes along the diagonal, as given in (3.3). With this scoring matrix, the cost of aligning two matched characters is zero, while aligning any two different characters has a cost of one.

$$M = \begin{pmatrix}
\Delta & A & G & C & T \\
\Delta & 0 & 1 & 1 & 1 \\
A & 1 & 0 & 1 & 1 \\
G & 1 & 1 & 0 & 1 \\
C & 1 & 1 & 1 & 0 \\
T & 1 & 1 & 1 & 0 \\
\end{pmatrix}$$

(3.3)
Consider the multiple alignment of four DNA sequences:

\[ x_1 = \text{ATAT}, \quad x_2 = \text{TAT}, \quad x_3 = \text{ATTTAA}, \quad \text{and} \quad x_4 = \text{TTTAA}. \]

Using dynamic programming, it can be shown that the SP-score for the optimal alignment of these four sequences is 16. To begin with, build an initial alignment that is close to the optimal by applying Gusfield’s center star approximation algorithm. This produces an initial alignment shown as Step 1 in the Figure 3.2. This alignment has a SP-score of 20, which is within \( 2 - 2/k = 3/2 \) of optimal for \( k = 4 \) sequences. This initial alignment can be further improved by applying a sequence of optimal Gap 0-1 alignments. (See Figure 3.2, steps 2-5.) At step 2, a Gap 0-1 alignment is performed at the third position in the alignment. It must be noted that the choice of where to place the space is arbitrary. The problem is essentially the same if the single space is inserted at the \( i \)th spot and shifted to the left. Step 2 produces an alignment with SP-score 19 and an entire column of spaces. This column of spaces can be deleted without affecting the SP-score, as shown in the step 3. Next, an optimal Gap 0-1 alignment can be applied again at the first position in the alignment, as was done in step 4. This produces an alignment with SP-score 16, which is optimal. Again, the column of spaces is produced and can be deleted without changing the SP-score as in step 5.
As this example illustrates, Gap 0-1 alignments can be used as a powerful tool for improving multiple sequence alignments. However, it must be noted that if the length of sequences increases, the chance that a Gap 0-1 alignment improves the overall alignment seems unlikely. It appears unlikely that an initial close-to-optimal alignment of long sequences can be improved by inserting single spaces in the beginnings or endings of the strings. To overcome this limitation, a Gap 0-1 alignment can be performed on a small window, as shown in Figure 3.3. Inside such a small window, a Gap 0-1 alignment seems to have a much greater chance of improving the alignment. This approach will be described later as a part of the evolutionary framework.

![Figure 3.3](image.png)

Figure 3.3 An example of a multiple alignment of long sequences where a small window of size 3 is chosen to apply a local Gap 0-1 improvement.

3.3 Fixed Parameter Tractable Algorithm for Gap 0-1

Although the Gap 0-1 alignment problem is NP-hard, the optimal solution can be computed in polynomial time if the number of strings is fixed. As shown in the following paragraph, Gap 0-1 alignment is computable in $O(2^k k^2 + 3k^2 n)$ steps. Hence, Gap 0-1 alignment is fixed parameter tractable. That is, if we fix the number of strings $k$, the running time will be linear. When $k$ is not fixed, it is exponential.

The fixed parameter tractable algorithm proceeds as follows. Initially, for each pair $(i, j)$ of strings, the SP-scores for the four possible alignments of the pair are precomputed. Since $d_M(\Delta x_i, \Delta x_j) = d_M(x_i \Delta, x_j \Delta)$, it requires $3n$ steps for each pair,
where \( n \) is the length of each string. For all different pairs of strings this precomputation phase takes \( O(3k^2n) \) steps. Once these values are computed, it suffices to find the maximum over all \( 2^k \) possible alignments. This step takes \( 2^k k^2 \) steps. Thus, the total running time from these two steps is \( O(2^k k^2 + 3k^2 n) \).

If the number of sequences is small (10 or less), this exact algorithm runs reasonably fast. However, as the number of sequences increases, this approach becomes infeasible. To avoid this, an approximation algorithm can be used for larger values of \( k \). One such algorithm, developed recently by Juedes [20], is a randomized approximation algorithm that uses techniques pioneered by Goemans and Williamson [12] to obtain good approximated solutions via semidefinite programming.

### 3.4 Semidefinite Programming Algorithm for Gap 0-1

As mentioned earlier, the lower bound approximation ratio for the Gap 0-1 alignment problem is \( 17/16 = 1.0624 \) [22]. While there is no known algorithm that achieves this lower bound, Juedes [20] developed a polynomial-time randomized approximation algorithm that achieves an approximation ratio of 1.2562. However, this approximation ratio can only be achieved under the following two conditions. First, all the cost values in the scoring matrix \( M \) must be positive, and second, the algorithm must consider the maximization version of the Gap 0-1 alignment problem. Thus, the question about the approximation complexity of the minimization version of the Gap 0-1 alignment problem remains open. This approximation algorithm uses quadratic integer and semidefinite programming techniques. The approach is based on Goemans and Williamson’s [12] approximation algorithms for MAX-CUT and MAX-SAT. In the following subsections, a detailed description of Juedes’s approximation algorithm and its implementation using the CSDP library [4] are presented.
3.4.1 Constructing Quadratic Integer Programming Problem

Although the semidefinite programming algorithm for the Gap 0-1 problem runs in polynomial time, it is composed from several complex steps, each of which deserves a separate discussion. First, we compute the values

1. \( a_{i,j} = d_M(\Delta x_i, \Delta x_j) = d_M(x_i \Delta, x_j \Delta) \)
2. \( b_{i,j} = d_M(x_i \Delta, \Delta x_j) \)
3. \( c_{i,j} = d_M(\Delta x_i, x_j \Delta) \)

The next step of the algorithm can be described in terms of the quadratic integer programming. Let \( x'_1, x'_2, \ldots, x'_k \) be a Gap 0-1 alignment of \( k \) sequences. These \( k \) sequences can be represented by the set of variables \( z_1, z_2, \ldots, z_k \), such that \( z_i \in \{-1, 1\} \), and the extra variable \( z_0 \) which determines the meaning of 1 and -1. That is, if \( z_i \cdot z_0 = 1 \), the space is inserted at the beginning of the string, i.e. \( x'_i = \Delta x_i \). Otherwise, if \( z_i \cdot z_0 = -1 \) the space is inserted at the end of the string, i.e \( x'_i = x_i \Delta \). It must be noted that there are exactly two settings of the variables \( z_0, z_1, \ldots, z_k \) for the set of sequences \( k \). As observed by Goemans and Williamson,

\[
\begin{align*}
\text{if } z_i = z_j &\Rightarrow (1 + z_i \cdot z_j)/2 = 1, \\
\text{if } z_i \neq z_j &\Rightarrow (1 + z_i \cdot z_j)/2 = 0, \\
\text{if } z_i = z_0 \neq z_j &\Rightarrow (1 + z_i \cdot z_0 - z_j \cdot z_0 - z_i \cdot z_j)/4 = 1, \text{ and finally} \\
(1 + z_i \cdot z_0 - z_j \cdot z_0 - z_i \cdot z_j)/4 &\Rightarrow 0 \text{ otherwise.}
\end{align*}
\]

Using these observations, the following quadratic integer program is constructed:

\[
\sum_{i<j} a_{i,j} \frac{1 + z_i \cdot z_j}{2} + b_{i,j} \frac{1 + z_i \cdot z_0 - z_j \cdot z_0 - z_i \cdot z_j}{4} + c_{i,j} \frac{1 + z_j \cdot z_0 - z_i \cdot z_0 - z_i \cdot z_j}{4}.
\]

Notice that the maximum Gap 0-1 alignment of \( k \) sequences is achieved when this quadratic integer program is maximized. This quadratic integer program will be used in the construction of the randomized approximation algorithm.
3.4.2 Constructing Semidefinite Programming Problem

The quadratic integer program (3.4) can now be used to construct a semidefinite program in the same way that Goemans and Williamson [12] did for MAX-CUT.

Because the problem of solving (3.4) is NP-complete, it cannot be solved exactly in polynomial time. In order to solve it in polynomial time, some of the constraints of (3.4) must be relaxed. The problem (3.4) can be viewed as the 1-dimensional projection of the following relaxed maximization problem:

\[
\sum_{i<j} a_{i,j} \frac{1 + v_i \cdot v_j}{2} + b_{i,j} \frac{1 + v_i \cdot v_0 - v_j \cdot v_0 - v_i \cdot v_j}{4} + c_{i,j} \frac{1 + v_j \cdot v_0 - v_i \cdot v_0 - v_i \cdot v_j}{4}, \tag{3.5}
\]

where each \( v_i \) is a \((k + 1)\)-dimensional vector in the unit Euclidean norm. This relaxed problem can be used instead of (3.4) since its optimal value is at least as large as the optimal value of (3.4). The values \( v_i \) can be chosen in such a way that the 1-dimensional projection of (3.5) gives the solution to the quadratic integer programming problem (3.4). To solve the relaxed problem, we solve the semidefinite program

\[
\sum_{i<j} a_{i,j} \frac{1 + y_{ij}}{2} + b_{i,j} \frac{1 + y_{i0} - y_{j0} - y_{ij}}{4} + c_{i,j} \frac{1 + y_{j0} - y_{i0} - y_{ij}}{4}, \tag{3.6}
\]

subject to \( y_{ii} = 1 \), \( y_{ij} = v_i \cdot v_j \), and the matrix \( Y = \{y_{ij}\} \) is positive semidefinite. The constraint that \( y_{ii} \) is always equal to 1 guarantees that each \( v_i \) is a unit vector. Maximizing this semidefinite program results in a symmetric positive semidefinite matrix \( Y = \{y_{ij}\} \). A matrix \( Y \) is positive semidefinite if and only if there exists a matrix \( X \) such that \( X^T X = Y \). The matrix \( Y \) can be factored into \( X^T X \) using an incomplete Cholesky factorization. The final solution can be obtained using randomization techniques which will be described later. Using the semidefinite program (3.6), an approximation algorithm for the Gap 0-1 problem can be constructed. In the next section, an algorithm that follows the template given by Goemans and Williamson [12] is described.
3.4.3 Randomized Approximation Algorithm

The randomized algorithm for Gap 0-1 alignment works as follows:

1. Construct the semidefinite program (3.6) for the problem. This requires computing the matrices $a, b$ and $c$, which is also the first step of the fixed parameter tractable algorithm. This takes $O(3k^2n)$ steps.

2. Solve the semidefinite program. Several methods exist for solving semidefinite programs. The CSDP library written by Borchers [4] implements one such method.

3. Factor the resulting symmetric positive semidefinite matrix $Y$ using an incomplete Cholesky factorization method. The resulting matrix $X$ contains column vectors $v_0, v_1, \ldots, v_k$. The factoring algorithm runs in $O(k^3)$ steps.

4. Generate a random $(k + 1)$ unit vector $r$. This can be done using techniques of Knuth [23] and takes $O(k)$ steps.

5. Set $z_i = 1$ if $\text{sgn}(v_i \cdot r) \geq 0$ and $z_i = -1$ if $\text{sgn}(v_i \cdot r) < 0$. It must be noted that the first column vector $v_0$ defines the value of $z_0$, and $z_0$ defines the meaning of the sign for the variables $z_1, z_2, \ldots, z_k$.

6. Set $x'_i = \Delta x_i$ if $z_i \cdot z_0 = 1$ and $x'_i = x_i \Delta$ if $z_i \cdot z_0 = -1$. (Steps 5 and 6 run in time $O(k)$.)

7. To improve the performance of this randomized algorithm, the alignment created in step 6 is recorded and then steps 4, 5 and 6 are repeated several times. (In practice, ten iterations appear to be enough for our purposes.) Finally, the best overall alignment is selected.

It can be shown that the expected value for the resulting solution is within 1.2561709 of optimal [20].
3.4.4 Implementation Using CSDP Library

CSDP is a software library written in C by Brian Borchers [4] for solving semidefinite programming problems. CSDP is a perfect choice to be used in the Gap 0-1 approximation algorithm because CSDP is not a stand-alone application. CSDP is a library containing a set of routines that can be called from a user program. CSDP is a very well written package. It is both flexible and powerful. Moreover, it is acceptably fast for our purposes.

CSDP solves semidefinite programming problems of the form:

$$\begin{align*}
\text{maximize} & \quad tr(CX), \\
\text{subject to} & \quad A(X) = a, \\
& \quad B(X) \leq b, \quad \text{and} \\
& \quad X \succeq 0,
\end{align*}$$

where $X \succeq 0$ indicates that $X$ is positive semidefinite.

To use the library, the semidefinite programming problem from the Gap 0-1 approximation algorithm must be restated in this form. This form can handle more complex semidefinite programming problems, but our case uses a simplified formulation. The constraints $A(X) = a$ and $B(X) \leq b$ can be largely omitted, though some values must be used to guarantee that the original constraints are satisfied. The primary data that must be generated for this form are the values for the matrix $C$.

Consider the semidefinite program (3.6). It can be rewritten in the following form:

$$\sum_{i<j} \frac{a_{ij}}{2} + \frac{a_{ij} y_{ij}}{2} + \frac{b_{ij} y_{ij}}{4} + \frac{b_{ij} y_{ja}}{4} - \frac{b_{ij} y_{ja}}{4} + \frac{c_{ij}}{4} - \frac{c_{ij} y_{ij}}{4} - \frac{c_{ij} y_{ja}}{4} + \frac{c_{ij} y_{ja}}{4}. \quad (3.8)$$
Now, the values for the matrix C can be computed using the following equations:

\[ C_{00} = \sum_{i<j} \frac{a_{ij}}{2} + \frac{b_{ij}}{4} + \frac{c_{ij}}{4}, \]  
\[ (3.9) \]

\[ C_{ij} = \frac{a_{ij}}{2} - \frac{b_{ij}}{4} - \frac{c_{ij}}{4}, \]  
\[ (3.10) \]

\[ C_{i0} = C_{0j} = \sum_{k<j} \left( -\frac{b_{kj}}{4} + \frac{c_{kj}}{4} \right) + \sum_{k>j} \left( -\frac{b_{kj}}{4} - \frac{c_{kj}}{4} \right), \]  
\[ (3.11) \]

The following is a short description of steps that the program takes to find an approximate solution for the Gap 0-1 alignment problem:

1. Read the scoring matrix \( M \) and the sequences from the file.
2. Compute matrices \( a, b \) and \( c \).
3. Compute the values for the matrix \( C \).
4. Generate parameters for the CSDP routine \( \text{sdp}() \).
5. Call the function \( \text{sdp}() \) that finds the solution to the semidefinite program.
6. Factor the result using the incomplete Cholesky factorization method.
7. Generate a random vector \( r \) (repeat it 10 times).
8. Find \( \text{sgn}(v_i \cdot r) \) which defines where to put space in the sequence \( x_i \).
9. Output the resulting Gap 0-1 multiple sequence alignment.
4. GENETIC ALGORITHM

4.1 Proposed Approach

As already mentioned in the introduction, the multiple sequence alignment problem is a natural candidate for evolutionary search techniques. The technique proposed here combines global and local approximation algorithms with evolutionary computation, which, in our case, is better described as a genetic algorithm [6, 25].

The basic idea is to simulate a population of individuals. As in nature, individuals grow, mutate, reproduce, and vie for resources. From generation to generation, the population evolves and the individuals usually become stronger. In this case each individual is simply a sequence alignment of the set of input sequences.

The "growth" of individuals is simulated by performing local improvements of sequence alignments (using Gap 0-1 alignment algorithm). The fitness of each individual is the SP-score of the sequence alignment computed after the individual has been allowed to "grow." As in nature, stronger individuals are involved in sexual reproduction, while others perish because their fitness is too low. Some percentage of individuals can mutate. Such mutations occur at a low and relatively constant rate. Hence, mutations correspond to insertions of spaces into sequences at random spots.

The reproduction phase selects two stronger individuals and creates two children. This is done by breaking the "parent" sequence alignments into two roughly equal parts and, then, swapping the left and right halves of the alignments. Since both parents have high fitness, this process should also produce individuals of high fitness.
To summarize, the evolutionary simulation algorithm can be described by the following three steps:

1. Seed an initial population of individuals (alignments) with "good" approximate alignment algorithm (e.g., Gusfield’s center star algorithm).

2. Locally improve alignments (if possible) with exact or approximate algorithms for restricted versions of multiple sequence alignment (e.g., Gap 0-1 alignment algorithm).

3. Allow the population of alignments to evolve through many generations. As population evolves, the sequence alignment hopefully improves.

The two near-optimal sequence alignment algorithms described in detail in the previous sections can now be utilized as parts of the genetic algorithm. Gusfield’s center star algorithm is used to create the initial population of individuals. The Gap 0-1 fixed parameter tractable algorithm (or the semidefinite programming algorithm) is used for local the improvements of individuals.
4.2 The Genetic Algorithm Description

In this section, the internal workings of the genetic algorithm are described. The algorithm performs the following steps.

1. The initial alignment of \( k \) input sequences is created by using Gusfield’s center star algorithm. As mentioned earlier, this algorithm can produce alignments that deviate significantly from optimal. However, Gusfield’s algorithm is reasonably simple and fast. Thus it is the ideal choice for the "starting point" of the genetic algorithm. Gusfield’s algorithm is used only in this step. However, its job is very important – to produce an initial alignment of a good quality. In the future steps, the local improvement algorithm would not be able to work on a "raw" data, if the initial alignment was poor. Fortunately, Gusfield’s algorithm produces alignments of acceptable quality.

2. The initial population of \( n \) individuals is built by making \( n - 1 \) copies of the individual created in the previous step. At this point all individuals are identical. To allow for individual uniqueness, small mutations are randomly introduced into the other \( n - 1 \) alignments. Mutations are not introduced into the original individual. A single mutation is performed by randomly selecting one of the sequences from the alignment and a location in that sequence and inserting a gap at that location into the sequence.

3. The population of \( n \) individuals is allowed to evolve through \( m \) generations. During each generation, the individuals with the best fitness (SP-score) are recorded. That is, SP-scores are computed for each individual in the current generation. Then, the maximum among these SP-scores and the SP-score of the best individual from the previous generation is found and the corresponding individual is recorded. During each generation individuals grow, fight, mutate
and sexually reproduce to create each subsequent generation via the following steps.

3.1 The individuals are allowed to grow. This step attempts to locally improve the sequence alignment. The $n$ individuals are locally optimized via Gap 0-1 alignments. (For an example, see Figure 4.1.) Given an individual alignment and the column position inside the alignment, an optimal Gap 0-1 alignment is performed on a window of characters formed from the $t$ contiguous columns to the right of the current column. In the current implementation, four local improvement passes are performed for each individual alignment at each generation. The first two passes have windows of size $t = 2$ and $t = 3$, respectively. These windows are moved from left to right by incrementing the starting column position, one at a time. The only difference for the second two passes is that they move from right to left. This scanning pattern, developed experimentally, gives better average results than other patterns.

![Figure 4.1 Local improvement using Gap 0-1 alignment. a) The characters V, K, Q and A are about to be moved. b) When the pass is finished, all characters are in the right columns.](image)

3.2 Each individual $x_i$ is assigned a fitness $f(x_i)$, which is the SP-score for the alignment. A fraction $p_1$ of the $n$ individuals ($n \cdot p_1$) with the lowest fitness perish. This step essentially allows the removal of poor alignments, while maintaining better ones for future generations.
3.3 Sexual reproduction is performed to replace \((n \cdot p_1)\) individuals that perished. Again, just as in nature, individuals with the higher fitness have a greater chance to perform sexual reproduction. If \(n'\) individuals remain after step 3.2, then a pair of individuals \((x_i, x_j)\) is picked for sexual reproduction using the roulette wheel selection, i.e., randomly with probability

\[
p(x_i, x_j) = \frac{f(x_i) + f(x_j)}{S}
\]

where

\[
S = \sum_{i<j} f(x_i) + f(x_j) = (n' - 1) \sum_i f(x_i).
\]

Sexual reproduction is performed as illustrated in Figure 4.2. We break two "parent" sequence alignments into two roughly equal halves by selecting positions within each sequence at random. Two new "child" alignments are then created by interchanging the left and right halves of the "parent" alignments. It must be noted that the alignments given by the left and right halves are preserved. To complete this, some extra spaces may need to be inserted between halves, as in Figure 4.2.

3.4 Random mutations are introduced into the sequences at a constant rate of \(p_2\). That is, with probability \(p_2\), each individual will have a mutation during each generation. A mutation is performed by inserting a constant number of spaces into the alignment at random spots. In the current implementation, two spaces are inserted into a single alignment if the mutation occurs. It must be noted that mutation and sexual reproduction can make sequence alignments worse. However, without mutations, individuals may improve at each generation, but tend to become very similar to each other. A whole population of identical alignments will no longer improve in the absence of mutation. Intuitively, the same would probably happen
in nature if mutations did not occur. In terms of sequence alignment problems, by slightly changing alignments with each generation, there are more chances for new local improvements to occur in future generations.

As you can see from the genetic algorithm, there are four parameters that can be modified to affect the performance of the algorithm:

- $n$ - the number of individuals,
- $m$ - the number of generations,
- $p_1$ - the fraction of individuals that perish, and
- $p_2$ - the mutation probability.
5. EXPERIMENTAL RESULTS

5.1 Test Suite

5.1.1 COG Database

The genetic algorithm proposed herein was implemented in C (for the Sun SPARC and Linux operating systems) and tested on a collection of 17 *clusters of orthologous groups of proteins* (COGs) [27]. Each COG is basically a set of existing biologically related protein sequences. The COG database is freely available and can be found on the Internet (see http://www.ncbi.nlm.nih.gov/). Table 5.1 shows the identifying numbers and names of COGs used in the tests. The size of the COGs used in the tests ranged from 3 to 9 proteins, and the average length of the sequences varied from 71 to 494. Table 5.2 gives the number of sequences per COG, the length of a longest sequence, and the average length of sequences in each of the 17 COGs.

5.1.2 Matrices Used For The Tests

When aligning proteins with the alphabet of size $|\Sigma| = 20$, it is not enough to just try to align matching characters and avoid mismatches. Instead, the goal is to try to achieve biologically meaningful alignments. In life, amino acids of a protein have very complex relationships. Unfortunately, it is not known exactly how one amino acid relates to the others in the sense of "evolutionary change." There has been extensive research done to define these relationships [7, 16].

Since there are 20 amino acids in the alphabet for the proteins, there are 210 distinct pairs of characters $(i, j)$, each of which corresponds to a substitution of character $i$ with $j$. The target probabilities for the matrix are an estimate of frequencies with
Table 5.1 Test Suite.

<table>
<thead>
<tr>
<th>COG #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2178</td>
<td>Translin (RNA-binding protein, recombination hotspot binding in eukaryotes)</td>
</tr>
<tr>
<td>1983</td>
<td>Putative stress-responsive transcriptional regulator PspC</td>
</tr>
<tr>
<td>1549</td>
<td>Queuine tRNA-ribosyltransferases, contain PUA domain</td>
</tr>
<tr>
<td>1603</td>
<td>Archaeal homologs of ribonuclease P subunit Rpp30</td>
</tr>
<tr>
<td>2157</td>
<td>Ribosomal protein LX (HL32)</td>
</tr>
<tr>
<td>1476</td>
<td>Predicted transcriptional regulator</td>
</tr>
<tr>
<td>1976</td>
<td>Eukaryotic translation initiation factor</td>
</tr>
<tr>
<td>2097</td>
<td>Ribosomal protein L31E</td>
</tr>
<tr>
<td>1510</td>
<td>Predicted transcriptional regulator</td>
</tr>
<tr>
<td>1761</td>
<td>DNA-directed RNA polymerase</td>
</tr>
<tr>
<td>1515</td>
<td>Deoxyinosine 3’endonuclease</td>
</tr>
<tr>
<td>1758</td>
<td>RNA polymerase-associated protein RpoZ</td>
</tr>
<tr>
<td>0565</td>
<td>rRNA methylase</td>
</tr>
<tr>
<td>2007</td>
<td>Ribosomal protein S8E</td>
</tr>
<tr>
<td>1514</td>
<td>2’-5’ RNA ligase and related enzymes</td>
</tr>
<tr>
<td>0219</td>
<td>Predicted rRNA methylase (SpoU class)</td>
</tr>
<tr>
<td>2003</td>
<td>DNA repair proteins, RadC family</td>
</tr>
</tbody>
</table>
Table 5.2 COG Sizes.

<table>
<thead>
<tr>
<th>COG #</th>
<th># of Seq.</th>
<th>Max. Length</th>
<th>Avg. Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>2178</td>
<td>3</td>
<td>222</td>
<td>211</td>
</tr>
<tr>
<td>1983</td>
<td>4</td>
<td>158</td>
<td>114</td>
</tr>
<tr>
<td>1549</td>
<td>4</td>
<td>570</td>
<td>494</td>
</tr>
<tr>
<td>1603</td>
<td>4</td>
<td>245</td>
<td>222</td>
</tr>
<tr>
<td>2157</td>
<td>4</td>
<td>78</td>
<td>72</td>
</tr>
<tr>
<td>1476</td>
<td>5</td>
<td>79</td>
<td>71</td>
</tr>
<tr>
<td>1976</td>
<td>5</td>
<td>245</td>
<td>227</td>
</tr>
<tr>
<td>2097</td>
<td>6</td>
<td>113</td>
<td>96</td>
</tr>
<tr>
<td>1510</td>
<td>6</td>
<td>185</td>
<td>170</td>
</tr>
<tr>
<td>1761</td>
<td>6</td>
<td>142</td>
<td>105</td>
</tr>
<tr>
<td>1515</td>
<td>6</td>
<td>238</td>
<td>218</td>
</tr>
<tr>
<td>1758</td>
<td>7</td>
<td>127</td>
<td>89</td>
</tr>
<tr>
<td>0565</td>
<td>7</td>
<td>246</td>
<td>236</td>
</tr>
<tr>
<td>2007</td>
<td>7</td>
<td>261</td>
<td>167</td>
</tr>
<tr>
<td>1514</td>
<td>8</td>
<td>188</td>
<td>182</td>
</tr>
<tr>
<td>0219</td>
<td>9</td>
<td>166</td>
<td>158</td>
</tr>
<tr>
<td>2003</td>
<td>9</td>
<td>243</td>
<td>206</td>
</tr>
</tbody>
</table>
which one can find a substitution \((i, j)\) in a database of proteins \([3]\). This approach gave rise to a family of scoring matrices, called BLOSUM \([16]\). The BLOSUM62 matrix given in Figure 5.1 was used in the current implementation of the genetic algorithm.

Dayhoff in (1978) \([7]\) used a stochastic process to model the evolution of proteins. At each discrete time instance, a symbol (an amino acid) has a certain fixed probability of substituting another symbol. A unit of measure of evolutionary change was defined as a \textit{Point Accepted Mutation} (PAM). Given evolutionary distances, the corresponding scoring matrices \(M\) were computed. Such matrices are called \textit{PAM} matrices. The most commonly used PAM matrix is the PAM250 matrix given in

\[
\begin{array}{cccccccccccccccc}
A & R & N & D & C & Q & E & G & H & I & L & K & M & F & P & S & T & W & Y & V \\
Q & -1 & 1 & 0 & 0 & -3 & 5 & & & & & & & & & & & & & & & & & & \\
\end{array}
\]

Figure 5.1 BLOSUM62 matrix
Figure 5.2 [7]. Notice that this matrix (as well as BLOSUM62) is designed for the maximization version of the sequence alignment problem. PAM250 matrix was also used in the proposed genetic algorithm implementation and our algorithm produced similar results to those with the BLOSUM62 matrix.

![PAM250 Matrix](image)

For each COG, we computed the SP-score given by the BLOSUM62 and PAM250 scoring matrices. The alignment of a character with a space was scored with a value -4 (causing a linear gap penalty). The gap opening penalty was assumed to be zero.
5.2 Test Results

For the first 10 COGs, a series of tests were performed using different parameter settings. The purpose of these tests was to find an optimal set of parameters for the genetic algorithm. Tables 5.3, 5.4 and 5.5 list the resulting sums of SP-scores of all 10 COGs for all parameter settings. The best performance was achieved when \( n = 15, m = 16, p_1 = 0.75, p_2 = 0.1 \) (the sum of SP-scores is 7069) and when \( n = 15, m = 24, p_1 = 0.75, p_2 = 0.1 \) (the sum of SP-scores is 7072). Since in the second case (with 24 individuals) the performance increase was insignificant, the setting with 16 individuals was chosen for the final tests.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( m )</th>
<th>( p_2 )</th>
<th>( 0.1 )</th>
<th>( 0.25 )</th>
<th>( 0.5 )</th>
<th>( 0.75 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>0.1</td>
<td>6385</td>
<td>6370</td>
<td>6408</td>
<td>6424</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.25</td>
<td>6388</td>
<td>6300</td>
<td>6257</td>
<td>6302</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5</td>
<td>6194</td>
<td>6290</td>
<td>6317</td>
<td>6193</td>
</tr>
<tr>
<td>16</td>
<td>0.1</td>
<td>6473</td>
<td>6375</td>
<td>6418</td>
<td>6425</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>6388</td>
<td>6300</td>
<td>6287</td>
<td>6302</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>6178</td>
<td>6310</td>
<td>6317</td>
<td>6209</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0.1</td>
<td>6473</td>
<td>6377</td>
<td>6423</td>
<td>6427</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>6388</td>
<td>6312</td>
<td>6291</td>
<td>6335</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>6197</td>
<td>6310</td>
<td>6317</td>
<td>6201</td>
<td></td>
</tr>
</tbody>
</table>

\(^1n\) is a number of generations, \( m \) is a number of individuals, \( p_1 \) is a fraction of individuals that perish at each generation, \( p_2 \) is a probability of mutation.
Table 5.4 Parameter settings and the sum of SP-Scores for 10 COGs ($n = 10$)

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$m$</th>
<th>$p_2$</th>
<th>0.1</th>
<th>0.25</th>
<th>0.5</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>8</td>
<td>0.1</td>
<td>6473</td>
<td>6375</td>
<td>6418</td>
<td>6424</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6388</td>
<td>6300</td>
<td>6287</td>
<td>6302</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6197</td>
<td>6310</td>
<td>6317</td>
<td>6293</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0.1</td>
<td>6234</td>
<td>6697</td>
<td>6518</td>
<td>6925</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6246</td>
<td>6500</td>
<td>6687</td>
<td>6802</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6388</td>
<td>6372</td>
<td>6617</td>
<td>6609</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>0.1</td>
<td>6678</td>
<td>6713</td>
<td>6823</td>
<td>6927</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6592</td>
<td>6642</td>
<td>6791</td>
<td>6835</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6440</td>
<td>6624</td>
<td>6517</td>
<td>6811</td>
</tr>
</tbody>
</table>

Table 5.5 Parameter settings and the sum of SP-Scores for 10 COGs ($n = 15$)

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$m$</th>
<th>$p_2$</th>
<th>0.1</th>
<th>0.25</th>
<th>0.5</th>
<th>0.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>8</td>
<td>0.1</td>
<td>6473</td>
<td>6377</td>
<td>6423</td>
<td>6427</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6388</td>
<td>6300</td>
<td>6287</td>
<td>6335</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6197</td>
<td>6310</td>
<td>6317</td>
<td>6201</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0.1</td>
<td>6377</td>
<td>6797</td>
<td>6518</td>
<td>7069</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6312</td>
<td>6500</td>
<td>6687</td>
<td>6841</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6310</td>
<td>6372</td>
<td>6617</td>
<td>6738</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>0.1</td>
<td>6678</td>
<td>6713</td>
<td>6823</td>
<td>7072</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.25</td>
<td>6592</td>
<td>6642</td>
<td>6791</td>
<td>6913</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
<td>6440</td>
<td>6624</td>
<td>6517</td>
<td>6845</td>
</tr>
</tbody>
</table>
Next, our genetic algorithm was applied to each of the 17 COGs with the above parameter settings. We compared the resulting multiple sequence alignments with alignments generated by CLUSTALW for each of the 17 COGs as given in the COG database. Both the original SP-scores given by Gusfield's center star algorithm and the final SP-scores given by our genetic algorithm were recorded and compared with the SP-scores of the CLUSTALW alignments. The results of these tests for the BLOSUM 62 matrix when \( n = 15, m = 16, p_1 = 0.75, p_2 = 0.1 \) are given in Table 5.6 and Figure 5.3.

It must be noted that Gusfield's algorithm performed rather poorly on some COGs. The reason could be that the matrices used in the experiments (BLOSUM62 and PAM250) fail to satisfy the triangle inequality resulting in alignments with SP-scores that are not within \( 2 - 2/k \) of optimal.

Surprisingly, the proposed genetic algorithm outperformed the CLUSTALW alignments for 9 of 17 of the COGs on this limited test (using BLOSUM62 matrix). An additional 6/17 SP-scores were near the SP-score of the CLUSTALW alignment. Similar results were also achieved when PAM250 matrix was used. However, this comparison with CLUSTALW is limited since CLUSTALW does not directly optimize the SP-score.

Table 5.7 shows the timing data for the experiments described above (all timings done on a SPARC Server 1000). Notice that as the number of sequences per alignment increases in a linear manner, the running time for the program increases exponentially. The reason is that the current implementation uses the fixed parameter tractable algorithm for Gap 0-1 alignments. This algorithm could be replaced with a faster approximation algorithm for the Gap 0-1 alignment problem, such as the semidefinite programming algorithm described in Chapter 3.
Table 5.6 SP-Scores using the BLOSUM 62 matrix when $n = 15$, $m = 16$, $p_1 = 0.75$, $p_2 = 0.1$.

<table>
<thead>
<tr>
<th>COG #</th>
<th>CLUSTAL W</th>
<th>Center Star</th>
<th>New Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>2178 *</td>
<td>531</td>
<td>477</td>
<td>562</td>
</tr>
<tr>
<td>1983 *</td>
<td>-702</td>
<td>-749</td>
<td>-561</td>
</tr>
<tr>
<td>1549 *</td>
<td>-1191</td>
<td>-5912</td>
<td>-1184</td>
</tr>
<tr>
<td>1603 *</td>
<td>220</td>
<td>72</td>
<td>281</td>
</tr>
<tr>
<td>2157 *</td>
<td>442</td>
<td>583</td>
<td>596</td>
</tr>
<tr>
<td>1476 +</td>
<td>1654</td>
<td>1439</td>
<td>1644</td>
</tr>
<tr>
<td>1976 +</td>
<td>2925</td>
<td>1675</td>
<td>2674</td>
</tr>
<tr>
<td>2097 +</td>
<td>1788</td>
<td>-41</td>
<td>1706</td>
</tr>
<tr>
<td>1510 *</td>
<td>1740</td>
<td>1117</td>
<td>1967</td>
</tr>
<tr>
<td>1761</td>
<td>-9</td>
<td>-2848</td>
<td>-362</td>
</tr>
<tr>
<td>1515 *</td>
<td>1873</td>
<td>-2110</td>
<td>2468</td>
</tr>
<tr>
<td>1758 *</td>
<td>-1892</td>
<td>-3705</td>
<td>-1438</td>
</tr>
<tr>
<td>0565 +</td>
<td>3948</td>
<td>-1365</td>
<td>3157</td>
</tr>
<tr>
<td>2007</td>
<td>-102</td>
<td>-8678</td>
<td>-2845</td>
</tr>
<tr>
<td>1514 +</td>
<td>5094</td>
<td>-4054</td>
<td>5062</td>
</tr>
<tr>
<td>0219 +</td>
<td>9948</td>
<td>1981</td>
<td>9888</td>
</tr>
<tr>
<td>2003 *</td>
<td>5353</td>
<td>-797</td>
<td>5860</td>
</tr>
<tr>
<td>$\sum$</td>
<td>31620</td>
<td>-22915</td>
<td>29475</td>
</tr>
</tbody>
</table>
Table 5.7  Timing data

<table>
<thead>
<tr>
<th>COG #</th>
<th># of Seq.</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2178</td>
<td>3</td>
<td>3 minutes 5 sec.</td>
</tr>
<tr>
<td>1983</td>
<td>4</td>
<td>3 minutes 6 sec.</td>
</tr>
<tr>
<td>1549</td>
<td>4</td>
<td>25 minutes 50 sec</td>
</tr>
<tr>
<td>1603</td>
<td>4</td>
<td>5 minutes 28 sec.</td>
</tr>
<tr>
<td>2157</td>
<td>4</td>
<td>1 minute 7 sec.</td>
</tr>
<tr>
<td>1476</td>
<td>5</td>
<td>1 minute 55 sec.</td>
</tr>
<tr>
<td>1976</td>
<td>5</td>
<td>8 minutes 18 sec.</td>
</tr>
<tr>
<td>2097</td>
<td>6</td>
<td>5 minutes 24 sec.</td>
</tr>
<tr>
<td>1510</td>
<td>6</td>
<td>12 minutes 1 sec.</td>
</tr>
<tr>
<td>1761</td>
<td>6</td>
<td>7 minutes 43 sec.</td>
</tr>
<tr>
<td>1515</td>
<td>6</td>
<td>14 minutes 26 sec.</td>
</tr>
<tr>
<td>1758</td>
<td>7</td>
<td>12 minutes 34 sec.</td>
</tr>
<tr>
<td>0565</td>
<td>7</td>
<td>29 minutes 2 sec.</td>
</tr>
<tr>
<td>2007</td>
<td>7</td>
<td>33 minutes 6 sec.</td>
</tr>
<tr>
<td>1514</td>
<td>8</td>
<td>+30 Minutes</td>
</tr>
<tr>
<td>0219</td>
<td>9</td>
<td>+30 Minutes</td>
</tr>
<tr>
<td>2003</td>
<td>9</td>
<td>+30 Minutes</td>
</tr>
</tbody>
</table>
5.3 The Resulting Alignment for COG2157

The following is the example of the multiple sequence alignments for COG2157 (Ribosomal protein LX) produced by the CLUSTALW, Gusfield’s center star algorithm, and the genetic algorithm, described above. The asterix (*) characters point to columns where all characters match. The plus (+) characters point to columns with a predominant number of matches. It must be noted that CLUSTALW tries to minimize the amount of gaps, while the center star algorithm and the genetic approach produce more sparse alignments. In this example, both the Gusfield’s and the genetic algorithms outperform CLUSTALW significantly.
Alignment by CLUSTAL W

+ * +++ * + *++++ * ++ * ** *++++ * +++ * ++ * + + 

---LAKIFRITG---IMSUGKGDPLYFRKEYKALKPEDALEILYSEFGGCRYVKRSRIRKINALIEEIKPEDVTDPLVLLVTA-
MKMTKIFRVKGKFGLKWGDKLQP---FTKELNAIREEIIYERLYSEFGSKHRVPVSKVKEEIIEEEEISPEEVDQDPVVKALVQR-
---MEVKVFRVSG---YFEKDGK---FKFKEYLRAKKEHFEHKEVHVEDSRLHRKVKRRKIFIKIEREIKKEAEDIVVRRLSLEL
-----MKFEVGR---AFKTEGQKFTKVVEANNERYAELKVLQYSLIGSNHKVTRNLKIEVQA-----------------

SP-score = 442

Exact column matches = 12

Near column matches = 21

Alignment by the Gusfield’s Center Star Algorithm

+ * +++ * + *++++ * ++ * ** *++++ * +++ * ++ * + + 

L AK IFRITGIMSK KGKDP LYFRKEYKALKPEDALEILYSEFGGCRYVRNSRIRKINALIEEIKPEDVTDPLVLLVTA
MKMTKIFRVKGKFGLMKQFTHKNALIREEIIYERLYSEFGSKHRVPVSKVKEEIIEEEEISPEEVDQDPVVKALVQR
MEVK VFRVSGYFKE DGRK FK FTKCRAKKEHFEHKEVHVEDSRLHRKVKRRKIFIKIEREIKKEAEDIVVRRLSLEL
MK FEVGRAKFTLEGWQ KFTKVVEANNERYAELKVLQYSLIGSNHKVTRNLKIEVQA

SP-score=583

Exact Column matches = 16

Near Column matches = 20

Alignment by the proposed genetic algorithm

+ * +++ * + *++++ * ++ * ** *++++ * +++ * ++ * + + 

L AK IFRITGIMSK KGKDP LKYFRKEYKALKPEDALEILYSEFGGCRYVRNSRIRKINALIEEIKPEDVTDPLVLLVTA
MKMTKIFRVKGKF LMG DKLQFTHKNALIREEIIYERLYSEFGSKHRVPVSKVKEEIIEEEEISPEEVDQDPVVKALVQR
MEVK VFRVSGYFKE KDGK FK FTKCRAKKEHFEHKEVHVEDSRLHRKVKRRKIFIKIEREIKKEAEDIVVRRLSLEL
MK FEVGRAFKTLQIQ WQ KFTKVVEANNERYAELKVLQYSLIGSNHKVTRNLKIEVQA

SP-score = 587

Exact Column matches = 17

Near Column matches = 18
6. CONCLUSION

The algorithmic framework for multiple sequence alignment described in this thesis combines several well known techniques in a unique way. This hybrid approach uses a bounded-error approximation algorithm, local improvement through Gap 0-1 alignments, and genetic search. The novelty of this approach lies in the local improvement phase. As implemented in our software, the local improvement phase behaves in a way that is very similar to peephole optimization techniques employed in many production compilers [1]. The investigation of the applicability of these local improvements is the primary contribution of this research. Notice also that our approach provides the guarantees given by the bounded-error approximation algorithm, but may produce much better results.

The limited tests described above indicate that the combination of global, local and evolutionary approximation techniques can be successfully applied to the multiple sequence alignment. However, there is still much work to be done. The next logical task is to verify how well the semidefinite programming approximation algorithm for the Gap 0-1 problem would perform in place of the fixed parameter tractable version given here. The simple genetic algorithm presented here might also benefit from a wide range of other variations and local improvement techniques. This is the task for future research.
BIBLIOGRAPHY


A. WEB INTERFACE

As mentioned earlier, the genetic algorithm was implemented in C programming language. This program does not have its own user interface, thus, the data is entered from the file and the parameters are the command line keys.

To remedy this, a web interface was created as a front end for the program. The main part of this interface is the CGI script (written in Perl) that provides several services. First, it allows user to search a database of biological sequences called Entrez (see http://www.ncbi.nlm.nih.gov/entrez), as well as the COG database (see http://www.ncbi.nlm.nih.gov/COG). Users can enter a protein sequence name or a COG number. A script then forms a search query and sends it to the database search engine. The resulting web page is fetched and if a sequence or COG is found, it is extracted and saved locally as a file. If the data is found, it also appears on a screen in a tabular form (COGs are broken into the separate sequences). Entries in this table can be selected using check boxes and then used for creating alignments.

The second service provided by the user interface is the access to the genetic algorithm application. Users can specify the parameters for the program interactively, then select sequences found earlier by clicking check boxes. Finally, a multiple sequence alignment can be created when a user clicks the "Create alignment" button. At this point the script calls the C program, passing it specified parameters and sequences, and obtaining the resulting sequence alignment which it outputs on a screen.
Figure A.1 Screenshot of the web interface for the program
B. FRAGMENTS OF THE PROGRAM CODE

The following is the code for several functions from the genetic algorithm implemented by the author. These functions implement different important parts of the genetic algorithm, including local improvements and sexual reproduction.

/* Compute optimal alignment for two strings */
int align(int s1, int s2)
{
    int c1, c2, c3, c, i, x, y, xx, yy, px, py, len1, len2, cost = 0;
    int **mat, **ptr, *data, *data2;
    char t;

    len1 = strlen(str[s1]); len2 = strlen(str[s2]);

    data = malloc ((len1+1) * (len2+1) * sizeof(int));
    data2 = malloc ((len1+1) * (len2+1) * sizeof(int));
    mat = malloc ((len1+1) * sizeof(int *));
    ptr = malloc ((len1+1) * sizeof(int *));
    for (i=0; i<=len1; i++)
    { mat[i] = data+(i*(len2+1)); ptr[i] = data2+(i*(len2+1)); }

    /* compute values at the edge */
    mat[0][0] = 0;
    for (i=1; i<=len1; i++)
    { mat[i][0] = mat[i-1][0]+scores[gindex(str[s1], i-1)][20];
      ptr[i][0] = 1;
    }
for (i=1;i<=len2;i++) {
    mat[0][i] = mat[0][i-1]+scores[20][gindex(str[2],i-1)];
    ptr[0][i] = 2;
}

/ * compute the rest of the matrix */
for (y=i;y<=len2;y++)
    for (x=i;x<=len1;x++) {
        c1 = mat[x-1][y]+scores[gindex(str[1],x-1)][20];
        c2 = mat[x][y-1]+scores[20][gindex(str[2],y-1)];
        c3 = mat[x-1][y-1]+scores[gindex(str[1],x-1)][gindex(str[2],y-1)];
        c = c1; ptr[x][y]=1;
        if (c2>c) { c = c2; ptr[x][y]=2; }
        if (c3>c) { c = c3; ptr[x][y]=3; }
        mat[x][y] = c;
    }

/ * backtrace the matrix and recreate aligned strings */
sx = malloc(len1+len2+1); sy = malloc(len2+len1+1);
xx=0; yy=0; px=len1-1; py=len2-1;

do {
    if (ptr[px+1][py+1]==1)
        { *(sy+yy)=*; *(sx+xx)=*(str[1]+px);
            cost+=scores[20][gindex(str[1],px)]; yy++; xx++; px--; }
    if (ptr[px+1][py+1]==2)
        { *(sx+xx)=*; *(sy+yy)=*(str[2]+py);
            cost+=scores[gindex(str[2],py)][20]; yy++; xx++; py--; }
    if (ptr[px+1][py+1]==3)
        { *(sx+xx)=*(str[1]+px); *(sy+yy)=*(str[2]+py);
            if (*=*(sy+yy))
                cost+=scores[gindex(str[1],px)][gindex(str[2],py)];
            yy++; xx++; }
    py--; px--;
}
while ((px>=0)||(py>=0));
*(sx+xx)=0; *(sy+yy)=0;

/* reverse strings */
x = strlen(sx); y = strlen(sy);
for (i=0;i<x/2;i++) { t=*(sx+i); *(sx+i)=*(sx+x-i-1); *(sx+x-i-1)=t; }
for (i=0;i<y/2;i++) { t=*(sy+i); *(sy+i)=*(sy+y-i-1); *(sy+y-i-1)=t; }

free(data); free(data2); free(mat); free(ptr); return (cost); }

/****************************************************************************
Compute gap01 on a set of substrings
****************************************************************************/

void gap01(int f, int w)
{
int sc,s1,s2,i,j,x,**left,**right,**mid,*dl,*dr,*dm;
int len=1,p,sp,maxsp;
char *bin,*maxbin,*ptr;

/* allocate memory for arrays */
dl = malloc(k * k * sizeof(int));
dr = malloc(k * k * sizeof(int));
dm = malloc(k * k * sizeof(int));
left = malloc(k * sizeof(int *));
right = malloc(k * sizeof(int *));
mid = malloc(k * sizeof(int *));
for (i=0; i<k; i++) {
    left[i] = dl+(i*k);
    right[i] = dr+(i*k);
    mid[i] = dm+(i*k);
}
/* compute matrix a */
for (i=0;i<k;i++)

for (j=0; j<k; j++) {
    sc = 0;
    for (x=f; x<f+w; x++) {
        s1 = gindex(str[j], x);
        s2 = gindex(str[i], x);
        sc += scores[s1][s2];
    }
    mid[i][j] = sc;
}

/* compute matrix b */
for (i=0; i<k; i++)
    for (j=0; j<k; j++) {
        sc = 0;
        for (x=f; x<f+w-1; x++) {
            s1 = gindex(str[i], x);
            s2 = gindex(str[j], x+1);
            sc += scores[s1][s2];
        }
        s1 = gindex(str[j], f);
        sc += scores[20][s1];
        s2 = gindex(str[i], f+w-1);
        sc += scores[s2][20];
        left[i][j] = sc;
    }

/* compute matrix c */
for (i=0; i<k; i++)
    for (j=0; j<k; j++) {
        sc = 0;
        for (x=f+w-1; x<f+w; x++) {
            s1 = gindex(str[i], x);
            s2 = gindex(str[j], x-1);
            sc += scores[s1][s2];
        }
        s1 = gindex(str[j], f+w-1);
sc += scores[20][s1];
s2 = gindex(str[i], f);
sc += scores[s2][20];
right[i][j] = sc;
}

/* find alignment */
for (i=0; i<k;i++) len*=2;
bin = malloc(k+1); maxbin = malloc(k+1);
for (i=0; i<k; i++) *(bin+i)='0'; *(bin+k)=0;
maxsp = -BIG;
for (i=0; i<len; i++) {
    sp=0;
    for (j=0; j<k-1; j++)
        for (z=j+1; z<k; z++)
            if (((*(bin+j)=='0')&&(*(bin+z)=='0'))
                sp+=mid[j][z];
            if (((*(bin+j)=='1')&&(*(bin+z)=='1'))
                sp+=mid[j][z];
            if (((*(bin+j)=='0')&&(*(bin+z)=='1'))
                sp+=left[j][z];
            if (((*(bin+j)=='1')&&(*(bin+z)=='0'))
                sp+=right[j][z];
    }
    if (sp>maxsp) { maxsp=sp; strcpy(maxbin, bin); }
p=TRUE;
    for (j=0; j<k; j++)
        if (((*(bin+j)=='0')||(p)) { *(bin+j)='1'; p=FALSE; }
        if (((*(bin+j)=='1')&&(p)) { *(bin+j)='0'; p=TRUE; }
}

if (strchr(maxbin,'1')!=NULL) {
    for (j=0; j<k; j++)
        if (*(maxbin+j)!='0') {
            ptr = strdup(str[j]);
            *(str[j]+f) = '1'; *(str[j]+f+1) = 0;
            strcat(str[j],ptr+f); free(ptr);
        }
if (*(maxbin+j)=='i') {
    ptr = strdup(str[j]);
    *(str[j]+f+w) = ' '; *(str[j]+f+w+1) = 0;
    strcat(str[j],ptr+f+w); free(ptr);
}
}
free(maxbin); free(bin); free(d1); free(dr); free(dm);
free(left); free(right); free(mid);
}

/**************************************************************************
 Compute SP-score
**************************************************************************/
int spscore()
{
    int x,y,z,p1,p2,score=0,mins,maxs;

    maxs = maxstr(); mins = minstr();
    for (x=0;x<maxs;x++) {
        for (y=0;y<k-1;y++)
            for (z=y+1;z<k;z++) {
                if (strlen(str[y])<=x) p1 = 20;
                else p1 = gindex(str[y],x);
                if (strlen(str[z])<=x) p2 = 20;
                else p2 = gindex(str[z],x);
                score += scores[p1][p2];
            }
        printf("SP-score=%.f\n",score);
    }
    return(score);
}

/***************************************************************************/
Mutate random species
***************************************************************************
inline int p_mutation(double prob_p)
{
    double random_var = rand()/(RAND_MAX+1.0);
    return (random_var < prob_p);
}

***************************************************************************
Insert spaces into random spots
***************************************************************************
void mutate()
{
    int t,x,y,z;
    char *temp;

    for (t=0;t<POP;t++)
    {
        if (p_mutation(P2)) { printf("%i mutated\n",t);
            for (x=0;x<k;x++)
                for (y=0;y<SPACES;y++)
                    if (p_mutation(P2)) {
                        z = ceil((rand())/(RAND_MAX+1.0))*(strlen(sets[t][x])-1));
                        temp = strdup(sets[t][x]);
                        *(sets[t][x]+z) = ' '; *(sets[t][x]+z+1) = 0;
                        strcat(sets[t][x],temp+z);
                        free(temp);
                    }
    }
}

***************************************************************************
Mature species by applying gap01
***************************************************************************
void mature()
{
    int n,m,i,j,mins;

    for (n=0; n<POP; n++) {
        mins = minstr();
        for (m=0; m<3; m++) {
            for (i=2; i<5; i++) {
                for (j=0; j<mins-i+1; j++) {
                    inwork(n); gap01(j, i); cleanup(); outwork(n);
                    mins = minstr();
                }
            }
        }
    }

    for (i=3; i>=2; i--) {
        for (j=mins-i; j>=2; j--) {
            inwork(n); gap01(j, i); cleanup(); outwork(n);
            mins = minstr();
        }
    }

    /***********************
    Simulate sexual reproduction
    ***********************/

    void sex(int s1, int s2, int d1, int d2)
    {
        int x,z,y1,y2,p1,gap[k],max,min,yy1[k],yy2[k];
        char c, *ptr;

        printf("%i and %i will be born\n", d1, d2);
        for (x=0; x<k; x++) {
            y1 = ceil((rand())/(RAND_MAX+1.0))*(strlen(sets[s1][x])-10));
            while (*((sets[s1][x]+y1-1)==' ')) y1++;
ptr = sets[s2][x]; pi=0;
do {
while (*(sets[s1][x]+pi)==' ') pi++;
c = *(sets[s1][x]+pi);
ptr = strchr(ptr, c); ptr++; pi++;
} while (pi<y1);
y2 = ptr-sets[s2][x];
gap[x] = y1-y2; yy1[x] = y1; yy2[x] = y2;
}
max = -BIG; min = BIG;
for (x=0;x<k;x++) {
if (gap[x]>max) max = gap[x];
if (gap[x]<min) min = gap[x];
}
for (x=0;x<k;x++) {
strcpy(sets[d1][x], sets[s1][x]); strcpy(sets[d2][x], sets[s2][x]);

for (z=0;z<max-gap[x];z++)
  *(sets[d1][x]+yy1[x]+z) = ' '; *(sets[d1][x]+yy1[x]+z) = 0;
strcat(sets[d1][x], sets[s2][x]+yy2[x]);

for (z=0;z<gap[x]-min;z++)
  *(sets[d2][x]+yy2[x]+z) = ' '; *(sets[d2][x]+yy2[x]+z) = 0;
strcat(sets[d2][x], sets[s1][x]+yy1[x]);
}

/* ***********************************************
Pick a pair that will have sex
************************************************/
void pick_pair(int scores[], int n, int *x, int *y) {
  double random_var = rand()/(RAND_MAX+1.0);
  int sum, i, j;
double prob_sum, S;
sum = 0;

for (i=0; i<n; i++) sum = sum + scores[i];
S = (n-1)*sum*1.0;

prob_sum = 0;
for (i=0; i<n; i++)
for (j=i+1; j<n; j++) {
    prob_sum = prob_sum + (scores[i] + scores[j])/S;
    if (prob_sum > random_var) {
        (*x) = i;
        (*y) = j;
        return;
    }
}
(*x) = n-2;
(*y) = n-1;
return;

/H******************************************************************************/
/* Simulate fighting between species */
/H******************************************************************************/
void fight_sex()
{
    int x, y, s, t[POP], p[POP], die, min, max, m, s1, s2, live;
    for (x=0; x<POP; x++) { inwork(x); t[x] = spscore(); }
    max = -BIG; for (x=0; x<POP; x++) if (t[x]>max) { max = t[x]; m = x; }
    if (max>maxb) {
        maxb = max; free(best); best = malloc (k*sizeof(char *));
        for (x=0; x<k; x++) { best[x] = malloc(strlen(sets[m][x])+1));
            strcpy(best[x], sets[m][x]); }
    }
{,

die = (int) (POP*PI*1.0); if ((POP-die)<2) die = POP-2;
live = POP-die;
for (x=0;x<POP;x++) p[x]=TRUE;
printf("%i species will die!\n",die);

for (s=0;s<die;s++) {
    min = BIG;
    for (x=0;x<POP;x++) if ((t[x]<=min)&&(p[x])) { min = t[x]; m = x; }
    p[m] = FALSE;
    printf("%i","m);
} printf(" are died :-(\n");

/* sort species (alive go to the top) */
for (s=0;s<die;s++)
    for (x=0;x<live;x++)
        if (!p[x]) {
            m = POP-1; while (!p[m]) m--;
            p[x] = TRUE; p[m] = FALSE;
            t[x] = t[m];
            for (y=0;y<k;y++) { free(sets[x][y]);
                sets[x][y] = malloc(strlen(sets[m][y])+BUFFER);
                strcpy(sets[x][y],sets[m][y]); }
}

/* reproduction */
for (s=0;s<die;s+=2) {
    pick_pair(t, live, &s1, &s2);
    printf("%i and %i will have sex, ",s1,s2);
    if ( (live+s1)<POP) sex(s1,s2, live+s, live+s+1);
    else sex(s1,s2, live+s-1, live+s);
}
/****************************************************************************
Gusfield's algorithm
****************************************************************************/

void gusfield()
{
    int **d,*data,x,y,z,sum,maxsum,sc,xk,yk,xx;
    char *temp;

    data = malloc(k * k * sizeof(int));
    d = malloc(k * sizeof(int *));
    for (z=0; z<k; z++) d[z] = data+(z*k);

    best = malloc (BUFFER);

    /* fill out matrix of scores */
    for (y=0;y<k;y++) {
        for (x=0;x<k;x++) {
            if (x!=y) d[x][y] = align(x,y); else d[x][y]=0;
        }
    }

    /* find Sc - star center */
    maxsum = -BIG;
    for (y=0;y<k;y++) {
        sum = 0;
        for (x=0;x<k;x++) sum += d[x][y];
        if (sum>maxsum) { maxsum = sum; sc = y; }
    }
    printf("maxsum=%i, sc=%i\n",maxsum,sc);

    /* create alignment */
    for (x=0;x<k;x++) {
        if (x!=sc) {

align(sc,z);
if (strlen(str[sc])!=strlen(sx)) { xx=0;
for (xk=0;xk<strlen(str[sc]);xk++,xx++)
  if (*(str[sc]+xk)!=*(sx+xx)) {
    for (yk=0;yk<x;yk++)
      if ((yk!=sc)&&(yk!=x)) {
        temp = malloc(strlen(str[yk])+10); strcpy(temp,str[yk]);
        *(str[yk]+xk) = ' '; *(str[yk]+xk+1) = 0;
        strcat(str[yk],temp+xk);
      }
    xx++;
  }
  free(temp);
}
xx++;
}
strcpy(str[sc],sx);
strcpy(str[x],sy);
}
}
printf("Gusfield's Alignment:\n");
cleanup(); output();
printf("---------------------------------------\n");
maxb = -BIG;
init_mutate();
for (x=0;x<GEN;x++) {
mature(); fight_sex(); mutate();
printf("-[%d]----------------------------------------\n",x+1);
}
for (x=0;x<k;x++) strcpy(str[x],best[x]);
output();
}