AN INTERACTIVE SYSTEM FOR SEQUENCE ANALYSIS

AN INTERACTIVE SYSTEM FOR SEQUENCE ANALYSIS

By

XIANGDONG CHEN, Ph.D. (Mathematics)

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (1994)	MCMASTER UNIVERSITY
(Computation)	Hamilton, Ontario

TITLE:	An Interactive System for Sequence Analysis
AUTHOR:	Xiangdong Chen
	M. Sc. (Mathematics, Hunan University, China)
	Ph.D. (Mathematics, McMaster University, Canada)
SUPERVISOR:	Dr. Tao Jiang
NUMBER OF PAGES:	ix, 75

Abstract

Sequence Analysis Tool (SAT) is an X-window (OPEN LOOK version) based interactive system developed for sequence analysis. In this first version, it provides a friendly graphical user interface and convenient functions for performing various tasks required in sequence alignment. In particular, space-efficient algorithms for pairwise alignment and 3-star alignment are implemented as functionalities, which can be used to serve most sequence alignment tasks and therefore provide a basis for further improvement of this tool.

SAT is also targeted at providing a testing platform for performance analysis of various alignment-related algorithms. A set of procedures is developed to provide an application programming interface with which other related programs can be easily connected to SAT.

SAT is programmed in C/Xlib/OLIT. The object-oriented style makes further maintenance and improvement easy.

Acknowledgements

I would like to express my thanks to Dr. T. Jiang for his encouragement, support and guidance. He has contributed greatly to my interest in the subject. I am also grateful for his careful review of this manuscript.

I am grateful to Dr. P.J. Ryan for his careful reading of the final draft and valuable comments, to Dr. S. Qiao for his advice on the Unix system and his participation in the evaluation of the project, and to Mrs. M. Belec for her suggestions.

I am also grateful to Mr. L. Wang for his valuable discussion, cooperation and the sharing of computer resources.

A special thanks also goes to Mr. D. Trottier for his technical support and advice on the system.

I wish to thank the professors, staff, and my fellow graduate students in the department, who made my study at McMaster University an extremely enriching experience.

I am especially grateful to my wife, daughter and my parents for their constant encouragement, understanding, love and patient cooperation.

Contents

List of Figures

Chapter 1
Introduction

With the development of Molecular Biology, the analysis of DNA, RNA and Protein sequences is playing an important role in the study of evolution of life. The large amount of information and computation involved in the analysis has also brought many challenging computational problems to mathematicians and computer scientists. Sequence alignment, comparison and the inference of phylogenies are no doubt among the most important topics in sequence analysis. Since these problems are usually computationally hard (some are NP-hard in terms of computer science), optimal solutions are not available or not practical. Therefore heuristic and approximate methods are applied and new proposals continue to emerge. For a new algorithm, although it is important that the algorithm is good in the sense of complexity theory, it will be more convincing if a systematic performance analysis is conducted on the computed results, and the required time and resources. This demand gives a reason for the birth of this Sequence Analysis Tool.

Sequence Analysis Tool (SAT) is an X Window based interactive system aimed for providing a platform so that performance analysis of algorithms can be conducted easily. At its early age, SAT can only address algorithms related to a limited set of alignment problems related to the research of the algorithm group at McMaster University. Right now it provides facilities for doing pairwise alignment and 3-star alignment, which are essential for other tasks such as aligning multiple sequences and inferring phylogenies.

SAT includes a graph editor, which provides basic editing functions such as *create, modify, delete* and *undo* for graphs. When a graph represents the relationship between a set of sequences, each node is assigned a sequence and each edge corresponds to a pairwise alignment. The users can (1) view the sequences and alignments through pop-up windows; (2) specify the score system and alignment parameters; and (3) conduct pairwise alignment and 3-star alignment locally and globally.

The format for denoting sequences in a file is compatible with other popular sequence alignment software such as **clustalv**, and the output alignments are readable by humans and the system itself.

A set of procedures has been developed as a programming interface so that independent programs can be developed to talk with SAT and use SAT's drawing and displaying facilities employing the multiprocessing utilities of Unix. As a practice, we have implemented the algorithm proposed by Jiang, Lawler and Wang [25] for the problem of tree alignment with a fixed phylogeny in such a way that the program runs by itself and can also communicate with SAT. This should set a model for other developments involving interfacing with SAT.

SAT is developed with the OPEN LOOK Intrinsics Toolkit (OLIT) and tested in the OpenWindows environment on a SUN SPARCstation (LX).

This report contains seven chapters and three appendices.

Chapter 2 introduces the sequence alignment problems in detail. Dynamic programming is the major approach for solving alignment problems. We present and implement space-saving algorithms for pairwise alignment, tree alignment with a given phylogeny and 3-star alignment. Test results on the 3-star alignment program are also listed to see how the space-saving approach influences the computing time.

Chapter 3 briefly reviews user interface style and development issues in general. A brief introduction to the X Window System is also included.

Chapter 4 discusses the development and implementation techniques used in this project. Main data structures and algorithms for manipulating objects to be drawn are presented.

Chapter 5 is a user guide for the main features of SAT.

Chapter 6 demonstrates an application of SAT in tree alignment.

Chapter 7 gives some suggestions for further improvement.

Appendix A is a sample input file for SAT.

Appendix B is extracted from the output of the test in Chapter 6.

Appendix C explains the score matrix file format required by SAT.

Chapter 2
Sequence Analysis

The central dogma of modern Molecular Biology is that DeoxyriboNucleic Acid (DNA) is the primary genetic material. DNA is a molecule composed of four nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T), which, conceptually, are linked linearly to form long chains called polynucleotides. These chains are called DNA sequences. Proteins are sequences made from 20 amino acids. The study of these sequences results in important insights into human biochemistry, physiology and disease processes.

An important method for studying these biosequences is the sequence comparison, which is necessary for the detection of common structure and function as well as for the study of evolutionary relationship. The basic idea of most sequence comparison algorithms is to obtain a measure of the similarity (or distance) among a collection of sequences. Alignments are usually constructed so as to maximize the measure of similarity (or minimize the distance) between sequences. Because of the existence of various clever alignment techniques and algorithms, comparative sequence analysis is an active and fruitful area for the application of computation to biological problems.

To obtain an optimal (or quasi-optimal) alignment, the dynamic programming technique is usually applied. Dynamic programming algorithms are theoretically important and beautiful. However, these methods require exhaustive computation, which may become impractical in many cases because of the limits of computing time and memory. How to improve various algorithms to save storage or to increase speed is a very important research topic.

In this chapter, we introduce the fundamentals of sequence alignment, especially pairwise alignment, tree alignment with a given phylogeny, and 3-star alignment. These three types of alignments are what the SAT system is able to handle at the time being.

2.1 Pairwise Alignment

In this section, we formally define the terminology and notations needed for alignment. Pairwise alignment is the basis of all other sequence comparison methods.

A *null* is the symbol "-" or "$\emptyset$".

An *alphabet* Σ is a finite set of symbols containing null.

An *element* is a member of the alphabet Σ.

A *letter* is an element other than null.

A *sequence* (or *string*) is a finite string of letters.

A *padded sequence* (or *padded string*) is a finite string of elements.

Given an alphabet Σ, a scoring function $w : \Sigma \times \Sigma \longrightarrow$**Real** can be defined to measure differences or similarities between any two elements. In this project, we use difference-measuring scoring functions. Therefore the following discussions are based on the distance criteria.

Given two sequences A and B, an alignment of A and B is expressed as

$$a_1^* a_2^* \cdots a_l^*$$
$$b_1^* b_2^* \cdots b_l^*$$

where the two rows are padded sequences obtained with the insertion of nulls into A and B respectively, and no column contains two nulls.

To determine the quality of an alignment, one needs some scoring and optimization criterion defined for alignments. For the above alignment, it is natural to consider the sum

$$\sum_{i=1}^{l} w(a_i^*, b_i^*).$$

However, to get biologically reasonable alignments, additional costs must be charged for *gaps*, which are maximal strings of adjacent nulls in one sequence aligned with letters in the other. Therefore we can say the *cost* (or *score*) of a pairwise alignment is the sum of the cost of all aligned pairs of elements and the cost of all gaps.

An optimal alignment of A and B is one that minimizes the cost over all possible alignments. For any two sequences, there may exist many optimal alignments. The sequence alignment problem is to find one or more optimal alignments and the optimal cost. The standard method uses *dynamic programming* on variants of the following recurrence relations.

Let $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$. Define a cost matrix CC such that $CC[i,j]$ denotes the minimum cost of aligning $a_1a_2\cdots a_i$ and $b_1b_2\cdots b_j$. When gap costs are not charged, we have the recurrence relation:

$$CC[0,0] = 0,\ CC[0,j] = \sum_{k=1}^{j} w(\phi, b_k),\ CC[i,0] = \sum_{k=1}^{i} w(a_k, \phi),$$

and

$$CC[i,j] = \min \begin{cases} CC[i-1,j] + w(a_i, \phi) \\ CC[i-1,j-1] + w(a_i, b_j) \\ CC[i,j-1] + w(\phi, b_j) \end{cases}$$

When gap penalties are considered, let g_k be the gap cost for a gap of k bases. Then we have the recurrence relation:

$$CC[0,0] = 0,\ CC[0,j] = g_j,\ CC[i,0] = g_i,$$

and

$$CC[i,j] = \min \begin{cases} CC[i-1,j-1] + w(a_i, b_j) \\ \min_{1\le k\le j}\{CC[i,j-k] + g_k\} \\ \min_{1\le k\le i}\{CC[i-k,j] + g_k\} \end{cases}$$

A direct implementation of this relation needs $O(mn)$ space. Dynamic programming with this kind of recurrence relations has been studied extensively. Lots of time or space saving strategies have been proposed and actually applied, see [14, 6].

It is important to realize that an optimal alignment is optimal only for the particular scoring system and gap costs. To make significant biological application of these mathematical models, we have to consider how to choose the scoring function w on Σ, and also the gap cost function. In practice, the primary scoring system used for nucleic acid sequences is the identity matrix. For protein sequences, the most common choice for measuring similarity is the Dayhoff mutation matrices (PAM matrices), see [28]. Among possible gap costs, the simplest and most commonly used are *linear gap costs*, which charges a fixed amount for each space. Because they have been taken to subsume null costs, they are usually expressed as

$$g(k) = a + bk,$$

where a is called as the *open gap penalty* and b as the *gap extension penalty*, which is actually the null cost. Gap penalties have a large effect on an alignment and it is wise to sample a wide range of values in order to find the most interesting optimal alignment. The following example is taken from the book [8] page 130, which shows how gap penalties influence sensible alignments. Two alignments of human pancreatic hormone and chicken pancreatic hormone are shown.

```
  Human  ALLLQPLLGAQGAPLEPVYPGDNATP.EQMAQ.YAAD.LRRYINMLTRPRYGKRHKEDTLAF
Chicken  G....P..S.Q..P..T.YPGDDA.PVEDLIRFY..DNLQQYLNVVT......RHR.....Y
```

An optimal alignment without gap penalties.

```
  Human  ALLLQPLLGAQGAPLEPVYPGDNATPEQMAQYAADLRRYINMLTRPRYGKRHKEDTLAF
Chicken  ............GPSQPTYPGDDAPVEDLIRFYDNLQQYLNVVTRHRY...........
```

An optimal alignment with gap penalty of $1.0 + 0.1\times$(gap length).

2.2 Tree Alignment

An *evolutionary tree* is a tree whose nodes are associated with sequences. The cost of an edge in the tree is defined as the edit distance (optimal cost) between the two sequences associated with the ends of the edge. The cost of a tree is the sum of the costs of all edges. Given sequences X, the *optimal evolutionary tree* or *multiple tree alignment* or, simply, *tree alignment* problem is to find a set of sequences Y and an evolutionary tree T (sequences from X are assigned to leaves and those from Y are assigned to internal nodes of T) which minimizes the cost of T over possible sets Y and trees T. This problem is proved to be MAX SNP-hard [24]. Here we are mainly interested in the so-called tree alignment with a given phylogeny, that is, *given a set X of sequences and a phylogeny T which is defined as a tree structure such that each* *leaf is assinged a unique sequence of X, we need to construct a sequence for each internal node such that the total cost of the tree is minimized.*

The problem of tree alignment with a given phylogeny is NP-hard even if the phylogeny is a binary tree [24]. Some heuristic algorithms have been proposed [9, 18]. In the following, we outline an efficient approximation algorithm based on recent results of Jiang, Lawler and Wang [25].

First, we need some notations. For a (rooted) tree T, $r(T)$ denotes the root of T, $c(T)$ denotes the cost of T, $Leaf(T)$ denotes the set of the leaves of T. For each node v of T, T_v denotes the subtree of T rooted at v. A leaf that is a descendant of node v is called a *descendant leaf* of v. When all leaves have been assigned sequences, we define $S(v)$ to be the set of sequences assigned to the descendant leaves of v. An evolutionary tree is called a *lifted* tree if the sequence associated with each node equals the sequence associated with some child of the node.

Let $X = \{s_1, \ldots, s_k\}$ be a set of sequences and T a phylogeny for X such that the degree of each internal node of T is 3. To construct an evolutionary tree, we need the following steps.

Step 1: **Initialize the internal nodes.**

Take an arbitrary edge uv of the tree T. Adding a new node r and replacing the edge uv with two new edges ru and rv, we get a rooted tree $\bar{T}$ of root r.

For each $v \in \bar{T}$ and each $s_i \in S(v)$, let $D[v, s_i]$ denote the cost of an optimal lifted tree for $\bar{T}_v$ with v being assigned the sequence s_i. $D[v, s_i]$ can be computed as follows. For each leaf v, we define $D[v, s_i] = 0$ if s_i is assigned to v. Let v be an internal node, and v_1 and v_2 its children. For each $s_i \in S(v)$, s_i must belong to one of $S(v_1)$ and $S(v_2)$. We have the recurrence relation:

$$D[v, s_i] = \begin{cases} \min_{s_j \in S(v_2)} \{D[v_1, s_i] + D[v_2, s_j] + dist(s_i, s_j)\} & \text{if } s_i \in S(v_1) \\ \min_{s_j \in S(v_1)} \{D[v_1, s_j] + D[v_2, s_i] + dist(s_i, s_j)\} & \text{if } s_i \in S(v_2) \end{cases}$$

The full algorithm is described in Figure 2.1. It outputs an evolutionary tree with cost at most $2c(T^{min})$ and requires $O(k^3 + k^2n^2)$ time in the worst case, where T^{min} denotes an optimal evolutionary tree and n denotes the maximum length of the given sequences.

```
Input: $X = \{s_1, \ldots, s_k\}$ (sequences set), $T$ (a phylogeny for $X$)
Output: Lifted tree $T$.
1. begin
2.   for each pair $(i, j)$, $1 \le i < j \le k$, do
3.       compute $dist(s_i, s_j)$.
4.   Construct $\bar{T}$.
5.   for each level of $\bar{T}$, with the bottom level first, do
6.       for each node $v$ at the level do
7.          for $i = 1$ to $k$
8.             if $s_i \in S(v)$ then compute $D[v, s_i]$.
9.   Select an $s \in X$ such that $D[r(\bar{T}), s]$ is minimized.
10.   Compute the lifted tree $T$ by back-tracing.
11. end.
```

Figure 2.1: Algorithm **initialization.**

Step 2: **Local Optimization.**

Starting with an evolutionary tree, we can use an iterative improvement method to update and improve the sequences assigned to the internal nodes to get a better approximation. Recall that the degree of each internal node of the phylogeny under our consideration is three. Based on each internal node, we can construct a 3-component which is a subtree consisting of the internal node and three edges connecting to it.

Then on each 3-component the local optimization is performed by the star-alignment technique introduced in next section.

To illustrate these two procedures more clearly, consider the phylogeny in Figure 2.2, which contains nine given species on its nine leaves. (The same example is

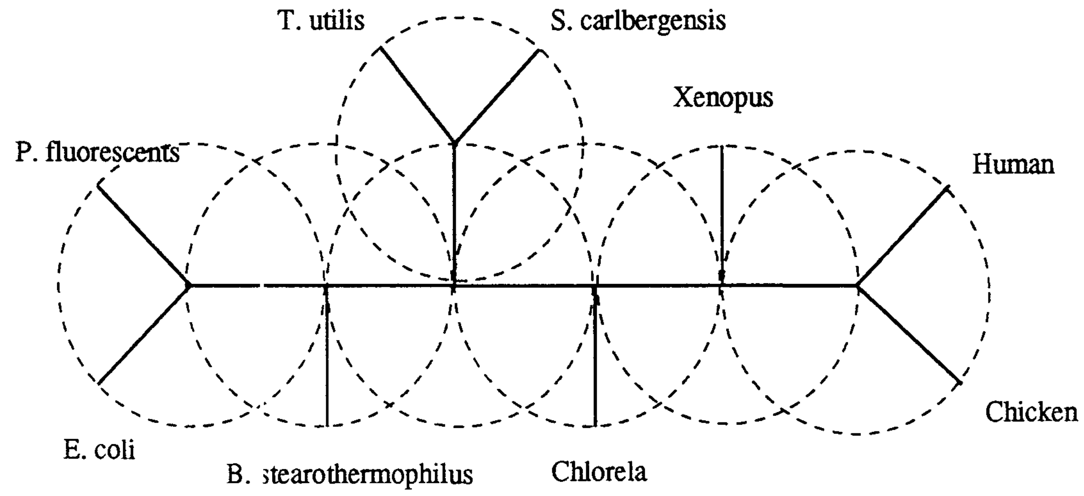


Figure 2.2: A phylogeny with nine species, which is divided into seven 3-components.

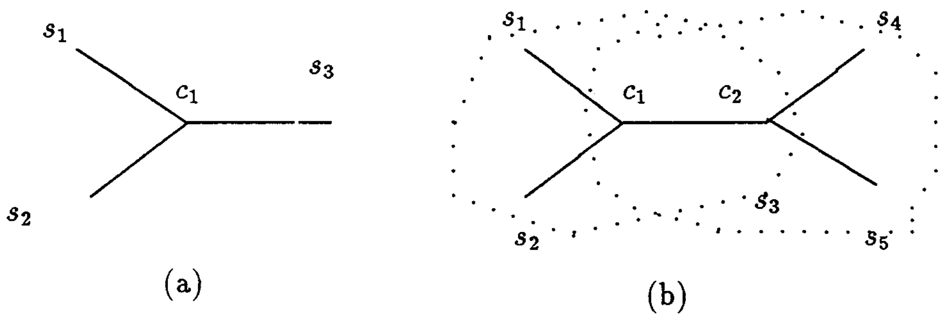


Figure 2.3: (a) A 3-component. (b) Two overlapping 3-components.

also used in [18]). To construct an evolutionary tree, we assign one of the nine given sequences to each internal node by applying algorithm introduced in Step 1. Then we divide the phylogeny into seven 3-*components* as shown in Figure 2.2. Local optimization is done for every 3-component as follows. For the 3-component in Figure 2.3(a), from the labels (sequences) s_1, s_2, and s_3 of the three terminals, we can compute the label c_1 (sequence) of the center using dynamic programming (introduced in next section) to minimize the cost of the component [17, 9]. The revised c_1 can then be used to update the center label c_2 of an overlapping 3-component (see Figure 2.3(b)). The algorithm converges since each local optimization reduces the cost of the tree by at least one. Thus, if the process is repeated long enough, every 3-component will become optimal. However, this does not necessarily result in an optimal evolutionary tree. Nonetheless, it seems the algorithm can produce a reasonably good evolutionary tree after 5 iterations [18].

2.3 Star Alignment

A *star-alignment* is a special case of tree alignment in which the tree has only one internal node. Here we are especially interested in star-alignment of three strings, which is a process stated as: *Given three strings A, B and C, construct a new string D and optimally align D with each of A, B and C. The sum of costs of the three pairwise alignments is defined as the cost of the star-alignment.* This process will be denoted as *star-alignment*(A, B, C). An optimal star-alignment(A, B, C) is one attaining the minimum cost among all possible star-alignment(A, B, C). The string D newly constructed in an *optimal* star-alignment(A, B, C) is called a *center-string* of (A, B, C).

In the following, we present an algorithm using the dynamic programming method to find a center-string of (A, B, C) with given strings $A = a_1a_2\cdots a_M$, $B = b_1b_2\cdots b_N$ and $C = c_1c_2\cdots c_K$.

Let A_i denote the i-symbol prefix $a_1a_2\cdots a_i$ of A, B_j the j-symbol prefix $b_1b_2\cdots b_j$ of B and C_k the prefix $c_1c_2\cdots c_k$ of C. Define

$$Cost(i,j,k) = \text{the cost of an optimal star-alignment}(A_i, B_j, C_k).$$

Then we can obtain the following recurrence relation:

$$Cost(i,j,k) = \min_{\delta\neq 0}[Cost(i-\delta_1, j-\delta_2, k-\delta_3)+\min_{x\in\Sigma}\{w(\delta_1 a_i, x)+w(\delta_2 b_j, x)+w(\delta_3 c_k, x)\}]$$

where $\delta_1, \delta_2, \delta_3 \in \{0, 1\}$, $1a = a$, $0a = \emptyset$, and $\delta = (\delta_1, \delta_2, \delta_3)$. There are seven different values of δ with $\delta \neq 0$. The element x where the minimum is attained is the last element (maybe $\emptyset$) of the constructed center-string of (A_i, B_j, C_k).

In the implementation of this recurrence relation, we first preprocess the part

$$\min_{x\in\Sigma}\{w(\delta_1 a_i, x) + w(\delta_2 b_j, x) + w(\delta_3 c_k, x)\}.$$

Define a relation *lookup* : $\Sigma \times \Sigma \times \Sigma \longrightarrow \Sigma \times \mathbf{Real}$, such that, for any $e_1, e_2, e_3 \in \Sigma$, *lookup*(e_1, e_2, e_3) has two fields, one is denoted as *cost* and represents the minimum sum, and the other is denoted as *letter* and stores the letter x such that the minimum sum is attained at x.

To obtain a center-string of (A, B, C), one can use the straightforward *back-tracing technique.* That is, set up a 3-dimensional matrix *TraceMat*, compute the recurrence relation of $Cost(i,j,k)$ and store, in the (i,j,k) cell of TraceMat, the element x and a pointer pointing to $(i-\delta_1, j-\delta_2, k-\delta_3)$ such that x and $\delta = (\delta_1, \delta_2, \delta_3)$ lead to the minimum value $Cost(i,j,k)$. With the information stored in the matrix

TreeMat, we can then simply start from the (M, N, K) cell of TraceMat. By following the pointers, a linked list is established and a center sequence is obtained in the reverse order. Obviously, an implementation based on this technique needs $O(NMK)$ space. In practice, this space requirement often limits the method's applicability.

If we are only interested in the cost of an optimal star-alignment(A, B, C), the space requirement can be reduced dramatically. More specifically, for each fixed $1 \leq i \leq M$, we use $level(i)$ to represent the 2-dimensional array consisting of $\{Cost(i,j,k) | 1 \leq j \leq N,\ 1 \leq k \leq K\}$. The recurrence relation shows that $Cost(i,j,k)$ depends only on seven values in $level(i-1)$ and $level(i)$. Therefore, two matrices of size $N * K$ are adequate to compute successive levels. In fact, with a little care, one matrix of size $N * K$ and one vector of size $K + 1$ suffice. Suppose $Cost(i, \bar{j}, \bar{k})$ needs to be computed and values preceding it have already been obtained. Then we can define a matrix CC_{N*K} and a vector CB of size $K + 1$ represented as follows:

$$CC(j,k) = \begin{cases} Cost(i,j,k) & \text{if } j < \bar{j} \text{ and } k < \bar{k} \\ Cost(i-1,j,k) & \text{otherwise} \end{cases}$$

$$CB(k) = \begin{cases} Cost(i-1,j,k-1) & \text{if } k < \bar{k} \\ Cost(i-1,j-1,k) & \text{otherwise} \end{cases}$$

$$CB(K+1) = Cost(i-1, \bar{j}-1, \bar{k}-1).$$

With this loop-invariant condition, we can present an algorithm as shown in Figure 2.4 for calculating the cost of an optimal star-alignment(A, B, C) using $O(N * K)$ space. To make the algorithm shorter and easy to read, we include the testing of the boundary situation in the main body. But in our implementation, we deal with the boundary cases separately to eliminate the "**if**" instructions.

```
Algorithm $costonly\_star\_align(A, B, C, CC)$
Input: $A$, $B$ and $C$ (strings of size $M$, $N$ and $K$).
    { We assume the lookup relation has been computed}
Output: $CC$ (a matrix of size $N * K$) and $cost$
where $CC(j,k)$ represents the cost of an optimal 3-star-align$(A_M, B_i, C_j)$.
var array $CC[0 \cdots N][0 \cdots K]$, $CB[0 \cdots (K+1)]$.
begin
   $CC(0,0) = 0$
   for $i = 0$ to $M$ do
     for $j = 0$ to $N$ do
       for $k = 0$ to $K$ do
         begin
           if $(i-1, j-1, k-1)$ is valid then
             $value(0) = CB(K+1) + lookup(a_i, b_i, c_k).cost$;
           if $(i-1, j-1, k)$ is valid then
             $value(1) = CB(k) + lookup(a_i, b_j, \phi).cost$;
           if $(i-1, j, k-1)$ is valid then
             $value(2) = CB(k-1) + lookup(a_i, \phi, c_k).cost$;
           if $(i-1, j, k)$ is valid then
             $value(3) = CC(j,k) + w(a_i, \phi)$;
           if $(i, j-1, k-1)$ is valid then
             $value(4) = CC(j-1, k-1) + lookup(\phi, b_j, c_k).cost$;
           if $(i, j-1, k)$ is valid then
             $value(5) = CC(j-1, k) + w(b_j, \phi)$;
           if $(i, j, k-1)$ is valid then
             $value(6) = CC(j, k-1) + w(c_k, \phi)$;
           {update $CB$ and $CC$ as follows:}
           if $(i > 0)$ and $(j > 0)$ then
             $CB(K+1) = CB(k)$;
           if $(i > 0)$ then
             $CB(k) = CC(j,k)$;
           $CC(j,k) = \min$ of $\{value(i) | 0 \leq i \leq 6\}$;
         end
   $cost = CC(N,K)$;
   Output $cost$ and CC;
end
```

Figure 2.4: Algorithm **costonly_star_align**.

To actually produce a center-string of a star-alignment, we generalize the recursive divide-and-conquer technique of Hirschberg [11] and Myers and Miller [14] so that we obtain an algorithm with $O(N * K + \log M)$ space requirement. The central idea is to find the "midpoints" of an optimal star alignment of three strings by using a "forward" and "backward" application of the quadratic space *Costonly_star_align* algorithm. Then a center-string can be obtained by recursively determining optimal star alignments on both side of the midpoints.

For a sequence X, let $rev(X)$ denote the reverse of X and let X_i^T denote the suffix $x_{i+1}x_{i+2}\cdots x_M$ of X. Given three sequences A, B and C of sizes M, N and K respectively, applying the algorithm *Costonly_star_align* to $rev(A)$, $rev(B)$ and $rev(C)$, we obtain a matrix RR such that the entry $RR(N-j, K-k)$ represents the cost of an optimal star-alignment (A, B_j^T, C_k^T).

Now we are in the position of explaining our algorithm of delivering a center-string of a 3-star alignment. Again, suppose three strings A, B and C are of non-zero length M, N and K respectively. Let $i^* = \lfloor M/2 \rfloor$, then $level(i^*)$ bisects the cube associated with the recursive $Cost$ function (defined on Page 13). Applying the *Costonly_star_align* algorithm to the strings A_{i^*}, B and C, we get a matrix $matf$ satisfying:

$$matf(j,k) = \text{ the cost of an optimal star-alignment}(A_{i^*}, B_j, C_k).$$

Then applying the *Costonly_star_align* algorithm to the strings $rev(A_{i^*}^T)$, $rev(B)$ and $rev(C)$, we get a matrix $matb$ satisfying:

$$matb(j,k) = \text{ the cost of an optimal star-alignment}(A_{i^*}^T, B_j^T, C_k^T).$$

For any star-alignment(A, B, C), there exist $j \in [0, N]$ and $k \in [0, K]$ such that the star-alignment is the concatenation of a star-alignment(A_{i^*}, B_j, C_k) and a star-alignment$(A^T_{i^*}, B^T_j, C^T_k)$. Thus the cost of an optimal star-alignment(A, B, C) is

$$\min\{matf(j,k) + matb(N-j, K-k) | j \in [0, N] \text{ and } k \in [0, K]\}.$$

If the minimum is attained at j^* and k^*, then (i^*, j^*, k^*) is an optimal midpoint for the problem. Now, the crucial point for using the divide-and-conquer method is that the concatenation of an optimal star-alignment$(A_{i^*}, B_{j^*}, C_{k^*})$ and an optimal star-alignment$(A^T_{i^*}, B^T_{j^*}, C^T_{k^*})$ is an optimal star-alignment(A, B, C). Therefore we can employ the midpoint (i^*, j^*, k^*) to split the star-alignment problem into two sub-problems of star-aligning shorter strings. The sub-problems are solved by calling the above processing recursively.

The recursion's boundary cases, i.e. the size of one of the three strings is 1 or 0, are handled directly by using *Backtracing* technique since only quadratic space is required now.

The full algorithm for finding a center-string is outlined in Figure 2.5. It uses $O(NK + \log M)$ space: $O(NK)$ for the dynamical allocated space for $matf$ and $matb$ or for the boundary cases, and $O(\log M)$ for the implicit activation s-tack needed for no more than $\lfloor \log M \rfloor + 1$ levels of recursion. Now consider the time requirement. Obviously, the procedure *costonly_star_align* for strings of sizes M, N and K takes $O(MNK)$ time; we assume it is c_1MNK. Then in the algorithm *find_center_string_star_align*, boundary cases take $O(M + NK)$; line (α) takes $c_1(M/2)NK$, line (β) also takes $c_1(M/2)NK$ and line (γ) takes c_2NK time. So the main body (lines α, β and γ) of the top-level call takes $c_1MNK + c_2NK$ time. The time spent in the main bodies of the two recursive calls at lines (δ) and (ϵ) is

```
Algorithm find_center_string_star_align(A, B, C)
Input: A, B and C (strings of size M, N and K).
    { We assume the lookup relation has been computed}
Output: final_seq (a center-string of star alignment) and
             cost (the optimal cost of star alignment)

begin
    cost = findcenter(A, B, C)
    print(final_seq) {a center string for the star-alignment}
end

recursive function findcenter(A, B, C)
var array matf[0 ··· N][0 ··· K], matb[0 ··· N][0 ··· K];
      { dynamically allocated in the implementation }
begin
   if ($M \leq 1$) or ($N \leq 1$) or ($K \leq 1$) then
       { Take I1 as the minimum of {M, N, K}, I2 and I3 as others,
        allocate at most two matrices of size $I2 \times I3$ to store information
        for backtracing }
       apply backtracing technique directly to find a partial center-string,
       which will be appended to final_seq.
   else
      begin
           $i^* = \lfloor M/2 \rfloor$;
           allocate space for matf and matb;
($\alpha$)       $Costonly\_star\_align(A_{i^*}, B, C, matf)$;
($\beta$)       $Costonly\_star\_align(rev(A^T_{i^*}), rev(B), rev(C), matb)$;
($\gamma$)       Find $j^*$ and $k^*$ minimizing $(matf(j,k) + matb(N-j, K-k))$;
           Free the space of matf and matb;
($\delta$)       $cost1 = findcenter(A_{i^*}, B_{j^*}, C_{k^*})$;
($\epsilon$)       $cost2 = findcenter(A^T_{i^*}, B^T_{j^*}, C^T_{k^*})$;
           output $cost1 + cost2$.
      end
end
```

Figure 2.5: Algorithm **find_center_string_star_align.**

$c_1(M/2)[jk + (N - j)(K - k)] + c_2[jk + (N - j)(K - k)]$, which is no more than $c_1(M/2)NK + c_2NK$. It follows by induction that the total time taken in the worst case, including recursive calls and boundary cases, is no more than

$$c_1MNK(1 + \frac{1}{2} + \frac{1}{4} + \cdots) + c_2(\log M)NK + O(M + NK),$$

which equals $2c_1MNK + c_2(\log M)NK + O(M + NK)$. Therefore the time required for algorithm *find_center_string_star_align* is approximately twice that for the cost-only version *costonly_star_align* .

We have developed a C-implementation of both algorithms shown in Figure 2.4 and 2.5. To compare the actual time spent on *costonly_star_align* part and the time spent on *find_center_string_star_align* part, we have conducted tests on random sequences with certain sizes. The results are listed in Figure 2.6.

sizes	cost-only	find-center-string	ratio (f/c)
(146, 17, 42)	3	10	2.3333
(100, 25, 50)	4	12	2.0000
(100, 50, 50)	9	15	1.6667
(100, 100, 100)	41	58	1.41463
(200, 200, 200)	330	456	1.38182
(400, 400, 400)	1457	1953	1.34043
(800, 800, 400)	11616	13798	1.18784

(Unit: second)

Figure 2.6: Test results of program star.

In the above discussion we did not consider gap costs. When gap costs are involved, the situation is much complicated. If the recurrence relation $Cost(i, j, k)$ is to be re-defined to include the consideration of gap costs, we will have to not only consider the values of $Cost(i - \delta_1, j - \delta_2, k - \delta_3)$, but also analyze the ending pattern of the partial alignment of $(A_{i-\delta_1}, B_{j-\delta_2}, C_{k-\delta_3})$. Define the *history* of a partial alignment to be the amount of information necessary to determine the cost of any possible extensions. To find an optimal alignment it is necessary in general to know, at each node, the minimum cost of the partial alignment in each historical situation. Alstchul [1] presents a general analysis of gap costs for tree and star-alignments and infers that *the number of relevant histories for star-alignment of n input strings using gap costs is*

$$2^n - 1 + \sum_{i=0}^{n-1} \binom{n}{i} 3^i = 4^n - 3^n + 2^n - 1.$$

In our case, $n = 3$, so there are 44 histories to be considered for each (i, j, k).

2.4 Implementation

Since pairwise alignments have been well studied, many excellent implementations and strategies have been developed to strive for higher speed and less space. Realizing that space may be the limiting factor for our applications, we adopt the approach introduced by Myers and Miller [14]. The method can construct an optimal pairwise alignment in linear space.

There are two steps for performing a tree alignment with a given phylogeny. As the first step, initialization of internal nodes is conducted by following the algorithm shown in Figure 3.3. Then local optimalization is performed on each internal node. More discussion about implementation of the initialization algorithm can be found in section 4.5.

Chapter 3
Graphical User Interface

In computer systems, the user interface is considered as the mechanism through which a dialogue between the computer and the user is established. It plays a vital part in the computer system's efficiency. The speciality called Human-Computer Interaction (HCI) has emerged as the study of people, computer technology and the ways these influence each other. Both the developers of computer systems and users are starting to accept that just being able to do a task on a computer is not the only important factor. The question 'Can this goal be achieved with a computer?' is starting to be replaced by the question 'How easily can the user achieve the goal using the computer?'. The interface is in many ways the "packaging" for a computer system. If it is easy to learn, simple to use, straightforward, and forgiving, the user will be inclined to make good use of what is inside.

In this chapter, we will briefly review interface style and development issues in general and then review the architecture of the X Window System and its fundamentals which are related to this project.

3.1 Interface Development

The term "user interface" can be defined as the software component of an application that translates user's actions into requests for functions, and that provides to the user feedback about the consequences of his/her action.

A good user interface should provide an end user with a facile, natural environment for conducting various tasks fast, efficiently, accurately, and inexpensively. The nature of the software component of the user interface has been driven and limited by the hardware component. As hardware has become more sophisticated, options for interaction style have grown.

The following are some of common interface styles:

- **command and query interface**: Communication is purely textual and is driven via commands and responses to system-generated queries.

- **menu interface**: The set of options available to the user is presented on the screen. An option is selected by either using the mouse or typing some key. Since the options are visible, they are less demanding on the user, relying on recognition rather than recall.

- **form-fills and spreadsheets**: The user is presented with a display comprising a grid of cells, each of which can contain a value. This type of interface is used primarily for data entry and data analysis applications.

- **WIMP interface**: This type of user interface is characterized by windows, icons, menus and pointing devices. It is the default interface style for the majority of interactive systems in use today. The important features are (1) displaying different types of information simultaneously; (2) enabling the user to switch context without losing visual connection with other work; (3) enabling the user to perform various tasks in a facile manner; (4) increasing the interaction efficiency.

Each of them is encountered across every application area. The trend is toward multitasking, window-oriented, and point and pick interfaces. Ideally, users can customize the interface to suit their working style, rather than adapting their own working style to accommodate the interface's way of doing things.

The most important and natural method for a user interface development is the iterative development methodology, which includes building one or more prototypes to get requirement specification and comments from clients. To make the user interface easier to program, many different kinds of tools have been created. These include window systems, toolkits, interface builders, and user interface management systems.

The survey conducted by Myers and Rosson [15] seems very interesting. It has shown that in today's applications, an average of 48% of the code is devoted to the user interface portion. The average time spent on the user interface portion is 45% during the design phase, 50% during the implementation phase, and 37% during the maintenance phase.

3.2 The X Window System

The X Window System is an industry-standard software system that allows programmers to develop portable graphical user interfaces.

The X Window System's architecture is based on the client-server model. A single process, known as the *server*, is responsible for all input and output devices. An application that uses the facilities provided by the X server is know as a *client.* The syntax and semantics of the conversation between servers and clients are defined by *X Protocol.* Clients use the protocol to send requests to the server to create and manipulate windows, to generate text and graphics, to receive input from the user, and to communicate with other clients. The server uses the protocol to send information back to the client in response to various requests and to deliver keyboard and other user input on to the appropriate clients. The X Window System allows clients to be run on any machine in a network, and be displayed on any other machine(s) in that network.

The X protocol has been implemented with a library so that application programmers do not have to think in low level terms. This library provides a procedural interface that conceals many of the details of the protocol. Various utility functions are also provided that are not protocol-related but important in building applications. The exact interface for the library may differ for each programming language. The C libraries are the most widely used. They include a low-level procedural interface to the X protocol called *Xlib*, which defines an extensive set of functions that provide complete access and control over the display, windows, and input devices.

Although programmers can use Xlib to build applications, this relative low-level library can be tedious and difficult to use correctly. Many programmer prefer to use the higher-level X Toolkit to mask some of the complexity of the X protocol. The X Toolkit consists of two parts: a layer known as the *Xt Intrinsics*, and a set of user interface components known as *widgets*. A widget set implements user interface components, while the Xt Intrinsics provides a framework that allows the programmer to combine these components to produce a complete user interface.

There are several widget sets provided by system vendors to implement their particular user-interface styles. This project uses the widgets from the OPEN LOOK Intrinsics Toolkit (OLIT). The OLIT, based on the Xt Intrinsics from MIT, is one of the three GUI toolkits from OpenWindows. The OpenWindows environment supports the OPEN LOOK Graphical User Interface, which specifies windows and menus with common graphic symbols so that users are presented with a consistent screen layout that can be used across various platforms and operating systems.

Both the Xt Intrinsics and the OLIT widget set are written in C and built on the top of Xlib. Applications often use Xlib, the Xt Intrinsics, and the OLIT widget set as a complete system for constructing user interfaces. Figure 3.1 shows the architecture of an application based on a widget set and the Xt Intrinsics.

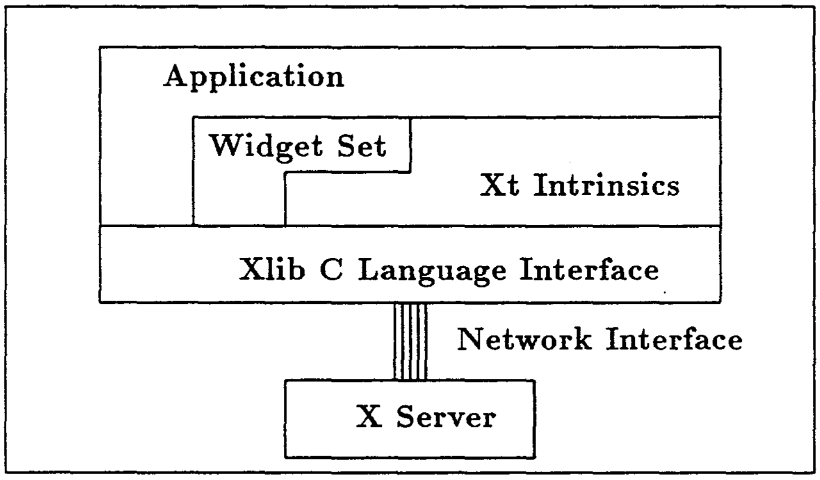


Figure 3.1: Programmer's view of the X Window System.

An important X concept which needs to be introduced here is the *event*. An event is a notification, sent by the X server to a client, that some condition has changed. The server generates events as a result of some user input, or as a side effect of a request to the X server. The server sends each event to all interested clients, who determine what kind of event has occurred by looking at the type of the event. To receive events, applications must specifically request the X server to send the types of events in which they are interested. Most X applications are completely *event-driven* and are designed to wait until an event occurs, respond to the event, and then wait for the next event. The event-driven approach provides a natural model for interactive applications. The user does not need to navigate a deep menu structure and can perform any action at any time. The user, not the application, is in control. The application simply performs some setup and goes into a loop from which application functions may be invoked in any order as events arrive.

Managing resources is an important part of programming with X. Resources are named data units that specify widget attribute values such as colors, fonts, images, text, positions and sizes of windows, or any customizable parameter that affects the behavior of the application. Resources can be set in four ways:

- In the application code when/after the widget is created.
- Through the resource database.
- In a command line option.
- Dynamically while the application is running.

If a resource is not set in any of these ways, OLIT will set the resource to a default value. Setting resources in source code is considered as *hard.* Users cannot customize hard coded resources unless the source code is modified and recompiled. Therefore, the application programmer should only set the resource values in program for the resources that are not allowed to be changed by a user. To make programs customizable, a good approach is to provide an application default resource file for every program so that uses can customize an application by simply changing the appropriate entries in the resource file.

When an OLIT program is initialized, the connection to the X server is set and the **resource database** is created and embedded with the program. The resource specifications in the user's resource files are loaded into the resource database. The four resource files are the *application defaults file* whose path can be identified by the environment variable **XFILESEARCHPATH,** the *per-user application defaults file* whose path is identified by the environment variable **XUSERFILESEARCHPATH** , the *user's defaults* which is the file **.Xdefaults**, and the *user's per-host defaults* whose path can be identified by the environment variable **XENVIRONMENT.**

Chapter 4
Project Development

This project is an application based on OPEN LOOK Intrinsics Toolkit, which follows an object-oriented and event-driven model. In general, such an application consists of three parts:

1. creating and manipulating OLIT widgets to build the desired user interface;

2. using C/C++/Xlib to develop the application code, i.e., the code that performs the actual work on the application's data;

3. attaching the application code to the user interface via callback procedures and event handlers that are executed when the user performs some action on a widget. Attaching specific procedures to specific widgets allows programmers to produce modular source code.

4.1 Interface Structure

In this section, we discuss the widget hierarchy used in SAT. As in Figure 2.2, we can divide the initial screen of SAT into four areas: **Control Panel**, **Canvas**, **Message Panel**, and **Information Panel**. The hierarchy structure is depicted in Figure 4.1 and explained below. The names of OLIT widget classes are underlined.

The **toplevel** is a shell widget, created by the call to initialize the Toolkit. A shell widget must have exactly one child widget. The shell widget serves as a

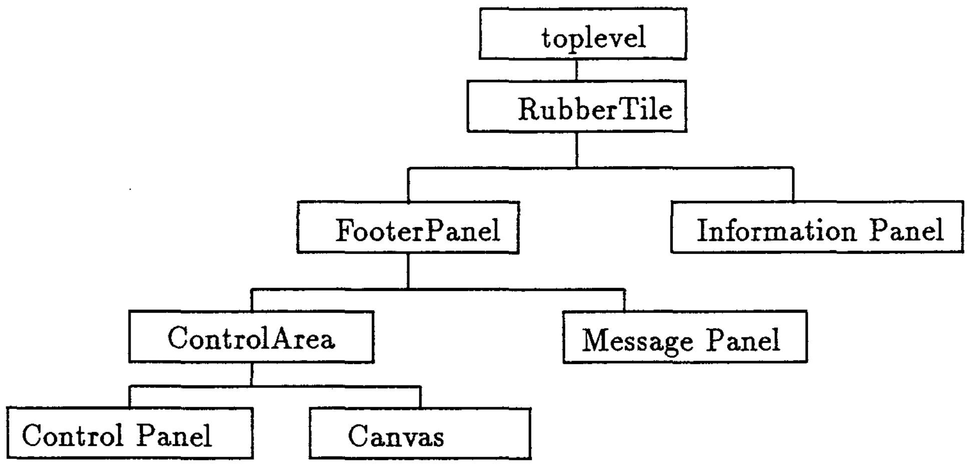


Figure 4.1: Primary structure of SAT.

wrapper around its child, providing an interface between the child widget and the window manager. The RubberTile widget, as the only child of **toplevel**, manages two children widgets in a row. Relative weights can be assigned to each child so that it expands or contracts a certain percentage of size changes of the RubberTile. The FooterPanel widget attaches a footer at the bottom of a window so that various messages can be displayed in the footer area. The ControlArea widget manages two important parts: **Control Panel** and **Canvas.**

The **Control Panel** contains six MenuButton widgets: **File, Check, Undo, Parameters, Mode** and **Calculation**, managed by a ControlArea widget. Each of them has its own pop-up menu, containing further OblongButton widgets through which users call application functions. SAT uses callbacks to link these widgets with application functions.

The **Canvas** is implemented as a DrawArea widget. This is the window where a graph is created, manipulated and displayed. In this area, the events *ButtonPress,* *ButtonMotion, ButtonRelease* are all the events in which this application is interested. *Event Handlers* are invoked by the Xt Intrinsics when a specific type of event occurs. The event type and associated application function are registered using the *XtAddEventHandler* function.

The **Message Panel** contains two StaticText widgets to display messages.

The **Information Panel** is made of a ControlArea widget managing two Caption widgets, a ScrollingList widget and a OblongButton widget in a column. The Caption widgets are used to create labels for their child widgets. In our case, one StaticText widget shows the number of sequences already loaded and the other shows the total score after an alignment is conducted. The scrollingList widget is able to display a list of items in a scrollable pane and provides a sophisticated set of widget-defined functions for manipulating the items in the list.

Moreover, there are other interface areas brought up by PopupWindowShell and NoticeShell widgets. They represent a way of using widgets. Popup widgets are subclassed off the TransientShell class. Popups are not visible until a certain user command is given, or a situation arises in which the program requires user input.

In SAT, Popup widgets are used for a variety of purposes including (1) prompting for input/output file names; (2) setting parameters for calculation; (3) displaying sequences and alignments; and (4) getting confirmation from users if some "dangerous action" happens such as pressing the **Clear** button.

4.2 Displaying Graphs

The SAT system contains a graph editor. The user is able to edit a (undirected) graph in the drawing area in which vertices are represented by circles and edges are represented by lines.

In X Window programming, drawing things like points and lines can be easily done by first creating a *graphics context* (GC) and then calling Xlib graphics functions. The GC is an X Window System resource which contains 23 distinct attributes to specify things like color and line width. When one object could be overdisplayed by another, the GC's *GCFunction* attribute should be considered since it specifies how each pixel of a new image is combined with the current contents of a destination. This attribute is commonly set as the XOR mode so that drawing a figure twice restores the screen to its original state. The SAT system uses this property to erase an image and perform rubber banding operations.

When drawing an object, the coordinates have to be specified in pixel units. Coordinates are always relative to the upper left corner of a drawable window. The x coordinate increases toward the right and the y coordinate increases downwards.

In the following, we explain how vertices and edges are displayed in SAT.

A **vertex** is represented on the screen as a circle (also called as a node) with a radius of r (= 5 pixels). When a vertex is created by clicking the mouse, the circle is displayed such that its center is at the position of the *hotspot* of the cursor. The center's coordinates are also called the coordinates of the vertex and stored as a part of the internal representation of the vertex.

An edge is represented on the screen by a line connecting two nodes. Internally, it is represented by a relation between two vertices. Notice that we cannot simply draw a line connecting the centers of two nodes since the overlapping parts between the line and nodes will not be displayed properly. The natural choice is to choose a boundary point from each node. However, to ease the calculation involved, it is much easier to simply pick up a boundary point from the surrounding rectangle of each node. Therefore we choose the two ending positions of the line as follows. Assume two vertices are of coordinates (x_1, y_1) and (x_2, y_2) and displayed as nodes of radius r. Define the slope

$$k = \frac{y_2 - y_1}{x_2 - x_1}.$$

Define $\sigma_x = 1$ if $x_1 \leq x_2$ and -1 otherwise, and $\sigma_y = 1$ if $y_1 \leq y_2$ and -1 otherwise.

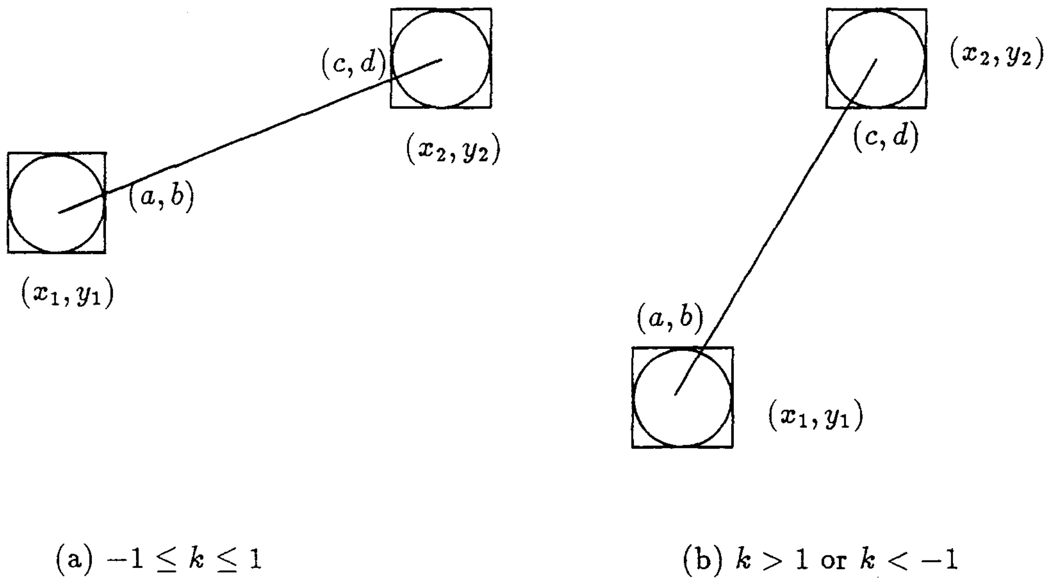


Figure 4.2: Deriving the line for an edge.

If $-1 \leq k \leq 1$, as shown in Figure 4.2(a), we calculate

$$a = x_1 + \sigma_x r; \quad b = y_1 + \sigma_x r k; \quad c = x_2 - \sigma_x r; \quad d = y_2 - \sigma_x r k$$

If $k > 1$ or $k < -1$, as shown in Figure 4.2(b), we calculate

$$a = x_1 + \sigma_y \lfloor \frac{r}{k} \rfloor; \quad b = y_1 - \sigma_y r; \quad c = x_2 - \sigma_y \lfloor \frac{r}{k} \rfloor; \quad d = y_2 + \sigma_y r$$

Then the line connecting points (a, b) and (c, d) is displayed as the edge connecting vertices (x_1, y_1) and (x_2, y_2).

Once nodes and lines are drawn, information about them has to be saved by the application program since the workstation has no memory of the fact that something is drawn.

Manipulation of vertices and edges is conducted by first selecting corresponding nodes and lines. Then procedures are invoked by event handlers to perform some actions. From the user's point of view, a selection can be done by moving the cursor onto nodes and lines displayed on the screen and then pressing/releasing a mouse button. However, from the programmer's point of view, the program has to do (maybe a lot of) background computation to compare the user-selected position of the cursor with the coordinates of each vertex and see if the difference is small enough (predefined). If we imagine vertices and edges as objects in an object-oriented model, then every object will (the frequency depends on the types of events registered) receive messages concerning whether the hotspot of the cursor is within its neighbourhood and the object with a confirmative answer is selected to accept some actions. How is the neighbourhood defined for a vertex and for a edge? Assume a vertex has (a, b) as its coordinates, then its neighbourhood is defined as $\{(x, y) | \; |x - a| < r \text{ and } |y - b| < r\}$ ($r = 5$ pixels in SAT). The neighbourhood of an edge, requiring much more computation, is defined as a $2 * r$-width band surrounding the line segment. Assume the two ends of the line are (a, b) and (c, d) as shown in Figure 4.2, then the line's algebraic equation is

$$\frac{y-b}{x-a} = \frac{b-d}{a-c},$$

which can be re-written as

$$(b-d)x + (a-c)y + cb - ad = 0.$$

The distance between a point (x_0, y_0) and the line is calculated by

$$\text{dist}(x_0, y_0, a, b, c, d) = \frac{|(b-d)x_0 + (a-c)y_0 + cb - ad|}{\sqrt{(b-d)^2 + (a-c)^2}}.$$

Now we can define the neighborhood of the edge (see Figure 4.3) to be

$$\{(x,y) | \text{dist}(x, y, a, b, c, d) < r,\ \min(a,c) < x < \max(a,c), \min(b,d) < y < \max(b,d)\}.$$

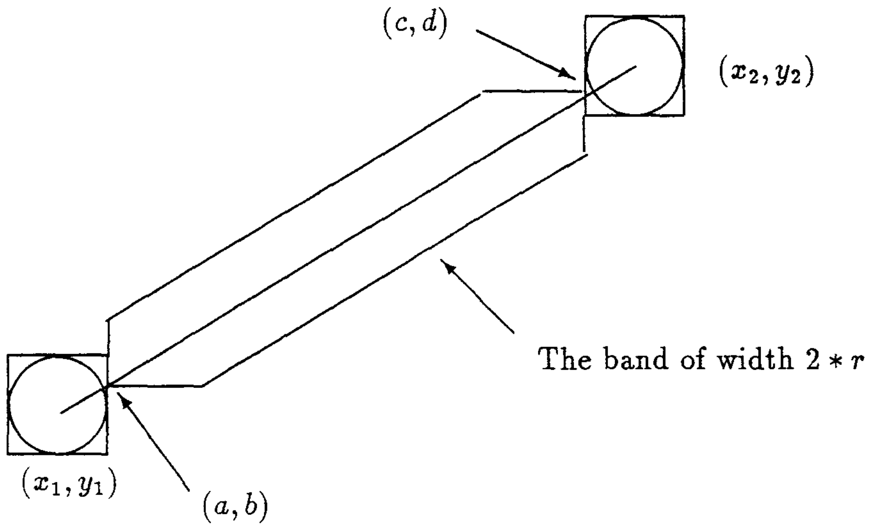


Figure 4.3: The neighbourhood of an edge.

4.3 Data Definition

In this section, we discuss the primary data structures used in SAT.

1. Sequence.

The common set of information of sequences includes *the name*, *a short description*, *the sequence itself*, *format* (how the sequence is kept in a file), and *the category of the sequence.* Therefore we use the data structure shown in Figure 4.4 for the type of sequences. Loaded sequences will be stored in **SEQArray**, which is

```
typedef struct {
        string   name;
        string   description;
        char     *content;      /* sequences */
        short    format;        /* FASTA or NBRF/PIR*/
        Boolean  isdna;         /* DNA/RNA or Protein? */
} SEQtype;
```

Figure 4.4: The structure for sequences.

declared as an array of *SEQtype.*

2. Graph.

The graph under the consideration is undirected and weighted. Information of a vertex includes its coordinates at the drawing area and the sequence associated with it. Knowledge of an edge covers the information of the associated alignment of two sequences assigned to the ends.

We use the *adjacency lists* structure to represent a graph. That is, each vertex is associated with a linked list consisting of all the edges adjacent to this vertex, and all vertices are stored in an array. The data types, shown in Figure 4.5, are used to describe the adjacency lists structure. The meaning of each member is easy to understand with the given comments.

```
typedef struct _EDGE {
            int             index;    /* index of the vertex */
            float           weight;   /* cost of the alignment */
            char            *seq1;    /* padded sequence_1 after alignment*/
            char            *seq2;    /* padded sequence_2 after alignment*/
            PARAtype        *para;    /* parameters for doing alignment */
            struct _EDGE    *next;    /* next edge */
} EDGEtype

typedef struct _VERTEX {
            int             _x;       /* x-coordinate */
            int             _y;       /* y-coordinate */
            SEQtype         *con;     /* sequence assinged to the vertex */
            EDGEtype        *head;    /* edge list associated to the vertex*/
} VERTEXtype;
```

Figure 4.5: The adjacency lists structure.

Considering the size limitation of the display screen, it is unlikely the user will draw a graph with over 100 vertices. So, in this package, the number of vertices of a graph is restricted to 100. Of course, this number is easily adjusted by redefining a constant and recompiling the program. During processing, users can handle only one graph at a time. The graph is represented by **NODEArray**, which is initialized statically by declaring an array, with size 100, of VERTEXtype. The offset of each element in the array coincides with the vertex label. This permits direct access to vertex data and thus reduces the searching operations which would otherwise be used so frequently in many functions. The variable **node_num** is used to keep the number of vertices of the graph. Existing vertices are always kept in the first **node_num** cells of **NODEArray**.

The insertion/deletion operations are illustrated as follows:

- add a vertex: coordinates and/or a sequence of the vertex are inserted into the **node_num**-th cell of **NODEArray** and **node_num** increments by 1.
- add an edge: Assume it connects *i*th and *j*th vertex. Then processing will go through allocating memory space for one EDGEtype variable, filling it with the label of *j*th vertex and other edge-related information, and inserting it to the beginning of the linked list (edges) associated with *i*th vertex. The similar processing is repeated with the exchange of i and j in the above sentence.
- delete an edge: The linked list associated with each of the two ends is searched. The involved items, one from each list, are removed and their memory location become free.
- delete a vertex: All edges connecting to the vertex are deleted. The last vertex is moved to the cell of the deleted vertex. The linked lists of the vertices adjoining to the former last vertex are updated. **node_num** decrements by 1.

Information of an edge is stored in two lists so that it is very convenient to design and implement other edge-related procedures.

4.4 Undoing Facility

Undoing and redoing are basic facilities that should be provided by any good interactive system. These facilities allow users to cancel a command, to recover from operating mistakes that may be damaging and also allow users to do input testing, knowing that they can back up easily if the result is not what they expect.

To perform an Undo operation, something has to be saved. For example, we cannot undo a delete operation unless we have stored the deleted material. Therefore a natural question is how to save information efficiently.

In this project, we developed an one-level undo facility for the graph editor. Any editing operation can be undone up to one level. Executing undo twice successively will let system go back to the state as if the undo operations were not executed.

We are interested in six types of editing operations. They are *moving a node, changing the content of a node, inserting a node, deleting a node, inserting an edge and deleting an edge*. According to the actual amount of information to be saved, we define the following data type, as shown in Figure 4.6, to guide information saving. The **union** construct is used to allow storage sharing and a member called *flag* is used for the interpretation of the stored information.

A memory space **undo_buffer** is used to save editing information. When an editing operation is performed, we first check which part of the graph will be changed and old data is saved in **undo_buffer**, and then the graph is updated as required.

For the undo command, the most expensive editing is the deletion of a node, which involves deleting the node, the edges connecting to it, and rearranging the vertex array of the graph. To save all this information, the space of **undo_buffer** is not enough. So we dynamically allocate memory for this purpose.

4.5 Communication

One of goals of this software is to provide a tool for testing and analyzing algorithms related to sequence alignment. One way of doing this is to implement new algorithms

```
typedef struct {
        UNDOTYPE      flag;           /* one of six types */
        union {
          struct {                    /* node is moved */
            int       index;
            int       x, y;
          } move_node;
          struct {                    /* node's content is changed */
            int       index;
            SEQtype   *content;
          } change_con;
          struct {                    /* changing edge */
            int       index;
            int       index2;
            float     weigh;
            char      *seq1, *seq2;
            PARAtype  *para;
          } re_edge;
          struct {                    /* changing node */
            int       index;
            NODEtype  node;
          } re_node;
        } set;
} undo_type;
```

Figure 4.6: Structure for storing editing information.

as modules and add them to the code of SAT. The obvious disadvantage of this method is that the internal structure of SAT has to be understood and the source code is subject to change. Alternatively, researchers can implement programs separately, and then employ SAT as a graphical user interface. That is, users can use SAT to draw a graph, load sequences, connect to a computing program and send all the data to the program. After computation is done, all results are sent back and displayed in SAT. In this way, the development of a new program does not involve the internal structure of SAT and will not affect SAT in any way. For this to work, we need a communication protocol with which SAT and other programs can talk to and understand each other.

First, we design an abstract data type **HEADERtype**, similar to the type defined in Figure 4.5, for representing a graph structure and the sequences/alignments information associated with it. Then we develop a set of functions which support loading variables of type **HEADERtype** from formatted files and storing data of the type into files with a given format. The type **HEADERtype** is based on the concept of adjacency lists, a very common data structure for graphs. This should be useful in implementation of algorithms dealing with evolutionary tree construction problems. The most important point is that a program supporting data of type **HEADERtype** can communicate with SAT.

As an exercise, we have developed a program, called **tree**, based on the tree initialization algorithm as shown in Figure 2.1. The organization of the program **tree** is shown in Figure 4.7.

Steps 1, 2 and 4 can be easily achieved by calling functions defined in our library, which are also used in SAT for input/output and retrieving /storing data. Therefore the same set of functions for handling input/output provides a base for the communication between SAT and **tree**. The third step of **tree** is allowed to be extended and modifyed to do more task without worrying about SAT. Now the relation between these two can be described as:

- Use SAT to create a graph with certain information;
- Use the **fork**, **exec** and **pipe** Unix system calls to create a new process, to run **tree** on the new process and to open a communication channel so that data of SAT "flow" over to **tree**. The functions designed for getting input/output file

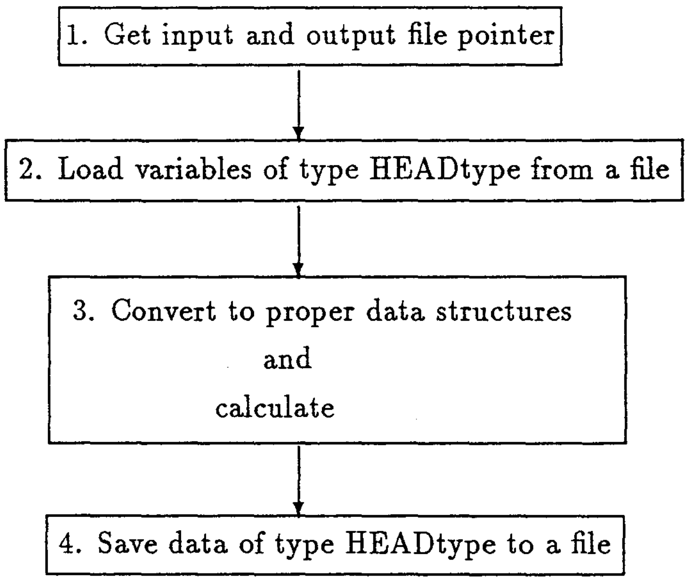

1. Get input and output file pointer
2. Load variables of type HEADtype from a file
3. Convert to proper data structures and calculate
4. Save data of type HEADtype to a file

Figure 4.7: The organization of **tree**.

pointers include manipulation of pipes.

- **tree** gets data from the pipe, performs the calculation and puts resulting data back to the pipe.

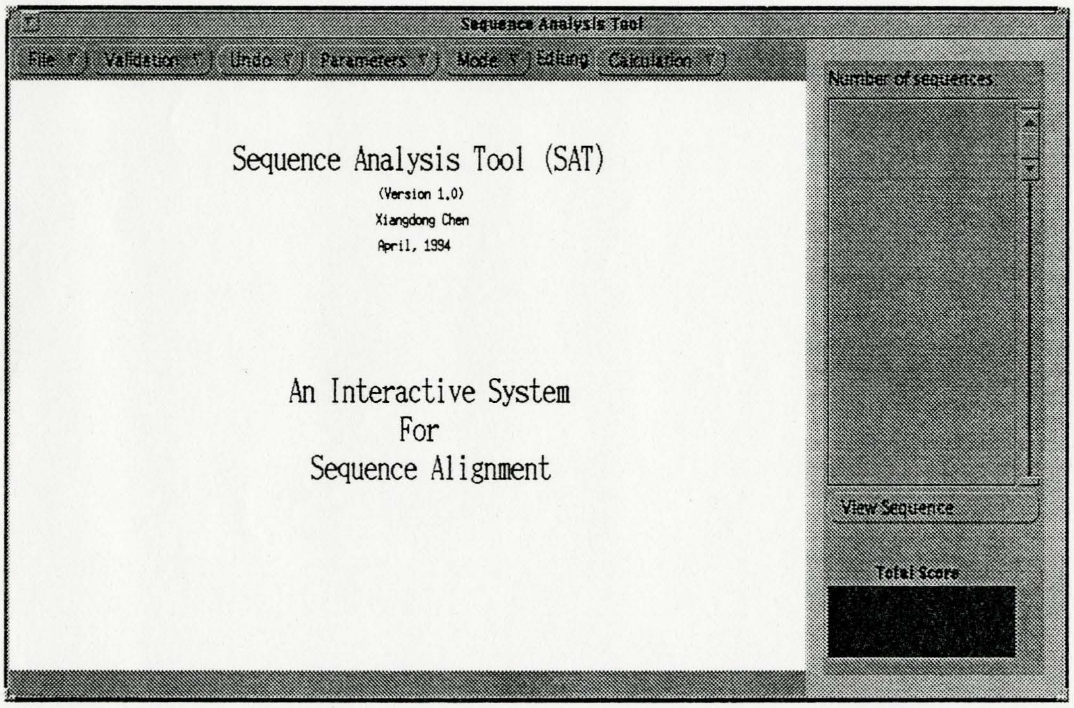
- SAT gets data from the pipe and displays the data.

The successful separation and communication between SAT and **tree** sets a model as how to use SAT as a friendly user interface to support related programs. This gives our system some flavour of client-server computing.

Chapter 5
Sequence Analysis Tool

This chapter is a brief introduction to Sequence Analysis Tool (SAT). To produce a good interface display, it is the best to run the program in the OpenWindows environment on a Sun SPARCstation with a color monitor.

5.1 Getting Started



Sequence Analysis Tool (SAT)
(Version 1.0)
Xiangdong Chen
April, 1994

An Interactive System
For
Sequence Alignment

Figure 5.1: Initial screen of SAT.

To get started, move to the directory where SAT is installed and type

```
setenv XENVIRONMENT Resources
sat
```

Figure 5.1 shows the initial screen with an introduction message. Pressing any mouse button in the window where the message is displayed will erase the message and set the system ready to work for you. If the screen is not displayed properly, check whether the file *Resources* is included in the directory and check the value of the XENVIRONMENT variable.

5.2 Screen Display

To introduce the features, we divide the SAT screen into four main areas as shown in Figure 5.2. The Control Panel provides six menu-buttons, each has a menu associated

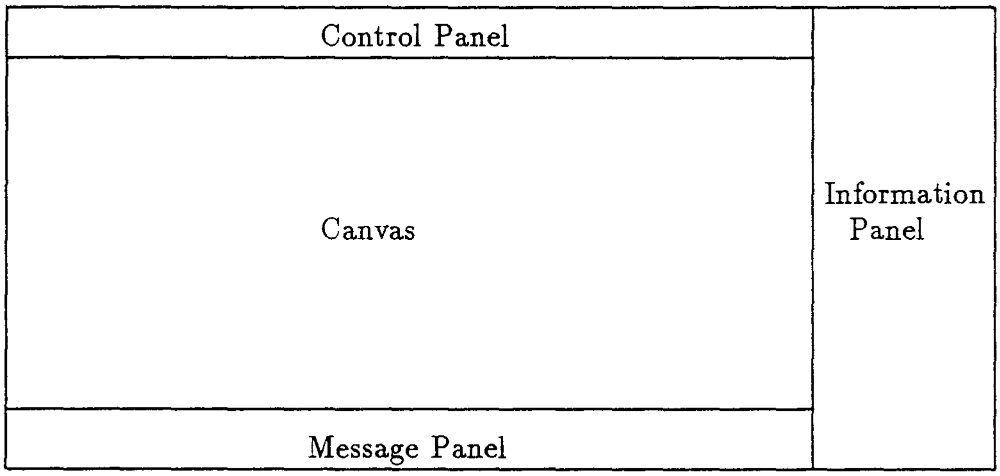


Figure 5.2: The main screen structure of SAT.

with it. The user chooses a menu item to invoke an action or set the mouse to a working mode. The Canvas takes up most of the screen. It is the area where graphs are composed and manipulated, and relevant data are displayed. The Information Panel shows the number and the names of loaded sequences, and displays the score of an alignment. The Message Panel is the place where SAT displays messages responding to the user's operations. In the following sections, we will show how to operate SAT and some input requirements.

Since SAT is mouse-operated, it is important to be familiar with the functions of each mouse button. Following the **OpenWindows Version 3 User's Guide**, we will refer to mouse buttons by functions, that is,

1. SELECT = the left mouse button
2. ADJUST = the middle mouse button
3. MENU = the right mouse button.

But we will also refer to mouse buttons by their positions when it is more convenient.

5.3 Canvas and the Representation of Graphs

On the Canvas a graph is represented as a collection of nodes and edges. In the context of sequence analysis, a node is assigned a sequence and an edge is assigned an alignment of the two sequences associated with the two end nodes. In order to let the user visualize the assignment, we use the following display strategy.

A small circle is used to denote a node, called an *empty* node, which has not been assigned any sequence. If a sequence is assigned to a node, the node becomes solid, also called a *full* node. The sequence's name is displayed above the node .

An edge has a dark color if it has been assigned a pairwise alignment. Otherwise an edge has a light color.

Information about a sequence includes

- sequence description.
- sequence name.
- sequence storing format.
- DNA/RNA or Protein.

Information about a pairwise alignment includes

- score.
- length.
- the number of matches.
- the number of mismatches.
- the number of gaps.
- the number of insert/delete operations.
- the actual alignment.

How to manipulate a graph, assign sequences and alignments and view the information will be shown later.

5.4 Command Panel

This panel contains six menu-buttons. Press MENU on a menu-button to bring up its menu. To choose an item on the menu, simply click SELECT on the item and then its associated function is invoked. Directly press SELECT on a menu-button to select the default menu item.

1. **File Menu** contains items to handle loading a sequence/graph file, saving a graph file and quitting SAT.

    - Press MENU on **Load**, and then select the **Sequence** menu item to display the Load Sequence File window, through which a sequence file in any directory can be selected. Then the sequences are loaded and their names are displayed in the Information Panel. The format of a sequence file is explained in Section 5.10.
    - Press MENU on **Load**, and then select the **Graph** menu item to display the Load Graph File window, which is similar to the Load Sequence File window. The graph files should have been generated by SAT.
    - Selecting the **Save** menu item displays a window prompting for a file name to save the information about the graph shown on the screen, and the sequences and alignments assigned to the graph. No restriction has been set for choosing a file name.
    - Select the **Quit** menu item to quit SAT.

2. **Validation Menu** contains items to verify whether the drawn graph is a tree or connected. The **Check** menu item is used to print internal data for debugging purposes.

3. **Undo Menu** contains **Undo Last Action** and **Clear** commands.

    - The **Undo Last Action** command will reverse the effect of the last graph editing command issued.

- The **Clear** command will erase all objects in the Canvas. A window will show up to get the use's confirmation.

4. **Parameters Menu** contains one command which will display a window containing three fields (1) file of score matrix; (2) gap-open penalty and (3) gap-extension penalty. Without a valid setting of these values, the system will refuse to perform alignment operations. For the file format of a score matrix, refer to Appendix C.

5. **Mode Menu** contains five menu items. Each assigns a mode to the mouse so that it can play many different roles in Canvas. The name of the selected mode is displayed on the right side of the menu-button to remind the user of the current mode of the mouse. The five modes will be explained in next five sections. When designing mouse buttons to deal with nodes and edges, the general principle is that the left button manipulates nodes and the middle button manipulates edges.

6. **Calculation Menu** contains four commands.

   - The **Alignment** command calculates an optimal alignment of the two sequences associated with each edge of the graph drawn in the Canvas. The total cost is displayed in the Information Panel.

   - The **Tree Align** command conducts a 3-star alignment on each node of degree 3 of the graph to have a local optimization. A star ("*") is appended to the name of each affected sequence. The user can tell how many times 3-star alignment has been conducted on a node by counting the number of stars appended to the name of the original sequence assigned to the node. Based on the modified sequences, an optimal pairwise alignment for each edge is conducted again so that the alignment associated with each edge is also updated, and the total score window is updated.

- The **WJ Method** command, for initializing the internal nodes of the graph (now it must be a tree such that each node has a degree of either 1 or 3), invokes the **tree** program and sends it all the sequences and the tree structure drawn in the Canvas. After the completion of the **tree** program, its output, including the initial sequences for internal nodes of the graph, is sent back to SAT.
- The **Others** command displays a window so that the user can specify a compatible program and execute it. A program is compatible when it is developed using several convenient input/output functions provided by this package and designed to communicate with SAT.

5.5 Manipulating Graphs

Select **Editing** from the **Mode Menu**. Now the mouse is ready for graph editing.

- Creating a node: Press/Release the left button. An empty node is created at the location of the mouse cursor when the button is released.
- Moving a node: Press the left button on a node, move the cursor around and release the button. The node is moved to the cursor's position when the button is released.

- Creating an edge: Press the middle button on one node, move the cursor and release the button when the cursor is on the other node.
- Deleting a node: Click the right button on the node.
- Deleting an edge: Click the right button on the edge.

Each node will have an index generated by the system. To reverse an editing action, select **Undo Last Action** command from the **Undo Menu.**

5.6 Assigning Sequences

Select **Labeling** from the **Mode Menu.** Now the mouse is ready for assigning sequences to nodes.

Press SELECT on an item in the scrolling list of Information Panel to select a sequence. Then click the left button on a node. Now the chosen sequence is assigned to the node. If the node is full before, the new sequence overwrites the old one.

To remove a sequence from a node, simply click the right button on the node.

5.7 Alignments

Select **Aligning** from the **Mode Menu.** Now the mouse is ready for alignment. Suppose the parameters for doing alignment have been set.

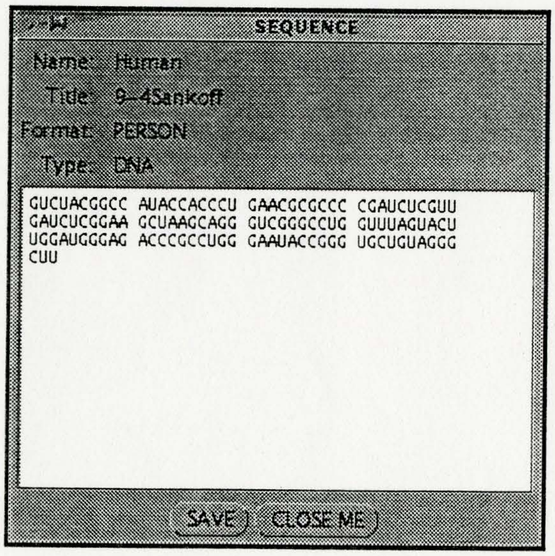
- Click the left button on a node of degree 3. A 3-star alignment on the node is conducted and the newly obtained sequence is assigned to the node. The new sequence's name is the concatenation of the old name with a "*".

- Click the middle button on an edge. An optimal pairwise alignment associated with the edge is performed. The edge is assigned the information of the new alignment.

5.8 Displaying Sequences and Alignments

Select **Display** from the **Mode Menu.** Now the mouse is ready for displaying sequences and alignments.

- Click the left button on a node to display the **SEQUENCE** window, which shows the information related to the sequence assigned to this node. The sequence itself is displayed on a text pane and is editable. If the sequence is modified and saved, a "?" is appended to the sequence's name. See Figure 5.3.

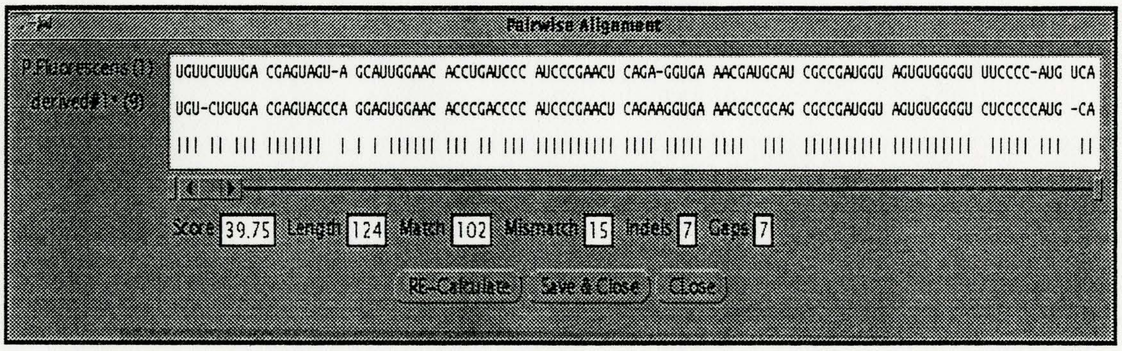


Name: Human
Title: 9-4Sankoff
Format: PERSON
Type: DNA

```
GUCUACGGCC AUACCACCCU GAACGCGCCC CGAUCUCGUU
GAUCUCGGAA GCUAAGCAGG GUCGGGCCUG GUUUAGUACU
UGGAUGGGAG ACCCGCCUGG GAAUACCGGG UGCUGUAGGG
CUU
```

Figure 5.3: Information about a sequence.

- Click the middle button on an edge to display the **Pairwise Alignment** window, showing the alignment information associated with the edge. The alignment is editable. See Figure 5.4.



```
P.Fluorescens(1)  UGUUCUUUGA CGAGUAGU-A GCAUUGGAAC ACCUGAUCCC AUCCCGAACU CAGA-GGUGA AACGAUGCAU CGCCGAUGGU AGUGUGGGGU UUCCCC-AUG UCA
derived#1*(9)     UGU-CUGUGA CGAGUAGCCA GGAGUGGAAC ACCCGACCCC AUCCCGAACU CAGAAGGUGA AACGCCGCAG CGCCGAUGGU AGUGUGGGGU CUCCCCCAUG -CA
                  ||| || ||| |||||||  | | | |||||| ||| || ||| |||||||||| |||| ||||| ||||  |||  |||||||||| ||||||||||  ||||| |||  ||
```

Score 39.75 Length 124 Match 102 Mismatch 15 Indels 7 Gaps 7

Figure 5.4: Information about a pairwise alignment.

5.9 Orienting a Tree

Select **Rooting** from the **Mode Menu**. Now the mouse is ready for orienting a tree, i.e., redrawing it as a rooted tree. If the system detects the graph is not a tree, it will refuse to be in this mode and reset to the **Editing** mode.

If the graph is a tree, press/release the left button on a node. Then a rooted tree will be drawn such that the selected node is the root and is at the cursor's position when the button is released.

5.10 File Format

For SAT to be able to read/write the files of sequences, alignments and graph structures, we need to specify some file formats.

Two types of files are used here. One is the pure sequence file, where sequences are listed in either *FASTA* or *NBRF/PIR* format, which are described later in this section. A sample file is shown in Appendix A.

Another format is designed to include information about sequences, alignments and graph structures. A file is divided into into 5 blocks. Block 1 consists of an integer and a four-bit number. The integer shows the number of vertices of the graph and the four bits show which of next four blocks are valid. Block 2 preceded by "/Edges" shows pairs of vertices and real numbers which represent edges and their weights. Block 3 starting with the key line "/Coordinates of nodes" shows the coordinates of vertices at the SAT's drawing area, i.e., Canvas. Block 4 preceded by "/Contents of nodes" contains sequences in the order of vertices. The recognized sequence formats include *FASTA* and *NBRF/PIR*. Finally Block 5, preceded by "/Information of edges", gives calculated pairwise alignments associated with the graph and other data. A sample file is shown in Appendix B.

FASTA (PEARSON AND LIPMAN, 1988) FORMAT: The sequences are delimited by an angle bracket ">" in column 1. The text immediately after the ">" is used as the name and the title. Everything on the following lines until the next ">" or the end of the file is one sequence. An example is

```
> RABSTOUT   rabbit Guinness receptor
   LKMHLMGHLKMGLKMGLKGMHLMHLKHMHLMTYTYTTYRRWPLWMWLPDFGHAS
   ADSCVCAHGFAVCACFAHFDVCFGAVCFHAVCFAHVCFAAAVCFAVCAC
```

NBRF/PIR FORMAT is similar to FASTA format but immediately after the ">", you find the characters "P1;" if the sequence is protein or "DL;" if it is nucleic acid. The text after the ";" is treated as the sequence name while the entire next line is treated as the title. The sequence is terminated by a star ("*") and the next sequence can then begin (with a >P1; etc). This is just the basic format description (there are other variations and rules). An example is

```
>P1;RABSTOUT
rabbit Guinness receptor
LKMHLMGHLKMGLKMGLKGMHLMHLKHMHLMTYTYTTYRRWPLWMWLPDFGHAS
ADSCVCAHGFAVCACFAHFDVCFGAVCFHAVCFAHVCFAAAVCFAVCAC*
```

Chapter 6
Testing

In this chapter, we present an example execution of SAT. The testing data (sequences, tree structure and score system) are chosen from the paper [18] of Sankoff, Cedergren and Lapalme. The input/output files and score matrix are presented in Appendices A, B and C. The test proceeds as follows.

1. **Load the sequences.**

   The required sequences with the FASTA format are manually typed in a file (see Appendix A). To load the sequences, click on

   $$\textbf{file} \longrightarrow \textbf{open} \longrightarrow \textbf{sequence}$$

   and then select the file from the newly popped-up window. Now sequences are loaded into SAT with their names displayed on the scrolling menu of the right hand side.

2. **Draw the tree.**

   The tree can be drawn easily by using graph editor facility of SAT.

3. **Assign sequences to the external nodes (leaves) of the tree.**

   Click on

   $$\textbf{Mode} \longrightarrow \textbf{Labelling}$$

to set the mouse mode so that, when an item is selected from the scrolling menu of sequence names and a node on the drawing screen is selected, the sequence is assigned to the node. Figure 6.1 shows the constructed phylogeny, whose external nodes have been assigned sequences.

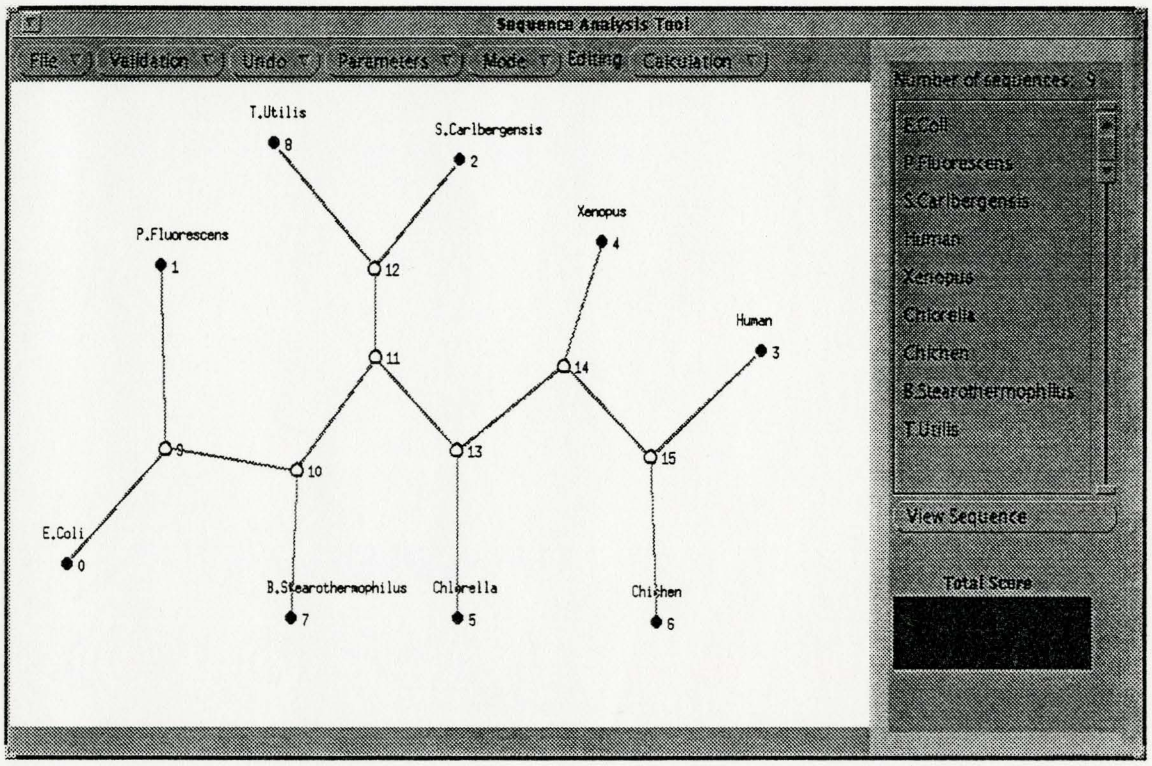
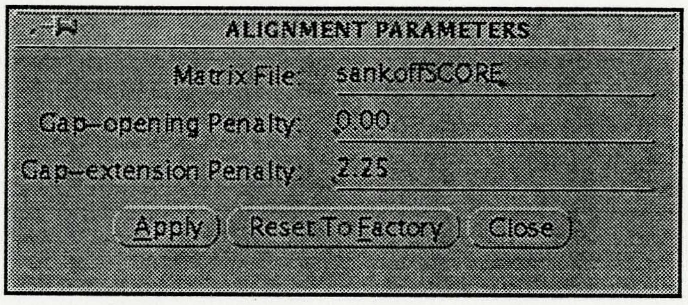


Figure 6.1: The phylogeny.

4. **Select parameters for the alignment.**

   Use the **Parameters** button to display the *ALIGNMENT PARAMETERS* window, then specify the score matrix file and values for gap penalties. See Figure 6.2. Refer to Appendix C for the score matrix file *sankoffSCORE*.



Matrix File: sankoffSCORE
Gap-opening Penalty: 0.00
Gap-extension Penalty: 2.25

Figure 6.2: ALIGNMENT PARAMETERS window.

5. **Run the program** "tree" to initialize the internal nodes.

   Click on

   Calculation $\longrightarrow$ **WJMethod.**

   Or click on

   Calculation $\longrightarrow$ **Others...**

   to display the *Calculation Method* window, then type (tree [option1] [option2]) to invoke the **tree** program such that an root is chosen on the edge of connecting the node of index *option* with that of index *option2*. All the data about the sequences and the tree structure are sent to the **tree** program, which will do calculation and construct sequences to be associated with internal nodes. Then results are sent back to SAT and the screen is updated.

6. Perform pairwise alignments on all edges

   Click on

   Calculation $\longrightarrow$ Alignment.

   After the pairwise alignment of the two sequences associated with each edge is completed, the current score of the tree is obtained and displayed in the *Total Cost* window. See Figure 6.3.

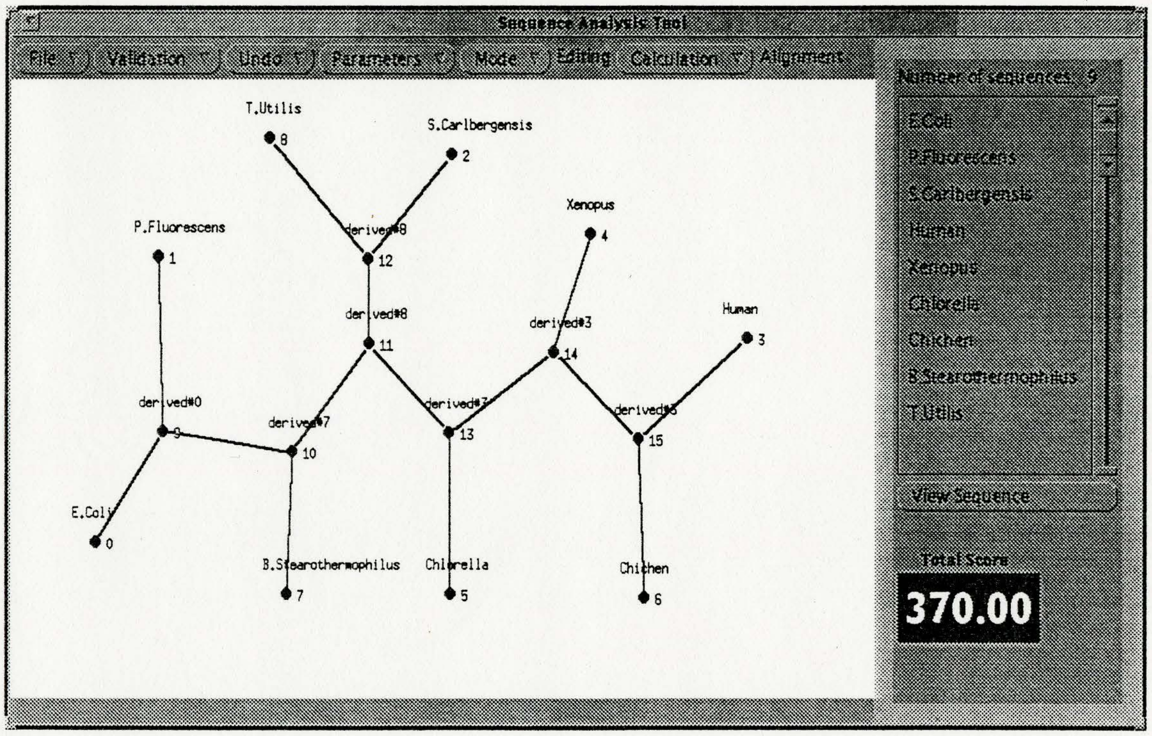


Figure 6.3: Alignment after initialization of internal nodes.

7. **Perform tree alignment**

   Click on

   Calculation $\longrightarrow$ **treealign.**

   A local optimization is conducted on each internal node once. A "*" will be appended to an internal node if a local optimization is done on it. The new score is displayed. See Figure 6.4.

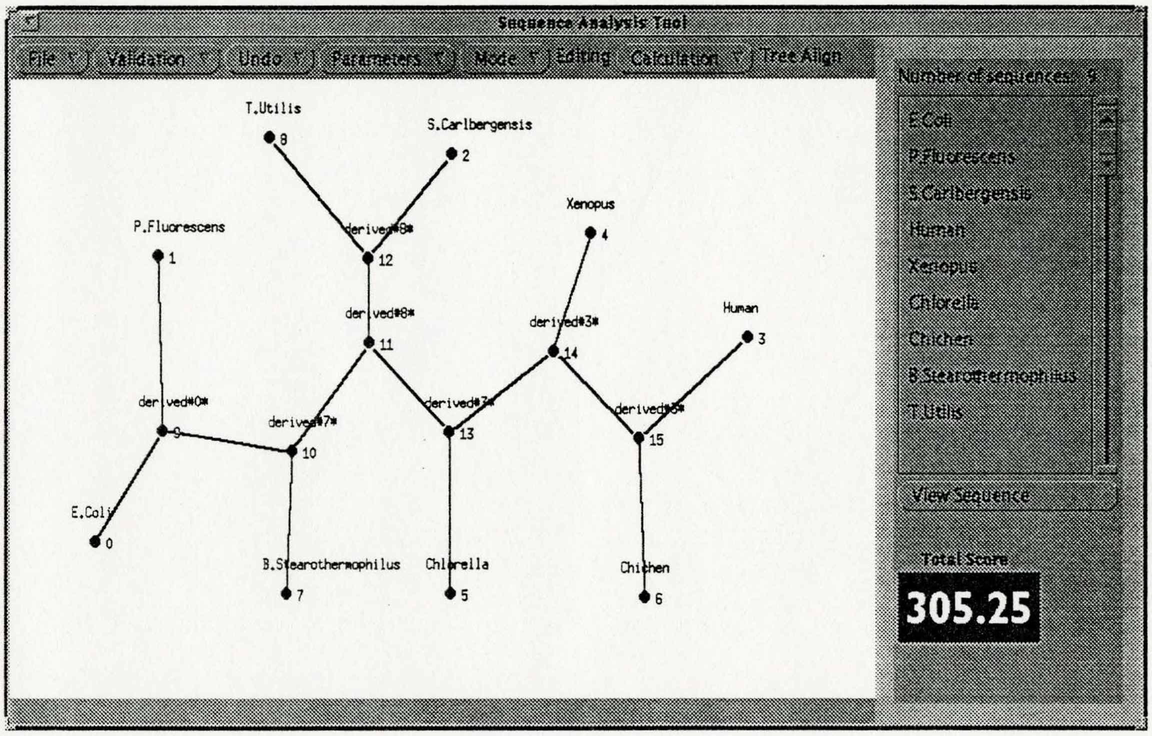


Figure 6.4: Executing local optimization (first round).

8. **Repeat tree alignment**

   Repeat the above local optimization step, the result is displayed in Figure 6.5.

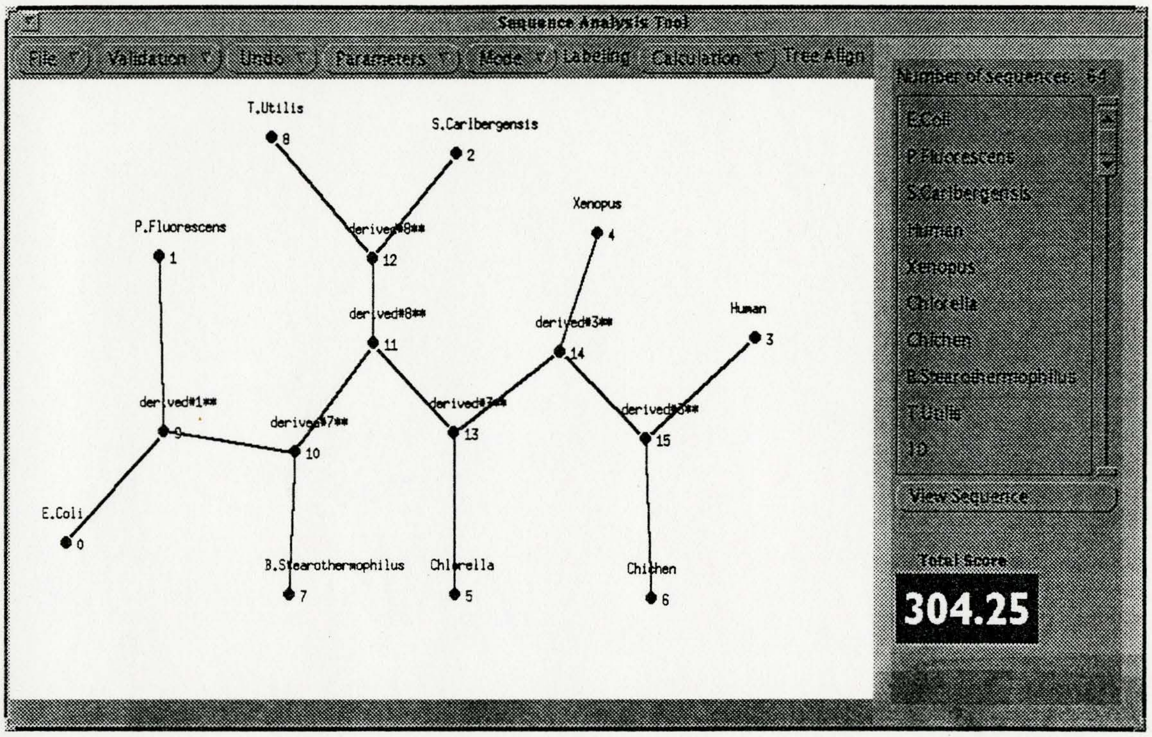


Figure 6.5: Executing local optimization (second round).

9. **Save the alignment result to a file.**

   Click on

   file $\longrightarrow$ **save**

   to display the *Save File* window. Then type a file name. The alignment result is shown in Appendix B.

As shown in the above figures, after the internal nodes of the tree are initialized with the **tree** program. The total score is 370.00. Two successive rounds of local optimization on every node bring the score down to 305.25 and 304.25. We have continued more rounds of local optimalizations, conducted in various order of nodes. The score was unchanged. However, this does not mean that we have found an optimal tree alignment for the given data. Assigning different sequences to internal nodes at the initialization stage keeps the final score stable with different values. So far, the best score we have got is 297.25 obtained by assigning sequences extracted from [18] to the internal nodes at the initialization stage.

Chapter 7
Concluding Remarks

Sequence Analysis Tool is aimed to provide an interactive tool for the performance analysis of various alignment-related algorithms. Now it provides a friendly graphical user interface for performing tasks related to pairwise alignment, tree alignment with a given structure and 3-star alignment. It may also be used to preprocess data for other activities such as multiple alignments, phylogenetic reconstruction, etc, which requires the distance between any two sequences under consideration.

As regard adding more features to SAT, a good candidate that may take advantage of the graphical displaying service of SAT is the feature for doing phylogenetic reconstructions, which can be described as: *given a collection of sequences, reconstruct a branching structure, termed a phylogeny or tree, that illustrates the ancestral relationships between the sequences.* A very commonly used algorithm for this purpose is the Neighbor Joining method proposed by Saitou and Nei [22].

An interesting survey concerning sequence comparison methods has been conducted by Chan, Wong and Chiu [4]. A useful resource of information on various software of sequence analysis is the site, accessible through anonymous ftp to

evolution.genetics.washington.edu

SAT, in its first version, serves as a prototype for further improvement. In the future development, the following points should be considered.

Concerning the X Toolkit, although OLIT and OpenWindows are quite friendly and well-developed, the market has made the X/Motif Toolkit the most popular development tool in the X-based GUI industry. So, for a serious development, X/Motif Toolkit should be applied instead. This was actually given as an advice, in a private talk with a representative of Sun Microsystem.

There are lots of applications that use graphs to represent relationships. A graph editor is actually needed in many interactive software systems. More generic approaches should be considered in the development of a graph editor so that the source code can be reused and extended in different applications. Therefore objected-oriented design and programming should be followed. In fact, there are generic C++ libraries [23], including a generic class definition for representing graphs, which is implemented through the adjacency lists data structure. By the class inheritance feature of objected-oriented programming, the generic graph class can be easily extended to represent and handle graphs required in various applications.

Therefore, C++ and Motif are recommended for the continuation of this project.

Appendix A
Sample Sequences File (input)

The following nine sequences are extracted from [18]. The sequences are written in FASTA format.

```
>e.Coli 9-1Sankoff

UGCCUGGCGG CCGUAGCGCG GUGGUCCCAC CUGACCCCAU GCCGAACUCA GAAGUGAAAC

GCCGUAGCGC CGAUGGUAGU GUGGGGUCUC CCCAUGCGAG AGUAGGGAAC UGCCAGGCAU


>P.Fluorescens 9-2Sankoff

UGUUCUUUGA CGAGUAGUAG CAUUGGAACA CCUGAUCCCA UCCCGAACUC AGAGGUGAAA

CGAUGCAUCG CCGAUGGUAG UGUGGGGUUU CCCCAUGUCA AGAUCUCGAC CAUAGAGCAU


>S.Carlbergensis 9-3Sankoff

GGUUGCGGCC AUACCAUCUA GAAAGCACCG UUCUCCGUCC GAUAACCUGU AGUUAAGCUG

GUAAGAGCCU GACCGAGUAG UGUAGUGGGU GACCAUACGC GAAACCUAGG UGCUGCAAUC U


>Human 9-4Sankoff

GUCUACGGCC AUACCACCCU GAACGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG

GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCGGGUG CUGUAGGCUU
```

```
>Xenopus 9-5Sankoff

GCCUACGGCC ACACCACCCU GAAAGUGCCC GAUCUCGUCU GAUCUCGGAA GCCAAGCAGG

GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG UCGUAGGCUU


>Chlorella 9-6Sankoff

AUGCUACGUU CAUACACCAC GAAAGCACCC GAUCCCAUCA GAACUCGGAA GUUAAACGUG

GUUGGGCUCG ACUAGUACUG GGUUGGGAGG AUUACCUGAG UGGGAACCCC GACGUAGUGU


>Chichen 9-7Sankoff

GCCUACGGCC AUCCCACCCC UGUAACGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGGG

UCGGGCCUGG UUAGUACUUG GAUGGGAGAC CUCCUGGGAA UACCGGGUGC UGUAGGCUU


>B.Stearothermophilus 9-8Sankoff

CCUAGUGACA AUAGCGAGGA GAGAAACACC CGUCUCCAUC CCGAACACGA AGGUUAAGCUC

UCCCAGCGCC GAUGGUAGUU GGGGCCAGCG CCCCUGCAAG AGUAGGUUGU CGCUAGGC


>T.Utilis 9-9Sankoff

GGUUGCGGCC AUAUCUAGCA GAAAGCACCG UUCUCCGUCC GAUCAACUGU AGUUAAGCUGC

UAAGAGCCUG AUCGAGUAGU GUAGUGGGUG ACCAUACGCG AAACUCAGGU GCGCAAUCU
```

Appendix B
Sample Output File

This is extracted from the output of the testing described in Chapter 6. The local optimization has been applied three times on each internal node.

```
/presentation of a graph.

16  1111


/Edges

0 9 21.50

[the above line means the edge connecting 0,9 has weight 39.00]

[other lines were deleted]


/Coordinates of nodes

48 367   /* the first node's coordinates */

[other lines were deleted]


/Contents of nodes

[Here come first 9 sequences assigned to leaves, they are listed

 in Appendix A,  and therefore omitted here. ]

>derived#0*** updated by star-align

UGCCUAGUGA CAGUAGUAGC AGUGGAACAC CUGACCCCAU CCCGAACUCA GAGGUGAAAC

GCCGCAGCGC CGAUGGUAGU GUGGGGUCUC CCCAUGCAAG AGUAGGGAAC CGCUAGGCAU

>derived#7*** updated by star-align

GCCUAGUGAC AAUAGCUAGC AGAGAAACAC CCGUCUCCAU CCCGAACACA GAGGUUAAGC

UCCCCAGCGC CGAUGGUAGU GUGGGGUCAC CCCAUGCAAG AGUAGGGUGC CGCUAGGCU
```

```
>derived#8*** updated by star-align
GCCUACGGCC AUACCUAGCA GAAAGCACCC GUCUCCGUCC GAUCACAGAA GUUAAGCUGC
UCAGAGCCUG AUGAGUAGUG UAGUGGGUGA CCACAUGCGA AAAUCAGGUG CUGCAGUCU
>derived#8**** updated by star-align
GGUUGCGGCC AUACCUAGCA GAAAGCACCG UUCUCCGUCC GAUCACCUGU AGUUAAGCUG
CUAAGAGCCU GAUCGAGUAG UGUAGUGGGU GACCAUACGC GAAACUCAGG UGCUGCAAUC U
>derived#3*** updated by star-align
GUCUACGGCC AUACCACCAC GAAAGCACCC GAUCCCGUCC GAUCUCGGAA GUUAAGCAUG
GUCGGGCCUG AUUAGUACUG GGAUGGGAGA CCACCUGGGA AAACCAGGUG CUGUAGUCU
>derived#3*** updated by star-align
GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG CUGUAGGCUU
>derived#3*** updated by star-align
GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCGGGUG CUGUAGGCUU


/Information of edges
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =21.50, LENGTH =121,
#MATCH =105, MISMATCH =14, GAPs =2, INDEL =2
   0:   UGCCUGGCGG CCGUAGC-GC GGUGGUCCCA CCUGACCCCA UGCCGAACUC AGAAGUGAAA
   9:   UGCCUAGUGA CAGUAGUAGC AGUGGAAC-A CCUGACCCCA UCCCGAACUC AGAGGUGAAA
        ||||| | |  | ||||  ||  ||||  | | |||||||||| | |||||||| ||| ||||||
   0:   CGCCGUAGCG CCGAUGGUAG UGUGGGGUCU CCCCAUGCGA GAGUAGGGAA CUGCCAGGCA U
   9:   CGCCGCAGCG CCGAUGGUAG UGUGGGGUCU CCCCAUGCAA GAGUAGGGAA CCGCUAGGCA U
        ||||| |||| |||||||||| |||||||||| |||||||| | |||||||||| | || ||||| |
***************************************************
```

```
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =34.50, LENGTH =123,
#MATCH =102, MISMATCH =15, GAPs =6, INDEL =6
   1:   UGUUCUUUGA CGAGUAGUAG CAUUGGAACA CCUGAUCCCA UCCCGAACUC AGAGGUGAAA
   9:   UGCCUAGUGA C-AGUAGUAG CAGUGGAACA CCUGACCCCA UCCCGAACUC AGAGGUGAAA
        ||     ||| | |||||||| || ||||||| ||||| |||| |||||||||| ||||||||||
   1:   CGAUGCAUCG CCGAUGGUAG UGUGGGGUUU CCCCAUGUCA AGA-UCUCGA -CCA-UAGAG CAU
   9:   CGCCGCAGCG CCGAUGGUAG UGUGGGGUCU CCCCAUG-CA AGAGUAGGGA ACCGCUAG-G CAU
        ||  ||| || |||||||||| |||||||| | ||||||| || ||| |   ||  ||  ||| | |||
***************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =12.75, LENGTH =122,
#MATCH =114, MISMATCH =6, GAPs =2, INDEL =2
   2:   GGUUGCGGCC AUACC-AUCU AGAAAGCACC GUUCUCCGUC CGAUAACCUG UAGUUAAGCU
  12:   GGUUGCGGCC AUACCUAGC- AGAAAGCACC GUUCUCCGUC CGAUCACCUG UAGUUAAGCU
        |||||||||| ||||| | |  |||||||||| |||||||||| |||| ||||| ||||||||||
   2:   GGUAAGAGCC UGACCGAGUA GUGUAGUGGG UGACCAUACG CGAAACCUAG GUGCUGCAAU CU
  12:   GCUAAGAGCC UGAUCGAGUA GUGUAGUGGG UGACCAUACG CGAAACUCAG GUGCUGCAAU CU
        | |||||||| ||| |||||| |||||||||| |||||||||| ||||||  || |||||||||| ||
***************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =1.75, LENGTH =120,
#MATCH =119, MISMATCH =1, GAPs =0, INDEL =0
   3:   GUCUACGGCC AUACCACCCU GAACGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
  15:   GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
        |||||||||| |||||||||| ||| |||||| |||||||||| |||||||||| ||||||||||
   3:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCGGGUG CUGUAGGCUU
  15:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCGGGUG CUGUAGGCUU
        |||||||||| |||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =6.00, LENGTH =120,
#MATCH =114, MISMATCH =6, GAPs =0, INDEL =0
   4:   GCCUACGGCC ACACCACCCU GAAAGUGCCC GAUCUCGUCU GAUCUCGGAA GCCAAGCAGG
  14:   GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
        | |||||||| | |||||||| ||||| |||| |||||||||| |||||||||| || |||||||
   4:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG UCGUAGGCUU
  14:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG CUGUAGGCUU
        |||||||||| |||||||||| |||||||||| |||||||||| ||||||||||   ||||||||
*****************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =42.00, LENGTH =123,
#MATCH =95, MISMATCH =21, GAPs =7, INDEL =7
   5:   AUGCUACGUU CAUAC-ACCA CGAAAGCACC CGAUCCCAUC AGAACUCGGA AGUUAAACGU
  13:   GU-CUACGGC CAUACCACCA CGAAAGCACC CGAUCCCGUC CGAUCUCGGA AGUUAAGCAU
         | |||||   ||||| |||| |||||||||| ||||||| ||  || |||||| |||||| | |
   5:   GGUUGGGCUC GACUAGUACU GGGUUGGGAG GAUUACCUGA GUGGGAACCC CG-AC-GUAG UGU
  13:   GGUCGGGCCU GAUUAGUACU GGGAUGGGAG -ACCACCUG- G-GAAAACCA GGUGCUGUAG UCU
        ||| ||||   || ||||||| ||| ||||||  |  |||||  | |  ||||   |  |  |||| | |
*****************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =10.75, LENGTH =121,
#MATCH =115, MISMATCH =4, GAPs =2, INDEL =2
   6:   GCCUACGGCC AUCCCACCCC UGUAA-CGCC CGAUCUCGUC UGAUCUCGGA AGCUAAGCAG
  15:   GUCUACGGCC AUACCACCC- UGAAAGCGCC CGAUCUCGUC UGAUCUCGGA AGCUAAGCAG
        | |||||||| || ||||||  || || |||| |||||||||| |||||||||| ||||||||||
   6:   GGUCGGGCCU GGUUAGUACU UGGAUGGGAG ACCUCCUGGG AAUACCGGGU GCUGUAGGCU U
  15:   GGUCGGGCCU GGUUAGUACU UGGAUGGGAG ACCGCCUGGG AAUACCGGGU GCUGUAGGCU U
        |||||||||| |||||||||| |||||||||| ||| |||||| |||||||||| |||||||||| |
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =24.50, LENGTH =122,
#MATCH =108, MISMATCH =8, GAPs =6, INDEL =6
   7:   -CCUAGUGAC AAUAGCGAGG AGAGAAACAC CCGUCUCCAU CCCGAACACG AAGGUUAAGC
  10:   GCCUAGUGAC AAUAGCUAGC AGAGAAACAC CCGUCUCCAU CCCGAACACA GAGGUUAAGC
         ||||||||| |||||| ||  |||||||||| |||||||||| |||||||||   |||||||||
   7:   UCUCCCAGCG CCGAUGGUAG U-UGGGGCCA GCGCCCCUGC AAGAGUAGGU UGUCGCUAGG C-
  10:   UC-CCCAGCG CCGAUGGUAG UGUGGGGUCA -C-CCCAUGC AAGAGUAGGG UGCCGCUAGG CU
        || ||||||| |||||||||| | ||||| ||  | ||| ||| |||||||||  || ||||||| |
****************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =2.75, LENGTH =121,
#MATCH =119, MISMATCH =2, GAPs =0, INDEL =0
   8:   GGUUGCGGCC AUAUCUAGCA GAAAGCACCG UUCUCCGUCC GAUCAACUGU AGUUAAGCUG
  12:   GGUUGCGGCC AUACCUAGCA GAAAGCACCG UUCUCCGUCC GAUCACCUGU AGUUAAGCUG
        |||||||||| ||| |||||| |||||||||| |||||||||| ||||| |||| ||||||||||
   8:   CUAAGAGCCU GAUCGAGUAG UGUAGUGGGU GACCAUACGC GAAACUCAGG UGCUGCAAUC U
  12:   CUAAGAGCCU GAUCGAGUAG UGUAGUGGGU GACCAUACGC GAAACUCAGG UGCUGCAAUC U
        |||||||||| |||||||||| |||||||||| |||||||||| |||||||||| |||||||||| |
****************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =26.75, LENGTH =121,
#MATCH =104, MISMATCH =14, GAPs =3, INDEL =3
   9:   UGCCUAGUGA CAGUAG-UAG CAGUGGAACA CCUGACCCCA UCCCGAACUC AGAGGUGAAA
  10:   -GCCUAGUGA CAAUAGCUAG CAGAGAAACA CCCGUCUCCA UCCCGAACAC AGAGGUUAAG
         ||||||||| || ||| ||| ||| | |||| || | | ||| |||||||| | |||||| ||
   9:   CGCCGCAGCG CCGAUGGUAG UGUGGGGUCU CCCCAUGCAA GAGUAGGGAA CCGCUAGGCA U
  10:   CUCCCCAGCG CCGAUGGUAG UGUGGGGUCA CCCCAUGCAA GAGUAGGGUG CCGCUAGGC- U
        | || ||||| |||||||||| |||||||||  |||||||||| ||||||||   |||||||||  |
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =44.75, LENGTH =123,
#MATCH =95, MISMATCH =20, GAPs =8, INDEL =8
  10:   GCCUAGUGAC AAUAGCUAGC AGAGAAACAC CCGUCUCCAU CCCGAACACA GAGGUUAAGC
  11:   GCCUA-CGGC CAUACCUAGC AGA-AAGCAC CCGUCUCCGU CC-GAUCACA GAAGUUAAGC
        |||||  | |  ||| ||||| ||| || ||| |||||||| | || || |||| || |||||||
  10:   UCCCCAGCGC C-GAUG-GUA GUGU-G-GGG UCACCCCAUG CAAGAGUAGG GUGCCGCUAG GCU
  11:   UGCUCAGAGC CUGAUGAGUA GUGUAGUGGG UGACCACAUG CGAAAAUCAG GUGCUGC-AG UCU
        | | ||| || | |||| ||| |||| | ||| | ||| |||| | | | |  | |||| || ||  ||
***************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =35.50, LENGTH =121,
#MATCH =98, MISMATCH =19, GAPs =4, INDEL =4
  11:   GCCUACGGCC AUACCUAGCA -GAAAGCACC CGUCUCCGUC CGAUCACAGA AGUUAAGC-U
  13:   GUCUACGGCC AUACC-ACCA CGAAAGCACC CGAUCCCGUC CGAUCUCGGA AGUUAAGCAU
        | |||||||| ||||| | ||  ||||||||| ||   ||||| ||||| | || |||||||| |
  11:   GCUCAGAGCC UGAUGAGUAG UGUAGUGGGU GACCACAUGC GAAAAUCAGG UGCUGCAGUC U
  13:   GGUCGG-GCC UGAUUAGUAC UGGGAUGGGA GACCACCUGG GAAAACCAGG UGCUGUAGUC U
        | || | ||| |||| ||||  ||   ||||  |||||| ||  ||||| |||| ||||| |||| |
***************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =21.75, LENGTH =121,
#MATCH =107, MISMATCH =12, GAPs =2, INDEL =2
  11:   GCCUACGGCC AUACCUAGCA GAAAGCACCC GUCUCCGUCC GAUCACA-GA AGUUAAGCUG
  12:   GGUUGCGGCC AUACCUAGCA GAAAGCACCG UUCUCCGUCC GAUCACCUGU AGUUAAGCUG
        |  | ||||| |||||||||| |||||||||    ||||||||| ||||||  |  ||||||||||
  11:   CUCAGAGCCU GAU-GAGUAG UGUAGUGGGU GACCACAUGC GAAAAUCAGG UGCUGCAGUC U
  12:   CUAAGAGCCU GAUCGAGUAG UGUAGUGGGU GACCAUACGC GAAACUCAGG UGCUGCAAUC U
        || ||||||| ||| |||||| |||||||||| ||||| | || |||| ||||| ||||||| || |
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =18.00, LENGTH =120,
#MATCH =107, MISMATCH =12, GAPs =1, INDEL =1
  13:   GUCUACGGCC AUACCACCAC GAAAGCACCC GAUCCCGUCC GAUCUCGGAA GUUAAGCAUG
  14:   GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
        |||||||||| ||||||||   |||||| ||| |||| ||||  |||||||||| | |||||| |
  13:   GUCGGGCCUG AUUAGUACUG GGAUGGGAGA CCACCUGGGA AAACCAGGUG CUGUAGUCU-
  14:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG CUGUAGGCUU
        ||||||||||  ||||||||  |||||||||| || ||||||| | |||||||| |||||| ||
****************************************************
#MATRIX = sankoffSCORE, GAP OPEN PANELITY =0.00,GAP EXTEND PANELITY =2.25,
#SCORE =1.00, LENGTH =120,
#MATCH =119, MISMATCH =1, GAPs =0, INDEL =0
  14:   GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
  15:   GUCUACGGCC AUACCACCCU GAAAGCGCCC GAUCUCGUCU GAUCUCGGAA GCUAAGCAGG
        |||||||||| |||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
  14:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCAGGUG CUGUAGGCUU
  15:   GUCGGGCCUG GUUAGUACUU GGAUGGGAGA CCGCCUGGGA AUACCGGGUG CUGUAGGCUU
        |||||||||| |||||||||| |||||||||| |||||||||| ||||| |||| ||||||||||
****************************************************
```

Appendix C
Score Matrix File Format

The following are two sample matrix files for score systems used in SAT. They should clearly demostrate how to construct a compatible matrix file. The first one show a matrix where different pair has a different score.

```
## file name:  sankoffSCORE
## Score scheme from Sankoff's paper.
## Number of letters in alphabet =4
##
A      C       U       G
0   1.75    1.75       1
       0       1    1.75
               0    1.75
                       0
```

The next format has a constant score on each match pair and mismatch pair.

```
## A constant cost for mismatch
# The mismatch and match cost  are    1   0
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Bibliography

[1] S. Altschul, *Gap Costs for Multiple Sequence Alignment,* J. Theor. Biol. 138, pp. 279-309, 1989.

[2] S. Altschul and D. Lipman, *Trees, Stars, and Multiple Sequence Alignment,* SIAM Journal on Applied Math. 49, pp. 197-209, 1989.

[3] H. Carrillo and D. Lipman, *The Multiple Sequence Alignment Problem in Biology,* SIAM Journal on Applied Math. 48, pp. 1073-1082, 1988.

[4] S. C. Chan, A. K. C. Wong and D. K. T. Chiu, *A Survey of Multiple Sequence Comparison Methods,* Bulletin of Mathematical Biology 54(4), pp. 563-598, 1992.

[5] Alan Dix, Janet Finlay, Gregory Abowd and Russell Beale, *Human-Computer Interaction*, Prentice Hall, 1993.

[6] D. Eppstein, R. Giancarlo and G. F. Italiano, *Sparse Dynamic Programming 1: Linear Cost Functions,* J. Assoc. Comp. Machinery, 39(3), pp. 519-545, 1992.

[7] D. Gusfield, *Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds,* Bulletin of Mathematical Biology 55, pp. 141-154, 1993.

[8] M. Gribskov and J. Devereux, *Sequence Analysis Primer,* Stockton Press, 1991.

[9] J. J. Hein, *A Tree Reconstruction Method that is Economical in the Number of Pairwise Comparisons Used,* Mol. Biol. Evol. 6(6), pp. 669-684, 1989.

[10] J. J. Hein, *A New Method that Simultaneously Aligns and Reconstructs Ancestral Sequences for Any Number of Homologous Sequences, When the Phylogeny is Given,* Mol. Biol. Evol. 6(6), pp. 649-668, 1989.

[11] D.S. Hirschberg, *A Linear Space Algorithm for Computing Longest Common Subsequences,* Commun. Assoc. Comput. Mach., 18, pp. 341-343, 1975.

[12] T. Jiang and M. Li, *Optimization Problems in Molecular Biology,* to appear in Advances in Optimization, D.-Z. Du (ed.), 1993.

[13] Oliver Jones, *Introduction to the X Window System*, Prentice Hall, 1989.

[14] E. W. Myers and W. Miller, *Optimal Alignments in Linear Space,* CABIOS, 4(1), pp. 11-17, 1988.

[15] Brad A. Myers and Mary Beth Rosson, *Survey on User Interface Programming,* CHI'92 Conference Proceedings on Human Factors in Computer Systems, pp. 195-202, ACM Press, New York, 1992.

[16] D. Penny, *Criteria for Optimising Phylogenetic Trees and the Problem of Determining the Root of a Tree,* J. Mol. Evol. 8, pp. 95-116, 1976.

[17] D. Sankoff, *Minimal Mutation Trees of Sequences,* SIAM J. APPL. Math. 28(1), pp. 35-42, 1975.

[18] D. Sankoff, R. J. Cedergren and G. Lapalme, *Frequency of Insertion-Deletion, Transversion, and Transition in the Evolution of 5S Ribosomal RNA,* J. Mol. Evol. 7, pp. 133-149, 1976.

[19] D. Sankoff and R. Cedergren, *Simultaneous Comparisons of Three or More Sequences Related by a Tree,* Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison, pp. 253-264, Addison Wesley, Reading Mass., 1983.

[20] D. Sankoff and J. Kruskal (Eds), *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison,* Addison Wesley, Reading Mass., 1983.

[21] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[22] N. Saitou and M. Nei, *The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees,* Mol. Biol. Evol. 4-4, pp. 406-425, 1987.

[23] Namir C. Shammas, *Visual C++ Generic Programming,* Windcrest/McGraw-Hill, 1994.

[24] L. Wang and T. Jiang, *On the Complexity of Multiple Sequence Alignment,* to appear in Journal of Computational Biology, 1993.

[25] T. Jiang, E. L. Lawler and L. Wang, *Aligning Sequences Via an Evolutionary Tree: Complexity and Approximation,* Proc. of the 26th ACM Symp. on Theory of Computing, pp. 760-769, 1994.

[26] M. S. Waterman, *Mathematical Methods for DNA sequences*, CRC Press, 1989.

[27] Douglas A. Young and John A. Pew, *The X Window System Programming and Applications With Xt (OPEN LOOK Edition)*, Prentice Hall, 1992.

[28] M. Dayhoff, *Atlas of Protein Sequence and structure*, Vol.5, Suppl. 3, pp.345-358, 1978, National Biomedical Research Foundation, Washington.