Rt
489
c.1

Heuristics: Encoding for Parsimony Phylogenetic Trees & Generic Implementations

By
Chadi KALLAB

Submitted in partial fulfillments of the requirements
For the degree of Masters of Science

Thesis Advisor:
Dr. Danielle AZAR

Department of Computer Science & Mathematics
Lebanese American University – Byblos
June 2006

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences

Thesis Approval

Student Name	**CHADI KALLAB**	I.D.#:	**199730860**
Thesis Title:	**HEURISTICS: ENCODING FOR PARSIMONY PHYLOGENETIC TREES GENERIC IMPLEMENTATIONS**		
Program:	**Computer Science**		
Division /Dept:	**Computer Science and Mathematics**		
School:	**School of Arts and Sciences, Byblos**		

Approved by:

Thesis Advisor:	***DANIELLE AZAR***	[redacted]
Member	***HAIDAR M. HARMANANI***	[redacted]
Member	***JEAN TAKCHE***	[redacted]
Member	***JOSEPH KHALIFE***	[redacted]
Date:	JUNE 16, 2006	

Plagiarism Policy Compliance Statement

I certify that I have read and understood the Lebanese American University's plagiarism policy. I understand that failure to comply with this can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have that by acknowledging its sources.

Name: Chadi KALLAB

Signature

[redacted]

Date: June 19, 2006

To my instructors and advisor who bore me

To my family and friends who supported me

And finally the world of Bioinformatics
& Artificial Intelligence

Acknowledgments

I would like to express my profound gratitude and appreciation to my thesis committee chairman Dr. Danielle AZAR for her guidance, patience, and sincere advices throughout this thesis, and thank her for giving me an important amount of valuable time, constructive criticism, and stimulating discussions; but, also my thesis committee members: Dr Haidar HARMANANI, Mr Joe KHALIFE and Dr Jean TAKCHI for their support, comments and critical review of the thesis.

Also I would like to thank the LEBANESE AMERICAN UNIVERSITY, especially the Computer Science and Mathematics department, who made possible to excel in this accomplishment, by providing me with financial and technical resources and support throughout my graduate studies.

Special thanks to my family, especially my father (neurologist), who were a constant source of motivation and support. Their love, care and knowledge did help me a lot throughout the stages of this thesis.

Last but not least, I would like to thank my friends who supported and cheered me up, and my colleagues who shared with me their knowledge, during the elaboration of the different components of the thesis.

Abstract

This thesis focuses on the NP-Hard problem of finding an optimal tree topology where leaves represent biological sequences. The problem consists of minimizing the number of changes between given and/or derived sequences. As the number of sequences to be compared increases, the size of the search space grows exponentially, requesting the use of optimization methods, to come up with an acceptable optimal topology. Even though, research is relating a large number of species and gene families to each others, the computation intensive load of many popular methods for evaluating trees (ex: parsimony and maximum likelihood) establish the quasi-inexistence of an exact solution for more than about 20 sequences.

The alignment of two sequences (pairwise alignment), or multiple sequences (Multiple Sequence Alignment - MSA), and the alignment of short or long sequences such as an entire genome may require different types of algorithms. The algorithms used in all of these four cases are dynamic programming, linear programming based or heuristic-based or a combination of them. For large numbers of sequences to be aligned, heuristic methods may give a result similar to the exact solution offered by the dynamic programming or linear programming based algorithms, but in much shorter time. Heuristics used in phylogenetic inference include greedy algorithms, hill climbing, simulated annealing, and genetic algorithms.

Thus, this research aims to suggest a general way to encode the problem into instances of different heuristic algorithms. Another focus of the document would be to suggest that the heuristics used, be implemented in a most optimal way, trying to get a compromise between speedup, flexibility and detailed tracing/chronology of each run of the algorithms.

Table of Contents

Table of Figures

Chapter 1 Introduction

This thesis focuses on the NP-Hard problem of finding an optimal tree topology where leaves represent biological sequences. [2, 3] The problem consists of minimizing the number of changes between given and/or derived sequences. As the number of sequences to be compared increases, the size of the search space grows exponentially. This fast growth induces the necessity of using optimization methods in order to come up with an acceptable optimal topology.

The average size of phylogenetic analyses is increasing. Research is being performed on relationships among large numbers of species and large gene families. However, many of the popular methods for evaluating trees, including parsimony [7] and maximum likelihood [6], are so computationally intensive that an exact solution is not possible for more than about 20 sequences.

The alignment of two sequences (pair wise alignment), or multiple sequences (Multiple Sequence Alignment - MSA), and the alignment of short or long sequences such as an entire genome may require different types of algorithms. The algorithms used in all of these four cases are dynamic programming, linear programming based or heuristic-based or a combination of them. For larger numbers of sequences, heuristic methods may give a result similar to the exact solution, but in much shorter time. Heuristics used in phylogenetic inference include greedy algorithms (stepwise addition - Swofford 1993), hill climbing (Croes 1958), simulated annealing (Kirkpatrick et al. 1983, Lundy 1985, Salter and Pearl 2001), and genetic algorithms (Holland 1975, Lewis 1998).

Thus, this research aims to suggest a general way to encode the problem into instances of different heuristic algorithms. Another focus of the document would be to suggest that the heuristics used, be implemented in a most optimal way, trying to get a compromise between speedup, flexibility and detailed tracing/chronology of each run of the algorithms.

Chapter 2 Problem Statement

"All biological disciplines are united by the idea that species share a common history. The genealogical history of life - also called an "evolutionary tree" - is usually represented by a bifurcating, leaf-labeled tree." [1] Many problems in biology can be solved through the use of evolutionary trees as a fundamental step. Such problems touch the fields of MSA, prediction of protein structure and function, or drug design. Over the last decade or so, computer scientists started to design and analyze the performance of phylogenetic methods under statistical models that approaches the process stochastically. [1] Since it is widely thought that all species did successively mutate from common ancestors, biology, mathematics and computing science have researched methods to model this evolutionary process trying to find the optimal arrangement of these species and their ancestors.

"An evolutionary tree (also called a phylogenetic tree) models the evolution of a set of taxa (species, bio-molecular sequences, etc) from a common origin. Thus, an evolutionary tree is rooted at the most recent common ancestor of the taxa, and the internal nodes of the tree are each labeled by a hypothesized or known ancestor. The common practice today is to use bio-molecular sequences as representatives of the species set, so that the leaves of the tree are labeled by bio-molecular (DNA, RNA, or amino acid) sequences." [1] In a phylogenetic tree, leaves are labeled by the species in the set. The objective then of a phylogenetic reconstruction algorithm is to find a tree, which arranges the most optimally the leaf sequences. Parsimony is one of the most popular methods for phylogenetic tree inference. In order to explain the parsimony method, we begin with some definitions.

2.1 Definitions

A base can be defined as the most basic element of a sequence. It is generally represented by one single character, and has a complex chemical structure that may or may not interact with that of other bases. The set of all possible character representations is called the base alphabet.

The *Hamming distance* between two sequences X and Y of the same length is the number of different bases between x and y, and is denoted H(X; Y), i.e. $H(X,Y)=count\left({}^{i}\!/\!{}_{X_i \neq Y_i} \right)$

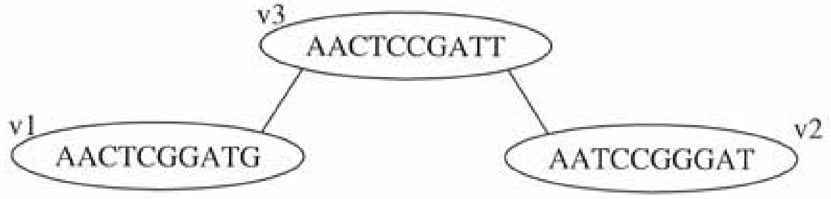
For instance, assuming that the alphabet is {A, T, C, G} and sequences X and Y are respectively: AACTCGGATG and AATCCGGGAT, the hamming distance will be:

Seq. X:	A	A	C	T	C	G	G	A	T	G
Seq. Y:	A	A	T	C	C	G	G	G	A	T

Therefore, H (X, Y) = 5

The *parsimony length* of a tree, in which each node v is labeled by a sequence s^v, is the sum of the Hamming distances of sequences labeling endpoints of edges in the tree, i.e. $\sum_{(v1,v2)\in E} H\left(s^{v1}, s^{v2}\right)$

For instance, let us consider a tree of 3 nodes, where the sequences are s^{v1} = AACTCGGATG, s^{v2} = AACTCGGATG, and s^{v3} = AACTCCGATT, thus the parsimony length of the tree is computed as follow:



Thus: $ParsimonyLength = H\left(s^{v1}, s^{v3}\right) + H\left(s^{v2}, s^{v3}\right) = 2 + 5 = 7$

An *operational taxonomic unit* (OTU) represents a sequence in the set of given sequences to be compared. The number of OTUs is the number of leaf nodes in a tree illustrating this set. Hence, in the above tree, the OTU is 2.

Given a set S of sequences, a *Most Parsimonious* tree for S is a tree, of minimum parsimony length, in which each leaf node is labeled by a sequence in S, and each internal node assigned a sequence derived by some phylogenetic reconstruction method (ex: Fitch algorithms that will be discussed in a later chapter). Thus, the parsimony criterion is to find a tree of minimum parsimony length or in other terms the tree of maximum parsimony.

2.2 Statement & Motivation

The motivation for the parsimony criterion is the observation that if evolution is assumed to operate only through point mutations (for example, substitutions of one nucleotide with another) then the parsimony length of a tree is the minimum possible number of evolutionary events needed to obtain the set of sequences observed at the leaves through point mutations.

"The Parsimony Phylogenetic Tree Problem is an NP-hard problem, even when the sequences are binary (i.e. the alphabet size is two, and characters allowed are only 1 and 0)." [1] The most typical maximum parsimony approach (MPA) tries to come up with as many possible topologies, generating the best internal node labeling for each topology. Then this approach figures out the most optimal one among them, by comparing their parsimony scores. [1]

The phylogenetic tree handled during an iteration of the "Maximum Parsimony" approach has the minimum number and/or cost of steps in the evolution of a given set of data (set of DNA/RNA bases or proteins – in general called sequences or set of characters). Usually, the step is defined as the substitution of one character to another character of the same alphabet. There are two main approaches used when computing the parsimony score of a phylogenetic tree of proteins. One of these approaches, treats each character location is handled independently of other locations.

"Maximum parsimony approach searches all possible tree topologies for the optimal (minimal) tree. However, the number of un-rooted trees that have to be analyzed rapidly increases with the number of OTU" [3]

Equations 1 and 2 show the number of rooted and un-rooted trees for n OTU. Table 1 shows the number of trees versus the number of OTU.

The number of **rooted** trees (N_R) for n OTU is given by:

$$N_R = 1*3*5*...*(2n-3) = \frac{Fact(2n-3)}{Fact(n-2)*2^{n-2}} \qquad [3]$$

The number of **un-rooted** trees (N_U) for n OTU is given by:

$$N_U = 1*3*5*...*(2n-5) = \frac{Fact(2n-5)}{Fact(n-3)*2^{n-3}} \qquad [3]$$

n **Number of OTUs**	N_U **Number of un-rooted trees**	N_R **Number of rooted trees**
2	1	1
3	1	3
4	3	15
5	15	105
6	105	945
7	945	10,395
8	10,395	135,135
9	135,135	34,459,425
10	34,459,425	2.13E15
15	2.13E15	8.E21

Table 1: Number of Sequences vs. Size of Search Space [3]

Since the number of trees to analyze grows exponentially with the number of OTUs, using the above mentioned parsimony method becomes inefficient. In other terms, the computation, involved in figuring out the most parsimonious tree, will grow exponentially as the number of leaf sequences increase.

Indeed, Fitch shows, in his standard and weighted algorithms, that: for a given leaf-labeled tree topology, the exact minimum number of nucleotide replacements can be systematically computed. In addition, all possible ancestral node sequences can be systematically generated, for this computed number. Fitch also suggests that the standard algorithm can be improved by considering some weights on the changes between nucleotides. [7, 10]

For this reason, Heuristic search techniques are one good approach to the problem. Among the heuristic search methods, there are the ones that apply local search (e.g., hill climbing) and the ones that use a non-convex optimization approach, in which cost-deteriorating neighbors are accepted also. The most popular methods which go beyond simple local search are Genetic Algorithms (GA) (and other evolutionary techniques, like evolutionary programming, evolutionary strategies, etc.), Simulated Annealing (SA), and Tabu Search (TS).

Many papers found in the literature, concerning Phylogenetic trees using Heuristics, have discussed the use of an exact method (Fitch) or only one or at most two Heuristics (Genetic Algorithms, Simulated Annealing, Tabu Search, Simulated Evolution and Stochastic Evolution) in each one. Few of them have some pseudo-code. Others have some C/C++ code. Some techniques were not implemented at all.

Since most papers only describe one or two Heuristics, I suggest an encoding schema that can be applied to any or set of the above-mentioned algorithms. In the same time, I am proposing different pseudo-code alternatives for these algorithms.

Chapter 3 Background

3.1 Biology Overview

"While the period from the early 1900s to World War II has been considered the "golden age" of genetics, scientists still had not determined that DNA, not protein, was the hereditary material of living species. However, during this time great many genetic discoveries were made and the link between genetics and evolution was made." [5]

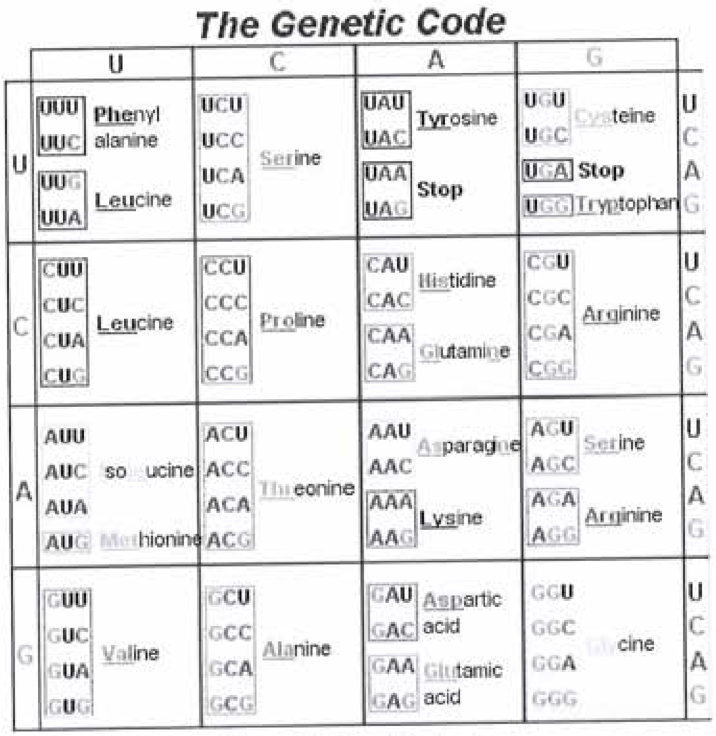
Since DNA came from nuclei, core element of any part of living cells, Meischer named this new chemical: nuclein. Subsequently the name was changed to nucleic acid and lastly to deoxyribonucleic acid (DNA). In 1914, Robert Feulgen found the DNA in the nucleus of all cells, with a distinct membrane nucleus, called eukaryotic cells.

"During the 1920s, biochemist P.A. Levene analyzed the components of the DNA molecule. He found out that it contained four nitrogenous bases: cytosine, thiamine, adenine, and guanine; deoxyribose sugar, and a phosphate group. He concluded that the basic unit (nucleotide) was composed of a base attached to a sugar and that the phosphate was attached to the sugar." [5] Thus, the nucleotide is considered as the fundamental unit of DNA.

After the hereditary material was settled to be DNA, and Watson et al. had decoded the DNA structure, research was tailored towards the understanding of the process by which DNA expresses its information in the phenotype, the expression of the genetic constitution in the form of traits that can be seen and measured, such as hair or eye color. Three models of DNA replication were considered, by Matthew Meselson and Franklin W. Stahl: conservative, semi-conservative and dispersive replication. Conservative replication would create a copy of the entire DNA strand. Semi-conservative replication will result in having two DNA molecules, each composed of one new and one old strand of DNA. As for dispersive replication, each strand of the resulting DNA would be a mixture of fragments of both parental strands. [5]

DNA is transcribed into a complementary RNA (ribonucleic acid) strands. The process involves a powerful enzymatic complex called "RNA Polymerase". This enzyme splits the DNA helix, and adds the RNA nucleotide corresponding to the currently examined DNA gene sequence.

Once the transcription process is finished, the resulting RNA undertakes translation into a protein, which is a major constituent of cells. Every three RNA nucleotides (A, C, G or U) gather to form a triplet, also called codon. Therefore, we have $4^3 = 64$ possible triplets. This set of possible codons is defined as the Genetic Code. Three of those triplets are considered as stop codons, also called termination codons, which stop the translation process. The rest encode 20 amino acids. Since one or more triplet may translate to the same amino acid, the Genetic Code is redundant or degenerated. [5]



The Genetic Code

	U	C	A	G	
U	UUU Phenyl alanine	UCU Serine	UAU Tyrosine	UGU Cysteine	U
	UUC	UCC	UAC	UGC	C
	UUG Leucine	UCA	UAA Stop	UGA Stop	A
	UUA	UCG	UAG	UGG Tryptophan	G
C	CUU Leucine	CCU Proline	CAU Histidine	CGU Arginine	U
	CUC	CCC	CAC	CGC	C
	CUA	CCA	CAA Glutamine	CGA	A
	CUG	CCG	CAG	CGG	G
A	AUU Isoleucine	ACU Threonine	AAU Asparagine	AGU Serine	U
	AUC	ACC	AAC	AGC	C
	AUA	ACA	AAA Lysine	AGA Arginine	A
	AUG Methionine	ACG	AAG	AGG	G
G	GUU Valine	GCU Alanine	GAU Aspartic acid	GGU Glycine	U
	GUC	GCC	GAC	GGC	C
	GUA	GCA	GAA Glutamic acid	GGA	A
	GUG	GCG	GAG	GGG	G

Figure 2: Genetic Code

Molecular biologists are currently engaged in impressive data collection projects. Recent genome-sequencing projects are generating an enormous amount of data related to the function and the structure of biological molecules and sequences.

Artificial Intelligence (AI) and heuristic techniques can provide key solutions for the new challenges posed by the progressive transformation of biology into a data-massive science. AI problems, that are particularly promising, include the prediction of protein structure and function, semi-automatic drug design, the interpretation of nucleotide sequences, and knowledge acquisition from genetic data. Thus, such AI and heuristic techniques and methods are extremely important for the present and future developments of bioinformatics, a very recent and strategic discipline having the potential for a revolutionary impact on biotechnology, pharmacology, and medicine. While computation has already transformed our industrial society, a comparable biotechnological transformation is on the horizon. In the last few years, it has become clear that these two exponentially growing areas are actually converging in some points.

Even though heuristics do not guarantee optimality, they are commonly used to compensate the exponential growth of computation, which is due to the use of classical methods of Operations Research (OR). Indeed, over the last 30 years, some heuristic algorithms were born from trials to mimic natural processes. These methods have produced interesting results in reasonable short runs. Even with their flexibility regarding modification in the problem description, bionic heuristics turn out to give better results than those of classical problem specific heuristics. [9] We give a brief overview of Heuristics in the next section.

3.2 Heuristics Overview

Basic Heuristic search methods utilize different mechanisms in order to explore the state space. These mechanisms are based on three basic features:

– The use of memory-less search (ex: standard Simulated Annealing – SA, and Genetic Algorithms – GA) or adaptive memory (Tabu Search – TS);

– The kind of neighborhood exploration used, i.e., random (e.g., SA and GAs) or systematic (e.g., TS); and

– The number of current solutions taken from one iteration to the next (GAs, as opposed to SA and TS, take multiple solutions to the next iteration).

The combination of these mechanisms for exploring the search space determines the search diversification (global exploration) and intensification (local exploitation) capabilities. Next, we give an overview of each of the most popular heuristics.

3.2.1 Genetic Algorithms

As shown in Figure 3, most Evolutionary Computing algorithms, including Genetic Algorithms, work iteratively on a subset of the solution space. We refer to this subset as a "population". After the *Initialization* of the solution space, a *Selection* is made to choose two or more chromosomes (solutions, ex: phylogenetic trees) of the current POPULATION. These chromosomes, denoted as PARENTS, are subjected to a *Crossover / Recombination* operation, which consists of combining bits and pieces taken from 2 or more solutions to create as many solutions, called "offspring". In their turn, the resulting OFFSPRING may be subjected to a *Mutation* process, which consists of slightly changing the solution. Both *Crossover* and *Mutation* operators are applied given certain respective probabilities. An individual *Fitness Evaluation* function is tested on each of the last operator's results, which will be added to an INTERMEDIARY POPULATION refreshed per iteration. An *Overall Evaluation* goes through this new population, evaluating the whole population, by computing the fitness of the entire group. The intermediary population replaces the current one and the whole process is repeated until a termination criterion is met (for example: a certain number of iterations have elapsed).

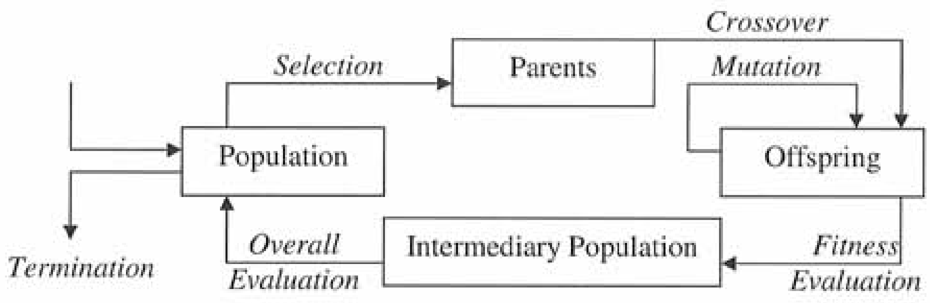


Figure 3: Genetic Algorithms General Flow Diagram

3.2.2 Simulated Annealing

Simulated Annealing (SA) was initially designed to simulate the cooling of metals into crystalline structure (annealing process). Since the natural annealing process aims to minimize energy in resulting crystals, SA tackles problems from the point of view of minimizing cost/energy.

SA was intended in 1983 to be used with non-linear problems. "SA approaches the global maximization problem similarly to using a bouncing ball that can bounce over mountains from valley to valley." [12] Initially, the temperature is set high enough so that this ball is given enough time to bounce, between and within valleys. The generic annealing algorithm/process suggests that, as time passes, the ball tends to get closer to its optimal location. Therefore, as the temperature cools down, the frequency of the large bounces (between valleys) tends to lower, while that of local bounces (within valleys) increases. However, to avoid falling in a local optimum valley, a probabilistic formula is applied, related to the current temperature gain and cost of bounce from the current location to a new one. In SA, the bounce (change in location) implies to perturb the current solution S into a new one S'. In the SA algorithm, the temperature parameter is denoted as T, and the number of moves (perturbations / bounces) is denoted by M. "It has been proved that by carefully controlling the rate of cooling of the temperature, SA can find the global optimum". [12]

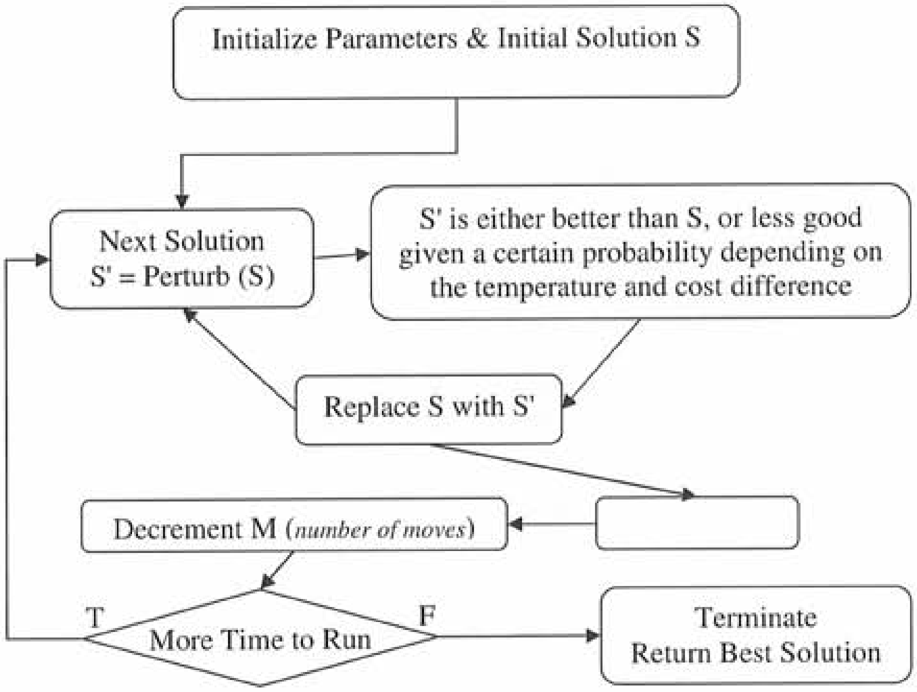

Initialize Parameters & Initial Solution S
Next Solution
S' = Perturb (S)
S' is either better than S, or less good given a certain probability depending on the temperature and cost difference
Replace S with S'
Decrement M (number of moves)
T
More Time to Run
F
Terminate
Return Best Solution

Figure 4: Simulated Annealing General Flow Diagram

3.2.3 Simulated Evolution

"Simulated evolution (SimE) is a general search strategy for solving a variety of combinatorial optimization problems. The SimE algorithm starts from an initial assignment of status to elements of the entire initial solution, and then, following an evolution-based approach, it seeks to reach better assignments at each generation. A cost function called goodness measure is used by SimE algorithm in order guide the algorithm in the search space." [14]

The data, being handled during a Simulated Evolution run, flows iteratively according to the following sequence of steps: Evaluation, Selection, and Allocation. "SimE operates on a single solution S_0. This solution is termed as population. Each population consists of elements, which represent the assignment of a state to a sub-part of the initial set of data. In the EVALUATION step, the goodness of each element is measured. The goodness of an element is a single real number between '0' and '1', which is a measure of how near is the element to its optimal state. Higher value of goodness means that the element is near its optimal state." [16] The result of this step is a goodness set $G = \{i, g_i\}$ where g_i is the goodness value of element i.

The SELECTION process chooses those elements, which have a low goodness in the current solution, by comparing this value with a random real number between '0' and '1', acting as a probability. If the element was not selected, it is assigned to an empty location. The number of selected elements can be further reduced by applying a selection bias B, with value typically between –0.2 and 0.2, which may also compensate for errors made in the estimation of goodness. The process outputs two complementary sets: P_s (selected) and P_r (remaining).

The ALLOCATION operator refreshes the states of each element that was selected in P_s. Since setting this process randomly gears the algorithm towards random search, greedy strategies would improve the algorithm in general, and particularly the measurement of the elements. Therefore, it reduces the overall cost of the solution. This operator takes as inputs the P_S and P_R sets, and outputs a new overall state. [16]

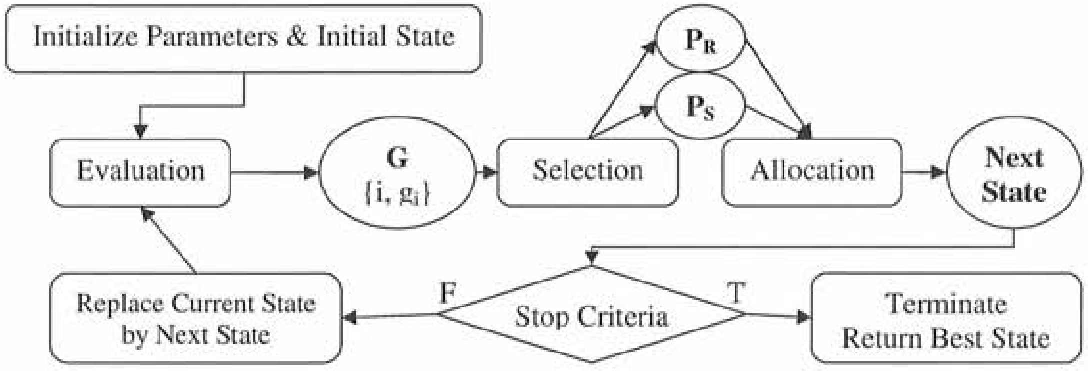

Initialize Parameters & Initial State
Evaluation
G {i, gi}
Selection
PR
PS
Allocation
Next State
Replace Current State by Next State
F
Stop Criteria
T
Terminate Return Best State

Figure 5: Simulated Evolution General Flow Diagram

3.2.4 Stochastic Evolution

To escape the random search of SA, and random selection of SimE, research in AI has recently designed a new heuristic, called Stochastic Evolution (StocE). This algorithm takes as inputs: an initial state S_0, an initial control parameter p_0, and a stopping criteria R. This parameter "represents the expected number of iterations the StocE algorithm needs until an improvement in the cost with respect to the best solution seen so far takes place. It is suggested in that the best results are obtained with $10 < R < 20$. Upon termination, the algorithm returns the best solution found so far by the heuristic" [15]. During any iteration, the algorithm moves the current state to a new one that will be checked for consistency. All violations have to be un-done, before final validation. If the cost of the current state is the same as that of the state in the previous iteration, the bounds of the control parameter are enlarged; otherwise, this parameter is reset to its initial value. If a better state is found, the number of iterations is decreased by R to allow potential better states to be found.

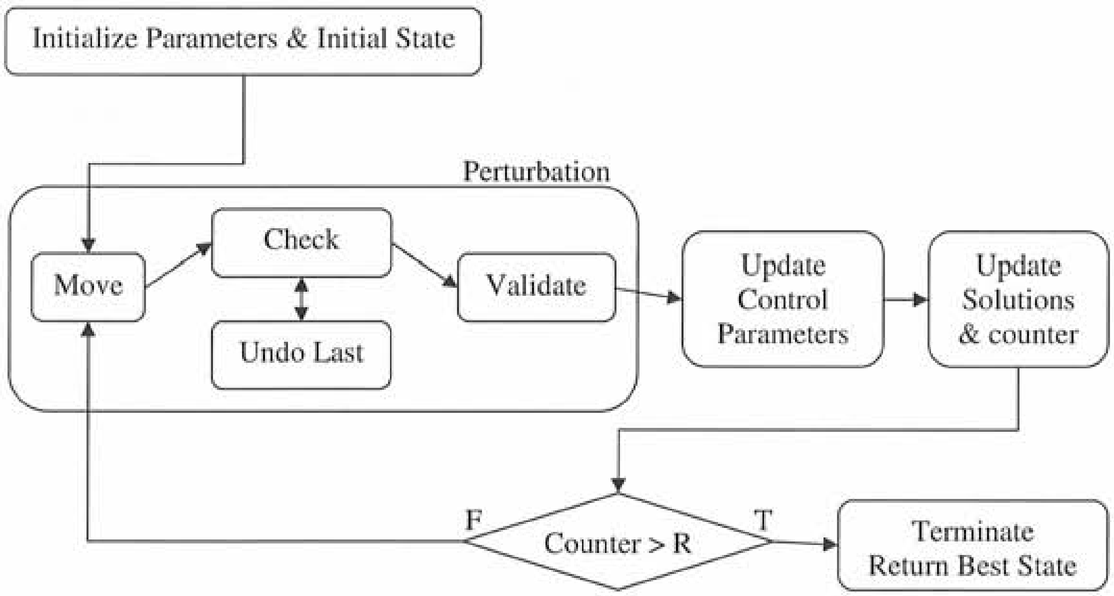

Initialize Parameters & Initial State
Perturbation
Move
Check
Undo Last
Validate
Update Control Parameters
Update Solutions & counter
F
Counter > R
T
Terminate Return Best State

Figure 6: Stochastic Evolution General Flow Diagram

3.2.5 Tabu Search

Tabu Search (TS) is a heuristic procedure proposed by Fred Glover to solve discrete combinatorial optimization problems. The basic idea is to avoid that the search for best solutions stops when a local optimum is found, by maintaining a list of non-acceptable or forbidden (taboo) solutions/costs, called Tabu list or Short-Term Memory (STM). Advanced TS algorithms suggest that some of the best solutions found so far be saved, for search diversification, in a list called Long-Term Memory (LTM). An additional list, called Medium-Term Memory (MTM) may be used to intensify locally the search, by keeping track of solutions, with estimate close enough to that of the best solutions in the LTM.

The use of these memory lists bring to light the fact that the updating process of the current and best solution does ignore those that were marked in the lists, which grow and shrink per iteration. Occasionally, moving the current solution to a "forbidden" one is allowed given a certain "aspiration" criteria, usually involving an improvement in cost from the current one.

As opposed to other algorithms, the current solution of the inner loop next iteration is selected from a set of N neighbor solutions, deduced from the actual current one by perturbing it N times. This solution will be overwritten if, at the beginning of the next iteration, a better solution was previously acknowledged in memory. [13]

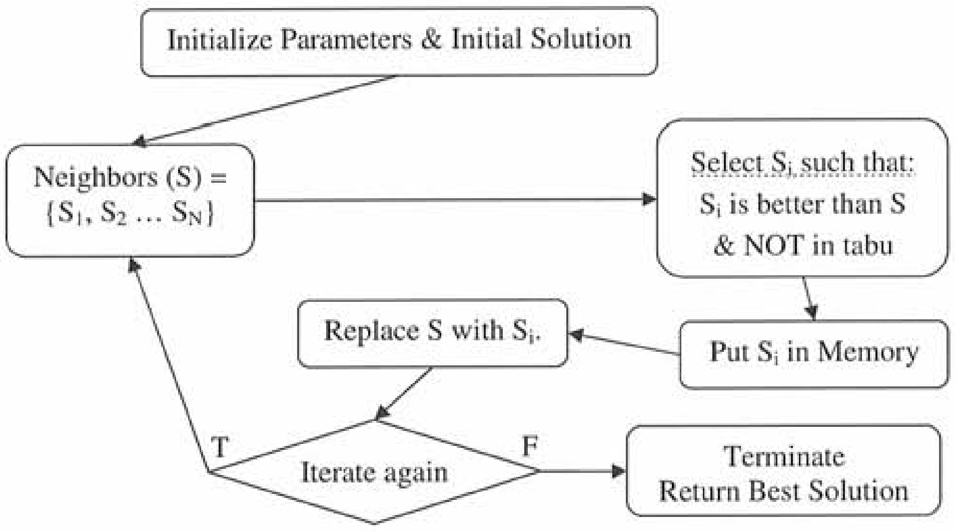

Initialize Parameters & Initial Solution
Neighbors (S) = {S1, S2 ... SN}
Select Si such that: Si is better than S & NOT in tabu
Put Si in Memory
Replace S with Si.
Iterate again
T
F
Terminate Return Best Solution

Figure 7: Tabu Search General Flow Diagram

Chapter 4 Attempted Non-Heuristic Approach

Maximum Parsimony Analysis can be solved using Fitch's algorithm. This algorithm helps evaluate parsimony, by assigning all sequences to be compared to the leave nodes of a given topology, and labeling internal nodes according to its children and parent. This procedure is done per location of each base within the sequence. The sequence could be a DNA or RNA or protein sequences. If the data length (number of characters within each sequence) is more than one, the algorithm is run in parallel: one run for each location of the sequences.

4.1 Standard Fitch Algorithm

The standard Fitch algorithm consists of 3 steps:

1. Bottom-Up Traversal: during which the algorithm assigns a set of possible values for each ancestral node. This is the union/intersection of the possible values for its left and right children nodes
2. Top-Down Traversal: during which the algorithm assigns a value for each ancestral node, either its parent's value if the latter is found in its set of states, or randomly from this set
3. Compute Parsimony: during which the algorithm accumulates the number of changes between each parent and its children nodes.

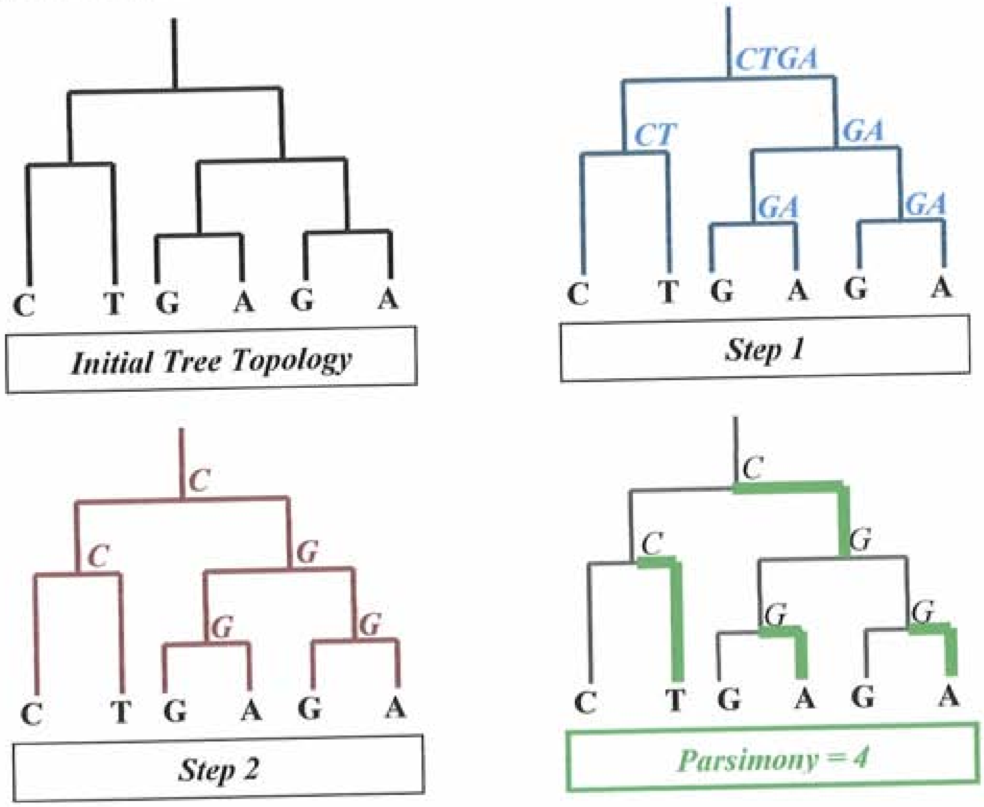


Figure 8: Sample Run of Fitch's Standard Algorithm

As illustrated in Figure 8, step 1 (blue) of the standard Fitch algorithm traverses the Initial Tree Topology (***bold and italic***) in a bottom-up fashion, assigning to each parent the union of the children states if no common sub-states were found. Then, in step 2 (dark red), the algorithm assigns randomly a value for the root chosen from the set computed in step 1, and for each ancestral node either its parent value, if found within its state, or randomly one of the values in its state. Step 3 (green) computes the parsimony score, and illustrates it by showing in **thick green lines** the edges connected to two nodes with different values.

The standard Fitch algorithm treats the same way all character changes between each internal node and each of its children nodes, thus ignoring the probable divergence in the outcome of such mutations (for ex: the mutation from G to A is considered equally valuable as a mutation from C to A). On the other hand, and due to the randomness of the choice of each node label, when the parent's label is not found in its states for the current location, different runs of the standard Fitch algorithm for the same tree topology may result in different trees, with the same parsimony cost. Figure 9 illustrate this drawback for 6 sequences, where the root node can randomly be assigned one of its 4 state values. For each of the many solutions give the same total parsimony score, each shows a different set of node value assignments and thus a different set of edges (**thick red lines**) connected to two nodes with different values.

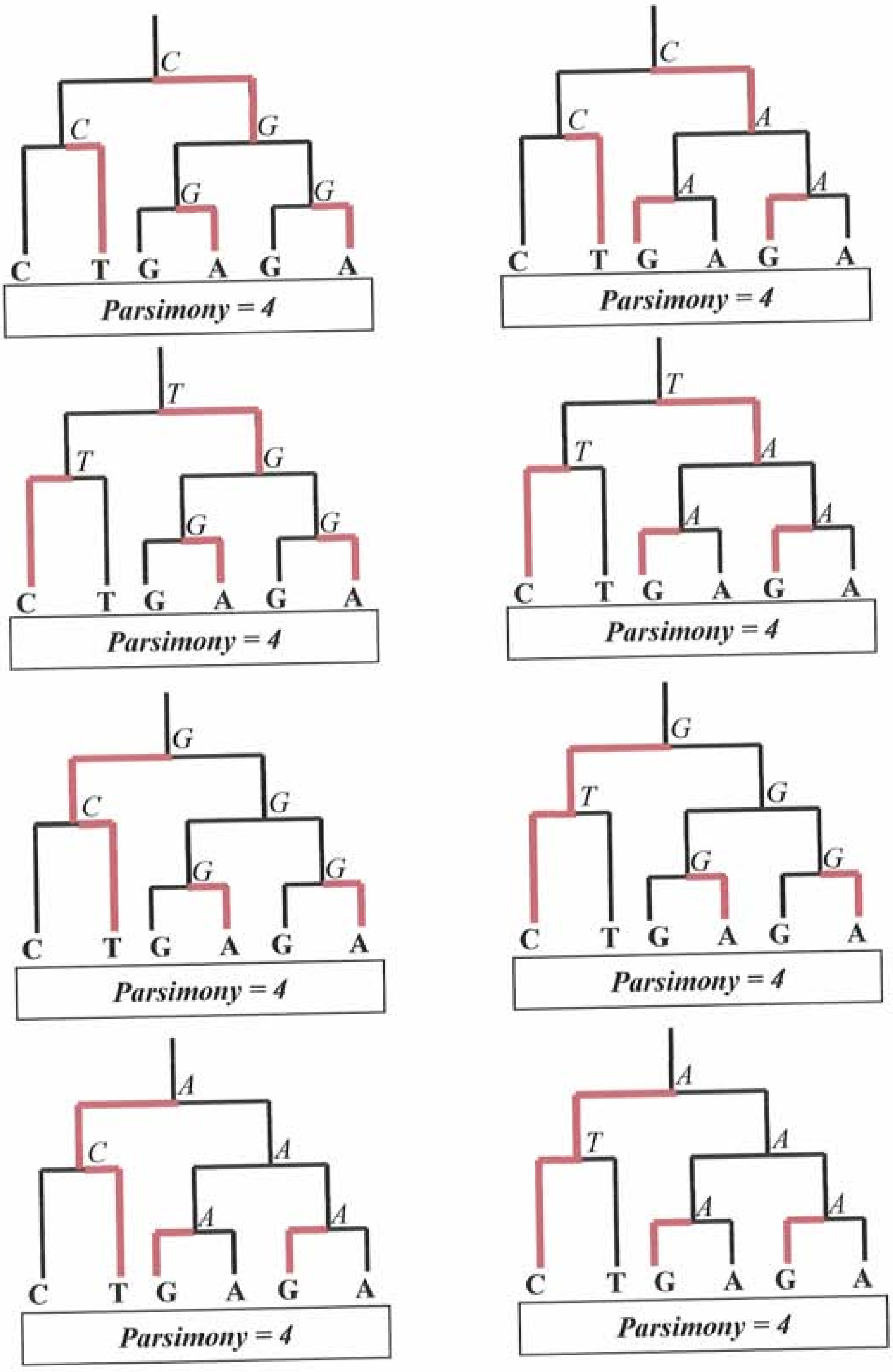


Figure 9: Standard Fitch Maximum Parsimony Trees for 6 Sequences

4.1.1 Main Standard Fitch Procedure

Input: Binary tree topology "T"
Pre-condition: Leaf nodes are labeled according to the sequences in concern
Output: Parsimony value: "$Cost$" & Binary fully labeled tree "T_2"
Post-condition: "$Cost$" corresponds to the number of changes in "T_2"
Steps:

```
Backup Tree $T$ into $T_2$
{Step 1: Bottom-Up Traversal}
Arrange the internal nodes of $T_2$ with respect to their access to leafs (BOTTOM-UP)
For each node $n_i$ of these nodes do
    Get left and right child of node $n_i$
    For each character location $l_j$ in value of node $n_i$ do
        Get state values $SV_L$ of left child for location $l_j$
        Get state values $SV_R$ of right child for location $l_j$
        If $SV_L \cap SV_R = \{\}$ then
            Set state Values of $n_i$ as the union of $SV_L$ and $SV_R$
        Else    Set state Values of $n_i$ as the intersection of $SV_L$ with $SV_R$
{Step 2: Top-Down Traversal}
Arrange the internal nodes of $T_2$ with respect to their access to leafs (TOP-DOWN)
For each node $n_i$ of these nodes do
    Get parent node $p_i$ of node $n_i$
    For each character location $l_j$ in value of node $n_i$ do
        Get state values $SV$ of node $n_i$ for location $l_j$
        If $n_i$ is root node of $T_2$ OR character of $p_i$ is NOT found in $SV$ then
            Assign random state value from $SV$ to character of $n_i$ at location $l_j$
        Else    Assign character of $p_i$ at location $l_j$ to character of $n_i$ at location $l_j$
{Final Step: Computing Total Score}
Reset $Cost$
For each node $n_i$ of these nodes do
    Get parent node $p_i$ of node $n_i$
    If $n_i$ is NOT root of $T_2$ then
        For each character location $l_j$ in value of node $n_i$ do
            If character of $n_i$ at location $l_j$ is NOT character of $p_i$ at location $l_j$ then
                Increment $Cost$ by 1
Return $\{Cost , T_2\}$
```

4.1.2 ArrangeInternalNodes Procedure

Input: Binary tree topology: "*T*" & Boolean flag "*TopDown*"

Pre-conditions:

- Leaf nodes are labeled according to the sequences in concern
- $TopDown = \begin{cases} FALSE & for \quad Step\_1 \\ TRUE & for \quad Step\_2 \end{cases}$

Output: Vector of nodes "*Nodes*"

Post-condition: "*Nodes*" contains all internal nodes in the desired order, with a priority for nodes with both children having the same tree level.

Steps:

```
Get all internal nodes into a dynamic circular list Internal.
Create an empty dynamic list Nodes.

While list Internal is NOT empty yet do
    Let n be the first available internal node in list Internal.
    Get left and right children nodes of n into Left and Right.

    Set flag T1 iff Nodes does not already have node n, AND both Left and Right are leafs.
    Set flag T2 iff node Right is leaf, but Left is an already selected internal node.
    Set flag T3 iff node Left is leaf, but Right is an already selected internal node.
    Set flag T4 iff both nodes Left and Right are already selected internal nodes.

    If either T1 OR T2 OR T3 OR T4 is/are set then
        Add node n to list Nodes.
        Remove node n from circular list Internal.
    Else
        Leave node n in list Internal, but jump to next available node index.

If TopDown then
    Reverse list Nodes so that the first one entered is the last to deal with.

Return Nodes
```

4.2 Weighted Fitch Algorithm

The weighted Fitch algorithm consists of 3 steps:

1. Bottom-Up Traversal: during which the algorithm assigns a set of possible values for each ancestral node, which is the union/intersection of those for its left and right children nodes. It also propagates up, in the tree, min costs for each character in alphabet, initializing each leaf node to have a cost of "0" for its character, and "∞" for the other characters.
2. Top-Down Traversal: during which the algorithm assigns a value for each ancestral node, one of the characters that have the minimum cost. The parent's value is chosen if found in its set of states, otherwise a value is randomly chosen from the set of characters in state with minimum cost.
3. Compute Parsimony: during which the algorithm accumulates the weight of all changes between each parent and its children nodes.

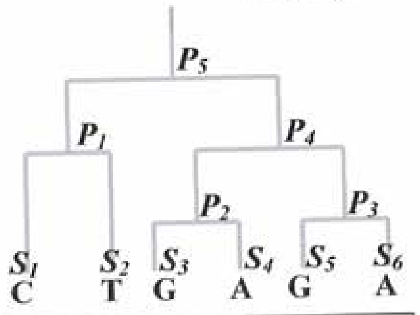
The score/weights of each character-to-character change are filled into a matrix that will be used during the run of the algorithm, when computing for each node sequence the minimum weights of changing from each alphabet character to that of the left and right children sequences at each location. The algorithm fills these minimum scores in another scoring matrix.

	A	T	C	G
A	0	0.5	0.5	0.9
T	0.5	0	0.6	0.3
C	0.5	0.6	0	0.3
G	0.9	0.3	0.3	0

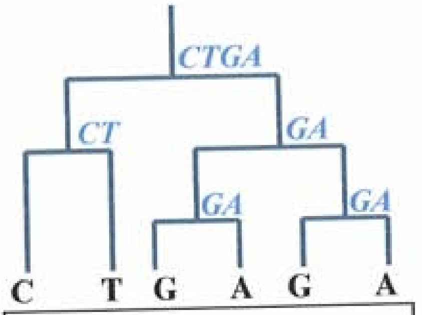
Scoring Matrix

	A	T	C	G
S_1	∞	∞	0	∞
S_2	∞	0	∞	∞
S_3	∞	∞	∞	0
S_4	0	∞	∞	∞
S_5	∞	∞	∞	0
S_6	0	∞	∞	∞
P_1	1	0.6	0.6	0.6
P_2	0.9	0.8	0.8	0.9
P_3	0.9	0.8	0.8	0.9
P_4	1.8	1.6	1.6	1.8
P_5	2.8	2.2	2.2	2.4

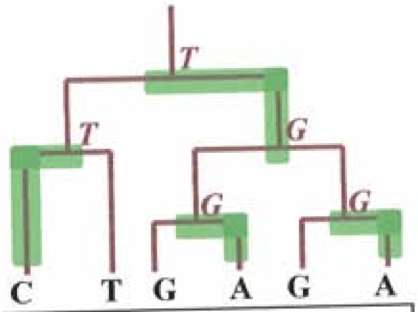
Min Scores Matrix



Initial Tree Topology



Step 1



Step 2

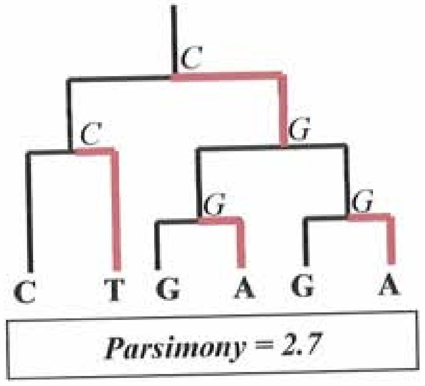
Parsimony = 0.3 + 0.9 + 0.9 + 0.6 = 2.7

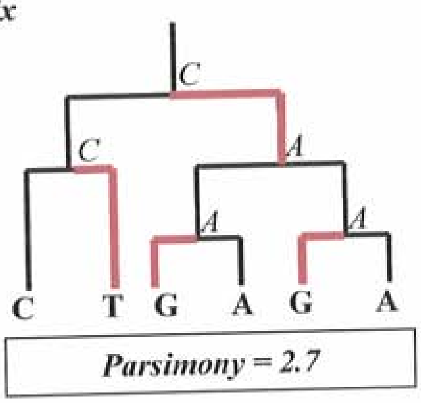
Figure 10: Sample Run of Weighted Fitch Algorithm

As illustrated in Figure 10, the leaf sequences of the ***Initial Tree Topology*** are respectively {S_1, S_2 … S_6} and the ancestral/internal nodes {P_1, P_2 … P_5}. Step 1 of the algorithm, is show in blue. The tree topology is traversed in a bottom-up fashion, and each parent is assigned the union of the children states, in the case where no common sub-states were found. The "***Min Scores Matrix***" illustrates the propagation up of min costs for each character in alphabet, according to the given "***Scoring Matrix***", with leaf nodes being initialized to have a cost of 0 for a character in its set and ∞ for the other characters. Then, step 2 (dark red) assigns randomly a value for the root, and for each ancestral node either its parent value or one chosen randomly from the set of characters in its state with minimum score. Step 3 (green) computes the parsimony score by adding all scores between parent and children nodes.

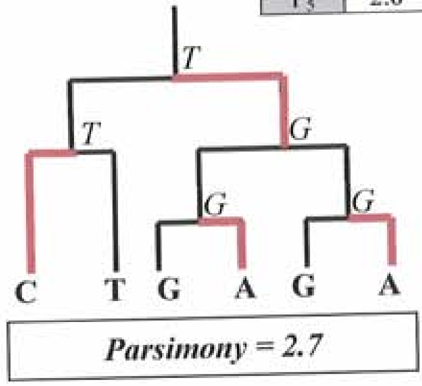
	A	T	C	G
A	0	0.5	0.5	0.9
T	0.5	0	0.6	0.3
C	0.5	0.6	0	0.3
G	0.9	0.3	0.3	0

Scoring Matrix





	A	T	C	G
S_1	∞	∞	0	∞
S_2	∞	0	∞	∞
S_3	∞	∞	∞	0
S_4	0	∞	∞	∞
S_5	∞	∞	∞	0
S_6	0	∞	∞	∞
P_1	1	0.6	0.6	0.6
P_2	0.9	0.8	0.8	0.9
P_3	0.9	0.8	0.8	0.9
P_4	1.8	1.6	1.6	1.8
P_5	2.8	2.2	2.2	2.4



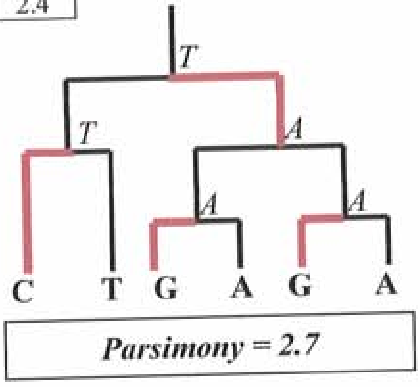


Figure 11: Weighted Fitch Maximum Parsimony Trees for 6 Sequences

4.2.1 Main Weighted Fitch Procedure

<u>Input</u>: Binary tree topology: "T" & Scoring Matrix "Mat"
<u>Pre-condition</u>: Leaf nodes are labeled according to the sequences in concern
For each leaf node, and each character location, the weight of character in location is initialized to 0 and the rest are initialized to ∞.
$Rows(Mat) = Columns(Mat) = Length(Letters)$
$Mat[i, j] = S(character_i, character_j)$
<u>Output</u>: Parsimony value: "$Cost$" & Binary fully labeled tree: "T_2"
<u>Post-condition</u>: "$Cost$" corresponds to the weight of all changes in "T_2"
<u>Steps</u>:

```
Backup Tree $T$ into $T_2$
{Step 1: Bottom-Up Traversal}
Arrange the internal nodes of $T_2$ with respect to their access to leafs (BOTTOM-UP)
For each node $n_i$ of these nodes do
    Get left and right child of node $n_i$
    For each character location $l_j$ in value of node $n_i$ do
        Get state values $SV_L$ of left child for location $l_j$
        Get state values $SV_R$ of right child for location $l_j$
        If $SV_L \cap SV_R = \{\ \}$ then
            Set state Values of $n_i$ as the union of $SV_L$ and $SV_R$
        Else
            Set state Values of $n_i$ as the intersection of $SV_L$ with $SV_R$

        Get scores of all characters in left children at location $l_j$
        Get scores of all characters in left children at location $l_j$
        For each alphabet character $ch1$ do
            Initialize to ∞ both min scores of character $ch1$ for left and for right
            For each alphabet character $ch2$ do
                Update min left score of $ch1$ after weighting change of $ch1$ to $ch2$ in left
                Update min right score of $ch1$ after weighting change of $ch1$ to $ch2$ in right
            Set score of $ch1$ as the sum of both min scores of $ch1$
```

```
{Step 2: Top-Down Traversal}
Arrange the internal nodes of $T_2$ with respect to their access to leafs (TOP-DOWN)
For each node $n_i$ of these nodes do
    Get parent node $p_i$ of node $n_i$
    For each character location $l_j$ in value of node $n_i$ do
        Get state values $SV$ of node $n_i$ for location $l_j$
        If $n_i$ is root node of $T_2$ OR character of $p_i$ is NOT found in $SV$ then
            Assign randomly a state value, with minimal score, from $SV$ to character of $n_i$
        Else
            If character of $p_i$ is found in $SV$ then
                Assign character of $p_i$ at location $l_j$ to character of $n_i$

{Final Step: Computing Total Score}
Reset $Cost$
For each node $n_i$ of these nodes do
    Get parent node $p_i$ of node $n_i$
    If $n_i$ is NOT root of $T_2$ then
        For each character location $l_j$ in value of node $n_i$ do
            If character of $n_i$ at location $l_j$ is NOT character of $p_i$ at location $l_j$ then
                Increase $Cost$ by weight of change between these two characters
Return $\{Cost \quad , \quad T_2\}$
```

4.3 Advantages and Disadvantages of these Approaches

Maximum parsimony searches for the optimal (minimal) tree. In this process, more than one minimal tree may be found. In order to guarantee to find the best possible tree an exhaustive evaluation of all possible tree topologies has to be carried out. This fact renders the Maximum Parsimony Phylogenetic Tree NP-hard. In other words, the process becomes impossible to solve in polynomial time, when there are more than 12 OTUs in a dataset.

The main weakness of the standard Fitch algorithm is that it treats all kind of mutations and changes equally, meaning that it applies the same score to a character that varies, irrespective of the new character into which it becomes. This is a weakness because most of the time, scientists are not interested in just a score that reflects the amount of mutation generally, but want to give different weights to the score depending on how relevant a particular mutation is with respect to the goal of one's research.

To prove and improve this weakness, the weighted Fitch algorithm offers to apply, instead of a fixed score to all possible state changes, a complex scoring technique that takes into consideration the relevance of any particular change to the studies performed and applies the score accordingly.

Despite the obvious improvements with respect to the standard version, the algorithm still has a flaw in the scoring technique. This flaw lies in the fact that it treats the sequences analyzed in a parallel way; i.e. it compares characters in a sequence separately, which makes it lose some information related to the way the characters interact together. Therefore, the final score we obtain may not be as accurate as we would expect it to be, as it reflects the results of comparing the characters, grouped by locations in sequences, in parallel and independently. One might think that even if the comparison is done in parallel, still the total score should be as meaningful as possible since we are anyway comparing all characters in all sequences and applying meaningful scoring, but this is not the case. Cases whereby two character mutations should result differently, according to the context (group of characters) they are in, are omitted when comparing characters in parallel, using a valid scoring matrix for all possible cases, giving them the same score. For instance, having the mutation (ACT → ACC) and (GCT → GCC) will give us the same score: S (T, C). Treating characters in a word, independently from each others, might mislead the analysis, since these mutations could influence differently the evolution process

Some already proposed alternatives to Fitch's Algorithm, for finding the optimal parsimony phylogenetic tree, are the famous Branch-and-Bound algorithm and a combination of one or more Heuristic Search algorithms. The first is the Branch and Bound algorithm, which is a variation on maximum parsimony that helps finding the minimal tree by reducing the number of possible trees to evaluate. Thus, a larger number of taxa can be evaluated, but it is still limited. Heuristic search is an approach with systematic addition and rearrangement (branch swapping) of OTUs, not guaranteed to find the best tree.

4.3.1 MPA: Branch-and-Bound

The optimal set of solutions has at least the score of the bounding solution. If during the construction of a solution the score is higher/lower than the bounding score, then all complete solutions containing this sub-solution have a higher/lower score than the bounding solution.

"Therefore, the sub-solutions having higher score are worse than the bounding solution and we can stop the construction of these solutions, this means cutting the sub-tree under sub-solutions with a worse score." [8]

Figure 12 illustrates the Branch-And-Bound process for a set of 5 sequences. We start by building a tree with 3 of these sequences, and evaluate it and record that value as the "Min Cost". Then, when adding a new sequence, we branch only from the candidate topologies that have a score ≤ last "Min Cost" value recorded, and update the latter if needed.

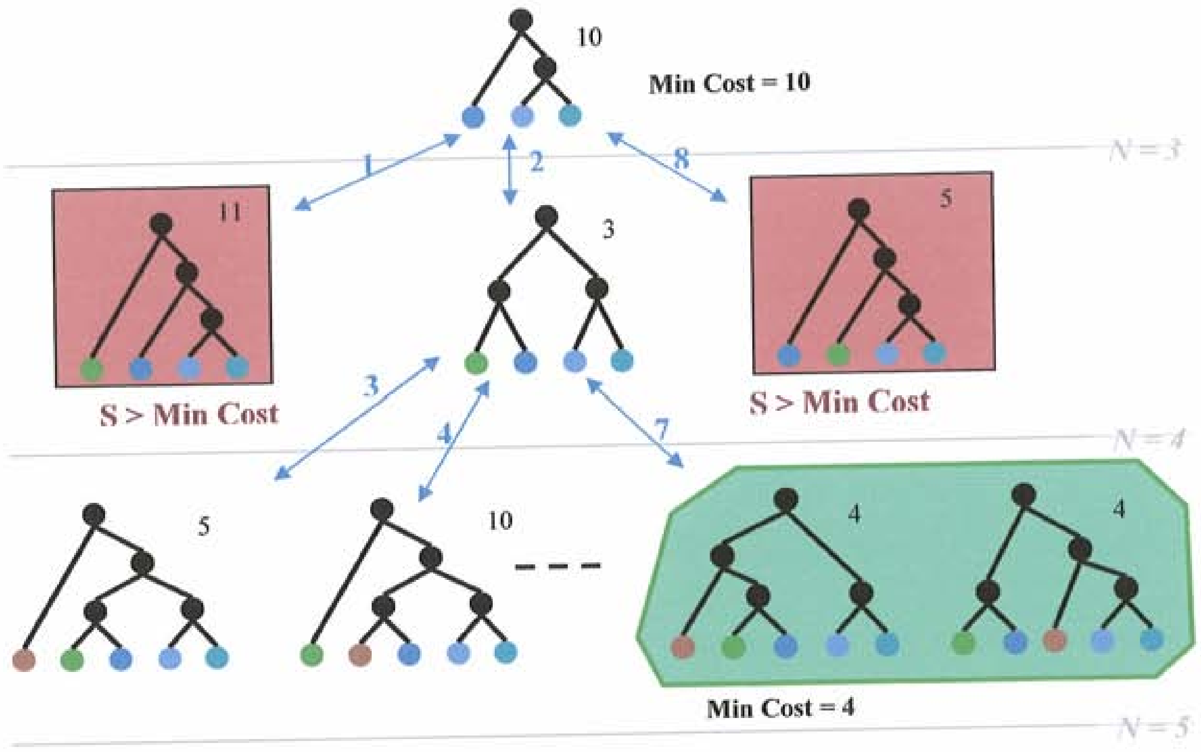


Figure 12: Sample Run of Branch-And-Bound for 5 Sequences

The worst-case time complexity of the Branch-And-Bound algorithm is the same as the complexity of exhaustive search, which is worse than exponential. Nevertheless, with a wisely chosen bound, many sub-trees will be cut and therefore the running time will decrease. Sometimes, a special traversal order finds better solutions faster.

```
Input:  Tree Topology: "$T$"                      Set of Sequences "$Species$"
        Scoring Matrix: "$ScoreMat$"              Set of characters: "$Letters$"
        Optimal Result found so far: Parsimony "$MinCost$"
                                     Set of trees "$MinTrees$"
Pre-condition:   $NumberOfLeafs(T) + Length(Species) = N$
                 $Rows(ScoreMat) = Columns(ScoreMat) = Length(Letters)$
                 $ScoreMat[i, j] = S(character_i, character_j)$
                 First call of method: $T$ has 3-leafs,
                    $MinCost = \infty$,          $MinTrees$ is empty
Output:   Optimal Parsimony: "$Cost$"     &     Set of optimal trees: "$Trees$"
Post-condition:  Value "$Cost$" corresponds to the minimal number of changes
                 Set "$Trees$" contains all trees, which evaluation leads to "$Cost$"
Steps:
  Compute the weighted parsimony score $S$ of input topology $T$
  If $S > MinCost$ then
      Return $\{MinCost \quad MinTrees\}$
  If there are no more sequences to consider then
      If $S < MinCost$ then
          Update the value of $MinCost$
          Empty the search space list
      If $S \leq MinCost$ then
          Add topology $T$ to search space list $MinTrees$
      Return $\{MinCost \quad MinTrees\}$
  Extract set of all branches of topology $T$
  Extract first available specie to handle $Specie$ & Remove it from given set of species
  For each branch $B_i$ of topology $T$ do
      Add a new branch connected at one endpoint to $B_i$ and the other labeled by $Specie$
      Update $MinCost$ and $MinTrees$ with those of recursive call on updated topology $T$
      Remove newly added branch to $B_i$ labeled by $Specie$
  Return $\{MinCost \quad MinTrees\}$
```

Chapter 5 Suggested Improvement to Fitch:

As already discussed, Fitch algorithms treats all changes quasi-identically, by working in parallel per character location. The Weighted Fitch algorithm deals with this matter, by assigning a cost to each character-to-character change.

However, neither standard nor weighted Fitch algorithms handle the fact that one change in a certain word might affect differently the outcome of the word. There is no character, within any DNA/RNA/protein sequence, that is totally independent of at least the direct neighbor characters in the sequence. This group of characters, called a word, is supposed to decode a specific functionality in the final result.
For instance:

Standard Fitch → $S(ATG,ATT)=S(CTG,CTT)=S(ATG,ATC)$

Weighted Fitch → $S(ATG,ATT)=S(CTG,CTT)\neq S(ATG,ATC)$

Assuming that the mutation of the word CTG to CTT is less/more harmful than transforming ATG to ATT, the computed score should also include a weight for that factor. This introduces the idea of having a "word scoring matrix". This matrix adds to the cost of each character-to-character mutation a score that reflects how it affects the whole word. Therefore, for example: $S(ATG,ATT)\neq S(CTG,CTT)\neq S(ATG,ATC)$

Another disadvantage of Fitch algorithms is the non-inclusion of the environmental and/or time factor of each mutation. In reality, gene mutations are very often affected directly by its entourage. This fact, research subject by itself, can be handled by allowing the input tree topology to include edge weights that estimate the effect of external factors on the mutation.

Therefore, given the two suggested enhancements discussed above, the score of a certain character-to-character mutation will be treated differently depending on its context and location within the sequences in concern. However, even though no 2 mutations will be treated the same way, this whole improvement does not resolve the load of computations required by Fitch algorithms, enough to put aside the need for Heuristics to try to solve the phylogenetic problem.

For this, to solve the problem of phylogenetic trees using heuristics, the first step is to establish an encoding schema, to represent the trees into data-structures, and map real-life process on elements of these trees into functional operators to be used by the heuristics. In the next chapter, we describe the suggested general way to encode the problem into instances of different heuristic algorithms. A suggested use or inclusion of the environmental and/or time factors, during the evaluation process of different phylogenetic trees, will be discussed in the chapter concerning the suggested encoding schema.

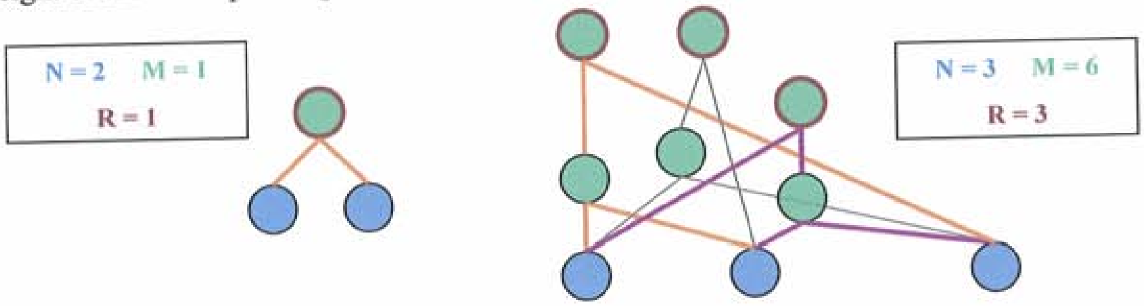
Chapter 6 Suggested Common Heuristic Encoding

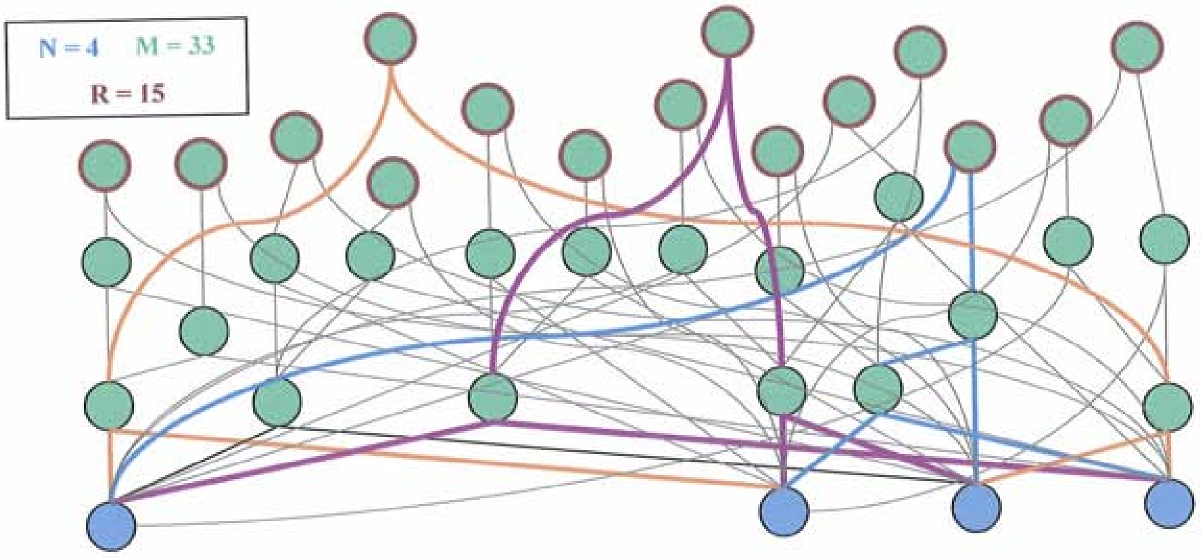
6.1 Analysis

Let SL be the set of sequences to compare, shown as leave nodes in a phylogenetic tree (colored below in blue). Let N be the number of sequences in the set SL.
Let R be the number of rooted trees, exponentially related to N:

i.e.: $R = \dfrac{Fact(2N-3)}{Fact(N-2) * 2^{N-2}}$ [3]

The root nodes of those trees are highlighted below in dark red. For visibility purposes, we show only few trees (colored entirely in blue or orange or **violet**).
A "parent sequence" can be defined as the sequence from which two sequences are derived by mutation or perturbation. These sequences are called "children sequences". Let SP be the set of all possible parent sequences, colored below in sea green. Let M = |SP|.
A level-1 parent, denoted by L1-Parent, is a parent sequence having both children sequences belonging to the set of sequences SL. Let P be the number of those parent sequences.
In our phylogenetic context, for a given tree topology, the level of a sequence is defined as the height of the corresponding node in the tree.





If each tree has $N-1$ parent nodes (holding parent sequences), then the total number of parent sequences *M*, including L1-Parents, satisfies $M \le (N-1)*R$.
In order to find the value of *P*, number of L1-Parents, we can suggest the following numbering of internal nodes:
If S_i and S_j both have SP_K as a parent, then k is computed as a function of both indices i and j. In other terms: $k = F(i,j) \quad with \quad (i,j) \in [1,N] \quad and \quad k \in [1,P]$
The following tables explicit the mathematical generalization process that leads to the formula for that function: $k = F(i,j)$

$\underline{N=3}$

i	j	K	$j-i$	$K-i$
1	2	1	1	0
1	3	3	2	2
2	3	2	1	0

$\underline{N=4}$

i	j	K	$j-i$	$K-i$
1	2	1	1	0
1	3	4	2	3
1	4	6	3	5
2	3	2	1	0
2	4	5	2	3
3	4	3	1	0

$\underline{N=5}$

i	j	K	$j-i$	$K-i$
1	2	1	1	0
1	3	5	2	4
1	4	8	3	7
1	5	10	4	9
2	3	2	1	0
2	4	6	2	4
2	5	9	3	7
3	4	3	1	0
3	5	7	2	4
4	5	4	1	0

$\underline{N=6}$

i	j	K	$j-i$	$K-i$
1	2	1	1	0
1	3	6	2	5
1	4	10	3	9
1	5	13	4	12
1	6	15	5	14
2	3	2	1	0
2	4	7	2	5
2	5	11	3	9
2	6	14	4	12
3	4	3	1	0
3	5	8	2	5
3	6	12	3	9
4	5	4	1	0
4	6	9	2	5
5	6	5	1	0

By generalization, for each couple (*i, j*), we can see that we have:
If $j-i=1$ **then** $\quad k=i$
Else $\quad k = i + [N-1] + \ldots + [N-(j-i-1)]$

Therefore: $k = F(i,j) = i + \sum_{s=1}^{j-i-1}(N-s)$

When applying the formula $F(i,j) = i + \sum_{s=1}^{j-i-1}(N-s)$ on any set of couple (*i, j*) the suggested numbering was recovered, i.e.: the numbering starts at $i=1$ and $j=2$, and increments by one whenever $(j-i)=1$, then when $(j-i)=2$, and so on until $(j-i)=N-1$

Since the couples *(i, j)* are sorted such that i < j, the value of *P* is found for the couple *(1, N)*. Thus:

$$P = F(1,N) = 1 + \sum_{s=1}^{N-2}(N-s) = 1 + \sum_{s=1}^{N-2}(N) - \sum_{s=1}^{N-2}(s) = 1 + (N-2)*(N) - \sum_{s=1}^{N-2}(s)$$

$$P = 1 + (N-2)*(N) - \frac{(N-2)(N-1)}{2} = 1 + (N-2)*\left[N - \left(\frac{N}{2} - \frac{1}{2}\right)\right]$$

Therefore:

$$P = 1 + \frac{(N-2)*(N+1)}{2}$$

This formula could help speeding up the algorithms, when traversing trees in a Bottom-Up fashion.

The following algorithm, useful when creating initial individual or group of solutions or states, to be used by the different algorithms, computes the exact number of parent sequences M:

```
Input:          Number of Sequences to compare: $N$
Pre-condition:  $N = |S_L| \geq 2$
Output:         Exact Number of Parent Sequences $M$
Post-condition: $1 \leq M = |S_P| \leq (N-1)*OddFactorial(2N-3)$
Steps:
    Create an empty dynamic list to store the parent sequences
    Create an empty dynamic list to store the sets of these parents' children sequences
    Create an empty dynamic list to store the sets of these parents' descendent leaf sequences
    {Initialize all 3 Lists of Sequences}
    For each leaf sequence $S_i$ do
        Add $S_i$ to the list of parent sequences
        Add an empty dynamic set to the list of children sequences
        Add an empty dynamic set to the list of descendent leaf sequences
    Initialize total number of handled sequences $T$ to $N$
    Initialize effective number of parent sequences $M$ to 0
    For each handled sequence $Seq_i$ do
        For each sequence $Seq_j$ previously handled do
            If Sequences $Seq_j$ and $Seq_i$ DO NOT have same children then
                If $Seq_j$ and $Seq_i$ have same direct parent then
                    Increment $M$ & add $P_M$ to list of parent sequences
                    Create an empty set of children sequences
                    Create an empty set of descendent leaf sequences
                    Uniquely add $Seq_i$ and $Seq_j$ to set of children sequences
            **If** $Seq_i$ is a leaf sequence **then**
                Uniquely add $Seq_i$ to the set of descendent leaf sequences
            **Else**
                Extract all leaf children of $Seq_i$
                Uniquely add these leaf sequences to the set of descendent leaf sequences

            **If** $Seq_j$ is a leaf sequence **then**
                Uniquely add $Seq_j$ to the set of descendent leaf sequences
            **Else**
                Extract all leaf children of $Seq_j$
                Uniquely add these leaf sequences to the set of descendent leaf sequences

            Add set of children sequences to list of children sequences sets
            Add set of descendent leaf sequences to list of descendent leaf sequences sets
            Increment $T$
    **Return** $M$
```

To retrieve the connections for all parent sequences, instead of just returning the number of parent sequences, the above algorithm should remove the initial leaf sequences (sequences that we are initially comparing) from the global lists: list of parent sequences, list of children sets, and list of descendent leaf sequences sets.

```
**For** each handled sequence $Seq_i$ **do**
    **If** $Seq_i$ is a leaf sequence **then**
        Remove $Seq_i$ from list of parent sequences
        Remove its children set from list of children sequences sets
        Remove its descendent leaf sequences set from list of descendent leaf sequences sets
**Return** list of children sets
```

6.2 Encoding Schema Design

Establishing any encoding schema starts with designing/suggesting a data-structure to be used during the run of the algorithms, then accordingly implement some general functions, concerned mainly with the EVALUATION of an instance of the problem, the CREATION of new instances and the PERTURBATION of an instance into another. Some algorithms, as for instance Genetic Algorithm, may require a function to handle the RECOMBINATION of a set of instances into another.

6.2.1 Suggested Data-Structure

Since the primary focus of our problem spins around binary trees, and handles a fixed set of nodes for a given number of sequences, one suggestion would be to use two array data-structures. One of them is fixed and handles all sequences, to compare and parent sequences. The second one is composed of integers representing the selected edges connecting two nodes, and might be changed when applying the operators, mainly the perturbation ones.

Let N be the number of Sequences to compare, R the number of rooted trees, M the number of internal sequence nodes, and P the number of L1-Parents.

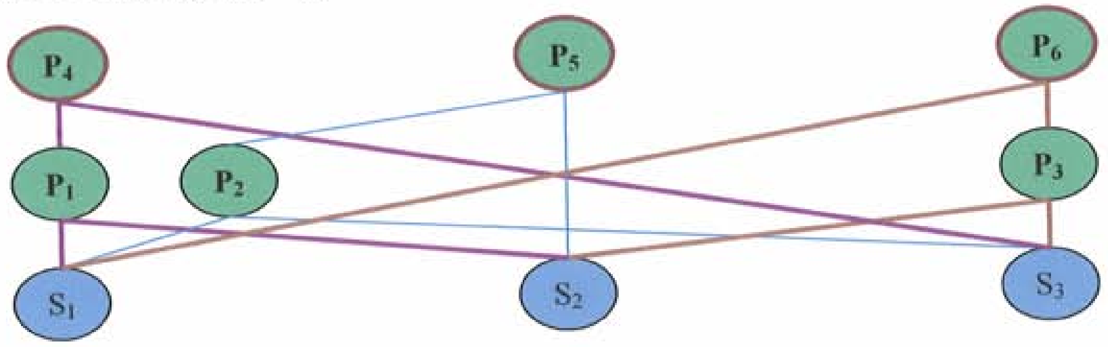
For convenience, the first location of the arrays, left as garbage, can be used to refer to the root of the corresponding phylogenetic tree. Then, the length of the two data-structures would be $L = N + M + 1$

In an instance S of the suggested integer data-structure, if 2 nodes *i* and *j*, are connected to node k, such that $Seq[i]$ and $Seq[j]$ have the same parent node $Seq[k]$ then: $S[i] = S[j] = k + N$

Initially all values in the two data-structures are either 0 or NULL. During the evaluation of the instance, the parent sequences are generated if previously set to NULL. In some cases, an instance might be intentionally ignored, thus erasing all generated parent sequences.

In this way, some of the parent sequences, ignored or never used, would not be generated, leaving the space for other more critical sequences.

For $N = 3 \Rightarrow M = 6 \Rightarrow L = 10$



	Root	Sequences			Parent Sequences					
Seq		S1	S2	S3	P1	P2	P3	P4	P5	P6

	0	1	2	3	4	5	6	7	8	9
S1	4	1	1	4	4	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
S2	5	2	5	2	0	5	0	0	0	0

	0	1	2	3	4	5	6	7	8	9
S3	6	6	3	3	0	0	6	0	0	0

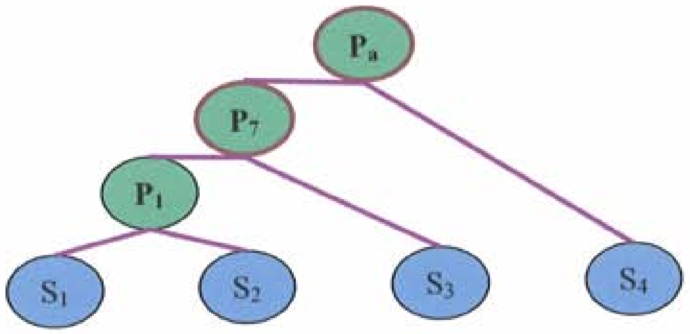
The data-structures S_1, S_2, S_3 are respectively related to the trees drawn in the above figure. For instance:

S1 [0] = 4
$\rightarrow$ Root of current tree is P_4.
S1 [1] = S1 [2] = 1
$\rightarrow$ Parent of S_1 and S_2 is P_1.
S1 [3] = S1 [4] = 4
$\rightarrow$ Parent of S_3 and P_1 is P_4.
S1 [i] = 0 for any index in [5, 9]
$\rightarrow$ NO Parent in tree for P_{i-3}.

For $N = 4 \Rightarrow M = 33 \Rightarrow L = 38$

	Root	Sequences				Parent Sequences										
Seq		S1	S2	S3	S4	P1	P2	P3	P4	P5	P6	P7		Pa		P33
	0	1	2	3	4	5	6	7	8	9	10	11	...	4+a	...	37
S	a	1	1	7	a	7	0	0	0	0	0	a		0		0

In other terms, we have the following decoded tree:



6.2.2 Evaluation Function

Start with the leaf sequences, and generate the parent sequence for each pair. This generation process can be done in many ways (enhanced Weighted Fitch, another heuristic ...). What ever way is chosen, the sequences are compared one pair at a time (not the entire set of sequences). One suggestion for generating parent sequences, according to a given scoring matrix, would be to assign, for each location of the parent sequence, a base that will eventually lead to the lowest cost of change with the base that is in that location of the child sequence.

Since we are dealing with maximum parsimony or minimum change between two sequences (a parent and one of its children), the evaluation function spots the changes in each pair (parent – child) and adds the corresponding weight to the total score. The weight is looked up in a given scoring matrix, depending on the alphabet used in the sequences (DNA / RNA / protein), and some other environmental parameters. The scores at each leaf node, representing an initial sequence, is ZERO.

Since the evaluation function tends to minimize the weights/scores of changes between all parent and children sequences, maximization problems will have to negate the result. Indeed, in mathematical terms, minimizing a function is identical to maximizing the negation of all the values of this function.

Consider the example of comparing three DNA sequences of one base each, and assume the following:

→ The scoring matrix is as follows:

	A	C	G	T
A	0	0.5	0.9	0.475
C	0.5	0	0.3	0.6
G	0.9	0.3	0	0.25
T	0.475	0.6	0.25	0

→ The environmental parameters are plugged in the evaluation equation through the use of the children sequences levels when generating the parent. All initial sequences are at level ZERO.

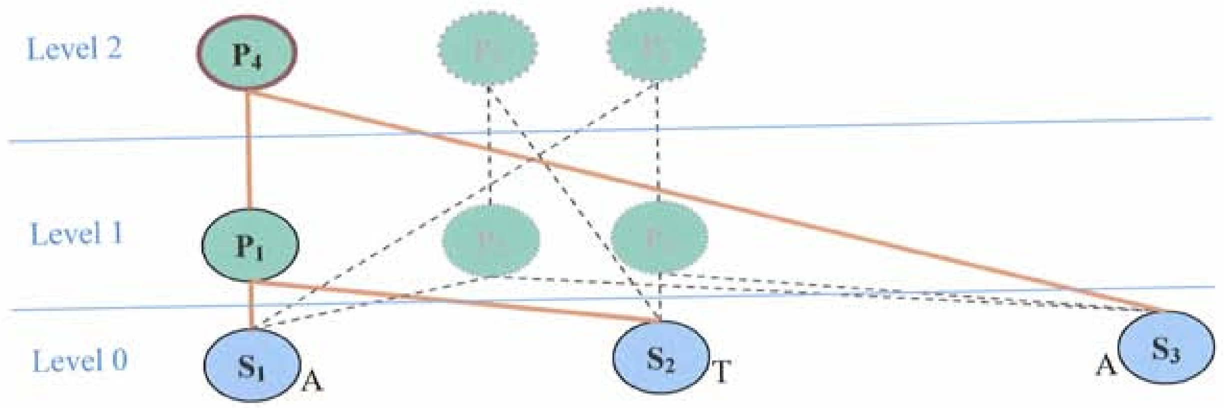
$$Score(Parent) = \begin{array}{l} [1 + Level(Child_1)] * [ScoreOfChange(Parent, Child_1) + Score(Child_1)] \\ + [1 + Level(Child_2)] * [ScoreOfChange(Parent, Child_2) + Score(Child_2)] \end{array}$$

→ $S_1 = Seq[1] = "A"$ $S_2 = Seq[2] = "T"$ $S_3 = Seq[3] = "A"$

	Root	Sequences			Parent Sequences					
	0	1	2	3	4	5	6	7	8	9
Seq		A	T	A	P1	P2	P3	P4	P5	P6

6.2.2-a: Example 1:

→ Assume the tree being evaluated is the following:



The corresponding encoded solution will be as follows:

	Root	Sequences			Parent Sequences					
	0	1	2	3	4	5	6	7	8	9
Seq		A	T	A	P1	P2	P3	P4	P5	P6
	0	1	2	3	4	5	6	7	8	9
S	4	1	1	4	4	0	0	0	0	0
level	2	0	0	0	1					

$$Score(P_1) = \begin{array}{l} [1+Level(S_1)]*[ScoreOfChange(P_1,S_1)+Score(S_1)] \\ +[1+Level(S_2)]*[ScoreOfChange(P_1,S_2)+Score(S_2)] \end{array}$$

$$Score(P_4) = \begin{array}{l} [1+Level(P_1)]*[ScoreOfChange(P_4,P_1)+Score(P_1)] \\ +[1+Level(S_3)]*[ScoreOfChange(P_4,S_3)+Score(S_3)] \end{array}$$

Since S_1 = "A" and S_2 = "T", P_1 could be either "A" or "T", forcing P_4 to be "A" in the first case, and "A" or "T" in the second, given that S_3 = "A". Thus computing the score for each case will give different scores for the corresponding values of P_4.

P_1	$Score(P_1)$	P_4	$Score(P_4)$
"A"	$\{[1+0]*[0+0]\}+\{[1+0]*[0.475+0]\}=0.475$	"A"	$\{[1+1]*[0+0.475]\}+\{[1+0]*[0+0]\}=0.95$
"T"	$\{[1+0]*[0.475+0]\}+\{[1+0]*[0+0]\}=0.475$	"A"	$\{[1+1]*[0.475+0.475]\}+\{[1+0]*[0+0]\}=1.9$
		"T"	$\{[1+1]*[0+0.475]\}+\{[1+0]*[0.475+0]\}=1.425$

Since we ought to have the tree with values giving the lowest cost, we will have the following set of values at the end of the evaluation: $P_1 = P_4 =$ "A", with a total score of 0.95

	Root	Sequences			Parent Sequences					
	0	1	2	3	4	5	6	7	8	9
Seq		A	T	A	A	P2	P3	A	P5	P6
	0	1	2	3	4	5	6	7	8	9
S	4	1	1	4	4	0	0	0	0	0

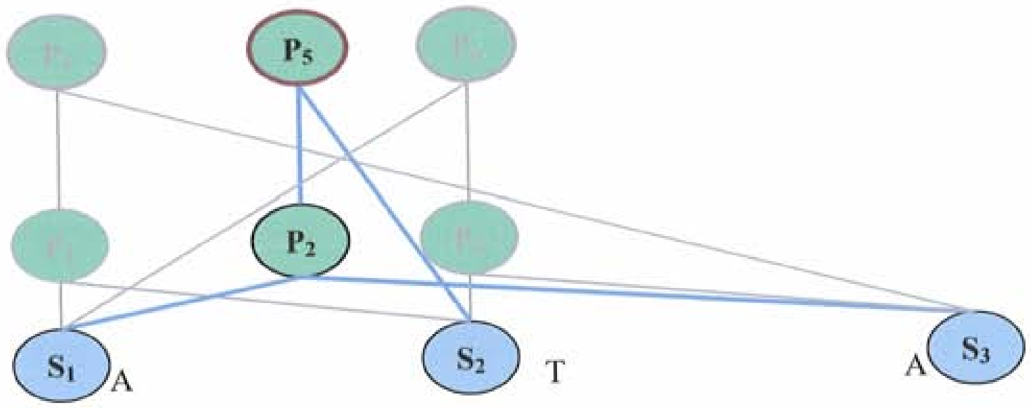
Score = 0.95

6.2.2-b: Example 2:

→ Assume the solution being evaluated is the following:

	Root	Sequences			Parent Sequences					
	0	1	2	3	4	5	6	7	8	9
S	5	2	5	2	0	5	0	0	0	0
level	2	0	0	0		1				

The corresponding decoded tree is as follows:



$$Score(P_2) = \begin{array}{l}[1 + Level(S_1)] * [ScoreOfChange(P_2, S_1) + Score(S_1)] \\ + [1 + Level(S_3)] * [ScoreOfChange(P_2, S_3) + Score(S_3)]\end{array}$$

$$Score(P_5) = \begin{array}{l}[1 + Level(P_2)] * [ScoreOfChange(P_5, P_2) + Score(P_2)] \\ + [1 + Level(S_2)] * [ScoreOfChange(P_5, S_2) + Score(S_2)]\end{array}$$

Since $S_1 = S_3 =$ "A", P_2 can only be "A", forcing P_5 to be "A" or "T", given that $S_3 =$ "T". Thus computing the score for each case will give different scores for the corresponding values of P_5.

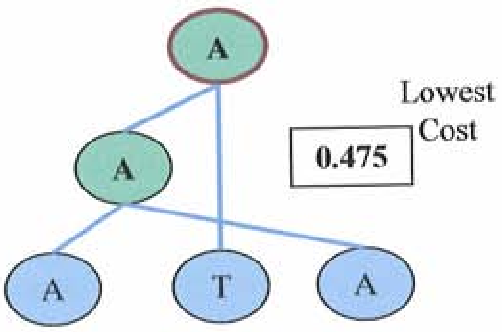
P_2	$Score(P_2)$	P_5	$Score(P_5)$
"A"	$\{[1+0]*[0+0]\}+\{[1+0]*[0+0]\}=0$	"A"	$\{[1+1]*[0+0]\}+\{[1+0]*[0.475+0]\}=0.475$
		"T"	$\{[1+1]*[0.475+0]\}+\{[1+0]*[0+0]\}=0.95$

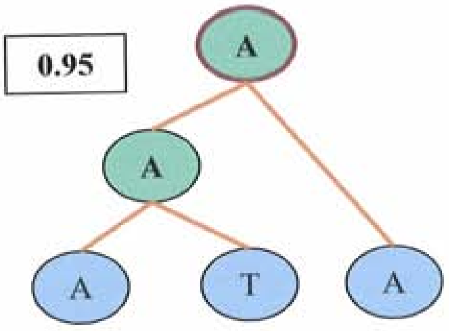
Since we ought to have the tree with values giving the lowest cost, we will have the following set of values at the end of the evaluation: $P_2 = P_5 =$ "A", with a total score of 0.475

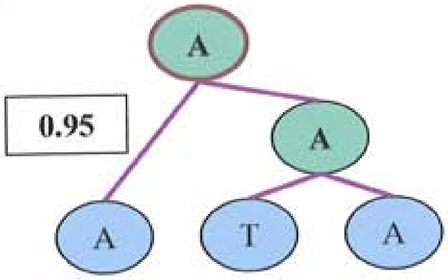
	Root	Sequences			Parent Sequences					
	0	1	2	3	4	5	6	7	8	9
Seq		A	T	A	P1	A	P3	P4	A	P1
	0	1	2	3	4	5	6	7	8	9
S	5	2	5	2	0	5	0	0	0	0

Score = 0.475

As a summary, the solutions extracted from this example affirm that the lowest cost (optimal cost) was found when combining S_1 = "A" with S_3 = "A", then their parent = "A" with S_2 = "T" into a root = "A", giving a score of 0.475 (lowest among the 3 scores 0.95, 0.475 and 0.95)







6.2.3 Propagate Changes Up

Unfortunately the importance of this function is not obvious at first sight. Indeed, it is embedded in many of the main functions used in the different heuristics, such as the CREATION, local and global PERTURBATION, RECOMBINATION functions, which will be discussed later.

In each solution, we should have at least 1 and at most $N/2$ pairs of leaf sequences. The algorithm starts by choosing one ore more pairs of leaf nodes, joining them and propagate changes up. The propagation process may be affected by another solution (usually when perturbing the latter solution). If this second solution is valid, at least one pair of the new solution does not correspond to a pair of that solution. For the solution to be completed, one or more of the leaf sequences slots are set to NULL. For internal node sequences, random combination of pairs is no longer a choice, but a requirement.

Input: A Solution S Number of Leaf Sequences N Initial Solution S_{init}
Pre-condition: $N \geq 2$ S_{init} represents a valid tree topology
Output: -
Post-condition: S is completed / modified to represent a valid tree topology
Steps:

```
{If one leaf sequence location is left NULL, fill randomly the entire solution S}
$Seq \leftarrow 1$;
$Randomly \leftarrow FALSE$;
While $Seq \leq N$ and $NOT(Randomly)$
    $Randomly \leftarrow (S[Seq] = NULL)$;
    $Seq \leftarrow Seq + 1$;
{If perturbing solution S init, check if at least one pair of leaf sequences should be changed}
$ChkChangedPairs \leftarrow FALSE$;
If $S_{init} \neq NULL$ then
    $Seq \leftarrow 1$;
    While $Seq \leq N$ and $NOT(ChkChangedPairs)$
        $ChkChangedPairs \leftarrow (S_{init}[Seq] \neq NULL)$;
        $Seq \leftarrow Seq + 1$;
{If solution S is not randomly filled, extract all pairs of leaf sequences already bounded}
$Pairs[\ ] \leftarrow NULL$;
$NumPairs \leftarrow 0$
If $NOT(Randomly)$ then
    $Pairs[\ ] \leftarrow getLeafSeqPairs(S)$;
    $NumPairs \leftarrow \dim(Pairs)$;
```

```
{The number of leaf pairs "NumPairs" is at most half that of the leaf sequences "N"}
{If solution S is randomly filled, OR has leaf sequence pairs more than needed}
{    Handle a random number of pairs at each tree level}
$RandomPairs \leftarrow Randomly \quad OR \quad (NumPairs < 1) \quad OR \quad \left( NumPairs > \frac{N}{2} \right);$
{Initialize set of sequences, to be combined, as set of indices of all leaf sequences}
{Start from leaves and loop until entire tree is built and validated}
$Leaves \leftarrow TRUE;$
$ListSeq[\ ] \leftarrow createIndexList(N);$
Loop
    $NumSeq \leftarrow \dim(ListSeq);$
    If $NumSeq < 2$ then
        Exit Loop
    {Set Number of pairs at current tree level "C"}
    {Either to "NumPairs" OR randomly between 1 and "NumSeq"/2}
    $C \leftarrow NumPairs;$
    If $RandomPairs$ then
        $C \leftarrow random(1 \quad , \quad NumSeq/2);$
    {Create a list of indices, from 1 to NumSeq, to enforce uniqueness of each pair seq}
    $L \leftarrow createIndexList(NumSeq);$
    {If dealing with parent (not leaf) sequences, pairs are combined randomly}
    $RandomChoice \leftarrow RandomPairs \quad OR \quad NOT(Leaves);$
    {Initialize current set of parents to be of size C}
    $ListParents[\ ] \leftarrow createList(C);$
    For $i$ from 1 to $C$
        If $RandomChoice$ then
            {Select first seq "Seq1" in current pair, randomly AND uniquely}
            $IndexSeq_1 \leftarrow randomUniqueIndex(L);$
            If $IndexSeq_1 < 1$ then
                Exit Loop
            $Seq_1 \leftarrow ListSeq[IndexSeq_1];$
            If $NOT(Leaves)$ then
                $Seq_1 \leftarrow N + Seq_1;$
        Else {"RandomChoice" is FALSE}
            {Select first seq "Seq1" in current pair, as first seq of a previous leaf pair}
            $Seq_1 \leftarrow Pairs[i,1];$
            $IndexSeq_1 \leftarrow indexOf(L, Seq_1);$                    $remove(L, Seq_1);$
```

```
{Get index of seq combined with $Seq_1$, in either solution $S_{init}$ or solution S}
$Seq_{pair} \leftarrow getPairIndex(S, Seq_1)$;
If $S_{init} \neq NULL$ then
    $Seq_{pair} \leftarrow getPairIndex(S_{init}, Seq_1)$;
$CheckPair \leftarrow (Seq_{pair} \geq 1)$  and  $(Seq_{pair} \leq NumSeq)$;
$Index \leftarrow indexOf(ListSeq, Seq_{pair})$;
{If perturbing $S_{init}$ and at least one pair should be changed, AND selected seq}
{Already combined, enforce that its pair seq is not selected as second seq}
If ChkChangedPairs and CheckPair then
    $remove(L, index)$;
If RandomChoice then
    {Select second seq "$Seq_2$" in current pair, randomly AND uniquely}
    $IndexSeq_2 \leftarrow randomUniqueIndex(L)$;
    If $IndexSeq_2 < 1$ then
        Exit Loop
    $Seq_2 \leftarrow ListSeq[IndexSeq_2]$;
    If $NOT(Leaves)$ then
        $Seq_2 \leftarrow N + Seq_2$;
Else {" RandomChoice " is FALSE}
    {Select second seq "$Seq_2$" in current pair, as that of a previous leaf pair}
    $Seq_2 \leftarrow Pairs[i,2]$;
    $IndexSeq_2 \leftarrow indexOf(L, Seq_2)$;
    $remove(L, Seq_2)$;
{If perturbing $S_{init}$ and at least one pair should be changed, AND selected seq}
{Already combined, enforce that its pair seq is could be selected in next pairs}
If ChkChangedPairs and CheckPair then
    $add(L, Index)$;
{Get parent of sequences $Seq_1$ and $Seq_2$ and set it in solution S and in ListParents}
$Seq_{Par} \leftarrow getParent(Seq_1, Seq_2)$;
If $Seq_{Par} < 1$ then
    Exit For
$ListParents[i] \leftarrow Seq_{par}$;
$S[Seq_1] \leftarrow Seq_{par}$;
$S[Seq_2] \leftarrow Seq_{par}$;
```

```
        {After all C pairs have been formed, combine remaining sequences in ListSeq}
        {One-by-one to a parent of these pairs, chosen randomly AND uniquely}
        $P \leftarrow createIndexList(C)$;
        While $\dim(L) > 0$
            $IndexSeq_1 \leftarrow randomUniqueIndex(L)$;
            If $IndexSeq_1 < 1$ then
                Exit Loop
            $Seq_1 \leftarrow ListSeq[IndexSeq_1]$;
            If $NOT(Leaves)$ then
                $Seq_1 \leftarrow N + Seq_1$;
            $IndexSeq_2 \leftarrow randomUniqueIndex(P)$;
            If $IndexSeq_2 < 1$ then
                $Seq_2 \leftarrow Seq_{Par}$;
            Else
                $Seq_2 \leftarrow N + ListPar[IndexSeq_2]$;
                $removeAt(ListPar, IndexSeq_2)$;
            $Seq_{par} \leftarrow getParent(Seq_1, Seq_2)$;
            If $Seq_{Par} < 1$ then
                Exit While
            $S[Seq_1] \leftarrow Seq_{par}$;
            $S[Seq_2] \leftarrow Seq_{par}$;
        {Once all sequences in ListSeq have been combined in a pair}
        {Swap sets to pass to next tree level}
        If $IndexSeq_2 < 1$ then
            Exit Loop
        $ListSeq[\ ] \leftarrow ListParents[\ ]$;
        $Leaves \leftarrow FALSE$;
    Until $TRUE$
    {The final step is to discover and set the root of the tree}
    {In addition, make the values in solution S consistent in their format}
    $S[ROOT\_INDEX] \leftarrow findRoot(S)$;
    $S \leftarrow makeValuesConsistent(S)$;
    $evaluate(S)$;
    Return $S$
```

6.2.4 Create Random Solution

Those heuristics, which start from a random initial state or solution, generate it by first allocating enough space in memory. Then it combines randomly pairs of sequences, starting from the leaf sequences, and propagating up the combinations to reach the root node of the phylogenetic tree decoded from the data-structure.

```
Input:          A Number of Leaf Sequences $N$
Pre-condition:  $N \geq 2$
Output:         A Solution $S$
Post-condition: $S$ represents a valid random tree topology
Steps:
    $L \leftarrow 1 + N + numberOfParentNodes(N)$;
    $S \leftarrow createSolution(L)$;
    $propagateUp(S, N, NULL)$;
    $evaluate(S)$;
```

6.2.5 Perturb Solution

Some heuristics may need to perturb a given solution globally, by modifying randomly one or more pairs. Others may perturb the solution locally, by selecting a given leaf sequence and re-coupling it with another sequence chosen randomly. Broken pairs are re-coupled either together or with the parent sequence of the newly formed pair.

The global perturbation algorithm starts by creating an "empty" solution (with NULL value), and then propagate the change up by randomly combining pairs of leaf and parent sequences.

```
Input:          A Number of Leaf Sequences $N$        Given Solution $S_{given}$
Pre-condition:  $N \geq 2$                            $S_{given}$ represents a valid tree topology
Output:         A Solution $S$
Post-condition: $S$ represents a valid random tree topology, but based on $S_{given}$
Steps:
    $L \leftarrow \dim(S_{given})$;
    $S \leftarrow createSolution(L)$;
    $propagateUp(S, N, S_{given})$;
    $evaluate(S)$;
    Return $S$
```

The local perturbation algorithm is a bit more complex, as it only perturbs the given solution at a certain index, by breaking-up pairs and gluing others if needed.

```
Input:      A Number of Leaf Sequences $N$          Given Solution $S_{given}$
            Sequence Index $Index$
Pre-condition:    $N \geq 2$          $S_{given}$ represent a valid topology
Output:     A Solution $S$
Post-condition:   $S$ randomly represent a valid tree topology, but based on $S_{given}$
Steps:
    $L \leftarrow \dim(S_{given})$;
    {If sequence index is valid then set it as first seq "Seq1", otherwise select "Seq1" randomly}
    $Seq_1 \leftarrow Index$;
    If $Seq_1 < 1$ and $Seq_1 > N$ then
        $Seq_1 \leftarrow random(1, N)$;
    $S \leftarrow copy(S_{given})$;                {Copy solution $S_{given}$ to solution S}
    $List \leftarrow createIndexList(N)$;  {Create a list of indices, from 1 to N, to enforce uniqueness}
    $remove(List, Seq_1)$;               {Remove "Seq1" from the list not to be re-selected}
    {Retrieve index of pair sequence of "Seq1" and remove it from list if leaf sequence}
    $SeqPair_1 \leftarrow getPairIndex(S, Seq_1)$;
    $IsSeq_1 \leftarrow (SeqPair_1 \geq 1)$  AND  $(SeqPair_1 \leq N)$;
    If $IsSeq_1$ then
        $remove(List, SeqPair_1)$;
    {Select second seq "Seq2" randomly AND uniquely}
    $Seq_2 \leftarrow randomUniqueIndex(List)$;
    {Get parent of "Seq1" and "Seq2"}
    $SeqPar \leftarrow getParent(Seq_1, Seq_2)$;
    If $isParentSeq(SeqPar)$ then
        {Retrieve index of pair sequence of "Seq2" and test if leaf sequence}
        $SeqPair_2 \leftarrow getPairIndex(S, Seq_2)$;
        $IsSeq_2 \leftarrow (SeqPair_2 \geq 1)$  AND  $(SeqPair_2 \leq N)$;
        {Set parent of "Seq1" and "Seq2" in solution S}
        $S[Seq_1] = SeqPar$;
        $S[Seq_2] = SeqPar$;
        {Glue broken pairs caused by the combination of "Seq1" with "Seq2"}
        If $IsSeq_1$ or $IsSeq_2$ then
            {If pair of "Seq1" was broken, consider the other index of that pair as "Seq3"}
            {Otherwise, consider the parent of "Seq1" and "Seq2" as "Seq3"}
            $Seq_3 \leftarrow SeqPar$;
            If $IsSeq_1$ then
                $Seq_3 \leftarrow SeqPair_1$;
            {If pair of "Seq2" was broken, consider the other index of that pair as "Seq3"}
            {Otherwise, consider the parent of "Seq1" and "Seq2" as "Seq3"}
            $Seq_4 \leftarrow SeqPar$;
            If $IsSeq_2$ then
                $Seq_4 \leftarrow SeqPair_2$;
            {Get parent of "Seq3" and "Seq4" and combine them if valid parent sequence}
            $SeqPar_2 \leftarrow getParent(Seq_3, Seq_4)$;
            If $isParentSeq(SeqPar_2)$ then
                $S[Seq_3] = SeqPar_2$;
                $S[Seq_4] = SeqPar_2$;
    $propagateUp(S, N, S_{given})$;
    $evaluate(S)$;
    Return $S$
```

6.2.6 Recombine Parent Solutions

Since one focus of this paper is the flexibility of the algorithms, solutions have to be recombined in a general way. The general recombination schema of heuristics implies that a given set of solutions are recombined to form a set of offspring solutions, by applying a given number of cuts in the parent solutions. The length of the children set must be greater or equal to that of the parents set (at least one child for each parent). There must be at least two parents involved in the recombination process, generating at least two offspring.

For each parent instance, and each pair of sequences, if both sequences involved are within the same cut, they are copied to the corresponding child; otherwise, they are recombined according to their common parent sequence. The choice of the child is determined by both the index of the parent in the set, and the number of cut the pair sequences are in.

<u>Input</u>: A set of Solutions "$Parents[\ ]$" Number of Leaf Sequences "N"
Number of Cuts "$Cuts$" Parent-Children Ratio "$Ratio$"

<u>Pre-condition</u>: $\dim(Parents) \geq 2$ $1 \leq Cuts < N$ $Ratio \geq 1$

<u>Output</u>: A set of Solutions "$Offspring[\ ]$"

<u>Post-condition</u>: S randomly represent a valid tree topology, but based on S_{given}

<u>Steps</u>:

```
$L \leftarrow \dim(Parents[1])$;               {Length of each solution}
$P \leftarrow \dim(Parents)$;                  {Number of solutions to recombine (parents)}
$O \leftarrow Ratio * P$;                      {Number of offspring to generate from given parents}
$M \leftarrow N - 1$;                          {Maximum number of cuts in parents = Number leaf seqs -1}
$C \leftarrow \max(1, \min(M, Cuts))$;         {Effective number of cuts possible}
$Offspring[\ ] \leftarrow createList(O)$;      {Initialize set of O offspring to be empty}
For $i$ from 1 to $O$
    $imp \leftarrow \mathrm{mod}(i-1, p)$;     {index of parent to start current offspring with}
    {If all parents have been recombined, set "numCuts" randomly between 1 and M}
    $numCuts \leftarrow C$;
    If $i > P$ then
        $numCuts \leftarrow random(1, M)$;
    $Slice \leftarrow \left\lceil \frac{N}{NumCuts + 1} \right\rceil$;
    For $j$ from 1 to $N$
        {Retrieve index "k" of sequence "j" in "Parents [imp]"}
        {If "k" is below "j" or not a leaf sequence, set it to "j"}
        $k \leftarrow getPairIndex(Parents[imp], j)$;
        If $k < j$ or $k > N$ then
            $k \leftarrow j$;
        {Cut X includes all sequences in interval: $]X * Slice, \quad (X+1) * Slice]$}
        {First sequence is included in first interval}
        $jj \leftarrow \lfloor (j-0.5)/Slice \rfloor$;          $kk \leftarrow \lfloor (k-0.5)/Slice \rfloor$;
        {Check if sequence "j" and "k" fall within the same interval "jj"}
        **If** $jj = kk$ **then**
            {Copy the interval values from "Parents [imp+jj % P]"}
            {Recall: "imp" is the index of the parent to start the current offspring with}
            $parIndex \leftarrow \mathrm{mod}(imp + jj, P)$;
            $Offspring[i, j] \leftarrow Parents[parIndex, j]$;
            $Offspring[i, k] \leftarrow Parents[parIndex, k]$;
        **Else**
            {Get parent of both sequences "j" and "k" and set it in current offspring}
            $SeqPar \leftarrow getParent(j,k)$;
            $Offspring[i, j] \leftarrow SeqPar$;
            $Offspring[i, k] \leftarrow SeqPar$;
    {After setting the leaf sequences slots in the current offspring, validate and evaluate it}
    $propagateUp(Offspring[i], N, NULL)$;
    $evaluate(Offspring[i])$;
**Return** $Offspring$
```

The next chapter discusses, for each algorithm, the suggested generic implementations for each algorithm (GA, SA, SimE, StocE, TS) in more details, showing the implementation of the flexibility and detailed tracing/chronology.

Chapter 7 Generic Heuristic Implementations

7.1 Genetic Algorithms

Since Genetic Algorithms deal, in concept, with maximization, and since the suggested evaluation approach retrieves its values from a scoring (cost) matrix, the fitness function should be the negative of the evaluation function.

7.1.1 Main Procedure

The steps of the algorithm implemented are as follows:

```
Initialize the population randomly.
Initialize to zero the number of generations to deal with
Select and backup the fittest individual (group of chromosomes).
Record in history the initial population & initialization time
While NOT    Terminate Process
    Create a new empty population.
    Do
        Select two or more individuals to apply operators on.
        Apply crossover to get the offspring, given a certain probability
        Apply mutation to each individual of the offspring, given a certain probability
        Save the newly generated individuals in the new population
    While (new population is not full)
    Get fittest individual of the new population.
    Compare it to the previous fittest individual.
    If (its fitness value is greater) Then
        Update fittest chromosome variable.
    Set the new population as the current population
    Increment number of generations
    Record in history: new population & time needed to generate this new population
End While
Return the fittest individual found so far
```

Even though crossover and mutation operators are applied given certain probabilities, it is very rare to find a parent individual (group of chromosomes) selected and passed to the new population without modification. The initial population is filled with a fixed number of individuals, whose chromosomes and genes are randomly generated. The fitness function of an individual is proportional evaluation of each of the chromosomes that individual includes. The algorithm supports elitism, to keep track of the best-fit individual found in previously handled and in the current population. It keeps on iterating and handling populations until the process termination criterion. In most problems, the termination criterion is satisfied when the algorithm reaches a certain number of generations.

For our Parsimony Phylogenetic Trees problem encoding, recombining (crossover for 2 or more instances) a set of individuals, also called parents, is done by applying the RECOMBINATION function. [Section 6.2.6 – p 48] As for the mutation operator, the algorithm will need to involve the general PERTURB function(s) [Section 6.2.5 – p 45] with the individual in concern.

7.1.2 Object-Oriented Design

Figure 13 shows an overview of the interaction between the different classes instantiated during the runtime of the algorithm, discussed below. An allele can be found in at least one gene, which in turn may be part of one or more chromosomes. An individual can be described as a non-empty pool of chromosomes, which may or may not be related to each other. A population includes a fixed non-negative number of chromosomes, called the population size S.

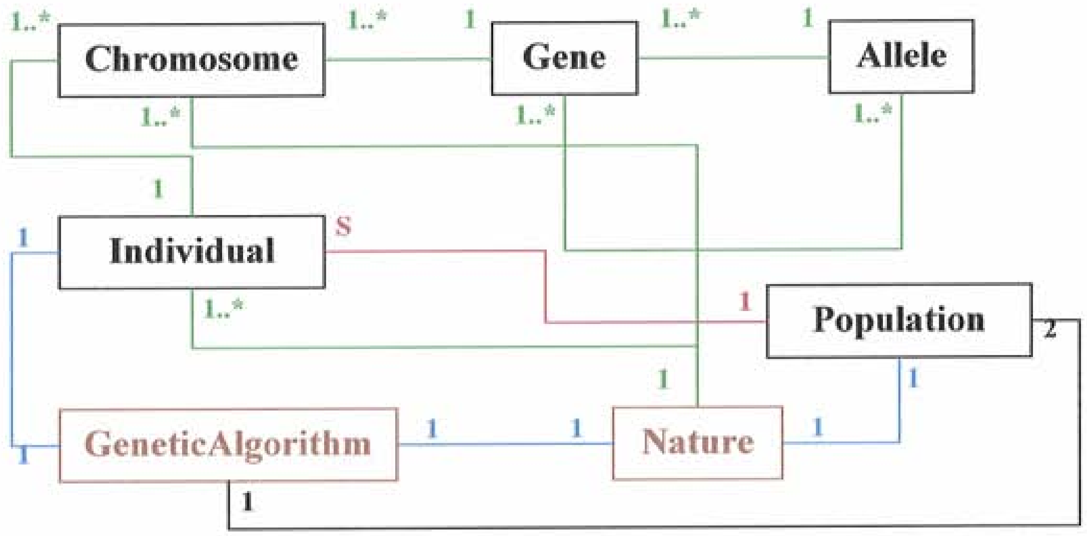


Figure 13: Brief Overview of GA Classes Interaction

As shown in Figure 13, the suggested flexible Genetic Algorithm tries to mimic as close as possible the genome as known in Biology. In other terms, each instance of an NP-problem, encoded to be used in this algorithm as an Individual, would be composed of one or more chromosomes. Each chromosome is constituted of a set of genes, which is defined as a combination of different alleles. In our parsimony phylogenetic problem, when dealing with DNA, we are consequently encoding each of the four {A, T, C, G} nucleotides into an allele. Therefore, a gene will encode a fixed group of DNA nucleotides.

This design makes the algorithm flexible, in a way that it allows changing the encoding schema by just modifying the design and implementation of alleles and genes according to the problem being handled. For instance, when dealing with binary encoding, an allele would have a value of "0" or "1". For decimal encoding an allele value could be one of {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}. Moreover, a non-standard decimal encoding could give, for example, allele values between "00" and "79".

Since the encoding can consider different alternatives of values for alleles, some methods that are problem-specific methods have to be implemented, among them: creating random genes, creating random chromosomes/individuals/ populations and their fitness functions), and handled by a sub-class of class "Nature". The algorithm-dependent procedures (selection, crossover, mutation, termination) are handled by a sub-class of the "GeneticAlgorithm" class.

7.2 Simulated Annealing

The main idea behind Simulated Annealing was to simulate the annealing of metals, where the metal is mapped into an initial solution, and the atom moves mapped into moves to neighbor solutions, each of which is accepted, with a certain probability, if the cost of the neighbor solution is better than the current and best one found so far. The probability of accepting a solution is less or equal to a threshold value, directly proportional to the current temperature, and inversely to the difference in cost between the neighbor and current solutions. As the temperature cools down according to a given schedule, the algorithm is getting us closer to the optimal solution. Therefore, it allows the current solution to perturb more often. This process is repeatedly done for a given maximum number of moves, by applying the cooling rate α to the temperature, and motion rate β.

7.2.1 Standard Simulated Annealing

<u>*7.2.1-a: Main Procedure*</u>

```
Inputs:   Initial Solution $S_0$        Initial Temperature $T_0$        Cooling Rate $\alpha$
          Motion Rate $\beta$           Initial number of moves $M_0$
          Maximum Annealing time $MaxTime$
Precondition:        $\alpha < 1$ and $\beta > 1$
Outputs:  Optimal Solution $BestS$
Algorithm:
    {Initialization of CurS, BestS, T, M and Time}
    $CurS \leftarrow S_0$;        $T \leftarrow T_0$;        $M \leftarrow M_0$;
    $BestS \leftarrow CurS$;
    $CurCost \leftarrow Cost(CurS)$;
    $BestCost \leftarrow Cost(BestS)$;
    $Time \leftarrow 0$;
    Repeat
        $updateHistory(CurS, BestS, T, M, Time)$;
        {Update CurS and BestS, by moving M times, at temperature T}
        $Metropolis(CurS, CurCost, BestS, BestCost, T, M)$;
        $updateHistory(MetropolisHistory)$;
        {Time = total number of moves at the end of each iteration}
        $Time \leftarrow Time + M$;
        {Cooling down the temperature by 100*$\alpha$ %}
        $T \leftarrow \alpha * T$;
        {Increasing the number of moves by 100*$(\beta - 1)$ %}
        $M \leftarrow \beta * M$;
    Until $Time \geq MaxTime$
    Return $BestS$
```

7.2.1-b: Metropolis Procedure

```
Inputs:   Current Solution $CurS$                Current Cost $CurCost$
          Best Solution so far $BestS$           Best Cost so far $BestCost$
          Current Temperature $T$                Current Number of Moves $M$
Outputs:  #N/A
Algorithm:
    $Moves \leftarrow M$ ;
    Repeat
        {Get a neighbor solution into NewS}
        $NewS \leftarrow neighbor(CurS)$;
        $NewCost \leftarrow \cos t(NewS)$;
        $UpdateIterationHistory(CurS, NewS, BestS)$
        {Check out the cost gain/lost}
        $DeltaCost \leftarrow NewCost - CurCost$;
        If $DeltaCost < 0$ then
            {NewS has a cost lower than that of CurS}
            $CurS \leftarrow NewS$ ;            $CurCost \leftarrow NewCost$;
            If $NewCost < BestCost$ then
                {NewS has a cost even lower than that of BestS}
                $BestS \leftarrow NewS$ ;            $BestCost \leftarrow NewCost$;
        Else
            {NewS has a cost higher than CurCost; however, accept it with a probability}
            {The probability decreases with the decrease of T and increase of DeltaCost}
            If $random(\ ) < e^{\frac{-DeltaCost}{T}}$ then
                $CurS \leftarrow NewS$ ;            $CurCost \leftarrow NewCost$;
        $Moves \leftarrow Moves - 1$;
    Until $Moves = 0$
```

Per simulation iteration, the Metropolis procedure computes M neighbor solutions. Like the "Cost" method, the "Neighbor" function is problem specific. Every problem that needs to run Simulated Annealing needs to implement both "Cost" and "Neighbor" functions.

Below is an algorithm that helps tuning the initial temperature parameter of the Simulated Annealing, according to the standard and simple cooling schedule.

7.2.2 Tuning Initial Temperature Procedure

Inputs: Current Solution $CurS$ — Current Cost $CurCost$
Starting Temperature T — Number of Moves Attempted M
Result tolerance $Threshold$

Outputs: Initial Temperature T_0

Algorithm:

```
$T_0 \leftarrow T$;
$Ratio \leftarrow (1 - Threshold)$;
**While** $Ratio \geq (1 - Threshold)$ **do**
    $Attempted \leftarrow M$;
    $Accepted \leftarrow 0$;
    $Moves \leftarrow M$;
    **Repeat**
        $NewS \leftarrow neighbor(CurS)$;
        $NewCost \leftarrow \mathrm{cost}(NewS)$;
        $DeltaCost \leftarrow NewCost - CurCost$;
        **If** $DeltaCost < 0$ **then**
            $CurS \leftarrow NewS$;
            $CurCost \leftarrow NewCost$;
            $Accepted \leftarrow Accepted + 1$;
        **Else**
            **If** $random(\ ) < e^{\frac{-DeltaCost}{T}}$ **then**
                $CurS \leftarrow NewS$;
                $CurCost \leftarrow NewCost$;
                $Accepted \leftarrow Accepted + 1$;
        $Moves \leftarrow Moves - 1$;
    **Until** $Moves = 0$
    $Ratio \leftarrow \frac{Accepted}{Attempted}$;
**Return** $T_0$
```

7.2.3 Customized Simulated Annealing

Various cooling schedules, mentioned later, can be used with a Simulated Annealing optimization. Let T_i be the temperature for iteration *i*, where *i* increases from 1 to N. The number of iterations is indirectly determined by the user through: *MaxTime*

This customized SA algorithm computes the number of cooling down iterations, before hand, and tries to make each one as independent of the others as possible. Thus, the temperature, number of moves and time should be as unrelated as possible to the respective values computed in previous iterations.

Let t_i be the time elapsed up to iteration *i*, and M_i the number of moves for this iteration.

$$t_i = \begin{cases} t_{i-1} + M_{i-1} & \text{if} \quad 1 < i \le N \\ M_0 & \text{if} \quad i = 1 \end{cases} \qquad \text{Where: } M_i = \beta * \begin{cases} M_{i-1} & \text{if} \quad 1 < i \le N \\ M_0 & \text{if} \quad i = 1 \end{cases}$$

$$M_i = \beta^i M_0 \quad for \quad i = 1,2,\ldots,N$$

$$t_1 = M_0$$
$$t_2 = t_1 + M_1 \Rightarrow t_2 = M_0 + M_1 \quad \Rightarrow \quad t_2 = (1+\beta) * M_0$$
$$t_3 = t_2 + M_2 \Rightarrow t_3 = (1+\beta) * M_0 + M_2 \quad \Rightarrow \quad t_3 = (1+\beta+\beta^2) * M_0$$
$$t_4 = t_3 + M_3 \Rightarrow t_4 = (1+\beta+\beta^2) * M_0 + M_3 \quad \Rightarrow \quad t_4 = (1+\beta+\beta^2+\beta^3) * M_0$$

Similarly:

$$t_k = M_0 * \sum_{j=0}^{k-1} \beta^j \quad \Rightarrow \quad t_k = \left(\frac{\beta^k - 1}{\beta - 1}\right) * M_0 \quad for \quad k = 1,2,\ldots,N$$

At the end of the simulation – $t_N = MaxTime$

$$MaxTime = \left(\frac{\beta^N - 1}{\beta - 1}\right) * M_0 \Rightarrow \left(\beta^N - 1\right) = \frac{(\beta - 1) * MaxTime}{M_0}$$

$$\text{Then: } \beta^N = 1 + \frac{(\beta - 1) * MaxTime}{M_0} \Rightarrow N = Integer\left(\log_\beta\left\{1 + \frac{(\beta - 1) * MaxTime}{M_0}\right\}\right)$$

Therefore:

$$N = IntegerPart\left[\frac{1}{\ln(\beta)} * \ln\left(1 + \frac{MaxTime * (\beta - 1)}{M_0}\right)\right]$$

The different schedules, implemented in the cooling schedule described below, are illustrated in Table 14. These schedules are used in the customized algorithm by referring to the corresponding code, which is an integer value between 1 and 9. For flexibility reasons, code 10 is left to allow the algorithm to handle any externally defined schedule.

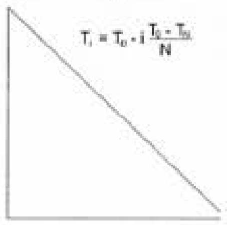
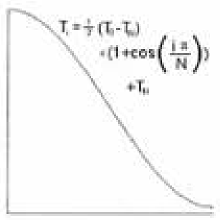
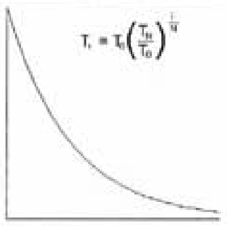
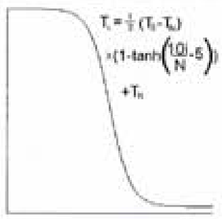
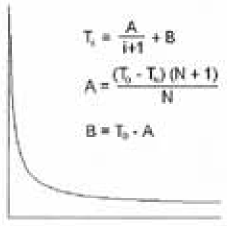
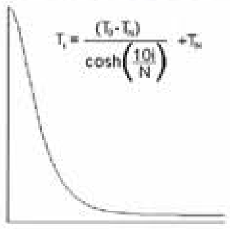
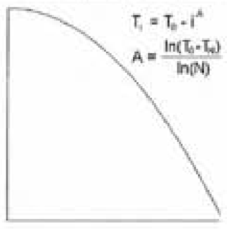
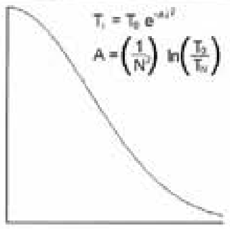
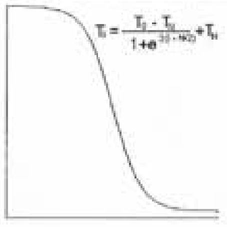
Code	***Name***	***Graph***			
1	*Linear*		6	*Cos*	
2	*Scalar*		7	*Tanh*	
3	*Hyperbolic*		8	*Cosh*	
4	*Exponential*		9	*Squared Scalar*	
5	*Sigmoid*				

Table 14: Some Cooling Schedule for Simulated Annealing

7.2.3-a: Main Procedure

Inputs: Initial Solution S_0; Cooling Schedule $code$; Initial Temperature T_0; Total cooling Rate R; Motion Rate β; Initial number of moves M_0; Maximum Annealing time $MaxTime$

Precondition: $\beta > 1$

Schedule =	*Linear*	*Scalar*	*Hyperbolic*	*Exponential*	*Sigmoid*
code =	*1*	*2*	*3*	*4*	*5*
	6	*7*	*8*	*9*	*10*
Schedule =	*Cos*	*Tanh*	*Cosh*	*SquaredScalar*	*Customized*

Outputs: Optimal Solution $BestS$

Algorithm:

```
    $CurS \leftarrow S_0$;                $CurCost \leftarrow Cost(CurS)$;
    $BestS \leftarrow CurS$;              $BestCost \leftarrow CurCost$;
    $N \leftarrow IntegerPart\left[\frac{1}{\ln(\beta)} * \ln\left(1 + \frac{MaxTime * (\beta - 1)}{M_0}\right)\right]$;
    $T \leftarrow T_0$;
    $M \leftarrow M_0$;
    For $i$ from 1 to $N$ do
        $updateHistory(CurS, BestS, T, M, Time)$;
        $Metropolis(CurS, CurCost, BestS, BestCost, T, M)$;
        $updateHistory(MetropolisHistory)$;
        $T \leftarrow CoolingSchedule(code, i, N, R, T_0)$;
        $M \leftarrow \beta^i * M_0$;
    Return $BestS$
```

In comparison with the standard S.A algorithm, this algorithm allows the simulation to occur with a non-scalar cooling schedule, by specifying the code of the cooling schedule. One difference is the absence of the time variable replaced by a loop-iteration index. Another difference is the fact that the temperature and the number of moves are computed independently of previous iterations, but according to the iteration index and some other parameters. Since the global initialization phase was replaced by an iteration initialization, we have $i-1$ instead of i. This algorithm uses the Metropolis procedure defined above by the Standard Simulated Annealing procedure.

7.2.3-b: CoolingSchedule Procedure

Inputs: Cooling Schedule *code*; Current iteration index i; Number of cooling steps N; Total cooling Rate R; Initial Temperature T_0

Precondition:

Schedule =	*Linear*	*Scalar*	*Hyperbolic*	*Exponential*	*Sigmoid*
code =	*1*	2	3	4	5
code =	6	7	8	9	10
Schedule =	*Cos*	*Tanh*	*Cosh*	*SquaredScalar*	*Customized*

Outputs: Current Temperature T

Algorithm:

```
Case code is
    1  Then   Return $T_0 - \left[\frac{i*(1-R)}{N}\right]$
    2  Then   $A \leftarrow \frac{i}{N}$;                    Return $T_0 * R^A$
    3  Then   Return $T_0 * \left[\frac{(1-R)(N+1)}{N(i+1)} + \frac{R(N+1)-1}{N}\right]$
    4  Then   $A \leftarrow \frac{\ln(T_0 * (1-R))}{\ln(N)}$;    Return $T_0 - i^A$
    5  Then   $A \leftarrow 0.3\left(i - \frac{N}{2}\right)$;      Return $T_0 * \left[\frac{(1-R)}{1+e^A} + R\right]$
    6  Then   Return $\left(\frac{T_0}{2}\right) * \left[1 + R + \left\{(1-R) * Cos\left(\frac{\pi * i}{N}\right)\right\}\right]$
    7  Then   Return $\left(\frac{T_0}{2}\right) * \left[1 + R - \left\{(1-R) * Tanh\left(\frac{10i}{N} - 5\right)\right\}\right]$
    8  Then   Return $T_0 * \left[(1-R) * Sech\left(\frac{10i}{N}\right)\right]$
    9  Then   $A \leftarrow \left(\frac{i}{N}\right)^2$;                Return $T_0 * R^A$
    10 Then   $T \leftarrow customizedSchedule(i, N, R, T_0)$;
              If $T < T_0$ then                Return $T$
              Return $coolingSchedule(2, i, N, R, T_0)$
```

Since the Simulated Annealing algorithm deals, in concept, with minimization, the COST of a solution should be the result of the evaluation function, mentioned in a previous chapter. The result of the NEIGHBOR function for a given solution is, in other words, that of the perturbation for this solution.

7.2.4 Object Oriented Design

Figure 15 shows an overview of the interaction between the different classes instantiated during the runtime of the algorithm, discussed below. The algorithm creates only one neighbor for the current solution per iteration from its current solution, where some of them may be compared to it and some to the best solution found so far. Therefore, the algorithm handles only 3 solutions per iteration.

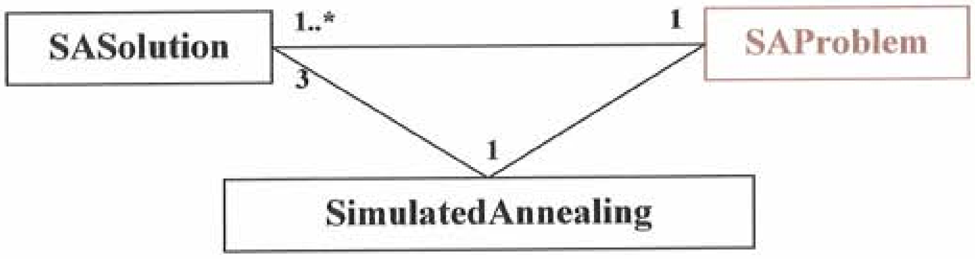


Figure 15: Brief Overview of SA Classes Interaction

The problem-specific methods (create solution, neighbor(s), cost) are handled by the class "SAProblem". The class "SimulatedAnnealing" handles the algorithmic-dependent procedures; for instance, getting the temperature at the next iteration, for the given cooling schedule.

7.3 Simulated Evolution

The Simulated Evolution algorithm was first designed to simulate the evolution of components of a process in real-life. Each element is associated a value, called goodness, that measures the distance between its current location and its optimal one, in the entire evolution process. Given a certain probability, some of these components are considered as eligible to be added to the more evolved state. This probability is controlled by the goodness of the component along with a pre-defined bias parameter B < 1, usually set to be within the range [–0.2, 0.2]. The last step of this evolution iteration is to re-allocate these selected components to their positions in the set (also called state). When the whole set has been validated, it is compared to the optimal one found so far. Therefore, given a finite set M of distinct moveable elements, and a finite set L of locations, a state is defined as an assignment function S: M → L satisfying certain constraints mentioned above.

7.3.1 Simulated Evolution Procedure

Inputs: Initial State: S_0 Probability Bias: B

Acceptance Value for Optimal Value: *Threshold*

Precondition: $|B| < 1$

Outputs: Optimal State: *BestS*

Algorithm:

```
$L \leftarrow \dim(S_0)$;
$CurS \leftarrow S_0$;          $CurGoodness \leftarrow stateGoodness(CurS)$;
$BestS \leftarrow CurS$;        $BestGoodness \leftarrow CurGoodness$;
$G[\ ] \leftarrow createArray(L)$;
Repeat
    $updateHistory(CurS)$;
    {Evaluation}
    For $i$ from 1 to $L$ do
        $G[i] \leftarrow goodness(CurS, i)$;
    {Selection}
    $P_S \leftarrow createDynamicList(\ )$;
    For $i$ from 1 to $L$ do
        If $Random(\ ) \le (1 - G[i] - B)$ Then
            $Add(P_S, i)$;
    $sortAscending(P_S, G)$;
    {Allocation}
    For $i$ from 1 to $\dim(P_S)$ do
        $CurS = Allocation(CurS, P_S[i])$;
    $CurGoodness \leftarrow stateGoodness(CurS)$;
```

```
        If $CurGoodness > BestGoodness$ Then
            $BestS \leftarrow CurS$ ;
            $BestGoodness \leftarrow CurGoodness$ ;
        $finishUpdateHistory(BestS, P_S)$;
    While $NOT\{stopSimulation(BestS, CurS, Threshold)\}$
```

Since the Simulated Evolution algorithm deals, in concept, with maximization, the GOODNESS of a solution should multiply by (–1) the result of the evaluation function, mentioned in a previous chapter. The result of the ALLOCATION function applied on an element in a given state is, in other words, that of the perturbation of this state.

7.3.2 Object Oriented Design

Figure 16 shows an overview of the interaction between the different classes instantiated during the runtime of the algorithm, discussed below. The algorithm creates only one neighbor for the current solution per iteration from its current solution. This neighbor solution is compared to the current solution and potentially to the best solution found so far. Therefore, the algorithm handles only 3 solutions per iteration: a current and a neighbor solution, and the best one so far.

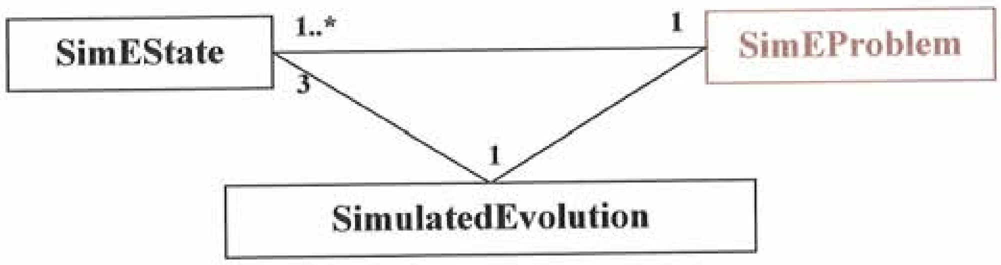


Figure 16: Overview of SimE Objects Interaction

The problem-specific methods are handled by the class "SimEProblem". The algorithmic-dependent procedures are handled by the class "SimulatedEvolution".

7.4 Stochastic Evolution

The essence of the Stochastic Evolution algorithm lies in stochastically (randomly) considering the move of items from the current state to form a new state given a certain control parameter. Let us define the gain in costs as the difference of the costs between the current and new state. The stochastic probability, affecting the selection process, is bounded by the varying value of a control parameter and compared with the gain, which is supposed to be negative. In the last part of any iteration, the algorithm expands the control parameter, if there is no change in cost from previous iteration, otherwise tightens it back to its initial value. Then, before going into the next iteration, if the current state is better than the best one found so far, the algorithm rewards itself with a given number of loop iterations R.

7.4.1 Stochastic Evolution Procedure

```
Inputs:    Initial State $S_0$                  Control Parameter $p_0$
           Number of iterations $R$             Control Parameter Increment $p_{incr}$
Outputs:   Optimal Solution $BestS$
Algorithm:
    $items \leftarrow \dim(S_0)$;
    $CurS \leftarrow S_0$;              $CurCost \leftarrow cost(CurS)$;
    $BestS \leftarrow CurS$;            $BestCost \leftarrow cost(BestS)$;
    $p \leftarrow p_0$;                 $numIterations \leftarrow 0$;
    $counter \leftarrow 0$;
    While $counter \le R$
        $updateHistory(CurS)$;
        $PrevCost \leftarrow CurCost$;
        {Perturb}
        For $i$ from 1 to $items$ do
            $S \leftarrow move(CurS, i)$;             $Gain \leftarrow CurCost - cost(S)$;
            If $Gain > random(-p, 0)$ Then
                $CurS \leftarrow S$;
        $add(Moves, S)$;                          $add(Moves, CurS)$;
        $CurS \leftarrow makeState(CurS)$;            $CurCost \leftarrow cost(CurS)$;
        {Update control parameter}
        If $PrevCost = CurCost$ Then
            $P \leftarrow P + P_{incr}$;
        Else
            $P \leftarrow P_0$;
```

```
{Update BestS if needed, and counter}
If $CurCost < BestCost$ Then
    $BestS \leftarrow CurS$;          $BestCost \leftarrow CurCost$;
    $Counter \leftarrow Counter - R$;
Else
    $Counter \leftarrow Counter + 1$;
$NumIterations \leftarrow NumIterations + 1$;
$finishUpdateHistory(BestS, P, Counter, Moves)$;
```

Since the Stochastic Evolution algorithm deals, in concept, with minimization, the COST of a state should be the result of the evaluation function, mentioned in a previous chapter. The result of the MOVE of an element in a given state is, in other words, the solution resulting from the perturbation of this state, without trying to validate the changes. The MAKESTATE function validates the perturbed state by propagating the changes to the entire data-structure.

7.4.2 Object Oriented Design

Figure 17 shows an overview of the interaction between the different classes instantiated during the runtime of the algorithm, discussed below. The algorithm creates only one neighbor for the current solution per iteration from its current solution. This neighbor solution is compared to the current solution and potentially to the best solution found so far. Therefore, the algorithm handles only 3 solutions per iteration: a current and a neighbor solution, and the best one so far.

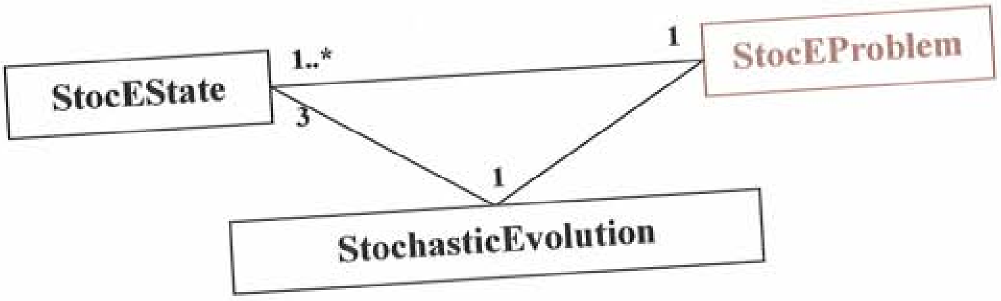


Figure 17: Brief Overview of StocE Classes Interaction

The problem-specific methods are handled by the class "StocEProblem". The algorithmic-dependent procedures are handled by the class "StochasticEvolution".

7.5 Tabu Search

Tabu Search (TS) is a heuristic procedure proposed by Fred Glover to solve discrete combinatorial optimization problems. The basic idea is to avoid that the search for best solutions stops when a local optimum is found, by maintaining a list of non-acceptable or forbidden (taboo) solutions/costs, called Tabu list or Short-Term Memory (STM). Advanced TS algorithms suggest that some of the best solutions found so far be saved, for search diversification, in a list called Long-Term Memory (LTM). An additional list, called Medium-Term Memory (MTM) may be used to intensify locally the search, by keeping track of solutions, with estimate close enough to that of the best solutions in the LTM.

The use of these memory lists explicit the fact that the updating process of the current and best solution does elude those that were marked in the lists, which grow and shrink per iteration. Occasionally, moving the current solution to a "forbidden" solution is allowed given a certain "aspiration" criteria, usually involving an improvement in cost from the current one.

As opposed to other algorithms, the current solution of the inner loop next iteration is selected from a set of N neighbor solutions, deduced from the actual current one by perturbing it N times. This solution will be overwritten if, at the beginning of the next iteration, a better solution was previously found in memory.

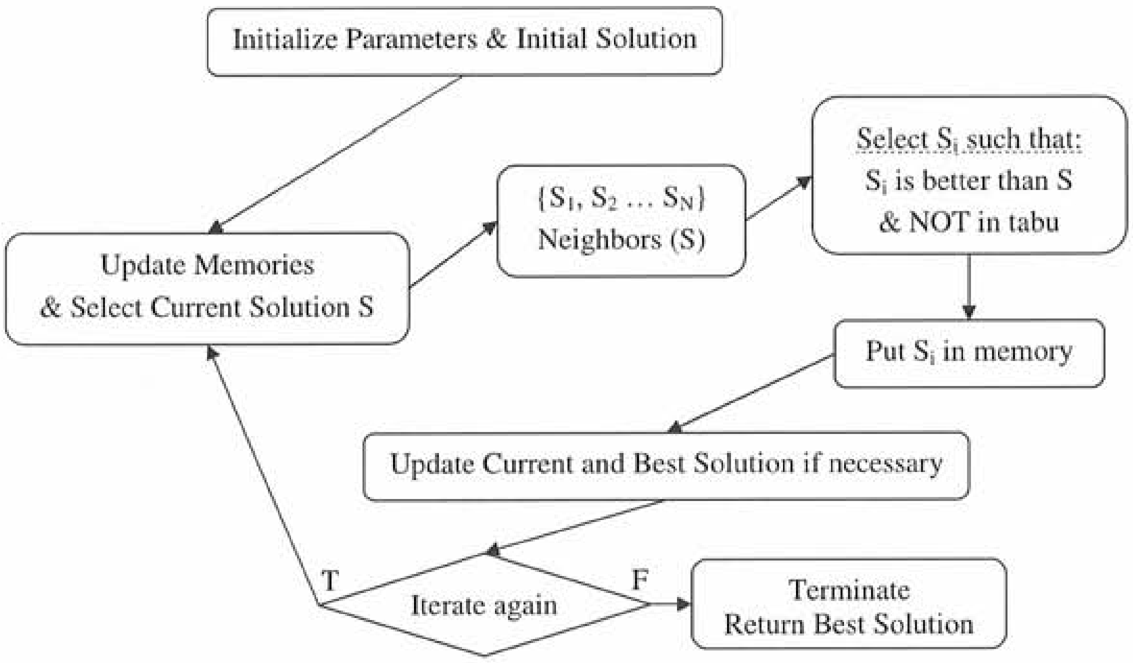

Initialize Parameters & Initial Solution
Update Memories & Select Current Solution S
$\{S_1, S_2 \ldots S_N\}$ Neighbors (S)
Select S_i such that: S_i is better than S & NOT in tabu
Put S_i in memory
Update Current and Best Solution if necessary
Iterate again
T
F
Terminate Return Best Solution

Figure 18: Advanced Tabu Search General Flow Diagram

7.5.1 Tabu Search Procedure

```
Inputs:   Initial Solution $S_0$        Number of iterations $numIterations$
          Number of neighbors $numNeighbors$
Precondition:       $numIterations > 0$ and $numNeighbors > 0$
Outputs:  Optimal Solution $BestS$
Algorithm:
    {Initialization of CurS, and BestS, STM}
    $CurS \leftarrow S_0$;
    $CurEstimate \leftarrow estimate(CurS)$;
    $BestS \leftarrow CurS$;
    $BestEstimate \leftarrow CurEstimate$;
    $LTM \leftarrow createDynamicList(\ )$;
    $i \leftarrow 1$;
    While $TRUE$ do
        If $Terminate$ or $i > numIterations$ then
            Exit While
        {LTM is updated with the current solution and/or the best one, STM is cleared}
        $updateMemory(\ )$;
        {Current Solution and Estimate are selected from LTM first if possible}
        $CurS \leftarrow selectFromMemory(\ )$;
        $CurEstimate \leftarrow estimate(CurS)$;
        $updateHistory(CurS)$;
        $add(STM, CurEstimate)$;
        {Generate a fixed number of perturbed solutions from current one}
        $Neighbors[\ ] \leftarrow neighbors(CurS, numNeighbors)$;
        For $j$ from 1 to $numNeighbors$ do
            $Sol \leftarrow Neighbors[j]$;
            $SolEstimate \leftarrow estimate(Sol)$
            {If neighbor is NOT tabu OR tabu but aspires to be better than current solution}
            {Otherwise, consider the neighbor as tabu}
            If $NOT\{isTabu(Sol)\}$ or $aspiration(Sol, CurS)$ Then
                $CurS \leftarrow Sol$;          $CurEstimate \leftarrow SolEstimate$;
                {Update best solution so far if neighbor aspires to be the best solution}
                If $aspiration(Sol, BestS)$ Then
                    $BestS \leftarrow Sol$;        $BestEstimate \leftarrow SolEstimate$;
            Else If $NOT\{isTabu(Sol)\}$ and $NOT\{aspiration(Sol, CurS)\}$ Then
                $add(STM, SolEstimate)$;
        $finishUpdateHistory(BestS, Neighbors[\ ], STM)$;      $i \leftarrow i + 1$;
```

In this implementation, only the estimates are moved to the STM list, instead of the whole solution's properties. Therefore, the check to see if a solution is tabu becomes as simple as testing for its penalty. The aspiration method implemented relies on finding the minimal or maximal solution estimate.

If the Tabu Search algorithm deals with minimization, the ESTIMATE of a solution should be the result of the evaluation function, mentioned in a previous chapter, and multiplied by –1 otherwise. For both approaches, the result of the NEIGHBOR function for a given solution is, in other words, a set of results of the perturbation of this solution.

The fact that TS may deal, with either minimization or maximization, yields the obligation to implement accordingly the method "$BestSolution(sol1, sol2)$", on which relies the aspiration procedure.

7.5.2 Object Oriented Design

Figure 19 shows an overview of the interaction between the different classes instantiated during the runtime of the algorithm, discussed below. The algorithm creates N neighbors per iteration from its current solution, where some of them may be compared to it and some to the best solution found so far. Therefore, the algorithm handles $N+2$ solutions per iteration.

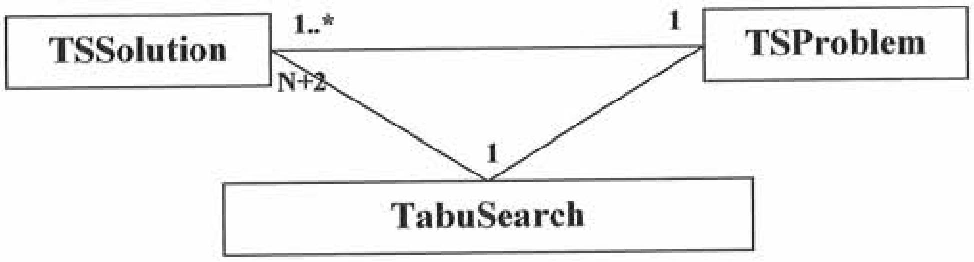


Figure 19: Brief Overview of TS Classes Interaction

The problem-specific methods are handled by the class "TSProblem". The algorithmic-dependent procedures are handled by the class "TabuSearch".

Chapter 8 Conclusion

This thesis focused on the NP-Hard problem of finding an optimal tree topology where leaves represent biological sequences. [2, 3] The problem consists of minimizing the number of changes between given and/or derived sequences. As the number of sequences to be compared increases, the size of the search space grows exponentially. This fast growth induces the necessity of using optimization methods in order to come up with an acceptable optimal topology.

Since the phylogenetic trees literature have only discussed the use of an exact method (Fitch) or only one or at most two Heuristics (Genetic Algorithms, Simulated Annealing, Tabu Search, Simulated Evolution and Stochastic Evolution) in each one, with few of them having some pseudo-code/code or none what so ever, this thesis attempted to suggest an encoding schema ready to use in one or more of the above-mentioned algorithms, including the suggested alternatives.

Thus, as a conclusion, this research tried to settle Genetic Algorithms, Simulated Annealing, Simulated Evolution, Stochastic Evolution and Tabu Search, to a common encoding ground, allowing researches in that field to be able to compare them all simultaneously, and come up with a relatively good ordering of these heuristics. In addition, the generic implementations of these heuristics can be used in other NP problems, not necessarily dealing with bio-informatics. The detailed tracing/chronology property of the algorithms can be helpful essentially for detailed evaluation and analysis of the runs executed for the algorithms, with the corresponding encoding.

Chapter 9 Future Work Ideas

This research work can be extended by:

- Specifying more accurately the scoring matrix and external factors
  Even thought this task is the responsibility of the biologist, the computer scientist's opinion does count, in a way that it will help mapping the factors into an accurate mathematical and computing science formula.
- Applying benchmarks to validate the algorithms
  In order to accurately analyze the suggested algorithms generic alternatives, benchmarks for different problem encodings have to be applied.
- Properly combining Heuristics to get optimal tree the fastest possible
  In the suggested encoding schema, the evaluation function of a data-structure, representing a phylogenetic tree, by itself, has a non-negligible computational load, which might be handled by encoding and applying a heuristic or other fast algorithm. In other terms, computing the optimal evaluation can be treated as a separate problem to deal with using heuristics or fast algorithms.

Some problems that appeared while researching the heuristics are the following:

- Fixed Input Parameters: which means that there is a need to guess or write an algorithm to tune those parameters before running the algorithm itself, so that the heuristic runs faster.
  - → "Genetic Algorithms" requests a given number of generations, and population size, among others.
  - → "Simulated Annealing" needs an initial temperature T_0, motion and cooling rates, and initial number of jumps M_0, among others.
  - → "Simulate Evolution" requests a given fixed bias number ($|B| < 1$)
  - → "Stochastic Evolution" works for a given number of iterations, and a control parameter and increment, among others.
  - → "Tabu Search" explores the search space for a given number of iterations, and a number of neighbors per iteration.
- Random-ness doesn't handle potentially repeated moves/changes:

→ "Genetic Algorithms"	in methods:	MUTATION & CROSSOVER.
→ "Simulated Annealing"	in method:	NEIGHBOR.
→ "Simulated Evolution"	in method:	ALLOCATION.
→ "Stochastic Evolution"	in method:	MOVE.

- Starting Solution / State may be too far from the optimal one, thus the algorithm will take a longer time to discover that optimal one.
- Getting stuck with one procedure, even very close to a fairly acceptable solution.
  The use of a heuristic or fast algorithm might depend on how far the current solution/state is from the optimal one, thus switching between algorithms might aver to be helpful.

Chapter 10 References

1. Kim, J & Warnow, T. *Tutorial on phylogenetic tree estimation*. Retreived March 05, 2005, from: http://kim.bio.upenn.edu/~jkim/media/
2. Moret, B, Bader, D, & Warnow, T. *High-performance algorithm engineering for computational phylogenetics*. Retreived March 05, 2005, from: http://www.cs.umd.edu/class/spring2003/cmsc838t/papers2/
3. Stamatakis, A., Ott, M., & Ludwig, T. *RAxML-OMP: An efficient program for phylogenetic inference on SMPs*. Retreived March 06, 2005, from: http://www.ics.forth.gr/~stamatak/publications/
4. Opper, D. *Parsimony phylogenetic trees*. Retreived March 06, 2005, from: http://www.icp.ucl.ac.be/~opperd/private/parsimony.html
5. Thinkquest 2001: International internet challenge. Retreived March 06, 2005, from Genetic engineering, the creation website: http://library.thinkquest.org/C0123260/
6. Felsenstein, J. (1982). Numerical methods for inferring evolutionary trees. *The Quarterly Review of Biology*, *57* (4).
7. Fitch, Wm. (1971). Toward defining the course of evolution: minimum change for a specified tree topology. *Syst Zool*, *20*, 406-416.
8. Shürer, K. *Branch and bound and its application to phylogeny*. [n.p.] : [n.p.]
9. Affenzeller, M & Mayrhofer, R. *Generic heuristics for combinatorial optimization problems*. [n.p.] : [n.p.]
10. Fitch, W.M. (1975). Toward finding the tree of maximum parsimony. In G.F. Estabrook (Ed.) Proceedings of the *Eighth International Conference on numerical taxonomy*, (pp. 189-230). San Francisco: W.H. Freeman
11. Barker, D. (2004). *LVB: Parsimony and simulated annealing in the search for phylogenetic trees*. [n.p.] : [n.p.]
12. Busetti, F. (2004). *Simulated annealing overview*. [n.p.] : [n.p.]
13. Clarke, J, Harman, M, Hierons, R, Jones, B, Lumkin, M, Roper, M, et al. *The application of meta-heuristic search techniques to problems in software engineering*. [n.p.] : [n.p.]
14. Sait, S., Abd-El-Barr, M., Al-Saiari, U., & Sarif, B. *Fuzzified simulated evolution algorithm for combinational digital logic design targeting multi-objective optimization*. [n.p.] : [n.p.]
15. Sait, S, Youssef, H, Khan, J, El-Maleh, A. *Fuzzified iterative algorithms for performance driven low-power VLSI placement*. [n.p.] : [n.p.]
16. Khan, J. *Performance driven, low-power, standard VLSI cell placement using simulated evolution*. [n.p.]:[n.p.]