Spring 2011

RNA SECONDARY STRUCTURE PREDICTION TOOL

Meenakshee Mali
San Jose State University

RNA SECONDARY STRUCTURE PREDICTION TOOL

A project

Presented to

The Faculty of the Computer Science Department

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Meenakshee Mali

May 2011

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

RNA SECONDARY STRUCTURE PREDICTION TOOL

by
Meenakshee Mali

APPROVED FOR THE COMPUTER SCIENCE DEPARTMENT

Dr. Sami Khuri,	Computer Science Department	Date
Dr. Chris Pollett,	Computer Science Department	Date
Dr. Robert Fowler,	Biology Department	Date

ABSTRACT

Ribonucleic Acid (RNA) is one of the major macromolecules essential to all forms of life. Apart from the important role played in protein synthesis, it performs several important functions such as gene regulation, catalyst of biochemical reactions and modification of other RNAs. In some viruses, instead of DNA, RNA serves as the carrier of genetic information. RNA is an interesting subject of research in the scientific community. It has lead to important biological discoveries. One of the major problems researchers are trying to solve is the RNA structure prediction problem. It has been found that the structure of RNA is evolutionary conserved and it can help to determine the functions served by them. In this project, I will be developing a tool to predict the secondary structure of RNA using simulated annealing. The aim of this project is to understand in detail the simulated annealing algorithm and implement it to find solutions to RNA secondary structure. The results will be compared with the very famous tool Mfold, developed by Michael Zuker, using the minimum free energy approach.

ACKNOWLEDGEMENTS

I would like to thank Dr.Sami Khuri for his continuous guidance and support for my academic journey at San Jose State University. I express my sincere gratitude towards him as he was the one who introduced me to the field of Bioinformatics. He has supported me constantly to accomplish my project successfully. I would also like to thank Dr. Chris Pollett and Dr. Robert Fowler for serving on my defense committee.

Last but not the least, I would not have achieved this milestone without continuous support of my husband and family. I also thank all my friends at San Jose State University who have made my years as a graduate student unforgettable.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

The central dogma of molecular biology states that the genetic information of an organism is transferred from Deoxyribonucleic Acid (DNA) to Ribonucleic Acid (RNA) and then to Proteins. For a long time DNA was considered as the primary actor in storing the genetic code with RNA cast into secondary role of carrier of this information. But a string of discoveries in the last decade have proved that smaller RNA molecules operate many cell controls. The knowledge about RNA is expanding rapidly. It is now known that RNA catalyzes reactions, directs the site-specific modification of RNA nucleotides, modulates protein expression and serves in protein localization. Therefore, understanding the function of RNA molecules is key to unlocking the pathways of disease and biology.

Knowing the precise three dimensional structure of RNA is one of the foremost goals of molecular biology, for it is this structure that determines the molecule's function [1]. Nuclear Magnetic Resonance and X-ray crystallography are some of the available experimental methods generally used for this purpose. But these are very costly, time consuming and not always feasible methods. As a result, it is easy to determine the sequence of RNA compared to the three dimensional structure. The gap between the number of proteins whose sequence is known (in thousands) compared to whose complete three dimensional structure is known (in hundreds) is widening on an yearly basis. This has lead to intense research into structure predicting methods using computational algorithms.

The building blocks of DNA and RNA are nucleotides. Three components are present in RNA nucleotides: the nitrogenous base, the sugar and the phosphate group. The RNA backbone is made of ribose five atom carbon-sugar counted from 1' through 5' and it is attached by two phosphate groups in 3' and 5', respectively [2]. The nitrogen base in RNA are made of four different bases, Adenine(A), Guanine(G), Cytosine(C), and Uracil(U). Uracil is replaced by Thymine(T) in DNA. The phosphate groups in the backbone of RNA have a negative charge which makes RNA a charged molecule [3]. Due to this, the RNA molecule in a cell is not inherently stable and to gain stability, it folds on itself. A nucleotide in one part of RNA can make base-pair with a complementary nucleotide in another part of RNA. Furthermore a nucleotide sequence uniquely determines the folding pattern and hence we can attempt to predict its structure. Listing out all the base pairs given a nucleotide sequence is considered as secondary structure prediction. The secondary structure of RNA is the scaffolding of its tertiary structure. It is well known that RNA folding is hierarchical: "the primary sequence determines the secondary structure and the secondary structure in turn determines the tertiary folding."[4]

The nucleotides form Watson-Crick base pairs namely, AU, GC, and their mirrors. Also GU base pair is found in many RNAs. This base pair is called as wobble pair. With these combinations of bases, RNA structure forms two large groups: Stem-loops and Pseudoknots.

There are multiple motifs possible in RNA secondary structure. These are,

- Single strand
- Double strand / Stem
- Single-nucleotide bulge
- Three-nucleotide bulge
- Hairpin loop
- Internal loop

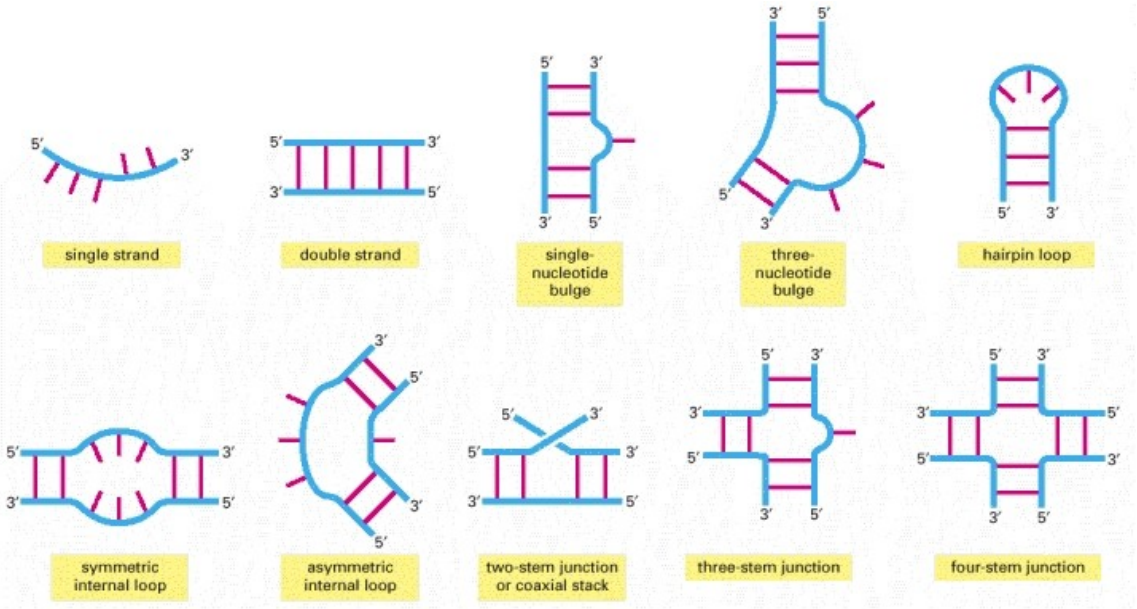

Figure 1 Representation of different elements of RNA secondary structures [5]

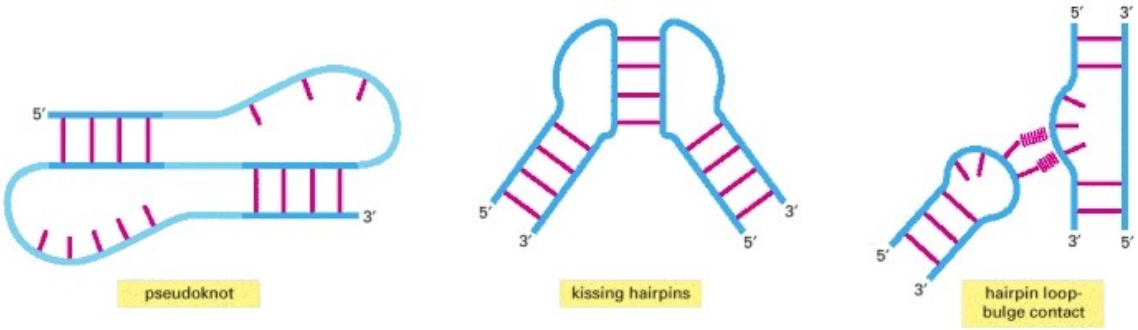

Figure 2 Representation of different tertiary structures [5]

When the secondary structures represented in Figure 1 form base pairs between them, tertiary structures are formed. Tertiary structures are very difficult to predict. Figure 2 shows the different tertiary structures.

1.1 Project Scope

In this project, we plan to implement the simulated annealing algorithm to predict the secondary structure. In Chapter 2, we describe the methods used for secondary structure prediction along with simulated annealing algorithm. Simulated annealing has been used extensively to solve optimization problems in various disciplines. We will use efn2 model as measure of acceptance criteria for new structures predicted from mutation operation. In Chapter 3, we describe the computing environment we have setup for performing the experiments. The GNU scientific library will be used to provide the framework for simulated annealing whereas RNAStructure library will be used to compute the efn2 energy of a secondary structure. Chapter 4 explains the mapping of the secondary structure problem into simulated annealing algorithm. We will be predicting secondary structure for a set of sequences drawn from NCBI. The results will then be compared with corresponding results from Mfold using a defined comparative measure. These would be listed in Chapter 5. We will also describe in chapter 5 the effect of different perturbation functions on the quality of the secondary structure. We also plan to study different cooling schedules and its influence on the convergence rate of the algorithm.

2 LITERATURE REVIEW

2.1 Dynamic Programming

Pioneering work in RNA secondary structure prediction was done by Nussinov,R. and Jacobson, A. Their dynamic programming algorithm finds an optimal structure possible for a given sequence. The basic algorithm tries to find maximum base pairs in a given nucleotide sequence. Later they incorporated rules for calculating loop stability based on free energy into the algorithm [6]. The algorithm for maximal matching is based on a rule that the stability of G-C pairs is equal to that of A-U pairs. The stabilizing energy of stacking base pairs and destabilizing energy of single stranded loops is ignored in this algorithm. The dynamic programming technique builds an optimal solution to the problem by solving sub-problems. This approach applied to find structure, tries to find sub-structure for different length of given sequence. The algorithm uses following recurrence equation,

$$S(i, j) = \max \begin{cases} S(i+1, j-1) + w(i, j) \\ S(i+1, j) \\ S(i, j-1) \\ \max_{i<k<j} S(I, k) + S(k+1, j) \end{cases} \qquad (2.1.1)$$

w(i,j) is 1 if i and j form complementary base pair. It is assigned to 0 otherwise. The 3 steps in this algorithm are initialization, recursion, and traceback. The sequence is compared against itself and dynamic programming matrix is created. Figure 3 shows the initialization step of Nussinov algorithm [7].

	G	G	G	A	A	A	U	C	C
G	0								
G	0	0							
G		0	0						
A			0	0					
A				0	0				
A					0	0			
U						0	0		
C							0	0	
C								0	0

Example:

GGGAAAUCC

$S(i,i) = 0 \quad \forall i,\ 1 \le i \le L$ the main diagonal
$S(i, i-1) = 0 \quad \forall i,\ 2 \le i \le L$ the diagonal below

Figure 3 Initialization of dynamic programming matrix [7]

In the next step, this matrix is filled up using the recursion relation stated in (1). Figure 4 is the matrix build with the recurrence.

	G	G	G	A	A	A	U	C	C
G	0	0	0	0					
G	0	0	0	0	0				
G		0	0	0	0	0			
A			0	0	0	0	?		
A				0	0	0	1		
A					0	0	1	1	
U						0	0	0	0
C							0	0	0
C								0	0

Figure 4 Second step of dynamic programming algorithm [7]

After the matrix is complete, the last step is to traceback the matrix to get optimal structure. Depending on the sequence the dynamic programming approach can also yield different optimal structures. Figure 5 is the traceback of the matrix and resulting optimal structure of the sequence.

	G	G	G	A	A	A	U	C	C
G	0	0	0	0	0	0	1	2	3
G	0	0	0	0	0	0	1	2	3
G		0	0	0	0	0	1	2	2
A			0	0	0	0	1	1	1
A				0	0	0	1	1	1
A					0	0	1	1	1
U						0	0	0	0
C							0	0	0
C								0	0

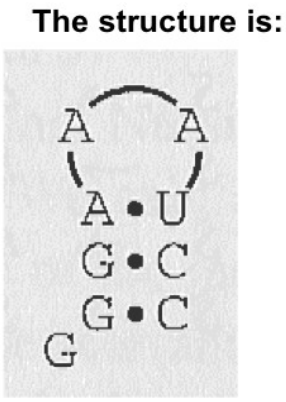

Figure 5 Traceback of the dynamic programming matrix and optimal structure[7]

2.2 Minimum Free Energy (MFE)

The basic dynamic programming algorithm was then modified to calculate minimum free energy of the structure. Gibbs energy (also referred to as ΔG) is the chemical potential that is minimized when a system reaches equilibrium at constant pressure and temperature [8]. It is calculated by the following formula,

$$\Delta G = \Delta H - T\Delta S \qquad (2.2.1)$$

where ΔH = enthalpy
ΔS = entropy
T = temperature in unit Kelvin

The different interactions (hydrogen bonds, van der Waals, and electrostatic) between the molecules, define the enthalpy i.e ΔH of the system. Or in simple words, it is the energy contained within the system. ΔS or entropy is the change in energy of the system. When ΔG is 0 the system is said to be in an equilibrium state. If ΔG is greater than 0, the system is in unfavorable process and if it is less than 0, system is in favorable process [9].

The minimum energy for structure is calculated by adding the experimentally pre-determined values for each base pair found in the dynamic programming matrix. The free energy of each motif depends only on the sequence of that motif and the most adjacent base pairs. The total free energy is the sum of the increments. This algorithm is implemented in the benchmark tool for RNA secondary structure prediction tool, MFold. This approach is called as Minimum Free Energy (MFE) and was developed by M. Zuker [10].

There are certain limitations to MFE method. In this method the energies of bulge loops and single non-canonical pairs are not taken into account. RNA folding process does not always occur at equilibrium. Kinetics of the process is also important. Because of this, the structure obtained by MFE might not be the same as the native fold. Other drawback of Mfold is lack of predicting pseudoknots in a structure.

2.3 Evolutionary Algorithm (EA)

This type of algorithm is developed around an evolutionary model that mimics the process of natural evolution. It gives a number of probable solutions at each generation. When applied to RNA structure prediction, EA will give a set of low energy structures at each generation. The initial population of solutions is generated randomly before the algorithm commences. The next population of solutions is formed by evaluating the solutions in previous generation with some criteria and discarding the solutions which do not satisfy.

Wiese and Glen designed a serial EA, RnaPredict [11], which encodes RNA secondary structures as permutations. The quality of the predictions by RnaPredict was compared with the predictions of Nussinov dynamic algorithm. Initial step in RnaPredict is to generate a set of valid helices. A valid helix has a minimum of three adjacent canonical base pairs and a minimum hairpin of size three. To generate helices, first set of all base pairs in a given sequence is found. The algorithm then iterates over this set of base pairs and builds a helix by stacking valid base pairs. If the resulting helix meets or exceeds the above requirement it is added to the set H of possible helices. Once the set of possible helices is formed the structure prediction problem becomes combinatorial optimization problem[11]. To ensure chemically feasible structure, no predicted structure may contain helices that share bases. Depending on how the helices conflict, both permutations could result in vastly different structures. Helix conflicts are eliminated by decoding the permutation from left to right. The helix specified at each point in the permutation is checked for conflicts with helices to its left.If no conflicts are found the helix is retained; otherwise it is discarded[11]. For example, assuming set H contains five helices (0,1,2,3,4) and (4,0,1,3,2) are two possible structures. Then for the second set, we start with helix 4. Then if helix 0 does not share any bases with helix 4 it is considered part of the final structure. We continue this process until the end of the set. RnaPredict attempts to optimize the structures such that they are both chemically feasible and have free energy close to ΔG. Since it yields, a population of candidate solutions it is possible to investigate not only the minimum free energy structure but also other low energy structures that may be close to native fold[11]. Each generation of EA has three key steps.

2.3.1 Crossover

In this step, offspring solutions are formed by combining the two parent solutions. All solutions have parts that are favorable and unfavorable. Crossover operator is chosen such that all favorable parts go into one solution and unfavorable into the other. There are three different types of crossover operators.

- Order Crossover

Several helix positions are chosen randomly and the order in which these helices appear in one parent is imposed on the other parent[11].

- Partially Mapped Crossover

Two crossover points are chosen randomly and a series of successive swapping is done between the two parents[11].

- Cycle Crossover

In this type of crossover, any one parent and a random position in this permutation is chosen. The offspring solution then inherits the helix in that position from selected parent. This cycle of length $1 \leq k \leq l$ continues until the length of the permutation. ($x_k = x_l$) All remaining helices are inherited from second parent[11].

2.3.2 Mutation

Random changes in the population are introduced via mutations. This step is used to avoid premature genetic convergence in the population. It also maintains genetic diversity in the solution.

2.3.3 Selection

This is a step where new solutions are chosen from old solutions. The choice is made by scoring each solution against a fitness function. It is a task of an EA to select good solutions and reject others based on their scores. Selection can act on parents, the old population, and the new population. It can be local (within a subpopulation) or global (within entire population)

These steps are repeated for a predetermined number of generations, a predetermined amount of time or until the population converges[11].

2.4 Simulated Annealing

Physical annealing is the process of heating the metal to a temperature above it's crystallization point and then gradually reducing the temperature to make the metal hard. In an annealing process, a metal, initially at high temperature and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at T=0. Hence the process can be thought of as an adiabatic approach to the lowest energy state. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in metastable states (that is trapped in a local minimum energy state) [32].

The traveling salesman problem can be used as an example application of simulated annealing. In this problem, a salesman must visit some large number of cities while minimizing the total mileage traveled. If the salesman starts with a random itinerary, he can then pairwise trade the order of visits to cities, hoping to reduce the mileage with each exchange. The difficulty with this approach is that while it rapidly finds a local minimum, it cannot get from there to the global minimum.

Simulated annealing tries to improve this strategy through the introduction of two approaches. The first approach is the Metropolis scheme. The original Metropolis scheme was that an initial state of a thermodynamic system was chosen at energy *E* and temperature T, holding T constant the initial configuration is perturbed and the change in energy *dE* is computed. If the change in energy is negative the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by the Boltzmann factor *exp -(dE/T)*. This process is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at T=0. This allows the solver to explore more of the possible space of solutions. If T is large, many "bad" configurations are accepted, and a large part of solution space is thus accessed [12].

The second approach is, again by analogy with annealing of a metal, to lower the temperature. After making many choices for possible configuration and observing that the cost function declines only slowly, one lowers the temperature, and thus limits the size of invalid choices of configuration. After lowering the temperature several times to a low value, one may then quench the process by accepting only "good" configurations in order to find the local minimum of the cost function. There are various annealing schedules for lowering the temperature, but the results are generally not very sensitive to the details.

There is another faster strategy called threshold acceptance [13]. In this strategy, all good configurations are accepted, as are any bad configurations that raise the cost function by less than a fixed threshold. The threshold is then periodically lowered, just as the temperature is lowered in annealing. This eliminates exponentiation and random number generation in the Boltzmann criterion. As a result, this approach can be faster in computer simulations.

Formally, there are four main parts of Simulated Annealing.

1. Initial State:
   In this phase, problem and it's parameters are represented.
2. Mutation Function:
   This phase, creates random changes in the state of problem.
3. Cost Function:
   Cost function is used to determine how good the current solution is.
4. Decision Mechanism:
   It is used to decide either to accept or reject the solution.

These parts can be understood by analyzing the basic structure of iteration optimization algorithm described below. Initial design is formed using problem's parameters and it is evaluated with cost function. Random changes are then introduced to the design using mutation function and this new design is again evaluated.

If the cost of this new design is better (minimum or maximum) then this new design is accepted as a solution and next solution is formed using this solution [14].

```
Design = InitialDesign;
        Cost = Evaluate(Design);
        while not done do
                NewDesign = Mutate(Design);
                NewCost = Evaluate(NewDesign);
                DeltaCost = NewCost - Cost;
                if appropriate then
                        Cost = NewCost;
                        Design = NewDesign;
                end if
        end while
```

The decision mechanism is a probabilistic function. The probability of accepting the new solution is specified by an acceptance probability function [15].

H. Tsang applied Simulated Annealing for RNA secondary structure prediction in SARNA-Predict [16]. The main difference between the algorithm described above and SARNA-Predict is in it's decision mechanism. For structure prediction the criteria for accepting or rejecting a solution is based on it's energy. To avoid the problem getting stuck in local minima at the beginning the solutions with higher energy are accepted with some probability. The problem is encoded as an integer permutation of helices similar to the Evolutionary Algorithm discussed in above section. The constraints under which a helix is formed are,

1. A stem (stacked pairs) is formed only when three or more adjacent pairs form.
2. At least three nucleotides are required to form the loop connecting to the stem.
3. There should not be any conflicting base pairs in the helices, i.e one helix should not share base pairs with others.

Using permutation-based SA, we can view the problem of predicting the secondary structure of RNA as one of picking the subset S of helices from the set of all possible helices H, such that the free energy E(S) is minimized and that no helices in S share one or more bases [17]. If the set of all helices, H, contains n helices, then use a permutation of length n to represent a candidate solution. The order in which a helix appears in the permutation is the order in which it is picked by the decoder to be inserted into the final structure. Helices that are incompatible with any previously selected helices are rejected.

```
Structure = InitialStructure;
FreeEnergy = Evaluate(Structure) ;
Temperature = InitialTemperature;
while (Temperature > FinalTemperature) do
        for (i = 1 to NumberOfIterations) do
                NewStructure = Mutate(Structure);
                NewFreeEnergy = Evaluate(NewStructure);
                ΔEnergy = NewFreeEnergy - FreeEnergy
                if (ΔEnergy ≤ 0) OR (with Probability[Accept]= e^-ΔEnergy/Temperature)then
                   FreeEnergy = NewFreeEnergy;
                   Structure = NewStructure;
                end if
        end for
        decrease Temperature
end while
```

2.4.1 Cost Function

In SARNA-Predict, the energy of structure is calculated using three different thermodynamic models namely, Individual Nearest Neighbor with Hydrogen Bonds (INN-HB), efn2, and HotKnots[16]. The difference between these models is in how they assign energies to different structure elements. INN-HB model doesn't consider the structure elements such as bulge loop. efn2 and HotKnots are improved to take into account the different structural elements. HotKnots model is used in the SARNA-Predict-pk algorithm which can predict the structures with pseudoknots [18].

2.4.2 Perturbation Function

A novel combination of permutation based encoding and swap mutation is implemented in SARNA-Predict as a mutation function. Swap mutation as the name suggests chooses two random points in the permutation and swaps the two helices.For a permutation vector, p = (H_I,...H_i,...H_j,...H_n) , where n is the number of potential helices. A swap mutation is defined as,

P_{old} = { H_1, ...H_i,.....H_j,.....H_n) --> P_{new} = {H_1,.....H_j,.....H_i,.....H_n) (2.4.2.1)

where i and j subset [1,n] are randomly chosen positions [17].In the classical SA sense, each perturbation step will only swap by one swap mutation step. The difference between the new conformation and the old conformation is one step. Another mutation operator used is percentage swap mutation operator. The number of swap mutations is found by taking the product of percentage of total number of helices and the current annealing temperature.

2.4.3 Cooling Schedule

Annealing schedule makes use of temperature as the main controlling parameter. The algorithm starts with very high temperature. At this stage, some solutions with high energy are accepted with some probability. After the temperature reaches 0, this probability should tend to 0 and thus accepting only the solutions with low energy. The algorithm thus becomes greedy once temperature reaches 0. In this way the algorithm can produce a global optimum solution. There are different types of annealing schedules. Geometric schedule and Adaptive schedule are implemented in SARNA-Predict[16]. In general, the choice of suitable cooling schedule have a profound effect on the performance of the algorithm and it is highly problem dependent.

SARNA-Predict has implemented two methods to decrement the value of the temperature parameters: geometric and adaptive rate schedulers. Geometric scheduler is defined as $T_{new} = \alpha T_{old}$, here α is the cooling ratio (set to 0.95) According to previous research [19] , α should be set to between 0.8 and 0.99. T_{new} and T_{old} are the new and old temperature values respectively. In the adaptive scheduler, the length of a subchain with constant temperature is set to the number of the local neighborhood. The number of iterations per temperature is reduced according to following equation [20].

$$T_n = T_{n-1} \left(1 + \frac{\ln(1+\delta)\, T_{n-1}}{3\sigma(T_{n-1})} \right)^{-1} \qquad (2.4.3.1)$$

where $\sigma(T_{n-1})$ is the standard deviation of the values of the cost function at the current temperature and δ is the distance parameter. The size of δ determines the speed of the reduction of the temperature and Aarts et. al suggest the value $\delta = 0.1$.[20]

The main advantages of SA over other local search optimization algorithms are its flexibility and ability to approach global optimality. The algorithm is quite versatile since it does not rely on any restrictive properties of the model. Although SA is a powerful tool for finding the approximate solution to combinatorial optimization problems, SA is slow to converge when compared to other deterministic algorithms, due to the slow cooling schedule required [20]. As a result, the computationally intensive nature of this algorithm has been its major drawback [21].

However, since this technique has proven to be so useful, and increases in computational power are inevitable, it will only be a matter of time before a functionally superior design for this technique is found. In fact, there is research being done on implementing SA to run on parallel architecture. These parallel versions of the algorithm definitely reduce the time spent in evaluating the solution [22].

Other major weaknesses of SA include the tailoring work required to account for different classes of constraints and the need to fine-tune the parameters of the algorithm, which can be rather delicate [23]. Also, the precision of the numbers used in the implementation of SA can have a significant effect upon the quality of the outcome. Finally, there is a clear trade-off between the quality of the solutions and the time required to compute them.

In the next chapter, we describe the computing environment set up to implement the simulated annealing algorithm described in section 2.4 and compute the results.

3 COMPUTING ENVIRONMENT

3.1 Hardware Environment

Simulated Annealing is computationally intensive. The original algorithm developed by H. Tsang was evaluated on 128- node beowulf cluster [16], each node with Pentium 4 running at 3GHz. These nodes were connected with a Gigabit Ethernet Network.
The hardware environment we set up for the evaluation is a single desktop system with following configuration.

Operating System	Ubuntu 10.04
Processor	AMD Athlon Quad-Core
Speed	3.0 GHz
RAM	8.0 GB

Table 1 Hardware Environment Setup

3.2 Software Environment

The algorithm was implemented using C++. The GNU Scientific Library (GSL) is a collection of routines for numerical computing. It is free software under the GNU General Public License[24]. The library implements C routines for simulated annealing. Additionally we used 'Boost', which is a set of free peer-reviewed portable C++ source libraries. [25]. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) and will be in the new C++0x Standard now being finalized.

RNAStructure [26] is another package we used to implement wrapper functions and utilities. It is a package for RNA and DNA secondary structure prediction and analysis developed at University of Rochester Medical Center. It provides multiple algorithmic implementations for secondary structure prediction. It can also predict secondary structures common to two, unaligned sequences, which is much more accurate than single sequence secondary structure prediction. To build the software, we used Scons, which is an Open Source software construction tool implemented in Python designed to replace the classic *Make* utility. SCons is an easier, more reliable and faster way to build software. The implementation is developed under Ubuntu 10.04 operating system.

In the next chapter we describe the process of mapping the structure prediction problem into simulated annealing framework.

4 MAPPING STRUCTURE PREDICTION INTO SA

Here we describe the design details of mapping the secondary structure prediction into simulated annealing framework. Formally, the problem can be broken into 4 different steps.

4.1 State Representation

First task in applying the simulated annealing method to problem is representing the problem's parameters. The algorithm implemented in this project uses the permutation of integers (helices) to represent the secondary structure of RNA. In case of Traveling Salesman Problem the solution can be represented as a permutation of integers where each integer corresponds to a particular city. So, for example one of the solutions is (1,3,2,6,4,5) represents a tour which starts at city number 1, following 3 and so on.

The secondary structure predicted by this algorithm is a permutation of integers. Candidate helices are encoded as integers. There are three constraints on formation of helix from all possible base pairs[17],

- A stack is formed only when three or more adjacent base pairs are formed.
- The loop connecting the stacked pair must be at least three nucleotides long.
- The helices should not conflict with each other i.e. they should not share bases with each other.

The helix generation algorithm was first specified by Wiese K.C. and Hendriks. A during the development of an evolutionary algorithm for RNA secondary structure prediction. The details of this algorithm are stated in [27]. The algorithm tries to build a set H of all helices which could form in a given sequence. This set H should agree to the constraints listed above. The algorithm for generating potential helices starts with generating all possible base pairs in a given sequence. Next, a stack of base pairs is formed by iterating through the set of base pairs and adding base pairs on existing base pairs. This step is repeated until first non-canonical base pair is encountered. The constraint to validate a loop is checked at this point. A potential helix is shown in Figure 6 and the helix generation algorithm is described below it.

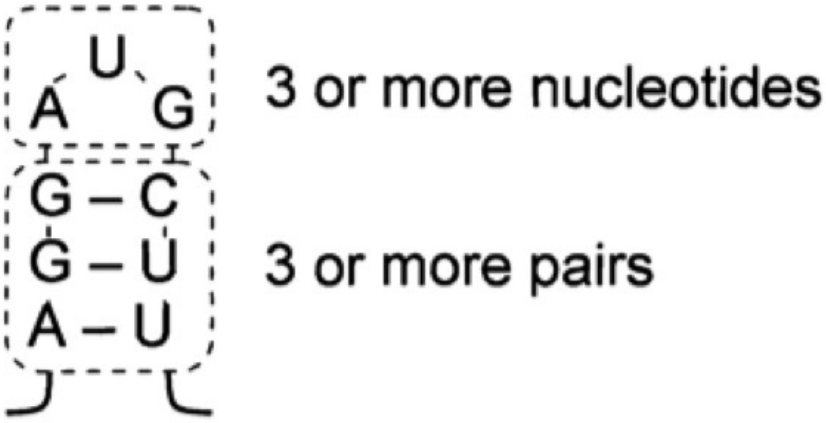

Figure 6 A potential helix generated by the helix generation algorithm[28]

```
Generate set of possible base pairs from (ri,rj) from given sequence;
    Initialize helix h;
       for each pair (ri,rj) do
         while (helix h is valid) and (helix h is incomplete) do
             if((ri,rj) is canonical base pair and (ri,rj) is not part of an existing helix then
                   add base pair (ri,rj) to helix h;
                   increment index i;
                   increment index j;
             else
                   if (helix h contains less than 3 base pairs) then
                     helix h is invalid;
                   else if (helix h has less than 3 bases between the last base pair) then
                     helix h is invalid;
                   else
                     helix h is complete;
                   end if;
            end if;
         end while;
         if (helix h is valid) then
           insert helix h into set of all helices H;
         end if;
       end for;
```

Helix generation algorithm [14]

With the help of permutational encoding of helices, the problem of RNA secondary structure prediction can be viewed as one of picking the subset S of helices from the set of all possible helices H, such that the free energy E(S) is minimized and that no helices in S share bases with each other.

The table below describes the total number of helices found in each sequence.

Organism	RNA Class	Number of Nucleotides	Base pairs in known structure	Number of Helices
B. subtilis	5s rRNA	118	70	188
S. cerevisiae (X67579)	5s rRNA	118	37	213
H. marismortui (AF034620)	5s rRNA	122	38	211
A. proteobacterium (L13132)	16s rRNA	250	85	891
R. sp. (UNP00394)	16s rRNA	261	155	1343
M anisopliae (3) (AF197120)	Group I Intron	375	120	2004
S. sp. (AF007254)	16s rRNA	400	199	2438
N. subterraneum(U20773)	I intron	573	311	4327

Table 2 Test Sequences, number of helices found in each sequence.

4.2 Perturbation/Mutation Function

The purpose of mutation function is to alter the structures in a controlled and intuitive fashion [14]. Different types of mutations are,

4.2.1 Swap Mutation

In this type of mutation, two random points in a permutation are chosen and the two digits at these positions are interchanged. For example, if we have a permutation of helices such as, p1 = (3,4,2,1,5) and the random points chosen are 1 and 3 then it gives a new permutation as p2 = (3,1,2,4,5)

4.2.2 Percentage mutation operator

In this case the number of mutations is calculated as the product of the percentage of total number of available helices and the current annealing temperature. This operator

is more efficient as it guarantees a different structure after the mutations. Single swap operator may not be effective as the resulting structure again goes through the validation of helices phase.

4.3 Evaluation Function

Gibbs free energy is a measure of energy available in a system to do work. It can be expressed by following equation,

$$\Delta G = \Delta H - T\Delta S \quad (4.3.1)$$

where ΔG is the change in free energy, ΔH is the change in enthalpy, a measure of the heat content of a chemical system, T is temperature in degree Kelvin, and ΔH is the change in entropy, a measure of the disorder in a chemical system[29].

The quality of a structure is determined in terms of free energy. There are several thermodynamic models available to determine the free energy. Individual Nearest Neighbor (INN), Individual Nearest Neighbor with Hydrogen Bonds (INN-HB), efn2, and HotKnots are the thermodynamic models widely used in Bioinformatics applications. For the purpose of this project, we have used efn2 energy model. The basic assumption in this model is that energy contributions by neighboring base pairs are independent and additive [14]. This model has been developed at University of Rochester and the energy parameters are calculated by performing large number of experiments. This model takes into account the INN-HB parameters as well as tandem GU pairs. Tandem GU pairs are two pairs of GU that are located side by side of each other. Also appropriate bonuses or penalties are given to terminal mismatches or dangling ends .

The efn2 model uses a more precise free energy computation that takes into account coaxial stacking and Jacobson-Stockmeyer theory[14] for multi-branched loops. Coaxial stacking is a result of a bend in the axis of helix, because of which helices are stacked on each other[16] . It is found to be associated with large favorable free energy change. The study by Jacobson and Stockmeyer showed that the free energy's dependence on the size of the loop should be logarithmic. The equation used for approximating the multi-branch loop free energy depends on the number of unpaired nucleotides. For less than seven unpaired nucleotides, the equation is,

$$\Delta G_L = a + bn + ch + \Delta G_{stack} \quad (4.3.2)$$

where a,b,c are empirically derived parameters (a =offset, b = base penalty, and c = helix penalty), n is the number of unpaired nucleotides, h is the number of helices in the multi-branch loop. ΔG_{stack} calculates the free energy of stacking interactions [14].
When the number of unpaired nucleotides is more than seven, efn2 uses more realistic parameters for the equation above and recalculates the energy. The new equation in this case is,

$$\Delta G_L = a + 6b + 1.75 * RT * \ln(l_s(L)/6) + c * l_d(L) + \Delta G_{stack} \quad (4.3.3)$$

This is still an active area of research with new and modified parameters being published. The accuracy of secondary structure prediction is supposed to improve as the accuracy of energy parameters improves[14].

4.4 Decision Mechanism

The decision mechanism of simulated annealing algorithm plays an important role as it makes sure that system doesn't get stuck into local minimum. This is achieved by accepting solutions even with higher energy change with some probability. This probability is referred as Acceptance probability. This probability is calculated as a function of energy and temperature. If the change in energy is less than or equal to zero, the solution is accepted. Also the solution for which change in energy is greater than 0 will be accepted with some probability. The acceptance probability in this implementation is calculated by following equation[14],

$$\text{Probability}_{[\text{Accept}]} = e^{-(E_{new} - E_{old})/T} = e^{-\Delta E_{nergy}/T} \quad (4.4.1)$$

This equation models the probability as Boltzmann distribution. The idea behind Boltzmann distribution is that every specific state of system at equilibrium has equal probability. This function will accept low energies most of the times, and sometimes high energies. When temperature is reduced slowly enough, theoretically simulated annealing will give best solution. The tradeoff here is the number of iterations it will take and computational time. In general the convergence of simulated annealing can be represented as in the following figure,

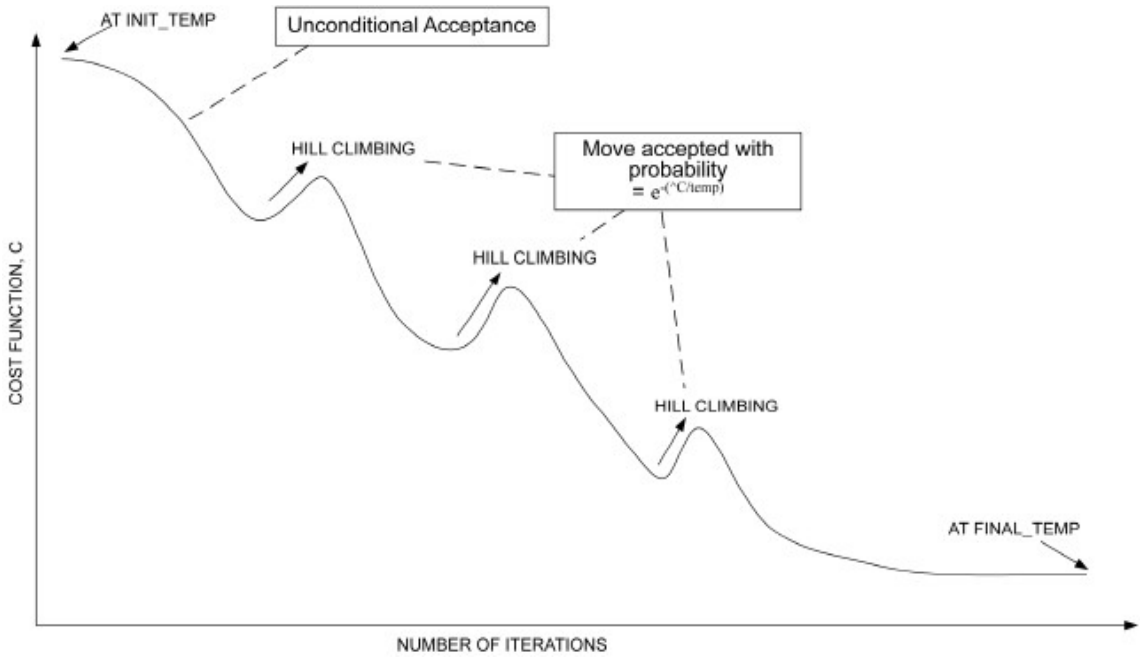

Figure 7 Convergence of simulated annealing algorithm [30].

In the current chapter, we described the main steps in mapping the structure prediction problem into simulated annealing framework. We also described the energy function and the decision mechanism in detail. In the next chapter, we discuss the experiments and results obtained with our implementation of the simulated annealing algorithm.

5 EXPERIMENTS AND RESULTS

In this chapter, we describe the RNA sequences used. We also list out the experiments we performed as well as comparison with other algorithms.

Sequences : We chose a set of 8 sequences with known secondary structure from the Comparative RNA Website[31]. SARNA-Predict has used sequences with maximum lengths up to 1494. But because of lack of computational resources, we kept the maximum length of the sequence to around 500. We have used 5 of the same sequences used in [14] to verify the correct functioning of the algorithm. Although for most of the sequences the data base has been updated with latest results available from the Comparative RNA Website. This set of sequences represent a good cross-section of organisms and the types of RNA.

Organism	RNA Class	Number of Nucleotides	Base pairs in known structure
B. subtilis (D11460)	5s rRNA	118	70
S. cerevisiae (X67579)	5s rRNA	118	37
H. marismortui (AF034620)	5s rRNA	122	38
A. proteobacterium (L13132)	16s rRNA	250	85
R. sp. (UNP00394)	16s rRNA	261	155
M anisopliae (3) (AF197120)	Group I Intron	375	120
S. sp. (AF007254)	16s rRNA	400	199
N. subterraneum(U20773)	Group I intron	573	311

Table 3 Test sequences, number of base pairs in known structures

The performance of the simulated annealing is compared with other state of the art secondary structure folding algorithms. We have used Mfold to do a relative comparison. Mfold is chosen as a representative from the dynamic programming language.The metrics used for evaluation is described below. We also report the results of experiments with various annealing schedules and different sets of permutation parameters.

5.1 Energy flow of Simulated annealing

Figure 8 shows the energy minimization path for a typical run of *H. marismortui* (AF034620) sequence. This sequence has 122 nucleotide bases. The total number of helices found for this particular RNA sequence is 211. As we can see from the plot in figure 8, during the initial phase of the algorithm, energy accepted is fluctuating heavily. This is because the algorithm allows to take steps which don't necessarily decrease the energy. As the number of iterations increase, the probability that the a bad step is accepted decreases and after 1500 iterations the algorithm reaches a stable state.

Figure 9 shows the flow of temperature as the number of iterations increases. The damping factor for the geometric cooling schedule is set at 0.99. The temperature approaches the final value in about 600000 iterations.

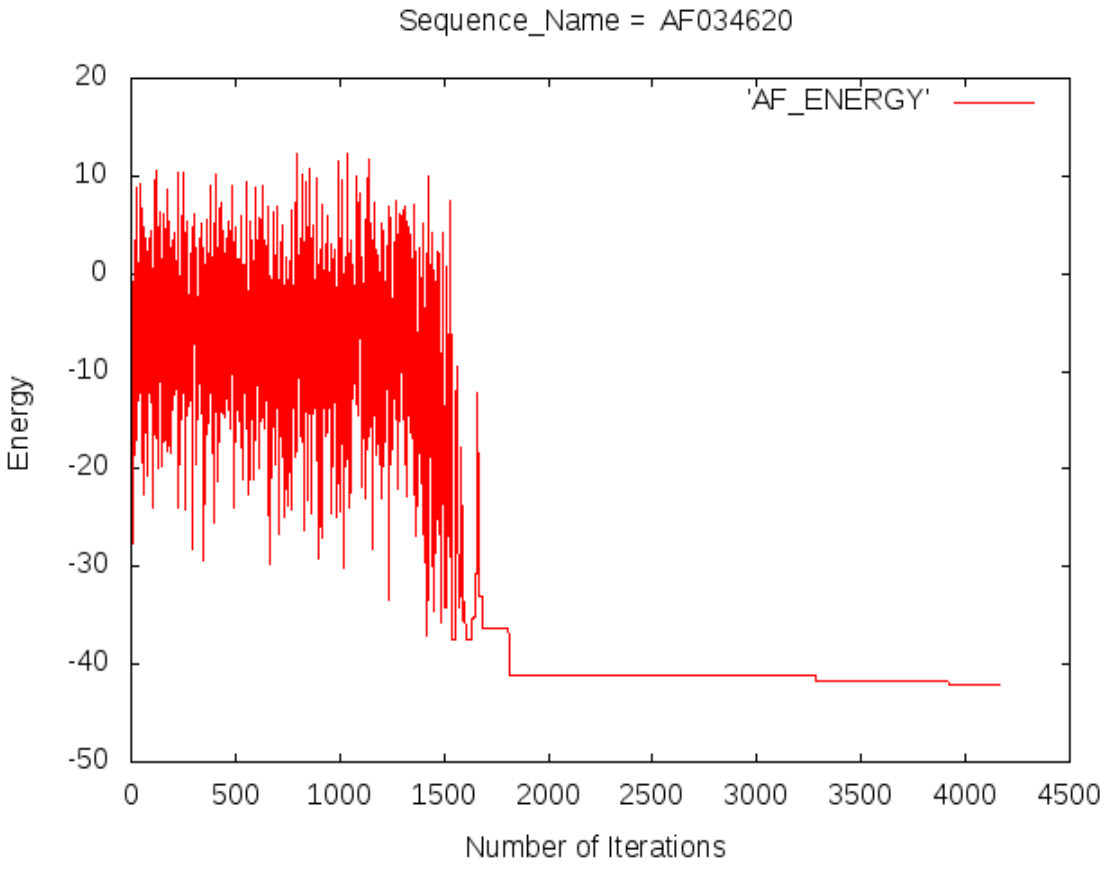

Figure 8 Graph showing the energy convergence behavior for *H. marismortui* (AF034620)

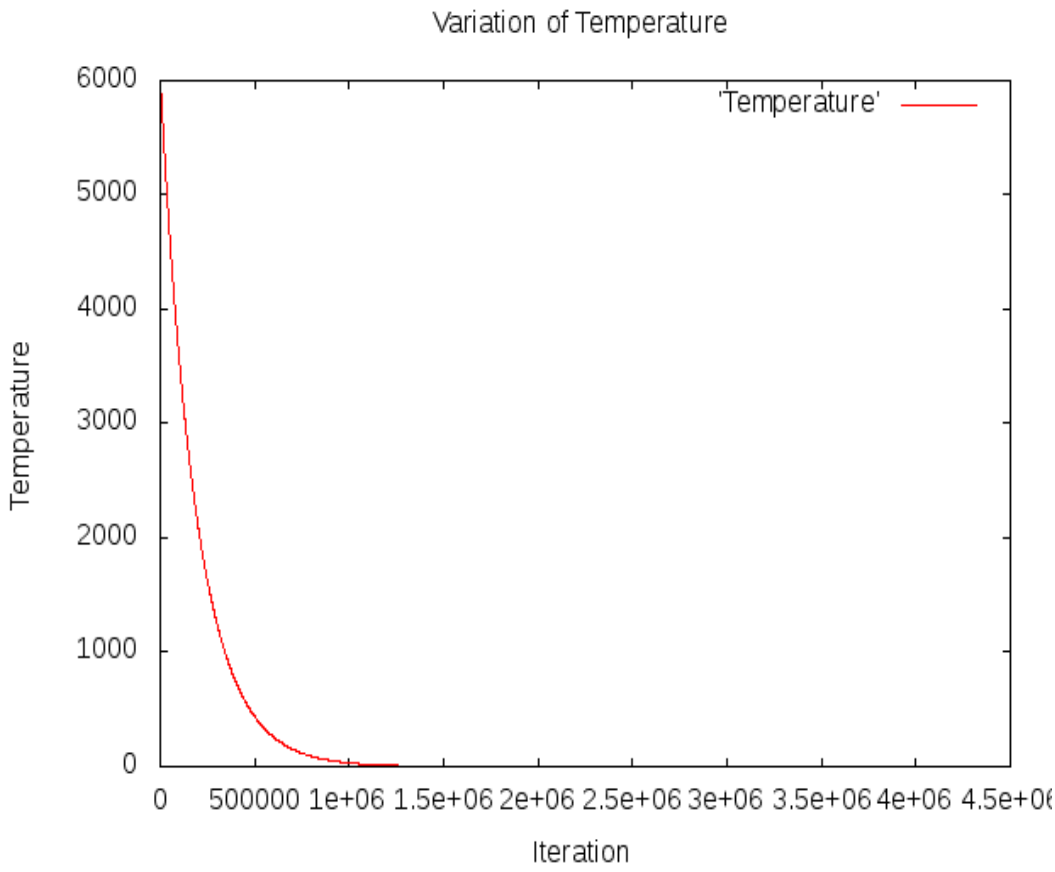

Figure 9 Graph for temperature variation

5.2 Free Energy convergence variation with mutation parameter

The mutation step in the simulated algorithm implements a change in the problem structure in a gradual fashion. This allows a type of control on the direction of the step. By swapping only one element of the helix set, the classical algorithm would allow only minor modification to the final permutation encoding. Most of the time this one step gets lost when the permutation is decoded. This happens when we swap elements in the middle of the set and they are discarded because they conflict with the helices in the beginning of the set. The result is an increase in the number iterations that are needed to converge to a lower free energy of the secondary structure. The increase is more prominent with larger sequences as they produce a very large number helices.

An alternative is to use the modified permutation function developed in [12]. Here a large number of swaps are allowed initially (when the temperature is high) . These large number of swaps enable the algorithm to explore significantly greater area in the problem space. As a result, the algorithm converges to minimum free energy much faster than the classical simulated annealing perturbation function.

Figure 10 shows the secondary structure detected using the classical approach. After 300,000 iterations the free energy achieved is -35.1Kcal/mol. The number of correctly identified base pairs is 68%. Figure 11 shows the secondary structure achieved with modified perturbation function. The swap parameter is set to 0.1. This results into 600 swaps for initial temperature of 6000. In same number of iterations, this run converges to -48.6 Kcal/mol. We observe the same pattern with greater intensity as the number of bases in a sequence increases, i.e. the bigger the sequence, the perturbation function is more effective.

Following table shows the experimental results on two of the sequences with varying lengths.

	Number of Swaps (Initial – Final)	Correct Base Pairs	Minimum Free Energy
Haloarcula marismortui (AF034620)	1200 – 1 600 – 1 300 – 1	74.23% 70.31% 53.67%	-49.3Kcal/mol -48.6Kcal/mol -44.1Kcal/mol
Acanthamoeba grifini (U02540)	1200 – 1 600 – 1 300 – 1	59.40% 58.3% 54.23%	-151.2Kcal/mol -147.5Kcal/mol -141.1Kcal/mol

Table 4 Results of varying mutation parameter and minimum free energy

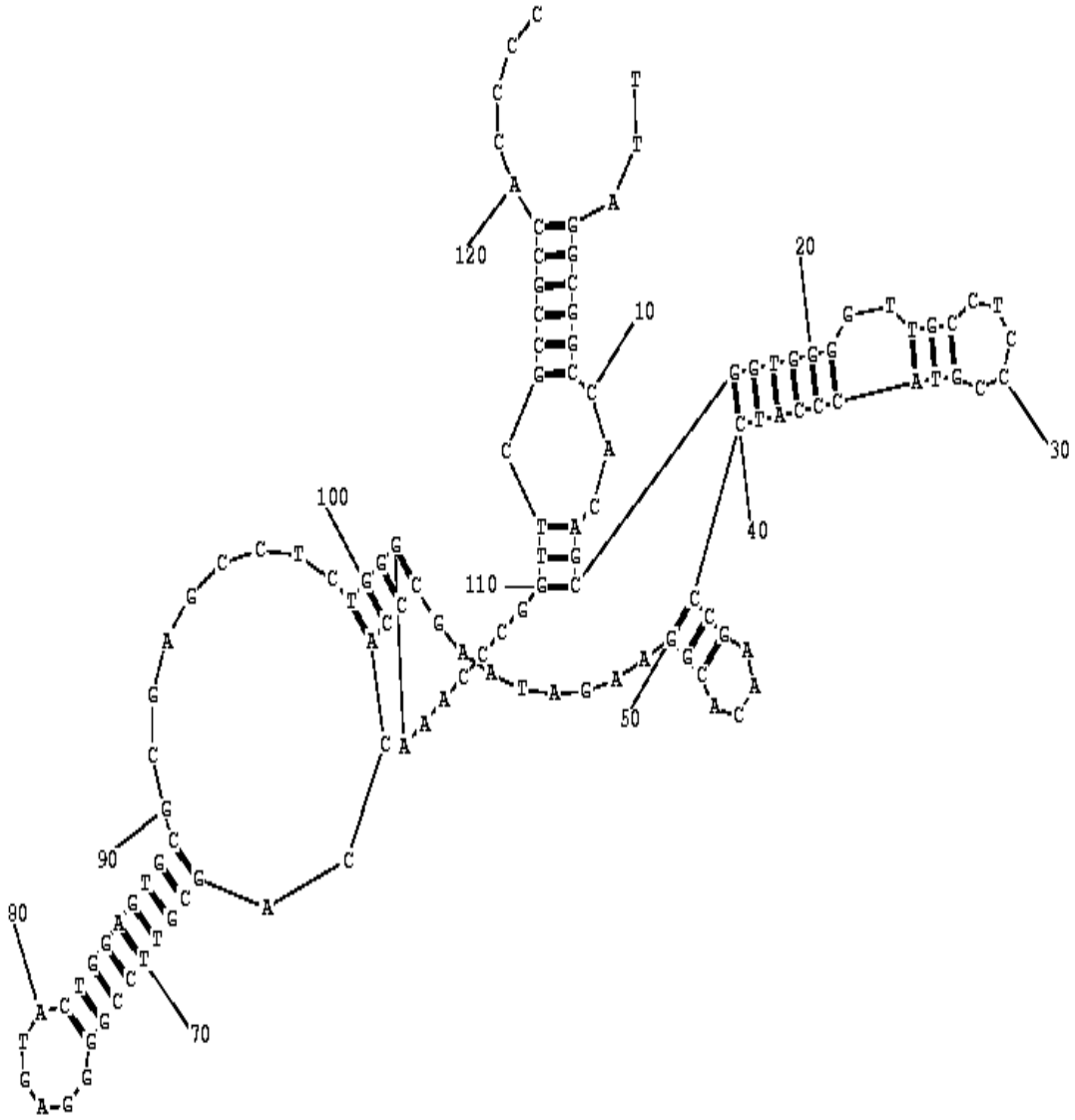

Figure 10 Secondary structure predicted using classical mutation approach for sequence *H. marismortui* (AF034620)

This structure is produced using the draw method from RNAStructure package described in section 3.

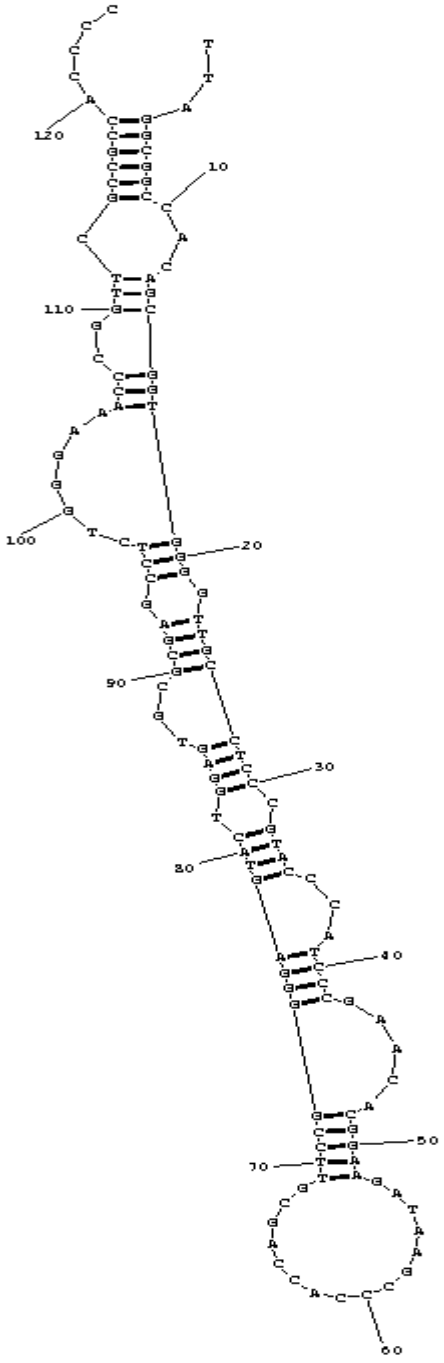

Figure 11 Secondary structure predicted using the modified mutation approach for sequence *H. marismortui* (AF034620). Image produced using draw method from RNAStructure package.

5.3 Results of Simulated Annealing for typical sequences

In this section we describe the results obtained for *A. proteobacterium* (L13132), *S. sp.*(AF007254) and *M. anisopliae* var.(AF197120).

5.3.1 Alpha Proteobacterium 16s RNA (250 bases)

Table 5 shows the results obtained for *A. Proteobacterium*. The results were obtained with using two different sets of temperature decay factors and combinations of varying amounts of swaps. The best results were obtained with decay factor of 0.95 and 1200 initial swaps in the perturbation step. 65.40% of base pairs were identified correctly. Note. even though there are structures which have lower minimum free energy, they don't have better performance in terms of correctly identified base pairs. The results became better with increasing number of swaps. This could be attributed to the fact that Alpha Proteobacterium is a relatively small sequence. Thus a large number of swaps allow a greater proportion of helix structures to formed and explored.

Figure 12 shows the structure obtained with 65.4% correct base pairs whereas the Figure 13 shows the energy convergence flow over this particular run. This particular energy plot exemplifies the principle of simulated annealing. It shows that during initial phase of the algorithm a conformation of helix stacks is formed whose free energy is couple of order of magnitude more than the best achievable energy. But the algorithm picks this as the possible structure and thus jumps away from the local minima. The algorithm then stabilizes around -73.2 Kcal/mol.

Temperature Decay Factor	Number of Swaps (Initial – Final)	Correct Base Pairs	Minimum Free Energy
0.93	1200 – 1 600 – 1 300 – 1	65.12% 62.64% 61.67%	-74.3Kcal/mol -76.6Kcal/mol -70.1Kcal/mol
0.95	1200 – 1 600 – 1 300 – 1	65.40% 63.3% 61.23%	-73.2Kcal/mol -75.5Kcal/mol -71.6Kcal/mol

Table 5 Results of variation of temperature decay factor and minimum free energy

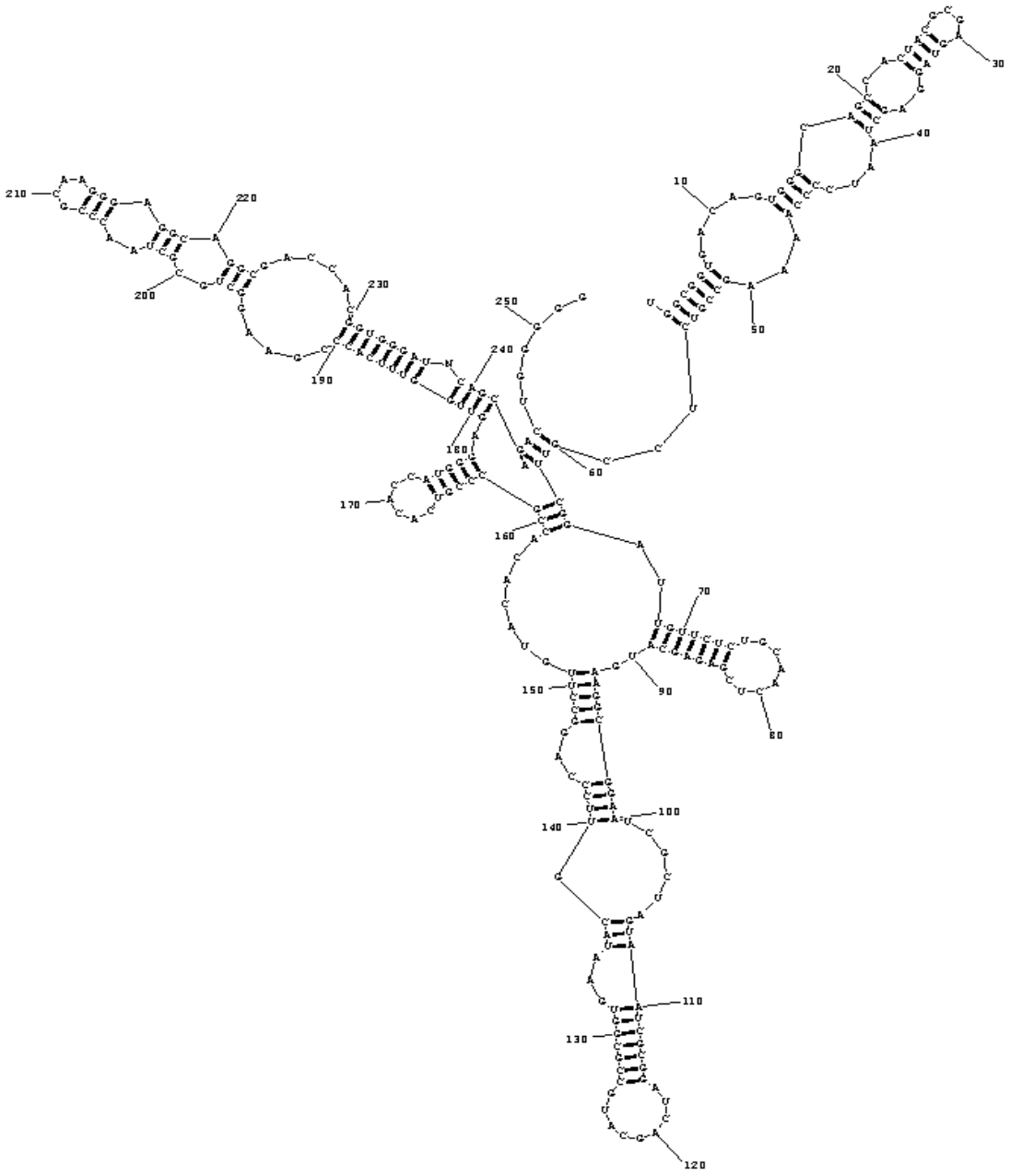

Figure 12 Secondary Structure of *A.proteobacterium* 16s RNA produced using draw method provided by RNAStructure package

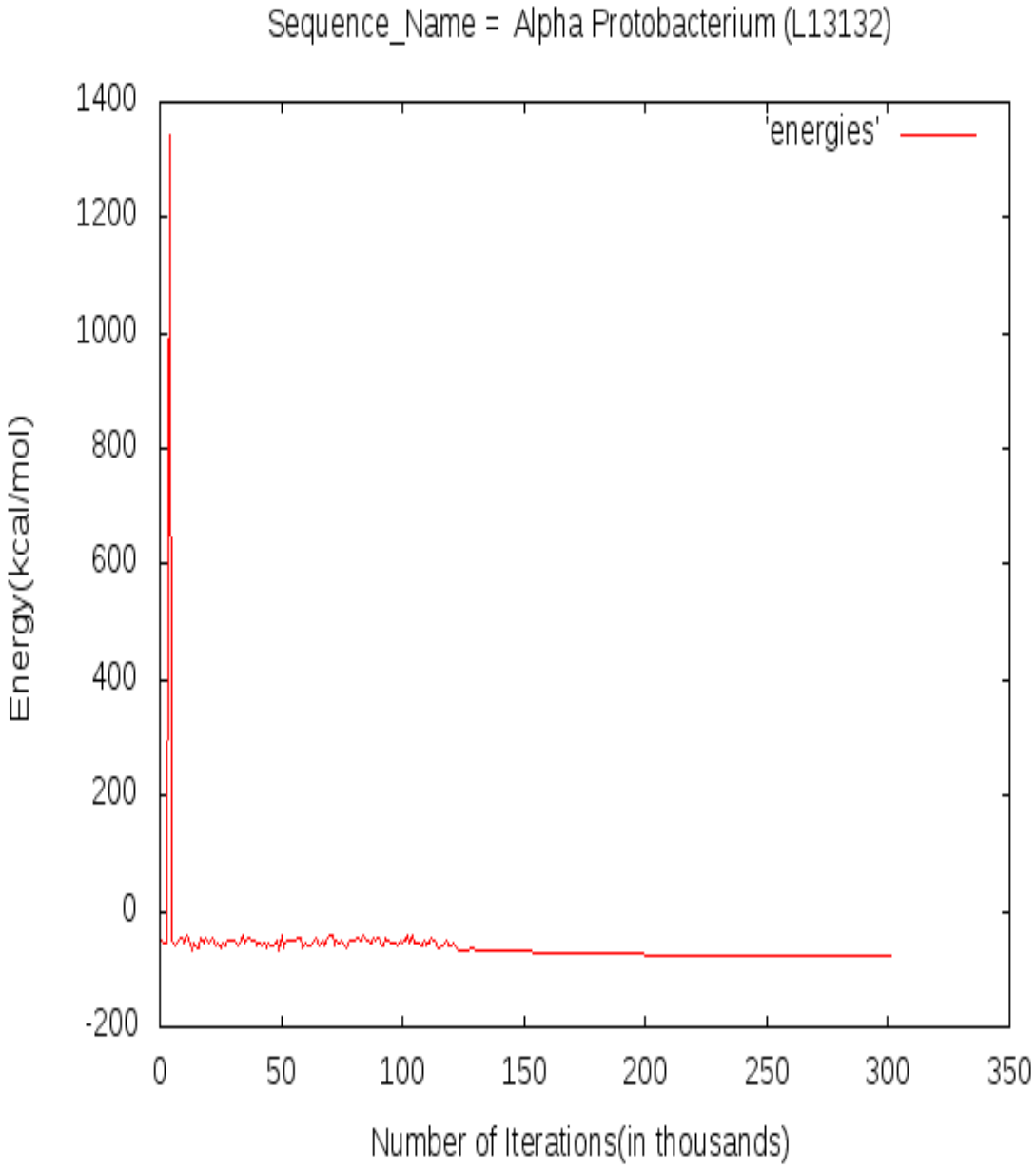

Figure 13 Energy plot for *A.proteobacterium*. The algorithm allows a structure with very large free energy (1341Kcal/mol) and thus does not get stuck in local minima. Algorithm then converges to -73.2Kcal/mol.

5.3.2 Sulfitobacter sp. 16S RNA (AF007254) with 400 nucleotide bases

Table 6 shows the results obtained after running 40 random seeds for each of the combination of temperature decay and initial number of swaps. Compared to *A. proteobacterium* whose results are described previously, this particular sequence is larger and results into 2438 different helices. This warranted a jump in the number of swaps as possible number of configurations became exorbitantly large. The runs show that we large initial swaps, the algorithm is allowed to explore a rather large problem space. The algorithm showed best results at -131.23 Kcal/mol with 57.34% correct base pairs. matches. Figure 14 shows the secondary structure obtained for this particular RNA sequence.

Temperature Decay Factor	Number of Swaps (Initial – Final)	Correct Base Pairs	Minimum Free Energy
0.93	2400 – 1 1200 – 1 600 – 1	57.12% 54.64% 51.67%	-131.3Kcal/mol -124.5Kcal/mol -121.7Kcal/mol
0.95	2400 – 1 1200 – 1 600 – 1	55.82% 54.21% 50.23%	-130.4Kcal/mol -129.1Kcal/mol -122.6Kcal/mol
0.97	2400 – 1 1200 – 1 600 – 1	57.40% 55.23% 51.38%	-131.3Kcal/mol -128.5Kcal/mol -123.6Kcal/mol

Table 6 Variation of temperature, mutation for *S. sp*. 16S RNA (AF007254)

5.3.3 Bacillus subtilis (D11460) with 118 nucleotide bases

Table 7 shows the results with 40 random seeds for the *B. subtilis* sequence. This is a relatively small sequence and the algorithm found 83.12% of the correct base pairs. There is no significant variation in correct base pairs detected when the swap number for the permutation step is changed. The restriction of defining a helix with at least 3 pairs of nucleotides stops the algorithm from correctly predicting intermediate base pairs. The minimum free energy achieved in this case is -37.2 Kcal/mol. Figure 15 shows the secondary structure obtained for *B. subtilis* sequence.

Temperature Decay Factor	Number of Swaps (Initial – Final)	Correct Base Pairs	Minimum Free Energy
0.95	600 – 1 300 – 1 150 – 1	83.12% 82.64% 84.67%	-37.2Kcal/mol -36.5Kcal/mol -37.8Kcal/mol

Table 7 Variation of temperature, mutation for *B.subtilis* (D11460)

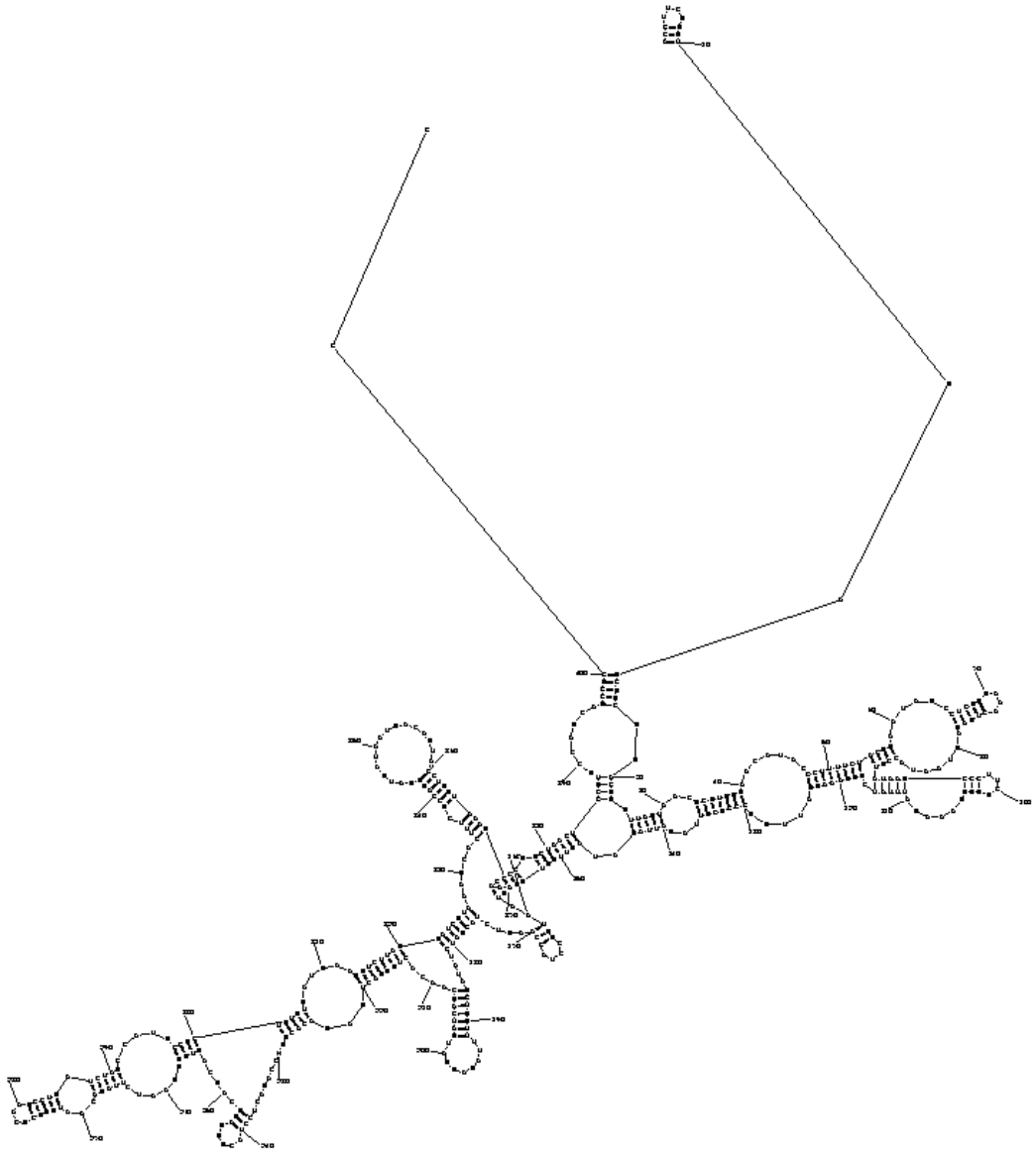

Figure 14 Secondary structure of *S. sp*. 16S RNA (AF007254) produced using draw method provided by RNAStructure package.

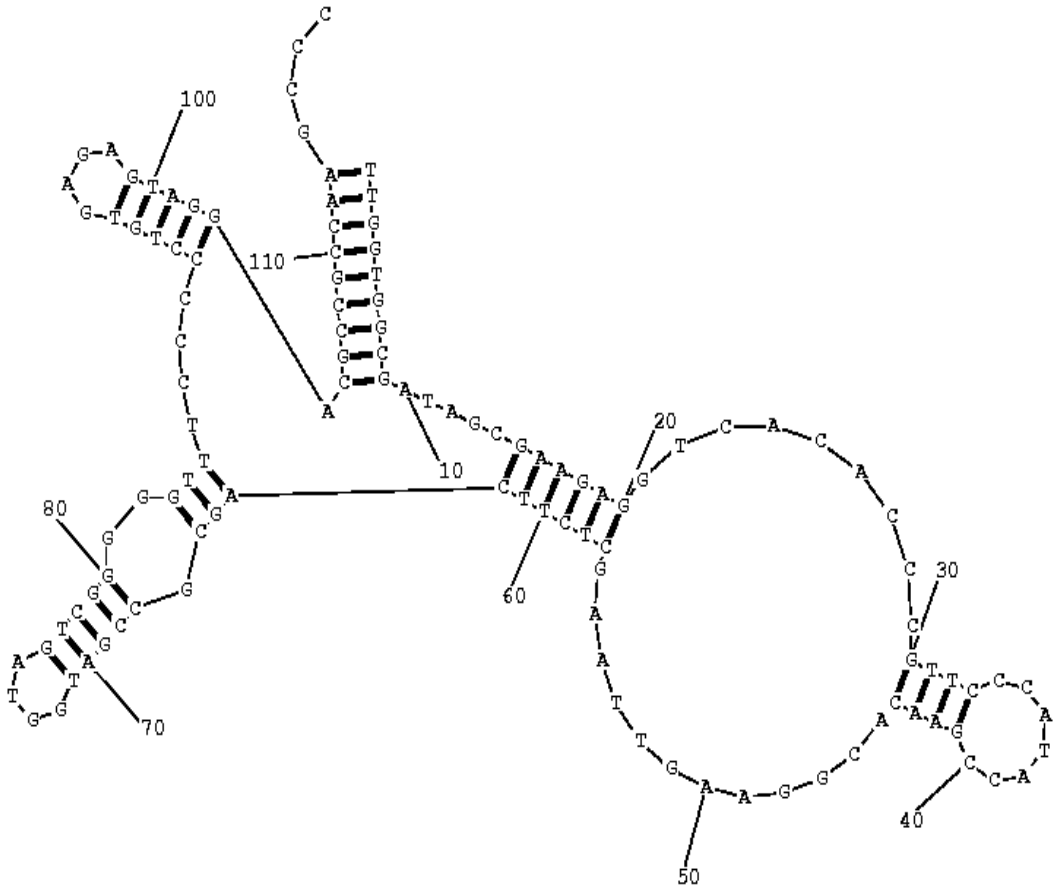

Figure 15 Secondary structure of *B. subtilis* (D11460) produced using draw method provided by RNAStructure package.

5.3.4 Secondary structure for *S. cerevisiae* (X67579)

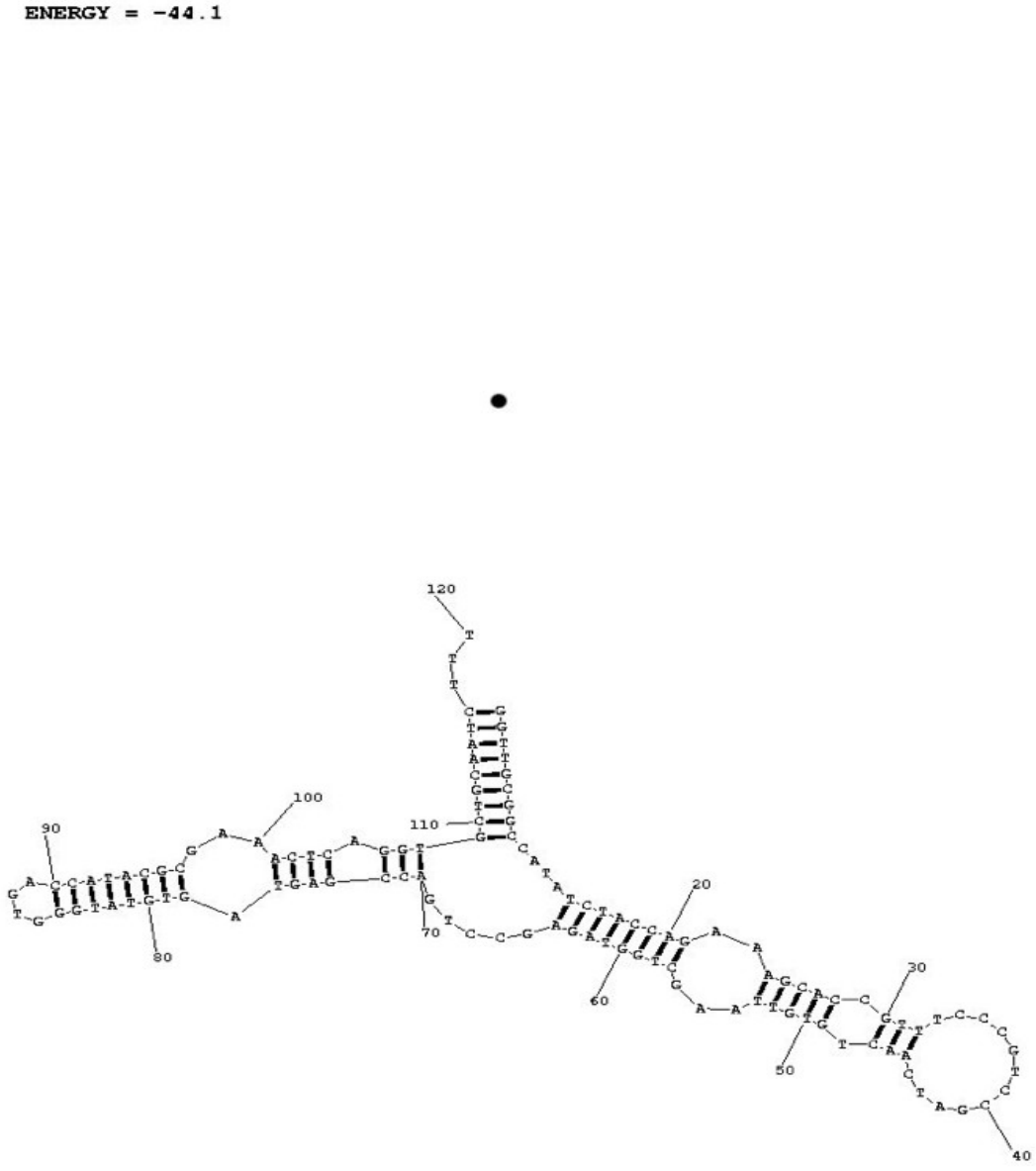

Figure 16 Secondary structure for *S.cerevisiae* produced using draw method provided by RNAStructure package

5.4 Comparison with Mfold

Michael Zuker developed Mfold algorithm to predict the secondary structure. This is a dynamic programming algorithm which also produces sub-optimal structures as result. A web implementation has been made available at http://mfold.rna.albany.edu. The version of Mfold that was used is 3.2. To compare the results with simulated annealing, we chose the structure which gave the lowest free energy. The tests were run at 37 degree Celsius setting with other attributes in the form set to default.

We found comparable and in some cases better results than Mfold. Table 8 describes the comparison.

Organism	RNA Class	Number of Nucleotides	Base pairs in known structure	Simulated Annealing (%BP)	Mfold (% BP)
B. subtilis	5s rRNA	118	70	83.17	84.00%
S. cerevisiae (X67579)	5s rRNA	118	37	89	89
H. marismortui (AF034620)	5s rRNA	122	38	74.23	71.45
A. proteobacterium (L13132)	16s rRNA	250	85	81.2	79.3
R. sp. (UNP00394)	16s rRNA	261	155	65.23	62.34
M. anisopliae var. (3) (AF197120)	Group I Intron 23S rRNA	375	120	77.89	76.66
S. sp. (AF007254)	16s rRNA	400	199	54.77	59.63
N. subterraneum (U20773)	I intron	573	311	52.54	53.12

Table 8 Comparison with Mfold

6 CONCLUSION

We studied one of the heuristic approaches to solve the problem of RNA secondary structure prediction, that is, simulated annealing. The algorithm was implemented using C++ and tested on a quad-core AMD machine. The method of mapping the secondary structure determination into simulated annealing framework is realized. Various experiments in helix generation and perturbation functions were performed in order to expedite the convergence of the algorithm. We compared the results obtained from simulated annealing with Mfold. We found the results similar for shorter sequences in general and in some cases a little better as the length of the sequence increased. The experiments were restricted to RNA sequences with length shorter than 600 nucleotides because of lack of computational resources.

7 FUTURE WORK

We suggest multiple directions in which this work can be extended.

1. The current implementation does not work well if there are pseudo-knots in the native structure. This could be done with improved thermodynamic model which takes into account the effect of pseudo-knots. The Hotknots software implements one such thermodynamic model which could be integrated into current energy calculating function.

2. The algorithm could be improved with weighted helix set approach. During the repair of the helix set, we can choose helices within a defined range of stacked pair with greater probability. This would allow more stable structures to be formed more easily.

3. There exists significant opportunity to optimize the code-base. The focus for the implementation was correctness of the algorithm rather than optimality. Especially the memory intensive data movement could be reduced to increase the efficiency of the program.

4. Also, the code could be mapped to a parallel processing architecture. This way the runtime could be significantly reduced by taking advantage of the efficiency of parallel machines.

8 REFERENCES

[1] Setubal, J. & Meidanis, J. (1997). *Introduction to Computational Molecular Biology.* Boston, MA: PWS Publishing Company

[2] Al-Khatib, R., Abdullah, R., Rashid, N. (2010) *A Comparative Taxonomy of Parallel Algorithms for RNA Secondary Structure Prediction.*Evolutionary Bioinformatics, 6, 27-45.

[3] Draper D. E. (2004). *A guide to ions and RNA structure*. RNA, 10(3), 335-443

[4] I. Tinoco Jr., C. Bustamante. (1999). *How RNA folds.* J. Molecular Biology, 293, 271-281.

[5] Alberts B., Johnson A. , Lewis. J, et al. (2002). *Molecular biology of the cell*. New York: Garland Science.

[6] Nussinov, R., Jacobson AB. (1980) *Fast algorithm for predicting the secondary structure of single-stranded RNA.*Proc Natl Acad Sci U S A. 77(11) , 6309-13.

[7] Ye, Y. *RNA folding and ncRNA finding* [PowerPoint slides]. Retrieved from http://mendel.informatics.indiana.edu/~yye/lab/teaching/fall2010-I519.php

[8] *RNA structure determination Experimental techniques & Computational prediction* [PDF document]. Retrieved from http://www.ibi.vu.nl/teaching/masters/prot_struc/2008/ps-lec12-2008.pdf

[9] *Gibbs Free Energy* Retrieved from http://en.wikipedia.org/wiki/Gibbs_free_energy

[10] Zuker,M. (2003) *Mfold web server for nucleic acid folding and hybridization prediction.* Nucleic Acids Res. 31 (13), 3406-3415.

[11] Wiese, K.C., Deschenes, A.A., Hendriks, A.G.,(2008). *RnaPredict—An Evolutionary Algorithm for RNA Secondary Structure Prediction.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 5, 25-41,2008.

[12] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P. (1983). *Optimization by Simulated Annealing.* Science, 220(4598), 671-680.

[13] Dueck, G., Scheuer,T. (1990). *Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing.* Journal of Computational Physics, 90 (1),161-175.

[14] Tsang, H.H., (2007) *SARNA-Predict: A Permutation-based Simulated Annealing Algorithm for RNA Secondary Structure Prediction* [PDF document]. Retrieved from http://ir.lib.sfu.ca/bitstream/1892/9613/1/etd3184.pdf

[15] *Simulated Annealing* (n.d.) Retrieved from http://en.wikipedia.org/wiki/Simulated_annealing

[16] Tsang,H.H., and Wiese, K.C. (2006). *SARNA-Predict: A simulated annealing algorithm for RNA secondary structure prediction.* IEEE Symposium on Computational Intelligence in Bioinfomatics and Computational Biology. CIBCB'O6, 466-475.

[17] Tsang, H.H.; Wiese, K.C. (2010). *SARNA-Predict: Accuracy Improvement of RNA Secondary Structure Prediction Using Permutation-Based Simulated Annealing.* Computational Biology and Bioinformatics*,* IEEE/ACM Transaction*s* , 7, 727-740.

[18] Tsang, H.H.; Wiese, K.C. (2008). *SARNA-Predict-pk: Predicting RNA secondary structures including pseudoknots*. Computational Intelligence in Bioinformatics and Computational Biology. *CIBCB '08. IEEE Symposium on* , vol., no., pp.1-8, 15-17.

[19] Yinghao, Li. (1997). *Directed Annealing Search In Constraint Satisfaction and Optimization*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, London.

[20] Emile H. L. Aarts, F. M. J. de Bont, J. H. A. Habers, and Peter J. M. van Laarhoven. (1986). *Parallel implementations of the statistical cooling algorithm.* Integration, the VLSI Journal, 4(3):209-238.

[20] Gruener, W., Giegerich, R., Strothmann,D., et al. (1996). *Analysis of RNA sequence structure maps by exhaustive enumeration structures of neutral networks and shape space covering*. Monath. Chem., 127:375-389, SF1 preprint 95-10-099.

[21] Trosset, M.W. (2001). *What is simulated annealing?* Optimization and Engineering, 2(2):201- 213.

[22] Robic,B., and Silc, J., (1995). *Algorithm mapping with parallel simulated annealing.* Technical Report CSD-TR-95-1, Jozef Stefan Institute, Computer Science Department.

[23] M. Irgens. (1997). *Why simulated annealing works and why it doesn't*. T R 97-17, Department of Computer Science, Simon Fraser University.

[24] *GSL- GNU Scientific Library* Retrieved from, http://www.gnu.org/software/gsl/

[25] boost C++ Libraries Retrieved from, http://www.boost.org/doc/libs/1_46_1/libs/bind/bind.html#Purpose

[26] RNAStructure, Version 5.2 Retrieved from http://rna.urmc.rochester.edu/RNAstructure.html

[27] Wiese, K.C and Hendriks, A., (2006) *Comparison of P-RnaPredict and mfold—algorithms for RNA secondary structure prediction* Bioinformatics (2006) 22(8): 934-942 doi:10.1093/bioinformatics/btl043

[28] Wiese, K.C and Glen, E., (2003) *A permutation based genetic algorithm for the RNA folding problem: a critical look at selection strategies, crossover operators, and representation issues*. 72:49-41 BioSystems - Special Issue on Computational Intelligence in Bioinformatics.

[29] Wiese, K.C and Hendriks, A., *Analysis of Thermodynamic Models and Performance in RnaPredict - An Evolutionary Algorithm for RNA Folding.* IEEE Xplore.

[30] Akella. P. *Simulated Annealing* [PowerPoint slides]. Retrieved from www.ecs.umass.edu/ece/labs/vlsicad/.../SimulatedAnnealing.ppt

[31] *Comparative RNA Website And Project* Retrieved from, http://www.rna.ccbb.utexas.edu/

[32] Flechsig, M., Bohm, U., Nocke,T., et al. *The Multi-Run Simulation Environment SimEnv* Retrieved from, http://www.pik-potsdam.de/research/research-domains/transdisciplinary-concepts-and-methods/modsimenv/simenv/simenv124.pdf

9 Appendix

Here we describe the source code.

```
**************************************************************************
File : base_class.h

/*
  Defines the data types used by the final implementation.
*/
**************************************************************************
 #if !defined(BASE_CLASS_H)
#define BASE_CLASS_H

#include <vector>
#include <string>

/ * The most basic data type is nucleotide with location and base_type */

class nucleotide_base
{
private:
      int base_location;
      string base_type;
      int helix_id;
public:
      nucleotide_base(string s, int i);
      int    get_base_location(void) const;
      string get_base_type();
};

/* Define the base pair of rna. */

class base_pair
{

private:
         int associated_with_helix;
      nucleotide_base i_loc,j_loc;

public:
      base_pair(string s_i,string s_j,int i, int j);
      ~base_pair();
```

```
        bool is_partof_other_helix();
        bool associate_with_helix(int i);
        bool is_canonical_base_pair();
        int  get_i_loc();
        int  get_j_loc();
        string  get_i_base();
        string  get_j_base();
};

/* Define helix structure along with suitable access function. */
class helix
{

private:
        bool valid,complete;
        int  helix_id;
public:
           vector<base_pair> bases_in_helix;
        vector<int> stack_bases;
        vector<int> loop_bases;
        vector<nucleotide_base> loop;
        helix();
        ~helix();
        bool is_valid();
        bool is_complete();
        bool add_to_loop(const nucleotide_base& b);
        void add_base_pair(base_pair& bp);
        void set_valid(bool v);
        void set_complete(bool c);
        int  get_num_bases();
        int  get_loop_size();
        int  get_helix_id();
        void set_helix_id(int id);
        void display_helix();
        bool intersects_with( helix& h);
};

#endif
```

```
**********************************************************************
File : base_class.cpp

/*
  Defines the class implementation of data types used by the final implementation.
*/
**********************************************************************
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <algorithm>
#include <string>
#include <set>

using namespace std;

#include "base_class.h"

//-----------------------------------------------------------------------------------------------
// Nucleotide base class
//-----------------------------------------------------------------------------------------------
nucleotide_base::nucleotide_base(string s,int i)
{
   base_location =  i;
   base_type     =  s;
   helix_id      =  -1;
}


int nucleotide_base::get_base_location()const
{
  if (base_location < 0){
     cerr << "Base location not set yet." << endl;
  }
  else{
     return base_location;
  }
}
```

```
string nucleotide_base::get_base_type()
{
 return base_type;
}

//-----------------------------------------------------------------------------------------------
// Nucleotide base pair class
//-----------------------------------------------------------------------------------------------
base_pair::base_pair(string s_i, string s_j, int i, int j):
        i_loc(s_i,i),
        j_loc(s_j,j)
{
   associated_with_helix = -1;
}

base_pair::~base_pair()
{

}


bool base_pair::is_partof_other_helix()
{
   return (associated_with_helix != -1);
}

bool base_pair::associate_with_helix(int i)
{
   associated_with_helix = i;
}

bool base_pair::is_canonical_base_pair()
{
  bool val = false;
  string i_t = i_loc.get_base_type();
  string j_t = j_loc.get_base_type();
  std::transform(i_t.begin(), i_t.end(),i_t.begin(), ::toupper );
  std::transform(j_t.begin(), j_t.end(),j_t.begin(), ::toupper );

  if(
    (
      !strcmp ( i_t.c_str(), "A" ) &&
      !strcmp ( j_t.c_str(), "U" )
```

```
)
||
(
  !strcmp ( i_t.c_str(), "U" ) &&
  !strcmp ( j_t.c_str(), "A" )
)
||
(
  !strcmp ( i_t.c_str(), "A" ) &&
  !strcmp ( j_t.c_str(), "T" )
)
||
(
  !strcmp ( i_t.c_str(), "T" ) &&
  !strcmp ( j_t.c_str(), "A" )
)
||
(
  !strcmp ( i_t.c_str(), "G" ) &&
  !strcmp ( j_t.c_str(), "C" )
)
||
(
  !strcmp ( i_t.c_str(), "C" ) &&
  !strcmp ( j_t.c_str(), "G" )
)
||
(
  !strcmp ( i_t.c_str(), "G" ) &&
  !strcmp ( j_t.c_str(), "U" )
)
||
(
  !strcmp ( i_t.c_str(), "U" ) &&
  !strcmp ( j_t.c_str(), "G" )
)
||
(
  !strcmp ( i_t.c_str(), "G" ) &&
  !strcmp ( j_t.c_str(), "T" )
)
||
(
```

```
     !strcmp ( i_t.c_str(), "T" ) &&
       !strcmp ( j_t.c_str(), "G" )
      )

    )
    {
      val = true;
    }

   //cout << " I = " << i_t << " i = " << i_loc.get_base_location() << "||| J = " << j_t << " j
= " << j_loc.get_base_location() << " || " << val << endl;

   return val;
}

int base_pair::get_i_loc()
{
  return i_loc.get_base_location();
}

int base_pair::get_j_loc()
{
  return j_loc.get_base_location();
}

string base_pair::get_i_base()
{
  return i_loc.get_base_type();
}


string base_pair::get_j_base()
{
  return j_loc.get_base_type();
}

//-----------------------------------------------------------------------------------------------
// Helix  class
//-----------------------------------------------------------------------------------------------
helix::helix()
{
    valid    = true;
```

```
   complete = false;

}

helix::~helix()
{

}

void helix::add_base_pair(base_pair& bp)
{
 bases_in_helix.push_back(bp);
 stack_bases.push_back(bp.get_i_loc());
 stack_bases.push_back(bp.get_j_loc());
}


void helix::set_valid(bool v)
{
 valid = v;
}


void helix::set_complete(bool c)
{
 complete = c;
}

bool helix::is_valid()
{
 return valid ;
}


bool helix::is_complete()
{
 return complete;
}

int helix::get_num_bases()
{
 return bases_in_helix.size();
```

```
}

bool helix::add_to_loop(const nucleotide_base& b)
{
 loop.push_back(b);
 loop_bases.push_back(b.get_base_location());
}

int helix::get_loop_size()
{
 return loop.size();
}

void helix::set_helix_id(int id)
{
 helix_id = id;
}


int helix::get_helix_id()
{
 return helix_id;
}

void helix::display_helix()
{
 vector<base_pair>:: iterator h_i = bases_in_helix.begin();
 while(h_i != bases_in_helix.end()){
    cout << "         " <<  h_i->get_i_base()  <<  "    " << h_i->get_i_loc() + 1 <<
"--------------------" << h_i->get_j_loc() + 1 << "    " << h_i->get_j_base() << endl;
    h_i++;
 }

}

bool helix::intersects_with( helix& h)
{


 sort(h.stack_bases.begin(),h.stack_bases.end());
 //cout << " H = " ;
 //for(int i = 0; i < h.stack_bases.size() ; i++)
```

```
  sort(this->stack_bases.begin(),this->stack_bases.end());

  bool stacked_pair = (fabs( this->stack_bases.back() - h.stack_bases.front()) < 5) |
                      (fabs( this->stack_bases.front() - h.stack_bases.back()) < 5);

  //cout << " S = " ;
  //for(int i = 0; i < this->stack_bases.size() ; i++)
  //    cout << " " << this->stack_bases.at(i) ;
  //cout << endl;
  vector<int> z(1000);
  vector<int>::iterator z_i = z.begin();
  vector<int> y(1000);
  vector<int>::iterator y_i = y.begin();
  vector<int> t(1000);
  vector<int>::iterator t_i = t.begin();
  vector<int>::iterator r;
  vector<int>::iterator s;
  vector<int>::iterator i;

  r = set_intersection(h.stack_bases.begin(),h.stack_bases.end(),this-
>stack_bases.begin(),this->stack_bases.end(),z_i);

  sort(this->loop_bases.begin(),this->loop_bases.end());
  s = set_intersection(h.stack_bases.begin(),h.stack_bases.end(),this-
>loop_bases.begin(),this->loop_bases.end(),y_i);

  sort(h.loop_bases.begin(),h.loop_bases.end());
  i = set_intersection(h.loop_bases.begin(),h.loop_bases.end(),this-
>stack_bases.begin(),this->stack_bases.end(),t_i);
   return ( ( int(r - z_i) > 0 ) || ( (int(s - y_i) > 0) && ( int (i-t_i)>0) ));

}
```

```
**********************************************************************
File : test_base.cpp

/*
  Defines the main algorithm helix generation and simulated annealing.
*/
**********************************************************************
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <cstring>
#include <algorithm>
#include <boost/bind.hpp>
#include <boost/lexical_cast.hpp>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_math.h>
#include <assert.h>
#include <gsl/gsl_machine.h>
#include <gsl/gsl_siman.h>
#include <gsl/gsl_ieee_utils.h>
#include "RNA.h"
#include "ErrorChecker.h"

using namespace std;

#include "base_class.h"


/* set up parameters for this simulated annealing run */

/* how many points do we try before stepping */
#define N_TRIES 5

/* how many iterations for each T? */
#define ITERS_FIXED_T 1000

/* max step size in random walk */
#define STEP_SIZE 1.0

/* Boltzmann constant */
#define K 1.0
```

```
/* initial temperature */
#define T_INITIAL 6000

/* damping factor for temperature */
#define MU_T 1.075
#define T_MIN 2.0e-6

/* Number of Stacked bases */
#define STACK_BASE 3
/* Number of bases in loop */
#define NUM_LOOP 3
#define MAX_LOOP 4000
/* Number of swaps per perturbation step */
#define SWAPS 1
#define GUIDE_NUM 3

gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,K,
T_INITIAL, MU_T, T_MIN};

struct data_type_Helix{
  float curr_temp;
  RNA* a;
  vector<helix>* Helix_Set;
  vector<helix>* Final_Set;
  vector<helix>* Guide_Set;
  int seed;
};


/* Generate the helix set using the helix-generation algorithm */

void generate_helix_set(vector<base_pair>& all_bps,vector<helix>& Helix_Set,int
mode, vector<nucleotide_base>& all_bases )
{
   vector<base_pair>::iterator locator_bp ;
   vector<base_pair>::iterator abp = all_bps.begin();
   vector<nucleotide_base>::const_iterator nb;

   int helix_id = 0;
   base_pair dummy("X","X",0,0);
   int bp_n = 0;
   while(abp != all_bps.end())
   {
```

```
   if (abs(abp->get_i_loc() - abp->get_j_loc()) < 9){
       abp++;
       continue;
     }

     if (bp_n % 1000 == 0)
        cout << " Bp = " << bp_n << endl;
     bp_n++;
     helix h;
     int i,j;
     base_pair bp = *abp;
     while ( h.is_valid() && !h.is_complete() )
     {
      //cout << "Begin BP " << endl;
      if(bp.is_canonical_base_pair() && !bp.is_partof_other_helix())
      {
       h.add_base_pair(bp);
       i = bp.get_i_loc() + 1 ;
       j = bp.get_j_loc() - 1 ;
       if(i >= j)
       {
        bp = dummy ;
       }
       else
       {
         locator_bp = find_if(all_bps.begin(),all_bps.end(),
(boost::bind(&base_pair::get_i_loc,_1) == i) && (boost::bind(&base_pair::get_j_loc,_1)
== j ) );
         if(  (locator_bp == all_bps.end()) && (mode == 0) )
           cerr << "ERROR : Can not locate base for ri = " << i << " rj = " <<  j << endl;
         else if( mode == 0 )
           bp = *locator_bp;
         else if( (locator_bp == all_bps.end()) && ( mode == 1 ) )
           bp = dummy;
         else
           bp = *locator_bp;
         //cout << " Next bp i = " << bp.get_i_loc() << " j = " << bp.get_j_loc() << endl;
       }
      }
      else{

        if (h.get_num_bases() < STACK_BASE){
```

```
          h.set_valid(false);
          //cout << " STACK_BASE INVALID " << endl;
        }
        else if ( ( (j-i) < NUM_LOOP) || ( (j-i) > MAX_LOOP ) ){
          h.set_valid(false);
          //cout << " NUM_LOOP INVALID " << endl;
        }
        else{
          for (int x = i ; x < j ; x++)
          {
            nb = find_if(all_bases.begin(),all_bases.end(),
(boost::bind(&nucleotide_base::get_base_location,_1) == x)) ;
            if(nb == all_bases.end())
              cerr << "ERROR : Cannot find all bases" << endl;
            h.add_to_loop(*nb);
          }
          h.set_complete(true);
        }
      }
    }

    if (h.is_valid())
    {
      vector<base_pair>::iterator i = h.bases_in_helix.begin();
      h.set_helix_id(helix_id);
      while(i != h.bases_in_helix.end())
      {
        locator_bp = find_if(all_bps.begin(),all_bps.end(),
(boost::bind(&base_pair::get_i_loc,_1) == i->get_i_loc()) &&
(boost::bind(&base_pair::get_j_loc,_1) == i->get_j_loc() ) );
        if(locator_bp == all_bps.end())
          cerr << "ERROR: h_is_valid not located" << endl;
        else
          locator_bp->associate_with_helix(helix_id);

        i++;
      }
      Helix_Set.push_back(h);
      helix_id++;
      if(helix_id % 100 == 0)
       cout << " Found valid = " << helix_id  << endl;
    }
```

```
      abp++;


    }

    int sizeofhelix;
    sizeofhelix = Helix_Set.size();
    cout << "Number of helices found is  " << sizeofhelix << endl;
    vector<helix>::iterator h_i = Helix_Set.begin();
    while(h_i != Helix_Set.end())
    {
      h_i->display_helix();
      cout <<
"-----------------------------------------------------------------------------------------------------------
------" << endl;
      h_i++;
    }
    cout << "DONE : Number of helices found is  " << sizeofhelix << endl;

}


/* Repair the permuation of helix set by discarding any helixes that conflict with earlier
used helixes. */

void repair_permuatation(vector<helix>& Helix_Set,vector<helix>& Uniq_Set)
{
   bool uniq_v;
   vector<helix>:: iterator h_i = Helix_Set.begin();
   vector<helix>:: iterator u_i;
   Uniq_Set.clear();
   Uniq_Set.push_back(*h_i);
   h_i++;
   while(h_i != Helix_Set.end()){
       uniq_v = true;
       u_i = Uniq_Set.begin();
       //cout <<
"-----------------------------------------------------------------------------------------------------------
------" << endl;
       //cout << " iterator " << endl;
       //h_i->display_helix();
       while(u_i != Uniq_Set.end()){
       //cout << " Uniqe " << endl;
```

```
 //u_i->display_helix();
         if(h_i->intersects_with(*u_i)){
           //cout << " Not chosen" <<endl;
           uniq_v = false;
           break;
         }
         u_i++;
       }
       if(uniq_v){
         //cout << " Chosen " << endl;
         //h_i->display_helix();
         Uniq_Set.push_back(*h_i);
         //cout <<
"-------------------------------------------------------------------------------------------------------------
------" << endl;
       }
       h_i++;
   }


}


/* Create RNA data structure from the helix set generated. This is done so that we can use
the efn2 energy calculation function. */
void map_helix_to_RNA(vector<helix>& Repaired_Set,RNA& a)
{
   a.RemovePairs();
   int error;
   vector<helix>::iterator h_u = Repaired_Set.begin();
   while(h_u != Repaired_Set.end())
   {

     //cout <<
"-------------------------------------------------------------------------------------------------------------
------" << endl;
     //h_u->display_helix();
     //cout <<
"-------------------------------------------------------------------------------------------------------------
------" << endl;
     vector<base_pair>:: iterator b_h_u = (*h_u).bases_in_helix.begin();
     while(b_h_u != (*h_u).bases_in_helix.end())
     {
```

```
      error = a.SpecifyPair((*b_h_u).get_i_loc()+1,(*b_h_u).get_j_loc()+1,1);
       //error = a.ForcePair((*b_h_u).get_i_loc()+1,(*b_h_u).get_j_loc()+1);
       if(error!=0) {
            //check to make sure that the return is zero, or else an error has occured
            std::cerr << a.GetErrorMessage(error);
            exit(0);
       }
       b_h_u++;

    }
    h_u++;
  }

}

/* Energy calculation function whose pointer is given to simulated annealing routine.
   First repair the permutation and then calculate the energy of structure.                */

double E1(void *xp)
{
  static int x = 0;
  int s = 1;
  int error;
  //vector<helix> Repaired_Set;
  //repair_permuatation( (*( ((data_type_Helix *)xp)->Helix_Set) ),Repaired_Set);
  //int tmp_size1 = (*( ((data_type_Helix *)xp)->Helix_Set) ).size();
  //cout << "Number of helices Original found is  " << tmp_size1 << endl;
  repair_permuatation( (*( ((data_type_Helix *)xp)->Helix_Set) ),(*( ((data_type_Helix
*)xp)->Final_Set) ));
  //int tmp_size2 = (*( ((data_type_Helix *)xp)->Final_Set)).size();
  //cout << "Number of helices Used  is  " <<  tmp_size2 << endl;
  map_helix_to_RNA((*( ((data_type_Helix *)xp)->Final_Set) ),(*( ((data_type_Helix
*)xp)->a) ));
  if ((*( ((data_type_Helix *)xp)->a) ).ContainsPseudoknot(1))
        cerr << " Yes contains ps" << endl;

   double free_energy = (*( ((data_type_Helix *)xp)->a) ).CalculateFreeEnergy(s);
   error = (*( ((data_type_Helix *)xp)->a) ).GetErrorCode();
   if (error==0) {
        //Note that when calculate energy is called the first time, RNA reads parameter
files from
        //disk at the location specified by environment variable DATAPATH.
```

```
        //These are the .dat files found in the data_tables directory of RNAstructure
        std::cout << "Free energy change is: "<<(*( ((data_type_Helix *)xp)-
>a) ).CalculateFreeEnergy(s)  << "    GetFree = " << (*( ((data_type_Helix *)xp)-
>a) ).GetFreeEnergy(s) << "\n";
    }
    else {
        std::cerr << (*( ((data_type_Helix *)xp)->a) ).GetErrorMessage(error);
    }
     if(free_energy < -25){
       string x_s    =  boost::lexical_cast<string>(x);
       string seed_s =  boost::lexical_cast<string>( ((data_type_Helix *)xp)->seed);
       string energy_s = boost::lexical_cast<string>(free_energy);
       const string f_s = "File_seed" + seed_s + "i_" +   x_s + "Free_EN" + energy_s;
       const string t_s = "Thermo_seed" + seed_s + "i_" + x_s + "Free_EN" + energy_s;
       (*( ((data_type_Helix *)xp)->a) ).WriteCt(f_s.c_str());
       (*( ((data_type_Helix *)xp)->a) ).WriteThermodynamicDetails(t_s.c_str());
     }
     x++;
     //exit(0);
    return free_energy;
}

/* Metric function to determine the difference between two structures. */
double M1(void *xp, void *yp)
{
  double free_energy_xp = E1(xp);
  double free_energy_yp = E1(yp);

  return fabs(free_energy_yp - free_energy_xp);
}

/* Step function. Here we implement helix swap in classical sense as well as the multiple
swaps dependent on temperature. */

void S1(const gsl_rng * r, void *xp, double step_size)
{

  int Helix_Size = (*( ((data_type_Helix *)xp)->Helix_Set) ).size();
  int Guide_Size = (*( ((data_type_Helix *)xp)->Guide_Set) ).size();


  int n1 = gsl_rng_uniform_int(r,Helix_Size);
```

```
 int n2 = gsl_rng_uniform_int(r,Helix_Size);
  if(Guide_Size > 0){
      n1 = n1 < GUIDE_NUM ? GUIDE_NUM + 1 : n1;
      n2 = n2 < GUIDE_NUM ? GUIDE_NUM + 1 : n2;
  }
  int num_swaps = ( (( ((data_type_Helix *)xp)->curr_temp) )*0.1 < 1 ) ? 1 :
(( ((data_type_Helix *)xp)->curr_temp) )*0.1 ;
  for(int num_mu = 0; num_mu < num_swaps; num_mu++){
    swap((*( ((data_type_Helix *)xp)->Helix_Set) )[n1],(*( ((data_type_Helix *)xp)-
>Helix_Set) )[n2]);
    n1 = gsl_rng_uniform_int(r,Helix_Size);
    n2 = gsl_rng_uniform_int(r,Helix_Size);
    if(Guide_Size > 0){
      n1 = n1 < GUIDE_NUM ? GUIDE_NUM + 1 : n1;
      n2 = n2 < GUIDE_NUM ? GUIDE_NUM + 1 : n2;
    }
  }

}

/* Print the helix set  function*/
void P1(void *xp)
{
  ((data_type_Helix *)xp)->curr_temp /= MU_T;
  vector<helix>::iterator h_i = (*( ((data_type_Helix *)xp)->Final_Set) ).begin();
  cout << endl <<  " Final Helix Set Begin " << endl;
  while(h_i !=  (*( ((data_type_Helix *)xp)->Final_Set) ).end())
  {
    cout <<
"-------------------------------------------------------------------------------------------------------
------" << endl;
    h_i->display_helix();
    cout <<
"-------------------------------------------------------------------------------------------------------
------" << endl;
    h_i++;
  }

  cout << endl <<  " Final Helix Set End " << endl;
}
```

```
/* The main loop of the implementation */

int main(int argc, char* argv[])
{

   char chr;
   string s1,s2,orig_nucleotide;
   ifstream myfile(argv[1]);

   if (myfile.is_open())
   {
    cout << "Opened file = " << argv[1] << endl;
   }
   else{
    cout << "Can not open file" << endl;
   }

   int mode = 0;

   vector <nucleotide_base> all_bases;
   vector<base_pair> all_bps;
   int i = 0;
   while(myfile)
   {
    myfile.get(chr);
    string s;
    s.insert(0,1,chr);
    orig_nucleotide.append(1,chr);
    nucleotide_base b(s,i++);
    all_bases.push_back(b);
   }

   if(mode == 0) {
      if (all_bases.size() < 1){
       cout << "ERROR : Empty base list " << endl;
      }
      else{
       //FIXME Crude method of dealing with null terminated string.
       all_bases.pop_back();
       all_bases.pop_back();
       cout << " number of bases = " << all_bases.size() << endl;
      }
```

```
      vector<base_pair> all_canonical_bps;

      for ( int i = 0 ; i < all_bases.size()-1 ; i++ ){
         for ( int j = i+1; j < all_bases.size(); j++ ) {
            base_pair
bp(all_bases[i].get_base_type(),all_bases[j].get_base_type(),all_bases[i].get_base_locatio
n(),all_bases[j].get_base_location());
            if (bp.is_canonical_base_pair())
              all_canonical_bps.push_back(bp);

            all_bps.push_back(bp);
         }
      }
         cout << "Size of canonical bps = " << all_canonical_bps.size() << endl;
   }

   cout << "Size of all       bps = " << all_bps.size() << endl;
   vector<helix> Helix_Set;
   vector<helix> Final_Set;
   generate_helix_set(all_bps,Helix_Set,mode,all_bases);

   vector<helix> Guide_Set;
   mode = 1;
   if(mode){
      vector<base_pair> guide_bps;
      string line;
      int i1,i2,i3,i4,i5;
      ifstream mybpfile(argv[2]);
      while(mybpfile){
        mybpfile >> i1 >> s1 >> i2 >> i3 >> i4 >> i5  ;
        if ( i4 != 0 ){
             s1 = all_bases[i1-1].get_base_type();
             s2 = all_bases[i4-1].get_base_type();
             base_pair bp(s1,s2,i1-1,i4-1);
            guide_bps.push_back(bp);
        }
      }
      generate_helix_set(guide_bps,Guide_Set,1,all_bases);
   }
```

```
  const gsl_rng_type * T;
  gsl_rng * r;
  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc(T);

  printf ("seed = %lu\n", gsl_rng_default_seed);
  srand (gsl_rng_default_seed );
  random_shuffle ( Helix_Set.begin(), Helix_Set.end() );


  const char* orig_string = orig_nucleotide.c_str();
  cout << " String="<<orig_nucleotide << endl;
  RNA a(orig_string,true);
  float init_temp = T_INITIAL;


  data_type_Helix xp =
{init_temp,&a,&Helix_Set,&Final_Set,&Guide_Set,gsl_rng_default_seed};
  if ( mode ){
    int Guide_Size = (*( ((data_type_Helix )xp).Guide_Set) ).size();
    cout << " Guide Size = " << Guide_Size << endl;
    if(Guide_Size > 0){
       for ( int i = 0; i < GUIDE_NUM ; i++ ){
          (*( ((data_type_Helix )xp).Helix_Set) )[i] = (*( ((data_type_Helix )
xp).Guide_Set) )[i];

          cout << " First Helix " << endl;
          cout << " ---------------------------------------------------------------- " << endl;
          ((*( ((data_type_Helix )xp).Helix_Set) )[i]).display_helix();
          cout << " ---------------------------------------------------------------- " << endl;
       }
     }
    gsl_siman_solve(r,&xp,E1,S1,M1,P1,NULL,NULL,NULL,64,params);
  }
  else{
    gsl_siman_solve(r,&xp,E1,S1,M1,P1,NULL,NULL,NULL,64,params);
  }


}
```

```
**********************************************************************
File : evaluate.pl

/*
  Calculate the evaluation metric between predicted structure and native strucure.
*/
**********************************************************************
#!/usr/bin/perl
use strict;
use warnings;

print (scalar(@ARGV)!=2 ? die "Please specify two ct files\n" : "\n");
if ( -z $ARGV[0] || -z $ARGV[1])
{
die( "File is empty\n");
}


my %hash = ();
for (my $i=1; $i<3; $i++ )
{
 my $firstline = 0;
 open( IN, $ARGV[$i-1] ) or die "Cannot open the file\n";
 while( my $line = <IN>)
 {
        chomp($line);
        chop ($line) if ( $line =~/\r/ );
        if($firstline == 0)
        {
         $firstline = 1;
        }
        else
        {
                my @base_pairs = split(" ", $line);
                #print "@base_pairs\n";
                if ( $base_pairs[4] != 0 )
                {
                 $hash{$i}{$base_pairs[0]} = $base_pairs[4];
                }
                else
                {
                 $hash{$i}{$base_pairs[0]} = -1;
```

```
                }
        }

 }
close(IN);
}

my @keys = sort {$a <=> $b } keys(%hash);
my $key;
foreach $key (@keys)
{
      #print "$key\n";
         #print "*" x 10 ;
         #print "\n";
         my @bases = keys(%{$hash{$key}});
         foreach my $pair (@bases)
         {
          #   print "$pair => $hash{$key}{$pair}\n";
         }

         #print "*" x 10 ;
         #print "\n";

}
my $TP = 0;
my $FP = 0;
my $FN = 0;
my $pre;
my $nav;
my @predict_bases = sort {$a <=> $b} keys(%{$hash{1}});
my @native_bases  = sort {$a <=> $b} keys(%{$hash{2}});


foreach $pre (@predict_bases)
{
        if ( exists($hash{1}{$pre}) &&  exists($hash{2}{$pre}) )
        {
                if( $hash{1}{$pre} == $hash{2}{$pre} )
                {
                   print " TP $hash{1}{$pre}  $hash{2}{$pre}\n";
                   $TP = $TP+1;
                }
                elsif ( $hash{1}{$pre} !=  $hash{2}{$pre})
```

```
                    {
                    if ( $hash{1}{$pre} == -1 && $hash{2}{$pre} != -1)
                    {
                      print " FN $hash{1}{$pre}  $hash{2}{$pre}\n";
                      $FN = $FN +1;
                    }
                    elsif ( $hash{1}{$pre} != -1 && $hash{2}{$pre} == -1)
                    {
                      print " FP $hash{1}{$pre}  $hash{2}{$pre}\n";
                      $FP = $FP+1;
                    }
                    else
                    {
                      print " FN $hash{1}{$pre}  $hash{2}{$pre}\n";
                           $FN = $FN + 1;
                    }
                  }
          }

}

print "TP = $TP\n";
print "FP = $FP\n";
print "FN = $FN\n";
```