View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by SJSU ScholarWorks

San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

2006

Study of RNA Secondary Structure Prediction Algorithms

Lisa Yu
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
Yu, Lisa, "Study of RNA Secondary Structure Prediction Algorithms" (2006). *Master's Projects*. 23.
DOI: https://doi.org/10.31979/etd.3ggp-5fwe
https://scholarworks.sjsu.edu/etd_projects/23

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Study of RNA Secondary Structure Prediction Algorithms

A Writing Project

Presented to

The Faculty of the Department of Computer Science
and the Department of Biological Science
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by
Lisa Yu
Advisor: Professor Sami Khuri
December 2006

© 2006
Lisa L.Yu
ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Sami Khuri

Dr. John Avila

Dr. Leslee Parr

APPROVED FOR THE UNIVERSITY

ABSTRACT

Dynamic programming algorithms such as Nussinov algorithm and Zuker algorithm define criteria to search the most stable RNA secondary structures. Stochastic Context-Free Grammar (SCFG) predicts the most possible RNA secondary structure using context-free grammar and a defined set of probabilities for each grammar rule. These algorithms form the base of using computer programs to predict RNA secondary structures without pseudoknots.

In this report, we review these RNA secondary structure prediction algorithms and present our own software implementations of these algorithms. The Nussinov algorithm is easy to understand. But our results show that the Nussinov algorithm is overly simplified and can not produce the most accurate result. The SCFG algorithm may be powerful. But its result is also inaccurate because there are no accurate probabilities for each corresponding grammar rule. The Zuker's minimum free energy method incorporated far more biological knowledge in its energy definitions. Thus, its predictions are much better than the other two algorithms.

Our implementations use both recursive and non-recursive function calls. Recursion is easy to understand, but recursion introduces significant overhead. We are able to rearrange the function calls to effectively stop the recursion. The non-recursion feature allows us to parallelize the most computing intensive part of the calculation.

By abstracting a secondary structure to a tree representation and a string representation, we compared our prediction results with the results from experiment measurement or non-conventional general purpose computational methods, and results from popular package such as MFOLD.

Our results also illustrate the limitation of these algorithms. The limitations clearly demonstrate that more biological and chemical knowledge of RNA need to be incorporated into the RNA secondary structure prediction algorithms.

TABLE OF CONTENTS

1. INTRODUCTION.. 9
1.1 The RNAs, DNAs and Proteins 9
1.2 RNA Secondary Structure.. 9
1.3 Computational Prediction of RNA Secondary Structure....................10
1.4 Algorithms... 10
1.4.1 The Nussinov Algorithm... 11
1.4.2 The Zuker Algorithm... 12
1.4.3 Stochastic Context-Free Grammar 14
1.5 Online Resources.. 15
2. SCOPE OF THIS PROJECT... 18
3. SOFTWARE FRAMEWORK AND DESIGN...19
3.1 Common Modules... 19
3.2 Algorithm Dependent Modules.. 20
3.3 Implementation Platform... 21
4. IMPLEMENTATION...22
4.1 Common Modules 22
4.2 The Implementation of the Nussinov Algorithm..................................23
4.3 The Implementation of SCFG with Nussinov rules............................ 23
4.4 The implementation of Zuker Algorithm.. 25
4.4.1 Energy Calculation and Tracking Matrices 25
4.4.2 Trace-back.. 27
4.4.3 Zuker Energy Functions Descriptions... 28
4.4.4 Calling Fortran from C... 29
4.5 Parallelization Using OpenMP... 30
4.6 “Trace-All-Results” Nussinov Algorithm.. 30
5. RESULTS AND ANALYSIS...33
5.1 Visual Comparison of Folding Results...33
5.2 Methods to Compare Secondary Structures...................................... 36
5.2.1 Tree Representation of RNA Secondary Structure............................36
5.2.2 String Notation of the Tree Representation 37
5.2.3 Editing Distance of Tree Representations 37
5.3 Comparison Using Tree Representations and String Notations........... 39
5.4 Comparison of Run Time.. 40
6. CONCLUSION..43
APPENDICES.. 45
Appendix A: Reference... 45
Appendix B: Reference Source Code from MFOLD [ZMT98]: maxn2.inc......... 47
Appendix C: Reference Source Code from MFOLD [ZMT98]: rna_or_efn.inc...48
Appendix D: Reference Source Code from MFOLD [ZMT98]: efiles.f..............49
Appendix E: Source Code : ct.h.. 69
Appendix F: Source Code : ct.c...70
Appendix G: Scource Code: fasta.h...73
Appendix H: Source Code : fasta.c..74

Appendix I: Source Code : n4ct.c 76
Appendix J: Source Code: scfg.c 80
Appendix K: Source Code: n4m.cc 85
Appendix L: Source Code: readMfoldDat.h 91
Appendix M: Source Code: readMfoldDat.c 93
Appendix N: Source Code: freeEnergy.h 96
Appendix O: Source Code: freeEnergy.c 98
Appendix P: Source Code: z4ct.c 104

LIST OF TABLES

Table 1.1 A typical RNA entry in RNase P database……………………………………16
Table 5.1 Comparison of various folding results with
RNase P database and MFOLD.....…………………………………………39
Table 5.2 Comparison of folding results of several other RNAs………………………40
Table 5.3 Measurement of CPU time usage vs. RNA length for
our implementation of Zuker's minimum free energy method……………...41
Table 5.4 Measurement of CPU time usage vs. RNA length
for our implementation of the Nussinov algorithm..…………………………41
Table 5.5 Measurement of CPU time usage vs. RNA length for
our implementation of the SCFG algorithm…………………………………41
Table 5.6 CPU time usage comparison of recursive function calls
and non-recursive function calls……………………………………………..42
Table 5.7 CPU time and wall clock time measurement when
parallelizing the inner loop of the Zuker's minimum
free energy calculation with OpenMP………………………………………42

LIST OF FIGURES

Figure 1.1 Zuker classifies a structure into five types of loops:……………………........12
Figure 1.2 A short sequence folded by the Zuker algorithm (left)
and the Nussinov algorithm (right)…………………………………………13
Figure 1.3 The secondary structure of the RNA in Table 1.1…………………………16
Figure 1.4 Optimal secondary structure of the RNA in Table 1.1
folded by MFOLD………………………………………………………17
Figure 3.1 Flowchart of our implementations…………………………………………19
Figure 3.2 The mutual dependency of matrix element calculation……..………………20
Figure 4.1 A list of sublist to stores base pairs for all best possible foldings………….31
Figure 4.2 Combination of a base pair (i,j) and
the folding results of subsequence i+1…j-1……………………………..…32
Figure 4.3 Concatenate X number of sublists of i…k and
Y number of sublists of k+1…j……………………………………………32
Figure 5.1 The secondary structure of *Bordetella bronchiseptica*……………………..33
Figure 5.2 RNA secondary structure comparison of MFOLD
vs. our implementation of Zuker's minimum free energy method…………34
Figure 5.3 RNA Secondary structure folded by
the Nussinov algorithm and SCFG algorithm………………………………35
Figure 5.4 RNA secondary structure and its tree representation.………………………37
Figure 5.5 Edit operations of a tree……………………………………………………38

1. INTRODUCTION

This chapter describes the description of RNA and RNA secondary structure, and provides detailed descriptions of several common computational algorithms for RNA secondary structure prediction.

1.1 The RNAs, DNAs and Proteins

A DNA molecule is a two-strand structure. A DNA strand is a sequence of nucleotides that stores genetic information. The two strands are essentially complement and "anti-parallel", via Watson-Crick pairs. The two strands store the same, redundant information. Genetic information is stored in DNA molecules as sequence of nucleotides. There are four types of nucleotides in DNA. They are guanine, cytosine, adenine and thymine, denoted by G, C, A, T. In particular, C and G can form a stable base pair. A and T can also form a stable base pair. Those base pairs are called Watson-Crick pairs. Other combinations do not have the right space and chemical interaction to form stable base pairs. DNA is replicated relatively faithfully from one generation to the next generation [KR03].

RNA has single-strand structure. In a RNA sequence, thymine (T) is "replaced" by uracil (U). U and A can form a Watson-Crick pair. U and G can form a wobble pair. Some viruses do not have DNA to store information. Instead, their genetic information is stored in RNA. By convention, the starting end of a strand in DNA or RNA is labeled 5', and the tail end is labeled 3' [KR03].

DNA is the genetic information carrier. Protein does the actual chemical and biological work according to the information. Protein is made of single strand sequences of twenty types of amino acids. The "Central Dogma" of molecular biology states that all living things on earth pass genetic information from DNA to RNA to protein. However, RNA's role is not limited to just passing genetic information from DNA to protein. RNA has many other functions. Therefore, there are many types of RNAs, such as messenger RNA (mRNA), transfer RNA (tRNA), ribsomal RNA (rRNA), small nuclear RNA (snRNA), etc [KR03].

1.2 RNA Secondary Structure

RNA and protein are single stranded sequences. Each of them can fold and form base pairs among nucleotides or amino acids on the same sequence. This three-dimensional tertiary structure maximizes the chemical and biological effects of RNA and protein. Secondary structure refers to how a sequence pairs with itself in two-dimension. Understanding the secondary structure is the first step for comprehending the far more complicated tertiary structure. In fact, over the history of evolution, members of a RNA family conserve their secondary structures more than they conserve their primary sequences [DEK98]. In this report, we focus on RNA secondary structure prediction.

The experimental measurement of RNA secondary structure provides raw data for later studying. There are two primary experimental methods to determine

RNA (tertiary) structure [KR03]. The X-ray crystallography method measures the RNA structure based on deflected light pattern of a crystallized RNA molecule. This method is not completely effective due to the difficulty of crystallizing RNA and the lack of short wave X-ray laser. The Nuclear Magnetic Resonance spectroscope method measures the RNA spacial structure via the energy level split of molecule atoms under external magnetic field [KR03]. However, the complex data analysis and Fouier transformation take a long time even with supercomputers. Because both experimental methods are expensive and not very efficient, using a computer to predict RNA secondary structures becomes attractive.

1.3 Computational Prediction of RNA Secondary Structure

All computational prediction methods involve some kind of modeling. Virtually, it is not possible to compute a RNA secondary structure directly using physical, chemical or biological methods. Typical computer models define modeling criteria in order to search for the most stable or possible structure. These criteria include scores, energy, and probabilities. It is also possible to use information from comparative analysis which is based on the fact that the RNA family members share similar secondary structure [DEK98] [WDG03] [TCG98].

Base triplets and pseudoknots are very challenging for the computer prediction. Base triplet refers to the case when one nucleotide forms chemical bonds with two other nucleotides that are not directly before or after itself in the primary sequence. All existing computer models avoid base triplets because there is no knowledge about how base triplets form so far. Pseudoknot refers to the case where nucleotides in a loop pair with other nucleotides outside of the loop. Early algorithms ignore the existence of pseudoknots when predicting RNA secondary structures. Pseudoknot prediction is one of the interesting research topics in recent years [AKU00].

Results of various algorithms are often compared to experimental measurement, or sometimes, results from other algorithms. Several large scale genetic sequence databases are available to general public, such as the Nucleotides database in GenBank provided by the National Center for Biotechnology Information (NCBI), the DNA DataBank of Japan (DDBJ), and the European Molecular Biology Laboratory (EMBL). Those databases contain some RNA's structures either measured by experiments or predicted by some well-known algorithms. There are also smaller specialized databases available. The RNase P RNA database contains secondary structure of many RNA families using non-conventional comparative method.

1.4 Algorithms

The number of possible secondary structures increases exponentially with the sequence length [DEK98]. An algorithm not only needs to distinguish the biologically correct secondary structures from all other incorrect secondary

structures, but also needs to find out all of the biologically correct secondary structures for a given RNA sequence.

1.4.1 The Nussinov Algorithm

The Nussinov folding algorithm [NPG78] [DEK98] searches a secondary structure with the maximum number of base pairs. It is a simple and efficient dynamic programming algorithm. It defines a function δ(i, j), which is equal to 1 if the ith and j^{th} bases (nucleotides) is a complementary base pair; otherwise δ(i, j) = 0. Then, it recursively calculates the elements of a score matrix γ(i, j), which is the maximal number of base pairs that can be found in subsequence i…j. The algorithm includes a fill stage and a traceback stage. The fill stage is defined by Nussinov as the following:

Initialization:

$$\gamma(i,i-1) = 0 \quad \text{for } i = 2 \text{ to } L$$
$$\gamma(i,i) = 0 \quad \text{for } i = 1 \text{ to } L$$

Recursion:

$$\gamma(i,j) = \max\begin{cases} \gamma(i+1,j-1) + \delta(i,j) \\ \gamma(i+1,j) \\ \gamma(i,j-1) \\ \max[\gamma(i,k) + \gamma(k+1,j)] \quad (i < k < j) \end{cases}$$

The final score is given by γ(1, L), where L is the length of the sequence. This algorithm has $O(L^3)$ in time and $O(L^2)$ in memory.

The traceback stage of the Nussinov algorithm is defined as:

Initialization:
```
push (1, L) into the stack
```

Recursion:

Repeat until the stack is empty:
```
Pop (i,j)
     if i ≥ j continue
     else if γ(i+1,j) = γ(i,j) push(i+1,j)
     else if γ(i,j-1) = γ(i,j) push(i,j-1)
     else if γ(i+1,j-1) = γ(i,j)
             record i,j base-pair
             push(i+1, j-1)
     else for k=i+1 to j-1
             if γ(i,k)+ γ(k+1,j) = γ(i,j)
                  push(k+1,j)
                  push(i,k)
                  break
```

The traceback stage has O(L) in time. Sometimes, there are several structures with the same number of base pairs. However, this traceback algorithm only traces one of the best structures.

The Nussinov algorithm does not deal with pseudoknots. Maximizing the number of base pairs is an overly simplistic criterion which can not give an accurate prediction. But it is the first algorithm and shares the same idea with those more sophisticated energy minimization algorithms and probabilistic based SCFG algorithms.

1.4.2 The Zuker Algorithm

Zuker divides a secondary structure into elements that can be described as graphs, sometimes called loops, and assigns free energy based on the face of those graphs [ZS81]. The structure with the lowest free energy is the optimal structure. Figure 1.1 shows different types of loops.

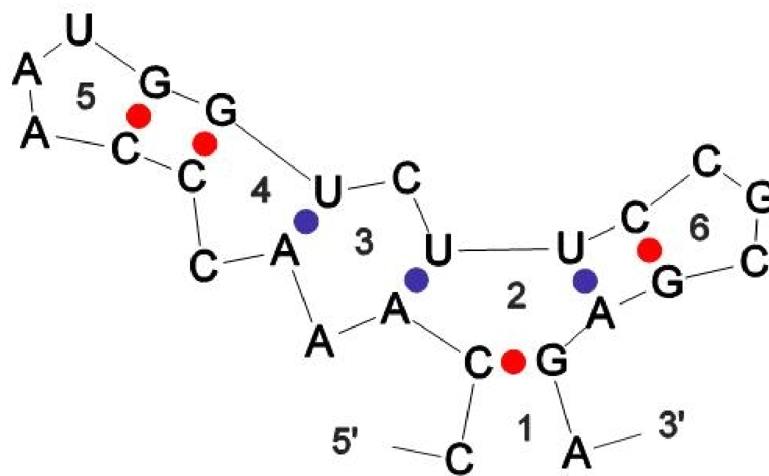


Figure 1.1 Zuker classifies a structure into five types of loops: hairpin loop (5, 6), stack region (consecutive base pairs between 5 and 4, or between 2 and 6), bulge loop (4), interior loop (3) and bifurcation loop (2).

Zuker defines two matrices W(i,j) and V(i,j) [ZS81]. W(i,j) is the total free energy of subsequence i to j. V(i,j) is also defined as the total free energy of subsequence i to j if i and j pairs, otherwise, V(i,j) = ∞. The recursion relations for V(i,j) is defined as:

$$V(i,j) = \min\begin{cases} FH(i,j) \\ \min[FL(i,j,h,k) + V(h,k)] \quad \text{where} \quad i < h < k < j \\ \min[W(i+1,k) + W(k+1,j-1)] \quad \text{where} \quad i+1 < k < j-1 \end{cases}$$

Here, FH(i,j) is the energy of hairpin loop i…j. FL(i,j,h,k) is the energy of 2nd order loop such as stack region, bulge loop and interior loop i…h…k…j. The last item is the energy for bifurcation loop. The last item repeats over i+1 <k < j-1 because i and j must be a base pair, otherwise V(i,j) = ∞

The recursion relation for W(i,j) is:

$$W(i,j) = \min \begin{cases} W(i+1,j) \\ W(i,j-1) \\ V(i,j) \\ \min[W(i,k) + W(k+1,j)] \quad \text{where} \quad i < k < j \end{cases}$$

W(1,L) gives the final total minimum free energy. The algorithm is O(L^4) in time. The traceback stage in the Zuker algorithm is similar to the traceback stage in the Nussinov algorithm. The Zuker algorithm does not deal with pseudoknots. Figure 1.2 is a comparison of the secondary structures of the same sequence, folded by the Zuker algorithm and the Nussinov algorithm.

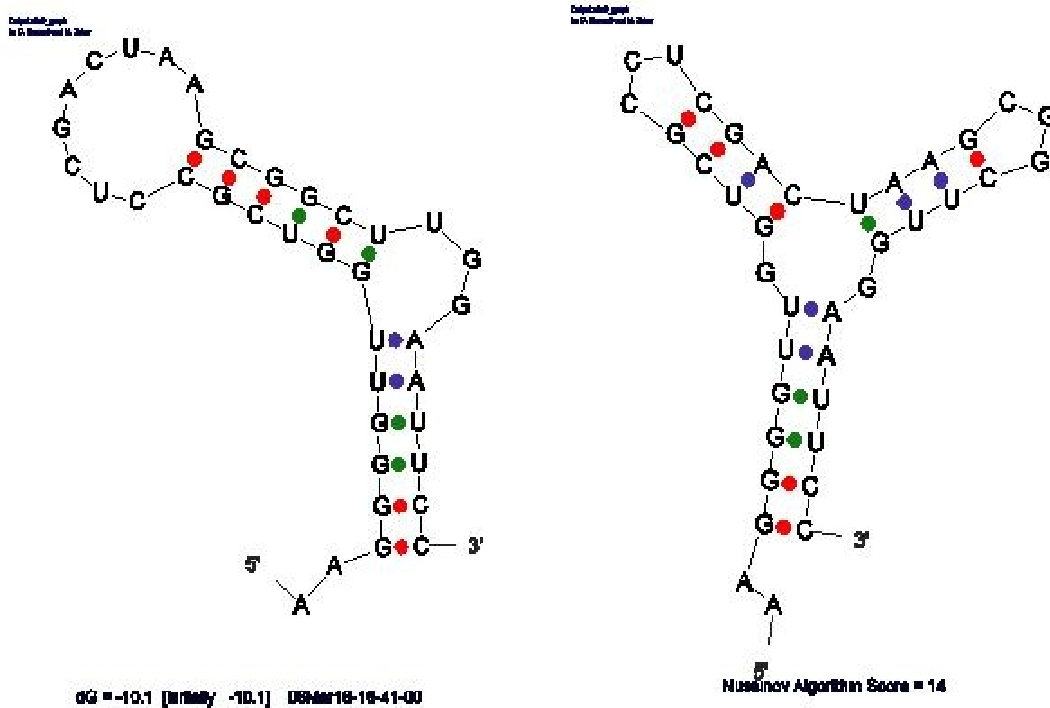


Figure 1.2 A short sequence folded by the Zuker algorithm (left) and the Nussinov algorithm (right). Hairpin loops with length less than 3 nucleotides (sharp U-turn) are not allowed. Note that there are less base pairs in the structure folded by Zuker algorithm than the structure folded by the Nussinov algorithm. Also note that the two algorithms [ZMT99] predict the same stack region near 5’ and 3’, but significantly different for the rest. The Nussinov algorithm obviously favors more base pairs, while the Zuker algorithm favors long stack region.

One of the problems with the Zuker algorithm is that the complex energy functions are not accurate enough. Therefore, a minimum free energy “optimal structure” can not be considered better than those structures with very close free energies [ZMT98]. For this reason, Zuker’s MFOLD package calculates the optimal structure as well as the suboptimal structures within a user defined percentage.

1.4.3 Stochastic Context-Free Grammar

A RNA secondary structure can be captured by a context-free grammar with 4 terminals (c, g, a, u) and a number of non-terminals. In order to predict the best structure, probabilities are introduced for each grammar rule, so the algorithm is named Stochastic Context-Free Grammar (SCFG) [SBH94] [DE04]. A structure’s probability is the joint probability of all grammar rules used to derive the structure, and summed over all possible grammar derivations (parse trees). The predicted structure is the one with the highest structure probability. A simple set of grammar rules that works the same way as the Nussinov algorithm can be defined as the following:

$$S \to aS \mid Sa \mid aSa \mid SS \mid \varepsilon \qquad \text{where} \quad a \in \{\mathrm{A}, \mathrm{U}, \mathrm{C}, \mathrm{G}\}$$

These rules correspond to the following algorithm in the fill stage:
Initialization:

$$\gamma(i, i-1) = \log p(S \to \varepsilon)$$

Recursion:

$$\gamma(i, j) = \begin{cases} \gamma(i+1, j) + \log p(S \to aS) \\ \gamma(i, j-1) + \log p(S \to Sa) \\ \gamma(i+1, j-1) + \log p(S \to aSa) \\ \max\limits_{i<k<j}[\gamma(i,k) + \gamma(k+1, j) + \log p(S \to SS)] \end{cases}$$

$$\sum p = 1 \quad (\textstyle\sum \text{ should sum over all rules and all parse trees})$$

Although it looks very much like the Nussinov algorithm, the SCFG algorithm allows far more comparing criteria than just the number of base pairs. In principal, the probabilities defined here can incorporate biological and chemical knowledge.

Comparative sequence analysis can take advantage of SCFG’s flexible probability definition. This type of analysis compares sequences of many known members of the same RNA family, and defines a co-variation matrix to record the probability of base pairing for any two nucleotides. Then this probability matrix can be used to predict base pairing probabilities for an unknown member of the same RNA family. With the correct SCFG rules, comparative sequence analysis method can be used to predict the new member’s secondary structure.

To convert the Zuker algorithm into a SCFG form, one needs a context-sensitive grammar. For example, the Zuker free energy for stack region is a

function of the the stack region's position in the helix. A context-sensitive grammar rule describing the stack region for Zuker algorithm is cVg → caVug. This can be converted to a context-free grammar rule:

$$V^{b\bar{b}} \rightarrow aV^{a\bar{a}}\bar{a} \qquad a,\bar{a} \in \{A,U,C,G\} \text{ and } a,\bar{a} \text{ basepair}$$

$$V^{a\bar{a}} \text{ "remembers" that it just generated an } a,\bar{a} \text{ basepair.}$$

$$\text{The probability of } V^{b\bar{b}} \rightarrow aV^{a\bar{a}}\bar{a} \text{ depends on previous basepair } b,\bar{b}$$

Thus SCFG rules with the Zuker stack region can be written as:

$$S \rightarrow aV^{a\bar{a}}\bar{a} \mid aS \mid Sa \mid SS \mid \varepsilon$$

$$V^{b\bar{b}} \rightarrow aV^{a\bar{a}}\bar{a} \mid S$$

More SCFG rules are needed to describe other 1st order and 2nd order loops in the Zuker algorithm.

One way to define the probabilities for these SCFG rules is to convert the Zuker energy functions to probabilities using Gibbs-Boltzmann equation: p ~ exp(E/kT), where E is the energy, k is the Boltzmann constant and T is the temperature. However, it is not possible to interpret individual energy correction in the Zuker energy functions in terms of probability.

In reality, defining probabilities for SCFG is not an easy task. Combining unrelated knowledge such as free energy and co-variance matrix together to form a single set of probabilities is even more difficult.

Grammar ambiguity also presents a challenge in SCFG [DE04]. Consider a short sequence AGUCCCC. To derive a secondary structure where the G pairs with the C second to the last nucleotide using SCFG rules for Nussinov algorithm, one can either use

S → aS → aSc → agScc → aguScc → agucScc → aguccScc → agucccc, or

S → aS → aSc → agSc → agScc → …

The final probability of that secondary structure is the sum of the probabilities of all derivations. But the SCFG version of the Nussinov algorithm only calculates the probabilities for each individual derivation. This is called grammar ambiguity. In general, grammar ambiguity is undesirable for usual formal languages. So far there is no systematic way to generate unambiguous grammars.

1.5 Online Resources

Two online resources are particularly useful for studying RNA folding algorithm. The online version of the Zuker MFOLD package [ZMT98] provides easy access

to the state-of-the-art Zuker algorithm. The RNase P database [BRO99] provides many known RNA secondary structures obtained by non-computational methods.

The RNase P database [BRO99] is maintained by the North Carolina State University. It is available online at the following web page: http://www.mbio.ncsu.edu/RNaseP/seqs&structures.html. The web page provides tables of known RNA, including the secondary structures. Table 1.1 is an example, and Figure 1.3 is its secondary structure.

Organism		RNA			Protein
Genus/species	Strain	Seq ID	Seq citation ID	Structure files	
Agrobacterium tumefaciens	A384	M59354	91267952	.gb | .ct | .rnaml | .pdf	

Table 1.1 A typical RNA entry in RNase P database. Seq ID links to NCBI summary page of this RNA; Seq Citation ID links to the corresponding research paper indexed by NCBI PubMed; .ct is the RNA secondary structure in CT format; .pdf is the RNA secondary structure plot in PDF.

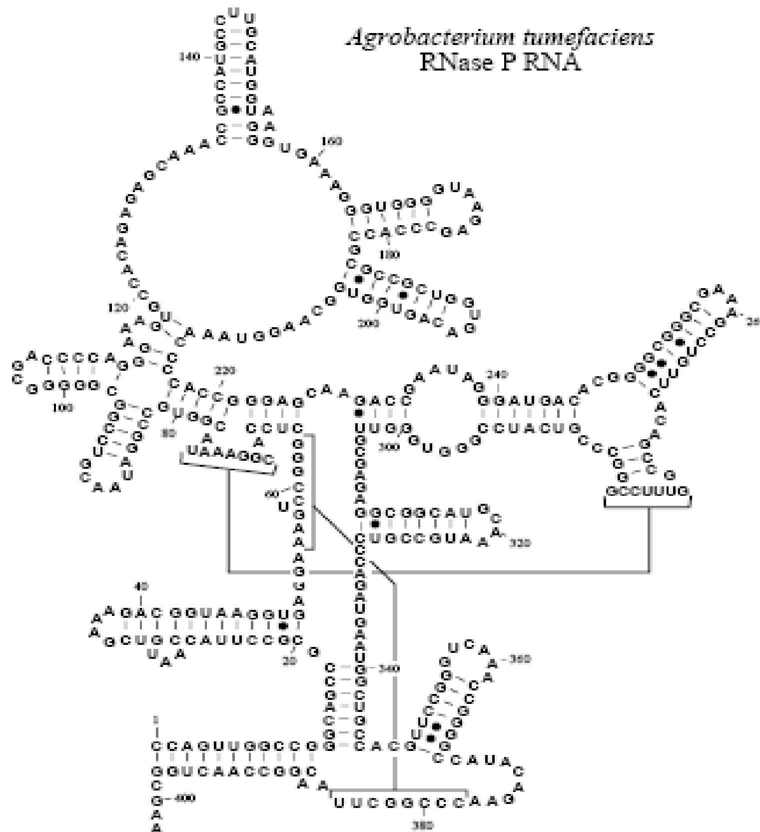


Figure 1.3 The secondary structure of the RNA in Table 1.1. This secondary structure is obtained by non-computational method.

The Zuker MFOLD package [ZMT98] accepts the FASTA format RNA sequence for both linear and circular RNA. It allows users to enter additional constraints and other parameters. The MFOLD package provides the optimal structure as well as suboptimal structures. The web version of Zuker MFOLD package is available at http://www.bioinfo.rpi.edu/applications/mfold/old/rna/. Figure 1.4 is the secondary structure of the same RNA folded by MFOLD.

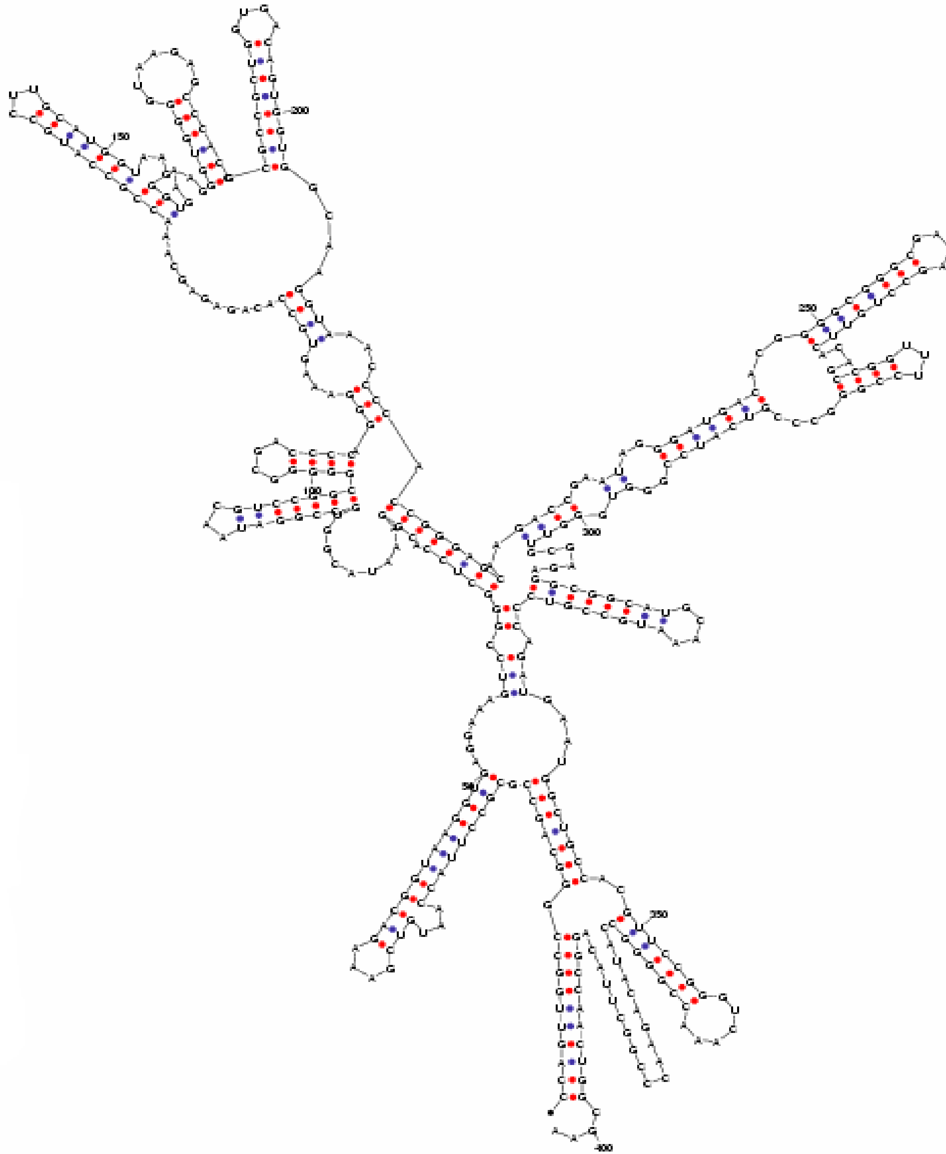


Figure 1.4 Optimal secondary structure of the RNA in Table 1.1 folded by MFOLD.

2. SCOPE OF THIS PROJECT

This chapter describes the scope of this project. In this project, we implement the original Nussinov algorithm [NPG78], a variance of the Nussinov algorithm that traces all possible RNA secondary structures with maximum number of base pairs, an algorithm based on Stochastic Context Free Grammar (SCFG) [SBH94][DE04], and Zuker's minimum free energy method [ZS81].

We discuss modules in our software implementation, problems and solutions during the implementation of recursive calculation and trace-back, an output format to fit the need of graphic display of RNA secondary structure, and an output format to help the comparison of two RNA secondary structure predictions.

Recursive relations are described in all of the above algorithms. Recursion is easy to understand, and thus preferred by algorithms. However, recursive calls consume more CPU time than non-recursive calls. For this reason, we describe a simple technique to effectively cease using recursive relation. This allows us to use some simple parallelization techniques for those algorithms.

We measure the performance of our implementation and compare the results with the predicted time complexity. We also evaluate the accuracy of our implementations. We compare the accuracy via the alignment of coarse-grained string representations and the edit distance of coarse-grained tree representations [SHA88] [SZ90] [HFS94]. We compare our predictions to RNA secondary structures obtained via non-conventional methods or via experiment measurement. We also compare our prediction to the prediction produced by the state-of-the-art MFOLD package from Michael Zuker, et al [ZMT98].

We do not measure the space usage of these implementations because there is no easy to access tool to measure the memory usage during program execution.

The actual implementation of MFOLD package is not in the scope of this report, though the energy correction data used by MFOLD will also be used by our implementation of Zuker's minimum free energy method. For this reason, we provide a detail description of those energy functions and data. Algorithms to calculate the string alignment and tree edit distance are not within the scope of this report. We use a program in the Vienna package [HFS94] for this purpose. Details of the parallelization techniques used in the report are also excluded. Our implementation uses some Container classes from the C++ Standard Template Library. Details of the Container classes are excluded from this report. These Container class instances are only used for storage when we implement a Nussinov algorithm for tracing all possible best results. They are not the core part of the algorithm.

3. SOFTWARE FRAMEWORK AND DESIGN

Our implementations of the algorithms include modules for reading input RNA sequence, initialization, RNA folding calculation, and output. The implementations of the original Nussinov algorithm, SCFG algorithm with Nussinov rules and the Zuker's minimum free energy methods include a similar module for trace-back. The trace-back module is completely different for the modified Nussinov algorithm that can trace all possible best folding results. From now on, we will call this variant of the Nussinov algorithm implementation "Trace-all-results Nussinov". Figure 3.1 shows the execution sequence of these modules.

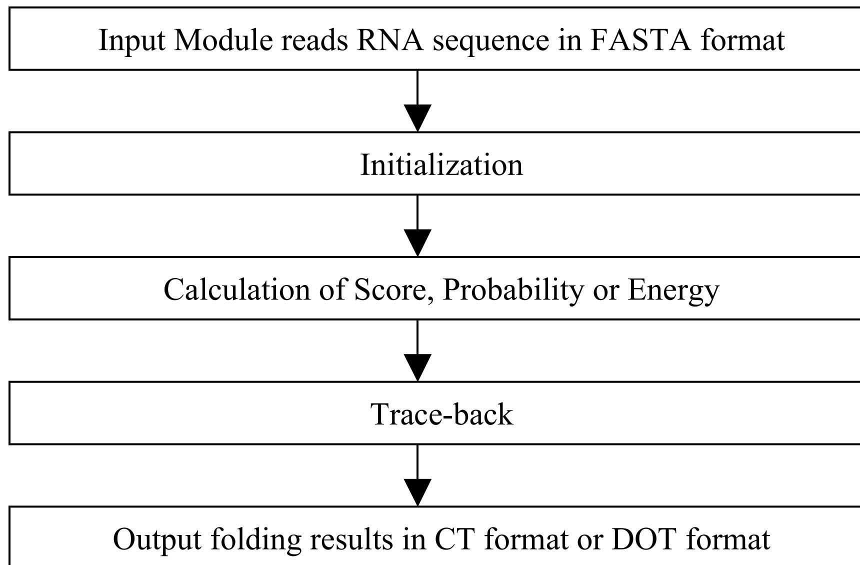


Figure 3.1 Flowchart of our implementations.

3.1 Common Modules

We use the FASTA format for input RNA sequence. The corresponding module is simple and is used by all the implementations in this project.

The output module retrieves base-pair information from a stack, which is filled by the trace-back module. The output module will then print the corresponding output format using the sequence and the base-pair stack. In the case of multiple best folding results, the output modules should be initialized before each use. The output format will be in "connection table" (CT) and DOT format. CT format is used by sir_graph to generate folding graph. Sir_graph program is available in the MFOLD_UTIL package [ZMT98]. DOT format is used by VIENNA package's RNAdistance program [HFS94] to calculate string alignment or edit distance. The output module is common for all implementations.

3.2 Algorithm Dependent Modules

The tasks in the initialization module are algorithm dependent. In Nussinov algorithm, SCFG algorithm, the task is to simply assign initial values for variables and matrix that will be used later. In addition, in the implementation of the Zuker's minimum free energy method, this module reads energy files.

Our implementation of the Zuker's minimum free energy method reuses Michael Zuker's energy correction values in the MFOLD package. Instead of writing our own code to read those complex files, we reuse a FORTRAN 77 code in MFOLD package that is responsible for energy file reading. We provide a C interface to call FORTRAN functions and make the FORTRAN data available to our C language code. Many compiler suites such as GNU compilers, INTEL compilers and SUN Studio compilers provide both FORTRAN and C compilers, and a set of common rules to allow FORTRAN and C to call each other's functions and subroutines, and to exchange data between C's global data and FORTRAN's common blocks.

The folding score, probability or energy is calculated recursively. To avoid repeated calculations for the same subsequence, we setup a $N \times N$ "score" matrix to record the corresponding values for each subsequence. Here N is the length of the RNA sequence, and "score" is the Nussinov algorithm score. For other algorithms, this "score" is probability, energy, etc. The score matrix element (1, N) gives the final score. Obviously, this module is algorithm dependent.

The score matrix is actually more useful than just avoiding repeated calculations. As illustrated in Figure 3.2, when calculating matrix element (i, j), only matrix elements in the shadowed area are needed in the recursive relation. So if the calculation starts with the diagonal line (i, i) and gradually moves to off-diagonal line (i, i+1), (i, i+2) … (i, N), the recursions are not necessary. Therefore, the computational overhead associated with the recursions can be removed, and we expect improvement on performance.

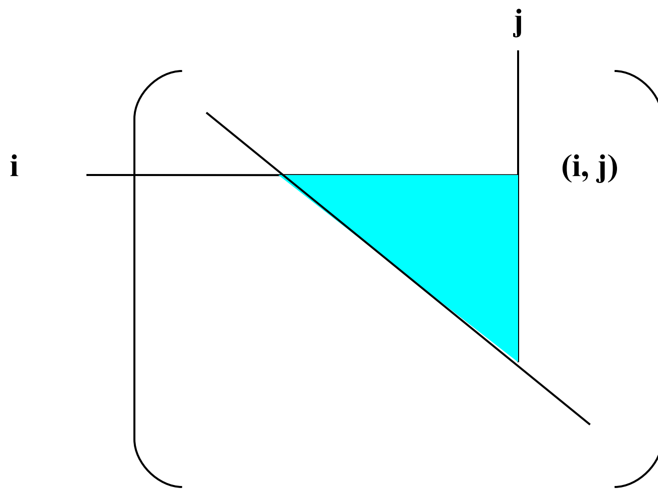


Figure 3.2. The mutual dependency of matrix element calculation. Element (i, j) only depends on the values of elements (h, k) where $h>i$ and $k<j$

Moreover, matrix elements on the same diagonal or off-diagonal line such as, (i, j), (i+1, j+1) …, can be calculated independently. With a modern compiler, it is easy to parallelize these independent calculations.

The trace-back module is also algorithm dependent. In Nussinov algorithm, SCFG and the Zuker's minimum free energy methods, a stack is needed by each of these algorithms. Another stack is needed to store all base pairs found during the traceback stage. The limitation of the trace-back method is that only one of the all possible best "score" results is traced. The "Trace-all-results Nussinov" does not use the trace-back module. As it calculates score for subsequence i … j, it maintains a list of sublists. Each sublist is a unique set of base pairs corresponding to the score for this RNA subsequence i … j. A good candidate to implement this list of pairs is the List class from the C++ Standard Template Library.

3.3 Implementation Platform

Our implementation runs in most UNIX-like environments. It also runs in Microsoft Windows under the CYGWIN environment. GNU compiler is the default choice because it is widely available on many platforms. Compiled executables of the VIENNA package's RNAdistance [HFS94] is available for both Linux and MS Windows. MFOLD package and MFOLD_UTIL package [ZMT98] are available for most UNIX platforms. MFOLD_UTIL provides a set of tools to plot, manipulate and convert RNA secondary structure in CT format. MROLD_UTIL requires OpenGL. We are able to compile MFOLD and MFOLD_UTIL in Linux environment and CYGWIN environment.

4. IMPLEMENTATION

This chapter presents our implementation and pseudocode for Nussinov algorithm and its SCFG peer, as well as the Zuker algorithm. Each algorithm's implementation framework includes four parts. They are responsible for reading the input RNA sequence, initializing the score matrix γ and recursively calculating the elements of score matrix γ, traceback, and outputting the RNA secondary structure. In addition to those algorithms, we discuss the mechanisms of trace-back folding results, Zuker's energy functions, the technique to use C and FORTRAN together, and the parallelization of the folding calculations.

4.1 Common Modules

The common modules shared by each algorithm are the input module and the output module. For example, the sequence reading part described in Nussinov algorithm implementation subsection can actually be reused in other algorithms' implementations.

The RNA sequence reading module accepts the FASTA format, which is widely used in GenBank and other databases. The FASTA format consists of a comment line followed by a line of sequence. The comment line starts with symbol ">" and ends with UNIX new-line character. The sequence line consists of the four nucleotides: C,G,A,U/T. Anything other than C,G,A,U/T is ignored. FASTA files from many databases contain multiple lines of comments and sequences. In our implementation, we only take the first sequence.

Printing out the RNA sequence is another module shared by our implementations of Nussinov, SCFG and Zuker. The output module allows output in the CT format (.ct file), and the DOT format. The CT format is widely used by many packages, including Zuker MFOLD package [ZMT98] to describe the final RNA secondary structure. Many existing programs can take the CT format and plot the RNA secondary structure. The DOT format is used by the secondary structure comparison tools in the VIENNA package [HFS94].

The first line in a CT file is basically a comment, except that the first word on the left should be a number equals to the length of the sequence. Starting from the second line, there are six data columns each line. The first column is the index of the nucleotides within the RNA sequence. The index starts with 1 and ends with sequence length L. The second column is nucleotide for the corresponding index. The third column shows index -1. The fourth column contains index +1. The fifth column displays base-pair nucleotide's index. If no nucleotide base-paired with the nucleotide, it shows 0. And the sixth column is the same as the first column.

The DOT format displays base-pair positions with parentheses. For example, for a 10 nucleotide sequence with two base pairs (2,5) and (7,10), the corresponding DOT expression is

```
.(..).(..)
```

4.2 The Implementation of the Nussinov Algorithm

In Nussinov, the scores are saved in a 2-D matrix γ[N, N]. γ[i, j] is set to -1 to indicate that this element's score hasn't been calculated. Function score(i,j) calculates each γ[i, j] and avoids repeated calculations. We choose "int" type for score matrix γ because the scores are really just integers. Choosing "int" type allows us to implement the Nussinov traceback algorithm without modification. As we will see later, when "scores" are number of "float" type or "double" type, we have to introduce a tracing matrix and modify the tracing back algorithm to avoid testing equality of two floating point numbers in our program.

The recursive relation given by the Nussinov algorithm can be simplified to the following form:

$$\gamma(i,j) = \max\begin{cases} \gamma(i+1, j-1) + \delta(i,j) \\ \max[\gamma(i,k) + \gamma(k+1,j)] \quad (i \le k < j) \end{cases}$$

Here we slightly modify the original Nussinov algorithm, and do not allow sharp U-turn in hairpin loop (e.g. we request at least 3 unpaired nucleotides in a hairpin loop). The recursion tries to avoid repeated calculations of the same score matrix element. The pseudocode for the recursion part is:

```
int γ[N,N];
int score(int i, int j)
{
     return γ[i,j] if γ[i,j] is calculated
     return 0 if i and j makes a sharp U-turn

     γ[i, j] = score(i+1,j-1) + δ(i,j)
     for (k = i to j-1)
          if ( score(i,k)+score(k+1,j) > γ[i,j] )
               γ[i,j] = score(i,k) + score(k+1,j)
     return γ[i,j]
}
```

In the end, γ [1, L] should show the final score.

Traceback pseudocode is the same as shown in the original Nussinov algorithm, see Chapter 1 for detail. In the module, the "record i, j base pair" step is implemented using a stack. We extend the original Nussinov tracing back algorithm to allow printing out in CT and DOT format.

4.3 The Implementation of SCFG with Nussinov rules

In SCFG, probabilities replace scores. Probabilities are real numbers. They are

saved in a matrix, probability[N, N] of 'double'. Function prob(i,j) will calculate each probability[i, j].

Although the rules we used in the SCFG algorithm are based on the Nussinov algorithm, we have to make some modifications for the trace-back. This is because real numbers should not be tested for equality directly. For example, it is wrong to test 3.14159 == 3.14159. Therefore we introduce a tracing matrix pathmatrix[i,j] which stores the actual parsing rules used during the recursions. And we use this information during the traceback step to avoid testing the equality of probabilities. We use matrix pathmatrix[i,j] in the following way

- Initialize pathmatrix[i,j] to -10
- pathmatrix[i,j] > -10 indicates that probability[i,j] has been calculated, and should not be calculated again.
- In function prob(i,j), set pathmatrix[i,j] = -1 when i and j pair.
- pathmatrix[i,j] = k, i<k<j for bifurcation at (i,k)+(k+1,j)

The pseudocode for recursion relation is the following. To write the pseudocode short, we replace probability[] with γ [], pathmatrix[] with p[]:

```
int p[N,N]
double γ[N,N]

double prob(int i, int j)
{
    return γ[i,j] if γ[i,j] is calculated
    if (i and j pair)
    {
        γ[i,j] = prob(i+1,j-1) + log P(S→aSa)
        p[i,j] = -1
    }
    if (prob(i+1,j) + log P(S→aS) > γ[i,j])
    {
        γ[i,j] = prob(i+1,j) + log P(S→aS)
        p[i,j] = i;
    }
    if (prob(i,j-1) + log P(S→Sa) > γ[i,j])
    {
        γ[i,j] = prob(i,j-1) + log P(S→Sa)
        p[i,j] = j
    }
    for (k = i+1 to j-2)
        if (prob(i,k)+prob(k+1,j)+logP(S→SS) > γ[i,j])
        {
            γ[i,j] = prob(i,k)+prob(k+1,j) + log P(S→SS)
            p[i,j] = k
        }
```

```
        return γ[i,j]
}
```

The pseudocode for trace-back is now:

```
push(1, L)
while pop(i, j)
{
        if ( i >= j ) continue
        k = p[i,j]
        if (k == -1)
        {
            record pair (i,j)
            push(i+1,j-1)
        }
        else if (k >= 0)
        {
            push(i,k)
            push(k+1,j)
        }
}
```

Similar to the Nussinov algorithm, the “record pair (i,j)” step is implemented using a stack.

4.4 The implementation of Zuker Algorithm

4.4.1 Energy Calculation and Tracking Matrices

In our implementation of Zuker algorithm, energy for subsequence i to j is stored in element [i, j] of a two-dimension array, Wij[N, N] of type 'float'. Vij[N, N] is defined in the same way, but for ‘V’ energy. Function w(i,j) will calculate each Wij[i, j]. A tracking matrix pathmatrix[N, N] (type 'int') is set to -10 initially. Function w(i,j) changes pathmatrix[i, j] to:

- pathmatrix[i, j] = -5 when the length of i, j is shorter than allowed hairpin loop length (sharp U-turn).
- pathmatrix[i, j] = -5 when i, j don’t pair. In this case and the above, Wij[i, j] = 0, Vij[i, j] = infinity.
- pathmatrix[i, j] = -1 when i, j pairs.
- pathmatrix[i, j] = k for bifurcation at (i,k) + (k+1, j) in recursion of W energy .
- pathmatrix[i, j] > -10 means Wij[i, j] (and Vij[i, j]) has been calculated, and should not be calculated again.

Function v(i,j) calculates Vij[i, j]. v(i,j) function searches in pathmatrix[[i, j] to see if Vij[i, j] has been calculated. For this reason, function w(i,j) always calls v(i,j). But v(i, j) never changes pathmatrix[i, j]. We use a three-dimension tracking matrix, loopmatrix[N, N, 2] of type 'int' to help tracing Vij. The initial value

of loopmatrix[N, N, 2] is not important because v(i,j) will always change the value according to the following rules :

- loopmatrix[i, j, 0] = i+1, loopmatrix[i, j, 1] = j-1 if this is a hairpin or stack.
- loopmatrix[i, j, 0] = ip, loopmatrix[i, j, 1] = jp (ip != jp) if this is a bulge or interior loop of (i,ip,jp,j).
- loopmatrix[i, j, 0] = loopmatrix[i, j, 1] = k if this is a bifurcation at (i,k) + (k+1,j) in recursion of V energy

To simplify the following pseudocode, we write Wij[] as W[], Vij[] as V[], pathmatrix[] as p[] and loopmatrix[] as l[]. FH() is the function to calculate energy for hairpin loop. FL() represents energy functions for stack, bulge loop and interior loop.

```
double V[N,N], W[N,N]

double v(int i, int j)
{
     return v[i,j] if  V[i,j] is calculated
     if (i and j don't pair OR
          i and j makes a sharp U-turn)
     {
          V[[i,j] = ∞
          Return V[i,j]
     }
     V[i,j] = FH(i,j)
     l[i,j,0] = i+1 and l[i,j,i] = j-1

     for (i < h < k < j)
          if (V[i,j] > FL(i,j,h,k) + V(h,k))
          {
               if (stack)
               {
                    l[i,j,0] = i+1 and l[i,j,i] = j-1
                    V[i,j] = FL(i,j,h,k) + v(h,k)
               }
               if (bulge or interior)
               {
                    l[i,j,0] = h and l[i,j,i] = k
                    V[i,j] = FL(i,j,h,k) + v(h,k)
               }
          }
     for (i < k < j )
          if (V[i,j] > w(i,k) + w(k+1,j))
          {
               V[i,j] = w(i,k) + w(k+1,j)
               l[i,j,0] = k and l[i,j,i] = k
         }
```

```
    return V[i,j]
}

double w(int i, int j)
{
      return if W[i,j] is calculated

      if (i and j makes a sharp U-turn)
      {
          V(i, j)
          W[i,j] = 0
          P[i,j] = -5
      }
      else
      {
          W[i,j] = V(i,j)
          if (i, j pairs)
             p[i,j] = -1
          else
             p[i,j] = -5

          for (i ≤ k < j)
             if (W[i,j] > w(i,k) + w(k+1,j))
             {
                  W[i,j] = w(i,k) + w(k+1,j)
                  p[i,j] = k
             }
      }
      return w[i,j]
}
```

Please note that V[ ] and W[ ] represent the minimum free energy matrix, and v() and w() represent the recursive functions that calculate the corresponding matrix elements of V[ ] and W[ ].

4.4.2 Trace-back

The trace-back pseudocode in the Zuker algorithm is:

```
push(1, L)
while pop(i, j)
{
        if ( i >= j ) continue
        k = p[i,j]
        if (k == -1)
        {
            record pair (i,j)
```

```
            ip = l[i,j,0] and jp = l[i,j,1]
            if (ip != jp)
                push(ip,jp)
            else
            { /* bifurcation in V energy calculation */
                push(i+1,ip)
                push(ip+1,j-1)
            }
        }
        else if (k >= 0)
        { /* bifurcation in W energy calculation */
            push(i,k)
            push(k+1,j)
        }
}
```

4.4.3 Zuker Energy Functions Descriptions

Zuker's energy functions [ZMT98] are the combinations of formula and tabulated corrections. We briefly list items in each of these energy functions here.
For hairpin loops, the energy function includes:

- $1.75 \times RT \times \ln(L)$, here **R** is gas constant; **T** is temperature in Kelvin; **L** is the number of single stranded nucleotides. The values are tabulated for L = 1 ~ 30
- Terminal mismatched pairs for subsequence i … j where i+1, j-1 can't pair and L > 4. The values are tabulated.
- Special case for L = 3 or 4, tabulated.
- Special rules from experiment, tabulated.

For stack regions, the energy function includes:

- Tabulated values for Watson-Crick pairs CG and AU, and wobble pairs GU
- A penalty for AU pairs appearing at the end of helix. This penalty is built in the Zuker energy table.

The stack energy does not depend on the length of helix.

For bulge loops, the definition of energy function depends on the bulge size L, which is defined as the number of single stranded nucleotides:

- For L = 1, the bulge is treated as a stack
- Use tabulated values for $1 < L \leq 30$.
- Use formula $1.75 \times RT \times \ln(L)$ for L > 30.

For interior loops, the energy function depends on L and a(L). L is defined as =L1+L2, where L1 and L2 are single stranded nucleotides at each side. a(L) describes the asymmetrical part of the two sides. $a(L) = | L1 – L2 |$. The energy function is:

- For L > 4 or a(L)>1
    - Size dependent contribution identical to bulge.

    - Terminal mismatch, tabulated values.
    - Asymmetric penalty, tabulated values.
- Otherwise, special rules for 1x1, 1x2 and 2x2 interior loop, tabulated.

The above energy functions are originally defined in MFOLD [ZMT98]. We used them in our implementation as well. We have not defined penalty energy functions for bifurcation loops. MFOLD, on the other hand, does include a penalty energy function for bifurcation loops. Due to the difference in algorithm, we can't use this penalty energy function, and we expect our implementation gives a lower energy than MFOLD for the same RNA secondary structure.

4.4.4 Calling Fortran from C

Zuker's energy functions incorporated many tabulated special cases and corrections. They are stored in flat files, with complex structures. One Fortran file in MFOLD is responsible to read these files and store them in Fortran arrays. These arrays are shared with other Fortran files via Fortran's Common Blocks. We re-use this Fortran code in our C implementation, thus we need to access these Fortran data from our C program.

Most modern compiler suite follows a set of common methods to allow Fortran and C programs to share data. Here are some of the methods we use in our implementation:

- A 5 dimensional array 'integer asint3(6,6,5,5,5)' in FORTRAN corresponds to a 5 dimensional array of 'int' type and with the indexes in reversed order in C.
- A Fortran common block 'efiles' corresponds to global structure 'efiles_' in C. For example, to access the following Fortran common block

```
integer asint3(6,6,5,5,5),asint5(6,6,5,5,5,5,5),bulge(30)
common /efiles/ asint3,asint5,bulge
```

  We rewrite the above code in C as the following:

```
struct {
    int asint3[5][5][5][6][6];
    int asint5[5][5][5][5][5][6][6];
    int bulge[30];
} efiles_;
```

- A Fortran subroutine engread() corresponds to a C function 'void engread_()'.

Using the corresponding components in the above struct efiles_ is confusing, because we need to take care of both the 'efiles_' name prefix and the reversed order of the array. To makes it easy to use and less confusion, we define another set of arrays as the following:

```
asint3[n][m][k][j][i] = efiles_.asint3[i][j][k][m][n]
```

4.5 Parallelization Using OpenMP

The final minimum free energy is Wij[1,N], which is calculated by simply call w(1,N). The calculation of Wij[1,N] produces grand recursions all the way down and induce the overhead associated with the recursions. As we discussed early in Chapter 3, recursion can be avoided by starting the calculation from the diagonal elements of matrix Wij[ ] and gradually move to the off-diagonal elements as indicated in the following pseudocode.

```
for (k=HAIRPINLEN-1; k<seqlen; k++)
#pragma omp parallel for
    for (i=0; i<seqlen-k; i++)
        Energy(i,i+k);
```

Because the calculations of the same elements of the diagonal or off-diagonal line (the inner loop) are independent, we can calculate them in parallel. OpenMP is perhaps the best tool to parallelize this type of calculation. We only need to add a #progma directive in the code. The directive is ignored by any compiler that doesn't support OpenMP, such as the old GNU compiler. The directive is used by an OpenMP awared compiler to generate executable for parallel execution.

4.6 "Trace-All-Results" Nussinov Algorithm

Here we describe a variant of the Nussinov algorithm that is capable of tracing all possible best folding results. The basic idea is that when calculating subsequence i … j, we not only record its score, but also record all possible base-pairs sets. Like the original Nussinov algorithm, besides input and output modules, our implementation has two stages (module[1]). The first stage is the original score calculation. In the second stage, trace-back, we record all possible base-pair sets for each i…j. The number of base pairs in each set equals to the score of i and j, $\gamma[i, j]$.

We implement base pair in a simple struct:

```
struct Pair {
    int i,
    int j,
}
```

The set is implemented with the List class in C++ Standard Template Library. When we find a new pair to add into the list, we push the pair to the end of the list. A list iterator can be used to loop through all elements of the list.

[1] We do not have to use a two stage calculation. But using two stages allows us to avoid repeated calculation of the list of list, thus provides some optimization.

```
struct Pair p;
list<struct Pair> l;
list<struct Pair>::iterotor i;

// push an element into list l
l.push_bach(p);

// loop through all elements of l
for (i = l.begin(); i != l.end(); i++) { . . . }

// Clear all elements of list l
l.erase( l.begin(), l.end() );
```

We also define a struct with all information of a subsequence, a … z

```
struct Gamma {
    int a, z;
    int pairListLen;
    list<struct Pair> pairLists
}
struct Gamma GammaMatrix[N, N];
```

For a given subsequence a to z, GammaMatrix[a, z] stores all base-pair sets for all possible best foldings. Because base pair set is implemented as list, `Gamma.pairLists` is actually a list of sublists, with each sublist corresponding to one of the best foldings.

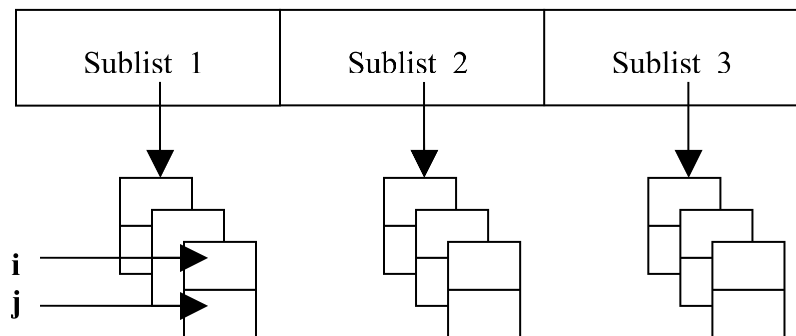


Figure 4.1 A list of sublists to stores base pairs for all best possible foldings.

`Gamma.pairListLen` is the length of each sub-list. The number of base pairs in each sublist equals to the score of subsequence a…z.

The calculation of GammaMatrix[i, j] consists of two steps. The first step calculates $\gamma(i+1, j-1) + \delta(i, j)$. If i and j pairs, then we need to combine pair i, j with all sublists in subsequence i+1…j-1

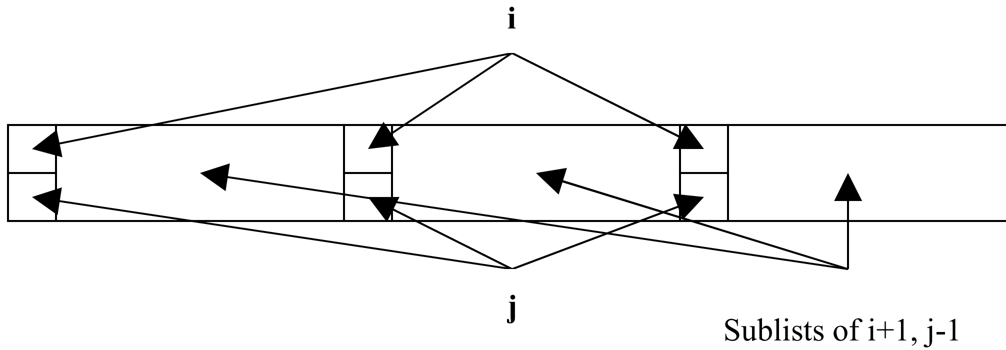


Figure 4.2 Combination of a base pair (i,j) and the folding results of subsequence i+1...j-1.

The second step is to calculate γ(i, k)+ γ(k+1, j), i≤k<j, and concatenate each sublist of k+1…j to each sublist of i…k. Because the length of sublist equals to the corresponding score, the pseudocode is:

```
pairListLen=score[i, j];
for (k=i; k<j; k++)
    if ( γ[i, j] == γ[i, k]+ γ[k+1, j] )
       concatenate sublists of i…k and sublists k+1…j
```

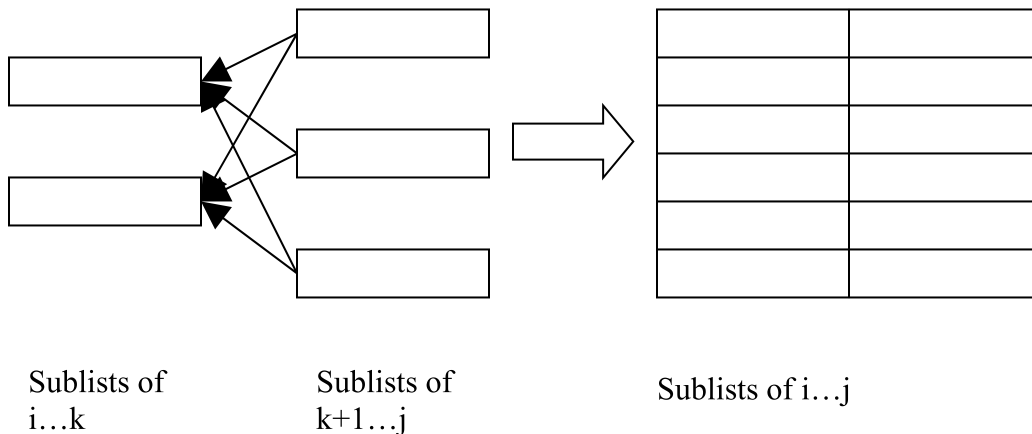


Figure 4.3 Concatenate X number of sublists of i…k and Y number of sublists of k+1…j. This forms X*Y sublists for subsequence i…j

After step one and step two, each sublist needs to be sorted, and redundant sublists should be removed. The sorting can be done by fetching elements of a sublist into an instance of C++ Standard Template Library class Set. The Set class automatically sorts its members according to a user defined comparison function. In the end, GammaMatrix[N, N].pairLists stores all possible best folding results.

5. RESULTS AND ANALYSIS

We are primarily interested in the accuracy and run time measurement of our implementations. Folding accuracy is measured by the edit distance of the tree representation and alignment of string representation. We compare our folding implementations with non-conventional folding results when available. We also compare our folding results with the results from MFOLD.

The run time is measured by the CPU time usage. The CPU time usages should be measured for the folding modules only. Because the complexity of the folding module is at least one order higher than the complexity of the trace-back module, it makes little difference for our testing to use the CPU time of the whole program.

5.1 Visual Comparison of Folding Results

We obtain some folded structures from the RNase P RNA database [BRO99]. Most secondary structures in that database are folded by using a method that is specially targeted to tRNAs [HWF01]. Figure 5.1 is the secondary structure of *Bordetella bronchiseptica* listed in that database.

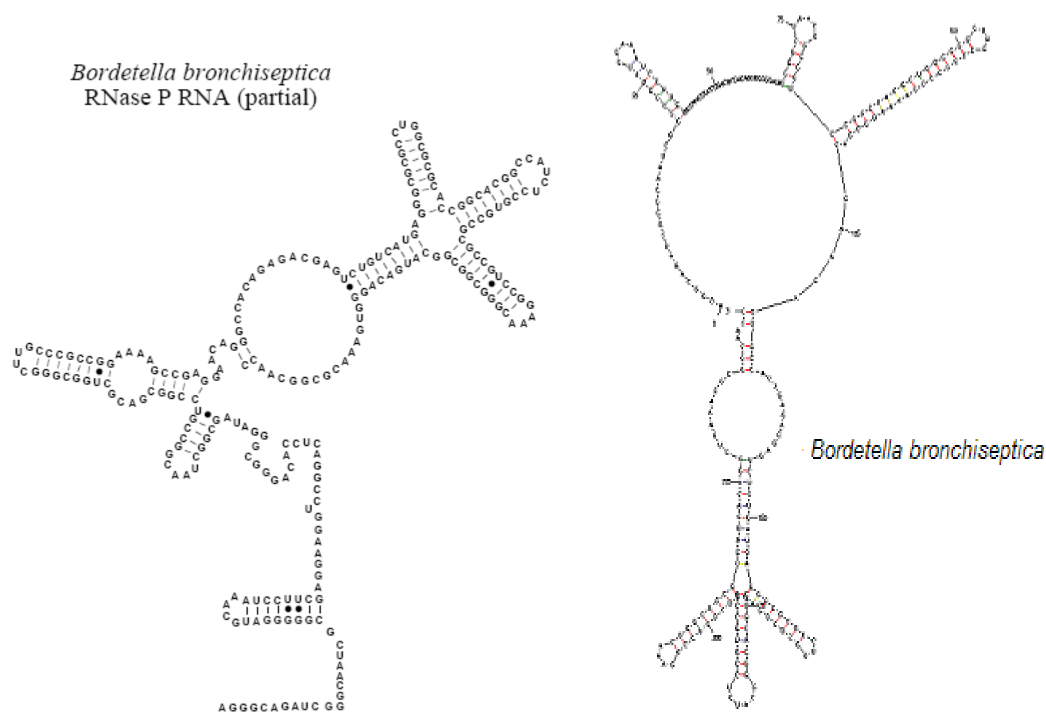


Figure 5.1 The secondary structure of *Bordetella bronchiseptica*. On the left side, the graph is excerpted from RNase P RNA database [BRO99] [HWF01]. On the right side, the graph is generated by MFOLD's sir_graph utility [ZMT98] using the secondary structure's CT file from RNase P database.

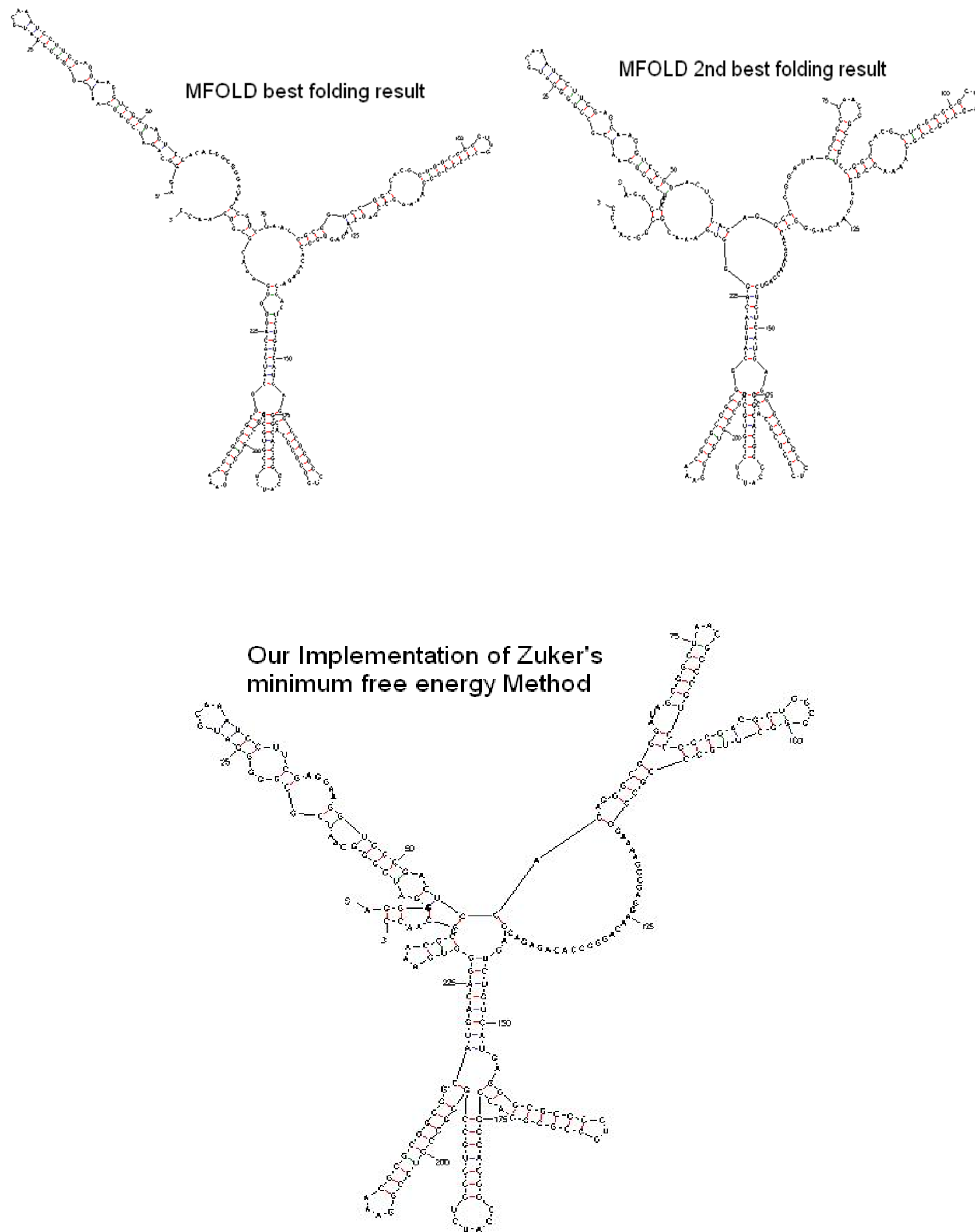
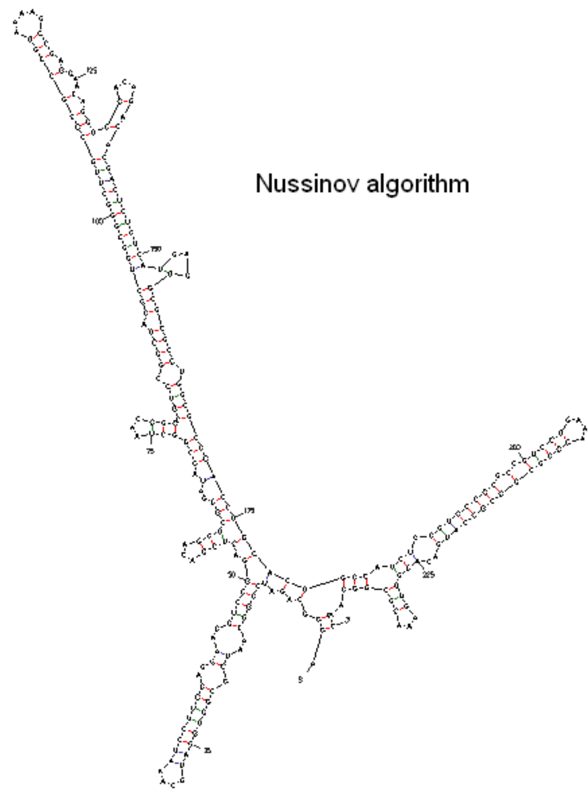


Figure 5.2 RNA secondary structure comparison of MFOLD vs. our implementation of Zuker's minimum free energy method. The test RNA is *Bordetella bronchiseptica*. The upper left figure shows the best folding result from MFOLD. The upper right figure shows the 2nd best folding result from MFOLD. The bottom figure shows the folding result of our implementation of Zuker's minimum free energy method.



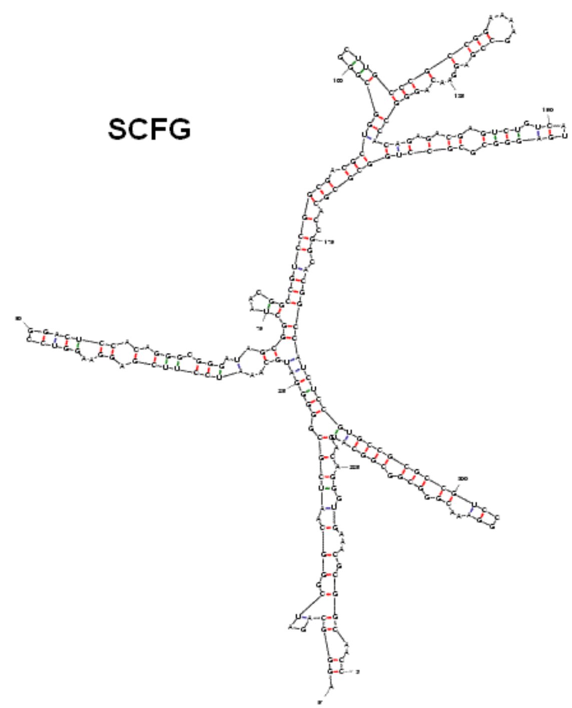


Figure 5.3 RNA Secondary structure folded by the Nussinov algorithm and SCFG algorithm. The test RNA is *Bordetella bronchiseptica*.

Figure 5.2 shows the folding results of the same RNA using MFOLD and our implementation of Zuker's method. In these figures, the minimum free energy of the optimal secondary structure folded by MFOLD is -126.3 kcal/mole. For the best suboptimal results folded by MFOLD, it is -128.50 kcal/mole. The minimum free energy of the secondary structure fold by our implementation is -170.0 kcal/mole.

As expected, the minimum free energy of our folding program is lower than MFOLD's. Most free energy functions used by our implementation of Zuker algorithm are the same as those used by MFOLD. The energy function for bifurcation loop is different. The minimum free energy algorithm used by MFOLD is a modified version based on the one we use [ZS81]. The bifurcation loop's energy function, which has a penalty (a positive energy) used in MFOLD can not be used in our implementation. Instead, we ignored the penalty energy in a bifurcation. MFOLD provides a tool, efn to calculate minimum free energy of a given secondary structure according to MFOLD's energy functions and algorithms [ZMT98]. If we use that tool to calculate the secondary structure we folded, the energy is -67.2 kcal/mole.

5.2 Methods to Compare Secondary Structures

This section will describe the coarse-grained tree representation and the string notation of the RNA secondary structure. These two representations are often used in the comparison of RNA secondary structures.

5.2.1 Tree Representation of RNA Secondary Structure

It is interesting to know the similarity of the secondary structures in Figure 1.3 and Figure 1.4. But before we can tell how similar they are, we need to define what we mean by the similarity of two RNA secondary structures. In fact, this is a general problem if we want to compare results of various RNA folding algorithms, or compare results from folding algorithms to experimental results. The tree representation of RNA secondary structure is a way to compare the similarity of secondary structures.

It is generally assumed that the members of the same RNA family preserve their RNA secondary structure more than they preserve their primary sequences [KR03], and similar structures provide similar functions. Therefore, a method to abstracting the RNA secondary structure, rather than the primary sequence is desired for the structures' comparison. Figure 5.4 shows an abstract tree which represents a secondary structure [SZ90] [BIL03]. Using the tree representation, we can focus on elements' structure of hairpin loop, stack region, bulge loop, interior loop and bifurcation loop. Several methods have been developed to compare the similarity of these tree representations.

Figure 5.4 RNA secondary structure and its tree representation. The graph on the left is a secondary structure with each elements labeled as Hairpin, Bulge, Interior loop and Multi-loop (bifurcation). The graph on the right is the structure's corresponding tree representation, with N standards for the starting point of the structure (5') and other letters represent elements' structure.

5.2.2 String Notation of the Tree Representation

A tree representation can be expressed by a string notation [SHA88]. The rule of recursively writing a string is

(root (subtree-1) (subtree-2) … (subtree-n))

thus, the tree in Figure 5.3 is written as (N(M(H)(I(B(H))))), or simply (N(M(H)(IBH))).

The string notation allows us to align two strings and do the comparisons, just like the sequence alignment. The following is an example:

```
(NIBII(M(I   (M( IH)(  IIIIIH)))(H)))
 ||||| | |    |  ||          |    |
(NIBII(M(IB(M(M(BIH)(H)(H))(H)))(H)))
```

The string notation alignment is slightly different from the sequence alignment. Not only the structure element N, H, B, I, M should be aligned; but also the parenthesis should be aligned, because the parenthesis are part of the structures.

5.2.3 Editing Distance of Tree Representations

Another way to compare two tree representations is to look at the edit distance. An edit distance [BIL03] is defined as the minimum number of edit operations

needed to transfer one tree to another. There are three types of edit operations: re-labeling, insertion and deletion. Re-labeling changes one type of element to another type. Insertion inserts an element. Deletion removes an element. Figure 5.5 gives an example of these three edit operations [BIL03].

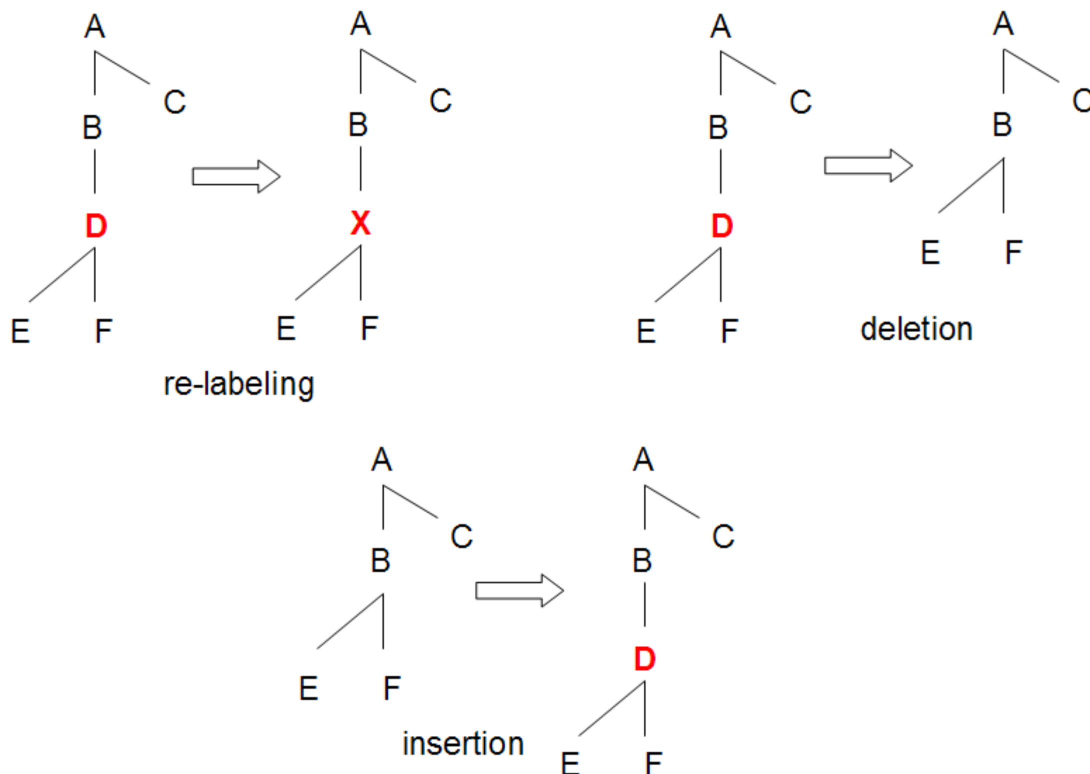


Figure 5.5 Edit operations of a tree. Three edit operations are defined. They are re-labeling, insertion and deletion.

A sequence of edit operations to convert a tree T_1 to another tree T_2 forms a cost function. This cost function, or distance between T_1 and T_2 describes the similarity of T_1 and T_2.

The above tree representation is often referred to as coarse-grained tree representation. More information can be added into tree representations. For example, it is possible to add the size of hairpin loop and helical stem, size of loop components, or even the actual sequence into a tree representation. However, much of this detailed information is sometimes irrelevant to the functionalities of RNA. Working with this type of fine-grained tree representation often causes the loss of focus to major structure features. For this reason, most researchers choose to use coarse-grained tree representation, and only add certain detail information when needed.

The Vienna RNA Package [HFS94] provides several tools for studying the RNA secondary structure. Among all the tools provided by the Vienna RNA package, RNAdistance and RNAforester do compare edit distance of secondary structures. RNAdistance is a command line tool running on major UNIX platforms as well as Microsoft Windows. It accepts structures in bracket dot format and coarse grained representations. RNAforester provides a web interface and a web

service interface for the comparison.

5.3 Comparison Using Tree Representations and String Notations

Table 5.1 lists the comparison of various folding algorithms for the *Bordetella bronchiseptica*. We use the RNAdistance program in VIENNA package to calculate edit distance of tree representations and alignment of string representations. We compare both coarse-grained representations and full-grained representations. Note that the results of SCFG are based on a set of pre-defined probabilities that do not reflect the situation in the real world.

Our implementation of Zuker's minimum free energy method is better than the Nussinov algorithm and SCFG with Nussinov rules. We notice that MFOLD's second best result is actually closer to the result in RNase P database. And our energy folding results is closer to the second best result of MFOLD in all type of comparison except with coarse-grained string notation.

Bordetella bronchiseptica 243 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD best	33	28	114	66	N/A	N/A	N/A	N/A
My Zuker	49	40	150	76	30	23	128	64
Nussinov	71	67.5	208	130	73	55	184	110
SCFG	99	96	234	138	91	81	214	115

Bordetella bronchiseptica 243 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD 2nd best	24	18.5	100	57	N/A	N/A	N/A	N/A
My Zuker	49	40	150	76	25	25	102	56
Nussinov	71	67.5	208	130	63	57	182	115
SCFG	99	96	234	138	95	87	208	123

Table 5.1 Comparison of various folding results with RNase P database and MFOLD. The folding structures in RNase P database are based on non-conventional method for tRNA. The symbols in the third row of each table indicate the following: -Dc stands for edit distance of coarse-grained tree representation. –DC stands for string alignment of coarse-grained string notation. –Df stands for edit distance of full-grained tree representation, and –DF stands for string alignment of full-grained string notation. For the results of MFOLD, we use both the optimal folding and a sub-optimal folding which is the closest to our folding, and within 5% in energy comparing to the best folding of MFOLD. In the above table, this sub-optimal folding happens to be the second best folding.

Table 5.2 shows the comparison with several other tRNAs in the RNase P database.

Heliobacterium chlorum 342 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD	61	38.5	264	152	N/A	N/A	N/A	N/A
My Zuker	58	44.5	276	152	64	41.5	214	113
Nussinov	121	104	368	188	103	91.5	240	124
SCFG	135	119.5	338	180	135	112.5	308	185

Heliobacillus mobilis 342 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD	65	41.5	278	156	N/A	N/A	N/A	N/A
My Zuker	66	47	272	171	54	42	244	116
Nussinov	124	104	360	190	92	86	222	129
SCFG	119	103.5	392	198	96	94.5	232	125

Methanothermus fervidus 244 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD	22	18	124	65	N/A	N/A	N/A	N/A
My Zuker	37	37	138	68	40	37	142	69
Nussinov	71	69.5	214	110	62	61.5	180	95
SCFG	95	89	236	112	86	83.5	216	117

Halobacterium sp. 375 nucleotides								
Algorithm	Compare to RNase P database				Compare to MFOLD package			
	-Dc	-DC	-Df	-DF	-Dc	-DC	-Df	-DF
MFOLD	57	40	218	120	N/A	N/A	N/A	N/A
My Zuker	78	60.5	310	174	71	46.5	276	146
Nussinov	123	110	322	171	113	100	280	154
SCFG	130	130	292	167	127	122	262	140

Table 5.2 Comparison of folding results of several other RNAs.

5.4 Comparison of Run Time

The direct comparison of complexity among different algorithms is not meaningful because it depends on the data type and mathematical functions used by those algorithms. For example, the Nussinov algorithm use integer numbers only. The SCFG algorithm with Nussinov rules use floating point numbers and logarithm function. Even though these two algorithms have the same time complexity of $O(n^3)$, SCFG uses more CPU time than Nussinov algorithm. However, we can

still use various length RNAs to test our implementations' time complexity. Table 5.3 is a measurement of CPU time of our implementation of Zuker's method, using RNAs of length 243, 1179 and 2224. Table 5.4 and 5.5 are similar measurements for the Nussinov algorithm and SCFG algorithm. These results are consistent with the predicted time complexity of the three algorithms.

RNA length	243	1179	2224
CPU time	8.76s	86m19.37s	1089m13.53s
Projected CPU time based on $O(n^4)$ and CPU time usage of the previous RNA	N/A	80m54.36s	1093m32.34s

Table 5.3 Measurement of CPU time usage vs. RNA length for our implementation of Zuker's minimum free energy method. The last row is the projected CPU time usage based on the algorithm's time complexity and the CPU time usage of the previous RNA.

RNA length	243	1179	2224
CPU time	0.06s	7.18s	49.16s
Projected CPU time based on $O(n^3)$ and CPU time usage of the previous RNA	N/A	6.85s	48.02s

Table 5.4 Measurement of CPU time usage vs. RNA length for our implementation of the Nussinov algorithm. The last row is the projected CPU time usage based on the algorithm's time complexity and the CPU time usage of the previous RNA.

RNA length	243	1179	2224
CPU time	0.11s	13.57s	94.20s
Projected CPU time based on $O(n^3)$ and CPU time usage of the previous RNA	N/A	12.56s	91.08s

Table 5.5 Measurement of CPU time usage vs. RNA length for our implementation of the SCFG algorithm. The last row is the projected CPU time usage based on the algorithm's time complexity and the CPU time usage of the previous RNA.

We also compare the CPU time usage of recursive calculation and non-recursive calculation. We generate random sequences of length 150, 300 and 500. We use our implementation of the Zuker's method as an example. Table 5.6 lists the results. It clearly shows that recursion is expensive.

Sequence Length	CPU time without recursion	CPU Time with recursion
150	2.41	4.04
300	21.77	47.91
500	163.68	392.28

Table 5.6 CPU time usage comparison of recursive function calls and non-recursive function calls. The above data is measured using our implementation of Zuker's minimum free energy method.

For our Zuker method implementation without recursion, we use OpenMP to parallelize the inner loop that calculates Wij and Vij, as indicated by the "#progma" directive in our code. We test the time usage of two RNA sequence of length 1179 and 2224. The CPU time and run time usage results are listed in Table 5.7. The result shows a reduction of run time when using OpenMP with multiple threads. It is understandable that using more OpenMP thread does not reduce the elapse time proportionally, due to the cost of thread generation and synchronization. But we do not have a reason to explain CPU time usages under four OpenMP threads are shorter than the total CPU time usage under one thread.

Length vs Time / Number of OpenMP Threads	1179		2224	
	CPU time	Elapsed time	CPU time	Elapsed time
1	86m19.37s	86m22.05s	1089m13.53s	1089m52.28s
2	88m14.94s	45m35.40s		
4	84m27.93s	24m25.45s	1061m20.51s	299m12.98s

Table 5.7 CPU time and wall clock time measurement when parallelizing the inner loop of the Zuker's minimum free energy calculation with OpenMP.

The above timing measurement is performed on a machine with 2.2 GHz dual-core dual chip AMD Opteron 275 processor and 1MB L2 cache per core, 8GB memory. It ran 64 bit RedHat Enterprise Linux version 4. The code itself was compiled to an ELF 32 bit binary by Intel C/C++/Fortran compiler suite version 9.1.

6. CONCLUSION

In this report, we study several RNA secondary structure prediction algorithms, describe our software framework design, and discuss the details of our implementations' details. We also present our analysis of folding accuracy, CPU time usage, and performance improvement.

The Nussinov algorithm tries to maximize base pairs. It is one of the simplest algorithms, yet it has fundamental influence to the later computer prediction algorithms. The SCFG version of the Nussinov algorithm demonstrates the concept of using Stochastic Context Free Grammar to address the issue. It is a powerful method that allows the incorporation of various information sources for computational prediction. The SCFG algorithm has its unique problems such as the strategy of information incorporation, language ambiguity, etc. The Zuker algorithm focuses on minimizing free energy. It classifies a secondary structure into graph elements, and defines energies to the elements rather than defines energy on base pairs.

Our implementation framework uses common format for input and output. FASTA format is used for the input RNA sequence. CT format and DOT format are used for the output of the folding results.

Our implementation keeps as close as possible to the original Nussinov algorithm, though we simplify the recursive formula in the original algorithm. We also implement a variant of Nussinov algorithm that traces all possible best folding results. For SCFG version of the Nussinov algorithm, we introduce a tracing matrix to store the actual parsing rules used during the recursion. Using the tracing matrix allows us to avoid testing equality of two floating point numbers during traceback stage. We make some modifications to the original recursion and the trace-back accordingly. In the implementation of the Zuker's minimum free energy method, we use similar but more complex modifications to trace the actual folding. We reuse most of the energy data from Zuker's MFOLD package and the corresponding FORTRAN file that reads the data.

Though all these algorithms are implemented with recursive function calls, we actually introduce a method to effectively do the calculation without recursion. The goal is to reduce the overhead associated with recursion, and to parallelize the calculation.

Besides the visual comparison of folding results, we introduce several other commonly used comparison methods based on coarse-grained tree representation and string representation. Edit distance and string alignment are used respectively upon these representations to describe the similarity of RNA secondary structures.

We compare our implementation results with those secondary structures in the RNase P database and with the folding results from the state-of-the-art MFOLD package. Comparison shows that results of our implementation of the Zuker's minimum free energy are somehow close to the results of MFOLD's optimal or suboptimal results. And these results are much better than the folding results from the original Nussinov algorithm and SCFG algorithm with Nussinov rules.

The result of SCFG with Nussinov rules is not as good as the Nussinov algorithm. This is expected due to the fact that the probabilities used in the SCFG are pre-defined random numbers that do not reflect the real world situation. Our comparison also shows the distance between any computational predicted secondary structures and real world secondary structures.

By measuring the CPU time usage under different lengths of RNAs, we demonstrate that our implementations are consistent with the predicted time complexity of $O(N^3)$ for Nussinov algorithm and SCFG algorithm, and $O(N^4)$ for the Zuker's minimum free energy method. In addition, we find that non-recursive calculations significantly reduce the CPU time usage over recursive calculations. In other word, recursion produces significant overhead in these algorithms. Using non-recursive calculations, we are able to further reduce wall clock time by parallelize the computation using OpenMP.

APPENDICES

Appendix A: Reference

[AKU00] Akutsu, T. (2000). Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics,* 104: 45-62.

[BIL03] Bille, P. (2003). Tree Edit Distance, Alignment Distance and Inclusion, *The IT University of Copenhagen Technical report serial.* TR-2003-23

[BRO99] Brown, J.W. (1999). The Ribonuclease P Database. *Nucl. Acids res.* 27:314 http://www.mbio.ncsu.edu/RNaseP/seqs&structures.html

[DEK98] Durbin, R., Eddy, S. R., Krogh, A. and Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids,* Cambridge University Press.

[DE04] Dowell, R.D., Eddy, S.R. (2004). Evaluation of several lightweight stochastic context-free grammars. *BMC Bioinformatics,* 5, 71.

[HFS94] Hofacker, I.L., Fontana, W., Stadler, P.F., Bonhoeffer, S., Tacker and M., Schuster, P. (1994). Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte f. Chemie,* 125, 167-188.

[HWF01] Harris, J.K., Haas, E.S., Williams, D., Frank, D.N., Brown, J.W. New insight into RNase P RNA structure from comparative analysis of the archaeal RNA. *RNA*. 2001 February; 7(2): 220–232.

[KR03] Krane, D.and Raymer, M. (2003). *Fundamental Concepts of Bioinformatics*, Benjamin Cummins Press, San Francisco.

[NPG78] Nussinov, R., Pieczenik, G., Griggs, J. R. and Kleitman, D.J. (1978). Algorithms for Loop Matchings. *SIAM Journal of Applied Mathematics,* 35, 68-82.

[SBH94] Sakakibara, Y., Brown, M., Hughey, R., Mian, I.S., Sjolander, K., Underwood, R.C., Haussler, D.(1994). Sotchastic context-free grammers for tRNA modeling. *Nucleic Acids Research*, 22(23), 5112-512.

[SHA88] Shapiro, B.A. (1988). An algorithm for comparing multiple RNA secondary structures. *CABIOS* 4, 381-393.

[SZ90] Shapiro, B.A., Zhang, K. (1990). Comparing multiple RNA secondary structures using tree comparison. *CABIOS,* 6, 309-318.

[TCG98] Tabaska, J.E., Cary, R.B., Gabow, H.N. and Stormo, G.D. (1998). An RNA folding method capable of identifying pseudoknots and base triples. *Bioinformatics* 14(8), 691-699

[WDG03] Wiese, K.C., Deschenes, A. and Glen, E. (2003). Permuation-based RNA Secondary Structure Prediction via a Genetic Algorithm. *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, 335-342. Canberra, Dec. 8-12, 2003.

[ZMT98] Zuker, A.M., Mathews, B.D.H., Turner, C.D.H. (1998). Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide. http://www.bioinfo.rpi.edu/ ~zukerm/seqanal/

[ZS81] Zuker, M. and Stiegler, P. (1981). Optimal Computer Folding of Larger RNA Sequences Using Thermodynamics and Auxillary Information. *Nucleic Acids Research,* 9(1), 133-148.

Appendix B: Reference Source Code from MFOLD [ZMT98]: maxn2.inc

```
c maxn is also defined in ct.h as MAXLEN
      parameter (maxn=3000)
```

Appendix C: Reference Source Code from MFOLD [ZMT98]: rna_or_efn.inc

```
      implicit integer (a-z)
      include 'maxn2.inc'
      parameter (fldmax=2*maxn)
      parameter (infinity=999999,sortmax=90000)
      parameter (sortmaxp1=sortmax+1)
      parameter (mxbits=(maxn*(maxn+1)+29)/30)
      parameter (maxtloops=100)
      parameter (maxtriloops=50)
      parameter (maxsiz=200000)

      character*1 seq(maxsiz),aux(maxn),lorc
      character*4 usage
      character*50 seqlab

      integer vst(maxn*maxn),wst(maxn*maxn)
      integer*2 marks(mxbits),force2(2*mxbits)

      integer maxpen,newnum(maxsiz),hstnum(fldmax),force(fldmax),
     .
strand(fldmax),numseq(fldmax),work(fldmax,0:2),wmb(fldmax,0:2),
     .        w5(-1:maxn),w3(maxn+2),cntrl(10),nsave(2),list(3000,4),
     .        listsz,basepr(maxn),heapi(sortmaxp1),heapj(sortmaxp1),
     .        n,break,vmin,num,numtloops,numtriloops,maxbp
      integer asint3(6,6,5,5,5),asint5(6,6,5,5,5,5,5),bulge(30),
     .
dangle(5,5,5,2),eparam(16),hairpin(30),inter(30),poppen(4),
     .        sint2(6,6,5,5),sint4(6,6,5,5,5,5),sint6(6,6,25,5,5,5,5),
     .        stack(5,5,5,5),tloop(maxtloops,2),triloop(maxtriloops,2),
     .        tstkh(5,5,5,5),tstki(5,5,5,5)

      real prelog,prec/100.0/

      common /chars/ seq,aux,lorc,usage,seqlab

      common /bigstuff/ vst,wst,marks,force2

      common /main/
maxpen,newnum,hstnum,force,strand,numseq,work,wmb,w5,
     .     w3,cntrl,nsave,list,listsz,basepr,heapi,heapj,n,break,vmin,
     .     num,numtloops,numtriloops,maxbp
      common /efiles/
asint3,asint5,bulge,dangle,eparam,hairpin,inter,poppen,
     .        sint2,sint4,sint6,stack,tloop,triloop,tstkh,tstki,prelog

c      common /list/ list,listsz
c      common /traceback/ basepr
c      common /heap/ heapi,heapj,num
c      common /bits/marks,force2
```

Appendix D: Reference Source Code from MFOLD [ZMT98]: efiles.f

```
c     Reads energy file names and open the files for reading.
      subroutine enefiles
      character*80 filen,path

      call getenv('MFOLDDAT',path)
      in = index(path,' ')
      if (path.eq.'     ') then
         call getenv('MFOLD',path)
         in = index(path,' ')
         path(in:in+4) = '/dat/'
      else
         path(in:in) = '/'
      endif

      filen = 'asint1x2.dat'
      open(8,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'asint2x3.dat'
c      open(9,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'dangle.dat'
      open(10,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'loop.dat'
      open(11,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'miscloop.dat'
      open(32,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'sint2.dat'
      open(33,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'sint4.dat'
      open(34,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'sint6.dat'
c      open(35,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'stack.dat'
      open(12,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'tloop.dat'
      open(29,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'triloop.dat'
      open(39,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'tstackh.dat'
      open(13,file=path(1:index(path,' ')-1)//filen,status='old')
      filen = 'tstacki.dat'
      open(14,file=path(1:index(path,' ')-1)//filen,status='old')

2     return
      end

c     Reads energy files.
      subroutine ergread

      include 'rna_or_efn.inc'
      logical  endfile,zfind
      character*96 inrec
      character*6 temp
      real a,b,c,d,rj,rk

c     TLoop INFORMATION IN
```

```
      call gettloops

c     TriLoop INFORMATION IN
      call gettri

c     Get misc loop info
      if(zfind(32,'-->')) then
         write(6,*) 'STOP: Premature end of miscloop file'
         call exit(1)
      endif
      read (32,*) prelog
      prelog = nint(prelog*prec)
c     asymmetric internal loops: the ninio equation
      if(zfind(32,'-->')) then
         write(6,*) 'STOP: Premature end of miscloop file'
         call exit(1)
      endif
      read (32,*) a
      maxpen = nint(a*prec)
      if(zfind(32,'-->')) then
         write(6,*) 'STOP: Premature end of miscloop file'
         call exit(1)
      endif
      read (32,*) a,b,c,d
      poppen(1) = nint(a*prec)
      poppen(2) = nint(b*prec)
      poppen(3) = nint(c*prec)
      poppen(4) = nint(d*prec)
c     Set default values of eparam.
      eparam(1) = 0
      eparam(2) = 0
      eparam(3) = 0
      eparam(4) = 0
      eparam(7) = 30
      eparam(8) = 30
c      eparam(9) = -500 Bonus energies are no longer used!
c     multibranched loops
      if(zfind(32,'-->')) then
         write(6,*) 'STOP: Premature end of miscloop file'
         call exit(1)
      endif
      read (32,*) a,b,c
      eparam(5) = nint(a*prec)
      eparam(6) = nint(b*prec)
      eparam(9) = nint(c*prec)
c     efn2 multibranched loops
      if(zfind(32,'-->')) then
c     Version 2.3 rules and DNA rules do not need extra parameters
         write(6,*) 'End of miscloop file. Parameters 10 -> end set to
0.'
         do k= 10,16
            eparam(k) = 0
         enddo
      else
         read (32,*) a,b,c
c        Don't need these parameters yet in nafold
c        terminal AU penalty
```

```
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(10) = nint(a*prec)
c        bonus for GGG hairpin
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(11) = nint(a*prec)
c        c hairpin slope
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(12) = nint(a*prec)
c        c hairpin intercept
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(13) = nint(a*prec)
c        c hairpin of 3
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(14) = nint(a*prec)
c        Intermolecular initiation free energy
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) a
         eparam(15) = nint(a*prec)
c        GAIL (Grossly Asymmetric Interior Loop) Rule (on/off <-> 1/0)
         if(zfind(32,'-->')) then
            write(6,*) 'STOP: Premature end of miscloop file'
            call exit(1)
         endif
         read (32,*) eparam(16)
      endif

c     DANGLE IN

      do a = 1,5
        do b = 1,5
          do c = 1,5
            do d = 1,2
              dangle(a,b,c,d) = 0
            enddo
```

```
          enddo
        enddo
      enddo
      endfile = zfind(10,'<--')
      if (.not.endfile) then
         do var4 = 1,2
            do var1 = 1,4
               if (endfile) goto 150
               read(10,100,end=150,err=91) inrec
               do var2 = 1,4
                  do var3 = 1,4
                     j = 0
                     tstart = (var2-1)*24 + (var3-1)*6 + 1
                     temp = inrec(tstart:tstart+5)
                     do i = 2,5
                        if (temp(i-1:i+1).eq.' . ') j = infinity
                     enddo
                     if (temp(1:1).eq.'.'.or.temp(6:6).eq.'.') j =
infinity
                     if (j.eq.0) then
                        read(temp,50) rj
                        dangle(var1,var2,var3,var4) = nint(prec*rj)
                     endif
                  enddo
               enddo
               endfile = zfind(10,'<--')
            enddo
         enddo
      else
         write(6,*) 'STOP: DANGLE ENERGY FILE NOT FOUND'
         call exit(1)
      endif

 50   format(f6.2)
 100  format(a96)
      goto 200

 150  write(6,*) 'STOP: PREMATURE END OF DANGLE ENERGY FILE'
      call exit(1)


c    INTERNAL,BULGE AND HAIRPIN IN

 200  endfile = zfind(11,'---')
      if (endfile) then
         write(6,*) 'STOP: --- header not found in loop energy file'
         call exit(1)
      endif
      i = 1
 201  read(11,100,end=300) inrec
      j = -1
      do ii = 1,3
         j = j + 6
         do while (inrec(j:j).eq.' ')
            j = j + 1
         enddo
         temp = inrec(j:j+5)
```

```
            k = 0
            do jj = 2,4
               if (temp(jj-1:jj+1).eq.' . ') k = infinity
            enddo
            if (temp(1:1).eq.'.'.or.temp(6:6).eq.'.') then
               k = infinity
            endif
            if (k.eq.0) then
               read(temp,50) rk
               if (ii.eq.1) inter(i) = nint(prec*rk)
               if (ii.eq.2) bulge(i) = nint(prec*rk)
               if (ii.eq.3) hairpin(i) = nint(prec*rk)
            endif
         enddo
         i = i + 1
         if (i.le.30) goto 201

c        STACK IN

300      do a = 1,5
           do b = 1,5
             do c = 1,5
               do d = 1,5
                 stack(a,b,c,d) = infinity
               enddo
             enddo
           enddo
         enddo
         endfile = zfind(12,'<--')
         if (.not.endfile) then
            do var1 = 1,4
               do var3 = 1,4
                  if (endfile) goto 350
                  read(12,100,end=350,err=92) inrec
                  do var2 = 1,4
                     do var4 = 1,4
                        j = 0
                        tstart = (var2-1)*24 + (var4-1)*6 + 1
                        temp = inrec(tstart:tstart+5)
                        do i = 2,5
                           if (temp(i-1:i+1).eq.' . ') j = infinity
                        enddo
                        if (temp(1:1).eq.'.'.or.temp(6:6).eq.'.') j =
infinity
                        if (j.eq.0) then
                           read(temp,50) rj
                           stack(var1,var2,var3,var4) = nint(prec*rj)
                        endif
                     enddo
                  enddo
               enddo
               endfile = zfind(12,'<--')
            enddo
         else
            write(6,*) 'STOP: STACK ENERGY FILE NOT FOUND'
            call exit(1)
         endif
```

```
      call stest(stack,'STACK ')

      goto 400

 350   write(6,*) 'STOP: PREMATURE END OF STACK ENERGY FILE'
       call exit(1)

400   do a = 1,5
        do b = 1,5
          do c = 1,5
            do d = 1,5
              tstkh(a,b,c,d) = 0
            enddo
          enddo
        enddo
      enddo
      endfile = zfind(13,'<--')
      if (.not.endfile) then
         do var1 = 1,4
            do var3 = 1,4
               if (endfile) goto 350
               read(13,100,end=450,err=93) inrec
               do var2 = 1,4
                  do var4 = 1,4
                     j = 0
                     tstart = (var2-1)*24 + (var4-1)*6 + 1
                     temp = inrec(tstart:tstart+5)
                     do i = 2,5
                        if (temp(i-1:i+1).eq.' . ') j = infinity
                     enddo
                     if (temp(1:1).eq.'.'.or.temp(6:6).eq.'.') j =
infinity
                     if (j.eq.0) then
                        read(temp,50) rj
                        tstkh(var1,var2,var3,var4) = nint(prec*rj)
                     endif
                  enddo
               enddo
            enddo
            endfile = zfind(13,'<--')
         enddo
      else
         write(6,*) 'STOP: TSTACKH ENERGY FILE NOT FOUND'
         call exit(1)
      endif

c**   CALL STEST(TSTK,'TSTACK')
      goto 500

 450  write(6,*) 'STOP: PREMATURE END OF TSTACKH ENERGY FILE'
      call exit(1)
 500  do a = 1,5
        do b = 1,5
          do c = 1,5
            do d = 1,5
              tstki(a,b,c,d) = 0
            enddo
```

```
          enddo
        enddo
      enddo

      endfile = zfind(14,'<--')
      if (.not.endfile) then
         do var1 = 1,4
            do var3 = 1,4
               if (endfile) goto 450
               read(14,100,end=550,err=94) inrec
               do var2 = 1,4
                  do var4 = 1,4
                     j = 0
                     tstart = (var2-1)*24 + (var4-1)*6 + 1
                     temp = inrec(tstart:tstart+5)
                     do i = 2,5
                        if (temp(i-1:i+1).eq.' . ') j = infinity
                     enddo
                     if (temp(1:1).eq.'.'.or.temp(6:6).eq.'.') j =
infinity
                     if (j.eq.0) then
                        read(temp,50) rj
                        tstki(var1,var2,var3,var4) = nint(prec*rj)
                     endif
                  enddo
               enddo
            enddo
            endfile = zfind(14,'<--')
         enddo
      else
         write(6,*) 'STOP: TSTACKI ENERGY FILE NOT FOUND'
         call exit(1)
      endif

c**   call stest(tstki,'TSTACKI')
      goto 600

550   write(6,*) 'STOP: PREMATURE END OF TSTACK ENERGY FILE'
      call exit(1)

c  Read in symmetric interior loop energies
600   call symint

c  Read in asymmetric interior loop energies
      call asmint

      call symtest

      close(8)
      close(9)
      close(10)
      close(11)
      close(12)
      close(13)
      close(14)
      close(33)
      close(34)
```

```
      close(35)
      close(39)

      return
 91   write(6,*) 'STOP: ERROR reading dangle energy file'
      call exit(1)

 92   write(6,*) 'STOP: ERROR reading stacking energy file'
      call exit(1)

 93   write(6,*) 'STOP: ERROR reading tstackh energy file'
      call exit(1)

 94   write(6,*) 'STOP: ERROR reading tstacki energy file'
      call exit(1)
      end

c     Symmetry test on stacking and terminal stacking energies.
c     For all i,j,k,l between 1 and 4, stack(i,j,k,l) MUST equal
c     stack(l,k,j,i). If this fails at some i,j,k,l; these numbers
c     are printed out and the programs grinds to an abrupt halt!
      subroutine stest(stack,sname)
      integer stack(5,5,5,5),a,b,c,d
      character*6 sname

      do a = 1,4
        do b = 1,4
          do c = 1,4
            do d = 1,4
              if (stack(a,b,c,d).ne.stack(d,c,b,a)) then
                 write(6,*) 'SYMMETRY ERROR in stack'
                 write(6,101) sname,a,b,c,d,stack(a,b,c,d)
                 write(6,101) sname,d,c,b,a,stack(d,c,b,a)
                 call exit(1)
              endif
            enddo
          enddo
        enddo
      enddo
      return
101   format(5x,a6,'(',3(i1,','),i1,') = ',i10)
      end

c     Writes out the numbers in the energy arrays of the folding
program.
      subroutine out(u)
      include 'rna_or_efn.inc'
      integer*2 tlptr,key,bptr,nbase
      character*4 tlbuf

c     used for testing contents of energy arrays only
c     not used in the mature program

      write(u,100) 'DANGLE'
      do var4 = 1,2
        do var1 = 1,4
                write(u,101) ((dangle(var1,var2,var3,var4),var3=1,4),
```

```
.                            var2=1,4)
    write(6,103)
  enddo
  write(6,104)
enddo

write(u,100) 'TSTACKH'
do var1 = 1,4
  do var3 = 1,4
            write(u,101) ((tstkh(var1,var2,var3,var4),var4=1,4),
.                            var2=1,4)
    write(6,103)
  enddo
  write(6,104)
enddo

write(u,100) 'TSTACKI'
do var1 = 1,4
  do var3 = 1,4
            write(u,101) ((tstki(var1,var2,var3,var4),var4=1,4),
.                            var2=1,4)
    write(6,103)
  enddo
  write(6,104)
enddo

write(u,100) 'STACK'
do var1 = 1,4
  do var3 = 1,4
            write(u,101) ((stack(var1,var2,var3,var4),var4=1,4),
.                            var2=1,4)
    write(6,103)
  enddo
  write(6,104)
enddo

write(u,200) 'INTER','BULGE','HAIRPIN'
do i = 1,30
  write(u,201) i,inter(i),bulge(i),hairpin(i)
enddo

write(u,100) 'TLoops'
do tlptr=1,numtloops
key=tloop(tlptr,1)
   do bptr=1,4
      nbase=mod(key,8)
      key=key/8
      if (nbase.eq.1) then
         tlbuf(bptr:bptr)='A'
      elseif (nbase.eq.2) then
         tlbuf(bptr:bptr)='C'
      elseif (nbase.eq.3) then
         tlbuf(bptr:bptr)='G'
      else
         tlbuf(bptr:bptr)='U'
     endif
   enddo
```

```
      write(u,205) tlbuf,tloop(tlptr,2)
      enddo
      return
100   format(//,a40,//)
101   format(16i6)
103   format(' ')
104   format(/)
200   format(3a20,/,60('-'),/)
201   format(i4,i16,2i20)
205   format(a4,2x,i8)
      end

c     Used in reading the energy files.
c     Locates markers in the energy files so that data can be read
c     properly.
      function zfind(unit,str)
      implicit integer (a-z)
      logical zfind
      character*3 str
      character*96 inrec

      read(unit,100,end=200) inrec
      do  while (index(inrec,str).eq.0)
         read(unit,100,end=200) inrec
      enddo
      zfind = .false.
      return
 100  format(a96)
 200  zfind = .true.
      return
      end

      subroutine gettloops
c------------Kevin added below------------------------------------------
      include 'rna_or_efn.inc'
      integer flag,i,j,numseqn(6)
      real energy
      character row*80,seqn*6

 2020 format(a80)
 2021 format(1x,a6,1x,f6.2)

      flag=0
      do while(flag.eq.0)
         read(29,2020,err=91)row
         if(index(row,'---').gt.0)then
            flag=1
         endif
      enddo

      flag=0
      j=0
      do while(flag.eq.0)
         j=j+1
         read(29,2021,end=99,err=91)seqn,energy

         if(index(seqn,'    ').gt.0.or.j.eq.maxtloops)then
```

```
            flag=1
            numtloops=j-1
         else
            do i=1,6
               call letr2num(seqn(i:i),numseqn(i))
            enddo

            tloop(j,1)=((((numseqn(6)*8+numseqn(5))*8+numseqn(4))*8+
     .       numseqn(3))*8+numseqn(2))*8+numseqn(1)
            tloop(j,2)=nint(prec*energy)
         endif
      enddo

      close(29,status='KEEP')

      return

 91   write(6,*) 'STOP: ERROR reading tetraloop file'
      call exit(1)

 99   close(29,status='KEEP')
      numtloops=j-1

      return

      end

c---------------Kevin added above------------------------------------

c   Reads sequence and energy for triloop.d__ and stores info in array
triloop
      subroutine gettri

      include 'rna_or_efn.inc'
      integer i,j,flag,numseqn(5)
      real energy
      character row*80,seqn*5

 3030 format(a80)
 3031 format(1x,a5,2x,f6.2)

      flag=0
      do while(flag.eq.0)
         read(39,3030,err=91)row
         if(index(row,'---').gt.0)then
            flag=1
         endif
      enddo
      flag=0
      i=0
      do while(flag.eq.0)
         i=i+1

         if(i.eq.maxtriloops)then
            numtriloops=i
            flag=1
         endif
```

```
         read(39,3031,end=98,err=91)seqn,energy

         if(index(seqn,'   ').gt.0)then
            numtriloops=i-1
            flag=1
         else
            do j=1,5
               call letr2num(seqn(j:j),numseqn(j))
            enddo

            triloop(i,1)=(((numseqn(5)*8+numseqn(4))*8+numseqn(3))*8+
     .       numseqn(2))*8+numseqn(1)
            triloop(i,2)=nint(prec*energy)
         endif
      enddo

      return

 98   continue
      numtriloops=i-1

      return

 91   write(6,*) 'STOP: ERROR reading triloop file'
      call exit(1)
      end
c-----------------------------------------------------------------
         subroutine letr2num(letr,num)

         integer num
         character*1 letr

         if(letr.eq.'A')then
            num=1
         elseif(letr.eq.'C')then
            num=2
         elseif(letr.eq.'G')then
            num=3
         elseif(letr.eq.'U'.or.letr.eq.'T')then
            num=4
         endif

      return
1     format(/)
2     format (a)
5     format (1x,'Too many characters in numeric field of this line
of',/,
     1        1x,'tloop.dat file: ',a)
      end
c-----------------------------------------------------------------------
      subroutine symint

      include 'rna_or_efn.inc'
      integer test1,test2,i,j,test3,test4,i1,j1,i2,j2,x1,y1,ip,jp,
     .        i3,j3,i4,test5,flag,worst
      real rsint2(6,6,5,5),rsint4(6,6,5,5,5,5),rsint6(6,6,25,5,5,5,5)
```

```
      character rowcha*144,row2*96

c  Explanation of sint6(a,b,c,e,f,g,h); variable changes as
corresponding
c  pair changes.
c  a  c   e   g  b
c     U   W   Y
c  A             C    Symmetric interior loop of size 6
c  B             D
c     V   X   Z
c         f   h

c  Reads in energies for symmetric interior loop of size 2

      i1=0
      flag=0

 3030 format(a144)
 3031 format(24f6.2)

      do while(flag.lt.4)

         read(33,3030,end=93,err=91)rowcha

         test1=index(rowcha(25:144),'<--')
         test2=index(rowcha(1:24),'                        ')

         if(test2.gt.0)then
            flag=flag+1
         elseif(test1.gt.0)then
            i1=i1+1

            do i2=1,4
               read(33,3031,err=91)((rsint2(i1,j1,i2,j2),j2=1,4),j1=1,6)
            enddo
            flag=0
         else
            flag=0
         endif
      enddo

 93   continue

      do i=1,6
         do j=1,6
            worst = 0
            do ip=1,4
               do jp=1,4
                  sint2(i,j,ip,jp) = nint(prec*rsint2(i,j,ip,jp))
                  worst = max0(worst,nint(prec*rsint2(i,j,ip,jp)))
               enddo
            enddo
            do ip = 1,5
               sint2(i,j,ip,5) = worst
               sint2(i,j,5,ip) = worst
            enddo
         enddo
```

```
      enddo

c   Reads in energies for symmetric interior loop of size 4

 3032 format(a96)
 3033 format(16f6.2)

      test3=0

      do while(test3.eq.0)
         read(34,3032) row2
         test3=index(row2,'<-----')
      enddo

      do x1=1,6
         do y1=1,6
            test4=0
            do while(test4.eq.0)
               read(34,3032,err=92) row2
               test4=index(row2,'<------')
            enddo
            do i2=1,4
               do j2=1,4
                  read(34,3032,err=92) row2
                  read(row2,3033,err=92)((rsint4(x1,y1,i2,j2,i3,j3),j3=
1,4),
     .            i3=1,4)
               enddo
            enddo
         enddo
      enddo

      do i=1,6
         do j=1,6
            worst = 0
            do ip=1,4
               do jp=1,4
                  do i1=1,4
                     do j1=1,4
                        sint4(i,j,ip,jp,i1,j1) = nint(prec*
     .                  rsint4(i,j,ip,jp,i1,j1))
                        worst =
max0(worst,nint(prec*rsint4(i,j,ip,jp,i1,j1)))
                     enddo
                  enddo
               enddo
            enddo
c            do ip = 1,5
c               do jp = 1,5
c                  do i1 = 1,5
c                     sint4(i,j,ip,jp,i1,5) = worst
c                     sint4(i,j,ip,jp,5,i1) = worst
c                     sint4(i,j,ip,5,jp,i1) = worst
c                     sint4(i,j,5,ip,jp,i1) = worst
c                  enddo
c               enddo
c            enddo
```

```
         enddo
      enddo

c   Reads in energies for symmetric interior loop size 6
c
c   KEY
c      c  e  g
c   a           b    Capital letters:  RNA/DNA bases
c      U--W--Y       Lower case letters:  variables assigned to
c   A_/       \_C                         specific bases
c   B \       / D
c      V--X--Z       sint6(a,b,c,e,f,g,h)
c
c         f  h

c3034 format(24F6.2)

c     test1=0

c     do while(test1.eq.0)
c        read(35,3030,err=94) rowcha
c        test1=index(rowcha(41:120),'-->')
c     enddo
c
c     do i=1,6
c        do i1=1,19
c            if ((i1/5)*5.ne.i1) then
c               do i3 = 1,4
c                 do j3 = 1,4
c                     test5 = 0
c                     do while(test5.eq.0)
c                       read(35,3030,err=94)rowcha
c                       test5=index(rowcha(41:120),'<--')
c                    enddo
c
c                    do i2=1,4
c                       read(35,3034,err=94)((rsint6(i,j,i1,i2,j2,
c    .                  i3,j3),j2=1,4),j=1,6)
c                    enddo
c                 enddo
c              enddo
c           endif
c        enddo
c     enddo
c      do i=1,6
c        do j=1,6
c           do i1=1,25
c              do i2=1,5
c                 do j2=1,5
c                    do i3=1,5
c                       do j3=1,5
c                          if((i1/5)*5.eq.i1) then
c                             sint6(i,j,i1,i2,j2,i3,j3) = infinity
c                          elseif(i2.eq.5.or.j2.eq.5) then
c                             sint6(i,j,i1,i2,j2,i3,j3) = infinity
c                          elseif(i3.eq.5.or.j3.eq.5) then
c                             sint6(i,j,i1,i2,j2,i3,j3) = infinity
```

```
c                          else
c                             sint6(i,j,i1,i2,j2,i3,j3) =
c    .                        nint(rsint6(i,j,i1,i2,j2,i3,j3)*prec)
c                          endif
c                       enddo
c                    enddo
c                 enddo
c              enddo
c           enddo
c        enddo
c     enddo
c
      return

 91   write(6,*) 'STOP: ERROR reading symmetric interior of size 2 file'
      call exit(1)

 92   write(6,*) 'STOP: ERROR reading symmetric interior of size 4 file'
      call exit(1)

c94   write(6,*) 'STOP: ERROR reading symmetric interior of size 6 file'
      call exit(1)
      end


c------------------------------------------------------------------
      subroutine asmint

      include 'rna_or_efn.inc'
      integer a,b,u,w,x,y,z,test
      character row*120
      real raint3(6,6,4,4,4),raint5(6,6,4,4,4,4,4)

 11   format(a144)
 12   format(24f6.2)

c   Reads in asymmetric interior loop of size 3 energies

      do a=1,6
         do b=1,6
            do x=1,5
               do y=1,5
                  do z=1,5
                     asint3(a,b,x,y,z)=infinity

                     do u=1,5
                        do w=1,5
                           asint5(a,b,x,w,y,u,z)=infinity
                        enddo
                     enddo

                  enddo
               enddo
            enddo
         enddo
      enddo
```

```
      do a=1,6
         do z=1,4
            test=0
            do while(test.eq.0)
               read(8,11,err=91)row
               test=index(row(41:120),'<--')
            enddo

            do x=1,4
               read(8,12,err=91)((raint3(a,b,x,y,z),y=1,4),b=1,6)

               do b=1,6
                  do y=1,4
                     asint3(a,b,x,y,z)=nint(prec*raint3(a,b,x,y,z))
                  enddo
               enddo

            enddo
         enddo
      enddo

c   Reads in asymmetric interior loop of size 5 energies
c-----------------------------------------------------------------
c   Diagram and explanation of asint5 dimensions:
c
c     a  x w    b
c        A-C--          asint5(a,b,x,w,y,u,z)
c  ===U_/     \_A===
c  ===G \     / U===
c        G-C-A
c        y u z           *  raint5 has same dimensions
c------------------------------------------------------------------
cTEMP do a=1,6
cTEMP    do x=1,4
cTEMP       do y=1,4
cTEMP          do u=1,4
cTEMP             test=0
cTEMP             do while(test.eq.0)
cTEMP                read(9,11,err=92)row
cTEMP                test=index(row(41:120),'<--')
cTEMP             enddo
cTEMP
cTEMP             do w=1,4
cTEMP
read(9,12,err=92)((raint5(a,b,x,w,y,u,z),z=1,4),b=1,6)
cTEMP                 do b=1,6
cTEMP                   do z=1,4
cTEMP                      asint5(a,b,x,w,y,u,z)=nint(prec*
cTEMP.                     raint5(a,b,x,w,y,u,z))
cTEMP                   enddo
cTEMP                enddo
cTEMP              enddo
cTEMP          enddo
cTEMP       enddo
cTEMP    enddo
cTEMP  enddo
```

```
      return

 91   write(6,*) 'STOP: ERROR reading asymmetric interior 1 x 2 file'
      call exit(1)

 92   write(6,*) 'STOP: ERROR reading asymmetric interior 2 x 3 file'
      call exit(1)
      end

c-------------------------------------------------------------------
      subroutine symtest

      include 'rna_or_efn.inc'
      integer diff,i,j,ii,jj,ip,jp,i1,i1p,j1p,i2,j2,i2p

c   Symmetry test for symmetric interior loops of size 2
      do i=1,6
         do j=1,6
            if(i.gt.4.and.i.le.6)then
               ip=11-i
            else
               ip=5-i
            endif
            if(j.gt.4.and.j.le.6)then
               jp=11-j
            else
               jp=5-j
            endif

            do ii=1,4
               do jj=1,4
                  diff=sint2(i,j,ii,jj)-sint2(jp,ip,jj,ii)
                  if(diff.ne.0)then
                     write(6,*) 'STOP: sint2 FAILED SYMMETRY TEST'
                     write(6,93) i,j,ii,jj,sint2(i,j,ii,jj)
 93                  format('sint2(',4i2,')= ',i6,' BUT ')
                     write(6,94) jp,ip,jj,ii,sint2(jp,ip,jj,ii)
 94                  format('sint2(',4i2,')= ',i6,'!')
                     call exit(1)
                  endif
               enddo
            enddo
         enddo
      enddo

c   Symmetry test for symmetric interior loops of size 4
      do i=1,6
         do j=1,6
            if(i.gt.4.and.i.le.6)then
               ip=11-i
            else
               ip=5-i
            endif
            if(j.gt.4.and.j.le.6)then
               jp=11-j
            else
               jp=5-j
```

```
            endif

            do ii=1,4
               do jj=1,4
                  do i2=1,4
                     do j2=1,4
                        diff=sint4(i,j,ii,jj,i2,j2)-
     .                  sint4(jp,ip,j2,i2,jj,ii)
                        if(diff.ne.0)then
                           write(6,*) 'STOP: sint4 FAILED SYMMETRY TEST'
                           write(6,*)
i,j,ii,jj,i2,j2,sint4(i,j,ii,jj,i2,j2)
                           write(6,*)
jp,ip,j2,i2,jj,ii,sint4(jp,ip,j2,i2,jj,ii)
                           call exit(1)
                        endif
                     enddo
                  enddo
               enddo
            enddo
         enddo
      enddo

c   Symmetry test for symmetric interior loops of size 6
c      do i=1,6
c         do j=1,6
c            if(i.gt.4.and.i.le.6)then
c               ip=11-i
c            else
c               ip=5-i
c            endif
c            if(j.gt.4.and.j.le.6)then
c               jp=11-j
c            else
c               jp=5-j
c            endif
c
c            do i1=1,19
c               if ((i1/5)*5.ne.i1)) then
c                  i1p = i1/5 + 1
c                  j1p = i1 - 5*(i1/5)
c                  do i2 = 1,4
c                     do j2 = 1,4
c                        i2p = (j2-1)*5 + i2
c                        do ii = 1,4
c                           do jj = 1,4
c                              diff = sint6(i,j,i1,ii,jj,i2,j2) -
c     .                               sint6(jp,ip,i2p,jj,ii,j1p,i1p)
c                              if (diff.ne.0) then
c                                 write(6,*) 'STOP: sint6 FAILED
SYMMETRY TEST'
c                                 call exit(1)
c                              endif
c                           enddo
c                        enddo
c                     enddo
c                  enddo
```

```
c                endif
c             enddo
c          enddo
c       enddo

      return
      end
```

Appendix E: Source Code : ct.h

```
/*********************************************************************
# Name  : ct.h
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : header file for ct.c
#
#
#
#
#
#*******************************************************************/

/* MAXLEN is also defined in maxn2.inc as maxn */
#ifndef MAXLEN
#define MAXLEN 3000

#ifdef __cplusplus
  extern "C" {
#endif
void basepairs_init();
void basepairs_store(int i, int j);
void basepairs_pop();
void basepairs_print(char seq[], char comment[]);
#ifdef __cplusplus
  }
#endif

#endif
```

Appendix F: Source Code : ct.c

```
/*********************************************************************
# Name  : ct.c
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : common module is shared by our program for Nussinov,
#         SCFG and ZUKER.
#         It provide functions to store and print out the RNA sequence
#         allows output in the CT format and DOT format.
#
#
#
#*******************************************************************/

#include <stdio.h>
#include "ct.h"

int basepairs[MAXLEN][2];

/******************************
*  base pairs initialization
******************************/

void basepairs_init() {
    int i;
    for (i=0; i<MAXLEN; i++) {
        basepairs[i][0]=-1;
        basepairs[i][1]=-1;
    }
}

/**********************************
*   Store  base pair(i,j)
**********************************/

void basepairs_store(int i, int j) {
    int k;
    for (k=0; k<MAXLEN; k++)
        if (basepairs[k][0] == -1 ) break;
    if (k==MAXLEN)
        exit(1); /*well, we can't have more than MAXLEN/2 base pairs */
    else {
        basepairs[k][0]=i;
        basepairs[k][1]=j;
    }
}
```

```
/**********************************
*   POP base pair from stack
**********************************/

void basepairs_pop() {
    int k;
    for (k=0; k<MAXLEN; k++)
        if (basepairs[k][0] == -1 ) break;
    if (k==MAXLEN)
        exit(1);  /* well, we can't have more than MAXLEN basepairs */
    else {
        k--;
        basepairs[k][0]=-1;
    }
}

/****************************************
*  Search for nucleotide pairing with i
****************************************/

int basepairs_find(int i) { /* search for the nucleotide pairing with i
*/
    int k;
    for (k=0; k<MAXLEN && basepairs[k][0] != -1 ; k++) {
        if (basepairs[k][0] == i) return basepairs[k][1];
        else if (basepairs[k][1] == i) return basepairs[k][0];
    }
    return -1; /* no pair found for this one */
}

/**************************************
*   Print base pairs
**************************************/
void basepairs_print(char seq[], char comment[]) {
    int i,j,k,seqlen;
    seqlen=strlen(seq);

/* Print full . format for RNAdistance */
    for (i=0; i<seqlen; i++) {
        j=i+1;
        k=basepairs_find(i)+1;
        if (k==0)
            printf(".");
        else if (k>j)
            printf("(");
        else if (k<j)
            printf(")");
    }
    printf("\n");

/* Now, generate .ct format output for Zuker's sir_graph */
    for (k=0; k<MAXLEN; k++)
       if (basepairs[k][0] == -1 ) break;
    printf("%d nucleotides, %d pairs %s\n",seqlen,k,comment);
    for (i=0; i<seqlen; i++) {
```

```
        j=i+1; /* In Zuker's MFOLD plot package, sequence # starts with
1 */
        printf("%5d %c %5d %5d %5d %5d\n",
               j,seq[i],j-1,(j+1)%(seqlen+1),basepairs_find(i)+1,j);
    }
}
```

Appendix G: Scource Code: fasta.h

```
/*******************************************************************
# Name  : fasta.h
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : header file for fasta.c
#
#
#
#
#
#******************************************************************/
#include "ct.h"

#ifdef __cplusplus
  extern "C" {
#endif

char pseq[MAXLEN];  /* RNA sequence to be folded */
int HAIRPINLEN;

char uppercase(char i);
int readFasta(int argc, char *argvs[]);

#ifdef __cplusplus
  }
#endif
```

Appendix H: Source Code : fasta.c

```
/*******************************************************************
# Name  : fasta.c
#
# Input : -h : hairpin length  (default 3 if no -h)
#         <filename>:  input file name (default stdin)
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : Read fasta format from stdin input (default) or inputfile
#         Receive Parm for HAIRPINLEN  or default to 3
#
#
#******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include "fasta.h"

/****************************
 *    Convert to uppercase
 ****************************/

char uppercase(char i) { return (i<97?i:i-32); }

/***************************
 *    print usage menu
 *************************/
void usage(char *myname) {
    fprintf(stderr,"Usage: %s -h minimum_hairpin_length
[FASTA_file]\n", myname);
    fprintf(stderr,"Note : option -h has no effect in SCFG\n");
    fprintf(stderr,"       if FASTA_file is not provided, read from
STDIN\n");
}

/*******************************************
 *    Read parm: myname for Fasta file name
 *    Read parm: HAIRPINLEN
 ****************************************/

int readFasta(int argc, char *argvs[]) {
    char rc, a, A, myname[256];
    FILE* seqfile;
    int i, seqlen;
```

```
    HAIRPINLEN=3;     /* default HAIRPINLEN = 3 */
    strcpy(myname,argvs[0]);

/****************************************************************
 * read command line options.
 * see "man 3 getopt" for usage of variable 'optarg' and 'optind'
 ****************************************************************/

    while ((rc=getopt(argc,argvs,"+h:"))>0) {
        switch (rc) {
        case 'h':   /* option -h hair_pin_length */
            i=0;    /* check to make sure optarg is a integer */
            while (optarg[i] != '\0') {
                if (! isdigit(optarg[i])) {
                    usage(myname);
                    exit(1);
                }
                i++;
            }
            sscanf(optarg,"%d",&HAIRPINLEN);
            break;
        default:
            usage(myname);
            exit(1);
        }
    }

/* ****************************************************************
 * if there is a string left (point to by argv[optind], treat it as the
 * optional FASTA file. Otherwise, read FASTA file from STDIN.
 ****************************************************************/
    if ( argvs[optind] != 0 )
        seqfile=fopen(argvs[optind],"r");
    else
        seqfile=stdin;

/* read sequence FASTA file */
    i=0;
    while ((a=fgetc(seqfile)) != EOF) {
        if (a=='>') /* comment line for FASTA format start with '>' */
            do { a=fgetc(seqfile); } while (a!='\n');
        else {
            if (a=='\n')
                break;
            else {
                A=uppercase(a); /* convert everything to uppercase */
                if (A=='T') A='U';
                if (A=='A' || A=='U' || A=='C' || A=='G') pseq[i++]=A;
            }
        }
    }
    fclose(seqfile);
    pseq[i]='\0';
    seqlen=i;

    return seqlen;
}
```

Appendix I: Source Code : n4ct.c

```
/*******************************************************************
# Name  : n4ct.c
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : This program implementx Nussinov Algrithm
#         for RNA secondary structure prediction.
#
#
#
#******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include "fasta.h"
#include "ct.h"

int stack[MAXLEN][2];
int score[MAXLEN][MAXLEN];
int stacktop=-1;
unsigned int totalcalls=0;

int maxscore(int i, int j);
void push(int i, int j);
void printpairs(int seqlen, char comment[]);

main (int argc, char* argvs[]) {
    int i, j, k, seqlen;

/* read RNA sequence and store to char string psep[] */
    seqlen=readFasta(argc,argvs);

    for (i=0; i<seqlen; i++) {
        // score[i][j] = -1 means it has not been calculated.
        for (j=0; j<seqlen; j++) score[i][j]=-1;
        for (k=0; k<HAIRPINLEN+1 && (i-k)>=0 ; k++) score[i-k][i]=0;
    }

    clock_t start, end;
    clock_t cpuclocks;
    start=clock();
/*****************************************************
 * Use one of the following three to calculate maxscore
 *     Way 1) 2 for loops: i: 0--seqlen-1; j: i+1--seqlen
 *     Way 2) parallel the calculation
```

```
 *     Way 3) grant recursive calculation:
 *            maxscore(0,seqlen-1)
 *******************************************************/
/********** WAY 1************/
/*
    for (i=0; i<seqlen-1; i++)
        for (j=i+1; j<seqlen; j++)
            score[i][j]=maxscore(i,j);
*/
/********** WAY 2 **********/

    for (k=0; k<seqlen; k++)
#pragma omp parallel for
       for (i=0; i<seqlen-k; i++)
           maxscore(i,i+k);


/*********** Way 3**********/
/*
    score[0][seqlen-1]=maxscore(0,seqlen-1);
*/

    end=clock();

/******************************************************
 * In CYGWIN, CLOCKS_PER_SEC is 1000,
 *            which means clock_t will not
 *            overflow within (2^31-1)/1000 seconds,
 *            or ~596.5 hours/24.8 days.
 * In Linux,  CLOCKS_PER_SEC = 1000000.
 *            So it overflows at ~ 36 minutes.
 ****************************************************/
#ifdef __CYGWIN__
  /*  printf(">>>> %d %d\n",end,start); */
    if (end >= start)
        cpuclocks=end-start;
    else
        cpuclocks=end-start+(clock_t)(pow(2,sizeof(clock_t)*8-1)-1);
#endif

    push(0,seqlen-1);
    fprintf(stderr,"Total score: %d\n", score[0][seqlen-1]);
    printf("Sequence length: %d\n", seqlen);
#ifdef __CYGWIN__
    printf("CPUtime: %g\n",((double) cpuclocks) / CLOCKS_PER_SEC);
#endif
/*  calling trace-back stage */
    printpairs(seqlen,"Nussinov");

/* Print Score Matrix */
/*
  for (i=0; i<seqlen; i++) {
    for (j=0; j<seqlen; j++) {
        printf("%d\t",maxscore(i,j));
      }
        printf("\n");
```

```
}
*/
    return(0);
} /* end MAIN */

/*************************************
 * Check if (i,j) is a canonical pair
 *************************************/
int canonical_pairs(char i, char j) {
    if (i=='C' && j=='G') return 1;
    else if (i=='G' && j=='C' ) return 1;
    else if (i=='A' && j=='U' ) return 1;
    else if (i=='U' && j=='A' ) return 1;
    else if (i=='G' && j=='U' ) return 1;
    else if (i=='U' && j=='G' ) return 1;
    else return 0;
}
/********************************
 * Push the (i,j) into the Stack
 ********************************/
void push(int i, int j) {
    stacktop++;
    stack[stacktop][0]=i;
    stack[stacktop][1]=j;
}

/**************************
 * Pop from the Stack
 **************************/
int pop(int *i, int *j) {
    if (stacktop>=0) {
        (*i)=stack[stacktop][0];
        (*j)=stack[stacktop][1];
        stacktop--;
        return(1);
    }
    else {
        return(0);
    }
}

/* ***************************************
 * recursively calculate the maximum score
 ****************************************/
int maxscore(int i, int j) {
    double thisscore, tmp;
    int k, seqlen=strlen(pseq);
    if (score[i][j]>=0) return score[i][j];
    totalcalls++;

    if ((j-i)<HAIRPINLEN+1) return 0;   /* neighbor can not pair */

    thisscore=maxscore(i+1,j-1)+canonical_pairs(pseq[i],pseq[j]);
    for (k=i; k<j; k++) {
        tmp=maxscore(i,k)+maxscore(k+1,j);
        if (tmp>thisscore) thisscore=tmp;
    }
```

```
    score[i][j]=thisscore;
    return thisscore;
}

/*******************************
 * Trace-back and print base pairs
 ******************************/

void printpairs(int seqlen, char comment[]) {
    int i,j,k;
    basepairs_init();
    while (pop(&i,&j)) {
        if  ((j-i)<HAIRPINLEN+1 ) continue;
        if (score[i][j] == (score[i+1][j-
1]+canonical_pairs(pseq[i],pseq[j]))) {
            if (canonical_pairs(pseq[i],pseq[j])) basepairs_store(i,j);
            push(i+1,j-1);
        }
        else
            for (k=i; k<j; k++)
                if ((score[i][k]+score[k+1][j]) == score[i][j]) {
                    push(i,k);
                    push(k+1,j);
                    break;
                }
    }
/*************************************************
 * Generate .ct format output for Zuker's sir_graph
 * and .dot format for RNAdistance in the Vienna package
 ************************************************/
    basepairs_print(pseq, comment);
}
```

Appendix J: Source Code: scfg.c

```
/*********************************************************************
# Name  : scfg.c
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : This program implements the Stochastic Context Free
#         Grammar (SCFG) using Nussinov rules for RNA secondary
#         structure prediction.
#
#
#
#
#********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include "fasta.h"
#include "ct.h"

#define PaS 0.045 /* *4 = 0.18 S -> {a,u,c,g}S  */
#define PSa 0.045 /* *4 = 0.18 S -> S{a,u,c,g}  */
#define PaSa  0.1 /* *6 = 0.6  S -> aSu | uSa | cSg | gSc | uSg | gSu */
#define PSS  0.04 /* *1 = 0.04 S -> SS */

/***********************
#define logPaS  -3.101093
#define logPSa  -3.101093
#define logPaSa -2.302585
#define logPSS  -3.218876
***********************/

int stack[MAXLEN][2], pathmatrix[MAXLEN][MAXLEN];
double probability[MAXLEN][MAXLEN];

int stacktop=-1;
unsigned int totalcalls=0;


double maxprob(int i, int j);
void push(int i, int j);
void printpairs(int seqlen, char comment[]);
```

```
/*********************
 * MAIN
 ********************/

main (int argc, char* argvs[]) {
    int i, j, k, seqlen;

/* read RNA sequence and store into char string psep[] */
    seqlen=readFasta(argc,argvs);
/*****************************
 *  pathmatrix[i][j] = -10 means
 *  it has not been calculated.
 *  The initialization set
 *  the pathmatrix to -10
 ****************************/
    for (i=0; i<seqlen; i++)
        for (j=0; j<seqlen; j++)
            pathmatrix[i][j]=-10;

    clock_t start, end;
    clock_t cpuclocks;
    start=clock();
 /**************************************
 * USE one of the two ways to calculate
 *    WAY 1)  parallel  the calculations
 *    WAY 2)  grand recursive calculation
 **************************************/

/********** WAY 1)*********************/
    for (k=0; k<seqlen; k++)
#pragma omp parallel for
       for (i=0; i<seqlen-k; i++)
           maxprob(i,i+k);

/********** WAY 2) ********************/
/*
    probability[0][seqlen-1]=maxprob(0,seqlen-1);
*/

    end=clock();
/*******************************************************************
 * In CYGWIN, CLOCKS_PER_SEC = 1000,
 *                  which means clock_t will not overflow
 *                  within (2^31-1)/1000 seconds,
 *                  or ~596.5 hours/24.8 days.
 * In Linux, CLOCKS_PER_SEC = 1000000.
 *                  So it overflows at ~ 36 minutes.
 ******************************************************************/
#ifdef __CYGWIN__
    if (end >= start)
        cpuclocks=end-start;
    else
        cpuclocks=end-start+(clock_t)(pow(2,sizeof(clock_t)*8-1)-1);
#endif

    push(0,seqlen-1);
```

```
    fprintf(stderr,"Total log probability: %f\n",
probability[0][seqlen-1]);
    printf("Sequence length: %d\n", seqlen);
#ifdef __CYGWIN__
    printf("CPUtime: %g\n",((double) cpuclocks) / CLOCKS_PER_SEC);
#endif
    printpairs(seqlen,"SCFG");

/* Print Probability matrix */
/*
    for (i=0; i<seqlen; i++) {
        for (j=0; j<seqlen; j++)
            printf("%7.2f,",probability[i][j]);
        printf("\n");
    }

*/

    return(0);
}


/****************************************
 *  Check if i,j is a canonical pair
 ****************************************/

int canonical_pairs(char i, char j) {
    if (i=='C' && j=='G') return 1;
    else if (i=='G' && j=='C' ) return 1;
    else if (i=='A' && j=='U' ) return 1;
    else if (i=='U' && j=='A' ) return 1;
    else if (i=='G' && j=='U' ) return 1;
    else if (i=='U' && j=='G' ) return 1;
    else return 0;
}

/**************************************
 *  push i,j onto the stack
 **************************************/

void push(int i, int j) {
    stacktop++;
    stack[stacktop][0]=i;
    stack[stacktop][1]=j;
}

/************************************
 * pop from the stack
 ************************************/

int pop(int *i, int *j) {
    if (stacktop>=0) {
        (*i)=stack[stacktop][0];
        (*j)=stack[stacktop][1];
        stacktop--;
        return(1);
    }
```

```
    else {
        return(0);
    }
}

/**************************************************************
 * recursively calculate maximum probability
 *
 * Additional notes about probability[][] and pathmatrix[][]
 *  probabilities are saved in matrix probabilty[N][N].
 *  It has type of 'double'.
 *  Function maxprob(i,j) calculates each probability[i][j].
 *  A tracking matrix pathmatrix[N][N] (type 'int') is
 *  set to -10 initially.
 *  Function maxprob(i,j) will change pathmatrix[i][j] to:
 *  pathmatrix[i][j] = -1  when i, j pairs
 *  pathmatrix[i][j] = k   for bi-furcation at (i,k) + (k+1,j)
 *  pathmatrix[i][j] > -10 means probablity[i][j] has been
 *                         calculated, and should not be
 *                         calculated again.
 *************************************************************/

double maxprob(int i, int j) {
    double thisprobability, tmp;
    int k, seqlen=strlen(pseq);
    if (pathmatrix[i][j]>-10) return probability[i][j];
    totalcalls++;

    if (i==j) {
        pathmatrix[i][j]=-5;
        probability[i][j]=-0;
        return probability[i][j];
    }

    thisprobability=maxprob(i+1,j)+log(PaS);
    pathmatrix[i][j]=i;

    tmp=maxprob(i,j-1)+log(PSa);
    if (tmp>thisprobability) {
        thisprobability=tmp;
        pathmatrix[i][j]=j-1;
    }

    for (k=i+1; k<j-1; k++) {
        tmp=maxprob(i,k)+maxprob(k+1,j)+log(PSS);
        if (tmp>thisprobability) {
            thisprobability=tmp;
            pathmatrix[i][j]=k;
        }
    }

    if (i+1<j-1) {
        if (canonical_pairs(pseq[i],pseq[j])==1)
            tmp=maxprob(i+1,j-1)+log(PaSa);
        if (tmp>thisprobability) {
            thisprobability=tmp;
            pathmatrix[i][j]=-1;
```

```
        }
    }

    probability[i][j]=thisprobability;
    return thisprobability;
}

/************************************
 * Trace back stage
 ************************************/

void printpairs(int seqlen, char comment[]) {
    int i,j,k;
    basepairs_init();
    while (pop(&i,&j)) {
        if (i>=j) continue;
        k=pathmatrix[i][j];
        if (k==-1) {
            basepairs_store(i,j);
            push(i+1,j-1);
        }
        else if (k>=0) {
            push(i,k);
            push(k+1,j);
        }
    }

/*********************************************************
 * Now, generate .ct format output for Zuker's sir_graph
 * and .dot format for RNAdistance of the Vienna package
 ********************************************************/
    basepairs_print(pseq, comment);
}
```

Appendix K: Source Code: n4m.cc

```
/*********************************************************************
# Name  : n4m.cc
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : This program implements a variance of the Nussinov
#         algorithm that traces all possible RNA secondary structures
#         with maximum number of base pair.
#
#
#
#
#
#********************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "fasta.h"
#include "ct.h"
#include <list>
#include <set>

using namespace std;

/*******************************************
 * Check if i,j is a canonical pair
 ******************************************/
int canonical_pairs(char i, char j) {
    if (i=='C' && j=='G') return 1;
    else if (i=='G' && j=='C' ) return 1;
    else if (i=='A' && j=='U' ) return 1;
    else if (i=='U' && j=='A' ) return 1;
    else if (i=='G' && j=='U' ) return 1;
    else if (i=='U' && j=='G' ) return 1;
    else return 0;
}

struct Pair {  // struct for a base pair
    short i, j;
    bool initialized;
};

/*******************************************
 *  Initialization of pair
 ******************************************/
```

```
void PairInit(struct Pair *pair, short i, short j) {
    pair->i=i; pair->j=j;
    pair->initialized=true;
}

/*****************************************************
 * User defined class for comparing pair x and pair y
 * set<struct Pair, comp> take this as the '<' operator
 *     for the Pairs
 *****************************************************/

class comp {
public:
    bool operator() (const struct Pair& x, const struct Pair& y) {
        return ((x.i == y.i)? x.j < y.j : x.i < y.i);
    }
};

/**********************************************
 * Check if pair sets sl1 and sl2 are identical
 **********************************************/

bool PairSetEqual(set<struct Pair, comp> &sl1, set<struct Pair, comp>
&sl2) {
    set<struct Pair, comp>::iterator l1, l2;

    if (sl1.size() != sl2.size()) return false;
    l1=sl1.begin();
    l2=sl2.begin();
    for (int i=0; i<sl1.size(); i++) {
        if ( (*l1).i != (*l2).i || (*l1).j != (*l2).j) return false;
        l1++;
        l2++;
    }
    return true;
}

// struct GAMMA for the RNA subsequence a...z

struct Gamma {
    char *seq;
    short a,z;
    unsigned int pairListLen;
    list<struct Pair> pairLists;
    Gamma() { seq = NULL; }
};

struct Gamma GammaMatrix[MAXLEN][MAXLEN];

/*******************************************
 *  Copy g2 (gamma type of strct) to g1
 *******************************************/
void GammaCopy(struct Gamma *g1, struct Gamma *g2) { // g1 = g2
    list<struct Pair>::iterator l;
    g1->seq=g2->seq;
    g1->a=g2->a;
```

```
    g1->z=g2->z;
    g1->pairListLen=g2->pairListLen;
    g1->pairLists.erase(g1->pairLists.begin(),g1->pairLists.end());
    for (l=g2->pairLists.begin(); l != g2->pairLists.end(); l++)
        g1->pairLists.push_back(*l);
}

/***********************************************
 *   Initialization of g (gamma type of stuct)
 ***********************************************/

void GammaInit(struct Gamma *g, char *seq, short i, short j) {
    g->seq=seq;
    g->a=i;
    g->z=j;
    g->pairLists.erase(g->pairLists.begin(),g->pairLists.end());
    g->pairListLen=0;
}


int score[MAXLEN][MAXLEN];

/***********************************************
 *  Recursive calculation of subsequence a...z
 *  result is stored in matrix element GammaMatrix[a][z]
 *
 ***********************************************/
void GammaDo(char *seq, short a, short z) {
    struct Pair pair;
    struct Gamma *g, *g1, *g2;
    int i, j, k, numList, numList1, numList2, i1, j1, i2, j2;
    bool duplicate;
    list<struct Pair> p1, p2;
    list<struct Pair>::iterator l,l1,l2;
    set<struct Pair, comp> s, stmp;
    set<struct Pair, comp>::iterator sl;

    g=&GammaMatrix[a][z];
    if (g->seq != NULL ) return;

    GammaInit(g,seq,a,z);
    if ( (g->z - g->a) < HAIRPINLEN+1) {
        return;
    }
    else {
        pair.initialized=false;
        if (canonical_pairs(g->seq[g->a],g->seq[g->z])) {
            PairInit(&pair,g->a,g->z);
        }
        GammaDo(seq,a+1,z-1);
        g1=&GammaMatrix[a+1][z-1];

        l=g1->pairLists.begin();
        if (g1->pairListLen != 0) {
            numList = g1->pairLists.size()/g1->pairListLen;
            for (i=0; i < numList; i++) {
                g->pairListLen=0;
```

```
                if (pair.initialized) {
                    g->pairLists.push_back(pair);
                    g->pairListLen++;
                }
                for (j=0; j<g1->pairListLen; j++) {
                    g->pairLists.push_back(*l);
                    l++;
                    g->pairListLen++;
                }
            }
        }
        else if (pair.initialized) {
            g->pairLists.push_back(pair);
            g->pairListLen++;
        }
// bifurcatoin
        for (k=g->a; k<g->z; k++) {
            GammaDo(seq,g->a,k);
            GammaDo(seq,k+1,g->z);

            g1=&GammaMatrix[g->a][k];
            g2=&GammaMatrix[k+1][g->z];
            if ((g1->pairListLen+g2->pairListLen) > score[g->a][g->z]) {
                printf("Something is wrong ! %d %d %d %d %d %d\n",
a,k,z, score[g->a][g->z],g1->pairListLen,g2->pairListLen);
                exit(1);
            }
            else if ((g1->pairListLen+g2->pairListLen) < score[g->a][g-
>z]) continue;
            if ((g1->pairListLen+g2->pairListLen) > g->pairListLen) {
                g->pairLists.erase(g->pairLists.begin(),g-
>pairLists.end());
                g->pairListLen=g1->pairListLen+g2->pairListLen;
            }
            if ((g1->pairListLen+g2->pairListLen) >= g->pairListLen) {
                numList1 =
                    ((g1->pairListLen==0)? 0:g1->pairLists.size()/g1-
>pairListLen);
                numList2 =
                    ((g2->pairListLen==0)? 0:g2->pairLists.size()/g2-
>pairListLen);
                l1=g1->pairLists.begin();
                i1=0;
                do {
                    i1++;
                    p1.erase(p1.begin(),p1.end());
                    for (j1=0; j1<g1->pairListLen; j1++) {
                        p1.push_back(*l1);
                        l1++;
                    }
                    l2=g2->pairLists.begin();
                    i2=0;
                    do {
                        i2++;
                        for (l=p1.begin(); l != p1.end(); l++)
                            g->pairLists.push_back(*l);
                        for (j2=0; j2<g2->pairListLen; j2++) {
```

```
                            g->pairLists.push_back(*l2);
                            l2++;
                        }
                    } while (i2 < numList2);
                } while (i1 < numList1);
            } // end if
        } // end bifurction loop

// check for duplicated pairList in g->pairLists
        p1.erase(p1.begin(),p1.end());
        for (l=g->pairLists.begin(); l != g->pairLists.end(); ) {
            s.erase(s.begin(),s.end());
            for (i=0; i<g->pairListLen; i++) {
                s.insert(*l);
                l++;
            }
            duplicate=false;
            for (l1=p1.begin(); l1 != p1.end(); ) {
                stmp.erase(stmp.begin(),stmp.end());
                for (j=0; j<g->pairListLen; j++) {
                    stmp.insert(*l1);
                    l1++;
                }
                if (PairSetEqual(stmp,s)) {
                    duplicate=true;
                    break;
                }
            }
            if (! duplicate)
                for (sl=s.begin(); sl != s.end(); sl++)
                    p1.push_back(*sl);
        }
        g->pairLists.erase(g->pairLists.begin(),g->pairLists.end());
        for (l1=p1.begin(); l1 != p1.end(); l1++)
             g->pairLists.push_back(*l1);
/*
        if (g->pairListLen != 0)
            fprintf(stderr,
                    "[ %3d, %3d ] Calculated, Total %d lists, %d pairs
each\n",
                    g->a,g->z,g->pairLists.size()/g->pairListLen,g-
>pairListLen);
 */
    }
} // end of function GammaDo()

/*************************************
 *  Maxscore Calculation for score(i,j)
 *************************************/

int maxscore(char *pseq, short i, short j) {
    int thisscore, tmp;
    int k, seqlen=strlen(pseq);
    if (score[i][j]>=0) return score[i][j];

    if ((j-i)<4) return 0;   /* neighbor can not pair */
```

```
    thisscore=maxscore(pseq,i+1,j-1)+canonical_pairs(pseq[i],pseq[j]);
    for (k=i; k<j; k++) {
        tmp=maxscore(pseq,i,k)+maxscore(pseq,k+1,j);
        if (tmp>thisscore) thisscore=tmp;
    }
    score[i][j]=thisscore;
    return thisscore;
}

/*****************************************
 *   MAIN
 *****************************************/
int main(int argc, char* argvs[]) {
    int i, j, k, seqlen, numList;
    struct Gamma *g;
    list<struct Pair>::iterator l;

/* read RNA sequence and store to char string psep[] */
    seqlen=readFasta(argc, argvs);

    for (i=0; i<seqlen; i++)
        for (j=0; j<seqlen; j++) score[i][j]=-1;
/*********************************************************
 *  USE one of the twos ways for the energy calculation
 *      WAY 1) grant recursion calls
 *      WAY 2) parallelization
 *********************************************************/
/************** WAY 1) ***********************/
/*
   maxscore(pseq,0,seqlen-1);
    GammaDo(pseq,0,seqlen-1);
*/

/************* WAY 2) ***********************/
    for (k=HAIRPINLEN; k<seqlen; k++)
#pragma omp parallel for
        for (i=0; i<seqlen-k; i++) {
            maxscore(pseq, i, i+k);
            GammaDo(pseq,i,i+k);
        }

    g=&GammaMatrix[0][seqlen-1];
    l=g->pairLists.begin();
    numList = ((g->pairListLen==0)? 0 : g->pairLists.size()/g-
>pairListLen);
    for (i=0; i<numList; i++) {
        basepairs_init();
        for (j=0; j<g->pairListLen; j++) {
            basepairs_store(l->i,l->j);
            l++;
        }
        basepairs_print(pseq," ");
    }
    return(0);
}
```

Appendix L: Source Code: readMfoldDat.h

```
/*******************************************************************
# Name  : readMfoldDat.h
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : header file for readMfoldDat.c
#
#
#
#
#
#******************************************************************/

#define maxtloops  100
#define maxtriloops 50
#define infinity 999999

struct {
  int asint3[5][5][5][6][6],asint5[5][5][5][5][5][6][6],bulge[30],
      dangle[2][5][5][5],eparam[16],hairpin[30],inter[30],poppen[4],
      sint2[5][5][6][6],sint4[5][5][5][5][6][6],sint6[5][5][5][5][25][6
][6],
      stack[5][5][5][5],tloop[2][maxtloops],triloop[2][maxtriloops],
      tstkh[5][5][5][5],tstki[5][5][5][5],prelog;
} efiles_;

float asint3[6][6][5][5][5],asint5[6][6][5][5][5][5][5],bulge[30],
      dangle[5][5][5][2],eparam[16],hairpin[30],inter[30],poppen[4],
      sint2[6][6][5][5],sint4[6][6][5][5][5][5],sint6[6][6][25][5][5][5
][5],
      stack[5][5][5][5],tloop[maxtloops][2],triloop[maxtriloops][2],
      tstkh[5][5][5][5],tstki[5][5][5][5],prelog;
static int maxpen=3.0; /* second number in miscloop.dat */
static int numtloops=30; /* number of entries in tloop.dat */

void enefiles_();
void ergread_();
void readMfoldDat();
int n2n(char); /* convert nucleotide to number, see readMfoldDat.c */
int p2n(char, char); /* map a pair of nucleotides to number */
/*******************************************************************
****
* NOTE: The following is excerpted from mfold v3.2 Fortran include
        file src/rna_or_efn.inc

      parameter (maxtloops=100)
```

```
      parameter (maxtriloops=50)

      integer asint3(6,6,5,5,5),asint5(6,6,5,5,5,5,5),bulge(30),
     .
dangle(5,5,5,2),eparam(16),hairpin(30),inter(30),poppen(4),
     .        sint2(6,6,5,5),sint4(6,6,5,5,5,5),sint6(6,6,25,5,5,5,5),
     .        stack(5,5,5,5),tloop(maxtloops,2),triloop(maxtriloops,2),
     .        tstkh(5,5,5,5),tstki(5,5,5,5)

      common /efiles/
asint3,asint5,bulge,dangle,eparam,hairpin,inter,poppen,
     .        sint2,sint4,sint6,stack,tloop,triloop,tstkh,tstki,prelog

 *******************************************************************
*****/
```

Appendix M: Source Code: readMfoldDat.c

```
/**********************************************************************
# Name  : readMfoldDat.c
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : This program reads the Zuker's MFOLD's (v3.2) energy data
#         files. This is an interface of C to Fortran.
#
#
#
#
#********************************************************************/

#include <stdio.h>
#include "readMfoldDat.h"
/*********************************************************************
* NOTE: Excerpt from mfold v3.2 Fortran src/misc-quik.f

c           Non-excised bases are examined to determine their type.
c           A - type 1
c           C - type 2
c           G - type 3
c           U/T - type 4
c           anything else - type 5

********************************************************************/

/*********************************************
 * Map a nucleotide to numeric
 ********************************************/
int n2n(char nucltd) { /* nucleotide type */
    if (nucltd=='A' || nucltd=='a') return 0;
    if (nucltd=='C' || nucltd=='c') return 1;
    if (nucltd=='G' || nucltd=='g') return 2;
    if (nucltd=='U' || nucltd=='u') return 3;
    return -1;
}

char uppercase(char i); /* { return (i<97?i:i-32); } */

/*******************************************
 * Map a pair of nucleotides to number
 * NOTE:  used by sint2, asint3, sint4
 ******************************************/
int p2n(char n1, char n2) {
```

```
    n1=uppercase(n1);
    n2=uppercase(n2);
    if (n1=='A' && n2=='U')
        return 0;
    else if (n1=='C' && n2=='G')
        return 1;
    else if (n1=='G' && n2=='C')
        return 2;
    else if (n1=='U' && n2=='A')
        return 3;
    else if (n1=='G' && n2=='U')
        return 4;
    else if (n1=='U' && n2=='G')
        return 5;
    else
        return -1;
}
/*****************************************
 * Read  Mfold's Data files
 *****************************************/
void readMfoldDat() {
    int i,j,k,l,m,n,o;
    enefiles_();
    ergread_();

/*******************************************************************
 * Transfer data from FORTRAN's common block 'efiles' to C variables
 *
 * Note: A Fortran array F[k=1..K][h=1..H] cooresponds to a C array
 *       C[h=0..H-1][k=0..K-1]. To make the usage consistent with that
 *       in Fortran, we create another C array C'[k][h] = C[h][k]
 *
 * Note: When efiles read in data, it magnifies them by 100 and
 *       converts them to integers. We scale the magnitude down by 100.
 ******************************************************************/
    for (i=0; i<5; i++)
        for (j=0; j<5; j++) {
            for (k=0; k<5; k++) {
                for (l=0; l<2; l++)
                    dangle[l][k][j][i]=(float)efiles_.dangle[i][j][k][l
]/100;
                for (l=0; l<5; l++) {
                    stack[l][k][j][i]=(float)efiles_.stack[i][j][k][l]/
100;
                    tstkh[l][k][j][i]=(float)efiles_.tstkh[i][j][k][l]/
100;
                    tstki[l][k][j][i]=(float)efiles_.tstki[i][j][k][l]/
100;
                    for (m=0; m<5; m++)
                        for (n=0; n<6; n++)
                            for (o=0; o<6; o++)
                                asint5[o][n][m][l][k][j][i]=(float)efil
es_.asint5[i][j][k][l][m][n][o]/100;
                    for (m=0; m<6; m++)
                        for (n=0; n<6; n++)
                            sint4[n][m][l][k][j][i]=(float)efiles_.sint
4[i][j][k][l][m][n]/100;
```

```
                    for (m=0; m<25; m++)
                        for (n=0; n<6; n++)
                            for (o=0; o<6; o++)
                                sint6[o][n][m][l][k][j][i]
=(float)efiles_.sint6[i][j][k][l][m][n][o]/100;
                }
                for (l=0; l<6; l++)
                    for (m=0; m<6; m++)
                        asint3[m][l][k][j][i] =
(float)efiles_.asint3[i][j][k][l][m] / 100;
                }
            for (k=0; k<6; k++)
                for (l=0; l<6; l++)
                    sint2[l][k][j][i] =
(float)efiles_.sint2[i][j][k][l] / 100;
        }
    for (i=0; i<2; i++)
        for (j=0; j<maxtloops; j++)
            tloop[j][i]=(float)efiles_.tloop[i][j]/100;
    for (i=0; i<2; i++)
        for (j=0; j<maxtriloops; j++)
            triloop[j][i]=(float)efiles_.triloop[i][j]/100;
    for (i=0; i<30; i++) {
        bulge[i]=(float)efiles_.bulge[i]/100;
        hairpin[i]=(float)efiles_.hairpin[i]/100.0;
        inter[i]=(float)efiles_.inter[i]/100;
    }
    for (i=0; i<16; i++) eparam[i]=(float)efiles_.eparam[i]/100;
    for (i=0; i<4;  i++) poppen[i]=(float)efiles_.poppen[i]/100;

    prelog=(float)efiles_.prelog/100;
}
```

Appendix N: Source Code: freeEnergy.h

```
/*********************************************************************
# Name  : freeEnergy.h
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : header file for freeEnergy.c
#
#
#
#
#
#********************************************************************/

#include <stdio.h>
#include "fasta.h"

/************************************************
 * pathmatrix is a 2-D matrix to help tracing Wij
 * looppatrix is a 3-D matrix to help tracing Vij
 * See additional notes below:
 ***********************************************/
short pathmatrix[MAXLEN][MAXLEN];
short loopmatrix[MAXLEN][MAXLEN][2];

float Vij[MAXLEN][MAXLEN], Wij[MAXLEN][MAXLEN];

float v(int, int);
float w(int, int);

/*********************************************************************
Energy function Wij[N][N] is a 2-D matrix of type 'double' or 'float',
so does Vij[N][N].
Function w(i,j) calculate each Wij[i][j].
A tracking matrix pathmatrix[N][N] (type 'int') is set to -10
initially.

Function w(i,j) also change pathmatrix[i][j] to:
pathmatrix[i][j] = -5 when the length of i, j is shorter than allowed
                      hairpin loop length (sharp U-turn). Wij[i][j] =
0,
                      Vij[i][j] = infinity.
pathmatrix[i][j] = -1 when i, j pairs,
pathmatrix[i][j] = k  for bi-furcation at (i,k) + (k+1, j)
pathmatrix[i][j] > -10 means Wij[i][j] (and Vij[i][j]) has been
calculated, and should not be calculated again.
```

```
Function v(i,j) calculates Vij[i][j]. v(i,j) looks at pathmatrix[[i][j]
to see if Vij[i][j] is calculated. For this reason, function w(i,j)
will always call v(i,j). But v(i, j) never changes pathmatrix[i][j].
We also use another 3-D tracking matrix loopmatrix[N][N][2] (of type
'int') to help tracing Vij. Initial value of loopmatrix[N][N][2] is not
important.

loopmatrix[i][j][0] = i+1, loopmatrix[i][j][1] = j-1 if this is a
                           hairpin or stack
loopmatrix[i][j][0] = ip, loopmatrix[i][j][1] = jp (ip != jp)  if this
                          is a bulge or interior loop of (i,ip,jp,j)
loopmatrix[i][j][0] = loopmatrix[i][j][1] = k if this is a bi-fucation
                                              at (i,k) + (k+1,j).
**********************************************************************/
```

Appendix O: Source Code: freeEnergy.c

```
/**********************************************************************
# Name  : freeEnergy.c
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : The ZUKER's energy functions
#
#
#
#
#
#*********************************************************************/

#include <math.h>
#include "ct.h"
#include "readMfoldDat.h"
#include "freeEnergy.h"

/*****************************************
 *    min of x and y
 *****************************************/

int min(int x, int y) {
    return (x<y? x:y);
}

/*****************************************
 *   Check if (i,j) is canonical pair,
 *   return 1, if it is
 *   return 0, if it is NOT
 ****************************************/

int canonical_pairs(char i, char j) {
    if (i=='C' && j=='G') return 1;
    else if (i=='G' && j=='C' ) return 1;
    else if (i=='A' && j=='U' ) return 1;
    else if (i=='U' && j=='A' ) return 1;
    else if (i=='G' && j=='U' ) return 1;
    else if (i=='U' && j=='G' ) return 1;
    else return 0;
}

/* AU or GU penalty for Stack */

float AUpenalty(char n1, char n2) {
    if (canonical_pairs(n1,n2) &&
```

```
        (n1=='U' || n1=='u' || n2=='U' || n2=='u'))
        return eparam[9];
    else
        return 0;
}


/***************************************
 * Energy function for Hairpin loop
   5': i  i+1 ...
   3': j  j-1 ...
 ***************************************/

float Hairpin(int i, int j) {
    int nUnpaired, key, k;
    float energy;
    nUnpaired=j-i-1;
    if (nUnpaired<HAIRPINLEN) {
        energy=infinity;
        return energy;
    }
    else if (nUnpaired<=30)
        energy=hairpin[nUnpaired];
    else
        energy=hairpin[29]+prelog*log(nUnpaired/30);

/* terminal mismatch */
    energy+=tstkh[n2n(pseq[i])][n2n(pseq[j])][n2n(pseq[i+1])][n2n(pseq[
j-1])];

/* triloop and tetraloop. Note triloop file is empty */
    if (nUnpaired==3)
        energy+=AUpenalty(pseq[i],pseq[j]);
    if (nUnpaired==4) {
      key=(n2n(pseq[i+5])+1)*32768+
          (n2n(pseq[i+4])+1)*4096+
          (n2n(pseq[i+3])+1)*512+
            (n2n(pseq[i+2])+1)*64+
          (n2n(pseq[i+1])+1)*8+
          (n2n(pseq[i+0])+1);
      for (k=0; k<numtloops; k++) {
          if (tloop[k][0]==key) {
              energy+=tloop[key][1];
            break;
          }
      }
    }
/* poly-C loop */
    k=i+1;
    while (uppercase(pseq[k])=='C' && k<j) k++;
    if (k==j)
      if (nUnpaired==3) energy+=eparam[13];
        else energy+=eparam[12]+nUnpaired*eparam[11];
/* CCC hairpin loop */
    if (i>=2 &&
      uppercase(pseq[i])=='G' && uppercase(pseq[i-1])=='G' &&
        uppercase(pseq[i-2])=='G' && uppercase(pseq[j])=='U')
```

```
      energy+=eparam[10];

/* contribution from miscloop */
    energy+=eparam[3];
    return energy;
}

/*******************************************************
 * Energy function for Bulge Loop
   5': i ip ...                or     5': i ... ip ...
   3': j ... jp ...                   3': j jp ...
 *******************************************************/

float Bulge(int i, int j, int ip, int jp) {
    int nUnpaired;
    float energy;
    nUnpaired=((ip-i-1 > j-jp-1)? ip-i-1 : j-jp-1);
    if (nUnpaired<1)
        energy=infinity;
    else if (nUnpaired==1)
        energy=bulge[nUnpaired]+eparam[1]+
               stack[n2n(pseq[i])][n2n(pseq[j])][n2n(pseq[ip])][n2n(pse
q[jp])];
    else if (nUnpaired<=30)
        energy=bulge[nUnpaired]+eparam[1]+
               AUpenalty(pseq[i],pseq[j])+AUpenalty(pseq[ip],pseq[jp]);
    else
        energy=bulge[29]+prelog*log(nUnpaired/30)+eparam[1]+
               AUpenalty(pseq[i],pseq[j])+AUpenalty(pseq[ip],pseq[jp]);
    return energy;
}

/*******************************************************
 *  Energy function for Interior Loop

   5': i  ... ip ...
   3': j  ... jp ...
 *******************************************************/

float Inter(int i, int j, int ip, int jp) {
    int nUnpaired, nAsym;
    float energy;
    energy=0;
    nUnpaired=ip-i-1+j-jp-1;
    nAsym=ip-i-1-(j-jp-1);
    energy=eparam[2];
    if (nUnpaired > 4 || abs(nAsym) > 1) {
        if (nUnpaired<=30)
            energy+=inter[nUnpaired];
        else
            energy+=inter[29]+prelog*log(nUnpaired/30);
      energy+=tstki[n2n(pseq[i])][n2n(pseq[j])]
                 [n2n(pseq[i+1])][n2n(pseq[j-1])]+
              tstki[n2n(pseq[ip])][n2n(pseq[jp])]
                 [n2n(pseq[ip-1])][n2n(pseq[jp+1])]+
              min(maxpen,(nAsym*poppen[min(4,min(ip-i-1,j-jp-1))]));
    }
```

```
    else {
/*****************************************************
   5': i   i+1   ip

   3': j   j-1   jp
 *****************************************************/
        if (nUnpaired==2 && nAsym==0)
          energy+=sint2[p2n(pseq[i],pseq[j])][p2n(pseq[ip],pseq[jp])]
                     [n2n(pseq[i+1])][n2n(pseq[j-1])];
/*****************************************************
   5': i   i+1   ip

   3:' j j-1 j-2 jp
 *****************************************************/
        else if (nUnpaired==3 && nAsym==-1 )
            energy+=asint3[p2n(pseq[i],pseq[j])][p2n(pseq[ip],pseq[jp])]
                      [n2n(pseq[i+1])][n2n(pseq[j-1])][n2n(pseq[j-2])];
/************************************************************
 * 5': i i+1 i+2 ip                      jp    jp+1   j    :3'
 *                   rotate 180 degree
 * 3': j   j-1   jp                      ip ip-1 ip-2 i    :5'
 ************************************************************/
        else if (nUnpaired==3 && nAsym==1)
            energy+=asint3[p2n(pseq[jp],pseq[ip])][p2n(pseq[j],pseq[i])]
                      [n2n(pseq[jp+1])][n2n(pseq[ip-1])][n2n(pseq[ip-
2])];
/******************************************************
 * 5': i i+1 i+2 ip
 *
 * 3': j j-1 j-2 jp
 *****************************************************/
        else if (nUnpaired==4 && nAsym==0)
            energy+=sint4[p2n(pseq[i],pseq[j])][p2n(pseq[ip],pseq[jp])]
                     [n2n(pseq[i+1])][n2n(pseq[j-1])]
                     [n2n(pseq[i+2])][n2n(pseq[j-2])];
      else
          energy=infinity;
    }
    return ( (energy>infinity)? infinity:energy );
}

/*******************************************************
   Energy function for  Stack Region
   5':  i i+1 ...
   3':  j j-1 ...
 ******************************************************/
float Stack(int i, int j) {
    float energy;
    if ((j-i-1) < 3) return infinity; /* limited by the size of hairpin
*/
    if (! canonical_pairs(pseq[i],pseq[j])) return infinity;
    energy=eparam[0]+
           stack[n2n(pseq[i])][n2n(pseq[j])][n2n(pseq[i+1])][n2n(pseq[j
-1])];
    return energy;
}
```

```
/*****************************************************
 *  Vij Function
 *****************************************************/
float v(int i, int j) {
    int ip, jp, k;
    float energy, tmp;

    if (pathmatrix[i][j] > -10)  /* already calculated */
         return Vij[i][j];

/* if i j not pair or shorter than hairpin minimum length ... */
    if (! canonical_pairs(pseq[i],pseq[j]) || (j-i)<HAIRPINLEN+1) {
      Vij[i][j]=infinity;
        return infinity;
    }

    energy=Hairpin(i,j);  /* Hairpin */
    loopmatrix[i][j][0]=i+1;
    loopmatrix[i][j][1]=j-1;

    for (ip=i+1; ip<j; ip++)
        for (jp=j-1; jp>ip; jp--) {
            if (ip==(i+1) && jp==(j-1)) { /* Stack */
                tmp=Stack(i,j)+v(i+1,j-1);
                if (tmp<energy) {
                    energy=tmp;
                    loopmatrix[i][j][0]=i+1;
                    loopmatrix[i][j][1]=j-1;
                }
            }
            else if (ip==(i+1) || jp==(j-1)) { /* Bulge */
                tmp=Bulge(i,j,ip,jp)+v(ip,jp);
            if (tmp<energy) {
                    energy=tmp;
                loopmatrix[i][j][0]=ip;
                loopmatrix[i][j][1]=jp;
            }
          }
            else {  /* Interior */
                tmp=Inter(i,j,ip,jp)+v(ip,jp);
            if (tmp<energy) {
                    energy=tmp;
                loopmatrix[i][j][0]=ip;
                loopmatrix[i][j][1]=jp;
            }
          }
        }

    for (k=i+1; k<j-1; k++) {  /* bifurcation */
        tmp=w(i+1,k)+w(k+1,j-1);
      if (tmp<energy) {
          energy=tmp;
          /* When path[][]=-1 && loop[][][0]==loop[][][1], it is a
           * special case of bifucation within a V */
          loopmatrix[i][j][0]=k;
          loopmatrix[i][j][1]=k;
```

```
      }
    }

    (energy>infinity? infinity:energy);
    Vij[i][j]=energy;
    return energy;
}

/*************************************************************
 *  Wij Function
 *************************************************************/
float w(int i, int j) {
    int k;
    float energy, tmp;

    if (pathmatrix[i][j] > -10)  /* already calculated */
        return Wij[i][j];

/* w(i,j) should always call v(i,j) */
    if ((j-i)<HAIRPINLEN+1) {
        v(i,j);
        energy=0;
        pathmatrix[i][j]=-5;
    }
    else {
        energy=v(i,j);
        if (! canonical_pairs(pseq[i],pseq[j]))
            pathmatrix[i][j]=-5;
        else
            pathmatrix[i][j]=-1;

        for (k=i; k<j; k++) {  /* bifurcation */
            tmp=w(i,k)+w(k+1,j);
            if (tmp<energy) {
                energy=tmp;
                pathmatrix[i][j]=k;
            }
        }
    }

    (energy>infinity? infinity:energy);
    Wij[i][j]=energy;
    return energy;
}
```

Appendix P: Source Code: z4ct.c

```
/*******************************************************************
# Name  : z4ct.c
#
# Input :
#
# Output:
#
# Author: Lisa Yu
#
# Course: CS 298
#
# Date  : Fall 2006
#
# Desc  : This program implements the Zuker's minimum free energy
#         methods [ZMT98] for RNA secondary structure prediction.
#
#
#
#******************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include "fasta.h"
#include "ct.h"
#include "freeEnergy.h"

void push(int i, int j);
void printpairs(int seqlen, char comment[]);
void readMfoldDat();

char comment[63];
int stack[MAXLEN][2];
int stacktop=-1;

main (int argc, char* argvs[]) {
    int i, j, k, seqlen;

/* read RNA sequence and store to char string psep[] */
    seqlen=readFasta(argc,argvs);

/*  pathmatrix[i][j] = -10 means it has not been calculated */
    for (i=0; i<seqlen; i++)
        for (j=0; j<seqlen; j++) pathmatrix[i][j]=-10;

    readMfoldDat();

    clock_t start, end;
    clock_t cpuclocks;
    start=clock();
/*******************************************
 * USE one of the two ways to calculate Wij
 *     WAY 1) Parallzation calculation
```

```
 *     WAY 2) grand recursive calls
 *******************************************/
/****** WAY 1) **************/
    for (k=0; k<seqlen; k++)
#pragma omp parallel for
       for (i=0; i<seqlen-k; i++)
           w(i,i+k);

/****** WAY 2) **************/
/*
    w(0,seqlen-1);
 */

    end=clock();
/**********************************************************
 * In CYGWIN system, CLOCKS_PER_SEC =1000,
 *                   which means clock_t will not
 *                   overflow within (2^31-1)/1000 seconds,
 *                   or ~596.5 hours/24.8 days.
 * In Linux, CLOCKS_PER_SEC = 1000000.
 *                   So it overflows at ~ 36 minutes.
 *********************************************************/
#ifdef __CYGWIN__
    if (end > start)
        cpuclocks=end-start;
    else
        cpuclocks=end-start+(clock_t)(pow(2,sizeof(clock_t)*8-1)-1);
#endif

    push(0,seqlen-1);
    fprintf(stderr,"Free energy: %g\n", Wij[0][seqlen-1]);
    printf("Sequence length: %d\n", seqlen);
#ifdef __CYGWIN__
    printf("CPUtime: %g\n",((double) cpuclocks) / CLOCKS_PER_SEC);
#endif
    sprintf(comment,"Zuker dE = %g kcal/mole", Wij[0][seqlen-1]);
    printpairs(seqlen,comment);
    return(0);
}


/*********************************
 * Push (i,j) onto the stack
 ********************************/

void push(int i, int j) {
    stacktop++;
    stack[stacktop][0]=i;
    stack[stacktop][1]=j;
}

/*******************************
 * Pop from the stack
 ****************************/

int pop(int *i, int *j) {
    if (stacktop>=0) {
```

```
        (*i)=stack[stacktop][0];
        (*j)=stack[stacktop][1];
        stacktop--;
        return(1);
    }
    else {
        return(0);
    }
}

/********************************************
 * Trace-back stage and print the base pairs
 ********************************************/

void printpairs(int seqlen, char comment[]) {
    int i,j,k,ip,jp;
    basepairs_init();
    while (pop(&i,&j)) {
        if (i>=j) continue;
        k=pathmatrix[i][j];
        if (k==-1) {
            basepairs_store(i,j);
          ip=loopmatrix[i][j][0];
          jp=loopmatrix[i][j][1];
          if (ip!=jp)
                push(ip,jp);
          else {
            push(i+1,ip);
            push(ip+1,j-1);
          }
        }
        else if (k>=0) {
            push(i,k);
            push(k+1,j);
        }
    }

/********************************************************
 * Now, generate .ct format output for Zuker's sir_graph
 * and .dot format for RNAdistance of the Vienna package
 ********************************************************/
    basepairs_print(pseq, comment);
}
```