5-2009

Predicting RNA Secondary Structures By Folding Simulation: Software and Experiments

Joel Omni Gillespie
Utah State University

PREDICTING RNA SECONDARY STRUCTURES BY FOLDING SIMULATION:

SOFTWARE AND EXPERIMENTS

by

Joel Gillespie

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Minghui Jiang Major Professor	Nicholas Flann Committee Member
Stephen Allan Committee Member	Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

ABSTRACT

Predicting RNA Secondary Structures by Folding Simulation:

Software and Experiments

by

Joel Gillespie, Master of Science

Utah State University, 2009

Major Professor: Dr. Minghui Jiang
Department: Computer Science

We present a new method for predicting the secondary structure of RNA sequences. Using our method, each RNA nucleotide of an RNA Sequence is represented as a point on a 3D triangular lattice. Using the Simulated Annealing technique, we manipulate the location of the points on the lattice. We explore various scoring functions for judging the relative quality of the structures created by these manipulations. After near optimal configurations on the lattice have been found, we describe how the lattice locations of the nucleotides can be used to predict a secondary structure for the sequence. This prediction can be further improved by using a greedy, 2-interval post-processing step to find the maximum independent set of the helices predicted by the lattice. The complete method, DeltaIS, is then compared with HotKnot, a popular secondary structure prediction program. We evaluate the relative effectiveness of DeltaIS and HotKnot by predicting 252 sequences from the Pseudobase Database. The predictions of each method are then scored against the true structures. We show DeltaIS to be superior to HotKnot for shorter RNA sequences, and in the number of perfectly predicted structures.

(176 pages)

ACKNOWLEDGMENTS

I would like to thank all those who have helped me through the process of writing my thesis. I want to thank Martin Mayne for his help and contribution on the Delta Library. I would like to thank Yuguang Zeng for his help in obtaining energy parameters for our scoring function, and Jeremy J. Neff for helping with gathering pseudoknot data from PseudodoBase. I would also like to thank the students of USU's Bioinformatics II Spring 2009 class, Shanhong Cheng, D. Scott Fechser, Ramoni O. Lasisi, Pedro Tejada, and Yu Zhou, for their help in evaluating the software and source code. I would also like to recognize and thank Dr. Minghui Jiang for his encouragement and guidance throughout my work with the RNA Project.

I want to thank all my friends for their encouragement, without which I doubt that I would have ever completed this project. Finally, I want to thank my best friend, my wonderful wife, Julie. She has been my great support during my years at the university and has been nothing but encouraging and supportive.

Joel Gillespie

CONTENTS

LIST OF TABLES

LIST OF FIGURES

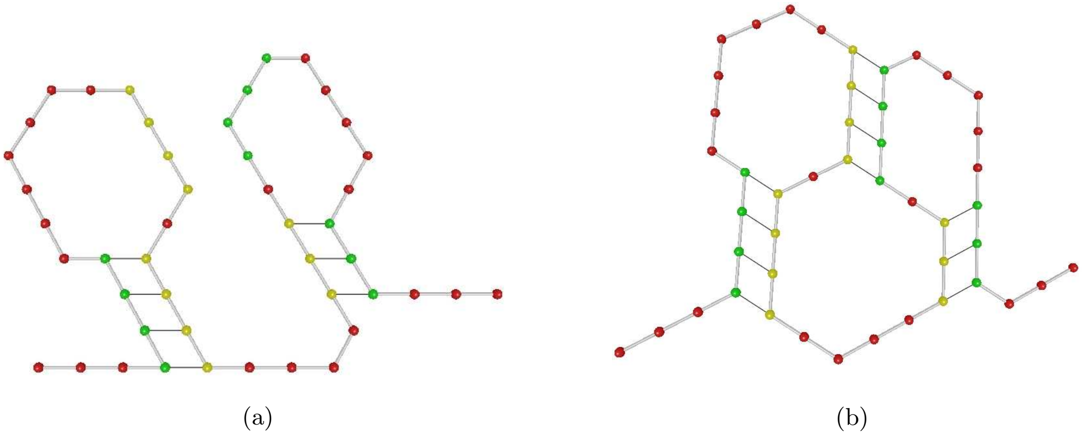
Figure Page

CHAPTER 1
INTRODUCTION

Ribonucleic acid (RNA) folding has become an increasingly important field of research within the fields of Bioinformatics and Computational Biology. It has reached this level of importance because of the relationship between the tertiary (3D) structure of a sequence and the function of the RNA sequence. Researchers believe that a sequence's tertiary structure is strongly related to its function, influencing whether a sequence facilitates the copying of a gene, transfers amino acids during translation, decodes mRNA, or performs RNA splicing and regulation. Because studying RNA sequences directly can be expensive and slow, using computational methods to predict how a sequence folds together, thus predicting its function, has become increasingly popular.

To solve the RNA folding problem, many different solutions have been proposed. Many of these solutions follow a straightforward dynamic programming paradigm. In these solutions, RNA folding is simulated using a scoring function. The scoring function solves for the best score for the entire RNA sequence by defining the best solution in terms of simpler sub problems. The computation is then organized to allow the final solution to be progressively built up from the base cases to the final solution. These solutions have proved to be relatively fast, $O(n^3)$, and relatively accurate. Indeed, because the final solution is built up from optimal solutions to smaller problems, the final solution can be guaranteed to be optimal for the given scoring function. Unfortunately, these solutions are generally unable to handle arbitrary RNA configurations.

For dynamic programming solutions to work, we must be able to build up from the smaller ones. This restriction limits what types of configurations can be solved. One common limit of dynamic programming solutions to the folding problem is the inability to handle pseudoknots, a sequence in which two or more sets of base pairings interleave with each other. In other words, if pairs (i_1, j_1) and (i_2, j_2) form a pseudoknot, then $i_1 < i_2 < j_1 < j_2$. In Figure 1.1 we show an example of a simple pseudoknotted structure. Pseudoknots are not generally handled by dynamic programming solutions because the


(a)

(b)

Figure 1.1. Example pseudoknotted sequence. (a) predicted structure without pseudoknots, (b) predicted structure with pseudoknots.

ability of the pairs to interleave means that sub-solutions are no longer independent. To understand how this affects the quality of structure prediction, we again refer to Figure 1.1. Part (a) of the figure shows a sequence prediction when pseudoknots are not supported, part (b) shows the same sequence correctly predicted with pseudoknots. In some cases, dynamic programming solutions have been able to handle specific types of pseudoknots, but this results in a greatly increased running time of $O(n^6)$, making this approach unfeasible for all but the smallest sequences.

To reduce the running time of dynamic programming solutions, heuristic algorithms emerged. These algorithms attempted to build a final solution in a greedy fashion. Instead of directly solving for the final solution, the final solutions were built by incrementally adding loops into the current structure. These solutions were faster and more flexible than the dynamic programming solutions, but they were no longer guaranteed to be optimal on the current scoring function. Additionally, the early heuristic approaches were too rigid; once a change was committed to the current structure, the change could not be undone. Recent heuristic approaches have largely removed this constraint. One recent solution to this problem, the HotKnots program created by Ren et al., solves this problem by maintaining multiple incomplete solutions. These solutions are then incrementally improved, giving preference to the most "energetically favorable substructures" [1].

We now present a new solution to the RNA folding problem. This solution builds off our proof-of-concept implementation presented in [2, 3]. Our method uses a combination of two unique approaches, 3D Triangular lattices and 2-Interval graphs, to predict the secondary structure of RNA sequences. First, we propose using simple *pull* moves to incrementally manipulate the structure of an RNA sequence on a 3D triangular lattice. After each pull move has been completed, the resulting arrangement is scored and either accepted or rejected using a *Simulated Annealing* approach. This approach allows the search space to be reduced from an infinite number of positions for any given nucleotide, to a relative handful of neighboring locations for each consecutive nucleotide. After Simulated Annealing has completed, and a reasonable approximation of the 3D structure of the sequence has been obtained, we use the lattice adjacencies to create a candidate set of base pairings. Second, the candidate set of base pairs is converted into a 2-Interval graph. The 2-Interval graph is used to model the helices, and to make the final secondary structure prediction. The remainder of this paper is organized as follows. In Chapter 2 we discuss the features of the Delta Library and Toolset (the *Delta* portion of DeltaIS). The Delta Library consists of the 3D Triangular lattice, pull moves, and everything necessary to simulate RNA folding on the 3D Triangular lattice. The Delta Toolset consists of two tools: *Fold*, a tool used for folding simulation, and *Show*, a tool used for structure visualization. Chapter 3 discusses the process of secondary structure prediction (the *IS* portion of DeltaIS). We present the shortfallings of predictions made directly from the 3D Triangular lattice and introduce the 2-Interval graph. Chapter 4 presents a detailed description of the data, experiments, and experimental results. Conclusions are presented in Chapter 5.

CHAPTER 2
RNA FOLDING SIMULATION

To predict the tertiary (3D) structure of an RNA sequence, we utilize the Delta Library and Toolset. The Library provides the functions and data structures upon which the tools are then built. In this Chapter we discuss this library in detail. Following our presentation of the Delta Library, we present two tools, FOLD and SHOW, which were developed using the library. We discuss how these tools can be used to directly predict the tertiary structure of an RNA sequence.

2.1 The Delta Library

The Delta Library consists of a set of functions intended to manipulation RNA. Included in the library are all the functions needed to input the RNA sequence, create a specified or default configuration for the sequence, manipulate the sequence in a safe manner, and then write the sequence out to file. The core of the Delta library is the 3D triangular lattice. The input RNA sequence is wrapped on the lattice; the sequence manipulations take place on the lattice; and the final structure is shown in 3D using the lattice.

2.1.1 3D Triangular Lattice

The 3D triangular lattice can seem complicated, but by comparing it with simpler lattices we can easily understand both how the 3D triangular lattice is constructed, and why it is useful. We start first with a simple Square lattice in 2D, an example of which is shown in Figure 2.1.

The square lattice has exactly two axes: the familiar $\vec{X}$ and $\vec{Y}$ axes. This lattice is simple, intuitive, and can be understood by anyone familiar with 2D graphing. Each point on the square lattice can be referenced using a single point as the origin and then counting the displacement in the $\vec{X}$ and $\vec{Y}$ directions. Even though the 2D lattice is simple, it has proved useful in protein-folding algorithms and should not be discounted because of its simplicity. Nevertheless, the Square lattice suffers from two major limitations. First, it

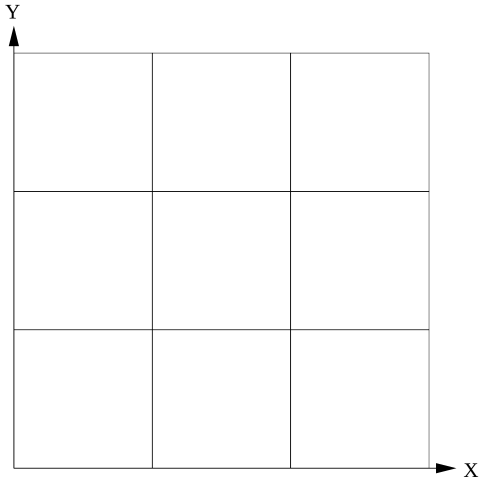


Figure 2.1. Square lattice.

cannot approximate anything resembling a 3D structure. This inibility is to be expected because the square lattice only models 2D interactions. The second limitation of the square lattice is the parity problem. Given a base with index n, the parity problem is that base n cannot bond with a base of index $n + 2$. Figure 2.2 demonstrates the problem. In Figure 2.2 we see a 2×2 section of the square lattice. We show a point, n, in the center of the lattice. From this point we had four choices of positions to place the $n + 1$ point. For our example, we placed the point in the right position, corresponding to the positive $\vec{X}$ direction. Assuming that we want to pair $n + 2$, we again have four choices of where to place $n + 2$. The left position is invalid, as the position is already occupied by n. The right position, while valid, is not adjacent and is 2 unit distances away from n. Finally, the two remaining directions keep the n and $n + 2$ points close, but neither of these $n + 2$ positions is adjacent to n. This parity problem would be of negligible consequence, except this parity problem exists for not only n and $n + 2$, but for n and *all* $n + 2k$ where k is any integer greater than zero. These limitations make the square lattice inadequate to accurately simulate RNA sequences.

Having shown that a square lattice is insufficient, we move to the 2D triangular lattice. To construct the triangular lattice, we start by defining a square lattice and then split each square into two equal halves. Each square is divided by connecting the bottom left point

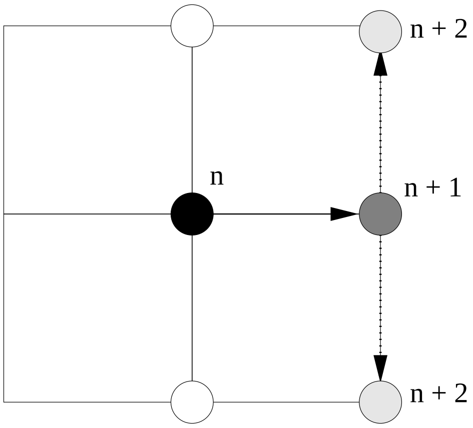


Figure 2.2. Parity problem of 2D square lattice.

of each square with the top right point of the same square. We define this new axis as the $\vec{U}$ axis. In other words, $\vec{U} = \vec{X} + \vec{Y}$. To complete the lattice, we skew the lattice until the angle between $\vec{X}$ and $\vec{Y}$ is exactly $\frac{2\pi}{3}$. Manipulating the lattice in this way skews each line segment on the lattice until each segment is exactly 1 unit distance. The process of creating the lattice is shown in Figure 2.3.

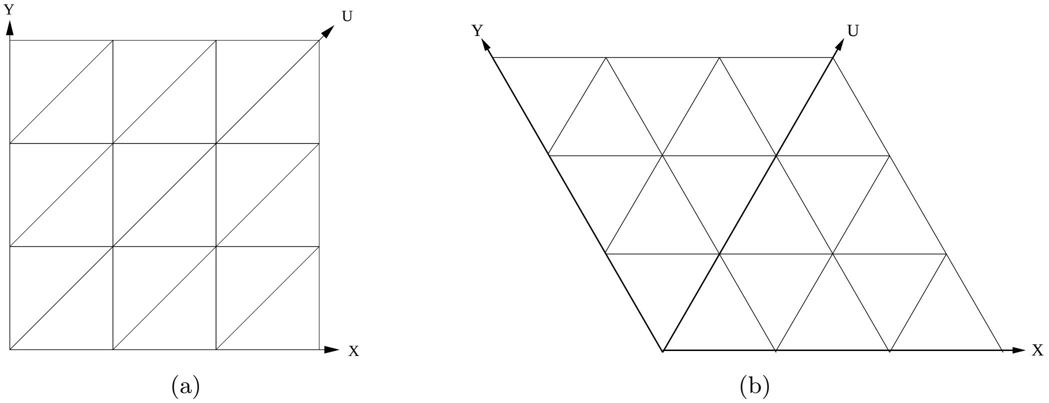


Figure 2.3. Creating the 2D triangular lattice. (a) The divided square lattice with the new U axis. (b) The completed 3D triangular lattice.

The 2D triangular lattice is a significant improvement over the square lattice for two reasons: (1) it is more flexible, and (2) it does not suffer from the parity problem. To see how the 2D triangular lattice increases flexibility, we consider the number of neighbors for each lattice. The square lattice allowed a maximum of four neighbors. Using the additional axis $\vec{U}$, the triangular lattice allows any given point to have six neighbors. In addition to the increased number of neighbors, the triangular lattice does not suffer from the parity problem. Using the 2D triangular lattice, we again attempt to realize a configuration where n is adjacent to $n+2$. Figure 2.4 demonstrates an easy solution to this problem. We start by placing point n. This point can be placed on any lattice position. After placing the first point, we again choose to place point $n+1$ in the $\vec{X}$ direction. Just as in our previous example, the second position could have been any of the other possible positions, as by rotation all the secondary positions are identical. From this point we now have 2 positions adjacent to n in which we can place the $n+2$ point: the positive $\vec{Y}$ and negative $\vec{U}$ positions. We can see that by using the 2D triangular lattice, any two bases can be arranged to form a pair. Allowing any two bases to potentially be paired allows the triangular lattice to predict a much wider range and much more realistic set of configurations. It is also interesting to note that while the $\vec{U}$ axis defines additional adjacencies, each point on the lattice can still be uniquely identified using only the $\vec{X}$ and $\vec{Y}$ axes.

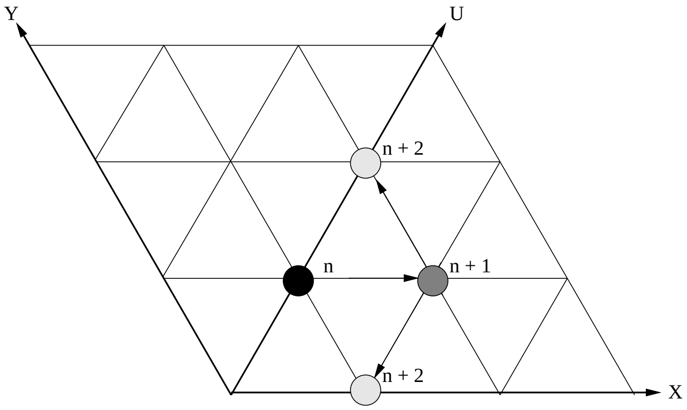


Figure 2.4. The 2D triangular lattice solves the parity problem of the square lattice. Shown are two possible configurations in which n pairs with $n+2$.

We have demonstrated the significant advantages of the 2D triangular lattice over the square lattice, but the lattice is not yet powerful enough to predict a tertiary structure. To accomplish this task, we need to expand the triangular lattice into 3D. We can stop this combining the square and 2D triangular lattices in 3D. We start with a cubic lattice. The cubic lattice is simply the square lattice extended into 3D. Instead of being limited to the $\vec{X}$ and $\vec{Y}$ axes, we have expanded the lattice to include the $\vec{Z}$ axis as well. Just as we did with the original square lattice, we will now skew the $\vec{Z}$ axis of the cubic lattice until it forms a $\frac{2\pi}{3}$ angle with both the $\vec{X}$ and $\vec{Y}$ axes. Because only the $\vec{Z}$ axis is skewed, the $\vec{X}$ and $\vec{Y}$ axes remain perpendicular to each other, the distance between the top and bottom planes is reduced to $\frac{\sqrt{2}}{2}$ unit distance, and each consecutive XY-plane is shifted by $-\frac{1}{2}$ in both the $\vec{X}$ and $\vec{Y}$ directions. Finally, as a last step, we complete the 3D triangular lattice by creating the auxiliary axes $\vec{U}$, $\vec{V}$, and $\vec{W}$, defined respectively as:

$$\vec{U} = \vec{X} + \vec{Z}$$

$$\vec{V} = \vec{Y} + \vec{Z}$$

$$\vec{W} = \vec{X} + \vec{Y} + \vec{Z}$$

In Figure 2.5 we show the completed 3D triangular lattice. To make this figure easier to read, we have show only the $\vec{X}$, $\vec{Y}$, and $\vec{Z}$ axes. Each point with its respective coordinate (X, Y, Z).

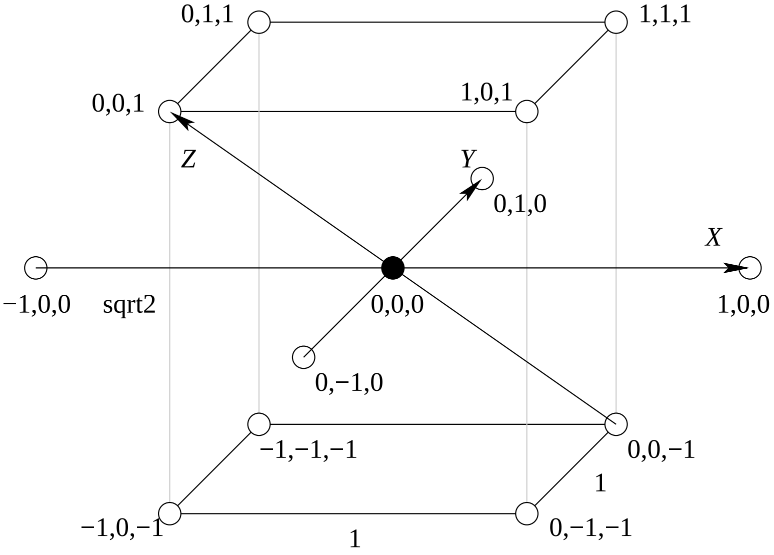


Figure 2.5. The 3D triangular lattice with auxiliary axes suppressed.

Even with the addition of 3 axes, each point on the 3D triangular lattice can be specified using only the $(\vec{X}, \vec{Y}, \vec{Z})$ displacement from an origin point, and, like the 2D triangular lattice, the 3D triangular lattice does not suffer from the parity problem. The 3D triangular lattice is also much more versatile than the square, 2D triangular, or cubic lattices. Each point on the 3D triangular lattice has 12 neighbors: $\pm\vec{X}, \pm\vec{Y}, \pm\vec{Z}$ and the adjacency vectors $\pm\vec{U}, \pm\vec{V}$, and $\pm\vec{W}$. Each of these features make the 3D triangular lattice a more compelling choice for simulation work. This lattice can directly predict 3D structures and has enough flexibility to predict a huge number of configurations, thus making it possible to predict much biologically realistic configurations.

2.1.2 Sequence Representation and Data Structures

The Delta Library provides functions to read in, manipulate, and then save RNA sequences and sequence structure. To understand this process, we first discuss the external representation of an RNA sequence. Each sequence is represented as two sequences: a sequence of `bases`, and a sequence of `turns`. The `bases` sequence identifies the organic compounds that make up the sequence, and the `turns` sequence defines the spacial relationship between these compounds on the lattice. The `bases` sequence is the simplest, and it represents a direct mapping of the nucleotides Adenine, Cytosine, Guanine, and Uracil to their common abbreviations of A, C, G, and U, respectively. The `turns` sequence is very similar, with a set of twelve characters ($\{X, x, Y, y, Z, z, U, u, V, v, W, w\}$) representing the six axes, and the lower and upper cases corresponding to the negative and positive directions on those axes, respectively.

Internally, each RNA sequence is represented as arrays of points, bases, and turns. The `bases` and `turns` arrays are used to directly store the external representation; that is, a character array is used as the internal structure for storing both the base and turn sequences. In addition to the external information, each nucleotide has a location on the lattice and in cartesian space. This location information is stored inside the `points` array. The `bases` and the `points` together form the sequence's *configuration.* When an RNA sequence is read into Delta, the `turns` sequence is translated into lattice coordinates. This translation is done in two steps. Step one, we place the first nucleotide at the origin point

$(0, 0, 0)$. Each subsequent nucleotide then has its location assigned by adding the lattice coordinates of the previous nucleotide to the vector corresponding to the turn between them. We define a function `axis_c2v(t)` to transform each individual turn (`t`)into a vector (Mapping shown in Table 2.1).

Table 2.1. Mapping of turns to vectors.

w	$(-1, -1, -1)$
v	$(0, -1, -1)$
u	$(-1, 0, -1)$
z	$(0, 0, -1)$
y	$(0, -1, 0)$
x	$(-1, 0, 0)$
X	$(1, 0, 0)$
Y	$(0, 1, 0)$
Z	$(0, 0, 1)$
U	$(1, 0, 1)$
V	$(0, 1, 1)$
W	$(1, 1, 1)$

Now, given an RNA sequence S of length n and a `turn` sequence T of length $n-1$, we can define the `points` array for P, where $P[i]$ is defined to be the lattice coordinates of the ith point, and $T[i]$ is the vector representation of the ith element in the turn sequence. We now have the following:

$$P[i]_X = \begin{cases} 0 & i = 1 \\ P[i-1]_X + T[i-1]_X & i > 1 \end{cases}$$

$$P[i]_Y = \begin{cases} 0 & i = 1 \\ P[i-1]_Y + T[i-1]_Y & i > 1 \end{cases}$$

$$P[i]_Z = \begin{cases} 0 & i = 1 \\ P[i-1]_Z + T[i-1]_Z & i > 1. \end{cases}$$

As part of the process of reading in the `bases` and `turns`, Delta ensures that the sequence forms a `valid` configuration. A valid configuration is any configuration of points resulting in a non-intersecting walk along the 3D triangular lattice. To ensure that a given configuration is valid, Delta verifies that a lattice location is empty before inserting a point into the location. In order to make this check efficient, we implemented a hashtable to ensure an expected constant running time. The hashtable is initialized to be twice the size of the input sequence length to ensure that collisions are rare. When collisions do occur, they are handled using a linked list. To verify a lattice location is empty—the lattice coordinates—the target coordinates, of any candidate lattice location are hashed using a formula adapted from [4]:

$$h(X, Y, Z) = |2na \cdot (((aX + Y) \cdot a) + z)| \tag{2.1}$$

where $a = \frac{\sqrt{5}-1}{2}$, and n is the length of the RNA sequence. This hashcode is then used as the key (index) into the hashtable. If the key location is not empty, we check each entry in the locations linked list to see if the entry coordinates match the lattice coordinates of the current point. If the coordinates match, the configuration is invalid, and the input turn sequence is replaced with the default *stem-loop* configuration. If the key location is empty, or none of the entry coordinates match the lattice coordinates of the current point, then the configuration is valid.

2.1.3 Structural Manipulation

In addition to providing a structure on which to manipulate RNA sequences, the Delta Library also provides a set of functions with which to manipulate them. These functions are based on the *Pull* move concept first introduced by [5] for use on a simple square lattice. The pull move has since been adapted for use on a hexagon lattice [6], and we here adapt it for use on the 3D triangular lattice. A pull move is a simple, reversible way of moving between valid configurations while affecting as few nucleotides as possible. Each pull move consists of the index of a nucleotide to be moved, (i), and the vector, (t), along which it should be moved. The vector, t, corresponds to a direction along one of the six axes, and for convenience, we use the same character set as the `turn` array. We now define the function, `p(i, d)`, to mean the lattice location that corresponds to walking from the lattice location of the ith nucleotide in the direction of d. To complete a pull move, a nucleotide is pulled to `p(i, d)`, and its neighbor bases are updated to make sure they remain adjacent.

Before we perform the move itself, we must first verify the resulting configuration is valid. This validation is done by the `test(i, d)` function. The `test` function takes the candidate `move` as input and first verifies the location `p(i, d)` is unoccupied. A configuration where two or more nucleotides occupy the same lattice location is invalid, so a pull move resulting in such a configuration is also invalid. Next, `test` verifies that i is either an endpoint of the sequence, or that an *anchor* point exists for the move. Given a point i in the sequence S, an anchor point is either $S[i-1]$ or $S[i+1]$, and it must be adjacent to $p(i,d)$. The anchor point is so named because it indicates the side of the sequence that is unchanged if the pull move is executed. The side opposite of the anchor will be affected by the pull move.

After verifying that the move is valid, the function `pull(i, d)` can be executed. The `pull` function executes the pull move by first moving $S[i]$ to the location `p(i, d)`. Assuming the pull move (i,d) was valid, we have one of three cases:

1. $S[i+1]$ and $S[i-1]$ are both adjacent to `p(i, d)`. This special case is called a *flip*. In this case the pull move is completed immediately after moving $S[i]$, and no additional updating is required.

2. $S[i+1]$ was adjacent to `p(i, d)`. In this case $S[i+1]$ is the anchor, and the nucleotides $S[i+k]$ $(k \geq 1)$ are not moved. Each element opposite the anchor point ($S[i-k]$ where $k \geq 1$) may need to be updated.

3. $S[i-1]$ was adjacent to `p(i, d)`. In this case $S[i-1]$ is the anchor, and the nucleotides $S[i-k]$ $(k \geq 1)$ are not moved. Each element opposite the anchor point ($S[i+k]$ where $k \geq 1$) may need to be updated.

After $S[i]$ is moved to `p(d, i)`, the point $S[i]$ and the point opposite the anchor point (for example, $S[i-1]$, if $S[i+1]$ was the anchor), are tested for adjacency. If these points are adjacent, the pull move is completed. If the points are not adjacent, then we continue moving consecutive points until this condition is true. In the example where $S[i+1]$ is the anchor, $S[i-1]$ is moved to the former location of $S[i]$. We then check to see if $S[i-1]$ is adjacent to $S[i-2]$. If not, we move $S[i-2]$ to the former location of $S[i-1]$. This process is continued for each pair $S[i-k]$ and $S[i-k-1]$ $(k \geq 1)$ until all pairs are adjacent and we again have a valid configuration. The process for completing the pull move in the case that $S[i-1]$ is the anchor is exactly symmetric, with each pair $S[i+k]$ and $S[i+k+1]$ $(k \geq 0)$ being tested for adjacency, and $S[i+k+1]$ being moved to the former location of $S[i+k]$ when the pairs are not adjacent.

After completing a pull move, we sometimes want to undo the pull move. Fortunately, as mentioned previously, pull moves are completely reversible. Reversing a pull move is as simple as knowing the last base affected by the pull move and knowing the location from which the point originated. To understand how this works, we observe that while several nucleotides may be moved during the course of a pull move, only one new lattice location is occcupied. With the exception of the first nucleotide, all bases are moved to the former location of another nucleotide. To undo a move, the last nucleotide is pulled back to its original location. This change, a valid pull move, then causes all the other nucleotides to be sequentially moved back to their original locations. This process is demonstrated in Figure 2.6. In this figure we see two valid configurations. Each configuration has a line extending from one of the nucleotides. When pulled in the direction of the arrow, the configuration is transformed into the other configuration. By examining this figure, we can see both how reversible a pull move is, as well as how small a change a single pull move makes to the configuration.

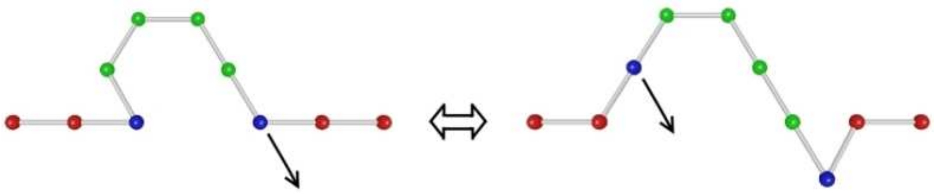


Figure 2.6. Example pull moves. Pulling a nucleotide in the direction of the arrow pulls one configuration into the other.

We have discussed what a pull move is and demonstrated its use within the Delta library. In addition to the use described above, there are two additional, noteworthy properties.

The first noteworthy property of pull moves is that a single pull move moves a relatively small number of bases [5]. This locality of change makes pull moves ideal for folding biological sequences because pull moves can be used to make small, incremental changes to the structure. These changes can then be evaluated by some predefined criteria and then either accepted or rejected without much effort being spent on any particular move.

As part of our research into the 3D triangular lattice, we verified that this general property of pull moves holds true for our specific implementation. To verify the property held true, we created five sequences of lengths 32, 64, 128, 256, and 512. Each of these sequences was loaded into Delta and subjected to $10^5 n$ random pull moves (starting from a straight line configuration). After these initial moves were completed, each sequence was subjected to $10^5 n$ additional random pull moves. For each of the moves in the second set of $10^5 n$ moves, the total number of bases moved was recorded. We present the results of this test in Table 2.2. These results demonstrate that the number of elements moved by a single pull move is indeed a small constant.

Table 2.2. Statistics on point displacement for a single pull move.

Sequence Size (n)	Average	Standard Deviation
32	2.40625	1.29164
64	3.79688	2.65581
128	4.10938	3.21716
256	3.36328	1.97754
512	3.48438	2.40347

The second important property of pull moves is that they are *complete*. This property means that any valid configuration can be reached from any other valid configuration through a series of pull moves. Given that a pull move can be undone, this property is very easy to demonstrate. One can easily imagine taking a valid configuration and pulling the first or last nucleotide away from the rest of the sequence. Through repeatedly pulling this nucleotide in a single direction, we eventually reach a straight line configuration. We can use this straight line configuration as the link between any configuration (a source configuration) and any desired configuration (the target configuration). We first pull the source configuration into a straight line configuration. Next we pull the target configuration into a straight line configuration. These two sequences are now, for all intents and purposes, identical. We know that pull moves are reversible, so if we can pull the target configuration to a straight line, then we can pull the straight line back to the target configuration.

2.1.4 Delta Library Quick Reference

As has been discussed, the Delta Library is designed to allow the bioinformatics community at large to develop their own tools for structure prediction on the 3D triangular lattice. The Delta Library provides all the functions necessary to input, manipulate, and output sequences on the 3D triangular lattice. Appendix D contains the source code for the DELTA library, as well as the source for the tools designed from it. Appendix G containes the makefiles used to compile the library and build the tools.

When using the library to create your own tool, you will, at a minimum, need to use several of the provided functions. These functions, along with a short description of the function, are listed in Table 2.3. For a more complete description of Delta library functions, please see Appendix C for the website[1] or see Appendix B for the Application Programming Interface.

Table 2.3. Delta library minimum required functions.

Function	Description
`input_bases_turns`	Reads in a sequence or sequence 2 file. This function initializes the bases and turns arrays. Returns the length of the bases array.
`turns_to_points`	Populates the points array based on the turns array. This is the function used to wrap the points onto the 3D Triangular lattice.
`points_to_turns`	Used to save a desired configuration. Sets the turns array based on the points array.
`output_bases_turns`	Writes the bases and turns array out to file, creating a sequence 2 file.

[1] `http://www.cs.usu.edu/∼mjiang/rna/`,

2.2 Folding Simulation

Delta's structural manipulation tool is one of the major accomplishments of this work. This tool, Fold, uses the 3D triangular lattice to fulfill two purpose. First, it simulates tertiary structure interactions to predict a given sequence's tertiary structure. This structure can then be used to predict the given sequence's secondary structure. Second, Fold can be used to reproduce, or reconstruct, a previously defined secondary structure. To accomplish these purposes, we make use of *Simulated Annealing*. This technique, first proposed by Nicholas Metropolis [7], has become popular in other folding algorithms [8, 9, 10, 11, 12, 13]. In this section we first define the Prediction and Reconstruction Modes. These two modes correspond to the two purposes of the structural manipulation tool. We then discuss using Simulated Annealing in the tool itself. Finally, we provide a quick reference for Fold.

2.2.1 Prediction and Reconstruction Mode

Predicting a tertiary structure and reconstructing a secondary structure in 3D are quite similar. In fact, to accomplish either objective, we simply change the scoring function. In Prediction mode, the tool uses a scoring function designed to emulate, to the greatest degree possible, the tertiary interactions of the individual nucleotides composing the sequence that Fold is trying to predict. The more closely the structure approximates the actual interactions, the higher the score it receives. In Reconstruction Mode, the tool uses a simpler scoring function. This function returns a score based on how much of the desired structure has been realized. We discuss each scoring function in detail.

2.2.1.1 Prediction Mode

The purpose of Prediction mode is to reliably predict a sequence's secondary structure by simulating its tertiary structure interactions. It is thus logical that our scoring function starts by looking at the pairs formed by individual nucleotides. Although exceptions exist, RNA bonds generally consist of two types of pairings: the Watson-Crick pairs, $\{AU, CG, GC, UA\}$, and the Wobble pairs, $\{GU, UG\}$. We started our search for a good scoring function by awarding a score for each pair type, awarding a score of 8 for each GC (or CG)

pair, 5 for each AU (UA) pair, 3 for each GU (UG) pair, and a penalty of -2 for all other pairings [2]. This simple scoring scheme was derived from the Nussinov scoring matrix, which, in turn, was based off the actual number of bonds formed by each of the pairs [14]. This scoring scheme proved to be relatively ineffective. This was entirely expected, as the scoring scheme was entirely too simplistic.

After determining the pair-based model was too simple, we moved to a stack-based mode. This stack-based scoring scheme models the idea that RNA sequences fold to a global structure minimizing the free energy of the structure [15]. According to model [16], any given RNA sequence can be reduced to a series of loops. These loops have an independent free energy associated with them. In order to fold to the minimum free energy structure, stacking pairs (pairs with negative free energy) are required. Our stack scoring scheme attempted to build up an approximation of the stacking pairs by using the following steps:

1. Determine if two adjacent bases form a pair. While the 3D triangular lattice allows sharp turns and any two arbitrary bases to be adjacent, RNA sequences are not nearly as flexible, requiring a separation of at least three bases [17]. We thus only consider two lattice adjacent bases to be paired if they are separated by at least three bases in the RNA sequence.

2. Calculate the sum of all pairs in which the base (i) could participate by virtue of being adjacent to the pair on the lattice and being separated by the minimum number of bases in the sequence:
$$sum(i) = \sum_{(i,j)} |s(i,j)| \tag{2.2}$$
where $s(i,j)$ is the score of the base pairing between base i and base j, as defined in our original pair-based scoring scheme.

3. Calculate the *normalized score* for each pairing:
$$s'(i,j) = \frac{s(i,j)}{sum(i)} \cdot \frac{s(i,j)}{sum(j)} \cdot s(i,j) \tag{2.3}$$

In this scoring scheme, individual bases are not required to participate in exclusive pairings. In other words, one base may participate in several pairings. The normalized score is designed to scale the pair score achieved by the pair (i, j) by how committed each of the bases is to the pairing.

4. Calculate the *stacking score* for each stacking pair (a stacking pair consists of any two pairings (i, j) and $(i + 1, j - 1)$ whose normalized scores are both positive):

$$s''(i, j) = s'(i, j) + s'(i + 1, j - 1) \tag{2.4}$$

5. Calculate the sum of all stacking scores.

While this method of computing scores proves to be relatively effective when compared with the previous method, it requires a great deal of extra computation. In the end this stack-based scoring scheme is just an estimation of the values we really want to compute. To more accurately model stacking interactions on the lattice, we move to our final scoring scheme. This scoring scheme is based on direct computation of the structure's free energy. Instead of calculating a score for a pair of bases and then trying to extrapolate the stack score based on commitment, we use the MFOLD [18] free energy parameters shown in Table 2.4.

Table 2.4. MFOLD Stacking Energy.

	AU	*CG*	*GC*	*UA*	*GU*	*UG*
AU	−0.90	−2.20	−2.10	−1.10	−0.60	−1.40
CG	−2.10	−3.30	−2.40	−2.10	−1.40	−2.10
GC	−2.40	−3.40	−3.30	−2.20	−1.50	−2.50
UA	−1.30	−2.40	−2.10	−0.90	−1.00	−1.30
GU	−1.30	−2.50	−2.10	−1.40	−0.50	1.30
UG	−1.00	−1.50	−1.40	−0.60	0.30	−0.50

The MFOLD energies were obtained by first downloading MFOLD and then parsing out the data from the `coaxial.dat` and `stack.dat` files. Throughout the rest of this paper, we will reference the MFOLD stacking energy table as the $e[i][j]$ table, and we will index into the array by using two indexes, i and j. These two indexes represent the two pair type that will have its stacking energy returned. For example, if the two pairs, (i, j), are the pairs AU and UG respectively, the free energy of the stacked pairs can be obtained by finding AU on the left, UG on the top, and then reading the energy from the table. In this case we see the stacking energy of the pairs AU and UG is -1.40.

Having the free energies of each stacking pair pre-calculated allows us to simplify the calculations considerably, while at the same time increasing the accuracy of our scoring schema. We increase our accuracy by defining a new function: `stack_score_ij(i, j)`. This function takes, as input, the indexes of two points from the RNA sequence S. The function first determines if the pairs (i, j), $(i+1, j-1)$, and $(i-1, j+1)$ are canonical pairs (Watson-Crick or Wobble pairs). For simplicity in explaning our scoring scheme, let us refer to the first pair (i, j) as simply pair k, and pairs $(i-1, j+1)$ and $(i+1, j-1)$ as pairs l and m respectively. `stack_score_ij(i, j)` then assigns scores as follows:

1. If k is not a canonical pair, then the score of the pair is 0.

2. If the first pair, k, is a Watson-Crick pair, then the stack score is non-zero only in the case where l and/or m is also a Watson-Crick pair. In this case the stack score for k is the summation of the stacking energies of k and the other Watson-Crick pair(s).

3. If k is a Wobble pair, then the stack score is non-zero only in the case where both l and m are Watson-Crick pairs. In this case the stack score is twice the summation of the stacking energies lk and km. We double the score for this case because the Wobble pair will only be scored once, while the Watson-Crick pairs could be counted multiple times.

After using `stack_score_ij(i, j)` to calculate the Stack Score for a given pair (i, j), we have only to determine how to use the individual pair scores to calculate the RNA configuration's global score. The calculation of the global score is computed by first using `stack_score_ij(i, j)` to calculate the score of each single nucleotide i, and then using these individual scores to compute the global score. Given that any nucleotide could participate in multiple pairs, and thus participate in multiple stackings as well, we elected to check each nucleotide for possible pairings and then use the minimum of all positive scores. This system allows us to reward the annealer for finding good stackings, while simultaneously discouraging it from finding configurations that are too tightly clustered. We now have:

$$\text{stack\_score(i)} = \min\{\text{stack\_score\_ij(i, j) for all } j\}. \tag{2.5}$$

Finally, the RNA configuration's global score can be calculated by adding the `stack_score(i)` for all nucleotides in the sequence:

$$\text{Global Score} = \sum_{i=1}^{n} \text{stack\_score(i)}. \tag{2.6}$$

2.2.1.2 Reconstruction Mode

In Reconstruct mode, the scoring function no longer considers how nucleotides interact. Instead the scoring function returns a score based on how completely a specific set of pairs is realized. The set of pairs to be realized is first read from a file. The input file specifies the pairs in the format:

index$_1$ index$_2$

with the lower index listed first. To determine the score of a given configuration, the scoring function awards a score of 5 for each realized pair. The scoring function awards no penalty and no bonus for each pairing that the program was not asked to construct. In addition to the pair-based scoring, the reconstruct scoring is heavily based on the angles formed along the lattice. We emphasize the angles because real RNA sequences are not as flexible as adjacencies on the lattice make them appear. To encourage less intense angles, a turn penalty is defined:

$$\text{turn penalty} = f \cdot \frac{c}{n} \tag{2.7}$$

where n is the sequence length, c is set to 5 (the score for each realized pair), and f is a constant defined for each of the four possible angles (shown in Table 2.5). This penalty scheme is designed to encourage 180° and 120° angles and discourage 90° and 60° angles. These

Table 2.5. Turn penalty constant (f).

Angle	Penalty Factor(f)
180 °	0
120 °	0
90 °	1
60 °	4

turn penalties are small enough to be negligible relative to the pair score, but significant enough to cause the structure to take on a smoother configuration.

Having defined the two scoring schemes, we turn now to a discussion on Simulated Annealing, the method that uses these scoring schemes to find the desired structures.

2.2.2 Simulated Annealing

Simulated Annealing is a heuristic technique for finding near-optimal solutions to specific problems. It derives its name from the metallurgical process it is designed to simulate, the metallurgical process of Annealing. During the course of Annealing, metals are heated and then allowed to slowly cool. This process is used to increase the metal's ductility, decrease its hardness, or, in other words, transform it to a more desirable state [19].

Simulated Annealing is a heuristic technique designed to optimize solutions for a problem. We say optimize because Simulated Annealing is not guaranteed to find the optimal solution to any problem. Rather, Simulated Annealing takes solutions to a given problem and, through local modifications, attempts to find better solutions. Like all optimization methods, Simulated Annealing is dependent on a scoring function (previously discussed in Section 2.2.1). Because this function judges the quality of individual solutions, it is directly responsible for the quality of the final solution found by the simulated annealer.

Simulated annealing is set apart from other problem-solving methods by its use of *temperature* and *cooling functions*. Instead of repeatedly generating new states and then selecting only the best choices, a process known as *hill-climbing* or *gradient descent*, Simulated Annealing selects worse choices with some probability (p). This probability is the current *temperature*. When the Simulated Annealer is in a high temperature state, this means the annealer is more likely to accept a poor choice. This does not mean the annealer will not accept a better choice when presented the opportunity, only that when presented with a poor choice the annealer will still accept it with high probability. When the annealer is in a low temperature state, on the other hand, the annealer is less likely to accept a poor choice. If the termperature is sufficiently low, the Simulated Annealer will act exactly the same as a hill-climber, accepting only the best solutions. The cooling function is used to control the current temperature. As in the process that simulated annealing imitates, the "temperature" (or the probability of picking a bad choice) starts out high, and then is gradually lowered. The heating and cooling process may be repeated several times and may be much more complex than a simple linear decrease from a high to low temperature.

The purpose of the cooling function is to allow the annealer the freedom to escape from local maxima or minima, while still progressing towards a global maximum or minimum. Metropolis suggested computing the probability (p) of accepting a configuration at any given step as:

$$p_i = e^{\frac{-\Delta E}{kT}} \tag{2.8}$$

where kT is the cooling function [7]. We have modified our acceptance probability to:

$$p_i = 2^{\frac{\Delta S}{T_i}} \tag{2.9}$$

In this function, i is the current step, ΔS is the change in score, and T_i represents the temperature of the cooling function T at step i. Equation (2.10) defines the temperature T at step i:

$$T_i = \frac{c}{\log_2(\frac{n+i}{n})} \tag{2.10}$$

where n is the total number of steps, c is the scaling factor of the function, and $c = 1/\log_2 10$. These changes to Metropolis' acceptance function provide two key features. First, our c constant is used to scale the cooling function to guarantee the acceptance probability of the last step is exactly 10%. Second, the $1/\log$ temperature function is the fastest decaying function that converges to the optimal solution [20].

Now that we have a useful cooling function, we manipulate the function in order to make pull moves more efficient. Recall that at each step of the simulated annealing process, solutions are accepted with a probability based on the current step, and the relative improvement from the previous step. In practice this means an annealing program must keep track of the current score at any stage, and must be able to revert to the previous configuration if a move is not accepted. The most straightforward way of using the scoring function above is to execute a pull move using a scoring fuction. The scoring function calculates the score before the move, executes the move, and then calculates the score again. The current score is saved, and the change in score is returned. When a configuration is rejected, which is the most likely outcome of any given pull move, another pull move (the undo move) is taken, and score calculations are done again. Instead of following this process, we use a more efficent method. Under this method we use Equations (2.9) and (2.10) to solve for the *threshold* difference in score, ΔS. If, after a pull move has been taken, the score difference is at least the threshold difference, the move is accepted and the difference in score is returned. If the score difference is less than the threshold, then the pull move is immediately undone and a difference of zero is returned. In both cases the returned difference is then used to update the current configuration score. We can see this method is more efficient because it requires fewer pull moves, and less time spent calculating a new score. We solve for the threshold difference (ΔS) as follows:

$$\begin{aligned} p_i &= 2^{\frac{\Delta S}{T_i}} \\ \log_2 p_i &= \frac{\Delta S}{T_i} \\ T_i \cdot \log_2 p_i &= \Delta S \\ \frac{c}{\log_2(\frac{n+i}{n})} \cdot \log_2 p_i &= \Delta S \qquad (2.11) \end{aligned}$$

To finish the manipulation, we must solve for p_i. In our initial equation, Equation (2.9), p_i represented the probability that a given configuration would be accepted. After ΔS has been solved and p_i computed, a random real number between 0 and 1 is generated. If the probability is no more than p_i, then the configuration is accepted. In our new equation,

Equation (2.11), p_i represents our random real number. Instead of using the random real number to specify whether or not to accept a specific configuration, the p_i is now used to calculate the minimum difference in score that will be accepted.

After developing our cooling schedule and experimenting with the results, we were still not satisfied with the results. We investigated several other cooling functions but found none that improved upon our original results. After failing to find a better cooling function, we began searching for ways to improve the current cooling schedule. In [9], Schmitz and Steger suggested it iss not necessary to directly progress from start to finish along the cooling schedule curve. Instead of following a cooling schedule exactly (a *progressive* approach), Schmitz suggests better results can be achieved by selecting a random step from the cooling schedule. We then use the temperature corresponding to this step as the acceptance probability for the candidate solution. We refer to this strategy as the *sampling* approach. Instead of using either of these approaches directly, we have chosen to use a novel *mixing* strategy. Under this strategy the progressive approach is used half the time, while the sampling approach is used the remainder of the time. We chose to use a combination of the sampling and progressive approaches because we found that while both methods were effective individually, neither method was as good as some form of the mixing approach. This is the case because the benefit of the mixing strategy is that the sampling approach can be used to break out of local maximums in our RNA configuration search. We performed extensive experimentation to try to determine the exact mixing ratio, but the optimal strategy seems to be sequence dependent. Given there is no clearly superior ratio between the progressive and sampling strategies, we decided on the 50-50 split as a logical compromise between the 2 cooling strategies.

After defining the scoring and cooling functions for Fold, we now define how these functions are used in the folding process. The Fold tool first reads an RNA sequence into the library. The turn sequence, if it exists, is decoded, and the points array is allocated. If a turn sequence is not provided, the RNA sequence is initialized to a *stem-loop* configuration. After the initial configuration has been stored in the points array, the annealing process begins. The annealing process consists of a predetermined number of repeats, each consisting of a predetermined number of steps. At each step, a random pull move is generated. This pull move is first investigated using the `test` function to verify that the move is valid. For efficiency the `test` function also returns the indices of the first and last bases to be moved (stored as s and t, respectively). This information can then be used to determine the total list of bases that will be affected:

1. For convenience, we test to see if $s > t$. If it is, we swap s and t so that we know s is the lower index.

2. We subtract one from s if $s > 1$ and add one to t if $t < n$. This is done to make sure that any base that stacks with any moved base is included.

3. We gather into an array all points in the range $[s : t]$. Then, making sure that we do not include duplicate points, we include the immediate neighbors of the original points s through t. To ensure that points aren't duplicated, we keep a `flags` array that is parallel to the `points` array. When a point is added to the the neighbor array, we simply set the flag, indicating the point should not be included again.

After all affected bases have been gathered, we use Equation (2.5) to determine the stack score of each affected base. The sum of these scores is then computed and saved as the initial score of the affected bases. The pull move is then executed and the score calculation is repeated, generating a modified score for the affected bases. These scores' difference is then computed. Those configurations in which scores show an improvement are accepted automatically; the remaining configurations are accepted with the probability determined by Equation (2.9) and the "mixing" strategy. This process is repeated for each step of each repeat, with each repeat using the same cooling schedule. After all repeats have been completed, we use a specialized Doubling Step Mode to determine if the folding process is complete. This is done because, while we have endeavored to tune the parameters as much as possible, not all sequences require the same amount of manipulation to find an optimal solution. This may be caused by either the optimal structure's complexity or by the annealer's random nature.

To determine if additional folding is required in order to find the optimal solution, we simply keep track of how much improvement a sequence has undergone throughout the annealing process. Before we start annealing, we record the initial score. When we have finished all annealing rounds and steps, we record the final score (this is the best score found over the course of all annealing). The improvement can thus be measured by simply computing the improvement ratio between the final and initial scores. If the final score shows an improvement of greater than the improvement threshold of 1.02, then we save the final score as the initial score, the number of steps per round is doubled, and the entire set of repeats is done again. This process continues until the improvement ratio is less than the threshold.

2.2.3 Fold Tool Quick Reference

Now that we have explained the inner workings of Fold, we provide a Quick Reference explaining how to use the Fold tool. The Fold tool is configured exclusively from the command line and provides many useful options. These options include command line arguments for inputting and outputting both sequences and pairs, preparing data for demonstrations, setting annealing options, specifying which lattice type should be used, and controlling randomization. These command line options are shown in Table 2.6. The reader is referred to the Software Manual in Appendix A for a more in depth explanation of Fold.

2.3 Structure Visualization

In addition to structure manipulation, the Delta toolset includes support for structure visualization. Structure visualization is done using a simple *ball-and-stick* model. Under this model each nucleotide is represented as a ball, with A, C, G, and U bases represented as red, yellow, green, and blue balls respectively. Each nucleotide is connected to the previous and next nucleotide in the sequence using a stick. Finally, each paired base is connected

Table 2.6. FOLD command line options.

Command	Use	Description
`-i <file>`	Required	Reads bases and turns from `file`.
`-i2 <file>`	Required*	Reads base pairs from `file` (*required for reconstruction only). Setting this option will automatically change FOLD to reconstruct mode.
`-o <file>`	Optional	Writes bases and turns to `file`.
`-o2 <file>`	Optional	Writes base pairs to `file`.
`-movie`	Optional	Writes all accepted pull moves to `stdout` (one pull move per line) in the format: `index direction`. This output can then be fed into the SHOW tool to display a visual representation of the folding process.
`-v`	Optional	Writes verbose messages to `stderr`.
`-a [steps][repeats]`	Optional	Performs simulated annealing for the given number of `repeats` and `steps`. By default FOLD will set repeats to 5 and steps to $100n^2$ where n is the number of points in the sequence.
`-d`	Optional	Enables Doubling Steps.
`-t`	Optional	Terminates early. This option is used exclusively with reconstruct and causes the the annealer to terminate as soon as all pairs have been realized. This means the configuration will not be as smooth as may be desired, but fulfills all the pair requirements.
`-s <seed>`	Optional	Initializes the random number generator with `seed`.
`-l2`	Optional	Turns on 2D triangular lattice mode.

to its pair by a thin wire. In Figure 2.7 we can see a simple example of the ball-and-stick model for the sequence $ACGUGCA$. In the model each type of nucleotide is shown, and a pairing between the 3^{rd} and 6^{th} indicies (G and C) is shown.

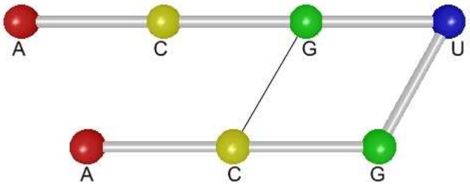


Figure 2.7. Ball-and-stick representation of $ACGUGCA$.

2.3.1 Graphical Representation

In order to visualize a sequence, the visualizer takes an RNA file as input, just as the structure manipulation tool did. Using the same process as the structure manipulation tool, the input sequences are loaded onto the lattice. As discussed previously, each lattice location corresponds to a location in cartesian space, and this corresponding location is the location displayed by the visualizer. To translate the lattice points ($\vec{p} = (x, y, z)$) to cartesian coordinates ($\vec{p'} = (x', y', z')$), we use the following equations:

$$x' = x - \frac{z}{2} \tag{2.12}$$

$$y' = y - \frac{z}{2} \tag{2.13}$$

$$z' = \frac{z}{\sqrt{2}} \tag{2.14}$$

Using these equations we translate each point to cartesian space, and we translate it in a way that is visually pleasing to the user. We here note that the translation specified is not the only possible translation. We use this translation because it provides two distinct advantages. First, it defines a uniform lattice. This means that from any given point, each of the twelve neighbors is exactly one unit distance from the origin point. Second, using this definition allows us to separate the points along the three parallel square planes, with each of these three planes containing exactly four of the origin point's neighbors. This configuration has proved to be much more intuitive than other configurations. In addition to accurately representing the 3D triangular lattice, the visualization tool can be used to represent other lattice types. Other lattice types can be represented by simply changing the axes allowed[2]. For example, a square lattice can be represented by allowing only the $\vec{X}$ and $\vec{Y}$ axes to be used. It should be noted, however, that the skew in the Z axis may lead to a somewhat misleading view of the structure when other lattice types are displayed on the 3D triangular lattice.

2.3.2 Viewing

After a structure has been represented in this simple *ball-and-stick* model, the next step is to display the structure to the user. This is done by creating a `view`. The `view` specifies exactly what the user sees at any given point in time by controlling the perspective from which the structure is viewed. The `view` consists of:

`center` : The X, Y, and Z coordinates of the center of the view. Using these coordinates we translate, or shift, the viewing perspective in order to position the model in the viewscreen.

`rotation` : The vector around which the scene is rotated and the angle of the rotation.

`zoom` : The zoom level of the scene.

The view is initialized to be centered at the center of mass, with no additional rotation. We define the initial zoom as the average of the minimum and maximum zooms. We define the minimum zoom (Z_{min}) as:

$$Z_{min} = 6.0c, \tag{2.15}$$

where c is the circumradius (calculated using the previously determined center of mass). The maximum zoom (the closest the sequence can be to the screen without being clipped)

[2] While the visualization tool can easily be customized to allow other lattice types, the structure manipulation tool requires additional coding to support these lattice types. Specifically, when attempting to use the structural manipulation tool, the user must define the `pull` and `adjacency` functions for any additional lattice type.

is set to a fixed constant.

Having defined the initial `view`, we use OpenGL library to construct the specified scene. To construct the scene, OpenGL uses a display function. The display function first `translates` to the `zoom` defined in the default `view`. The scene is then `rotated` to the correct viewing position (`rotation`). Finally, the scene is `translated` to the appropriate `center` location. Using this simple display function, we can now change the scene by simply changing the current `view`.

2.3.2.1 View Rotation

One of the structure visualizer's most important features is its ability to rotate the scene. To accomplish this efficiently, we used Quaternion Math. Recall that the rotation of any scene consists of the vector $(\vec{X}, \vec{Y}, \vec{Z})$ around which to rotate and the angle of rotation. This rotation can be easily represented as a single array of length four, where the first subscript location is the angle of rotation, and the following three locations represent the vector around which to rotate. When the scene is rotated, we need to add the new rotation to the current rotation. We add the two rotations by first converting the current rotation, and the rotation to be added to the current rotation, to quaternions (also arrays of length four). The method for converting a rotation (r) to a quaternion (q) is shown below:

$$q_0 = \cos(\frac{r_0}{2} \cdot \frac{\pi}{180}) \tag{2.16}$$

$$q_1 = r_1 \cdot \sin(\frac{r_0}{2} \cdot \frac{\pi}{180}) \tag{2.17}$$

$$q_2 = r_2 \cdot \sin(\frac{r_0}{2} \cdot \frac{\pi}{180}) \tag{2.18}$$

$$q_3 = q_3 \cdot \sin(\frac{r_0}{2} \cdot \frac{\pi}{180}) \tag{2.19}$$

In each of these equations, the angle r_0 is assumed to be in degrees.

After the current and new rotations have been converted to quaternions, we can multiply the two to effectively add the new rotation to the current rotation. Multiplying the two quaternions (p, q) creates a new quaternion of the same dimensions. This quaternion

(o) is defined as:

$$o_0 = p_0 \cdot q_0 - p_1 \cdot q_1 - p_2 \cdot q_2 - p_3 \cdot q_3 \tag{2.20}$$

$$o_1 = p_0 \cdot q_1 + p_1 \cdot q_0 + p_2 \cdot q_3 - p_3 \cdot q_2 \tag{2.21}$$

$$o_2 = p_0 \cdot q_2 - p_1 \cdot q_3 + p_2 \cdot q_0 + p_3 \cdot q_1 \tag{2.22}$$

$$o_3 = p_0 \cdot q_3 + p_1 \cdot q_2 - p_2 \cdot q_1 + p_3 \cdot q_0 \tag{2.23}$$

Finally, this new quaternion is converted back to the rotation (r') vector as follows:

$$r'_0 = \arccos(o_0) \cdot \frac{360}{\pi} \tag{2.24}$$

$$r'_1 = \frac{o_1}{\sin(\arccos(q_0))} \tag{2.25}$$

$$r'_2 = \frac{o_2}{\sin(\arccos(q_0))} \tag{2.26}$$

$$r'_3 = \frac{o_3}{\sin(\arccos(q_0))} \tag{2.27}$$

The original rotation (r) can now be replaced with the final rotation (r'), causing the display function to update the scene to the new rotation.

2.3.2.2 View Animation

Having demonstrated how changing the rotation in the `view` causes the scene to update, we can intuitively see that changing the center coordinates, or the zoom, will have the same effect. We now show how these simple changes can be upgraded to support a variety of scene animations. Instead of working with a single view, the current view, we now define the current view as simply the *current* position of a scene in transition from one view (the source view) to another (the destination view). When we are not moving, the view we are moving from and the view we are moving to are both defined to be the current view. Animation can now be defined as simply a source view (V_s), destination view (V_d), and the amount of time spent transitioning between them. The current view (V_c) is then defined as:

$$V_c = V_s \cdot (1.0 - d) + V_d \cdot d \tag{2.28}$$

where d is the percentage of the transition time passed.

Using view animation the user is allowed to change focus from one nucleotide to another by transitioning from the view centered at one nucleotide to the view centered at another nucleotide. The visualizer also supports a *drifting* mode where the view drifts from one nucleotide to another while changing the rotation and zoom. This is accomplished by creating a new view such that the center is the cartesian coordinates of a random nucleotide, and the rotation and zoom are chosen at random.

2.3.3 RNA Folding Movie

Because the visualizer and the structural manipulation tool are built on the same Delta framework, the visualizer can also call the `pull` library function. This function allows the visualizer to make small structural changes to the sequence by selecting a nucleotide to move and the direction in which the nucleotide should be moved. While this is not useful for finding the optimal structure—the visualizer cannot use scoring functions—this ability can either allow users to manipulate the structure as they see fit or allow the visualizer to automatically execute a series of pull moves. The latter use gives users the impression of watching a simple movie. For this reason we refer to this mode as *Movie Mode.* When the visualizer is in movie mode, the visualizer creates a movie by performing the following steps:

1. Read a pull move from `stdin`.

2. Test the move to make sure it is valid.

3. If the move is valid, complete the pull move.

4. Update the nucleotides cartesian coordinates by translating the updated lattice coordinates provided by the Delta library.

5. Display the new scene.

6. Wait t ms and then repeat.

2.3.4 Graphical Performance Tuning

In order to provide a rich user experience, the Show tool provides a great deal of animation. Movie mode and Drift mode can be used to provide interesting demonstrations; pulsing can be used to make it easy to see, at a glance, which nucleotide is selected; and drifting from one nucleotide to another provides a sense of perspective when observing or manipulating a sequence. In addition to these features, users interact with the displayed sequence by rotating and zooming. All of these features require some sort of animation, and all of these features may possibly interfere with each other. To understand and solve this problem, we first need to understand the structure used to keep track of the sequence orientation at any given time. For obvious reasons, this new structure is the `view`.

Each view consists of the current shift, rotation, and zoom. The shift can be understood as how far left or right, and how far towards the top or bottom of the window the image is. Rotation and zoom are intuitive and require no further explanation. We now define drifting as simply transitioning between two views; thus; *Drifting Mode* is simply drifting between random views. Zooming in and out, rotating, and shifting between bases are then just special subsets of drifting, where only the corresponding component of the view structure is manipulated (see section 2.3.2).

Given a current view, we can now create an animation using the new target view and a timer. Animation is accomplished by simply displaying the current view and then the target view, separated by some number of transition views. The key to animating the view is determining the number of transition views to use. If we use too few views, the animation is choppy and distracting. If we use too many, the animation is smooth, but may be too slow to be useful.

Rather than trying to solve for the optimal number of steps, we have chosen to dynamically calculate the number of views. This solution is more useful because we not only find a middle ground allowing for both as many transition frames as possible and an animation taking a desired amount of time, but we can also account for the vast difference in computer speeds. To dynamically scale the frame rate, we simply use a timer and specify a target duration. When the display function is called, we use the amount of time that has passed, as a ratio of the target duration, to decide how far the current view should be along the animation. The relationships between the current view (V_c), the starting (source) view (V_s), and the target view (V_t) are described in the following interpolate equation:

$$V_c = \begin{cases} V_s \cdot (1.0 - \triangle T) + V_t \cdot \triangle T & \triangle T < 1 \\ V_t & \triangle T \geq 1 \end{cases} \tag{2.29}$$

$$\triangle T = \frac{T_c - T_i}{D} \tag{2.30}$$

where T_c is the current time, T_i is the initial time, and D is the target duration.

We see that drifting can be used to describe shifting, drifting mode, zoom, and, rotation. To guarantee optimal performance of the visualization tool, we must now guarantee that additional animations do not interfere with each other or with the drifting process. We now explain how pulsing, movie mode, and user interactions are handled.

When using the structure visualization tool, it is useful to cause the currently selected nucleotide to pulse. Using this visual cue provides a point of reference for users, and makes navigating through the sequence much easier. This animation is accomplished using a variation of the interpolate equation (Equation (2.29)). In the original version of interpolate, the equation measures time from a fixed point, and it scales from one point to another. With pulse we need to continuously cycle from small to large and back again. To incorporate this behavior, we use the sin function and the current time (in ms) to scale the radius of the current nucleotide. In this case we use the current time because we do not want to measure time from any specific point; we just want to vary the radius over time. We now have $\triangle T = \frac{T_{ms}}{1000000.0} \cdot \pi \cdot 2.0$. We then simply use equation (2.29), substituting a target and initial radius for the target and initial views. In this way we easily update the radius of the current nucleotide each time the display function is called. Calling the display function in quick succession creates the pulsing effect. To call the display function in quick succession, we create a variable, `need_redisplay` indicating a call to the display function should be scheduled. Then, at the end of the display function, the value of `need_redisplay` is checked.

If need_redisplay is set to true, which it will be, the schedule_redisplay function is called, and a new variable, redisplay_scheduled, is set to false. As the name implies, this new variable indicates whether a display function has already been scheduled, and is functionally the same as indicating a display is in progress. Using redisplay_scheduled, the schedule_redisplay function fills a simple purpose: when called, it checks to see if a call to the display function has already been scheduled. If it has, it does nothing; if not, it marks the display function as scheduled and then calls display. Scheduling the display function in this way causes the display function to be called every few milliseconds and guarantees that only one display function is scheduled at a time.

Scheduling a display every few milliseconds seems safe enough, and indeed it may seem unnecessary to use the need_redisplay variable, or to keep track of whether a display has been scheduled or not (since up to this point only one display could be scheduled). To understand the necessity of need_redisplay and schedule_redisplay, we look at how drifting interacts with pulsing. If we run a drifting function (whether "drifting mode" or one of the simpler drift methods discussed previously), this animation is accomplished by repeatedly calling the display function. As the values in the current view are changed, the display function slowly shifts the user perspective. If these interactions are handled separately, we could expect the scheduled displays of the drifting function to overlap and collide with the scheduled displays of the pulsing function. It is for the purpose of preventing these collision that we use need_redisplay and schedule_redisplay to keep track of whether or not a display has been scheduled. When a call to the display function is completed, whether initiated to cause either the drift function or the pulsing function, we check to see if a redisplay is needed, and then call the schedule_redisplay function. Using this unified path allows us to handle drifting and pulsing together and prevent undesired effects from the schedules colliding.

The second interaction that must be handled is the "movie mode" interaction. This mode causes additional scheduling problems because the movie speed can be adjusted using keyboard shortcuts. We can imagine the movie speed set at a very slow rate, while pulsing and/or drifting occur at a very fast rate. To solve this problem we introduce a sentry variable. The sentry variable is used to guarantee a move is taken only at the desired intervals. We discuss how this is accomplished. When the movie first starts, this sentry variable, `movie_busy`, is set to false. Setting `movie_busy` to false indicates that no movie step is in progress, and that it is safe to schedule a movie step. At the end of each call to the display function, if "movie mode" has been enabled, is not paused, and `movie_busy` is set to false, then the `movie_busy` is set to true. After setting `movie_busy` to true, `need_redisplay` is set to true and the move is taken. The move is used to update the sequence, which in turn changes what is displayed (the display function is immediately called because `need_redisplay` was set to true). After re-displaying the scene, the sentry variable prevents any additional movies from being taken until the current move is completed. At the same time the `movie_busy` is set to true, a call to another function, `movie_advance`, is scheduled for some time in the future, and the `need_redisplay` variable is set to true. The scheduling of the `movie_advance` function depends on a user controlled time delay variable. Using keyboard shortcuts, the user can increase or decrease the delay between each consecutive movie step. After the specified amount of time has passed, `movie_advance` is called. This function signals the movie step in progress has completed by setting `movie_busy` to false, and scheduling the display function. This process continues until the movie completes.

The final interaction that must be handled is user interaction. While a screen is displayed, the user may wish to rotate, zoom, or shift the view. This interaction is also easily solved using the `schedule_redisplay` function. Instead of calling the display function immediately after the user attempts to rotate, zoom, or shift, and then continuing to schedule a refresh as soon as the current refresh completes—a process that may waste all our CPU cycles doing nothing more than displaying and re-displaying what we already have— each of the rotate, zoom, and shift functions immediately call the `schedule_redisplay` function. Recall that at the end of each call to the display function, the `redisplay_scheduled` variable is set to false. This means that if `redisplay_scheduled` is set to true, the display function is already in progress, and calling display again would not increase the speed with which the screen is updated, and thus would not increase the responsiveness of the program. Then, when the current display function is completed, the `need_redisplay` variable is checked to see if pulsing, drifting, or movie mode needs the display to be updated. If so, `schedule_redisplay` is called again; otherwise, the program will wait for a rotate, zoom, shift, or other user interaction for display process to be initiated again.

2.3.5 Show Tool Quick Reference

The Show tool is intuitive to use, but it is also more complex than the Fold tool. The Show function includes command line options defining the sequence to be displayed, whether the optional "movie mode" is used, which base pairings to display, and where to save changes to the initial configuration. These command line options are shown in Table 2.7. Once configured from the command line and executed, Show makes exploring and manipulating the sequence easy by providing both mouse and keyboard support. Left-clicking and dragging with the mouse rotates the model around the $\vec{X}$ and $\vec{Y}$ axes. Holding shift while doing this switches to rotating around the $\vec{Z}$ axis and adjusting the zoom. Finally, right-clicking shows the menu. From the menu, you may select a command directly; it is also a useful reference to learn the keyboard commands (summarized in Table 2.8).

Table 2.7. Show command line options.

Command	Use	Description
`-i <file>`	Required	Reads bases and turns from `file`.
`-i2 <file>`	Optional	Reads base pairs from `file`. If specified only the pairs defined in `file` will be shown paired.
`-o <file>`	Optional	Specifies the `file` to which the bases and turns will be saved when the user selects to save the currently displayed configuration.
`-movie`	Optional	Reads pull moves from `stdin` in the format: `index direction`

Table 2.8. Show keyboard shortcuts.

Key	Description
‘w’	Write the current configuration to file specified at the command line.
<Space>	Adjust the zoom and perspective so that the entire sequence can be viewed.
‘i’	Zoom in.
‘o’	Zoom out.
‘a’	Toggle animation on and off. By default this option is turned on.
‘d’	Toggle drift mode. When this option is turned on, the camera will zoom, rotate, and pan randomly. This option is turned off by default, but it automatically turns on when movie mode is turned on.
‘b’	Toggle RNA bonds on and off. By default this option is turned on.
‘v’	Toggle vector arrow on and off. By default this option is turned off, but this option is automatically turned on whenever the <Up>/<Down> keys are pressed.
‘m’	Pause the movie. This option only has effect when Movie Mode has been initiated from the command line.
‘f’	Make the movie faster.
‘s’	Make the movie slower.
‘p’	Move the selected base in the direction indicated by the axis arrow.
‘r’	Redo the last undone pull move.
‘u’	Undo the last pull move.
<Left>/<Right>	Navigate through the current sequence.
<Up>/<Down>	Switch which axis the axis arrow is showing. This option automatically sets the axis arrow to on.
<Esc>	Quit.

CHAPTER 3
RNA SECONDARY STRUCTURE PREDICTION

In Chapter 2 we discussed how the 3D triangular lattice can be used to simulate an RNA sequence's tertiary structure interactions. This method, by itself, has been shown to be very effective. In [2, 3], the authors show that without any additional processing, this method is comparable in sensitivity (see Equation (3.1)) to some of the commonly used programs. In other words, the triangular lattice is able to predict a comparable number of the true base pairings. Unfortunately, using the 3D triangular lattice for prediction is not sufficient for general purpose secondary structure prediction. In this chapter we present the problem with using the 3D triangular lattice, alone, for prediction, as well as the steps necessary to solve this problem.

3.1 3D Triangular Lattice Prediction Problem

To understand the limitation and underlying problem with using only the 3D triangular lattice to predict a sequence's secondary structure, it is important to understand how the secondary structure is predicted from the lattice. After the structure manipulation tool has been completed, and a tertiary structure has been predicted, we must translate this tertiary structure into a secondary structure. This translation is done by first identifying all adjacencies. Each adjacency (i, j) is then considered a valid base pairing if:

1. The pairs (i, j) are separated by a minimum of three bases.

2. The pair (i, j) is a Watson-Crick pair, and either $(i - 1, j + 1)$ or $(i + 1, j - 1)$ is also a Watson-Crick pair.

3. The pair (i, j) is a Wobble Pair, and both $(i-1, j+1)$ and $(i+1, j-1)$ are Watson-Crick pairs.

If an adjacency is a valid pair, the structural manipulation tool writes the base pairing to an output file. It is written with each line representing one pairing and the indices of the two pairs separated by a single space. As we can see, instead of predicting a secondary structure, the lattice really just predicts a filtered set of adjacencies. In any moderately complex RNA folding, many of the bases are adjacent to multiple other pairs. Thus, the 3D triangular lattice, alone, gives us little help in separating the true pairings from the simple adjacencies.

To demonstrate why considering adjacencies to be pairings can be a problem, we first introduce three quality measures: sensitivity, specificity, and accuracy. The first measure, sensitivity, measures how many of the correct pairs are predicted. The next measure, specificity, measures how reliable the predictions are. The final measure, accuracy, is the most informative; it can be thought of as a summary statistic, combining the sensitivity and specificity measures. For purposes of comparison, we define the sensitivity, specificity, and accuracy as follows:

$$\text{Sensitivity} = \frac{\text{tp}}{\text{tp} + \text{fn}} \tag{3.1}$$

$$\text{Specificity} = \frac{\text{tp}}{\text{tp} + \text{fp}} \tag{3.2}$$

$$\text{Accuracy} = \frac{\text{tp}}{\text{tp} + \text{fn} + \text{fp}} \tag{3.3}$$

where tp is the number of true positives, fn is the number of false negatives, and fp is the number of false positives.

Returning to the quality of the 3D triangular lattice predictions, we observe that this method is sensitive: it predicts many or most of the correct pairs. The problem is the method is neither specific nor accurate. To understand why the specificity and accuracy of an algorithm are important, consider the following example. Imagine an algorithm which always predicts every nucleotide will pair with every other nucleotide in a sequence. This algorithm would predict all true pairings 100% of the time, but wouldn't be worth anything to anyone in terms of understanding the structure. It is for this reason that we use all three statistics—sensitivity, specificity, and accuracy—for evaluating our solution.

3.2 Helicies and 2-interval Graph

To create a more effective solution, and to maximize the sensitivity, specificity, and accuracy of the solution, we use the 3D triangular lattice as only a portion of the solution. The next step of our solution requires a new data representation. When the structural manipulation tool completes, it returns a simple list of base pairings. We translate[1] these pairs into helices. Each helix consists of a starting index (i), ending index(j), and the number of consecutive pairs contained in the helix (the length L). For example, the base pairings:

```
1 10
2 9
1 15
2 14
3 13
4 12
5 11
5 12
6 11
16 20
```

can be more succinctly represented by the helicies:

```
1 10 2
1 15 5
5 12 2
16 20 1
```

In this example, we see that the pairs $(1, 15)$, $(2, 14)$, $(3, 13)$, $(4, 12)$, and $(5, 11)$ are grouped together to form the second helix $(1, 15, 5)$. The helix representation $(1, 15, 5)$ means the pair $(1, 15)$ is the first pair in the helix, and each pair $(1 + k, 15 - k)$ is also a pair, where k is all integers in the range $0 < k < L$. As we can see, the helix representation contains the same information as the pair representation, but the helix representation is much more compact.

Translating the pairs' output from the structural manipulation tool to the helices is a relatively simple process. Each pair is converted to a helix using a simple `pair` data structure. This structure keeps track of the starting index (i) and the ending index (j); it also records whether or not a pair has been used. We then follow this simple algorithm (Bp2hx) to make the conversion:

[1] Appendix E contains source code for all our format manipulations

1. Read in each base pair. Each pair has the format:

   $i\ j$

   where i is the lower index in the pairing, and j is the higher index in the pairing.

2. For each pair read, create a pair structure and insert it into the collection of pairs. When the insert is done, we use an insertion sort to order the pairs first by i and then by j. At the time of each insertion, we set the variable that indicates whether or not the pair has been used to false, indicating it has not been used.

3. Iterate over each pair in the sorted collection. For each pair:

   (a) Test if the pair has been used. If it has, do nothing, and move to the next pair in the collection.

   (b) Mark the pair as used, and set the length to 1.

   (c) Using the sorted nature of the collection, look for the k^{th} adjacent pair $(i+k, j-k)$, where k is initially set to 1. We denote this target pair as t. For convenience we denote the indices of t as simply t_i and t_j, corresponding to the lower and upper indecies respectively. We also denote the pair under evaulation as c, with its pair indecies corresponding to c_i and c_j. We can now efficiently search the collection for c using, in order, the following properties of the sorted list:

      i. If a pair is marked as used, we can immediately skip to the next pair.

      ii. If c_i is less than $i + 1$, we can immediately skip to the next pair.

      iii. If c_i is greater than t_i, we can stop looking for t.

      iv. If c_j is less than $j - 1$, we skip to the next pair.

      v. If c_j is greater than $j - 1$, we stop looking for t.

      vi. If none of the previous conditions is true, then we have found t, and we can mark t as used, and increment the length of our original pair. We can then increment k by one and continue at step $3(c)$.

(d) Once we have failed to find an adjacent neighbor (t), print out the helix we have found. This helix is defined to be the indices for the initial pair and the length.

Each helix can now be used to create a 2-interval graph. A 2-interval graph is a simple graph representation of helix interactions first proposed by Vialette [21]. An example 2-interval graph is shown in Figure 3.1. In this figure there are 8 helices represented. Each helix has been decomposed into lower and upper line segments. The lower segment represents the lower half of the helix, the indices ranging from i to $i+L-1$. The upper segment is similar, connecting the indices ranging from j to $j-L+1$. These two line segments are then connected with an arc. The arcs help us see the relationship between each helix, showing pseudoknotting when lines cross, or indicating a nested helix when an arch is completely contained within another arch. While Figure 3.1 models a set of independent helices, this is not necessarily the case with the helices converted from the structural manipulation tool. Because each base on the lattice may participate in multiple pairs, the helices converted from the structural manipulation tool may overlap. The 2-interval graph constructed from these overlapping helices are confusing, allowing us to easily observing the problem with using the structural prediction method alone.

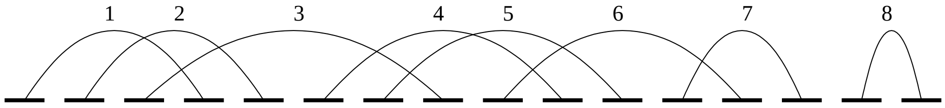


Figure 3.1. 2-interval graph.

3.3 Maximum Weighted Independent Set

Using the helix representation of the predicted pairs and the 2-interval concept we have introduced, we can now use the 2-interval graph to construct the maximum weighted independent set of helices. We compute this independent set using our independent set program, called simply Is for short. Is computes the independent set using the RNA sequence, a set of base pairs, and a 2-interval graph. The algorithm is as follows:

1. Read the RNA sequence from an input file.

2. Read each helix from stdin into a simple helix data structure. Similar to the pair data structure discussed previously, the helix structure contains the starting index i, the ending index j, and the length. Additionally, the helix structure contains the color, validity, and energy of the helix. As part of reading the helix file in, each of the helix structure variables are initialized. The process is as follows:

   (a) The helix is tested for validity. This testing includes checking to make sure the starting and ending indices fall within the valid range of indices, verifying the minimum distance between paired bases has been satisfied and verifying the helix has a negative folding energy. The helix's folding energy is determined by using the MFOLD energies previously presented in Table 2.4. Valid helicies are added to the helix collection; invalid helices are filtered out.

   (b) IS next attempts to extend the inside of the helix. Given that each helix is defined by the indices of the outermost paired nucleotides (i, j) and the length (L), the innermost paired nucleotides must then be $i + L - 1, j - L - 1$. In attempting to extend the inside of the helix, we first test to see if adding the pair $(i + L, j - L)$ increases the score of the helix while still maintaining a valid helix. If this is true, we increase the length of the sequence by simply incrementing L. This extending process continues until it either results in a lower scoring helix or the resulting helix is invalid.

   (c) IS next attempts to extend the outside of the helix. This is done in exactly the same manner as in step (b), except that instead of just incrementing L, we must increment j and decrement i at the same time as we increment k.

   (d) The helix is verified to be at least as long as the minimum length, defined as 3 in this program.

3. Begin the process of creating the independent set of helices as follows:

   (a) Find the candidate helix (U) with the highest score (the lowest energy). A candidate is any unused helix.

(b) Compare each candidate helix to all previously accepted helices (V) to see if the helices cross. A helix is said to cross if $U_i < V_i < U_j < V_j$ or if $V_i < U_i < V_j < U_j$. We can understand this by remembering the 2-interval representation previously introduced. Recall that the 2-interval representation consists of the first half of the helix (indices ($[i : i + L - 1]$), connected by an arc to the second half of the helix (indices ($[j - L - 1 : j]$). If the arcs of two helices cross each other, then the helices are crossing. Figure 3.2 shows a simple example of a crossing pair. In this example U does not cross $V2$, but U is crossing $V1$ ($V1_i < U_i$ and $V1_j < U_j$).

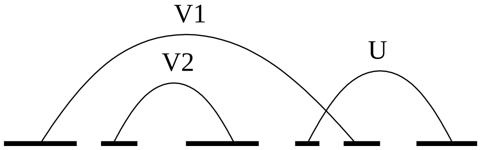


Figure 3.2. 2-interval graph with crossing: U crosses $V1$.

For reasons that will be explained in step (d), we can assume helices u and v are disjoint, or in other words, helices u and v do not share any nucleotide indices. It is important here to look ahead a little further in the algorithm. When a candidate helix is accepted as part of the independent set of helices, it is assigned a color. This color is one of two possible options, corresponding to upper or lower. The Is algorithm uses these two colors to define whether the arc is drawn above the intervals or below the intervals. If the arc is above the intervals, we say that it exists on page one; if the arc is below, it exists on page two. Figure 3.3 demonstrates this concept. In the figure we show four helices. If the arcs could only be drawn above the helix sections (as they would be in a single page solution), then helices 1 and 2 would be conflicting as would $(2, 3)$ and $(3, 4)$. In a two page solution we are allowed to use both the top and the bottom sections. This means that while there are crossing helices in Figure 3.3, there are no conflicting helices in the figure.

Returning to the present part of the algorithm, if our candidate helix (U)

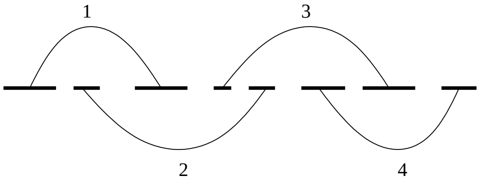


Figure 3.3. 2-interval graph on two-pages with no conflicting helices.

crosses a helix, we mark the color of the crossed helix as invalid for (U). This marking is equivalent to specifying that the candidate helix cannot be on the same page as the helix with which it conflicted.

(c) If the candidate helix U conflicts on both colors, discard the helix and mark it as invalid. If the candidate helix is valid on the first color, then assign it the first color; otherwise, assign it the second color. If the helix is valid, add the helix to the collection.

(d) Compare the helix to all remaining unused helices. This comparison checks for overlap between the newly accepted helix and the remaining potential candidates. If there is overlap, the overlapping pairings are removed from the potential candidates, and the helices' indices, length, and score are updated. In some cases this results in removing entire helices from the set of potential candidates. It is because of this step, which removes all overlapping pairs from the remaining candidates, that we can assume the candidate helices are independent of the selected helices in step (b).

(e) Repeat steps (a) through (d) until there are no remaining candidate helices.

4. Print to stdout the helices in the collection; these are the helices that make up the independent set.

3.4 Final Prediction Methodology

Having developed a method for determining the maxium weighted independent set of helices, we can use this method, in conjunction with the structural prediction tool, to generate more accurate secondary structure predictions. To make our final secondary structure predictions, we first use FOLD to predict a tertiary structure. The predicted bases are then translated from base pair format to helices. These helices are read into the 2-interval tool; the maximum weighted independent set of helices is calculated; and the resulting helices are saved. As a final step, the helices of the independent set are translated back into base pairs. These base pairs are then read back into FOLD. Using FOLD's Reconstruction Mode, we can realize the tertiary structure of our final predictions. This final step does not increase the accuracy of the secondary structure prediction, but is useful for generating more visually pleasing input into the SHOW tool. The final tertiary structure is much cleaner and easier to understand than the original tertiary structure predicted by the structural manipulation tool alone. This process is summarized in Figure 3.4.

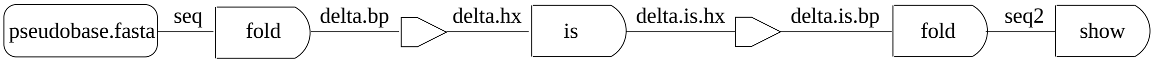


Figure 3.4. Overview of the RNA project prediction process.

It may not be immediately obvious how predicting the maximum weighted independent set of helices improves a prediction's sensitivity, specificity, and accuracy. To demonstrate the necessity of this step, we show how this step improves the prediction of sequence PKB00003. After using FOLD to predict a near-optimal tertiary structure, we save the predicted base pairs to file. We can then convert the base pairs to helices using the BP2HX program previously described. At this stage we use a simple script, `hx2i.awk`, to create a text art representation of the structure. Each line of the structure's text art representation represents a single helix. Using this format we can can quickly get an idea of the prediction's quality. Shown in 3.5 is the PKB00003 sequence, along with the initial prediction. As we can see in this simple diagram, the FOLD-predicted structure includes five helices. The helices range in length from 2 to 5, and contain several overlapping helices. The true structure of PKB00003 consists of two helices: $(2, 20, 6)$ and $(11, 36, 4)$. Grading the predictions at this point we see that FOLD predicted all but one of the true pairs (tp = 9, fn = 1). Using Equation (3.1), we see that FOLD receives a sensitivity score $\frac{9}{10} = 0.9$, which is relatively high. Unfortunately, the specificity and accuracy are not as good. FOLD predicted a total

```
AGGGGCUCAAGGGAGGCCCCAGAAACAAACUUUCCCG
 -----         -----
      --             --
          ----                  ----
           -----             -----
                          --   --
```

Figure 3.5. The initial prediction of PKB00003 sequence, using Fold alone.

of 18 pairings, resulting in 9 false positives. Using Equations (3.2) and (3.3), respectively, to determine specificity and accuracy, we see Fold has a specificity of $\frac{9}{9+9} = 0.5$ and an accuracy of $\frac{9}{9+1+9} = 0.47$.

Now that we have computed the quality measures obtained from Fold alone, contrast these results with the improved maximum weighted set prediction (shown in 3.6). A quick glance at Figure 3.6 is all that is required to see this step's value in the prediction process. We can see that when Fold and Is are used together, the prediction not only includes no overlapping helices, but correctly predicts both true helices and makes this prediction without making any false predictions. In this case the sensitivity, specificity, and accuracy are all 100%. In other words, using Fold and Is together resulted in perfectly predicting the secondary structure of PKB00003.

```
AGGGGCUCAAGGGAGGCCCCAGAAACAAACUUUCCCG
 ------       ------
          ----                  ----
```

Figure 3.6. The prediction of PKB00003 sequence, using Fold and Is together.

CHAPTER 4
EXPERIMENTS

4.1 Data Acquisition

In order to measure our prediction method's effectiveness, on May 25, 2008, we downloaded 275 RNA sequences from the Pseudobase Database [22]. This database is ideal for testing our prediction process, as the database is dedicated to pseudoknotted sequences. As of the writing of this paper, no method existed for downloading all the sequences together. Rather than download the sequences individually, we designed a script to automatically download, filter, and save the data. This script is called PBDB, deriving its name from its function, to translate the **P**seudo**b**ase RNA sequences to sequences in our **d**ata**b**ase file.

PBDB is a simple algorithm designed to download each individual RNA file, stored as HTML, and translate it into one simple FASTA style file called pseudobase.fasta. The algorithm follows these simple steps:

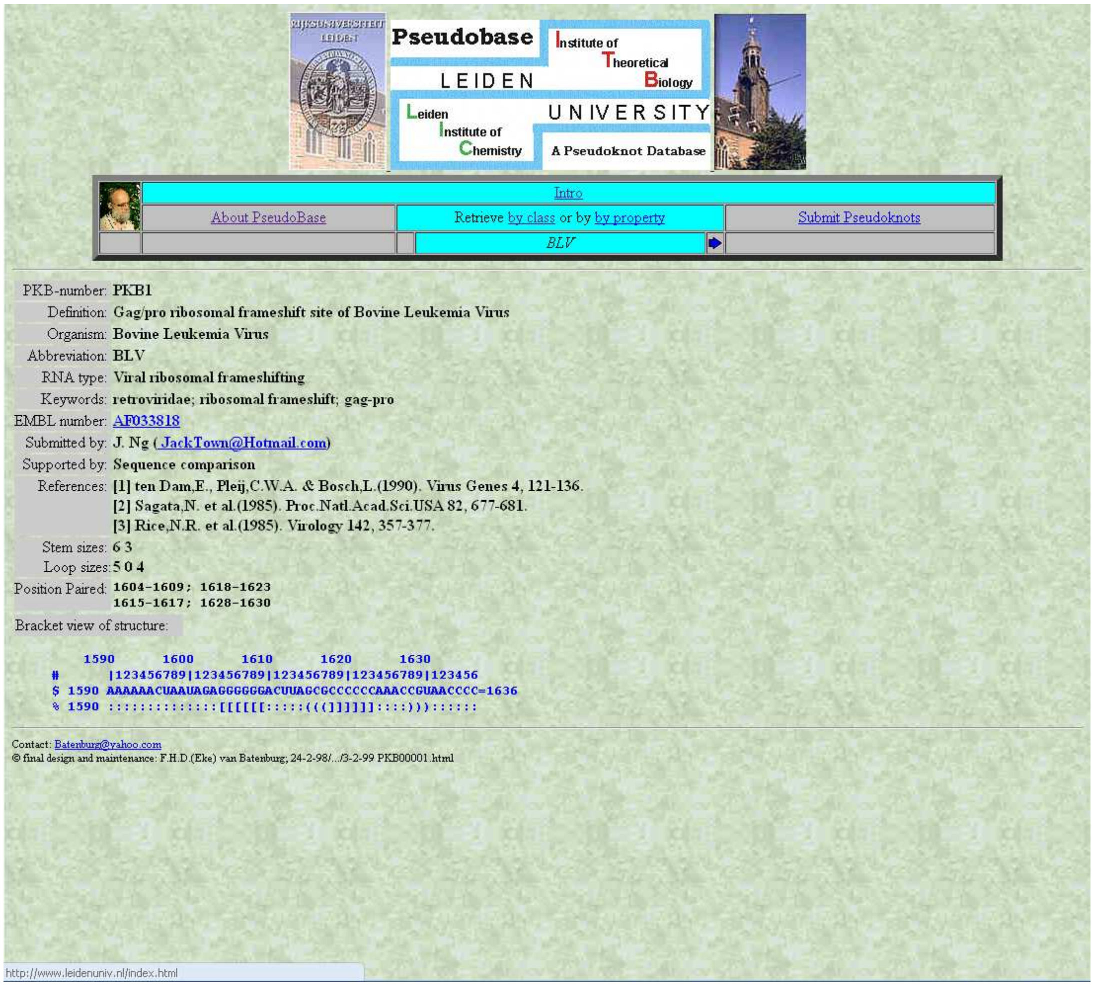
1. Download the main Pseudobase page. This page is important because it contains links to all the Pseudobase sequences.

2. Parse the main page and identify all RNA sequence links. This identification process can be accomplished using pattern matching. We know that all RNA sequences are stored on a page with the format `PKB`, followed by a series of numbers and ending with `.HTML`. To search for this pattern, we use the regular expression "PKB[0-9]+.HTML".

3. Use `wget` to download all the individual files. An example of a Pseudobase sequence page is shown in Figure 4.1.

4. For each download html file:

   (a) Separate out the nucleotide and secondary structure information. Each html file contains a block of code defining the RNA nucleotide sequence, the index into its parent sequence, and a series of colons, brackets, braces, and parentheses defining the known secondary structure (we refer to this method of specifying



RIJKSUNIVERSITEIT LEIDEN
Pseudobase
Institute of Theoretical Biology
LEIDEN
Leiden Institute of Chemistry
UNIVERSITY
A Pseudoknot Database
Intro
About PseudoBase
Retrieve by class or by by property
Submit Pseudoknots
BLV
http://www.leidenuniv.nl/index.html

PKB-number: PKB1

Definition: Gag/pro ribosomal frameshift site of Bovine Leukemia Virus

Organism: Bovine Leukemia Virus

Abbreviation: BLV

RNA type: Viral ribosomal frameshifting

Keywords: retroviridae; ribosomal frameshift; gag-pro

EMBL number: AF033818

Submitted by: J. Ng (JackTown@Hotmail.com)

Supported by: Sequence comparison

References: [1] ten Dam,E., Pleij,C.W.A. & Bosch,L.(1990). Virus Genes 4, 121-136.
[2] Sagata,N. et al.(1985). Proc.Natl.Acad.Sci.USA 82, 677-681.
[3] Rice,N.R. et al.(1985). Virology 142, 357-377.

Stem sizes: 6 3

Loop sizes: 5 0 4

Position Paired: 1604-1609; 1618-1623
1615-1617; 1628-1630

Bracket view of structure:

```
         1590      1600      1610      1620      1630
#        |123456789|123456789|123456789|123456789|123456
$ 1590 AAAAAACUAAUAGAGGGGGGACUUAGCGCCCCCCAAACCGUAACCCC=1636
% 1590 ::::::::::::::[[[[[[:::::(((]]]]]]::::)))::::::
```

Contact: Batenburg@yahoo.com

© final design and maintenance: F.H.D.(Eke) van Batenburg; 24-2-98/.../3-2-99 PKB00001.html

Figure 4.1. Screenshot of Pseudobase sequence website.

the secondary structure as the dot-bracket method and a file consisting of this information as a dot-bracket file). An example of this section is shown below:

```
<pre class=show>
         1590      1600      1610      1620      1630
     #      |123456789|123456789|123456789|123456789|123456
     $ 1590 AAAAAACUAAUAGAGGGGGGACUUAGCGCCCCCCAAACCGUAACCCC=1636
     % 1590 ::::::::::::::[[[[[[:::::(((]]]]]]::::)))::::::
</pre>
```

Using `awk` to perform additional pattern matching, separate out the nucleotide sequence and the structure information.

(b) Create a more favorable test environment by removing extra data with an `awk` script. Extra data exists because, while the Pseudobase database of Pseudoknotted structures contains a significant number of sequences, not all of the structures are completely known. Sometimes sequences contain a significant amount of unknown or unpaired structures. Sometimes the nucleotide bases themselves are not known.

(c) Save the resulting data in a temporary file.

5. After each html file has been processed, combine the individual temporary files together in a fasta format. Our adaptation of this format specifies that each record begin with a '>' followed by the sequence name or id. The next line of the record should be the nucleotide sequence. The third, and last, line of the record is the dot-bracket representation of the secondary structure. To illustrate this format, we present a small section of our resulting pseudobase file:

```
>PKB00081
GCGAUUUCUGACCGCUUUUUUGUCAG
(((::::[[[[[)))::::::]]]]]
>PKB00082
UAAAGUUUGUGUUUCUAAAACACAC
:(((:::[[[[)))::::::]]]]:
>PKB00083
```

```
ACGUGGUACGUACGAUAACGUACA
:(((:[[[[[[))):::]]]]]]:
```

4.2 HotKnot

To determine our method's effectiveness, we compared our results with those of HotKnot. HotKnot was a natural choice for comparison because it has also been shown to be quite effective in predicting pseudoknotted structures. In [1], Ren et al. tested HotKnot using sequences from PseudoBase and demonstrated their program to be comparable to or better than several other RNA prediction programs including: Rivas and Eddy's Pseudoknots algorithm [23], NUPACK [24], the iterative loop matching algorithm by Ruan et al. [25], STAR [26], and pknotsRG-mfe [27]. HotKnot has been demonstrated to be a highly effective program and a good program from which to gauge the success of our own approach.

4.3 Experiment Setup

The experiment itself consists of two parts. The first part is designed to evaluate the predictive accuracy of DeltaIS. The second part evaluates the ability of DeltaIS to reconstruct a previously determined secondary structure in 3D.

4.3.1 Prediction Experiment

Evaluating the prediction process of DeltaIS is accomplished in three steps: evaluating the sensitivity, selectivity, and accuracy of HotKnot, evaluating the same statistics for DeltaIS, and finally comparing the two results.

To determine the sensitivity, selectivity, and accuracy of HotKnot, each RNA sequence is fed into HotKnot and the resulting bpseq files are saved. The individual bpseq files are then translated into base pair files using awk. Each of the initial sequence files has a known structure, saved in dot-bracket format. These dot-bracket files are then translated to base pair file formats using a simple utility program (Db2bp). The base pair file produced for each sequence by HotKnot are then compared to the known structure using the utility program Ssa. Ssa compares the predicted base pairs with the true base pairs and computes each sequence's sensitivity, selectivity, and accuracy. The individual statistics, along with the counts of true positivies, true negatives, false positives, and false negatives are returned.

Our prediction method's sensitivity, selectivity, and accuracy were determined in a similar way. Each sequence was fed into the FOLD program and initialized to a default configuration, in this case a stem-loop configuration. FOLD then performed $100 \cdot n^2$ simulated annealing steps, with the Doubling Steps mode turned on. After completing FOLD's predictions, the resulting base pair files were converted to helix files and then fed into the independent set program as discussed in Chapter 3, and summarized in Figure 3.4. After the process completed, the base pair files from DELTAIS are compared to the true structure base pair files, again using SSA.

After determining the accuracies of both DELTAIS and HOTKNOT, we compared the two methods. Rather than average the resulting statistics, a final utility program, STATS, determined the results. STATS reads SSA's output for each method's resulting sequences. This program allows us to compare the statistics while weighting the results based on the actual number of bases predicted. SSA is also capable of handling the results of several test iterations, allowing us to provide the average statistics over several runs. This test's process is depicted in Figure 4.2, which uses a simple flow-chart diagram to illustrate the entire experiment process. Appendix F contains the source code used for evaluating the effectiveness of HOTKNOT and DELTAIS, and can be used to evaluate the relative effectiveness of other algorithms against DELTAIS.

To provide the best test environment, and to generate a significant number of test runs, we performed this experiment on several machines. We list each of the machines, along with a basic description of each machine's configuration.

1. Dell XPS M1710 Laptop

   (a) Intel(R) Core(TM)2 CPU with a T7400 Processor clocked at 2.16 GHz, and 2.00 GB of RAM

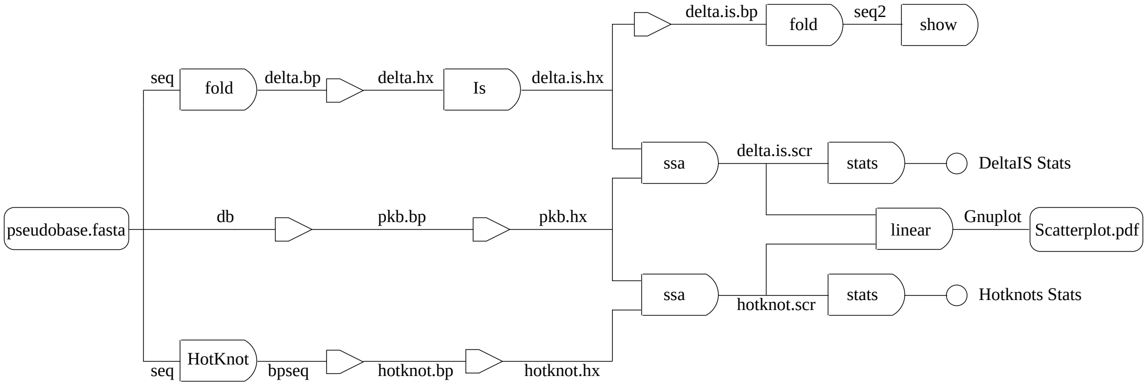


Figure 4.2. Beginning-to-end diagram of experiment.

   (b) Windows XP Professional Service Pack 3, Cygwin Version 2.573.2.3, and GCC Version 3.4.4

2. Apple iMac

   (a) 2.0 GHz Power PC G5 Processor with 2.0 GB DDR SDRAM RAM

   (b) MAC OS X Version 10.4.11 and GCC Version 4.0.0

3. Dell OptiPlex GX620 Desktop

   (a) Pentium(R) D CPU 2.8 GHz with 2.00 GB RAM

   (b) Microsoft XP Service Pack 3, Cygwin Version 2.573.2.3, and GCC Version 3.4.4

4. Dell OptiPlex GX620 Desktop

   (a) Pentium(R) D CPU 3.0 GHz with 2.00 GB RAM

   (b) Microsoft XP Service Pack 3, Cygwin Version 2.573.2.3, and GCC Version 3.4.4

4.3.2 Reconstruction Experiment

To evaluate how accurately DeltaIS can reproduce specified structures, we again use the 252 PseudoBase sequences. This experiment is not used to compare the performance of DeltaIS to the performance of another algorithm. Rather, this experiment is used to determine if DeltaIS can be used as a reconstruction tool, and, if it can, how accurately it can reconstruct pre-specified sequences. To evaluate DeltaIS's ability to reconstruct, the dot-bracket representation of each sequence in the PseudoBase database is translated into base pair format using the DB2BP utility. These base pairs, along with the sequence itself, are read into FOLD. Each sequence is run until one of two conditions is met: the specified number of annealing steps and rounds is complete, or the structure has been predicted perfectly. Once FOLD has completed the annealing process, the resulting prediction is graded using the SSA utility.

4.4 Results and Analysis

4.4.1 Predictive Results

After completing 66 runs of all 252 sequences (each run taking 20-30 hours), we found the predictive portion of our method had mixed results when compared with HOTKNOT. For all 252 sequences, DELTAIS had better sensitivity (79.1% compared with 71.73%) and accuracy (64.25% compared with 59.93%), while HOTKNOT had better selectivity (78.47% compared with 77.37%). These final results are presented in Table 4.1.

Table 4.1. Final experiment results with standard deviations.

Method	Sensitivity	Selectivity	Accuracy
DELTAIS	79.1% ± 0.82%	77.37% ± 0.83%	64.25% ± 1.09%
HOTKNOT	71.73%	78.47%	59.93%

A close examination of the individual results, shown as a scatter plot in Figure 4.3, indicates that while individual runs vary, DELTAIS is much more accurate with short sequences, while HOTKNOT is more accurate with larger sequences. To emphasize where, on average, each method is more accurate, Figure 4.3 uses a exponential plot with a simple linear regression trend line.

In addition to looking at the overal statistics and evaluating which method is more accurate for which sequence lengths, it is informative to look at which method predicts more perfect secondary structures. This method not only gives us an additional method of comparison, but reminds us of one of the key differences between HOTKNOT and DELTAIS

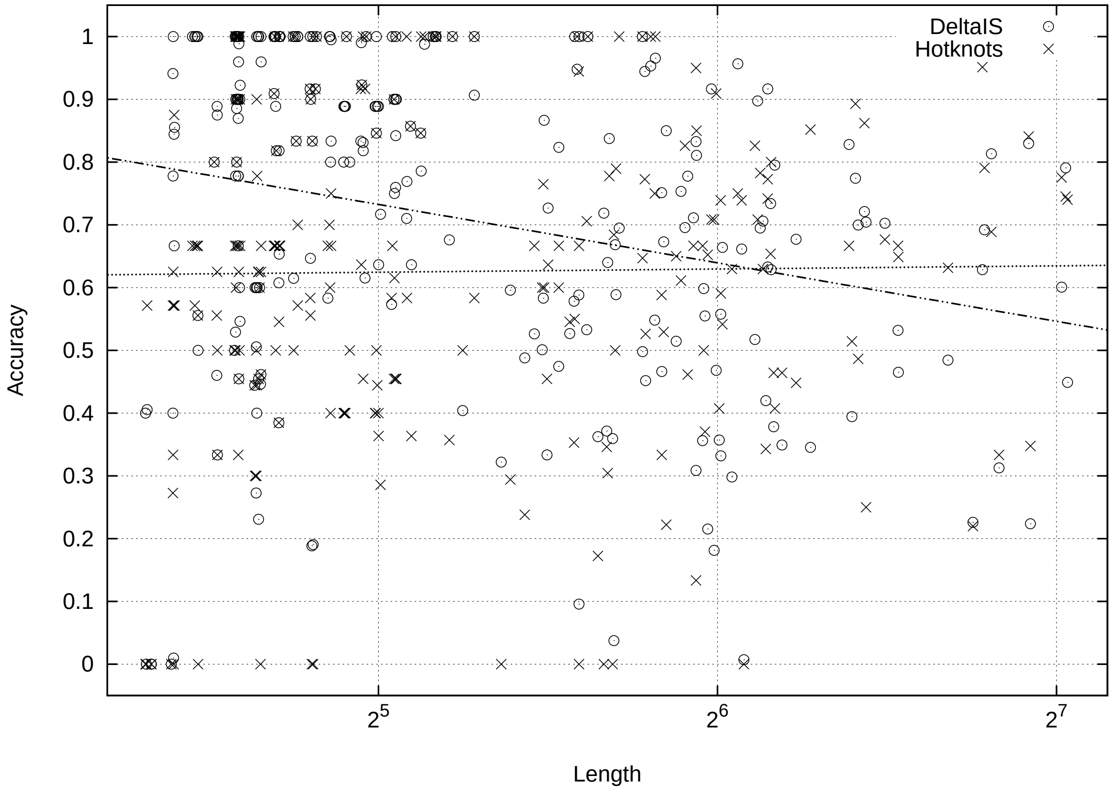


Figure 4.3. Scatter plot results of HotKnot and DeltaIS. A trend line is show for both methods: the increasing line is HotKnot, the decreasing DeltaIS. The trend lines intersect at 68.59, indicating DeltaIS is better for lengths less than 69, while HotKnot is better for lengths 69 and greater.

method. This key difference is that while HotKnot always predicts the same secondary structure for the same sequence, DeltaIS' predicted structures may vary. To provide an accurate comparison between the two methods, in Figure 4.4 we present a histogram comparing the number of structures predicted by HotKnot and DeltaIS. Since the predictions of DeltaIS vary, we present two separate bars for comparison. The first, DeltaIS-AND, represents the percent of the sequences predicted perfectly in all test runs. The second, DeltaIS-OR, represents the percent of the sequences predicted perfectly in at least one of the test runs. These results are also summarized in Table 4.2. As the table and histogram have shown, DeltaIS is able, in all cases, to predict more perfect structures than HotKnot.

Table 4.2. Number of sequences perfectly predicted by HotKnot and the DeltaIS.

Method	Sequences
HotKnot	14.29%
DeltaIS-AND	18.65%
DeltaIS-OR	32.54%

4.4.2 Reconstructive Results

While DeltaIS proved to be very accurate for predicting RNA secondary structure, DeltaIS proves still more accurate in reconstructing specified RNA configurations. In our tests, DeltaIS is able to successfully reconstruct all 252 sequences with an accuracy of 100%. In addition to reconstructing these sequences accurately, our method is able to reconstruct these sequences quickly, with most sequences reconstructed in a single round, and all 252 sequences correctly reconstructed in less than 15 minutes.

While it may seem strange that reconstruction is so much faster and more accurate than prediction, the difference can be understood by examining the key difference between the problems. When predicting, the best solution is unknown, while reconstructing a portion of the solution is given. During prediction a single pairing may be accepted and rejected several times before settling on a solution, while in the reconstruction process once a solution

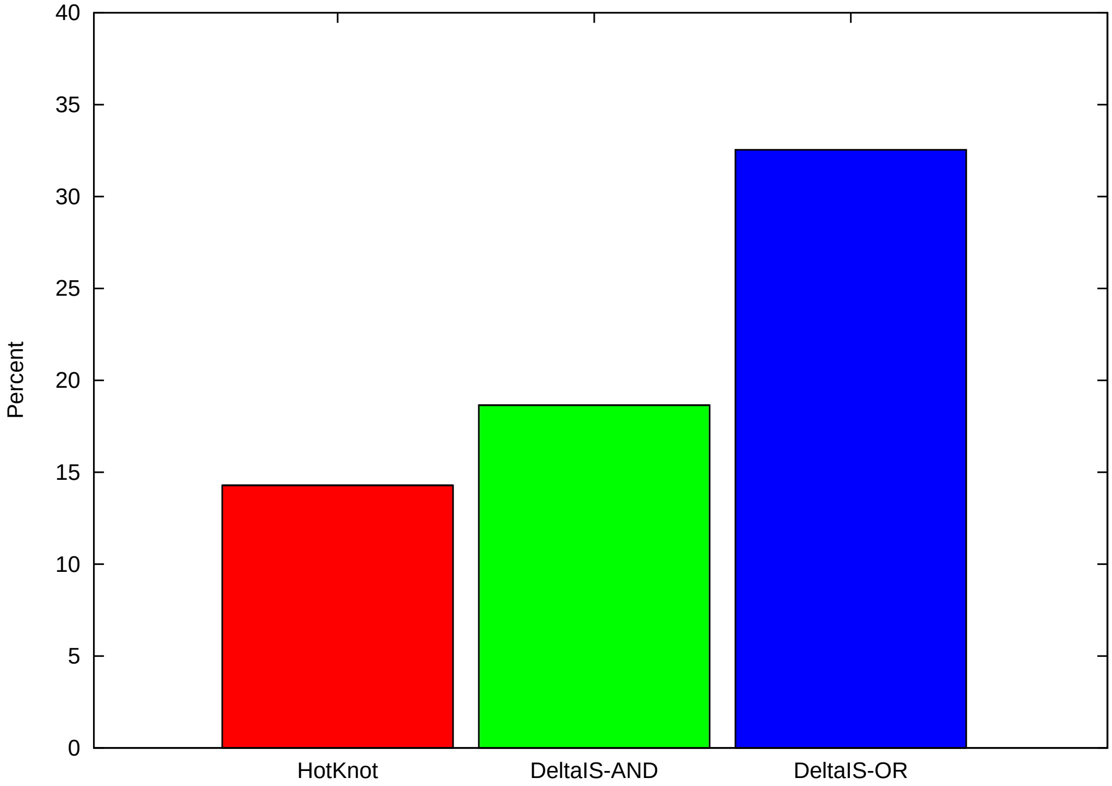


Figure 4.4. Percentage of sequences perfectly predicted by HotKnot and DeltaIS. HotKnot perfectly predicts 14.29% of the sequences. DeltaIS perfectly predicts 18.65% of the sequences all the time, and predicts 32.54% of the sequences at least once.

is found, the scoring function is designed to save the known solution. Having portions of the solution saved makes finding the remainder of the solution significantely easier.

CHAPTER 5
CONCLUSIONS

Solving the RNA folding problem is an important step to understanding how RNA functions. Through the years this problem has been extensively studied, and many interesting and efficient strategies have been proposed. These solution include dynamic programming solutions, heuristic solutions, genetic algorithms, and many more. In this work we have demonstrated that the 3D triangular lattice, when combined with a 2-interval graph post processing step, is an efficient way to accurately predict RNA secondary structures as well. We have shown that it is particularly effective at predicting short RNA sequences. While we have shown this solution is an improvement on current methods, and we provide the tools to immediately start using the DeltaIS prediction process, this is not the only value we provide to the bioinformatics community. A major accomplishment of this work is the Delta Library. This library provides all the tools necessary to work with the 3D triangular lattice, and it has been reviewed, refactored, and optimized to ensure its accuracy, efficiency, and usability. Through the use of this library and the accompanying application programming interface, we hope to provide future researchers the tools to build on our work and continue working towards the ultimate solution to the RNA folding problem.

In the future we hope to continue improving DeltaIS. We are currently working on improving our independent set algorithm. Currently Is only implements a greedy algorithm, and while this algorithm has substantially improved the accuracy of DeltaIS, we believe it can be improved upon further.

Another potential improvement to DeltaIS could be achieved by creating a hybrid search process between the standard and reconstruction modes of Fold. Instead of either looking for an optimal structure or trying to reproduce a specified structure while using a minimum number of sharp turns, the new mode could accept “hints.” These hints could be used to guide the program to good initial configurations, while still providing the freedom to find better solutions. This new search method could be used alternately with 2-interval, with one method’s output being used to influence the other’s result. The accuracy of any prediction done on the 3D triangular lattice is partially determined by the quality of the starting solution. This is intuitive becauses the closer you are to the optimal configuration, the more likely you are to find it. The problem we currently face is that the high initial temperatures required by our cooling function largely negate any benefit of a better starting configuration. The walk is initially so random (because the probability of acceptance is so high), that benefits of the initial configuration may become so distant that the annealer never returns to the configuration, getting stuck in local maximums or just not finding an acceptable path back to the configurations. Implementing this new search method could provide a way to guide the search, keeping the search close to an initial configuration.

More work could also be done with the cooling function. Our research has shown that increasing Delta's base move set has very little effect on the outcome, but changing the cooling function of the simulated annealer can change the results drastically. This drastic change is largely due to the function's ability to escape local maximums. Using the mixing strategy provided an excellent method for leaving local maximums, as did Doubling Steps and multiple Rounds, but these are clearly not the only ways to solve this problem. Reinforcement learning is a possible avenue for research into this problem. A learning agent could be tasked with learning a more efficient cooling schedule, or with learning how to adjust the temperature in response to the rate of improvement of the best solution found, or in response to the relative quality of the solutions currently being found.

The DeltaIS project, consisting of the Fold, Show, and Is tools, along with the Delta library, has been shown to be an effective means for predicting an RNA sequence's secondary structure. It is hoped the tools will be quickly accepted for everyday use in RNA structure prediction, and that the provided library will be valuable to the Bioinformatics community at large.

REFERENCES

[1] J. Ren, B. Rastegari, A. Condon, and H. H. Hoos, “Hotknots: heuristic prediction of rna secondary structures including pseudoknots,” *RNA*, vol. 11, no. 10, pp. 1494–1504, October 2005.

[2] M. Jiang, M. Mayne, and J. Gillespie, “Delta: A toolset for the structural analysis of biological sequences on a 3d triangular lattice,” 2007, pp. 518–529.

[3] M. Mayne, “Folding biological sequences on a three-dimensional triangular lattice: Algorithms and tools,” Master’s thesis, Utah State University, Logan, UT, 2007.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[5] N. Lesh, M. Mitzenmacher, and S. Whitesides, “A complete and effective move set for simplified protein folding,” in *RECOMB ’03: Proceedings of the seventh annual international conference on Research in computational molecular biology.* New York, NY, USA: ACM Press, 2003, pp. 188–195.

[6] M. Jiang and B. Zhu, “Protein folding on the hexagonal lattice in the hp model,” *J Bioinform Comput Biol*, vol. 3, no. 1, pp. 19–34, February 2005.

[7] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equation of state calculations by fast computing machines,” *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[8] P. J. Fleming, H. Gong, and G. D. Rose, “Secondary structure determines protein topology,” *Protein Sci*, vol. 15, no. 8, pp. 1829–1834, August 2006.

[9] M. Schmitz and G. Steger, “Description of rna folding by ‘simulated annealing’,” *Journal of Molecular Biology*, vol. 255, no. 1, pp. 254–266, January 1996.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[11] K. T. Simons, C. Kooperberg, E. Huang, and D. Baker, “Assembly of protein tertiary structures from fragments with similar local sequences using simulated annealing and bayesian scoring functions,” *J Mol Biol*, vol. 268, no. 1, pp. 209–225, April 1997.

[12] Y. Okamoto, M. Fukugita, T. Nakazawa, and H. Kawai, “alpha-helix folding by monte carlo simulated annealing in isolated c-peptide of ribonuclease a,” *Protein Eng.*, vol. 4, no. 6, pp. 639–647, August 1991.

[13] M. Nilges, A. M. Gronenborn, A. T. Brunger, and M. G. Clore, “Determination of three-dimensional structures of proteins by simulated annealing with interproton distance restraints. application to crambin, potato carboxypeptidase inhibitor and barley serine proteinase inhibitor 2,” *Protein Eng.*, vol. 2, no. 1, pp. 27–38, April 1988.

[14] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, “Algorithms for loop matchings,” *SIAM Journal on Applied Mathematics*, vol. 35, no. 1, pp. 68–82, July 1978.

[15] S. R. Eddy, “How do rna folding algorithms work?” *Nature Biotechnology*, vol. 22, no. 11, pp. 1457+.

[16] I. Tinoco, P. N. Borer, B. Dengler, M. D. Levin, O. C. Uhlenbeck, D. M. Crothers, and J. Bralla, “Improved estimation of secondary structure in ribonucleic acids,” *Nature: New Biology*, vol. 246, no. 150, pp. 40–41, November 1973.

[17] G. Vernizzi and H. Orland, “Large-n random matrices for rna folding,” *Acta Physica Polonica B*, vol. 36, pp. 2821+, September 2005.

[18] M. Zuker, “Mfold web server for nucleic acid folding and hybridization prediction,” *Nucleic Acids Res*, vol. 31, no. 13, pp. 3406–3415, July 2003.

[19] S. Russell and P. Norvig, *Local Search Algorithms and Optimization Problems*, 2nd ed.

Upper Saddle River, New Jersey 07458: Pearson Education, Inc., 2003, ch. 4.3, pp. 110–119.

[20] M. S. Waterman, *Introduction to Computational Biology: Maps, Sequences and Genomes.* London: Chapman & Hall, 1995.

[21] S. Vialette, “On the computational complexity of 2-interval pattern matching problems,” *Theor. Comput. Sci.*, vol. 312, no. 2-3, pp. 223–249, 2004.

[22] F. Batenburg, A. P. Gultyaev, C. W. A. Pleij, J. Ng, and J. Oliehoek, “Pseudobase: a database with rna pseudoknots,” *Nucleic Acids Research*, vol. 28, no. 1, pp. 201–204, 2000.

[23] E. Rivas and S. R. Eddy, “A dynamic programming algorithm for rna structure prediction including pseudoknots,” *J Mol Biol*, vol. 285, no. 5, pp. 2053–2068, February 1999.

[24] R. M. Dirks and N. A. Pierce, “A partition function algorithm for nucleic acid secondary structure including pseudoknots,” *J Comput Chem*, vol. 24, no. 13, pp. 1664–1677, October 2003.

[25] J. Ruan, G. D. Stormo, and W. Zhang, “An iterative loop matching approach to the prediction of rna secondary structures with pseudoknots,” in *CSB ’03: Proceedings of the IEEE Computer Society Conference on Bioinformatics.* Washington, DC, USA: IEEE Computer Society, 2003.

[26] F. H. van Batenburg, A. P. Gultyaev, and C. W. Pleij, “An apl-programmed genetic algorithm for the prediction of rna secondary structure,” *J Theor Biol*, vol. 174, no. 3, pp. 269–280, June 1995.

[27] J. Reeder and R. Giegerich, “Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics,” *BMC Bioinformatics*, vol. 5, August 2004.

APPENDICES

APPENDIX A

SOFTWARE MANUAL

Fold

A folding simulation program for the prediction of RNA secondary structures and the reconstruction of RNA tertiary structures.

OPTIONS

`-i file`

Read the sequence of bases (and optionally the sequence of turn directions) in `seq` or `seq2` format from file.

`-i2 file`

Read base pairs in `bp` format from file for reconstruction.

`-o file`

Write lattice conformation (a sequence of bases and a sequence of turn directions) in `seq2` format to file.

`-o2 file`

Write base pairs (realized by lattice conformation) in `bp` format to file.

`-movie`

Turn on movie mode. Write the movie sequence of pull moves to `stdout`; each line of the output contains two numbers for the base index and the turn direction of a pull move. By default, movie is off.

`-v`

Write verbose messages of folding simulation to `stderr`. By default, verbose is off.

`-a repeats [steps]`

Repeat the annealing procedure for multiple iterations and use the given number of steps in each iteration. By default, the number of repeats (iterations) is 5, and the number of steps in each iteration is 100 times the square of the sequence length.

`-d`

Turn on doubling steps for folding simulation. By default, doubling is off.

`-e`

Turn on early-terminate for reconstruction so that the annealing procedure is stopped as soon as a good lattice conformation is reconstructed. By default, early-terminate is off.

-s seed

Initialize random number generator with seed instead of the current time.

-2

Use 2D triangular lattice instead of 3D triangular lattice.

EXAMPLES

```
./fold.exe -i xxx.seq -o xxx.seq2 -o2 xxx.bp -d
```

Perform folding simulation (predict mode) on sequence xxx.seq, use doubling steps, write lattice conformation to xxx.seq2 and predicted base pairs to xxx.bp.

```
./fold.exe -i xxx.seq -i2 xxx.bp -o xxx.seq2 -e
```

Perform folding simulation (reconstruct mode) on sequence xxx.seq with base pairs xxx.bp, use early-terminate, write reconstructed lattice conformation to xxx.seq2.

Show

A 3D visualization program for displaying the lattice conformations of RNA secondary structures and for playing RNA folding movies.

OPTIONS

`-i file`

Read the sequence of bases (and optionally the sequence of turn directions) in `seq` or `seq2` format from file.

`-i2 file`

Read base pairs in `bp` format from file. If file is not specified, then all complementary bases adjacent in the lattice are considered base pairs.

`-o file`

Write lattice conformation (a sequence of bases and a sequence of turn directions) in `seq2` format to file when requested by user.

`-movie`

Turn on movie mode. Read the movie sequence of pull moves from `stdin`; each line of the input contains two numbers for the base index and the turn direction of a pull move. By default, movie is off.

EXAMPLES

```
./show.exe -i xxx.seq2
```

Display lattice conformation `xxx.seq2`.

```
./show.exe -i xxx.seq2 -i2 xxx.bp
```

Display lattice conformation `xxx.seq2` with base pairs `xxx.bp`.

```
./show.exe -i xxx.seq2 -o yyy.seq2
```

Display lattice conformation `xxx.seq2` and save modified conformation to `yyy.seq2` when requested by user.

```
./fold.exe -i xxx.seq -movie | ./show.exe -i xxx.seq -movie
```

Play folding simulation movie for `xxx.seq`.

Graphical User Interface

MOUSE

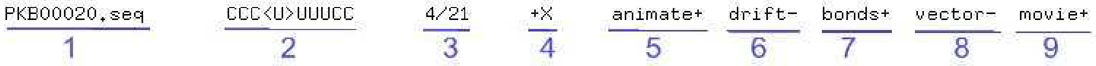
Left-Click and Drag
    Rotate.
Shift Left-Click and Drag
    Rotate (around the Z axis) and zoom.
Right-Click (or Control Click on Apple)
    Menu.

KEYBOARD

`<Escape>`
    Exit from the visualization program.
`'w'`
    Write lattice conformation to file.
`<Left>/<Right>`
    Focus on previous/next base in sequence.
`<Up>/<Down>`
    Change lattice direction of pull move.
`'p'`
    Execute pull move.
`'r'`
    Redo pull move.
`'u'`
    Undo pull move.
`'a'`
    Toggle animation: pulsing ball under focus; smooth transition of focus change. By default, animation is on.
`'d'`
    Toggle drift: slow, continuous, and random view changes. This option is turned off by default, but is automatically turned on in movie mode.
`'b'`
    Toggle bonds: base pairs shown as line segments. By default this option is turned on.
`'v'`
    Toggle vector: direction vector of pull move shown as an arrow. This option is turned off by default, but is automatically turned on whenever the `<Up>/<Down>` keys are pressed.
`'i'`
    Zoom in.
`'o'`
    Zoom out.
`<Space>`
    Reset the view.
`'m'`
    Pause the movie. This option only has effect in movie mode.
`'f'`
    Make the movie faster. This option only has effect in movie mode.
`'s'`
    Make the movie slower. This option only has effect in movie mode.

INFO-BAR

An info-bar is located at the bottom of the visualization window, and provides information on the status of the visualization program.



```
PKB00020.seq    CCC<U>UUUCC    4/21    +X    animate+  drift-  bonds+  vector-  movie+
```

1. The sequence file name.
2. The base under focus and the bases next to it in the sequence.
3. The index of the base under focus / the sequence length.
4. The lattice direction of pull move.
5. Whether animation is on or off. In this case animation is on.
6. Whether drift is on or off. In this case drifting is off.
7. Whether bonds are on or off. In this case bonds are on.
8. Whether vector is on or off. In this case vector is off.
9. Whether movie is running or paused. In this case movie is running. When not in movie mode, neither + nor - is displayed.

Select

Reads helices from `stdin` and sequence from input file, computes a set of disjoint base pairs, and writes the corresponding helices to `stdout`.

INPUT

```
1 10 2
2 21 6
11 29 2
14 27 2
```

OUTPUT

```
2 21 6
13 28 3
```

USAGE

```
cat PKB00001.delta.hx | ./is.exe PKB00001.seq > PKB00001.delta.is.hx
```

APPENDIX B

APPLICATION PROGRAMMING INTERFACE

Delta Library

[Data Structures] [Lattice] [Pull Move] [RNA-specific] [Input/Output]

A library that includes basic data structures and functions for the 3D triangular lattice, pull moves, file input/output, and RNA-specific information such as stacking pair energies.

Data Structures

```
extern char *bases;
extern char *turns;
```

The array `bases` stores the sequence of RNA bases. The array `turns` stores the sequence of turn directions of consecutive bases, which encodes the lattice conformation; the 12 directions ±x, ±y, ±z, ±u, ±v, ±w are represented by `'X'`, `'x'`, `'Y'`, `'y'`, `'Z'`, `'z'`, `'U'`, `'u'`, `'V'`, `'v'`, `'W'`, `'w'`. The length of the turn sequence is exactly one less the length of the base sequence.

```
typedef struct point {
        struct point *next;
        int x, y, z;
} s_point;

extern s_point *points;
extern int n_points;
```

The structure `s_point` has three fields `x, y, z` for the coordinates of a lattice point, and has a pointer `next` for linking the lattice points of all bases into a hashtable for efficient collision detection and neighborhood exploration. The array `points` stores the lattice points corresponding to the bases in the array `bases`. The integer `n_points` stores the sequence length. Hence the sizes of the three arrays `points`, `bases`, and `turns` are `n_points`, `n_points`, and `n_points - 1`, respectively.

```
extern int *pairs;
extern int n_pairs;
```

The two-dimensional array `pairs` stores the input base pairs (for reconstruction or visualization) as an adjacency matrix. The integer `n_pairs` is the number of such pairs.

Lattice

```
void lattice2();
```

The function `lattice2` configures the Delta library to work on the 2D triangular lattice instead of the default 3D triangular lattice.

```
#define C_w 'w'
#define C_v 'v'
#define C_u 'u'
#define C_z 'z'
#define C_y 'y'
#define C_x 'x'
#define C_0 '0'
#define C_X 'X'
#define C_Y 'Y'
#define C_Z 'Z'
#define C_U 'U'
#define C_V 'V'
#define C_W 'W'

#define I_w -6
#define I_v -5
#define I_u -4
#define I_z -3
#define I_y -2
#define I_x -1
#define I_0  0
#define I_X  1
#define I_Y  2
#define I_Z  3
#define I_U  4
#define I_V  5
#define I_W  6
```

The macros `C_?` and `I_?` specify the character name and integer index, respectively, of the 12 lattice directions in the 3D triangular lattice and a dummy direction 0.

```
char axis_i2c(int i);
int axis_c2i(char c);
```

The function axis_i2c converts the integer index of an axis to its character name. The function axis_c2i performs the reverse conversion, form the character name to the integer index.

```
extern const s_point *vectors;
```

The array vectors is indexed by I_?, and stores the (x, y, z) components of the 12 lattice directions and the dummy direction.

```
int axis_pq(const s_point *p, const s_point *q);
```

This function axis_pq finds the vector from p and q, and returns the corresponding integer index I_? (the dummy index I_0 is returned if the vector is not one of the 12 axis vectors).

```
int adjacent_pq(const s_point *p, const s_point *q);
int adjacent_ij(int i, int j);
```

The boolean function adjacent_pq checks whether two lattice points p and q are adjacent. The boolean function adjacent_ij checks whether points[i] and points[j] are adjacent. The function adjacent_ij calls the function adjacent_pq internally.

```
int neighbor_pd(const s_point *p, int d);
int neighbor_id(int i, int d);
```

The function neighbor_pd returns an index j such that points[j] is the neighbor of p in the direction d, where d is one of the indices I_?. The value -1 is returned if no such index exists. Similarly, the function neighbor_id returns an index j such that points[j] is the neighbor of points[i] in the direction d. The function neighbor_id calls the function neighbor_pd internally.

```
void walk(const s_point *p, int d, s_point *q);
```

The function walk sets q to the lattice point obtained by walking from p in the direction d.

```
void turns_to_points();
void points_to_turns();
```

The function turns_to_points decodes the lattice conformation of the sequence from the array turns (lattice directions) to the array points (lattice points organized into a hashtable). The function points_to_turns encodes `points` to `turns`.

Pull Move

```
typedef struct {
        int i;  /* index */
        int d;  /* direction */
        int i_; /* index for undo */
        int d_; /* direction for undo */
} s_move;
```

The structure `s_move` has two fields `i, d` for the base index and the lattice direction of a pull move, and two fields `i_, d_` for the corresponding undo move.

```
int test(s_move *move);
void pull(const s_move *move);
void undo(const s_move *move);
```

The boolean function `test` checks whether the two fields `i, d` in `move` specify a valid pull move. As a side effect, the function `test` also writes the two fields `i_, d_` in `move` for the corresponding undo move. The function `pull` executes `move` by `i, d`. The function `undo` executes `move` by `i_, d_`.

Always `test` before `pull` to validate `move`. Only `undo` immediately after `pull`, use the same `move`.

RNA-specific

```
int valid_ij(int i, int j);
```

The boolean function `valid_ij` checks whether the two indices `i` and `j` are valid, that is, are within the range of `0` and `n_points - 1` and differ by more than `3` (the hairpin constraint).

```
extern double energies[][6];

#define _AU_    0
#define _CG_    1
#define _GC_    2
#define _UA_    3
#define _GU_    4
#define _UG_    5
```

The two-dimensional array `energies` stores the free energies of stacking pairs formed by Watson-Crick and wobble base pairs with indices `_AU_`, `_CG_`, `_GC_`, `_UA_`, `_GU_`, and `_UG_`. For example, if the two base pairs AU and UG form a stacking pair, then the energy is `energies[_AU_][_UG_]`.

```
int pair_ab(char a, char b);
int pair_ij(int i, int j);
```

The function `pair_ab` returns the index (`_AU_`, `_CG_`, `_GC_`, `_UA_`, `_GU_`, or `_UG_`) of the base pair formed by the two bases `a` and `b`. For example, `pair_ab('A', 'U')` returns `_AU_`. The function `pair_ij` returns the index of the base pair formed by the two bases `bases[i]` and `bases[j]`.

```
int watson_crick(int index);
int wobble(int index);
```

The boolean function `watson_crick` checks whether `index` is one of `_AU_`, `_CG_`, `_GC_`, and `_UA_`. The boolean function `wobble` checks whether `index` is either `_GU_` or `_UG_`.

```
int valid_adjacent_pair(int i, int j);
```

The boolean function `valid_adjacent_pair` checks whether `i` and `j` are valid indices, `bases[i]` and `bases[j]` form a Watson-Crick or wobble base pair, and `points[i]` and `points[j]` are adjacent lattice points.

Input/Output

```
void input_bases_turns(const char *filename);
void output_bases_turns(const char *filename);
void input_pairs(char *filename);
void output_pairs(char *filename);
```

The function `input_bases_turns` reads the input file in `seq` or `seq2` format, and populates the two arrays `bases` and `turns`. If the input file is in `seq` format (which does not include the turn sequence), then the array `turns` is initialized to a stem-loop. The function `output_bases_turns` writes `bases` and `turns` to the output file in `seq2` format.

The function `input_pairs` reads the input file in `bp` format, and populates the adjacency matrix `pairs`. The function `output_pairs` extracts the base pairs from stacking base pairs of the lattice conformation, and writes them to the output file in `bp` format.

APPENDIX C

DELTAIS WEBSITE

Download and Install Software

Download the file all.zip and unzip it. There are three platform-specific Makefiles:

- `Makefile.Cygwin`
- `Makefile.Linux`
- `Makefile.MacOSX`

Copy the file `Makefile.[Your Platform]` to a new file `Makefile`. For example, if you are using an Apple, copy `Makefile.MacOSX` to `Makefile`. Or, if you are using a Windows PC, install Cygwin (with at least these additional packages), then copy `Makefile.Cygwin` to `Makefile`.

To verify that everything works correctly, run the following four commands:

1. `make all`
2. `make PKB00001.result`
3. `make PKB00001.show`
4. `make PKB00001.movie`

The first command `make all` compiles the programs.

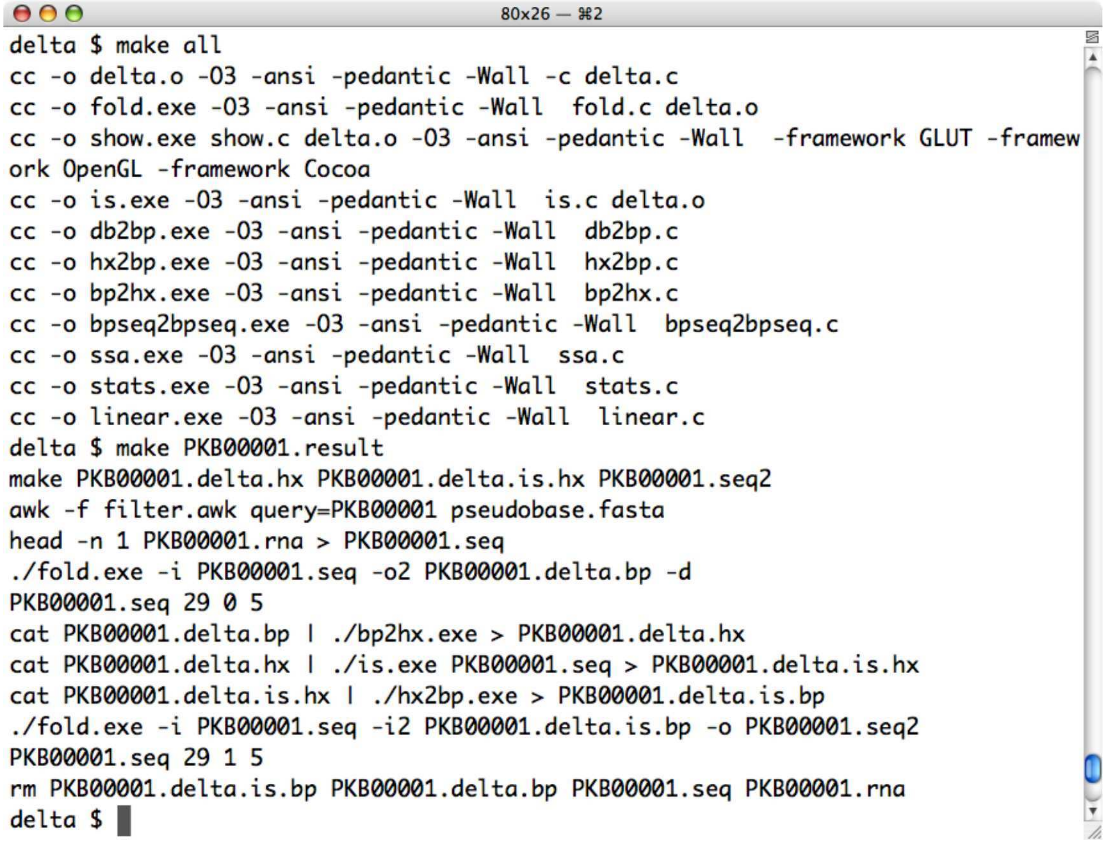
The second command `make PKB00001.result` predicts the secondary structure of the sequence `PKB00001` from PseudoBase. This command takes a few minutes to complete and generates three files:

- `PKB00001.delta.hx`
- `PKB00001.delta.is.hx`
- `PKB00001.seq2`

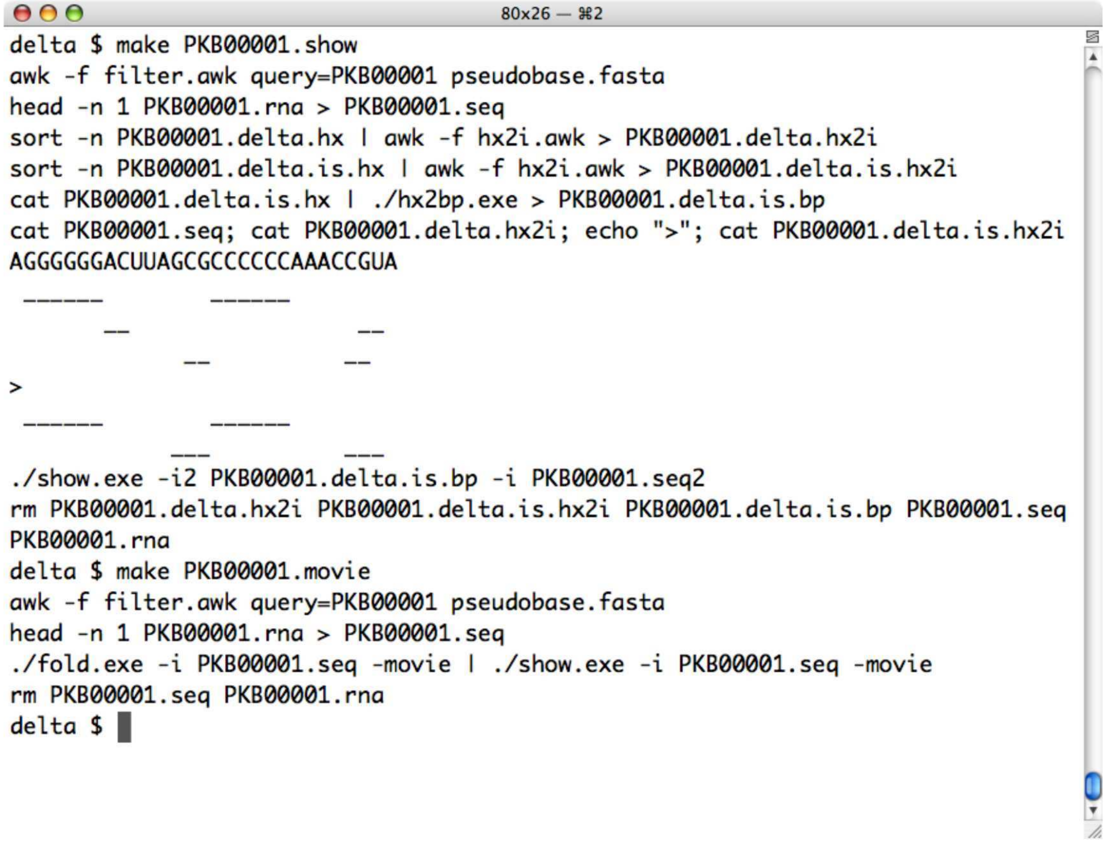
The third command `make PKB00001.show` displays the predicted structure of `PKB00001` using the visualization program. Press the escape key to exit from the visualization program.

The fourth command `make PKB00001.movie` shows a movie of the folding simulation of `PKB00001` in real time. Press the escape key to exit from the visualization program.

The console output of the four commands should be similar to the following two screenshots:



```
delta $ make all
cc -o delta.o -O3 -ansi -pedantic -Wall -c delta.c
cc -o fold.exe -O3 -ansi -pedantic -Wall  fold.c delta.o
cc -o show.exe show.c delta.o -O3 -ansi -pedantic -Wall  -framework GLUT -framew
ork OpenGL -framework Cocoa
cc -o is.exe -O3 -ansi -pedantic -Wall  is.c delta.o
cc -o db2bp.exe -O3 -ansi -pedantic -Wall  db2bp.c
cc -o hx2bp.exe -O3 -ansi -pedantic -Wall  hx2bp.c
cc -o bp2hx.exe -O3 -ansi -pedantic -Wall  bp2hx.c
cc -o bpseq2bpseq.exe -O3 -ansi -pedantic -Wall  bpseq2bpseq.c
cc -o ssa.exe -O3 -ansi -pedantic -Wall  ssa.c
cc -o stats.exe -O3 -ansi -pedantic -Wall  stats.c
cc -o linear.exe -O3 -ansi -pedantic -Wall  linear.c
delta $ make PKB00001.result
make PKB00001.delta.hx PKB00001.delta.is.hx PKB00001.seq2
awk -f filter.awk query=PKB00001 pseudobase.fasta
head -n 1 PKB00001.rna > PKB00001.seq
./fold.exe -i PKB00001.seq -o2 PKB00001.delta.bp -d
PKB00001.seq 29 0 5
cat PKB00001.delta.bp | ./bp2hx.exe > PKB00001.delta.hx
cat PKB00001.delta.hx | ./is.exe PKB00001.seq > PKB00001.delta.is.hx
cat PKB00001.delta.is.hx | ./hx2bp.exe > PKB00001.delta.is.bp
./fold.exe -i PKB00001.seq -i2 PKB00001.delta.is.bp -o PKB00001.seq2
PKB00001.seq 29 1 5
rm PKB00001.delta.is.bp PKB00001.delta.bp PKB00001.seq PKB00001.rna
delta $ █
```



```
delta $ make PKB00001.show
awk -f filter.awk query=PKB00001 pseudobase.fasta
head -n 1 PKB00001.rna > PKB00001.seq
sort -n PKB00001.delta.hx | awk -f hx2i.awk > PKB00001.delta.hx2i
sort -n PKB00001.delta.is.hx | awk -f hx2i.awk > PKB00001.delta.is.hx2i
cat PKB00001.delta.is.hx | ./hx2bp.exe > PKB00001.delta.is.bp
cat PKB00001.seq; cat PKB00001.delta.hx2i; echo ">"; cat PKB00001.delta.is.hx2i
AGGGGGGACUUAGCGCCCCCCAAACCGUA
 ------        ------
       --                 --
             --          --
>
 ------        ------
            ---          ---
./show.exe -i2 PKB00001.delta.is.bp -i PKB00001.seq2
rm PKB00001.delta.hx2i PKB00001.delta.is.hx2i PKB00001.delta.is.bp PKB00001.seq
PKB00001.rna
delta $ make PKB00001.movie
awk -f filter.awk query=PKB00001 pseudobase.fasta
head -n 1 PKB00001.rna > PKB00001.seq
./fold.exe -i PKB00001.seq -movie | ./show.exe -i PKB00001.seq -movie
rm PKB00001.seq PKB00001.rna
delta $ █
```

Make

The three platform-specific makefiles `Makefile.Cygwin`, `Makefile.Linux`, and `Makefile.MacOSX` handles the small differences in operating systems. The generic Makefile `Makefile.generic`, which is included by the three platform-specific makefiles, does the bulk of the work. The file `Makefile.generic` consists of several sections:

- **Top Section**: This section defines macros for groups of files and dependencies for executables. In particular,
    - `make all` compiles all programs,
    - `make clean` removes temporary files,
    - `make clobber` removes compiled executables,
    - `make spotless` removes results files.
- **PseudoBase**: This section provides rules to extract RNA data from `pseudobase.fasta`, to split the data into RNA sequence and dot-brackets in `seq` and `db` formats, and to convert the dot-brackets to base pairs and helices in `pkb.bp` and `pkb.hx` formats. It also contains a target `pkb` for the preparation of experiments.
- **HotKnot**: This section provides rules to convert the `bpseq` files precomputed by HotKnot to `hotknot.bp` and `hotknot.hx` formats, and a target `hot` for the preparation of experiments. Note that this section also contains rules to compute RNA secondary structures using HotKnot, but these rules are commented out because HotKnot works on Linux only.
- **Delta**: This section provides rules to use our folding simulation program and 3D visualization program on individual sequences. Note in particular the three targets `result`, `show`, and `movie` discussed in Download and Install Software.
- **Experiments and Analysis**: This section includes all the scripts for automating experiments and analysis. Note in particular the targets `delta`, `stop`, `txt`, `scr`, `scatter.pdf`, and `reconstruct` discussed in Repeat Experiments.
- **Software Package** This section provides rules for two targets: `all.zip` is an archive of the complete software package; `code.pdf` is a single-pdf printout of the complete source code.

The file `Makefile.generic` contains many shortcuts for typical tasks. For example,

- `make PKB00001.seq` extracts the sequence `PKB00001.seq` from `pseudobase.fasta`.

- `make PKB00001.delta.bp` uses the folding simulation program to predict base pairs for the sequence `PKB00001.seq`.
- `make PKB00001.delta.hx` converts the base pairs to helices.
- `make PKB00001.delta.hx2i` converts helices to 2-intervals in textual format.

Repeat Experiments

RNA Secondary Structure Prediction

make delta

Run the command `make delta`. This command starts an infinite loop. Each run of the loop uses the folding simulation program to predict the secondary structures for all 252 sequences in the data set, and collects the result files `PKB?????.delta.hx` into a zip file `mmddHHMM.delta.zip`, where the eight numbers `mmddHHMM` record the month, day, hour, and minute of the starting time of the run. Each run take 20-30 hours to complete. During each run, hundreds of temporary files `PKB?????.rna`, `PKB?????.seq`, `PKB?????.db`, `PKB?????.pkb.hx`, and `PKB?????.delta.bp` are generated.

make stop

After getting a sufficient number of the zip files `mmddHHMM.delta.zip`, you need to stop the experiment. Do NOT press Ctrl-C. Instead, run the command `make stop` in the same directory from another shell. This command uses the program `killall` to terminate all processes of the infinite loop.

make txt

Next run the command `make txt`. This command generates a text file `mmddHHMM.summary.txt` for each zip file `mmddHHMM.delta.zip`. The creation of each text file takes about two minutes. Each text file `mmddHHMM.summary.txt` contains a summary of prediction results for each sequence as the following:

```
PKB00001  29   9   5  0.62   9   0   0  1.00  1.00  1.00 DeltaIS
PKB00001  29   9   5  0.62   6   3   0  0.67  1.00  0.67 HotKnot
AGGGGGGACUUAGCGCCCCCCAAACCGUA
:[[[[[[:::::(((]]]]]]::::))):
 ______        ______
       __                 __
          __               __
             __          __
>
 ______        ______
```

```
                ___           ___
>
 _______        _______
>
 _______        _______
                ___           ___
```

The example above is the summary for the sequence `PKB00001`. The first two lines contain the following: sequence ID, sequence length, number of base pairs, maximum gap (maximum number of consecutive unpaired bases), density (fraction) of paired bases, true positives, false negatives, false positives, sensitivity, selectivity, accuracy, and DeltaIS or HotKnot. The third line contains the RNA sequence. The fourth line contains the known secondary structure from PseudoBase in dot-bracket format. The remaining lines are separated by `>` into four groups of helices in 2-interval format: the first group and the second group are the predictions of DeltaIS before and after the selection; the third group is the prediction of HotKnot; the fourth group is the known secondary structure from PseudoBase.

make sts and scatter.pdf

Next run the command `make -s sts`. This command calculates the statistics of the overall performances of DeltaIS and HotKnot. The output contains five lines as the following:

```
0.7911  0.0082  0.7739  0.0083  0.6426  0.0109
0.7169  0.0000  0.7847  0.0000  0.5990  0.0000
      36
      82
      47
```

The first line is for DeltaIS: here the sensitivity has average 79.11% and standard deviation 0.82%, the selectivity has average 77.39% and standard deviation 0.83%, and the accuracy has average 64.26% and standard deviation 1.09%. The second line is for HotKnot. The number in the third line says that HotKnot predicted 36 secondary structures perfectly. The number in the fourth line says that DeltaIS predicted 82 perfectly in at least one run and 47 perfectly in all runs.

Finally, run the command `make scatter.pdf`. This command generates a scatter plot of the prediction accuracies of DeltaIS and HotKnot on individual sequences.

RNA Tertiary Structure Reconstruction

Run the command `make reconstruct`. This commands takes about 15 minutes to complete and generates a file `reconstruct.txt`. Each line of the file has four fields: sequence ID, sequence length, 1 or 0 indicating whether the reconstruction was successful, and number of iterations actually used for the reconstruction.

Results of Our Experiments

- delta_zip.zip contains 66 zip files `*.delta.zip` obtained by our experiment on RNA secondary structure prediction.
- summary_txt.zip contains the corresponding 66 summary files `*.summary.txt`.
- scatter.pdf is the scatter plot for the 66 runs.
- reconstruct.txt is the output of our experiment on RNA tertiary structure reconstruction.

Data Formats

1. RNA (.rna):

   Two lines: a sequence of bases in the first line, and a dot-bracket representation of the base pairs in the second line. Individual RNA files can be extracted from the file `pseudobase.fasta`.
   Example:

   ```
   CGGUCAUAAGAGAUAAGCUAGCGUCCUAAUCUAUCCCGGGUUAUGGCGCGAAACUCAGGGA
   (((((((((::::::::::::::::::::::::[[[[[[[))))))):))):::]]]:]]]]
   ```

2. Sequence (.seq):

   A sequence of bases in one line, as in the first line of the RNA file.
   Example:

   ```
   CGGUCAUAAGAGAUAAGCUAGCGUCCUAAUCUAUCCCGGGUUAUGGCGCGAAACUCAGGGA
   ```

3. Dot-brackets (.db):

   A sequence of dots (colons) and brackets (parentheses, square brackets, and curly braces) that represent the base pairs, as in the second line of the RNA file.
   Example:

   ```
   (((((((((::::::::::::::::::::::::[[[[[[[))))))):))):::]]]:]]]]
   ```

4. Base pairs (.bp):

   A file consists of multiple lines. Each line consists of two numbers separated by a space. The two numbers are the indices of the two bases forming a base pair (the first base in the RNA sequence has index 1).
   Example:

   ```
   11 32
   12 31
   13 30
   14 29
   21 49
   22 48
   23 47
   24 46
   25 45
   34 61
   35 60
   ```

```
36 59
37 58
```

5. Helices (.hx):

   A compact representation of base pairs in multiple lines. Each line consists of three numbers separated by spaces: the first two numbers are the indices of the outer-most base pair in the helix; the third number is the helix length, that is, the number of base pairs in the helix.
   Example:

```
11 32 4
21 49 5
34 61 4
```

6. Lattice conformation (.seq2):

   A sequence of bases and a sequence of turn directions. The two sequences are separated by an empty line. The number of turns is exactly the number of bases minus one.
   Example:

```
CGGUCAUAAGAGAUAAGCUAGCGUCCUAAUCUAUCCCGGGUUAUGGCGCGAAACUCAGGGA

zvXVUyxyZWYxVVZwzwYuuWXXvZXZvyXYYwwuZxzzVZXVYxyyuuXvyZXUzXWX
```

7. BPSEQ (.bpseq):

   A file consists of multiple lines, one line for each base in the sequence. Each line contains three fields: the index, the base, and the index to the other base in the pair (zero means unpaired).
   Example:

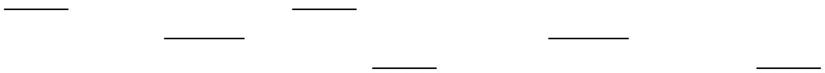
```
1 C 0
2 G 0
3 G 0
4 U 0
5 C 0
6 A 0
7 U 0
8 A 0
9 A 0
10 G 0
11 A 32
12 G 31
13 A 30
14 U 29
15 A 0
16 A 0
17 G 0
18 C 0
19 U 0
20 A 0
```

```
21 G 49
22 C 48
23 G 47
24 U 46
25 C 45
26 C 0
27 U 0
28 A 0
29 A 14
30 U 13
31 C 12
32 U 11
33 A 0
34 U 61
35 C 60
36 C 59
37 C 58
38 G 0
39 G 0
40 G 0
41 U 0
42 U 0
43 A 0
44 U 0
45 G 25
46 G 24
47 C 23
48 G 22
49 C 21
50 G 0
51 A 0
52 A 0
53 A 0
54 C 0
55 U 0
56 C 0
57 A 0
58 G 37
59 G 36
60 G 35
61 A 34
```

8. Helices as 2-intervals (.hx2i):

 A textual representation of helices.

 Example:



Analyze

linear
: Linear regression.

ssa
: Sensitivity, specificity, and accuracy.

stats
: Average and standard deviation.

linear

Reads lines of two numbers (X and Y coordinates) from `stdin`, performs linear regression, and writes the slope and the Y intersection to `stdout`.

INPUT

```
1   100.0
10  88.0
20  78.0
30  65.0
40  61.2
50  50.1
```

OUTPUT

```
-0.992582 98.6966
```

USAGE

```
cat data_file | ./linear.exe
```

ssa

Reads two sets of helices from two input files, compares the base pairs, and writes six numbers true positives, false negatives, false positives, sensitivity, specificity, and accuracy to `stdout`.

INPUT

```
1 10 3
4 14 2
```

INPUT

```
2 9 3
5 14 2
16 20 2
```

OUTPUT

```
  2   5   3  0.29  0.40  0.20
```

USAGE

./ssa *prediction_file answer_file*

stats

Reads lines of three numbers (1: sensitivity, 2: specificity, and 3: accuracy) from `stdin`, calculates the average and standard deviation of these numbers, and writes six numbers (average 1, stdev 1, average 2, stdev 2, average 3, stdev3) to `stdout`.

INPUT

```
0.4 19.7 19.7
2.8 19.1 19.3
4.0 18.2 18.6
6.0 5.2 7.9
1.1 4.3 4.4
2.6 9.3 9.6
7.1 3.6 8.0
5.3 14.8 15.7
9.7 11.9 15.4
3.1 9.3 9.8
9.9 2.8 10.3
5.3 9.9 11.2
6.7 15.4 16.8
4.3 2.7 5.1
6.1 10.6 12.2
9.0 16.6 18.9
4.2 11.4 12.2
4.5 18.8 19.3
5.2 15.6 16.5
4.3 17.9 18.4
```

OUTPUT

```
5.0800  2.5665  11.8550  5.8527  13.4650  5.0053
```

USAGE

```
cat data ./stats.exe > results_file
```

Convert

bpseq2bpseq

From `bases pairs` and `sequence` to `BPSEQ`.

db2bp

From `dot-brackets` to `base pairs`.

hx2bp

From `helices` to `base pairs`.

bp2hx

From `base pairs` to `helices`.

hx2i

From `helices` to `2-intervals`.

_

bp2hx

Reads base pairs from `stdin`, writes helices to `stdout`.

INPUT

```
1 10
2 9
2 21
3 20
4 19
5 18
6 17
7 16
11 29
12 28
14 27
15 26
```

OUTPUT

```
1 10 2
2 21 6
11 29 2
14 27 2
```

USAGE

```
cat PKB00001.delta.bp | ./bp2hx.exe > PKB00001.delta.hx
```

bpseq2bpseq

Reads base pairs from `stdin` and sequence from input file, writes base pairs and sequence in `bpseq` format to `stdout`.

INPUT

```
2 21
3 20
4 19
5 18
6 17
7 16
13 28
14 27
15 26
```

INPUT

```
AGGGGGGACUUAGCGCCCCCCAAACCGUA
```

OUTPUT

```
1 A 0
2 G 21
3 G 20
4 G 19
5 G 18
6 G 17
7 G 16
8 A 0
9 C 0
10 U 0
11 U 0
12 A 0
13 G 28
14 C 27
15 G 26
16 C 7
17 C 6
18 C 5
19 C 4
20 C 3
21 C 2
22 A 0
23 A 0
24 A 0
25 C 0
26 C 15
27 G 14
28 U 13
29 A 0
```

USAGE

```
cat PKB00001.delta.is.bp | ./bpseq2bpseq.exe PKB00001.seq > PKB00001.delta.is.bpseq
```

–

db2bp

Reads dot-brackets from `stdin`, writes base pairs to `stdout`.

INPUT

```
:[[[[[[:::::(((]]]]]]::::))):
```

OUTPUT

```
7 16
6 17
5 18
4 19
3 20
2 21
15 26
14 27
13 28
```

EXAMPLE USAGE

```
cat PKB00001.db | ./db2bp.exe > PKB00001.bp
```

hx2bp

Reads helices from `stdin`, writes base pairs to `stdout`.

INPUT

```
1 10 2
2 21 6
11 29 2
14 27 2
```

OUTPUT

```
1 10
2 9
2 21
3 20
4 19
5 18
6 17
7 16
11 29
12 28
14 27
15 26
```

USAGE

```
cat PKB00001.delta.hx | ./hx2bp.exe > PKB00001.delta.bp
```

hx2i

Reads helices from `stdin`, writes 2-intervals in textual format to `stdout`.

INPUT

```
1 10 2
2 21 6
11 29 2
14 27 2
```

OUTPUT

```
__      __
 ______        ______
          __               __
             __          __
```

USAGE

```
cat PKB00001.delta.hx | awk -f hx2i.awk > PKB00001.delta.hx2i
```

APPENDIX D

DELTAIS SOURCE CODE

D .1 delta.h

```
/*
 *    delta.h - Delta library header
 *
 *    Minghui Jiang, Martin Mayne, and Joel Gillespie
 *    Tue Feb 24 10:18:43 MST 2009
 */

/* data structure */

extern char *bases;
extern char *turns;

typedef struct point {
    struct point *next;
    int x, y, z;
} s_point;

extern s_point *points;
extern int n_points;

extern int *pairs;
extern int n_pairs;

/* lattice */

void lattice2();

#define C_w 'w'
#define C_v 'v'
#define C_u 'u'
#define C_z 'z'
#define C_y 'y'
#define C_x 'x'
#define C_0 '0'
#define C_X 'X'
#define C_Y 'Y'
#define C_Z 'Z'
#define C_U 'U'
#define C_V 'V'
#define C_W 'W'

#define I_w -6
#define I_v -5
#define I_u -4
#define I_z -3
#define I_y -2
#define I_x -1
#define I_0  0
```

```
#define I_X  1
#define I_Y  2
#define I_Z  3
#define I_U  4
#define I_V  5
#define I_W  6

char axis_i2c(int i);
int axis_c2i(char c);

extern const s_point *vectors;

int axis_pq(const s_point *p, const s_point *q);

int adjacent_pq(const s_point *p, const s_point *q);
int adjacent_ij(int i, int j);

int neighbor_pd(const s_point *p, int d);
int neighbor_id(int i, int d);

void walk(const s_point *p, int d, s_point *q);

void turns_to_points();
void points_to_turns();

/* pull move */

typedef struct {
    int i;    /* index */
    int d;    /* direction */
    int i_;    /* index for undo */
    int d_;    /* direction for undo */
} s_move;

int test(s_move *move);
void pull(const s_move *move);
void undo(const s_move *move);

/* RNA-specific */

int valid_ij(int i, int j);

extern double energies[6][6];

#define _AU_    0
#define _CG_    1
#define _GC_    2
#define _UA_    3
#define _GU_    4
#define _UG_    5

int pair_ab(char a, char b);
int pair_ij(int i, int j);

int watson_crick(int index);
int wobble(int index);

int valid_adjacent_pair(int i, int j);

/* input/output */

void input_bases_turns(const char *filename);
void output_bases_turns(const char *filename);
void input_pairs(const char *filename);
void output_pairs(const char *filename);
```

D .2 delta.c

```
/*
 *    delta.c - Delta library code
 *
 *    Minghui Jiang, Martin Mayne, and Joel Gillespie
 *    Tue Feb 24 10:18:43 MST 2009
 */

#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "delta.h"

/* hashtable */

static s_point **hashtable = NULL;
static int hashtable_size = 0;
static double hashcode_magic;

static void hashtable_init(int size) {
    if (hashtable)
        free(hashtable);
    if ((hashtable = malloc(size * sizeof(s_point *))) == NULL) {
        fprintf(stderr, "hashtable_init: malloc error\n");
        exit(1);
    }
    memset(hashtable, 0, size * sizeof(s_point *));
    hashtable_size = size;
    hashcode_magic = (sqrt(5.0) - 1.0) / 2.0;
}

static int hashcode(const s_point *p) {
    double f = ((((p->x * hashcode_magic) + p->y)
                * hashcode_magic) + p->z) * hashcode_magic;

    if (f < 0.0)
        f = -f;
    return (int) (hashtable_size * f) % hashtable_size;
}

static int hashtable_find(const s_point *p) {
    int i = hashcode(p);
    s_point *point;

    for (point = hashtable[i]; point; point = point->next)
        if (point->x == p->x && point->y == p->y && point->z == p->z)
            return point - points;
    return -1;
}

static void hashtable_insert(s_point *p) {
    int i = hashcode(p);
    p->next = hashtable[i];
    hashtable[i] = p;
}

static void hashtable_remove(s_point *p) {
    int i = hashcode(p);
    s_point *point, *prev = NULL;

    for (point = hashtable[i]; point; point = point->next) {
        if (point == p) {
            if (prev)
                prev->next = point->next;
            else
                hashtable[i] = point->next;
            point->next = NULL;
            return;
        }
        prev = point;
    }
}

/* data structure */

char *bases = NULL;
char *turns = NULL;

s_point *points = NULL;
int n_points = 0;

int *pairs = NULL;
int n_pairs = 0;

/* lattice */

#define LATTICE_3D     0
#define LATTICE_2D     1

static const int lattice_axes[4][14] = {
 /* {w,v,u,z,y,x, 0, X,Y,Z,U,V,W} */
    {1,1,1,1,1,1, 0, 1,1,1,1,1,1},    /* LATTICE_3D */
    {0,0,1,1,0,1, 0, 1,0,1,1,0,0},    /* LATTICE_2D: xXzZuU */
};

static const int *valid_vectors = &(lattice_axes[LATTICE_3D][6]);

void lattice2() {
    valid_vectors = &(lattice_axes[LATTICE_2D][6]);
}

char axis_i2c(int i) {
    static const char C_[14] =
        { C_w, C_v, C_u, C_z, C_y, C_x, C_0, C_X, C_Y, C_Z, C_U, C_V, C_W };
    static const char *C = &(C_[6]);
    return C[i];
}

int axis_c2i(char c) {
    switch (c) {
    case 'w': return I_w;
    case 'v': return I_v;
    case 'u': return I_u;
    case 'z': return I_z;
    case 'y': return I_y;
    case 'x': return I_x;
    case 'X': return I_X;
    case 'Y': return I_Y;
    case 'Z': return I_Z;
    case 'U': return I_U;
    case 'V': return I_V;
    case 'W': return I_W;
    default:  return I_0;
    }
}

const s_point vectors_[13] = {
    {NULL, -1,-1,-1},    /* w = -X + -Y + -Z */
    {NULL,  0,-1,-1},    /* v = -Y + -Z */
    {NULL, -1, 0,-1},    /* u = -X + -Z */
    {NULL,  0, 0,-1},    /* z = -Z */
    {NULL,  0,-1, 0},    /* y = -Y */
    {NULL, -1, 0, 0},    /* x = -X */
    {NULL,  0, 0, 0},    /* 0 */
    {NULL,  1, 0, 0},    /* X */
    {NULL,  0, 1, 0},    /* Y */
    {NULL,  0, 0, 1},    /* Z */
    {NULL,  1, 0, 1},    /* U = X + Z */
    {NULL,  0, 1, 1},    /* V = Y + Z */
    {NULL,  1, 1, 1}    /* W = X + Y + Z */
};

const s_point *vectors = &(vectors_[6]);

int axis_pq(const s_point *p, const s_point *q) {
    static const int I[3][3][3] =
    {{{ I_w, I_0, I_0}, { I_u, I_x, I_0}, { I_0, I_0, I_0}},
     {{ I_v, I_y, I_0}, { I_z, I_0, I_Z}, { I_0, I_Y, I_V}},
     {{ I_0, I_0, I_0}, { I_0, I_X, I_U}, { I_0, I_0, I_W}}};
    int x = q->x - p->x;
    int y = q->y - p->y;
    int z = q->z - p->z;

    return (x < -1 || x > 1 || y < -1 || y > 1 || z < -1 || z > 1)
        ? I_0 : I[x + 1][y + 1][z + 1];
}

int adjacent_pq(const s_point *p, const s_point *q) {
    return valid_vectors[axis_pq(p, q)];
}
```

```
int adjacent_ij(int i, int j) {
    return adjacent_pq(&points[i], &points[j]);
}

int neighbor_pd(const s_point *p, int d) {
    s_point o;

    if (!valid_vectors[d])
        return -1;

    walk(p, d, &o);
    return hashtable_find(&o);
}

int neighbor_id(int i, int d) {
    return neighbor_pd(&points[i], d);
}

void walk(const s_point *p, int d, s_point *q) {    /* p + axis[d] = q */
    q->x = p->x + vectors[d].x;
    q->y = p->y + vectors[d].y;
    q->z = p->z + vectors[d].z;
}

void points_to_turns() {
    int i;

    for (i = 0; i < n_points - 1; i++)
        turns[i] = axis_i2c(axis_pq(&points[i], &points[i + 1]));
}

void turns_to_points() {
    int i;

    if (!turns || strlen(turns) < n_points - 1) {
        if (turns)
            free(turns);
        if ((turns = malloc(n_points * sizeof(char))) == NULL) {
            fprintf(stderr, "turns_to_points: malloc error\n");
            exit(1);
        }

        /* stem loop */
        for (i = 0; i < (n_points - 1) / 2; i++)
            turns[i] = ’X’;
        turns[i++] = ’Z’;
        for (; i < n_points - 1; i++)
            turns[i] = ’x’;
        turns[n_points - 1] = ’\0’;
    }

    if (points)
        free(points);
    if ((points = malloc(n_points * sizeof(s_point))) == NULL) {
        fprintf(stderr, "turns_to_points: malloc error\n");
        exit(1);
    }

    hashtable_init(n_points * 2);    /* load factor 0.5 */
    points[0].x = points[0].y = points[0].z = 0;
    hashtable_insert(&points[0]);
    for (i = 1; i < n_points; i++) {
        s_point *q = &points[i];
        s_point *p = &points[i - 1];
        int d = axis_c2i(turns[i - 1]);

        if (!valid_vectors[d]) {
            fprintf(stderr, "turns_to_points: invalid turn %c at %d\n",
                    axis_i2c(d), i);
            exit(1);
        }
        walk(p, d, q);
        if (hashtable_find(q) >= 0) {
            fprintf(stderr, "turns_to_points: collision at %d\n", i);
            exit(1);
        }
        hashtable_insert(q);
    }
}

/* pull move */

int test(s_move *move) {
    s_point o, *p;
    int i = move->i;
    int d = move->d;
    int anchor;

    if (i < 0 || i >= n_points)
        return 0;
    if (d < -6 || d > 6 || !valid_vectors[d])
        return 0;

    walk(&points[i], d, &o);
    if (hashtable_find(&o) >= 0)
        return 0;

    if (i == 0)
        anchor = -1;
    else if (i == n_points - 1)
        anchor = 1;
    else if (adjacent_pq(&o, &points[i - 1]))
        anchor = -1;
    else if (adjacent_pq(&o, &points[i + 1]))
        anchor = 1;
    else
        return 0;

    p = &o;
    if (anchor < 0)
        while (i + 1 < n_points && !adjacent_pq(p, &points[i + 1])) {
            p = &points[i];
            i++;
        }
    else
        while (i - 1 >= 0 && !adjacent_pq(p, &points[i - 1])) {
            p = &points[i];
            i--;
        }
    move->i_ = i;
    move->d_ = axis_pq(p, &points[i]);
    return 1;
}

void pull(const s_move *move) {
    s_point *p = &points[move->i];
    int i, anchor;

    anchor = move->i < move->i_ ? -1 : 1;
    for (i = move->i_; i != move->i; i += anchor) {
        s_point *s = &points[i];
        s_point *t = &points[i + anchor];

        hashtable_remove(s);
        s->x = t->x;
        s->y = t->y;
        s->z = t->z;
        hashtable_insert(s);
    }
    hashtable_remove(p);
    walk(p, move->d, p);
    hashtable_insert(p);
}

void undo(const s_move *move) {
    s_point *p = &points[move->i_];
    int i, anchor;

    anchor = move->i > move->i_ ? -1 : 1;
    for (i = move->i; i != move->i_; i += anchor) {
        s_point *s = &points[i];
        s_point *t = &points[i + anchor];

        hashtable_remove(s);
        s->x = t->x;
        s->y = t->y;
        s->z = t->z;
        hashtable_insert(s);
    }
    hashtable_remove(p);
    walk(p, move->d_, p);
    hashtable_insert(p);
}
```

```
/* RNA-specific */

int valid_ij(int i, int j) {
    int k;

    if (i > j) {
        k = i;
        i = j;
        j = k;
    }
    return i >= 0 && j < n_points && j >= i + 4;
}

double energies[6][6] = {
    {-0.90, -2.20, -2.10, -1.10, -0.60, -1.40},
    {-2.10, -3.30, -2.40, -2.10, -1.40, -2.10},
    {-2.40, -3.40, -3.30, -2.20, -1.50, -2.50},
    {-1.30, -2.40, -2.10, -0.90, -1.00, -1.30},
    {-1.30, -2.50, -2.10, -1.40, -0.50,  1.30},
    {-1.00, -1.50, -1.40, -0.60,  0.30, -0.50}
};

int pair_ab(char a, char b) {
    int index = -1;

    if (a == 'A' && b == 'U')
        index = _AU_;
    else if (a == 'C' && b == 'G')
        index = _CG_;
    else if (a == 'G' && b == 'C')
        index = _GC_;
    else if (a == 'U' && b == 'A')
        index = _UA_;
    else if (a == 'G' && b == 'U')
        index = _GU_;
    else if (a == 'U' && b == 'G')
        index = _UG_;
    return index;
}

int pair_ij(int i, int j) {
    return pair_ab(bases[i], bases[j]);
}

int valid_adjacent_pair(int i, int j) {
    return valid_ij(i, j) && adjacent_pq(&points[i], &points[j])
        ? pair_ab(bases[i], bases[j]) : -1;
}

int watson_crick(int index) {
    return index >= _AU_ && index <= _UA_;
}

int wobble(int index) {
    return index == _GU_ || index == _UG_;
```

```
}

/* input/output */

static char *read1(FILE *file) {
    char c, *s, *t;
    int i, size;
    int count;

    i = 0;
    size = 64;
    if ((s = malloc(size * sizeof(char))) == NULL) {
        fprintf(stderr, "read1: malloc error\n");
        exit(1);
    }

    count = 0;
    while (1) {
        if ((c = fgetc(file)) == EOF)
            break;

        if (c == ’\n’ && ++count >= 2)
            break;
        if (isspace(c))
            continue;
        count = 0;

        if (i == size - 1) {    /* buffer full */
            size *= 2;
            if ((t = malloc(size * sizeof(char))) == NULL) {
                fprintf(stderr, "read1: malloc error\n");
                exit(1);
            }

            s[i] = ’\0’;
            strcpy(t, s);
            free(s);
            s = t;
        }
        s[i++] = c;
    }
    s[i] = ’\0’;
    return s;
}

static void write1(FILE *file, char *s) {
    while (s[0] != ’\0’) {
        s += fprintf(file, "%.40s", s);
        fprintf(file, "\n");
    }
}

void input_bases_turns(const char *filename) {
    FILE *file;
    if ((file = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "input_bases_turns: fopen(%s) error\n", filename);
        exit(1);
    }

    if (bases)
        free(bases);
    bases = read1(file);
    n_points = strlen(bases);
    if (turns)
        free(turns);
    turns = read1(file);
    fclose(file);
}

void output_bases_turns(const char *filename) {
    FILE *file;

    if ((file = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "output_bases_turns: fopen(%s) error\n", filename);
        exit(1);
    }

    write1(file, bases);
    fprintf(file, "\n");
    write1(file, turns);
    fclose(file);
}

void input_pairs(const char *filename) {
    FILE *file;
    int i, j;

    if (pairs)
        free(pairs);
    if ((pairs = malloc(n_points * n_points * sizeof(int))) == NULL) {
        fprintf(stderr, "input_pairs: malloc error\n");
        exit(1);
    }

    memset(pairs, 0, n_points * n_points * sizeof(int));
    n_pairs = 0;

    if ((file = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "input_pairs: fopen(%s) error\n", filename);
        exit(1);
    }

    while (fscanf(file, "%d%d", &i, &j) == 2) {
        if (i < 1 || i > n_points || j < 1 || j > n_points) {
            fprintf(stderr, "input_pairs: invalid pair (%d, %d)\n", i, j);
            exit(1);
        }

        i--;
        j--;
        pairs[i * n_points + j] = pairs[j * n_points + i] = 1;
        n_pairs++;
    }
    fclose(file);
}

void output_pairs(const char *filename) {
    FILE *file;
    int i, j;

    if (pairs)
        free(pairs);
    if ((pairs = malloc(n_points * n_points * sizeof(int))) == NULL) {
        fprintf(stderr, "output_pairs: malloc error\n");
        exit(1);
    }

    memset(pairs, 0, n_points * n_points * sizeof(int));
    n_pairs = 0;

    for (i = 0; i < n_points; i++)
        for (j = i + 1; j < n_points; j++) {
            int k = valid_adjacent_pair(i, j);
            int l = valid_adjacent_pair(i - 1, j + 1);
            int m = valid_adjacent_pair(i + 1, j - 1);

            if (watson_crick(k)) {
                if ((watson_crick(l) || watson_crick(m)))
                    pairs[i * n_points + j] = 1;
            } else if (wobble(k)) {
                if (watson_crick(l) && watson_crick(m)) {
                    pairs[i * n_points + j] = 1;
                    pairs[(i - 1) * n_points + (j + 1)] = 1;
                    pairs[(i + 1) * n_points + (j - 1)] = 1;
                }
            }
        }

    if ((file = fopen(filename, "w")) == NULL) {
        fprintf(stderr, "output_pairs: fopen(%s) error\n", filename);
        exit(1);
    }

    for (i = 0; i < n_points; i++)
        for (j = i + 1; j < n_points; j++)
            if (pairs[i * n_points + j])
                fprintf(file, "%d %d\n", i + 1, j + 1);
    fclose(file);
}
```

D .3 fold.c

```
/*
 *    fold.c - RNA folding simulation
 *
 *    Minghui Jiang, Martin Mayne, and Joel Gillespie
 *    Fri Feb 27 14:28:48 MST 2009
 */

#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include "delta.h"

int movie_mode = 0;
int verbose = 0;

/* scoring for predict: stack_score_i */

double stack_score_ij(int i, int j) {
    double score = 0.0;
    int k, l, m;

    if (i > j) {
        k = i;
        i = j;
        j = k;
    }

    k = valid_adjacent_pair(i, j);
    if (watson_crick(k)) {
        l = valid_adjacent_pair(i - 1, j + 1);
        if (watson_crick(l))
            score -= energies[l][k];
        m = valid_adjacent_pair(i + 1, j - 1);
        if (watson_crick(m))
            score -= energies[k][m];
    } else if (wobble(k)) {
        l = valid_adjacent_pair(i - 1, j + 1);
        if (watson_crick(l)) {
            m = valid_adjacent_pair(i + 1, j - 1);
            if (watson_crick(m))
                score -= (energies[l][k] + energies[k][m]) * 2.0;
        }
    }
    return score;
}

double stack_score_i(int i) {
    double score = 0.0;
    int d, j;
    for (d = -6; d <= 6; d++) {
        double score_d;

        if ((j = neighbor_id(i, d)) < 0)
            continue;

        score_d = stack_score_ij(i, j);

        /* score = minimum! of positive score_d */
        if (score_d > 0.0 && (score_d < score || score == 0.0))
            score = score_d;
    }
    return score;
}

/* scoring for reconstruct: match_count_i, turn_score_i */

#define MATCH_BONUS    5.0

int match_count_i(int i) {
    int count = 0;
    int d, j;

    for (d = -6; d <= 6; d++) {
        if ((j = neighbor_id(i, d)) < 0)
            continue;

        count += pairs[i * n_points + j];
    }
    return count;
}

#define A180 0
#define A120 0
#define A090 1
#define A060 4
#define A000 1000    /* not used */
const int ANGLE_FACTOR[13][13] =
    {{A180,A120,A120,A090,A120,A120,A000,A060,A060,A090,A060,A060,A000},
     {A120,A180,A090,A120,A120,A060,A000,A120,A060,A060,A090,A000,A060},
     {A120,A090,A180,A120,A060,A120,A000,A060,A120,A060,A000,A090,A060},
     {A090,A120,A120,A180,A060,A060,A000,A120,A120,A000,A060,A060,A090},
     {A120,A120,A060,A060,A180,A090,A000,A090,A000,A120,A120,A060,A060},
     {A120,A060,A120,A060,A090,A180,A000,A000,A090,A120,A060,A120,A060},
     {A000,A000,A000,A000,A000,A000,A000,A000,A000,A000,A000,A000,A000},
     {A060,A120,A060,A120,A090,A000,A000,A180,A090,A060,A120,A060,A120},
     {A060,A060,A120,A120,A000,A090,A000,A090,A180,A060,A060,A120,A120},
     {A090,A060,A060,A000,A120,A120,A000,A060,A060,A180,A120,A120,A090},
     {A060,A090,A000,A060,A120,A060,A000,A120,A060,A120,A180,A090,A120},
     {A060,A000,A090,A060,A060,A120,A000,A060,A120,A120,A090,A180,A120},
     {A000,A060,A060,A090,A060,A060,A000,A120,A120,A090,A120,A120,A180}};

double turn_score_i(int i) {
    if (i == 0 || i == n_points - 1)
        return 0.0;
    else {
        int d0 = axis_pq(&points[i - 1], &points[i]);
        int d1 = axis_pq(&points[i], &points[i + 1]);

        /* match_count is more important than turn_score:
           sum of turn_score_i for all i approx< -MATCH_BONUS for one match */
        return -ANGLE_FACTOR[d0 + 6][d1 + 6] * (MATCH_BONUS / n_points);
    }
}

/* scoring: total_score, init_delta, delta */

int n_matches = 0;    /* for reconstruct */

double total_score() {
    double score = 0.0;
    int i;

    if (pairs) {    /* for reconstruct */
        n_matches = 0;
        for (i = 0; i < n_points; i++) {
            score += turn_score_i(i);
            n_matches += match_count_i(i);
        }
        score += n_matches * MATCH_BONUS;
    } else    /* for predict */
        for (i = 0; i < n_points; i++)
            score += stack_score_i(i);
    return score;
}

int *indices = NULL;
int *flags = NULL;

void init_delta() {
    if ((indices = malloc(n_points * sizeof(int))) == NULL) {
        fprintf(stderr, "init_delta: malloc error\n");
        exit(1);
    }
    if ((flags = malloc(n_points * sizeof(int))) == NULL) {
        fprintf(stderr, "init_delta: malloc error\n");
        exit(1);
    }
    memset(flags, 0, n_points * sizeof(int));
}

int gather_neighbors(s_point *p, int n) {
    int d, j;

    for (d = -6; d <= 6; d++) {
        if ((j = neighbor_pd(p, d)) < 0)
            continue;

        if (!flags[j]) {
            flags[j] = 1;
            indices[n++] = j;
        }
    }
    return n;
}

double delta(double threshold) {
    s_move move;
    s_point o;
    double diff;
    int n_matches_;
    int k, n;
    int i, s, t;

    do {
        move.i = random() % n_points;
        move.d = random() % 13 - 6;
    } while (!test(&move));

    /* determine range [s, t] of indices to bases to be moved */
    s = move.i;
    t = move.i_;
    if (s > t) {
        i = s;
        s = t;
        t = i;
    }
    if (--s < 0)    /* extend by 1 on each end to account for stacking */
        s = 0;
    if (++t > n_points - 1)
        t = n_points - 1;

    /* gather indices to affected bases (bases to be moved and neighbors) */
    n = 0;
    for (i = s; i <= t; i++) {
        if (!flags[i]) {
            flags[i] = 1;
            indices[n++] = i;
        }
        n = gather_neighbors(&points[i], n);
    }
    walk(&points[move.i], move.d, &o);
    n = gather_neighbors(&o, n);

    /* calculate difference in total score before and after pull */
    n_matches_ = n_matches;
    diff = 0;
    if (pairs) {    /* for reconstruct */
        for (i = s; i <= t; i++)
            diff -= turn_score_i(i);
        for (k = 0; k < n; k++) {
            i = indices[k];
            n_matches -= match_count_i(i);
        }
    } else    /* for predict */
```

```
        for (k = 0; k < n; k++) {
            i = indices[k];
            diff -= stack_score_i(i);
        }

    pull(&move);

    if (pairs) {    /* for reconstruct */
        for (i = s; i <= t; i++)
            diff += turn_score_i(i);
        for (k = 0; k < n; k++) {
            i = indices[k];
            n_matches += match_count_i(i);
            flags[i] = 0;
        }
        diff += (n_matches - n_matches_) * MATCH_BONUS;
    } else    /* for predict */
        for (k = 0; k < n; k++) {
            i = indices[k];
            diff += stack_score_i(i);
            flags[i] = 0;
        }

    if (diff < threshold) {    /* undo */
        undo(&move);
        diff = 0.0;
        n_matches = n_matches_;
    } else {    /* commit */
        if (movie_mode)
            printf("%d %d\n", move.i, move.d);
    }
    return diff;
}

/* simulated annealing: anneal */

volatile int anneal_interrupted = 0;    /* flag set on ctrl-c */

void anneal_interrupt(int signum) {
    anneal_interrupted = 1;
}

int anneal_repeats = 5;
int anneal_steps = 0;

int anneal_doubling = 0;
int anneal_rounds = 0;

int early_terminate = 0;    /* for reconstruct */
int reconstructed = 0;    /* for reconstruct */

void anneal() {
    double c = -M_LN2 / log(0.1);
    double improvement = 1.02;
    double increment = 1e-3;
    double best_best, best_best_;    /* best of all rounds */
    int repeat, step;

    if (anneal_steps == 0)
        anneal_steps = n_points * n_points * 100;
    best_best = increment;
    init_delta();
    turns_to_points();

    signal(SIGINT, anneal_interrupt);

anneal_start:
    anneal_rounds++;
    best_best_ = best_best;

    for (repeat = 1; repeat <= anneal_repeats; repeat++) {
        double best = 0.0;    /* best of current repeat */
        double score = total_score();

        if (verbose)
            fprintf(stderr, "Repeat %d\n", repeat);

        for (step = 1; step <= anneal_steps; step++) {
            double T, threshold;
            int step_mix;

            if (anneal_interrupted)
                goto anneal_end;

            if ((double) random() / RAND_MAX < 0.5)
                step_mix = random() % anneal_steps + 1;
            else
                step_mix = step;
            T = c / log(1.0 + (double) step_mix / anneal_steps);
            threshold = T * log((double) random() / RAND_MAX);

            score += delta(threshold);

            if (score > best + increment) {
                best = score;

                if (verbose)
                    fprintf(stderr, "%f  %.2f\n",
                            (double) step / anneal_steps, score);

                if (best > best_best + increment) {
                    best_best = best;
                    points_to_turns();    /* save the record */

                    if (pairs)    /* for reconstruct */
                        if (n_matches >= n_pairs * 2) {
                            reconstructed = 1;
                            if (early_terminate) {
                                anneal_repeats = repeat;
                                goto anneal_end;
                            }
                        }
                }
            }
        }

        if (verbose)
            fprintf(stderr, "Repeat %d: %.2f %.2f\n", repeat, best, best_best);
    }

    if (anneal_doubling && best_best / best_best_ > improvement) {
        anneal_steps *= 2;
        goto anneal_start;
    }

anneal_end:
    turns_to_points();
}

/* main */

void print_help_and_exit() {
    printf("FOLD\n");
    printf("Options for input/output:\n"
        "  -i  <file>  read bases and turns from file\n"
        "  -i2 <file>  read base pairs from file (for reconstruct)\n"
        "  -o  <file>  write bases and turns to file\n"
        "  -o2 <file>  write base pairs to file\n"
        "  -movie      write movie to stdout\n"
        "  -v          write verbose messages to stderr\n");
    printf("Options for simulation:\n"
        "  -a repeats [steps]  perform simulated annealing\n"
        "  -d                  iterate rounds with doubling steps\n"
        "  -e                  early terminate (for reconstruct)\n"
        "  -2                  use 2-dimensional lattice\n"
        "  -s <seed>           initialize random number generator with seed\n");
    exit(0);
}

int main(int argc, char* argv[]) {
    char *input_filename = NULL;
    char *input2_filename = NULL;
    char *output_filename = NULL;
    char *output2_filename = NULL;
    unsigned long seed = -1;
    int i;

    for (i = 1; i < argc; i++)
        if (!strcmp(argv[i], "-i")) {
            if (i + 1 < argc && argv[i + 1][0] != '-')
                input_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-i2")) {
            if (i + 1 < argc && argv[i + 1][0] != '-')
```

```
                input2_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-o")) {
            if (i + 1 < argc && argv[i + 1][0] != '-')
                output_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-o2")) {
            if (i + 1 < argc && argv[i + 1][0] != '-')
                output2_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-movie")) {
            movie_mode = 1;
        } else if (!strcmp(argv[i], "-v")) {
            verbose = 1;
        } else if (!strcmp(argv[i], "-a")) {
            if (i + 1 < argc && argv[i + 1][0] != '-') {
                anneal_repeats = atoi(argv[++i]);
                if (i + 1 < argc && argv[i + 1][0] != '-')
                    anneal_steps = atoi(argv[++i]);
            } else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-d")) {
            anneal_doubling = 1;
        } else if (!strcmp(argv[i], "-e")) {
            early_terminate = 1;
        } else if (!strcmp(argv[i], "-2")) {
            lattice2();
        } else if (!strcmp(argv[i], "-s")) {
            if (i + 1 < argc && argv[i + 1][0] != '-')
                seed = atoi(argv[++i]);
            else
                print_help_and_exit();
        }

    if (seed == -1) {
        struct timeval tv;

        gettimeofday(&tv, NULL);
        seed = (unsigned long) tv.tv_sec;
    }
    srandom(seed);

    if (input_filename)
        input_bases_turns(input_filename);
    if (n_points <= 0)
        print_help_and_exit();
    if (input2_filename)
        input_pairs(input2_filename);

    if (verbose)
        fprintf(stderr, "bases: %s\nturns: %s\n", bases, turns);
    anneal();
    if (verbose)
        fprintf(stderr, "bases: %s\nturns: %s\n", bases, turns);

    if (output_filename)
        output_bases_turns(output_filename);
    if (output2_filename)
        output_pairs(output2_filename);

    if (movie_mode)
        return 0;

    if (pairs)    /* for reconstruct */
        printf("%s %d %d %d\n",
                input_filename, n_points, reconstructed, anneal_repeats);
    else    /* for predict */
        printf("%s %d %g\n",
                input_filename, n_points, total_score());
    return 0;
}
```

D .4 show.c

```
/*
 *    show.c - RNA structure visualization
 *
 *    Minghui Jiang, Martin Mayne, and Joel Gillespie
 *    Tue Feb 24 10:18:43 MST 2009
 */

#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include "delta.h"
#ifdef __APPLE__
#include <GLUT/glut.h>    /* MacOSX */
#else
#include <GL/glut.h>    /* Cygwin/Linux */
#endif

int movie_mode = 0;

int animate_on = 1;    /* reset_view, shift_view, display_scene/pulse */
int drift_on = 0;    /* random_view */

int bonds_on = 1;
int vector_on = 0;

/* input/output */

char *input_filename = NULL;
char *input2_filename = NULL;
char *output_filename = NULL;

void read_files() {
    if (input_filename)
        input_bases_turns(input_filename);
    if (input2_filename)
        input_pairs(input2_filename);
}

void write_file() {
    if (output_filename)
        output_bases_turns(output_filename);
}

/* vec-math, coordinates, rotations, quaternions */

typedef float (coord_t)[3];

coord_t *coords;

void points_to_coords() {
    int i;

    for (i = 0; i < n_points; i++) {    /* from lattice to Cartesian */
        coords[i][0] = points[i].x - points[i].z / 2.0;
        coords[i][1] = points[i].y - points[i].z / 2.0;
        coords[i][2] = points[i].z * M_SQRT1_2;
    }
}

float dist(coord_t p, coord_t q) {
    float x = p[0] - q[0];
    float y = p[1] - q[1];
    float z = p[2] - q[2];

    return sqrt(x * x + y * y + z * z);
}

typedef float (rotate_t)[4];
typedef float (quat_t)[4];

float r2d(float r) {    /* radian to degree */
    return r * 180.0 / M_PI;
}

float d2r(float d) {    /* degree to radian */
    return d * M_PI / 180.0;
}

void rotate_to_quat(rotate_t r, quat_t q) {
    float angle = d2r(r[0] * 0.5);
    float cos_ = cos(angle);
    float sin_ = sin(angle);

    q[0] = cos_;
    q[1] = r[1] * sin_;
    q[2] = r[2] * sin_;
    q[3] = r[3] * sin_;
}

void quat_to_rotate(quat_t q, rotate_t r) {
    float angle = acos(q[0]);
    float sin_ = sin(angle);

    r[0] = r2d(angle * 2.0);
    r[1] = q[1] / sin_;
    r[2] = q[2] / sin_;
    r[3] = q[3] / sin_;
}

void quat_multiply(quat_t p, quat_t q, quat_t o) {    /* p times q = o */
    o[0] = p[0] * q[0] - p[1] * q[1] - p[2] * q[2] - p[3] * q[3];
    o[1] = p[0] * q[1] + p[1] * q[0] + p[2] * q[3] - p[3] * q[2];
    o[2] = p[0] * q[2] - p[1] * q[3] + p[2] * q[0] + p[3] * q[1];
    o[3] = p[0] * q[3] + p[1] * q[2] - p[2] * q[1] + p[3] * q[0];
}
```

```
/* colors */

typedef float (color_t)[4];

const color_t COLORS[7] = {
    {0.0, 0.0, 0.0, 1.0},    /* BLACK */
    {1.0, 1.0, 1.0, 1.0},    /* WHITE */
    {0.4, 0.4, 0.4, 1.0},    /* GRAY  */
    {1.0, 0.0, 0.0, 1.0},    /* RED    */
    {1.0, 1.0, 0.0, 1.0},    /* YELLOW */
    {0.0, 1.0, 0.0, 1.0},    /* GREEN  */
    {0.0, 0.0, 1.0, 1.0}     /* BLUE   */
};

#define BLACK   0
#define WHITE   1
#define GRAY    2
#define RED     3
#define YELLOW  4
#define GREEN   5
#define BLUE    6

int *icolors;

void bases_to_icolors() {
    int i;

    for (i = 0; i < n_points; i++)
        switch (bases[i]) {
        case 'A':
            icolors[i] = RED;
            break;
        case 'C':
            icolors[i] = YELLOW;
            break;
        case 'G':
            icolors[i] = GREEN;
            break;
        case 'U':
            icolors[i] = BLUE;
            break;
        default:
            icolors[i] = BLACK;
        }
}

/* execute */

#define MAX_MOVES 128
s_move moves[MAX_MOVES];

int i_moves = 0;
int n_redos = 0;
int n_undos = 0;
int dir = 1;
int idx = 0;

void execute_pull() {
    s_move move;

    if (movie_mode || drift_on)
        return;

    move.i = idx;
    move.d = dir;

    if (test(&move)) {
        pull(&move);
        points_to_coords();
        points_to_turns();
        moves[i_moves] = move;
        if (++i_moves == MAX_MOVES)
            i_moves = 0;

        n_redos = 0;
        if (++n_undos > MAX_MOVES)
                n_undos = MAX_MOVES;
    }
}

void execute_redo() {
    if (movie_mode || drift_on)
        return;

    if (n_redos) {
        s_move move;

        move = moves[i_moves];
        if (++i_moves == MAX_MOVES)
            i_moves = 0;

        pull(&move);
        points_to_coords();
        points_to_turns();

        n_redos--;
        n_undos++;
    }
}

void execute_undo() {
    if (movie_mode || drift_on)
        return;

    if (n_undos) {
        s_move move;

        if (--i_moves < 0)
```

```
                i_moves = MAX_MOVES - 1;
            move = moves[i_moves];

            undo(&move);
            points_to_coords();
            points_to_turns();

            n_redos++;
            n_undos--;
        }
    }

    /* timer */

    struct timeval start_time;
    float duration;    /* 1.0 = 1/4 sec */

    void set_timer(float d) {
        gettimeofday(&start_time, NULL);
        duration = d;
    }

    float timer_progress() {
        struct timeval tv;
        float delta;

        gettimeofday(&tv, NULL);
        delta = tv.tv_sec - start_time.tv_sec
            + (tv.tv_usec - start_time.tv_usec) / 1000000.0;    /* secs */
        delta *= 4.0;    /* convert to 1/4 secs */
        return delta / duration;
    }

    /* view */

    typedef struct {
        coord_t c;
        rotate_t r;
        float zoom;
    } s_view;

    s_view source, target, current;

    coord_t center;
    float circumradius;
    float zoom_near, zoom_far;

    int same_view(s_view *v1, s_view *v2) {
        return !memcmp(v1, v2, sizeof(s_view));
    }

    void reset_view() {
        int i;

        if (!same_view(&source, &target))
            return;

        vector_on = 0;

        center[0] = center[1] = center[2] = 0.0;
        for (i = 0; i < n_points; i++) {
            center[0] += coords[i][0];
            center[1] += coords[i][1];
            center[2] += coords[i][2];
        }
        center[0] /= n_points;
        center[1] /= n_points;
        center[2] /= n_points;

        circumradius = 0.0;
        for (i = 0; i < n_points; i++) {
            float d = dist(coords[i], center);

            if (d > circumradius)
                circumradius = d;
        }

        zoom_near = 2.0;
        zoom_far = circumradius * 6.0;

        target.c[0] = center[0];
        target.c[1] = center[1];
        target.c[2] = center[2];
        target.r[0] = 45.0;
        target.r[1] = 1.0;
        target.r[2] = 0.0;
        target.r[3] = 0.0;
        target.zoom = (zoom_near + zoom_far) / 2.0;

        if (animate_on)
            set_timer(4);    /* 1 sec */
        else
            current = source = target;
    }

    void shift_view(int offset) {
        if (!same_view(&source, &target))
            return;

        idx += offset;
        if (idx > n_points - 1)
            idx = 0;
        else if (idx < 0)
            idx = n_points - 1;
        target.c[0] = coords[idx][0];
        target.c[1] = coords[idx][1];
        target.c[2] = coords[idx][2];

        if (animate_on)
            set_timer(2);    /* 0.5 sec */
        else
            current = source = target;
    }

    void random_view() {
        float v;
        int i;

        if (!same_view(&source, &target))
            return;

        i = random() % n_points;
        target.c[0] = coords[i][0];
        target.c[1] = coords[i][1];
        target.c[2] = coords[i][2];
        target.r[0] = (float) random() / RAND_MAX * 360;

        target.r[1] = (float) random() / RAND_MAX * 2.0 - 1.0;
        target.r[2] = (float) random() / RAND_MAX * 2.0 - 1.0;
        target.r[3] = (float) random() / RAND_MAX * 2.0 - 1.0;
        v = sqrt( target.r[1] * target.r[1]
                + target.r[2] * target.r[2]
                + target.r[3] * target.r[3]);
        target.r[1] /= v;
        target.r[2] /= v;
        target.r[3] /= v;

        target.zoom = zoom_near
            + (float) random() / RAND_MAX * (zoom_far - zoom_near) * 0.5;
        set_timer(20);    /* 5 sec */
    }

    void interpolate_view() {
        float delta = timer_progress();

        if (delta < 1.0) {
            int i;

            for (i = 0; i < 3; i++)
                current.c[i] = source.c[i] * (1.0 - delta) + target.c[i] * delta;
            for (i = 0; i < 4; i++)
                current.r[i] = source.r[i] * (1.0 - delta) + target.r[i] * delta;
            current.zoom = source.zoom * (1.0 - delta) + target.zoom * delta;
        } else
            current = source = target;
    }

    void rotate_view(rotate_t r) {
        rotate_t q1, q2, q;
        int i;

        rotate_to_quat(r, q1);
        rotate_to_quat(current.r, q2);
        quat_multiply(q1, q2, q);
        quat_to_rotate(q, current.r);
        for (i = 0; i < 4; i++)
            source.r[i] = target.r[i] = current.r[i];
    }

    void zoom_to(float zoom) {
        if (zoom < zoom_near)
            zoom = zoom_near;
        if (zoom > zoom_far)
            zoom = zoom_far;
        source.zoom = target.zoom = current.zoom = zoom;
    }

    void zoom_in() {
        zoom_to(current.zoom - (zoom_far - zoom_near) / 20);
    }

    void zoom_out() {
        zoom_to(current.zoom + (zoom_far - zoom_near) / 20);
    }

    /* movie */

    int movie_paused = 0;

    void movie_pause() {
        if (movie_mode)
            movie_paused = !movie_paused;
    }

    int movie_delay = 5;    /* (1 << 5) = 32 msec */

    void movie_faster() {
        if (movie_mode)
            if (--movie_delay < 0)
                movie_delay = 0;    /* (1 << 0) = 1 msec  */
    }

    void movie_slower() {
        if (movie_mode)
            if (++movie_delay > 11)
                movie_delay = 11;    /* (1 << 11) = 2048 msec; about 2 seconds */
    }

    /* display */

    int width;
    int height;

    #define _B_    256
    #define _R_    5

    void display_console() {
        static char buffer[_B_];
        static char s[_R_ + 3 + _R_ + 1];
        int i;
        s[_R_ + 3 + _R_] = ’\0’;
        s[_R_] = ’<’;
        s[_R_ + 1] = bases[idx];
        s[_R_ + 2] = ’>’;
        for (i = 0; i < _R_; i++)
            s[_R_ + 3 + i] = (idx + i + 1 < n_points ? bases[idx + i + 1] : ’ ’);
        for (i = 0; i < _R_; i++)
            s[_R_ - 1 - i] = (idx - i - 1 >= 0 ? bases[idx - i - 1] : ’ ’);
        snprintf(buffer, _B_, " %s"
            "     %s"
            "   %4d/%-4d"
            "   %c%c"
            "     animate%c  drift%c  bonds%c  vector%c  movie%c",
            input_filename,
            s,
            idx + 1, n_points,
            dir > 0 ? ’+’ : ’-’, toupper(axis_i2c(dir)),
            animate_on ? ’+’ : ’-’,
            drift_on ? ’+’ : ’-’,
            bonds_on ? ’+’ : ’-’,
            vector_on ? ’+’ : ’-’,
            movie_mode ? (!movie_paused ? ’+’ : ’-’) : ’ ’);

        glMaterialfv(GL_FRONT, GL_AMBIENT, COLORS[BLACK]);
        glRasterPos2f(1, 5);
        for (i = 0; i < strlen(buffer); i++)
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, buffer[i]);
    }

    const rotate_t TURN_ROTATIONS[13] = {
        {135.0, M_SQRT1_2,-M_SQRT1_2, 0.0},        /* w */
        {135.0, M_SQRT1_2, M_SQRT1_2, 0.0},        /* v */
        {135.0,-M_SQRT1_2,-M_SQRT1_2, 0.0},        /* u */
        {135.0,-M_SQRT1_2, M_SQRT1_2, 0.0},        /* z */
        { 90.0,       1.0,       0.0, 0.0},        /* y */
        {-90.0,       0.0,       1.0, 0.0},        /* x */
        {  0.0,       0.0,       0.0, 0.0},     /* NOT USED */
        { 90.0,       0.0,       1.0, 0.0},        /* X */
        {-90.0,       1.0,       0.0, 0.0},        /* Y */
        { 45.0, M_SQRT1_2,-M_SQRT1_2, 0.0},        /* Z */
        { 45.0, M_SQRT1_2, M_SQRT1_2, 0.0},        /* U */
        { 45.0,-M_SQRT1_2,-M_SQRT1_2, 0.0},        /* V */
        { 45.0,-M_SQRT1_2, M_SQRT1_2, 0.0}        /* W */
    };
    const rotate_t *turn_rotations = &(TURN_ROTATIONS[6]);

    #define TURN(i)    glRotatef(turn_rotations[(i)][0], \
            turn_rotations[(i)][1], turn_rotations[(i)][2], turn_rotations[(i)][3])

    #define RADIUS_BALL    0.12
    #define RADIUS_STICK    0.04

    float LIGHT[] = { 0.0, 0.0, 1.0, 0.0 };    /* from +z */
```

```
void switch_light(int on) {
    if (on) {
        glLightfv(GL_LIGHT0, GL_AMBIENT, COLORS[GRAY]);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, COLORS[GRAY]);
        glLightfv(GL_LIGHT0, GL_SPECULAR, COLORS[WHITE]);
    } else {
        glLightfv(GL_LIGHT0, GL_AMBIENT, COLORS[BLACK]);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, COLORS[BLACK]);
        glLightfv(GL_LIGHT0, GL_SPECULAR, COLORS[BLACK]);
    }
}

void display_scene() {
    static GLUquadricObj *quadric = NULL;
    int i, j, d;

    if (quadric == NULL) {
        quadric = gluNewQuadric();
        gluQuadricNormals(quadric, GLU_SMOOTH);
        gluQuadricDrawStyle(quadric, GLU_FILL);
    }

    glTranslatef(0.0, 0.0, -current.zoom);
    glRotatef(current.r[0], current.r[1], current.r[2], current.r[3]);
    glTranslatef(-current.c[0], -current.c[1], -current.c[2]);

    if (bonds_on) {
        switch_light(0);
        for (i = 0; i < n_points; i++)
            for (d = -6; d <= 6; d++) {
                if ((j = neighbor_id(i, d)) < i)
                    continue;

                if ((pairs && !pairs[i * n_points + j])
                        || (!pairs && pair_ij(i, j) < 0))
                    continue;

                glBegin(GL_LINES);
                glVertex3fv(coords[i]);
                glVertex3fv(coords[j]);
                glEnd();
            }
        switch_light(1);
    }

    /* shared material property of sticks, balls, and direction vector */
    glMaterialfv(GL_FRONT, GL_DIFFUSE, COLORS[GRAY]);
    glMaterialfv(GL_FRONT, GL_SPECULAR, COLORS[WHITE]);
    glMaterialf(GL_FRONT, GL_SHININESS, 100.0);

    /* draw sticks */
    glMaterialfv(GL_FRONT, GL_AMBIENT, COLORS[WHITE]);
    for (i = 0; i < n_points - 1; i++) {
        glPushMatrix();
        glTranslatef(coords[i][0], coords[i][1], coords[i][2]);
        TURN(axis_c2i(turns[i]));
        gluCylinder(quadric, RADIUS_STICK, RADIUS_STICK, 1.0, 32, 1);
        glPopMatrix();
    }

    /* draw balls */
    for (i = 0; i < n_points; i++) {
        glPushMatrix();
        glTranslatef(coords[i][0], coords[i][1], coords[i][2]);
        glMaterialfv(GL_FRONT, GL_AMBIENT, COLORS[icolors[i]]);
        if (i == idx && animate_on) {    /* pulse */
            struct timeval tv;
            float theta, scale;

            gettimeofday(&tv, NULL);
            theta = (tv.tv_usec / 1000000.0) * M_PI * 2.0;    /* period: 1 sec */
            scale = 1.0 + 0.2 * (1.0 + sin(theta));    /* range: [1.0, 1.4] */

            glutSolidSphere(RADIUS_BALL * scale, 32, 16);
        } else
            glutSolidSphere(RADIUS_BALL, 32, 16);
        glPopMatrix();
    }

    /* draw vector */
    if (vector_on) {
        glTranslatef(coords[idx][0], coords[idx][1], coords[idx][2]);
        TURN(dir);

        glMaterialfv(GL_FRONT, GL_AMBIENT, COLORS[WHITE]);
        gluCylinder(quadric, RADIUS_STICK, RADIUS_STICK, 1, 32, 1);

        glTranslatef(0.0, 0.0, 1.0);
        gluCylinder(quadric, RADIUS_BALL, 0.0, 0.3, 32, 1);
    }
}

int redisplay_scheduled = 0;

void schedule_redisplay() {
    if (!redisplay_scheduled) {
        redisplay_scheduled = 1;
        glutPostRedisplay();
    }
}

int movie_busy = 0;

void movie_advance(int value) {
    movie_busy = 0;
    schedule_redisplay();
}

void display() {
    float ratio = (float) width / height;    /* aspect ratio */
    int need_redisplay = 0;

    glClear(GL_COLOR_BUFFER_BIT);
    glClear(GL_DEPTH_BUFFER_BIT);

    glViewport(0, 0, width, 20);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, width, 0, 20);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    display_console();

    glViewport(0, 20, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (ratio < 1.0)
        glFrustum(-0.5, 0.5, -0.5 / ratio, 0.5 / ratio, 1.0, circumradius * 8);
    else
        glFrustum(-0.5 * ratio, 0.5 * ratio, -0.5, 0.5, 1.0, circumradius * 8);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    if (!same_view(&source, &target))
        interpolate_view();
    display_scene();

    glutSwapBuffers();

    if (animate_on)
        need_redisplay = 1;
    if (drift_on) {
        random_view();
        need_redisplay = 1;
    }
    if (movie_mode && !movie_paused && !movie_busy) {
        s_move move;

        movie_busy = 1;
        if (scanf("%d %d\n", &move.i, &move.d) != EOF) {
            if (test(&move)) {
                pull(&move);
                points_to_coords();
                points_to_turns();
            }
            glutTimerFunc(1 << movie_delay, movie_advance, 0);
        }
        need_redisplay = 1;
    }

    redisplay_scheduled = 0;
    if (need_redisplay)
        schedule_redisplay();
}

void reshape(int w, int h) {
    width = w;
    height = h;
}

/* keyboard, special, mouse, motion */

void keyboard(unsigned char key, int x, int y) {
    switch (key) {

    /* execute */
    case 'p':
        execute_pull();
        break;
    case 'r':
        execute_redo();
        break;
    case 'u':
        execute_undo();
        break;

    /* movie */
    case 'm':
        movie_pause();
        break;
    case 'f':
        movie_faster();
        break;
    case 's':
        movie_slower();
        break;

    /* toggle */
    case 'a':
        animate_on = !animate_on;
        break;
    case 'd':
        drift_on = !drift_on;
        if (!drift_on)
            source = target = current;
        break;
    case 'b':
        bonds_on = !bonds_on;
        break;
    case 'v':
        vector_on = !vector_on;
        break;

    /* view */
    case 'i':
        zoom_in();
        break;
    case 'o':
        zoom_out();
        break;
    case ' ':
        reset_view();
        break;

    /* top */
    case 'w':
        write_file();
        break;
    case 27:    /* <esc> */
        exit(0);
    }
    schedule_redisplay();
}

void cycle_vector(int offset) {
    dir += offset;
    if (dir == 0)
        dir += offset;
    else if (dir < -6)
        dir = 6;
    else if (dir > 6)
        dir = -6;
    vector_on = 1;
}

void special(int key, int x, int y) {
    switch (key) {
    case GLUT_KEY_LEFT:
        shift_view(-1);
        break;
    case GLUT_KEY_RIGHT:
        shift_view(1);
        break;
    case GLUT_KEY_UP:
        cycle_vector(-1);
        break;
    case GLUT_KEY_DOWN:
        cycle_vector(1);
        break;
    }
    schedule_redisplay();
}

int key_modifiers = 0;
int mouse_x, mouse_y;

void mouse(int button, int state, int x, int y) {
    mouse_x = x;
    mouse_y = y;
    key_modifiers = (state == GLUT_DOWN ? glutGetModifiers() : 0);
}

void motion(int x, int y) {
    int dx = x - mouse_x;
    int dy = y - mouse_y;
    mouse_x = x;
    mouse_y = y;

    if (dx == 0 && dy == 0)
        return;

    if (!same_view(&source, &target))
        return;

    if (!(key_modifiers & GLUT_ACTIVE_SHIFT)) {    /* x and y rotation */
        float d = sqrt(dx * dx + dy * dy);
        rotate_t r;

        r[0] = d;    /* amount */
        r[1] = dy / d;
        r[2] = dx / d;
        r[3] = 0.0;    /* perpendicular to (dx, dy) direction */
        rotate_view(r);
    } else if (abs(dx) > abs(dy)) {    /* z rotation */
        rotate_t r;

        r[0] = dx;    /* amount */
        r[1] = 0.0;
        r[2] = 0.0;
        r[3] = 1.0;    /* in +z direction */
        rotate_view(r);
    } else
        zoom_to(current.zoom + dy / 20.0);
    schedule_redisplay();
}

/* menu */

void menu_top(int id) {
    switch (id) {
    case 1:
        write_file();
        break;
    case 2:
        exit(0);
        break;
    }
    schedule_redisplay();
}

void menu_execute(int id) {
    switch (id) {
    case 1:
        shift_view(-1);
        break;
    case 2:
        shift_view(1);
        break;
    case 3:
        cycle_vector(-1);
        break;
    case 4:
        cycle_vector(1);
        break;
    case 5:
        execute_pull();
        break;
    case 6:
        execute_redo();
        break;
    case 7:
        execute_undo();
        break;
    }
    schedule_redisplay();
}

void menu_movie(int id) {
    switch (id) {
    case 1:
        movie_pause();
        break;
    case 2:
        movie_faster();
        break;
    case 3:
        movie_slower();
        break;
    }
    schedule_redisplay();
}

void menu_toggle(int id) {
    switch (id) {
    case 1:
        animate_on = !animate_on;
        break;
    case 2:
        drift_on = !drift_on;
        if (!drift_on)
            source = target = current;
        break;
    case 3:
        bonds_on = !bonds_on;
        break;
    case 4:
        vector_on = !vector_on;
        break;
    }
    schedule_redisplay();
}

void menu_view(int id) {
    switch (id) {
    case 1:
        zoom_in();
        break;
    case 2:
        zoom_out();
        break;
    case 3:
        reset_view();
        break;
    }
    schedule_redisplay();
}

/* main */

void print_help_and_exit() {
    printf("SHOW\n"
        "Options:\n"
        "  -i  <file>  read bases and turns from file\n"
        "  -i2 <file>  read base pairs from file\n"
        "  -o  <file>  write bases and turns to file\n"
        "  -movie      turn on movie mode\n");
    exit(0);
}

int main(int argc, char *argv[]) {
    int m_execute, m_movie, m_toggle, m_view;
    int i;

    for (i = 1; i < argc; i++)
        if (!strcmp(argv[i], "-movie")) {
            movie_mode = 1;
            drift_on = 1;
        } else if (!strcmp(argv[i], "-i")) {
            if (i + 1 < argc && argv[i + 1][0] != ’-’)
                input_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-o")) {
            if (i + 1 < argc && argv[i + 1][0] != ’-’)
                output_filename = argv[++i];
            else
                print_help_and_exit();
        } else if (!strcmp(argv[i], "-i2")) {
            if (i + 1 < argc && argv[i + 1][0] != ’-’)
                input2_filename = argv[++i];
            else
                print_help_and_exit();
        }

    read_files();
    if (n_points == 0)
        print_help_and_exit();

    if ((coords = malloc(n_points * sizeof(coord_t))) == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    turns_to_points();
    points_to_coords();

    if ((icolors = malloc(n_points * sizeof(int))) == NULL) {
        fprintf(stderr, "malloc error\n");
        exit(1);
    }
    bases_to_icolors();

    reset_view();

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 800);
    glutCreateWindow("SHOW");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(special);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);

    glLightfv(GL_LIGHT0, GL_POSITION, LIGHT);
    switch_light(1);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint(GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glShadeModel(GL_SMOOTH);
    glClearColor(1.0, 1.0, 1.0, 0.0);    /* white background */
    glColor3f(0.0, 0.0, 0.0);    /* black bonds */

    m_execute = glutCreateMenu(menu_execute);
    glutAddMenuEntry("i.prev - left", 1);
    glutAddMenuEntry("i.next - right", 2);
    glutAddMenuEntry("d.prev - up", 3);
    glutAddMenuEntry("d.next - down", 4);
    glutAddMenuEntry("Pull - p", 5);
    glutAddMenuEntry("Redo - r", 6);
    glutAddMenuEntry("Undo - u", 7);

    m_movie = glutCreateMenu(menu_movie);
    glutAddMenuEntry("Pause - m", 1);
    glutAddMenuEntry("Faster - f", 2);
    glutAddMenuEntry("Slower - s", 3);

    m_toggle = glutCreateMenu(menu_toggle);
    glutAddMenuEntry("Animate - a", 1);
    glutAddMenuEntry("Drift - d", 2);
    glutAddMenuEntry("Bonds - b", 3);
    glutAddMenuEntry("Vector - v", 4);

    m_view = glutCreateMenu(menu_view);
    glutAddMenuEntry("Zoom In - i", 1);
    glutAddMenuEntry("Zoom Out - o", 2);
    glutAddMenuEntry("Reset - space", 3);

    glutCreateMenu(menu_top);
    glutAddSubMenu("Execute", m_execute);
    if (movie_mode)
        glutAddSubMenu("Movie", m_movie);
    glutAddSubMenu("Toggle", m_toggle);
    glutAddSubMenu("View", m_view);
    glutAddMenuEntry("Write - w", 1);
    glutAddMenuEntry("Quit - esc", 2);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    glutMainLoop();
    return 0;
}
```

D .5 is.c

```
/*
 *    is.c - maximum weight independent set of RNA helices in two pages
 *
 *    Joel Gillespie and Minghui Jiang
 *    Wed Jan 28 14:10:55 MST 2009
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "delta.h"    /* energies, pair_ij */

#define MAXSEQ    256    /* maximum length of the sequence of bases */
#define MAXHLS    64    /* maximum number of helices */

#define MINDST    4    /* minimum distance between two indices of a base pair */
#define MINLEN    3    /* minimum length of a helix */

int n_bases = 0;

typedef struct {
    int valid;
    int color;    /* colors 1 and 2 for two-page structure */
    int i;
    int j;
    int length;
    double energy;
} s_helix;

s_helix helices[MAXHLS];
int n_helices = 0;

int valid_helix(int i, int j, int length) {
    return i >= 0 && j < n_bases && length >= 1
        && j - length + 1 >= i + length - 1 + MINDST;
}

double energy(int i, int j, int length) {
    double e = 0.0;

    while (length >= 2) {
        int i_ = pair_ij(i, j);
        int j_ = pair_ij(i + 1, j - 1);

        if (i_ >= 0 && j_ >= 0)
            e += energies[i_][j_];
        i++;
        j--;
        length--;
    }
    return e;
}
```

```
void add_helix(int i, int j, int length, double energy) {
    int k = n_helices++;

    helices[k].valid = 1;
    helices[k].color = 0;
    helices[k].i = i;
    helices[k].j = j;
    helices[k].length = length;
    helices[k].energy = energy;
}

void input_helix(int i, int j, int length) {
    double e, e_;
    int k;

    if (i > j) {
        k = i;
        i = j;
        j = k;
    }
    if (valid_helix(i, j, length) && (e = energy(i, j, length)) < 0.0) {
        while (valid_helix(i, j, length + 1)
                && (e_ = energy(i, j, length + 1)) < e) {
            /* extend inside */
            length++;
            e = e_;
        }
        while (valid_helix(i - 1, j + 1, length + 1)
                && (e_ = energy(i - 1, j + 1, length + 1)) < e) {
            /* extend outside */
            i--;
            j++;
            length++;
            e = e_;
        }
        if (length >= MINLEN)
            add_helix(i, j, length, e);
    }
}

int cross(s_helix *u, s_helix *v) {    /* assume that u and v are disjoint! */
    return (u->i < v->i && v->i < u->j && u->j < v->j)
        || (v->i < u->i && u->i < v->j && v->j < u->j);
}

int trim_ab(s_helix *u, int a, int b) {
    double e;
    int a_ = u->i;
    int b_ = u->i + u->length - 1;
    int ij = u->i + u->j;
    int i, j, length;

    if (ij < a + b) {    /* center of [a b] on the right of center of u */
        int c;
        /* flip to the left */
        a = ij - a;
        b = ij - b;
        c = a;
        a = b;
        b = c;
    }
    if (a > b_ || b < a_)    /* no overlap */
        return 0;

    if (a >= a_ + MINLEN) {    /* [a_ a) b_ */
        i = a_;
        j = ij - i;
        length = a - a_;
        if (valid_helix(i, j, length) && (e = energy(i, j, length)) < 0.0)
            add_helix(i, j, length, e);
    }
    if (b <= b_ - MINLEN) {    /* a_ (b b_] */
        i = b + 1;
        j = ij - i;
        length = b_ - b;
        if (valid_helix(i, j, length) && (e = energy(i, j, length)) < 0.0)
            add_helix(i, j, length, e);
    }
    return 1;
}

int trim_helix(s_helix *u, s_helix *v) {
    int ia = v->i;
    int ib = v->i + v->length - 1;
    int ja = v->j - v->length + 1;
    int jb = v->j;

    /* use v to trim u, return 1 if u has no remains */
    return trim_ab(u, ia, ib) || trim_ab(u, ja, jb);
}

void is() {
    int k;

    while (1) {
        int conflict[3] = {0, 0, 0};    /* for colors 0, 1, 2 */
        s_helix *v = NULL;
        double e = 0.0;

        /* find candidate v with lowest energy */
        for (k = 0; k < n_helices; k++) {
            s_helix *u = &helices[k];

            if (u->valid && !u->color && u->energy < e) {
                e = u->energy;
                v = u;
            }
        }
        if (v == NULL)
```

```
            break;

        /* assign color to candidate v */
        for (k = 0; k < n_helices; k++) {
            s_helix *u = &helices[k];

            if (u != v && u->valid && u->color
                    && !conflict[u->color] && cross(u, v)) {
                conflict[u->color] = 1;
                if (conflict[1] && conflict[2])
                    break;
            }
        }
        if (!conflict[1])
            v->color = 1;
        else if (!conflict[2])
            v->color = 2;
        else {
            v->valid = 0;
            continue;
        }

        /* trim other helices! */
        for (k = 0; k < n_helices; k++) {
            s_helix *u = &helices[k];

            if (u != v && u->valid && !u->color && trim_helix(u, v))
                u->valid = 0;
        }
    }
}

int main(int argc, char *argv[]) {
    int i, j, length, k;

    if (argc < 2) {
        fprintf(stderr, "usage: cat <hx_input> | %s <seq_file> > <hx_output>\n",
                argv[0]);
        exit(1);
    }
    input_bases_turns(argv[1]);
    n_bases = strlen(bases);

    while (scanf("%d %d %d\n", &i, &j, &length) != EOF)
        input_helix(i - 1, j - 1, length);

    is();

    for (k = 0; k < n_helices; k++) {
        s_helix *u = &helices[k];

        if (u->valid)
            printf("%d %d %d\n", u->i + 1, u->j + 1, u->length);
    }
    return 0;
```

```
}
```

APPENDIX E

FORMAT MANIPULATION SOURCE CODE

E .1 db2bp.c

```
/*
 *     db2bp.c - dot brackets to base pairs
 *
 *    Joel Gillespie and Minghui Jiang
 *    Tue Feb  3 09:11:32 MST 2009
 */

#include <stdio.h>

#define    MAXPAIRS    256

int p[MAXPAIRS];    /* parentheses */
int b[MAXPAIRS];    /* brackets */
int c[MAXPAIRS];    /* curly-braces */

void output(int i, int j) {
    printf("%d %d\n", i + 1, j + 1);
}

int main() {
    int ip = 0;
    int ib = 0;
    int ic = 0;
    int i = 0;

    while (1)
        switch (getchar()) {
        case EOF:
            return 0;
        case ’:’:
            i++;
            break;
        case ’(’:
            p[ip++] = i++;
            break;
        case ’[’:
            b[ib++] = i++;
            break;
        case ’{’:
            c[ic++] = i++;
            break;
        case ’)’:
            output(p[--ip], i++);
            break;
        case ’]’:
            output(b[--ib], i++);
            break;
        case ’}’:
            output(c[--ic], i++);
            break;
        }
    return 0;
}
```

E .2 hx2bp.c

```
/*
 *    hx2bp.c - helices to base pairs
 *
 *    Joel Gillespie and Minghui Jiang
 *    Mon Jan 12 13:59:57 MST 2009
 */

#include <stdio.h>

int main() {
    int i, j, length;
    int k;

    while (scanf("%d %d %d\n", &i, &j, &length) != EOF)
        for (k = 0; k < length; k++)
            printf("%d %d\n", i++, j--);
    return 0;
}
```

E .3 bp2hx.c

```
/*
 *    bp2hx.c - base pairs to helices
 *
 *    Joel Gillespie and Minghui Jiang
 *    Mon Jan 12 15:48:24 MST 2009
 */

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int used;
    int i;
    int j;
} s_pair;

s_pair *pairs = NULL;
int n_pairs = 0;
int max_pairs = 16;

void insert_pair(int i, int j) {
    int k;

    if (i > j) {
        k = i;
        i = j;
        j = k;
    }
    if (n_pairs == 0) {
        if ((pairs = malloc(max_pairs * sizeof(s_pair))) == NULL) {
            fprintf(stderr, "insert_pair: malloc error\n");
            exit(1);
        }
    } else if (n_pairs == max_pairs) {
        s_pair *temp = pairs;

        max_pairs *= 2;
        if ((pairs = malloc(max_pairs * sizeof(s_pair))) == NULL) {
            fprintf(stderr, "insert_pair: malloc error\n");
            exit(1);
        }
        for (k = 0; k < n_pairs; k++)
            pairs[k] = temp[k];
        free(temp);
    }
    k = n_pairs++;
    while (k > 0
            && (pairs[k - 1].i > i
                || (pairs[k - 1].i == i && pairs[k - 1].j > j))) {
        /* insertion sort */
        pairs[k] = pairs[k - 1];
        k--;
    }
    pairs[k].used = 0;
    pairs[k].i = i;
    pairs[k].j = j;
}

void make_helix(int k) {    /* pairs[k] is the outermost pair */
    int i, j, l, length;

    if (pairs[k].used)
        return;

    pairs[k].used = 1;
    length = 1;
    i = pairs[k].i + 1;
    j = pairs[k].j - 1;
    for (l = k + 1; l < n_pairs; l++) {    /* extends the current helix */
        if (pairs[l].used)
            continue;

        /* find (i, j) */
        if (pairs[l].i < i)
            continue;
        if (pairs[l].i > i)
            break;
        if (pairs[l].j < j)
            continue;
        if (pairs[l].j > j)
            break;

        pairs[l].used = 1;
        length++;
        i++;
        j--;
    }
    printf("%d %d %d\n", pairs[k].i, pairs[k].j, length);
}

int main() {
    int i, j, k;

    while (scanf("%d %d\n", &i, &j) != EOF)
        insert_pair(i, j);
    for (k = 0; k < n_pairs; k++)
        make_helix(k);
    return 0;
}
```

E .4 hx2i.awk

```
#
#    hx2i.awk - helices to 2-intervals
#
#    Minghui Jiang
#    Wed Jan 14 14:48:51 MST 2009
#

{
    for (i = 1; i <= $2; i++)
        if (i >= $1 && i < $1 + $3 || i > $2 - $3 && i <= $2)
            printf("_")
        else
            printf(" ")
    printf("\n")
}
```

E .5 bpseq2bpseq.c

```
/*
 *    bpseq2bpseq.c - base pairs and base sequence to bpseq
 *
 *    Joel Gillespie and Minghui Jiang
 *    Sat Feb  7 08:19:17 MST 2009
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSEQ    256

int pairs[MAXSEQ];
char bases[MAXSEQ];
int n_bases = 0;

int main(int argc, char *argv[]) {
    FILE *file;
    int i, j, k;

    if (argc < 2) {
        fprintf(stderr, "usage: cat <bp_file> | %s <seq_file> > <bpseq_file>\n",
                argv[0]);
        exit(1);
    }
    if ((file = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "fopen(%s) error\n", argv[1]);
        exit(1);
    }
    if (fgets(bases, MAXSEQ, file) == NULL) {
        fprintf(stderr, "fgets error\n");
        exit(1);
    }
    fclose(file);
    n_bases = strlen(bases);
    bases[--n_bases] = ’\0’;  /* overwrite ’\n’ */

    /* [1, n_bases] <-> [0, n_bases-1] */
    for (k = 0; k < n_bases; k++)
        pairs[k] = -1;
    while (scanf("%d %d\n", &i, &j) != EOF)
        if (i >= 1 && i <= n_bases && j >= 1 && j <= n_bases) {
            pairs[i - 1] = j - 1;
            pairs[j - 1] = i - 1;
        } else
            fprintf(stderr, "invalid pair (%d, %d)\n", i, j);
    for (k = 0; k < n_bases; k++)
        printf("%d %c %d\n", k + 1, bases[k], pairs[k] + 1);
    return 0;
}
```

APPENDIX F

EVALUATION SOURCE CODE

F .1 ssa.c

```
/*
 *    ssa.c - sensitivity, selectivity, accuracy
 *
 *    Joel Gillespie and Minghui Jiang
 *    Sat Feb  7 08:21:43 MST 2009
 */

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int i;
    int j;
    int length;
} s_helix;

#define PREDIC 0
#define ANSWER 1

#define    MAXSIZE    512
s_helix helices[2][MAXSIZE];
int n_helices[2] = {0, 0};
int n_pairs[2] = {0, 0};

void input_helices(char *filename, int type) {
    FILE *file;
    int i, j, length;

    if ((file = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "input_helices: fopen(%s) error\n", filename);
        exit(1);
    }
    n_helices[type] = 0;
    while (fscanf(file, "%d %d %d\n", &i, &j, &length) != EOF) {
        int k;

        if (i > j) {
            k = i;
            i = j;
            j = k;
        }

        k = n_helices[type];
        helices[type][k].i = i;
        helices[type][k].j = j;
        helices[type][k].length = length;
        n_helices[type]++;
        n_pairs[type] += length;
    }
    fclose(file);
}

int overlap(int a, int b, int c, int d) {
    int ac = a > c ? a : c;
    int bd = b < d ? b : d;

    return ac <= bd ? bd - ac + 1 : 0;
}

int intersection() {
    int i, j, n_pairs = 0;

    for (i = 0; i < n_helices[PREDIC]; i++)
        for (j = 0; j < n_helices[ANSWER]; j++) {
            s_helix *u = &helices[PREDIC][i];
            s_helix *v = &helices[ANSWER][j];

            if (u->i + u->j == v->i + v->j)    /* aligned with the same center */
                n_pairs += overlap(    /* compare two left intervals */
                        u->i, u->i + u->length - 1,
                        v->i, v->i + v->length - 1);
        }
    return n_pairs;
}

int main(int argc, char *argv[]) {
    int tp, fp, fn;
    double sensitivity, specificity, accuracy;

    if (argc < 3) {
        fprintf(stderr, "usage: %s <prediction_file> <answer_file>\n", argv[0]);
        exit(1);
    }
    input_helices(argv[1], PREDIC);
    input_helices(argv[2], ANSWER);
    tp = intersection();
    fp = n_pairs[PREDIC] - tp;
    fn = n_pairs[ANSWER]- tp;
    sensitivity = (double) tp / (tp + fn);
    specificity = (double) tp / (tp + fp);
    accuracy = (double) tp / (tp + fn + fp);
    printf("%3d %3d %3d %5.2f %5.2f %5.2f",
            tp, fn, fp, sensitivity, specificity, accuracy);
    return 0;
}
```

F .2 stats.c

```c
/*
 *    stats.c - average and standard deviation
 *
 *    Joel Gillespie and Minghui Jiang
 *    Thu Feb 26 14:02:22 MST 2009
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define    N    512

double data[3][N];    /* sensitivity, specificity, accuracy */

int main() {
    double average[3] = {0.0, 0.0, 0.0};
    double stdev[3] = {0.0, 0.0, 0.0};
    int i, k, n = 0;

    while (scanf("%lf %lf %lf\n", &data[0][n], &data[1][n], &data[2][n]) != EOF)
        n++;
    for (k = 0; k < 3; k++) {
        for (i = 0; i < n; i++)
            average[k] += data[k][i];
        average[k] /= n;
        if (n > 1) {
            for (i = 0; i < n; i++) {
                double diff = data[k][i] - average[k];

                stdev[k] += diff * diff;
            }
            stdev[k] = sqrt(stdev[k] / (n - 1));
        } else
            stdev[k] = 0;
        printf("%.4f  %.4f", average[k], stdev[k]);
        printf(k < 2 ? "  " : "\n");
    }
    return 0;
}
```

F .3 linear.c

```
/*
 *    linear.c - linear regression
 *
 *    Joel Gillespie and Minghui Jiang
 *    Sat Feb  7 08:20:35 MST 2009
 */

#include <math.h>
#include <stdio.h>

#define MAXPOINTS 500

double x[MAXPOINTS] = {0.0};
double y[MAXPOINTS] = {0.0};

int main() {
    double ax = 0.0;
    double ay = 0.0;
    double xx = 0.0;
    double xy = 0.0;
    double slope;
    int i, n_points = 0;

    while (scanf("%lf %lf\n", &x[n_points], &y[n_points]) != EOF) {
        ax += x[n_points];
        ay += y[n_points];
        n_points++;
    }
    ax /= n_points;
    ay /= n_points;
    for (i = 0; i < n_points; i++) {
        double diff_x = x[i] - ax;
        double diff_y = y[i] - ay;

        xx += diff_x * diff_x;
        xy += diff_x * diff_y;
    }
    slope = xy / xx;

    /* a and b as in y=ax+b */
    printf("%g %g\n", slope, ay - slope * ax);
    return 0;
}
```

F .4 ssa.awk

```
#
#    ssa.awk - sensitivity, specificity, accuracy
#
#    Joel Gillespie and Minghui Jiang
#    Sat Feb  7 08:18:43 MST 2009
#

{
    l[$1] = $2
    tp[$1] += $3
    fn[$1] += $4
    fp[$1] += $5
}
END {
    for (id in l) {
        sensitivity = tp[id] / (tp[id] + fn[id])
        specificity = tp[id] / (tp[id] + fp[id])
        accuracy = tp[id] / (tp[id] + fn[id] + fp[id])
        print l[id], sensitivity, specificity, accuracy
    }
}
```

F .5 filter.awk

```
#
#    filter.awk - filter pseudobase.fasta
#
#    Minghui Jiang and Joel Gillespie
#    Fri Jan  9 15:51:54 MST 2009
#

BEGIN {
    if (!min_len)
        min_len = 0
    if (!max_len)
        max_len = 100000
    if (!max_gap)
        max_gap = 100000
    if (!min_den)
        min_den = 0
    if (!max_den)
        max_den = 1
}

/^>/ {
    name = substr($1, 2)
    bases = ""
}

! /^>/ {
    if (!name || query && query != name)
        next

    # 1st line
    if (!bases) {
        bases = $0
        next
    }

    # 2nd line
    if (!allow_hole && bases ~ /N/)
        next
    len = length(bases)
    if (len < min_len || len > max_len)
        next
    db = $0    # dot-brackets
    gap = 0    # max number of consecutive dots in a gap
    n = 0    # total number of dots in gaps
    while (match(db, ":+")) {
        if (RLENGTH > gap)
            gap = RLENGTH
        db = substr(db, RSTART + RLENGTH)
        n += RLENGTH
    }
    if (gap > max_gap)
        next
    den = (len - n) / len
    if (den < min_den || den > max_den)
```

```
        next
    command = "cat > " name ".rna"
    print bases | command
    print $0 | command
    close(command)
}
```

F .6 count.awk

```
#
#    count.awk - count dot-brackets
#
#    Minghui Jiang and Joel Gillespie
#    Sat Feb 21 09:17:52 MST 2009
#

{
    s = $0    # dot-brackets
    k = 0    # max number of consecutive dots
    n = 0    # total number of dots
    while (match(s, ":+")) {
        if (RLENGTH > k)
            k = RLENGTH
        s = substr(s, RSTART + RLENGTH)
        n += RLENGTH
    }
    l = length($0)    # number of bases
    p = (l - n) / 2    # number of base pairs
    d = (l - n) / l    # density of paired bases
    printf("%3d %3d %3d %5.2f", l, p, k, d)
}
```

APPENDIX G

MAKEFILES

G .1 Makefile.generic

```
#
#    Makefile.generic - platform-independent Makefile
#
#    Minghui Jiang and Joel Gillespie
#    Mon Mar  2 16:27:14 MST 2009
#

EXE1 = fold.exe show.exe is.exe
EXE2 = db2bp.exe hx2bp.exe bp2hx.exe bpseq2bpseq.exe
EXE3 = ssa.exe stats.exe linear.exe

SRC0 = README Makefile.generic Makefile.Cygwin Makefile.Linux Makefile.MacOSX
SRC1 = delta.h delta.c fold.c show.c is.c
SRC2 = db2bp.c hx2bp.c bp2hx.c bpseq2bpseq.c
SRC3 = ssa.c stats.c linear.c
SRC4 = filter.awk hx2i.awk count.awk ssa.awk scatter.gp.save

PKB = pseudobase.fasta
HOT = hotknot.zip

.PHONY: all clean clobber spotless

%.o: %.c
    cc -o $@ $(CFLAGS) -c $<
%.exe: %.c
    cc -o $@ $(CFLAGS) $(LDFLAGS) $<

all: delta.o $(EXE1) $(EXE2) $(EXE3)
clean:
    rm -f *.rna *.seq *.db *.bp *.hx *.hx2i *.bpseq *.ct *.seq2
    rm -f *.scr *.sum *.raw *.dat *.gp *.eps *.pdf *.rec
clobber: clean
    rm -f *.o *.exe
spotless: clean clobber
    rm -f *.delta.zip *.summary.txt reconstruct.txt

delta.o: delta.h
fold.exe: fold.c delta.o
    cc -o $@ $(CFLAGS) $(LDFLAGS) $^
show.exe: show.c delta.o
    cc -o $@ $^ $(CFLAGS) $(LDFLAGS) $(GLFLAGS)
is.exe: is.c delta.o
    cc -o $@ $(CFLAGS) $(LDFLAGS) $^

##
#    Pseudo(Knot)Base:
#        .rna .seq .db
#        .pkb.bp .pkb.hx
#        .hx2i
#        pkb
##

.PHONY: pkb

%.rna: $(PKB) filter.awk
    awk -f filter.awk query=$* $<
%.seq: %.rna
    head -n 1 $< > $@
%.db: %.rna
    tail -n 1 $< > $@

%.pkb.bp: %.db db2bp.exe
    cat $< | ./db2bp.exe > $@
%.pkb.hx: %.pkb.bp bp2hx.exe
    cat $< | ./bp2hx.exe > $@

%.hx2i: %.hx hx2i.awk
    sort -n $< | awk -f hx2i.awk > $@

pkb: $(PKB) filter.awk db2bp.exe bp2hx.exe
    rm -f *.rna
    awk -f filter.awk $<
    ls *.rna | sed -e ’s/rna/seq/’ | awk ’{system("make -s " $$0)}’
    ls *.rna | sed -e ’s/rna/db/’ | awk ’{system("make -s " $$0)}’
    ls *.rna | sed -e ’s/rna/pkb.hx/’ | awk ’{system("make -s " $$0)}’

##
#    HotKnot:
#        .bpseq
#        .hotknot.bp .hotknot.hx
#        hot
##

.PHONY: hot

%0.bpseq: $(HOT)    # extract pre-computed PKB?????0.bpseq
    unzip $(HOT) $@

# install HotKnot_v1.2 (Linux only)
# HotKnots.tar.gz from http://www.cs.ubc.ca/labs/beta/Software/HotKnots/
#HotKnot: HotKnots.tar.gz
#    tar zxf $<
#    cp HotKnot_v1.2/hotspot/$@ .
#    cp -r HotKnot_v1.2/hotspot/params .
#
# repeat HotKnot experiment (Linux only)
#%0.bpseq: %.seq
#    ./HotKnot -b -noPS -I $* > /dev/null
#$(HOT): HotKnot pkb
#    rm -f *0.bpseq
#    ls *.rna | sed -e ’s/\.rna/0.bpseq/’ | awk ’{system("make " $$0)}’
#    zip -rmT $@ *0.bpseq

%.hotknot.bp: %0.bpseq
```

```
    awk '$$1<$$3 {print $$1, $$3}' $< > $@
%.hotknot.hx: %.hotknot.bp bp2hx.exe
    cat $< | ./bp2hx.exe > $@

hot: $(HOT) bp2hx.exe
    rm -f *0.bpseq
    unzip $(HOT) > /dev/null
    touch *0.bpseq
    ls *0.bpseq | sed -e 's/0.bpseq/.hotknot.hx/' | awk '{system("make -s " $$0)}'

##
#    Delta:
#        .delta.bp .delta.hx .delta.is.hx .delta.is.bp .seq2 .bpseq
#        .result .show .movie
##

%.delta.bp: %.seq fold.exe
    ./fold.exe -i $< -o2 $@ -d
%.delta.hx: %.delta.bp bp2hx.exe
    cat $< | ./bp2hx.exe > $@

%.delta.is.hx: %.delta.hx %.seq is.exe
    cat $*.delta.hx | ./is.exe $*.seq > $@
%.delta.is.bp: %.delta.is.hx hx2bp.exe
    cat $< | ./hx2bp.exe > $@

%.seq2: %.seq %.delta.is.bp fold.exe
    ./fold.exe -i $*.seq -i2 $*.delta.is.bp -o $@

%.bpseq: %.seq %.delta.is.bp bpseq2bpseq.exe
    cat $*.delta.is.bp | ./bpseq2bpseq.exe $*.seq > $@

%.result:
    make $*.delta.hx $*.delta.is.hx $*.seq2

%.show: %.seq %.delta.hx2i %.delta.is.hx2i %.delta.is.bp %.seq2 show.exe
    cat $*.seq; cat $*.delta.hx2i; echo ">"; cat $*.delta.is.hx2i
    ./show.exe -i2 $*.delta.is.bp -i $*.seq2

%.movie: %.seq fold.exe show.exe
    ./fold.exe -i $< -movie | ./show.exe -i $< -movie

##
#    Do prediction experiment and save results in mmddHHMM.delta.zip
#        delta stop
##

.PHONY: delta stop

%.delta.zip:
    rm -f *.delta.hx
    ls *.rna | sed -e 's/rna/delta.hx/' | awk '{system("make " $$0)}'
    zip -rmT $@ *.delta.hx

delta: all pkb
```

```
    while true; do make `date +%m%d%H%M`.delta.zip; done
stop:
    killall make awk gawk ./fold.exe

##
#    Analyze results in mmddHHMM.delta.zip
#        txt sts scatter.pdf
##

.PHONY: txt

%.delta.is.scr: %.db %.delta.is.hx %.pkb.hx count.awk ssa.exe
    echo -n $* "" > $@
    awk -f count.awk $*.db >> $@
    echo -n " " >> $@
    ./ssa.exe $*.delta.is.hx $*.pkb.hx >> $@
    echo " DeltaIS" >> $@

%.hotknot.scr: %.db %.hotknot.hx %.pkb.hx count.awk ssa.exe
    echo -n $* "" > $@
    awk -f count.awk $*.db >> $@
    echo -n " " >> $@
    ./ssa.exe $*.hotknot.hx $*.pkb.hx >> $@
    echo " HotKnot" >> $@

%.sum: %.delta.is.scr %.hotknot.scr %.seq %.db %.delta.hx2i %.delta.is.hx2i\
 %.hotknot.hx2i %.pkb.hx2i
    cat $*.delta.is.scr > $@
    cat $*.hotknot.scr >> $@
    cat $*.seq >> $@
    cat $*.db >> $@
    cat $*.delta.hx2i >> $@
    echo ">" >> $@
    cat $*.delta.is.hx2i >> $@
    echo ">" >> $@
    cat $*.hotknot.hx2i >> $@
    echo ">" >> $@
    cat $*.pkb.hx2i >> $@
    echo >> $@

%.summary.txt: %.delta.zip
    rm -f *.delta.hx
    unzip $*.delta.zip > /dev/null
    touch *.delta.hx
    rm -f *.delta.is.hx
    ls *.delta.hx | sed -e 's/delta.hx/delta.is.hx/' | awk '{system("make -s " $$0)}'
    rm -f *.delta.is.scr
    ls *.delta.hx | sed -e 's/delta.hx/delta.is.scr/' | awk '{system("make -s " $$0)}'
    rm -f *.sum
    ls *.delta.hx | sed -e 's/delta.hx/sum/' | awk '{system("make -s " $$0)}'
    cat *.sum > $@

txt: all pkb hot
    ls *.rna | sed -e 's/rna/hotknot.scr/' | awk '{system("make -s " $$0)}'
    rm -f *.summary.txt
    ls *.delta.zip | sed -e ’s/delta.zip/summary.txt/’ | awk ’{system("make " $$0)}’

.PHONY: sts

sts: ssa.awk stats.exe
    awk ’$$12=="DeltaIS" {print FILENAME, FILENAME, $$6, $$7, $$8}’ *.summary.txt \
    | awk -f ssa.awk | awk ’{print $$2, $$3, $$4}’ | ./stats.exe
    awk ’$$12=="HotKnot" {print FILENAME, FILENAME, $$6, $$7, $$8}’ *.summary.txt \
    | awk -f ssa.awk | awk ’{print $$2, $$3, $$4}’ | ./stats.exe
    # FILENAME, FILENAME, true positives, false negatives, false positives
    #    | FILENAME, sensitivity, specificity, accuracy
    #    | sensitivity, specificity, accuracy
    #    | average and stdev of sensitivity, specificity, accuracy
    awk ’$$12=="HotKnot" && $$11==1 {print $$1}’ *.summary.txt  | sort | uniq | wc -l
    awk ’$$12=="DeltaIS" && $$11==1 {print $$1}’ *.summary.txt  | sort | uniq | wc -l
    awk ’$$12=="DeltaIS" && $$11==1 {print $$1}’ *.summary.txt  | sort | uniq -c | \
    awk ’$$1 == $(shell ls *.summary.txt | wc -l)’ | wc -l

# scatter.pdf

delta.is.raw: ssa.awk
    awk ’$$12=="DeltaIS" {print $$1, $$2, $$6, $$7, $$8}’ *.summary.txt | \
    awk -f ssa.awk | awk ’{print log($$1)/log(2), $$4}’ > $@
hotknot.raw: ssa.awk
    awk ’$$12=="HotKnot" {print $$1, $$2, $$6, $$7, $$8}’ *.summary.txt | \
    awk -f ssa.awk | awk ’{print log($$1)/log(2), $$4}’ > $@
    # id, length, true positives, false negatives, false positives
    #    | length, sensitivity, specificity, accuracy
    #    | log_2(length), accuracy

scatter.gp: scatter.gp.save delta.is.raw hotknot.raw linear.exe
    cp $< $@
    cat delta.is.raw | ./linear.exe | awk ’{print ",", $$1, "* x +", $$2, \
    "with lines ls 3\\"}’ >> $@
    cat hotknot.raw | ./linear.exe | awk ’{print ",", $$1, "* x +", $$2, \
    "with lines ls 4"}’ >> $@

%.dat: %.raw
    awk ’BEGIN{srand()} {print $$1 + (rand() - 0.5) * 0.02, $$2}’ $< > $@
    # perturb the raw data

scatter.eps: scatter.gp delta.is.dat hotknot.dat
    gnuplot $<
scatter.pdf: scatter.eps
    ps2pdf $<

##
#    Do reconstruct experiment
#        reconstruct
##

%.rec: %.seq %.pkb.bp fold.exe
    ./fold.exe -i $*.seq -i2 $*.pkb.bp -e > $@

reconstruct: all pkb
```

```
    ls *.rna | sed -e 's/rna/rec/' | awk '{system("make " $$0)}'
    cat *.rec > reconstruct.txt

##
#    Software package and source code
#        all.zip code.pdf
##

all.zip: $(SRC0) $(SRC1) $(SRC2) $(SRC3) $(SRC4) $(PKB) $(HOT)
    zip -r $@ $^

%.h.ps: %.h
    enscript -MLetter -2r -Ec -T4 --header='$$n||$$%' -p $@ $< || true
%.c.ps: %.c
    enscript -MLetter -2r -Ec -T4 --header='$$n||$$%' -p $@ $< || true
%.awk.ps: %.awk
    enscript -MLetter -2r -Eawk -T4 --header='$$n||$$%' -p $@ $< || true
%.ps: %
    enscript -MLetter -2r -T4 --header='$$n||$$%' -p $@ $< || true
%.pdf: %.ps
    ps2pdf $<

code.pdf: delta.h.pdf delta.c.pdf fold.c.pdf show.c.pdf is.c.pdf \
        Makefile.generic.pdf \
        db2bp.c.pdf hx2bp.c.pdf bp2hx.c.pdf bpseq2bpseq.c.pdf \
        ssa.c.pdf stats.c.pdf linear.c.pdf \
        filter.awk.pdf hx2i.awk.pdf count.awk.pdf ssa.awk.pdf
    pdftk $^ output $@
```

G .2 Makefile.Cygwin

```
CFLAGS = -O3 -Wall
GLFLAGS = -I/usr/include/w32api -lglut32 -lglu32 -lopengl32 -L/usr/lib/w32api
include Makefile.generic
```

G .3 Makefile.Linux

```
CFLAGS = -O3 -Wall
LDFLAGS = -lm
GLFLAGS = -lglut -lGLU -lGL
include Makefile.generic
```

G .4 Makefile.MacOSX

```
CFLAGS = -O3 -ansi -pedantic -Wall
GLFLAGS = -framework GLUT -framework OpenGL -framework Cocoa
include Makefile.generic
```