5-29-2003

A Progressive Folding Algorithm for RNA Secondary Structure Prediction

Samuel J. Stearns
Dartmouth College

A Progressive Folding Algorithm for RNA Secondary Structure Prediction

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Samuel J. Stearns

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 29, 2003

Examining Committee:

Robert L. Drysdale, Ph.D, Chair

Robert H. Gross, Ph.D, Member

Bruce R. Donald, Ph.D, Member

Carol L. Folt
Dean of Graduate Studies

Dartmouth Computer Science Technical Report TR2017-835

Abstract

RNA secondary structure prediction is an area where computational techniques have shown great promise. Most RNA secondary structure prediction algorithms use dynamic programming to compute a secondary structure with minimum free energy. Energy minimization algorithms are less accurate on larger RNA molecules. One potential reason is that larger RNA molecules do not fold instantaneously. Instead, several studies show that RNA molecules fold progressively during transcription. This process could encourage the molecule to fold into a structure that is not at the global lowest energy level. Additionally, dynamic programming algorithms do not allow for a important type of structure called a pseudoknot. Secondary structure prediction allowing pseudoknots was recently shown to be NP-complete.

We have created a simulation that captures these biological insights. Our simulation uses a probabilistic approach to fold the molecule progressively as it is synthesized. This thesis evaluates the performance of the simulation and presents several enhancements to improve efficiency and accuracy. Our results show that our progressive folding algorithm did not improve on current techniques. Additionally, we found that a simulated annealing algorithm using our probability models was more accurate than our progressive folding algorithm.

Acknowledgements

I feel very fortunate to have two great advisors, Scot Drysdale and Bob Gross. The multi-disciplinary nature of this project sparked many interesting discussions and I feel I have learned a great deal about Computer Science, Biology and the research process from them.

I am very grateful to my parents for being so supportive of my decision to pursue graduate studies. I would also like to thank my grandparents, Charles Stearns and Anne McCarthy for imbuing whatever scholarly aptitude and quantitative ability I may possess. Finally, I would like to thank Bhavnesh Kaushik for his extraordinary hospitality during my time in Hanover.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

High-throughput genome sequencing is one of the most exciting technical achievements in modern science. Today, many sequencing projects, most notably the Human Genome Project, have overwhelmed Biologists with raw sequence data. With the genome available through a few mouse clicks, "proteomics", or the quest to understand the function of all the proteins expressed by a particular organism, has become the most important challenge in Molecular Biology. The specific shape of a macromolecule determines how it interacts with other components and therefore, its function. In order to understand the machinery of life, we need to solve the three-dimensional structures of tens of thousands of interesting macromolecules.

Unfortunately, the leading technologies for structural biology, X-Ray crystallography and NMR, have not scaled to keep up with the explosion of genomic data. This situation creates a pressing need for Computer Scientists to devise new algorithmic approaches for predicting structural information

from nucleic or amino acid sequences. This thesis describes a new approach for predicting the secondary structure of RNA molecules. We hope that this research will help improve on our understanding of RNA folding by simulating the underlying Biology more faithfully.

1.2 RNA

Ribonucleic Acids (RNAs) are one of the most important classes of molecules in the cell. An RNA molecule is a polymer made up of four types of nucleotides: adenine, uracil, guanine and cytosine. RNA differs from DNA in three important ways: it contains the sugar ribose instead of deoxyribose, it contains uracil instead of thymine, and most importantly it usually exists as a single strand, while typically DNA is double stranded. DNA's double stranded structure is the famous double helix discovered by Watson and Crick and provides the basis for genetic replication. Since RNA is single stranded, it can form into a variety of different shapes that allow it to carry out different functions in the cell.

RNA is an important part of the "Central Dogma of Molecular Biology," which describes how information encoded in DNA is used to make proteins. In this process, an RNA polymerase enzyme binds to a stretch of DNA and transcribes genetic information from the DNA to a messenger RNA molecule (mRNA). Since RNA and DNA share the same alphabet of nucleotides (with the substitution of uracil for thymine), RNA is a natural medium for carrying genetic information. Once transcribed, mRNAs travel outside of the nucleus into the cytoplasm. In the cytoplasm, ribosomes translate the information coded in the mRNA to synthesize a specific sequence of amino acids called a protein. Since mRNA molecules are intermediary messengers, their structures are thought to be uninteresting, aside from a possible role mRNA in splicing.

"Non-coding" RNAs have interesting 3-D structures that allow them to participate in critical cellular reactions. Different types of non-coding RNA molecules perform a variety of important tasks in the cell. Transfer RNAs (tRNAs) carry amino acids to ribosomes where they are assembled into proteins. Ribosomal RNAs (rRNAs) make up parts of the ribosome in conjunction with ribosomal proteins and carry out enzymatic reactions that form peptide bonds. Five small nuclear RNAs make up part of the spliceosome, a protein/RNA complex that removes introns from pre-mRNA transcripts in eukaryotes [2]. The signal recognition particle, which allows proteins to be secreted, is an RNA/protein complex [11]. Some RNA molecules are part of metabolic pathways that create other RNAs. These molecules include small nuclear RNAs, which help process ribosomal RNAs [14] and RNAse P molecules, which are involved in the production of tRNAs from precursor RNAs. Finally, RNAs also play a part in several post-transcriptional genetic regulatory mechanisms [15, 16, 21].

RNA molecules may also hold the key to understanding the origin of life. In 1986, Walter Gilbert proposed the "RNA World" hypothesis, which posits that RNA molecules formed the basis for primordial life forms [9]. In the RNA world, organisms consisted of RNA genomes that were replicated by catalytic RNA molecules. Gilbert's hypothesis also proposes that modern structural, catalytic RNAs may be "molecular fossils" that modern organisms inherited from the RNA World.

Biologists describe the structure of an RNA molecule at three different levels. The primary structure is the sequence of nucleotides in the molecule from the $5'$ to the $3'$ direction. The secondary structure consists of the particular base pairings between the nucleotides. Finally, the tertiary structure refers to the specific 3-D spatial arrangement of all the atoms in the RNA molecule. The tertiary structure of an RNA molecule determines its function. Thus, mis-folding can lead to disease states. Unfortunately, the "RNA folding problem", determining the specific tertiary structure of an RNA

given its sequence, is a daunting computational task that is at the frontier of the techniques of modern Computer Science and Biophysics.

RNA secondary structure prediction is a much more tractable problem. Currently there exist polynomial time algorithms that can accurately predict the secondary structure of small RNAs. These predictions can give important clues about the molecule's tertiary structure, aiding Biologists in the wet lab.

1.3 RNA Secondary Structure

The secondary structure of an RNA molecule refers to the specific base pairing of "complementary" nucleotides. A "base pairing" refers to the interaction between complementary bases through hydrogen bonding. Complementary nucleotides include G and C, which interact through three hydrogen bonds, A and U, which interact through two hydrogen bonds, and G and U, which interact through a single hydrogen bond. Other base pairings exist, and are called "non-canonical." Non-canonical pairs distort regular A-form helices formed by the G-C and A-U pairings. These distortions help proteins recognize specific RNAs [8].

Contiguous base pairs are referred to as *stems*. The three-dimensional shape of a base pair is approximately planar. When base pairs stack upon each other in a stem, the three dimensional shape is a double helix. Single stranded subsequences between stems are called *loops*. If the loop is between two halves of a stem, it is called a *hairpin loop*. Some loops occur in the middle of stems. A loop only on one side of the stem is called a *bulge*, while a pair of loops on each side of the stem is called an *interior loop*. Finally, *multi-branched loops* have three or more stems radiating from them. Secondary structures can be represented by a two-dimensional picture such as the one in

figure 1.1.

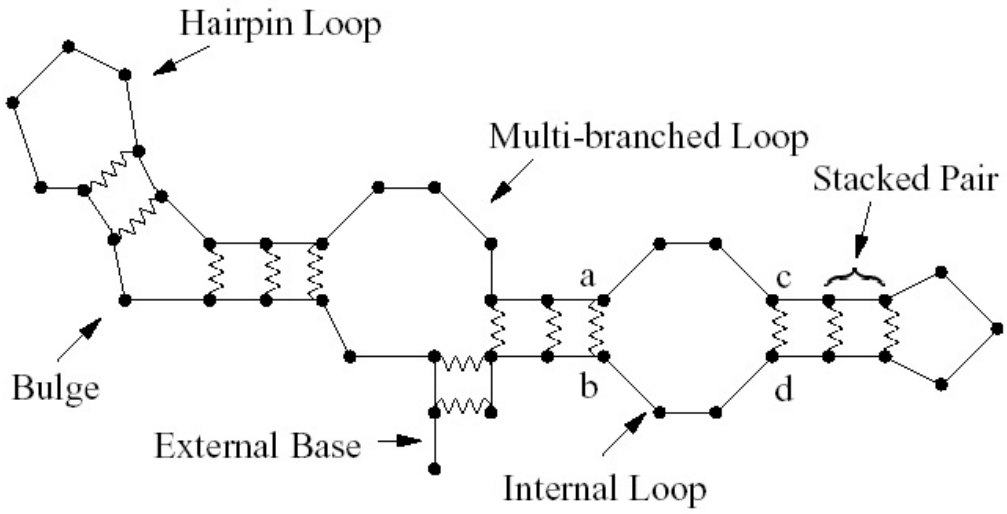

Figure 1.1: Secondary structure diagram [27]

Base pairs are usually nested. Formally, base pairs (i, j) and (i', j') are nested if: $i < i' < j' < j$ or $i' < i < j < j'$. Non-nested base pairs are called *pseudoknots*. While the number of pseudoknots is usually small, they are an important feature for accurate three dimensional structure prediction.

Biologists analyze new protein and DNA sequences by searching for similar sequences, known as "homologs". Homologous sequences can give clues to an unknown sequence's function and offer evidence about evolutionary relationships. Many popular algorithms have been developed for this task, including BLAST [1] and FASTA [20].

RNA sequence analysis is more complicated, because non-coding RNA molecules sometimes conserve secondary structure more than primary sequence. This task is further complicated because an RNA sequence may have a very large number of plausible structures. For example, a sequence of just 200 nucleotides can have as many as 10^{50} possible base paired structures [8]. Today, the

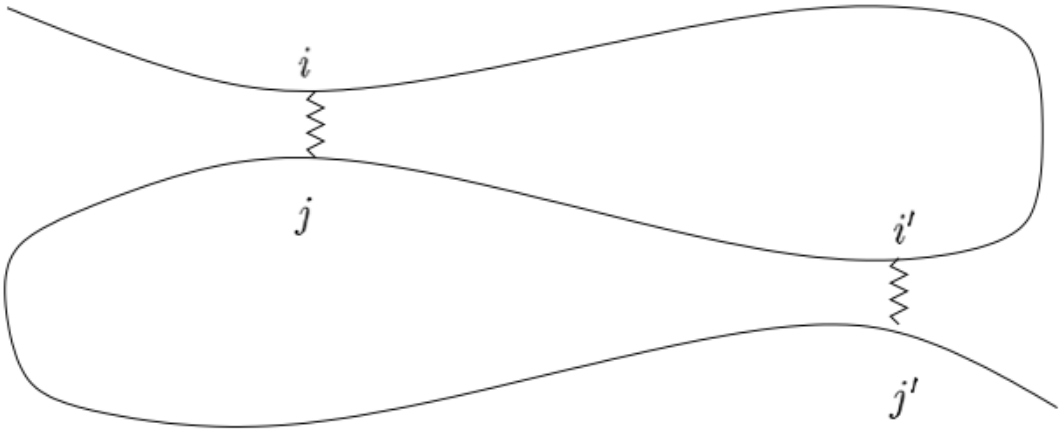

Figure 1.2: A pseudoknot [27]

most accurate RNA secondary structure prediction is done by comparative sequence analysis, a tedious manual process [30]. This method is time-consuming and requires significant expertise. In practice, most biologists use the leading secondary structure prediction algorithm, MFOLD [31].

1.4 Motivations

Algorithms for automated RNA secondary structure prediction have been an active area of research in Computational Biology for close to twenty years. A review of the literature is provided in chapter two. Generally speaking, secondary structure prediction algorithms have shown some success in predicting the secondary structures of smaller RNAs.

The most sophisticated secondary structure prediction algorithm, MFOLD, uses a free energy minimization function on the entire RNA sequence [32]. We hypothesize that this assumption may account for a degradation in accuracy on larger RNAs. In nature, RNAs fold progressively as they

are transcribed. This allows the molecule to form local states that may not be part of the structure with the global lowest energy. This effect is likely to be more pronounced on larger molecules that have more possible structures.

Our hypothesis is supported by several studies. In 1993, Mougey, *et al.* [17] took electron microscope photographs showing that RNA molecules begin to fold during transcription. Then in 1999, Pan, *et al.* [19] provided biochemical evidence for transcriptional folding. Their experiments showed that a denatured *B.subtilis* RNAse P molecule folded into a different structure than one produced *in vivo*.

To test our hypothesis, we created an RNA folding simulation, "ProFold," that contains a progressive folding algorithm and a stochastic model for forming and breaking stems. We hoped that our progressive folding algorithm would model the underlying folding mechanism more faithfully. Additionally, by using a stochastic approach for creating and breaking stems, our algorithm can predict pseudoknots, which are ignored by current methods. The general problem of predicting RNA secondary structures with pseudoknots allowed was shown to be NP-complete by Lyngsø and Pederson [12].

1.5 Contributions of this Thesis

Scot Drysdale and Bob Gross have been working on progressive folding algorithms for RNA secondary structure prediction with several Dartmouth students. The most recent, Charlie DeZiel '01, implemented the first complete version of the folding algorithm in Java. Charlie's code improved on the organization and robustness of the previous efforts, which were written in C. Charlie also refined the algorithms and probability models as he implemented the simulation. Along with the

implementation he wrote a program to test various combinations of parameters and used it to evaluate the simulation on an *S.cerevisiae* alanine tRNA molecule.

Since picking up where Charlie left off in Winter 2002, I have made progress in several areas. First, I spent a lot of time optimizing and debugging the code. This makes folding larger RNAs faster. Second, I made several improvements to the folding algorithm and probability models to more accurately reflect the underlying biology. To help better understand the working of the algorithm, I developed two visualization tools and an intuitive GUI-based front end. Next, I tested the simulation on a variety of different RNA molecules, including tRNAs, RNAse Ps, and 5S, 16S, and 23S rRNAs. The results of these experiments show that our algorithm gets most of its accuracy from simulated annealing in "cleanup mode," rather than during transcription. Additionally, the accuracy of our algorithm degrades on larger RNA molecules. We were disappointed to find that our results contradicted our hypothesis, but we hope that this work provides a first step in creating an architecture for further efforts to develop heuristic methods for RNA secondary structure prediction including pseudoknots.

Briefly, here is a sketch of the remaining chapters:

- **Chapter 2** reviews previous work in RNA secondary structure prediction algorithms
- **Chapter 3** introduces our folding algorithm and probability models
- **Chapter 4** provides details on how our simulation was implemented
- **Chapter 5** describes our visualization tools
- **Chapter 6** lists the results of our experiments
- **Chapter 7** presents our conclusions and outlines directions for future research

Chapter 2

Previous Work

Biologists, Mathematicians and Computer Scientists have been researching algorithms for RNA secondary structure prediction for nearly twenty-five years. To date, the most effective algorithms are based on energy minimization methods [24]. Base pairing lowers an RNA's free energy compared to its single stranded state. This increases the molecules' stability, so this approach is a meaningful approximation of the underlying biophysics. Dynamic programming techniques provide an efficient means to generate optimal structures (without pseudoknots) given a particular energy minimization function. This chapter discusses three algorithms based on dynamic programming and a proof that the general RNA secondary structure prediction problem is NP-complete if pseudoknots are included. We also mention some alternative approaches in the literature. Durbin and Eddy [8] *et al.* present a well-written introduction to the field.

2.1 Base Pairing Maximization

In 1978, Ruth Nussinov and colleagues presented an efficient dynamic programming algorithm that maximizes set of base pairings in a nucleic acid [18]. This is a very coarse energy minimization model, since base pairs lower the free energy of the molecule. Since the free energy function is so simple, it does not make very accurate predictions. Additionally, it is limited by the fact that it does not allow for pseudoknots or non-canonical base pairs.

The Nussinov algorithm has two parts: fill stage, which computes the dynamic programming matrix, and a traceback stage which finds the optimal path through the matrix. Traceback stage runs in linear time and memory, while the fill stage runs $O(n^3)$ in time and $O(n^2)$ memory. Interestingly, the authors note that the $O(n^2)$ memory requirements are a "practical difficulty" for larger RNAs with n in the few thousand. How times have changed!

Nussinov's algorithm can be defined as follows. Given a sequence x of length N with symbols $x_1, \ldots x_N$. Define the function $\delta(i, j) = 1$ if x_i and x_j are a complementary base pair; else $\delta(i, j) = 0$. We recursively calculate a table of maximum matching size $MMS(i, j)$ which represent the secondary structure with the maximum number of base pairs for the subsequence $x_i, \ldots x_j$.

Initialization:

First, the MMS table is initialized, to account for the fact that nucleotides are not allowed to base pair with themselves or their neighbor.

for i = 2 to N **do**

$MMS(i, i-1) = 0$;

end for

for i = 1 to N **do**

$MMS(i, i) = 0$;

end for

Fill Stage:

The fill stage operates recursively. It finds the maximum number of base pairings for a subsequence (i, j) by evaluating four possible operations: adding the unpaired nucleotide i onto the best structure for the subsequence $(i + 1, j)$, adding the unpaired nucleotide j onto the best structure for the subsequence $(i, j - 1)$, adding the base pair between i and j to the best structure for $(i + 1, j - 1)$, and combining two neighboring optimal substructures (i, k) and $(k + 1, j)$, where $i < k < j$. These four operations do not allow for pseudoknots.

for all subsequences of length 2 to length N: **do**

$$MMS(i,j) = \max \begin{cases} MMS(i+1,j), \\ MMS(i,j-1), \\ MMS(i+1,j-1) + \delta(i,j), \\ \max_{i<k<j} \{MMS(i,k) + MMS(k+1,j)\}. \end{cases}$$

end for

After the fill stage computes the dynamic programming tables, $MMS(1, N)$ contains the number of base pairs in the structure with the maximum matching. There can be more than one structure with this many base pairs. In order to enumerate one of these maximally base paired structures, we use a traceback procedure common to many dynamic programming algorithms.

Traceback stage:

push $(1, L)$ onto the stack.

repeat

pop (i, j).

if $i >= j$ **then**

continue.

else if $MMS(i+1, j) = MMS(i, j)$ **then**

push $(i+1, j)$.

else if $MMS(i, j-1) = MMS(i, j)$ **then**

push $(i, j-1)$.

else if $MMS(i+1, j-1) + \delta_{i,j} = MMS(i, j)$ **then**

record i, j base pair.

push $(i+1, j-1)$.

else

for $k = i+1$ to $j-1$: **do**

if $MMS(i, k) + MMS(k+1, j) = MMS(i, j)$ **then**

push $(k+1, j)$.

push(i, k).

break.

end if

end for

end if

until the stack is empty.

2.2 Energy Minimization

Michael Zuker's MFOLD [31] algorithm is the most popular method for RNA secondary structure prediction. Zuker's algorithm is implemented in the programs MFOLD and ViennaRNA [10] and is included in the popular GCG Wisconsin package [6]. Like Nussinov's work, Zuker's algorithm is an efficient dynamic programming algorithm for finding the secondary structure with the lowest free energy [32]. It runs in $O(n^3)$ time and requires $O(n^2)$ space for sequence of length n. The two major limitations of Zuker's algorithm are that it does not calculate pseudoknots and is less accurate for larger RNAs, which often in reality do not fold into their global lowest energy state.

Zuker's work is a more sophisticated version of the energy minimization approach introduced by Nussinov. It assumes that the correct structure has the lowest equilibrium free energy ($\triangle G$). The free energy is calculated by summing the energy of the various features of the secondary structure. The algorithm also introduces a mechanism to deal with "stacking," the interaction between neighboring base pairs. Experimental data suggests that stacked base pairs lower $\triangle G$, so Zuker's algorithm calculates the energy of a stem with length n as the sum of $n-1$ base stacking terms.

Zuker's algorithm uses estimates of $\triangle G$ based on experimental evidence with small RNAs [28]. These tables include parameters for most of the secondary structure features mentioned in Section 1.3, including hairpin loops, bulge loop lengths, interior loop lengths, multi-branch loops and single dangling nucleotides. Pseudoknots are notably absent. Improved parameters were determined in 1999 [13] and are included in MFOLD version 3.1.

Zuker's algorithm is very similar to Nussinov's dynamic programming algorithm. It recursively

calculates three arrays: $V(i,j)$, which records the minimum energy for a secondary structure on substring $i,\ldots,j$ when i and j form a base pair, $WM(i,j)$, when the structure for $i,\ldots,j$ is part of a multi-branched loop, and $W(i)$, which records the minimum energy structure on the substring $(1,\ldots,i)$.

MFOLD uses several energy functions to account for different types of secondary structure features. These include eH for hairpin loops, eS for stacking base pairs, eL for internal loops and bulges and eM for multi-branched loops. The energy function for stacking base pairs, eS, calculates $\triangle G$ based on the interaction between the base pair i,j and a neighboring or "stacked" base pair, $i+1,j-1$. The function for internal loops and bulges is given a base pair i,j, which represents one side of the internal loop and the base pair i',j', which represents the first base pair at the other side of the internal loop. This function enforces the condition that $i'-i+j'-j>2$ in order to ensure there is at least one (for a bulge) or more (for an internal loop) unpaired nucleotides between the two base pairs i,j and i',j'. Lastly, the energy function for multi-branched loops, eM, take two parameters, k and k', which represent the number of helices and number of unpaired bases in the multi-branched loop, respectively.

$$V(i,j)=\min\begin{cases} eH(i,j), \\ eS(i,j,i+1,j-1)+V(i+1,j-1), \\ \min\limits_{\substack{i<i'<j'<j \\ i'-i+j-j'>2}}\{eL(i,j,i',j')\} \\ \min\limits_{i+1<k<j}\{WM(i+1,k-1)+WM(k,j-1)+a\} \end{cases}$$

$$WM(i,j) = \min \begin{cases} V(i,j) + b, \\ WM(i,j-1) + c, \\ WM(i+1,j) + c \\ \min_{i<k<=j} \{WM(i,k-1) + WM(k,j)\}. \end{cases}$$

$$W(i) = \min \begin{cases} W(i-1), \\ \min_{0<=k<i} \{W(k) + V(k+1,i)\}. \end{cases}$$

In later work, Zuker further extended his algorithm to calculate suboptimal folds [31]. This gave researchers the ability to incorporate experimental constraints into the fold and to see a wider range of structures. This is very useful since the actual structures of RNA molecules usually are not at the global lowest energy level.

2.3 The Pseudoknot Problem

Rivas and Eddy extended Zuker's algorithm to predict most types of pseudoknots [22]. This is the first algorithm to calculate minimum energy foldings including pseudoknots with the accepted RNA thermodynamic model used by MFOLD. Rivas and Eddy use gap matrices in addition to dynamic programming matrices and attempt to line up the gap matrices recursively in order to find pseudoknots. Their algorithm runs in $O(n^6)$ time and requires $O(n^4)$ storage. The authors note that the algorithm's high polynomial complexity makes it impractical for RNAs with more than 140 nucleotides. Folding a 100 nucleotide RNA takes four hours and 22.5 megabytes of memory on a SGI R10K Origin 200. Considering pseudoknots leads to a much larger search space of possible structures, so their algorithm is less accurate than Zuker's on RNAs that do not have pseudoknots.

Lyngsø and Pedersen extended Rivas and Eddy's work by devising a secondary structure prediction algorithm that handles pseudoknots that runs in $O(n^5)$ time with a $O(n^3)$ space requirement [12]. The reduced runtime makes their algorithm practical on RNAs with length up to 350-375 bases. To make this improvement possible, they restrict their algorithm to only allow one pseudoknot of arbitrary complexity. Their algorithm splits the sequence into four subsequences, forms a non-pseudoknotted secondary structure between alternating sequences and then combines the two secondary structures to produce the entire secondary structure.

In the same paper, [12] Lyngsø and Pederson also present a proof that the general case of secondary structure prediction is NP-complete. They present a simplified secondary structure model, the Nearest Neighbor Pseudoknot Model, which calculates energy based on a base pair and its surrounding bases. To further simplify the model they restrict it to Watson and Crick base pairings. The proof is based on a restricted $3SAT$ form. Based on this result, the authors contend that the secondary structure prediction with the energy model used by Rivas and Eddy is NP-hard.

2.4 Other Approaches

RNA secondary structure prediction is an active area of inquiry and many researchers have proposed algorithms in addition to the ones we have previously described [18, 31, 22, 12]. While dynamic programming techniques, specifically MFOLD, have become the *de facto* standard, there have been several other interesting approaches to the problem. Many of these are similar to our work in that they are heuristics that allow for pseudoknots.

Tabaska *et al.* formulated the problem as a maximal weighted matching problem [26]. His algorithm predicts RNA secondary structures with pseudoknots in $O(n^3)$ time, but requires extensive

statistical information on the secondary structures of similar RNAs in order to generate the weighting function. Several researchers have explored genetic algorithms [29, 25]. Eddy and colleagues implemented Nussinov's algorithm using a stochastic context free grammar [8] and published a formal grammar for RNA secondary structure [23].

Chapter 3

Overview

This chapter introduces our folding algorithm and the probability models it employs. A detailed description of how the algorithm is implemented is provided in Chapter 4.

3.1 Folding Algorithm

Unlike other secondary structure prediction algorithms, our algorithm folds the molecule as it is transcribed. The folding algorithm has two stages: transcription mode and cleanup mode. During each stage, we generate possible form and break stem actions based on the molecule's current state. An action represents a change in the molecule's secondary structure through the forming or breaking of a stem. Each action has an associated probability. We store the actions in a queue, sorted in descending order of probability. Actions are performed stochastically, if a random number is equal to or less than the probability associated with the action. The rules for generating actions are described in section 3.2.

Transcription Mode:

Transcription mode models the state of the molecule as it is transcribed.

```
for all nucleotides in the RNA molecule do
  transcribe the nucleotide
  if the nucleotide can lengthen a stem then
    lengthen the stem
    update queue by removing outdated actions, generating new actions
  else
    add any form stem actions involving this nucleotide to the action queue. Use current time
    as start time so they can be performed during this iteration
  end if
  update the probabilities of all of the actions in the action queue.
  for all actions in the queue in decreasing order of probability do
    generate a random number $n$
    if the probability of the current action is $\leq n$ then
      perform the action
      update queue by removing outdated actions and generating new actions
    end if
    increment the RNA molecule's internal time.
  end for
end for
```

Cleanup Mode:

Cleanup mode models the molecule's continued folding once it is fully transcribed. It performs the

same operations as transcription mode. Cleanup mode iterates for an arbitrary number of times before exiting. In Chapter 6, we describe the results of experiments we performed in order to determine the best length for cleanup mode.

for i = 1 to Params.CLEANUP_ITERATIONS **do**
  update the probabilities of all of the actions in the action queue.
  for all actions in the queue in decreasing order of probability **do**
    generate a random number n
    if the probability of the current action is $\leq n$ **then**
      perform the action
      update queue by removing outdated actions, generating new actions
    end if
  end for
  increment the RNA molecule's internal time.
end for
return the molecule's secondary structure

3.2 Probability Models

Actions represent changes to the RNA's secondary structure via the forming and breaking of stems. This section describes the models used to compute the probability of the action occurring.

3.2.1 Forming Stems

The FormStemAction probability model depends on two variables, distance and stem length. In order to simplify our algorithm, we enforce the restriction that a FormStemAction must contain at least three consecutive nucleotides with no skips. This covers most of the stems found in nature and excludes "bulges" that we defined in Chapter 1. Three-nucleotide stems is referred to as "minimum-sized'.'

A "minimum-sized" stem has the probability $p(d) = c\frac{m}{d(i,j)}$, where $d(i,j)$ is the distance between nucleotides i and j, m is the minimum distance between the two halves of any possible stem and and c is the form stem probability parameter. The nucleotides i and j are the middle nucleotides in the respective halves of the stem. The value of m is five since we require at least three nucleotides between the two halves of a hairpin loop and measure from the middle nucleotide in the half stem. The form stem probability c is in the range $[0, 1]$ and is used as a scaling parameter for this function. Additionally, both m and $d(i,j)$ are converted to polymer distances using the formula in section 4.4.1.

To calculate the probability of a larger form stem action, we treat it as a set of independent minimum-sized form stem actions. The set consists of each pair of minimum-sized subsets of the larger form stem action. If any of these minimum-sized actions were to occur, the resulting stem would "zip up," producing the larger form stem action.

Formally, the model can be expressed as follows:

$$P(l,i,j) = 1 - \prod_{n=1}^{l-2}(1 - c\frac{m}{d(i+n, j-n)})$$

where l is the length of the stem, i is the first nucleotide in the lower stem, and j is the last nucleotide in the higher stem (in regards to the $5'$ to $3'$ direction). The terms m, $d(i, j)$, and c have the definitions from the previous paragraph. The model is shown in figure 3.1.

3.2.2 Breaking Stems

Our simulation is different than other secondary structure prediction algorithms because it allows for stems to break as well as form. We think this may reflect the underlying biological process more accurately, by allowing the RNA to form intermediate structures before settling into its final state. Our concept of stem breaking is based on the idea that hydrogen bonds periodically "breathe," which temporarily weakens the bond. We assume that the probability of a stem breaking is the probability that all of the hydrogen bonds in that stem will be "breathing" simultaneously. This makes longer stems more stable than shorter ones.

The probability of a BreakStemAction is defined as: c^h, where c is the break stem probability and h is the number of hydrogen bonds in the stem. The break stem probability lies in the range $[0, 1]$, and represents the probability that one hydrogen bond will be "breathing" at a given time. If a BreakStemAction's start time is greater than the current time, it is assigned a probability of 0. This is to disallow stems from forming and breaking in the same iteration. The probability distribution for various values of the break stem probability is shown in figure 3.2.

3.3 Parameters

The simulation has four parameters that can be set by the end user.

- **Form stem probability** - the term c from section 3.2.1. Used as a scaling factor for the form stem probability distribution.
- **Break stem probability** - the probability that a single hydrogen bond is "breathing" at a given time.
- **Diffusion rate** - when a stem breaks, we assume that the two half stems are initially still in the same vicinity. Each clock tick thereafter, we increment the distance between formerly base paired nucleotides by this parameter.
- **Cleanup iterations** - the number of iterations in cleanup mode.

We performed a series of experiments to try to determine the best values for each parameter. For both the form stem and break stem parameters, we tested ten values in the range $[0, 1]$. For the diffusion parameter, we tested the values $4.0, 8.0, 12.0 and 16.0$. We tested six different values of cleanup iterations the range $[0, 2000]$.

Evaluating the parameter space helped us understand the working of the simulation. Early on in the testing process, we discovered that the diffusion rate did not make a significant impact on the simulation's accuracy, and in later experiments left it fixed at four. In Chapter 6, we describe the experiments we performed to understand the effects of the length of cleanup mode as well as the form stem and break stem probabilities.

3.4 Architecture

The simulation consists of four main components: an RNAFolder class, an ActionQueue class, an RNAMolecule class and a DistanceCalculator class. The RNAFolder class contains the implemen-

tation of our folding algorithm. The ActionQueue class contains a list of all the form and break stem actions that can operate on the molecule, sorted in decreasing order of probability. The RNAMolecule class maintains the state of the molecule and contains methods to generate actions based on this state. It contains an array of Nucleotides and a DistanceCalculator object, which maintains a graph that approximates the spatial relationships between the nucleotides in the RNA. The architecture of the simulation is shown in figure 3.3.

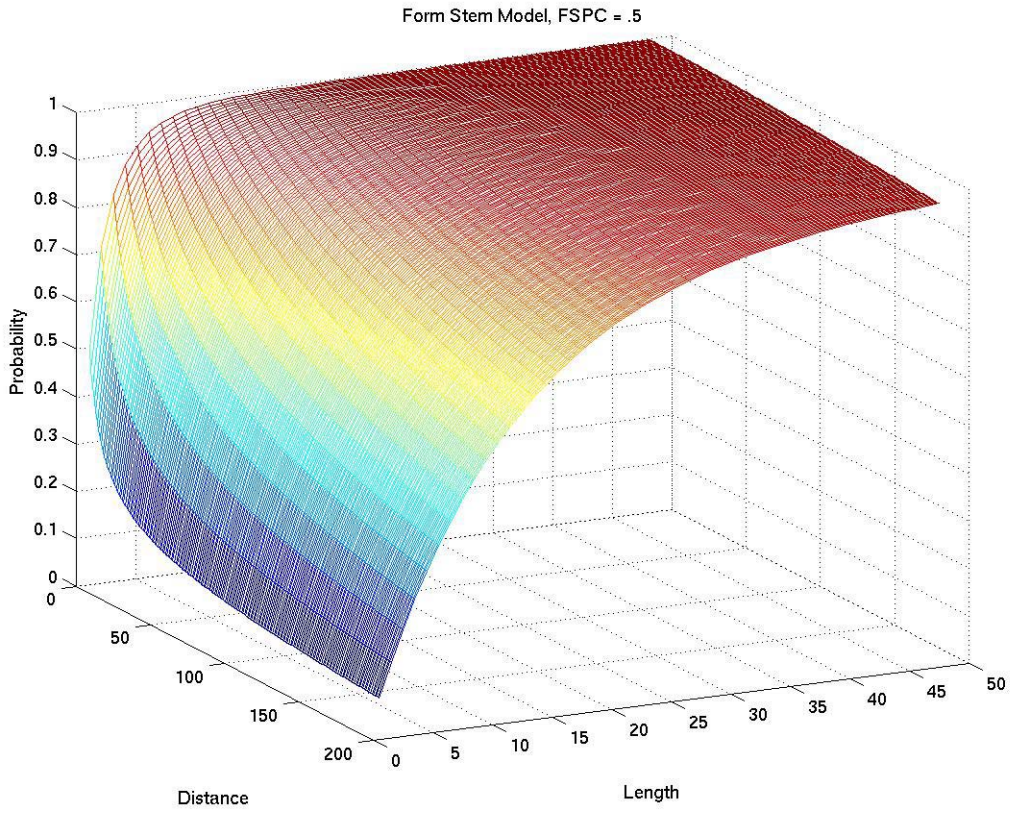

Figure 3.1: Form stem probability model, c = .5

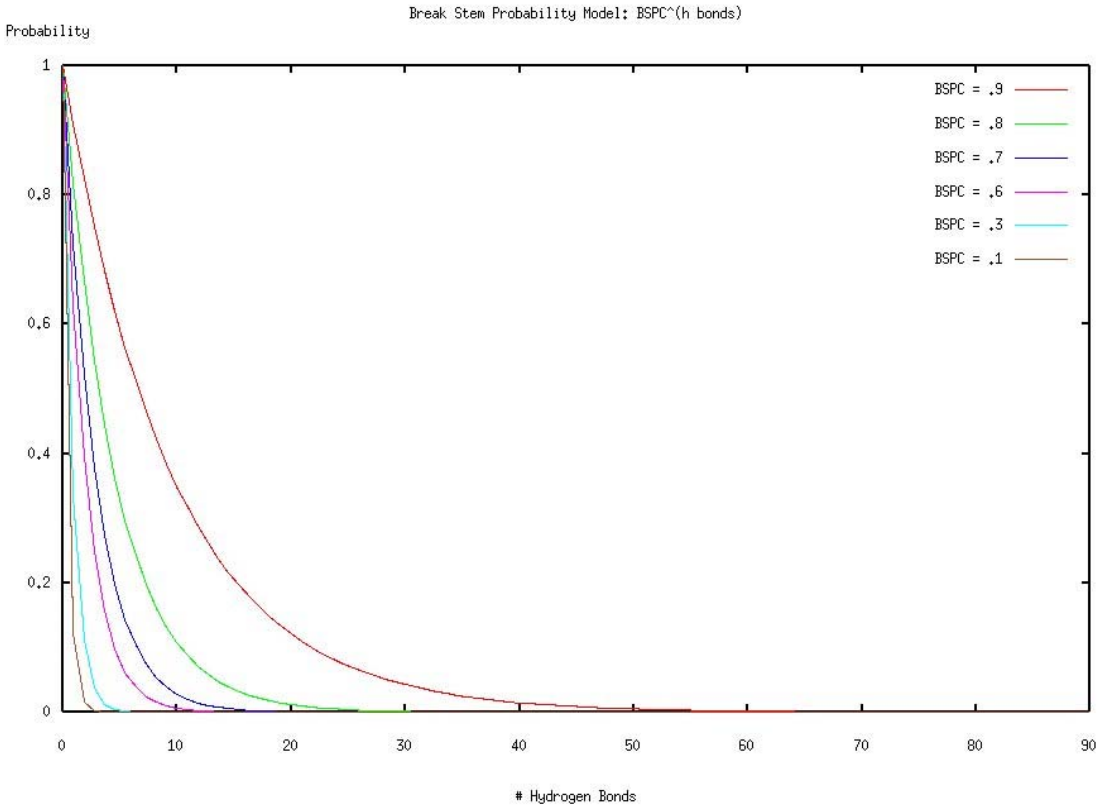

Figure 3.2: Break stem probability model

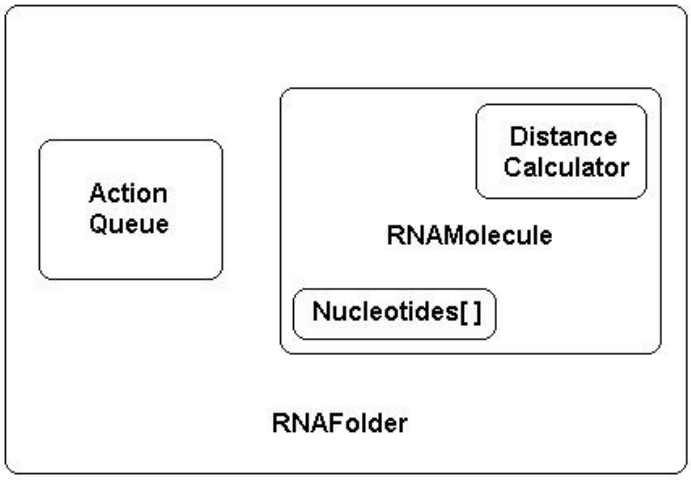

Figure 3.3: Simulation architecture

Chapter 4

Implementation

This chapter describes the details of the algorithms and data structures used in the implementation of our simulation. The simulation is implemented in Java, which facilitates code re-use and encapsulation. The choice of Java helped us write clean, bug-free code but also hurt performance compared to other algorithms that are implemented in C [31, 22].

In our description of each class, we cover the object's member fields and important algorithms. Trivial methods for accessing member fields are not included. These descriptions were adopted from the original javadoc documentation [7] and reflect the changes I have made to the code.

4.1 Nucleotide

The Nucleotide class describes the state of a single nucleotide in the RNA. This information includes the base type and information about base pairing, such as the nucleotide's present and previous partners.

Fields:

- char baseType - the type of nucleic acid.
- int partner - index of base paired nucleotide, if any.
- int previousPartner - index of the nucleotide this nucleotide was previously base paired to, if any.
- double distanceToPartner - distance to the nucleotide this nucleotide is base paired to, if any.
- double distanceToPreviousPartner - distance to the nucleotide this nucleotide was previously base paired to, if any.
- Subsequence parentSS - the subsequence (paired or unpaired) this nucleotide is a part of.

Methods:

- **void pairWith(int newPartner):** pairs this nucleotide to another nucleotide specified by the index newPartner. Sets distance to partner to be Constants.HYDROGEN_BOND_LENGTH, which is currently 1.0.

  Runtime: $O(1)$.

- **void breakBond():** breaks this nucleotide's base pairing and updates its previous partner information, setting the distance to its previous partner to Constants.INITIAL_BREAK_DISTANCE, which is currently 1.0.

  Runtime: $O(1)$.

- **void incrementDistanceToPreviousPartner():** increments the distance to this nucleotide's previous partner by Params.BROKEN_BASE_PAIR_DIFFUSION_CONSTANT, or does nothing if no previous partner exists.

  Runtime: $O(1)$.

- **boolean canPairWith(Nucleotide other):** returns true if this nucleotide can base pair with the nucleotide *other*.

  Runtime: $O(1)$.

4.2 Subsequence

The subsequence class represents a range of consecutive nucleotides. The subsequence must either be part of a stem or entirely unpaired. Unpaired subsequences must be bordered by stems or the end of the RNA. The RNAMolecule object maintains a list of non-overlapping subsequences covering the full transcribed range of the molecule. When the state of the RNA changes, due to the transcription of another nucleotide, or a stem forming or breaking, the queue of unpaired subsequences in the RNAMolecule object is updated to enforce this condition.

This class is a subclass of Range, which stores the low and high indexes of a chain of nucleotides. The Range class provides several methods for accessing these fields and comparing different ranges.

Range is often used as an argument to many of the methods described in this chapter

Fields:

- int lowIndex the beginning nucleotide.
- int highIndex the ending nucleotide.
- Subsequence otherHalf the subsequence it is base paired to (if any).

Methods:

- **void bondWith(Subsequence restOfStem):** bonds this subsequence to another subsequence.

  Runtime: $O(1)$.

4.3 RNAMolecule

The RNAMolecule class maintains the state of the RNA molecule. In includes data structures to represent the sequence of nucleotides, the current secondary structure and the transcription point of the molecule. The class includes a DistanceCalculator object to calculate spatial distances between nucleotides. Additionally, it keeps track of the simulation's "time." One nucleotide is transcribed per unit of time. Lastly, the class keeps track of the state of the molecule via two priority queues of half stems and unpaired subsequences. These queues are sorted in descending order of subsequence length.

The RNAMolecule class has several important methods that change the state of the molecule.

These include reset, breakStem, formStem, lengthenLastStem and transcribeNextNucleotide. Additionally, it includes methods to generate new actions based on the molecule's current state.

Fields:

- Nucleotide [ ] sequence - sequence of nucleotides in the RNA.
- int pointOfTranscription - the next nucleotide to be transcribed.
- int time - internal time for the molecule. One step corresponds to transcribing one nucleotide.
- DistanceCalculator distanceCalculator - calculates spatial distances between nucleotides.
- PriorityQueue unpairedSubsequenceQueue - stores all the unpaired subsequences.
- PriorityQueue halfStemQueue - stores all the half stems in the molecule.

Methods:

- **void reset()**: resets the RNA molecule to an unfolded state with no nucleotides transcribed.

  Runtime: $O(n)$, where n is the number of nucleotides in this RNA molecule.

- **boolean canLengthenLastStem():** returns true if the last transcribed nucleotide is unpaired, adjacent to a stem, and can form a base pair that lengthens the stem.

  Runtime: $O(1)$, since it is comprised of three constant-time comparisons.

- **boolean canChemicallyBond(Range range1, Range range2):** returns true if two ranges of nucleotides are within the transcribed part of this RNA molecule, are separated by least three nucleotides, are entirely unpaired, and can base pair with each other.

  Runtime: $O(l)$, where l is the length of the prospective stem.

- **boolean canChemicallyBondWithEnoughHBonds(Range range1, Range range2):** returns true if two ranges of nucleotides can chemically bond (as determined by canChemicallyBond) with at least six hydrogen bonds between them.

  Runtime: $O(l)$, where l is the length of the prospective stem.

- **boolean canSpatiallyBond(Range range1, Range range2):** returns true if two ranges of transcribed nucleotides in this RNA molecule can physically reach each other.

  The two ranges of nucleotides can physically reach each other if, for both pairs of nucleotides at the ends of those ranges, there are no paths between the pair that contain half stems that are more than half the length of the path. This is checked by looking at each of the half stems in this RNA molecule and verifying that, for both pairs of nucleotides at the ends of the ranges in question, the half stem in question is shorter than the sum of the shortest paths from the ends of the half stem to the pair of end nucleotides. For each pair of end nucleotides, we only check half stems that are more than half the length of the shortest path between the pair of end nucleotides.

  Runtime: $O(mc)$, where m is the number of half stems in this RNA molecule and c is the cost of a call to DistanceCalculator.getDistance(), which is either $O(1)$ or $O(v \lg v)$, depending on whether the relevant table entries are up-to-date.

- **void incrementTime():** increments the internal "time" of this RNA molecule and updates the distance to each transcribed nucleotide's previous partner. These distances are updated by the diffusion rate parameter. This captures the assumption that unpaired nucleotides should spatially diffuse at a constant rate.

  Runtime: $O(nb)$, where n is the number of transcribed nucleotides and b is the number of nucleotides that have a previous partner from a stem that broke.

- **void transcribeNextNucleotide():** transcribes the next nucleotide in this RNA molecule.

  Runtime: $O(m + n)$, where m is the number of nucleotides in the newly transcribed nucleotide's parent subsequnce and n is the number of transcribed nucleotides. The n factor is caused by updating each nucleotide's parent subsequence. The n factor is caused by calls to DistanceCalculator.addVertex().

- **FormStemAction lengthenLastStem(Vector outdatedActions, Vector newActions)**: this method checks if the last stem in this RNA molecule can be lengthened with a call to canLengthenLastStem(). If yes, the stem is lengthened and the break stem action corresponding to the previous stem is added to *outdatedActions*. The break stem action corresponding to the lengthened stem is added to *newActions* along with any new form stem actions made possible by the updated structure. These actions are assigned start times in the next clock tick so that they will not be possible until the next nucleotide is transcribed. This method returns a FormStemAction representing the lengthened stem, which is used by RNAFolder to eliminate any conflicting form stem actions from the queue.

  Runtime: $O(n + p + m)$, where n is the number of transcribed nucleotides in this RNA molecule. This n term results from a call to DistanceCalculator.addEdge(), while the $(p+m)$ term results from a call to getFormStemActionsForMinStemSizeRange().

- **void formStem(Range range1, Range range2):** makes a stem out of two ranges of transcribed, unpaired nucleotides in this RNA molecule. Adds the two halves of the stem to the halfStemQueue and removes the two halves of the stem from the unpairedSubsequenceQueue.

  Runtime: $O(mn(\lg n))$, where m is the length of the stem being formed and n is the number of transcribed nucleotides in this RNA molecule. This runtime results from m calls to DistanceCalculator.addEdge() for each base pairing in the stem.

- **void breakStem(Range halfStemRange):** breaks a stem corresponding to the range of nucleotides in halfStemRange. Removes the corresponding entries from the halfStemQueue and updates the unpairedSubsequenceQueue to include the broken stem.

  Runtime: $O(nl)$, where l length of the stem being broken and n is the number of transcribed nucleotides in the RNAMolecule. The n factor results from a call to DistanceCalculator.removeEdge().

- **Vector getNeighboringUnpairedRanges(Range range):** returns a vector containing ranges of transcribed, unpaired nucleotides that neighbor *range*.

  Runtime: $O(1)$.

- **Vector getFormStemActionsForMinStemSizeRange(Range range):** returns a vector containing all valid form stem actions involving minimum-sized range, *range*. In order for a form stem action to be considered valid, the two ranges of nucleotides involved must be able to chemically bond as specified by canChemicallyBondWithEnoughHBonds(). Then, the method extends each valid FormStemAction to its maximum possible length with the idea that in real life stems tend to "zip up" as far as is chemically and spatially possible once they have started to form.

  Runtime: $O(p + m)$, where p is the number of possible form stems involving the given range and m is the sum of the numbers of nucleotides involved in each of the form stem actions in the vector that is returned.

- **Vector getFormStemActions(Range range):** returns a vector containing all valid form stem actions that can operate on this RNA molecule that involve the passed range of nucleotides. This includes form stem actions that extend beyond the boundaries of the range of nucleotides passed, but does not include form stem actions that extend fewer than three

nucleotides into the range.

Runtime: $O((s-2)(p+m))$, where s is the size of the argument range, where p is the number of possible form stems involving the given range and m is the sum of the numbers of nucleotides involved in each of the form stem actions in the vector that is returned. This results from $(s-2)$ calls to getFormStemActionsForMinStemSizeRange().

- **Vector getAllFormStemActions():** returns a vector containing all valid form stem actions that can operate on this RNA molecule.

  Runtime: $O(t((s-2)(n+m)))$, where t is the number of unpaired subsequences in this RNA molecule, n is the number of possible form stems involving the given range and m is the sum of the numbers of nucleotides involved in each of the form stem actions in the vector that is returned. The runtime results from t calls to getFormStemActions().

- **Vector getFormStemActionsForEnd():** returns a vector containing all valid form stem actions involving the final transcribed nucleotide of this RNA molecule. Sets the start time of each of these actions to the current internal time of this RNA molecule so that they can be tried before the next nucleotide is transcribed.

  Runtime: $O(n+m)$, where p is the number of possible form stems involving the given range and m is the sum of the numbers of nucleotides involved in each of the form stem actions in the vector that is returned.

- **BreakStemAction getBreakStemAction(Range halfStemRange):** creates a break stem action corresponding to the stem specified by halfStemRange.

  Runtime: $O(1)$.

4.3.1 RNACube

We created the RNACube class in order to optimize one of the most frequently used methods in RNAMolecule, getFormStemActionsForMinStemSizedRange. Previously, given a minimum-sized range, the method iterated through all unpaired, minimum-sized subsequences and attempted to create a form stem action. Since the majority of these subsequences could not base pair with an arbitrary minimum-sized range, this method was very inefficient.

The RNACube class is a lookup table that returns a list of all the possible places a minimum-sized range could base pair with. The lookup table, locations, is a three-dimensional array of linked lists. Each dimension has length four, to represent the four types of nucleotides. The sequence of a three nucleotide range is used as a key into the three dimensional array so that each possible three nucleotide sequence has its own ArrayList recording the indices of where it begins in the RNA.

The constructor, which takes the sequence of nucleotides as input, initializes the lookup tables and three dimensional array. First, it iterates through the sequence, adding an entry to the ArrayList in locations that corresponds the minimum-sized sequence beginning at the particular nucleotide. Then it creates two arrays of vectors, normalMode and cleanupMode, which store the list of matches for each nucleotide in the sequence, with and without G-U bonds. The runtime of the constructor is $O(n)$, where n is the length of the RNA's sequence.

Fields:

- int moleculeLength - the length of the RNA.
- static int CUBE_SIZE - 4, to represent the four possible base pairs.

- Vector normalMode[ ] - an array containing a list of ranges that can base pair with the three nucleotide ranges beginning with each nucleotide under Watson & Crick rules.

- Vector cleanupMode[ ] - an array containing a list of ranges that can base pair with the three nucleotide ranges beginning with each nucleotide under canonical rules.

- ArrayList locations[ ][ ][ ] - a three-dimensional array representing all the possible sequences for a three nucleotide range. Records the index of the beginning nucleotide in an ArrayList.

Methods:

- **Vector getMatches(int index):** returns the starting nucleotide of all the three nucleotide subsequences that the subsequence beginning with *index* can base pair with under strict Watson and Crick rules.

  Runtime: $O(1)$.

- **Vector getCleanupModeMatches(int index):** returns the starting nucleotide of all the three nucleotide subsequences that the subsequence beginning with *index* can base pair with allowing G-C, A-U and G-U bonds.

  Runtime: $O(1)$.

- **Vector findMatches(String RNA, int index):** returns the output of returnItems for the reverse complement (Watson & Crick) of the string RNA beginning at index. If no matches exist, return a null list.

  Runtime: $O(i)$, where i is the the number of items at $locations[x][y][z]$. This method determines the reverse-complement of the three-nucleotide sequence beginning at *index* and then calls returnItems(), which creates the i term in the runtime.

- **Vector findCleanupModeMatches(String RNA, int index):** this function concatenates the results of several calls to findMatches in order to account for all the possible half stems that can form a base pair with the half stem beginning at *index* with at least six hydrogen bonds. This accounts for G-U bonds.

  Runtime: $O(i)$, where i is the the number of items at $locations[x][y][z]$. This method calls returnItems for all the possible reverse-complements of the three-nucleotide sequence beginning at *index*, which creates the i term in the runtime.

- **Vector returnItems(int x, int y, int z):** returns a list of indexes stored in $locations[x][y][z]$. Ensures that values that are returned are more than 5 nucleotides apart from *index* on the backbone of the RNA molecule, since half stems must be separated by at least three nucleotides.

  Runtime: $O(i)$, where i is the the number of items at $locations[x][y][z]$.

4.4 Action

Action is an abstract class representing a change to the molecule's current secondary structure. It is subclassed by FormStemAction and BreakStemAction. Each action has an associated probability and a start time. The start time is used to limit when actions can occur, so the break stem action associated with a newly formed stem cannot occur until the next time tick. Actions with future start times have a probability of zero.

There are three types of Actions: Form Stem Actions, Break Stem Actions and Lengthen Stem actions. Lengthen stem actions occur when a newly-transcribed nucleotide extends a stem at the end of the molecule and are represented as a form stem action with length 1.

Fields:

- RNAMolecule molecule - reference to the parent RNAMolecule object.
- Range highRange - the higher range.
- Range lowRange - the lower range.
- double probability - the probability of the action.
- int startTime - the first time this action is allowed to take place.

4.4.1 FormStemAction

FormStemAction is a subclass of Action. Additionally, instances of FormStemAction are subject to the following rules which are enforced by methods in RNAMolecule. The two ranges of nucleotides must:

- be non-overlapping
- have a distance of at least three nucleotides between them
- be entirely unpaired
- able to base pair with each other
- form at least 6 hydrogen bonds.

Methods:

- **boolean conflictsWith(FormStemAction other):** returns true if either of the ranges involved in this form stem action overlap either of the ranges involved in *other*.

  Runtime: $O(1)$.

- **boolean isContainedWithin(FormStemAction other):** returns true if this FormStemAction is a subset of *other*.

  Runtime: $O(1)$.

- **void calculateProbability():** calculates the action's probability using the model described in section 3.2.

  Runtime: $O(1)$, $O(m)$, or $O((m+(np))$, where m is the cost of a call to RNAMolecule.canSpatiallyBond(), n is the length of the stem, and p is the cost of a call to DistanceCalculator.getDistance(). $O(1)$ if this form stem action is scheduled for a future time. $O(m)$ if the start time is valid but the stem can't form due to spatial constraints. $O((m + (np))$ if the stem can form.

- **double GetPolymerDistance(double length)**: to provide a rough consideration for the three dimensional shape of the RNA, we calculate the polymer distance based on the length of the backbone chain of nucleotides. Polymer chains are flexible and usually form irregular shapes, so they are usually modeled polymers with a random walk method. The polymer distance is calculated as follows: $f(d, l) = l\frac{1}{\sqrt{6}}\sqrt{d}$, where d is the graph distance between the nucleotides and l is the length of one nucleotide [3]. We assume that the length of a nucleotide is equal to 1.0. This corresponds to $\sqrt{\frac{1}{6}}$ of the RMS end-to-end distance.

  Runtime: $O(1)$.

4.4.2 BreakStemAction

Break stem actions represent the possibility of a stem breaking. Whenever a stem is formed a corresponding BreakStemAction is created.

Methods:

- **void calculateProbability ():** calculates the action's probability using the model described in section 3.2. *Runtime:* $O(1)$.

4.4.3 ActionQueue

The Action Queue Object stores a list of FormStemActions and BreakStemActions, sorted in decreasing order of probability. ActionQueue implements the java.util.Comparable interface, so it sorts the queue in $O(n \lg n)$ time.

Fields:

- Vector list - the list of actions.
- boolean isSorted - a flag we set when the queue needs to be sorted.

Methods:

- **void java.util.Collections.sort(Listlist):** Sorts the action queue using the standard method in the java class libraries. This method uses a modified mergesort that offers guaranteed $O(n \lg n)$ performance. In order to sort the action queue in descending order of probability,

we violate the contract of this interface in defining Action.compareTo(), which returns the opposite of what normally would be expected.

Runtime: $O(n \lg n)$, where n is the number of elements in the queue.

4.5 DistanceCalculator

A DistanceCalculator object maintains a simple spatial model of the RNA's secondary structure. The structure is stored as undirected graph, with a vertex for each nucleotide. Edges represent the distance between nucleotides that are base paired, recently base paired (to account for broken stems), or adjacent along the backbone of the RNA. These considerations restrict the graph's structure, enforcing a maximum of four edges per vertex.

The graph is stored in an adjacency-list representation. We also maintain tables of shortest-path distances and shortest-path predecessors, which are calculated with Dijkstra's algorithm. Our implementation of Dijkstra's algorithm is based on [4]. We employ a min-priority queue, which is implemented as an inner class and is based on [5].

Since the graph is updated frequently, calling Dijkstra's algorithm is the major performance bottleneck in our simulation. To alleviate this, we took advantage of the graph's restricted structure to avoid recalculating the tables whenever possible. We made the observation that operations to remove an edge or set an edge length always increase the distance of an edge. When an edge is removed, the distance between its vertices is increased by a constant factor that represents the nucleotides diffusion through space. The method to set an edge length is only called for this purpose. Given this information, we only have to recalculate the shortest paths involving that edge.

We also noticed that we always add vertices to the graph at the same place: the $3'$ end of the RNA. This new vertice is only connected to its neighbor. Thus, we can calculate all the shortest paths to the newly transcribed nucleotide by copying the shortest path entries for its neighbor and adding 1. Forming a stem adds an edge at an arbitrary location in the graph, which reduces the distance between the surrounding nucleotides. Since the location of this edge is generated stochastically, we have to regenerate the full tables.

Fields:

- Vector g - the adjacency-list representation of the graph.
- double[ ][ ] d - the table of shortest-path distances between vertices in the graph.
- int[ ][ ] pi - table of predecessors for the shortest paths.
- boolean validTableEntries[ ] - tracks whether a particular vertex's table entries must be re-calculated.

Methods:

- **void reset(int n):** resets this distance calculator, giving it a graph with n vertices and no edges. Resets all values of validTableEntries to true.

  Runtime: $O(l)$, where l is the length of the RNA.

- **int addVertex():** adds a vertex to the graph to represent the transcription of a new molecule. Since the distance between backbone nucleotides is assumed to be constant, we initialize the distances from the new vertex, v, to the existing vertices $i = (1, \ldots, v-1)$ to $d[v][i] = d[v-1][i]+1$ and $d[i][v] = d[i][v-1]+1$. The table of shortest path predecessors is initialized

in a similar manner, since all paths to and from the new vertex must run through its neighbor. Returns the new number of vertices in the graph.

Runtime: $O(n)$, where n is the number of vertices in the graph.

- **void addEdge(int u, int v, double l):** adds edge between vertex u and vertex v with length l to this distance calculator's graph.

  Runtime: $O(n)$, where n is the number of vertices in the graph. Adding an edge could effect any shortest path in the graph so we invalidate all the table entries for each vertex in the graph.

- **void setEdgeLength(int u, int v, double l):** sets the length of edge (u, v) in this distance calculator's graph to l. Calls invalidateTableEntries for u and v. This method always lengthens the edge because it is only called by RNAMolecule.breakBond(), which breaks a stem, and RNAMolecule.incrementTime(), which diffuses a recently broken stem.

  Runtime: $O(n)$, where n is the number of transcribed nucleotides in the RNA molecule. This runtime results from a call to invalidateTableEntries.

- **void removeEdge(int u, int v):** removes edge (u, v) from this distance calculator's graph. Calls invalidateTableEntries for u and v.

  Runtime: $O(n)$, where n is the number of transcribed nucleotides in the RNA molecule. This runtime results from a call to invalidateTableEntries.

- **double getDistance(int u, int v):** returns the length of the shortest path between two vertices in this distance calculator's graph.

  Runtime: $O(1)$ if the table entries for u are valid, otherwise $O(v(\lg v))$, where v is the number of vertices in the graph, since we use Dijkstra's algorithm to recalculate all the single-source

shortest paths from u to all the other vertices in the graph.

- **void dijkstra(int s):** uses Dijkstra's algorithm to calculate single-source shortest paths from s to all other vertices in this distance calculator's graph. Fills in row s in the this.d and this.pi tables. Based on [4].

  Runtime: $O(v(\lg v))$, where v is the number of vertices in the graph.

- **void initializeSingleSource(int s):** resets row s in this distance calculator's d and pi tables.

  Runtime: $O(v)$, where v is the number of vertices in the graph.

- **void relax(int s, int u, int v, DistanceCalculator.MinPriorityQueue q):** tests whether the shortest path from vertex s to vertex v can be improved by going through an intermediate vertex u. If so, updates $d[s][v]$, $pi[s][v]$, and q accordingly. Based on [4].

  Runtime: $O(\lg v)$, where v is the number of vertices in q.

- **double length(int u, int v):** returns the length of edge (u, v) in this distance calculator's graph.

  Runtime: $O(1)$.

- **void invalidateTableEntries(int u, int v):** invalidates the table entries for any vertex that is a predecessor to u or v. This method iterates through all the vertices in the graph $(1, \ldots, i)$ and invalidates the table entries for vertex i if pi$[i][u] = v$ or pi$[i][v] = u$.

  Runtime: $O(n)$, where n is the number of transcribed nucleotides in the molecule.

4.5.1 DistanceCalculator.Edge

This inner class is used to represent an edge in the graph.

Fields:

- int v - the second vertex of the edge.
- double l - the length of the edge.

4.5.2 DistanceCalculator.MinPriorityQueue

This inner class is used by the implementation of Dijkstra's algorithm to represent a min-priority queue of vertices that is sorted by the values in $d[s]$ for some int s. It doesn't work quite like a normal min-priority queue because it is only used in one place for a specific purpose. This implementation is based on [5].

Fields:

- int[ ] heap - represents this min-priority queue.
- int heapSize - the number of vertices currently in this min-priority queue.
- int[ ] invertedHeap - stores the indices of the vertices in the heap so that these vertices can be located in constant time. Provides a lookup table for the values in heap, which are rearranged as the heap is sorted.
- int row - used to find the keys for the vertices in this min-priority queue: $d[row][i]$ contains the key for vertex i.

Methods:

- **int parent(int i):** returns the index of the parent of the vertex in heap[i].

  Runtime: $O(1)$.

- **int left(int i):** returns the index of the left child of the vertex in heap[i].

  Runtime: $O(1)$.

- **int right(int i):** returns the index of the right child of the vertex in heap[i].

  Runtime: $O(1)$.

- **double key(int v):** returns the key of vertex v.

  Runtime: $O(1)$.

- **void swap(int heapIndex1, int heapIndex2):** swaps the positions of $heapIndex1$ and $heapIndex2$ in heap[ ]. Updates invertedHeap to keep track of this change.

  Runtime: $O(1)$.

- **void minHeapify(int i):** lets the vertex at heap[i] "float down" in the heap so that the subtree rooted at index i becomes a min-heap. Assumes that the trees rooted at left(i) and right(i) are min-heaps already.

  Runtime: $O(h)$, where h is the height of the argument i in the tree.

- **int extractMin():** returns the index of the vertex v with the lowest key, and removes v from this min-priority queue.

  Runtime: $O(\lg s)$, where s is the size of the heap. This method calls minHeapify, so $\lg s$ is the maximum depth of the heap.

- **void decreaseKey(int v):** decreases the key of vertex v to the current value in d[this.row][v]. This method doesn't check to make sure that the new key is smaller than the old key, because the old key is not stored anywhere.

  Runtime: $O(\lg s)$, where s is the size of the heap.

4.6 RNAFolder

The RNAFolder class expresses our folding algorithm. It contains an RNAMolecule object and an ActionQueue object to maintain the state of the molecule and keep track of possible actions.

Fields:

- RNAMolecule molecule - the molecule currently being folded.
- ActionQueue queue - the list of actions that can happen to the molecule.

Methods:

- **void tryAllActionsInQueue():** tries every action in this RNA folder's action queue in decreasing order of probability. An action is performed if a generated random number is less than or equal to the action's probability. Whenever an action is performed, any actions that are no longer chemically possible are removed from the action queue, and any new actions that have become possible are added to the action queue.

  Runtime: $O(nc)$, where n is the number of actions in the queue and c is the cost of performing an action.

- **String fold():** transcribes this RNA folder's RNA molecule and folds it according to the results of actions generated by our probability model. Implements the folding algorithm described in section 3.1. Returns a string representing the final secondary structure in connect format.

Chapter 5

Visualization Tools

This chapter presents two visualization tools and a graphical user interface for the simulation.

5.1 Circles

Circles plots are a standard way to view RNA secondary structures. In a circles plot, the outside circle represents the backbone of the RNA and inscribed arcs show the base pairs in the structure. A tool for drawing circles plots is included in the popular GCG Wisconsin package. Figure 5.1 shows an example circles plot for a *S.cerevisiae* 5S RNA.

To help understand our simulation, we needed a means to overlay circles plots to compare two possible folds of an RNA. Since this feature was not included in the GCG Circles tool, we developed our own tool for circles plotting. It is capable of drawing circles plots for single molecules, as shown in figure 5.1 as well as for comparing two folds, as shown in figure 5.2. In 5.2, green arcs represent

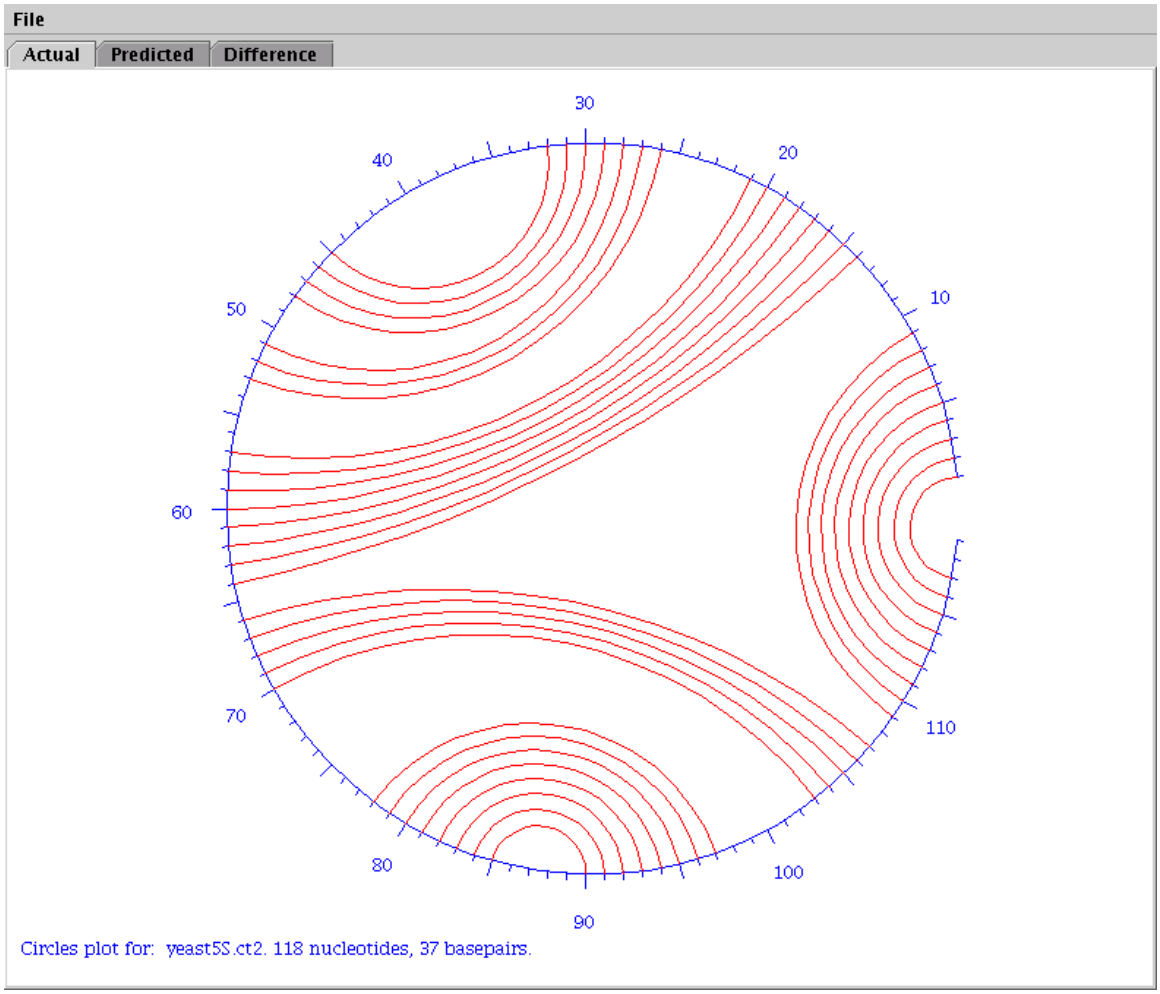

Figure 5.1: Circles plot

correctly predicted base pairs, red arcs represent incorrectly predicted base pairs, and blue arcs represent base pairs that are part of the correct structure but are missing in the predicted structure. Circles reads the standard "connect" (.ct2) format and can save images as jpegs.

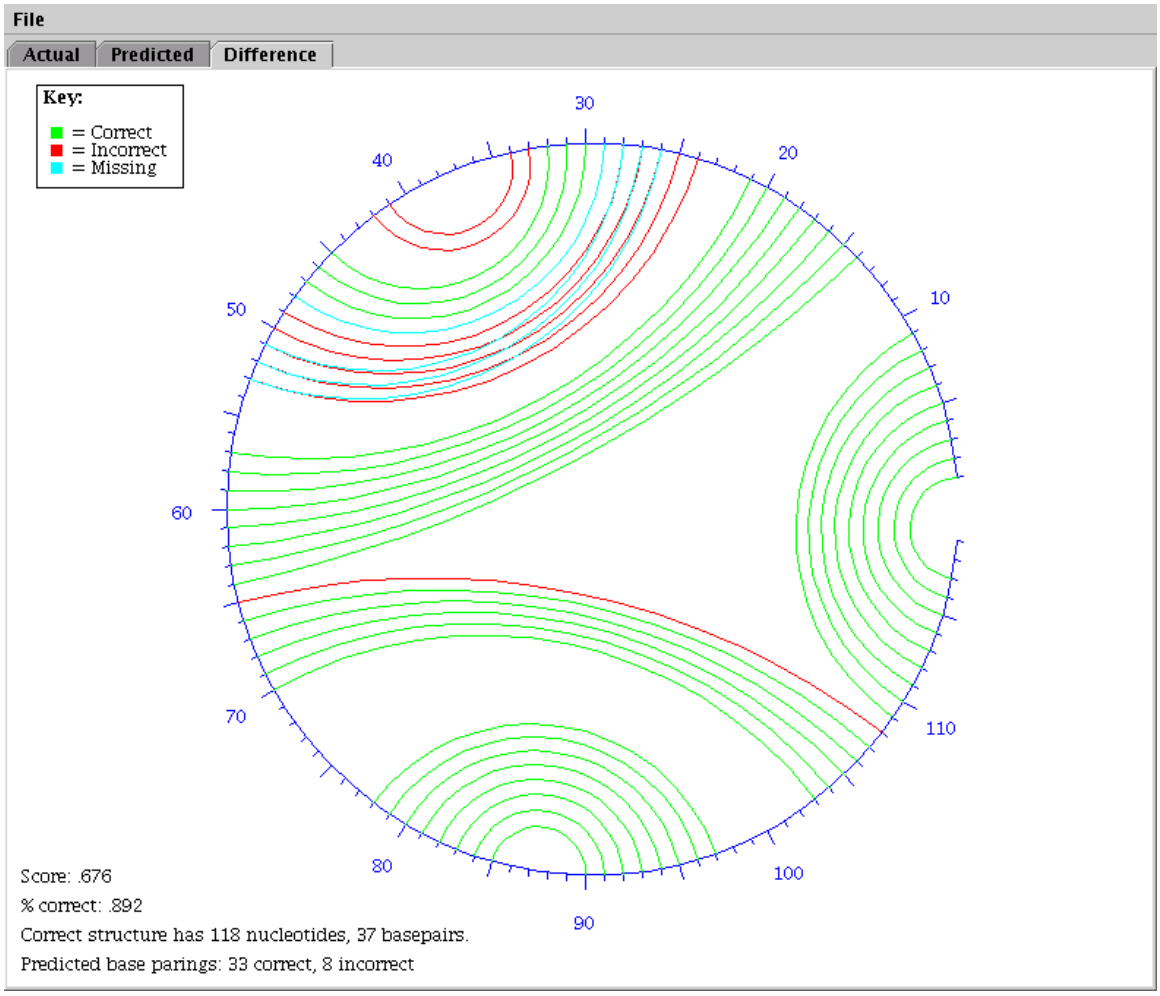

Figure 5.2: Comparison plot using Circles

5.2 Trace

To fully understand the working of our simulation we developed an animated visualization tool, Trace, that tracks the progress of an RNA as it is folded. The trace tool draws an animated circles

plot of that represents a molecule's secondary structure as it is folded.

Figure 5.3 shows a screenshot of the trace tool in action. At the bottom of the window is a status bar displaying information about the most recent action. The current transcription point is tracked by a red circle on the outside of the circles plot. The user can change the speed of the animation, as well as toggle back and forth through the actions one by one. The stop button halts the animation and resets the molecule, while the "End" button draws the final structure. The user is able to overlay the correct structure, which is drawn using the same color scheme as described in the Circles code.

Additionally, Trace provides the ability to view the probability distribution of actions that were performed during the fold. This helped us refine our probability models by understanding the distributions they generated when run on real molecules. Figure 5.4 shows a histogram including a breakdown of form stem (in blue) and break stem (in red) actions.

5.3 ProFold

The ProFold tool presents an an intuitive graphical user interface developed using the Swing toolkit. The interface is shown in figure 5.5.

The ProFold tool allows the end user to fold the molecule. Since the simulation is stochastic, multiple folds are recommended. ProFold has an option to choose the number of folds to perform, which are stored in the format *OutputFile.FoldNumber.extension*. Additionally, ProFold can output the standard "connect" file format (.ct2) or create .trace files for use with our Trace tool.

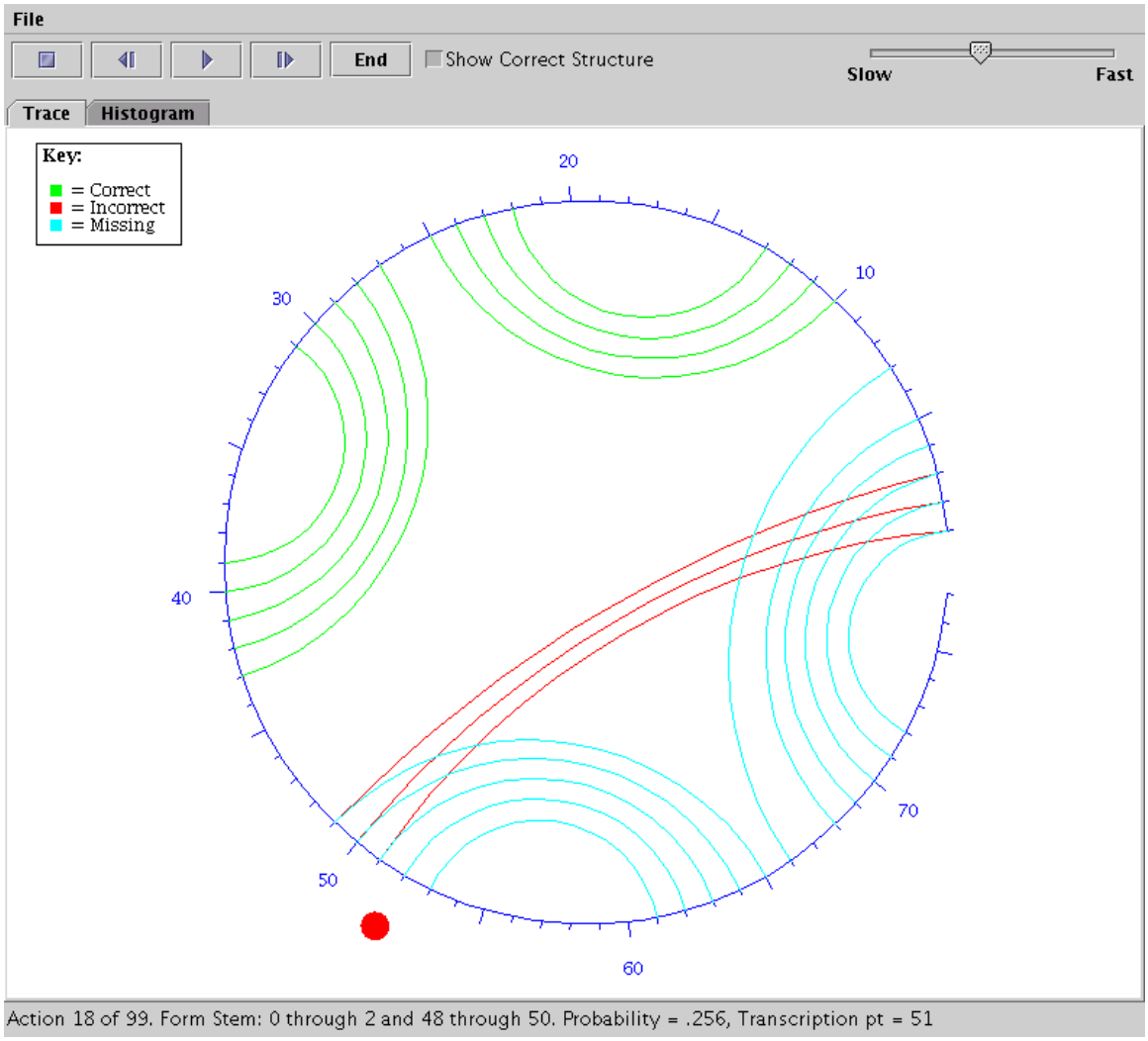

Figure 5.3: Trace

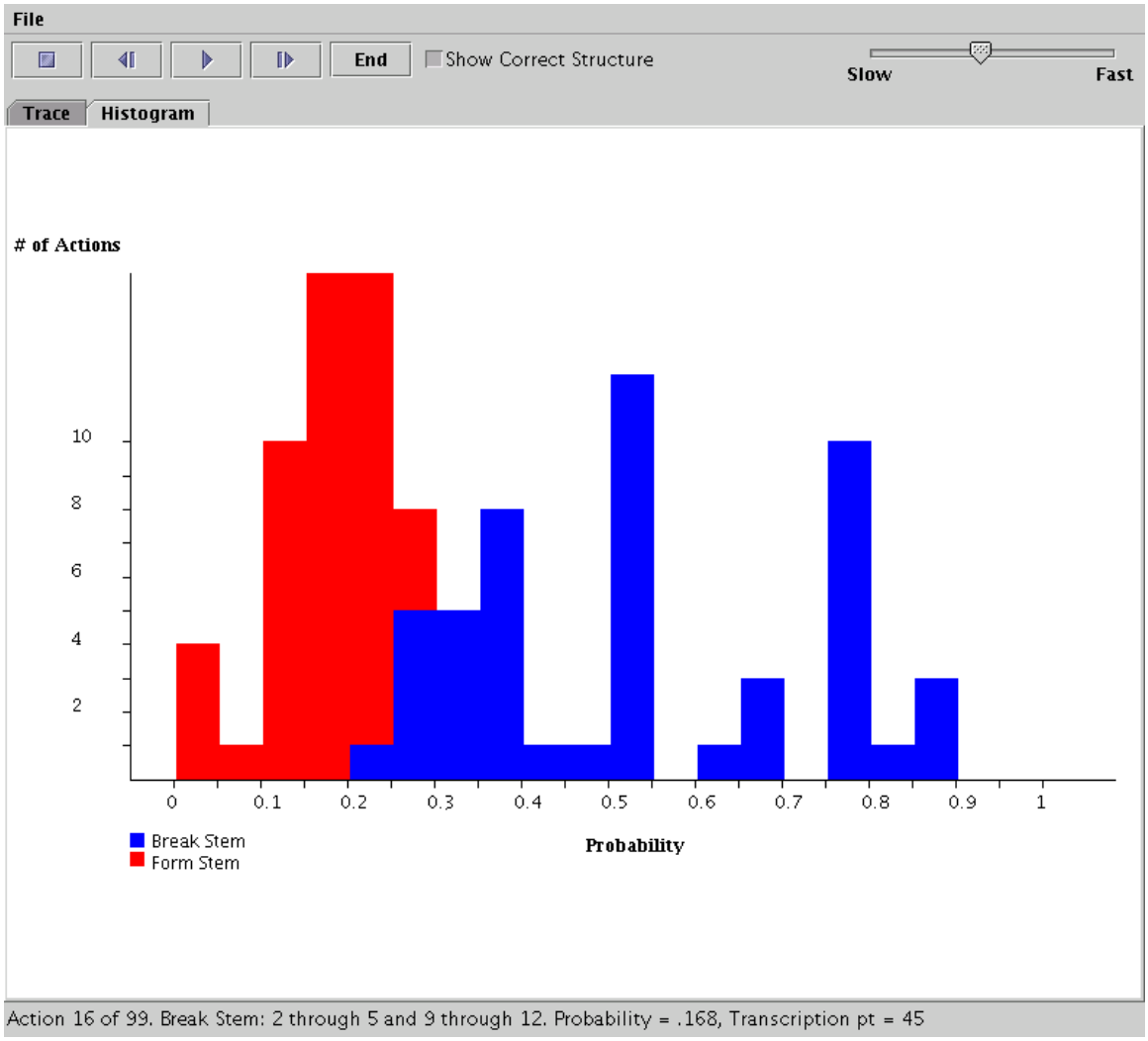

Figure 5.4: Histogram view

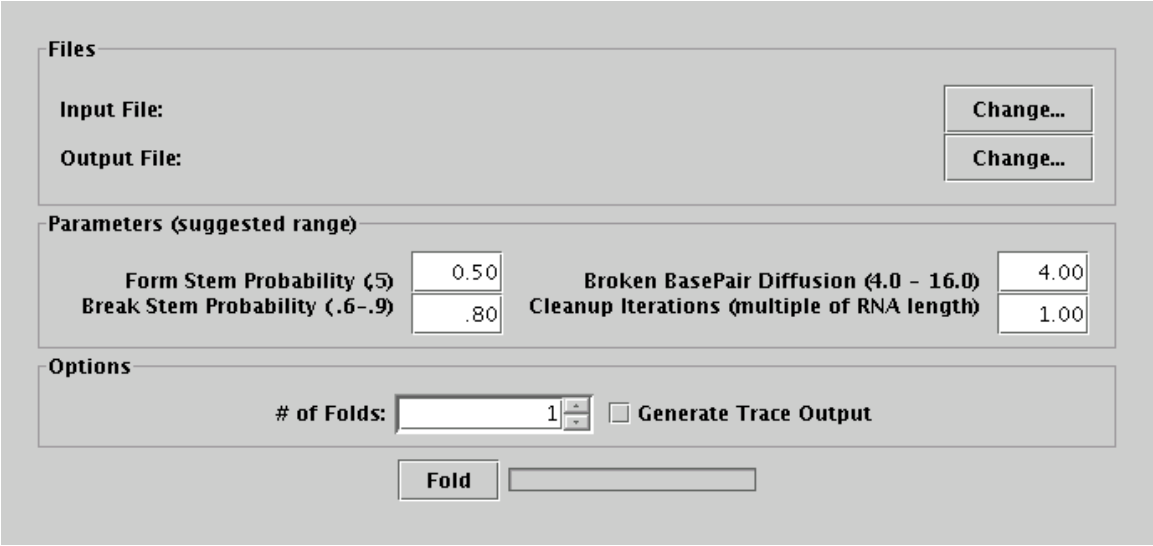

Figure 5.5: ProFold graphical user Interface

Chapter 6

Results

This chapter presents the findings of our research. We begin with a review of our algorithmic improvements and scoring system, and then compare the accuracy of MFOLD and ProFold on a range of RNAs. We also present the results of several experiments to investigate the effect of our algorithm as a form of simulated annealing. Lastly, we show the speedups gained from our optimization efforts.

6.1 Preliminaries

6.1.1 Algorithmic Improvements

In order to assess the performance of ProFold, we initially spent a lot of time trying to determine the best parameters for the simulation. To accomplish this we used a parameter evaluation program [7] that repeatedly folded a molecule with various combinations of parameters and output the

combinations that produced the most accurate folds. These results showed us that the break stem and form stem parameters were the most important variables.

This process lead to several improvements to the algorithm as well as the optimizations described in section 6.7. First, we decided to simplify the folding algorithm. Originally, the algorithm had three stages: transcription, pre-cleanup mode and cleanup mode. Both transcription and pre-cleanup mode disallowed stems containing G-U base pairs. After the molecule was transcribed, it entered pre-cleanup mode, where it iterated until the queue did not contain any actions with probability greater than a parameter called the cutoff probability. After leaving pre-cleanup mode, the algorithm entered cleanup mode, where rules for forming stems were relaxed to allow G-U base pairs. The algorithm iterated in cleanup mode until the queue didn't contain any possible form stem actions. Both pre-cleanup and cleanup mode shared a cap of 1000 as the maximum number of iterations.

We simplified the folding algorithm by making three changes: eliminating pre-cleanup mode, allowing G-U base pairs throughout the fold, and having cleanup mode iterate for a fixed number of times. We the first two changes are more faithful to the underlying biology. The last change makes understanding the effect of cleanup mode easier, since it removes its stochastic component.

Additionally, our parameter evaluation results led to changes to our probability models. We adopted a new probability model for generating form stem actions, discussed in 3.2.

Previously, we assumed that the chance of two half stems bumping into another was inversely proportional to the volume defined by their maximum distance. We thus assigned the probability $\frac{c}{d^3}$ to a minimum-sized form stem action, where c was the form stem probability parameter and d was the distance between the two half stems. We decided to replace this with the current model

after implementing the polymer distance calculation. After the polymer distance calculation was added we had difficulty searching for a good range of values for c. This made it apparent that the parameter c was a "magic number" that did not have any biological or probabilistic significance. This was reinforced by the discovery that many values of c resulted in probabilities that exceeded 1.0.

Lastly, we were troubled with our probability model for breaking stems, since we got the best results with high (close to 1.0) parameters that did not make biological sense - since a base pair should not be "breathing" regularly. We tested an alternative model, but it was much less accurate. This result led us to focus on the simulated annealing hypothesis that we discuss later in this chapter.

6.1.2 Scoring System

The results presented in this chapter use a common scoring system. Each fold was scored on the basis of correct base pairings out of total base pairings. Since both our algorithm and MFOLD generate multiple structures, we calculate the average score for all folds for each molecule.

We had several discussions over which scoring system was most appropriate. Previously, we included a penalty for incorrect predictions into our scoring system, $\frac{c-i}{t}$, were c was the number of correctly predicted base pairs, i was the number of incorrectly predicted base pairs and t was the number of base pairs in the actual structure. Under this scoring system, perfect folds would have a score of 1.0 and inaccurate folds could have a negative score. We also considered scoring systems that would give partial credit to stems that were displaced by a nucleotide or two, since these stems would be an approximation of the overall secondary structure.

Eventually, we decided to adopt the simpler scoring system that does not penalize incorrect stems.

This method allowed us to easily understand how accurate our folds were. This decision was also based on the assumption that increasing the percentage of correct stems would prevent incorrect stems from forming. Down the road, if we achieved a high accuracy rate for the simulation using this scoring system, it would make sense to adopt the scoring system with penalties to further fine tune the simulation.

6.2 MFOLD Performance

To assess the current state of the art in RNA secondary structure prediction, we first present the results of folding several types of RNAs from a variety of organisms. We folded each molecule with version 3.1 of MFOLD, using the default parameters. The results are summarized in table 6.1.

RNA	**Length**	**Number of Structures**	**Mean Score**
S.cerevisiae Ala tRNA	76	1	.95
S.cerevisiae Phe tRNA	76	2	.595
S.cerevisiae 5S	118	2	.803
E.coli 5S	120	20	.27
B.subtilis 5S	118	4	.421
S.cerevisiae RNAse P	369	17	.442
R.prowazekii RNAse P	385	16	.482
B.subtilis RNAse P	401	18	.501
B.subtilis 16S	1552	16	.505
C.pneumoniae 16S	1554	21	.441
E.coli 16S	1542	27	.479
R.prowazekii 16S	1502	24	.505
B.subtilis 23S	2927	37	.481
E.coli 23S	2904	43	.492

Table 6.1: MFOLD benchmarks

As expected, MFOLD offers good performance on smaller RNAs, such as tRNA and 5S RNAs. On both tRNAs that we tested, MFOLD generated folds that had at least 95% of the base pairings

correct. Of the 5S molecules, there is a noticeable performance gap between *S.cerevisiae* and *B.subtilis* and *E.coli*. Upon inspecting the correct secondary structures and sequences of these molecules, we determined that this is due to non-canonical base pairings in several stems, which MFOLD does not handle. Since 5S RNA molecules typically have five stems, disallowing any one of them due to base pairing considerations has a significant performance impact.

Aside from the smaller molecules, we were surprised by MFOLD's consistent performance on the larger RNAs. While 23S RNAs have approximately nine times as many nucleotides as RNAse P molecules, there was no significant degradation in performance across the 16S, 23S and RNAse P molecules. Lastly, the folds took a reasonable amount of time, with the longest 23S molecules lasting about an hour or two to compute forty folds.

6.3 ProFold Performance

The following presents the best scores for our simulation on a variety of RNAs. These scores represent the average score for a number of folds given a set of parameters: the form stem probability (FSP), the break stem probability (BSP) and the number of iterations in cleanup mode. These results represent the set of parameters with the best average score.

While our initial results with smaller molecules looked promising, our accuracy on 16S RNAs dropped off dramatically, and are approximately one-fifth as accurate as MFOLD on the same molecules. We hypothesize that this is due to the combinatorial explosion in the number of possible form stems. Since our algorithm forms stems stochastically, it would be more susceptible to this problem than MFOLD, which minimizes a known energy function. Due to the disappointing performance on 16S molecules, we did not perform extensive tests on 23S rRNAs, which have

RNA	Length	Mean Score	# of Folds	FSP	BSP	Cleanup Iterations
S.cerevisiae 5S	118	.78	20	.35	.8	2000
E.coli 5S	120	.316	20	.55	.9	2000
B.subtilis 5S	118	.347	20	.55	.8	2000
S.cerevisiae RNAse P	369	.388	10	.580	.85	2000
R.prowazekii RNAse P	385	.236	10	.75	.4	2000
B.subtilis RNAse P	401	.181	10	.55	.4	2000
B.subtilis 16S	1552	.124	10	.950	.2	1000
E.coli 16S	1542	.089	10	.950	.3	1000

Table 6.2: Best ProFold scores

approximately twice as many nucleotides as the 16S RNAs.

The results in 6.3 show a range of parameters. To conduct a fairer test, we chose a set of "good" parameters and folded several molecules with them. This is more of a real-world test, since trying a wide range of parameters would not be useful on an unknown RNA. Based on the results of the previous test, we set the form stem probability to .55, the break stem probability to .8 and the number of iterations in cleanup mode to 2000. Each molecule was folded ten times, and the average was calculated.

RNA	Length	Mean Score	MFOLD Score
S.cerevisiae 5S	118	.686	.803
E.coli 5S	120	.245	.27
B.subtilis 5S	118	.347	.421
S.cerevisiae RNAse P	369	.240	.442
R.prowazekii RNAse P	385	.133	.482
B.subtilis RNAse P	401	.127	.501
B.subtilis 16S	1552	.124	.505
E.coli 16S	1542	.059	.479

Table 6.3: Mean ProFold scores with FSP = .55, BSP = .8, Cleanup = 2000

6.4 Effect of Cleanup Mode Length

While transcription mode is determined by the length of our model, the length of cleanup mode is set by an arbitrary parameter to our model. To understand how the length of cleanup mode effects the accuracy of the fold, we we ran our parameter evaluation program with varying lengths of cleanup mode. Each combination of parameters and cleanup mode length was folded twenty times and the average score was computed. The results of this experiment are summarized in table 6.4.

RNA	**0**	**100**	**250**	**500**	**1000**	**2000**
S.cerevisiae 5S	.404	.546	.657	.710	.716	.783
E.coli 5S	.180	.268	.293	.308	.300	.316
B.subtilis 5S	.180	.283	.256	.306	.321	.347
S.cerevisiae RNAse P	.151	.212	.242	.271	.316	.311

Table 6.4: Effect of cleanup mode length

In all these cases, the length of cleanup mode has a positive correlation with the score of the foldings. A cleanup mode with 2000 repetitions is close to twice as accurate as folding the molecule without cleanup mode. Interestingly, the effect seems to level off, with most of the gains occurring between 500 and 1000 repetitions.

6.5 Effect of Non-Progressive Algorithm

To conclusively determine whether transcription mode had a positive effect on folding accuracy, we ran our parameter evaluation program with two folding algorithms, transcriptional and non-transcriptional. The transcriptional algorithm was the standard ProFold procedure with a cleanup

mode lasting 1000 iterations. The non-transcriptional algorithm folded the whole RNA for 1000 iterations of cleanup mode. Since RNAse P molecules have approximately 400 nucleotides, transcription mode would have a much larger effect. For these tests, we compared transcription mode and 1000 iterations of cleanup mode against $1000 + L$ runs of cleanup mode, where L was the length of the RNA. We folded each combination of parameters twenty times to increase statistical precision and computed the average score, based on percentage of base pairings correctly predicted. The results for both the transcriptional and non-transcriptional algorithms are summarized in table 6.5.

Algorithm	**RNA**	**Best Mean Score**	**FSP**	**BSP**
Progressive	*E.coli* 5S	.3	.25	.9
Progressive	*B.subtilis* 5S	.321	.85	.9
Progressive	*S.cerevisiae* 5S	.716	.45	.8
Progressive	*S.cerevisiae* RNAse P	.315	.35	.8
Progressive	*B.subtilis* RNAse P	.154	.15	.7
Progressive	*R.prowazekii* RNAse P	.226	.55	.4
Non-progressive	*E.coli* 5S	.384	.75	.7
Non-progressive	*B.subtilis* 5S	.363	.25	.8
Non-progressive	*S.cerevisiae* 5S	.77	.75	.8
Non-progressive	*S.cerevisiae* RNAse P	.308	.35	.9
Non-progressive	*B.subtilis* RNAse P	.181	.55	.4
Non-progressive	*R.prowazekii* RNAse P	.261	.95	.6

Table 6.5: Progressive vs. non-progressive folding algorithm results

Our results contradict our initial hypothesis. In each case, except for *S.cerevisiae* RNAse P, the non-transcriptional algorithm had better results. In the case where the transcriptional algorithm was better, it only outperformed the non-transcriptional algorithm by 2%. We think these results are consistent with the idea that our algorithm is essentially a simulated annealing version of Nussinov's algorithm, in that it maximizes the number of long stems that are close together along the backbone. We hypothesize that transcriptional mode hampers accuracy because it forms smaller

stems early on which later must be broken to form larger stems.

6.6 Combining MFOLD and Annealing

To further evaluate our algorithm's performance as a simulated annealing technique, we tried feeding it folds generated by MFOLD. We wanted to see if annealing the secondary structure using our probability models for breaking and forming stems could improve on the fold generated by MFOLD.

Each structure was annealing five times with the form stem probability set at .5 for 1000 iterations. We ran trials with three different values, .1, .5, and .9 to cover the full range of possible break stem probabilities. The results are summarized in table 6.6.

RNA	**Size**	**MFOLD Score**	**BSP = .1**	**BSP = .5**	**BSP = .9**
S.cerevisiae 5S	118	.892	.892	.811	.676
E.coli 5S	120	.250	.250	.250	.280
S.cerevisiae RNAse P	360	.526	.538	.503	.245
B.subtilis RNAse P	401	.535	.460	.103	
B.subtilis 16S	1552	.5021	.505	.426	.026

Table 6.6: MFOLD output with 1000 iterations of annealing

When the break stem probability was set to .5 or .9, annealing produced a less accurate structure, especially in the latter case. When the break stem probability was set to a very low value, .1 annealing was able to improve the score a bit. We think this is because the low break stem value prevented almost all stems from breaking and then any other possible stems formed. However, these annealed folds also contained a large number of incorrectly predicted stems, which our algorithm

does not penalize. We think that the low signal to noise ratio prevents annealing from being a useful technique for complementing MFOLD.

6.7 Optimization

When I began working on this research project, the major bottleneck was determining good parameters for the simulation. Performing ten folds for each combination of four parameters was estimated to take two years for an RNAse P molecule [7]. To make the parameter evaluation process more tractable, we invested a lot of time optimizing the code. These optimizations included creating the RNACube data structure and our modifications to Dijkstra's algorithm.

To test the speedup resulting from our optimization efforts, we folded a set of molecules ten times with each version of the code and took the average time. Both trials had the break stem probability set at .9 and the diffusion rate set at 4.0. The optimized version of the code had the form stem probability set at .75 and cleanup mode set at 500 iterations. The original version of the code had the form stem probability set at 80.00 and the cutoff probability set at .1. These differences resulted from our changes to the form stem probability model and cleanup mode. All tests were performed on an Intel XEON operating at 2.8 GHz. The results are summarized in table 6.7. Our

RNA	**Length**	**Original**	**Optimized**	**Speedup**
S.cerevisiae Ala tRNA	76	1.0	1.36	.74
E.coli 5S	120	11.87	6.71	1.77
S.cerevisiae 5S	118	6.52	5.87	1.11
S.cerevisiae RNAse P	360	823.32	129.21	6.37
B.subtilis RNAse P	401	1090.97	162.5	6.71
S.cerevisiae 16S	1800	14321.19	139582.37	9.74

Table 6.7: Optimization results

optimizations provide significant speedup that increases as with the size of the molecule. These effects were most pronounced on the larger RNAse P and 16S RNA molecules, where the speedup was approximately 6 and 10 times, respectively. The original version of the code was faster on the tRNA and was more competitive on the smaller 5S RNAs. We attribute this to overhead in initializing the RNACube and the new version of cleanup mode. In the old algorithm cleanup mode and pre-cleanup mode operated stochastically, with pre-cleanup mode terminating when the queue did not contain any actions whose probability exceeded the cutoff probability and cleanup mode terminating when there were no possible form stem actions left in the queue. These factors led to a wide range of runtimes for the smaller molecule, with the times to fold the tRNA ranging from (.081-3.218) and for the *e.coli* 5S from (2.666-21.372).

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis work we evaluated a new approach for RNA secondary structure prediction. We hoped improve on current methods by developing an algorithm that simulated how the RNA might fold as it is transcribed. Additionally, our heuristic approach allows us to predict a common secondary structure feature, psuedoknots, that other algorithms ignore.

Our results do not support our initial hypothesis. While initially promising, our simulation suffered a marked decrease in performance on larger molecules, such as 16S RNAs. Additionally, we were not able to find a clear-cut set of parameters that worked the best. Further testing showed that our algorithm got the most benefit out of "cleanup mode," which simulated how the molecule might fold once it has been fully transcribed. We think that this is similar to a simulated annealing version of Nussinov's algorithm, in that our form stem model gives higher probabilities to larger stems. Repeatedly forming and breaking stems would cause the molecule to tend toward a state

that approaches the maximum number of base pairs, since larger stems are less likely to break. This conclusion is supported by the fact that the break stem probability tended to be extremely high $\approx .8 - .9$. These values indicate that each hydrogen bond would be "breathing" far more often than what makes sense biologically.

We think that considering more biophysical factors could help create a more accurate transcriptional folding simulation. This could include more sophisticated models for breaking stems based on torsional forces, incorporating more three-dimensional information, and considering the effect of pausing due to transcription factors. Despite our disappointing results, we feel we have created a good first step by creating a modular, object-oriented architecture and visualization tools that can be used to evaluate new folding algorithms and probability models.

7.2 Future Work

The results of this thesis suggest several directions for future research. First, we could improve our current method for generating FormStemActions by incorporating the different energy parameters used in MFOLD. This could provide more accurate probabilities for forming different stems, since the MFOLD parameters distinguish between different types of secondary structural features.

The simulation also needs a more meaningful model for breaking stems. The current model, based on hydrogen bonds "breathing" only works when the probability that a bond is breathing is much higher than what makes sense biologically. A new model might incorporate torsional stresses caused by the folding process to provide a more accurate representation of the underlying biophysics.

Our simulation can easily be extended to model protein interactions during folding. Protein inter-

actions may play an important part in RNA folding by preventing stretches of the molecule from forming base pairs, or bringing parts of the RNA together to catalyze base pairings. These interactions could be modeled in our simulation by increasing or decreasing the probability of certain stems forming. Developing the facility to model protein interactions and testing it on known secondary structures could suggest possible wet lab experiments to better understand the mechanics of RNA folding.

Lastly, we think adapting our probability models into a greedy algorithm is an interesting research question. The greedy algorithm would generate possible stems, and always form the stem with the greatest number of H bonds. This method would be considerably faster than our current simulation.

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] S. J. Baserga and J. A. Steitz. *The diverse world of small ribonucleoproteins*, pages 359–381. Cold Spring Harbor Lab Press, 1993.

[3] G. Beaucage. Polymer physics lecture notes, 1998. Chapter 1: The Isolated Polymer chain. http://www.eng.uc.edu/ gbeaucag/Classes/Physics/.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 581–582, 591–595. McGraw-Hill, second edition, 2001.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 125–133, 136–139. McGraw-Hill, second edition, 2001.

[6] Accelrys Corporation. GCG Wisconsin package. http://www.accelrys.com/.

[7] C. DeZiel. RNA folding simulation documentation, 2001. Dartmouth College.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[9] W. Gilbert. The RNA world. *Nature*, 319:618, 1986.

[10] I.L. Hofacker, W. Fontana, P.F. Stadler, L.S. Bonhoeffer, M. Tacker, and P. Schuster. Vienna RNA package, 1994. ftp://ftp.itc.univie.ac.at/pub/RNA/ViennaRNA-1.03.

[11] N. Larsen and C. Zweib. The signal recognition particle database (srpdb). *Nucleic Acids Research*, 21:3019–3020, 1993.

[12] R.B. Lyngsø and C.N.S. Pedersen. Pseudoknots in RNA secondary structures. In *RECOMB00: Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, Tokyo, Japan, 2000. ACM.

[13] D.H. Mathews, J. Sabina, M. Zuker, and H. Turner. Expanded sequence dependence of thermodynamic parameters provides robust prediction of RNA secondary structure. *Journal of Molecular Biology*, 288:911–940, 1999.

[14] E.S. Maxwell and M.J. Fournier. The small nucleolar RNAs. *Annual Review of Biochemistry*, 64:897–934, 1995.

[15] M McKeown. Alternative RNA splicing. *Annual Review of Cell Biology*, 8:133–155, 1992.

[16] O. Melefors and M. W. Hentze. Translational regulation by mRNA/protein interactions in eukaryotic cells: ferritin and beyond. *BioEssays*, 15:85–90, 1993.

[17] E.B. Mougey, M. O'Reilly, Y. Osheim, O.L. Jr Miller, A. Beyer, and B. Sollner-Webb. The terminal balls characteristic of eukaryotic rRNA transcription units in chromatin spreads are rRNA processing complexes. *Genes and Development*, 7:1609–1619, 1993.

[18] R. Nussinov, G. Piecznik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35:68–82, 1978.

[19] T. Pan, I. Artsimovich, X.W. Fang, R. Landick, and T.R. Sosnick. Folding of a large ribozyme during transcription and the effect of the elongation factor NusA. *Proceedings of the National Academy of Sciences, USA*, 96:9545–9550, Aug 1999.

[20] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences, USA*, 85:2444– 2448, 1988.

[21] S.W. Peltz and A. Jacobson. mRNA stability: in trans-it. *Current Opinion in Cell Biology*, 4:979–983, 1992.

[22] E. Rivas and S.R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285:2053–2068, 1999.

[23] E. Rivas and S.R. Eddy. The language of RNA: A formal grammar that includes pseudoknots. *Bioinformatics*, 16:334–340, 2000.

[24] P. Schuster, W. Fontana, P.F. Stadler, and A. Renner. RNA structures and folding: from conventional to new issues in structure predictions. *Current Opinion in Structural Biology*, 7:229–235, 1997.

[25] B. A. Shapiro and J. C. Wu. An annealing mutation operator in the genetic algorithms for RNA folding. *Computer Applications in the Biosciences*, 12:171–180, 1996.

[26] J.E. Tabaska, R.B. Cary, H.N. Gabow, and G.D. Stormo. An RNA folding method capable of identifying pseudoknots and base triples. *Bioinformatics*, 14:691–699, June 1998.

[27] M. Tompa. Lecture notes on biological sequence analysis, 2000. Lecture 16: RNA Secondary Structure Prediction. http://www.cs.washington.edu/education/courses/517/00wi/.

[28] D. H. Turner, N. Sugimoto, J. A. Jaeger, C. E. Longfellow, S. M. Freier, and R. Kierzek. Improved parameters for prediction of RNA structure. In *Cold Spring Harbor Symposia Quantitative Biology*, volume 52, pages 123–133, 1987.

[29] F.H.D. van Batenburg, A. P. Gultyaev, and C. W. A. Pleij. An APL-programmed genetic algorithm for the prediction of rna secondary structure. *Journal of Theoretical Biology*, 174:269–280, 1995.

[30] C.R. Woese and N.R. Pace. *Probing RNA structure, function, and history by comparative analysis*, pages 91–117. Cold Spring Harbor Lab Press, 1993.

[31] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244:48–52, 1989.

[32] M. Zuker and D. Sankoff. RNA secondary structures and their prediction. *Bulletin of Mathematical Biology*, 46:591–621, 1984.