Fall 2012

Algorithms For Predicting Secondary Structures Of Human Viruses

Hardik Shah
San Jose State University

Algorithms For Predicting Secondary Structures Of Human Viruses

A Writing Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Hardik Shah
Advisor: Dr. Sami Khuri
December 2012

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

ALGORITHMS FOR PREDICTING SECONDARY
STRUCTURES OF HUMAN VIRUSES

by

Hardik Shah

APPROVED FOR THE DEPARTMENT OF
COMPUTER SCIENCE

Dr. Sami Khuri, Department of Computer Science, SJSU	Date
Dr. Mark Stamp, Department of Computer Science, SJSU	Date
Natalia Khuri, Bioengineering and Therapeutic Sciences, UCSF	Date

ABSTRACT

The mechanism and role of RNA secondary structure elements in the replication and translation of human positive-strand RNA viruses remains poorly understood. These secondary structures are formed when a single RNA strand folds over and base pairs with itself, forming various types of loop structures. RNA strands fold into specific shapes. This unique shape for each nucleic acid chain is the most stable state it can adopt. The lower the energy, i.e., the fold with highest number of base pairs, the higher the stability of the structure.

The Dynamic Programming technique, such as the one used in Nussinov-Jacobson algorithm, predicts the locations to fold the sequence to give us an optimal solution. But, the Nussinov algorithm does not necessarily generate the most stable structure and may produce scattered matches that are not biologically relevant. More complex algorithms are needed to solve this problem. Hence, we study Zuker's energy minimization algorithm that uses thermodynamic details with dynamic programming principles at the core. Nussinov-Jacobson and Zuker algorithms give the maximum number of base pairs that the given RNA molecule might have upon folding onto itself.

We analyze the outputs produced by both algorithms for small subsequences and compare the predicted structures. Using a sliding window approach, we focus on specific parts of RNA and analyze their structure. Studying the genomes of RNA viruses will give an insight into the nucleotide positions that determine the virulence of the different virus strains.

ACKNOLEDGEMENT

I am thankful to Dr. Sami Khuri for teaching me the value of punctuality. The Algorithms and Bioinformatics courses were as instructive as the numerous meetings filled with anecdotes about travel, life, and bicycles.

I would also like to thank Natalia Khuri for being the best mentor I could have asked for my project. Her kindness and clarity of thoughts are a source of inspiration to me.

I want to express my gratitude to Dr. Mark Stamp for his time and teaching my most favorite course- CS 265.

I want to especially thank my friend Rohit Garg for suggesting and lending me the best reference books on the subject.

Table of Contents

List of Figures

List of Tables

1 Introduction

This chapter describes RNA, its structure, and computational algorithms that predict RNA secondary structures.

1.1 Nucleic Acids

RNA is more primitive than DNA. DNA is only a store of genetic information. Before DNA evolved, RNA carried genetic information. While DNA fulfills this function, DNA itself still needs to be copied into RNA through the process of transcription for a gene to be expressed. [1]

RNA can play many roles in cells: they are carriers of the stored genetic information (e.g. mRNA), regulatory molecules (e.g. microRNAs), guides or templates (e.g. telomerase), and enzymes (e.g. RNase P). Some of the worst plagues that affect humanity are RNA pathogens (e.g. viruses that cause AIDS, influenza, polio, and dengue fever).

1.2 Structure of RNA

RNA is a globular, single stranded structure. RNA is a polymer of repeating monomer units called nucleotides. A nucleotide is composed of sugar, base, and phosphate groups. They are Adenosine, Guanosine, Cytidine, and Uridine. [1]

a) Sugars:

Ribose sugars are five-membered ring-shaped molecules containing carbon atoms and a single oxygen atom with side groups attached to the carbons. The carbon atoms are numbered 1' through to 5'.

The additional 2' –OH group in the ribose sugar has two consequences for the function of RNA:

1. The 2' –OH group in ribose sugar is polar that makes RNA chemically more reactive than DNA.

2. The ribose sugar molecule is slightly twisted to minimize interactions between the polar 2' –OH group and the other non-bonding atoms attached to the ring. This

twisted shape has implications for the kinds of secondary structures that RNA forms compared with DNA. [1]

b) Bases:

RNA is made from four different nucleotides. Each contains different bases connected to ribose sugar. The four bases form two chemically distinct groups called purines and pyrimidines. Both purine and pyrimidine bases are ring-shaped molecules containing carbon and nitrogen atoms.

Purine bases are Adenine and Guanine. They contain a double ring with nine carbon atoms. Pyrimidine bases are Uracil and Cytosine. They contain a single ring. Hence, pyrimidines have less (six) carbon atoms than purines.

A clear difference between RNA and DNA is that RNA uses uracil as a base whereas DNA uses thymine. The reason is related to the chemical stability of nucleotides and the repair of nucleic acid damage in the cell. [1]

1.3 RNA Structural Organization

RNA molecules can fold into complex three-dimensional structures with three hierarchical levels of organization. These three levels are [1]:

1. Primary Structure: This is the linear sequence of nucleotides in RNA.

2. Secondary Structure: There are helices that form through base pairing. RNA helices form within single molecules of RNA (intra-molecular base pairing) and between different RNA molecules (intermolecular base pairing).

3. Tertiary Structure: RNA molecules fold up into very compact and highly organized structures.

1.4 Functions of RNA

DNA acts as a store of genetic information only. RNA forms more complex structures than DNA. The ability to fold into diverse structures enables RNA to be involved in a number of biological processes. Some of them are as follows [1]:

1. Store of genetic information: Some viruses have RNA genomes. E.g. Poliovirus.

2. Template function: Telomerase RNA acts as template for making new DNA ends of chromosomes.

3. Carrier of genetic information: mRNA carries out this function.

RNA is much more versatile. This makes it really interesting to study.

1.5 Algorithms for Predicting RNA Secondary Structures

The three dimensional structure of an RNA molecule is determined by the information held inside the sequence of nucleotides. Analyzing the secondary structure of the nucleotide sequence may provide an insight into first draft of the molecule. [2] Hence, predicting the secondary structure given a primary sequence of RNA is crucial. Many RNA families conserve their secondary structure more than they conserve their primary sequence. [3] Many RNAs or functional elements in RNAs can only be identified by comparative analysis of secondary structure. Also, the structure of molecules is conserved across many species and may be used to infer phylogenetic relationships and to determine two-dimensional and three-dimensional structure. [3]

1.5.1 Nussinov Algorithm

This is a dynamic programming algorithm that calculates maximum number of base pairs in a folded RNA molecule. Nussinov algorithm is simple and acts as a basis for all the other advanced RNA prediction algorithms.

The algorithm is designed to evaluate the contribution of individual base pairs to the secondary structure of a polynucleotide chain. [4] Consider a sequence of n nucleotides $S_1 \ldots S_n$. To identify the structure with the maximum number of base pairs, the scoring system adds one per base pair to the score, adds zero for anything else. [5] The optimal score, $S(i, j)$, of a subsequence of the RNA from position i to position j, can be defined recursively in terms of optimal scores of smaller subsequences. [3]

There are only four possible ways that a structure of nested base pairs can be constructed (See Figure 1): [5]

1. *i and j* base pair, added on to a structure for $i+1, j-1$.

2. *i* is unpaired, added onto a structure for *i + 1, j.*

3. *j* is unpaired, added onto a structure for *i. j – 1.*

4. *i and j* base pair but not to each other; the structure for *i..j* adds together sub-structures for two sub-sequences, *i..k* and *k + 1..j* (a bifurcation).

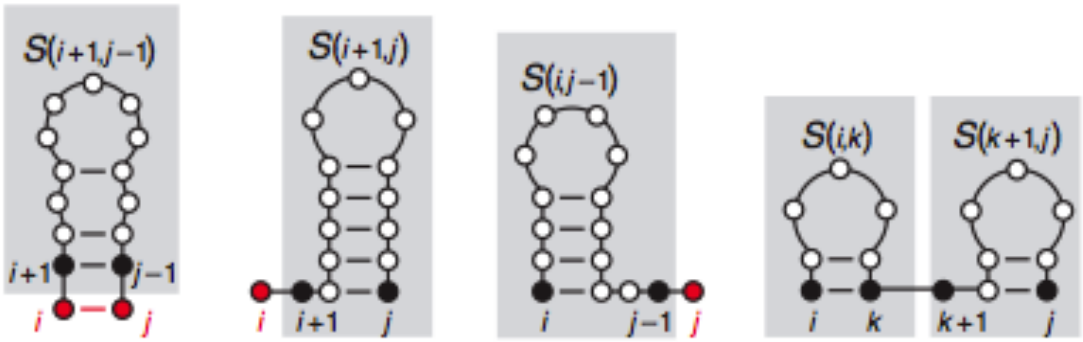

Figure 1: Four possibilities of the recursive definition [5]

The algorithm uses the following mathematical recursion:

$$
\begin{aligned}
S(i, j) = max \; \{ & \\
& S(i + 1, j - 1) + 1, \\
& S(i + 1, j), \\
& S(i, j - 1), \\
& max_{i<k<j} S(i, k) + S(k + 1, j) \\
& \}
\end{aligned}
$$

Storing the S(i,j) matrix requires memory in the order of N^2 where N is the number of nucleotides in the sequence to be folded. [2] However, the innermost loop of having to find most optimal potential bifurcation points makes the time complexity of the order N^3. [5]

Base pair maximization is an inferior scoring scheme for prediction of RNA secondary structures. Using minimum global energy of a structure is a better scoring scheme and Zuker algorithm uses a dynamic programming approach that incorporates energies of substructures.

1.5.2 Zuker Algorithm

RNA molecules adopt a structure with minimum global energy than the structure with maximum number of base pairs. Zuker's algorithm considers energies of different loops and base pairing interactions.

Consider an *N* nucleotides long sequence, S. The nucleotides of an RNA molecule are referred as vertices in the graph representation. The N-1 arcs of the semicircle between the bases are called exterior edges. Exterior edges represent the phosphodiester bonds between consecutive nucleotides. Chords on the semicircle represent base pairing between two nucleotides. These chords are called interior edges. Edges and vertices combine to form the graph of a RNA sequence. Chords are not allowed to intersect or touch each other as it rules out all the knotted substructures. The free energy of the structure is associated with the regions between bonds. For graphical representation, the energy depends on the faces of the graph. [2]

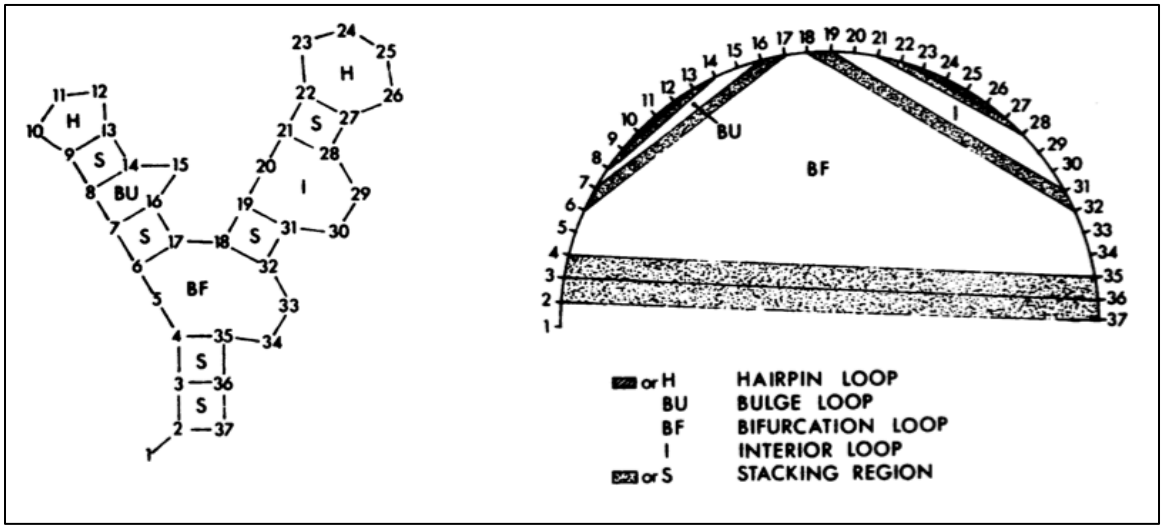

Figure 2: Conventional and Graph representation of a RNA secondary structure [2]

There are overall five types of faces as follows: [2]

1. Hairpin loop: A face with a single interior edge.

The following three types have two interior edges.

2. Stacking Region: Interior edges are separated by single exterior edges on each side.

3. Bulge Loop: Interior edges are separated by a single exterior edge on one side, but by more than one exterior edge on the other side.

4. Interior Loop: Face with exactly two interior edges with no restrictions on exterior edges on sides.

5. Bifurcation Loop: Face with three or more interior edges.

Refer figure 2 for graphical representation of the five faces described above.

Energy of a structure is the sum of the energies of its faces. Zuker algorithm finds a secondary structure having the minimum energy using dynamic programming principles. [2]

The algorithm uses the following recursion [2]:

For all pairs i, j where $1 \leq i < j \leq N$, *define*

W (i, j) = minimum free energy of all possible admissible structures formed by the subsequence $S_{ij.}$

V (i, j) = minimum free energy of all possible admissible structures formed from S_{ij} *in which* S_i *and* S_j *base pair with each other.*

Rules:

$V(i, j) = \infty$ *If* S_i *and* S_j *can't base pair*

$W(i, j) = 0$ *if* $j - i = 4$

Calculation of V (i, j) [2]:

$V(i, j) = \min \{E1, E2, E3\}$

$E1 = E(FH(i, j))$,

where FH(i ,j) is the hairpin loop containing the interior edge between S_i and S_j

E2 = min {E (FL (i, j, i', j')) + V (i', j')},

where i < i' < j'< j and FL (i, j, i', j') is the face containing exactly two interior edges, one between S_i and S_j and other between $S_{i'}$ and $S_{j'}$

E3 = min {W (i + 1, i') + W (i' + 1, j – 1)},

where i+1 < i' < j-2

See Figure 3 for graphical representation.

Calculation of W (i,j) [2]:

W (i, j) = min {

W (i+1, j),

W (i, j-1),

V (i,j),

Min {W (i , i') + W (i'+1, j) }

}

See Figure 3 for graphical representation.

See Appendix A for a detailed manual simulation of Zuker Algorithm.

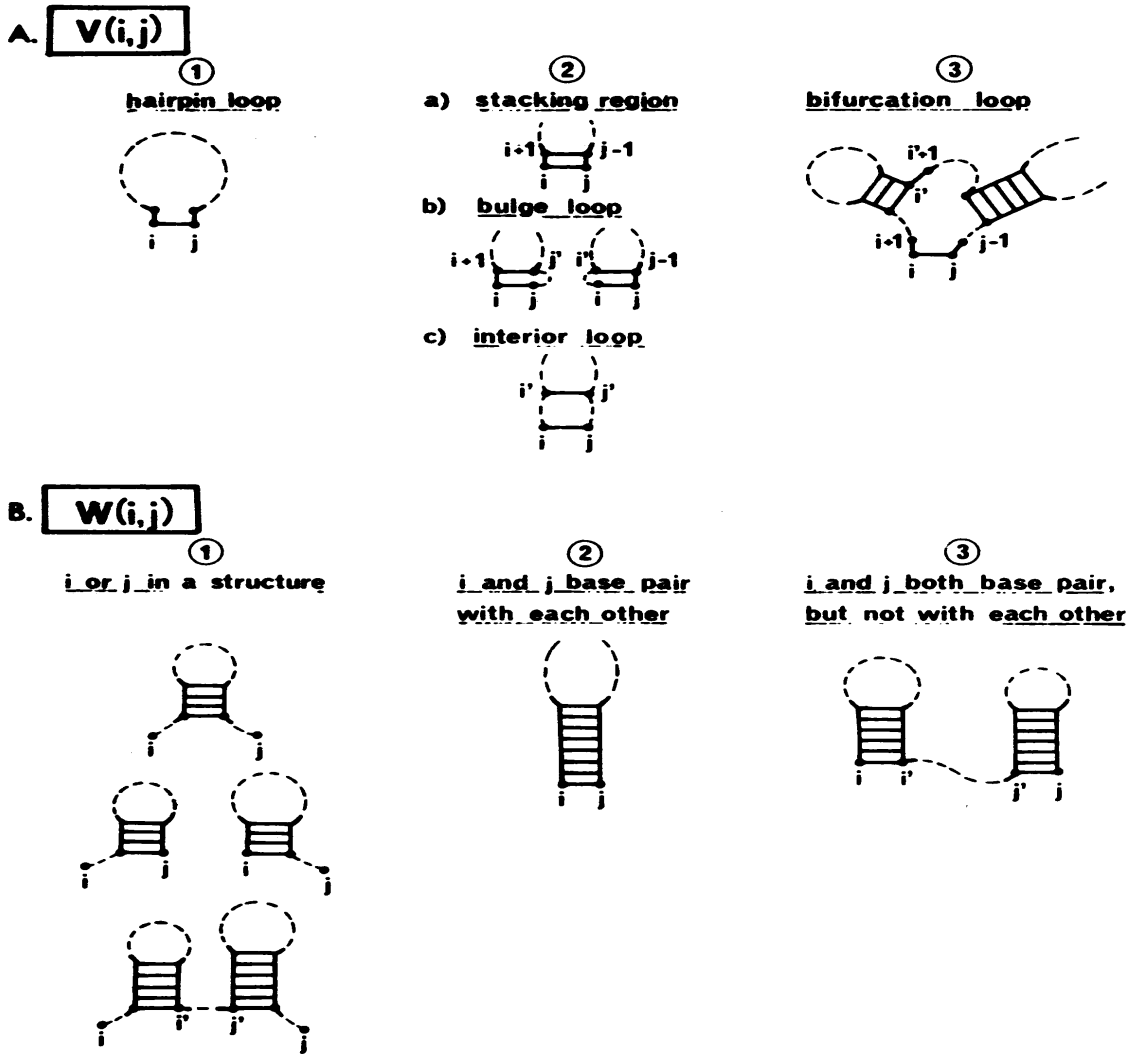

Figure 3: Zuker-Possibilities of the recursion for W and V [2]

Using the Nussinov and Zuker algorithms, we will predict secondary structures of subsequences using a sliding window approach. Knowing the secondary structure of a specific region can help us predict the binding sites for other RNAs or proteins. Using sequence alignment, secondary structures of interesting regions of different virus strains can be compared. The nucleotide variations at the positions can be investigated to see their effect on the combination ability of virus to the host and virus RNA translation efficiency.

In the next chapter, we will discuss the scope and limitations of this project.

2 Scope of the project

This chapter defines the scope and limitations of the algorithms and the software package implemented in this project.

Tracing back a secondary structure is implemented but not discussed in this report. Many secondary structures may have the same score. Only the optimal score is used for analysis.

A pseudoknot forms by base pairing between a single-stranded region of RNA and a loop. Pseudoknots break the recursive definition of the optimal score *S (i,j)* and energy functions. Hence, Nussinov and Zuker algorithms can't handle pseudoknots. [5]

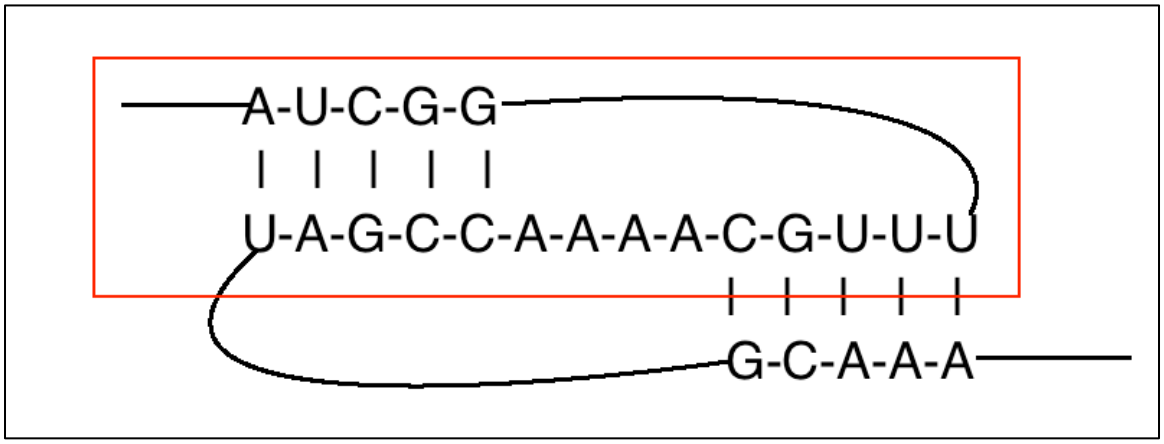

Figure 4: Pseudoknots

For example, see Figure 4. The stem and loop structure inside the rectangle has 5 base pairs. The algorithms will recursively calculate the score or energy assuming the nucleotide subsequence UUUGCAAAA has no base pairing. If you add pseudoknot base pairs (C-G, G-C, U-A, U-A, U-A) onto the sub-sequence (AUCGGUUUGCAAAACCGAU) then, the algorithms can't add the score or energy, as they don't keep track of individual nucleotides. The algorithms just add the score or energy on a previously calculated subsequence. [5]

Nussinov algorithm only uses the basic folding rules. Zuker algorithm uses basic folding rules and thermodynamic information. To predict an accurate secondary structure,

more information is necessary irrespective of the computing power available. This is a limitation of the project.

The Nussinov algorithm can execute on any window size but it takes a lot of time if the window size is greater than 300. The attached code has set the limit to 800 for the maximum allowed window size.

Zuker algorithm is more complicated and more computationally intensive than Nussinov algorithm. The energy values used in the software can only handle window size of less than 35. Replacing the energy values with more comprehensive values can easily reduce this limitation.

In the next chapter, we will discuss the high level design and software framework of this project.

3 Software Frameworks and Design

In this chapter, we describe the high level design of the project and all the modules involved. There are four important steps in the software as shown in Figure 5.

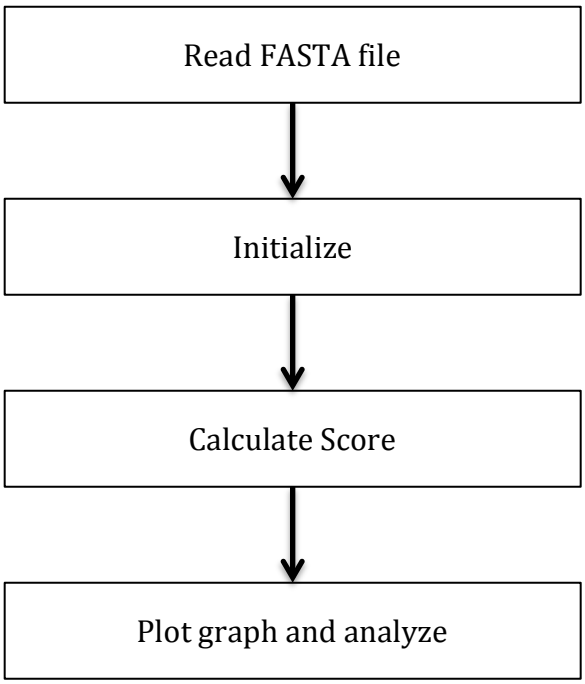

Figure 5: High Level Design of Project

3.1 Common Modules

We only allow the FASTA format for input RNA sequences. Hence, input module is the same for both algorithms. The input module will copy the contents of a FASTA file into a string, remove the comment from the sequence, convert all the nucleotide characters into uppercase, and replace all 'T's with 'U's. The output is exported as a table in Microsoft Excel for further analysis.

3.2 Algorithm Specific Modules

Initialization module is different for both algorithms. Initialized values are useful as they are used to check for terminating a recursive call in the program. Nussinov algorithm uses a matrix to store the scores for each subsequence. Zuker algorithm needs

to initialize two matrices, since the traceback is more involved than with Nussinov algorithm.

Nussinov algorithm calculates the number of maximum possible base pairs in a sequence. This score is calculated recursively and stored in an N*N table (where N is number of nucleotides). This table is useful for making the recursion efficient. When we try to calculate the score S (i,j), we have already calculated the scores of all the smaller subsequences.

Zuker algorithm deals with two N*N matrices, V and W, for calculating minimum global energy of a folded sequence. The W matrix depends on the values from the V matrix. V matrix values are calculated by using various loop modules and the energy table. Using the two matrices, we can easily find base pairs of the folded sequence if needed.

In both scoring modules discussed above, the diagonals of the matrices are initialized to a suitable value so that the recursion never accesses an empty cell of the matrices.

3.3 Implementation Platform

The UNIX platform is used for the implementation of the program. C is the programming language used to develop the software. For analysis of the results, Microsoft Excel is used.

Implementation details will be discussed in the next chapter.

4 Implementation

This chapter discusses the implementation of Nussinov and Zuker algorithms. Both algorithms follow similar steps. First, a FASTA file is read as an input and stored in a string with the necessary input formatting. Second, the user is asked to enter a window size for implementing the sliding window analysis of the algorithms. Third, recursive functions are executed to calculate the maximum score for Nussinov algorithm or the minimum energy for Zuker algorithm. Fourth, the individual output for each window is plotted against the window number for the analysis using Microsoft Excel.

4.1 File Operation Module

Nussinov and Zuker algorithms use the same independent module to read a RNA sequence from FASTA file. FASTA format is a standard for representing nucleotide sequences.

A sequence in FASTA format has two parts. First part, the description line, begins with a greater-than (">") symbol, followed by a single-line description and ends with the UNIX new-line character. The second part, sequence data, consists of *A, C, G, and U/T*. [10] [11]

e.g. >gi|12408699|ref|NC_002058.3| Poliovirus, complete genome
TTAAAACAGCTCTGGGGTTGTACCCACCCCAGAGGCCCACGTGGCGGCTAGTACTCCGGTATTGCG
GTACCCTTGTACGCCTGTTTTATACTCCCTTCCCGTAACT [8]

We convert all *T* to *U* in the input module as we are dealing with RNA sequences only. The above sequence will be stored in a string as follows:

UUAAAACAGCUCUGGGGUUGUACCCACCCCAGAGGCCCACGUGGCGGCUAGUACUCCGGUAUU
GCGGUACCCUUGUACGCCUGUUUUAUACUCCCUUCCCGUAACU

4.2 Implementation of Nussinov Algorithm

We will discuss the initialization and scoring functions in this section. [4][5] Score (i, j) is the matrix that stores the maximum possible number of base pairs in a sequence. The initialization function assigns the value '0' to all the sequences of length 0 and 1. All the other values are initialized to -1 to denote that they are not yet calculated.
Pseudocode:

```
Score[i][i] = 0;
Score[i][i-1] = 0;
```

This also assures that the recursion never accesses an empty cell of the Score matrix.

The scoring function is the heart of Nussinov algorithm and uses recursion explained in the first section of this report. If the nucleotides base pair, the *do_basepair* function (used in the pseudocode below) adds a '1' to the already calculated score. Otherwise, it adds zero.
Pseudocode:

```
calculate_max_score(i, j)
{
   max = score[i+1][j];

   if (score[i][j-1] > max)
      max = score[i][j-1];

   if (score[i+1][j-1] + do_basepair(i,j) > max)
      max = score[i+1][j-1] + do_basepair(i,j);

   for (k = i+1 ; k < j ; k++)
      if (score[i][k] + score[k+1][j] > max)
         max = score[i][k] + score[k+1][j];
   return max;
}
```

4.3 Implementation of Zuker Algorithm

Zuker's initialization function is similar to the one in Nussinov algorithm. Zuker uses two matrices to store energy values namely, V and W. [2] V (i, j) stores the minimum global energy of the subsequence S_{ij} when i and j base pair. W (i,j) stores the

minimum global energy of the subsequence S_{ij} . The V and W values of subsequences with lengths 0 and 1 are initialized to infinity.

Pseudocode:

```
V[i][i] = V[i][i+1] = ∞;
W[i][i] = W[i][i+1] = ∞;
```

Scoring function for Zuker algorithm is a complicated yet beautiful dynamic programming recursion that incorporates conformational rules and thermodynamic details.

Pseudocode for calculating W matrix:

```
calculate_W(i, j) {

if (W[i][j] is calculated) {
    return W[i][j];
  }

  minimum_energy = calculate_V(i, j);

  if (   minimum_energy  >   calculate_W(i+1,j) ) {
    minimum_energy = W[i+1][j];
  }

  else if (   minimum_energy  >   calculate_W(i,j-1) ) {
    minimum_energy = W[i][j-1];
  }

  else{
    for (k = i+1; k < j; k++) {
       if (minimum_energy > calculate_W(i,k) + calculate_W(k+1,j)  ) {
          minimum_energy = W[i][k] + W[k+1][j]; }
         }
  }
  W[i][j] = minimum_energy;
  return minimum_energy;
}
```

Pseudocode for calculating V matrix:

```
calculate_V(i, j)
{
```

```
if(i and j don't base pair) { V[i][j] = infinity; return infinity;}

if (V[i][j] is calculated) { return V[i][j];}
   // case 1: FH(i,j)
minimum_energy = hairpin_loop(i, j);
   // case 2: min[FL(i,j,h,k) + V(h,k)]

   for (h = i+1; h < j; h++) {
      for (k = j-1; k > h; k--) {
         if (h and k base pair) {
            if (h == i+1 && k == j-1) {
                  // Stacking Loop Energy calculation
               temp_energy = stacking_loop(i, j) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;}
            }
            else if (h == i+1 || k == j-1){
                  //Bulge Loop energy calculation
               temp_energy = bulge_loop(i, j, h, k) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;}
            }
            else{
                  // Interior Loop energy calculation
               temp_energy = interior_loop(i, j, h, k) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;}
            }
         }
      }
   }
   // Bifurcation Loop calculation
   for (k = i+2; k < j-1; k++) {
      temp_energy = W[i+1][k] + W[k+1][j-1];
      if (temp_energy < minimum_energy) {
         minimum_energy = temp_energy;}
   }

   V[i][j] = minimum_energy;
   return minimum_energy;
}
```

In the next chapter, we discuss the analysis and results of this project.

5 ANALYSIS AND RESULTS

This chapter discusses the various analyses of the output from the software.

Nussinov algorithm has a time complexity of $O(N^3)$[4] and Zuker has $O(N^4)$[2]. Table 1 shows the execution time (in seconds) required for a given window size. The sequence under consideration is the complete genome of Human Polio virus and it is 7440 base pairs long. [8]

Window Size	Nussinov	Zuker
15	**0.135304**	**0.231193**
20	**0.266494**	**0.455693**
25	**0.448400**	**0.947798**
30	**0.732963**	**1.885217**
35	**1.107857**	**3.887104**
100	**22.029250**	**NA**
200	**165.314504**	**NA**
300	**539.644728**	**NA**
350	**846.215939**	**NA**
700	**6739.113487**	**NA**

Table 1: Execution times for different windows

As the window size increases, the execution time increases exponentially. Window size is the length of a subsequence that is folded by the algorithm each execution cycle.

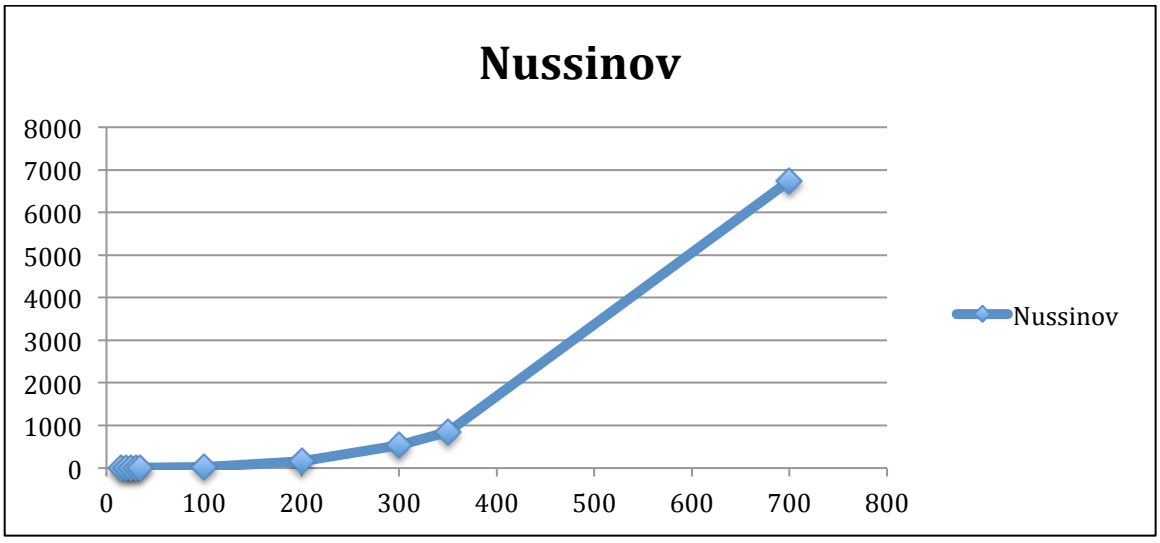

Figure 6: Nussinov execution time for different window sizes

Figure 7 shows the comparison of execution times for Nussinov and Zuker algorithms for different window sizes. Zuker execution time rises more exponentially than Nussinov.

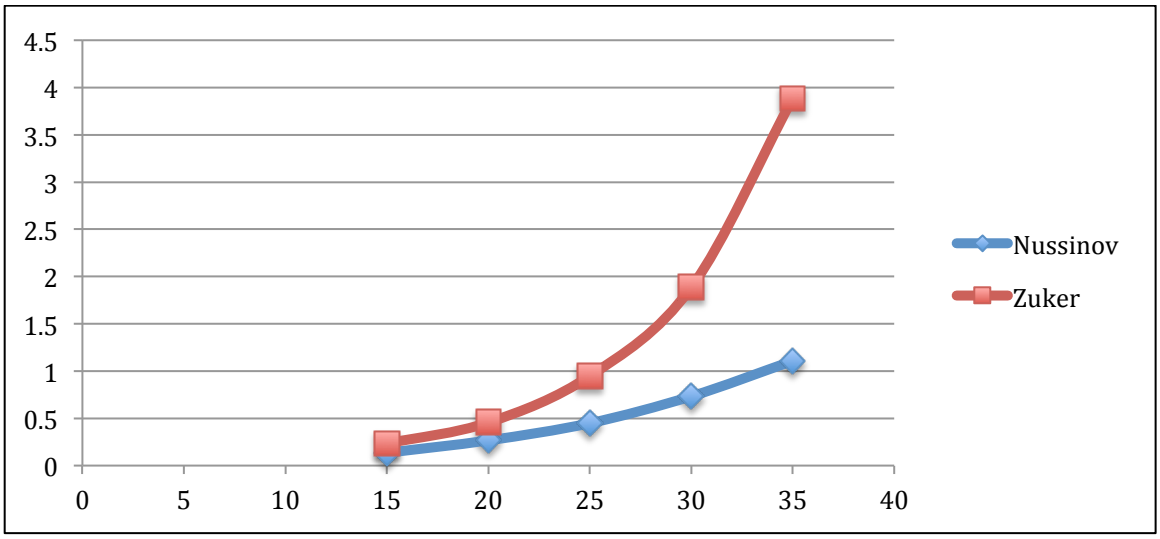

Figure 7: Comparison of execution times for small window sizes

There are many interesting correlations and properties that can be inferred from the graphs. Figure 8 is a graph obtained from the output of Zuker algorithm with window size 30.

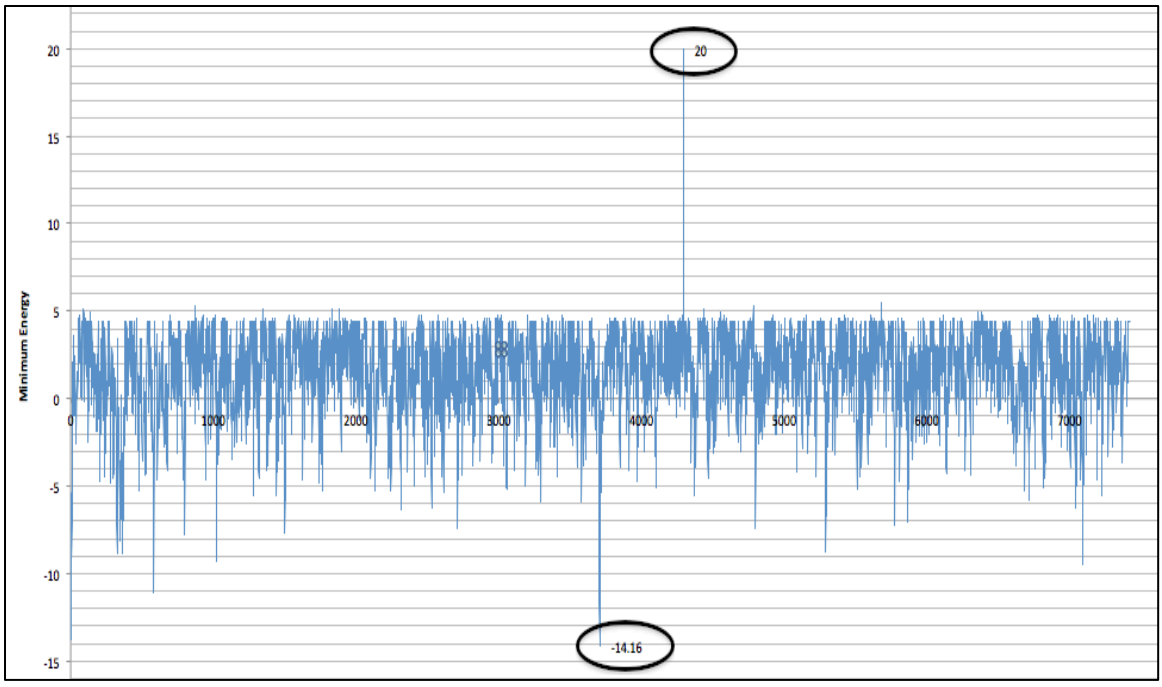

Figure 8: Zuker Output for Window Size 30

In Figure 8, position (3708, -14.16) represents the minimum energy of all the windows for the given sequence.

In simple words, the sequence that starts from position 3709 to 3738, with length 30, if folded with Zuker algorithm will have energy of -14.16 kcal/mole.

The sequence is ”UUGUGGUGGCAUACUCAGAUGUCACCACGG”. It has 6 ‘A’s, 8 ‘U’s, 7 ‘C’s, and 9 ‘G’s.

Theoretically,
Maximum possible number of base pairs = (A-U) + (C-G) + (G-U)
= 6 + 7 + 2 = 15

From a similar graph for Nussinov,
Base pairs for window 3708 = 14

Energy = -14.16 kcal/mole
See Figure 9 for the predicted secondary structure.

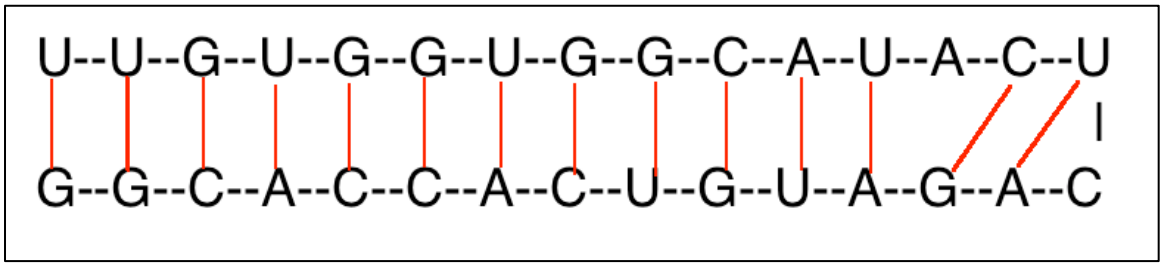

Figure 9: Secondary Structure for E=-14.16kcal/mole

Let’s consider another example. The most impossible structure is of the window number 4294 with energy as infinity. (In Figure 8, it is assigned as 20 for improving readability.)

According to Zuker algorithm,
Sequence that starts from 4295 and ends at 4324, with length 30, is impossible to fold.

The sequence is “AACCAAAUCUCAACUAUACACCAAUCAUGC”. It has 13 ‘A’s, 6 ‘U’s, 10 ‘C’s and 1 ‘G’.

Theoretically,
Maximum possible number of base pairs = (A-U) + (C-G) + (G-U)
= 6 + 1 + 0 = 7

According to Nussinov algorithm output,
Base pairs for window 4294 = 7
See Figure 10 for one of the predicted secondary structure from the output of Nussinov algorithm.

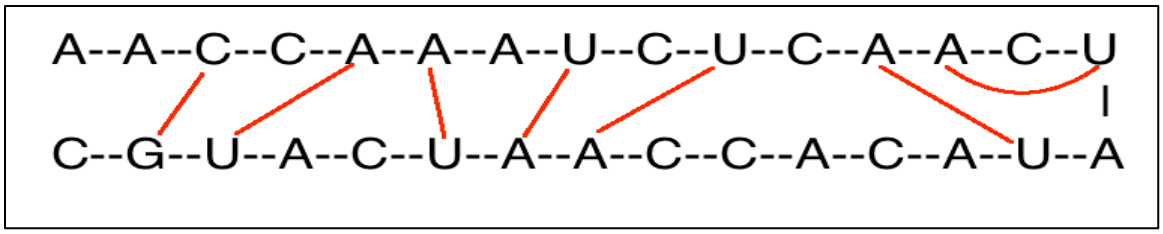

Figure 10: Secondary Structure for 7 Base Pairs

Figure 11 shows the corresponding Nussinov output for the Poliovirus sequence.

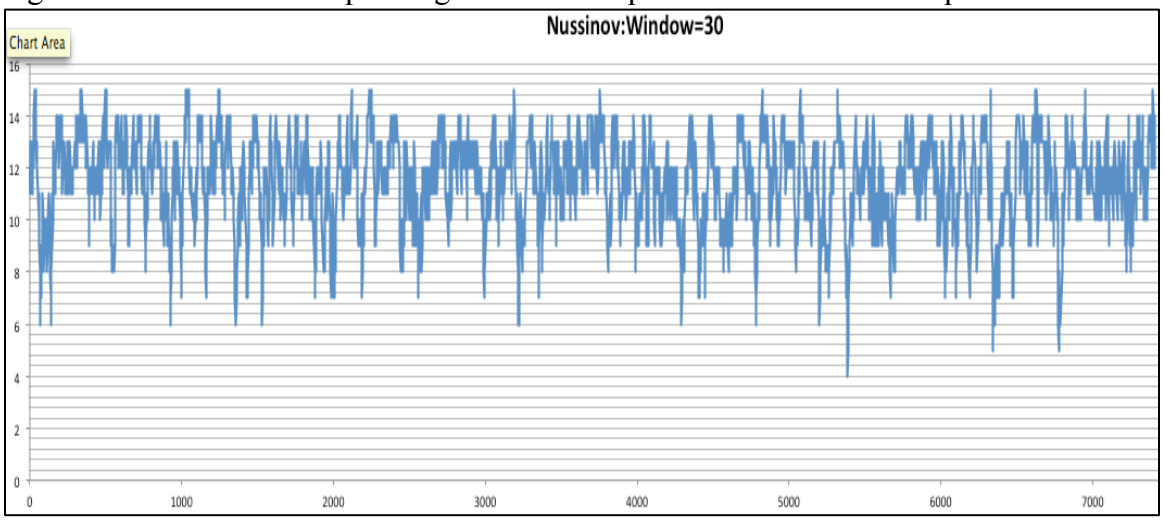

Figure 11: Nussinov output for Window Size 30

Table 2 shows the corresponding window numbers for minimum and maximum outputs generated by both algorithms for the Poliovirus sequence. [8]

Window No.	Nussinov	Zuker	
			Nussinov Max
343	15	-8.12	
7396	15	2.86	
			Nussinov Min
5387	4	2.66	
5388	4	1.86	
			Zuker Min
3708	14	-14.16	
			Zuker Max
4294	7	infinity	
4295	7	infinity	
4296	7	infinity	

Table 2: Minimum and Maximum output values for NC_002058

Figure 12 shows the outputs for each window of the Human Enterovirus B sequence. [7] The top graph plots the Nussinov results and the lower graph plots the Zuker results.

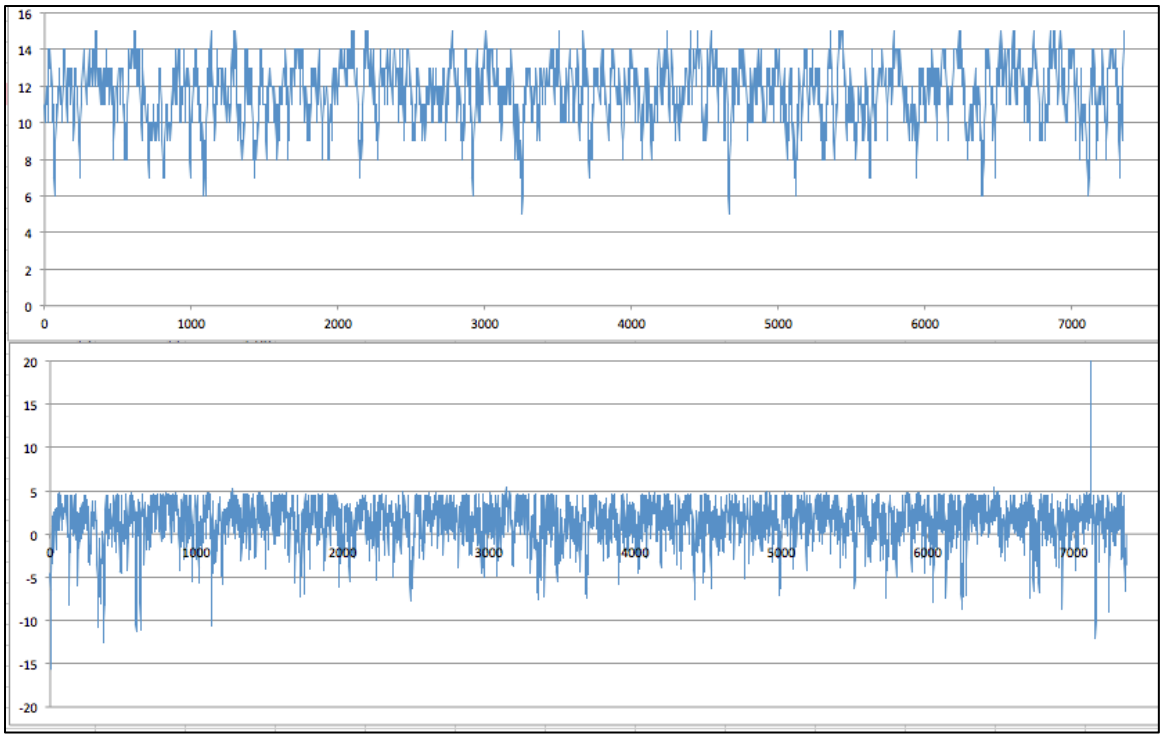

Figure 12: Outputs for Sequence NC_001472

Table 3 shows the corresponding window numbers for minimum and maximum outputs generated by both algorithms for the Human Enterovirus B sequence.

Window No.	Nussinov	Zuker	
			Nussinov Max
3007	15	3.66	
618	15	-11.1399	
			Nussinov Min
3254	5	4.4	
3255	5	4.4	
4663	5	4.4	
4664	5	4.4	
			Zuker Min
5	11	-15.68	
6	11	-15.68	
7	11	-15.68	
			Zuker Max
7112	6	infinity	

Table 3: Minimum and Maximum output values for NC_001472

Figure 13 shows the outputs for each window of the Human Enterovirus A sequence. [6] The top graph plots the Nussinov results and the lower graph plots the Zuker results.

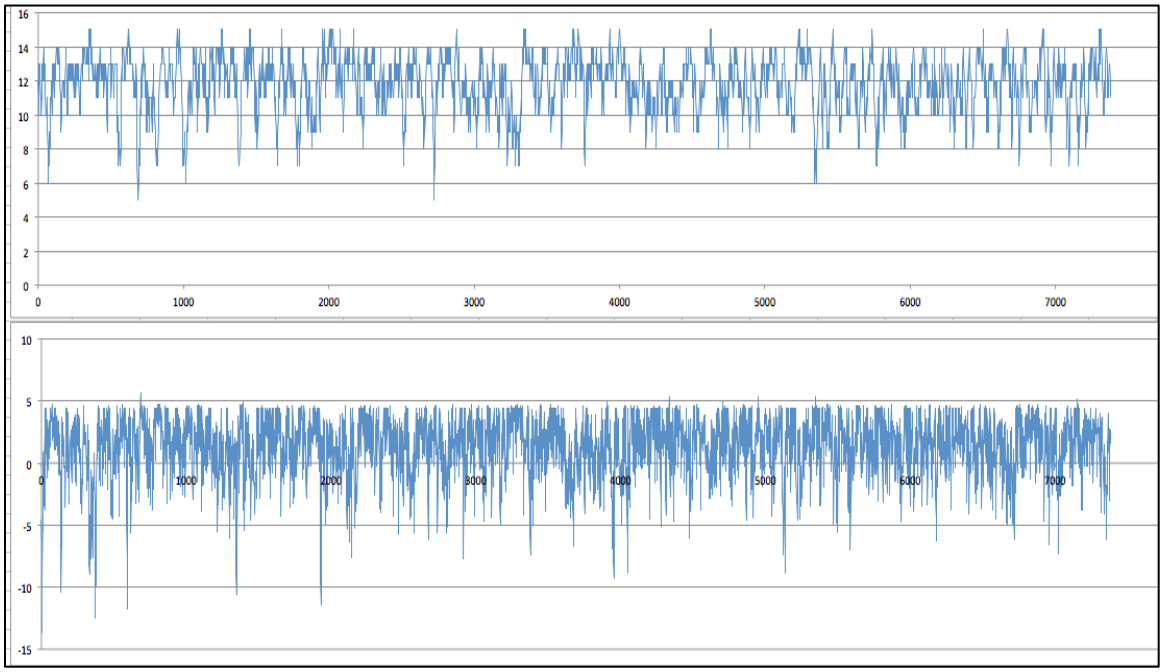

Figure 13: Outputs for Sequence NC_001612

Table 4 shows the corresponding window numbers for minimum and maximum outputs generated by both algorithms for the Human Enterovirus A sequence. [6]

Window No.	Nussinov	Zuker	
			Nussinov Max
351	15	-5.82	
353	15	-5.82	
			Nussinov Min
687	5	4.4	
688	5	4.4	
689	5	4.4	
			Zuker Min
4	12	-13.68	
5	12	-13.68	
			Zuker Max
685	6	5.62	

Table 4: Minimum and Maximum output values for NC_001612

This analysis can be used to study RNA viruses at a comprehensive level as it gives ability to the user to study specific genome regions that are interesting to her.

6 CONCLUSION

We studied two RNA secondary structure prediction algorithms – Nussinov algorithm and Zuker algorithm. We described the high level design of our algorithms, their framework and implementation details. We analyzed the performance of both algorithms.

Nussinov and Zuker algorithms give the mathematically optimal structure but not the most accurate. Many different structures may have the same score. The shortcoming is due to the scoring system and not the algorithm. If more information is added to the scoring system, then the accuracy will improve. But, the underlying dynamic programming principles remain the same.

The implementation only uses FASTA format for input that is common to both algorithms. The output is two columns of numbers that show the score for each window that is then used by Microsoft Excel for further analysis.

The software can't deal with pseudoknots and really large sequences. Thermodynamic energy values are limited and hence, Zuker can deal with only smaller subsequences. This is not a serious limitation as adding extra values to the energy table and using simple linear interpolation can overcome it.

The goal of this project was not to improve the accuracy of the existing algorithms but to look at other properties for exploratory analysis. For this, we implemented a sliding window approach. The user specifies a window size and then, a long sequence is used for a sliding window of subsequences of the given window size. The score of each subsequence is stored and plotted on a graph. Execution times of algorithms were plotted for different window sizes and analyzed.

The programs will be very useful for biologists to study virulence determinants of an RNA family. It can compare specific regions of genomes of various strains and predict the corresponding secondary structures.

7 FUTURE DIRECTIONS

Enterovirus 71 causes the large outsets of hand, foot, and mouth disease. Though it is a self-limiting disease, fatal cases are increasing. [9] I want to study the virulence of EV-71 virus and its strains by comparing the secondary structures of ‘self-limiting’ strains to ‘fatal’ strains.

Adding chemical reactivity properties and phylogenetic data for related sequences will further improve the accuracy of Zuker algorithm.

8 References

[1] Elliott, D. & Ladomery,M. (2004) *Molecular Biology of RNA*. Oxford, UK: Oxford University Press.

[2] Zuker M. & Stiegler, P. (1982) Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information.

[3] Prof. Sami Khuri CS123B Lecture Notes

[4] Nussinov R. & Jacobson A.(1980) Fast algorithm for predicting the secondary structure of single-stranded RNA.

[5] Eddy, S. (2004) How do RNA folding algorithms work? *Nature Biotechnology*. Vol.22, Number 11, 1457-1458.

[6] Human enterovirus A, complete genome - Nucleotide - NCBI. (n.d.). *National Center for Biotechnology Information*. Retrieved December 3, 2012, from http://www.ncbi.nlm.nih.gov/nuccore/NC_001612

[7] Human enterovirus B, complete genome - Nucleotide - NCBI. (n.d.). *National Center for Biotechnology Information*. Retrieved December 3, 2012, from http://www.ncbi.nlm.nih.gov/nuccore/NC_001472

[8] Poliovirus, complete genome - Nucleotide - NCBI. (n.d.). *National Center for Biotechnology Information*. Retrieved December 3, 2012, from http://www.ncbi.nlm.nih.gov/nuccore/NC_002058.3

[9] Li R., Zou Q., Chen L., Zhang H., Wang Y. Molecular Analysis of Virulent Determinants of Enterovirus 71. *PLoS ONE*. 6(10): e26237. doi:10.1371/journal.pone.0026237.

[10] BLAST Help. Retrieved December 3, 2012, from http://www.ncbi.nlm.nih.gov/blast/blastcgihelp.shtml

[11] Yu, Lisa, "Study of RNA Secondary Structure Prediction Algorithms" (2006). Master's Projects. Paper 23. http://scholarworks.sjsu.edu/etd_projects/23

APPENDICES

Appendix A: Zuker manual simulation

	A	C	G	U	U	U	C	G	U
A	∞	∞	∞	4.3	4.4	4.6	∞	∞	0.7
C	∞	∞	∞	∞	∞	∞	∞	2.4	∞
G	∞	∞	∞	∞	4.1	4.1	4.4	∞	2.9
U	4.3	∞	∞	∞	∞	∞	∞	4.4	∞
U	4.1	4.1	4.1	∞	∞	∞	∞	4.2	∞
U	4.1	4.1	4.1	∞	∞	∞	∞	4.1	∞
C	4.1	4.1	4.1	4.1	∞	∞	∞	∞	∞
G	4.1	4.1	4.1	4.1	4.1	4.1	∞	∞	∞
U	0.7	2.9	2.9	4.1	4.1	4.1	∞	∞	∞

W(1,9) = V(1,9)

V(1,9) = FL(1,9,2,8) + V(2,8)

V(2,8) = FL(2,8,3,7) + V(3,7)

V(3,7) = FH(3,7)

Base pairs are:(1,9) (2,8) (3,7)

A nucleotide can't base pair with itself.

Hence,

$V(1,1) = V(2,2) = V(3,3) = V(4,4) = V(5,5) = V(6,6) = V(7,7) = V(8,8) = V(9,9) = \infty$

A nucleotide can't base pair with the neighboring nucleotide.

Hence,

$V(1,2) = V(2,3) = V(3,4) = V(4,5) = V(5,6) = V(6,7) = V(7,8) = V(8,9) = \infty$

Also, rules for base pairing are:

1. Only A-U, G-C, and G-U base pairings are allowed.
2. Pseudoknots are not allowed.

Hence,

As A and G don't base pair,
$V(1,3) = V(1,8) = \infty$

As A and C don't base pair,
$V(1,7) = \infty$

As C and U don't base pair,
$V(2,4) = V(2,5) = V(2,6) = V(2,9) = V(4,7) = V(5,7) = V(7,9) = \infty$

As G, C, and U don't base pair with themselves,
$V(2,7) = V(3,8) = V(4,6) = V(4,9) = V(5,9) = V(6,9) = \infty$

Recursive relations are,

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

V(1, 4) = min {
FH(1,4) = 4.3
FL(1,4,2,3) + V(2,3) = ∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(1, 5) = min {
FH(1, 5) = 4.4
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(3,5) = min {
FH(3,5) = 4.1
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(1,6) = min {
FH(1, 6) = 4.6
FL(1,6,3,5) + V(3,5) = 4.8 + 4.1 = 8.9
∞ (Doesn't have three or more interior edges)
}

V(3,6) = min {
FH(3,6) = 4.1
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(3,7) = min {
FH(3,7) = 4.4
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(4,8) = min {
FH(4,8) = 4.4
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(5,8) = min {
FH(5,8) = 4.4
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(6,8) = min {
FH(6,8) = 4.1
∞ (Doesn't have two interior edges)
∞ (Doesn't have three or more interior edges)
}

V(2,8) = min {
FH(2,8) = 4.8
FL(2,8,3,7) + V(3,7) = -2.0 + 4.4 = 2.4
∞ (Doesn't have three or more interior edges)
}

V(3,9) = min {
FH(3,9) = 4.8
FL(3,9,4,8) + V(4,8) = -1.5 + 4.4 = 2.9
∞ (Doesn't have three or more interior edges)
}

V(1,9) = min {
FH(1,9) = 5.2
min {
FL(1,9,2,8) + V(2,8) = -1.7 + 2.4 = 0.7
FL(1,9,3,7) + V(3,7) = 5.5 + 4.4 = 9.9
FL(1,9,3,6) + V(3,6) = 5.8 + 4.1 = 9.9
FL(1,9,3,5) + V(3,5) = 6.2 + 4.1 = 10.3
FL(1,9,6,8) + V(6,8) = 6.2 + 4.1 = 10.3
}
∞ (Doesn't have three or more interior edges)
}

Let's calculate the other score matrix, W.

According to the rules of base pairing,
$W(i, i) = \infty$
$W(i, i+1) = \infty$

$W(i,j) = \min \{$
$W(i + 1, j),$
$W(i, j - 1),$
$V(i, j),$
$\min[W(i,k) + W(k + 1, j)]$ for $i < k < j$
$\}$

$W(1,3) = \min \{$
$W(2,3) = \infty$
$W(1,2) = \infty$
$V(1,3) = \infty$
Case 4 not possible.
$\}$

$W(2,4) = \min \{$
$W(3,4) = \infty$
$W(2,3) = \infty$
$V(2,4) = \infty$
Case 4 not possible.
$\}$

$W(3,5) = \min \{$
$W(4,5) = \infty$
$W(3,4) = \infty$
$V(3,5) = 4.1$
Case 4 not possible.
$\}$

$W(4,6) = \min \{$
$W(5,6) = \infty$
$W(4,5) = \infty$
$V(4,6) = \infty$
Case 4 not possible.
$\}$

W(5,7) = min {
W(6,7) = ∞
W(5,6) = ∞
V(5,7) = ∞
Case 4 not possible.
}

W(6,8) = min {
W(7,8) = ∞
W(6,7) = ∞
V(6,8) = 4.1
Case 4 not possible.
}

W(7,9) = min {
W(8,9) = ∞
W(7,8) = ∞
V(7,9) = ∞
Case 4 not possible.
}

W(1,4) = min {
W(2,4) = ∞
W(1,3) = ∞
V(1,4) = 4.3
W(1,2) + W (3,4) = ∞
}
W(2,5) = min {
W(3,5) = 4.1
W(2,4) = ∞
V(2,5) = ∞
W(2,3) + W (4,5) = ∞
}

W(3,6) = min {
W(4,6) = ∞
W(3,5) = 4.1
V(3,6) = 4.1
W(3,4) + W(5,6) = ∞
}

W(4,7) = min {
W(5,7) = ∞
W(4,6) = 4.1
V(4,7) = ∞
W(4,5) + W(6,7) = ∞
}

W(5,8) = min {
W(6,8) = 4.1
W(5,7) = ∞
V(6,8) = 4.1
W(5,6) + W (6,7) = ∞
}

W(6,9) = min {
W(7,9) = ∞
W(6,8) = 4.1
V(6,9) = ∞
W(6,7) + W (8,9) = ∞
}

W(1,5) = min {
W(2,5) = 4.1
W(1,4) = 4.3
V(1,5) = 4.4
min {
W(1,2) + W(3,5) = ∞
W(1,3) + W(4,5) = ∞
}
}

W(2,6) = min {
W(3,6) = 4.1
W(2,5) = 4.1
V(2,6) = ∞
min {
W(2,3) + W(4,6) = ∞
W(2,4) + W(5,6) = ∞
}
}

W(3,7) = min {
W(4,7) = 4.1
W(3,6) = 4.1
V(3,7) = 4.4
min {
W(3,4) + W(5,7) = ∞
W(3,5) + W(6,7) = ∞
}
}

W(4,8) = min {
W(5,8) = 4.1
W(3,7) = 4.1
V(4,8) = 4.4
min {
W(4,5) + W(6,8) = ∞
W(4,6) + W(7,8) = ∞
}
}
W(5,9) = min {
W(6,9) = 4.1
W(5,8) = 4.1
V(5,9) = ∞
min {
W(5,6) + W(7,9) = ∞
W(5,7) + W(8,9) = ∞
}
}

W(1,6) = min {
W(2,6) = 4.1
W(1,5) = 4.1
V(1,6) = 4.6
min {
W(1,2) + W(3,6) = ∞
W(1,3) + W(4,6) = ∞
W(1,4) + W(5,6) = ∞
}
}

W(2,7) = min {
W(3,7) = 4.1
W(2,6) = 4.1
V(2,7) = ∞
min {
W(2,3) + W(4,7) = ∞
W(2,4) + W(5,7) = ∞
W(2,5) + W(6,7) = ∞
}
}

W(3,8) = min {
W(4,8) = 4.1
W(3,7) = 4.1
V(3,8) = ∞
min {
W(3,4) + W(5,8) = ∞
W(3,5) + W(6,8) = ∞
W(3,6) + W(7,8) = ∞
}
}

W(4,9) = min {
W(5,9) = 4.1
W(4,8) = 4.1
V(4,9) = ∞
min {
W(4,5) + W(6,9) = ∞
W(4,6) + W(7,9) = ∞
W(4,7) + W(8,9) = ∞
}
}

W(1,7) = min {
W(2,7) = 4.1
W(1,6) = 4.1
V(1,7) = ∞
min {
W(1,2) + W(3,7) = ∞
W(1,3) + W(4,7) = ∞
W(1,4) + W(5,7) = ∞
W(1,5) + W(6,7) = ∞
}

}

W(2,8) = min {
W(3,8) = 4.1
W(2,7) = 4.1
V(3,8) = ∞
min {
W(2,3) + W(4,8) = ∞
W(2,4) + W(5,8) = ∞
W(2,5) + W(6,8) = 4.1 + 4.1 = 8.2
W(2,6) + W(7,8) = ∞
}
}

W(3,9) = min {
W(4,9) = 4.1
W(3,8) = 4.1
V(3,9) = 2.9
min {
W(3,4) + W(5,9) = ∞
W(3,5) + W(6,9) = 4.1 + 4.1 = 8.2
W(3,6) + W(7,9) = ∞
W(3,7) + W(8,9) = ∞
}
}

W(1,8) = min {
W(2,8) = 4.1
W(1,7) = 4.1
V(1,8) = ∞
min {
W(1,2) + W(3,8) = ∞
W(1,3) + W(4,8) = ∞
W(1,4) + W(5,8) = 4.3 + 4.1 = 8.4
W(1,5) + W(6,8) = 4.1
W(1,6) + W(7,8) = ∞
}
}

W(2,9) = min {
W(3,9) = 2.9
W(2,8) = 4.1
V(2,9) = ∞

min {
W(2,3) + W(4,9) = ∞
W(2,4) + W(5,9) = ∞
W(2,5) + W(6,9) = 4.1
W(2,6) + W(7,9) = ∞
W(2,7) + W(8,9) = ∞
}
}

W(1,9) = min {
W(2,9) = 2.9
W(1,8) = 4.1
V(1,9) = 0.7
min {
W(1,2) + W(3,9) = ∞
W(1,3) + W(4,9) = ∞
W(1,4) + W(5,9) = 4.3
W(1,5) + W(6,9) = 4.1
W(1,6) + W(7,9) = ∞
W(1,7) + W(8,9) = ∞
}
}

Appendix B: Nussinov Algorithm Implementation

```
//
//  main.c
//  NussinovJacobson
//
//  Created by Hardik Shah on 11/20/12.
//  Copyright (c) 2012 San Jose State University. All rights reserved.
//

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

char input_sequence[8000], window_sequence[700];
int input_length;
int score[700][700];
int window_begin,window_end;

void read_fastafile(void);
void initialize_score_matrix(int);// Window size is the argument
int do_basepair(char, char);
int calculate_maxscore(int, int);
void print_score_matrix(int);


int main (int argc, const char * argv[])
{
   int window_size;//,window_begin,window_end;
   clock_t start_time;

   start_time = clock();

   read_fastafile();

   printf("\nEnter the Window Size:");
   scanf("%d", &window_size);

   initialize_score_matrix(window_size);

   for (window_begin = 0; window_begin < (input_length - window_size);
window_begin++) {
```

```
        initialize_score_matrix(window_size);

        strncpy(window_sequence, input_sequence+window_begin, window_size);

        printf("%d \t %d\n",window_begin, calculate_maxscore(0, window_size-1));
    }

    printf("Time elapsed: %f\n", ((double)clock() - start_time) / CLOCKS_PER_SEC);

    return 0;

}

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

void read_fastafile(void)
{
    FILE *fasta_fp;
    char current_fp;
    int index = 0;

    fasta_fp = fopen("/Users/hardikshah/Desktop/NC_0020583.txt", "r");

    if (fasta_fp == NULL) {
        printf("Error Opening File");
    }

    while ( (current_fp = fgetc(fasta_fp)) != EOF ) {
        // Not storing the Fasta comment that begins with '>' and ends with '\n'
        if (current_fp == '>') {
            do {
                current_fp = fgetc(fasta_fp);
            } while (current_fp != '\n');

        }
        else
        {
            if (current_fp == '\n') {
                continue;
            }
            else{
                current_fp = toupper(current_fp);
                if (current_fp == 'T') {
                    current_fp = 'U';
```

```
            }
            input_sequence[index++] = current_fp;
        }
    }
}

fclose(fasta_fp);
input_sequence[index] = '\0';
input_length = index;

//   printf("\nInput String is: %s",input_sequence);
//   printf("\nIndex: %d",index);
}

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


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


//*************************************************************************
void initialize_score_matrix(int window_size)
{
   int i, j;
   for (i = 0; i < window_size; i++)
      for (j = 0; j < window_size; j++){
         if (i == j) {
            score[i][j] = 0;           // score[i][i] = 0 by definition (Initializing the Diagonal to
0 as no nucleotide can basepair with itself)
         }
         else
            score[i][j] = -1;
      }
```

```
   // Intializing the next diagonal to 0 .. Base pairs of length 1..
   for (i = 1; i< window_size; i++) {
      score[i][i - 1] = 0;
   }
}
//**********************************************************************


//**********************************************************************
void print_score_matrix(int window_size)
{
   int r, c;
   for (r = 0; r < window_size; r++)
   {
      printf("\n");
      for (c = 0; c < window_size; c++)
      {
         printf("\t");
         printf("%d",score[r][c]);
      }
   }
   printf("\n");
}
//**********************************************************************


//**********************************************************************
int calculate_maxscore(int r, int c)
{
   int max_score, k;

   if (score[r][c] != -1) {
      return score[r][c];
   }

   max_score = calculate_maxscore(r+1,c);        // Case 'V'

   if (max_score < calculate_maxscore(r, c-1)) {
      max_score = score[r][c-1];              // Case 'H'
   }
                                    // Case 'D'
   else if ( max_score < (calculate_maxscore(r+1,c-1) +
do_basepair(window_sequence[r], window_sequence[c])) ){
```

```
        max_score = score[r+1][c-1] + do_basepair(window_sequence[r],
window_sequence[c]);
    }

    for (k = r+1; k < c; k++) {
        if ((calculate_maxscore(r,k) + calculate_maxscore(k+1,c) ) > max_score) {
            max_score = score[r][k] + score[k+1][c];
        }
    }
    score[r][c] = max_score;
    return max_score;
}
//*************************************************************************
```

Appendix C: Zuker Algorithm Implementation

```
//
//  main.c
//  zuker
//
//  Created by Hardik Shah on 11/11/12.
//  Copyright (c) 2012 San Jose State University. All rights reserved.
//
//*********************************************************************

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

# define MAXLEN 40
# define infinity 999
# define not_calculated 4567

float calculate_V(int,int);
float calculate_W(int,int);
void initialize_everything(void);
int do_basepair(char,char);
float hairpin_loop(int,int);
float stacking_loop(int,int);
int getindex_stacking(int,int);
float bulge_loop(int,int,int,int);
float interior_loop(int,int,int,int);
void read_fastafile(void);

char input_sequence[8000],window_sequence[MAXLEN];
int input_length,window_begin;

float V[MAXLEN][MAXLEN],W[MAXLEN][MAXLEN];
//int loop_type[MAXLEN][MAXLEN], pathmatrix[MAXLEN][MAXLEN][2];

//int traceback_stack[MAXLEN][2];
//int stack_pointer = -1;

float stacking_energy[6][6]={
    {-0.9,-1.8,-2.3,-1.1,-1.1,-0.8},
    {-1.7,-2.9,-3.4,-2.3,-2.1,-1.4},
```

```
    {-2.1,-2.0,-2.9,-1.8,-1.9,-1.2},
    {-0.9,-1.7,-2.1,-0.9,-1.0,-0.5},
    {-0.5,-1.2,-1.4,-0.8,-0.4,-0.2},
    {-1.0,-1.9,-2.1,-1.1,-1.5,-0.4}
};
/* A/U = 0, C/G = 1, G/C = 2, U/A = 3, G/U = 4, U/G = 5.
 Sequence of the rows and columns in stacking_energy
 Stacking energy in double-stranded region when the base
 pair listed in left column is followed by the base pair listed in top row.C/G followed by
 U/A is therefore the dinucleotide 5'CU 3' paired to 5' AG 3' with stacking energy as
 stacking_energy[2][4] = 2.3

 */

float destabilizing_energy[3][5] = {
    {infinity, 5.3, 6.6, 7.0, 7.4},
    {3.9, 4.8, 5.5, 6.3, 6.7},
    {infinity, 4.4,5.3,6.1,6.5}
};
/* Rows*Columns  1  5   10  20  30
 interior,
 Bulge,
 Hairpin
 */
//*************************************************************************


//*************************************************************************
/
int main (int argc, const char * argv[])
{
    int window_size; //,row,column;
    clock_t start_time;

    start_time = clock();

    read_fastafile();


    printf("\nEnter the Window Size:");
    scanf("%d", &window_size);

    initialize_everything();
```

```
   for (window_begin = 0; window_begin < (input_length - window_size);
window_begin++) {

      initialize_everything();

      strncpy(window_sequence, input_sequence+window_begin, window_size);

      printf("%d\t%f\n",window_begin, calculate_W(0, window_size-1));
   }
   printf("Time elapsed: %f\n", ((double)clock() - start_time) / CLOCKS_PER_SEC);
   return 0;
}
//**********************************************************************



//**********************************************************************
float calculate_V(int i, int j)
{
   float minimum_energy,temp_energy;
   int h,k;

   // When they don't base pair
   if (!do_basepair(window_sequence[i],window_sequence[j])) {
      V[i][j] = infinity;
      return infinity;
   }

   if (V[i][j] != not_calculated) {   //crazy way of saying if V is calculated
      return V[i][j];
   }

   // case 1: FH(i,j)

   minimum_energy = hairpin_loop(i, j);
   //pathmatrix[i][j][0] = i+1;
   //pathmatrix[i][j][1] = j-1;
   //loop_type[i][j] = -4; //-4 for hairpin

   /*
    case 2: min[FL(i,j,h,k) + V(h,k)]
    a-> If there are not two interior edges then energy is infinity
    Also, if there are only 4 nucleotides in the subsequence then energy is infinity as it
can't have two interior edges.
```

```
    b-> If there are two interior edges then it can be either stacking region or bulge loop or
interior loop
    */

   for (h = i+1; h < j; h++) {
      for (k = j-1; k > h; k--) {
         if (do_basepair(window_sequence[h],window_sequence[k])) {
            if (h == i+1 && k == j-1) {
               //Check if i+1 and j-1 base pair, else return infinity. Stacking loop will have
two basepairs
               temp_energy = stacking_loop(i, j) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;
//                  pathmatrix[i][j][0] = h;
//                  pathmatrix[i][j][1] = k;
//                  loop_type[i][j] = -1; // -1 for stacking
               }
            }
            else if (h == i+1 || k == j-1){
               temp_energy = bulge_loop(i, j, h, k) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;
//                  pathmatrix[i][j][0] = h;
//                  pathmatrix[i][j][1] = k;
//                  loop_type[i][j] = -2; // -2 for bulge
               }
            }
            else{
               temp_energy = interior_loop(i, j, h, k) + calculate_V(h, k);
               if (temp_energy < minimum_energy) {
                  minimum_energy = temp_energy;
//                  pathmatrix[i][j][0] = h;
//                  pathmatrix[i][j][1] = k;
//                  loop_type[i][j] = -3; //-3 for interior
               }
            }
         }
      }
   }

   /*
   case 3: min[W(i+1,k) + W(k+1,j-1)] condition is i+1 < k < j-1
    This has more than two interior edges.*/
```

```
   for (k = i+2; k < j-1; k++) {
      temp_energy = W[i+1][k] + W[k+1][j-1];
      if (temp_energy < minimum_energy) {
         minimum_energy = temp_energy;
//          pathmatrix[i][j][0] = k;
//          pathmatrix[i][j][1] = k;
//          loop_type[i][j] = -4; // value of the bifurcation position
      }
   }

   V[i][j] = minimum_energy;

   return minimum_energy;
}
//*************************************************************************


//*************************************************************************
float calculate_W(int i, int j)
{
   int k;
   float minimum_energy;

   if (W[i][j] != not_calculated) {   // if W is calculated then return its value.
      return W[i][j];
   }

   minimum_energy = calculate_V(i, j);

   /*
      (i  , j-1)     (i   ,   j)
      (i+1, j-1)     (i+1 ,   j)
    */

   // All the following cases require the earlier values to be calculated. So, I should call
calculate_W function in increasing diagonal order.

   if (    minimum_energy  >   calculate_W(i+1,j) ) {
      minimum_energy = W[i+1][j];
//       loop_type[i][j] = -5; // don't basepair
   }

   else if (   minimum_energy  >   calculate_W(i,j-1) ) {
      minimum_energy = W[i][j-1];
```

```
//       loop_type[i][j] = -5; // don't basepair
   }

   else{
      for (k = i+1; k < j; k++) {
         if (    minimum_energy > calculate_W(i,k) + calculate_W(k+1,j)  ) {
            minimum_energy = W[i][k] + W[k+1][j];
//             loop_type[i][j] = k;// a little dubious
         }
      }
   }
   W[i][j] = minimum_energy;
   return minimum_energy;
}
//*************************************************************************


//*************************************************************************
// This function will be used after secondary structure of each subsequence is calculated.
void initialize_everything()
{
   int i,j;

   for (i=0; i<MAXLEN; i++) {
      for (j=0; j<MAXLEN; j++) {
         V[i][j] = not_calculated;
         W[i][j] = not_calculated;
//          loop_type[i][j] = -5;//-5 for don't base pair
//          pathmatrix[i][j][0] = -1;
//          pathmatrix[i][j][1] = -1;
      }
   }
   // Sequences of length 0 and 1 can't base pair.
   //V
   for (i=0; i<MAXLEN; i++) {
      V[i][i] = infinity;
      if (i < MAXLEN-1) V[i][i+1] = infinity;
   }
   //W
   for (i=0; i<MAXLEN; i++) {
      W[i][i] = infinity;
      if (i < MAXLEN-1) W[i][i+1] = infinity;
   }
```

```
}
//*************************************************************************


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


//*************************************************************************
float hairpin_loop(int row, int column)
{
   float hairpin_energy,interpolation;
   int hairpin_nucleotides;

   hairpin_nucleotides = abs(column-row) + 1;

   //Pentanucleotide hairpins are stable. Anything less than that is not permitted.
   if (hairpin_nucleotides <= 4)
      return infinity;

   if (hairpin_nucleotides == 5) {
      hairpin_energy = destabilizing_energy[2][1];// Value is 4.4. Refer the hardcoded
values.
      //return hairpin_energy;
   }
   else if (hairpin_nucleotides <= 10) {
      interpolation = (   (10-hairpin_nucleotides)   *   (   (destabilizing_energy[2][2] -
destabilizing_energy[2][1])   /   5)  );
      hairpin_energy = destabilizing_energy[2][2] - interpolation;

      /*hairpin_energy = 5.3 - ((10-temp) * 0.18);// Good trick!
      return hairpin_energy;*/
   }
   else if (hairpin_nucleotides <= 20) {
```

```
      interpolation = (   (20-hairpin_nucleotides)   *   (   (destabilizing_energy[2][3] -
destabilizing_energy[2][2])   /   10) );
      hairpin_energy = destabilizing_energy[2][3] - interpolation;

      /*hairpin_energy = 6.1 - ((20-temp) * 0.08);
      return hairpin_energy;*/
   }
   else if (hairpin_nucleotides <= 30){
      interpolation = (   (30-hairpin_nucleotides)   *   (   (destabilizing_energy[2][4] -
destabilizing_energy[2][3])   /   10) );
      hairpin_energy = destabilizing_energy[2][4] - interpolation;

      //hairpin_energy = 6.5 - ((30-temp) * 0.04);
   }

   return hairpin_energy;
}
//*********************************************************************


//*********************************************************************
int getindex_stacking(int row, int column)
{

   if (window_sequence[row] == 'A' && window_sequence[column] == 'U') {
      return 0;
   }

   if (window_sequence[row] == 'C' && window_sequence[column] == 'G') {
      return 1;
   }

   if (window_sequence[row] == 'G' && window_sequence[column] == 'C') {
      return 2;
   }

   if (window_sequence[row] == 'U' && window_sequence[column] == 'A') {
      return 3;
   }

   if (window_sequence[row] == 'G' && window_sequence[column] == 'U') {
      return 4;
   }
```

```
   if (window_sequence[row] == 'U' && window_sequence[column] == 'G') {
      return 5;
   }
   return -1;
}

float stacking_loop(int row,int column)
{
   float stacking_energy_value;
   int row_pointer,column_pointer;

   row_pointer = getindex_stacking(row, column);
   //What if i+1 and j-1 don't base pair? Column_pointer will go
   column_pointer = getindex_stacking(row+1, column-1);

   stacking_energy_value = stacking_energy[row_pointer][column_pointer];

   return stacking_energy_value;
}
//*********************************************************************


//*********************************************************************
float bulge_loop(int row1, int column1, int row2, int column2)
{

   float bulge_energy,interpolation;
   int bulge_nucleotides;

   bulge_nucleotides = ( (row2 - row1)  >  (column1 - column2)?   (row2 - row1)  :
(column1 - column2) ) + 2;


   if (bulge_nucleotides <= 5) {
      interpolation = (   (5 - bulge_nucleotides)   *   (   (destabilizing_energy[1][1] -
destabilizing_energy[1][0])   /   5)  );
      bulge_energy = destabilizing_energy[1][1] - interpolation;
   }

   else if (bulge_nucleotides <= 10) {
      interpolation = (   (10 - bulge_nucleotides)   *   (   (destabilizing_energy[1][2] -
destabilizing_energy[1][1])   /   5)  );
      bulge_energy = destabilizing_energy[1][2] - interpolation;
   }
```

```
   else if (bulge_nucleotides <= 20) {
      interpolation = (   (20 - bulge_nucleotides)   *   (   (destabilizing_energy[1][3] -
destabilizing_energy[1][2])   /   10) );
      bulge_energy = destabilizing_energy[1][3] - interpolation;
   }

   else if (bulge_nucleotides <= 30) {
      interpolation = (   (30 - bulge_nucleotides)   *   (   (destabilizing_energy[1][4] -
destabilizing_energy[1][3])   /   10) );
      bulge_energy = destabilizing_energy[1][4] - interpolation;
   }

   return bulge_energy;
}
//**********************************************************************


//**********************************************************************
float interior_loop(int row1, int column1, int row2, int column2)
{
   float interior_energy, interpolation;
   int interior_nucleotides;

   interior_nucleotides = (row2-row1) + (column1 - column2) + 2;

   if (interior_nucleotides == 5) {
      interior_energy = destabilizing_energy[0][1];
   }

   else if (interior_nucleotides <= 10) {
      interpolation = (   (10-interior_nucleotides)  *   (   (destabilizing_energy[0][2] -
destabilizing_energy[0][1])   /   5)  );
      interior_energy = destabilizing_energy[0][2] -  interpolation;
   }

   else if (interior_nucleotides <= 20) {
      interpolation = (   (20-interior_nucleotides)  *   (   (destabilizing_energy[0][3] -
destabilizing_energy[0][2])   /   10)  );
      interior_energy = destabilizing_energy[0][3] -  interpolation;
   }

   else if (interior_nucleotides <= 30) {
      interpolation = (   (30-interior_nucleotides)  *   (   (destabilizing_energy[0][4] -
```

```
destabilizing_energy[0][3])   /   10)  );
        interior_energy = destabilizing_energy[0][4] -  interpolation;
    }
    // what about the case with more than 30 edges. How do we handle that?

    return interior_energy;
}
//*********************************************************************


//*********************************************************************
void read_fastafile(void)
{
    FILE *fasta_fp;
    char current_fp;
    int index = 0;

    fasta_fp = fopen("/Users/hardikshah/Desktop/NC_0016121.txt", "r");

    if (fasta_fp == NULL) {
        printf("Error Opening File");
    }

    while ( (current_fp = fgetc(fasta_fp)) != EOF ) {
        // Not storing the Fasta comment that begins with '>' and ends with '\n'
        if (current_fp == '>') {
            do {
                current_fp = fgetc(fasta_fp);
            } while (current_fp != '\n');

        }
        else
        {
            if (current_fp == '\n') {
                continue;
            }
            else{
                current_fp = toupper(current_fp);
                if (current_fp == 'T') {
                    current_fp = 'U';
                }
                input_sequence[index++] = current_fp;
            }
        }
```

```
    }

    fclose(fasta_fp);
    input_sequence[index] = '\0';
    input_length = index;

}
//*********************************************************************
```