IMPROVING THE PORTABILITY AND PERFORMANCE OF JVIZ.RNA, A DYNAMIC RNA VISUALIZATION SOFTWARE

by

Boris Shabash

BHSc, University of Calgary, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

APPROVAL

Name:	Boris Shabash
Degree:	Master of Science
Title of Thesis:	Improving the Portability and Performance of jViz.RNA, a Dynamic RNA Visualization Software
Examining Committee:	Dr. Tamara Smyth Chair
	Dr. Kay C. Wiese Associate Professor, Computing Science Simon Fraser University Senior Supervisor
	Dr. S. Cenk Sahinalp Professor, Computing Science Simon Fraser University Supervisor
	Dr. Richard Zhang, Associate Professor, Computing Science Simon Fraser University Examiner
Date Approved:	28 July 2011

Abstract

jViz.RNA is a Java based software that focuses on the visualization of RNA and its structural elements. It has been employed by many research groups around the world and has prompted excellent and constructive feedback from those groups, along with several suggestions for improvements.

In this thesis, two major areas of jViz.RNA have been explored for the purpose of improvement;

First, RNAML and FASTA file format support was added to jViz.RNA's repertoire. This allows jViz.RNA users to utilize file formats used by other software, expanding jViz.RNA's capabilities in working in pipeline systems, and also contributes to the standardization of RNA data exchange by supporting the use of RNAML.

Second, five methods were explored in the context of improving the run time of jViz.RNA's structure drawing algorithm. First, simple parameter optimization of the existing algorithm was attempted, then the use of the Barnes-Hut algorithm was explored, Thirdly, the effects of using multiple threads to handle the calculations were measured, additionally, the use of dynamic C libraries to integrate C code into the Java environment was investigated, and finally, a technique termed 'structure recall', whereby the program uses files to register the layout of structures so they can be loaded for future runs, was examined.

The results demonstrated that the use of approximation based techniques such as parameter optimization and the Barnes-Hut algorithm produces the most drastic improvements in run time, but does so at the cost of aesthetics, which may be unacceptable for visualization based software such as jViz.RNA. Multithreading and integration of C code, however, proved to be beneficial techniques since these improved the speed at which calculations are done, without distorting the structures in ways that obscure important information.

To everyone that believes in the RNA world theory!

“Computer Science is a science of abstraction -creating the right model for a problem and devising the appropriate mechanizable techniques to solve it”

— A. AHO AND J. ULLMAN

Acknowledgments

I'd like to thank, first and foremost, my senior supervisor Dr. Kay Wiese. Since our meeting in 2009 Kay has made sure to always support me through my Masters career be it through sharing his experience and ideas, helping me to get funding, and sometimes, if necessary, dishing a degree of direct and well intended constructive criticism. A smart person is not hard to find in a university, but finding someone worth working for, therein lies the challenge. On that note, I would also like to thank my previous supervisor, Dr. Christian Jacob from the University of Calgary; both for recommending me to go to SFU, and for inspiring me to work in the field of bio-informatics. Without him, I would probably think that bio-informatics starts and ends at sequence analysis.

I would also like to extend a very big ”Thank you” to everyone in the administrative staff in the Computing Sciences department; Gerdi Snyder, Val Galat, Jeanie Hillis and everyone else who have had part in my degree completion. It's amazing how much there is to making one Masters student pass through aside from academics, and the administrative staff is the one responsible for this miracle.

I would also like to acknowledge the support my family gave me, particularly my parents; Evgeny and Rita Shabash. Although they did not have any background in computing sciences or bioinformatics, it did not stop them from supporting my work, offering help with whatever they could, encourage me to brainstorm, and support my endeavor in moving to a different province to pursue my graduate academic career. Thank you very much everyone.

Finally, a great big ”Thank you” goes to my girlfriend, Kristen, who was always there for me to share the woes of getting through grad school and would always put a smile on my face, and the wind back in my sails.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Ribonucleic Acid (RNA) is a pivotal molecule in the creation of life, and in their continuity today. As such, it is critical that we have tools that offer insight into RNA research and the relationship that exists between its structure and the many functions it can assume inside cells. jViz.RNA is a Java based software developed in SFU for the purpose of inspecting RNA secondary structures and analyzing their effect on the RNA's function.

In this thesis, jViz.RNA was improved in several categories that were requested by users from different research groups: First, some groups requested jViz.RNA portability's extended by including support for both RNA Markup Language (RNAML) and FASTA files. Second, many groups requested improvements in the speed at which the dynamic RNA model constructed by jViz.RNA stabilizes.

With regards to extending file formats, this thesis outlines the general structure of the files used, as well as the Java code required to support them. The code outlines for the file support are shown in the main thesis body, while full classes are present in the appendix. Speed optimization occupies the major part of this thesis body and addresses four focal points: First, the Barnes-Hut algorithm, an approximation algorithm for the *n*-body problem, is introduced and the thesis outlines the algorithm and code required to implement it. Second, multithreading is inspected in the context of Java and is then applied to the construction of the dynamic RNA model in jViz.RNA. Additionally, native C code is introduced as a potential optimization method by linking the Java interface with back-end C code to perform fast calculations. Finally, the notion of structure recall is presented, whereby a sequence that was viewed before is recalled as it was viewed last by jViz.RNA, eliminating the need for redundant calculations.

Sixteen sequences are the main focus in the experiments shown in this thesis. The short

Bacillus stearothermophilus 5S ribosomal RNA (AJ251080)(composed of 117nt[1]), *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579) (composed of 118nt), *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627) (composed of 120nt), *Arthrobacter globiformis* 5S ribosomal RNA (M16173) (composed of 122nt) and *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515) (composed of 124nt) sequences, the medium length *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) (composed of 436nt), *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) (composed of 517nt), *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) (composed of 940nt), *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) (composed of 945nt), *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) (composed of 954nt) and *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) (composed of 964nt) sequences, and the long *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) (composed of 2080nt), *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) (composed of 2236nt), *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) (composed of 2283nt), *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) (composed of 2404nt) and *Chlorella saccharophila* 18S ribosomal RNA (AB058310) (composed of 2510nt) sequences. With these 16 sequences, the optimization methods are compared to see how well they improve on running times for different sized sequences. The results show that the Barnes-Hut algorithm and the parameter optimization methods give drastic improvements, sometimes reducing the run time by 90%. However, they also cause aesthetic distortions which obscure some vital structure information. On the other hand, multithreading cause a slowdown in the construction of the shorter sequence, but indeed improve the construction time for the long one, and does so without the distortion of vital structural elements. At the same time, Integration of native C code and structure recall show favorable results for both long and short sequences.

In addition to improving jViz.RNA, there are also two major research contributions that can be derived from this thesis. The behaviour of different length sequences was inspected under different algorithms and compared to one another. Chapter 4 shows that some methods offer faster stabilization time for long sequences but extend the stabilization time of short sequences; while other methods provide benefits to both long and short sequences. Additionally, the affect different methods have on the layout of sequences was inspected in Chapter 4 as well. Approximation algorithms such as the Barnes-Hut algorithm tended to introduce distortions into the sequences' layouts while multithreading and use of C code introduced none.

The concluding portion of this thesis simply summarizes what was seen in the main

[1]nt means 'nucleotides' and is the measurement unit for single stranded sequences

thesis body and postulates future directions for the research and improvement of jViz.RNA.

Chapter 2

Background

2.1 RNA: Structures, Functions and the Significance of RNA Research

RNA is a molecule that, much like Deoxyribonucleic acid (DNA), is found in all life forms discovered up to date. Like DNA, RNA is a string of nitrogenous bases that are connected together by phosphates connected to a sugar portion. However, RNA is quite different from DNA both structurally and functionally, and it is those differences and their resulting functions in the cell that call RNA to our attention in the study of biological life. In DNA, one would normally find four nucleotides along the sequence: Adenine (denoted A), Guanine (denoted G), Cytosine (denoted C) and Thymine (denoted T). These four nucleotides compose the DNA sequence and create a double helix where A pairs with T (via a double hydrogen bond) and G pairs with C (via a triple hydrogen bond) In RNA, there are four nucleotides as well: A, G, C and Uracil (denoted U) instead of thymine to bind to adenine.

However, RNA is produced as a single strand in the cell, and not a double helix. As a result, nucleotides in RNA often pair with other nucleotides from the same strand, causing RNA to take on a secondary structure. This secondary structure might then fold around itself to form a three dimensional tertiary structure.

From this short description it is evident that RNA carries properties that are reminiscent of both DNA and proteins. Indeed, RNA has the ability to carry and transmit information coded by DNA, as well as act as functional molecule in the cell carrying its own functions. For example, messenger RNA (mRNA) is the mediator between the DNA, which holds the cell's genetic ”code”, and the ribosomes, a collection of proteins that translate genetic information into protein agents. mRNA carries a copy of the genetic ”code” in DNA by the virtue of paring G with C, and A with U (Figure 2.1 and Figure 2.2).

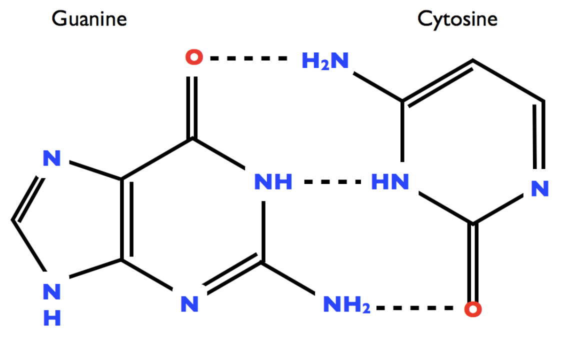


Figure 2.1: Guanine and Cytosine bonded via a triple hydrogen bond

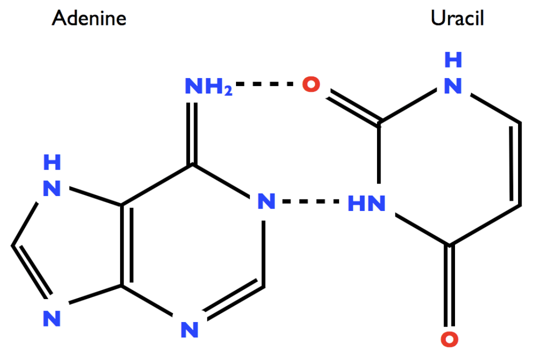
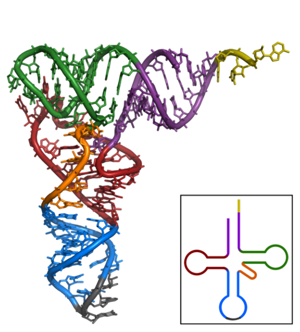
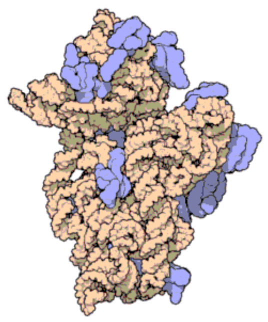


Figure 2.2: Adenine and Uracil bonded via a double hydrogen bond

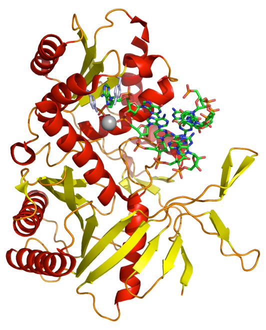
This copy is then used by the ribosomes to construct proteins. In the protein construction process, another special form of RNA called transfer RNA (tRNA, shown in Figure 2.3a) binds an amino acid on one end, and displays a three nucleotide sequence called an anti-codon on the opposite end. The anti-codon pairs with a codon (a three nucleotide sequence on the mRNA) when the mRNA is docked in the ribosome. When multiple tRNA molecules dock into the ribosome and string their amino acids together, a protein is formed. With the previous example it is evident RNA fulfills two crucial tasks in the protein synthesis process both as a data carrier and a functional agent. Furthermore, RNA is also part of the ribosome structural unit itself. Special RNA called ribosomal RNA (rRNA, shown in Figure 2.3b) make up the structure that allows the ribosome to dock around the mRNA and bind the tRNA. It is in fact the rRNA present in the ribosome that allows it to interact with mRNA and tRNA instances. Although the ribosome is not purely RNA, but contains protein components in it as well, it also serves to demonstrate the many functions RNA can assume depending on its sequence or structure. RNA-protein complexes also serve to protect the cell from foreign genetic material in both eukaryotes and prokaryotes (Figure



(a) The crystal structure of yeast phenylalanine tRNA. PDB ID:1EHZ (taken from http://en.wikipedia.org/wiki/File:TRNA-Phe_yeast_1ehz.pdf)



(b) The small subunit of the Thermus thermophilis ribosome. The ribosomal RNA is marked in light orange while the protein component is marked in blue (taken from http://en.wikipedia.org/wiki/File:10_small_subunit.gif)



(c) A protein-RNA complex involved in cleaving foreign RNA that enters cells. PDB ID:1YTU (taken from http://en.wikipedia.org/wiki/File:Argonaute_1u04_1ytu_composite.png)

Figure 2.3: The different functions RNA can assume inside cells

2.3c). Eukaryotes contain a mechanism that allows them to fight off foreign RNA material that could be viral. This mechanism is known as the RNA interference (RNAi) and is propagated by another form of RNA: small interfering RNA (siRNA) [13]. siRNA are short, double stranded, sequences of RNA that are derived from RNA that is foreign to the cell. An enzyme known as Dicer cuts double stranded RNA that enters cells into siRNA. Each of these short RNA sequences binds to a protein complex that is known as RISC (RNAi Induced Silencing Complex) and unwinds into a single stranded short RNA strand that remains attached to RISC. RISC than uses the RNA strand bound to it (derived from the original foreign RNA) to bind to other instances of the foreign RNA and direct degradation of those instances, thus preventing infection of the cell.

RNA also plays a role in the regulation of protein expression. Short RNA sequences called small nucleolar RNA (snoRNA) are responsible for modification of other RNA molecules [2]. The modifications usually result in either methylation or pseudouridylation, and RNAs that have been modified experience changes in their stability, function and/or structure. Similarly, another group of small RNA sequences known as microRNAs (miRNAs) binds to particular sites in the mRNA of plants and animals to induce post transcriptional repression (meaning an mRNA is produced, but no protein product can be formed from it) [4].

It is evident that by studying the RNA molecule and comprehending the relationship between an RNA's sequence and structure, one can obtain comprehension of many events in living cells and how they would unfold. A detailed enough understanding of RNA could allow researchers to design RNAs with a variety of functions, or that code for a variety of proteins. In essence, understanding the synthesis and role of RNA, would be a major stepping stone in understanding living cells and even designing them via synthetic biology. In order to help understand the relationship that exists between RNA sequences and their resulting structures, it is necessary to make software that displays that information in a clear, concise and intuitive way that lends itself to easy interpretation by researchers of the natural sciences. The focus of this thesis is the visualization of RNA structure information, focusing on jViz.RNA, a software designed to help unlock the mysteries of RNA structure and functions.

2.2 Visualization in Bioinformatics and jViz.RNA

2.2.1 A General Overview of Visualization

Visualization is critical in data representation. A good visualization displays the data of a particular problem such as to show the information relevant to a problem while hiding the irrelevant. Bio-informatics problem often include finding patterns in data or noting small details hiding in a larger pool of information, and visualization becomes critical at those times.

Bio-informatics problems can be divided into two rough categories: pattern visualization and structure visualization. Pattern visualization requires the user to find patterns in larger data pools. Tools such as GeneShelf [20] and ClustalW [11] let the user do so. GeneShelf aims to show gene expression patterns in microarrays. While ClustalW aids the user in comparing DNA sequences and find differences between closely related sequences that could account in differences in function or shape between the sequences' protein products. The main focus of this thesis is problems of the second type, where structure visualization is required. Not just in RNA research, but in many other biological fields, shape and structure are key features of biological agents. The structure an agent assumes dictates how it interacts with other agents in biological systems, and the right representation of this structure can be invaluable in understanding how RNA, proteins, viruses, cells, hormones, receptors and other agents interact with each other. The class of visualization tools available for this set of problems is very varied with each tool offering strengths in a different area. Some tools such as PyMol [1, 26] and Chimera [24] offer great insight into protein structure inspection, and are event extensible in case a particular group wishes to add a function unique to their research. Other software such Jmol [17] and BALL [6] are made for viewing smaller agents such as short polymers in greater detail. These display the information relevant to problems involving small molecules very well, but may present a clutter of data when used for large proteins.

So far, it is visible that the area of bioinformatics visualization has many faces and many directions that lend themselves for exploration. Most software that render general molecules are also able to render RNA molecules. However, this approach ignores the unique properties an RNA structure may have that would effect its function, and also RNA's ability to transfer data. There are software products out there that focus specifically on RNA visualization, but they tend to focus on one or few display methods which convey a limited amount of information. Being such a unique and rare molecule, RNA requires specialized visualization tools that can address the different facets of RNA molecules and present them in an appropriate manner.

2.2.2 RNA Visualization Methods

There are numerous ways to present the structure an RNA sequence may assume. The most intuitive of those is called ”classical structure”, and displays the RNA strand as a two dimensional structure where different structure elements are clearly visible (Figure 2.4).

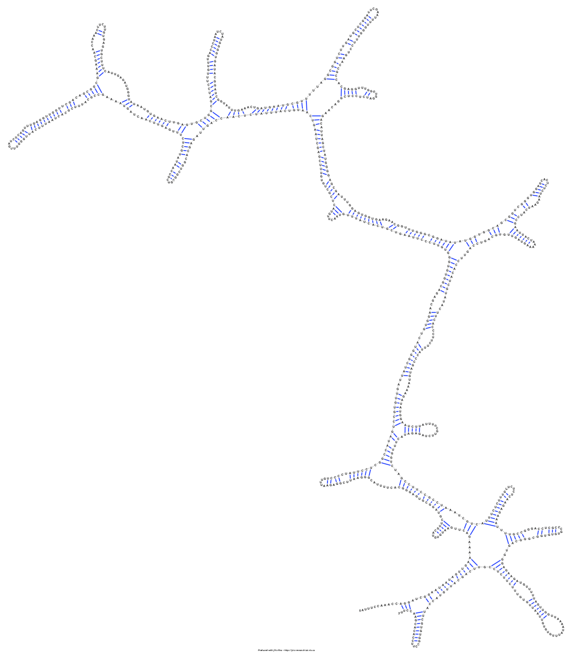


Figure 2.4: The classical structure representation of *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605). Image produced using jViz.RNA

This method of display is very useful for users and researchers. It allows people to very easily spot structural elements they may be interested in. However, large sequences may become convoluted and this form of display may prove to be overwhelming for users if they are looking for particular structural elements. Other display methods that exist try to abstract some of the information that may not be relevant to users.

One display method that abstracts some of the information is the use of linear or circular Feynman diagrams. Linear Feynman diagrams draw the sequence as a horizontal line with bonds between base pairs drawn as arcs from the location of one base to the other (Figure 2.5). This display method allows for easy identification of certain structural elements.

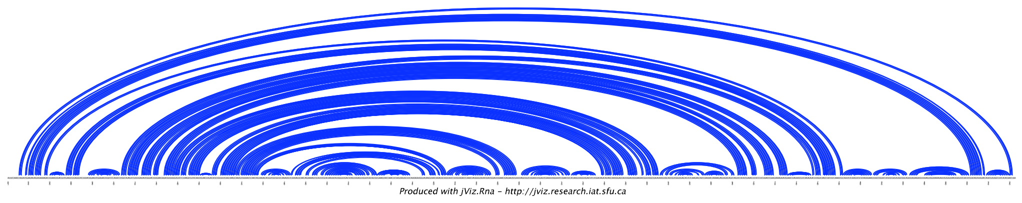


Figure 2.5: The linear Feynman representation of *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605). Image produced using jViz.RNA

However, large structures can require a large horizontal space and obscure certain details. For large structures, circular Feynman diagrams can be much more appropriate. In circular Feynman diagrams the sequence is laid out in a circle, rather than a horizontal line, and the bonded base pairs are then connected by arcs (Figure 2.6). This combines the abstraction of linear Feynman diagrams with a more efficient use of space.

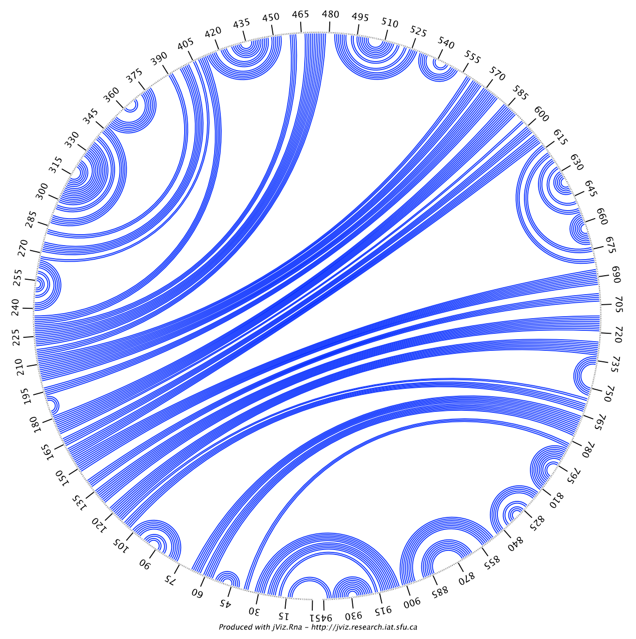


Figure 2.6: The circular Feynman representation of *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605). Image produced using jViz.RNA

Sometimes one might want to inspect the overall topology of a structure rather than its individual components. For that purpose jViz.RNA offers the structure to be viewed as a dual graph (Figure 2.7). In a dual graph, every structural element in the sequence is represented by a vertex and every stem is represented by an edge [12, 14]. The resulting structure displays a 'high level' overview of the structure topology without cluttering the user with the finer information.

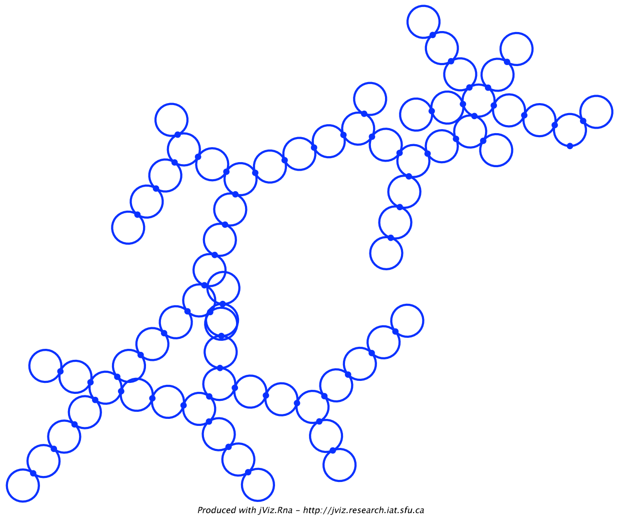


Figure 2.7: The dual graph representation of *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605). Image produced using jViz.RNA

In light of the variety of display methods available, it is important to inspect the existing body of software and what it offers users in regards to RNA structure display.

2.2.3 A Comparison of jViz.RNA to other RNA Viewing Tools

When speaking of RNA structure visualization, there are several tools available [9, 18, 19, 14, 10] and a comprehensive overview of them can be found in [14]. Instead, this section would focus on comparing jViz.RNA [14, 31, 30] with two of the available softwares: RNAViz [10] and VARNA [9].

RNAViz is written in C and Tk [23] where the user interface utilizes Tk and C is used for back-end functionality (mostly due to its fast computation). The software presents what is referred to as a ”native interface”, meaning it resembles many other interfaces, such as MS Word or Paint. RNA files that are loaded are placed as best as the software can place them, and then the user is offered with the possibility of re-arranging the structure if they so choose. RNAViz also offers the creation and use of ”skeleton files” which contain data on how a specific structure should be laid out. In essence, once a user has laid out a structure in a satisfactory method, he or she is able to save a skeleton file describing the layout such that the next time he or she loads the file, he or she does not need to readjust the structure layout again. RNAViz is described as a tool whose goal is to be ”a user-friendly, portable, windows-type program for producing publication-quality secondary structure drawings of RNA”(cited from [10]). As such, it offers not only the drawing and placement of RNA structures, but also the labeling of RNA strands and their components with costume labels. The labels are linked to the object they label such that if a nucleotide is moved, the label corresponding to it also moves. An example of RNAViz's interface can be seen in Figure 2.8

Indeed, RNAViz offers the user many features to inspect RNA strands, but it is also has two major disadvantages.
First, by offering the freedom it does with regards to RNA structure layout, it also places a great deal of responsibility in the user's lap. For small structure, the prospect of arranging them into a coherent and clear layout may be a simple task, but when the matter comes to RNA structures of over 1,000 bases, the task may become tedious, challenging and time consuming. Second, RNAViz only offers the display of RNA in the classical structure mode. This may be a very intuitive way to think of RNA representation, but this method lacks abstraction and display more data than is necessary sometimes. Other representations such as Feynman diagrams might offer a much easier way of comparing structures, and spotting interesting elements (such as pseudo knots) in larger structures.

VARNA addresses the drawbacks of RNAViz. First, it employes two ways of drawing the classical structure; The structure can be either drawn by an algorithm similar to the

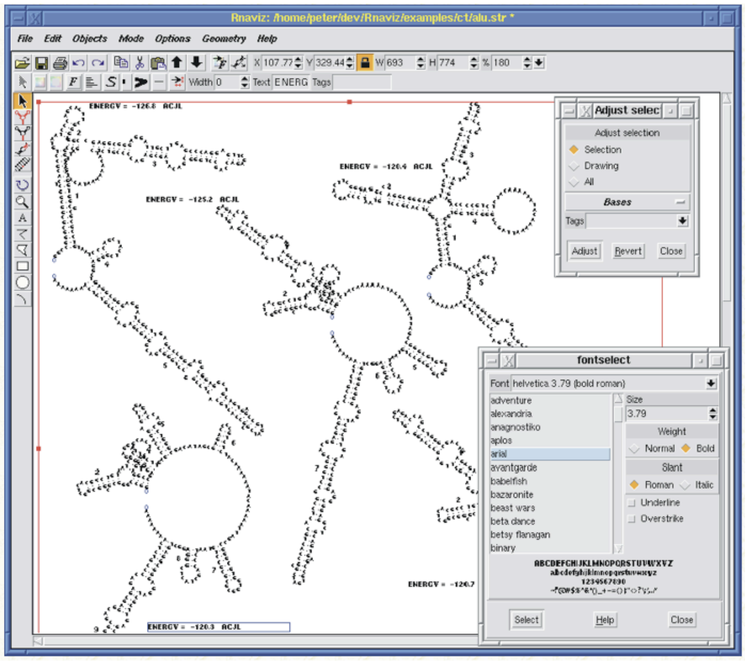


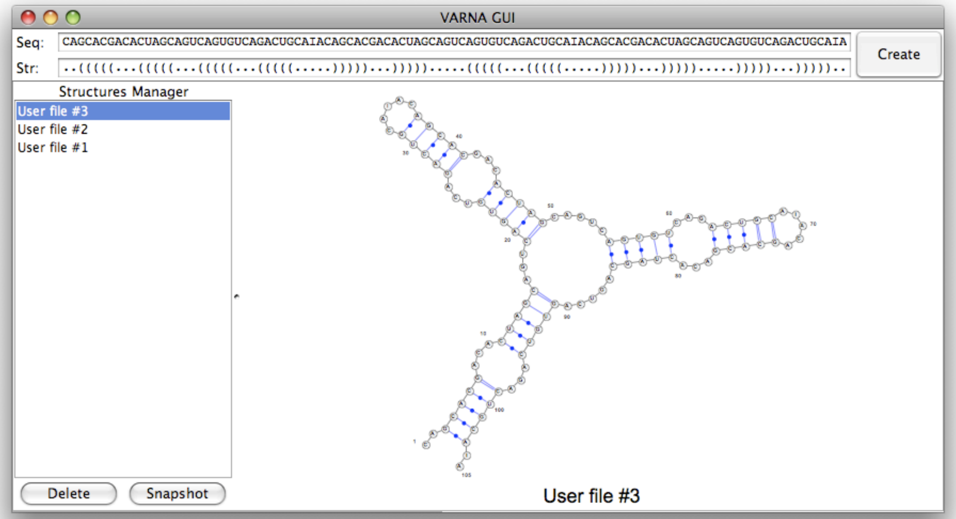
Figure 2.8: An interface snapshot of RNAViz, taken from http://rnaviz.sourceforge.net/tour/implementation.html

one used in RNAViz, constructing a structure that may be self intersecting, but at the same time allows for user interaction to remove the intersections. Alternatively, the structure can be drawn using a heuristic based method [7]. the resulting structure is much more neatly placed, but interactions are limited to moving single bases. In addition, VARNA allows users to view the structures at a variety of display modes. The first two are the two display methods for classical structure. Furthermore, linear Feynman diagrams are available as a display method, and finally, circular Feynman diagrams are also available (although VARNA uses chords instead of arcs for this display).

One of VARNA's greatest advantages is that is is written in Java^{TM}, allowing it to be used as a stand-alone software, part of an HTML page, or integrated into other Java^{TM} software via the available Application Programming Interface (API). This is an improvement over RNAViz's use of C, since Java^{TM} .jar files do not depend on the architecture in which they are used.

Unlike RNAViz, VARNA does not allow structures previously viewed to be recalled in any way, meaning every time a structure is loaded, it's layout needs to be recalculated and any changes the user may have made to the calculated layout need to be remade. Also, much like RNAViz, VARNA only allows the comparison of structures side by side, which can be daunting for large structures or structures with very close shape and nucleotide composition.

To address these drawbacks, and others that exist in other applications, Edward Glen


Seq: CAGCACGACACUAGCAGUCAGUGUCAGACUGCAIACAGCACGACACUAGCAGUCAGUGUCAGACUGCAIACAGCACGACACUAGCAGUCAGUGUCAGACUGCAIA

Str: ..(((((...(((((...(((((...(((((.....)))))...)))))).....(((((...(((((.....)))))...))))).....))))))...)))))..

Figure 2.9: An interface snapshot of VARNA, downloaded from http://varna.lri.fr/downloads.html

wrote the Java^{TM} based jViz.RNA software whose purpose is to ”assist in the analysis of structure predictions with three unique features : analysis of multiple structures using seven different visualization methods , dual graphs of RNA structures which highlight the topology of the RNA , and finally, the ability to overlay a predicted structure on top of the native structure of a given RNA in all visualizations apart from dual graphs which use a different comparison method” (cited from [14]). In essence, jViz.RNA offers users with the ability to inspect individual RNA strands and structures to appreciate the relationship between sequence and structure, as well as compare different RNA structures to give insight into how difference in sequence can cause difference in structure, or to compare predicted structures with known crystalized structures.

Aside from the representation of RNA classical structures (which jViz.RNA also does in a unique way), it offers the users to inspect structures as linear and circular Feynman plots, dot plots, and dual graphs. In regards to the classical representation of RNA, jViz.RNA takes a rather sophisticated approach drawn from the manner in which RNA behaves naturally. The reason that RNA doesn’t create overlap with itself where some bonds intersect others is because molecules and atoms have a natural repulsion force between each other. jViz.RNA simulates this repulsion force as well as the bonds between bases to create a simulation of the RNA structure. The result is a dynamic structure that naturally leans towards a layout of least overlaps, in the exact same manner as a real RNA molecule does. This algorithm will be discussed in more detail in Section 4.1, but the main point is that a minimum overlap structure is obtained without user intervention, relieving a degree of responsibility and manual labour from the user. Furthermore, Section 4.6 outlines how previously viewed structures and their layouts can be recalled in much the same manner as RNAViz employes skeleton files.

It is also appropriate to make note here of jViz.RNA's ability to display pseudo-knots. Pseudo-knots are very unique structural elements that appear in some instances of RNA. In essence pseudo-knots form when two base pairs(i, j) and (i', j') fulfill the condition $i < i' < j < j'$. This creates two bonds that intersect with each other (in 2D) (Figure 2.10).

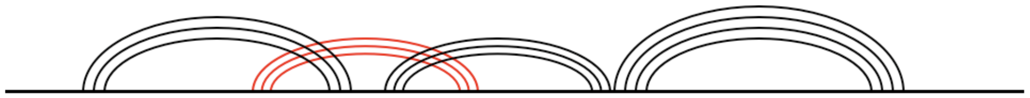
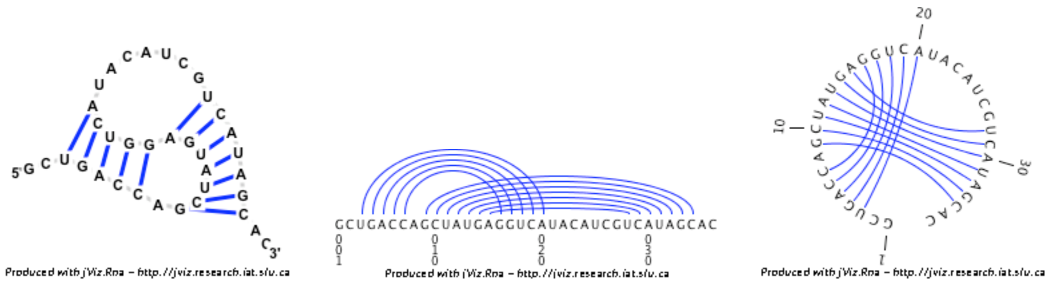


Figure 2.10: A pseudo-knot in an RNA sequence displayed as a linear Feynman diagram. The pseudo-knot is between the leftmost group of black arcs (bonds) and the red arcs (bonds)

Generally, these kinds of bonds are not easy to visualize. However, due to its unique method of drawing the RNA secondary structure, jViz.RNA displays a rather pleasing layout even if a sequence includes pseudo-knots.


Produced with jViz.Rna – http://jviz.research.iat.sfu.ca
GCUGACCAGCUAUGAGGUCAUACAUCGUCAUAGCAC
1
10
20
30

(a) The sequence displayed in classical structure mode

(b) The sequence displayed in linear Feynman mode

(c) The sequence displayed in circular Feynman mode

Figure 2.11: The different display methods of a pseudo-knot containing RNA sequence of bacteriophage T2 gene 32 mRNA pseudoknot; PDBID: 2TPK (obtained from http://cylofold.abcc.ncifcrf.gov/cylofold-0.1/sequenceJob/examples.gsp). All three images produced using jViz.RNA

Figure 2.11a shows how jViz.RNA renders a simple pseudo-knot when displaying a classical RNA structure, while Figure 2.11b shows how the same structure is displayed as a linear Feynman diagram and Figure 2.11c displays the same structure again, but as a circular Feynman diagram. In all three instances, the pseudoknot can be spotted immediately and seen very clearly.

Conclusively, it can be argued that jViz.RNA, as it stands, rivals two of the powerful RNA visualization software available. Furthermore, independently of other software, jViz.RNA has received excellent and encouraging feedback from satisfied users, as well as propositions for future developments. In fact, the design, implementation and results of these suggested developments is the focus of the thesis body that follows in Chapter 3.

2.3 Chapter Recap

This chapter introduces RNA and its important role in cells. Not only is it responsible for the transcription of genetic information from the genome to the protein building apparatus, but it is also an important structural unit in many processes in the cell. As a result it is important to have tools that specialize in the presentation of RNA structure information. In addition to the intuitive display of classical structure where the RNA structure is drawn with all its details, this chapter introduced other abstract presentation methods that emphasize particular elements in the structure (such as the bond arrangement in Feynman diagrams and the structure topology in dual graphs). In addition, this chapter drew a comparison between jViz.RNA, VARNA and RNAViz, and showed jViz.RNA's strengths rival two of the most used RNA visualization software. Glen has drawn a very in-depth comparison between jViz.RNA and other popular RNA visualization software and has also showed that jViz.RNA offers more versatile display than those tools in his work [14]. jViz.RNA also offers the unique attribute of creating a dynamic RNA classical structure that can be easily and intuitively manipulated by the user to satisfy their display needs, while also creating an automated system for the classical structure layout, both derived from the natural behavior of RNA.

Chapter 3

Support for RNAML and FASTA files

3.1 Original Supported Formats

jViz.RNA originally supported three file formats which were Connectivity Tables (.ct and .ct2) [32], Base-Pair Sequence (.bpseq) [8] and Dot-Bracket Notation (.dbn and .dbf) [16]. A comprehensive overview of these file formats and their origins is given in [14]. However, a short introduction of the files supported is necessary to understand the novelty of the extensions performed on jViz.RNA.

Dot-Bracket Notation (.dbn) and Dot-Bracket Format (.dbf) files are the simplest type of files. These contain two rows of information; the first row is the RNA sequence, and the row immediately below it is the dot-bracket notation of the sequence's structure. in jViz.RNA's implementation, the software accepts files which use the characters '(', ')', ':', '[' and ']' to represent the structure. Open ('(') and closed (')') brackets are used to denote bases that are paired up with each other, and a colon (':') is used to denote unpaired bases. Open and closed square brackets ('[' and ']') are used to denote bases that are paired in pseudo-knots.This method of representation can represent most RNA sequences, but doesn't do very well if pseudo-knots are interwoven into each other.

Connectivity Tables (.ct) and Base-Pair Sequence (.bpseq) files do better in describing complex structures. .bpseq files contain three columns across to describe each nucleotide in the sequence. The first column contains the index of the nucleotide (an integer between 1 and n where n is the sequence length), the second column contains the nucleotide symbol (either A, U, C or G), and the third column contains the index of the nucleotide that is paired with the current nucleotide (or 0 if the current nucleotide is unpaired). An optional header is also supported. A typical .bpseq file would look as follows:

```
Optional header
1 C 0
2 G 15
3 C 16
4 U 17
5 A 18
6 G 19
7 U 0
⋮
2307 U 0
2308 A 0
```

.ct files take on a similar format, but contain additional information across six columns, and a mandatory one line header that specifies the length of the sequence and the free energy of the structure. The first and second column in .ct and .bpseq files are equivalent. The third column specifies the index previous to the current nucleotide, while the fourth column specifies the index that follows it. The fifth column specifies the index of the nucleotide the current nucleotide is bound to, while the sixth column specifies the index of the current nucleotide again. A typical .bpseq file would look as follows:

```
2308 ENERGY=-44.90 Some Organism
1 C 0 2 0 1
2 G 1 3 15 2
3 C 2 4 16 3
4 U 3 5 17 4
5 A 4 6 18 5
6 G 5 7 19 6
7 U 6 8 0 7
⋮
2307 U 2306 2308 0 2307
2308 A 2307 0 0 2307
```

These files capture the information that is essential to construct a structure image very well, but present two problems; First, these files do not contain information that may be important for users (e.g. publication relating to the structure, modifications done on any of the bases, etc'), and second, the use of one file format over another is arbitrary and there is not attempt to institutionalize a standard format for the exchange of RNA information. Furthermore, there have been numerous request by jViz.RNA users to extend support for more files formats, mainly RNAML and FASTA. As a consequence, support for these two file formats was added and the details of that process are described in the following two sections.

3.2 RNAML - Explanation of the File Format and its Support in jViz.RNA

The RNA Markup Language (RNAML) is an extension of the Extensible Markup Language (XML) [29]. It is a markup language developed to exchange information about RNA in a convenient and reliable way, while allowing the easy extension of the language as the need arises. The RNAML file organizes the RNA information into five groups of data: Molecule information, interaction information, references, database-entries and analysis. The first two elements encapsulate the actual data regarding the RNA molecule(s) depicted in the file. Each RNA has its properties described in the molecule information section, If more than one RNA is depicted in the RNAML file then any interaction between the different RNA molecules is described in the interaction information element. The other three elements (references, database-entries, analysis) describe any papers that were used to obtain the information in the RNAML file, the database entries for the RNA molecule(s) described in the file, and any analysis that was used to obtain the information in the file. In general, the first two elements describe the molecule(s) data while the last three describe how that data was obtained and how it can be traced back to its sources.

Since RNAML is an extension of XML, the file structure for an RNAML file is similar to that of XML, meaning data would be arranged in sets of tags and values. The main difference being that in RNAML, some tags are necessary for the file to valid. A typical data element in an XML file would look like this:

```
<data collection name>
     <tag name1>
          value1
     </tag name 1>
     <tag name2>
          value2
     </tag name 2>
     <tag name3>
          value3
     </tag name 3>
     <tag name4>
          value4
     </tag name 4>
```

```
</data collection name>
```

And in RNAML, the overall structure of the file would be as follows:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE rnaml SYSTEM "rnaml.dtd" >
<rnaml version="1.0" >
   <molecule type="rna"  id="1" >
      <sequence>
         // Sequence information...
      </sequence>
      <structure>
        // Structure information such as how nucleotides are bonded and positioned...
      </structure>
   </molecule>
   <interactions>
      // Interaction information in the case more than one RNA molecule is described...
   </interactions>
   <reference>
     // References that were used to obtain this RNA information
   </reference>
   <database-entry>
     // Database entry information for the RNA molecules such as PDB IDs...
   </database-entry>
   <analysis>
     // Any other analysis that is relevant to this RNA molecule(s)...
   </analysis>
</rnaml>
```

Of the five major tags described here, only the first one is essential in order for an RNAML file to be valid for jViz.RNA. The first tag `<molecule>` supplies the user with all information regarding a single RNA molecule. All other tags simply add more information that might be useful in the context of studying that RNA molecule in the lab.

As mentioned before, RNAML is an extension of XML. This means that any XML parser would be able to parse the RNAML file. Luckily, Java contains a built in XML parsing API called the Document Object Model (DOM) [28]. DOM parses an XML document and stores it as a `Document` object, which is a tree like structure that stores the values of different elements as `Node` objects in the tree.

The first step in the incorporation of RNAML files into jViz.RNA was the subclassing of the `StructureFile` class and defining a `RnamlFile` class in the following matter:

```
public class RnamlFile extends StructureFile
{
         public RnamlFile(String pathname) // Constructor
         {
                   super(pathname);
         }

         public RnaStructure readFile()
         {
                  //Some code to read the file...
         }

         public boolean writeFile(ArrayList<Nucleotide> structure)
         {
                 // Some code to write the file...
         }

         public int getType()
         {
                 return FileUtils.RNAML;
         }
}
```

The `RnamlFile` class has to implement the four methods specified here in order for instances of this class to be declarable.

The constructor method, `public rnamlFile(String pathname)`, is responsible for setting a pointer to the file specified by `pathname`. This action is done by the constructor of the superclass, but this constructor has to call the superclass constructor with the command `super(pathname)`. In this particular case, the constructor also does some actions unique to the `RnamlFile` class such as initiating the *`Document`* object that would store the RNAML data.

The second method defined, `public RnaStructure readFile()`, reads the file pointed to by the *`RnamlFile`* object and builds an *`RnaStructure`* object to represent the data in the file. jViz.RNA then uses this *`RnaStructure`* object to perform all other operations such as building the graphs to draw and perform statistical calculations. This method definition is very long and as a result was not put in the thesis body for brevity purposes. It can be found in the appendices as part of the full `RnamlFile` class code.

The method `public boolean writeFile(ArrayList<Nucleotide> structure)` is used to write the information represented by the list of nucleotides `structure` into a file of the type represented by the current class. In this case it would be an RNAML file. However, in the current implementation, writing files as RNAML files is not yet supported by jViz.RNA.

Finally, the method `public int geType()` returns an integer value that denotes the type of file handled by the class in question. In this case it would a value that is defined as the constant `FileUtils.RNAML` to represent RNAML files.
The full code for the class can be seen in Appendix C.1

3.3 FASTA - Explanation of the File Format and its Support in jViz.RNA

FASTA is another widely used file format to exchange information regarding genetic material. However, usually FASTA is used to describe sequences, rather than structures. From personal correspondence that was received from other research groups some requested support for .fasta files that host information regarding RNA sequence. The FASTA files for which support was requested had the following format:

```
> header information (usually name of organism and sequence details)
sequence
dot-bracket notation of the structure
```

For example, a FASTA file could look like this:

```
>gb|AF034620.1|AF034620:5658-5779 Haloarcula marismortui rrnB operon 16S ribosomal RNA, 23S
ribosomal RNA, and 5S ribosomal RNA genes, complete sequence
UUAGGCGGCCACAGCGGUGGGGUUGCCUCCCGUACCCAUCCCGAACACGGAAGAUAAGCCCACCAGCGUUCCGGGGAGUACUGGAGUGCGCGAG
CCUCUGGGAAACCCGGUUCGCCGCCACC
...((((((....((((((((.(((.((.((((............)))).)).))).)))))).(((((((((......)))))))))))(((
((...((.....)).)))))))))))...
```

The FASTA files have one strict specification; the header (if present) is always a single line and contains no newline characters.

Much like with the RNAML file format, a new class needed to be defined for the FASTA files: `FastaFile`. The `FastaFile` class was defined as follows:

```
public class FastaFile extends StructureFile
    {
        BufferedReader inputStream;
        public FastaFile(String pathname)
        {
             super(pathname);
        }

        public RnaStructure readFile()
        {
             // Some code to read from the file this
             // class points to...
        }

        public boolean writeFile(ArrayList<Nucleotide> structure)
        {
             // Some code to write to a file...
        }//end writeFile(ArrayList<Nucleotide> structure)

        @Override
        public int getType()
        {
             return FileUtils.FASTA;
        }
   }
```

Due to the simple nature of the FASTA files for which support was requested, the `readFile()` method was simple to implement as long as two properties were kept in mind. First, as mentioned before, the header will always be the first line if it is present. Second, the sequence and structure information can extend over several lines.

The basic algorithm to implement the `readFile()` method would be as follows:

```
1. Read header
2. Read the sequence first line
    (a) If the first line is not valid
        i. Alert the user something is wrong
    (b) Otherwise
        i. Keep reading until an invalid line comes by
```

3. Read the first structure line
4. If it is not valid
    (a) Alert the user something is wrong
5. Otherwise
    (a) Keep reading the lines and make sure they are all valid
    (b) If an invalid line appears
        i. Alert the user something is wrong

Since, the sequence and structure elements can extend over several lines, it is important to make sure that algorithm makes note of when one element stops and the other begins. Hence in step 2.b.i, the program does not alert the user that something is wrong. The invalidity of a sequence line may indicate the beginning of the structure portion of the file.

Once the sequence and structure are both read. An `RnaStructure` object needs to be constructed from them by reading the sequence, and building the nucleotide chain, while keeping in mind the connections stored in the structure. For a dot-bracket notation, a `Stack` object was used to map the opening and closing brackets to nucleotide bonds. In essence, whenever an open bracket is encountered, that nucleotide number is pushed into the stack. Whenever a closing bracket is encountered, the last number in the stack is popped out, and the two nucleotides are declared as a nucleotide pair.

At the moment, jViz.RNA can only read single structure FASTA files (even though one FASTA file can store multiple structures) and does not support writing of FASTA files. The full code for the `FastaFile` class can be seen in Appendix C.2

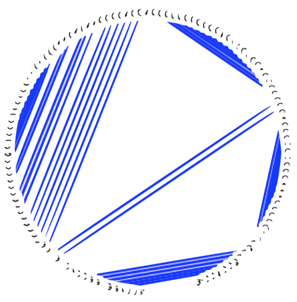
Chapter 4

Run Time Optimization

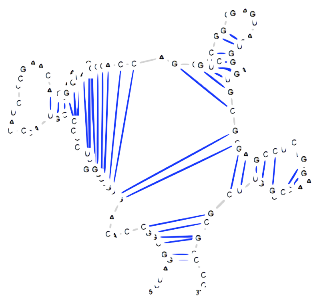
4.1 The Original Algorithms used by jViz.RNA

As mentioned in Section 2.2.3, jViz.RNA uses a very unique way of drawing the classical structure of RNA. It simulates the interaction between the nucleotides to create a dynamic structure that the user can manipulate. This approach is more time consuming than a simple layout of the RNA since some calculations need to be done iteratively in order to calculate the interactions of the nucleotides. In order to understand the need for any speed optimization of this aspect in jViz.RNA, it is necessary to first understand how jViz.RNA creates the dynamic model of the RNA structure.

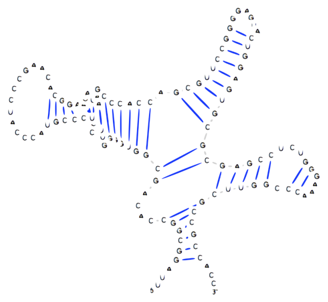
When a new structure is loaded into jViz.RNA, the nucleotides of the structure are first laid out in a circle (Figure 4.1).



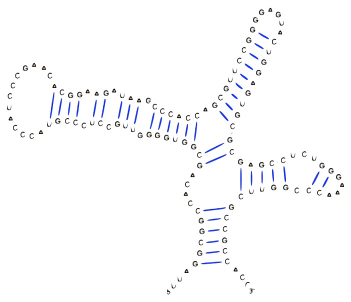
(a) The structure begins as a collection of nucleotides in a circle



(b) The forces pull the structure together



(c) As the structure collapses into itself, repulsion forces begin to stabilize the distance between nucleotides



(d) The finished structure layout

Figure 4.1: The stabilization of the RNA structure of *Haloarcula marismortui* rrnB operon 5S ribosomal RNA (AF034620:5658-5779)

The radius of the circle is proportional to the length of the structure. Each nucleotide is bonded to two other nucleotides to form the backbone (except the first and last nucleotides, which are only bonded to one other nucleotide like that), and possibly to another nucleotide in the structure to form a base-pair. Each nucleotide then experiences a set of attracting and repelling forces. The attraction comes from the other nucleotide it forms a base-pair with, while the repulsion comes from all other nucleotides in the simulation (including the one it is forms a base-pair with). The structure that is initially laid out in a circle (Figure 4.1a) begins collapsing into itself as the base-paired nucleotides pull each other closer together (Figure 4.1b). As the nucleotides of the structure come closer and closer together, the repulsion forces begin to strengthen between them, countering the effect of the attractive forces (Figure 4.1c). Finally, the structure stabilizes when the attraction and repulsion forces are equal in strength (Figure 4.1d).

The attractive forces between bonded nucleotides are calculated as if the bonds between them were springs and follow the general formula:

$$\vec{F} = k \times \Delta x$$

Where k is a constant and Δx is the difference in location between the two nucleotides bonded. As one can see, the force decreases as the distance between the two nucleotides decreases. On the other hand, the repulsion forces between every two nucleotides follows the formula:

$$\vec{F} = \frac{k_1 \times k_2}{(r_{21})^2}$$

Where k_1 and k_2 are repulsion constants between two nucleotides and r is the distance between the two nucleotides. One can see that in this instant, the force increases as the two nucleotides come closer and closer together. As mentioned before, as two nucleotides come together, the forces would work antagonistically to each other, and the nucleotides will reach a stable state when the two forces are equal in strength.

As noted before, this calculation would be done in iterations. At every iterations the forces on every nucleotide would be calculated, and once the force for every nucleotide would be calculated, the nucleotides will be displaced accordingly. The pseudo-code would be:

```
while (the structure is not stable) do
{
    for each nucleotide n1 of the n nucleotides do
    {
        if (the nucleotide is bonded in a base pair)
            calculate attracting forces
```

```
          for each nucleotide n2 of the other (n-1) nucleotides
          {
               calculate repulsion forces between n1 and n2
          }
     }
}
displace all nucleotides according to the forces acting on them
```

A quick inspection of this algorithm shows that the order of magnitude for such a simple implementation is $O(n^2)$, which could get problematic when the structure gets large. Indeed, such a simple implementation will not serve well when large structures such as ribosomal RNA are inspected. To compensate for that, jViz. RNA introduces a ”neighborhood” for each nucleotide, which is in essence a table of the distances between this nucleotides and all others. With the neighborhood set, each nucleotide only checks a selected number of nucleotides that are at a certain distance from it. Once the structure collapses close enough together, the neighborhood is discarded and the simple, brute force, method is used for the final iterations to account for all the nucleotides. The general pseudo-code used by jViz.RNA is as follows:

```
while (the structure is not stable) do
{
     for each nucleotide n1 of the n nucleotides do
     {
          if (the nucleotide is bonded in a base pair)
               calculate attracting forces
          retrieve n1's distance table
          if (the radius is large enough to account for all nucleotides)
          {
               for each nucleotide n2 of the other nucleotides
               {
                    calculate repulsion forces between n1 and n2
               }
          }
          otherwise
          {
               for each nucleotide n2 of the other nucleotides in the
               distance table
               {
                    calculate repulsion forces between n1 and n2
               }
               increase the radius by some factor
          }


     }
}
displace all nucleotides according to the forces acting on them
```

This version of the algorithm begins at an order of magnitude almost close to $O(n)$ and slowly increases the number of nucleotides added as the number of iterations increases. This approximation method assumes that if nucleotides are far enough, their repulsion forces would be unmeasurable compared to those that are close to it.

Although the employment of a neighborhood is a sophisticated enough solution for small structures, large structures still pose a problem for the later iterations. This slowdown at the later part of the stabilization process set the stage for the improvements discussed in the remainder of this chapter Section 4.2 discusses some improvements and measurements done over the summer, while Section 4.3, Section 4.4 and Section 4.5 introduce other optimization methods developed over the course of this thesis. Finally, Section 4.6 introduces an extension added to jViz.RNA that improves performance for subsequent inspections of a previously inspected sequence.

4.2 First Improvements to jViz.RNA

Some improvements were attempted at jViz.RNA to improve on the existing algorithm without introducing any new algorithms or computation methods [27]. The goal was to become closely familiar with the jViz.RNA code and search for optimizations that can be done to the code. It was determined that there were five potential bottlenecks that can be improved on:

1. The program was written in Java so there was overhead caused by using the Java Virtual Machine (JVM) as an intermediate between the executable .jar file and the operating system.

2. The algorithm for force calculation could be optimized in regards to the rate at which it expends the neighborhood of nucleotides every nucleotide takes into account.

3. Some display methods or rendering methods might cause a slight slowdown of the software.

4. Synchronized methods are common in the software, and these run slower than their non-synchronized counterparts

5. Method call overhead can create a noticeable slowdown if there are unnecessary method calls nested within loops

Although the third and fifth points were addressed to briefly in [27], and the first point is addressed to in this thesis in Section 4.5, the major improvement during the summer came from looking at the force calculation algorithm. The size of the radius of the neighborhood at every iteration was given by the line of code:

```
int radius = (distanceTableSize + 1) - (int)maxMotion
```

where `distanceTableSize` is a constant that stands for the maximum distance a nucleotide is from the current nucleotide of interest (i.e. the smallest neighborhood size that includes all nucleotides), and `maxMotion` is the maximum displacement done by any nucleotide during the previous iteration. Looking at this formula, one can see the radius is expressed as a linear equation of the form:

$$r = mx + n$$

where $n = (\texttt{distanceTableSize} + 1)$, $m = -1$ and $x = \texttt{maxMotion}$. In other words, the rate at which the neighborhood radius increases is a function of `maxMotion` and the coefficient *m*. Through a series of experiments, it was shown that there was merit in increasing the coefficient *m* from -1 to -20 for larger sequences. The increase in the *m* coefficient caused the neighborhood radius to increase in value slower than it used to. In essence, the algorithm remained in the order of $O(n)$ for a longer time duration. This adjustment cut the running time by 90% of its original value for the large *Haloarcula marismortui* rrnB operon 16S ribosomal RNA (AF034620:720-2191) sequence. One might think that this improvement would also be helpful for smaller sequence. However, experiments on the *Haloarcula marismortui* rrnB operon 5S ribosomal RNA (AF034620:5658-5779) showed that it actually caused a slowdown of the stabilization of the sequence. The set of experiment introduces two concepts that are the main focal points of this chapter. First, the idea that improvements, both code-wise and algorithm-wise, can be introduced into jViz.RNA to improve the speed of RNA stabilization, and second, that these improvements might depend on the size of the sequence they are applied on, and that for small sequences, the simplest, seemingly least efficient method, can be the fastest one.

4.3 The Barnes-Hut Algorithm

4.3.1 The Theory and Rationale Behind the Barnes-Hut Algorithm

The Barnes-Hut algorithm is an approximation algorithm invented as a solution to the n-body problem [3], which is in essence the one presented here. Given n-bodies, each of which exerts some force $\vec{F}$ on each of the other of the $n-1$ bodies in the system, one can approximate the force experienced on each body by the following idea: if a group of bodies is far enough from the target body, and close enough together, one can evaluate their combined force by creating a virtual body at the center of those bodies which encapsulates all their properties.

To demonstrate, suppose one has the following scenario: one wishes to computer the gravitational forces the body ***B*** experiences from the bodies T_1, T_2, and T_3 as shown in Figure 4.2.

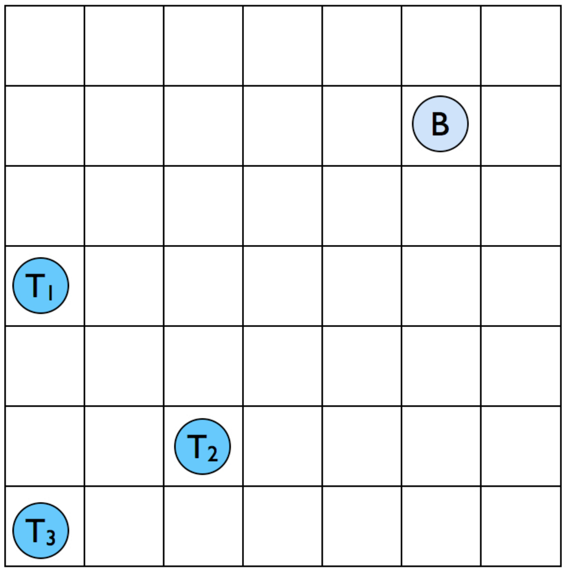


Figure 4.2: The body ***B*** is experiencing gravitational forces by three other bodies T_1, T_2, T_3

Suppose the gravitational forces are calculated as follows:

$$\vec{F} = \frac{k_1 \times k_2}{(r_{21})^2}$$

where k_1 is a repulsion constant of one body, k_2 is a repulsion constant of the second body and r_{21} is the distance between them. In order to calculate the force exerted on the body ***B*** from the three bodies, three calculations need to be done:

$$\vec{F}_1 = \frac{k_B \times k_{T_1}}{(r_{BT_1})^2}$$

$$\vec{F}_2 = \frac{k_B \times k_{T_2}}{(r_{BT_2})^2}$$

and

$$\vec{F}_3 = \frac{k_B \times k_{T_3}}{(r_{BT_3})^2}$$

However, one can construct a virtual body, ***V*** where $k_v = k_1 + k_2 + k_3$ and ***V*** lays at an equal distance from T_1, T_2, and T_3 as shown in Figure 4.3.

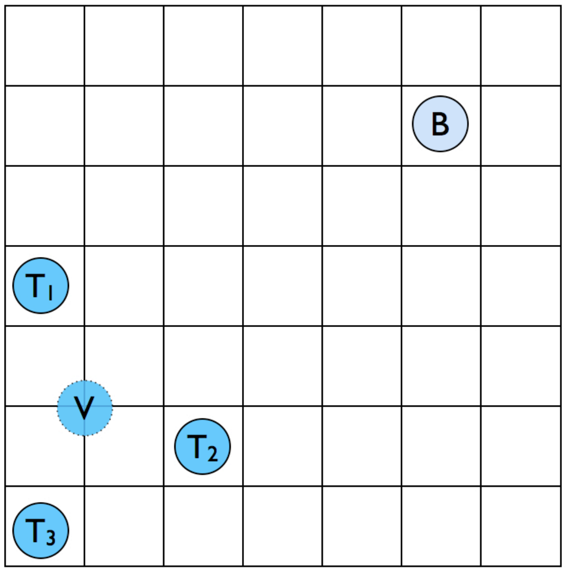


Figure 4.3: The three bodies T_1, T_2, T_3 can be replaced by a virtual body V

Then, the force exerted on ***B*** can be approximated by:

$$\vec{F} = \frac{k_B \times k_V}{(r_{BV})^2}$$

Granted, even though the number of force calculations was reduced, new calculations required to build the new body ***V*** were introduced. Furthermore, the use of this method is conditioned on the fact that the area enclosing the bodies approximated is far enough from the body ***B***, and small enough. This would be the area shown enclosed in orange in Figure 4.4.

This might actually increase the computational time required to calculate the forces on one body. However, if this virtual body is created once, and then used for all bodies far enough, the computational complexity would decrease.

In the Barnes-Hut algorithm, the area containing the bodies is recursively subdivided into four partitions in 2D and a quad-tree (or an oct-tree for 3D) is constructed (Figure 4.5).

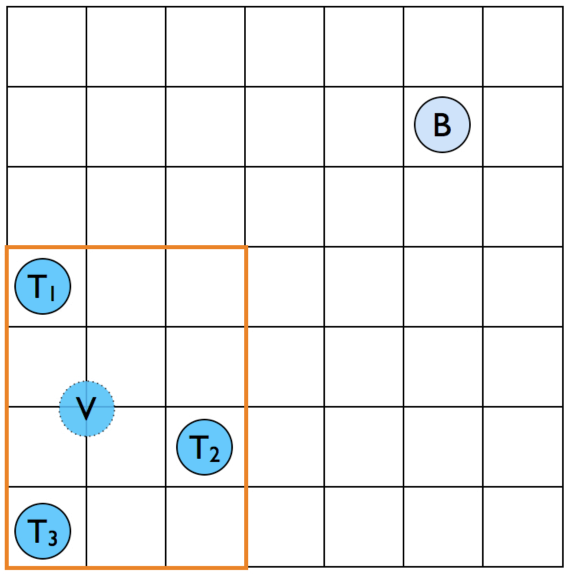
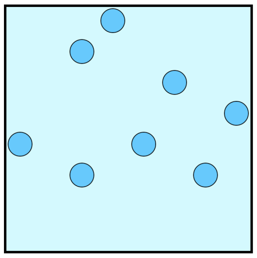
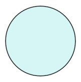
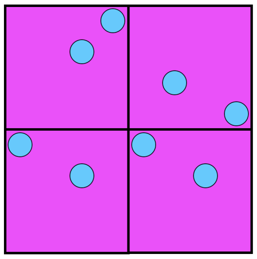


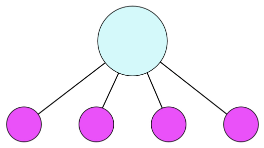
Figure 4.4: The area which encompass the three bodies T_1, T_2, T_3

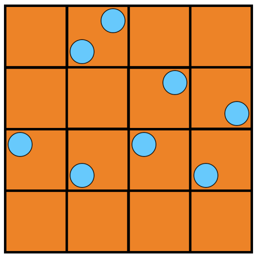
Each of the leafs in the tree is a body from the system or an empty space, and each of the nodes is an area that contains four bodies (or virtual bodies). The tree is build once, and then all the bodies in the system have the forces exerted on them calculated using the tree. A Depth-First Search is run on the body through all the branches in the tree until it hits a leaf, or a node that is small enough and far enough (denoting a collection of bodies close enough together and far enough). This size to distance ratio is usually expressed as $\theta = \frac{d}{r}$ where d is the size of the space (the orange rectangle in Figure 4.4), and r is the distance from the node currently looked at (such as ***B*** in Figure 4.4), to the virtual body (***V*** in Figure 4.4).The force is then calculated on that body from that node, and the search continues through all other branches in the same manner until there are no more branches to explore. Figure 4.5 demonstrates how the quad tree is build in a space with eight bodies. The entire space (Figure 4.5a) is represented by the root (Figure 4.5b). The space is then divided into four subspaces (Figure 4.5c) and four children, each representing a subspace, are added to the root (Figure 4.5d). Since there is more than one object in each space, the subspaces are divided again (Figure 4.5e) and each of the previous nodes receives four children (Figure 4.5f). Finally, there are some nodes which represent empty spaces, and some represent spaces where there is more than one object. The latter are subdivided into four subspaces again (Figure 4.5g) and those nodes whose spaces were divided receive four children (Figure 4.5h). Since each leaf in the tree now represent an empty space or the space just around a node, no further division occurs.

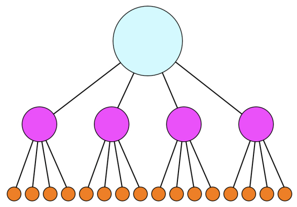

(a) The space undivided

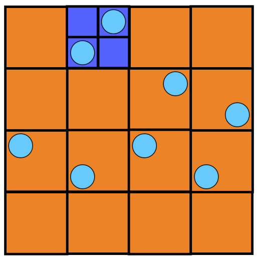

(b) The undivided space is depicted by the root


(c) The space is divided once into four subspaces


(d) Four nodes are added to the root to depict the subspaces


(e) The four subspaces are divided again, each into four subspaces


(f) More nodes are added to depict the new spaces


(g) Some spaces still contain more than one object, so they are divided again

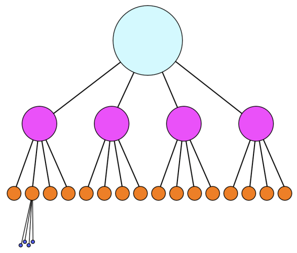

(h) Nodes are added to represent the last division, now each node represent a body, or an empty space

Figure 4.5: Construction of a Quad-Tree in a 2D space with eight bodies

4.3.2 Implementation of the Barnes-Hut Algorithm

To implement the algorithm, we first need to define the nodes in the tree and the tree itself. The node is defined as follows:

```
public class QuadTreeNode {
     private boolean empty = true;     // Whether this node is empty or not
     private boolean leaf = true;     // Whether this node is a leaf or not in the tree
     private boolean hasChildren = false;     // Whether this node has children
     private QuadTreeNode parent;     // A pointer to the parent. This will be used to update
                                      // the tree
     mass center
     private QuadTreeNode child1, child2, child3, child4; // Pointers to the children, always 4
     private int mass = 0;        // The mass of this node. The mass will always be an integer
     multiple of 1
     private double centerOfMassX, centerOfMassY;   // The x and y co-ordinates of the center of
                                                    // mass of this node
     private double centerX, centerY;     // The x and y co-ordinates of the center of this node.
                                          // This is NOT the same as the center of mass.
                                          // This is the center that this node forms the division
                                          // in if it has children.
     private double size;
     public int repulsion;



     // Empty constructor
     public QuadTreeNode()
     {
          // Some initialization code...
     }//end constructor

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

     // A constructor with some default parameters
     public QuadTreeNode(QuadTreeNode theParent,
                         int theMass,
                         double theXComponent,
                         double theYComponent,
                         double centerXComponent,
                         double centerYComponent,
                         int theRepulsion)
     {
          // Some more initialization code...
     }//end 2nd constructor

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

     // This method will create 4 children for THIS node
     public void createChildren()
     {
          // Create four empty children...
     }//end createChildren
```

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

    // This method updates the center of mass and mass for this node and the subtree below it.
    // It uses recursion
    public void updateMassAndCenterOfMass()
    {
         // Updates THIS node's children's center of mass and mass first (through recursion)
         // and then calculates THIS node's mass and center of mass...
    }//end updateMassAndCenterOfMass

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

    // The toString() method

    public String toString()
    {
         /// Returns a string representation of THIS node...
    }//end toString


}
```

Each node can represent an empty space, a space with one body, or a space with many bodies. Leafs represent empty spaces and single bodies, while internal nodes represent spaces with multiple bodies.

Each node stores its location in the `centerX` and `centerY` attributes while storing its center of mass location in the `centerOfMassX` and `centerOfMassY` attributes. For leafs, the two sets of coordinates are the same since the center of mass and the center of a body are the same. However, for internal nodes, which represent virtual bodies, the center of the node is the center of the subspace it represents while the center of mass at that node is the center of mass of the virtual body it represents.

Each node also stores a `mass` attribute which acts as the repulsion coefficient k in this context. For leafs representing a single body, that mass is 1 (naturally, for leafs representing an empty space that mass is 0). For internal nodes, their mass is the sum of all the masses of their children. This means every internal node's mass is an indicator for how many bodies it represents. Furthermore, the mass is a key attribute in calculating the center of mass for internal nodes. The center of mass cannot be a simple average of the center of mass of all four children, it must be a weighted average, and the weight comes from the `mass`. In essence, for every internal node n that has four children: c_1, c_2, c_3 and c_4 the mass would be:

$$m_n = m_{c_1} + m_{c_2} + m_{c_3} + m_{c_4}$$

and the center of mass would be:

$$X_{m_n} = \frac{(m_{c_1} \times X_{m_{c_1}}) + (m_{c_2} \times X_{m_{c_2}}) + (m_{c_3} \times X_{m_{c_3}}) + (m_{c_4} \times X_{m_{c_4}})}{m_n}$$
$$Y_{m_n} = \frac{(m_{c_1} \times Y_{m_{c_1}}) + (m_{c_2} \times Y_{m_{c_2}}) + (m_{c_3} \times Y_{m_{c_3}}) + (m_{c_4} \times Y_{m_{c_4}})}{m_n}$$

Where X_{m_n} is the x-coordinate of the center of mass of node n (`centerOfMassX`), Y_{m_n} is the y-coordinate of the center of mass of node n (`centerOfMassY`), m_n is the mass of the node (`mass`), m_{c_1}, m_{c_2}, m_{c_3} and m_{c_4} are the masses of the four children, $X_{m_{c_1}}$, $X_{m_{c_2}}$, $X_{m_{c_3}}$ and $X_{m_{c_4}}$ are the x-coordinates of the center of mass of the four children, and $Y_{m_{c_1}}$, $Y_{m_{c_2}}$, $Y_{m_{c_3}}$ and $Y_{m_{c_4}}$ are the y-coordinates of the center of mass of the four children.

With the nodes defined, the next step was defining a quad-tree that employes the nodes and allows DFS to be performed. The tree class was defined as follows:

```
public class QuadTree {

    private QuadTreeNode root;
    private double topLeftX, topLeftY, bottomRightX, bottomRightY;
    private static final int CHILD1 = 1;
    private static final int CHILD2 = 2;
    private static final int CHILD3 = 3;
    private static final int CHILD4 = 4;
    private static final double theta = 5.0;


    // ....

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

    // This quad tree can only be defined within a region. The region is a rectangle determined
    // by its top left point and bottom right point. Hence there is no empty constructor.
    public QuadTree(double newTopLeftX,
                    double newTopLeftY,
                    double newBottomRightX,
                    double newBottomRightY)
    {
        // Initialization code...
        // Notice there is no empty constructor
    }

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

    // Inserts the node [theNode] in its rightful place
    public void insertNode(QuadTreeNode theNode)
    {
     // Code to insert the node...
    }//end insertNode

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

    // This method updates the tree's center of mass and mass from the root down
    public void updateMassAndCenterOfMass()
    {
        // In essence this method calls the root's updateMassAndCenterOfMass() ...
    }//end updateMassAndCenterOfMass
    /*******************************************************************************************/

    // This method updates the forces on the node [theNode] and stores them in the array dxdy
    // where dxdy[0] = dx and dxdy[1] = dy. The array dxdy is assumed to have exactly
    // two elements.
    public void updateForcesOnNode(double nodeX,
                                   double nodeY,
                                   int nodeRepulsion,
                                   double[] dxdy,
                                   double rigidity,
                                   boolean justMadeLocal)
    {
        // Code to update the forces from the entire tree...
    }//end updateForcesOnNode
    /*******************************************************************************************/

}
```

The first thing that is important to note are the attributes `topLeftX`, `topLeftY`, `bottomRightX`, and `bottomRightY`. These define two points that define the rectangle enclosing the space which the tree represents. Only two points are needed to describe the rectangle since the other two points can be assembled from the available coordinates (Figure 4.6)

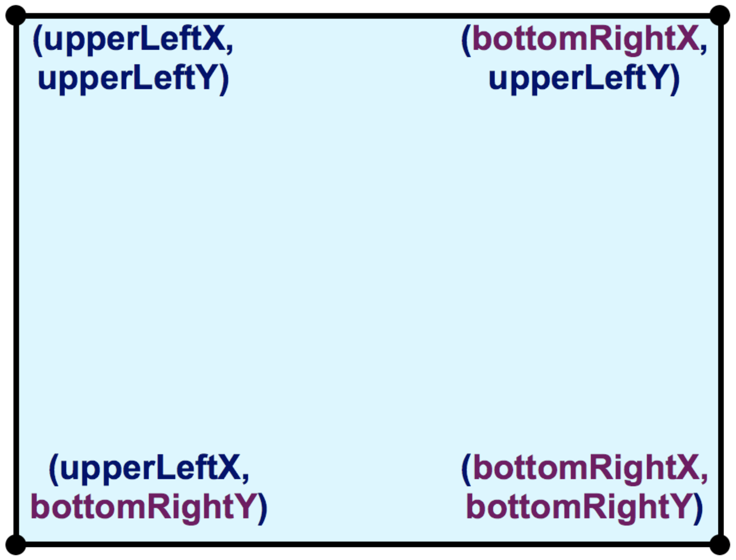


Figure 4.6: The rectangular area is defined by two points `(upperLeftX, upperLeftY)` and `(bottomRightX, bottomRightY)`, the other two points have to be `(upperLeftX, bottomRightY)` and `(bottomRightX, upperLeftY)`

The constant defined as `private static final double theta = 5.0` refers to the θ that describes the ratio between a space's size, and its distance from a node. The `theta` constant represents the minimum value this ratio must meet in order for the approximation to take place. In other words, if the size of the space (d), divided by its distance from a node n (r) is smaller than or equal to `theta`, then the bodies within the space are approximated using a virtual body.

The methods shown here are the most important methods in the tree. First, it is noteworthy that there is no empty constructor and a tree must be initialized within a certain region. In jViz.RNA, the location of the greatest and smallest values of x-coordinates and greatest and smallest values of y-coordinates out of all nodes are used. Also, every tree requires a method to insert a node, in this case it is `public void insertNode(QuadTreeNode theNode).` It is worth pointing out this tree does not contain any code to remove a node since bodies do not disappear and the tree is reconstructed at every iteration.
The next method shown is `public void updateMassAndCenterOfMass().` This method actually calls the update on the root which recursively calls the method first on its children, and then updates its own mass and mass center accordingly. Finally, the most important method in the tree is probably `public void updateForcesOnNode(double nodeX, double nodeY, int nodeRepulsion,`
`double[] dxdy, double rigidity, boolean justMadeLocal).` This method takes the information of a body (`nodeX, nodeY, nodeRepulsion, rigidity,`
`justMadeLocal`) and calculates the forces activated on this body using the tree and the Barnes-Hut Algorithm. The force's x and y components are then stored in the two dimensional array `dxdy` which is a 2×1 array. Since arrays in Java are essentially pointers, the values written to `dxdy` are available as the method returns (equivalent to passing by reference in C++). This method is called on every body in the system, and the forces are calculated on that body from every body, or collection of bodies far enough. When the method returns the sum of forces' components is available in `dxdy`, since adding vectors is equivalent to adding their individual components individually.

Every iteration of the Barnes-Hut algorithm involves the following steps:

1. Build a quad-tree of all the bodies in the system (order of $O(n)$)

2. Perform Depth-First Search (DFS) on the tree for every body in the system and calculate the forces operating on each body (order of $O(log\ n)$ for each of the n bodies)

3. Once all bodies have been used to search the tree, displace them according to the forces applied on each body (order of $O(n)$)

One can see from this flow of events that even though the search time has been decreased, the additional operation of building a tree at each iteration may increase the time required to perform the computation. Indeed, the following section demonstrates several different cases and how well the Barnes-Hut algorithm works in each. The full code for the `QuadTreeNode` and `QuadTree` can be seen in Appendix B.1 and Appendix B.2

4.3.3 Measured Results and Changes in Running Time

Sixteen sequences were used for these experiments: *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080), *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579), *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627), *Arthrobacter globiformis* 5S ribosomal RNA (M16173), *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515), *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122), *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416), *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387), *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605), *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601), *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511), *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876), *Aureoumbra lagunensis* 18S ribosomal RNA (U40258), *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345), *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) and *Chlorella saccharophila* 18S ribosomal RNA (AB058310). Each of the sequences was loaded to jViz.RNA and had 20 measurements of the elapsed time between the sequence being loaded and the sequence stabilizing (no further movement occurs). This time was referred to as the 'stabilization time'. The measurement was performed using Java based `ThreadMXBean` factory object. The code to capture the time used to perform the stabilization was:

```
ThreadMXBean bean = ManagementFactory.getThreadMXBean();
long startUserTimeNano = bean.getCurrentThreadUserTime();

//perform task ...

long taskUserTimeNano =-(startUserTimeNano- bean.getCurrentThreadUserTime());
jVizCore.CPUnanosecondsElapsed += (taskUserTimeNano/1000000.0);
```

Where `jVizCore.CPUnanosecondsElapsed` is the time required to perform the task that is eventually printed as output. This value is reported in milliseconds while `ThreadMXBean` reports in nanoseconds.

The intuition was that for longer sequences the Barnes-Hut algorithm would prove to be useful, and shorten the time required to stabilize. On the other hand, short sequences may take less time to stabilize under the original algorithm and would have their stabilization time increased under the optimization methods. The running times for all 16 sequences are shown in Figure 4.7 and a table outlining the size of each sequence can be seen in Table 4.1.

Sequence Name	Sequence Length (nt)
Bacillus stearothermophilus 5S ribosomal RNA (AJ251080)	117
Saccharomyces cerevisiae 5S ribosomal RNA (X67579)	118
Agrobacterium tumefaciens 5S ribosomal RNA (X02627)	120
Arthrobacter globiformis 5S ribosomal RNA (M16173)	122
Deinococcus radiodurans 5S ribosomal RNA (AE000513:254392-254515)	124
Metarhizium anisopliae var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122)	436
Tetrahymena thermophila 26S ribosomal RNA fragment (with intron) (V01416)	517
Acomys cahirinus mitochondrial 12S ribosomal RNA (X84387)	940
Xenopus laevis mitochondrial 12S ribosomal RNA (M27605)	945
Homo sapiens mitochondrial 16S ribosomal RNA (J01415:648-1601)	954
Ailurus fulgens mitochondrial 16S ribosomal RNA (Y08511)	964
Sulfolobus acidocaldarius 16S ribosomal RNA (D14876)	2080
Aureoumbra lagunensis 18S ribosomal RNA (U40258)	2236
Hildenbrandia rubra 18S ribosomal RNA (with intron) (L19345)	2283
Porphyra leucosticta 18S ribosmal RNA (with intron) (AF342746)	2404
Chlorella saccharophila 18S ribosomal RNA (AB058310)	2510

Table 4.1: The sequences used and their respective sizes

Despite the large confidence interval error bars present for the original algorithm used, there are several facts made clear by the graph in Figure 4.7. First, it is evident that all optimization attempts (introducing a multiplier as discussed in Section 4.2, or implementing the Barnes-Hut algorithm) have yielded much lower running times than the original algorithm used by jViz.RNA. Furthermore, it can be argued from this graph that the Barnes-Hut offers relatively little improvement over the optimization discussed in Section 4.2. Third, it is evident that aside from the five very short sequences used whose length is between 100nt and 130nt, Figure 4.7 shows that the improvements are valid for all sequences, implying that the large majority of sequences that would be analyzed by jViz.RNA would benefit from the integration of the optimization methods. Figure 4.8, Figure 4.9 and Figure 4.10 further support the third point.

However, if one looks at Figure 4.11, one can see that the original algorithm performs more or less as well as the optimized methods. Only the use of the Barnes-Hut algorithm with a high θ value offers an improvement in the stabilization time of the structures between 100nt and 130nt in length. In fact, for all sequences, the use of the Barnes-Hut algorithm with a high θ value offers a running time that is indistinguishable from an order of $O(n)$.

Even so, it is important to keep in mind that even though an optimization of run-time is the primary focus of this thesis, run-time is not the primary focus of this software. The primary focus is visualization and information presentation. As a result, it is important to look at each optimization method and inspect if it causes any distortion large enough to obstruct information presentation. If a certain method performs the required calculations faster, but the resulting sequence is convoluted and the layout is overwhelming to users, the method is counterproductive.

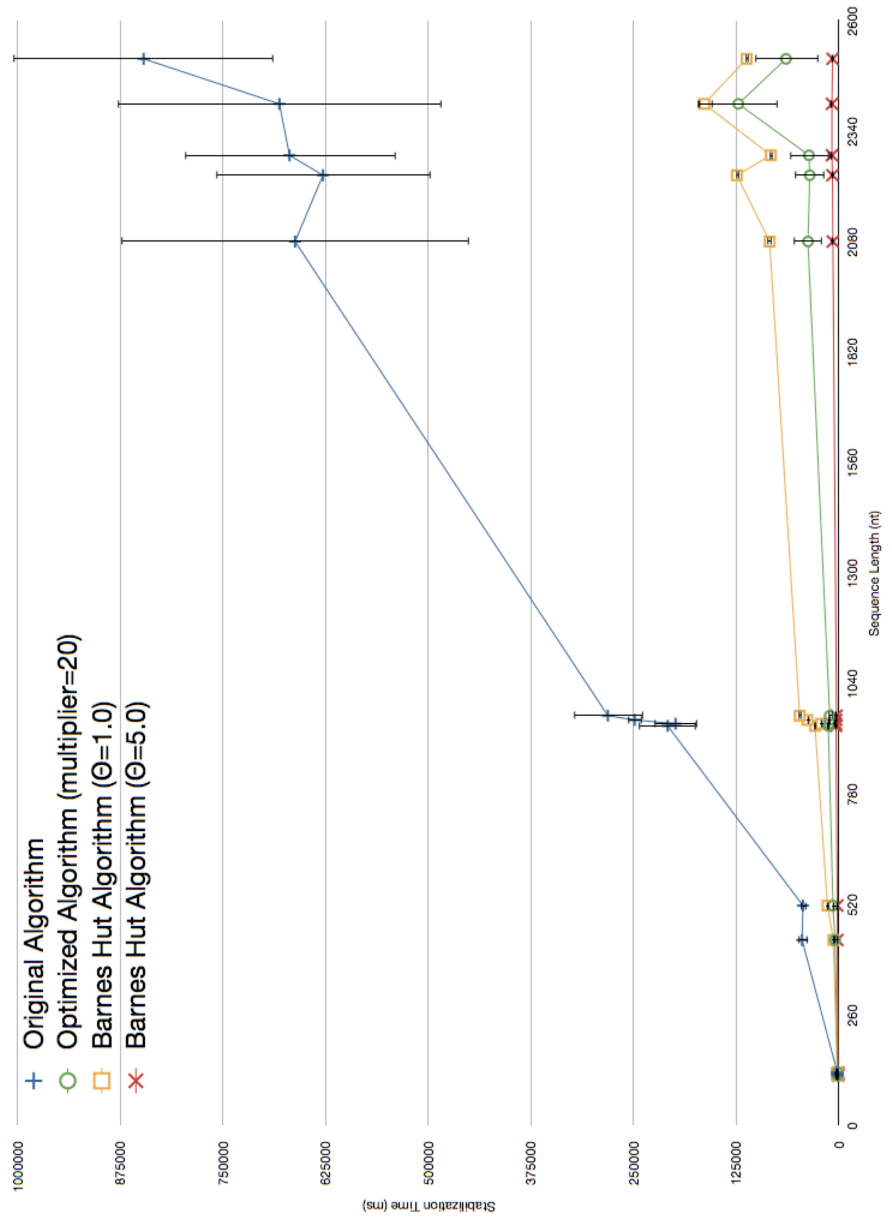


Figure 4.7: The stabilization times (in milliseconds) for the different RNA sequences using the optimized algorithm presented in Section 4.2 and Barnes-Hut algorithm under different θ values. 99% confidence intervals are used as error bars

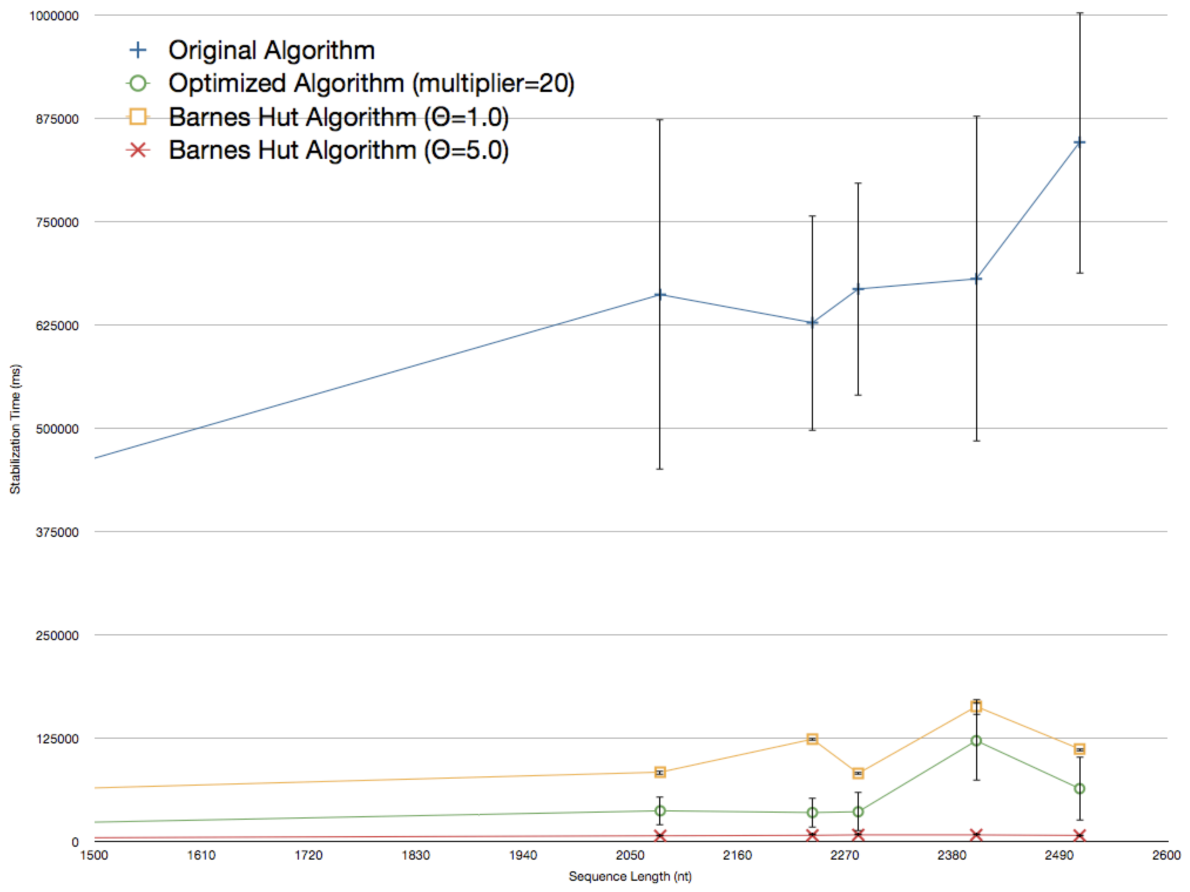


Figure 4.8: The stabilization times (in milliseconds) for *Chlorella saccharophila* 18S ribosomal RNA (AB058310), *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746), *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345), *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) and *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the optimized algorithm presented in Section 4.2 and Barnes-Hut algorithm under different θ values. 99% confidence intervals are used as error bars

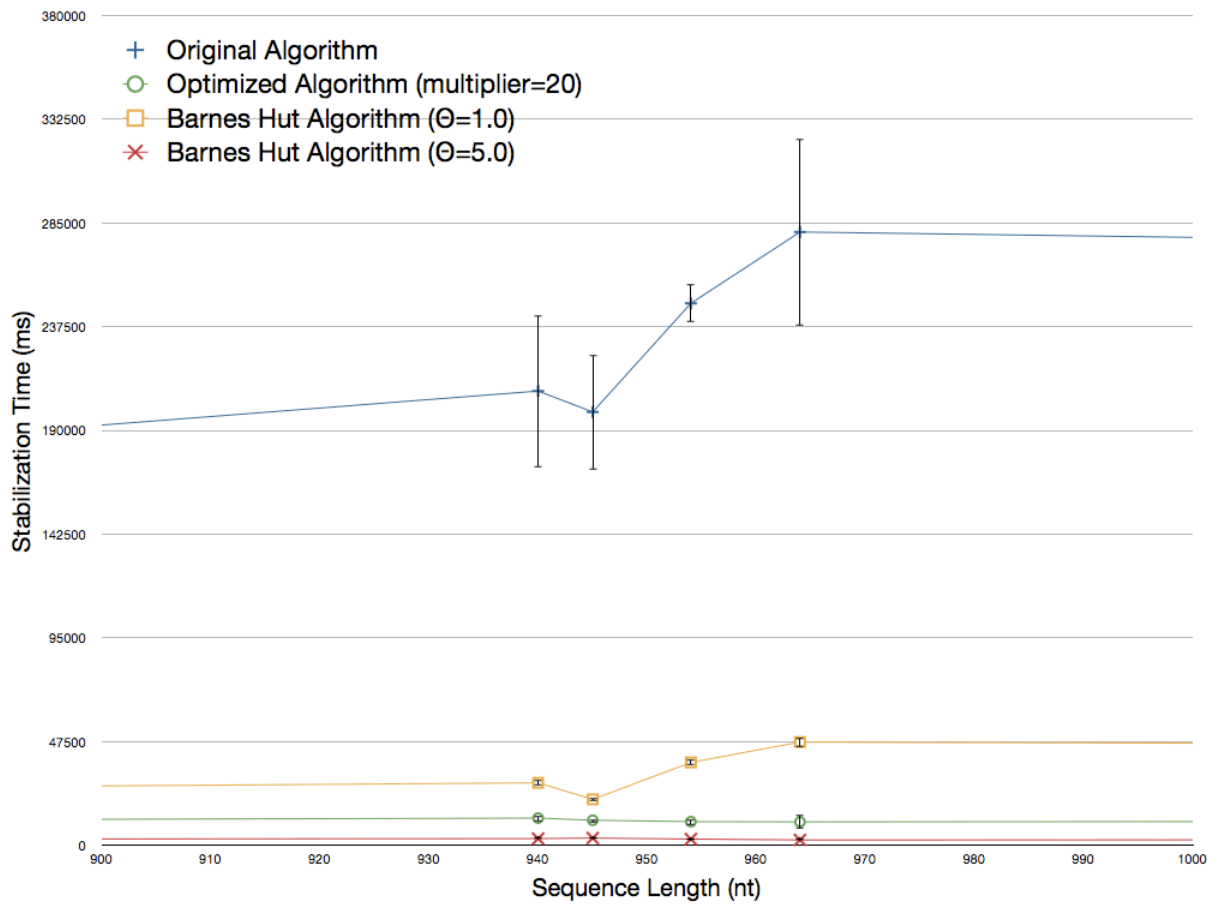


Figure 4.9: The stabilization times (in milliseconds) for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511), *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601), *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) and *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the optimized algorithm presented in Section 4.2 and Barnes-Hut algorithm under different θ values. 99% confidence intervals are used as error bars

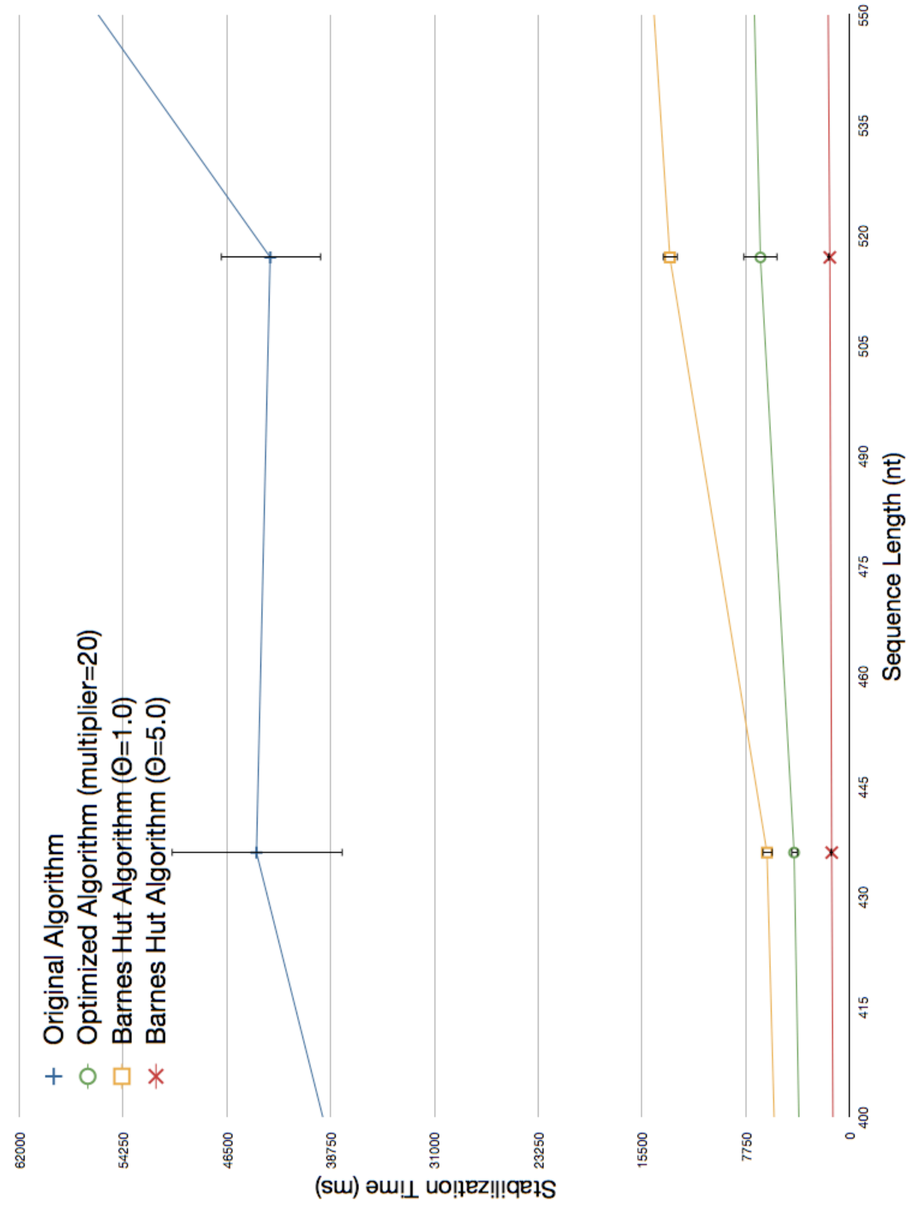


Figure 4.10: The stabilization times (in milliseconds) for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) and *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the optimized algorithm presented in Section 4.2 and Barnes-Hut algorithm under different θ values. 99% confidence intervals are used as error bars

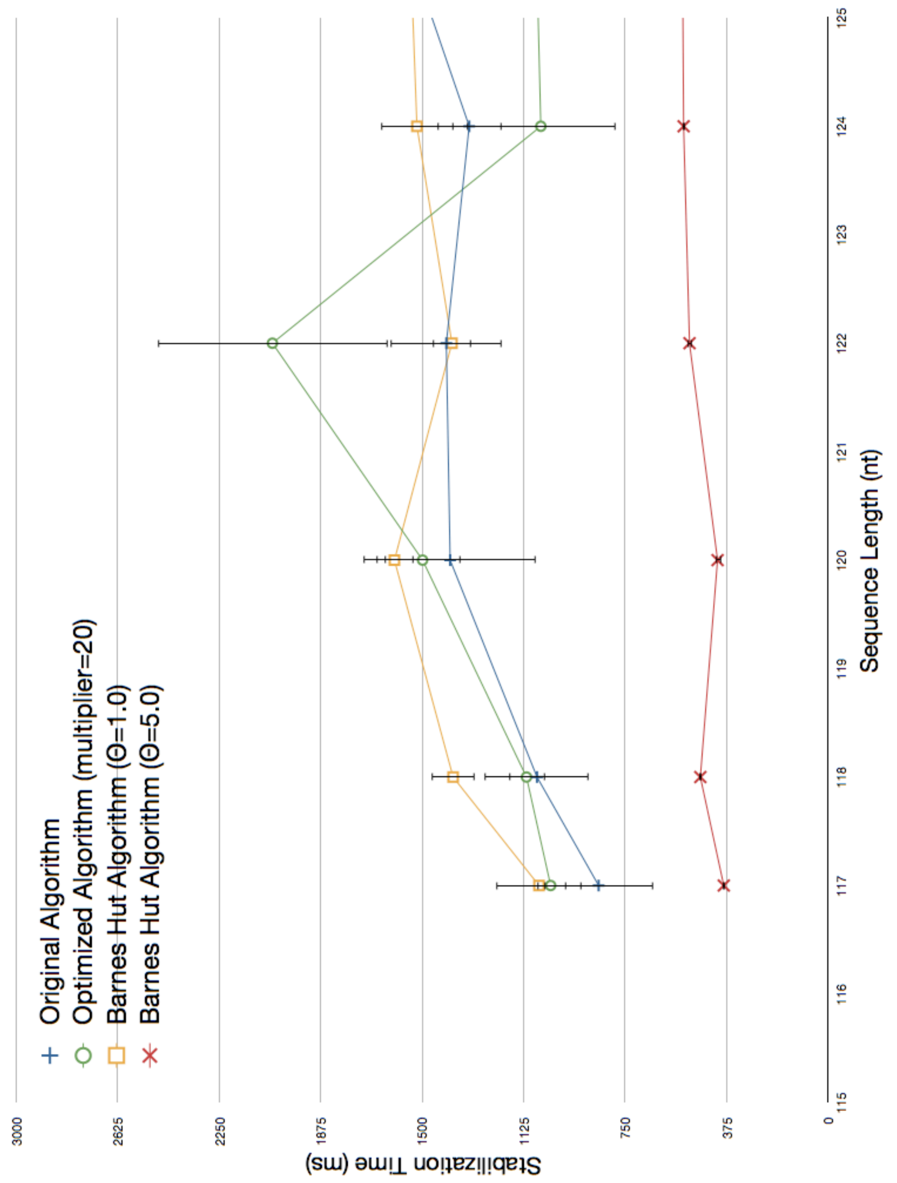


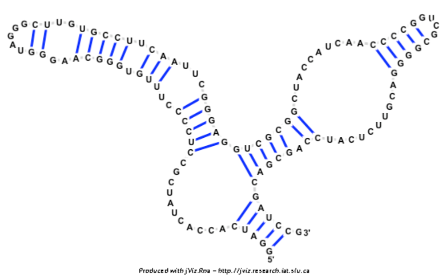
Figure 4.11: The stabilization times (in milliseconds) for *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515), *Arthrobacter globiformis* 5S ribosomal RNA (M16173), *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627), *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579) and *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080) using the optimized algorithm presented in Section 4.2 and Barnes-Hut algorithm under different θ values. 99% confidence intervals are used as error bars

4.3.4 Effects on Structure Presentation

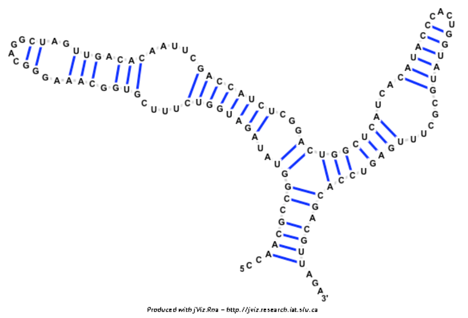
The run time required for a structure to stabilize is a major concern for any user interested in inspecting large RNA sequences. However, decreasing the run time cannot be done at the expense of the main function jViz.RNA offers; namely, the presentation of RNA structural elements in a clear manner. This section inspects the effect that employing the algorithm based optimization methods has on sequences' layout. Since both the Barnes-Hut algorithm and the previous optimization method exchange computational accuracy for computational speed, one would expect to see some degree of difference between the algorithms. The major issue is the amount of leniency that can be given to an algorithm before its graphical output no longer serves jViz.RNA's original purpose.

Figures 4.12 - 4.15 demonstrate the structure layouts resulting from using the original algorithm jViz.RNA employes. These complete set of sequences is not presented here for brevity purposes. Instead, a selected set of sequences is used throughout this chapter to illustrate the effect different optimization methods have on the sequences. The remaining eight sequences' layouts under jViz.RNA's original algorithm can be found in Appendix A, Figures A.1 - A.8. The main points to consider are the way these layouts show structural elements such as loops and stems in the structure, and the level of overlap and entanglement that the structures have. Looking at Figure 4.12a one can spot an entanglement in the structure of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515), and another one can be seen in Figure 4.15. However, these are all mild distortions that can be easily spotted, and later adjusted, by the user (since the structure is interactive). Generally though, the original algorithm does a very good job of displaying a structure layout that allows for easy identification of elements that may be significant to users. The layouts are also fairly sparse so that there usually isn't overlap between structural elements.

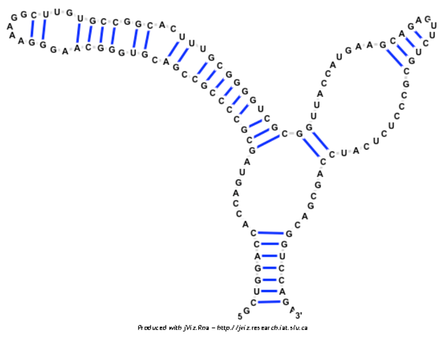
In essence, the structures depicted in Figures 4.12 - 4.15 will be the standard against which the other algorithms' layouts would be measured. The results are not meant to be an exact match, or close to one, but are meant to still create structures with minimal overlap where structural elements can be spotted easily.



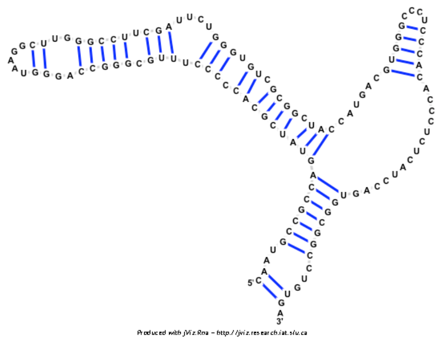
(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)



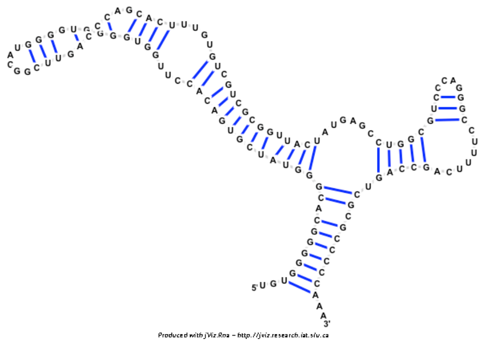
(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)



(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)



(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)



(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.12: The resulting sequence layout for the shortest five sequences using the original algorithm employed in jViz.RNA

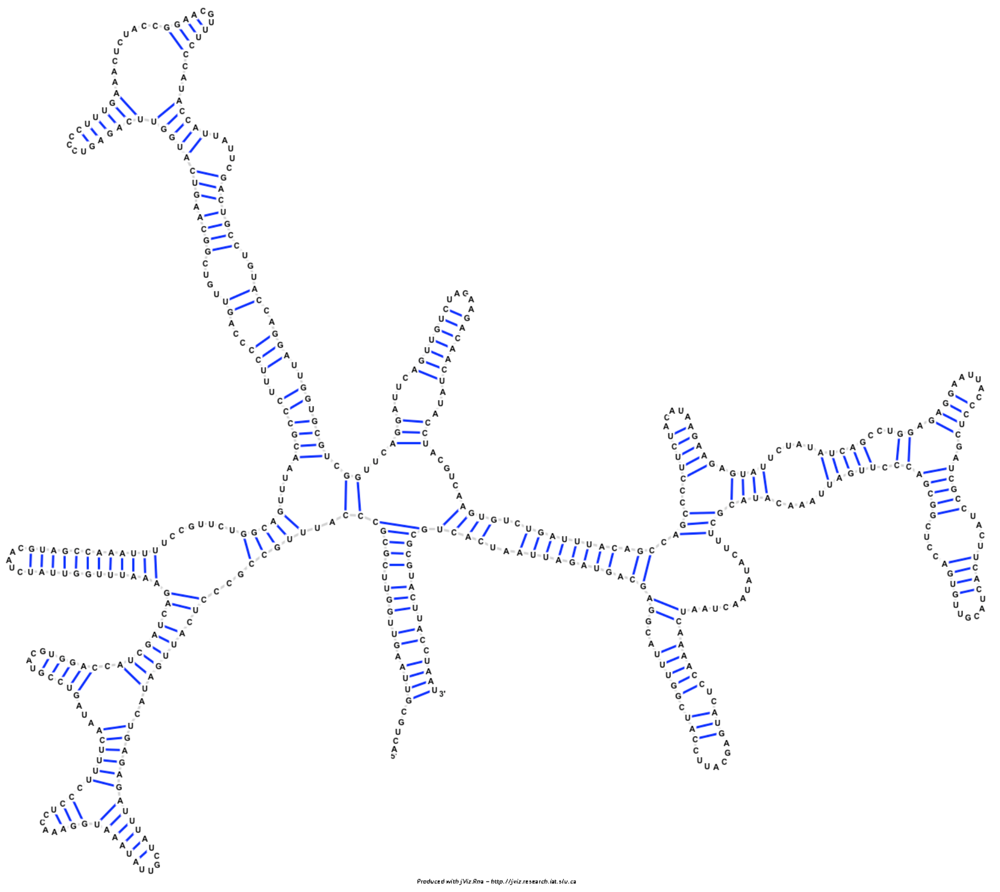


Figure 4.13: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using the original algorithm employed in jViz.RNA

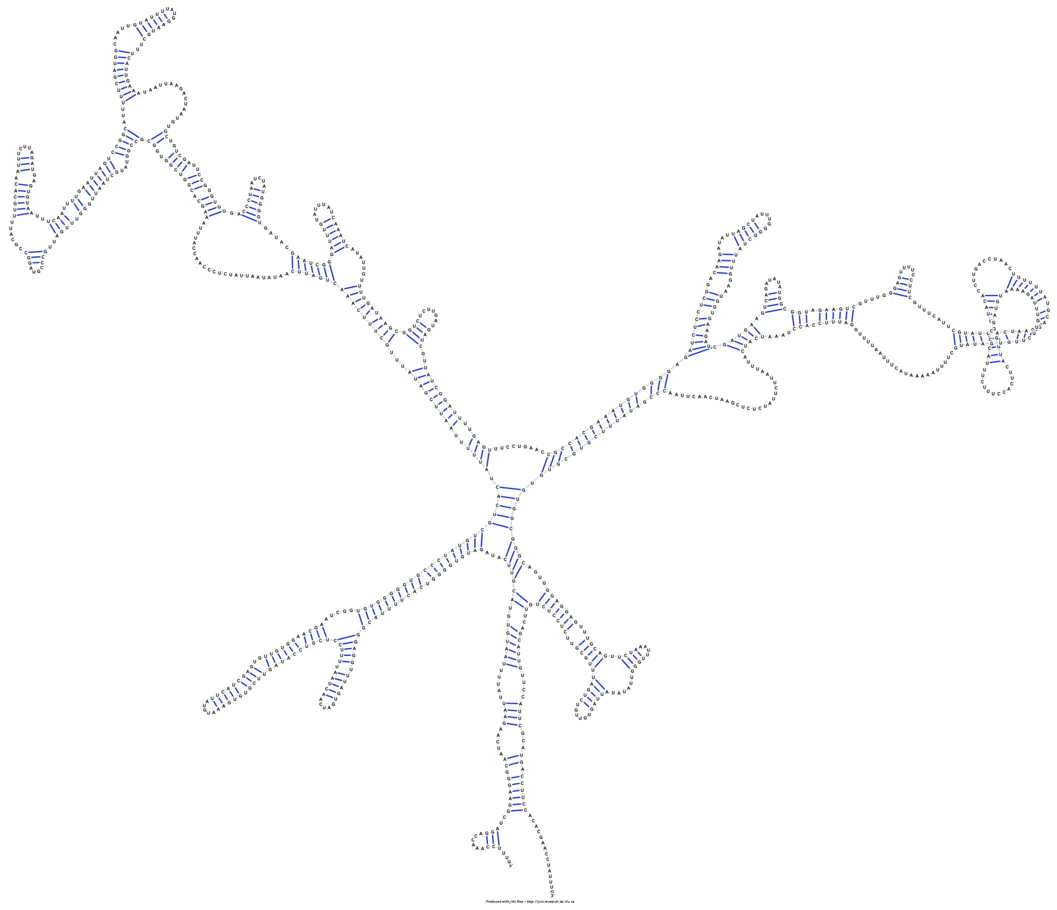


Figure 4.14: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using the original algorithm employed in jViz.RNA

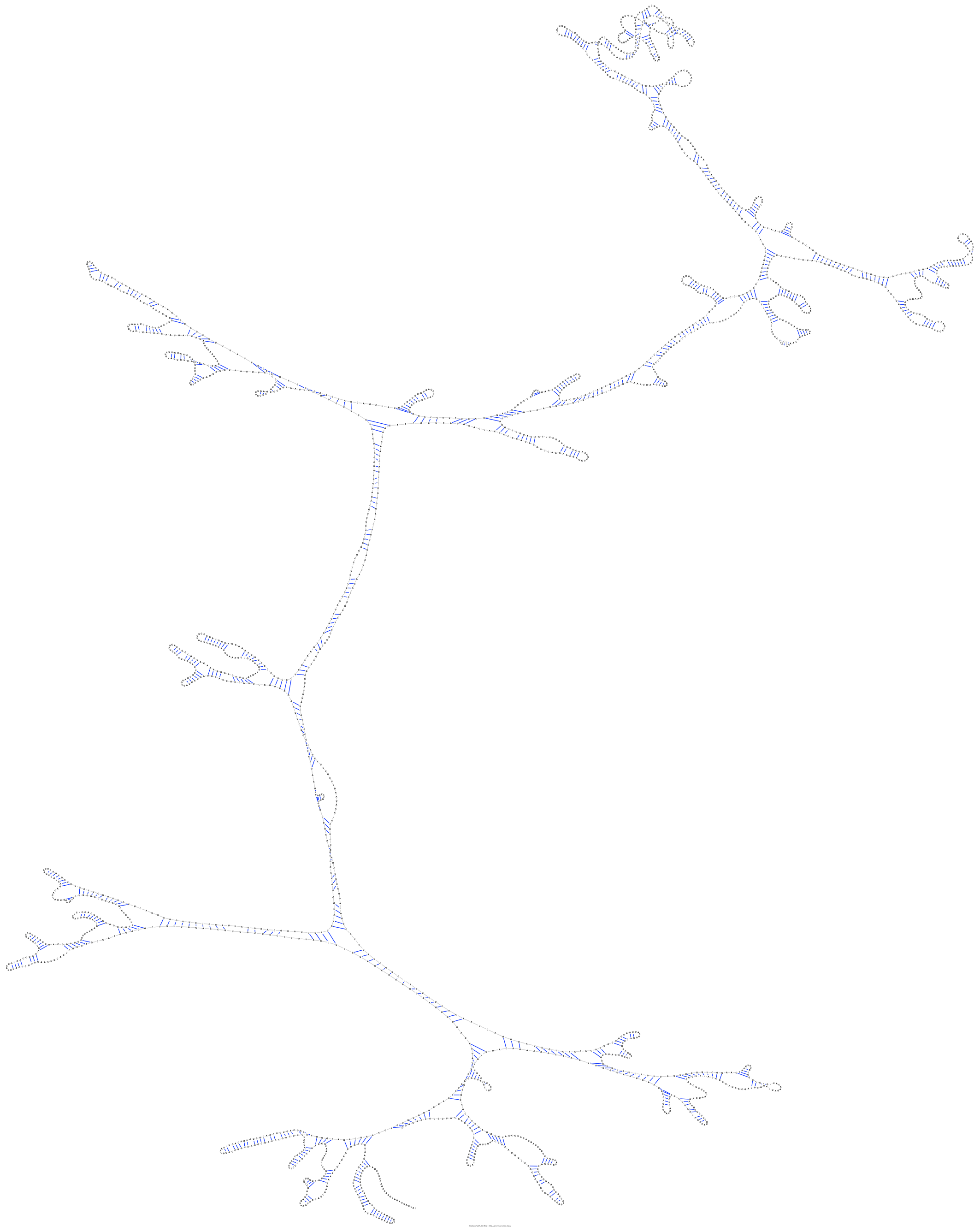


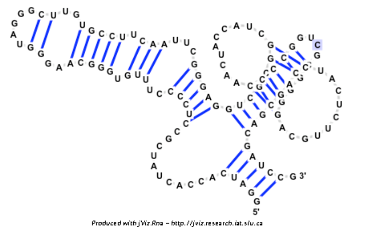
Figure 4.15: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using the original algorithm employed in jViz.RNA

Figures 4.16 - 4.19 show the same sequences presented above. However, these images were produced by jViz.RNA as it implements the optimization mentioned in Section 4.2. The sequences presented here are, again, only a subset chosen to illustrate the effect parameter optimization had on the sequences' layouts. The remaining eight sequences of this set can be found in Appendix A, Figures A.9 - A.16. Examining the structures, it is fairly visible that the majority of each structure is laid out fairly well, in a way that accentuates structural elements within the sequence. However, some structural elements become entangled or begin to overlap each other. Most notably it seems to occur with small structural elements that are only connected to the sequence by one stem.

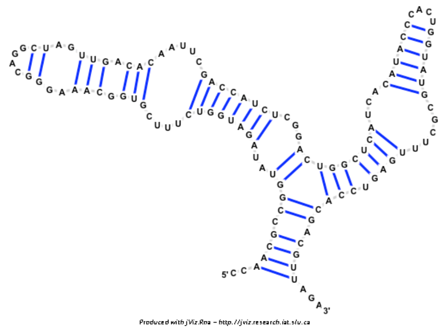
In order to hypothesize why such is that case, it's beneficial to remember what the optimization described entails. The original algorithm used an increasingly larger ”neighborhood” to explore more and more nucleotides at each iteration of the stabilization. The theory being that when the sequence is laid out in a circle, some nucleotides are so far away their effect need not be taken into account, and as they get closer, they are included in the ”neighborhood” of other nucleotides. The optimization decreased the rate at which this ”neighborhood” grows and thus allowed for the algorithm to run at a time closer to order $O(n)$ longer. This tradeoff between speed and accuracy could be what created the observed effect on isolated elements. The nucleotides in those parts of the sequence may have experienced a repulsion force that was much lower than the original algorithm's version. This smaller force allowed the force exerted by the spring-like bonds to collapse the nucleotide groups back into the sequence, or twist them.

This introduced distortion does not seem to vary between larger or smaller sequences, implying that particular structural elements, rather than structures of certain lengths, are vulnerable to it. In [27] it was mentioned that different values can be used as multipliers for the optimization procedure. Smaller values give smaller returns in improvements of run time, but they would also minimize the distortion observed here. Also, even with more entanglements present, the user can still manually resolve the distortions and end up with a clear structure layout. This structure layout would still take less time to produce than the original algorithm considering the dramatic improvements seen in Figure 4.7 for large structures of over 1,500nt.

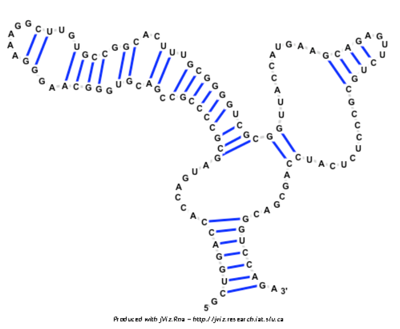
In general, the approximation based optimization introduced here definitely shows some shortcomings in the layout of the sequence, but that is to be expected from approximation based methods. Furthermore, these shortcomings are very much resolvable by users in a way that does not significantly diminish the advantage in run time this optimization method holds.



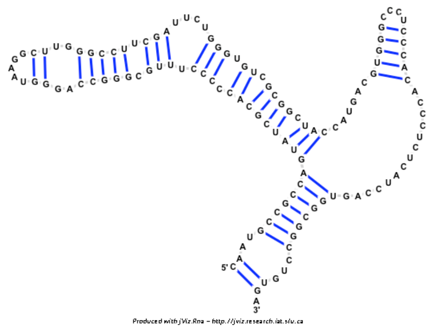
(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)



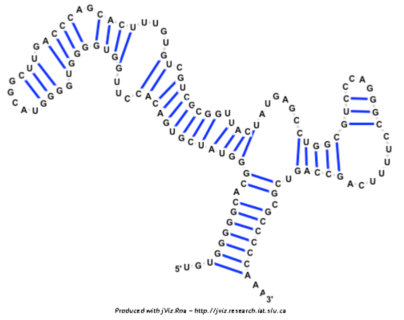
(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)



(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)



(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)



(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.16: The resulting sequence layout for the shortest five sequences using the parameter optimization method

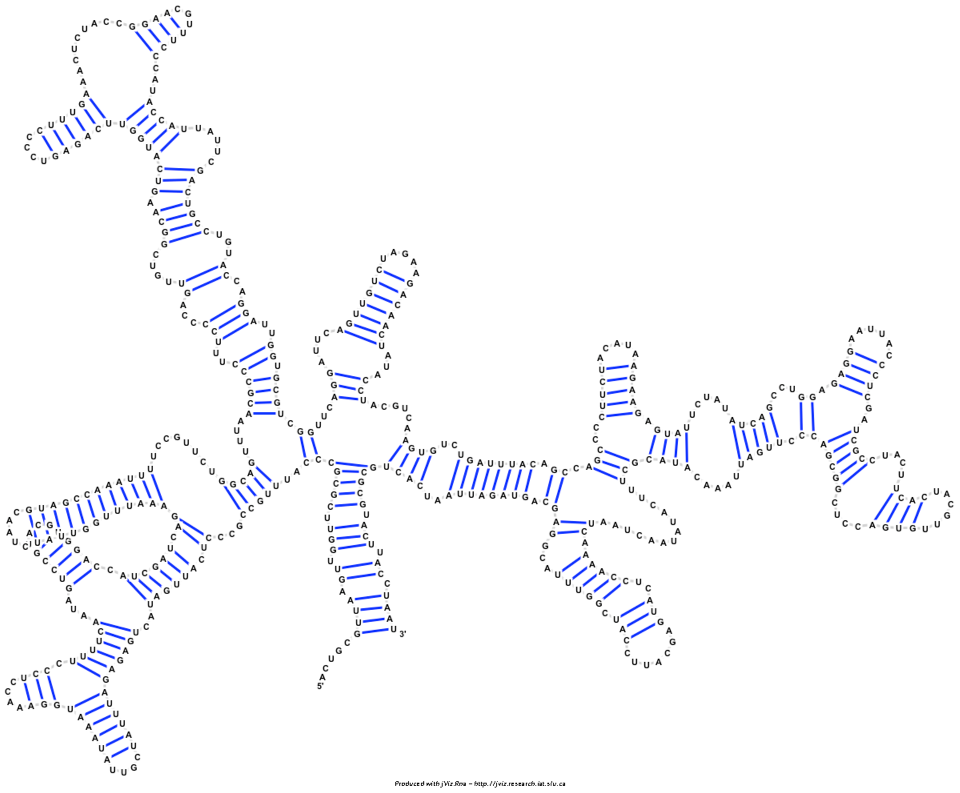


Figure 4.17: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using the parameter optimization method

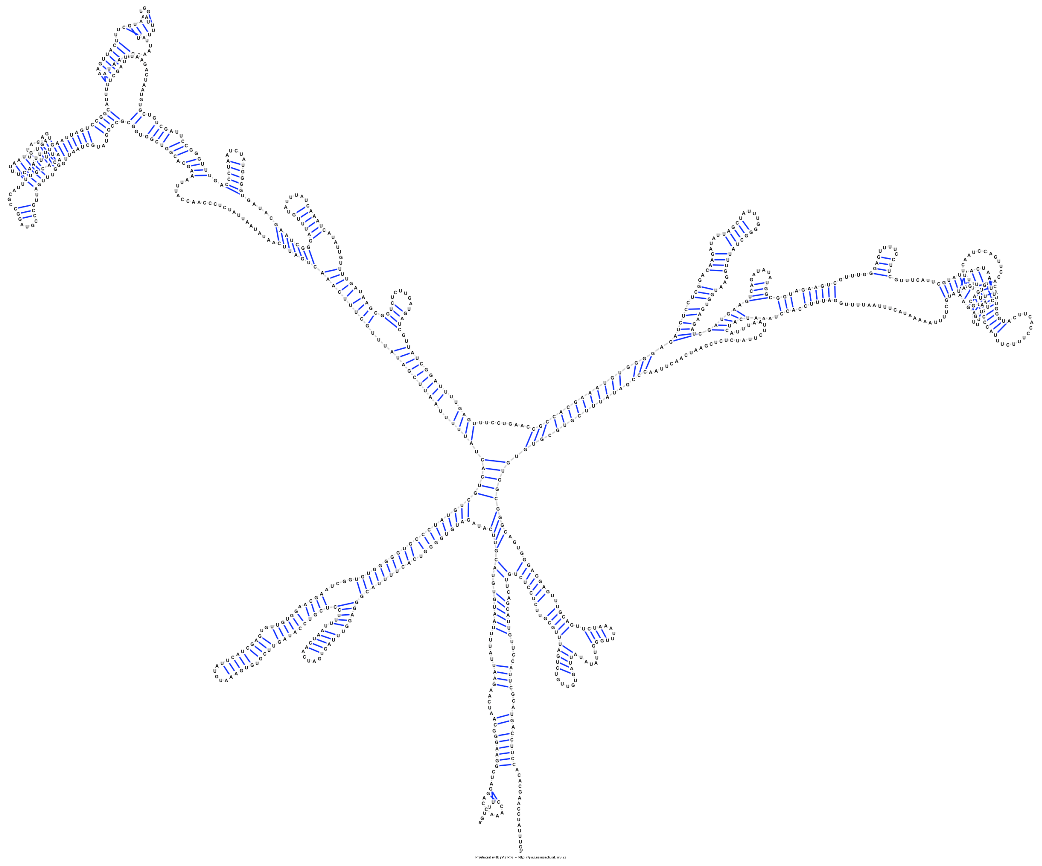


Figure 4.18: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using the parameter optimization method

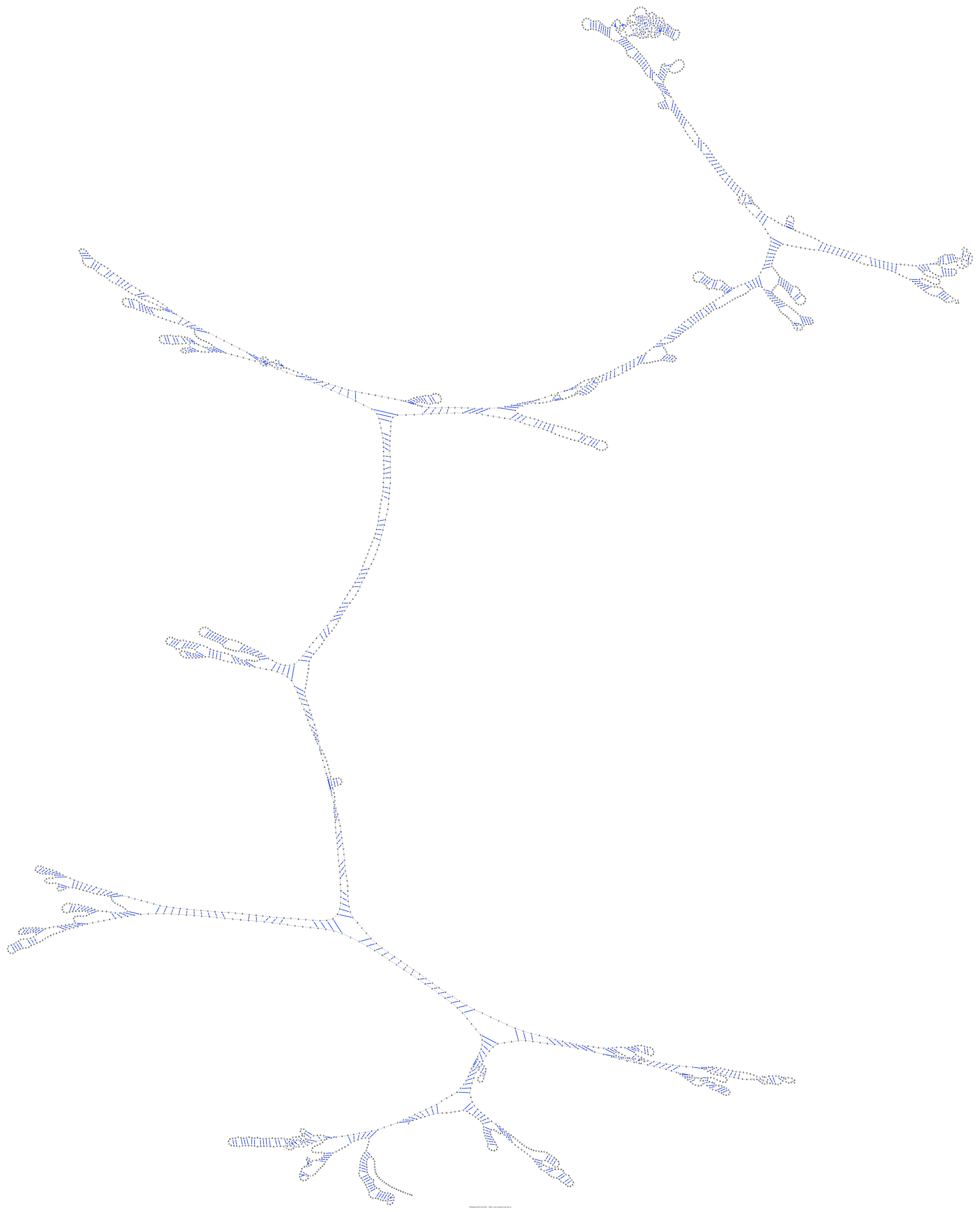


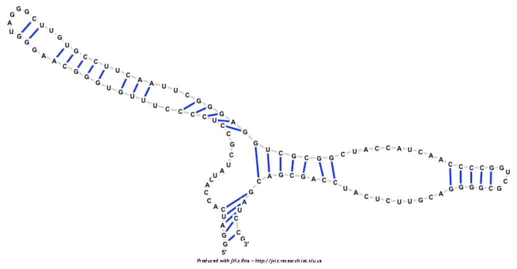
Figure 4.19: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using the parameter optimization method

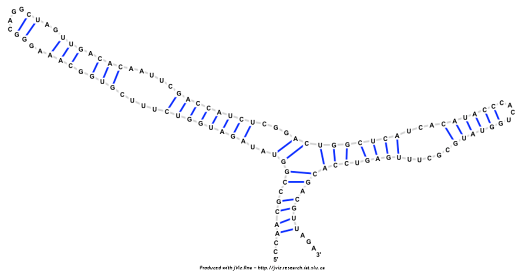
The next set of images (Figures 4.20 - 4.27) depicts the sequence layouts resulting from the use of the Barnes-Hut algorithm. Figures 4.20 - 4.23 show the results obtained by using a θ value of 1.0, while Figures 4.24 - 4.27 show the results obtained by using a θ value of 5.0. Similarly to the previous set of images, only several images of sequences' layouts were used here. The remaining eight sequences layouts for the Barnes-Hut Algorithm where $\theta = 1.0$ can be found in Appendix A, Figures A.17 - A.24, while the images for the sequences where $\theta = 5.0$ can be found in Appendix A, Figures A.25 - A.32. Inspecting Figure 4.20 one can see that with the implementation of the Barnes-Hut algorithm, the sequences experience a slight distortion from the layout presented in Figure 4.12 (using the original algorithm). The sequence elements are still fairly recognizable, but the structures seem more stretched out outward. However, Figure 4.21 displays a more serious issue; at the top right side of the image one can spot what seems to be a very thin string of nucleotides. Comparison to Figure 4.13 implies that this thin string of nucleotides is actually supposed to be a long stem with several loops through it. This problem persists through the other sequences showing long thin strings of nucleotides where long stems should be. This problem seems to become even worse when inspecting Figures 4.24 - 4.27 where even the short structures presented in Figure 4.24 have their stems stretched and thinned out. This issue is inherent to the use of the Barnes-Hut algorithm and in order to decide if the integration of this calculation method is appropriate, it is important to locate the source of this phenomena.

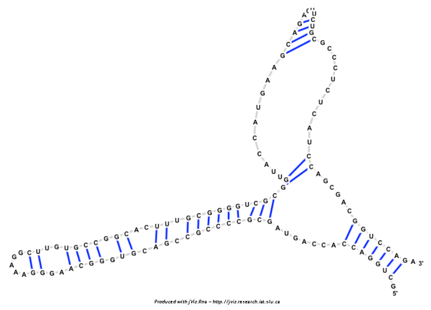
Again, one must re-inspect the theory behind the Barnes-Hut algorithm in order to hypothesize a source for this anomaly. The Barnes-Hut algorithm works by clustering bodies together into virtual bodies, it gives the virtual body the mass of all other bodies it was composed of, but it also places it at location which is the average of the real bodies' location. This set up works well with bodies that are very far apart from a single point and very close together to each other, but as the distance to closeness ratio decreases, an unwanted effect comes into play. A group of bodies that were originally spread beside another body (which we will denote B) will now be collapsed into a virtual body which resides in only one location. That force generated from this virtual body would be in a single direction and would push body B in that direction alone. Now, assuming a stem of nucleotides, the stem may be collapsed into a single body that would push the nucleotide at the edge of the stem very far away. That nucleotide would then pull the rest of the stem since all nucleotides are linked via the bonds in the RNA backbone. This set up could result in a single nucleotide, or a group of nucleotides, ”stretching” a stem to become thinner. This logic would imply that the effect would become more profound as the value of θ (symbolizing the ratio between the distance to a group of bodies and their closeness to each other) increases, and indeed such is the case. The structure layouts seen in Figures 4.24 - 4.27 display this problem much more than the set of layouts seen in Figures 4.20 - 4.23.

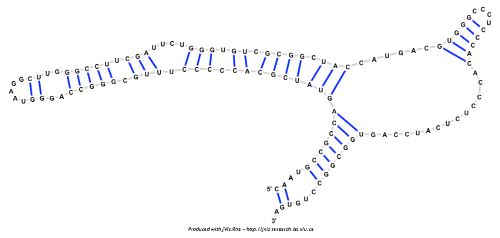
So far, the cause behind the distortion seen when employing the Barnes-Hut algorithm was established. Nevertheless, it is important to consider if the results seen after using the Barnes-Hut algorithm can be modified by users to give more aesthetically pleasing layouts. Unfortunately, logic would suggest that such is not the case. The previous anomalies seen could have been fixed since the algorithms underestimated the force experienced by the nucleotides, so the user could have intervened and correct for the forces the algorithm did not account for. However in this case, the algorithm causes an overestimation of the force generated on a particular nucleotide in a particular direction. In this case, even if the user were to intervene and drag the nucleotides that have been pushed away too far, the algorithm would attempt to compensate for what seems as a greater repulsion force than there actually should be. In other words, a structure produced by using the Barnes-Hut algorithm would require excessive user interactions to be corrected to a layout that is more aesthetically pleasing. One can argue that using a smaller value of θ could produce satisfactory results. Yet the graph in Figure 4.7 shows that using a θ value of 1.0 produced a run time that is very similar to the optimized original algorithm. Introducing a smaller θ value would most likely result in a run time that is greater than simply using the optimized original algorithm, negating the need for the Barnes-Hut algorithm to begin with.

In conclusion, in light of the resulting structure layouts seen from using the Barnes-Hut algorithm it is appropriate to say that despite the slight improvement it offers over the optimized algorithm insofar as run time goes, the distortion of the sequence layout it causes renders it inappropriate for this application.


(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)


(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)


(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)


(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)

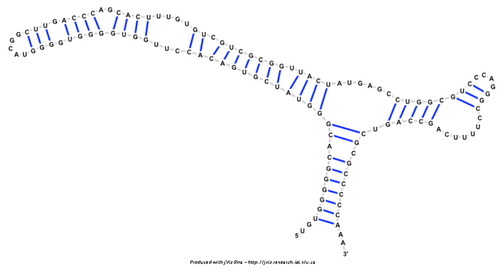

(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.20: The resulting sequence layout for the shortest five sequences using the Barnes-Hut algorithm ($\theta = 1.0$)

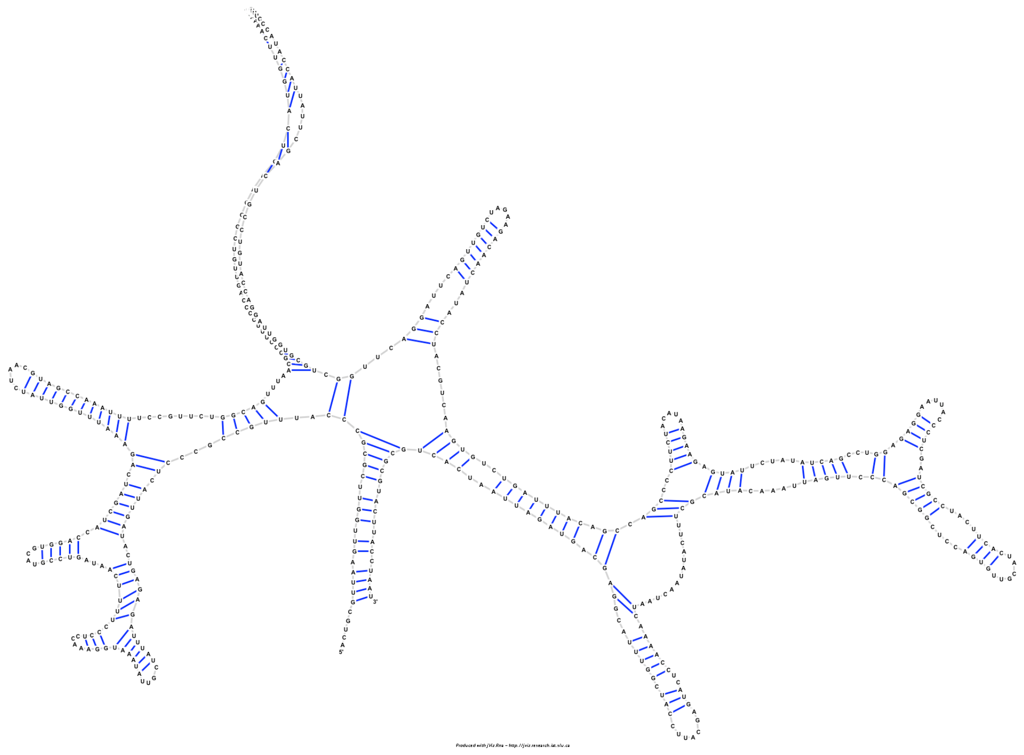


Figure 4.21: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using the Barnes-Hut algorithm ($\theta = 1.0$)

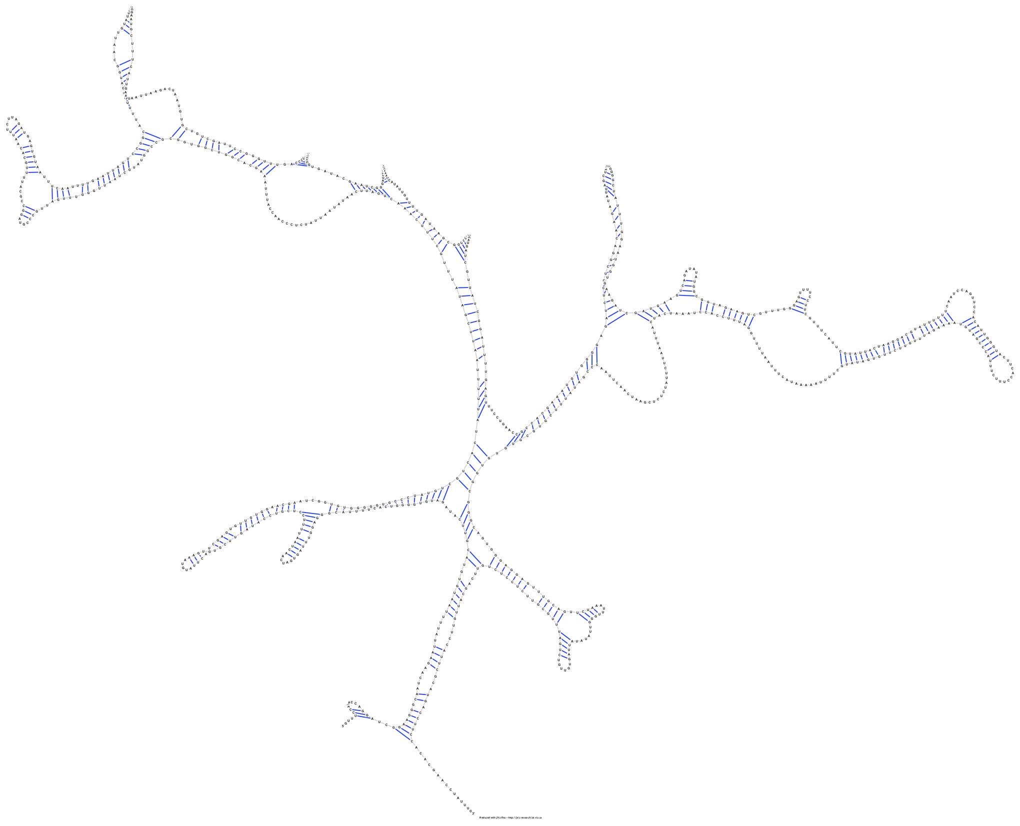


Figure 4.22: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using the Barnes-Hut algorithm ($\theta = 1.0$)

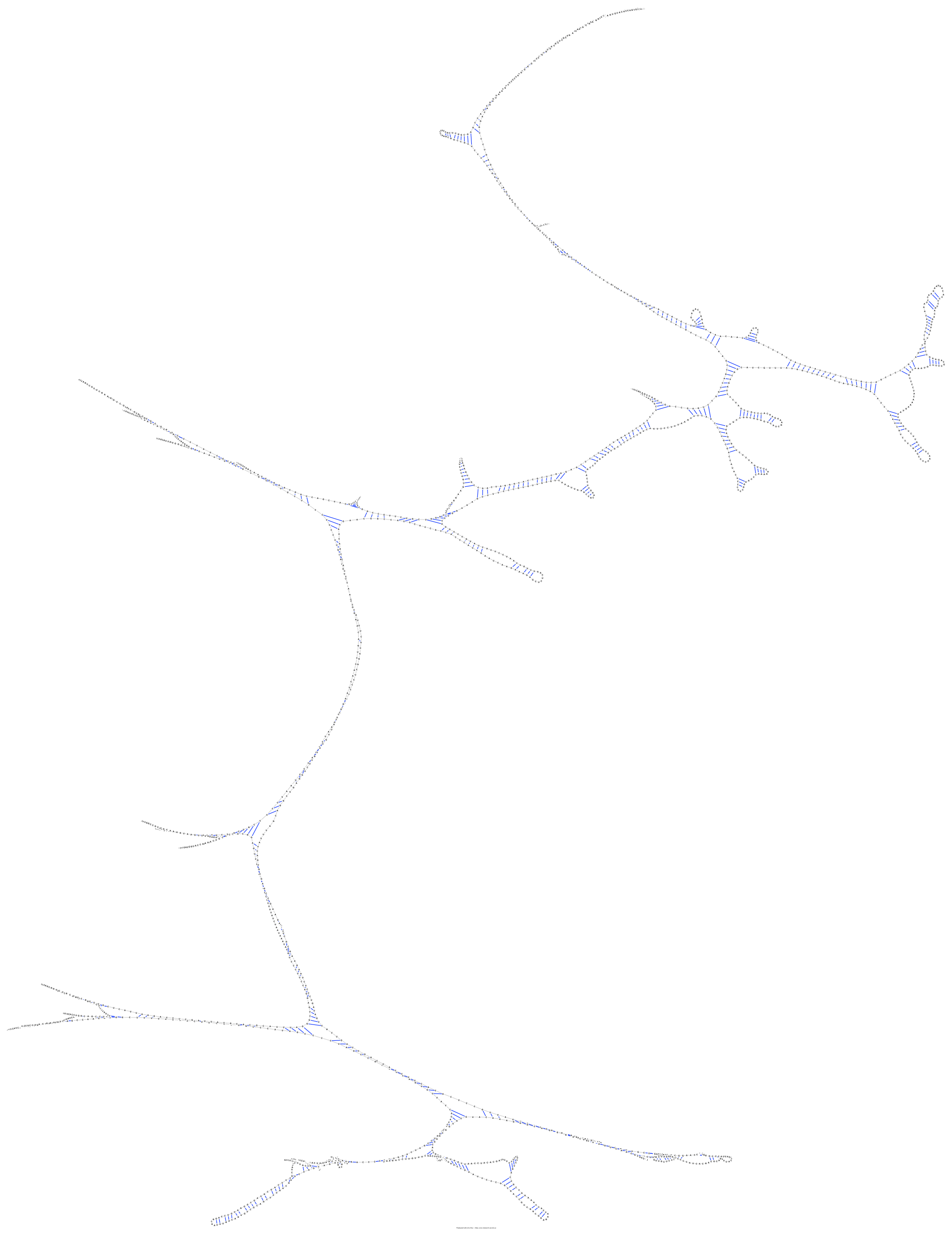
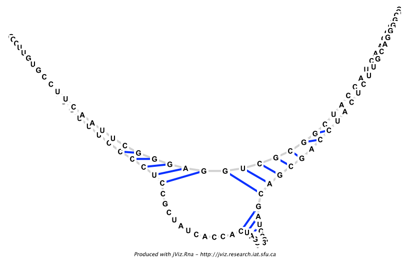
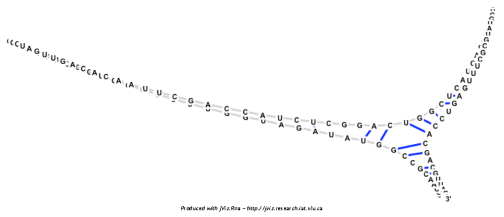


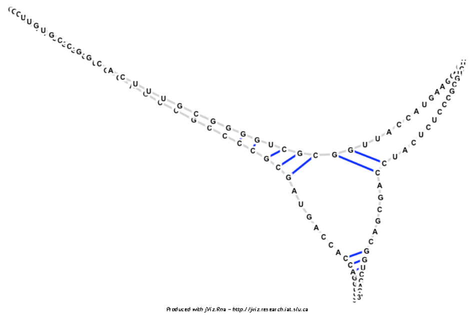
Figure 4.23: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using the Barnes-Hut algorithm ($\theta = 1.0$)



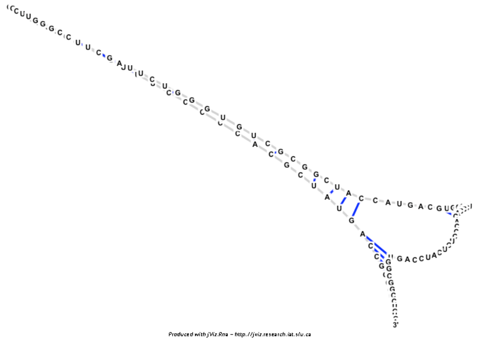
(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)



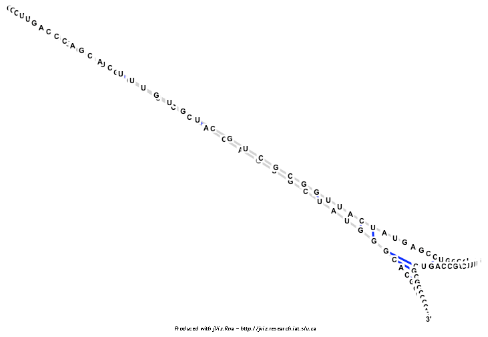
(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)



(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)



(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)



(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.24: The resulting sequence layout for the shortest five sequences using the Barnes-Hut algorithm ($\theta = 5.0$)

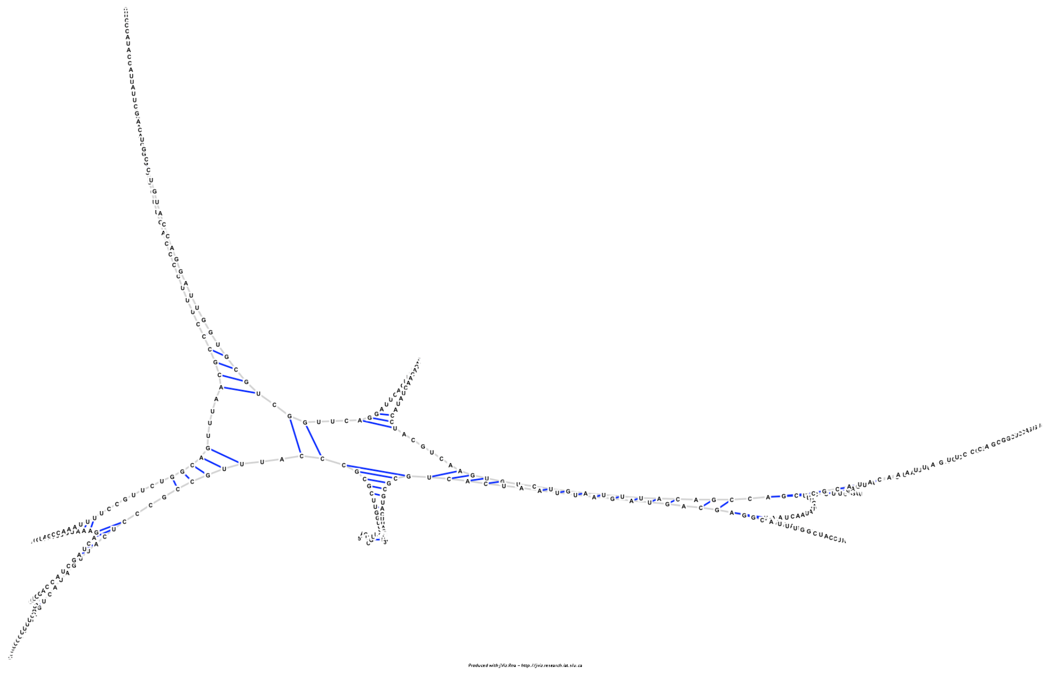


Figure 4.25: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using the Barnes-Hut algorithm (θ = 5.0)

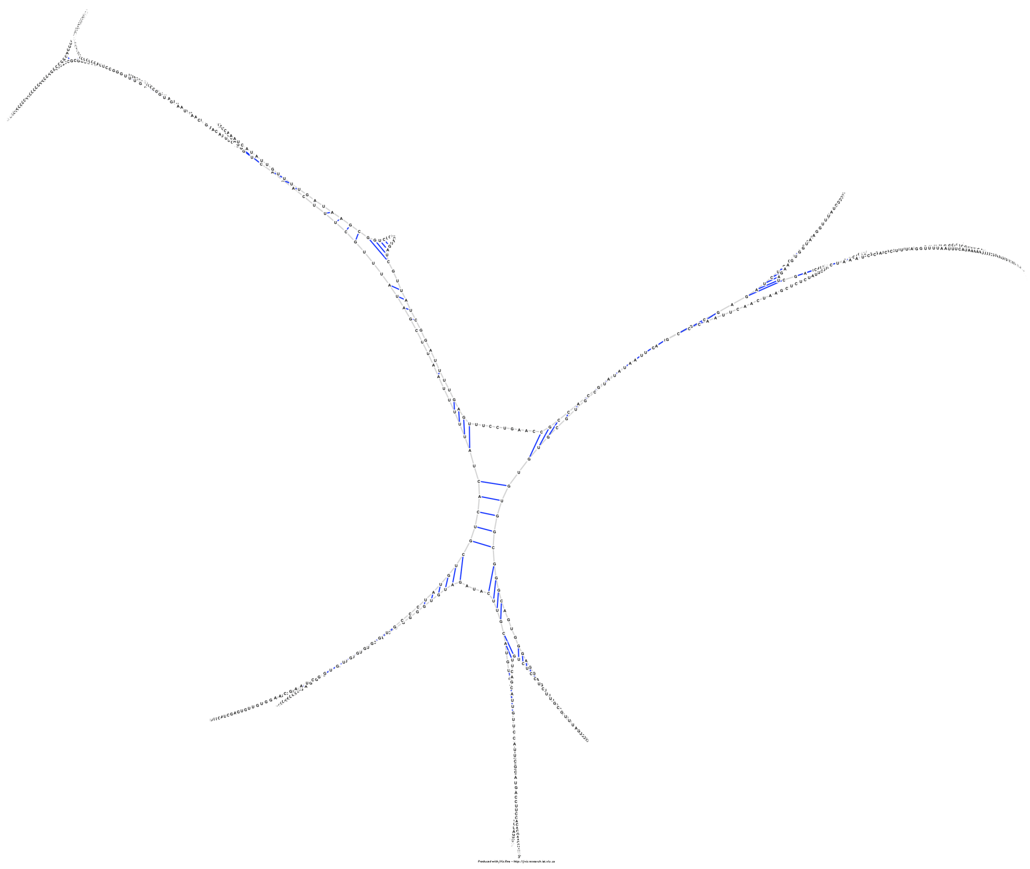


Figure 4.26: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using the Barnes-Hut algorithm ($\theta = 5.0$)

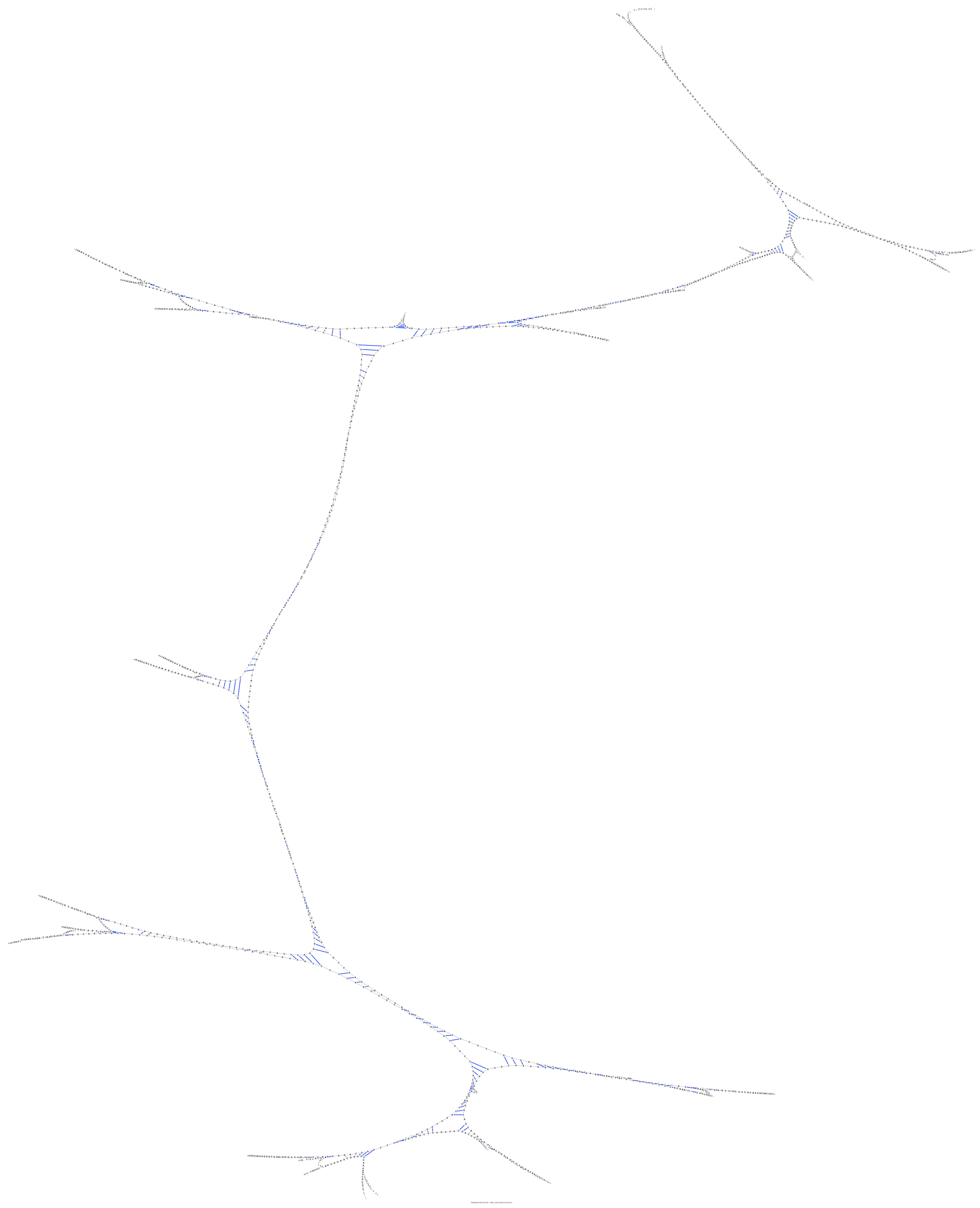


Figure 4.27: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using the Barnes-Hut algorithm ($\theta = 5.0$)

So far, only algorithm based methods were discussed. Although bringing an algorithm to maximum efficiency is important, it is by no means the only way to improve a software's run-time performance. The following two sections describe two other methods that were tested in order to increase the stabilization speed of structures simulated by jViz.RNA.

4.4 Multithreading

4.4.1 Implementation of Multithreading in jViz.RNA

With the rise of multiprocessor machines, the idea of multithreading offered an even larger improvement in speed. With a single processor, the jobs of different software components can be divided into two or more different threads such that when there is no need to process one job (e.g. the location of the mouse pointer), another job can be processed. With two or more processors, two or more jobs can be processed in parallel; meaning that a collection of jobs that should take T time, could take closer to $\frac{T}{2}$ time.
However, as often happens, the decrease in processing time comes with an increase in complexity. If the collection of jobs handled is completely unrelated to each other, then no complications arise. The jobs can be then completed in a random order by as many processors as available. Unfortunately, in this case, the goal was to split a single job (calculating the interaction of n bodies with each other) into several threads. This implies that all the interactions must first be calculated, and only then can the nodes be moved. In that sense, there must be some central "overlord" that sees to that all threads involved in calculating the interactions of the bodies have finished running.

In jViz,RNA, this was implemented by a `Searcher` class which implements Java's `Runnable` interface [22]. The `Runnable` interface allows classes that implement it to use separate threads, Conditioned on the fact that they implement the method `public void run()`. When the method `public void run()` is called, a new thread is spawned and everything inside that method runs separate from the main thread.

The `Searcher` class was defined as follows:

```
public class Searcher implements Runnable{

    private Thread myThreat;
    private int searcherID;
    static private int searcherNumber;
    static private int nodeNumber;
    static private double[] xSet;
    static private double[] ySet;
    static private int[] repulsionSet;
    static private double[] dxSet;
    static private double[] dySet;
    static private double rigidity;
    static private int[] finishedThreads;
    private static final double K = 10.0;
    private static final double radius = 360000.0;


    /*******************************************************************************************/
    /*****************************         METHOD DEFINITIONS        ************************/
    /*******************************************************************************************/

    public Searcher(int id)
    {
      // Initialization code...
    }//end constructor

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

    public static void setParameters(int theSearcherNumber,
                                int theNodeNumber,
                                double[] theXSet,
                                double[] theYSet,
                                int[] theRepulsionSet,
                                double[] theDXSet,
                                double[] theDYSet,
                                double theRigidity,
                                int[] theFinishedThreads)
    {
    // Set the pointers of the parameters...
    }//end setParameters

    /*******************************************************************************************/
    public void start()
    {
        // Actions to perform when this object's thread starts running...
    }//end start()

    /*******************************************************************************************/
    public void stop()
    {
     // Stop the current thread...
    }//end start()
    /*******************************************************************************************/
    public void run()
    {
         // Run the current thread...
    }//end run
    /*******************************************************************************************/
}//end class
```

The first attribute `private Thread myThread` defines the thread that this object is going to use. In the method `public void start()` this thread must be initialized as follows:

```
    public void start()
    {
         myThreat = new Thread(this, "Searcher"+searcherID);
```

```
        myThreat.start();
    }//end start()
```

The `searcherID` variable is an integer used to identify this particular `Searcher` object. The `searcherNumber` attribute denotes the total number of `Searcher`s that exist at any iteration.
`searcherID` will always be an integer between 0 and `searcherNumber`. Next, the set of attributes `nodeNumber`, `xSet`, `ySet`, `repulsionSet`, `dxSet` and `dySet` are all attributes describing the bodies in the simulation. There are `nodeNumber` bodies whose *x* and *y* coordinates are stored in `xSet` and `ySet`, and their repulsion constants (used in repulsion force calculation) are stored in `repulsionSet`. `dxSet` and `dySet` are arrays that will carry the displacement values for each body after the force calculation is done, and these arrays will be available to jViz.RNA when the `Searcher` has finished its job.
The attribute `static private int[] finishedThreads` is the most interesting part of this class. As mentioned before, whenever multithreading is involved, there is always the possibility of data corruption where one thread tries to read data that has not yet been written by the appropriate thread. In this case, it's important to make sure that all nodes had their displacements calculated before applying this displacement to the position, and that is what this variable does. First off, the fact that it is `static` means that all instances of `Searcher` carry a reference to the same memory location denoted by this variable. In other words, only one copy of the variable `finishedThreads` exists at all times and all `Searcher` objects have equal access to it. Second, it is defined as an `int[]` rather than `int`. In essence this means that what is defined here is a pointer to an `int`. This statement would be equivalent to defining an `int*` in C/C++. However, since Java does not allow the explicit definition of pointers, the array notion is used here. In practice however, the two are interchangeable.
So far, this means all `Searcher` objects will have access to the same pointer to an `int` (or an array of `int`s). In the `public static void setParameters()` method, when all the array pointers are set, `finishedThreads` is set to point to an array with one `int` element in it. This element would get incremented every time a `Searcher` finished its assigned work. With this set up in place, it is possible to then process the collection of bodies in the simulation in the following manner:

1. Declare `searcherNumber` `Searcher` objects.

2. Group the data of all the bodies in the current iteration to a collection of three arrays: the *x* and *y* coordinates and repulsion constants

3. Allocate two arrays for the *x* and *y* displacement coordinates.

4. Allocate a single element `int` array and set the value of its only element to 0.

5. Assign each `Searcher` object to every `searcherNumber`-th element of the array. i.e. if there are 3 `Searchers` then `Searcher`0 calculates the forces on body0, body3, body6,. ..., `Searcher`1 calculates the forces on body1, body4, body7, ..., and `Searcher`2 calculates the forces on body2, body5, body8,....

    - When a `Searcher` finishes calculating the forces over all the bodies it was assigned to, have it increment the value stored in the single element array `finishedThreads` by 1.

6. While the number stored in `finishedThreads` is smaller than the number of total `Searcher` objects, suspend all other activity relating to the bodies in the simulation (UI and other Graphics actions are not suspended).

It is important to notice that step 5 and step 6 do not occur sequentially. Since each `Searcher` runs its own thread, instruction 6 is performed before instruction 5 has completed. That is, in fact, why there is a need for instruction 6 to suspend all other calculation activity until instruction 5 has finished. One might argue that if the block of code would be put inside a `synchronized` block, ”locking” all arrays that describe the nodes until the code is done, it would be a much more elegant solution. However, this would also prevent multiple `Searcher`s from accessing the arrays at the same time, which would be counterproductive.

In theory, the more `Searcher`s one uses, the faster the process will be. However, anyone experienced enough in multithreading would have an intuitive feeling that such is not the case. First, it is clear that there is no point in declaring more `Searcher`s than the number of bodies, i.e. if one is simulating the *Arthrobacter globiformis* 5S ribosomal RNA (M16173) which contains 122 bases, there is no point in declaring more than 122 threads. In addition, the number of threads ideally used greatly depends on the number of processors in the machine that is employed. For the purposes of this thesis, a MacBook Pro with two cores of 2.4GHz was used, and it is reasonable to assume that most labs that will employ jViz.RNA will not have more than 4 cores available to them per computer. As a result, one would expect the greatest improvements to occur when two threads are used instead of one, since then both cores can participate in the force calculation where each core is responsible for one thread. Addition of more than two threads might be beneficial or detrimental since more threads require more prioritization of the threads. The following section shows the result of multithreaded runs on all 16 sequences.

4.4.2 Multithreading Results

For these experiments, a varying number of threads (from one to ten) was used and 20 runs were made under each thread. The running times of these 20 times were averaged out and the averages (as well as 99% confidence intervals) were recorded. Again, the measurement was performed using the code:

```
ThreadMXBean bean = ManagementFactory.getThreadMXBean();
long startUserTimeNano = bean.getCurrentThreadUserTime();

//perform task ...

long taskUserTimeNano = -(startUserTimeNano-bean.getCurrentThreadUserTime());
jVizCore.CPUnanosecondsElapsed += (taskUserTimeNano/1000000.0);
```

Figure 4.28 shows the running times in milliseconds as different numbers of `Searcher` objects are used from one to ten.

It can be difficult to make out a trend for all sequences displayed here. Indeed, the size of the sequence makes a great difference in the effect adding additional threads has. It is best to group the sequences to groups based on their size and then look for trends within these groups.

Looking first at the largest six sequences, the trend becomes rather apparent (Figure 4.29). The addition of a second thread greatly reduces the computation time required to calculate a steady layout for a sequence. The time drop is almost by 50%, which is not a surprising number considering that the work is now divided between two threads rather than one. The addition of a 3rd and 4th thread also show a degree of improvement in run time. However, with the addition of a 5th thread, the improvement in run-time is very slight and is only seen clearly for the largest sequence (*Chlorella saccharophila* 18S ribosomal RNA (AB058310)). The addition of a 6th, 7th, 8th, 9th and 10th thread is almost indistinguishable from the use of only five threads, and the confidence intervals show that any results obtained from these runs are inconclusive regarding actual improvement in run time. This occurs since the use of additional threads also requires additional thread management, which could nullify any run time improvements the threads bring.

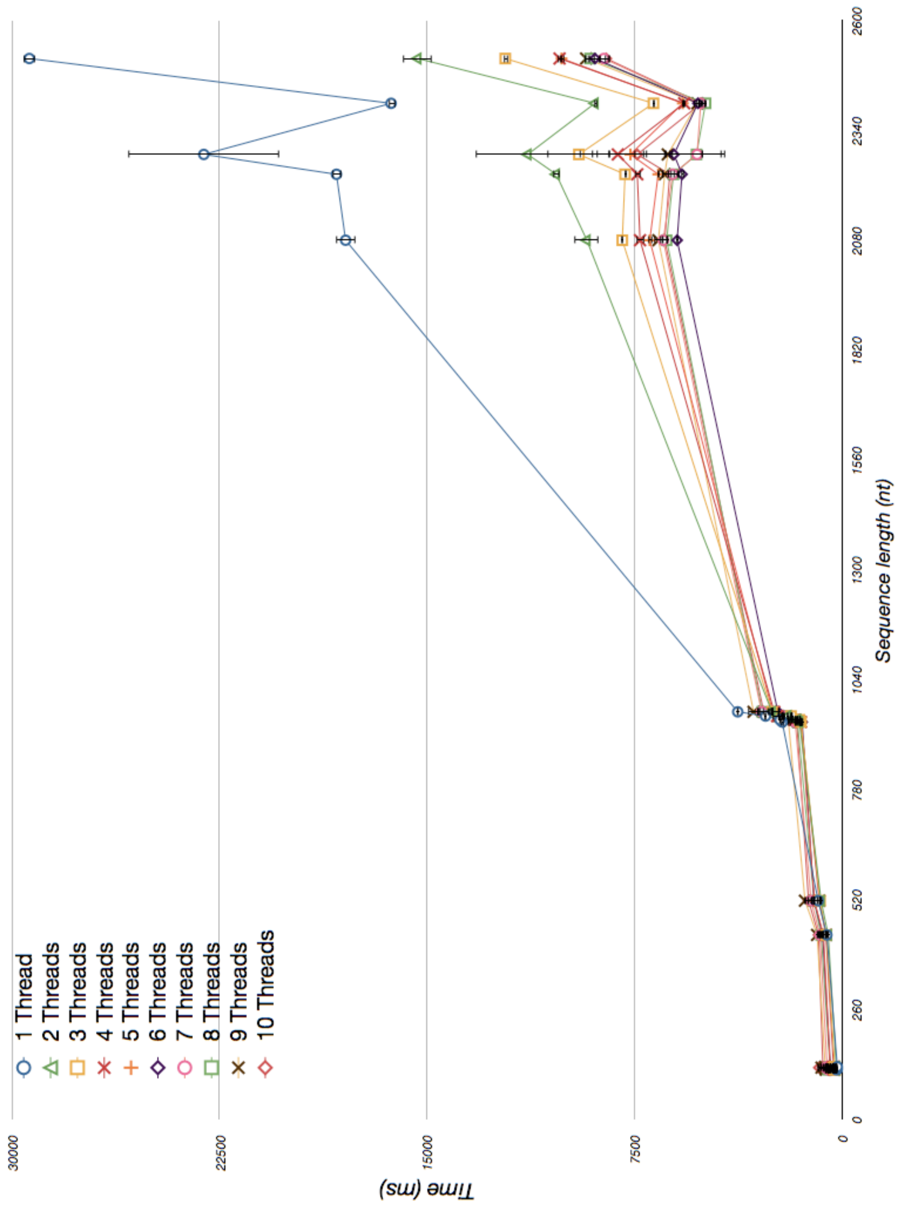


Figure 4.28: The stabilization times (in milliseconds) for the different RNA sequences as the number of threads used increases. 99% confidence intervals are used as error bars

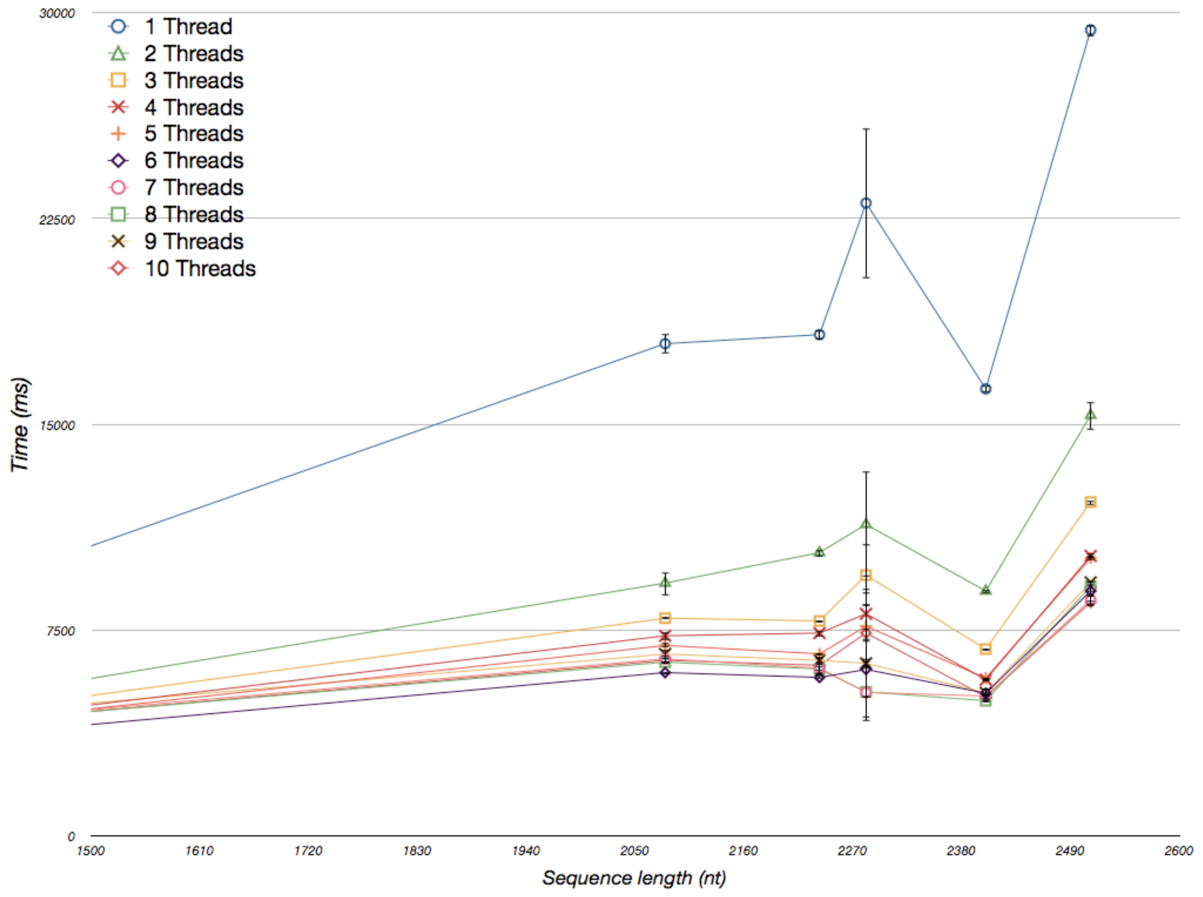


Figure 4.29: The stabilization times (in milliseconds) for *Chlorella saccharophila* 18S ribosomal RNA (AB058310), *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746), *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345), *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) and *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) as the number of threads used increases. 99% confidence intervals are used as error bars

The second group contains the sequences between 900nt and 1000nt and is depicted in Figure 4.30. One can see that the use of multiple threads is still superior to using only one thread. However, an interesting observation is that the use of nine threads seems to be more time consuming that the use of two to eight threads. Since these results are the averages of 20 runs they are not mere coincidence. This chart implies then that for these medium-large sized sequences, the use of an excessive number of threads doesn't simply fail to aid in processing time (as with the previous group of sequences depicted in Figure 4.29), but it actually serves to increase the run-time and impair the performance of jViz.RNA. Unlike the previous group of sequences, where one could argue that increasing the number of threads leads to some benefit, these results demonstrate that the use of six threads produces the fastest running time, and the addition of any additional threads simply causes an increase in performance time. This is, again, probably due to the introduction of the time required to manage threads as they are prioritized.

The third group of sequences are the RNA sequences whose size lies between 400nt and 600nt, the group of seqences depicted in Figure 4.31. An initial inspection of these sequences shows the curve describing the run-time for a single thread is not at the top of the graph. In the previous groups, using multithreading was always effective compared to using a single thread. However in this case, it is visible that using multithreading isn't necessary more effective than a single thread. Moreover, it seems the use of any number of threads past two (which splits the work to the additional processor on the machine these experiments were done) introduces an overhead from the management of threads. In this particular case, the overhead is so great (in relation to the regular run-time) that it becomes counter-productive to use more than two threads.

The final group of sequences includes five sequences whose size is between 100nt and 130nt. This group of sequences is depicted in Figure 4.32. This group displays two interesting trends. First it is easily visible that the curve depicting performance under one thread is at the bottom of the graph, implying that using a single thread is better than multithreading in this case. Furthermore, it is also visible that the increment in the number of threads corresponds to an increment in the run-time of jViz.RNA. It is evident from this graph shows that this group of small sequences stands out from the others. While for the longer sequences inspected before, multithreading bore some benefit, small sequences seem to only suffer from the addition of threads. The overhead created by thread management becomes larger than the return of using more than one thread for these sequences. Unlike the previous group of sequences (depicted in Figure 4.31), this group of sequences does not benefit even from the addition of a second thread, which divides the work between the two processors on the machine used for testing.

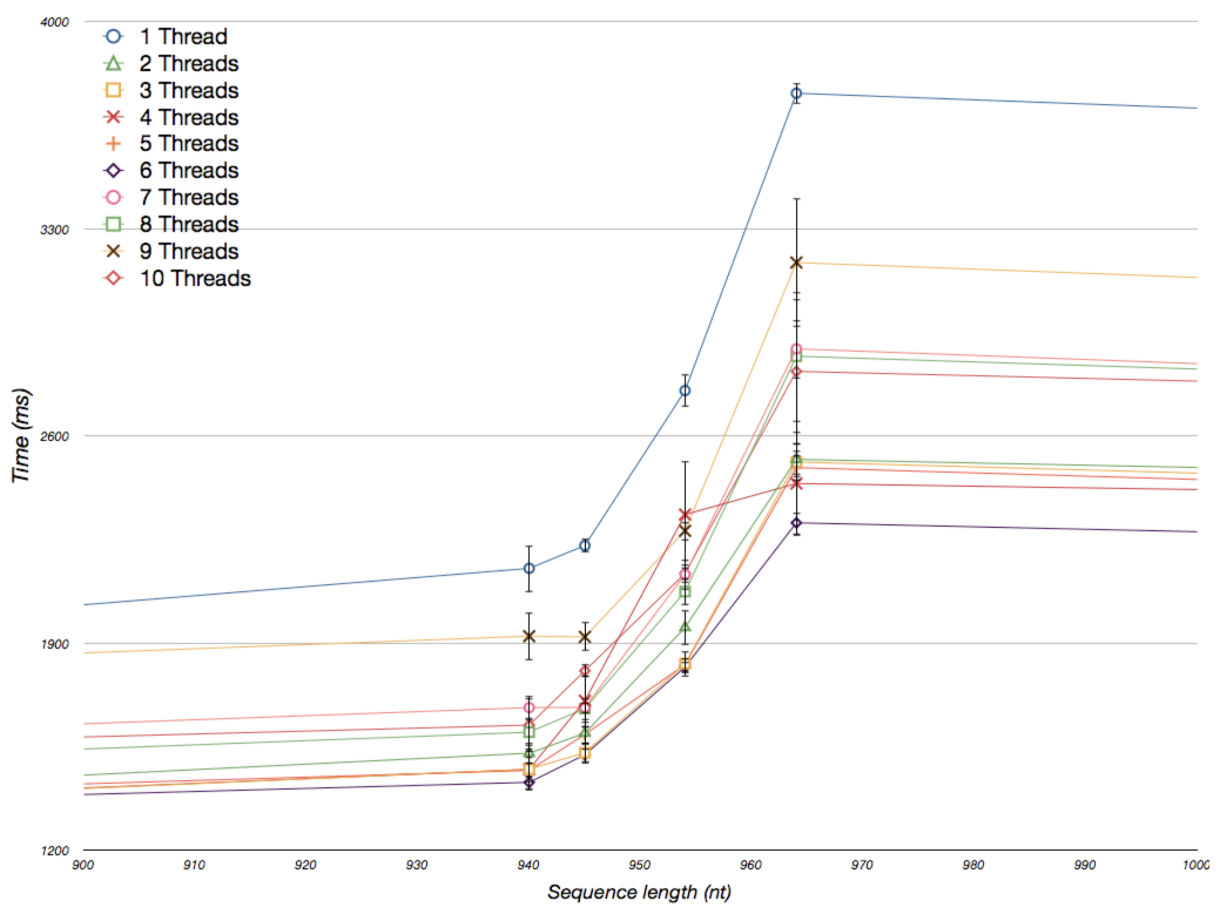


Figure 4.30: The stabilization times (in milliseconds) for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511), *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601), *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) and *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) as the number of threads used increases. 99% confidence intervals are used as error bars

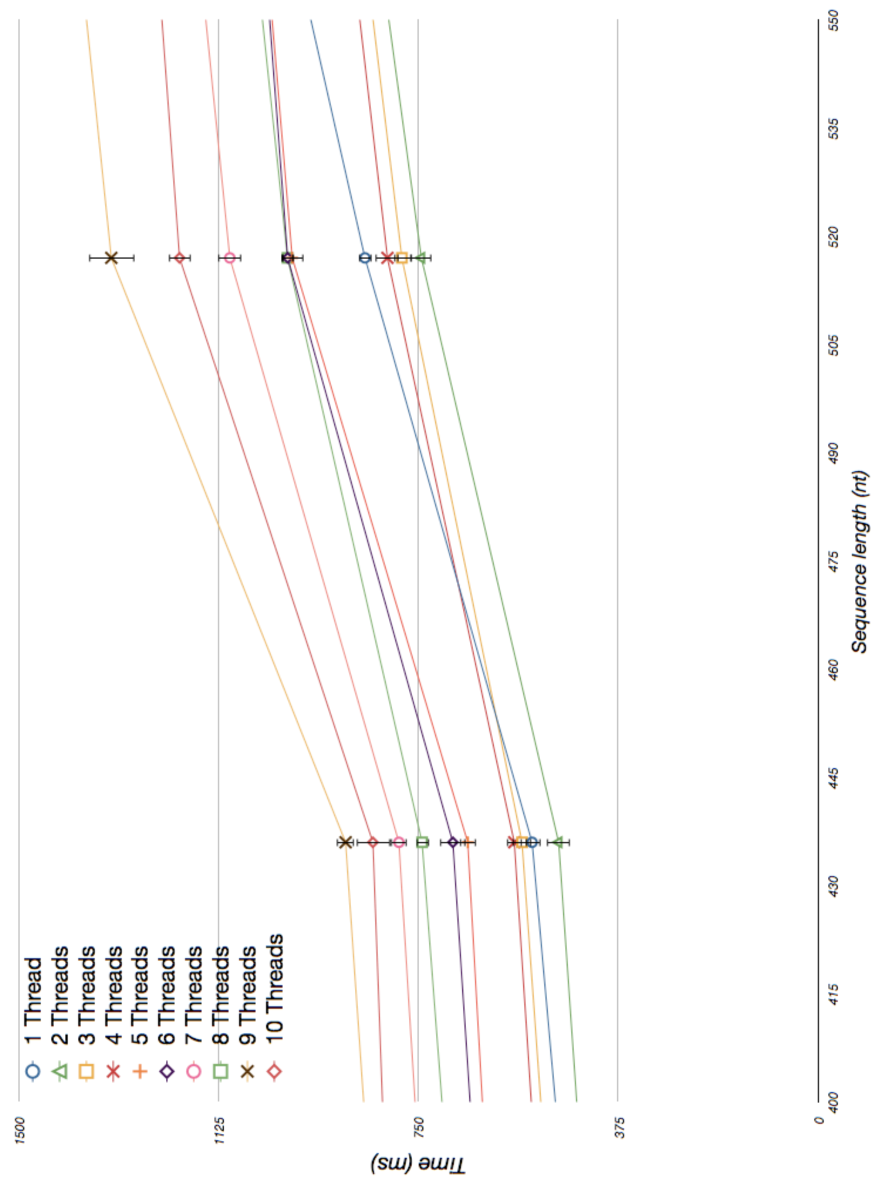


Figure 4.31: The stabilization times (in milliseconds) for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) and *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) as the number of threads used increases. 99% confidence intervals are used as error bars

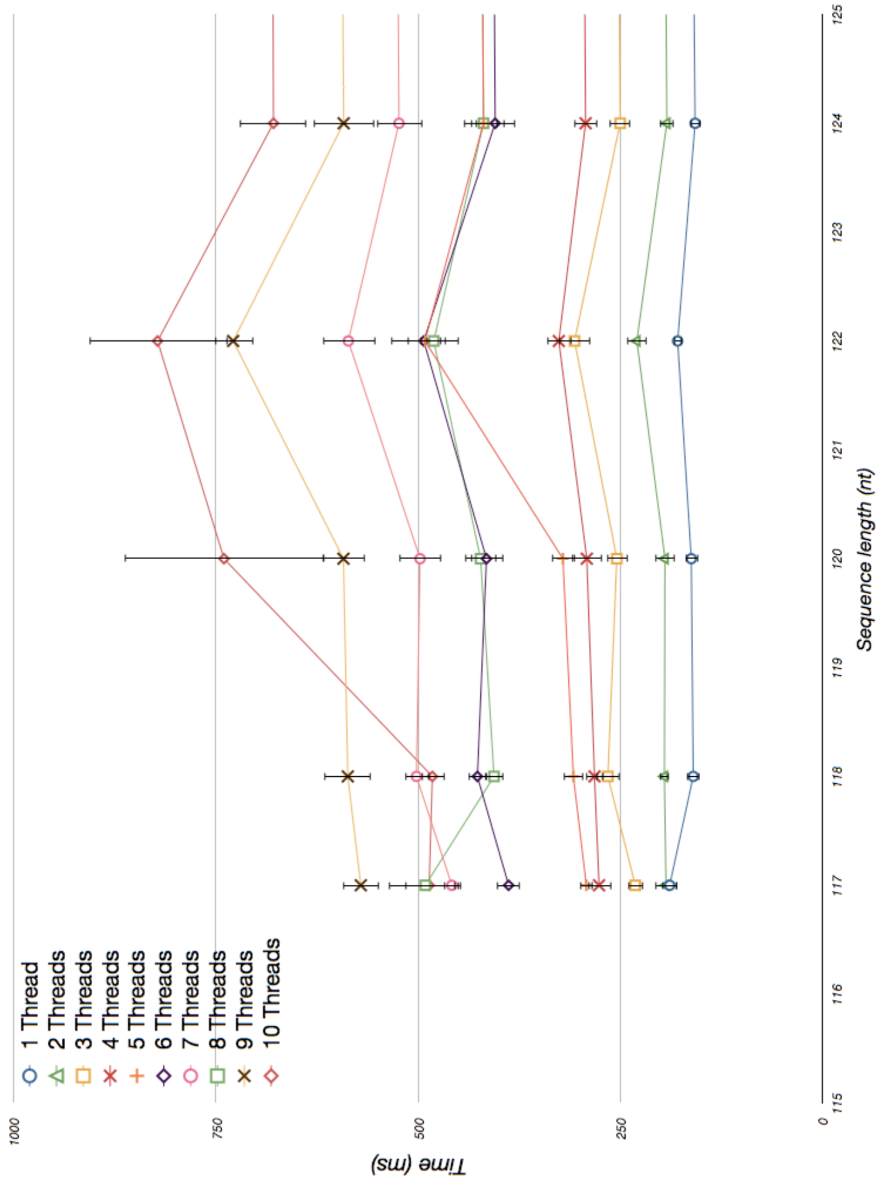


Figure 4.32: The stabilization times (in milliseconds) for *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515), *Arthrobacter globiformis* 5S ribosomal RNA (M16173), *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627), *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579) and *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080) as the number of threads used increases. 99% confidence intervals are used as error bars

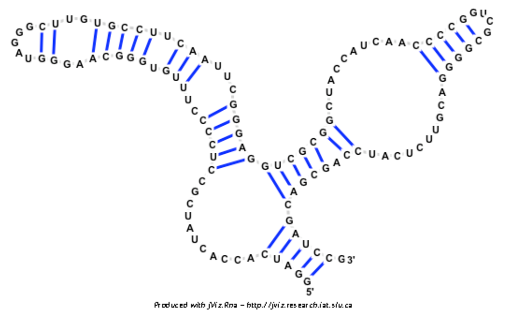
The results displayed here show a link between sequence size and any benefits that can be drawn from multithreading. Although it is hard to use these graphs to extrapolate an exact formula for the relationship between sequence size and ideal thread number, these results give some heuristic on how multithreading can be used effectively in jViz.RNA. It is evident that small sequences gain no benefit from multithreading, and that large sequences of about 2000nt benefit greatly from up to five threads. Also, it is evident that as the sequence size increases, more threads can be used to process it effectively. These results could be the basis to set a switch in jViz.RNA that determines the number of threads to be used as a function of the sequence length. Furthermore, these results are useful in the sense that they emulate the typical work environment that will be available for people who use jViz.RNA. In essence, the improvements seen here are the improvements one could expect when employing multithreading in jViz.RNA.

4.4.3 Effects on Structure Presentation

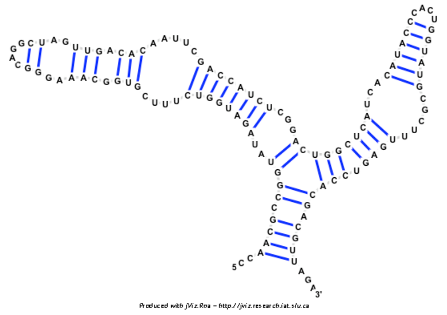
As with the previous section, it is important to inspect if introducing multithreading into the force calculation algorithm causes any unwanted effects on the structures' layouts. Figures 4.33 - 4.36 show the structure layouts resulting from using five threads. Since the multithreading implementation used here uses the simple $O(n^2)$ approach, without an increasing ”neighborhood”, one would expect the structure layout would be no worse than the one observed in Figures 4.12 - 4.15. Once again, the images used here are a sample selected to show the sequences' layouts. The layouts of the remaining eight sequences can be found in Appendix A, Figures A.33 - A.40. Inspecting the structures resulting from the use of multithreading, once can see that there are almost no entanglements of the structures, and no overlap is observed in all 16 structures. The small sections that are entangled in some sequences can easily be resolved by the user. In addition, structural elements such as loops and stems can be seen very clearly and lose no accentuation from the introduction of multiple threads.

The results observed here are consistent with what one might expect since no modification to the calculations is done, the calculations are merely dispersed between different threads and processors. Since the calculations that are dispersed are independent of each other, the completion of either one before the other is also irrelevant. It is also safe to assume that a more sophisticated implementation of multithreading, involving the ”neighborhood” used by jViz.RNA currently, would achieve the same results since independent calculations are going to be dispersed between the threads.

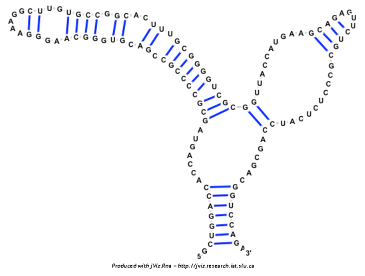
Conclusively, inspecting Figures 4.33 - 4.36 shows that the use of several threads to perform the calculations supplies a powerful run time improvement without diminishing the quality of structure layout.



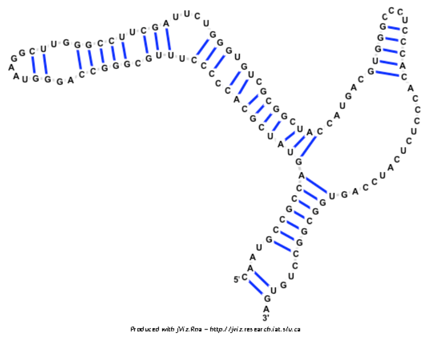
(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)



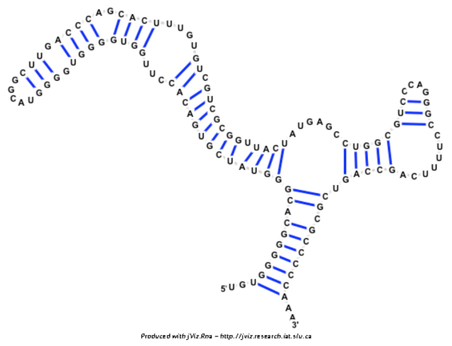
(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)



(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)



(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)



(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.33: The resulting sequence layout for the shortest five sequences using the multi-threading approach (thread number = 5)

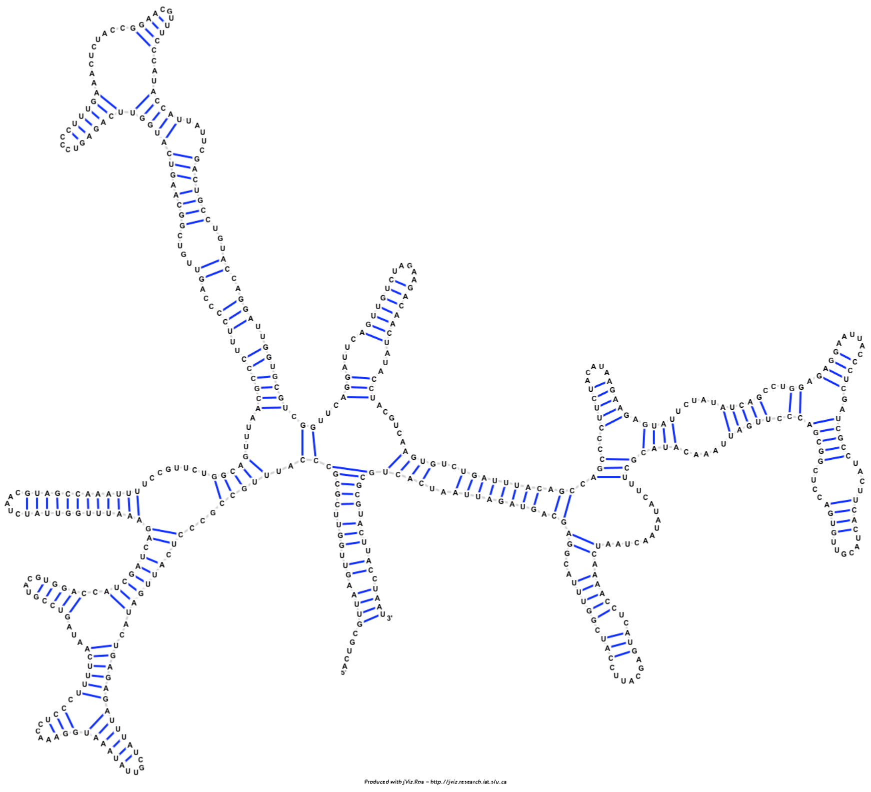


Figure 4.34: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using the multithreading approach (thread number = 5)

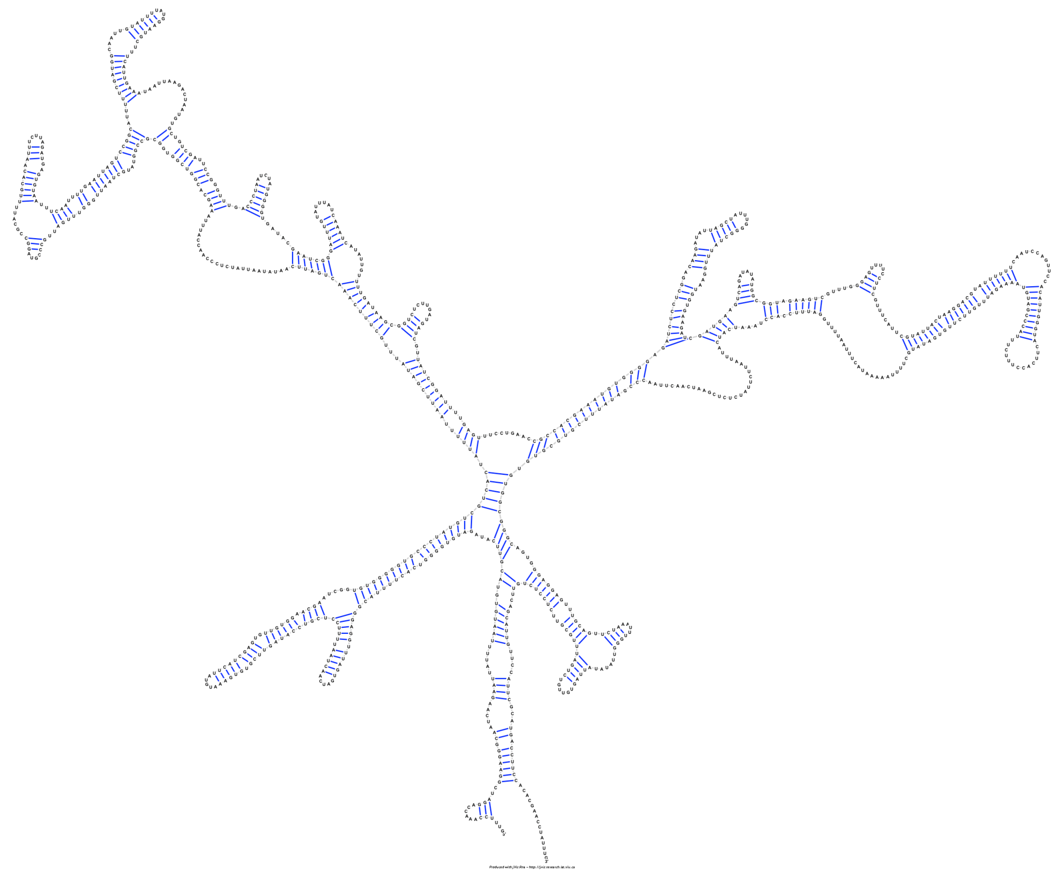


Figure 4.35: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using the multithreading approach (thread number = 5)

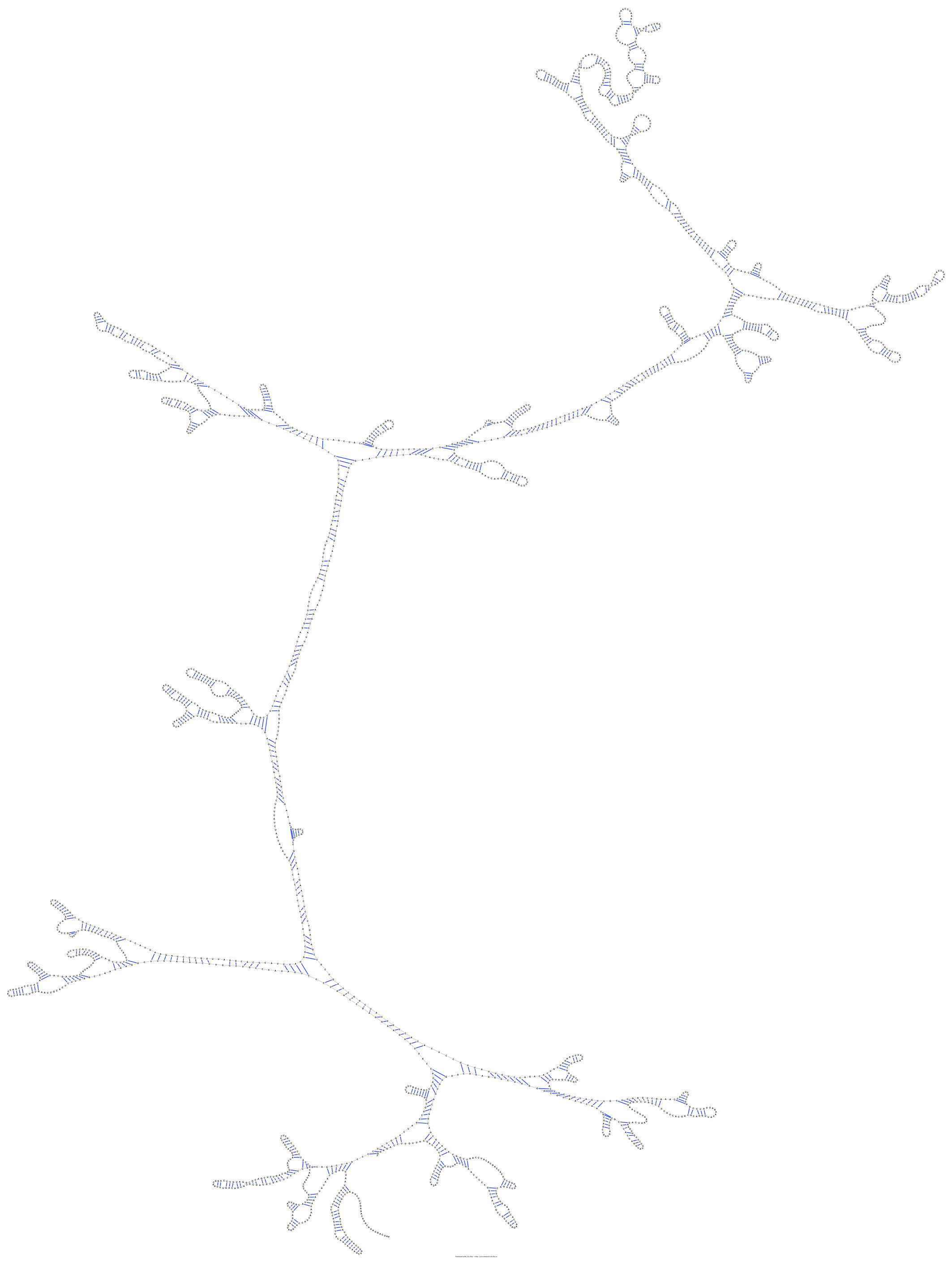


Figure 4.36: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using the multithreading approach (thread number = 5)

4.5 Native C Code

4.5.1 Linking C and Java Code and the Implementation of the Native C Code

Java is a widely used programming language mainly due to the fact it employes the Java Virtual Machine (JVM). There are different JVMs for each of the widely used operating systems such as Windows, MacOS and Linux. Each JVM takes a Java program and can execute it according to the operating system it is on. This makes programs written in Java independent of the operating systems they are run on, and so one program written in Java need not be readjusted, or recompiled, for every new operating system. One .jar file (the product of compiling a Java program) can run on all operating systems that have a JVM.

However, this also creates a drawback in terms of processing speed; as a result of this setup instructions written in the .jar file have to be executed by the JVM, where the JVM acts as a ”translator” between the operating system and the program. Much like conversations can run faster when two people speak the same language, rather than when there is a translator present, programs which create machine readable executables tend to work faster than programs that require an intermediate such as the JVM.

To answer the requirements for faster computation, Java offers the definition and implementation of ”Native Code” in C. Native C code can be invoked from a Java program by defining an appropriate method name in the Java code, and then implementing it with a dynamic C library. In order to define the appropriate method in Java the keyword `native` needs to be added to the method.

For example, defining a method as follows:

```
private native void doSomething();
```

declares a method named `doSomething()` that would be implemented in C.

Since the C files used to implement these methods are linked to the Java declaration, the headers for C libraries that implement native methods need to be produced automatically. The command `javah` is used, for example:

```
$ javah myClass
```

To produce a header file for a native method defined in `myClass.java`.

However, this course of action does not always result in an increase of speed. Creating the C objects that represent Java objects and copying their data takes up some portion of time. In other words, it would be pointless to implement a method such as:

```
private int sum(int a, int b)
```

```
{
     return a+b;
}
```

in C code, since the amount of time required to call the C method, create the correct data types and copy the values from Java would be more costly than the time required for the JVM to execute the instructions required to calculate the sum and return it.

Evidently, a software developer must take note of this fact and realize that defining C code for methods is only effective if the methods execute a large set of operations. Only large chunks of code would benefit from paying the price required to create the appropriate C object and assign the right values to them.

In this case, the algorithm required to calculate the forces on all nodes in a brute-force method was compared when implemented in Java and in C. Also, to reduce the time required to copy objects, only primitive[1] arrays were passed into the C routine.
The method definition on the Java end was:

```
private native void nativeAvoidLabels(double[] xSet,
                                      double[] ySet,
                                      int[] repulsionSet,
                                      double[] dxSet,
                                      double[] dySet,
                                      double rigidity)
```

To produce the appropriate C header file for the method defined in `TGLayout.java` the following command was used:

```
$ javah TGLayout
```

That command creates the file `TGLayout.h` which contained the definition for the following method:

```
JNIEXPORT void JNICALL Java_com_touchgraph_graphlayout_TGLayout_nativeAvoidLabels
(JNIEnv *theEnv,
 jobject theObejct,
 jdoubleArray jXSet,
 jdoubleArray jYSet,
 jintArray jRepulsionSet,
 jdoubleArray jDXSet,
 jdoubleArray jDYSet,
 jdouble rigidity)
```

[1]primitives being the atomic objects int, char, boolean, double and float

The parameters `jdoubleArray jXSet`, `jdoubleArray jYSet`, `jintArray jRepulsionSet`, `jdoubleArray jDXSet`, `jdoubleArray jDYSet`, `jdouble rigidity` are clearly copies of the original parameters in the java method, so these do not need extensive explanation. However, there are two additional parameters that are critical to understand in order to implement the C code for calculating the forces within the system. `JNIEnv* the Env` is a reference to the Java environment from which the call to this method came. This pointer allows for access to functions that mediate between Java and C/C++ such as object casting. The value of the second parameter, `jobject theObject`, depends on declaration in the Java environment. If a native method is defined `static` then the reference is to the class of the object from which the method is called. However, if the method is not declared `static`, then the reference is to the object instance from which the method is called. Either way, it allows for access to the object from which the method was called.

In order to do any calculation on the data, the references passed from Java must first be cast to the proper C types. C does not have a concept of objects and Object Oriented Programming (OOP), hence the references must be cast to primitive types' pointers as follows:

```
jdouble* xSet = (*theEnv)->GetDoubleArrayElements(theEnv, jXSet, 0);
jdouble* ySet = (*theEnv)->GetDoubleArrayElements(theEnv, jYSet, 0);
jint* repulsionSet = (*theEnv)->GetIntArrayElements(theEnv, jRepulsionSet, 0);
jdouble* dxSet = (*theEnv)->GetDoubleArrayElements(theEnv, jDXSet, 0);
jdouble* dySet = (*theEnv)->GetDoubleArrayElements(theEnv, jDYSet, 0);
```

There is no particular distinction between an `int` or a `jint` so the references are interchangeable. However, a `jintArray` is not the same as a `int*`, so a conversion needs to take place. Once the conversion takes place, the force calculation algorithm is trivial. The program iterates through all the nodes calculating the repulsion forces between all nodes in a brute force manner, making the algorithm of order $O(n^2)$.

4.5.2 Run Time Results of Employing C Code

This algorithm was compared to its counterpart in Java by taking the original algorithm used by java and removing the neighborhood radius check such that the neighborhood used is always the complete set of nucleotides. This setup tests if the calculations are done faster in C or in Java. For this set of experiments, 20 runs were made to test how long it takes a particular structure to stabilize, and these times were averaged with both the average and 99% confidence intervals recorded. The code:

```
ThreadMXBean bean = ManagementFactory.getThreadMXBean();
long startUserTimeNano = bean.getCurrentThreadUserTime();
```

```
//perform task ...

long taskUserTimeNano = -(startUserTimeNano-bean.getCurrentThreadUserTime());
jVizCore.CPUnanosecondsElapsed += (taskUserTimeNano/1000000.0);
```

was employed again to obtain the stabilization time readings.

Figure 4.37 shows the running times for all the structure, while Figure 4.38, Figure 4.39, Figure 4.40 and Figure 4.41 show the stabilization time of sequences as they are divided into four groups based on sequence length.

Figure 4.37 shows that across all sequence sizes, the C code seems to work faster. This does not come as a surprising discovery since the C code performs exactly the same calculations as its Java counterpart. The only element that is being traded off in favour of faster calculations is the overhead required to enter and exit the method responsible for these calculations. This is no longer an issue of simply loading the function on the stack and unloading it, but now there is need to copy all the elements being passed as arguments and perform a cast of the Java elements into appropriate C elements. It is not surprising to find that the trade-off reaps beneficial outcomes with large sequences (as Figure 4.38 shows), the time spent performing the calculations for all sequences probably greatly outweighs the time required to copy the elements and cast them into proper types. However, it is more surprising to see the trend persist as smaller sequences are being inspected. Figure 4.39 and Figure 4.40 show that for medium sized sequences there is still an decrease of between 50% to 70% of the running time when using C code.

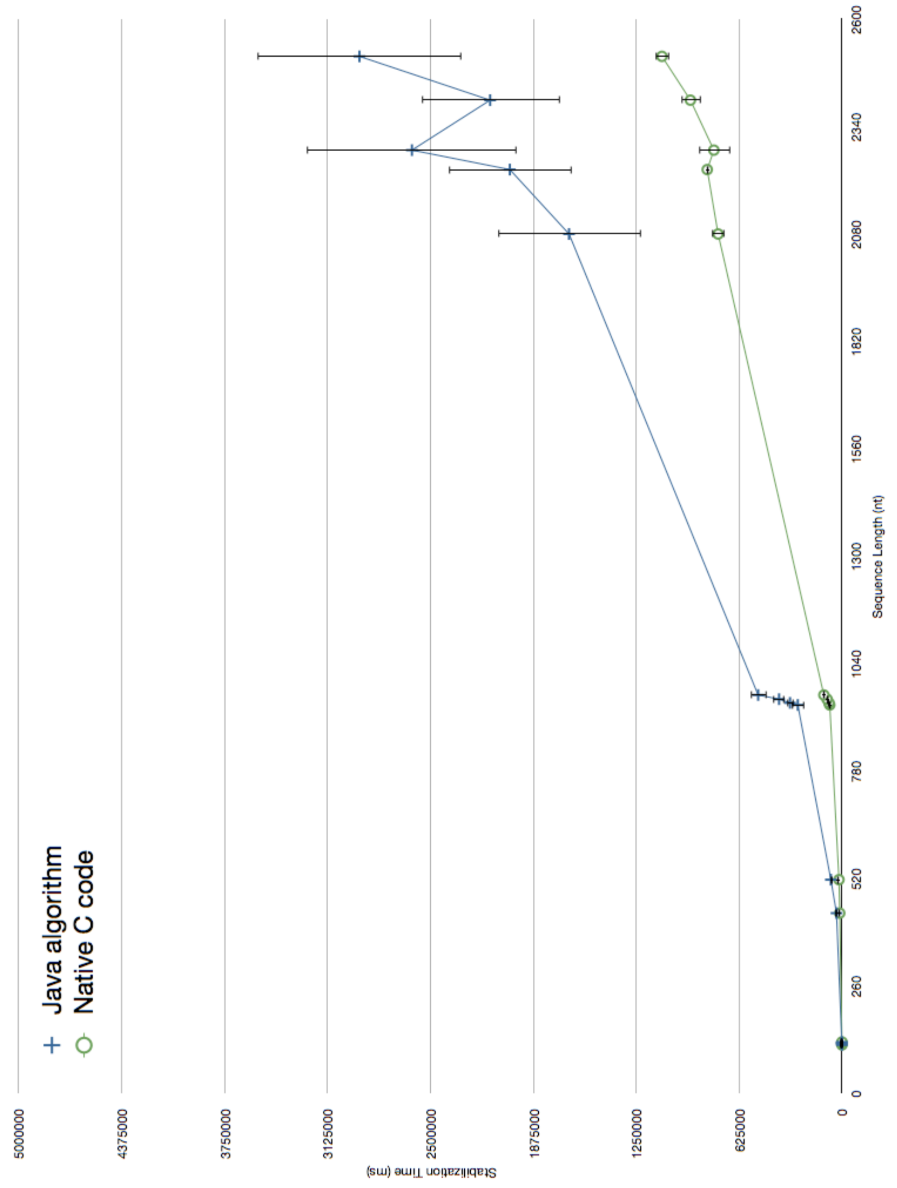


Figure 4.37: The stabilization times (in milliseconds) for the different RNA sequences using Java code versus C code. 99% confidence intervals are used as error bars

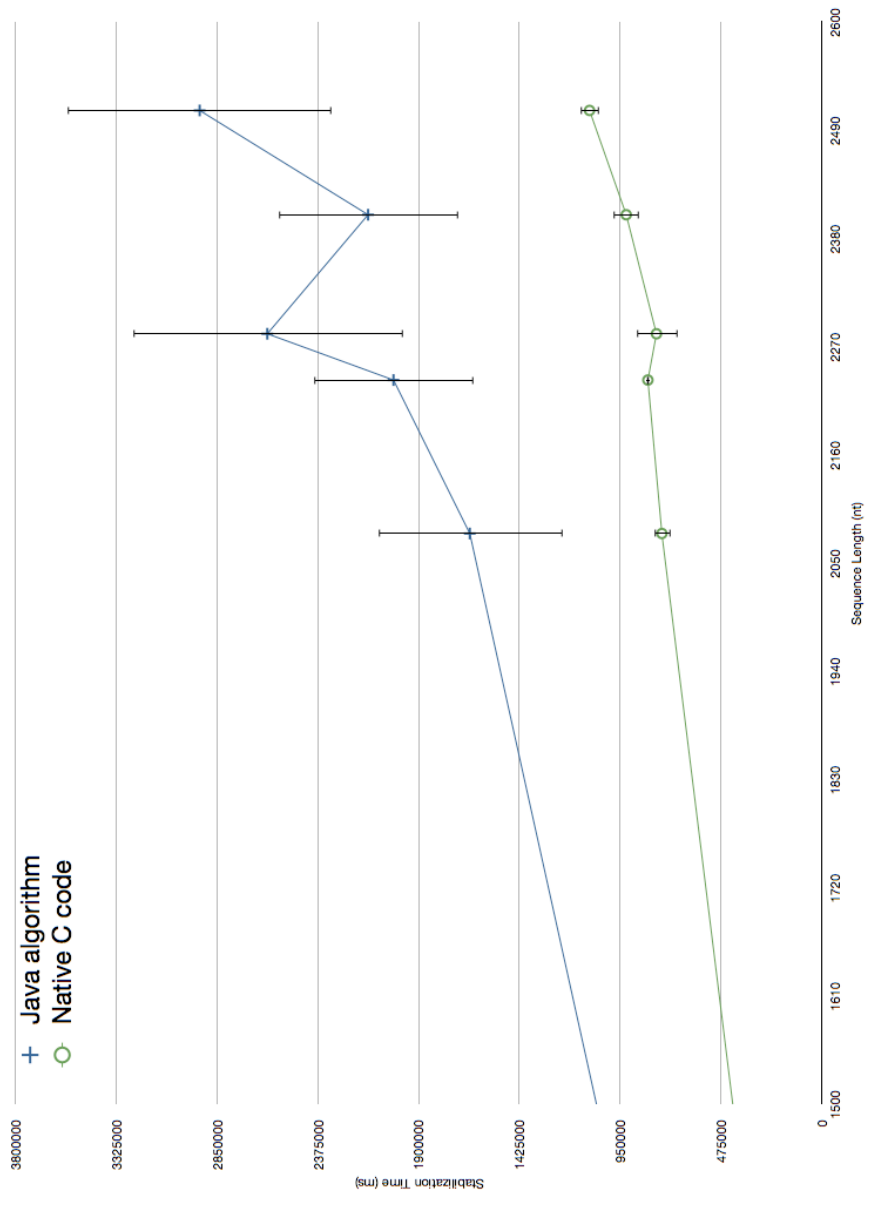


Figure 4.38: The stabilization times (in milliseconds) for *Chlorella saccharophila* 18S ribosomal RNA (AB058310), *Porphyra leucosticta* 18S ribosomal RNA (with intron) (AF342746), *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345), *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) and *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) Using Java code versus C code. 99% confidence intervals are used as error bars

It is even far more surprising to find that a meaningful improvement in running time even occurs with sequences between 117nt and 124nt in length, as Figure 4.41 shows. Despite the confidence intervals suggesting that for some smaller sequences the improvements may not be as effective, the truth remains that for all sequences some degree of improvement occurs. This means that insofar as run time optimization is concerned, if the calculation of the bodies' repulsion forces is done in C in a way that offers minimal function overhead, there is merit to implementing this method universally for all sequences inspected by jViz.RNA.

The only drawback may occur if there is any sort of distortion of the structure to the extent that the drawn structure does not do well in highlighting the RNA structural elements. Intuitively, this should not happen since, again, the calculations remain the same and its only the medium used to perform them that changes.

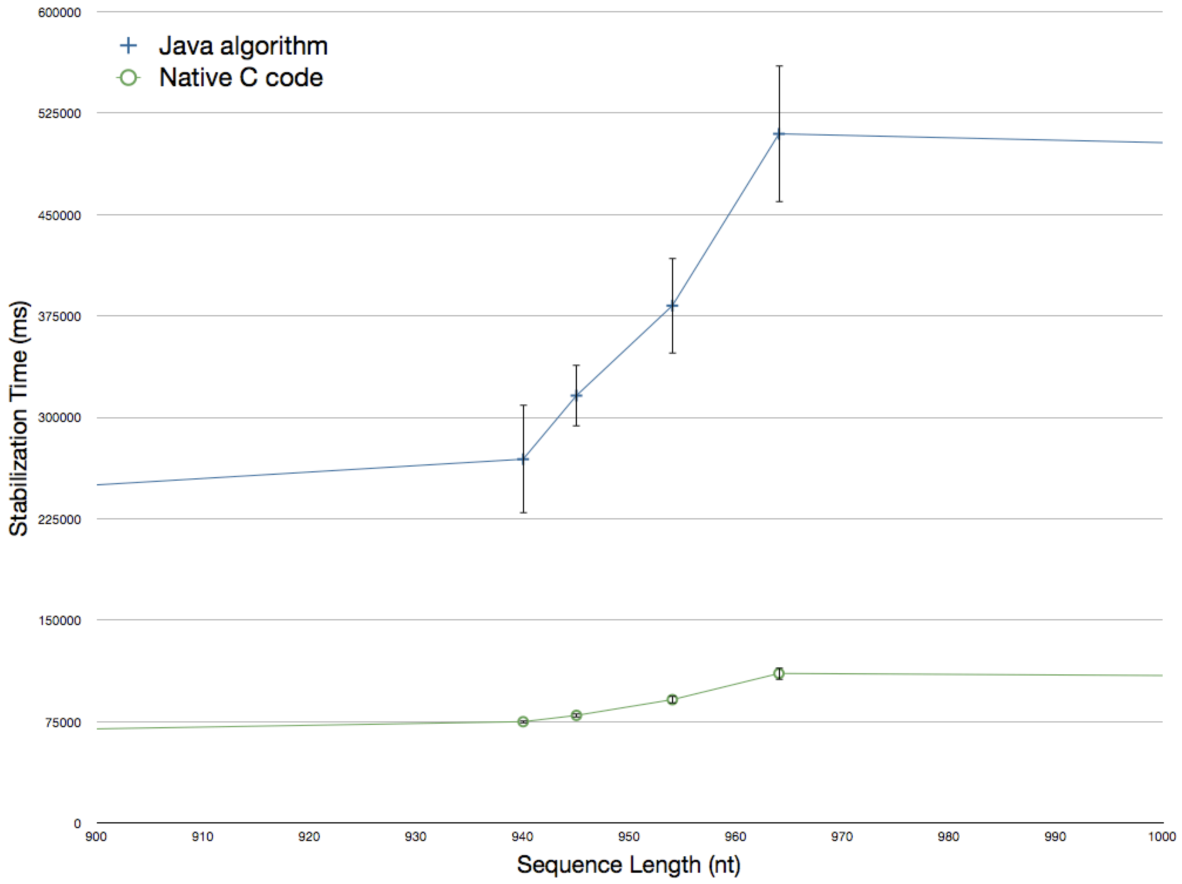


Figure 4.39: The stabilization times (in milliseconds) for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511), *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601), *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) and *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using Java code versus C code. 99% confidence intervals are used as error bars

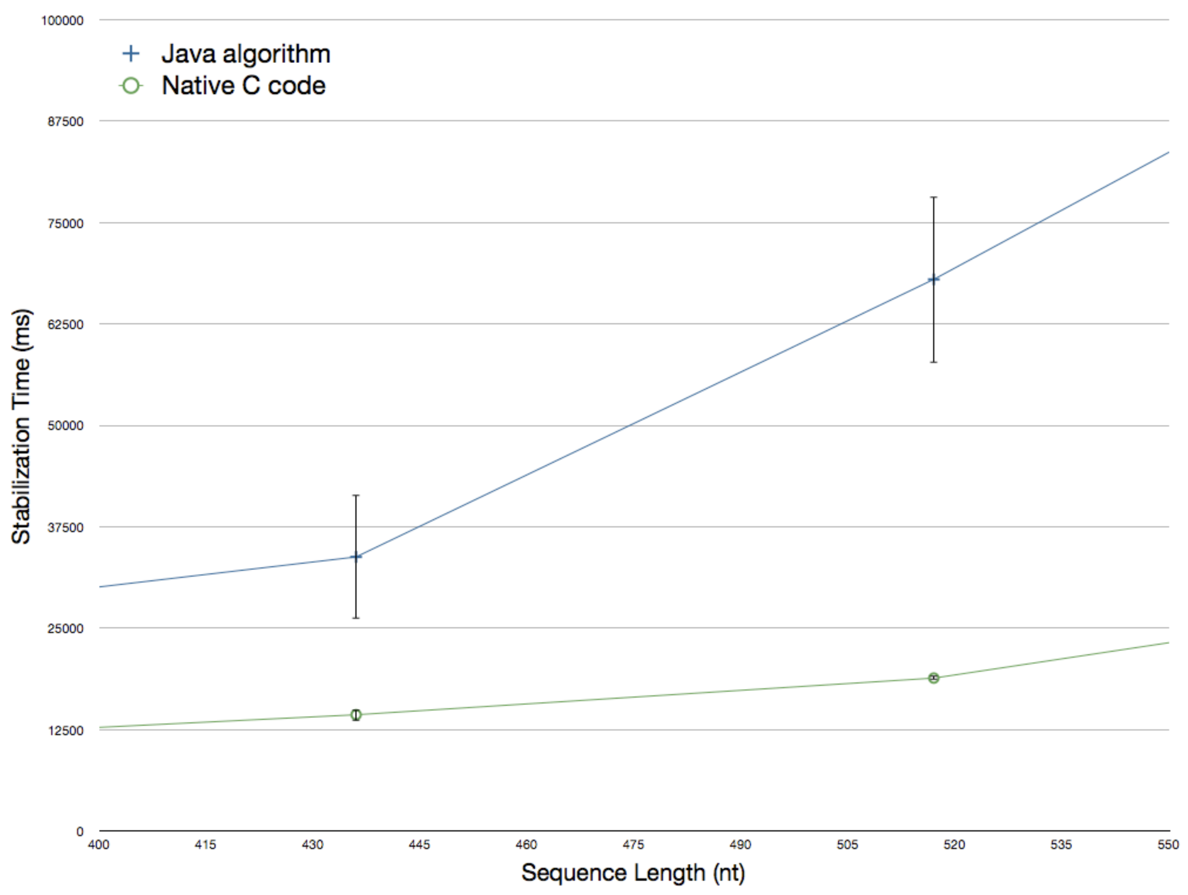


Figure 4.40: The stabilization times (in milliseconds) for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) and *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using Java code versus C code. 99% confidence intervals are used as error bars

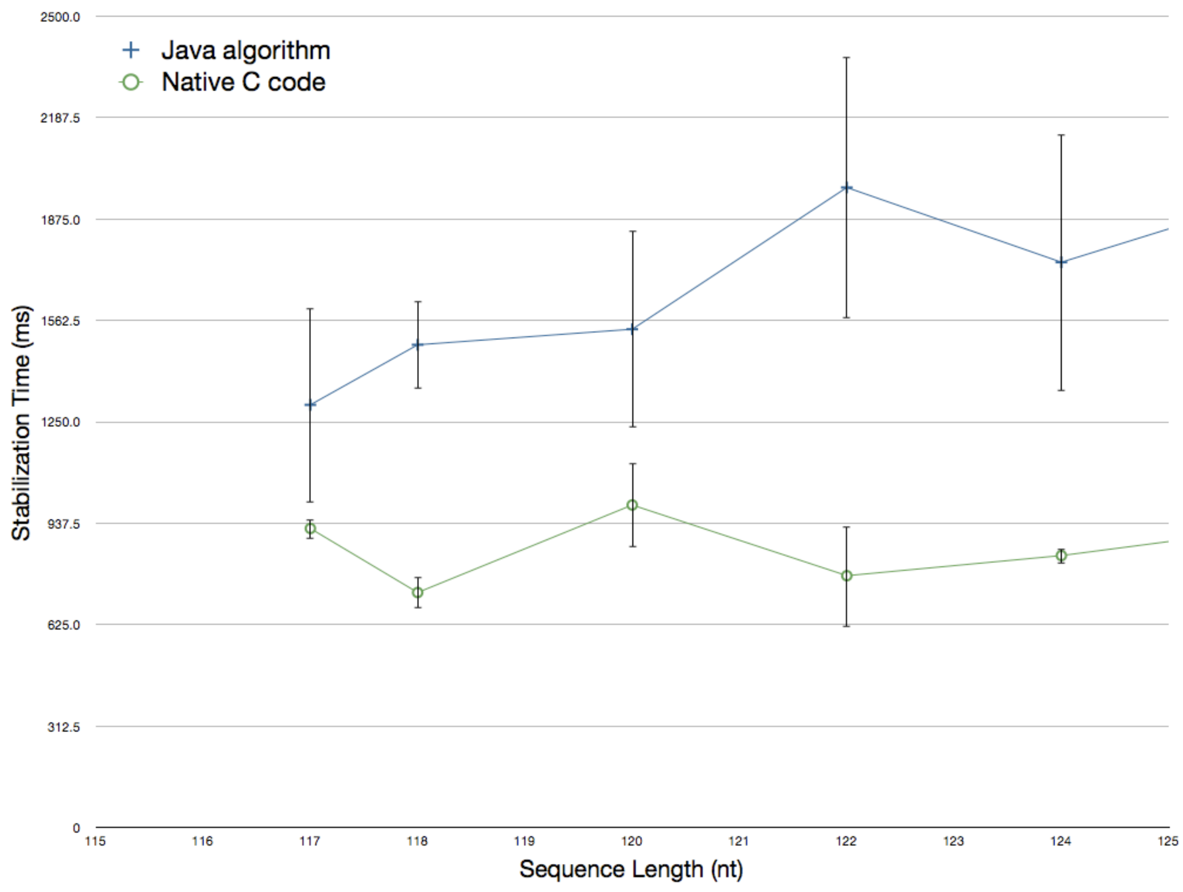


Figure 4.41: The stabilization times (in milliseconds) for *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515), *Arthrobacter globiformis* 5S ribosomal RNA (M16173), *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627), *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579) and *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080) using Java code versus C code. 99% confidence intervals are used as error bars

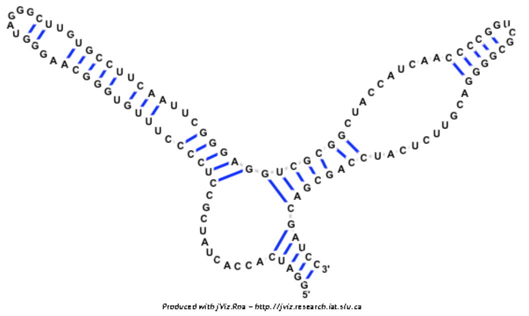
4.5.3 Effects on Structure Presentation

Using a different environment to perform the same calculations should not introduce any variance between results. However, for the sake of completeness it is important to inspect the resulting structure layouts obtained from using C code to perform the calculations. Figures 4.42 - 4.45 show the structure layouts obtained from the use of C code to perform the force calculations. As before, these are images selected to illustrate the general trend of structure layout. The layouts of the additional eight sequences can be found in Appendix A, Figures A.41 - A.48. Again, the implementation explored here used the $O(n^2)$ brute-force method to perform the calculations and so one would expect results no worse than the ones observed in Figures 4.12 - 4.15.

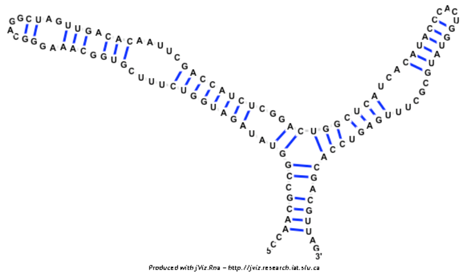
Looking at the images of the resulting structure layouts, one can see that much like the results observed when using multithreading, this implementation presents a very clear presentation of the major structural elements, while preventing overlap of structural elements over one another. Furthermore, any small distortions in the sequences can again be corrected by users to suit their needs. It is easy to understand why no major distortions occur. The use of C code to calculate the displacement for all nucleotides first, and only then apply it to the entire set of nucleotides guarantees that the calculations for the nucleotide displacements stay independent, allowing the same values that are calculated in Java to be calculated in C.

The fact that integrating C libraries into the Java based jViz.RNA creates no distortions in the sequence guarantees that C code can also be used for the more sophisticated nucleotide interaction calculations that involve using the increasing ”neighborhood.” Furthermore, since both C code and multithreading are different methods of performing the same calculations, it is very likely these methods can be combined to create an even faster implementation of the algorithm, but without introducing any significant distortions to the structure.

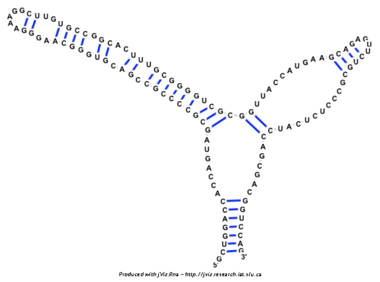
Conclusively, Figures 4.42 - 4.45 demonstrate the using C code to perform a large number of calculations could prove very beneficial to jViz.RNA. This method drastically decrease computation time with minimal effect on the layout of the structures viewed.



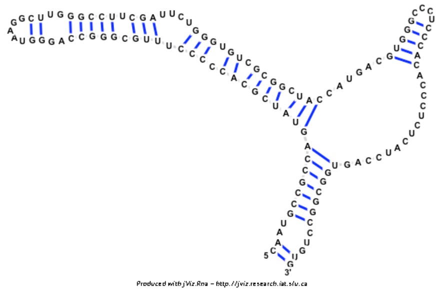
(a) the sequence layout of *Bacillus stearothermophilus* 5S ribosomal RNA (AJ251080)



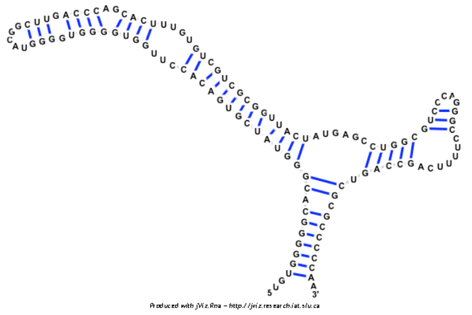
(b) the sequence layout of *Saccharomyces cerevisiae* 5S ribosomal RNA (X67579)



(c) the sequence layout of *Agrobacterium tumefaciens* 5S ribosomal RNA (X02627)



(d) the sequence layout of *Arthrobacter globiformis* 5S ribosomal RNA (M16173)



(e) the sequence layout of *Deinococcus radiodurans* 5S ribosomal RNA (AE000513:254392-254515)

Figure 4.42: The resulting sequence layout for the shortest five sequences using native C code

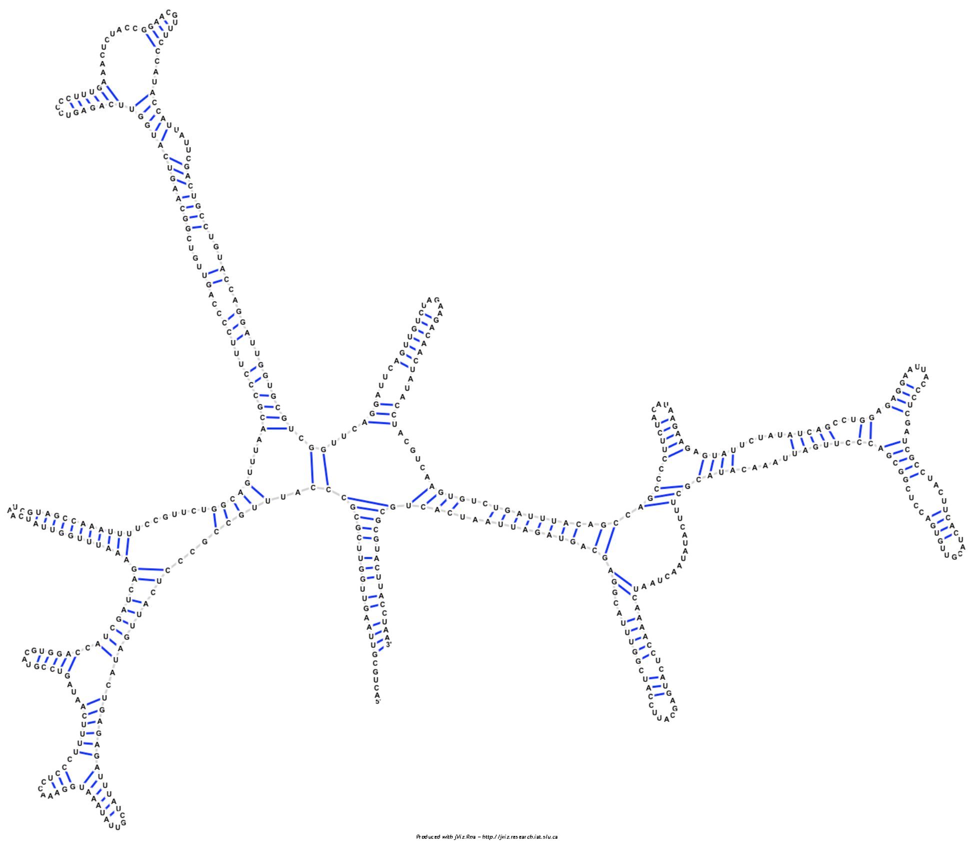


Figure 4.43: The resulting sequence layout for *Tetrahymena thermophila* 26S ribosomal RNA fragment (with intron) (V01416) using native C code

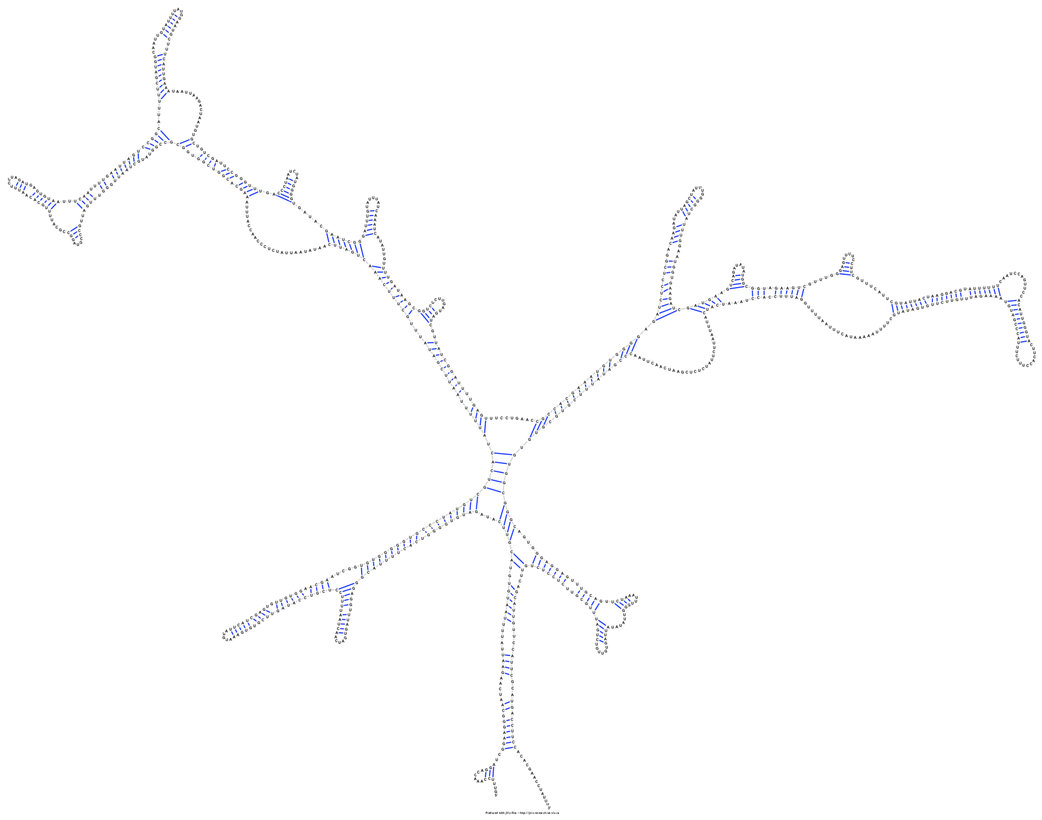


Figure 4.44: The resulting sequence layout for *Ailurus fulgens* mitochondrial 16S ribosomal RNA (Y08511) using native C code

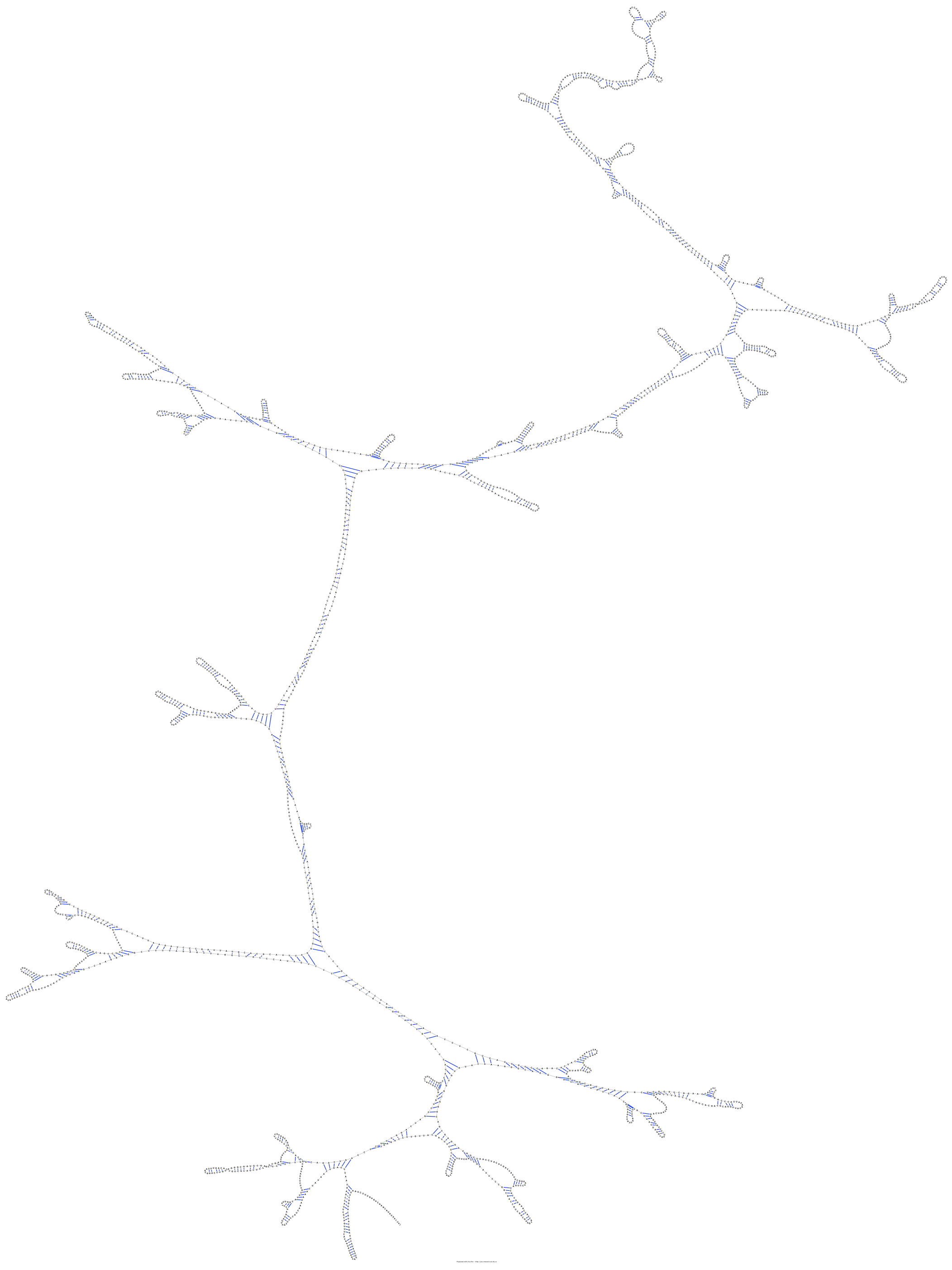


Figure 4.45: The resulting sequence layout for *Chlorella saccharophila* 18S ribosomal RNA (AB058310) using native C code

Inspecting these results, it becomes apparent that using C code is favorable to using Java code even despite the fact that this introduces an overhead for using a different environment. C code calculations are done without any interpretation, unlike Java which uses the JVM. Even with the introduction of the JVM HotSpot, which allows for code the is used often to be converted into machine code [15], C code is just more efficient since it is first compiled into machine code, and then executed.

However, with these results in mind it is also important to remember the drawbacks of C/C++ integration into Java. Java's unique property is that due to the JVM, .jar files can be run on any OS that contains the JVM regardless of which architecture they were compiled on. With the introduction of C code, jViz.RNA loses its universality and becomes architecture dependent. If one wishes to run jViz.RNA on their machine, and they wish to take advantage of the speed improvement C can offer, they need to make sure that there is a dynamic C library compatible with their architecture. That means a separate library for MacOS, Windows and Linux architectures, as well as constant updates as the different operating systems change. This problem is usually hidden from users since it is the JVM that is being updated and tailored to operating systems, such that the java programs do not have to be.

With the current setup in place, and the experiments that show there is indeed merit into integrating C code into jViz.RNA, it may be a good idea to develop separate dynamic libraries for the major operating systems that exist, and supply those along with jViz.RNA. The block loading the libraries can be surrounded by a `try catch` block to prevent errors from crashing the program as follows:

```
static
{
    System.out.print("Attempting to load library from ");
    System.out.println(""+System.getProperty("java.library.path"));
    try
    {
        System.loadLibrary("DynamicCLibrary);
    }
    catch (Error e)
    {
        System.out.print("Could not load library from ");
        System.out.println("" + System.getProperty("java.library.path");
        System.out.println("Error: "+e);
        // Set some flag that C code cannot be used...
    }
}
```

This way, if loading the appropriate library fails, the program can still run using Java code, and the users are aware of the reason for the failure (this helps in reporting bugs).

4.6 Structure Recall

Often times a particular structure is viewed more than once. If a structure is inspected several times during one sitting, the user does not need to wait for the structure to stabilize for every viewing session. The graph layout is always available during a single run. However, a structure may be viewed multiple times over multiple sittings. In a situation like that it would convenient for jViz.RNA to reload the structure in the way it was last seen and avoid recalculating its stable layout again. This is the essence of structure recall, where jViz.RNA recalls the shape and layout of a structure the way it was last seen. This new functionality introduced several problems for implementation. First, a way to store the information of the sequence layout while jViz.RNA is not running is required, more specifically, the kind and content of files to do that would have to be decided. Second, the mechanism of saving and loading the data of such files could be either automatic and hidden from the user, or manual, giving the user more control. These design decisions would have to be resolved in order for an effective structure recall method to be available to the user.

4.6.1 Designing the Files for Recall

First, there are two main methods of storing the data of structure co-ordinates in files: flat files and structured files. Flat files would be files where the binary data of the structure x and y coordinates are written. These would be only useful in the context of jViz.RNA since only jViz.RNA would know how to interpert the data in the files. However, access time would be much quicker since no interpretation would be necessary, the binary data would be available for jViz.RNA immediately after reading. Structure files would structure the coordinates in some way and perhaps even have tags denoting which coordinates apply to which base. These would be useful to, and editable by, users of jViz.RNA, but saving and reading these files would take longer since interpretation would have to happen between binary data and string representations. The structured file choice doesn't seem very good since a user won't be able to map the set of points to a mental picture easily, so reading and editing the files as are would not be a very useful feature. As a result, the flat binary file system was chosen.

Coordinate files produced with jViz.RNA would have the ending `.rnacor` (stands for RNA coordinate file), and they would be named according to the original file's name. The original file's name would have it's dot (.) removed and replaced with a dash (_), then the ending `.rnacor` would be appended.

For example, if a file's original name is

`Haloarcula marismortui 16S ribosomal RNA gene sequence_1470bp.ct`

the new coordinate file corresponding to that structure would be

`Haloarcula marismortui 16S ribosomal RNA gene sequence_1470bp_ct.rnacor`

The files content would contain a binary data of *x*,*y* coordinates for all the bases of the sequence. i.e. when read, the data could be put into a $n \times 2$ array, where n is the number of bases for the sequence. The files would be written to the same directory where the original structure files are.

4.6.2 Reading the Recall Files

With the file structure determined, the next design challenge would be how, and when, should file data be loaded into jViz.RNA and written by it. The most intuitive decision would be to load a `.rnacor` file when a structure file that has a matching `.rnacor` file is loaded. In other words, if the user chooses to view the structure of `Desktop/RNAfiles/RNAStructure1.ct`, then jViz.RNA would look for `Desktop/RNAfiles/RNAStructure1_ct.rnacor` and would use the information in that file to display the structure rather than stabilizing it again.

The design of the `RNACoordinateFile` class was as follows:

```
public class RNACoordinateFile extends File{

     double[][] XYDataPoints;
     FileInputStream myInputStream;
     FileOutputStream myOutputStream;
     DataInputStream myDataInputStream;
     DataOutputStream myDataOutputStream;



     /*****************************************************************************************/
     /****************************          METHOD DEFINITIONS        ************************/
     /*****************************************************************************************/

     public RNACoordinateFile(String filePath)
     {
          super(filePath);
          try
          {
               createNewFile();
          }//end try
          catch (java.io.IOException e)
          {
               //do some stuff
          }//end
          try
          {
               myInputStream = new FileInputStream(filePath);
          }//end try
          catch (java.io.FileNotFoundException e)
          {
               // do some stuff
          }//end catch (java.io.FileNotFoundException)
     }//end constructor

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

     public double[][] getDataPoints(int nodeNumber)
     {
         // Some code to return the data points represented in THIS file...
     }//end getDataPoints()


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

     public void writeDataPoints(double[][] theDataPoints, int nodeNumber)
     {
           // Some code to write the data points represented in THIS file to a file on
           // the hard drive...
     }//end writeDataPoints(double[][] theDataPoints, int nodeNumber)

}
```

The class contains five attributes, of which only the first requires extensive explanation. `XYDataPoints` is a reference to a two dimensional array that contains the *x*,*y* pairs of a structure. `myInputStream`, `myOutputStream`, `myDataInputStream` and `myDataOutputStream` are all attributes needed to read and write the points' data to and from file.

When an instance of `RNACoordinateFile` is created, the constructor is called. It is important to notice here that the constructor uses the `File.createNewFile()` method. This method creates a new file on the hard drive if and only if a file by the name `filePath` does not exist [21]. As a result, a new instance of `RNACoordinateFile` does not always create a new file. However, it always connects to a file by the name `filePath`, whether it needs to create it or not, since the statement
`myInputStream = new FileInputStream(filePath);` is always executed.

When a new RNA structure file is loaded, a new instance of a subclass of the `StructureFile` class is created (e.g. CtFile, Bpseq, etc'). Every instance of a subclass of `StructureFile` contains a reference to an instance of `RNACoordinateFile` as follows:

```
public abstract class StructureFile extends File
{
    protected RNACoordinateFile myCoordinateFile;

    // The rest of StructureFile.java...
}
```

In addition, whenever an instance of a subclass of `StructureFile` is created, it links to a `.rnacor` file (even if one has to be created) of the appropriate name as follows:

```
public abstract class StructureFile extends File
{
     protected RNACoordinateFile myCoordinateFile;



    public StructureFile(String pathname)
    {
         super(pathname);
         String coordinateFileName = pathname.replace('.', '_')+".rnacor";
         myCoordinateFile = new RNACoordinateFile(coordinateFileName);
    }

    // The rest of StructureFile.java...
}
```

This results in that whenever a new structure is loaded, that structure is immediately linked to a `.rnacor` file for I/O purposes. This process is hidden from the user, so he or she does not need to concern himself or herself with it.

After a structure file is loaded, a structure is created using the data in the structure file. At that point a new instance of `RnaStructure` is created. Every instance of `RnaStructure` also contains a reference to a `RNACorrdinateFile`, as well as two other related attributes as follows:

```
public class RnaStructure extends StructureGroup
{
     private RNACoordinateFile myCoordinateFile;//BORIS CHANGE
     private double[][] coordinateArray;
     public boolean hasCoordinateFile;


     // The rest of RnaStructure.java...
}
```

The two dimensional array `coordinateArray` keeps track of the coordinates of the current RNA structure's bases, while `hasCoordinateFile` is used to determine if the linked `RNACoordinateFile` has just been created (meaning it contains no data), or if it was written to before (and thus can be used to skip the structure stabilization process).

Unlike the `StructureFile` class, `RnaStructure` does not create any new instances of the `RNACoordinateFile` class. Instead, the method `RnaStructure.setCoorFile(RNACoordinateFile theFile)` is called when the structure is created. The method's body is as follows:

```
public void setCoorFile(RNACoordinateFile theFile)
{
     hasCoordinateFile = true;
     myCoordinateFile = theFile;
     this.coordinateArray = myCoordinateFile.getDataPoints(nucList.size()+2);
     if (coordinateArray == null)
```

```
       {
              hasCoordinateFile = false;
              this.coordinateArray = new double[2][nucList.size()+2];
       }//end if (coordinateArray == null)
}//end setCoorFile(RNACoordinateFile theFile)
```

It begins by checking if the coordinate file passed to it has any data in it. If it does, the resulting data points would be put into the array `coordinateArray`, if it doesn't, `coordinateArray` is declared to contain the *x* and *y* coordinates of the current structure. The reason the structure is defined to have `nucList.size+2` bodies is since the 5' and 3' ends are shown as labels in the structure and are considered part of the collection of bodies. By the end of this method's execution, the `RnaStructure` instance is pointing to a coordinate file and has an array of *x*, *y* coordinates.

When the user chooses to view the structure, a `StructureEvent` object gets created and is sent to all other elements of jViz.RNA that deal with the structure presentation. This `StructureEvent` object can deliver information about the backbone of the selected structure using a `BackBoneInfo` object. The `BackBoneInfo` object is defined as follows:

```
public class BackBoneInfo
{

       public ArrayList backBone;
       public double[][] rnaCoordinates;
       public boolean hasCoordinateFile;


       public BackBoneInfo(ArrayList backBone,
                           double[][] rnaCoordinates,
                           boolean hasCoordinateFile)
       {
              this.backBone = backBone;
              this.rnaCoordinates = rnaCoordinates;
              this.hasCoordinateFile = hasCoordinateFile;
       }

}
```

The three attributes of this object are all public allowing for easy access. The `backBone` attribute is an array of all the nucleotides of the structure, the `rnaCoordinates` attribute is a reference to the *x*,*y* coordinates an `rnaStructure` has, and the `hasCoordinateFile` variable denotes if the *x*,*y* coordinate array can be used for laying out the structure, or if it is an empty coordinate array, meaning the structure has to be stabilized.

The canvas responsible for displaying the structure (`CsCanvas`) receives the `StructureEvent` object and requests the `BackBoneInfo` associated with that event to set up the backbone structure. In short, when the user selects to view a new structure, the coordinates of that structure are passed along via references to the canvas that is responsible for displaying the structure. The `CsCavas` then calls the `buildBackBone` method, which is defined as follows:

```
public Vector<Node> buildBackBone(ArrayList backbone,
                                  double[][] backboneCoordinates,
                                  boolean validCoordinateFile)
{
    //Setup a vector for holding all these nodes
    Vector<Node> nodeList = new Vector<Node>();

    //Setup the transform for drawing the nodes in a circle so
    //that we don't get crossing over in the structure

    //First get the radius we're going to use
    //We'll treat the circumference of the circle as being the nucleotide length
    int radius = (int)((((backbone.size() + 1) * 50) / Math.PI)/2);

    //Now we'll figure out how many degrees we need to rotate for each nucleotide
    double rotationDegrees = 360.00 / (backbone.size() + 10);

    //Convert to radians
    double rotationRadians = Math.toRadians(rotationDegrees);

    //Make an Affine Transform
    AffineTransform rotationTransform = new AffineTransform();
    rotationTransform.setToRotation(rotationRadians);

    //Make the point we're going to rotate
    Point2D.Double drawPoint = new Point2D.Double(0,radius);

    //Do a massive try (catch errors during node creation)
    try{
        //...
        Node previousNode = new Node();

        //Build the 5' End
        previousNode = new Node("5'","5'");

        //Find our point to draw at in the circle
        rotationTransform.transform(drawPoint, drawPoint);
        //Set the node to that point


           if ((backboneCoordinates != null) && (validCoordinateFile))
           {
                previousNode.x = backboneCoordinates[0][0];
                previousNode.y = backboneCoordinates[1][0];
           }
           else
           {
                previousNode.x = drawPoint.getX();
                previousNode.y = drawPoint.getY();
           }//end else
        glPanel.addNode(previousNode);

        //Iterate and build all nodes
        for(int iNode = 0; iNode < backbone.size(); iNode++)
        {
            //Build the node
            Node currentNode = new Node(Integer.toString(iNode),(String)backbone.get(iNode));
            //config.setNodeProperties(currentNode);
            //Find our point to draw at in the circle
            rotationTransform.transform(drawPoint, drawPoint);
            //Set the node to that point
                if ((backboneCoordinates != null) && (validCoordinateFile))
                {
                     currentNode.x = backboneCoordinates[0][iNode+1];
                     currentNode.y = backboneCoordinates[1][iNode+1];
                }
                else
                {
                     currentNode.x = drawPoint.getX();
                     currentNode.y = drawPoint.getY();
                }//end else

            //Add it in
            glPanel.addNode(currentNode);
            nodeList.add(currentNode);

            //Build and add the edge...

            //And shuffle along
            previousNode = currentNode;
        }
        //Build the 3' End
        Node threePrimeNode = new Node("3'","3'");
        //Find our draw point on the circle
        rotationTransform.transform(drawPoint, drawPoint);
        //Set the node to that point
           if ((backboneCoordinates != null) && (validCoordinateFile))
           {
                threePrimeNode.x = backboneCoordinates[0][backbone.size()];
                threePrimeNode.y = backboneCoordinates[1][backbone.size()];
           }
           else
           {
                threePrimeNode.x = drawPoint.getX();
                threePrimeNode.y = drawPoint.getY();
           }//end else

        glPanel.addNode(threePrimeNode);
        // ...
    } catch (Exception e)
    {
        System.out.println("Error while building the backbone");
```

```
            System.out.println(e.getMessage());
        }
        return nodeList;

    }
```

The essence of this method is that it attempts to first lay out the structure as the coordinates file suggests, if the coordinates file is invalid (empty), it resorts to the circular layout and stabilization of the structure[2].

4.6.3 Writing the Recall Files

The writing portion of the `.rnacor` files is much simpler. The files for all active structures are written when the user shuts down jViz.RNA by pressing the ”close” button. When the user selects this option, a message is sent to the `StructureManager` to execute the method `saveData()`. This method calls each `RnaStructure`'s `saveData()` method which is defined as follows:

```
public void saveData()
{
     myCoordinateFile.writeDataPoints(coordinateArray, nucList.size()+2);
}//end saveData
```

This method calls the `RNACoordinateFile` associated with the current structure to write the data of this structure's x, y coordinates to the flat file.
This setup begs the question as to what happens when a user views several different structure with jViz.RNA. Since only one structure is visible at all times, the coordinates of any structure are updated when the user views a new structure. This way, at any given moment in time, every structure has a list of the coordinates of its bases as they were last seen on the screen. When the user chooses to exit jViz.RNA, all structures save their data into the appropriate flat files.

This structure recall mechanism allows for a dramatic increase in user interaction for both large and small structures. When a user requests to see the structure a 2^{nd}, 3^{rd}, etc' time, the structure would load at a time of order $O(n)$ where n is the size of the structure. Furthermore, there is no obligation of the user to use the produced `.rnacor` files or keep them. If he or she does not want the data in the files anymore, deleting them would allow jViz.RNA to re-stabilize the structure when it is loaded. Furthermore, the files can be renamed to include their dates so a user can copy the `.rnacor` file and rename the copy to keep track of how different structures were viewed and laid out across many sessions. In essence, this setup produces a quick and hidden mechanism for the user to view structures

[2]The stabilization of the structure is actually controlled by another class, `TGLayout`, but a flag is set in it if the layout is loaded from the coordinates file not to stabilize.

quickly if they have been viewed before, while still giving the user some control of this mechanism by allowing them to remove or modify the `.rnacor` files produced. Despite applying a simple principle, this is one of the most significant contributions to jViz.RNA done during this thesis work.

4.7 Chapter Recap

In this chapter five major run time optimizations were reviewed and the results of their individual integration into jViz.RNA were inspected.

First, simple optimization of the existing algorithm was presented. Under this method, the rate at which the "neighborhood" each nucleotide inspects for the force calculation is decreased to give a rougher approximation of the structure layout in exchange for an improvement in run time. The results presented show that this method indeed delivers a drastic decrease of run time, but does so at the price of aesthetics. Nonetheless, the resulting structures can be corrected by users rather easily, offering a drastic increase in run time in exchange for a slight increase in the involvement of users.

Additionally, the use of the Barnes-Hut algorithm was inspected as this algorithm was specifically designed for the *n*-body problem. In this algorithm the area containing the body is recursively subdivided into smaller portions and then portions which contain a group of bodies close enough together are treated as though they contain one single body which encompasses the properties of the many bodies. This method also showed very promising results insofar as decrease in run-time goes, but created aesthetics distortions which could not have been corrected by users. Any algorithm that would introduce such distortions would probably not be fitting to be employed in jViz.RNA since the major focus of the software is visualization.

Next, multithreading was attempted as an addition to jViz.RNA. Multithreading does not add or remove any of the calculations performed, but rather disperses independent calculations so that they can be done in parallel, thus decreasing the time required to perform the entire set of calculations. Although this approach did not show as drastic of improvements as the previous two, it indeed showed a non-negligable improvement coupled with no introduction of distortions into the structure layouts.

Fourthly, the integration of C code to perform the large chunk of calculations involved with force calculations for the structure layout was attempted. In this approach, C code was used to perform the calculations for the repulsion forces which were previously done in Java. This method also proved both its improvement in run time and its loyalty to the original layouts such that the resulting structure layouts still uphold their intended role at displaying major structural elements to the user.

Lastly, structure recall, a process by which a structure can be loaded as it was last viewed, was integrated into jViz.RNA. This method offers colossal run time improvements for structures that were previously loaded since their previously seen layout need not be recalculated by jViz.RNA. This method trades memory use (the production of .rnacor files) for a decrease in run time.

This chapter has outlined the major and most significant contributions done to jViz.RNA not only in finding more promising methods for the force calculation algorithm, but also in exploring methods that have shown to be less promising and disqualifying them as unsuitable candidates for the purposes of this research. The methods described here and their resulting potential improvements offer users of jViz.RNA a much more time efficient visualization experience.

Chapter 5

Conclusion - jViz.RNA, then and now

This thesis introduced an array of improvements to jViz.RNA, augmenting both the number of data sources it can take advantage of, and how fast it uses these data sources. When discussing the implementation of additional file format support, it is easy to see that the implementation of each additional file support extends jViz.RNA to a new audience and allows more people to employ it for their needs. In this thesis, jViz.RNA was expanded to support FASTA and RNAML file formats. The main focus of this expansion is the integration of RNAML files which are gaining more acceptance as a standard file to transfer RNA data.

In addition, this thesis presented several methods which were inspected in the attempt of improving jViz.RNA's run time when calculating a visually appealing layout for the structures. Algorithmic approximation based methods were attempted (Optimization of jViz.RNA's algorithm and the use of the Barnes-Hut algorithm), as well as the use of different environments (multithreading and C) to perform the same calculation originally performed, but faster. The approximation based methods demonstrated a very high improvement in speed, but sacrificed the aesthetics of the resulting structure to obtain a shorter run time. The code optimization presented in Section 4.2 does show promise, since the distortions it presents can be corrected by the user. On the other hand, the Barnes-Hut algorithm does not seem to be an appropriate method to be used in jViz.RNA since it distorts the layout of structures beyond repair. The export of the calculations that need to be done into different environments shows much more promise since these methods offered a speed improvement without jeopardizing the clarity of the drawn structures. Both multithreading and the use of the Java Native Interface allows for the same calculations to be done, but much faster than they are originally performed in Java.

The coding effort which was involved in this thesis is also not trivial. The Barnes-Hut algorithm introduced approximately 1,500 new lines of code into jViz.RNA, the integration of C code into jViz.RNA introduced approximately 700 new lines of code, and multithreading and structure recall each introduced approximately 200 new lines of code each. Combined this expands the code by approximately 2,600 new lines of code. Although it is granted that more code does not always produce better results, it is noteworthy that a great portion of time for this thesis was dedicated to coding and programming, in addition to the time invested in research and design.

The significance of the work presented in this thesis lies mainly in the run time improvements explored and integrated into jViz.RNA. When discussing speed optimizations, the work doesn't just stop with the individual implementations. A second, and no less important, part of the art of optimization is the integration of several ideas into the system. As was noted before, both multithreading and the use of C code lend themselves to an easy combination since the use of one does not require any modifications to the implementation of the other. It is possible to employ multithreading in C [5, 25] and thus combining the advantages both dynamic C libraries and multithreading offer. Furthermore, the algorithm optimization discussed in Section 4.2 can also be integrated along with the other two. In such an implementation, the "neighborhood" of each nucleotide could be searched by multiple threads once it becomes large enough (there may be less merit to employing multithreading over small "neighborhoods"). The combination of these methods remains a topic that needs to be explored since integration of these operations may introduce additional run time cost for efficient management, but the overall result should be a greater improvement in run time than each of the individual methods offers.

This thesis also offers contribution in the research based context. Although specific examples are used here, the work presented here also looks at how different length sequences behaved under different methods of run time optimization, and how the nature of the method used can affect the visual representation of RNA, which is a critical attribute of jViz.RNA. As such, these results can be used as guidelines in other problems such as 3D RNA structure representation, or even protein structure representation.

The work presented here is the first major step in improving the versatility and run-time of jViz.RNA so that it continues to serve more people, better and faster than before. With the introduction of the iPad, and the constant improvement of video cards, one might hope to see jViz.RNA show dynamic ball-and-stick structures of RNA in less then a second. In the future, holographic display might even be common in every biomedical lab, and so jViz.RNA would adjust to that. The main point is that jViz.RNA is a tool designed to mediate between researchers in the life sciences and the powerful computational capacity of computers, and as such it will have to remain evolving just as both the understand of biology and the capacity of computers grows.

It is important to keep in mind that there are also several potential developments that are implementable even now, but were not included in this work due to time constraints. For example, the export and writing of RNAML and FASTA files remains an addition that would be useful. Also, even though the results in Section 4.2 show that there is merit to increasing the multiplier value of the algorithm in order to speed up the stabilization of structures, one must also remember a larger multiplier value means a cruder estimation. This estimation might be acceptable for larger structures, but smaller structures can suffer from over-approximating a problem. Section 4.4 also demonstrated that different size sequences benefit from a different ideal thread number to process them, and using too many threads for short sequences could prove to be counterproductive. As a result, another key element would be integrating a mechanism that would set all the parameters of the different algorithms depending on a given sequence's length.

jViz.RNA is a software employed by many users interested in the research of RNA and its role in cells. This thesis has further built upon two major existing functions in jViz.RNA in order to improve them. The improvement of the stabilization time for classical structures will allow users to go through a greater throughput of sequences or get important results from very large sequences faster. The integration of FASTA and RNAML will allow users to benefit more from the use of jViz.RNA with different software that predict RNA folding and produce a variety of output files. Considering the last two points, it is appropriate to say that in addition to the multitude of functions jViz.RNA offered, the work presented in this thesis allows jViz.RNA to serve more people, and better, in their inquiry of RNA and its properties.

Appendix A

Images of RNA Structures

A.1 Sequences' layouts under the original algorithm employed by jViz.RNA

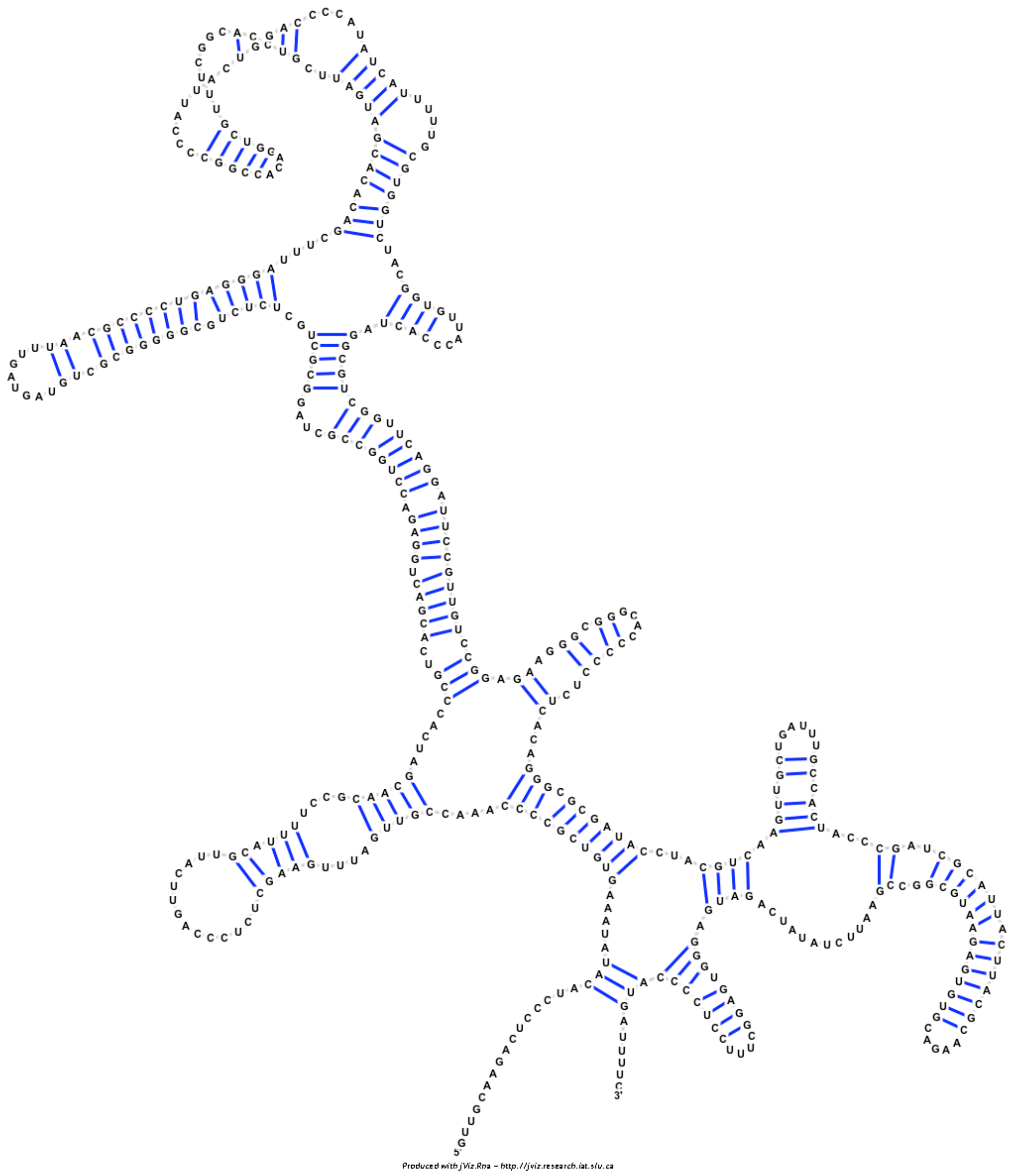


Figure A.1: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the original algorithm employed in jViz.RNA

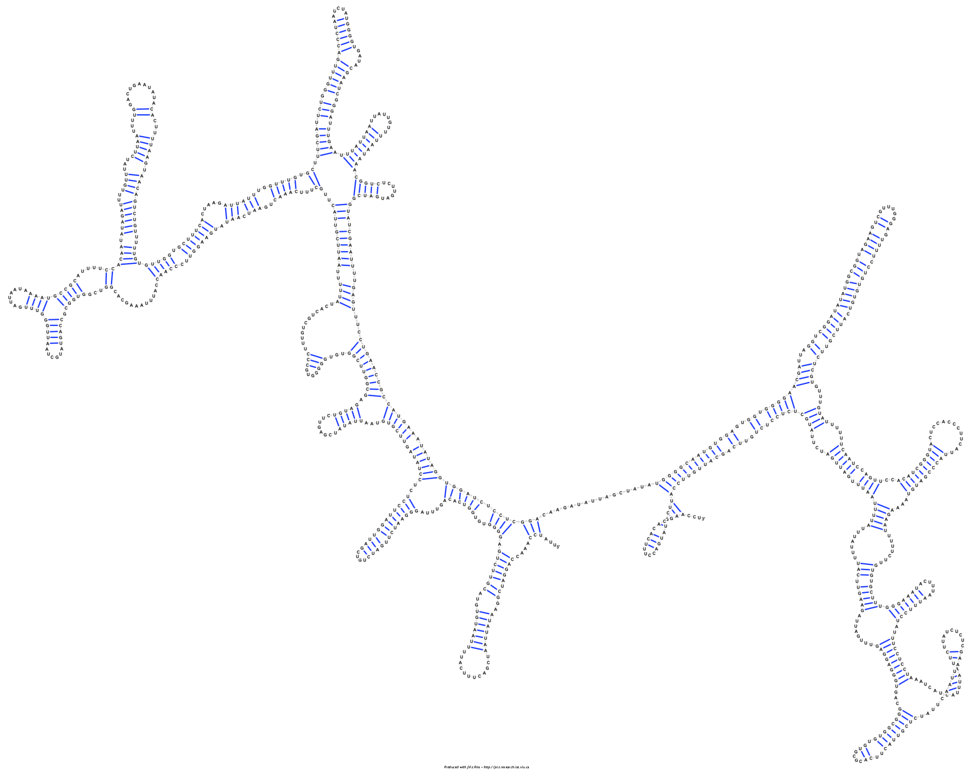


Figure A.2: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the original algorithm employed in jViz.RNA

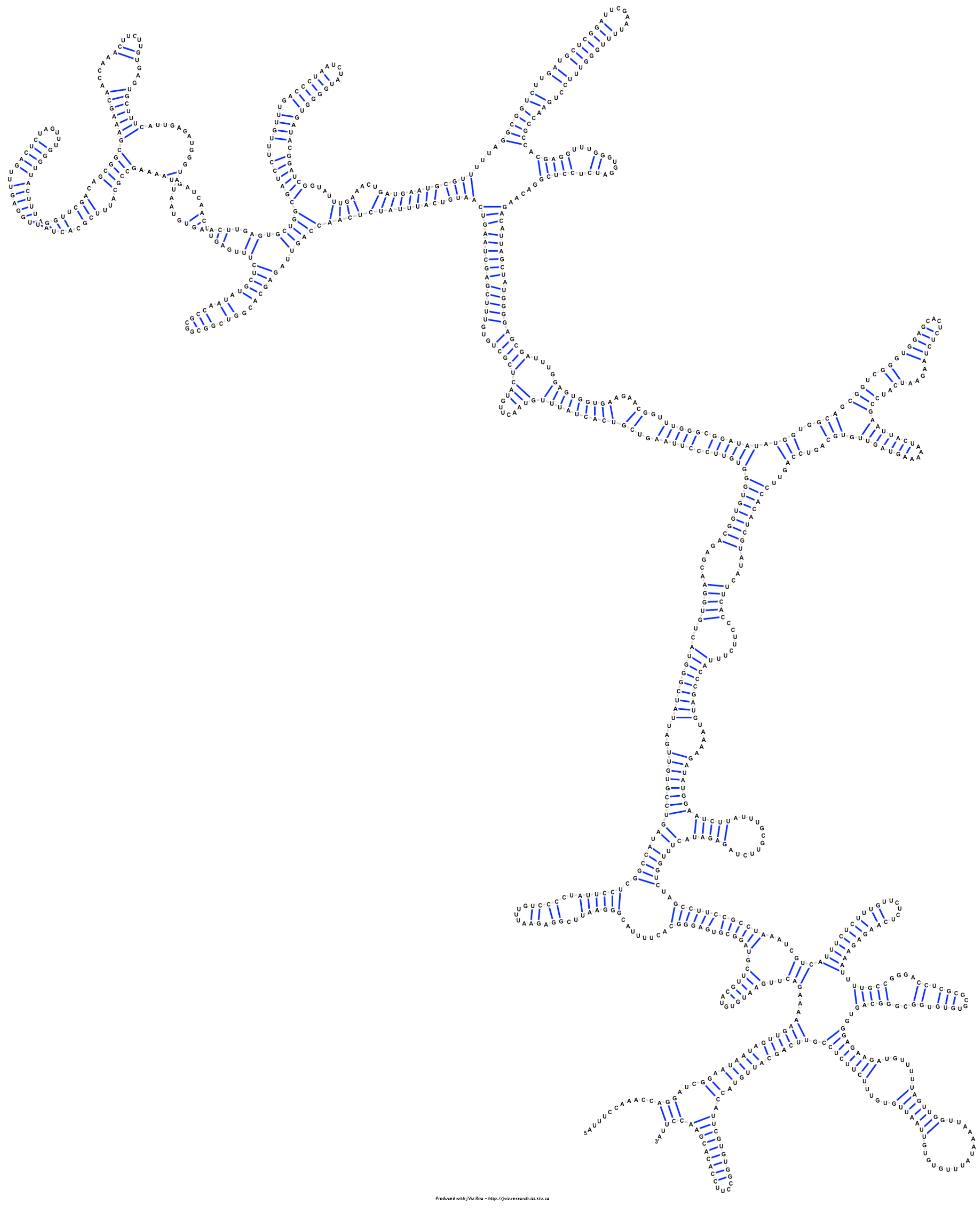


Figure A.3: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using the original algorithm employed in jViz.RNA

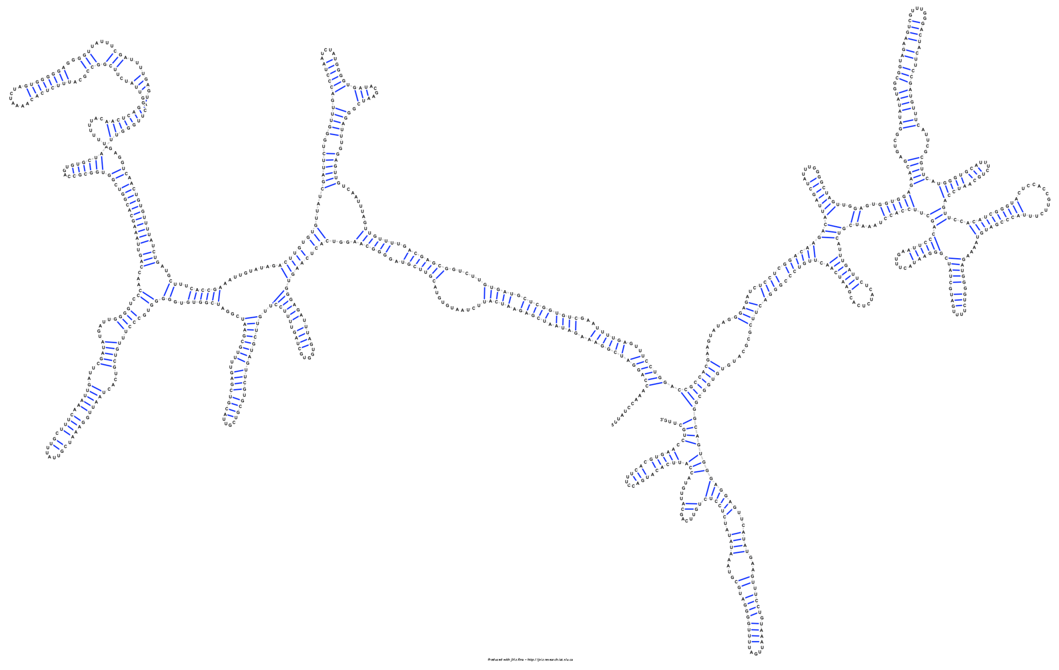


Figure A.4: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using the original algorithm employed in jViz.RNA

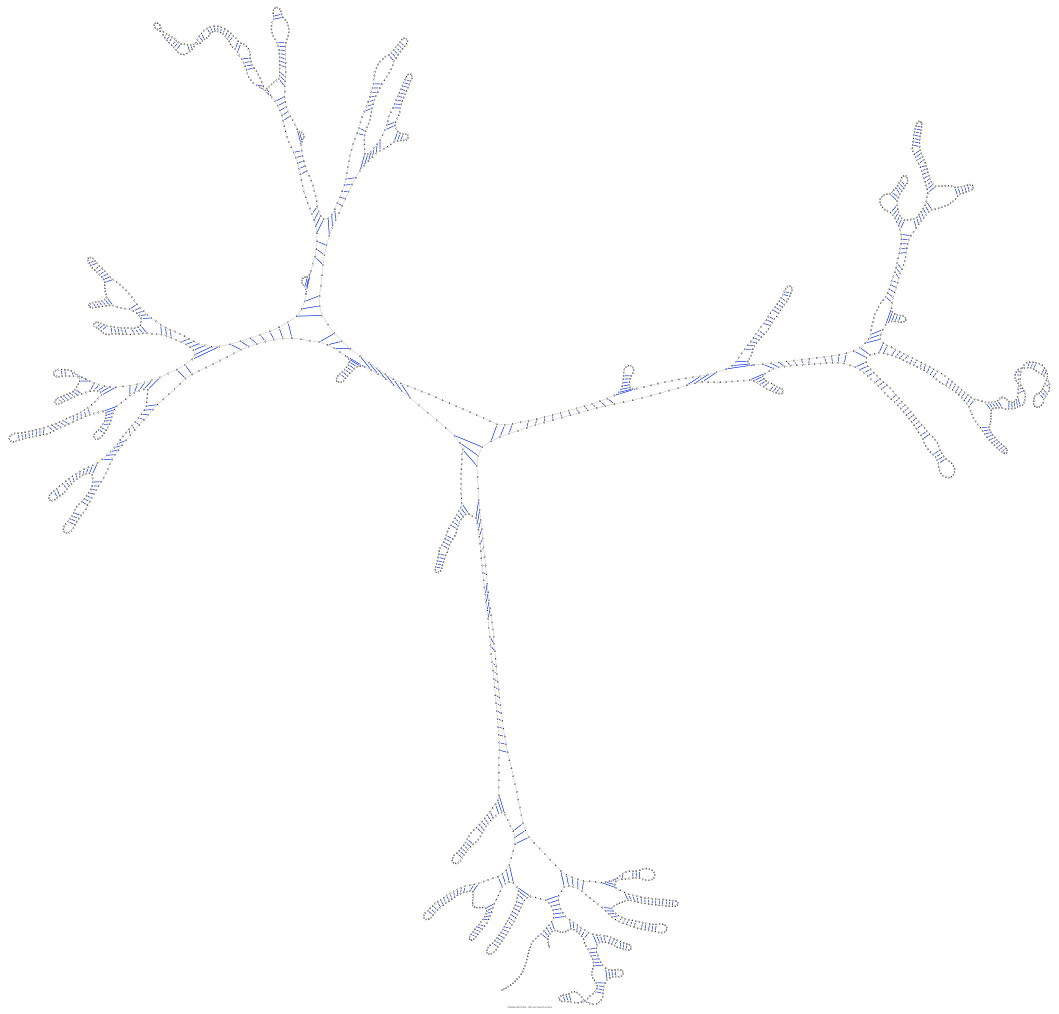


Figure A.5: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the original algorithm employed in jViz.RNA

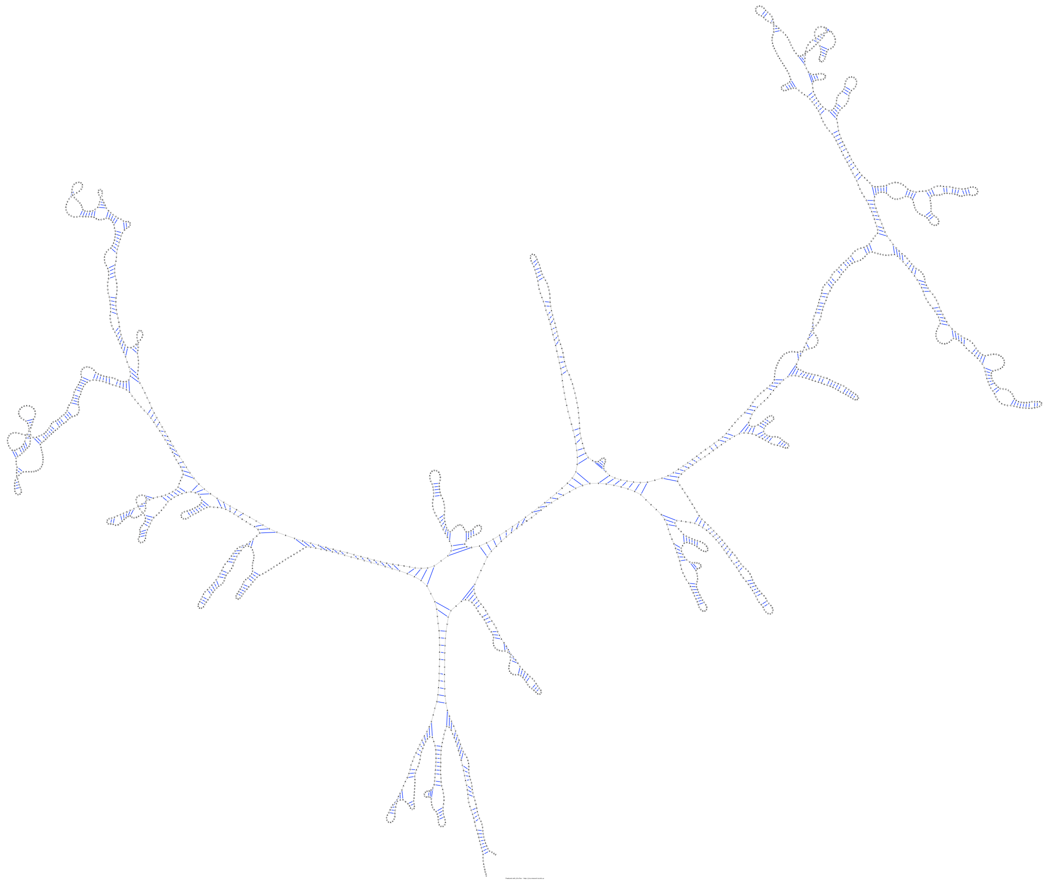


Figure A.6: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using the original algorithm employed in jViz.RNA

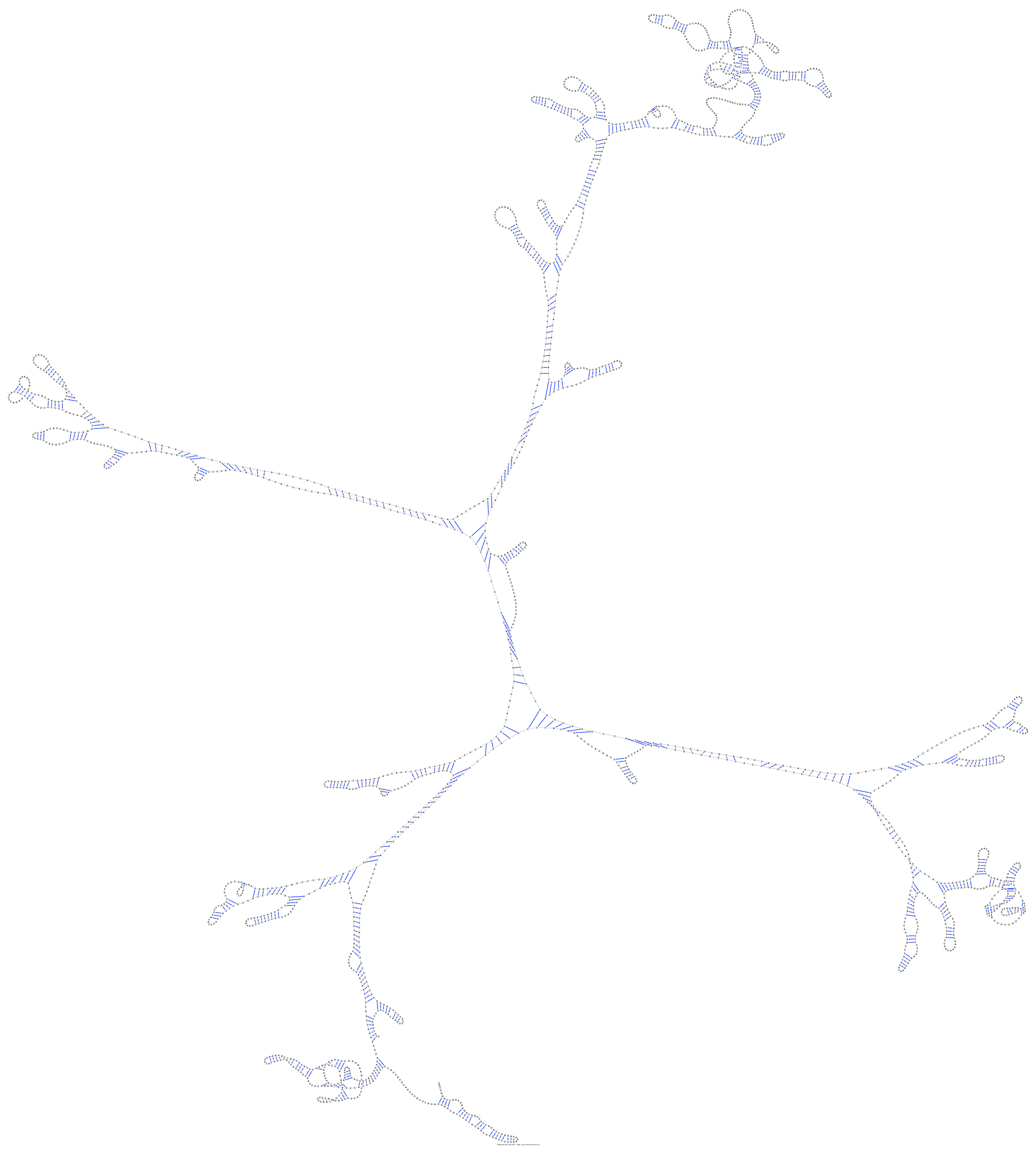


Figure A.7: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using the original algorithm employed in jViz.RNA

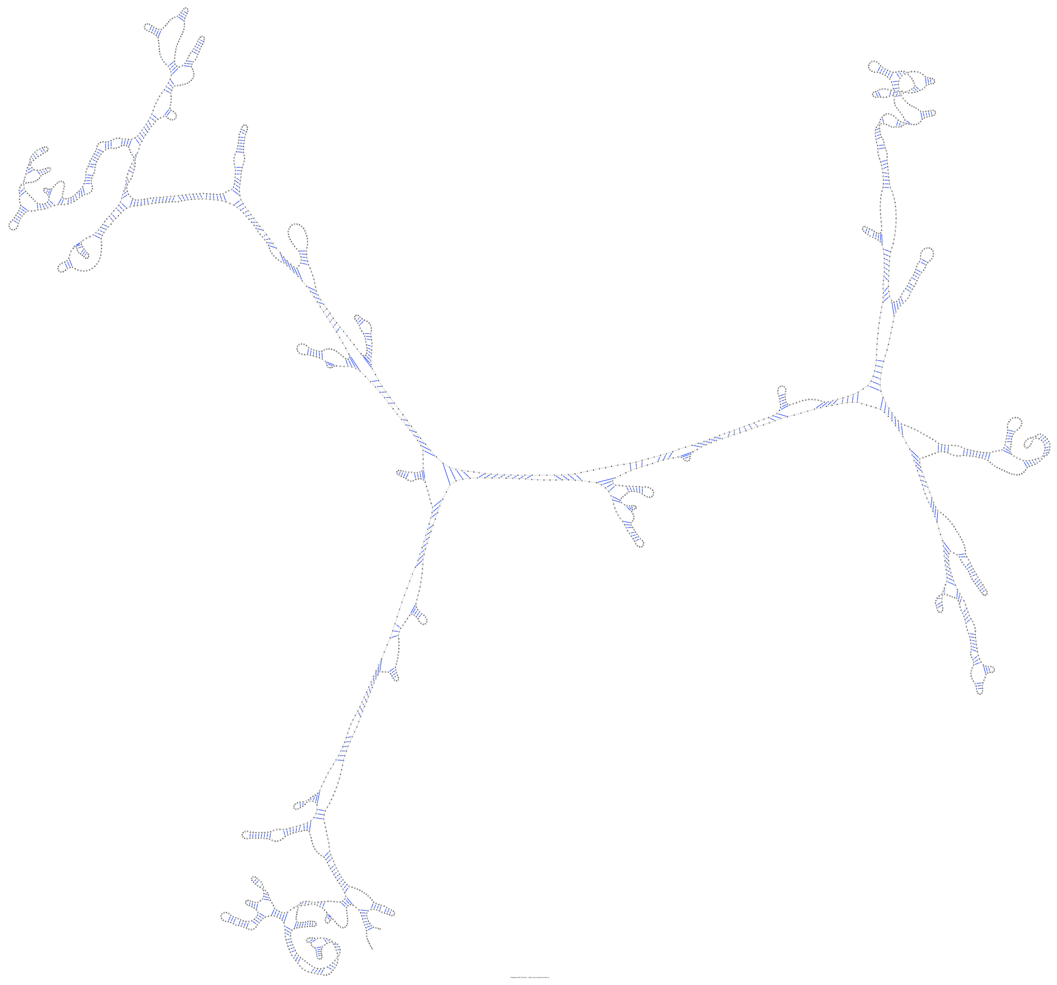


Figure A.8: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using the original algorithm employed in jViz.RNA

A.2 Sequences' layouts under the parameter optimization method

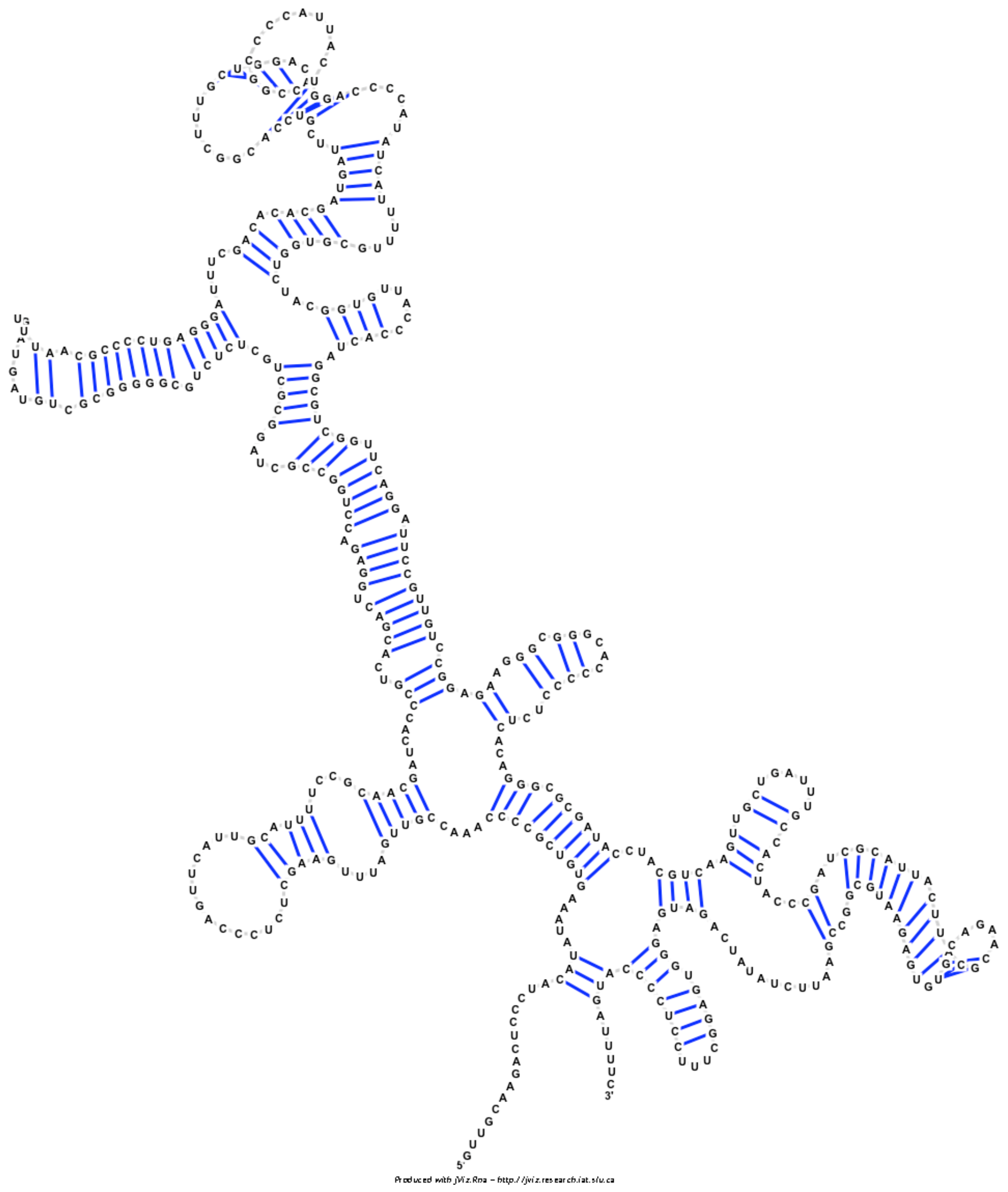


Figure A.9: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the parameter optimization method

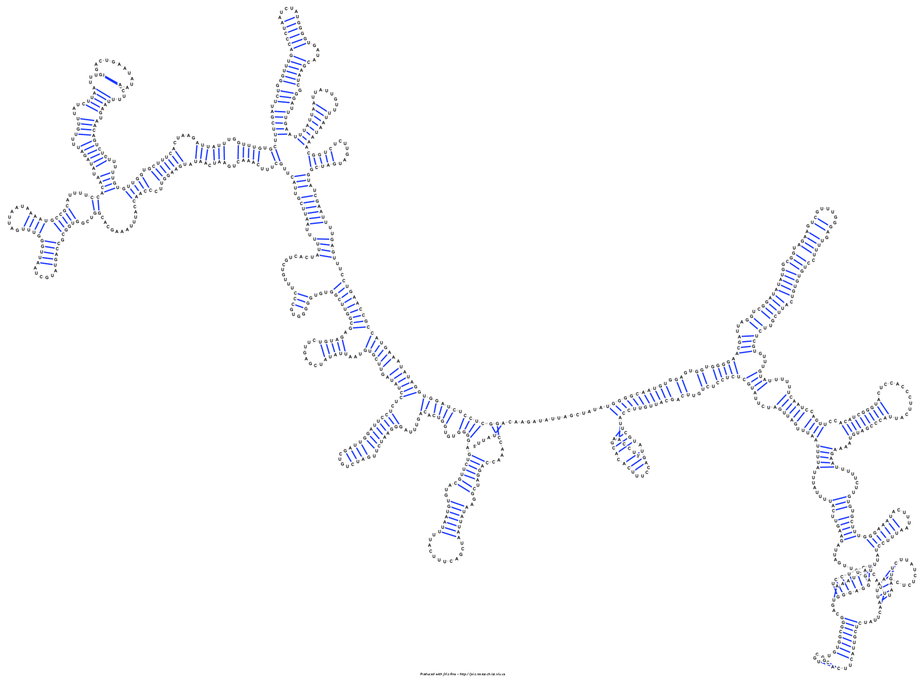


Figure A.10: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the parameter optimization method

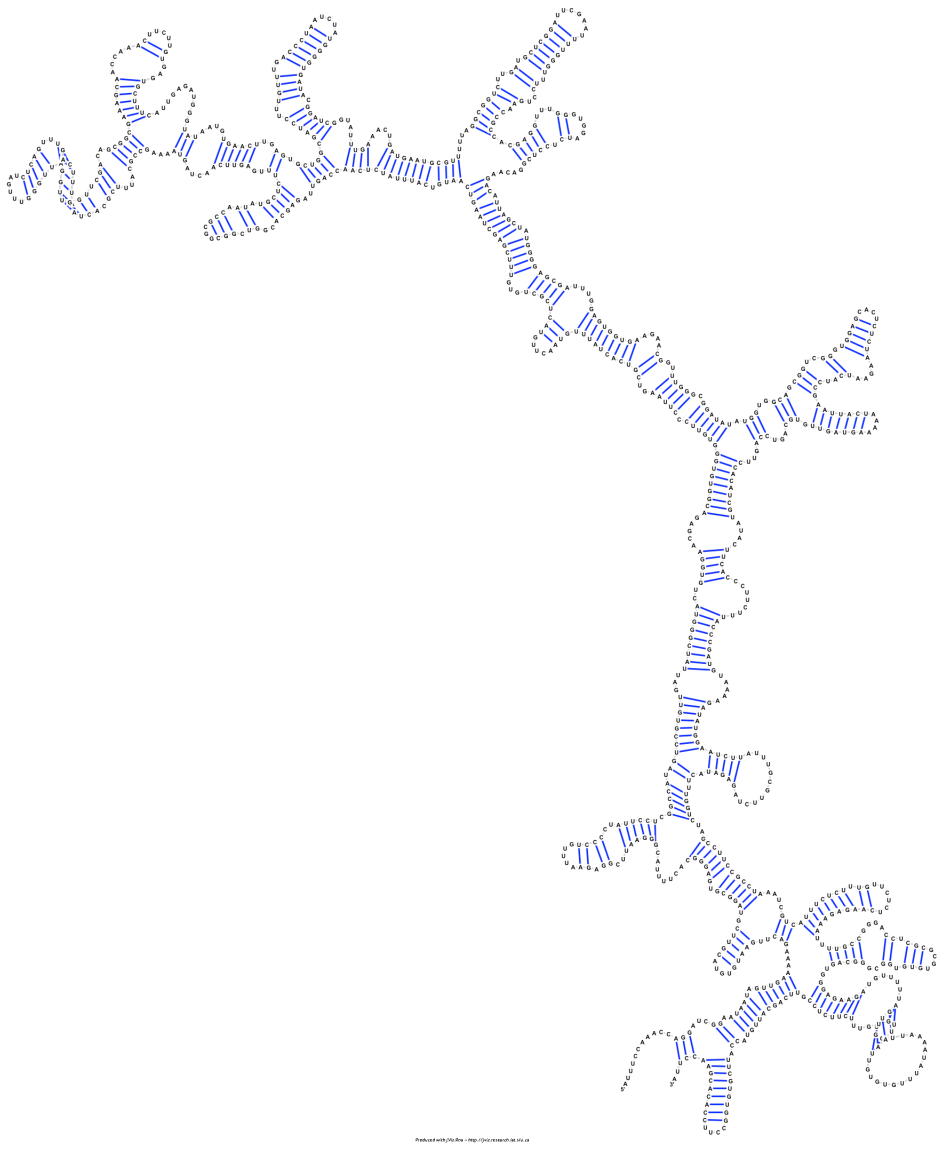


Figure A.11: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using the parameter optimization method

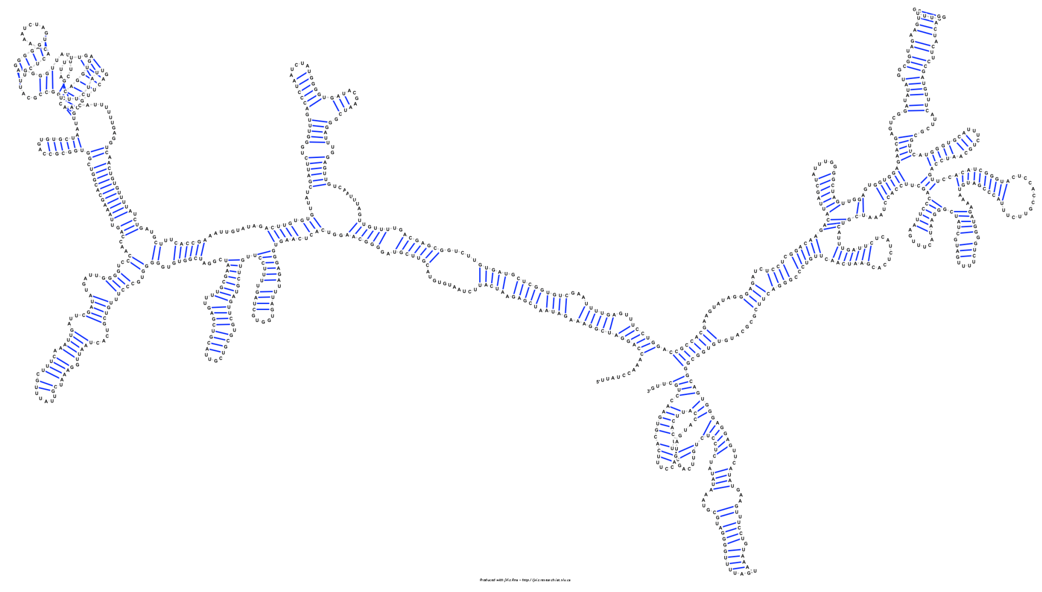


Figure A.12: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using the parameter optimization method

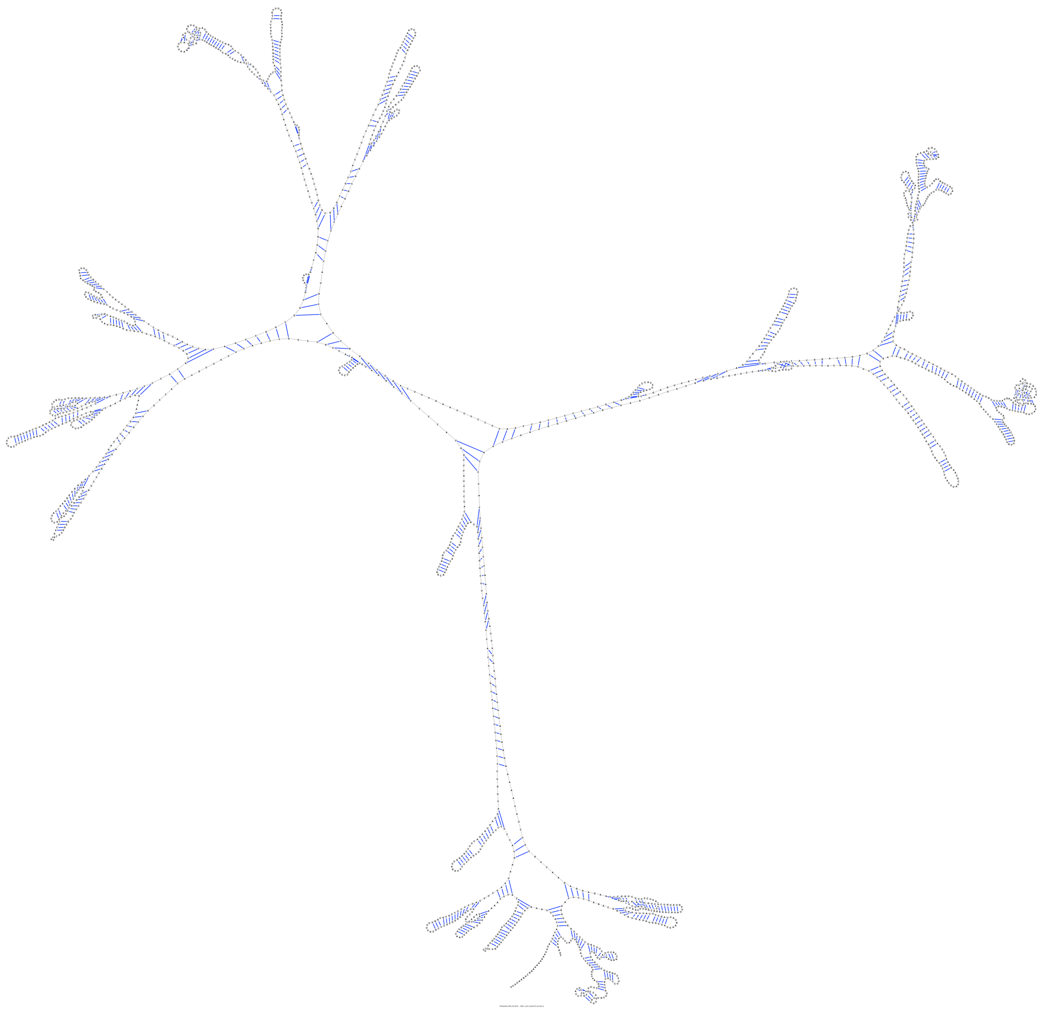


Figure A.13: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the parameter optimization method

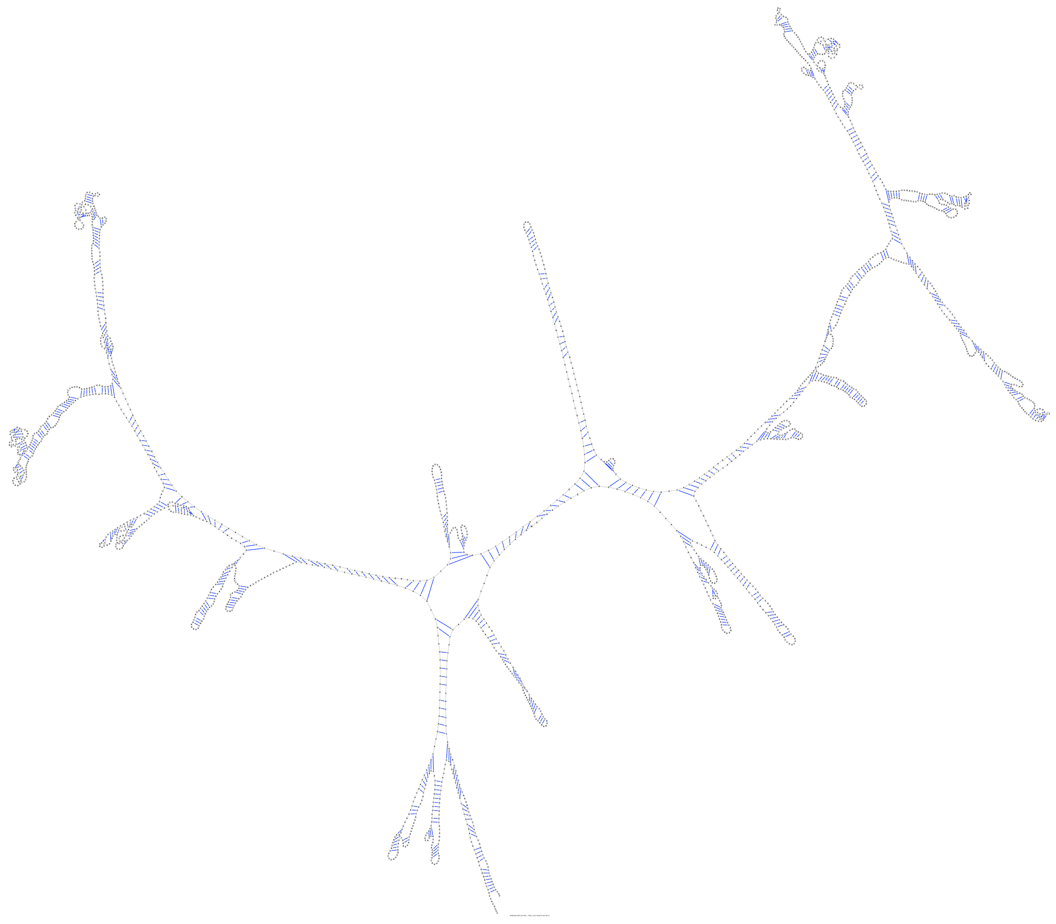


Figure A.14: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using the parameter optimization method

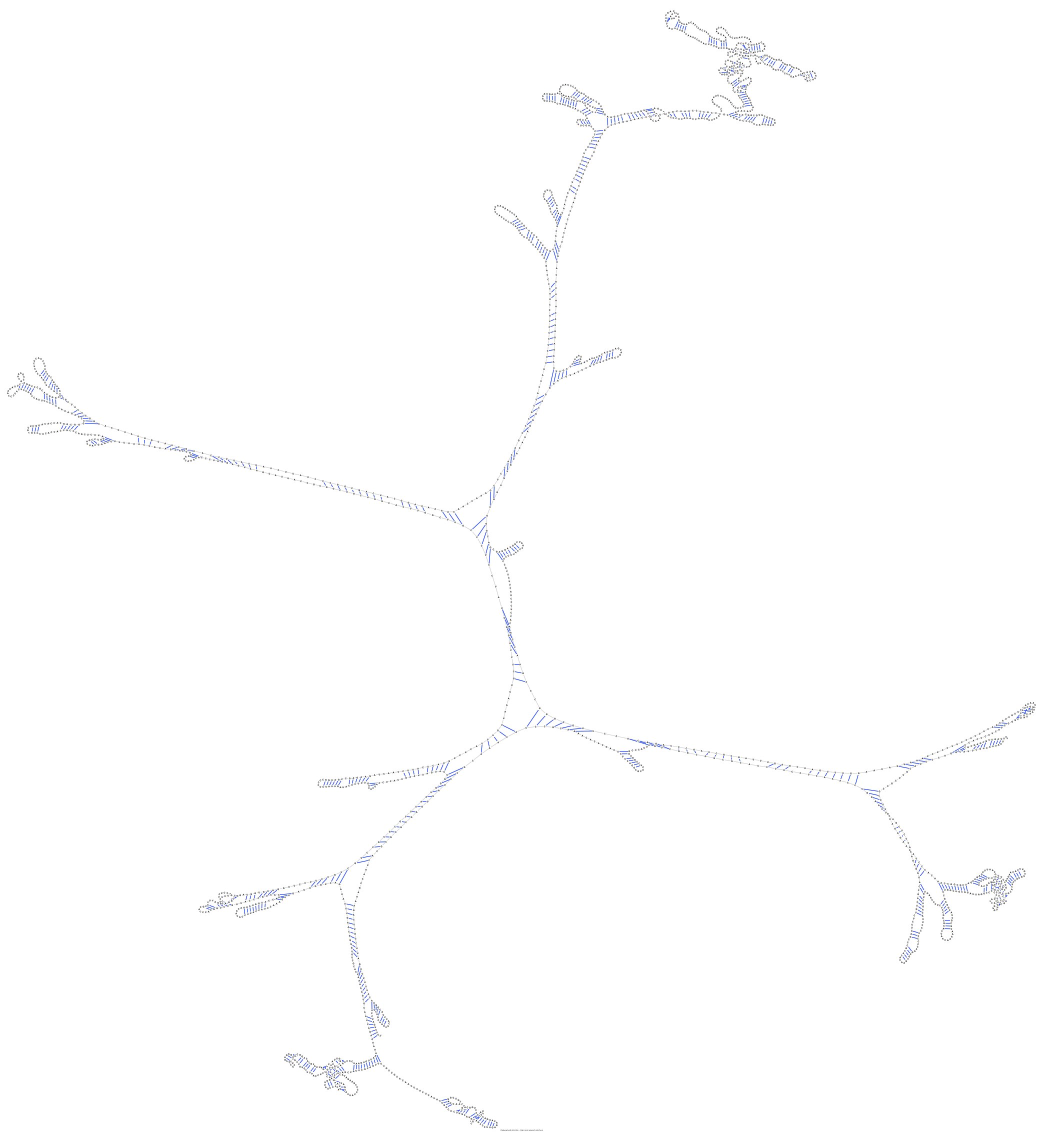


Figure A.15: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using the parameter optimization method

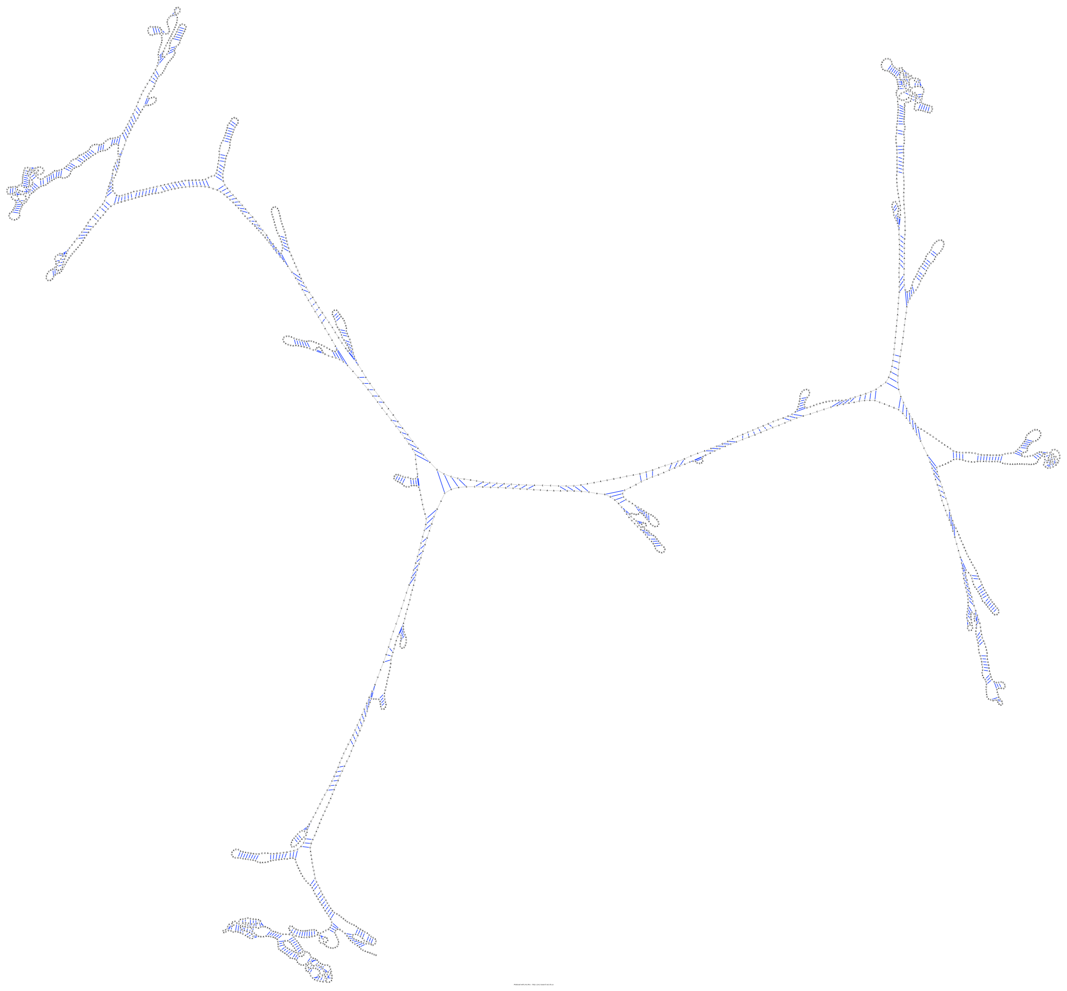


Figure A.16: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using the parameter optimization method

A.3 Sequences' layouts under the Barnes-Hut algorithm where $\theta = 1.0$

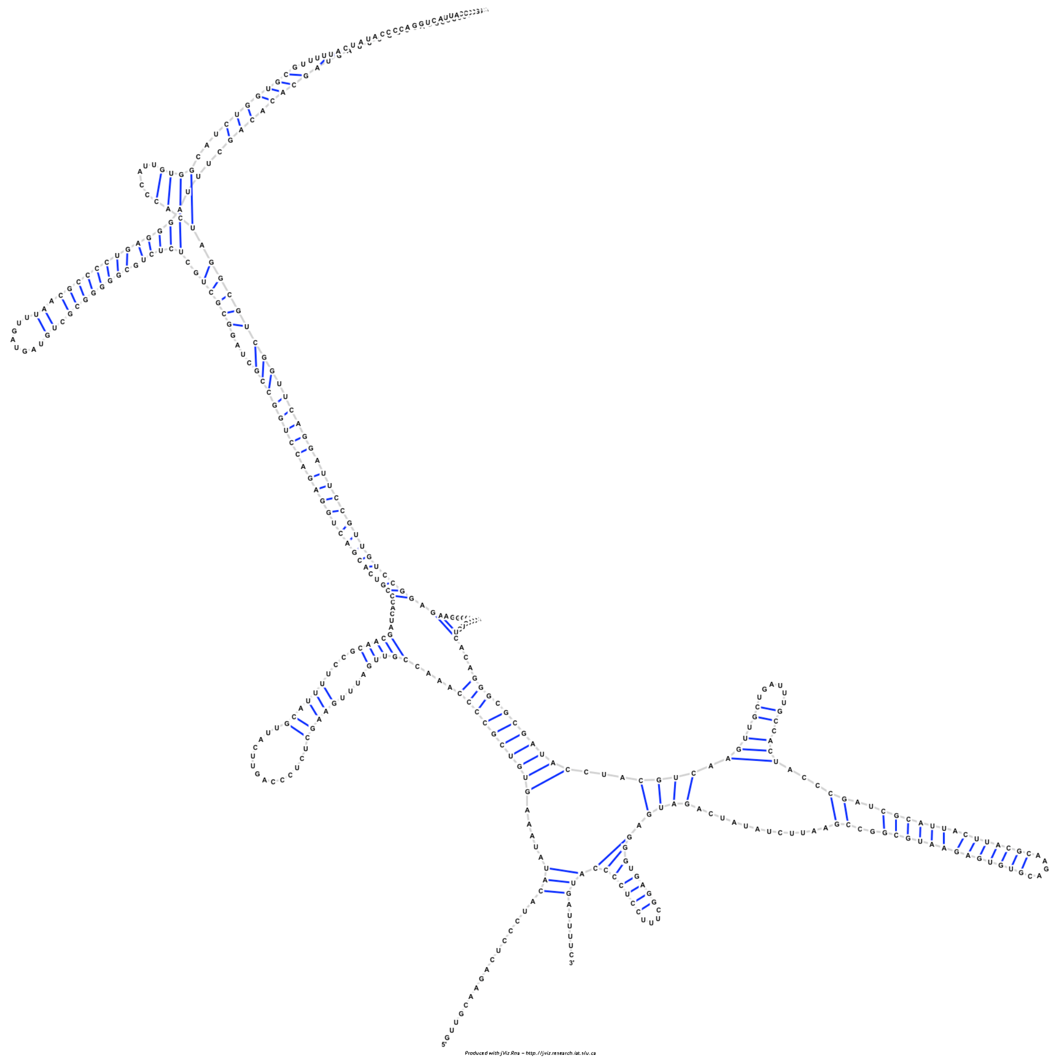


Figure A.17: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the Barnes-Hut algorithm ($\theta = 1.0$)

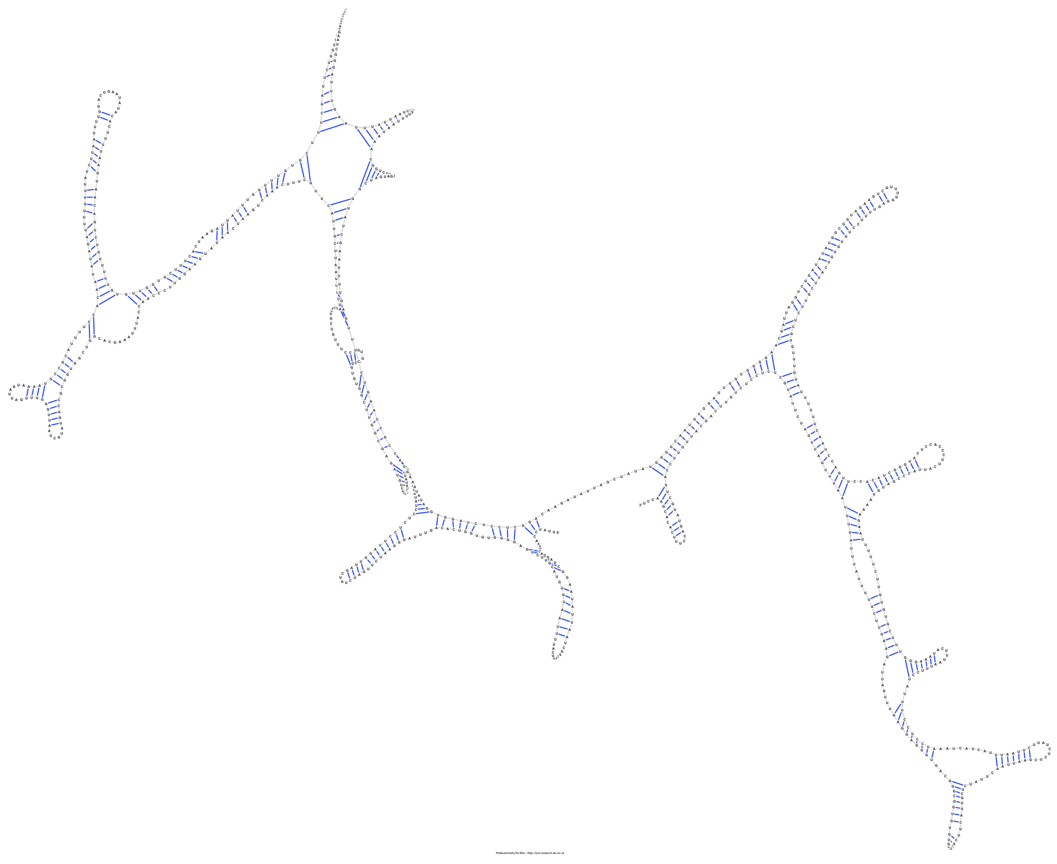


Figure A.18: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the Barnes-Hut algorithm ($\theta = 1.0$)

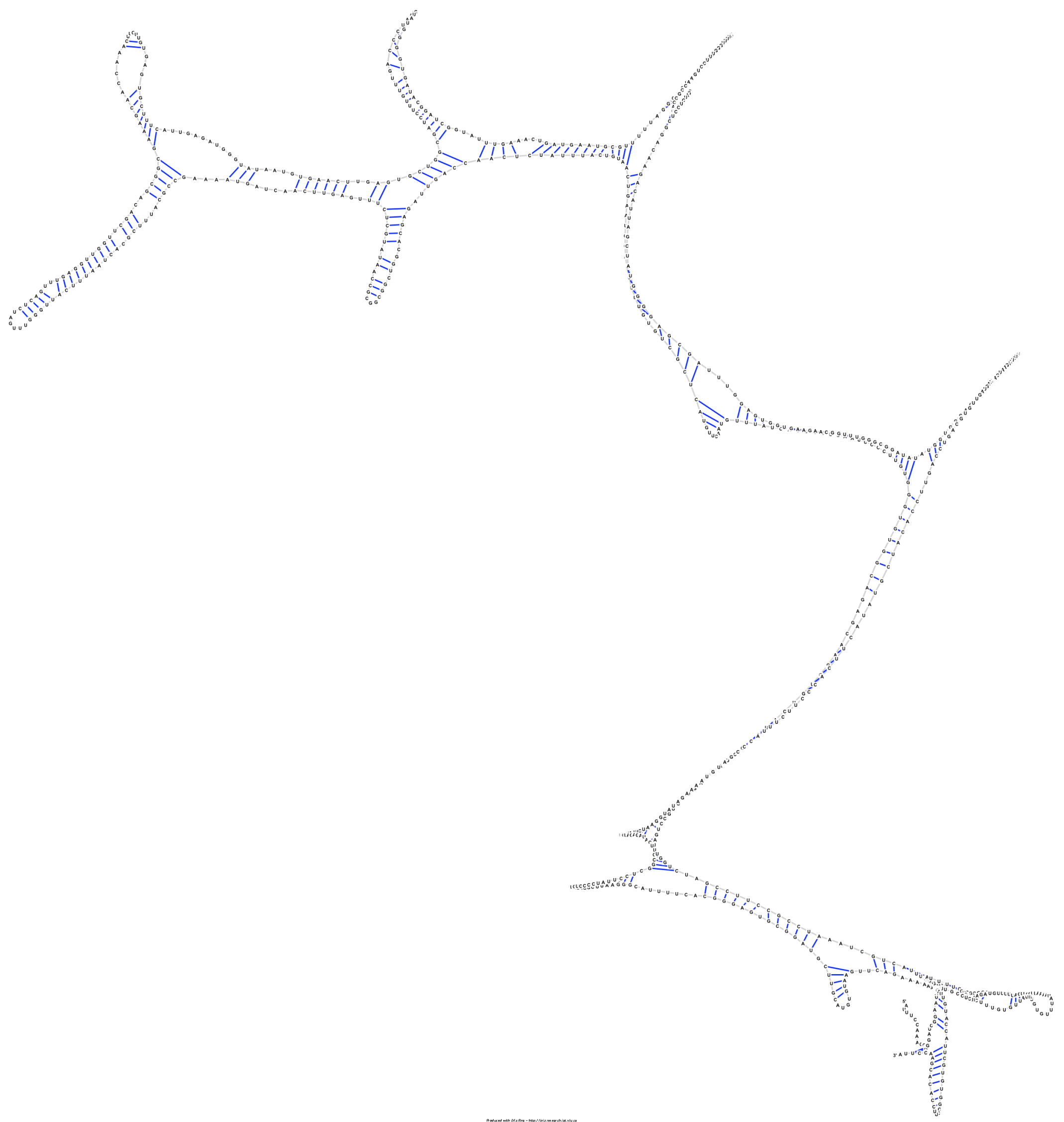


Figure A.19: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using the Barnes-Hut algorithm ($\theta = 1.0$)

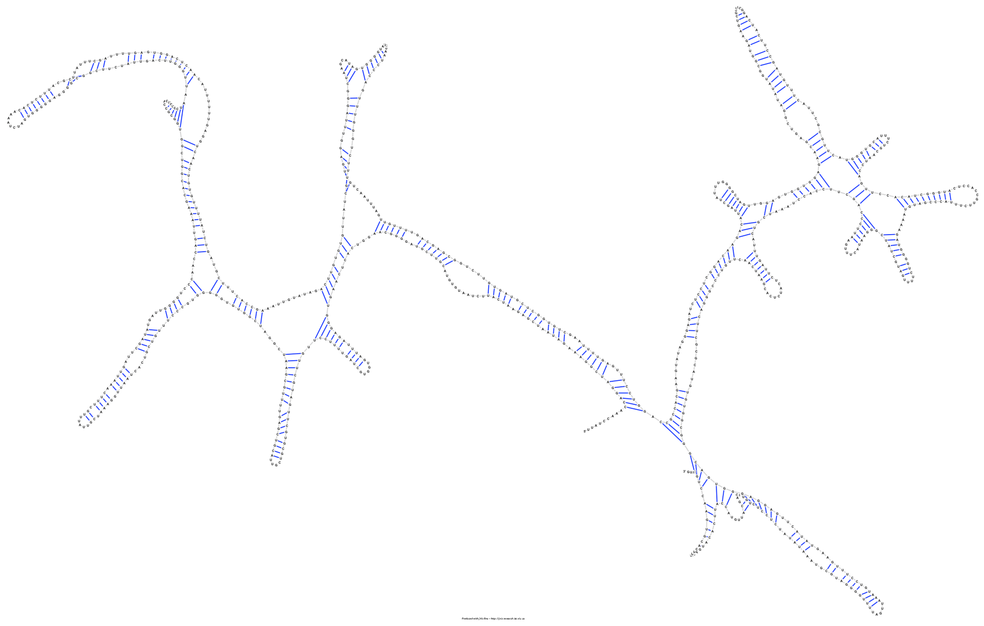


Figure A.20: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using the Barnes-Hut algorithm (θ = 1.0)

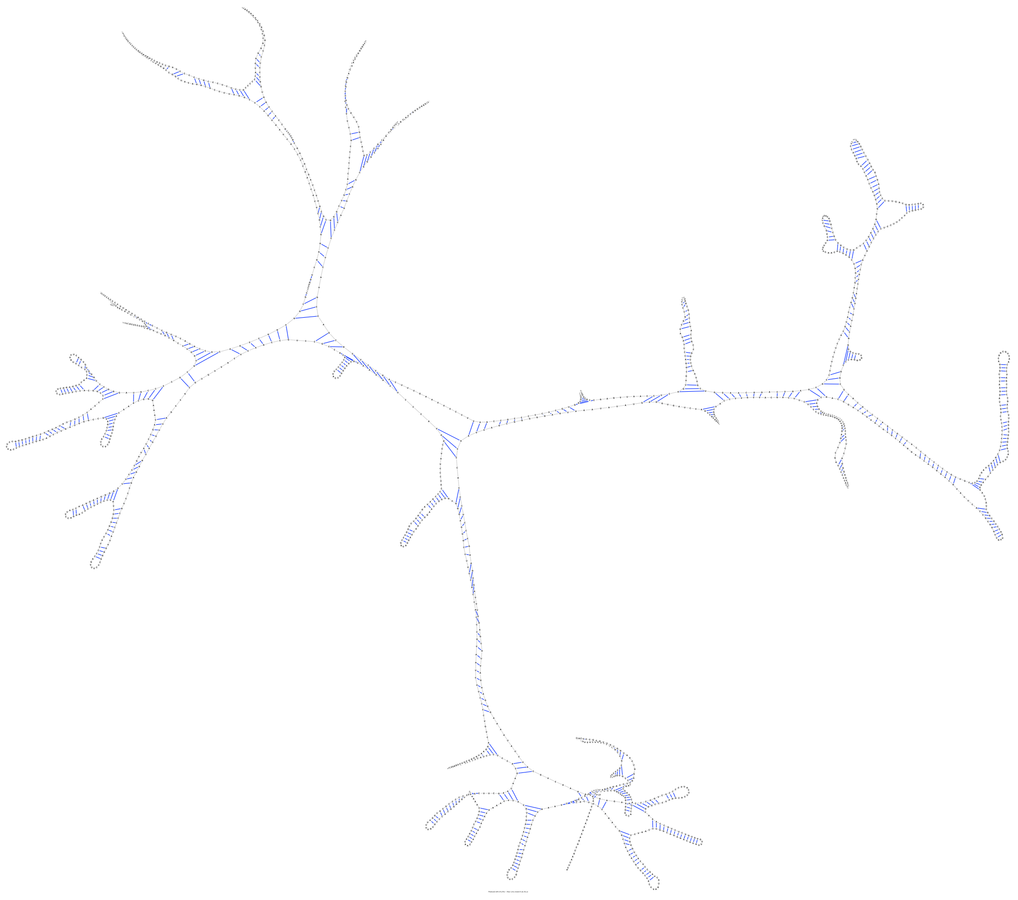


Figure A.21: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the Barnes-Hut algorithm (θ = 1.0)

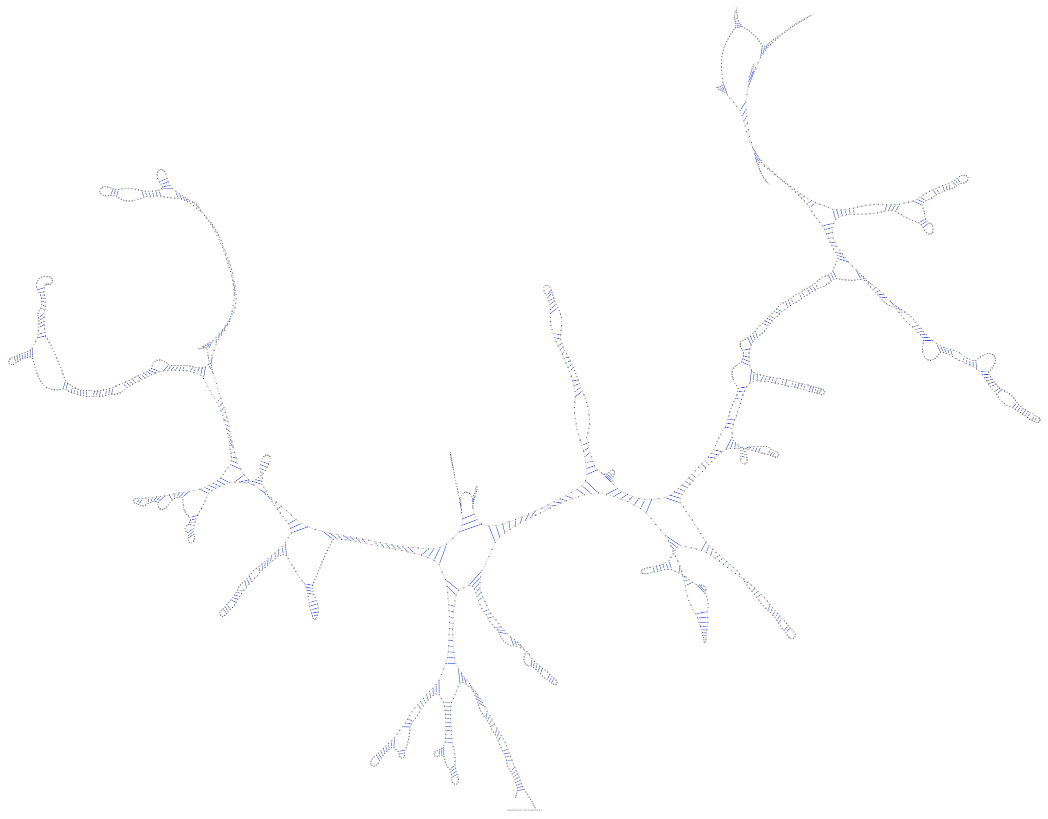


Figure A.22: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using the Barnes-Hut algorithm ($\theta = 1.0$)

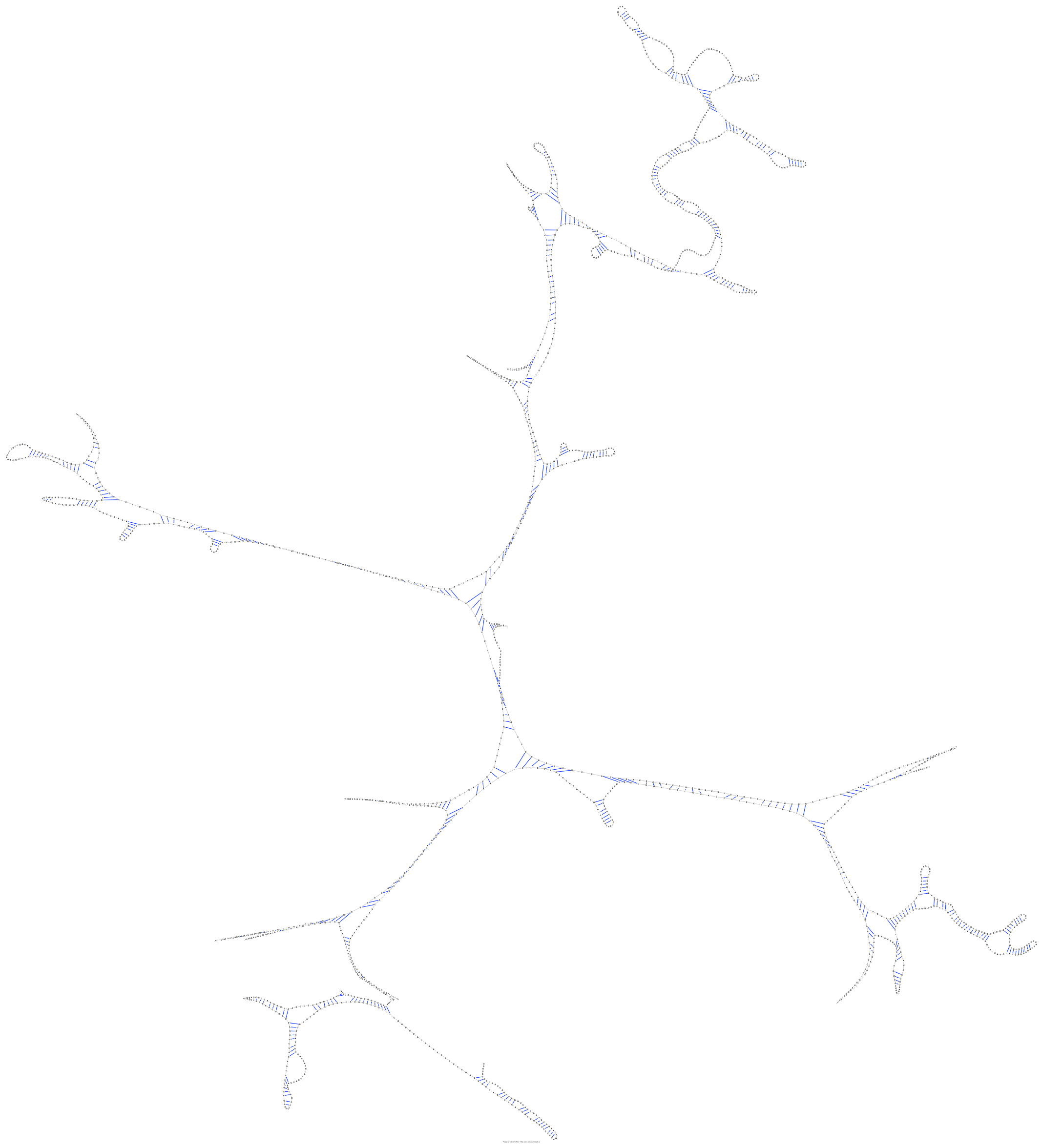


Figure A.23: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using the Barnes-Hut algorithm ($\theta = 1.0$)

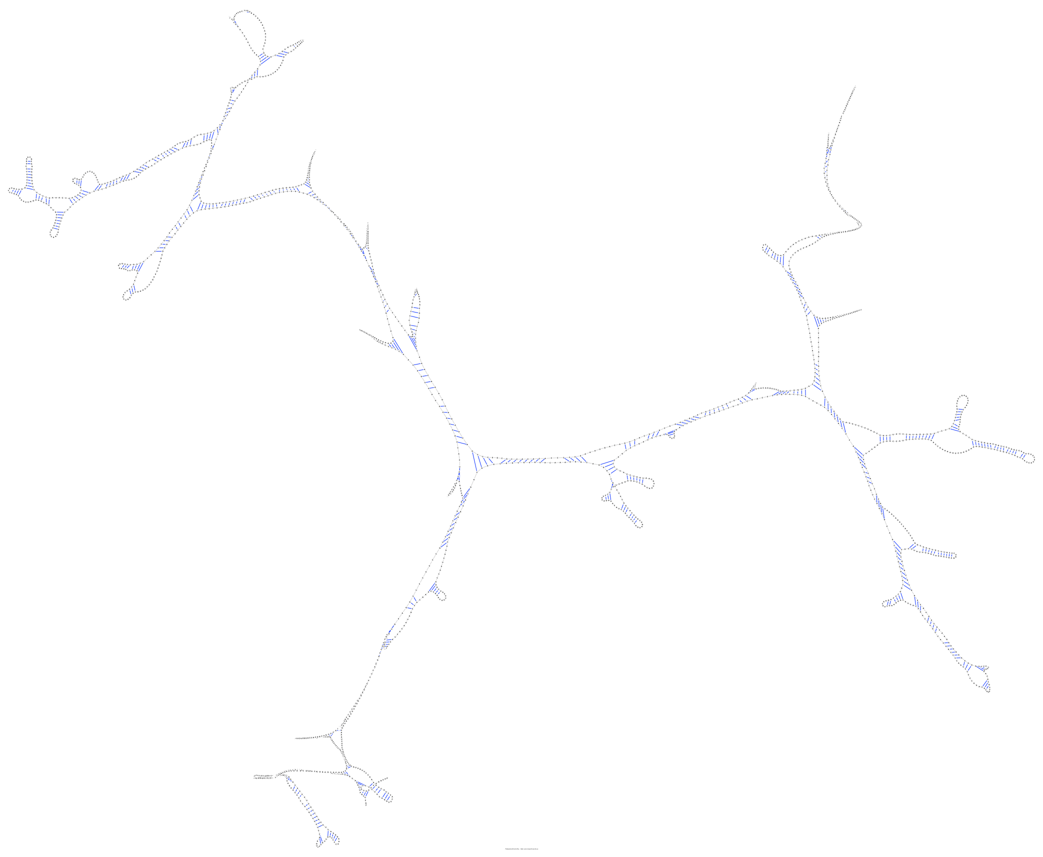


Figure A.24: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using the Barnes-Hut algorithm ($\theta = 1.0$)

A.4 Sequences' layouts under the Barnes-Hut algorithm where $\theta = 5.0$

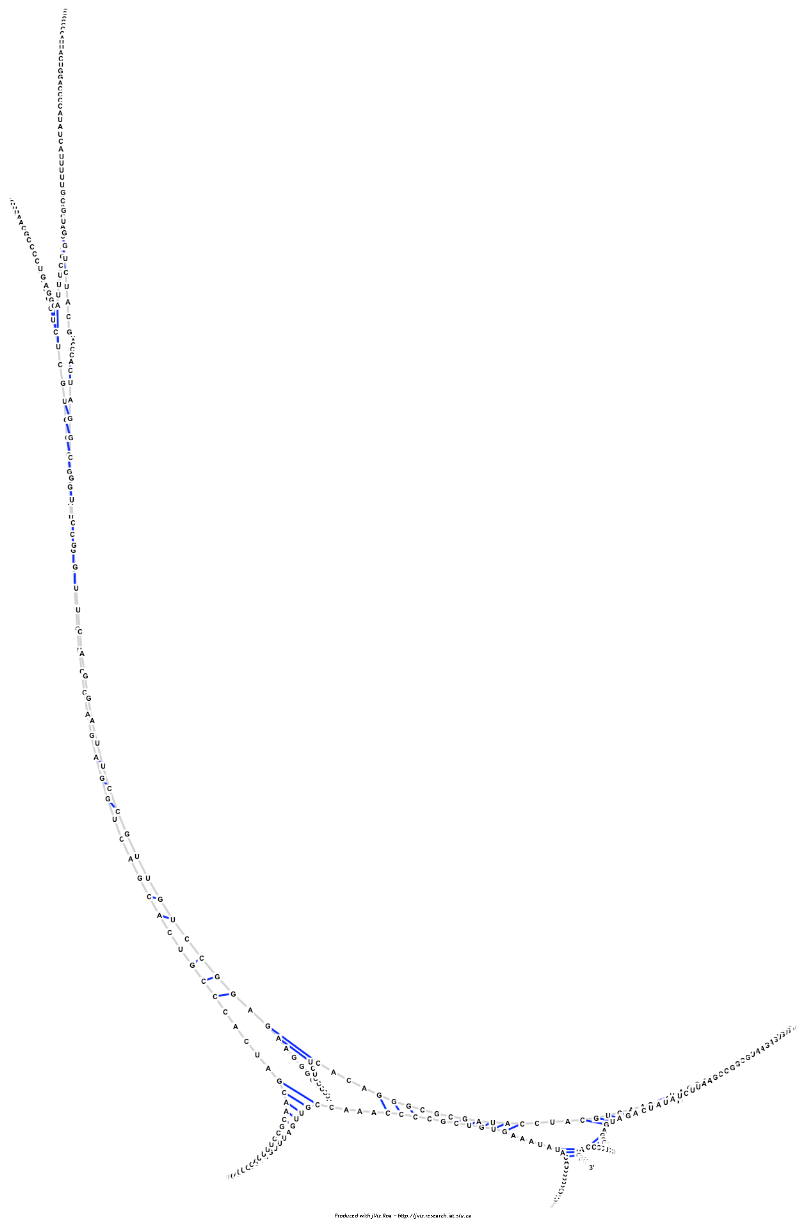


Figure A.25: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the Barnes-Hut algorithm ($\theta = 5.0$)

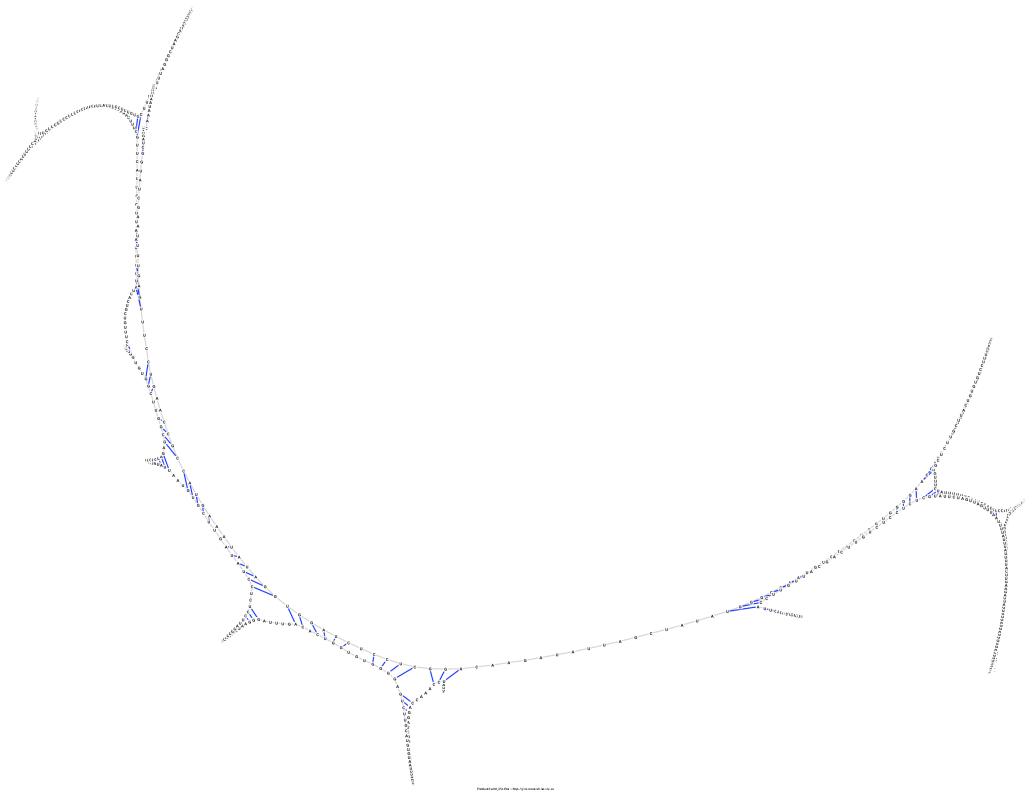


Figure A.26: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the Barnes-Hut algorithm ($\theta = 5.0$)

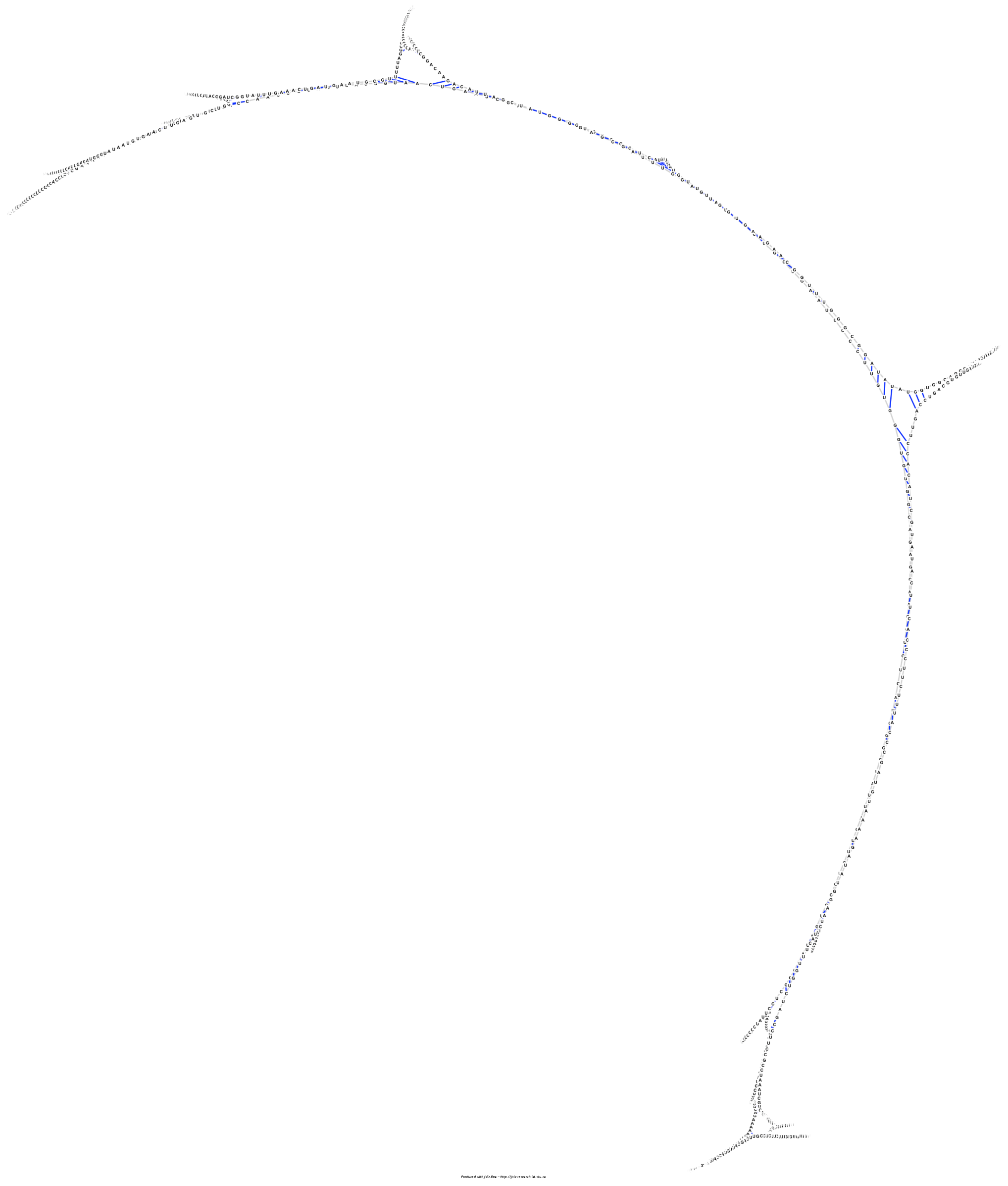


Figure A.27: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using the Barnes-Hut algorithm ($\theta = 5.0$)

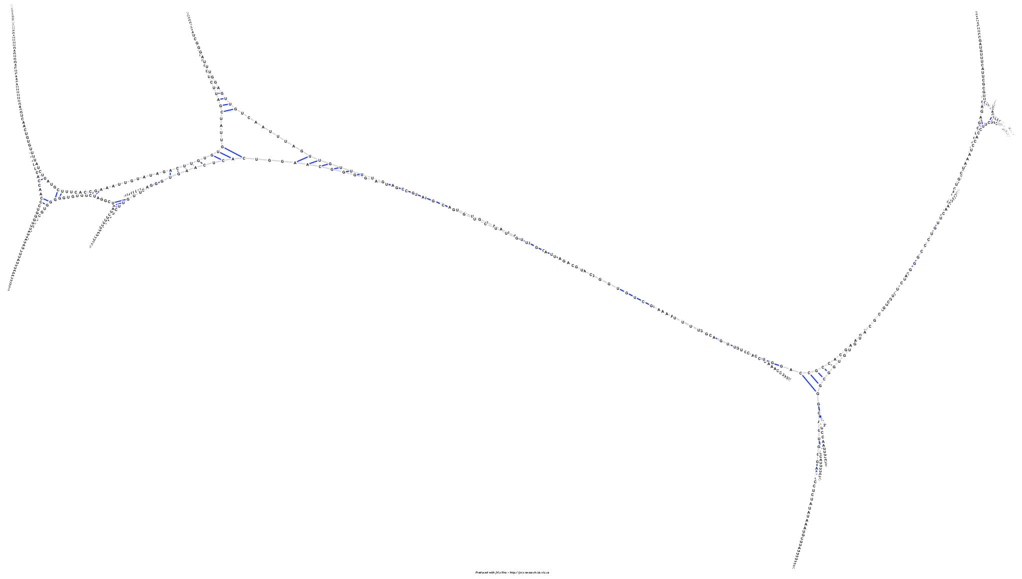


Figure A.28: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using the Barnes-Hut algorithm (θ = 5.0)

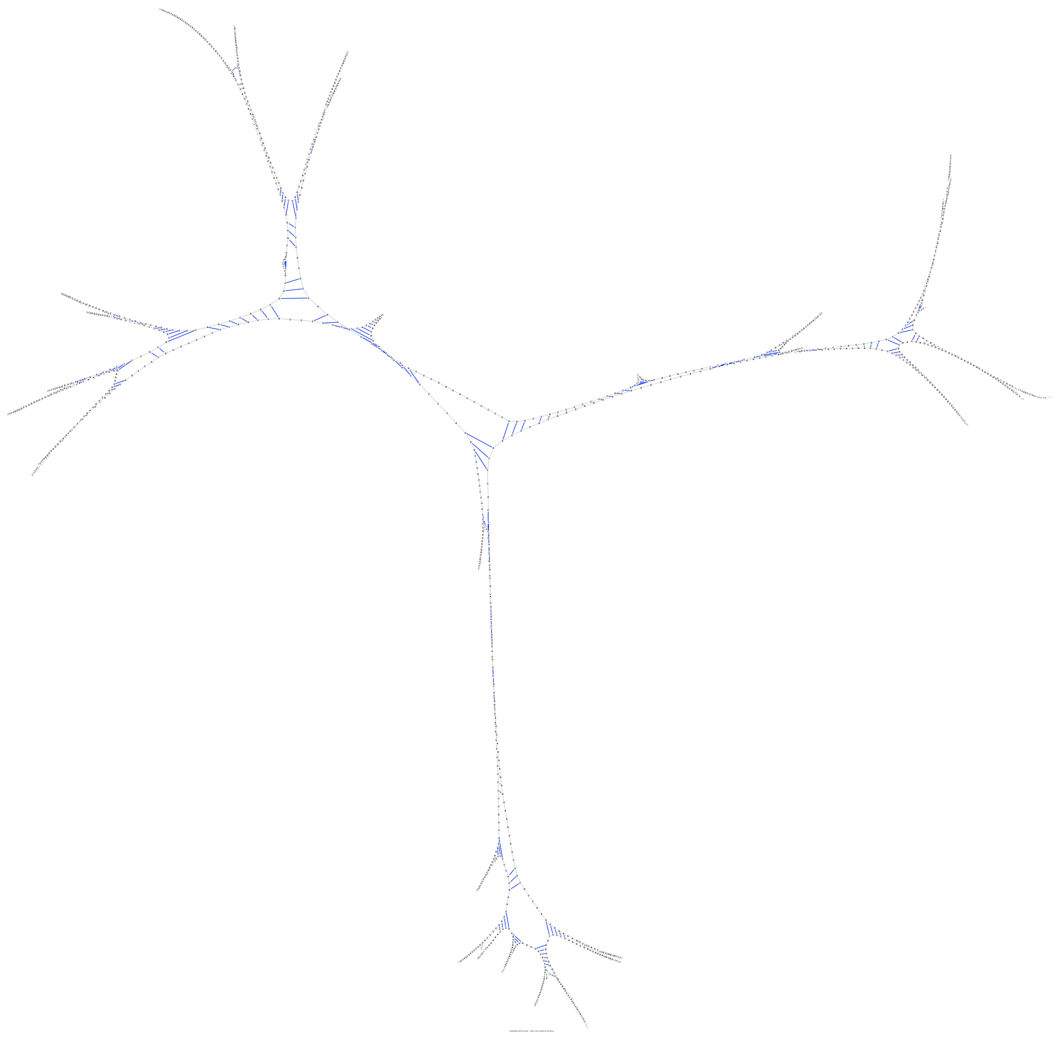


Figure A.29: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the Barnes-Hut algorithm ($\theta = 5.0$)

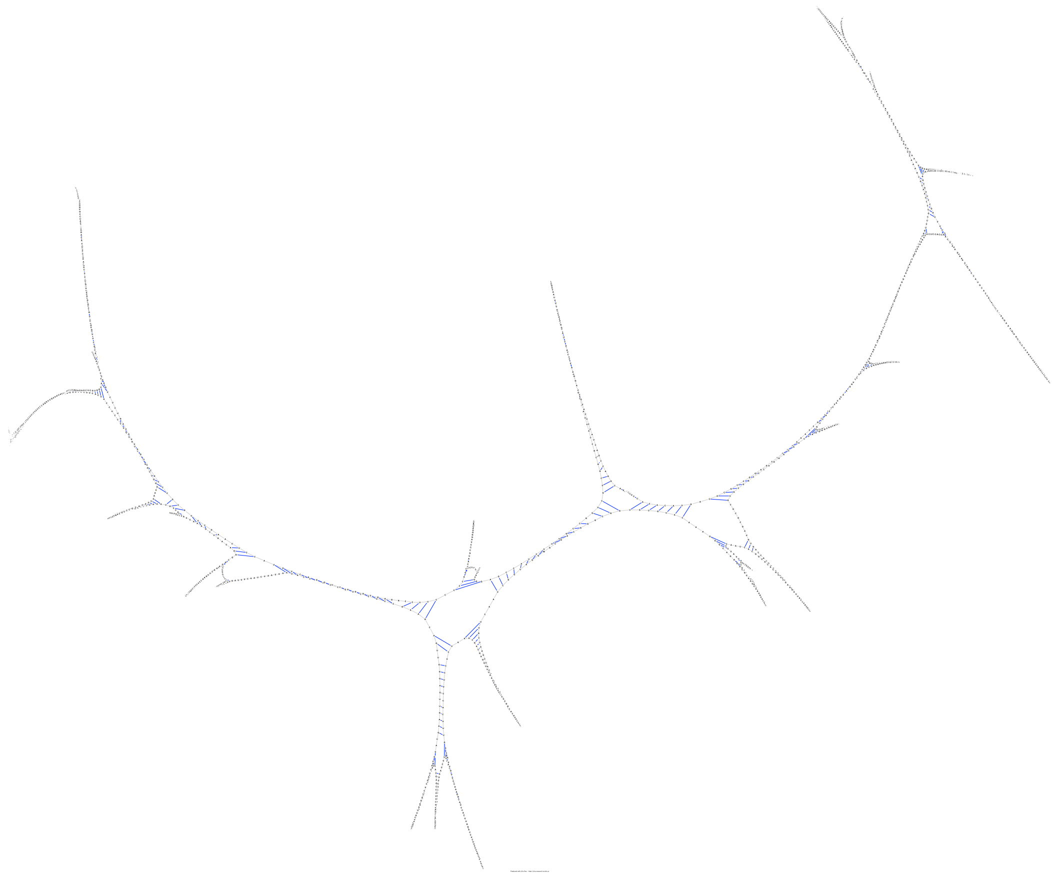


Figure A.30: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using the Barnes-Hut algorithm ($\theta = 5.0$)

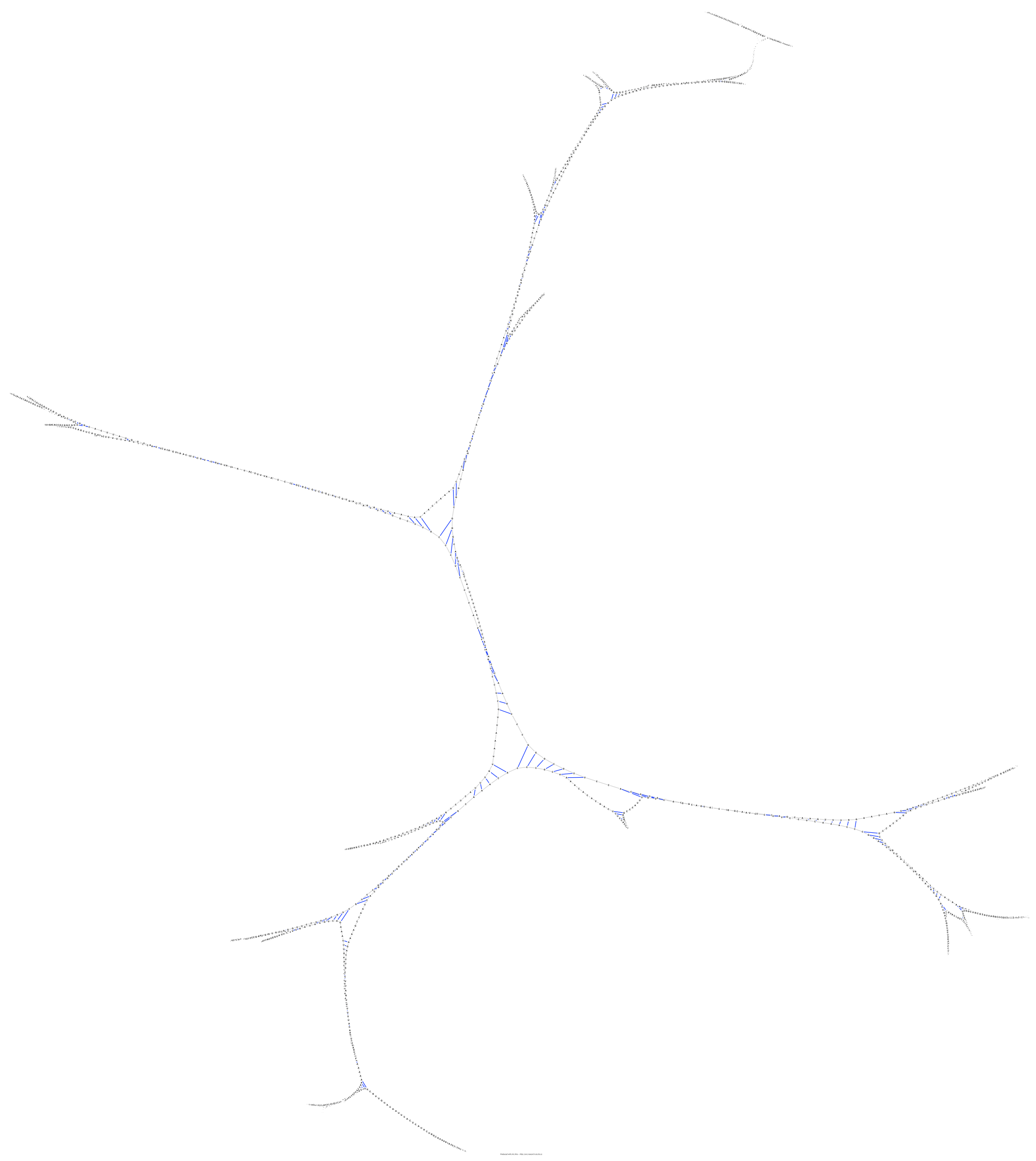


Figure A.31: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using the Barnes-Hut algorithm ($\theta = 5.0$)

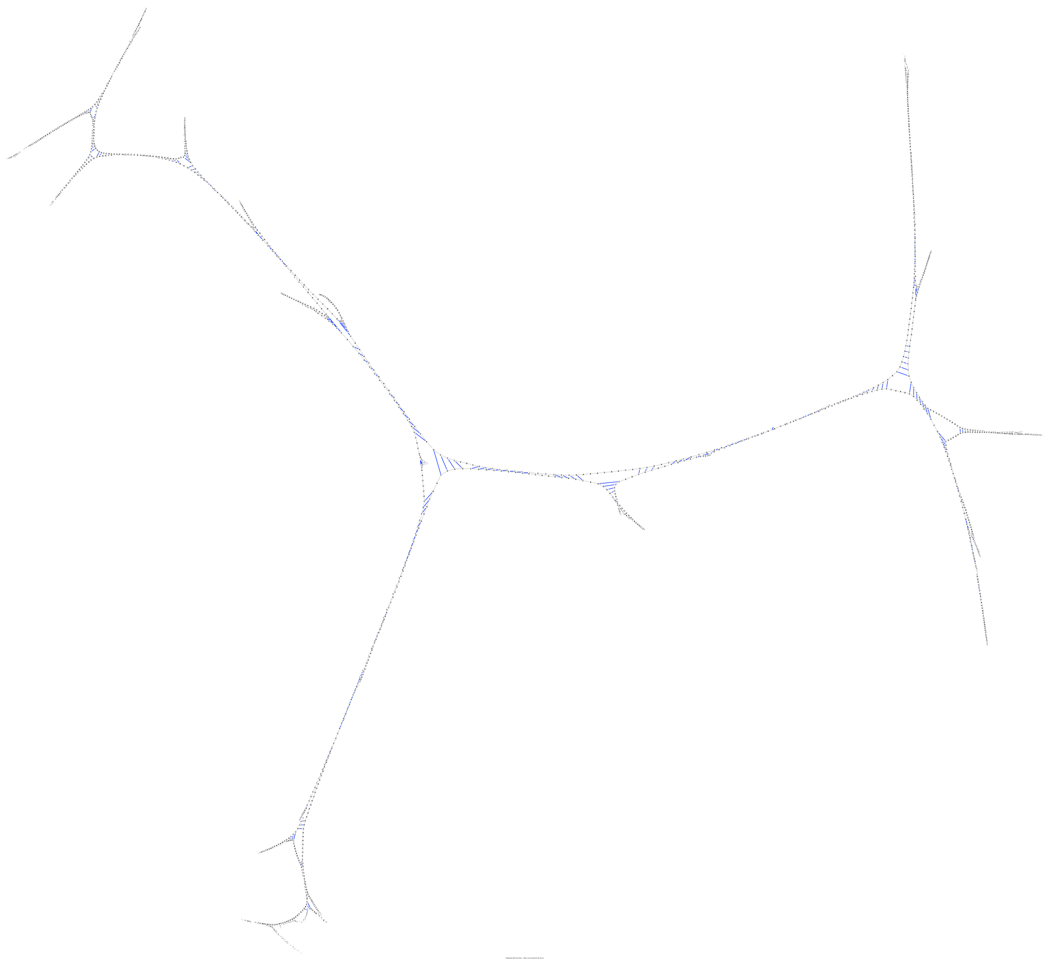


Figure A.32: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using the Barnes-Hut algorithm ($\theta = 5.0$)

A.5 Sequences' layouts under the multithreading approach where thread number = 5

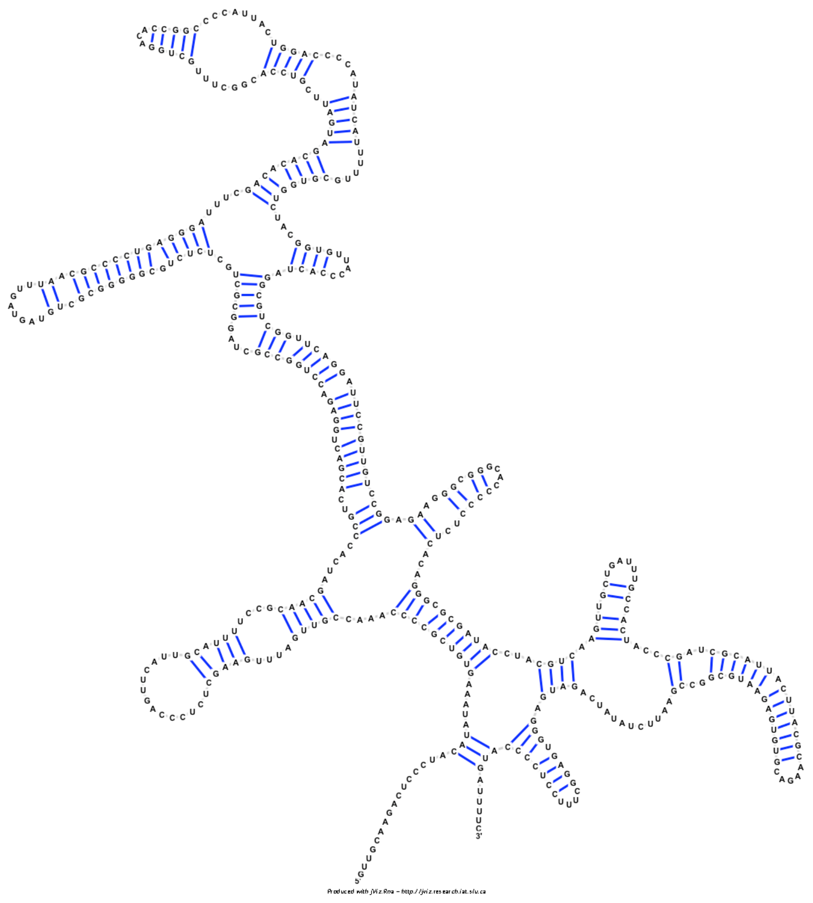


Figure A.33: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using the multithreading approach (thread number = 5)

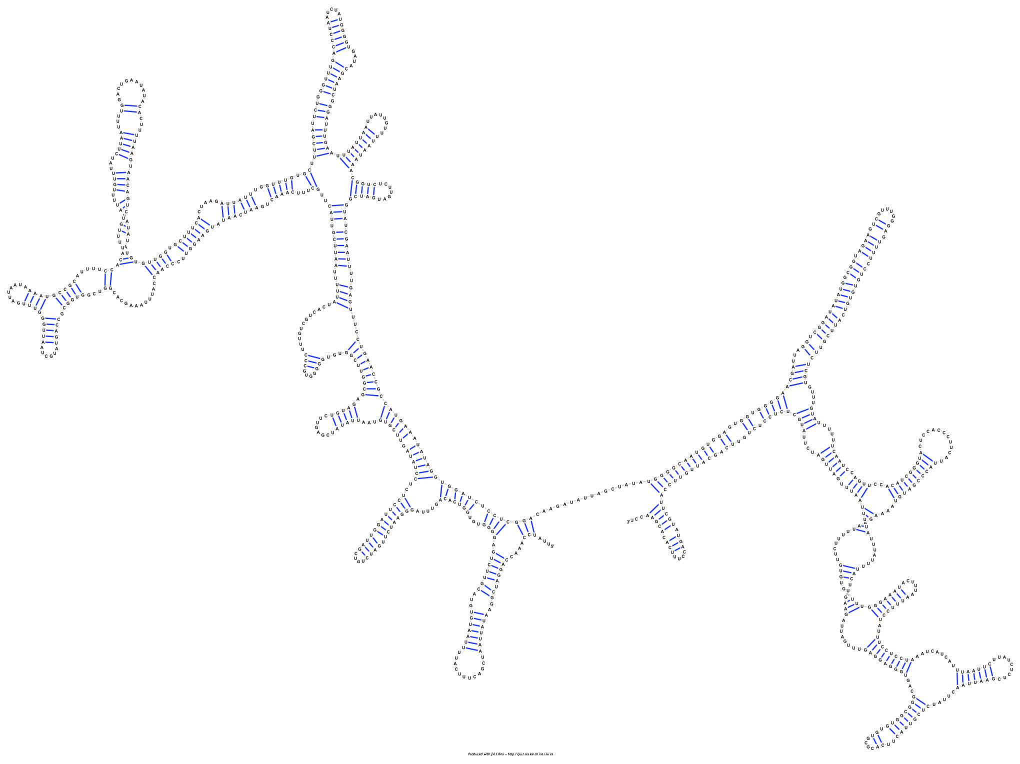


Figure A.34: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using the multithreading approach (thread number = 5)

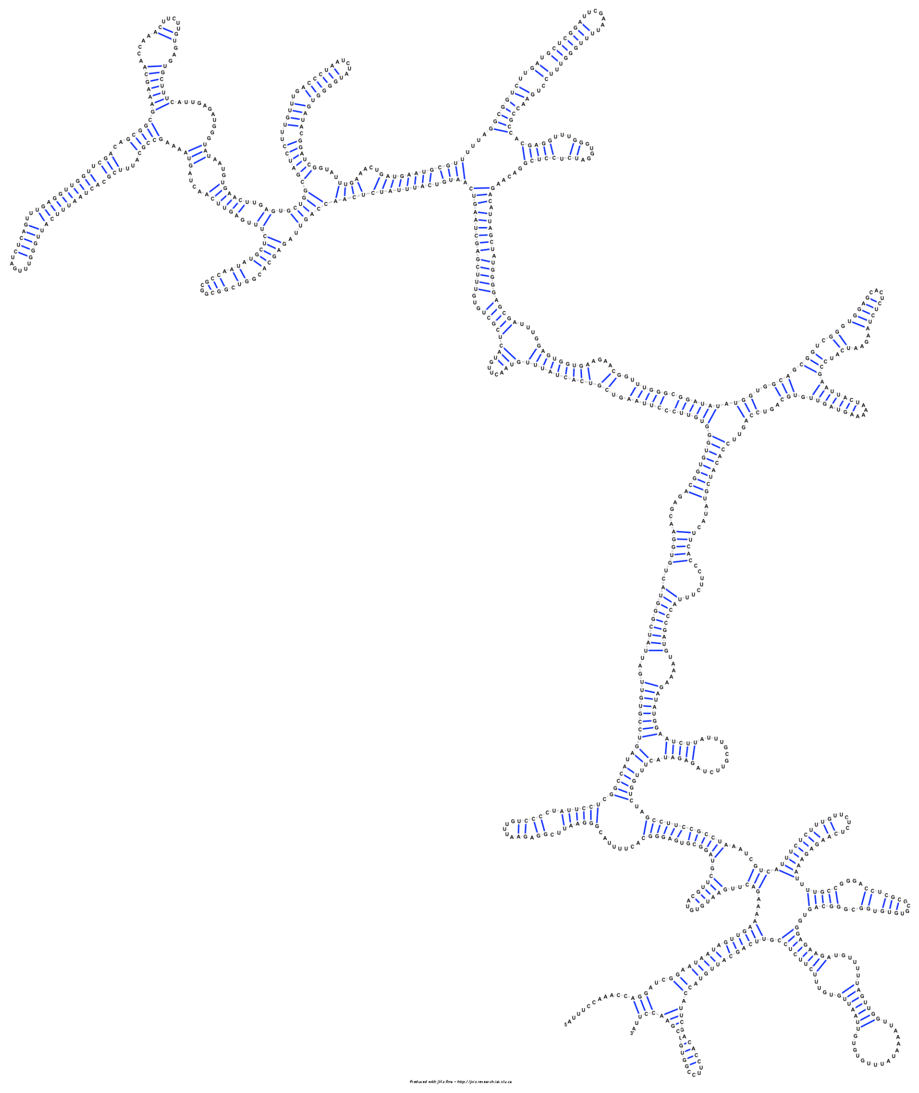


Figure A.35: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using the multithreading approach (thread number = 5)

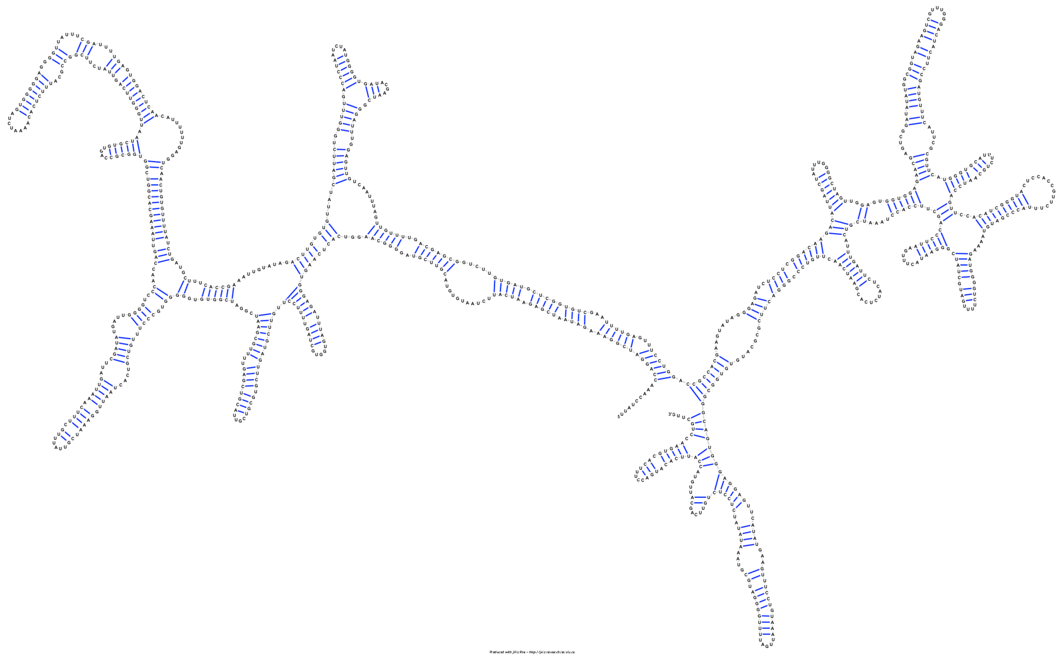


Figure A.36: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using the multithreading approach (thread number = 5)

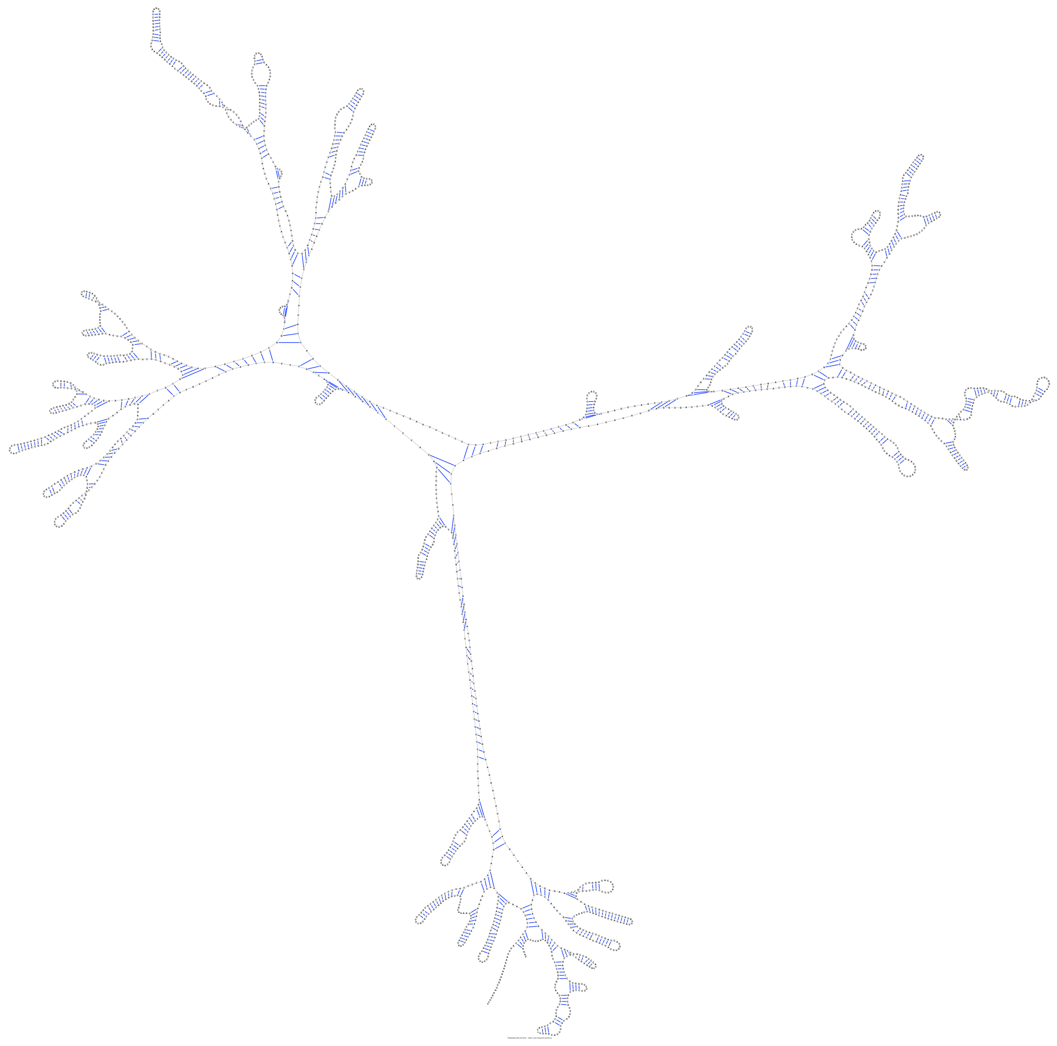


Figure A.37: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using the multithreading approach (thread number = 5)

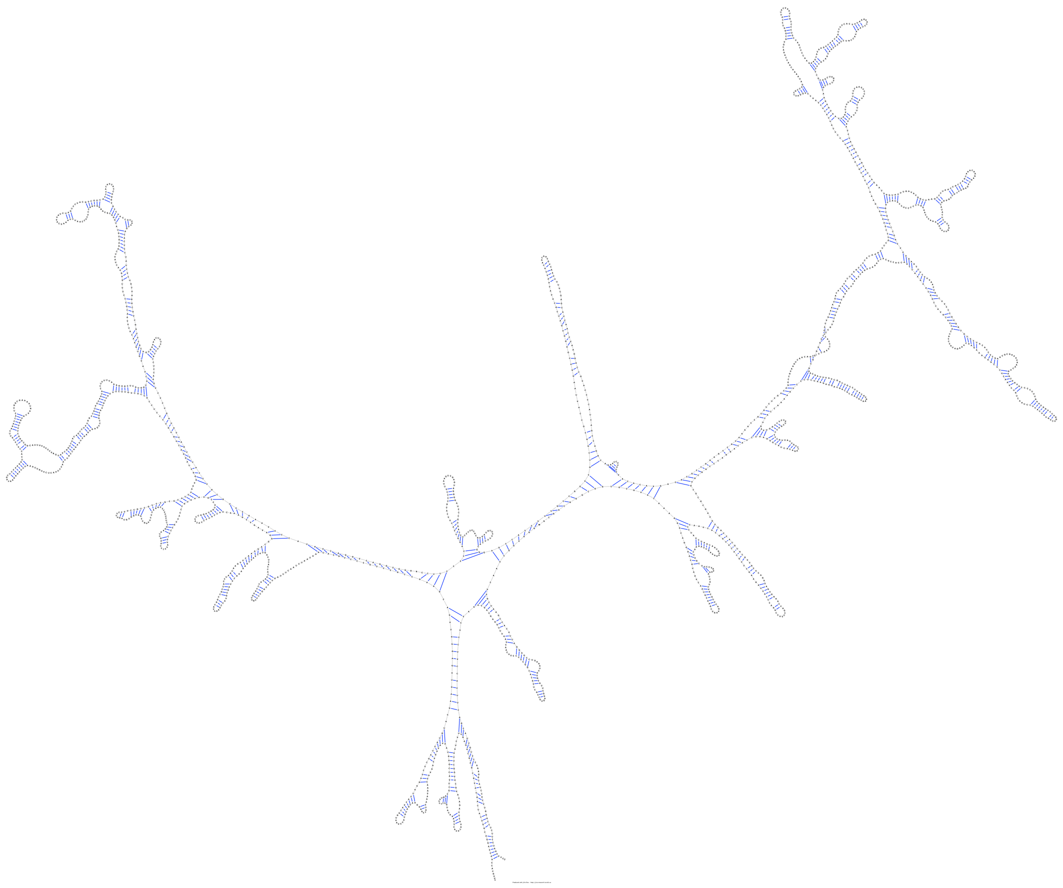


Figure A.38: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using the multithreading approach (thread number = 5)

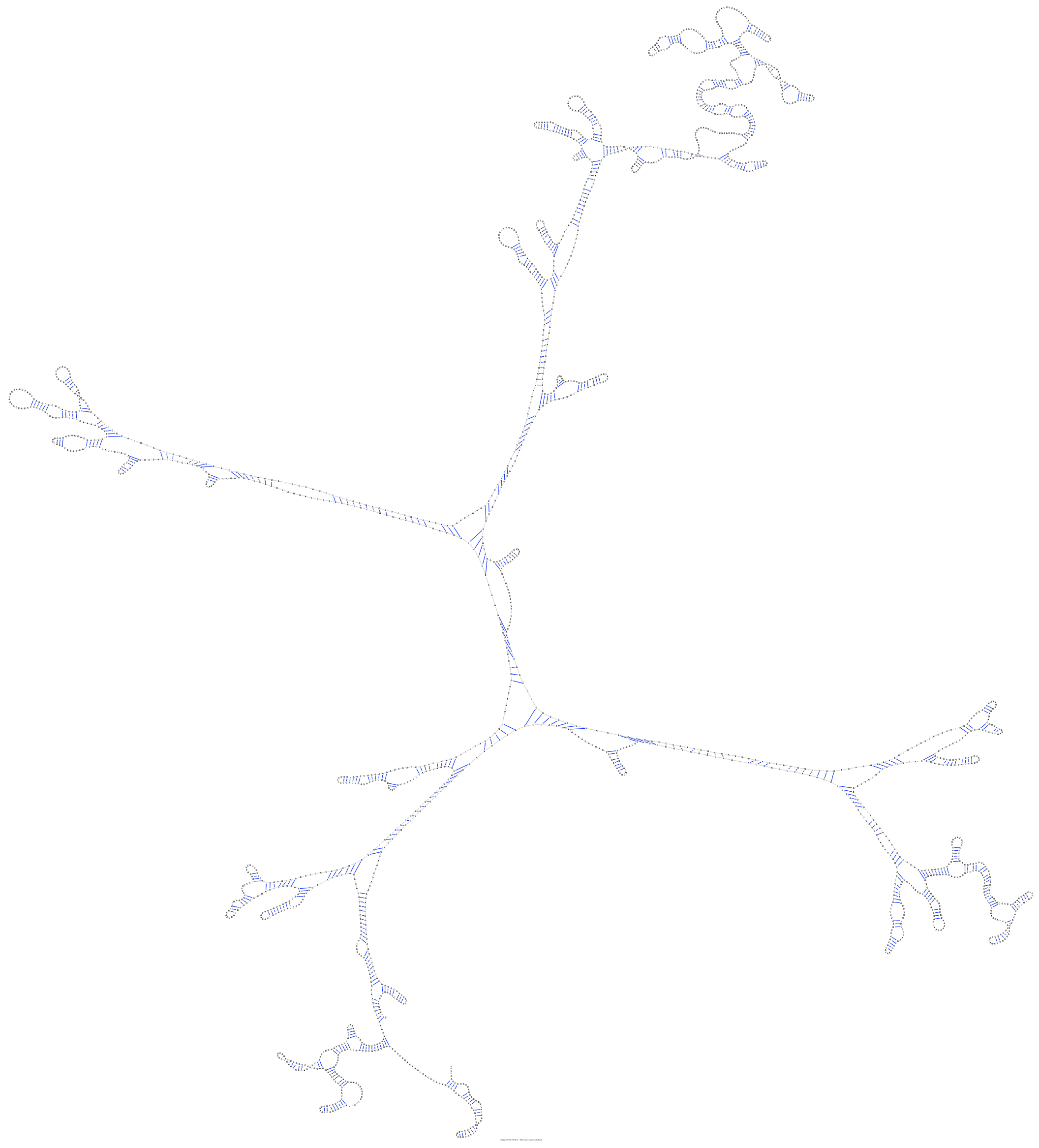


Figure A.39: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using the multithreading approach (thread number = 5)

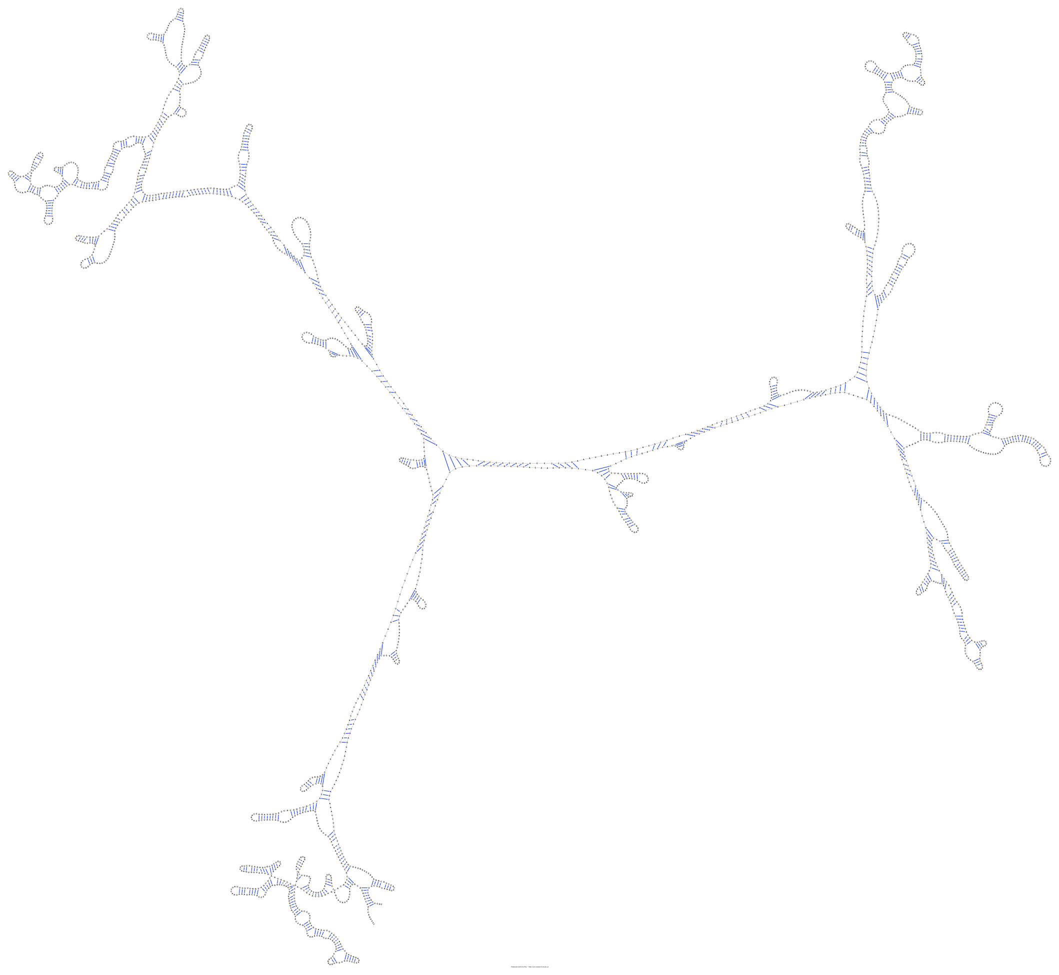


Figure A.40: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using the multithreading approach (thread number = 5)

A.6 Sequences' layouts under native C code

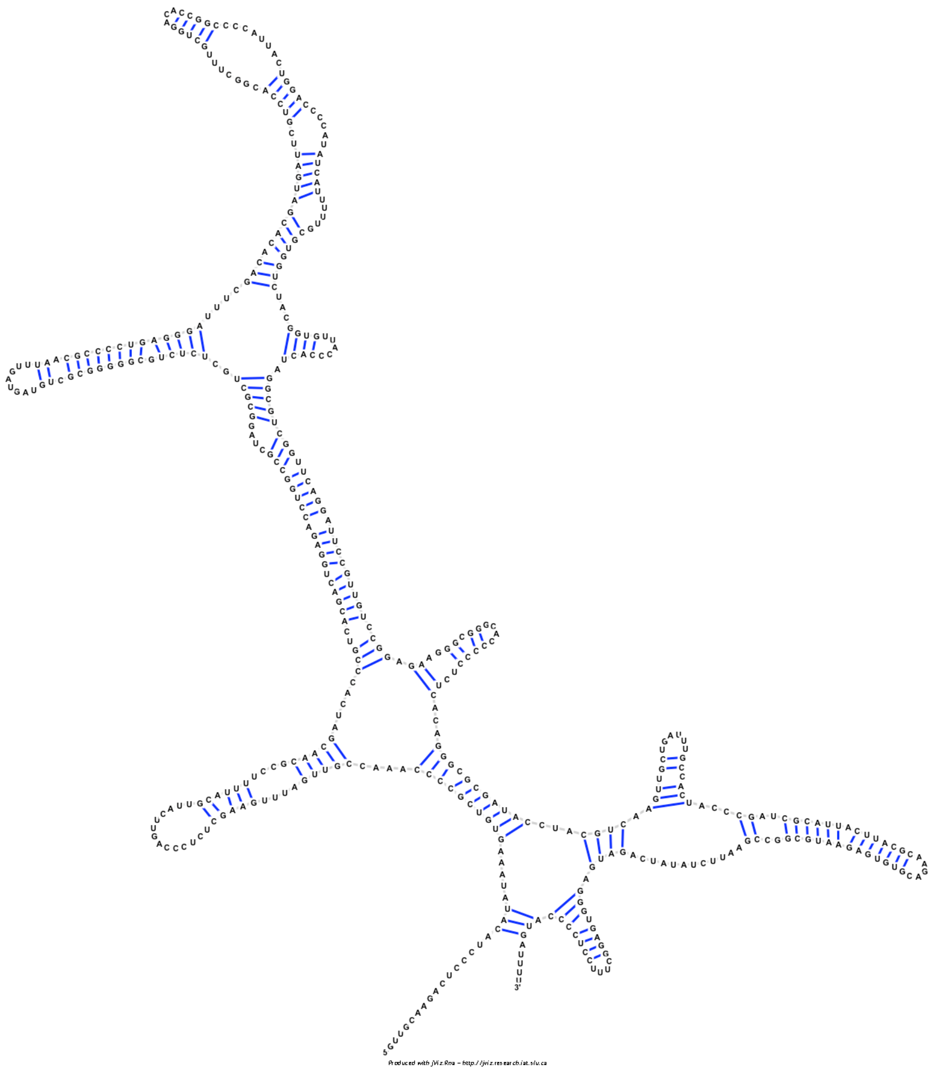


Figure A.41: The resulting sequence layout for *Metarhizium anisopliae* var. anisopliae strain 33 28S ribosomal RNA group IB intron (AF197122) using native C code

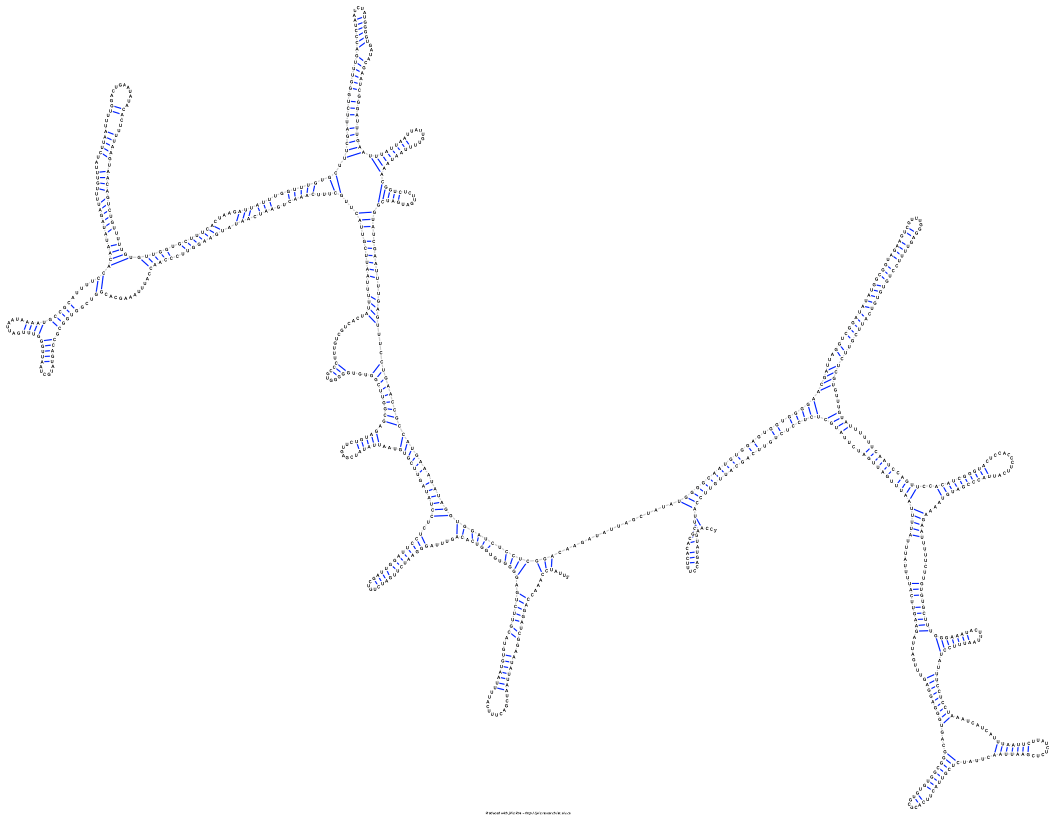


Figure A.42: The resulting sequence layout for *Acomys cahirinus* mitochondrial 12S ribosomal RNA (X84387) using native C code

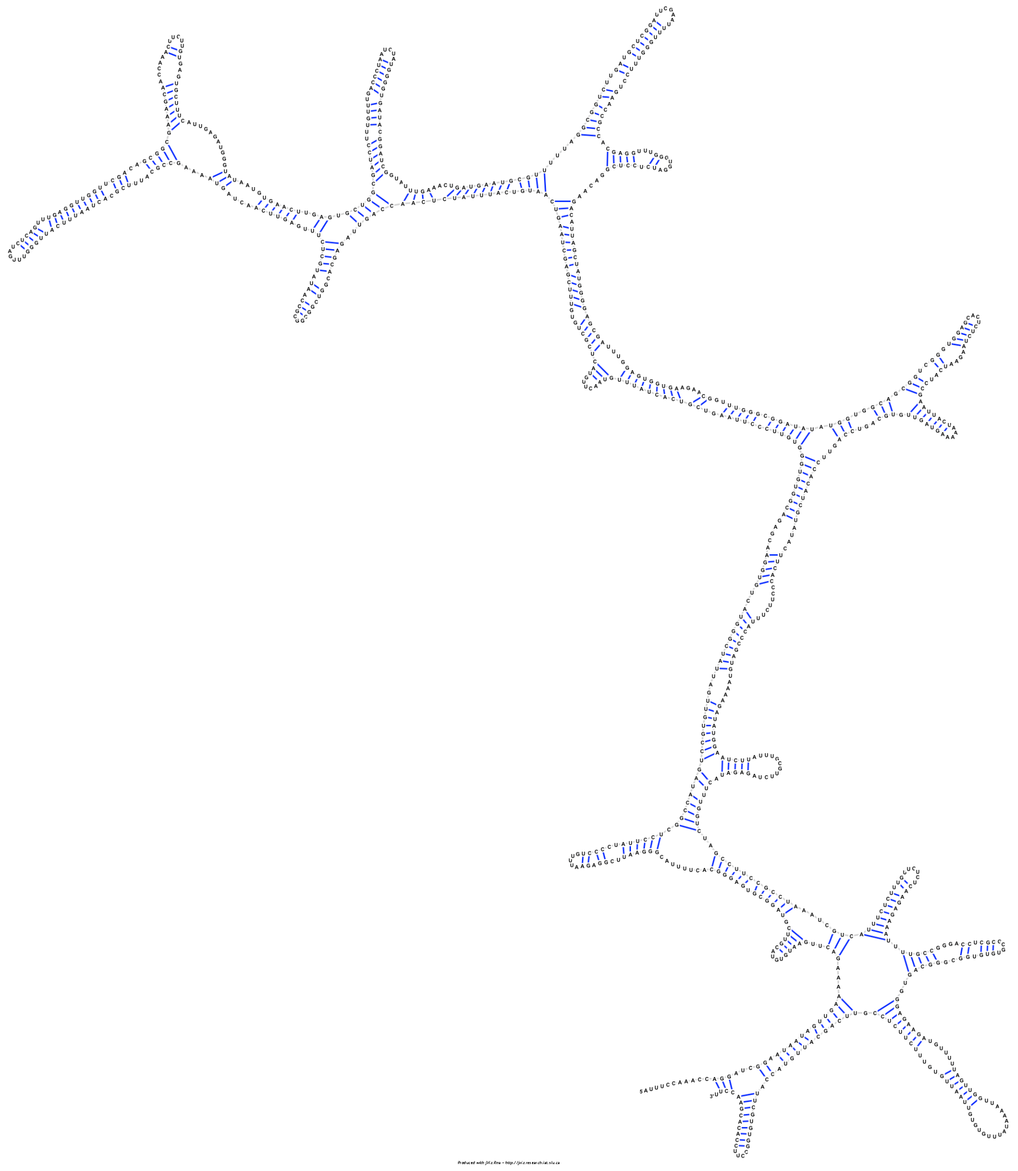


Figure A.43: The resulting sequence layout for *Xenopus laevis* mitochondrial 12S ribosomal RNA (M27605) using native C code

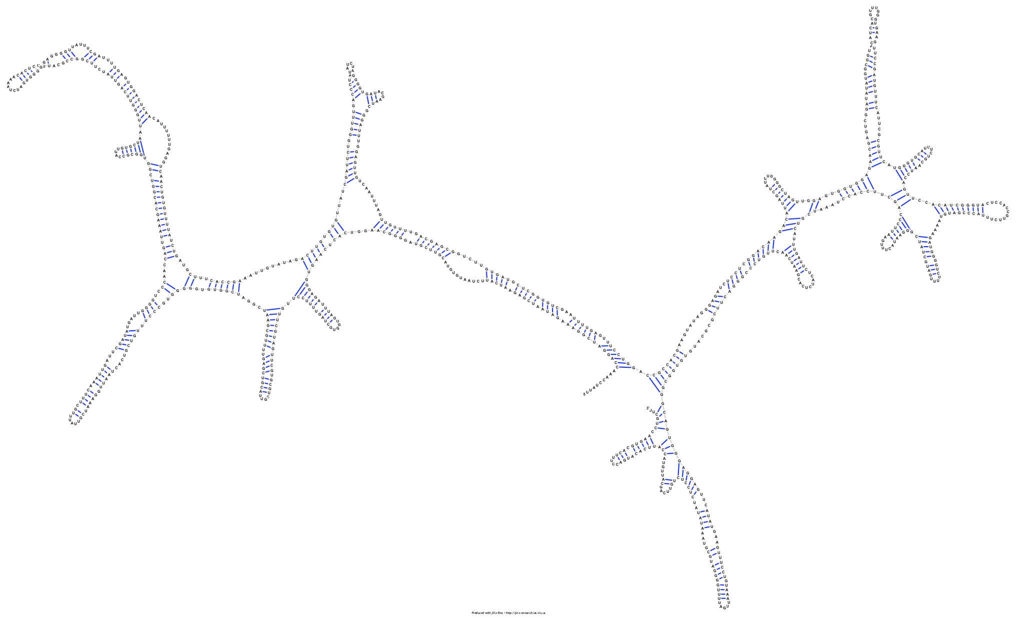


Figure A.44: The resulting sequence layout for *Homo sapiens* mitochondrial 16S ribosomal RNA (J01415:648-1601) using native C code

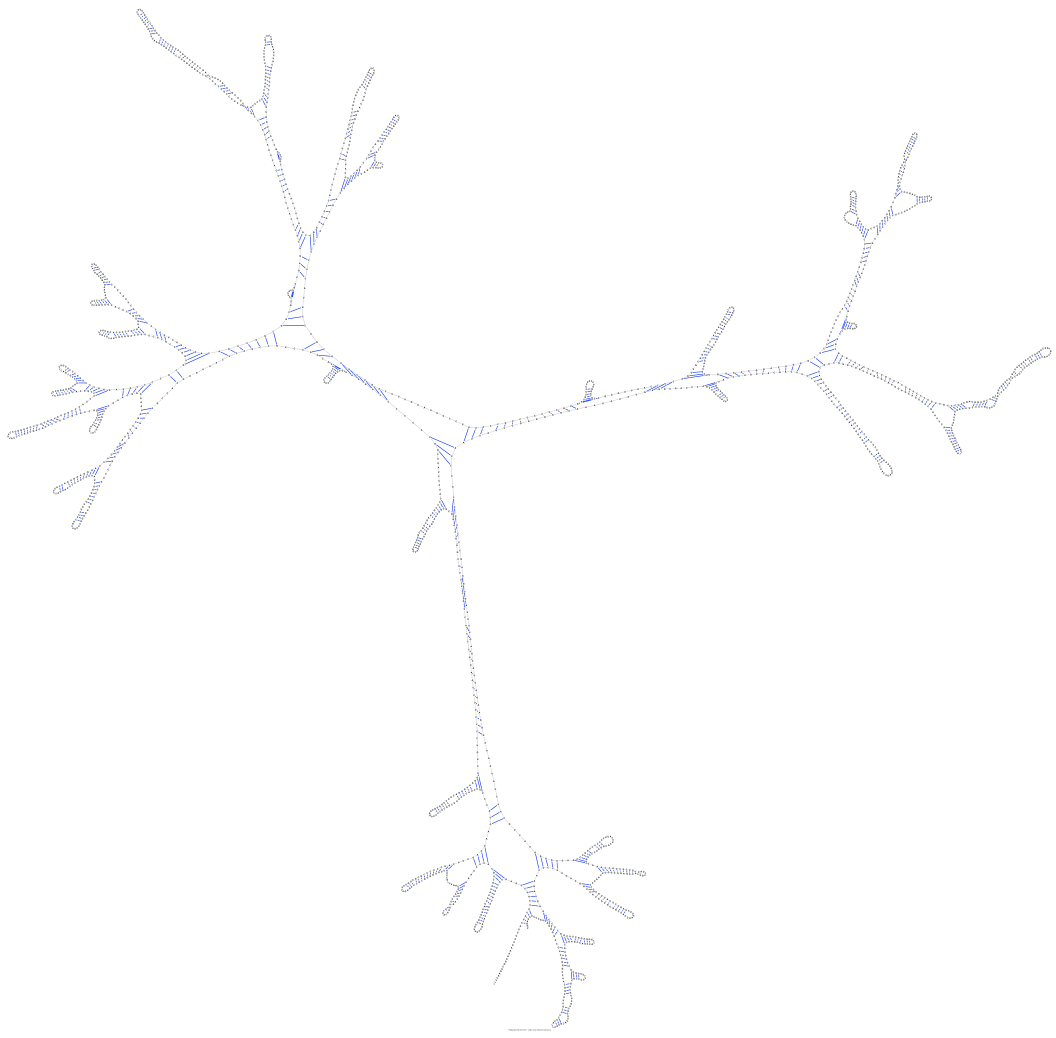


Figure A.45: The resulting sequence layout for *Sulfolobus acidocaldarius* 16S ribosomal RNA (D14876) using native C code

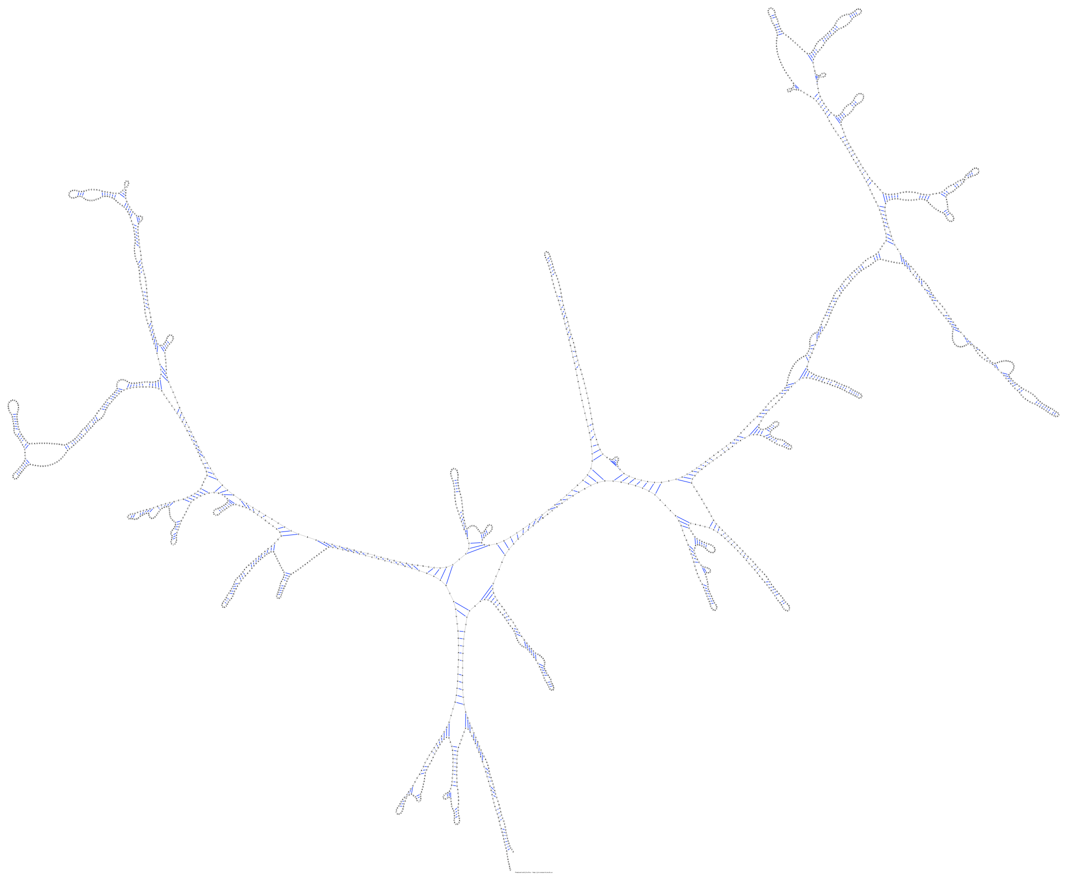


Figure A.46: The resulting sequence layout for *Aureoumbra lagunensis* 18S ribosomal RNA (U40258) using native C code

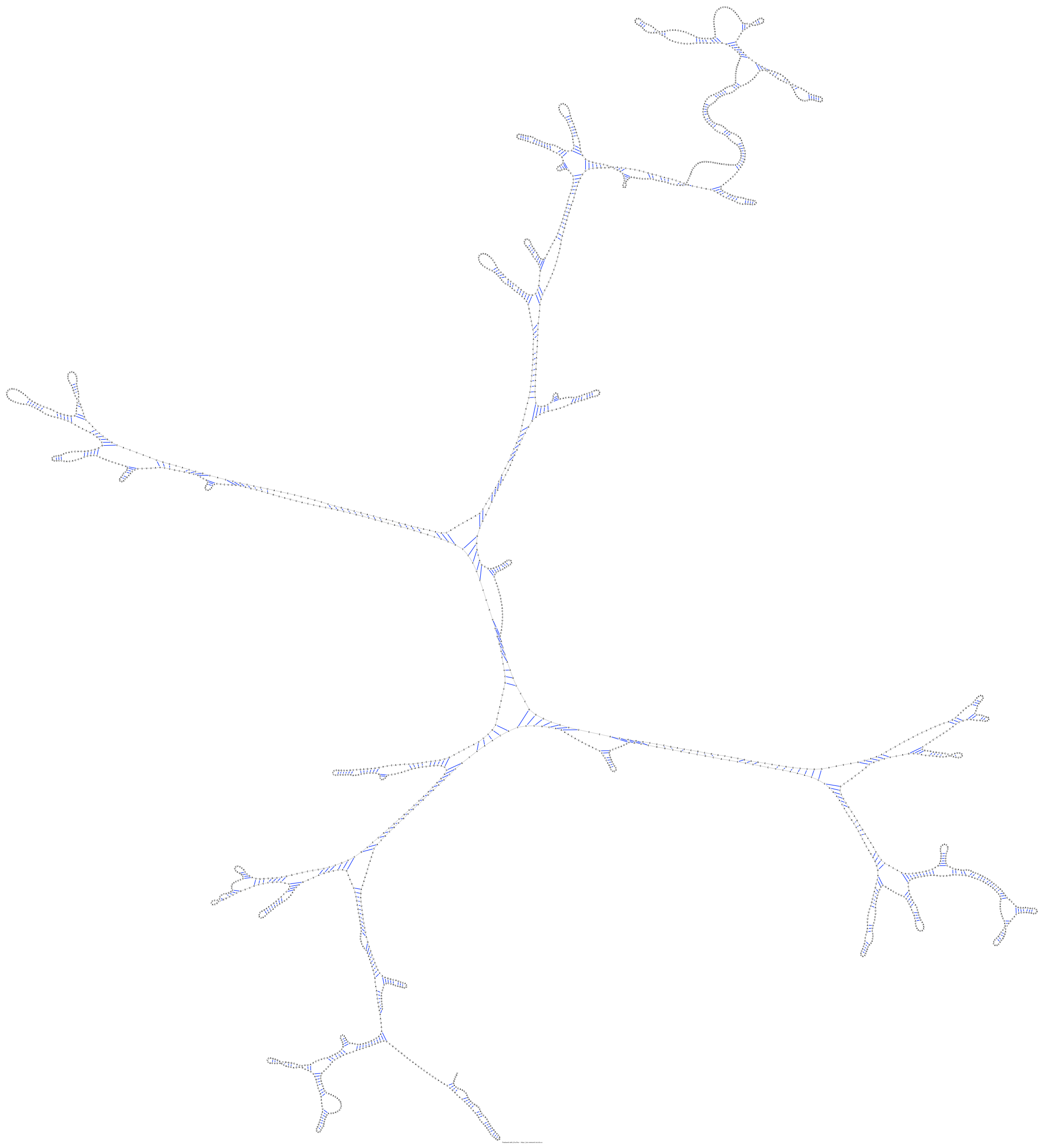


Figure A.47: The resulting sequence layout for *Hildenbrandia rubra* 18S ribosomal RNA (with intron) (L19345) using native C code

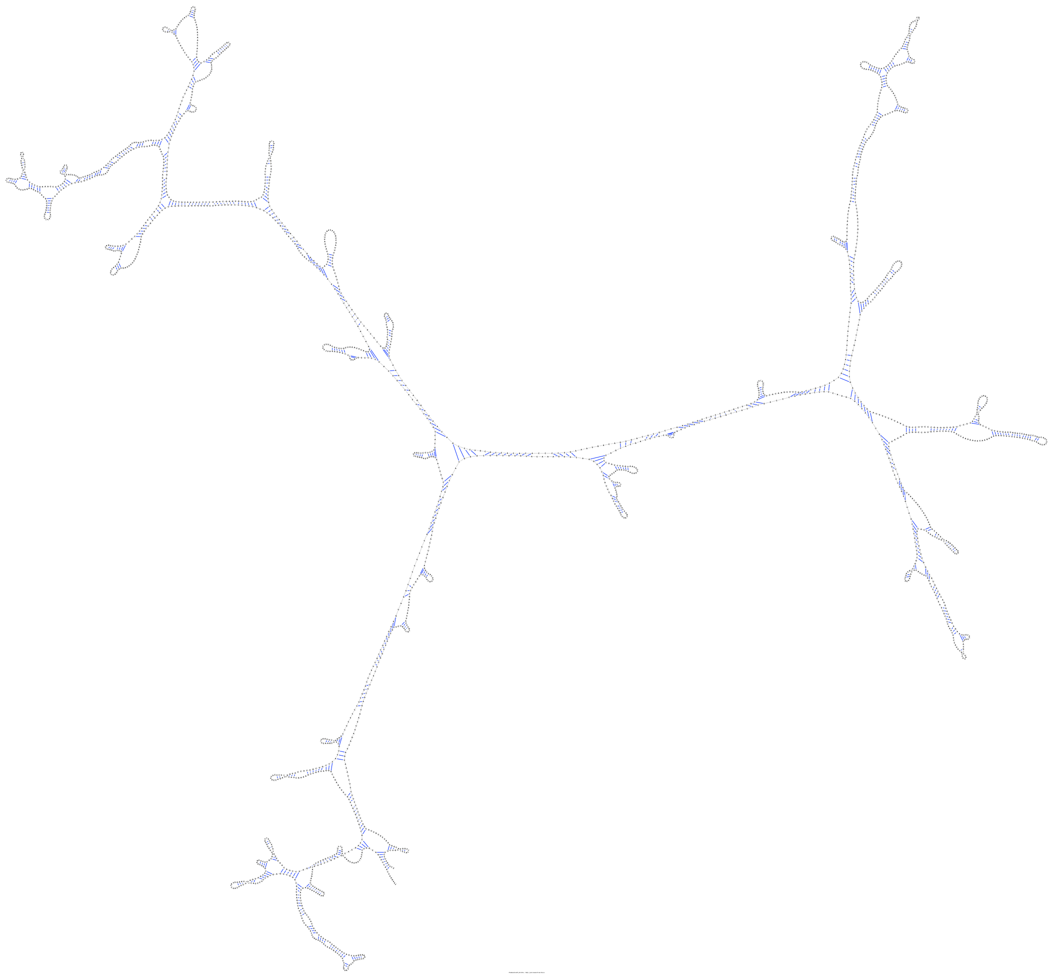


Figure A.48: The resulting sequence layout for *Porphyra leucosticta* 18S ribosmal RNA (with intron) (AF342746) using native C code

Appendix B

Full Classes for Speed Optimization

Algorithm B.1 The full code for the `QuadTreeNode` class

```
public class QuadTreeNode {
     private boolean empty = true;     // Whether this node is empty or not
     private boolean leaf = true;     // Whether this node is a leaf or not in the tree
     private boolean hasChildren = false;     // Whether this node has children
     private QuadTreeNode parent;     // A pointer to the parent. This will be used to update
                                      // the tree mass center
     private QuadTreeNode child1, child2, child3, child4; // Pointers to the children, always 4
     private int mass = 0;               // The mass of this node. The mass will always be an
                                         // integer multiple of 1
     private double centerOfMassX, centerOfMassY;     // The x and y co-ordinates of the center
                                                      // of mass of this node
     private double centerX, centerY;     // The x and y co-ordinates of the center of this
                                          // node. This is NOT the same as the center of mass.
                                          // This is the center that this node forms the
                                          // division in if it has children.
     private double size;
     public int repulsion;



     // Empty constructor
     public QuadTreeNode()
     {
          empty = true;
          leaf = true;
          hasChildren = false;
          mass = 0;
          centerOfMassX = centerOfMassY = centerX = centerY =  0.0;
          parent = null;
          child1 = child2 = child3 = child4 = null;
          size = 0.0;
     }//end constructor
     /*************************************************************************************/

     // A constructor with some default parameters
     public QuadTreeNode(QuadTreeNode theParent,
                         int theMass,
                         double theXComponent,
                         double theYComponent,
                         double centerXComponent,
                         double centerYComponent,
                         int theRepulsion)
     {
          empty = true;
          leaf = true;
          hasChildren = false;
          mass = theMass;
          centerOfMassX = theXComponent;
          centerOfMassY = theYComponent;
          centerX = centerXComponent;
          centerY = centerYComponent;
          parent = theParent;
          child1 = child2 = child3 = child4 = null;
          repulsion = theRepulsion;
          size = 0.0;
     }//end 2nd constructor
     /*************************************************************************************/

     // Sets whether this node is empty or not
     public boolean setEmptyState(boolean newState)
     {
          empty = newState;
          return empty;
     }//end setEmptyState
     /*************************************************************************************/
```

```
// Returns whether this node is empty or not
public boolean isEmpty()
{
     return empty;
}//end isEmpty
/**************************************************************************************/

// Sets whether this node is a leaf or not
public boolean setLeafState(boolean newState)
{
     leaf = newState;
     return leaf;
}//end setLeafState
/**************************************************************************************/

// Returns whether this node is a leaf or not
public boolean isLeaf()
{
     return leaf;
}//end getLeaf
/**************************************************************************************/

// Sets the center of mass of this node
public void setCenterOfMass(double X, double Y)
{
     centerOfMassX = X;
     centerOfMassY = Y;
}// end setCenterOfMass
/**************************************************************************************/
// The following two methods return the center of mass of this node
public double getCenterOfMassXComponent()
{
     return centerOfMassX;
}//end getCenterOfMassXComponent


public double getCenterOfMassYComponent()
{
     return centerOfMassY;
}//end getCenterOfMassYComponent
/**************************************************************************************/
// Sets the center of mass of this node
public void setCenterOfNode(double X, double Y)
{
     centerX = X;
     centerY = Y;
}// end setCenterOfMass
/**************************************************************************************/
```

```
    // The following two methods return the center of mass of this node
    public double getCenterOfNodeXComponent()
    {
         return centerX;
    }//end getCenterOfMassXComponent


    public double getCenterOfNodeYComponent()
    {
         return centerY;
    }//end getCenterOfMassYComponent
    /************************************************************************************/

    // Sets the new mass value
    public void setMass(int newMass)
    {
         mass = newMass;
    }//end setMass
    /************************************************************************************/

    // Gets the mass value
    public int getMass()
    {
         return mass;
    }//end getMass
    /************************************************************************************/

    // Sets the parent of this node to [theParent]
    public void setParent(QuadTreeNode theParent)
    {
         parent = theParent;
    }//end setParent
    /************************************************************************************/

    // Gets the parent of this node
    public QuadTreeNode getParent()
    {
         return parent;
    }//end getParent
    /************************************************************************************/

    // The following two methods set the children of this node
    public void setChildren(QuadTreeNode newChild1,
                            QuadTreeNode newChild2,
                            QuadTreeNode newChild3,
                            QuadTreeNode newChild4)
    {
         child1 = newChild1;
         child2 = newChild2;
         child3 = newChild3;
         child4 = newChild4;
    }//end setChildren

    public boolean setChild(int number, QuadTreeNode newChild)
    {
         if (newChild != null)
         {
              hasChildren = true;
         }
         switch (number) // If the number is between 1 and 4, change the corrosponding child
         {
              case 1:
                   child1 = newChild;
                   return true;
```

```
          case 2:
               child2 = newChild;
               return true;
          case 3:
               child3 = newChild;
               return true;
          case 4:
               child4 = newChild;
               return true;
          default:    // If the number is invalid, return with a false return status
                      // indicating failure
               return false;
     }//end switch (number)
}//end setChild
/***************************************************************************************/

// Returns the appropriate child based on [number]
public QuadTreeNode getChild(int number)
{
     switch (number)  // If the number is between 1 and 4, return the corrosponding child
     {
          case 1:
               return child1;
          case 2:
               return child2;
          case 3:
               return child3;
          case 4:
               return child4;
          default:            // If the number is invalid, return null indicating failure
               return null;     // NEED EXCEPTION
     }//end switch (number)
}//end getChild
/***************************************************************************************/

public boolean hasChildren()
{
     return hasChildren;
}
/***************************************************************************************/

// This method sets the size of this node to [theSize]
public void setSize(double theSize)
{
     size = theSize;
}//end setSize
/***************************************************************************************/

// This method gets the size of this node
public double getSize()
{
     return size;
}//end setSize
/***************************************************************************************/

// This method will copy data into THIS node from another one
public void copyData(QuadTreeNode theNode)
{
     empty = theNode.isEmpty();
     leaf = theNode.isLeaf();
     mass = theNode.getMass();
     centerX = theNode.getCenterOfNodeXComponent();
     centerY = theNode.getCenterOfNodeYComponent();
}//end copyData
/***************************************************************************************/
```

```
// This method will create 4 children for THIS node
public void createChildren()
{
     child1 = new QuadTreeNode();
     child1.setParent(this);
     child2 = new QuadTreeNode();
     child2.setParent(this);

     child3 = new QuadTreeNode();
     child3.setParent(this);

     child4 = new QuadTreeNode();
     child4.setParent(this);
}//end createChildren
/***************************************************************************************/

// Updates the mass center of mass for THIS node AND for its parent
public void updateCenterOfMass()
{
     int newMass = 0;                    // New parameters to hold the values
     double newCenterOfMassX = 0.0;
     double newCenterOfMassY = 0.0;
     int numOfNodes = 0;                    // How many nodes were counted

     // Add the data of each child, if they are available
     if (child1 != null)
     {
          newMass += child1.getMass();
          numOfNodes++;
          newCenterOfMassX += child1.getCenterOfMassXComponent();
          newCenterOfMassY += child1.getCenterOfMassYComponent();
     }//end if (child1 != null)
     if (child2 != null)
     {
          newMass += child2.getMass();
          numOfNodes++;
          newCenterOfMassX += child2.getCenterOfMassXComponent();
          newCenterOfMassY += child2.getCenterOfMassYComponent();
     }//end if (child2 != null)
     if (child3 != null)
     {
          newMass += child3.getMass();
          numOfNodes++;
          newCenterOfMassX += child3.getCenterOfMassXComponent();
          newCenterOfMassY += child3.getCenterOfMassYComponent();
     }//end if (child1 != null)
     if (child4 != null)
     {
          newMass += child4.getMass();
          numOfNodes++;
          newCenterOfMassX += child4.getCenterOfMassXComponent();
          newCenterOfMassY += child4.getCenterOfMassYComponent();
     }//end if (child1 != null)

     newCenterOfMassX /= numOfNodes;          // Average out the co-ordinates
     newCenterOfMassY /= numOfNodes;

     mass = newMass;                              // Set the new parameters
     centerOfMassX = newCenterOfMassX;
     centerOfMassY = newCenterOfMassY;

     if (parent != null) parent.updateCenterOfMass();     // If there is a parent,
                                                          // it needs to update too
}//end updateCenterOfMass
/***************************************************************************************/
```

```
// The method above won’t usually be called on THIS node, since this node’s mass and
// center will be set externally when the tree is built. The following method will be
// called by this node to begin a "bubble up" of updates up the tree all the way to the
// root


public void updateParent()
{
     if (parent != null) parent.updateCenterOfMass();     // Call the parent to update
                                                          // its data
}//end updateParent
/*********************************************************************************/

// This method updates the center of mass and mass for this node and the subtree below it.
// It uses recursion if you have a faster way in mind, go nuts
public void updateMassAndCenterOfMass()
{
     int newMass = 0;                   // initiate parameters to capture the parameters
                                        // from the children
     int nodeNumber = 0;
     double newCenterOfMassX = 0.0;
     double newCenterOfMassY = 0.0;
     int newRepulsion = 0;


     // If this node has children, its mass is the sum of the children’s mass and its
     // location is the average of the children’s locations
     if (child1 != null)
     {
          child1.updateMassAndCenterOfMass();     // First calculate the children’s
                                                  // new mass and center of mass
                                                  // so that this one is accurate
          newMass += child1.getMass();
          newCenterOfMassX += child1.getCenterOfMassXComponent();
          newCenterOfMassY += child1.getCenterOfMassYComponent();
          newRepulsion += child1.repulsion;
          nodeNumber++;
     }
     if (child2 != null)
     {
          child2.updateMassAndCenterOfMass();
          newMass += child2.getMass();
          newCenterOfMassX += child2.getCenterOfMassXComponent();
          newCenterOfMassY += child2.getCenterOfMassYComponent();
          newRepulsion += child2.repulsion;
          nodeNumber++;
     }
     if (child3 != null)
     {
          child3.updateMassAndCenterOfMass();
          newMass += child3.getMass();
          newCenterOfMassX += child3.getCenterOfMassXComponent();
          newCenterOfMassY += child3.getCenterOfMassYComponent();
          newRepulsion += child3.repulsion;
          nodeNumber++;
     }
     if (child4 != null)
     {
          child4.updateMassAndCenterOfMass();
          newMass += child4.getMass();
          newCenterOfMassX += child4.getCenterOfMassXComponent();
          newCenterOfMassY += child4.getCenterOfMassYComponent();
          newRepulsion += child4.repulsion;
          nodeNumber++;
     }
```

```
          if (nodeNumber!=0)     // This is only true if this node has children, if it doesn't
                                 // we don't need to
                                 // change its location or mass
          {
               mass = newMass;
               centerOfMassX = (newCenterOfMassX/nodeNumber);
               centerOfMassY = (newCenterOfMassY/nodeNumber);
               repulsion = newRepulsion/nodeNumber;
          }
     }//end updateMassAndCenterOfMass
}
```

Algorithm B.2 The full code for the `QuadTree` class

```
import QuadTree.QuadTreeNode;
import com.touchgraph.graphlayout.Node;

public class QuadTree {

     private QuadTreeNode root;
     private double topLeftX, topLeftY, bottomRightX, bottomRightY;
     private static final int CHILD1 = 1;
     private static final int CHILD2 = 2;
     private static final int CHILD3 = 3;
     private static final int CHILD4 = 4;
     private static final double theta = 5.0;


     // This quad tree can only be defined within a region. The region is a rectangle
     // determined by its top left point and bottom right point. Hence there is no empty
     // constructor.
     public QuadTree(double newTopLeftX,
                     double newTopLeftY,
                     double newBottomRightX,
                     double newBottomRightY)
     {
          root = null;

          topLeftX = newTopLeftX;          // Define the closure of the tree
          topLeftY = newTopLeftY;
          bottomRightX = newBottomRightX;
          bottomRightY = newBottomRightY;
     }
     /*************************************************************************************/

     // Inserts the node [theNode] in its rightful place
     public void insertNode(QuadTreeNode theNode)
     {
          QuadTreeNode currentNode = root;     // First try and insert at the root
          QuadTreeNode currentNodesParent = root;     // Keep track of the parent
          double currentTopLeftX = topLeftX;
          double currentTopLeftY = topLeftY;
          double currentBottomRightX = bottomRightX;
          double currentBottomRightY = bottomRightY;
          double currentNodeX, currentNodeY;
          int whichChild = 0;

          if (root == null)
          {
               root = theNode;
               return;                    // simplest case
          }//end if (root == null)

          while (currentNode != theNode)     // We will set the [currentNode] to null when we
                                             // are done
          {
               if (currentNode == null)
               {
                    currentNodesParent.setChild(whichChild, theNode);
                    theNode.setParent(currentNodesParent);
                    currentNode = theNode;
               }
               else
               {
```

```
if (!(currentNode.hasChildren()))     // This means that [currentNode] holds
                                      // the value of an object
                                      // in our 2D space. We need to first
                                      // set that object as a child
                                      // of [currentNode], set current
                                      // node to hold the value of a
                                      // subspace, and only THEN can we
                                      // insert [theNode]
{

     QuadTreeNode newNode = new QuadTreeNode(); // Create a new node and
                                                // set it responsible for
                                                // the current subspace
     newNode.setCenterOfNode(((currentBottomRightX + currentTopLeftX)/2),
                             ((currentBottomRightY + currentTopLeftY)/2));
     newNode.setLeafState(false);
     newNode.setEmptyState(false);


     if (currentNode == root)
     {
          root = newNode;
     }
     else if (currentNodesParent != currentNode) // This will be true in
                                                 // every case unless
                                                 // [currentNode] is the
                                                 // root. If this is true
                                                 // we need to set
                                                 // [newNode] as the
                                                 // child of
                                                 // [currentNodesParent]
                                                 // to  replace
                                                 // [currentNode]
     {
          whichChild = findLocationOFNodeAsChild(currentNodesParent,
                                                 newNode);
          currentNodesParent.setChild(whichChild, newNode);
     }//end if (currentNodesParent != currentNode)
     else
     {
          System.out.println("third case");
          System.exit(0);
     }

     whichChild = findLocationOFNodeAsChild(newNode, currentNode);
     newNode.setChild(whichChild, currentNode);
     newNode.setParent(currentNode.getParent());
     currentNode.setParent(newNode);

     currentNode = newNode;    // This is important. In the following code
                               // block we want to check the placement of
                               // [theNode] with respect to the subspace
                               // we just created. Which is represented
                               // by [newNode]

}//end if (!(currentNode.hasChildren()))

currentNodeX = currentNode.getCenterOfNodeXComponent();
currentNodeY = currentNode.getCenterOfNodeYComponent();

whichChild = findLocationOFNodeAsChild(currentNode, theNode);
currentNodesParent = currentNode;

currentNode.setSize(java.lang.Math.abs(currentTopLeftX -
                                       currentBottomRightX));
```

```
                switch(whichChild)
                {
                     case CHILD1:                    // SW
                          // Make the space in question smaller
                          // Leave [currentTopLeftX] as is
                          currentTopLeftY = currentNodeY;
                          currentBottomRightX = currentNodeX;
                          // Leave [currentBottomRightY] as is

                          currentNode = currentNode.getChild(1);     // Look at child1 of
                                                                     // the current node
                          break;
                     case CHILD2:                    // NW
                          // Make the space in question smaller
                          // Leave [currentTopLeftX] as is
                          // Leave [currentTopLeftY] as is
                          currentBottomRightX = currentNodeX;
                          currentBottomRightY = currentNodeY;

                          currentNode = currentNode.getChild(2);     // Look at child1 of
                                                                     // the current node
                          break;
                     case CHILD3:                    // SE
                          // Make the space in question smaller
                          currentTopLeftX = currentNodeX;
                          currentTopLeftY = currentNodeY;
                          // Leave [currentBottomRightX] as is
                          // Leave [currentBottomRightY] as is

                          currentNode = currentNode.getChild(3);     // Look at child1 of
                                                                     // the current node
                          break;
                     case CHILD4:                    // NE
                          // Make the space in question smaller
                          currentTopLeftX = currentNodeX;
                          // Leave [currentTopLeftY] as is
                          // Leave [currentBottomRightX] as is
                          currentBottomRightY = currentNodeY;

                          currentNode = currentNode.getChild(4);     // Look at child1 of
                                                                     // the current node
                          break;
                }//end switch(whichChild)
           }//end else
      }//end while (curentNode != thNode)
}//end insertNode

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

// This method takes a node [theParent] and finds where [theChild] should go in relation
// to [theParent].
public int findLocationOFNodeAsChild(QuadTreeNode theParent, QuadTreeNode theChild)
{
     double parentX = theParent.getCenterOfNodeXComponent();
     double parentY = theParent.getCenterOfNodeYComponent();
     double childX = theChild.getCenterOfNodeXComponent();
     double childY = theChild.getCenterOfNodeYComponent();

     if (childX < parentX)               // The child is west of the parent
     {
          if (childY < parentY)          // The child is SW of the parent
          {
               return CHILD1;
          }//end if (childY < parentY)
```

```
            else                                // The child is NW (or center-west) of the
                                                // parent
            {
                return CHILD2;
            }//end else
        }//end if (childX < parentX)
        else                                    // The child is east (or center to) the
                                                // parent
        {
            if (childY < parentY)           // The child is SE (or south-center) to the
                                            // parent
            {
                return CHILD3;
            }//end if (childY < parentY)
            else                                // The child is NE (or in the same place) to
                                                // the parent
            {
                return CHILD4;
            }//end else
        }//end else
    }//end findLocationOfNodeAsChild

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

    // This method updates the tree's center of mass and mass from the root down
    public void updateMassAndCenterOfMass()
    {
        if (root != null)
        {
            root.updateMassAndCenterOfMass();
        }//end if (root != null)
    }//end updateMassAndCenterOfMass
    /*************************************************************************************/

    // This method updates the forces on the node [theNode] and stores them in the array dxdy
    // where dxdy[0] = dx and dxdy[1] = dy. The array dxdy is assumed to have exactly
    // two elements.
    public void updateForcesOnNode(double nodeX,
                                   double nodeY,
                                   int nodeRepulsion,
                                   double[] dxdy,
                                   double rigidity,
                                   boolean justMadeLocal)
    {
        updateForcesOnNode(nodeX, nodeY, nodeRepulsion,root, dxdy, rigidity, justMadeLocal);
    }//end updateForcesOnNode
    // This method is updates the forces on the node [theNode] by the subtree under the node
    // [byNode]. It puts the x and y displacement values (dx and dy) into the array [dxdy].
    // The array [dxdy] is assumed to have exactly two elements where dxdy[0] = dx and
    // dxdy[1] = dy. dx and dy symbolize the forces on the node [theNode]
    public void updateForcesOnNode(double theNodeX,
                                   double theNodeY,
                                   int nodeRepulsion,
                                   QuadTreeNode byNode,
                                   double[] dxdy,
                                   double rigidity,
                                   boolean justMadeLocal)
    {
        double targetNodeX = byNode.getCenterOfNodeXComponent();      // Get the location
                                                                      // of the target node
        double targetNodeY = byNode.getCenterOfNodeYComponent();      // we wish to
                                                                      // estimate
        double targetNodeSize = byNode.getSize();                         // Get its size
```

```
// This chunk of code is almost identical to the one appearing in
// TGLayout::avoidLabels()
double vx = targetNodeX-theNodeX;  // Calculate distance and ratio between
double vy = targetNodeY-theNodeY;  // distance and size of the node
                                   // (nodes that represent single bases
                                   // have size 0.0)

double len = vx*vx + vy*vy;

double distance = java.lang.Math.sqrt(len);
if (distance == 0)
{
     return;
}//end if (distance == 0)

double localTheta = targetNodeSize/distance;
if (localTheta <= theta)                    // The node is far enough and small
                                            // enough to estimate
{
     double nodeMassCenterX = byNode.getCenterOfMassXComponent();
     double nodeMassCenterY = byNode.getCenterOfMassYComponent();

     vx = nodeMassCenterX - theNodeX;
     vy = nodeMassCenterY - theNodeY;

     len = vx*vx + vy*vy;

     double dx = vx/len;
     double dy = vy/len;

     int repSum = nodeRepulsion * byNode.repulsion/100;
     if (justMadeLocal)
     {
          dxdy[0] += dx*repSum*byNode.getMass()*rigidity;
          dxdy[1] += dy*repSum*byNode.getMass()*rigidity;
     }//end if (justMadeLocal)
     else
     {
          dxdy[0] += dx*repSum*byNode.getMass()*rigidity/10;
          dxdy[1] += dy*repSum*byNode.getMass()*rigidity/10;
     }//end else
}//end if (localTheta <= theta)
else                       // otherwise calculate the forces from each child
{
     if (byNode.getChild(1) != null)
     {
       updateForcesOnNode(theNodeX,
                          theNodeY,
                          nodeRepulsion,
                          byNode.getChild(1),
                          dxdy,
                          rigidity,
                          justMadeLocal);
     }
     if (byNode.getChild(2) != null)
     {
       updateForcesOnNode(theNodeX,
                          theNodeY,
                          nodeRepulsion,
                          byNode.getChild(2),
                          dxdy,
                          rigidity,
                          justMadeLocal);
     }
```

```
          if (byNode.getChild(3) != null)
          {
            updateForcesOnNode(theNodeX,
                               theNodeY,
                               nodeRepulsion,
                               byNode.getChild(3)
                               ,dxdy,
                               rigidity,
                               justMadeLocal);
          }
          if (byNode.getChild(4) != null)
          {
            updateForcesOnNode(theNodeX,
                               theNodeY,
                               nodeRepulsion,
                               byNode.getChild(4),
                               dxdy,
                               rigidity,
                               justMadeLocal);
          }
     }//end else

}//end updateForcesOnNode
/*************************************************************************************/

// Prints out the quadtree. This method is very slow and uses recursion, but it
// will be used to verify correctness of the tree
public void printTree()
{
     System.out.println("\n\n\n --------NEW TREE--------");
     if (root != null)
     {
          root.printSubtree(0);
     }//end if (root != null)
     else
     {
          System.out.println("Empty tree");
     }
}//end printTree
/*************************************************************************************/
}
```

Algorithm B.3 The full code for the `Searcher` class

```
import ca.sfu.iat.research.jviz.uielements.jVizCore;
import java.lang.management.*;
import FasterQuadTree.FasterQuadTree;
import com.touchgraph.graphlayout.Node;
import com.touchgraph.graphlayout.graphelements.GraphEltSet;
import com.touchgraph.graphlayout.graphelements.TGForEachNodePair;
import java.util.Iterator;
import java.util.Vector;

public class Searcher implements Runnable{

     private Thread myThreat;
     private int searcherID;
     static private int searcherNumber;
     static private int nodeNumber;
     static private double[] xSet;
     static private double[] ySet;
     static private int[] repulsionSet;
     static private double[] dxSet;
     static private double[] dySet;
     static private double rigidity;
     static private int[] finishedThreads;
     private static final double K = 10.0;
     private static final double radius = 360000.0;


     /**************************************************************************************/
     /****************************         METHOD DEFINITIONS        *********************/
     /**************************************************************************************/

     public Searcher(int id)
     {
          searcherID = id;
     }

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

     public static void setParameters(int theSearcherNumber,
                                int theNodeNumber,
                                double[] theXSet,
                                double[] theYSet,
                                int[] theRepulsionSet,
                                double[] theDXSet,
                                double[] theDYSet,
                                double theRigidity,
                                int[] theFinishedThreads)
     {
          searcherNumber = theSearcherNumber;
          nodeNumber = theNodeNumber;
          xSet = theXSet;
          ySet = theYSet;
          repulsionSet = theRepulsionSet;
          dxSet = theDXSet;
          dySet = theDYSet;
          rigidity = theRigidity;
          finishedThreads = theFinishedThreads;
     }

     /**************************************************************************************/
     public void start()
     {
          myThreat = new Thread(this, "Searcher"+searcherID);
          myThreat.start();
     }

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

```
    public void stop()
    {
         myThreat = null;
    }
    /*************************************************************************************/
    public void run()
    {
         if (xSet == null)
         {
              System.out.println("You have not initalized the parameters");
              return;
         }
         for (int nodeInspecting=searcherID; nodeInspecting<nodeNumber;)
         {
              double currentX = xSet[nodeInspecting];
              double currentY = ySet[nodeInspecting];
              int currentRepulsion = repulsionSet[nodeInspecting];
              for (int nodeInspected=0; nodeInspected<nodeNumber; nodeInspected++)
              {
                   double dx = 0;
                   double dy = 0;

                   double vx = currentX - xSet[nodeInspected];
                   double vy = currentY - ySet[nodeInspected];

                   double len = vx*vx + vy*vy;

                   if (len<radius)
                   {

                        if (len !=0)
                        {
                             dx = vx/len;
                             dy = vy/len;

                             int repSum = currentRepulsion * repulsionSet[nodeInspected]/100;

                             dx = K*dx*repSum*rigidity;
                             dy = K*dy*repSum*rigidity;

                             dx /= 10;
                             dy /= 10;

                             dxSet[nodeInspecting] -= dx;
                             dySet[nodeInspecting] -= dy;
                        }
                   }
              }
              nodeInspecting+=searcherNumber;
         }
         finishedThreads[0]++;
    }
    /*************************************************************************************/
    public double[] getDXSet()
    {
         return dxSet;
    }
    /*************************************************************************************/
    public double[] getDYSet()
    {
         return dySet;
    }

    /*************************************************************************************/
    public boolean isAlive()
    {
         return myThreat.isAlive();
    }

    /*************************************************************************************/
    public boolean didSetParameters()
    {
         if (xSet == null) return false;
         else return true;
    }
    /*************************************************************************************/
}
```

Appendix C

Full Classes For File I/O

Algorithm C.1 The full code for the `RnamlFile` class

```
package ca.sfu.iat.research.jviz.file;

import org.w3c.dom.*;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;


import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;

import ca.sfu.iat.research.jviz.structuralelements.Nucleotide;
import ca.sfu.iat.research.jviz.structuralelements.RnaStructure;

public class RnamlFile extends StructureFile
{
  private Document theDocument = null;

  public RnamlFile(String pathname)
  {
   super(pathname);
   System.out.println("Reading File "+this.getPath());
   try
   {
    // First we need to create the document object
    // For that, we need a document builder
    DocumentBuilderFactory theBuilderFactory = DocumentBuilderFactory.newInstance();

    // Set a few features so that the document does not search for a .dtd file
    // found on
    // http://stackoverflow.com/questions/155101/make-documentbuilder-parse-ignore-
    // dtd-references
    theBuilderFactory.setValidating(false);
    theBuilderFactory.setFeature("http://xml.org/sax/features/namespaces", false);
    theBuilderFactory.setFeature("http://xml.org/sax/features/validation", false);
theBuilderFactory.setFeature("http://apache.org/xml/features/nonvalidating/load-dtd-grammar",
                             false);
theBuilderFactory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",
                             false);


       DocumentBuilder theBuilder = theBuilderFactory.newDocumentBuilder();
       // Now we can make the document by parsing the file
       theDocument = theBuilder.parse(this);

       System.out.println("Parsed the document");
     }
     catch (Exception e)
     {
       e.printStackTrace();
     }
   }

   public RnaStructure readFile()
   {
     System.out.println("RnamlFile::readFile() - reading "+this.getPath());
     String RNASequence = null;
     int[] basePairArray = null;
     try
     {
       // Let’s start looking at the RNA molecule
       Element rootElementRNAML = theDocument.getDocumentElement();
       // First we get the molecule
       NodeList molecules = rootElementRNAML.getElementsByTagName("molecule");
       Element molecule = (Element)molecules.item(0);
       if (molecule != null)
       {
          // Then it has two components, the sequence and the structure
          NodeList sequences = molecule.getElementsByTagName("sequence");
          Element sequenceComponent = (Element)sequences.item(0);
          NodeList structures = molecule.getElementsByTagName("structure");
          Element structureComponent = (Element)structures.item(0);
          if (sequenceComponent != null)
          {
            // Now let’s get the sequence (string) from the sequence component
            NodeList sequenceStrings = sequenceComponent.getElementsByTagName("seq-data");
            Element sequenceString = (Element)sequenceStrings.item(0);
            if (sequenceString != null)
            {
              NodeList sequence = sequenceString.getChildNodes();
              RNASequence = ((Node)sequence.item(0)).getNodeValue();

              System.out.println("RNA Sequence right after reading: "+RNASequence);

              // Replace unneccessry characters such as spaces, lower case letters and
              // new lines
              RNASequence = RNASequence.replace(" ", "");
              RNASequence = RNASequence.replace("\n", "");
              RNASequence = RNASequence.replace(’a’, ’A’);
              RNASequence = RNASequence.replace(’g’, ’G’);
              RNASequence = RNASequence.replace(’u’, ’U’);
              RNASequence = RNASequence.replace(’c’, ’C’);

              System.out.println("RNA Sequence after processing: "+RNASequence);
            }//end if (sequenceString != null)
          }//end if (sequenceComponent != null)
          if (structureComponent != null)
          {
            // Let’s find all the base pairs
            NodeList basePairElements = structureComponent.getElementsByTagName("base-pair");
            int basePairNumber = basePairElements.getLength();
            System.out.println("There are "+basePairNumber+" base pairs in this sequence");

            // This array will store all the base pairs in the form of base1,base2 where
            // base1 is connected to base2
            basePairArray = new int[RNASequence.length()];
            for (int i=0; i<RNASequence.length(); i++)
            {
               basePairArray[i] = -1;
            }//end for (int i=0; i<basePairArray; i++)

            // loop through the list and find info on each base pair
            for (int i=0; i<basePairNumber; i++)
            {
               Element basePairElement = (Element)basePairElements.item(i);
               if (basePairElement != null)
               {
                 // get the value for the type of edge as seen from the 3’ and 5’ end
                 // 5’
                 NodeList edgeDescriptions = basePairElement.getElementsByTagName("edge-5p");
                 Element edgeDescription5p = (Element)edgeDescriptions.item(0);

                 NodeList edgeDescription5pValues = edgeDescription5p.getChildNodes();
                 String edgeDescription5pValue =
                                      ((Node)edgeDescription5pValues.item(0)).getNodeValue();
```

```
            // 3'
            edgeDescriptions = basePairElement.getElementsByTagName("edge-3p");
            Element edgeDescription3p = (Element)edgeDescriptions.item(0);

            NodeList edgeDescription3pValues = edgeDescription5p.getChildNodes();
            String edgeDescription3pValue =
                         ((Node)edgeDescription5pValues.item(0)).getNodeValue();

             if (((edgeDescription5pValue.equals("+")) ||
                  (edgeDescription5pValue.equals("-"))) &&
                ((edgeDescription3pValue.equals("+")) ||
                  (edgeDescription3pValue.equals("-"))))
             {

              NodeList basePairPositions =
                           basePairElement.getElementsByTagName("position");
              // Two elements should be returned, base1 and base2
              Element basePairPosition1 = (Element)basePairPositions.item(0);
              Element basePairPosition2 = (Element)basePairPositions.item(1);

              NodeList base1 = basePairPosition1.getChildNodes();
              NodeList base2 = basePairPosition2.getChildNodes();

              String base1value = ((Node)base1.item(0)).getNodeValue();
              String base2value = ((Node)base2.item(0)).getNodeValue();

              basePairArray[(Integer.parseInt(base1value)-1)] =
                                        (Integer.parseInt(base2value)-1);
              basePairArray[(Integer.parseInt(base2value)-1)] =
                                        (Integer.parseInt(base1value)-1);

             }//end if (((edgeDescription5pValue.equals("+")) ||
              //         (edgeDescription5pValue.equals("-"))) &&
             //        ((edgeDescription3pValue.equals("+")) ||
             //         (edgeDescription3pValue.equals("-"))))

          }//end if (basePairElement != null)
        }//end for (int i=0; i<basePairNumber; i++)
     }//end if (structureComponent != null)

     ArrayList<Nucleotide> nucList = new ArrayList<Nucleotide>();
     Nucleotide nuc;
     int basePairNumber = RNASequence.length();
     int k=0;
     for (int i=0; i<basePairNumber; i++)
     {
        nuc = new Nucleotide(i,
                  ""+RNASequence.charAt(i),
                  basePairArray[i]);
        nucList.add(nuc);
     }//end for (int i=0; i<basePairNumber; i++)

     RnaStructure returnStructure = new RnaStructure(nucList,
                        this.getName(),
                        "");
     returnStructure.setCoorFile(myCoordinateFile);

     return returnStructure;
   }//end if (molecule != null)

 }//end try
```

```
      catch (Exception e)
      {
         e.printStackTrace();
      }


      return null;
    }

    public boolean writeFile(ArrayList<Nucleotide> structure)
    {
      return true;
    }//end writeFile(ArrayList<Nucleotide> structure)

    @Override
    public int getType()
    {
      return FileUtils.RNAML;
    }
}
```

Algorithm C.2 The full code for the `FastaFile` class

```
package ca.sfu.iat.research.jviz.file;


import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;
import java.util.Stack;
import java.util.EmptyStackException;

import ca.sfu.iat.research.jviz.structuralelements.Nucleotide;
import ca.sfu.iat.research.jviz.structuralelements.RnaStructure;

public class FastaFile extends StructureFile
   {
     BufferedReader inputStream;
     public FastaFile(String pathname)
     {
        super(pathname);
     }

     public RnaStructure readFile()
     {
        try
        {
          inputStream = new BufferedReader(new FileReader(this.getPath()));   //Open the file
          String myHeader, mySequence, myStructure;
          myHeader = inputStream.readLine();

          // Make sure the header starts with the '>'
          // character to make it a proper header
          if (!(myHeader.charAt(0) == '>'))
          {
             mySequence = myHeader;
             myHeader = "";
          }
          else
          {
             mySequence = inputStream.readLine();
          }//end else

          // At this point we have a sequence.
          // Make sure it is a valid sequence
          // Otherwise, the file is bad
          mySequence = mySequence.toUpperCase();
          if (mySequence.matches(".*[^ACGU].*"))
          {
             throw new IOException();
          }
          else
          {
             String newSequence;
             myStructure = "";
             do
             {
               newSequence = inputStream.readLine();
               if (!(newSequence.matches(".*[^ACGU].*")))
               {
                  mySequence.concat(newSequence);
               }
```

```
      else
      {
         myStructure = newSequence;
      }
   }while(!(newSequence.matches(".*[^ACGU].*")));

}

// At this point, the sequence is good
// Now let’s test the structure to see
// if it contains something other than
// dots or brackets
if (myStructure.matches(".*[^\\.\\(\\)].*"))
{
   throw new IOException();
}
else
{
   String newStructure;
   do
   {
      newStructure = inputStream.readLine();
      if (newStructure != null)
      {
         if (newStructure.matches(".*[^\\.\\(\\)].*"))
         {
            throw new IOException();
         }
         else
         {
            myStructure.concat(newStructure);
         }
      }
      else
      {
         newStructure = "unmatchingString";
      }
   }while(!(newStructure.matches(".*[^\\.\\(\\)].*")));
}

// If we made it here, all lines are
// valid, let’s process them

Stack structureStack = new Stack();
int[] basePairArray = new int[mySequence.length()];
// Initialize the array
for (int i=0; i<mySequence.length(); i++)
{
   basePairArray[i] = -1;
}//end for (int i=0; i<basePairArray; i++)

// Go through the sequence and match
// brackets
for (int i=0; i<mySequence.length(); i++)
{
   // Whenever it’s an open bracket,
   // push that base number into the
   // stack
   if (myStructure.charAt(i) == ’(’)
   {
      structureStack.push(i);
   }
   // Whenever it’s a closing bracket,
   // pop the base number from the stack
   // and update both bases to be bonded
   // to each other via the
   // basePairArray
   else if (myStructure.charAt(i) == ’)’)
```

```
          {
            try
            {
               int bondedBase = ((Integer)structureStack.pop()).intValue();
               basePairArray[bondedBase] = i;
               basePairArray[i] = bondedBase;
            }
            catch (EmptyStackException e)
            {
               System.out.println("Invalid fasta file: "+this.getPath());
               System.out.print("Check that the brackets match in ");
               System.out.println("the structure description");
               System.out.println(e.getMessage());
               return null;
            }
          }
       }

       ArrayList<Nucleotide> nucList = new ArrayList<Nucleotide>();
       Nucleotide nuc;
       int basePairNumber = mySequence.length();
       int k=0;
       for (int i=0; i<basePairNumber; i++)
       {
          nuc = new Nucleotide(i,
                      ""+mySequence.charAt(i),
                      basePairArray[i]);
          nucList.add(nuc);
       }//end for (int i=0; i<basePairNumber; i++)

       RnaStructure returnStructure = new RnaStructure(nucList,
                                  this.getName(),
                                  myHeader);
       returnStructure.setCoorFile(myCoordinateFile);

       return returnStructure;
     }
     catch (IOException e) {
       System.out.println("Invalid fasta file: "+this.getPath());
       System.out.println(e.getMessage());
       return null;
     }
  }

  public boolean writeFile(ArrayList<Nucleotide> structure)
  {
     return false;
  }//end writeFile(ArrayList<Nucleotide> structure)

  @Override
  public int getType()
  {
     return FileUtils.FASTA;
  }
}
```

Bibliography

[1] The PyMOL Molecular Graphics System, 2010.

[2] Jean-Pierre Bachellrie, Jerome Cavaille, and Alexander Huttenhofer. The Expanding snoRNA World. *Biochimie*, 84(8), 2002.

[3] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[4] David P. Bartel. MicroRNAs: Target Recognition and Regulatory Functions. *Cell*, 2009.

[5] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multi-threaded programming for C/C++. In *ACM SIGPLAN Notices - OOPSLA '09*. ACM, 2009.

[6] Nicolas Boghossian, Oliver Kohlbacher, and Hans-Peter Lenhof. Ball: Biochemical algorithms library. *Algorithm Engineering*, pages 330–344, 1999.

[7] Robert E. Broccoleri and Gerhard Heinrich. An Improved Algorithm for Nucleic Acid Secondary Structure Display. *Bioinformatics*, 4(1):167–173, 1988.

[8] Cannone JJ, Subramanian S, Schnare MN, Collet JR, D'Souza LM, Du Y, Feng B, Lin N, Madabusi LV, Muller KM, Pande N, Shang Z, Yu N, and Guttel RR. The comparative rna web (crw) site: an online database of comparative sequence and structure information for ribosomal, intron, and other rnas. *BMC Bioinformatics*, 3(1):15, July 2002.

[9] Kevin Darty, Alain Denise, and Yann Ponty. Varna: Interactive drawing and editing of the rna secondary structure. *Bioinformatics*, 25(15), 2009.

[10] P De Risjk and R De Wachter. Rnaviz, a program for the visualisation of rna secondary structure. *Nucleic Acids Research*, 25(22):4679–4684, 1997.

[11] European Bioinformatics Institue. Clustalw. http://www.ebi.ac.uk/Tools/clustalw2/index.html, 2010.

[12] Hin Hark Gan, Samuela Pasquali, and Tamar Schlick. Exploring the repertoire of rna secondary motifs using graph theory; implications for rna design. *Nucleic Acids Research*, 31(11):2926–2943, 2003.

[13] GeneLink. What is RNAi and siRNA? http://www.genelink.com/sirna/RNAiwhatis.asp#section2.

[14] Edward Glen. JVIZ.RNA - A TOOL FOR VISUAL COMPARISON AND DNALYSIS OF RNA SECONDARY STRUCTURES. Master's thesis, Simon Fraser University, 2007.

[15] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Cuncurrency In Practice*. Addison-Wesley Professional, May 2006.

[16] Kyungsook Han and Yanga Byun. Pseudoviewer2: visualization of rna pseudoknots of any type. *Nucleic Acids Research*, 31(13):3432–3440, 2003.

[17] Angel Herráez. Biomolecules in the computer: Jmol to the rescue. *Biochemistry and Molecular Biology Education*, 34(4):255–261, 2006.

[18] Ivo L. Hofacker. Vienna RNA secondary structure server. *Ivo L. Hofacker*, 31(13):3429–3431, 2003.

[19] Fabrice Jossinet and Eric Westhof. *The RnamlView Project*. Institut de biologie moleculaire et cellulaire du CNRS.

[20] Bohyoung Kim, Bongshin Lee, Susan Knoblach, Eric Hoffman, and Jinwook Seo. Geneshelf: A web-based visual interface for large gene expression time-series data repositories. *IEEE Transactions on Visualization and Computer Graphics*, 15:905–912, 2009.

[21] Oracle. Class File. http://download.oracle.com/javase/1.3/docs/api/java/io/File.html#createNewFile().

[22] Oracle. Interface Runnable. http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Runnable.html.

[23] John K. Ousterhout. *Tcl and the Tk Toolkit*. Department of Electrical Engineering and Computer Sciences, University of California, 1993.

[24] Eric F Pettersen, Thomas D Goddard, Conard C Huang, Gregory S Couch, Daniel M Greenblatt, Elaine C Meng, and Thomas E Ferrin. Ucsf chimera—a visualization system for exploratory research and analysis. May 2004.

[25] Yair Sade, Mooly Sagiv, and Ran Shaham. Optimizing C Multithreaded Memory Management Using Thread-Local Storage. In *Lecture Notes in Computer Science*, volume 3443/2005. Springer, 2005.

[26] Schrödinger, LLC. The PyMOL Molecular Graphics System, Version 1.3rl. PyMOL The PyMOL Molecular Graphics System, Version 1.3, Schrödinger, LLC., August 2010.

[27] Boris Shabash. Improving the Processing Speed of jViz.RNA. August 2010.

[28] The W3C DOM WG. Document Object Model FAQ. http://www.w3.org/DOM/faq.html.

[29] W3C. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/xml/.

[30] Kay C. Wiese and Edward Glen. jViz.Rna -a java tool for RNA secondary structure visualization. *NanoBioscience, IEEE Transactions on*, 4(3):212–218, September 2005.

[31] Kay C. Wiese and Edward Glen. jViz.Rna - An Interactive Graphical Tool for Visualizing RNA Secondary Structure Including Pseudoknots. In *CBMS*, pages 659–664. IEEE Computer Society, 2006.

[32] Zuker M. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415, July 2003.