A Predictive Model Which Uses Descriptors of RNA Secondary Structures Derived from Graph Theory

---

A thesis

presented to

the faculty of the Department of Mathematics and Statistics

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Mathematical Sciences

---

by

Alissa A. Rockney

May 2011

Debra Knisley Ph.D., Chair

Jeff Knisley, Ph.D.

Teresa Haynes, Ph.D.

Keywords: Graph Theory, RNA, Neural Network, Graphical Invariants

ABSTRACT

A Predictive Model Which Uses Descriptors of

RNA Secondary Structures Derived from Graph Theory

by

Alissa A. Rockney

The secondary structures of ribonucleic acid (RNA) have been successfully modeled with graph-theoretic structures. Often, simple graphs are used to represent secondary RNA structures; however, in this research, a multigraph representation of RNA is used, in which vertices represent stems and edges represent the internal motifs. Any type of RNA secondary structure may be represented by a graph in this manner. We define novel graphical invariants to quantify the multigraphs and obtain characteristic descriptors of the secondary structures. These descriptors are used to train an artificial neural network (ANN) to recognize the characteristics of secondary RNA structure. Using the ANN, we classify the multigraphs as either RNA-like or not RNA-like. This classification method produced results similar to other classification methods. Given the expanding library of secondary RNA motifs, this method may provide a tool to help identify new structures and to guide the rational design of RNA molecules.

Copyright by Alissa A. Rockney 2011

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. Debra Knisley, my thesis chair, for providing the idea for my thesis project. I am also thankful for her guidance at various stages of this project and for her patience in working with me. I am also grateful to Dr. Jeff Knisley for his provision of neural networks and assistance with them. I would also like to thank Dr. Teresa Haynes for serving on my committee and assisting me with the revision process. The completion of this project would have been impossible without the RAG: RNA-as-Graphs Webpage; I am grateful to Tamar Schlick and her research associates at NYU for the creation and maintenance of this database. I would also like to thank Trina Wooten and undergraduate Chelsea Ross for their assistance in the beginning stages of this research.

Finally, I would like to thank all of the different “families” I have. Without the support of my parents, brother, and sister, I would not be where I am today. I'm grateful also for the people in my Kroger family, who have encouraged me all the way through college. I'm thankful for all of my fellow students and faculty at ETSU, and all of the friendships that have been formed there. I'd also like to thank my family at Christ Community Church for their encouragement and for pointing me back to Jesus when I forget what's important in life.

CONTENTS

ABSTRACT 2

ACKNOWLEDGMENTS 4

LIST OF TABLES 6

LIST OF FIGURES 7

1 BACKGROUND 8

1.1 RNA 8

1.2 Basic Graph Theory Terminology 9

1.3 RNA Terminology and the RAG Database ([18]) 11

2 METHODS 16

2.1 Overview 16

2.2 Graphical Invariants 16

2.3 Artificial Neural Network 20

2.3.1 Training Set 22

2.3.2 Classification and Cross-Validation 23

3 RESULTS AND DISCUSSION 26

4 CONCLUSIONS 29

BIBLIOGRAPHY 31

APPENDICES 35

VITA 68

LIST OF TABLES

1 Number of Possible Dual Graphs of Orders $2-9$ [18] 13

2 Dual Graphs of Order 4 and Their RAG Color Classes [18] 15

3 Comparison of RAG Results with t Confidence Intervals 27

4 Graphical Invariants $1-3$ [18] 65

5 Graphical Invariants $4-7$ [18] 66

6 Graphical Invariants $8-12$ [18] 67

LIST OF FIGURES

1 Illustration of Some Basic Graph Theory Terminology 10

2 Dual Graph 4.04 and its Line Graph 10

3 Red Dual Graphs in Training Set [18] 23

4 Black Dual Graphs in Training Set [18] 24

5 ROC Analysis of Original ANN 26

6 The Dual Graphs with RAG IDs 4.12 and 4.29, Respectively 28

1 BACKGROUND

Until recently, ribonucleic acid (RNA) has been understood to have essentially one purpose: to facilitate the construction of proteins. The discovery of several additional types of RNA (non-coding RNA, or ncRNA), whose functions are still under investigation, has motivated new avenues of research in computational biology.

1.1 RNA

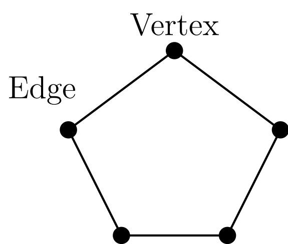
RNA has three basic structures - primary, secondary, and tertiary. The primary structure is the sequence of nucleotides (adenine (A), cytosine (C), guanine(G), and uracil (U)). The primary structure folds back on itself to form the secondary structure, on which we focus in the present research. When the sequence folds, different combinations of nucleotides (A-U and C-G) bond with each other, forming Watson-Crick base pairs [10]. In some situations, one of four structures may be formed when the nucleotides bond; these structures are *hairpin loops* ("unmatched bases in single-strand turn region of a helical stem of RNA" [18]), *bulges and internal loops* (instances of unmatched base pairs appearing in a helical stem [18]) and *junctions* ("three or more helical stems converging to form a closed structure" [18]). The tertiary structure is formed when secondary structures fold back on themselves. Fortunately, RNA secondary structures have been found to be indicative of their functions [8]; thus, we study secondary structures of RNA in this research.

1.2 Basic Graph Theory Terminology

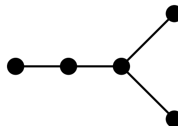
In previous research involving RNA secondary structures, the field of mathematics known as *graph theory* has been used to model these structures [5, 10, 18, 21]. Before explaining in more detail how graph theory is used to represent RNA secondary structures, we give some basic graph theory definitions (for more information, as well as any undefined notations, see [25]). A *graph* G is comprised of a *vertex set* $V(G)$ and an *edge set* $E(G)$. We call a graph G with $|V(G)| = n$ a graph of *order* n. A graph G with $|E(G)| = m$ has *size* m. Each edge in $E(G)$ is necessarily paired with at least one vertex in $V(G)$ such that each edge has two endpoints (note that the edge may begin and end with the same vertex; in this case, the edge is called a *loop*) [25]. We represent *vertices* as dots and *edges* as lines. It is possible that more than one edge may have the same pair of vertices as endpoints; in this case, we call the edges *multiple edges*. A graph which contains no loops or multiple edges is known as a *simple graph*. A graph which may contain both loops and multiple edges is called a *multigraph*. It is possible that a graph G may contain vertices which do not serve as endpoints for any edges in G; in this case, G is *disconnected*. In the present research, we consider only *connected* graphs.

In a graph G, all edges which have the vertex v as an endpoint are called the *incident* edges of v. The *degree* of a vertex v, denoted $deg_G(v)$, is its number of incident edges. A vertex u which has only one incident edge is called a *leaf*, and $deg_G(u) = 1$. For example, let G be the graph with $V(G) = \{u, v\}$ and $E(G) = \{uv\}$. Then $deg_G(u) = deg_G(v) = 1$. A *cycle* C_k is a simple graph which contains as many vertices as it does edges, and which contains vertices $V(C_k) = \{1, 2, \ldots, k\}$. Then

A Simple Graph C_5



Tree Graph of Order 5



A Dual Graph (RAG ID 4.13)

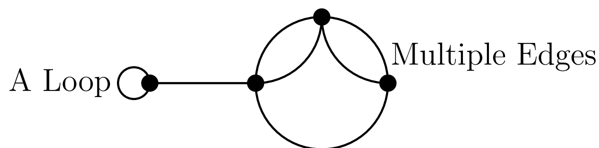


Figure 1: Illustration of Some Basic Graph Theory Terminology

$E(C_k) = \{12, 23, 34, \ldots, (k-1)k, k1\}$. Notice also that for all vertices v in $V(C_k)$, $deg_{C_k}(v) = 2$. A simple graph which does not contain a cycle is known as a *tree* [25]. The *line graph* of a graph G, denoted $L(G)$, is formed by first labeling the edges of G, then using these labels to label the vertices of $L(G)$. The vertices of $L(G)$ are adjacent if the edges of the same name in G are adjacent to each other.

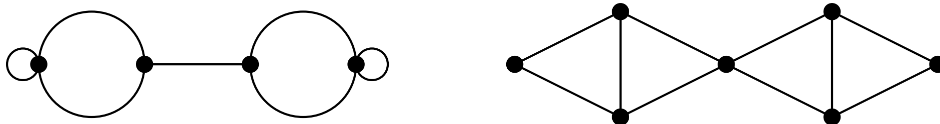


Figure 2: Dual Graph 4.04 and its Line Graph

1.3 RNA Terminology and the RAG Database ([18])

In [18], both trees and multigraphs (called *dual graphs* here) are used to represent RNA secondary structures. In this research, we focus specifically on dual graph representations of RNA structures. Dual graphs can represent all types of RNA secondary structures because of their capability to represent *pseudoknots*, which are a type of RNA motif formed when nucleotides in a hairpin loop bond with nucleotides outside the hairpin loop [18]. The simplicity of tree graphs prevents them from accurately representing RNA pseudoknots. Since dual graphs can represent both trees and pseudoknots, as well as a third type of RNA motif known as a *bridge* ("RNAs with stems connected by a single strand" [18]) which also cannot be represented with a tree graph, dual graphs are of particular interest. To represent an RNA secondary structure with a dual graph, we place a vertex everywhere there is a *stem* (that is, a structure containing more than one complementary base pair) in the secondary structure. If the RNA secondary structure contains a hairpin loop, we represent this with a loop in the form of a circular edge. Bulges and internal loops are represented by double edges between two vertices. Junctions are also drawn as circular edges, and may appear in a dual graph as either a set of double edges, a C_3, or a C_4. [18]

Dual graphs are allowed to have loops and multiple edges, but not required to have either. There are other constraints placed on dual graphs as well; for example, every vertex v in a dual graph must have at most $deg_G(v) = 4$. Also, dual graphs contain no leaves. A property of dual graphs noted by Schlick et al. is that if a graph G is order n, then G has size $2n - 1$ [18].

Given the constraints on the number of edges, minimum degree, and maximum

degree for a dual graph of order n, it is possible to count the number of possible structures of dual graphs of order n. Schlick et al. have done this via two different methods: a probabilistic method [19] and by $k - NN$ clustering [18]. In addition, Schlick et al. have given the number of *RNA motifs* (i.e., RNA secondary structures which correspond to a dual graph in the database) which have been found in nature [18]. Table 1 describes the most recent calculations of the number of possible dual graphs of orders $2 - 9$, as well as the number of RNA motifs which have been found thus far [18]. The RNA motifs which have been found in nature are colored red in [18]. The motifs which are thought to be RNA-like by Schlick et al. are colored blue, and those thought to be not-RNA-like are colored black [18]. These graphs are ordered by the value of the second-smallest eigenvalue λ_2 of their normalized Laplacian matrix [3, 11, 18]. They may be referred to by a label of the form $n.r$, where n is the order of G and r is the rank in list of values of λ_2 for graphs of this order. For example, the dual graph whose corresponding value of λ_2 is the third largest in the list of all values of λ_2 for order four dual graphs would be labeled 4.3 (note that for the purposes of the present research, in order to distinguish the label 4.3 from the label 4.30 in software, we label this graph 4.03). Since there are a total of thirty dual graphs of order four, the labels range from 4.01 to 4.30. Until November 24, 2010, the RAG classifications for some of the graphs of order four were different; the present research reflects the current classifications [18, 19]. See Table 2 for a comparison of the previous classifications (denoted "Old Rag Color Class") and the current classifications (denoted "New RAG Color Class") [18, 19].

These motifs are often found in online RNA databases such as [9, 15] and those

Table 1: Number of Possible Dual Graphs of Orders $2-9$ [18]

Dual Graph Order	Predicted Number of Dual Graphs	Number of Motifs Found
2	3	3
3	8	8
4	30	17
5	108	18
6	494	12
7	2388	6
8	12184	3
9	38595	4

which may be found in [27]. Also, online RNA-folding algorithms have been used to identify RNA secondary structures (see [27] for a list of these RNA folding algorithms). These algorithms usually fold sequences into secondary structures based on the resulting free energy of the secondary structure, which has been shown to be indicative of the structure's stability [2, 7]. Nonetheless, this is not the only way in which RNA structure has been predicted. RNA pseudoknots have proven difficult to predict in some cases (see [17, 26] for more information). Additionally, it has been shown that graphical invariants, which can be calculated mathematically and highlight specific properties of graphs, are also indicative of RNA secondary structure [5, 6, 10]. That is, combinations of invariants which describe red and blue graphs are different from combinations of invariants which describe black graphs. In [6], a statistical analysis, combined with graphical invariants involving domination (see 2 for a definition of the domination number of a graph G), showed this result. Also, in [5], a neural network trained with invariants from known RNA graphs was able

to distinguish invariants quantifying RNA-like graphs from those quantifying not-RNA-like graphs. Also, in [10], a neural network was able to distinguish RNA-like graphical invariants involving the merging of two trees from those from not-RNA-like trees; however, in all of these previous studies, RNA tree graphs were studied. In this paper, we focus on RNA dual graphs. We redefine widely known graphical invariants for simple graphs so that they may be used with multigraphs, and we use them to quantify the dual graphs of order four in the RAG database [18].

Table 2: Dual Graphs of Order 4 and Their RAG Color Classes [18]

RAG ID	Old RAG Color Class	New RAG Color Class
4.01	Blue	Red
4.02	Blue	Red
4.03	Black	Black
4.04	Black	Red
4.05	Black	Red
4.06	Black	Black
4.07	Black	Black
4.08	Black	Red
4.09	Red	Red
4.10	Blue	Blue
4.11	Black	Black
4.12	Blue	Blue
4.13	Black	Black
4.14	Red	Red
4.15	Red	Red
4.16	Blue	Red
4.17	Red	Red
4.18	Black	Black
4.19	Red	Red
4.20	Red	Red
4.21	Black	Black
4.22	Red	Red
4.23	Blue	Black
4.24	Black	Black
4.25	Red	Red
4.26	Blue	Black
4.27	Red	Red
4.28	Red	Red
4.29	Black	Red
4.30	Blue	Blue

2 METHODS

2.1 Overview

First, we operationally defined several graphical invariants for use with multigraphs. We then calculated these invariants for each of the thirty dual graphs of order four in the RAG database. Also, we found the line graph of each dual graph and calculated other invariants of these line graphs (which are simple graphs). We used a subset of these invariants (for each dual graph or line graph corresponding to the dual graph) to train an artificial neural network (ANN) to recognize RNA-like and not-RNA-like sets of invariants. We then classified the RNA-like dual graphs in the RAG database using the ANN and compared our results with those obtained by Schlick et al. [18]. Also, using a combinatorial method, we enumerated the dual graphs of order five.

2.2 Graphical Invariants

We calculated a number of graphical invariants for each order four dual graph in the RAG database. Many invariants are familiar to graph theorists; however, others were redefined for use with multigraphs. Several other invariants were derived from those used often in chemical graph theory [23, 21]. Some invariants were also calculated using the line graphs of the dual graphs of order four.

We list the definitions of the invariants used with the dual graphs below. Unless otherwise noted, let G be a dual graph from the RAG database [18]. Also, as it is used in several invariant definitions, we now define the *distance matrix* of a dual graph G

to be the matrix $D_{i,j}$ where entry $d_{ij} = d_{ji}$ is the distance from vertex i to vertex j for vertices $i, j \in V(G)$ [23]. If vertex k in $V(G)$ has a loop, then $d_{kk} = 1$. If there are n edges between vertices l and m in $V(G)$, then $d_{lm} = d_{ml} = 1/n$.

Dual Graph Invariants

1. [23] Let q be the size of G. Let $\mu(G)$ be the number of cycles in G. Define $s_i = \sum_{j=1}^{n}(d_{ij})$ to be the sum of all distances in row i of $D_{i,j}$. Then the *Balaban Index* is defined as $J(G) = \frac{q}{\mu(G)+1}\sum_{i=1}^{n}(s_i)^{(-1/2)}$.

2. Let a *cycle* c_{ii} in G be a walk on the vertices of G such that if we begin at vertex $i \in V(G)$, we also end at vertex i. If $c_{ii} = 121$, for example, we count this cycle once, even if there are multiple ways to traverse that cycle. Then the *number of cycles* in G is the number of all unique cycles c_{ii} in G. When calculating the number of cycles, we counted a loop as one cycle. We denote the number of cycles of G as $N_c(G)$.

3. [25] The *diameter* of G is commonly defined to be $diam(G) = max_{i,j \in V(G)}\{d_{ij}\}$.

4. [25] The *girth* of G is the smallest induced cycle of G, where a cycle is as defined in 2. We do not count loops in this case. If G has no cycle, we say that the girth of G is 0 for calculation purposes. (Note that in most instances, an acyclic graph has girth ∞.) We shall denote the girth of G as $g(G)$.

5. [23] The *Wiener number* is defined as $W(G) = 1/2\sum_{i,j=1}^{n}(d_{ij})$ where $i \neq j$.

6. [25] The *edge chromatic number* of a dual graph G is the size of a proper edge coloring (i.e., the minimum number of colors required such that no two adjacent

edges are the same color) of G. We denote the edge chromatic number of G as $\chi'(G)$.

7. [23] Define the *edge degree*, denoted $D(e_i)$, of an edge e_i in $E(G)$ as the number of edges adjacent to e_i. If e_j is a loop edge in $E(G)$, then we say that $D(e_j) = 1$. Then the *Platt number* $F(G)$ is defined to be $F(G) = \sum_{i=1}^{m} D(e_i)$, where m is the size of G.

8. [1] Define the *Randić Index* $R(G)$ as $R(G) = \sum_{ij \in E(G)} \frac{1}{\sqrt{deg_G(i) deg_G(j)}}$. When calculating $R(G)$, we count each multiple edge; that is, if there are three edges ij in $E(G)$, we have a term for each edge ij in the sum $R(G)$.

9. [25] The *domination number* $\gamma(G)$ is defined (for a simple graph G) to be the minimum size of a set $S \subseteq V(G)$ where S is a *dominating set* if each vertex in $V(G) \setminus S$ is adjacent to a member of S. For dual graphs, we define $\gamma(G)$ to be the sum of edges incident to vertices in S, where each incident edge is counted once. Also, if S contains a vertex with a loop edge, we count the loop edge only once.

10. [25] A *matching* in G is defined as a set S of edges in G such that if edge ij is in S, then no other edge in S is incident to either vertex i or vertex j. The *maximum size of a matching* is the largest such set S.

We now define the graph invariants we calculated for the line graphs of the dual graphs of order four. Note that all graphs $L(G)$ are simple graphs.

Line Graph Invariants

11. [25] The *number of edges* of $L(G)$ is simply $|E(L(G))|$.

12. [25] The *maximum degree* of $L(G)$ is the maximum number of incident edges of a vertex i in $V(L(G))$. We denote the maximum degree of a vertex in $V(L(G))$ as $max\{deg_{L(G)}(i)\}$.

13. [25] The *vertex chromatic number* of $L(G)$ is the size of a proper coloring of $L(G)$ (i.e., the minimum number of colors required such that no two adjacent vertices are the same color). The vertex chromatic number of $L(G)$ is denoted $\chi(L(G))$.

14. [25] A *clique* in $L(G)$ is "a set of pairwise adjacent vertices" [25]. The *clique number* of $L(G)$ is the size of the largest such set in $V(L(G))$. We denote the clique number of G as $\omega(G)$.

15. [25] The *edge chromatic number* of $L(G)$, denoted $\chi\prime(L(G))$, is the size of a proper edge coloring (i.e., the minimum number of colors required such that no two adjacent edges are the same color) of $L(G)$.

16. [25] The *diameter* of $L(G)$ is the maximum distance d_{ij} between vertices $i, j \in V(L(G))$. In $L(G)$, we define d_{ij} as the minimal number of edges which must be traversed to reach vertex j from vertex i (or vice-versa). We denote the diameter of $L(G)$ as $diam(L(G))$.

17. [25] The *girth* of $L(G)$ is the smallest induced cycle in $L(G)$, where a cycle is as defined in Section 1.2. We will denote the girth of $L(G)$ as $g(L(G))$.

18. [25] The *edge connectivity* of $L(G)$, denoted $\kappa'(L(G))$, is the minimum number of edges which must be removed from $E(L(G))$ in order for $L(G)$ to become disconnected.

19. [25] The *vertex connectivity* of $L(G)$ is the minimum number of vertices which must be removed from $V(L(G))$ in order for $L(G)$ to become disconnected. Vertex connectivity is denoted $\kappa(L(G))$.

20. [25] The *independence number* of $L(G)$ is the size of the largest set of pairwise nonadjacent vertices (i.e., the size of the largest *independent set*) in $L(G)$ [25]. We denote the independence number as $\alpha(L(G))$.

2.3 Artificial Neural Network

Once all invariants were calculated, twelve invariants were chosen for which the values of the invariant for dual graphs that were red in the RAG database were noticeably different from those for dual graphs that were black [18]. Other invariants that were calculated did not appear to discriminate well between red and black dual graphs, and thus were not included in the analysis. The invariants chosen were as follows:

1. The Balaban Index of G ($J(G)$)

2. The number of cycles of G ($N_c(G)$)

3. The diameter of G ($diam(G)$)

4. The number of edges of $L(G)$ ($|E(L(G))|$)

5. The maximum degree of a vertex in $L(G)$ $(max\{deg_{L(G)}(i)\})$

6. The chromatic number of $L(G)$ $(\chi(L(G)))$

7. The girth of G $(g(G))$

8. The Wiener number of G $(W(G))$

9. The edge chromatic number of G $(\chi'(G))$

10. The Platt number of G $(F(G))$

11. The Randić Index of G $(R(G))$

12. The domination number of G $(\gamma(G))$

The values of these invariants were entered into Minitab 14 ([16]). Each of these invariants was then normalized. We now define normalization for a generic invariant, call it t. Let R be the label used in the RAG database for each graph of order four [18]. Denote the value of invariant t for graph R as t_R. We first find the mean $\overline{x}_t$ and standard deviation s_t of t. We then calculate z_t, where

$$z_t = \frac{t_R - \overline{x}_t}{s_t}.$$

We call z_t the *normalized* value of the invariant t for the dual graph R.

Once each of the twelve invariants were normalized, a vector was created using the normalized values of the invariants for each dual graph. This vector was of the form

$$< z_{J(G)}, z_{N_c(G)}, z_{diam(G)}, z_{|E(L(G))|}, z_{max\{deg_{L(G)}(i)\}}, z_{\chi(L(G))}, z_{g(G)}, z_{W(G)}, z_{\chi'(G)}, z_{F(G)}, z_{R(G)}, z_{\gamma(G)} >_R .$$

We constructed a vector of this form for each dual graph R in order to enter this vector into a multi-layer perceptron (MLP) artificial neural network (ANN). The ANN was trained using a back-propagation algorithm [5, 10, 22]. The ANN used in the present research contained three layers - an input layer, a hidden layer, and an output layer. Each layer contains *perceptrons*, or nodes, which act like artificial neurons in that each perceptron contains an activation function (or sigmoid function [4]). This design makes each perceptron similar to an actual neuron in that the activation function in a perceptron, or artificial neuron, simulates the action potentials in an actual neuron [4, 22]. In the present research, the *input layer* contains twelve perceptrons, one for each graph invariant in the input vector. The *hidden layer* contains sixteen perceptrons, and the *output layer* contains two perceptrons, one for each component in the output vector $< a, b >$.

2.3.1 Training Set

To train the ANN, we randomly chose a subset of ten red dual graphs and used all ten black dual graphs. The purpose of training the ANN is so that it can recognize graphs which are similar to either the red graphs or the black graphs which are in the TS. Furthermore, once the ANN is trained, it should classify dual graphs as RNA-like or not-RNA-like. Thus, in the TS, each of the twenty graphs was denoted by a set of two vectors. The first vector was a vector of invariant values as defined in Section 2.3, and the second vector was $< 1, 0 >$ if dual graph R was red in the RAG database, and $< 0, 1 >$ if dual graph R was black in the RAG database [18]. It should be noted that no blue dual graphs were used in the TS.

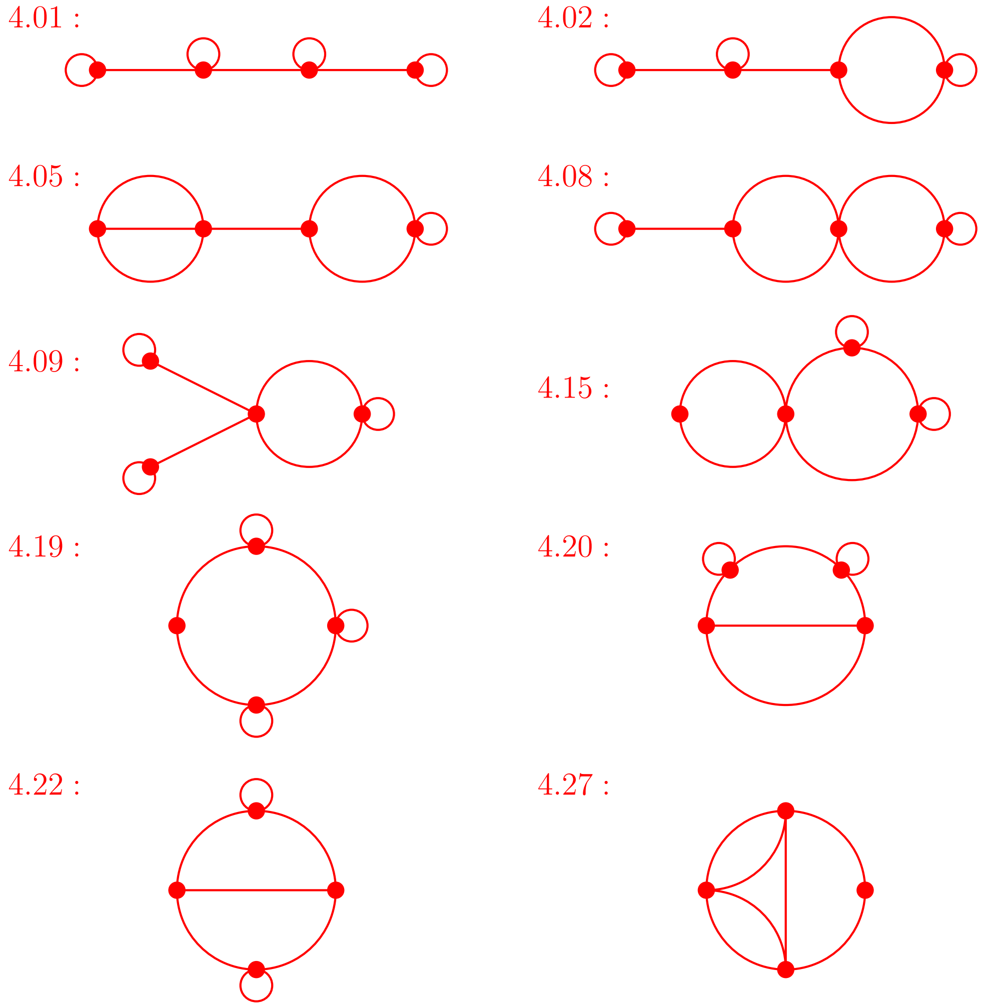


Figure 3: Red Dual Graphs in Training Set [18]

2.3.2 Classification and Cross-Validation

Once the ANN was trained, the invariant vectors for the remaining ten dual graphs were given to the ANN for classification. The neural network output was in the form of a vector $< a, b >$ where $0 \leq a \leq 1$ and $0 \leq b \leq 1$. Both a and b are probabilities; if a is closer to 1, then the dual graph has been classified by the ANN as RNA-like, and if b is closer to 1, then the dual graph is classified as not-RNA-like. The

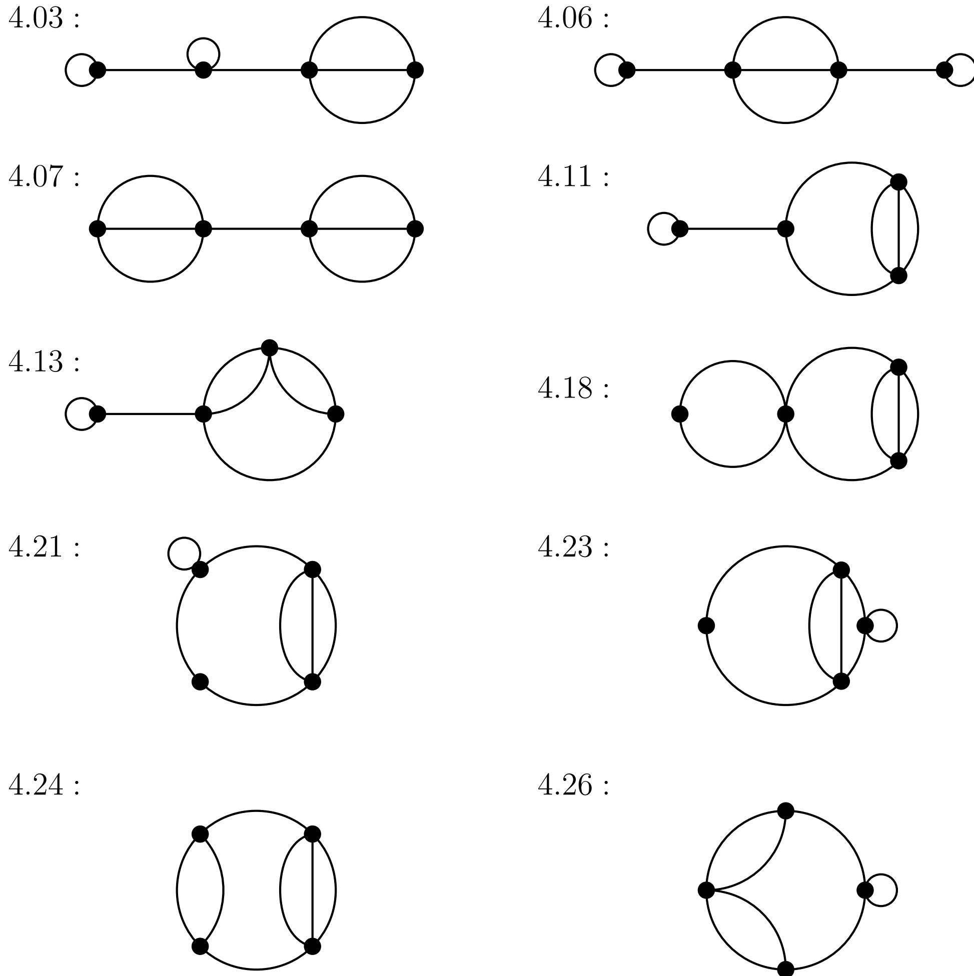


Figure 4: Black Dual Graphs in Training Set [18]

ANN's performance was evaluated using a receiver operating characteristic (ROC) curve. ROC curves are a plot of sensitivity, or true positive rate, versus specificity, or false positive rate [12]. Ideally, the sensitivity will be 1, and the specificity will be 0; this situation gives the maximum area under the ROC curve (known as the AUC), which is 1. The threshold which gives the maximum AUC is also reported in the results of the ROC analysis. In the present research, an ROC analysis was performed each time the ANN was trained [12].

To test the results, a method known as cross-validation, or leave-v-out cross validation [5, 10, 20], was implemented. A vector v was removed from the TS, the network was trained on the new TS, and the dual graphs not in the TS were classified using the newly trained ANN. This procedure was conducted for each vector v in the TS; furthermore, each time this was done, three different training lengths were used. Thus, sixty vectors $< a, b >$ were obtained for each dual graph not in the original TS. The a values were used to develop a t-confidence interval (computed using the statistical software Minitab 14 [16]) for the value of a for each dual graph. Also, since an ROC analysis was performed for each training of the ANN, a confidence interval was produced for the threshold which gave the maximum AUC [12].

3 RESULTS AND DISCUSSION

The TS consisted of ten randomly selected RNA motifs (graphs with RAG ID 4.01, 4.02, 4.05, 4.08, 4.09, 4.15, 4.19, 4.20, 4.22, and 4.27 [18]) and all ten black dual graphs from the RAG database (graphs 4.03, 4.06, 4.07, 4.11, 4.13, 4.18, 4.21, 4.23 4.24, and 4.26 [18]). First, the neural network was trained using this training set, and then the vectors corresponding to dual graphs not in the TS were classified by the network as RNA-like or not-RNA-like. The network converged in approximately thirty training sessions, with an AUC of 1 and a threshold of 0.05, as shown in the figure below.

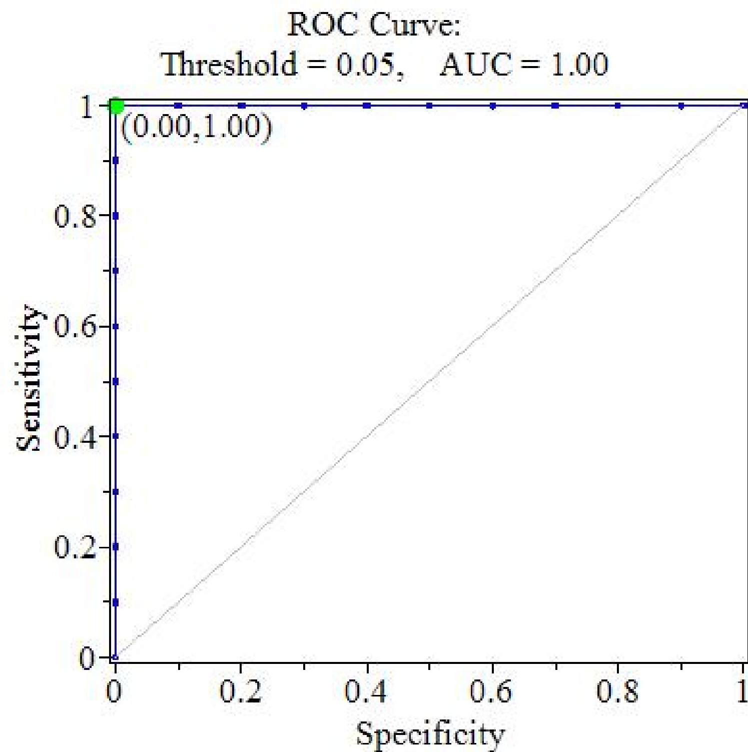


Figure 5: ROC Analysis of Original ANN

We then implemented leave-v-out cross-validation, as described above. Each modified TS was trained for thirty, thirty-two, and thirty-five training sessions. Higher numbers of training sessions tended to produce a phenomenon known as over-fitting, in which graphs are not classified correctly simply due to overtraining [13]. For each of the sixty ROC analyses performed, the AUC was 1. With 95 confidence, the threshold from the ROC analysis fell between (.030333, .036000). The t-confidence intervals for the values of a for each dual graph are reported in Table 3.

Table 3: Comparison of RAG Results with t Confidence Intervals

RAG ID	RAG Class	Mean Value of a	Standard Deviation of a	t CI for Value of a
4.04	Red	0.99997	0.0000321	(0.999957, 0.999974)
4.10	Blue	0.99729	.00342	(0.996404, 0.998173)
4.12	Blue	0.5615	0.3175	(0.479504, 0.643549)
4.14	Red	0.9414	0.1235	(0.908476, 0.972308)
4.16	Red	0.99874	0.00170	(0.998298, 0.999175)
4.17	Red	0.9045	0.1596	(0.863227, 0.945673)
4.25	Red	0.9120	0.2138	(0.856754, 0.967236)
4.28	Red	0.9749	0.1091	(0.946760, 1.003117)
4.29	Red	0.6771	0.2908	(0.601964, 0.752204)
4.30	Blue	0.99695	0.01454	(0.993194, 1.000707)

Thus, since the threshold for the ROC analysis was so low, our results are similar to those found by Schlick et al. [18]. It should be noted that the network did not classify the dual graphs with RAG IDs 4.12 and 4.29 with as much accuracy as the other dual graphs. In the RAG database, 4.12 is classified as RNA-like, but no actual RNA secondary structures have yet been found which can be modeled using this dual graph. The methods used here did classify graph 4.12 as RNA-like, but not to the

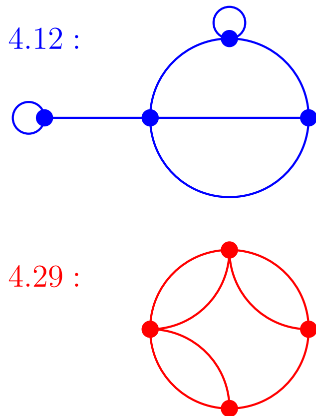


Figure 6: The Dual Graphs with RAG IDs 4.12 and 4.29, Respectively

same extent as the other dual graphs. Graph 4.29 is classified as an RNA motif (as of November 24, 2010) [18]. However, prior to that date, graph 4.29 was classified by the RAG database as a black graph (i.e., as not-RNA-like) [19]; this may partially explain why the methods used in the present research did not classify 4.29 as RNA-like with as much accuracy as other graphs.

4 CONCLUSIONS

The approach taken in the present research followed the design of research previously conducted [5, 6, 10] in that RNA secondary structures were quantified using graphical invariants; however, this research is novel in that the definitions of graphical invariants were modified for use with multigraphs. The invariants used here have been used in previous research (e.g., [5, 6, 21]) to effectively distinguish between graphical structures which are RNA-like and those which are not. The approach of training an artificial neural network to classify dual graphs as RNA-like or not-RNA-like using graphical invariants was fairly successful, though not as accurate for two of the dual graphs as perhaps is desired.

Some possible explanations for the less-than-desired accuracy of the classifications of graphs 4.12 and 4.29 were given in the previous section; however, another possible reason may be the small size of the training set. There are only thirty dual graphs of order four. Further research could improve upon the results presented here by calculating invariants for dual graphs of other orders in order to increase the size of the training set.

The results from the present work imply that this method, with some improvements, could be quite useful in RNA secondary structure prediction. Another possible area of application of this research is that of rational drug design [5, 24]. In rational drug design, the structures of drug receptors are analyzed, and then candidate drugs are designed based on the structure that would best fit inside the structure of the receptor [24]. Additionally, the structures of existing drugs/compounds may be analyzed, and those drugs whose structures are most likely to fit into the drug receptor

are tested [24]. Since RNA structure is often indicative of RNA function [8], these methods could be useful in the rational design of drugs [5].

Also, the results presented here serve as further evidence that it may not be necessary to consider the free energy (e.g., see [2]) of graphical structures in order to classify them as RNA-like or not-RNA-like, as none of the invariants used here depend on the free energy of the graphical structure (for other supporting evidence, see [5, 6, 10, 21]).

It should also be mentioned that the graphical invariants used in the present work represent just a subset of the available graphical invariants. Many more have been defined in the field of mathematical graph theory, as well as in chemical graph theory (see [23]). Further research may involve the calculation of different graphical invariants which may be used to quantify RNA graphs and perhaps may allow for more accurate prediction.

BIBLIOGRAPHY

[1] M. Aouchiche and P. Hansen, On a conjecture about the Randić Index, *Discrete Mathematics* **307** (2007) 262-265.

[2] L. Childs, Z. Nikoloski, P. May, and D. Walther, Identification and classification of ncRNA molecules using graph properties, *Nucleic Acids Research* **37**(9) (2009) 1-12.

[3] F. R. K. Chung, Eigenvalues and the Laplacian of a graph, In *Lectures on Spectral Graph Theory*, Philadelphia: Chung (1996) 1-22.

[4] C. Gershenson, Artificial Neural Networks for Beginners, [http://arxiv.org/abs/cs/0308031], (2003). Accessed March 2011.

[5] T. Haynes, D. Knisley, and J. Knisley, Using a Neural Network to Identify Secondary RNA Structures Quantified by Graphical Invariants, *MATCH Communications in Mathematical and in Computer Chemistry* **60** (2008) 277-290.

[6] T. Haynes, D. Knisley, E. Seier, and Y. Zoe, A Quantitative Analysis of Secondary RNA Structure Using Domination Based Parameters on Trees. *BMC Bioinformatics* **7** (2006) 108.

[7] Y. Karklin, R. F. Meraz, and S. R. Holbrook, Classification of non-coding RNA using graph representations of secondary structure, In *PSB Proceedings: 2004* (2004) 1-12.

[8] N. Kim, N. Shiffeldrim, H. H. Gan, and T. Schlick, Candidates for novel RNA topologies, *Journal of Molecular Biology* **341** (2004) 1129-1144.

[9] N. Kim, J. S. Shin, S. Elmetwaly, H. H. Gan, and T. Schlick, RAGPools: RNA-As-Graph Pools - A web server for assisting the design of structured RNA pools for in vitro selection, *Structural Bioinformatics*, Oxford University Press (2007) 1-2.

[10] D. R. Koessler, D. J. Knisley, J. Knisley, T. Haynes, A Predictive Model for Secondary RNA Structure Using Graph Theory and a Neural Network, *BMC Bioinformatics* **11** (Suppl 6):S21 (2010) 1-10.

[11] E. W. Weisstein, "Laplacian Matrix." From MathWorld–A Wolfram Web Resource. [`http://mathworld.wolfram.com/LaplacianMatrix.html`], (2011). Accessed April 2011.

[12] T. Lasko, J. Bhagwat, K. Zou and L. Ohno-Machado, The use of receiver operating characteristic curves in biomedical informatics, *Journal of Biomedical Informatics* **38** (2005) 404-415.

[13] S. Lawrence, C. Giles and A. Tsoi, Lessons in Neural Network Training: Overfitting May be Harder than Expected, *Proceedings of the Fourteenth National Conference on Artificial Intelligence* **AAAI-97**, 540-545 (1997).

[14] Maple 14. (2010). Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. Maple is a trademark of Waterloo Maple Inc.

[15] N. Markham, The Rensselaer bioinformatics web server, [http://mfold.bioinfo.rpi.edu] (2005). Accessed July 2010.

[16] Minitab Inc. (2003). MINITAB Statistical Software, Release 14 for Windows, State College, Pennsylvania. MINITAB is a registered trademark of Minitab Inc.

[17] S. Pasquali, H. H. Gan, and T. Schlick, Modular RNA architecture revealed by computational analysis of existing pseudoknots and ribosomal RNAs, *Nucleic Acids Research* **33** (2005) 1384-1398.

[18] T. Schlick, N. Kim, S. Elmetwaly, G. Quarta, J. Izzo, C. Laing, S. Jung, and A. Iqbal, RAG: RNA-As-Graphs Web Resource, [http://monod.biomath.nyu.edu/rna/rna.php], (2010). Accessed February 2011.

[19] T. Schlick, H. H. Gan, N. Kim, Y. Xin, G. Quarta, J. S. Shin, and C. Laing, RAG: RNA-As-Graphs Web Resource (Old version), [http://monod.biomath.nyu.edu/oldrag/rna.php], (2010). Accessed February 2011.

[20] J. Shao, Linear model selection by cross-validation, *J. Am. Statistical Association*, **88** (1993) 486-494.

[21] W. Shu, X. Bo, Z. Zheng, and S. Wang, A Novel Representation of RNA Secondary Structure Based on Element-Contact Graphs, *BMC Bioinformatics* **9**:188 (2008).

[22] L. Smith, An Introduction to Neural Networks, [http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html], (2003). Accessed March 2011.

[23] N. Trinajstić, *Chemical Graph Theory: Volume II.* Boca Raton: CRC Press, Inc. (1983) 166.

[24] R. Twyman, Rational drug design: Using structural information about drug targets or their natural ligands as a basis for the design of effective drugs. Copyright by The Wellcome Trust, [`http://genome.wellcome.ac.uk/doc_wtd020912.html`], (2002). Accessed April 2011.

[25] D. B. West, Introduction to Graph Theory, Second Edition, Prentice-Hall, Upper Saddle River, NJ (2001) 588.

[26] J. Zhao, R. L. Malmberg, and L. Cai, Rapid *ab initio* prediction of RNA pseudoknots via graph tree decomposition, In *Proceedings of 6th Workshop on Algorithms for Bioinformatics: 2006* (2006) 1-11.

[27] M. Zuker, Thermodynamics, software and databases for RNA structure, [`http://www.bioinfo.rpi.edu/zukerm/rna/node3.html`], (1998). Accessed February 2011.

APPENDICES

Appendix A: Maple 14 Code [14]

In this Appendix, the code used in Maple 14 ([14]) for the multilayer perceptron artificial neural network used in this research is given verbatim. The reader should note that comments in Maple are preceded by the symbol #.

The following serves as startup code for the Maple worksheet:

```
restart:with(GraphTheory);with(plots):with(Statistics):with(LinearAlgebra):with(StringTools):
    interface(displayprecision = 5);
interface(displayprecision=5);interface(rtablesize=22);
```

Below are compiler options, to make the neural network compatible with a Windows Vista operating system.

```
Neural Network Module
http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html

Compiler options.
forward := proc( inputs::Vector(datatype = float[ 8 ]), hiddens::Vector(datatype = float[ 8 ])
    , outputs::Vector(datatype = float[ 8 ]),
            whi::Matrix(datatype = float[ 8 ]), woh::Matrix(datatype = float[ 8 ]),
            alpha::float[8], beta::Vector(datatype=float[8]),
            lbH::Vector(datatype=float[8]), ubH::Vector(datatype=float[8]),
            lbO::Vector(datatype=float[8]), ubO::Vector(datatype=float[8]),
            ni::posint, nh::posint, no::posint, ActOnOutputs::posint )

local i::posint, j::posint, acc::float[8]:

for i from 1 to nh do
   acc:=beta[i]:
   for j from 1 to ni do
      acc := acc + whi[i,j]*inputs[j] :
   end do:
   hiddens[i] := lbH[i] + (ubH[i] - lbH[i])/(exp(-(ubH[i] - lbH[i])*acc)+1):
end do:
```

```
  for i from 1 to no do
     acc := 0.0:
     for j from 1 to nh do
        acc := acc + woh[i,j]*hiddens[j]:
     end do:
     if( ActOnOutputs = 2) then
        outputs[i] := lbO[i] + (ubO[i] - lbO[i])/(exp(-(ubO[i] - lbO[i])*acc)+1):
     else
        outputs[i] := acc:
     end if:
  end do:

  return true;

end proc:

backprop := proc( lrninput::Vector(datatype = float[8]),
                  lrnoutput::Vector(datatype = float[8]),
                  hiddens::Vector(datatype = float[ 8 ]),
                  outputs::Vector(datatype = float[8]),
                  whi::Matrix(datatype = float[ 8 ]),
                  woh::Matrix(datatype = float[ 8 ]),
                 pwhi::Matrix(datatype = float[ 8 ]),
                 pwoh::Matrix(datatype = float[ 8 ]),
                  delto::Vector(datatype=float[8]),
                  delth::Vector(datatype=float[8]),
                  alpha::float[8], BetaDecay::float[8],
                  beta::Vector(datatype = float[8]),
                  betachange::Vector(datatype = float[8]),
                  lbH::Vector(datatype=float[8]), ubH::Vector(datatype=float[8]),
                  lbO::Vector(datatype=float[8]), ubO::Vector(datatype=float[8]),
                  N::float[8], M::float[8],
                  ni::posint, nh::posint, no::posint, ActOnOutputs::posint, TrainThresholds::
                       posint )

   local i::posint,j::posint,k::posint, incError::float[8],acc::float[8], tmp::float[8], change::
        float[8];


   ##Calculate error and delto
   incError:=0.0:
   for i from 1 to no by 1 do
```

```
    incError := incError + 0.5*(outputs[i]-lrnoutput[i])*(outputs[i]-lrnoutput[i]):
    if(ActOnOutputs = 2) then
       delto[i]:=(outputs[i]-lbO[i])*(ubO[i] - outputs[i])*(lrnoutput[i]-outputs[i]):
    else
       delto[i]:=lrnoutput[i]-outputs[i]:
    end if:
end do:

##Calculate delth
for j from 1 to nh do
    acc := 0.0:
    for k from 1 to no do
       acc := acc + delto[k]*woh[k,j]:
    end do:
    delth[j] := (hiddens[j]-lbH[j])*(ubH[j]-hiddens[j])*acc:
end do:

## Update woh
for j from 1 to nh do
    for k from 1 to no do
       change  := delto[k] * hiddens[j]:
        woh[k,j] := woh[k,j] + N*change + M*pwoh[k,j]:
       pwoh[k,j] := change:
    end do:
end do:

##update whi
for i from 1 to ni do
    for j from 1 to nh do
       change  := delth[j]*lrninput[i]:
        whi[j,i] := whi[j,i] + N*change + M*pwhi[j,i]:
       pwhi[j,i] := change:
    end do:
end do:

##update beta
if(TrainThresholds=2) then
    for j from 1 to nh do
       beta[j] := BetaDecay*beta[j] + N*delth[j] + M*betachange[j]:
       betachange[j] := delth[j]:
    end do:
end if:
```

```
    return incError;

end proc:

#try
    cforward:=Compiler:-Compile(forward);
    cbackprop:=Compiler:-Compile(backprop);
#catch:
 #   cforward:=forward;
#    cbackprop:=backprop;
#    Maplets:-Examples:-Alert( "Compiling Failed. Using interpreted versions instead." );
#end try;
```

The following code creates the artificial neural network and defines commands for the ROC analysis.

```
Creates the neural network:
MakeANN := proc(ni::posint,no::posint)
    local  nh,tmp, ActOnOut, thebetadecay, ThThresh, themomentum, annealing, annealingscaler, bds,
         thelrnrate;

    nh := ni*no;
    try
       if(hasoption([args[3..-1]],'hiddens','tmp') or hasoption([args[3..-1]],'Hiddens','tmp') or
          hasoption([args[3..-1]],'hidden','tmp' ) or hasoption([args[3..-1]],'Hidden','tmp' )   )
              then
          if(type(tmp,'posint')) then
             nh := tmp;
          else
             error
          end if:
       end if:

       ActOnOut := 1:
       if(hasoption([args[3..-1]],'ActivationOnOutputs','tmp') ) then
         if(tmp = false or tmp = 0) then ActOnOut := 0 else error end if:
       end if:

       ThThresh := 1:
       if(hasoption([args[3..-1]],'ThresholdTraining','tmp') ) then
         if(tmp = true or tmp = 1) then ThThresh := 1 else error end if:
```

```
    end if:

    thelrnrate := 0.5:
    if(hasoption([args[3..-1]],'LearningRate','tmp') ) then
      if(0 < tmp and tmp <= 1) then thelrnrate := tmp else error end if:
    end if:

    thebetadecay := 1:
    if(hasoption([args[3..-1]],'ThresholdScaling','tmp') ) then
      if(0 < tmp and tmp <= 1) then thebetadecay := tmp else error end if:
    end if:

    themomentum := 0.0:
    if(hasoption([args[3..-1]],'Momentum','tmp') ) then
      if(0 <= tmp and tmp <= 1) then themomentum := tmp else error end if:
    end if:

    #Annealing is "internal" via the thresholds
    annealing := 0.0:
    if(hasoption([args[3..-1]],'Annealing','tmp') ) then
      if(0 <= tmp ) then annealing := tmp end if:
    end if:

    annealingscaler := 0.0:
    if(hasoption([args[3..-1]],'AnnealingScaler','tmp') ) then
      if(0 <= tmp ) then annealingscaler := tmp end if:
    end if:

    bds := [0.0, 1.0, 0.0, 1.0]:
    if(hasoption([args[3..-1]],'Bounds','tmp') or hasoption([args[3..-1]],'bounds','tmp') ) then
      if( type(tmp,'list(numeric)' ) ) then bds := tmp end if:
    end if:

catch:
    error cat("Invalid Option. Valid options are 'ActivationOnOutputs' = true/false, 'Annealing'
          = x>=0, 'AnnealingScaler' = x>= 0 ",
               "'hiddens' = positiveinteger, 'Momentum'= 0<=x<=1, 'Thresholds' = numeric, '
                    ThresholdScaling' = 0<x<=1 ",
               "or 'ThresholdTraining'=true/false." );
end try:

return module()
```

```
local learnrate, numints, numouts, numhids, inputs, hiddens, outputs, delto, delth, whi, woh
    , forward, backprop, lbH, ubH, lbO, ubO,
      outputsbetas, hiddensbetas, errs, cnt, f, TrainingSet,Patterns,Complements,
          WeightsUpdated, momentum, Annealing, betachange,
      AnnealingScaler, prevwhi, prevwoh, ActivateOutputs,pttns,preds, betadecay,
          ThresholdTraining, alpha;

export Initialize, Forward,BackProp,Classify, Get,
       Set, SetAlpha, SetBetas, SetLearningRate, SetThresholdScaling, SetThresholdTraining,
           SetThresholds, SetActivationOnOutputs,
       SetMomentum, SetAnnealing, SetAnnealingScaler, Train, AddPattern, RemovePattern,
           CreateComplement,
       ResetPatterns, PredictComplements,Errors,TrainingResults,ShowPatterns, SetBounds;

global cforward, cbackprop;

learnrate := convert(thelrnrate,'float[8]'):
momentum := convert(themomentum,'float[8]'):

numints:=ni: #This is the number of inputs in each pattern
numouts:=no: #This is the number of outputs
numhids:=nh:
WeightsUpdated := false:
errs := []:
cnt := 0:
ActivateOutputs := ActOnOut:
ThresholdTraining := ThThresh:
Annealing := convert(annealing,'float[8]'):
AnnealingScaler := convert(annealingscaler, 'float[8]'):

# Initialize arrays
inputs  := Vector[column](numints,datatype=float[8]):
hiddens := Vector[column](numhids,datatype=float[8]):
outputs := Vector[column](numouts,datatype=float[8]):

#the "delta's"
delto := Vector[column](numouts,datatype=float[8]):
delth := Vector[column](numhids,datatype=float[8]):

#weights in the form whi[hidden][input] and woh[outputs][hidden]
whi := Matrix(numhids,numints,datatype=float[8]): #Hidden layer weights
woh := Matrix(numouts,numhids,datatype=float[8]): #Output layer weights
```

```
prevwhi := Matrix(numhids,numints,datatype=float[8]): #Hidden layer momentum tmpvar
prevwoh := Matrix(numouts,numhids,datatype=float[8]): #Output layer momentum tmpvar

#Define thresholds
hiddensbetas := Vector[column](numhids,datatype=float[8],fill=0.0):
betachange := Vector[column](numhids,datatype=float[8],fill=0.0):

#Define Bounds
lbH := Vector[column](numhids,datatype=float[8],fill=min( bds[1], bds[2] ) ):
ubH := Vector[column](numhids,datatype=float[8],fill=max( bds[1], bds[2] ) ):
lbO := Vector[column](numhids,datatype=float[8],fill=min( bds[3], bds[4] ) ):
ubO := Vector[column](numhids,datatype=float[8],fill=max( bds[3], bds[4] ) ):

##Activation is of the form:
f:=entryv -> lbO[1] + (ubO[1] - lbO[1])/(exp(-(ubO[1] - lbO[1])*entryv)+1);
alpha := convert(1.0,'float[8]'):
betadecay := convert(thebetadecay,'float[8]'):

Get := proc( )
   local item, ref:

   if( nargs > 0 ) then ref := args[1]
   else
       return [ "learnrate" = learnrate, "momentum" = momentum, "numints"=numints, "numhids"
             =numhids, "numouts"  = numouts,
                "whi" =whi, "woh"=woh, "alpha"=alpha, "bounds" =[lbH[1], ubH[1], lbO[1], ubO
                     [1] ], "hiddenbetas" = hiddensbetas,
                "betachange" = betachange, "thresholdscaling" = betadecay, "
                     thresholdtraining" = ThresholdTraining,
                "annealing"  = Annealing, "trainingset" = TrainingSet, "patterns" =Patterns,
                "complements" = Complements ]:
   end if:

   item := StringTools:-LowerCase(convert(ref,'string')):
   if( item = "learnrate"   ) then return learnrate    end if:
   if( item = "momentum"    ) then return momentum     end if:
   if( item = "numints"     ) then return numints      end if:
   if( item = "numouts"     ) then return numouts      end if:
   if( item = "numhids"     ) then return numhids      end if:
   if( item = "whi"         ) then return whi          end if:
   if( item = "woh"         ) then return woh          end if:
   if( item = "alpha"       ) then return alpha        end if:
```

```
    if( item = "bounds"      ) then return [lbH[1], ubH[1], lbO[1], ubO[1] ] end if:
    if( item = "hiddenbetas" ) then return hiddensbetas end if:
    if( item = "betachange" )  then return betachange end if:
    if( item = "thresholdscaling")     then return betadecay         end if:
    if( item = "thresholdtraining")    then return ThresholdTraining end if:
    if( item = "activationonoutputs")  then return ActivateOutputs   end if:
    if( item = "annealing")            then return Annealing         end if:
    if( item = "annealingscaler")      then return Annealing         end if:
    if( item = "errs"        ) then return errs        end if:
    if( item = "cnt"         ) then return cnt         end if:
    if( item = "f"           ) then return f           end if:
    if( item = "fplot"       ) then return
       plot( f,-10/(ubH[1]-lbH[1])..10/(ubH[1]-lbH[1]),color=blue, title="Activation Function
           : Threshold=0", titlefont=[TIMES,BOLD,14])
                                                   end if:
    if( item = "trainingset" ) then return TrainingSet  end if:
    if( item = "patterns"    ) then return Patterns     end if:
    if( item = "complements" ) then return Complements  end if:
  end proc:

  Set := proc( ref  )
    local item, tmp:

    try
       if( type(ref,'equation') ) then
          item := StringTools:-LowerCase(convert(lhs(ref),'string')):
          tmp  := rhs(ref):
       else
          item := StringTools:-LowerCase(convert(ref,'string')):
          tmp  := args[2]:
       end:
    catch:
       error "Invalid input into Set."
    end try:
print(tmp, type(tmp,'list'), whattype(tmp));
    if( item = "learnrate"   )          then  return  SetLearningRate(tmp)        end if:
    if( item = "alpha"       )          then  return  SetAlpha(tmp)               end if:
    if( item = "bounds"      )          then  return  SetBounds(tmp)              end if:
    if( item = "momentum"    )          then  return  SetMomentum(tmp)            end if:
    if( item = "thresholdtraining" )    then  return  SetThreholdTraining(tmp)    end if:
    if( item = "thresholdscaling"  )    then  return  SetThresholdScaling(tmp)    end if:
    if( item = "thresholds"        )    then  return  SetThresholds(tmp)          end if:
```

```
   if( item = "activationonoutputs" ) then  return  SetActivationOnOutputs(tmp) end if:
   if( item = "annealing"          )  then  return  SetAnnealing(tmp)           end if:
   if( item = "annealingscaler"    )  then  return  SetAnnealingScaler(tmp)     end if:

   return FAIL

end proc:

SetAlpha:=proc(val::numeric)
   local tmp:
   tmp:=alpha:
   alpha := convert(val,'float[8]'):
   return tmp:
end proc:

SetBounds:=proc(val )
   local tmp, i:
   tmp:=[lbH[1], ubH[1], lbO[1], ubO[1] ]:
   if( nops(val) = 2) then
      for i from 1 to numhids do
         lbH[i], ubH[i] := min( val[1], val[2]), max( val[1], val[2] ):
      end do:
      for i from 1 to numouts do
         lbO[i], ubO[i] := min( val[1], val[2]), max( val[1], val[2] ):
      end do:
   elif( nops(val) = 4) then
      for i from 1 to numhids do
         lbH[i], ubH[i] := min( val[1], val[2]), max( val[1], val[2] ):
      end do:
      for i from 1 to numouts do
         lbO[i], ubO[i] := min( val[3], val[4]), max( val[3], val[4] ):
      end do:
   else
      error("List of bounds must have either 2 or 4 entries")
   end if:
   return tmp:
end proc:

SetThresholds:=proc(val)
   #Option of a number, a list that is distributed across hiddens, then betas until
       exhausted, or two lists in which the
   #first changes hiddens and the second outputs, but only the numeric entries do so
```

```
    local ind, i:

    if( type(val,'numeric') ) then
        for i from 1 to numhids do
            hiddensbetas[i] := val:
        end do:
        for i from 1 to numouts do
            outputsbetas[i] := val:
        end do:
        return true:
    elif( type(val,'list(list)') ) then
        for i from 1 to nops(val[1]) do
            if(i <= numhids and type(val[1][i],'numeric') ) then
                hiddensbetas[i] := val[1][i]:
            end if:
        end do:
        for i from 1 to nops(val[2]) do
            if(i <= numouts and type(val[2][i],'numeric') ) then
                outputsbetas[i] := val[2][i]:
            end if:
        end do:
    elif( type(val,'list(numeric)') ) then
        while(ind < nops(val) and ind <= numhids + numouts) do
            try
                if(ind <= numhids) then
                    hiddensbetas[ind] := val[ind]:
                else
                    outputsbetas[ind-numhids] := val[ind] :
                end if:
            catch:
                break:
            end try:
            ind := ind + 1:
        end do:
        return ind:
    else
        return false:
    end if:

end proc:

SetLearningRate:=proc(val::numeric)
```

```
   local tmp:
   tmp:=learnrate:
   if( val > 0 ) then
      learnrate := convert(val,'float[8]');
   end if:
   return learnrate:
end proc:

SetThresholdScaling:=proc(val::numeric)
   local tmp:
   tmp:=betadecay:
   if( val > 0.0 and val <= 1.0 ) then
      betadecay := convert(val,'float[8]');
   end if:
   return tmp:
end proc:

SetMomentum:=proc(val::numeric)
   local tmp:
   tmp:=momentum:
   if( val > 0.0 and val <= 1.0 ) then
      momentum := convert(val,'float[8]');
   end if:
   return tmp:
end proc:

SetAnnealing:=proc(val::numeric)
   local tmp:
   tmp:=Annealing:
   if( val >= 0.0 ) then
      Annealing := convert(val,'float[8]');
   end if:
   return tmp:
end proc:

SetAnnealingScaler :=proc(val::numeric)
   local tmp:
   tmp:=AnnealingScaler:
   if( val >= 0.0 ) then
      AnnealingScaler := convert(val,'float[8]');
   end if:
   return tmp:
```

```
end proc:

SetThresholdTraining:=proc(val::boolean)
   local tmp:
   tmp := ThresholdTraining:
   if(val) then ThresholdTraining := 1 else ThresholdTraining := 0 end if:
   return tmp;
end proc:

SetActivationOnOutputs:=proc(val::boolean)
   local tmp:
   tmp := ActivateOutputs:
   if(val) then ActivateOutputs := 1 else ActivateOutputs := 0 end if:
   return tmp;
end proc:

Initialize:=proc(maxamt) #initalize weights to small random values with mag
   local i, h, o:
   for h from 1 to numhids do
      hiddensbetas[h]:=(rand()*10^(-11)-5)*maxamt:
      for i from 1 to numints do
            whi[h,i]:=(rand()*10^(-11)-5)*maxamt:
         prevwhi[h,i] := whi[h,i]:
      end do:
   end do:
   for o from 1 to numouts do
      outputsbetas[o]:=(rand()*10^(-11)-5)*maxamt:
      for h from 1 to numhids do
            woh[o,h]:=(rand()*10^(-11)-5)*maxamt:
         prevwoh[o,h]:=woh[o,h]:
      end do:
   end do:
   cnt := 0:
   WeightsUpdated := true:
   errs:=[]:
end proc:

Forward := proc(inpt::Vector)
   local inpfl8;

   inpfl8 := Vector(inpt,datatype=float[8]):
   cforward(inpfl8, hiddens, outputs, whi, woh, alpha, hiddensbetas, lbH, ubH, lbO, ubO,
```

```
            numints, numhids, numouts, ActivateOutputs+1):
   return outputs;

end proc:

#Define the forward function

Classify:=proc(inpat::Vector)
   LinearAlgebra[Transpose](Forward(inpat)):
end proc:

BackProp:=proc(lrninput::Vector,lrnoutput::Vector)
   local tmpLinpt, tmpLoutp, annl, annlseq, threshvalues,i;

   Forward(lrninput):
   tmpLinpt := Vector(lrninput,datatype=float[8]);
   tmpLoutp := Vector(lrnoutput, datatype=float[8]);
   WeightsUpdated := true:

   if( Annealing >= 0.01 ) then
      if( Annealing <= 1 ) then
         annl := floor( Annealing*(numhids + numouts) ):
      else
         annl := min(numhids+numouts, floor(Annealing) ):
      end if:

      annlseq := combinat[randperm]([seq(i,i=1..numhids+numouts)] )[1..annl];

      for i in annlseq do
         if( i <= numhids) then
            hiddensbetas[i] := AnnealingScaler*hiddensbetas[i]:
         else
            outputsbetas[i-numhids+1] := AnnealingScaler*outputsbetas[i-numhids+1]:
         end if:
      end do:

   end if:

   cbackprop(tmpLinpt,tmpLoutp, hiddens, outputs,whi,woh,prevwhi,prevwoh,delto,delth,alpha,
      betadecay,
         hiddensbetas, betachange, lbH, ubH, lbO, ubO, learnrate,momentum,
         numints, numhids, numouts, ActivateOutputs, ThresholdTraining+1):
```

```
  end proc:

##Training Set Algorithms

  Patterns:=[]:

  AddPattern := proc(inp,oup)
    local inpt, oupt, attr;

    attr := "":
    hasoption([args[3..-1]],'attributes','attr'):

    inpt := Vector(convert(inp,'list'),orientation=column): #print(inpt);
    oupt := Vector(convert(oup,'list'),orientation=column): #print(oupt);

    if(LinearAlgebra:-Dimensions(inpt) = numints and LinearAlgebra:-Dimensions(oupt) =
         numouts ) then
       Patterns := [ Patterns[], [inpt,oupt,attr] ]:
       TrainingSet := [seq(i,i=1..nops(Patterns))]:
       true:
    else
       false:
    end if:
  end proc:

  RemovePattern  := proc(PatIndex::posint)
    if(PatIndex <= nops(Patterns) ) then
       Patterns := [ Patterns[1..PatIndex-1],Patterns[PatIndex+1..-1]]:
       true:
    else
       false:
    end if:

  end proc:

  ShowPatterns := proc()
    Patterns;
  end proc:

  ResetPatterns := proc()
    Patterns:=[ ]:
```

```
    TrainingSet := []:
    Complements := []:
end proc:

CreateComplement := proc( amt ) #amt is the proportion or number to use to put into the
      complement
    local tmp:

    if(0 <= amt and amt < 1 ) then
        tmp := floor(amt * nops(Patterns) ):
    elif( 1 <= amt and amt < nops(Patterns) ) then
        tmp := amt:
    end if:

    TrainingSet := {seq(i,i=1..nops(Patterns))}:
    Complements := {seq(combinat[randperm](TrainingSet)[j],j=1..tmp)}:
    TrainingSet := convert(TrainingSet minus Complements,'list');
    Complements := convert(Complements, 'list'):

    return 'TrainBy',TrainingSet,'ValidateWith',Complements;
end proc:

Train := proc(NumberOfReps::posint, amt := 0 )
    local InitValue, cls, terr, Indices, TSI, VSI, i, n, esum, tsum, v, tmp;

    if(0 <= amt and amt < 1 ) then
        tmp := floor(amt * nops(TrainingSet) ):
    elif( 1 <= amt and amt < nops(TrainingSet) ) then
        tmp := amt:
    end if:
    ## tmp is number of patterns to partition into "test against" set
    if( tmp >= nops(TrainingSet) ) then error( "Test set of size ", tmp," is larger than
          training set." ) end if:

    terr := []:
    TSI := combinat[randperm](TrainingSet):
    VSI := TSI[1..tmp]:
    TSI := TSI[tmp+1..-1]:

    for n from 1 to NumberOfReps do
        Indices := combinat[randperm](TSI):
        esum:=0:
```

```
        pttns := []:
        preds := []:

        for i in Indices do
           v:=Patterns[i]:
           esum := esum + BackProp(v[1],v[2]):
           pttns := [ pttns[], v[2][1] ];
           preds := [ preds[], outputs[1] ]:
        end do:

        tsum := 0:
        for i in VSI do
           v:=Patterns[i]:
           cls:=Forward(v[1]):
           tsum := tsum + LinearAlgebra:-VectorNorm(cls-v[2]):
        end do:
        cnt := cnt+1:

        errs := [errs[], esum]:
        terr := [terr[],tsum]:
    end do:

    return errs, terr;
end proc:

Errors := () -> errs;

TrainingResults := proc(binary:=true)
     local v,i,err,cls;

     err := 0:
     pttns:=[]:
     preds:=[]:
     for i in TrainingSet do
        v:=Patterns[i]:
        cls:=Forward(v[1]):
        pttns := [ pttns[], v[2][1] ];
        preds := [ preds[], outputs[1] ]:
        err := err + LinearAlgebra:-VectorNorm(cls-v[2]):
     end do:

     if( binary) then
```

```
            return err/nops(Patterns), ROCanalysis(pttns,preds )
         else
            return err/nops(Patterns), pttns, preds ;
         end if:
      end proc:

      PredictComplements := proc(binary:=true)
         local v,i,err,cls;

         err := 0:
         pttns:=[]:
         preds:=[]:
         for i in Complements do
            v:=Patterns[i]:
            cls:=Forward(v[1]):
            pttns := [ pttns[], v[2][1] ];
            preds := [ preds[], outputs[1] ]:
            err := err + LinearAlgebra:-VectorNorm(cls-v[2]):
         end do:

         if( binary) then
            return err/nops(Patterns), ROCanalysis(pttns,preds )
         else
            return err/nops(Patterns), pttns, preds ;
         end if:
      end proc:

   end module:

end proc:

SaveNet := proc(nnme) # ANNfN is optional 2nd argument
   local ANNfN,Args,NetName;

   ANNfN := NULL:
   NetName := convert(nnme,'string'):
   if( nargs>1 and type(args[2],'string' ) ) then
      ANNfN := args[2]:
      Args:=args[3..-1]:
   else
      ANNfN := Maplets[Utilities][GetFile]('title' = "Save as...", filename = cat(NetName,".mnn"),
            'directory' = currentdir(), approvecaption="Save", approvecheck=false,
```

```
            'filefilter' = "mnn", 'filterdescription' = "Maple Neural Network"):
      Args:=args[2..-1]:
   end if:

   use StringTools in
      if( LowerCase(SubString(ANNfN,-4..-1))<> ".mnn" and SubString(ANNfN,-1..-1) <> " " ) then
         ANNfN := cat(ANNfN,".mnn"):
      end if:
   end use:

   save parse(NetName), ANNfN;

end proc:

LoadNet := proc()
   local ANNfN ;

   ANNfN := NULL:
   if( nargs>0 and type(args[1],'string') ) then
      ANNfN := args[1]:
   else
      ANNfN := Maplets[Utilities][GetFile]('title' = "Open Neural Network",
            'directory' = currentdir(), approvecaption="Open",
            'filefilter' = "mnn", 'filterdescription' = "Maple Neural Network"):
   end if:

   read ANNfN;

end proc:
ROCanalysis := proc(Actual,Predicted)
   local k, PosValues, NegValues, PosNum, NegNum, thresh, threshs, tmin, tmax, npts,
         PosInd, NegInd, BestThresh, opt, plt, tmp, dist, _ROC_, _AUC_;

   if(nops(Predicted) <> nops(Actual) ) then
      error "Actual and Predicted are lists of corresponding values"
   end if:

   PosValues := []:
   NegValues := []:
   npts := 10: #Later we'll make this an option
   for k from 1 to nops(Actual) do
      if( Actual[k] = 1 ) then
```

```
      PosValues := [ PosValues[], Predicted[k] ]:
    else
      NegValues := [ NegValues[], Predicted[k] ]:
    end if:
end do:

PosValues := sort(PosValues);
PosNum := nops(PosValues):
PosValues := queue[new]( PosValues[] ):
NegValues := sort(NegValues):
NegNum := nops(NegValues):
NegValues := queue[new]( NegValues[] ):

##Eliminate duplicate thresholds by converting to a set
tmax := max(Predicted);
tmin := min(Predicted):
threshs := convert(Predicted,'set'):
threshs := sort(convert(Predicted,'list')):

_ROC_:=[[1,1]]:
BestThresh := [2,0,[1,1]]:
for thresh in threshs do     ##from tmin to tmax by evalf((tmax-tmin)/npts) do
    while( ( not queue[empty](PosValues) ) and queue[front](PosValues) <= thresh ) do
      queue[dequeue](PosValues):
    end do:
    while( ( not queue[empty](NegValues) ) and queue[front](NegValues) <= thresh ) do
      queue[dequeue](NegValues):
    end do:

    tmp  := [evalf(queue[length](NegValues)/NegNum),evalf(queue[length](PosValues)/PosNum) ]:
        ### debugging
        #print(thresh,queue[length](NegValues),queue[length](PosValues),tmp);
    dist := evalf(sqrt(tmp[1]^2 + (tmp[2]-1)^2)):
    if( dist < BestThresh[1]) then
      BestThresh := [dist,thresh,tmp]:
    end if:

    _ROC_ := [ _ROC_[], tmp ]:

 end do:

 _AUC_ := add( abs((_ROC_[k][1] - _ROC_[k+1][1]))*(_ROC_[k][2]+_ROC_[k+1][2]), k=1..(nops(_ROC_
```

```
        )-1))/2;

    plt := plots[display](
                  plots[listplot]( _ROC_, color=blue ),
                  plots[listplot]( _ROC_, style=point, symbol=solidcircle, color=blue),
                  plots[listplot]( [[0,0],[1,1]], style=line, color=grey),
                  plots[pointplot]( BestThresh[3], style=point, symbol=solidcircle, color=green,
                       symbolsize=20),
                  plots[textplot]( [BestThresh[3][1],BestThresh[3][2],
                     cat("(",sprintf("%0.2f",BestThresh[3][1]),",",sprintf("%0.2f",BestThresh
                          [3][2]),")")],
                     font=[TIMES,ROMAN,14],align=[BELOW,RIGHT]),
               axes=boxed,labels=["Specificity","Sensitivity"],labeldirections=[horizontal,
                    vertical],
               font = [TIMES,ROMAN,14],view=[0..1,0..1], title = cat("ROC Curve: \n Threshold =
                    ",
                         sprintf("%0.2f",BestThresh[2]), ",     AUC = ", sprintf("%0.2f",_AUC_))
         );

opt:="none":
hasoption([args[3..-1]], 'output', 'opt'):
opt:=StringTools:-UpperCase(convert(opt,'string')):
if( opt = "SOLUTIONMODULE" ) then
   return
      module()
         local auc, best, roc, clst ;
         export AUC, ROCplot, BestThreshold, ROC ;
         best := BestThresh[2]:
         clst := BestThresh[3]:
         auc := _AUC_;
         roc := _ROC_;

         AUC := () -> auc;
         BestThreshold := () -> best;
         ROC := () -> roc ;
         ROCplot := () -> plt ;
     end module;
elif( opt = "AUC") then
   return _AUC_
elif( opt = "ROC") then
   return _ROC_
elif( opt = "OOP" or opt = "BESTTHRESHOLD" or opt = "BEST" ) then
```

```
      return BestThresh[2]
   elif( opt = "LOCATION") then
      return BestThresh[3]
   else
      return plt
   end if:
end proc:
rnd:=rand(1..100):
ROCanalysis([seq(0,i=1..200), seq(1,i=1..200)],[seq(rnd()*i,i=1..400)] );
ROCanalysis([seq(0,i=1..200), seq(1,i=1..200)],[seq(rnd(),i=1..400)] , output="solutionmodule");
rr:=%;
rr:-AUC():
rr:-ROCplot();
```

The following code executes the neural network, defines the original TS, executes thirty training sessions, and performs an ROC analysis.

```
RNANet: First Training Set (described above)
#Red in TS: 4.01, 4.02, 4.05, 4.08, 4.09, 4.15, 4.19, 4.20, 4.22, and 4.27
RNANet := MakeANN(12,2, hiddens=16, Momentum = 0.1);
TrainingSet
     :={[<2.70149,0.03195,2.77377,-2.50280,-2.95843,-2.61216,-3.39110,2.82848,-1.67967,-1.92803,2.51898,

-0.56106>, <1,0>],
     [<1.88125,0.03195,1.57931,-1.70407,-1.83483,-1.26081,-0.21145,1.61636,-1.67967,-1.49315,1.43993,0.32389>,
      <1,0>],
     [<0.78996,-0.93611,-0.01338,-0.50599,0.41236,0.09054,-0.21145,-0.20182,-0.05366,-0.62340,-0.43680,1.20885>,
      <1,0>],
     [<0.75765,0.03195,0.38485,-0.50599,-0.71124,0.09054,-0.21145,0.10121,-0.05366,-0.62340,0.64225,0.32389>,
      <1,0>],
     [<1.55979,0.03195,0.38485,-0.90535,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,1.63153,-0.56106>,
      <1,0>],
     [<-0.35361,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.14292,-0.56106>,
      <1,0>],
     [<0.13570,0.03195,0.38485,-0.90535,-0.71124,-1.26081,2.96820,1.31333,-1.67967,-1.05827,1.03037,0.32389>,
      <1,0>],
     [<-0.30520,0.03195,0.38485,-0.10663,-0.71124,-1.26081,-0.21145,0.10121,-0.05366,-0.62340,0.36088,1.20885>,
      <1,0>],
     [<-0.76353,1.00000,0.38485,-0.10663,-0.71124,-1.26081,1.37838,0.40424,-1.67967,-0.18852,0.36088,0.32389>,
      <1,0>],
     [<-1.04899,1.00000,-0.80960,1.09145,1.53596,1.44189,-0.21145,-1.11091,1.57236,1.98587,-1.31934,-0.56106>,
      <1,0>],
     [<1.59418,-0.93611,1.18108,-1.30471,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,0.64225,0.32389>,
      <0,1>],
```

```
    [<0.99270,-0.93611,1.18108,-0.50599,-0.71124,0.09054,-0.21145,0.60624,-0.05366,-0.62340,0.91380,-0.56106>,
     <0,1>],
    [<0.50827,-1.90416,-0.41137,-0.10663,1.53596,0.09054,-0.21145,-0.80788,-0.05366,-0.18852,-1.20768,1.20885>,
     <0,1>],
    [<-0.36349,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,0.30321,1.57236,0.24636,-0.07548,-1.44602>,
     <0,1>],
    [<-0.57366,0.03195,0.38485,0.69209,0.41236,1.44189,-0.21145,-0.20182,1.57236,0.68124,-0.14694,-0.56106>,
     <0,1>],
    [<-0.63643,-0.93611,0.38485,1.09145,1.53596,1.44189,-0.21145,-0.90891,1.57236,1.11611,-1.31934,-0.56106>,
     <0,1>],
    [<-0.47360,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,-0.20182,-0.05366,0.24636,-0.53015,1.20885>,
     <0,1>],
    [<-0.88702,0.03195,0.38485,1.09145,0.41236,0.09054,-0.21145,-0.20182,1.57236,1.11611,-0.60161,-0.56106>,
     <0,1>],
    [<-0.70762,-0.93611,-0.80960,0.69209,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.68124,-1.15453,2.09381>,
     <0,1>],
    [<-0.56804,0.03195,-0.80960,0.29273,-0.71124,0.09054,-0.21145,-0.80788,-0.05366,0.24636,-0.43680,1.20885>,
     <0,1>]};
nops(TrainingSet);
for TS in TrainingSet do
  RNANet:-AddPattern(TS[1],TS[2])
end do:
RNANet:-ShowPatterns();
RNANet:-Get(fplot);
RNANet:-Initialize(0.1);
RNANet:-Train(30);
plots[listplot](%);


RNANet:-TrainingResults();%[2];
```

The following code tells the neural network to classify graph invariant vectors for the dual graphs not in the TS.

```
RNANet:-Classify
    (<1.07417,0.03195,0.38485,-0.90535,-0.71124,-1.26081,-0.21145,0.40424,-1.67967,-1.05827,0.36088,0.32389>)
    ; %[1]+%[2]; #Red4.04
RNANet:-Classify
    (<-0.04566,0.03195,0.38485,-1.30471,-0.71124,-1.26081,1.37838,1.31333,-0.05366,-1.05827,1.43993,-1.44602>)
    ; %[1]+%[2]; #Blue4.10
RNANet:-Classify
    (<-0.31290,0.03195,0.38485,-0.10663,0.41236,0.09054,1.37838,0.40424,1.57236,-0.18852,0.59714,-0.56106>)
    ; %[1]+%[2]; #Blue4.12
RNANet:-Classify
    (<-1.24837,3.90416,-2.00406,1.89018,0.41236,0.09054,1.37838,-1.11091,-0.05366,1.98587,-1.18133,-1.44602>)
```

```
; %[1]+%[2]; #Blue4.30
RNANet:-Classify
    (<0.14688,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.24636,-0.60161,0.32389>)
    ; %[1]+%[2]; #Red4.14
RNANet:-Classify
    (<-0.37170,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.18758,-0.56106>)
    ; %[1]+%[2]; #Red4.16
RNANet:-Classify
    (<-0.58348,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-0.50485,-0.05366,0.24636,-0.43680,-0.56106>)
    ; %[1]+%[2]; #Red4.17
RNANet:-Classify
    (<-0.96015,1.00000,-0.80960,1.09145,0.41236,1.44189,-0.21145,-0.50485,-0.05366,0.68124,-0.43680,-1.44602>)
    ; %[1]+%[2]; #Red4.25
RNANet:-Classify
    (<-1.06683,1.00000,-0.80960,1.49081,1.53596,0.09054,-0.21145,-1.11091,-0.05366,1.55099,-1.20768,-0.56106>)
    ;%[1]+%[2]; #Red4.28
RNANet:-Classify
    (<-0.86698,0.03195,-2.00406,1.09145,0.41236,0.09054,-0.21145,-1.71697,-0.05366,1.11611,-1.18133,2.09381>)
    ; %[1]+%[2]; #Red4.29
```

The code below performs the method of leave-v-out cross-validation. The code given here is when the first vector, corresponding to dual graph 4.01, is removed from the TS. First, the network is trained thirty times, and the non-TS dual graphs are classified; then, the network is trained thirty-two times, and the non-TS dual graphs are classified; finally, the network is trained thirty-five times, and the non-TS dual graphs are classified. The code below may be repeated for other adjusted training sets in which vectors for dual graphs other than 4.01 are removed from the TS.

```
Training Session 1: Leave-v-out Cross Validation
#In this training session, the first vector is left out of the original TS. Thus, we label this
    new TS as TS1.
RNANet1 := MakeANN(12,2, hiddens=16, Momentum = 0.1);
TrainingSet1:={
    [<1.88125,0.03195,1.57931,-1.70407,-1.83483,-1.26081,-0.21145,1.61636,-1.67967,-1.49315,1.43993,0.32389>,
     <1,0>],
    [<0.78996,-0.93611,-0.01338,-0.50599,0.41236,0.09054,-0.21145,-0.20182,-0.05366,-0.62340,-0.43680,1.20885>,
     <1,0>],
    [<0.75765,0.03195,0.38485,-0.50599,-0.71124,0.09054,-0.21145,0.10121,-0.05366,-0.62340,0.64225,0.32389>,
     <1,0>],
    [<1.55979,0.03195,0.38485,-0.90535,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,1.63153,-0.56106>,
     <1,0>],
    [<-0.35361,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.14292,-0.56106>,
     <1,0>],
    [<0.13570,0.03195,0.38485,-0.90535,-0.71124,-1.26081,2.96820,1.31333,-1.67967,-1.05827,1.03037,0.32389>,
```

```
    <1,0>],
   [<-0.30520,0.03195,0.38485,-0.10663,-0.71124,-1.26081,-0.21145,0.10121,-0.05366,-0.62340,0.36088,1.20885>,
    <1,0>],
   [<-0.76353,1.00000,0.38485,-0.10663,-0.71124,-1.26081,1.37838,0.40424,-1.67967,-0.18852,0.36088,0.32389>,
    <1,0>],
   [<-1.04899,1.00000,-0.80960,1.09145,1.53596,1.44189,-0.21145,-1.11091,1.57236,1.98587,-1.31934,-0.56106>,
    <1,0>],
   [<1.59418,-0.93611,1.18108,-1.30471,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,0.64225,0.32389>,
    <0,1>],
   [<0.99270,-0.93611,1.18108,-0.50599,-0.71124,0.09054,-0.21145,0.60624,-0.05366,-0.62340,0.91380,-0.56106>,
    <0,1>],
   [<0.50827,-1.90416,-0.41137,-0.10663,1.53596,0.09054,-0.21145,-0.80788,-0.05366,-0.18852,-1.20768,1.20885>,
    <0,1>],
   [<-0.36349,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,0.30321,1.57236,0.24636,-0.07548,-1.44602>,
    <0,1>],
   [<-0.57366,0.03195,0.38485,0.69209,0.41236,1.44189,-0.21145,-0.20182,1.57236,0.68124,-0.14694,-0.56106>,
    <0,1>],
   [<-0.63643,-0.93611,0.38485,1.09145,1.53596,1.44189,-0.21145,-0.90891,1.57236,1.11611,-1.31934,-0.56106>,
    <0,1>],
   [<-0.47360,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,-0.20182,-0.05366,0.24636,-0.53015,1.20885>,
    <0,1>],
   [<-0.88702,0.03195,0.38485,1.09145,0.41236,0.09054,-0.21145,-0.20182,1.57236,1.11611,-0.60161,-0.56106>,
    <0,1>],
   [<-0.70762,-0.93611,-0.80960,0.69209,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.68124,-1.15453,2.09381>,
    <0,1>],
   [<-0.56804,0.03195,-0.80960,0.29273,-0.71124,0.09054,-0.21145,-0.80788,-0.05366,0.24636,-0.43680,1.20885>,
    <0,1>]};
nops(TrainingSet1);
for TS in TrainingSet1 do
  RNANet1:-AddPattern(TS[1],TS[2])
end do:
RNANet1:-ShowPatterns();
RNANet1:-Get(fplot);
RNANet1:-Initialize(0.1);
RNANet1:-Train(30);
plots[listplot](%);
RNANet1:-TrainingResults();%[2];
RNANet1:-Classify
   (<2.70149,0.03195,2.77377,-2.50280,-2.95843,-2.61216,-3.39110,2.82848,-1.67967,-1.92803,2.51898,-0.56106>)
   ; %[1]+%[2]; #Red4.01 (left out of this TS)

RNANet1:-Classify
```

```
    (<1.07417,0.03195,0.38485,-0.90535,-0.71124,-1.26081,-0.21145,0.40424,-1.67967,-1.05827,0.36088,0.32389>)
    ; %[1]+%[2]; #Red4.04
RNANet1:-Classify
    (<-0.04566,0.03195,0.38485,-1.30471,-0.71124,-1.26081,1.37838,1.31333,-0.05366,-1.05827,1.43993,-1.44602>)
    ; %[1]+%[2]; #Blue4.10
RNANet1:-Classify
    (<-0.31290,0.03195,0.38485,-0.10663,0.41236,0.09054,1.37838,0.40424,1.57236,-0.18852,0.59714,-0.56106>)
    ; %[1]+%[2]; #Blue4.12
RNANet1:-Classify
    (<-1.24837,3.90416,-2.00406,1.89018,0.41236,0.09054,1.37838,-1.11091,-0.05366,1.98587,-1.18133,-1.44602>)
    ; %[1]+%[2]; #Blue4.30
RNANet1:-Classify
    (<0.14688,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.24636,-0.60161,0.32389>)
    ; %[1]+%[2]; #Red4.14
RNANet1:-Classify
    (<-0.37170,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.18758,-0.56106>)
    ; %[1]+%[2]; #Red4.16
RNANet1:-Classify
    (<-0.58348,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-0.50485,-0.05366,0.24636,-0.43680,-0.56106>)
    ; %[1]+%[2]; #Red4.17
RNANet1:-Classify
    (<-0.96015,1.00000,-0.80960,1.09145,0.41236,1.44189,-0.21145,-0.50485,-0.05366,0.68124,-0.43680,-1.44602>)
    ; %[1]+%[2]; #Red4.25
RNANet1:-Classify
    (<-1.06683,1.00000,-0.80960,1.49081,1.53596,0.09054,-0.21145,-1.11091,-0.05366,1.55099,-1.20768,-0.56106>)
    ;%[1]+%[2]; #Red4.28
RNANet1:-Classify
    (<-0.86698,0.03195,-2.00406,1.09145,0.41236,0.09054,-0.21145,-1.71697,-0.05366,1.11611,-1.18133,2.09381>)
    ; %[1]+%[2]; #Red4.29
#Now we will use a second training length on TrainingSet1:
RNANet2 := MakeANN(12,2, hiddens=16, Momentum = 0.1);
 TrainingSet1:={
    [<1.88125,0.03195,1.57931,-1.70407,-1.83483,-1.26081,-0.21145,1.61636,-1.67967,-1.49315,1.43993,0.32389>,
     <1,0>],
    [<0.78996,-0.93611,-0.01338,-0.50599,0.41236,0.09054,-0.21145,-0.20182,-0.05366,-0.62340,-0.43680,1.20885>,
     <1,0>],
    [<0.75765,0.03195,0.38485,-0.50599,-0.71124,0.09054,-0.21145,0.10121,-0.05366,-0.62340,0.64225,0.32389>,
     <1,0>],
    [<1.55979,0.03195,0.38485,-0.90535,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,1.63153,-0.56106>,
     <1,0>],
    [<-0.35361,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.14292,-0.56106>,
     <1,0>],
```

```
     [<0.13570,0.03195,0.38485,-0.90535,-0.71124,-1.26081,2.96820,1.31333,-1.67967,-1.05827,1.03037,0.32389>,
      <1,0>],
     [<-0.30520,0.03195,0.38485,-0.10663,-0.71124,-1.26081,-0.21145,0.10121,-0.05366,-0.62340,0.36088,1.20885>,
      <1,0>],
     [<-0.76353,1.00000,0.38485,-0.10663,-0.71124,-1.26081,1.37838,0.40424,-1.67967,-0.18852,0.36088,0.32389>,
      <1,0>],
     [<-1.04899,1.00000,-0.80960,1.09145,1.53596,1.44189,-0.21145,-1.11091,1.57236,1.98587,-1.31934,-0.56106>,
      <1,0>],
     [<1.59418,-0.93611,1.18108,-1.30471,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,0.64225,0.32389>,
      <0,1>],
     [<0.99270,-0.93611,1.18108,-0.50599,-0.71124,0.09054,-0.21145,0.60624,-0.05366,-0.62340,0.91380,-0.56106>,
      <0,1>],
     [<0.50827,-1.90416,-0.41137,-0.10663,1.53596,0.09054,-0.21145,-0.80788,-0.05366,-0.18852,-1.20768,1.20885>,
      <0,1>],
     [<-0.36349,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,0.30321,1.57236,0.24636,-0.07548,-1.44602>,
      <0,1>],
     [<-0.57366,0.03195,0.38485,0.69209,0.41236,1.44189,-0.21145,-0.20182,1.57236,0.68124,-0.14694,-0.56106>,
      <0,1>],
     [<-0.63643,-0.93611,0.38485,1.09145,1.53596,1.44189,-0.21145,-0.90891,1.57236,1.11611,-1.31934,-0.56106>,
      <0,1>],
     [<-0.47360,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,-0.20182,-0.05366,0.24636,-0.53015,1.20885>,
      <0,1>],
     [<-0.88702,0.03195,0.38485,1.09145,0.41236,0.09054,-0.21145,-0.20182,1.57236,1.11611,-0.60161,-0.56106>,
      <0,1>],
     [<-0.70762,-0.93611,-0.80960,0.69209,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.68124,-1.15453,2.09381>,
      <0,1>],
     [<-0.56804,0.03195,-0.80960,0.29273,-0.71124,0.09054,-0.21145,-0.80788,-0.05366,0.24636,-0.43680,1.20885>,
      <0,1>]};
nops(TrainingSet1);
for TS in TrainingSet1 do
   RNANet2:-AddPattern(TS[1],TS[2])
end do:
RNANet2:-ShowPatterns();
RNANet2:-Get(fplot);
RNANet2:-Initialize(0.1);
RNANet2:-Train(32);
plots[listplot](%);
RNANet2:-TrainingResults();%[2];
RNANet2:-Classify
     (<2.70149,0.03195,2.77377,-2.50280,-2.95843,-2.61216,-3.39110,2.82848,-1.67967,-1.92803,2.51898,-0.56106>)
     ; %[1]+%[2]; #Red4.01 (left out of this TS)
```

```
RNANet2:-Classify
    (<1.07417,0.03195,0.38485,-0.90535,-0.71124,-1.26081,-0.21145,0.40424,-1.67967,-1.05827,0.36088,0.32389>)
    ; %[1]+%[2]; #Red4.04
RNANet2:-Classify
    (<-0.04566,0.03195,0.38485,-1.30471,-0.71124,-1.26081,1.37838,1.31333,-0.05366,-1.05827,1.43993,-1.44602>)
    ; %[1]+%[2]; #Blue4.10
RNANet2:-Classify
    (<-0.31290,0.03195,0.38485,-0.10663,0.41236,0.09054,1.37838,0.40424,1.57236,-0.18852,0.59714,-0.56106>)
    ; %[1]+%[2]; #Blue4.12
RNANet2:-Classify
    (<-1.24837,3.90416,-2.00406,1.89018,0.41236,0.09054,1.37838,-1.11091,-0.05366,1.98587,-1.18133,-1.44602>)
    ; %[1]+%[2]; #Blue4.30
RNANet2:-Classify
    (<0.14688,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.24636,-0.60161,0.32389>)
    ; %[1]+%[2]; #Red4.14
RNANet2:-Classify
    (<-0.37170,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.18758,-0.56106>)
    ; %[1]+%[2]; #Red4.16
RNANet2:-Classify
    (<-0.58348,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-0.50485,-0.05366,0.24636,-0.43680,-0.56106>)
    ; %[1]+%[2]; #Red4.17
RNANet2:-Classify
    (<-0.96015,1.00000,-0.80960,1.09145,0.41236,1.44189,-0.21145,-0.50485,-0.05366,0.68124,-0.43680,-1.44602>)
    ; %[1]+%[2]; #Red4.25
RNANet2:-Classify
    (<-1.06683,1.00000,-0.80960,1.49081,1.53596,0.09054,-0.21145,-1.11091,-0.05366,1.55099,-1.20768,-0.56106>)
    ;%[1]+%[2]; #Red4.28
RNANet2:-Classify
    (<-0.86698,0.03195,-2.00406,1.09145,0.41236,0.09054,-0.21145,-1.71697,-0.05366,1.11611,-1.18133,2.09381>)
    ; %[1]+%[2]; #Red4.29
#We will now use a third training length.
RNANet3 := MakeANN(12,2, hiddens=16, Momentum = 0.1);
TrainingSet1:={
    [<1.88125,0.03195,1.57931,-1.70407,-1.83483,-1.26081,-0.21145,1.61636,-1.67967,-1.49315,1.43993,0.32389>,
     <1,0>],
    [<0.78996,-0.93611,-0.01338,-0.50599,0.41236,0.09054,-0.21145,-0.20182,-0.05366,-0.62340,-0.43680,1.20885>,
     <1,0>],
    [<0.75765,0.03195,0.38485,-0.50599,-0.71124,0.09054,-0.21145,0.10121,-0.05366,-0.62340,0.64225,0.32389>,
     <1,0>],
    [<1.55979,0.03195,0.38485,-0.90535,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,1.63153,-0.56106>,
     <1,0>],
    [<-0.35361,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.14292,-0.56106>,
```

```
      <1,0>],
     [<0.13570,0.03195,0.38485,-0.90535,-0.71124,-1.26081,2.96820,1.31333,-1.67967,-1.05827,1.03037,0.32389>,
      <1,0>],
     [<-0.30520,0.03195,0.38485,-0.10663,-0.71124,-1.26081,-0.21145,0.10121,-0.05366,-0.62340,0.36088,1.20885>,
      <1,0>],
     [<-0.76353,1.00000,0.38485,-0.10663,-0.71124,-1.26081,1.37838,0.40424,-1.67967,-0.18852,0.36088,0.32389>,
      <1,0>],
     [<-1.04899,1.00000,-0.80960,1.09145,1.53596,1.44189,-0.21145,-1.11091,1.57236,1.98587,-1.31934,-0.56106>,
      <1,0>],
     [<1.59418,-0.93611,1.18108,-1.30471,-0.71124,0.09054,-0.21145,1.01030,-0.05366,-1.05827,0.64225,0.32389>,
      <0,1>],
     [<0.99270,-0.93611,1.18108,-0.50599,-0.71124,0.09054,-0.21145,0.60624,-0.05366,-0.62340,0.91380,-0.56106>,
      <0,1>],
     [<0.50827,-1.90416,-0.41137,-0.10663,1.53596,0.09054,-0.21145,-0.80788,-0.05366,-0.18852,-1.20768,1.20885>,
      <0,1>],
     [<-0.36349,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,0.30321,1.57236,0.24636,-0.07548,-1.44602>,
      <0,1>],
     [<-0.57366,0.03195,0.38485,0.69209,0.41236,1.44189,-0.21145,-0.20182,1.57236,0.68124,-0.14694,-0.56106>,
      <0,1>],
     [<-0.63643,-0.93611,0.38485,1.09145,1.53596,1.44189,-0.21145,-0.90891,1.57236,1.11611,-1.31934,-0.56106>,
      <0,1>],
     [<-0.47360,-0.93611,0.38485,0.29273,0.41236,1.44189,-0.21145,-0.20182,-0.05366,0.24636,-0.53015,1.20885>,
      <0,1>],
     [<-0.88702,0.03195,0.38485,1.09145,0.41236,0.09054,-0.21145,-0.20182,1.57236,1.11611,-0.60161,-0.56106>,
      <0,1>],
     [<-0.70762,-0.93611,-0.80960,0.69209,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.68124,-1.15453,2.09381>,
      <0,1>],
     [<-0.56804,0.03195,-0.80960,0.29273,-0.71124,0.09054,-0.21145,-0.80788,-0.05366,0.24636,-0.43680,1.20885>,
      <0,1>]};
nops(TrainingSet1);
for TS in TrainingSet1 do
   RNANet3:-AddPattern(TS[1],TS[2])
end do:
RNANet3:-ShowPatterns();
RNANet3:-Get(fplot);
RNANet3:-Initialize(0.1);
RNANet3:-Train(35);
plots[listplot](%);
RNANet3:-TrainingResults();%[2];
RNANet3:-Classify
     (<2.70149,0.03195,2.77377,-2.50280,-2.95843,-2.61216,-3.39110,2.82848,-1.67967,-1.92803,2.51898,-0.56106>)
     ; %[1]+%[2]; #Red4.01 (left out of this TS)
```

```
RNANet3:-Classify
    (<1.07417,0.03195,0.38485,-0.90535,-0.71124,-1.26081,-0.21145,0.40424,-1.67967,-1.05827,0.36088,0.32389>)
    ; %[1]+%[2]; #Red4.04
RNANet3:-Classify
    (<-0.04566,0.03195,0.38485,-1.30471,-0.71124,-1.26081,1.37838,1.31333,-0.05366,-1.05827,1.43993,-1.44602>)
    ; %[1]+%[2]; #Blue4.10
RNANet3:-Classify
    (<-0.31290,0.03195,0.38485,-0.10663,0.41236,0.09054,1.37838,0.40424,1.57236,-0.18852,0.59714,-0.56106>)
    ; %[1]+%[2]; #Blue4.12
RNANet3:-Classify
    (<-1.24837,3.90416,-2.00406,1.89018,0.41236,0.09054,1.37838,-1.11091,-0.05366,1.98587,-1.18133,-1.44602>)
    ; %[1]+%[2]; #Blue4.30
RNANet3:-Classify
    (<0.14688,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-1.11091,-0.05366,0.24636,-0.60161,0.32389>)
    ; %[1]+%[2]; #Red4.14
RNANet3:-Classify
    (<-0.37170,0.03195,-0.80960,-0.10663,0.41236,0.09054,-0.21145,0.10121,-0.05366,-0.18852,0.18758,-0.56106>)
    ; %[1]+%[2]; #Red4.16
RNANet3:-Classify
    (<-0.58348,0.03195,-0.80960,0.29273,0.41236,0.09054,-0.21145,-0.50485,-0.05366,0.24636,-0.43680,-0.56106>)
    ; %[1]+%[2]; #Red4.17
RNANet3:-Classify
    (<-0.96015,1.00000,-0.80960,1.09145,0.41236,1.44189,-0.21145,-0.50485,-0.05366,0.68124,-0.43680,-1.44602>)
    ; %[1]+%[2]; #Red4.25
RNANet3:-Classify
    (<-1.06683,1.00000,-0.80960,1.49081,1.53596,0.09054,-0.21145,-1.11091,-0.05366,1.55099,-1.20768,-0.56106>)
    ;%[1]+%[2]; #Red4.28
RNANet3:-Classify
    (<-0.86698,0.03195,-2.00406,1.09145,0.41236,0.09054,-0.21145,-1.71697,-0.05366,1.11611,-1.18133,2.09381>)
    ; %[1]+%[2]; #Red4.29
```

Appendix B: Graphical Invariants

The graphical invariants used in this work were computed either by hand or by using Maple 14's Graph Theory package ([14]). Their normalized values may be seen in the Training Set code in Appendix A. Below, the original values of the graphical invariants are given. For definitions of notation used, please see Section 2.2.

Table 4: Graphical Invariants $1-3$ [18]

RAG ID	$J(G)$	$N_c(G)$	$diam(G)$
4.01	117.825	4	3.0000
4.02	96.351	4	2.5000
4.03	88.836	3	2.3333
4.04	75.222	4	2.0000
4.05	67.781	3	1.8333
4.06	73.089	3	2.3333
4.07	60.407	2	1.6667
4.08	66.935	4	2.0000
4.09	87.935	4	2.0000
4.10	45.905	4	2.0000
4.11	37.584	3	2.0000
4.12	38.908	4	2.0000
4.13	32.082	4	2.0000
4.14	50.945	4	1.5000
4.15	37.843	4	1.5000
4.16	37.369	4	1.5000
4.17	31.825	4	1.5000
4.18	30.438	3	2.0000
4.19	50.653	4	2.0000
4.20	39.110	4	2.0000
4.21	34.701	3	2.0000
4.22	27.111	5	2.0000
4.23	23.878	4	2.0000
4.24	28.575	3	1.5000
4.25	21.963	5	1.5000
4.26	32.229	4	1.5000
4.27	19.638	5	1.5000
4.28	19.170	5	1.5000
4.29	24.403	4	1.0000
4.30	14.418	8	1.0000

Table 5: Graphical Invariants $4-7$ [18]

RAG ID	$\lvert E(L(G))\rvert$	$max\{deg_{L(G)}(i)\}$	$\chi(L(G))$	$g(G)$
4.01	6	2	2	0
4.02	8	3	3	2
4.03	9	4	4	2
4.04	10	4	3	2
4.05	11	5	4	2
4.06	11	4	4	2
4.07	12	6	4	2
4.08	11	4	4	2
4.09	10	4	4	2
4.10	9	4	3	3
4.11	13	5	5	2
4.12	12	5	4	3
4.13	14	5	5	2
4.14	13	5	4	2
4.15	12	5	4	2
4.16	12	5	4	2
4.17	13	5	4	2
4.18	15	6	5	2
4.19	10	4	3	4
4.20	12	4	3	2
4.21	13	5	5	2
4.22	12	4	3	3
4.23	15	5	4	2
4.24	14	5	4	2
4.25	15	5	5	2
4.26	13	4	4	2
4.27	15	6	5	2
4.28	16	6	4	2
4.29	15	5	4	2
4.30	17	5	4	3

Table 6: Graphical Invariants $8-12$ [18]

RAG ID	$W(G)$	$\chi\prime(G)$	$F(G)$	$R(G)$	$\gamma(G)$
4.01	12.0000	3	16	2.8165	2.0
4.02	10.0000	3	18	2.5749	2.5
4.03	9.0000	4	20	2.3963	2.5
4.04	8.0000	3	20	2.3333	2.5
4.05	7.0000	4	22	2.1547	3.0
4.06	8.3333	4	22	2.4571	2.0
4.07	6.0000	4	24	1.9821	3.0
4.08	7.5000	4	22	2.3963	2.5
4.09	9.0000	4	20	2.6178	2.0
4.10	9.5000	4	20	2.5749	1.5
4.11	7.8333	5	26	2.2356	1.5
4.12	8.0000	5	24	2.3862	2.0
4.13	7.0000	5	28	2.2196	2.0
4.14	5.5000	4	26	2.1178	2.5
4.15	7.5000	4	24	2.2845	2.0
4.16	7.5000	4	24	2.2945	2.0
4.17	6.5000	4	26	2.1547	2.0
4.18	5.8333	5	30	1.9571	2.0
4.19	9.5000	3	20	2.4832	2.5
4.20	7.5000	4	22	2.3333	3.0
4.21	7.0000	4	26	2.1338	3.0
4.22	8.0000	3	24	2.3333	2.5
4.23	7.0000	5	30	2.1178	2.0
4.24	5.5000	4	28	1.9940	3.5
4.25	6.5000	4	28	2.1547	1.5
4.26	6.0000	4	26	2.1547	3.0
4.27	5.5000	5	34	1.9571	2.0
4.28	5.5000	4	32	1.9821	2.0
4.29	4.5000	4	30	1.9880	3.5
4.30	4.5000	4	34	1.9880	1.5

VITA

ALISSA A. ROCKNEY

Education:	B.S. Psychology and Mathematics, East Tennessee State University, Johnson City, Tennessee 2009
	M.S. Mathematical Sciences, East Tennessee State University, Johnson City, TN 2011
Professional Experience:	Graduate Teaching Assistant, East Tennessee State University, Johnson City, Tennessee, 2010-2011
	Math Lab Tutor, East Tennessee State University, Johnson City, Tennessee, 2009-2010
Publications:	R. A. Beeler, A. A. Rockney, and C. E. Yearwood, "The Infinigon and its Properties," *Congressus Numerantium*, **198** (2009) 39–49.
	R. A. Beeler and A. A. Rockney, "Decompositions of Directed Graphs Using Orientations of P_4 with a Pendant Edge," *Congressus Numerantium*, **202** (2010) 211–218.
	G. C. Blackhart, B. C. Nelson, A. Winter, and A. Rockney, "Self-control in Relation to Feelings of Belonging and Acceptance," *Self and Identity*, (2011) 152–165.

Awards: Student Travel Award, $400, for the Midsouth Computational Biology and Bioinformatics Society (MCBIOS), held at Texas A&M University, April 1-2, 2011

Travel Funding, $300, from the Graduate and Professional Student Association, East Tennessee State University, for travel to the Eighth Annual Conference of MCBIOS, held at Texas A&M University, April 1-2, 2011

Mathematics Award, Department of Mathematics and Statistics, ETSU, 2009

Faculty Award for the Department of Mathematics and Statistics, ETSU, 2009