5-2014

Information and Hardness Quantification of Graphs: A Computational Study

Brent Dutson
Utah State University

INFORMATION AND HARDNESS QUANTIFICATION OF GRAPHS: A COMPUTATIONAL STUDY

by

Brent Dutson

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

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Nicholas Flann Major Professor	Dr. Daniel Watson Committee Member
Dr. Curtis Dyreson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

Information and Hardness Quantification of Graphs: A Computational Study

by

Brent Dutson, Master of Science

Utah State University, 2014

Major Professor: Dr. Nicholas Flann
Department: Computer Science

With the advance of technology we are faced with more and more complex data. There are many examples of problems that are too large to solve. We need a new set of tools to analyze and manipulate this data. A recently developed algorithm, designed to determine how hard a problem is, could be one of these tools. The research done for this thesis selected a set of problems that could be solved, used a traditional depth first search to actually solve them, and measured how hard they really were. The same set of problems were then analyzed using the algorithm. This report compares the results to see how well the hardness measured by solving the problem correlates to the hardness value provided by the algorithm. It also looks at an expanded problem set to see if the algorithm could be considered general purpose, or if it is only effective with specific types of problems.

(157 pages)

PUBLIC ABSTRACT

A study of the effectiveness of new techniques in determining hardness of a problem

BRENT J. DUTSON

New techniques to measure the information contained within a network of interconnected nodes (such as links between computers in the Internet) have recently been developed. This work studies the relationship between the computer time needed to solve a common network problem and the information contained within the given network.

ACKNOWLEDGMENTS

I must begin by thanking Dr. Nicholas Flann and the entire Computer Science department at Utah State University. I have learned much and enjoyed the experience. Coming into the program with more than 20 years of professional software development experience, I was skeptical about how much I might actually learn. To my surprise and delight whole new worlds have been opened to me and I now realize that this journey of discovery will go on for the rest of my life.

As a non traditional student with commitments to full time employment, family, and other responsibilities, I have spent far longer in this process than I ever anticipated. Thanks again to Dr. Flann and the department for their patience and continuing support. They have stayed with me through thick and thin. I only hope the university does something nice with several years of tuition for continuing advisement.

Finally, special thanks for the love and support of my family. When my daughter was diagnosed with leukemia, my priorities changed and I seriously considered ending this educational journey. It has only been through their encouragement that I have continued on to this point. Thanks for letting me be absent on the many days and evenings required to complete this effort. My daughter is recovering from a second stem cell transplant, and we hope to have many more days together in the future. To each of my children, my two daughters-in-law, my two grandsons who haven't known a time when grandpa wasn't in school, and especially to my wife of more than 30 years, thanks for allowing me to have this experience. I only hope to be equally supportive as each of you pursue your dreams.

Brent J. Dutson

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We live in a world of big data, which becomes more complex every day. Researchers model the behavior of billions of cells within tissue to understand the characteristics of disease so we can better fight it [1]. Designers lay out integrated circuits with transistor counts that now reach into the billions [2]. Marketing groups collect and process data detailing the buying habits and preferences of millions of people. The Internet has an estimated 8.7 billion devices connected [3] creating a very complex network. Perhaps the field with the most exciting advances is biology, where large data sets help improve our understanding of what makes us tick, and where data related to topics such as the human genome are leading to advances in diagnosing and treating disease [4]. As each of these continue to grow in size and complexity it becomes more difficult to process and analyze the data.

To gain perspective on how quickly data becomes unmanageable, we can look at the real world example of package delivery as described by Marcus Wohlsen in Wired Magazine [5]. By traveling the shortest possible distance a delivery company can save time, fuel, and wear on vehicles, which saves money for the company. Less fuel use also reduces pollution, which is good for the environment and ultimately good for all of us. To make this possible we simply need to calculate the shortest route a driver can take to reach all destinations. Figure 1.1 shows the six possible routes if the driver has three destination. We can calculate this value by considering that upon leaving the office, the driver has a choice of three possible first stops. After that stop there is now a choice between the two remaining destinations. After the second stop, there is a single destination remaining. The algorithm to calculate the number of possible routes is shown in figure 1.2.

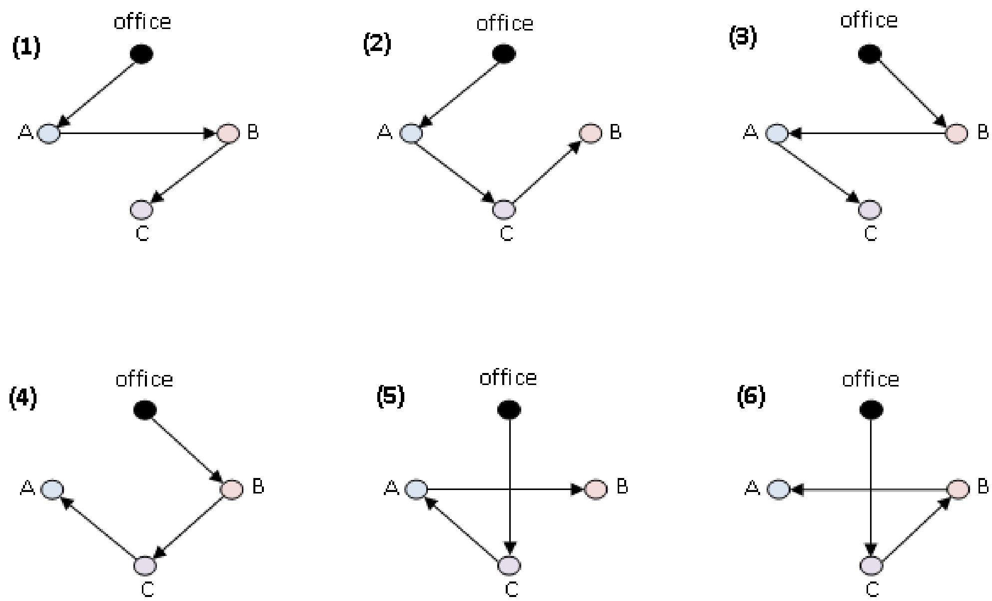


Figure 1.1: Possible Delivery Routes with Three Destination.

$$3 * 2 * 1 = 6$$

Figure 1.2: Possible Routes for Three Destinations

If we add a fourth destination, the possible number of routes increases to 24 as shown in figure 1.3. That is an exponential growth as we can see in table 1.1.

$$4 * 3 * 2 * 1 = 24$$

Figure 1.3: Possible Routes for Four Destinations

With 20 destinations we have 2,432,902,008,176,640,000 possible routes. If we could calculate 1 billion routes a second, it would take approximately 77 years to find all of them. Now consider that the average UPS driver makes 120 stops per day, and that UPS has approximately 55,000 delivery vehicles in the United States [5] and the size of the problem becomes enormous.

Even with computing power now available at a petaflop (10^{15} floating-point operations per second), many of these large problems cannot be solved as their size grows. All these problems have been shown to be NP Complete, and as explained in many studies

Table 1.1: Calculation for Number of Routes

Number of Destinations	Number of Possible Routes
3	3 * 2 * 1 or 3! = 6
4	4! = 24
5	5! = 120
6	6! = 720
7	7! = 5,040
8	8! = 40,320
9	9! = 362,880
10	10! = 3,628,800
15	15! = 1,307,674,368,000
20	20! = 2,432,902,008,176,640,000

on computational complexity [6] [7] [8], we will most likely never be able to solve these larger problems perfectly. Some faster algorithms such as an iterative repair hill-climber approximate an answer, where the longer the algorithm is allowed to run, the more likely the answer will be close to optimal. None of these approaches however, can be guaranteed to solve this or any of the other "big data" problems that are becoming common in many different problem domains. There is an urgent need for a new set of tools to manipulate, analyze, and in some fashion, improve the effectiveness of solvers for these problems.

One technique that has emerged in recent years that has the potential to improve problem solving effectiveness is a method for quantifying the information contained within a problem instance, know as Kolmogorov Complexity [9]. This method takes a graph, representing the problem instance, and returns a number known as Ψ that is maximized when the graph contains the most information. The specific method and intuition of the approach is described in more detail later in the report.

The hypothesis of this work is that by applying this information theoretic measure Ψ, insights may be gained specific to each problem instance that could be utilized to identify the best specific solution method or identify approaches to modify the problem to reduce its difficulty. The focus of this thesis is to discover, through experimentation and analysis, whether there is a relationship between the information quantification of a problem instance and the hardness of the problem, where hardness is measured as the number of steps needed by an exact solution algorithm to solve the problem. Table 1.2 identifies the possible outcomes of the research.

Table 1.2: Possible Significant Research Outcomes

Hardness	$\perp\!\!\!\perp$	Ψ
Hardness increases	$\Rightarrow$	Ψ *increases*
Hardness increases	$\Rightarrow$	Ψ *decreases*
Ψ *increases*	$\Rightarrow$	*Hardness increases*
Ψ *increases*	$\Rightarrow$	*Hardness decreases*

Hardness $\perp\!\!\!\perp$ Ψ**:** The $\perp\!\!\!\perp$ symbol represents independence and there is no relationship between the hardness of a problem instance and the information contained within that instance. While this result would be interesting, it would imply that the Ψ measure cannot help in predicting the hardness of a problem nor that hard problems tend to be information rich.

Hardness increases $\Rightarrow$ Ψ ***increases*****:** The $\Rightarrow$ symbol represents "implies" and then as the hardness of a problem instance increases, the information contained within the instance also increases. This result would be significant since it would provide insights into what makes a problem instance difficult to solve and what Ψ quantifies. However, since the implication goes from hardness to Ψ, it does not mean that all problems with high Ψ will have a high hardness.

Hardness increases $\Rightarrow$ Ψ ***decreases*****:** As the hardness of a problem instance increases, the information contained within the instance decreases. This case is similar to the above case. However, since the implication goes from hardness to Ψ, it does not mean that all problems with low Ψ will have a high hardness.

Ψ ***increases*** $\Rightarrow$ ***Hardness increases*****:** As the information contained within the problem instance increases, the hardness of the problem instance increases. If this were found to be true, the result would have major significance since it would provide a way to predict the hardness of a problem by running a simple and fast procedure over a given problem instance. In this case Ψ could be applied to NP Complete problems in general to help improve the run-time of solution methods. Given the pervasiveness of NP Complete problems of commercial, medical and scientific interest, this result could have a significant impact on all aspects of human endeavor.

Ψ ***increases*** $\Rightarrow$ ***Hardness decreases*:** As the information contained within the problem instance increases, the hardness of the problem instance decreases. This result would have a similar significance as the above result since problems with a low Ψ could be predicted to be hard to solve and thus worthy of further study and potential simplification to improve running speed.

The follow section lays out the methodology of the study to determine whether any of these relationships between Ψ and hardness hold.

CHAPTER 2
METHODS

To investigate the questions posed in the introduction, first a specific problem class must be chosen for study. In this work the well known NP Complete problem known as "graph coloring" was used. This problem is simple to understand and problem solvers are simple to implement. Further more, since the problem instances are undirected graphs, the Ψ measure can be directly computed rather than introduce the need for some kind of problem reduction technique that would translate another problem class into graphs.

The graph coloring problem can be described as:

Given: An undirected graph

Find: An assignment of color drawn from three distinct colors

Such That: No two nodes that are connected by an edge are assigned the same color.

Each graph coloring problem is solved using a standard depth first search to measure hardness. The information contained in the graph is calculated to produce a single value, referred to as Ψ. We will use Ψ to represent this value through the remainder of this report.

First the details of hardness and Ψ calculations are provided. Second, different classes of graphs are defined and described in detail: random, biconnected, scale free and small world. Finally the overall method of generating graphs of increasing hardness and increasing Ψ are described. Here a hill-climbing search is applied to maximize the objective function of hardness or Ψ

2.1 Calculation of Hardness

The depth of the search tree generated by the problem solver is bound by the number of nodes in the graph and the branching factor is bound by the maximum number of colors allowed to solve the problem. For this research, the color value was set at 3, meaning we were solving the 3-colorability problem. Initially the current color for each node was set to 0 to show that no color had been assigned yet. The three colors used throughout the search are represented by the number 0 for no color, 1 for the first color, 2 for the second color, and 3 for the third color.

The search begins by setting the color value of the first node to 1. Each adjacent node in the graph is then tested to see if it also has a value of 1, which would result in a color conflict. If no conflict is found, the depth level is increased by one by moving down to the next node and repeating the process of setting the node color and testing against adjacent nodes. If the bottom node has no conflict, then a solution has been found and the current graph is colorable.

When a color conflict is found, the color value of the current node is incremented and the testing is repeated. If all possible colors for the current node have a conflict, there is no value in continuing down the current path so we backtrack by setting the color value of the current node to 0 and moving back up one level. The process of backtracking will continue until we find a node that hasn't had all colors tried. If all colors have been tried for all nodes in the backtracking chain, then there is no solution to color this graph.

The hardness value used throughout this paper is a count of the number of times we were forced to backtrack. This value has nothing to do with whether the graph was colorable or not. A graph could be solved with very little backtracking, just as an unsolvable graph could find the conflicts quickly and require very little backtracking to discover that there is no solution. In either case, the hardness value will be low because very little work was required to arrive at the answer. On the other hand, a hard graph will require a lot of backtracking regardless of whether we ultimately find that the graph is solvable or not.

2.2 Calculation of Ψ

The algorithm used to calculate Ψ for this study was developed by Dr. David J. Galas primarily for use in the study of biological information contained in graph representation of biological data such as gene interaction diagrams [10] and gene regulatory networks [11].

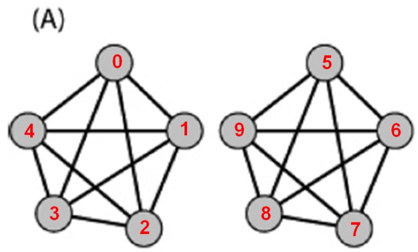


Figure 2.1: Example A of a simple undirected graph used to validate the Ψ code. The Ψ of this graph is 0.0766

Two papers with Dr. Galas as the lead author describe the development of the algorithm [12] [13]. Additional papers in which Dr. Galas is a contributor demonstrate how the algorithm is used in a biological systems setting [14] [15]. Dr. Flann, in conjunction with the Institute for Systems Biology in Seattle Washington, developed code to implement the algorithm. All instances of Ψ used in this report were calculated using that code. The following paragraphs describe, in some detail, the sequence of steps used in the calculation. Figure 2.1 shows an undirected graph with 10 nodes, taken directly from the papers by Dr. Galas. This graph was chosen because the value of Ψ was already known, which allowed us to verify that the results returned by the code were correct.

2.2.1 Direct Connections

In step 1 we determine the probability that a node i is or is not connected to another node j. For each node we will count the number of connections. The computation is shown with C as a connectivity matrix where $C_{i,j} = 1$ if node i is connected to j and $C_{i,j} = 0$ if not. $p_i(a)$ then shows the probability of node i being connected $(a = 0)$ or not $(a = 1)$.

$$p_i(a) = \sum_{j=0}^{n-1} \delta((C_{i,j} = a)\&(i \neq j))/(n-1)$$

For example, looking at Figure 2.1, node 3 is connected to four other nodes $(0, 1, 2, 4)$, and is not connected to five other nodes $(5, 6, 7, 8, 9)$. Since self connection is not allowed, we divide by the total number of nodes minus one $(n - 1)$. This shows that there is a 44.44

percent probability that node 3 will be connected to any other random node in the graph, and a 55.55 percent probability that it will not be connected.

2.2.2 Indirect Connections

In step 2 we determine the probability that a node i is connected to another node j through a third distinct node k. This probability $p_{i,j}(a, b)$ takes into account if node i is connected to k ($C_{i,k} = a$, where $a = 1$ if connected or $a = 0$ if not), and if node k is connected to j ($C_{k,j} = b$, where $b = 1$ if connected, or $b = 0$ if not). This results in four probability values for each combination of nodes, where the values of a and b are $(0, 0)$ or $(0, 1)$ or $(1, 0)$ or $(1, 1)$.

$$p_{i,j}(a, b) = \sum_{k=0}^{n-1} \delta((C_{i,k} = a)\&(C_{k,j} = b)\&(i \neq j)\&(i \neq k)\&(j \neq k))/(n-2)$$

As an example from figure 2.1, we will select $i = 3$ and $j = 7$. The remaining eight nodes $(0, 1, 2, 4, 5, 6, 8, 9)$ will be used as k. If k is one of nodes $0, 1, 2$, or 4, then the values of a and b are $(1, 0)$ and the first of our four probability values will be $4/8 = 50$ percent, where 4 is the number of values for k that match this condition, and 8 is the total number of nodes that are eligible to be k. If k is one of nodes $5, 6, 8$, or 9, then the values of a and b are $(0, 1)$ and the second of our four probability values will also be $4/8 = 50$ percent. The remaining two probability values will be 0 percent, since there is no case where a and b are $(0, 0)$ or $(1, 1)$. We now have our probability values for $i = 3$ and $j = 7$. This same process is repeated for every other combination of nodes. This computation is implemented as three nested for loops and thus dominates the computation time.

2.2.3 Shannon Entropy

In step 3 we used the direct connection probabilities from step 1 to calculate the Shannon Entropy (K_i) for each node. Shannon Entropy is commonly used in data compression and has been described as the number of bits required from each character in a message in order to have enough information to fully recover the message. A formal definition is available from a paper written in 1948 by Shannon [16]. The equation is:

$$K_i = -\sum_{a=0}^{1} p_i(a) \log_2(p_i(a))$$

2.2.4 Marginalize Probabilities

Step 4 sums up indirect connection probabilities from step 2 to create four marginalized probability values for each combination of a and b.

$$p'_{ij}(a, \cdot) = \sum_{b=0}^{1} p_{ij}(a, b)$$

$$p'_{ij}(\cdot, b) = \sum_{a=0}^{1} p_{ij}(a, b)$$

2.2.5 Mutual Information

Step 5 uses indirect connection probabilities from step 2 and marginalized probabilities from step 4 to calculate the mutual information. Given two random variables, mutual information is a measure of how much one of them tells us about the other. In other words, what proportion of information about the first variable intersects with information about the second [17].

$$M_{i,j} = \sum_{a=0}^{1} \sum_{b=0}^{1} p_{i,j}(a, b) \log_2 \left(\frac{p_{i,j}(a, b)}{p'_{ij}(a, \cdot) p'_{ij}(\cdot, b)} \right)$$

2.2.6 Calculate Ψ

Finally, step 6 uses the Shannon Information results from step 3 and the Mutual information results from step 5 to compute Ψ. The formula used is:

$$\Psi = 4 \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \max(K_i, K_j) M_{i,j} (1 - M_{i,j}) / (n(n-1))$$

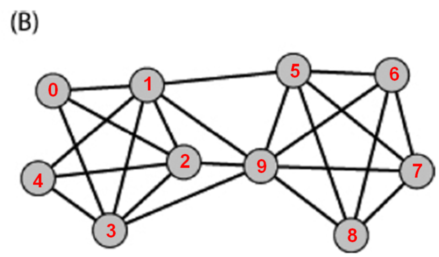


Figure 2.2: Example B of a simple undirected graph used to validate the Ψ code. The Ψ of this graph is 0.756

$$for 0 \leq \Psi \leq 1.0$$

Figure 2.2 shows a second version of the graph used to validate the computation. With several additional links, this version contains more information which should result in a larger value for Ψ. As seen from the results, this supposition holds true with $\Psi = 4 * 0.189 = 0.756$.

2.3 Problem Set

One of the first steps in doing the research was finding an acceptable set of problems to solve. Because of the nature of the algorithm we must be able to represent the problem as a graph. One of the first challenges is to convert a real world problem into a graph, which provides a model of the problem that can be analyzed, solved, and modified. Consequently, we must select a set of problems that can be converted to a graph.

The first problem set came from solving the satisfiability problem. We specifically selected 3-SAT problems. A set of 431 of these were downloaded from a 2003 SAT competition. Another set of problems were randomly generated. After writing code and running a large number of tests on these problems, the problem was found to be unsuitable. Each of these 3-SAT problems required an additional step to convert the logical expression into a graph structure. There was concern that this additional step could have tainted the results. The decision was made to abandon the 3-SAT problem for the research.

A second set of research problems were taken from graph coloring. With a problem set selected, the next step was to introduce variability into the problems to see if the results produced by the algorithm would be consistent, or if they would become more or less accurate as the nature of the problem changed. The decision was made to use the following graph classes in the research.

1. Random
2. Bi-Connected
3. Scale Free
4. Small World

The algorithms used to create and manipulate each of these graph classes are presented later in this section.

2.4 Data Set

Figure 2.3 shows a layout of the problem sets used in the research.

As shown in the figure, variability in the problems were introduced at three levels, the first being the type of graph. More will be said about graph classes later.

The second level is the number of nodes in the graph, referred to as node count. Node count was varied from 10 to 21 in increments of 1. The time required to actually solve a problem increases exponentially as the node count increased. With the available computer resources it was not practical to solve problems with node counts greater than 21.

The third level is connection probability, which affects the density of the graph. In general, this value defines the probability that there will be a link between any two nodes in the graph. The value was varied from 5% to 50% in increments of 5%.

Each combination of the three independent variables just described were repeated five times over graphs drawn randomly from the appropriate graph space. The results were averaged to give a more consistent result. The purpose of this was to minimize the affect of any extreme results that fell outside of a "normal" range.

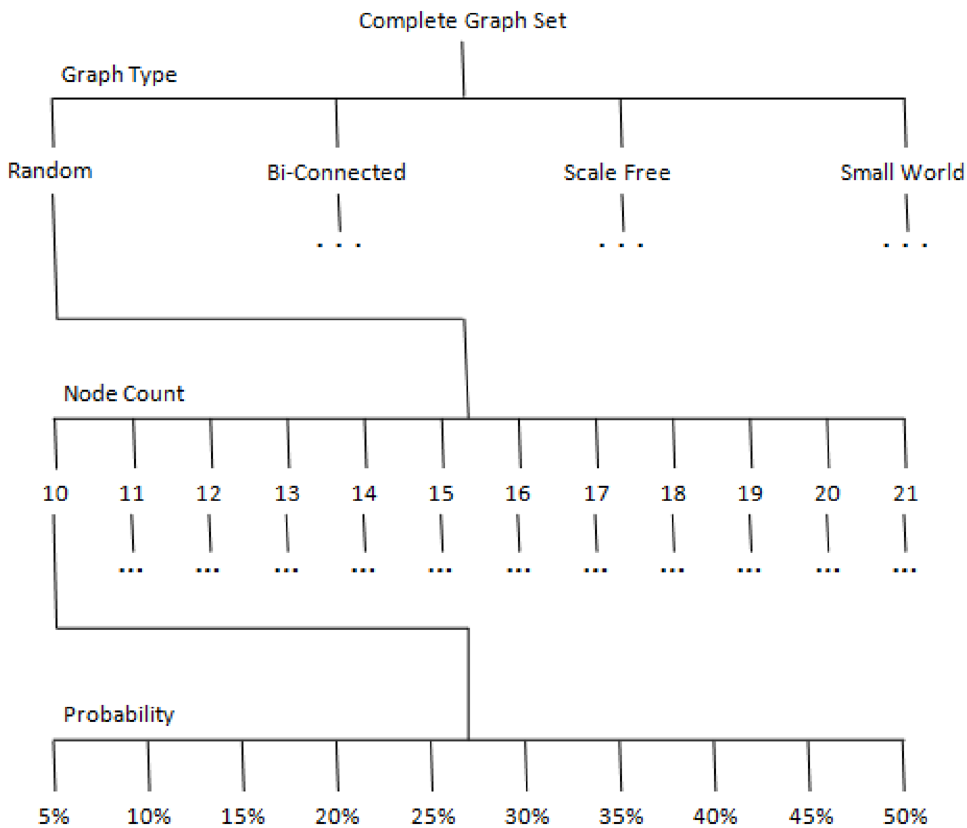


Figure 2.3: Working Directory Structure of the distinct experiments performed over the three independent variables.

2.5 Hill Climbing to Increase Hardness or Ψ

In the introduction the overall purpose of the study was described as the elucidation of the relationships between hardness and Ψ. Specifically, the questions ask whether increasing hardness or Ψ implies increasing/decreasing Ψ or hardness respectively. Recall that an implication is always in the form of: antecedent $\Rightarrow$ consequent. So to answer these questions a method was applied that deliberately generated a sequence of graphs that increases the antecedent and measured the consequent. For instance, to determine whether increasing hardness implies increasing Ψ, the method produces a sequence of graphs with monotonically increasing hardness and then for each graph we calculate and record the value of Ψ. Each graph in the sequence will have a hardness value which increases monotonically and a Ψ value that may or may not increase. The result of a run can be plotted as a trajectory on a scatter plot with one axis being the value of hardness and the other being Ψ. Once multiple trajectories are plotted on the graph, relationships between the two measures may be determined.

To generate this sequence of graphs with increasing antecedents a simple hill-climber algorithm is used where the objective function to be maximized is switched between hardness and Ψ. The method always begins with a randomly created graph and is repeated for each graph class and for each objective function as described earlier. For each of the graph classes, three algorithms were required to run the hill-climber. These are:

1. Create - Generate a graph that meets the requirements for the graph class.
2. Is_Valid - Check a graph to make sure it meets the requirements for the specified type.
3. Generate_Neighbor - Modify a graph slightly by adding or deleting an edge, then run Is_Valid to ensure that it still meets the requirements for the specified graph class.

With these three algorithms in place, we can then run the tests on each graph class by using the hill-climbing algorithm shown in figure 2.4.

The following sections describe each of the graph classes in more detail and then present the three required algorithms for each graph class. Much of the content in these algorithms was taken from existing works on graph theory [18] [19] [20].

2.5.1 Random Graphs

The random graph is introduced first as a generic graph class that could represent a variety of real world situations such as an arbitrary network [21]. These graphs give us a baseline that can be used for comparison with the other more specific types shown later on. Figures 2.5, 2.6, and 2.7 show the required three algorithms for random graphs.

```
Generate a new graph of the specified type (Create Algorithm)
Calculate initial values for hardness and Ψ
Set retryCount to 0
While retryCount < Specified Hill Climber Maximum Retries {
    Do until new graph is valid (or until we hit a 'find neighbor' retry
         limit) {
        Generate neighbor (Generate_Neighbor Algorithm)
        Check if new graph is valid (Is_Valid Algorithm)
        If not valid {
            Undo change (revert to previous graph - increment retry
                limit)
        }
    }
    Solve for monotonic value (Ψ or Hardness)
    If monotonic value is greater than previous value {
        Compute other (non-monotonic value)
        Save new values for hardness and Ψ in problem File
        Set retryCount to 0
    }
    else {
        Undo change (revert to previous graph)
        Increment retryCount
    }
}
Save final results in ".results" file
```

Figure 2.4: Hill Climbing Algorithm to run the tests. Note that to increase Ψ, it is set to the monotonic value to be maximized and hardness is measured. To increase hardness, the measures are reversed.

2.5.2 Bi-connected

A bi-connected network is one that can lose any link or edge from the network without any node becoming disconnected. Once again, this may have application in a network where the loss of any single link will not leave any customers without service. Figures 2.8, 2.9, and 2.10 show the required three algorithms for bi-connected graphs.

```
Set target connection probability p (5 <= p <= 50)
For each node n where 1 <= n <= max node count - 1 {
    For each node m where n < m <= max node count {
        If n and m are not already connected {
            Get random number r where 0 <= r <= 100
            If r <= p {
                Connect the nodes
            }
        }
    }
}
For each node (except the last node) {
    Verify there is some connection to the next neighbor node
    If no connection found {
        Add an edge between the 2 nodes
    }
}
```

Figure 2.5: Create a Random Graph of a specific size n and a specific connection density, set by the connection probability.

```
Start at any node
Traverse the graph using a depth-first search to make sure all other
    nodes are reachable
```

Figure 2.6: Validate a Random Graph

2.5.3 Scale Free

In Scale Free graphs the degree distribution follows a power law. A few nodes have a high degree while many nodes have a low degree. The high degree nodes are hubs [19]. A good examples of a scale free network is an electrical grid where a few hubs feed many smaller stations or in biological networks where key genes control significant developmental switches. Figures 2.11, 2.12, and 2.13 show the required three algorithms for Scale Free graphs.

```
Do until success or max retries {
    Select a random edge e for random node n
    Remove the edge
    Select 2 random nodes that aren't already connected
    Connect the random nodes
    If the graph is still valid {
        Return Success
    }
    else {
        Restore the original edge that was removed
        Increment the retry count
    }
}
```

Figure 2.7: Generate a Neighbor Random Graph

```
Set target connection probability p (5 <= p <= 50)
Create a ring network (primary links)
For each node n where 1 <= n <= max node count {
    For each node m where n < m <= max node count {
        If n and m are not already connected {
            Get random number r where 0 <= r <= 100
            If r <= p {
                Connect the nodes (secondary link)
            }
        }
    }
}
```

Figure 2.8: Create a Bi-connected Graph of a specific size n and a specific connection density, set by the connection probability.

```
For each edge in the network {
    Remove the edge
    If all nodes are still connected {
        The graph is valid
    }
    else {
        The graph is not valid
    }
    Put the removed edge back in
}
```

Figure 2.9: Validate a Bi-Connected Graph

```
Do until success or max retries {
    Select a random edge e for node n
    Remove the edge
    Select 2 random nodes that aren't already connected
    Connect the random nodes
    If the graph is still valid (still a bi-connected graph) {
        Success
    }
    else {
        Restore the original edge that was removed
        Increment the retry count
    }
}
```

Figure 2.10: Generate a Neighbor Bi-connected Graph

2.5.4 Small World

Small world graphs are recognized by their relatively short path length and high cluster coefficient [18]. The name reflects the "six degrees of separation" experiment that showed how closely we are all connected to everyone else on the planet. In other words, most nodes are not neighbors, but most nodes can be reached from every other node by a small number of hops or steps. These graphs occur in social networks [22] and gene-gene regulatory

```
Create a 3 nodes (scale free) graph
Do until we reach the target node count
{
    Create a new node
    Find the maximum degree count for any existing node (MaxDegree)
    For each existing node, calculate the probability that a link will
       be added between the new and existing nodes using p = (
       ExistingDegree/MaxDegree) x MaxProbability {
        Get random number r where 0 <= r <= 100
        If r <= p {
            Connect the nodes
        }
    }
    If new node has degree = 0 {
        Connect new node to the first node with the max degree
    }
}
```

Figure 2.11: Create a Scale Free Graph of a specific size n and a specific connection density, set by the connection probability.

```
Start at any node
Traverse the graph to make sure all other nodes are reachable
Use a depth first search
```

Figure 2.12: Validate a Scale Free Graph

```
Select a random node
Remove all existing links from that node
Follow the same procedure to reconnect as was done when adding new nodes
```

Figure 2.13: Generate a Neighbor Scale Free Graph

```
Set target probability p (5 <= p <= 50)
Create a 2-regular network (primary links)
For each node n where 1 <= n <= max node {
    For each node m where n < m <= max node count {
        If n and m are not already connected {
            Get random number r where 0 <= r <= 100
            If r <= p {
                Connect the nodes (secondary link)
            }
        }
    }
}
```

Figure 2.14: Create a Small World Graph of a specific size n and a specific connection density, set by the connection probability.

```
Start at any node
Traverse the graph to make sure all other nodes are reachable
Use a depth first search
```

Figure 2.15: Validate a Small World Graph

networks [23]. Figures 2.14, 2.15, and 2.16 show the required three algorithms for Small World graphs.

```
Select 2 random nodes
If the 2 nodes have a secondary link {
    Remove the link
}
Select 2 random nodes
If the 2 nodes have no connection (primary or secondary) {
    Add a secondary link
}
```

Figure 2.16: Generate a Neighbor Small World Graph

2.6 User Interface

The user interface is made up of two main screens. The first screen allows the user to select the location where the collected data should be stored, and select the type of graph to be used. The second screen is divided into two halves. On the left side the user can enter other parameters such as the probabilities and node counts to use in the tests. The right hand side displays progress data as the tests are run. Figures 2.17 and 2.18 show these two screens.

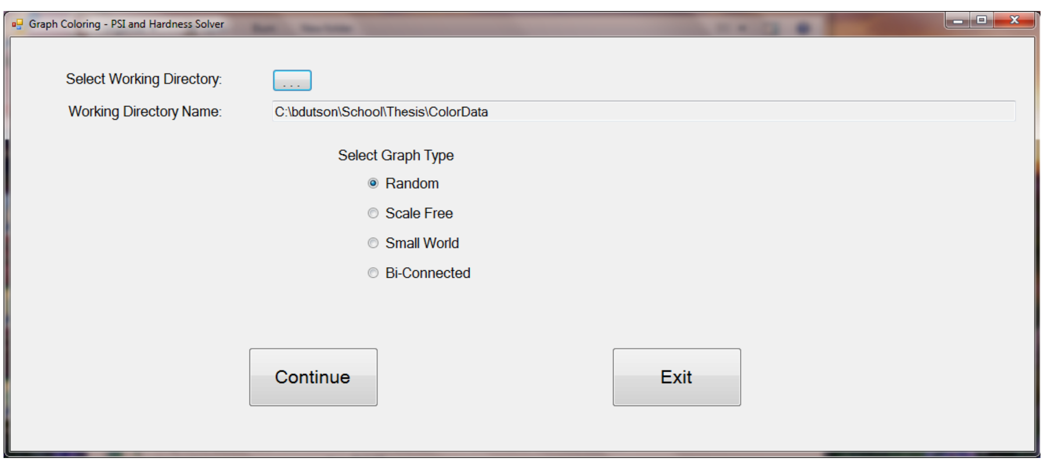


Figure 2.17: Main User Interface Screen

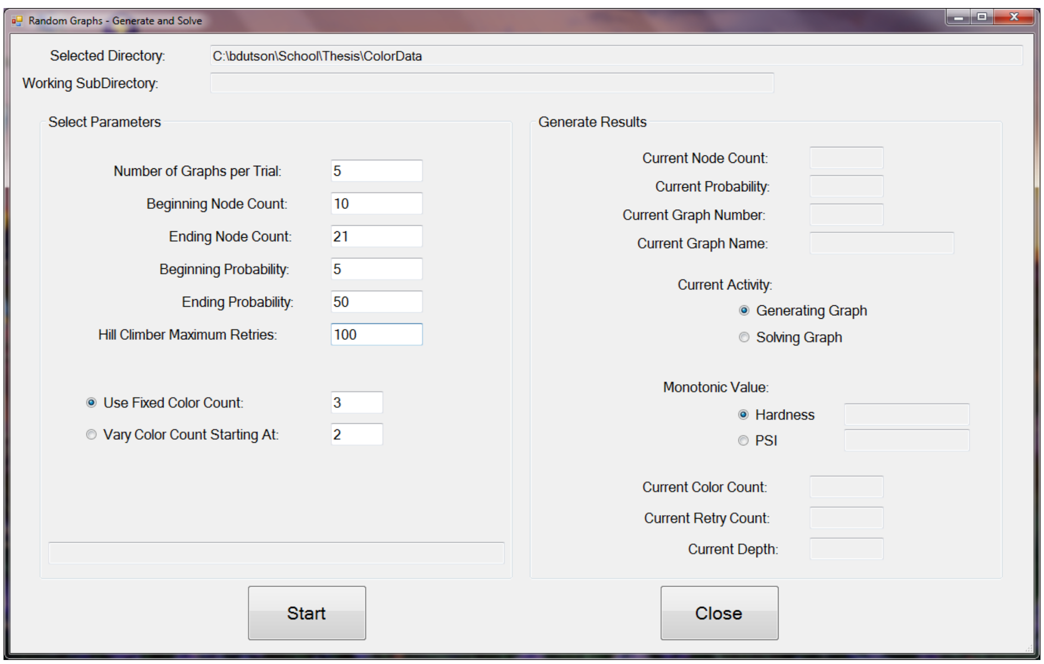


Figure 2.18: Operation Management Screen

CHAPTER 3
RESULTS

Figure 2.3 shows the various independent variables (parameters) that were used during experimentation to measure their effect on Ψ and hardness. The results in this section are presented in the order outlined in that diagram, moving from the simplest case to the most complex. Each section includes four graphs, one for each of the graph classes. The results are presented in the following order:

- Section 3.1 considers individual trials for a single run of the hill-climber Ψ/hardness maximization algorithm. For each graph class, a single solution is randomly generated and utilized as the starting state for both Ψ/hardness maximization. This gives a preliminary look at a raw sample from the results.

- Section 3.2 considers Ψ maximization over the four graph classes each with 10 individual runs for a graph density of 30% and node count of 18.

- Section 3.3 considers the hardness maximization over the four graph classes each with 10 individual runs for a graph density of 30% and node count of 18.

- Section 3.4 considers the influence of connection density on the hardness maximization results.

- Section 3.5 considers the influence of node count on the Ψ and hardness values over the four graph classes each with 5 individual runs.

- Section 3.6 uses heat maps to illustrate the combined effect of connection density and node count on the values of hardness and Ψ (hardness maximization).

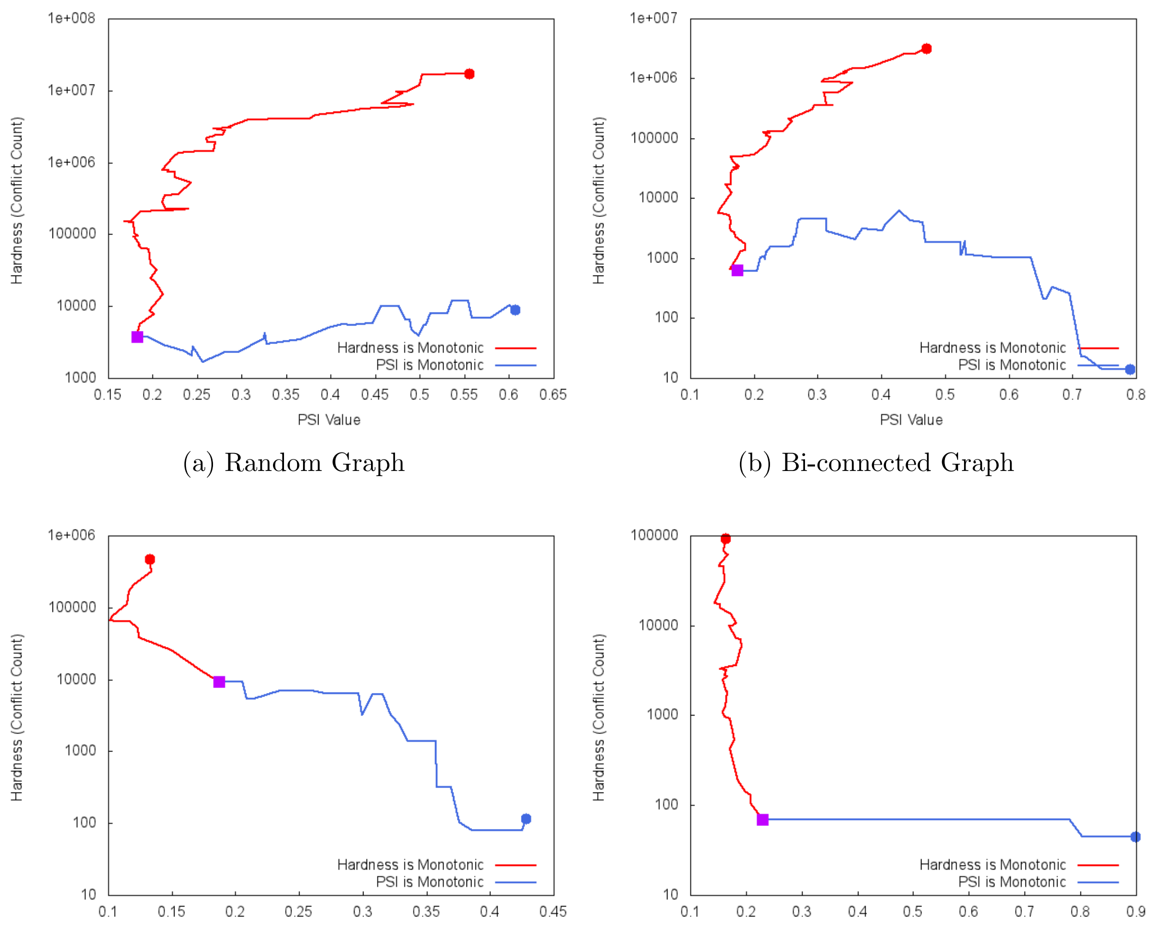
- Section 3.7 provides a comparison of execution times on the run time required to solve problems vs. the time required to run the Ψ algorithm as a function of node count, to demonstrate that running the Ψ algorithm will be significantly faster than calculating hardness.

Each trial during the experimentation was run twice, the first time with hardness being the objective function that is held monotonically increasing, and the second with Ψ. As described in the methods section, a single run with hardness being monotonic would involve solving the initial graph for both hardness and Ψ, saving the results, and then running the hill climber algorithm to find a neighbor with a larger hardness value. This neighbor would then be solved for Ψ. This continues until a harder neighbor cannot be found in a reasonable number of tries. The result of this process is that the hardness value always increases, and thus is monotonic, while the Ψ value doesn't necessarily increase. This entire process was then repeated with Ψ now being the monotonic value. The first set of plots (individual trials) show both sets of results. This process was repeated for multiple trials and illustrated in Sections 3.2, where Ψ is maximized, and 3.3, where hardness is maximized. As the results will show, it became clear that when hardness is the monotonic value, there is evidence of a relationship between hardness and Ψ for certain graph classes. When Ψ was the monotonic value however, there was not a clear relationship between the two.

Throughout all of the graphs, results for the hardness evaluation grows in an exponential fashion and so these results are always plotted on a logarithmic scale. Other values such as Ψ grow in a polynomial fashion, and so they are plotted on a linear scale. This was true even for the aggregated data in heat maps.

3.1 Individual Trials

Although literally thousands of individual trials were run during the experimentation, as an introduction to the data only four (one for each graph class) have been selected to illustrate the result produced with each individual run. Those selected have a connection probability (a measure of graph density) of 30 percent because this is in the range where the more difficult problems were found (see Section 3.4). A node count of 18 was selected because it represented one of the largest problem sizes that is feasible to solve. Of the 10 trials run at this level, the 5th trial was selected. The selection was done before looking at the actual plots to avoid the temptation of picking certain plots because they looked more interesting, which is known as "cherry picking."



(a) Random Graph

(b) Bi-connected Graph

(c) Scale Free Graph

(d) Small World Graph

Figure 3.1: Individual Trial. Results using node count = 18, connection probability = 30 percent, trial run number 5. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Each point on the graph represents an individual graph created during optimization search. Both searches start at the same graph. Note the square point represents the initial randomly sampled graph and the star represents the final graph produced by the hill-climber.

Figure 3.1a uses hardness as the monotonic value (always increasing) in the red plot. For this line the value of Ψ was measured for each graph created during search and is therefore not monotonic and so we see several individual points where the value of Ψ decreases. Overall there does appear to be a tendency for the Ψ value to track the increase in hardness. The blue plot shows that when Ψ is the monotonic value the corresponding hardness value does not appear to follow it at all. There is not even a trend in that direction. It is important to note however, that these graphs are to illustrate the effect of hill climbing search and the choice of the monotonically increasing objective only. Since they are single runs, no general conclusions concerning a potential relationship between Ψ and hardness can be drawn.

3.2 Multiple Trials maximizing Ψ Monotonically

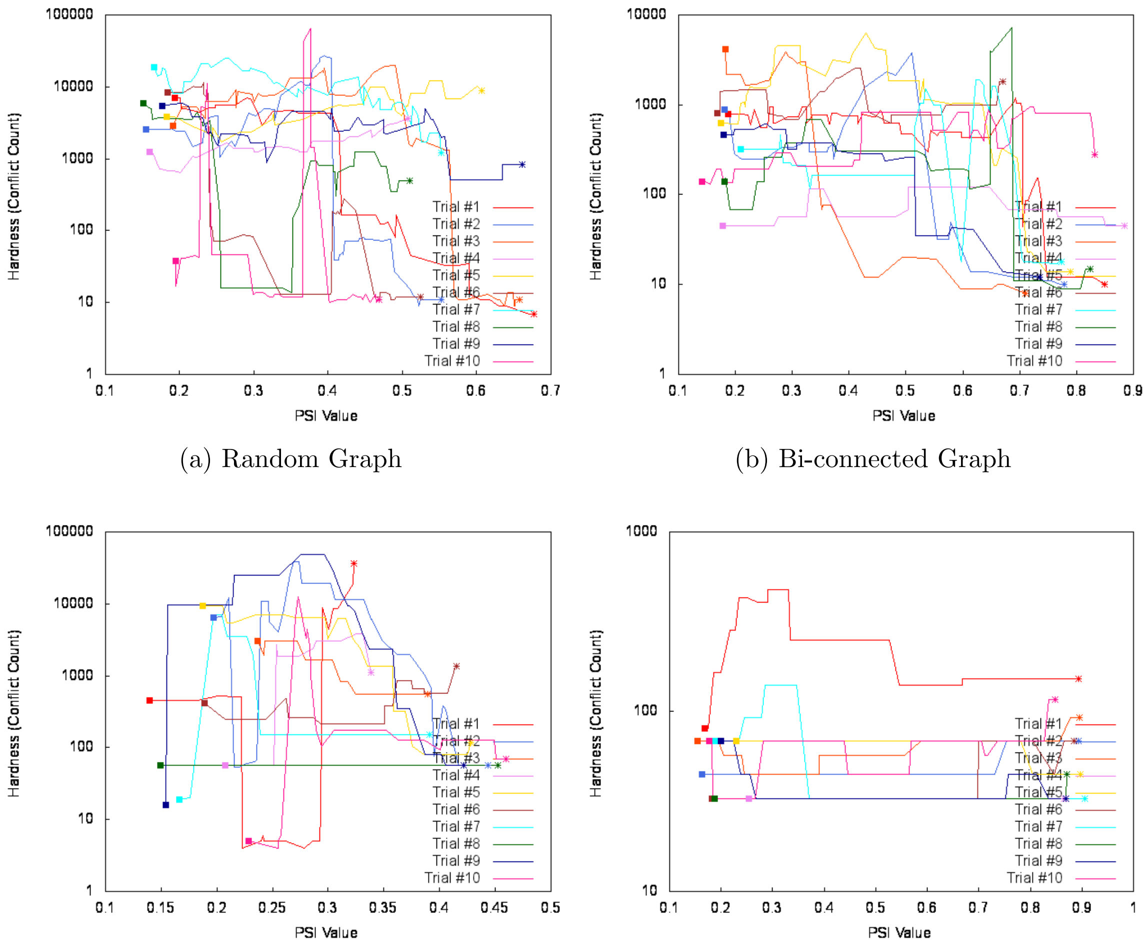
Figure 3.2 illustrates 10 trials run with a connection probability of 30 percent and a node count of 18 when Ψ was the monotonically maximized value. This view shows that there appears to be no relationship over the randomly sampled graphs for the given parameters and all the graph classes.

3.3 Multiple Trials maximizing Hardness Monotonically

These plots show 10 trials run with a connection probability of 30 percent and a node count of 18. This view shows that the results are consistent over randomly sampled graphs for the given parameters. If they were not consistent, then the results we saw in the individual trials have no real meaning.

The comparison of 10 trials in figure 3.3a reinforces the observation we made for the single trial for a random graph in figure 3.1a. In general, the value of Ψ appears to follow hardness, but not in every case.

Figure 3.3b shows that multiple trial runs on a bi-connected graph leads to the same conclusion found with the single run in figure 3.3a. For this graph class there appears to be a relationship between Ψ and hardness when hardness is the monotonic value.



(a) Random Graph

(b) Bi-connected Graph

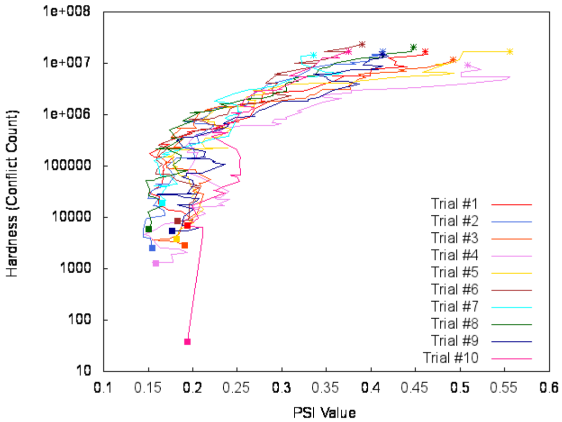
(c) Scale Free Graph

(d) Small World Graph

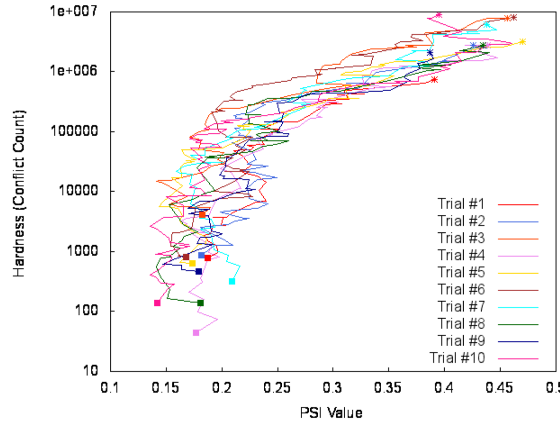
Figure 3.2: Individual Comparisons when optimizing for Ψ. Results using node count = 18, connection probability = 30 percent. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Note in each run Ψ is monotonically increasing.

3.4 The Influence of Connection Density

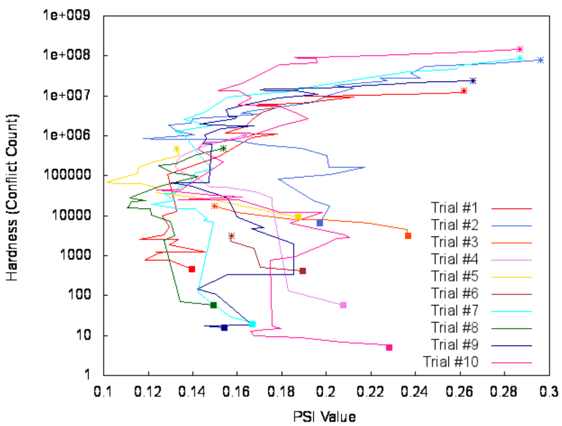
The next set of plots continue to use trials with a node count of 18, but now show the effect of changing the density of the graphs by changing the probability that two nodes will be connected by an edge, referred to as the connection probability. In these studies, all results are for the condition where hardness is maximized. Each of the individual trials have been combined into an average value for each connection probability level. The plots we have seen to this point used raw data from individual trial runs. All plots from now on



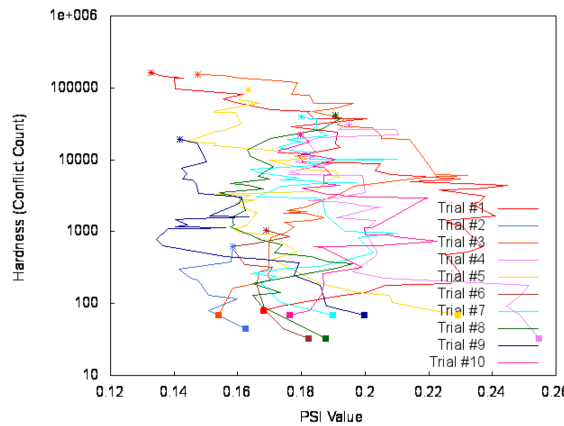
(a) Random Graph



(b) Bi-connected Graph



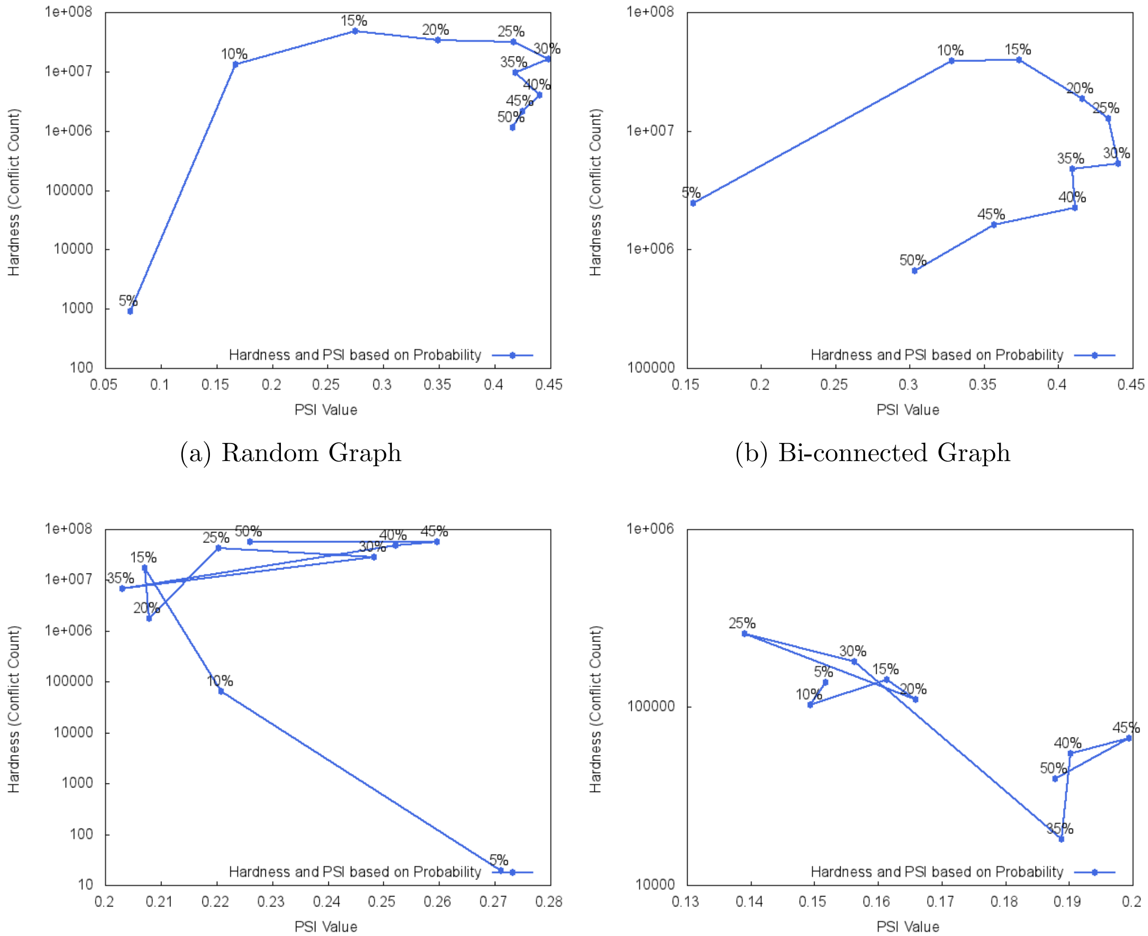
(c) Scale Free Graph



(d) Small World Graph

Figure 3.3: Individual Comparisons when optimizing for hardness. Results using node count = 18, connection probability = 30 percent. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Note in each run hardness is monotonically increasing.

will use averages, taken from five trial runs for each combination of graph class, connection probability, and node count. By using averages, we minimize the effect of individual variation that might occur. For example, in a depth first search a single graph might coincidentally be solved with no backtracking, giving it a hardness value of zero. Another graph might do an unusually large amount of backtracking, giving it a very high hardness value. By averaging five graphs for each set of input parameters the effect of these unusual cases will be minimized in the final result. The averages based on probability has been plotted to show the relationship between hardness and Ψ based on the connectivity of the graph.



(a) Random Graph (b) Bi-connected Graph

(c) Scale Free Graph (d) Small World Graph

Figure 3.4: The influence of connection probability on hardness and Ψ under hardness maximization. Results using node count = 18. Probability ranges from 5 percent to 50 percent in steps of 5 percent. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale.

The results in figure 3.4a show a relationship between hardness and Ψ. At a low connection probability the number of edges in the graph are at a minimum and so the graph has low information and the problem is under constrained. Whether the coloring problem can be solved or not, it should not take very long to resolve, so the problem is not very hard. Both the hardness and Ψ values reflect that. As the probability increases and more edges are added to the graph the problem becomes harder and Ψ increases. Again, the hardness and Ψ values reflect that. As more edges are added, we reach a point where the problem actually starts to get easier to solve. In the case of the coloring problem that may be because the problem is over constrained, thus the problem becomes easier. The interesting result for the random graph is that the hardness value peaks around 15 percent probability, but the Ψ value peaked around 30 percent. Both values followed the same pattern, but reached maximum hardness at different points.

As seen in earlier examples, figure 3.4b shows that the results for the bi-connected graph are similar to those of the random graph. Again the hardness value peaks around 15 percent, and the Ψ value peaks around 30 percent.

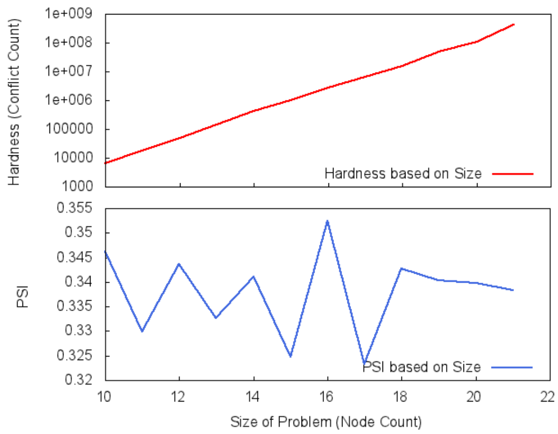
Figure 3.4c shows no obvious pattern between hardness and Ψ regardless of the connection probability value. The interesting point in these results is that the hardness value peaks at 50 percent connection probability, while the Ψ value peaks at 5 percent.

Scale free and small world graphs show a completely different relationship between hardness and Ψ as a function of connection probability. Figure 3.4d shows the results for small world graphs. There appears to be no relationship between hardness and Ψ.

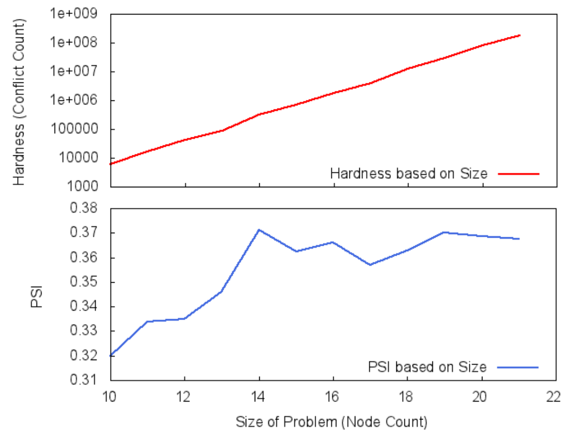
3.5 The Influence of Node Count

All results plotted before now have used the size of 18 nodes since it represents one of the largest feasible graph sizes for coloring. In this study the range of connection probability values for each node count has been averaged into a single value, and these values are then plotted to show what effect graph size (node count) has on the Ψ and hardness values.

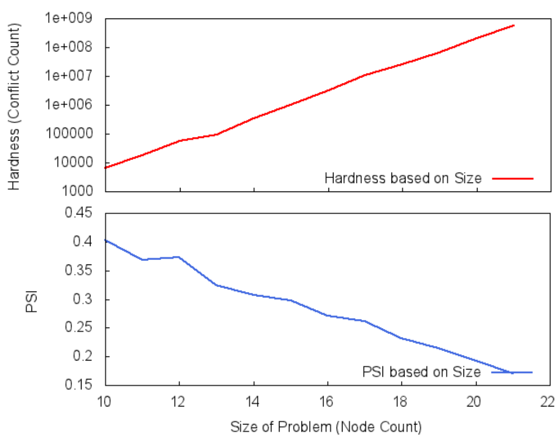
As the node count goes up, figure 3.5a shows that the hardness value grows exponentially since the slope of the hardness graph is constant when illustrated on a semi linear plot. This is not a surprise, considering that a depth first search was used to calculate the hardness value. In the case of the random graph there is no corresponding increase in Ψ value. This is somewhat surprising, since all previous measures on random graphs show a positive relationship between hardness and Ψ. The only conclusion we can make is that the value of Ψ is not dependent on the node count of a graph, at least when dealing with a random graph.



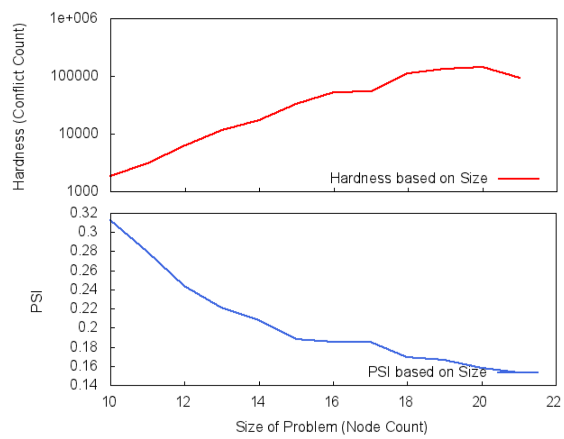
(a) Random Graph



(b) Bi-connected Graph



(c) Scale Free Graph



(d) Small World Graph

Figure 3.5: Comparing the influence of node count on hardness vs. Ψ. X axis for both plots shows node count on a linear scale. Y axis for the top plot shows hardness on a logarithmic scale. Y axis for the bottom plot shows Ψ on a linear scale.

Figure 3.5b shows the anticipated exponential increase in the hardness value as the node count increases. The value of Ψ follows the same increasing trend for the bi-connected graph, but considering that Ψ is shown on a linear scale the increase is not as dramatic.

Figure 3.5c shows a surprising inverse relationship between hardness and Ψ for scale free graphs. As the hardness value goes up with increasing node count, the Ψ value trends down. The reason for this is an open problem.

Figure 3.5d shows the same inverse relationship between hardness and Ψ for small world graphs as we saw in the scale free results in figure 3.5c. Again, there is no obvious explanation for these results. The other interesting note is that the hardness value actually decreases at the highest node count. At higher node counts the depth first search did less backtracking, which would indicate that it was able to find a solution or prove that there was no solution earlier on. Further research would be required to determine which is the case.

3.6 Aggregated Analysis

This section uses heat maps to represent a summary of all collected data for when hardness is maximized. Each figure shows two maps; one for hardness and one for Ψ. Node count is used as the x axis and connection probability as the y axis. The entire range of hardness values was divided into 10 sections and a heat value (0-9) was assigned to each, with 9 being the largest (hardest) values and 0 being the smallest. A single average value was then calculated for each combination of node count and connection probability. This average was then assigned a heat value based on the range it fell into. These values were then plotted on the heat map. The high heat values are represented by red (a "hot" color) while low values are represented by blue (a "cool" color). Values in between are gradual color steps from red to blue using the familiar "jet" pallet in Matlab. The same process just described was then followed for the Ψ values. The heat values for hardness were calculated on a logarithmic scale, while the heat values for Ψ were calculated on a linear scale.

Calculating the heat map values for hardness on the Random graph class is shown as an example in the following steps:

1. Basic Data: The five trials that were run at each connection probability and node count level are averaged to get a single hardness value.

2. Determine the range of data: From the lowest level averages calculated in the previous step, find the minimum and maximum values for hardness. This gives a range of possible values. For the random graphs the minimum hardness value seen was 5, while the maximum was $\approx 1.7 \times 10^9$. This range is then broken into 10 segments with a heat value assigned to each segment. The value 0 is assigned to the minimum

segment, and the value 9 is assigned to the maximum segment as shown in table 3.1. Note that the hardness scale is logarithmic.

Table 3.1: Hardness Ranges for the Heat Plot

0	1 - 10
1	11 - 100
2	101 - 1,000
3	1,001 - 10,000
4	10,001 - 100,000
5	100,001 - 1,000,000
6	1,000,001 - 10,000,000
7	10,000,001 - 100,000,000
8	100,000,001 - 1,000,000,000
9	1,000,000,001 - 10,000,000,000

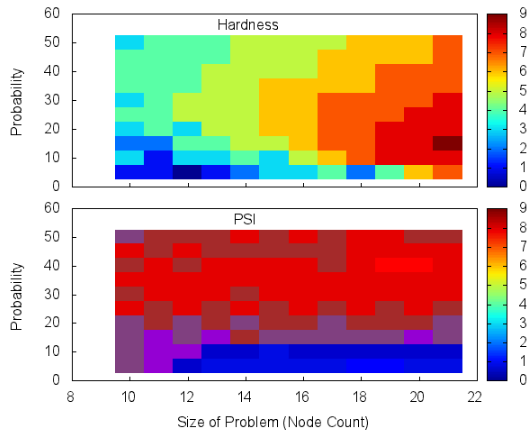
3. Assign Heat Values: Assign a heat value (0-9) for each combination of node count and connection probability.

4. Create Data File: Each heat value assigned in the previous step now becomes a line in a data file. The line consists of three values which are node count (x axis), connection probability (y axis), and heat value (z axis).

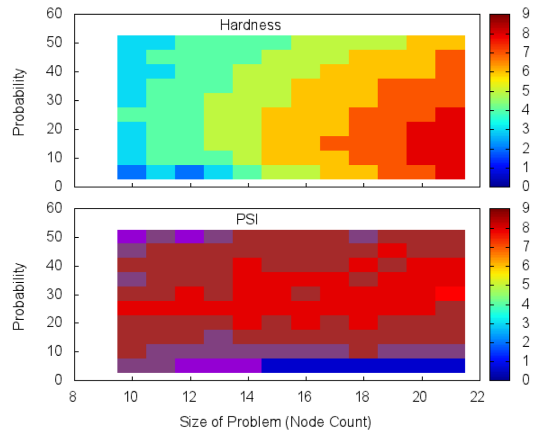
5. Create the Plot: The plot can now be created using the data file as input. Blue is used for low numbers (cool) and red for high numbers (hot) with values in between using gradual color steps as defined by the "jet" pallet.

3.6.1 Discussion of Aggregate Results

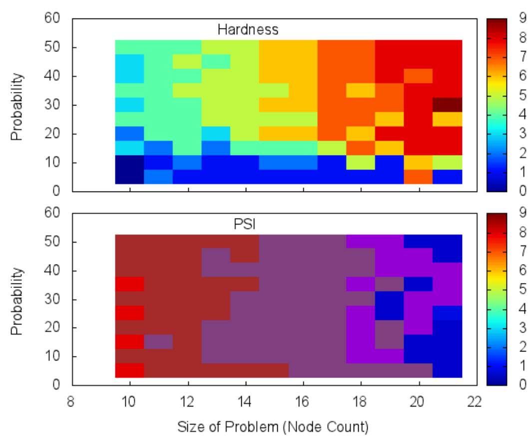
Random graphs: With all parameters in play, the heat maps of figure 3.6a show no pattern between hardness and Ψ for random graphs. The hardness value steadily increases with node count and is consistently higher at the middle ranges of connection probability. While Ψ values are also higher at the middle connection probability levels there appears to be little if any relationship between Ψ and the node count.



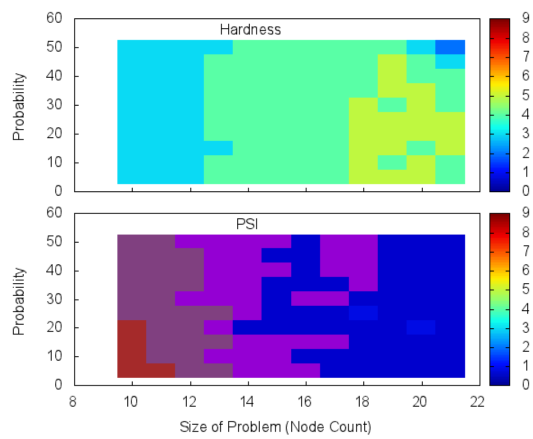
(a) Random Graph



(b) Bi-connected Graph



(c) Scale Free Graph



(d) Small World Graph

Figure 3.6: Aggregated Data. Heat plot showing the effect of connection probability and node count on both hardness and Ψ. X axis for both plots shows node count on a linear scale. Y axis for both plots shows connection probability on a linear scale. Colors in the top plot represent hardness values. Colors in the bottom plot represent Ψ values.

Bi-connected graphs: Figure 3.6b shows very similar results for bi-connected graphs as those seen in figure 3.6a for random graphs. The hardness value follows the expected pattern, but the value of Ψ tends to increase with connection probability, but not with node count.

Scale free graphs: In figure 3.6c we see the expected results for hardness, increasing with node count and connection probability. These aggregated results make apparent an interesting and potentially significant result: *scale free graphs of connectivity probability* *greater than 10% with low Ψ values tend to be hard.*

Small world graphs: In Figure 3.6d we see the familiar relationship between node count and hardness. Again we see an interesting and potentially significant result: *small world graphs with high Ψ tend to be easy and conversely, graphs with low Ψ tend to be hard.*

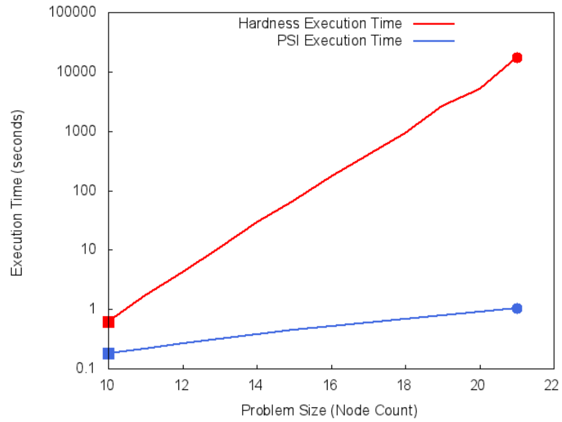
3.7 Comparison of Execution Times

For every trial, the execution times for both the hardness and Ψ calculations were measured. The final set of plots compares the average of these execution times. If the time required to calculate Ψ is not significantly smaller than the hardness calculation time, there may be little point in using the algorithm to provide insights into anticipated hardness. In addition, if the time required to calculate Ψ increases as the problem size grows there will be a point where the algorithm may take too long.

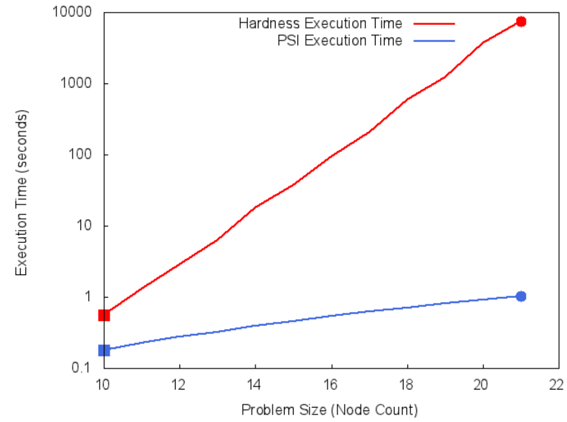
Figure 3.7 shows a clear relationship between node count and execution time. Keeping in mind that execution time is shown on a logarithmic scale, the time required to calculate the hardness value increases exponentially as the node count goes up. During the experimentation phase the maximum node count was limited to 21 because beyond that the time to solve one problem exceeded 10 hours. This is why Ψ could be valuable. Under some circumstances (small world and scale free graphs) it may allow us to estimate how hard a problem is by running an algorithm in much less time than would be used to actually solve the problem. We notice in figure 3.7a that the time required to calculate Ψ increases as the node count increases. Analysis shows that the complexity of the Ψ calculation grows at $O(n^3)$ compared to $O(3^n)$ for colorability. This means that Ψ can be quickly calculated for all feasible colorability problems.

Figure 3.7b shows results for the bi-connected graphs that is almost identical to those discussed for the random graphs in figure 3.7a. Figure 3.7c shows results for the scale free graph that is similar to those shown in figures 3.7a and 3.7b.

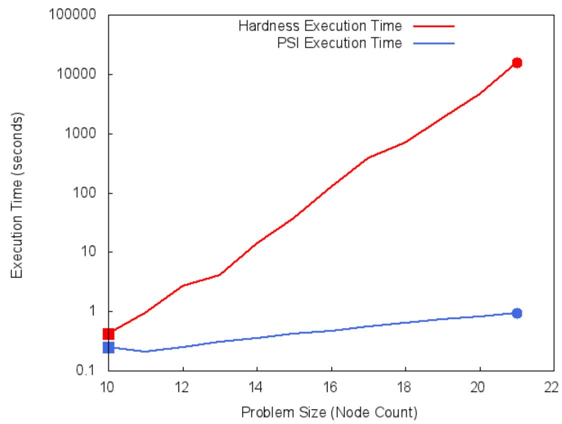
Although an initial look at figure 3.7d is similar to the previous graphs it actually answers some persistent questions related to the small world graph. The maximum execution time to calculate hardness for any small world graph was less than 10 seconds compared



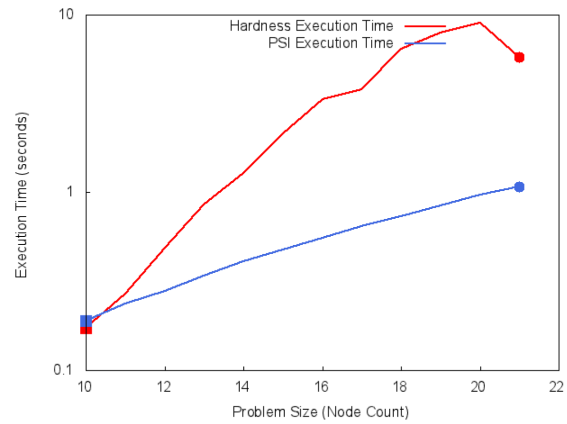
(a) Random Graph



(b) Bi-connected Graph



(c) Scale Free Graph



(d) Small World Graph

Figure 3.7: Comparison of execution time required to calculate hardness vs. time to calculate Ψ. X axis shows node count on a linear scale. Y axis shows execution time measured in seconds, on a logarithmic scale.

to the other graph class that all required more than 10×10^3 seconds to solve the most difficult cases. The nature of the small world graph allows all problems to be solved quickly because path lengths are small, limiting the depth of the search tree during the depth first problem solver. This explains why all of the small world results have been so different when compared to results for the other graph classes.

CHAPTER 4

CONCLUSIONS

The primary objective of this work was to explore whether there is a relationship between the information contained within a graph Ψ and the hardness of solving an NP Complete problem over that graph. As a further extension of this objective, four specific classes of graphs were identified and studies were performed on each one. A method was introduced that applied hill-climbing to generate sequences of graphs with monotonically increasing hardness or monotonically increasing Ψ to elucidate any relationship that may exist. Additionally, studies were performed investigating the effects of graph size and density on hardness and Ψ for each graph class.

First a series of studies were performed to better understand the influence of the node count and connection density of graphs in each class.. Here results demonstrated that the hardness of a graph problem tends to increase exponentially with the node count as expected, see Figure 3.5 and Figure 3.7. Based on this study the maximum node count for problem instances was set at 21 with most experiments applied to graphs of node count 18. The effect of graph density was studied and differences were found among the graph classes, see Figure 3.4. For random and bi-connected graphs hardness was maximized with a density of around 20% to 30% which makes sense since graphs with too few connections will be under constrained and easy to color while graphs with too many connections are over constrained and can quickly be determined to have no valid coloring. In contrast, the hardest scale-free graphs to solve occur over a broader range when the density is between 15% and 50%. This result is due to the inherent distribution of node degree and so the graph will be comprised of a mixture of over and under constrained subgraphs. Interestingly, small world graphs were shown to be universally easy to solve independent of their density due to their short path lengths. A graph with short path lengths will result in very shallow search trees when solved by the backtracking algorithm and hence quick solution times.

Interesting results were obtained in the study of the relationship between Ψ and the number of nodes in scale free and small world graphs, illustrated in Figure 3.5. Here an inverse relationship was observed implying that as the graphs get larger, the information contained in the graph shrinks. This is counter-intuitive since it would be expected that as a graph grows it would contain more information. The reason for this relationship is unknown and merits further study.

Returning to the key objective of the study, results suggest that indeed there are relationships between hardness and Ψ at least for some graph classes.

First consider for which graph class there appears to be no relationship between Ψ and hardness, introduced formally as: $Hardness \perp\!\!\!\perp \Psi$. Reviewing the data collected for multiple runs where hardness is maximized, illustrated in Figures 3.3, and runs where Ψ is maximized, illustrated in Figure 3.2 it appears that this is true for two of the graph classes: scale free graphs and small world graphs. In Figures 3.2(c) and (d) the hardness of the graph switches from high to low randomly as changes are made that increase the value of Ψ. Likewise, in Figures 3.3(c) and (d) the Ψ value appears to mostly decrease then increase again as the hardness of the graphs is increased. Although some of the graphs optimized for hardness have high Ψ the pattern is not consistent.

The most sought after result is whether there is a relationship where changing Ψ implies changes in hardness, introduced formally as Ψ *increases* $\Rightarrow$ *Hardness increases*, or Ψ *decreases* $\Rightarrow$ *Hardness increases*. If this result were found it could have a profound effect on improving the efficiency of problem solvers for NP Complete problems in general since Ψ, which can be calculated quickly, could be applied to predict which problems are difficult to solve. Unfortunately, based on this preliminary study, there appears to be no such relationship for any of the graph classes. This conclusion is clear from a review of all four graphs illustrated in Figure 3.2 where as Ψ increases monotonically changes in hardness appear random.

Turning now to whether increases in hardness implies changes in Ψ, stated formally as

Hardness increases $\Rightarrow \Psi$ *increases*. This result is of interest to researchers in information theory, since it would support the significance of Ψ as a quantification of contextual information and link problem solving search to information. This result could provide further insights for researchers studying distinctions between hard and easy NP Complete problem instances. In this study, evidence supports the hypothesis that in both random and bi-connected graphs problems of high difficulty tend to have a high information content as illustrated in Figure 3.3(a) and (b). In all runs, as hardness was monotonically increased, Ψ tended to increase near monotonically.

This study was preliminary in nature, but has identified additional areas of research that merit further study summarized by the following questions: Why do hard coloring problems in random and bi-connected graphs have high Ψ? Is there a relationship between hardness and Ψ in scale free graphs? Why does Ψ tend to reduce for scale free and small world graphs as the number of nodes increase?

REFERENCES

[1] N. S. Flann et al., "Kolmogorov complexity of epithelial pattern formation: The role of regulatory network configuration," *Biosystems*, vol. 112, pp. 131–138, May, 2013.

[2] Intel Corporation. (2012). Transistors to transformations: From sand to circuits - how intel makes chips. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/corporate-information/museum-transistors-to-transformations-brochure.pdf

[3] R. Soderbery. (2013, Jan). How many things are currently connected to the "internet of things" (iot)? [Online]. Available: http://www.forbes.com/sites/quora/2013/01/07/how-many-things-are-currently-connected-to-the-internet-of-things-iot/

[4] M. Ridley, *Genome; The Autobiography of a Species in 23 Chapters*, 1st ed., Harper Perennial, 1999.

[5] M. Wohlsen. (2013, Jun). The atronomical math behind ups' new tool to deliver packages faster, [Online]. Available: wired.com/business/2013/06/ups-astronomical-math

[6] O. Goldreich, *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, 2008.

[7] S. Arora and B. Barak, *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Macmillan Higher Education, 1979.

[9] M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*, 3rd ed. Springer, 2009.

[10] J. Dutkowski et al., “A gene ontology inferred from molecular networks,” *Nature biotechnology*, vol. 31, no. 1, pp. 38–45, 2013.

[11] N. Soranzo et al., “Comparing association network algorithms for reverse engineering of large-scale gene regulatory networks: synthetic versus real data,” *Bioinformatics*, vol. 23, no. 13, pp. 1640–1647, 2007.

[12] David J. Galas et al., “Set-based complexity and biological information,” *Bio-Information*, Jan 2008.

[13] David J. Galas et al., “Biological information as set-based complexity,” *IEEE Transactions on Information Theory*, vol. 56, no. 2, Feb 2010.

[14] Gregory W. Carter et al., “Maximal extraction of biological information from genetic interaction data,” *PLoS Computational Biology*, vol. 5, Apr 2009. [Online]. Available: www.ploscompbiol.org

[15] Gregory W. Carter et al., “A systems-biology approach to modular genetic complexity,” *Chaos*, 2010.

[16] C.E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[17] T. M. Cover and J. A. Thomas, “Entropy, Relative Entropy and Mutual Information,” in *Elements of Information Theory*. John Wiley and Sons, Inc., 1991, pp. 12–49.

[18] T. G. Lewis, “Small-World Networks,” in *Network Science*. John Wiley and Sons, Inc., 2009, pp. 131–175.

[19] T. G. Lewis, “Scale-Free Networks,” in *Network Science*. John Wiley and Sons, inc., 2009, pp. 177–215.

[20] G. Agnarsson and R. Greenlaw, *Graph Theory: Modeling, Applications, and Algorithms.* Pearson, 2006.

[21] S. Bornholdt and H. G. Schuster, *Handbook of Graphs and Networks: From the Genome to the Internet.* Wiley-VCH, Feb 2003.

[22] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[23] P. Brazhnik et al., "Gene networks: how to put the function in genomics," *TRENDS in Biotechnology*, vol. 20, no. 11, pp. 467–472, 2002.

APPENDICES

Appendix A

Source Code

This appendix contains the source code that was created to run the experiments. The code was written in C# using Microsoft Visual Studio 2008. Each section represents a single source file.

A.1 Main User Interface Screen (Form1.cs)

The main form allows the user to select a data directory where all output files will be written, and select which of the four graph types to run. The user then clicks the 'Continue' button to move to the operation management screen, or the 'Exit' button to end the program.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace GraphandPSISolver
{
    public partial class frmMain : Form
    {
        string myDirName;

        public frmMain()
        {
            InitializeComponent();
        }

```

```
        private void btnExit_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void btnDirectory_Click(object sender, EventArgs e)
        {
            string szCurDirectory;

            szCurDirectory = Directory.GetCurrentDirectory();
            if (!Directory.Exists(szCurDirectory))
            {
                Directory.CreateDirectory(szCurDirectory);
            }
            fbDirName.SelectedPath = szCurDirectory;
            DialogResult result = fbDirName.ShowDialog();
            if (result == DialogResult.OK)
            {
                myDirName = fbDirName.SelectedPath;
                txtDirName.Text = myDirName;
                btnContinue.Enabled = true;
            }
            else
            {
                myDirName = "";
                txtDirName.Text = myDirName;
                btnContinue.Enabled = false;
            }
        }

        private void btnContinue_Click(object sender, EventArgs e)
        {

            if (rbRandom.Checked == true)
            {
                frmGenerate myDialog = new frmGenerate();
                myDialog.DirectoryName = myDirName;
                myDialog.GraphType = "Random";
                myDialog.ShowDialog(this);
            }
            else if (rbScaleFree.Checked == true)
            {
                frmGenerate myDialog = new frmGenerate();
                myDialog.DirectoryName = myDirName;
                myDialog.GraphType = "ScaleFree";
                myDialog.ShowDialog(this);
            }
```

```
            else if (rbSmallWorld.Checked == true)
            {
                frmGenerate myDialog = new frmGenerate();
                myDialog.DirectoryName = myDirName;
                myDialog.GraphType = "SmallWorld";
                myDialog.ShowDialog(this);
            }
            else if (rbBiConnected.Checked == true)
            {
                frmGenerate myDialog = new frmGenerate();
                myDialog.DirectoryName = myDirName;
                myDialog.GraphType = "BiConnected";
                myDialog.ShowDialog(this);
            }
        }

        private void frmMain_Shown(object sender, EventArgs e)
        {
            rbRandom.Checked = true;
        }
    }
}
```

A.2 Operation Management Screen (Generate.cs)

The management screen allows the user to select all of the parameters that will control the experiments that are run. These include:

1. Number of Graphs per Trial - Specifies how many graphs will be created and solved for each combination of graph type, node count, and probability. The majority of our testing was done with a value of five, but a few trials used for comparisons in sections 3.2 and 3.3 used a value of ten.

2. Beginning Node Count - Specifies the smallest number of nodes used in the trials. Node count begins at this number and increases in increments of one until reaching the Ending Node Count. Our testing used a value of 10.

3. Ending Node Count - Specifies the largest number of nodes used in the trials. Our testing used a value of 21.

4. Beginning Probability - Specifies the smallest probability value used in the trials. Probability begins at this number and increases in increments of five until reaching the Ending Probability. Our testing used a value of five.

5. Ending Probability - Specifies the largest probability value used in the trials. Our testing used a value of 50.

6. Hill Climber Maximum Retries - Specifies the maximum number of consecutive tries to find a neighbor graph where the monotonic value is larger than the previous try. If a larger value is not found in this number of retries, the hill climber is done. Our testing used a value of 100.

7. Use Fixed Color Count - An entry that specifies a fixed color count that is used regardless of whether the problem can actually be solved or not. Our testing used this option with a fixed color count of three.

8. Vary Color Count - An entry that allows the user to specify a starting color count. If the problem can't be solved using that number, the value is incremented by one and the problem is rerun. This continues until the problem can be solved. Our testing did not use this option

In addition to the input parameters, the management screen displays current status so the user can see details about progress for the current graph.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.Threading;

```

```
namespace GraphandPSISolver
{
    public partial class frmGenerate : Form
    {
        //-----------------------------------------------------------------
        // Define constants.
        //-----------------------------------------------------------------
        // Task States
        const int TASK_NONE = 0;
        const int TASK_START = 1;
        const int TASK_BEGIN = 2;
        const int TASK_ACTIVE = 3;
        const int TASK_PAUSED = 4;
        const int TASK_CANCELLED = 5;
        const int TASK_DONE = 6;
        // Current Activity
        const int ACT_GENERATE_GRAPH = 0;
        const int ACT_SOLVE_GRAPH = 1;
        // Current Monotonic Value
        const int MON_HARDNESS = 0;
        const int MON_PSI = 1;
        //-----------------------------------------------------------------
        // Define data elements for the class.
        //-----------------------------------------------------------------
        // Values entered on the current screen
        private int nGraphsPerTrial = 0;
        private int nBegNodeCount = 0;
        private int nEndNodeCount = 0;
        private int nBegProbability = 0;
        private int nEndProbability = 0;
        private int nHCRetries = 0;
        private int nFixedColor = 0;
        private int nVaryColor = 0;
        // Values updated/used by the current screen
        private string SubDirName = string.Empty;
        private string FullPathName = string.Empty;
        private static grfFile wrkFile = new grfFile();
        private Thread WorkThread;
        // Values passed in from the previous screen
        private string DirName = string.Empty;
        private string GrfTyp = string.Empty;
        // Values for current results
        private int nCurNodeCount = 0;
        private int nCurProbability = 0;
        private int nCurGraphNumber = 0;
        private string CurGraphName = string.Empty;
        private int nCurActivity = ACT_GENERATE_GRAPH;
```

```
        private int nMonotonicVal = MON_HARDNESS;
        private long nHardnessVal = 0;
        private double fPSIVal = 0.0;
        private int nCurColorCount = 0;
        private int nCurRetryCount = 0;
        private int nCurDepth = 0;
        // Status values
        private int nTaskState = TASK_NONE;

        public frmGenerate()
        {
            InitializeComponent();
        }

        public string DirectoryName
        {
            get { return DirName; }
            set { DirName = value; }
        }

        public string GraphType
        {
            get { return GrfTyp; }
            set { GrfTyp = value; }
        }

        private void frmGenerate_Shown(object sender, EventArgs e)
        {
            this.Text = GrfTyp + " Graphs - Generate and Solve";
            txtSubDirName.Text = DirName;
            ClearInputFields();
            ClearResultFields();
            btnStart.Text = "Start";
            btnStart.Enabled = false;
            btnClose.Text = "Close";
            btnClose.Enabled = true;
            txtGraphsPerTrial.Focus();
        }

        private void txtGraphsPerTrial_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
        }

        private void txtBegNodeCnt_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
```

```
        }

        private void txtEndNodeCnt_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
        }

        private void txtBegProbability_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
        }

        private void txtEndProbability_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
        }

        private void txtHCRetries_TextChanged(object sender, EventArgs e)
        {
            SetStartButtonState();
        }

        private void SetStartButtonState()
        {

            if ((txtGraphsPerTrial.Text.Length > 0) &&
                (txtBegNodeCnt.Text.Length > 0) &&
                (txtEndNodeCnt.Text.Length > 0) &&
                (txtBegProbability.Text.Length > 0) &&
                (txtEndProbability.Text.Length > 0) &&
                (txtHCRetries.Text.Length > 0))
            {
                btnStart.Enabled = true;
            }
            else
            {
                btnStart.Enabled = false;
            }
        }

        private void btnStart_Click(object sender, EventArgs e)
        {
            int nRtnval = 0;

            switch (nTaskState)
            {
                case TASK_NONE:        // Start button pressed
```

```
                    nRtnval = ValidateInputFields();
                    if (nRtnval == 0)
                    {
                        DisableInputFields();
                        btnStart.Text = "Pause";
                        btnStart.Enabled = false;
                        btnClose.Text = "Cancel";
                        wrkFile.DoneFlag = 0;
                        nTaskState = TASK_START;
                        timer1.Enabled = true;
                    }
                    break;
                case TASK_START:        // Start button is disabled in these
                    states
                case TASK_BEGIN:
                    break;
                case TASK_ACTIVE:       // Pause button pressed
                    wrkFile.PauseFlag = 1;
                    btnStart.Text = "Continue";
                    nTaskState = TASK_PAUSED;
                    break;
                case TASK_PAUSED:       // Continue button pressed
                    wrkFile.PauseFlag = 0;
                    btnStart.Text = "Pause";
                    nTaskState = TASK_ACTIVE;
                    break;
                case TASK_CANCELLED:    // Start button is disabled in these
                    states
                case TASK_DONE:
                    break;
                default:
                    break;
            }
        }

        private void btnClose_Click(object sender, EventArgs e)
        {
            switch (nTaskState)
            {
                case TASK_NONE:         // Close button pressed
                    Close();
                    break;
                case TASK_START:        // Cancel button pressed
                case TASK_BEGIN:
                case TASK_ACTIVE:
                case TASK_PAUSED:
                    btnStart.Enabled = false;
```

```
                    btnClose.Enabled = false;
                    nTaskState = TASK_CANCELLED;
                    break;
                case TASK_CANCELLED:   // Cancel button is disabled in these states
                case TASK_DONE:
                    break;
                default:
                    break;
            }
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            string CalcPathName = string.Empty;

            //-----------------------------------------------------------------
            // Each instance when the timer fires we want to check the state
            // of the problem being generated or solved and make any updates
            // required.
            //-----------------------------------------------------------------
            switch (nTaskState)
            {
                case TASK_NONE:        // No solution in progress
                    timer1.Enabled = false;
                    break;
                case TASK_START:
                    btnStart.Enabled = false;
                    ClearResultFields();
                    nCurNodeCount = nBegNodeCount;
                    nCurProbability = nBegProbability;
                    nCurGraphNumber = 1;
                    DisplayResults();
                    nTaskState = TASK_BEGIN;
                    break;
                case TASK_BEGIN:
                    // Clear the result fields and initialize local variables
                    CheckDirectory();
                    txtWrkDirName.Text = SubDirName;
                    FullPathName = DirectoryName + "\\" + SubDirName;
                    // Set initial values for the worker thread
                    wrkFile.DirectoryName = FullPathName;
                    wrkFile.FileName = "grf" + nCurGraphNumber.ToString("D4");
                    wrkFile.GraphType = GrfTyp;
                    wrkFile.RetryCount = nHCRetries;
                    wrkFile.NodeCount = nCurNodeCount;
                    wrkFile.Probability = nCurProbability;
```

```
                    wrkFile.CurrentActivity = 0;
                    wrkFile.MonotonicValue = MON_HARDNESS;
                    wrkFile.ConflictCount = 0;
                    wrkFile.PSIvalue = 0;
                    wrkFile.CurrentRetries = 0;
                    wrkFile.UseFixedColorCount = rbFixedCount.Checked;
                    if (rbFixedCount.Checked == true)
                    {
                        wrkFile.MinimumColorCount = nFixedColor;
                    }
                    else
                    {
                        wrkFile.MinimumColorCount = nVaryColor;
                    }
                    wrkFile.StatusMessage = string.Empty;
                    // Display the initial values
                    DisplayResults();
                    //Start the work thread
                    wrkFile.DoneFlag = 0;
                    wrkFile.PauseFlag = 0;
                    WorkThread = new Thread(new ThreadStart(wrkFile.
                        CreateAndSolveGraph));
                    WorkThread.Name = "WorkThread";
                    WorkThread.Start();
                    btnStart.Enabled = true;
                    nTaskState = TASK_ACTIVE;
                    break;
                case TASK_ACTIVE:
                    if ((wrkFile.DoneFlag != 0) || (WorkThread.IsAlive != true))
                    {
                        nCurGraphNumber++;
                        if (nCurGraphNumber > nGraphsPerTrial)
                        {
                            CalcPathName = DirectoryName + "\\" + GraphType + "\\
                                " +
                                               nCurNodeCount.ToString("D4") + "\\" +
                                               nCurProbability.ToString("D3");
                            wrkFile.CalculateLeafResults(CalcPathName, GraphType,
                                                                nCurNodeCount,
                                                                    nCurProbability);
                            nCurGraphNumber = 1;
                            nCurProbability += 5;
                            if (nCurProbability > nEndProbability)
                            {
                                CalcPathName = DirectoryName + "\\" + GraphType +
                                     "\\" +
                                                   nCurNodeCount.ToString("D4");
```

```
                            wrkFile.CalculateProbabilityResults(CalcPathName, GraphType,
                                                                nCurNodeCount);
                            nCurProbability = nBegProbability;
                            nCurNodeCount++;
                            if (nCurNodeCount > nEndNodeCount)
                            {
                                CalcPathName = DirectoryName + "\\" + GraphType;
                                wrkFile.CalculateNodeCountResults(CalcPathName, GraphType);
                                nTaskState = TASK_DONE;
                            }
                            else
                            {
                                nTaskState = TASK_BEGIN;
                            }
                        }
                        else
                        {
                            nTaskState = TASK_BEGIN;
                        }
                    }
                    else
                    {
                        nTaskState = TASK_BEGIN;
                    }
                }
                DisplayResults();
                break;
            case TASK_PAUSED:
                break;
            case TASK_CANCELLED:
            case TASK_DONE:
                EnableInputFields();
                wrkFile.DoneFlag = 1;
                wrkFile.PauseFlag = 0;
                btnStart.Text = "Start";
                btnStart.Enabled = true;
                btnClose.Text = "Close";
                btnClose.Enabled = true;
                nTaskState = TASK_NONE;
                MessageBox.Show("The current run is complete");
                break;
            default:
                break;
```

```
            }
        }

        private void CheckDirectory()
        {

            //------------------------------------------------------------------
            // Get the directory name at the 'type' level and make sure it
            // exists.  If it doesn't, create it.
            //------------------------------------------------------------------
            SubDirName = GraphType;
            FullPathName = DirectoryName + "\\" + SubDirName;
            if (!Directory.Exists(FullPathName))
            {
                Directory.CreateDirectory(FullPathName);
            }
            //------------------------------------------------------------------
            // Get the directory name at the 'node count' level and make sure
            // it exists.  If it doesn't, create it.
            //------------------------------------------------------------------
            SubDirName = SubDirName + "\\" + nCurNodeCount.ToString("D4");
            FullPathName = DirectoryName + "\\" + SubDirName;
            if (!Directory.Exists(FullPathName))
            {
                Directory.CreateDirectory(FullPathName);
            }
            //------------------------------------------------------------------
            // Get the directory name at the 'probability' level and make sure
            // it exists.  If it doesn't, create it.
            //------------------------------------------------------------------
            SubDirName = SubDirName + "\\" + nCurProbability.ToString("D3");
            FullPathName = DirectoryName + "\\" + SubDirName;
            if (!Directory.Exists(FullPathName))
            {
                Directory.CreateDirectory(FullPathName);
            }
        }

        private void DisplayResults()
        {
            // Read the current results
            CurGraphName = wrkFile.FileName;
            nCurActivity = wrkFile.CurrentActivity;
            nMonotonicVal = wrkFile.MonotonicValue;
            nCurColorCount = wrkFile.CurrentColorCount;
            nCurRetryCount = wrkFile.CurrentRetries;
            nCurDepth = wrkFile.CurrentDepth;
```

```
            nHardnessVal = wrkFile.MaxColorConflict;
            fPSIVal = wrkFile.MaxPSIvalue;
            // Display the results
            txtCurNodeCnt.Text = nCurNodeCount.ToString();
            txtCurProbability.Text = nCurProbability.ToString();
            txtCurGraphNumber.Text = nCurGraphNumber.ToString();
            txtCurGraphName.Text = CurGraphName;
            if (nCurActivity == ACT_GENERATE_GRAPH)
            {
                rbGenerateGraph.Checked = true;
            }
            else
            {
                rbSolveGraph.Checked = true;
            }
            if (nMonotonicVal == MON_HARDNESS)
            {
                rbHardness.Checked = true;
            }
            else
            {
                rbPSI.Checked = true;
            }
            txtHardnessVal.Text = nHardnessVal.ToString();
            txtPSIVal.Text = fPSIVal.ToString();
            txtCurColorCnt.Text = nCurColorCount.ToString();
            txtCurRetryCnt.Text = nCurRetryCount.ToString();
            txtCurDepth.Text = nCurDepth.ToString();
            txtMessage.Text = wrkFile.StatusMessage;
        }

        private void ClearResultFields()
        {
            nCurActivity = ACT_GENERATE_GRAPH;
            rbGenerateGraph.Checked = true;
            nMonotonicVal = MON_HARDNESS;
            rbHardness.Checked = true;
            nHardnessVal = 0;
            txtHardnessVal.Text = "";
            fPSIVal = 0.0;
            txtPSIVal.Text = "";
            nCurColorCount = 0;
            txtCurColorCnt.Text = "";
            nCurRetryCount = 0;
            txtCurRetryCnt.Text = "";
            nCurDepth = 0;
            txtCurDepth.Text = "";
```

```
        }

        private void ClearInputFields()
        {
            txtGraphsPerTrial.Text = "";
            txtBegNodeCnt.Text = "";
            txtEndNodeCnt.Text = "";
            txtBegProbability.Text = "";
            txtEndProbability.Text = "";
            txtHCRetries.Text = "";
            rbFixedCount.Checked = true;
            txtFixedCount.Text = "3";
            txtVaryCount.Text = "2";
        }

        private void EnableInputFields()
        {
            txtGraphsPerTrial.Enabled = true;
            txtBegNodeCnt.Enabled = true;
            txtEndNodeCnt.Enabled = true;
            txtBegProbability.Enabled = true;
            txtEndProbability.Enabled = true;
            txtHCRetries.Enabled = true;
            rbFixedCount.Enabled = true;
            rbVaryCount.Enabled = true;
            if (rbFixedCount.Checked == true)
            {
                txtFixedCount.Enabled = true;
            }
            if (rbVaryCount.Checked == true)
            {
                txtVaryCount.Enabled = true;
            }
        }

        private void DisableInputFields()
        {
            txtGraphsPerTrial.Enabled = false;
            txtBegNodeCnt.Enabled = false;
            txtEndNodeCnt.Enabled = false;
            txtBegProbability.Enabled = false;
            txtEndProbability.Enabled = false;
            txtHCRetries.Enabled = false;
            rbFixedCount.Enabled = false;
            rbVaryCount.Enabled = false;
            txtFixedCount.Enabled = false;
            txtVaryCount.Enabled = false;
```

```
        }

        private int ValidateInputFields()
        {
            int nRtnval = 0;

            //------------------------------------------------------------------
            // Limit the number of graphs per trial to 100.
            //------------------------------------------------------------------
            if (nRtnval == 0)
            {
                nGraphsPerTrial = Convert.ToInt32(txtGraphsPerTrial.Text);
                if (nGraphsPerTrial < 1)
                {
                    MessageBox.Show("The number of graphs must be greater than 0");
                    txtGraphsPerTrial.Focus();
                    nRtnval = -1;
                }
                else if (nGraphsPerTrial > 100)
                {
                    MessageBox.Show("The maximum number of graphs allowed is 100");
                    txtGraphsPerTrial.Focus();
                    nRtnval = -1;
                }
            }
            //----------------------------------------------------------
            // Validate the node count.  We are going to limit it to
            // 1000.
            //----------------------------------------------------------
            if (nRtnval == 0)
            {
                nBegNodeCount = Convert.ToInt32(txtBegNodeCnt.Text);
                nEndNodeCount = Convert.ToInt32(txtEndNodeCnt.Text);
                if (nBegNodeCount < 10)
                {
                    MessageBox.Show("The beginning node count must be >= 10");
                    txtBegNodeCnt.Focus();
                    nRtnval = -1;
                }
                else if (nEndNodeCount > 1000)
                {
                    MessageBox.Show("The ending node count must be <= 1000");
                    txtEndNodeCnt.Focus();
                    nRtnval = -1;
                }
```

```
            else if (nBegNodeCount > nEndNodeCount)
            {
                MessageBox.Show("The ending node count must be >= to the beginning node count");
                txtEndNodeCnt.Focus();
                nRtnval = -1;
            }
        }
        //-----------------------------------------------------------
        // Validate the probability.  It must be in the range from
        // 2 to 50.
        //-----------------------------------------------------------
        if (nRtnval == 0)
        {
            nBegProbability = Convert.ToInt32(txtBegProbability.Text);
            nEndProbability = Convert.ToInt32(txtEndProbability.Text);
            if (nBegProbability < 2)
            {
                MessageBox.Show("The beginning probability must be >= 2");
                txtBegProbability.Focus();
                nRtnval = -1;
            }
            else if (nEndProbability > 50)
            {
                MessageBox.Show("The ending probability must be <= 50");
                txtEndProbability.Focus();
                nRtnval = -1;
            }
            else if (nBegProbability > nEndProbability)
            {
                MessageBox.Show("The ending probability must be >= to the beginning probability");
                txtEndProbability.Focus();
                nRtnval = -1;
            }
        }
        //-----------------------------------------------------------
        // Validate the Hill Climber Retry count.  It must be in
        // the range from 10 to 10000.
        //-----------------------------------------------------------
        if (nRtnval == 0)
        {
            nHCRetries = Convert.ToInt32(txtHCRetries.Text);
            if (nHCRetries < 10)
            {
                MessageBox.Show("The number of retries must be at least 10");
                txtHCRetries.Focus();
```

```
                    nRtnval = -1;
                }
                else if (nHCRetries > 10000)
                {
                    MessageBox.Show("The maximum number of retries allowed is 10,000");
                    txtHCRetries.Focus();
                    nRtnval = -1;
                }
            }
            //--------------------------------------------------------------
            // Validate the fixed color count.
            //--------------------------------------------------------------
            if (nRtnval == 0)
            {
                nFixedColor = Convert.ToInt32(txtFixedCount.Text);
                if (nFixedColor < 2)
                {
                    MessageBox.Show("The number of fixed colors must be at least 2");
                    txtFixedCount.Focus();
                    nRtnval = -1;
                }
                else if (nFixedColor > 100)
                {
                    MessageBox.Show("The maximum number of fixed colors allowed is 100");
                    txtFixedCount.Focus();
                    nRtnval = -1;
                }
            }
            //--------------------------------------------------------------
            // Validate the variable color count.
            //--------------------------------------------------------------
            if (nRtnval == 0)
            {
                nVaryColor = Convert.ToInt32(txtVaryCount.Text);
                if (nVaryColor < 2)
                {
                    MessageBox.Show("The number of variable colors must be at least 2");
                    txtVaryCount.Focus();
                    nRtnval = -1;
                }
                else if (nVaryColor > 100)
                {
```

```
                    MessageBox.Show("The maximum number of variable colors
                        allowed is 100");
                    txtVaryCount.Focus();
                    nRtnval = -1;
                }
            }
            return nRtnval;
        }
    }
}
```

A.3 Main Solver and Algorithm Implementation (grfFile.cs)

This is by far the largest file and the code that does most of the actual problem solving. This includes the implementation of the algorithms described in section 2 of this report. It also includes the code that implements the algorithm used to calculate the value for PSI.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Diagnostics;
using System.Threading;

namespace GraphandPSISolver
{
    class grfFile
    {
        //------------------------------------------------------------------
        // Define constants.
        //------------------------------------------------------------------
        // Graph types
        const int RANDOM_GRAPH = 1;
        const int SCALE_FREE_GRAPH = 2;
        const int SMALL_WORLD_GRAPH = 3;
        const int BI_CONNECTED_GRAPH = 4;
        // Misc defines
        const int MAX_NODES = 1000;
        const int REPLACE_EDGE = 1;
        const int MAX_NEIGHBOR_RETRY = 1000;
        // Current Activity
        const int ACT_GENERATE_GRAPH = 0;
```

```
        const int ACT_SOLVE_GRAPH = 1;
        // Monotonic values
        const int MON_HARDNESS = 0;
        const int MON_PSI = 1;
        //-------------------------------------------------------------------
        // Define data elements for the class.
        //-------------------------------------------------------------------
        private int nDoneFlag = 0;
        private int nPauseFlag = 0;
        private int nNodes = 0;
        private int nProbability = 0;
        private int nStep = 0;
        private int nRetries = 0;
        private int nRetryCount = 0;
        private int nActivity = 0;
        private int nMonotonic = MON_HARDNESS;
        private int nGraphType = 0;
        private int nMinColorCount = 0;
        private int nCurDepth = 0;
        private int nCurColorCount = 0;
        private bool bColorIsSolvable = false;
        private bool bFixedColorCount = true;
        private long nConflictCount = 0;
        private long nMaxColorConflict = 0;
        private double fMaxPSIvalue = 0;
        private double PSIcalculatedValue = 0.0;
        private double fHardnessTime = 0.0;
        private double fPSITime = 0.0;
        private string myDirName = string.Empty;
        private string myPrefix = string.Empty;
        private string myFileName = string.Empty;
        private string myActivity = string.Empty;
        private string myGraphType = string.Empty;
        private string myStatus = string.Empty;
        private short[,] nodeArray = new short[MAX_NODES, MAX_NODES];
        private IList<NodeElement> nodeList = new List<NodeElement>();
        private IList<HillClimbResult> myHCResults = new List<HillClimbResult>();
        private IList<RestorePoint> restoreList = new List<RestorePoint>();
        private IList<ResultsData> resultsData = new List<ResultsData>();
        private AveragesData averagesData = new AveragesData();
        private Random myRand = new Random((int)DateTime.Now.Ticks);
        private Stopwatch sw = new Stopwatch();
        static StreamWriter fileWriter;
        static FileStream output;
        //-------------------------------------------------------------------
        // Define methods for the class.
        //-------------------------------------------------------------------
```

```
        //******************************************************************
        // Constructor
        //******************************************************************
        public grfFile()
        {
        }

        //******************************************************************
        // Get and Set methods.
        //******************************************************************
        public int DoneFlag
        {
            get { return nDoneFlag; }
            set { nDoneFlag = value; }
        }

        public int PauseFlag
        {
            get { return nPauseFlag; }
            set { nPauseFlag = value; }
        }

        public string DirectoryName
        {
            get { return myDirName; }
            set { myDirName = value; }
        }

        public string FileName
        {
            get { return myFileName; }
            set { myFileName = value; }
        }

        public string GraphType
        {
            get { return myGraphType; }
            set { myGraphType = value; }
        }

        public int NodeCount
        {
            get { return nNodes; }
            set { nNodes = value; }
        }

        public int Probability
```

```
        {
            get { return nProbability; }
            set { nProbability = value; }
        }

        public string PrefixName
        {
            get { return myPrefix; }
            set { myPrefix = value; }
        }

        public double PSIvalue
        {
            get { return PSIcalculatedValue; }
            set { PSIcalculatedValue = value; }
        }

        public int PSIstep
        {
            get { return nStep; }
            set { nStep = value; }
        }

        public bool ColorSolvable
        {
            get { return bColorIsSolvable; }
            set { bColorIsSolvable = value; }
        }

        public long ConflictCount
        {
            get { return nConflictCount; }
            set { nConflictCount = value; }
        }

        public double MaxPSIvalue
        {
            get { return fMaxPSIvalue; }
            set { fMaxPSIvalue = value; }
        }

        public long MaxColorConflict
        {
            get { return nMaxColorConflict; }
            set { nMaxColorConflict = value; }
        }

```

```
        public int RetryCount
        {
            get { return nRetries; }
            set { nRetries = value; }
        }

        public int CurrentRetries
        {
            get { return nRetryCount; }
            set { nRetryCount = value; }
        }

        public int CurrentActivity
        {
            get { return nActivity; }
            set { nActivity = value; }
        }

        public int MonotonicValue
        {
            get { return nMonotonic; }
            set { nMonotonic = value; }
        }

        public bool UseFixedColorCount
        {
            get { return bFixedColorCount; }
            set { bFixedColorCount = value; }
        }

        public int MinimumColorCount
        {
            get { return nMinColorCount; }
            set { nMinColorCount = value; }
        }

        public string StatusMessage
        {
            get { return myStatus; }
            set { myStatus = value; }
        }

        public int CurrentColorCount
        {
            get { return nCurColorCount; }
            set { nCurColorCount = value; }
        }
```

```
        public int CurrentDepth
        {
            get { return nCurDepth; }
            set { nCurDepth = value; }
        }

        //*******************************************************************
        // Main worker thread.
        //*******************************************************************
        public void CreateAndSolveGraph()
        {
            int nRtnval = 0;
            int i;
            long nCurColorConflict = 0;
            double fCurPSIvalue = 0.0;

            CurrentActivity = ACT_GENERATE_GRAPH;
            //---------------------------------------------------------------
            // Determine the type of graph we are working with.
            //---------------------------------------------------------------
            if (nRtnval == 0)
            {
                myStatus = "Determine the graph type";
                if (myGraphType.Equals("Random"))
                {
                    nGraphType = RANDOM_GRAPH;
                }
                else if (myGraphType.Equals("ScaleFree"))
                {
                    nGraphType = SCALE_FREE_GRAPH;
                }
                else if (myGraphType.Equals("SmallWorld"))
                {
                    nGraphType = SMALL_WORLD_GRAPH;
                }
                else if (myGraphType.Equals("BiConnected"))
                {
                    nGraphType = BI_CONNECTED_GRAPH;
                }
                else
                {
                    myStatus = "Unrecognized graph type = " + myGraphType;
                    nRtnval = -1;
                }
            }

```

```
            //------------------------------------------------------------
            // Clear the node list so we start out empty.  Add the correct
            // number of nodes to our list.  Each node will be initialized
            // with an ID (which is really the offset) and node color of
            // 0, which means we haven't colored it yet.
            //------------------------------------------------------------
            if (nRtnval == 0)
            {
                myStatus = "Create and Initialize (" + nNodes.ToString() + ")
                    nodes";
                foreach (NodeElement ne in nodeList)
                {
                    ne.ClearEdgeList();
                }
                nodeList.Clear();
                for (i = 0; i < nNodes; i++)
                {
                    nodeList.Add(new NodeElement(i, 0));
                }
            }
            //------------------------------------------------------------
            // Generate a new graph of the specified type.  Save the new
            // graph to a file.
            //------------------------------------------------------------
            if (nRtnval == 0)
            {
                myStatus = "Generate the new graph and save to a file";
                switch (nGraphType)
                {
                    case SMALL_WORLD_GRAPH:
                        CreateSmallWorldGraph();
                        break;
                    case SCALE_FREE_GRAPH:
                        CreateScaleFreeGraph();
                        break;
                    case BI_CONNECTED_GRAPH:
                        CreateBiConnectedGraph();
                        break;
                    case RANDOM_GRAPH:
                        CreateRandomGraph();
                        break;
                    default:
                        nGraphType = RANDOM_GRAPH;
                        CreateRandomGraph();
                        break;
                }
                SaveGraphToFile();
```

```
}

//------------------------------------------------------------
// Initialize our working variables.  Enter a loop where we
// try to maximize the hardness value by generating neighbors
// to the graph and calculating for hardness.  Continue in the
// loop until we exceed a retry limit without finding any
// improvement in the hardness value.
//------------------------------------------------------------
nMonotonic = MON_HARDNESS;
myHCResults.Clear();                      // Clear any previous results
nRetryCount = 0;
nMaxColorConflict = CalculateHardness();  // Initial hardness value
fMaxPSIvalue = CalculatePSI();            // Initial PSI value
myHCResults.Add(new HillClimbResult(nCurColorCount, bColorIsSolvable, nMaxColorConflict,
                                    fHardnessTime, fMaxPSIvalue, fPSITime));
CurrentActivity = ACT_SOLVE_GRAPH;
//------------------------------------------------------------
// With our initial values set, we are ready to go into a loop
// where we try to first maximize the hardness value, and next
// try to maximize the PSI value.  As we find higher values we
// will save the results in the hill climber results list.
// In this first loop hardness is the monotonic value.
//------------------------------------------------------------
while ((nRetryCount < nRetries) && (nDoneFlag == 0) && (nRtnval == 0))
{
    if (nPauseFlag != 0)
    {
        Thread.Sleep(100);
    }
    else
    {
        //------------------------------------------------------
        // Find a random neighbor.
        //------------------------------------------------------
        myStatus = "Find a random neighbor (hardness is monotonic)";
        switch (nGraphType)
        {
            case SMALL_WORLD_GRAPH:
                GenerateSmallWorldNeighbor();
                break;
            case SCALE_FREE_GRAPH:
                GenerateScaleFreeNeighbor();
```

```
                    break;
                case BI_CONNECTED_GRAPH:
                    GenerateBiConnectedNeighbor();
                    break;
                case RANDOM_GRAPH:
                default:
                    GenerateRandomNeighbor();
                    break;
            }
            //-------------------------------------------------------
            // Solve for hardness.  If the hardness value is
            // larger than the current max, this one becomes our
            // new max.  Otherwise, undo the last neighbor change
            // and increment the retry value.
            //-------------------------------------------------------
            myStatus = "Solve for hardness (hardness is monotonic)";
            nCurColorConflict = CalculateHardness();
            if (nCurColorConflict > nMaxColorConflict)
            {
                myStatus = "Solve for PSI (hardness is monotonic)";
                fCurPSIvalue = CalculatePSI();
                nMaxColorConflict = nCurColorConflict;
                fMaxPSIvalue = fCurPSIvalue;
                myHCResults.Add(new HillClimbResult(nCurColorCount,
                    bColorIsSolvable, nMaxColorConflict,
                                                        fHardnessTime,
                                                            fMaxPSIvalue,
                                                            fPSITime));
                nRetryCount = 0;
            }
            else
            {
                RestoreNeighbor();
                nRetryCount++;
            }
        }
    }
    SaveHCResults(MON_HARDNESS);
    //----------------------------------------------------------------
    // Re-load the original graph and do the second loop where
    // PSI is the monotonic value.
    //----------------------------------------------------------------
    nMonotonic = MON_PSI;
    LoadGraphFromFile();
    myHCResults.Clear();                        // Clear any previous
        results
    nRetryCount = 0;
```

```
            nMaxColorConflict = CalculateHardness();  // Initial hardness value
            fMaxPSIvalue = CalculatePSI();             // Initial PSI value
            myHCResults.Add(new HillClimbResult(nCurColorCount, bColorIsSolvable, nMaxColorConflict,
                                                    fHardnessTime, fMaxPSIvalue, fPSITime));
            while ((nRetryCount < nRetries) && (nDoneFlag == 0) && (nRtnval == 0))
            {
                if (nPauseFlag != 0)
                {
                    Thread.Sleep(100);
                }
                else
                {
                    //-----------------------------------------------------
                    // Find a random neighbor.
                    //-----------------------------------------------------
                    myStatus = "Find a random neighbor (PSI is monotonic)";
                    switch (nGraphType)
                    {
                        case SMALL_WORLD_GRAPH:
                            GenerateSmallWorldNeighbor();
                            break;
                        case SCALE_FREE_GRAPH:
                            GenerateScaleFreeNeighbor();
                            break;
                        case BI_CONNECTED_GRAPH:
                            GenerateBiConnectedNeighbor();
                            break;
                        case RANDOM_GRAPH:
                        default:
                            GenerateRandomNeighbor();
                            break;
                    }
                    //-----------------------------------------------------
                    // Solve for PSI.  If the PSI value is larger than the
                    // current max, this one becomes our new max.
                    // Otherwise, undo the last neighbor change and
                    // increment the retry value.
                    //-----------------------------------------------------
                    myStatus = "Solve for PSI (PSI is monotonic)";
                    fCurPSIvalue = CalculatePSI();
                    if (fCurPSIvalue > fMaxPSIvalue)
                    {
                        myStatus = "Solve for hardness (PSI is monotonic)";
                        nCurColorConflict = CalculateHardness();
```

```
                        fMaxPSIvalue = fCurPSIvalue;
                        nMaxColorConflict = nCurColorConflict;
                        myHCResults.Add(new HillClimbResult(nCurColorCount,
                            bColorIsSolvable, nMaxColorConflict,
                                                                    fHardnessTime,
                                                                        fMaxPSIvalue,
                                                                        fPSITime));
                        nRetryCount = 0;
                    }
                    else
                    {
                        RestoreNeighbor();
                        nRetryCount++;
                    }
                }
            }
            SaveHCResults(MON_PSI);
        }

        public long CalculateHardness()
        {
            long nHardness = 0;
            bool bDone = false;

            //------------------------------------------------------------
            // Solve for color based on current settings.  The end result
            // should be the hardness value (which is really the conflict
            // count), whether the problem was solvable or not, and how
            // many ticks it took to solve.  We ignore the pause flag at
            // this level because it invalidates our timing.
            //------------------------------------------------------------
            bDone = false;
            nCurColorCount = nMinColorCount;
            sw.Reset();
            sw.Start();
            while ((bDone == false) && (nDoneFlag == 0))
            {
                SolveForColor(nCurColorCount);
                if (bColorIsSolvable == true)
                {
                    bDone = true;
                }
                else
                {
                    if (bFixedColorCount == false)
                    {
                        nCurColorCount++;
```

```
                }
                else
                {
                    bDone = true;
                }
            }
        }
        nHardness = nConflictCount;
        sw.Stop();
        fHardnessTime = sw.Elapsed.TotalMilliseconds;
        return (nHardness);
    }

    public double CalculatePSI()
    {
        double fPSI = 0.0;

        //---------------------------------------------------------------
        // Convert the node list into an array, then solve for PSI.
        // Keep track of how many ticks it took to solve.  We ignore
        // the pause flag at this level because it invalidates our
        // timing.
        //---------------------------------------------------------------
        CopyGraphToArray();
        sw.Reset();
        sw.Start();
        SolveForPSI();
        fPSI = PSIcalculatedValue;
        sw.Stop();
        fPSITime = sw.Elapsed.TotalMilliseconds;
        return (fPSI);
    }

    //*******************************************************************
    // Graph Creation Methods.
    //*******************************************************************
    public void CreateRandomGraph()
    {
        int i;
        int j;
        int myProbability;

        //---------------------------------------------------------------
        // Loop through every combination of 2 nodes in the graph.  For
        // each combination determine if they should be connected based
        // on the current probability.  If yes, connect the nodes.
        //---------------------------------------------------------------
```

```
            for (i = 0; i < (this.NodeCount - 1); i++)
            {
                for (j = i + 1; j < this.NodeCount; j++)
                {
                    if (nodeList[i].IsConnected(j) == false)
                    {
                        myProbability = myRand.Next(0, 101);
                        if (myProbability <= this.Probability)
                        {
                            nodeList[i].AddEdge(j);
                            nodeList[j].AddEdge(i);
                        }
                    }
                }
            }
            //-------------------------------------------------------------
            // For each node except the last one, make sure there is some
            // connection between that node and the next one in the list.
            // If there is no connection, go ahead and add an edge to
            // connect them.
            //-------------------------------------------------------------
            for (i = 0; i < (this.NodeCount - 1); i++)
            {
                if (AreTwoNodesConnected(i, i + 1) == false)
                {
                    nodeList[i].AddEdge(i + 1);
                    nodeList[i + 1].AddEdge(i);
                }
            }
        }

        public void CreateSmallWorldGraph()
        {
            int i;
            int j;
            int nNextNode;
            int myProbability;

            //-------------------------------------------------------------
            // Start by creating a 2-regular graph.  This means each node
            // will be connected to the nodes that are one and two
            // sequential positions away from it.
            //-------------------------------------------------------------
            for (i = 0; i < this.NodeCount; i++)
            {
                nNextNode = FindPrimaryNode(i);
                if (nodeList[i].IsConnected(nNextNode) == false)
```

```
                {
                    nodeList[i].AddEdge(nNextNode);
                    nodeList[nNextNode].AddEdge(i);
                }
                nNextNode = FindSecondaryNode(i);
                if (nodeList[i].IsConnected(nNextNode) == false)
                {
                    nodeList[i].AddEdge(nNextNode);
                    nodeList[nNextNode].AddEdge(i);
                }
            }
            //---------------------------------------------------------------
            // Next we will add some other random edges to the graph.  We
            // do this by looping through every combination of 2 nodes and
            // determine if they should be connected based on the current
            // probability.
            //---------------------------------------------------------------
            for (i = 0; i < (this.NodeCount - 1); i++)
            {
                for (j = i + 1; j < this.NodeCount; j++)
                {
                    if (nodeList[i].IsConnected(j) == false)
                    {
                        myProbability = myRand.Next(0, 101);
                        if (myProbability <= this.Probability)
                        {
                            nodeList[i].AddEdge(j);
                            nodeList[j].AddEdge(i);
                        }
                    }
                }
            }
        }

        public void CreateScaleFreeGraph()
        {
            int i;

            //---------------------------------------------------------------
            // Create a complete graph with the first 3 nodes.
            //---------------------------------------------------------------
            nodeList[0].AddEdge(1);
            nodeList[0].AddEdge(2);
            nodeList[1].AddEdge(0);
            nodeList[1].AddEdge(2);
            nodeList[2].AddEdge(0);
            nodeList[2].AddEdge(1);
```

```
        //------------------------------------------------------------
        // For each of the remaining nodes, we will add edges based
        // on a probability calculation that favors those existing
        // nodes with the highest degree count.
        //------------------------------------------------------------
        for (i = 3; i < this.NodeCount; i++)
        {
            AddScaleFreeEdges(i, i - 1);
        }
    }

    public void CreateBiConnectedGraph()
    {
        int i;
        int j;
        int nNextNode;
        int myProbability;

        //------------------------------------------------------------
        // Start by creating a ring network.  This will verify that
        // every node is connected and, in a ring network, if any edge
        // is removed all nodes are connected.  This ensures that the
        // graph is bi-connected.
        //------------------------------------------------------------
        for (i = 0; i < this.NodeCount; i++)
        {
            nNextNode = FindPrimaryNode(i);
            if (nodeList[i].IsConnected(nNextNode) == false)
            {
                nodeList[i].AddEdge(nNextNode);
                nodeList[nNextNode].AddEdge(i);
            }
        }
        //------------------------------------------------------------
        // We now add a few more random edges to make the graph more
        // interesting.  Loop through every combination of 2 nodes in
        // the graph.  For each combination determine if they should
        // be connected based on the current probability.  If yes,
        // connect the nodes.
        //------------------------------------------------------------
        for (i = 0; i < (this.NodeCount - 1); i++)
        {
            for (j = i + 1; j < this.NodeCount; j++)
            {
                if (nodeList[i].IsConnected(j) == false)
                {
                    myProbability = myRand.Next(0, 101);
```

```
                if (myProbability <= this.Probability)
                {
                    nodeList[i].AddEdge(j);
                    nodeList[j].AddEdge(i);
                }
            }
        }
    }
}

//*******************************************************************
// Graph Validation Methods.
//*******************************************************************
public bool ValidateRandomGraph()
{
    bool bRtnval = false;

    //-------------------------------------------------------------
    // Our only validation on a random graph is to make sure all
    // of the nodes are still connected.
    //-------------------------------------------------------------
    bRtnval = AreAllNodesConnected();
    return (bRtnval);
}

public bool ValidateSmallWorldGraph()
{
    bool bRtnval = false;

    //-------------------------------------------------------------
    // Our only validation on a small world graph is to make sure
    // all of the nodes are still connected.
    //-------------------------------------------------------------
    bRtnval = AreAllNodesConnected();
    return (bRtnval);
}

public bool ValidateScaleFreeGraph()
{
    bool bRtnval = false;

    //-------------------------------------------------------------
    // Our only validation on a small world graph is to make sure
    // all of the nodes are still connected.
    //-------------------------------------------------------------
    bRtnval = AreAllNodesConnected();
    return (bRtnval);
```

```
        }

        public bool ValidateBiConnectedGraph()
        {
            int i;
            int j;
            int nCurNode;
            int nNextNode;
            bool bRtnval = true;

            //-------------------------------------------------------------
            // To validate our bi-connected graph, we individually remove
            // every edge in the graph and then verify that the graph is
            // still connected without that single edge.  Begin by marking
            // all edges as not being checked yet.
            //-------------------------------------------------------------
            foreach (NodeElement ne in nodeList)
            {
                for (i = 0; i < ne.DegreeCount; i++)
                {
                    ne.MarkEdgeChecked(i, false);
                }
            }
            //-------------------------------------------------------------
            // Loop through each edge for each node.  If the edge hasn't
            // been checked yet, remove the edge and then see if the
            // graph is still connected.  If yes, mark the edge as checked.
            // If not, the graph is not bi-connected.  In either case,
            // replace the edge we removed.
            //-------------------------------------------------------------
            for (j = 0; (j < this.NodeCount) && (bRtnval == true); j++)
            {
                nCurNode = nodeList[j].NodeID;
                for (i = 0; (i < nodeList[j].DegreeCount) && (bRtnval == true); i++)
                {
                    if (nodeList[j].IsEdgeChecked(i) == false)
                    {
                        nNextNode = nodeList[j].FindNextNode(i);
                        nodeList[nCurNode].RemoveEdge(nNextNode);
                        nodeList[nNextNode].RemoveEdge(nCurNode);
                        if (AreAllNodesConnected() == true)
                        {
                            nodeList[j].MarkEdgeChecked(i, true);
                        }
                        else
                        {
```

```
                    bRtnval = false;
                }
                nodeList[nCurNode].AddEdge(nNextNode);
                nodeList[nNextNode].AddEdge(nCurNode);
            }
        }
    }
    return (bRtnval);
}

//*****************************************************************
// Neighbor Generation Methods.
//*****************************************************************
public void GenerateRandomNeighbor()
{
    bool bValid = false;
    int nCount = 0;
    int nFirstNode = 0;
    int nSecondNode = 0;

    restoreList.Clear();
    //-------------------------------------------------------------
    // Loop through the process of removing a random edge and then
    // adding a random edge until we get a valid graph.
    //-------------------------------------------------------------
    while (bValid == false)
    {
        //---------------------------------------------------------
        // Select a random node and then pick a random edge from
        // that node.  Remove the edge from both directions.
        //---------------------------------------------------------
        nFirstNode = SelectRandomNode(-1);
        nSecondNode = SelectRandomEdge(nFirstNode);
        nodeList[nFirstNode].RemoveEdge(nSecondNode);
        nodeList[nSecondNode].RemoveEdge(nFirstNode);
        restoreList.Insert(0, new RestorePoint(nFirstNode, nSecondNode,
            RestorePoint.ACTION_REMOVE));
        //---------------------------------------------------------
        // Select 2 random nodes and if they are not already
        // connected, connect them.
        //---------------------------------------------------------
        nFirstNode = SelectRandomNode(-1);
        nSecondNode = SelectRandomNode(nFirstNode);
        if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
        {
            nodeList[nFirstNode].AddEdge(nSecondNode);
            nodeList[nSecondNode].AddEdge(nFirstNode);
```

```
                    restoreList.Insert(0, new RestorePoint(nFirstNode,
                        nSecondNode, RestorePoint.ACTION_ADD));
                    if (ValidateRandomGraph() == true)
                    {
                        //-------------------------------------------------
                        // If we have a valid graph, we're done.
                        //-------------------------------------------------
                        bValid = true;
                    }
                }
                if (bValid == false)
                {
                    //-----------------------------------------------------
                    // Our new graph was not valid.  Restore the original
                    // edge so we're back to our original graph.  Check our
                    // retry count to see if we've exceeded the limit.  If
                    // we have, we're just going to give up and return with
                    // the original graph.
                    //-----------------------------------------------------
                    RestoreNeighbor();
                    nCount++;
                    if (nCount > MAX_NEIGHBOR_RETRY)
                    {
                        bValid = true;
                    }
                }
            }
        }

        public void GenerateSmallWorldNeighbor()
        {
            bool bValid = false;
            int nCount = 0;
            int nFirstNode = 0;
            int nSecondNode = 0;

            restoreList.Clear();
            //-------------------------------------------------------------
            // Loop through the process of removing and adding edges until
            // we get a valid graph.
            //-------------------------------------------------------------
            while (bValid == false)
            {
                //---------------------------------------------------------
                // Select 2 random nodes.  If they are not sequential
                // nodes, and if they share a link, remove it.
                //---------------------------------------------------------
```

```
            nFirstNode = SelectRandomNode(-1);
            nSecondNode = SelectRandomNode(nFirstNode);
            if (NodesAreSequential(nFirstNode, nSecondNode) == false)
            {
                if (nodeList[nFirstNode].IsConnected(nSecondNode) == true)
                {
                    nodeList[nFirstNode].RemoveEdge(nSecondNode);
                    nodeList[nSecondNode].RemoveEdge(nFirstNode);
                    restoreList.Insert(0, new RestorePoint(nFirstNode,
                        nSecondNode, RestorePoint.ACTION_REMOVE));
                }
            }
            //----------------------------------------------------------
            // Select 2 random nodes.  If they are not sequential
            // nodes, and if they don't share a link, add one.
            //----------------------------------------------------------
            nFirstNode = SelectRandomNode(-1);
            nSecondNode = SelectRandomNode(nFirstNode);
            if (NodesAreSequential(nFirstNode, nSecondNode) == false)
            {
                if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
                {
                    nodeList[nFirstNode].AddEdge(nSecondNode);
                    nodeList[nSecondNode].AddEdge(nFirstNode);
                    restoreList.Insert(0, new RestorePoint(nFirstNode,
                        nSecondNode, RestorePoint.ACTION_ADD));
                }
            }
            if (ValidateSmallWorldGraph() == true)
            {
                //------------------------------------------------------
                // If we have a valid graph, we're done.
                //------------------------------------------------------
                bValid = true;
            }
            if (bValid == false)
            {
                //------------------------------------------------------
                // Our new graph was not valid.  Restore the original
                // edges so we're back to our original graph.  Check
                // our retry count to see if we've exceeded the limit.
                // If we have, we're just going to give up and return
                // with the original graph.
                //------------------------------------------------------
                RestoreNeighbor();
                nCount++;
                if (nCount > MAX_NEIGHBOR_RETRY)
```

```
                {
                    bValid = true;
                }
            }
        }
    }

    public void GenerateScaleFreeNeighbor()
    {
        bool bValid = false;
        int nCount = 0;
        int myNodeID;

        restoreList.Clear();
        //------------------------------------------------------------
        // Loop through the process of removing and adding edges until
        // we get a valid graph.
        //------------------------------------------------------------
        while (bValid == false)
        {
            //--------------------------------------------------------
            // Select a random node, remove all of the links from that
            // node and then add new links based on the scale free
            // algorithm we used when we first created the graph.
            //--------------------------------------------------------
            myNodeID = SelectRandomNode(-1);
            RemoveAllEdges(myNodeID);
            AddScaleFreeEdges(myNodeID, this.NodeCount - 1);
            if (ValidateScaleFreeGraph() == true)
            {
                //----------------------------------------------------
                // If we have a valid graph, we're done.
                //----------------------------------------------------
                bValid = true;
            }
            if (bValid == false)
            {
                //----------------------------------------------------
                // Our new graph was not valid.  Restore the original
                // edges so we're back to our original graph.  Check
                // our retry count to see if we've exceeded the limit.
                // If we have, we're just going to give up and return
                // with the original graph.
                //----------------------------------------------------
                RestoreNeighbor();
                nCount++;
                if (nCount > MAX_NEIGHBOR_RETRY)
```

```
                    {
                        bValid = true;
                    }
                }
            }
        }

        public void GenerateBiConnectedNeighbor()
        {
            bool bValid = false;
            int nCount = 0;
            int nFirstNode = 0;
            int nSecondNode = 0;

            restoreList.Clear();
            //---------------------------------------------------------------
            // Loop through the process of removing a random edge and then
            // adding a random edge until we get a valid graph.
            //---------------------------------------------------------------
            while (bValid == false)
            {
                //-----------------------------------------------------------
                // Select a random node and then pick a random edge from
                // that node.  Remove the edge from both directions.
                //-----------------------------------------------------------
                nFirstNode = SelectRandomNode(-1);
                nSecondNode = SelectRandomEdge(nFirstNode);
                nodeList[nFirstNode].RemoveEdge(nSecondNode);
                nodeList[nSecondNode].RemoveEdge(nFirstNode);
                restoreList.Insert(0, new RestorePoint(nFirstNode, nSecondNode,
                    RestorePoint.ACTION_REMOVE));
                //-----------------------------------------------------------
                // Select 2 random nodes and if they are not already
                // connected, connect them.
                //-----------------------------------------------------------
                nFirstNode = SelectRandomNode(-1);
                nSecondNode = SelectRandomNode(nFirstNode);
                if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
                {
                    nodeList[nFirstNode].AddEdge(nSecondNode);
                    nodeList[nSecondNode].AddEdge(nFirstNode);
                    restoreList.Insert(0, new RestorePoint(nFirstNode,
                        nSecondNode, RestorePoint.ACTION_ADD));
                    if (ValidateBiConnectedGraph() == true)
                    {
                        //---------------------------------------------------
                        // If we have a valid graph, we're done.
```

```
                    //-----------------------------------------------
                    bValid = true;
                }
            }
            if (bValid == false)
            {
                //---------------------------------------------------
                // Our new graph was not valid.  Restore the original
                // edge so we're back to our original graph.  Check our
                // retry count to see if we've exceeded the limit.  If
                // we have, we're just going to give up and return with
                // the original graph.
                //---------------------------------------------------
                RestoreNeighbor();
                nCount++;
                if (nCount > MAX_NEIGHBOR_RETRY)
                {
                    bValid = true;
                }
            }
        }
    }

    //******************************************************************
    // Restore Neighbor Methods.
    //******************************************************************
    public void RestoreNeighbor()
    {
        int myCount;
        int myNode1;
        int myNode2;
        int i;

        //-------------------------------------------------------------
        // To restore the random neighbor we will remove the new edge
        // and re-add the original edge.
        //-------------------------------------------------------------
        myCount = restoreList.Count();
        for (i = 0; i < myCount; i++)
        {
            myNode1 = restoreList[i].Node1ID;
            myNode2 = restoreList[i].Node2ID;
            if (restoreList[i].Action == RestorePoint.ACTION_ADD)
            {
                nodeList[myNode1].RemoveEdge(myNode2);
                nodeList[myNode2].RemoveEdge(myNode1);
            }
```

```
                else
                {
                    nodeList[myNode1].AddEdge(myNode2);
                    nodeList[myNode2].AddEdge(myNode1);
                }
            }
            restoreList.Clear();
        }

        //*********************************************************************
        // Other support Methods.
        //*********************************************************************
        public bool NodesAreSequential(int nNode1, int nNode2)
        {
            bool bRtnval = false;
            int nNextNode;

            nNextNode = nNode1 + 1;
            if (nNextNode >= this.NodeCount)
            {
                nNextNode = (nNode1 + 1) - this.NodeCount;
            }
            if (nNextNode == nNode2)
            {
                bRtnval = true;
            }
            else
            {
                nNextNode = nNode2 + 1;
                if (nNextNode >= this.NodeCount)
                {
                    nNextNode = (nNode2 + 1) - this.NodeCount;
                }
                if (nNextNode == nNode1)
                {
                    bRtnval = true;
                }
            }
            return (bRtnval);
        }

        public bool AreAllNodesConnected()
        {
            int myNodeID;
            bool bRtnval = true;

            //--------------------------------------------------------------
```

```
            // Starting at the first node in the list, mark all nodes that
            // are currently connected.  Loop through the node list and see
            // if all nodes have been visited.  If they have, then we know
            // that all nodes are connected.
            //---------------------------------------------------------------
            myNodeID = 0;
            MarkConnectedNodes(myNodeID);
            foreach (NodeElement ne in nodeList)
            {
                if ((ne.IsVisited) == false)
                {
                    bRtnval = false;
                }
            }
            return (bRtnval);
        }

        public bool AreTwoNodesConnected(int Node1, int Node2)
        {
            bool bRtnval = false;

            //---------------------------------------------------------------
            // Mark all nodes that are currently connected with our first
            // node.  If the second node is marked as visited, then we
            // know the two nodes are connected.
            //---------------------------------------------------------------
            MarkConnectedNodes(Node1);
            if ((nodeList[Node1].IsVisited) && (nodeList[Node2].IsVisited))
            {
                bRtnval = true;
            }
            return (bRtnval);
        }

        public void MarkConnectedNodes(int firstNodeID)
        {
            int i;
            int nNext = 0;
            int nCheck = 0;

            //---------------------------------------------------------------
            // Start by clearing the visited flag for each node in the
            // list.
            //---------------------------------------------------------------
            foreach (NodeElement ne in nodeList)
            {
                ne.IsVisited = false;
```

```
                ne.IsChecked = false;
            }
            //------------------------------------------------------------
            // Starting with the first node, mark all of the nodes that
            // are connected to the first node through any path.  Since
            // we are starting with the first node, mark it as visited.
            //------------------------------------------------------------
            nodeList[firstNodeID].IsVisited = true;
            while (nNext != -1)
            {
                nNext = -1;
                for (i = 0; (i < this.NodeCount) && (nNext == -1); i++)
                {
                    if ((nodeList[i].IsVisited == true) && (nodeList[i].IsChecked
                         == false))
                    {
                        nNext = i;
                    }
                }
                if (nNext != -1)
                {
                    for (i = 0; i < nodeList[nNext].DegreeCount; i++)
                    {
                        nCheck = nodeList[nNext].FindNextNode(i);
                        if ((nodeList[nCheck].IsVisited) == false)
                        {
                            nodeList[nCheck].IsVisited = true;
                        }
                    }
                    nodeList[nNext].IsChecked = true;
                }
            }
        }

        public int SelectRandomNode(int nNotNode)
        {
            bool bDone = false;
            int nRandNode = -1;
            int nCount;

            //------------------------------------------------------------
            // Select a random node that is not equal to the node ID that
            // was passed in.  To select any random node, the calling
            // routine can pass in -1.
            //------------------------------------------------------------
            nCount = 0;
            while (bDone == false)
```

```
            {
                nRandNode = myRand.Next(0, this.NodeCount);
                if (nRandNode != nNotNode)
                {
                    bDone = true;
                }
                else
                {
                    nCount++;
                    //---------------------------------------------------
                    // If we haven't found a valid random node after a
                    // bunch of tries, either take the next node up or
                    // down from the currently selected node.  If neither
                    // of these will work then we have some kind of weird
                    // illegal condition and we'll just return -1;
                    //---------------------------------------------------
                    if (nCount > 100)
                    {
                        if (nRandNode < (this.NodeCount - 1))
                        {
                            nRandNode++;
                        }
                        else if (nRandNode > 0)
                        {
                            nRandNode--;
                        }
                        else
                        {
                            nRandNode = -1;
                        }
                        bDone = true;
                    }
                }
            }
            return (nRandNode);
        }

        public int SelectRandomEdge(int myNodeID)   // Returns a Node ID
        {
            int nRandEdge;
            int nCount;

            //-------------------------------------------------------------
            // Make sure there are edges associated with the current node.
            // If so, select one at random.
            //-------------------------------------------------------------
            nCount = nodeList[myNodeID].DegreeCount;
```

```
            if (nCount > 0)
            {
                nRandEdge = myRand.Next(0, nCount);
            }
            else
            {
                nRandEdge = -1;
            }
            return (nodeList[myNodeID].GetEdgeID(nRandEdge));
        }

        public int FindPrimaryNode(int myNodeID)      // Returns a Node ID
        {
            int nNextNode;

            //---------------------------------------------------------------
            // The primary node will be just one position away from the
            // current node.
            //---------------------------------------------------------------
            nNextNode = myNodeID + 1;
            if (nNextNode >= this.NodeCount)
            {
                nNextNode = (myNodeID + 1) - this.NodeCount;
            }
            return (nNextNode);
        }

        public int FindSecondaryNode(int myNodeID)    // Returns a Node ID
        {
            int nNextNode;

            //---------------------------------------------------------------
            // The secondary node will be just two positions away from the
            // current node.
            //---------------------------------------------------------------
            nNextNode = myNodeID + 2;
            if (nNextNode >= this.NodeCount)
            {
                nNextNode = (myNodeID + 2) - this.NodeCount;
            }
            return (nNextNode);
        }

        public void RemoveAllEdges(int myNodeID)
        {
            int nNextNode;
            int myDegreeCount;
```

```

            //------------------------------------------------------------
            // This method removes all edges from the target node.
            //------------------------------------------------------------
            while (nodeList[myNodeID].DegreeCount > 0)
            {
                myDegreeCount = nodeList[myNodeID].DegreeCount;
                nNextNode = nodeList[myNodeID].GetEdgeID(myDegreeCount - 1);
                nodeList[myNodeID].RemoveEdge(nNextNode);
                nodeList[nNextNode].RemoveEdge(myNodeID);
                restoreList.Insert(0, new RestorePoint(myNodeID, nNextNode,
                    RestorePoint.ACTION_REMOVE));
            }
        }

        public void AddScaleFreeEdges(int myNodeID, int MaxNodeID)
        {
            int i;
            int nMaxDegree = -1;
            int myMaxNodeID = -1;
            double myProbability = 0.0;
            int randProbability = 0;

            //------------------------------------------------------------
            // Find the maximum degree count for the existing nodes.
            //------------------------------------------------------------
            for (i = 0; i <= MaxNodeID; i++)
            {
                if (i != myNodeID)
                {
                    if (nodeList[i].DegreeCount > nMaxDegree)
                    {
                        nMaxDegree = nodeList[i].DegreeCount;
                        myMaxNodeID = i;
                    }
                }
            }
            //------------------------------------------------------------
            // For each existing node, calculate the probability that the
            // new node will be linked to the existing node.  Add links
            // based on those probabilities.
            //------------------------------------------------------------
            for (i = 0; i <= MaxNodeID; i++)
            {
                if (i != myNodeID)
                {
```

```
                    myProbability = (((double)nodeList[i].DegreeCount) / ((double
                        )nMaxDegree)) * ((double)this.Probability);
                    randProbability = myRand.Next(0, 101);
                    if (((double)randProbability) <= myProbability)
                    {
                        nodeList[myNodeID].AddEdge(i);
                        nodeList[i].AddEdge(myNodeID);
                        restoreList.Insert(0, new RestorePoint(myNodeID, i,
                            RestorePoint.ACTION_ADD));
                    }
                }
            }
            //-------------------------------------------------------------
            // If after all of this activity the new node still has a
            // degree of 0, add an edge to the node that had the maximum
            // degree count.
            //-------------------------------------------------------------
            if (nodeList[myNodeID].DegreeCount == 0)
            {
                nodeList[myNodeID].AddEdge(myMaxNodeID);
                nodeList[myMaxNodeID].AddEdge(myNodeID);
                restoreList.Insert(0, new RestorePoint(myNodeID, myMaxNodeID,
                    RestorePoint.ACTION_ADD));
            }
        }

        //*****************************************************************
        // File Load and Save Methods.
        //*****************************************************************
        public void LoadGraphFromFile()
        {
            int i;
            int nCurNode;
            int nValue;
            int nCount;
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;

            foreach (NodeElement ne in nodeList)
            {
                ne.ClearEdgeList();
            }
            nodeList.Clear();
```

```
//---------------------------------------------------------------
// Open the file, read in and parse each line.
//---------------------------------------------------------------
szFilename = this.DirectoryName + "\\" + this.FileName + ".grf";
input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
fileReader = new StreamReader(input);
while (nDone == 0)
{
    szWrkbuf = fileReader.ReadLine();
    if (szWrkbuf != null)
    {
        inputFields = szWrkbuf.Split(' ');
        if (inputFields[0].Equals("#"))
        {
            // This is a comment line
        }
        else if (inputFields[0].Equals("c"))
        {
            // This is the line that defines the node count
            this.NodeCount = Convert.ToInt32(inputFields[1]);
            for (i = 0; i < this.NodeCount; i++)
            {
                nodeList.Add(new NodeElement(i, 0));
            }
        }
        else if (inputFields[0].Equals("n"))
        {
            // This line describes a node
            nCurNode = Convert.ToInt32(inputFields[1]);
            nCount = Convert.ToInt32(inputFields[2]);
            for (i = 0; i < nCount; i++)
            {
                nValue = Convert.ToInt32(inputFields[i + 3]);
                if (nValue >= 0)
                {
                    nodeList[nCurNode].AddEdge(nValue);
                    nodeList[nValue].AddEdge(nCurNode);
                }
            }
        }
    }
    else
    {
        nDone = 1;
    }
}
```

```
            //-----------------------------------------------------------
            // Close the file.
            //-----------------------------------------------------------
            fileReader.Close();
            input.Close();
        }

        public void SaveGraphToFile()
        {
            int i;
            int j;
            int nDone = 0;
            int nCurDegree = 0;
            string szFilename;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

            //-----------------------------------------------------------
            // Make sure we have data to write out to the file before
            // we start.
            //-----------------------------------------------------------
            if (nodeList.Count < 1)
            {
                nDone = 1;
            }

            //-----------------------------------------------------------
            // Open the file and write the initial data into it.  Then
            // write 1 line for each node in the problem.
            //-----------------------------------------------------------
            if (nDone == 0)
            {
                szFilename = this.DirectoryName + "\\" + this.FileName + ".grf";
                output = new FileStream(szFilename, FileMode.Create, FileAccess.
                    Write);
                fileWriter = new StreamWriter(output);
                fileWriter.WriteLine("#");
                fileWriter.WriteLine("#  Legend:");
                fileWriter.WriteLine("#    # - Comment line");
                fileWriter.WriteLine("#    c - Node Count");
                fileWriter.WriteLine("#    n - Individual node data");
                fileWriter.WriteLine("#          Value 1 - Node ID");
                fileWriter.WriteLine("#          Value 2 - Node Degree");
                fileWriter.WriteLine("#          Value >= 3 - Edge ID");
                fileWriter.WriteLine("#");
                fileWriter.WriteLine("c " + this.NodeCount);
```

```
                for (i = 0; i < this.NodeCount; i++)
                {
                    nCurDegree = nodeList[i].DegreeCount;
                    szWrkbuf = "n " + nodeList[i].NodeID + " " + nCurDegree;
                    for (j = 0; j < nCurDegree; j++)
                    {
                        szWrkbuf += " " + nodeList[i].GetEdgeID(j);
                    }
                    fileWriter.WriteLine(szWrkbuf);
                }
                //-----------------------------------------------------------
                // Close the file.
                //-----------------------------------------------------------
                fileWriter.Close();
                output.Close();
            }
        }

        public void LoadHCResults(int myMonotonic)
        {
            int nValue;
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;
            HillClimbResult tmphc = new HillClimbResult();

            myHCResults.Clear();
            //---------------------------------------------------------------
            // Open the file, read in and parse each line.
            //---------------------------------------------------------------
            szFilename = this.FileName;
            input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
            fileReader = new StreamReader(input);
            while (nDone == 0)
            {
                szWrkbuf = fileReader.ReadLine();
                if (szWrkbuf != null)
                {
                    inputFields = szWrkbuf.Split(' ');
                    if (inputFields[0].Equals("#"))
                    {
                        // This is a comment line
                    }
                    else
```

```
                    {
                        // This line describes a hill climber entry
                        tmphc.ColorCount = Convert.ToInt32(inputFields[0]);
                        nValue = Convert.ToInt32(inputFields[1]);
                        tmphc.ColorIsSolvable = true;
                        if (nValue == 0)
                        {
                            tmphc.ColorIsSolvable = false;
                        }
                        tmphc.ConflictCount = Convert.ToInt64(inputFields[2]);
                        tmphc.HardnessTime = Convert.ToDouble(inputFields[3]);
                        tmphc.PSIValue = Convert.ToDouble(inputFields[4]);
                        tmphc.PSITime = Convert.ToDouble(inputFields[5]);
                        myHCResults.Add(new HillClimbResult(tmphc.ColorCount, tmphc.ColorIsSolvable,
                                                            tmphc.ConflictCount, tmphc.HardnessTime,
                                                            tmphc.PSIValue, tmphc.PSITime));
                    }
                }
                else
                {
                    nDone = 1;
                }
            }

            //--------------------------------------------------------------
            // Close the file.
            //--------------------------------------------------------------
            fileReader.Close();
            input.Close();
        }

        public void SaveHCResults(int myMonotonic)
        {
            int nDone = 0;
            int nSolvable = 0;
            string szFilename;
            string szExt;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

            //--------------------------------------------------------------
            // Make sure we have results to write out to the file before
```

```
            // we start.
            //------------------------------------------------------------
            if (myHCResults.Count < 1)
            {
                nDone = 1;
            }

            //------------------------------------------------------------
            // Open the file and write each of the result lines.
            //------------------------------------------------------------
            if (nDone == 0)
            {
                if (myMonotonic == MON_HARDNESS)
                {
                    szExt = ".hrd";
                }
                else
                {
                    szExt = ".psi";
                }
                szFilename = this.DirectoryName + "\\" + this.FileName + szExt;
                output = new FileStream(szFilename, FileMode.Create, FileAccess.Write);
                fileWriter = new StreamWriter(output);
                foreach (HillClimbResult hc in myHCResults)
                {
                    nSolvable = 0;
                    if (hc.ColorIsSolvable == true)
                    {
                        nSolvable = 1;
                    }
                    szWrkbuf = hc.ColorCount.ToString() + " " + nSolvable.ToString() + " " +
                               hc.ConflictCount.ToString() + " " + hc.HardnessTime.ToString() + " " +
                               hc.PSIValue.ToString() + " " + hc.PSITime.ToString();
                    fileWriter.WriteLine(szWrkbuf);
                }
                //--------------------------------------------------------
                // Close the file.
                //--------------------------------------------------------
                fileWriter.Close();
                output.Close();
            }
        }

```

```
        public void LoadLeafResults(string LeafDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string szExt;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;
            ResultsData rData = new ResultsData();

            //------------------------------------------------------------
            // Clear the results list and open the target file.
            //------------------------------------------------------------
            resultsData.Clear();
            if (myMonotonic == MON_HARDNESS)
            {
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = LeafDir + "\\Lresults" + szExt;
            input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
            fileReader = new StreamReader(input);
            //------------------------------------------------------------
            // Read each line from the file, parse it into a results
            // structure and add it to the results list.
            //------------------------------------------------------------
            while (nDone == 0)
            {
                szWrkbuf = fileReader.ReadLine();
                if (szWrkbuf != null)
                {
                    inputFields = szWrkbuf.Split(' ');
                    if (inputFields[0].Equals("#"))
                    {
                        // This is a comment line
                    }
                    else
                    {
                        // This is a results line
                        rData.NodeCount = 0;        // Not used
                        rData.Probability = 0;      // Not used
                        rData.FileNumber = inputFields[0];
                        rData.MinHardness = Convert.ToInt64(inputFields[1]);
```

```
                        rData.MinPSI = Convert.ToDouble(inputFields[2]);
                        rData.MaxHardness = Convert.ToInt64(inputFields[3]);
                        rData.HardnessTime = 0.0;  // Not used
                        rData.MaxPSI = Convert.ToDouble(inputFields[4]);
                        rData.PSITime = 0.0;       // Not used
                        resultsData.Add(new ResultsData(rData.NodeCount, rData.
                            Probability,
                                                        rData.MinHardness, rData.
                                                            MaxHardness,
                                                        rData.MinPSI, rData.
                                                            MaxPSI,
                                                        rData.HardnessTime, rData
                                                            .PSITime,
                                                        rData.FileNumber));
                    }
                }
                else
                {
                    nDone = 1;
                }
            }

            //------------------------------------------------------------
            // Close the file.
            //------------------------------------------------------------
            fileReader.Close();
            input.Close();
        }

        public void SaveLeafResults(string LeafDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szExt;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

            //------------------------------------------------------------
            // Make sure we have results to write out to the file before
            // we start.
            //------------------------------------------------------------
            if (resultsData.Count < 1)
            {
                nDone = 1;
            }

```

```
            //----------------------------------------------------------
            // Open the file and write each of the result lines.
            //----------------------------------------------------------
            if (nDone == 0)
            {
                if (myMonotonic == MON_HARDNESS)
                {
                    szExt = ".hrd";
                }
                else
                {
                    szExt = ".psi";
                }
                szFilename = LeafDir + "\\Lresults" + szExt;
                output = new FileStream(szFilename, FileMode.Create, FileAccess.
                    Write);
                fileWriter = new StreamWriter(output);
                foreach (ResultsData rd in resultsData)
                {
                    szWrkbuf = rd.FileNumber + " " + rd.MinHardness.ToString() +
                        " " +
                               rd.MinPSI.ToString() + " " + rd.MaxHardness.
                                   ToString() + " " +
                               rd.MaxPSI.ToString();
                    fileWriter.WriteLine(szWrkbuf);
                }
                //------------------------------------------------------
                // Close the file.
                //------------------------------------------------------
                fileWriter.Close();
                output.Close();
            }
        }

        public void LoadLeafAverages(string LeafDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string szExt;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;

            //----------------------------------------------------------
            // Open the target file.
            //----------------------------------------------------------
```

```
            if (myMonotonic == MON_HARDNESS)
            {
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = LeafDir + "\\Laverages" + szExt;
            input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
            fileReader = new StreamReader(input);
            //----------------------------------------------------------------
            // Averages files only contain one line of data, but we will
            // read multiple lines just in case there are some comments
            // included.  If we happen to read multiple data lines, the
            // last line read will be the one we keep.
            //----------------------------------------------------------------
            while (nDone == 0)
            {
                szWrkbuf = fileReader.ReadLine();
                if (szWrkbuf != null)
                {
                    inputFields = szWrkbuf.Split(' ');
                    if (inputFields[0].Equals("#"))
                    {
                        // This is a comment line
                    }
                    else
                    {
                        // This is an averages line
                        averagesData.GraphType = inputFields[0];
                        averagesData.NodeCount = Convert.ToInt32(inputFields[1]);
                        averagesData.Probability = Convert.ToInt32(inputFields
                            [2]);
                        averagesData.AvgMinHardness = Convert.ToInt64(inputFields
                            [3]);
                        averagesData.AvgMinPSI = Convert.ToDouble(inputFields[4])
                            ;
                        averagesData.AvgMaxHardness = Convert.ToInt64(inputFields
                            [5]);
                        averagesData.AvgHardnessTime = Convert.ToDouble(
                            inputFields[6]);
                        averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[7])
                            ;
                        averagesData.AvgPSITime = Convert.ToDouble(inputFields
                            [8]);
                    }
```

```
                }
                else
                {
                    nDone = 1;
                }
            }

            //------------------------------------------------------------
            // Close the file.
            //------------------------------------------------------------
            fileReader.Close();
            input.Close();
        }

        public void SaveLeafAverages(string LeafDir, int myMonotonic)
        {
            string szFilename;
            string szExt;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

            //------------------------------------------------------------
            // Open the file and write out our single line of averages.
            //------------------------------------------------------------
            if (myMonotonic == MON_HARDNESS)
            {
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = LeafDir + "\\Laverages" + szExt;
            output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
                );
            fileWriter = new StreamWriter(output);
            szWrkbuf = averagesData.GraphType + " " +
                       averagesData.NodeCount.ToString() + " " +
                       averagesData.Probability.ToString() + " " +
                       averagesData.AvgMinHardness.ToString() + " " +
                       averagesData.AvgMinPSI + " " +
                       averagesData.AvgMaxHardness.ToString() + " " +
                       averagesData.AvgHardnessTime.ToString() + " " +
                       averagesData.AvgMaxPSI.ToString() + " " +
                       averagesData.AvgPSITime.ToString();
            fileWriter.WriteLine(szWrkbuf);
```

```
        //----------------------------------------------------------
        // Close the file.
        //----------------------------------------------------------
        fileWriter.Close();
        output.Close();
    }

    public void LoadProbabilityResults(string ProbabilityDir, int myMonotonic
        )
    {
        int nDone = 0;
        string szFilename;
        string szWrkbuf;
        string szExt;
        string[] inputFields;
        StreamReader fileReader;
        FileStream input;
        ResultsData rData = new ResultsData();

        //--------------------------------------------------------------
        // Clear the results list and open the target file.
        //--------------------------------------------------------------
        resultsData.Clear();
        if (myMonotonic == MON_HARDNESS)
        {
            szExt = ".hrd";
        }
        else
        {
            szExt = ".psi";
        }
        szFilename = ProbabilityDir + "\\Presults" + szExt;
        input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
        fileReader = new StreamReader(input);
        //--------------------------------------------------------------
        // Read each line from the file, parse it into a results
        // structure and add it to the results list.
        //--------------------------------------------------------------
        while (nDone == 0)
        {
            szWrkbuf = fileReader.ReadLine();
            if (szWrkbuf != null)
            {
                inputFields = szWrkbuf.Split(' ');
                if (inputFields[0].Equals("#"))
                {
                    // This is a comment line
```

```
                    }
                    else
                    {
                        // This is a results line
                        rData.NodeCount = 0;              // Not used
                        rData.Probability = Convert.ToInt32(inputFields[0]);
                        rData.FileNumber = string.Empty;  // Not used
                        rData.MinHardness = Convert.ToInt64(inputFields[1]);
                        rData.MinPSI = Convert.ToDouble(inputFields[2]);
                        rData.MaxHardness = Convert.ToInt64(inputFields[3]);
                        rData.HardnessTime = Convert.ToDouble(inputFields[4]);
                        rData.MaxPSI = Convert.ToDouble(inputFields[5]);
                        rData.PSITime = Convert.ToDouble(inputFields[6]);
                        resultsData.Add(new ResultsData(rData.NodeCount, rData.Probability,
                                                        rData.MinHardness, rData.MaxHardness,
                                                        rData.MinPSI, rData.MaxPSI,
                                                        rData.HardnessTime, rData.PSITime,
                                                        rData.FileNumber));
                    }
                }
                else
                {
                    nDone = 1;
                }
            }

            //---------------------------------------------------------------
            // Close the file.
            //---------------------------------------------------------------
            fileReader.Close();
            input.Close();
        }

        public void SaveProbabilityResults(string ProbabilityDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szExt;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

```

```
            //-----------------------------------------------------------
            // Make sure we have results to write out to the file before
            // we start.
            //-----------------------------------------------------------
            if (resultsData.Count < 1)
            {
                nDone = 1;
            }

            //-----------------------------------------------------------
            // Open the file and write each of the result lines.
            //-----------------------------------------------------------
            if (nDone == 0)
            {
                if (myMonotonic == MON_HARDNESS)
                {
                    szExt = ".hrd";
                }
                else
                {
                    szExt = ".psi";
                }
                szFilename = ProbabilityDir + "\\Presults" + szExt;
                output = new FileStream(szFilename, FileMode.Create, FileAccess.
                    Write);
                fileWriter = new StreamWriter(output);
                foreach (ResultsData rd in resultsData)
                {
                    szWrkbuf = rd.Probability.ToString() + " " + rd.MinHardness.
                        ToString() + " " +
                               rd.MinPSI.ToString() + " " + rd.MaxHardness.
                                   ToString() + " " +
                               rd.HardnessTime.ToString() + " " + rd.MaxPSI.
                                   ToString() + " " +
                               rd.PSITime.ToString();
                    fileWriter.WriteLine(szWrkbuf);
                }
                //-------------------------------------------------------
                // Close the file.
                //-------------------------------------------------------
                fileWriter.Close();
                output.Close();
            }
        }

        public void LoadProbabilityAverages(string ProbabilityDir, int
            myMonotonic)
```

```
        {
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string szExt;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;

            //------------------------------------------------------------
            // Open the target file.
            //------------------------------------------------------------
            if (myMonotonic == MON_HARDNESS)
            {
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = ProbabilityDir + "\\Paverages" + szExt;
            input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
            fileReader = new StreamReader(input);
            //------------------------------------------------------------
            // Averages files only contain one line of data, but we will
            // read multiple lines just in case there are some comments
            // included.  If we happen to read multiple data lines, the
            // last line read will be the one we keep.
            //------------------------------------------------------------
            while (nDone == 0)
            {
                szWrkbuf = fileReader.ReadLine();
                if (szWrkbuf != null)
                {
                    inputFields = szWrkbuf.Split(' ');
                    if (inputFields[0].Equals("#"))
                    {
                        // This is a comment line
                    }
                    else
                    {
                        // This is an averages line
                        averagesData.GraphType = inputFields[0];
                        averagesData.NodeCount = Convert.ToInt32(inputFields[1]);
                        averagesData.Probability = 0;   // Not used
                        averagesData.AvgMinHardness = Convert.ToInt64(inputFields
                            [2]);
```

```
                    averagesData.AvgMinPSI = Convert.ToDouble(inputFields[3]);
                    averagesData.AvgMaxHardness = Convert.ToInt64(inputFields[4]);
                    averagesData.AvgHardnessTime = Convert.ToDouble(inputFields[5]);
                    averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[6]);
                    averagesData.AvgPSITime = Convert.ToDouble(inputFields[7]);
                }
            }
            else
            {
                nDone = 1;
            }
        }

        //------------------------------------------------------------
        // Close the file.
        //------------------------------------------------------------
        fileReader.Close();
        input.Close();
    }

    public void SaveProbabilityAverages(string ProbabilityDir, int myMonotonic)
    {
        string szFilename;
        string szExt;
        string szWrkbuf;
        StreamWriter fileWriter;
        FileStream output;

        //------------------------------------------------------------
        // Open the file and write out our single line of averages.
        //------------------------------------------------------------
        if (myMonotonic == MON_HARDNESS)
        {
            szExt = ".hrd";
        }
        else
        {
            szExt = ".psi";
        }
        szFilename = ProbabilityDir + "\\Paverages" + szExt;
```

```
            output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
                );
            fileWriter = new StreamWriter(output);
            szWrkbuf = averagesData.GraphType + " " +
                       averagesData.NodeCount.ToString() + " " +
                       averagesData.AvgMinHardness.ToString() + " " +
                       averagesData.AvgMinPSI + " " +
                       averagesData.AvgMaxHardness.ToString() + " " +
                       averagesData.AvgHardnessTime.ToString() + " " +
                       averagesData.AvgMaxPSI.ToString() + " " +
                       averagesData.AvgPSITime.ToString();
            fileWriter.WriteLine(szWrkbuf);
            //----------------------------------------------------------
            // Close the file.
            //----------------------------------------------------------
            fileWriter.Close();
            output.Close();
        }

        public void LoadNodeCountResults(string NodeCountDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szWrkbuf;
            string szExt;
            string[] inputFields;
            StreamReader fileReader;
            FileStream input;
            ResultsData rData = new ResultsData();

            //--------------------------------------------------------------
            // Clear the results list and open the target file.
            //--------------------------------------------------------------
            resultsData.Clear();
            if (myMonotonic == MON_HARDNESS)
            {
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = NodeCountDir + "\\Nresults" + szExt;
            input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
            fileReader = new StreamReader(input);
            //--------------------------------------------------------------
            // Read each line from the file, parse it into a results
```

```
            // structure and add it to the results list.
            //------------------------------------------------------------
            while (nDone == 0)
            {
                szWrkbuf = fileReader.ReadLine();
                if (szWrkbuf != null)
                {
                    inputFields = szWrkbuf.Split(' ');
                    if (inputFields[0].Equals("#"))
                    {
                        // This is a comment line
                    }
                    else
                    {
                        // This is a results line
                        rData.NodeCount = Convert.ToInt32(inputFields[0]);
                        rData.Probability = 0;             // Not used
                        rData.FileNumber = string.Empty;  // Not used
                        rData.MinHardness = Convert.ToInt64(inputFields[1]);
                        rData.MinPSI = Convert.ToDouble(inputFields[2]);
                        rData.MaxHardness = Convert.ToInt64(inputFields[3]);
                        rData.HardnessTime = Convert.ToDouble(inputFields[4]);
                        rData.MaxPSI = Convert.ToDouble(inputFields[5]);
                        rData.PSITime = Convert.ToDouble(inputFields[6]);
                        resultsData.Add(new ResultsData(rData.NodeCount, rData.
                            Probability,
                                                        rData.MinHardness, rData.
                                                            MaxHardness,
                                                        rData.MinPSI, rData.
                                                            MaxPSI,
                                                        rData.HardnessTime, rData
                                                            .PSITime,
                                                        rData.FileNumber));
                    }
                }
                else
                {
                    nDone = 1;
                }
            }

            //------------------------------------------------------------
            // Close the file.
            //------------------------------------------------------------
            fileReader.Close();
            input.Close();
        }
```

```
        public void SaveNodeCountResults(string NodeCountDir, int myMonotonic)
        {
            int nDone = 0;
            string szFilename;
            string szExt;
            string szWrkbuf;
            StreamWriter fileWriter;
            FileStream output;

            //-------------------------------------------------------------
            // Make sure we have results to write out to the file before
            // we start.
            //-------------------------------------------------------------
            if (resultsData.Count < 1)
            {
                nDone = 1;
            }

            //-------------------------------------------------------------
            // Open the file and write each of the result lines.
            //-------------------------------------------------------------
            if (nDone == 0)
            {
                if (myMonotonic == MON_HARDNESS)
                {
                    szExt = ".hrd";
                }
                else
                {
                    szExt = ".psi";
                }
                szFilename = NodeCountDir + "\\Nresults" + szExt;
                output = new FileStream(szFilename, FileMode.Create, FileAccess.
                    Write);
                fileWriter = new StreamWriter(output);
                foreach (ResultsData rd in resultsData)
                {
                    szWrkbuf = rd.NodeCount.ToString() + " " + rd.MinHardness.
                        ToString() + " " +
                               rd.MinPSI.ToString() + " " + rd.MaxHardness.
                                   ToString() + " " +
                               rd.HardnessTime.ToString() + " " + rd.MaxPSI.
                                   ToString() + " " +
                               rd.PSITime.ToString();
                    fileWriter.WriteLine(szWrkbuf);
                }
```

```
            //----------------------------------------------------------
            // Close the file.
            //----------------------------------------------------------
            fileWriter.Close();
            output.Close();
        }
    }

    public void LoadNodeCountAverages(string NodeCountDir, int myMonotonic)
    {
        int nDone = 0;
        string szFilename;
        string szWrkbuf;
        string szExt;
        string[] inputFields;
        StreamReader fileReader;
        FileStream input;

        //--------------------------------------------------------------
        // Open the target file.
        //--------------------------------------------------------------
        if (myMonotonic == MON_HARDNESS)
        {
            szExt = ".hrd";
        }
        else
        {
            szExt = ".psi";
        }
        szFilename = NodeCountDir + "\\Naverages" + szExt;
        input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
        fileReader = new StreamReader(input);
        //--------------------------------------------------------------
        // Averages files only contain one line of data, but we will
        // read multiple lines just in case there are some comments
        // included.  If we happen to read multiple data lines, the
        // last line read will be the one we keep.
        //--------------------------------------------------------------
        while (nDone == 0)
        {
            szWrkbuf = fileReader.ReadLine();
            if (szWrkbuf != null)
            {
                inputFields = szWrkbuf.Split(' ');
                if (inputFields[0].Equals("#"))
                {
                    // This is a comment line
```

```
                }
                else
                {
                    // This is an averages line
                    averagesData.GraphType = inputFields[0];
                    averagesData.NodeCount = 0;     // Not used
                    averagesData.Probability = 0;   // Not used
                    averagesData.AvgMinHardness = Convert.ToInt64(inputFields
                        [3]);
                    averagesData.AvgMinPSI = Convert.ToDouble(inputFields[4])
                        ;
                    averagesData.AvgMaxHardness = Convert.ToInt64(inputFields
                        [5]);
                    averagesData.AvgHardnessTime = Convert.ToDouble(
                        inputFields[6]);
                    averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[7])
                        ;
                    averagesData.AvgPSITime = Convert.ToDouble(inputFields
                        [8]);
                }
            }
            else
            {
                nDone = 1;
            }
        }

        //------------------------------------------------------------
        // Close the file.
        //------------------------------------------------------------
        fileReader.Close();
        input.Close();
    }

    public void SaveNodeCountAverages(string NodeCountDir, int myMonotonic)
    {
        string szFilename;
        string szExt;
        string szWrkbuf;
        StreamWriter fileWriter;
        FileStream output;

        //------------------------------------------------------------
        // Open the file and write out our single line of averages.
        //------------------------------------------------------------
        if (myMonotonic == MON_HARDNESS)
        {
```

```
                szExt = ".hrd";
            }
            else
            {
                szExt = ".psi";
            }
            szFilename = NodeCountDir + "\\Naverages" + szExt;
            output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
                );
            fileWriter = new StreamWriter(output);
            szWrkbuf = averagesData.GraphType + " " +
                       averagesData.AvgMinHardness.ToString() + " " +
                       averagesData.AvgMinPSI + " " +
                       averagesData.AvgMaxHardness.ToString() + " " +
                       averagesData.AvgHardnessTime.ToString() + " " +
                       averagesData.AvgMaxPSI.ToString() + " " +
                       averagesData.AvgPSITime.ToString();
            fileWriter.WriteLine(szWrkbuf);
            //----------------------------------------------------------
            // Close the file.
            //----------------------------------------------------------
            fileWriter.Close();
            output.Close();
        }

        public void CalculateLeafResults(string LeafDir, string myGraphType,
                                         int myNodeCount, int myProbability)
        {
            int nRtnval = 0;
            string filePath = string.Empty;
            ResultsData rData = new ResultsData();
            AveragesData aData = new AveragesData();

            //--------------------------------------------------------------
            // If results and averages files already exist in the current
            // directory, delete them.
            //--------------------------------------------------------------
            if (nRtnval == 0)
            {
                filePath = LeafDir + "\\Lresults.*";
                if (File.Exists(filePath))
                {
                    File.Delete(filePath);
                }
                filePath = LeafDir + "\\Laverages.*";
                if (File.Exists(filePath))
                {
```

```
                File.Delete(filePath);
            }
            filePath = LeafDir + "\\Legend.txt";
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
            }
        }
        //---------------------------------------------------------------
        // Get a list of the hardness files in the directory.  For each
        // file, find the first and last data lines which represent
        // the minimum and maximum values for that specific file.  Save
        // the contents of that line in the results list and also add
        // the contents to the totals and count which will be used to
        // calculate the averages.
        //---------------------------------------------------------------
        if (nRtnval == 0)
        {
            string[] fileList = Directory.GetFiles(LeafDir, "grf*.hrd");
            resultsData.Clear();
            int nLastOff = 0;
            int nCurOff = 0;
            int nFilCnt = 0;
            this.DirectoryName = LeafDir;
            aData.AvgMinHardness = 0;
            aData.AvgMinPSI = 0.0;
            aData.AvgMaxHardness = 0;
            aData.AvgMaxPSI = 0.0;
            aData.AvgHardnessTime = 0.0;
            aData.AvgPSITime = 0.0;
            foreach (string filename in fileList)
            {
                nFilCnt++;
                //-------------------------------------------------------
                // Initialize working variables and load the hill
                // climber results from the file.
                //-------------------------------------------------------
                this.FileName = filename;
                LoadHCResults(MON_HARDNESS);
                nLastOff = myHCResults.Count() - 1;
                nCurOff = 0;
                rData.NodeCount = 0;        // Not used
                rData.Probability = 0;      // Not used
                rData.MinHardness = 0;
                rData.MinPSI = 0.0;
                rData.MaxHardness = 0;
                rData.MaxPSI = 0.0;
```

```
rData.HardnessTime = 0.0;
rData.PSITime = 0.0;
rData.FileNumber = filename;
//----------------------------------------------------
// Loop through each entry in the file so we can save
// the minimum, maximum, and average values.
//----------------------------------------------------
foreach (HillClimbResult hc in myHCResults)
{
    //------------------------------------------------
    // If this is the first entry, it will be the
    // minimum values for the hill climb.
    //------------------------------------------------
    if (nCurOff == 0)
    {
        rData.MinHardness = myHCResults[nCurOff].
            ConflictCount;
        rData.MinPSI = myHCResults[nCurOff].PSIValue;
        aData.AvgMinHardness += rData.MinHardness;
        aData.AvgMinPSI += rData.MinPSI;
    }
    //------------------------------------------------
    // Save totals that we will use to calculate the
    // average values.
    //------------------------------------------------
    rData.HardnessTime += myHCResults[nCurOff].HardnessTime;
    rData.PSITime += myHCResults[nCurOff].PSITime;
    //------------------------------------------------
    // If this is the last entry, it will be the
    // maximum values for the hill climb.
    //------------------------------------------------
    if (nCurOff == nLastOff)
    {
        rData.MaxHardness = myHCResults[nCurOff].
            ConflictCount;
        rData.MaxPSI = myHCResults[nCurOff].PSIValue;
        aData.AvgMaxHardness += rData.MaxHardness;
        aData.AvgMaxPSI += rData.MaxPSI;
        aData.AvgHardnessTime += (rData.HardnessTime / (
            nCurOff + 1));
        aData.AvgPSITime += (rData.PSITime / (nCurOff + 1));
    }
    nCurOff++;
}
//----------------------------------------------------
// Write out the hardness results file.
//----------------------------------------------------
```

```
                    resultsData.Add(new ResultsData(rData.NodeCount, rData.Probability,
                                                    rData.MinHardness, rData.MaxHardness,
                                                    rData.MinPSI, rData.MaxPSI, 0.0, 0.0,
                                                    rData.FileNumber));
                }
                SaveLeafResults(LeafDir, MON_HARDNESS);
                if (nFilCnt > 0)
                {
                    //------------------------------------------------------
                    // Calculate the hardness averages and write those out
                    // to the harness averages file.
                    //------------------------------------------------------
                    averagesData.GraphType = myGraphType;
                    averagesData.NodeCount = myNodeCount;
                    averagesData.Probability = myProbability;
                    averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                    averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                    averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                    averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
                    averagesData.AvgHardnessTime = aData.AvgHardnessTime / nFilCnt;
                    averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
                    SaveLeafAverages(LeafDir, MON_HARDNESS);
                }
            }

            //--------------------------------------------------------------
            // Get a list of the PSI files in the directory.  For each
            // file, find the first and last data lines which represent
            // the minimum and maximum values for that specific file.  Save
            // the contents of that line in the results list and also add
            // the contents to the totals and count which will be used to
            // calculate the averages.
            //--------------------------------------------------------------
            if (nRtnval == 0)
            {
                string[] fileList = Directory.GetFiles(LeafDir, "grf*.psi");
                resultsData.Clear();
                int nLastOff = 0;
                int nCurOff = 0;
                int nFilCnt = 0;
                this.DirectoryName = LeafDir;
                aData.AvgMinHardness = 0;
                aData.AvgMinPSI = 0.0;
```

```
                aData.AvgMaxHardness = 0;
                aData.AvgMaxPSI = 0.0;
                aData.AvgHardnessTime = 0.0;
                aData.AvgPSITime = 0.0;
                foreach (string filename in fileList)
                {
                    nFilCnt++;
                    //---------------------------------------------------
                    // Initialize working variables and load the hill
                    // climber results from the file.
                    //---------------------------------------------------
                    this.FileName = filename;
                    LoadHCResults(MON_PSI);
                    nLastOff = myHCResults.Count() - 1;
                    nCurOff = 0;
                    rData.NodeCount = 0;       // Not used
                    rData.Probability = 0;     // Not used
                    rData.MinHardness = 0;
                    rData.MinPSI = 0.0;
                    rData.MaxHardness = 0;
                    rData.MaxPSI = 0.0;
                    rData.HardnessTime = 0.0;
                    rData.PSITime = 0.0;
                    rData.FileNumber = filename;
                    //---------------------------------------------------
                    // Loop through each entry in the file so we can save
                    // the minimum, maximum, and average values.
                    //---------------------------------------------------
                    foreach (HillClimbResult hc in myHCResults)
                    {
                        //-----------------------------------------------
                        // If this is the first entry, it will be the
                        // minimum values for the hill climb.
                        //-----------------------------------------------
                        if (nCurOff == 0)
                        {
                            rData.MinHardness = myHCResults[nCurOff].
                                ConflictCount;
                            rData.MinPSI = myHCResults[nCurOff].PSIValue;
                            aData.AvgMinHardness += rData.MinHardness;
                            aData.AvgMinPSI += rData.MinPSI;
                        }
                        //-----------------------------------------------
                        // Save totals that we will use to calculate the
                        // average values.
                        //-----------------------------------------------
                        rData.HardnessTime += myHCResults[nCurOff].HardnessTime;
```

```
                    rData.PSITime += myHCResults[nCurOff].PSITime;
                    //------------------------------------------------
                    // If this is the last entry, it will be the
                    // maximum values for the hill climb.
                    //------------------------------------------------
                    if (nCurOff == nLastOff)
                    {
                        rData.MaxHardness = myHCResults[nCurOff].ConflictCount;
                        rData.MaxPSI = myHCResults[nCurOff].PSIValue;
                        aData.AvgMaxHardness += rData.MaxHardness;
                        aData.AvgMaxPSI += rData.MaxPSI;
                        aData.AvgHardnessTime += (rData.HardnessTime / (nCurOff + 1));
                        aData.AvgPSITime += (rData.PSITime / (nCurOff + 1));
                    }
                    nCurOff++;
                }
                //----------------------------------------------------
                // Write out the PSI results file.
                //----------------------------------------------------
                resultsData.Add(new ResultsData(rData.NodeCount, rData.Probability,
                                                rData.MinHardness, rData.MaxHardness,
                                                rData.MinPSI, rData.MaxPSI, 0.0, 0.0,
                                                rData.FileNumber));
            }
            SaveLeafResults(LeafDir, MON_PSI);
            if (nFilCnt > 0)
            {
                //----------------------------------------------------
                // Calculate the PSI averages and write those out to
                // the PSI averages file.
                //----------------------------------------------------
                averagesData.GraphType = myGraphType;
                averagesData.NodeCount = myNodeCount;
                averagesData.Probability = myProbability;
                averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
                averagesData.AvgHardnessTime = aData.AvgHardnessTime / nFilCnt;
                averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
                SaveLeafAverages(LeafDir, MON_PSI);
```

```
        }
    }

    //------------------------------------------------------------
    // Finally, write out the legend file.
    //------------------------------------------------------------
    if (nRtnval == 0)
    {
        string szFilename;
        szFilename = LeafDir + "\\Legend.txt";
        output = new FileStream(szFilename, FileMode.Create, FileAccess.Write);
        fileWriter = new StreamWriter(output);
        fileWriter.WriteLine("#");
        fileWriter.WriteLine("#  There are two results files in the directory");
        fileWriter.WriteLine("#    Lresults.hrd - Results when solving for hardness");
        fileWriter.WriteLine("#    Lresults.psi - Results when solving for PSI");
        fileWriter.WriteLine("#  Each line in each file contains the following elements");
        fileWriter.WriteLine("#    File number - The file number");
        fileWriter.WriteLine("#    Min Hardness - The minimum hardness value");
        fileWriter.WriteLine("#    Min PSI - The minimum PSI value");
        fileWriter.WriteLine("#    Max Hardness - The maximum hardness value");
        fileWriter.WriteLine("#    Max PSI - The maximum PSI value");
        fileWriter.WriteLine("#");
        fileWriter.WriteLine("#  There are two averages files in the directory");
        fileWriter.WriteLine("#    Laverages.hrd - Averages when solving for hardness");
        fileWriter.WriteLine("#    Laverages.psi - Averages when solving for PSI");
        fileWriter.WriteLine("#  Each file contains a single line with averages");
        fileWriter.WriteLine("#  for the current leaf directory.");
        fileWriter.WriteLine("#  The single line contains the following elements");
        fileWriter.WriteLine("#    Graph Type - The graph type");
        fileWriter.WriteLine("#    Node Count - The node count");
        fileWriter.WriteLine("#    Probability - The probability");
        fileWriter.WriteLine("#    AvgMinHardness - The average minimum hardness value");
```

```
                fileWriter.WriteLine("#    AvgMinPSI - The average minimum PSI value");
                fileWriter.WriteLine("#    AvgMaxHardness - The average maximum hardness value");
                fileWriter.WriteLine("#    AvgHardnessTime - The average time to calculate hardness (in ms)");
                fileWriter.WriteLine("#    AvgMaxPSI - The average maximum PSI value");
                fileWriter.WriteLine("#    AvgPSITime - The average time to calculate PSI (in ms)");
                fileWriter.WriteLine("#");
                fileWriter.Close();
                output.Close();
            }
        }

        public void CalculateProbabilityResults(string ProbabilityDir, string myGraphType,
                                                int myNodeCount)
        {
            int nRtnval = 0;
            string filePath = string.Empty;
            ResultsData rData = new ResultsData();
            AveragesData aData = new AveragesData();

            //-------------------------------------------------------------
            // If results and averages files already exist in the current
            // directory, delete them.
            //-------------------------------------------------------------
            if (nRtnval == 0)
            {
                filePath = ProbabilityDir + "\\Presults.*";
                if (File.Exists(filePath))
                {
                    File.Delete(filePath);
                }
                filePath = ProbabilityDir + "\\Paverages.*";
                if (File.Exists(filePath))
                {
                    File.Delete(filePath);
                }
                filePath = ProbabilityDir + "\\Legend.txt";
                if (File.Exists(filePath))
                {
                    File.Delete(filePath);
                }
            }
```

```
            //-----------------------------------------------------------
            // Get a list of all of the probability directories in the
            // current directory.  For each directory, find the leaf
            // averages for hardness and add them to the probability
            // results file.  Also calculate the averages for all of the
            // probability directories and write those into the new
            // hardness probability averages file.
            //-----------------------------------------------------------
            if (nRtnval == 0)
            {
                int nFilCnt = 0;
                string[] dirList = Directory.GetDirectories(ProbabilityDir);
                string myPath = string.Empty;
                resultsData.Clear();
                aData.AvgMinHardness = 0;
                aData.AvgMinPSI = 0.0;
                aData.AvgMaxHardness = 0;
                aData.AvgMaxPSI = 0.0;
                aData.AvgHardnessTime = 0.0;
                aData.AvgPSITime = 0.0;
                foreach (string dirname in dirList)
                {
                    nFilCnt++;
                    myPath = dirname;
                    LoadLeafAverages(myPath, MON_HARDNESS);
                    rData.NodeCount = 0;       // Not used
                    rData.Probability = averagesData.Probability;
                    rData.MinHardness = averagesData.AvgMinHardness;
                    aData.AvgMinHardness += averagesData.AvgMinHardness;
                    rData.MinPSI = averagesData.AvgMinPSI;
                    aData.AvgMinPSI += averagesData.AvgMinPSI;
                    rData.MaxHardness = averagesData.AvgMaxHardness;
                    aData.AvgMaxHardness += averagesData.AvgMaxHardness;
                    rData.MaxPSI = averagesData.AvgMaxPSI;
                    aData.AvgMaxPSI += averagesData.AvgMaxPSI;
                    rData.HardnessTime = averagesData.AvgHardnessTime;
                    aData.AvgHardnessTime += averagesData.AvgHardnessTime;
                    rData.PSITime = averagesData.AvgPSITime;
                    aData.AvgPSITime += averagesData.AvgPSITime;
                    rData.FileNumber = string.Empty;   // Not used
                    resultsData.Add(new ResultsData(rData.NodeCount, rData.Probability,
                                                    rData.MinHardness, rData.MaxHardness,
                                                    rData.MinPSI, rData.MaxPSI,
                                                    rData.HardnessTime, rData.PSITime,
```

```
                                                        rData.FileNumber));
                }
                //------------------------------------------------------------
                // Write out the hardness results file.
                //------------------------------------------------------------
                SaveProbabilityResults(ProbabilityDir, MON_HARDNESS);
                if (nFilCnt > 0)
                {
                    //--------------------------------------------------------
                    // Calculate the hardness averages and write those out
                    // to the hardness averages file.
                    //--------------------------------------------------------
                    averagesData.GraphType = myGraphType;
                    averagesData.NodeCount = myNodeCount;
                    averagesData.Probability = 0;   // Not used
                    averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                    averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                    averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                    averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
                    averagesData.AvgHardnessTime = aData.AvgHardnessTime / nFilCnt;
                    averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
                    SaveProbabilityAverages(ProbabilityDir, MON_HARDNESS);
                }
            }
            //----------------------------------------------------------------
            // Get a list of all of the probability directories in the
            // current directory.  For each directory, find the leaf
            // averages for PSI and add them to the probability results
            // file.  Also calculate the averages for all of the
            // probability directories and write those into the new
            // PSI probability averages file.
            //----------------------------------------------------------------
            if (nRtnval == 0)
            {
                int nFilCnt = 0;
                string[] dirList = Directory.GetDirectories(ProbabilityDir);
                string myPath = string.Empty;
                resultsData.Clear();
                aData.AvgMinHardness = 0;
                aData.AvgMinPSI = 0.0;
                aData.AvgMaxHardness = 0;
                aData.AvgMaxPSI = 0.0;
                aData.AvgHardnessTime = 0.0;
                aData.AvgPSITime = 0.0;
                foreach (string dirname in dirList)
                {
```

```
                    nFilCnt++;
                    myPath = dirname;
                    LoadLeafAverages(myPath, MON_PSI);
                    rData.NodeCount = 0;        // Not used
                    rData.Probability = averagesData.Probability;
                    rData.MinHardness = averagesData.AvgMinHardness;
                    aData.AvgMinHardness += averagesData.AvgMinHardness;
                    rData.MinPSI = averagesData.AvgMinPSI;
                    aData.AvgMinPSI += averagesData.AvgMinPSI;
                    rData.MaxHardness = averagesData.AvgMaxHardness;
                    aData.AvgMaxHardness += averagesData.AvgMaxHardness;
                    rData.MaxPSI = averagesData.AvgMaxPSI;
                    aData.AvgMaxPSI += averagesData.AvgMaxPSI;
                    rData.HardnessTime = averagesData.AvgHardnessTime;
                    aData.AvgHardnessTime += averagesData.AvgHardnessTime;
                    rData.PSITime = averagesData.AvgPSITime;
                    aData.AvgPSITime += averagesData.AvgPSITime;
                    rData.FileNumber = string.Empty;    // Not used
                    resultsData.Add(new ResultsData(rData.NodeCount, rData.
                        Probability,
                                                    rData.MinHardness, rData.
                                                        MaxHardness,
                                                    rData.MinPSI, rData.MaxPSI,
                                                    rData.HardnessTime, rData.
                                                        PSITime,
                                                    rData.FileNumber));
                }
                //------------------------------------------------------------
                // Write out the PSI results file.
                //------------------------------------------------------------
                SaveProbabilityResults(ProbabilityDir, MON_PSI);
                if (nFilCnt > 0)
                {
                    //--------------------------------------------------------
                    // Calculate the PSI averages and write those out to
                    // the PSI averages file.
                    //--------------------------------------------------------
                    averagesData.GraphType = myGraphType;
                    averagesData.NodeCount = myNodeCount;
                    averagesData.Probability = 0;    // Not used
                    averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                    averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                    averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                    averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
                    averagesData.AvgHardnessTime = aData.AvgHardnessTime /
                        nFilCnt;
                    averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
```

```
                SaveProbabilityAverages(ProbabilityDir, MON_PSI);
            }
        }
        //-------------------------------------------------------------
        // Finally, write out the legend file.
        //-------------------------------------------------------------
        if (nRtnval == 0)
        {
            string szFilename;
            szFilename = ProbabilityDir + "\\Legend.txt";
            output = new FileStream(szFilename, FileMode.Create, FileAccess.Write);
            fileWriter = new StreamWriter(output);
            fileWriter.WriteLine("#");
            fileWriter.WriteLine("#  There are two results files in the directory");
            fileWriter.WriteLine("#    Presults.hrd - Results when solving for hardness");
            fileWriter.WriteLine("#    Presults.psi - Results when solving for PSI");
            fileWriter.WriteLine("#  Each line in each file contains the following elements");
            fileWriter.WriteLine("#    Probability - The probability level (percent)");
            fileWriter.WriteLine("#    Min Hardness - The minimum hardness value");
            fileWriter.WriteLine("#    Min PSI - The minimum PSI value");
            fileWriter.WriteLine("#    Max Hardness - The maximum hardness value");
            fileWriter.WriteLine("#    Hardness Time - The time to calc hardness value");
            fileWriter.WriteLine("#    Max PSI - The maximum PSI value");
            fileWriter.WriteLine("#    PSI Time - The time to calc PSI value");
            fileWriter.WriteLine("#");
            fileWriter.WriteLine("#  There are two averages files in the directory");
            fileWriter.WriteLine("#    Paverages.hrd - Averages when solving for hardness");
            fileWriter.WriteLine("#    Paverages.psi - Averages when solving for PSI");
            fileWriter.WriteLine("#  Each file contains a single line with averages");
            fileWriter.WriteLine("#  for the current probability directories.");
            fileWriter.WriteLine("#  The single line contains the following elements");
```

```
            fileWriter.WriteLine("#    Graph Type - The graph type");
            fileWriter.WriteLine("#    Node Count - The node count");
            fileWriter.WriteLine("#    AvgMinHardness - The average minimum
                hardness value");
            fileWriter.WriteLine("#    AvgMinPSI - The average minimum PSI
                value");
            fileWriter.WriteLine("#    AvgMaxHardness - The average maximum
                hardness value");
            fileWriter.WriteLine("#    AvgHardnessTime - The average time to
                calculate hardness (in ms)");
            fileWriter.WriteLine("#    AvgMaxPSI - The average maximum PSI
                value");
            fileWriter.WriteLine("#    AvgPSITime - The average time to
                calculate PSI (in ms)");
            fileWriter.WriteLine("#");
            fileWriter.Close();
            output.Close();
        }
    }

    public void CalculateNodeCountResults(string NodeCountDir, string
        myGraphType)
    {
        int nRtnval = 0;
        string filePath = string.Empty;
        ResultsData rData = new ResultsData();
        AveragesData aData = new AveragesData();

        //---------------------------------------------------------------
        // If results and averages files already exist in the current
        // directory, delete them.
        //---------------------------------------------------------------
        if (nRtnval == 0)
        {
            filePath = NodeCountDir + "\\Nresults.*";
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
            }
            filePath = NodeCountDir + "\\Naverages.*";
            if (File.Exists(filePath))
            {
                File.Delete(filePath);
            }
            filePath = NodeCountDir + "\\Legend.txt";
            if (File.Exists(filePath))
            {
```

```
                File.Delete(filePath);
            }
        }
        //-------------------------------------------------------------
        // Get a list of all of the node count directories in the
        // current directory.  For each directory, find the probability
        // averages for hardness and add them to the node count
        // results file.  Also calculate the averages for all of the
        // node count directories and write those into the new
        // hardness node count averages file.
        //-------------------------------------------------------------
        if (nRtnval == 0)
        {
            int nFilCnt = 0;
            string[] dirList = Directory.GetDirectories(NodeCountDir);
            string myPath = string.Empty;
            resultsData.Clear();
            aData.AvgMinHardness = 0;
            aData.AvgMinPSI = 0.0;
            aData.AvgMaxHardness = 0;
            aData.AvgMaxPSI = 0.0;
            aData.AvgHardnessTime = 0.0;
            aData.AvgPSITime = 0.0;
            foreach (string dirname in dirList)
            {
                nFilCnt++;
                myPath = dirname;
                LoadProbabilityAverages(myPath, MON_HARDNESS);
                rData.NodeCount = averagesData.NodeCount;
                rData.Probability = 0;  // Not used
                rData.MinHardness = averagesData.AvgMinHardness;
                aData.AvgMinHardness += averagesData.AvgMinHardness;
                rData.MinPSI = averagesData.AvgMinPSI;
                aData.AvgMinPSI += averagesData.AvgMinPSI;
                rData.MaxHardness = averagesData.AvgMaxHardness;
                aData.AvgMaxHardness += averagesData.AvgMaxHardness;
                rData.MaxPSI = averagesData.AvgMaxPSI;
                aData.AvgMaxPSI += averagesData.AvgMaxPSI;
                rData.HardnessTime = averagesData.AvgHardnessTime;
                aData.AvgHardnessTime += averagesData.AvgHardnessTime;
                rData.PSITime = averagesData.AvgPSITime;
                aData.AvgPSITime += averagesData.AvgPSITime;
                rData.FileNumber = string.Empty;   // Not used
                resultsData.Add(new ResultsData(rData.NodeCount, rData.
                    Probability,
                                                rData.MinHardness, rData.
                                                    MaxHardness,
```

```
                                                        rData.MinPSI, rData.MaxPSI,
                                                        rData.HardnessTime, rData.PSITime,
                                                        rData.FileNumber));
                }
                //----------------------------------------------------------
                // Write out the hardness results file.
                //----------------------------------------------------------
                SaveNodeCountResults(NodeCountDir, MON_HARDNESS);
                if (nFilCnt > 0)
                {
                    //------------------------------------------------------
                    // Calculate the hardness averages and write those out
                    // to the hardness averages file.
                    //------------------------------------------------------
                    averagesData.GraphType = myGraphType;
                    averagesData.NodeCount = 0;     // Not used
                    averagesData.Probability = 0;   // Not used
                    averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                    averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                    averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                    averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
                    averagesData.AvgHardnessTime = aData.AvgHardnessTime / nFilCnt;
                    averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
                    SaveNodeCountAverages(NodeCountDir, MON_HARDNESS);
                }
            }
            //--------------------------------------------------------------
            // Get a list of all of the node count directories in the
            // current directory.  For each directory, find the probability
            // averages for PSI and add them to the node count results
            // file.  Also calculate the averages for all of the
            // node count directories and write those into the new
            // PSI node count averages file.
            //--------------------------------------------------------------
            if (nRtnval == 0)
            {
                int nFilCnt = 0;
                string[] dirList = Directory.GetDirectories(NodeCountDir);
                string myPath = string.Empty;
                resultsData.Clear();
                aData.AvgMinHardness = 0;
                aData.AvgMinPSI = 0.0;
                aData.AvgMaxHardness = 0;
                aData.AvgMaxPSI = 0.0;
                aData.AvgHardnessTime = 0.0;
```

```
            aData.AvgPSITime = 0.0;
            foreach (string dirname in dirList)
            {
                nFilCnt++;
                myPath = dirname;
                LoadProbabilityAverages(myPath, MON_PSI);
                rData.NodeCount = 0;       // Not used
                rData.Probability = 0;     // Not used
                rData.MinHardness = averagesData.AvgMinHardness;
                aData.AvgMinHardness += averagesData.AvgMinHardness;
                rData.MinPSI = averagesData.AvgMinPSI;
                aData.AvgMinPSI += averagesData.AvgMinPSI;
                rData.MaxHardness = averagesData.AvgMaxHardness;
                aData.AvgMaxHardness += averagesData.AvgMaxHardness;
                rData.MaxPSI = averagesData.AvgMaxPSI;
                aData.AvgMaxPSI += averagesData.AvgMaxPSI;
                rData.HardnessTime = averagesData.AvgHardnessTime;
                aData.AvgHardnessTime += averagesData.AvgHardnessTime;
                rData.PSITime = averagesData.AvgPSITime;
                aData.AvgPSITime += averagesData.AvgPSITime;
                rData.FileNumber = string.Empty;   // Not used
                resultsData.Add(new ResultsData(rData.NodeCount, rData.Probability,
                                                    rData.MinHardness, rData.MaxHardness,
                                                    rData.MinPSI, rData.MaxPSI,
                                                    rData.HardnessTime, rData.PSITime,
                                                    rData.FileNumber));
            }
            //----------------------------------------------------------
            // Write out the PSI results file.
            //----------------------------------------------------------
            SaveNodeCountResults(NodeCountDir, MON_PSI);
            if (nFilCnt > 0)
            {
                //------------------------------------------------------
                // Calculate the PSI averages and write those out to
                // the PSI averages file.
                //------------------------------------------------------
                averagesData.GraphType = myGraphType;
                averagesData.NodeCount = 0;     // Not used
                averagesData.Probability = 0;   // Not used
                averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
                averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
                averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
                averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
```

```
                averagesData.AvgHardnessTime = aData.AvgHardnessTime / nFilCnt;
                averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
                SaveNodeCountAverages(NodeCountDir, MON_PSI);
            }
        }
        //--------------------------------------------------------------
        // Finally, write out the legend file.
        //--------------------------------------------------------------
        if (nRtnval == 0)
        {
            string szFilename;
            szFilename = NodeCountDir + "\\Legend.txt";
            output = new FileStream(szFilename, FileMode.Create, FileAccess.Write);
            fileWriter = new StreamWriter(output);
            fileWriter.WriteLine("#");
            fileWriter.WriteLine("#  There are two results files in the directory");
            fileWriter.WriteLine("#    Nresults.hrd - Results when solving for hardness");
            fileWriter.WriteLine("#    Nresults.psi - Results when solving for PSI");
            fileWriter.WriteLine("#  Each line in each file contains the following elements");
            fileWriter.WriteLine("#    Node Count - The node count value");
            fileWriter.WriteLine("#    Min Hardness - The minimum hardness value");
            fileWriter.WriteLine("#    Min PSI - The minimum PSI value");
            fileWriter.WriteLine("#    Max Hardness - The maximum hardness value");
            fileWriter.WriteLine("#    Hardness Time - The time to calc hardness value");
            fileWriter.WriteLine("#    Max PSI - The maximum PSI value");
            fileWriter.WriteLine("#    PSI Time - The time to calc PSI value");
            fileWriter.WriteLine("#");
            fileWriter.WriteLine("#  There are two averages files in the directory");
            fileWriter.WriteLine("#    Naverages.hrd - Averages when solving for hardness");
            fileWriter.WriteLine("#    Naverages.psi - Averages when solving for PSI");
            fileWriter.WriteLine("#  Each file contains a single line with averages");
            fileWriter.WriteLine("#  for the current node count directories.");
```

```
                fileWriter.WriteLine("#  The single line contains the following
                    elements");
                fileWriter.WriteLine("#    Graph Type - The graph type");
                fileWriter.WriteLine("#    AvgMinHardness - The average minimum
                    hardness value");
                fileWriter.WriteLine("#    AvgMinPSI - The average minimum PSI
                    value");
                fileWriter.WriteLine("#    AvgMaxHardness - The average maximum
                    hardness value");
                fileWriter.WriteLine("#    AvgHardnessTime - The average time to
                    calculate hardness (in ms)");
                fileWriter.WriteLine("#    AvgMaxPSI - The average maximum PSI
                    value");
                fileWriter.WriteLine("#    AvgPSITime - The average time to
                    calculate PSI (in ms)");
                fileWriter.WriteLine("#");
                fileWriter.Close();
                output.Close();
            }
        }

        //*********************************************************************
        // Miscellaneous Methods.
        //*********************************************************************
        public void ClearNodeArray()
        {
            int i;
            int j;

            for (i = 0; i < MAX_NODES; i++)
            {
                for (j = 0; j < MAX_NODES; j++)
                {
                    nodeArray[i, j] = 0;
                }
            }
        }

        public void CopyGraphToArray()
        {
            int i, j;
            int nCount;
            int nDegree;
            int nNode;

            //-------------------------------------------------------------
            // Clear the node array.
```

```
        //------------------------------------------------------------
        ClearNodeArray();

        //------------------------------------------------------------
        // For each entry in the node list, add the edges to the array.
        //------------------------------------------------------------
        nCount = nodeList.Count();
        for (i = 0; i < nCount; i++)
        {
            //--------------------------------------------------------
            // Look at each 'edge' slot for this node.  For each that
            // has a value in the legal range we will add the edges
            // to the node array.
            //--------------------------------------------------------
            nDegree = nodeList[i].DegreeCount;
            for (j = 0; j < nDegree; j++)
            {
                nNode = nodeList[i].GetEdgeID(j);
                if ((nNode >= 0) && (nNode < nCount))
                {
                    nodeArray[i, nNode] = 1;
                    nodeArray[nNode, i] = 1;
                }
            }
        }
    }

    public void SolveForColor(int nColorCount)
    {
        int i = 0;
        int nCount = 0;
        int nCurDegree = 0;
        int nEdgeOffset = 0;
        int nConflict = 0;
        int nColor = 0;
        int nDone = 0;
        int nSolved = 0;

        //------------------------------------------------------------
        // Initialize our node count and current depth in the node
        // list.  Do some validation to make sure we have at least
        // one node and at least 2 colors.
        //------------------------------------------------------------
        nCount = nodeList.Count();
        nCurDepth = 0;
        this.ConflictCount = 0;
        if ((nColorCount < 2) || (nCount < 1))
```

```
            {
                nSolved = -1;
            }
            //-----------------------------------------------------------
            // Loop through the node list and set the current color for
            // each node to 0, which means no color is assigned yet.
            //-----------------------------------------------------------
            if (nSolved == 0)
            {
                foreach (NodeElement ne in nodeList)
                {
                    ne.NodeColor = 0;
                }
            }
            //-----------------------------------------------------------
            // Set the initial color for the first node, then go into our
            // processing loop.
            //-----------------------------------------------------------
            nodeList[nCurDepth].NodeColor = 1;
            while ((nSolved == 0) && (nDoneFlag == 0))
            {
                //-------------------------------------------------------
                // Check for a color conflict with our current node.
                //-------------------------------------------------------
                nCurDegree = nodeList[nCurDepth].DegreeCount;
                nConflict = 0;
                for (i = 0; (i < nCurDegree) && (nConflict == 0); i++)
                {
                    nEdgeOffset = nodeList[nCurDepth].GetEdgeID(i);
                    if (nodeList[nCurDepth].NodeColor == nodeList[nEdgeOffset].
                        NodeColor)
                    {
                        nConflict = 1;
                    }
                }
                //-------------------------------------------------------
                // If we have a color conflict, increment the color for
                // the current node.  If we have tried all colors for this
                // node, start back tracking until we find a node that
                // hasn't had all colors tried.  If we've tried all colors
                // for all nodes in the backtracking chain, then there is
                // no solution to color this graph.
                //-------------------------------------------------------
                if (nConflict == 1)
                {
                    nConflictCount++;
                    nDone = 0;
```

```
                while ((nDone == 0) && (nSolved == 0))
                {
                    nColor = nodeList[nCurDepth].NodeColor;
                    nColor++;
                    if (nColor > nColorCount)
                    {
                        nodeList[nCurDepth].NodeColor = 0;
                        if (nCurDepth > 0)
                        {
                            nCurDepth--;
                        }
                        else
                        {
                            nSolved = -1;
                        }
                    }
                    else
                    {
                        nodeList[nCurDepth].NodeColor = nColor;
                        nDone = 1;
                    }
                }
            }
            else
            {
                //---------------------------------------------------
                // There is no color conflict for the current node.
                // Move down to the next level and try that one.  If
                // we've reached the bottom level (leaf node) then
                // we've found a solution for the problem.
                //---------------------------------------------------
                nCurDepth++;
                if (nCurDepth >= nCount)
                {
                    nSolved = 1;
                }
                else
                {
                    nodeList[nCurDepth].NodeColor = 1;
                }
            }
        }
        if (nSolved == 1)
        {
            this.ColorSolvable = true;
        }
        else
```

```
            {
                this.ColorSolvable = false;
            }
        }

        public void SolveForPSI()
        {
            //------------------------------------------------------------
            // This function takes the graph represented in the nodeArray
            // matrix where a value of 1 represents an edge and 0 no edge.
            // The function returns a double that represents the complexity
            // of the graph (which should be higher when more information
            // is contained in the graph).  We assume that the graph has
            // no self connections.
            //------------------------------------------------------------
            int n = this.NodeCount;
            int[,] c_i = new int[n, 2]; // the count of connections to a single node. the second index is not to i ,0] or connections to i ,1]
            double[,] p_i = new double[n, 2]; //as above but probabilities
            int[, , ,] c_ij = new int[n, n, 2, 2]; //count of connections between two nodes through another node, last two indexes similar to above
            double[, , ,] p_ij = new double[n, n, 2, 2]; // as above but probabilities

            // fill in p_i by looking at connectivity between each i and all the other nodes
            nStep = 1;
            myActivity = "Fill in p_i based on node connectivity";
            for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
                for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                    for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                        if (i != j && nodeArray[i, j] == connect)
                            c_i[i, connect]++;
            // fill in c_ij by looking at connectivity between i and j through all other nodes
            nStep = 2;
            myActivity = "Fill in c_ij based on node connectivity";
            for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0); connect_i++)
                for (int connect_j = 0; (connect_j < 2) && (nDoneFlag == 0); connect_j++)
                    for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                        for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                            for (int m = 0; (m < n) && (nDoneFlag == 0); m++) // middle node
```

```
                        if (i != j && i != m && j != m && nodeArray[i, m] == connect_i && nodeArray[m, j] == connect_j)
                            c_ij[i, j, connect_i, connect_j]++;
            // convert to probabilities by dividing by the total number of possibilities
            //(taking into account that each node must be distinct)
            nStep = 3;
            myActivity = "Convert to Probabilities - part 1";
            for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
                for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                    p_i[i, connect] = c_i[i, connect] / (double)(n - 1); // n-1 because connecting to self is not an option
            // convert to probabilities by dividing by the total n-2 because middle node must not be start or end
            nStep = 4;
            myActivity = "Convert to Probabilities - part 2";
            for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0); connect_i++)
                for (int connect_j = 0; (connect_j < 2) && (nDoneFlag == 0); connect_j++)
                    for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                        for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                            p_ij[i, j, connect_i, connect_j] = c_ij[i, j, connect_i, connect_j] / (double)(n - 2);
            // compute the shannon entropy
            nStep = 5;
            myActivity = "Compute the Shannon Entropy";
            double[] k = new double[n];
            for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
                for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                    if (p_i[i, connect] != 0)
                        k[i] += -1 * p_i[i, connect] * Math.Log(p_i[i, connect], 2);
            // compute the marginalized pi and pj (sum probabilities over other variable)
            nStep = 6;
            myActivity = "Compute Marginalized pi and pj";
            double[, ,] p_ijMa = new double[n, n, 2];
            double[, ,] p_ijMb = new double[n, n, 2];
            for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                    if (i != j)
                    {
                        // sum over b (second index) for Ma
                        p_ijMa[i, j, 0] += p_ij[i, j, 0, 0] + p_ij[i, j, 0, 1];
                        p_ijMa[i, j, 1] += p_ij[i, j, 1, 0] + p_ij[i, j, 1, 1];
```

```
                    // sum over a (first index) for Mb
                    p_ijMb[i, j, 0] += p_ij[i, j, 0, 0] + p_ij[i, j, 1, 0];
                    p_ijMb[i, j, 1] += p_ij[i, j, 0, 1] + p_ij[i, j, 1, 1];
                }
            // compute the mutual information m[]
            nStep = 7;
            myActivity = "Compute the Mutual Information";
            double[,] mI = new double[n, n];
            // for each connection compute the mutual information
            for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                    for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0); connect_i++)
                        for (int connect_j = 0; (connect_j < 2) && (nDoneFlag == 0); connect_j++)
                            if (p_ij[i, j, connect_i, connect_j] != 0.0 && p_i[i, connect_i] != 0.0 && p_i[j, connect_j] != 0.0)
                                mI[i, j] += p_ij[i, j, connect_i, connect_j] * Math.Log(p_ij[i, j, connect_i, connect_j]
                                    / (p_ijMa[i, j, connect_i] * p_ijMb[i, j, connect_j]), 2);
            // compute psi
            nStep = 8;
            myActivity = "Compute PSI";
            double psi = 0.0, sum = 0.0;
            for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
                for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
                    if (i != j)
                        sum += Math.Max(k[i], k[j]) * mI[i, j] * (1 - mI[i, j]);
            psi = 4 * sum / (n * (n - 1));
            PSIcalculatedValue = psi;
        }

        public void LogDebug(string szDebug)
        {
            string szFilename;

            //------------------------------------------------------------
            // Open the file and append the results to it.
            //------------------------------------------------------------
            szFilename = DirectoryName + "\\Debug.txt";
            output = new FileStream(szFilename, FileMode.Append, FileAccess.Write);
            fileWriter = new StreamWriter(output);
            fileWriter.WriteLine(szDebug);

            //------------------------------------------------------------
```

```
            // Close the file.
            //------------------------------------------------------------
            fileWriter.Close();
            output.Close();
        }
    }
}
```

A.4 Calculate Averages Data (AveragesData.cs)

This class is used to calculate and store all of the averages data included in the results.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class AveragesData
    {
        private string szGraphType = string.Empty;
        private int nNodeCount = 0;
        private int nProbability = 0;
        private long nAvgMinHardness = 0;
        private double fAvgMinPSI = 0.0;
        private long nAvgMaxHardness = 0;
        private double fAvgHardnessTime = 0.0;
        private double fAvgMaxPSI = 0.0;
        private double fAvgPSITime = 0.0;

        public AveragesData()
        {
        }

        public AveragesData(string myGraphType, int myNodeCount,
                            int myProbability, long myAvgMinHardness,
                            double myAvgMinPSI, long myAvgMaxHardness,
                            double myAvgHardnessTime, double myAvgMaxPSI,
                            double myAvgPSITime)
        {
            this.szGraphType = myGraphType;
            this.nNodeCount = myNodeCount;
            this.nProbability = myProbability;
```

```
            this.nAvgMinHardness = myAvgMinHardness;
            this.fAvgMinPSI = myAvgMinPSI;
            this.nAvgMaxHardness = myAvgMaxHardness;
            this.fAvgHardnessTime = myAvgHardnessTime;
            this.fAvgMaxPSI = myAvgMaxPSI;
            this.fAvgPSITime = myAvgPSITime;
        }

        public string GraphType
        {
            get { return szGraphType; }
            set { szGraphType = value; }
        }

        public int NodeCount
        {
            get { return nNodeCount; }
            set { nNodeCount = value; }
        }

        public int Probability
        {
            get { return nProbability; }
            set { nProbability = value; }
        }

        public long AvgMinHardness
        {
            get { return nAvgMinHardness; }
            set { nAvgMinHardness = value; }
        }

        public double AvgMinPSI
        {
            get { return fAvgMinPSI; }
            set { fAvgMinPSI = value; }
        }

        public long AvgMaxHardness
        {
            get { return nAvgMaxHardness; }
            set { nAvgMaxHardness = value; }
        }

        public double AvgHardnessTime
        {
            get { return fAvgHardnessTime; }
```

```
            set { fAvgHardnessTime = value; }
        }

        public double AvgMaxPSI
        {
            get { return fAvgMaxPSI; }
            set { fAvgMaxPSI = value; }
        }

        public double AvgPSITime
        {
            get { return fAvgPSITime; }
            set { fAvgPSITime = value; }
        }
    }
}
```

A.5 Edge Class (EdgeElement.cs)

This is a simple class that is instantiated to create an object for each edge in a graph.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class EdgeElement
    {
        private int nNodeID;
        private bool bEdgeChecked;

        public EdgeElement()
        {
        }

        public EdgeElement(int nID)
        {

            //-------------------------------------------------------------
            // Set the ID of the node that this edge connects to.
            //-------------------------------------------------------------
            this.nNodeID = nID;
```

```
        }

        public int NodeID
        {
            get { return nNodeID; }
            set { nNodeID = value; }
        }

        public bool IsChecked
        {
            get { return bEdgeChecked; }
            set { bEdgeChecked = value; }
        }
    }
}
```

A.6 Hill Climber Results (HillClimbResult.cs)

This class tracks results from the hill climber algorithm.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class HillClimbResult
    {
        private int nColorCount = 0;
        private bool bColorIsSolvable = false;
        private long nConflictCount = 0;
        private double fHardnessTime = 0.0;
        private double fPSIValue = 0.0;
        private double fPSITime = 0.0;

        public HillClimbResult()
        {
        }

        public HillClimbResult(int myColorCount, bool myColorIsSolvable,
                               long myConflictCount, double myHardnessTime,
                               double myPSIValue, double myPSITime)
        {
```

```
            this.nColorCount = myColorCount;
            this.bColorIsSolvable = myColorIsSolvable;
            this.nConflictCount = myConflictCount;
            this.fHardnessTime = myHardnessTime;
            this.fPSIValue = myPSIValue;
            this.fPSITime = myPSITime;
        }

        public int ColorCount
        {
            get { return nColorCount; }
            set { nColorCount = value; }
        }

        public bool ColorIsSolvable
        {
            get { return bColorIsSolvable; }
            set { bColorIsSolvable = value; }
        }

        public long ConflictCount
        {
            get { return nConflictCount; }
            set { nConflictCount = value; }
        }

        public double HardnessTime
        {
            get { return fHardnessTime; }
            set { fHardnessTime = value; }
        }

        public double PSIValue
        {
            get { return fPSIValue; }
            set { fPSIValue = value; }
        }

        public double PSITime
        {
            get { return fPSITime; }
            set { fPSITime = value; }
        }
    }
}
```

A.7 Node Class (NodeElement.cs)

This is a simple class that is instantiated to create an object for each node in a graph.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class NodeElement
    {
        private int nNodeID;
        private int nNodeColor;
        private int nNodeDegree;
        private bool bNodeVisited;
        private bool bNodeChecked;
        private IList<EdgeElement> edgeList = new List<EdgeElement>();

        public NodeElement()
        {
        }

        public NodeElement(int nID, int nColor)
        {

            //------------------------------------------------------------
            // Set the ID and color for the node.  Then clear the edge
            // list to show no edges.
            //------------------------------------------------------------
            this.nNodeID = nID;
            this.nNodeColor = nColor;
            this.nNodeDegree = 0;
            this.bNodeVisited = false;
            this.edgeList.Clear();
        }

        public int NodeID
        {
            get { return nNodeID; }
            set { nNodeID = value; }
        }

        public int NodeColor
```

```
        {
            get { return nNodeColor; }
            set { nNodeColor = value; }
        }

        public int DegreeCount
        {
            get { return nNodeDegree; }
            set { nNodeDegree = value; }
        }

        public bool IsVisited
        {
            get { return bNodeVisited; }
            set { bNodeVisited = value; }
        }

        public bool IsChecked
        {
            get { return bNodeChecked; }
            set { bNodeChecked = value; }
        }

        public int AddEdge(int nCompanionNode)
        {
            int nRtnval = 0;
            bool bFound = false;

            //------------------------------------------------------------
            // Search the edge list to see if we are aleady connected to
            // the target node.  If so, this method does nothing.
            // Otherwise we will make the connection.
            //------------------------------------------------------------
            foreach (EdgeElement ee in edgeList)
            {
                if (ee.NodeID == nCompanionNode)
                {
                    bFound = true;
                }
            }
            if (bFound == false)
            {
                edgeList.Add(new EdgeElement(nCompanionNode));
                this.DegreeCount++;
            }
            return (nRtnval);
        }
```

```
    public bool IsConnected(int nNextNode)
    {
        bool bRtnval = false;

        foreach (EdgeElement ee in edgeList)
        {
            if (ee.NodeID == nNextNode)
            {
                bRtnval = true;
            }
        }
        return (bRtnval);
    }

    public int GetEdgeID(int nOffset)
    {
        int nRtnval = -1;
        int nCount = 0;

        nCount = edgeList.Count();
        if ((nOffset >= 0) && (nOffset < nCount))
        {
            nRtnval = edgeList[nOffset].NodeID;
        }
        return (nRtnval);
    }

    public bool IsEdgeChecked(int nOffset)
    {
        bool bRtnval = false;
        int nCount = 0;

        nCount = edgeList.Count();
        if ((nOffset >= 0) && (nOffset < nCount))
        {
            bRtnval = edgeList[nOffset].IsChecked;
        }
        return (bRtnval);
    }

    public void MarkEdgeChecked(int nOffset, bool bChecked)
    {
        int nCount = 0;

        nCount = edgeList.Count();
        if ((nOffset >= 0) && (nOffset < nCount))
```

```
        {
            edgeList[nOffset].IsChecked = bChecked;
        }
    }

    public int FindEdgeOffset(int nEdgeID)
    {
        int i;
        int nCount = 0;
        int nOffset = -1;

        nCount = edgeList.Count();
        for (i = 0; (i < nCount) && (nOffset == -1); i++)
        {
            if (edgeList[i].NodeID == nEdgeID)
            {
                nOffset = i;
            }
        }
        return (nOffset);
    }

    public void RemoveEdge(int nEdgeID)
    {
        int nOffset = -1;

        nOffset = this.FindEdgeOffset(nEdgeID);
        if (nOffset != -1)
        {
            this.DegreeCount--;
            if (this.DegreeCount < 0)
            {
                this.DegreeCount = 0;
            }
            edgeList.RemoveAt(nOffset);
        }
    }

    public int FindNextNode(int myEdgeOff)
    {
        int nNextNodeID;

        nNextNodeID = this.edgeList[myEdgeOff].NodeID;
        return (nNextNodeID);
    }

    public void ClearEdgeList()
```

```
        {
            this.edgeList.Clear();
        }
    }
}
```

A.8 Track Restore Point (RestorePoint.cs)

While running the hill climber, neighbors are created in an attempt to find the next graph where the monotonic value increases. Each time a neighbor is created where this value doesn't increase, we throw the neighbor away and revert to the previous graph. The restore point class keeps enough information to allow a return to the previous graph.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class RestorePoint
    {
        //---------------------------------------------------------------
        // Define constants.
        //---------------------------------------------------------------
        public const int ACTION_ADD = 0;
        public const int ACTION_REMOVE = 1;
        //---------------------------------------------------------------
        // Define data elements for the class.
        //---------------------------------------------------------------
        private int nNode1ID;
        private int nNode2ID;
        private int nAction;

        public RestorePoint()
        {
        }

        public RestorePoint(int myNode1, int myNode2, int myAction)
        {

            //-----------------------------------------------------------
```

```
            // Set the ID of the 2 nodes involved and the action that
            // was taken.
            //------------------------------------------------------------
            this.nNode1ID = myNode1;
            this.nNode2ID = myNode2;
            this.nAction = myAction;
        }

        public int Node1ID
        {
            get { return nNode1ID; }
            set { nNode1ID = value; }
        }

        public int Node2ID
        {
            get { return nNode2ID; }
            set { nNode2ID = value; }
        }

        public int Action
        {
            get { return nAction; }
            set { nAction = value; }
        }
    }
}
```

A.9 Results Data (ResultsData.cs)

This class is used to track results while we are solving the problem.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GraphandPSISolver
{
    class ResultsData
    {
        private int nNodeCount = 0;
        private int nProbability = 0;
        private long nMinHardness = 0;
```

```
        private long nMaxHardness = 0;
        private double fMinPSI = 0.0;
        private double fMaxPSI = 0.0;
        private double fHardnessTime = 0.0;
        private double fPSITime = 0.0;
        private string szFileNumber = string.Empty;

        public ResultsData()
        {
        }

        public ResultsData(int myNodeCount, int myProbability,
                           long myMinHardness, long myMaxHardness,
                           double myMinPSI, double myMaxPSI,
                           double myHardnessTime, double myPSITime,
                           string myFileNumber)
        {
            this.nNodeCount = myNodeCount;
            this.nProbability = myProbability;
            this.nMinHardness = myMinHardness;
            this.nMaxHardness = myMaxHardness;
            this.fMinPSI = myMinPSI;
            this.fMaxPSI = myMaxPSI;
            this.fHardnessTime = myHardnessTime;
            this.fPSITime = myPSITime;
            this.szFileNumber = myFileNumber;
        }

        public int NodeCount
        {
            get { return nNodeCount; }
            set { nNodeCount = value; }
        }

        public int Probability
        {
            get { return nProbability; }
            set { nProbability = value; }
        }

        public string FileNumber
        {
            get { return szFileNumber; }
            set { szFileNumber = value; }
        }

        public long MinHardness
```

```
        {
            get { return nMinHardness; }
            set { nMinHardness = value; }
        }

        public double MinPSI
        {
            get { return fMinPSI; }
            set { fMinPSI = value; }
        }

        public long MaxHardness
        {
            get { return nMaxHardness; }
            set { nMaxHardness = value; }
        }

        public double HardnessTime
        {
            get { return fHardnessTime; }
            set { fHardnessTime = value; }
        }

        public double MaxPSI
        {
            get { return fMaxPSI; }
            set { fMaxPSI = value; }
        }

        public double PSITime
        {
            get { return fPSITime; }
            set { fPSITime = value; }
        }
    }
}
```