

Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations

Graduate Studies

5-2014

Information and Hardness Quantification of Graphs: A Computational Study

Brent Dutson
Utah State University

Follow this and additional works at: <https://digitalcommons.usu.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Dutson, Brent, "Information and Hardness Quantification of Graphs: A Computational Study" (2014). *All Graduate Theses and Dissertations*. 3903.

<https://digitalcommons.usu.edu/etd/3903>

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.



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

Copyright © Brent Dutson 2014

All Rights Reserved

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

	Page
ABSTRACT	iii
PUBLIC ABSTRACT	iv
ACKNOWLEDGMENTS	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER	
1 INTRODUCTION	1
2 METHODS	6
2.1 Calculation of Hardness	6
2.2 Calculation of Ψ	7
2.3 Problem Set	11
2.4 Data Set	12
2.5 Hill Climbing to Increase Hardness or Ψ	13
2.6 User Interface	21
3 RESULTS	23
3.1 Individual Trials	24
3.2 Multiple Trials maximizing Ψ Monotonically	26
3.3 Multiple Trials maximizing Hardness Monotonically	26
3.4 The Influence of Connection Density	27
3.5 The Influence of Node Count	30
3.6 Aggregated Analysis	32
3.7 Comparison of Execution Times	35
4 CONCLUSIONS	37
REFERENCES	40
APPENDICES	43
Appendix A Source Code	44
A.1 Main User Interface Screen (Form1.cs)	44
A.2 Operation Management Screen (Generate.cs)	46
A.3 Main Solver and Algorithm Implementation (grfFile.cs)	61
A.4 Calculate Averages Data (AveragesData.cs)	135
A.5 Edge Class (EdgeElement.cs)	137

A.6 Hill Climber Results (HillClimbResult.cs)	138
A.7 Node Class (NodeElement.cs)	140
A.8 Track Restore Point (RestorePoint.cs)	144
A.9 Results Data (ResultsData.cs)	145

LIST OF TABLES

Table		Page
1.1	Calculation for Number of Routes	3
1.2	Possible Significant Research Outcomes	4
3.1	Hardness Ranges for the Heat Plot	33

LIST OF FIGURES

Figure	Page
1.1 Possible Delivery Routes with Three Destinations	2
1.2 Possible Routes for Three Destinations	2
1.3 Possible Routes for Four Destinations	2
2.1 Example A - Simple Undirected Graph	8
2.2 Example B - Simple Undirected Graph	11
2.3 Working Directory Structure	13
2.4 Hill Climbing Algorithm to run the tests	15
2.5 Create a Random Graph	16
2.6 Validate a Random Graph	16
2.7 Generate a Neighbor Random Graph	17
2.8 Create a Bi-connected Graph	17
2.9 Validate a Bi-Connected Graph	18
2.10 Generate a Neighbor Bi-connected Graph	18
2.11 Create a Scale Free Graph	19
2.12 Validate a Scale Free Graph	19
2.13 Generate a Neighbor Scale Free Graph	19
2.14 Create a Small World Graph	20
2.15 Validate a Small World Graph	20
2.16 Generate a Neighbor Small World Graph	21
2.17 Main User Interface Screen	21
2.18 Operation Management Screen	22

3.1	Individual Trial Results	25
3.2	Maximizing Ψ Results	27
3.3	Maximizing Hardness Results	28
3.4	Influence of Connection Probability Results	29
3.5	Influence of Node Count Results	31
3.6	Aggregated Data Results	34
3.7	Execution Times Results	36

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.

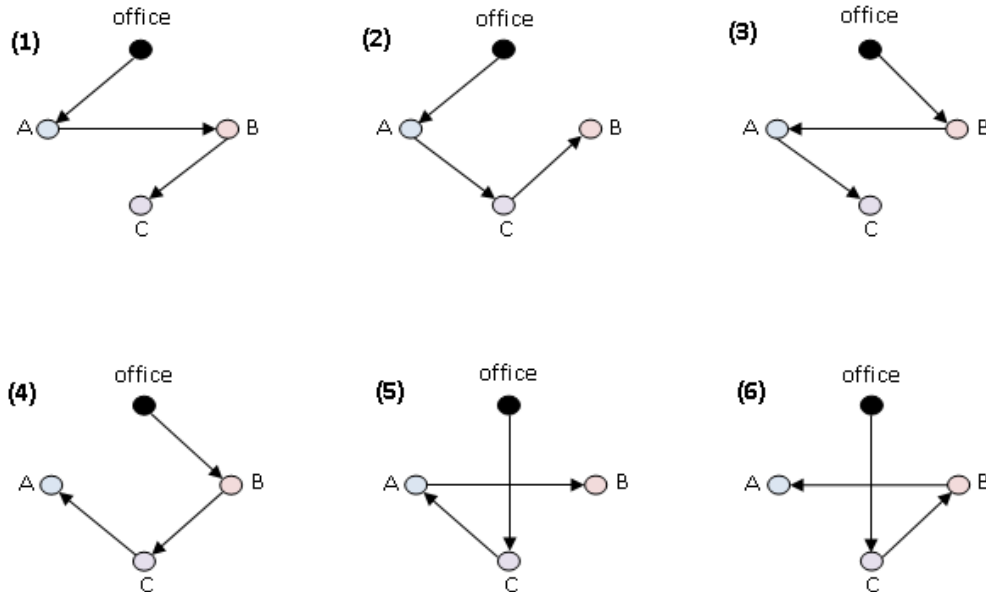


Figure 1.1: Possible Delivery Routes with Three Destination.

$$3 * 2 * 1 = 6$$

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, known 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

<i>Hardness</i>	\perp	Ψ
<i>Hardness increases</i>	\Rightarrow	<i>Ψ increases</i>
<i>Hardness increases</i>	\Rightarrow	<i>Ψ decreases</i>
<i>Ψ increases</i>	\Rightarrow	<i>Hardness increases</i>
<i>Ψ increases</i>	\Rightarrow	<i>Hardness decreases</i>

***Hardness* \perp Ψ :** The \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].

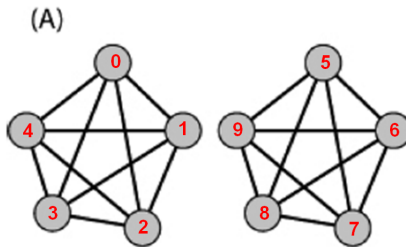


Figure 2.1: Example A of a simple undirected graph used to validate the Ψ code. The Ψ of this graph is 0.0766

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:

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

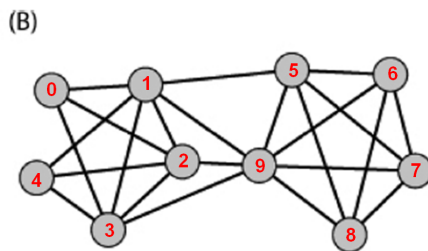


Figure 2.2: Example B of a simple undirected graph used to validate the Ψ code. The Ψ of this graph is 0.756

$$\text{for } 0 \leq \Psi \leq 1.0$$

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.

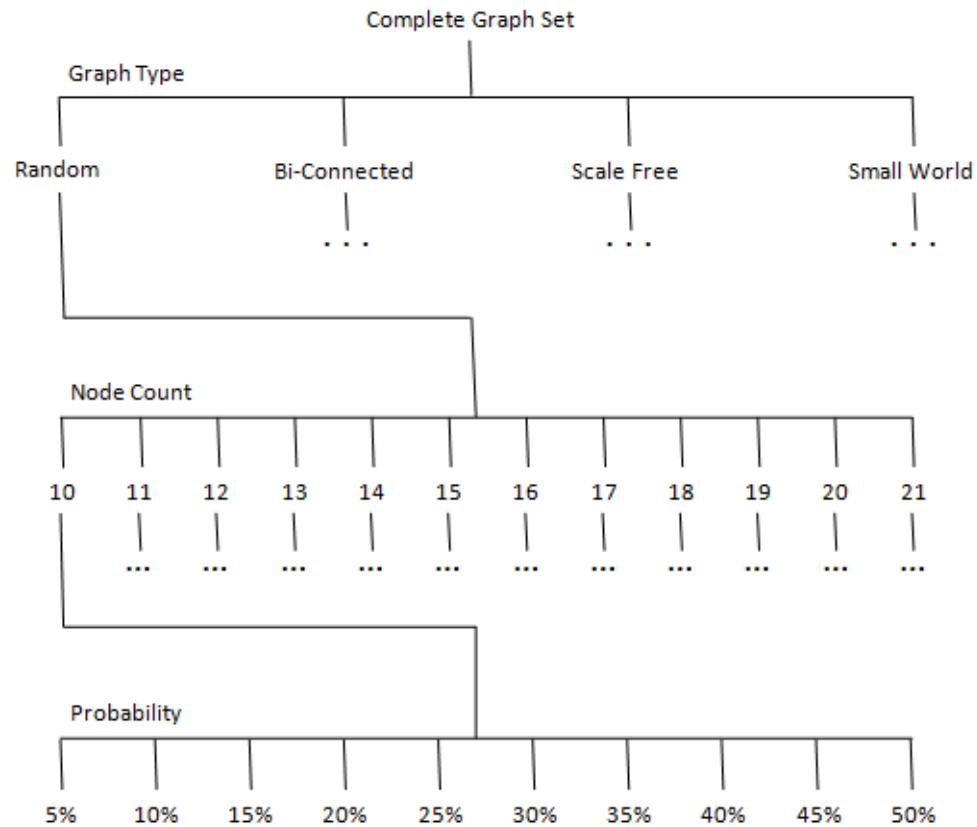


Figure 2.3: Working Directory Structure of the distinct experiments performed over the three independent variables.

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.

```

100 Generate a new graph of the specified type (Create Algorithm)
101 Calculate initial values for hardness and  $\Psi$ 
102 Set retryCount to 0
103 While retryCount < Specified Hill Climber Maximum Retries {
104     Do until new graph is valid (or until we hit a 'find neighbor' retry
        limit) {
105         Generate neighbor (Generate_Neighbor Algorithm)
106         Check if new graph is valid (Is_Valid Algorithm)
107         If not valid {
108             Undo change (revert to previous graph - increment retry
                limit)
109         }
110     }
111     Solve for monotonic value ( $\Psi$  or Hardness)
112     If monotonic value is greater than previous value {
113         Compute other (non-monotonic value)
114         Save new values for hardness and  $\Psi$  in problem File
115         Set retryCount to 0
116     }
117     else {
118         Undo change (revert to previous graph)
119         Increment retryCount
120     }
121 }
122 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.

```

100 Set target connection probability p (5 <= p <= 50)
101 For each node n where 1 <= n <= max node count - 1 {
102     For each node m where n < m <= max node count {
103         If n and m are not already connected {
104             Get random number r where 0 <= r <= 100
105             If r <= p {
106                 Connect the nodes
107             }
108         }
109     }
110 }
111 For each node (except the last node) {
112     Verify there is some connection to the next neighbor node
113     If no connection found {
114         Add an edge between the 2 nodes
115     }
116 }

```

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

```

100 Start at any node
101 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.

```

100 Do until success or max retries {
101     Select a random edge e for random node n
102     Remove the edge
103     Select 2 random nodes that aren't already connected
104     Connect the random nodes
105     If the graph is still valid {
106         Return Success
107     }
108     else {
109         Restore the original edge that was removed
110         Increment the retry count
111     }
112 }

```

Figure 2.7: Generate a Neighbor Random Graph

```

100 Set target connection probability p ( $5 \leq p \leq 50$ )
101 Create a ring network (primary links)
102 For each node n where  $1 \leq n \leq \text{max node count}$  {
103     For each node m where  $n < m \leq \text{max node count}$  {
104         If n and m are not already connected {
105             Get random number r where  $0 \leq r \leq 100$ 
106             If  $r \leq p$  {
107                 Connect the nodes (secondary link)
108             }
109         }
110     }
111 }

```

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

```
100 For each edge in the network {
101     Remove the edge
102     If all nodes are still connected {
103         The graph is valid
104     }
105     else {
106         The graph is not valid
107     }
108     Put the removed edge back in
109 }
```

Figure 2.9: Validate a Bi-Connected Graph

```
100 Do until success or max retries {
101     Select a random edge e for node n
102     Remove the edge
103     Select 2 random nodes that aren't already connected
104     Connect the random nodes
105     If the graph is still valid (still a bi-connected graph) {
106         Success
107     }
108     else {
109         Restore the original edge that was removed
110         Increment the retry count
111     }
112 }
```

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

```

100 Create a 3 nodes (scale free) graph
101 Do until we reach the target node count
102 {
103     Create a new node
104     Find the maximum degree count for any existing node (MaxDegree)
105     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 {
106         Get random number r where  $0 \leq r \leq 100$ 
107         If  $r \leq p$  {
108             Connect the nodes
109         }
110     }
111     If new node has degree = 0 {
112         Connect new node to the first node with the max degree
113     }
114 }

```

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

```

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

```

Figure 2.12: Validate a Scale Free Graph

```

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

```

Figure 2.13: Generate a Neighbor Scale Free Graph

```
100 Set target probability p (5 <= p <= 50)
101 Create a 2-regular network (primary links)
102 For each node n where 1 <= n <= max node {
103     For each node m where n < m <= max node count {
104         If n and m are not already connected {
105             Get random number r where 0 <= r <= 100
106             If r <= p {
107                 Connect the nodes (secondary link)
108             }
109         }
110     }
111 }
```

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

```
100 Start at any node
101 Traverse the graph to make sure all other nodes are reachable
102 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.

```
100 Select 2 random nodes
101 If the 2 nodes have a secondary link {
102     Remove the link
103 }
104 Select 2 random nodes
105 If the 2 nodes have no connection (primary or secondary) {
106     Add a secondary link
107 }
```

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.

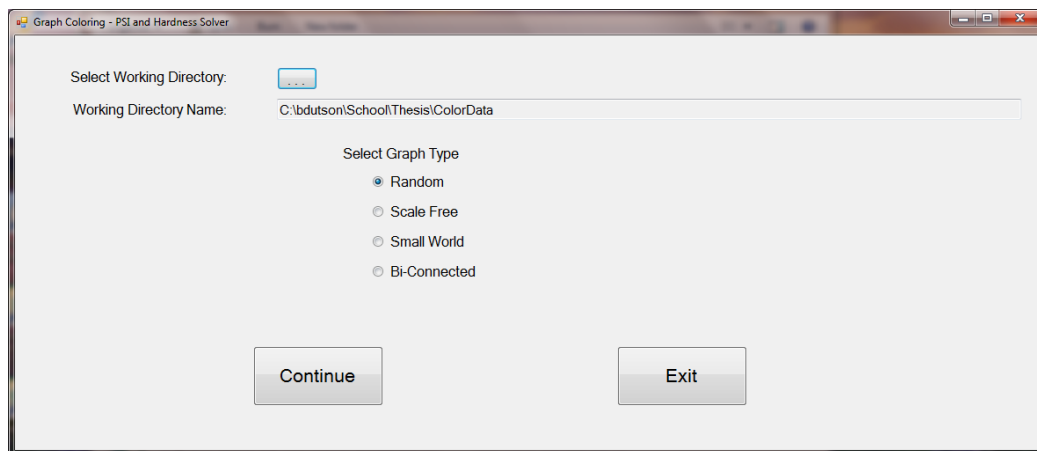


Figure 2.17: Main User Interface Screen

Random Graphs - Generate and Solve

Selected Directory: C:\bdutson\School\Thesis\ColorData

Working SubDirectory:

Select Parameters

Number of Graphs per Trial:

Beginning Node Count:

Ending Node Count:

Beginning Probability:

Ending Probability:

Hill Climber Maximum Retries:

Use Fixed Color Count:

Vary Color Count Starting At:

Generate Results

Current Node Count:

Current Probability:

Current Graph Number:

Current Graph Name:

Current Activity:

Generating Graph

Solving Graph

Monotonic Value:

Hardness

PSI

Current Color Count:

Current Retry Count:

Current Depth:

Start

Close

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.”

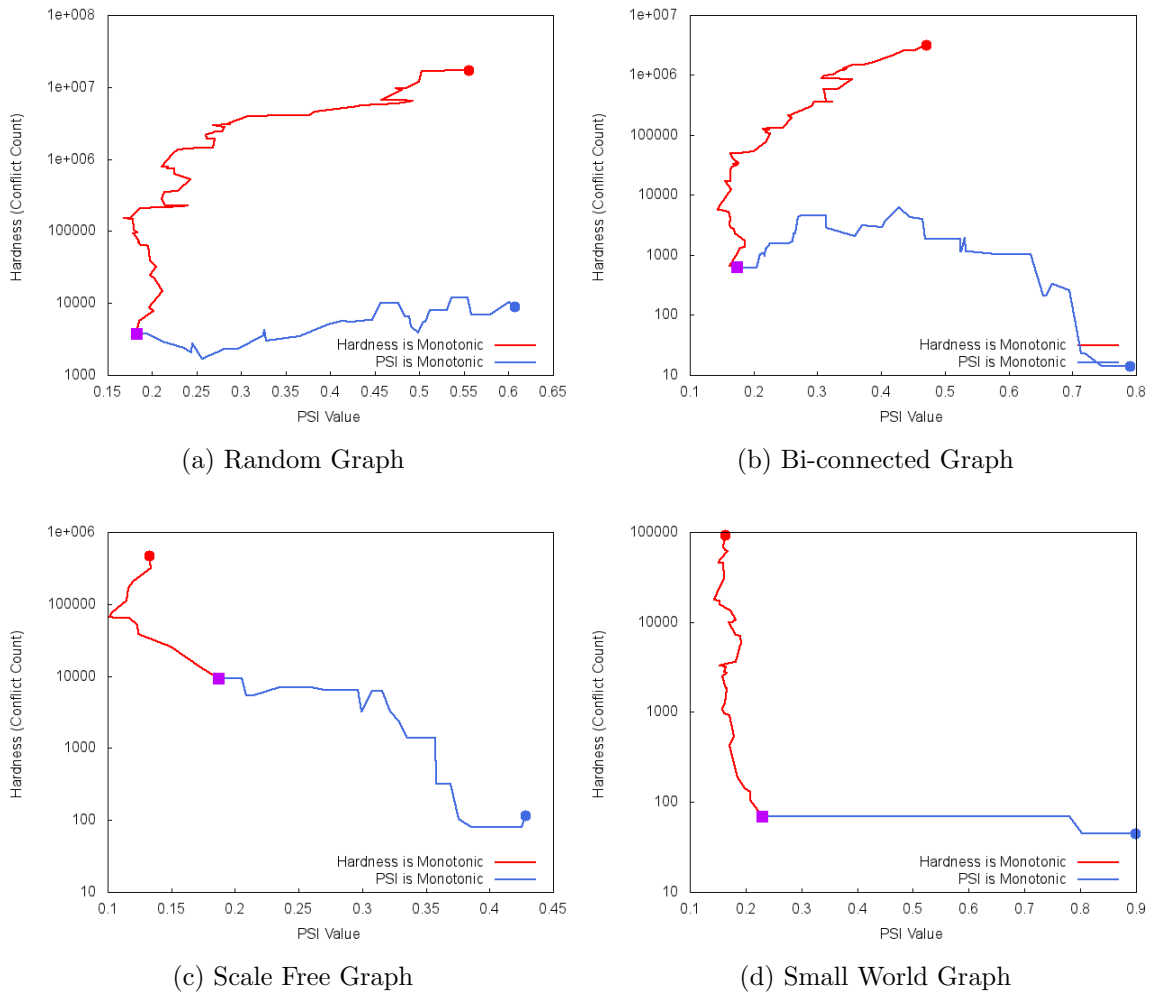


Figure 3.1: Individual Trial. Results using node count = 18, connection probability = 30 percent, trial run number 5. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Each point on the graph represents an individual graph created during optimization search. Both searches 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 there-

fore 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.

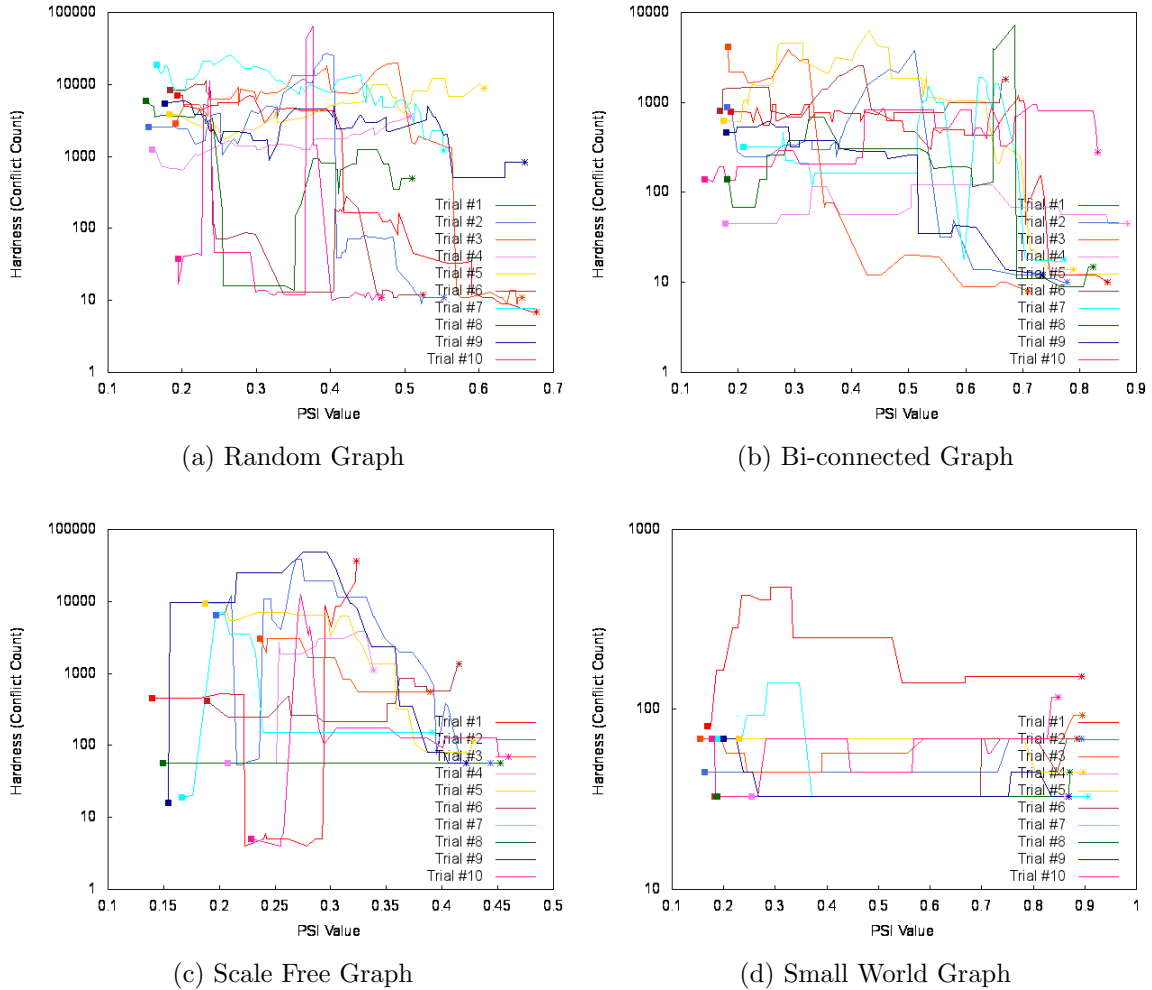


Figure 3.2: Individual Comparisons when optimizing for Ψ . Results using node count = 18, connection probability = 30 percent. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Note in each run Ψ is monotonically increasing.

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

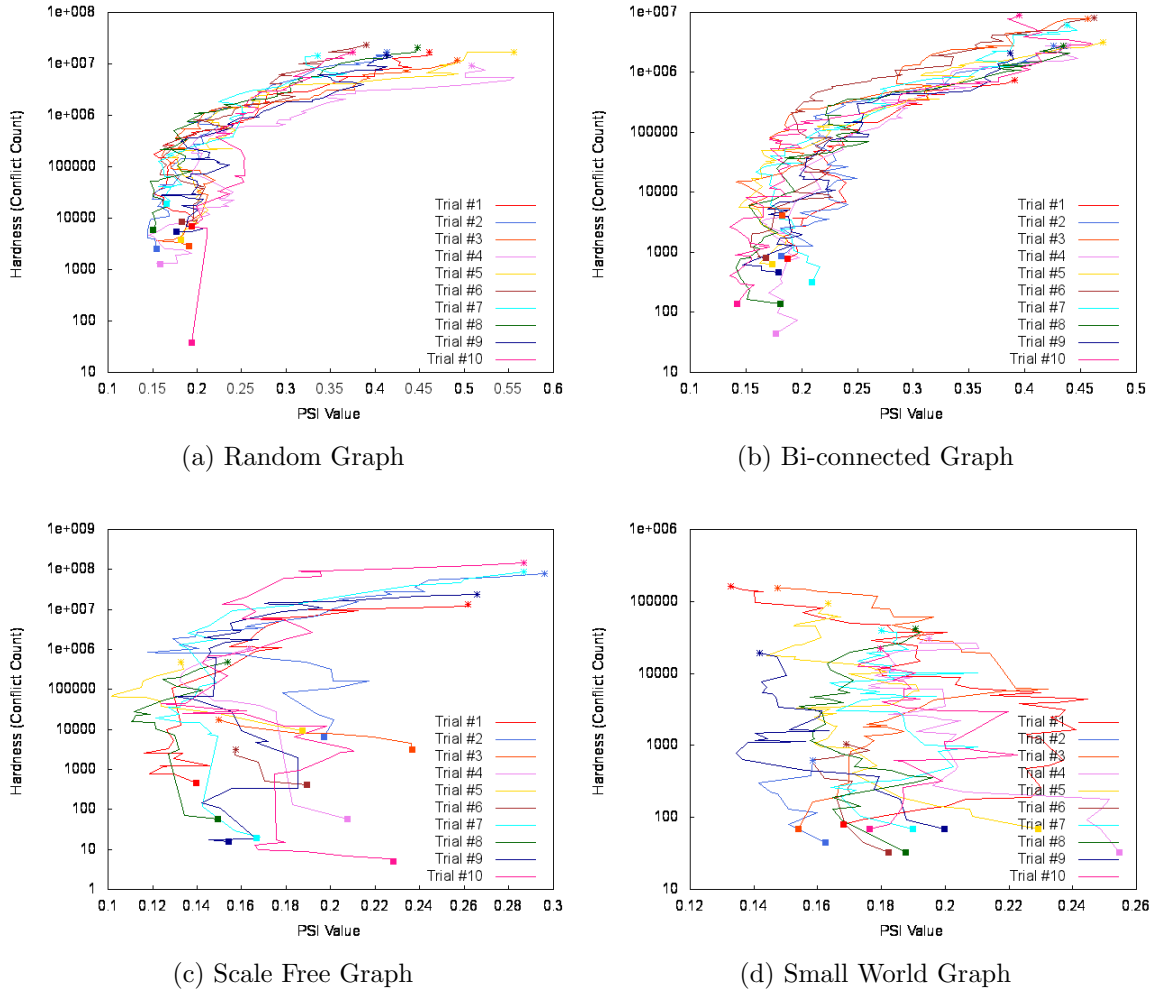
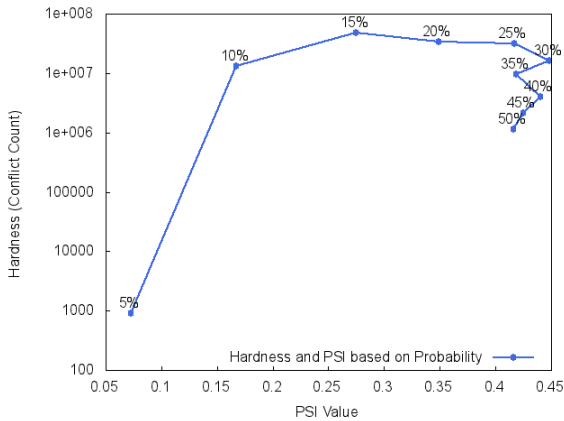
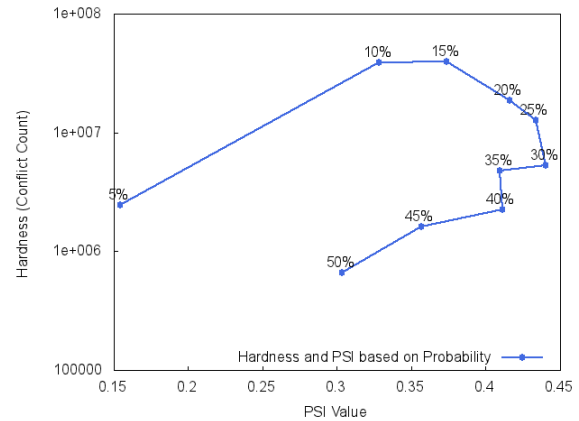


Figure 3.3: Individual Comparisons when optimizing for hardness. Results using node count = 18, connection probability = 30 percent. X axis shows Ψ on a linear scale. Y axis shows hardness on a logarithmic scale. Note in each run hardness is monotonically increasing.

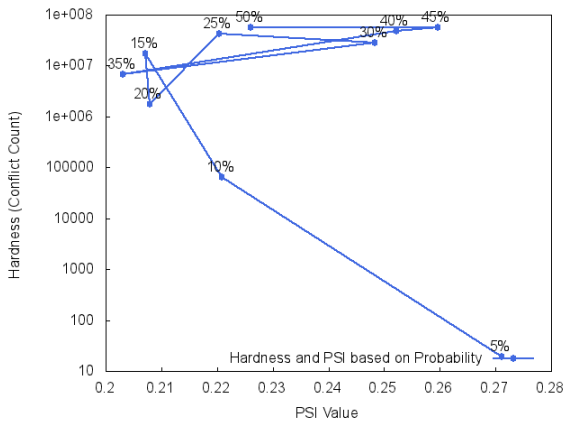
will use averages, taken from five trial runs for each combination of graph class, connection probability, and node count. By using averages, we minimize the effect of individual variation that might occur. For example, in a depth first search a single graph might coincidentally be solved with no backtracking, 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.



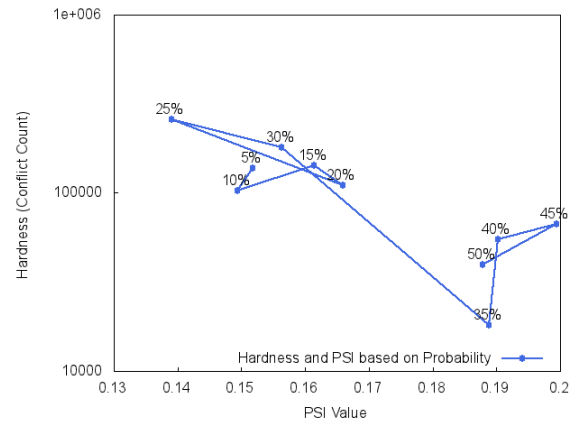
(a) Random Graph



(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.

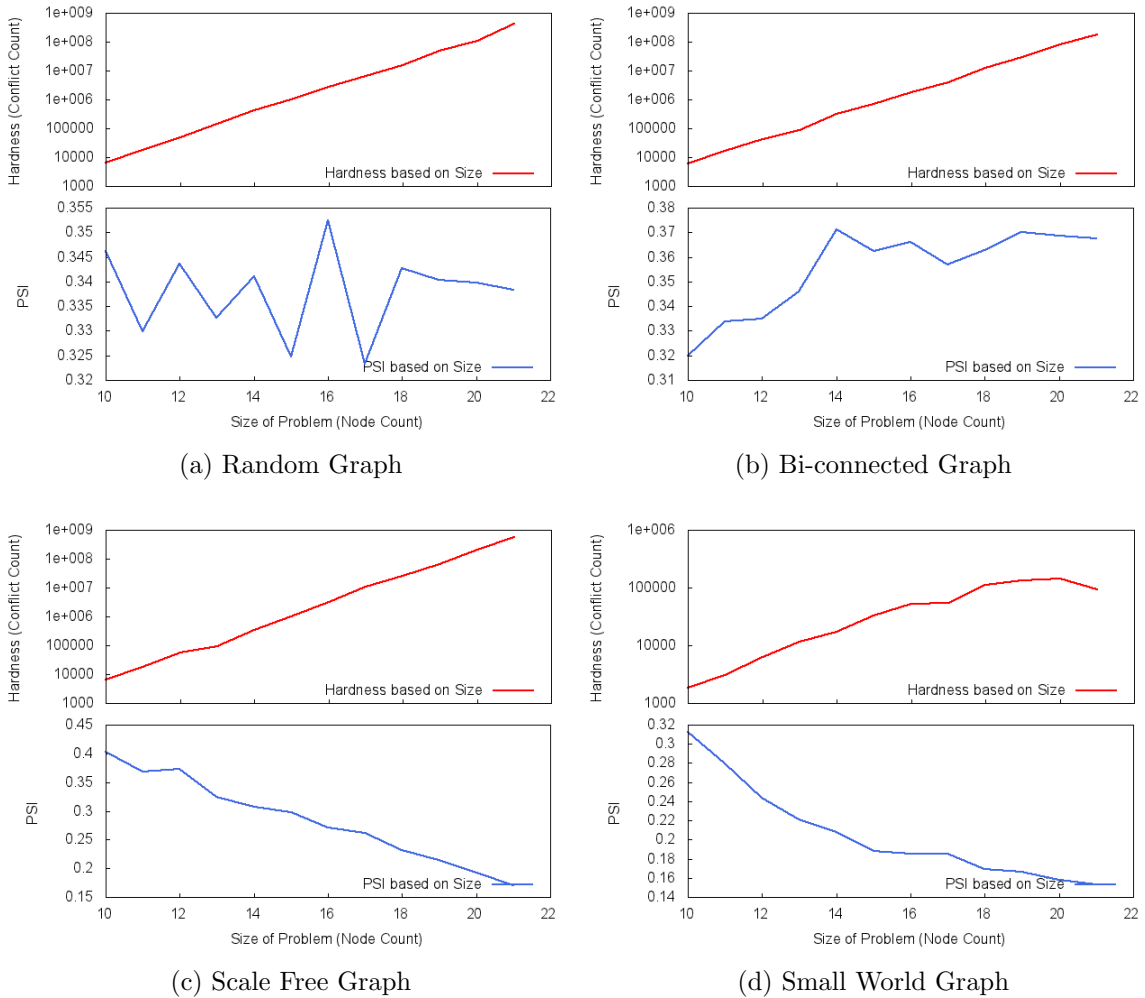


Figure 3.5: Comparing the influence of node count on hardness vs. Ψ . X axis for both plots shows node count on a linear scale. Y axis for the top plot shows hardness on a logarithmic scale. Y axis for the bottom plot shows Ψ on a linear scale.

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.

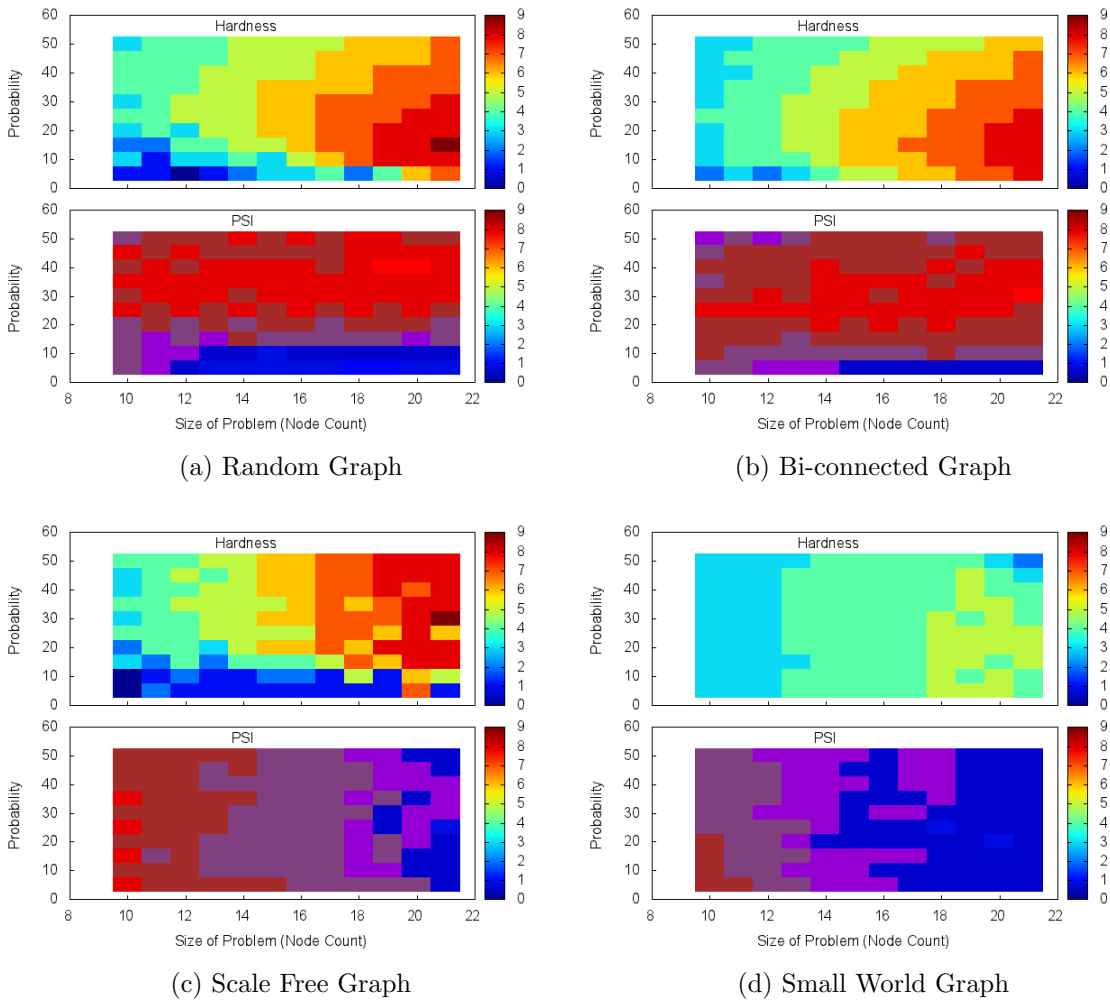


Figure 3.6: Aggregated Data. Heat plot showing the effect of connection probability and node count on both hardness and Ψ . X axis for both plots shows node count on a linear scale. Y axis for both plots shows connection probability on a linear scale. Colors in the top plot represent hardness values.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

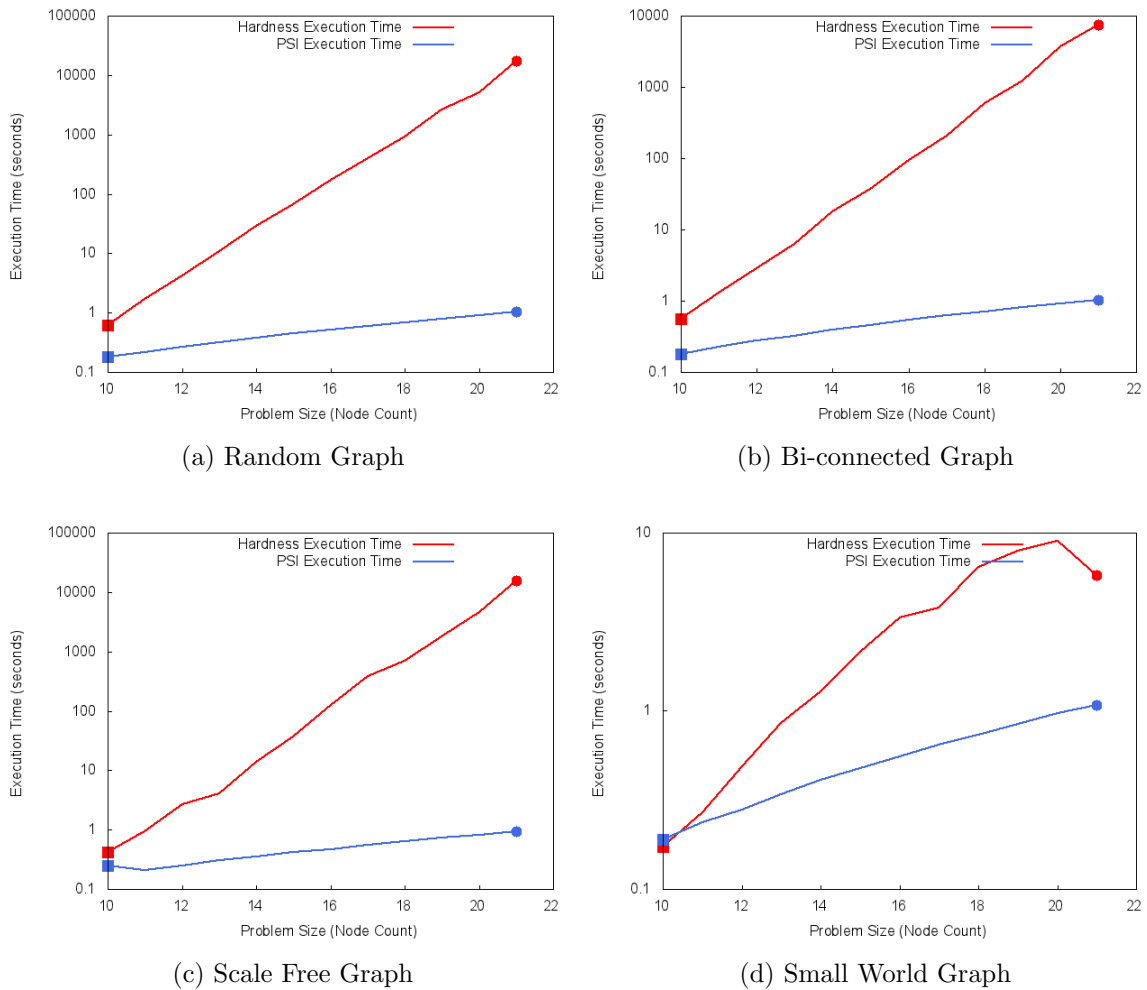


Figure 3.7: Comparison of execution time required to calculate hardness vs. time to calculate Ψ . X axis shows node count on a linear scale. Y axis shows execution time measured in seconds, on a logarithmic scale.

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 Ψ . 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 Ψ *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 astronomical 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.

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9 using System.IO;
10
11 namespace GraphandPSISolver
12 {
13     public partial class frmMain : Form
14     {
15         string myDirName;
16
17         public frmMain()
18         {
19             InitializeComponent();
20         }
21     }
```

```
22     private void btnExit_Click(object sender, EventArgs e)
23     {
24         Close();
25     }
26
27     private void btnDirectory_Click(object sender, EventArgs e)
28     {
29         string szCurDirectory;
30
31         szCurDirectory = Directory.GetCurrentDirectory();
32         if (!Directory.Exists(szCurDirectory))
33         {
34             Directory.CreateDirectory(szCurDirectory);
35         }
36         fbDirName.SelectedPath = szCurDirectory;
37         DialogResult result = fbDirName.ShowDialog();
38         if (result == DialogResult.OK)
39         {
40             myDirName = fbDirName.SelectedPath;
41             txtDirName.Text = myDirName;
42             btnContinue.Enabled = true;
43         }
44         else
45         {
46             myDirName = "";
47             txtDirName.Text = myDirName;
48             btnContinue.Enabled = false;
49         }
50     }
51
52     private void btnContinue_Click(object sender, EventArgs e)
53     {
54
55         if (rbRandom.Checked == true)
56         {
57             frmGenerate myDialog = new frmGenerate();
58             myDialog.DirectoryName = myDirName;
59             myDialog.GraphType = "Random";
60             myDialog.ShowDialog(this);
61         }
62         else if (rbScaleFree.Checked == true)
63         {
64             frmGenerate myDialog = new frmGenerate();
65             myDialog.DirectoryName = myDirName;
66             myDialog.GraphType = "ScaleFree";
67             myDialog.ShowDialog(this);
68         }
```



```

69         else if (rbSmallWorld.Checked == true)
70         {
71             frmGenerate myDialog = new frmGenerate();
72             myDialog.DirectoryName = myDirName;
73             myDialog.GraphType = "SmallWorld";
74             myDialog.ShowDialog(this);
75         }
76         else if (rbBiConnected.Checked == true)
77         {
78             frmGenerate myDialog = new frmGenerate();
79             myDialog.DirectoryName = myDirName;
80             myDialog.GraphType = "BiConnected";
81             myDialog.ShowDialog(this);
82         }
83     }
84
85     private void frmMain_Shown(object sender, EventArgs e)
86     {
87         rbRandom.Checked = true;
88     }
89 }
90 }

```

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.

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9 using System.IO;
10 using System.Threading;
11
```

```

12 namespace GraphandPSISolver
13 {
14     public partial class frmGenerate : Form
15     {
16         //-----
17         // Define constants.
18         //-----
19         // Task States
20         const int TASK_NONE = 0;
21         const int TASK_START = 1;
22         const int TASK_BEGIN = 2;
23         const int TASK_ACTIVE = 3;
24         const int TASK_PAUSED = 4;
25         const int TASK_CANCELLED = 5;
26         const int TASK_DONE = 6;
27         // Current Activity
28         const int ACT_GENERATE_GRAPH = 0;
29         const int ACT_SOLVE_GRAPH = 1;
30         // Current Monotonic Value
31         const int MON_HARDNESS = 0;
32         const int MON_PSI = 1;
33         //-----
34         // Define data elements for the class.
35         //-----
36         // Values entered on the current screen
37         private int nGraphsPerTrial = 0;
38         private int nBegNodeCount = 0;
39         private int nEndNodeCount = 0;
40         private int nBegProbability = 0;
41         private int nEndProbability = 0;
42         private int nHCRetrieves = 0;
43         private int nFixedColor = 0;
44         private int nVaryColor = 0;
45         // Values updated/used by the current screen
46         private string SubDirName = string.Empty;
47         private string FullPathName = string.Empty;
48         private static grfFile wrkFile = new grfFile();
49         private Thread WorkThread;
50         // Values passed in from the previous screen
51         private string DirName = string.Empty;
52         private string GrfTyp = string.Empty;
53         // Values for current results
54         private int nCurNodeCount = 0;
55         private int nCurProbability = 0;
56         private int nCurGraphNumber = 0;
57         private string CurGraphName = string.Empty;
58         private int nCurActivity = ACT_GENERATE_GRAPH;

```

```
59     private int nMonotonicVal = MON_HARDNESS;
60     private long nHardnessVal = 0;
61     private double fPSIVal = 0.0;
62     private int nCurColorCount = 0;
63     private int nCurRetryCount = 0;
64     private int nCurDepth = 0;
65     // Status values
66     private int nTaskState = TASK_NONE;
67
68     public frmGenerate()
69     {
70         InitializeComponent();
71     }
72
73     public string DirectoryName
74     {
75         get { return DirName; }
76         set { DirName = value; }
77     }
78
79     public string GraphType
80     {
81         get { return GrfTyp; }
82         set { GrfTyp = value; }
83     }
84
85     private void frmGenerate_Shown(object sender, EventArgs e)
86     {
87         this.Text = GrfTyp + " Graphs - Generate and Solve";
88         txtSubDirName.Text = DirName;
89         ClearInputFields();
90         ClearResultFields();
91         btnStart.Text = "Start";
92         btnStart.Enabled = false;
93         btnClose.Text = "Close";
94         btnClose.Enabled = true;
95         txtGraphsPerTrial.Focus();
96     }
97
98     private void txtGraphsPerTrial_TextChanged(object sender, EventArgs e)
99     {
100         SetStartButtonState();
101     }
102
103     private void txtBegNodeCnt_TextChanged(object sender, EventArgs e)
104     {
105         SetStartButtonState();
```

```
106     }
107
108     private void txtEndNodeCnt_TextChanged(object sender, EventArgs e)
109     {
110         SetStartButtonState();
111     }
112
113     private void txtBegProbability_TextChanged(object sender, EventArgs e)
114     {
115         SetStartButtonState();
116     }
117
118     private void txtEndProbability_TextChanged(object sender, EventArgs e)
119     {
120         SetStartButtonState();
121     }
122
123     private void txtHCRetries_TextChanged(object sender, EventArgs e)
124     {
125         SetStartButtonState();
126     }
127
128     private void SetStartButtonState()
129     {
130
131         if ((txtGraphsPerTrial.Text.Length > 0) &&
132             (txtBegNodeCnt.Text.Length > 0) &&
133             (txtEndNodeCnt.Text.Length > 0) &&
134             (txtBegProbability.Text.Length > 0) &&
135             (txtEndProbability.Text.Length > 0) &&
136             (txtHCRetries.Text.Length > 0))
137         {
138             btnStart.Enabled = true;
139         }
140         else
141         {
142             btnStart.Enabled = false;
143         }
144     }
145
146     private void btnStart_Click(object sender, EventArgs e)
147     {
148         int nRtnval = 0;
149
150         switch (nTaskState)
151         {
152             case TASK_NONE:           // Start button pressed
```

```

153         nRtnval = ValidateInputFields();
154         if (nRtnval == 0)
155         {
156             DisableInputFields();
157             btnStart.Text = "Pause";
158             btnStart.Enabled = false;
159             btnClose.Text = "Cancel";
160             wrkFile.DoneFlag = 0;
161             nTaskState = TASK_START;
162             timer1.Enabled = true;
163         }
164         break;
165         case TASK_START:           // Start button is disabled in these
                                   states
166         case TASK_BEGIN:
167             break;
168         case TASK_ACTIVE:         // Pause button pressed
169             wrkFile.PauseFlag = 1;
170             btnStart.Text = "Continue";
171             nTaskState = TASK_PAUSED;
172             break;
173         case TASK_PAUSED:        // Continue button pressed
174             wrkFile.PauseFlag = 0;
175             btnStart.Text = "Pause";
176             nTaskState = TASK_ACTIVE;
177             break;
178         case TASK_CANCELLED:     // Start button is disabled in these
                                   states
179         case TASK_DONE:
180             break;
181         default:
182             break;
183     }
184 }
185
186 private void btnClose_Click(object sender, EventArgs e)
187 {
188     switch (nTaskState)
189     {
190         case TASK_NONE:           // Close button pressed
191             Close();
192             break;
193         case TASK_START:         // Cancel button pressed
194         case TASK_BEGIN:
195         case TASK_ACTIVE:
196         case TASK_PAUSED:
197             btnStart.Enabled = false;

```

```

198         btnClose.Enabled = false;
199         nTaskState = TASK_CANCELLED;
200         break;
201     case TASK_CANCELLED:    // Cancel button is disabled in these
                             states
202     case TASK_DONE:
203         break;
204     default:
205         break;
206     }
207 }
208
209 private void timer1_Tick(object sender, EventArgs e)
210 {
211     string CalcPathName = string.Empty;
212
213     //-----
214     // Each instance when the timer fires we want to check the state
215     // of the problem being generated or solved and make any updates
216     // required.
217     //-----
218     switch (nTaskState)
219     {
220     case TASK_NONE:        // No solution in progress
221         timer1.Enabled = false;
222         break;
223     case TASK_START:
224         btnStart.Enabled = false;
225         ClearResultFields();
226         nCurNodeCount = nBegNodeCount;
227         nCurProbability = nBegProbability;
228         nCurGraphNumber = 1;
229         DisplayResults();
230         nTaskState = TASK_BEGIN;
231         break;
232     case TASK_BEGIN:
233         // Clear the result fields and initialize local variables
234         CheckDirectory();
235         txtWrkDirName.Text = SubDirName;
236         FullPathName = DirectoryName + "\\\" + SubDirName;
237         // Set initial values for the worker thread
238         wrkFile.DirectoryName = FullPathName;
239         wrkFile.FileName = "grf" + nCurGraphNumber.ToString("D4");
240         wrkFile.GraphType = GrfTyp;
241         wrkFile.RetryCount = nHCRetries;
242         wrkFile.NodeCount = nCurNodeCount;
243         wrkFile.Probability = nCurProbability;

```

```

244     wrkFile.CurrentActivity = 0;
245     wrkFile.MonotonicValue = MON_HARDNESS;
246     wrkFile.ConflictCount = 0;
247     wrkFile.PSIvalue = 0;
248     wrkFile.CurrentRetries = 0;
249     wrkFile.UseFixedColorCount = rbFixedCount.Checked;
250     if (rbFixedCount.Checked == true)
251     {
252         wrkFile.MinimumColorCount = nFixedColor;
253     }
254     else
255     {
256         wrkFile.MinimumColorCount = nVaryColor;
257     }
258     wrkFile.StatusMessage = string.Empty;
259     // Display the initial values
260     DisplayResults();
261     //Start the work thread
262     wrkFile.DoneFlag = 0;
263     wrkFile.PauseFlag = 0;
264     WorkThread = new Thread(new ThreadStart(wrkFile.
        CreateAndSolveGraph));
265     WorkThread.Name = "WorkThread";
266     WorkThread.Start();
267     btnStart.Enabled = true;
268     nTaskState = TASK_ACTIVE;
269     break;
270 case TASK_ACTIVE:
271     if ((wrkFile.DoneFlag != 0) || (WorkThread.IsAlive != true))
272     {
273         nCurGraphNumber++;
274         if (nCurGraphNumber > nGraphsPerTrial)
275         {
276             CalcPathName = DirectoryName + "\\\" + GraphType + "\\\"
                +
277                 nCurNodeCount.ToString("D4") + "\\\" +
278                 nCurProbability.ToString("D3");
279             wrkFile.CalculateLeafResults(CalcPathName, GraphType,
280                 nCurNodeCount,
                nCurProbability);
281
282             nCurGraphNumber = 1;
283             nCurProbability += 5;
284             if (nCurProbability > nEndProbability)
285             {
286                 CalcPathName = DirectoryName + "\\\" + GraphType +
                    "\\\" +
                nCurNodeCount.ToString("D4");

```



```

287         wrkFile.CalculateProbabilityResults( CalcPathName,
288             GraphType,
289             nCurNodeCount
290             );
291         nCurProbability = nBegProbability;
292         nCurNodeCount++;
293         if ( nCurNodeCount > nEndNodeCount )
294         {
295             CalcPathName = DirectoryName + "\\\" +
296                 GraphType;
297             wrkFile.CalculateNodeCountResults(
298                 CalcPathName, GraphType);
299             nTaskState = TASK_DONE;
300         }
301         else
302         {
303             nTaskState = TASK_BEGIN;
304         }
305     }
306     else
307     {
308         nTaskState = TASK_BEGIN;
309     }
310 }
311 DisplayResults();
312 break;
313 case TASK_PAUSED:
314     break;
315 case TASK_CANCELLED:
316 case TASK_DONE:
317     EnableInputFields();
318     wrkFile.DoneFlag = 1;
319     wrkFile.PauseFlag = 0;
320     btnStart.Text = "Start";
321     btnStart.Enabled = true;
322     btnClose.Text = "Close";
323     btnClose.Enabled = true;
324     nTaskState = TASK_NONE;
325     MessageBox.Show("The current run is complete");
326     break;
327 default:
328     break;
329 
```

```

330     }
331 }
332
333 private void CheckDirectory()
334 {
335
336     //-----
337     // Get the directory name at the 'type' level and make sure it
338     // exists. If it doesn't, create it.
339     //-----
340     SubDirName = GraphType;
341     FullPathName = DirectoryName + "\\\" + SubDirName;
342     if (!Directory.Exists(FullPathName))
343     {
344         Directory.CreateDirectory(FullPathName);
345     }
346     //-----
347     // Get the directory name at the 'node count' level and make sure
348     // it exists. If it doesn't, create it.
349     //-----
350     SubDirName = SubDirName + "\\\" + nCurNodeCount.ToString("D4");
351     FullPathName = DirectoryName + "\\\" + SubDirName;
352     if (!Directory.Exists(FullPathName))
353     {
354         Directory.CreateDirectory(FullPathName);
355     }
356     //-----
357     // Get the directory name at the 'probability' level and make sure
358     // it exists. If it doesn't, create it.
359     //-----
360     SubDirName = SubDirName + "\\\" + nCurProbability.ToString("D3");
361     FullPathName = DirectoryName + "\\\" + SubDirName;
362     if (!Directory.Exists(FullPathName))
363     {
364         Directory.CreateDirectory(FullPathName);
365     }
366 }
367
368 private void DisplayResults()
369 {
370     // Read the current results
371     CurGraphName = wrkFile.FileName;
372     nCurActivity = wrkFile.CurrentActivity;
373     nMonotonicVal = wrkFile.MonotonicValue;
374     nCurColorCount = wrkFile.CurrentColorCount;
375     nCurRetryCount = wrkFile.CurrentRetries;
376     nCurDepth = wrkFile.CurrentDepth;

```

```
377         nHardnessVal = wrkFile.MaxColorConflict;
378         fPSIVal = wrkFile.MaxPSIvalue;
379         // Display the results
380         txtCurNodeCnt.Text = nCurNodeCount.ToString();
381         txtCurProbability.Text = nCurProbability.ToString();
382         txtCurGraphNumber.Text = nCurGraphNumber.ToString();
383         txtCurGraphName.Text = CurGraphName;
384         if (nCurActivity == ACT_GENERATE_GRAPH)
385         {
386             rbGenerateGraph.Checked = true;
387         }
388         else
389         {
390             rbSolveGraph.Checked = true;
391         }
392         if (nMonotonicVal == MON_HARDNESS)
393         {
394             rbHardness.Checked = true;
395         }
396         else
397         {
398             rbPSI.Checked = true;
399         }
400         txtHardnessVal.Text = nHardnessVal.ToString();
401         txtPSIVal.Text = fPSIVal.ToString();
402         txtCurColorCnt.Text = nCurColorCount.ToString();
403         txtCurRetryCnt.Text = nCurRetryCount.ToString();
404         txtCurDepth.Text = nCurDepth.ToString();
405         txtMessage.Text = wrkFile.StatusMessage;
406     }
407
408     private void ClearResultFields()
409     {
410         nCurActivity = ACT_GENERATE_GRAPH;
411         rbGenerateGraph.Checked = true;
412         nMonotonicVal = MON_HARDNESS;
413         rbHardness.Checked = true;
414         nHardnessVal = 0;
415         txtHardnessVal.Text = "";
416         fPSIVal = 0.0;
417         txtPSIVal.Text = "";
418         nCurColorCount = 0;
419         txtCurColorCnt.Text = "";
420         nCurRetryCount = 0;
421         txtCurRetryCnt.Text = "";
422         nCurDepth = 0;
423         txtCurDepth.Text = "";
```

```
424     }
425
426     private void ClearInputFields()
427     {
428         txtGraphsPerTrial.Text = "";
429         txtBegNodeCnt.Text = "";
430         txtEndNodeCnt.Text = "";
431         txtBegProbability.Text = "";
432         txtEndProbability.Text = "";
433         txtHCRetries.Text = "";
434         rbFixedCount.Checked = true;
435         txtFixedCount.Text = "3";
436         txtVaryCount.Text = "2";
437     }
438
439     private void EnableInputFields()
440     {
441         txtGraphsPerTrial.Enabled = true;
442         txtBegNodeCnt.Enabled = true;
443         txtEndNodeCnt.Enabled = true;
444         txtBegProbability.Enabled = true;
445         txtEndProbability.Enabled = true;
446         txtHCRetries.Enabled = true;
447         rbFixedCount.Enabled = true;
448         rbVaryCount.Enabled = true;
449         if (rbFixedCount.Checked == true)
450         {
451             txtFixedCount.Enabled = true;
452         }
453         if (rbVaryCount.Checked == true)
454         {
455             txtVaryCount.Enabled = true;
456         }
457     }
458
459     private void DisableInputFields()
460     {
461         txtGraphsPerTrial.Enabled = false;
462         txtBegNodeCnt.Enabled = false;
463         txtEndNodeCnt.Enabled = false;
464         txtBegProbability.Enabled = false;
465         txtEndProbability.Enabled = false;
466         txtHCRetries.Enabled = false;
467         rbFixedCount.Enabled = false;
468         rbVaryCount.Enabled = false;
469         txtFixedCount.Enabled = false;
470         txtVaryCount.Enabled = false;
```

```

471     }
472
473     private int ValidateInputFields()
474     {
475         int nRtnval = 0;
476
477         //-----
478         // Limit the number of graphs per trial to 100.
479         //-----
480         if (nRtnval == 0)
481         {
482             nGraphsPerTrial = Convert.ToInt32(txtGraphsPerTrial.Text);
483             if (nGraphsPerTrial < 1)
484             {
485                 MessageBox.Show("The number of graphs must be greater than 0"
486                     );
487                 txtGraphsPerTrial.Focus();
488                 nRtnval = -1;
489             }
490             else if (nGraphsPerTrial > 100)
491             {
492                 MessageBox.Show("The maximum number of graphs allowed is 100"
493                     );
494                 txtGraphsPerTrial.Focus();
495                 nRtnval = -1;
496             }
497         }
498         //-----
499         // Validate the node count. We are going to limit it to
500         // 1000.
501         //-----
502         if (nRtnval == 0)
503         {
504             nBegNodeCount = Convert.ToInt32(txtBegNodeCnt.Text);
505             nEndNodeCount = Convert.ToInt32(txtEndNodeCnt.Text);
506             if (nBegNodeCount < 10)
507             {
508                 MessageBox.Show("The beginning node count must be >= 10");
509                 txtBegNodeCnt.Focus();
510                 nRtnval = -1;
511             }
512             else if (nEndNodeCount > 1000)
513             {
514                 MessageBox.Show("The ending node count must be <= 1000");
515                 txtEndNodeCnt.Focus();
516                 nRtnval = -1;
517             }
518         }
519     }

```

```

516         else if (nBegNodeCount > nEndNodeCount)
517         {
518             MessageBox.Show("The ending node count must be >= to the
                    beginning node count");
519             txtEndNodeCnt.Focus();
520             nRtnval = -1;
521         }
522     }
523     //-----
524     // Validate the probability. It must be in the range from
525     // 2 to 50.
526     //-----
527     if (nRtnval == 0)
528     {
529         nBegProbability = Convert.ToInt32(txtBegProbability.Text);
530         nEndProbability = Convert.ToInt32(txtEndProbability.Text);
531         if (nBegProbability < 2)
532         {
533             MessageBox.Show("The beginning probability must be >= 2");
534             txtBegProbability.Focus();
535             nRtnval = -1;
536         }
537         else if (nEndProbability > 50)
538         {
539             MessageBox.Show("The ending probability must be <= 50");
540             txtEndProbability.Focus();
541             nRtnval = -1;
542         }
543         else if (nBegProbability > nEndProbability)
544         {
545             MessageBox.Show("The ending probability must be >= to the
                    beginning probability");
546             txtEndProbability.Focus();
547             nRtnval = -1;
548         }
549     }
550     //-----
551     // Validate the Hill Climber Retry count. It must be in
552     // the range from 10 to 10000.
553     //-----
554     if (nRtnval == 0)
555     {
556         nHCRetries = Convert.ToInt32(txtHCRetries.Text);
557         if (nHCRetries < 10)
558         {
559             MessageBox.Show("The number of retries must be at least 10");
560             txtHCRetries.Focus();

```

```
561         nRtnval = -1;
562     }
563     else if (nHCRetries > 10000)
564     {
565         MessageBox.Show("The maximum number of retries allowed is
                    10,000");
566         txtHCRetries.Focus();
567         nRtnval = -1;
568     }
569 }
570 //-----
571 // Validate the fixed color count.
572 //-----
573 if (nRtnval == 0)
574 {
575     nFixedColor = Convert.ToInt32(txtFixedCount.Text);
576     if (nFixedColor < 2)
577     {
578         MessageBox.Show("The number of fixed colors must be at least
                    2");
579         txtFixedCount.Focus();
580         nRtnval = -1;
581     }
582     else if (nFixedColor > 100)
583     {
584         MessageBox.Show("The maximum number of fixed colors allowed
                    is 100");
585         txtFixedCount.Focus();
586         nRtnval = -1;
587     }
588 }
589 //-----
590 // Validate the variable color count.
591 //-----
592 if (nRtnval == 0)
593 {
594     nVaryColor = Convert.ToInt32(txtVaryCount.Text);
595     if (nVaryColor < 2)
596     {
597         MessageBox.Show("The number of variable colors must be at
                    least 2");
598         txtVaryCount.Focus();
599         nRtnval = -1;
600     }
601     else if (nVaryColor > 100)
602     {
```

```

603             MessageBox.Show("The maximum number of variable colors
                        allowed is 100");
604             txtVaryCount.Focus();
605             nRtnval = -1;
606         }
607     }
608     return nRtnval;
609 }
610 }
611 }

```

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.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.IO;
6 using System.Diagnostics;
7 using System.Threading;
8
9 namespace GraphandPSISolver
10 {
11     class grfFile
12     {
13         //-----
14         // Define constants.
15         //-----
16         // Graph types
17         const int RANDOM_GRAPH = 1;
18         const int SCALE_FREE_GRAPH = 2;
19         const int SMALL_WORLD_GRAPH = 3;
20         const int BI_CONNECTED_GRAPH = 4;
21         // Misc defines
22         const int MAX_NODES = 1000;
23         const int REPLACE_EDGE = 1;
24         const int MAX_NEIGHBOR_RETRY = 1000;
25         // Current Activity
26         const int ACT_GENERATE_GRAPH = 0;

```



```

27     const int ACT_SOLVE_GRAPH = 1;
28     // Monotonic values
29     const int MON_HARDNESS = 0;
30     const int MON_PSI = 1;
31     //-----
32     // Define data elements for the class.
33     //-----
34     private int nDoneFlag = 0;
35     private int nPauseFlag = 0;
36     private int nNodes = 0;
37     private int nProbability = 0;
38     private int nStep = 0;
39     private int nRetries = 0;
40     private int nRetryCount = 0;
41     private int nActivity = 0;
42     private int nMonotonic = MON_HARDNESS;
43     private int nGraphType = 0;
44     private int nMinColorCount = 0;
45     private int nCurDepth = 0;
46     private int nCurColorCount = 0;
47     private bool bColorIsSolvable = false;
48     private bool bFixedColorCount = true;
49     private long nConflictCount = 0;
50     private long nMaxColorConflict = 0;
51     private double fMaxPSIvalue = 0;
52     private double PSICALCULATEDVALUE = 0.0;
53     private double fHardnessTime = 0.0;
54     private double fPSITime = 0.0;
55     private string myDirName = string.Empty;
56     private string myPrefix = string.Empty;
57     private string myFileName = string.Empty;
58     private string myActivity = string.Empty;
59     private string myGraphType = string.Empty;
60     private string myStatus = string.Empty;
61     private short[,] nodeArray = new short[MAX_NODES, MAX_NODES];
62     private IList<NodeElement> nodeList = new List<NodeElement>();
63     private IList<HillClimbResult> myHCResults = new List<HillClimbResult>();
64     private IList<RestorePoint> restoreList = new List<RestorePoint>();
65     private IList<ResultsData> resultsData = new List<ResultsData>();
66     private AveragesData averagesData = new AveragesData();
67     private Random myRand = new Random((int)DateTime.Now.Ticks);
68     private Stopwatch sw = new Stopwatch();
69     static StreamWriter fileWriter;
70     static FileStream output;
71     //-----
72     // Define methods for the class.
73     //-----

```

```
74 //*****
75 // Constructor
76 //*****
77 public grfFile()
78 {
79 }
80
81 //*****
82 // Get and Set methods.
83 //*****
84 public int DoneFlag
85 {
86     get { return nDoneFlag; }
87     set { nDoneFlag = value; }
88 }
89
90 public int PauseFlag
91 {
92     get { return nPauseFlag; }
93     set { nPauseFlag = value; }
94 }
95
96 public string DirectoryName
97 {
98     get { return myDirName; }
99     set { myDirName = value; }
100 }
101
102 public string FileName
103 {
104     get { return myFileName; }
105     set { myFileName = value; }
106 }
107
108 public string GraphType
109 {
110     get { return myGraphType; }
111     set { myGraphType = value; }
112 }
113
114 public int NodeCount
115 {
116     get { return nNodes; }
117     set { nNodes = value; }
118 }
119
120 public int Probability
```

```
121     {
122         get { return nProbability; }
123         set { nProbability = value; }
124     }
125
126     public string PrefixName
127     {
128         get { return myPrefix; }
129         set { myPrefix = value; }
130     }
131
132     public double PSIvalue
133     {
134         get { return PSICALCULATEDVALUE; }
135         set { PSICALCULATEDVALUE = value; }
136     }
137
138     public int PSISTEP
139     {
140         get { return nStep; }
141         set { nStep = value; }
142     }
143
144     public bool ColorSolvable
145     {
146         get { return bColorIsSolvable; }
147         set { bColorIsSolvable = value; }
148     }
149
150     public long ConflictCount
151     {
152         get { return nConflictCount; }
153         set { nConflictCount = value; }
154     }
155
156     public double MaxPSIvalue
157     {
158         get { return fMaxPSIvalue; }
159         set { fMaxPSIvalue = value; }
160     }
161
162     public long MaxColorConflict
163     {
164         get { return nMaxColorConflict; }
165         set { nMaxColorConflict = value; }
166     }
167
```

```
168     public int RetryCount
169     {
170         get { return nRetries; }
171         set { nRetries = value; }
172     }
173
174     public int CurrentRetries
175     {
176         get { return nRetryCount; }
177         set { nRetryCount = value; }
178     }
179
180     public int CurrentActivity
181     {
182         get { return nActivity; }
183         set { nActivity = value; }
184     }
185
186     public int MonotonicValue
187     {
188         get { return nMonotonic; }
189         set { nMonotonic = value; }
190     }
191
192     public bool UseFixedColorCount
193     {
194         get { return bFixedColorCount; }
195         set { bFixedColorCount = value; }
196     }
197
198     public int MinimumColorCount
199     {
200         get { return nMinColorCount; }
201         set { nMinColorCount = value; }
202     }
203
204     public string StatusMessage
205     {
206         get { return myStatus; }
207         set { myStatus = value; }
208     }
209
210     public int CurrentColorCount
211     {
212         get { return nCurColorCount; }
213         set { nCurColorCount = value; }
214     }
```

```

215
216     public int CurrentDepth
217     {
218         get { return nCurDepth; }
219         set { nCurDepth = value; }
220     }
221
222     //*****
223     // Main worker thread.
224     //*****
225     public void CreateAndSolveGraph()
226     {
227         int nRtnval = 0;
228         int i;
229         long nCurColorConflict = 0;
230         double fCurPSIvalue = 0.0;
231
232         CurrentActivity = ACT_GENERATE_GRAPH;
233         //-----
234         // Determine the type of graph we are working with.
235         //-----
236         if (nRtnval == 0)
237         {
238             myStatus = "Determine the graph type";
239             if (myGraphType.Equals("Random"))
240             {
241                 nGraphType = RANDOM_GRAPH;
242             }
243             else if (myGraphType.Equals("ScaleFree"))
244             {
245                 nGraphType = SCALE_FREE_GRAPH;
246             }
247             else if (myGraphType.Equals("SmallWorld"))
248             {
249                 nGraphType = SMALL_WORLD_GRAPH;
250             }
251             else if (myGraphType.Equals("BiConnected"))
252             {
253                 nGraphType = BI_CONNECTED_GRAPH;
254             }
255             else
256             {
257                 myStatus = "Unrecognized graph type = " + myGraphType;
258                 nRtnval = -1;
259             }
260         }
261

```

```

262 //-----
263 // Clear the node list so we start out empty. Add the correct
264 // number of nodes to our list. Each node will be initialized
265 // with an ID (which is really the offset) and node color of
266 // 0, which means we haven't colored it yet.
267 //-----
268 if (nRtnval == 0)
269 {
270     myStatus = "Create and Initialize (" + nNodes.ToString() + ")
                nodes";
271     foreach (NodeElement ne in nodeList)
272     {
273         ne.ClearEdgeList();
274     }
275     nodeList.Clear();
276     for (i = 0; i < nNodes; i++)
277     {
278         nodeList.Add(new NodeElement(i, 0));
279     }
280 }
281 //-----
282 // Generate a new graph of the specified type. Save the new
283 // graph to a file.
284 //-----
285 if (nRtnval == 0)
286 {
287     myStatus = "Generate the new graph and save to a file";
288     switch (nGraphType)
289     {
290     case SMALL_WORLD_GRAPH:
291         CreateSmallWorldGraph();
292         break;
293     case SCALE_FREE_GRAPH:
294         CreateScaleFreeGraph();
295         break;
296     case BI_CONNECTED_GRAPH:
297         CreateBiConnectedGraph();
298         break;
299     case RANDOM_GRAPH:
300         CreateRandomGraph();
301         break;
302     default:
303         nGraphType = RANDOM_GRAPH;
304         CreateRandomGraph();
305         break;
306     }
307     SaveGraphToFile();

```

```

308     }
309
310     //-----
311     // Initialize our working variables. Enter a loop where we
312     // try to maximize the hardness value by generating neighbors
313     // to the graph and calculating for hardness. Continue in the
314     // loop until we exceed a retry limit without finding any
315     // improvement in the hardness value.
316     //-----
317     nMonotonic = MON_HARDNESS;
318     myHCResults.Clear();           // Clear any previous
        results
319     nRetryCount = 0;
320     nMaxColorConflict = CalculateHardness(); // Initial hardness value
321     fMaxPSIvalue = CalculatePSI();         // Initial PSI value
322     myHCResults.Add(new HillClimbResult(nCurColorCount, bColorIsSolvable,
        nMaxColorConflict,
323                                     fHardnessTime, fMaxPSIvalue,
        fPSITime));
324     CurrentActivity = ACT_SOLVE_GRAPH;
325     //-----
326     // With our initial values set, we are ready to go into a loop
327     // where we try to first maximize the hardness value, and next
328     // try to maximize the PSI value. As we find higher values we
329     // will save the results in the hill climber results list.
330     // In this first loop hardness is the monotonic value.
331     //-----
332     while ((nRetryCount < nRetries) && (nDoneFlag == 0) && (nRtnval == 0)
        )
333     {
334         if (nPauseFlag != 0)
335         {
336             Thread.Sleep(100);
337         }
338         else
339         {
340             //-----
341             // Find a random neighbor.
342             //-----
343             myStatus = "Find a random neighbor (hardness is monotonic)";
344             switch (nGraphType)
345             {
346                 case SMALL_WORLD_GRAPH:
347                     GenerateSmallWorldNeighbor();
348                     break;
349                 case SCALE_FREE_GRAPH:
350                     GenerateScaleFreeNeighbor();

```

```

351         break;
352     case BI_CONNECTED_GRAPH:
353         GenerateBiConnectedNeighbor();
354         break;
355     case RANDOM_GRAPH:
356     default:
357         GenerateRandomNeighbor();
358         break;
359     }
360     //-----
361     // Solve for hardness. If the hardness value is
362     // larger than the current max, this one becomes our
363     // new max. Otherwise, undo the last neighbor change
364     // and increment the retry value.
365     //-----
366     myStatus = "Solve for hardness (hardness is monotonic)";
367     nCurColorConflict = CalculateHardness();
368     if (nCurColorConflict > nMaxColorConflict)
369     {
370         myStatus = "Solve for PSI (hardness is monotonic)";
371         fCurPSIvalue = CalculatePSI();
372         nMaxColorConflict = nCurColorConflict;
373         fMaxPSIvalue = fCurPSIvalue;
374         myHCResults.Add(new HillClimbResult(nCurColorCount,
375                                             bColorIsSolvable, nMaxColorConflict,
376                                             fHardnesTime,
377                                             fMaxPSIvalue,
378                                             fPSITime));
379
380         nRetryCount = 0;
381     }
382     else
383     {
384         RestoreNeighbor();
385         nRetryCount++;
386     }
387 }
388 SaveHCResults(MON_HARDNESS);
389 //-----
390 // Re-load the original graph and do the second loop where
391 // PSI is the monotonic value.
392 //-----
393 nMonotonic = MON_PSI;
394 LoadGraphFromFile();
395 myHCResults.Clear(); // Clear any previous
396     results
397 nRetryCount = 0;

```



```

394         nMaxColorConflict = CalculateHardness(); // Initial hardness value
395         fMaxPSIvalue = CalculatePSI();           // Initial PSI value
396         myHCResults.Add(new HillClimbResult(nCurColorCount, bColorIsSolvable,
                                           nMaxColorConflict,
397                                           fHardnessTime, fMaxPSIvalue,
                                           fPSITime));
398     while ((nRetryCount < nRetries) && (nDoneFlag == 0) && (nRtnval == 0)
399           )
400     {
401         if (nPauseFlag != 0)
402         {
403             Thread.Sleep(100);
404         }
405         else
406         {
407             //-----
408             // Find a random neighbor.
409             //-----
410             myStatus = "Find a random neighbor (PSI is monotonic)";
411             switch (nGraphType)
412             {
413                 case SMALL_WORLD_GRAPH:
414                     GenerateSmallWorldNeighbor();
415                     break;
416                 case SCALE_FREE_GRAPH:
417                     GenerateScaleFreeNeighbor();
418                     break;
419                 case BI_CONNECTED_GRAPH:
420                     GenerateBiConnectedNeighbor();
421                     break;
422                 case RANDOM_GRAPH:
423                     default:
424                         GenerateRandomNeighbor();
425                         break;
426             }
427             //-----
428             // Solve for PSI. If the PSI value is larger than the
429             // current max, this one becomes our new max.
430             // Otherwise, undo the last neighbor change and
431             // increment the retry value.
432             //-----
433             myStatus = "Solve for PSI (PSI is monotonic)";
434             fCurPSIvalue = CalculatePSI();
435             if (fCurPSIvalue > fMaxPSIvalue)
436             {
437                 myStatus = "Solve for hardness (PSI is monotonic)";
438                 nCurColorConflict = CalculateHardness();

```

```

438         fMaxPSIvalue = fCurPSIvalue;
439         nMaxColorConflict = nCurColorConflict;
440         myHCResults.Add(new HillClimbResult(nCurColorCount,
441             bColorIsSolvable, nMaxColorConflict,
442             fHardnesTime,
443             fMaxPSIvalue,
444             fPSITime));
445
446         nRetryCount = 0;
447     }
448     else
449     {
450         RestoreNeighbor();
451         nRetryCount++;
452     }
453 }
454 SaveHCResults(MON_PSI);
455 }
456
457 public long CalculateHardness()
458 {
459     long nHardness = 0;
460     bool bDone = false;
461
462     //-----
463     // Solve for color based on current settings. The end result
464     // should be the hardness value (which is really the conflict
465     // count), whether the problem was solvable or not, and how
466     // many ticks it took to solve. We ignore the pause flag at
467     // this level because it invalidates our timing.
468     //-----
469     bDone = false;
470     nCurColorCount = nMinColorCount;
471     sw.Reset();
472     sw.Start();
473     while ((bDone == false) && (nDoneFlag == 0))
474     {
475         SolveForColor(nCurColorCount);
476         if (bColorIsSolvable == true)
477         {
478             bDone = true;
479         }
480         else
481         {
482             if (bFixedColorCount == false)
483             {
484                 nCurColorCount++;

```

```

482         }
483         else
484         {
485             bDone = true;
486         }
487     }
488 }
489     nHardness = nConflictCount;
490     sw.Stop();
491     fHardnessTime = sw.Elapsed.TotalMilliseconds;
492     return (nHardness);
493 }
494
495 public double CalculatePSI()
496 {
497     double fPSI = 0.0;
498
499     //-----
500     // Convert the node list into an array, then solve for PSI.
501     // Keep track of how many ticks it took to solve. We ignore
502     // the pause flag at this level because it invalidates our
503     // timing.
504     //-----
505     CopyGraphToArray();
506     sw.Reset();
507     sw.Start();
508     SolveForPSI();
509     fPSI = PSICALCULATEDVALUE;
510     sw.Stop();
511     fPSITime = sw.Elapsed.TotalMilliseconds;
512     return (fPSI);
513 }
514
515 //*****
516 // Graph Creation Methods.
517 //*****
518 public void CreateRandomGraph()
519 {
520     int i;
521     int j;
522     int myProbability;
523
524     //-----
525     // Loop through every combination of 2 nodes in the graph. For
526     // each combination determine if they should be connected based
527     // on the current probability. If yes, connect the nodes.
528     //-----

```

```

529     for (i = 0; i < (this.NodeCount - 1); i++)
530     {
531         for (j = i + 1; j < this.NodeCount; j++)
532         {
533             if (nodeList[i].IsConnected(j) == false)
534             {
535                 myProbability = myRand.Next(0, 101);
536                 if (myProbability <= this.Probability)
537                 {
538                     nodeList[i].AddEdge(j);
539                     nodeList[j].AddEdge(i);
540                 }
541             }
542         }
543     }
544     //-----
545     // For each node except the last one, make sure there is some
546     // connection between that node and the next one in the list.
547     // If there is no connection, go ahead and add an edge to
548     // connect them.
549     //-----
550     for (i = 0; i < (this.NodeCount - 1); i++)
551     {
552         if (AreTwoNodesConnected(i, i + 1) == false)
553         {
554             nodeList[i].AddEdge(i + 1);
555             nodeList[i + 1].AddEdge(i);
556         }
557     }
558 }
559
560 public void CreateSmallWorldGraph()
561 {
562     int i;
563     int j;
564     int nNextNode;
565     int myProbability;
566
567     //-----
568     // Start by creating a 2-regular graph. This means each node
569     // will be connected to the nodes that are one and two
570     // sequential positions away from it.
571     //-----
572     for (i = 0; i < this.NodeCount; i++)
573     {
574         nNextNode = FindPrimaryNode(i);
575         if (nodeList[i].IsConnected(nNextNode) == false)

```

```

576         {
577             nodeList[i].AddEdge(nNextNode);
578             nodeList[nNextNode].AddEdge(i);
579         }
580         nNextNode = FindSecondaryNode(i);
581         if (nodeList[i].IsConnected(nNextNode) == false)
582         {
583             nodeList[i].AddEdge(nNextNode);
584             nodeList[nNextNode].AddEdge(i);
585         }
586     }
587     //-----
588     // Next we will add some other random edges to the graph. We
589     // do this by looping through every combination of 2 nodes and
590     // determine if they should be connected based on the current
591     // probability.
592     //-----
593     for (i = 0; i < (this.NodeCount - 1); i++)
594     {
595         for (j = i + 1; j < this.NodeCount; j++)
596         {
597             if (nodeList[i].IsConnected(j) == false)
598             {
599                 myProbability = myRand.Next(0, 101);
600                 if (myProbability <= this.Probability)
601                 {
602                     nodeList[i].AddEdge(j);
603                     nodeList[j].AddEdge(i);
604                 }
605             }
606         }
607     }
608 }
609
610 public void CreateScaleFreeGraph()
611 {
612     int i;
613
614     //-----
615     // Create a complete graph with the first 3 nodes.
616     //-----
617     nodeList[0].AddEdge(1);
618     nodeList[0].AddEdge(2);
619     nodeList[1].AddEdge(0);
620     nodeList[1].AddEdge(2);
621     nodeList[2].AddEdge(0);
622     nodeList[2].AddEdge(1);

```

```

623 //-----
624 // For each of the remaining nodes, we will add edges based
625 // on a probability calculation that favors those existing
626 // nodes with the highest degree count.
627 //-----
628 for (i = 3; i < this.NodeCount; i++)
629 {
630     AddScaleFreeEdges(i, i - 1);
631 }
632 }
633
634 public void CreateBiConnectedGraph()
635 {
636     int i;
637     int j;
638     int nNextNode;
639     int myProbability;
640
641 //-----
642 // Start by creating a ring network. This will verify that
643 // every node is connected and, in a ring network, if any edge
644 // is removed all nodes are connected. This ensures that the
645 // graph is bi-connected.
646 //-----
647 for (i = 0; i < this.NodeCount; i++)
648 {
649     nNextNode = FindPrimaryNode(i);
650     if (nodeList[i].IsConnected(nNextNode) == false)
651     {
652         nodeList[i].AddEdge(nNextNode);
653         nodeList[nNextNode].AddEdge(i);
654     }
655 }
656 //-----
657 // We now add a few more random edges to make the graph more
658 // interesting. Loop through every combination of 2 nodes in
659 // the graph. For each combination determine if they should
660 // be connected based on the current probability. If yes,
661 // connect the nodes.
662 //-----
663 for (i = 0; i < (this.NodeCount - 1); i++)
664 {
665     for (j = i + 1; j < this.NodeCount; j++)
666     {
667         if (nodeList[i].IsConnected(j) == false)
668         {
669             myProbability = myRand.Next(0, 101);

```

```

670             if (myProbability <= this.Probability)
671             {
672                 nodeList[i].AddEdge(j);
673                 nodeList[j].AddEdge(i);
674             }
675         }
676     }
677 }
678 }
679
680 //*****
681 // Graph Validation Methods.
682 //*****
683 public bool ValidateRandomGraph()
684 {
685     bool bRtnval = false;
686
687     //-----
688     // Our only validation on a random graph is to make sure all
689     // of the nodes are still connected.
690     //-----
691     bRtnval = AreAllNodesConnected();
692     return (bRtnval);
693 }
694
695 public bool ValidateSmallWorldGraph()
696 {
697     bool bRtnval = false;
698
699     //-----
700     // Our only validation on a small world graph is to make sure
701     // all of the nodes are still connected.
702     //-----
703     bRtnval = AreAllNodesConnected();
704     return (bRtnval);
705 }
706
707 public bool ValidateScaleFreeGraph()
708 {
709     bool bRtnval = false;
710
711     //-----
712     // Our only validation on a small world graph is to make sure
713     // all of the nodes are still connected.
714     //-----
715     bRtnval = AreAllNodesConnected();
716     return (bRtnval);

```

```

717     }
718
719     public bool ValidateBiConnectedGraph()
720     {
721         int i;
722         int j;
723         int nCurNode;
724         int nNextNode;
725         bool bRtnval = true;
726
727         //-----
728         // To validate our bi-connected graph, we individually remove
729         // every edge in the graph and then verify that the graph is
730         // still connected without that single edge. Begin by marking
731         // all edges as not being checked yet.
732         //-----
733         foreach (NodeElement ne in nodeList)
734         {
735             for (i = 0; i < ne.DegreeCount; i++)
736             {
737                 ne.MarkEdgeChecked(i, false);
738             }
739         }
740         //-----
741         // Loop through each edge for each node. If the edge hasn't
742         // been checked yet, remove the edge and then see if the
743         // graph is still connected. If yes, mark the edge as checked.
744         // If not, the graph is not bi-connected. In either case,
745         // replace the edge we removed.
746         //-----
747         for (j = 0; (j < this.NodeCount) && (bRtnval == true); j++)
748         {
749             nCurNode = nodeList[j].NodeID;
750             for (i = 0; (i < nodeList[j].DegreeCount) && (bRtnval == true); i
751                 ++)
752             {
753                 if (nodeList[j].IsEdgeChecked(i) == false)
754                 {
755                     nNextNode = nodeList[j].FindNextNode(i);
756                     nodeList[nCurNode].RemoveEdge(nNextNode);
757                     nodeList[nNextNode].RemoveEdge(nCurNode);
758                     if (AreAllNodesConnected() == true)
759                     {
760                         nodeList[j].MarkEdgeChecked(i, true);
761                     }
762                     else
763                     {

```



```

763         bRtnval = false;
764     }
765     nodeList[nCurNode].AddEdge(nNextNode);
766     nodeList[nNextNode].AddEdge(nCurNode);
767 }
768 }
769 }
770     return (bRtnval);
771 }
772
773 //*****
774 // Neighbor Generation Methods.
775 //*****
776 public void GenerateRandomNeighbor()
777 {
778     bool bValid = false;
779     int nCount = 0;
780     int nFirstNode = 0;
781     int nSecondNode = 0;
782
783     restoreList.Clear();
784     //-----
785     // Loop through the process of removing a random edge and then
786     // adding a random edge until we get a valid graph.
787     //-----
788     while (bValid == false)
789     {
790         //-----
791         // Select a random node and then pick a random edge from
792         // that node. Remove the edge from both directions.
793         //-----
794         nFirstNode = SelectRandomNode(-1);
795         nSecondNode = SelectRandomEdge(nFirstNode);
796         nodeList[nFirstNode].RemoveEdge(nSecondNode);
797         nodeList[nSecondNode].RemoveEdge(nFirstNode);
798         restoreList.Insert(0, new RestorePoint(nFirstNode, nSecondNode,
799             RestorePoint.ACTION_REMOVE));
800         //-----
801         // Select 2 random nodes and if they are not already
802         // connected, connect them.
803         //-----
804         nFirstNode = SelectRandomNode(-1);
805         nSecondNode = SelectRandomNode(nFirstNode);
806         if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
807         {
808             nodeList[nFirstNode].AddEdge(nSecondNode);
809             nodeList[nSecondNode].AddEdge(nFirstNode);

```

```

809         restoreList.Insert(0, new RestorePoint(nFirstNode,
810             nSecondNode, RestorePoint.ACTION_ADD));
811     if (ValidateRandomGraph() == true)
812     {
813         //-----
814         // If we have a valid graph, we're done.
815         //-----
816         bValid = true;
817     }
818     if (bValid == false)
819     {
820         //-----
821         // Our new graph was not valid. Restore the original
822         // edge so we're back to our original graph. Check our
823         // retry count to see if we've exceeded the limit. If
824         // we have, we're just going to give up and return with
825         // the original graph.
826         //-----
827         RestoreNeighbor();
828         nCount++;
829         if (nCount > MAX_NEIGHBOR_RETRY)
830         {
831             bValid = true;
832         }
833     }
834 }
835
836
837 public void GenerateSmallWorldNeighbor()
838 {
839     bool bValid = false;
840     int nCount = 0;
841     int nFirstNode = 0;
842     int nSecondNode = 0;
843
844     restoreList.Clear();
845     //-----
846     // Loop through the process of removing and adding edges until
847     // we get a valid graph.
848     //-----
849     while (bValid == false)
850     {
851         //-----
852         // Select 2 random nodes. If they are not sequential
853         // nodes, and if they share a link, remove it.
854         //-----

```

```

855     nFirstNode = SelectRandomNode(-1);
856     nSecondNode = SelectRandomNode(nFirstNode);
857     if (NodesAreSequential(nFirstNode, nSecondNode) == false)
858     {
859         if (nodeList[nFirstNode].IsConnected(nSecondNode) == true)
860         {
861             nodeList[nFirstNode].RemoveEdge(nSecondNode);
862             nodeList[nSecondNode].RemoveEdge(nFirstNode);
863             restoreList.Insert(0, new RestorePoint(nFirstNode,
            nSecondNode, RestorePoint.ACTION_REMOVE));
864         }
865     }
866     //-----
867     // Select 2 random nodes. If they are not sequential
868     // nodes, and if they don't share a link, add one.
869     //-----
870     nFirstNode = SelectRandomNode(-1);
871     nSecondNode = SelectRandomNode(nFirstNode);
872     if (NodesAreSequential(nFirstNode, nSecondNode) == false)
873     {
874         if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
875         {
876             nodeList[nFirstNode].AddEdge(nSecondNode);
877             nodeList[nSecondNode].AddEdge(nFirstNode);
878             restoreList.Insert(0, new RestorePoint(nFirstNode,
            nSecondNode, RestorePoint.ACTION_ADD));
879         }
880     }
881     if (ValidateSmallWorldGraph() == true)
882     {
883         //-----
884         // If we have a valid graph, we're done.
885         //-----
886         bValid = true;
887     }
888     if (bValid == false)
889     {
890         //-----
891         // Our new graph was not valid. Restore the original
892         // edges so we're back to our original graph. Check
893         // our retry count to see if we've exceeded the limit.
894         // If we have, we're just going to give up and return
895         // with the original graph.
896         //-----
897         RestoreNeighbor();
898         nCount++;
899         if (nCount > MAX_NEIGHBOR_RETRY)

```

```

900         {
901             bValid = true;
902         }
903     }
904 }
905 }
906
907 public void GenerateScaleFreeNeighbor()
908 {
909     bool bValid = false;
910     int nCount = 0;
911     int myNodeID;
912
913     restoreList.Clear();
914     //-----
915     // Loop through the process of removing and adding edges until
916     // we get a valid graph.
917     //-----
918     while (bValid == false)
919     {
920         //-----
921         // Select a random node, remove all of the links from that
922         // node and then add new links based on the scale free
923         // algorithm we used when we first created the graph.
924         //-----
925         myNodeID = SelectRandomNode(-1);
926         RemoveAllEdges(myNodeID);
927         AddScaleFreeEdges(myNodeID, this.NodeCount - 1);
928         if (ValidateScaleFreeGraph() == true)
929         {
930             //-----
931             // If we have a valid graph, we're done.
932             //-----
933             bValid = true;
934         }
935         if (bValid == false)
936         {
937             //-----
938             // Our new graph was not valid. Restore the original
939             // edges so we're back to our original graph. Check
940             // our retry count to see if we've exceeded the limit.
941             // If we have, we're just going to give up and return
942             // with the original graph.
943             //-----
944             RestoreNeighbor();
945             nCount++;
946             if (nCount > MAX_NEIGHBOR_RETRY)

```

```

947         {
948             bValid = true;
949         }
950     }
951 }
952 }
953
954 public void GenerateBiConnectedNeighbor()
955 {
956     bool bValid = false;
957     int nCount = 0;
958     int nFirstNode = 0;
959     int nSecondNode = 0;
960
961     restoreList.Clear();
962     //-----
963     // Loop through the process of removing a random edge and then
964     // adding a random edge until we get a valid graph.
965     //-----
966     while (bValid == false)
967     {
968         //-----
969         // Select a random node and then pick a random edge from
970         // that node. Remove the edge from both directions.
971         //-----
972         nFirstNode = SelectRandomNode(-1);
973         nSecondNode = SelectRandomEdge(nFirstNode);
974         nodeList[nFirstNode].RemoveEdge(nSecondNode);
975         nodeList[nSecondNode].RemoveEdge(nFirstNode);
976         restoreList.Insert(0, new RestorePoint(nFirstNode, nSecondNode,
977             RestorePoint.ACTION_REMOVE));
978         //-----
979         // Select 2 random nodes and if they are not already
980         // connected, connect them.
981         //-----
982         nFirstNode = SelectRandomNode(-1);
983         nSecondNode = SelectRandomNode(nFirstNode);
984         if (nodeList[nFirstNode].IsConnected(nSecondNode) == false)
985         {
986             nodeList[nFirstNode].AddEdge(nSecondNode);
987             nodeList[nSecondNode].AddEdge(nFirstNode);
988             restoreList.Insert(0, new RestorePoint(nFirstNode,
989                 nSecondNode, RestorePoint.ACTION_ADD));
990             if (ValidateBiConnectedGraph() == true)
991             {
992                 //-----
993                 // If we have a valid graph, we're done.

```

```

992             //-----
993             bValid = true;
994         }
995     }
996     if (bValid == false)
997     {
998         //-----
999         // Our new graph was not valid. Restore the original
1000        // edge so we're back to our original graph. Check our
1001        // retry count to see if we've exceeded the limit. If
1002        // we have, we're just going to give up and return with
1003        // the original graph.
1004        //-----
1005        RestoreNeighbor();
1006        nCount++;
1007        if (nCount > MAX_NEIGHBOR_RETRY)
1008        {
1009            bValid = true;
1010        }
1011    }
1012 }
1013 }
1014
1015 //*****
1016 // Restore Neighbor Methods.
1017 //*****
1018 public void RestoreNeighbor()
1019 {
1020     int myCount;
1021     int myNode1;
1022     int myNode2;
1023     int i;
1024
1025     //-----
1026     // To restore the random neighbor we will remove the new edge
1027     // and re-add the original edge.
1028     //-----
1029     myCount = restoreList.Count();
1030     for (i = 0; i < myCount; i++)
1031     {
1032         myNode1 = restoreList[i].Node1ID;
1033         myNode2 = restoreList[i].Node2ID;
1034         if (restoreList[i].Action == RestorePoint.ACTION_ADD)
1035         {
1036             nodeList[myNode1].RemoveEdge(myNode2);
1037             nodeList[myNode2].RemoveEdge(myNode1);
1038         }

```

```

1039         else
1040         {
1041             nodeList[myNode1].AddEdge(myNode2);
1042             nodeList[myNode2].AddEdge(myNode1);
1043         }
1044     }
1045     restoreList.Clear();
1046 }
1047
1048 //*****
1049 // Other support Methods.
1050 //*****
1051 public bool NodesAreSequential(int nNode1, int nNode2)
1052 {
1053     bool bRtnval = false;
1054     int nNextNode;
1055
1056     nNextNode = nNode1 + 1;
1057     if (nNextNode >= this.NodeCount)
1058     {
1059         nNextNode = (nNode1 + 1) - this.NodeCount;
1060     }
1061     if (nNextNode == nNode2)
1062     {
1063         bRtnval = true;
1064     }
1065     else
1066     {
1067         nNextNode = nNode2 + 1;
1068         if (nNextNode >= this.NodeCount)
1069         {
1070             nNextNode = (nNode2 + 1) - this.NodeCount;
1071         }
1072         if (nNextNode == nNode1)
1073         {
1074             bRtnval = true;
1075         }
1076     }
1077     return (bRtnval);
1078 }
1079
1080 public bool AreAllNodesConnected()
1081 {
1082     int myNodeID;
1083     bool bRtnval = true;
1084
1085     //-----

```

```

1086         // Starting at the first node in the list, mark all nodes that
1087         // are currently connected. Loop through the node list and see
1088         // if all nodes have been visited. If they have, then we know
1089         // that all nodes are connected.
1090         //-----
1091         myNodeID = 0;
1092         MarkConnectedNodes(myNodeID);
1093         foreach (NodeElement ne in nodeList)
1094         {
1095             if ((ne.IsVisited) == false)
1096             {
1097                 bRtnval = false;
1098             }
1099         }
1100         return (bRtnval);
1101     }
1102
1103     public bool AreTwoNodesConnected(int Node1, int Node2)
1104     {
1105         bool bRtnval = false;
1106
1107         //-----
1108         // Mark all nodes that are currently connected with our first
1109         // node. If the second node is marked as visited, then we
1110         // know the two nodes are connected.
1111         //-----
1112         MarkConnectedNodes(Node1);
1113         if ((nodeList[Node1].IsVisited) && (nodeList[Node2].IsVisited))
1114         {
1115             bRtnval = true;
1116         }
1117         return (bRtnval);
1118     }
1119
1120     public void MarkConnectedNodes(int firstNodeID)
1121     {
1122         int i;
1123         int nNext = 0;
1124         int nCheck = 0;
1125
1126         //-----
1127         // Start by clearing the visited flag for each node in the
1128         // list.
1129         //-----
1130         foreach (NodeElement ne in nodeList)
1131         {
1132             ne.IsVisited = false;

```



```

1133         ne.IsChecked = false;
1134     }
1135     //-----
1136     // Starting with the first node, mark all of the nodes that
1137     // are connected to the first node through any path. Since
1138     // we are starting with the first node, mark it as visited.
1139     //-----
1140     nodeList[firstNodeID].IsVisited = true;
1141     while (nNext != -1)
1142     {
1143         nNext = -1;
1144         for (i = 0; (i < this.NodeCount) && (nNext == -1); i++)
1145         {
1146             if ((nodeList[i].IsVisited == true) && (nodeList[i].IsChecked
1147                 == false))
1148             {
1149                 nNext = i;
1150             }
1151         }
1152         if (nNext != -1)
1153         {
1154             for (i = 0; i < nodeList[nNext].DegreeCount; i++)
1155             {
1156                 nCheck = nodeList[nNext].FindNextNode(i);
1157                 if ((nodeList[nCheck].IsVisited) == false)
1158                 {
1159                     nodeList[nCheck].IsVisited = true;
1160                 }
1161             }
1162             nodeList[nNext].IsChecked = true;
1163         }
1164     }
1165
1166     public int SelectRandomNode(int nNotNode)
1167     {
1168         bool bDone = false;
1169         int nRandNode = -1;
1170         int nCount;
1171
1172         //-----
1173         // Select a random node that is not equal to the node ID that
1174         // was passed in. To select any random node, the calling
1175         // routine can pass in -1.
1176         //-----
1177         nCount = 0;
1178         while (bDone == false)

```

```

1179     {
1180         nRandNode = myRand.Next(0, this.NodeCount);
1181         if (nRandNode != nNotNode)
1182         {
1183             bDone = true;
1184         }
1185         else
1186         {
1187             nCount++;
1188             //-----
1189             // If we haven't found a valid random node after a
1190             // bunch of tries, either take the next node up or
1191             // down from the currently selected node. If neither
1192             // of these will work then we have some kind of weird
1193             // illegal condition and we'll just return -1;
1194             //-----
1195             if (nCount > 100)
1196             {
1197                 if (nRandNode < (this.NodeCount - 1))
1198                 {
1199                     nRandNode++;
1200                 }
1201                 else if (nRandNode > 0)
1202                 {
1203                     nRandNode--;
1204                 }
1205                 else
1206                 {
1207                     nRandNode = -1;
1208                 }
1209                 bDone = true;
1210             }
1211         }
1212     }
1213     return (nRandNode);
1214 }
1215
1216 public int SelectRandomEdge(int myNodeID) // Returns a Node ID
1217 {
1218     int nRandEdge;
1219     int nCount;
1220
1221     //-----
1222     // Make sure there are edges associated with the current node.
1223     // If so, select one at random.
1224     //-----
1225     nCount = nodeList[myNodeID].DegreeCount;

```

```

1226         if (nCount > 0)
1227         {
1228             nRandEdge = myRand.Next(0, nCount);
1229         }
1230         else
1231         {
1232             nRandEdge = -1;
1233         }
1234         return (nodeList[myNodeID].GetEdgeID(nRandEdge));
1235     }
1236
1237     public int FindPrimaryNode(int myNodeID)        // Returns a Node ID
1238     {
1239         int nNextNode;
1240
1241         //-----
1242         // The primary node will be just one position away from the
1243         // current node.
1244         //-----
1245         nNextNode = myNodeID + 1;
1246         if (nNextNode >= this.NodeCount)
1247         {
1248             nNextNode = (myNodeID + 1) - this.NodeCount;
1249         }
1250         return (nNextNode);
1251     }
1252
1253     public int FindSecondaryNode(int myNodeID)    // Returns a Node ID
1254     {
1255         int nNextNode;
1256
1257         //-----
1258         // The secondary node will be just two positions away from the
1259         // current node.
1260         //-----
1261         nNextNode = myNodeID + 2;
1262         if (nNextNode >= this.NodeCount)
1263         {
1264             nNextNode = (myNodeID + 2) - this.NodeCount;
1265         }
1266         return (nNextNode);
1267     }
1268
1269     public void RemoveAllEdges(int myNodeID)
1270     {
1271         int nNextNode;
1272         int myDegreeCount;

```

```

1273
1274 //-----
1275 // This method removes all edges from the target node.
1276 //-----
1277 while (nodeList[myNodeID].DegreeCount > 0)
1278 {
1279     myDegreeCount = nodeList[myNodeID].DegreeCount;
1280     nNextNode = nodeList[myNodeID].GetEdgeID(myDegreeCount - 1);
1281     nodeList[myNodeID].RemoveEdge(nNextNode);
1282     nodeList[nNextNode].RemoveEdge(myNodeID);
1283     restoreList.Insert(0, new RestorePoint(myNodeID, nNextNode,
1284                                         RestorePoint.ACTION_REMOVE));
1285 }
1286
1287 public void AddScaleFreeEdges(int myNodeID, int MaxNodeID)
1288 {
1289     int i;
1290     int nMaxDegree = -1;
1291     int myMaxNodeID = -1;
1292     double myProbability = 0.0;
1293     int randProbability = 0;
1294
1295 //-----
1296 // Find the maximum degree count for the existing nodes.
1297 //-----
1298 for (i = 0; i <= MaxNodeID; i++)
1299 {
1300     if (i != myNodeID)
1301     {
1302         if (nodeList[i].DegreeCount > nMaxDegree)
1303         {
1304             nMaxDegree = nodeList[i].DegreeCount;
1305             myMaxNodeID = i;
1306         }
1307     }
1308 }
1309 //-----
1310 // For each existing node, calculate the probability that the
1311 // new node will be linked to the existing node. Add links
1312 // based on those probabilities.
1313 //-----
1314 for (i = 0; i <= MaxNodeID; i++)
1315 {
1316     if (i != myNodeID)
1317     {

```

```

1318         myProbability = (((double)nodeList[i].DegreeCount) / ((double)
                                nMaxDegree)) * ((double)this.Probability);
1319         randProbability = myRand.Next(0, 101);
1320         if (((double)randProbability) <= myProbability)
1321         {
1322             nodeList[myNodeID].AddEdge(i);
1323             nodeList[i].AddEdge(myNodeID);
1324             restoreList.Insert(0, new RestorePoint(myNodeID, i,
                                RestorePoint.ACTION_ADD));
1325         }
1326     }
1327 }
1328 //-----
1329 // If after all of this activity the new node still has a
1330 // degree of 0, add an edge to the node that had the maximum
1331 // degree count.
1332 //-----
1333 if (nodeList[myNodeID].DegreeCount == 0)
1334 {
1335     nodeList[myNodeID].AddEdge(myMaxNodeID);
1336     nodeList[myMaxNodeID].AddEdge(myNodeID);
1337     restoreList.Insert(0, new RestorePoint(myNodeID, myMaxNodeID,
                                RestorePoint.ACTION_ADD));
1338 }
1339 }
1340
1341 //*****
1342 // File Load and Save Methods.
1343 //*****
1344 public void LoadGraphFromFile()
1345 {
1346     int i;
1347     int nCurNode;
1348     int nValue;
1349     int nCount;
1350     int nDone = 0;
1351     string szFilename;
1352     string szWrkbuf;
1353     string[] inputFields;
1354     StreamReader fileReader;
1355     FileStream input;
1356
1357     foreach (NodeElement ne in nodeList)
1358     {
1359         ne.ClearEdgeList();
1360     }
1361     nodeList.Clear();

```

```

1362 //-----
1363 // Open the file, read in and parse each line.
1364 //-----
1365 szFilename = this.DirectoryName + "\\\" + this.FileName + ".grf";
1366 input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1367 fileReader = new StreamReader(input);
1368 while (nDone == 0)
1369 {
1370     szWrkbuf = fileReader.ReadLine();
1371     if (szWrkbuf != null)
1372     {
1373         inputFields = szWrkbuf.Split(' ');
1374         if (inputFields[0].Equals("#"))
1375         {
1376             // This is a comment line
1377         }
1378         else if (inputFields[0].Equals("c"))
1379         {
1380             // This is the line that defines the node count
1381             this.NodeCount = Convert.ToInt32(inputFields[1]);
1382             for (i = 0; i < this.NodeCount; i++)
1383             {
1384                 nodeList.Add(new NodeElement(i, 0));
1385             }
1386         }
1387         else if (inputFields[0].Equals("n"))
1388         {
1389             // This line describes a node
1390             nCurNode = Convert.ToInt32(inputFields[1]);
1391             nCount = Convert.ToInt32(inputFields[2]);
1392             for (i = 0; i < nCount; i++)
1393             {
1394                 nValue = Convert.ToInt32(inputFields[i + 3]);
1395                 if (nValue >= 0)
1396                 {
1397                     nodeList[nCurNode].AddEdge(nValue);
1398                     nodeList[nValue].AddEdge(nCurNode);
1399                 }
1400             }
1401         }
1402     }
1403     else
1404     {
1405         nDone = 1;
1406     }
1407 }
1408

```

```

1409         //-----
1410         // Close the file.
1411         //-----
1412         fileReader.Close();
1413         input.Close();
1414     }
1415
1416     public void SaveGraphToFile()
1417     {
1418         int i;
1419         int j;
1420         int nDone = 0;
1421         int nCurDegree = 0;
1422         string szFilename;
1423         string szWrkbuf;
1424         StreamWriter fileWriter;
1425         FileStream output;
1426
1427         //-----
1428         // Make sure we have data to write out to the file before
1429         // we start.
1430         //-----
1431         if (nodeList.Count < 1)
1432         {
1433             nDone = 1;
1434         }
1435
1436         //-----
1437         // Open the file and write the initial data into it. Then
1438         // write 1 line for each node in the problem.
1439         //-----
1440         if (nDone == 0)
1441         {
1442             szFilename = this.DirectoryName + "\\\" + this.FileName + ".grf";
1443             output = new FileStream(szFilename, FileMode.Create, FileAccess.
1444                 Write);
1445             fileWriter = new StreamWriter(output);
1446             fileWriter.WriteLine("#");
1447             fileWriter.WriteLine("# Legend:");
1448             fileWriter.WriteLine("# # - Comment line");
1449             fileWriter.WriteLine("# c - Node Count");
1450             fileWriter.WriteLine("# n - Individual node data");
1451             fileWriter.WriteLine("# Value 1 - Node ID");
1452             fileWriter.WriteLine("# Value 2 - Node Degree");
1453             fileWriter.WriteLine("# Value >= 3 - Edge ID");
1454             fileWriter.WriteLine("#");
1455             fileWriter.WriteLine("c " + this.NodeCount);

```

```

1455         for (i = 0; i < this.NodeCount; i++)
1456         {
1457             nCurDegree = nodeList[i].DegreeCount;
1458             szWrkbuf = "n " + nodeList[i].NodeID + " " + nCurDegree;
1459             for (j = 0; j < nCurDegree; j++)
1460             {
1461                 szWrkbuf += " " + nodeList[i].GetEdgeID(j);
1462             }
1463             fileWriter.WriteLine(szWrkbuf);
1464         }
1465         //-----
1466         // Close the file.
1467         //-----
1468         fileWriter.Close();
1469         output.Close();
1470     }
1471 }
1472
1473 public void LoadHCResults(int myMonotonic)
1474 {
1475     int nValue;
1476     int nDone = 0;
1477     string szFilename;
1478     string szWrkbuf;
1479     string[] inputFields;
1480     StreamReader fileReader;
1481     FileStream input;
1482     HillClimbResult tmphc = new HillClimbResult();
1483
1484     myHCResults.Clear();
1485     //-----
1486     // Open the file, read in and parse each line.
1487     //-----
1488     szFilename = this.FileName;
1489     input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1490     fileReader = new StreamReader(input);
1491     while (nDone == 0)
1492     {
1493         szWrkbuf = fileReader.ReadLine();
1494         if (szWrkbuf != null)
1495         {
1496             inputFields = szWrkbuf.Split(' ');
1497             if (inputFields[0].Equals("#"))
1498             {
1499                 // This is a comment line
1500             }
1501             else

```



```

1545         // we start.
1546         //-----
1547         if (myHCResults.Count < 1)
1548         {
1549             nDone = 1;
1550         }
1551
1552         //-----
1553         // Open the file and write each of the result lines.
1554         //-----
1555         if (nDone == 0)
1556         {
1557             if (myMonotonic == MON_HARDNESS)
1558             {
1559                 szExt = ".hrd";
1560             }
1561             else
1562             {
1563                 szExt = ".psi";
1564             }
1565             szFilename = this.DirectoryName + "\\\" + this.FileName + szExt;
1566             output = new FileStream(szFilename, FileMode.Create, FileAccess.
                Write);
1567             fileWriter = new StreamWriter(output);
1568             foreach (HillClimbResult hc in myHCResults)
1569             {
1570                 nSolvable = 0;
1571                 if (hc.ColorIsSolvable == true)
1572                 {
1573                     nSolvable = 1;
1574                 }
1575                 szWrkbuf = hc.ColorCount.ToString() + " " + nSolvable.
                    ToString() + " " +
1576                     hc.ConflictCount.ToString() + " " + hc.
                        HardnessTime.ToString() + " " +
1577                     hc.PSIValue.ToString() + " " + hc.PSITime.ToString
                        ();
1578                 fileWriter.WriteLine(szWrkbuf);
1579             }
1580             //-----
1581             // Close the file.
1582             //-----
1583             fileWriter.Close();
1584             output.Close();
1585         }
1586     }
1587

```

```

1588     public void LoadLeafResults(string LeafDir, int myMonotonic)
1589     {
1590         int nDone = 0;
1591         string szFilename;
1592         string szWrkbuf;
1593         string szExt;
1594         string[] inputFields;
1595         StreamReader fileReader;
1596         FileStream input;
1597         ResultsData rData = new ResultsData();
1598
1599         //-----
1600         // Clear the results list and open the target file.
1601         //-----
1602         resultsData.Clear();
1603         if (myMonotonic == MON_HARDNESS)
1604         {
1605             szExt = ".hrd";
1606         }
1607         else
1608         {
1609             szExt = ".psi";
1610         }
1611         szFilename = LeafDir + "\\Lresults" + szExt;
1612         input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1613         fileReader = new StreamReader(input);
1614         //-----
1615         // Read each line from the file, parse it into a results
1616         // structure and add it to the results list.
1617         //-----
1618         while (nDone == 0)
1619         {
1620             szWrkbuf = fileReader.ReadLine();
1621             if (szWrkbuf != null)
1622             {
1623                 inputFields = szWrkbuf.Split(' ');
1624                 if (inputFields[0].Equals("#"))
1625                 {
1626                     // This is a comment line
1627                 }
1628                 else
1629                 {
1630                     // This is a results line
1631                     rData.NodeCount = 0;           // Not used
1632                     rData.Probability = 0;        // Not used
1633                     rData.FileNumber = inputFields[0];
1634                     rData.MinHardness = Convert.ToInt64(inputFields[1]);

```

```

1635         rData.MinPSI = Convert.ToDouble(inputFields[2]);
1636         rData.MaxHardness = Convert.ToInt64(inputFields[3]);
1637         rData.HardnessTime = 0.0; // Not used
1638         rData.MaxPSI = Convert.ToDouble(inputFields[4]);
1639         rData.PSITime = 0.0; // Not used
1640         resultsData.Add(new ResultsData(rData.NodeCount, rData.
                Probability,
1641                                     rData.MinHardness, rData.
                MaxHardness,
1642                                     rData.MinPSI, rData.
                MaxPSI,
1643                                     rData.HardnessTime, rData.
                .PSITime,
                rData.FileName));
1644     }
1645 }
1646 }
1647 else
1648 {
1649     nDone = 1;
1650 }
1651 }
1652
1653 //-----
1654 // Close the file.
1655 //-----
1656 fileReader.Close();
1657 input.Close();
1658 }
1659
1660 public void SaveLeafResults(string LeafDir, int myMonotonic)
1661 {
1662     int nDone = 0;
1663     string szFilename;
1664     string szExt;
1665     string szWrkbuf;
1666     StreamWriter fileWriter;
1667     FileStream output;
1668
1669     //-----
1670     // Make sure we have results to write out to the file before
1671     // we start.
1672     //-----
1673     if (resultsData.Count < 1)
1674     {
1675         nDone = 1;
1676     }
1677

```

```

1678 //-----
1679 // Open the file and write each of the result lines.
1680 //-----
1681 if (nDone == 0)
1682 {
1683     if (myMonotonic == MON_HARDNESS)
1684     {
1685         szExt = ".hrd";
1686     }
1687     else
1688     {
1689         szExt = ".psi";
1690     }
1691     szFilename = LeafDir + "\\Lresults" + szExt;
1692     output = new FileStream(szFilename, FileMode.Create, FileAccess.
        Write);
1693     fileWriter = new StreamWriter(output);
1694     foreach (ResultsData rd in resultsData)
1695     {
1696         szWrkbuf = rd.FileNumber + " " + rd.MinHardness.ToString() +
            " " +
1697             rd.MinPSI.ToString() + " " + rd.MaxHardness.
                ToString() + " " +
1698             rd.MaxPSI.ToString();
1699         fileWriter.WriteLine(szWrkbuf);
1700     }
1701     //-----
1702     // Close the file.
1703     //-----
1704     fileWriter.Close();
1705     output.Close();
1706 }
1707 }
1708
1709 public void LoadLeafAverages(string LeafDir, int myMonotonic)
1710 {
1711     int nDone = 0;
1712     string szFilename;
1713     string szWrkbuf;
1714     string szExt;
1715     string[] inputFields;
1716     StreamReader fileReader;
1717     FileStream input;
1718
1719     //-----
1720     // Open the target file.
1721     //-----

```

```

1722         if (myMonotonic == MON_HARDNESS)
1723         {
1724             szExt = ".hrd";
1725         }
1726         else
1727         {
1728             szExt = ".psi";
1729         }
1730         szFilename = LeafDir + "\\Lverages" + szExt;
1731         input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1732         fileReader = new StreamReader(input);
1733         //-----
1734         // Averages files only contain one line of data, but we will
1735         // read multiple lines just in case there are some comments
1736         // included. If we happen to read multiple data lines, the
1737         // last line read will be the one we keep.
1738         //-----
1739         while (nDone == 0)
1740         {
1741             szWrkbuf = fileReader.ReadLine();
1742             if (szWrkbuf != null)
1743             {
1744                 inputFields = szWrkbuf.Split(' ');
1745                 if (inputFields[0].Equals("#"))
1746                 {
1747                     // This is a comment line
1748                 }
1749                 else
1750                 {
1751                     // This is an averages line
1752                     averagesData.GraphType = inputFields[0];
1753                     averagesData.NodeCount = Convert.ToInt32(inputFields[1]);
1754                     averagesData.Probability = Convert.ToInt32(inputFields
1755                         [2]);
1756                     averagesData.AvgMinHardness = Convert.ToInt64(inputFields
1757                         [3]);
1758                     averagesData.AvgMinPSI = Convert.ToDouble(inputFields[4])
1759                         ;
1760                     averagesData.AvgMaxHardness = Convert.ToInt64(inputFields
1761                         [5]);
1762                     averagesData.AvgHardnessTime = Convert.ToDouble(
1763                         inputFields[6]);
1764                     averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[7])
1765                         ;
1766                     averagesData.AvgPSITime = Convert.ToDouble(inputFields
1767                         [8]);
1768                 }
1769             }

```

```

1762         }
1763         else
1764         {
1765             nDone = 1;
1766         }
1767     }
1768
1769     //-----
1770     // Close the file.
1771     //-----
1772     fileReader.Close();
1773     input.Close();
1774 }
1775
1776 public void SaveLeafAverages(string LeafDir, int myMonotonic)
1777 {
1778     string szFilename;
1779     string szExt;
1780     string szWrkbuf;
1781     StreamWriter fileWriter;
1782     FileStream output;
1783
1784     //-----
1785     // Open the file and write out our single line of averages.
1786     //-----
1787     if (myMonotonic == MON_HARDNESS)
1788     {
1789         szExt = ".hrd";
1790     }
1791     else
1792     {
1793         szExt = ".psi";
1794     }
1795     szFilename = LeafDir + "\\Laverages" + szExt;
1796     output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
1797         );
1798     fileWriter = new StreamWriter(output);
1799     szWrkbuf = averagesData.GraphType + " " +
1800         averagesData.NodeCount.ToString() + " " +
1801         averagesData.Probability.ToString() + " " +
1802         averagesData.AvgMinHardness.ToString() + " " +
1803         averagesData.AvgMinPSI + " " +
1804         averagesData.AvgMaxHardness.ToString() + " " +
1805         averagesData.AvgHardnesTime.ToString() + " " +
1806         averagesData.AvgMaxPSI.ToString() + " " +
1807         averagesData.AvgPSITime.ToString();
1808     fileWriter.WriteLine(szWrkbuf);

```

```

1808          //-----
1809          // Close the file.
1810          //-----
1811          fileWriter.Close();
1812          output.Close();
1813      }
1814
1815      public void LoadProbabilityResults(string ProbabilityDir, int myMonotonic
1816      )
1817      {
1818          int nDone = 0;
1819          string szFilename;
1820          string szWrkbuf;
1821          string szExt;
1822          string[] inputFields;
1823          StreamReader fileReader;
1824          FileStream input;
1825          ResultsData rData = new ResultsData();
1826
1827          //-----
1828          // Clear the results list and open the target file.
1829          //-----
1830          resultsData.Clear();
1831          if (myMonotonic == MON_HARDNESS)
1832          {
1833              szExt = ".hrd";
1834          }
1835          else
1836          {
1837              szExt = ".psi";
1838          }
1839          szFilename = ProbabilityDir + "\\Presults" + szExt;
1840          input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1841          fileReader = new StreamReader(input);
1842
1843          //-----
1844          // Read each line from the file, parse it into a results
1845          // structure and add it to the results list.
1846          //-----
1847          while (nDone == 0)
1848          {
1849              szWrkbuf = fileReader.ReadLine();
1850              if (szWrkbuf != null)
1851              {
1852                  inputFields = szWrkbuf.Split(' ');
1853                  if (inputFields[0].Equals("#"))
1854                  {
1855                      // This is a comment line

```



```

1854         }
1855         else
1856         {
1857             // This is a results line
1858             rData.NodeCount = 0; // Not used
1859             rData.Probability = Convert.ToInt32(inputFields[0]);
1860             rData.FileNumber = string.Empty; // Not used
1861             rData.MinHardness = Convert.ToInt64(inputFields[1]);
1862             rData.MinPSI = Convert.ToDouble(inputFields[2]);
1863             rData.MaxHardness = Convert.ToInt64(inputFields[3]);
1864             rData.HardnessTime = Convert.ToDouble(inputFields[4]);
1865             rData.MaxPSI = Convert.ToDouble(inputFields[5]);
1866             rData.PSITime = Convert.ToDouble(inputFields[6]);
1867             resultsData.Add(new ResultsData(rData.NodeCount, rData.
                Probability,
1868                                     rData.MinHardness, rData.
                MaxHardness,
1869             rData.MinPSI, rData.
                MaxPSI,
1870             rData.HardnessTime, rData
                .PSITime,
                rData.FileNumber));
1871         }
1872     }
1873     }
1874     else
1875     {
1876         nDone = 1;
1877     }
1878 }
1879
1880 //-----
1881 // Close the file.
1882 //-----
1883 fileReader.Close();
1884 input.Close();
1885 }
1886
1887 public void SaveProbabilityResults(string ProbabilityDir, int myMonotonic
    )
1888 {
1889     int nDone = 0;
1890     string szFilename;
1891     string szExt;
1892     string szWrkbuf;
1893     StreamWriter fileWriter;
1894     FileStream output;
1895

```

```

1896 //-----
1897 // Make sure we have results to write out to the file before
1898 // we start.
1899 //-----
1900 if (resultsData.Count < 1)
1901 {
1902     nDone = 1;
1903 }
1904
1905 //-----
1906 // Open the file and write each of the result lines.
1907 //-----
1908 if (nDone == 0)
1909 {
1910     if (myMonotonic == MON_HARDNESS)
1911     {
1912         szExt = ".hrd";
1913     }
1914     else
1915     {
1916         szExt = ".psi";
1917     }
1918     szFilename = ProbabilityDir + "\\Presults" + szExt;
1919     output = new FileStream(szFilename, FileMode.Create, FileAccess.
        Write);
1920     fileWriter = new StreamWriter(output);
1921     foreach (ResultsData rd in resultsData)
1922     {
1923         szWrkbuf = rd.Probability.ToString() + " " + rd.MinHardness.
            ToString() + " " +
1924             rd.MinPSI.ToString() + " " + rd.MaxHardness.
            ToString() + " " +
1925             rd.HardnessTime.ToString() + " " + rd.MaxPSI.
            ToString() + " " +
1926             rd.PSITime.ToString();
1927         fileWriter.WriteLine(szWrkbuf);
1928     }
1929     //-----
1930     // Close the file.
1931     //-----
1932     fileWriter.Close();
1933     output.Close();
1934 }
1935 }
1936
1937 public void LoadProbabilityAverages(string ProbabilityDir, int
        myMonotonic)

```

```

1938     {
1939         int nDone = 0;
1940         string szFilename;
1941         string szWrkbuf;
1942         string szExt;
1943         string[] inputFields;
1944         StreamReader fileReader;
1945         FileStream input;
1946
1947         //-----
1948         // Open the target file.
1949         //-----
1950         if (myMonotonic == MON_HARDNESS)
1951         {
1952             szExt = ".hrd";
1953         }
1954         else
1955         {
1956             szExt = ".psi";
1957         }
1958         szFilename = ProbabilityDir + "\\Paverages" + szExt;
1959         input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
1960         fileReader = new StreamReader(input);
1961         //-----
1962         // Averages files only contain one line of data, but we will
1963         // read multiple lines just in case there are some comments
1964         // included. If we happen to read multiple data lines, the
1965         // last line read will be the one we keep.
1966         //-----
1967         while (nDone == 0)
1968         {
1969             szWrkbuf = fileReader.ReadLine();
1970             if (szWrkbuf != null)
1971             {
1972                 inputFields = szWrkbuf.Split(' ');
1973                 if (inputFields[0].Equals("#"))
1974                 {
1975                     // This is a comment line
1976                 }
1977                 else
1978                 {
1979                     // This is an averages line
1980                     averagesData.GraphType = inputFields[0];
1981                     averagesData.NodeCount = Convert.ToInt32(inputFields[1]);
1982                     averagesData.Probability = 0; // Not used
1983                     averagesData.AvgMinHardness = Convert.ToInt64(inputFields
1984                         [2]);

```

```

1984         averagesData.AvgMinPSI = Convert.ToDouble(inputFields[3])
1985         ;
1986         averagesData.AvgMaxHardness = Convert.ToInt64(inputFields
1987         [4]);
1988         averagesData.AvgHardnessTime = Convert.ToDouble(
1989         inputFields[5]);
1990         averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[6])
1991         ;
1992         averagesData.AvgPSITime = Convert.ToDouble(inputFields
1993         [7]);
1994     }
1995 }
1996 else
1997 {
1998     nDone = 1;
1999 }
2000 }
2001
2002 //-----
2003 // Close the file.
2004 //-----
2005 fileReader.Close();
2006 input.Close();
2007 }
2008
2009 public void SaveProbabilityAverages(string ProbabilityDir, int
2010 myMonotonic)
2011 {
2012     string szFilename;
2013     string szExt;
2014     string szWrkbuf;
2015     StreamWriter fileWriter;
2016     FileStream output;
2017
2018     //-----
2019     // Open the file and write out our single line of averages.
2020     //-----
2021     if (myMonotonic == MON_HARDNESS)
2022     {
2023         szExt = ".hrd";
2024     }
2025     else
2026     {
2027         szExt = ".psi";
2028     }
2029     szFilename = ProbabilityDir + "\\Paverages" + szExt;

```

```

2024         output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
2025         );
2026         fileWriter = new StreamWriter(output);
2027         szWrkbuf = averagesData.GraphType + " " +
2028         averagesData.NodeCount.ToString() + " " +
2029         averagesData.AvgMinHardness.ToString() + " " +
2030         averagesData.AvgMinPSI + " " +
2031         averagesData.AvgMaxHardness.ToString() + " " +
2032         averagesData.AvgHardnessTime.ToString() + " " +
2033         averagesData.AvgMaxPSI.ToString() + " " +
2034         averagesData.AvgPSITime.ToString();
2035         fileWriter.WriteLine(szWrkbuf);
2036         //-----
2037         // Close the file.
2038         //-----
2039         fileWriter.Close();
2040         output.Close();
2041     }
2042     public void LoadNodeCountResults(string NodeCountDir, int myMonotonic)
2043     {
2044         int nDone = 0;
2045         string szFilename;
2046         string szWrkbuf;
2047         string szExt;
2048         string[] inputFields;
2049         StreamReader fileReader;
2050         FileStream input;
2051         ResultsData rData = new ResultsData();
2052
2053         //-----
2054         // Clear the results list and open the target file.
2055         //-----
2056         resultsData.Clear();
2057         if (myMonotonic == MON_HARDNESS)
2058         {
2059             szExt = ".hrd";
2060         }
2061         else
2062         {
2063             szExt = ".psi";
2064         }
2065         szFilename = NodeCountDir + "\\Nresults" + szExt;
2066         input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
2067         fileReader = new StreamReader(input);
2068         //-----
2069         // Read each line from the file, parse it into a results

```

```

2070 // structure and add it to the results list.
2071 //-----
2072 while (nDone == 0)
2073 {
2074     szWrkbuf = fileReader.ReadLine();
2075     if (szWrkbuf != null)
2076     {
2077         inputFields = szWrkbuf.Split(' ');
2078         if (inputFields[0].Equals("#"))
2079         {
2080             // This is a comment line
2081         }
2082         else
2083         {
2084             // This is a results line
2085             rData.NodeCount = Convert.ToInt32(inputFields[0]);
2086             rData.Probability = 0; // Not used
2087             rData.FileNumber = string.Empty; // Not used
2088             rData.MinHardness = Convert.ToInt64(inputFields[1]);
2089             rData.MinPSI = Convert.ToDouble(inputFields[2]);
2090             rData.MaxHardness = Convert.ToInt64(inputFields[3]);
2091             rData.HardnessTime = Convert.ToDouble(inputFields[4]);
2092             rData.MaxPSI = Convert.ToDouble(inputFields[5]);
2093             rData.PSITime = Convert.ToDouble(inputFields[6]);
2094             resultsData.Add(new ResultsData(rData.NodeCount, rData.
2095                 Probability,
2096                 rData.MinHardness, rData.
2097                     MaxHardness,
2098                     rData.MinPSI, rData.
2099                         MaxPSI,
2100                         rData.HardnessTime, rData.
2101                             PSITime,
2102                             rData.FileNumber));
2103         }
2104     }
2105     nDone = 1;
2106 }
2107 //-----
2108 // Close the file.
2109 //-----
2110 fileReader.Close();
2111 input.Close();
2112 }

```

```

2113
2114     public void SaveNodeCountResults(string NodeCountDir, int myMonotonic)
2115     {
2116         int nDone = 0;
2117         string szFilename;
2118         string szExt;
2119         string szWrkbuf;
2120         StreamWriter fileWriter;
2121         FileStream output;
2122
2123         //-----
2124         // Make sure we have results to write out to the file before
2125         // we start.
2126         //-----
2127         if (resultsData.Count < 1)
2128         {
2129             nDone = 1;
2130         }
2131
2132         //-----
2133         // Open the file and write each of the result lines.
2134         //-----
2135         if (nDone == 0)
2136         {
2137             if (myMonotonic == MON_HARDNESS)
2138             {
2139                 szExt = ".hrd";
2140             }
2141             else
2142             {
2143                 szExt = ".psi";
2144             }
2145             szFilename = NodeCountDir + "\\Nresults" + szExt;
2146             output = new FileStream(szFilename, FileMode.Create, FileAccess.
                Write);
2147             fileWriter = new StreamWriter(output);
2148             foreach (ResultsData rd in resultsData)
2149             {
2150                 szWrkbuf = rd.NodeCount.ToString() + " " + rd.MinHardness.
                    ToString() + " " +
2151                     rd.MinPSI.ToString() + " " + rd.MaxHardness.
                        ToString() + " " +
2152                     rd.HardnessTime.ToString() + " " + rd.MaxPSI.
                        ToString() + " " +
2153                     rd.PSITime.ToString();
2154                 fileWriter.WriteLine(szWrkbuf);
2155             }

```

```

2156         //-----
2157         // Close the file.
2158         //-----
2159         fileWriter.Close();
2160         output.Close();
2161     }
2162 }
2163
2164 public void LoadNodeCountAverages(string NodeCountDir, int myMonotonic)
2165 {
2166     int nDone = 0;
2167     string szFilename;
2168     string szWrkbuf;
2169     string szExt;
2170     string[] inputFields;
2171     StreamReader fileReader;
2172     FileStream input;
2173
2174     //-----
2175     // Open the target file.
2176     //-----
2177     if (myMonotonic == MON_HARDNESS)
2178     {
2179         szExt = ".hrd";
2180     }
2181     else
2182     {
2183         szExt = ".psi";
2184     }
2185     szFilename = NodeCountDir + "\\Naverages" + szExt;
2186     input = new FileStream(szFilename, FileMode.Open, FileAccess.Read);
2187     fileReader = new StreamReader(input);
2188     //-----
2189     // Averages files only contain one line of data, but we will
2190     // read multiple lines just in case there are some comments
2191     // included. If we happen to read multiple data lines, the
2192     // last line read will be the one we keep.
2193     //-----
2194     while (nDone == 0)
2195     {
2196         szWrkbuf = fileReader.ReadLine();
2197         if (szWrkbuf != null)
2198         {
2199             inputFields = szWrkbuf.Split(' ');
2200             if (inputFields[0].Equals("#"))
2201             {
2202                 // This is a comment line

```



```

2203         }
2204         else
2205         {
2206             // This is an averages line
2207             averagesData.GraphType = inputFields[0];
2208             averagesData.NodeCount = 0;    // Not used
2209             averagesData.Probability = 0;  // Not used
2210             averagesData.AvgMinHardness = Convert.ToInt64(inputFields
2211                 [3]);
2212             averagesData.AvgMinPSI = Convert.ToDouble(inputFields[4])
2213                 ;
2214             averagesData.AvgMaxHardness = Convert.ToInt64(inputFields
2215                 [5]);
2216             averagesData.AvgHardnessTime = Convert.ToDouble(
2217                 inputFields[6]);
2218             averagesData.AvgMaxPSI = Convert.ToDouble(inputFields[7])
2219                 ;
2220             averagesData.AvgPSITime = Convert.ToDouble(inputFields
2221                 [8]);
2222         }
2223     }
2224     //-----
2225     // Close the file.
2226     //-----
2227     fileReader.Close();
2228     input.Close();
2229 }
2230
2231 public void SaveNodeCountAverages(string NodeCountDir, int myMonotonic)
2232 {
2233     string szFilename;
2234     string szExt;
2235     string szWrkbuf;
2236     StreamWriter fileWriter;
2237     FileStream output;
2238
2239     //-----
2240     // Open the file and write out our single line of averages.
2241     //-----
2242     if (myMonotonic == MON_HARDNESS)
2243     {

```

```

2244         szExt = ".hrd";
2245     }
2246     else
2247     {
2248         szExt = ".psi";
2249     }
2250     szFilename = NodeCountDir + "\\Naverages" + szExt;
2251     output = new FileStream(szFilename, FileMode.Create, FileAccess.Write
        );
2252     fileWriter = new StreamWriter(output);
2253     szWrkbuf = averagesData.GraphType + " " +
2254         averagesData.AvgMinHardness.ToString() + " " +
2255         averagesData.AvgMinPSI + " " +
2256         averagesData.AvgMaxHardness.ToString() + " " +
2257         averagesData.AvgHardnesTime.ToString() + " " +
2258         averagesData.AvgMaxPSI.ToString() + " " +
2259         averagesData.AvgPSITime.ToString();
2260     fileWriter.WriteLine(szWrkbuf);
2261     //-----
2262     // Close the file.
2263     //-----
2264     fileWriter.Close();
2265     output.Close();
2266 }
2267
2268 public void CalculateLeafResults(string LeafDir, string myGraphType,
2269     int myNodeCount, int myProbability)
2270 {
2271     int nRtnval = 0;
2272     string filePath = string.Empty;
2273     ResultsData rData = new ResultsData();
2274     AveragesData aData = new AveragesData();
2275
2276     //-----
2277     // If results and averages files already exist in the current
2278     // directory, delete them.
2279     //-----
2280     if (nRtnval == 0)
2281     {
2282         filePath = LeafDir + "\\Lresults.*";
2283         if (File.Exists(filePath))
2284         {
2285             File.Delete(filePath);
2286         }
2287         filePath = LeafDir + "\\Laverages.*";
2288         if (File.Exists(filePath))
2289         {

```

```

2290         File.Delete(filePath);
2291     }
2292     filePath = LeafDir + "\\Legend.txt";
2293     if (File.Exists(filePath))
2294     {
2295         File.Delete(filePath);
2296     }
2297 }
2298 //-----
2299 // Get a list of the hardness files in the directory. For each
2300 // file, find the first and last data lines which represent
2301 // the minimum and maximum values for that specific file. Save
2302 // the contents of that line in the results list and also add
2303 // the contents to the totals and count which will be used to
2304 // calculate the averages.
2305 //-----
2306 if (nRtnval == 0)
2307 {
2308     string[] fileList = Directory.GetFiles(LeafDir, "grf*.hrd");
2309     resultsData.Clear();
2310     int nLastOff = 0;
2311     int nCurOff = 0;
2312     int nFilCnt = 0;
2313     this.DirectoryName = LeafDir;
2314     aData.AvgMinHardness = 0;
2315     aData.AvgMinPSI = 0.0;
2316     aData.AvgMaxHardness = 0;
2317     aData.AvgMaxPSI = 0.0;
2318     aData.AvgHardnessTime = 0.0;
2319     aData.AvgPSITime = 0.0;
2320     foreach (string filename in fileList)
2321     {
2322         nFilCnt++;
2323         //-----
2324         // Initialize working variables and load the hill
2325         // climber results from the file.
2326         //-----
2327         this.FileName = filename;
2328         LoadHCResults(MON_HARDNESS);
2329         nLastOff = myHCResults.Count() - 1;
2330         nCurOff = 0;
2331         rData.NodeCount = 0;           // Not used
2332         rData.Probability = 0;        // Not used
2333         rData.MinHardness = 0;
2334         rData.MinPSI = 0.0;
2335         rData.MaxHardness = 0;
2336         rData.MaxPSI = 0.0;

```

```

2337         rData.HardnessTime = 0.0;
2338         rData.PSITime = 0.0;
2339         rData.FileNumber = filename;
2340         //-----
2341         // Loop through each entry in the file so we can save
2342         // the minimum, maximum, and average values.
2343         //-----
2344         foreach (HillClimbResult hc in myHCResults)
2345         {
2346             //-----
2347             // If this is the first entry, it will be the
2348             // minimum values for the hill climb.
2349             //-----
2350             if (nCurOff == 0)
2351             {
2352                 rData.MinHardness = myHCResults[nCurOff].
2353                     ConflictCount;
2354                 rData.MinPSI = myHCResults[nCurOff].PSIValue;
2355                 aData.AvgMinHardness += rData.MinHardness;
2356                 aData.AvgMinPSI += rData.MinPSI;
2357             }
2358             //-----
2359             // Save totals that we will use to calculate the
2360             // average values.
2361             //-----
2362             rData.HardnessTime += myHCResults[nCurOff].HardnessTime;
2363             rData.PSITime += myHCResults[nCurOff].PSITime;
2364             //-----
2365             // If this is the last entry, it will be the
2366             // maximum values for the hill climb.
2367             //-----
2368             if (nCurOff == nLastOff)
2369             {
2370                 rData.MaxHardness = myHCResults[nCurOff].
2371                     ConflictCount;
2372                 rData.MaxPSI = myHCResults[nCurOff].PSIValue;
2373                 aData.AvgMaxHardness += rData.MaxHardness;
2374                 aData.AvgMaxPSI += rData.MaxPSI;
2375                 aData.AvgHardnessTime += (rData.HardnessTime / (
2376                     nCurOff + 1));
2377                 aData.AvgPSITime += (rData.PSITime / (nCurOff + 1));
2378             }
2379             nCurOff++;
2380         }
2381         //-----
2382         // Write out the hardness results file.
2383         //-----

```

```

2381         resultsData.Add(new ResultsData(rData.NodeCount, rData.
2382             Probability,
2383             rData.MinHardness, rData.
2384                 MaxHardness,
2385                 rData.MinPSI, rData.MaxPSI,
2386                 0.0, 0.0,
2387                 rData.FileName));
2388     }
2389     SaveLeafResults(LeafDir, MON_HARDNESS);
2390     if (nFilCnt > 0)
2391     {
2392         //-----
2393         // Calculate the hardness averages and write those out
2394         // to the harness averages file.
2395         //-----
2396         averagesData.GraphType = myGraphType;
2397         averagesData.NodeCount = myNodeCount;
2398         averagesData.Probability = myProbability;
2399         averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2400         averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2401         averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2402         averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
2403         averagesData.AvgHardnessTime = aData.AvgHardnessTime /
2404             nFilCnt;
2405         averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
2406         SaveLeafAverages(LeafDir, MON_HARDNESS);
2407     }
2408 }
2409
2410 //-----
2411 // Get a list of the PSI files in the directory. For each
2412 // file, find the first and last data lines which represent
2413 // the minimum and maximum values for that specific file. Save
2414 // the contents of that line in the results list and also add
2415 // the contents to the totals and count which will be used to
2416 // calculate the averages.
2417 //-----
2418 if (nRtnval == 0)
2419 {
2420     string[] fileList = Directory.GetFiles(LeafDir, "grf*.psi");
2421     resultsData.Clear();
2422     int nLastOff = 0;
2423     int nCurOff = 0;
2424     int nFilCnt = 0;
2425     this.DirectoryName = LeafDir;
2426     aData.AvgMinHardness = 0;
2427     aData.AvgMinPSI = 0.0;

```

```

2424         aData.AvgMaxHardness = 0;
2425         aData.AvgMaxPSI = 0.0;
2426         aData.AvgHardnessTime = 0.0;
2427         aData.AvgPSITime = 0.0;
2428         foreach (string filename in fileList)
2429         {
2430             nFilCnt++;
2431             //-----
2432             // Initialize working variables and load the hill
2433             // climber results from the file.
2434             //-----
2435             this.FileName = filename;
2436             LoadHCResults(MON_PSI);
2437             nLastOff = myHCResults.Count() - 1;
2438             nCurOff = 0;
2439             rData.NodeCount = 0;           // Not used
2440             rData.Probability = 0;        // Not used
2441             rData.MinHardness = 0;
2442             rData.MinPSI = 0.0;
2443             rData.MaxHardness = 0;
2444             rData.MaxPSI = 0.0;
2445             rData.HardnessTime = 0.0;
2446             rData.PSITime = 0.0;
2447             rData.FileNumber = filename;
2448             //-----
2449             // Loop through each entry in the file so we can save
2450             // the minimum, maximum, and average values.
2451             //-----
2452             foreach (HillClimbResult hc in myHCResults)
2453             {
2454                 //-----
2455                 // If this is the first entry, it will be the
2456                 // minimum values for the hill climb.
2457                 //-----
2458                 if (nCurOff == 0)
2459                 {
2460                     rData.MinHardness = myHCResults[nCurOff].
                        ConflictCount;
2461                     rData.MinPSI = myHCResults[nCurOff].PSIValue;
2462                     aData.AvgMinHardness += rData.MinHardness;
2463                     aData.AvgMinPSI += rData.MinPSI;
2464                 }
2465                 //-----
2466                 // Save totals that we will use to calculate the
2467                 // average values.
2468                 //-----
2469                 rData.HardnessTime += myHCResults[nCurOff].HardnessTime;

```

```

2470         rData.PSITime += myHCResults[nCurOff].PSITime;
2471         //-----
2472         // If this is the last entry, it will be the
2473         // maximum values for the hill climb.
2474         //-----
2475         if (nCurOff == nLastOff)
2476         {
2477             rData.MaxHardness = myHCResults[nCurOff].
                ConflictCount;
2478             rData.MaxPSI = myHCResults[nCurOff].PSIValue;
2479             aData.AvgMaxHardness += rData.MaxHardness;
2480             aData.AvgMaxPSI += rData.MaxPSI;
2481             aData.AvgHardnessTime += (rData.HardnessTime / (
                nCurOff + 1));
2482             aData.AvgPSITime += (rData.PSITime / (nCurOff + 1));
2483         }
2484         nCurOff++;
2485     }
2486     //-----
2487     // Write out the PSI results file.
2488     //-----
2489     resultsData.Add(new ResultsData(rData.NodeCount, rData.
        Probability,
2490                                     rData.MinHardness, rData.
                MaxHardness,
2491                                     rData.MinPSI, rData.MaxPSI,
                0.0, 0.0,
2492                                     rData.FileNumber));
2493 }
2494 SaveLeafResults(LeafDir, MON_PSI);
2495 if (nFilCnt > 0)
2496 {
2497     //-----
2498     // Calculate the PSI averages and write those out to
2499     // the PSI averages file.
2500     //-----
2501     averagesData.GraphType = myGraphType;
2502     averagesData.NodeCount = myNodeCount;
2503     averagesData.Probability = myProbability;
2504     averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2505     averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2506     averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2507     averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
2508     averagesData.AvgHardnessTime = aData.AvgHardnessTime /
        nFilCnt;
2509     averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
2510     SaveLeafAverages(LeafDir, MON_PSI);

```

```

2511     }
2512 }
2513
2514 //-----
2515 // Finally, write out the legend file.
2516 //-----
2517 if (nRtnval == 0)
2518 {
2519     string szFilename;
2520     szFilename = LeafDir + "\\Legend.txt";
2521     output = new FileStream(szFilename, FileMode.Create, FileAccess.
        Write);
2522     fileWriter = new StreamWriter(output);
2523     fileWriter.WriteLine("#");
2524     fileWriter.WriteLine("# There are two results files in the
        directory");
2525     fileWriter.WriteLine("# Lresults.hrd - Results when solving
        for hardness");
2526     fileWriter.WriteLine("# Lresults.psi - Results when solving
        for PSI");
2527     fileWriter.WriteLine("# Each line in each file contains the
        following elements");
2528     fileWriter.WriteLine("# File number - The file number");
2529     fileWriter.WriteLine("# Min Hardness - The minimum hardness
        value");
2530     fileWriter.WriteLine("# Min PSI - The minimum PSI value");
2531     fileWriter.WriteLine("# Max Hardness - The maximum hardness
        value");
2532     fileWriter.WriteLine("# Max PSI - The maximum PSI value");
2533     fileWriter.WriteLine("#");
2534     fileWriter.WriteLine("# There are two averages files in the
        directory");
2535     fileWriter.WriteLine("# Laverages.hrd - Averages when solving
        for hardness");
2536     fileWriter.WriteLine("# Laverages.psi - Averages when solving
        for PSI");
2537     fileWriter.WriteLine("# Each file contains a single line with
        averages");
2538     fileWriter.WriteLine("# for the current leaf directory.");
2539     fileWriter.WriteLine("# The single line contains the following
        elements");
2540     fileWriter.WriteLine("# Graph Type - The graph type");
2541     fileWriter.WriteLine("# Node Count - The node count");
2542     fileWriter.WriteLine("# Probability - The probability");
2543     fileWriter.WriteLine("# AvgMinHardness - The average minimum
        hardness value");

```



```

2544         fileWriter.WriteLine("#      AvgMinPSI - The average minimum PSI
                value");
2545         fileWriter.WriteLine("#      AvgMaxHardness - The average maximum
                hardness value");
2546         fileWriter.WriteLine("#      AvgHardnessTime - The average time to
                calculate hardness (in ms)");
2547         fileWriter.WriteLine("#      AvgMaxPSI - The average maximum PSI
                value");
2548         fileWriter.WriteLine("#      AvgPSITime - The average time to
                calculate PSI (in ms)");
2549         fileWriter.WriteLine("#");
2550         fileWriter.Close();
2551         output.Close();
2552     }
2553 }
2554
2555 public void CalculateProbabilityResults(string ProbabilityDir, string
        myGraphType,
2556                                     int myNodeCount)
2557 {
2558     int nRtnval = 0;
2559     string filePath = string.Empty;
2560     ResultsData rData = new ResultsData();
2561     AveragesData aData = new AveragesData();
2562
2563     //-----
2564     // If results and averages files already exist in the current
2565     // directory, delete them.
2566     //-----
2567     if (nRtnval == 0)
2568     {
2569         filePath = ProbabilityDir + "\\Presults.*";
2570         if (File.Exists(filePath))
2571         {
2572             File.Delete(filePath);
2573         }
2574         filePath = ProbabilityDir + "\\Paverages.*";
2575         if (File.Exists(filePath))
2576         {
2577             File.Delete(filePath);
2578         }
2579         filePath = ProbabilityDir + "\\Legend.txt";
2580         if (File.Exists(filePath))
2581         {
2582             File.Delete(filePath);
2583         }
2584     }

```

```

2585 //-----
2586 // Get a list of all of the probability directories in the
2587 // current directory. For each directory, find the leaf
2588 // averages for hardness and add them to the probability
2589 // results file. Also calculate the averages for all of the
2590 // probability directories and write those into the new
2591 // hardness probability averages file.
2592 //-----
2593 if (nRtnval == 0)
2594 {
2595     int nFilCnt = 0;
2596     string[] dirList = Directory.GetDirectories(ProbabilityDir);
2597     string myPath = string.Empty;
2598     resultsData.Clear();
2599     aData.AvgMinHardness = 0;
2600     aData.AvgMinPSI = 0.0;
2601     aData.AvgMaxHardness = 0;
2602     aData.AvgMaxPSI = 0.0;
2603     aData.AvgHardnessTime = 0.0;
2604     aData.AvgPSITime = 0.0;
2605     foreach (string dirname in dirList)
2606     {
2607         nFilCnt++;
2608         myPath = dirname;
2609         LoadLeafAverages(myPath, MON_HARDNESS);
2610         rData.NodeCount = 0; // Not used
2611         rData.Probability = averagesData.Probability;
2612         rData.MinHardness = averagesData.AvgMinHardness;
2613         aData.AvgMinHardness += averagesData.AvgMinHardness;
2614         rData.MinPSI = averagesData.AvgMinPSI;
2615         aData.AvgMinPSI += averagesData.AvgMinPSI;
2616         rData.MaxHardness = averagesData.AvgMaxHardness;
2617         aData.AvgMaxHardness += averagesData.AvgMaxHardness;
2618         rData.MaxPSI = averagesData.AvgMaxPSI;
2619         aData.AvgMaxPSI += averagesData.AvgMaxPSI;
2620         rData.HardnessTime = averagesData.AvgHardnessTime;
2621         aData.AvgHardnessTime += averagesData.AvgHardnessTime;
2622         rData.PSITime = averagesData.AvgPSITime;
2623         aData.AvgPSITime += averagesData.AvgPSITime;
2624         rData.FileNumber = string.Empty; // Not used
2625         resultsData.Add(new ResultsData(rData.NodeCount, rData.
2626             Probability,
2627             rData.MinHardness, rData.
2628                 MaxHardness,
2629                 rData.MinPSI, rData.MaxPSI,
2630                 rData.HardnessTime, rData.
2631                     PSITime,

```

```

2629                                     rData.FileName));
2630     }
2631     //-----
2632     // Write out the hardness results file.
2633     //-----
2634     SaveProbabilityResults(ProbabilityDir, MON_HARDNESS);
2635     if (nFilCnt > 0)
2636     {
2637         //-----
2638         // Calculate the hardness averages and write those out
2639         // to the hardness averages file.
2640         //-----
2641         averagesData.GraphType = myGraphType;
2642         averagesData.NodeCount = myNodeCount;
2643         averagesData.Probability = 0;    // Not used
2644         averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2645         averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2646         averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2647         averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
2648         averagesData.AvgHardnessTime = aData.AvgHardnessTime /
                nFilCnt;
2649         averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
2650         SaveProbabilityAverages(ProbabilityDir, MON_HARDNESS);
2651     }
2652 }
2653 //-----
2654 // Get a list of all of the probability directories in the
2655 // current directory. For each directory, find the leaf
2656 // averages for PSI and add them to the probability results
2657 // file. Also calculate the averages for all of the
2658 // probability directories and write those into the new
2659 // PSI probability averages file.
2660 //-----
2661 if (nRtnval == 0)
2662 {
2663     int nFilCnt = 0;
2664     string[] dirList = Directory.GetDirectories(ProbabilityDir);
2665     string myPath = string.Empty;
2666     resultsData.Clear();
2667     aData.AvgMinHardness = 0;
2668     aData.AvgMinPSI = 0.0;
2669     aData.AvgMaxHardness = 0;
2670     aData.AvgMaxPSI = 0.0;
2671     aData.AvgHardnessTime = 0.0;
2672     aData.AvgPSITime = 0.0;
2673     foreach (string dirname in dirList)
2674     {

```

```

2675         nFilCnt++;
2676         myPath = dirname;
2677         LoadLeafAverages(myPath, MON_PSI);
2678         rData.NodeCount = 0;           // Not used
2679         rData.Probability = averagesData.Probability;
2680         rData.MinHardness = averagesData.AvgMinHardness;
2681         aData.AvgMinHardness += averagesData.AvgMinHardness;
2682         rData.MinPSI = averagesData.AvgMinPSI;
2683         aData.AvgMinPSI += averagesData.AvgMinPSI;
2684         rData.MaxHardness = averagesData.AvgMaxHardness;
2685         aData.AvgMaxHardness += averagesData.AvgMaxHardness;
2686         rData.MaxPSI = averagesData.AvgMaxPSI;
2687         aData.AvgMaxPSI += averagesData.AvgMaxPSI;
2688         rData.HardnessTime = averagesData.AvgHardnessTime;
2689         aData.AvgHardnessTime += averagesData.AvgHardnessTime;
2690         rData.PSITime = averagesData.AvgPSITime;
2691         aData.AvgPSITime += averagesData.AvgPSITime;
2692         rData.FileName = string.Empty; // Not used
2693         resultsData.Add(new ResultsData(rData.NodeCount, rData.
                Probability,
2694                                     rData.MinHardness, rData.
                MaxHardness,
2695                                     rData.MinPSI, rData.MaxPSI,
2696                                     rData.HardnessTime, rData.
                PSITime,
2697                                     rData.FileName));
2698     }
2699     //-----
2700     // Write out the PSI results file.
2701     //-----
2702     SaveProbabilityResults(ProbabilityDir, MON_PSI);
2703     if (nFilCnt > 0)
2704     {
2705         //-----
2706         // Calculate the PSI averages and write those out to
2707         // the PSI averages file.
2708         //-----
2709         averagesData.GraphType = myGraphType;
2710         averagesData.NodeCount = myNodeCount;
2711         averagesData.Probability = 0; // Not used
2712         averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2713         averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2714         averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2715         averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
2716         averagesData.AvgHardnessTime = aData.AvgHardnessTime /
                nFilCnt;
2717         averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;

```

```

2718         SaveProbabilityAverages(ProbabilityDir, MON_PSI);
2719     }
2720 }
2721 //-----
2722 // Finally, write out the legend file.
2723 //-----
2724 if (nRtnval == 0)
2725 {
2726     string szFilename;
2727     szFilename = ProbabilityDir + "\\Legend.txt";
2728     output = new FileStream(szFilename, FileMode.Create, FileAccess.
        Write);
2729     fileWriter = new StreamWriter(output);
2730     fileWriter.WriteLine("#");
2731     fileWriter.WriteLine("# There are two results files in the
        directory");
2732     fileWriter.WriteLine("# Results.hrd - Results when solving
        for hardness");
2733     fileWriter.WriteLine("# Results.psi - Results when solving
        for PSI");
2734     fileWriter.WriteLine("# Each line in each file contains the
        following elements");
2735     fileWriter.WriteLine("# Probability - The probability level (
        percent)");
2736     fileWriter.WriteLine("# Min Hardness - The minimum hardness
        value");
2737     fileWriter.WriteLine("# Min PSI - The minimum PSI value");
2738     fileWriter.WriteLine("# Max Hardness - The maximum hardness
        value");
2739     fileWriter.WriteLine("# Hardness Time - The time to calc
        hardness value");
2740     fileWriter.WriteLine("# Max PSI - The maximum PSI value");
2741     fileWriter.WriteLine("# PSI Time - The time to calc PSI value"
        );
2742     fileWriter.WriteLine("#");
2743     fileWriter.WriteLine("# There are two averages files in the
        directory");
2744     fileWriter.WriteLine("# Paverages.hrd - Averages when solving
        for hardness");
2745     fileWriter.WriteLine("# Paverages.psi - Averages when solving
        for PSI");
2746     fileWriter.WriteLine("# Each file contains a single line with
        averages");
2747     fileWriter.WriteLine("# for the current probability directories.
        ");
2748     fileWriter.WriteLine("# The single line contains the following
        elements");

```

```

2749         fileWriter.WriteLine("#    Graph Type - The graph type");
2750         fileWriter.WriteLine("#    Node Count - The node count");
2751         fileWriter.WriteLine("#    AvgMinHardness - The average minimum
           hardness value");
2752         fileWriter.WriteLine("#    AvgMinPSI - The average minimum PSI
           value");
2753         fileWriter.WriteLine("#    AvgMaxHardness - The average maximum
           hardness value");
2754         fileWriter.WriteLine("#    AvgHardnessTime - The average time to
           calculate hardness (in ms)");
2755         fileWriter.WriteLine("#    AvgMaxPSI - The average maximum PSI
           value");
2756         fileWriter.WriteLine("#    AvgPSITime - The average time to
           calculate PSI (in ms)");
2757         fileWriter.WriteLine("#");
2758         fileWriter.Close();
2759         output.Close();
2760     }
2761 }
2762
2763 public void CalculateNodeCountResults(string NodeCountDir, string
           myGraphType)
2764 {
2765     int nRtnval = 0;
2766     string filePath = string.Empty;
2767     ResultsData rData = new ResultsData();
2768     AveragesData aData = new AveragesData();
2769
2770     //-----
2771     // If results and averages files already exist in the current
2772     // directory, delete them.
2773     //-----
2774     if (nRtnval == 0)
2775     {
2776         filePath = NodeCountDir + "\\Nresults.*";
2777         if (File.Exists(filePath))
2778         {
2779             File.Delete(filePath);
2780         }
2781         filePath = NodeCountDir + "\\Naverages.*";
2782         if (File.Exists(filePath))
2783         {
2784             File.Delete(filePath);
2785         }
2786         filePath = NodeCountDir + "\\Legend.txt";
2787         if (File.Exists(filePath))
2788         {

```

```

2789         File.Delete(filePath);
2790     }
2791 }
2792 //-----
2793 // Get a list of all of the node count directories in the
2794 // current directory. For each directory, find the probability
2795 // averages for hardness and add them to the node count
2796 // results file. Also calculate the averages for all of the
2797 // node count directories and write those into the new
2798 // hardness node count averages file.
2799 //-----
2800 if (nRtnval == 0)
2801 {
2802     int nFilCnt = 0;
2803     string[] dirList = Directory.GetDirectories(NodeCountDir);
2804     string myPath = string.Empty;
2805     resultsData.Clear();
2806     aData.AvgMinHardness = 0;
2807     aData.AvgMinPSI = 0.0;
2808     aData.AvgMaxHardness = 0;
2809     aData.AvgMaxPSI = 0.0;
2810     aData.AvgHardnessTime = 0.0;
2811     aData.AvgPSITime = 0.0;
2812     foreach (string dirname in dirList)
2813     {
2814         nFilCnt++;
2815         myPath = dirname;
2816         LoadProbabilityAverages(myPath, MON_HARDNESS);
2817         rData.NodeCount = averagesData.NodeCount;
2818         rData.Probability = 0; // Not used
2819         rData.MinHardness = averagesData.AvgMinHardness;
2820         aData.AvgMinHardness += averagesData.AvgMinHardness;
2821         rData.MinPSI = averagesData.AvgMinPSI;
2822         aData.AvgMinPSI += averagesData.AvgMinPSI;
2823         rData.MaxHardness = averagesData.AvgMaxHardness;
2824         aData.AvgMaxHardness += averagesData.AvgMaxHardness;
2825         rData.MaxPSI = averagesData.AvgMaxPSI;
2826         aData.AvgMaxPSI += averagesData.AvgMaxPSI;
2827         rData.HardnessTime = averagesData.AvgHardnessTime;
2828         aData.AvgHardnessTime += averagesData.AvgHardnessTime;
2829         rData.PSITime = averagesData.AvgPSITime;
2830         aData.AvgPSITime += averagesData.AvgPSITime;
2831         rData.FileNumber = string.Empty; // Not used
2832         resultsData.Add(new ResultsData(rData.NodeCount, rData.
                Probability,
2833                                     rData.MinHardness, rData.
                MaxHardness,

```

```

2834                                     rData.MinPSI, rData.MaxPSI,
2835                                     rData.HardnessTime, rData.
                                           PSITime,
2836                                     rData.FileNumber));
2837     }
2838     //-----
2839     // Write out the hardness results file.
2840     //-----
2841     SaveNodeCountResults(NodeCountDir, MON_HARDNESS);
2842     if (nFilCnt > 0)
2843     {
2844         //-----
2845         // Calculate the hardness averages and write those out
2846         // to the hardness averages file.
2847         //-----
2848         averagesData.GraphType = myGraphType;
2849         averagesData.NodeCount = 0;    // Not used
2850         averagesData.Probability = 0; // Not used
2851         averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2852         averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2853         averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2854         averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;
2855         averagesData.AvgHardnessTime = aData.AvgHardnessTime /
                                           nFilCnt;
2856         averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
2857         SaveNodeCountAverages(NodeCountDir, MON_HARDNESS);
2858     }
2859 }
2860 //-----
2861 // Get a list of all of the node count directories in the
2862 // current directory. For each directory, find the probability
2863 // averages for PSI and add them to the node count results
2864 // file. Also calculate the averages for all of the
2865 // node count directories and write those into the new
2866 // PSI node count averages file.
2867 //-----
2868 if (nRtnval == 0)
2869 {
2870     int nFilCnt = 0;
2871     string[] dirList = Directory.GetDirectories(NodeCountDir);
2872     string myPath = string.Empty;
2873     resultsData.Clear();
2874     aData.AvgMinHardness = 0;
2875     aData.AvgMinPSI = 0.0;
2876     aData.AvgMaxHardness = 0;
2877     aData.AvgMaxPSI = 0.0;
2878     aData.AvgHardnessTime = 0.0;

```



```

2879         aData.AvgPSITime = 0.0;
2880     foreach (string dirname in dirList)
2881     {
2882         nFilCnt++;
2883         myPath = dirname;
2884         LoadProbabilityAverages(myPath, MON_PSI);
2885         rData.NodeCount = 0;           // Not used
2886         rData.Probability = 0;        // Not used
2887         rData.MinHardness = averagesData.AvgMinHardness;
2888         aData.AvgMinHardness += averagesData.AvgMinHardness;
2889         rData.MinPSI = averagesData.AvgMinPSI;
2890         aData.AvgMinPSI += averagesData.AvgMinPSI;
2891         rData.MaxHardness = averagesData.AvgMaxHardness;
2892         aData.AvgMaxHardness += averagesData.AvgMaxHardness;
2893         rData.MaxPSI = averagesData.AvgMaxPSI;
2894         aData.AvgMaxPSI += averagesData.AvgMaxPSI;
2895         rData.HardnessTime = averagesData.AvgHardnessTime;
2896         aData.AvgHardnessTime += averagesData.AvgHardnessTime;
2897         rData.PSITime = averagesData.AvgPSITime;
2898         aData.AvgPSITime += averagesData.AvgPSITime;
2899         rData.FileNumber = string.Empty; // Not used
2900         resultsData.Add(new ResultsData(rData.NodeCount, rData.
2901             Probability,
2902             rData.MinHardness, rData.
2903             MaxHardness,
2904             rData.MinPSI, rData.MaxPSI,
2905             rData.HardnessTime, rData.
2906             PSITime,
2907             rData.FileNumber));
2908     }
2909     //-----
2910     // Write out the PSI results file.
2911     //-----
2912     SaveNodeCountResults(NodeCountDir, MON_PSI);
2913     if (nFilCnt > 0)
2914     {
2915         //-----
2916         // Calculate the PSI averages and write those out to
2917         // the PSI averages file.
2918         //-----
2919         averagesData.GraphType = myGraphType;
2920         averagesData.NodeCount = 0; // Not used
2921         averagesData.Probability = 0; // Not used
2922         averagesData.AvgMinHardness = aData.AvgMinHardness / nFilCnt;
2923         averagesData.AvgMinPSI = aData.AvgMinPSI / nFilCnt;
2924         averagesData.AvgMaxHardness = aData.AvgMaxHardness / nFilCnt;
2925         averagesData.AvgMaxPSI = aData.AvgMaxPSI / nFilCnt;

```

```

2923             averagesData.AvgHardnessTime = aData.AvgHardnessTime /
                nFilCnt;
2924             averagesData.AvgPSITime = aData.AvgPSITime / nFilCnt;
2925             SaveNodeCountAverages(NodeCountDir, MON_PSI);
2926         }
2927     }
2928     //-----
2929     // Finally, write out the legend file.
2930     //-----
2931     if (nRtnval == 0)
2932     {
2933         string szFilename;
2934         szFilename = NodeCountDir + "\\Legend.txt";
2935         output = new FileStream(szFilename, FileMode.Create, FileAccess.
                Write);
2936         fileWriter = new StreamWriter(output);
2937         fileWriter.WriteLine("#");
2938         fileWriter.WriteLine("# There are two results files in the
                directory");
2939         fileWriter.WriteLine("# Nresults.hrd - Results when solving
                for hardness");
2940         fileWriter.WriteLine("# Nresults.psi - Results when solving
                for PSI");
2941         fileWriter.WriteLine("# Each line in each file contains the
                following elements");
2942         fileWriter.WriteLine("# Node Count - The node count value");
2943         fileWriter.WriteLine("# Min Hardness - The minimum hardness
                value");
2944         fileWriter.WriteLine("# Min PSI - The minimum PSI value");
2945         fileWriter.WriteLine("# Max Hardness - The maximum hardness
                value");
2946         fileWriter.WriteLine("# Hardness Time - The time to calc
                hardness value");
2947         fileWriter.WriteLine("# Max PSI - The maximum PSI value");
2948         fileWriter.WriteLine("# PSI Time - The time to calc PSI value"
                );
2949         fileWriter.WriteLine("#");
2950         fileWriter.WriteLine("# There are two averages files in the
                directory");
2951         fileWriter.WriteLine("# Naverages.hrd - Averages when solving
                for hardness");
2952         fileWriter.WriteLine("# Naverages.psi - Averages when solving
                for PSI");
2953         fileWriter.WriteLine("# Each file contains a single line with
                averages");
2954         fileWriter.WriteLine("# for the current node count directories."
                );

```

```

2955         fileWriter.WriteLine("# The single line contains the following
                elements");
2956         fileWriter.WriteLine("# Graph Type - The graph type");
2957         fileWriter.WriteLine("# AvgMinHardness - The average minimum
                hardness value");
2958         fileWriter.WriteLine("# AvgMinPSI - The average minimum PSI
                value");
2959         fileWriter.WriteLine("# AvgMaxHardness - The average maximum
                hardness value");
2960         fileWriter.WriteLine("# AvgHardnessTime - The average time to
                calculate hardness (in ms)");
2961         fileWriter.WriteLine("# AvgMaxPSI - The average maximum PSI
                value");
2962         fileWriter.WriteLine("# AvgPSITime - The average time to
                calculate PSI (in ms)");
2963         fileWriter.WriteLine("#");
2964         fileWriter.Close();
2965         output.Close();
2966     }
2967 }
2968
2969 //*****
2970 // Miscellaneous Methods.
2971 //*****
2972 public void ClearNodeArray()
2973 {
2974     int i;
2975     int j;
2976
2977     for (i = 0; i < MAX_NODES; i++)
2978     {
2979         for (j = 0; j < MAX_NODES; j++)
2980         {
2981             nodeArray[i, j] = 0;
2982         }
2983     }
2984 }
2985
2986 public void CopyGraphToArray()
2987 {
2988     int i, j;
2989     int nCount;
2990     int nDegree;
2991     int nNode;
2992
2993     //-----
2994     // Clear the node array.

```

```

2995 //-----
2996 ClearNodeArray();
2997
2998 //-----
2999 // For each entry in the node list, add the edges to the array.
3000 //-----
3001 nCount = nodeList.Count();
3002 for (i = 0; i < nCount; i++)
3003 {
3004     //-----
3005     // Look at each 'edge' slot for this node. For each that
3006     // has a value in the legal range we will add the edges
3007     // to the node array.
3008     //-----
3009     nDegree = nodeList[i].DegreeCount;
3010     for (j = 0; j < nDegree; j++)
3011     {
3012         nNode = nodeList[i].GetEdgeID(j);
3013         if ((nNode >= 0) && (nNode < nCount))
3014         {
3015             nodeArray[i, nNode] = 1;
3016             nodeArray[nNode, i] = 1;
3017         }
3018     }
3019 }
3020 }
3021
3022 public void SolveForColor(int nColorCount)
3023 {
3024     int i = 0;
3025     int nCount = 0;
3026     int nCurDegree = 0;
3027     int nEdgeOffset = 0;
3028     int nConflict = 0;
3029     int nColor = 0;
3030     int nDone = 0;
3031     int nSolved = 0;
3032
3033     //-----
3034     // Initialize our node count and current depth in the node
3035     // list. Do some validation to make sure we have at least
3036     // one node and at least 2 colors.
3037     //-----
3038     nCount = nodeList.Count();
3039     nCurDepth = 0;
3040     this.ConflictCount = 0;
3041     if ((nColorCount < 2) || (nCount < 1))

```

```

3042     {
3043         nSolved = -1;
3044     }
3045     //-----
3046     // Loop through the node list and set the current color for
3047     // each node to 0, which means no color is assigned yet.
3048     //-----
3049     if (nSolved == 0)
3050     {
3051         foreach (NodeElement ne in nodeList)
3052         {
3053             ne.NodeColor = 0;
3054         }
3055     }
3056     //-----
3057     // Set the initial color for the first node, then go into our
3058     // processing loop.
3059     //-----
3060     nodeList[nCurDepth].NodeColor = 1;
3061     while ((nSolved == 0) && (nDoneFlag == 0))
3062     {
3063         //-----
3064         // Check for a color conflict with our current node.
3065         //-----
3066         nCurDegree = nodeList[nCurDepth].DegreeCount;
3067         nConflict = 0;
3068         for (i = 0; (i < nCurDegree) && (nConflict == 0); i++)
3069         {
3070             nEdgeOffset = nodeList[nCurDepth].GetEdgeID(i);
3071             if (nodeList[nCurDepth].NodeColor == nodeList[nEdgeOffset].
3072                 NodeColor)
3073             {
3074                 nConflict = 1;
3075             }
3076         }
3077         //-----
3078         // If we have a color conflict, increment the color for
3079         // the current node. If we have tried all colors for this
3080         // node, start back tracking until we find a node that
3081         // hasn't had all colors tried. If we've tried all colors
3082         // for all nodes in the backtracking chain, then there is
3083         // no solution to color this graph.
3084         //-----
3085         if (nConflict == 1)
3086         {
3087             nConflictCount++;
3088             nDone = 0;

```

```

3088         while ((nDone == 0) && (nSolved == 0))
3089         {
3090             nColor = nodeList[nCurDepth].NodeColor;
3091             nColor++;
3092             if (nColor > nColorCount)
3093             {
3094                 nodeList[nCurDepth].NodeColor = 0;
3095                 if (nCurDepth > 0)
3096                 {
3097                     nCurDepth--;
3098                 }
3099                 else
3100                 {
3101                     nSolved = -1;
3102                 }
3103             }
3104             else
3105             {
3106                 nodeList[nCurDepth].NodeColor = nColor;
3107                 nDone = 1;
3108             }
3109         }
3110     }
3111     else
3112     {
3113         //-----
3114         // There is no color conflict for the current node.
3115         // Move down to the next level and try that one. If
3116         // we've reached the bottom level (leaf node) then
3117         // we've found a solution for the problem.
3118         //-----
3119         nCurDepth++;
3120         if (nCurDepth >= nCount)
3121         {
3122             nSolved = 1;
3123         }
3124         else
3125         {
3126             nodeList[nCurDepth].NodeColor = 1;
3127         }
3128     }
3129 }
3130 if (nSolved == 1)
3131 {
3132     this.ColorSolvable = true;
3133 }
3134 else

```

```

3135     {
3136         this.ColorSolvable = false;
3137     }
3138 }
3139
3140 public void SolveForPSI()
3141 {
3142     //-----
3143     // This function takes the graph represented in the nodeArray
3144     // matrix where a value of 1 represents an edge and 0 no edge.
3145     // The function returns a double that represents the complexity
3146     // of the graph (which should be higher when more information
3147     // is contained in the graph). We assume that the graph has
3148     // no self connections.
3149     //-----
3150     int n = this.NodeCount;
3151     int[,] c_i = new int[n, 2]; // the count of connections to a single
3152     node. the second index is not to i ,0] or connections to i ,1]
3153     double[,] p_i = new double[n, 2]; //as above but probabilities
3154     int[, , ,] c_ij = new int[n, n, 2, 2]; //count of connections between
3155     two nodes through another node, last two indexes similar to
3156     above
3157     double[, , ,] p_ij = new double[n, n, 2, 2]; // as above but
3158     probabilities
3159
3160     // fill in p_i by looking at connectivity between each i and all the
3161     other nodes
3162     nStep = 1;
3163     myActivity = "Fill in p_i based on node connectivity";
3164     for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
3165         for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3166             for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3167                 if (i != j && nodeArray[i, j] == connect)
3168                     c_i[i, connect]++;
3169     // fill in c_ij by looking at connectivity between i and j through
3170     all other nodes
3171     nStep = 2;
3172     myActivity = "Fill in c_ij based on node connectivity";
3173     for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0);
3174         connect_i++)
3175         for (int connect_j = 0; (connect_j < 2) && (nDoneFlag == 0);
3176             connect_j++)
3177             for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3178                 for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3179                     for (int m = 0; (m < n) && (nDoneFlag == 0); m++) //
3180                         middle node

```

```

3172         if (i != j && i != m && j != m && nodeArray[i, m]
3173             == connect_i && nodeArray[m, j] == connect_j
3174             )
3175             c_ij[i, j, connect_i, connect_j]++;
3176 // convert to probabilities by dividing by the total number of
3177 // possibilities
3178 // (taking into account that each node must be distinct)
3179 nStep = 3;
3180 myActivity = "Convert to Probabilities - part 1";
3181 for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
3182     for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3183         p_i[i, connect] = c_i[i, connect] / (double)(n - 1); // n-1
3184 // because connecting to self is not an option
3185 // convert to probabilities by dividing by the total n-2 because
3186 // middle node must not be start or end
3187 nStep = 4;
3188 myActivity = "Convert to Probabilities - part 2";
3189 for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0);
3190     connect_i++)
3191     for (int connect_j = 0; (connect_j < 2) && (nDoneFlag == 0);
3192         connect_j++)
3193         for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3194             for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3195                 p_ij[i, j, connect_i, connect_j] = c_ij[i, j,
3196                 connect_i, connect_j] / (double)(n - 2);
3197 // compute the shannon entropy
3198 nStep = 5;
3199 myActivity = "Compute the Shannon Entropy";
3200 double[] k = new double[n];
3201 for (int connect = 0; (connect < 2) && (nDoneFlag == 0); connect++)
3202     for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3203         if (p_i[i, connect] != 0)
3204             k[i] += -1 * p_i[i, connect] * Math.Log(p_i[i, connect],
3205             2);
3206 // compute the marginalized pi and pj (sum probabilities over other
3207 // variable)
3208 nStep = 6;
3209 myActivity = "Compute Marginalized pi and pj";
3210 double[, ,] p_ijMa = new double[n, n, 2];
3211 double[, ,] p_ijMb = new double[n, n, 2];
3212 for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3213     for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3214         if (i != j)
3215             {
3216                 // sum over b (second index) for Ma
3217                 p_ijMa[i, j, 0] += p_ij[i, j, 0, 0] + p_ij[i, j, 0, 1];
3218                 p_ijMa[i, j, 1] += p_ij[i, j, 1, 0] + p_ij[i, j, 1, 1];

```



```

3209             // sum over a (first index) for Mb
3210             p_ijMb[i, j, 0] += p_ij[i, j, 0, 0] + p_ij[i, j, 1, 0];
3211             p_ijMb[i, j, 1] += p_ij[i, j, 0, 1] + p_ij[i, j, 1, 1];
3212         }
3213         // compute the mutual information m[]
3214         nStep = 7;
3215         myActivity = "Compute the Mutual Information";
3216         double[,] mI = new double[n, n];
3217         // for each connection compute the mutual information
3218         for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3219             for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3220                 for (int connect_i = 0; (connect_i < 2) && (nDoneFlag == 0);
3221                     connect_i++)
3222                     for (int connect_j = 0; (connect_j < 2) && (nDoneFlag ==
3223                         0); connect_j++)
3224                         if (p_ij[i, j, connect_i, connect_j] != 0.0 && p_i[i,
3225                             connect_i] != 0.0 && p_i[j, connect_j] != 0.0)
3226                             mI[i, j] += p_ij[i, j, connect_i, connect_j] *
3227                                 Math.Log(p_ij[i, j, connect_i, connect_j]
3228                                     / (p_ijMa[i, j, connect_i] * p_ijMb[i, j,
3229                                         connect_j]), 2);
3230
3231         // compute psi
3232         nStep = 8;
3233         myActivity = "Compute PSI";
3234         double psi = 0.0, sum = 0.0;
3235         for (int i = 0; (i < n) && (nDoneFlag == 0); i++)
3236             for (int j = 0; (j < n) && (nDoneFlag == 0); j++)
3237                 if (i != j)
3238                     sum += Math.Max(k[i], k[j]) * mI[i, j] * (1 - mI[i, j]);
3239         psi = 4 * sum / (n * (n - 1));
3240         PSICALCULATEDVALUE = psi;
3241     }
3242
3243     public void LogDebug(string szDebug)
3244     {
3245         string szFilename;
3246
3247         //-----
3248         // Open the file and append the results to it.
3249         //-----
3250         szFilename = DirectoryName + "\\Debug.txt";
3251         output = new FileStream(szFilename, FileMode.Append, FileAccess.Write
3252             );
3253         fileWriter = new StreamWriter(output);
3254         fileWriter.WriteLine(szDebug);
3255
3256         //-----

```

```

3250         // Close the file.
3251         //-----
3252         fileWriter.Close();
3253         output.Close();
3254     }
3255 }
3256 }

```

A.4 Calculate Averages Data (AveragesData.cs)

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class AveragesData
9     {
10         private string szGraphType = string.Empty;
11         private int nNodeCount = 0;
12         private int nProbability = 0;
13         private long nAvgMinHardness = 0;
14         private double fAvgMinPSI = 0.0;
15         private long nAvgMaxHardness = 0;
16         private double fAvgHardnessTime = 0.0;
17         private double fAvgMaxPSI = 0.0;
18         private double fAvgPSITime = 0.0;
19
20         public AveragesData()
21         {
22         }
23
24         public AveragesData(string myGraphType, int myNodeCount,
25                             int myProbability, long myAvgMinHardness,
26                             double myAvgMinPSI, long myAvgMaxHardness,
27                             double myAvgHardnessTime, double myAvgMaxPSI,
28                             double myAvgPSITime)
29         {
30             this.szGraphType = myGraphType;
31             this.nNodeCount = myNodeCount;
32             this.nProbability = myProbability;

```

```
33         this.nAvgMinHardness = myAvgMinHardness;
34         this.fAvgMinPSI = myAvgMinPSI;
35         this.nAvgMaxHardness = myAvgMaxHardness;
36         this.fAvgHardnessTime = myAvgHardnessTime;
37         this.fAvgMaxPSI = myAvgMaxPSI;
38         this.fAvgPSITime = myAvgPSITime;
39     }
40
41     public string GraphType
42     {
43         get { return szGraphType; }
44         set { szGraphType = value; }
45     }
46
47     public int NodeCount
48     {
49         get { return nNodeCount; }
50         set { nNodeCount = value; }
51     }
52
53     public int Probability
54     {
55         get { return nProbability; }
56         set { nProbability = value; }
57     }
58
59     public long AvgMinHardness
60     {
61         get { return nAvgMinHardness; }
62         set { nAvgMinHardness = value; }
63     }
64
65     public double AvgMinPSI
66     {
67         get { return fAvgMinPSI; }
68         set { fAvgMinPSI = value; }
69     }
70
71     public long AvgMaxHardness
72     {
73         get { return nAvgMaxHardness; }
74         set { nAvgMaxHardness = value; }
75     }
76
77     public double AvgHardnessTime
78     {
79         get { return fAvgHardnessTime; }
```

```

80         set { fAvgHardnessTime = value; }
81     }
82
83     public double AvgMaxPSI
84     {
85         get { return fAvgMaxPSI; }
86         set { fAvgMaxPSI = value; }
87     }
88
89     public double AvgPSITime
90     {
91         get { return fAvgPSITime; }
92         set { fAvgPSITime = value; }
93     }
94 }
95 }

```

A.5 Edge Class (EdgeElement.cs)

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class EdgeElement
9     {
10         private int nNodeID;
11         private bool bEdgeChecked;
12
13         public EdgeElement()
14         {
15         }
16
17         public EdgeElement(int nID)
18         {
19
20             //-----
21             // Set the ID of the node that this edge connects to.
22             //-----
23             this.nNodeID = nID;

```

```

24     }
25
26     public int NodeID
27     {
28         get { return nNodeID; }
29         set { nNodeID = value; }
30     }
31
32     public bool IsChecked
33     {
34         get { return bEdgeChecked; }
35         set { bEdgeChecked = value; }
36     }
37 }
38 }

```

A.6 Hill Climber Results (HillClimbResult.cs)

This class tracks results from the hill climber algorithm.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class HillClimbResult
9     {
10         private int nColorCount = 0;
11         private bool bColorIsSolvable = false;
12         private long nConflictCount = 0;
13         private double fHardnessTime = 0.0;
14         private double fPSIValue = 0.0;
15         private double fPSITime = 0.0;
16
17         public HillClimbResult()
18         {
19         }
20
21         public HillClimbResult(int myColorCount, bool myColorIsSolvable,
22                                 long myConflictCount, double myHardnessTime,
23                                 double myPSIValue, double myPSITime)
24         {

```

```
25         this.nColorCount = myColorCount;
26         this.bColorIsSolvable = myColorIsSolvable;
27         this.nConflictCount = myConflictCount;
28         this.fHardnessTime = myHardnessTime;
29         this.fPSIValue = myPSIValue;
30         this.fPSITime = myPSITime;
31     }
32
33     public int ColorCount
34     {
35         get { return nColorCount; }
36         set { nColorCount = value; }
37     }
38
39     public bool ColorIsSolvable
40     {
41         get { return bColorIsSolvable; }
42         set { bColorIsSolvable = value; }
43     }
44
45     public long ConflictCount
46     {
47         get { return nConflictCount; }
48         set { nConflictCount = value; }
49     }
50
51     public double HardnessTime
52     {
53         get { return fHardnessTime; }
54         set { fHardnessTime = value; }
55     }
56
57     public double PSIValue
58     {
59         get { return fPSIValue; }
60         set { fPSIValue = value; }
61     }
62
63     public double PSITime
64     {
65         get { return fPSITime; }
66         set { fPSITime = value; }
67     }
68 }
69 }
```

A.7 Node Class (NodeElement.cs)

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class NodeElement
9     {
10         private int nNodeID;
11         private int nNodeColor;
12         private int nNodeDegree;
13         private bool bNodeVisited;
14         private bool bNodeChecked;
15         private IList<EdgeElement> edgeList = new List<EdgeElement>();
16
17         public NodeElement()
18         {
19         }
20
21         public NodeElement(int nID, int nColor)
22         {
23
24             //-----
25             // Set the ID and color for the node. Then clear the edge
26             // list to show no edges.
27             //-----
28             this.nNodeID = nID;
29             this.nNodeColor = nColor;
30             this.nNodeDegree = 0;
31             this.bNodeVisited = false;
32             this.edgeList.Clear();
33         }
34
35         public int NodeID
36         {
37             get { return nNodeID; }
38             set { nNodeID = value; }
39         }
40
41         public int NodeColor

```

```

42     {
43         get { return nNodeColor; }
44         set { nNodeColor = value; }
45     }
46
47     public int DegreeCount
48     {
49         get { return nNodeDegree; }
50         set { nNodeDegree = value; }
51     }
52
53     public bool IsVisited
54     {
55         get { return bNodeVisited; }
56         set { bNodeVisited = value; }
57     }
58
59     public bool IsChecked
60     {
61         get { return bNodeChecked; }
62         set { bNodeChecked = value; }
63     }
64
65     public int AddEdge(int nCompanionNode)
66     {
67         int nRtnval = 0;
68         bool bFound = false;
69
70         //-----
71         // Search the edge list to see if we are already connected to
72         // the target node. If so, this method does nothing.
73         // Otherwise we will make the connection.
74         //-----
75         foreach (EdgeElement ee in edgeList)
76         {
77             if (ee.NodeID == nCompanionNode)
78             {
79                 bFound = true;
80             }
81         }
82         if (bFound == false)
83         {
84             edgeList.Add(new EdgeElement(nCompanionNode));
85             this.DegreeCount++;
86         }
87         return (nRtnval);
88     }

```



```
89
90     public bool IsConnected(int nNextNode)
91     {
92         bool bRtnval = false;
93
94         foreach (EdgeElement ee in edgeList)
95         {
96             if (ee.NodeID == nNextNode)
97             {
98                 bRtnval = true;
99             }
100        }
101        return (bRtnval);
102    }
103
104    public int GetEdgeID(int nOffset)
105    {
106        int nRtnval = -1;
107        int nCount = 0;
108
109        nCount = edgeList.Count();
110        if ((nOffset >= 0) && (nOffset < nCount))
111        {
112            nRtnval = edgeList[nOffset].NodeID;
113        }
114        return (nRtnval);
115    }
116
117    public bool IsEdgeChecked(int nOffset)
118    {
119        bool bRtnval = false;
120        int nCount = 0;
121
122        nCount = edgeList.Count();
123        if ((nOffset >= 0) && (nOffset < nCount))
124        {
125            bRtnval = edgeList[nOffset].IsChecked;
126        }
127        return (bRtnval);
128    }
129
130    public void MarkEdgeChecked(int nOffset, bool bChecked)
131    {
132        int nCount = 0;
133
134        nCount = edgeList.Count();
135        if ((nOffset >= 0) && (nOffset < nCount))
```

```
136         {
137             edgeList[nOffset].IsChecked = bChecked;
138         }
139     }
140
141     public int FindEdgeOffset(int nEdgeID)
142     {
143         int i;
144         int nCount = 0;
145         int nOffset = -1;
146
147         nCount = edgeList.Count();
148         for (i = 0; (i < nCount) && (nOffset == -1); i++)
149         {
150             if (edgeList[i].NodeID == nEdgeID)
151             {
152                 nOffset = i;
153             }
154         }
155         return (nOffset);
156     }
157
158     public void RemoveEdge(int nEdgeID)
159     {
160         int nOffset = -1;
161
162         nOffset = this.FindEdgeOffset(nEdgeID);
163         if (nOffset != -1)
164         {
165             this.DegreeCount--;
166             if (this.DegreeCount < 0)
167             {
168                 this.DegreeCount = 0;
169             }
170             edgeList.RemoveAt(nOffset);
171         }
172     }
173
174     public int FindNextNode(int myEdgeOff)
175     {
176         int nNextNodeID;
177
178         nNextNodeID = this.edgeList[myEdgeOff].NodeID;
179         return (nNextNodeID);
180     }
181
182     public void ClearEdgeList()
```

```

183     {
184         this.edgeList.Clear();
185     }
186 }
187 }

```

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.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class RestorePoint
9     {
10         //-----
11         // Define constants.
12         //-----
13         public const int ACTION_ADD = 0;
14         public const int ACTION_REMOVE = 1;
15         //-----
16         // Define data elements for the class.
17         //-----
18         private int nNode1ID;
19         private int nNode2ID;
20         private int nAction;
21
22         public RestorePoint()
23         {
24         }
25
26         public RestorePoint(int myNode1, int myNode2, int myAction)
27         {
28
29             //-----

```

```

30         // Set the ID of the 2 nodes involved and the action that
31         // was taken.
32         //-----
33         this.nNode1ID = myNode1;
34         this.nNode2ID = myNode2;
35         this.nAction = myAction;
36     }
37
38     public int Node1ID
39     {
40         get { return nNode1ID; }
41         set { nNode1ID = value; }
42     }
43
44     public int Node2ID
45     {
46         get { return nNode2ID; }
47         set { nNode2ID = value; }
48     }
49
50     public int Action
51     {
52         get { return nAction; }
53         set { nAction = value; }
54     }
55 }
56 }

```

A.9 Results Data (ResultsData.cs)

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

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace GraphandPSISolver
7 {
8     class ResultsData
9     {
10         private int nNodeCount = 0;
11         private int nProbability = 0;
12         private long nMinHardness = 0;

```

```
13     private long nMaxHardness = 0;
14     private double fMinPSI = 0.0;
15     private double fMaxPSI = 0.0;
16     private double fHardnessTime = 0.0;
17     private double fPSITime = 0.0;
18     private string szFileNumber = string.Empty;
19
20     public ResultsData()
21     {
22     }
23
24     public ResultsData(int myNodeCount, int myProbability,
25                       long myMinHardness, long myMaxHardness,
26                       double myMinPSI, double myMaxPSI,
27                       double myHardnessTime, double myPSITime,
28                       string myFileNumber)
29     {
30         this.nNodeCount = myNodeCount;
31         this.nProbability = myProbability;
32         this.nMinHardness = myMinHardness;
33         this.nMaxHardness = myMaxHardness;
34         this.fMinPSI = myMinPSI;
35         this.fMaxPSI = myMaxPSI;
36         this.fHardnessTime = myHardnessTime;
37         this.fPSITime = myPSITime;
38         this.szFileNumber = myFileNumber;
39     }
40
41     public int NodeCount
42     {
43         get { return nNodeCount; }
44         set { nNodeCount = value; }
45     }
46
47     public int Probability
48     {
49         get { return nProbability; }
50         set { nProbability = value; }
51     }
52
53     public string FileNumber
54     {
55         get { return szFileNumber; }
56         set { szFileNumber = value; }
57     }
58
59     public long MinHardness
```

```
60     {
61         get { return nMinHardness; }
62         set { nMinHardness = value; }
63     }
64
65     public double MinPSI
66     {
67         get { return fMinPSI; }
68         set { fMinPSI = value; }
69     }
70
71     public long MaxHardness
72     {
73         get { return nMaxHardness; }
74         set { nMaxHardness = value; }
75     }
76
77     public double HardnesTime
78     {
79         get { return fHardnesTime; }
80         set { fHardnesTime = value; }
81     }
82
83     public double MaxPSI
84     {
85         get { return fMaxPSI; }
86         set { fMaxPSI = value; }
87     }
88
89     public double PSITime
90     {
91         get { return fPSITime; }
92         set { fPSITime = value; }
93     }
94 }
95 }
```
