EXPLORING THE UNIQUE GAMES CONJECTURE

by

RYAN DARRAS

B.I., University of Colorado, 2016

A thesis submitted to the Graduate Faculty of the

University of Colorado Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2021

This thesis for the Master of Science degree by

Ryan Darras

has been approved for the

Department of Computer Science

by

Sudhanshu Semwal, Chair

Albert Chamillard

Yanyan Zhuang

Date March 4, 2021

Darras, Ryan (M.S., Computer Science)

Exploring the Unique Games Conjecture

Thesis directed by Professor Sudhanshu K. Semwal

ABSTRACT

In 2002, mathematician and theoretical computer scientist Subhash Khot proposed the Unique Games Conjecture (UGC) [1] which has been a point of contention amongst theoretical computer scientists as they try to prove or disprove its validity. The Unique Games Conjecture is so theoretically interesting and so, in this thesis, we are going to explore its practice using a set of problems called Constraint Satisfaction Problems (CSPs). CSPs are problems that we interact with almost every day. CSPs have been studied under optimization research for a long time, and in 1950s-60s CSPs found academic focus as possibly better approximate algorithms that can usually save time and money. Most CSPs were classified as NP-complete and NP-Hard in the seventies, and since then computationally tractable approximate solutions has come about with the possibility of solving these using machine learning algorithms. CSPs are relevant and can directly impact and improve daily situations. We design a random game to study three CSPs to gain a better understanding of Khot's Conjecture or UGC.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Everybody loves Amazon, the modern age option to shop in the comfort of home and have the items shipped directly to our doorsteps. However, many of us might not understand the complexity behind managing such a massive operation. The traveling salesperson problem is a common lesson to learn in the study of Analysis and NP-Completeness yet may dwarf in comparison to what Amazon and other distributors must deal with in practice. In the United States alone, Amazon delivers millions of packages per day throughout thousands of cities. Each delivery driver works a scheduled shift anywhere from roughly four to ten hours a day, with many cities managing multiple drivers. Every single route needs to be planned as optimally as it can to have the availability of two-day shipping. The scheduler also needs to consider shipping between warehouses that are on the opposite ends of the country. Without researchers designing effective algorithms to increase the efficiency of this massive problem, we would not have the luxury of having our online purchases appearing on our doorstep in just a few days of the order.

Due to the potential NP-Hardness of many constraint satisfaction problems (CSPs), they have been a subject of profound study in both artificial intelligence and operations research. Often, new NP-Hard problems contain or can be derived from existing NP-Complete and NP-Hard problems to provide a research area for more specified problems that severely impact the real world. A solution to these problems can be found by using branch-and-bound (BnB) algorithms. BnB algorithms are a very common tool when solving NP-Hard problems due to their nature to manage an exhaustive search effectively which will provide every possible answer, and not tracking solution which are sub-optimal. Exhaustive BnB algorithms are NP-Complete or NP-Hard, so they can be heavily optimized by implementing a heuristic which determines if a branch cannot be a potential solution, in which case the branch is pruned. A BnB algorithm that prunes branches like this is commonly referred to as a branch-and-cut (BnC) algorithm. These heuristics are generally obtained by studying the given problem and using insight, logic, and reason to determine the most optimal values and weights.

Khot’s Unique Games Conjecture (UGC) states that it may not be possible to find an efficient solution to some problems regardless of how accurate your heuristics are, or how good your algorithm is. The UGC also states that it might not even be possible to find an efficient approximation to some of these problems. In this thesis, we want to explore why this may be the case and why the UGC has remained unverified for so long.

Khot’s Unique Games Conjecture

Khot’s Conjecture proposes that it is NP-hard to even approximate the value of the Unique Game [1]. The value of a Unique Game is the largest fraction of constraints that can be satisfied by any state of the problem; such that a given problem that is satisfiable has a value of 1 and a fully unsatisfiable problem has a value of 0. In Khot’s original paper, he refers to the unique 2-prover 1-round game which is an instance of a Unique Game if for every question and every answer by the first player, there is exactly one answer from the second player that results in a win for the players, and vice versa. The value of the game is the maximum winning probability for all the players over all strategies. What makes Khot's Conjecture so interesting is that solving for the value of a satisfiable problem is relatively simple. Once you find a satisfied state of the problem, the answer is 1 and you are done. Some problems have efficient approximation algorithms that will converge on a solution state very quickly, however, trying to find the value of an un-satisfiable problem is much more complicated because some algorithms will diverge into an exhaustive search if no solution is found.

Khot's Uniqueness Games Conjecture (UGC) provides us an opportunity to fundamentally focus on Graph Coloring problem throughout our proposed research work. We are also interested in Khot's UGC implications on practice of developing approximate algorithms such as Constraint Satisfaction Problems.

Constraint Satisfaction Problems

CSPs consist of a set of objects V, each with their own variables, and a set of constraints E on the variables of the objects. To solve a CSP a state V must be found that satisfies every e ε E. CSPs often require a combination of heuristics and search algorithms to solve due to their exponential complexity of NP-Completeness[1]. BnB algorithms are commonly used for solving NP-Complete problems due to its nature of searching a state space. A BnB algorithm uses

[1] An answer to NP-Complete problems can be verified in polynomial time (quickly) but an optimized answer can only be solved or provided in exponential time (very slowly). Because of this, NP-Complete problems are often solved using approximation algorithms.

heuristics to optimize the upper and lower bounds, reducing the total amount of space searched to optimize efficiency. However, heuristic variables are a challenge due to the complexity of the problems which prevents us from being able to easily determine the optimal values. We will be implementing a genetic algorithm that will search for optimal heuristic values to speed up our BnB algorithms.

Purpose of this Research

Reasonable (approximate) solutions for a CSP can be useful for many projects and goals that countless people run into on a day-to-day basis. This research will look at a different way to solve these problems by providing a way to search the solution space.

In the past, software engineers and software developers relied on Moore’s Law to increase the range of possibilities with faster hardware. However, Moore’s Law is coming to an end[2], and hardware improvements may not see the same level of speedup as we have the past half-century. Due to this, the responsibility of writing fast and effective algorithms falls directly on the software engineers and software developers themselves. By looking into Khot’s UGC, we hope to better understand why solving these problems is incredibly difficult.

A genetic algorithm has been used to find a sub-optimal or optimal solution to a CSP and opened a logical pathway for many other NP-Hard[3]

[2] Gordon Moore
[3] The set of NP-Hard problems has solutions that can be used to derive the solutions to problems in NP in polynomial time.

problems to be studied with a way to algorithmically generate states to test for optimality, and possibly converging on one solution and solving the problem. We provide more insight into Khot's Conjecture in this way.

The Focus Problems

There is an insurmountable amount of CSPs and even more sub-problems that have derived from them. Due to this, it is impossible to apply this research to every single CSP so we have pulled a subset of three problems that can be used as an entry point, and give us a good breadth of data that should be representative of constraint satisfaction problems as a whole.

0-1 Knapsack Problem

The 0-1 knapsack problem is defined by having a knapsack with a given weight limit W. Given a collection of items N each with a value v and weight w, the goal of this problem is to select items to put in the knapsack that results in the highest possible total value. While the 0-1 knapsack problem is the most common, there are many derivations of the knapsack problem such as the unbounded knapsack problem, multi-objective knapsack problem, multi-dimensional knapsack problem, and the multiple knapsack problem.

Item	0	1	2	3	4
Weight (w)	1	4	1	12	2
Value (v)	3	7	2	9	8
v/w	3	1.75	2	0.75	4

Table 1. 0-1 Knapsack problem example

Graph Coloring Problem

The simplest form of the graph coloring problem focuses on coloring vertices of a graph in such a way that no two adjacent vertices share the same color. However, the graph coloring problem is also an example of dynamic constraint satisfaction problem due to the additional constraints that can be added into the problem, as explained below.

The graph coloring problem can be separated into three different problems: vertex coloring where no two adjacent vertices share a color, edge coloring where no two adjacent edges share a color, and face coloring where no face on a planar graph shares a boundary with another face of the same color. Not only that, but the problem can be altered to consider alternative constraints such as: blue cannot be adjacent to green, or red must only be adjacent to purple or yellow. We can apply the graph coloring technique to an uncolored map to determine the smallest number of colors we need, as well as to solve a configuration of colors to color the map. Another use of the graph coloring problem, for example, would be to match *n* number of candidates with *n* number of jobs where each candidate has the appropriate qualifications for the job they are assigned.

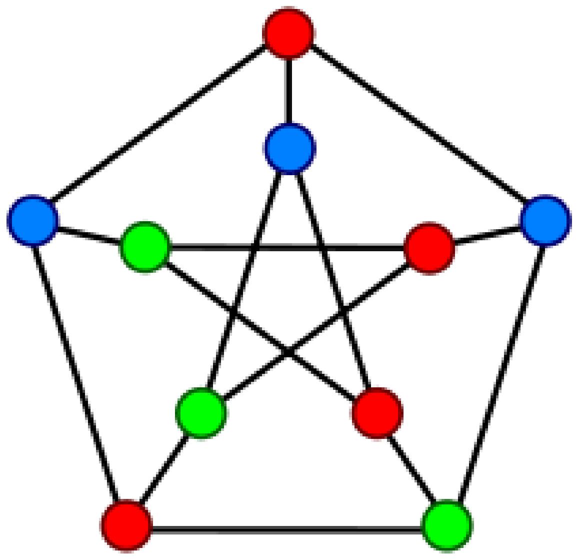
Figure 1. A solution for vertex coloring of the Petersen graph

Khot’s Uniqueness Games Conjecture [20] provides us motivation to use the instance of Graph Coloring problem. Considering the 3-coloring problem as an example, we have possible $3^{|V|}$ combination for set V of vertices. Khot’s conjecture is related to approximate algorithms [22-25] but all k-coloring problem solutions are $O(k^{|V|})$. The open question is: Is there a polynomial time algorithm for k-Graph Coloring problem? [20]. Starting with some informal definitions: A computational problem A reduces (polynomial time transformed) to a computational problem B, if there is a way to *encode* an instance of problem A to an instance of problem B (under reasonable encoding scheme), so that a polynomial algorithm for B implies a polynomial time algorithm for A as well [20, 26]. What Khot’s conjecture suggests is that some computational problems are so hard that apart from being impossible to solve in polynomial time unless P=NP, it is also impossible to get a good polynomial time approximation algorithm. This fact provides the motivation of this thesis: to investigate the k-coloring problem as the starting point for our research as well. For the graph size n=100 (for now) it is assumed that k^{100} will be large enough that the 100-coloring problem is not solvable in polynomial time by any computational algorithm, and k <50 might just be the limit for a human working without a computer in this case. We plan to start with Khot's conjecture that there are Graph Coloring instances of graphs, some of which can be solved and other are not solvable in polynomial time by a computer. In addition, it is difficult to differentiate between the two instances algorithmically, which means computationally it is difficult (NP-complete/NP-Hard) for a machine to determine that as well.

The Random Game

We developed what we call The Random Game to see The Unique Games Conjecture's statements firsthand. We will use The Random Game to gain a better understanding of why The Unique Games Conjecture is such a significant problem and create a simplification of the conjecture that makes it more easily understood.

The Random Game consists of a graph G with edges E and vertices V. Each vertex in V can be assigned a color of red, green, or blue and each edge in E is assigned a random constraint from the following:

- Connected nodes must be different colors.
- Connected nodes must be the same color.
- 2-3 constraints saying that one node must be a certain color, and the other node must be the other. If any of these 2-3 constraints pass, this constraint passes.

- 2-3 constraints saying that the connected nodes must not be two specific colors. If any of these 2-3 constraints fail, this constraint fails.

The goal is to color each vertex in V so that the graph satisfies all the constraints on the edges in E. We chose three colors and the constraints to simplify the problem so we could generate a large number of results.

There are branch-and-bound style algorithms out there that can solve this game, but it is called the random game because we want to attempt to solve it "randomly". By this, we mean that we want to see how long it would take to solve this problem by simply randomly coloring the nodes as this would demonstrate The Unique Games Conjecture. However, The Unique Games Conjecture mentions that while it is easy to verify and solve for some instances of The Unique Game, it is NP-Hard to verify some instances that does not have a solution that satisfies the constraints on the edges in E because this would mean exhaustive search will ultimately fail as well so trying one state-space at a time remains exhaustive as verification will imply checking all possible state-spaces. We can also, instead of generating state by state, possibly solve for the entirety of the possible states of V given the constraints on the edges in E. With the entire state space of V, we can determine how many states satisfy the constraints. More the number of such states more will be the probability of finding that in our random game.

See below an example with the following constraints:

- Edge 0 must be different colors.
- Edge 1 must be either red/blue, green/green, or red/green.
- Edge 2 must either be green/red or green/blue.
- Edge 3 must be the same color.

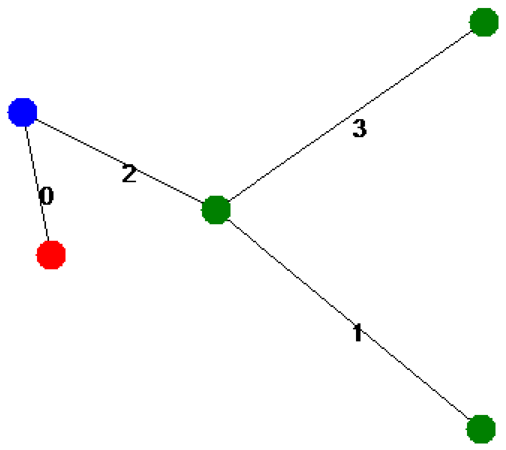

Figure 2. A satisfied state of V

In the previous example, we can see that given the constraints on each of the edges there exists a solution that satisfies all the constraints on the edges in E.

CHAPTER II

REVIEW OF THE RELEVENT TERMS

The following sections are definitions of some of the terms used in our work: heuristics and genetic algorithms.

Heuristics

A heuristic is a method for obtaining a solution. Constraint Satisfaction Problems are faced with the problem of not being able to find the most optimal solution in polynomial time so heuristic could allow a way to find a solution in a systematic, and/or sometimes random manner and terminate after certain polynomial time or steps. Taking a step further, Burke et al.[4] have surveyed the topic of Hyper-Heuristics, “heuristics to choose heuristics” or “a search method or learning mechanism for selecting or generating heuristics to solve computational search problems”.

Selecting heuristics is a difficult problem. The scientific communities understanding regarding why certain heuristics work well, or do not work well, does not create a simple solution when selecting heuristics. Because of this, there is a lack of guidance as to how to select the heuristics to use for any given problem. Such hyper-heuristics attempt to determine the most effective heuristic to use, which results in a more efficient solution to the problem.

These hyper-heuristics can be broken down in to the two main categories of: heuristic selection and heuristic generation. We will be applying the heuristic

[4] Burke et al.

generation category to each of our focus problems in hopes that we may determine effective heuristics for solving these problems.

Similarly, meta-heuristics are heuristics that are designed to eventually fit the case for many problems. Genetic algorithms are a good way to implement meta-heuristics as they use evolution and mutation to escape the known problem domain knowledge. D.F. Jones et al. surveys meta-heuristics [3] for problems that have multiple objects and argue that meta-heuristics are powerful considering they can handle integer variables, discrete variables, and/or logical variables which allows for applications against many problems. Non-standard goals, constraints, objectives, and conditions can be easily incorporated into a meta-heuristic which even further demonstrates their flexibility.

Our research could accelerate algorithm development when using heuristics for CSP such as Graph Coloring and understanding Khot's UGC. We also develop relationships and correlations between heuristics and evaluation metrics that should help researchers narrow down to more effective, accurate results.

CHAPTER III

IMPLEMENTATION

The objective is to make our implementation as generic and adaptable as possible with an eye to understand the UGC, meaning implementing new constraint satisfaction problems and solving for their heuristic variables should be as easy as possible. A generic solution would allow this framework to be implemented in the form of a NuGet package which would make this project easy to use for other researchers.

Framework

This project is written in C# and uses tools and features from Visual Studio. We provide an interface that allows for running the program on our focus problems so we can easily test outputs vs inputs.

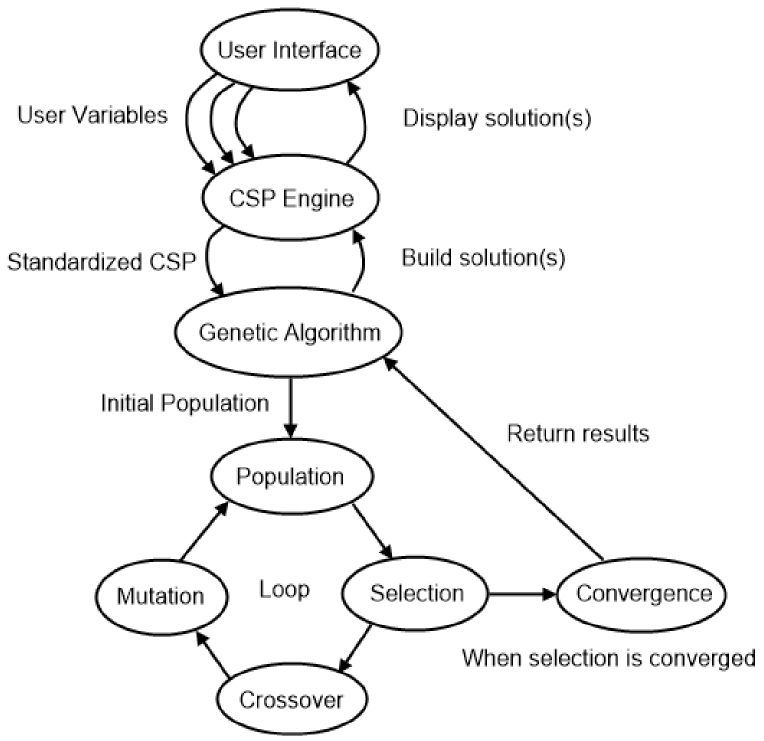

Figure 3. Genetic algorithm model

Genetic Algorithm

The simplest form of a genetic algorithm creates a population of chromosomes where the chromosomes is an array of bits. These bits are toggled on or off throughout the genetic algorithms runtime in hopes to converge on an optimal solution. While this was a potential option, we chose to go with a different route. An alternate form of a genetic algorithm creates a population of chromosomes where the chromosomes are any type of mutable value. By organizing our chromosomes into an array of mutable values we can develop our algorithm to where it modifies weights that are used to calculate fitness.

The algorithm takes the following approach to converge on a result:

1. Generate initial population with a set of default chromosomes and said chromosomes mutated the initial amount.

2. Calculate the fitness of each chromosome in the population by passing the fitness delegate for the specific constraint satisfaction problem we are considering.

3. Determine if the population has converged to similar fitness values. If so, return the result.

4. Selection: Keep the chromosomes with the highest fitness and dispose of the remaining chromosomes.

5. Crossover: Randomly selects a crossover point valued between 1 and n - 1 where n is the number of genes in a chromosome. Then take two chromosomes and split them at that point to form a new child chromosome. These new children are combined with the parents from the previous generation form the new generation.

6. Mutation: At this point we have a new and complete generation, so we randomly select a set of chromosomes to mutate. For each of those chromosomes, we randomly select the set of individual genes to mutate, and then mutate them. The chromosome selection chance, gene selection chance, and mutation amount are all values between 0 and 1 that can be adjusted. 0 means 0% chance for selection, or a maximum mutation amount of 0%. 1 means 100% chance for selection, or a maximum mutation amount of 100%. A

maximum mutation amount of 100% means that the value in our gene can at maximum double, and at a minimum mutate to 0, or anywhere in between. Pseudo code below; underlined <u>0-1</u> means a randomly generated value between 0 and 1:

```
for (Chromosome c in Chromosomes)
  if (0-1 <= chromosomeSelectionChance)
    for (Gene g in c)
      if (0-1 <= geneSelectionChance)
        g = g+(g*(((0-1)*2)-1)*mutationVar)
```

7. Go to step 2 and repeat.

After convergence, we return all unique solutions to the problem that were in the convergence threshold. With these solutions, we can either select the best one, or view all solutions as a set to extrapolate data.

Chromosome

In some genetic algorithms, a chromosome is an array of bits that are toggled on or off based on the state of the genetic algorithm. In our algorithm, chromosomes are arrays of mutable values, namely integers and floating-point values. Each individual value is called a gene, and it is used within the fitness algorithm to determine which chromosomes yield the most accurate results.

Population

The population is the collection of chromosomes used when populating the genetic algorithm. What makes our implementation of the genetic algorithm relatively unique is that our population size must be of a certain format:

size = x + y, where x = y(y-1)/2

This formula forces the size of our population into a form where x is the number of children chromosomes generated each generation and y is the number of parent chromosomes that were the y fittest chromosomes of the previous generation. This formula allows us to have every single parent breed with every other parent, giving us strong coverage for each new generation.

```
for (int i = 0; i < Chromosomes.Count - 1; i++)
{
    for (int j = i + 1; j < Chromosomes.Count; j++)
    {
        object[] newGenes = new object[GeneCount];
        for (int x = 0; x < crossoverPoint; x++)
            newGenes[x] = Chromosomes[i].Genes[x];
        for (int y = crossoverPoint; y < GeneCount; y++)
            newGenes[y] = Chromosomes[j].Genes[y];
        chromosomesToAdd.Add(new Chromosome(newGenes));
    }
}
```

Mapping to Algorithm implementations

The following sections will explain, in detail, specific information about how each problem was implemented. They will include information about the heuristic variables used in the chromosomes of the genetic algorithm, as well as details about the fitness algorithm used to determine which chromosomes are superior.

How the genetic algorithm is applied so that the solution instances are being created to test if they satisfy the solution or not will also be defined, which includes termination criteria such as a maximum number of loops or iterations.

0-1 Knapsack Problem

When solving the 0-1 knapsack problem, the variables we need to consider include weight and value. The objective is to obtain the highest available value within the capacity constraints of the knapsack. Traditional techniques for estimating the solution to this problem include finding the ratio R between weight W and value V; specifically, R = V / W.

The greedy algorithm takes the largest R values until the maximum capacity will be overwhelmed to approximate the optimal solution. We will be following this approach, but will instead be using a heuristic to determine the value of R. Our heuristic will be the combination of priority of high/low weight and value. We assume that our genetic algorithm will push our heuristic to prioritize low weight, and high value more than high weight and low value.

We formatted the chromosomes in the following form:

- [0] = priority of low weight
- [1] = priority of high weight
- [2] = priority of low value
- [3] = priority of high value

Our fitness algorithm is as follows:

```
Items = Items.OrderByDescending(item =>
    (MaximumWeight - item.Weight) * Convert.ToSingle(genes[0]) +
    item.Weight * Convert.ToSingle(genes[1]) +
    (MaximumValue - item.Value) * Convert.ToSingle(genes[2]) +
    item.Value * Convert.ToSingle(genes[3])
    ).ToArray();

List<KnapsackItem> inBag = new List<KnapsackItem>();
foreach (KnapsackItem item in Items)
    if (inBag.Sum(t => t.Weight) + item.Weight < Capacity)
        inBag.Add(item);
return inBag.Sum(t => t.Value);
```

Our fitness algorithm first orders the available items to add to the knapsack in descending order based on the chromosome's values. The sorting algorithm is broken into 4 parts:

1. "MaximumWeight - item.Weight * [0]" allows the first gene to directly influence positively when the items weight is low.

2. "item.Weight * [1]" allows for the second gene to directly influence positively when the items weight is high.

3. "MaximumValue - item.Value * [2]" allows for the third gene to directly influence positively when the items value is low.

4. "item.Value * [3]" allows for the fourth gene to directly influence positively when the items value is high.

When summed, these four values can be used to sort the available items in such a way that when following the greedy algorithm for the 0-1 knapsack problem all we must do is take from the items list if we have the available space.

Once we have fit all that we can into the knapsack, we simply return the total value of all the items in the knapsack as the overall fitness score.

Graph Coloring Problem

Graph Coloring can be broken down into many subproblems, but we opted to focus on the most basic form, vertex coloring. Vertex coloring is a way of coloring the vertices of a graph such that no two adjacent vertices are the same color. Vertex coloring is commonly used for practical applications such as drawing political maps, because political maps need to show distinct differences between boundaries as to prevent confusion.

We will be using a branch-and-cut algorithm to solve this problem, and we will be specifically cutting branches due to the four-color theorem. In 1890, British mathematician Percy John Heawood proved that any political map could be colored using at a maximum five different colors. His work was based off an attempt at the four-color proof by mathematician Alfred Kempe. It was not until 1976 that Kenneth Appel and Wolfgang Haken finally proved the four-color theorem using computers. Despite the proof, there were many doubts due to the fact that the computer-assisted proof was impossible for a human to verify which led to continued research on the topic which resulted in additional, simpler computer-assisted proofs in the following decades. Any branch that gets to the point where it would need to add a fifth color to be valid will be cut and ignored from any further processing.

Before writing an algorithm to solve the Graph Coloring Problem, we needed to write an algorithm that would generate a variable limited graph to test on. The variables that we used to generate these graphs were: numNodes, minEdgesPerNode, and maxEdgesPerNode. With these variables in mind, our graph generating algorithm took the following steps:

- Generate n nodes at random locations on the plane.
- Continuously iterate over the nodes, moving them away from any nodes they that are too close to. This prevents overlapping nodes.
- For every node, we determined how many neighbors it wanted based on a random number between the minEdgesPerNode and maxEdgesPerNode and stored these “desired neighbors” in a collection to use later.
- Using the desired neighbor’s collection, we make an edge between pairs of nodes that had a common interest in being neighbors.
- For all the desired neighbors that were not common interest, we evaluated whether completing these edges resulted in any node going over maxEdgesPerNode neighbors and deny the connection if so.
- Any remaining “unhappy” nodes that could not connect to their expected number of neighbors are iterated through and connected to each other within reason.

Below are example graphs generated from this algorithm:

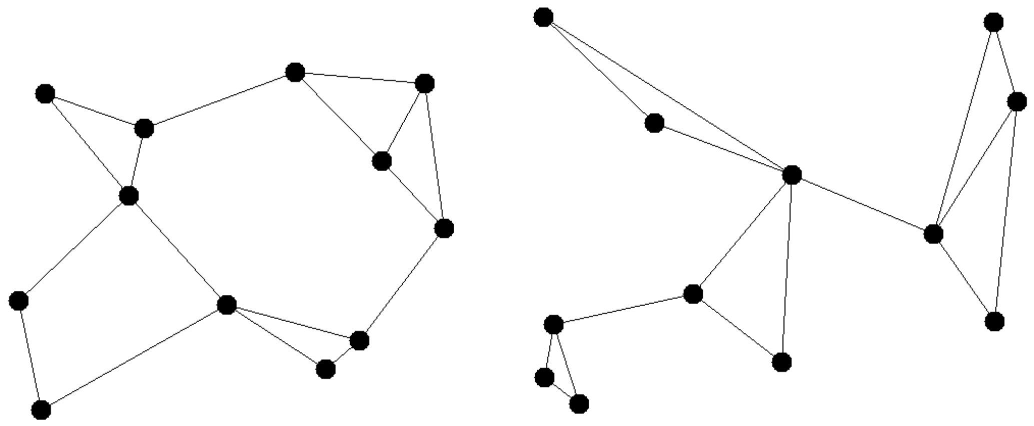

Figure 4. Twelve nodes, 25% density

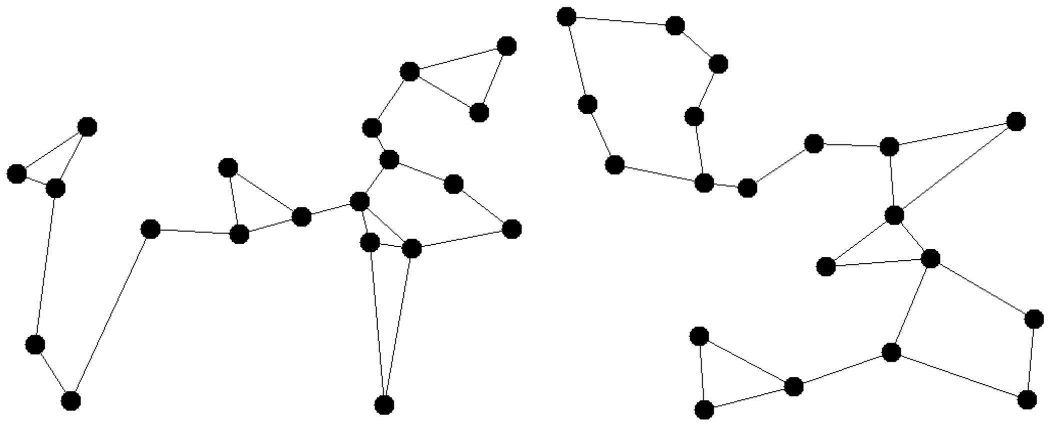

Figure 5. Twenty Nodes, 10% edge density

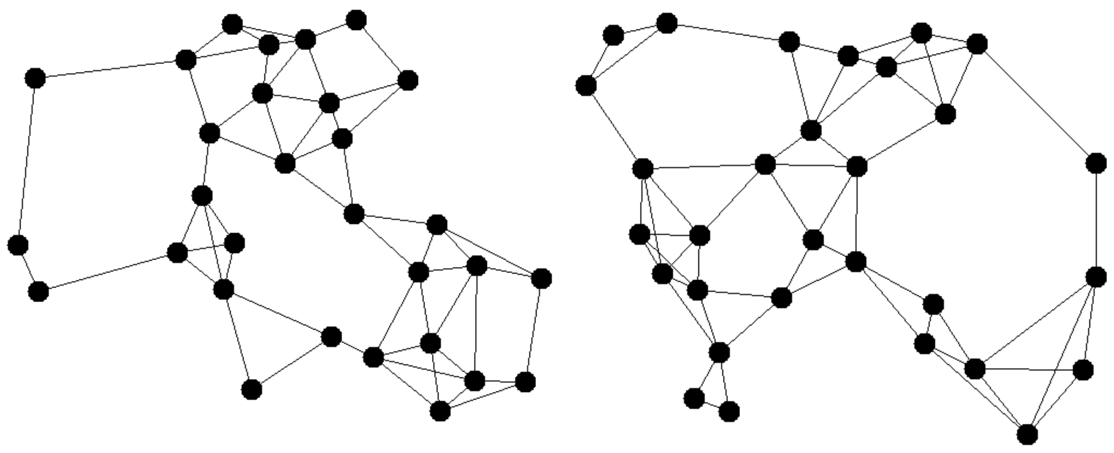

Figure 6. Thirty Nodes, 25% edge density

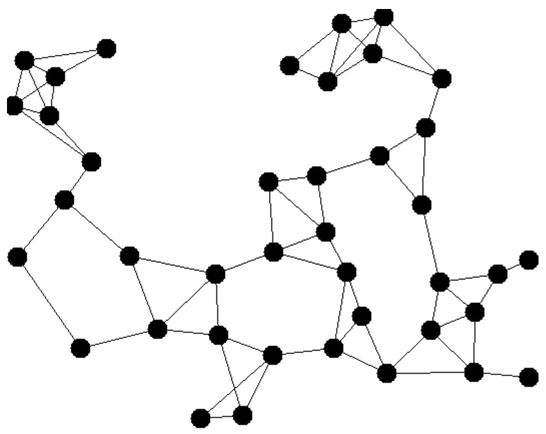

Figure 7. Forty Nodes, 10% density

With the graph generating algorithm, we can generate a variety of graphs to test on. We will be using variations of graphs to determine if the heuristic variables generated by the genetic algorithm are similar on graphs with different characteristics.

The algorithm is a very simple branch-and-cut algorithm that stores all leaf nodes into a priority queue based on the heuristic values generated by the genetic algorithm; each of these leaf nodes are partially completed graphs. Originally, we planned on using a heuristic to determine how many branches should be made off any given leaf but quickly realized that any value above two resulted in an explosion of algorithmic complexity. To keep the algorithmic complexity manageable, the tree has been limited to two branches per node, or to a binary tree. To check a leaf node, each uncolored vertex in the leaf is valued using another set of heuristics and the two top prioritized are applied and added to the tree. After generations have improved the heuristics, the optimizations produced a tree that looked more like a linked list, which means the algorithm found a solution very quickly without having to search other spaces.

Our heuristics on the graphs themselves were as follows:

- [0] = priority of the total number of current colors
- [1] = priority on number of colored nodes in the graph
- [2] = priority on total number of edges neighboring an uncolored node

Our heuristics on the nodes in each graph were as follows:

- [3] = priority on total number of uncolored neighbors
- [4] = priority on the node's degree

The fitness functions for the graph solving algorithm are as follows:

- Graphs – [0] * c + [1] * u + [2] * b where c is number of colors currently used in the graph, u is the number of colored nodes in the graph, and b is the total number of edges on the graph that have an uncolored vertex.
- Nodes – The sums of the normalized [3] and normalized [4] which allows the two variables to be used beside each other without being completely overpowered in certain circumstances.

The fitness function for the genetic algorithm is simply the number of paths it took down the tree. Where n is the best possible outcome and $2^n – 1$ is the worst possible outcome, so preferring the least number of paths first. We opted for this as the fitness for the genetic algorithm because using metrics like time are unstable due to factors out of our control.

The Random Game

The same algorithm as described in the previous section is used for generating the graph coloring map to create the initial map for The Random Game.

Then, to fully understand what edge constraints passed and fail on the entire set of graph coloring possibilities we wrote an algorithm that recursively creates every single map possibility with the given colors. This method limited us in terms of graph size and color possibilities because the exponential growth of the problem expanded so rapidly that we could not fit complex graphs in the available memory. However, we could still test reliably with three colors and graphs of 5-6 nodes. The following is the algorithm used to create every graph:

```
public List<Graph> ExhaustiveSolve(Color[] possibleColors)
{
    List<Graph> results = new List<Graph>();
    foreach (var node in Nodes)
    {
        if (node.Color != Color.Black)
            continue;
        foreach (var color in possibleColors)
        {
            Graph newGraph = new Graph(this);
            newGraph.Nodes[Nodes.IndexOf(node)].Color = color;
            results.AddRange(newGraph.ExhaustiveSolve(possibleColors));
        }
    }
    if (Nodes.All(t => t.Color != Color.Black))
    {
        results.Add(this);
    }
    return results;
}
```

This recursive algorithm creates every possible graph coloration by simply applying each color to a node on a new graph and calling itself recursively with the rule that if the node is already colored, do not recurse.

With the possible graph colorings loaded, we could then randomly generate constraints for each edge that we would test against. The following algorithm was used to generate these constraints:

```
for (int i = 0; i < Graphs[0].GetAllEdges().Count; i++)
{
  int selection = rand.Next(1, 5); //1-4 corresponding to the constraint methods
  switch (selection)
  {
    case 1:
      constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("", new
        List<Func<Graph, int, bool>>() { (g, edge) =>
        Constraint_MustBeDifferentColor(g, edge) });
      break;
    case 2:
      constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("", new
        List<Func<Graph, int, bool>>() { (g, edge) =>
        Constraint_MustBeSameColor(g, edge) });
      break;
    case 3:
      List<Func<Graph, int, bool>> mustBeOneOfEachCons = new List<Func<Graph, int,
        bool>>();
      for (int j = 0; j < rand.Next(2, 4); j++) //rand 2 or 3
      {
        Color a = possibleColors[rand.Next(0, possibleColors.Count)];
        Color b = possibleColors[rand.Next(0, possibleColors.Count)];
        mustBeOneOfEachCons.Add((g, edge) =>
          Constraint_MustBeOneOfEach(g, edge, a, b));
      }
      constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("OR",
        mustBeOneOfEachCons);
      break;
    case 4:
      List<Func<Graph, int, bool>> mustNotBeOneOfEachCons =
        new List<Func<Graph, int, bool>>();
      for (int j = 0; j < rand.Next(2, 4); j++) //rand 2 or 3
      {
        Color a = possibleColors[rand.Next(0, possibleColors.Count)];
        Color b = possibleColors[rand.Next(0, possibleColors.Count)];
        mustNotBeOneOfEachCons.Add((g, edge) =>
          Constraint_MustNotBeOneOfEach(g, edge, a, b));
      }
      constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("AND",
        mustNotBeOneOfEachCons);
      break;
  }
}
```

This algorithm simply adds a single type of constraint to each edge. However, some constraint types will apply multiple constraints. For example, if the constraint is that the two nodes must not be green and red, the algorithm might also apply another constraint saying that the two nodes must not be green or blue either.

We then ran the constraints across every single-colored graph to see how many constraints failed, and how many constraints passed to get our data. We wanted to look at more than just pass/fail data, so for each graph we determined the percentage of constraints on the edges that passed in total so we could get a better understanding of how many graphs fully failed, fully passed, or somewhere in between. The following algorithm was used to check each graph:

```
foreach (var graph in Graphs)
{
  int passCount = 0;
  int failCount = 0;
  for (int i = 0; i < graph.GetAllEdges().Count; i++)
  {
    if (constraints[i].Item1 == "")
    {
      if (constraints[i].Item2[0](graph, i))
        passCount++;
      else
        failCount++;
    }
    else if (constraints[i].Item1 == "OR")
    {
      if (constraints[i].Item2.Any(t => t(graph, i)))
        passCount++;
      else
        failCount++;
    }
    else if (constraints[i].Item1 == "AND")
    {
      if (constraints[i].Item2.All(t => t(graph, i)))
        passCount++;
      else
        failCount++;
    }
  }
  results.Add(new Tuple<int, int>(passCount, failCount));
}
```

This algorithm sums the total number of constraints that were passed, and the total number of constraints that failed for the given coloration by testing each constraint individually.

At this point we have all the data showing Khot’s definition of “value” for each graph given the constraints. Regarding Khot’s Conjecture, the value is the highest number we generated on any single graph. So, if we had any single graph hit a value of 1, meaning it satisfied every single constraint, then we could have stopped the algorithm because we knew it was a satisfiable problem.

CHAPTER

DATA & RESULTS

0-1 Knapsack Problem

As expected, the Genetic Algorithm based heuristics to solve 0-1 Knapsack Problem were simple, and the results were as expected being that genetic algorithm decided that high value is good, where high weight is not. All our tests placed a low priority on large weight and small value, and a high priority on small weight and large value.

We created a knapsack with 300 capacity and randomly generated 300 objects with weights and value between 10-20. Our max R (value/weight) was 1.889 with a value of 19.318 and a weight of 10.229. Our min R (value/weight) was .528 with a value of 10.042 and a weight of 19.025.

All tests used a population size of 528 and a convergence requirement of within 5% fitness, but the genetic algorithm variables used to select which chromosomes and genes were passed on or mutated were modified to discover how these elements effect the results.

% Chromosomes Mutated	50%
% Genes Mutated	50%
Maximum % Mutation Deviation	25%

Table 2. Balanced genetic algorithm settings

We started with the above variables as we felt it was a reasonable starting point with no clear outliers that would heavily affect the data. Essentially, all genes were obtained from the previous generation and 25% of them were mutated by up to 25% of their current value. We felt that this allowed us to keep the good genes, yet variation of a 25% mutation allowed us to see if we found an improvement. Below are the top five heuristic values generated by the genetic algorithm and the total value that was obtained from the knapsack.

Min Weight	Max Weight	Min Value	Max Value	Total Value
0.937	0.334	0.394	1	490.279
1	0.17	0.267	0.972	488.994
1	0.294	0.284	0.878	488.303
1	0.153	0.267	0.92	485.735
1	0.361	0.248	0.629	484.262

Table 3. Balanced approach

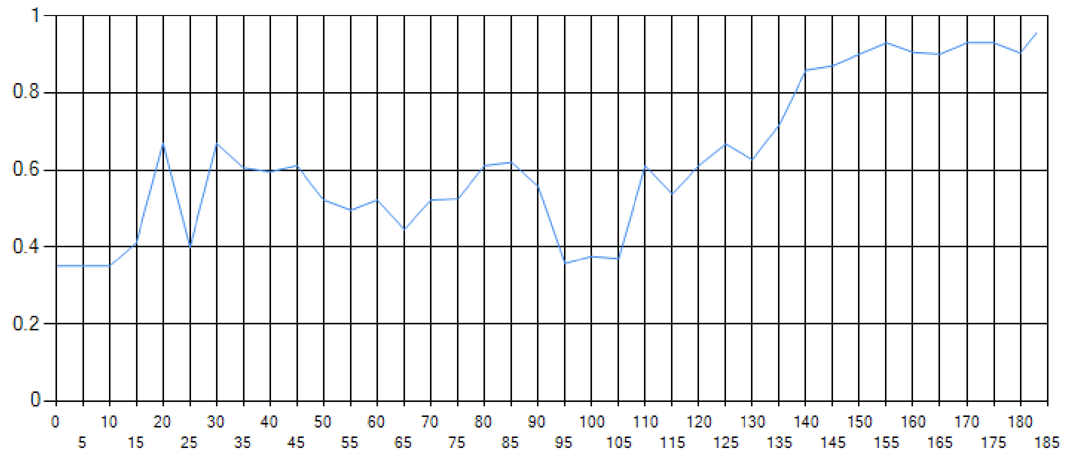

Figure 8. Convergence of the balanced genetic algorithm

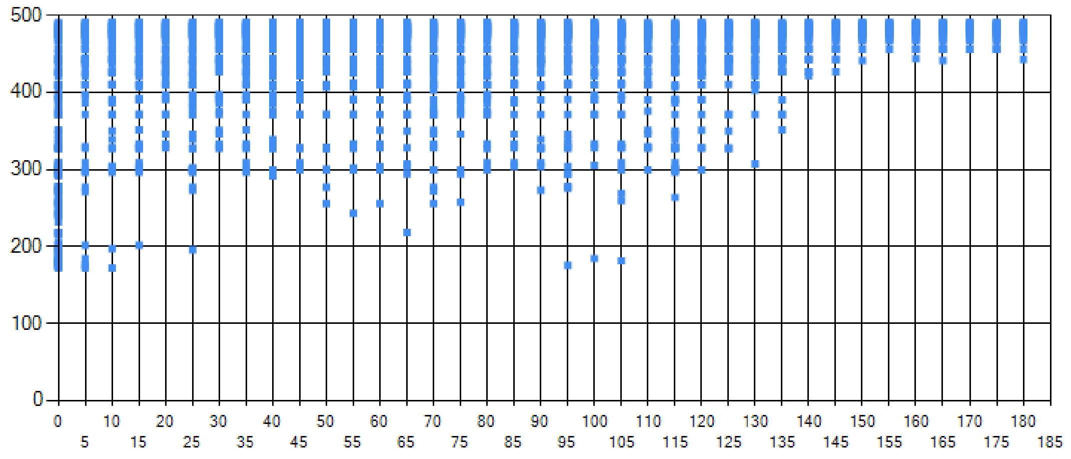

Figure 9. Fitness of each parent for the balanced genetic algorithm

Next, we ran the same simulation with the same knapsack and knapsack objects but we wanted to test a very liberal approach, but we quickly noticed that when being too liberal (100% on everything) we ran into an issue where the mutations got out of hand to the point where passing genes hardly mattered because they were immediately mutated. Hence, we tried the following:

% Chromosomes Mutated	100%
% Genes Mutated	100%
Maximum % Mutation Deviation	25%

Table 4. Liberal genetic algorithm settings

These results were impressive -- by offering every single gene the ability to mutate every generation, you open doors that would otherwise be shut by forcing 50% of the chromosomes to exist as-is, as a direct descendant of two chromosomes from the previous generation and only actually mutating 50% of the genes of the chromosomes that were selected to mutate. Below are the results, which are surprisingly similar when it comes to the actual goal of the algorithm, but what we find most interesting is the heuristic values.

Min Weight	Max Weight	Min Value	Max Value	Total Value
1	0.015	0.039	0.976	490.279
1	0.006	0.04	0.89	488.994
1	0.006	0.032	0.854	488.303
1	0.007	0.043	0.837	485.735
1	0.007	0.021	0.35	485.472

Table 5. Liberal approach

Judging by the common theme of the minimum weight being the most heavily prioritized heuristic variable we think that this was due to our 300 randomly generated knapsack objects were generally heavier than the middle of the allowed generation weight range, while it was likely that the values were generally lower than the middle of the allowed generation value range. So, we calculated the average weight and value of our 300 objects and discovered that our average weight was 15.06 and our average value was 14.79, thus, our theory that our randomly generated objects were on the heavier side, and lower value side was correct.

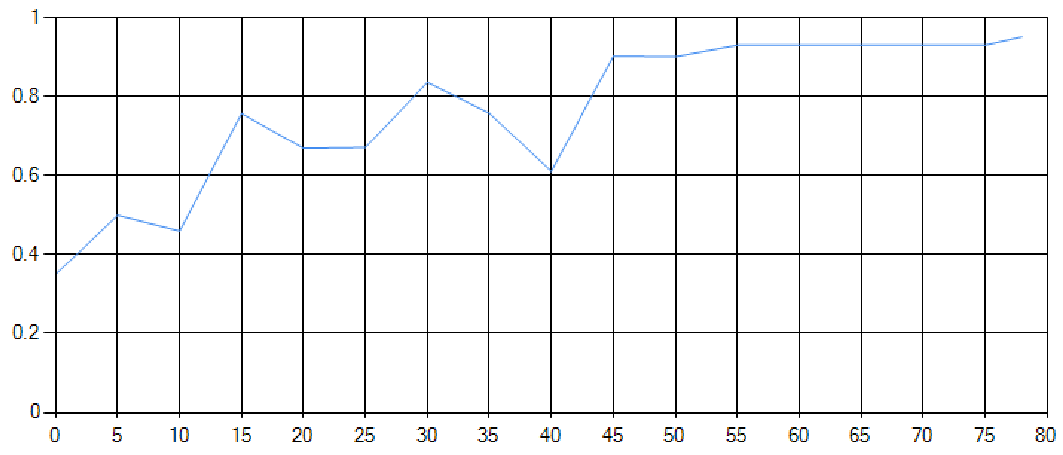

Figure 10. Convergence of the liberal genetic algorithm

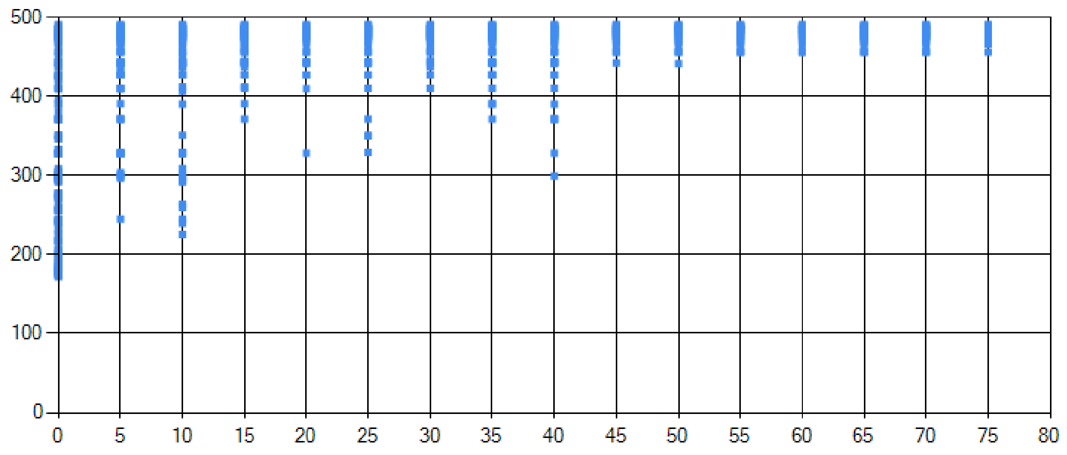

Figure 11. Fitness of each parent for the liberal genetic algorithm

Finally, after trying the liberal strategy we wanted to try a more conservative approach to mutating the genes. We were surprised that it converged with similar results even faster than the liberal approach!

% Chromosomes Mutated	25%
% Genes Mutated	25%
Maximum % Mutation Deviation	25%

Table 6. Conservative genetic algorithm settings

By only selecting 25% of the population to mutate, and only selecting 25% of each chromosome’s genes, we only ended up mutating 6.25% of the total gene pool. However, the algorithm still converged quickly because it was given the ability to test the new generations mutations directly against the previous generations genes whereas the liberal approach was more so just trying new things as much as it could.

Min Weight	Max Weight	Min Value	Max Value	Total Value
0.956	0.624	0.672	1	490.279
0.956	0.618	0.714	1	488.994
1	0.618	0.62	0.936	488.303
1	0.771	0.643	0.811	485.735
1	0.711	0.703	0.787	485.472

Table 7. Conservative approach

The conservative approach generated the exact same solutions but had wildly different heuristic variables to do so.

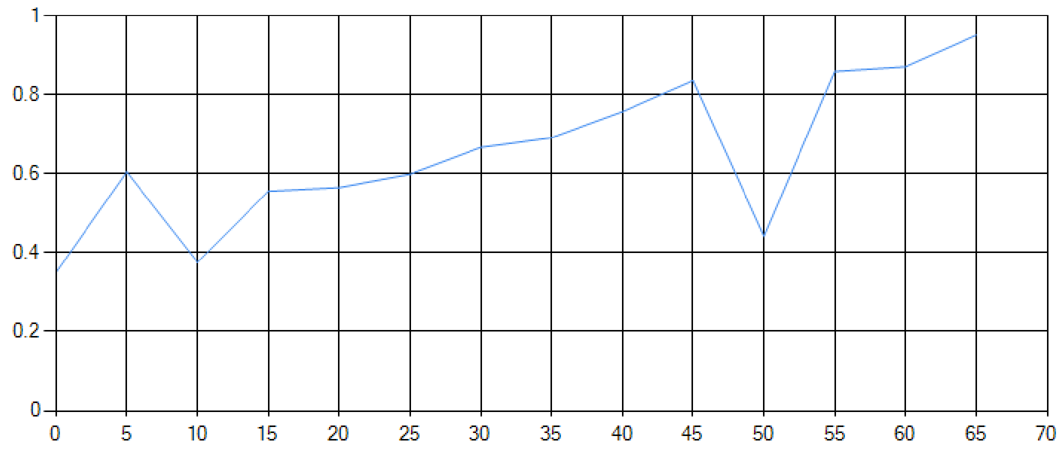

Figure 12. Convergence of the conservative genetic algorithm

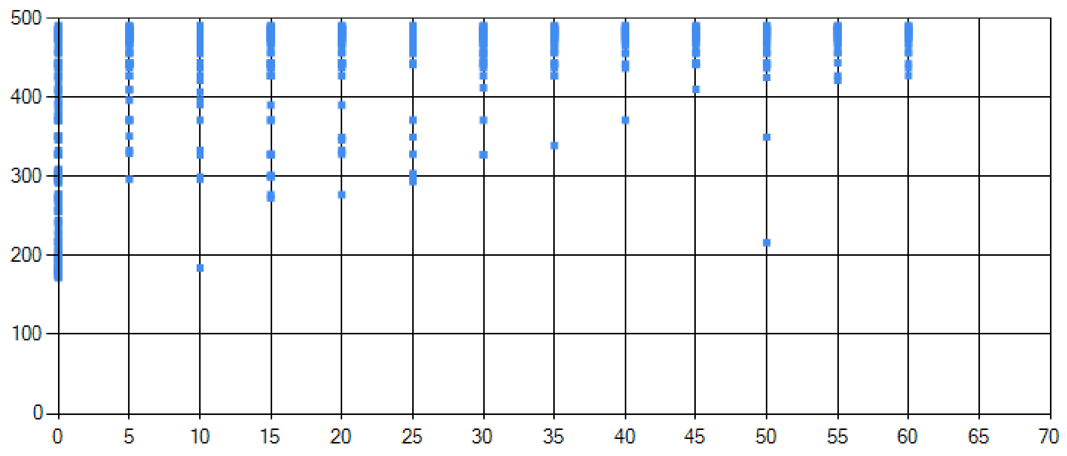

Figure 13. Fitness of each parent for the conservative genetic algorithm

The results generated from the conservative approach has a very aggressive dip in convergence at generation 50. This is due to the conservative approach being more selective in what it mutates. If it mutates something in a very negative way, it will eventually use the chromosomes it did not mutate to overwrite the problem.

To compare our results, we used the standard 0-1 Knapsack approximation algorithm, the greedy algorithm. This traditional algorithm finds the ratio R between weight W and value V, specifically R = V / W, and takes the largest R values until the maximum capacity will be overwhelmed. The greedy algorithm calculated a maximum total value of 485.737. All three genetic algorithm heuristic values produced a higher total value than the traditional greedy algorithm! All three genetic algorithm heuristic values also found a few possible heuristic settings to get a better result than the greedy algorithm. Thus, proving our generated heuristics can be used to generate an equally efficient algorithm that can generates more accurate results.

To test our claim, we generated new random sets of knapsack objects and applied the best heuristic variables generated by the three genetic heuristics as well as the greedy algorithm to see how it worked. These are the results:

Seed	Balanced	Liberal	Conservative	Greedy	Best
1	469.269	469.269	469.269	478.673	482.398
2	487.921	487.921	487.921	494.275	497.26
3	475.156	475.156	475.156	471.45	478.833
4	479.334	479.334	479.334	475.94	491.844
5	493.8	492.868	493.8	491.2	502.341
6	481.49	480.289	481.49	474.299	486.407
7	480.984	480.779	480.984	480.779	480.984

Table 8. Applying heuristics to a new knapsack problem results

From these tests we can see that our heuristic is generated for a specific knapsack instance, and we only generated the best result in one of our seven tests. We also noticed that it is entirely possible for our heuristic to generate an algorithm that works less effective than the greedy algorithm.

Graph Coloring Problem

At first, we were very surprised with our initial results. We expected a sort of “learning curve” for the genetic algorithm where each generation performed a little bit better than the previous, like our results for the 0-1 knapsack problem. However, what we found was that the first generation of randomly generated heuristics had many terrible results causing a large portion of each tree to be searched, but by the second generation these bad heuristics were almost completely wiped out. This shows why his problem was the basis of showing Khot’s UGC because there are some graphs which are inherently harder than others to solve – yet we do not have a pattern to know what makes the graph harder to solve or not solve according to UGC in case of graph coloring.

We approached this problem similarly to the 0-1 knapsack problem such that we studied a balanced, liberal, and conservative approach to the genetic algorithm. Note that unlike most genetic algorithm results, a lower fitness in this case is better as we calculated the fitness based on how many steps the algorithm took to complete, where a step is coloring a node.

The Random Game: Graph coloring variation

Following are the iterations of The Random Game. The graphs appear to follow a bell curve pattern just as we would expect, so we wanted to crank up our graph to be as complex as we could but were limited by the technology available to us. For seed 2500 we tried to make our scenario as complicated as possible, within a reasonable node count as to limit the total possible number of graphs that could be created.

Seed 17 – Five nodes, four edges

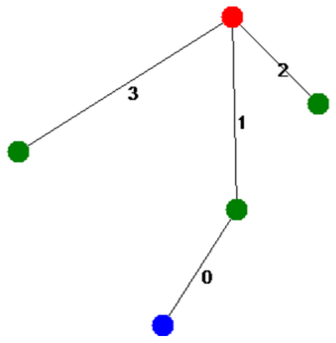

Figure 14. Solution for seed 17

- Edge 0 – Must be either: Blue/Red, Red/Red, or Green/Blue
- Edge 1 – Must be: Red/Green
- Edge 2 – Must be either: Red/Green or Blue/Blue
- Edge 3 – Must not be: Green/Green

Satisfied Percentage

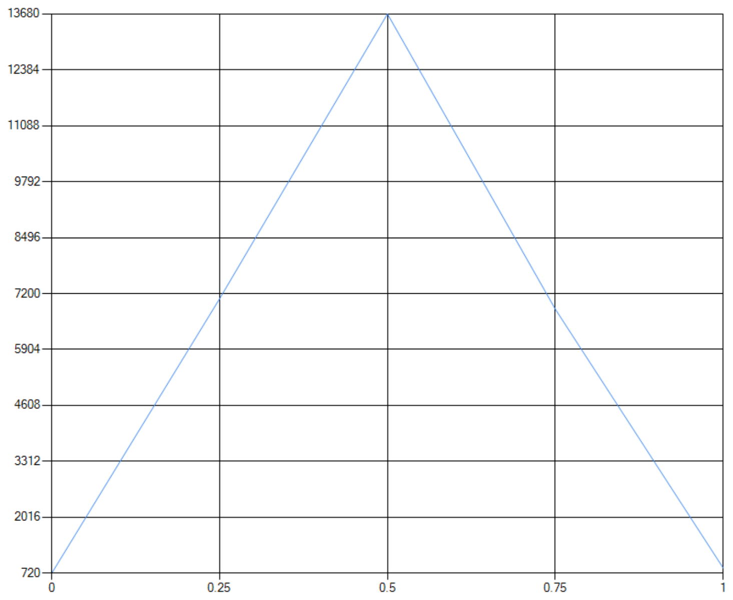

Figure 15. Satisfied percentage for seed 17

Seed 1337 – Five nodes, six edges

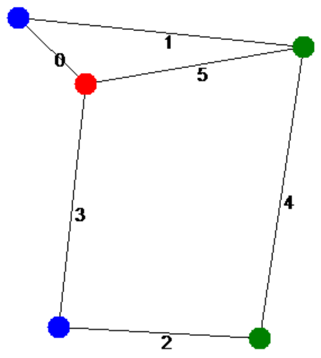

Figure 16. Solution for seed 1337

- Edge 0 – Must be a different color
- Edge 1 – Must not be either: Green/Red, Green/Green
- Edge 2 – Must be either: Blue/Red, Green/Green, Green/Blue
- Edge 3 – Must be either: Blue/Red, Green/Red
- Edge 4 – Must be either Red/Red, Green/Green
- Edge 5 – Must not be either: Red/Red, Blue/Blue

Satisfied Percentage

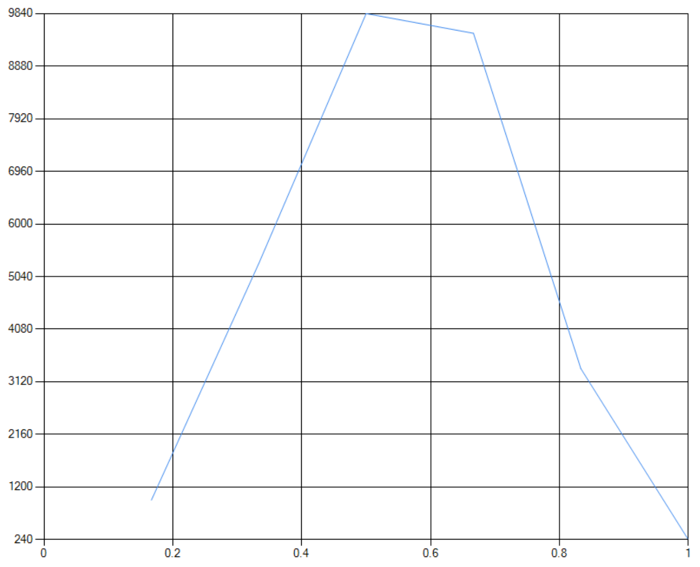

Figure 17. Satisfied percentage for seed 1337

Seed 2000 – Six nodes, seven edges

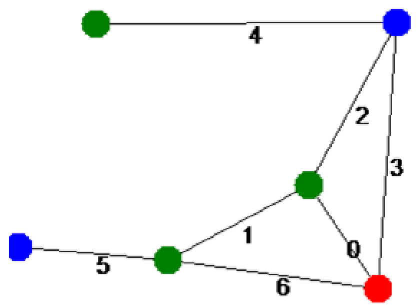

Figure 18. Solution for seed 2000

- Edge 0 – Must not be either: Blue/Blue, Green/Green, Blue/Green
- Edge 1 – Must not be either: Red/Blue, Green/Blue
- Edge 2 – Must not be either: Blue/Blue, Red/Green
- Edge 3 – Must be different color
- Edge 4 – Must be different color
- Edge 5 – Must not be either Red/Blue, Green/Red
- Edge 6 - Must be different color

Satisfied Percentage

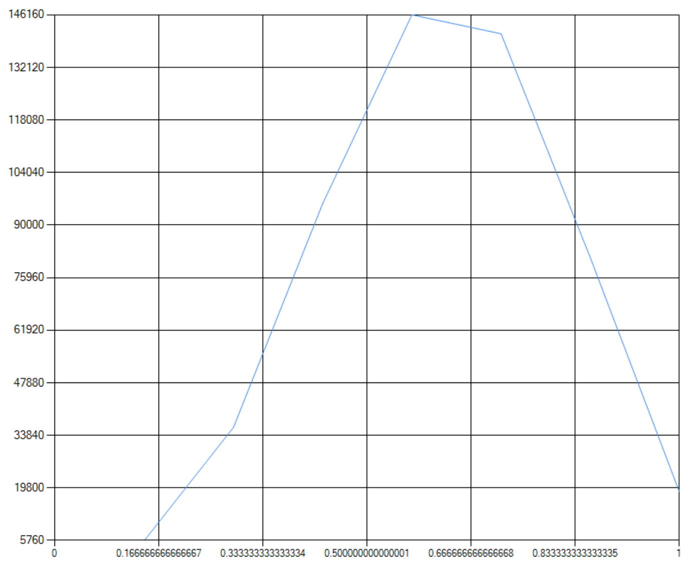

Figure 19. Satisfied percentage for seed 2000

Seed 2500 – Six nodes, fourteen edges

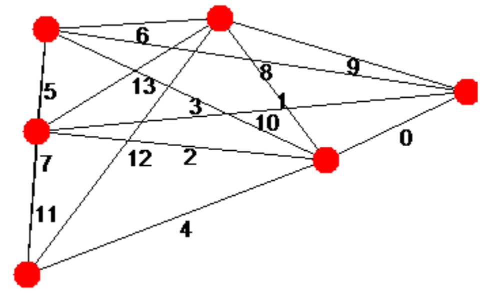

Figure 20. Unsatisfiable problem with seed 2500

- Edge 0 – Must be same color
- Edge 1 – Must be same color
- Edge 2 – Must not be either: Blue/Blue, Blue/Red
- Edge 3 – Must not be either: Blue/Blue, Green/Red
- Edge 4 – Must be same color
- Edge 5 – Must be either: Blue/Red, Blue/Green, Blue/Blue
- Edge 6 - Must be same color
- Edge 7 – Must be either: Green/Blue
- Edge 8 – Must not be either: Blue/Green, Red/Blue
- Edge 9 – Must be either: Green/Blue, Green/Red
- Edge 10 – Must be same color
- Edge 11 – Must be either: Red/Blue, Blue/Blue, Green/Green
- Edge 12 – Must be different color
- Edge 13 – Must be either: Blue/Blue, Green/Red

Satisfied Percentage

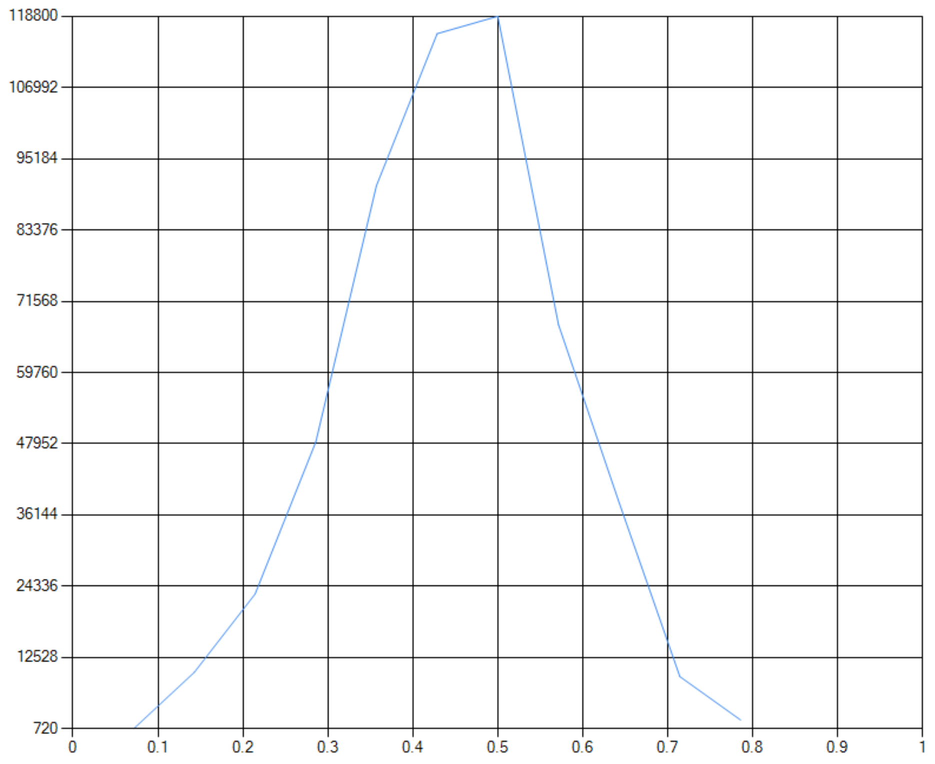

Figure 21. Satisfied percentage for seed 2500

Seed 2500 gives us a perfect example of what Khot suggested with his Unique Games Conjecture, where it can be easy to determine the value of a satisfiable problem, but difficult to find the value of an unsatisfiable problem. To find the value of some of the previous problems, we can just solve until we get the first 100% satisfied state. We know we cannot improve from this point, so we have our solution and halt the algorithm. For seed 2500, we needed to check every possible state to know that there was indeed no solution.

CONCLUSION AND FUTURE RESEACH

In 2002, when the UGC was proposed by Subhas Khot it provided a breakthrough for extending the study of NP-completeness and NP-hardness, possibly suggesting that some problems are harder than others within the spectrum of NP-completeness. The community is divided about the UGC, yet all agree that it is a major research thread moving forward. We used the random game to create an understanding of Khot's UGC by generating the entire solution space to discover that some graphs were unsatisfiable, and unable to be proven unsatisfiable in polynomial time. The satisfiable percentage of the solution space could be one way to understand the Khot's UGC. Particularly one could argue that based on the instance of the problem, as well as randomness of the genetic algorithm, a solution might hit or miss. Our method at best can be used to understand the difficulty facing the scientific community and UGC's as the degree which Khot is talking about applies to all instances and counting (number of iterations) may not tell the full story. What we conclude is that coming up with a measure (in our case percentage of solution space satisfied) is equally difficult yet UGC says that such a degree might exist. Still our thesis provides a very practical way of running the exploration algorithm and terminating after them many runs. One way to increase possibility of genetic algorithm will be to use machine learning algorithm for generating a heuristic which has resulted in good results with data-biases, so it is debatable which of the two (genetic our method vs machine learning by samples) is better eventually. Still the question remains – is there a degree of difficulty among the NP-complete and NP-hard problems which UGC alludes to, and with quantum computers on the horizon with significant promise – our conclusion is that we can arguably say that understanding implications of UGC is work which will be continued.

Like every problem, work is never finished. Dr. Yanyan Zhuang has suggested that we consider more experimentation with the genetic algorithm specifically. She has stated that it would be interesting to try comparing the last generation with the new generation to see if we made a wrong turn and should perhaps revert and try another mutation on the previous generation, rather than the sub-optimal new generation. She has also suggested that we dynamically alter our genetic algorithm variables as the algorithm runs. Perhaps starting with a very liberal approach which will cause very significant mutations, slowly dwindling down to a more conservative approach that will result in minor adjustments will be a good way to consider this.

BIBLIOGRAPHY/REFERENCES

[1] Subhash Khot. 2002. On the power of unique 2-prover 1-round games. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing (STOC '02). Association for Computing Machinery, New York, NY, USA, 767–775. DOI:https://doi.org/10.1145/509907.510017

[2] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan & Rong Qu (2013) Hyper-heuristics: a survey of the state of the art, Journal of the Operational Research Society, 64:12, 1695-1724, DOI: 10.1057/jors.2013.71

[3] D.Jones, S. Mirrazavi, and M. Tamiz, “Multi-objective meta-heuristics: An overview of the current state-of-the-art,” European Journal of Operational Research, vol. 137, no. 1, pp. 1-9, 2002.

[4] S. L. Braunstein, "Quantum computation: a tutorial," [Online]. Available: https://www.saylor.org/site/wp-content/uploads/2011/06/CS411-5.1-1.pdf/.

[5] D. Beckman, A. Chari, S. Devabhaktuni and J. Preskill, "Efficient networks for quantum factoring", *Physical Review A*, vol. 54, no. 2, pp. 1034-1063, 1996.

[6] C. Pomerance, "Smooth numbers and the quadratic sieve", Mathematical Sciences Research Institute, vol. 44, pp. 69-81, 2008.

[7] C. Pomerance, "A Tale of Two Sieves", Notices of the American Mathematical Society, vol. 43, no. 12, pp. 1473-1484, 1996.

[8] Chu, "The beginning of the end for encryption schemes?", MIT News, 2016. [Online]. Available: http://news.mit.edu/2016/quantum-computer-end-encryption-schemes-0303.

[9] M. Hirvensalo, Quantum computing, 2nd ed. Berlin: Springer-Verlag, 2010, pp. 1-47.

[10] E. Horowitz, S. Sahni and S. Rajasekeran, Computer Algorithms, 2nd ed. New Jersey: Silicon Press, 2008.

[11] M. Garey and D. Johnson, Computers and Intractability A Guide to the Theory of NP-Completeness. New York: W.H. Freeman and Co., 1979.

[12] M. Hayward, "Quantum Computing and Shor’s Algorithm", University of Illinois, Urbana-Champaign, 2005.

[13] E. Weisstein, "Hilbert Space", MathWorld--A Wolfram Web Resource. [Online]. Available: http://mathworld.wolfram.com/HilbertSpace.html. [Accessed: 03- May- 2018].

[14] A. Price, J. Rarity and C. Erven, "A quantum key distribution protocol for rapid denial of service detection", in QCrypt 2017, Cambridge, 2017.

[15] P. Pajic, "Quantum Cryptography", University of Vienna, Vienna, 2013.

[16] M. Repka and P. Zajac, "Overview of the Mceliece Cryptosystem and its Security", Tatra Mountains Mathematical Publications, vol. 60, no. 1, 2014.

[17] H. Dinh, C. Moore and A. Russell, "The McEliece Cryptosystem Resists Quantum Fourier Sampling Attacks", 2010.

[18] A. Iqbal, M. Aslam and H. Nayab, "Quantum Cryptography: A brief review of the recent developments and future perspectives", in The International Conference on

Digital Information Processing, Electronics, and Wireless Communications, Dubai, 2016, pp. 43-46.

[19] Li, N. Dattani, X. Chen, X. Liu, H. Wang, R. Tanburn, H. Chen, X. Peng and J. Du, "High-fidelity adiabatic quantum computation using the intrinsic Hamiltonian of a spin system: Application to the experimental factorization of 291311", 2017.

[20] Luca Trevisan, On Khot's Uniqueness Games Conjecture, American Mathematical Society, vol. 49, number 1, January 2012, pp. 91-111, (2011).

[21] Melanie Mitchell, Complexity – A guided Tour, pp.1-349, Oxford University Press, 2009.

[22] Subhash Khot, On the power of unique 2-prover 1-round games, Proceedings of the 34th ACM Symposium on Theory of Computing, 2002, pp. 767–775. MR2121525

[23] Subhash Khot, Inapproximability of NP-complete problems, discrete Fourier analysis, and geometry,Proceedings of the International Congress of Mathematicians, 2010.

[24] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell, Optimal inapproximability results for MAX-CUT and other two-variable CSPs?, Proceedings of the 45th IEEE Symposium on Foundations of Computer Science, 2004, pp. 146–154.

[25] Subhash Khot and Nisheeth Vishnoi, The unique games conjecture, integrality gap for cutproblems and the embeddability of negative type metrics into _1, Proceedings of the 46thIEEE Symposium on Foundations of Computer Science, 2005, pp. 53–63.

[26] David S. Johnson and Michael Gary, Computers and Intractability- Theory of NP Completeness, pp. 1-313, WH Freeman and Company (1979).

[27] Paul O'Hearn, Wizzard Battle, Term Report, CS 6770 VR and HCI by Dr. Sudhanshu Semwal, pp. 1-17, University of Colorado, Colorado Springs (Spring 2018).

[28] Drone Augmented Human Vision: Exocentric Control for Drones Exploring Hidden Areas, ACM Transaction on Visualization and Computer Graphics, Vol 23, number 4, April 2018. pp. 1437-1446 (2018).

[29] Damia Fuentes and Eric Velazquez, Drone VR FPV with Head Tracking Control, Department of Computer Science, CS 6770 VR &HCI Spring 2018 Term report, Instructor: Dr. Sudhanshu Semwal, pp. 1-45 (2018).

[30] Keith Johnson and Sudhanshu K Semwal, Shapes: A Multi-Sensory Environment for the B/VI and Hearing Impaired Community, 2nd International Workshop on Virtual and Augmented Assistive Technology (VAAT) at IEEE Virtual Reality 2014, 29 March - 2 April, Minneapolis, MN, USA, pp.1-6 (2014).

[31] Helen Huang, Analysis of Shor's Algorithm and Quantum Resistant Cryptographic Systems, term report in the Analysis of Algorithms (CS5720) course, Instructor: Dr. SK Semwal, Spring 2018, pp. 1-7 (2018).

[32] Maurice Blanchot, Book titled The Writing of Disaster, Translated by Ann Smock, University of Nebraska Press, pp. 1-150 (1995).

[33] Bessel Van Der Kolk, MD, The Body Keeps the Score: Brain, Mind, and Body in the Healing of Trauma, LLC Glidan Media (Publisher) (2014).

Appendix A: Code

Repository available at: https://github.com/darrare/MS-Thesis

GeneticAlgorithm.cs

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

namespace GeneticAlgorithm
{
    public delegate double FitnessAlgorithm(double[] genes);

    public static class GeneticAlgorithmClass
    {
        public static Action<int, double, int, double, double, double, Tuple<int,
List<double>>> UpdateProgressBar { get; set; }

        /// <summary>
        /// Runs the genetic algorithm using the given parameters to solve for a list of
Chromosomes that satisfy the problem
        /// </summary>
        /// <param name="populationSize">Total size of the population
        /// <para/>(must be in the form of Size = X + Y where x = y(y-1)/2).
        /// <para/>Example: Set Y = 100, therefore X = 100(99)/2 = 4950. X + Y = 5050 where Y
is the number of parents we keep per iteration.)
        /// <para/>Some valid values: 36, 136, 528, 2080</param>
        /// <param name="maxConvergenceDeviationToAccept">0 means every chromosome produces
the exact same result
        /// <para/>.01 means the worst chromosome is within 1% of the best chromosome
        /// <para/>(1 - min fitness score / max fitness score)</param>
        /// <param name="defaultGenes">Default gene set, determined by specific CSP</param>
        /// <param name="chanceToSelectEachChromosome">% chance for each chromosome to be
selected for mutation</param>
        /// <param name="chanceToMutateEachGene">% chance for each gene in the selected
chromosomes to be mutated</param>
        /// <param name="maxGeneMutationDeviation">maximum % amount a genes value can change
after a mutation
        /// <para/>.5 means +- 50%, IE: 90 -> 45-135 and 180 -> 90-270
        /// <para/>1 means the mutation will range from 0-double whatever the genes value
was</param>
        /// <param name="fitnessAlgorithm">The algorithm provided by the CSP to determine the
success of the chromosome</param>
        /// <param name="maxIterationCount">Maximum number of evolutions. Prevents algorithm
from running forever when fitness scores won't converge. Default is infinite</param>
        /// <param name="isHigherFitnessBetter">Determines how to order the results.</param>
        /// <param name="seed">random seed</param>
        /// <returns>List of chromosomes that was the last set before a convergence was found,
ordered from highest to lowest</returns>
        public static List<Chromosome> RunGeneticAlgorithm(
            int populationSize,
            double maxConvergenceDeviationToAccept,
            double[] defaultGenes,
            double chanceToSelectEachChromosome,
            double chanceToMutateEachGene,
            double maxGeneMutationDeviation,
            FitnessAlgorithm fitnessAlgorithm,
            int maxIterationCount,
            int logInterval = 5,
            bool isHigherFitnessBetter = true,
            int seed = 0)
        {
            //Creates a new population, automatically mutates each gene by up to
maxGeneMutationDeviation
            Population pop = new Population(populationSize, defaultGenes,
maxGeneMutationDeviation, isHigherFitnessBetter, seed);

            double convergence = 0;
            double averageFitness = 0;
            double maximumFitness = 0;
            double minimumFitness = 0;
            int i;
            for (i = 0; i < maxIterationCount; i++)
            {
                pop.CalculateFitness(fitnessAlgorithm);
                if ((convergence = pop.CalculateConvergence()) <=
maxConvergenceDeviationToAccept)
                {
                    averageFitness = pop.CalculateAverageFitness();
                    minimumFitness = pop.CalculateMinimumFitness();
                    maximumFitness = pop.CalculateMaximumFitness();
                    UpdateProgressBar?.Invoke((int)(((double)i / (double)maxIterationCount) *
100), convergence, i, averageFitness, minimumFitness, maximumFitness, new Tuple<int,
List<double>>(i, new List<double>()));
                    break;
                }

                if (i % logInterval == 0)
                {
                    averageFitness = pop.CalculateAverageFitness();
                    minimumFitness = pop.CalculateMinimumFitness();
                    maximumFitness = pop.CalculateMaximumFitness();
                    UpdateProgressBar?.Invoke((int)(((double)i / (double)maxIterationCount) *
100), convergence, i, averageFitness, minimumFitness, maximumFitness, new Tuple<int,
List<double>>(i, pop.GetFitnesses()));
                }

                pop.RemoveUnworthy();
                pop.MatePopulation();
                pop.Mutate(chanceToSelectEachChromosome, chanceToMutateEachGene,
maxGeneMutationDeviation);
            }

            if (isHigherFitnessBetter)
                return pop.Chromosomes.OrderByDescending(t => t.FitnessScore).ToList();
            else
                return pop.Chromosomes.OrderBy(t => t.FitnessScore).ToList();
        }
    }
}
```

Population.cs

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

namespace GeneticAlgorithm
{
    class Population
    {
        static Random Rand = new Random();

        public List<Chromosome> Chromosomes = new List<Chromosome>();

        public int GeneCount { get; private set; } = 0;
        public int PopulationSize { get; private set; } = 0;

        public bool IsHigherFitnessBetter { get; private set; } = true;

        public int NumParentsToKeepEachIteration { get; private set; } = 0;

        /// <summary>
        /// public constructor used to generate the first generation
        /// </summary>
        /// <param name="populationSize">Total size of the population
        /// <para />(must be in the form of Size = X + Y where x = y(y-1)/2).
        /// <para />Example: Set Y = 100, therefore X = 100(99)/2 = 4950. X + Y = 5050 where Y
is the number of parents we keep per iteration.)
        /// <para />Some valid values: 36, 136, 528, 2080</param>
        /// <param name="defaultGenes">The set of default genes for the first
generation</param>
        /// <param name="maxDerivation">Percentage wise, how much can we deviate from the
default genes values</param>
        /// <param name="isHigherFitnessBetter">(OPTIONAL) Is a higher fitness score better
than a lower fitness score?</param>
        /// <param name="seed">(OPTIONAL) Seed for randomization to get same results</param>
        public Population(int populationSize, double[] defaultGenes, double maxDerivation,
bool isHigherFitnessBetter = true, int seed = 0)
        {
            if (seed != 0)
                Rand = new Random(seed);

            GeneCount = defaultGenes.Length;
            PopulationSize = populationSize;
            IsHigherFitnessBetter = isHigherFitnessBetter;
            NumParentsToKeepEachIteration =
CalculateNumParentsToKeepEachIteration(PopulationSize);
            GenerateFirstGeneration(defaultGenes, maxDerivation);
        }

        /// <summary>
        /// Generates the first generation of the genetic algorithm
        /// </summary>
        /// <param name="defaultGenes">The set of default genes for the first
generation</param>
        /// <param name="maxDerivation">Percentage wise, how much can we deviate from the
default genes values</param>
        private void GenerateFirstGeneration(double[] defaultGenes, double maxDerivation)
        {
            for (int i = 0; i < PopulationSize; i++)
            {
                double[] newGenes = new double[GeneCount];
                for (int j = 0; j < GeneCount; j++)
                {
                    newGenes[j] = defaultGenes[j] + defaultGenes[j] * ((Rand.NextDouble() * 2)
- 1) * maxDerivation;
                }
                Chromosomes.Add(new Chromosome(newGenes));
            }
        }

        /// <summary>
        /// Compute the number of parents to keep each round
        /// </summary>
        /// <param name="total">The total size of our population</param>
        /// <returns>The number of parents to keep each round</returns>
        private int CalculateNumParentsToKeepEachIteration(int total)
        {
            for (int Y = 1; Y < total; Y++)
            {
                int X = total - Y;
                int result = (Y * (Y - 1)) / 2;
                if (X == result)
                    return Y;
                else if (X < result)
                    throw new Exception("Invalid population size, must be in the form of Size
= X + Y where x = y(y-1)/2). Example: Set Y = 100, therefore X = 100(99)/2 = 4950. X + Y =
5050 where Y is the number of parents we keep per iteration.");
            }
            throw new Exception("Total must be greater than 2");
        }

        /// <summary>
        /// Runs through each chromosome to calculate its fitness against the provided fitness
algorithm
        /// </summary>
        /// <param name="fitnessAlgorithm">The fitness algorithm</param>
        public void CalculateFitness(FitnessAlgorithm fitnessAlgorithm)
        {
            foreach (Chromosome c in Chromosomes)
            {
                c.FitnessScore = fitnessAlgorithm(c.Genes);
            }
        }

        /// <summary>
        /// Calulcates the average fitness of this population
        /// </summary>
        /// <returns>The average fitness</returns>
        public double CalculateAverageFitness()
        {
            return Chromosomes.Average(t => t.FitnessScore);
        }

        public double CalculateMaximumFitness()
        {
            return Chromosomes.Max(t => t.FitnessScore);
        }

        public double CalculateMinimumFitness()
        {
            return Chromosomes.Min(t => t.FitnessScore);
        }

        /// <summary>
        /// Gets a list of all fitnesses
        /// </summary>
        /// <returns>All fitnesses combined with the step to generate the graph</returns>
        public List<double> GetFitnesses()
        {
            double min = Chromosomes.Min(t => t.FitnessScore);
            double max = Chromosomes.Max(t => t.FitnessScore);
            if (min == max)
                return new List<double>() { max };

            return Chromosomes.Select(t => t.FitnessScore).Distinct().ToList();
        }

        /// <summary>
        /// Often considered the "Selection" part of the algorithm.
        /// Picks the strongest to survive.
        /// </summary>
        public void RemoveUnworthy()
        {
            if (Chromosomes.Sum(t => t.FitnessScore) == 0)
                throw new Exception("Must calculate fitness before selection");

            if (IsHigherFitnessBetter)
                Chromosomes = Chromosomes.OrderByDescending(t =>
t.FitnessScore).Take(NumParentsToKeepEachIteration).ToList();
            else
                Chromosomes = Chromosomes.OrderBy(t =>
t.FitnessScore).Take(NumParentsToKeepEachIteration).ToList();
        }

        /// <summary>
        /// Often called the "Crossover" part of the algorithm
        /// Picks shares values between each pair of parents to create offsprings
        /// The offspring will be added to the collection of parents to form the next
generation
        /// </summary>
        public void MatePopulation()
        {
            //Select a random crossover point that will always include at least 1 gene from
each parent
            int crossoverPoint = Rand.Next(1, GeneCount - 1);

            //New list so we can iterate over just the parents
            List<Chromosome> chromosomesToAdd = new List<Chromosome>();

            //Force every parent to breed with every other parent
            //Take the first part of parent 1 up to crossoverPoint
            //Take the last part of parent 2 from crossoverPoint
            for (int i = 0; i < Chromosomes.Count - 1; i++)
            {
                for (int j = i + 1; j < Chromosomes.Count; j++)
                {
                    double[] newGenes = new double[GeneCount];
                    for (int x = 0; x < crossoverPoint; x++)
                        newGenes[x] = Chromosomes[i].Genes[x];
                    for (int y = crossoverPoint; y < GeneCount; y++)
                        newGenes[y] = Chromosomes[j].Genes[y];
                    chromosomesToAdd.Add(new Chromosome(newGenes));
                }
            }
            //Add the new children to the list with the parents
            Chromosomes.AddRange(chromosomesToAdd);
        }
        /// <summary>
        /// Often called the mutation part of the algorithm
        /// Slightly alters some of the chromosomes randomly.
        /// </summary>
        /// <param name="initialSelectionChance">0-1 chance of mutating each
chromosome</param>
        /// <param name="individualGeneSelectionChance">0-1 chance of selecting each
individual gene from the selected chromosomes</param>
        /// <param name="mutationPercentageMax">0-1 upper bound percentage of how much we can
mutate each gene. 0 = no mutation, 1 = can either set to 0 or double</param>
        public void Mutate(double initialSelectionChance, double
individualGeneSelectionChance, double mutationPercentageMax)
        {
            if (initialSelectionChance < 0 || initialSelectionChance > 1)
                throw new Exception("Initial selection chance must be between 0 and 1");
            if (individualGeneSelectionChance < 0 || individualGeneSelectionChance > 1)
                throw new Exception("Individual gene selection chance must be between 0 and
1");
            if (mutationPercentageMax < 0 || mutationPercentageMax > 1)
                throw new Exception("Mutation percentage max must be between 0 and 1");

            //For every chromosome
            foreach (Chromosome c in Chromosomes)
            {
                //Roll the dice to see if we should mutate the chromosome
                if (Rand.NextDouble() <= initialSelectionChance)
                {
                    //For every gene in chromosome
                    for (int i = 0; i < GeneCount; i++)
                    {
                        //Roll the dice to see if we should mutate the gene
                        if (Rand.NextDouble() <= individualGeneSelectionChance)
                        {
                            c.Genes[i] = c.Genes[i] + c.Genes[i] * ((Rand.NextDouble() * 2) -
1) * mutationPercentageMax;
                        }
                    }
                }
                c.Normalize(); //This is what balances the genetic algorithm
            }
        }

        /// <summary>
        /// Calculates the difference between min and max values based on percentage.
        /// </summary>
        /// <returns>Percentage of max that the min is</returns>
        public double CalculateConvergence()
        {
            return 1 - (Chromosomes.Min(t => t.FitnessScore) / Chromosomes.Max(t =>
t.FitnessScore));
        }
    }
}
```

Chromosome.cs

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

namespace GeneticAlgorithm
{
    public class Chromosome
    {
        public double[] Genes { get; set; }
        public double FitnessScore { get; set; } = 0;

        public Chromosome(double[] genes)
        {
            Genes = genes;
        }

        /// <summary>
        /// Normalizes the gene array to ensure a balanced genetic algorithm
        /// </summary>
        public void Normalize()
        {
            double magnitude = Math.Sqrt(Genes.Sum(t => t * t));

            for (int i = 0; i < Genes.Length; i++)
            {
                Genes[i] /= magnitude;
            }
        }
    }
}
```

Knapsack.cs

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

namespace _0_1Knapsack
{
    class Knapsack
    {
        public int Capacity { get; private set; }
        public KnapsackItem[] Items { get; private set; }
        public double MaximumWeight { get; private set; }
        public double MinimumWeight { get; private set; }
        public double MaximumValue { get; private set; }
        public double MinimumValue { get; private set; }

        Random rand = new Random();


        /// <summary>
        /// Constructor for a knapsack
        /// </summary>
        /// <param name="capacity">Max weight holdable by knapsack</param>
        /// <param name="minWeight">minimum weight for each object</param>
        /// <param name="maxWeight">maximum weight for each object</param>
        /// <param name="minValue">minimum value for each object</param>
        /// <param name="maxValue">maximum value for each object</param>
        /// <param name="count">number of objects we need to find optimal solution for</param>
        /// <param name="seed">(OPTIONAL) randomized seed</param>
        public Knapsack(int capacity, double minWeight, double maxWeight, double minValue,
double maxValue, int count, int seed = 0)
        {
            Capacity = capacity;
            MaximumWeight = maxWeight;
            MinimumWeight = minWeight;
            MaximumValue = maxValue;
            MinimumValue = minValue;

            if (seed != 0)
                rand = new Random(seed);

            //Set up items
            Items = new KnapsackItem[count];
            for (int i = 0; i < count; i++)
            {
                Items[i] = new KnapsackItem()
                {
                    Weight = (rand.NextDouble() * (maxWeight - minWeight) + minWeight),
                    Value = (rand.NextDouble() * (maxValue - minValue) + minValue)
                };
            }
        }

        /// <summary>
        /// The fitness is the total value of the load. Weight cannot go over capacity
        /// </summary>
        /// <param name="genes">
        /// [0] = priority of low weight (double)
        /// [1] = priority of high weight (double)
        /// [2] = priority of low value (double)
        /// [3] = priority of high value (double)
        /// </param>
        /// <returns>The value of the load</returns>
        public double Fitness(double[] genes)
        {
            //Use heuristic to sort array such that most wanted items are at the beginning
            List<KnapsackItem> inBag = new List<KnapsackItem>();
            foreach (KnapsackItem item in Items.OrderByDescending(item =>
                (MaximumWeight - item.Weight) * Convert.ToDouble(genes[0]) +
                item.Weight * Convert.ToDouble(genes[1]) +
                (MaximumValue - item.Value) * Convert.ToDouble(genes[2]) +
                item.Value * Convert.ToDouble(genes[3])))
            {
                if (inBag.Sum(t => t.Weight) + item.Weight < Capacity)
                {
                    inBag.Add(item);
                }
            }
            return inBag.Sum(t => t.Value);
        }

        /// <summary>
        /// Generates a list of all solutions based on the converged generation
        /// </summary>
        /// <param name="genes">The converged genes</param>
        public List<KnapsackSolution> GetKnapsackSolutionsFromGenes(List<double[]> genes)
        {
            List<KnapsackSolution> solutions = new List<KnapsackSolution>();
            foreach (double[] geneSet in genes)
            {
                List<KnapsackItem> inBag = new List<KnapsackItem>();
                foreach (KnapsackItem item in Items.OrderByDescending(item =>
                    (MaximumWeight - item.Weight) * Convert.ToDouble(geneSet[0]) +
                    item.Weight * Convert.ToDouble(geneSet[1]) +
                    (MaximumValue - item.Value) * Convert.ToDouble(geneSet[2]) +
                    item.Value * Convert.ToDouble(geneSet[3])
                    ))
                {
                    if (inBag.Sum(t => t.Weight) + item.Weight < Capacity)
                    {
                        inBag.Add(item);
                    }
                }

                KnapsackSolution solution = new KnapsackSolution() { Items = inBag, Genes =
geneSet };
                if (!solutions.Contains(solution))
                {
                    solutions.Add(solution);
                }
            }
            return solutions;
        }
    }

    /// <summary>
    /// Used for storage
    /// </summary>
    public class KnapsackItem
    {
        public double Weight { get; set; }
        public double Value { get; set; }

        /// <summary>
        /// Custom hash. Might run into overflow?!?!?
        /// </summary>
        /// <returns>The calculated hash</returns>
        public override int GetHashCode()
        {
            return Weight.GetHashCode() + Value.GetHashCode();
        }
    }

    public class KnapsackSolution : IEquatable<KnapsackSolution>
    {
        public List<KnapsackItem> Items { get; set; }
        public double TotalWeight { get { return Items.Sum(t => t.Weight); } }
        public double TotalValue { get { return Items.Sum(t => t.Value); } }
        public double[] Genes { get; set; }

        public bool Equals(KnapsackSolution other)
        {
            if (other == null)
                return false;
            Items = Items.OrderBy(t => t.Weight).ThenBy(t => t.Value).ToList();
            other.Items = other.Items.OrderBy(t => t.Weight).ThenBy(t => t.Value).ToList();

            for (int i = 0; i < Items.Count; i++)
            {
                if (Items[i].Weight != other.Items[i].Weight
                    || Items[i].Value != other.Items[i].Value)
                    return false;
            }
            return true;
        }

        public int GetHashCode(KnapsackSolution obj)
        {
            if (obj == null)
                return 0;
            int hash = 23;
            foreach (KnapsackItem item in obj.Items)
            {
                hash = hash * 31 + item.GetHashCode();
            }
            return hash;
        }
    }
}
```

FormZeroOneKnapsack.cs

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

namespace _0_1Knapsack
{
    public partial class FormZeroOneKnapsack : Form
    {
        Random rand = new Random();

        Dictionary<Control, string> toolTips;

        public FormZeroOneKnapsack()
        {
            InitializeComponent();

            //Set tooltip strings for all input fields
            toolTips = new Dictionary<Control, string>()
            {
                { TxtBx_Capacity, "The maximum space the knapsack can hold." },
                { TxtBx_MinimumWeight, "The minimum possible randomly generated weight for a 
knapsack object." },
                { TxtBx_MaximumWeight, "The maximum possible randomly generated weight for a 
knapsack object." },
                { TxtBx_MinimumValue, "The minimum possible randomly generated value for a 
knapsack object." },
                { TxtBx_MaximumValue, "The maximum possible randomly generated value for a 
knapsack object." },
                { TxtBx_PopulationSize , "Must be in the form of Size = X + Y where x = y(y-
1)/2) where y is the number of chromosomes kept each iteration to breed." + 
Environment.NewLine + "Some valid values: 36, 136, 528, 2080" },
                { TxtBx_Convergence, "The percentage of convergence at which we stop the 
algorithm because all chromosomes are within this range of each other." },
                { TxtBx_DefaultGeneValue, "All genes will start at this value, and will be 
mutated once prior to calculating fitness." },
                { TxtBx_MaxIterations, "Maximum amount of times the algorithm will reproduce." 
},
                { TxtBx_PercentChromosomesMutated, "% chance a chromosome will be selected to 
mutate." + Environment.NewLine + "Some chromosomes are left the same if not 100%" },
                { TxtBx_PercentGenesMutated, "After a chromosome has been selected to mutate, 
this is the percentage chance for each gene to actually be mutated." },
                { TxtBx_PercentMutationDeviation, "How much off of the value can we mutate." + 
Environment.NewLine + ".5 means +- 50%, IE: 90 -> 45-135 and 180 -> 90-270" },
                { TxtBx_Seed, "Random seed used to generate same results every time you run 
the algorithm." },
                { TxtBx_NumberKnapsackObjects, "Number of knapsack objects to generate 
randomly with the below weights and values randomized." }
            };

            //Add mouseover and leave events for each control that has a tooltip associated 
with it
            foreach (KeyValuePair<Control, string> tip in toolTips)
            {
                ToolTip tt = new ToolTip();
                tip.Key.MouseEnter += (s, o) =>
                {
                    tt = new ToolTip();
                    tt.InitialDelay = 300;
                    tt.Show(tip.Value, tip.Key, 0);
                };

                tip.Key.MouseLeave += (s, o) =>
                {
                    tt.Dispose();
                };

            }

            //Load the previous runs values into the fields
            if (File.Exists(AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/ZeroOneKnapsack/MostRecentRun.rd"))
            {
                try
                {
                    using (StreamReader sr = new
StreamReader(AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/ZeroOneKnapsack/MostRecentRun.rd"))
                    {
                        TxtBx_Capacity.Text = sr.ReadLine();
                        TxtBx_MinimumWeight.Text = sr.ReadLine();
                        TxtBx_MaximumWeight.Text = sr.ReadLine();
                        TxtBx_MinimumValue.Text = sr.ReadLine();
                        TxtBx_MaximumValue.Text = sr.ReadLine();
                        TxtBx_NumberKnapsackObjects.Text = sr.ReadLine();
                        TxtBx_Seed.Text = sr.ReadLine();
                        TxtBx_PopulationSize.Text = sr.ReadLine();
                        TxtBx_Convergence.Text = sr.ReadLine();
                        TxtBx_DefaultGeneValue.Text = sr.ReadLine();
                        TxtBx_MaxIterations.Text = sr.ReadLine();
                        TxtBx_PercentChromosomesMutated.Text = sr.ReadLine();
                        TxtBx_PercentGenesMutated.Text = sr.ReadLine();
                        TxtBx_PercentMutationDeviation.Text = sr.ReadLine();
                    }
                }
                catch { /*Just leave values at 0 if the above crashes (someone messed with the
file or debugging issues, should fix after next run) */ }
            }
        }

        /// <summary>
        /// The randomize parameters button has been clicked.
        /// Randomize the values of the parameter fields
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Btn_RandomizeParameters_Click(object sender, EventArgs e)
        {
            TxtBx_Capacity.Text = rand.Next(10, 100).ToString();

            double weightA = (rand.NextDouble() * 5), weightB = (rand.NextDouble() * 5);
            TxtBx_MinimumWeight.Text = (weightA > weightB ? weightB :
weightA).ToString("#.###");
            TxtBx_MaximumWeight.Text = (weightA < weightB ? weightB :
weightA).ToString("#.###");

            double valueA = (rand.NextDouble() * 5), valueB = (rand.NextDouble() * 5);
            TxtBx_MinimumValue.Text = (valueA > valueB ? valueB : valueA).ToString("#.###");
            TxtBx_MaximumValue.Text = (valueA < valueB ? valueB : valueA).ToString("#.###");
            TxtBx_NumberKnapsackObjects.Text = rand.Next(25, 200).ToString();

            int y = rand.Next(10, 50);
            TxtBx_PopulationSize.Text = (y + (y * (y - 1) / 2)).ToString();
            TxtBx_Convergence.Text = rand.NextDouble().ToString("#.###");
            TxtBx_DefaultGeneValue.Text = (rand.NextDouble() * 10).ToString("#.###");
            TxtBx_MaxIterations.Text = 1000.ToString();
            TxtBx_PercentChromosomesMutated.Text = rand.NextDouble().ToString("#.###");
            TxtBx_PercentGenesMutated.Text = rand.NextDouble().ToString("#.###");
            TxtBx_PercentMutationDeviation.Text = rand.NextDouble().ToString("#.###");
        }

        /// <summary>
        /// The run algorithm button has been clicked
        /// Store the value of the parameters in a file and then proceed to run the genetic
algorithm
        /// </summary>
        private async void Btn_RunAlgorithm_Click(object sender, EventArgs e)
        {
            if (IsParameterError())
                return;

            //Save our current parameters to be spawned next time we run the program
            string destination = AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/ZeroOneKnapsack";
            if (!Directory.Exists(destination))
                Directory.CreateDirectory(destination);

            using (StreamWriter sw = new StreamWriter(new FileStream(destination +
"/MostRecentRun.rd", FileMode.Create)))
            {
                sw.WriteLine(TxtBx_Capacity.Text);
                sw.WriteLine(TxtBx_MinimumWeight.Text);
                sw.WriteLine(TxtBx_MaximumWeight.Text);
                sw.WriteLine(TxtBx_MinimumValue.Text);
                sw.WriteLine(TxtBx_MaximumValue.Text);
                sw.WriteLine(TxtBx_NumberKnapsackObjects.Text);
                sw.WriteLine(TxtBx_Seed.Text);
                sw.WriteLine(TxtBx_PopulationSize.Text);
                sw.WriteLine(TxtBx_Convergence.Text);
                sw.WriteLine(TxtBx_DefaultGeneValue.Text);
                sw.WriteLine(TxtBx_MaxIterations.Text);
                sw.WriteLine(TxtBx_PercentChromosomesMutated.Text);
                sw.WriteLine(TxtBx_PercentGenesMutated.Text);
                sw.WriteLine(TxtBx_PercentMutationDeviation.Text);
            }

            Knapsack knapsack = new Knapsack(
                int.Parse(TxtBx_Capacity.Text),
                double.Parse(TxtBx_MinimumWeight.Text),
                double.Parse(TxtBx_MaximumWeight.Text),
                double.Parse(TxtBx_MinimumValue.Text),
                double.Parse(TxtBx_MaximumValue.Text),
                int.Parse(TxtBx_NumberKnapsackObjects.Text),
                string.IsNullOrEmpty(TxtBx_Seed.Text) ? 0 : int.Parse(TxtBx_Seed.Text));

            GenerateKnapsackObjectsDataGridView(knapsack.Items);

            Btn_RandomizeParameters.Enabled = false;
            Btn_RunAlgorithm.Enabled = false;

            List<Tuple<int, double, double, double, double>> dataPoints = new List<Tuple<int,
double, double, double, double>>();
```

```
            List<Tuple<int, List<double>>> selectedChromosomesFitness = new List<Tuple<int,
List<double>>>();
            GeneticAlgorithm.GeneticAlgorithmClass.UpdateProgressBar = (percent, convergence,
iteration, averageFitness, minimumFitness, maximumFitness, selectedFitness) =>
            {
                PB_Knapsack.Invoke((MethodInvoker)delegate ()
                {
                    PB_Knapsack.Value = percent;
                    LBL_Convergence.Text = convergence.ToString();
                    LBL_Iteration.Text = iteration.ToString();
                    dataPoints.Add(new Tuple<int, double, double, double, double>(iteration,
convergence, averageFitness, minimumFitness, maximumFitness));
                    selectedChromosomesFitness.Add(selectedFitness);
                });
            };
            int logInterval = 5;
            List<GeneticAlgorithm.Chromosome> chromosomes = await Task.Factory.StartNew(() =>
            {
                return GeneticAlgorithm.GeneticAlgorithmClass.RunGeneticAlgorithm(
                    int.Parse(TxtBx_PopulationSize.Text),
                    double.Parse(TxtBx_Convergence.Text),
                    new double[] { double.Parse(TxtBx_DefaultGeneValue.Text),
double.Parse(TxtBx_DefaultGeneValue.Text), double.Parse(TxtBx_DefaultGeneValue.Text),
double.Parse(TxtBx_DefaultGeneValue.Text) },
                    double.Parse(TxtBx_PercentChromosomesMutated.Text),
                    double.Parse(TxtBx_PercentGenesMutated.Text),
                    double.Parse(TxtBx_PercentMutationDeviation.Text),
                    knapsack.Fitness,
                    int.Parse(TxtBx_MaxIterations.Text),
                    logInterval,
                    true,
                    string.IsNullOrEmpty(TxtBx_Seed.Text) ? 0 : int.Parse(TxtBx_Seed.Text));
            });

            Btn_RandomizeParameters.Enabled = true;
            Btn_RunAlgorithm.Enabled = true;

            //handle chromosomes in results data grid view

GenerateResultsDataGridView(knapsack.GetKnapsackSolutionsFromGenes(chromosomes.Select(t =>
t.Genes).ToList()), knapsack.Items.ToList());
            Common.FormResults form = new Common.FormResults();
            if (form.InitializeChart(dataPoints.Select(t => t.Item1).ToList(),
dataPoints.Select(t => t.Item2).ToList(), dataPoints.Select(t => t.Item3).ToList(),
dataPoints.Select(t => t.Item4).ToList(), dataPoints.Select(t => t.Item5).ToList(),
selectedChromosomesFitness, logInterval))
            {
                form.Show();
            }
            else
            {
                MessageBox.Show("Not enough iterations to show results.");
            }
        }

        /// <summary>
        /// Checks to make sure all values are doubles
        /// </summary>
        /// <returns>True: an error was returned</returns>
        private bool IsParameterError()
        {
            string errors = "";
            double resultD;
            int resultI;
            //knapsack parameters
            if (!int.TryParse(TxtBx_Capacity.Text, out resultI))
                errors += "Invalid value for Capacity" + Environment.NewLine;
            if (!double.TryParse(TxtBx_MinimumWeight.Text, out resultD))
                errors += "Invalid value for Min Weight" + Environment.NewLine;
            if (!double.TryParse(TxtBx_MaximumWeight.Text, out resultD))
                errors += "Invalid value for Max Weight" + Environment.NewLine;
            if (!double.TryParse(TxtBx_MinimumValue.Text, out resultD))
                errors += "Invalid value for Min Value" + Environment.NewLine;
            if (!double.TryParse(TxtBx_MaximumValue.Text, out resultD))
                errors += "Invalid value for Max Value" + Environment.NewLine;
            if (!int.TryParse(TxtBx_NumberKnapsackObjects.Text, out resultI))
                errors += "Invalid value for # Knapsack Objects" + Environment.NewLine;


            //genetic algorithm parameters
            if (!int.TryParse(TxtBx_PopulationSize.Text, out resultI) &&
ValidatePopulationSize(int.Parse(TxtBx_PopulationSize.Text)))
                errors += "Invalid value for Population Size" + Environment.NewLine;
            if (!double.TryParse(TxtBx_Convergence.Text, out resultD))
                errors += "Invalid value for Convergence" + Environment.NewLine;
            if (!double.TryParse(TxtBx_DefaultGeneValue.Text, out resultD))
                errors += "Invalid value for Default Gene Value" + Environment.NewLine;
            if (!int.TryParse(TxtBx_MaxIterations.Text, out resultI) &&
!string.IsNullOrEmpty(TxtBx_MaxIterations.Text))
                errors += "Invalid value for Max Iterations" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentChromosomesMutated.Text, out resultD))
                errors += "Invalid value for % Chromosomes Mutated" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentGenesMutated.Text, out resultD))
                errors += "Invalid value for % Genes Mutated" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentMutationDeviation.Text, out resultD))
                errors += "Invalid value for % Deviation Mutation" + Environment.NewLine;
            if (!int.TryParse(TxtBx_Seed.Text, out resultI) &&
!string.IsNullOrEmpty(TxtBx_Seed.Text))
                errors += "Invalid value for Seed" + Environment.NewLine;

            if (!string.IsNullOrEmpty(errors))
            {
                MessageBox.Show(errors);
                return true;
            }
            return false;
        }

        /// <summary>
        /// Verifies that the population size follows the formula size = y + y(y-1)/2
        /// Uses the quadratic forumla to verify with a = -.5, b = -.5 and c = size
        /// </summary>
        /// <param name="size"></param>
        /// <returns></returns>
        private bool ValidatePopulationSize(int size)
        {
            int y;
            if (!int.TryParse(((.5 - Math.Sqrt(.25 - (4 * -.5 * size))) / -1).ToString(), out
y))
            {
                return false;
            }
            return (size == y + y * (y - 1) / 2);
        }

        private void GenerateKnapsackObjectsDataGridView(KnapsackItem[] items)
        {
            DataTable table = new DataTable("Knapsack Objects");
            table.Columns.Add("id");
            table.Columns.Add("weight");
            table.Columns.Add("value");
            table.Columns.Add("value/weight");

            for (int i = 0; i < items.Length; i++)
            {
                table.Rows.Add(new object[] { i, items[i].Weight.ToString("#.###"),
items[i].Value.ToString("#.###"), (items[i].Value / items[i].Weight).ToString("#.###") });
            }

            DGV_KnapsackObjects.DataSource = table;
            for (int i = 0; i < DGV_KnapsackObjects.Columns.Count; i++)
            {
                DGV_KnapsackObjects.Columns[i].AutoSizeMode =
DataGridViewAutoSizeColumnMode.ColumnHeader;
            }
        }

        /// <summary>
        /// Generates the results data grid view
        /// </summary>
        /// <param name="solutions">The solutions generated by the algorithm</param>
        /// <param name="items">Used to get the index of these items to connect to the other
data grid view</param>
        private void GenerateResultsDataGridView(List<KnapsackSolution> solutions,
List<KnapsackItem> items)
        {
            DataTable table = new DataTable("Final Evolution");
            table.Columns.Add("id");
            table.Columns.Add("Min Weight Gene");
            table.Columns.Add("Max Weight Gene");
            table.Columns.Add("Min Value Gene");
            table.Columns.Add("Max Value Gene");
            table.Columns.Add("Total Weight");
            table.Columns.Add("Total Value");
            table.Columns.Add("Total Value / Total Weight");
            table.Columns.Add("Contained Knapsack Objects");

            for (int i = 0; i < solutions.Count; i++)
            {
                double[] normalizedGenes = solutions[i].Genes.Select(t => (double)t /
(double)solutions[i].Genes.Max()).ToArray();
                table.Rows.Add(new object[] { i,
                    normalizedGenes[0].ToString("#.###"),
                    normalizedGenes[1].ToString("#.###"),
                    normalizedGenes[2].ToString("#.###"),
                    normalizedGenes[3].ToString("#.###"),
                    solutions[i].TotalWeight.ToString("#.###"),
                    solutions[i].TotalValue.ToString("#.###"),
                    (solutions[i].TotalValue / solutions[i].TotalWeight).ToString("#.###"),
                    solutions[i].Items.Select(t => items.IndexOf(t)).OrderBy(t => t).Select(t
=> t.ToString()).Aggregate((accum, cur) => accum += ", " + cur)
                });
            }

            DGV_Results.DataSource = table;
            for (int i = 0; i < DGV_Results.Columns.Count; i++)
            {
                DGV_Results.Columns[i].AutoSizeMode =
DataGridViewAutoSizeColumnMode.ColumnHeader;
            }
        }
        private void Btn_CompareHeuristicsAgainstGreedyAlgorithm_Click(object sender,
EventArgs e)
        {
            if (DGV_Results.DataSource == null || DGV_KnapsackObjects.DataSource == null)
            {
                MessageBox.Show("Must have results and knapsack objects generated prior to
compare.", "Error", MessageBoxButtons.OK);
                return;
            }

            FormCompareAgainstGreedy form = new FormCompareAgainstGreedy();
            form.Show();
            form.InstantiateWithExistingData((DataTable)DGV_Results.DataSource,
(DataTable)DGV_KnapsackObjects.DataSource, int.Parse(TxtBx_Capacity.Text));
        }
    }
}
```

Graph.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;
using System.Diagnostics;

namespace MapColoring
{
    public class Graph
    {
        /// <summary>
        /// Due to the four color theorem, purple and on should NEVER be used in an optimal
solution
        /// </summary>
        public static List<Color> colorOrder = new List<Color>() { Color.Red, Color.Blue,
Color.Green, Color.Yellow };

        public List<Node> Nodes { get; set; } = new List<Node>();
        const double UNCOMFORTABLE_DISTANCE_IN_PERCENTAGE = .05; //within 5% distance compared
to top left to bottom right of image
        public List<Edge> Edges;

        public Graph validGraph;

        /// <summary>
        /// Deep copy constructor for a graph
        /// </summary>
        /// <param name="other">The graph to copy</param>
        public Graph(Graph other)
        {
            //Copy nodes over
            for (int i = 0; i < other.Nodes.Count; i++)
            {
                Nodes.Add(new Node(other.Nodes[i]));
            }

            for (int i = 0; i < other.Nodes.Count; i++)
            {
                Node curNode = other.Nodes[i];
                for (int j = 0; j < curNode.Neighbors.Count; j++)
                {
                    Node neighborNode = curNode.Neighbors[j].GetNeighbor(curNode);
                    //If we don't have the connection yet, we want to add it
                    if (!Nodes[i].Neighbors.Any(t => t.GetNeighbor(Nodes[i]) ==
Nodes[other.Nodes.IndexOf(neighborNode)]))
                    {
                        Edge e = new Edge(Nodes[i], Nodes[other.Nodes.IndexOf(neighborNode)]);
                        Nodes[i].Neighbors.Add(e);
                        Nodes[other.Nodes.IndexOf(neighborNode)].Neighbors.Add(e);
                    }
                }
            }
            if (other.validGraph != null)
                validGraph = new Graph(other.validGraph);
        }

        public Graph(int numNodes, double edgeDensity, int minEdgesPerNode, int
maxEdgesPerNode, int width, int height, Random rand)
        {
            //Create all the nodes at random locations
            for (int i = 0; i < numNodes; i++)
            {
                Nodes.Add(new Node(rand.Next(0, width - 10), rand.Next(0, height - 10), width,
height));
            }

            double uncomfortableDistance = Math.Sqrt(Math.Pow(width, 2) + Math.Pow(height, 2))
* UNCOMFORTABLE_DISTANCE_IN_PERCENTAGE;
            double distanceToMove = uncomfortableDistance * .5; //half of the uncomfortable
distance for now
            bool isComfortable = true;
            do
            {
                isComfortable = true;
                for (int i = 0; i < Nodes.Count - 1; i++)
                {
                    for (int j = i + 1; j < Nodes.Count; j++)
                    {
                        if (Nodes[i].Distance(Nodes[j]) <= uncomfortableDistance)
                        {
                            isComfortable = false;
                            Nodes[i].MoveAwayFrom(Nodes[j], distanceToMove);
                            Nodes[j].MoveAwayFrom(Nodes[i], distanceToMove);
                        }
                    }
                }
            } while (!isComfortable);

            //At this point the graph has all the nodes that are spaced nicely. Time to add
neighbors.

            //Find out all the neighbors each node wants (for itself)
            Dictionary<Node, List<Node>> desiredNeighbors = new Dictionary<Node,
List<Node>>();
            for (int i = 0; i < Nodes.Count; i++)
            {
                int numNeighborsToAdd = Extensions.Clamp((int)Math.Round((numNodes - 1) *
edgeDensity), minEdgesPerNode, maxEdgesPerNode);
                List<Node> sortedByDistance = Nodes.OrderBy(t =>
t.Distance(Nodes[i])).ToList();
                sortedByDistance.RemoveAt(0);
                desiredNeighbors[Nodes[i]] =
sortedByDistance.Take(numNeighborsToAdd).ToList();
            }

            //Separate any agreed neighbors and non-agreed neighbors
            Dictionary<Node, List<Node>> commonDecisionNeighbors = new Dictionary<Node,
List<Node>>();
            Dictionary<Node, List<Node>> disagreeNeighbors = new Dictionary<Node,
List<Node>>();
            foreach(Node node in Nodes)
            {
                commonDecisionNeighbors.Add(node, new List<Node>());
                disagreeNeighbors.Add(node, new List<Node>());
            }

            foreach (var top in desiredNeighbors)
            {
                foreach (var other in top.Value)
                {
                    if (desiredNeighbors[other].Contains(top.Key))
                    {
                        if (!commonDecisionNeighbors[top.Key].Contains(other))
```

```
                            commonDecisionNeighbors[top.Key].Add(other);
                        if (!commonDecisionNeighbors[other].Contains(top.Key))
                            commonDecisionNeighbors[other].Add(top.Key);
                    }
                    else
                        disagreeNeighbors[top.Key].Add(other);
                }
            }

            //unhappy nodes are nodes that cannot be valid with neighbors they want to be, so
they have to be neighbors with others.
            List<Node> unhappyNodes = new List<Node>();
            //if any node is invalid, go through disagree neighbors and force it to make it
valid.
            foreach (var top in disagreeNeighbors)
            {
                //Invalid node, lets take some of the disagree neighbors to fix.
                if (commonDecisionNeighbors[top.Key].Count < minEdgesPerNode)
                {
                    int countNeededToFix = minEdgesPerNode -
commonDecisionNeighbors[top.Key].Count;
                    List<Node> canAdd = disagreeNeighbors[top.Key].Where(t =>
commonDecisionNeighbors[t].Count < maxEdgesPerNode).ToList();
                    if (canAdd.Count >= countNeededToFix)
                    {
                        List<Node> toAdd = canAdd.Take(countNeededToFix).ToList();
                        commonDecisionNeighbors[top.Key].AddRange(toAdd);
                        foreach(Node node in toAdd)
                            commonDecisionNeighbors[node].Add(top.Key);
                    }
                    else
                    {
                        unhappyNodes.Add(top.Key);
                    }
                }
            }

            for (int i = 0; i < unhappyNodes.Count; i++)
            {
                int countNeededToFix = minEdgesPerNode -
commonDecisionNeighbors[unhappyNodes[i]].Count;
                if (countNeededToFix <= 0)
                    continue; //Possibly fixed from a previous iteration of i

                //Cant add neighbors that are already added.
                List<Node> candidates = Nodes.Where(t =>
!commonDecisionNeighbors[unhappyNodes[i]].Contains(t) && commonDecisionNeighbors[t].Count <
maxEdgesPerNode).ToList();

                if (candidates.Count < countNeededToFix)
                    throw new Exception("Impossible graph setup. Try different variables.");

                //Sort candiates by distance and takes the number needed to fix the unhappy
node
                candidates = candidates.OrderBy(t =>
t.Distance(unhappyNodes[i])).Take(countNeededToFix).ToList();

                foreach(Node node in candidates)
                {
                    commonDecisionNeighbors[unhappyNodes[i]].Add(node);
                    commonDecisionNeighbors[node].Add(unhappyNodes[i]);
                }
            }
            //Last double check to make sure they agree
            foreach (var top in commonDecisionNeighbors)
            {
                if (top.Value.Count < minEdgesPerNode || top.Value.Count > maxEdgesPerNode)
                    throw new Exception("Some node has too many or too few neighbors.");
                foreach (var other in top.Value)
                {
                    if (!commonDecisionNeighbors[other].Contains(top.Key))
                        throw new Exception("commonDecisionNeighbors doesn't agree with 
itself... Something bad in the algorithm");
                }
            }

            //Build edges between nodes based on commonDecisionNeighbors
            List<Node> alreadyHandled = new List<Node>();
            foreach (var top in commonDecisionNeighbors)
            {
                alreadyHandled.Add(top.Key);
                foreach (var other in top.Value)
                {
                    if (!alreadyHandled.Contains(other))
                    {
                        Edge edge = new Edge(top.Key, other);
                        top.Key.Neighbors.Add(edge);
                        other.Neighbors.Add(edge);
                    }
                }
            }
        }

        public List<Edge> GetAllEdges()
        {
            if (Edges == null)
            {
                List<Edge> result = Nodes.Select(t => t.Neighbors).Aggregate(new List<Edge>(), 
(accumulator, next) => accumulator.Concat(next).ToList()).Distinct(new 
EdgeReferenceComparer()).ToList();
                Edges = result;
                return Edges;
            }
            return Edges;
        }

        /// <summary>
        /// Gets the number of edges in the graph
        /// </summary>
        public int GetEdgeCount()
        {
            return Edges?.Count ?? GetAllEdges().Count;
        }


/***********************************************************************************************
*******/
        //The following functions are used in calculating the graph heuristic.
        //The graph heuristic prioritizes the order in which we advance the graphs.

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

        /// <summary>
        /// This should run the algorithm to find a solution. The fitness score should be the 
time it takes to finish.
        /// </summary>
        /// <param name="totalColorWeight"></param>
        /// <param name="coloredWeight"></param>
        /// <param name="numEdgesNeighboringBlackWeight"></param>
        /// <returns></returns>
        public double Fitness(double totalColorWeight, double coloredWeight, double
numEdgesNeighboringBlackWeight)
        {
            double tccw = totalColorWeight * GetTotalColorCount();
            double ucw = coloredWeight * GetColoredCount();
            double nenbw = numEdgesNeighboringBlackWeight * GetNumEdgesNeighboringBlack();
            return tccw + ucw + nenbw;
        }

        /// <summary>
        /// This should run the algorithm to find a solution. The fitness score should be the
time it takes to finish.
        /// </summary>
        public double Solve(double[] genes)
        {
            //Stopwatch timer = new Stopwatch();
            //timer.Start();

            SortedList<double, Graph> toSearch = new SortedList<double, Graph>();
            toSearch.Add(this.Fitness((double)genes[0], (double)genes[1], (double)genes[2]),
new Graph(this));

            //Store the maximum degree of the nodes in this graph for future use
            double maxDegreeNode = this.Nodes.Max(t => t.GetNodeDegree());

            bool foundResult = false;
            int count = 0;
            while (toSearch.Any())
            {
                count++;
                Graph curGraph = toSearch.Pop(); //Take the most efficient graph according to
the heuristic
                //Find the most valuable nodes to check in curGraph
                List<Node> priorityNodes = curGraph.Nodes.OrderByDescending(t =>
t.Fitness((double)genes[3], (double)genes[4], maxDegreeNode)).Take(2).ToList();
                //Get the index of the priority nodes
                List<int> priorityNodeIndexes = priorityNodes.Select(t =>
curGraph.Nodes.IndexOf(t)).ToList();

                for (int i = 0; i < priorityNodeIndexes.Count; i++)
                {
                    Graph newGraph = new Graph(curGraph);
                    //If after the advancement we have a graph that is satisfied, we are done.
Note, this result is required to use <= 4 colors.
                    bool? result = newGraph.AdvanceOneStage(priorityNodeIndexes[i]);
                    if (result == null)
                    {
                        //The result is trying to use more than 4 colors. Don't add this to
the search list
                        break;
                    }

                    if (result.Value)
                    {
                        foundResult = true;
                        //We have found a solution, so set the nodes and edges of the solution
to "this" so that we can display it.
                        if (validGraph == null)
                            validGraph = new Graph(newGraph);
                        else if (validGraph.GetTotalColorCount() >
newGraph.GetTotalColorCount())
                            validGraph = new Graph(newGraph);
                        break;
                    }
                    toSearch.SafeAdd(newGraph.Fitness((double)genes[0], (double)genes[1],
(double)genes[2]), newGraph);
                }
                //No sense continuing if we have found the result. Note, this result is
required to use <= 4 colors.
                if (foundResult)
                {
                    break;
                }
            }

            if (!foundResult)
                throw new Exception("There is always a result.");

            //timer.Stop();
            return count;
            //return timer.ElapsedTicks; //inverted fitness, lower is better
        }

        /// <summary>
        /// Advances this graph one stage by altering the node at the given index.
        /// Node is colored the first available color in the colorOrder list
        /// </summary>
        /// <param name="index">Index of the node to alter</param>
        public bool? AdvanceOneStage(int index)
        {
            Color[] neighborColors = Nodes[index].Neighbors.Select(t =>
t.GetNeighbor(Nodes[index]).Color).ToArray();
            bool foundColor = false;
            foreach (Color c in colorOrder)
            {
                if (!neighborColors.Contains(c))
                {
                    Nodes[index].Color = c;
                    foundColor = true;
                    break;
                }
            }

            if (!foundColor)
            {
                return null;
            }

            //After coloring the node, validate to see if it satisfied the graph
            return Validate();
        }

        /// <summary>
        /// Verifies that the graph is actually solved
        /// Doing this way because this form of validation is much faster than comparing all
edges.
        /// Only compare edges if we are finished with the graph.
        /// </summary>
        /// <returns>True if it is satisfied, false otherwise</returns>
        private bool Validate()
        {
            //If no more uncolored
            if (GetColoredCount() == Nodes.Count)
            {
                //If satisfies theorem https://en.wikipedia.org/wiki/Four_color_theorem
                if (GetTotalColorCount() <= 4)
                {
                    //If no two neighbors have the same color
                    if (InDepthValidate())
                    {
                        return true;
                    }
                    else
                    {
                        throw new Exception("Logic error. Should never pass
GetUncoloredCount() == 0 and then fail InDepthValidate().");
                    }
                }
            }
            return false;
        }

        /// <summary>
        /// Checks each edge to make sure the graph is logically satisfied
        /// </summary>
        /// <returns>True if it is satisfied, false otherwise</returns>
        private bool InDepthValidate()
        {
            foreach (var edge in GetAllEdges())
            {
                if (edge.Nodes[0].Color == edge.Nodes[1].Color)
                    return false;
            }
            return true;
        }


        /// <summary>
        /// Total number of distinct colors used. Lower is better
        /// If this is a high value compared to other branches, our heuristic should stray
away from it
        /// </summary>
        public double GetTotalColorCount() => Nodes.Select(t => t.Color).Distinct().Count();

        /// <summary>
        /// Number of nodes that have been colored
        /// Low uncolored count combined with a low total color count is a respectable graph.
        /// </summary>
        public double GetColoredCount() => Nodes.Where(t => t.Color != Color.Black).Count();

        /// <summary>
        /// The number of edges that have at least one black node connection
        /// A high number here likely means total color count is going to go up for this
graph.
        /// Example: all nodes but 1 are colored, but it has 8 edges connected to it. Very
high chance it is going to have to be a new color.
        /// </summary>
        public double GetNumEdgesNeighboringBlack() => GetAllEdges().Where(t =>
t.IsNeighboringBlack()).Count();
    }

    public class Node
    {
        public Color Color { get; set; } = Color.Black;
        public int X { get; set; }
        public int Y { get; set; }
        public int Width { get; set; } //used for generating the map, not in the bnb algorithm
        public int Height { get; set; } //used for generating the map, not in the bnb
algorithm

        public bool DoneWithNeighbors { get; set; } = false; //used for generating the map,
not in the bnb algorithm

        public List<Edge> Neighbors { get; set; } = new List<Edge>();

        /// <summary>
        /// partial deep copy constructor for node
        /// Does not copy neighbors within the constructor
        /// Used for the algorithm, not generation
        /// </summary>
        public Node(Node other)
        {
            this.Color = other.Color;
            this.X = other.X;
            this.Y = other.Y;
        }

        public Node(int x, int y, int width, int height)
        {
            X = x;
            Y = y;
            Width = width;
            Height = height;
        }

        public double Distance(Node other)
        {
            return Math.Sqrt(Math.Pow(other.X - X, 2) + Math.Pow(other.Y - Y, 2));
        }

        public double Distance(Point point)
        {
            return Math.Sqrt(Math.Pow(point.X - X, 2) + Math.Pow(point.Y - Y, 2));
        }

        public void MoveAwayFrom(Node other, double distance)
        {
            Vector2 direction = new Vector2(X - other.X, Y - other.Y);
            direction.Normalize();
            direction.X *= distance;
            direction.Y *= distance;

            X = Extensions.Clamp(X + (int)Math.Round(direction.X), 0, Width - 10);
            Y = Extensions.Clamp(Y + (int)Math.Round(direction.Y), 0, Height - 10);
        }


/*********************************************************************************************
*******/
        //The following functions are used in calculating the node heuristics
        //The node heuristic prioritizes which node in the graph we want to try to color first

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

        /// <summary>
        /// Gets the fitness of the given node to compare against other nodes in the graph
        /// </summary>
        /// <param name="uncoloredNeighborWeight">Weight determined by heuristic for uncolored
neighbor interest</param>
```

```
        /// <param name="nodeDegreeWeight">Weight determined by heuristic for node degree
interest</param>
        /// <param name="maxDegreeNode">The node with the highest degree's degree</param>
        /// <returns>The fitness</returns>
        public double Fitness(double uncoloredNeighborWeight, double nodeDegreeWeight, double
maxDegreeNode)
        {
            //If this node has already been colored. We need not consider it.
            if (this.Color != Color.Black)
                return 0;
            double unw = ((uncoloredNeighborWeight * GetUncoloredNeighborCount()) *
maxDegreeNode) / GetNodeDegree(); //normalized with maxDegreeNode
            double ndw = ((nodeDegreeWeight * GetNodeDegree()) * maxDegreeNode) /
GetNodeDegree();//normalized with maxDegreeNode
            return unw + ndw;
        }

        /// <summary>
        /// Number of uncolored neighbors. Lower the better
        /// If this is 0, then we know exactly what colors this node cannot be
        /// </summary>
        public double GetUncoloredNeighborCount() => Neighbors.Where(t =>
t.GetNeighbor(this).Color == Color.Black).Count();

        /// <summary>
        /// The number of neighbors this node has
        /// Generally you want to color high degrees first
        /// </summary>
        public double GetNodeDegree() => Neighbors.Count();
    }

    /// <summary>
    /// Edge class. Only holds nodes as we don't care about distance in this problem.
    /// </summary>
    public class Edge : IEqualityComparer<Edge>
    {
        public Node[] Nodes { get; set; } = new Node[2];

        public Edge(Node first, Node second)
        {
            Nodes[0] = first;
            Nodes[1] = second;
        }

        public Node GetNeighbor(Node self)
        {
            if (Nodes[0] == self)
                return Nodes[1];
            return Nodes[0];
        }

        public bool Equals(Edge x, Edge y)
        {
            if (x.Nodes[0] == y.Nodes[0] && x.Nodes[1] == y.Nodes[1])
                return true;
            if (x.Nodes[0] == y.Nodes[1] && x.Nodes[1] == y.Nodes[0])
                return true;
            return false;
        }

        public int GetHashCode(Edge obj)
        {
            return obj.GetHashCode();
        }
        /// <summary>
        /// Should we alter to only consider if it is not double black?
        /// </summary>
        public bool IsNeighboringBlack()
        {
            if (Nodes[0].Color == Color.Black)
                return true;
            if (Nodes[1].Color == Color.Black)
                return true;
            return false;
        }
    }

    class EdgeReferenceComparer : IEqualityComparer<Edge>
    {
        public bool Equals(Edge x, Edge y)
        {
            return x == y;
        }

        public int GetHashCode(Edge obj)
        {
            return obj.GetHashCode();
        }
    }


    class Vector2
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Vector2(double x, double y)
        {
            X = x;
            Y = y;
        }

        public void Normalize()
        {
            double magnitude = Math.Sqrt(Math.Pow(X, 2) + Math.Pow(Y, 2));
            X /= magnitude;
            Y /= magnitude;
        }
    }

}
```

MapColoring.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace MapColoring
{
    class MapColoring
    {
        /// <summary>
        /// Solve the graph with the given genes.
        /// Used for testing individual gene sets
        /// </summary>
        /// <returns>Fitness score of the given graph</returns>
        public static double Solve(Graph graph, int numBranchesToSplit,
            double getTotalColorCountWeight,
            double getUncoloredCountWeight,
            double getNumEdgesNeighboringBlackWeight,
            double getUncoloredNeighborCountWeight,
            double getNodeDegreeWeight)
        {
            double[] genes = new double[] { numBranchesToSplit, getTotalColorCountWeight,
getUncoloredCountWeight, getNumEdgesNeighboringBlackWeight, getUncoloredNeighborCountWeight,
getNodeDegreeWeight };

            return graph.Solve(genes);
        }

        /// <summary>
        /// Solve the graph with the given genes.
        /// Used for testing individual gene sets
        /// </summary>
        /// <returns>Fitness score of the given graph</returns>
        public static double Solve(Graph graph, double[] genes)
        {
            if (genes.Length != 6)
                throw new Exception("Map coloring algorithm uses 6 genes.");
            return graph.Solve(genes);
        }
    }
}
```

FormMapColoring.cs

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

namespace MapColoring
{
    public partial class FormMapColoring : Form
    {
        Random rand = new Random();

        Dictionary<Control, string> toolTips;

        Graph graph;

        public FormMapColoring()
        {
            InitializeComponent();

            //Set tooltip strings for all input fields
            toolTips = new Dictionary<Control, string>()
            {
                { TxtBx_NumCountries, "The number of countries to generate" },
                { TxtBx_EdgeDensity, "The percentage of connections between countries. "+
Environment.NewLine+".5 = each country is neighbors to roughly 1/2 all countries. "+
Environment.NewLine+"1 = every country is connected to every other country."+
Environment.NewLine+"This is overruled by min and max edge requirements."  },
                { TxtBx_MaxEdgesPerCountry, "Maximum amount of edges a country can have." },
                { TxtBx_MinEdgesPerCountry, "Minimum amount of edges a country can have. Must
be less than Num Countries" },
                { TxtBx_PopulationSize , "Must be in the form of Size = X + Y where x = y(y-
1)/2) where y is the number of chromosomes kept each iteration to breed." +
Environment.NewLine + "Some valid values: 36, 136, 528, 2080" },
                { TxtBx_Convergence, "The percentage of convergence at which we stop the
algorithm because all chromosomes are within this range of each other." },
                { TxtBx_DefaultGeneValue, "All genes will start at this value, and will be
mutated once prior to calculating fitness." },
                { TxtBx_MaxIterations, "Maximum amount of times the algorithm will reproduce."
},
                { TxtBx_PercentChromosomesMutated, "% chance a chromosome will be selected to
mutate." + Environment.NewLine + "Some chromosomes are left the same if not 100%" },
                { TxtBx_PercentGenesMutated, "After a chromosome has been selected to mutate,
this is the percentage chance for each gene to actually be mutated." },
                { TxtBx_PercentMutationDeviation, "How much off of the value can we mutate." +
Environment.NewLine + ".5 means +- 50%, IE: 90 -> 45-135 and 180 -> 90-270" },
                { TxtBx_Seed, "Random seed used to generate same results every time you run
the algorithm." },
            };

            //Add mouseover and leave events for each control that has a tooltip associated
with it
            foreach (KeyValuePair<Control, string> tip in toolTips)
            {
                ToolTip tt = new ToolTip();
                tip.Key.MouseEnter += (s, o) =>
                {
                    tt = new ToolTip();
                    tt.InitialDelay = 300;
                    tt.Show(tip.Value, tip.Key, 0);
                };

                tip.Key.MouseLeave += (s, o) =>
                {
                    tt.Dispose();
                };

            }

            //Load the previous runs values into the fields
            if (File.Exists(AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/MapColoring/MostRecentRun.rd"))
            {
                try
                {
                    using (StreamReader sr = new
StreamReader(AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/MapColoring/MostRecentRun.rd"))
                    {
                        TxtBx_NumCountries.Text = sr.ReadLine();
                        TxtBx_EdgeDensity.Text = sr.ReadLine();
                        TxtBx_MaxEdgesPerCountry.Text = sr.ReadLine();
                        TxtBx_MinEdgesPerCountry.Text = sr.ReadLine();
                        TxtBx_Seed.Text = sr.ReadLine();
                        TxtBx_PopulationSize.Text = sr.ReadLine();
                        TxtBx_Convergence.Text = sr.ReadLine();
                        TxtBx_DefaultGeneValue.Text = sr.ReadLine();
                        TxtBx_MaxIterations.Text = sr.ReadLine();
                        TxtBx_PercentChromosomesMutated.Text = sr.ReadLine();
                        TxtBx_PercentGenesMutated.Text = sr.ReadLine();
                        TxtBx_PercentMutationDeviation.Text = sr.ReadLine();
                    }
                }
                catch { /*Just leave values at 0 if the above crashes (someone messed with the
file or debugging issues, should fix after next run) */ }
            }

            Btn_GenerateGraph_Click(null, null);
        }

        /// <summary>
        /// Logic of the algorithm
        /// </summary>
        private void GenerateGraph()
        {
            if (!string.IsNullOrEmpty(TxtBx_Seed.Text))
                rand = new Random(int.Parse(TxtBx_Seed.Text));

            graph = new Graph(int.Parse(TxtBx_NumCountries.Text),
double.Parse(TxtBx_EdgeDensity.Text),
                int.Parse(TxtBx_MinEdgesPerCountry.Text),
int.Parse(TxtBx_MaxEdgesPerCountry.Text),
                PictureBox_Graph.Width, PictureBox_Graph.Height, rand);
            DrawGraph(graph);
            if (!TestGraph())
            {
                throw new Exception("Graph did not pass test. Duplicate edges");
            }
        }
        private void DrawGraph(Graph graph)
        {
            Dictionary<Color, Brush> brushColors = new Dictionary<Color, Brush>();
            brushColors.Add(Color.Black, new SolidBrush(Color.Black));
            foreach (Color c in Graph.colorOrder)
            {
                brushColors.Add(c, new SolidBrush(c));
            }
            Pen pen = new Pen(Color.Black);
            Bitmap image = new Bitmap(PictureBox_Graph.Width, PictureBox_Graph.Height);
            Graphics g = Graphics.FromImage(image);

            foreach (var edge in graph.Nodes.SelectMany(t => t.Neighbors).Distinct())
            {
                g.DrawLine(pen, edge.Nodes[0].X, edge.Nodes[0].Y, edge.Nodes[1].X,
edge.Nodes[1].Y);
            }

            foreach (var node in graph.Nodes)
            {
                g.FillEllipse(brushColors[node.Color], node.X - 10, node.Y - 10, 20, 20);
//TODO: Make the width and height scale based on image size and numnodes
            }

            PictureBox_Graph.Image = image;
        }

        /// <summary>
        /// Test the graph to make sure it doesn't have duplicate edges
        /// </summary>
        private bool TestGraph()
        {
            this.Update();
            List<Edge> edges = graph.GetAllEdges();

            if (edges.Distinct().Count() != edges.Count())
                return false;

            return true;
        }

        /// <summary>
        /// The randomize parameters button has been clicked.
        /// Randomize the values of the parameter fields
        /// </summary>
        private void Btn_RandomizeParameters_Click(object sender, EventArgs e)
        {
            TxtBx_NumCountries.Text = rand.Next(10, 30).ToString();

            TxtBx_EdgeDensity.Text = rand.NextDouble().ToString("#.###");
            TxtBx_MaxEdgesPerCountry.Text = rand.Next(6, 10).ToString();
            TxtBx_MinEdgesPerCountry.Text = rand.Next(1, 6).ToString();

            int y = rand.Next(10, 50);
            TxtBx_PopulationSize.Text = (y + (y * (y - 1) / 2)).ToString();
            TxtBx_Convergence.Text = rand.NextDouble().ToString("#.###");
            TxtBx_DefaultGeneValue.Text = (rand.NextDouble() * 10).ToString("#.###");
            TxtBx_MaxIterations.Text = 1000.ToString();
            TxtBx_PercentChromosomesMutated.Text = rand.NextDouble().ToString("#.###");
            TxtBx_PercentGenesMutated.Text = rand.NextDouble().ToString("#.###");
            TxtBx_PercentMutationDeviation.Text = rand.NextDouble().ToString("#.###");
        }

        private void Btn_GenerateGraph_Click(object sender, EventArgs e)
        {
            if (IsParameterError())
                return;

            //Save our current parameters to be spawned next time we run the program
            string destination = AppDomain.CurrentDomain.BaseDirectory +
@"/StoredVariables/MapColoring";
            if (!Directory.Exists(destination))
                Directory.CreateDirectory(destination);

            using (StreamWriter sw = new StreamWriter(new FileStream(destination +
"/MostRecentRun.rd", FileMode.Create)))
            {
                sw.WriteLine(TxtBx_NumCountries.Text);
                sw.WriteLine(TxtBx_EdgeDensity.Text);
                sw.WriteLine(TxtBx_MaxEdgesPerCountry.Text);
                sw.WriteLine(TxtBx_MinEdgesPerCountry.Text);
                sw.WriteLine(TxtBx_Seed.Text);
                sw.WriteLine(TxtBx_PopulationSize.Text);
                sw.WriteLine(TxtBx_Convergence.Text);
                sw.WriteLine(TxtBx_DefaultGeneValue.Text);
                sw.WriteLine(TxtBx_MaxIterations.Text);
                sw.WriteLine(TxtBx_PercentChromosomesMutated.Text);
                sw.WriteLine(TxtBx_PercentGenesMutated.Text);
                sw.WriteLine(TxtBx_PercentMutationDeviation.Text);
            }

            GenerateGraph();
        }

        private bool IsParameterError()
        {
            string errors = "";
            double resultD;
            int resultI;

            //map coloring default parameters
            if (!int.TryParse(TxtBx_NumCountries.Text, out resultI))
                errors += "Invalid value for Num Countries" + Environment.NewLine;
            if (!double.TryParse(TxtBx_EdgeDensity.Text, out resultD) && 1 >= resultD &&
resultD >= 0)
                errors += "Invalid value for Edge Density (0-1)" + Environment.NewLine;
            if (!int.TryParse(TxtBx_MaxEdgesPerCountry.Text, out resultI))
                errors += "Invalid value for Max Edges Per Country" + Environment.NewLine;
            if (!int.TryParse(TxtBx_MaxEdgesPerCountry.Text, out resultI))
                errors += "Invalid value for Min Edges Per Country" + Environment.NewLine;

            //map coloring conflict parameters
            if (string.IsNullOrEmpty(errors))
            {
                if (int.Parse(TxtBx_NumCountries.Text) <=
int.Parse(TxtBx_MinEdgesPerCountry.Text))
                    errors += "Min edges must be less than num countries" +
Environment.NewLine;
                if (int.Parse(TxtBx_MaxEdgesPerCountry.Text) <
int.Parse(TxtBx_MinEdgesPerCountry.Text))
                    errors += "Min edges cannot be greater than max edges" +
Environment.NewLine;
            }
            else
            {
                errors += "Error within map generating parameters, cannot check parameter
conflicts" + Environment.NewLine;
            }
            //genetic algorithm parameters
            if (!int.TryParse(TxtBx_PopulationSize.Text, out resultI) ||
!ValidatePopulationSize(int.Parse(TxtBx_PopulationSize.Text)))
                errors += "Invalid value for Population Size" + Environment.NewLine;
            if (!double.TryParse(TxtBx_Convergence.Text, out resultD))
                errors += "Invalid value for Convergence" + Environment.NewLine;
            if (!double.TryParse(TxtBx_DefaultGeneValue.Text, out resultD))
                errors += "Invalid value for Default Gene Value" + Environment.NewLine;
            if (!int.TryParse(TxtBx_MaxIterations.Text, out resultI) &&
!string.IsNullOrEmpty(TxtBx_MaxIterations.Text))
                errors += "Invalid value for Max Iterations" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentChromosomesMutated.Text, out resultD))
                errors += "Invalid value for % Chromosomes Mutated" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentGenesMutated.Text, out resultD))
                errors += "Invalid value for % Genes Mutated" + Environment.NewLine;
            if (!double.TryParse(TxtBx_PercentMutationDeviation.Text, out resultD))
                errors += "Invalid value for % Deviation Mutation" + Environment.NewLine;
            if (!int.TryParse(TxtBx_Seed.Text, out resultI) &&
!string.IsNullOrEmpty(TxtBx_Seed.Text))
                errors += "Invalid value for Seed" + Environment.NewLine;

            if (!string.IsNullOrEmpty(errors))
            {
                MessageBox.Show(errors);
                return true;
            }
            return false;
        }

        /// <summary>
        /// Verifies that the population size follows the formula size = y + y(y-1)/2
        /// Uses the quadratic forumla to verify with a = -.5, b = -.5 and c = size
        /// </summary>
        /// <param name="size"></param>
        /// <returns></returns>
        private bool ValidatePopulationSize(int size)
        {
            int y;
            if (!int.TryParse(((.5 - Math.Sqrt(.25 - (4 * -.5 * size))) / -1).ToString(), out
y))
            {
                return false;
            }
            return (size == y + y * (y - 1) / 2);
        }

        private async void Btn_RunAlgorithm_Click(object sender, EventArgs e)
        {
            if (IsParameterError())
                return;

            Btn_RandomizeParameters.Enabled = false;
            Btn_GenerateGraph.Enabled = false;
            Btn_SolveGraph.Enabled = false;
            Btn_RunAlgorithm.Enabled = false;

            List<Tuple<int, double, double, double, double>> dataPoints = new List<Tuple<int,
double, double, double, double>>();
            List<Tuple<int, List<double>>> selectedChromosomesFitness = new List<Tuple<int,
List<double>>>();
            GeneticAlgorithm.GeneticAlgorithmClass.UpdateProgressBar = (percent, convergence,
iteration, averageFitness, minimumFitness, maximumFitness, selectedFitness) =>
            {
                PB_MapColoring.Invoke((MethodInvoker)delegate ()
                {
                    PB_MapColoring.Value = percent;
                    LBL_Convergence.Text = convergence.ToString();
                    LBL_Iteration.Text = iteration.ToString();
                    dataPoints.Add(new Tuple<int, double, double, double, double>(iteration,
convergence, averageFitness, minimumFitness, maximumFitness));
                    selectedChromosomesFitness.Add(selectedFitness);
                });
            };
            int logInterval = 1;
            List<GeneticAlgorithm.Chromosome> chromosomes = await Task.Factory.StartNew(() =>
            {
                return GeneticAlgorithm.GeneticAlgorithmClass.RunGeneticAlgorithm(
                   int.Parse(TxtBx_PopulationSize.Text),
                   double.Parse(TxtBx_Convergence.Text),
                   new double[] { double.Parse(TxtBx_DefaultGeneValue.Text),
double.Parse(TxtBx_DefaultGeneValue.Text), double.Parse(TxtBx_DefaultGeneValue.Text),
double.Parse(TxtBx_DefaultGeneValue.Text), double.Parse(TxtBx_DefaultGeneValue.Text) },
                   double.Parse(TxtBx_PercentChromosomesMutated.Text),
                   double.Parse(TxtBx_PercentGenesMutated.Text),
                   double.Parse(TxtBx_PercentMutationDeviation.Text),
                   graph.Solve,
                   int.Parse(TxtBx_MaxIterations.Text),
                   logInterval,
                   false,
                   string.IsNullOrEmpty(TxtBx_Seed.Text) ? 0 : int.Parse(TxtBx_Seed.Text));
            });


            Btn_RandomizeParameters.Enabled = true;
            Btn_GenerateGraph.Enabled = true;
            Btn_SolveGraph.Enabled = true;
            Btn_RunAlgorithm.Enabled = true;

            DrawGraph(graph.validGraph);

            GenerateResultsDataGridView(chromosomes);
            Common.FormResults form = new Common.FormResults();
            if (form.InitializeChart(dataPoints.Select(t => t.Item1).ToList(),
dataPoints.Select(t => t.Item2).ToList(), dataPoints.Select(t => t.Item3).ToList(),
dataPoints.Select(t => t.Item4).ToList(), dataPoints.Select(t => t.Item5).ToList(),
selectedChromosomesFitness, logInterval))
            {
                form.Show();
            }
            else
            {
                MessageBox.Show("Not enough iterations to show results.");
            }
        }

        private void GenerateResultsDataGridView(List<GeneticAlgorithm.Chromosome> solutions)
        {
            DataTable table = new DataTable("Final Evolution");
            table.Columns.Add("id");
            table.Columns.Add("Duration");
            table.Columns.Add("Total Color Count Gene");
            table.Columns.Add("Colored Nodes Gene");
            table.Columns.Add("Num Edges Neighboring Black Gene");
            table.Columns.Add("Uncolored Neighbor Count Gene");
            table.Columns.Add("Node Degree Gene");

            List<List<double>> genes = new List<List<double>>();
            solutions.ForEach(t => genes.Add(t.Genes.Select(x => (double)x /
(double)t.Genes.Max()).ToList()));


            table.Rows.Add(new object[] {0, "AVERAGE",
                genes.Average(t => (double)t[0]),
                genes.Average(t => (double)t[1]),
                genes.Average(t => (double)t[2]),
                genes.Average(t => (double)t[3]),
                genes.Average(t => (double)t[4])});

            for (int i = 0; i < solutions.Count; i++)
            {
                double[] normalizedGenes = solutions[i].Genes.Select(t => (double)t /
(double)solutions[i].Genes.Max()).ToArray();
                table.Rows.Add(new object[] { i + 1,
                    solutions[i].FitnessScore,
                    normalizedGenes[0].ToString("#.###"),
                    normalizedGenes[1].ToString("#.###"),
                    normalizedGenes[2].ToString("#.###"),
                    normalizedGenes[3].ToString("#.###"),
                    normalizedGenes[4].ToString("#.###")
                });
            }

            DGV_Results.DataSource = table;
            for (int i = 0; i < DGV_Results.Columns.Count; i++)
            {
                DGV_Results.Columns[i].AutoSizeMode =
DataGridViewAutoSizeColumnMode.ColumnHeader;
            }
        }

        private void Btn_SolveGraph_Click(object sender, EventArgs e)
        {
            FormSolveGraph form = new FormSolveGraph();
            form.GenerateGraph(graph);
            form.Show();
        }
    }
}
```

FormRandomGame.cs

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

namespace RandomGame
{
    public partial class FormRandomGame : Form
    {
        //This should be plenty of colors, will never want to try more than this
        List<Color> colors = new List<Color>() { Color.Red, Color.Blue, Color.Green, 
Color.Yellow, Color.Purple, Color.Aqua, Color.Pink, Color.Orange, Color.Gray};

        List<Graph> Graphs { get; set; } = new List<Graph>();

        Random rand = new Random();

        public FormRandomGame()
        {
            InitializeComponent();

            //Load the previous runs values into the fields
            if (File.Exists(AppDomain.CurrentDomain.BaseDirectory + 
@"/StoredVariables/RandomGame/MostRecentRun.rd"))
            {
                try
                {
                    using (StreamReader sr = new 
StreamReader(AppDomain.CurrentDomain.BaseDirectory + 
@"/StoredVariables/RandomGame/MostRecentRun.rd"))
                    {
                        TxtBx_NumberOfNodes.Text = sr.ReadLine();
                        TxtBx_MaxEdgesPerNode.Text = sr.ReadLine();
                        TxtBx_MinEdgesPerNode.Text = sr.ReadLine();
                        TxtBx_RandomSeed.Text = sr.ReadLine();
                        TxtBx_NumberOfIterations.Text = sr.ReadLine();
                        TxtBx_NumberOfConstraintsPerEdge.Text = sr.ReadLine();
                        TxtBx_MaxNumberOfColorsAllowed.Text = sr.ReadLine();
                    }
                }
                catch { /*Just leave values at 0 if the above crashes (someone messed with the 
file or debugging issues, should fix after next run) */ }
            }
        }

        private void Btn_Start_Click(object sender, EventArgs e)
        {
            //When start, save these values
            string destination = AppDomain.CurrentDomain.BaseDirectory + 
@"/StoredVariables/RandomGame";
            if (!Directory.Exists(destination))
                Directory.CreateDirectory(destination);

            using (StreamWriter sw = new StreamWriter(new FileStream(destination + 
"/MostRecentRun.rd", FileMode.Create)))
```

```
            {
                sw.WriteLine(TxtBx_NumberOfNodes.Text);
                sw.WriteLine(TxtBx_MaxEdgesPerNode.Text);
                sw.WriteLine(TxtBx_MinEdgesPerNode.Text);
                sw.WriteLine(TxtBx_RandomSeed.Text);
                sw.WriteLine(TxtBx_NumberOfIterations.Text);
                sw.WriteLine(TxtBx_NumberOfConstraintsPerEdge.Text);
                sw.WriteLine(TxtBx_MaxNumberOfColorsAllowed.Text);
            }


            Random rand = string.IsNullOrEmpty(TxtBx_RandomSeed.Text) ? new Random() : new
Random(int.Parse(TxtBx_RandomSeed.Text));
            Graph gra = new Graph(int.Parse(TxtBx_NumberOfNodes.Text),
int.Parse(TxtBx_MinEdgesPerNode.Text), int.Parse(TxtBx_MaxEdgesPerNode.Text),
PictureBox_Graph.Width, PictureBox_Graph.Height, rand);
            Graphs =
gra.ExhaustiveSolve(colors.Take(int.Parse(TxtBx_MaxNumberOfColorsAllowed.Text)).ToArray());

            //populate edges for debuggings sake
            Graphs.ForEach(t => t.GetAllEdges());

            Label_GraphsCount.Text = Graphs.Count.ToString();
            DrawGraph(Graphs[0]);
            currentDrawIndex = 0;

            //Now that the graphs are generated, apply edge constraints to them
            //Create the edge constraints
            Tuple<string, List<Func<Graph, int, bool>>>[] constraints = new Tuple<string,
List<Func<Graph, int, bool>>>[Graphs[0].GetAllEdges().Count];
            List<Color> possibleColors =
colors.Take(int.Parse(TxtBx_MaxNumberOfColorsAllowed.Text)).ToList();
            List<string> textBuilderForPrinting = new List<string>();

            for (int i = 0; i < Graphs[0].GetAllEdges().Count; i++)
            {
                int selection = rand.Next(1, 5); //1-4 corresponding to the constraint methods
                switch (selection)
                {
                    case 1:
                        constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("",
new List<Func<Graph, int, bool>>() { (g, edge) => Constraint_MustBeDifferentColor(g, edge) });
                        textBuilderForPrinting.Add($"Edge {i} constraint:
Constraint_MustBeDifferentColor");
                        break;
                    case 2:
                        constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("",
new List<Func<Graph, int, bool>>() { (g, edge) => Constraint_MustBeSameColor(g, edge) });
                        textBuilderForPrinting.Add($"Edge {i} constraint:
Constraint_MustBeSameColor");
                        break;
                    case 3:
                        StringBuilder sb = new StringBuilder();
                        List<Func<Graph, int, bool>> mustBeOneOfEachCons = new
List<Func<Graph, int, bool>>();
                        for (int j = 0; j < rand.Next(2, 4); j++) //rand 2 or 3
                        {
                            Color a = possibleColors[rand.Next(0, possibleColors.Count)];
                            Color b = possibleColors[rand.Next(0, possibleColors.Count)];
                            mustBeOneOfEachCons.Add((g, edge) => Constraint_MustBeOneOfEach(g,
edge, a, b));
                            sb.Append($"Constraint_MustBeOneOfEach({a.ToString()},
{b.ToString()}) - ");
                        }
                        constraints[i] = new Tuple<string, List<Func<Graph, int, bool>>>("OR",
mustBeOneOfEachCons);
                        textBuilderForPrinting.Add($"Edge {i} constraint: {sb.ToString()}");
                        break;
                    case 4:
                        StringBuilder sb2 = new StringBuilder();
                        List<Func<Graph, int, bool>> mustNotBeOneOfEachCons = new
List<Func<Graph, int, bool>>();
                        for (int j = 0; j < rand.Next(2, 4); j++) //rand 2 or 3
                        {
                            Color a = possibleColors[rand.Next(0, possibleColors.Count)];
                            Color b = possibleColors[rand.Next(0, possibleColors.Count)];
                            mustNotBeOneOfEachCons.Add((g, edge) =>
Constraint_MustNotBeOneOfEach(g, edge, a, b));
                            sb2.Append($"Constraint_MustNotBeOneOfEach({a.ToString()},
{b.ToString()}) - ");
                        }
                        constraints[i] = new Tuple<string, List<Func<Graph, int,
bool>>>("AND", mustNotBeOneOfEachCons);
                        textBuilderForPrinting.Add($"Edge {i} constraint: {sb2.ToString()}");
                        break;
                }
            }

            TxtBx_EdgeConstraints.Lines = textBuilderForPrinting.ToArray();
            List<Tuple<int, int>> results = new List<Tuple<int, int>>();
            //Now all the edges have randomized constraints on them, so run the constraints
against every single graph to find the interesting values
            foreach (var graph in Graphs)
            {
                int passCount = 0;
                int failCount = 0;
                for (int i = 0; i < graph.GetAllEdges().Count; i++) //GetAllEdges optimized to
only be expensive once.
                {
                    if (constraints[i].Item1 == "")
                    {
                        if (constraints[i].Item2[0](graph, i))
                            passCount++;
                        else
                            failCount++;
                    }
                    else if (constraints[i].Item1 == "OR")
                    {
                        if (constraints[i].Item2.Any(t => t(graph, i)))
                            passCount++;
                        else
                            failCount++;
                    }
                    else if (constraints[i].Item1 == "AND")
                    {
                        if (constraints[i].Item2.All(t => t(graph, i)))
                            passCount++;
                        else
                            failCount++;
                    }
                }
                results.Add(new Tuple<int, int>(passCount, failCount));
            }

            listViewSatisfiabilityResults.Items.Clear();

            //We have our results from the constraints on the edges, now time to show some
results to the user
```

```
            for(int i = 0; i < results.Count; i++)
            {
                double satisfiability = ((double)results[i].Item1 / ((double)results[i].Item1
+ (double)results[i].Item2));
                if (satisfiability >= .8)
                {

                    listViewSatisfiabilityResults.Items.Add(new ListViewItem(new string[] {
i.ToString(), satisfiability.ToString() }));
                }
            }

            FormResults fResults = new FormResults();
            fResults.Show();
            fResults.Initialize(results);

            //Might want to do something with later
            //double highestValue = results.Max(t => t.Item1 / (t.Item1 + t.Item2));
        }

        //1
        public bool Constraint_MustBeDifferentColor(Graph g, int edge)
        {
            return g.GetAllEdges()[edge].Nodes[0].Color !=
g.GetAllEdges()[edge].Nodes[1].Color;
        }

        //2
        public bool Constraint_MustBeSameColor(Graph g, int edge)
        {
            return g.GetAllEdges()[edge].Nodes[0].Color ==
g.GetAllEdges()[edge].Nodes[1].Color;
        }

        //3
        public bool Constraint_MustBeOneOfEach(Graph g, int edge, Color a, Color b)
        {
            return g.GetAllEdges()[edge].Nodes[0].Color == a &&
g.GetAllEdges()[edge].Nodes[1].Color == b
                || g.GetAllEdges()[edge].Nodes[0].Color == b &&
g.GetAllEdges()[edge].Nodes[1].Color == a;
        }

        //4
        public bool Constraint_MustNotBeOneOfEach(Graph g, int edge, Color a, Color b)
        {
            return !Constraint_MustBeOneOfEach(g, edge, a, b);
        }



        private void DrawGraph(Graph graph)
        {
            Dictionary<Color, Brush> brushColors = new Dictionary<Color, Brush>();
            brushColors.Add(Color.Black, new SolidBrush(Color.Black));
            foreach (Color c in colors)
            {
                brushColors.Add(c, new SolidBrush(c));
            }
            Pen pen = new Pen(Color.Black);
            Bitmap image = new Bitmap(PictureBox_Graph.Width, PictureBox_Graph.Height);
            Font font = new Font("Arial", 12);
            Graphics g = Graphics.FromImage(image);
            foreach (var edge in graph.Nodes.SelectMany(t => t.Neighbors).Distinct())
            {
                g.DrawString(graph.Edges.IndexOf(edge).ToString(), font,
brushColors[Color.Black], new PointF(((edge.Nodes[0].X + edge.Nodes[1].X) / 2),
((edge.Nodes[0].Y + edge.Nodes[1].Y) / 2)));
                g.DrawLine(pen, edge.Nodes[0].X, edge.Nodes[0].Y, edge.Nodes[1].X,
edge.Nodes[1].Y);
            }

            foreach (var node in graph.Nodes)
            {
                g.FillEllipse(brushColors[node.Color], node.X - 10, node.Y - 10, 20, 20);
//TODO: Make the width and height scale based on image size and numnodes
            }

            PictureBox_Graph.Image = image;
        }

        private void Btn_DrawGraphAtIndex_Click(object sender, EventArgs e)
        {
            try
            {
                DrawGraph(Graphs[int.Parse(TxtBx_DrawNodeAtIndex.Text)]);
                currentDrawIndex = int.Parse(TxtBx_DrawNodeAtIndex.Text);
                TxtBx_DrawNodeAtIndex.Text = currentDrawIndex.ToString();
            }
            catch(Exception ex)
            {
                //Just don't do anything
            }
        }

        int currentDrawIndex = 0;
        private void Btn_DrawPrevious_Click(object sender, EventArgs e)
        {
            if (currentDrawIndex > 0)
            {
                DrawGraph(Graphs[--currentDrawIndex]);
                TxtBx_DrawNodeAtIndex.Text = currentDrawIndex.ToString();
            }
        }

        private void Btn_DrawNext_Click(object sender, EventArgs e)
        {
            if (currentDrawIndex <= Graphs.Count)
            {
                DrawGraph(Graphs[++currentDrawIndex]);
                TxtBx_DrawNodeAtIndex.Text = currentDrawIndex.ToString();
            }
        }
    }
}
```

FormResults.cs

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

namespace Common
{
    public partial class FormResults : Form
    {
        public FormResults()
        {
            InitializeComponent();
        }

        public bool InitializeChart(List<int> iterations, List<double> convergences, 
List<double> averageFitnesses, List<double> minimumFitness, List<double> maximumFitness, 
List<Tuple<int, List<double>>> selectedFitnesses, int logInterval)
        {
            if (iterations.Count < 2)
                return false;

            Chart_Results.Series[0].LegendText = "Convergence";
            Chart_Results.Series[0].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
            Chart_Results.ChartAreas[0].AxisX.Minimum = 0;
            Chart_Results.ChartAreas[0].AxisX.Maximum = iterations.Last() + logInterval - 
iterations.Last() % logInterval;
            Chart_Results.ChartAreas[0].AxisX.Interval = logInterval;
            Chart_Results.ChartAreas[0].AxisY.Minimum = 0;
            Chart_Results.ChartAreas[0].AxisY.Maximum = 1;
            Chart_Results.Series[0].Points.DataBindXY(iterations, convergences.Select(t => 
t).ToList());

            //Chart_Results.Series.Add("Average Fitness");
            //Chart_Results.Series[1].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
            //Chart_Results.Series[1].Points.DataBindXY(iterations, averageFitnesses);

            Chart_FitnessScores.Series[0].LegendText = "Selected Chromosomes Fitness";
            Chart_FitnessScores.Series[0]["IsXAxisQuantitative"] = "true";
            Chart_FitnessScores.Series[0].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Point;
            Chart_FitnessScores.ChartAreas[0].AxisX.Minimum = 0;
            Chart_FitnessScores.ChartAreas[0].AxisX.Maximum = iterations.Last() + logInterval 
- iterations.Last() % logInterval;
            Chart_FitnessScores.ChartAreas[0].AxisX.Interval = logInterval;

            int parentCount = selectedFitnesses[0].Item2.Count;
            List<int> iters = new List<int>();
            List<double> points = new List<double>();
            foreach (var yData in selectedFitnesses)
            {
                foreach (var yDataPoints in yData.Item2)
                {
                    iters.Add(yData.Item1);
                }
```

```
                    points.AddRange(yData.Item2);
                }
                Chart_FitnessScores.Series[0].Points.DataBindXY(iters, points);

                Chart_FitnessRange.Series[0].LegendText = "Average fitness";
                Chart_FitnessRange.Series[0].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRange.ChartAreas[0].AxisX.Minimum = 0;
                Chart_FitnessRange.ChartAreas[0].AxisX.Maximum = iterations.Last() + logInterval - 
iterations.Last() % logInterval;
                Chart_FitnessRange.ChartAreas[0].AxisX.Interval = logInterval;
                Chart_FitnessRange.ChartAreas[0].AxisY.Minimum = 0;
                //Chart_FitnessRange.ChartAreas[0].AxisY.Maximum = 
Math.Ceiling(averageFitnesses.Average()); //average of averages
                Chart_FitnessRange.Series[0].Points.DataBindXY(iterations, averageFitnesses);

                Chart_FitnessRange.Series.Add(new 
System.Windows.Forms.DataVisualization.Charting.Series());
                Chart_FitnessRange.Series[1].LegendText = "Maximum fitness";
                Chart_FitnessRange.Series[1].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRange.Series[1].Points.DataBindXY(iterations, maximumFitness);

                Chart_FitnessRange.Series.Add(new 
System.Windows.Forms.DataVisualization.Charting.Series());
                Chart_FitnessRange.Series[2].LegendText = "Minimum fitness";
                Chart_FitnessRange.Series[2].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRange.Series[2].Points.DataBindXY(iterations, minimumFitness);


                Chart_FitnessRangeFocused.Series[0].LegendText = "Average fitness";
                Chart_FitnessRangeFocused.Series[0].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRangeFocused.ChartAreas[0].AxisX.Minimum = 0;
                Chart_FitnessRangeFocused.ChartAreas[0].AxisX.Maximum = iterations.Last() + 
logInterval - iterations.Last() % logInterval;
                Chart_FitnessRangeFocused.ChartAreas[0].AxisX.Interval = logInterval;
                if (minimumFitness.Min() != minimumFitness.Max())
                {
                    Chart_FitnessRangeFocused.ChartAreas[0].AxisY.Minimum = minimumFitness.Min();
                    Chart_FitnessRangeFocused.ChartAreas[0].AxisY.Maximum = minimumFitness.Max();
                }
                Chart_FitnessRangeFocused.Series[0].Points.DataBindXY(iterations, 
averageFitnesses);

                Chart_FitnessRangeFocused.Series.Add(new 
System.Windows.Forms.DataVisualization.Charting.Series());
                Chart_FitnessRangeFocused.Series[1].LegendText = "Maximum fitness";
                Chart_FitnessRangeFocused.Series[1].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRangeFocused.Series[1].Points.DataBindXY(iterations, maximumFitness);

                Chart_FitnessRangeFocused.Series.Add(new 
System.Windows.Forms.DataVisualization.Charting.Series());
                Chart_FitnessRangeFocused.Series[2].LegendText = "Minimum fitness";
                Chart_FitnessRangeFocused.Series[2].ChartType = 
System.Windows.Forms.DataVisualization.Charting.SeriesChartType.Line;
                Chart_FitnessRangeFocused.Series[2].Points.DataBindXY(iterations, minimumFitness);

                return true;
            }
        }
}
```