**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 at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2020

© Copyright Ryan Darras 2020

All Rights Reserved

This thesis for the Master of Science degree by

Ryan Darras

has been approved for the

|  |
| --- |
| Date |

Department of Computer Science

by

|  |
| --- |
| Sudhanshu K. Semwal, Chair |

|  |
| --- |
| Albert Chamillard |

|  |
| --- |
| Yanyan Zhuang |

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

Exploring the Unique Games conjecture

Thesis directed by Professor Sudhanshu K. Semwal

Constraint satisfaction problems, even if one does not know it, are problems that we interact with almost every day. Many researchers are studying constraint satisfaction problems and focusing on optimizing them because they are relevant and can directly impact and improve daily situations.

Everybody loves Amazon, the modern age option to shop in the comfort of home and have the items shipped directly to your doorstep. 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 Analysis/NP-Completeness yet dwarfs in comparison to what Amazon and other distributors must deal with. 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. This is simply because of the massive complexity of their problem, which is why research on constraint satisfaction problems, which are NP-hard problems, is so important.

Due to the potential NP-Hardness of many constraint satisfaction problems and their relationships that allow research to share across problems, they have been 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 algorithms. Branch and bound algorithms are a very common tool when solving NP-Hard problems due to their nature to become an exhaustive search which will provide every possible answer and look at it before reporting the best answer. However, exhaustive branch and bound algorithms are incredibly inefficient, yet 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. These heuristics are generally obtained by studying the given problem and using insight, logic, and reason to determine the most optimal values and weights.

Due to the possible increase in efficiency, creating a heuristic that accurately measures each branch is an incredibly important part of optimizing these problems. Determining which branch is not worth searching heavily reduces the algorithms runtime. In this research, we propose a genetic algorithm that will determine the optimal heuristic values for any given CSP.

Table of Contents

[CHAPTER 1 1](#_Toc49185802)

[INTRODUCTION 1](#_Toc49185803)

[Constraint Satisfaction Problems 1](#_Toc49185804)

[Khot’s Unique Games Conjecture 1](#_Toc49185805)

[Purpose of this Research 2](#_Toc49185806)

[The Focus Problems 3](#_Toc49185807)

[0-1 Knapsack Problem 3](#_Toc49185808)

[Graph Coloring Problem 4](#_Toc49185809)

[The Random Game 5](#_Toc49185810)

[CHAPTER 2 7](#_Toc49185811)

[REVIEW OF THE LITERATURE 7](#_Toc49185812)

[Heuristics 7](#_Toc49185813)

[Genetic Algorithms 8](#_Toc49185814)

[General Constraint Satisfaction Problems 8](#_Toc49185815)

[0-1 Knapsack Problem 8](#_Toc49185816)

[Graph Coloring Problem 8](#_Toc49185817)

[SIGNIFICANCE OF THIS RESEARCH 8](#_Toc49185818)

[CHAPTER 3 9](#_Toc49185819)

[PROJECT IMPLEMENTATION 9](#_Toc49185820)

[Framework 9](#_Toc49185821)

[Genetic Algorithm 10](#_Toc49185822)

[Chromosome 12](#_Toc49185823)

[Population 12](#_Toc49185824)

[Focus Problems 13](#_Toc49185825)

[0-1 Knapsack Problem 13](#_Toc49185826)

[Graph Coloring Problem 15](#_Toc49185827)

[The Random Game 20](#_Toc49185828)

[CHAPTER 4 23](#_Toc49185829)

[DATA & RESULTS 23](#_Toc49185830)

[0-1 Knapsack Problem 23](#_Toc49185831)

[Graph Coloring Problem 31](#_Toc49185832)

[The Random Game 31](#_Toc49185833)

[CONCLUSION 37](#_Toc49185834)

[BIBLIOGRAPHY/REFERENCES (TODO) 37](#_Toc49185835)

# CHAPTER 1

## INTRODUCTION

### Constraint Satisfaction Problems

Constraint satisfaction problems 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 constraint satisfaction problem a state V must be found that satisfies every e ε E. Constraint satisfaction problems often require a combination of heuristics and search algorithms to solve due to their exponential complexity of NP-Completeness[[1]](#footnote-1). Branch-and-bound algorithms are commonly used for solving NP-Complete problems due to its nature of searching a state space. A branch-and-bound algorithm uses 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. Hence, in our work, we will be implementing a genetic algorithm that will search for optimal heuristic values to speed up our branch-and-bound algorithms.

### Khot’s Unique Games Conjecture

TODO: Describe in my own words what Khot’s conjecture is, and why I think it is important.

### Purpose of this Research

Constraint satisfaction problems (CSPs) are hurdles 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 simply optimizing the way we attempt their solution.

In the past, software engineers and software developers relied on Moore’s Law to increase the range of possibilities for modern computers. However, Moore’s Law is coming to an end[[2]](#footnote-2), and computers 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 conjecture, we hope to better understand why solving these problems is incredibly difficult.

By providing the research that demonstrates the values and benefits of having a genetic algorithm find the optimal method to solve constraint satisfaction problems, we open a pathway for many other NP-Hard[[3]](#footnote-3) problems to be studied to provide more optimal solutions. We may also discover more insight into Khot’s Conjecture and find a more efficient way to determine the value of a given problem.

### The Focus Problems

There are an insurmountable amount of constraint satisfaction problems and even more sub-problems that have derived from them. Due to this, it is impossible to apply this research to every single constraint satisfaction problem 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 . Shows an example of a 0-1 knapsack problem, including v/w which is used in the approximation algorithm called the greedy knapsack algorithm.

#### 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 one of the most dynamic constraint satisfaction problems due to the additional constraints that can be added into the problem.

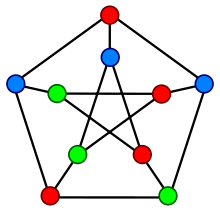
The graph coloring problem can be separated into three different problems: vertex coloring where no two adjacent vertices share a color, edge coloring where no two adjacent edges share a color, and face coloring where no face on a planar graph shares a boundary with another face of the same color. Not 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 . A solution for vertex coloring of the Petersen graph.

#### 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 amount 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. So instead of randomly applying colors to the vertices in V, we instead 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, thus we can estimate how long it would take to solve The Random Game if we were solving it completely randomly.

See below an example with the following constraints:

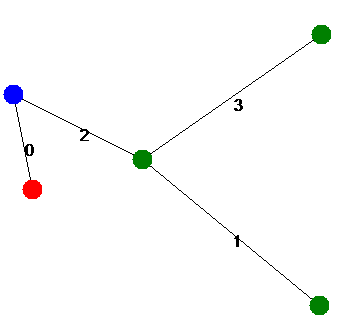
* Edge 0 must be different colors.
* Edge 1 must be either red/blue, green/green, or red/green.
* Edge 2 must either be green/red or green/blue.
* Edge 3 must be the same color.

Figure 7. An example of a state of V that is satisfied given the existing constraints

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 2

## REVIEW OF THE LITERATURE

The following sections are simple surveys of the topics studied in this thesis. These surveys will consider the most state-of-the-art to ensure our work is based on the top of the line research.

### Heuristics

The power of a heuristic does not come from the solution, but rather the method of obtaining the solution. Constraint Satisfaction Problems are faced with the problem of not being able to find the most optimal solution in polynomial time, but by using a heuristic you can approximate the solution in polynomial time. Taking a step further, Burke et al. 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”[[4]](#footnote-4) .

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

### Genetic Algorithms

### General Constraint Satisfaction Problems

### 0-1 Knapsack Problem

### Graph Coloring Problem

## SIGNIFICANCE OF THIS RESEARCH

The results of this research may be taken and used to accelerate algorithm development when using heuristics. It will help show relationships and correlations between heuristics and evaluation metrics that should help researchers narrow down to more effective, accurate results.

# CHAPTER 3

## PROJECT IMPLEMENTATION

The objective is to make the project as generic and adaptable as possible, 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 our outputs vs our inputs.

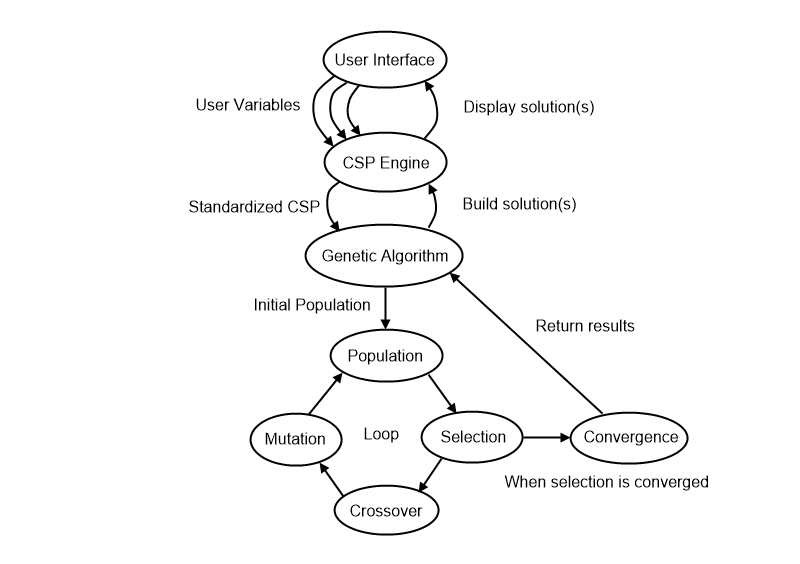


Figure . Demonstration of framework architecture used to test our thesis.

### 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 combined with the parents from the previous generation form the new generation.
6. Mutation: At this point we have a new, complete generation and 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.
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 percolating 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));

}

}

### Focus Problems

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.

#### 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 have to 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.

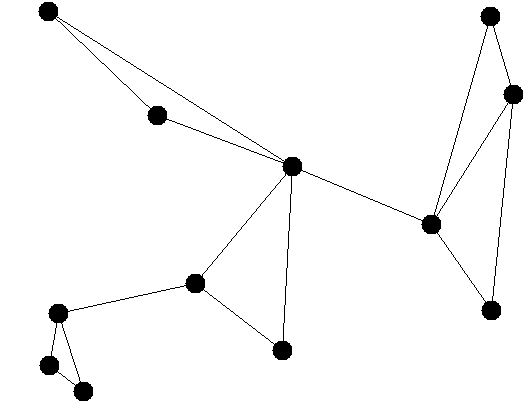
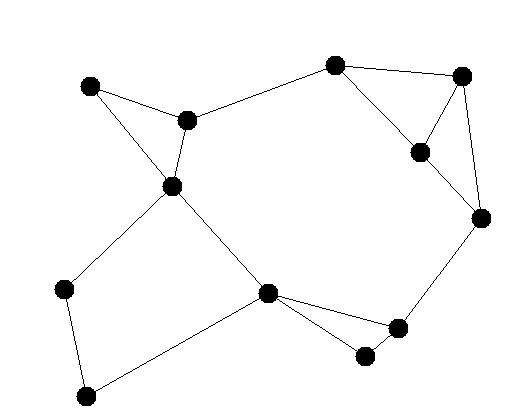
We will be using a branch-and-cut algorithm to solve this problem, and we will be specifically cutting branches due to the four-color theorem. In 1890, British mathematician Percy John Heawood proved that any political map could be colored using at a maximum five different colors. His work was based 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:

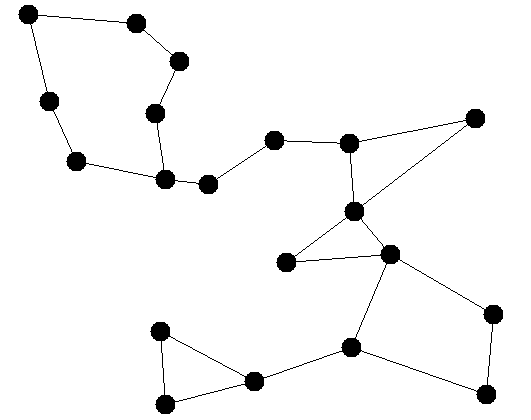
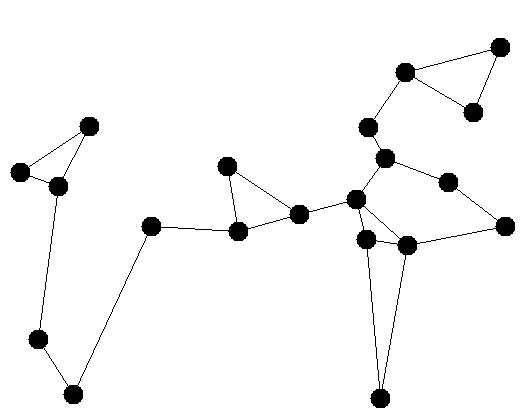
* Generate n nodes at random locations on the plane.
* Continuously iterate over the nodes, moving them away from any nodes they that are too close to. This prevents overlapping nodes.
* For every node, we determined how many neighbors it wanted based on a random number between the minEdgesPerNode and maxEdgesPerNode and stored these “desired neighbors” in a collection to use later.
* Using the desired neighbors collection, we made 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 if completing these edges resulted in any node going over maxEdgesPerNode neighbors and denied 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:

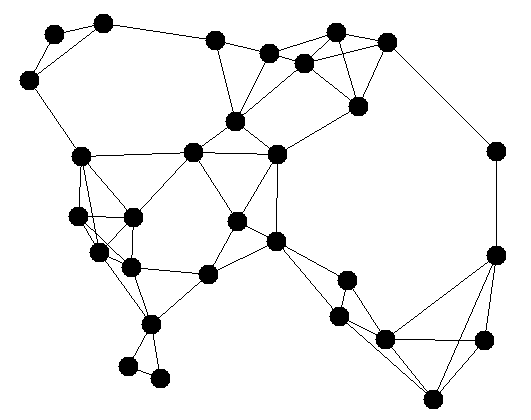
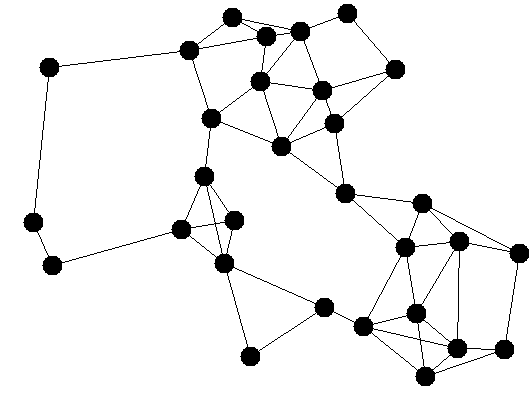
* Twelve nodes, 25% edge density



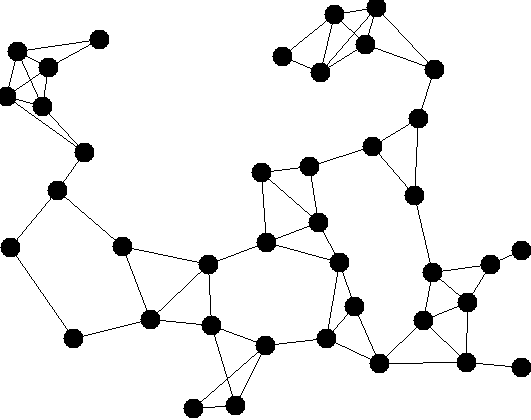
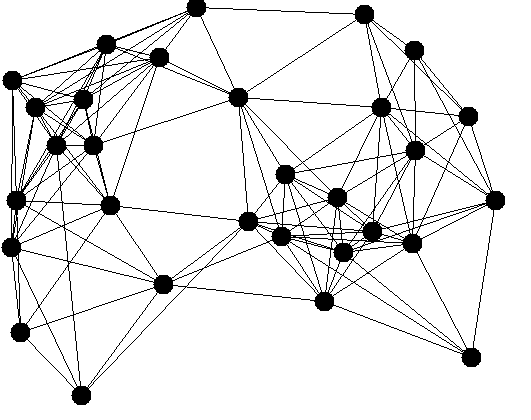
* Twenty Nodes, 10% edge density



* Thirty Nodes, 25% edge density



* Thirty Nodes, 100% edge density; 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 by modern standards, 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 nodes 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 2n – 1 is the worst possible outcome. 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

We used the same algorithm as described in the previous section 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;

}

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;

}

}

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));

}

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 4

## DATA & RESULTS

### Knapsack Problem

As expected, the heuristics to solve 0-1 Knapsack Problem were very simple, and the results were like what we expected; being that value is attractive, where 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 was 1.889 with a value of 19.318 and a weight of 10.229. Our min R value was .528 with a value of 10.042 and a weight of 19.025.

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

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.

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. This put us in a position where we were keeping the good genes but testing the waters with up to a 25% mutation 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 |

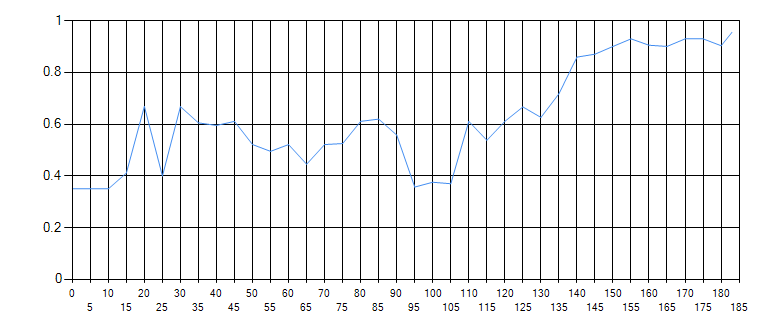


Figure shows the convergence over 185 generations until the convergence threshold of being within 5% was reached.

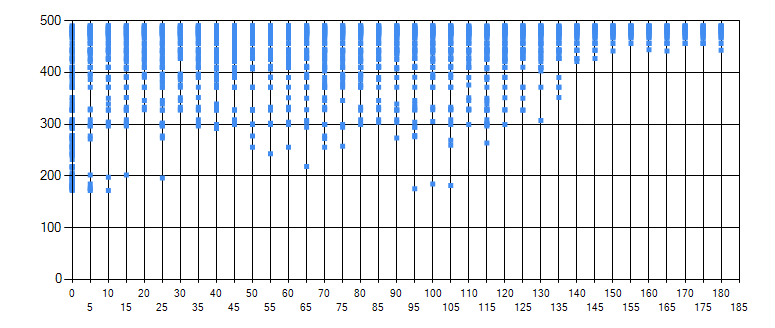


Figure shows the fitness of each parent chromosome selected in each generation.

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

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. So we tried the following:

These results blew us away! By offering every single gene the ability to mutated 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 |

Judging by the common theme of the minimum weight being the most heavily prioritized heuristic variable here we theorized that it was due to our 300 randomly generated knapsack objects were generally more heavy 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 more heavy side, and lower value side was correct.

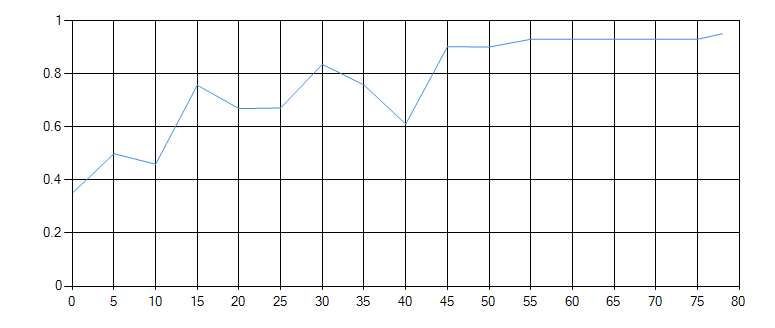


Figure shows that the liberal approach to genetic mutation converged in only 80 generations.

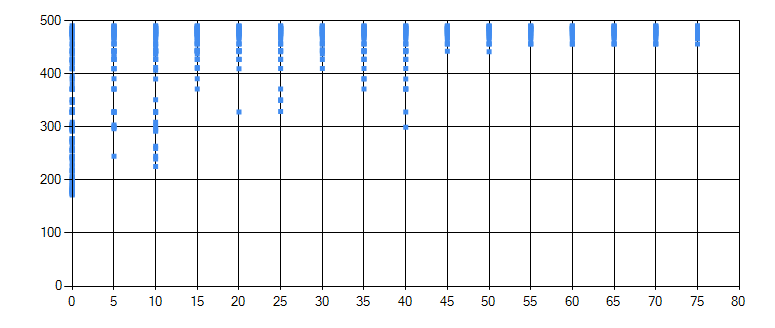


Figure shows the fitness of each parent chromosome selected in each generation.

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% |

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 |

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

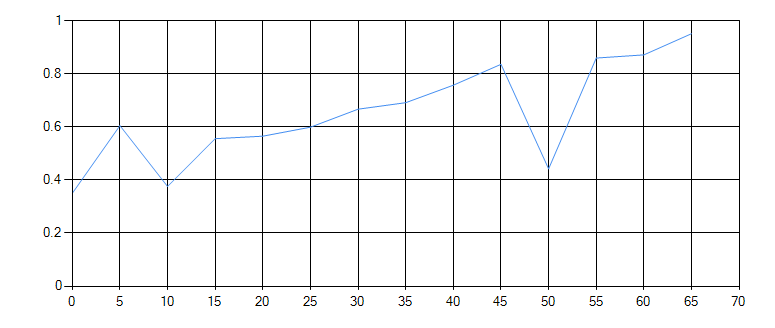


Figure shows a convergence in 65 generations using the conservative approach.

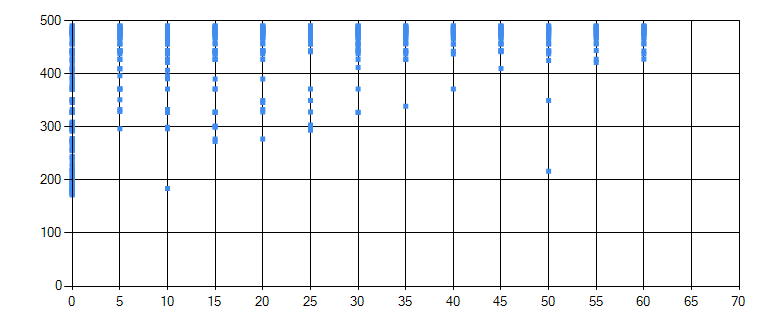


Figure shows the fitness of each parent chromosome in each generation.

The results generated from the conservative approach has a very aggressive dip in convergence at generation 50. We theorize 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(Lib) |
| 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 |

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 a large amount of 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.

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

Following are the iterations of The Random Game we ran to help us understand randomness. 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.

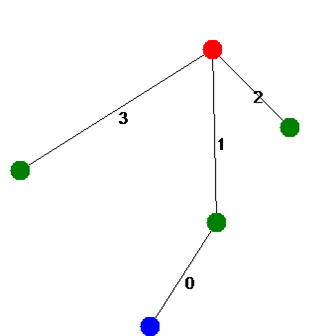
Seed 17 – Five nodes, four edges

Figure 7. A solution for seed 17 that satisfies all the constraints

* 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

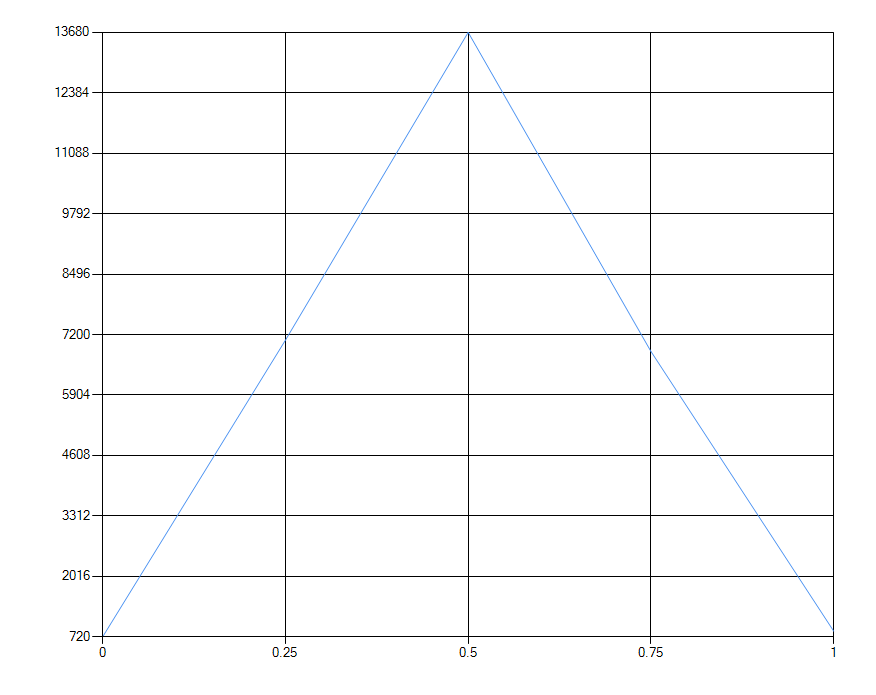
Satisfied Percentage

Figure 8. Shows the number of graphs in the complete problem space that satisfies the given percentage of constraints: 0% - 720, 25% - 7080, 50% - 13680, 75% - 6840, 100% - 840

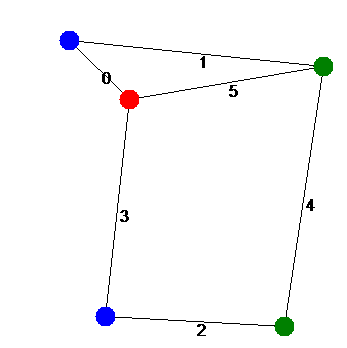
Seed 1337 – Five nodes, six edges

Figure 9. A solution for seed 1337 that satisfies all the constraints

* 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

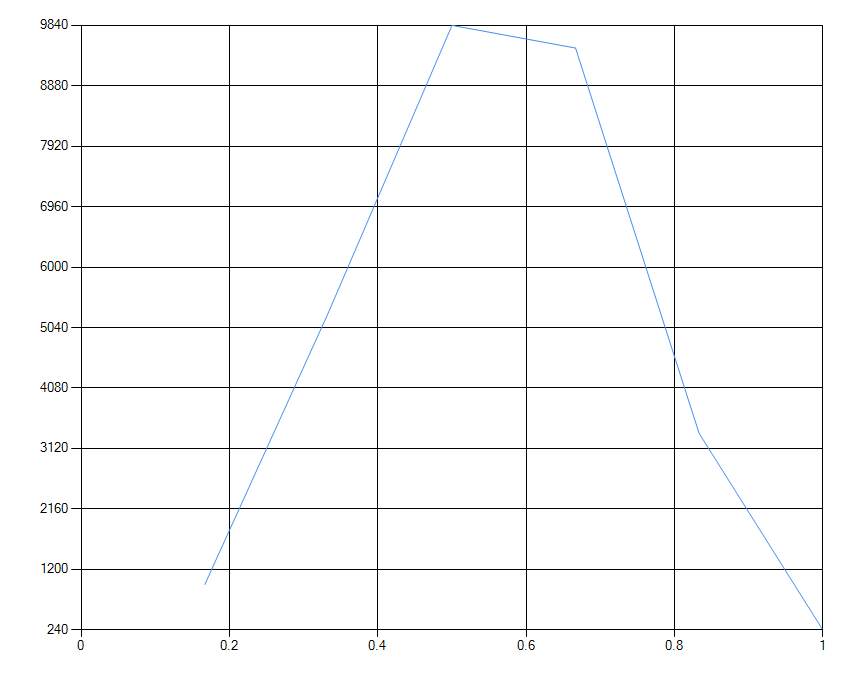
Satisfied Percentage

Figure 10. Shows the number of graphs in the complete problem space that satisfies the given percentage of constraints: 16% - 960, 33% - 5280, 50% - 9840, 66% -9480, 83% - 3360, 100% - 240

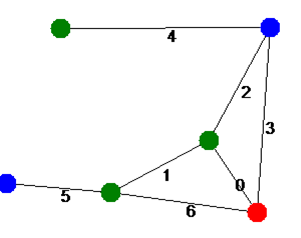
Seed 2000 – Six nodes, seven edges

Figure 11. A solution for seed 2000 that satisfies all the constraints

* 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

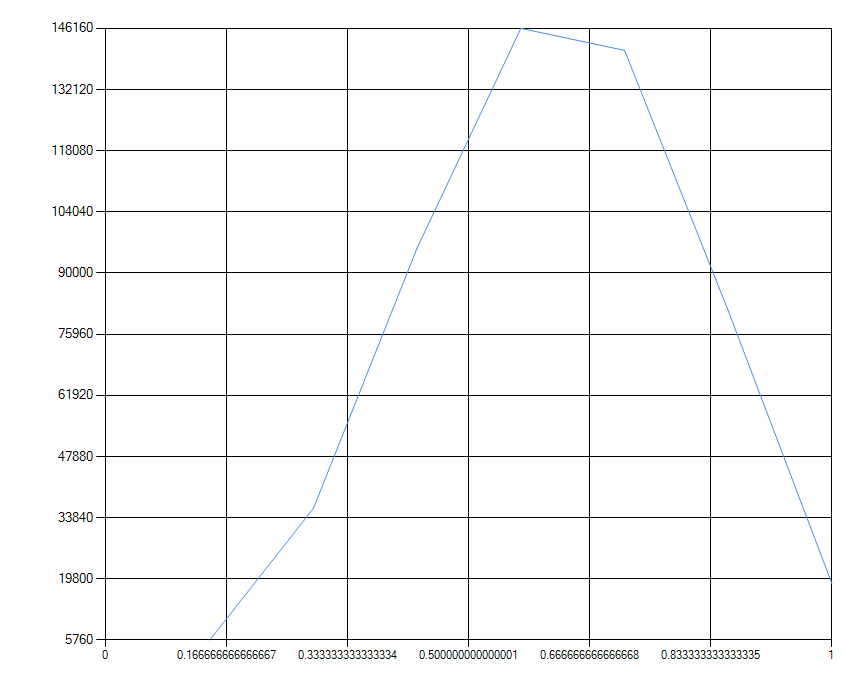
 Satisfied Percentage

Figure 12. Shows the number of graphs in the complete problem space that satisfies the given percentage of constraints: 14% - 5760, 28% - 36000, 42% - 95760, 57% - 146160, 71% - 141120, 85% - 81360, 100% - 18720

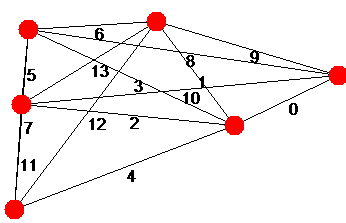
Seed 2500 – Six nodes, fourteen edges

Figure 13. The graph for seed 2500. There was no possible solution that satisfied every constraint

* 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

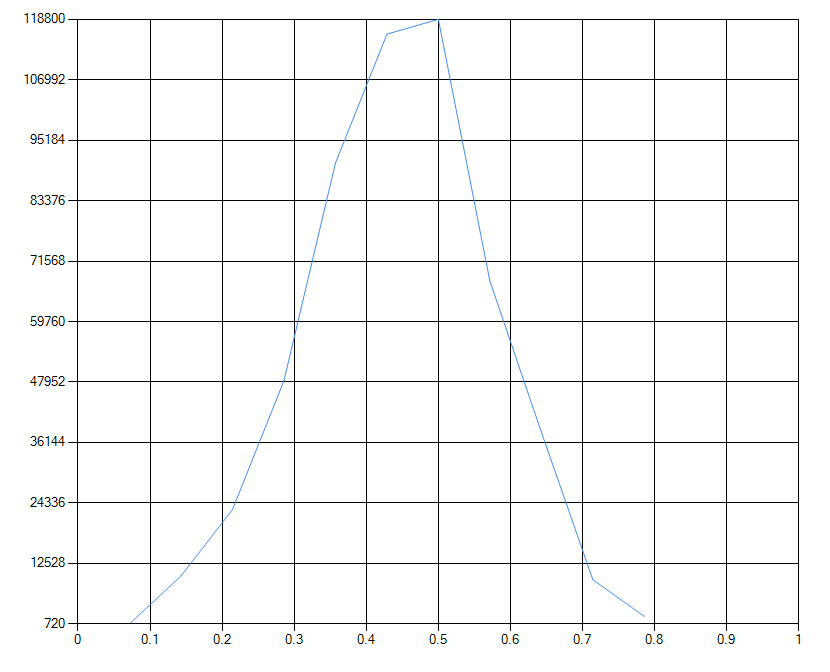
 Satisfied Percentage

Figure 12. Shows the number of graphs in the complete problem space that satisfies the given percentage of constraints: 7% - 720, 14% - 10080, 21% - 23040, 28% - 48240, 35% - 90720, 42% - 115920, 50% - 118800, 57% - 67680, 64% - 38160, 71% - 9360, 78% - 2160

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

# BIBLIOGRAPHY/REFERENCES (TODO)

<https://www.researchgate.net/publication/220059985_David_L_Applegate_Robert_E_Bixby_Vasek_Chvatal_William_J_Cook_The_Traveling_Salesman_Problem_A_Computational_Study_Princeton_University_Press_Princeton_2007_ISBN-13_978-0-691-12993-8_606_pp>

1. An answer to NP-Complete problems can be verified in polynomial time (quickly) but an optimized answer can only be solved/provided in exponential time (very slowly). Because of this, NP-Complete problems are often solved using approximation algorithms. [↑](#footnote-ref-1)
2. Gordon Moore [↑](#footnote-ref-2)
3. The set of NP-Hard problems has solutions that can be used to derive the solutions to problems in NP in polynomial time. [↑](#footnote-ref-3)
4. Burke et al. [↑](#footnote-ref-4)