**A GENETIC ALGORITHM TO OPTIMIZE HEURISTIC VARIABLES FOR CONSTRAINT SATISFACTION PROBLEMS**

by

RYAN DARRAS

B.I., University of Colorado, 2017

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

2019

© Copyright Ryan Darras 2019

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)

A Genetic Algorithm to Optimize Heuristic Variables

for Constraint Satisfaction Problems

Thesis directed by Professor Sudhanshu K. Semwal

Constraint satisfaction problems, even if you don’t know it, are problems that you interact with almost every day. Many researchers are studying constraint satisfaction problems and focusing on optimizing them because of how relevant they are in today’s society, and how much of the world’s day-to-day can be improved by these optimizations.

Everybody loves Amazon, the modern age option to shop in the comfort of your own home and have the items shipped directly to your door step. However, most everybody does not understand the complexity behind managing such a massive operation. The traveling salesman problem is a common lesson for undergraduate students to teach them the topic of NP-hard problems, but it is a small problem compared to what Amazon and other distributors have to 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. They also need to consider shipping between warehouses that are on the opposite side of the country from one another. Without researchers designing effective algorithms to increase the efficiency of this massive problem, we wouldn’t 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, or any NP-hard problem, 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 times, new NP-Hard problems can be derived from existing 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. However, exhaustive branch and bound algorithms are incredibly inefficient but 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 man made 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](#_Toc9498255)

[INTRODUCTION 1](#_Toc9498256)

[Purpose of this Research 1](#_Toc9498257)

[The Focus Problems 2](#_Toc9498258)

[0-1 Knapsack Problem 2](#_Toc9498259)

[Traveling Salesman Problem 4](#_Toc9498260)

[Job Shop Scheduling Problem 5](#_Toc9498261)

[N-Queens Problem 7](#_Toc9498262)

[Graph Coloring Problem 9](#_Toc9498263)

[EVALUATION METRICS 10](#_Toc9498264)

[Time 10](#_Toc9498265)

[Space 10](#_Toc9498266)

[Accuracy 11](#_Toc9498267)

[GOALS (TODO: Refine goals) 11](#_Toc9498268)

[CHAPTER 2 (TODO: simple survey of each topic) 12](#_Toc9498269)

[REVIEW OF THE LITERATURE 12](#_Toc9498270)

[Heuristics 12](#_Toc9498271)

[Genetic Algorithms 12](#_Toc9498272)

[General Constraint Satisfaction Problems 12](#_Toc9498273)

[0-1 Knapsack Problem 12](#_Toc9498274)

[Traveling Salesman Problem 12](#_Toc9498275)

[Job Shop Scheduling Problem 12](#_Toc9498276)

[N-Queens Problem 12](#_Toc9498277)

[Graph Coloring Problem 12](#_Toc9498278)

[SIGNIFICANCE OF THIS RESEARCH 12](#_Toc9498279)

[CHAPTER 3 (TODO: Finish all problems) 13](#_Toc9498280)

[PROJECT IMPLEMENTATION 13](#_Toc9498281)

[Framework 13](#_Toc9498282)

[Genetic Algorithm 14](#_Toc9498283)

[Chromosome 16](#_Toc9498284)

[Population 16](#_Toc9498285)

[Focus Problems 17](#_Toc9498286)

[0-1 Knapsack Problem 17](#_Toc9498287)

[Traveling Salesman Problem 19](#_Toc9498288)

[Job Shop Scheduling Problem 19](#_Toc9498289)

[N-Queens Problem 19](#_Toc9498290)

[Graph Coloring Problem 19](#_Toc9498291)

[LONG TERM IMPROVEMENTS & ADAPTIONS (TODO: Probably dont need?) 19](#_Toc9498292)

[NuGet Package 19](#_Toc9498293)

[Generic Genetic Algorithm 19](#_Toc9498294)

[CHAPTER 4 (TODO: When chapter 3 is done) 19](#_Toc9498295)

[SYSTEM OVERVIEW 19](#_Toc9498296)

[DATA & RESULTS 19](#_Toc9498297)

[CONCLUSION 19](#_Toc9498298)

[BIBLIOGRAPHY/REFERENCES (TODO) 19](#_Toc9498299)

# CHAPTER 1

## INTRODUCTION

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 of 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 high complexity and NP-Hardness. Branch-and-bound algorithms are commonly used for solving NP-Hard problems due to its nature of being a state space search. 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, we will be implementing a genetic algorithm that will search for optimal heuristic values to speed up our branch-and-bound algorithms.

### Purpose of this Research

Constraint satisfaction problems 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[[1]](#footnote-1), 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 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 problems to be studied to provide more optimal solutions.

### 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 five 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

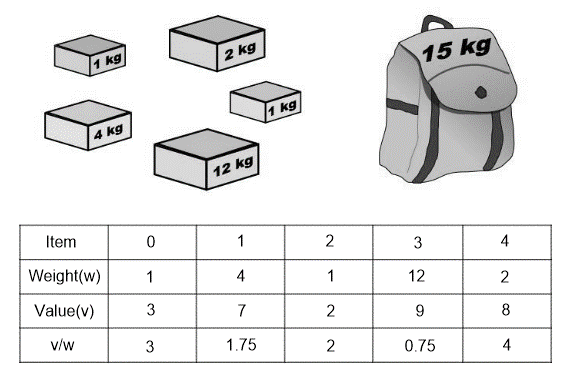
The 0-1 knapsack problem is defined by having a knapsack with a given weight limit W. Given a collection of items N each with a value v and weight w, the goal of this problem is to select items to put in the knapsack that results in the highest possible total value. While the 0-1 knapsack problem is 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.

Figure 1. Shows an example of a 0-1 knapsack problem, including v/w which is used in the approximation algorithm called the greedy knapsack algorithm.

The 0-1 knapsack problem is actually a pseudo-polynomial problem as opposed to just outright NP-Hard. This is because the algorithm is O(n\*W) where n is the number of objects available and W is the size of the knapsack. On the surface, this looks to be polynomial, or a problem that is a subset of P. However, due to the nature that W scales in both size and length, the 0-1 knapsack problem is actually defined as NP-Complete.

The reason we started here, is that as long as we manage the length of W we can ensure the problem be solvable in a fixed, small length period of time. This allows for testing of the genetic algorithm and becoming more familiar with the framework to make the following, more complicated problems easier to work with.

#### Traveling Salesman Problem

The traveling salesman problem has long been studied as an intro to algorithms and big-O notation. It is a problem that many people, even those not studying computer science, may have heard about. This problem involves a traveling salesman that needs to visit n cities to sell his merchandise. In order to optimize his time, and therefore his profits, he needs to plan out the route that is the shortest possible route between the cities. Traditional forms of the problems limit the salesman to a single visit to each city, whereas other forms allow the salesman to pass through a city multiple times if it yields a faster route.

This is a very rigorously studied problem as it fully demonstrates the property of NP-Hardness. Not only that, but this problem affects billions of people every single day. Without optimal path planning, Amazon’s costs go up, which in turn causes their merchandise to increase in price which affects all of their users.

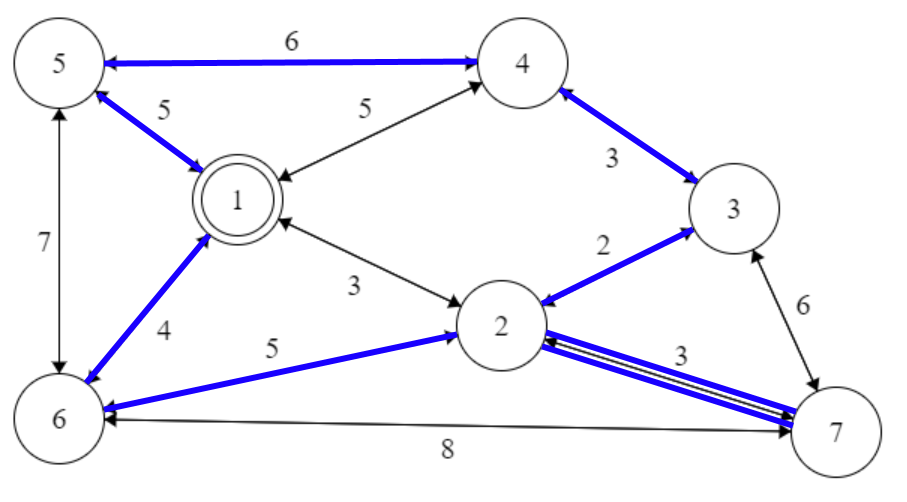
A brute force search for the optimal route would result in an algorithm that has a running time of O(n!) where n is the number of cities that need to be visited. This makes a brute force algorithm non-feasible even when the number of cities is only 15. Linear programming techniques work well for up to 200 cities, but the current method for solving large instances is an approach using a derivative of a branch-and-bound algorithm called a branch-and-cut algorithm. This solution holds the current record, solving an instance with 85,900 unique cities[[2]](#footnote-2)­.

Figure 2. An example of a solved traveling salesman problem starting at city 1, and ending back at city 1. This example demonstrations the version of the problem where the salesman can backtrack if doing so would result in a more optimal path.

#### Job Shop Scheduling Problem

The job shop scheduling problem is another very popular problem that is shared with students during their architecture and operating systems classes because it is an impressive optimization problem that severely impacts computers today.

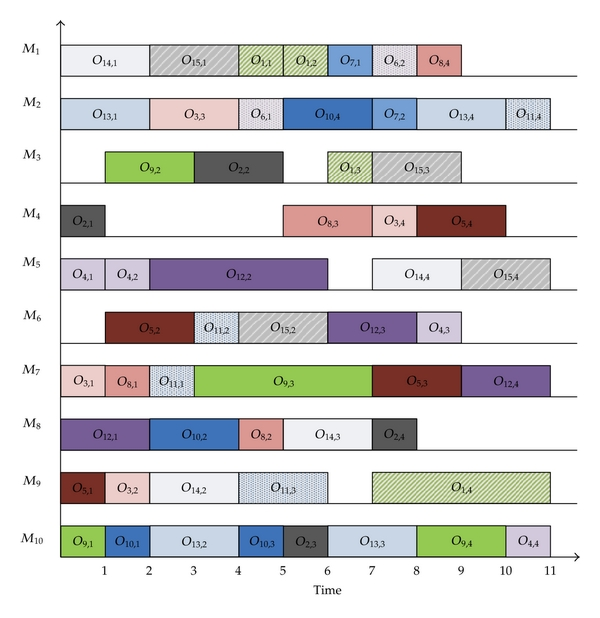
The problem focuses on scheduling computer tasks (jobs) in such a way that optimizes performance. What makes the job shop scheduling problem so tricky is that the method of optimizing performance is different for many different systems based on the objective of that machine. For example, if we have a uniprocessor system we need to determine if multiprogramming is important to us, or if we want to frontload all processing power to singular tasks.

Figure 3. This is an example of a job shop scheduling problem with each job is a specific color and each objective for each job must be completed in order (denoted by Oj,n where j is the job and n is the order at which it must complete. This example uses 10 machines to optimize the schedule.

The job shop scheduling problem actually encapsulates the traveling salesman problem. The traveling salesman problem is a special case of a job shop scheduling problem where the salesman is a machine, and the cities are the jobs. Knowing that the traveling salesman problem is NP-Hard, we can infer that the job shop scheduling problem is also NP-Hard.

The problem consists of n jobs J­1, J2, … , Jn each with a set of operations O1, O2, … , On that need to be processed in a specific order. Throughout the variations of this problem the many constraints include, but are not limited to: each operation having a specific machine that it needs to be processed on, is multiprocessing available, focus on minimizing the average response time, job dependencies, deterministic or probabilistic processing times, and minimizing the makespan.

#### N-Queens Problem

The n-queens problem is described as placing n queens on an nxn chessboard, where no queen can attack another queen. More specifically, no two queens share the same row, column, or diagonal. This problem was first established in 1848 by Max Bezzel who rigorously studied the strategies of chess[[3]](#footnote-3).

What makes the n-queens problem interesting is that while researchers are interested in finding a solution, they are also interested in finding the number of possible solutions for any value of n. Similarly to the traveling salesman problem, any value of n is relatively simple to solve up until n=15 where you run into the same situation due to the n-queens problem for solving the number of possible solutions being O(n!).

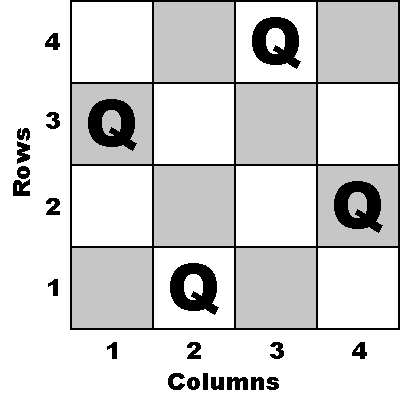


Figure 5. Example of an n=4 n-queens problem. Note that no queen shares a row, column, or diagonal with any other queen.

The n-queens problem has many related problems that make research towards it more valuable. These related problems include: considering higher dimensions, n-knights/bishops/kings/rooks, considering a mix of different chess pieces, and completing a partial n-queens solution.

#### Graph Coloring Problem

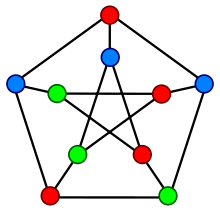
 The simplest form of the graph coloring problem focuses on coloring vertices of a graph in such a way that no two adjacent vertices share the same color. However, the graph coloring problem is one of the most dynamic constraint satisfaction problems due to the additional constraints that can be added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

Figure 6. A solution for vertex coloring of the Petersen graph.

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.

## EVALUATION METRICS

Due to the nature of these problems, it is important to evaluate multiple dimensions of complexity to fully understand the implications of these problems in large scale. Therefore, we have recorded and studied the three dimensions of time, space, and accuracy. By considering multiple dimensions, we can discover relationships and correlations that will further our research.

### Time

Time is a tricky evaluation metric due to factors such as computer specifications, project implementation, and outside interference. Therefore, we are logging both real-time as well as any data pertaining to the algorithm. Specifically: array accesses, array copies, swaps, failed branches, etc. By recording this additional information, an understanding can be made of the relationships between different heuristics for each problem.

### Space

Easier than time, space can be calculated by determining the total memory usage used to calculate the problem. However, evaluating space needs to consider maximum space required during the algorithm and average space required during the algorithm in order to more fully encompass the problem solution. To do this, data structure sizes and count of data structures in use will be calculated to provide a reasonable space evaluation.

### Accuracy

These problems indeed have optimal solutions, but to find the perfect answer is exponentially more complex and time consuming than using heuristics for educated estimates. Therefore, our answers might not be 100% correct 100% of the time. For many of these problems, estimates are more than enough to claim that the problem is satisfied, but at which point is an estimate not good enough? Specifically, at what point of accuracy is that estimate not valid? To record and evaluate accuracy, we will simply be comparing to the optimal solution for each problem. Our solution / optimal solution = accuracy.

## GOALS (TODO: Refine goals)

This research is aimed at achieving the following:

* Studying the individual focus problems, to gather an understanding of constraint satisfaction problems as a whole.
* Discover the relationships between heuristic variables and the results generated through their use.
* Discover the relationships between evaluation metrics time, space and accuracy in hopes to find more efficient solutions.
* Determine the effectiveness of using genetic algorithms to influence heuristic variables.

# CHAPTER 2 (TODO: simple survey of each topic)

## 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

### Genetic Algorithms

### General Constraint Satisfaction Problems

### 0-1 Knapsack Problem

### Traveling Salesman Problem

### Job Shop Scheduling Problem

### N-Queens 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 researches narrow down to more effective, accurate results.

# CHAPTER 3 (TODO: Finish all problems)

## 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.

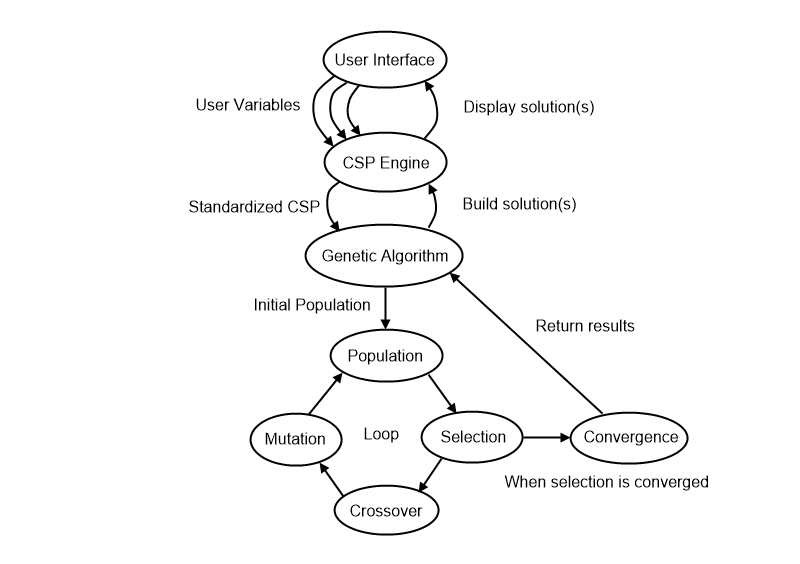


Figure 7. Demonstration of framework architecture used to test our thesis.

### 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. Selection: Keep the chromosomes with the highest fitness, and dispose of the remaining chromosomes.
4. Determine if the remaining chromosomes have converged on a small set of results. If so, return the result.
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.

#### Knapsack Problem

We formatted the chromosomes in the following form:

* [0] = priority of low weight (float)
* [1] = priority of high weight (float)
* [2] = priority of low value (float)
* [3] = priority of high value (float)

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 chromosomes 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.

#### Traveling Salesman Problem

#### Job Shop Scheduling Problem

#### N-Queens Problem

#### Graph Coloring Problem

## LONG TERM IMPROVEMENTS & ADAPTIONS (TODO: Probably dont need?)

### NuGet Package

### Generic Genetic Algorithm

# CHAPTER 4 (TODO: When chapter 3 is done)

## SYSTEM OVERVIEW

## DATA & RESULTS

## 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. Gordon Moore [↑](#footnote-ref-1)
2. Applegate et al. (2006) [↑](#footnote-ref-2)
3. W.W. Rouse Ball [↑](#footnote-ref-3)