# The Caravan Trader Problem (CTP) and A Tournament-Based Lamarckian Genetic Algorithm Solution

By Gierdino Julian Santoso

A thesis submitted in fulfillment
of the requirements for the degree of
Master of Computer and Information Sciences
Auckland University of Technology

School of Engineering, Computer and Mathematical Sciences
2019

**Declaration**

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a University or other institution of higher learning, except where due acknowledgement is made as referenced.

Signed,

Gierdino Julian Santoso

ACKNOWLEDGEMENTS

This thesis was completed in the School of Engineering, Computer and Mathematical Sciences at Auckland University of Technology, New Zealand. I have been blessed to be supported by so many important people in my life during the writing of this thesis. I would like to use this chance to thank all of you who have supported me throughout the course of my degree, because without all of you, none of this would have been accomplished.

First of all, I would like to thank my family for their support, especially my mother, for always staying strong and always there to support me. Even when faced with hardships she has never stopped to constantly give me prayers, guidance, and words of motivation. Her encouragement is the reason why I was able to pursue a higher level of education in the first place. I could never have a better mother than her. I would also give special thanks to my sister for always looking out for me and supporting my study and life in New Zealand. Without her help I would never be able to have an opportunity to live and study here, and I couldn't be a prouder brother to a kind and caring sister like her. Finally, I would like to thank my beloved wife for staying with me and supporting my decision for completing my degree. Without her constant emotional support, I would not be able to continue my education.

I would also like to thank Professor Ajit Narayanan for his support in guiding me throughout my thesis. He has given me a lot of supportive comments and suggestions, and it has been a pleasure to be supervised by him.

ABSTRACT

This research introduces the Caravan Trading Problem (CTP) which is a routing problem with an additional economic model, and attempts to optimize two different objectives, namely profit and distance, using Genetic Algorithm (GA). A hybrid GA solution using a combination of Greedy Algorithm and a tournament-based Lamarckism heuristic was developed, which was named Tournament-Based Lamarckian Genetic Algorithm (TBLGA). The result was that the proposed method was able to improve the evolutionary process by reducing the occurrence of local minima. We conclude that Lamarckian Evolution can be used to guide local search in multi objective optimization problems such as CTP.

Keywords: Genetic Algorithm, Multi Objective Genetic Algorithm, Lamarckism, Caravan Trading Problem, Metaheuristics

# TABLE OF CONTENTS

# List of Figures



# List of Tables



# List of Abbreviations

**CTP - Caravan Trading Problem**

**GA - Genetic Algorithm**

**LGA - Lamarckian Genetic Algorithm**

**MOGA - Multi-Objective Genetic Algorithm**

**PTP - Profitable Tour Problem**

**SOGA - Single-Objective Genetic Algorithm**

**TBLGA - Tournament-Based Lamarckian Genetic Algorithm**

**TSP - Travelling Salesman Problem**

**VRP - Vehicle Routing Problem**

**VRP-SPD - Vehicle Routing Problem with Simultaneous Pick-up and Deliveries**

# Chapter 1: Introduction

## 1.1 Background

People have been searching for efficient methods to perform their tasks since the beginning of human history. Ancient humans utilized stone tools in order to help them hunt and survive. In the middle of the 1700s during the Industrial Revolution, people used machineries to revolutionize the way they produce products. Today, we are living in the Information Age, and we are at the point where we have just recently begun to be able to process massive amounts of information in order to help us make decisions for a wide range of applications. Despite our progresses and advanced technologies, the search for more efficient methods to perform our tasks has not stopped. Our rapid progress in the fields of computing and informatics within the last decade has shown that we are still finding ways to optimize our methods. Currently, one of the most widely applied methods in optimization is the use of evolutionary algorithms such as GAs.

The desire to search for more efficient methods still continues. The research on algorithms and effective implementations of parameters and heuristics is still an active area of research in the field of computer science, and as humanity continues to observe nature and society, we continue to draw new inspiration and create new methods for solving problems. Optimization algorithms are still a growing area of study due to their applicability in a wide range of industries, and with every new method that has been created, the collective human knowledge expands, and so will our collective ability to solve past, current, and future problems.

With the advancement of automation, the search for optimization algorithms will even have more tangible benefits towards the advancement of humanity. Taxis, package deliveries, and even warehouse management, are examples of transportation sectors where automation is already being used.

In the process of optimization, there can be multiple objectives that are required to be optimized and might conflict with one another. While TSP remains a classic optimization problem where distance is required to be minimized, CTP adds a layer of complexity by presenting an additional objective, and can be used to model real world case such as transportation sectors where optimization requires both distance and profit. This paper will go into further detail on exploring CTP and the methods on solving the problem.

## 1.2 Objective

The goal of this research is to introduce the CTP and attempt to provide optimized solutions using GA.

The CTP is inspired by historical caravan traders. According to *Encyclopædia Britannica,* a caravan can be defined as a group of traders. (Chisholm, 1911). Caravans travelled from one place to another, visiting many different cities along the way, buying and selling products in order to gain profit before returning to their homeland. The idea of CTP is to maximize the amount of profit by selecting the path between cities, and selecting which products to buy and sell.

CTP can be considered to be an extension of TSP. Because CTP is derived from TSP, it also shares several of its characteristics, such as the rule that a route can visit a city only once, and also the rule that each route must end at the starting city. This means that the solution to CTP is also in the form of a Hamiltonian cycle. The main difference between TSP and CTP is that CTP includes another layer of subproblem with extra parameters to optimize. In CTP, there is a simple economic model in which each city has different prices for each product, and the caravan starts with a limited amount of initial funds and capacity to carry products. In order to produce optimized solutions, the caravan should only buy products from the city currently visited when it has a lower price than the price of the same products in another city. Therefore, the general problem that requires to be solved within the context of CTP can be described as follows:

> **Given a limited amount of capacity and initial funds, which route should the caravan traverse and which products should be bought and sold at each city in order to maximize the amount of profit?**

An alternative explanation of the CTP can be worded as follows:

> **How can a caravan which has limited amount of capacity and initial funds for trading accrue maximum profit in a single journey by trading in different locations with different pricing for different products without visiting the same location twice?**

## 1.3 Research Question

The research question is as follows:

> **How can GA optimize the maximum profit and minimum distance for the CTP?**

## 1.4 Applicability and Novelty

This research can be used to give insight on solving optimization problems when there are multiple, possibly conflicting objectives, and highlights potential challenges during implementations of such objectives. This research is also useful for exploring the implementation of Lamarckism within the context of GA and evolutionary algorithms. Moreover, this research can be used to explore the complexity and potential issues of optimizing a routing problem in a context where there are additional parameters to be considered, because in CTP there is an additional complexity of a simple economic model and the caravan's capacity limits to be considered. The main difference between CTP and other routing optimization problems is that CTP takes account of the condition or state of the traveler, which has a cause and effect relationship with the route itself. This could potentially be explored in other contexts and applications in the future.

The novelty of this research is that it attempts to provide an alternative method to the conventional GA by introducing hybrid methods, using Greedy Algorithm and Lamarckism to guide the local search. Moreover, the introduction of CTP, a routing problem with economic model, can be used as an example of an optimization problem with multiple objectives to be optimized. Furthermore, the TBLGA is a newly explored metaheuristic variant in Lamarckism, which can be further explored in the future.

**Real world context: Autonomous Taxi**

A possible applicability of the CTP is the implementation of an autonomous taxi system. An autonomous taxi has to work out a route for taking paying customers from one part of a city to another.

## 1.5 Methodology

The framework of this research is based on the constructive method design science approach and will produce a software artifact. According to Ken et al. (2007), design science research in information systems follows set guidelines in order to create successful artifacts:

**Design as an Artifact**

Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation. One of the deliverables of this research is a working source code, and it will be used to support and provide evidence to the hypothesis and theoretical frameworks discussed in this research. Furthermore, the existence of a source code makes it easy for future research to replicate the results.

**Problem Relevance**

The objective of design-science research is to develop technology-based solutions to important and relevant business problems. The main goal of introducing the CTP is to create a mathematical problem that has the

combined elements of a routing problem and an economic problem, which then will be systematically solved and optimized. This research attempts to solve both elements of problems simultaneously, because in reality, both problems are intertwined.

**Design Evaluation**

The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods. Several key metrics are designed in order to test the rigor of the design artifact. The metrics serve as an evaluation tool as well as a general guideline in order to proceed in the research.

**Research Contributions**

Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies. This research contributes to the area of design foundation by introducing the novelty of CTP, as well as the approach of using Lamarckism for a multi objective class of problem.

**Research Rigor**

Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact. The software development framework will follow iterative steps similar to Scrum that will focus on the implementation of a single objective per iteration. Each iterative step will be tested with several key metrics that will be identified in the early iterations of the research. In order to support the process of both construction and evaluation of the software artifact, there are two different stages of metrics to be used.

**Design as a Search Process**

The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment. This research features several alternative approaches that were explored in each iterations, along with justifications on the selection of these alternatives.

**Communication of Research**

Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences. While this research is mainly tailored to experts in computer science, it does explain the financial benefits of conducting such research, which would also be attractive to management-oriented audiences.

### 1.5.1 Workflow

The workflow in this research is divided into several parts over the span of a year.

Technical Process (36 weeks)

- Pilot testing (8 weeks)
- Prototype design phase (4 weeks)
- Prototype implementation phase (4 weeks)
- Applied design phase (4 weeks)
- Applied implementation phase (4 weeks)
- Iterative improvement phase 1 (4 weeks)
- Iterative improvement phase 2 (4 weeks)
- Iterative improvement phase 3 (4 weeks)

Report Writing (16 weeks)

- Literature Review (2 weeks)
- Methodology (2 weeks)
- Results Discussion (2 weeks)
- Research Write-up (6 weeks)
- Proof Reading and Final Review (6 weeks)

### 1.5.2 Key Metrics

In order to improve the consistency and integrity of the research, several key metrics are identified and used as a general guideline. The key metrics are primarily used for self-assessment during the span of the research in order to monitor the quality of the research, and each iteration in the research be tested with several key heuristics that will be identified at the beginning of the research.

There are two stages of metrics that are used in this research. The first stage is used at the beginning of the research to select the methods for implementing the code, which includes programming language and environment. The second stage is used to evaluate the results of each iteration and attempts to check if the process is successful.

#### 1.5.1.1 Method Selection

The method selection highlights some key metrics that are used in choosing the methods for implementation. The reason why this metric is important to use is because it allows critical analysis of the methods used to implement the solutions. This allows the researcher to avoid foreseeable difficulties or delays in the research which are caused by methods that are not effective to use.

**Runtime speed**

In any software development process for real-world applications, runtime speed is an important metric to consider. Software that is too slow cannot be used. Time is of the essence. Work has to be done on schedule. When working with large amounts of data, processing speed could become very slow, therefore this metric should be considered when dealing with data processing applications. Decisions have to be made on a timely basis, because any form of optimization has to be economically viable, meaning the cost/benefit has to be effective.

For our purposes, this metric may be similar to overhead cost metric. We use this as the main metric in our research because in business, time is a more familiar concept in business terms. Overhead is not a common concept in business. We assume that the overhead and runtime speed both depend on the efficiency of the process, so if the process is efficient it will save overhead cost and reduce runtime speed. Because the end result is the solution to the problem, not the system itself, it is acceptable to not focus on overhead as long as the solution is being worked on correctly. In other words, we will evaluate changes to our algorithms based on the overhead cost of introducing those changes against the benefits that result in better solutions.

**Customizability**

In order to work on a specific problem, there needs to be a degree of customizability so that it is possible to tailor existing solutions to fit the requirements of the problem. This means that the critical parts of the implementation have to be interchangeable when it is required to do so. A rigid solution that is difficult to modify could potentially cause problems in the future, because it will be hard to change, especially in the field of research where not all of the answers are obvious at first.

In terms of programming, availability of existing libraries and modules is also important in order to ensure that the code is easily customizable. Libraries and modules provide excellent shortcuts to implement solutions into code, in order to not spend time “reinventing the wheel”. When working to a solution to a problem, it is best to learn from what others have made and improve from it. That is the essence of research, “standing on the shoulder of giants”.

**Familiarity with programming environment**

This key metric is more relevant to the researcher than the research itself, but is no less important than the other metrics. The reason is because the researcher is considered to be a key component to this research, as the researcher is involved in designing and implementing code for the software.

In the design process, the researcher has to pick and choose methods to solve problems. There are multiple ways to solve problems and some are faster than others, but the key in solving problems efficiently is to draw knowledge from experience and apply them to the problem. This means that the researcher will often need to choose the method that is most familiar. This also applies to programming methods.

Familiarity is not a metric without flaw, because sometimes the best solutions to the problem may be beyond the scope of the researcher's experience. But the reason why this was considered is because in the implementation process, researchers have to use programming tools and environments that they are most familiar with in order to implement solutions efficiently and effectively. Researchers are also limited by their current skill sets, therefore they have to be able to be resourceful and choose the best programming tools and environments they can use.

### 1.5.1.2 Implementation Evaluation

The implementation evaluation metrics are used to evaluate the results of the software artifact.

**Number of Iterations**

This metric is used in the implementation evaluation in order to measure the overhead cost. The difference between the runtime speed metric in the method selection stage and this metric is that the runtime speed metric mainly refers to the speed of the programming environment used, while the number if iterations metric in implementation evaluation refers to the speed of the actual runtime of each software cycle. This metric is used to determine if the speed of the software implementation is satisfactory or not.

**Fitness**

Fitness is a key metric for evaluating the results. This metric is simple and straightforward to measure because it can be easily quantified, as it is in numerical format. In GA, the fitness value determines the direction of the evolution process, and at the end of the algorithm, the best result is the solution with the

highest or lowest fitness value, depending on the nature of the optimization. Fitness value determines the end result of the optimization, and the success of the optimization can be easily measured with the fitness value.

**Correctness**

Correctness is the key metric in measuring the results. The resulting implementation has to be correct, in which it is free from software bugs. This is because in order to provide proof by example, the example itself has to be correct. If the software implementation is not correct, it would be difficult to prove theoretical assertion from the software demonstration example.

## 1.5.3 Scope and Limitations

This research focuses on exploring viable solutions and optimization to the CTP using GA via in depth analysis of GA metaheuristics and techniques. Other advanced optimization methods such as Ant Colony Optimization is not considered due to the scope and time limitations of this research.

# Chapter 2: Literature Review

## 2.1 Optimization

Optimization can be defined as a process of solving a problem where the objective is to find the best solutions, or to find a feasible region which has the maximum or minimum value, depending on the search function (Atallah & Blanton, 1999, pp. 20-29). Optimization can also be defined as the selection of a best element, with regard to some criterion, from some set of available alternatives (*The Nature of Mathematical Programming*, 2014). Optimization problems have been a research subject before modern computing became prevalent. For example, the TSP has been around since the 18th century (Biggs, 1977). The goal of optimization is to search for a solution or a set of solutions that reaches a global optimum without being trapped in a local optimum. Global optimum can be defined as the best possible solution to a problem, while a local optimum can be defined as a solution of the problem that is better than most other solutions, but worse than the global optimum (Black, 2004). Some examples of optimization problems, or problems that can be solved with optimization, include TSP, knapsack problem, and vehicle routing problem.

Optimization is used in many different fields for a wide range of applications. In the field of mechanical engineering, optimization is taken into account during mechanical design when considering certain physical objectives or constraints such as strength, deflection, weight, wear, corrosion (Rao & Savsani, 2012, p.1). In the field of medicine, optimization is used to increase the effectiveness of certain therapies, such as radiotherapy. In the case of radiotherapy, optimization is done by selecting particular parameters such as the angle and intensity of the beam (Alves & Vicente, 2008, pp. 1 & 47). In economics, optimization is used by managers to improve the decision making process of their organizations (Gavalec, 2015), and by optimizing logistical decisions such as delivery truck routing (Tasan & Gen, 2012), The justification of using optimization in industrial applications is that even a small amount of improvement in cost reduction or quality improvement can be financially rewarding.

Optimization does not have to be perfect in order to be useful. Some problems cannot be easily solved through an exhaustive search because of the large size of search space. This is because the more parameters a problem has, the longer it takes to solve the problem, often to the point that the time taken to solve the problem grows exponentially. These types of problems that cannot be easily solved are called NP-hard problems. In the real world, it is often extremely difficult to achieve perfect optimization, because some problems have many different parameters to satisfy that would otherwise be impossible to perfectly optimize, especially in the case of NP-hard problems. Instead, most advanced optimization techniques rely

on approximation methods, using different sets of heuristics in order to achieve a satisfactory level of optimization. Advanced optimization techniques such as the GA, for example, are designed to be run until the best solution reaches a certain number of iterations or until the solutions converge to a stopping point.

Even though some optimization techniques can perform better than others in specific scenarios, most of the time the effectiveness of the optimization techniques is dependent on the nature and the context of the problem itself. The advantages that one optimization techniques has over another optimization technique in a specific problem are offset by its disadvantage in another problem, and this phenomenon can be explained in a theorem called the 'No Free Lunch' theorem (Wolpert & Macready, 1999). The No Free Lunch Theorem implies that there is no optimization technique that could outperform all other optimization techniques in all problem contexts. This means that in order to select the effective method to optimize the solution to a problem, there needs to be research done in that specific context of problem.

### 2.1.1 Multi Objective Optimization

Multi objective optimization is a class of optimization that is tailored for solving problems that have multiple objectives which are often conflicting with each other. Many problems in reality have multiple objectives that may be conflicting with each other, and the goal of multi objective optimization is to create a set of trade-off solutions that are viable enough to satisfy all of the objectives to some extent. An example of conflicting objectives is the choice between quality and price of a product: a product that has a good quality will often be expensive, a product that is cheap will often have poor quality, and a product that is average in both quality and price will have a tradeoff between quality and price. Some optimization techniques are designed to solve multi objective optimization problems, such as the Multi Objective GAs.

**Pareto Optimum**

A multi objective optimization should have solutions that are Pareto optimal, meaning that they should maximize in at least one of the objectives or achieve a good tradeoff between all of the objectives. According to Legriel et al. (2010), A Pareto Front is a collection of Pareto Optimum solutions that cannot be improved in one objective without sacrificing other objectives, meaning that the set of solutions can be considered optimal choices for decision making.

## 2.2 Travelling Salesman Problem

TSP is an optimization problem that focuses on finding a path through a weighted graph that starts and ends at the same vertex, includes every other vertex exactly once, and minimizes the total cost of edges (Black,

2014). The problem can be simulated as a salesman travelling through a list of cities, creating a route that ends up back at the first city. TSP is a classic optimization problem that has existed in the 18th century (Biggs, 1977), and has been widely researched by researchers in multiple fields from the 1960s (Lawler et al, 1985). The solution to the TSP results in a creation of a shortest Hamiltonian cycle, which is a path through the weighted graph that starts and ends at the same vertex and includes every other vertex exactly once.

TSP is considered to be NP-hard. The goal of TSP is to find the shortest Hamiltonian cycle, and because the complexity of a Hamiltonian cycle problem has been mathematically proven to be NP-complete (Karp, 1972), therefore it follows that TSP is a NP-hard problem. Some researchers have also classified the TSP as a NP-hard problem (Larranaga & Lozano, 2001; Aarts & Korst, 1989). TSP can be applied to a wide variety of problems. It can be applied to logistics, and has spawned many subsets of optimization problems to address specific problems, such as the Vehicle Routing Problem. This makes TSP and its derivatives regarded as one of the most widely researched problems in the field of computer science, particularly in the area of optimization.

There is, however, an additional complicating factor to the CTP which makes it different from the TSP. The choices made at a node will affect the subsequent route taken. That is, the TSP in its classical form is ‘static’ in that the information carried by the network and traveler are constant and the choices made by the traveler do not depend on previous choices made along a route or by other travelers. So a route can be calculated for the traveler without the traveler taking a single step. In the CTP, however, there is a ‘dynamic’ element, which is that the state or condition of the route is altered by the route so far taken by traders. While it may still be possible to calculate an entire route for the trader without the trader taking a single step, the trader is not likely to be the only trader. The information currently at a node may no longer be valid once the trader gets there, because other traders have visited and caused changes at that node. Planning a route is best done stage by stage so that deciding on the next node in the route can take into account the latest information available at that point in the route traversal, where the latest information includes the state of remaining nodes as well as the state of the trader. The optimization problem for the CTP must take into account the degree of ‘look-ahead’ required to provide an optimal route in a dynamically changing network. More details of this dynamic CTP aspect and differences with TSP are provided in Chapter 3.

**TSP variants**

There are several existing TSP variants that explored the optimization of profits. Feillet et al (2005) has reviewed TSP variants with the additional goal of profit optimization, a class of variants that are called TSP with Profits. From the review, the closest form of TSP variant to CTP is the PTP.

The main difference between PTP and CTP is that, in PTP, the goal is to find cycles in the graph that are most profitable without going through all the nodes, while in CTP, all the nodes are visited and trade is only

conducted between profitable nodes. The distinction is made in order to emphasize that in a situation where the traveler is routed to go through several destinations, the traveler would be able to accrue maximum profit by trading along the way.

CTP is proposed in order to model the unique problem of optimizing the route where the traveler has to visit several destinations without skipping through some of the destinations. In some real world cases, a transport often has to go through several routes without trading in order to pass through, and CTP would be able to model those specific cases.

## 2.3 Genetic Algorithm

GA is a heuristic search approach inspired by evolution in nature (Kramer, 2017). The concept of evolution was coined by Charles Darwin in his book, The Origin of Species. The main premise of evolution is 'survival of the fittest', meaning that species survive by adapting to their environment. The surviving species are considered to be fit and can carry their genes to the next generation. The set of heuristics used in GA are similar to how evolution occurs in nature. First, some individuals, which are usually represented by chromosome notations, are selected from the population with a selection criterion. Next, the selected individuals undertake a crossover process, which will mix genetic material between them and create a new individual. After that, some of the population will undergo a mutation that will change their genetic material. Finally, the fitness of the population will be evaluated by a fitness function and the surviving individuals will form the next generation and the process repeated until a set number of iterations or a stopping condition has been reached. Some variations of GA feature several metaheuristics that are used to improve the evolution process or to tailor the algorithm for a specific type of problem. There are several types of metaheuristics, which are used in different stages of the algorithm.

### 2.3.1 Selection Strategy

The selection process determines which individuals will be used to crossover genetic material with each other and create a new offspring solution. There are several possible metaheuristics that can be employed in the selection process.

#### 2.3.1.1 Random Selection

Random selection is one of the conventional metaheuristics used in the selection process of GA. In a random selection, the process of selecting the individuals to crossover is left to chance, and therefore the random selection is a purely stochastic process.

#### 2.3.1.2 Roulette Selection

The Roulette Selection is similar to the random selection, but the main difference is that instead of using a pure stochastic method, the roulette selection process selects individuals based on a uniform distribution. That is, the total fitness of a population is first calculated and the probability of an individual being chosen depends on its fitness contribution to the total fitness.

#### 2.3.1.3 Tournament Selection

A tournament selection is performed by selecting a random pool of individuals, and then selecting the fittest individual from that pool. By selecting only the best individual out of the pool of individuals, the tournament selection process could potentially select a better variety of individuals. Tournament selection has been proven to outperform other selection processes like the roulette selection in specific problems such as the TSP (Razali, 2011).

### 2.3.2 Mutation Strategy

The mutation process changes the genetic material of an individual. Unlike the crossover process, mutation does not require other individuals in order to change the genetic material.

#### 2.3.2.1 Random Mutation

Random mutation is the default metaheuristic used in the mutation process of GA. In random mutation, an individual is selected at random and its genetic material is randomly changed, through a purely stochastic method.

#### 2.3.2.2 Lamarckism

One example of a hybrid approach mutation strategy is the usage of Lamarckism as a means to mutate the population. Lamarckism is based on the idea that an individual can pass certain traits that it has acquired throughout its lifecycle onto its offspring. In the context of GAs, the mutation process in Lamarckism is based on the same concept. Ross (1999) managed to implement Lamarckian GA by selecting one or more individuals from the population, mutates them, and if the mutated versions are more fit than the original individuals puts them back in the population. Lamarckism has been proven to be able to improve the quality of the results of a GA in specific optimization problems including TSP (Wellock & Ross, 2001), and a hybrid solution utilizing Lamarckism has been shown to produce better results than a conventional GA in a multi-objective optimization application (Cuadra et. al., 2016).

#### 2.3.2.3 Baldwin Effect

Another example of hybrid approach mutation strategy is using the Baldwin Effect, or Baldwinism. Baldwinism is based on the theory that learned behaviors of an individual in nature are passed to its offspring. In the context of GAs, the mutation process is different from Lamarckism. As opposed to

Lamarckism, the mutated individual is not put back into the population. Instead, the fitness of the mutated individual, if it is better than its parent, replaces the fitness of the parent (Qi, 2012).

### 2.3.3 Multi Objective Genetic Algorithm

A class of GA that is specialized for multi objective optimization problems is the Multi Objective Genetic Algorithm (MOGA). Unlike a conventional single objective GA, MOGA is designed to optimize multiple objectives. In order to use MOGA, the fitness function has to be customized so that the evolution process results in a set of Pareto Optimum solutions.

### 2.3.4 GA Variants

One variant to the evolutionary algorithm is the Estimation of Distribution Algorithm (EDA), which operates by guiding the search process using constructed probabilistic models of solutions that are promising. Example of EDA includes the Restricted Boltzmann Machine, the Univariate Marginal Distribution Algorithm, and the Population Based Incremental Learning (Shim, et. al., 2011).

## 2.4 Vehicle Routing Problem

VRP is a class of optimization problem that deals with the transportation of products across multiple routes. Tasan and Gen (2012) described the use of GA for solving the VRP-SPD (Vehicle Routing Problem with Simultaneous Pickup and Deliveries) in their paper. There are several parameters for VRP-SPD, such as vehicle capacity, distance between nodes, delivery amount demanded by node, pick-up amount of node, and number of nodes. The difference between VRP-SPD and CTP is that CTP adds several new parameters such as price for each product (one parameter per product type) and the amount of available funds for trading, while also removing the delivery amount demanded by node parameter because of the different optimization goal. CTP also differs from VRP-SPD in that CTP is about monetary optimization while VRP-SPD is about customer delivery optimization.

## 2.5 Chapter conclusion

This chapter has elaborated the theories that are related to the CTP, such as optimization, TSP, and GA, and has provided the theoretical groundwork needed to explore CTP. In the next chapter, analysis and design of the CTP will be discussed in further detail.

# Chapter 3: Preliminary Analysis and Initial Design

This research is conducted in several iterative processes, and this chapter deals with the initial design phase. Before the implementation process begins, a preliminary analysis is conducted in order to explain the concept of the problem, as well as the potential issues and their possible solutions.

## 3.1 Preliminary Analysis

The purpose of conducting a preliminary analysis is to establish the CTP in a tangible form by defining the problem and designing a software implementation in order to have a general idea what the problem looks like. In order to establish the concept of CTP, we have broken down key characteristics, constraints, and goals of the problem into several logical statements. The result of such deduction is as follows.

**Characteristics of the CTP:**

- There are multiple cities.
  - There is a location coordinate for each city
  - There are multiple products in each city.
  - There are different prices for each product in every city.
- There is a travelling caravan.
  - The travelling caravan carries products.
  - The travelling caravan carries money.
- There are trades between cities, in the form of buying and selling.
- There is a route between cities.

**Constraints:**

- At the end of the route, the travelling caravan has return to initial city (forming a Hamiltonian cycle).
- The travelling caravan has a maximum carry capacity for products.
- The travelling caravan has a set initial amount of money.
- The travelling caravan can trade by buying from one city and selling to the next city in the route.
- A trade can only occur if the caravan carries enough money.
- A trade can only occur if it produces a positive sum of profits

**Goal:**

- Maximum amount of money by the end of the route
- Minimum amount of total distance in the route

After conducting a preliminary analysis of the CTP, we found that there are similarities between the CTP and the TSP. We have also broken down the TSP into a similar form as above in order to prove the similarities with the CTP.

**Characteristics of the TSP:**

- There are multiple cities.
  - There is a location coordinate for each city
- There is a route between cities.

**Constraints:**

- At the end of the route, the travelling caravan has return to initial city (forming a Hamiltonian cycle).

**Goal:**

- Minimum amount of total distance in the route.

A quick analysis of TSP and CTP shows that the two optimization problems have several similarities that overlap with each other. CTP contains all of the characteristics, constraints, and goals of TSP, and therefore show that CTP is an extension of the TSP. The implication is that it is possible to implement CTP by initially implementing TSP and then adding the rest of the features until the CTP is successfully implemented.

### 3.1.1 Chromosome Notation

In GA, a chromosome is a representation of a solution to a problem. In the case of TSP, the chromosome represents the cities visited by the travelling salesman in sequential order. The chromosome notation can be written in any format as long as it represents cities, routes and any other relevant information. Figure 3.1 shows an example of a chromosome notation for one solution for the TSP, with the numbers representing Cartesian coordinates of the city in two dimensions, with the X and Y value separated by a comma. The number of genes in the chromosome represents the number of cities that the travelling salesman is travelling through.

| | | | | |
|---|---|---|---|---|
| 60, 70 | 120,80 | 180,100 | 140, 140 | 150, 150 |

**Figure 3.1 TSP chromosome notation example for five cities**

In the CTP, the genes contain more information that includes the trading that occurs between the current city and the next destination city. This means that the information about the trade will become part of the chromosome and be included in the chromosome notation. Figure 3.2 contains a proposed example of a chromosome notation for CTP, with additional information in each gene that represents the products and the quantity of each of the products that are being traded.

| | | | | |
|---|---|---|---|---|
| 60, 70,<br>0, 2, 0 | 120,80,<br>3, 0, 0 | 180,100,<br>0, 0, 1 | 140, 140,<br>4, 0, 0 | 150, 150,<br>1, 0, 0 |

**Figure 3.2 Caravan Trading Problem chromosome notation example for five cities and three products**

In addition to moving between cities, the caravan in CTP performs trade between cities. The gene in CTP contains two additional pieces of information: the type of products being traded and its corresponding quantity. There can be multiple amounts of different products available for trade, and for each amount of trade, the caravan buys products from the current city and sells it to the next. In the example above, there are three varieties of products, which we will name products #1, #2, #3. Looking at the example above, the caravan begins at city location (60, 70) and buys two units of product #2. Then the caravan travels to city location (120,80) to sell the two units of product #2 that were previously bought, and then proceeding to buy three units of product #1; and then travelling to city (180, 100) and selling the three units of product #1 previously bought. This process continues until the end of the destination, and then the caravan will go back to the starting city and sell the last product there. This is an example of a CTP solution.

The cities in the CTP also have additional information in order to accommodate the trading feature. Each city will have a random price for each product, and therefore the caravan will need to utilize the price differences from each city in order to gain profit. At the end of the route, the total profits are calculated and the total distance is measured, and then the fitness of the solution will be determined on the largest profit and the smallest distance between all cities in the route.

### 3.1.2 Generating Correct Solutions

The goal of this phase is to experiment with various methods in order to create a working implementation of the CTP. The preliminary analysis has provided some insight on the possible requirements to the CTP, and

the initial implementation of the TSP source code has created the groundwork for implementing the CTP source code. The next step is to begin the initial implementation by adding the requirements of CTP to the TSP code. We compiled a list of features by using the insight gathered during the preliminary analysis.

| Preliminary Analysis Results | Additional Requirements |
|---|---|
| **Characteristics:**<br>◆ There are multiple cities.<br>- There is a location coordinate for each city<br>- There are multiple **products** in each city.<br>- There are different **prices** for each product in every city.<br>◆ There is a travelling caravan.<br>- The travelling caravan **carries products.**<br>- The travelling caravan **carries money.**<br>◆ There are **trades** between cities, in the form of buying and selling. | - A list of **products**, each with different **prices** in every cities<br>- A list of **carried products**<br>- A variable representing **carried money**<br>- A **trade** function that enables the caravan to buy or sell |
| **Constraints:**<br>◆ At the end of the route, the travelling caravan has return to initial city (forming a Hamiltonian cycle).<br>◆ The travelling caravan has a maximum **carry capacity** for products.<br>◆ The travelling caravan has a set **initial amount of money**.<br>◆ The travelling caravan can trade by buying from one city and selling to the next city in the route.<br>◆ A trade can only occur if the caravan carries enough money.<br>◆ A trade can only occur if it produces a positive sum of profits<br><br>**Goal:**<br>◆ Maximum amount of money by the end of the route | - A variable representing **carry capacity**<br>- A preset variable representing **initial money**<br>- A variable representing **money**<br>- The existence of **carried products** |

| | |
|---|---|
| ◆ Minimum amount of total distance in the route | |

**Table 2.1 The additional requirements analyzed from the preliminary analysis results**

One characteristic of the CTP that we observed during the preliminary analysis is the compounding nature of the trade. If the caravan starts with a small amount of money and continues to make profit as it progresses, then we can assume that the further the caravan travels in the route, the more profit it can accumulate because the caravan will be able to buy more products to sell, assuming that there is enough capacity to carry the goods. This implies that the trading process in one city will affect the profitability of the trading process in the next city, and this effect is compounded as the caravan travels further in the tour.

This compound trade can affect the integrity of creating new solutions, as well as the crossover and mutation operations. In the TSP, the crossover operation is conducted by selecting two solutions and producing an offspring solution by randomly mixing parts of their routes, and mutation is done by swapping destination cities in a solution. However, the problem of using this method of crossover and mutation is that they do not take account of the trade that has already occurred in the route. The compound trading effect adds a routing continuity problem to the CTP, which means that a purchase in one city is dependent on whether there's enough money or not for purchase in the next city. This makes it more difficult to do crossovers and mutation in GA, and the implication of this continuity problem is that a crossover or mutation can possibly result in incorrect solutions.

Consider the following scenario. In the case of a CTP where the caravan travels through cities A, B, C, with prices of products [$1, $2, $3], [$3, $2, $1], [$3, $3, $3], and starting money $5, there are possible incorrect solutions. For example, in the case when there is a purchase when there is not enough money.

[5, 0, 0], [0, 0, 20], [0, 0, 0]

The example above shows a randomly generated trade of each product as the caravan passes through each city. In city A, the caravan purchases five units of product #1 for $1 and sells them at city B for $3, which gives a total of $15. However, in city B, the caravan purchases twenty units of product #3, which is not possible considering the amount of money, since $15 is not enough to buy $20 worth of products.

This incorrect solution shows that we need an extra step at the beginning of the algorithm to filter out incorrect solutions. We will also need to make sure that the solutions stay correct after crossover and mutation. If a solution has been mutated or created through crossover, the route will change and the trade will not be valid anymore, because the destination cities also change.

In order to generate correct solutions that address this continuity problem, the method is to use Greedy Algorithm in conjunction with GA, or a GA-Greedy Algorithm hybrid method. The purpose of using a Greedy Algorithm is to find an optimum trade without generating incorrect solutions, because the algorithm ensures continuity of trade by traversing each city in the route and finding the optimum trade between each city (Figure 3.3). The Greedy Algorithm is used after every mutation and crossover, in order to make sure that the trade in every route is correct.

```
For each city
    For each product
        Compare prices of all product with next city
    Select the product with most profitable gap between prices
    Buy that product in this city and sell in next city
    If we reach the last city, sell all products
```

**Figure 3.3 Pseudocode for the greedy algorithm**

**Consistency of experiment baseline**

In order to set up a baseline for the experiments, the location and prices at each city have to be consistent. The purpose of maintaining a consistent baseline is to make sure that we can properly observe changes in the results, and to make sure that every change in results can be attributed to a specific change in parameter. This is done by using a random seed when generating the location and prices of each city, so that every time the software runs, the location and prices of each city remain consistent for each seed. The experiment runs the GA multiple times, each for every seed, and then the results are averaged.

### 3.1.3 Tournament-Based Lamarckian Genetic Algorithm

In the process of searching for suitable Lamarckian mutation heuristics, a new strategy of using Lamarckism in the context of selection strategy was developed, and was named the Tournament-Based Lamarckian Genetic Algorithm (TBLGA). The difference between conventional Lamarckism and TBLGA is that conventional Lamarckism works by mutating and seeing the potential of the chromosome before selecting them in order to avoid local minima, while TBLGA only mutates parent chromosomes and choosing the best among the mutation to be selected for crossover, resulting in a superior offspring. More details are provided in the next chapter.

## 3.2 Chapter conclusion

In-depth analysis of the CTP has managed to reveal the characteristics, constraints, and goals of the problem. The analysis has also made it possible to design a customized chromosome notation for the CTP and the TBLGA method.

# Chapter 4: Implementation

Before actual implementation process begins, some experimentation was conducted to explore multiple tools and methods in order to successfully implement the CTP. After analyzing the CTP, we have deduced the requirements for its implementation, and we have shown that the CTP is an extension of the TSP. From this information, we considered that in order to successfully implement the CTP into software artifact, the first step is to implement the TSP and add in the extra requirements of the CTP. The reason for this is because the CTP is an extension of TSP; therefore it would be more effective to start from implementing a TSP and then add additional features on top of the implementation.

## 4.1 Method Selection

We explored several methods on how the implementation of the software should be conducted. TSP is a classic optimization problem; therefore there would already be multiple software implementations and tools ready to be used to optimize the problem. We briefly used these tools and provide our analysis on which tool would be best to be used in this research.

### Method 1: Optaplanner

Optaplanner is a Java-based tool that has optimization solutions ready to use for practical purposes.

**Experimentation Results:** After briefly using the tool and reading through the documentation, it was shown that there is no GA feature in the optimization tool currently available. The tool seems to be configured to be a black box, which focuses on the results of the optimization rather than the heuristics used to obtain the results.

**Analysis**: While we find the tool to be useful in practical use of common optimization problems, there is a lack of ability to customize the metaheuristics of the optimization algorithms. Furthermore, there is a lack of GA optimization methods, which is a core subject in this research. We conclude that this tool not suitable for the context of this research.

### Method 2: Matlab GA Toolbox

Matlab provides toolboxes for many optimization problems, and one of them is the GA Toolbox.

**Experimentation Result**: A brief usage shows that the toolbox has an intuitive user interface that allows the user to run the GA with little programming required. There are specific solutions tailored for the TSP and Multi Objective GA.

**Analysis**: The nature of the toolbox is a black box with not much room for customizing possible available parameters quickly. While we found that this tool could be useful for this research, we compared with other methods and decided to use other methods.

### Method 3: Find existing code and tweak

In software development, it is considered to be common practice to search for existing solutions and modify them to fit the context of the problem. Our justification for using this method in this research is that the TSP source code that we found is only used as a base template for the final CTP software artifact, which will be vastly different after adding all the other functionalities that we identified during the preliminary analysis process. Therefore, we argue that this method produces an original software artifact.

The initial process is to search for a source code for a TSP solution using GA, and we found a source code which was written in Python (Piotr, 2012). After that, we proceeded to look for an alternative source code in order to find a comparison, and we found one which was written in Java (Jacobson, 2012). Parameters for both source codes are identical, with the only difference being the programming language. Both source codes are run ten times and we measured their performance.

**Average runtime of GA implementations on a 10 city TSP**

| | |
|---|---|
| Python implementation: | 9204 ms |
| Java implementation: | 326 ms |

One noticeable difference between the source codes is that the Java source code performs significantly faster than its Python counterpart. One possible explanation for this difference in speed is that Java is a compiled language, which is faster during execution than scripting languages such as Python. Merelo-Guervós (2016) has conducted several experiments in running evolutionary algorithms using different programming languages and testing them with benchmarks, and has arrived to the same conclusion that supports our explanation.

## 4.2 Source Code Classes

After experimenting with various methods, we selected the Java TSP source code to be used as a base template for the CTP. The Java source code contains several classes that have their own functionalities.

Core functionalities:

- CTP_GA (Main class)
- City
- Tour
- TourManager
- Population
- GA

User interfaces:

- CartesianGraph
- FitnessGraph
- ParetoGraph

| **City** |
| --- |
| - x: integer<br>- y: integer<br>- price: double[] |
| + City()<br>+ getX(): integer<br>+ getY(): integer<br>+ getPrice(): double[]<br>+ distanceTo(City otherCity): double<br>+ toString(): String |

The City class is added and modified to contain additional data about a city necessary for the CTP. The class contains the X and Y Cartesian coordinates of the city denoted in integer values, and an array of prices for different types of products. The function distanceTo calculates the distance to another City using the Pythagorean formula, and the function toString outputs the coordinates of the city as well as the prices for all products in a string format.

| **Tour** |
| --- |
| - distance: double<br>- profit: double<br>- fitness: double<br>- tour: List<City><br>- money: List<double><br>- trade: List<List<Integer>> |
| + Tour()<br>+ containsCity(City selectedCity): boolean<br>+ generateIndividual()<br>+ generateTrade()<br>+ getCity(integer tourPosition): City<br>+ setCity(integer tourPostion, City city)<br>+ getDistance(): double<br>+ getFitness(): double<br>+ getMoney(): double<br>+ getProfit(): double<br>+ getTrade(int tourPosition): List<Integer><br>+ tourSize(): integer<br>+ toString(): String |

The Tour class is added and modified to contain data about a single journey between cities, which is referred to as a tour. The class contains the total distance between all cities within the tour, the fitness value of the tour, a sequential list of City objects that represents the actual tour sequence, as well as the trade for each product and the resulting profits in each city. The containsCity function checks if a city exists in the tour list. The generateIndividual function randomly creates a tour from all the cities. The getCity and setCity functions get and set cities from an index that represents the position of the city within the tour. getDistance calculates the distance between all cities in the tour and stores the value, The getFitness calculates and stores the fitness value of the tour, in which the smaller the distance, the bigger the fitness value. The tourSize function returns the size of the tour, and the toString function outputs the tour in a string format. In addition to the all the features, the getMoney, getTrade, and getProfit adds the trade functionalities of the CTP into the code. The function for the fitness evaluation is put here.

| <<static>><br>**TourManager** |
| --- |
| - maxX: integer<br>- maxY: integer<br>- destinationCities: List |
| + addCity (City newCity)<br>+ getCity(integer index): City<br>+ getMaxX(): integer |

| |
|---|
| + getMaxY(): integer<br>+ getProductVariety(): integer<br>+ getRandomSeed(): integer<br>+ getStartingMoney(): double<br>+ numberOfCities(): integer |

The TourManager is a static class that is added and modified to store some meta information about the tour, as well as important starting parameters for the CTP, such as the number of products for sale, the random seed for the random city generation, the amount of starting money, and the maximum carrying capacity of the caravan. The class contains the list of all destination cities to be put in the Tour, and it also stores the maximum value of the X and Y coordinates between all cities. The function addCity adds a new city to the list of possible destinations.

| **Population** |
|---|
| - tours: List |
| + Population(integer populationSize, boolean initialize)<br>+ getFittest(): Tour<br>+ getTour(integer index): Tour<br>+ populationSize(): integer<br>+ saveTour(integer index, Tour tour) |

The Population class represents a collection of Tours that are ready to be evolved. The function getFittest selects the fittest Tour from the entire population. The fitness function is modified from a single objective to a multi objective, adding the maximization of profit as an additional objective in addition to the minimization of distance.

| **GA** |
|---|
| - elitism: boolean<br>- lamarckian: boolean<br>- mutationRate: double<br>- tournamentSize: integer<br>lamarckianMutationSize: integer |
| + crossover(Tour parent1, Tour parent2): Tour<br>+ evolvePopulation(Population pop): Population<br>+ mutate(Tour tour)<br>+ tournamentSelection(Population pop): Tour<br>+ lamarckianSelection(Population pop): Tour |

The GA class contains the core functionality and modifiable parameters of the GA. The selection strategy uses a Tournament-Based Lamarckian Selection and is compared with a same GA using a regular tournament selection for a comparison.

```
Initialize parameters
Generate random population
While iteration < max iterations
    Calculate fitness of each solutions
    Selection strategy (TBLGA or non-TBLGA depending on parameter)
    Crossover
    Mutation
    Run greedy algorithm to calculate profit for this iteration
```

**Figure 4.1 Pseudocode for the system**

The system is then run as many times as the changes for the parameters are needed. The parameters that are used for the system are described in Chapter 5.

**Testing**

The functionality of the modifications to the Java TSP was tested at the beginning of the software development life cycle. During the initial implementation, there were several bugs that were then fixed in the following iterations of the life cycle.

**Chapter Summary**

In summary, the basic classes of Java TSP have been significantly modified and added to for extending the approach to the CTP in the following ways:

- The addition of prices into the City class.
- The addition of money and trade into the Tour class.
- The addition of product variety, random map seed, and starting caravan money into the TourManager class.
- The modifications of the fitness function in the Population class to include the maximization of profit as an additional objective.
- The modification of GA crossovers to include a Tournament-Based Lamarckian Selection.

Appendix A contains the full Java code for the project. In the next chapter, the results of the implementation of the CTP will be discussed in detail.

# Chapter 5: Experiments and Results

## 5.1 Experiment Parameters

We conducted experiments in order to test the effectiveness of TBLGA. The parameters used for the GA and the CTP initializations used in this research are as follows:

CTP Parameters

- **Random Map Seed: 0 to 9**
- **Amount of cities: 10, 20, 50, 100**
- Starting money: $50.0
- Product Variety: 3
- Maximum caravan carry capacity: 100

GA Parameters

- Population size: 100
- Iteration: 5000
- **Selection Strategy: Tournament Selection, TBLGA (Tournament Size: 5)**
- Mutation Rate: 0.002
- Fitness Function: Total Profit / Total Distance
- Elitism: True

The parameters above are run in order to discover the significance of TBLGA selection strategy compared to a non-TBLGA. Different graphs with city locations are randomized with map seeds, and each seed is run with different number of cities. The number of iterations were set to 5000 after it was shown that the fitness value continued to rise gradually for the TBLGA, after going over the local minimum for the non-TBLGA counterpart (see Figure 5.4).

Appendix B describes the output of all experiments reported in this chapter. The experiments are run through ten different maps which are generated through ten different randomization seeds, and the results of each run are averaged. The GA produces a written output of the solution. However, the results would be easier to

check if there is a visual representation of the output. A Cartesian graph was made to visualize the X and Y Cartesian coordinates of the cities and their corresponding trade (Figure 5.1). The interface generates dots that represent the location of the cities, and then it draws lines between the cities in order to highlight the order of which cities are visited first. In addition to the TSP, the user interface was designed to be compatible with CTP. The interface shows the city coordinates and prices of products in each city, as well as the profits. There are also other visualization outputs such as the Fitness Graph which plots the historical fitness values at each generation, and a scatter plot graph which plots all solutions in the population into a graph with the distance and profit objectives.

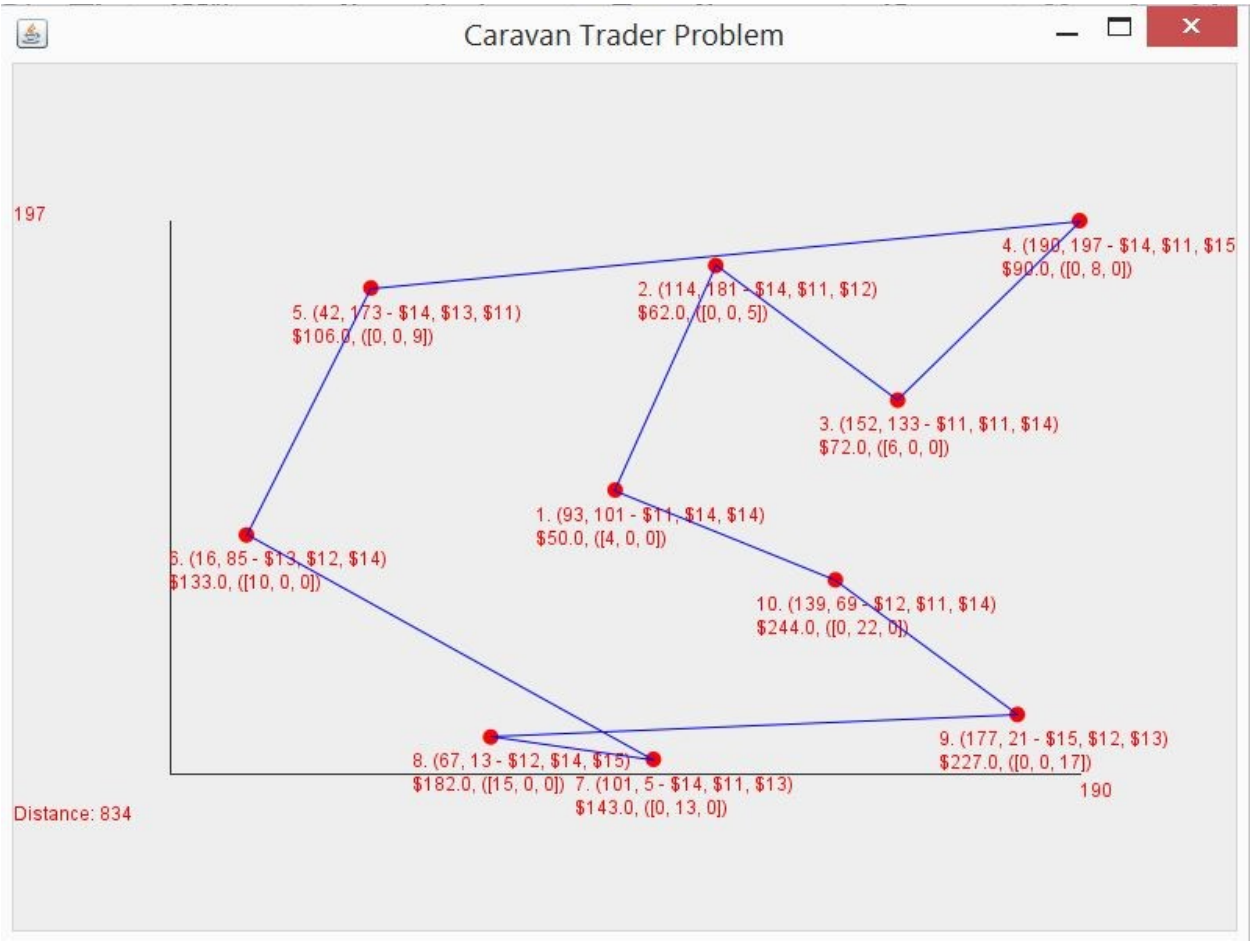


**Figure 5.1 Cartesian graphs showing an example of a route.**

In Figure 5.1, the top left city is the figure is the fifth city visited in the route, with the Cartesian coordinate of (42, 173) and the prices for product #1, #2, and #3 are $14, $13, and $11 accordingly. At this position, the caravan has $106 in cash to purchase products, and is preparing to purchase 9 units of product #3 to be sold in the next city in the route.

# 5.2 Benchmark Testing Results

We have conducted a benchmark test using the results in Appendix B that compares the fitness value between TBLGA and non-TBLGA across different number of cities. The result shows that on average, TBLGA tends to improve the fitness values of larger number of cities.

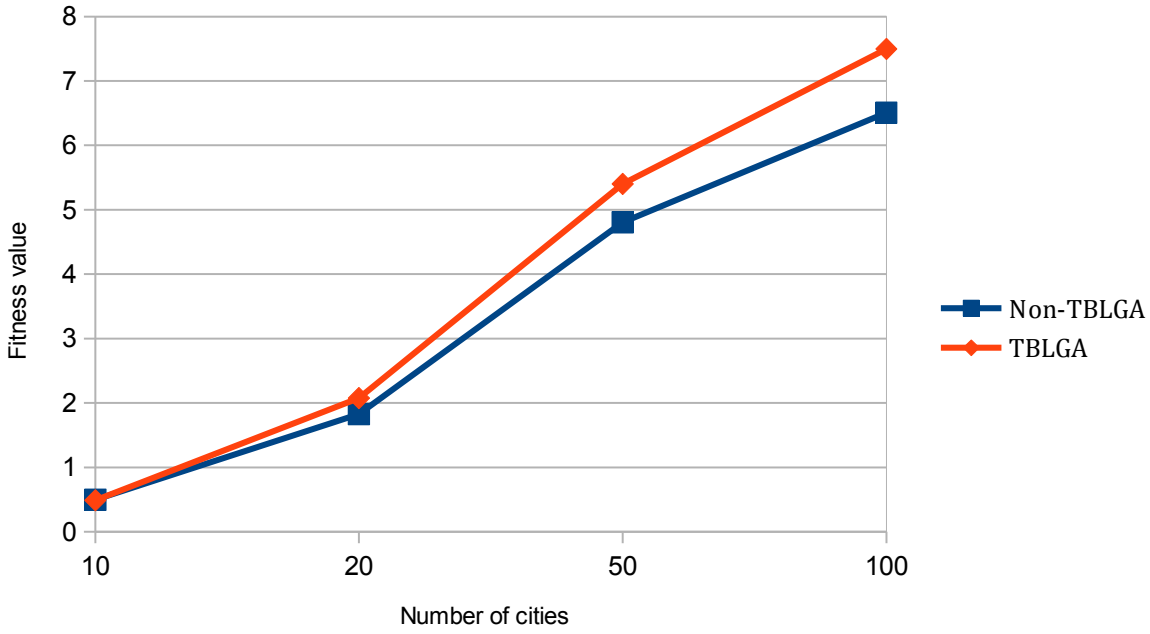


**Figure 5.2 Differences in fitness values between TBLGA and Non-TBLGA based on results in Appendix B**

The fitness function of the GA is the total profit divided by the distance of the tour. The reason why the fitness values increase in proportion with the number of cities is because the more cities that the caravan visits, the more opportunities for trading, and subsequently more profit at the end of the tour. We have found that there are slight improvements on the final fitness value of the TBLGA solutions when compared to the non-TBLGA solutions.

| Cities | Non-TBLGA average fitness value | TBLGA average fitness value | Fitness value increase |
|---|---|---|---|
| 10 | 0.4577973526 | 0.4673917309 | 2% |
| 20 | 1.877239659 | 2.1003462085 | 12% |
| 50 | 4.7254050547 | 5.1494609327 | 9% |
| 100 | 6.6213913125 | 7.2188570384 | 9% |

**Figure 5.3 Average fitness values of each runs and the percentage of increases in fitness values based on results in Appendix B**

The experiments that were done on ten different randomized maps were then averaged and grouped based on the number of cities in the experiments. The result is that the fitness value of TBLGA outperforms its non-TBLGA counterpart by a small margin.

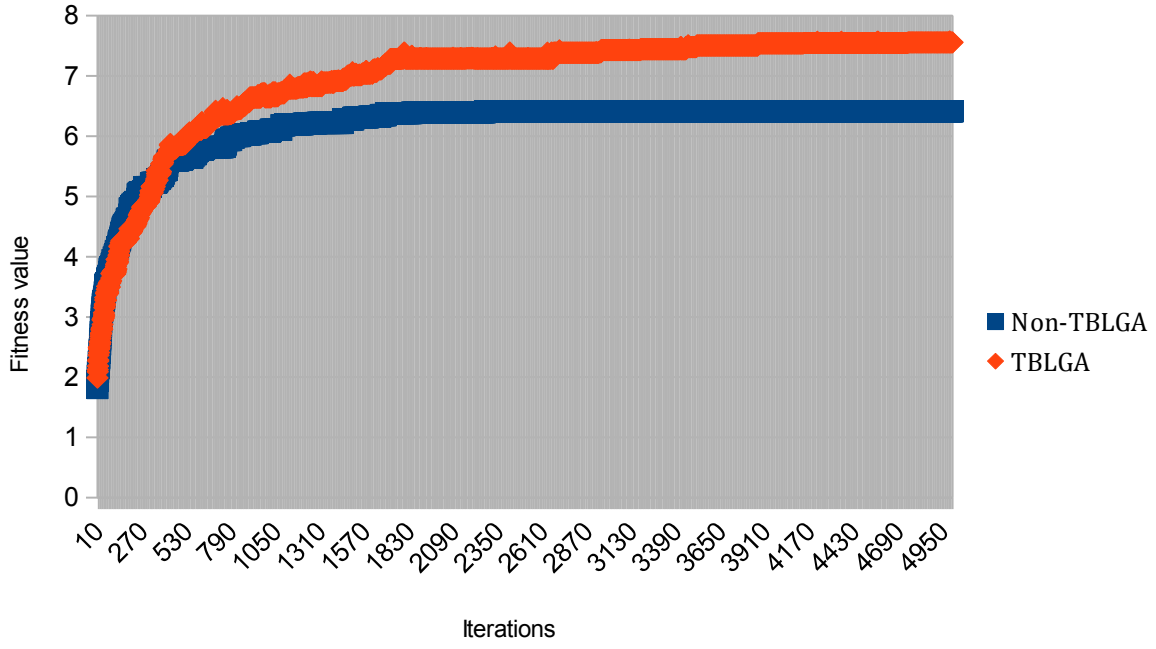


**Figure 5.4 Fitness Graph of a 100 city TBLGA route compared to a non-TBLGA route in 5000 iterations**

Several of the random maps were selected and changes to the fitness value over 5000 iterations were measured. The result was that the non-TBLGA solutions reached a local optimum faster than the TBLGA solutions. This shows that TBLGA solutions have the potential to overcome local optima.

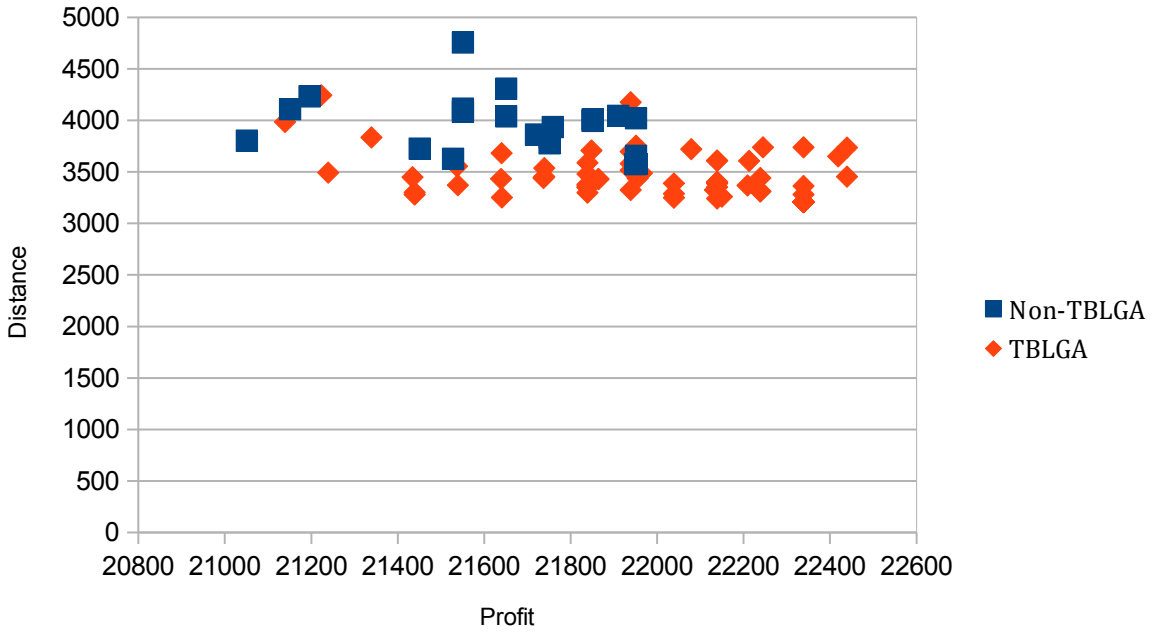


**Figure 5.5 Scatter Plot of a TBLGA population in comparison to a non-TBLGA population**

The population of the TBLGA and the non-TBLGA is compared and plotted in a graph that explains the profit and distance. In Figure 5.5, the optimum solutions are the solutions that have the least distance and the most profit, which means that the solutions closest to the bottom right corner of the graph are the most

optimum solutions. TBLGA resulted in a population that provided better tradeoff solutions than the non-TBLGA population.

## 5.3 Statistical Significance Analysis

The significance in fitness values between TBLGA and non-TBLGA solutions are compared with a one-way ANOVA test for each group of experiment, and the results are as follows:

| Cities | P-value |
|---|---|
| 10 | 0.7404409667 |
| 20 | 0.082856653 |
| 50 | 0.017418451 |
| 100 | 0.0007951605 |

**Figure 5.6 Results from one-way ANOVA test between TBLGA and non-TBLGA solutions**

This analysis shows that the statistical significance between of TBLGA and non-TBLGA solutions can be concluded using the results of the experiment with at least 50 cities. ($P < 0.05$). This analysis supports the conclusion that the slight increase in fitness value applies on higher number of cities, which is observed to be 50 or more. The cutoff point might be higher than 20 and lower than 50, but this has yet to be proven.

**Summary**

The results show that TBLGA provided a slight increase in fitness value for graphs with nodes higher than 50 points, and helped guide the search avoid reaching a local optimum, and resulted in a population with comparatively better tradeoff than a non-TBLGA population.

# Chapter 6: Discussion and Conclusion

## 6.1 Critical Analysis

### 6.1.1 Lack of existing benchmark testing

Unlike TSP where the problem has been challenged and benchmarked multiple times, there is no benchmark test for CTP to judge how effective the solution was. This prompted us to conduct our own test. The limitation of our benchmark testing is that it does not take account of conventional Lamarckism compared to our method. In the future, more benchmark tests need to be conducted in order to have more conclusive results.

### 6.1.2 Justification on why systematic exhaustive approach is not used

It would be difficult to do an exhaustive approach such as with pure dynamic programming, because the search space of a multiple objective optimization is too big and will not work effectively. The currently selected method is a tradeoff between the exhaustive nature of greedy algorithm and the stochastic nature of GA. The advantage of combining both methods is that we can have the flexibility of GA that is good for searching the global space, while at the same time, utilizing the exhaustive nature of the greedy algorithm to search the local space.

### 6.1.3 Convergence

The value of both profit and distance converged towards the highest fitness value, and did not produce a true conflicting multi-objective space. We expected that the value of profit and distance would become inversely proportional and would produce a multi- objective space, but the optimization results in both objectives being optimized as they converge to a single point. Such convergence was not predicted at the start of the project. Our assumption that the CTP is an example of a possibly conflicting multi-objective problem has therefore been shown to be wrong as a result of our initial experiments reported here. In other words, while the CTP is subject to two different objectives, these objectives are not necessarily conflicting for the CTP.

## 6.2 Conclusion

An initial analysis on the CTP shows that the problem has more parameters to optimize than TSP, and requires a multiple objective to be solved. A software model was designed to emulate the CTP and a proposed form of GA, namely the TBLGA, was implemented and compared with the standard GA. The result of the experiments was that TBLGA improved the evolution process by slightly increasing the fitness value, reducing the occurrence of local optima, and producing populations with comparatively better tradeoffs. We tested the statistical significance of the results, and discovered that this result is conclusive on higher number of nodes (50 or more). We concluded that TBLGA could be used to guide the local search space in multi objective optimization problems such as CTP. Since TBLGA has been shown to be effective for the CTP, the same approach, with suitable modification of parameters and data structures, can be recommended for other problems similar to the CTP. The issue of whether TBLGA can deal with genuinely conflicting multi-objective problems remains open.

## 6.3 Alternative Approaches and Future Work

Since the time to conduct the research is limited, we could only choose to experiment with GA with Lamarckism. We analyzed several possible scenarios that could possibly occur if the experiment had gone a different direction.

### 6.3.1 Ant Colony Optimization

The application of Ant Colony Optimization in solving the CTP could be one of the most interesting alternative approaches. In Ant Colony Optimization, the concept of pheromone evaporation could be explored in solving the CTP. A potential issue in implementing ACO is that ants might converge on a local minimum and accumulate pheromone, making the results trapped in a local minimum. However, the ‘no free lunch’ theorem states that without prior information, all optimization techniques in general performs relatively similar, and it suggests that this hypothesis might not be guaranteed and has to be tested in a future work in order to be proven. Due to scope limitations, this research does not conduct experiments using ACO.

### 6.3.2 Implementation of GA Variants

In the literature review, we explored several specialized GA variants, such as the Estimation of Distribution Algorithm (EDA). EDA, such as the Restricted Boltzmann Machine, the Univariate Marginal Distribution Algorithm, and the Population Based Incremental Learning, and their viability in optimizing the CTP could be explored in this research as an alternative approach.

### 6.3.3 Distance Cost in CTP

One possible addition to the CTP model to be added in future research is distance cost. The CTP with distance cost would have its profit margin is deducted by the amount of distance travelled multiplied by a fixed fuel cost. If distance cost is added, it is possible that it would increase the inverse proportionality of both objectives and create an objective space instead of a convergence, possibly resulting in a creation of a Pareto Front. We did not have enough time to create an alternative CTP model that applies the distance cost, but we considered that this could be a good alternative approach to be explored in the future.

### 6.3.4 Pure Stochastic Trade Generation in CTP

One possible solution to generating individuals is to generate trade numbers in a random manner. The trade can be either in the positive or negative, which represents both buying and selling. The advantage of using this method is that it allows greater flexibility of trade by allowing the caravan to trade to distant cities in addition to the next city visited. Using a model of the subset sum problem, the total sum of trade has to reach zero or more at the end of the tour, meaning that for every sale of the product, there has to be a purchase of the product beforehand. This is in line with the physical limitation of objects, because in the real world, it would be impossible to sell products when there is none in stock, meaning that the total sum of purchase cannot be negative.

In the subset sum problem, solutions are either correct or incorrect. If the sum is not zero then it is not correct. There is a GA solution to the subset sum problem (Wang, 2004) where the fitness value is a binary value, and it may provide the method to achieve this. This type of problem becomes a “take all or none” problem. If the subset sum becomes zero when added, it passes the fitness function. If the subset sum is a non zero number, it does not pass the fitness function. This fitness value can be denoted in binary value, either a 0 or a 1.

The reason why this method was not implemented is that it would take a long time to generate a correct trade because for each trade generated, there are chances that the sum of the trade does not equal zero. The time taken to filter out the non-zero sum solutions and replacing them until there is enough solutions would be better used for creating trade that are guaranteed to be correct and then optimizing them. This is the reason why the Greedy Algorithm hybrid was used instead of a pure stochastic method.

# References


Aarts, E. H., & Korst, J. H. (1989). Boltzmann machines for travelling salesman problems. *European Journal of Operational Research*, *39*(1), 79-95.

Abdullah Konak, David W. Coit, Alice E. Smith, Multi-objective optimization using genetic algorithms: A tutorial, Reliability Engineering & System Safety, Volume 91, Issue 9, 2006, Pages 992-1007, ISSN 0951-8320, http://dx.doi.org/10.1016/j.ress.2005.11.018(http://www.sciencedirect.com/science/article/pii/S0951832005002020

Alves, C. J., Pardalos, P. M., & Vicente, L. N. (2008). *Optimization in medicine*. New York: Springer.

Atallah, M. J., & Blanton, M. (Eds.). (1999). *Algorithms and theory of computation handbook, volume 2: special topics and techniques*. CRC press.

Biggs, N. (1977). *Graph Theory, 1736-1936*.

Black, P & Pieterse, V. (2004), *Global Optimum,* Dictionary of Algorithms and Data Structures. Retrieved on November 1, 2017 from https://www.nist.gov/dads/HTML/globalOptimum.html

Black, P & Pieterse, V. (2004), *Local Optimum,* Dictionary of Algorithms and Data Structures. Retrieved on November 1, 2017 from https://www.nist.gov/dads/HTML/localoptimum.html

Chisholm, Hugh. (1911). "Caravan".*Encyclopædia Britannica* (11th ed.). Cambridge University Press.

Craven, B. D., & Islam, S. N. (2005). *Optimization in economics and finance : some advances in non-linear, dynamic, multi-criteria and stochastic models*. Dordrecht : Springer, 2005.

Cuadra, L., Aybar-Ruíz, A., del Arco, M., Navío-Marco, J., Portilla-Figueras, J., & Salcedo-Sanz, S. (2016). A Lamarckian Hybrid Grouping Genetic Algorithm with repair heuristics for resource assignment in WCDMA networks. *Applied Soft Computing*, *43*619-632. doi:10.1016/j.asoc.2016.01.046

Feillet, D., Dejax, P., & Gendreau, M. (2005). Traveling salesman problems with profits. *Transportation science*, *39*(2), 188-205.

Gavalec, M. (2015). *Decision making and optimization : special matrices and their applications in economics and management*. Cham : Springer, 2015.

International Business Machines Corporation. (n.d.). How companies use optimization. Retrieved November 16, 2017, from https://www-01.ibm.com/software/websphere/optimization/benefits/

Jacobson, L. (2012). *Applying a genetic algorithm to the traveling salesman problem*. Retrieved from http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5

Karp, R. M. (1972). *Reducibility among combinatorial problems*. Berkeley: University of California.

Ken, P., Tuure, T., Marcus A., R., & Samir, C. (2007). A Design Science Research Methodology for Information Systems Research. *Journal Of Management Information Systems*, (3), 45. doi:10.2753/MIS0742-1222240302

Kramer, O. (2017). *Genetic algorithm essentials*. Cham, Switzerland : Springer, 2017.

Laporte, G., & Martello, S. (1990). The selective travelling salesman problem. *Discrete applied mathematics*, *26*(2-3), 193-207.

Larrañaga, P., & Lozano, J. A. (Eds.). (2001). *Estimation of distribution algorithms: A new tool for evolutionary computation* (Vol. 2). Springer Science & Business Media.

Lawler, E. L., Lenstra, J. K., George, R. K., Shmoys, D. B., & Hurkens, C. A. (1985). *The traveling salesman problem: a guided tour of combinatorial optimization*. Chichester: Wiley.

Legriel, J., Le Guernic, C., Cotton, S., & Maler, O. (2010, March). Approximating the pareto front of multi-criteria optimization problems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 69-83). Springer, Berlin, Heidelberg.

Merelo-Guervós, J. J., Blancas-Álvarez, I., Castillo, P. A., Romero, G., Rivas, V. M., García-Valdez, M., ... & Romáin, M. (2016, July). A comparison of implementations of basic evolutionary algorithm operations in different languages. In *Evolutionary Computation (CEC), 2016 IEEE Congress on* (pp. 1602-1609). IEEE.

Moore, G. E. (1975) Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-

13.]. (2006). *IEEE Solid-State Circuits Society Newsletter, Solid-State Circuits Society Newsletter, IEEE, IEEE Solid-State Circuits Soc. Newsl*, (3), 36. doi:10.1109/N-SSC.2006.4804410

Paul E. Black, "traveling salesman", in*Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black, eds. 2 September 2014. Available from:https://www.nist.gov/dads/HTML/travelingSalesman.html

Piotr (2012) *Applying a genetic algorithm to the traveling salesman problem*. Retrieved from https://gist.github.com/turbofart/3428880

Rao, R. V., & Savsani, V. J. (2012). *Mechanical design optimization using advanced optimization techniques*. London: Springer.

Razali, N. M., & Geraghty, J. (2011, July). Genetic algorithm performance with different selection strategies in solving TSP. In *Proceedings of the world congress on engineering* (Vol. 2, pp. 1134-1139).

Ross, B. J. (1999). A Lamarckian evolution strategy for genetic algorithms. *Practical handbook of genetic algorithms: complex coding systems*, *3*, 1-16.

Shim, V. A., Tan, K. C., Chia, J. Y., & Chong, J. K. (2011). Evolutionary algorithms for solving multi-objective travelling salesman problem. *Flexible Services And Manufacturing Journal*, *23*(2), 207-241.

"The Nature of Mathematical ProgrammingArchived 2014-03-05 at theWayback Machine.," *Mathematical Programming Glossary*, INFORMS Computing Society.

Wang, R. L. (2004). A genetic algorithm for subset sum problem. *Neurocomputing*, *57*, 463-468.

Wellock, C., & Ross, B. J. (2001). An examination of Lamarckian genetic algorithms. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers* (Vol. 9, pp. 474-481).

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, *1*(1), 67-82.

# Appendix A: Source Code

```
/*
* CTP_GA.java
* Create a tour and evolve a solution
*/

package ctpga;


import java.util.ArrayList;
import java.util.List;


public class CTP_GA {

    public static void main(String[] args) {

        // Create and add our cities
        int seed = TourManager.getRandomSeed();
        int cityAmount = TourManager.getCityAmount();
        for(int i = 0; i < cityAmount; i++){
            TourManager.addCity(new City(seed+i));
        }

        long startTime = System.currentTimeMillis();

        // Initialize population
        Population pop = new Population(100, true);
        System.out.println("Initial distance: " + pop.getFittest().getDistance());
        System.out.println("Initial profit: " + pop.getFittest().getProfit());

        // Graph that plots fitness with generation
        List deltaFitness = new ArrayList<Double>();

        // Evolve population for x generations
        pop = GA.evolvePopulation(pop);
        for (int i = 0; i < TourManager.getIteration(); i++) {
            pop = GA.evolvePopulation(pop);
            Tour fittest = pop.getFittest();
            double fitness = fittest.getFitness();
            int distance = fittest.getDistance();
            double profit = fittest.getProfit();
            deltaFitness.add(fitness);
            System.out.println("Fitness/Profit/Distance at generation "+(i+1)+": "
+fitness+"/"+profit+"/"+distance);
        }

        long totalTime = System.currentTimeMillis() - startTime;

        // Print final results
        System.out.println("Finished");
```

```
        System.out.println(pop.populationSize()+" total solutions");
        for(int i = 0; i < pop.populationSize(); i++)
            System.out.println(pop.getTour(i));
        System.out.println("Solution:");
        System.out.println(pop.getFittest());
        System.out.println("Final distance: " + pop.getFittest().getDistance());
        System.out.println("Profit amount: "+pop.getFittest().getProfit());
        System.out.println("Profit percentage: "+
(pop.getFittest().getProfit()/TourManager.getStartingMoney()*100)+"%");
        System.out.println("Time taken: "+totalTime+" ms");

        // Draw graphs
        Tour fittest = pop.getFittest();
        new IterationGraph(fittest);
        new FitnessGraph(deltaFitness);
        new ParetoGraph(pop);


    }
}
```

```
/*
* City.java
* Models a city
*/

package ctpga;

import java.util.Random;

public class City {
    int x;
    int y;
    double[] price;

    // Constructs a randomly placed city
    public City(){
        this.x = (int)(Math.random()*200);
        this.y = (int)(Math.random()*200);
    }

    // Constructs a randomly placed city with a seed
    public City(int seed){
        Random r = new Random(seed);
        this.x = r.nextInt(200)+1;
        this.y = r.nextInt(200)+1;
        price = new double[TourManager.getProductVariety()];
        for(int i = 0; i < TourManager.getProductVariety(); i++){
            price[i] = r.nextInt(5)+11;
        }
    }
```

```
    // Constructs a city at chosen x, y location
    public City(int x, int y){
        this.x = x;
        this.y = y;
    }

    // Constructs a city at chosen x, y location with prices
    public City(int x, int y, double... prices){
        this.x = x;
        this.y = y;
        price = prices;
    }

    // Gets city's x coordinate
    public int getX(){
        return this.x;
    }

    // Gets city's y coordinate
    public int getY(){
        return this.y;
    }

    // Gets city's prices
    public double[] getPrice(){
        return price;
    }

    // Gets the distance to given city
    public double distanceTo(City city){
        int xDistance = Math.abs(getX() - city.getX());
        int yDistance = Math.abs(getY() - city.getY());
        double distance = Math.sqrt( (xDistance*xDistance) + (yDistance*yDistance) );

        return distance;
    }

    @Override
    public String toString(){
        String print = getX()+", "+getY()+" - ";
        for(int i = 0; i < price.length; i++){
            print = print + "$"+(int)price[i];
            if(i < price.length-1) print = print + ", ";
        }
        return print;
    }

}
```

```
/*
```

```
* Tour.java
* Stores a possible solution
*/

package ctpga;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Tour{

   // Holds our tour of cities
   private List tour = new ArrayList<City>();
   // Holds our money at any point in the tour
   private List money = new ArrayList<Double>();
   // Holds the trade for each city
   private List<List<Integer>> trade = new ArrayList<List<Integer>>();
   // Cache
   private double fitness = 0;
   private int distance = 0;
   private double profit = 0;

   // Constructs a blank tour
   public Tour(){
      for (int i = 0; i < TourManager.numberOfCities(); i++) {
         tour.add(null);
         money.add(0.0);
         trade.add(new ArrayList<Integer>());
         for (int j = 0; j < TourManager.getProductVariety(); j++) {
            trade.get(i).add(0);
         }
      }
   }

   public Tour(ArrayList tour){
      this.tour = tour;
   }

   // Creates a random individual
   public void generateIndividual() {
      // Loop through all our destination cities and add them to our tour
      for (int cityIndex = 0; cityIndex < TourManager.numberOfCities(); cityIndex++) {
        setCity(cityIndex, TourManager.getCity(cityIndex));
      }
      // Randomly reorder the tour
      Collections.shuffle(tour);
      generateTrade();
   }
   // Uses greedy algorithm to achieve maximum trade
   public void generateTrade(){
      double startingMoney = TourManager.getStartingMoney();
      money.set(0, startingMoney);
```

```
    for(int i = 0; i < tour.size(); i++){

        double maxProductProfit = 0;
        int maxProductProfitIndex = -1;
        City currentCity = (City)tour.get(i);
        City nextCity;
        int nextIndex;

        if(i < tour.size()-1) nextIndex = i+1;
        else nextIndex = 0;
        nextCity = (City)tour.get(nextIndex);

        for(int j = 0; j < TourManager.getProductVariety(); j++){
            // reset all trade
            trade.get(i).set(j, 0);
            // get max profit
            double productProfit = nextCity.getPrice()[j] - currentCity.getPrice()[j];
            if(productProfit > maxProductProfit){
                maxProductProfit = productProfit;
                maxProductProfitIndex = j;
            }
        }
        // trade if any profitable
        if(maxProductProfitIndex >= 0){
            int pr = trade.get(i).get(maxProductProfitIndex);

            Double baseCost = currentCity.getPrice()[maxProductProfitIndex];
            Double buyCalc =  (Double)money.get(i)/baseCost;
            int buyAmount = buyCalc.intValue();
            if(buyAmount > TourManager.getMaxCapacity()) buyAmount =
TourManager.getMaxCapacity(); // limit number of trade
            trade.get(i).set(maxProductProfitIndex, buyAmount);
            double nextMoney = (Double)money.get(i) + buyAmount*maxProductProfit;


            money.set(nextIndex, nextMoney);

        }else money.set(nextIndex, money.get(i));

    }

    // get final profit & reset starting money
    profit = (double)money.get(0);
    money.set(0, startingMoney);
  }

  public void swapTrade(){
    double startingMoney = TourManager.getStartingMoney();
    money.set(0, startingMoney);

    for(int i = 0; i < tour.size(); i++){

        double maxProductProfit = 0;
        int maxProductProfitIndex = -1;
```

```
        City currentCity = (City)tour.get(i);
        City nextCity;
        int nextIndex;

        if(i < tour.size()-1) nextIndex = i+1;
        else nextIndex = 0;
        nextCity = (City)tour.get(nextIndex);

        // random selection of trade
        for(int j = 0; j < TourManager.getProductVariety(); j++){
           // reset all trade
           trade.get(i).set(j, 0);
           // get max profit
           double productProfit = nextCity.getPrice()[j] - currentCity.getPrice()[j];
           if(productProfit > maxProductProfit){
              maxProductProfit = productProfit;
              maxProductProfitIndex = j;
           }
        }

        // trade if any profitable
        if(maxProductProfitIndex >= 0){
           int pr = trade.get(i).get(maxProductProfitIndex);

           Double baseCost = currentCity.getPrice()[maxProductProfitIndex];
           Double buyCalc =  (Double)money.get(i)/baseCost;
           int buyAmount = buyCalc.intValue();
           if(buyAmount > TourManager.getMaxCapacity()) buyAmount =
TourManager.getMaxCapacity(); // limit number of trade
           trade.get(i).set(maxProductProfitIndex, buyAmount);
           double nextMoney = (Double)money.get(i) + buyAmount*maxProductProfit;


           money.set(nextIndex, nextMoney);

        }else money.set(nextIndex, money.get(i));



     }

     // get final profit & reset starting money
     profit = (double)money.get(0);
     money.set(0, startingMoney);
  }

  public double getProfit(){
     return profit;
  }

  // Gets the money in the current tour position
  public double getMoney(int tourPosition){
     return (double)money.get(tourPosition);
  }
```

```
 // Gets the money in the current tour position
public List<Integer> getTrade(int tourPosition){
   return trade.get(tourPosition);
}


// Gets a city from the tour
public City getCity(int tourPosition) {
   return (City)tour.get(tourPosition);
}

// Sets a city in a certain position within a tour
public void setCity(int tourPosition, City city) {
   tour.set(tourPosition, city);
   // If the tours been altered we need to reset the fitness and distance
   fitness = 0;
   distance = 0;
}

// Gets the tours fitness
public double getFitness() {
          generateTrade();
   return profit/(double)getDistance();
}

// Gets the total distance of the tour
public int getDistance(){
   if (distance == 0) {
      int tourDistance = 0;
      // Loop through our tour's cities
      for (int cityIndex=0; cityIndex < tourSize(); cityIndex++) {
         // Get city we're travelling from
         City fromCity = getCity(cityIndex);
         // City we're travelling to
         City destinationCity;
         // Check we're not on our tour's last city, if we are set our
         // tour's final destination city to our starting city
         if(cityIndex+1 < tourSize()){
            destinationCity = getCity(cityIndex+1);
         }
         else{
            destinationCity = getCity(0);
         }
         // Get the distance between the two cities
         tourDistance += fromCity.distanceTo(destinationCity);
      }
      distance = tourDistance;
   }
   return distance;
}

// Get number of cities on our tour
public int tourSize() {
```

```
        return tour.size();
    }

    // Check if the tour contains a city
    public boolean containsCity(City city){
        return tour.contains(city);
    }

    @Override
    public String toString() {
        String geneString = "|";
        for (int i = 0; i < tourSize(); i++) {
            geneString += getCity(i)+" - "+getTrade(i)+" - $"+getMoney(i)+"|";
        }
        return geneString;
    }
}
```

```
/*
* TourManager.java
* Holds the cities of a tour
*/

package ctpga;

import java.util.ArrayList;

public class TourManager {

    // Holds our cities
    private static ArrayList destinationCities = new ArrayList<City>();
    private static int maxX = 0;
    private static int maxY = 0;

    // Customizable parameters
    public static int getCityAmount(){ return 100; }
    public static int getRandomSeed(){ return 9; }
    public static int getIteration(){ return 5000; }
    public static double getStartingMoney(){ return 50.0; }
    public static int getProductVariety(){ return 3; }
    public static int getMaxCapacity(){ return 100; }

    // Adds a destination city
    public static void addCity(City city) {
        destinationCities.add(city);
        if(city.x>maxX) maxX = city.x;
        if(city.y>maxY) maxY = city.y;
    }

    // Get a city
    public static City getCity(int index){
```

```
        return (City)destinationCities.get(index);
    }

    // Get the number of destination cities
    public static int numberOfCities(){
        return destinationCities.size();
    }

    public static int getMaxX(){ return maxX;}
    public static int getMaxY(){ return maxY;}
}
```

```
package ctpga;
/*
* Population.java
* Manages a population of candidate tours
*/

public class Population {

    // Holds population of tours
    Tour[] tours;

    // Construct a population
    public Population(int populationSize, boolean initialise) {
        tours = new Tour[populationSize];
        // If we need to initialise a population of tours do so
        if (initialise) {
            // Loop and create individuals
            for (int i = 0; i < populationSize(); i++) {
                Tour newTour = new Tour();
                newTour.generateIndividual();
                saveTour(i, newTour);
            }
        }
    }

    // Saves a tour
    public void saveTour(int index, Tour tour) {
        tours[index] = tour;
    }

    // Gets a tour from population
    public Tour getTour(int index) {
        return tours[index];
    }

    // Gets the best tour in the population
    public Tour getFittest() {
        Tour fittest = tours[0];
```

```
        // Loop through individuals to find fittest
        for (int i = 1; i < populationSize(); i++) {
            if (fittest.getFitness() <= getTour(i).getFitness()) {
                fittest = getTour(i);
            }
        }
        return fittest;
    }

    // Gets population size
    public int populationSize() {
        return tours.length;
    }
}
```

```
/*
* GA.java
* Manages algorithms for evolving population
*/

package ctpga;

public class GA {

    /* GA parameters */
    private static final double mutationRate = 0.002;
    private static final int tournamentSize = 5;
    private static final boolean elitist = true;

    private static final boolean lamarckian = true;
    private static final int lamarckianMutationSize = 5;

    // Evolve the entire population
    public static Population evolvePopulation(Population pop) {
        Population newPopulation = new Population(pop.populationSize(), false);

        // Keep our best individual if elitism is enabled
        int elitismOffset = 0;
        if (elitist) {
            newPopulation.saveTour(0, pop.getFittest());
            elitismOffset = 1;
        }

        // Crossover function
        for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {
            // Select parents
            Tour parent1, parent2;
            if(lamarckian){
                parent1 = LamarckianSelection(pop);
                parent2 = LamarckianSelection(pop);
            }
```

```
        else{
            parent1 = tournamentSelection(pop);
            parent2 = tournamentSelection(pop);
        }

        // Crossover parents
        Tour child = crossover(parent1, parent2);
        // Add child to new population
        newPopulation.saveTour(i, child);
    }

    // Mutate the new population a bit to add some new genetic material
    for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {
        mutate(newPopulation.getTour(i));
    }

    return newPopulation;
}

// Applies crossover to a set of parents and creates offspring
public static Tour crossover(Tour parent1, Tour parent2) {
    // Create new child tour
    Tour child = new Tour();

    // Get start and end sub tour positions for parent1's tour
    int startPos = (int) (Math.random() * parent1.tourSize());
    int endPos = (int) (Math.random() * parent1.tourSize());

    // Loop and add the sub tour from parent1 to our child
    for (int i = 0; i < child.tourSize(); i++) {
        // If our start position is less than the end position
        if (startPos < endPos && i > startPos && i < endPos) {
            child.setCity(i, parent1.getCity(i));
        } // If our start position is larger
        else if (startPos > endPos) {
            if (!(i < startPos && i > endPos)) {
                child.setCity(i, parent1.getCity(i));
            }
        }
    }

    // Loop through parent2's city tour
    for (int i = 0; i < parent2.tourSize(); i++) {
        // If child doesn't have the city add it
        if (!child.containsCity(parent2.getCity(i))) {
            // Loop to find a spare position in the child's tour
            for (int ii = 0; ii < child.tourSize(); ii++) {
                // Spare position found, add city
                if (child.getCity(ii) == null) {
                    child.setCity(ii, parent2.getCity(i));
                    break;
                }
            }
        }
```

```
    }
    child.generateTrade();
    return child;
}

// Mutate a tour using swap mutation
private static void mutate(Tour tour) {
    // Loop through tour cities
    for(int tourPos1=0; tourPos1 < tour.tourSize(); tourPos1++){
        // Apply mutation rate
        if(Math.random() < mutationRate){
            // Get a second random position in the tour
            int tourPos2 = (int) (tour.tourSize() * Math.random());

            // Get the cities at target position in tour
            City city1 = tour.getCity(tourPos1);
            City city2 = tour.getCity(tourPos2);

            // Swap them around
            tour.setCity(tourPos2, city1);
            tour.setCity(tourPos1, city2);

            tour.generateTrade();
        }
    }
}

// Selects candidate tour for crossover
private static Tour tournamentSelection(Population pop) {
    // Create a tournament population
    Population tournament = new Population(tournamentSize, false);
    // For each place in the tournament get a random candidate tour and
    // add it
    for (int i = 0; i < tournamentSize; i++) {
        int randomId = (int) (Math.random() * pop.populationSize());
        tournament.saveTour(i, pop.getTour(randomId));
    }
    // Get the fittest tour
    Tour fittest = tournament.getFittest();
    return fittest;
}

private static Tour LamarckianSelection(Population pop){
    // Create a pool of tours to mutate
    Population pool = new Population(lamarckianMutationSize, false);
    // Add unmutated tour to pool
    int randomId = (int) (Math.random() * pop.populationSize());
    pool.saveTour(0, pop.getTour(randomId));
    // get random pool and mutate them
    for (int i = 1; i < lamarckianMutationSize; i++) {
        randomId = (int) (Math.random() * pop.populationSize());
        Tour mutatedTour = pop.getTour(randomId);
        mutate(mutatedTour);
        pool.saveTour(i, mutatedTour);
```

```
        }
        // Get the fittest tour
        Tour fittest = pool.getFittest();
        return fittest;
    }
}
```

```
/*
* FitnessGraph.java
* A graph for visualizing the change of fitness over generations
*/

package ctpga;


import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;;
import java.util.List;
import java.util.ArrayList;

public class FitnessGraph extends JPanel {

    private List<Double> deltaFitness = new ArrayList<Double>();

    String cap ="Fitness Graph";
    final int PAD = 40;

    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
        int w = getWidth();
        int h = getHeight();
        // Draw ordinate.
        g2.draw(new Line2D.Double(PAD, PAD, PAD, h-PAD));
        // Draw abcissa.
        g2.draw(new Line2D.Double(PAD, h-PAD, w-PAD, h-PAD));
        double xInc = (double)(w - 2*PAD)/deltaFitness.size();
        double scale = (double)(h - 2*PAD)/deltaFitness.get(deltaFitness.size()-1);
        // Mark data points.
        g2.setPaint(Color.red);
        double px=PAD;
        double py=h-PAD;
        double x=0;
        double y=0;
        g2.drawString(cap, PAD, PAD/2);

        // find data points to plot
        double lastFitness = 0;
```

```
      for(int i = 0; i < deltaFitness.size(); i++) {
         x = PAD + i*xInc;
         y = h - PAD - scale*deltaFitness.get(i);
         g2.draw(new Line2D.Double(px,py,x,y));
         px=x;
         py=y;

         if(deltaFitness.get(i)>lastFitness || i==deltaFitness.size()-1){
            g2.drawString(i+"",(int)x,h-PAD+15);
            g2.drawString(deltaFitness.get(i)+"".toString(),0,(int)y);
            g2.fill(new Ellipse2D.Double(x-2, y-2, 4, 4));
         }
         lastFitness = deltaFitness.get(i);
      }

   }

   public FitnessGraph(List deltaFitness){
      super();
      this.deltaFitness = deltaFitness;
      JFrame f = new JFrame(cap);
      f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      f.add(this);
      f.setSize(800,600);
      f.setLocation(200,200);
      f.setVisible(true);
   }
}
```

```
/*
* CartesianGraph.java
* A graph for visualizing the location of cities, the tour route, city prices, and trade of the
* fittest solution
*/

package ctpga;


import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

public class CartesianGraph extends JPanel {

   Tour tour;

   String cap ="Caravan Trader Problem";
   final int PAD = 100;
```

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    int w = getWidth();
    int h = getHeight();
    // Draw ordinate.
    g2.draw(new Line2D.Double(PAD, PAD, PAD, h-PAD));
    // Draw abcissa.
    g2.draw(new Line2D.Double(PAD, h-PAD, w-PAD, h-PAD));
    double xScale = (double)(w - 2*PAD)/TourManager.getMaxX();
    double yScale = (double)(h - 2*PAD)/TourManager.getMaxY();
    g2.setPaint(Color.red);
    double px=PAD;
    double py=h-PAD;
    double x=0;
    double y=0;

    // Make drawn string of max and min and closest
    g2.drawString(TourManager.getMaxX()+"",w-PAD,h-PAD+15);
    g2.drawString(TourManager.getMaxY()+"".toString(),0,PAD);
    g2.drawString("Distance: "+tour.getDistance(),0,h-PAD+30);

    // Mark data points.
    g2.drawString("", PAD, PAD/2);
    for(int i = 0; i < tour.tourSize(); i++) {
        x = PAD + tour.getCity(i).getX()*xScale;
        y = h - PAD - yScale*tour.getCity(i).getY();
        g2.fill(new Ellipse2D.Double(x-5, y-5, 10, 10));
        px=x;
        py=y;
        // Write city details
        g2.drawString(i+1+". ("+tour.getCity(i).toString()+")",(int)x-50,(int)y+20);
        // Write purchase & money detail
        g2.drawString("$"+tour.getMoney(i)+", ("+tour.getTrade(i)+")",(int)x-50,(int)y+35);

    }

    // Make line for all points
    g2.setPaint(Color.BLUE);
    for(int i = 0; i < tour.tourSize(); i++) {
        int start;
        int end;
        if(i!=tour.tourSize()-1){ start=i; end=i+1; } //Connect final line back
        else{start=tour.tourSize()-1; end=0;}

        double x1 = PAD + tour.getCity(start).x*xScale;
        double y1 = h - PAD - yScale*tour.getCity(start).y;
        double x2 = PAD +tour.getCity(end).x*xScale;
        double y2 = h - PAD - yScale*tour.getCity(end).y;
        g2.draw(new Line2D.Double(x1,y1, x2,y2));
    }
}
```

```
   public CartesianGraph(Tour tour){
      super();
      this.tour = tour;
      JFrame f = new JFrame(cap);
      f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      f.add(this);
      f.setSize(800,600);
      f.setLocation(200,200);
      f.setVisible(true);
   }
}
```

```
/*
* ParetoGraph.java
* A graph for visualizing the spread of all solutions that achieves the profit and distance
* objectives
*/

package ctpga;


import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

public class ParetoGraph extends JPanel {

   Population pop;

   String cap ="Pareto Graph";
   final int PAD = 30;

   protected void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;
      g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                  RenderingHints.VALUE_ANTIALIAS_ON);
      int w = getWidth();
      int h = getHeight();
      // Draw ordinate.
      g2.draw(new Line2D.Double(PAD, PAD, PAD, h-PAD));
      // Draw abcissa.
      g2.draw(new Line2D.Double(PAD, h-PAD, w-PAD, h-PAD));
      g2.setPaint(Color.red);
      double px=PAD;
      double py=h-PAD;
      double x=0;
      double y=0;
```

```
      // Make drawn string of max and min and closest
      g2.drawString("Profit",w-PAD,h-PAD+15);
      g2.drawString("Distance".toString(),0,PAD);
//       g2.drawString("Distance: "+tour.getDistance(),0,h-PAD+30);

      // get max profit & min distance
      double maxProfit = 0;
      int maxDistance = 0;
      int minDistance = Integer.MAX_VALUE;
      for(int i = 0; i < pop.populationSize(); i++) {
         Tour tour = pop.tours[i];
         int distance = tour.getDistance();
         double profit = tour.getProfit();
         if(distance < minDistance) minDistance = distance;
         if(distance > maxDistance) maxDistance = distance;
         if(profit > maxProfit) maxProfit = profit;
      }
      double xScale = (double)(w - 2*PAD)/maxProfit;
      double yScale = (double)(h - 2*PAD)/maxDistance;

      int shortestDistance = Integer.MAX_VALUE;
      Tour shortestTour = null;
      // Mark data points.
      g2.drawString("", PAD, PAD/2);
      for(int i = 0; i < pop.populationSize(); i++) {
         Tour tour = pop.tours[i];
         if(tour.getDistance() < shortestDistance){
            shortestDistance = tour.getDistance();
            shortestTour = tour;
         }
         // x = profit, y = distance
         x = PAD + tour.getProfit()*xScale;
         y = h - PAD - yScale*tour.getDistance();

         // set plot point color
         float alpha = 0.2f;
         Color color = new Color(1, 0, 0, alpha); //Red
         g2.setPaint(color);
         g2.fill(new Ellipse2D.Double(w+2*PAD-x-5, 2*PAD+y-5, 5, 5));
         g2.setPaint(Color.red);

         px=x;
         py=y;
         // Color the fittest blue
//          if(tour.equals(pop.getFittest())){
//             g2.setPaint(Color.blue);
//             g2.fill(new Ellipse2D.Double(w+2*PAD-x-5, 2*PAD+y-5, 10, 10));
//             g2.setPaint(Color.red);
//          }

//          // Write city details
//          g2.drawString(i+1+". ("+tour.getCity(i).toString()+")",(int)x-50,(int)y+20);
//          // Write purchase & money detail
```

```
//            g2.drawString("$"+tour.getMoney(i)+", ("+tour.getTrade(i)+")",(int)x-50,(int)y+35);

        }
        // Color the shortest distance solution green
//            x = PAD + shortestTour.getProfit()*xScale;
//            y = h - PAD - yScale*shortestTour.getDistance();
//            g2.setPaint(Color.GREEN);
//            g2.fill(new Ellipse2D.Double(w+2*PAD-x-5, 2*PAD+y-5, 10, 10));
//            g2.setPaint(Color.red);

    }

    public ParetoGraph(Population pop){
        super();
        this.pop = pop;
        JFrame f = new JFrame(cap);
        f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        f.add(this);
        f.setSize(800,600);
        f.setLocation(200,200);
        f.setVisible(true);
    }
}
```

# Appendix B: Experiment Data

| Cities | Seed | TBLGA | Distance | Profit | Fitness | Average Fitness |
|---|---|---|---|---|---|---|
| 10 | 0 | false | 803 | 385 | 0.4794520548 | 0.4577973526 |
| 10 | 1 | false | 709 | 355 | 0.5007052186 | |
| 10 | 2 | false | 760 | 388 | 0.5105263158 | |
| 10 | 3 | false | 786 | 319 | 0.4058524173 | |
| 10 | 4 | false | 782 | 384 | 0.4910485934 | |
| 10 | 5 | false | 685 | 382 | 0.5576642336 | |
| 10 | 6 | false | 656 | 340 | 0.5182926829 | |
| 10 | 7 | false | 869 | 365 | 0.420023015 | |
| 10 | 8 | false | 897 | 317 | 0.353400223 | |
| 10 | 9 | false | 912 | 311 | 0.3410087719 | |
| 10 | 0 | true | 624 | 322 | 0.516025641 | 0.4673917309 |
| 10 | 1 | true | 672 | 336 | 0.5 | |
| 10 | 2 | true | 760 | 392 | 0.5157894737 | |
| 10 | 3 | true | 845 | 384 | 0.4544378698 | |
| 10 | 4 | true | 782 | 384 | 0.4910485934 | |
| 10 | 5 | true | 771 | 386 | 0.5006485084 | |
| 10 | 6 | true | 656 | 324 | 0.493902439 | |
| 10 | 7 | true | 708 | 313 | 0.4420903955 | |
| 10 | 8 | true | 794 | 314 | 0.395465995 | |
| 10 | 9 | true | 834 | 304 | 0.3645083933 | |
| 20 | 0 | false | 1105 | 2405 | 2.1764705882 | 1.877239659 |
| 20 | 1 | false | 1114 | 2168 | 1.9461400359 | |
| 20 | 2 | false | 1132 | 2230 | 1.9699646643 | |
| 20 | 3 | false | 1107 | 2288 | 2.0668473351 | |
| 20 | 4 | false | 1394 | 2162 | 1.5509325681 | |
| 20 | 5 | false | 1029 | 2418 | 2.3498542274 | |
| 20 | 6 | false | 1279 | 2270 | 1.7748240813 | |
| 20 | 7 | false | 1091 | 1964 | 1.8001833181 | |
| 20 | 8 | false | 1235 | 1934 | 1.5659919028 | |
| 20 | 9 | false | 1187 | 1865 | 1.5711878686 | |
| 20 | 0 | true | 1199 | 2425 | 2.0225187656 | 2.1003462085 |
| 20 | 1 | true | 859 | 2135 | 2.4854481956 | |
| 20 | 2 | true | 1128 | 2335 | 2.070035461 | |
| 20 | 3 | true | 1059 | 2093 | 1.9763928234 | |
| 20 | 4 | true | 1077 | 2162 | 2.0074280409 | |
| 20 | 5 | true | 882 | 2312 | 2.6213151927 | |
| 20 | 6 | true | 1132 | 2330 | 2.0583038869 | |
| 20 | 7 | true | 947 | 2024 | 2.1372756072 | |
| 20 | 8 | true | 928 | 1801 | 1.9407327586 | |
| 20 | 9 | true | 1057 | 1780 | 1.6840113529 | |
| 50 | 0 | false | 2006 | 9686 | 4.8285144566 | 4.7254050547 |
| 50 | 1 | false | 1838 | 9440 | 5.1360174102 | |
| 50 | 2 | false | 2147 | 9504 | 4.4266418258 | |
| 50 | 3 | false | 1722 | 8713 | 5.0598141696 | |
| 50 | 4 | false | 2184 | 9175 | 4.201007326 | |
| 50 | 5 | false | 2098 | 9431 | 4.4952335558 | |
| 50 | 6 | false | 1926 | 8923 | 4.6329179647 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| 50 | 7 | false | 1822 | 9330 | 5.1207464325 | |
| 50 | 8 | false | 1831 | 8636 | 4.7165483342 | |
| 50 | 9 | false | 1852 | 8587 | 4.6366090713 | |
| 50 | 0 | true | 1605 | 9695 | 6.0404984424 | 5.1494609327 |
| 50 | 1 | true | 1677 | 9045 | 5.3935599284 | |
| 50 | 2 | true | 1727 | 9354 | 5.416328894 | |
| 50 | 3 | true | 1946 | 9430 | 4.8458376156 | |
| 50 | 4 | true | 1781 | 8980 | 5.0421111735 | |
| 50 | 5 | true | 1993 | 9452 | 4.7425990968 | |
| 50 | 6 | true | 1729 | 8758 | 5.0653556969 | |
| 50 | 7 | true | 1733 | 9118 | 5.2613964224 | |
| 50 | 8 | true | 1663 | 8385 | 5.0420926037 | |
| 50 | 9 | true | 1847 | 8579 | 4.6448294532 | |
| 100 | 0 | false | 2903 | 21713 | 7.4795039614 | 6.6213913125 |
| 100 | 1 | false | 3296 | 22046 | 6.6887135922 | |
| 100 | 2 | false | 3346 | 22026 | 6.5827854154 | |
| 100 | 3 | false | 3590 | 22353 | 6.2264623955 | |
| 100 | 4 | false | 3381 | 22124 | 6.5436261461 | |
| 100 | 5 | false | 3411 | 21867 | 6.4107299912 | |
| 100 | 6 | false | 3197 | 22250 | 6.9596496716 | |
| 100 | 7 | false | 3375 | 21932 | 6.4983703704 | |
| 100 | 8 | false | 3237 | 21647 | 6.687364844 | |
| 100 | 9 | false | 3577 | 21951 | 6.1367067375 | |
| 100 | 0 | true | 3046 | 21761 | 7.1441234406 | 7.2188570384 |
| 100 | 1 | true | 3167 | 21827 | 6.8920113672 | |
| 100 | 2 | true | 3000 | 21609 | 7.203 | |
| 100 | 3 | true | 3078 | 21940 | 7.1280051982 | |
| 100 | 4 | true | 2785 | 21436 | 7.6969479354 | |
| 100 | 5 | true | 2942 | 22226 | 7.5547246771 | |
| 100 | 6 | true | 3063 | 22576 | 7.3705517467 | |
| 100 | 7 | true | 2972 | 21801 | 7.3354643338 | |
| 100 | 8 | true | 3217 | 22191 | 6.8980416537 | |
| 100 | 9 | true | 3207 | 22339 | 6.9657000312 | |