The Use of Genetic Algorithms with Heuristic Crossover for Efficiently Solving the Traveling Salesman Problem

Aaron Foltz
CS580 Project Report

December 7, 2011

Abstract

A salesman, given the problem of finding the shortest trip between a set of cities, visiting each city exactly once and starting where he ended, would be trying to solve the well-known Traveling Salesman Problem. Although it seems easy at the surface, this problem is one of the most difficult problems in Computer Science. Using Genetic Algorithms, a set of algorithms designated for their mimicking of natural processes, more specifically Darwinian evolution, we can make an attempt at solving this problem.

After a thorough introduction of the complex steps of Genetic Algorithms and the applications and complexity of the Traveling Salesman Problem, we will then discuss a few of the more successful techniques at solving the Traveling Salesman Problem. From this point, I will propose my own approach, taking an aggregation of previous work, including Grefenstette's heuristic crossover, with my own attempt at an alternative method of Genetic Algorithm initialization. Computational experiments and results will be provided for this approach, including a multitude of Traveling Salesman Problems ranging in size from 30 to 200 cities.

1 Introduction

1.1 Genetic Algorithms

Introduced in 1975 by John Holland, Genetic Algorithms (GAs) are a subcategory of algorithms found under the general topic of Evolutionary Algorithms (Larranaga *et al.* , 1999). The general basis for Evolutionary Algorithms is to develop algorithms which closely mimic and gather inspiration from everyday processes and mechanisms found in nature (Eiben & Smith, 2003). The hope is that these algorithms will function as successfully as the respective process found in nature, leading to a quicker, more efficient solution. From this starting point, GAs have their foundations in Darwinian evolution, simulating a natural process that has been occurring successfully for millions of years.

GAs can be seen as solution search algorithms, wandering around through complete solutions, more commonly known as solution space. Looking at Algorithm 1 below, we can see that GAs have several different steps which can be generalized into the following processes:

- Encoding
- Initialization
- Evaluation
- Selection
- Crossover
- Mutation

Algorithm 1 Simple Genetic Algorithm

```
Create initial population of solutions
Repeat
        Evaluate each individual in the population (fitness function)
        Select some individuals for crossover (based on the evaluation)
        Crossover those individuals to get offspring
        Mutate select individuals
Until some stopping point
```

1.1.1 Encoding

Before running a Genetic Algorithm, one must first come up with an appropriate encoding of the real world problem into something that is usable in the Genetic Algorithm. Not only must this encoding adequately represent the real world situation, but it must also be easily used and manipulated in the crossover process. Many tend to use a simple character string representation, encoding the information into zeros and ones, but more complicated encodings are possible(Bryant, 2000). Each separate character in this string can be referred to as a *gene* and its value can be referred to as an *allele*.

Problem: The Traveling Salesman Problem with the following city set: {Herndon, Fairfax, Chantilly, Oakton}

A sample encoding from the real world to a Genetic Algorithm chromosome could be "12341", with Herndon = 1, Fairfax = 2, etc...

This sample encoding could represent the order in which the cities are visited in a typical Traveling Salesman Problem.

Figure 1.1: A sample encoding

1.1.2 Initialization

To begin, the Genetic Algorithm must first be set up with an initial *population* of encoded solutions, referred to as *individuals* or *chromosomes*. Each of these chromosomes will have a set of genes determined by the previous encoding process. One should always be careful in choosing the number of individuals in their initial population as the population will not change during selection and crossover. Too few individuals may cause the Genetic Algorithm to never reach its stopping point, but on the other hand, too many individuals may be an unneeded tax and waste of processing capabilities. The majority of the time this initial population can be randomly selected, but as we will see later, other heuristics may be used in order to inject knowledge into this process.

1.1.3 Evaluation

The start of this inner loop signifies the start of a new *generation*, and the Genetic Algorithm must first receive a rating for each of the individuals in the population. In order to do this, you must provide a domain specific *fitness* *function* which should return higher values for more "fit" individuals leading to a better solution. These fitness values will end up being used in the selection step to decide if a particular individual will be selected to reproduce and create offspring for the next generation (Bryant, 2000).

1.1.4 Selection

Based on the previous step's evaluation of the fitness function, individuals can now be chosen to reproduce. For this step, a probability can be used in correlation with the fitness function, or a simple culling method can be used to discard any individuals with a fitness evaluation below a certain threshold (Russell & Norvig, 2010). No matter the method, the individuals must be paired in groups of two so that they may reproduce and keep a constant population size.

1.1.5 Crossover

Crossover is the main operator of the Genetic Algorithm, and can be thought of as typical sexual reproduction. It allows two fit parents to produce *offspring* which inherit genes partly from each parent in an attempt to increase the overall quality of the population (Larranaga *et al.* , 1999). Before doing so, we must first select the *crossover point(s)* for each parent. These crossover points acts as the switching points of the genes for both parents, essentially creating the genetic makeup for both of the children. As you can see in FIGURE 1.2, each parent represents a corresponding child (since the size of the population is constant), and each child inherits a portion of their parents genes depending on the location of the crossover point. Once a parent has finished its crossover, it can then replaced by its corresponding child.

This is the point in which you can really see if the encoding worked out. Take the Traveling Salesman Problem mentioned in FIGURE 1.1 as an example. This problem will be covered more in detail later, but the idea is to visit each city exactly once, starting and ending in the same city. This means that a single crossover between two candidate solutions could possibly create an invalid solution, so the appropriate monitors must be in place to check for this domain specific problem.

1. Two parents selected for crossover:
   $c_1 = 1111$
   $c_2 = 0110$

2. After selection of the crossover point:
   $c_1 = 11 \mid 11$
   $c_2 = 01 \mid 10$

3. Two children after crossover:
   $c_1' = 1110$
   $c_2' = 0111$

Figure 1.2: Simple crossover point and crossover

1.1.6 Mutation

Once we have the newly produced children, we can now do a probabilistic mutation on some of the individuals in order to avoid being trapped in a local optima (Bryant, 2000). The easiest mutation revolves around a simply binary string encoding, and requires that you change a single gene in the chromosome. For instance, if the following chromosome was probabilistically chosen with a mutation point at gene 0:

$c_1 = 1011$

It would simply become:

$c_1' = 0011$

Much like in the crossover section above, this can lead to illegal chromosomes and must be dealt with. One manner of dealing with this is inversion and segment inversion. Instead of merely changing a single gene, we can choose to switch two genes or groups of genes within the same chromosome(Bryant, 2000). As long as the children being produced during crossover are legal, this method guarantees that the mutation will retain the chromosome's legality .

1.2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is an easily explained but difficult problem found in theoretical Computer Science. The basis of the problem is to find a tour of minimal cost given a set a cities, visiting each city exactly once, and starting and ending in the same city.

1.2.1 Variations on the Problem

Before moving into the history of the TSP, it would first be advantageous to discuss the different variations of the problem found in academia. The first, and probably most popular form, is the Symmetric TSP. This form of the TSP allows for an equal bidirectional cost between each pair of cities, i.e., the cost of traveling from city i to city j is equal to the cost of traveling from city j to city i. The second, more difficult form, is the Asymmetric TSP. It allows for more leniency than the Symmetric TSP as it does not require bidirectional equality between any pair of cities, i.e, the cost of traveling from city i to city j does not have to equal the cost of traveling from city j to city i. Other less well known variations stemming from these forms include the Multisalesmen TSP, the Bottleneck TSP, and the Time-Dependent TSP. The Multisalesmen problem is a simple adaptation of the first two forms, and only requires that multiple salesmen be involved. The objective is to have the salesmen, as a group, finish the TSP tour before returning back to each of their originating cities. The Bottleneck TSP is a completely different take on the traditional TSP, instead focusing on minimizing the *largest* edge cost rather than the complete tour cost. Finally, the Time-Dependent TSP introduces another variable into the mix, taking the cost of traveling from city i to city j to be a function of the time period at that moment (Bryant, 2000).

1.2.2 History of the Traveling Salesman Problem

This problem first appeared under a different name in the mid 18th century at the hands of Euler. At the time, he was working to solve the "knights tour problem" in chess, which consisted of finding a tour where each knight would visit each square exactly once before returning to the starting square (Larranaga *et al.* , 1999).

The TSP first got its name in 1932 in a German book written by none other than a veteran traveling salesman. From this point on, the Research and Development Corporation (RAND) would pick up the problem in 1948, propelling it to the popularity that we see today (Larranaga *et al.* , 1999). Both Researchers and academia alike have followed this popularity stemming from RAND's introduction of this problem for a few main reasons: its computational complexity, and its application to other problems.

1.2.3 Complexity of the Traveling Salesman Problem

The main push for research on this problem seems to mostly deal with its computational complexity. In 1972, Richard Karp proved that the Hamiltonian Cycle Problem, a problem that the TSP is based on, is NP-Complete. With their close-knit relationship, this implies that the TSP is NP-Hard, or part of the hardest subset of non-deterministic polynomial problems in NP-Complete (Maredia, 2010). For the TSP, this simply means that there is no optimally known algorithm to solve it polynomial time. However, there still may be an algorithm that can quickly *guess* a suboptimal solution and verify its correctness within polynomial time (Russell & Norvig, 2010). As a side effect, this also means that the the addition of a single city will change the TSP time complexity exponentially, leading to solving a TSP with moderate city size to be all but infeasible through an exhaustive search. Take for example a set of cities of size 7. With the TSP's concretely structured search space, we can easily see that the running time for an exhaustive search would be *O(n!)*, or 5,040 the example set. By bumping that set up to 8 cities, we now see that the running time would be 40,320, a massive explosion just from adding a single city into the mix.

1.2.4 Applications of the Traveling Salesman Problem

The second reason for the popularity of this problem stems from its applicability to other issues. First and foremost, the TSP has a direct correlation to any type of routing problem where the cost is a function of the total distance. Take for example a series of food deliveries, equipment drop-offs, car pools, or bus stops. Each of these can depend heavily on the overall length of their trip, and will benefit greatly from an optimal or near optimal analysis using the TSP.

A second application of the TSP can be found in computer wiring, but the applicability can be more generalized to other sorts of wiring as well. The basis of the problem is that hardware modules have a given set of pins which need to be connected by at most a single wire. In order to optimize the cost and length of the wiring, we can view this problem as a formulation of the TSP with the pins being the cities and the cost being the distance between any pair of connectable pins (Lenstra & Kan, 1975).

1.3 Motivation

The main motivation for this study, much like many studies before it, comes from the computational complexity of the Traveling Salesman Problem. It's quite amazing how such a rather simple problem can be so computationally taxing, and it seems all the worthwhile to find an efficient solution given its many applications to seemingly unrelated problems.

Furthermore, the pairing of heuristics with Genetic Algorithms to solve the Traveling Salesman Problem seems rather lacking at best. As will be discussed later, there have been a handful of attempts at implementing heuristic crossovers, but that's about as far as the "knowledge" injection goes (Baraglia *et al.* , 2001; Lin & Kernighan, 1973; Montemanni *et al.* , 2007; Nilsson, 2003; Boyaci, 2007). As noted by Bryant, Genetic Algorithms with heuristic crossover seem to be a fairly efficient way of approximating the Traveling Salesman Problem, but I have yet to see the detailed effects of the pairing with heuristic-based mutations (Bryant, 2000). I also have yet to see any studies done on the use of heuristics in the last area that can make use of them, the initialization of the population, so that seems to be a welcomed area of investigation.

2 Background

2.1 Previous Attempts

2.1.1 Encoding

The first, and most popular approach to encoding the TSP is *tour* or *path notation* in the form of a character string. Here, a tour of size n is listed as

$$\mathrm{c} = \mathrm{g}_1\mathrm{g}_2 \cdots \mathrm{g}_n\mathrm{g}_1$$

with each city being visited in the order shown (Bryant, 2000; Larranaga *et al.* , 1999).

Another similar approach is called *cycle* or *adjacency notation*, and is a flat representation of an adjacency matrix. Take for example the chromosome representation of a TSP starting at city 1

$$\mathrm{c} = 3421$$

which encodes the tour

1 - 3 - 2 - 4 - 1

The basis of this representation is as follows: If city i is listed in position j, then the edge connecting city i and city j is included in the tour (Larranaga *et al.* , 1999). In any case, the starting city does not actually matter. For the example above, we could have as easily have chosen to start at city 3 giving a tour of

3 - 2 - 4 - 1 - 3

and we would have ended up with the exact same total cost.

The final approach to encoding the TSP takes a similar, but entirely different structural approach from the character string representation, and instead works with a simple binary matrix. Again, the meaning of the representation is an adjacency matrix, but in a different manner with this approach. Simply, if there is a 1 in location (i, j), then there is an edge between city i to city j (Homaifar *et al.* , 1992; Bryant, 2000; Larranaga *et al.* , 1999). For instance, the matrices

$$\begin{array}{cc|c|c|c|c|}
 & & \multicolumn{4}{c}{j} \\
 & & 1 & 2 & 3 & 4 \\ \cline{3-6}
 & 1 & 0 & 0 & 1 & 0 \\ \cline{3-6}
 & 2 & 0 & 0 & 0 & 1 \\ \cline{3-6}
i & 3 & 0 & 1 & 0 & 0 \\ \cline{3-6}
 & 4 & 1 & 0 & 0 & 0 \\ \cline{3-6}
\end{array}$$

represents the tour

1 - 3 - 2 - 4 - 1

and mimics the adjacency notation example above.

Several other less well known representations have been attempted and can be found in (Larranaga *et al.* , 1999).

2.1.2 Crossover

Although there have been a plethora of different crossover attempts for the TSP, we will only describe a fairly small subset of the most popular and efficient dealing with tour notation. A more thorough listing can be found in (Larranaga *et al.* , 1999).

Starting with the basic tour notation, one of the easiest, and most popular, seems to be *partially mapped crossover* (PMX). Contrary to the example noted in SECTION 1.1.5, we must first randomly select two crossover points for the crossover. The genes between each parent's crossover points are then switched in order to produce the genetic makeup of both children. Looking at FIGURE 2.1 PART 3, you can see that such a simple switch can cause the situation discussed in SECTION 1.1.5, resulting in an invalid chromosome in TSP terms. To combat this, PMX requires that you keep the relationship correlations between the swapped genes, i.e., $3 \Leftrightarrow 5$ and $4 \Leftrightarrow 2$. These relationships are then used in FIGURE 2.1 PART 3 to replace all applicable numbers outside of each parents crossover points (Bryant, 2000; Larranaga *et al.* , 1999).

1. Two parents selected for crossover:
   $c_1 = 123456$
   $c_2 = 345216$

2. After selection of the crossover points:
   $c_1 = 12 \mid 34 \mid 56$
   $c_2 = 34 \mid 52 \mid 16$

3. Two children after crossover:
   $c_1{}' = 12 \mid 52 \mid 56$
   $c_2{}' = 34 \mid 34 \mid 16$

4. Repairing the illegal children:
   $c_1{}'' = 14 \mid 52 \mid 36$
   $c_2{}'' = 52 \mid 34 \mid 16$

Figure 2.1: Partially mapped crossover

A similar technique to PMX is *order crossover* (OX). The technique seen in FIGURE 2.2 is the exact same as PMX until PART 4. At that point, instead of worrying about the relationships, we instead take note of the order of the genes of each parent starting immediately after the second crossover point. In step 5, for each parental gene ordering, we then delete the numbers found between the crossover points in its corresponding child. We then finish the children chromosomes by filling it in with the corresponding parent's gene ordering, starting after the second crossover point (Bryant, 2000; Larranaga *et al.* , 1999).

1. Two parents selected for crossover:
   c_1 = 123456
   c_2 = 345216

2. After selection of the crossover points:
   c_1 = 12 | 34 | 56
   c_2 = 34 | 52 | 16

3. Two children after crossover:
   $c_1{}'$ = _ _ | 52 | _ _
   $c_2{}'$ = _ _ | 34 | _ _

4. Parent gene orders:
   c_1 = 561234
   c_2= 163452

5. Parent gene orders with removal of crossover genes in children:
   c_1 = 6134
   c_2= 1652

6. After filling in the child with the parents ordering:
   $c_1{}'$ = 345261
   $c_2{}'$ = 523416

Figure 2.2: Ordered Crossover

A third technique works again under the basic tour notation, and is called *cycle crossover* (CX). The basic requirement of this technique is that each gene in the children must be in the same position as in one of the parents producing it. Keeping this in mind, we start in FIGURE 2.3 PART 2 by focusing on a single child and choosing the first gene from one of the parents to fill position 1. If we select 1 from c_1, we know that 2 cannot be in position 1, so we must place it in position 4 instead. Fixing element 2 in position 4 does not force any element since 1 has already been placed. We must now choose an element for position 2. Say we select 4 from c_2, then know that 3 cannot go there, so we must place it in position 3. After finishing c_1' , c_2' is as easy as choosing the numbers from the parents that were not used in same position in c_1' (Bryant, 2000).

1. Two parents selected for crossover:
   $c_1 = 1342$
   $c_2 = 2431$

2. Choosing the element for position 1:
   $c_1' = 1$ _ _ _

3. Choosing 1 forces 2 into position 4:
   $c_1' = 1$ _ _ 2

4. Choosing the element for position 2:
   $c_1' = 1$ 4 _ 2

5. Choosing 4 forces 3 into position 3:
   $c_1' = 1\ 4\ 3\ 2$

6. After cycle crossover:
   $c_1' = 1432$
   $c_2' = 2341$

Figure 2.3: Cycle crossover

On the notion of using heuristics in the crossover, the most well documented example seems to be *Grefenstette's heuristic crossover*. To construct a child tour, one first starts by choosing a random starting city from either of the parent's tours. Upon examining the two edges leaving that city in the parent's tours, choose to extend the child's tour with the shortest edge which does not introduce a cycle. If both of these edges cause a cycle, then we extend it by a random edge. This process repeats from the child's new city until a complete tour has been produced (Jog *et al.* , 1991; Kureichick *et al.* , 1996; Larranaga *et al.* , 1999; Bryant, 2000). Although a rather simple algorithm, this method makes use of the domain specific knowledge in the form of the distances between cities. According to Bryant, this inclusion of knowledge has proven itself and has propelled the heuristic crossover to be one of the absolute best. If it doesn't produce the optimal solution for a problem, it almost always returns a solution not too far off .

A second use of heuristics extends the heuristic crossover, and is called *Brady's crossover*. Rather than focusing on single edges, this approach instead analyzes partial paths, or subsets of a complete tour. To begin, this crossover checks both parents for a subset of the tour containing the same cities visited in a different order. The children would then inherit their parent's tour, substituting the shorter of the two tours for the longer one (Jog *et al.* , 1991).

Finally, a third use of heuristics can be found in the extremely complex algorithm, the *Lin-Kernighan crossover*. Although I will not go into the nuts and bolts of this extremely complex algorithm, I mention it for its superior TSP solving power. The Lin-Kernighan crossover has been known to solve TSP instance of up to 7,397 cities to known optimality. Furthermore, this crossover technique has improved many of the best known solutions for several large scale problems with unknown solutions, among them being a massive 85,900 city TSP (Helsgaun, 2000).

2.1.3 Mutation

Besides the simple binary mutation given in SECTION 1.1.6, there are a few other mainstream techniques in use. First, we have both the *2-opt* and *3-opt* mutations. 3-opt mutation begins by randomly selecting three edges (g_1, g_2), (g_3, g_4), and (g_5, g_6) from the tour. If it is the case that

$$Distance(g_1, g_2) + Distance(g_3, g_4) + Distance(g_5, g_6) > Distance(g_1, g_3) + Distance(g_2, g_5) + Distance(g_4, g_6)$$

then we can replaces the edges (g_1, g_2), (g_3, g_4), and (g_5, g_6) with (g_1, g_3), (g_2, g_5), and (g_4, g_6) (Bryant, 2000). The 2-opt mutation uses the same technique to compare and replaces two edges instead of three, and is discussed more in depth in SECTION 3.1.

A third mutation technique is *displacement* mutation. This technique is a very simple one to follow, and only requires that you remove a subset of the tour and insert it into a random position in the same individual. For example, taking the subtour "345" from "1234567" and placing it after city 7 would result in "1267345". Closely related to this method is *inversion* mutation, which is the exact same as displacement mutation except that the subtour is randomly inserted in reverse order(Larranaga *et al.* , 1999).

3 Study Overview

3.1 Proposed Approach

The proposed approach of this study will be multi-dimensional, intermingling many heuristics from different steps of the Genetic Algorithm. The hope is that a Genetic Algorithm with such coordination between the heuristics of each of these steps will outperform some of the already completed studies of GAs with a heuristic in only a single step.

Starting with encoding, I plan to use a simple *tour notation* such as

$$\mathrm{c} = \mathrm{g}_1\mathrm{g}_2 \dots \mathrm{g}_n\mathrm{g}_1$$

to represent a tour that visits each g_k in the corresponding order shown. In this case, each g_k could be a positive integer, much like we saw in FIGURE 1.1 (Goldberg, 1989).

For initialization, I propose a completely different approach from any of the other studies that I have found. Instead of merely taking a random selection of individuals to place in the population, I intend to build up each chromosome using a basic distance heuristic. This heuristic will be implemented in a basic *Stochastic Hill Climbing* approach, adding the first random edge whose cost is below a certain threshold or function of the remaining edge costs. For this study, that function will be the average of the remaining edges in the pool of unused cities. If the random chosen edge has cost less than that average, then we will choose it automatically, otherwise, we continue choosing random edges until the requirement is satisfied (See ALGORITHM 2).

Evaluation will simply entail a summation of the current edges for that individuals tour. The individuals in the population can then be ranked in ascending order before being selected. At selection, we will do culling, discarding those individuals in the bottom 25^{th} percentile of the population. Pairing between the remainder of the fittest individuals will be completely random, making sure to complete extra pairings to match the number of individuals discarded.

The main operator of the Genetic Algorithm, crossover, will be implemented using *Grefenstette's heuristic crossover* first mentioned in SECTION 2.1. The basis of the crossover is to first choose a random starting city from the parents tour, placing that as the starting point of the corresponding child's tour. For each child, compare the two edges leaving the city in its corresponding parent, and choose for inclusion in the child's tour, the shorter edge that does not create a cycle. If both edges create a cycle, we can instead extend the child's tour by a random edge from the parent. We can then continue comparing edges and extending the child's tour until a complete tour has been found (Jog *et al.* , 1991).

For mutations, I propose implementing the 2-opt segment swap heuristic in order to take advantage of edge cost. The processing of this mutation begins with the selection of two random edges (g_1, g_2) and (g_3, g_4). If it is the case that

$$Distance(g_1, g_2) + Distance(g_3, g_4) > Distance(g_1, g_4) + Distance(g_2, g_3)$$

then we can remove edges (g_1, g_2) and (g_3, g_4) from the tour and replace them with edges (g_1, g_4) and (g_2, g_3) (Jog *et al.* , 1991).

3.2 Proposed Experiments

The experiments needed for this study will be fairly straightforward. Using the TSP datasets given by both TSPLIB at the Universität Heidelberg and the National Traveling Salesman Problems at Georgia Tech, we can do adequate experimentation with my proposed approach . Both of these datasets offer a vast range TSP sizes, so the number of experiments can basically be limitless. In the essence of being punctual, I plan to use my Genetic Algorithm to experiment on a basic subset of this massive data collection, going through problems of size 30, 50, 100, and 200 (gte, n.d.; uni, n.d.). As a subset of this initial experiments, I also propose a set of experiments based solely on the Genetic Algorithm's parameters. Such parameters will be: population size, number of generations, mutation rate, and culling rate. Each of these rates can be tweaked, either in a independent or dependent matter, to see the effect on TSPs of varying size.

3.3 Evaluating the Results

To evaluate these results, we can do a few specific things. Firstly, since my Genetic Algorithm will be developed in Java, we can use a simplistic timer to keep track of the running time of each problem instance.

Second, and most importantly, both datasets being used happen to have optimal tour solutions for most problem instances. Once finished, this optimal tour length can be compared to the fitness evaluation of the most fit individual remaining in the Genetic Algorithm's population. For example, a good metric for NP-Hard problems and heuristics alike was given by Bryant, and simply involves the percentage difference between the fitness values of the most fit individual and the optimal individual, given by:

$$100 * ((optimal - best)/optimal)$$

This will allows us to make a critical analysis over the entire trend of proposed experiments, and will enable us to see just how well my proposed approach worked in comparison to the proven optima.

As a third option, we can also collect and analyze best-so-far data for each TSP. Instead of being merely being collected at the end of the TSP instance, this option requires direct data collection after the crossover and mutation of each GA generation. This allows us to collect the most fit individual over time, allowing us to see just how quickly and how well the algorithm converges to a solution.

3.4 Goals

The goals of this experiment are extremely simplistic. Being my first attempt at Genetic Algorithms in general, I would be elated to even compute a near optimal solution for a decently sized problem using my proposed approach. Going even further, I would like to see this algorithm contend with some of the best previous attempts at a solution. As mentioned before, Bryant found Grefenstette's heuristic crossover to work fairly well overall compared to many other standard crossover attempts, including a few of those discussed in SECTION 2.1.2. According to Bryant's discussion, my mixed heuristic approach should be close, if not optimal, so I expect my approach to be well within 10% of the optimum(Bryant, 2000). Extending this, it is also a big goal of mine to find that the inclusion and mixing of different heuristics through the initialization, crossover, and mutation steps allows the heuristic crossover to work more efficiently, thus causing the entire algorithm to trend towards better solutions.

Despite my goals, I expect that as a side effect of my mixed-bag design decision, my GA will be extremely sensitive to change. Not only are there dependencies between various parameters of the GA (mutation rate, population size, etc...), but there are serious dependencies between the crossover and mutation operators. Since my proposed crossover and mutation operator both involve TSP specific heuristics, I would not be surprised if the grouping of the two causes my GA to become brittle at times. This operator interdependency will only exacerbate the already prevalent troubles caused by the dependent GA parameters.

4 Results

4.1 Experiments

These experiments were all completed using my proposed approach as mentioned in SECTION 3.1, in conjunction with the following set of constant Genetic Algorithm parameters:

- Culling Percentage: 75%
- Maximum Evolutions: $(log(1 - \sqrt[n]{.99})/log\frac{n-3}{n-1})$ (Rintala, 1996)
- Population Size: $(log(1 - \sqrt[n]{.99})/log\frac{n-3}{n-1})$ (Rintala, 1996)
- Mutation Rate: 33%

The first set of experiments were done solely to test how well my base Genetic Algorithm is functioning with TSP problems of size 29, 48, 101. No formal experiments for the TSP of size 280 will be shown here as I had no access to computing resources that could adequately complete a meaningful testing suite.

Starting with the TSP of size 29 and 48, you can see the results from 100 TSP instances in FIGURE 6.1 and FIGURE 6.2. These results show the visual comparison between the TSP solution and the optimal solution for each of those problems. For both experiments, the solutions were all clustered within 7 and 18 percent of the optimal solution. The same can be seen in FIGURE 6.3 for the TSP of size 101, except that a pool of 30 TSP problems were used in order to be timely. Its results are expectedly worse, showing that the solutions are all clearly within 10 and 22 percent of the optimal solution.

The raw summary data for each of these three problem sizes can also be seen in TABLE 1, TABLE 2, and TABLE 3, respectively. From such tables, we can see the correspondence to the values in its respective graph as well as the average running time for each problem size. As you can see, the average running time for 29 cities was just under $\frac{1}{2}$ second, and the average running time for 48 cities was almost 4 seconds. This seems like a manageable waiting time to receive a suboptimal solution, especially compared to the 67 seconds needed on average for the 101 city problem.

The third general evaluation can be seen in FIGURE 6.4, FIGURE 6.5, and FIGURE 6.6, and comprises the best-so-far curves for five instances from each of the three problem sizes. As you can see, each of these three problem sizes seem to converge quite quickly, getting to the best solution in no more than 20, 40, and 60 generations, respectively. Anything after that generation is visualized as a plateau, representing the Genetic Algorithm finding a global optimum, or in many cases, a local optimum.

The second set of experiments were done in order to test the effect of different parameters and operators with respect to my basic proposal. First and foremost, FIGURE 6.7 and FIGURE 6.8 show the effect of experimentation with a different mutation operator than the one proposed, the Simple Swapping Mutation operator. The basic notion is to randomly swap the placement of cities in an individual, taking no notion of distance whatsoever. In FIGURE 6.7, you can see that in general, the 100 TSP problems cluster to within 16 and 29 percent of the optimal solution when using this mutation method. This makes it almost twice as bad as the 7 to 18 percent of my proposed attempt, and can be visualized in FIGURE 6.8.

An experiment was also completed in order to compare my proposed stochastic initialization technique with a technique that generates an individual in the population at random. As you can see in TABLE 4, both of the techniques were rather similar when compared over 1,000 TSP problem instances. Although the stochastic population initialization seems to win out by a mere half percent on average, it ends up losing ground in terms of average running time, coming in at almost two-hundredths of a second slower. Since the stochastic method takes many more CPU cycles calculating distances and averages, as well as finding a city that satisfies the requirements, I'm amazed that this number isn't worse.

The next grouping of experiments all centered around Genetic Algorithm parameter tweaking, checking to see which parameter values worked the best with my proposed approach, independently of the other parameters. Starting with mutation rates in TABLE 5, you can see that a mutation rate near 50% (probably much more than the actual rate in nature) is best, with rates above or below that becoming worse and worse on average. Moving to the maximum number of evolutions allowed, TABLE 6 shows that none of the chosen experiments really win out in terms of a better average fitness. Instead, they show a clear difference in the running time of each overall TSP problem. When using

$$i * ((log(1 - \sqrt[n]{.99})/log\tfrac{n-3}{n-1}))$$ (Rintala, 1996)

to compute the amount of evolutions allowed, we see that choosing $i = .5$ corresponds to the lowest running time per TSP, .2 seconds, compared to the 2 seconds needed when choosing $i = 5$.

Almost the exact same conclusion can be derived from the choosing of population size, which uses the same formula as above, except that the average fitness is actually affected. Looking at Table 7, one can see that although choosing $i = .5$ lowers the average running time to just .121 seconds, it also worsens the average fitness by nearly 2%. The winner when taking both of these metrics into consideration seems to be $i = 3$, having the best fitness evaluation on average, as well as having an acceptable average running time of .674 seconds. Choosing an i above or below that seems to have a negative effect on the average fitness values, so the only advantage would be a gain in speed.

Finally, the culling percentage seen in Table 8 is rather easy to interpret when used with my proposal. Culling percentages that partake in elitism, allowing only a small percentage (1 to 10%) of the most fit individuals to move on to the next generation, seem to do much worse on average. Once you get into the 25-75% culling range, the average fitness differences are indistinguishable, so anything within that range would seem appropriate for my approach.

4.2 Interpretation of the Results

Based on the results in the previous subsection, you can see that my approach definitely could have been more finely tuned. My proposed 33% mutation rate was quite a ways off of the best of 50%, and I think that realization can be seen best in the best-so-far graphs of Table 4, Table 5, and Table 6. Because of the low mutation rate, the Genetic Algorithm seems to converge quickly to a solution, reaching a local optimum that it then gets stuck at. With a higher mutation rate, I feel that my proposed algorithm would probably work marginally better, hopefully allowing for individuals to be bumped out of a local optimum, giving them a new chance to find the global optimum.

The results from the maximum number of evolutions is also shocking, showing that the implemented formula can be cut in half without actually harming the outcome of the average or best individual in the population. This can again be seen in Table 4, Table 5, and Table 6, showing that the Genetic Algorithm is doing almost all of its work within the first 20 or 30 percent of the generations before plateauing and converging to a solution. This means that further experimentation should uncover that setting i to a value as low as .2 or .3 will only help the running time of the algorithm without actually harming the outcome in terms of average fitness.

From the experiments of the last section, one can also see that my drastic stochastic initialization was a failure. It ended up not really improving the average fitness whatsoever, and it really only added to the already immense number of CPU cycles required of a Genetic Algorithm. Based on the discoveries from the best-so-far experiments, I feel quite certain that the heuristic crossover and mutation are already working well enough, converging to a *decent* solution quickly. By including a third layer of heuristics and complexity at the initialization level we really are not helping the crossover and mutation process in any way, we are only adding unneeded CPU cycles and ultimately wasting time.

Contrary to this, the second layer of heuristics at the mutation level seems to be well worthwhile. Without adding too much complexity or too many CPU cycles, the heuristic-based 2-Opt Mutation operator seems to better the average fitness by about half compared to a non-heuristic mutation operator. This outcome makes perfect sense in the grand scheme of things, the 2-Opt Mutation only mutates an individual to make them better fit, whereas the Simple Swapping Mutation can worsen an individual's fitness. This outcome also correlates with the mutation rate results, as allowing for a higher percentage of individuals to be mutated will enable a better average fitness when the mutations can only help an individual.

5 Conclusions and Further Research

In conclusion, we can see that my proposed approach is lacking something, it's just that this something is clouded behind the complexity of the intermingling parts of the Genetic Algorithm. The stochastic initialization was definitely a flop and can safely be removed from the Genetic Algorithm without causing any harm, and only improving the running time. Although my Genetic Algorithm parameters are fairly close to optimal choices on their own, the algorithm as a whole is still lacking in its optimality of solutions. According to Bryant, this algorithm should be optimal, and if not optimal, very close to it(Bryant, 2000). My results are far from that, producing average solutions in the range of 7 and 18 percent for small-citied instances, and 10 to 22 percent larger instances. Even though these numbers aren't totally horrible, they are still fairly far off from the 10% upper bound that I was hoping for and expecting.

The optimality isn't the only issue, as a main problem of mine was actually having the time to test the 101 and 280 city instances. Once you get up to this many cities, the running time of each TSP absolutely explodes, taking well over a minute for a TSP of size 101, and well over five minutes for a TSP of size 280. In order for this algorithm to be anywhere near practical these running times will have to be brought down greatly. Although some of the running time can be attributed to the Genetic Algorithm and the proposed approach in general, some of it can also be attributed to the evolutionary package used at the base of my approach, the Java Genetic Algorithms Package. Since this is a widely used package its classes are extremely intricate and easily extendable, allowing for the use of many different types of evolutionary algorithms at the cost of its speed. In the future, it may be a better idea to experiment on a platform designed from scratch with a single purpose in mind, allowing for a true calculation of speed and a chance at running the larger TSPs on a typical personal computer.

For further research, I greatly welcome the chance to dive deeper into the analysis of the parameters used in my proposed Genetic Algorithm. As explained in SECTION 3.4, the interplay and the interdependencies between the various parameters are what actually drives a Genetic Algorithm. With even one of these parameters marginally off, the algorithm will not run as expected, so this seems like an area of needed research.

Furthermore, I would enjoy researching the effects and differences of using varying crossover and mutation operators. Instead of sticking solely with Grefenstette's heuristic crossover, it would be nice to experiment with other heuristic and non-heuristic methods, comparing exactly how they work with respect to the average fitness. Although I do not foresee different heuristic mutation operators making much of a difference, the area may still be worth investigating, especially when pairing with various heuristic and non-heuristic crossovers.

Finally, I would like to put further effort and time into the research of Grefenstette's heuristic crossover, especially pertaining to my proposed method. Since the promises of this algorithm were so high, I feel quite incomplete leaving it at just 7 to 18 percent optimality. As expressed above, I would definitely choose to discard the pluggable evolutionary package, vowing to create a problem specific engine in order to have a better grasp on the heuristic crossover and mutation as well as the more inner workings of the Genetic Algorithm.

References

National Traveling Salesman Problems. http://www.tsp.gatech.edu/world/countries.html.

TSPLIB. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

Baraglia, R., Hidalgo, J.I., & Perego, R. 2001. A hybrid heuristic for the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, **5**(6), 613 –622.

Boyaci, Arman. 2007. Traveling Salesman Problem Heuristics.

Bryant, K. 2000. Genetic algorithms and the traveling salesman problem. *Department of Mathematics, Harvery Mudd College*.

Eiben, A., & Smith, J. 2003. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer.

Goldberg, D.E. 1989. *Genetic algorithms in search, optimization, and machine learning*. Addison-wesley.

Helsgaun, K. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, **126**(1), 106–130.

Homaifar, A., Guan, S., & Liepins, G.E. 1992. Schema analysis of the traveling salesman problem using genetic algorithms. *Complex Systems*, **6**(6), 533–552.

Jog, P., Suh, J.Y., & Van Gucht, D. 1991. Parallel genetic algorithms applied to the traveling salesman problem. *SIAM Journal on Optimization*, **1**, 515.

Johnson, D.S., & McGeoch, L.A. 1997. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 215–310.

Kureichick, V.M., Miagkikh, V.V., & Topchy, A.P. 1996. Genetic algorithm for solution of the traveling salesman problem with new features against premature convergence. *ECDC-96, Plymouth, UK*.

Larranaga, P., Kuijpers, C.M.H., Murga, R.H., Inza, I., & Dizdarevic, S. 1999. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, **13**(2), 129–170.

Lenstra, J. K., & Kan, A. H. G. Rinnooy. 1975. Some Simple Applications of the Travelling Salesman Problem. *Operational Research Quarterly (1970-1977)*, **26**(4), pp. 717–733.

Lin, S., & Kernighan, B. W. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, **21**(2), pp. 498–516.

Maredia, A. 2010. *History, Analysis, and Implementation of Traveling Salesman Problem (TSP) and Related Problems.* Ph.D. thesis, University of Houston.

Montemanni, R., Barta, J., Mastrolilli, M., & Gambardella, LM. 2007. Heuristic algorithms for the robust traveling salesman problem with interval data. *In: Proceedings of TRISTAN VI-The Sixth Triennial Symposium on Transportation Analysis, Phuket, Thailand.*

Nilsson, C. 2003. Heuristics for the traveling salesman problem. *Department of Computer Science, Linkoping University.*

Renaud, J., Boctor, F.F., & Laporte, G. 1996. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, **8**, 134–143.

Rintala, Tommi. 1996. *Population Size in GAs for TSP.* http://lipas.uwasa.fi/cs/publications/2NWGA/node11.html.

Russell, S., & Norvig, P. 2010. *Artificial Intelligence: A Modern Approach.* Third edn. Pearson Education, Inc.

6 Appendices

List of Figures

List of Tables

List of Algorithms

6.1 Figures

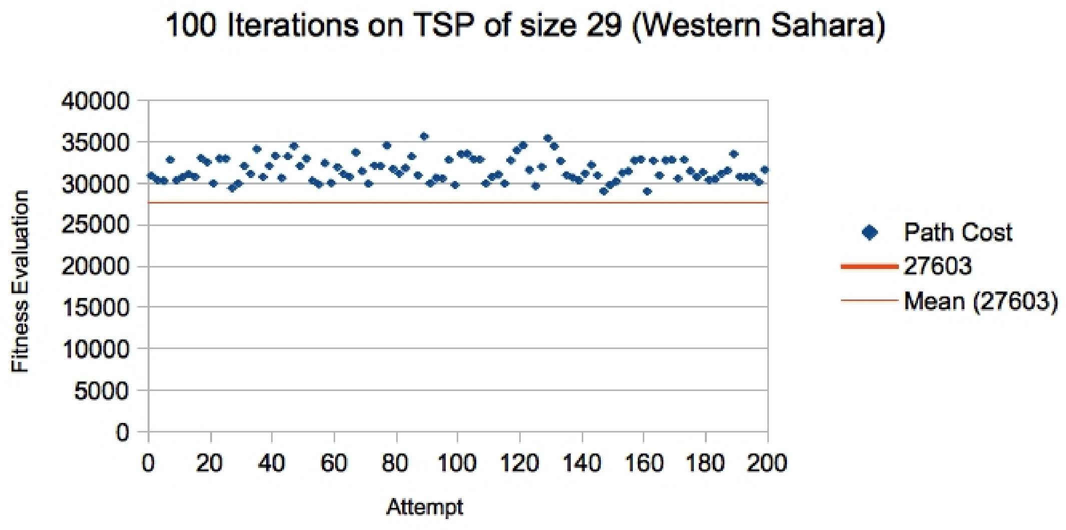

Figure 6.1: Comparison of 100 Iterations on TSP of size 29

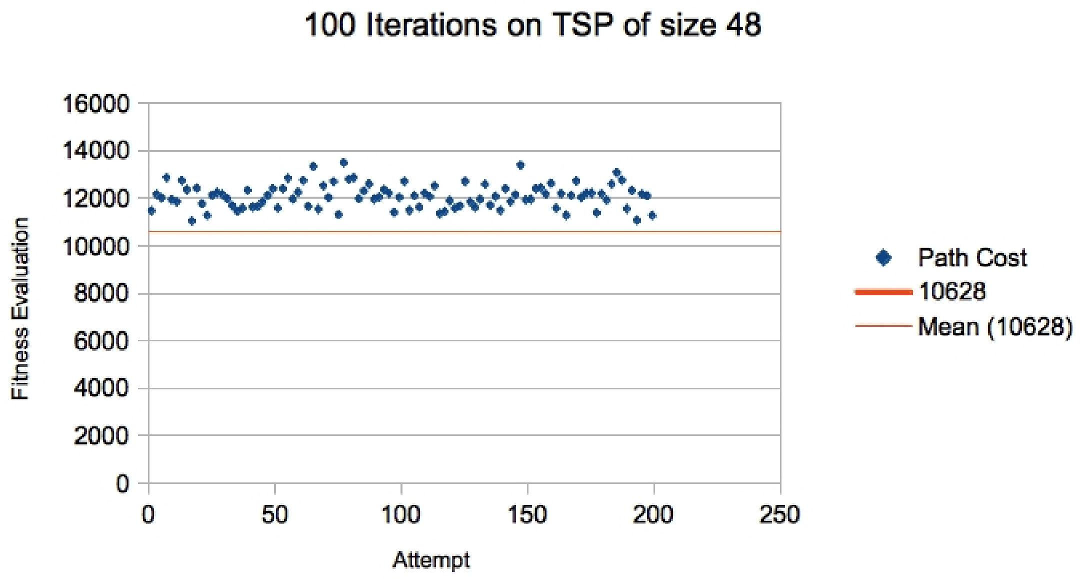

Figure 6.2: Comparison of 100 Iterations on TSP of size 48

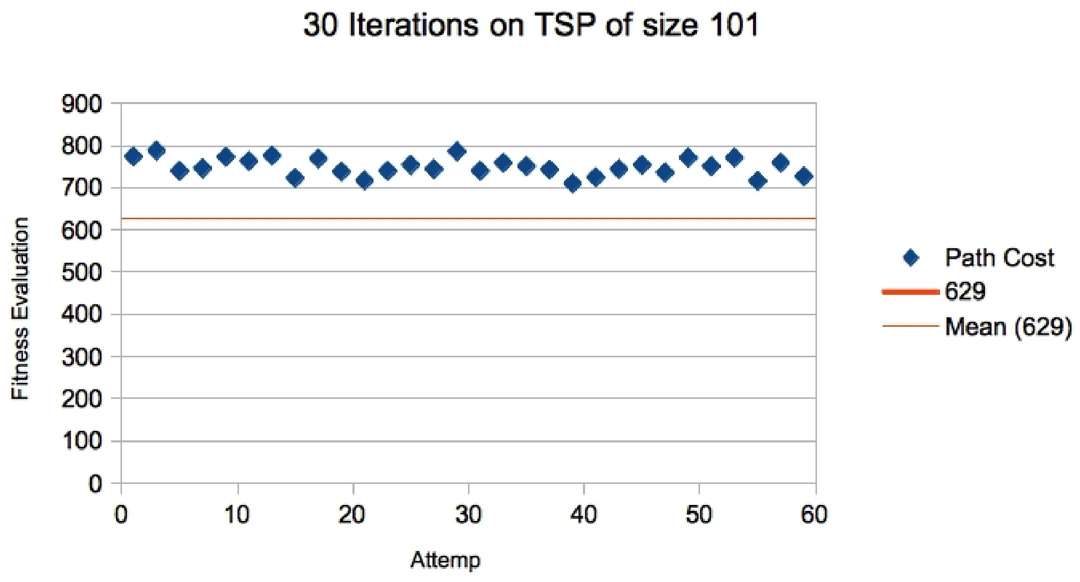

Figure 6.3: Comparison of 30 Iterations on TSP of size 101

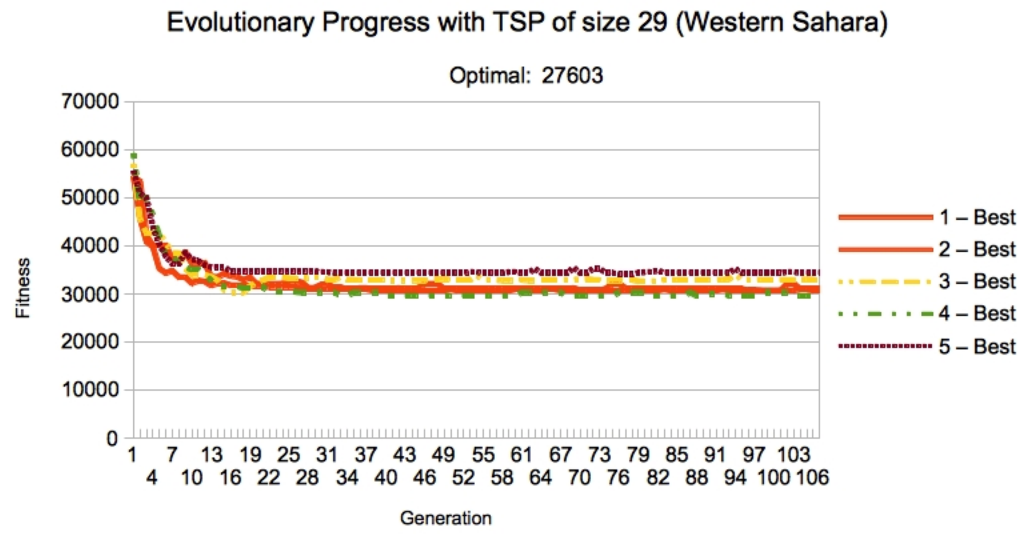

Figure 6.4: Evolutionary Progress with TSP of size 29

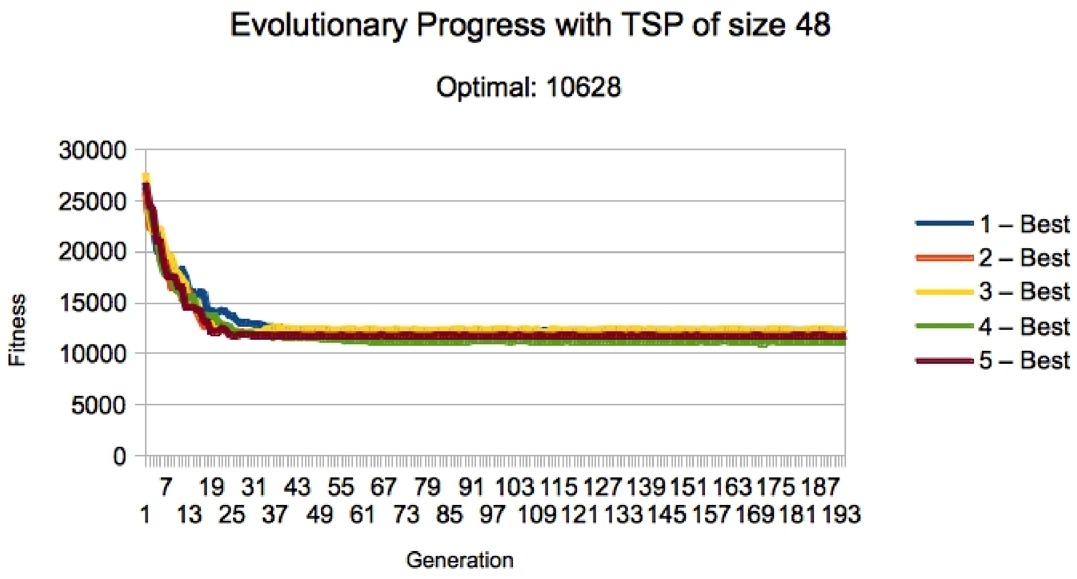

Figure 6.5: Evolutionary Progress with TSP of size 29

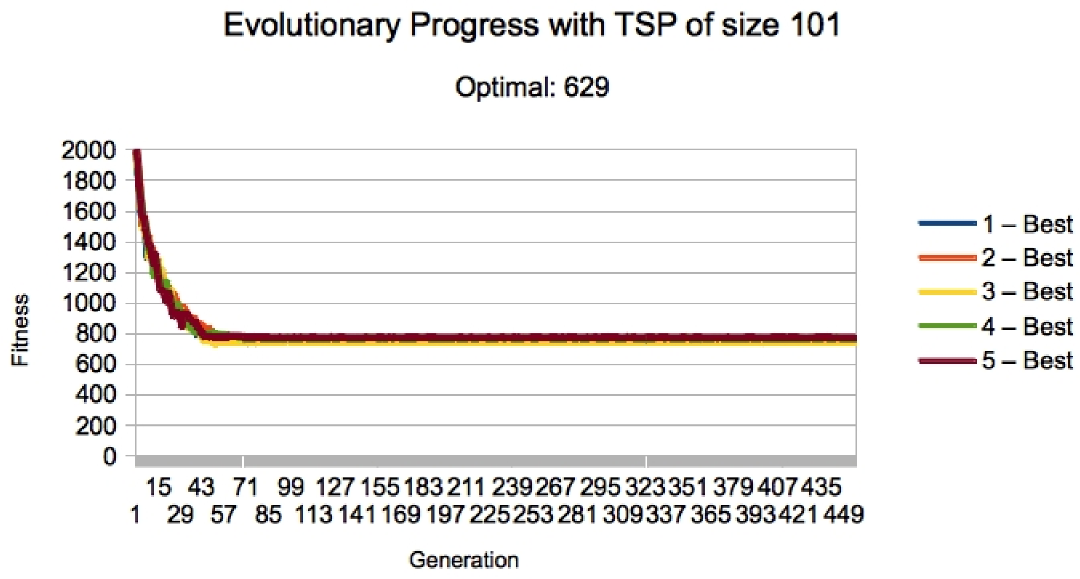

Figure 6.6: Evolutionary Progress with TSP of size 101

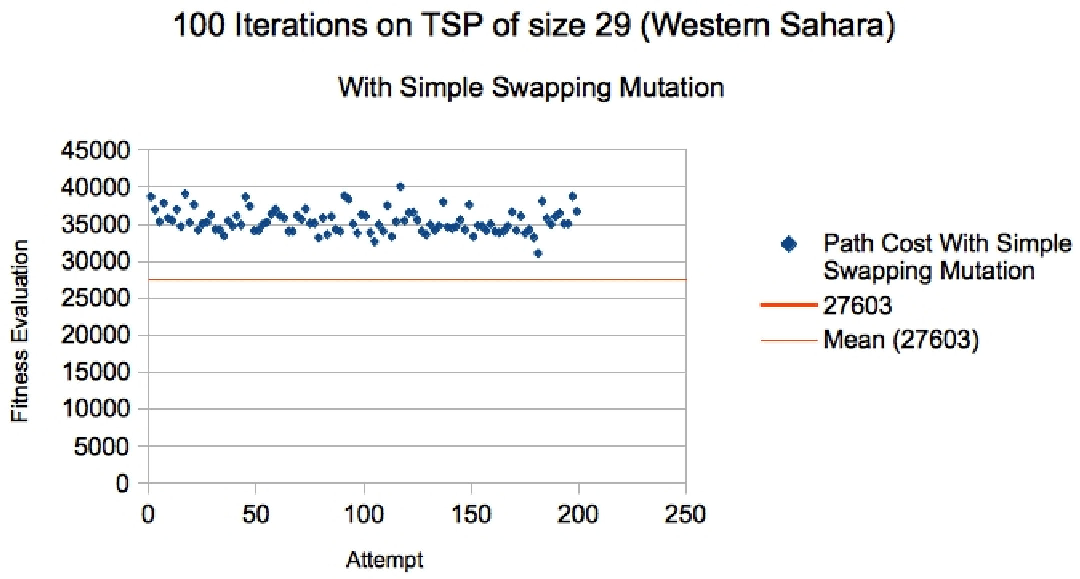

Figure 6.7: 100 Iterations on TSP of size 29 - with Simple Swapping Mutation

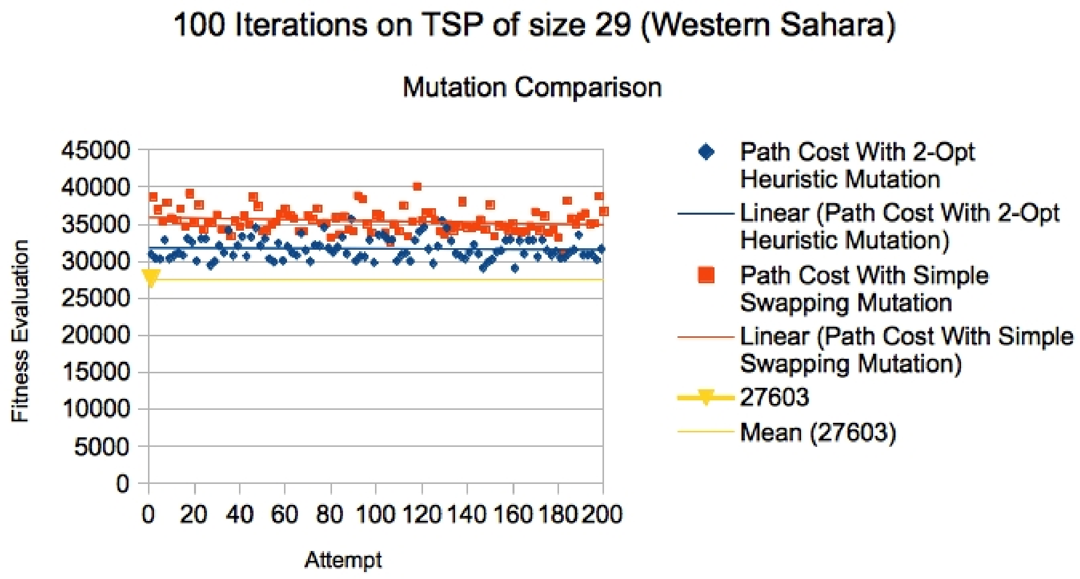

Figure 6.8: Comparison of the Simple Swapping Mutation with my proposed 2-Opt Heuristic Mutation

6.2 Tables

Average Fitness	31572
Average Percent from Optimal	12.698
Average Running Time	0.423 seconds
Best Fitness	29052
Best Percentage from Optimal	4.0%

Table 1: Summary Raw Data for 100 Iterations on TSP of size 29

Average Fitness	12067
Average Percent from Optimal	11.961
Average Running Time	3.847 seconds
Best Fitness	11043
Best Percentage from Optimal	3.0%

Table 2: Summary Raw Data for 100 Iterations on TSP of size 48

Average Fitness	748
Average Percent from Optimal	16.026
Average Running Time	67.56 seconds
Best Fitness	710
Best Percentage from Optimal	11.0%

Table 3: Summary Raw Data for 30 Iterations on TSP of size 101

	Random	**Stochastic**
Average Fitness	31694	31475
Average Percent from Optimal	12.942	12.465
Average Running Time	0.42 seconds	0.438 seconds
Best Fitness	27603	28191
Best Percentage from Optimal	0.0%	2.0%

Table 4: Comparison of Random Population Initialization with my proposed Stochastic Population Initialization - Over 1000 iterations of TSP of size 29

Mutation Rate	**.1%**	**1%**	**10%**	**25%**	**50%**	**100%**
Average Fitness	33010	33121	32244	31942	31255	32107
Average Percent from Optimal	18.174	17.198	14.641	13.875	12.005	14.596
Best Fitness	29983	29407	28856	28856	28405	29511
Best Percentage from Optimal	7.0%	6.0%	4.0%	4.0%	2.0%	6.0%

Table 5: Comparison of Mutation Rates with my proposed method - Over 100 iterations of TSP of size 29

Number of Evolutions	**5 * Original**	**3 * Original**	**1 * Original**	**.5 * Original**
Average Fitness	31354	31596	31572	31474
Average Percent from Optimal	12.112	12.7	12.698	12.470
Average Running Time (Seconds)	2.032	1.185	0.423	0.21
Best Fitness	28260	28375	29052	27946
Best Percentage from Optimal	2.0%	2.0%	4.0%	1.0%

* Original: $(log(1 - \sqrt[n]{.99})/log\frac{n-3}{n-1})$ (Rintala, 1996)

Table 6: Comparison of Maximum Evolutions with my proposed method - Over 100 iterations of TSP of size 29

Population Size	**5 * Original**	**3 * Original**	**1 * Original**	**.5 * Original**
Average Fitness	31417	31192	31572	32205
Average Percent from Optimal	12.565	11.919	12.698	14.518
Average Running Time (Seconds)	1.244	0.674	0.423	0.121
Best Fitness	29155	28557	29052	28612
Best Percentage from Optimal	5.0%	3.0%	4.0%	3.0%

* Original: $(log(1 - \sqrt[n]{.99})/log\frac{n-3}{n-1})$ (Rintala, 1996)

Table 7: Comparison of Population Size with my proposed method - Over 100 iterations of TSP of size 29

Culling Percentage (Percentage of population to keep)	**1%**	**10%**	**25%**	**75%**
Average Fitness	34369	33578	31660	31498
Average Percent from Optimal	21.096	18.131	12.907	12.527
Best Fitness	30312	29410	28070	28856
Best Percentage from Optimal	8.0%	6.0%	1.0%	4.0%

Table 8: Comparison of Culling Percentage with my proposed method - Over 100 iterations of TSP of size 29

Algorithm 2 Stochastic Initialization

```
public static int evaluate(LinkedList<Integer> cityList,
        TravelingSalesman salesman) {
  double s = 0;
  // If only one city, just return max value so it will be chosen
  if (cityList.size() == 1) {
        return Integer.MAX_VALUE;
  }
  for (int i = 0; i < cityList.size() - 1; i++) {
        // Gather the distance from the edge in the cityList
        s += salesman.distance(cityList.get(i), cityList.get(i + 1));
  }
  // add cost of coming back:
  s += salesman.distance(cityList.get(cityList.size() - 1), 0);

  // Take total edge average through the cityList
  return ((int) (s / cityList.size()));
}

public static Gene[] operate(Gene[] genes, Gene[] sampleGenes,
        TravelingSalesman salesman, LinkedList<Integer> cityList) {

  genes[0] = sampleGenes[0].newGene();
  genes[0].setAllele(sampleGenes[0].getAllele());
  Random generator = new Random();

  for (int i = 1; i < genes.length; i++) {
    int distance, average, location;
        Random random = new Random();

        do {
          // Grab a random location in the unused list
          location = generator.nextInt(cityList.size());

          average = evaluate(cityList, salesman);
          distance = (int) salesman.distance(((
                IntegerGene) genes[i - 1]).intValue(), cityList.get(location));

    } while (distance > average);

      genes[i] = sampleGenes[cityList.get(location)];
          genes[i].setAllele(sampleGenes[cityList.get(location)].getAllele());
          cityList.remove(location);
  }
  return genes;
}
```