Implementing a Genetic Algorithm to Solve the Traveling Salesman Problem

Dawson Merkle and Thomas Levitt

1. Introduction

In this report we will be examining and forming conclusions about the implementation of a genetic algorithm in solving the Traveling Salesman Problem (TSP). This implementation was developed using the Python programming language using the Visual Studio Code source code editor. The genetic algorithm used in this report emphasizes various functions to create populations of routes that are evaluated based on total route distance and chosen to be parents of a new generation. These operations are repeated, creating generations, and run until the number of generations is met. Using this algorithm, the goal is to find the shortest route possible using a sequence of the created cities.

2. Research and Planning

To begin, we had to decide our roles when designing this project. We both wanted to do the research, programming, and report as collaborative partners. When looking into this project we took the example from the knapsack genetic algorithm and built off the structure of that program. However, we also wanted to have a more object oriented approach to the TSP. We also read chapter 4 and 5 from the "*Grokking Artificial Intelligence Algorithms*" textbook by Rishal Hurbans to familiarize ourselves with the general structure of a genetic algorithm. We also used the book heavily in order to learn about which selection algorithms we would want to implement into our program. After reading chapter 4, we used the genetic algorithm lifecycle as a key piece of information to base our program off of. We planned to use each of the steps as a central idea

for our functions. Another resource used to understand the Traveling Salesman Problem was the video titled “Coding Challenge #35.4: Traveling Salesperson with Genetic Algorithm,” created by The Coding Train. This video was paramount in understanding the overall problem, how we could represent the population, and encode the solution space. We also wanted to keep a repository for our source control so we would be able to compare our changes and revert if needed.

3. Program Design

Our program starts with the initialization of the City class which returns an object of an individual city with x and y coordinates attached. The city class is then used to make ‘N’ number of cities which are then contained in a list. That list is then shuffled and appended to another list named population which loops over the population size, creating a population of randomized lists of cities. Since our population is a list of real values, our solution space was encoded using order encoding. The reason for this is because every valid route has to contain every city. We have defined each list of cities as a route for the Traveling Salesman to take: Index 0 being the start. After creating the population, we need to calculate the fitness of each route. This is done by a nested for loop where the outer loop is the route in the population and the inner loop is the city in the route. The inner loop gets the distance from one city to the next and then from the last city to the first and then sums it to create the fitness for that route. The outer loop then compares the routes and tries to find the global best route and appends that route to a list of route lengths. After the list of route lengths is made, we must set the probability of the routes. This is done by normalizing the route lengths and returning the normalized probabilities. These probabilities make up the slices of the roulette wheel when proportional roulette selection is run. This results in two parents from the existing population. We chose this selection because it allows for routes that don’t necessarily have the best fitness to still have a chance at being chosen, thus, this selection

can encourage more diversity. The selection is done by comparing the probability with a random number. In the context of the Traveling Salesman, there are not many options to choose from when crossover between parents is necessary. The reason for this is because the potential routes are permutations and require each city to be in the route only once. If the city is present in the route it would be considered invalid. With this in mind, we chose to make random indices that take a chunk of the first parent's route and append that to the child. After this, whatever cities are not present in the child yet are appended and the child is created. When this process completes, the child has a chance of being mutated, which directly relates to the mutation rate. If the child is chosen to be mutated, the child has two random cities swapped in its route. This mutation would be considered an order mutation. In this method of mutation, “two randomly selected genes in an order-encoded chromosome swap positions, ensuring that all items remain in the chromosome while introducing diversity” (Hurbans 143). Once the children are produced, the worst members of the population are killed off at a proportional number to how many children were produced. These children are then merged with the remaining members of the old population to form the new population. This entire process explains one generation which can be repeated for any amount of generations the user wants to create. The stopping condition of the algorithm is seen when the number of generations are fulfilled.This allows for the algorithm to run however long the user wants it to, but also have a good chance at finding the global best route. Another design specification that we decided on was to have explicit names associated with each city. This was useful in the development process when we were making sure that the coordinates of the cities weren’t changing and also when we were shuffling around lists. Also due to this, the maximum number of cities that we can simulate is 58, because of the amount of cities in the state of Florida. In the main function of our program, there are many hyperparameters that we decided on making. These include, the number of cities (N_SIZE), the number of generations (NUM_GENERATIONS), the population size (POP_SIZE), the mutation rate

(MUTATION_RATE), the kill rate (KILL_RATE), and the number of children created every generation (NUM_CHILDREN). Creating these to be used as arguments for our functions allows for the algorithm to be changed however the user desires. For example, using these values, diversity can be increased or decreased per generation, the program can be run for much shorter or longer, and the number of overall cities can be changed to create a more complex problem.

To illustrate our findings, we have plotted the best route length over the course of generations and also the best route shown as a diagram.

4. Representative Results

The results of our genetic algorithm are represented easily by the plots we have made. The first output of our plotting shows the initial route made with no computation. (Figure 1)

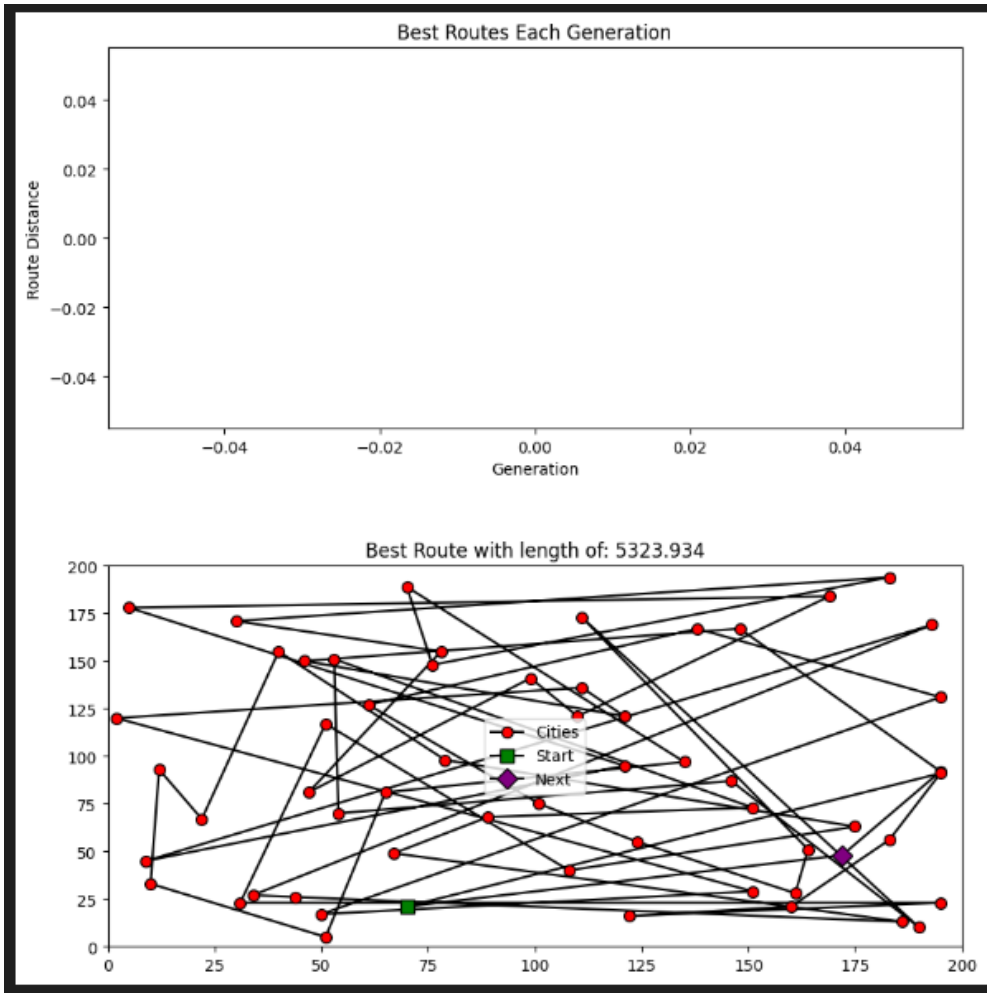

Figure 1

As shown, the “Best Route Each Generation” plot is empty as there is no data at the beginning. We programmed the plotting to occur every 500 generations so it could be evident to see the improvement. However, the most improvement happens within 500 generations. When the program is finished, the “Best Route Each Generation” plot provides a very nice curve showing that the most improvement happens very early on in the generations and then tapers off. The Best Route diagram also provides a much cleaner route for the Traveling Salesperson. (Figure 2)

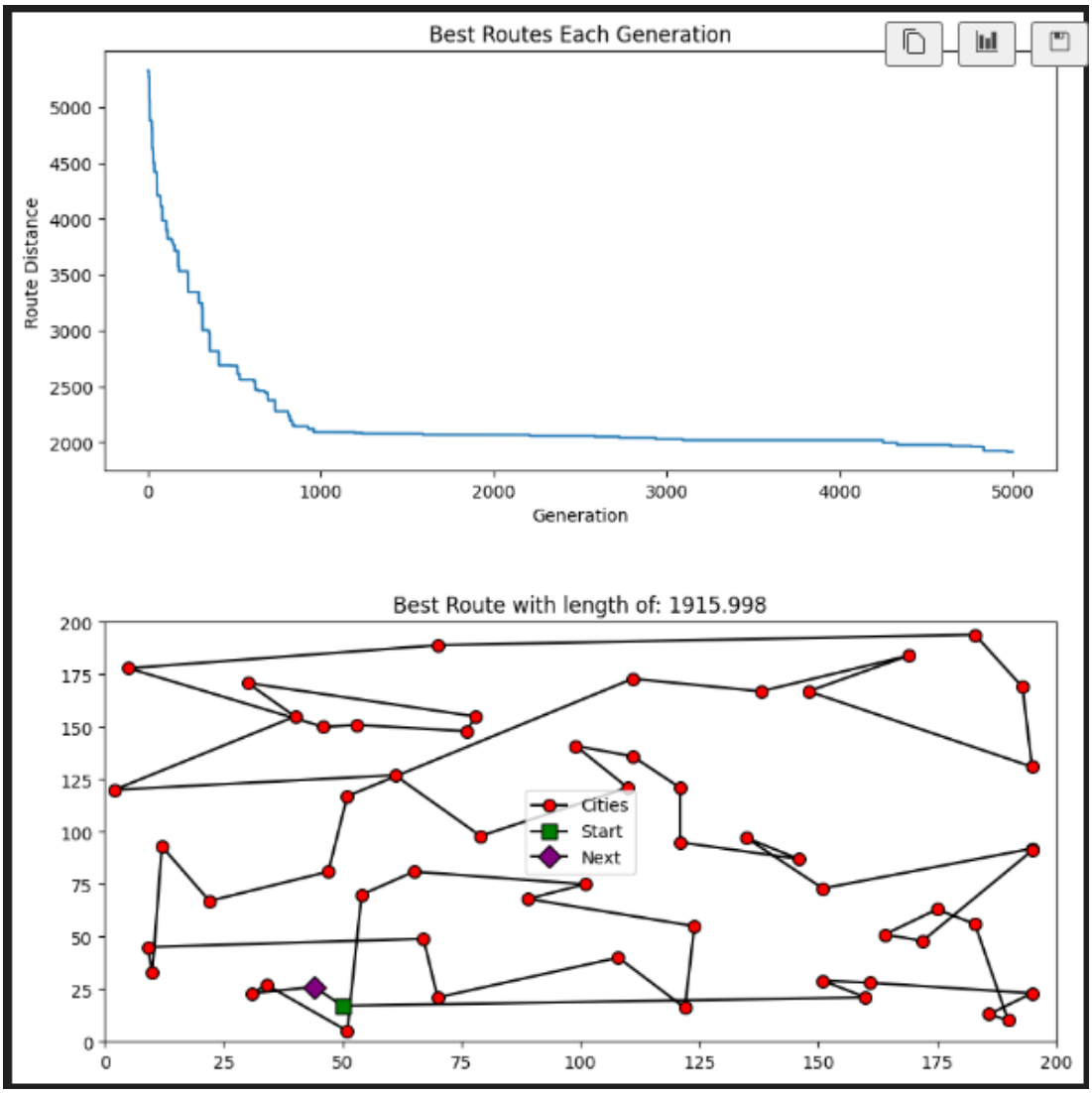

Figure 2

5. *Future Improvements*

While the project was completed and done to the best of our abilities, there are always improvements that could be made. One example of these includes making a single animated plot that is constantly updated, instead of displaying multiple plots per however many generations. This would allow for the user to see the best route being updated in real time. Along with this, we would choose to have the names of the cities included in this plot, allowing for visualization of the optimal route to have meaning. Another potential improvement would be adding an option for the user to select the specific selection strategy that they would prefer. Currently, we only included the proportional roulette selection strategy, but in a perfect world, tournament selection and ranked roulette selection would be options to run as well.

The following link navigates to the GitHub Repository where project can be found: https://github.com/tlevitt1/TSP-GA

6. *References*

Hurbans, Rishal. *Grokking Artificial Intelligence Algorithms*. Manning Publications Co. 2020, pp. 91-151.

Hurbans, Rishal. Knapsack Genetic Algorithm [source code]. https://github.com/rishal-hurbans/Grokking-Artificial-Intelligence-Algorithms/blob/master/ch04-evolutionary_algorithms/knapsack_genetic_algorithm.py.

Shiffman, Daniel. “Coding Challenge #35.4: Traveling Salesperson with Genetic Algorithm” *YouTube*, uploaded by The Coding Train, 1 May 2017, www.youtube.com/watch?v=M3KTWnTrU_c&ab_channel=TheCodingTrain.

Shiffman, Daniel. “Coding Challenge #35.5: TSP with Genetic Algorithm and Crossover” *YouTube*, uploaded by The Coding Train, 2 May 2017, www.youtube.com/watch?v=hnxn6DtLYcY&ab_channel=TheCodingTrain.