The Traveling Santa Problem

Travis Shumaker
Computer Science
Old Dominion University
Portsmouth, United States
tshum003@odu.edu

John Hessefort
Computer Science
Old Dominion University
Norfolk, United States
jhess004@odu.edu

Matthew Perry
Computer Science
Old Dominion University
Norfolk, United States
mperr006@odu.edu

Robert Hickman
Computer Science
Old Dominion University
Norfolk, United States
rhick012@odu.edu

Abstract — In this project our group attempts to find the two shortest and disjoint paths between one hundred and fifty thousand cities. Within the program is implemented a genetic algorithm in order to simulate natural biological processes in the hopes that we discover the shortest paths. Paths are generated in a pseudo-random manner through usage of a first order (Order 1) crossover method by using a fitness function to calculate the best path.

Keywords — TSP, Travelling Santa Problem, population, crossover, mutation, Fisher-Yates Shuffle, Path, Fitness, Distance, Disjointed, order, JavaScript, C++

I. Introduction: Problem Definition

The Travelling Santa Problem (The "TSP") is a more complex variation of the well-known Travelling Salesperson Problem wherein one is presented with a scenario involving a salesperson who wishes to visit a number of cities in order to market their goods to the public. In order to save time and money from travelling expenses, the salesperson would like to find the shortest path that visits every city and returns to the original departure point. In most instances, one is introduced to this problem in a scenario where the salesperson visits a relatively unambitious amount of cities in comparison to the TSP, in which case it may be considered feasible to use a brute-force method to find the shortest path between them; however, with a substantially larger amount of cities, this method is no longer feasible. For example, in the Travelling Santa Problem, Santa is not searching for the shortest path between a small amount of cities, as the traveling salesperson traditionally does. Instead, Santa must travel between one hundred and fifty thousand cities. Due to this fact, the brute-force method is very ineffective.

For the second portion of the TSP, one is required to find a "disjointed path" from the shortest found order of cities within the first section. From the definition of disjointed, the problem requires that there is found a secondary "unique" path for Santa and Rudolph to travel through on Christmas Eve. This secondary path is unique in the sense that it shares no edges with the first path that is found; moreover, this means that any two cities directly connected in the first path cannot have this same connection in the second path, regardless of whether or not this connection shares the same direction or if the direction is reversed. For example, if in the initial path there is a connected city named A to a city named B, then in the second generated path there can not be an edge that connects A to B nor can there be an edge that connects B to A, and this principle applies for all edges that appear in both paths.

II. Methods

This project is implemented using a genetic algorithm which makes use of various crossover and mutation methods to attempt to find what is, ideally in an early generation, the shortest-distance path between the 150,000 cities that Santa must visit.

A. Initial Population Setup and Evaluation by Fitness Function

Before the program can began the Travelling Santa Problem, a few prerequisites must be met.First, the cities must

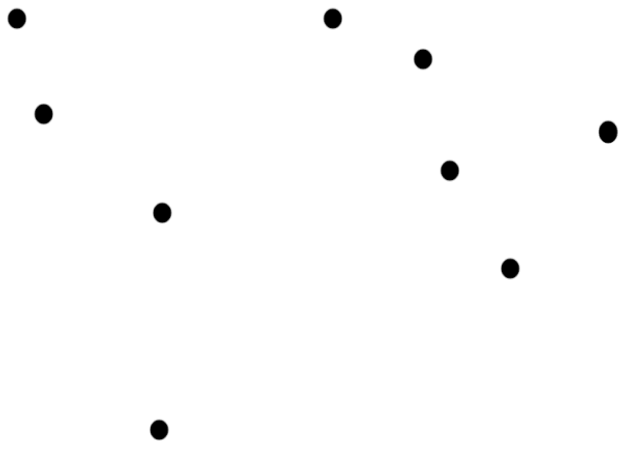

Figure 1: Unassigned and unidentified city collection

be loaded in from the provided kaggle Comma Separated Values (csv) file, which lists all of the cities for this situation. Upon doing this, each city will have a general location upon a two dimensional grid and an ID.

The program begins with the initialization of the first generation, or first population. The population stores within itself a variety of randomized orders, which are data collections used to describe the order in which the cities are to be visited. To create these randomized orders, there needs to be a default order created first, which is able to be randomized. Read in from the .csv input file are the coordinates and ID number for each city, and for a smaller group of cities, they can be seen as is in Figure 2.

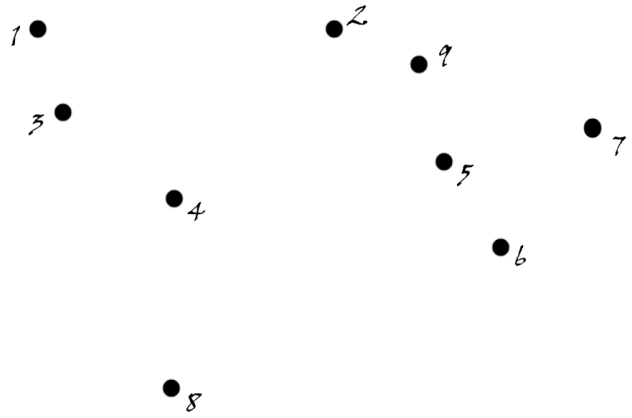

Figure 2: Allocated city collection identified by ID

The default order is created as follows. Cities are stored into a data collection vector as integer identification numbers between one and one hundred and fifty thousand. The IDs for each city are stored sequentially. The cities are placed in sequential order with respect to the city identification numbers, as depicted in Figure 3, and this signifies that by default Santa will visit the cities in order of ID.

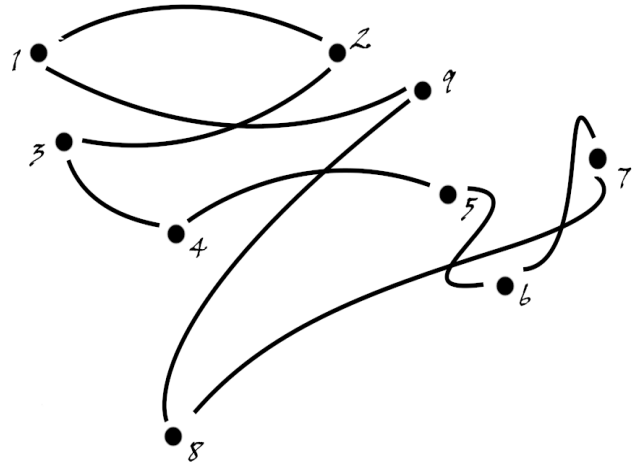

Figure 3: Default order wherein cities are visited sequentially by ID

After the path is initialized sequentially, the ordered path can now be randomized using a pure random method of sorting called the Fisher-Yates sorting or shuffle algorithm. This shuffle function is used a number of times equal to the size of our population. For each time the default order is randomized, a new order is created, set equivalent to the randomized post-shuffle order, and stored within our population vector. As the orders are data collections of such large size, it is not feasible to have a population size of large magnitude, and so the size of the population is set to a constant 10.

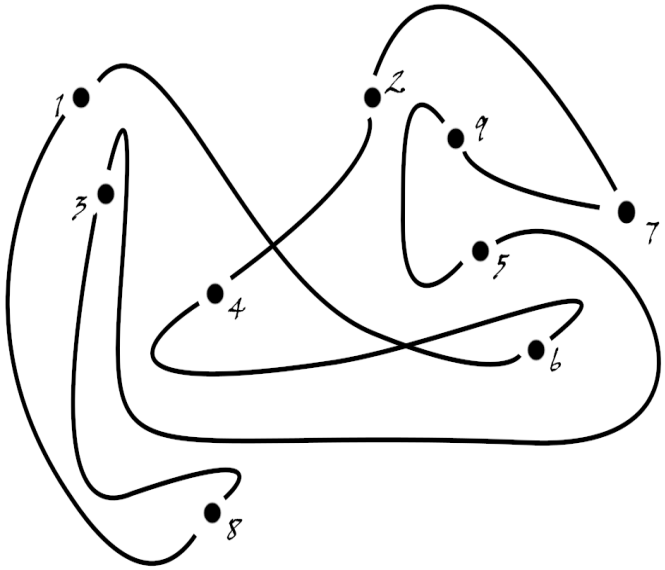

Figure 4: An example of an order shuffled through the Fisher-Yates method

With the population of the first generation successfully created, the evaluation of the fitness of each random order is calculated. The fitness function operates with the knowledge that fitness and distance are at odds with one another, in that they are inversely proportional; furthermore, the greater the distance of a path is, the lower its fitness score should be. Similarly, the greater the fitness of the path is, the lower the distance of the path should be. In other words, the lower its distance. Specifically, the distance is calculated using the distance

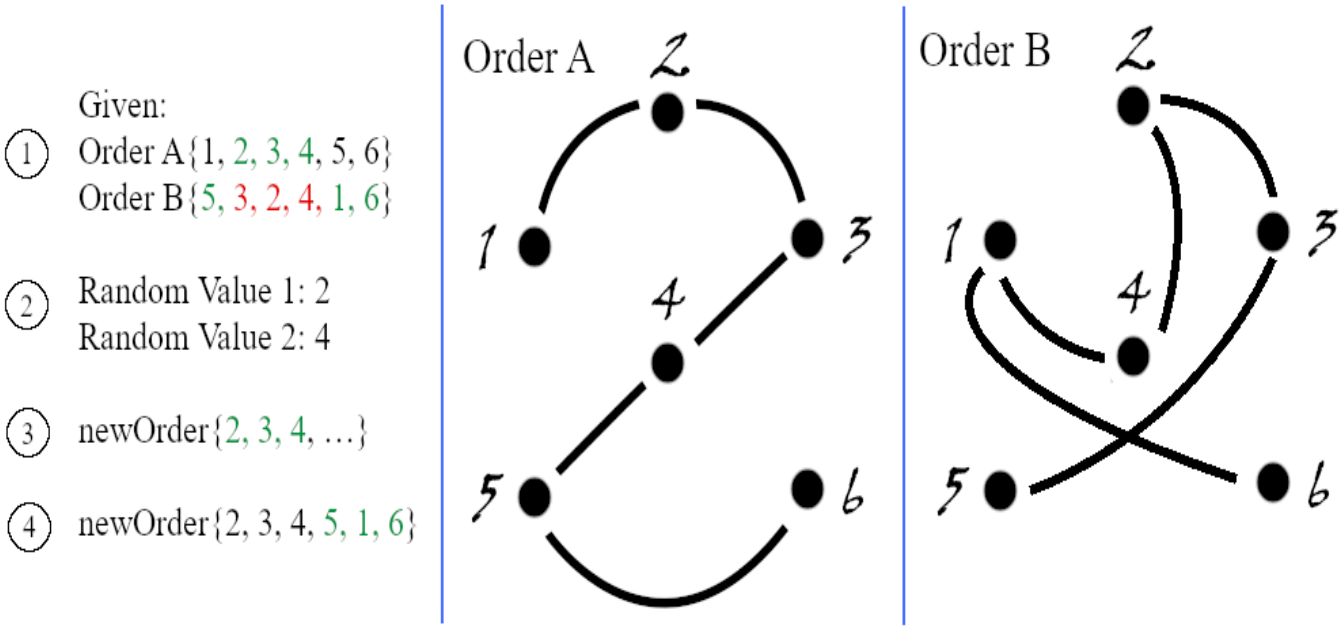

Figure 5: Demonstration of the Order 1 crossover being performed on two orders, A and B, to produce a child of the next population, newOrder

formula for two points and the fitness is calculated with the formula fitness function:

$$1 / (\text{pow}(d, 8) + 1) \quad (1)$$

Where 'd' is the distance and pow(a, b) is a function that raises a value 'a' to the power of 'b'. The fitness scores are then normalized through division by the sum of all fitness scores for the order.

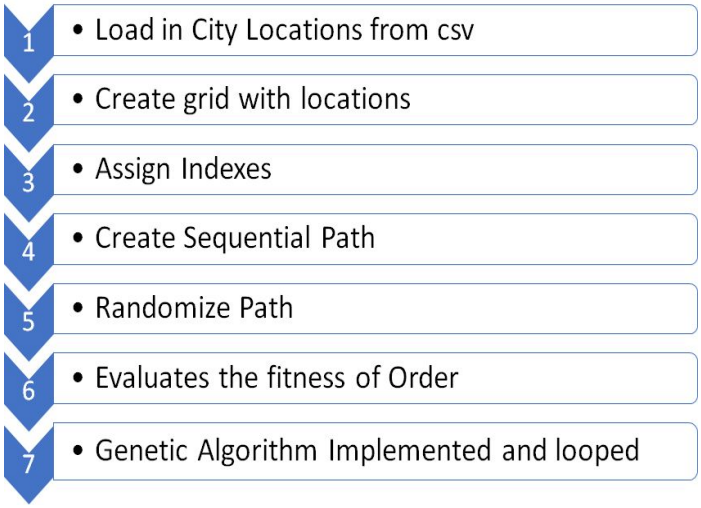

Figure 6. Methods Ordering

B. Genetic Algorithm Implementation

Implemented in this program are a number of crossover and mutation functions. After evaluation of the fitness score of each order, displayed to the top half of the browser window is the best path ever to have been found by the genetic algorithm, and this continues to be updated as crossovers are performed upon randomly-selected orders from within our current population.

For a number of times equivalent to the static size of the population, what is done is a crossover of the first order (Order 1) form on the two randomly selected orders. Each time the crossover is performed, that is, for each time there is a new child order to be added to the next generation's new population, there is a 5% chance for this child order to mutate. For the crossover, two values are selected at random. Two indices of the order, start and end, where the index start is guaranteed to come before index end. After this is done, all elements between the "start" and "end" feature are copied in that order into the new child order. From here, the second order, B, is searched sequentially for all city IDs that were not copied from the first order, A. The missing elements of order B are added to the new child C as they are encountered. The mutation that may take place on any newly generated order is a fairly simple swap mutation in which two indices are randomly-selected from the

order and then the elements at those locations are swapped and thus change locations.

III. Results

From this project we find that the program is unable to feasibly test every possible path between the cities to find the shortest path. From there we implement a genetic algorithm to mimic realistic biological processes in the hope that we can "breed" ideal members of the populations with one another and from this create child orders possessed of higher fitness so that we may achieve a low cost path in a short time. With regard to the implementation of our crossover method, we did not use a "true" Order 1 crossover. Our crossover would take a randomly-chosen section of elements from the first order and place it at the very first index of the new order, rather than at the same section of the new order, so perhaps the crossover method could have been implemented in a different fashion to improve the overall fitness scores of each population. Additionally, the mutation rate for each child order could have been altered to be either higher or lower to produce better results. The mutation as it is now has a 5% chance of occurring on the child order for the number of cities in the problem and for each city that exists within that order. After periodic experimentation it was found that, for this swap mutation implementation, it is actually on average and in the general case more efficient to have the mutation rate be lowered such that we avoid altering any of the positive changes that may have come of the crossover. We believe that we could have implemented a better wrapper function for our crossover method; moreover, the orders we choose to breed are not very specific, and are instead randomly-selected by fitness function; specifically, after normalizing the fitness scores for each order, we then generate randomly a number within the range of [0,1] and if, for example, we have a path with normalized fitness score of 0.75, then the remaining fitness scores could be added together for a sum of 0.25. If the randomly generated number falls at or below 0.75, then the path with the corresponding fitness score would be selected for crossover. It would likely have yielded better populations for us if we were to breed specifically higher fitness orders with one another. We planned to crossover orders in sequence of fitness function value, so the two orders of best fitness could be made to breed, followed by the second best pair of orders, and so on. Otherwise, we also planned to try breeding the best order with half of the remaining orders, and then have the other half breed at random.

IV. Conclusion

With regard to our first path, we are able to find a path of shorter distance than the upper bound of 12 million units. We began by testing our implementation methods on a smaller set of only one thousand randomly allocated cities in a JavaScript (or "JS") program that would graph the cities and path as ellipses connected by vectors on a canvas in a web browser. We found that it is not viable for someone to actually graph all of the cities on the canvas and dynamically connect them with vectors to represent the path of travel, through JavaScript, when the number of cities is greater than 100; moreover, for a larger collection of cities, one would need to increase the size of the canvas that the cities are appearing on, and for a large canvas, it becomes no longer feasible for the browser to run efficiently. While the locations of the cities themselves do not change, the edges that connect them are constantly being updated on the screen, which is a intense process on the CPU. Instead, we are forced to rid ourselves of the preferred graphing method, which is one of the primary reasons that JavaScript was initially chosen. Instead of graphing each path as necessary, we simply print the distance and corresponding fitness of each path that would normally appear to the screen, which saves a fair amount of execution time in situations where we limit ourselves to a specific number of generations for our genetic algorithm to produce. Through usage of the console.log() JS function, we found that after approximately 10 generations, on average, we were able to improve upon the base path order distance of approximately one hundred thousand distance units through exploration of paths and usage of the aforementioned genetic algorithm. Our final path after 10 generations was of size approximately fifty thousand units. This test was done without the usage of the draw function which, as seen in our presentation, would keep track of and graph in the upper half section of the canvas the best path found up to the current time in the program, and in the lower half section of the canvas there would be graphed the best path found in the current population up to that point. This process is highly intensive upon the CPU for a large amount of cities, as we encountered in the Travelling Santa Problem. The goal of the disjoint path was not achieved, as we had not figured out how to effectively compare every edge of the first path, both as normal and reversed, to every path in every potential second best path. We discovered on the day of the presentation that the maximum amount of paths that we have to check is not $O(N^2)$, where N is the number of cities on our canvas, rather O(2N), as each city is connected only to two other cities. Upon execution of the program, in search of our unique disjointed path, we could have run a checkUniqueness() function on our best path ever every time it has been updated

after the initial-most generation and population. Surely, this second program execution would take longer than our first even if we do only run the genetic algorithm for a number of generations equivalent to that of our first execution. From this project we learned that the best solutions, or at least those solutions that are of comparatively higher quality, to complicated problems are not always the most complicated solutions; this is seen with our implementation of the genetic algorithm. It is not a particularly difficult algorithm to implement, and the brute-force method can be argued to be more complicated. In spite of this, we find that the genetic algorithm approaches an ideal solution at a much faster rate.