Traveling Salesman Problem: Genetic Algorithm

Jacob Taylor Cassady
Computer Engineering & Computer Science
Speed School of Engineering
University of Louisville, USA
jtcass01@louisville.edu

1 Introduction

The Traveling Salesman Problem (TSP) is a well-known non-deterministic polynomial-time hard problem that has been studied within mathematics since the 1930s. The "salesman" is given a list of cities with their locations and is asked the shortest route to travel to each city once and then return to the starting point. A program was developed using Python 3.7 and accompanying 3rd party libraries: NumPy, Pandas, and matplotlib to determine the shortest path.

2 Approach

The approach taken to solving the TSP was to use a genetic algorithm. Throughout this document cities from TSP will be referred to as "vertices" and the route between a pair of vertices as an "edge". Development of the algorithm was aided by a graphical user interface (GUI). The GUI displays the previous best chromosome as the algorithm evolves.

The genetic algorithm implemented is inspired by sexual reproduction of gametes in biology. This algorithm retains a constant population of "chromosomes" which are representations of possible solutions/agents for/within the given problem. These chromosomes are a set of alleles that describe its performance. The algorithm makes use of two functions to evolve the population overtime to weed out the poor performers and mate the good performers and continues this cycle until 10 consecutive cycles occur without improvement. For the high-level implementation of the algorithm please refer to **Figure 4** in the appendices.

2.1 Crossover

The mating process is called "crossover". This process involves creating a new chromosome while inheriting sections of two parent chromosomes. The resultant chromosome is then used to replace poor performers. For reference to the implementation of the crossover method from the perspective of the population, please refer to **Figure 5** in the appendices.

Two crossover methods were defined. Both methods were developed by the authors of this paper; although, they may not be novel in concept. One method alternates the alleles of both parents. This method allows for substantial differentiation between parents and their offspring. In the results section, this method uses the alias CrossOver Every Other (COEO). The second method splits the parent chromosomes in half and then creates a child that has the first half of one parent's alleles and the second half of the other parents. In the results section, this method uses the alias Halfies. For reference to the implementation of these methods from the perspective of the parent chromosome, please refer to **Figure 6** in the appendices.

2.2 Mutation

An additional method is used after crossover to add more randomization into the process, this method is called "mutation". The mutation process involves randomly altering the alleles of some percentage of the population.

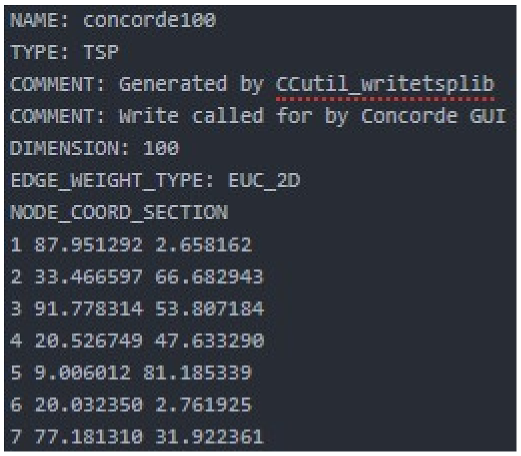
Only one mutation method was implemented. This method randomly chooses 2 alleles and switches their location in the chromosome. For reference to the implementation of this method, please refer to **Figure 7** in the appendices.

3 Results

The genetic algorithm was successfully implemented along with its functions: crossover and mutate. The algorithm produces a route that visits each city; although, the output is not necessarily the shortest route. The algorithm did not have any issues running with only 4 GB of RAM. No mitigation techniques were needed to reduce program memory usage or improve runtime efficiency. Tests were ran using different sample sizes as well as the two different crossover functions for comparison.

3.1 Data

The algorithm was tested on one dataset of 100 randomly generated cities. Within the test file, cities are enumerated, and x and y coordinates are provided. The input data was formatted like the example shown in **Figure 1** below. Additionally, a graphical representation of the graph of cities is shown in **Figure 2**.



```
NAME: concorde100
TYPE: TSP
COMMENT: Generated by CCutil_writetsplib
COMMENT: Write called for by Concorde GUI
DIMENSION: 100
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
5 9.006012 81.185339
6 20.032350 2.761925
7 77.181310 31.922361
```

Figure 1 : Random100.tsp Input File Format

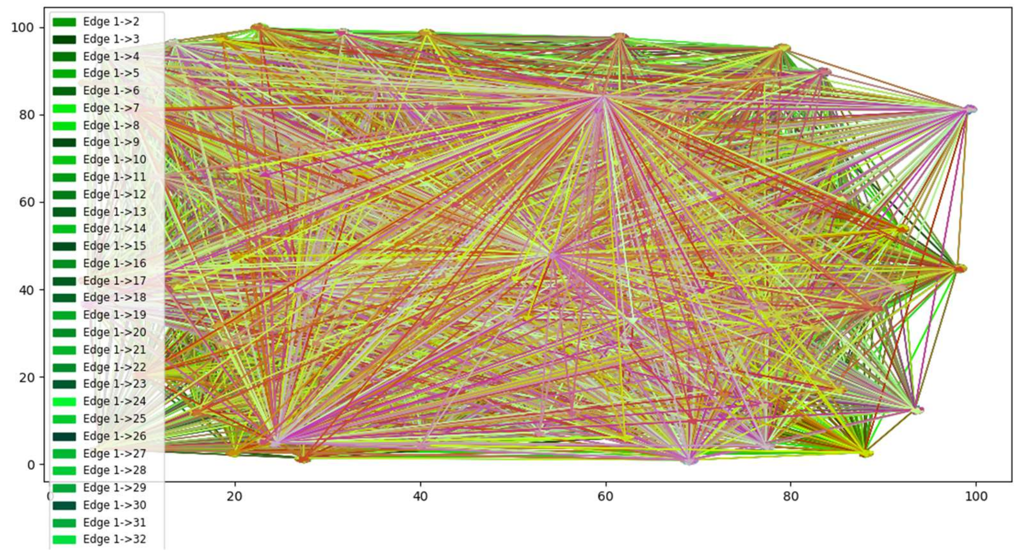


Figure 2 : Graphical Representation of Random100.tsp

3.2 Results

As previously mentioned, the algorithm was tested using two different crossover methods as well as two different population sizes. Both algorithms used a crossover probability of 60% and a mutation probability of 2%. Furthermore, costs for the algorithm over time were aggregated to keep track of how the algorithm improved. Below are tables populated with data from test runs. Graphs of the runs' costs can be found in **Figures 8 - 16** in the appendices. These values were gathered after each cycle of crossover and mutation. Additionally, the graphical output of each run can be found in **Figures 17** - **24** in the appendices.

		COEO	
Run #	**Population Size**	**Runtime**	**Shortest Distance Traveled**
1	50	663.846877	4316.454802
2	50	416.008783	4294.35774
1	100	1510.38222	4252.945025
2	100	253.966941	4218.685616

Table 1 : COEO Test Results

Table 1 above shows the COEO crossover method applied to two different population sizes.

		Halfsies	
Run #	**Population Size**	**Runtime**	**Shortest Distance Traveled**
1	50	58.3732955	4702.190834
2	50	53.1031973	4680.279964
1	100	100.996949	4597.500092
2	100	184.962564	4701.354649

Table 2 : Halfsies Test Results

Table 2 above shows the Halfsies crossover method applied to two different population sizes.

Crossover Method	Population Size	min	max	average	std_deviation
COEO	50	4294.358	4316.455	4305.4063	15.62498238
COEO	100	4218.686	4252.945	4235.8153	24.22506042
Halfies	50	4680.28	4702.191	4691.2354	15.49332476
Halfies	100	4597.5	4701.355	4649.4274	73.43626151

Table 3 : Crossover Method Performances

Table 3 above shows a comparison between the two tested crossover methods as well as population sizes. The min, max, average, and standard deviation columns are with respect to the distance traveled throughout the route.

4 Discussion

4.1 Performance

Output performance for population sizes of 50 and 100 produced very similar averages. This may be because 50 is already many chromosomes. On the other hand, the two crossover methods performed very differently from each other although the crossover probability, mutation probability, and mutation methods were all the same.

COEO produced cost graphs that monotonically decreased over the series of cycles. On the other hand, Halfies never improved its distance traveled. This may be a consequence of the relative higher level of entropy of COEO when compared to Halfies. It does less to preserve the order of alleles between the two parents than Halfies does. In conclusion, COEO is the preferred crossover method between the two.

4.2 Time

Although some runs produced similar results, the runtime of the genetic algorithm is variable. Some runs might never produce an improvement due to the luck of the random chromosomes generated and thus finish much faster than other. Others might make small improvements after several cycles of no improvements dragging out the runtime.

4.3 Issues

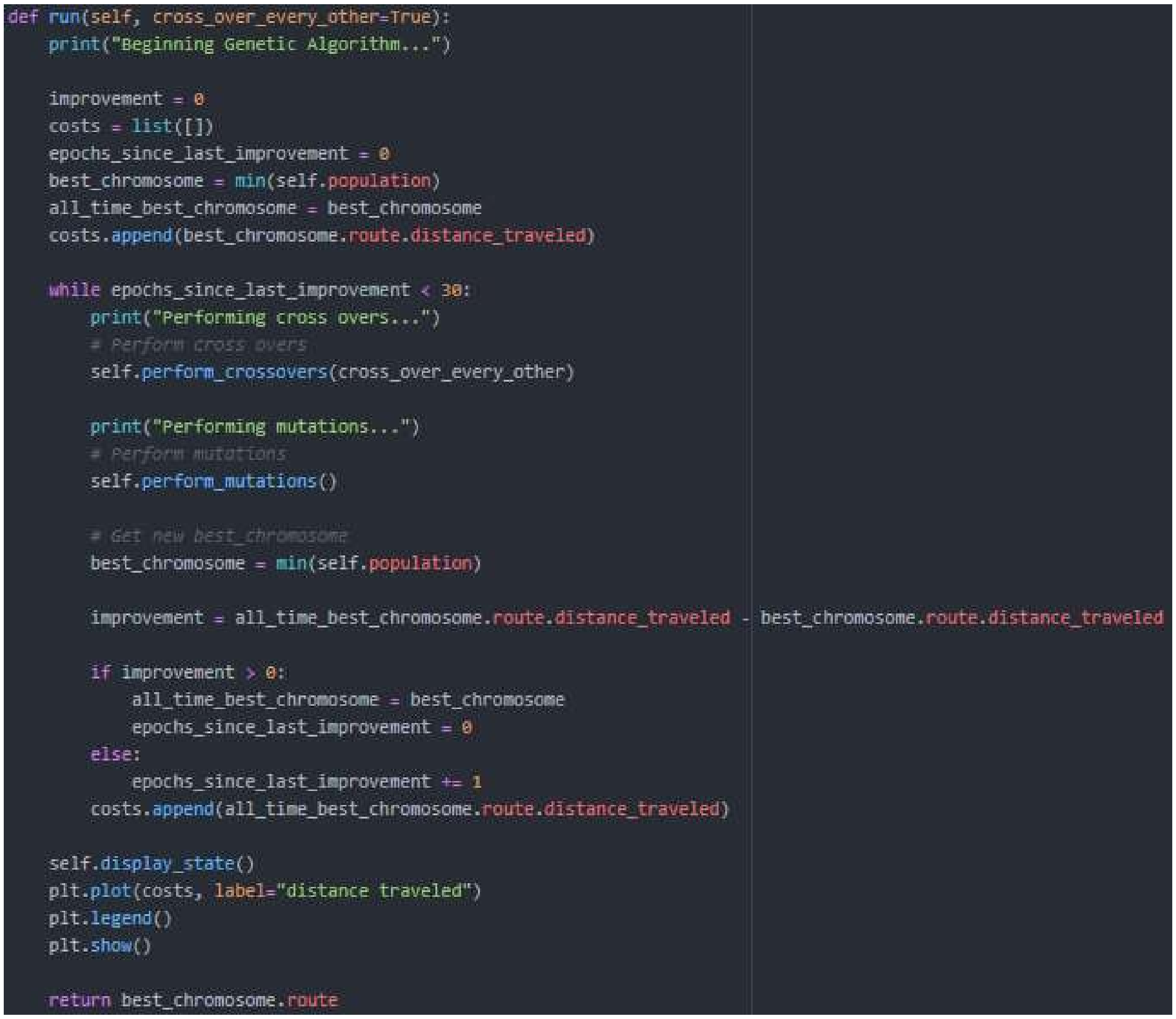
An issue when designing the algorithm was coming up with different crossover methods while attempting to choose the best one for the problem. I ended up choosing the two methods that were the most obvious to me, but a lot of time was spent trying to determine which of those available would be optimal for this problem. If I could change one thing in the implementation, I would add a more descriptive GUI that contained graphs of each member of the population as they evolved. This would help visualize the differences in how the crossover methods behave.

Genetic Algorithms are not deterministic and may produce less than optimal results on the first run. They can be great tools for attempting to solve problems there is not a defined solution for, but they are not the best tools for solving problems that have more deterministic solutions. Genetic Algorithms might be a great choice for implementing a program that works towards a solution when you don't readily know the next step to take. Especially if it involves a large solution search space, a well-defined fitness function, and a discrete set of alleles.

5 References

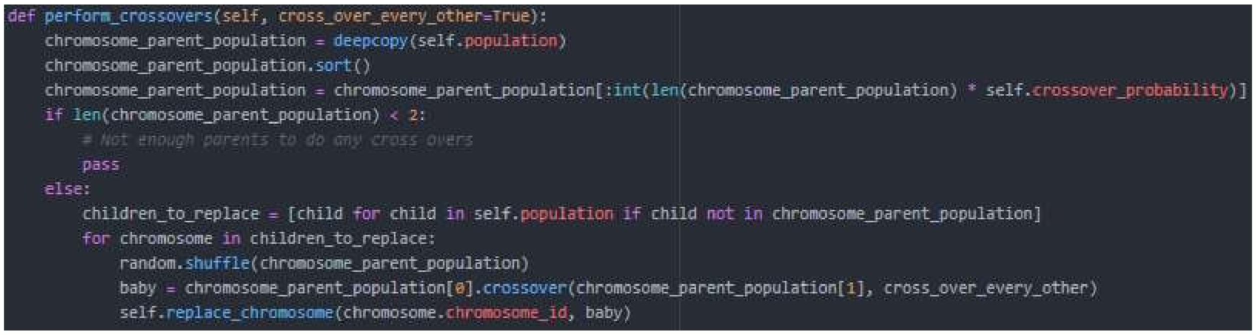
Wikipedia, Traveling Salesman Problem - https://en.wikipedia.org/wiki/Travelling_salesman_problem#History
NumPy Documentation - https://docs.scipy.org/doc/
Pandas Documentation - https://pandas.pydata.org/pandas-docs/stable/
Matplotlib Documentation - https://matplotlib.org/3.1.1/contents.html

6 APPENDIX



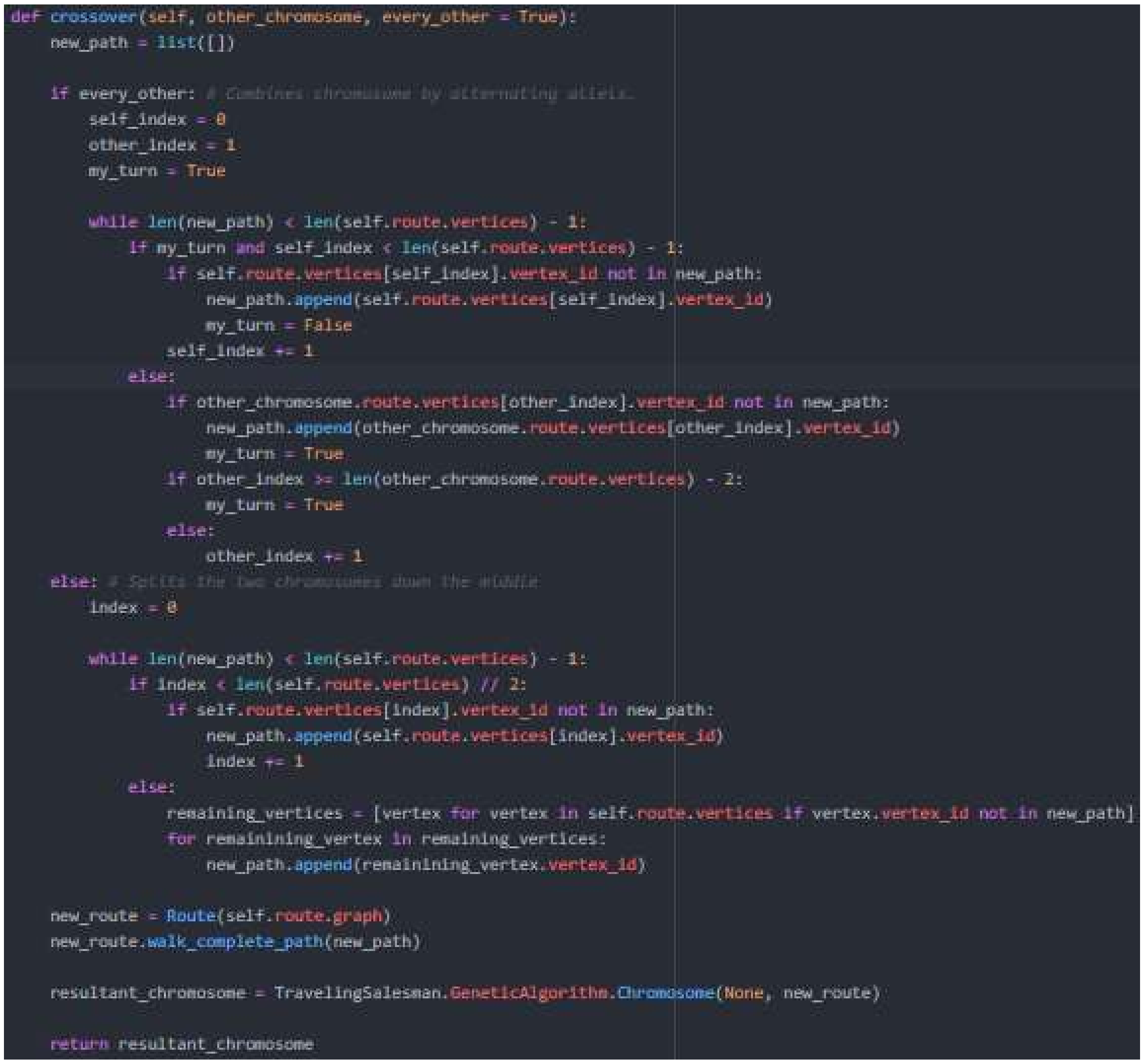
```python
def run(self, cross_over_every_other=True):
    print("Beginning Genetic Algorithm...")

    improvement = 0
    costs = list([])
    epochs_since_last_improvement = 0
    best_chromosome = min(self.population)
    all_time_best_chromosome = best_chromosome
    costs.append(best_chromosome.route.distance_traveled)

    while epochs_since_last_improvement < 30:
        print("Performing cross overs...")
        # Perform cross overs
        self.perform_crossovers(cross_over_every_other)

        print("Performing mutations...")
        # Perform mutations
        self.perform_mutations()

        # Get new best chromosome
        best_chromosome = min(self.population)

        improvement = all_time_best_chromosome.route.distance_traveled - best_chromosome.route.distance_traveled

        if improvement > 0:
            all_time_best_chromosome = best_chromosome
            epochs_since_last_improvement = 0
        else:
            epochs_since_last_improvement += 1
        costs.append(all_time_best_chromosome.route.distance_traveled)

    self.display_state()
    plt.plot(costs, label="distance traveled")
    plt.legend()
    plt.show()

    return best_chromosome.route
```

Figure 3 : Genetic Algorithm High Level Function



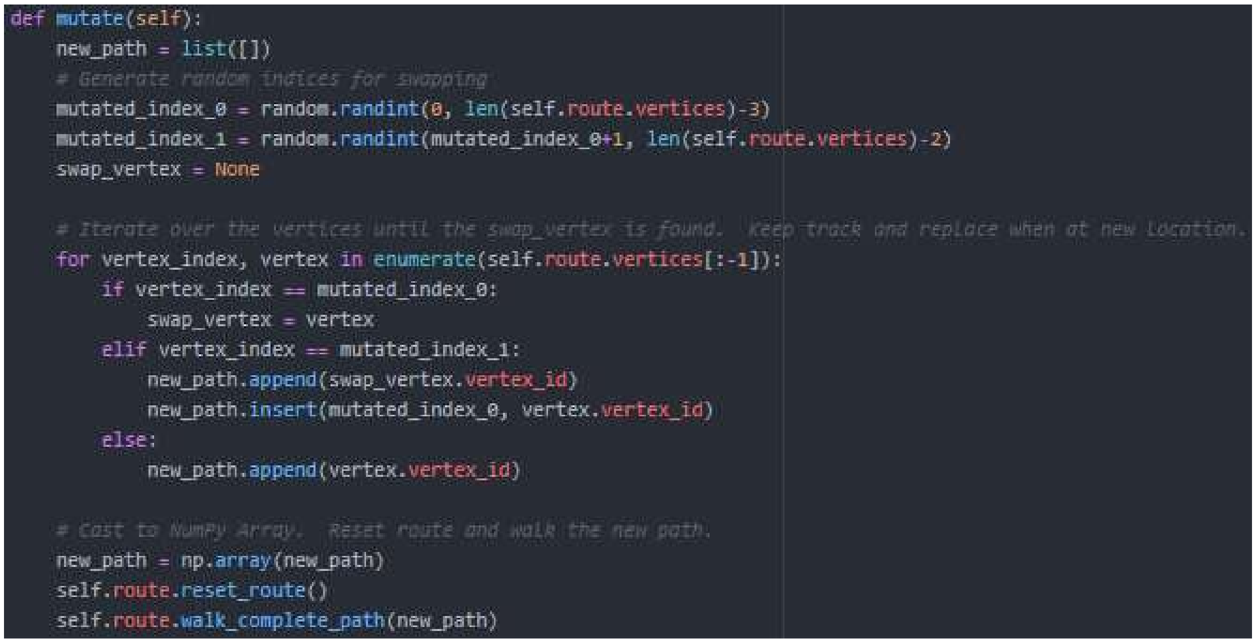
```python
def perform_crossovers(self, cross_over_every_other=True):
    chromosome_parent_population = deepcopy(self.population)
    chromosome_parent_population.sort()
    chromosome_parent_population = chromosome_parent_population[:int(len(chromosome_parent_population) * self.crossover_probability)]
    if len(chromosome_parent_population) < 2:
        # Not enough parents to do any cross overs
        pass
    else:
        children_to_replace = [child for child in self.population if child not in chromosome_parent_population]
        for chromosome in children_to_replace:
            random.shuffle(chromosome_parent_population)
            baby = chromosome_parent_population[0].crossover(chromosome_parent_population[1], cross_over_every_other)
            self.replace_chromosome(chromosome.chromosome_id, baby)
```

Figure 4 : Perform Crossover Method (From perspective of population)



```python
def crossover(self, other_chromosome, every_other = True):
    new_path = list([])

    if every_other: # Combines chromosome by alternating allels.
        self_index = 0
        other_index = 1
        my_turn = True

        while len(new_path) < len(self.route.vertices) - 1:
            if my_turn and self_index < len(self.route.vertices) - 1:
                if self.route.vertices[self_index].vertex_id not in new_path:
                    new_path.append(self.route.vertices[self_index].vertex_id)
                    my_turn = False
                self_index += 1
            else:
                if other_chromosome.route.vertices[other_index].vertex_id not in new_path:
                    new_path.append(other_chromosome.route.vertices[other_index].vertex_id)
                    my_turn = True
                if other_index >= len(other_chromosome.route.vertices) - 2:
                    my_turn = True
                else:
                    other_index += 1
    else: # Splits the two chromosomes down the middle
        index = 0

        while len(new_path) < len(self.route.vertices) - 1:
            if index < len(self.route.vertices) // 2:
                if self.route.vertices[index].vertex_id not in new_path:
                    new_path.append(self.route.vertices[index].vertex_id)
                    index += 1
            else:
                remaining_vertices = [vertex for vertex in self.route.vertices if vertex.vertex_id not in new_path]
                for remainining_vertex in remaining_vertices:
                    new_path.append(remainining_vertex.vertex_id)

    new_route = Route(self.route.graph)
    new_route.walk_complete_path(new_path)

    resultant_chromosome = TravelingSalesman.GeneticAlgorithm.Chromosome(None, new_route)

    return resultant_chromosome
```

Figure 5 : Crossover Method (from the perspective of a parent chromosome)



```
def mutate(self):
    new_path = list([])
    # Generate random indices for swapping
    mutated_index_0 = random.randint(0, len(self.route.vertices)-3)
    mutated_index_1 = random.randint(mutated_index_0+1, len(self.route.vertices)-2)
    swap_vertex = None

    # Iterate over the vertices until the swap_vertex is found.  Keep track and replace when at new location.
    for vertex_index, vertex in enumerate(self.route.vertices[:-1]):
        if vertex_index == mutated_index_0:
            swap_vertex = vertex
        elif vertex_index == mutated_index_1:
            new_path.append(swap_vertex.vertex_id)
            new_path.insert(mutated_index_0, vertex.vertex_id)
        else:
            new_path.append(vertex.vertex_id)

    # Cast to NumPy Array.  Reset route and walk the new path.
    new_path = np.array(new_path)
    self.route.reset_route()
    self.route.walk_complete_path(new_path)
```

Figure 6 : Mutate Method

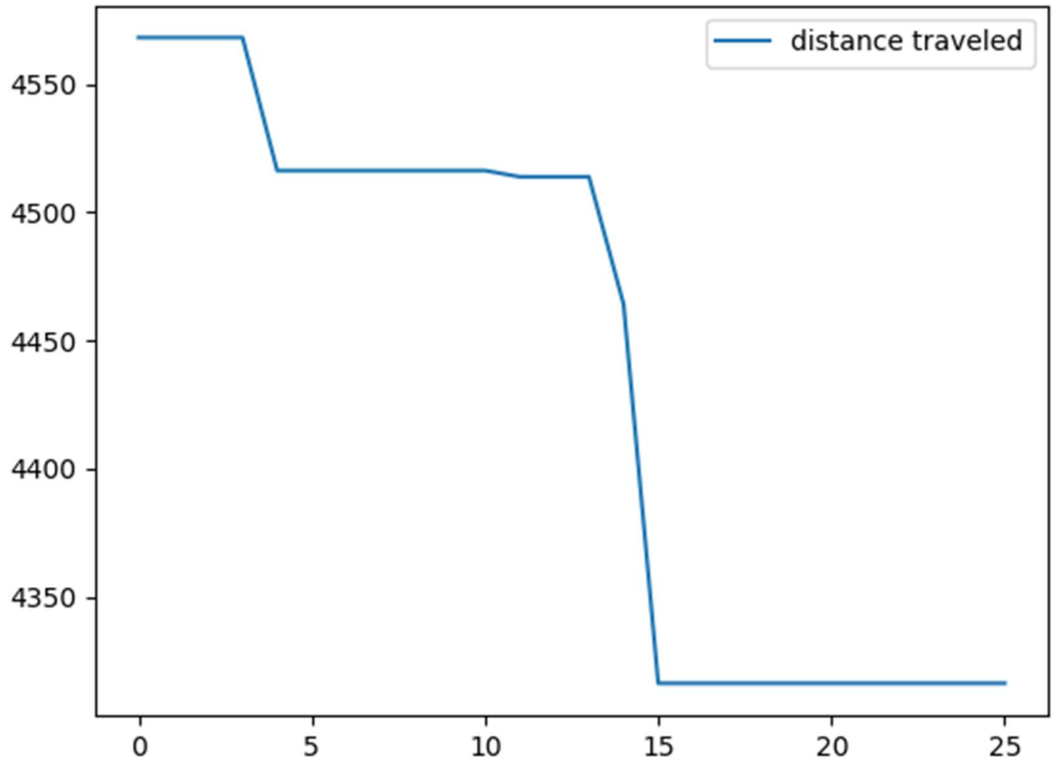


Figure 7 : COEO 50 Population Costs Run 1

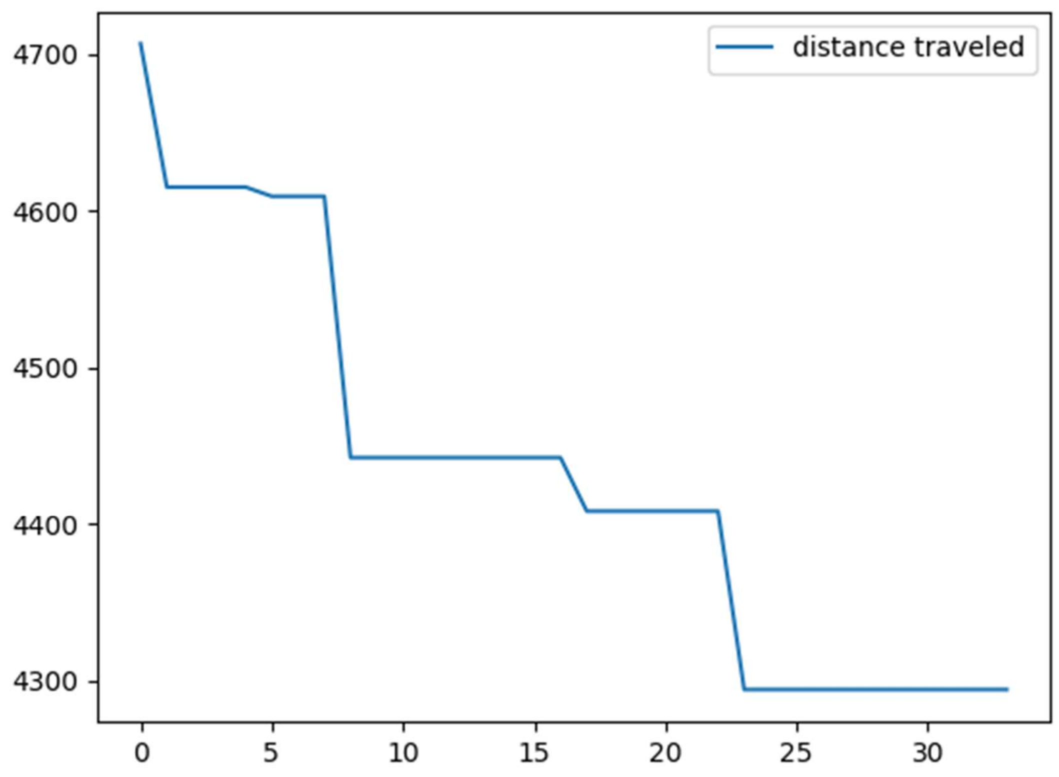


Figure 8 : COEO 50 Population Costs Run 2

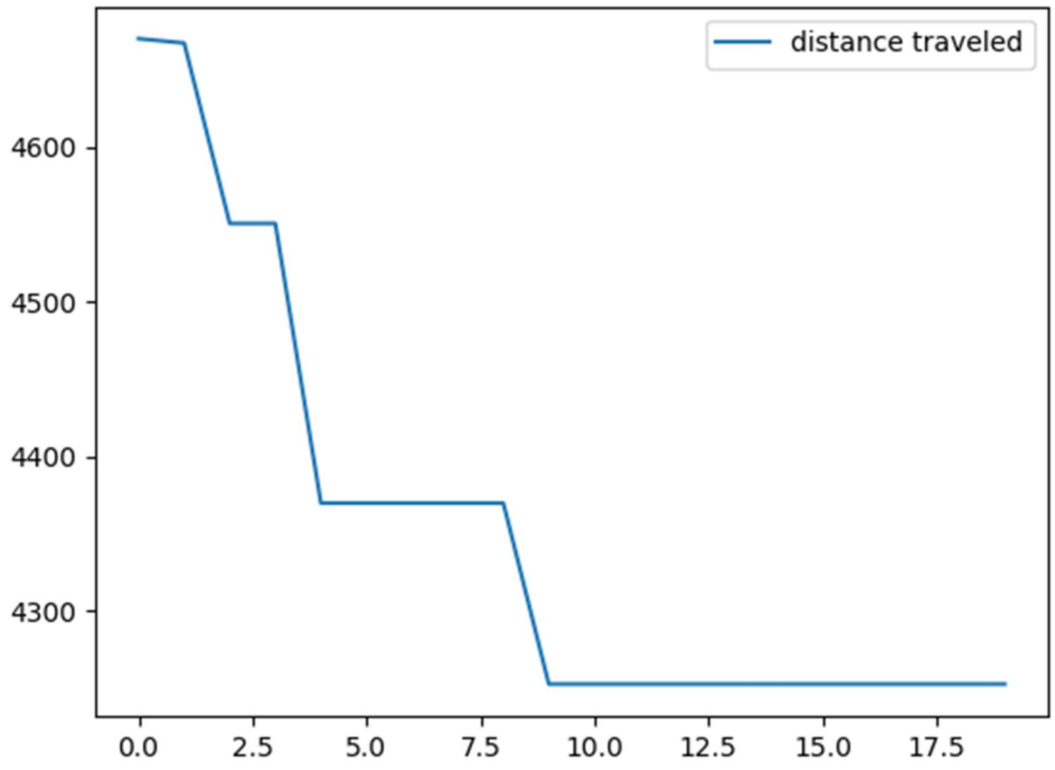


Figure 9 : COEO 100 Population Costs Run 1

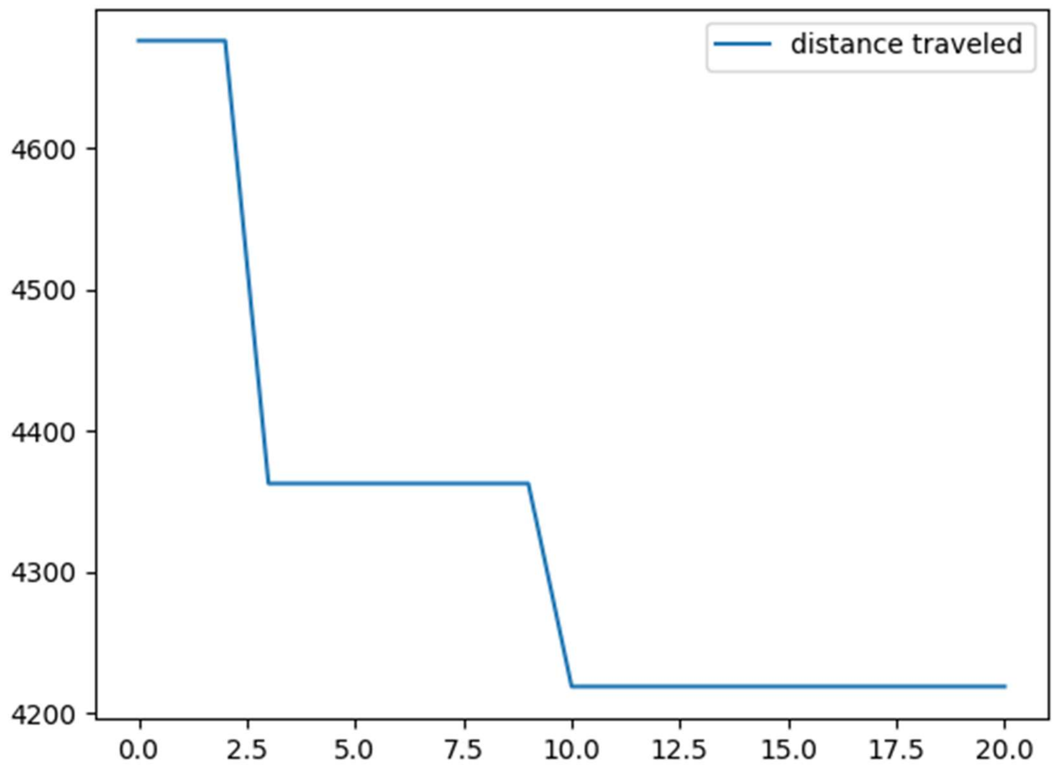


Figure 10 : COEO 100 Population Costs Run 2

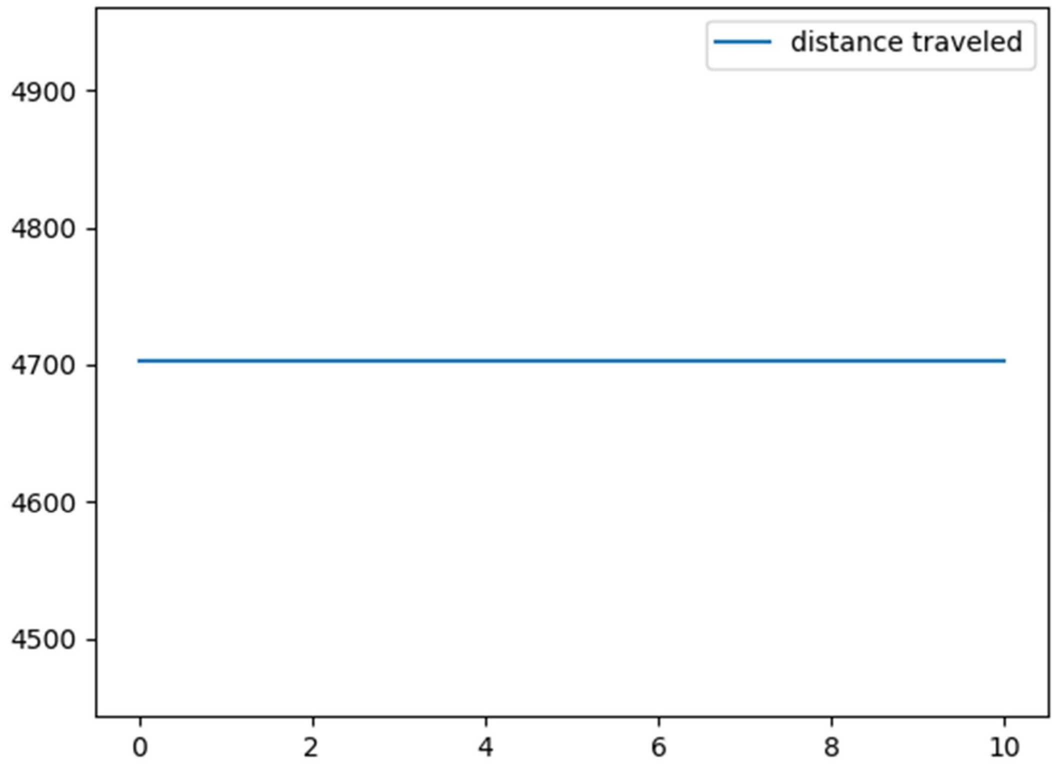


Figure 11 : Halfsies 50 Population Costs Run 1

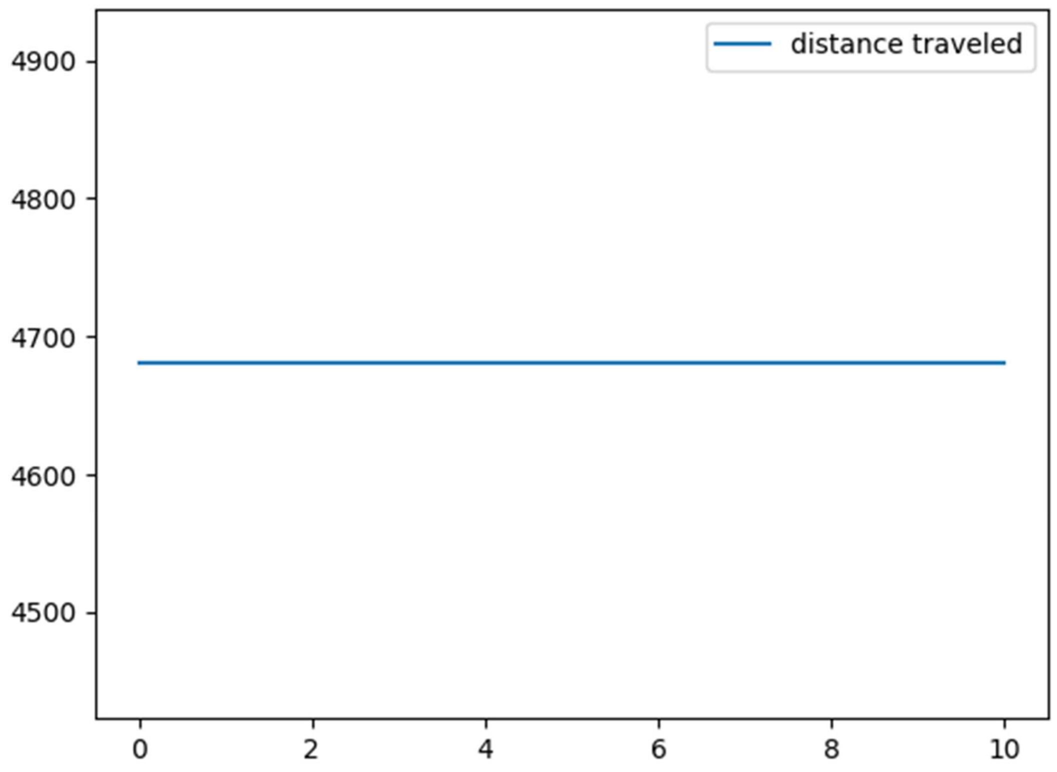


Figure 12 : Halfsies 50 Population Costs Run 1

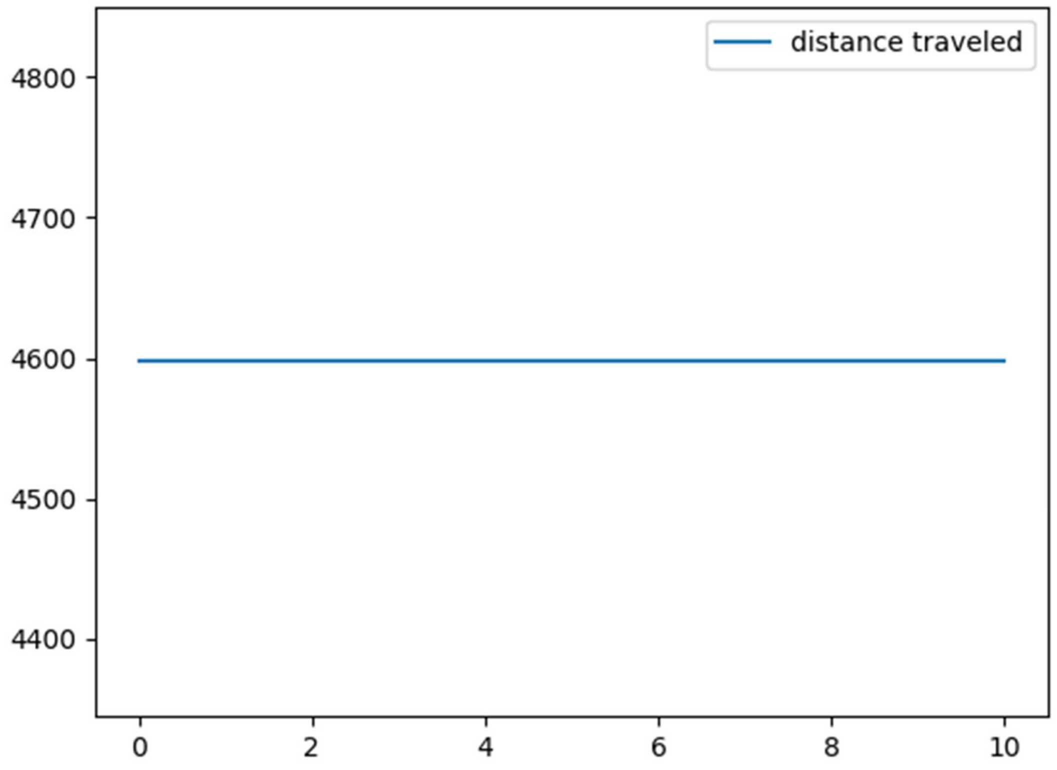


Figure 13 : Halfsies 50 Population Costs Run 2

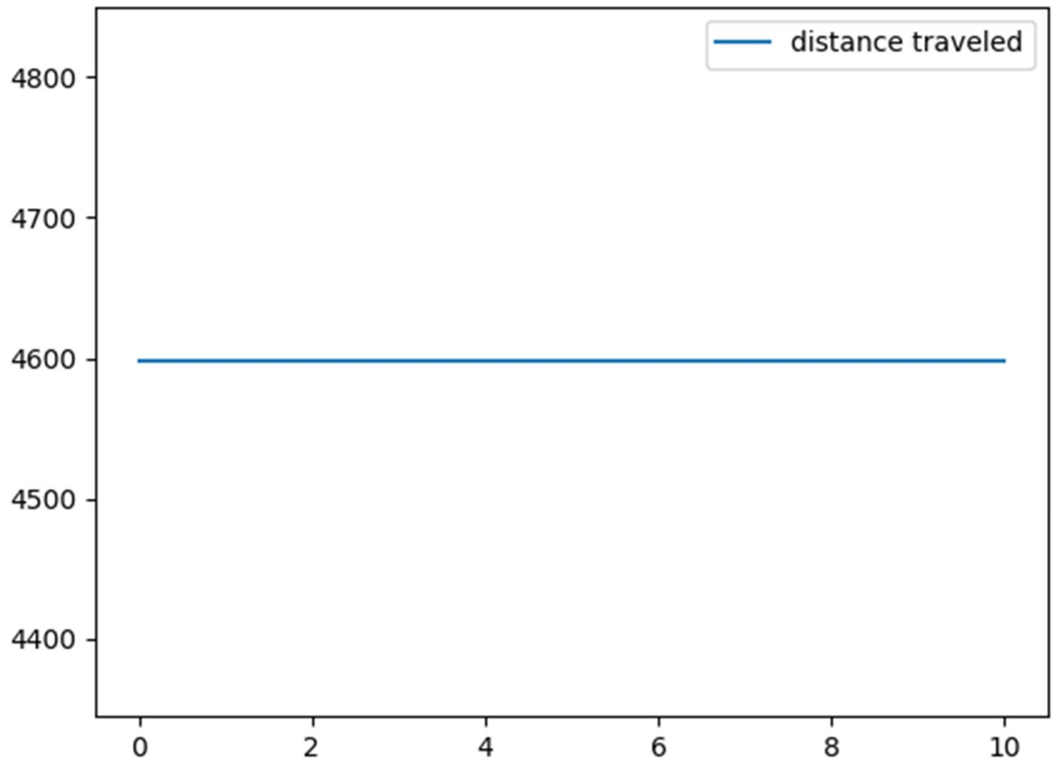


Figure 14 : Halfsies 100 Population Costs Run 1

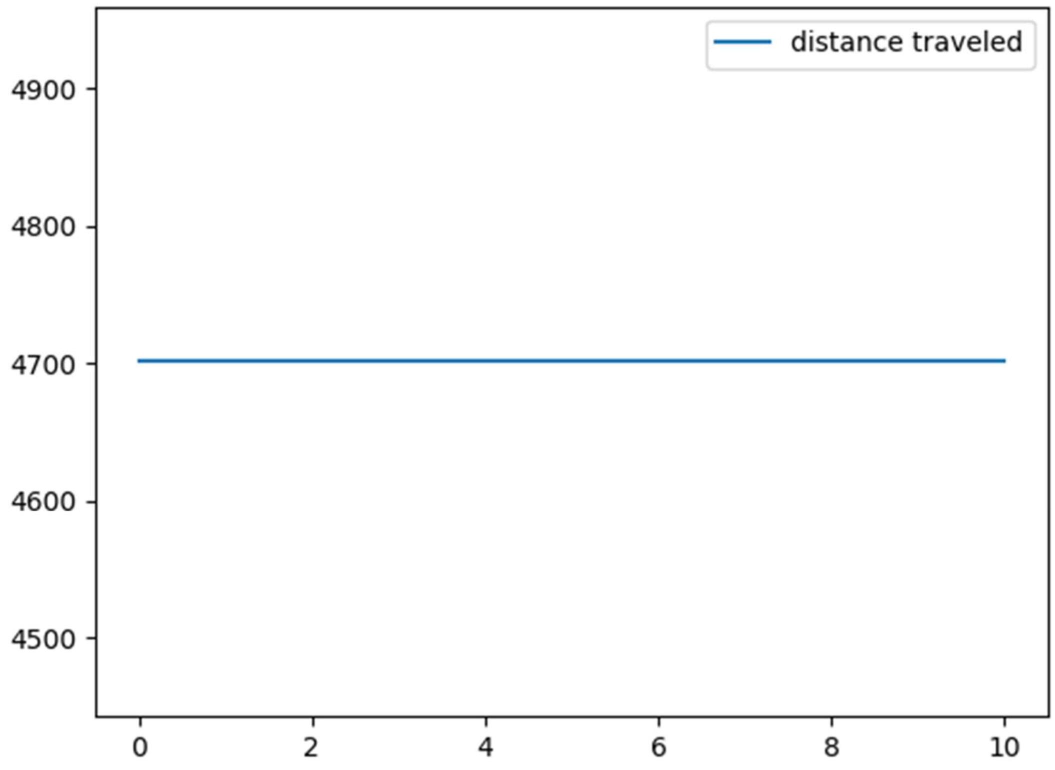


Figure 15 : Halfsies 100 Population Costs Run

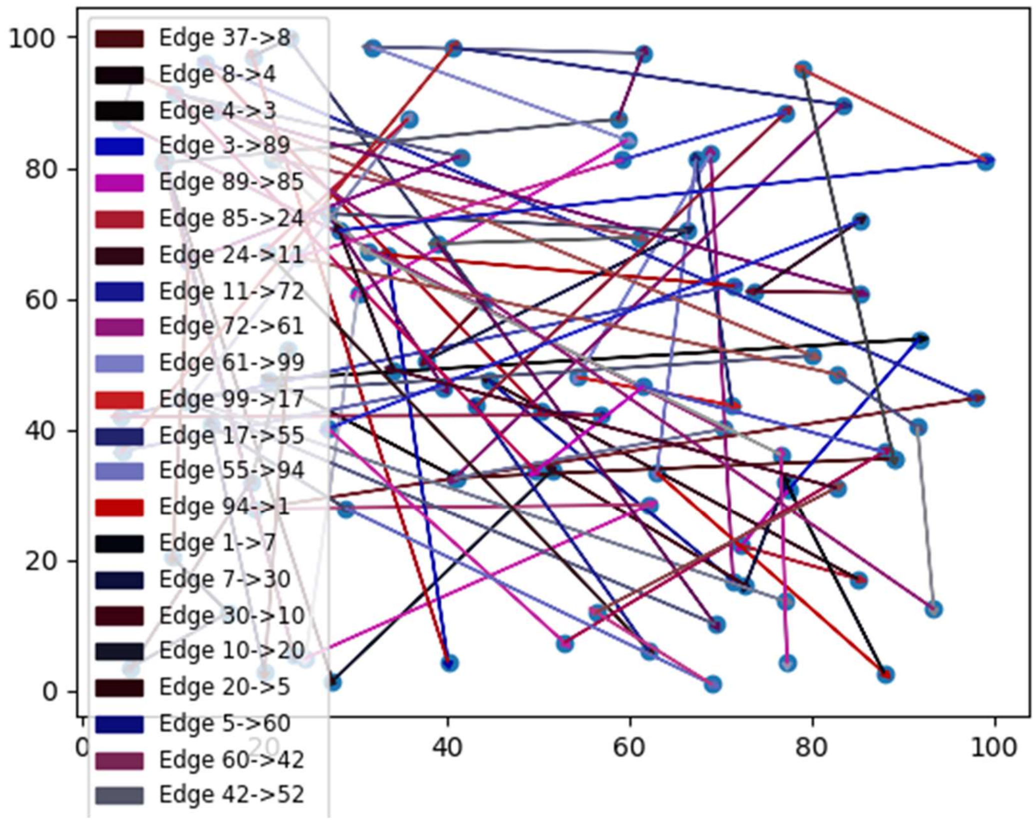


Figure 16 : COEO 50 Population Output Run 1

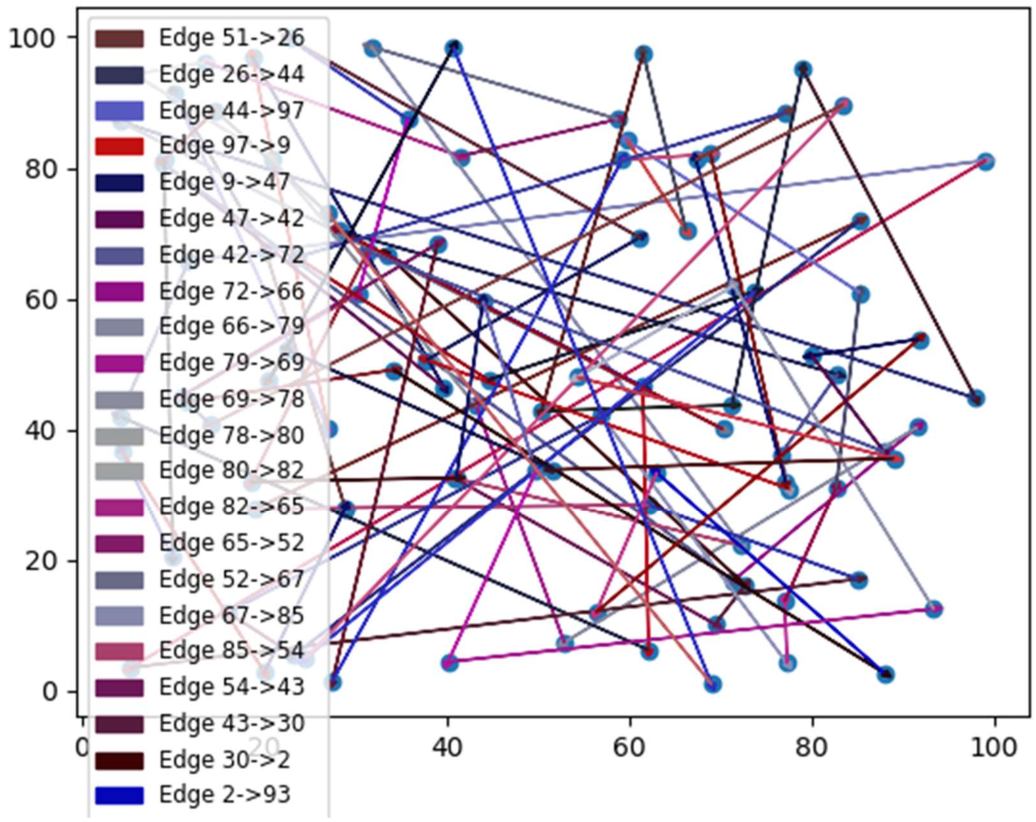


Figure 17 : COEO 50 Population Output Run 2

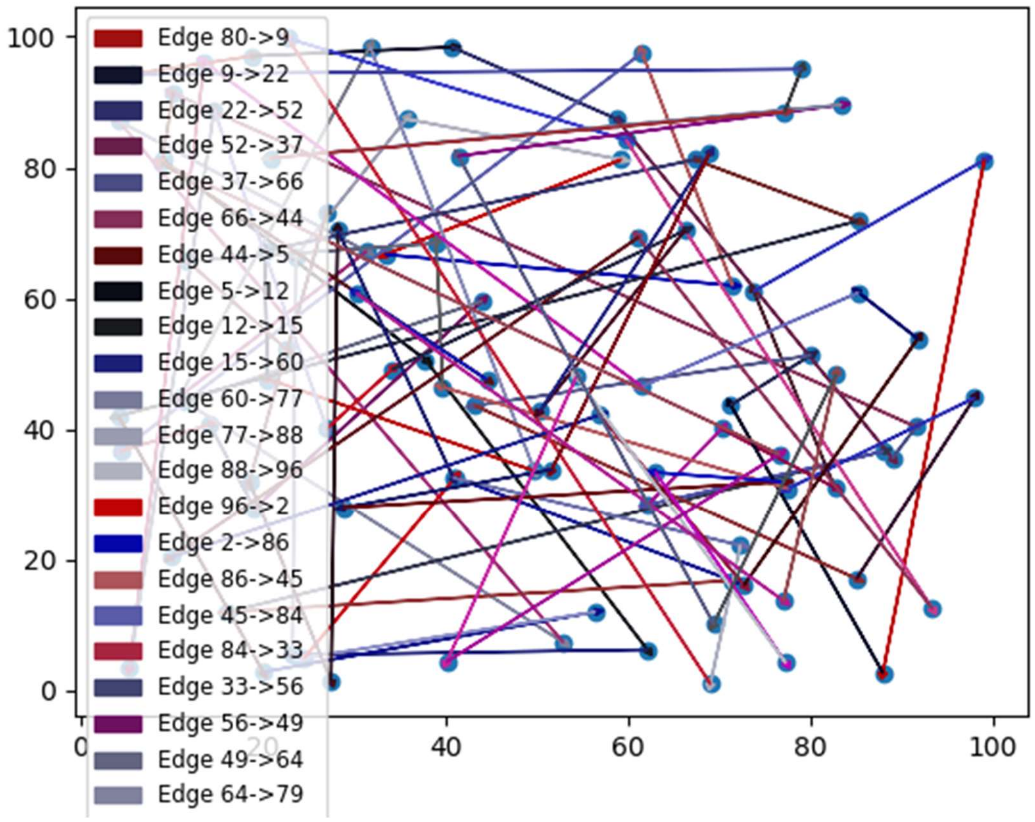


Figure 18 : COEO 100 Population Output Run 1

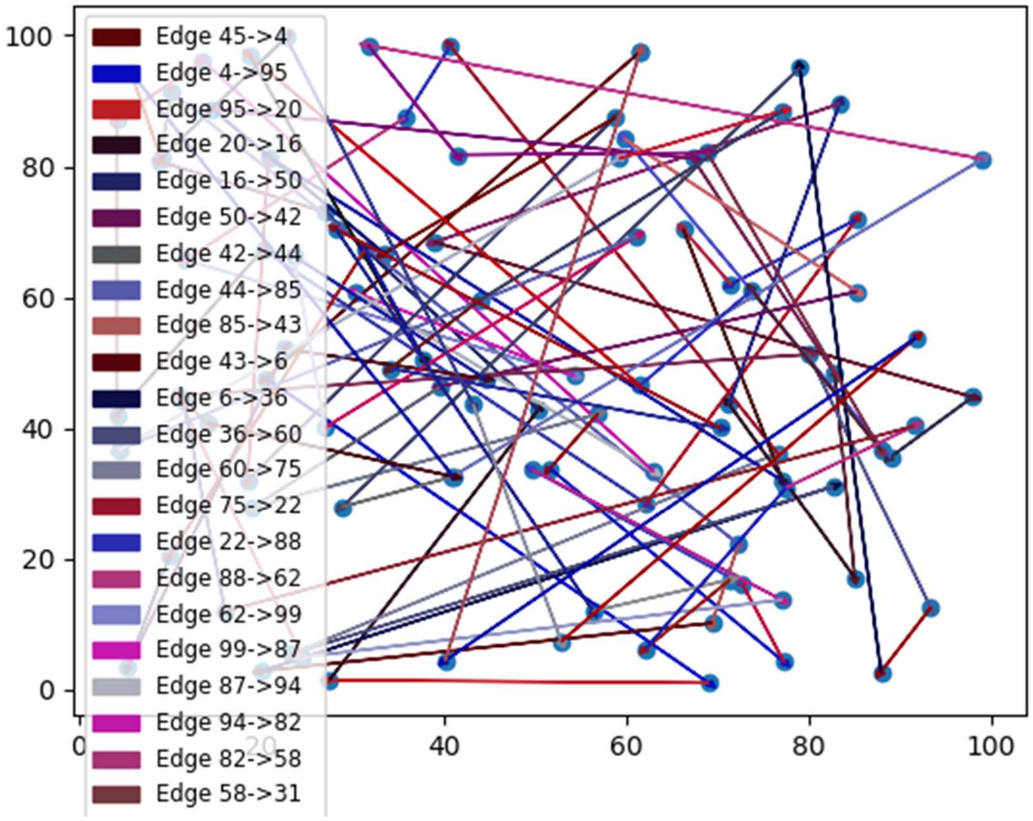


Figure 19 : COEO 100 Population Output Run 2

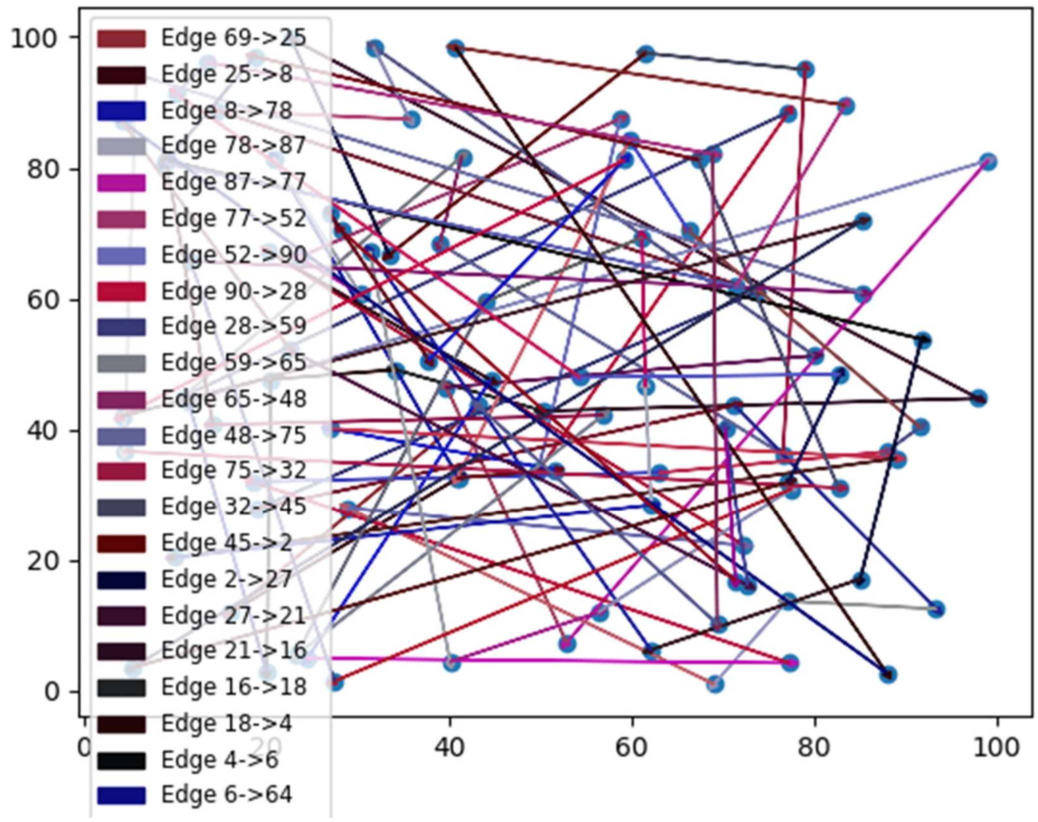


Figure 20 : Halfsies 50 Population Output Run 1

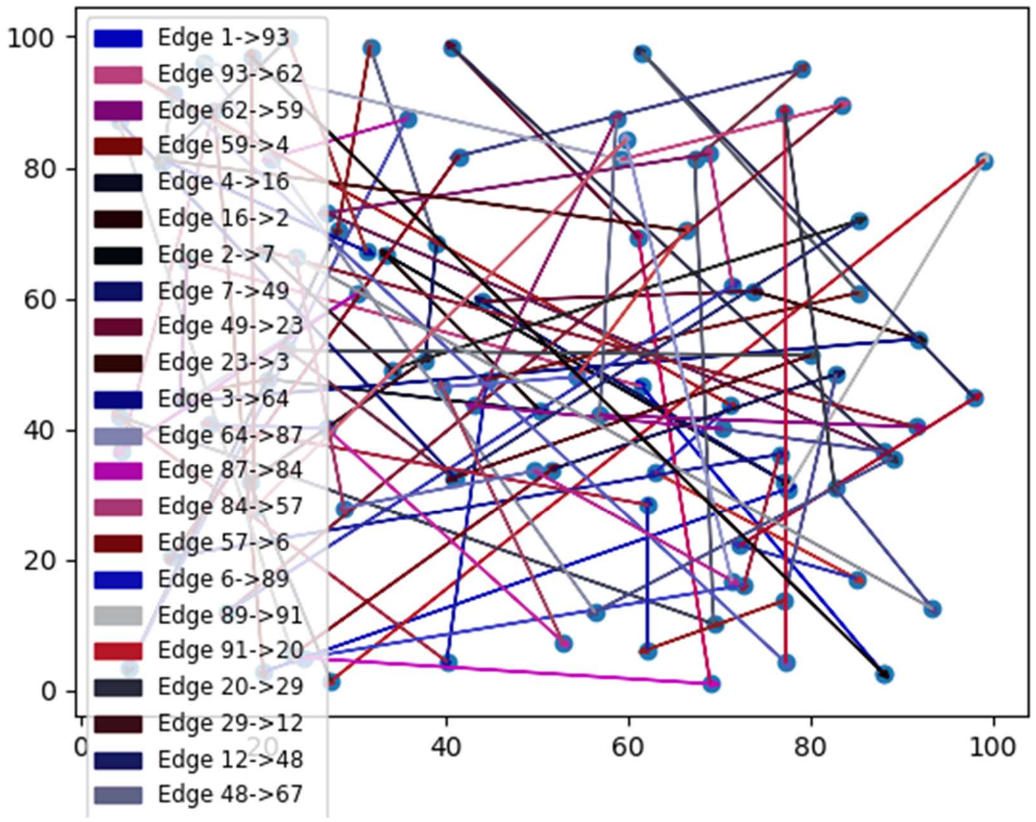


Figure 21 : Halfsies 50 Population Output Run 2

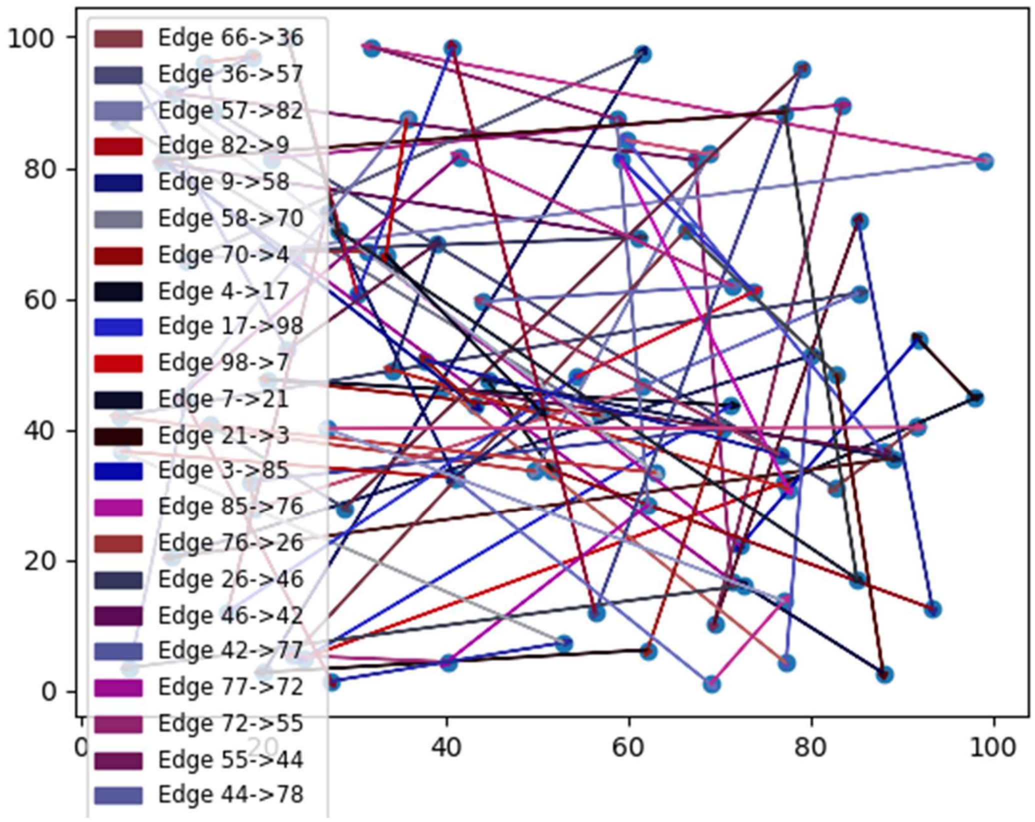


Figure 22 : Halfsies 100 Population Output Run 1

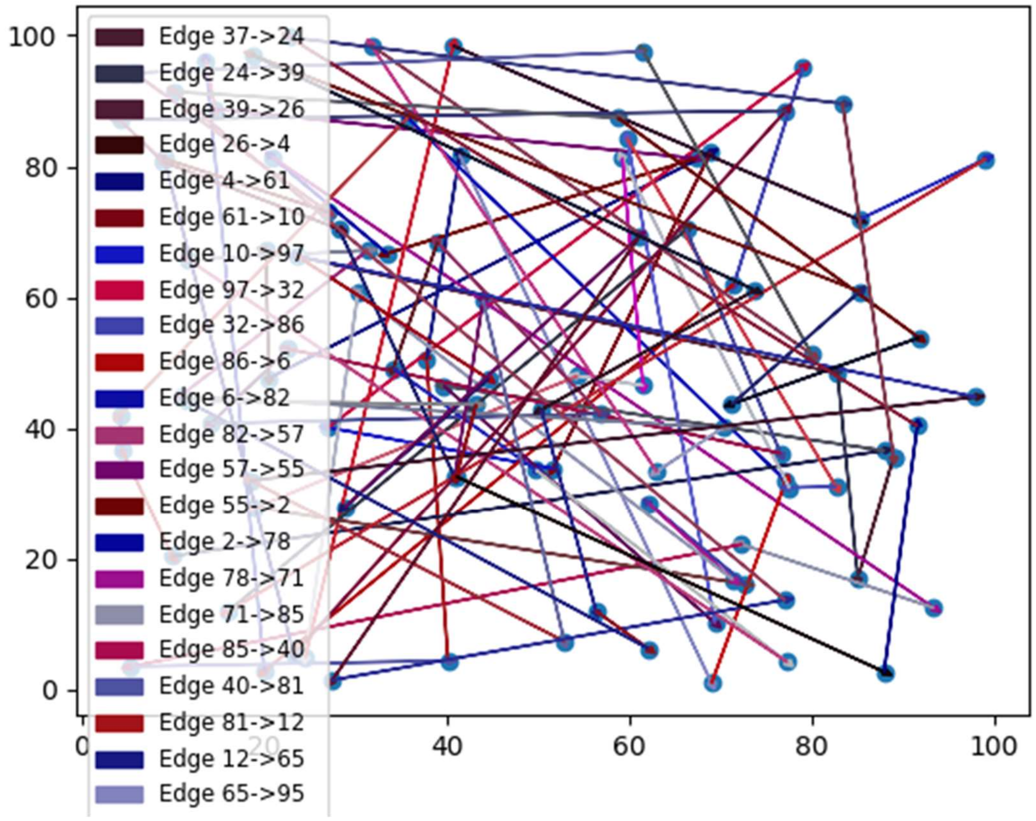


Figure 23 : Halfsies 50 Population Output Run 2