Vehicle Routing Problem: Wisdom of Crowds Using Genetic Algorithms

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

1 INTRODUCTION

The Vehicle Routing Problem (VRP) is a generalization of the well-known non-deterministic polynomial-time hard problem, the Traveling Salesman Problem. The VRP has been a topic of scientific publication since George Dantzig and John Ramser published "The Truck Dispatch Problem" in 1959. The context of the problem is that there are some number of depots each with some number of vehicles that must deliver goods to a set of customers with preference to traveling a shorter distance. The problem has many applications including path planning and classification.

2 APPROACH

The approach taken to solving the VRP was to use the "wisdom of crowds" principle coupled with a series of differing genetic algorithms. Each genetic algorithm was given an equal number of chromosomes. These chromosomes were represented as a one-way series of alleles or locations the vehicles must visit. To ease the problem, the current solution assumes there to be only one vehicle per depo; although a novel approach to designing a more complex genetic algorithm with multiple vehicles in mind is described in the discussion section of this document. Throughout this paper alleles inside of a chromosome may sometimes be referred to as vertices and the connections between the alleles as edges.

To advance the genetic algorithms, the chromosomes undergo a sequence of crossovers and mutations. After each mutation cycle, the chromosomes are inspected to see if any improvements have been made between generations. The comparison operator for the implementation presented in this paper was to select for the chromosomes with minimum total distance traveled. Once there have been more generations without improvement than some threshold, in this paper referred to as the epoch threshold, the genetic algorithm finishes. More information on the genetic algorithm's crossover and mutation methods is described in Section 2.1 of this document.

After all genetic algorithms within the crowd finish their cycles, the wisdom of crowd's algorithm aggregates the answer by looking at the frequency of edges in the series of genetic algorithm answers. To aid in this selection, a superiority threshold is used to weed out edges during aggregation. Edges that are found less frequent than this percentage threshold are left out of the aggregate. For tests described in this paper, the superiority threshold was kept at a constant 30% meaning that only edges that were in top 70% in occurrence frequency are considered in the aggregation. To break ties, the edges with the highest frequency are chosen over those that are less frequent and the shortest edge distance is chosen when two edges have equal frequency and similar starting or ending vertices.

2.1 GENETIC ALGORITHM

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.

2.1.1 Crossover Methods

The implemented crossover methods have been shown to improve performance in a genetic algorithmic approach to TSP (ABDOUN & ABOUCHABAKA, 2011). Each algorithm was run with an 80% crossover probability, meaning that with each generation the top 20% of the population was used to generate replacements for the bottom 80%.

2.1.1.1 Uniform Crossover

The uniform crossover forms a child by randomly alternating between the two parents. For reference to the implementation of this method please see **Figure 8** in the appendix.

2.1.1.2 Ordered Crossover

The ordered crossover breaks each parent into three sequences, S1, S2, and S3 with matching indices for both parents. The child is then produced by taking S2 from one parent and filling S1 and S3 with alleles from the other parents starting at S1 and leaping genes already included. For reference to the algorithm as formalized in the literature, please refer to **Figure 1**. For refence to the implementation of this method please see **Figure 9** in the appendix.

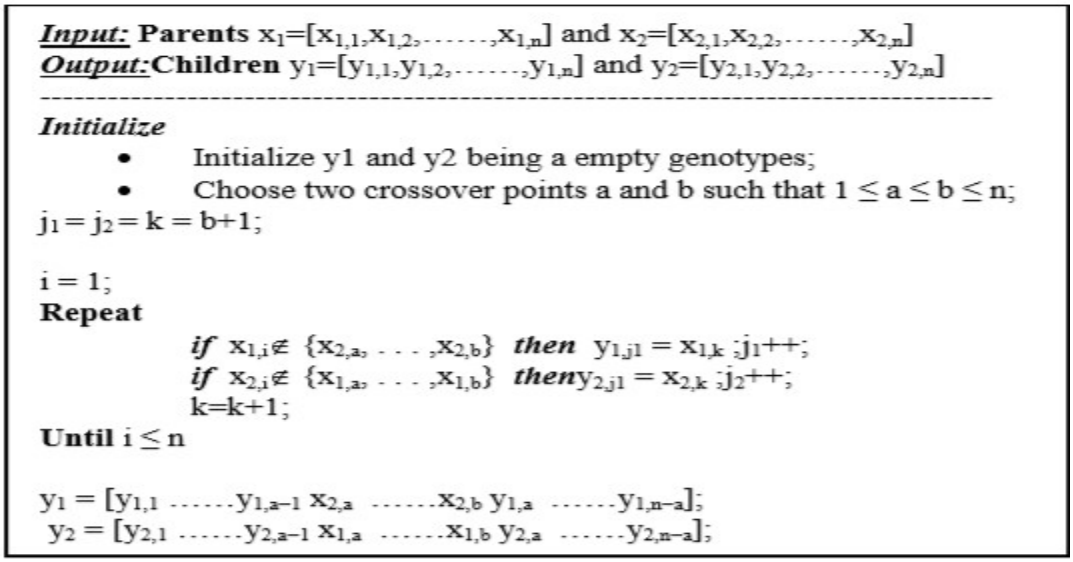

Figure 1 : Ordered Crossover Algorithm (ABDOUN & ABOUCHABAKA, 2011)

2.1.1.3 Partially Mapped (PM)

The partially mapped crossover breaks each parent into three sequences, S1, S2, and S3 with matching indices for both parents. The child is then produced by taking S1 and S3 from one parent and filling in S2 with alleles from the other parent starting at S2 and leaping genes already included. For reference to the algorithm as formalized in the literature, please refer to **Figure 2**. For reference to the implementation of this method please see **Figure 10** in the appendix.

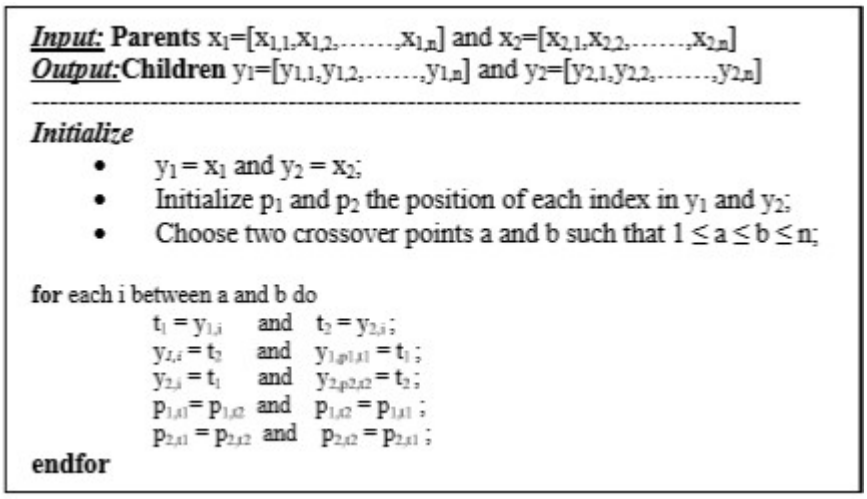

Figure 2 : Partially Mapped Algorithm (ABDOUN & ABOUCHABAKA, 2011)

2.1.2 Mutation Methods

The implemented mutation methods have been shown to improve performance in a genetic algorithmic approach to solving TSP (ABDOUN & ABOUCHABAKA, 2011). All genetic algorithms were run with a mutation rate of 2%, meaning that each generation had 2% of its chromosomes undergo mutation.

2.1.2.1 TWORS

The TWORS mutation method randomly swaps two alleles' locations within the chromosome. For reference to the implementation of this method, please see **Figure 11** in the appendix of this document.

2.1.2.2 Reverse Sequence (RSM)

The Reverse Sequence mutation method reverses the sequence of the chromosome. For reference to the implementation of this method, please see **Figure 12** in the appendix of this document.

2.2 GUI

A GUI was developed to visualize different stages of the algorithm. A heat map was developed to understand the crowd's edge frequency. Additionally, a route solution representation was generated to ensure proper connection of the final path.

2.2.1 Heat Map

Heat maps were generated to help understand the crowd's edge frequency. The edges were plotted with their RGB values denoting its frequency within the crowd. The most red colored edges are those that occur least frequent, while the most blue colored edges are those that occur most frequent. Please refer to **Figure 3** below for an example of a heat map generated from the edges with an occurrence rate in the top 80% for a set of 44 cities.

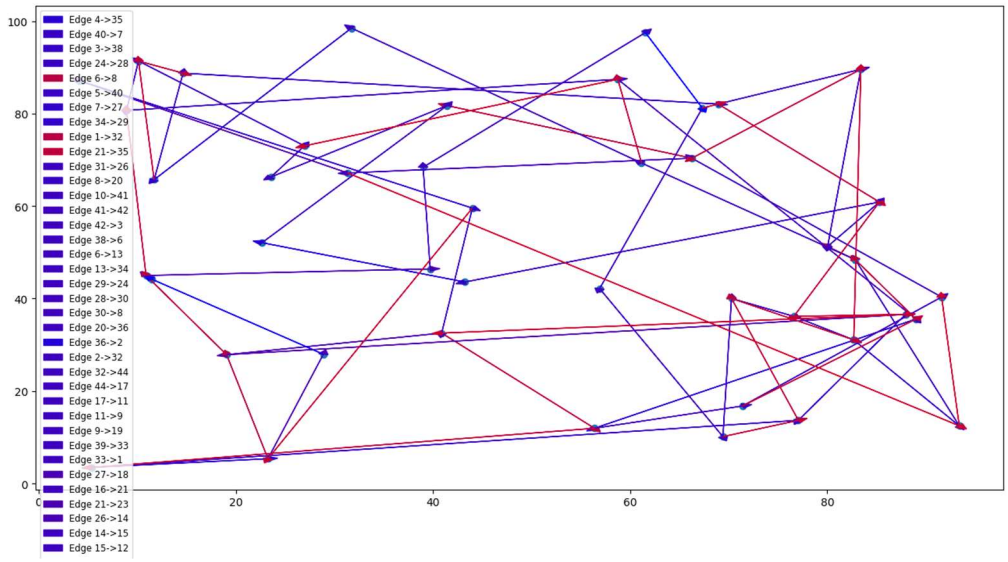

Figure 3 : Heat Map Random44.tsp 20% Superiority Threshold

2.2.2 Route Solution

A graphical representation of the final solution was generated to ensure it is reasonable. Each depot's route was graphed separately like the route shown in **Figure 4** below. Each edge is colored using its vertices' IDs, assigned by enumeration in dataset, to quantize its red and blue color magnitudes while the green magnitude is calculated from the modulus of the starting vertex id with respect to the ending vertex id. For reference to the implementation of this plotting method, please refer to **Figure 13** in the appendix.

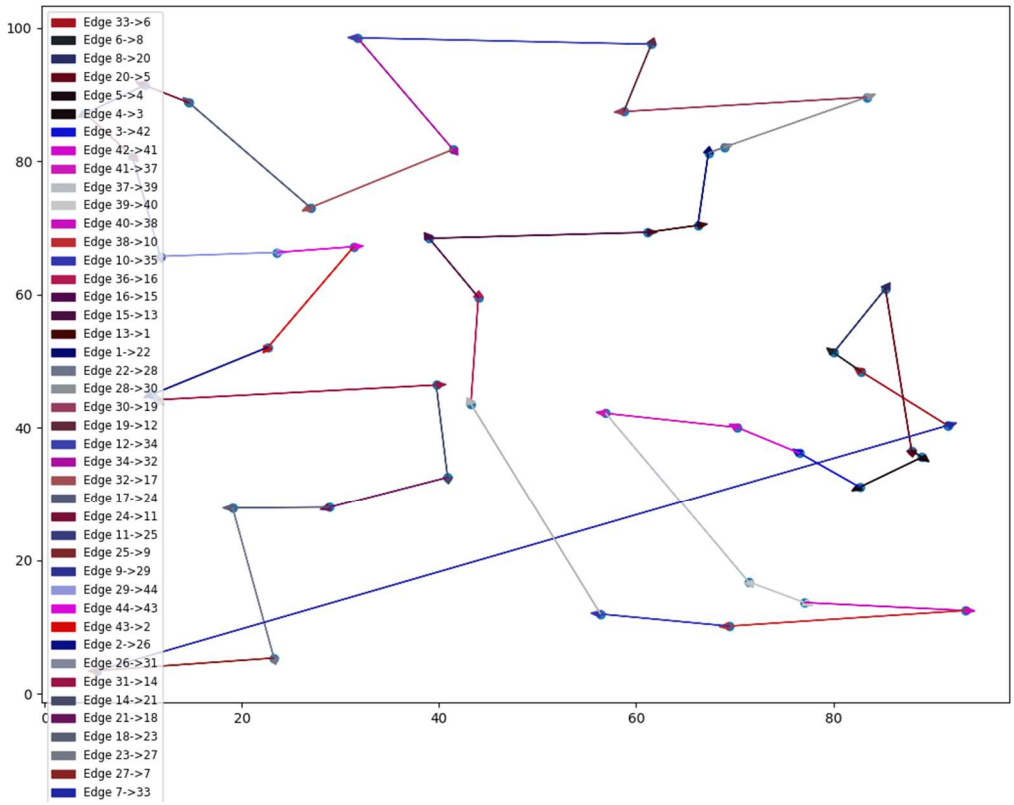

Figure 4 : Route Solution Random44.tsp

2.3 GREEDY HEURISTIC

As noted above, to combine the solutions of the crowd of genetic algorithms the algorithm creates a dictionary of edges across the entire crowd and keeps track of the frequency of each edge. Edges that meet some predetermined "superiority threshold" are kept to develop a fragmented graph. For edges that contain the same vertex, the edge with the highest edge count and lowest distance traveled is retained. The relevant code for creating the fragmented graph is shown in **Figure 14** in the appendix of this document.

Once a recombination route is generated, there could still be some stray vertices. The remaining vertices are iterated over and the nearest vertex to an existing route segment is chosen and "lassoed" into the route segment. For reference to the route's lasso function please refer to

Figure 15 in the appendix of this document. For reference to the method for choosing the next vertex to insert into the group of route segments, please refer to **Figure 16** in the appendix of this document.

Lastly, once all vertices are part of a route segment, a recombine method is used to connect the route segments into a contiguous route. This is done by iterating over the unvisited starting indices and connecting them in order as they appear in the list of edges generated from the lassoing of vertices. For reference to the implementation of this segment recombination method, please refer to **Figure 17** in the appendix of this document.

3 RESULTS

The Wisdom of Crowds with Genetic algorithms was successfully implemented to improve upon the approximation of an optimal solution to the Vehicle Routing problem assuming there is some number of depots with some number of customers and only one vehicle per depot. There was a recurring error produced when attempting to process some of the larger datasets from Python's deepcopy function found in their standard copy library. This is likely due to limitations imposed by the low RAM on the test computer.

3.1 DATA

The algorithms were tested on different datasets ranging from 11 cities to 222. The dataset files were generated randomly. 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 5** below.

```
NAME: concorde44
TYPE: TSP
COMMENT: Generated by CCutil_writetsplib
COMMENT: Write called for by Concorde GUI
DIMENSION: 44
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 66.258736 70.360424
2 22.656941 52.076785
3 82.680746 31.058687
4 88.995025 35.560167
5 87.939085 36.567278
6 82.845546 48.393200
7 5.371258 3.466903
8 80.028687 51.258889
9 8.908353 80.703146
10 69.411298 10.122990
```

Figure 5: Random44.tsp Input File Format

To simulate the vehicle routing problem having some number of depots and some number of customers, the number of depots and customers were provided to the algorithm. A set of depots was then chosen at random from the set of vertices. The customers for each depot was then further chosen at random with the understanding that no depot vertex can be its own customer. This can be further refined to include an N number of vehicles that break the route into N number of sub routes.

3.2 RESULTS

Due to the memory issue described above, test results were only gathered for datasets of sizes 11 to 97 cities with most tests occurring on graphs with 22, 44, and 77 cities. Successful runs were achieved on each of the datasets described but tables were only developed for specific tests that highlight the strengths and weaknesses of the wisdom of crowd's solution purposed in this document. A list of all the tests and the results can be found on the GitHub repository for this project.

To use as a benchmark in comparison with the wisdom of crowd's solution, tests were run using different variations of crossover methods and mutation methods. Each test listed here used the same population size, number of depots, and number of customers. Since the location of the depots and the location of the customers was chosen at random at runtime, the results are presented as relative values to the average of the entire crowd. This average calculation includes the wisdom of crowd's average along with the average of separate genetic algorithm ran on the same set of vertices, depots, and customers.

Table 1 shows the results of a survey of 20 tests each with the same population size per genetic algorithm with 10 chromosomes for each genetic algorithm. They also encompass epoch threshold sizes of 25 and 50. There were no significant changes in algorithm performance by doubling the epoch threshold. Each test listed in Table 1 had only one depot with 20 customers. The crossover probabilities and mutation probabilities were sampled from respective sets of values [0.2, 0.4, 0.6, 0.8] and [0.01, 0.1, 0.25, 0.5] with the best results being found with crossover probability of 0.8 and mutation probability of 0.5. **Figure 6** shows a graph of **Table 1** with the x-axis referring to an enumeration matching the row number of the algorithm while the y-axis refers to the algorithm's average runtime over the 20 surveyed tests.

Table 1 : Random22.tsp Survey of 20 Tests (1 Depot, 20 Customers)

ALGORITHM	average result	average runtime	average improvement	tests better than average	Tests Surveyed	Percent Superior to Avg
GA_UNIFORM_TWORS	860.4853784	0.990563631	10.92039594	10	20	50%
GA_UNIFORM_RSM	874.7368728	1.024454045	25.17189041	4	20	20%
GA_PM_TWORS	871.600102	1.002755773	22.03511956	6	20	30%
GA_PM_RSM	874.5010733	1.10004847	24.93609087	5	20	25%
GA_ORDERED_TWORS	862.469915	1.184431171	12.90493256	8	20	40%
GA_ORDERED_RSM	866.7959331	1.002625942	17.23095063	5	20	25%
WisdomOfCrowds_GA	736.3656025	6.730173063	-113.19938	19	20	95%
AVERAGE_FOR_ALGORITHMS	849.5649824	1.8621503	0	-	-	-

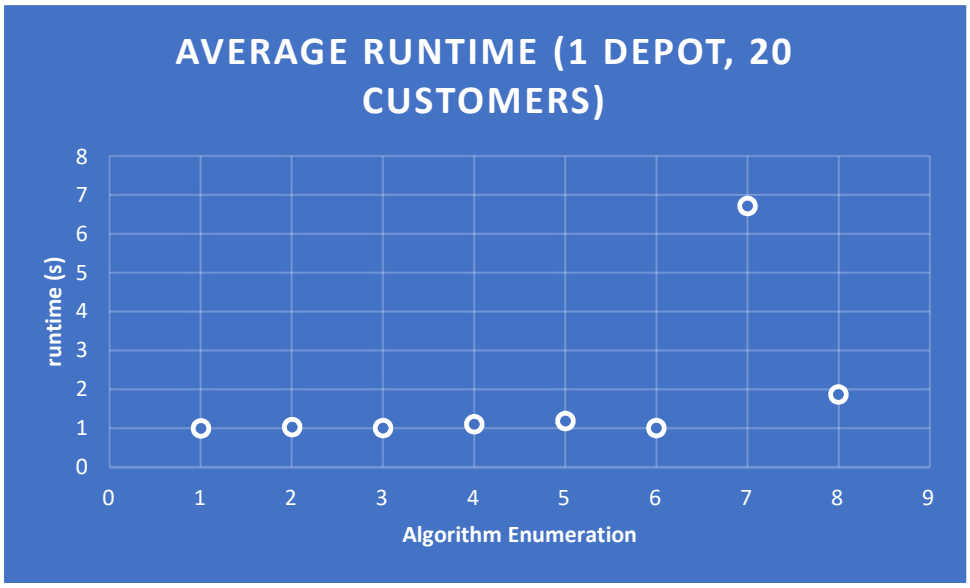

Figure 6 : Random22.tsp (1 Depot, 20 Customers) Algorithm Runtime Analysis

Table 2 shows the results of a survey of 20 tests each with a population size of 10 chromosomes per genetic algorithm and an epoch threshold of 25. Each test listed in **Table 2** had 19 Depot locations each with 20 Customers. The crossover probabilities and mutation probabilities were sampled from respective sets of values [0.2, 0.4, 0.6, 0.8] and [0.01, 0.1, 0.25, 0.5] with the best results being found with crossover probability of 0.8 and mutation probability of 0.5. **Figure 7** shows a graph of **Table 2** with the x-axis referring to an enumeration matching the row number of the algorithm while the y-axis refers to the algorithm's average runtime over the 20 surveyed tests.

Table 2 : Random22.tsp Survey of 20 Tests (19 Depots, 20 Customers)

ALGORITHM	average result	average runtime	average improvement	tests performing better than average	Tests Surveyed	Percent Superior to Avg
GA_UNIFORM_TWORS	16541.17224	19.3154446	378.9850493	0	20	0%
GA_UNIFORM_RSM	16438.16202	19.63145066	275.9748215	1	20	5%
GA_PM_TWORS	16520.74594	19.42928503	358.5587507	0	20	0%
GA_PM_RSM	16496.21312	18.6354623	334.0259264	0	20	0%
GA_ORDERED_TWORS	16431.534	19.41496243	269.3468112	2	20	10%
GA_ORDERED_RSM	16454.62301	19.2843079	292.4358171	2	20	10%
WisdomOfCrowds_GA	14252.86002	118.2461229	-1909.327176	20	20	100%
AVERAGE_FOR_ALGORITHMS	16162.18719	33.42243369	-	-	-	-

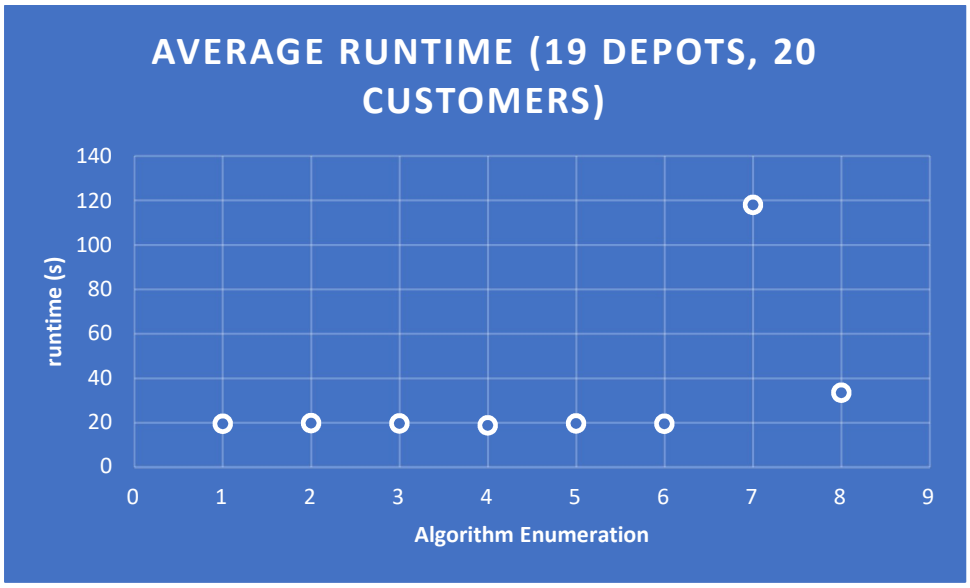

Figure 7 : Random22.tsp (19 Depot, 20 Customers) Algorithm Runtime Analysis

4 DISCUSSION

4.1 RESULTS

As shown in section 3 of this document, the wisdom of crowd's solution outperformed the average of all algorithms on the same problem 97.5% of the tests surveyed. In the first set of tests surveyed, the wisdom of crowd's solution performed better than the average on 19 of the 20 tests surveyed with the one poor performer occurring with a test having relatively low crossover probability, 0.2, and mutation probability 0.25. All surveyed tests with enough cross over

probability and mutation probability saw improvements in distance traveled from the wisdom of crowd's solution when compared to the standard genetic algorithms at the cost of runtime. This additional runtime is likely a consequence of overhead from the wisdom of crowd's solution having to initialize a set of genetic algorithms as well as combine the solutions of the crowd.

The performance difference described above was even further expressed when there were additional depots as you can see in the differences between **Tables 1** and **2**. With 19 depots, instead of 1, the algorithm performed much better than average, improving the average result by 11.8%. This further expression of improvement is likely the result of compounding improvement across each of the depot's route. This difference in improvement was magnified even further when testing larger numbers of customers and depots as expected; although, the errors occurring from a lack of RAM made performing many these tests for validation unreasonable given the time frame.

4.2 Future Work

The initial idea formulating the chromosomes and fitness function for this problem was to parallelize all depot's routes as well as their vehicles by concatenating each depots route into one long chromosome. This chromosome would have alleles equal to the dot product of a vector of number of depos and a vector of number of customers. The crossover and mutation methods would then happen on sections of these chromosomes, for instance RMS would piece wise reverse the chromosome at each route instead of reversing the entire sequence.

The chromosome's fitness function can be redefined from the total distance traveled across the chromosome's alleles into a series of smaller routes, each traveled by a different vehicle. To illustrate this, imagine if there were 3 depots each with 2 vehicles and 12 customers. The chromosome would be 39 alleles long with the first 13 representing the first depot. Of those 13, the first allele represents the depot's location while the next 6 represent the route one vehicle will take, and last 6 represent the route the other vehicle will take.

This reformulation of the VRP has not yet been implemented because of a misunderstanding when first approaching the problem. The current solution assumes there is only one vehicle and performs each depot's route optimization in serial instead of parallel like this section describes. These improvements would likely reduce runtime across all algorithms although, they may also have a negative effect on the algorithm's performance given the set of crossover and mutation methods used.

5 References

ABDOUN, O., & ABOUCHABAKA, J. (2011, October). A Comparative Study of Adaptive Crossover Operators for Genetic Algorithms to Resolve the Traveling Salesman Problem. *International Journal of Computer Applications, 31*(11). Retrieved from https://arxiv.org/ftp/arxiv/papers/1203/1203.3097.pdf

Baraglia, R., Hidalgo, J. I., & Perego, R. (2001, December). A Hybrid Heuristic for the Traveling Salesman Problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, 5*(6), 613-622. doi:10.1109/4235.974843

Yi , S. M., Steyvers , M., Lee, M. D., & Dry , M. J. (2011). Wisdom of the Crowds in Traveling Salesman Problems.

The Vehicle Routing Problem - https://andresjaquep.files.wordpress.com/2008/10/2627477-clasico-dantzig.pdf
Wikipedia, Traveling Salesman Problem - https://en.wikipedia.org/wiki/Travelling_salesman_problem#History
Wikipedia, Vehicle Routing Problem - https://en.wikipedia.org/wiki/Vehicle_routing_problem
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

```
if self.crossover_method == GeneticAlgorithm.Chromosome.CrossoverMethods.UNIFORM:
    self_turn = True

    while len(new_path) < len(self.route.vertices)-1:
        if self_turn:
            remaining_vertex_ids = [vertex.vertex_id for vertex in self.route.vertices if vertex.vertex_id not in new_path]
            if len(remaining_vertex_ids) > 0:
                new_path.append(random.choice(remaining_vertex_ids))

            self_turn = False
        else:
            remaining_vertex_ids = [vertex.vertex_id for vertex in other_chromosome.route.vertices if vertex.vertex_id not in new_path]
            if len(remaining_vertex_ids) > 0:
                new_path.append(random.choice(remaining_vertex_ids))

            self_turn = True
```

Figure 8 : Uniform Crossover Method

```
elif self.crossover_method == GeneticAlgorithm.Chromosome.CrossoverMethods.PARTIALLY_MAPPED:
    p1 = random.randint(1, len(self.route.vertices)-3)
    p2 = random.randint(p1+1, len(self.route.vertices)-2)

    self_ids = [vertex.vertex_id for vertex in self.route.vertices][:-1]
    self_s1 = self_ids[:p1]
    self_s2 = self_ids[p1:p2]
    self_s3 = self_ids[p2:]

    other_ids = [vertex.vertex_id for vertex in other_chromosome.route.vertices][:-1]
    other_s1 = other_ids[:p1]
    other_s2 = other_ids[p1:p2]
    other_s3 = other_ids[p2:]

    new_path = self_s1

    s2_left = list([])
    for vertex_id in other_s2:
        if vertex_id not in self_s1 and vertex_id not in self_s3:
            s2_left.append(vertex_id)

    s3_left = list([])
    for vertex_id in other_s3:
        if vertex_id not in self_s1 and vertex_id not in self_s3:
            s3_left.append(vertex_id)

    s1_left = list([])
    for vertex_id in other_s1:
        if vertex_id not in self_s1 and vertex_id not in self_s3:
            s1_left.append(vertex_id)

    remaining_vertex_ids = s2_left + s3_left + s1_left
    new_path += remaining_vertex_ids[:p2-p1]

    new_path += self_s3
```

Figure 9 : Partially Mapped Crossover Method

```
elif self.crossover_method == GeneticAlgorithm.Chromosome.CrossoverMethods.ORDERED_CROSSOVER:
    p1 = random.randint(1, len(self.route.vertices)-3)
    p2 = random.randint(p1+1, len(self.route.vertices)-2)
    j_1 = p1 + 1
    j_2 = j_1
    k = j_1

    to_p1 = self.route.vertices[:p1]
    from_p1 = self.route.vertices[p1:]
    mid = other_chromosome.route.vertices[p1:p2+1]

    for vertex in to_p1:
        if vertex not in mid:
            new_path.append(vertex.vertex_id)

    for vertex in mid:
        new_path.append(vertex.vertex_id)

    for vertex in from_p1:
        if vertex.vertex_id not in new_path:
            new_path.append(vertex.vertex_id)
```

Figure 10 : Ordered Crossover Method

```
if self.mutation_method == GeneticAlgorithm.Chromosome.MutationMethods.TWORS:
    # 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)
```

Figure 11 : TWORS Mutation Method

```
elif self.mutation_method == GeneticAlgorithm.Chromosome.MutationMethods.REVERSE_SEQUENCE_MUTATION:
    for vertex in np.flip(self.route.vertices[:-1]):
        new_path.append(vertex.vertex_id)
```

Figure 12 : Reverse Sequence Mutation Method

```
def plot(self):
    x = list([])
    y = list([])
    plots = list([])
    arrow_plots = list([])
    arrow_labels = list([])

    # Iterate over vertices, retrieving x and y coordinates
    for vertex in self.vertices:
        x.append(vertex.x)
        y.append(vertex.y)

    # Plot the vertices
    vertex_plot = plt.scatter(x, y, label="Vertices")
    plots.append(vertex_plot)

    # Plot the route
    for edge in self.edges:
        vertex = edge.vertices[0]
        adjacent_vertex = edge.vertices[1]

        arrow_label = "Edge {}->{}".format(vertex.vertex_id, adjacent_vertex.vertex_id)
        arrow_plot = plt.arrow(vertex.x, vertex.y, adjacent_vertex.x-vertex.x, adjacent_vertex.y-vertex.y,
                               head_width=1, head_length=1,
                               color='#{}{}{}'.format(Math.color_quantization(vertex.vertex_id, len(self.vertices)),
                                                      Math.color_quantization(vertex.vertex_id % adjacent_vertex.vertex_id + 1, len(self.vertices)),
                                                      Math.color_quantization(adjacent_vertex.vertex_id, len(self.vertices))),
                               label=arrow_label)
        arrow_labels.append(arrow_label)
        arrow_plots.append(arrow_plot)

    # Show the graph with a legend
    plt.legend(arrow_plots, arrow_labels, loc=2, fontsize='small')
    plt.show()
```

Figure 13 : Route Solution Plotting Method

```
# Update route to match current representation given superiority_tolerance
superiority_edges = [(edge_key, edge_entry) for (edge_key, edge_entry) in self.edge_dictionary.items() if edge_entry.edge_count >= (self.max_edge_count * superiority_tolerance)]

for edge_key, edge_entry in superiority_edges:
    better_edge = False
    for edge_key_1, edge_entry_1 in superiority_edges:
        if edge_entry.edge.vertices[0].vertex_id == edge_entry_1.edge.vertices[0].vertex_id or edge_entry.edge.vertices[1].vertex_id == edge_entry_1.edge.vertices[1].vertex_id:
            if edge_entry.edge_count == edge_entry_1.edge_count:
                if edge_entry.edge.distance > edge_entry_1.edge.distance:
                    better_edge = True
            elif edge_entry.edge_count < edge_entry_1.edge_count:
                better_edge = True

    if not better_edge:
        if self.route.edges is None:
            self.route.add_edge(edge_entry.edge)
        else:
            if not self.edge_create_circular_path(edge_entry.edge):
                self.route.add_edge(edge_entry.edge)
self.route.distance_traveled = self.route.recount_distance()
```

Figure 14 : Relevant Code Route Recombination

```
    def lasso(self, vertex, closest_item_to_next_vertex):
        if isinstance(closest_item_to_next_vertex, Edge):
            # Get v1, v2
            edge_vertex1 = closest_item_to_next_vertex.vertices[0]
            edge_vertex2 = closest_item_to_next_vertex.vertices[1]
            edge_v2_v3 = None
            v3 = None

            # use v2's index to get v3
            edge_vertex2_index = np.where(self.vertices == edge_vertex2)[0]
            if edge_vertex2_index < len(self.vertices) - 1:
                v3 = self.vertices[edge_vertex2_index+1][0]

            # Calculate different edge distances.
            v1_v2 = closest_item_to_next_vertex.distance
            if edge_vertex2_index < len(self.vertices) - 2 and v3 is not None:
                ## NEED TO TAKE CARE OF THE CASE WHEN V3 has not been visited.

                v1_v2_v0_v3 = v1_v2 + Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), vertex.get_location()) + \
                                                                    Math.calculate_distance_from_point_to_point(vertex.get_location(), v3.get_location())
                v1_v0_v2_v3 = Math.calculate_distance_from_point_to_point(edge_vertex1.get_location(), vertex.get_location()) + \
                                                                Math.calculate_distance_from_point_to_point(vertex.get_location(), edge_vertex2.get_location()) + \
                                                                Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), v3.get_location())
            else:
                v1_v2_v0_v3 = v1_v2 + Math.calculate_distance_from_point_to_point(edge_vertex2.get_location(), vertex.get_location())
                v1_v0_v2_v3 = Math.calculate_distance_from_point_to_point(edge_vertex1.get_location(), vertex.get_location()) + \
                                                                Math.calculate_distance_from_point_to_point(vertex.get_location(), edge_vertex2.get_location())

            # Choose the shortest configuration
            if v1_v2_v0_v3 < v1_v0_v2_v3: # Best to insert it after edge_vertex2 in vertices list
                # Calculate new vertex location in list and insert it
                new_vertex_location = np.where(self.vertices == edge_vertex2)[0] + 1
                self.vertices = np.insert(self.vertices, new_vertex_location, vertex)

                # calculate the new edge's location
                edge_v1_v2 = [edge for edge in self.edges if edge == closest_item_to_next_vertex][0]
                edge_v1_v2_index = np.where(self.edges == edge_v1_v2)[0]
                new_edge_location = edge_v1_v2_index + 1

                if v3 is not None:
                    # create edge v0_v3 and insert it.  Remove edge v2_v3
                    if new_vertex_location < len(self.vertices)-1:
                        for edge in self.edges:
                            if edge.vertices[0].vertex_id == edge_vertex2.vertex_id and edge.vertices[1].vertex_id == v3.vertex_id:
                                edge_v2_v3 = edge
                        if edge_v2_v3 is None:
                            edge_v2_v3 = Edge(edge_vertex2, v3)
                        self.edges = self.edges[self.edges != edge_v2_v3]
                        self.distance_traveled -= edge_v2_v3.distance
                        edge_v0_v3 = Edge(vertex, v3)
```

```
                        self.edges = np.insert(self.edges, new_edge_location, edge_v0_v3)
                        self.distance_traveled += edge_v0_v3.distance

                    # create edge v2_v0 and insert it
                    edge_v2_v0 = Edge(edge_vertex2, vertex)
                    self.edges = np.insert(self.edges, new_edge_location, edge_v2_v0)
                    self.distance_traveled += edge_v2_v0.distance
                else: # Best to insert it before edge_vertex2 in vertices list
                    # Calculate new vertex location in list and insert it
                    new_vertex_location = np.where(self.vertices == edge_vertex2)[0]
                    self.vertices = np.insert(self.vertices, new_vertex_location, vertex)

                    # Calculate v1_v2 edge index for reference
                    edge_v1_v2 = [edge for edge in self.edges if edge == closest_item_to_next_vertex][0]
                    edge_v1_v2_index = np.where(self.edges == edge_v1_v2)[0]

                    # create edges and insert them
                    edge_v1_v0 = Edge(edge_vertex1, vertex)
                    edge_v0_v2 = Edge(vertex, edge_vertex2)
                    self.edges = np.insert(self.edges, edge_v1_v2_index, edge_v0_v2)
                    self.edges = np.insert(self.edges, edge_v1_v2_index, edge_v1_v0)

                    # Remove unnecessary edge
                    self.edges = self.edges[self.edges != edge_v1_v2]

                    # Update distance
                    self.distance_traveled -= edge_v1_v2.distance
                    self.distance_traveled += edge_v1_v0.distance
                    self.distance_traveled += edge_v0_v2.distance
            elif isinstance(closest_item_to_next_vertex, Vertex):
                self.goto(vertex)

            # Set vertex to be true.
            vertex.visited = True
```

Figure 15 : Route Lasso Method

```
def choose_next_vertex():
    closest_item_next_to_closest_vertex = None
    r_type_of_closest_item = None
    closest_vertex = None
    closest_distance = None
    starting_vertex = self.route.vertices[0]
    remaining_vertices = [vertex for vertex in vertices if vertex not in self.route.vertices]

    for vertex in remaining_vertices:
        closest_item_next_to_vertex, item_distance = self.route.get_shortest_distance_to_route(vertex)

        if closest_vertex is None:
            closest_vertex = vertex
            closest_distance = item_distance
            closest_item_next_to_closest_vertex = closest_item_next_to_vertex
        else:
            if item_distance < closest_distance:
                closest_distance = item_distance
                closest_vertex = vertex
                closest_item_next_to_closest_vertex = closest_item_next_to_vertex

    if len(remaining_vertices) == 0:
        return self.route.vertices[0], self.route.vertices[-1]
    else:
        return closest_vertex, closest_item_next_to_closest_vertex


while len(self.route.vertices) < len(vertices):
    next_vertex, closest_item_next_to_vertex = choose_next_vertex()
    self.route.lasso(next_vertex, closest_item_next_to_vertex)
```

Figure 16 : Relevant Code Greedy Choose Next Vertex

```
def greedy_recombine(self):
    vertices = self.vertices
    self.reset_route()
    for vertex in vertices:
        self.goto(vertex)
    self.goto(vertices[0])
```

Figure 17 : Segment Recombine Method