Algorithms for The Travelling Salesman Problem

Eric Gastineau
Georgia Institute of Technology
Atlanta,GA, United States
egastineau3@gatech.edu

Jacob Beel
Georgia Institute of Technology
Atlanta,GA, United States
jbeel3@gatech.edu

Yiwen Bu
Georgia Institute of Technology
Atlanta,GA, United States
ybu@gatech.edu

Yong Jian Quek
Georgia Institute of Technology
Atlanta,GA, United States
yjquek@gatech.edu

ABSTRACT

The travelling salesman problem (TSP) is a well-known NP-Hard optimization problem which asks the question: "Given a list of cities and the distances between each pair of cities, what is the shortest route that visits each city and returns to the origin city?" Dating back to as early as 1832, the problem continues to be of research interest to this day. In this paper we present four algorithms (one exact and three approximate) for finding solutions to TSP. These are: branch and bound, nearest neighbor, a genetic local search algorithm, and simulated annealing. We cover these algorithms in detail, discuss worst-case time and space requirements and approximation ratios. We also perform empirical analyses of all four methods and present our findings.

KEYWORDS

travelling salesman problem, approximation, simulated annealing, genetic, branch and bound, nearest neighbor

1 INTRODUCTION

The Travelling Salesman Problem is a prominent NP-Hard problem in computer science and had been studied from as early the 1930s [7] and has not stopped ever since. The problem asks: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?".

The problem serves as a good testbed for various algorithms due to its nature as an NP-Hard problem.For this study, we tested 4 different approaches

Exact Algorithm	Branch and Bound
Construction Heuristics	Nearest Neighbor
Local Searches	Simulated Annealing
	Genetic Algorithm

From our research, both Branch and Bound and simulated annealing performed well on small datasets but struggled with larger datasets. The genetic algorithm achieves a good quality solution quickly, but then struggles to move forward. Using the nearest neighbor construction heuristic provided us with the bes results in a short amount of time but has no grantees on finding the true optimum solution.

2 PROBLEM DEFINITION

The TSP problem is defined as follows: given the x-y coordinates of N points in the plane (i.e.,vertices), and a cost function c (u, v) defined for every pair of points (i.e., edge), find the shortest simple cycle that visits all N points. This version of the TSP problem is metric , i.e. all edge costs are symmetric and satisfy the triangle inequality.

3 RELATED WORK

3.1 Branch and Bound

The first time that Branch and Bound was used for the traveling salesman problem was in the work of Little et al [8]. The lower bound was calculated by reducing the matrix of distances (which means substracting the smallest element of a row for every element of that row and then doing the same for the columns). The amount of reduction constitutes the lower bound and each time that we choose a new edge, we update the matrix and we reduce again. They found that the computation time grows exponentially.

3.2 Approximation Heuristic

Nearest Neighbor is a natural fit to a first attempt to solve the Travelling Salesman problem. [9] proved that nearest neighbor does not, in fact, have a constant approximation ratio. Instead, the approximation ratio of this algorithm scales with $log(n)$, which we will prove later. A high level description of the algorithm can be found in its name: When adding a node to the tour, add the nearest neighbor. While this does tend to produce a relatively good approximation, there are dangers to using it. Nearest neighbour tends to "forget" certain nodes as the algorithm proceeds and has to add them it at very high cost at the end. In rare cases, this algorithm can perform extremely poorly [4] but it has the benefit of being exceptionally simple and time-efficient.

3.3 Local Search

Local Search algorithms have been widely used for the TSP problem. For instance, in 1958, Croes introduced the 2-opt algorithm which has been really successful [1]. Genetic Algorithm has been used for the first time for the TSP in 1988 by Fogel [3]. He found that this approach has limits and that it eventually ends in evolutionary stagnation.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Overview. We want to find a sequence of cities with a an order that minimize the length of the path. The solution is then in the form of a list where each adjacent element are connected and where the last element of the list is connected with the first element. The final list has to be of size N (the number of points in the plane). We could have tested all the permutations possible but it would have taken too long time (N! permutations). That is why we are using Branch And Bound that is aimed to explore all possible permutations that are relevant.

We begin with a root node (a list with only one point), and then we expand the tree by adding a new point to the list. We calculate the lower bound of this new list and if the lower bound is superior to the current solution, we prune the subtree by removing the last point added. If the lower bound is inferior to the current solution, we recursively expand the problem. If we find a new solution better than the current solution, then this solution becomes the current solution.

We use the Minimum Spanning Tree (calculated with the - Kruskal's algorithm) to find the lower bound of the partial solution.

4.1.2 Strengths and Weaknesses. Branch and Bound ensures an exact solution if we wait long enough because we try every relevant potential solution. However, this approach has a really bad time complexity.

4.1.3 Data structure used.

- An input array Nx2 that represents the list of cities and their coordinates.
- NxN matrix that represents the distances between each pair of points.
- A NxN matrix where each row represents the list of points sorted according to their distances to a specific point (each row represents one point).
- A dynamic list that represents the solution. Each adjacent elements are connected and the last element of the list is connected with the first element.
- A list that contains every edges so that we can sort them for the Kruskal's algorithm. There is also a dictionary so that we can find an edge thanks to its distance.
- A dictionary that stores the past Minimum Spanning Trees already calculated so that we don't calculate them a second time.

4.1.4 Time and space complexities. In order to get the time complexity we have to take into account the worst case. And the worst case is a case here we explore every possible solution. Each solution is a permutation of the N points. Then there are N! permutations possible. And then the time complexity is factorial $O(n!)$.

For the space, we need to store the current and the partial solution which has a complexity $O(n)$ but we also need to store all the distances between every pair of points. As a result, the space complexity of the algorithm is $O(n^2)$. However, if we take into account our strategy to store the results of minimum spanning trees, the space complexity becomes exponential.

4.1.5 Problem Expansion for Branch and Bound. The first question is : How do we expand a subproblem ? If we are currently have a partial solution $[P_1, ..., P_i]$ (of size inferior to N), we need to add a new city to the list.

We select firstly the cities that are closest to the last point (P_i here) and which are not yet in the list.

That is why we calculated a matrix that represents for each point its closest neighbors.

In fact, this is a greedy choice that is aimed to save us computation time as we explore firstly the more relevant potential solutions.

4.1.6 Tactics to reduce the computational time. We used the Minimum Spanning Tree because it seemed to be the best option to have the highest lower bound (we use the Kruskal's algorithm). Also, without any strategies, we can compute the same Minimum Spanning Tree multiple times. For instance, for both the partial solution [1,2,3,4] and [2,4,3,1], we are going to compute the same Minimum Spaninng Tree with the other points. That's why we store the result for each combination of points so that we can save computation time.

4.1.7 Pseudocode. See Algorithm 1.

Algorithm 1 Branch and Bound

```
CalculateAllDistances()
PrepareKruskal()
Minimum=99999999999
RESEARCH(0)
function RESEARCH(Solution)
    if length(Solution)=N then
        Sum=CalculateSum(Solution)
        if $Sum < Minimum$ then
            Minimum=Sum
        end if
    end if
    for $Point \in RankedNeighbors(Solution[-1])$ do
        if $Point \notin Solution$ then
            Solution.add(Point)
            LowerBound=MST(Solution)+Value(Solution)
            if $LowerBound < Minimum$ then
                RESEARCH(Solution)
            end if
            Solution.pop()
        end if
    end for
end function
```

CalculateAllDistances()	Calculates all distances between each pair of points.
PrepareKruskal()	Prepares the edges for the Kruskal's algorithm.
CalculateSum()	Calculates the length of the path of the current solution.
MST()	Calculates the length of the minimum spanning tree of the graph of all cities not in the current partial

	solution with the first and the last cities of the partial solution.
Value()	Calculates the length of the path of the current partial solution.
RankedNeighbors()	Takes a city as an argument and return all the cities ranked according to their distances to this city.

4.2 Approximation Heuristic - Nearest Neighbor

4.2.1 Algorithm. The construction heuristic we have chosen to implement is nearest neighbor. The algorithm is simple: Start at any location c, it to the path, find the nearest location c_1 to c, and repeat the process with c_1 as the new starting point until there are no locations left to add. The pseudocode is as follows:

Algorithm 2 Nearest Neighbor Approximation Heuristic

```
start ⟵ c
visited ⟵ {}
path ⟵ []
while |visited| < |locations| do
    visited.add(start)
    path.append(start)
    start ⟵ find_nearest(start)
end while
return path
```

Where the find_nearest routine computes the distance, whichever metric is preferred, between start and all other locations and returns the location with the least distance from start. This requires looping through all locations. Since we must call find_nearest once for each location, we will loop through all locations once for each location. Thus, it is clear that this algorithm runs in $O(n^2)$ time, where n is the number of locations. What remains is the problem of choosing c. The algorithm uses a set to hold the visited locations, and a list to hold the total list of locations.

4.2.2 Choosing c. The choice of c can impact performance. When experimenting with different starting points, we found several improvements over the location that was listed first in the list of locations. We performed this evaluation by running the above algorithm with each city as a starting point. This, of course, runs an $O(n^2)$ algorithm n times, which would make trying each city as a starting point $O(n^3)$. We choose to use this version of the algorithm.

One could also choose a random starting point, which makes no guarantees of finding the most optimal solution (in the context of a nearest-neighbor approximation), but maintains an $O(n^2)$ runtime. Space complexity is simple, and and is irrespective of node selection: $O(n)$. We need a set storing n values and a list storing n values.

4.2.3 Approximation Ratio. The nearest neighbor algorithm does not have a constant approximation ratio [9]. Instead, the approximation ratio depends on the size of the number of nodes in the input graph. We will reproduce the proof from [9] here:

Let NEARNEIBER be the length of the solution obtained by the nearest neighbor algorithm. Then,

THEOREM 4.1. $\frac{NEARNEIBER}{OPTIMAL} \leq \frac{1}{2}\lceil log(n) \rceil + \frac{1}{2}$

First, we must prove the following lemma. Suppose that for an n node graph (N, d) there is a mapping assigning each node p a number l_p such that the following two conditions hold:

- $d(p, q) \geq \min(l_p, l_q)$ for all nodes p and q
- $l_p \leq \frac{1}{2}$OPTIMAL for all nodes p

Then, $\sum l_p \leq \frac{1}{2}(\lceil log(n) \rceil + 1)$ OPTIMAL.

Proof: We can assume without loss of generality that the node set N is $\{i|1 \leq i \leq n\}$ and that $l_i \geq l_j$ whenever $i \leq j$. We will also assume that

$$\text{OPTIMAL} \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{1}$$

for all k satisfying $1 \leq k \leq n$.

In order to prove the above inequality, let H be the complete subgraph of the set of nodes:

$$\{i|1 \leq i \leq \min(2k, n)\} \tag{2}$$

Let T be the tour in H which visits the nodes of H in the same order as these nodes are visited in an optimal tour of the original graph. Let LENGTH be the length of T. Since the original graphs we are dealing with satisfy the triangle inequality and the edges of T sum to LENGTH and the corresponding paths in the original graph sum to OPTIMAL, we can conclude that

$$\text{OPTIMAL} \geq \text{LENGTH} \tag{3}$$

By condition (a) of the lemma, for each (i, j) in T, $d(i, j) \geq min(l_i, l_j)$. Therefore,

$$\text{LENGTH} \geq \sum_{(i,j)\in T} \min(l_i, l_j) = \sum_{i \in H} \alpha_i l_i \tag{4}$$

where α_i is the number of edges (i, j) in T for which $i > j$. In order to obtain a lower bound on the right hand side of (5), we observe that every α_i is at most 2 because i is the endpoint of only two edges in tour T, as is the definition of tour, and that α_i sum to the number of edges in T. Since k is at least half of the number of edhes in T, assuming the k largest l_i have $\alpha_1 = 0$ and the remaining $\min(2k, n) - k$ of the l_i have an $\alpha_i = 2$ we arrive at a certain lower bound. By our previous sasumption, the k largest are $\{l_i|1 \leq i \leq k\}$ so, our lower bound is

$$\sum_{i \in H} \alpha_i l_i \geq 2 \sum_{i=k+1}^{\min(2k,n)} l_i \tag{5}$$

(5), (4), and (3) together establish (2).

Summing inequalities (2) for all values of k equal to a power of two less than s

$$\sum_{j=0}^{\lceil log(n) \rceil - 1} \text{OPTIMAL} \geq \sum_{j=0}^{\lceil log(n) \rceil - 1} 2 \times \sum_{i=2^j+1}^{\min(2^{j+1},n)} l_i \tag{6}$$

which reduces to

$$\lceil log(n) \rceil \times \text{OPTIMAL} \geq 2 \times \sum_{i=2}^{n} l_i \tag{7}$$

And condition (b) of the lemma implies

$$\text{OPTIMAL} \geq 2 \times l_i \tag{8}$$

And (8) and (9) combine to conclude the proof of this lemma.

In order to prove the theorem, for each node p let l_p be the length of the edge leaving node p and going to the node selected as the nearest neighbor to p. We want to show that the l_p satisfy the conditions of the lemma.

If node p was selected before node q, then q was a candidate for the closest unselected node to p but was not selected. Thus, the edge (p, q) is no shorter than the edge selected and hence

$$d(p, q) \geq l_p \tag{9}$$

The converse is also true. Since one of the nodes was selected before the other, (10) and its converse must hold and condition (a) is satisfied.

To prove condition (b) we must prove that for any edge (p, q)

$$d(p, q) \leq \frac{1}{2} \times \text{OPTIMAL} \tag{10}$$

The optimal tour can be considered to consist of two disjoint parts, each of which is a path between nodes p and q. From the triangle inequality, the length of any path between p and q cannot be less than $d(p, q)$, establishing (11). Because the l_p are the lengths of the pairs comprising tour T,

$$\sum l_p = \text{NEARNEIBER} \tag{11}$$

The conclusion of Lemma 1 together with (12) and the fact that the optimal path length must be greater than 0 prove the inequality of 4.1.

4.3 Local Search - Simulated Annealing

As a local search algorithm, Simulated Annealing (SA) not guaranteed to find the optimal solution. However, it is useful in speeding up the convergence rate for other algorithms with by selecting a high temperature at the beginning of the process. As it cools down and the temperature approaches 0, the probability of finding the optimum solution increases greatly [1]. This works as the probability of accepting neighboring solutions decreases as T decreases. When T is large, even solutions with a longer tour length will be accepted. However, when T is small, the selection will be stricter and this is no longer possible. The key design parameters for this algorithm is the frequency that the temperature is updated and its termination criteria. The algorithm occasionally gets stuck at a local optimum and thus a restart is required.

4.3.1 General Procedure. The Annealing simulation algorithm follows a basic procedure

(1) Set the initial temperature and a cooling rate
(2) Create a random initial solution
(3) Loop through steps 1 and 2 until the stop condition is met. The termination condition is usually when the system temperature is cool enough or a solution with a lower tour length is found.
(4) Select the neighbor by making a small change to the current solution.
(5) Decide whether we accept the neighbor solution or not
(6) Decrease the temperature and continue the looping

4.3.2 Design. The maximum iteration number is the

$$\frac{initialTemperature}{coolingDegrees} \tag{12}$$

.

A common acceptance method is always to accept the better solution and accept the worse solution with a probability of $P(accept) \leftarrow exp(\frac{e-e'}{T})$, where T is the current temperature, e is the cost of current solutio and e' is the cost of the candidate solution.

When the temperature decreases, T decreases and the probability of acceptance decrease as well. Therefore the acceptance criteria becomes more stringent.

Occasionally restarting the procedure can help to improve the quality.

4.3.3 Pseudocode. See Algorithm 3.

Algorithm 3 Simulated Annealing

```
                Input:ProblemSize, $iterations_{max}$, $temp_{max}$
                          Output:$S_{best}$
    function SIMULATEDANNEALING(ProblemSize)
        $S_{current} \leftarrow ScreateInitialSolution(ProblemSize)$
        $S_{best} \leftarrow S_{current}$
4:      for ($i = 1$   to   $iterations_{max}$) do
            $S_i \leftarrow createNeighborSolution(S_{current})$
            $temp_{current} \leftarrow calculateTemperature(i, temp_{max})$
            if $cost(S_i) \leq cost(S_{current})$ then
8:              $S_{current} \leftarrow S_i$
                if $cost(S_i) \leq cost(S_{best})$ then
                    $S_{best} \leftarrow S_i$
                end if
12:         elseIf   $Exp(\frac{CostS_{current} - CostS_i}{temp_c urrent}) > Rand()$
                $S_{current} \leftarrow S_i$
            end if
        end for
16: return $S_{best}$
    end function
```

4.3.4 TSP Implementation and design. For this optimization, variations of the hyperparameters were empirically tested against the Atlanta dataset, as seen in Figure 1 and 2. The best performing hyperparameters were then selected as for use on the other datasets.

4.3.5 Complexities. Simulated Annealing goes through $O(n^2)$ to look for the pairs for n points in the graph. Inside of the while loop, it takes $O(n)$ to go through the search space. The rejection for a temperature change cost $O(1)$ while the acceptance of a temperature

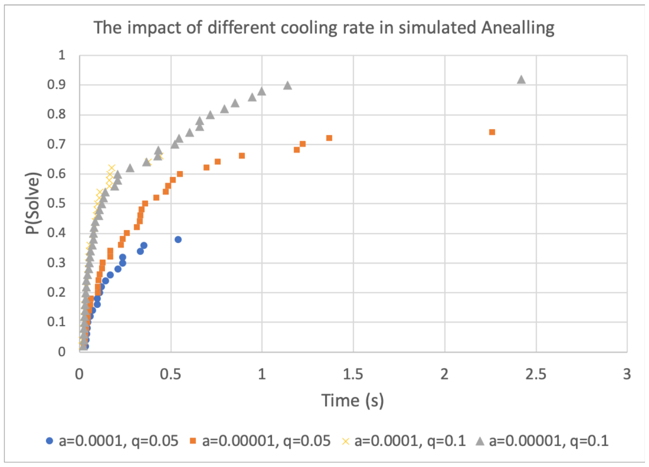

Figure 1: Effect of cooling rate on performance for simulated annealing

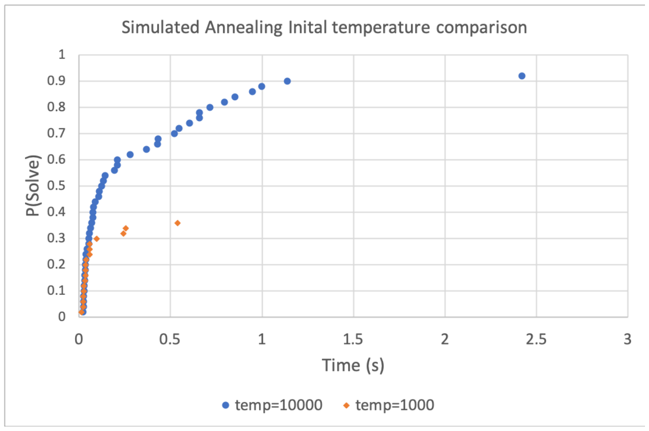

Figure 2: Effect of initial temperature on performance for simulated annealing, q=0.1

Table 1: Hyperparameters Selected

Parameter	Value
Cooling Rate *alpha*	0.00001
Initial Temperature T_{max}	10000

chance is the average path reversal cost of $O(n!)$ for swapping two points. Therefore the total time complexity is $T_n = O(n * n!)$.

The space complexity will be $O(n^2)$ since the data structure will be a graph dictionary. And this step will consume the most of the space [5].

4.4 Local Search - Genetic Algorithm

4.4.1 Introduction. Genetic Algorithms (GA) are a family of algorithms based on the theory of evolution [2]. They derive the neighborhood, as well as the heuristics, for a local search by adhering to core concepts derived from nature. The three core concepts are

(1) Selection of the fittest
(2) Crossover mating
(3) Random mutation

There are many options when implementing a GA, but the core principles remain largely constant.

4.4.2 General Procedure. The fundamental concept of a GA starts from initializing a randomly selected population of DNA. Each DNA represents a solution to the problem and is composed of genes, with each gene representing a specific parameter in the solution. The population of DNA and succeeding generations of it act as the neighborhood in a local search.

Selection:
The selection process follows principles from the first core concept such that the fittest DNA will be selected to pass their gene on to the next generation, in the hopes of preserving parameters that optimizes the fitness. Thus, from the initial population, the fitness of each DNA is evaluated, with the fitness function usually returning a value to be minimized or maximized. Common methods used to select a DNA based on its fitness include always selecting the highest fitness or using a probability distribution with weights proportional to the fitness.

Elitism:
An additional option at this stage is the inclusion of elitism. This sets a certain amount of fit DNA being automatically introduced into the next generation without any modification. This preserves the fittest DNA and guarantees that the next generation will have a lower bound of the fittest DNA. Thus, the algorithm always produces a solution that is better than or equal to the current one.

Mating:
The second core concept of crossover mating works by allowing a child to inherit the fitter parameters from its parents. The method in which parameters are inherited is highly dependent on the requirements of the problem. This is repeated until the next generation of the population has been generated.

Mutation:
Additionally, each gene in a DNA has a chance to undergo mutation. When mutation occurs, a gene is replaced by another randomly generated gene. Similar to its application in the theory of evolution, mutation is used in a GA to introduce variation into the population. Thus, a GA can adapt to the situation and is less likely to be stuck at a local optimum as compared to other local search algorithms.

Termination:
The steps from selection to mutation can be repeated for a fixed number of generations or until a certain cutoff time is reached. Even if multiple of the same solution is found in successive generations, there is no guarantee that optimal solution is found as the algorithm is reliant on random selection. The fittest DNA is then considered the optimal solution found by the algorithm.

4.4.3 TSP Implementation. For this implementation of TSP, we define the following terms can be defined as such

Gene	A set of x and y coordinates of a node
DNA	A route that passes through each node only once and arrives back at the origin. This is a solution to TSP.
Fitness	The reciprocal of the tour length of a route. The shorter the router, the fitter the DNA.

The following implementations have been chosen as for this algorithm.

Selection	Each DNA is selected randomly, with the probability of selection being proportional to the relative fitness of the DNA among the population. The number of parents is fixed at 2 in this scenario to easily preserve the conditions a solution is required to have by TSP.
Elitism	A percentage of the population with the fittest DNA are directly carried over to the next generation..
Crossover	A random segment, of half the length, of one parent is selected and forms the first half of the child. The remaining of half is comprised of the remaining genes in order of the other parent. This is done to ensure that no node is repeated in a DNA [6].
Mutation	If a gene undergoes mutation, another gene in the DNA is randomly selected and their places are swapped. This is done instead of simply picking a new gene to ensure that nodes are not repeated.
Termination	Upon termination, the fittest DNA will be the route with the shortest tour length found and is the most optimal solution produced by the algorithm.

4.4.4 Pseudocode. See Algorithm 4

4.4.5 Complexities. The genetic algorithm has the following complexities

Time Complexity
The time complexity for each generation can be calculated as follows

Initialization	$O(NPopulationSize)$
Selection	$O(NPopulationSize)$ $+O(PopulationSize \log PopulationSize)$
Mating	$O(NPopulationSize)$
Mutation	$O(NPopulationSize)$
Total Time Complexity	$O(NPopulationSize\ NumberOfGenerations)$

Thus, for a fixed set of hyperparameters,

Total Time Complexity	O(N)

This may seems fast, however, the constants, Population Size and Number of Generations, are usually very large and can easily dwarf the actual input size N.

Space Complexity

Space for Population	$O(NPopulationSize)$
Space for Next Generation	$O(NPopulationSize)$
Total Space Complexity	$O(NPopulationSize)$

Algorithm 4 Genetic Algorithm

```
function EVOLVE
    $population \leftarrow \emptyset$
    for i in POPULATION do
        $DNA \leftarrow generateDNA()$
        $population \bigcup DNA$
    end for
    for gen in POPULATION do
        $nextGeneration \leftarrow \emptyset$
        for DNA in population do
            $evaluateFitness(DNA)$
        end for
        $sort(population)$
        $parent_1, parent_2 \leftarrow select(population)$
        for i in POPULATION do
            $child = mate(parent_1, parent_2)$
            $mutate(child)$
            $nextGeneration \bigcup child$
        end for
        $population \leftarrow nextGeneration$
    end for
    return $population[0]$
end function
```

4.4.6 Design. A large number of methods and hyperparameters were taken into account when designing the algorithm. An attempt to validate the design using empirical methods can be seen in Figure 1, 2, and 3.

Hyperparameters
The following are hyperparameters that affect the optimal solution found.

Population Size	The amount of DNA in a pool available for mating.

The larger the population, the higher the variation and the better the solution, but the longer it takes for an iteration of the algorithm. When the number of unique DNA in a population is equivalent to the possible number of solutions, GA is guaranteed to provide and optimum solution.

Elitism Factor	The proportion of the population's fittest DNA being directly carried to the next generation without any modification.

The higher the factor, the better fitter the population will when mating to produce the next generation. However, this reduces the variation and thus increases the probability of the solution only being a local optimum. When this factor is at 0, there is no guaranteed lower bound for the solutions and the optimal solution may decrease at each iteration.

Mutation Probability	The probability a gene might be mutated.

The higher the probability, the higher the variation in the population. However, too high a probability prevents the child DNA from inheriting the fitter aspects from its parents. When the probability of mutation is 1, each child is completely random and does not inherit any parameters from its parents. This will perform worse than a brute force solution as the full solution space is explored at random with no guarantees of finding the actual optimum.

Number of Generations	The number of iterations the algorithm goes through.

With elitism, the higher the number of generations, the better the solution as the fittest DNA of each generation acts as the lower bound of the next. However, this increases the running time.

Selection of Hyperparameters
The hyperparameters are selected empirically. This is done by running the algorithm with different permutations of hyperparameters on a test dataset and selecting the parameters that consistently provide a low tour length. This has the drawback of selecting hyperparameters that might only be optimized for a certain dataset. Further extension to this can be choosing hyperparameters that are scaled to other heuristics such as the size of the dataset.

For this problem, the dataset from Atlanta was used as a test dataset as, due to its input size, it can reach a high level of quality in a reasonable amount of time. Datasets with smaller input size reach its optimum solution too quickly regardless of hyperparameters chosen and datasets with larger input size takes a much longer time to run and test. Each permutation was tested 3 times and the average time and tour length was used as the result.

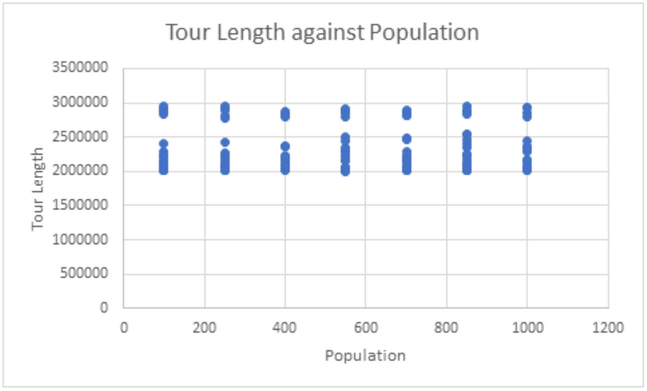

Figure 3: Genetic Algorithm: Graph of Tour Length against Population Size for varying hyperparameters

As can be seen from Figure 3, the effect of the population size on the variance of the solution length is negligible. A population of 250 was thus selected as it has the lowest mean. From Figure 4, an elitism factor of 0.2 was selected as it has the lowest variance. A mutation of 0.05 was selected instead of 0 even though the variance is slightly higher as to preserve the random nature of creating new neighborhoods for a local search. The number of generations was fixed at 10000 as the cutoff time serves as the termination condition.

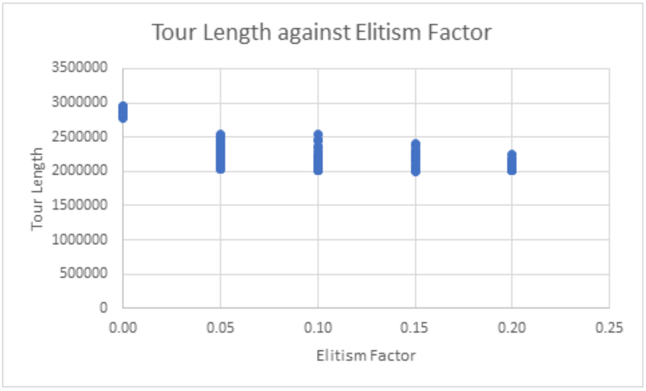

Figure 4: Genetic Algorithm: Graph of Tour Length against Elitism Factor for varying hyperparameters

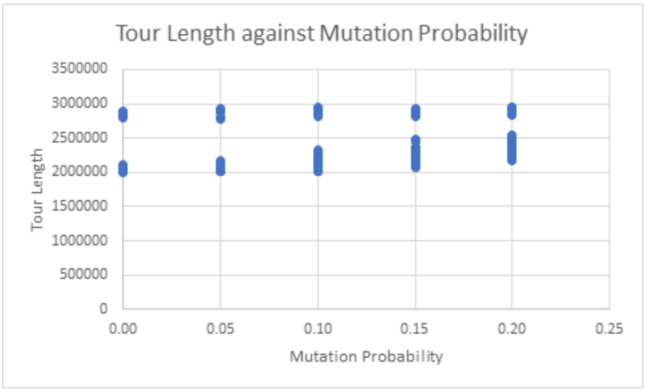

Figure 5: Genetic Algorithm: Graph of Tour Length against Mutation Probability for varying hyperparameters

Table 2: Hyperparameters Selected for Genetic Algorithm

Parameter	Value
Population Size	250
Elitism Factor	0.2
Mutation Probability	0.05
Number of Generations	10000

4.4.7 Theoretical Evaluation. A GA is usually known for being able to quickly reach the vicinity of the optimum solution but takes a much longer time reaching the actual optimum. This is due to its probabilistic nature rather than a fixed heuristic. Most implantation of GA is used to reduce the solution space, before another algorithm is used to find the actual optimal solution. This is akin to actual evolution, where specific evolutionary traits are determined randomly but only those that suit the environment survive.

Table 3: Results for Branch and Bound

Dataset	Time	Solution Quality	Relative Error
Atlanta	30.11	2003763	0.0000
Berlin	1.57	8091	0.0728
Boston	391.06	955440	0.0693
Champaign	475.40	53493	0.0161
Cincinnati	0.03	277952	0.0000
Denver	19.50	109235	0.0877
NYC	49.89	1774194	0.1409
Philadelphia	245.12	1437553	0.0298
Roanoke	42.06	780178	0.1903
SanFranciso	2.60	896718	0.1068
Toronto	239.04	1220114	0.0374
UKansasState	0.01	62962	0.0000
UMissouri	296.47	156572	0.1798

Table 4: Results for Nearest Neighbor

Dataset	Time	Solution Quality	Relative Error
Atlanta	0.21	2039906	0.0180
Berlin	0.82	8181	0.0847
Boston	0.54	1029012	0.1516
Champaign	0.96	61828	0.1745
Cincinnati	0.11	301260	0.0839
Denver	1.52	117617	0.1711
NYC	1.06	1796650	0.1554
Philadelphia	0.40	1611714	0.1545
Roanoke	11.66	773359	0.1799
SanFranciso	2.02	857727	0.0587
Toronto	2.36	1243370	0.0572
UKansasState	0.16	69987	0.1116
UMissouri	2.31	155307	0.1703

5 EMPIRICAL EVALUATION

4 different algorithms have been tested. There are 2 dimensions that we take into account for our study : the quality of the solution (the length of the path) and the running time of the algorithm. A comprehensive table below sum up all the results for all 4 algorithms.

5.1 Platform

All these algorithms have been tested on a single computer with the following configuration

- Windows 10
- Inter Core i5-6500 3.2GHz
- 8GB of RAM
- Python 3.6
  - numpy 1.17.4

5.2 Branch and Bound

Before using the Minimum Spanning Tree, we tried to calculate the lower bound with the matrix reduction method. However, this method takes much more time. For instance, with this method we had to wait 10 minutes to have the exact solution for Atlanta.tsp whereas we only need to wait 30 seconds with the Minimum Spanning Tree. Also we can note that we can quickly find the actual results for small datasets (Atlanta, Cincinnati, UKansasState) but it fails for bigger datasets. However, we always have a relative error inferior to 20%. Also, the fact that we store in memory the minimum spanning trees already calculated gives a great boost on our computation time. For instance, for Atlanta, I get the optimal solution in 750 seconds without storing the MSTs whereas I can get the solution in 30 seconds with memory. In the same way, for Cincinnati we need to wait 0.27 second without memory and 0.03 with memory.

5.3 Approximation Heuristic - Nearest Neighbor

Most of the scenarios also ran quickly and had low relative error. This is thanks to the fact that we run nearest neighbor from every possible starting point instead of from an arbitrary one; we are tightening the bounds on the approximation ratio. Given that the solutions are already of a high quality, this would serve as a good starting point for a more precise algorithm. Note the run times of Atlanta and Roanoke in Table 4. Atlanta has 20 nodes, while Roanoke has 230, a factor of 10 difference, meaning that the running time should be at most 100 times worse due to the $O(n^3)$ runtime of our algorithm We see that Roanoke runs in 11.66 seconds, which falls within that bound. Using all possible cities as a starting point offers about an 8% decrease in tour distance over using an arbitrary node. This raises an interesting question for those looking to use nearest neighbor as a construction heuristic: is 8% improvement worth multiplying the worst-case execution time by a factor of n? For this, we offer no answer other than that it depends on the application and what size of data one will be dealing with. With larger magnitude graphs (e.g. many thousands of nodes), picking an arbitrary node will likely be superior. One other technique to consider would be randomly sampling some of the nodes to use as starting point which hopes to find a middle ground between trying all nodes and trying one arbitrary node, but we did not examine that in the course of this project. We also note that all of our solutions fall within the bounded approximation ratio proved earlier. Also note that the approximation ratio scales with n and our empirical results reflect that, with the largest set of cities, Roanoke, having the largest relative error.

5.4 Local Search - Simulated Annealing

All values in Table 5 were averaged over 10 runs with a cutoff time of 30 seconds while Atlanta and NYC were evaluated to produce the following plots.

5.4.1 Qualified Runtime. See Figure 6 and Figure 7

5.4.2 Solution Quality Distributions. See Figure 8 and Figure 9

5.4.3 Box Plots. See Figure 10 and Figure 11

Table 5: Results for Simulated Annealing

Dataset	Time	Solution Quality	Relative Error
Atlanta	0.55	2217789	0.1068
Berlin	1.60	10261	0.3605
Boston	0.97	1090987	0.2210
Champaign	1.43	70370	0.3367
Cincinnati	0.33	278893	0.0034
Denver	1.99	167884	0.6716
NYC	1.56	2285021	0.4694
Philadelphia	0.74	1762568	0.2626
Roanoke	5.22	2363224	2.6055
SanFranciso	2.26	1684046	1.0786
Toronto	2.43	2975845	1.5302
UKansasState	0.35	62962	0.0000
UMissouri	2.58	235982	0.7782

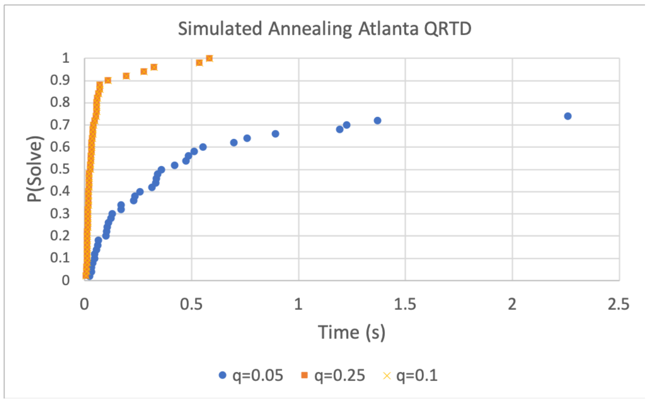

Figure 6: Simulated Annealing: Qualified Runtimes for Atlanta, q = 0.05, q = 0.25, and q = 0.1

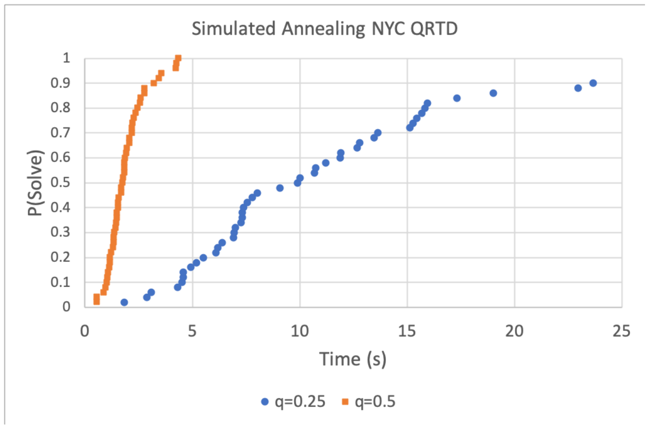

Figure 7: Simulated Annealing: Qualified Runtimes for NYC, q = 0.25 and q = 0.5

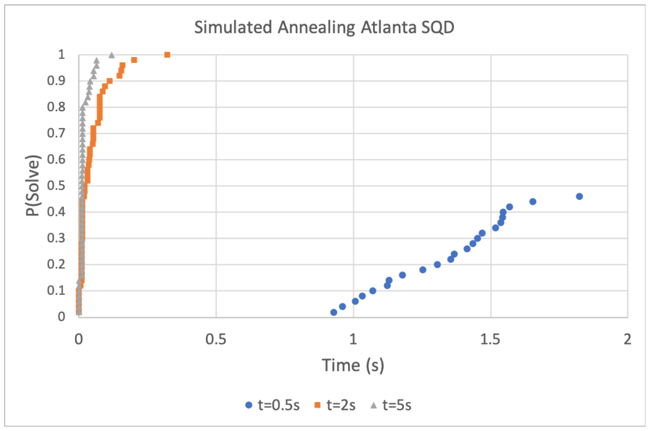

Figure 8: Simulated Annealing: Solution Quality Distributions for Atlanta, t = 0.5s, t = 2s and t = 90s

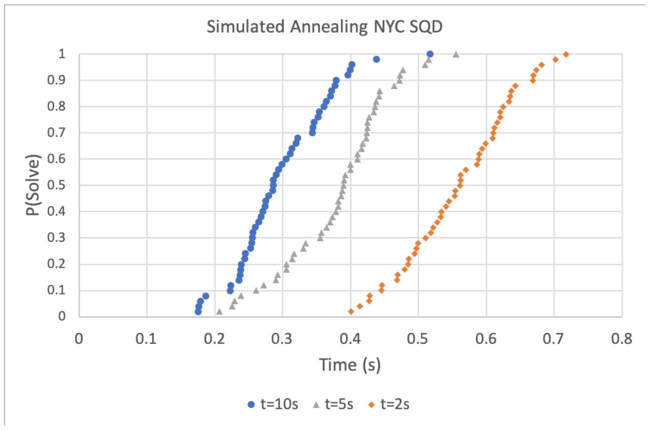

Figure 9: Simulated Annealing: Solution Quality Distributions for NYC, t = 2s, t = 5s and t = 10s

5.5 Local Search - Genetic Algorithm

All values in Table 6 were averaged over 10 runs with a cutoff time of 600 seconds while Atlanta and NYC were evaluated to produce the following plots.

5.5.1 *Qualified Runtime.* See Figure 12 and Figure 13

5.5.2 *Solution Quality Distributions.* See Figure 14 and Figure 15

5.5.3 *Box Plots.* See Figure 16 and Figure 17

6 DISCUSSION

As we could guess, the approximation algorithm performed remarkably well [4] and has the best running time among all algorithms. This is because we only construct and test n solutions. It could run even faster if we picked an arbitrary starting point, but the size of the inputs that we are evaluating our algorithms with do not warrant such drastic measures. Given, say, a dataset of all cities in the United States, for example, we would see the algorithm take

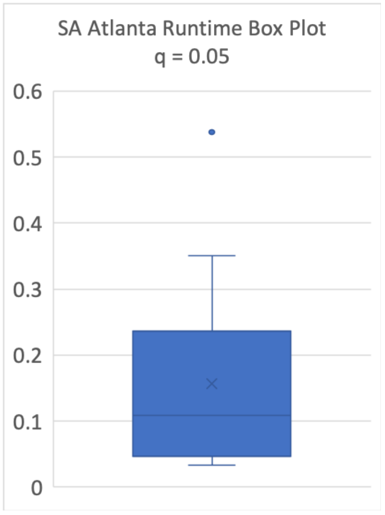

Figure 10: Simulated Annealing: Box Plot of Runtimes for Altanta, q = 0.05

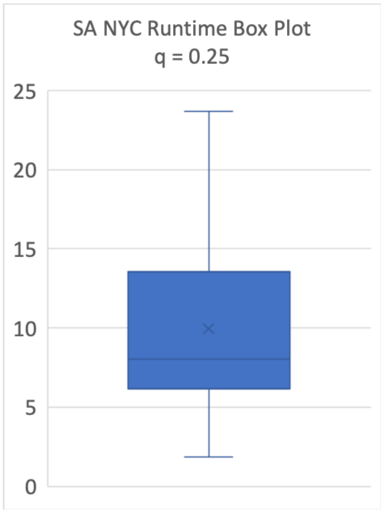

Figure 11: Simmulated Annealing: Box Plot of Runtimes for NYC, q = 0.25

significantly more time and have higher relative error. This construction heuristic metric will almost always make a "good enough" starting point for a more complex optimization algorithm, though may be inferior to other, more complex construction heuristics.

Comparing to the solutions produced by the Branch and Bound algorithm, we note that the performance is fast for small datasets but really slow for large datasets. We also did the test with Philadelphia, a dataset of intermediate size, the algorithm ran during one day without finding the solution. This is because the time complexity of this algorithm is O(n!) and then the computational time skyrockets when the dataset gets bigger. For the quality of the solution, we can see that Branch and Bound gives an exact solution for small datasets which was foreseeable because Branch and Bound always gives an optimal solution if we wait long enough. However, this algorithm takes too much time to compute with bigger datasets and then can't give proper solutions for these datasets.

Table 6: Results for Genetic Algorithm

Dataset	Time	Solution Quality	Relative Error
Atlanta	44.33	2018437	0.0073
Berlin	301.97	9495	0.2589
Boston	295.80	901325	0.0087
Champaign	300.08	73224	0.3910
Cincinnati	0.70	277953	0.0000
Denver	311.33	244313	1.4326
NYC	305.71	2660163	0.7106
Philadelphia	116.63	1395981	0.0000
Roanoke	281.03	5009371	6.6426
SanFranciso	294.56	3188118	2.9350
Toronto	275.84	5253972	3.4671
UKansasState	0.92	62962	0.0000
UMissouri	308.90	434068	2.2708

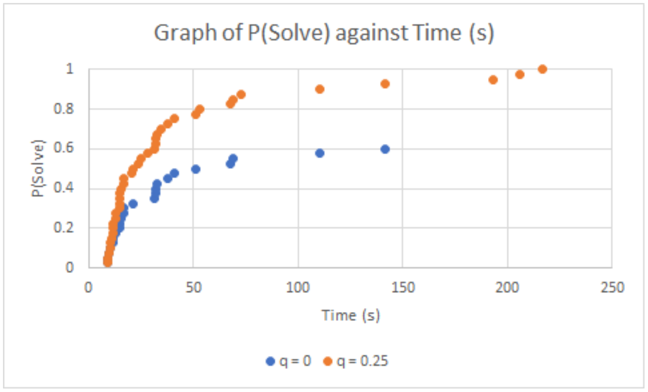

Figure 12: Genetic Algorithm: Qualified Runtimes for Atlanta, q = 0 and q = 0.25

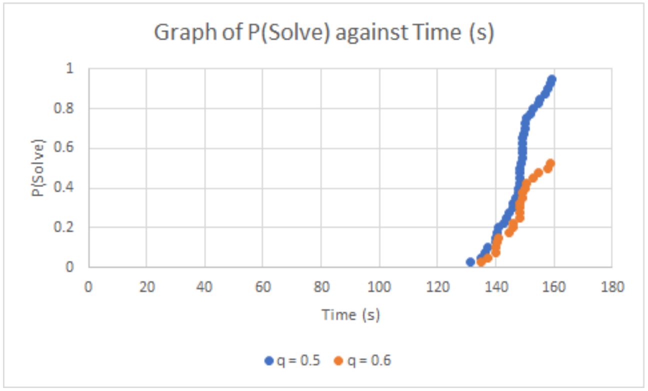

Figure 13: Genetic Algorithm: Qualified Runtimes for NYC, q = 0.5 and q = 0.6

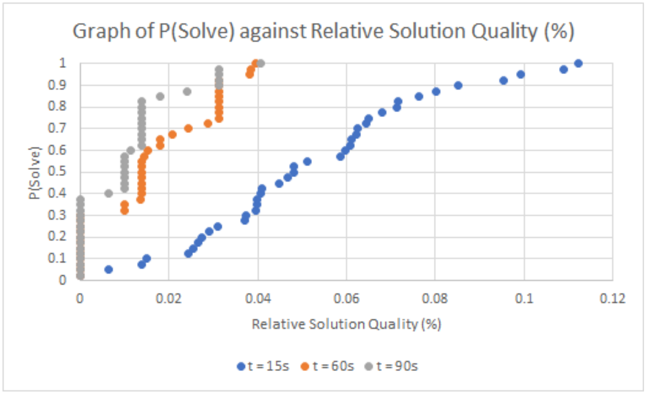

Figure 14: Genetic Algorithm: Solution Quality Distributions for Atlanta, t = 15s, t = 60s and t = 90s

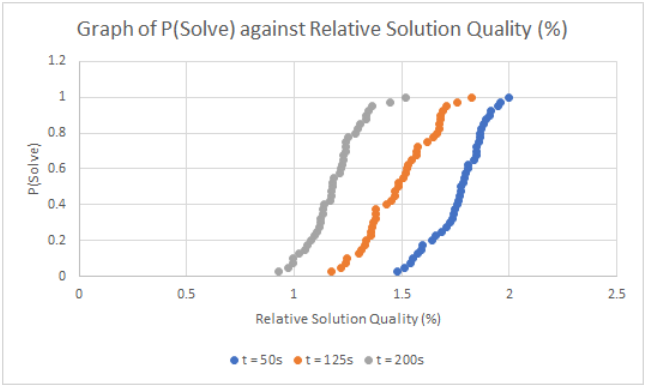

Figure 15: Genetic Algorithm: Solution Quality Distributions for NYC, t = 15s, t = 60s and t = 90s

The strength of Simulated Annealing lies in its ease of implementation. Furthermore, based on the empirical experiments we can see that the cooling rate has an inverse relationship with the performance. When α was reduced from 0.0001 to 0.00001, the number of iterations increased tenfold achieving a better solution at convergence. This however, comes at the cost of running time. For the initial temperature, we compared between 1000 and 10000. The higher initial temperature also performed better. An increase in the input size also adversely affects the level of performance. Therefore, for a dataset with a large neighborhood, simulated annealing performs poorly as each iteration only includes one additional node.

The results of the Genetic Algorithms are disappointing but expected from the literature review[3]. The empirical results, however, show a clear trend in its advantages and disadvantages. From the QRTD plots in figure 12 and 13, we see that solutions are quick to reach an acceptable level of optimality, but any marginal increase in quality takes considerably more time. This can be attributed to the presence of a upper bound in the form of the elitism factor that ensures that the upper bound always improves, but the solution is still affected by the random nature of the algorithm. Thus, it might simply get stuck at the upper bound for a long period of time. Additionally, the crossbreeding method used in this implementation also does not guarantee that the best characteristics are being passed down from the parents. Other problems may have less restrictive requirements where the Genetic Algorithm will perform better.

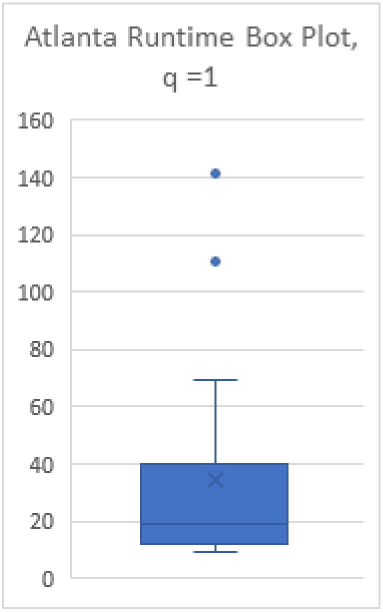

Figure 16: Genetic Algorithm: Box Plot of Runtimes for Altanta, q = 0

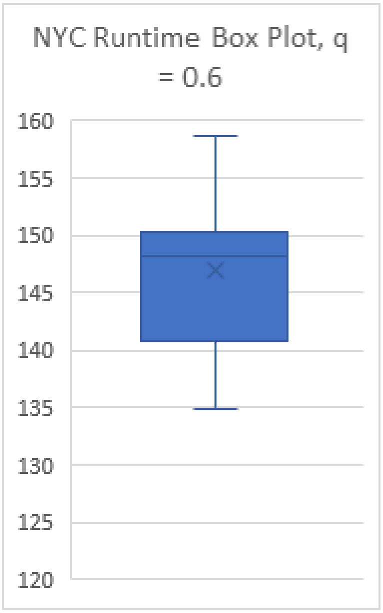

Figure 17: Genetic Algorithm: Box Plot of Runtimes for NYC, q = 6

7 CONCLUSION

Given the NP-Hard nature of the travelling salesman problem, our algorithms often fell short of finding the optimum solution. However, through our rigorous testing, we have identified algorithms that can realistically offer a solution that is sufficient for everyday use. Of the four algorithms tested, Branch and Bound and Simulated Annealing performed excellently when the size of the dataset was relatively small. Unfortunately, both algorithms fell short on larger datasets due to their runtime scaling poorly to an increase in the input size. The genetic algorithm provides a solution near the optimum within a short amount of time, but has great difficulty actually finding the optimum. The marginal returns for the Genetic Algorithm is thus very minimal, rather it can be used to quickly find an upper bound for other algorithms. The approximation Heuristic was the algorithm that had the best trade off between time and quality and provides a good solution in the shortest amount of time. However, such a heuristic is not guaranteed to find the true optimal even when given unlimited running time. If a case occurs where only the true optimal solution will suffice, then only the Branch and Bound algorithm can be used as its search space is exhaustive and is guaranteed to eventually return the optimum solution, even if it takes close to forever.

7.1 Future Studies

For this experiment, all algorithms were tested on a single processor. However, given today's improvement in parallel computer architecture, we are now capable of optimizing algorithms for multiple cores. We feel that further research into this field is warranted and may produce algorithms produce even better results.

REFERENCES

[1] G. A. Croes. 1958. A method for solving traveling salesman problems. *Operations Research* Volume 6, Issue 6 (1958), 791—-908.

[2] L. Davis. 1991. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

[3] D. B. Fogel. 1988. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics* Volume 60, Issue 2 (1988), 139—-144.

[4] G Gutin, A Yeo, and A Zverovich. 2002. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discrete Applied Mathematics* 117, 1-3 (2002), 81–86.

[5] P. B. Hansen. 1992. *Simulated Annealing.* Technical Report 170. Syracuse University, School of Computer and Information Science.

[6] B. Johnson. 2017. Genetic Algorithms: The Travelling Salesman Problem. https://medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1 [Online].

[7] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy. Kan, and D. B. Shmoys. 1987. *The Travelling Salesman Problem. A Guided Tour of Combinatorial Optimization.* Chichester.

[8] Murty K G. Sweeney D. W. Little, J. D. C. and C. Karel. 1963. An algorithm for the traveling salesman problem. *Operations Research* Volume 11, Issue 6 (1963), 863–1025.

[9] Stearns R. E. Rosenkrantz, D. J. and P. M. Lewis II. 1977. An analysis of several heuristics for the travelling salesman problem. *Siam J. Comput.* Volume 6, No. 3 (1977).