CSE 6140 Final Project Report

Connor Guerin
cguerin6@gatech.edu
Georgia Institute of Technology

Apurv Priyam
apriyam3@gatech.edu
Georgia Institute of Technology

Yotam Ghebre
@gatech.edu
Georgia Institute of Technology

Sushanto Praharaj
@gatech.edu
Georgia Institute of Technology

1 INTRODUCTION

The Traveling Salesman Problem (TSP) is a problem arising in operations research and computer science. The TSP is widely studied due to its many applications to practical problems in STEM fields. Despite being a fundamental problem in research since its first formulation in the 1930's, the TSP remains to be solved completely - there currently does not exist a method for producing an exact solution in polynomial time.

Due to the problem's wide applicability, there exists extensive research into constructing tractable methods for finding solutions to the TSP. The aim of this paper is to explore the theory and performance of several of these different methods. Sections 2 and 3 present a formal definition of the TSP and a survey of related work, respectively. Section 4 presents each of the algorithms studied - two local search methods, an exact branch-and-bound method, and a heuristic approximation method. Section 5 details the empirical results for each of the algorithms, and section 6 discusses the discusses the implications of the obtained results.

2 PROBLEM STATEMENT

2.1 Definition

Intuitively, the TSP is the following: given a list of cities and their locations, find the shortest tour which visits all cities exactly once and returns to the starting city.

Formally, this can be stated as, given a graph consisting of edges and vertices, $G = (V, E)$, if each edge $e = (v_i, v_j)$ has corresponding weight w_{ij}, find the cycle that minimizes the sum of weights such that each node is visited exactly once. In graph theory, a cycle that visits each node exactly once is referred to as a Hamiltonian Cycle. Thus, the TSP is equivalently to find the minimum Hamiltonian Cycle. The TSP can be framed in two ways - as a decision problem, or as a combinatorial optimization. As a decision problem the TSP is "does there exist a Hamiltonian cycle with weight smaller than some given weight w?" Formulated as an optimization problem, the TSP is an integer linear programming problem:

$$\min \quad \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} x_{ij} w_{ij}$$

$$\text{s.t.} \quad \sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad \forall j \in \{1, ..., n\}$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad \forall i \in \{1, ..., n\}$$

$$\sum_{i \in S}^{n} \sum_{j \in S}^{n} x_{ij} \leq |S| - 1 \qquad \forall S \subsetneq \{1, ..., n\}, |S| \geq 2$$

$$x_{ij} \in \{0, 1\}$$

$$\text{where } x_{ij} = \begin{cases} 1 & \text{if edge } (v_i, v_j) \text{ is included} \\ 0 & \text{else} \end{cases}$$

The objective function is to minimize the sum of weights for chosen graph edges. Constraints 1 and 2 ensure that each city is visited only once. Constraint 3 ensures that the constructed tour returns to the starting point last.

The algorithms presented in section 4 will be concerned with solving the optimization formulation, i.e. finding the minimum Hamiltonian cycle.

2.2 Classification

Presentation of the optimization form of the TSP lends itself to a mention of the difficulty of TSP. The TSP falls into the NP-complete class of problems. Without providing a formal proof, this can be easily seen due to the relation that the TSP has to Hamiltonian cycles. The TSP is simply a special case of the Hamiltonian cycle problem. It is well known that finding a Hamiltonian cycle in a graph is an NP-complete problem, and thus it can be shown via a reduction that the TSP also falls into this class.

Due to its NP-completeness, there does not exist an algorithm to efficiently and exactly solve the TSP. A naive approach to solve the problem would be with brute force, however, the number of computations expands exponentially as the number of cities increases - even with a few hundred cities, total enumeration could take a computer years to solve. For this reason, and because of the TSP's wide applicability, there have been many different approaches studied for finding near-optimal solutions efficiently.

2.3 Solution Approaches

Although there exist many approaches for solving the TSP, this section will detail the classes of approaches used in this paper.

- Branch-and-Bound: The Branch-and-Bound algorithm is a means of solving the TSP exactly. Despite the worst case complexity still being exponential, this approach can often be more efficient than a brute force approach.

  The Branch-and-Bound algorithm explores all options by constructing sub-problems where each sub-problem looks at a set of all possible tours. For each sub-problem, a function is used to calculate the lower bound on the solution. The goal of the algorithm is to find a tour in one of these sub-problem subsets whose sum of weights is less than or equal to the lower bound of all other tours.

  The obvious advantage of this approach is that it provides a globally optimal solution. However, using this method will not scale well as the number of nodes in the graph increases, and finding the global optima can quickly become infeasible within given time constraints.

- Construction Heuristics: Construction heuristics aim to reduce the computation time at the expense of reducing the solution quality. The idea behind construction heuristics is to build a tour by using some approximation method for determining the next best city to add to the tour. After the tour is constructed, the algorithm does not attempt to improve the solution.

  There are several common construction heuristics. Often these approaches are greedy, but do not have to be. Some examples of construction heuristics include Nearest Neighbors, which adds to the tour whichever city is closest to the current city, Farthest Insertion Heuristics, which inserts a city whose minimum distance to a city on the cycle is maximum, and Closest Insertion, which inserts a city closest to a city in the tour.

  The benefit of these approaches is that only one tour need be constructed. Cities are added to the tour one by one until the tour is complete. This has the advantage of reducing run time, however, since the solution is not improved upon, this increase in speed often comes at the cost of reduced solution quality.

- Local Search: Local search methods in many cases fall in between exact methods and construction heuristics in regards to balancing increased computation and increased solution quality.

  The idea of local search methods is to iteratively improve the solution by searching within a given neighborhood. This is done until the algorithm exceeds its allotted computation time/iterations or there is no improving solution in the current neighborhood.

  Generally speaking, a problem's neighborhood is the set of solutions differing from the current solution in some specified way. For example, with the local search algorithm 2-OPT, the neighborhood is the set of all possible tours that could be constructed after removing two particular edges in the current tour. Many local search algorithms exist, each with their own way of defining a neighborhood.

3 RELATED WORK

3.1 TSP Exact Solution with Branch-and-Bound

A lot of research has been conducted on the Travelling Salesman problem in general since it is a routing problem of generic interests and finds applications not just in delivery and routing traffic, but also in fabricating circuit boards and other avenues where finding optimal circuits and paths becomes of paramount importance.

The method of Branching and Bounding has been investigated for improvement and optimization. Droste[3] investigates the potential of using the Branch and Bound solution to routing problems vis-a-vis other approaches (suggested Ant-colony). The reports of this thesis suggest that Branch and bound has the potential to be used in a heuristic manner, since good tours were found in spite of them not being the global optimum.

A concurrent view of the Branch and Bound algorithm was cited by Wiener[10], who also suggested the application of the Branch and Bound approach to the Travelling Salesman Problem, in a heuristic manner, and suggested that it could be used as an approximate solution or could be used to return an exact value for the best possible solution when an optimal solution cannot be finalized in a reasonable amount of time.

3.2 TSP Heuristics

There are several related works that compare different construction heuristic methods and highlight the potential advantages of each.

Nilsson [9] gives the complexities and brief overviews of methods for implementing nearest neighbor, Christofides, and insertion algorithms. This paper also compares these approaches to many other popular algorithms and mentions data structures for implementation. The potentially most comprehensive and influential survey of construction heuristics is that of Bentley [1]. This paper was the first to present many of these construction heuristic algorithms. It splits construction heuristics into three categories: heuristics that grow fragments, heuristics that grow tours, and tree based heuristics. Class one includes the Nearest Neighbor, double ended nearest neighbor, and multiple fragment algorithms. The second class includes nearest insertion, farthest insertion, and random insertion algorithms. And the last class includes minimum spanning tree, Christofides, and fast recursive partitioning algorithms. This work presents empirical results along with the qualitative comparisons to detail how each of these algorithms compare in

computation time and solution quality for a test set of nodes.

Apart from these popular construction heuristics, new methods are still being created, although not frequently. Some examples of more recent approaches include that of Match twice and stitch in Kahng [7], where tour structure constraints are relaxed to allow formation of multiple cycles and then the cycles are joined together to form a tour. Another example is the recursive-selection with long-edge preference (RSL) method presented by Okano [10]. This method is a modification of multiple fragments that causes the nearest neighbor ratio to increase and subsequently cause the neighborhood for 2-opt algorithm to become larger.

3.3 TSP Local Search - Genetic algorithm

Many research has been done in the field of genetic algorithm due to its flexibility to use in different types of problems. Therefore, there are noteworthy publications that review genetic algorithm and give a comprehensive overview of the field. Getting influenced by the Darwin's theory of evolution, John Holland introduced genetic algorithm in 1960. Afterwards, it was extended by his student in 1989.

One of the earliest significant contributions was made by Fogel [4] in 1988. He points out the importance of crossover operator in the genetic algorithm and emphasizes that without crossover, the algorithm wouldn't be more than a random search. Fogel also warns of the over-ambitious mutation operators that could destory the link between offspring and parent but could lead to more exploration.

Many contributions have also been made in solving TSP through genetic algorithm by introduction of various types of crossover operator. Goldberg and Lingle [5] proposed partially mapped crossover in the year 1985. The order crossover (OX) was proposed by Davis [2] in 1985. Both of these methods have been tried in this paper and detailed information is provided in Algorithm section.

Various attempts have also been made to create hybrid genetic K-means algorithm [8] which aims to divide the large scale TSP problems into smaller sub-problems using clustering and then use genetic algorithm to find the shortest path for each sub-problems. Finally, the sub-groups are combined into one using an effective connection method to give the result for the main problem.

3.4 TSP Local Search - 2Opt

2opt has previously been used to attempt to approximate the TSP solution before. 2opt's primary strategy in the tsp problem is to eliminate all overlapping sub-routes in each iteration, making sure to have no overlaps in the final solution if converged. In the 2opt algorithm, in each iteration you can either make the best possible swap or the first encountered swap opportunity.

Studies have shown that choosing the first encountered opportunity, if start state is always random, will on average perform better than picking the best possible swap[6]. As a result of this survey, we've chosen to swap on every first available opportunity within each iteration to enhance runtime efficiency.

4 ALGORITHMS

This section presents the algorithms employed to solve instances of the Traveling Salesman Problem. The pseudocode, complexities, approximation guarantees, and methods of implementation are discussed for one approximation heuristic, one exact branch and bound algorithm, and two local search methods.

4.1 Branch and Bound Algorithm

The Branch and Bound algorithm aims to create a recursive tree structure in maintaining the number of nodes created and visited as part of the tour, and traverse each such path in an attempt to locate the optimal tour. The Branch and Bound takes this tree structure built, estimates a lower bound and prunes the tree structure in order to preempt unnecessary recursions and render a more efficient algorithm.

We estimate the lower bound as the half of the sum of the edges that are attached to the node under the premise that we will be entering the vertex through one of the edges, and exiting through the other. In the event that there are multiple edges attached to a node, our approach becomes a little more interesting. We then determine the lower bound to be half the sum of the minimum two edges that intersect the vertex in question.

Using the approach mentioned above, we estimate a lower bound on the entire tour for the Travelling Salesman problem by taking the sum of the lower bound for each vertex as described above. We then proceed to explore the vertices recursively and eliminate any branch in the tree structure that has a cost greater than the bound we have set by the method discussed herein. Thus we recursively prune out bad paths and refine our solution.

Branch and Bound:

Pseudocode for Branch and Bound

```
Final_path = [ ]

Function TSP ( Adjacency Matrix A)
    Nodes = [ 1....n]
    Tour = [ ]
    Visited = [ false ]
    Toursum = 0
    Level = 0
    Current_weight = 0
    Recursive TSP( A , bound, toursum, level, tour, current_weight)
End Function

Function RecursiveTSP( A, bound, toursum, level, tour,current_weight)
    If level = n :
      If tour[ last_element ]  != tour[0]
        current_res = current_weight + A[tour[level-1]][tour[0]]
      If current_res < final_res
        Final = tour
        Final_res = current_res
      Else:
        For all vertices v in n:
          If edge from last element of tour exists to v and visited[v] == false:
            Bound += (minimum edge(v)1 + minimum edge(v)2) / 2
          If Bound + weight < final_res :
            Explore this node, add it to tour
            Toursum += weight
            Current_weight +=weight
            RecursiveTSP( A, bound, toursum, level + 1, tour,current_weight)
          Else:
            Explore other branches of parent for more optimal solutions
```

4.1.1 Pseudocode and Implementation. The python implementation of the branch and bound algorithm for the travelling salesman problem, utilises an adjacency matrix to denote the edges computed between each pair of nodes from the dataset given.

It maintains the path visited as a list in python and for each recursive call, adds (appends) the respective node to this list of the current path in the graph.

We also maintain a list of visited nodes in the graph that take only boolean values representing if the node has already been visited.

When we branch, after making an inference, we compute the lower bounds for both children of the parent node. If we find the lower bound of any of the children nodes to be as high or higher than the lowest cost found so far, we prune this branch, and need not consider its descendant nodes.

If neither child can be pruned, we take the first the child with the smaller lower bound and then we must consider again whether the sibling can be pruned, since a new best solution may have been found while we were accomplishing this step.

4.1.2 Complexity Analysis.

(1) Time Complexity: The Branch and Bound algorithm's performance has been found to perform marginally better than the Brute Force approach with it approaching the runtime of that of the brute force solution to the Travelling Salesman problem in the worst case. This entails no circumstance being present where the tree subspace of the nodes can be pruned and thus the runtime complexity of the Branch and Bound algorithm is O(n!) as well.

We find this assumption to hold since the algorithm chooses between n cities every iteration, then n-1, then n-2, and so on, giving us a run time of $O(n!)$.

(2) Space Complexity: Once again we find ourselves with a worst case complexity equivalent to the Brute Force approach in the worst case since at every stage, we store each of the possible nodes in the memory during recursive calls, thus we have an overall space complexity of $O(n!)$ exponential space complexity.

4.1.3 Strengths and Weaknesses.

(1) Strengths: Although the Branch and Bound approach to the Travelling Salesman problem is not without its caveats, for the datasets provided, it with runtimes longer than the restrictive cutoff of six hundred seconds, the algorithm was seen to converge to the exact solution. Thus, given an appropriate amount of time and storage, the algorithm performs reasonably better and more accuracy than certain alternatives.

This approach maintains a state space tree during its computations which may be exploited for other computations' purposes.

Furthermore, it provides solutions that are significantly better by pruning cases that do not allow it to reacha global optimum while saving time and space complexity vis-a-vis the brute force approach.

(2) Weaknesses: While the Branch and Bound algorithm is guaranteed to provide an exact solution given the appropriate amount of resources - space and time, for larger graphs with a number vertices exceeding 100 it may not be practical to allot this much computational power when another algorithm can accomplish the task with a much better runtime and space complexity while detracting minimally from the accuracy of the result.
The Branch and Bound algorithm has been found to be difficult to parallelize.

4.2 Approximation Heuristic: Nearest Neighbors

The nearest neighbor heuristic is a simple greedy approach to approximating TSP solutions. This algorithm favors simplicity at the expense of almost always returning sub-optimal solutions. The algorithm is greedy in nature due to the selection criteria for choosing the next city to visit - select the closest city which has not yet been visited. Due to its simplicity, this approach can often find solutions quickly, but the quality of the solution is very dependent upon the chosen starting city as well as the structure of the graph being

traversed. The speed of the algorithm allows for testing each city as a starting point.

4.2.1 Pseudocode and Implementation. The python implementation of the nearest neighbors algorithm begins with the following initialized data structures.

A dictionary, *nodes*, with form {node number: location coordinates} where node number is an integer and location coordinates is a list.

A dictionary, *visited*, with form {node number: visit} where visit is a Boolean variable indicating whether the node has already been visited. To begin, all nodes in *visited* have value *False*.

Two empty lists *tour* and *current_tour* for storing the best solution and the current solution, respectively

For each iteration the algorithm begins with a city c. It iterates through each other node in the *nodes* dict and calculates the distance between c and all other cities. It tracks and stores the smallest distance and adds the node corresponding to that distance to *current_tour*. The node that has been added to the tour is subsequently marked as visited. The algorithm continues this process but now using the most recently visited city as c. Once all cities have been visited and added to the tour, the initial starting point is added as the final element.

This process is executed using each possible node as c. After each iteration, all values in *visited* are reinitialized to *False*. Further, if the cut-off time has not been reached and if *tour* is better than *current_tour*, *current_tour* becomes *tour*, otherwise continue to the next iteration. This is continued until the cut-off time is reached or an iteration has been run with every city, whichever happens first. The algorithm returns *current_tour*.

The entire process of the algorithm is detailed with the pseudocode below:

Algorithm 1: NearestNeighbors

```
NearestNeighbors:
    nodes = {1 : [x_1, y_1], ..., n : [x_n, y_n]}
    visited = {1 : False, ..., n : False}
    tour = []
    for c_i ∈ nodes do
        visited = {1 : False, ..., n : False}
        current_tour = [c_i]
        best_value = ∞
        while all nodes not visited do
            best_distance = ∞
            nearest = None
            for c_j ∈ nodes\c_i do
                if distance(c_i, c_j) ≤ best_distance then
                    best_distance = distance(c_i, c_j)
                    nearest = c_j
                end
                current_tour = current_tour ∪ nearest
            end
            if ∑_{i=1}^{n-1} distance(c_i, c_{i+1} ≤ best_value) then
                best_value = ∑_{i=1}^{n-1} distance(c_i, c_{i+1})
                tour = current_tour
        end
        return tour
```

4.2.2 Complexity Analysis.

(1) Time Complexity: The process of finding the nearest neighbor to a given node requires iterating through and calculating distances for $n-1$ nodes in order to find the nearest neighbor. Thus, this task requires $O(n^2)$ time.

This task must be executed repeatedly until all nodes have been visited, i.e. it must be performed n times to construct a tour. The process of creating a tour from a specific starting city and adding the starting city as the last element requires $O(n * n + 1) = O(n^2)$ time.

Tours will be constructed starting with the first city as the initial point and continuing until either all cities have been tested as the initial point or time runs out. If all cities are tested, there will be n different tours constructed. This means that the total time complexity in the worst case will be $O(n * n^2) = O(n^3)$

Note that it could be possible to speed up run time by only iterating over nodes that have not been visited when finding the nearest neighbor, however, this would not affect the overall complexity; n^2 distance comparisons would become $n + (n-1) + (n-2) + \ldots + 1$ which still evaluates to $O(n)$ time.

(2) Space Complexity: This implementation requires storing arrays for the current tour and the best tour. Each of these

arrays will be of length n. Total this is $O(2n) = O(2n)$ space.

Additionally, two dictionaries of length n each must be used to store information about the nodes. Since each dictionary has a key and a value for each index, this requires $O(2 * 2n) = O(4n)$ total space.

The total space complexity is then $O(2n) + O(4n) = O(6n) = O(n)$

4.2.3 *Strengths and Weaknesses.*

(1) Strengths: The strengths of this approach to solving the TSP problem are the simplicity of implementation as well as the reasonably fast computation time. As far as simplicity, in practice, the nearest neighbors implementation is very straightforward. Its simplicity also lends itself to having small space complexity.

For computation time, if the problem does not have a large number of nodes it is very feasible to try all starting points despite the complexity being $O(n^3)$. For a few hundred nodes this still takes less than a minute. Although the solution will most likely not be optimal, the nearest neighbor approach can often produce tours of reasonably good quality, and this could be a fair trade off for simplicity and fast computation.

(2) Weaknesses: The main weakness for this approach is that it will very likely return a sub-optimal solution and there is no way to guarantee the quality of this solution. The greedy approach of choosing the nearest neighbor as the next city is much more likely to result in settling on a local optimum as opposed to the global optimum. Further, the solution quality is very much dependent upon the structure of the graph - nearest neighbor may do well in some cases and poorly in other cases.

Another weakness of this implementation is that it will not scale well if there is a large number of points. Depending on the size, it may become infeasible within the time constraints to try each city for the starting point. In the case that only a portion of cities are tried for the initial point, there is limited improvement upon the solution (or no improvement if only one city is tried) and the solution quality would be highly dependent upon which cities where tested as initial points. In many cases, there may be no way of knowing which initial points are better than others, thus it is difficult to gauge how well the algorithm will perform.

4.3 Local Search Algorithm: 2opt

The 2opt local search is a simple optimization approach to approximating solutions to the TSP problem. This algorithm compares all valid combinations of the swapping mechanism. This algorithm, if at any point, finds it faster to travel from fi to fj by taking the reversed route, it will take that route instead. This algorithm works well for TSP because it eliminates overlapping routes. Since this is a local search algorithm, it is indeed possible to get stuck in suboptimal solutions.

4.3.1 *Pseudocode and Implementation.*

The python implementation of the nearest neighbors algorithm begins with the following initialized data structures.

A list, current_route, of, existing_nodes, with each node having form [xCoord, yCoord].

The algorithm then performs a nested for-loop; the outer loop goes over the current_route and the inner loop traverses from current position in current_route to the end. In each iteration, we check if a new route, created by reversing all nodes from the start position to the current position will be shorter than the existing best route. If so, we replace the best route with this current route and we restart the algorithm. This algorithm runs until no change is made after a full run of these checks within the nested for-loop or a timeout occurs.

The entire process of the algorithm is detailed with the pseudocode below:

2opt:

```
repeat until no improvement is made {
    start_again:
    best_distance = calculateTotalDistance(existing_route)
    for (i = 1; i <= number of nodes eligible to be swapped - 1; i++) {
        for (k = i + 1; k <= number of nodes eligible to be swapped; k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
                existing_route = new_route
                best_distance = new_distance
                goto start_again
            }
        }
    }
}
```

4.3.2 *Complexity Analysis.*

(1) Time Complexity: Performing a reversal on this list can be done in many ways, heavily depending on the list data structure type. If implemented efficiently, for example with a doubly linked list, reversing part of the list would be done in constant time. Within each search for a portion of the list that can be optimized (reversed), there is a nested for-loop going over the entire loop which worst case adds a $O(n^2)$ runtime complexity. Finally, the amount of times, worst case, that we'll need to perform a reversal would be 2^n, making the final runtime complexity $O(2^n * n^2)$.

(2) Space Complexity: The space complexity of this algorithm will vary depending on how reversing portions of the list is performed. However, if implemented efficiently, this reversing can be done in place meaning the entire space complexity of this algorithm would be O(1).

4.3.3 *Strengths and Weaknesses.*

(1) Strengths: One strength is that it approximates a solution for a really hard problem (TSP), and does it fairly accurately. Another strength is that its space complexity is very low, if the algorithm is implemented efficiently.

(2) Weaknesses: One obvious weakness, as for many local search algorithms, is that we can get stuck in sub-optimal solutions. In this algorithm's case, this can happen due to the fact that it strictly tries to eliminate all overlaps in our final path, but this doesn't necessarily mean that its the most optimal path.

4.4 Local Search: Genetic Algorithm

A genetic algorithm (GA) is a metaheuristic method for solving optimization problem inspired by the process of natural selection and 'survival of the fittest'. In a genetic algorithm, a group (called population) of candidate solutions (called individuals or chromosomes) to an optimization problem is evolved in each generation. Weak individuals are killed and stronger ones are given more chance to breed. A typical genetic algorithm has a genetic representation of the solution and a fitness function to score the solution. A standard representation of solution is an array of bits. Traditionally, solutions are represented in binary as strings of 0s and 1s, but encoding of solution can change depending on the problem, just like Travelling Salesman Problem.

A travelling salesman problem can be formulated as a genetic algorithm with following definitions:

- Gene: City (represented as their id)
- Individual (chromosome): A string of genes representing one route satisfying the constraints of TSP
- Population: A group of individuals
- Fitness function: A function which tells how good a solution is (the total distance for TSP)
- Parents: Two routes selected for mating based on their fitness
- Mating pool: A collection of parents selected for creating next generation
- Mutation: A way to bring variation in the solution (by swapping two cities in the route)
- Crossover: Substrings of two parents are interchanged to create two new routes.

The algorithm starts with a initially generated population of sequences. These individuals can be generated randomly or a small number of individuals can be generated using some heuristics to speed up the convergence. During each generation (or iteration), three operations are generally performed in sequence with some probabilities - selection, crossover, mutation. While the selection does the exploitation of good chromosomes, mutation and crossover do the exploration. Figure 1, shows a typical flowchart of genetic algorithm.

4.4.1 Initial population generation. While the most simplest method to generate the initial population is to generate the individuals randomly, several better methods have been proposed. Such methods uses algorithms like K-Means clustering to create the initial population. For this problem a simple Nearest Neighbour has been implemented to create the initial few individuals. To maintain the diversity in population only few individuals are being created using nearest neighbour.

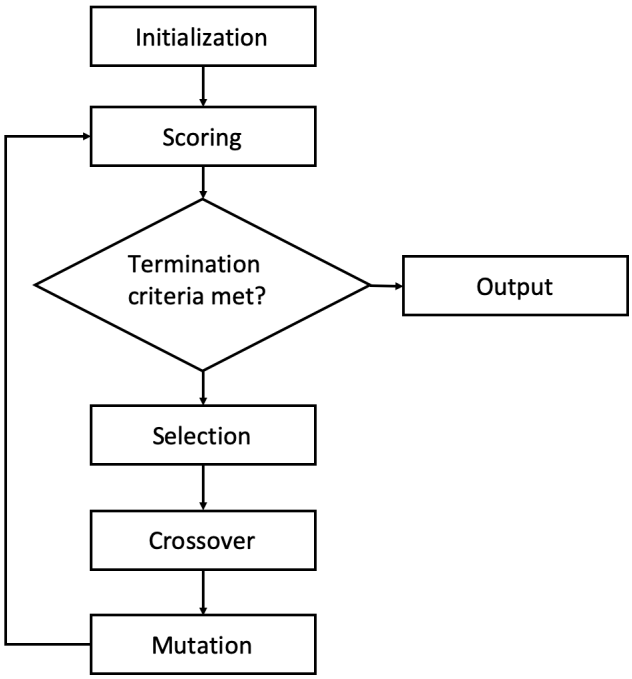


Figure 1: Steps of genetic algorithm

Table 1: Table shows the fitness value and probability of selection for 5 routes

Route	Fitness value	Probability
Route 1	5	5%
Route 2	5	5%
Route 3	10	10%
Route 4	15	15%
Route 5	65	65%

4.4.2 Genetic Operators.

(1) Selection: Selection operation is done on the population to select the fittest sub-population which will generate the offspring. Fitness value can be used as a criteria to judge which individuals are more fit. There are many methods developed on how to select the best parents, but in this project two of the most common methods have been implemented -

(a) Roulette wheel selection: Probability of a individual to get selected is directly proportional to its fitness value. If f_i is the fitness of individual i in a population with N individuals, then its probability of selection will be

$$\frac{f_i}{\sum_{n=1}^{N} f_n} \quad (1)$$

Table 1 shows the probability of 5 hypothetical chromosomes and figure 2 shows the probability with roulette wheel selection method.

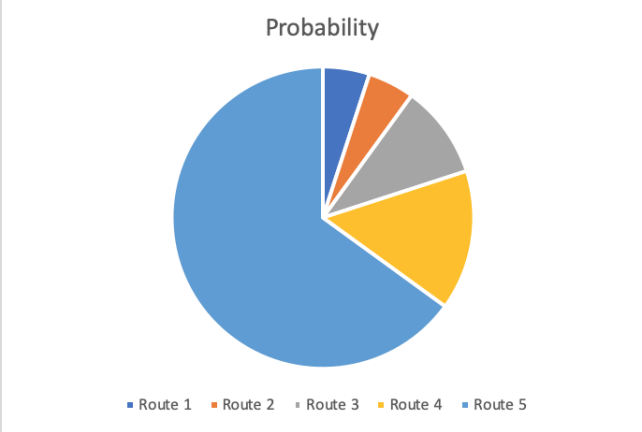


Figure 2: Probability distribution with roulette wheel selection

Table 2: Table shows the updated fitness value and probability of selection for 5 routes

Route	Fitness value	Rank	New fitness	Probability
Route 1	5	5	1	6.7%
Route 2	5	4	2	13.3%
Route 3	10	3	3	20.0%
Route 4	15	2	4	26.7%
Route 5	65	1	5	33.3%

(b) Rank based selection: Roulette's wheel selection criteria will have challenges when the difference in fitness values is very high among the individuals. For example, if the fittest individual has its probability of 90% then the remaining individuals will have very small chance of selection. The rank based approach sorts all the individual by their ranks and assign probability based on the new fitness value, which is their rank. The worst with rank N will have fitness of 1, and strongest individual with rank 1 will have fitness value of N. After this all chromosomes will have some chance to get selected.
Since, the initial random populations generated in this project had distance of almost 10 times the optimal value, this approach was also tried, and found to perform better with faster convergence.

(2) Crossover: The new generation is genetic algorithm is generated from the existing population by a set of operators - crossover and mutation. The crossover operator produces two new individuals from two parents by copying substring from each parents. In a typical crossover, the gene at position i in each offspring is copied from the gene at position i from either of the parents. After the mating pool is selected, two parent chromosomes is selected from the pool and based on a probability, called crossover probability, crossover operation is performed on them. Generally this crossover probability is kept high (0.6-0.8) to ensure that there is enough chance for a new offspring to be created which holds good genes of both the parents. In this project four types of crossover were implemented and tried -

- Single point with PMX crossover
- Single point with OX crossover
- Two points with PMX crossover
- Two points with OX crossover

Each of the four types of crossover is explained below:

(a) Single point crossover: A point on both parents' chromosomes is picked randomly, and designated a 'crossover point'. Genes to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.

(b) Two points crossover: In two-point crossover, two crossover points are picked randomly from the parent chromosomes. The bits in between the two points are swapped between the parent organisms.

Since not all the individual presents a feasible solution in a TSP problem due to presence of constraints, 2 specialized crossover techniques were tried along with single/two points crossover:

(c) partially mapped crossover (PMX) crossover: The crossover points are decided and the genes between these points from each parents are copied to the two new offspring (From parent P1 to offspring O2 and P2 to O1). The remaining genes in O1 are taken from P1 such that no city is repeated. If a city from P1 is already present in O1, the city is taken from the interchanged genes which were transferred to O2 from P1. Similar process is repeated for O2. Figure 4- 7 shows the steps of PMX.

(d) Ordered (OX) crossover: After the crossover points are decided, the genes between these two points are copied to new offspring (from parent P1 to offspring O2 and P2 to O1). Since O1 has the genes from P2, the rest are filled from P1 in the order they appear in P1, as long as the repetitions are avoided. Then same process is followed for O2. Figure 8- 10 shows the steps of OX.

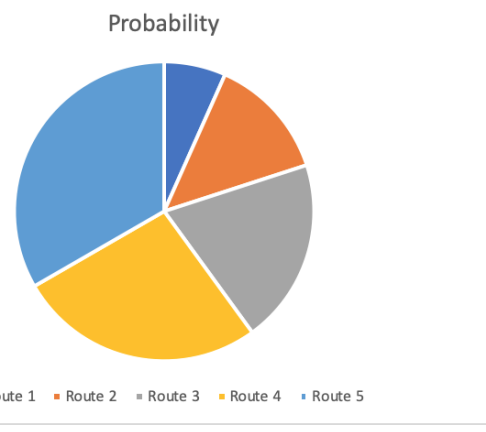


Figure 3: Probability distribution with rank based selection

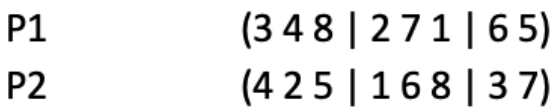


Figure 4: PMX step 1 - Two parents

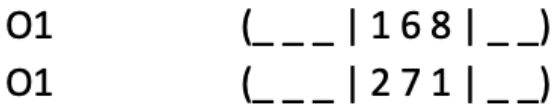


Figure 5: PMX step 2 - Genes from the crossover points are copied to two offsprings (P1 to O2 and P2 to O1

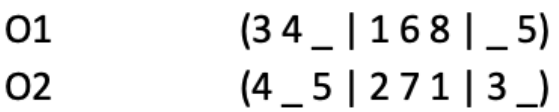


Figure 6: PMX step 3 - Genes are copied from parents such that they are not copied (P1 to O1 and P2 to O2

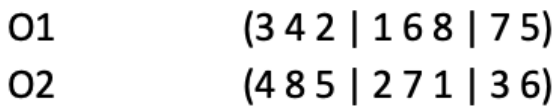


Figure 7: PMX step 4 - Remaining genes are filled (O1 are filled with whatever cities were transferred to O2 from P1

(3) Mutation: Typically with 0s and 1s encoded individual, the mutation operator randomly changes 0 to 1 and vice-versa. But since for TSP, the constraints needs to be satisfied that one city should only be visited one, the mutation operation swaps two genes (city) in the individual (route) with some small probability, called the mutation probability. Figure 11 shows the mutation process.

4.4.3 Fitness function. Since the probability of selected increases as the fitness value increases, the minimization problem needs to be converted in to a maximization problem. The fitness function was taken as the inverse of the total distance of the route. Since no route will have 0 distance, this approach was suitable. One other method was also tried where a diversity factor was included in the fitness score. The weight of diversity in the fitness score was reduced after each iteration like simulated annealing. During the initial generations, With high diversity weight, the algorithm forced the individuals to spread across the solution space. As the generations progressed more weights on distance were put to converge on minimum distance solution. This approach only gave marginal better result but increased both the complexity and run-time, therefore diversity part was removed.

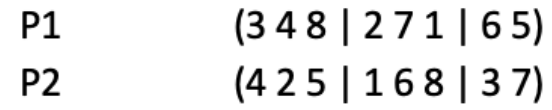


Figure 8: OX step 1 - Two parents

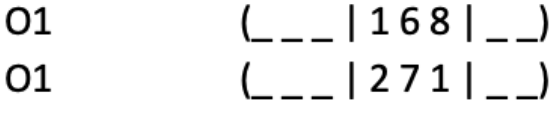


Figure 9: OX step 2 - Genes from the crossover points are copied to two offsprings (P1 to O2 and P2 to O1

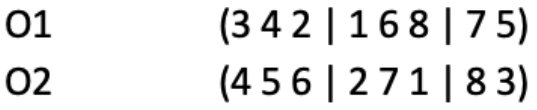


Figure 10: OX step 3 - Genes are filled in order they appear in parent without repetition. If repetition is found next gene is used (P1 to O1 and P2 to O2

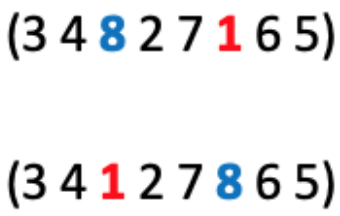


Figure 11: Step showing mutation between reg and green colored genes

4.4.4 Elitism. After crossover and mutation, there is a big chance that we may loose our current best route (individual). Elitism ensures that the top chromosomes from each generation are copied to the next generation's population without any change. This increases the performance of GA, because it prevents losing the best found solution in each generation. But since having high elite individuals in genetic algorithm can reduce the exploration and cause

Table 3: Result comparison after including randomly generated individuals in each iteration

TSP	Error %	# Generations with (without) random sequences
Atlanta	0.6%	109 (126)
UKansasState	0.0%	8 (19)

the algorithm to stuck in a local optimal solution, only a small part of new population is made through elitism.

4.4.5 Next Generation. New population for the next generation were created which had three types of individual:

- Elite chromosomes: Some of chromosomes with highest fitness values were transferred to the next generation
- Newly generated chromosomes were selection based on their fitness value
- Some newly randomly generated chromosomes were added in each iteration to help in exploration. The count of such chromosomes was decided based on the concept of simulated annealing:

$$N_3(i) = round\left(\frac{0.2N}{i^k}\right) \tag{2}$$

Where N is the population size and $N_3(i)$ is the number of chromosomes randomly generated in the generation i and i is the generation number. The value of k was decided on trial and error the final value was selected as 0.2. This improved the result by a small amount. Table 3 shows the comparison for two cities:

4.4.6 Pseudocode and Implementation. The python implementation was done by creating three classes - GeneticAlgorithm and Chromosomes and Node. The GeneticAlgorithm class has following parameters and objects:

- mut_prob: Mutation probability (Numeric)
- cross_prob: Crossover probability (Numeric)
- pop_size: Population size (Integer)
- Populaiton: List of Chromosomes class objects (List of Chromosomes of length pop_size)
- fitness: List of fitness value of all chromosomes (List of length pop_size)
- nodes: List of node id (List of length equal to number of nodes)
- length: Length of chromosomes (Integer)
- distanceMatrix: Dictionary with distance between each nodes
- mutation_type: Only accepts 'swap' (Dictionary of size square of number of nodes)
- crossover_type: Type of crossover (PMX or OX) (String)
- crossover_points: Crossover points (1 or 2) (Integer)
- new_population: New population generated after crossover and mutation. List of Chromosomes class objects (List of Chromosomes of length pop_size)
- new_population_fitness: List of fitness values of new_population (List of length pop_size)
- elite_num: Number of elite chromosomes (elitism) (Integer)
- num_generation: Number of generations to be performed (Integer)
- verbose: Print results at each iteration (0: Don't print, 1: print iteration number and best fitness, 2: print iteration number, best and average score) (Integer)
- seed: Seed value (taken from input) (Integer)
- seed_vals: New sets of seed generated using seed (List of integers)
- smart_pop: Number of initial chromosomes generated using heuristic (Integer)
- cutoff_time: Allowed running time (taken from input) (Numeric)
- tsp_name: TSP instance name (taken from input) (String)
- trace: Dictionary which stores trace of algorithm (Dictionary)
- prev_best: Stores previous best result (Numeric)

The Chromosome class following object and parameters:

- length: Length of sequence (Integer)
- nodes: List of nodes id (List of length equal to number of nodes)
- sequence: Route (List of length equal to number of nodes)
- fitness: Fitness value of the route (Numeric)

The Node class has following objects and parameters:

- id: ID of the node (Integer)
- x: X co-ordinate (Numeric)
- y: Y co-ordinate (Numeric)

Algorithm 2: Genetic Algorithm

GA:
  $nodes = [1, 2, ..., n]$
  $distanceMatrix$
  Step1: Initial Population Generation
  Step2: Calculate fitness of each individual
  for $i \leq$ *number of generation* **do**
    if *Stopping criteria is met* **then**
      STOP
    else
      Step3: Selection (Select mating pool)
      Step4: Select 2 parents from mating pool. Based on crossover probability do crossover. Repeat for all individuals in the pool.
      Step5: Mutate all crossovered individuals based on Mutation Probability.
      Step6: Calculate fitness of newly generated sequence.
      Step7: Create population for next generation.
    end
  end
  return Best sequence with minimum distance

4.4.7 Complexity Analysis.

(1) Time complexity: The process is divided into many parts which runs multiple chunks of code in sequential format. Different parts of algorithm implementation along with thier time complexity:

Table 4: Run time analysis

P	N	G	Time(s)
100	20	100	6.7
100	20	200	12.8
100	20	1000	70.1
100	30	100	9.4
100	170	100	65.2
200	20	100	14.6
1000	20	100	83.5

Notations: P is the population size, N is the sequence length (number of nodes), G is the number of generations.

(a) Step1 - Initial population generation: One initial random chromosome generation require $O(N)$ time. Therefore for the complete population the time complexity is $O(PN)$

(b) Step2 - Calculating fitness of initial population: We have to sum all the consecutive pairs of cities in a city, therefore one chromosome will take $O(N)$ time. Total time complexity $O(P * N)$

(c) Step3 - Selection: Selection is further divided into following parts:
- Store fitness of all chromosome in a list in loop - Time complexity $O(P)$
- Calculate probability of each chromosomes using fitness value (sorting of all the fitness values is required which is most time consuming in this step) - Time complexity $O(PlogP)$
- Create mating pool (To select one individual for the pool, one random number needs to be generated and two search needs to be performed in the sorted fitness scores list) - Time complexity for whole population is $O(P * logP)$

(d) Step4 - Crossover: For each pair of parents, for loops are required which traverse over all the genes in the sequence. The number of loops is constant and doesn't change with problem size - Total time complexity for whole population is $O(P * N)$

(e) Step5 - Mutation: For one chromosomes, maximum N genes can be mutates - Total time complexity for whole population is $O(P * N)$

(f) Step6 - Calculating fitness of modified population: Same as step2 time complexity - Time complexity $O(P * N)$

Since all the above mentioned steps are in sequence, the total time complexity of 1 generation is $O(PN * logP)$ only. For G generations the time complexity will be $O(GPN * logP)$. Table 4 shows time analysis on different values of P, G, and N.

The run times in table 4 shows almost linear increase in time with increase in all the parameters. Although increase in time when P is increased in slightly more than when other are increased.

(2) Space complexity: We can calculate the total space complexity by looking into individual classes space complexity:
- Node class: Holds three numeric values ($O(1)$). Total space complexity - $O(N)$
- Chromosomes class: Holds 3 numeric values and two lists of length N. Space complexity for 1 chromosome- $O(N)$
- GeneticAlgorithm class: Holds 12 numeric, 3 strings, 5 lists of numeric values of length N or P, 2 lists having P Chromosomes class objects ($O(P * N)$) and 1 dictionary to store the trace values ($O(G)$ because this will have maximum of G values). Total space complexity - $O(G + PN)$

Therefore, total space complexity will be $O(G + PN)$

4.4.8 *Strengths and Weaknesses:*

(1) Strengths: Genetic algorithm is suitable when the problem is very complex with very tough mathematics formulations. It covers large solution space simultaneously if formulated correctly. Although genetic algorithm doesn't guarantee global optimal solution, if the problem has many local optimal solutions, genetic algorithm can check most of them simultaneously to find the best among them. It can also support multiple-objective functions.
Since the operators of genetic algorithm are modular, it can solve various types of problems by changing its operators, encoding and parameters individually.

(2) Weakness: One of the main weakness that genetic algorithm faces is that it is sensitive to the initial population. Similarly generated initial populations can easily converge the genetic algorithm to a local optimal solution. Therefore, genetic algorithm can't guarantee optimal solution and it is very common to get sub-optimal solution. It is also difficult to prove the optimality of genetic algorithm solution unless we have a very good lower bound which matches the result of genetic algorithm. Unlike simpler local searches, genetic algorithm has more parameters including different probabilities and operators' types. Which makes it difficult to tune and select the best performing parameters and operators.
Modeling a genetic algorithm also becomes difficult when the problem has constraints. In such cases, either the operators and chromosome's encoding can be changed (like TSP) or some penalty can be added to the objective function, which again makes it difficult.

5 EMPIRICAL EVALUATIONS

5.1 The Branch and Bound Approach

The Branch and Bound algorithm was implemented in python3 without any known library optimizations. The tests were performed on a Personal Laptop with 8 Gigabytes of RAM, and on an Intel i7-5500U Broadwell processor with 2 processors and 4 threads, although, the implemented code appeared to run on a single core during its execution.

Since the Branch and Bound approach is known to perform poorly on graphs with node sizes greater than 100, the algorithm is stopped at the 10 minute checkpoint and the best result found at this point of time is returned with its length and overall path.

The results for the dataset of cities provided with their coordinates is described below. The approach we have followed in order

to implement the branch and bound algorithm involves extrapolating the given data in order to obtain an adjacency matrix for each graph. Since there are a total of n points, this process adds an additional complexity of $O(n^2)$ in order to generate the required adjacency matrix.

Table 5: The following results were recorded using the branch and bound algorithm on the cities dataset. A hard cutoff time of 600 seconds was applied in each case and the best solution found within this time was returned

Dataset	Time(s)	Sol.Qual.	RelError	Cutoff (s)
Atlanta	600	2087242	0.0416	600
Berlin	600	19032	1.5234	600
Boston	600	2203741	1.4663	600
Champaign	600	209943	2.9880	600
Cincinnati	0.77	277952	0.0	600
Denver	600	539776	4.3745	600
NYC	600	7110273	3.5723	600
Philadelphia	600	3645073	1.617	600
Roanoke	600	6767159	9.3243	600
SanFrancisco	600	5575076	5.881	600
Toronto	600	8895801	6.5634	600
UKansasState	0.17	62962	0.0	600
UMissouri	600	655995	3.9431	600

We find that the optimal solution is reached in the cases where the graph is small and that the relative error increases significantly with the number of nodes in the graph.
In fact the optimal solution is reached, i.e the algorithm converges to the given solution only in the cases of the cities Cincinnati and UKansasState since these are graphs with 10 nodes. The next best solution is found in Atlanta (Rel. Err 0.0416), a graph with 20 nodes, and we can see that the relative error is large for graphs with a large number of nodes.

The graph with the largest relative error was Roanoake and this had the largest relative error since it was a graph consisting of 230 nodes. The cutoff time was found to return a suboptimal result on all but two cities within the given dataset.

5.2 Genetic Algorithm

The genetic algorithm was implemented in Python3. The experiments were performed on a personal laptop with 8 GB of RAM and 2.7 GHz Intel Dual Core i5 processor. Since the running time varied even for the same dataset, 10 runs were made for each dataset with different random seed and average error and time were recorded. Comprehensive table 8 shows the average time, solution quality, relative error and selected cutoff-time for genetic algorithm performed on difference dataset. Cutoff-time was selected based on the size of the dataset, cutoff was set at 15 seconds for dataset with less than or equal to 20 nodes. The mutation and crossover probabilities were kept at 0.1 and 0.8 respectively with population size of 100. Maximum generations allowed were 1000, but for majority datasets iterations stopped because of cutoff-time. The probabilities were selected which gave overall minimum error over multiple runs for different datasets.

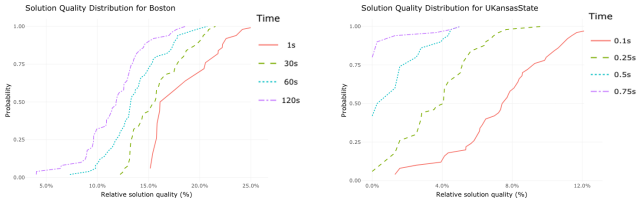
Table 6: Results of GA iteration on different datasets. Cutoff-time was selected based on dataset size.

Dataset	Time(s)	Sol.Qual.	RelError	Cutoff (s)
Atlanta	14.67	2031666	0.014	15
Berlin	30.00	8541	0.132	30
Boston	30.00	1020882	0.143	30
Champaign	30.00	61978	0.177	30
Cincinnati	3.66	278461	0.002	15
Denver	30.00	118401	0.179	30
NYC	30.00	1850087	0.190	20
Philadelphia	15	1567328	0.123	30
Roanoke	30.00	786297	0.200	30
SanFrancisco	30.00	927482	0.145	30
Toronto	30.00	1263996	0.075	30
UKansasState	0.57	62962	0.000	15
UMissouri	30.00	157746	0.189	30

Since analyzing accuracy with respect to time for local search is difficult and depends on initialization, Qualified RunTime Distribution (QRTD), Solution Quality Distribution (SQD) and Box plots for running times were added for 50 runs for the dataset UKansasState and Boston. Figure 15 shows the Qualified RunTime Distribution (QRTD) for the two dataset for different values of error. Figure 16 shows the Solution Quality Distribution (SQD) for different values of runtime. Figure 17 shows the runtime box plot for different level of error. As we increased the error threshold, the box plots went down indicating lower runtime requirement to get lower error.

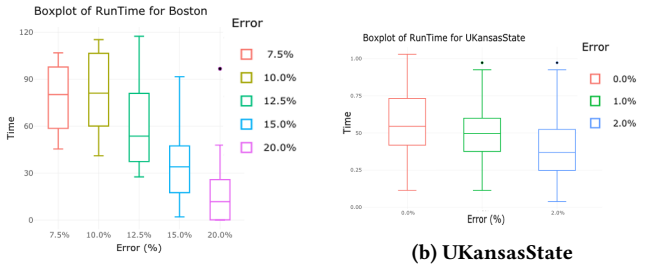
(a) Boston **(b) UKansasState**

Figure 12: QRTD plots for the two dataset



(a) Boston **(b) UKansasState**

Figure 13: RSQ plots for the two dataset



(a) Boston

(b) UKansasState

Figure 14: Box plots for the two dataset

It was observed that for smaller datasets like Atlanta, UKansasState and Cincinnati almost 0 error rate can be achieved in few seconds. For medium size dataset like Philadelphia, Toronto, Berlin the algorithm gave minimum of of 5-8% error with cut-off time of seconds. And for Boston and SanFrancisco alrogrithm was able to give minimum error of just above 10% in 30 seconds. For some runs on the larger datasets, the algorithm failed to performed better than Nearest Neighbour in under 30 seconds as the result failed to improve after the initial population generation using Nearest Neighbour.

5.3 Nearest Neighbor

The nearest neighbor algorithm was implemented in Python3. The experiments were performed on a personal laptop with 16 GB of RAM and 1.80 GHz Intel 4 Core i7 processor. Comprehensive table 4 shows the average time, solution quality, and relative error nearest neighbors performed on each dataset. There was no cutoff time chosen and each run time represents the time taken to construct a solution starting with each city and choosing the best tour. The algorithm will always arrive at the same solution for a given city and that solution will always be constructed in the same order (starting cities are always tested in the same order). Because of this, it was only necessary to run the algorithm on each city once. Any differences in time for multiple iterations would be system dependent and negligible.

Table 7: Results of Nearest Neighbor on different datasets

Dataset	Time(s)	Sol.Qual.	RelError
Atlanta	0.32	2039906	0.018
Berlin	0.85	8181	0.085
Boston	0.47	1029012	0.152
Champaign	0.87	61828	0.174
Cincinnati	0.10	301260	0.083
Denver	1.52	117617	0.171
NYC	1.18	1796650	0.155
Philadelphia	0.44	1611714	0.155
Roanoke	24.07	773359	0.179
SanFrancisco	2.87	857727	0.059
Toronto	3.70	1243370	0.057
UKansasState	0.14	69987	0.111
UMissouri	3.27	155307	0.170

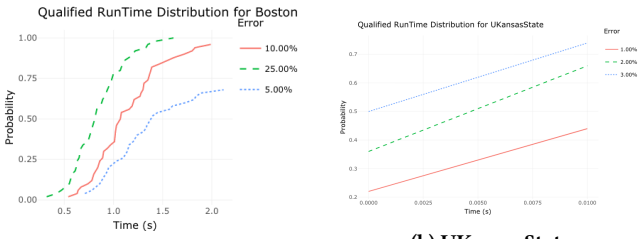
The results show that the Nearest Neighbor algorithm can perform very well in some cases. For small and mid-size cities, the algorithm runs very quickly. Even for the larger cities, the algorithm reaches approximate solutions quickly. It does not seem that solution quality is tied city size. For example, Toronto is a large city but the algorithm finds a solution with small relative error. On the other hand, some smaller instances like UKansas State had larger relative error. In general the error values were roughly between 5 and 15 percent with Atlanta being the lowest (1.8%) and Roanoke being the highest (17.9%).

5.4 2opt Local Search

The 2opt algorithm was implemented in Python3. The experiments were performed on a personal laptop with 8 GB of RAM and 2.7 GHz Intel Dual Core i5 processor. The run time was different on the datasets, 10 runs were performed for each dataset with varied random seeds and then the average errors and times were recorded. Comprehensive table 8 shows the average time, solution quality, relative error and selected cutoff-time for the 2opt algorithm performed on different datasets. Cutoff-time was chosen depending on the size of the dataset.

Table 8: Results of 2opt Algorithm on datasets. Cutoff-time varies per dataset size

Dataset	Time(s)	Sol.Qual.	RelError	Cutoff (s)
Atlanta	0.06	2003763	0.000	30
Berlin	3.00	8069	0.070	30
Boston	2.15	922658	0.033	30
Champaign	4.28	53988	0.026	30
Cincinnati	0.01	280243	0.008	30
Denver	26.98	105640	0.052	30
NYC	10.24	1854932	0.193	30
Philadelphia	0.34	1644766	0.178	30
Roanoke	2756.65	671893	0.025	3000
SanFrancisco	70.21	831188	0.025	300
Toronto	92.14	1350893	0.148	300
UKansasState	0.01	62962	0.000	30
UMissouri	52.01	138137	0.041	300

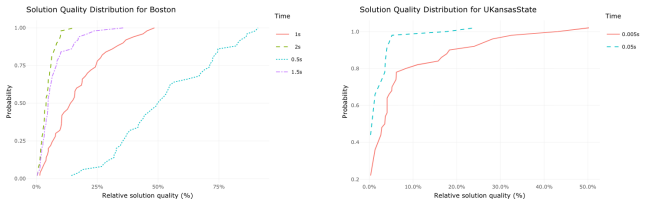


(a) Boston

(b) UKansasState

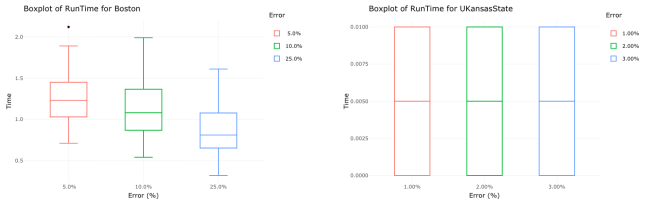
Figure 15: QRTD plots for the two dataset

We can see that for smaller datasets a 0 error rate can be achieved in just a few seconds.



(a) Boston **(b) UKansasState**

Figure 16: RSQ plots for the two dataset



(a) Boston **(b) UKansasState**

Figure 17: Box plots for the two dataset

6 DISCUSSION

Both the local searches outperformed the nearest neighbour for smaller datasets. This was mainly because of greedy nature of nearest neighbour approach. But for larger dataset, nearest neighbour gave almost equivalent results in less time compared to local search algorithms. One of the reason can be that initiating genetic algorithm population with some heuristic generated chromosomes led it to local optimal solution (similar to that of nearest neighbour) and it couldn't explore out of it in the assigned cut-off time of 30 seconds. When the genetic algorithm was tried without using some heuristic created chromosomes in the population, it took significantly higher time to converge. 2-opt local search found to be giving lower error for most of the dataset, although it ran with higher cutoff time.

The nearest neighbour gave the best solution it could, but both the local searches could have improved the result if more iterations were allowed. To prevent genetic algorithm from getting stuck in the local optimal, better diversity can be introduced in the population which can help in more exploration.

Each of these algorithms performed significantly better than the Branch and Bound approach to the Travelling Salesman problem within the given cutoff time of 600 seconds.

7 CONCLUSION

The Travelling salesman problem is a nuanced problem with multiple approaches and solutions. Over the course of this project we have explored four different algorithms and compared their results with respect to time and space complexities as well as accuracy of the solutions provided by each of these algorithms.

The Branch and Bound approach serves as a primary approach to the problem, performing marginally better than the Brute Force approach. While it takes a significant amount of time and computational resources to execute correctly on graphs that have a large number of nodes, we can be assured of the exactness of the results provided by this algorithm provided adequate time and computational resources.

A common observation for all algorithms that were run was that all of them gave lower performing results in terms of either accuracy or runtime for the dataset graphs with a large number of nodes like Roanoke and NYC. Large solution space can be one such reason.

Seeing the performance of different algorithms one approach of combinations of algorithms can be tried which should give better and faster results. Nearest neighbour can be used to generate the intial populations for the genetic algorithm. Once genetic algorithm completes its search and gives its final result, 2-opt algorithm can be used on that result with fewer iterations to check if there exists a better solution in the close proximity of the genetic algorithm generated results.

REFERENCES

[1] Jon Jouis Bentley. 1992. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on computing* 4, 4 (1992), 387–411.
[2] L. Davis. 1985. Applying adaptive algorithms to epistatic domains. *IJCAI* 85 (1985), 162–164.
[3] IEAC Droste. 2017. *Algorithms for the travelling salesman problem.* B.S. thesis.
[4] David B Fogel. 1988. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics* 54, Article 2 (1988), 139-144 pages.
[5] D. Goldberg and R. Lingle. 1985. Alleles, Loci and the Traveling Salesman Problem. *1st International Conference on Genetic Algorithms and Their Applications* 1985, Article 2 (1985), 154-159 pages.
[6] Pierre Hansen and Nenad Mladenović. 2006. First vs. best improvement: An empirical study. *Discrete Applied Mathematics* 154, 5 (2006), 802–817.
[7] Andrew B Kahng and Sherief Reda. 2004. Match twice and stitch: a new TSP tour construction heuristic. *Operations Research Letters* 32, 6 (2004), 499–509.
[8] G. Yun L. Tan, Y. Tan and Y. Wu. 2016. Genetic algorithms based on clustering for traveling salesman problems. *International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)* 12 (2016), 103–108.
[9] Christian Nilsson. 2003. Heuristics for the traveling salesman problem. *Linkoping University* 38 (2003), 00085–9.
[10] Hiroyuki Okano, Shinji Misono, and Kazuo Iwano. 1999. New TSP construction heuristics and their relationships to the 2-opt. *Journal of Heuristics* 5, 1 (1999), 71–88.
[11] W Richard. 2003. Branch and bound implementations for the traveling salesperson problem-part 1: A solution with nodes containing partial tours with constraints. *Journal of Object Technology* 2, 2 (2003), 65–86.