# Evaluation of traveling salesmen problem through various algorithms

Heejoo Cho
School of Computer Science
Atlanta, Georgia
joheeju@gatech.edu

Seong Wook Choi
School of Computer Science
Atlanta, Georgia
schoi330@gatech.edu

Frederick Chung
School of Civil and Environmental Engineering
Atlanta, Georgia
fchung3@gatech.edu

Myong Joon Kim
George W. Woodruff School of Mechanical Engineering
Atlanta, Georgia
mkim309@gatech.edu

## KEYWORDS

traveling salesman problem, branch and bound, local search, nearest neighbor, 2-opt exchange, simulated annealing

## 1 INTRODUCTION

The traveling salesman problem (TSP) is the challenge of figuring out the shortest distance to connect all the vertices by visiting each point once and returning to the starting point. The goal of this project is to utilize four different search algorithms for the TSP and assess their performances within a given time. The four algorithms that are explored are followings:

(1) Branch and Bound
(2) Nearest Neighbor
(3) 2-Opt Exchange
(4) Simulated Annealing

In this project report, the processes of the above algorithms are explained with pseudo-codes and time/space complexities. Then, an exhaustive comparison of the algorithms along with their achieved solutions and percent relative errors in the tables and box-plots to find best options in terms of accuracy and time.

### 1.1 Problem Statement and Motivation

The TSP problem is defined as follows: provided the $(x, y)$ coordinates of $N$ vertices in the plane, and a cost function $cost(u, v)$ defined for all pairs of vertices (or edges), find the shortest route possible that goes through all $N$ vertices. All edge costs satisfy the triangle inequality for this version of the TSP problem, and are symmetric. This problem can be applicable to the real-world issues such as X-ray crystallography, VLSI design, machine scheduling, vehicle routing, computational biology, and circuit board drilling.

### 1.2 Related Work

The TSP is visited by many researchers by numerous algorithms. In this section, some of the past efforts in using different algorithms are explored.

Little et al. proposed the branch and bound algorithm to solve the traveling salesman problem. The key step for their algorithm is to generate an adjacency matrix of distances to each pair of nodes and reducing the matrix to calculate the lower bound for partial solutions [4]. More recent works contributed to improving the performance by implementing other algorithms to solve the partial solution faster and setting more robust lower bounds [1, 6].

The nearest neighbor algorithm is one of the most straightforward TSP heuristic. This method has been widely explored in its original form and with large variations such as hybrid heuristic approach, multiple nearest neighbor, and dual nearest neighbor to name a few [8]. Johnson et al. showed that the nearest neighbor method would often keep its route within 25 percent of the Held-Karp lower bound [2].

Steiglitz and Weiner had observed that the search of 2-opt exchange can be pruned for two selected edges to swap, $(n_1, n_2)$ and $(n_3, n_4)$, if the distance between $n_1$ and $n_2$ is not greater than $n_2$ and $n_3$ [7]. The process

is originally a naive method, but by choosing a certain number ($m$) of nearest neighbors of each vertex, the time complexity can be reduced to $O(mn)$ from $O(n^2)$. The speed and accuracy will have to be balanced for the best outcome.

Liu et al. propose a modified cooling schedule, which allows simulated annealing to escape local optimum [5]. Traditional cooling schedules either decrease the temperature linearly or exponentially by a specific factor. While these schedules are simple to track, they cannot avoid local optimum if the algorithm reaches one when the temperature is already low. A modified cooling schedule proposed in the paper increases the temperature in proportion to the number of consecutive upward moves. Therefore, the longer the algorithm remains in a local optimum, the higher chance it will have to escape it. Details on the implementation of this modified cooling schedule will be further explained in a later section.

# 2 APPROACH

## 2.1 Approach 1: Branch and Bound

The branch and bound algorithm is structured to find the optimal path for TSP through exhaustive search. The algorithm finds the solution by generating a tree of possible solutions. It enumerates partial solutions to calculate the tour costs of various possibilities and to search for the path that generates the least cost. An advantage of using branch and bound algorithm is to find the optimal solution without calculating all possible solutions. While the Brute Force algorithm finds the optimal solution after calculating all possible solutions, the branch and bound only explores the paths that may have lesser cost than the solution found previously. To reduce the computational cost to solve the problem, the algorithm uses the least cost found so far to determine if it is worthwhile to explore the branch to find the best solution. If it is determined that exploring a certain branch further would not generate a better solution, the algorithm prune the sibling branches with bounding functions and moves on to explore other branches. When the algorithm finds a path that has lower cost than the one found previously, the least cost found so far gets updated and be used for the further calculations.

Rather than solving for every travel tour, the branch and bound algorithm calculates the partial solution first before identifying if the path is worthwhile to extend further. The partial solution is found by calculating the minimum possible cost to complete the partial path. In each partial solution, the cost of an edge is included into the partial solution found previously and compare it to the lower bound solution. The algorithm finds the partial solution by reducing the adjacency matrix. Reducing matrix refers to subtracting the minimum value of a row to all the elements in the same row. Similarly, after all the rows are reduced, the columns are reduced by subtracting the minimum value of a

**Algorithm 1:** Branch and Bound

```
Input: BnB(nodes)
Output: cost, tour
Global lowerBound
Global leastPath
lowerBound=0.5*sum of each nodes' two
  shortest edge
adjMat=2-D adjacency matrix that calculates
  distance of each node.
//  Run recursive function.
currPath=[]
currDepth=1
Function searchPath(adjMat, currPath,
  currDepth)
    Global lowerBound
    Global leastPath
    pathLength=len(adjMat)
    if currDepth == pathLength : then
        tempLB ←
          current edge cost + partial solution
        if tempLB == lowerBound : then
            lowerBound ← tempLB
            leastPath ← tempPath
        end
    end
    for iinrange(pathLength) : do
        tempLB ←
          current edge cost + partial solution
        if tempLB < lowerBound : then
            append node i to the currPath
            searchPath(adjMat, currPath,
              currDepth+1)
        end
    end
    return cost, tour
end
```

column to all elements in its column. The minimum possible cost for the partial tour can be calculated by summing up the values that were subtracted in all rows and columns and the cost required in previous steps. If the partial solution is lower than the lower bound, the partial solution will be extended and explore the path further. On the other hand, if the partial solution is higher than the lower bound, the path is discarded and stops to explore the path to avoid solving infeasible solutions.

The space complexity of the algorithm is O($n^3$). The algorithm generates adjacency matrix, which requires $n^2$. As the algorithm runs, the adjacency matrix gets reduced. In each node, cost of edges to every node should be calculated. This requires n number of adjacency matrices. Therefore, the space complexity is O($n^3$). The time complexity is O($n!$). In the worst case, the paths can be explored in the order of worst case. This means the algorithm would require to explore all the paths to find the optimal solution. But, in practice, it would be rare that all solutions need to be explored. Although the worst is time complexity would be O($n!$), the algorithm will perform better in general.

## 2.2 Approach 2: Heuristics - Nearest Neighbor

The heuristic approximation algorithm chosen for the project is nearest neighbor algorithm. This algorithm is a family of constructive search heuristics, which means it gradually designs a routine. The pseudo-code of the algorithm is provided below as Algorithm 2: construction heuristics: nearest neighbor. First, the algorithm randomly selects a start vertex from a given dataset and adds to an empty list. Then, the distance from the start vertex to the rest is calculated, and the shortest vertex is added to the list. In this step, any vertex added to the list is removed from the vertices and selection.

In the algorithm implementation, all unvisited vertices are stored as a list, and the distance from the current vertex to the unvisited vertex is calculated every time the current vertex is newly assigned. Therefore, the time complexity is $O(n^2)$, and space complexity is $O(n)$. Each time a vertex is added, the distance from the current vertex to the new nearest neighbor is added to the cost variable.

**Algorithm 2:** Nearest Neighbor

```
Input: nn(nodes)
Output: cost, tour
Function nn(N,S)
    curr ← randomly selected node from nodes
     unvisited ← nodes except curr
     tour, cost ← [curr], []
    while len(unvisited) > 0 : do
        next ← node w/ shortest dist from curr
        append distance of curr-next to cost
        remove next from unvisited
        append next to tour
        curr ← next
    end
    return cost, tour
end
```

We have to use approximate nearest neighbor (ANN) for practicality. The cost and order of visit for the exact nearest neighbor method will be very large with massive datasets and high dimensionality. The linear search would be exhaustive search, therefore, approximate nearest neighbor (ANN) is necessary. This algorithm will have clear advantage over the branch and bound algorithm for speed but will sacrifice accuracy. Fortunately, the approximation provides the bounds for the worst case scenario, which is refered to as "approximation guarantees". A reasonable solution is determined when the approximation ratio r is greater than or equal to 1 which is specified by the following equation.

$$max\left(\frac{A(n)}{OPT(n)}, \frac{OPT(n)}{A(n)}\right) \leq r \tag{1}$$

where n is an input size, $A(n)$ is the solution obtained from ANN search technique, and $OPT(n)$ is the optimum solution for the problem. As shown from above equation, this has a limitation that no ANN can guarantee the approximation if the ground truth optimum is unknown. However, the accuracy target can be achieved by appropriately optimizing system parameters of the search technique depending on the application, such as the decision tree structure, or training iterations, etc [3]. Therefore, heuristics ANN can be a powerful tool for the given problem when further optimized.

## 2.3 Approach 3: Local Search

In the last two approaches, two local search algorithms are explored to solve the traveling salesman problem: 2-opt exchange and simulated annealing. Local searches have an advantage of being relatively fast when compared to exhaustive search algorithms. When implemented properly, they often converge quickly to a solution which is often acceptable in real life situations. Disadvantages of the local search algorithms include possibility of falling into a local optimum, and not returning an exact solution to the problem.

*2.3.1 Approach 3.1: 2-Opt Exchange.* A 2-opt method is a part of the family of local search algorithms. It starts with an initially established traveling route and seeks improvements by iterations in the vicinity of the solution. The initial solution could be any solution, and in this particular study, randomized nearest neighbor (RNN) has been used. Starting from a vertex, the subsequent vertices are randomly chosen from three nearest neighbors to the current vertex and appended to create a solution. [9] We arbitrarily chose 3 as the size of neighborhood to greedily search for a short path and to escape from local optima by keeping randomness.

From this on, two different arcs are randomly chosen from the route, and they are swapped to recalculate new travel distance (cost). If the newly connected path has less cost ($cost_{new} < cost_{original}$), the original route will be replaced with the swapped route; otherwise, discarded. This process repeats until there is no improvement within $n^2$ trials to swap.

After the algorithm stops with no further improvement with the route, we again generate a new route using RNN and repeat all the processes until a set time is up.

**Algorithm 3:** 2-Opt Algorithm

```
Function 2-opt(cost, tour, nodes)
    Randomly select two edges that are not adjacent
    Swap the two edges and calculate new cost
    if new cost < cost: then
        update cost
        update tour
    end
    return cost, tour
```

Once we get a lower cost after swap, we need to update the tour by reversing the path between the two edges, and it takes $O(n)$ time for each run. While the theoretical algorithm compares every possible combination of two nodes to find the smallest distance to swap, which takes $O(n^2)$ time, we implemented the algorithm to randomly select two nodes to reduce the process time. Assuming we perform in-place inversion for the new tour, it requires constant space, $O(1)$.

**Algorithm 4:** Local Search 1 Algorithm

```
Input: nodes, runtime
Output: cost, tour
cost ← ∞
while runtime is not over do
    tour ← RNN(nodes)
    trial ← 0
    while trial < (num of nodes)^2 do
        cost, tour ← 2-opt(cost, tour, nodes)
        if cost improved then
            trial ← 0
        else
            trial ← trial + 1
        end
    end
end
return cost, tour
```

*2.3.2 Approach 3.2: Simulated Annealing.* Simulated annealing is another variation of a local search algorithm, which attempts to avoid local optima using a cooling schedule. It was chosen for this project due to its ability to escape local optimum, and also relative ease in implementation of the algorithm. A solution is randomly initialized in conventional simulated annealing, and a potential solution is looked for following a solution search algorithm. It accepts the newly found solution if it is better than the previous one. Even if the possible solution does not prove to be a better solution, the algorithm calculates a probability based on the cooling schedule and still accepts the potential solution with the computed likelihood. Taking non-promising solutions allows simulated annealing to avoid local optima with the introduced randomness. Few modifications have been made to the traditional simulated annealing algorithm in its solution search algorithm and cooling schedule to increase the possibility of avoiding a local optima and converging towards a global optimum.

The simulated annealing algorithm chooses the best solution from three possible candidates in this study. The three candidates are swap, reverse block and insert. The first step to all three candidates involves randomly selecting two vertices from the graph. The inputs to all three functions are identical. The current order of the vertices and orders of the two selected vertices. The first location is always less than the second location. Swap simply switches the order of visiting the two selected vertices.

**Algorithm 5:** Swap

```
Function swap(path, vertex1, vertex2)
    newPath = path.copy()
    newPath[vertex1] ← path[vertex2]
    newPath[vertex2] ← path[vertex1]
    return newPath
```

Reverse block switches orders of all the vertices in between the two vertices.

**Algorithm 6:** Reverse Block

```
Function revBlock(path, vertex1, vertex2)
    newPath = path.copy()
    flip(newPath[vertex1 + 1 : vertex2])
    return newPath
```

Insert moves the order of the first selected vertex to the order of the second selected vertex.

**Algorithm 7:** Insert

```
Function insert(path, vertex1, vertex2)
    newPath = path.copy()
    newPath.insert(path[vertex1], vertex2)
    newPath.remove(vertex1)
    return newPath
```

Out of the three newly found solutions, the final candidate solution is the one with minimum travel distance.

A modified cooling schedule used in this study can be represented in Equation 2.

$$T_i = T_{min} + \lambda ln(1 + r_i) \tag{2}$$

$T_i$ is the temperature at iteration i, while $T_min$ is the minimum possible temperature for the algorithm. In this study, $T_{min}$ is 1 for all instances. $\lambda$ is a constant which varies depending on the number of nodes and the distance of the initialized path. For this project $\lambda = distance/(\#nodes)^2$ is used. $r_i$ represents the number of consecutive upward moves at iteration i, therefore longer the algorithm remains in a local optimum, higher the temperature will be. Probability of accepting non-optimal solution is calculated based on Equation 3.

$$P = exp\left(\frac{Distance - newDistance}{T_i}\right) \tag{3}$$

With all the attributes mentioned above combined, the pseudo-code for this version of simulated annealing is represented in Algorithm 8.

Since simulated annealing randomly chooses two vertices at any given iteration and only returns the solution when the cut-off time is reached, an exact time complexity cannot be specified. However, the most costly computation occurs when calculating the total distance of the newly found path, which is in $O(n)$. Since the algorithm requires keeping track of the best found solution so far, the space complexity is also $O(n)$.

**Algorithm 8:** Simmulated Annealing

```
Function SA(nodes)
    dist, path ← randomInitialize(nodes)
    minDist ← inf
    bestPath ← None
    T_min ← 1
    λ ← dist/len(nodes)^2
    r_i ← 0
    while cut off time is not reached do
        newPath ← bestSolution(path)
        newDist ← findDist(newPath)
        if newDist < dist then
            r_i ← 0
            path = newPath
            dist = newDist
            if minDist > dist then
                minDist ← dist
                bestPath ← path
            end
        else
            r_i ← r_i + 1
            T_i = T_min + (1 + r_i)
            prob ← exp((dist − newDist)/T_i)
            with prob (path
              ← newPath, dist ← newDist)
    end
    return minDist, bestPath
```

**Table 1: Comprehensive Table for Branch and Bound and Heuristic**

| | | | Branch and Bound | | | Heuristic | | |
|---|---|---|---|---|---|---|---|---|
| Instance | # vertices | OPT | Time(s) | Sol.Qual | RelErr (%) | Time(s) | Sol.Qual | RelErr (%) |
| Atlanta | 20 | 2003763 | 600 | 2497224 | 24.63 | 0.01 | 2039906 | 1.80 |
| Berlin | 52 | 7542 | 600 | 19448 | 157.86 | 0.43 | 8181 | 8.47 |
| Boston | 40 | 893536 | 600 | 2227323 | 149.27 | 0.02 | 1029012 | 15.16 |
| Champaign | 55 | 52643 | 600 | 218074 | 314.25 | 0.03 | 61828 | 17.45 |
| Cincinnati | 10 | 277952 | 0.28 | 277952 | 0 | < 0.01 | 301260 | 8.39 |
| Denver | 83 | 100431 | 600 | 562770 | 460.35 | 0.99 | 117617 | 17.11 |
| NYC | 68 | 1555060 | 600 | 7159769 | 360.42 | 0.96 | 1796650 | 15.54 |
| Philadelphia | 30 | 1395981 | 600 | 3624919 | 159.67 | 0.05 | 1611714 | 15.45 |
| Roanoke | 230 | 655454 | 600 | 6857992 | 946.30 | 21.59 | 773359 | 17.99 |
| SanFrancisco | 99 | 810196 | 600 | 5600573 | 591.26 | 1.56 | 857727 | 5.87 |
| Toronto | 109 | 1176151 | 600 | 9184579 | 680.90 | 0.7 | 1243370 | 5.72 |
| UKansasState | 10 | 62962 | 0.4 | 62962 | 0 | < 0.01 | 69987 | 11.16 |
| UMissouri | 106 | 132709 | 600 | 668324 | 403.60 | 2.94 | 155307 | 17.03 |

**Table 2: Platform Specifications**

| | Specs |
|---|---|
| CPU | Ryzen 5 3600 @ 3.6 GHz |
| RAM | 16 GB |
| Language | Python 3.7 |
| Compiler | GCC 7.5.0 |

# 3 EMPIRICAL EVALUATION

Table 2 represents a detailed specification of the platform used for empirical evaluation of the algorithms.

Results for each of the four algorithms were tabulated in Table 1 and Table 3. For branch and bound and heuristic algorithm, each city was ran for 600 seconds, and its time, solution, and relative errors were calculated in Table 1. Table 3 displays the empirical results averaged over 10 runs of both local search algorithms with a cut-off time of 60 seconds. The time shown in Table 3 for each city stands for the time the algorithm found the best solution during the run time. Qualified Runtime Distributions (QRTDs), Solution Quality Distributions (SQDs) and boxplots in Figure 1 to 12 are created using 50 runs of the instances with the two local search algorithms. LS1 refers to 2-Opt Exchange algorithm, and LS2 refers to simulated annealing. For better readability of the report, all figures have been attached to the appendix at the end.

## 3.1 Branch and Bound

The branch and bound algorithm was operated for 600 seconds to collect the solution and relative errors. As shown in Table 1, the required amount of time becomes large even with moderate size of data. The initial lower bound for branch and bound algorithm is the half of summation of the two least edges of each node. The branch and bound algorithm can only solve for two cities, Cincinnati and UKansasState (10 nodes), with a time limit of 600 seconds. The branch and bound requires a lot of more time to solve for the other cities and did not generate a reasonable solution within the 600 seconds. The relative error of the solution for Atlanta, which had 20 points, is found to be about 25 percent. The relative error gets worse as the number of nodes increases. The main reason for the relative error to be high is not because the algorithm is not capable when dealing with large data file but because the algorithm did not have sufficient time to explore many branches. Although most of the time the complexity of the algorithm would not be $O(n!)$, it still required a lot of time to identify the optimal solution.

## 3.2 Heuristic Algorithm

Table 1 also shows the comprehensive results computed from nearest neighbor algorithm (heuristic). In most cases, the time it takes for the nearest neighbor method

**Table 3: Comprehensive Table for 2-Opt Exchange and Simulated Annealing**

| | | | 2-Opt Exchange | | | Simulated Annealing | | |
|---|---|---|---|---|---|---|---|---|
| Instance | # vertices | OPT | Time(s) | Sol.Qual | RelErr (%) | Time(s) | Sol.Qual | RelErr (%) |
| Atlanta | 20 | 2003763 | 0.02 | 2003763 | 0 | 0.39 | 2003763 | 0 |
| Berlin | 52 | 7542 | 28.53 | 7542 | 0 | 20.20 | 7564 | 0.31 |
| Boston | 40 | 893536 | 42.61 | 893536 | 0 | 6.01 | 893777 | 0.03 |
| Champaign | 55 | 52643 | 22.24 | 52643 | 0 | 30.97 | 52657 | 0.03 |
| Cincinnati | 10 | 277952 | 0.01 | 277952 | 0 | 0.03 | 277952 | 0 |
| Denver | 83 | 100431 | 43.53 | 102635 | 2.19 | 27.22 | 101607 | 1.17 |
| NYC | 68 | 1555060 | 43.50 | 1558247 | 0.20 | 27.13 | 1610738 | 3.58 |
| Philadelphia | 30 | 1395981 | 0.72 | 1395981 | 0 | 3.20 | 1395981 | 0 |
| Roanoke | 230 | 655454 | 4.07 | 697241 | 6.38 | 57.47 | 688566 | 5.05 |
| SanFrancisco | 99 | 810196 | 8.04 | 810503 | 0.04 | 40.99 | 839381 | 3.60 |
| Toronto | 109 | 1176151 | 42.89 | 1176323 | 0.01 | 34.093 | 1234818 | 4.99 |
| UKansasState | 10 | 62962 | 0.01 | 62962 | 0 | 0.01 | 62962 | 0 |
| UMissouri | 106 | 132709 | 57.56 | 135561 | 2.15 | 40.62 | 136629 | 2.95 |

to reach the best solution for each town is significantly lower than any other methods used in the study. Except for the city, Roanoke, all cities reach their best solution within less than 3 seconds. When the relative errors of the four algorithms are compared, the errors from the nearest neighbor are higher than those of the local search algorithms but lower than those of the branch and bound algorithm. There are also two exceptions in instances (Cincinnati and UKansasState) that show higher relative errors in the nearest neighbor than the branch and bound results. Five of the cities have shown the relative error of less than 10 percent, and the rest are less than 18 percent. Based on the tabulated results, it can be seen that a longer running time does not guarantee its improvements in the relative error. Also, there can be a certain set of nodes that works better on one method over others.

## 3.3 2-Opt Exchange

For cities with relatively small number of vertices, the 2-opt algorithm finds the optimal solution while it gives an sub-optimal solution for other cities with increasing relative errors as the number of vertices increases. Now, we analyze the 2-Opt Exchange with the result from 50 runs of the local search algorithm for two cities: Boston and Denver. As we can see the QRTDs in Figure 1 and 7 for both cities, most runs quickly find decent sub-optional solutions in a short time; however, as better solution quality is required, it takes more time to achieve. One interesting thing is even though the number of nodes is twice as many Denver as Boston, the plots for the cities draw similar shapes within the given time. (Note that the time given for Denver is twice of the time for Boston.)

On the other hand, SQDs in Figure 2 and 8 for Denver and Boston have different shape; lines for Denver draws S-shape while those for Boston are not. There are many factors making this difference, and one can be the fact that the given runtime for Denver is not enough to safely reach the optimal solution in terms of the size of the city ($n$); 80% of runs for Boston reach to the optimal within 30 seconds while most runs for Denver barely do within 120 seconds. However, note that the plot for Denver is the one we would see in practice as the general use of 2-Opt Exchange is to improve a route that is generated from a heuristic for a number of cities with which branch and bound algorithm takes an exponential time. Therefore, the the SQD for Denver in Figure 8 describes well how the algorithm behaves in general: the algorithm finds pretty good solutions in a short time, but the improvement gets slower as it is close to the optimal since the algorithm randomly searches for two edges, which is quick but does not have a certain sequence of combination not to miss a good pair to swap.

From Figure 3 and 9, we can see that the plots have wider range as lower solution quality is required since again the algorithm doesn't precisely pick a good pair to swap. Also, there are outliers with the fact that it takes time to escape from local optima using randomness.

## 3.4 Simulated Annealing

As is the case for the 2-Opt Exchange algorithm, simulated annealing also returns a solution with relatively low error in a fairly short amount of time. Although the simulated annealing method is not a exhaustive search algorithm, it still returns the global optimum for instances with number of vertices less than or equal to 30, as well as solutions within 2% error for most of the instances, excluding the ones that have number of vertices greater than or equal to 99 and NYC. One interesting thing to note from the table is that while the the relative error is generally higher for instances with higher number of vertices, NYC has about twice the relative error when compared to Denver, which has higher number of vertices. This is most likely due to the fact that simulated annealing was trapped in a local optimum when computing NYC, which happen to have somewhat high relative error, and could not escape that local optimum within 60 seconds, which is the cut-off runtime.

Results from Figure 5 and 11, SQDs for Boston and Denver, illustrate the algorithm's ability to escape local optima, due to the implementation of the modified cooling schedule as mentioned previously. This in turn indicates that longer the runtime is, higher the chance would be for the algorithm to converge to a global optimum. Another aspect of the algorithm that can be observed from Figure 4 and 10, QRTDs for Boston and Denver, is the algorithm's performance dependency on the number of vertices that the instance has. In the case of Boston, which has 40 vertices, approximately 80 percent of the 50 runs reach a relative solution quality of 0.1% in 30 seconds. With Denver, however, only about 50 percent of the 50 runs reach a relative solution quality of 1% even after 60 seconds since it has twice as many vertices compared to Boston.

And finally Figure 6 and 12, box plots for Boston and Denver, illustrate how the variation in the runtime grows wider as we apply more strict solution quality bounds to the algorithm. Even with less strict solution quality bounds, there are outliers that are more than 1.5 interquartile range away from the third quartile. Both of these observations are due to the fact that two vertices are selected at random on each iteration, thus making the algorithm highly dependent on the quality of the random choices.

# 4 DISCUSSION

The branch and bound algorithm is able to calculate the optimal solutions. Due to the exhaustive search to find the optimum, the algorithm requires excessive amount of time to reach the optimal solutions for the data. For the Cincinnati and UKansasState data, which had ten nodes to visit, took less than one second to find the optimal solution. However, for the data with equal or more than 20 nodes, the optimal solution could not be found within 600 seconds, which was the time limit for this project. Calculating the optimal solution for the Atlanta data that had 20 nodes required about 3000 seconds. The time required for the branch and bound algorithm increases excessively because the time complexity of the algorithm is factorial ($O(n!)$). Although it is unlikely for the algorithm to operate in $O(n!)$, the number of branches that the algorithm needs to visit becomes enormous as the number of nodes increases. The branch and bound algorithm has a benefit of calculating the exact optimal solutions. Therefore, if time and resources are allowed, using branch and bound algorithm could be considered to calculate more accurate results compared to the other algorithm developed in this project. However, considering the time required to achieve the solution, the use of branch and bound algorithm is impractical unless the optimal solution is needed.

The nearest neighbor has shown well balanced results between the time and accuracy. This method is not spending as much time as the local search algorithms and resulting less accuracy. This particular method is special in a way that the accuracy is determined by the initial point of travel. Therefore, within a given time of optimization, each node will become the initial point and cost will be compared with the subsequent cost calculated. Because of this characteristics, the time it takes to reach its convergence completion time at most linearly depends on the number of nodes as shown in the empirical results. The nearest neighbor could be a fast way to establish a solution that could be fed into a local search algorithm to further optimize. There could

be instances where the solution cost could be different even with the same initial vertex. When two or more vertices are positioned at exactly same distance from a vertex, the solutions would have to branch out at the vertex and be compared among each other. This will change the time complexity of the algorithm, but none was observed in the given example problems for the project.

The 2-Opt Exchange can be a great method to improve a given route within a relatively short time if its appropriate variant (e.g 3-Opt) and parameters (e.g max number of trials) are used for specific data; to escape from local optima, we need randomness, and to avoid inefficient selection, we need to filter out bad pairs. With the algorithm we implemented for 2-opt exchange, it is hard to expect to reach the global optimum as the size of nodes increases since it only uses randomness. On the other hand, for cities whose number of nodes is in a certain range, it effectively gets improvement.

When compared to 2-Opt Exchange, simulated annealing lacks in terms of finding a solution with less relative error within a given time limit. However, from Figure 9 and 12 we can observe that the variation is much greater for 2-Opt Exchange when compared to simulated annealing, which means that simulated annealing generally returns more consistent results. This is because 2-Opt Exchange implemented in this study is highly dependent on random restart as a method of escaping the local optimum, while the modified simulated annealing attempts to diverge away from the local optimum slightly to escape it rather than attempting to generate a totally new path to run the algorithm again on. In addition there are multiple implementations that can further improve the simulated annealing algorithm such as adopting random restart when the algorithm is trapped in a local optimum for a certain amount of time, introducing a tabu list of already visited solutions to reduce unnecessary computation or developing a better cooling schedule to avoid the local optimum while still converging quickly to a acceptable solution. These additions will most likely further increase the performance as well as the consistency of the algorithm, making it a generally a better choice to address the traveling salesman problem.

# 5 CONCLUSION

The objective of this project was to implement four algorithms to solve the Traveling Salesman Problem (TSP) and compare the performance of the algorithms. The TSP is one of the popular NP-hard problem that is to find the path to visit all the given nodes with the minimum cost. Robust algorithms can be implemented to find the solution for TSP in a reasonable accuracy and speed. The four algorithms utilized to solve the problem are branch and bound, heuristics algorithm (nearest neighbor), and two local search algorithms (2-Opt Exchange and simulated annealing). The branch and bound has the ability to find the optimal solution but took the longest time to operate. The heuristics algorithm calculated the solution much quicker (majority of solutions were calculated less than a second) with moderate amount of relative errors. The local search algorithms typically took a little longer but generated improved quality solutions compared to the heuristics algorithm.

The takeaway of this project is that a problem can be solved using multiple approaches. It was found that typically there is a trade off between solution quality and the required time when choosing an algorithm to solve problem. If one needs to solve for the absolute optimal solutions, it is recommended to choose the branch and bound algorithm. Sufficient time and high-performance computer would be required to solve for the optimal algorithm. However, if one needs to retrieve solution quickly within an acceptable solution quality, using the local search algorithms would produce reasonably accurate solutions within a relatively short time. As most practical applications consider time as the most valuable resource, we suspect local search algorithms to be more predominantly used in reality.

# REFERENCES

[1] M. Bazarra and J. Goode. 1977. The traveling salesman problem: A duality approach. *Mathematical Programming* 13, 13 (1977), 221–237. https://link.springer.com/article/10.1007/BF01584338

[2] DS Johnson and LA McGeoch. 1995. The Traveling Salesman Problem: A Case Study in Local Optimization, November 20, 1995. *Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches* 21 (1995).

[3] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *Proceedings of the 2020 ACM SIGMOD International Conference*

*on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2539–2554. https://doi.org/10.1145/3318464.3380600

[4] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. 1963. An Algorithm for the Traveling Salesman Problem. *Operations Research* 11, 6 (1963), 972–989. http://www.jstor.org/stable/167836

[5] Yi Liu, Shengwu Xiong, and Hongbing Liu. 2009. Hybrid simulated annealing algorithm based on adaptive cooling schedule for TSP. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. 895–898.

[6] Elias Munapo. 2013. A network branch and bound approach for the traveling salesman model. *South African Journal of Economic and Management Sciences* 16 (01 2013), 52 – 63. http://www.scielo.org.za/scielo.php?script=sci_arttext&pid=S2222-34362013000100005&nrm=iso

[7] Christian Nilsson. 2003. Heuristics for the traveling salesman problem. *Linkoping University* 38 (2003), 00085–9.

[8] Cheng-Fa Tsai, Chun-Wei Tsai, and Ching-Chang Tseng. 2004. A new hybrid heuristic approach for solving large traveling salesman problem. *Information Sciences* 166, 1-4 (2004), 67–81.

[9] Mikko Venhuis. 2019. The basics of search algorithms explained with intuitive visualisations. (2019). https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552

Evaluation of traveling salesmen problem through various algorithms

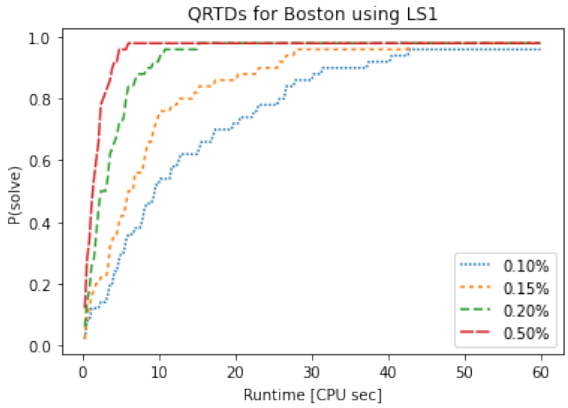


**Figure 1**

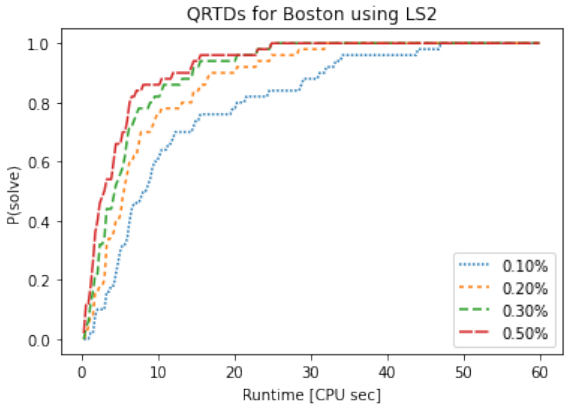


**Figure 4**

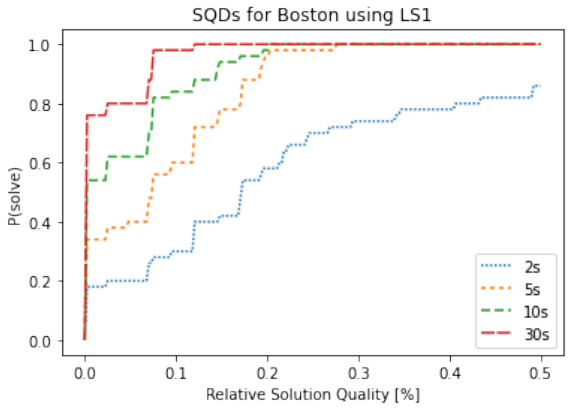


**Figure 2**

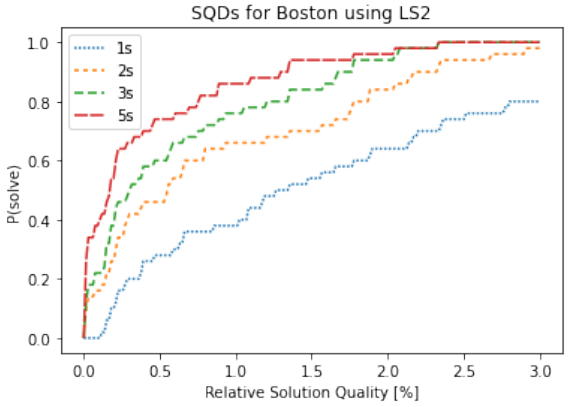


**Figure 5**

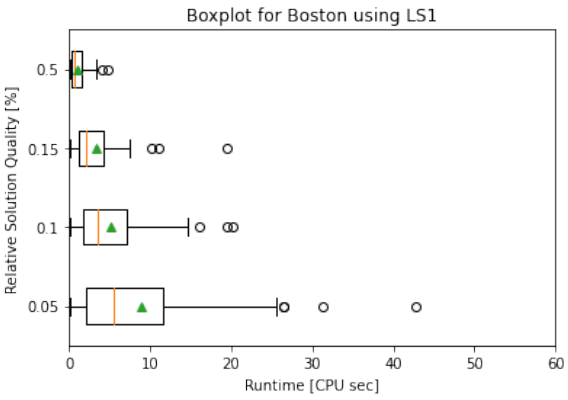


**Figure 3**

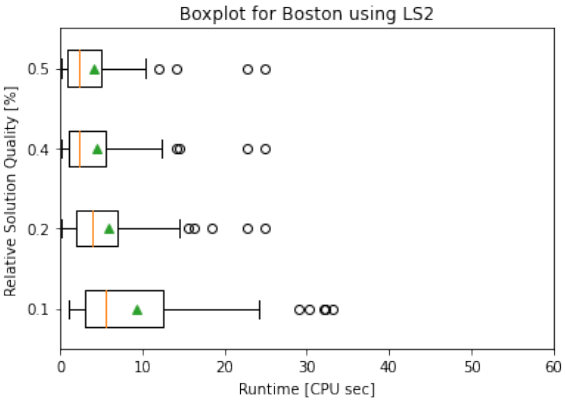


**Figure 6**

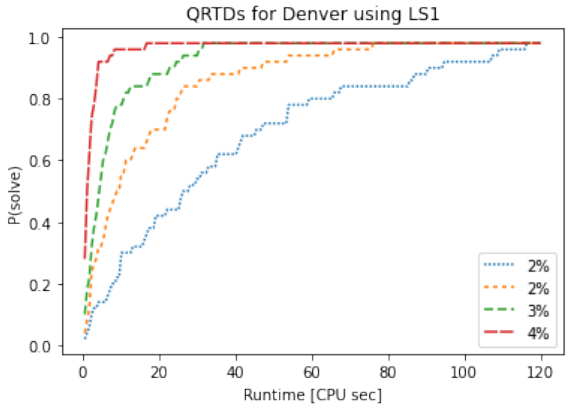


**Figure 7**

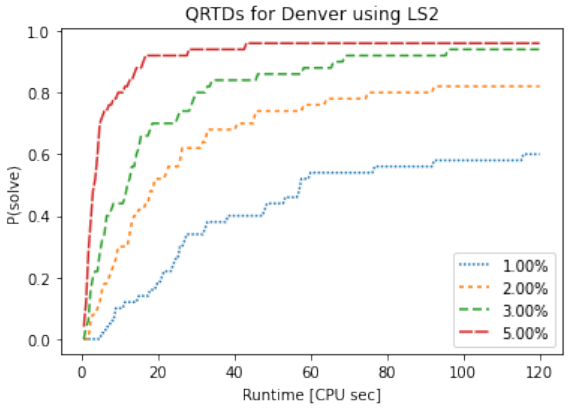


**Figure 10**

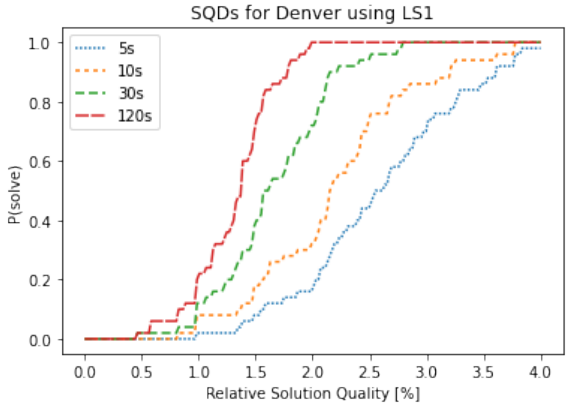


**Figure 8**

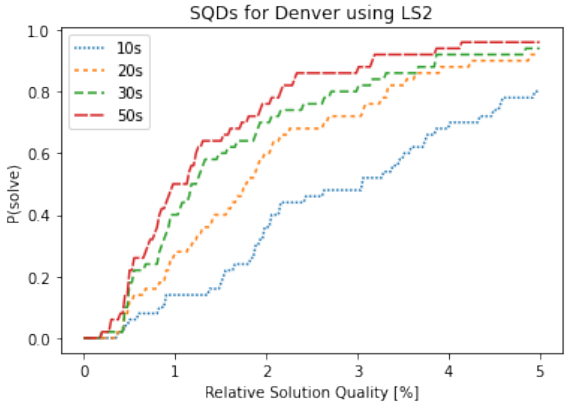


**Figure 11**

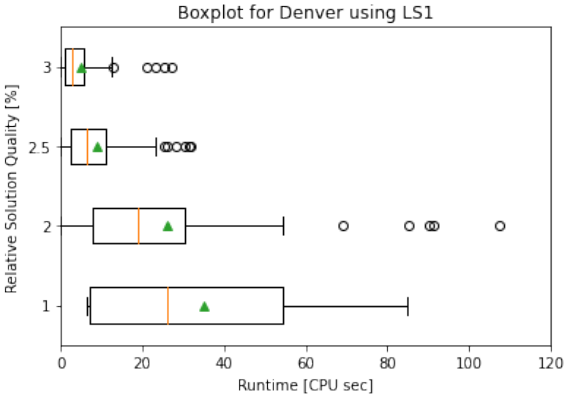


**Figure 9**

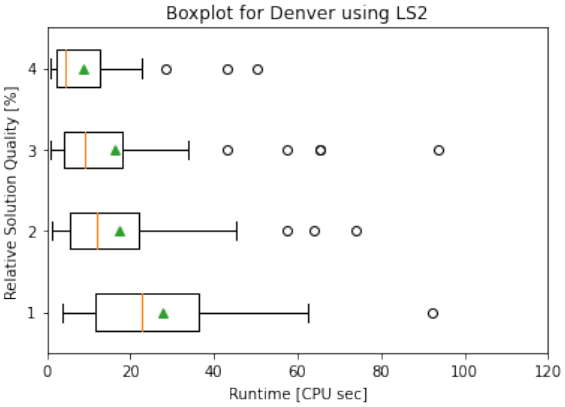


**Figure 12**