CSE6140 2021FALL Final Project

Algorithms for The Traveling Salesman Problem (TSP)

Shuhan Yang[1], Hanting Su[1], Xianyi Nie[1], and Guanming Chen[1]

[1]Georgia Tech

February 19, 2022

Abstract

The travelling salesman problem (TSP) is an NP-Hard optimization problem asking the question: "Given a list of cities and the distances between each pair of cities, what is the shortest route that visits each city and returns to the origin city?". In this paper, problem definition is introduced first and four used algorithms (branch and bound, nearest neighbor, two local search algorithms (Hill Climbing and Simulated Annealing) are under detailed description. Then the performance analysis of each algorithm is performed and the results are discussed for comparative numerical analysis.

Keywords Travelling Salesman Problem, Branch and Bound, Nearest Neighbor, Local Search

1 Introduction

The Traveling Salesman problem(TSP), is to find the least cost travel route from all the travel routes of graph G. There are (n-1) travel routes from the initial point, which is equal to the number of permutations of n-1 nodes except the initial node, so the travel quotient problem is a permutation problem. This problem is quite computationally intensive if used with the enumeration method and grows exponentially with the number of cities. For this reason, we compare four methods, namely, Branch and Bound, Branch and Bound, Local Search-Hill Climbing and Local Search-Simulated Annealing.

2 Problem Definition

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

3 Related Work

Branch and Bound

Branch and Bound algorithm solves TSP problem by using a set of subproblems. [1] We can present the process of constructing solutions as a decision tree. [2] It creates state-space for all nodes and expend on it. Each node is such tree refers to a subset of the solution set. A lower bound can be established using it.

Construction Heuristic

Construction Heuristics is a method that constructs the approximation tour by a sequence of steps in which tours are constructed for gradually larger subsets of the nodes. [3]. In the book "An analysis of several heuristics for the traveling salesman problem", different insertions are clearly explained. We have nearest insertion, which we chose to use in this problem, cheapest insertion and farthest insertion.
In nearest insertion, the next node is for which the cost is minimum among all the vi sited nodes that are not visited.
Given a subtour T and a node s, we define the distance $d(T, s)$ between T and s as

$$min\{d(x, s) \text{ for } x \text{ in } T\}$$

We say a tour is constructed by *nearest insertion* if each a_i, $1 \leq i \leq n$, satisfies

$$d(T_i, a_i) = \min\{d(T_i, x) \text{ for } x \text{ in } N - T_i\}[3]$$

In cheapest insertion, we choose the next node so that the total path length increases the least. We say a tour is constructed by *cheapest insertion* if a_i satisfies

$$\text{COST}(T_i, a_i) = \min\{\text{COST}(T_i, x) \text{ for } x \text{ in } N - T_i\}[3]$$

In farthest insertion, opposite to the nearest insertion, we find the farthest node among all the nodes that are not visited. We say a tour is constructed by *farthest insertion* if a_i satisfies

$$d(T_i, a_i) = \max\{d(T_i, x) \text{ for } x \text{ in } N - T_i\}[3]$$

Local Search

Many famous results including HELD/Karp-algorithm[4], the polynomial-time factor-1.5 approximation algorithm for METRIC TSP of Christofides [5] and other heuristic algorithm for TSP followed the paradigm of local search: By searching within its local neighborhood incrementally try to improve a solution by a distance measure [6]. Local Search algorithms can effectively use the characteristics of problems to find local optimal solution; however, with the expansion of the scale of TSPs, it requires much more time [7].

4 Algorithms

4.1 Branch and Bound

4.1.1 Description

The Branch and Bound algorithm uses lower bounds to skip some obvious non-optimal solutions for problem reduction. First, we use initial node to find the initialized lower bound. When calculating the lower bound, we will check if the current position has been visited by the current path. If not, we use 2 shortest edges to calculate the lower bound. Even though the performance is worse than the MST, if given more time, the algorithm will still perform the best results. The bound will be used to determine when to exclude unnecessary solutions in the algorithm. If the lower bound of the current path is larger than current bound, we won't expand the problem. Otherwise, we will find the next path with lower cost and next small lower bound. Repeat this idea for each branch until the best solution is found or the time limit is exceeded.

4.1.2 Data Structures

1. Variables to save the static and dynamic factors of the program. We use different variables to save the factors like timestamp, cutoff time, cost and so on
2. Lists to save the distance solution path and output trace. We use a 2-D list to save the distance, 2-D lists to save the best path and trace
3. Dictionaries to store the information of each node like cost and path. When in the loop, we will find cost and path information in these dictionaries
4. A point class to represent the position in each city in the graph.

4.1.3 Pseudo Code

:

Algorithm 1 Branch and Bound

calculateDistance(*city*)
Use start position 0 as the initialize input
List ← save the a node list
while not path stack is not empty **do**
  if out of time, return
  current ← pop last element in the list
  if (cost of current > minimum):
  skip the branch
  if (find leaf node)
  newcost ← current cost + new edge value
  then if (new cost < minimum):
  minimum ← new cost
  solution ← current path
  if (not find leaf node):
  find new least cost of excluded node
  then if (find next node having new least cost):
  nextnode ← create new node
  newbound ← findbound(next node
  if (new bound < current mini cost):
  add next node to the list
end while

4.1.4 Complexity Analysis

Time Complexity. As for BnB algorithm, the worst case is to explore every possible solution. So the time complexity is $O(n!)$.
Space Complexity. In the algorithm, path, cost and visited state of each node in the graph are stored into dictionaries. A N*N list is used to store the distance between each node and other nodes.Therefore, the space complexity of the algorithm is $O(n^2)$.

4.1.5 Strengths and Weakness

Strengths. For small dataset, both the accuracy and efficiency is good. Since we explore every solution to find the best one, if given enough time, the algorithm will finally find the best solution and the minimum cost.
Weakness. For larger datasets, the algorithm is extremely slow. Since the time complexity is so bad, it will take so much long time to find the best solution even though it can do that. Therefore, although it is fast and accurate for small dataset, the total performance of large-scale data is bad if given a small time limitation.

4.2 Approximation Heuristic - Nearest Neighbor

4.2.1 Description

In our approximation algorithm, we chose the nearest neighbor method. We construct the path as follows:
1. Start with any node.
2. Find the node that has the smallest distance to the previous node and it is not in the path yet. Add it to the path.
3. Continue until all the nodes have been added to the path.
4. Set each node as the start node, and find the shortest path.

4.2.2 Pseudo Code

Algorithm 2 Approximation Heuristic - Nearest Neighbor

```
start ← s
v ← s
path ← [v]
while not all vertices visited do
    visited ← closest neighbor w of v
    path.append(w)
    v = w
end while
```

4.2.3 Approximation Guarantee

In *An analysis of several heuristics for the traveling salesman problem*, it mentioned the approximation ratio for this algorithm.
For a traveling salesman graph with n nodes [3]

$$\frac{NEARNEIBER}{OPTIMAL} \leq \frac{1}{2}\lceil lg(n) \rceil + \frac{1}{2}$$

4.2.4 Complexity Analysis

Time Complexity. In this algorithm, for an arbitrary starting node, we go through all the neighbor nodes and calculate the distance between it and the start node. We find the closest node and add it to the path. We continue doing it (n-1) times until this path includes every node once where n is the total number of nodes. We find this path with every node being start node, which will run this algorithm n times. Therefore, the time complexity for this algorithm is $O(n^3)$.
Space Complexity. In this algorithm, we use a set of n space to keep track of nodes we visited, and a list of n space to store the path. Therefore, the space complexity is $O(n)$.

4.2.5 Strengths and Weakness

Strengths. Most of cases, this algorithm can run quickly with comparatively low relative errors.
Weakness. For larger datasets, going through each node as a start node will take extremely large amount of time. Although it may be fast and accurate for a size of data like what we use this time, it has its limits.

4.3 Local Search - Hill Climbing

4.3.1 Description

Hill Climbing belongs to the local search family and is useful for finding a local optimal solution. It starts with an initial solution as the current solution, which can be generated randomly or by following a certain strategy. To get the next solution, we generate the neighborhood of the current solution by making an incremental change, and choose a neighbor as the updated current solution. We repeat the steps of generating neighborhood and choosing the optimal neighbor until certain limitations are met. Throughout this process, we should keep track of the best solution and cost.

4.3.2 Procedure

Find Initial Solution. Randomly choose a solution as the starting point for our local search. We can also generate the initial solution following heuristic rules. Take the initial solution as the current solution.
Generate Neighborhood. Define the changing step from one solution to its neighbor. Apply all possible steps to the current solution and get a set of solutions as the neighborhood.
Choose A Neighbor. Choose a solution from the neighborhood of the current solution (like the one with the largest improvement compared to the current solution), and update the current solution to be the chosen neighbor.
Stopping Strategy. Iteratively generate

neighborhood and choose the neighbor until we reached certain pre-defined conditions (like maximum number of iterations, maximum execution time, etc.). We can restart the local search from another initial solution, or directly return the current best solution.

4.3.3 TSP Implementation

Find Initial Solution. Choose the first location as the start of the travel. Among all unvisited locations, pick the one that's nearest to the current location as the next location. Randomly shuffle certain slices of the generated solution.
Generate Neighborhood. Use 2-opt exchange technique to generate the neighborhood: get each neighbor by swapping the order of each pair of locations. Skip visited solutions.
Choose A Neighbor. Choose the neighbor with the largest improvement on the cost compared to the current solution.
Stopping Strategy. If the neighborhood of the current solution is empty, or the total execution time has reached a threshold, stop searching.
Iterative Restart. If we encounter an empty neighbor but we haven't used up the given execution time, randomly shuffle the current solution as the new initial solution, and restart the local search.

4.3.4 Pseudo Code

See Algorithm 3.

4.3.5 Complexity Analysis

Time Complexity. Let n denote the number of locations. Since we have a map to keep track of the distance between given pair of locations, the time complexity of calculating the cost of a solution should be $O(n)$.
To find the initial solution, we choose the first location as the start of the travel. In order to get the next location, we need to calculate the distance of each unvisited location with the current location, and pick the nearest one. Thus, the time complexity is $O(n^2)$. As for the random shuffle part, we use the Fisher-Yates shuffle algorithm which is $O(n)$.
To generate the neighborhood, we swap each pair of locations in the current solution. The total number of pairs is $\frac{n(n-1)}{2}$. And for each pair, we need $O(n)$ time to add up the total cost. Thus, the time complexity is $O(n^3)$.
Overall, let m denote the number of iterations within the execution time limit. One possible bound is $O(n^2) + O(n) + m * O(n^2) = O(mn^2)$. Since we have a visited set which keeps us from revisiting identical solutions, the other bound is

Algorithm 3 Local Search - Hill Climbing

```
1:  function getInitialSolution (locations)
2:      s ← [locations[0]]
3:      visited ← set(locations[0])
4:      while s.length < locations.length do
5:          next ← the nearest unvisited location
6:          s.add(next)
7:          visited.add(next)
8:      end while
9:      n ← s.length
10:     Randomly shuffle s[0.2n : 0.3n], s[0.5n : 0.6n], s[0.8n : 0.9n]
11:     return s
12: end function
13: function hillClimb (visited, curSolution, bestSolution)
14:     largestImprove ← 0
15:     chosenNeighbor ← empty list
16:     for each pair (i, j) do
17:         neighbor ← swap curSolution[i] and curSolution[j]
18:         if neighbor in visited then
19:             continue
20:         end if
21:         if cost(curSolution) - cost(neighbor) > largestImprove then
22:             largestImprove ← cost(curSolution) − cost(neighbor)
23:             chosenNeighbor ← neighbor
24:         end if
25:     end for
26:     if chosenNeighbor is empty then
27:         return True
28:     end if
29:     visited.add(chosenNeighbor)
30:     curSolution ← chosenNeighbor
31:     if cost(curSolution) < cost(bestSolution) then
32:         bestSolution ← curSolution
33:     end if
34:     return False
35: end function
36: function main (locations)
37:     curSolution ← getInitialSolution(locations)
38:     bestSolution ← curSolution
39:     visited ← empty set
40:     while execution time is within limit do
41:         noNeighbor ← hillClimb(visited, curSolution, bestSolution)
42:         if noNeighbor is True then
43:             randomly shuffle curSolution
44:         end if
45:     end while
46: end function
```

$O(n * n!)$ (we need $O(n)$ time to calculate the total cost for each solution).
Space Complexity. We used a map memorizing the distance between each pair of locations, which takes up $O(n^2)$ space. And the visited set containing all visited solutions is of $O(l * n)$ space, where l is the number of solutions (each of them is of length n). Thus, the total space complexity is $O(n * l)$, which is $O(n * n!)$ in the worst case.

4.3.6 Design And Motivation

Generating Initial Solution. We applied nearest neighbor strategy to get the first version of the initial solution, so that we have a starting point with reduced cost compared to a completely random one. Then we randomly shuffle 3 slices of the solution to handle scenarios where the solution generated by nearest neighbor strategy lies near a local but not global optima.
Iterative Restart. When we still have time left and the neighborhood of the current solution is empty, we randomly shuffle the current solution and take it as the new initial solution for an iterative hill climbing. In this way, we increase the possibility to find the genuine global optima.
Generating Neighborhood. We used 2-opt technique to get the neighborhood mostly for simplicity. Besides, 2-opt generates a reasonable number of neighbors ($O(n^2)$).

4.4 Local Search - Simulated Annealing

The simulated annealing algorithm is derived from the solid annealing principle, where a solid is heated to a sufficiently high temperature and then allowed to cool at a sufficiently slow rate to release the internal residual stress with the movement of atomic or lattice vacancies, eliminating the dislocation in the material by the process of reorganization of these atomic arrangements. When heating, the internal particles of the solid become disordered as the temperature rises and the internal energy increases, while when cooling slowly the particles tend to be ordered and reach an equilibrium state at each temperature, eventually reaching a ground state at room temperature with a minimum internal energy reduction according to the laws of physics.

4.4.1 Procedure

Set initial temperature Initialize the temperature T_0, so that the current temperature $T = T_0$, any initial solution
Create random initial solution A new solution S_2 is generated after a random perturbation of the current solution S_1
Calculate increments Calculate the increment of S_2, $df = f(S_2) - f(S_1)$
Determine solution If $d_f < 0$, accept S_2 as the new current solution; otherwise calculate the acceptance probability $\exp -d_f/T$ of S_2, then generate a random number rand uniformly distributed on the (0,1) interval, if $\exp -\frac{d_f}{T} > rand$, also accept S_2 as the new current solution $S_1 = S_2$ (i.e., accept the bad solution with a certain probability); otherwise keep the current solution S_1
Stopping Strategy If the termination condition stop is satisfied, the current solution S_1 is output as the optimal solution and the program ends; otherwise, return to step(Calculate increments) after decaying T by the decay function

4.4.2 Pseudo Code

See Algorithm 4.

4.4.3 Complexity Analysis

Time Complexity This method needs $O(n^2)$ time to find the initial solution. Besides, the total size of the search space is $O(n!)$ and to calculate the cost of the current solution, we need $O(n)$ time, so the time complexity is $O(n * n!)$.
Space Complexity We used the map to memorize the distance between each pair of locations, which takes $O(n^2)$ space. And we also use a set to memorize all visited solutions. So the Space Complexity is $O(n * n!)$.

4.4.4 Strengths And Weakness

Strength During the execution of the simulated annealing algorithm, the probability of the algorithm returning some overall optimal solution increases monotonically and the probability of returning some non-optimal solution decreases monotonically as the temperature parameter decreases; and with a sufficient number of perturbations and iterations, the simulated annealing algorithm converges asymptotically in polynomial time to an approximately optimal set of solutions.
Weakness To find the global optimum with probability 1, the simulated annealing algorithm needs to satisfy in the cooling schedule: a sufficiently high initial temperature, a sufficiently slow cooling rate, a sufficiently low termination temperature, and sufficient perturbations at each temperature, which makes the convergence rate usually slow.

Algorithm 4 Local Search - Simulated Annealing

```
function getInitialSolution (locations)
    s ← [locations[0]]
    visited ← set(locations[0])
    while s.length < locations.length do
        next ← the nearest unvisited location
        s.add(next)
        visited.add(next)
    end while
    n ← s.length
    Randomly shuffle s[0.2n : 0.3n], s[0.5n : 0.6n], s[0.8n : 0.9n]
    return s
end function
function simulatedAnnealing (visited, curSolution, bestSolution, probability)
    probability ← probability * 0.95
    cityNum ← curSolution.length
    chosenNeighbor ← empty list
    chosenNeighborCost ← 0
    for i in range(0,int(cityNum * (cityNum - 1) / 2)) do
        cityPair ← two random cities in the CityNum
        neighborSolution ← curSolution
        swap curSolution of two cities in cityPair
        if neighbor in visited then
            continue
        end if
        chosenNeighbor ← neighborSolution
        neighborSolution ← curSolution
    end for
    if chosenNeighbor is empty then
        return True
    end if
    if chosenNeighbor < curCost or random<= probability then
        visited.add(chosenNeighbor)
        curSolution ← chosenNeighbor
        curCost ← chosenNeighborCost
    end if
    if cost(curSolution) < cost(bestSolution) then
        bestSolution ← curSolution
    end if
    return False
end function
function main (locations)
    curSolution ← getInitialSolution(locations)
    bestSolution ← curSolution
    visited ← empty set
    while execution time is within limit do
        noNeighbor ← hillClimb(visited, curSolution, bestSolution)
        if noNeighbor is True then
            randomly shuffle curSolution
        end if
    end while
end function
```

5 Empirical Evaluation

5.1 Platform

RAM: 16GB
CPU: Apple M1 with 8 cores
Programming Language: Python 3.8.9

5.2 Experimental Procedure

The program runs with the given data sets, [city].tsp files, that each includes multiple positions in the city. We have four different algorithms running on each data file, and record the best shortest path and the time taken to find it. For local search algorithms, we tested a variety of random seeds from 1 to 10. Then, we visualized the results using following tables and plots so the quality, relative error and time can be viewed easily.

5.3 Evaluation Criteria

5.3.1 Comprehensive Tables

A comprehensive performance table with columns will be used for each algorithms. In each table, instances, time report, algorithms solution quality and relative error will provided for further analysis. Relative error is a matrix that is computed as (Alg − OPT)/OPT to analyze the accuracy.For local search algorithms, results will be the average of least 10 numbers of runs with different random seeds and the result should be average time (seconds) and average solution quality.

5.3.2 Evaluation Plots

Evaluation plots are only for local search algorithms. Two problem instances will be chosen. For each instance and each local search algorithm, three plots will be presented. (a) Qualified Runtime for various solution qualities (QRTDs). The x-axis is the runtime and y-axis is fraction of algorithm runs that have solved the problem with relative error no greater than a certain value. (b) Solution Quality Distributions for various run-times (SQDs). The x-axis is the relative error and y-axis is fraction of algorithm runs that reached the corresponding x within a given cut-off time. (c) Box plots for running times. This plot will display the distribution of running times to reach a relative error no greater than a given bound. In the figure, X coordinate is running time in seconds, and Y coordinate is relative error.

5.4 Results

5.4.1 Tables

City	Time	Quality	Relative Error
Atlanta	501.02	3212345	0.603
Berlin	93.37	19721	1.615
Boston	116.23	2189770	1.451
Champaign	356.6	211417	3.016
Cincinnati	0.05	277952	0.000
Denver	554.13	552187	4.498
NYC	174.72	7439017	3.784
Philadelphia	328.17	3232665	1.316
Roanoke	523.51	6893475	9.517
San Francisco	328.88	5543940	5.843
Toronto	318.42	9512153	7.088
UKansasState	0.43	62962	0.000
UMissouri	588.72	643037	3.845

Table 1: Results for Branch and Bound

City	Time	Quality	Relative Error
Atlanta	0.00	2039906	0.018
Berlin	0.03	8181	0.085
Boston	0.04	1029012	0.152
Champaign	0.06	61828	0.174
Cincinnati	0.01	301260	0.084
Denver	0.4	117617	0.171
NYC	0.09	1796650	0.155
Philadelphia	0.00	1611714	0.155
Roanoke	10.53	773359	0.180
San Francisco	0.38	857727	0.059
Toronto	0.57	1243370	0.057
UKansasState	0.00	69987	0.112
UMissouri	0.97	155307	0.170

Table 2: Results for Nearest Neighbor

City	Time	Quality	Relative Error
Atlanta	430.51	2059689	0.028
Berlin	8.74	8301	0.101
Boston	7.49	943050	0.055
Champaign	77.44	59181	0.124
Cincinnati	0.05	277953	0.000
Denver	3.38	127556	0.27
NYC	540.47	1785749	0.148
Philadelphia	0.07	1529223	0.095
Roanoke	481.83	975741	0.489
San Francisco	10.75	938852	0.159
Toronto	36.19	1479809	0.258
UKansasState	0.11	62962	0.00
UMissouri	386.5	150904	0.137

Table 3: Results for Local Search-Hill Climbing (random seed: 1)

City	Time	Quality	Relative Error
Atlanta	0.02	2080515	0.038
Berlin	1.17	8942	0.19
Boston	0.66	972967	0.089
Champaign	0.67	63541	0.207
Cincinnati	0.05	277953	0.000
Denver	2.48	120711	0.202
NYC	1.01	1983872	0.276
Philadelphia	0.32	1591629	0.14
Roanoke	22.56	1121039	0.71
San Francisco	2.96	1043617	0.288
Toronto	6.28	1805695	0.535
UKansasState	0.0	62962	0.00
UMissouri	4.79	163893	0.235

Table 4: Results for Local Search-Simulated Annealing (random seed: 1)

5.4.2 Qualified Runtime for various solution qualities (QRTDs)

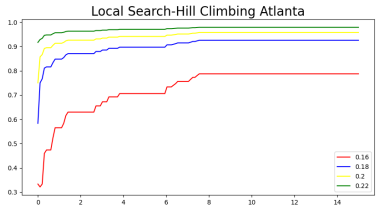

Figure 1: QRTD. Algorithm: Local Search (Hill Climbing). Instance: Atlanta.

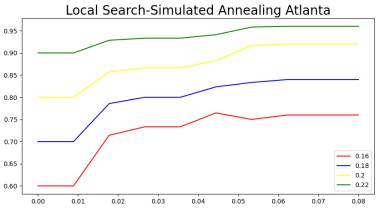

Figure 2: QRTD. Algorithm: Local Search (Simulated Annealing). Instance: Atlanta.

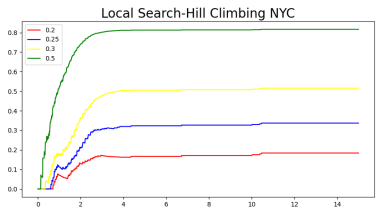

Figure 3: QRTD. Algorithm: Local Search (Hill Climbing). Instance: NYC.

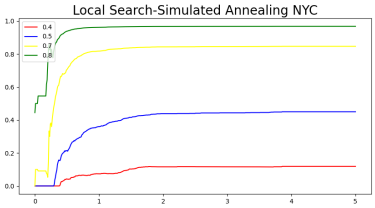

Figure 4: QRTD. Algorithm: Local Search (Simulated Annealing). Instance: NYC.

5.4.3 Solution Quality Distributions for various run-times (SQDs)

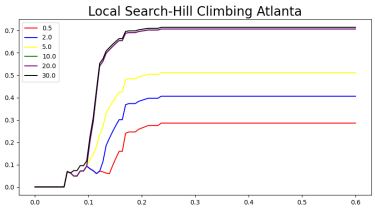

Figure 5: SQD. Algorithm: Local Search (Hill Climbing). Instance: Atlanta.

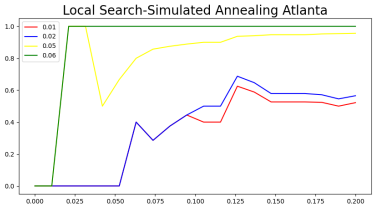

Figure 6: SQD. Algorithm: Local Search (Simulated Annealing). Instance: Atlanta.

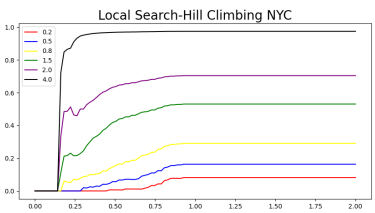

Figure 7: SQD. Algorithm: Local Search (Hill Climbing). Instance: NYC.

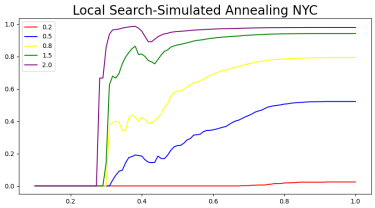

Figure 8: SQD. Algorithm: Local Search (Simulated Annealing). Instance: NYC.

5.4.4 Box Plots for run-times

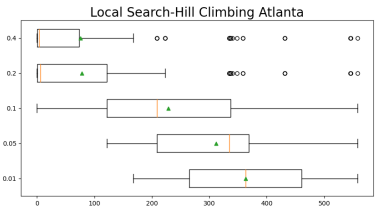

Figure 9: Box plot for run-times. Algorithm: Local Search (Hill Climbing). Instance: Atlanta.

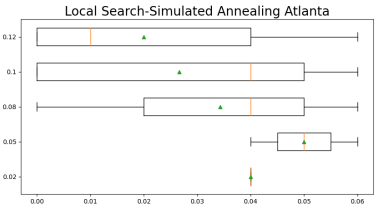

Figure 10: Box plot for run-times. Algorithm: Local Search (Simulated Annealing). Instance: Atlanta.

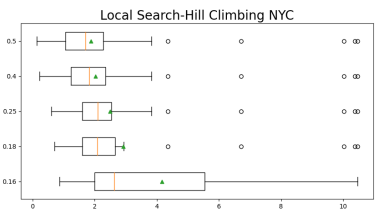

Figure 11: Box plot for run-times. Algorithm: Local Search (Hill Climbing). Instance: NYC.

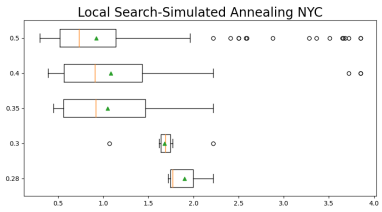

Figure 12: Box plot for run-times. Algorithm: Local Search (Simulated Annealing). Instance: NYC.

5.5 Analysis

Nearest Neighbor The lower bound for this algorithm is mentioned in **4.2.3**. By calculating with data in each file, the lower bounds of $\frac{NEARNEIBER}{OPTIMAL}$ all fall into [1.5, 2], and $\frac{NEARNEIBER}{OPTIMAL}$ from our results are in range [1.01, 1.18]. Besides, from the Approximation Ratio formula, it shows the bound is highly depending on the number of nodes we have, which we also reflect in our result table in city Roanoke. Roanoke has a significant larger number of nodes comparing with other cities, and it takes longer time with relative high error.

Branch and Bound The method to calculate lower bound is 2 shortest edges. The performance is better in small dataset such as Cincinnati and UKansasState. For large dataset, the runtime is much slower. Although the error is too large in the large dataset in the given 600 cutoff time, since the result of small set is correct, we believe if given much more time, the algorithm will finally find the best solution.

Local Search(Hill Climbing) Hill-climbing performs poorly when the amount of data is large, requiring a lot of time to complete the calculation, and the error is not reduced, but performs well when the amount of data is small, such as Cincinnati, UKansasState

Local Search(Simulated Annealing) Simulated Annealing outperforms hill climbing in all aspects, and performs very well when the data volume is relatively small, but when the data volume increases, it is obvious that the error is increasing, and the average time is second only to Branch and Bound

6 Discussion

Branch and Bound method takes the longer time and has more error. In the cases where the number of vertices is large, the time required for calculation will be long, and as the amount of data increases, the situation to be considered becomes very complex. The Branch and Bound method cannot raise the lower bound quickly, so it takes more time.
For local search, the time required for calculation fluctuates due to randomness, and edge cases are more likely to be covered since a random shuffle happens. If there is enough time to find the correct answer, Hill-climbing method is worse than the Simulated Annealing algorithm in terms of calculation time and error convenience among the two local search methods.
In terms of efficiency, Approximation Heuristic-Nearest Neighbor takes less time to compute, but there may be some cases where the correct answer is never found.
Therefore, we believe that Approximation Heuristic-Nearest Neighbor is the best algorithm in terms of running time, accuracy and error for this specific data set.

7 Conclusions

We analyzed a total of four algorithms to solve the Traveling Salesman Problem, including Branch and Bound, Approximation Heuristic-Nearest Neighbor, Local Search - Hill Climbing, and Local Search - Simulated Annealing. The Traveling Salesman Problem belongs to the so-called NP-complete problem, and the exact solution of the TSP can only be achieved by exhausting all combinations of paths, whose time complexity is O(N!). Therefore, we provide algorithms to optimize these problems. Through analysis we learned that different algorithms have advantages and disadvantages, although approximation method has the best comprehensive results in comparison, when we need to pursue accuracy, perhaps the Approximation method is no longer the best choice. In the future, we can try to optimize different multi-core algorithms instead of just running on one processor, which may guarantee the accuracy and reduce the time to get a more optimized algorithm.

References

[1] *The Traveling Salesman Problem and Its Variations*, edited by G. Gutin, and A. P. Punnen, Springer, 2002. ProQuest Ebook Central.

[2] Grymin R., Jagiełło S. (2016) *Fast Branch and Bound Algorithm for the Travelling Salesman Problem*. In: Saeed K., Homenda W. (eds) Computer Information Systems and Industrial Management. CISIM 2016. Lecture Notes in Computer Science, vol 9842. Springer, Cham.

[3] Daniel J Rosenkrantz, Richard E Stearns, Lewis, II, and Philip M. *An analysis of several heuristics for the traveling salesman problem*. SIAM J. Comput., 6(3):563–581, September 1977.

[4] Held, M., Karp, R.M.: *A dynamic programming approach to sequencing problems*. J. Soc. Ind. Appl. Math. 10(1), 196–210 (1962)

[5] Arora, S.: *Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems*. J. ACM 45(5), 753–782 (1998)

[6] Guo, Jiong, et al. *The Parameterized Complexity of Local Search for TSP, More Refined*. Algorithmica, vol. 67, no. 1, Springer US, 2012, pp. 89–110, doi:10.1007/s00453-012-9685-8.

[7] Dong, Ruyi, et al. *Hybrid Optimization Algorithm Based on Wolf Pack Search and Local Search for Solving Traveling Salesman Problem*. Shanghai Jiao Tong Da Xue Xue Bao, vol. 24, no. 1, Shanghai Jiaotong University Press, 2019, pp. 41–47, doi:10.1007/s12204-019-2039-9.