Travelling Salesman Problem

Jingdao Chen
jchen490@gatech.edu

Hung-Yi Li
hli603@gatech.edu

Leon C. Price
lprice8@gatech.edu

Shiqin Zeng
zsq123@gatech.edu

ABSTRACT

The travelling salesman problem is an important optimization problem that has been widely studied due to its numerous applications in logistics, robotic navigation, and data routing. In its general form, the travelling salesman problem deals with finding the minimum cost tour that passes through each node of a graph once. This report provides a comparative evaluation of four different algorithms to solve the travelling salesman problem. These four include the branch and bound algorithm, the nearest neighbor approximation algorithm, and two local search algorithms which are 2-opt and simulated annealing. The four algorithms are compared in terms of their relative error with respect to the optimal solution in a test dataset of 13 different cities. Additionally, the two local search algorithms were evaluated according to their Qualified Runtime Distribution (QRTD) and Solution Quality Distribution (SQD) plots.

Author Keywords

Algorithms, Optimization, Branch-and-bound, Local Search, Approximation

1. INTRODUCTION

The travelling salesman problem was mathematically formulated in the 1800s by the Irish mathematician W.R. Hamilton [1] and by the British mathematician Thomas Kirkman. It has multiple applications in logistics, robotic navigation, and data routing. The optimization form of the travelling salesman problem is known to NP-hard, however there are multiple approximation and local search approaches that have been shown to work well in practice.

This study aims to compare the performance of four different algorithms in solving the travelling salesman problem. A dataset consisting of 13 different cities, each having around 10 - 230 nodes, is used to quantitatively evaluate the algorithm performance. The following sections will describe the problem definition, the design of each algorithm, and the empirical results obtained.

2. PROBLEM DEFINITION

Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight. Here, a Hamiltonian cycle refers to a cycle that visits each vertex of the graph exactly once. For example, Figure 1 (adapted from [5]) shows the solution of an instance of the travelling salesman problem.

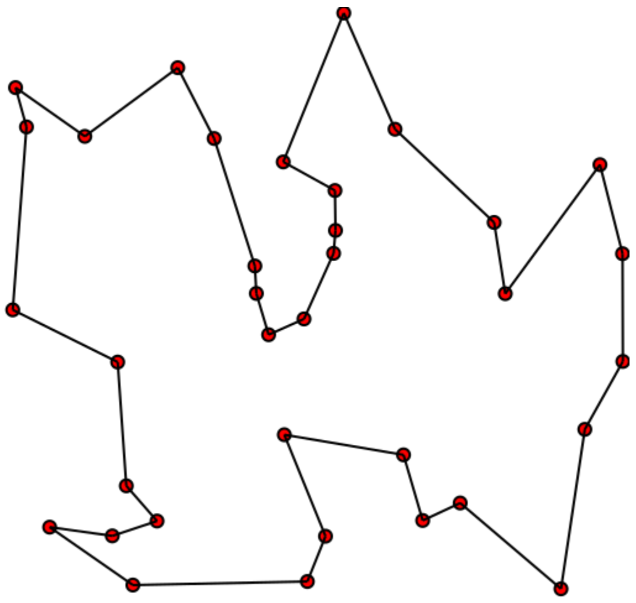


Figure 1: Example solution for an instance of the travelling salesman problem

3. RELATED WORK

There are multiple approaches in the literature to solve the travelling salesman problem. The first group are exact approaches which aim to find the provably minimum tour. These include the Held-Karp dynamic programming algorithm [7] and branch-and-cut algorithm by Applegate et al. [2]. However, the computation time of these approaches scale exponentially with respect to the problem size, and are not practical for large problem instances. Another group of algorithms use greedy approximation approaches to quickly find a solution even though it may not be optimal. Some examples include the nearest neighbor heuristic [10] and the minimum spanning tree approximation [4], which achieves a tour at most 2 times the optimal (i.e. a 2-approximation). Finally, many algorithms also use the local search approach, which is to initially generate a candidate solution and then improve it iteratively through a series of small perturbations. These local search algorithms include the k-opt method [3], and simulated annealing [9].

4. ALGORITHMS

4.1 Branch and Bound

4.1.1 Subproblem

The subproblem of branch and bound is to determine the traveling order for the remaining nodes to minimize the traveling costs of visiting all the remaining nodes and going back to the starting node given the order of previously chosen nodes. In the initial stage, no node is selected. Any node can be picked as the starting node since in the end all nodes are connected as a cycle. Assume there are n nodes. After the first node 0 is chosen, there are (n – 1) node options for the next hop, (n - 2) for the hop next to the next hop, and so on until the last node. Then the final node should link back to the first node. The size of the solution space is (n – 1)!.

4.1.2 Lower bound computation

For the lower bound computation, three kinds of lower bound functions are adopted. First, the sum of the minimum cost of leaving every node. Second, the sum of the two shortest adjacent edges (incoming and outgoing) divided by 2 for a symmetric graph. Third, the sum of the smallest value for each row and column of the cost matrix, which is similar to the process of reduced matrix.

4.1.3 How to choose a subproblem to expand?

Each iteration, the deepest branch with the lowest lower bound is processed to find the next hop with the lowest cost. The branch with lower bound updated after taking into account the next hop will be put back into the candidate pool again to be considered for the next iteration with the other branches. Every time a branch reaches to a leaf node, which means the length of nodes along the branch is equal to the number of all nodes, the upper bound could be updated if the overall cost of this branch is the minimum among all completed branches. Then the branches with the lower bound higher than this upper bound can be pruned. Also, the branch with the updated cost higher than the upper bound will not be put back into the candidate pool for consideration afterwards.

4.1.4 How to expand a subproblem?

As the deepest branch with lowest lower bound is picked, the costs will be computed for traveling next to each of the remaining nodes. The way of expanding a subproblem depends on the choice of lower bound function. For the first lower bound function – the sum of the minimum cost of leaving every node, the updated lower bound will be the original lower bound plus the true cost of traveling between the current node and the chosen next node minus the minimum cost of leaving current node. For the second lower bound function – the sum of the two shortest adjacent edges divided by 2, the updated lower bound will be the original lower bound plus the true cost of traveling between the current node and the chosen next node minus the sum of the second shortest edge of the current node and the shortest edge of the next node divided by 2. For the third lower bound function – the sum of the smallest value for each row and column of the cost matrix, the updated lower bound will be the original lower bound plus the true distance of traveling between the current node and the chosen next node plus the reduced costs of the current matrix.

4.1.5 Pseudocode

```
Input locations with x, y
For node in nodes:
      For node in nodes:
        Compute distance and put in matrix
Calculate lower bound with reduced matrix
or the sum of shortest edges
Initiate min heap with node 0
While min heap:
      Take current node from the top of min heap
      For next node from current node:
        Calculate updated lower bound
    Put the branch back into min heap
        If reaching the leaf -> new solution is found:
            Update the solution
            Prune the solution tree by removing
            branches whose current lower bound
            larger than the updated solution
            from min heap
Return solution
```

4.1.6 Time and space complexity

The worst time complexity is $O(n^3 * n!)$. For every implementation, there must be $O(n^2)$ for the initial pair distance calculation. For reduced matrix cost, matrix reduction takes $O(n^2)$ each. In total, there are (n – 1)! branches. For each branch, there are n steps to the leaf node. So it takes $O(n^2 * n!)$ in total.

For minimum edge cost or 2 shortest adjacent edges, it takes $O(1)$ to calculate the updated lower bound. In total, there are (n – 1)! branches. For each branch, there are n steps to the leaf node. There are n possible options for the next hop. So it takes $O(n * n!)$ in total.

The worst space complexity is bounded by $O(n^2 * n!)$ for a distance matrix in each branch. For minimum edge cost or 2 shortest adjacent edges, the space complexity is $O(n * n!)$ for the traveling order in each branch.

4.1.7 Strengths and weaknesses

The strength of branch and bound algorithm is that it examines over all the possible solutions with pruning, so it is more efficient than pure brute-force. The weakness is that the whole solution space is still huge if there is a large number of nodes. But if time and memory allow, branch-and-bound would finally generate the optimal solution.

In our implementation, we found 2 shortest adjacent edges divided by 2 outperforms the other two lower bound functions in most cases. The reason could be that the calculation of updated lower bound of 2 shortest adjacent edges is faster than reduced matrix cost and that 2 shortest adjacent edges divided by 2 better presents the lower bound than minimum cost of leaving each node does. Therefore, adopting 2 shortest adjacent edges allows us to explore more branches than adopting reduced matrix cost and to prioritize more promising branches than adopting minimum cost of leaving each node.

To expedite the process to reach at least one leaf, we adopt depth-first-search instead of breadth-first-search. That is, the min heap sort the deepest branch first and then the lowest lower bound, so that the deepest branch will be processed earlier.

4.2 Local Search 1: 2-opt

4.2.1 Algorithm design

The *2-opt* algorithm is a local search algorithm that attempts to iteratively improve the solution by swapping two edges of the candidate tour in a way that reduces the tour length. Starting from a randomly initialized tour, the 2-opt algorithm explores all possible combinations of swapping two edges. Each time a swap is found that reduces the tour length, the tour is then updated. This process is repeated until no further improvement can be made. Since this algorithm only performs local search, it tends to get stuck in local minima. Thus, the algorithm is repeated multiple times with different random starting tours and the result that has the shortest tour is returned (random restarts).

4.2.2 *Pseudocode*

```
#helper function that performs pairwise exchange
def two_opt(pairwise_dist, path,i,j):
    if pairwise_dist[path[i-1]][path[j]] +
        pairwise_dist[path[i]][path[j+1]] <
        pairwise_dist[path[i-1]][path[i]] +
        pairwise_dist[path[j]][path[j+1]]:
        return path[:i]+
            path[i:j+1][::-1]+path[j+1:]
    else:
        return None

#loop through all pairs until a
#valid pairwise exchange is found
while updated:
    updated = False
    for i in range(1,len(path)-2):
        for j in range(i+1,len(path)-1):
            new_path = two_opt(pairwise_dist,path,i,j)
            if new_path is not None:
                path = new_path
                updated = True
```

4.2.3 *Time and space complexity*

The time complexity of performing a 2-opt exchange is $O(n)$ since it involves reversing a subset of the tour, which has length $O(n)$. However, the 2-opt exchange is performed multiple times until the algorithm converges and the number of these exchanges that need to be performed depends on both the problem and the starting tour. The space complexity is $O(n)$ since the solution is stored as a array of n ordered nodes which can be modified in-place.

4.2.4 *Strengths and weaknesses*

The strength of the 2-opt algorithm is that it usually quickly converges since each 2-opt exchange is guaranteed to reduce the tour length and the tour rapidly reaches a state where no further valid exchanges can be performed. The weakness of the 2-opt algorithm is that it easily gets stuck in local optima. The 2-opt algorithm cannot improve on a local optima without restarting even though the global optimum has not been reached yet.

4.3 Local Search 2: Simulated Annealing

Simulated Annealing is a process that emulates the physical process of annealing which is a method used for cooling metals in a way that produces a desired granular structure. Similarly we accept moves that increase the length of the path early in the search algorithm while the temperature is high and then decrease our likelihood to accept bad moves as the time goes on and the temperature cools. This method helps to avoid local minimum early in the local search process. Our algorithm is a modified version of the algorithm outlined in by Russel and Norvig [11].

4.3.1 *Pseudocode*

```
def schedule_exponential(t_start,time_limit):
    #exponential cooling schedule
    t = time.time() - t_start
    alpha = 0.0001
    temp = time_limit*alpha**t
    if temp < 0:
        temp = 0
    return temp

#loop through all pairs testing
#local neighbor swaps at each step
for i in range(1,len(path)-2):
    for j in range(i+1,len(path)-1):
        #calculate the temperature based on
        #the cooling schedule
        T = schedule_exponential(start_time,cut_time)
        new_path = two_opt(pairwise_dist,path,i,j)
        if new_path is not None:
            cost_new = compute_tour(pairwise_dist,new_path)
            delta_E = cost_new - cost_current
            if delta_E < 0:
                path = new_path
            else:
                a = np.random.rand()
                if a < np.exp(delta_E/T):
                    path = new_path
```

4.3.2 *Time and space complexity*

The time complexity for simulated annealing is variable and depends heavily upon the topography of the problem search space and the move to a new neighbor. For this implementation, the move to a new neighbor was simply done by swapping the order of two nodes which is done in running time $O(1)$. To select which neighbors to swap, we iterate over the current path which takes $O(n)$ running time. However, the convergence condition is dependent upon reaching an optima where the temperature is cool enough that we are no longer accepting 'energy' increasing moves and also there are no more local moves that improve the solution. For this reason, it is not possible to bound the time complexity.

For space complexity, this implementation does not use memory to save the cost of previously calculated paths. Therefore, the space complexity is $O(n)$ as we are only ever storing the length of the current path or or new path and their respective scores.

4.3.3 *Strengths and weaknesses*

Simulated Annealing has some strengths and weakness when compared to the other approaches. Compared to the tree search methods like Branch and bound it generally reaches an acceptable solution more quickly. However, it can be slower compared to other local searches like hill climbing because it spends more time exploring random locations towards the beginning of the search. One advantage that simulated annealing has is its ability to avoid local optima compared to hill climbing algorithms.

In implementation, simulated annealing can be challenging because there is a significant amount of parameter tuning that goes into developing the cooling schedule which is not required for many other algorithms.

4.4 Approximation: Nearest Neighbor

4.4.1 *Algorithm design and Approximation*

The Nearest Neighbor algorithm randomly starts with a city, then it selects the nearest city to the current city. After all the cities have been visited, it goes back to the starting city to finish the tour. In this project, we want to get the minimum cost of a Hamiltonian cycle. Jerome Monnot[10] dealt with this situation and set the variants that the edge costs belong to interval $[a; ta]$, where $a > 0$ and $t > 1$. He proved that the Nearest Neighbor can guarantee a $\frac{2}{t+1}$ approximation for $\min TSP\ [a; ta]$ under the standard performance ratios which are tight for some instances.

4.4.2 *Pseudocode*

```
Input: a two-dimensional array to store
the distance between two cities.
def nearest_neighbor(distance matrix):
    #Initialize the variables
    #Pick a starting point randomly
    start_node = random.(0, num_node - 1)
    distance_to_start_node =
    distance_matrix[:, start_node].copy()
    distance_matrix[:, start_node] = np.inf
    tour = [start_node]
    len_tour = 1
    total_distance = 0

    #Start the tour
    while len_tour < num_node:
        current_node = tour[-1]
        next_node = np.argmin
        (distance_matrix[current_node, :])
        tour.append(next_node)
        len_tour += 1
        total_distance += distance_matrix
        [current_node, next_node]
        update distance_matrix

    #Go back to the starting point to finish the tour
    total_distance += distance_to_start_node[tour[-1]]
    return total_distance, tour
```

4.4.3 *Time and space complexity*

For each iteration, it takes $O(n)$ to search the nearest city, so the time complexity is $O(n^2)$ since there are n cities in total. For the space complexity, it requires a two-dimensional array to store the distance between two cities, and a array to store the tour results. Hence this Nearest-Neighbor algorithm has a space complexity of $O(n^2)$.

4.4.4 *Strengths and weaknesses*

The Nearest-Neighbor algorithm can be implemented easily and yields an effectively short path quickly. The results of this algorithm is acceptable for the Euclidean TSP. For the randomly distributed cities, the relative error of this algorithm is about 25% when compared to the shortest possible path[8]. However, the Nearest-Neighbor algorithm can generate very poor results for the general Symmetric and Asymmetric TSP[6]. Moreover, its relative error is larger compared to other methods in this project.

5. EMPIRICAL EVALUATION

This section shows an empirical evaluation of the performance of each algorithm based on the relative error. Each algorithm runs continuously until the cutoff time is reached, upon which the best solution found is recorded. The cutoff time for branch and bound is 600 seconds whereas the cutoff time for local search is 5 seconds. On the other hand, all runs of the nearest neighbor approximation algorithm finished under 1 second. To provide a fair comparison, the results for all algorithms are obtained on the same computer, an Intel Xeon E3-1200 CPU with 16GB of RAM.

5.1 Branch and Bound

When the number of nodes is small enough (under 20), most lower bound functions work well to find the optimal solutions. The results tightly fit the optimal lower bound. As the number goes up, we found that the best solution under time constraint is far away from the optimal solution when the number of nodes increases.

Table 1: Branch and Bound – Reduced Matrix Cost

Dataset	Time(s)	Sol.Q	Rel.Err
Cincinnati	0.05	277952	0.0000
UKansasState	0.17	62962	0.0000
Atlanta	24.17	2003763	0.0000
Philadelphia	0.05	1773796	0.2706
Boston	85.00	1177460	0.3178
Berlin	25.62	8972	0.1896
Champaign	300.67	62395	0.1852
NYC	335.16	1712976	0.1015
Denver	178.11	125963	0.2542
SanFrancisco	311.37	1018212	0.2567
UMissouri	103.83	175383	0.3216
Toronto	83.44	1526770	0.2981
Roanoke	20.39	894260	0.3643

Table 2: BnB – Min Cost of Leaving Each Node

Dataset	Time(s)	Sol.Q	Rel.Err
Cincinnati	0.00	277952	0.0000
UKansasState	0.01	62962	0.0000
Atlanta	16.94	2003763	0.0000
Philadelphia	378.04	1490426	0.0677
Boston	286.57	1032691	0.1557
Berlin	0.12	8947	0.1863
Champaign	0.01	59299	0.1264
NYC	0.05	1799063	0.1569
Denver	75.18	117713	0.1721
SanFrancisco	235.75	1024069	0.2640
UMissouri	10.10	148734	0.1208
Toronto	406.58	1535038	0.3051
Roanoke	486.77	897439	0.3692

Table 3: BnB– 2 Min Adjacent Edges Divided by 2

Dataset	Time(s)	Sol.Q	Rel.Err
Cincinnati	0.04	277952	0.0000
UKansasState	0.01	62962	0.0000
Atlanta	425.89	2003763	0.0000
Philadelphia	149.58	1461817	0.0472
Boston	268.13	937443	0.0491
Berlin	76.38	8809	0.1680
Champaign	84.56	58523	0.1117
NYC	1.11	1812960	0.1658
Denver	157.23	120921	0.2040
SanFrancisco	207.88	948100	0.1702
UMissouri	588.28	161308	0.2155
Toronto	237.79	1245760	0.0592
Roanoke	0.45	870656	0.3283

An interesting finding is that the relative error is not proportional to the number of nodes. The reasons could be that the hand-picked starting node 0 has a significant influence on the following branch development and that the smaller node is ordered ahead in the min heap when two nodes have the same lower bounds. Some good paths starting with large costs might grow costs slowly in a later stage. Also, a better path could connect to a larger node in an early stage. Thus, if we are lucky enough, an better solution would appear before time-out, but still the algorithm would know the solution is optimal after all branches were examined or pruned.

The comprehensive tables of all lower bound function implementations are shown in Tables 1, 2, and 3 by the order of (1) reduced matrix cost, (2) minimum cost of leaving each node, and finally (3) 2 shortest adjacent edges divided by 2.

5.2 Local Search 1: 2-opt

Table 4 below shows the solution quality and relative error for each problem instance using the 2-opt algorithm (averaged over 10 starting random seeds). In our implementation, random restart is performed whenever the 2-opt loop terminates. Results show that in general, the time taken to achieve the best solution increases with the problem size. Similarly, the relative error also increases with the problem size. Even though 2-opt is a local search method with no optimality guarantees, it is still able to achieve the optimal result for 6 problem instances (after random restarts).

Table 4: 2-opt results

Dataset	Time(s)	Sol.Q	Rel.Err
Cincinnati	0.00	277952	0.0000
UKansasState	0.00	62962	0.0000
Atlanta	0.00	2003763	0.0000
Philadelphia	0.01	1395981	0.0000
Boston	0.99	893536	0.0000
Berlin	2.10	7542	0.0000
Champaign	1.87	52649	0.0001
NYC	2.15	1559052	0.0026
Denver	3.04	101813	0.0138
SanFrancisco	2.31	819718	0.0118
UMissouri	2.48	134752	0.0154
Toronto	2.65	1177646	0.0013
Roanoke	2.19	673216	0.0271

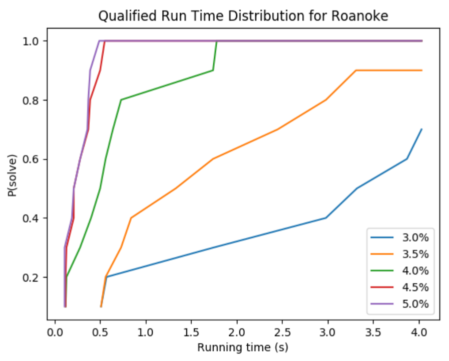
Figure 2 shows the Qualified Running Time Distribution (QRTD) using the 2-opt algorithm. The figure shows that the probability of solving the travelling salesman problem (over 10 different trials) to within x-% of optimal increases over time. Here x is selected to range from 3.0% to 5.0%. When the optimality margin is small (i.e. 3.0%), $P(solve)$ increases more slowly whereas when the optimality margin is large (i.e. 5.0%), $P(solve)$ increases more quickly.

Figure 3 shows the Solution Quality Distribution (SQD) using the 2-opt algorithm. The figure shows that the probability of solving the travelling salesman problem (over 10 different trials) in y seconds increases according to the relative solution quality margin. Here y is selected to range from 0.1s to 5.0s. When the cutoff time is small (i.e. 0.1s), $P(solve)$ increases more slowly whereas when the cutoff time is large (i.e. 5.0s), $P(solve)$ increases more quickly.

Figure 4 shows the box plot of running times to achieve a predefined solution quality (5% and 10% respectively) for the "Roanoke" and "Toronto" problem instances over 10 different runs. For both problem instances, the running time is longer and has wider variance when at 5% solution quality compared to at 10% solution quality. This suggests that most local optima solutions can achieve 10% solution quality. However, getting to 5% solution quality requires getting lucky with random restarts. Thus, there is a wide variance of the running time at 5% solution quality. In addition, the "Roanake" problem takes longer to solve compared to "Toronto" since it contains more nodes.

5.3 Local Search 2: Simulated Annealing

Table 5 below shows the solution quality and relative error for each problem instance using the Simulated Annealing



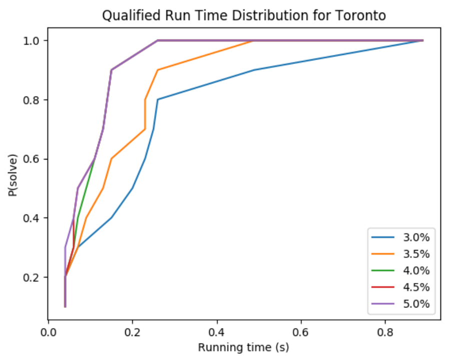
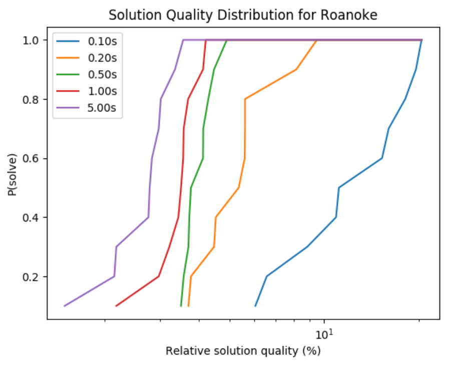


Figure 2: QRTD plots for 2-opt



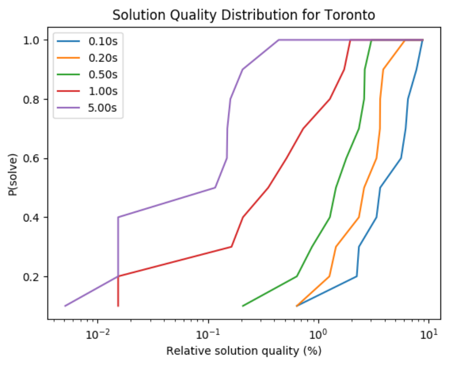


Figure 3: SQD plots for 2-opt

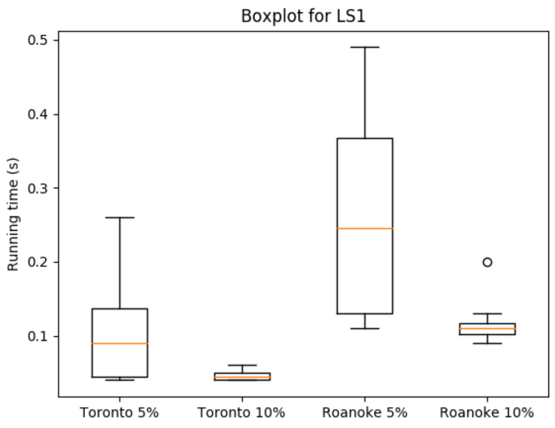


Figure 4: Boxplot for 2-opt results

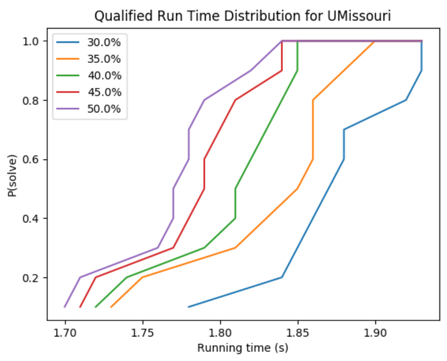
algorithm. In our implementation, we used an exponential cooling schedule with an alpha value of 0.01.To achieve low errors relatively quickly, the cooling schedule must be tuned to perform well across the different instance input sizes. We see that SA performs relatively well for all algorithms staying under 10% relative error until ”Roanoke” which has a significantly larger input size and causes the algorithm to time-out. As you can, see some of the algorithms did not reach the optimal solution even though they returned an answer under the allotted time of 5 seconds. This shows that they fell into a local optima late in the cooling schedule and the ”temperature” was not hot enough to get them out. In the future, random restarts could be added to help mitigate against this.

Table 5: Simulated Annealing results

Dataset	Time(s)	Sol.Q	Rel.Err
Cincinnati	0.49	277952	0.0000
UKansasState	0.43	62962	0.0000
Atlanta	1.36	2006060	0.0011
Philadelphia	1.73	1397957	0.0014
Boston	1.78	900244	0.0075
Berlin	1.88	7862	0.0424
Champaign	1.86	53372	0.0139
NYC	2.07	1607097	0.0335
Denver	2.23	105298	0.0485
SanFrancisco	2.56	887287	0.0952
UMissouri	2.57	140391	0.0579
Toronto	2.83	1258756	0.0702
Roanoke	4.99	811002	0.2373

Figure 5 shows the Qualified Running Time Distribution (QRTD) using the Simulated Annealing algorithm. The top image shows QRTD for UMissouri dataset and the bottom shows the Toronto dataset. The UMissouri QRTD clearly shows a flatter region in the beginning where the algorithm is reaching the goal percentage at a lower rate then it shows a steeper increase as time goes on; this is due to the high temperatures allowing sub-optimal local moves early. As the temperature cools, we accept less sub-optimal moves and thus progress more quickly towards the optimal solution. Therefore, even with small increases in the run time, we see large non-linear jumps in the percentage of runs reaching the desired percent of optimal. The time and cooling temperature work together to accelerate the algorithm towards the optimal solution as time increases.

Figure 6 shows the Solution Quality Distribution (SQD) using the Simulated Annealing algorithm. This plot is consistent with the QRTD graphs; with the decrease in time, we see more and more dramatic degradation in performance for the simulated annealing algorithm as a consequence of the exponential cooling schedule.



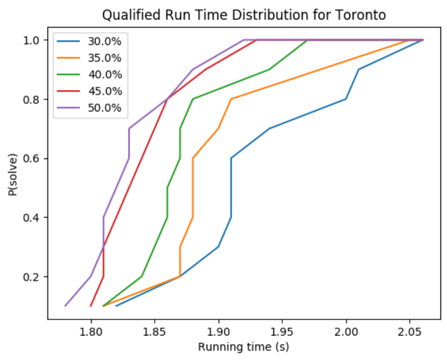
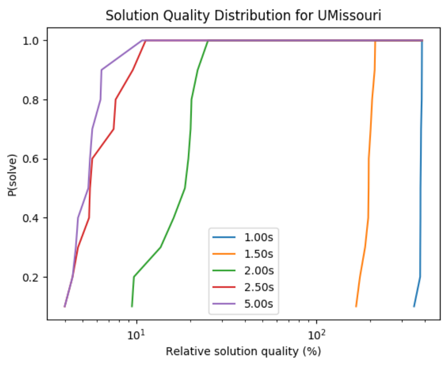


Figure 5: QRTD plots for Simulated Annealing

Figure 7 shows the box plot of running times to achieve a predefined solution quality (5% and 10% respectively) for ”UMissouri” and ”Toronto” problem instances. For both problems, we see that the spread in run time is greater for completion to 10% than for 5% with a slightly lower mean. This spread is likely a result of the random sub-optimal moves that occur early in the algorithm execution which decrease with time. This results in somewhat of a funneling effect; where though it runs slower than the 2-opt algorithm, it tends become more consistent in its run time as the quality solution approaches the global optima.

5.4 Approximation: Nearest Neighbor

Table 5 below shows the solution quality and relative error for each problem instance using the Nearest Neighbor algorithm. The rightmost column also shows the theoretical approximation factor according to [10], which is $\frac{t+1}{2}$.



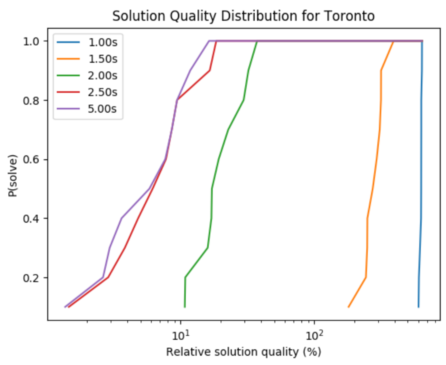


Figure 6: SQD plots for Simulated Annealing

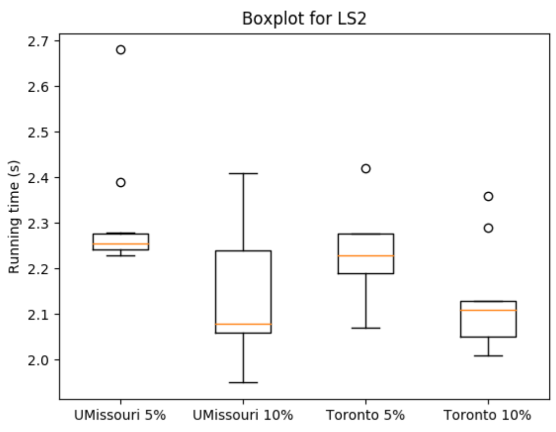


Figure 7: Boxplot for Simulated Annealing results

Table 6: Nearest Neighbor results

Dataset	Time(s)	Sol.Q	Rel.Err	Approx.Bound
Cincinnati	0.00	328215	0.1808	18.68
UKansasState	0.00	70846	0.1252	16.66
Atlanta	0.00	2316928	0.1563	22.83
Philadelphia	0.00	1778425	0.2740	154.04
Boston	0.00	1109454	0.2416	155.64
Berlin	0.01	10072	0.3355	57.70
Champaign	0.01	72183	0.3712	67.08
NYC	0.01	1859959	0.1961	140.74
Denver	0.01	120282	0.1977	44.35
SanFrancisco	0.01	983088	0.2134	2716.39
UMissouri	0.01	172408	0.2991	106.91
Toronto	0.01	1454436	0.2366	2281.16
Roanoke	0.06	857121	0.3077	1120.05

6. CONCLUSIONS

In conclusion, the travelling salesman problem is difficult to solve optimally but can still be approached with approximation or local search methods. Branch and bound algorithm provides the solution that is guaranteed to be optimal if time allows. However, the time to finish running the algorithm can be exponential. Even if the optimal solution is found early, it is until the algorithm completes that we can ensure the optimality. Conversely, local search algorithms are not guaranteed to provide the optimal solution, they quickly identify a feasible solution and gradually improves the solutions from the current best solution. Incorporating some randomness, local search algorithms can escape from the local optima and search for other possible local optima. The results found would not be identical in every experiment. The initial starting point have something to do with the following development of search process. Like local search algorithms, the initial starting point plays a more important role in nearest neighbor algorithm. Once the initial starting point is decided, the result is deterministic. The runtime of nearest neighbor algorithm is fixed. It provides an acceptable solution, though not necessarily the optimal, in a very short time. However, the search process terminates and does not continue optimizing once terminating, so the solution is not very good in most cases. Given the greedy designs, some possible solutions are not considered in nearest neighbor algorithm. Therefore, based on the time and memory constraints in real life, we can consider ensemble these algorithms in order to meet different needs. From our experiments, sometimes with time limit, heuristics could yield better results than the branch and bound algorithm.

7. ACKNOWLEDGMENTS

The authors would like to thank Dr. Xiuwei Zhang for her instruction and guidance in teaching this algorithms class.

8. REFERENCES

[1] Icosian game. `https://www.daviddarling.info/encyclopedia/I/Icosian_Game.html`. Accessed: 2019-11-19.

[2] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

[3] A. Blazinskas. Combining 2-opt , 3-opt and 4-opt with k-swap-kick perturbations for the traveling

salesman problem. 2011.

[4] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. 1976.

[5] W. Commons. Glpk solution of a travelling salesman problem, 2019.

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

[7] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[8] D. S. Johnson and L. A. Mcgeoch. The Traveling Salesman Problem : A Case Study in Local Optimization. pages 1–103.

[9] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[10] J. Monnot. Approximation results toward Nearest Neighbor heuristic. *Yugoslav Journal of Operations Research*, 12:11–16, 2002.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.