CSE6140 Final Project: Traveling Salesman Problem

Group 41 - Final Report

Yashovardhan Jallan
Georgia Institute of Technology
Atlanta, Georgia
yjallan@gatech.edu

Cynthia Lee
Georgia Institute of Technology
Atlanta, Georgia
cynthia.lee@gatech.edu

Somdut Roy
Georgia Institute of Technology
Atlanta, Georgia
somdut.roy@gatech.edu

Apaar Shanker
Georgia Institute of Technology
Atlanta, Georgia
ashanker9@gatech.edu

ABSTRACT

The Traveling Salesman Problem is an NP-complete problem with applications in diverse fields, including, but not limited to, computer science, civil engineering, and biomedical engineering. The objective of the project is to implement four different algorithms to solve this intractable problem. As per the project requirements, the selected algorithms represent an exact algorithm, an approximation algorithm with a construction heuristic, and two types of local search algorithms. Specifically, we describe and implement branch-and-bound, Christofides' algorithm, simulated annealing, and neighborhood 2-opt exchange algorithms. The algorithms are implemented on 14 benchmark datasets, and results are compared to the best known solution for each benchmark. The results show that the local search algorithms have the best solution qualities with reasonable run times.

KEYWORDS

Traveling salesman problem, NP-Completeness, branch-and-bound, approximation algorithms, Simulated Annealing, Neighborhood 2-OPT Exchange

1 INTRODUCTION

The Traveling Salesman Problem (TSP) searches for the lowest cost cycle that visits all nodes within a graph. TSP does not have a polynomial time complexity and is therefore intractable. To solve a TSP problem, different algorithms sacrifice run time or solution quality to cope with its NP-completeness. In this project, we investigate, implement, and compare the performances of four different algorithms for solving 14 TSP problems.

The algorithms implemented in this project are branch-and-bound, Christofides' algorithm, simulated annealing, and neighborhood 2-opt exchange. All four algorithms are implemented in Python, and each represents a different category or strategy of algorithms for solving NP-complete problems. Branch-and-bound is an exact algorithm that computes an optimal solution for TSP; Christofides' algorithm is an approximation algorithm based on a minimum spanning tree (MST) construction and its minimal weight matching; and the local search algorithms are two types of heuristic algorithms that try to continually improve upon an initial solution.

Christofides' algorithm runs in the fastest time, but has the highest relative errors compared to the best known solutions. Simulated annealing and neighborhood 2-Opt both produce high quality solutions for all instances and have fast run times for smaller problems. The branch-and-bound algorithm implemented is only able to obtain results for two instances with small city sizes, showing that exact algorithms must make large sacrifices to run time in order to find the optimal solution.

The remainder of this report is organized as follows: First, the definition of the traveling salesman problem (TSP) is restated from the project requirements, followed by a background and summary of related work on solving the TSP. Next, each algorithm implemented for this project is described in detail. We then present the experimental results of each algorithm implemented for 14 benchmark instances. Finally, we compare the performances of the algorithms and state our conclusions.

2 PROBLEM DEFINITION

The formal definition of the TSP for this project is defined in the project requirements: Given the $x - y$ coordinates (locations) of N nodes in a plane and the cost function for every pair of points (i.e., edge), find the lowest cost cycle that visits all N points.

The cost functions are either Euclidean or geographical distances, depending on the input $x - y$ coordinates. Distances between two nodes are equal in both directions (i.e., this is a symmetric TSP problem).

3 BACKGROUND AND RELATED WORK

3.1 Exact Algorithms

Branch-and-bound (BnB) algorithms are commonly used to find the exact solutions of TSPs. The branch-and-bound framework is a class of algorithms used for producing exact solutions to NP-hard optimization problems. This approach was first proposed by Land and Doig [7].

This procedure implicitly enumerates all possible solutions to the problem under consideration, by storing partial solutions called subproblems in a tree structure. Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively (i.e., branching), and rules are used to prune off regions of the search space that are probably suboptimal (i.e., bounding). Once the entire tree has been explored, the best solution found in the search is returned.

3.2 Approximation Algorithms - Construction Heuristics

A variety of algorithms provide approximate solutions to the TSP using construction heuristics. Some of these have approximation guarantees so that the approximated solution has an upper bound, or worst-case solution quality.

Approximation algorithms using construction heuristics generate a solution based on an input graph and do not continually improve upon their solutions. Construction heuristic algorithms include nearest neighbor, Clarke and Wright savings, nearest insertion, closest insertion, farthest insertion, minimum spanning tree approximation, and Christofides' algorithm. The algorithms' upper bounds can be constant factors of the optimal solution (e.g., $2 * OPT$ is the upper bound for minimum spanning tree approximation) or dependent on the input size (e.g. $O(logn) * OPT$) is the upper bound for the Clarke-Wright algorithm). [1]

Construction heuristic algorithms may not compute solutions close enough to an optimal solution for certain applications. However, they are computationally efficient compared to other algorithms and are beneficial for applications in which a α-approximation solution is sufficient.

3.3 Local Search Algorithms

A number of local search techniques have been developed to solve the TSP. Although these do not have approximation guarantees, such algorithms typically produce better solutions than tour construction heuristics. Classic local optimization techniques such as the 2-OPT and 3-OPT exist, in addition to more recently developed variants such as simulated annealing, iterated local search, and Tabu search. 2-OPT is a fast converging algorithm, which is expected to give near-optimal solutions. As a matter of fact, the 2-OPT algorithm has a performance guarantee (i.e. AlgFarthestInsertion/OptimalPathLength) of $\frac{\sqrt{N}}{4}$ [6]. The main drawback of this algorithm is that if it gets stuck at a local minima, it will not be able to find a better solution. We address this by varying starting points for the solution.

For the implementation of our simulated annealing algorithm, we refer to published results in [8]. The authors have experimented with various parameters inherent in the simulated annealing algorithm and also suggest an improved method to identify the best way to generate neighborhood candidate solutions. We implement some of these suggestions in our project.

3.4 Significant Results in Theory and Practice

The TSP is a well-known problem with many state-of-the-art algorithms and techniques to cope with its NP-completeness. For example, Concorde is publicly available code that efficiently solves TSP problems. Developed in 2006, it implements algorithms for finding cutting planes to solve TSP problems. It has since found solutions for all TSPLIB instances, which contain up to 85,900 cities. [3]
Another important implementation that has significant results is LKH, which computes an approximate solution to the TSP using the Lin-Kernighan heuristic. It is based on a local optimization algorithm and has the best solutions for all problems with unknown optima in the 2000 DIMACS TSP Challenge. The challenge has problems with up to 10,000,000 cities. [5]

Overall, the TSP problem has been the focus of many in-depth studies historically and recently in science and engineering applications as well as competitions. Some more recent challenges and applications include finding shortest Pokemon Go routes (2016), the Mona Lisa TSP challenge (2009), and the tour of United States landmarks (2015, based on a 1954 paper). [4]

4 ALGORITHMS

In this section, we describe each algorithm and present their time and space complexities. Certain findings in terms of algorithm performance and solution quality are briefly mentioned and are expanded on in Section 5.

4.1 Branch-and-Bound

Branch-and-bound is a framework used to design algorithms for problems in combinatorial optimization.

Suppose we have a set of feasible solutions to a certain combinatorial optimization problem. Each solution has an objective value, and we want to find the solution that minimizes that objective value. For TSP, this objective value is the length of the tour. The idea is to split the set of all solutions into two or more disjoint subsets. Each of these subsets can also be divided into subsets and so forth. The splitting is based on a constraint that solutions in the subset have to satisfy. This process is called branching.

In this way, we create a tree in which each node contains a subset of all solutions. The root node has no constraints and contains all solutions. Each child node inherits the constraints of its parent and also has an additional constraint. In the context of TSP, the constraints could be the presence of a certain edge in the tour. A node then has two children: the left child, which represents all tours that contain a certain edge and the right child, which represents all tours that do not contain the edge. At a certain point in the tree, the constraints define one single solution. These are the leaves of the tree, and no further constraints can be imposed. The approach is succinctly described below as:

Let $\mathcal{S}$ be some subset of solutions.
Let $L(\mathcal{S})$ = a lower bound on the cost of any solution belonging to S
Let C = cost of best solution found so far

Then if $C \leq L(S)$, there is no need to explore $\mathcal{S}$ because it does not contain any better solution, otherwise this subset needs to be explored

The 2-shortest edges heuristic was used to determine the lowest bound in our implementation. Our implementation was tested against a 4-node toy dataset, UkansasState.tsp and Cincinnati.tsp for which correct optimal solutions were obtained. The implementation could not finish execution for other larger datasets, as the worst case complexity of the branch and bound algorithm is same as that of the brute force method, $O(n!)$.

Algorithm 1 Branch and Bound

```
1:  procedure BnB(V,E)                          ▷ Nodes and edges (with cost)
2:      F ← {(φ, P)}                            ▷ All Possible Tours
3:      B ← (+inf, (φ, P))                      ▷ Best Cost and Tour
4:      while F not Empty do
5:          Choose a branching node v ∈ F
6:          Remove node v from F
7:          Generate children of node v, child_i, i = 1, ··· , n
8:          Corresponding lower cost bounds LB_i
9:          for i= 1 to n do
10:             if LB_i worse than BestCost then
11:                 kill child i
12:             else
13:                 if child is a complete solution then
14:                     best cost = cost(child_i)
15:                     current best = child_i
16:                 else
17:                     add child i to F
```

4.2 Christofides' Algorithm

Christofides' algorithm uses construction heuristics to approximate a solution to the TSP with a 1.5-approximation guarantee. The final approximate solution is based off of a MST of the input nodes and a minimum weight matching of the MST's odd degree nodes. A general description of our implementation of Christofides' algorithm (2) is as follows: We use Prim's algorithm to find a MST of the input graph. The odd degree nodes (i.e., nodes incident to an odd number of edges) in the MST are identified next. The original input graph is restricted to these nodes to create a subgraph, S. We then find a minimum weight matching of S and add the edges in the matching to the MST. Finally, we conduct a depth-first search (DFS) on the extended MST to determine the order of nodes in the final solution. We conduct the DFS $O(n)$ times to determine which root node will result in the shortest length tour. 2 is also outlined in the pseudocode displayed in this section.

Christofides' algorithm has a 1.5-approximation guarantee; each approximate tour will be no longer than 1.5 times the length of the optimal tour, OPT. We briefly describe the reason for this below:

The optimal tour cannot be shorter than the length of the MST because by definition, the MST is a connected tree whose weight is minimized. Moreover, the shortest cycle visiting each of the nodes in S is shorter than OPT and S has an even number of nodes (i.e., there are an even number of odd degree vertices in the MST). The shortest cycle in S can therefore be divided into two matchings. The minimum weight of these matchings is at most half the length of the shortest cycle in S and must also be shorter than half the length of the full optimal tour, $w(minMatch) \leq 0.5 * OPT$. The final approximate tour is constructed from the union of the MST and the minimum matching which is less than $OPT + 0.5 * OPT$. Due to the triangle inequality, the approximate tour - which shortcuts $M \cup minMatch$ to avoid revisiting the same nodes more than once - will be shorter than the length of $M \cup minMatch$. This makes the the final solution found by Christofides' algorithm at most $1.5 * OPT$ [2].

For this approximation guarantee to hold, a perfect minimum weight matching must be found. We use an iterative greedy algorithm to find the minimum weight matching of the subgraph, S (Line 5 of 2). The minimum matching will have $|S|/2$ edges because, in this problem, an edge exists between each pair of points. We sort the edge weights of S in ascending order, and, starting from any edge and its length, we select the next smallest available edge (i.e., an edge that is not incident to a node already covered) to add the the matching, continuing until all nodes are covered. However, depending on which weights are available after an edge is added to the matching, this process alone may not find the minimum weight matching of S. To find the overall minimum, we repeat this process for each edge and update the matching if a new minimum weight matching is found. At each iteration, this greedy process finds the minimum weight matching for a matching that must include the first edge. By repeating this process for all edges, we ensure that the minimum weight matching for S is found.

To implement the algorithm, we use dictionary data structures to store nodes and their priorities for Prim's Algorithm (Line 2 in Algorithm 2) and to construct the final approximate solution. The worst-case time for finding a MST of all nodes is $O(n^2)$. The time complexity for finding the minimum weight matching of S is $O(n^3)$ because the algorithm must iterate through all edges; go through at most all edges at each iteration to fill the matching with available edges; and check whether each node has been covered by the matching at every step.

The DFS to find the order of nodes runs in $O(n + m)$ (Line 12). Within this traverse, a tour and its length are computed from a root node. We repeat the DFS with each node as a root, to find the shortest tour available from the MST $\cup$ minimum matching. This process runs in $O(n^2)$. The total time complexity of the implementation of Algorithm 2 is $O(n^3)$. The space complexity of the algorithm is $O(n^2)$ because it stores at most all edges in both directions (similar to an nxn matrix).

To implement a time cutoff for the algorithm, we stop the algorithm if it has exceeded the time limit in either the minimum weight matching computation or the final approximation. If the algorithm is stopped before the minimum weight matching is complete, it will take the best current matching and the approximation guarantee may no longer apply. If the algorithm is stopped before all possible tours can be found with the DFS, then the algorithm will return the current best tour. In this case, the approximation guarantee will still hold.

Algorithm 2 is a relatively straightforward algorithm that is efficient in finding a solution for smaller graphs. With an approximation guarantee, one of its strengths is that it ensures a solution of at least a certain quality - $1.5 * OPT$. However, the algorithm does not continually improve itself, and our implementation may not be suitable as problem size increases due to its large time complexity. These weaknesses can be detrimental to both solution quality and run time. In practice, the algorithm is more efficient than branch-and-bound and local search algorithms because it does not need to converge to a solution.

4.3 LS1 - Simulated Annealing

Simulated annealing is a method for finding a good (not necessarily perfect) solution to an optimization problem. Problems seeking to

Algorithm 2 Christofides' Algorithm

```
1:  procedure CHRISTOFIDES(V,E) ▷ Nodes and edges (with cost)
2:      M ← Prim(V, E)          ▷ Find MST with Prim's Algorithm
3:      oddV ← FindOdd(M)       ▷ Find odd degree nodes of M
4:      S ← Subgraph of original (V,E) with only oddV
5:      minMatch ← Minimum Weight Matching of S
6:      M ← M ∪ minMatch        ▷ Add mm to the MST
7:      T = start               ▷ Initialize final tour
8:      sol = 0                 ▷ Initialize final tour cost (solution quality)
9:      for i in V do
10:         if i not yet visited then
11:             Mark i as visited
12:             C, curCost ← DFS(M, start) ▷ Starting at any node
13:             if curCost < sol then
14:                 sol ← curCost
15:                 T = C
        return T, sol
```

minimize or maximize a solution can be talked using simulated annealing.

The traveling salesman problem is a good example of such a problem. The salesman is looking to visit a set of cities in the order that minimizes the total number of miles he travels. As the number of cities gets large, it becomes too computationally intensive to check every possible itinerary. At this point, an approximate solution is helpful to improve efficiency.

Simulated annealing's strength is that it avoids getting caught at local maxima.The algorithm is presented in a pseudocode code format in Algorithm 3.

In our implementation, we have started the Simulated Annealing algorithm with an **initial Greedy Solution**. This serves as very good starting point, and we find that solutions obtained after starting at the Greedy solution are significantly improved over a purely random start. After the initialization, we generate neighboring solutions to the greedy solution and check for improved fitness. One way to pick a neighboring tour is to choose two cities on the tour randomly and reverse the portion of the tour that lies between them. For example, if the current solution is [1,3,6,4,5,2], we pick a start-city (e.g., 3) and an end-city (e.g., 5), and reverse the elements in the list from start-city to end-city. So the new candidate solution becomes [1,**5,4,6,3**,2]. We find that this methodology for generating neighboring solutions produces the best results. This is also supported by [8].

Algorithm 3 Simulated Annealing

```
1:  procedure SIM-ANN(runtime, α, T_0, T_end)
2:      while runtime < cutoff-time do
3:          set T = T_0
4:          candidate = initialize as the best known solution so far
5:          while T < T_end do
6:              generate a new candidate solution
7:              if fitness of new solution < best fitness so far then
8:                  update the best solution
9:              else
10:                 accept the new solution using probability check
11:             T = α * T
```

One of the drawbacks of simulated annealing is that by testing random changes, the solution can diverge very quickly. In order to solve this problem, we also implemented **restarts** in the annealing procedure from the best known solution so far. Implementing restarts was found to improve results.

Playing with the parameters: Simulated Annealing has three other parameters; (a) Cooling Factor α, (b) Starting Temperature $\mathbf{T_0}$ and (c) Stopping Temperature $\mathbf{T_{end}}$. We experiment with these parameters as well based on published results. The overall best set of parameters found were $\alpha = \mathbf{0.999}$, $\mathbf{T_0 = 1e + 10}$ and $\mathbf{T_{end} = 0.0001}$.

There are many ways to stop the Simulated Annealing algorithm. In our implementation, we use the user-provided time-cutoff as the stopping condition. For example, files such as *Cincinnati.tsp* take less than a second to converge to an optimal solution. However, *Roanoke.tsp* has very large dimensions, and we cannot verify the optimality of the obtained solution. In such cases, the longer we run the code, the better solutions we get.

Complexity: The time complexity for SA is dependent on the parameters used for starting and ending temperature, and whether or not restarts are used. Since the current algorithm continues running until the user provided time-cutoff, the overall time complexity of the algorithm cannot be attained. But for each iteration at a specific temperature, the operations are governed by calculation of the tour-length, and its time complexity is $O(n)$, where n is the number of locations in the data file. The space complexity of SA is governed by the nxn distance matrix, and hence is $O(n^2)$.

4.4 LS2 - Neighborhood 2-opt exchange

This is a local search algorithm based on considering a route that crosses itself and ensuring that it does not cross itself anymore by reordering the edges. We start with a feasible solution and then check all combinations of swapping to minimize the total weight. The swapping is done in the route until all improvements are reached in the given solution. This algorithm terminates when a local minima is reached. We randomize the starting route for this algorithm, thereby trying different starting solutions in quest for the global minimum distance. The pseudo-code for the 2-opt algorithm is given in Algorithm 4.

While most benchmark instances converge within reasonable

Algorithm 4 Neighborhood 2-Opt Exchange

```
1:  procedure NEIGHB-2-OPT(curr_route)
2:      while solution is improved do
3:          Algo-resume:
4:          best_distance ← TotalDistance(curr_route)
5:          N ← no. of nodes that can be swapped
6:          for i = 1; i < N − 1; i + + do
7:              for k = i + 1; k < N; k + + do
8:                  new_route ← 2OptSwap(curr_route, i, k)
9:                  dist ← TotalDistance(new_route).
10:                 if dist < best_distance then
11:                     curr_route ← new_route
12:                     Goto Algo-resume
```

times, the results for Roanoke did not converge as quickly. The cut-off of 600 seconds did not suffice for a good solution. For this instance, we instead used a time cut-off of 1800 seconds for this file to converge.

Complexity: The complexity of 2-opt will be $O(n^2)$ because at each step, the algorithm will look for at worst $O(n^2)$ combinations of swapping for improvement (lines 6-11 of the pseudo-code).

5 EMPIRICAL EVALUATION AND RESULTS

All programs are written in Python and run on Windows 10 operating system. Our best known solutions for this project are listed in Table 1. Solutions are either optimal solutions known from TSPLIB, or the best solution found at any time by any of the approximation algorithms (i.e., Christofides' or local searches). Exact results are known for Boston, Cincinnati, Berlin, and Ulysses16. All results (i.e., run times, solutions, and relative error rates) for the algorithms are available in Table 2. Relative errors are calculated as $(Alg - OPT)/OPT$. **Note: for local search algorithms, experiments were run with multiple random seeds and averaged results are presented in Table 2.**

Table 1: Best Known Solutions

Dataset	Sol. Qual.
Boston	893536
Cincinnati	277952
Berlin	7542
Ulysses16	6859
UKansasState	62962
UMissouri	134624
Toronto	1209235
SanFrancisco	834446
Roanoke	690354
Philadelphia	1395981
Denver	102116
Champaign	52643
Atlanta	2003763
NYC	1555060

5.1 Exact Solution - Branch and Bound

The algorithm was implemented in Python and tested on both Windows and Linux machines. The algorithm returned outputs for files with 10 nodes but could not finish execution for larger files. The results in the submitted output files are tabulated.

5.2 Approximation - Christofides' Algorithm

The implementation for Christofides' algorithm was run on a computer with 8GB RAM and a 2.30 GHz Intel i7 processor. The best solution quality for Christofides' Algorithm (in terms of relative error to the best known solution) is for Ulysses16 (0.033 relative error), and the worst solution quality is for San Francisco (0.2655 relative error). Run times are less than one second for all instances except Roanoke, which ran for 14.27 seconds.

As previously described, the upper bound for Christofides' algorithm is $1.5 * OPT$. For the four known exact solutions, the algorithm's results are well within this range. Its worst solution compared to a known optimal solution is $1.254 * OPT$.

5.3 LS1 - Simulated Annealing

Simulated Annealing was run on a system which has a RAM of 16 GB and a 3.41 GHz Intel i7 processor. All the instances of the TSP data files were run for several random seeds and averaged results are presented in Table 2.

As seen in Table 2, for all cities with dimensions less than or equal to 30 locations, (i.e., Cincinnati, UKansasState, Ulysses16, Atlanta, and Philadelphia), the algorithm finds the optimal solution in about 1 second for all random seeds (100 random seeds tested). As the number of locations within the city grew larger, not all random seeds produced a converged result. Specially, for the cities with a very large number of locations, we know that the best solution obtained is quite possibly not the optimal one.

Coming to the QRTDs, SQDs and Box-Plots (Figures 1-5), we present the plots for Atlanta and Philadelphia. This is because we are able to obtain optimal results for all 100 random seeds for these cities in a reasonable time, which allows for a good interpretation of the plots and the associated results.

From the QRTD plot for Atlanta (Figure 1), we can see that when the run-time is less than 0.1 second, the various solution qualities have a fraction of runs between 0.8 to 0.9 that have solved the problem. However, at about 1 second, almost all types of solution quality find the optimal solution. The same values are also presented in the SQD plot for Atlanta (Figure 2) with different time cut-offs as shown in the plot legend. The box plot for Atlanta (Figure 5a) shows that the mean run-time for the optimal solution was very close to 0.5 second. Similar interpretations can be derived from the plots for Philadelphia as well (Figure 3, Figure 4 and Figure 5b).

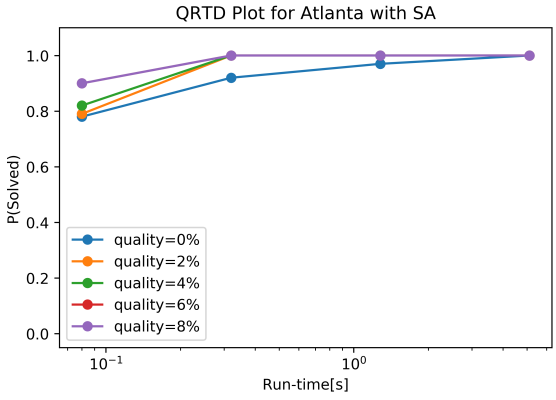


Figure 1: QRTD Plot - Atlanta: LS1

Table 2: Comprehensive Table

	Branch-and-Bound			Christofides'		
Dataset	Time (s)	Sol. Qual.	RelErr	Time (s)	Sol. Qual.	RelErr
Boston	-	-	-	0.011	1003112	0.1226
Cincinnati	2.25	277952	0	0.001	299282	0.0767
Berlin	-	-	-	0.049	9460	0.2543
Ulysses16	-	-	-	0.001	6955	0.033
UKansasState	0.41	62962	0	0.001	66335	0.0536
UMissouri	-	-	-	0.392	158369	0.1764
Toronto	-	-	-	0.596	1359463	0.1242
SanFrancisco	-	-	-	0.558	1055954	0.2655
Roanoke	-	-	-	14.270	766700	0.1106
Philadelphia	-	-	-	0.001	1549195	0.1098
Denver	-	-	-	0.248	126016	0.234
Champaign	-	-	-	0.057	60570	0.1506
Atlanta	-	-	-	0.012	2110955	0.0535
NYC	-	-	-	0.195	1774632	0.1412

	Simulated Annealing			Neighborhood 2-Opt		
Dataset	Time (s)	Sol. Qual.	RelErr	Time (s)	Sol. Qual.	RelErr
Boston	43.694	893706.73	0.0002	1.0125	927164.71	0.0376
Cincinnati	0.089	277952	0	0.0018	280023.27	0.0075
Berlin	14.269	7655.94	0.0151	2.1847	8121.75	0.0769
Ulysses16	0.389	6852	0	0.032	6852	0
UKansasState	0.083	62962	0	0.0018	62962	0
UMissouri	311.831	142382.1	0.0576	53.6245	139687.81	0.0376
Toronto	275.167	1264872.4	0.046	87.2537	1241752.8	0.0269
SanFrancisco	486.341	849754.7	0.0183	40.1695	872876.65	0.0461
Roanoke	493.659	837729.9	0.2135	1436.6279	703231	0.0187
Philadelphia	1.909	1395981	0	0.3783	1430524.79	0.0247
Denver	353.120	103168.1	0.0103	19.1068	106673.12	0.0446
Champaign	191.615	52653.1	0.0002	3.1369	54291.39	0.0313
Atlanta	0.735	2003763	0	0.0727	2047920.11	0.022
NYC	258.251	1560261.8	0.0033	10.9483	1610640.77	0.0357

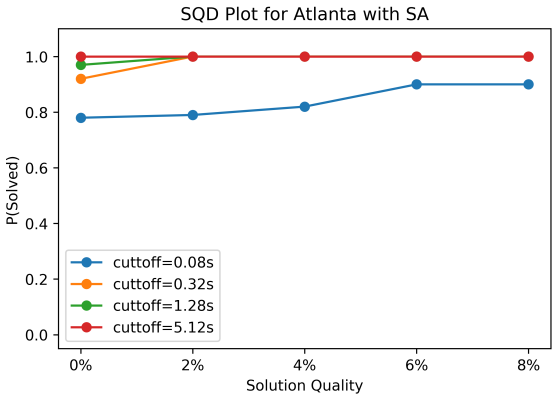


Figure 2: SQD Plot - Atlanta: LS1

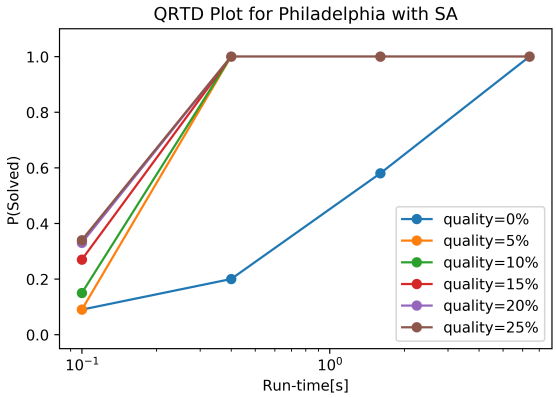


Figure 3: QRTD Plot - Philadelphia: LS1

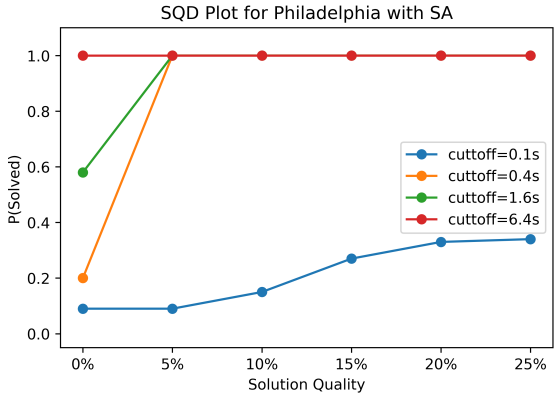
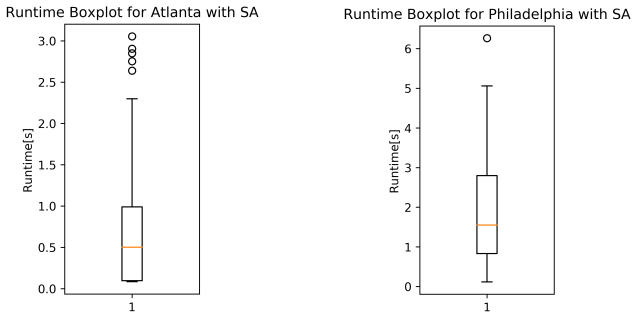


Figure 4: SQD Plot - Philadelphia:LS1



(a) Boxplot Atlanta **(b) Boxplot Philadelphia**

Figure 5: Boxplots: LS1

5.4 LS2 - Neighborhood 2-opt exchange

Neighborhood 2-opt exchange was run on a system which has a RAM of 16 GB and a 3.60 GHz Intel i7 processor. The TSP files for all provided cities were run multiple times with different random seeds. The results from the multiple runs are averaged out and compared to the best solution found from all the algorithms (in cases where the actual solution could not be found) in Table 2.

As with simulated annealing, box-plots, QRTDs and SQDs are presented in this section. We ran the algorithm for relatively small cities with 100 different random seeds. The run-time and solutions at different time-stamps are post-processed to create box-plots of run-time, QRTD, and SQD. The box-plots for run-time reflect how the different run-times varied with different starting solutions, whereas the QRTD and SQD plots show how close the algorithm can get to the optimal solution with varying time-constraints.

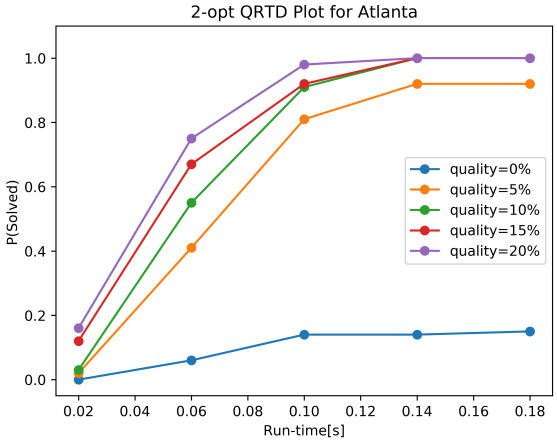


Figure 6: QRTD Plot - Atlanta: LS2

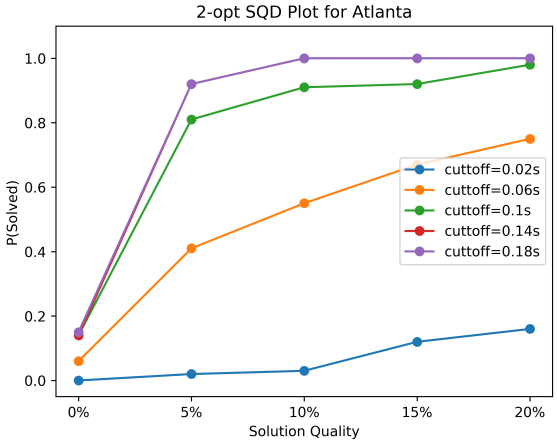


Figure 7: SQD Plot - Atlanta: LS2

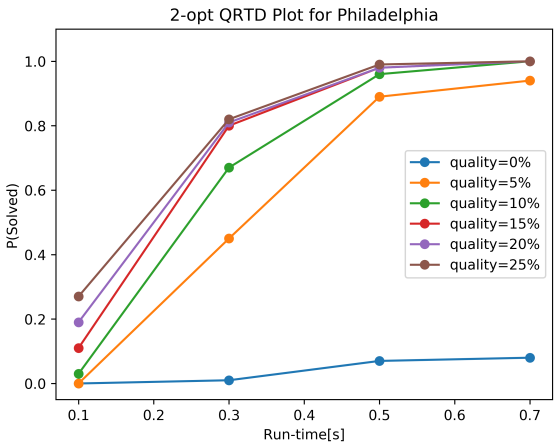


Figure 8: QRTD Plot - Philadelphia: LS2

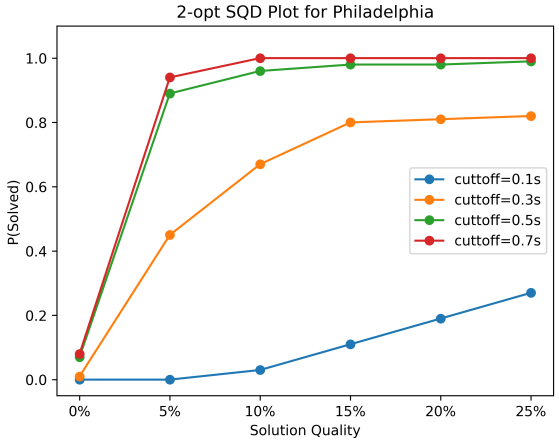
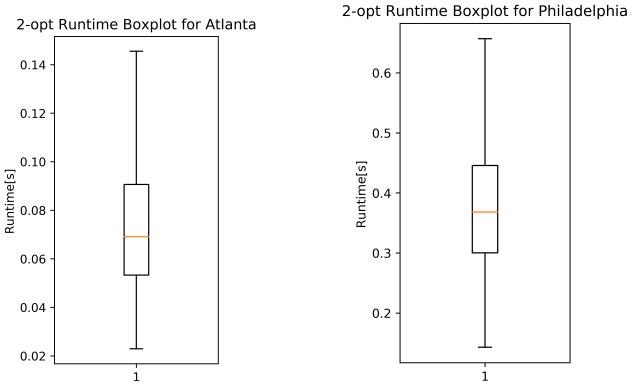


Figure 9: SQD Plot - Philadelphia: LS2



(a) Boxplot Atlanta **(b) Boxplot Philadelphia**

Figure 10: Boxplots: LS2

As is evident from the QRTD and SQD plots for both Atlanta and Philadelphia, around 20 percent of all 100 runs can achieve the best solution. The box-plot for Atlanta shows that the run-time distribution is slightly left-skewed, whereas the box-plot for Philadelphia shows the run-time distribution is closer to normal.

6 DISCUSSION

As seen in Table 2, we find that Branch-and-Bound is successful in finding the optimal solutions for all the instances where the number of locations in the city are relatively small. However, when as the problem size increases, it is virtually impossible to find an optimal solution in reasonable amount of time. For 10 nodes, the current implementation executes in 3 seconds. However, the next smallest file with 16 nodes could not be traversed. The program had to be terminated prematurely, such that the solution has repeated nodes.

In such cases, we look at alternatives like approximation and local search algorithms. With Christofides' algorithm, an approximation algorithm, we computed solutions less than 1.5 times the optimal solution in much less time than possible with branch-and-bound. While the time complexity for our implementation is still not ideal for large numbers of nodes, the algorithm runs with reasonable times for all 14 instances and has the fastest run times of the algorithms implemented for this project. Christofides' algorithm did not produce the best known solution quality for any of the benchmark instances. This is expected for a construction heuristic algorithm because, as previously stated, it does not continually improve upon its solution until convergence.

Using Local Search algorithms like Simulated Annealing and Neighborhood 2-Opt, we do not necessarily have a upper bound or a lower bound on our solutions, but the obtained results are either optimal or the best known solution. Neighborhood 2-Opt is a fast algorithm that converges at a local optima. The rate of convergence of the algorithm will depend on how the solution set is distributed. If the global minima is at the bottom of a very steep trough, this algorithm will find it hard to locate that absolute solution. However in cases where there is only one minima, this algorithm will find it very quickly. Thus, the convergence depends not only on the number of nodes n, but also on the distribution of the peak and trough of the solution set.

Compared to Neighborhood 2-Opt, Simulated Annealing has longer run times, but produces better solutions more consistently. This is because of the nature of the algorithm; it does not converge at a local optima, but keeps searching for the global optima until the algorithm is stopped.

7 CONCLUSIONS

The exact algorithm implemented using branch-and-bound was able to compute the optimal tour for the two smallest cities. However, it is the slowest algorithm and is prohibitively so for the files with larger numbers of nodes. The implementation can be made more efficient through better lower cost bounds to prune the tree and better exploration strategies such as greedy algorithms. The branch-and-bound implementation serves to benchmark the other algorithms, on smaller networks.

The approximation algorithm implemented for this project, Christofides', has the fastest run times and obtains solutions with up to 1.266 times the length of the best known solutions. With a correct computation of minimum weight matchings, the algorithm has a 1.5-approximation guarantee. For all instances for which the optimal solutions are known, the algorithm obtained solutions well within this upper bound.

The algorithm's time complexity may not be suitable for very large graphs, but the algorithm ensures that the solution quality will not be worse than its approximation guarantee. This is an important strength for any algorithm with an approximation guarantee, especially if the optimal solution is not known and sufficient solutions need to be found efficiently. Moreover, without a known optimal, it is important to know the worst possible quality of an approximate solution.

With simulated annealing, we find that this algorithm efficiently produces accurate solutions for all the cities with relatively small numbers of locations (50-55 locations or less). Most solutions are found within seconds, and at worst, in a couple of minutes. However, as the number of locations in the city increases, the solution

quality deteriorates and it is difficult to know if the obtained solution is close to optimal or not. In theory, if we run the algorithm for infinite time, it should give us the optimal solution. However, in practice, that is not possible.

As shown in Table 2, 2-opt converges to a solution the fastest for the cities with fewer locations. However, for a city like Roanoke, it takes 30 minutes to converge to a solution, even though it gets to a better solution than the other algorithms. By and large, it can be said that this algorithm when compared to simulated annealing is a trade-off between time (LS2) and quality (LS1).

The algorithms implemented for this project demonstrate the strengths and weaknesses of exact and approximation algorithms for solving the TSP. These are just four of many other - and more extensive - methods for coping with NP-complete problems. Each of the approximation algorithms was able to obtain reasonable solutions to all 14 benchmark instances within user input cutoff time windows. The branch-and-bound algorithm, while unable to reach a solution for most of the benchmark instances, is still an important experiment for this project to demonstrate the trade-off in run time that must be made to implement an exact algorithm.

REFERENCES

[1] T. Doyle B. Golden, L. Bodin and W. Stewart Jr. 1980. Approximate Traveling Salesman Algorithms. *Operations Research* 28, 3 (1980), 694–711.
[2] N. Christofides. 1976. Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem. *No. RR-388* (Feb. 1976). Carnegie Mellon University Management Sciences Research Group.
[3] William Cook. 2015. Concorde TSP Solver. http://www.math.uwaterloo.ca/tsp/concorde/index.html
[4] William Cook. 2018. The Traveling Salesman Problem. Retrieved December 4, 2018 from http://www.math.uwaterloo.ca/tsp/index.html
[5] K. Helsguan. [n. d.]. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *Writings on Computer Science* 81 ([n. d.]).
[6] D.S. Johnson and L.A. McGeoch. 1997. The traveling Salesman Problem: a Case Study in Local Optimization. In *Local Search In Combinatorial Optimization* (1st. ed.), E. Aarts and J.K. Lenstra (Eds.). Wiley, New York, NY.
[7] A.H. Land and A.G. Doig. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28, 3 (July 1960), 497–520.
[8] D. Tian Y. Wang and Y.H. Li. 2013. An Improved Simulated Annealing Algorithm for Travelling Salesman Problem. *International Journal of Online Engineering* 9, 4 (July 2013), 525–532. https://doi.org/10.3991/ijoe.v9i4.2822