A Study on Traveling Salesperson Problem

Rojin Aliehyaei
Georgia Institute of Technology
Atlanta, GA, USA
rojin@gatech.edu

ABSTRACT

This study evaluates six algorithms for solving metric version of Traveling Salesperson Problem. First, an exact algorithm based on branch-and-bound (BnB) strategy is implemented and evaluated. Since the exact algorithm is computationally expensive, we also implement three constructive heuristic methods based on Minimum Spanning Tree, Cheapest Insertion, and Christofides. Finally, we implement two algorithms based on stochastic local search strategy including Simulated Annealing and Iterative Local Search. The paper also provides the theoretical and empirical analyses on run-time, solution quality, and relative error of each algorithm.

We found the exact algorithm such as BnB works the best for smaller instances. The constructive heuristic methods implemented in this paper are polynomial and their solutions are generated in less than a second. The drawback of solutions based on constructive heuristic methods is their poor quality. The performance measures show the program based on stochastic local search have a good potential to achieve a high quality solution within a reasonable amount of time.

KEYWORDS

Traveling Salesperson Problem, Local Search, Branch-and-Bound, Approximation algorithm

1 INTRODUCTION

Traveling Salesperson Problem (TSP) is one of the NP-hard problems with a wide variety of applications. Several problems such as vehicle routing, minimizing the telescope move between sources, and printed circuit board manufacturing have been studied as instances of TSP [?]. This project reviews and evaluates different methods to solve TSP. The evaluations are conducted on 14 TSP benchmark instances from TSBLIB [?]. The TSP instance sizes are ranging from 10 to 230 locations.

The first algorithm considered for solving TSP is based on branch-and-bound (BnB) approach. To increase the efficiency of the implementation, we use lower and upper bounds for better pruning. However, for larger instances, the method is still computationally expensive. Thus, we also implement three heuristic algorithms including MST-based heuristic, Cheapest Insertion, and Christofides methods. To find a solution with high quality in reasonable amount of time, we also select two algorithms based on stochastic local search approach. Specifically, we implement the Simulated Annealing and an Iterated Local Search based on 2-opt method.

A formal definition of TSP is presented in Section 2. In Section 3, we briefly summarize prior works on TSP. In Section 4, a pseudo code and theoretical complexity and guarantee for each method are discussed. An empirical evaluation for each instance with the evaluation criteria, such as run-time, solution quality, and the relative error is provided in Section 5. Afterward, in Section 6, we discussed how different algorithms perform with respect to evaluation criteria and expected time complexity. Finally, we provide a summary and conclusion in Section 7.

2 PROBLEM DEFINITION

In TSP, we are given a set of cities $\{c_1, c_2, c_3, ..., c_n\}$ and for each pair of (c_i, c_j) s.t. $i \neq j$, a distance $dis(c_i, c_j)$ is defined. The objective function is minimizing the length of the tour a salesperson makes when visiting each city exactly once and returning back to origin [?]. Assuming we are given an ordering p of cities, the objective function is defined as follows:

$$total\_length = \sum_{i=1}^{n} dis(c_{p(i)}, c_{p(i+1)}) + dis(c_{p(n)}, c_{p(1)}))$$

We are solving a version of TSP that is symmetric, in which $dis(c_i, c_j) = dis(c_j, c_i)$ for all and the distances must satisfy triangle inequality. This means $dis(c_i, c_j) \leq dis(c_i, c_k) + dis(c_k, c_j)$ for all $1 \leq i, j, k \leq n$.

Depending on the data type of an instance, the distance is computed based on either Euclidean or Geographical distance. Each calculated distance is rounded to a nearest integer. The problem is formulated as a fully connected graph with n vertices representing n cities. The weight of an edge between a pair of vertices is equal to the distance between two cities representing those two vertices.

3 RELATED WORK

There are a wide range of methods for solving TSP from simple heuristic-based method to complex technique such as Ant Colony Optimization. In this Section we briefly reviewed some of prior works in three categories: (1) exact algorithms, (2) constructive heuristic, and (3) stochastic local search.

3.1 Exact Algorithms

A naive approach to produce an exact solution for TSP is a brute force approach. The algorithm enumerates all valid tours and selects the tour with minimum cost. A more systematic exact procedure for solving TSP is based on BnB algorithm. Over the last decade, a rapid growth of computer speed and storage capacities as well as parallel computing algorithms lead to an improvement in the speed of exact algorithms such as BnB. One of the important methods for improving the efficiency of BnB method is maintaining a provable lower and upper bounds on the cost [? ?]. Various relaxations and lower bound computation strategies have been proposed for solving metric TSP [?]. Held and Karp [?] developed a BnB algorithm based on minimum cost 1-tree for solving TSP. Later, one of the tightest lower bounds based on 1-tree Lagrangian relaxation was developed in [?] which originally proposed by Held and Karp [?]. In this study, a simple and efficient lower bound strategy based on 2 shortest edges is used in the implementation of BnB algorithm.

3.2 Constructive Heuristic Algorithms

A constructive heuristic algorithm starts with an empty solution and expands the current solution iteratively until a valid solution is generated. One of the classic constructive heuristic methods is called Nearest Neighbor (NN) algorithm with the worst case guarantee of $(1/2)(\log n + 1)$ and time complexity of $O(n^2)$, where n is the number of locations in a TSP instance. Closest Insertion, Cheapest Insertion, and Approximate TSP tour based on Minimum Spanning Tree are other known constructive heuristic methods. All these methods are considered as 2-approximation method and run in polynomial time. The other constructive heuristic methods are Clarke-Wright and Christofides algorithms. Christofides is a heuristic method with the worst-case ratio of 1.5 and time complexity of $O(n^3)$ [? ?]. In this study, we implement the approximate algorithms based on Minimum Spanning Tree, Cheapest Insertion, and Christofides algorithm.

3.3 Stochastic Local Search

Local search is a heuristic approach that finds a solution in the space of candidates by improving an initial solution. The search procedure is possible by defining a neighborhood relation on the search space. Generally, having a large neighborhood allows for an exploration in different regions of solution space, and as a result, finding a better solution. However, in most cases, searching in such neighborhoods are exponentially expensive. One way to deal with this problem is neighborhood pruning. Using a candidate list that contains a limited number of closest direct neighbors for each vertex in the graph is an effective pruning technique. One of the most famous and efficient local search algorithms for solving TSP is 2-opt algorithm which is based on 2-exchange neighborhood [? ?].

One challenge in local search is when the method eventually stuck in a region of the space, and we need a perturbation strategy, such as double-bridge to restart the local search procedure [?]. Some of the local search algorithms, such as Simulated Annealing (SA), Tabu Search, and Iterated Local Search (ILS) algorithms accept a worsening search step that allows more extensive search and eventually leads to better solutions. In this project, we implement and analyze the performance of SA and ILS on TSP instances from TSPLIB95.

4 ALGORITHMS

4.1 Branch-and-Bound

In this study a non-recursive branch and bound (BnB) based on Depth First Search (DFS) order is utilized. Apart from keeping track of the most optimal solution (optimal tour), the lower bound of each state is computed. This helps to prune the tree and discard certain branches. The branching ends when either the corresponding state produces a valid solution or when its lower bound is larger than current optimal cost [?]. BnB's pseudo code is shown below.

```
Branch-and-Bound($G$)
Initiate an empty stack called $stack$
Select a vertex $v_0$ and initiate its corresponding state $s_0$
$stack \leftarrow s_0$
$cost(current\_best\_solution) \leftarrow +\infty$
$current\_best\_solution \leftarrow \emptyset$
while $stack$ is not empty do
    $s \leftarrow$pop($stack$)
    if $lower\_bound(s) < cost(current\_best\_solution)$
        if $s$ is a feasible solution
            if $cost(s_i) < cost(current\_best\_solution)$
                $current\_best\_solution = s_i$
        else if $lower\_bound(s_i) < cost(current\_best\_solution)$
            expand $s_i$ into new states $s'_1, ..., s'_k$
            for $s'_1, ..., s'_k$
                if $lower\_bound(s'_i) \leq cost(current\_best\_solution)$
                    add $s'_i$ to $stack$
return $current\_best\_solution$
```

The input of the algorithm is the adjacency matrix G which is used for computing the costs. s_0 is the state of problem at the beginning that only one city is visited. Since the size of problem is always larger than one city, at this point no valid solution (complete tour) exists. Each state (called s_i) represents a partial solution and the remaining sub-problem. In TSP, each state s_i is associated with a group of visited cities, current cost, and a calculated lower-bound. If all cities have been visited, it means we have a complete solution and we can compare it against the cost of current best solution and update the best solution, if it is needed. If a state is not associated with the complete tour, we compare the state's lower bound against the current best solution. If the state's lower bound is smaller than cost of best solution, it is considered as a candidate for further exploration and added to the stack. We add the states to the stack in order of the largest to smallest lower bound. In this way, the state with the smallest lower bound is at the top of stack. Common strategies to compute the lower bound are: **(a)** $1/2 \times$ (Sum of the cost of two minimum weight edges adjacent to u) for all $u \in V$, and **(b)** estimation of lower bound based on Minimum Spanning Tree.

The BnB algorithm guarantees to find TSP's exact solution. The worst running time grows faster than an exponential function since there are a total of $n!$ different solutions. By defining a smart lower bound, we could limit the choices and prune the tree. In the implementation, we calculate the lower bound based on option (b). The time complexity of implemented algorithm is bounded by $O((n^2) \times n!)$ because lower bound calculation requires to sum over the two minimum edges adjacent to each vertex and there are at most $n!$ states in the search tree. To avoid redundant sorting operations, at the beginning, we create a sorted version of graph such that for every vertex we have a sorted set of its adjacent vertices. Thus, it takes $O(n^2)$ time to calculate a lower bound.
For each state, we need to keep track of visited vertices and the lower-bound. In DFS strategy, we have at most n levels and the branching factor is n. Since we perform the search in *DFS* fashion, the maximum number of states at any point is n^2. The maximum

required space for each state is an array to store the partial path. Thus, the total space complexity is $O(n^3)$. Since the calculated upper bounds and lower bounds help to prune the search tree, in practice we expect the BnB algorithm to perform significantly better than $O((n^2) \times n!)$. The benefit of the program based on BnB approach is that when it terminates normally, the final solution is optimal. The drawback is the program is computationally expensive and it is not a practical method for large TSP instances.

4.2 Approximate TSP Tour using Minimum Spanning Tree

When TSP instance size is small, exact algorithms with exponential time (such as BnB) is a perfect option since it gives one of the best solutions in a reasonable amount of time. As the problem size increases the run-time grows rapidly and other approaches to find a near optimal solution in a short time are preferable. In this section, we explain an approximate algorithm based on Minimum Spanning Tree (MST) for solving TSP. The pseudo code for the approximate TSP solver is shown below.

```
Approximate_TSP_Tour($G, c$)
Select a vertex $r \in V$ to be a root vertex
$T = MST(G, r)$
Let $P$ be a list of vertices that are ordered according to when they
first visited in a preorder_tree_walk($T$)
return $P$
```

First, we implement Kruskal's algorithm to find a MST tree called T for a given graph. Next, we create a tour using preorder tree walk implemented based on *DFS* algorithm. The list of vertices according to the order first visited using preorder tree walk is a solution for the TSP and its total cost is not more than twice the cost of an optimal tour. We can prove Approximate_TSP_Tour is 2-approximation algorithm for TSP with triangular inequality as follows. Assume a full walk, called W, traverses every edge of T exactly twice. Thus, we have: $W = 2cost(T)$. The full walk W is not a tour since it visits every vertex twice. Using triangle inequality, we can delete a vertex m from W between visit u and v and go directly from u to v. We can repeatedly applying this operation and the final ordering will be equivalent to P. Thus, we have: $cost(P) < cost(W) \longrightarrow cost(P) < 2cost(MST)$.

The overall complexity of Kruskal's with union find and path compression is $O(m \log n)$. The total time complexity of preorder_tree_walk (based on DFS) is $O(m + n)$. Since this a fully connected graph the total time complexity is $O(n^2 \log n)$. The space complexity is dominated by the amount of storage required for storing the adjacency matrix which is $O(n^2)$.

4.3 Cheapest Insertion

Various heuristic methods, such as Nearest Insertion, Farthest Insertion, Random Insertion, and Cheapest Insertion, have been designed to solve metric TSP [?]. For this project, we implement the cheapest insertion method. Its pseudo code is shown below.

```
Method of Cheapest Insertion(G)
$sub\_tour = []$
Choose a random vertex i as a starting node.
Find a node $j$ that is closest to $i$ and form a sub_tour = [$i$-$j$-$i$].
while ($length(sub\_tour) \neq n$)
    Find an edge $(i, j)$ of the sub_tour and a node $k$ not in the
        $sub\_tour$, s.t. $\min_k \Delta = G_{ik} + G_{kj} - G_{ij}$
    Modify the $sub\_tour$ by inserting $k$ between $i$ and $j$.
return the $sub\_tour$
```

In this method, we start with a *sub_tour* containing an arbitrarily chosen vertex i. Next, we find the closest vertex to i, called j, and build a sub_tour= [i-j-i]. Then, we find an optimal k such that $\min_k \Delta = G_{ik} + G_{kj} - G_{ij}$. We repeat the insertion until we have a complete tour. The total time complexity of implemented algorithm is $O(n^3)$. The auxiliary space for this method is $O(n)$ and the space required for input is $O(n^2)$. The total space complexity is $O(n^2)$. The total cost of a tour obtained by this method is not more than twice the cost of an optimal tour [?].

4.4 Christofides

Christofides algorithm is one the best known algorithm to find an approximate solution to metric TSP. It provides a worst-case guarantee of 1.5 that is better than MST approximation and cheapest insertion. A pseudo code for this method is shown on the next page. First, a MST tree using Kruskal's algorithm is created as T. Then, a list of odd degree vertices derived from T.

```
Christofides($G$)
Select a vertex $r \in V$ to be a root vertex
$T \longleftarrow MST(G, r)$
$odd\_list \longleftarrow$ find a set of vertices with odd degree in T.
Find a $minimum\_weight\_matching$ of subgraph derived by $odd\_list$
$H \longleftarrow$ Combine edges in $minimum\_weight\_matching$ withT
$ET \longleftarrow$ Form a Eulerian circuit in $H$
Let $tour$ be a list of vertices that are ordered according to when
they first visited in $ET$
return tour
```

Since $\sum_{v \in V} deg(v) = 2|E|$, the number of vertices with odd degree is always even. Thus, the algorithm can find a minimum weight perfect matching in induced sub-graph with odd degree vertices. Next, the edges in minimum weight perfect matching subgraph and T are combined and an Eulerian Tour from the resulting graph is built. Finally, we can create a path by skipping repeated vertices in the Eulerian Tour. The overall complexity of implemented Kruskal's algorithm is $O(n^2 \log n)$. Since the number of edges in MST tree is $n - 1$, finding a set of vertices with odd degree in T takes $O(n)$. To find a minimum weight matching in non-bipartite graph, a function from networkx package is used. This function is implemented based on blossom algorithm, which is developed by Jack Edmonds in 1961 [?]. The algorithm runs in $O(n^3)$ time. Finally, the time complexity for building an Eulerian circuit and final tour is $O(n^2)$. Thus, the overall time complexity of implemented algorithm is $O(n^3)$. To perform each step, we have created 3 different versions of graph input. Thus, the total space complexity is $O(m + n)$ or $O(n^2)$. The algorithm guarantees that its final tour cost is within a factor of

$\frac{3}{2}$ of the optimal solution cost [?]. The approximate algorithms presented in this study runs quickly and they provide an approximation guarantee. The drawback is that the solutions from greedy heuristic methods have poor quality.

4.5 Simulated Annealing

The simulated annealing (SA) algorithm is based on the physical process of heating the material and then gradually lowering down the temperature to increase the size of crystal and reduce the defects. In the implementation, we need to set several parameters including staring temperature, minimum temperature, and cooling rate. The initial solution is obtained using Nearest Neighbor (NN) method, which takes $O(n^2)$. SA's pseudo code is shown below [? ?].

For finding a random neighbor, we perform a single 2-opt exchange. The method first breaks the tour by deleting two edges and then reconnect the path using the other possible way. If a new candidate solution is better than current solution(s), we update the current solution and check if we need to update the best solution (s^*). However, if the cost of new candidate is not better than the cost of current solution, we may still accept it as the current solution randomly. Specifically, we compare a threshold value, $threshold = e^{-\frac{\Delta}{current\_temp}}$, against a randomly generated number in range $[0, 1]$. If the random value is smaller than calculated threshold, we accept the solution. To improve the search process, we only consider k nearest neighbors of a vertex for 2-opt exchange. For further perturbation, we also check if the solution has not improved for a certain number of iterations. If that is the case, we apply the double-bridge on the current solution. Double-bridge move is introduced by [?]. This technique combines two 2-opt exchanges and breaks four edges of current tour and introduces four new ones.

Simulated-Annealing

```
Generate an initial solution s using a greedy heuristic method
Set initial best solution as s, ($s^* = s$)
Set parameters including $current\_temp$, $min\_temp$, $cooling\_rate$
while($current\_time > cutoff\_time$ and $min\_temp < current\_temp$)
    if (cost(s) is not improving for $m$ consecutive iterations)
        $s' = Perturbation(s)$
    else:
        Choose a random neighbor $s' of s$.
    $\Delta = cost(s') - cost(s)$
    if ($\Delta \leq 0$)
        $s = s'$
        if ($cost(s) < cost(s^*)$)
            $s^* = s$
    else
        Choose a random number uniformly from [0,1]
        if $r < e^{-\frac{\Delta}{current\_temp}}$
            $s = s'$
    $current\_temp = cooling\_rate \times current\_temp$
```

Decreasing the temperature value helps to lower the acceptance rate of a worse solution over time. The time complexity for performing a single 2-opt move and a double-bridge move is $O(n)$. The main loop iterates based on user defined cut-off time and initial temperature. Thus, we cannot define a deterministic time bound on the algorithm. Since we initiate the solution using NN method, it is guaranteed that the final cost is not worse than $0.5(\lceil \log_2 n \rceil + 1)$ [? ?]. In terms of space complexity, the auxiliary space used by algorithm is $O(n)$. The required space for storing adjacency matrix is $O(n^2)$. Thus, the overall space complexity is $O(n^2)$.

4.6 Iterated Local Search

In most local search processes, after a period of time, the local search gets stuck at a local optimal point. We can skip the local optima by using a more complex method such as Iterated Local Search (ILS). The ILS procedure is shown on next page. Applying change or perturbation to a current s^* will result into a new candidate solution s' [?]. For the TSP, in each iteration, the algorithm performs a local search based on 2-opt method [?].

Iterated Local Search

```
$s_0$ = create a random tour
$s^* = LocalSearch(s_0)$
while ($time\_length < cutoff\_time$)
    $s' = Perturbation(s^*)$
    $s^* = LocalSearch(s')$
return $s^*$
```

The best performance guarantee possible for 2-opt is a ratio of at least $\frac{1}{4}\sqrt{n}$. Generally, the ratio of cost of the final optimal solution (generated by 2-opt) to the cost of best solution is a constant factor. The maximum number of iterations for finding an improving exchange is $O(n^2)$ and performing such exchange is bounded by $O(n)$. Thus, the overall complexity is $O(n^3)$ [?]. Double-bridge perturbation [?] is selected for restarting the local search. The time complexity for performing double-bridge is $O(n)$. The main loop iterates based on user defined cut-off time. Thus, we cannot define a deterministic time bound for the algorithm. In terms of space complexity, the auxiliary space used by algorithm is $O(n)$. The largest amount of storage is required for storing the adjacency matrix which is $O(n^2)$.

The benefit of stochastic local search methods is their high potential to generate solutions with good quality. The drawback is without any prior knowledge, we don't know if the final solution is optimal or sub-optimal and we cannot define a deterministic time bound for the local search.

5 EMPIRICAL EVALUATION

For this project, 14 instances from TSPLIB are considered as inputs. HP Z240 Workstation (Intel Core i7-7700 3.60GHz having 4 physical cores and 8 logical processors with 8MB Cache and 64GB of DDR4 RAM) is deployed for all experiments. All the algorithm is developed using Python2.7, NumPy, and networkx-2.2. Specifically, fix method used from NumPy library. We use networkx in Christofides method to find a minimum weight matching in the sub-graph.

The quality of the solution is the total tour length. The execution time is measured in seconds. Specifically, the start time is recorded after the program takes the required arguments through the command line. The results of all runs are stored in a solution output

Table 1: Cost of Best Tours for Each Instance

Instance	Size	Best Sol.Qual.	Best Algorithm
Atlanta	20	2003763	BnB, SA, ILS
Berlin	52	7542	SA, ILS
Boston	40	893536	SA, ILS
Champaign	55	52643	SA, ILS
Cincinnati	10	277952	BnB, SA, ILS
Denver	83	100431	ILS
NYC	68	1555060	SA, ILS
Philadelphia	30	1395981	SA, ILS
Roanoke	230	657790	ILS
SanFrancisco	99	810196	SA, ILS
Toronto	109	1176151	ILS
UKansasState	10	62962	BnB, SA, ILS
UMissouri	106	132709	SA, ILS
ulysses16	16	6859	BnB, SA, ILS

file and a trace output file. Before writing any results to the trace output file we check if the time does not exceed the cut-off time of 600 seconds. For calculating the relative error of each algorithm, we need the optimal cost. Since we only have the optimal cost of Cincinnati and Boston, we set the optimal cost of the remaining instances as the minimum tour length obtained from all runs. The optimal cost of Atlanta, UKansasState, and ulysses16 are accurate, since they are calculated by BnB algorithm before the program reached its time limit (600 seconds). The optimal values from all instances are shown in Table 1.

The solution quality, corresponding timestamp, and relative error of all algorithms are shown in Tables 2 and 3. The solution qualities for MST approximation, Cheapest Insertion, Christofides, SA, and ILS are the average of 10 runs with different random seeds. The relative errors of instances obtained by the MST approximation, Cheapest Insertion, and Christofides show the solution costs are within a factor of 2, 2, and $\frac{3}{2}$ of the optimal tour length, respectively. Run times of MST approximation, Cheapest Insertion and Christofides are less than a few seconds. However, the relative errors produce by MST approximation are the worst comparing to other algorithms. For instances with more than 100 locations, Christofides outperforms BnB in term of solution quality and runtime. The relative errors of solutions generated by Christofides in most cases are slightly smaller than Cheapest Insertion.

The BnB algorithm finishes quickly on the small instances, such as Atlanta, Cincinnati, UKansasState, and ulysses16. For the instances with a size greater than 20, the exact solution is not found within the time limit (600 seconds). The relative errors of BnB method across all instances are within the range of 0 and 0.1893. The relative error of SA is between 0 and 0.0527. The relative errors of ILS across all instances are between 0 and 0.0104. Qualified Run-Time Distributions (QRTDs) and Solution Quality Distributions (SQDs) plots for three instances with different sizes are generated for local search algorithms. The x-axis is the run-time and the y-axis is the fraction of runs that have solved the problem. The plots for a given algorithm and instance obtained from the results of running the program with 50 different random seeds. The relative solution qualities (or relative errors) are expressed in percent.

Atlanta (20 locations) is the first instance selected for creating the QRTDs and SQDs plots. Since the sample size is relatively small, a short time span on x-axis is set for comparing various relative solution qualities. Figure 1 shows at time 0.01 second, all runs achieve the quality of 5% and about 20% of the runs achieve the quality of 0.2%. The SQDs plot in Figure 2 shows about 20% of the runs, at about 0.002 seconds produce the optimal solution. Similar plots are produced in Figures 3 and 4 for Denver that has 83 locations. We consider the time-line of 50 seconds for comparing various relative errors. More than 60% of runs achieved the relative error of 0.8% in less than 50 seconds while only 20% of runs achieved the relative error of 0.4% in less than 50 seconds for Denver. As evident in Figure 4, each run takes about 40 seconds to achieve its optimal solution.

Finally, Figures 5 and 6 show QRTDs and SQDs plots for Roanoke, which has 230 locations. As the plot in Figure 5 shows, there are periods that the percentage of solved problems (with respect to assigned quality) does not increase when running time grows. This is likely related to the fact that the algorithm is stuck in the local optimum and it takes a while to find a better candidate. As evident in Figure 6, after 0.06 seconds of running the algorithm, about 50% of runs achieve the quality of 0.4%. We can also conclude at 600 seconds only about 20% of runs achieve the quality of 0.05%.

The QRTDs and SQDs plots of ILS for Atlanta are shown in Figures 7 and 8. As expected, when we consider a certain percentage of solved problem in QRTDs plot (e.g. 4%), for a better relative solution quality (or smaller relative error), the algorithm needs a longer period of time. Next, we create the plots for results from running ILS program on Denver instance. Figure 9 shows most of the runs reach the relative quality of 2% quickly. However, only about 60% of all runs achieve quality of 0.1%. As evident in Figure 10, when the program runs for 0.2 seconds, about 70% of the solutions reach the optimal value. Finally, Figures 11 and 12 show QRTDs and SQDs plots for Roanoke. When comparing the results from Roanoke to Atlanta and Denver, the program needs a longer time to produce a solution with relative error of 4%. Thus, QRTD's x-axis is set to 600 seconds. At 600 seconds, only about 10% of total runs reach the quality of 0.2%. The SQDs plot for Roanoke shows when the run-time is limited to 50 seconds the best relative solution quality or error is greater than 0.75%.

We also created box plots for running times of SA and ILS using 50 runs with different random seeds. Specifically, we consider certain relative solution quality (relative errors) and compare the running time statistics for a given instance and algorithm. We use the online box plot generator for generating the plot [?]. The box plots generated from the results of running SA and ILS on Denver are depicted in Figures 13 and 14, respectively. The plot in Figure 13 shows the median run-time to achieve a relative solution quality of 8% using SA method is about 28 seconds. For the same relative solution quality (8%), ILS requires less than 3 seconds. To generate a higher relative solution quality (2%), the program based on ILS

Table 2: Algorithm Evaluation for Constructive Heuristic Methods

	MST Approximation			Cheapest Insertion			Christofides		
Dataset	Time(s)	Sol.Qual.	Rel.Err.	Time(s)	Sol.Qual.	Rel.Err.	Time(s)	Sol.Qual.	Rel.Err.
Atlanta	1.082E-03	2459676.90	0.2275	4.364E-04	2114856.20	0.0554	2.270E-03	2123111.10	0.0596
Berlin	6.715E-03	9927.90	0.3163	3.738E-03	8908.50	0.1812	2.169E-02	8539.40	0.1322
Boston	4.006E-03	1095447.50	0.2260	2.003E-03	950425.60	0.0637	1.0366E-02	974164.50	0.0902
Champaign	7.720E-03	66542.90	0.2640	4.533E-03	57102.60	0.0847	1.6848E-02	55223.50	0.0490
Cincinnati	3.679E-04	305984.70	0.1009	1.626E-04	283226.70	0.0190	1.280E-03	283436	0.0197
Denver	1.592E-02	129618.90	0.2906	1.299E-02	116739.90	0.1624	6.053E-02	111932.70	0.1145
NYC	1.149E-02	1911007.90	0.2289	7.560E-03	1755886.40	0.1291	7.290E-02	1755131.40	0.1287
Philadelphia	2.529E-03	1700623.50	0.2182	1.055E-03	1473830.30	0.0558	8.629E-03	1494476.50	0.0706
Roanoke	8.383E-02	802790.80	0.2204	2.370E-01	755106.20	0.1479	1.015E+00	711036.20	0.0809
SanFrancisco	1.774E-02	1083074.30	0.3368	2.083E-02	1005967.30	0.2416	1.020E-01	935375.70	0.1545
Toronto	2.732E-02	1650492.40	0.4033	2.756E-02	1413694.70	0.2020	1.309E-01	1347563.10	0.1457
UKansasState	3.058E-04	69749.80	0.1078	1.632E-04	64312.20	0.0214	1.106E-03	66585.00	0.0575
UMissouri	2.346E-02	170407.30	0.2841	2.535E-02	153716.80	0.1583	1.077E-01	149054.60	0.1232
ulysses16	4.513E-03	7979.30	0.1633	2.869E-03	7393.40	0.0779	4.080E-03	6997.70	0.0202

Table 3: Algorithm Evaluation for Branch-and-Bound and Local Search

	Branch-and-Bound			Simulated Annealing			Iterated Local Search		
Dataset	Time(s)	Sol.Qual.	Rel.Err.	Time(s)	Sol.Qual.	Rel.Err.	Time(s)	Sol.Qual.	Rel.Err.
Atlanta	2.527E+02	2003763	0.0000	4.108E+00	2018820.7	0.0075	7.897E-02	2005764.60	0.0010
Berlin	2.822E+01	8250	0.0939	4.3671E+01	7542.00	0.0000	1.812E+01	7568.80	0.0036
Boston	1.561E+02	963667	0.0785	7.557E+00	893665.70	0.0001	4.143E+00	893668.20	0.0001
Champaign	1.038E+02	58797	0.1169	3.370E+01	52644.40	0.0000	1.353E+01	52700.60	0.0011
Cincinnati	6.529E-02	277952	0.0000	2.928E+00	279555.70	0.0058	2.809E-03	277952.00	0.0000
Denver	1.456E+01	109681	0.0921	4.197E+01	100930.80	0.0050	1.392E+02	100487.20	0.0006
NYC	2.525E+00	1816015	0.1678	1.388E+01	1608819.50	0.0346	1.672E+01	1555060.00	0.0000
Philadelphia	2.863E+02	1458269	0.0446	4.906E+00	1414343.80	0.0132	2.013E-01	1395981.00	0.0000
Roanoke	5.117E+02	787054	0.1893	1.115E+02	721762.70	0.0973	5.023E+02	664598.60	0.0104
SanFrancisco	1.708E+00	896718	0.1068	3.174E+01	869408.40	0.0731	2.686E+02	810196.00	0.0000
Toronto	5.950E+01	1358974	0.1554	4.263E+01	1238141.80	0.0527	1.798E+02	1176404.50	0.0002
UKansasState	4.198E-02	62962	0.0000	2.858E+00	62962.00	0.0000	7.238E-04	62962.00	0.0000
UMissouri	1.815E+02	157181	0.1844	4.970E+01	133689.80	0.0074	3.624E+02	132861.00	0.0011
ulysses16	3.698E+01	6859	0.0000	1.679E+01	6859.00	0.0000	3.709E-02	6859.00	0.0000

needs to run from 2 to 20 seconds. Next, we create the box plots for Roanoke using the results collected from SA and ILS. SA running times in Figure 15 show the runtimes are not distributed evenly, in other words, about half of the runs converge to the specific relative solution quality quickly and remaining takes a much longer time. This could be related to the fact that we initialize the solution using Nearest Neighbor method and in some cases the initial solutions are already close to specified relative solution quality. Figure 16 shows the box plot for the same instance and qualities using ILS. For all selected relative solution quality, ILS takes 80 to 240 seconds. The data points representing ILS's run time values (Figure 16) are distributed much more evenly in comparison to SA. The median of running times in both plots increase as the error rate decreases.

The distribution of final timestamps for Atlanta, Denver, and Roanoke

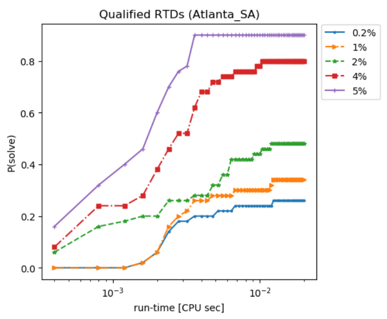

Figure 1: QRTDs plot for Atlanta using SA Results

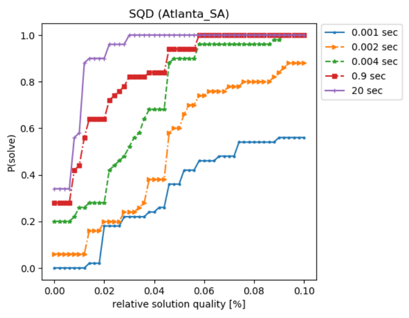

Figure 2: SQDs plot for Atlanta using SA Results

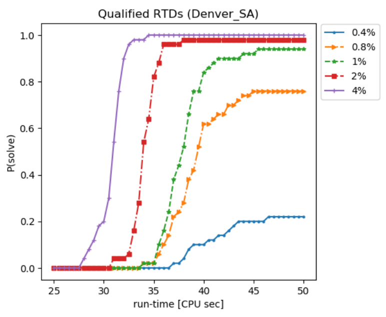

Figure 3: QRTDs plot for Denver using SA Results

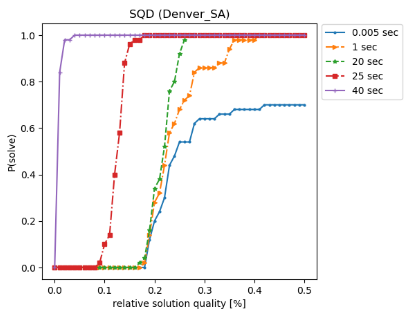

Figure 4: SQDs plot for Denver using SA Results

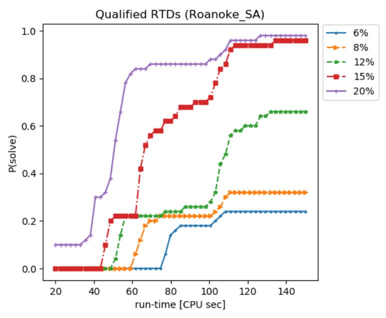

Figure 5: QRTDs plot for Roanoke using SA Results

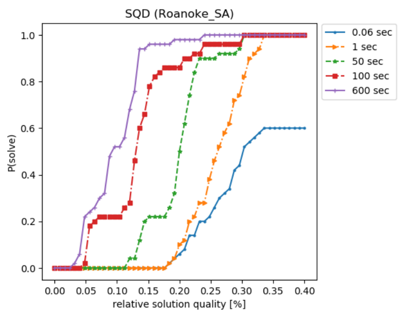

Figure 6: SQDs plot for Roanoke using SA Results

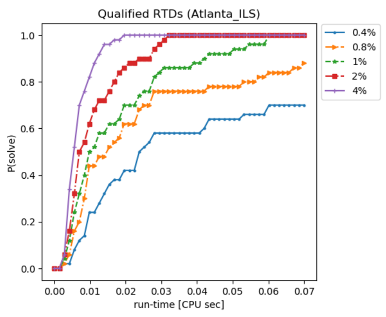

Figure 7: QRTDs plot for Atlanta - ILS

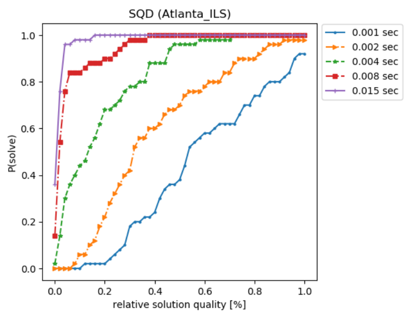

Figure 8: SQDs plot for Atlanta - ILS

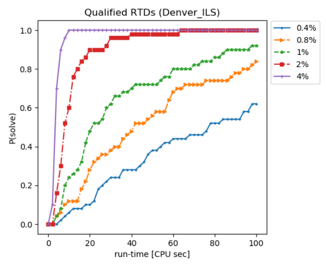

Figure 9: QRTDs plot for Denver -ILS

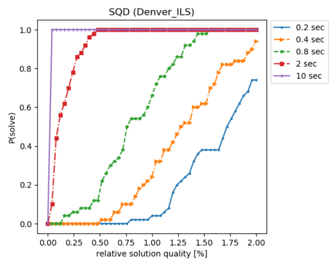

Figure 10: SQDs plot for Denver - ILS

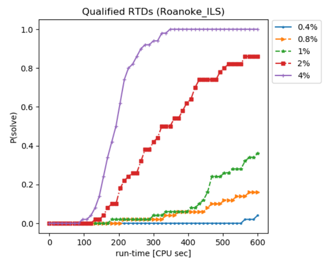

Figure 11: QRTDs plot for Roanoke-ILS

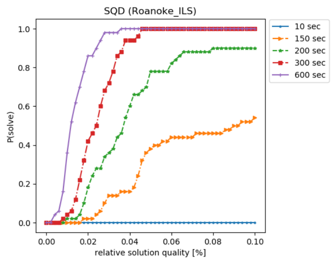

Figure 12: SQDs plot for Roanoke-ILS

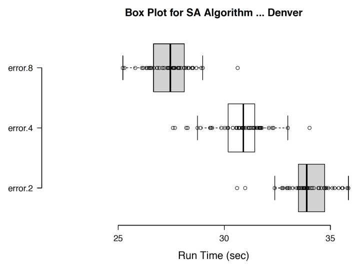

Figure 13: Box Plot for SA's Running Times - Denver

using SA and ILS are depicted in Figures 17 and 18. The median of run-time values for Atlanta, Denver, and Roanoke using SA method are 0.03, 41.37, and 111.48 seconds, respectively. The median of run-time for Atlanta, Denver, and Roanoke using ILS method are 0.02, 104.94, and 519.57 seconds, respectively.

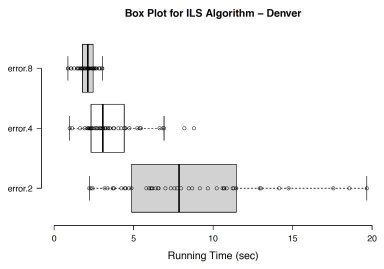

Figure 14: Box Plot for ILS's Running Times - Denver

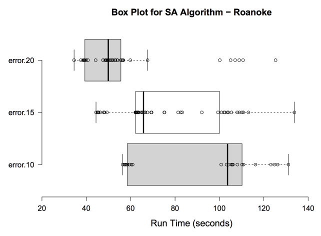

Figure 15: Box Plot for SA's Running Times - Roanoke

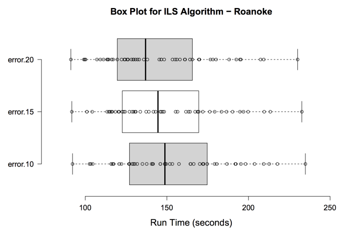

Figure 16: Box Plot for ILS's Running Times - Roanoke

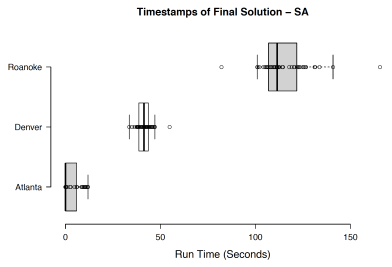

Figure 17: Box Plot for Simulated Annealing

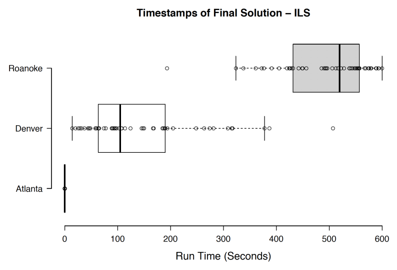

Figure 18: Box Plot for Iterated Local Search

6 DISCUSSION

Comparative performance evaluation of algorithms and expected time complexity are presented in this section. As expected, the constructive heuristic algorithms, such as MST approximation, Cheapest Insertion, and Christofides are very fast and they terminate quickly. This is expected since these algorithms have a polynomial time complexity. The empirical evaluation shows, under BnB method, as instance size increases the running time grows significantly. For a larger instance (greater than 30), we run the program for more than 24 hours but they only generate a partial solution. This confirms the theoretical analysis of BnB algorithm.

We assume at the time of running, there is no prior knowledge of optimal solution. Thus, the local search algorithms (such as SA and ILS) are running for the whole period of 600 seconds. For comparison, we choose the timestamp of best (or latest) solution for each instance. The results show for the smaller instance, such as Cincinnati, both SA and ILS take less than a few seconds to generate the best solution. However, for the larger instances (such as Roanoke), SA and ILS generate their best solutions at about 111 and 502 seconds, respectively.

Another performance measure in this study is the solution quality. The results show the best solution qualities on all instances are obtained from ILS algorithm. BnB and SA could obtain such quality on about 29% and 70% of instances, respectively. BnB only reach the best quality for sample with less than 20 locations. For the larger sample, the result is significantly worse than local search algorithms. MST approximation have the worst and best performance in terms of the solution quality and run time, respectively. Christofides and MST approximation have approximation factors of no worse than $\frac{3}{2}$ and 2 as expected. On average, both Christofides and Cheapest Insertion method are slightly better than MST approximation in terms of solution quality.

The run-times, relative errors, QRTDs, SQDs, and box plots obtained from program based on local search algorithm help to analyze the behavior of implemented algorithms further. For example, the QRTDs for running SA shows for a period of time the program does not generate any better solutions and suddenly it starts to improve. This could be related to a period that the program got stuck in local optimum for a while and a perturbation mechanism implemented in the program helps to exit the local optimum. Additionally, the box plots show running times from SA method are not distributed evenly. Specifically, about half of the runs converge to the desired quality in a short time and the remaining runs require a much longer time. This may be related to the way we initialized the SA algorithm with a solution obtained from nearest neighbor method (with a random initial node), while we initialize the ILS with a totally random solution.

7 CONCLUSION

We study and analyze six algorithms from the categories of exact algorithm, constructive heuristic, and local search approaches. The results show the program based on BnB approach is beneficial for the scenarios the instance size is small and the program terminates before the cut-off time. For the larger instances, the running times grow larger than exponential and we only have a sub-optimal solution. In such cases, the algorithm based on local search methods are more beneficial. ILS based on 2-opt method and double bridge reach the best performance for all instances. Finally, the constructive heuristic algorithms with approximation guarantees are desirable when we need to have a solution with a good enough quality in a very short time. We can also use constructive heuristic algorithms for initializing and restarting the local search process.

There are multiple ways to improve the current algorithms. For BnB approach, we could adopt a better method for calculating the higher and lower bound to accelerate the pruning process. Hybrid search strategy, such as using DFS in the first iteration and Best First Search in later iterations, could help to improve the pruning. We could also design a parallel BnB that allows concurrent execution. For local search algorithms, experimenting with various combination of strategies are required to improve the performance. For example, we can try various ways to set the initial solution and parameters, define new perturbation mechanisms, and use tabu memories to enhance and accelerate the search process.

ACKNOWLEDGMENTS

The author would like to thank Dr. Catalyurek and teaching assistant team for their guidance in this study.