CSE 6140 Project: Traveling Salesman Problem

Ruomeng Xu
rxu86@gatech.edu

Qinlan Zhang
qzhang342@gatech.edu

Chongwu Guo
cguo48@gatech.edu

Ankita Jain
ajain397@gatech.edu

ABSTRACT

In this paper, we implement various algorithms to solve the Traveling Salesman Problem, which is one of the oldest and most fundamental problems in computer science. First we have an introduction to this problem in the paper and after that there is a formal definition of the problem, literature survey of the related work and then several algorithms are provided to obtain the optimal solution for this problem. These algorithms include Branch and Bound, MST Approximation, Farthest Insertion, Simulated Annealing and Hill Climbing algorithm. Running time and relative errors for each algorithm are compared by means of comprehensive table, QRTDs, SQDs and box plots in the discussion and conclusion part.

Keywords

Algorithm; Traveling Salesman Problem; Heuristic; Branch and Bound; Local Search

1. INTRODUCTION

In this project, we are given the Pokemon location list of several cities and our goal is to find the shortest tour so that we can visit all the Pokemon locations in the city and return to where we start as quickly as possible.

We are implementing this project on the basis of the Traveling Salesman Problem(TSP) by five algorithms, including Branch and Bound, MST Approximation, Farthest Insertion, Simulated Annealing and Hill Climbing Algorithms. We have obtained the running time and the optimal tour of all these algorithms and the performance of each algorithm is evaluated by comprehensive tables, QRTDs, SQDs, and box plots. From the evaluation results, we find that Branch and Bound has good performance for small instances while with the increasing of the size of the instance, the running time becomes rather long and thus when we apply a cutoff time to it, the accuracy of the results will be affected. Heuristic Algorithms such as MST Approximation and Farthest Insertion are fast but have larger relative error. Local search algorithms such as Simulated Annealing and Hill Climbing seem to have the best performance in terms of both the running time and accuracy.

This paper is organized as follows: Section 2 is about the problem definition; Section 3 contains a literature review of previous work; Section 4 describes the implementation and pseudo code of each algorithm; Section 5 describes the empirical evaluation results; Sections 6 and 7 are discussion and conclusions respectively.

2. PROBLEM DEFINITION

In this project, we are implementing different algorithms to find the shortest tour so that we can visit all the Pokemon locations in the city and return to where we start as quickly as possible.

This is actually a Traveling Salesman Problem(TSP). And the TSP problem can be defined as follows: given the x-y coordinates of N points on a 2D plane (i.e. vertices), and a cost function c(u, v) defined for every pair of points (i.e. edge), find the shortest simple cycle that visits all N points. Here, we use the Euclidean distance between two points u and v to define the cost function c(u,v). Furthermore, we need to pay attention that this version of the TSP problem is metric: all edge costs are symmetric and satisfy the triangle inequality.

The algorithms we use include: Branch and Bound, MST Approximate, Farthest Insertion, Simulated Annealing and Hill Climbing algorithms. And empirical analysis of algorithm performance on different datasets will be conducted in terms of comprehensive table, QRTDs, SQDs and box plots to find out the trade-offs between accuracy and speed for each algorithm.

3. RELATED WORK

The traveling salesman problem is to minimize the distance of complete tour. Traditionally, there are mainly two ways to "solving" any TSP instance. The first is the exact algorithm, such as Branch and Bound, branch and cut, Held-Karp algorithm. It will generate a tour that is exactly the optimal tour. The other one is heuristic and approximation algorithm, which will quickly yield a relative good solution in a reasonable computational effort but do not guarantee an optimal solution. Tour construction approaches are typical heuristic and approximation algorithms. Closest neighbor heuristic, greedy heuristic, insertion heuristic and Christofide heuristic are common tour construction algorithms, which guarantee to yield a relative good solution within 10%-15%[1]. As for tour improvement algorithms,

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

they mainly focus on improving the tour after generating the tour by using tour construction algorithm. K-opt is a well-known algorithm for tour improvement. But the computational time should be considered when choosing k>3. Link-Kernighan heuristic is a variable k-way exchange heuristic algorithm investigated by Lin & Kernighan. Some other algorithm, such as Tabu search, Simulated annealing, Genetic algorithm, Ant colony optimization, have been successfully applied and adapted to yield fair enough approximate solutions for TSP instances [2].

In recent years, a lot of advanced methods have been come up with for solving the TSP. Coelho (2008)[3] presented a discrete differential evolution with local search method for TSP. In Marinakis and Marinaki (2010)[4], a hybrid algorithm was formulated by combing GA algorithm and particle swarm optimization algorithm.

3.1 Branch and Bound

The branch and bound technique can be viewed as a generalization of backtracking for optimization problems. We grow a tree of partial solutions that represents the search space. At each node of the recursion tree, we check whether the current partial solution can be extended to a solution which is better than the best solution found so far, otherwise, we will terminate this branch immediately. One of the first branch and bound algorithms was proposed by Little et al. (1963)[5]; it is based on reducing rows and columns of the cost matrix of an instance and calculating the lower bound of each solution branch. If a particular branch has a lower bound that is higher than an already obtained solution (upper bound), that branch will be pruned.

3.2 Heuristic Approaches

As we all know TSP is an NP-hard problem, thus heuristic approach can be implemented to solve the TSP problem within an acceptable running time and accuracy. The accuracy of heuristic approaches is guaranteed by a worst-case performance. Besides the MST approximation and Farthest Insertion algorithm, which will be implemented in our project, there are also other famous heuristic approaches for the TSP problem. One approach is improved on the basis of MST Approximation, proposed by Christofides (1976), which derives a Eulerian graph by linking its odd-degree vertices by minimum-cost matching[6]. Another one is the nearest-neighbour algorithm, whose time complexity is $O(n^2)$[7].

3.3 Local Search Approaches

Local Search, also referred as Neighborhood Search or Hill Climbing, is the basis of many heuristic methods for combination optimization problem. It is a simple iterative method for finding the good approximate solution.

In the article "Hill-climbing Search" by Bart Selman and Carla P Gomes, the authors explain the differences of local search versus global search, and give details of local search strategies. And one conclusion authors made is that for many optimization problems, such as the traveling salesperson problem, using hill-climbing to produce a local optimum, such local optimum is quite acceptable since it often is a reasonable approximation of the global optimum value[8].

However, local minimum is the main problem with local search. That is why Guided Local Search[9] is needed. Simulated Annealing is a kind of guided local search that can approximate the global optimum by accepting worse temporary solution with acceptance probabilities[10].

4. ALGORITHM

4.1 Branch and Bound

4.1.1 Algorithm Description

Our branch and bound algorithm uses reduced cost matrix in order to define a lower bound of each of the solution branch. Each row or column of a cost matrix is reduced by finding the smallest entry in a row or column and subtracting it from every other element of that row or column. This reduced cost matrix gives the additional cost of including an edge in the tour relative to a lower bound. Thus, the lower bound can be computed by taking the sum of the cheapest way to leave and enter each node. This will ensure that any tour must leave and enter each node exactly once.[11]

The main steps of the algorithm are as follows:

1. Set an initial upper bound as infinity. A tighter upper bound can also be set through any of the heuristic or local search approach.

2. We maintain a priority queue to keep track of the unexplored solutions. Each tuple of the priority queue will have information about the number of unvisited nodes, lower bound associated to the current path and the current path itself. Tracking the number of unvisited nodes will help in exploring that path in a depth-first manner. If the number of unvisited nodes remain same then it will select the solution with the minimum lower bound using best-first.

3. Pop a given unexplored solution from the priority queue

   - If the popped solution is complete, then check whether the total cost of the tour is less than the upper bound. If it is, then update the upper bound cost and tour with this solution.
   - If the popped solution is partial and its lower bound cost is less than the upper bound, we will further explore this branch. Through row-column reduction, we will compute the lower bound of each of the sub-problems that contains the current path and a new unvisited node. And push these into the priority queue.

4. Repeat step 3 until priority queue becomes empty or total execution time exceeds the cutoff time.

4.1.2 Pseudo Code

4.1.3 Time and Space Complexities

The tree can have at most $(n-1)!$ branches which is $\Omega(2^n)$. Moreover, at any given time, priority queue can have at most $O(n^2)$ elements, thus running the loop $O(n^2)$ times for each branch. Hence, overall time complexity is at least exponential. We are storing graph information in adjacency matrix, and priority queue can store at most $O(n^2)$ elements. Storing valid tours take $O(n)$ space. Thus overall space required will be $O(n^2)$.

Algorithm 1: Branch And Bound

Input : A file name corresponding to the instance, cutoff time and a seed value.
Output: Best tour of TSP and corresponding length until cutoff time by Branch and Bound

```
function BnB (filename, cutoff, seed);
Generate an adjacency matrix A;
N =number of nodes;
priorityQueue ← Enqueue(N − 1, 0, [0]);
upperBound ← ∞;
while priorityQueue not empty and time<cutoff do
    remNodes,curCost,curPath = priorityQueue.pop();
    if len(curPath)<N and curCost<upperBound then
        for node not in curPath do
            tempPath = curPath + node;
            lowerBound←Compute lower bound through
             row-col reduction on A;
            priorityQueue ← Enqueue(N −
             len(tempPath), lowerBound, tempPath);
        end
    end
    else if len(curPath)==N and curCost<upperBound
     then
        upperBound ← curCost;
        pathUB ← curPath;
    end
end
return upperBound, pathUB ;
```

Algorithm 2: MST Approx Algorithm

Input : A file containing all the location information of the city
Output: Optimal tour of TSP and corresponding length by MSTApprox

```
function runMSTApprox (filename);
Generate a Graph G containing all the edges between
 any two locations in file;
Find the MST of G ;
Double all the edges in MST to generate M';
n=number of nodes in G;
i=0;
OPTcost=+∞;
while i<n do
    Select the node i as the root node;
    Find a Eulerian tour T by applying depth-first
     search starting from root node on M';
    Remove the duplicate nodes from T according to
     the order of nodes in T to form path P;
    Add the root node to the end of P to form the tour
     solution of TSP, denoted by T';
    Calculate the total length of tour T' as cost(T');
    if cost(T')<OPTcost then
        OPTtour=T' ;
        OPTcost=cost(T');
    end
    i=i+1;
end
return OPTtour,OPTcost;
```

4.2 MST Approximation Algorithm

4.2.1 Algorithm Description

In our MST Approximation Algorithm, we want to find the approximate optimal TSP solution based on the MST of the graph and the triangle inequality.
The steps of the algorithm are as follows:

1. Compute the MST of the Graph G by Prim's algorithm, which is denoted as M
2. Double all the edges in MST to construct M'
3. Choose a root node and apply depth-first search on M' to get the Eulerian tour
4. Only add a vertex as part of the TSP tour the first time it is encountered to generate the path P
5. Generate the partial solution of TSP by adding the root node to the path P
6. Iterate through all the nodes in Graph G as the root node for depth-first search and choose the partial solution with the minimum length as the final solution of TSP by MST Approximation algorithm

4.2.2 PseudoCode

4.2.3 Approximation Guarantee

The MST Approximation is a 2-Approximation algorithm, in other words the relative error of its results is no more than 100%[12].

- Explanation of Approximation ratio
  Based on the definition of MST and TSP, we can get the conclusion that the weight of MST M, denoted by w(M), is less than OPT, the weight of TSP solution T. The proof is as follows:
  - Take T and remove an edge e. T-e is now a spanning tree. And the weight of this spanning tree is w(T)
  - Since M is the MST, according to the definition of MST, w(M) $\leq$ w(T-e)=w(T)-w(e) = OPT-w(e)
  - Since OPT-w(e)<OPT, thus w(M)<OPT

  According to the triangle inequality, the third edge is always shorter than the sum of the other two edges. Thus, in most natural applications of the traveling salesman problem, direct routes are inherently shorter than indirect routes.
  When performing a depth-first traversal of a spanning tree, we will visit each edge of MST twice, one is going down the tree when exploring it, and the other one is going up after exploring the entire sub-tree. Therefore, tour has weight twice that of the minimum spanning tree, and hence at most twice optimal. However, the TSP requires that each vertex except the starting one can only be visited once, while the vertices will be repeated on this depth-first search tour. Thus, to remove the extra vertices, at each step we can take a shortcut to the next unvisited vertex. By doing so we have replaced a chain of edges by a single direct edge, which guarantees that the tour can only get shorter based

on the triangle inequality. Thus the shortcut tour is within weight twice that of optimal.

- Explanation of the selection of optimal solution by MST Approx Algorithm

  Since the root node of depth-first search has great impact on the solution obtained by the MST Approximation Algorithm, we iterate through all the nodes in the graph as the root node for depth-first search and choose the partial solution with the minimum length as the final solution of TSP by MST Approximation algorithm.

4.2.4 Time and Space Complexities

There are two main steps in finding the tour. One is finding the MST by Prim's algorithm with adjacency list and priority queue, whose time complexity is $O(|E| \log n)$, in which $|E|$ is the number of edges, n is the number of nodes. The other one is Depth First Search(DFS), whose time complexity is $O(n^2)$, which selects all the n vertices as the root node of DFS and each DFS requires O(n) time complexity. Thus, the overall time complexity of MST Approximation Algorithm is $O(n^2)$, in which n is the number of vertices. The space complexity is the space required by the Adjacent List, which is $O(n^2)$.

4.2.5 Potential Strengths and weakness

The MST Approximation can compute the results rather quickly, however its relative error might be larger than other algorithms such as local search algorithms.

4.3 Farthest Insertion

4.3.1 Algorithm Description

Another heuristic approach to solving the TSP is to extend the sub-tour of TSP by inserting the remaining vertices one by one until all vertices are in the sub-tour. It is a heuristic algorithm to construct a tour following specific rules without trying to improve upon the tour.

Farthest insertion is one of several heuristics proposed by Rosenkrantz, Stern, and Lewis in the Journal of the Society for Industrial and Applied Mathematics in 1977. It rivals the the speed of the Greedy algorithm and returns improved solutions with quite simple implementation. The main ides of Farthest Insertion is inserting a vertex whose minimum distance to any vertex in the current sub-tour is maximal

4.3.2 Algorithm Explanation

As we mentioned above, farthest insertion algorithm follows some specific rules. In this section, we introduce the details about our insertion scheme.

- Initial Tour
  Typically, we randomly choose an initial vertex as our sub-tour and start to extend our tour. That is, our initial tour is $[v_i, v_i]$, where v_i is a random vertex chosen from our data set. In our code, we choose the first vertex as the start vertex and the initial partial tour is $[v_1, v_1]$.

- Choosing Insertion Vertex
  In farthest insertion algorithm, we choose the farthest vertex from the remaining vertex as the vertex that used for the insertion in the next step. Now, let's define the farthest. Given a tsp problem with vertices V and all edges information E, suppose the sub-tour is $S = \{v_1, v_2, \ldots, v_i\}(|S| < |V|)$. If $v_k \in V - S$ has the maximum distance to all vertices in S comparing with all $v_j \in V - S$, we say v_k is the farthest vertex. The distance from a vertex to a set of vertices is the minimum value among all distance from the vertex to every vertex in the set.

- Choosing Insertion Position
  After selecting the farthest vertex v_k from the remaining set, now we need to decide how to insert v_k into our sub-tour S. We should greedily choose the position that minimizes the the length of whole tour after adding v_k into S. So the position index should be m which minimizes $cost(S[m]+v_k)+cost(v_k, S[m+1])-cost(S[m], S[m+1])$

4.3.3 Pseudocode

Algorithm 3: Farthest Insertion for TSP

```
   Data: distance_matrix
1  V=all vertex
2  //Initialize sub tour
3  S=[V[0],V[0]]
4  while |S|! = |V| + 1 do
5  |  vertex=arg max_{i∈V−S} distance(i, S)
6  |  index=arg min_m cost(S[m] + v_k) + cost(v_k, S[m + 1])−
7  |  cost(S[m], S[m + 1])
8  |  S = [S[: index], vertex, S[index + 1 :]]
9  end
10 return S
```

4.3.4 Time and Space Complexities

The Farthest insertion use a $n \times n$ matrix to keep the distance information, a list for partial tour and a list for all vertices. So the space complexity should be $O(n^2)$.

As for time complexity, the while loop while run $n - 1$ times. In the i-th while loop, the choosing insertion vertex step will take $O(i(n-i))$ and choosing insertion position will take $O(i)$. So the running time should be $\sum_{i=1}^{n-1}(O(i(n-i)) + O(i)) = O(n^3)$

4.4 Simulated Annealing

4.4.1 Algorithm Description

Sometimes, local search will be stuck in the local optimal. In such case, especially with large search points, it is ineffective to simply repeat the local search a few times. Therefore, the simulated annealing method is introduced. Instead of being greedy and always taking the best solution, we try occasionally choosing worse solution, which may lead to a better final solution. Specifically, simulated annealing is a meta-heuristic to approximate global optimization in a large search space. Like annealing in metallurgy, it is more likely to choose a worse solution with higher temperature. However, when the temperature cools down, it is less likely to choose a worse solution.

4.4.2 Algorithm Explanation

In order to formulate the appropriate algorithm, there are several elements that we should figure out.

- Neighbor.
  We decide to use 2-opt exchange. For more details, please check the pseudo code of 2-opt exchange. In order to decrease the running time, we also use the length of tour T as an input in real code. So we can compute the length of new tsp tour by adding 2 new edges and deleting two discarded edges.

- Cooling Schedule.
  To achieve a better decision process, a cooling schedule must be specified. Here, we choose exponential multiplicative cooling schedule. That is, $T = \alpha^k T$ $\alpha \in (0.8, 0.9999)$. The specific parameters should be empirically adjusted for each problem. In our program, we set initial temperature $T = 1000000$ and $\alpha = 0.9999$. Also, in the process, we only update T when we choose to move to the neighborhood. Otherwise, the temperature will decrease too quickly to find a global optimal. For more details, please check the pseudocode of simulated annealing.

- Accepting Probability
  In simulated annealing, we always choose a good neighbor if we get into a good neighbor. However, we accept a bad neighbor with the probability p. And we define the probability based on the difference of length between the current tour and a random neighbor tour. For more details, please check the pseudocode of probability.

- Best Solution
  In simulated annealing, we have the chance to accept a worse solution. The worse solution may or may not lead to a better final solution than current solution. In case of missing a good solution in the searching process, it is necessary to record the temporary best solution in the process. That is why we need the trace file for the algorithm.

- Stop Criteria

  – Cutoff time.
    It will stop the algorithm after running it for a certain time.

  – Minimum temperature.
    When the temperature is too low, the probability of accepting worse solution equals to 0. It is meaningless to iterate any more when it cools down.

4.4.3 PseudoCode

Algorithm 4: Simulated Annealing for TSP

```
Data: distance_matrix, initial_T, α, max_iter_num,
      min_T, seed, cutoff
Set begin_time=time.time(), diff_time=0, T=initial_T
set random seed as given seed
generate initial TSP tour tsp_current with seed
score_current=score(tsp_current)
while T>min_T and diff_time <cutoff do
    generate two random index for 2-opt exchange
    tsp_neighbor,score_neighbor
    =neighbor_2opt(tsp_current,index)
    p=pro(score_current,score_neighbor,T)
    if random.random()<p then
        current_time=time.time()
        tsp_current=tsp_neighbor
        score_current=score_neighbor
        T=α×T
        Add the update into trace if improved
        if tour is improved then
            update the trace file
        end
    end
    diff_time=time.time()-begin_time
end
return tsp_current,score_tsp,trace
```

Algorithm 5: 2-opt Exchange

```
Given tour T, exchange index i, j(i < j)
T_new[0 : i − 1] = T[0 : i − 1]
T_new[i : j] = T[j : i]
T_new[j + 1 : end] = T[j + 1 : end]
return T_new
```

Algorithm 6: Probability

```
Given score_current, score_neighbor, T
if score_current > score_neighbor then
    return 1
else
    return exp((score_current-score_neighbor)/T)
end
```

4.4.4 Time and Space Complexity

Since SA is a meta-heuristic, a lot of choices are required to turn it into an actual algorithm. There is a clear trade-off between the quality of the solutions and complexity. The time and space complexity cannot be determined only by the size of graph. We should also take simulated annealing schedule, cutoff time, random seed and the distance information into consideration..And there is no doubt that the

running time will grow as the size of instance becomes bigger. In the following evaluation section, we will find that the big difference in running in the same instance with different random seed. In all, Simulated Annealing sacrifices running time for relative good solution.

4.5 Hill Climbing Algorithm

4.5.1 Algorithm Description

Hill climbing is a local search algorithm that given an initial arbitrary solution, it attempts to find a better solution by transforming the solution to one of its neighbors which optimize the objective function most significantly. However, since the algorithm is incrementally getting better, it may get stuck in the local optimum where its local neighbors in all directions are worse than itself. We can mitigate this problem by running a number of times of this algorithm with different initial solution constructions and pick the best solution or the average solution. The strength of hill climbing algorithm is that since it converges in the same direction, its running time will be theoretically faster than other global search algorithms
The steps:

1. generate a random tours bestTours as our initial construction
2. go through all the possible pair of edges and calculate the best pair with maximum improvement
3. transform our bestTours to its best neighbors we found
4. repeat last two steps until we cannot find a better neighbor for current solution
5. return the bestTours and its cost

4.5.2 Algorithm Explanation

- The initial tours:
  a random permutation of the cities, where the last element and the first element is the same in order to create a cycle.
- Neighbors:
  We use 2-opt as the metric of selecting local neighborhood. In details, we find the best pair of edges $e_1 = (i, i+1)$and $e_2 = (j, j+1)$ such that replacing them with new edge $e_3 = (i, j)$ and $e_4 = (i+1, j+1)$ will minimize our tour length.
- Stop Criteria:
  We stop after we cannot find a better neighbor of current solution, i.e., no pair of edges we can swap to minimize our tour length. We can also stop after we reach a cutoff time.

4.5.3 Complexity

- Guarantee:
  According to David Johnson and Lyle McGeoch, the guarantee for this algorithm is $(1/4)\sqrt{N}$
- Time complexity:
  According to David Johnson and Lyle McGeoch, consider only 2-Opt moves that remove crossings, the algorithm will perform at most $\Theta(N^3)$ moves for any 2-dimensional instance, since we need $\Theta(n^2)$ to evaluate a move $\Theta(n)$ to perform a move[13].

4.5.4 PseudoCode

Algorithm 7: Hill Climbing

```
Data: Array C of node list,Matrix dmat[][]: the
      distance matrix between all two nodes
Generate random tours bestTours
bestCost = calculateCost(bestTours)
while True do
    minchange = 0
    for i = 0 to len(bestTours)-2 do
        for j = i+2 to len(bestTours) do
            change = dmat[bestTours[i]][ bestTours [j]]
              + dmat[bestTours[i+1]][ bestTours [j+1]] -
            dmat[bestTours[i]][ bestTours [i+1]] -
              dmat[bestTours[j]][ bestTours [j+1]]
            if minchange > change then
                minchange = change
                min_i = i
                min_j = j
            end
        end
    end
    bestTours = twoOptSwap(bestTours,min_i,min_j)
    //swap nodes and rearrange the tours
    bestCost += minchange
    if minchange ≥ 0 then
        //no further improvement found
        return bestCost, bestTours
    end
end
```

4.5.5 Further Exploration

We explore a different way to have our initial tour, instead of randomly creation, we use greedy approach to create our initial tour, in which we start with node 0, and select the next unused node such that the distance between two nodes is shortest until we use all nodes. Running Hill Climbing with this greedy initial tour creation, we found that 6 out of 12 dataset have better result with 1% to 5% improvement, but the other 6 have 1% to 3% poorer result. But by this new approach, the running times for all dataset decrease about 50%, which is interesting. This approach will lose the randomness of Hill Climbing algorithm, which means all the seeds produce some result.

5. EMPIRICAL EVALUATION

5.1 Platform Description

- OS: macOS Sierra(Version 10.12.1)
- CPU: 2.7 GHz Intel Core i5
- RAM: 8 GB 1867 MHz DDR3
- Language: Python 2.7

5.2 Comprehensive Table

The table below shows that running time, length of best tour and relative error for every algorithm we implement. The time of Branch and Bound and Simulated Annealing

in the above table is the earliest time that the best solution that can be found in the run. That is, we take the last row of trace as our data. So the total running time of the algorithm must be longer than list above. Because we have time cutoff and temperature (only applied in Simulated Annealing) as our stop criteria.

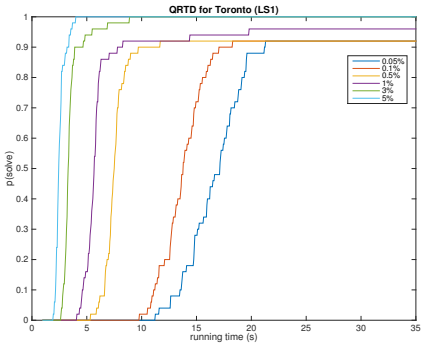
As for the comparison and discussion, it will be in the discussion section.

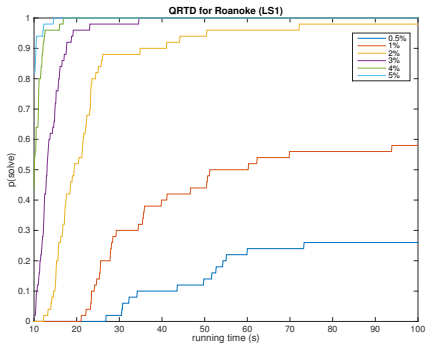
5.3 Evaluation of Local Search Algorithm

5.3.1 *Plot for Simulated Annealing*

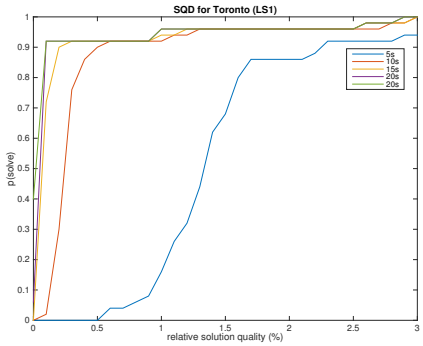
For simulated annealing, we randomly take 50 seeds with 5 minutes cutoff time and extract the average result. For most instances, we find some seeds from our random seeds list that can generate a tour that is exactly optimal tour for TSP. But we can see the big difference in the length when we choose difference seeds.

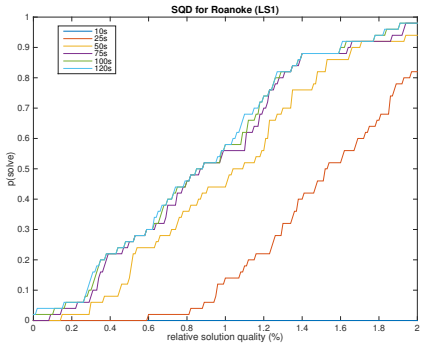
As we mentioned above, with trace files generated by 50 random seeds, we have Qualified Run-times for various solution qualities. The following two pictures are QRTDs plot for Roanoke and Toronto.





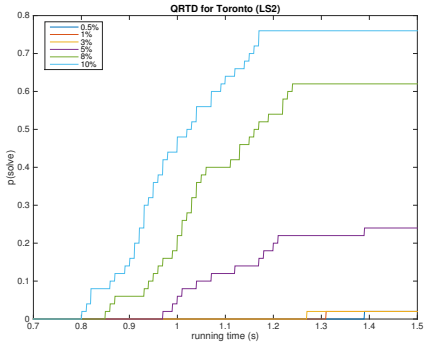
Following the instruction from lecture, we have Solution Quality Distributions for various run-times (SQDs) plots for Toronto and Roanoke.

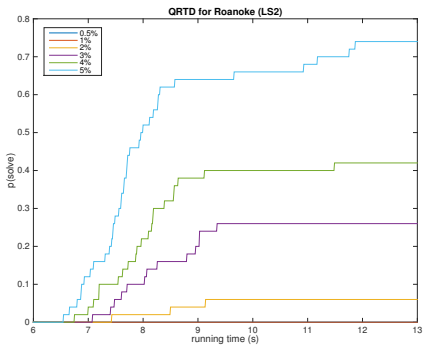




5.3.2 *Plot for Hill Climbing Algorithm*

Same as what we do in Simulated Annealing, we have QRTDs for Toronto and Roanoke.

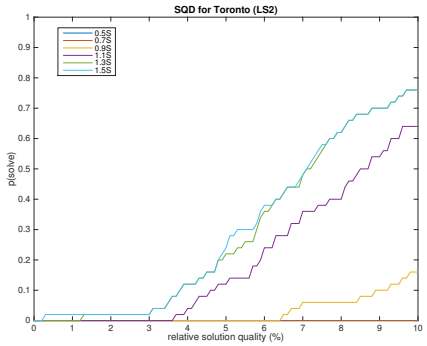


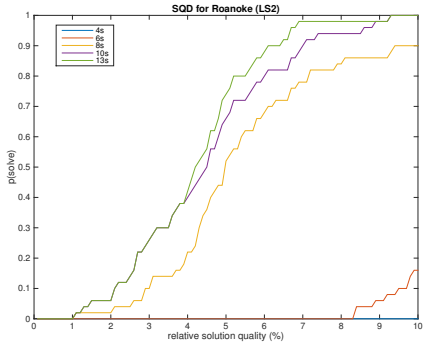


The following two graphs are SQDs for Toronto and Roanoke.

Table 1: Comprehensive Table															
	BnB, Cutoff=10 minutes			MST Approx			Fartherst Insertion			Simulated Annealing			Hill Climbing		
Dataset	Time(s)	Length	RelErr	Time(s)	Length	RelErr	Time(s)	Length	RelErr	Time(s)	Length	RelErr	Time(s)	Length	RelErr
Toronto	24.5	1532351	0.30	23.51	1648580	0.40	0.15	1229697	0.046	29.18	1176200	0.000050	0.81	1273208	0.083
Roanoke	455.73	897295	0.37	446.70	804126	0.23	1.04	684875	0.045	98.10	661300	0.0089	8.60	683995	0.044
Boston	4.34	1196232	0.34	0.52	1066706	0.19	0.01	956133	0.070	4.00	893650	0.00012	0.03	920807	0.031
Cincinnati	0.69	277952	0.0	0.00	297490	0.070	0.00	280496	0.0092	0.20	277952	0.0	0.00	279426	0.0053
Champaign	5.63	63155	0.20	1.67	59303	0.13	0.01	56315	0.070	6.66	52646	0.000050	0.08	54172	0.029
Denver	499.66	129469	0.29	8.11	130320	0.30	0.05	105035	0.046	11.72	100620	0.0019	0.31	106492	0.060
NYC	90.8	1740558	0.12	3.60	1847281	0.19	0.04	1614302	0.038	6.57	1568410	0.0086	0.17	1628704	0.047
Philadelphia	6.98	1773796	0.27	0.18	1617756	0.16	0.00	1455297	0.043	1.59	1395981	0.0	0.01	1419325	0.017
SanFrancisco	129.44	1049194	0.29	15.35	1045103	0.29	0.07	897797	0.11	22.51	821830	0.014	0.55	876028	0.081
UKansasState	3.28	62962	0.0	0.00	65561	0.041	0.00	62962	0.0	0.27	62962	0.0	0.00	62962	0.0
UMissouri	37.3	177437	0.34	20.07	160297	0.21	0.07	139340	0.050	28.31	132830	0.00095	0.71	141261	0.064
Atlanta	205.64	2003763	0.0	0.04	2327527	0.16	0.00	2003763	0.0	0.69	2003763	0.0	0.00	2053572	0.025

Table 1: Comprehensive Table





5.3.3 *BoxPlot for Local Search*

Since local search algorithms are randomized, there will be some variation in their running times. Here, we use Box plot to show the variation in running time.
The first Boxplot and table show the variation in running times in the instance Toronto. Here, we choose the threshold of relative error as 8%

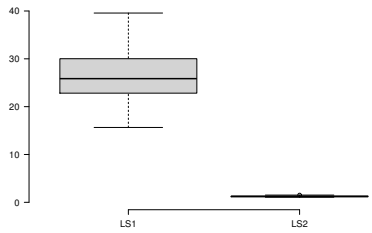


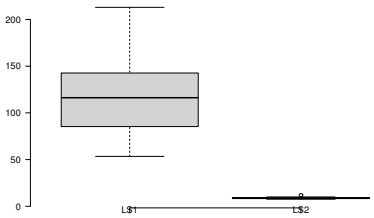
Table 3: Box plot statistics for Toronto		
Upper Whisker	39.57	1.51
3rd Quartile	30.12	1.31
Median	25.86	1.23
1st Quartile	22.82	1.16
Lower Whisker	11.56	1.07
Nr. of data point	50.00	31.00

Table 2: Box plot statistics for Toronto

Table 4: Box plot statistics for Roanoke		
Upper Whisker	212.96	9.97
3rd Quartile	142.67	9.36
Median	116.22	8.59
1st Quartile	85.34	8.29
Lower Whisker	53.41	7.60
Nr. of data point	50.00	21.00

Table 3: Box plot statistics for Roanoke

Repeating the same process, we have box plot and statistics table for Roanoke.Here, we choose the threshold of relative error as 4%



5.3.4 *Evaluation and Discussion*

Observing the QRTD plot for Toronto, after running algorithm until the last second, Simulated Annealing algorithm can achieve the relative error within 3%. But Hill Climbing only has less than 0.1 chance of getting 3% away from optimal solution. The same situation also happens to the Roanoke instance. Since the huge difference in running time

for the same instance between two algorithms, we cannot compare the performance within the same running time.

Now, let's observe the SQD plot for Roanoke. Given relative quality 3%, Simulated Annealing algorithm has 0.93 chance to achieve it within 5s whereas Hill Climbing has only 0.02 chance to achieve it within 1.55s. Since the average running time of Hill Climbing in Toronto instance is 0.81, we can say the Hill Climbing algorithm has no more than 0.1 chance to achieve 3% relative quality within 5s.

In the box plot, we control the relative solution quality within 8% for Toronto and 4% for Roanoke. We can find the different performance in running time between two algorithms. In both instances, Hill Climbing is much faster than Simulated Annealing. And the variance in running time is much bigger in Simulated Annealing than in Hill Climbing. It seems that the running time of simulated annealing is more dependent on random seeds and cutoff time.

6. DISCUSSION

In this study several approaches for solving the TSP have been implemented, and experiments were done to compare various approaches in terms of time and relative error.

6.1 Discussion of Branch and Bound Algorithm

Branch and bound works really well on small instances with zero relative error. However, for larger instances, within a cutoff duration of 10 minutes, it is not able to find the optimal solution. Thus, average relative error over 12 instances is relatively higher i.e., around 21%. I also tried setting initial upper bound through Hill Climbing that provides a much better average relative error of 2.7%. With a tighter initial upper bound, most of the branches terminate at an early stage thus exploration of sub-problems becomes faster. However, in this scenario, for most of the instances, branch and bound couldn't find a better solution that can beat the initial upper bound set by Hill Climbing within a 10 minute cutoff time.

A priority queue with number of unvisited nodes information helps in exploring branches in a depth-first manner. If number of unvisited node is same then I used best-first based on the lower bound of the sub-problems that makes the algorithm much faster. Also, exploring branches in a depth first manner rather than breadth first manner, helps in getting the complete tour faster and saves memory, especially in terms of priority queue storage. Although, branch and bound is better than simple brute-force exhaustive search, it does not lower the worst-case time complexity, and it is difficult to predict its exact effect in the average case. However, in real scenarios, it proved to speed up the search considerably.

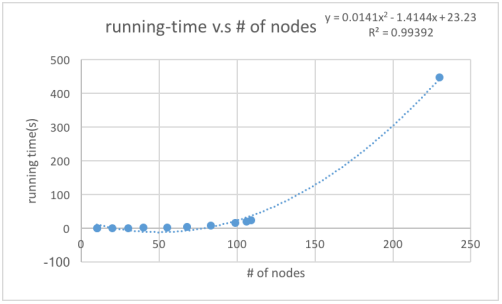
6.2 Discussion of MST Approximation Algorithm

In the MST Approximation Algorithm, we iterate through all the nodes in the MST as the root node in Depth First Search and then select the tour of the minimum length as the result of the algorithm to improve the accuracy. Thus the running time is related to the number of nodes in the graph.For the graph with small number of nodes,such as UKansasState.tsp, the running time is really small and is close to the one of local search algorithms, while for the graph with a large number of nodes, the running time is long, but still less than the one of the Branch and Bound Algorithm. However,the performance of the MST Approximation Algorithm in terms of relative error is the poorest among all the algorithms implemented, while the results are in accordance with the stated lower bound of the optimal solution.

City	# of nodes	Running Time(s)
Toronto	109	23.51
Roanoke	230	446.7
Boston	40	0.52
Cincinnati	100	0.0046
Champaign	55	1.67
Denver	83	8.11
NYC	68	3.6
Philadelphia	30	0.18
SanFrancisco	99	15.35
UKansasState	10	0.0049
UMissouri	106	20.07
Atlanta	20	0.04456

Table 4: MST Approx:Running-Time v.s Number of nodes

Their faster speed than Branch and Bound algorithm and the low accuracy may be due to the fact that these algorithms simply return the local optimal solution under some circumstances while not iterate through all the possible solutions to find the global optimal one. The theoretical time complexity of MST Approximation algorithm is $O(n^2)$, in which n is the number of nodes in the graph. And the empirical scaling observation of the running time versus number of nodes is shown as following, which accords with the theoretical time complexity $O(n^2)$.



6.3 Discussion of Farthest Insertion Algorithm

Comparing with other algorithms, farthest insertion algorithm is a quite simple one.The algorithm can be finished within quite short time and yield a relative good solution. According to the comprehensive table in the evaluation section, we can find that the relative error can be controlled within 10% for all instances. Even though it is not as satisfying as the solution generated by simulated annealing. Considering the running time, the quality of solution is fair enough. It is said that the relative error of farthest insertion is no worse than 11% (Worst-case Performance of farthest insertion is unknown and some other source says it should be 25%). Our implement meets the criterion.

6.4 Discussion of Local Search Algorithms

For the comparison of two local searches, Simulated Annealing and Hill Climbing, we can see from the comprehen-

sive table above, there is a trade-off between running time and real error. Hill Climbing runs much faster than Simulated Annealing (with the same cooling schedule and cutoff time) for our data set, but the real error higher than Simulated Annealing, even though their real errors are both below 0.10. The reason behind this is that Hill Climbing will never go down whereas Simulated Annealing will step back in some situations, which produces better results by running more time. This difference also results in another interesting observation, the running times for different seeds of Hill Climbing have very small variance comparing to those of Simulated Annealing. For more detailed discussion about algorithm performance in the same instance, please check the evaluation section.

7. CONCLUSIONS

We discuss the performance of exact algorithm Branch and Bound, Heuristic algorithms and Local Search algorithms in our project.Branch and Bound can get optimal tour in an acceptable time for small data sets, while for larger data sets, a cutoff time is needed to ensure the running time while sacrifice the accuracy. The running time of MST Approximation is related to the nodes in graph, and the performance in accuracy is the poorest among all the algorithms. Farthest Insertion can get a fair good result really quickly. The Local Search algorithms have the best performance in terms of both the accuracy and running time.

8. ACKNOWLEDGMENTS

Here we want to say thank you to Professor Bistra Dilkina and all the TAs of the 2016 Fall CSE 6140 course for their help.

9. REFERENCES

[1] Fredman, Michael L., et al.*"Data structures for traveling salesmen."*Journal of Algorithms 18.3 (1995): 432-479.

[2] Donald, Davendra. *Traveling salesman problem, theory and applications.* 2011.

[3] Sauer, Joao Guilherme, and Leandro dos Santos Coelho. *"Discrete differential evolution with local search to solve the traveling salesman problem: Fundamentals and case studies."* Cybernetic Intelligent Systems, 2008. CIS 2008. 7th IEEE International Conference on. IEEE, 2008.

[4] Marinakis, Yannis, and Magdalene Marinaki. *"A hybrid genetic–Particle Swarm Optimization Algorithm for the vehicle routing problem."* Expert Systems with Applications 37.2 (2010): 1446-1455.

[5] J. Little, K. Murty, D. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. Operations Research, 11(6):972–989, 1963

[6] Christofides, Nicos. *Worst-case analysis of a new heuristic for the travelling salesman problem.* No. RR-388. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[7] Rosenkrantz, Daniel J., Richard E. Stearns, and Philip M. Lewis, II. *"An analysis of several heuristics for the traveling salesman problem."* SIAM journal on computing 6.3 (1977): 563-581.

[8] Bart Selman and Carla P. Gomes. Selman, B., & Gomes, C. P. (n.d.). Hill-climbing Search. http://www.cs.cornell.edu/gomes/selman-gomes-encycl-hillclimbing.pdf

[9] Tsang, Edward PK, et al. *"A family of stochastic methods for constraint satisfaction and optimization."* The First International Conference on the Practical Application of Constraint Technologies and Logic Programming (PACLP), London. 1999

[10] Menovskaya, and B. Vainstein. *"A Statistical-Thermodynamic Approach to Determination of Structure Amplitude Phases."* Sov. Phys. Crystallogr 24 (1979): 519-524.

[11] Wiener Richard. *"Branch and Bound Implementations for the Traveling Salesperson Problem"* Journal of Object Technology. Vol.2, No.2, February-March 2003. http://www.jot.fm/issues/issue_2003_03/column7.pdf

[12] Laporte, Gilbert.*"The traveling salesman problem: An overview of exact and approximate algorithms."* European Journal of Operational Research 59.2 (1992): 231-247.

[13] D. Johnson and L. McGeoch. *The traveling salesman problem: A case study in local optimization.* In E. Aarts and J. Lenstra, editors, Local Search in Combinatorial Optimization. John Wiley and Sons, London, 1997.