Traveling Salesman Problem Final Report

CSE 6140 2018 Fall - Group 38

Binze Cai
Georgia Institute of Technology
Atlanta, Georgia
binzecai@gatech.edu

Bosheng Jian
Georgia Institute of Technology
Atlanta, Georgia
bosheng@gatech.edu

Minheng Wang
Georgia Institute of Technology
Atlanta, Georgia
mwang457@gatech.edu

Zhou Fang
Georgia Institute of Technology
Atlanta, Georgia
zhoufang@gatech.edu

ABSTRACT

Traveling salesman problem (aka TSP) is a problem to find a cycle that visits each city on the list exactly once with minimum total covered distance. It is one of the classical NP-Complete problems that no current algorithms are available to solve it in polynomial time. In this project, several separate algorithms are examined. Currently, we implemented two Local Search algorithms (Simulated Annealing and Tabu Search), and a Branch and Bound algorithm. Local Search iteratively move from current position to one of neighbor positions and choose among neighboring positions with evaluation function, while Branch and Bound tend to travel through all the scenarios, pruning branches to improve run time performance.

KEYWORDS

Traveling Salesman Problem, Local Search, Simulated Annealing, Tabu Search, Branch and Bound

1 INTRODUCTION

TSP itself sounds quite simple, yet the solution is much more difficult. TSP is one of the most intensely studied problems in combinatorial optimization and it is proven to be NP-hard. The beauty of the problem lies both in its simple formulation and the visualization of the solutions. It has many real word applications such as DNA, airport tour, etc. Therefore, for many years, solving this problem has been an open challenge for many computer scientists. The current record is an instance of 85900 locations on a computer chip and was solved in 2009 by Appelgate et al.

In this project we will come up with several algorithms for TSP. These algorithm can be divided into exact, heuristic and approximation algorithms. An exact algorithm guarantees to find the shortest tour. Heuristic will find a good tour but it is not guaranteed to be the best tour. Approximation algorithm can be a good start for the above two but its performance is worst with the growing input size.

2 PROBLEM DEFINITION

3 RELATED WORK

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Basic Idea. Like backtracking, Branch and Bound method tends to explore the whole State Space Tree. The improvement is by applying upper and lower bounds to prune the tree thus reduce time to compute.

For each node in the State Space Tree, we need to store which the step it takes, i.e. choose which edge, the path from the root to it and the cost to reach it.

4.1.2 Choice on Branching. We decide to choose reduced matrix method as our lower bound. Since for every node in the State Space tree, we can get their cost by the reduced matrix (thus keep them as the lower bound for its children).

Our branching strategy is to branch nodes with the highest cost first, because nodes with higher costs are more likely to exceed the upper bound, thus to be excluded as soon as possible and improve run time performance.

The initial upper bound is found by applying a Greedy Algorithm, i.e starting from point 0, choose the nearest neighbor, move to it and so forth. After visiting all the points, go back to the starting point.

Algorithm 1 Get-Initial-Upper-Bound (*int*[][] *Distances*, *int i*, *set Unvisited*)

```
if Unvisited is Empty then return Distances[0][i]
Uvisited.remove
int lowerest = INF
int closest = 0
for location in Unvisited do
    if Distances[i][location] < lowerest then
        lowerest = Distances[i][location]
        closest = location
return lowerest + Get-Initial-Upper-Bound(Distances, closest,)
```

4.2 Bounding

There are different bounds as mentioned in lecture and we decide to choose Reduced Matrix(RM) Bounds.

When we branch, first of all check if the cost of the current node we branch is greater than best cost, if it is, we can ignore and continue to the next candidate.

Then for each of its child, calculate their cost, if they are greater than the best cost, again ignore and do not add to the priority queue. By repeatedly doing this we successfully pruned the tree.

Algorithm 2 Compute-Cost (*int*[][] *RM*, *int len*)

```
Initialize int[] row and int[] col (size = len)
row.fill(INF), col.fill(INF)
for i from 0 to len do
    for j from 0 to len do
        if RM[i][j] < row[i] then
            row[i] = RM[i][j]
for i from 0 to len do
    for j from 0 to len do
        if RM[i][j] ! = INF and row[i] ! = INF then
            RM[i][j] = RM[i][j] - row[i]
for i from 0 to len do
    for j from 0 to len do
        if RM[i][j] < col[j] then
            col[j] = RM[i][j]
for i from 0 to len do
    for j from 0 to len do
        if RM[i][j] ! = INF and col[j] ! = INF then
            RM[i][j] = RM[i][j] - col[j]
int cost = 0
for i from 0 to len do
    cost += (row[i] ! = INF) ? row[i] : 0
    cost += (col[i] ! = INF) ? col[i] : 0
return cost
```

Algorithm 3 Compute(*int*[][] *Distances*, *int len*)

```
initialize PriorityQueue<Node> pq
Node root = new Node(Distances,initalpath, 0, 0, 0);
pq.push(root)
while pq is not empty do
    Node cur = pq.pop()
    int current = cur.currentLocation
    if cur.cost > BestCost then
        continue
    if cur.level == len then
        if cur.level < BestCost then
            BestPath = cur.path
            BestCost = cur.cost
    for next from 0 to len do
        if cur[current][next] equals INF then
            Continue
        Node child = new Node(cur.RM, cur.path, cur.level + 1,
current,next)
        child.cost = cur.cost + Distances[current][next] +
Compute-Cost(child.RM)
        if child.cost > BestCost then
            Continue
        pq.push(child)
```

4.3 Simulated Annealing

4.3.1 Basic Idea. Simulated annealing is named after a heat treatment process used on metals. In physics, this process minimizes the number of defects in the structure of the material. In algorithm, it slowly and increasingly restricts the freedom of the solution search until it only approves moves toward superior tours. The probability of accepting an inferior tour is defined by acceptance probability function, which is associated with the cooling schedule and the current changes. Generally, when this accepting probability is high, the solution will diversify and oppositely, it will more focus on the improving moves.

Algorithm 4 Simulated Annealing

```
Choose a random state s and define T_0 and cooling rate β
while T does not meet stopping conditions do
    Create a new state s' by randomly swapping two node in s
    Compute α = (cost(s') − cost(s)) / cost(s)
    if α ≤ 0 then
        s = s'
    else
        assign s = s' with probability P(β, T_k)
    T_{k+1} = βT_k
    k + +
return s
```

4.3.2 Implementation. First, the stopping criterion depends on initial temperature T and the cooling rate β.They are tuning parameters. In each iteration, current T will be multiplied by β and cools down until it meets its minimum threshold. If the algorithm were stopped too early, then it will not explore enough search space. In this project, initial temperature T is set to be the square root of input data dimension and the cooling rate is 0.9995 and stopping criterion is 1e-8.

Second, for local search algorithm, the initial start is important because it directly determines the search space in the next few iterations. Here we use a closet neighbor greedy algorithm: randomly pick a start from the tour and then includes current closest node to obtain a greedy solution. Then we set this greedy solution as an initial start to the simulated annealing.

Third, we compare 2opt and 3opt neighbor exchanging and find out that 3opt is shown to be harder for convergence and therefore 2opt is applied in this project. If the neighbor of current solution is better, then we just simply accept it. Otherwise, we accept it by following probability function. The goal of obtaining worse state is to avoid local optimum.

$$P(\beta, T_k) = \begin{cases} e^{\frac{-\beta}{T_k}} & , \beta \geq 0 \\ 1 & otherwise \end{cases}$$

4.4 Tabu Search

4.4.1 Basic Idea. The objective for Tabu search is to constrain an embedded heuristic from returning to recently visited areas of the search space, referred as cycling. The strategy of the approach is to maintain a short term memory of the specific changes of recent moves within the search space and preventing future moves from undoing those changes. Additional long-term memory structures that promote a general diversity in the search across the search space can be introduced.

Algorithm 5 Tabu Search

S_{best} = *initial solution*
set limit size of tabu list $tabu_{limit}$
tabulist = ⊘
while *not meet stopping conditions* **do**
    Create a new state s' by randomly swapping two node in s
    candidate list = ⊘
    for $S_{candidate} \in$ neighborhood of S_{best} **do**
        if $S_{candidate}$ not in tabu list **then**
            Candidate List ← $S_{candidate}$
    $S_{candidate} \leftarrow$ LocateBestCandidate(Candidate List)
    if $cost(S_{candidate} \leq cost(S_{best}))$ **then**
        $S_{best} \leftarrow S_{candidate}$
    while len(tabulist) ≥ $tabu_{limit}$ **do**
        Randomly Delete element from tabu list
return s_{best}

4.4.2 Implementation. The neighbor swapping strategy is the same as in simulated annealing but the differnce here is that we . Here the stopping criterion is by setting a maximum threshold(e.g 10 times) for the times that the algorithm does not find a neighbor better than the best cost so far. If it happens and the counters does not come to the stopping criterion, then our algorithm will randomly choose a refreshing start with greedy algorithm.

4.5 Minimum Spanning Tree Approximation

4.5.1 Basic Idea. Let's talk about Triangle-Inequality first. The least distant path to reach a vertex j from i is always to reach j directly from i, rather than go to some other vertex let's say k, which means $dist(i, j) \leq dist(i, k) + dist(k, j)$. When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP which returns a solution whose cost is never more than twice that of an optimal solution. The idea is to use Minimum Spanning Tree (MST)

The main idea of approximation using MST is traveling though the tree in a preorder way. We will first construct a MST for the given graph using Prim's algorithm which we have already implemented in assignment 2. Then, select any node in the constructed MST as the head node to begin our preorder traversal. Finally, after preorder traversal is finished, add the head node to our solution.

4.5.2 Implementation. We will not explain the construction process of MST in detail, since we have already discuss it in assignment 2. After we get the MST for our input data, we initial a visited set to store the node which we have already visited and a stack to store the remained nodes to visited. Begin from any node in MST, we start the loop of preorder traversal until there is not node in our stack. In each loop, we add the cost from previous node to current node and add the current position to our solution path. After updating the visited set, we will start the next loop and so on. Finally, add the head node, the node we begin our traversal, to our existing solution and we can get our final approximation answer.

Algorithm 6 MST Approximation

$MST = ConstructMST(inputFile)$
$head = MST.anyNode()$
$Visited = set()$
$Visited.add(head)$
$Stack = [head.adjacent()]$
$Cost = 0$
$Path = [head]$
$pre = head$
while $Stack$ is not empty **do**
    $cur = Stack.pop()$
    $Visited.add(cur)$
    $Cost+ = dist(pre, cur)$
    $Path.append(cur)$
    for each $node$ in $cur.adjacent()$ **do**
        if $node$ not in $Visited$ **then**
            $Visited.add(node)$
            $Stack.append(node)$
    $pre = cur$
$Path.add(head)$
$Cost+ = dist(cur, head)$
return $Cost, Path$

5 EMPIRICAL EVALUATION

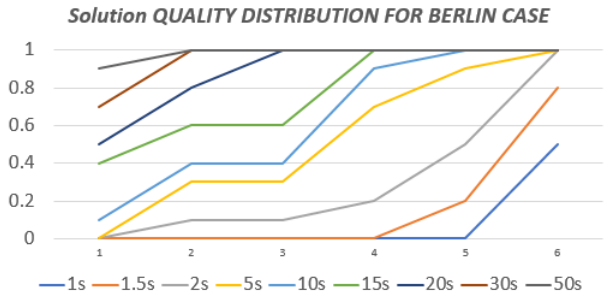

Figure 1: Qualified RTDS for Berlin with Simulated Annealing

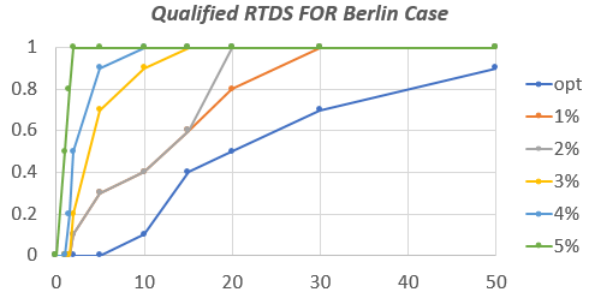

Figure 2: Solution Quality Distribution for Berlin with Simulated Annealing

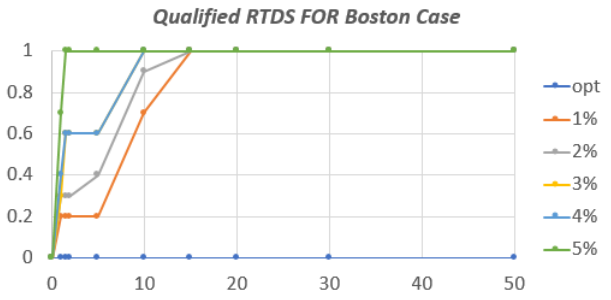

Figure 3: Qualified RTDS for Boston with Simulated Annealing

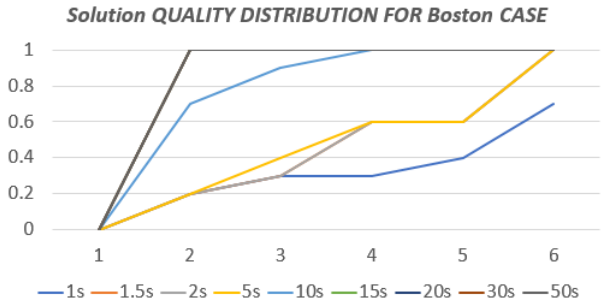

Figure 4: Solution Quality Distribution for Boston with Simulated Annealing

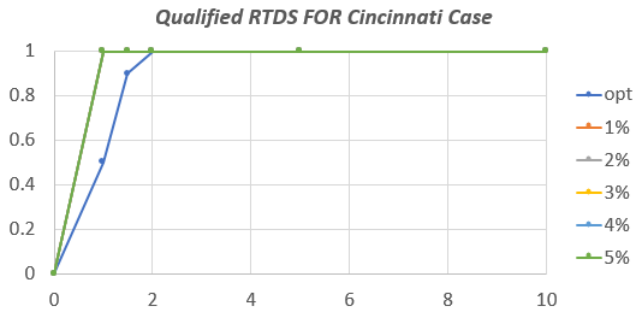

Figure 5: Qualified RTDS for Cincinnati with Simulated Annealing

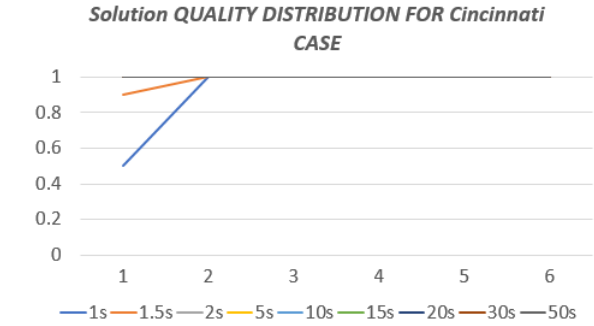

Figure 6: Solution Quality Distribution for Cincinnati with Simulated Annealing

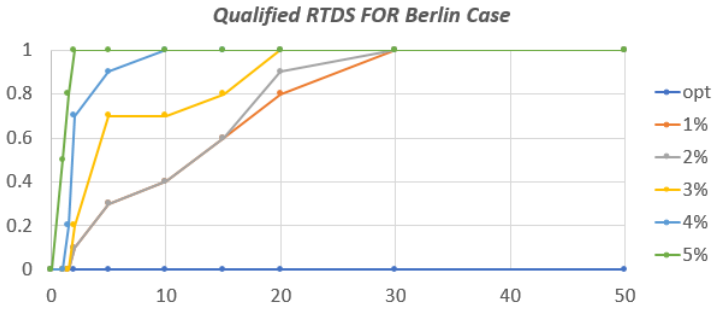

Figure 7: Qualified RTDS for Berlin with Tabu Search

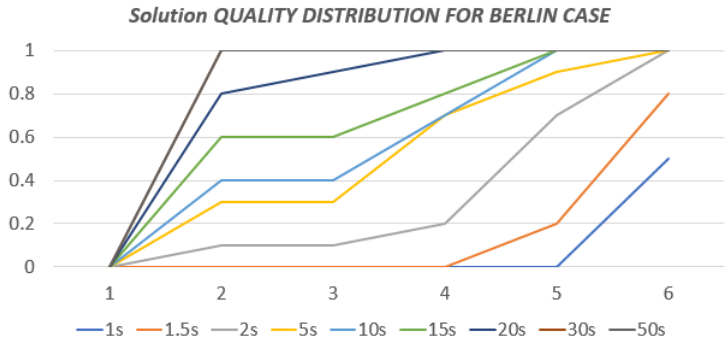

Figure 8: Solution Quality Distribution for Berlin with Tabu Search

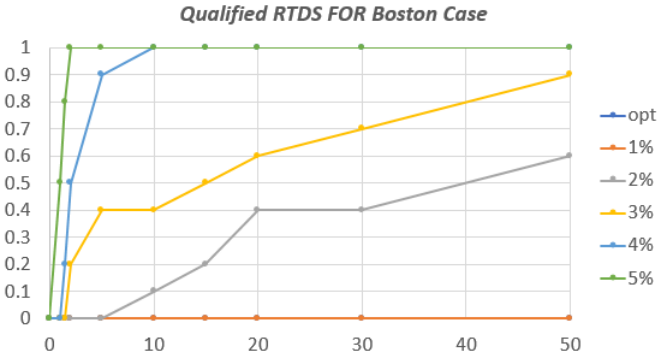

Figure 9: Qualified RTDS for Boston with Tabu Search

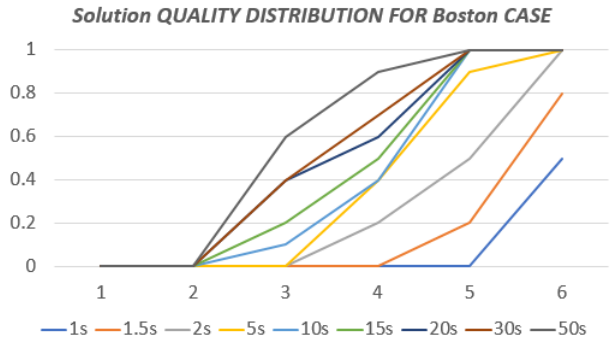

Figure 10: Solution Quality Distribution for Boston with Tabu Search

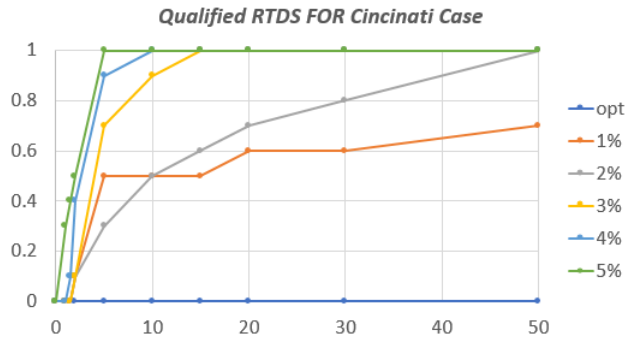

Figure 11: Qualified RTDS for Cincinnati with Tabu Search

Table 1: MST

City	Time(s)	Sol.Qual	RelErr
Atlanta	0.00	2314410	0.155029502
Berlin	0.00	9860	0.307345532
Boston	0.00	1142864	0.274997044
Champaign	0.00	64529	0.225785005
Cincinnati	0.00	308737	0.110756534
Denver	0.00	137474	0.350525085
NYC	0.01	2011514	0.289833057
Philadelphia	0.01	1648282	0.180733835
Roanoke	0.02	845944	0.212847283
SanFrancisco	0.01	1153336	0.397839011
Toronto	0.01	1754373	0.49154616
UKansasState	0.00	68006	0.080111813
ulysses16	0.00	7661	0.116926666
Umissouri	0.00	166220	0.232171979

Table 2: Branch and Bound

City	Time(s)	Sol.Qual	RelErr
Atlanta	500	2003767	0
Berlin	500	8980	0.190665606
Boston	500	1088419	0.214257346
Champaign	500	62920	0.195220637
Cincinnati	500	277952	0
Denver	500	119699	0.175906005
NYC	500	2008450	0.287868344
Philadelphia	500	1691226	0.211496432
Roanoke	500	798665	0.14506241
SanFrancisco	500	897275	0.087494016
Toronto	500	1386622	0.178888822
UKansasState	500	62962	0
ulysses16	500	6859	0
Umissouri	500	162751	0.206456635

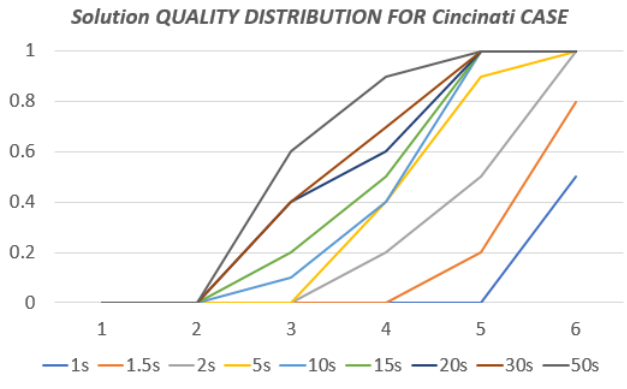

Figure 12: Solution Quality Distribution for Cincinnati with Tabu Search

Table 3: Simulated Annealing

City	Time(s)	Sol.Qual	RelErr
Atlanta	5.22	2003767	0
Berlin	50.31	7542	0
Boston	108.46	896366	0
Champaign	82.32	52643	0
Cincinnati	3.20	277952	0
Denver	94.63	101793	0
NYC	156.49	1559515	0
Philadelphia	83.43	1395981	0
Roanoke	478.50	697486	0
SanFrancisco	201.34	825085	0
Toronto	163.54	1176211	0
UKansasState	4.89	62962	0
ulysses16	8.21	6859	0
Umissouri	162.32	134900	0

Table 4: Tabu Search

City	Time(s)	Sol.Qual	RelErr
Atlanta	3.2	2003767	0
Berlin	46.18	7662	0.015910899
Boston	64.82	922009	0.028607734
Champaign	54.21	53981	0.025416485
Cincinnati	1.8	277952	0
Denver	32.89	102611	0.008035916
NYC	120.67	1654022	0.060600251
Philadelphia	43.21	1432954	0.026485317
Roanoke	250.89	710954	9.195120762
SanFrancisco	100.43	848308	0.028146191
Toronto	87.63	1240388	0.054562489
UKansasState	1.82	62962	0
ulysses16	1.23	8238	0.201049716
Umissouri	65.42	134900	0

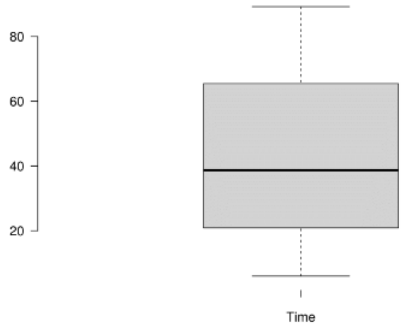

Figure 13: Tabu Search - Berlin -Time to get the best solution based on different seed

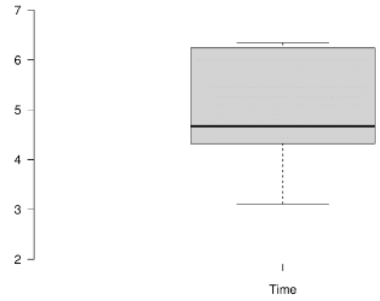

Figure 14: Tabu Search - Boston -Time to get the best solution based on different seed

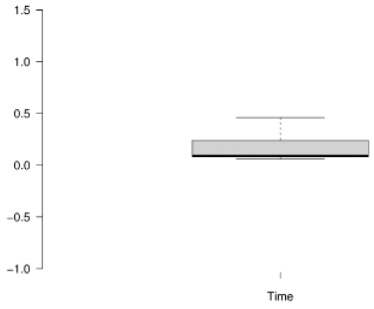

Figure 15: Tabu Search - Cincinnati -Time to get the best solution based on different seed

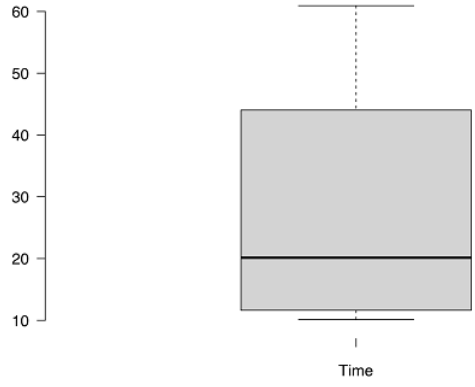

Figure 16: Annealing - Boston -Time to get the best solution based on different seed

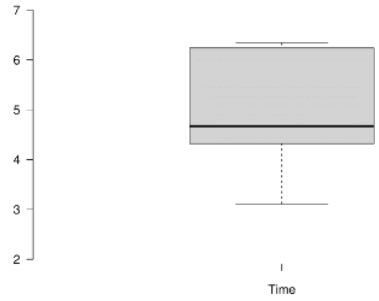

Figure 17: Annealing - Boston -Time to get the best solution based on different seed

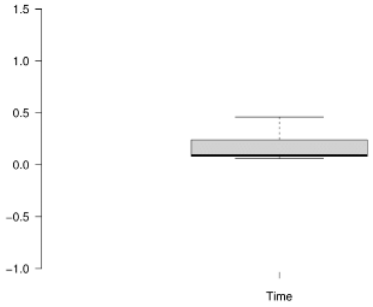

Figure 18: Annealing Search - Cincinnati -Time to get the best solution based on different seed

6 DISCUSSION

In our experiments, we can find that our result is well fitted to the characteristic of each algorithm. Our best approach is Simulated Annealing, which runs in relatively short time and produce desirable result.

MST algorithm runs fastest among all of our algorithm, since it is a 2-approximate algorithm whose final result will be less or equal 2 times of optimal solution. Because all the result comes from preorder traversal of constructed MST, whose complexity is relatively small, the running time is very short.

From the above figures, it is shown that Simulated Annealing and Tabu Search balance the running time and the solution quality, and that is to say, find a relatively good solution in a limited time. However, Tabu tends to get stuck in local optimum though we have chosen the refreshing start.

7 CONCLUSIONS

In this project we implemented several algorithms for TSP problem. These algorithm consist of exact Branch and Bound algorithm, heuristic Simulated Annealing, Tabu Search algorithm and Minimum Spanning Tree approximation algorithms. An exact algorithm guarantees to find the shortest tour. Heuristic finds a desirable tour but there is not guarantee of best solution. Approximation algorithm runs fast and can give us an acceptable tour but its performance is worst with the growing input size comparing with others.