# Finial Report of Group AH

Zhuoran Yu
Georgia Institute of Technology
zhuoran@gatech.edu

Mengdi Li
Georgia Institute of Technology
mli450@gatech.edu

JiaYing Wang
Georgia Institute of Technology
jwang777@gatech.edu

ShaoJun Ma
Georgia Institute of Technology
shaojunma@gatech.edu

## ABSTRACT

In this report, we describe the 5 algorithms implementations for CSE 6140 final project, TSP problem. The algorithms contain 1 for Branch-and-Bound, 2 for Approximation, and 2 for Local Search, which are 2-EX and Simulated Annealing, respectively.

## Keywords

TSP, Algorithm, Branch-and-Bound, Approximation, Local Search, 2-Opt exchange, Simulated Annealing.

## 1. INTRODUCTION

The travelling salesman problem (TSP) is looking for the answer of: Given a list of cities and distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? This is an NP-hard problem. In this report, we provided explanations of five algorithms implementations to solve TSP problems, the results can be seen from the table 1.

## 2. PROBLEM DEFINITION

The formal problem definition we solve here can be described in this way: Give a bunch of .tsp files of different cities, in each .tsp file, 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), we need to find the shortest simple cycle that visits all N points. The cost function $c(u, v)$ is defined as the Euclidean distance between points u and v.

## 3. LITERATURE REVIEW

TSP belongs to the category of NP-hard problems. A various number of methods have been designed to solve this problem. TSP can be modeled as an undirected weighted graph, such that cities are the graph vertices, paths are the graph edges, and a path distance is the edge length. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex only once.

A number of algorithms to solve TSP problem have been developed. Wiener, Richard provided the branch-and-bound algorithm and several implementations in 2003, it can be seen as a good tutorial to introduce BnB algorithm. Several authors have proposed BnB algorithms for the TSP which include Eastman (1958), Little et al. (1963), Shapiro (1966), Murty (1968), Bellmore and Malone (1971), Garfinkel (1973), Smith, Srinivasan and Thompson (1977), Carpaneto and Toth (1980), Balas and Christofides (1981) and Miller and Pekny (1991)[6]. There are lots of approximation algorithms to solve TSP problem.

In 1974, as Rosenkrantz et. al introduced, for the nearest neighbor method, the worst case ratio of the obtained tour to the optimal tour is shown to increase logarithmically with the number of cities. For another method, which we call the nearest insertion method, the worst case ratio is shown to approach 2 as the number of cities increases [7]. In 1985, cemy, Vladimı́r presented a Monte Carlo algorithm to find approximate solutions of the traveling salesman problem [8].

In 1992, Laporte, Gilbert concluded that TSP heuristics can be classical into tour construction procedures which involve gradually building a solution by adding a new vertex at each step, and tour improvement procedures which improve upon a feasible solution by performing various exchanges. Laporte, Gilbert also made a review of approximation algorithms in [9]. For local search method, simulated annealing was developed in 1987, the authors stated that the simulated annealing algorithm is a generally applicable, flexible, robust and easy-to-implement approximation algorithm, that is able to obtain near-optimal solutions for a wide range of optimization problems [10][11].

There are still other algorithms to solve TSP problem, such as, in 2004, Long Jin, Sanmj Huai-Kuang Tsai, Jinn-Moon Yang, Yuan-Fang Tsai, and Cheng-Yan Kao projected an evolutionary algorithm, called the Heterogeneous Selection Evolutionary Algorithm (HeSEA), for solving large TSP [12][21].

In 2008, Sumanta Basu and Diptesh Ghosh presented a survey that Tabu search is one of the most widely applied metaheuristic for solving the TSP [13][21].

In 2009, Rajan K and Anilkumar AK3 proposed an innovative search algorithm motivated by the evolutionary opti-

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

mization technique for the solution of TSP [14][21].

In 2012, Naveen Kumar, Karambir, Rajiv Kumar4 stated that genetic algorithm is one of the best methods which is used to solve various NP-hard problem such as TSP [15][21].

In 2012, Anshul Singh and Devesh Narayan highlighted that the Bee Colony Optimization is for solving the Traveling Salesman problem with its basic mechanism of bees foraging behavior and its efficiency in solving shortest path among various routes [16][21].

In 2012, Sumanta Basu stated that the Tabu search is one of the most widely applied metaheuristic for solving the TSP. In her paper, she reviewed the tabu search literature on the TSP and its variations [17][21].

In 2012, Krishna H. Hingrajiya, Ravindra Kumar Gupta, Gajendra Singh Chandel affirmed that the Ant Colony Optimization (ACO) is a heuristic algorithm which has been proven a successful technique and applied to a number of Combinatorial Optimization problems [18][21].

In 2013, Saloni Gupta and Poonam Panwar has proposed Genetic algorithms appear to find good solutions for the traveling salesman problem, however it depends very much on the way the problem is encoded and which crossover and mutation methods are used [19][21].

In 2014, Sonam Khattar and Dr. Puneet Gosawmi has concluded in their paper how Genetic Algorithm can be used for solving the Traveling Salesman Problem [20][21].

## 4. ALGORITHMS

### 4.1 Branch-and-Bound

The branch and bound algorithm is to find the exact solution in the candidate solution space $S$ for a optimization problem with a objective function $f(x)$. The algorithm branches the solution search tree, with pruning by the lower bound and upper bound to reduce the search space. The crucial idea is to choose a strong lower bound and prune the search tree to reduce the running time. In this algorithm, Held-Karp bound is implemented as a lower bound for solving the problem.

#### *4.1.1 1-Tree Bound*

The definition of 1-Tree is

*Definition 1.* given a graph $G(E, V)$ and a vertex 1, a 1-Tree is a tree of vertex $\{2,3,..,n\}$ and two distinct edges that connecting to vertex 1.

Then a minimum weighted 1-Tree is the minimum cost 1-Tree of all possible 1-Tree, where it can be found by a minimum spanning tree on vertex$\{2,3,...n\}$ plus two distinct shortest edge incident with city 1.

A tour is simply a 1-Tree with every vertex degree is 2. If a minimum 1-Tree is a tour, then it is also an optimal solution for the TSP problem. This can be simply proved by contradiction. If it is not an optimal solution, then since the optimal solution is also a 1-Tree with minimum weight, so the tour could not be worst than optimal. This is contradicted with this 1-Tree has minimum weight. Thus, the minimum 1-tree provides a lower bound for the TSP problem.

#### *4.1.2 The Held-Karp Lower Bound*

The Held-Karp lower bound is an improved 1-Tree lower bound.

A 1-Tree is a lower bound for a TSP problem. But it may not be a desirable bound since it may far from a tour with many vertex degree more than 2. Now consider to improve the 1-tree lower bound. What we want is that for every vertex in 1-tree the degree is 2, where an optimal is also found. Suppose now we add some cost "penalty" to the original edge by

$$c_{ij} = e_{ij} + \pi_i + \pi_j \tag{1}$$

, where $e_{ij}$ is the original edge cost of vertex $i$ and $j$. Notice the optimal tour path which visit each vertex twice does not change in the new distance matrix since it add $\sum \pi$ weight to the original solution, but the 1-tree does change because it is created by a MST.

Denote $L(c_{ij}, T)$ as the cost of 1-Tree or tour T with respect to cost $c_{ij}$, then since $c_{ij} = e_{ij} + \pi_i + \pi_j$.

$$L(c_{ij}, T) = L(e_{ij}, T) + \sum_{i=1}^{n} \pi_i d_i \tag{2}$$

if T is a tour then $L(c_{ij}, T) = L(e_{ij}, T) + \sum_{i=1}^{n} 2\pi_i$. If $T^*$ is a optimal tour then we have $L(c_{ij}, T) = L(e_{ij}, T) + \sum_{i=1}^{n} \pi_i d_i < L(e_{ij}, T^*) + \sum_{i=1}^{n} \pi_i 2$. Denote $c$ for $L(e_{ij}, T)$ and $c^*$ for $L(e_{ij}, T^*)$. Since the 1-Tree cost is always larger then optiaml solution we have

$$\min_T c_T + \sum_{i=1}^{n} \pi_i d_i \leq c^* + \sum_{i=1}^{n} \pi_i 2 \tag{3}$$

Thus for any $\pi$

$$\min_T c_T + \sum_{i=1}^{n} \pi_i (d_i - 2) \leq c^* \tag{4}$$

denote

$$w(\pi) = \{\min_T c_T + \sum_{i=1}^{n} \pi_i (d_i - 2) \tag{5}$$

$w(\pi)$ is also a lower bound for TSP problem, the Held-Karp lower bound is

$$\text{HK=maxw}(\pi) \tag{6}$$

A Held-Karp bound may not be a optimal tour but it is very close to the optimal length. Held and Karp use subgradient descent to update $\pi^m \to \pi^{m+1}$ with formula

$$\pi_i^{m+1} = \pi_i^m + t^m (d_i^m - 2), \text{for } 1 \leq i \leq n \tag{7}$$

,where $d_i^m$ is the $m$th iteration and $t^m$ is the step length. For a vertex $i$, $\pi_i$ will increase if $d_i < 2$ and decrease if $d_i > 2$. A high degree vertex $i$ and $j$ will result in high cost $c_{ij} = e_{ij} + \pi_i + \pi_j$ in the next 1-Tree evaluation step and vice versa. The iteration is to try to make all vertex with $d_i = 2$, approaching to the optimal tour.

The choice of $t^m$ is from Valenzuela[5] with the formula

$$t^m = (m-1)\frac{2M-5}{2(M-1)}t^1 - (m-2)t^1 + \frac{1}{2}\frac{(m-1)(m-2)}{(M-1)(M-2)}t^1 \tag{8}$$

,where $m$ is the iteration, $M$ is the length of the sequence and $t^1$ is the initialed step length. $t^1$ is obtain by

$$t^1 = \frac{1}{2n} L(e_{ij}, T) \tag{9}$$

,where $n$ is the size of the problem. The initial $\pi^0$ is set to be vector $\{0, 0, 0, .., 0\}$.

### *4.1.3 Algorithm in Detail*

The main idea of bnb algorithm with Held-Karp bound is to force the 1-Tree to construct a tour. The computation of HK bound can be very time consuming for a long step sequence. The HK bound has very clear pros to other lower bound like the reduced matrix and simply MST. The HK bound will improve dramatically the search space by pruning much the child node. However, the heavy computation for HK bound would make the branch step less available in limited time. It is important to find a good choice of $M$ to get a relatively strong lower bound with less time.

The branch step of this algorithm is to find a HK bound while it is also exactly a tour with every degree 2. The HK bound algorithm may not find a tour even after much iteration because the 1-Tree does not guarantee to become a tour. If a tour can not be constructed then there must be some vertex $u$ with maximum degree $D$ ($D$>2), which in practice is always 3 after iteration. The branch idea is to delete some edge $e_i(u, v_i)$ , and then search in the modified graph for a tour, which would result in higher lower bound for the missing edge.

Deleting the edge is to reduce the available edge for the vertex $u$ trying to decrease its degree in the next search. If we delete all the edges except for the optimal tour, we can always find the optimal solution at last, so this multi-way branch strategy can guarantee to find the optimal solution given unlimited time.

**Algorithm 1** Held-Karp Bound Algorithm

```
$HK(DistMatrix, M)$
    $\pi_0 = \{0, 0, .., 0\}$
    for $m = 1$ to $M$ do
        if $Deg.Max$==2 then
            Break
        end if
        Update $NewDM_{ij} = DM_{ij} + \pi_i + \pi_j$
        $Deg^m, L^m \leftarrow MST(NewDM) + e_1^m + e_2^m$
        if $HKbound < L^m$ then
            $HKbound \leftarrow L^m$
        end if
        Update $\pi_i^m = t_m(Deg_i^m - 2)$
    end for
```

The evaluation of Held-Karp bound is computed and updated by iteration. In every iteration, the distant matrix of the graph add the weight $\pi$ to develop a new MST. The MST function is developed by Prim algorithm with an array to maintain and record the path and degree of the MST. So the MST function return a minimum weight with modified degree and path information. Then search distinct edge $e_1$ and $e_2$ that adjacent to vertex 0 to derive the 1-Tree bound in step $m$.

The time step is controlled by parameter $M$ which will be determined by empirical experiment. If $M$ is too small, the iteration may result in a relatively weak lower bound and can't find a tour. If $M$ is too large, then the iteration would be very time consuming with large problem size since the time complexity is $O(Mn^2)$. The branch strategy is to search from the child tree with smallest lower bound. Since when a tour is found, it is exactly equal to the lower bound. So we want to find better solution as fast as possible for pruning other search tree.

**Algorithm 2** Branch and Bound Algorithm

```
$Branch(Ub, Lb, Deg, DistMatrix, M)$
 1: if $Lb > Optimal$ then
 2:     return
 3: end if
 4: if $Deg.Max$==2 then
 5:     A solution found ← Lb
 6:     if $Lb < optimal$ then
 7:         $Optimal = Lb$
 8:     end if
        return
 9: end if
10: $v \leftarrow Deg.max$
11: for $u \leftarrow v.adj$ do
12:     NewMatrix $ND \leftarrow DeleteEdge(u, v)$
13:     if $Optimal < Lb \leftarrow HK(ND, M)$ then
14:         Continue
15:     end if
16:     store in Q $ChildLb, ChildDeg \leftarrow HK(ND, M)$
17: end for
18: while (!Q.empty()) do
19:     Min Lower bound Child $c$=Q.pop()
20:     Branch($c : Ub, Lb, ND, M + 1$)
21: end while
$Ub$=heuristic result
Set time sequence $M$
Initial $Deg$ $Lb$
Recursive($Branch$(Ub,Lb,Deg,DistMatrix,M))
```

Let's take the problem Cincinnati for example to show how the branch works. The first evaluation of HK bound for the original graph generate a 1-Tree with vertex 8 degree 3(the vertex starts from 0 in the program so it is actually vertex 7). In the next branch step, we delete the edge adjacent to vertex 8, that is $e(8, 1), e(8, 4), e(8, 9)$ for the next search. Now we have three child to branch with 1 deleted edge for every child.

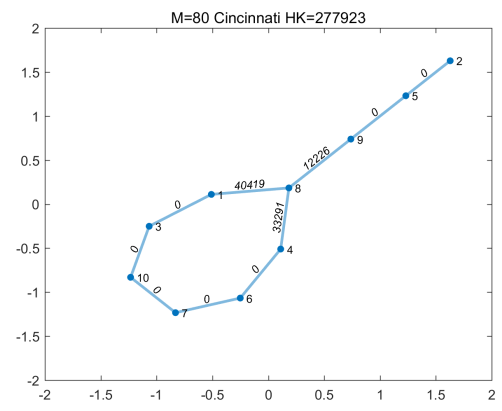

**Figure 1: Branch Step 1**

Then we derive the HK bound for the three child graphąč Use the best search first from the child with smallest lower bound. In this case, the HK-bound is also a solution for hound=284175 and bound=277952. For the problem, we know that HK bound=277952 is the optimal solution. Then continue for the unsolved child HK=278010 and update the current optimal solution to prune.

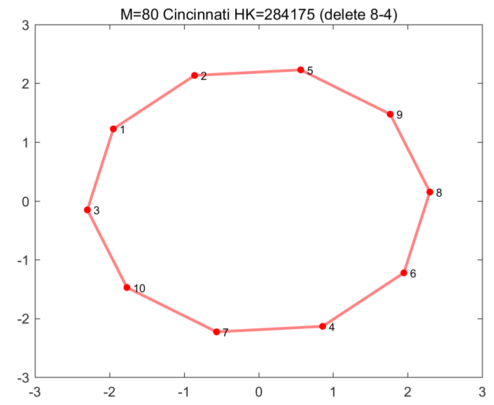

**Figure 2: Branch Step 2**

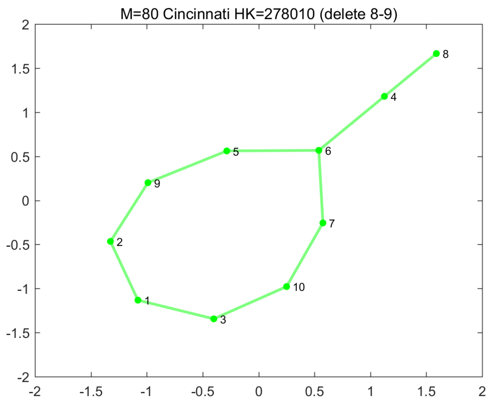

**Figure 3: Branch Step 2**

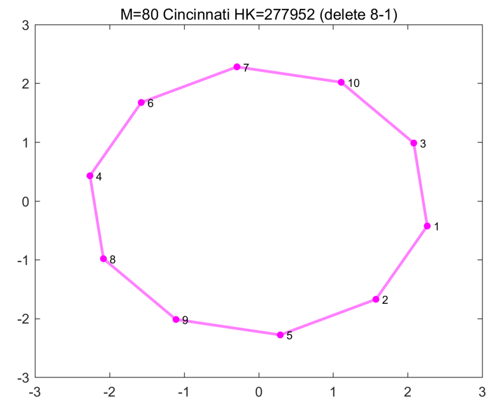

**Figure 4: Branch Step 2**

## 4.2 Approximation-MST

MST-Approximation is an algorithm for (Euclidean) TSP that uses the Minimum Spanning Tree (MST).
The high-level idea of the algorithm:

**Algorithm 3** MST Approximation algorithm

```
Input Graph $G(V, E)$
1: T← empty tour
2: $M$=Kruskal(G)
3: turn $M$ into rooted tree
4: Tree root $N_0$ ← arbitrary node
5: $DFS(N_0, M, T)$ //DFS search in MST
     **return** T
```

1. Firstly, find the Minimum Spanning Tree of the graph (map). Here I used the Kruskalaŕs algorithm to develop the MST.

2. Pick any node (point) to be the root of MST.

3. Traverse the tree in pre-order, which means to develop a DFS program.

4. Return the order of nodes visited in pre-order.

## 4.3 Approximation-Nearest Neighbor

Nearest Neighbor Approximation algorithm is kind of a greedy algorithm for solving TSP problem.

– Pick up the first node in the list and added it to the route.

– Search other nodes which have not been added into the route and pick up the node that has the shortest distance to the current node. Insert this picked up node into the route

– Pick the last added node and redo the step 2.

– Continue doing step 3 until we added all the nodes to the route.

**Algorithm 4** NN algorithm

```
Input $V = 1, 2, .., n$
Initialize M=$\emptyset$
$T \leftarrow 0$
1: **while** ($!V.empty()$) **do**
2:     $e = (u, v) \leftarrow$ shortest edge between $V$ and $M$
3:     $M \leftarrow M + u$
4:     $V \leftarrow V - v$
5:     $T = T + e$
6: **end while**
7: $T = T + e(0, n)$
     **return** T
```

This algorithm takes $O(n^2)$.
Space Complexity: It takes $O(n^2)$.
This is a $O(logn)$ Approximation Algorithm.

## 4.4 Local Search 1 - 2opt-Exchange Revised

### 4.4.1 Introduction of 2-EX algorithm

The basic idea of 2-Opt Exchange algorithm is to exchange two edges and if the total distance after exchange is smaller than before, then we accept the exchange, else we rechoose the edges. We tried the naive 2-OPT Exchange algorithm firstly, but we found the results are so bad, so that we considered another revised 2-Opt exchange algorithm, which has much more optimization than the naive 2-Opt algorithm. Here are the creative points in our revised algorithm:

(1) The Greedy initial path

First, instead of choosing the initial path randomly, we use the Greedy way to get the initial path, which is starting from a random start node, and add a new node that is nearest to the last node each time, until make it a tour. So that we can get a relative better initial path.

(2) Restart

Since we have the cut off time with 10 minutes, if we cannot find a better solution when we end of the exchange process, then we can jump out of the iteration and restart it, which means we can jump out and randomly choose another starting node, and then exchange the edges again. In that case, we can make full use of the 10 minutes, and it also provides us more chances to get the better solution.

(3)Consider only K=5 nearest neighbors and First - Improvement Selection

Instead of randomly exchanging each pair of edges without any criteria, for each node, we only consider the $K = 5$ nearest neighbor, which can help us to save a lot of time in each iteration. In order to do this, we need to firstly make a matrix to store the nodes with weights from minimum weights to maximum weights. When given a vertex $u_i$, consider each of the two tour neighbours $u_j$. For $u_i$, if in $K = 5$ radius neighbor, there is a neighbor uk that $w(u_i, u_k)$ is less than $w(u_i, u_j)$, then we accept the first improvement 2-exchange move[22].But how to decide whether to change the previous node of $u_i$ or the next node $u_i$? We also need to discuss it in cases. Firstly, if $w(u_i, u_k)$ is less than the weight of $u_i$ and its previous node, then we accept the exchange,and get the two new edges: $u_i.prev$ - $uk.prev$ and $u_i$ - $u_k$. Else, if $w(u_i, u_k)$ is less than the weight of $u_i$ and its next node, then we accept the exchange, and get the two new edges: $ui.next - uk.next$ and $u_i$ - $u_k$. Whenever we get the new smaller weight, we accept, and this is called First Improvement Selection.

(4)No look bits (DLB) strategy

The DLB strategy is: if no improvement steps for a given node u was found, then until an incident edge changes, do not consider u, which will help us to avoid a lot of unnecessary tries[22]. For each search step, the jump out criteria is if all nodes are marked DLB, then we need to jump out of this search step, and use the Restart strategy to reselect a new path. After each search step, reset the DLB for all nodes. This DLB strategy is significantly helps us to reduce the time complexity.

**Algorithm 5** 2-EX algorithm

```
Dm=Distance Matrix
NM: nearestneighbour Martrix Upper Bound(Ub)=Inf
1: while (time<cut off time) do
2:     randomly start node $s$;
3:     $T_0$=greedy initial path($s$)
4:     $T'$=Algorithm2opt($T_0$,cut off, Ub)
5:     if $T'<Ub$ then
6:         $Ub$=$T'$
7:     end if
8: end while
9: Return $Ub$
```

**Algorithm 6** Algorithm2opt

```
$k \leftarrow$ number of DLB nodes
while time<cutoff time and k<size do
    $u_i$=randomly select node
    if $u_i$ in DLB then
        Continue
    end if
    for i=0 to 5 do
        $u_k = NM[u_i][i]$
        if $c(u_i, u_k) < c(u_i, u_i.pre)$ then
            $T' < T_0$
            unmarked n=$\{u_i, u_k, u_i.pre, u_i.pre\}$
            k=k-number of already unmarked n
            flag1=1
            break
        else if $c(u_i, u_k) < c(u_i, u_i.next)$ then
            $T' < T_0$
            unmarked n=$\{u_i, u_k, u_i.pre, u_v.pre\}$
            k=k-number of already unmarked n
            flag1=1
            break
        else
            mark $u_i$
            k++
            flag2=1
            break
        end if
        if flag1==1 or flag2==1 then
            Continue
        else
            mark $u_i$
            k++
        end if
    end for
end while
Return $T'$
```

### 4.4.2 Time Saving Analysis

The revised 2-Opt Exchange algorithm can help to save a lot of time and improve the accuracy in this areas:

- It gives us a relative better initial path, if we just randomly get the initial path, that may be very large, so we need to exchange many times of edges to get a relative smaller path.

- It has the restart strategy, which will use all of the 10 minutes. However, in the naive 2-Opt method, once we get some specific path, we cannot get better improvement without restarting.

- Only consider the 5 nearest neighbor also helps us to save a lot of time, because we do not need to search for $O(n)$, but instead to search for $O(C)$, in where C is meanwhile a small constant.
- The most creative part is the DLB strategy. because we mark the nodes that has not changed after 5 iterations, and unmark it until it has changed. This will save us a lot of time when randomly choosing the node the consider. Although it is hard to calculate the specific time complexity it has saved, it does can save us a lot of time, because we do not need to check the meaningless exchange.

## 4.5 Local Search - Simulated Annealing

**Algorithm 7** Simulated Annealing algorithm

```
Input :EuC_2D cooridination of nodes
  $T \leftarrow$ Initial tour
  best tour $T^* \leftarrow T$
  while (time<cut off time) do
      swap$(i, j)$
      $T' \leftarrow$ New Tour
      if $T' < T^*$ then
          $T^* \leftarrow T'$
      else
          $u$=random(0,1)
          $p$=exp$(T^* - T')$/temp
          if then$u < p$
              $T^* = T'$
          end if
      end if
      temp=temp$\times$ coe
  end while
      return $T^*$
```

The creation here in SA algorithm is that we combine 2-EX and SA together to solve the problem, which means once it satisfies the condition in SA, we cut two edges accordingly and build new edges between these four nodes, and change the trace direction accordingly.

- Annealing is a metallurgic process for improving the strength of metals.
- Key idea: cool metal slowly during the forging process.

Key ideas in simulated annealing:

1. Simulated annealing = a modified local-search.
2. Use it to solve a combinatorial optimization problem.
3. Associate energy with cost $\rightarrow$ Goal: find lowest-energy state.
4. Recall problem with local-search: gets stuck at local minimum.
5. Simulated annealing will allow jumps to higher-cost states.
6. If randomly-selected neighbor has lower-cost, jump to it (like local-search does).
7. If randomly-selected neighbor is of higher-cost $\rightarrow$ flip a coin to decide whether to jump to higher-cost state
   - Suppose current state is s with cost $C(s)$.
   - Suppose randomly-selected neighbor is $s'$ with cost $C(s') > C(s)$.
   - Then, jump to it with probability
   - $e - [C(s') - C(s)]/kT$
8. Decrease coin-flip probability as time goes on $\rightarrow$ by decreasing temperature T.
9. Probability of jumping to higher-cost state depends on cost-difference

## 5. EMPIRICAL EVALUATION

### 5.1 Branch-and-Bound

The branch and bound algorithm is controlled by the step sequence length. The Valenzuela and Christine L[5] has mentioned

$$M = 28n^{0.6}$$

. In practice, if we implement the $M$ as this formula, the time consuming for iteration will be very larger to larger problem size. After several experience, observation that for problem size under 100 will converge to the maximum HK bound in about 50-80 iteration. So the initial $M$ is to set to 50 and 80 for all problem size under 100 for two experience. Also, for the next level search tree child, the iteration will increase by 1 in order to derive the solution as quickly as possible.

Two experience are conducted with $M = 50$ and $M = 80$ in the platform with following information.

CPU:I7-HQ 6700 2.60$GHz$.

RAM:60$GHz$.

Language:$c++$.

Compiler:Visual Studio 2015.

**Table 1: M=50 Branch and Bound Result**

| Dimension | Dataset | Time(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 0.11 | 277952 | 0 |
| 10 | UKansasState | 0.01 | 62962 | 0 |
| 20 | Atlanta | 0.05 | 2003763 | 0 |
| 30 | Philadelphia | 7.52 | 1395981 | 0 |
| 40 | Boston | 58.97 | 893536 | 0 |
| 55 | Champaign | 192.48 | 52643 | 0 |
| 68 | NYC | 600 | 1580684 | 0.016 |
| 83 | Denver | 600.00 | 104452 | 0.040 |
| 99 | SanFrancisco | 211.30 | 810196 | 0 |
| 106 | Umissouri | - | - | - |
| 109 | Toronto | 600.00 | 1226652 | 0.043 |
| 230 | Roanoke | - | - | - |

We can see that in the case $M = 50$, the branch and bound algorithm actually provides very desirable solution for most of the problem Then Another parameter $M = 80$ is selected for the experiment. The result is in the below table. Actually, $M = 80$ performs better than $M = 50$. In this case, NYC can get the optimal solution, and some other city like Champaign and SanFrancisco would find the optimal solution quickly. We can also get a solution for Umissouri. So $M = 50$ may be too small to search for a tour, and $M = 80$ could be effective to search the solution for larger dataset.

Actually, a well implemented branch and bound would maintain the accuracy and running time both at a small size problem. In this project, the branch and bound algorithm is very effective and impressive in solving most of the problem for the optimal solution in a short time. Unlike other algorithm, the branch and bound can always guarantee the optimal solution if it is not stopped by the cut time.

**Table 2: M=80 Branch and Bound Result**

| Dimension | Dataset | Time(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 0.10 | 277952 | 0 |
| 10 | UKansasState | 0.01 | 62962 | 0 |
| 20 | Atlanta | 0.07 | 2003763 | 0 |
| 30 | Philadelphia | 0.58 | 1395981 | 0 |
| 40 | Boston | 75.81 | 893536 | 0 |
| 55 | Champaign | 77.17 | 52643 | 0 |
| 68 | NYC | 0.61 | 1555060 | 0 |
| 83 | Denver | 600.00 | 104179 | 0.037 |
| 99 | SanFrancisco | 12.35 | 810196 | 0 |
| 106 | Umissouri | 600.00 | 133634 | 0.007 |
| 109 | Toronto | 600.00 | 1178770 | 0.0022 |
| 230 | Roanoke | - | - | - |

It is interesting to discover what kind of time step sequence would be optimal for the problem, and what is the relationship between the the problem size and $M$. More discussion may be presented in the future.

Both $M$ can not solve the largest data set for n=200. It shows that for both the parameter, the lower bound is still far below the optimal solution. This means a much larger time sequence is needed for this size of dataset. Actually, when $M = 2000$ the lower bound could be close to the optimal solution, but the time consuming is unacceptable when it takes minutes for just one branch step. It could not find a solution in short time.

## 5.2 Approximation-MST

The MST-approximation algorithm is a 2-ratio approximation algorithm, which means the worst result will not be worse than 2 optimal. In our result, we can see that the worst result has the RelError of 0.4242, which is not larger than 0.5. So, we can know that the MST-algorithm is correct. Meanwhile, comparing to the Nearest Neighbor Approximation algorithm, MST-approximation needs to use Kruskal algorithm and develop the DFS algorithm, it must cost more time than Nearest Neighbor Approximation algorithm.

**Table 3: MST-Approximation Result**

| Dimension | Dataset | Time(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 0.01 | 301215 | 0.084 |
| 10 | UKansasState | 0.00 | 68089 | 0.081 |
| 20 | Atlanta | 0.06 | 2380447 | 0.19 |
| 30 | Philadelphia | 0.01 | 1646247 | 0.18 |
| 40 | Boston | 0.07 | 1150959 | 0.29 |
| 55 | Champaign | 0.03 | 65714 | 0.25 |
| 68 | NYC | 0.09 | 2027108 | 0.30 |
| 83 | Denver | 0.09 | 134748 | 0.34 |
| 99 | SanFrancisco | 0.11 | 1134996 | 0.40 |
| 106 | Umissouri | 0.19 | 178251 | 0.34 |
| 109 | Toronto | 0.15 | 1675107 | 0.42 |
| 230 | Roanoke | 0.90 | 838287 | 0.28 |

## 5.3 Approximation-Nearest Neighbor

The experiment platform information: CPU:OSX i7 2.60$GHz$.
RAM:16$G$.
Language:$c$.
Compiler:Terminal.
The running time of this method is the most fast one among five algorithms, however, the output is not as correct as the optimal one. This algorithm will give the worst case logn approximation to the optimal output.

**Table 4: NN-Approximation Result**

| Dimension | Dataset | Times(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 0.000013 | 333784 | 0.20 |
| 10 | UKansasState | 0.000012 | 74962 | 0.19 |
| 20 | Atlanta | 0.000023 | 2117949 | 0.057 |
| 30 | Philadelphia | 0.00010 | 1691216 | 0.21 |
| 40 | Boston | 0.000069 | 1115462 | 0.25 |
| 55 | Champaign | 0.00013 | 62894 | 0.19 |
| 68 | NYC | 0.00019 | 2008425 | 0.29 |
| 83 | Denver | 0.00029 | 135395 | 0.35 |
| 99 | SanFrancisco | 0.00036 | 897238 | 0.11 |
| 106 | Umissouri | 0.00043 | 164536 | 0.24 |
| 109 | Toronto | 0.00045 | 1386570 | 0.18 |
| 230 | Roanoke | 0.0019 | 843497 | 0.29 |

## 5.4 Local Search - 2 Exchange

The platform information of 2EX is CPU:i7-4510 CPU 2.60$Ghz$
RAM:16$G$
Language:$C++$
Compiler: Eclipse

**Table 5: 2EX-revised Result**

| Dimension | Dataset | Times(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 0.0015 | 277952 | 0 |
| 10 | UKansasState | 0.0004 | 62962 | 0 |
| 20 | Atlanta | 0.0087 | 2003762 | 0 |
| 30 | Philadelphia | 0.053 | 1395980 | 0 |
| 40 | Boston | 600 | 894437 | 0.001 |
| 55 | Champaign | 2.19 | 52642 | 0 |
| 68 | NYC | 162 | 1555060 | 0 |
| 83 | Denver | 600 | 101460 | 0.01 |
| 99 | SanFrancisco | 600 | 819008 | 0.011 |
| 106 | Umissouri | 600 | 133772 | 0.008 |
| 109 | Toronto | 600 | 1178720 | 0.0022 |
| 230 | Roanoke | 600 | 692857 | 0.057 |

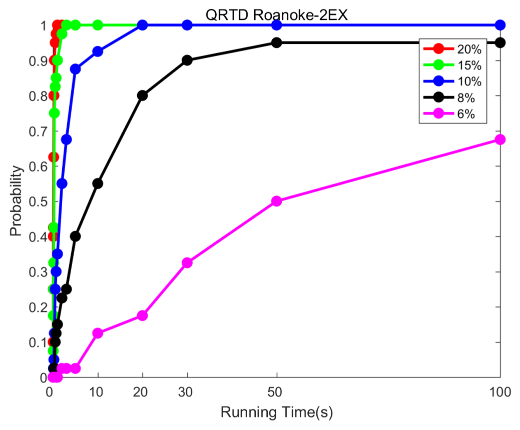

**Figure 7: QRTD-Roanoke-2EX-Revised**

The QRTD, SQD and Box-plot evaluations of Toronto and Roanoke for this algorithm are listed below:

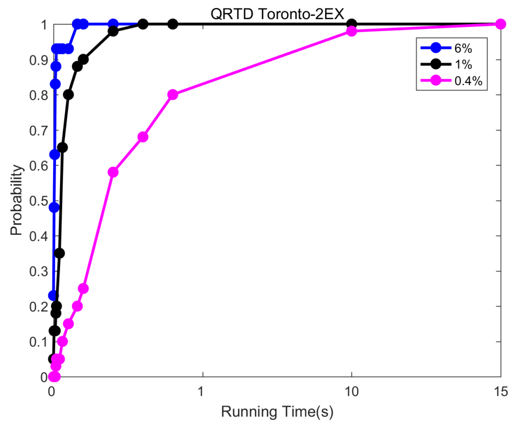

**Figure 5: QRTD-Toronto-2EX-Revised**

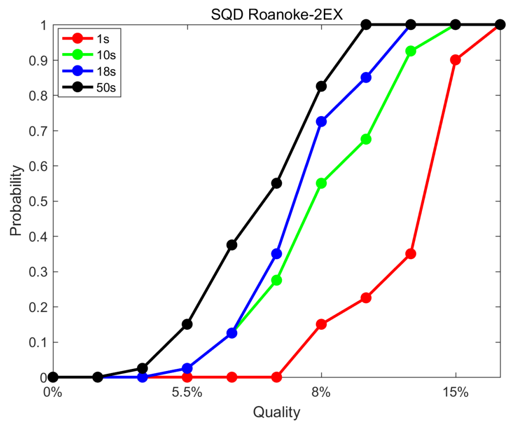

**Figure 8: SQD-Roanoke-2EX-Revised**

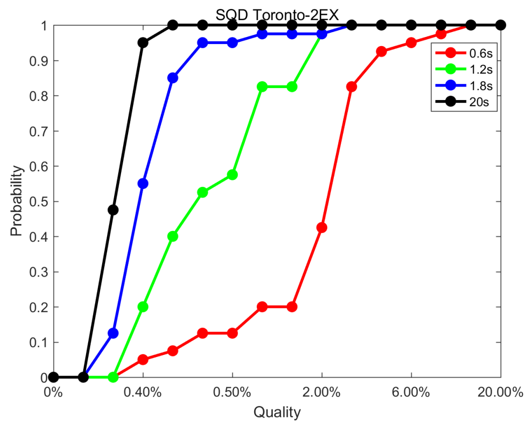

**Figure 6: SQD-Toronto-2EX-Revised**

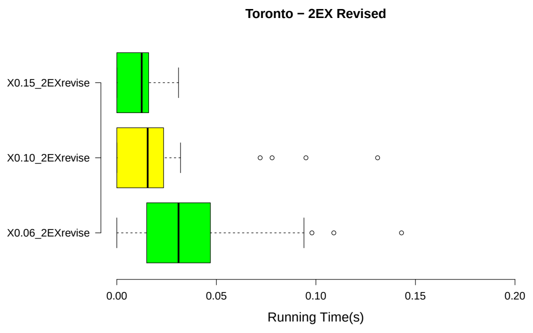

**Figure 9: Boxplot-Toronto-2EX-Revised**

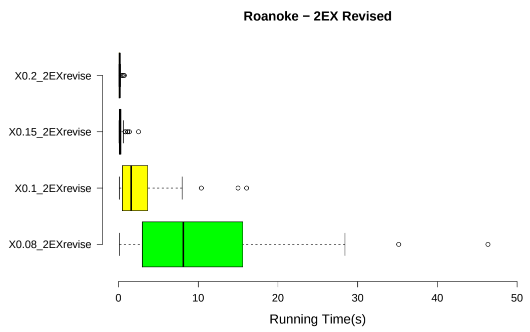

**Figure 10: Boxplot-Roanoke-2EX-Revised**

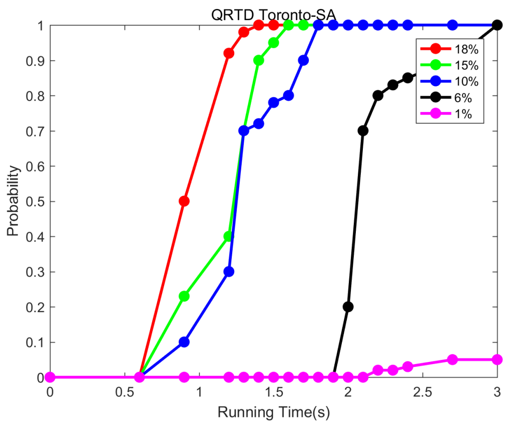

**Figure 11: QRTD-Toronto-SA**

As we can see in the table and QRTDs, the revised 2-EX algorithm has much better performance than the naive 2-EX algorithm. The reason for that has already been said in the former parts, so no repeat analysis here. The result is similar to our expectation.

## 5.5 Local Search- Simulated Annealing

The following is the platform information and empirical result. CPU:i7-HMQ 4700 2.40$GHz$.
RAM:16$G$.
Language: $c++$.
Compiler:Terminal.

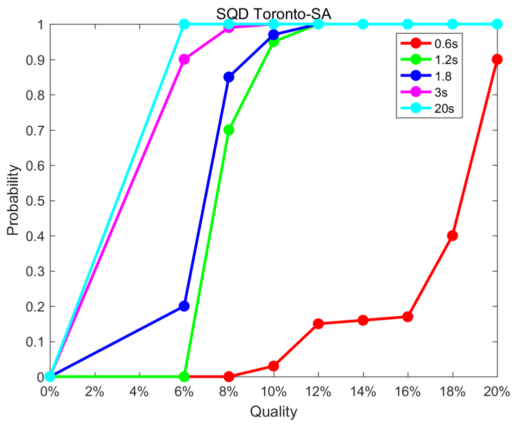

**Figure 12: SQD-Toronto-SA**

**Table 6: Simulated Annealing Result**

| Dimension | Dataset | Times(s) | Length | RelErr |
|---|---|---|---|---|
| 10 | Cincinnati | 60 | 280237 | 0.008 |
| 10 | UKansasState | 60 | 62958 | 0.00001 |
| 20 | Atlanta | 60 | 2093063 | 0.045 |
| 30 | Philadelphia | 60 | 1437589 | 0.030 |
| 40 | Boston | 60 | 938624 | 0.050 |
| 55 | Champaign | 60 | 55018 | 0.045 |
| 68 | NYC | 60 | 1602160 | 0.030 |
| 83 | Denver | 60 | 108649 | 0.082 |
| 99 | SanFrancisco | 60 | 866936 | 0.070 |
| 106 | Umissouri | 60 | 143799 | 0.084 |
| 109 | Toronto | 60 | 1228370 | 0.044 |
| 230 | Roanoke | 60 | 691600 | 0.055 |

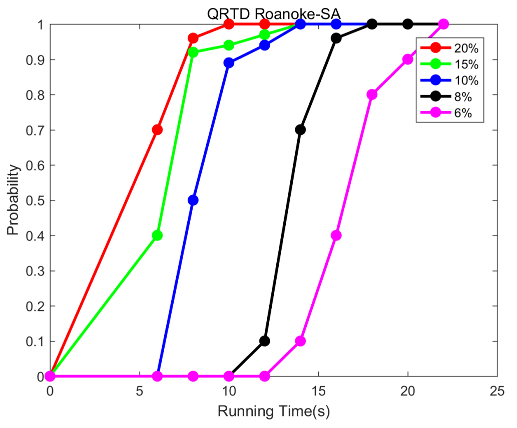

**Figure 13: QRTD-Roanoke-SA**

The QRTD, SQD and Box-plot evaluations of Toronto and Roanoke are listed below.

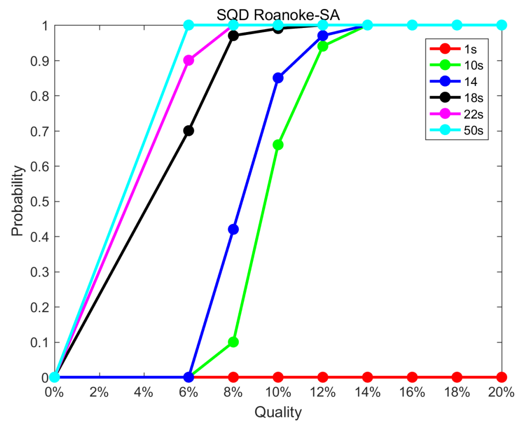

**Figure 14: SQD-Roanoke-SA**

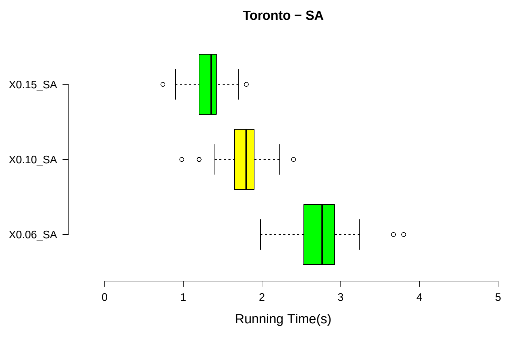

**Figure 15: Boxplot-Toronto-SA**

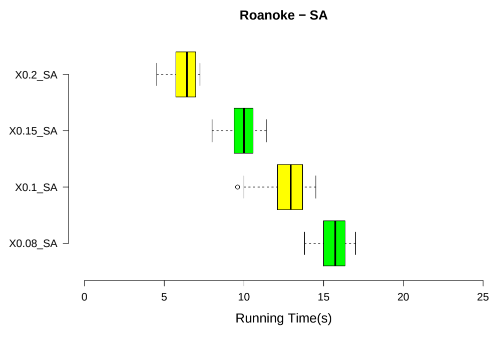

**Figure 16: Boxplot-Roanoke-SA**

From QRTD figures we can see the relationship between possibility to solve the problem and time under different precision requirement. As time pasts, we can always find a solution around 106% of optimal solution, which means the possibility to solve a 106% problem will always be 100% at last. But, still, in available cutting-off-time, we are not able to find a solution better than 106% precision or the optimal solution when using this algorithm.

From SQD figure we can see the relationship between possibility to solve the problem and precision requirements under different cutting-off-time. For Toronto, 3s seems to be a safe time to get enough precise results and for Roanoke, 22s is a safe time to get 100% percent of optimal solution in available cutting-off-time.

## 6. DISCUSSION

### 6.1 Comparison of Two Versions 2EX

As we can see in the above boxplot picture, in the same case (Toronto), and the same accuracy, the revised 2EX is much better than the naive 2EX algorithm, which means our optimization did help a lot on the both accuracy and time complexity.

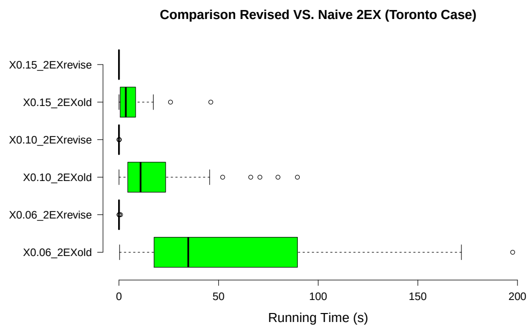

**Figure 17: Comparison REVISED VS NAIVE 2EX**

### 6.2 Comparison of Local Search

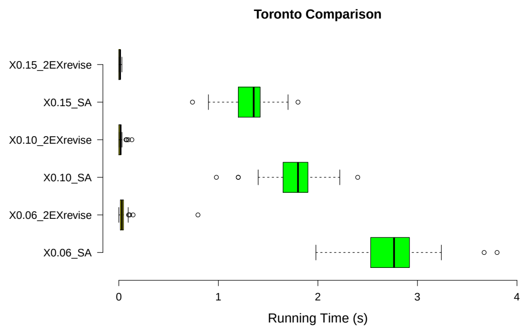

**Figure 18: Toronta-Comparison of Box Plot**

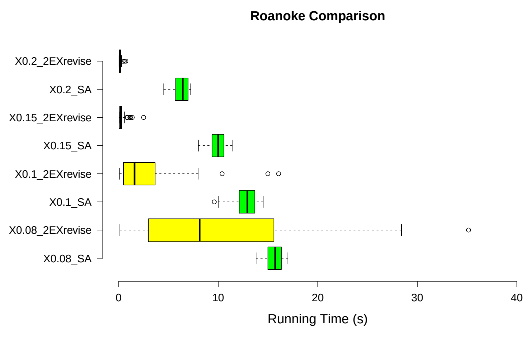

**Figure 19: Roanoke-Comparison of Box Plot**

As we can see from the QRTD and SQD figures of new 2-Ex and SA algorithms, the New 2-Ex speed and precision get much better than before. For Toronto city, whose size is smaller,the new 2-Ex is able to get almost optimum solution, which is 0.4% within 3 seconds. But within the same time, the SA can only get 6% at most, under best available parameters and seeds.

For Roanoke city, whose size is relatively much bigger than Toronto city, here we can see SA is better than New 2-Ex, not only on time, but also on precision. Specifically, SA can get 6% precision within 22s for each test. But it usually takes over 100 seconds for New 2-EX to get the similar results.

So what causes these difference? As we said before, despite of parameters and seeds selection, we consider the creations in New 2-EX algorithm. The first thing of creation is greedy search for initial path, the algorithm choose nearest nodes start from second node, end in last node to find the initial path. This process make sure that we can have a good base, much better than just link every node in sequence from the random first node chosen in SA. The second creation is restart mechanism, this is better than SA because there is no such mechanism in SA, which could be added to polish SA algorithm later in the future, by using this algorithm, we have the chance to get more different results, could be either better or worse. The third and fourth creation is k=5 nearest neighbor and DLB strategy, which can make sure that we can have the best candidate node for each run, which in turn leads to almost optimum results.

The interesting thing here is, for smaller size city like Toronto, New 2-EX algorithm works better, and for larger size city like Roanoke, SA works better, where better means that time and precision one algorithm ?are all better than the other algorithm.

And from Boxplot comparison, we can see the reason clearly, as the precision comes to 106%, SA performs much better than New 2-EX, the time of SA extends much less than New 2-EX, and the length is also shorter than New 2-EX algorithm. But before 106% precision, New 2-EX is much better than SA, which is reflected on the length and extending rate of the box.

## 6.3 Comparison between all Five algorithms

In these five algorithms, each one has its own property, however, there is no one will be definitely suitable for all kinds of actual problems. Generally, there is a trade-off between time-consuming and accuracy, and also, sometimes the size of problem also matters. So, for this project, we not only need to analyze the property of each algorithm, we also need to compare these five together, so that in actual problems, we can provide the relative best choice for our customers company.

### 6.3.1 *When time-consuming is the first priority*

We look at the differences between those five algorithms from the output table. When the time consuming is the most big concerns, approximation algorithm will be considered, however, the correctness of the outcome will be not as good as other four algorithm. Comparing MST-approximation and nearest neighbor algorithm, the nearest neighbor approximation runs faster and the RelErr is no worse than MST-approximation, so nearest neighbor algorithm is some way better than MST-approximation. However, as we can see in the result, the branch and bound and 2-exchange-revised algorithm are also very fast in the small and easy problem size.

### 6.3.2 *When accuracy is the first priority*

When time consuming is not a problem, and the correctness is the most important part, branch and bound, 2-EX-revised, and simulated annealing algorithms will be chosen. Also, among those three algorithms, the 2-EX-revised algorithm is the most fast one and the outcomes are more competitive to the other two algorithms. Our team used innovation and optimization for the 2-EX algorithm, which makes it somehow even better than branch and bound algorithm, which theoretically can generate optimal solution in exponential time. Our 2-EX algorithm can also generate optimal solutions for small size of data and the time consuming is far more less than the branch and bound algorithm. But in reality, if the customers really want to sacrifice time to get the optimal solution, it must be branch and bound.

### 6.3.3 *Data size and complexity matters*

Data set size also matters for which algorithms we will use. From our table, we can see that for the Roanoke city contains 230 points, branch and bound can not compute the output in reasonable time. If the data points are extremely large, 2 approximation algorithms and 2 local search algorithms will be recommended. Last but not the least, we found that the running time and results may be relative to the complexity of the problem. For example, comparing Denver to San Francisco, although the size of SF is bigger than Denver, branch and bound can find the optimal solution faster than SF, however, in 600s, Denver cannot find the optimal solution. From the result from bnb and 2-Opt-revised, we can see Denver and Boston are the complex problem, because although the size is small, it is not easy to get a good solution.

### 6.3.4 *Overall worst choice*

As for the worst choice, we think the Approximation algorithms are the worst. Although it can get the results within very short time, but the RelError is pretty huge. Actually, in the actual cases, we may not have such a strict limit on running time, which means in most cases, we may not only have 0.1s second for the results. So, although it is really fast, it may meaningless for us. On the other hand, in small size of problem, local search algorithms are also very fast, and meanwhile, they can get the optimal solutions as well.

### *6.3.5 Overall best choice*

So, from the above three points, we can see the Revised 2-Opt-Exchanges is the overall best choice in this specific TSP problem.

### *6.3.6 Some other unsure factors*

Although in our project, we already got 12 cases to analyze, we still cannot 100% sure about the Revised 2-Opt-Exchange, and we also cannot guarantee in other NPC problem or other cases in TSP problem, the Branch and Bound algorithm can be as good as in our project. For Revised 2-Opt-Exchange algorithm, we already noticed that the SA algorithm can do better in accuracy for Roanoke, so that we cannot sure when the problem size is bigger, whether the SA will be better than Revised 2-Opt-Exchange. Also, for Branch and Bound, the parameter matters, and it happens the M=80 is very good for these 12 cases, but may cannot guarantee other cases to be good like this.So, in the actual problem, we may still need to explore and try and compare.

## 7. CONCLUSIONS

In this report, we introduced five algorithms and their implementations to solve TSP problem, the algorithms are: Branch-and-Bound,Approximation-MST,Approximation Nearest Neighbour,Local Search-2-Opt and Local Search Simulated Annealing.

Among these algorithms, BnB method provides the most precise results which could be regarded as the optimal solution, but this method takes most time.

The Approximation methods are able to give results in a very short time, but the precise level is not as good as BnB and Local Search.

The two Local Search algorithms need more time than Approximation methods in larger size problem, but they can provide more precise results. However, the local search perform better in small size problem.

In the future, there are some optimizations that we can do, such as test more temperature parameters and more seeds to see if we can get better results from Local Search or Approximation methods

## 8. REFERENCES

[1] https://www.seas.gwu.edu/ simhaweb/champalg/tsp/tsp.html;

[2] J. Kleinberg and E. Tardos, Algorithm Design, Addison Wesley, 1st ed., 2005.

[3] https://www.seas.gwu.edu/ simhaweb/champalg/tsp/tsp.html;

[4] S.Kirkpatrick, C.D.Gelatt, and M.P.Vecchi. Optimization by Simulated Annealing. Science, 220 1983, pp.671-680.

[5] Valenzuela, Christine L., and Antonia J. Jones. ”Estimating the Held-Karp lower bound for the geometric TSP.” European Journal of Operational Research 102.1 (1997): 157-175.

[6] Wiener, Richard. ”Branch and Bound Implementations of the Traveling Salesperson Problem - Part 4: Distributed processing solution using RMI.” Journal of Object Technology 2.6 (2003): 51-65.

[7] Rosenkrantz, Daniel J., Richard Edwin Stearns, and Philip M.Lewis. ”Approximate algorithms for the traveling salesperson problem.” Switching and Automata Theory, 1974., IEEE Conference Record of 15th Annual Symposium on. IEEE, 1974.

[8] cemy, Vladimĺr. ”Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm.” Journal of optimization theory and applications 45.1 (1985): 41-51.

[9] Laporte, Gilbert. ”The vehicle routing problem: An overview of exact and approximate algorithms.” European Journal of Operational Research 59.3 (1992): 345-358.

[10] Hwang, Chii-Ruey. ”Simulated annealing: theory and applications.” Acta Applicandae Mathematicae 12.1 (1988):108-111.

[11] Kirkpatrick, Scott, C. Daniel Gelatt, and Mario P. Vecchi.”Optimization by simulated annealing.” science 220.4598(1983): 671-680.

[12] Tsai, Huai-Kuang, et al. ”An evolutionary algorithm for large traveling salesman problems.” IEEE Transactions on Systems,Man, and Cybernetics, Part B (Cybernetics) 34.4(2004): 1718-1729.

[13] Basu S, Ghosh D. A Review of the Tabu Search Literature on Traveling Salesman Problems. Indian Institute of Management. 2008; 10(1):1-16.

[14] Rajan K, Anilkumar. Genetically motivated search algorithm for solving Traveling salesman problem. SB Academic Review. 2009; XVI(1&2):19-31.

[15] Kumar N, Karambir, Kumar R. A Genetic algorithm approach to study Traveling salesman problem. Journal of Global Research in Computer Science. 2012; 3(3):33-8.

[16] Singh A, Narayan D. A Survey Paper on Solving Traveling Salesman problem Using Bee colony optimization. International Journal of Emerging Technology and Advanced Engineering.2012; 2(5):309-14.

[17] Basu S. Tabu Search Implementation on Traveling Salesman Problem and Its Variations: A Literature Survey. American Journal of Operations Research. 2012; 2:163-73.

[18] Hingrajiya KH, Gupta RK, Chandel GS. An Ant Colony Optimization Algorithm for Solving Traveling Salesman Problem. International Journal of Scientific and Research Publications. 2012; 2(8):1-6.

[19] Gupta S, Panwar P. Solving Traveling salesman problem using genetic algorithm. International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE). 2013; 3(6):376-80.

[20] Sathya, N., and A. Muthukumaravel. ”A Review of the Optimization Algorithms on Traveling Salesman Problem.” Indian Journal of Science and Technology 8.29 (2015).

[21] Khattar S, Gosawmi P. A Solution of Genetic Algorithm for Solving Traveling Salesman Problem. International Journal for Scientific Research and Development (IJSRD). 2014; 2(4):341-3.

[22] http://www.cs.ubc.ca/labs/beta/ Courses/CPSC532D-05/Slides/tsp-camilo.pdf