CSE 6140 - Team J Final Report – TSP Problem

Sheng-Chuan Hwang shwang89 @gatech.edu
Xiaoyu Yang xiaoyushowyou @gatech.edu
Yunhe Song ysong346 @gatech.edu
Xin Cao xincao @gatech.edu

ABSTRACT
In this project, we will attempt to solve the TSP by using different algorithms, evaluating their theoretical and experimental complexities on real and random datasets.

Keywords
Local Search, 2 Opt Algorithm, 3 Opt Algorithm, Minimum Sales Man Problem, Branch and Bound Algorithm, Approximation Algorithm

1. INTRODUCTION
The Traveling Salesman Problem (TSP) arises in numerous applications such as vehicle routing, circuit board drilling, VLSI design, robot control, X-ray crystallography, machine scheduling and computational biology. The decision version of this problem is a NP-complete [4] in computational complexity theory. The worst running time to solve the problem is exponential. Various approaches are made to deal with this problem including the branch and bound algorithm, approximation algorithms and local search algorithms.

2. Branch and Bound Algorithm
We first implement an exact algorithm to implement the TSP problem using branch and bound concept.

2.1 Lower Bound Computation
The lower bound we use is a very simple rule from TSP, since we need to visit all the nodes, every node needs to be entered and exit once. Hence, the initial lower bound of a tour will be the sum of the lowest two edges of each node and then divided by two. A simple example will be as following:

Table2-1. Cost Matrix example

Node	1	2	3
1	0	2	3
2	2	0	1
3	3	1	0

The lower bound of the above example will be 2+1+1+1=5. The computation can be done by finding the smallest value of each row and column, deduct such value to given row(column), then sum up the smallest value.

Table2-2. Deducted the smallest value of each row

Node	1	2	3
1	0	0	1
2	1	0	0
3	2	0	0

Now, the sum of the smallest value of all three row is 2+1+1=4. Similarly, we do the same thing to each column.

Table2-3. Deducted the smallest value of each column

Node	1	2	3
1	0	0	1
2	0	0	0
3	1	0	0

Hence, the total lower bound will be 4+1=5. We can expand this idea to any given edge is in the tour or not in the tour. For example, we can compute the lower bound for a tour that contains edge (1,2). We cross out the first row and second column if (1,2) is in the tour, then assign the cost from node 2 to 1 infinite. Then compute the new lower bound, in this case, the lower bound remains the same.

Table2-4. Cost Matrix of edge (1,2) is in the tour

Node	1	~~2~~	3
~~1~~	~~0~~	~~0~~	~~1~~
2	infinite	~~0~~	0
3	1	~~0~~	0

If edge (1,2) is not in the tour, we simply assign the cost from node 1 to 2 infinite. Then compute the new lower bound again, in this case, the lower bound remains the same.

Table2-5. Cost Matrix of edge (1,2) is not in the tour

Node	1	2	3
1	0	infinite	1
2	0	0	0
3	1	0	0

2.2 Branch and Bound
After we have a lower bound for a tour, we can construct a solution tree and prune the solution that is higher than the current optimal solution. For a solution tree, each step decides whether an edge should be in the tour or not. Compute the lower bound and compare to the current optimal value we found, if the lower bound is higher than the current optimal, that means the solution expand by such situation will never be smaller to our current optimal, so we can prune it. If not, we will expand this partial solution. The idea can be illustrated as the graph below:

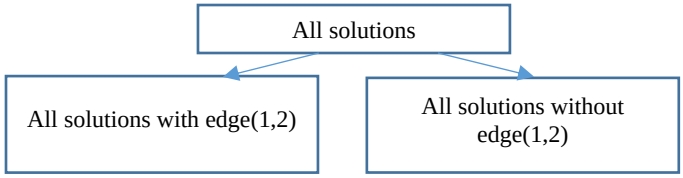


Figure2-1. Solution tree example

2.3 Choice of expansion

The choice of which edge should be expand can effect the performance of the algorithm. Due to the time issue, we will come up with an algorithm simply by choosing in lexicographic order. We expect that this strategy will slow down our algorithm.

2.4 Pseudo Code

The concept we use to implement this algorithm is recursion. The following is the pseudo code to explain how we implement:

Algorithm 1: Branch and Bound

```
if found a tour:
    if current tour length < best tour found now:
        best tour found now = current tour
else:
    pick an edge (u,v)
    recursive call: tour include edge(u,v)
    recursive call: tour not include edge(u,v)
return best tour
```

More detailed contents can be reviewed in our code. There are a lot of trivial conditions and situations need to be handled in the implement code, which makes it complicated.

2.5 Result

We run all datasets using the branch and bound algorithm, setting the cut of time as 500 seconds. The following table shows our result.

Table2-6. Cost Matrix of edge (1,2) is in the tour

Dataset	Time	Length	RelErr%	# of nodes
Cincinnati	0.13	277952	0%	10
UKansasState	0.27	62962	0%	10
Atlanta	411.98	2003763	0%	20
Boston	340.24	2063922	1.31%	40
Philadelphia	530.54	3331276	1.39%	30
Champaign	404.67	185418	2.52%	55
NYC	430.01	6869984	3.42%	68
UMissouri	576.45	615305	3.64%	106
Denver	490.44	513066	4.11%	83
SanFrancisco	576.45	5352912	5.61%	99
Toronto	370.28	8809227	6.49%	109
Roanoke	451.85	6767159	9.32%	230

2.6 Analysis and Conclusion

From the result above, our branch and bound algorithm can compute graph contains 20 nodes under 500 seconds. However, as the nodes increases, the time grows in exponential (as we know this is a NP-Complete Problem), hence for graph over 20 nodes, the time needed is unendurable.

There are still some further potential improvements for our algorithm, such as the choice to expand an edge. Now our algorithm is only using lexicographic order to expand, but there are several heuristic ways to choose, such as choosing the lowest current lower bound edge. Also, we can also make a smarter choice in which include this edge should be expanded first or not include this edge should be expanded. Our algorithm always first expand including such edge, however, we can still compute both situation's lower bound and compute the smaller one first.

In conclusion, even we use the simplest concepts to implement branch and bound algorithm, with the help of data structures and recursive calls, the result is acceptable. We can compute exact optimal solution with 20 nodes under 10 minutes.

3. Approximation Algorithm

3.1 MST-Approximation

Implement the 2-approximation algorithm based on MST detailed in lecture.

We use the 2-approximation MST algorithm to solve the metric-TSP. The basic strategy is to construct a MST on the given graph G=(V,E). Euler did prove that the necessary condition for a Euler path existing is that all of the vertices should have even number of edges connected [2]. Because of this, we naturally design a method that double all of the edges in the MST of the graph with a double cost of that of the MST, which is a lower bound of TSP.

The second step is to decide a root node or start node such that we can find out a Eulerian tour based on the MST. Since the Eulerian tour requires that each vertex can only be gone through once expect for the start node, we need to make a shortcut edge if our tour can only arrive a repeated vertex along the MST. Fortunately, because our TSP is metric, which satisfies the triangle inequalities, we can guarantee that the shortcut edge is not longer that the repeated path. We can repeat this process to develop the Eulerian tour until it is finally completed.

The algorithm and pseudocode is:

2-approximation MST algorithem:

Input: the given graph G=(V,E)

Output: the Metric TSP solution (Eulerian tour based on MST)

Construct a MST on the graph G;

Select a root vertex;

While (Eulerian tour is not competed):

Using DFS method to search the tree and output each node by the order we search;

If a vertex is encountered repeatedly:

Make shortcut edge and calculate its weight by Euclidean distance definition

Endif

end

Return the Eulerian tour (metric-TSP solution)

The results:

Table 3-1. Nearest Neighbor

Datasets	Times(ms)	Length	RelErr
Atlanta	9.00	2407650	0.202
Boston	57	1047604	0.172
Champaign	61	61480	0.168
Cincinnati	10	296963	0.068
Denver	441	124952	0.244
NYC	86	1825221	0.174
Philadelphia	11	1708688	0.224
Roanoke	1043	800407	0.221
SanFrancisco	856	1078768	0.331
Toronto	539	1608107	0.367
UKansasState	2	66523	0.057
UMissouri	942	153702	0.158

3.2 Nearest Neighbor

We use the nearest neighbor method in calculate the approximated TSP. The nearest neighbor algorithm was one of the first algorithms used to determine a solution to the travelling salesman problem. In it, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.

The nearest neighbor algorithm is easy to implement and executes quickly, but it can sometimes miss shorter routes which are easily noticed with human insight, due to its "greedy" nature. As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that there are much better tours. Another check is to use an algorithm such as the lower bound algorithm to estimate if this tour is good enough.

In the worst case, the algorithm results in a tour that is much longer than the optimal tour. To be precise, for every constant r there is an instance of the traveling salesman problem such that the length of the tour computed by the nearest neighbor algorithm is greater than r times the length of the optimal tour. Moreover, for each number of cities there is an assignment of distances between the cities for which the nearest neighbor heuristic produces the unique worst possible tour.[1].

These are the steps of the algorithm:

1 start on an arbitrary vertex as current vertex.
2 find out the shortest edge connecting current vertex and an unvisited vertex V.
3 set current vertex to V.
4 mark V as visited.
5 if all the vertices in domain are visited, then terminate. Go to step 2.

Algorithm 3: Nearest Neighbor

$C \leftarrow \phi$

$V_{current}$ = vertex 1

$V_{current} \leftarrow C$

While |C| < |V| **do**

Find the nearest vertex to $V_{current}$ in (V-C) = $V_{nearest}$

$V_{nearest} \leftarrow C$

$V_{current} = V_{nearest}$

End While

Return C

The results are shown in the table below. We can see that the relative errors are never larger than 1, which coincides with the nearest algorithm is a 2- approximation. The relative errors are not so significant actually are not far away from the optimal solution.

Table 3-2. Nearest Neighbor

Datasets	Times(ms)	Length	RelErr
Atlanta	3.81	2039905	1.8%
Boston	82.23	1029008	15.2%
Champaign	46.80	61828	17.4%
Cincinnati	0.63	301258	8.4%
Denver	1487.45	117618	17.1%
NYC	219.02	1796652	15.5%
Philadelphia	4.75	1611716	15.4%
Roanoke	6194.35	777638	18.6%
SanFrancisco	2352.76	857731	5.9%
Toronto	1503.71	1243369	5.7%
UKansasState	0.26	69986	11.1%
UMissouri	577.94	155304	17.0%

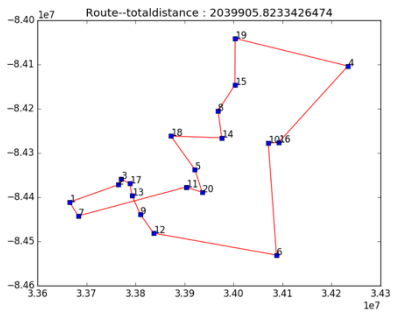


Figure 3-1. Route for Atlanta in Nearest Neighbor algorithm.

4. Local Search

4.1 Description of Stochastic Hill Climbing

The algorithm we used for local search is hill climbing algorithm. By starting from an arbitrary solution, the hill climbing can iteratively go through other solutions and try to find a better one by incrementally changing a single element of the solution. The algorithm keep changing the solution until no further improvements can be found. By using the stochastic hill climbing, we keep trying different neighbors of the candidate solution until we get the neighbor which has the biggest improvement for the solution. However, by using stochastic hill climbing, we can only get the optimal solution for local minimum, which may not be the global minimum. Therefore, it is necessary for hill climbing to start from different seeds and restart another local search.

Different ways can be applied to escape the local minima. First, one can try random restart hill climbing. Once a local minimum distance is reached, we restart at another random position. We use random generator to produce the random starting point and begin a new local search.

Except for the starting point, another factor that influences the performance of local search algorithm is the neighborhood that choose. We use different neighborhood for our hill climbing. First one is 2_Opt. Another one is 3_Opt.

4.2 2_Opt Local Search

The algorithm used for local search is 2 Opt heuristic. The main idea for this algorithm is to check whether by deleting 2 edges and reconnecting the fragments into 1 cycle can improve the performance of the local search. If the swap can improve the performance, then the endpoints are swapped and a new circle is created. The details can be seen as Fig.2.

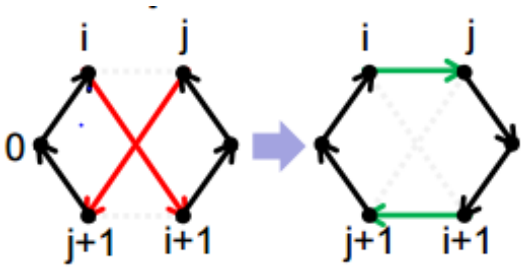


Fig.4-1 2_Opt Algorithm

For the original circle, the total distance with edges (N(i), N(i+1)) and (N(j),N(j+1)) can be calculated. By swapping the end points, we can get a new circle with the edges of (N(i), N(j)) and (N(i+1), N(j+1)). If the total distance of the original circle is greater than the distance after swapping, it means that the swap can improve the performance. Then the new order of nodes will be recorded and the total distance will be updated.

These are the steps of the algorithm:

1 Find starting tour which is depended on the input seed. In study, the seed will decide which is the initial point of the exploring and the initial order is same for all of the cases, which follows the order of the data file that was given.
2 Remember the order of nodes and the total distance.
3 Swapping end points to get a new circle
4 It the swap can decrease the total distance. Then I update both of the order and the total distance.
5 Repeat the process until on improvement.

Algorithm 4: 2_Opt Algorithm

```
T = starting tour (0,1,…,n)
for all possible edges in T
    for all of the rest edges of T-E(i,i+1)
        T' = tour by swapping end points
        if T' < T
            T = T'
            Reverse the points between the nodes i and j+1
            break
        endif
    endfor
endfor
Repeat until no improvement
return T
```

By using 2 Opt algorithm, for each nodes i, the edge created by i and i+1 need to be compared to the rest of edges. As a result, it is totally 2 “for” loop for the algorithm. For each situation, the algorithm only need to compare the edges before and after swap, which takes polynomial time. So the time complexity for the 2_ OPT algorithm is $O(n^2)$.

For the space required of the algorithm, the data which needs to be remembered are the ordered nodes information and total distance, which takes linear space O(n).

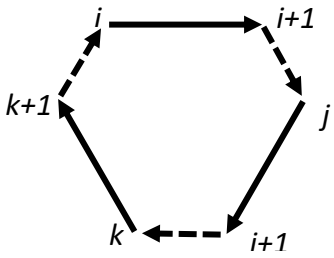
The potential strengths for the 2 Opt are: 1, Sometimes are much more efficient than other algorithm; 2, easy to implement, 3, easy to parallelize.

The weaknesses are: 1, no guarantees for finding existing solutions; 2, highly stochastic behavior; 3, often difficult to analyze theoretically.

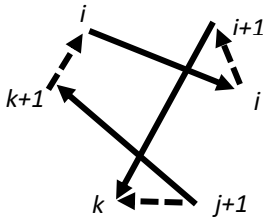
4.3 3_Opt Local Search

The second local search algorithm used for this project is 3_Opt heuristic. The main idea of this algorithm is same as 2_Opt heuristic. By deleting 3 edges and reconnecting them into 1 cycle, it is able to develop new possible solutions. If the swap can improve the performance, then I swap the endpoints and create a new circle.

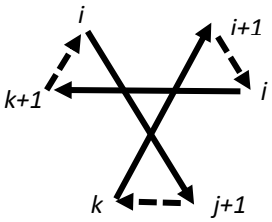
The main difference between 3_Opt and 2_Opt is that 3_Opt has much more possible swap alternatives than 2_Opt. Except the three 2_Opt solutions by changing either two points in the three, there are four more possible alternatives by exchanging all of the three nodes, which are shown in Figure 3.



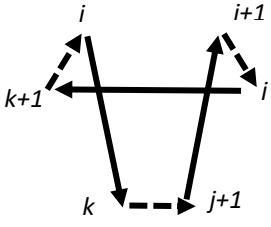
3(a). Original Graph



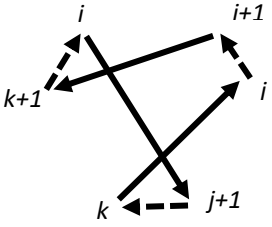
3(b). Possible alternative 1



3(c). Possible alternative 2



3(c). Possible alternative 3



3(d). Possible alternative 4

Fig.4-2 Possible swap alternatives for 3 Opt Algorithm (Except for 3 alternatives by using 2 Opt)

For the original circle, the total distance with edges (N(i), N(i+1)), (N(j),N(j+1)) and (N(k),N(k+1)) can be calculated. By swapping the end point, we can get a new circle with updated total distance information. If the total distance of the original circle is greater than the total distance after swapping, it means that the swap can improve the performance. Then the new order of nodes will be recorded and the total distance will be updated. Due to there are total 7 possible alternatives for 3 Opt algorithm, I choose the biggest improvement as the solution and use the order as the start of next change iteration.

These are the steps of the algorithm:

1 Find starting tour which is depended on the certain seed. In study, the seed will decide on which is the initial point of the exploring and the initial order is same for all of the cases, which follows the order of the data file that was given.
2 Remember the order of nodes and the total distance.
3 Swapping end points to get a new circle for 7 alternatives.
4 It the swap can decrease the total distance, the alternative which has the biggest improvement will be used. After that, I update both of the order and the total distance.
5 Repeat the process until no improvement.

Algorithm 5: 3_Opt Algorithm

```
T = starting tour (0,1,…,n)
for all possible edges in T
    for all of the rest edges of T-E(i,i+1)
        for all of the rest edges of T-E(i,i+1)-E(j,j+1)
            T' = tour by swapping end points
            if T' < T
               T = T'
                  Reverse the points between the nodes i and j+1
                  break
            endif
        endfor
    endfor
endfor
Repeat until no improvement
return T
```

By using 3 Opt algorithm, for each nodes i, the edge created by i and i+1 need to be swapped with two other edges among the rest edges. As a result, it is totally 3 "for" loop for the algorithm. For each situation, the algorithm only need to compare the edges before and after swap, which takes polynomial time. So the time complexity for the 3 OPT algorithm is $O(n^3)$.

For the space required by the algorithm, the data that needs to be remembered are the ordered nodes information and total distance, which takes linear space O(n).

The potential strengths for the 3 Opt are: 1, Sometimes are much more efficient than other algorithm; 2, easy to implement, 3, easy to parallelize.

The weaknesses are: 1, no guarantees for finding existing solutions; 2, highly stochastic behavior; 3, often difficult to analyze theoretically.

4.4 Results and Figures

By using the 2_Opt and 3_Opt algorithm that developed right now. The analyzed time and distance results for the project can be seen in Table 4-1 and Table 4-2 separately. We got the results of all of the graphs within 10 minutes with the initial seed from 0. However, due to there are too many nodes in the dataset of Roanoke, by trying different seeds, there are some cases where it can't be solved by using the 3_Opt algorithm under the cutoff time.

From Table 4-1 and Table 4-2 we can find that 3_Opt algorithm has less RelErr than 2_Opt when use the same initial seed. This is mainly because there are much more conditions for 3_Opt algorithm which makes it can search much more possible solutions that 2_Opt. However, due to there are much more edges need to check, 3_Opt requires more time than 2_Opt. For certain cases, such as Roanoke database, due to the nodes quantity is too big, by using 3_Opt may not get the solution in the cutoff time.

Table 4-1. 2OPT Algorithm

Dataset	Time	Length	RelErr
Atlanta	0	2003762.67	0.00%
Boston	3	895932.57	0.27%
Champaign	5	52787.98	0.28%
Cincinnati	0	277952.59	0.00%
Denver	46	102771.98	2.33%
NYC	24	1562703.92	0.49%
Philadelphia	1	1395980.78	0.00%
Roanoke	368	675830.45	3.11%
SanFrancisco	8	812472.1	0.28%
Toronto	37	1176151.22	0.00%
UKansasState	0	62962.31	0.00%
Umissouri	66	134900.35	1.65%

Table 4-2. 3OPT Algorithm

Dataset	Time	Length	RelErr
Atlanta	0	2003762.67	0.00%
Boston	2	894155.61	0.07%
Champaign	80	52642.54	0.00%
Cincinnati	0	277952.59	0.00%
Denver	287	101102.02	0.67%
NYC	43	1558107.6	0.20%
Philadelphia	0	1395980.78	0.00%
Roanoke	434	665810.46	1.58%
SanFrancisco	511	811241.6	0.13%
Toronto	37	1176151.22	0.00%
UKansasState	0	62962.31	0.00%
Umissouri	306	134074.46	1.03%

The QRTD, and STD of Roanoke and Toronto by using 2 Opt and 3 Opt algorithms are listed below in Figures 4-3-4-10. By looking into the plots, we can find out some interesting results. For very big dataset, such as Roanoke, both 2_Opt and 3_Opt method can not find the general optimal solutions. It takes longer time for 3_Opt to get same precision performance. Same as for SQD plot, with the same time settings, 2_Opt can achieve the same solution quality as 3_Opt in a much shorter time. However, with the increasing of time, 3_Opt is able to produce much more optimal solutions than 2_Opt. Meanwhile, 3_Opt can produce better solutions by using longer time.

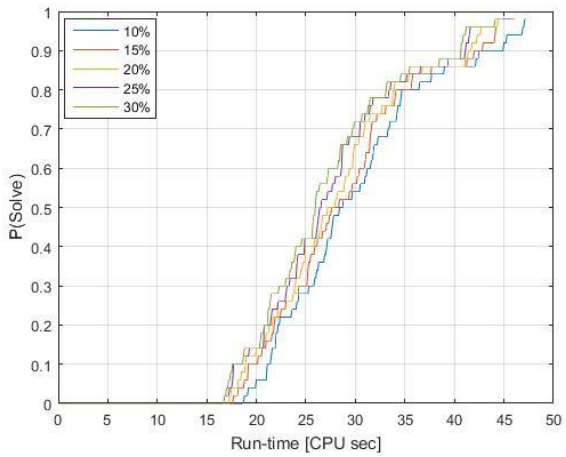


Figure 4-3. QRTD plot of Roanoke by using 2Opt local search

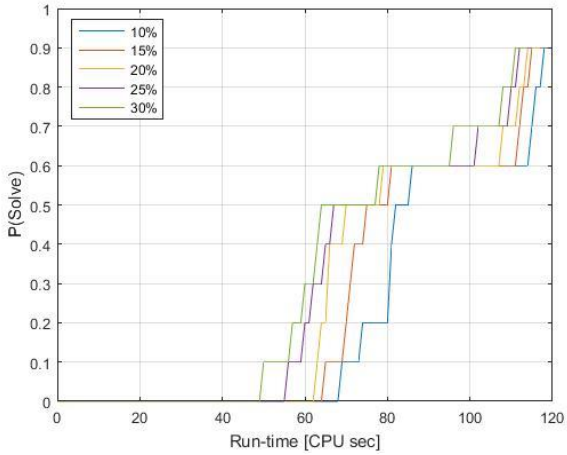


Figure 4-4. QRTD plot of Roanoke by using 3Opt local search

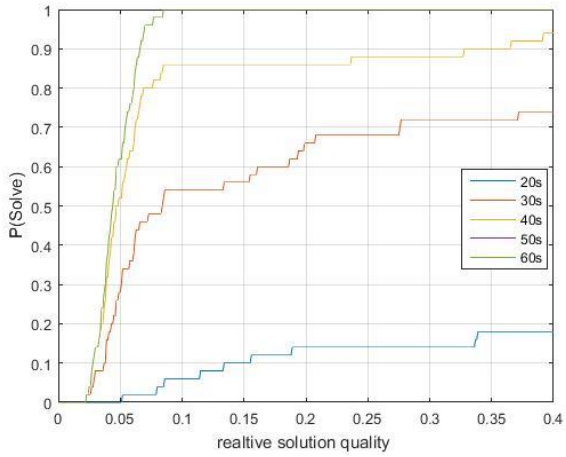


Figure 4-5. SQD plot of Roanoke by using 2Opt local search

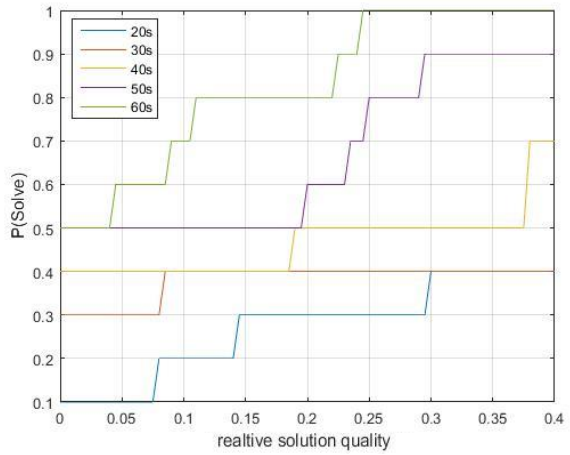


Figure 4-6. SQD plot of Roanoke by using 3Opt local search

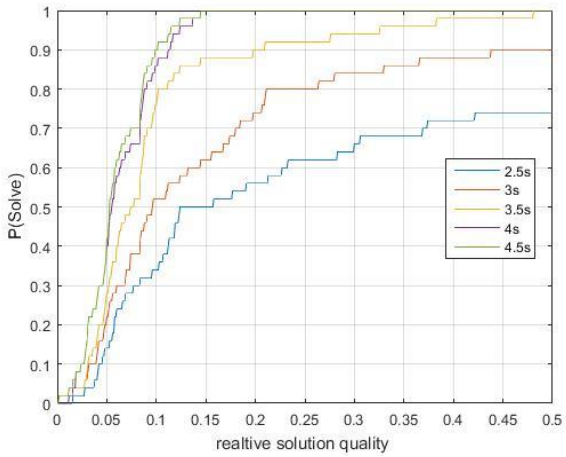


Figure 4-9. SQD plot of Toronto by using 2Opt local search

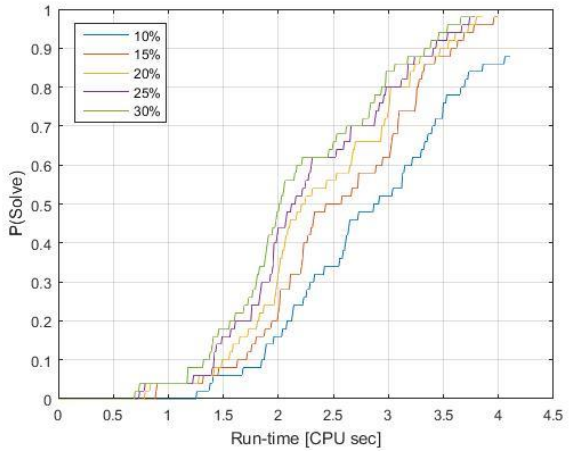


Figure 4-7. QRTD plot of Toronto by using 2Opt local search

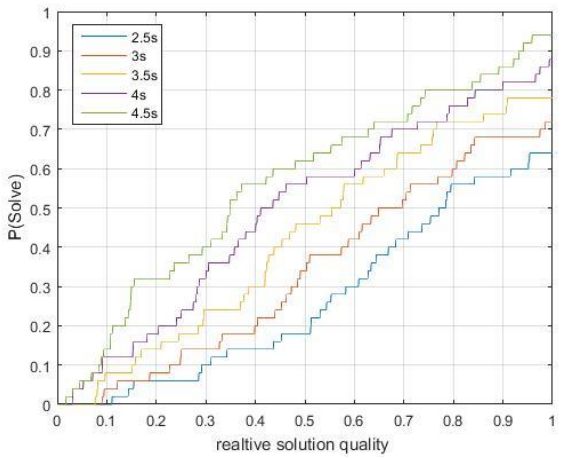


Figure 4-10. SQD plot of Toronto by using 3Opt local search

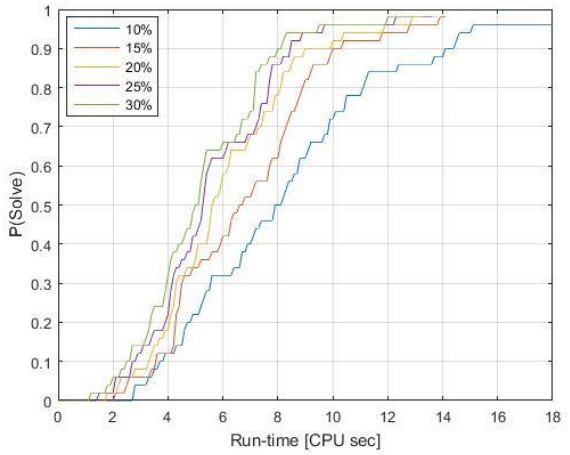


Figure 4-8. QRTD plot of Toronto by using 3Opt local search

The box plots of both Roanoke and Toronto data set by using 2_Opt and 3_Opt algorithms are listed below as Figure 4-9 and Figure 4-10. It is clear that 3_Opt takes much more calculation time than 2_Opt under same precision requirement. Meanwhile, the dispersion of the time is bigger by using 3_Opt. By comparing the results of Roanoke and Toronto, we can find it by increasing the nodes number of the dataset, the calculation time increasing dramatically because both algorithms are not polynomial. Compared 2_Opt, 3_Opt increases faster due to $O(n^3)$ calculation time.

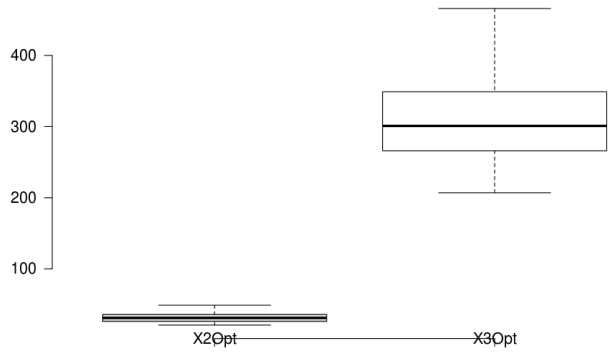


Figure 4-11. Box plot of Roanoke by using 2Opt & 3Opt local search (15% Precision)

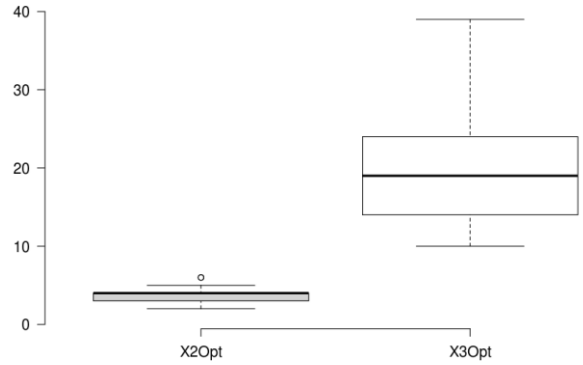


Figure 4-12. Box plot of Toronto by using 2Opt & 3Opt local search (10% Precision)

4.5 Discussion

For our Branch and Bound algorithm, we can compute exact optimal solution with 20 nodes under 10 minutes, which we think is acceptable. There are still some further potential improvements for our algorithm, such as the choice to expand an edge. Now our algorithm is only using lexicographic order to expand, but there are several heuristic ways to choose, such as choosing the lowest current lower bound edge. Also, we can also make a smarter choice in which include this edge should be expanded first or not include this edge should be expanded. Our algorithm always first expand including such edge, however, we can still compute both situation's lower bound and compute the smaller one first.

We used the MST approximation method to get a very fast algorithm, which set up a minimum spanning tree first, and then use the depth first search to find a relative good solution. The solution is acceptable and runs very fast, so that we can choose multiple different initial start node to get better solution. This method could be applied to the situation that requires high efficiency. Another approximation method nearest neighbor algorithm works almost the same as MST, this method is quite easy to understand, start from a random node and find its nearest neighbor, and for this neighbor, find its nearest neighbor and repeat this process for every new neighbor. In conclusion, both MST and nearest neighbor are quite fast method, however, their accuracy is not that good, as you can see from the table, the relative error is quite large. We can't say that the approximation method does not work, it can, but not that perfect when compared with BNB and LS.

We have applied the local search algorithm by using the hill climbing method. By using 2_Opt and 3_Opt, we can look into different neighbors. Compared to approximation, LS takes much longer time to get the solution for each seed. However, the results can be improved dramatically. 2_Opt takes O(n2) to process the data, where 3_Opt takes O(n3). By testing, we can find that the relative error for LS is really low. And by using 3_Opt, the results can be increased dramatically.

An obvious improvement may be made by start with a better approximation solution by using 2_Opt, and then apply the result as the lower bound for BnB. Due to the time limitation, we didn't apply this method yet. However, based on the performance of both algorithms, we can predict the improvement and will apply this method in the future cases.

5. REFERENCES

[1] Frieze, A. M. (1985). On the value of a random minimum spanning tree problem. Discrete Applied Mathematics, 10(1), 47-56.G.

[2] Gutin, A. Yeo and A. Zverovich, Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. Discrete Applied Mathematics 117 (2002), 81-86

[3] Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. Artificial Intelligence, 175(9):1672–1696, 2011