CSE6140 Fall 2018 Project
Traveling Salesperson Problem (TSP)

Implementation and Evaluation of Four Algorithms

Dongmin Han
Mechanical Engineering
hdmhust@gmail.com

Shan Xiong
Mechanical Engineering
sxiong33@gatech.edu

Chong Ye
Mechanical Engineering
chongtt.ye@gmail.com

Yuanlai Zhou
Computational Science & Engineering
yuanlai.zhou@gmail.com

ABSTRACT

In this paper, we adopted four different algorithms to find optimal or "good" solution to this problem, including branch and bound, construction heuristics with approximation guarantees (MST-approx), local search with simulated annealing (SA) and generic algorithm (GA). We have also evaluated their theoretical and experimental complexities on real world instances.

To get optimal solutions, we used penalty-based selection strategy in the Branch-and-Bound algorithm to obtain good quality of lower bound. Over half of all the given cases have been solved within 1.5 hours. Approximation algorithm in all the given cases takes less than 0.1 sec per case and can achieve solution qualities better than 40%. Local search algorithms are generally fast and can produce satisfactory results with carefully optimized parameters. All the given cases can be solved with qualities better than 15% within 2 minutes in our experiments with GA and SA algorithms.

KEYWORDS

Traveling salesperson problem, Branch and bound, Approximation, Heuristics, Local search, Generic algorithm, Simulated annealing

1 INTRODUCTION

The traveling Salesperson Problem (TSP) is a well-known NP-complete problem with numerous applications, such as transportation, planning, and logistics. In this study, we implemented four algorithms to solve the TSP with instances from real world database. These algorithms include branch-and-bound (BnB), construction heuristics (approximation) and two versions of local search (LS). All four algorithms are successfully implemented and evaluated with a systematic empirical analysis.

2 PROBLEM DEFINITION

Given the $x-y$ coordinates of N points in the plane (i.e., vertices) and a cost function $c(u,v)$ defined for every pair of points (i.e., edge), find the shortest simple cycle that visits all N points.

The cost function $c(u,v)$ is defined as either the Euclidean or Geographic distance between points u and v. Specifically, for the **EUC_2D** format, we have $x-y$ coordinates of u and v, which are (u_x, u_y) and (v_x, v_y), respectively. Then we can easily calculated the cost function (pairwise distance) by the following equation:

$$c(u,v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2} \tag{1}$$

The results will be rounded to the nearest integer for the following computation.

For **GEO** format, the $x-y$ coordinates (latitude and longitude) will first be converted to radii. Then we take the floor of x and y coordinates. The distance d_{ij} between points i and j can be calculated by the following steps:

- Step 1. Converting to radius

$$\pi = 3.141592;$$
$$deg = \lfloor x_i \rfloor;$$
$$min = \pi(deg + 5/3min)/180$$

- Step 2. Calculating the geographical distance

$$q_1 = \cos(longitude[i] - longitude[j]);$$
$$q_2 = \cos(latitude[i] - latitude[j]);$$
$$q_3 = \cos(latitude[i] + latitude[j]);$$
$$d_{ij} = \lfloor R \times \arccos[0.5q_2(1.0+q_1) - 0.5q_3(1.0-q_1)] + 1.0 \rfloor$$
$$(R = 6378.388km)$$

The proposed solution will provide a valid simple cycle that visits all the N points, together with its total length. An optimal solution will give the cycle with shortest total length.

3 RELATED WORK

3.1 Exact Algorithm: Branch-and-Bound

In 1963, John D. C. Little et al [1] proposed a Branch-and-Bound algorithm for solving the traveling salesman problem. The set of all tours (feasible solutions) is broken up into increasingly small subsets by a procedure called branching. For each subset a lower bound on the length of the tours therein is calculated. Eventually, a subset is found that contains a single tour whose length is less than or equal to some lower bound for every tour. The motivation of the branching and the calculation of the lower bounds are based on ideas frequently used in solving assignment problems. Computationally, the algorithm extends the size of problem that can

reasonably be solved without using methods special to the particular problem. This is also one of the algorithms that we adopted in this study.

In 2014, M. Battarra *et al* proposed three mathematical formulations as exact algorithms for the TSP [2]. The first algorithm is a branch-and-cut approach and is based on a compact formulation in which two sets of two-index binary variables and a polynomial number of constraints are employed. When strengthened with subtour elimination constraints and trivial constrains, this method is not empirically dominated by the best formulation proposed by Rakke et al [3]. The second algorithm is also a branch-and-cut, but the underlying formulation considers three index variables. It is proven to dominate the first method, both theoretically and empirically. The last method is a branch-cut-and-price and is based on a Dantzig-Wolfe decomposition of the second formulation. Columns are generated by dynamically introducing *ng*-paths to the formulation. This method is capable of solving to optimality all the benchmark instances from the literature.

Exact approaches to solving the TSP are successfully adopted only for relatively small problem sizes, although they can guarantee optimality based on different techniques. They usually use algorithms that generate both a lower and an upper bound on the true minimum value of the problem instance. If the upper and lower bound coincide, a proof of optimality is achieved. Due to the nature of TSP, most common solutions to the problem were found to run feasibly only for a graph with small number of nodes. Not much research was encountered in the survey over problem space analysis of the Traveling Salesman problem.

3.2 Construction Heuristics with Approximation Guarantees

Golden and Stewart proposed the CCAO algorithm in 1985. This heuristic was designed for symmetrical Euclidean TSPs. It exploits a well-known property of such problems, namely that in any optimal solution, vertices located on the convex hull of all vertices are visited in the order in which they appear on the convex hull boundary (Flood, 1956). One major drawback of the CCAO algorithm is that its insertion phase is myopic in the following sense: since insertions are executed sequentially without much concern for global optimality, they may result in a succession of bad decisions that the post-optimization phase will be unable to undo.

In 1992, Gendreau, Hertz and Laporte proposed the GENIUS algorithm to improve this type of algorithm. GENIUS executes each insertion more carefully, by performing a limited number of local transformation of the tour, simultaneously with the insertion itself. It consists of two parts: a generalized insertion phase, followed by a post-optimization phase that successively removes vertices from the tour and reinserts them, using the generalized insertion rule.

This algorithm has been extensively tested on randomly generated problems and on problems taken from the literature; all these problems were symmetrical and Euclidean. Tests revealed that GENIUS produces in shorter computing times better solutions than CCAO, superior to all tour construction heuristics developed in this section. This algorithm also appears to compare favourably to tabu search and simulated annealing, although the number of comparisons was more limited in the case of these two methods.

3.3 Local Search

In 2008, S. Basu et al [4] presented a survey of Tabu search approach for solving the TSP. As per the literature survey, the tabu search is said to be most widely used Meta heuristic procedures to solve combinatorial optimization problems. It is an improvement heuristic based on local search. It starts with an initial solution to the problem, (a tour in case of the TSP), calls it a current solution, and searches for the best solution in a suitably defined neighborhood of the solution. It then designates the best solution in the neighborhood as the current solution and starts the search process again. Tabu search terminates when certain terminating conditions, either involving execution time or maximum iteration count conditions, or solution quality objectives, or both, have been met.

In 2011, Z. Hlaing [5] presented an approach for solving the TSP based on improved ant colony algorithm. The main contribution of this work is a study of the avoidance of stagnation behavior and premature convergence by using distribution strategy of initial ants and dynamic heuristic parameter updating based on entropy. Then a mergence of local search solution is provided. The experimental results and performance comparison showed that the proposed system reaches the better search performance over ACO algorithms do.

In 2012, V. Dwivedi et al [6] proposed a new crossover operator for a genetic algorithm to solve the TSP. The crossover is the important stage in the genetic algorithm. A new crossover method called Sequential Constructive Crossover (SCX) operator is adopted. The SCX uses best edges of the parentâĂŹs structure and produces the new offspring. It is compared against other existing crossover operators and it is proved that SCX results in a high quality solutions. This paper also includes a comparative study on Greedy Approach, Dynamic Programming and Genetic Algorithm for solving TSP.

4 ALGORITHMS

4.1 Branch-and-Bound

4.1.1 Introduction of BnB.

A branch-and-bound (BnB) algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded by using upper and lower estimated bounds of the quantity being optimized. The BnB strategy divides a problem into a number of sub-problems. It is a system for solving a sequence of subproblems, each of which may have multiple possible solutions and where the solution chosen for one sub-problem may affect the possible solutions of later sub-problems.

Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solutions among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all.

Let S be some subset of solutions. $LB(S)$ = a lower bound on the cost of any solutions $\in S$. Let C = cost of the best solution found so far. Then if $C \leq LB(S)$, there is no need to explore S since it does not contain any better solutions. If $C > LB(S)$, then we need to continue exploring S.

4.1.2 *Outline of BnB.*

We adopted an algorithm proposed by John D. C. Little et al [1] in 1963. We will use the following example[7] to show the outline of this algorithm. Consider a TSP within five cities (vertices), which form an undirected, connected graph. The following Cost Matrix shows the distance between the five cities.

$$\text{Cost Matrix} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} - & 10 & 8 & 9 & 7 \\ 10 & - & 10 & 5 & 6 \\ 8 & 10 & - & 8 & 9 \\ 9 & 5 & 8 & - & 6 \\ 7 & 6 & 9 & 6 & - \end{pmatrix} \end{array}$$

As discussed in the Algorithm class, we first reduce the matrix using the minimum number in each row and each column. The sum of the row and column minimum will give us the lower bound, which is $7 + 5 + 8 + 5 + 6 = 31$ (row), $31 + 1 = 32$ (row + column). After this reduction, we obtain the reduced matrix as follows:

$$\text{Reduced Matrix} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} - & 3 & 0^{(2)} & 2 & 0^{(1)} \\ 5 & - & 4 & 0^{(1)} & 1 \\ 0^{(1)} & 2 & - & 0^{(0)} & 1 \\ 4 & 0^{(1)} & 2 & - & 1 \\ 1 & 0^{(0)} & 2 & 0^{(0)} & - \end{pmatrix} \end{array}$$

Note that in the reduced matrix, for each *Zero*, a penalty number is labeled in the bracket. This penalty number is calculated by adding up the lowest numbers (other than this *Zero*) from the same row and column as this particular *Zero*. This means if we don't make the assignment in this *Zero*, we will incur an additional cost equals to this penalty. Consider X_{13} since it has the highest penalty. If we don't have the path from 1 to 3, i.e., $X_{13} = 0$, then we will have an additional cost of 2 and the lower bound becomes $32 + 2 = 34$.

On the other hand, if we have the path from 1 to 3, i.e., $X_{13} = 1$, we will eliminate the respective row and column to explore the next path within the remaining cities (vertices). The remaining matrix is as follows:

$$\text{Remaining Matrix} = \begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 4 & 5 \end{array} \\ \begin{pmatrix} 5 & - & 0 & 1 \\ - & 2 & 0 & 1 \\ 4 & 0 & - & 1 \\ 1 & 0 & 0 & - \end{pmatrix} \end{array}$$

Do the reduction for column 1 and column 5, and update the current lower bound to $32 + 1 + 1 = 34$. Calculate the penalties for all the *Zero*$'s$ in the newly reduced matrix, select the one with highest penalty 3, which is X_{51}. If $X_{51} = 0$, lower bound will be updated to $LB = 34 + 3 = 37$. If $X_{51} = 1$, eliminate row 5 and column 1 with the $LB = 34$ unchanged. The remaining matrix becomes:

$$\text{Remaining Matrix} = \begin{array}{c} \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 4 & 5 \end{array} \\ \begin{pmatrix} - & 0^{(0)} & 0^{(0)} \\ 2 & 0^{(2)} & - \\ 0^{(2)} & - & 0^{(0)} \end{pmatrix} \end{array}$$

No more reduction is needed and the penalties are labeled in the matrix. Consider X_{34} since it has the highest penalty. If $X_{34} = 0$, $LB = 34 + 2 = 36$. If $X_{34} = 1$, we obtain the following matrix with $LB = 34$ unchanged:

$$\text{Remaining Matrix} = \begin{array}{c} \\ 2 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} 2 & 5 \end{array} \\ \begin{pmatrix} - & 0 \\ 0 & - \end{pmatrix} \end{array}$$

Here we have *Zero*$'s$ in each row and column, and the bound will remain the same $LB = 34 + 0 = 34$. We now obtain a feasible solution by assigning $X_{25} = 1$ and $X_{42} = 1$. The solution is $X_{13} + X_{34} + X_{42} + X_{25} + X_{51} = 34$, which is corresponding to the path 1-3-4-2-5-1. The following figure shows the decision tree of this example:

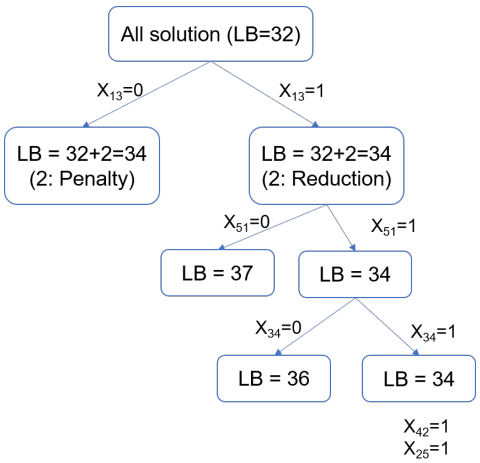

Figure 1: Decision Tree for an Example of BnB Algorithm

The above proposed solution for the example is following the right branch in the figure 1 and it gives us an upper bound ($UB = 34$) for feasible solutions. Since all the left branches in each step have a larger than the proposed solution ($LB \geq UB$), the proposed solution is optimal. In other cases, if there exits a certain branch with $LB < UB$, then the algorithm should go back to this level and explore this branch to see if it gives a better solution. In the end, this BnB algorithm will provide the optimal solution given long enough time.

4.1.3 *Complexity Analysis of BnB.*

The worst case time complexity of Branch-and-Bound algorithm remains the same as that of the Brute Force, which is $O(2^n)$, because in worst case, we may never get a chance to prune a node. Nevertheless, in practice, it may perform very well depending on the different instance of the TSP.

The worst case space complexity is $O(2^n \cdot n^2)$ since we need a $n \times n$ matrix in the beginning, two $(n-1) \times (n-1)$ matrices in the second level, ..., and 2^{n-2} (2×2) matrices in the lowest level, because in the worst case, we may never get a chance to prune a node and need to consider all the branches expanded in each step. Nevertheless, in practice, it may perform very well when most of the branches can be pruned when the first solution is obtained, which only requires $O(n^3)$ in terms of space complexity.

4.1.4 *Pseudo-code of BnB.*

The pseudo-code for BnB algorithm is shown in **Algorithm 1&2**.

4.2 Construction Heuristics with Approximation Guarantees

4.2.1 *Introduction of Approximation Algorithm.*

There are several approaches for dealing with NP-complete problems (TSP here). We have worked on exact solution via BnB approach as shown in Section **4.1**, although it may take super long time in some cases. Sometimes, we need a quick, yet good enough solution, this is where the approximation algorithm fits in. Approximation algorithms run in polynomial time and always produce a solution close to the optimal. We called an algorithm an α-approximation algorithm if it runs in polynomial time, and always outputs a solution that is at most α·OPT for a minimization problem (or at least $1/\alpha$·OPT for a maximization problem), where OPT denote the optimal value. Note that we must have $\alpha \geq 1$ and it does not have to be a constant. Here we will discuss approximation algorithms for the Traveling Salesman Problem.

In the Traveling Salesman Problem, we are given a complete graph G with nonnegative edge costs, and the goal is to find a minimum-cost cycle that visits every vertex exactly once. The key to designing approximation algorithm is to obtain a bound on the optimal value OPT. In the case of TSP, the minimum spanning tree (MST) gives a lower bound on OPT. This is the algorithm that we have discussed in the class, and we will solve the TSP with this MST-Approx algorithm.

4.2.2 *Outline of MST-Approximation Algorithm.*

MST-approximation gives us a simple 2-approximation algorithm for metric TSP. The algorithm is as follows:

(1) Take the minimum-weight spanning tree (MST) of the TSP graph.The MST can be computed in polynomial time using Kruskal's or Prim's algorithm[8].
(2) Do a depth-first search (DFS) of the MST, hitting every edge exactly twice. This "pseudo-tour" PT has the cost of 2× MST ≤ 2× OPT. Write down each vertex as it is visited.
(3) Rewrite the list of vertices, writing each vertex only the first time it appears in PT. Thus PT will be converted to the tour as required by TSP.

4.2.3 *Complexity Analysis of MST-Approximation Algorithm.*

The time complexity of MST algorithm is $O(n^2)$. The DFS step takes $O(n+m)$. Therefore, the total time complexity of MST-Approximation is $O(n^2)$. It takes $O(n)$ to store the sequence of nodes visited and $O(n^2)$ to store the distance matrix between each two cities in the beginning. We can see that this approximation algorithm is very efficient with a guarantee of solution quality within 2×OPT. It is very useful if we need a rough estimation of the solution quickly, but it may not give us very good quality compared to other optimization algorithms given longer computing time.

Algorithm 1: BnB algorithm

```
Data: n nodes with x − y coordinates (x_i, j_i)
Result: a path that visits all the n nodes, total length
Global variable: cost[N][N]                          ▷ cost matrix
function CheckBounds(str,end,cost[n][n])
pencost[0] = t
for i ← 0, n − 1 do
    for j ← 0, n − 1 do
        reduced[i][j] = cost[i][j]
for j ← 0, n − 1 do
    reduced[str][j]] = ∞
for i ← 0, n − 1 do
    reduced[i][end]] = ∞
reduced[end][str] = ∞
RowReduct(reduced)
ColReduct(reduced)
 pencost[end] = pencost[str] + row + col + cost[str][end]
 return pencost[end]
function RowMin(cost[n][n], i)
min = cost[i][0]
for j ← 0, n − 1 do
    if cost[i][j] < min then
        min = cost[i][j]
return min
function ColMin(cost[n][n], i)
min = cost[0][j]
for i ← 0, n − 1 do
    if cost[i][j] < min then
        min = cost[i][j]
return min
function RowReduct(cost[n][n])
row = 0
for i ← 0, n − 1 do
    rmin = rowmin(cost, i)
    if cmin ≠ ∞ then
        row = row + rmin
    for j ← 0, n − 1 do
        if cost[i][j] ≠ ∞ then
            cost[i][j] = cost[i][j] − rmin
function ColReduct(cost[n][n])
col = 0
for j ← 0, n − 1 do
    cmin = columnmin(cost, j)
    if cmin ≠ ∞ then
        col = col + cmin
    for i ← 0, n − 1 do
        if cost[i][j] ≠ ∞ then
            cost[i][j] = cost[i][j] − cmin
```

Algorithm 2: BnB algorithm (con't)

```
function Main()
for i ← 0, n − 1 do
    select[i] = 0
rowreduct(cost)
colreduct(cost)
t = row + col
while allvisited(select) ≠ 1 do
    for i ← 1, n − 1 do
        if select[i] = 0 then
            edgecost[i] = checkbounds(k, i, cost)
    min = ∞
    for i ← 1, n − 1 do
        if select[i] = 0 then
            if edgecost[i] < min then
                min = edgecost[i]
                k = i
    select[k] = 1
    for p ← 1, n − 1 do
        cost[j][p] = ∞
    for p ← 1, n − 1 do
        cost[p][k] = ∞
    cost[k][j] = ∞
    rowreduct(cost)
    colreduct(cost)
```

4.3 Local Search 1: Genetic Algorithm

4.3.1 Basic introduction.

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more chances they have to reproduce[9]. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

4.3.2 Outline of Genetic Algorithm.

1. (Start) Generate random population of n chromosomes (suitable solutions for the problem)
2. (Fitness) Evaluate the fitness f(x) of each chromosome x in the population
3. (New population) Create a new population by repeating following steps until the new population is complete

3.1 (Selection) Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

3.2 (Crossover) With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

3.3 (Mutation) With a mutation probability mutate new offspring at each position in chromosome

3.4 (Accepting) Place new offspring in a new population

4. (Replace) Use new generated population for a further run of algorithm
5. (Test) If the end condition is satisfied, stop, and return the best solution in current population[10]
6. (Loop) Go to step 2

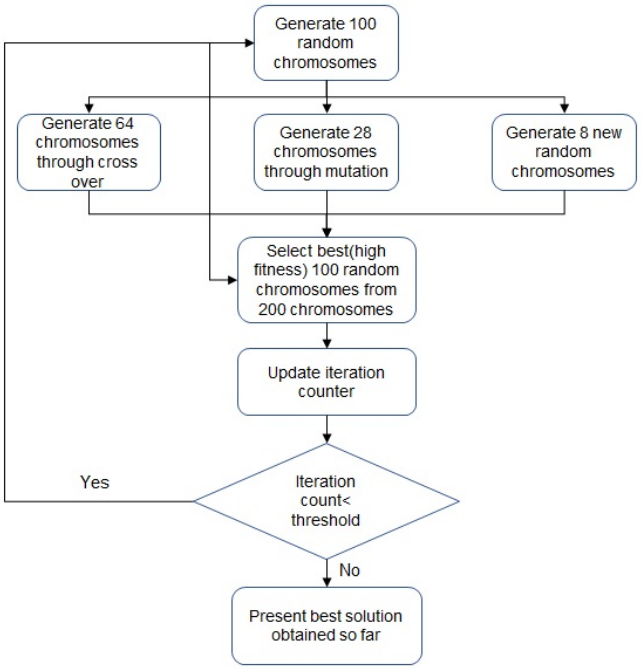

Figure 2: Flow chart of Genetic Algorithm

4.3.3 Example.

A specific example of Genetic Algorithm is as follows[11].

1. Generate chromosomes

We assume that there are 7 cities, we can generate an array of size 6 like below.

Random number	0.23	0.65	0.49	0.58	0.75	0.34
Sequence	2	3	4	5	6	7

Then we sort the array based on the first row. After this, the sequence in second row give us a chromosome, that is, 1− > 2− > 7− > 4− > 5− > 3− > 6− > 1.

We can further repeat this process to generate the required number of chromosomes.

2. Cross over

We use two parent chromosomes to form a new offspring.

Parent1	1	3	4	2	*5*	*7*	*6*
Parent2	1	*7*	*5*	2	3	4	*6*
Offspring	1	2	3	4	*5*	*7*	*6*

As the table shows, we use $5, 7, 6$ of Parent1 for cross over. We further remove $5, 7, 6$ from Parent2.Finally, we get $1, 2, 3, 4$ from

Parent2 and 5, 7, 6 from Parent1 to form a new offspring.

3. Mutation
We use one parent to generate offspring through mutaion.

Parent1	1	3	*4*	*2*	*5*	7	6
Offspring	1	3	*5*	*2*	*4*	7	6

As the table shows, we can mutate the Parent1 by changing the position of 4, 5 to form a new offspring.

4.Flow chart
The Figure 2 shows us the process of Genetic Algorithm based on above comments. Here we assume that population size is 100.

4.3.4 Complexity Analysis of GA.
We assume that g: number of generations, n: population size and m: size of the individuals, Genetic Algorithms complexity is $O(g(nm + nm + n))$. So, the time complexity is $O(gnm)$.
Since the population size is n and individuals size is m, we have n arrays and each has size m. So, the space complexity is $O(nm)$.

4.4 Local Search 2: Simulated Annealing

4.4.1 Introduction.
Simulated annealing algorithm (SA) is a stochastic local search technique inspired by the physical process of annealing.Annealing is a controlled heating and cooling process of metals to improve its strength. It allows worsening steps for the overall optimal solution. Analogous to the metal annealing process, when the temperature is high, atoms are moving around easily. And when it is cooling down, atoms moving slows down and tend to find configuration with lower internal energy. Similarly, higher temperature means more probability to accept worsening steps and lower temperature means shrinking the probability of accepting worse neighboring solution, thus focusing on improving moves.

The probability of accepting the neighboring solution over the current solution could be determined by Metropolis condition shown below:

$$P(s', s) = \begin{cases} 1 & \text{if } f(s') > f(s) \\ exp\left\{\frac{f(s')-f(s)}{T}\right\} & \text{otherwise} \end{cases}$$

As the probability condition shown above, it is obvious that when the neighboring solution is superior than the current solution evaluate by evaluation function, the neighboring solution is accepted. However, if the neighboring solution is worse, whether to choose the neighboring solution over the current solution depends on the probability which is related to the temperature. As the temperature is decreasing, the probability of accepting neighboring solution is decreasing as well. In the SA algorithm, it is also important to choose how to update the temperature and determine stopping criterion. Sometimes, we may also need to restart the algorithm and move back to a previous solution which is better than the current solution.

4.4.2 Outline of Simulated Annealing Algorithm.
The baseline implementation of SA for TSP is showing as follows:

(1) Start with a random initial solution S_0 and set the initial temperature T_0, calculate the total lengthe of the initial sulution $f(s_0)$. Initialize $i = 0$.
(2) $T = T_i$. Use two-exchange neighborhood method to form neighboring solution S' and calculate the total length of the neighboring solution. Take T, S_i and S' to Metropolis criterion and return the solution S. The current iteration $i = i + 1$ and the solution to the current state is $S_i = S$.
   (a) Calculate the total lengths of old solution $f(s_i)$ and new neighboring solution $f(s')$ respectively and their difference as $\Delta f = f(s') - f(s_i)$.
   (b) If $\Delta f > 0$, then return the new neighboring solution S'. Ohterwise, take the new neighboring solution according to probability P.
(3) Reduce the temperature based on some criterion and ensure $T_i < T_{i-1}$. Evaluate the current solution, it stop criterion is satisfied, go to the next step Otherwise, return to step (2).
(4) Return the current solution and terminate the algorithm.

4.4.3 Complexity Analysis of Simulated Annealing Algorithm.
Time and space complexity of implementing simulated annuealing algorithm to solve for TSP is $O(n)$. The outer loop of temperature iteration takes $O(a)$ time steps. At a specific temperature, the inner loop takes $O(b)$ time steps. Thus, the time complexity of the SA algorithm for TSP is $O(n)$.

4.4.4 Pseudo-code of Simulated Annealing.
The pseudo-code for simulated annealing algorithm is shown in **Algorithm 3**.

Algorithm 3: Simulated Annealing algorithm

```
Result: solution, state
function Simulated Annealing(problem, schedule)
current ← Make − Node(Initial − State[Problem])
for t ← 1 to infinity (iters,time cutoff) do
    T ← schedule[t]
    if T = 0 then
        return greedy from current
    next ← Random − successor(current)
    ΔE ← f − Value[next] − f − value[current]
    if ΔE > 0 then
        current ← next
    else
    current ← nextwithprobability
```

5 EMPIRICAL EVALUATION

5.1 Comprehensive Table

All of the code is implemented by desktop with CPU: Intel i7-6700 8 core and RAM: 16G.

5.1.1 Branch-and-Bound Algorithm.
The current results of Branch-and-Bound algorithms are shown in the table shown below. We tried to get as good results as possible with long period of runtime (up to 16 hours). The optimal solution is obtained from the NOES solver as stated in Acknowledgement.

City	Opt. Sol.	Result	Elap.Time	Sol.Qual.
Atlanta	2003763	2003763	0.09	0.00
Berlin	7542	7542	4834	0.00
Boston	893536	893536	6.7	0.00
Champaign	52643	52643	2400	0.00
Cincinnati	277952	277952	0.006	0.00
Denver	100431	123156	1944	3.64
NYC	1555060	1611626	29668	3.64
Philadelphia	1395981	1395981	55489	0.00
Roanoke	655454	832259	600	26.97
SanFrancisco	810196	944471	18093	16.57
Toronto	1176151	1593793	40172	35.51
UKansasState	62962	62962	0.019	0.00
ulysses16	6859	6859	0.01	0.00
Umissouri	132709	145512	600	9.65

5.1.2 Construction Heuristics with Approximation Guarantees.

City	Result	Elap.Time	Sol.Qual.
Atlanta	2270785	0	13.33
Berlin	9550	0.02	26.62
Boston	1028494	0.02	15.10
Champaign	62395	0.02	18.52
Cincinnati	297490	0	7.03
Denver	133065	0.02	32.49
NYC	2003747	0.02	28.85
Philadelphia	1626820	0.02	16.54
Roanoke	808235	0.09	23.31
SanFrancisco	1060717	0.03	30.92
Toronto	1618300	0.05	37.59
UKansasState	65561	0.02	4.13
ulysses16	7788	0.02	13.54
Umissouri	165116	0.02	24.42

5.1.3 Local Search 1: Genetic Algorithm.
The current results of Genetic Algorithm and Simulated Annealing are shown in the table below. The maximum run time is cut off to 2 minutes. The results shown in the table below are the best results achieved in 2 minutes with average value after running 10 times.

City	Result	Elap.Time	Sol.Qual.
Atlanta	2003763	8.23	0.00
Berlin	8020	35.12	6.34
Boston	912650	61.27	2.14
Champaign	54143	52.7	2.85
Cincinnati	277952	0.07	0.00
Denver	109210	108.43	8.74
NYC	1623724	65.06	4.42
Philadelphia	1398906	13.8	0.21
Roanoke	720217	117.4	9.88
SanFrancisco	876175	95.23	8.14
Toronto	1306955	114.72	11.12
UKansasState	62962	0.04	0.00
ulysses16	6859	2.06	0.00
Umissouri	141931	104.18	6.95

5.1.4 Local Search 2: Simulated Annealing.

City	Result	Elap.Time	Sol.Qual.
Atlanta	2102361	4.13	4.92
Berlin	8340	34.61	10.58
Boston	941078	18.52	5.32
Champaign	56416	33.68	7.17
Cincinnati	277952	0.17	0.00
Denver	108190	77.16	7.73
NYC	1690590	54.71	8.72
Philadelphia	1459913	12.34	4.58
Roanoke	753432	117.87	14.95
SanFrancisco	906497	110.28	11.89
Toronto	1313056	118.6	11.64
UKansasState	62980	0.13	0.03
ulysses16	6997	2.16	2.01
Umissouri	144095	112.44	8.58

5.2 Evaluation Plots for LS Algorithms

In this section, we will use multiple plots to evaluate the performance of two Local Search (LS) algorithms, namely, Generic Algorithm (GA) and Simulated Annealing (SA) algorithm. We selected two cities, Berlin and Champaign, as the samples to analyze. They are selected because, 1. we are confident about the exact optimal solutions for these two cities, therefore we can obtain accurate errors/quality ranges for the evaluation, and 2. the input sizes of these two cities are large enough and will take a while (>100s) to obtain solutions with good qualities (within 15%· quality), which will make more sense in terms of performance evaluations and comparisons.

For each algorithm and each city, 100 independent experiments with random seeds are performed. The evaluation plots are shown as follows.

5.2.1 Qualified Runtime for Various Solution Qualities (QRTDs).
The qualified runtime distributions for various solution qualities for two cities using Generic Algorithm (GA) are shown in Figure 3. The plot using Simulated Algorithm (SA) is shown in Figure 4.

From QRTD plots, we can see that for higher solution tolerance (lower quality bar such as 30%, 35%), the curve shifts to the left, meaning we can obtain good enough solutions (high probability) within shorter time. This is reasonable and therefore the algorithms are feasible. We can also see that SA can achieve similar qualities within shorter times, since it can quickly converge to a solution with the controlled cooling process.

5.2.2 Solution Quality Distributions (SQDs).
The SQD plots for two cities using Generic Algorithm (GA) are shown in Figure 5. The plot using Simulated Algorithm (SA) is shown in Figure 6.

From SQD plots, we can see that for longer cutoff times, the curve shifts to the left, meaning we can obtain better enough solutions more easily. This is reasonable and therefore the algorithms are feasible.

5.2.3 Box plots.
The box plots for running times are shown in Figure 7. Note that

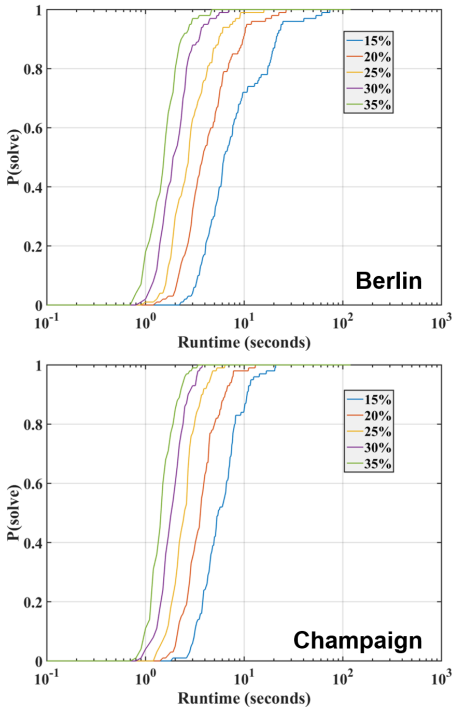

Figure 3: QRTD plot using GA for Berlin and Champaign

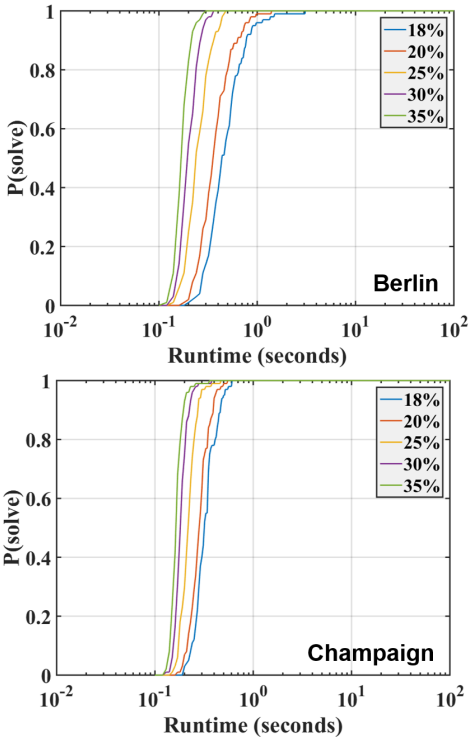

Figure 4: QRTD plot using SA for Berlin and Champaign

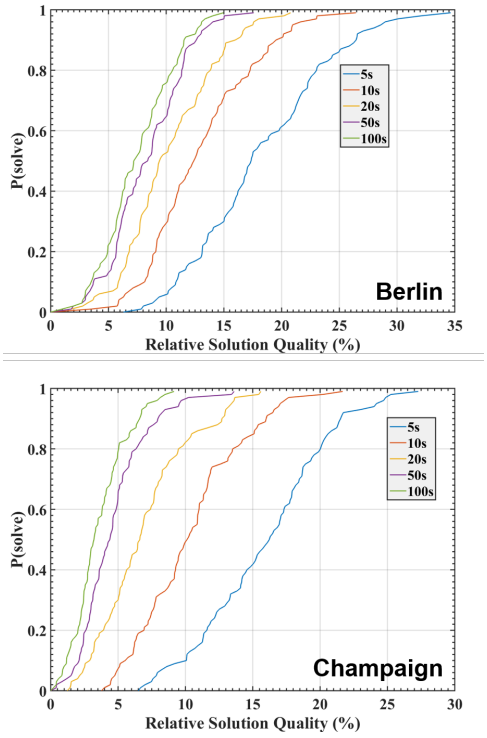

Figure 5: SQD plot using GA for Berlin and Champaign

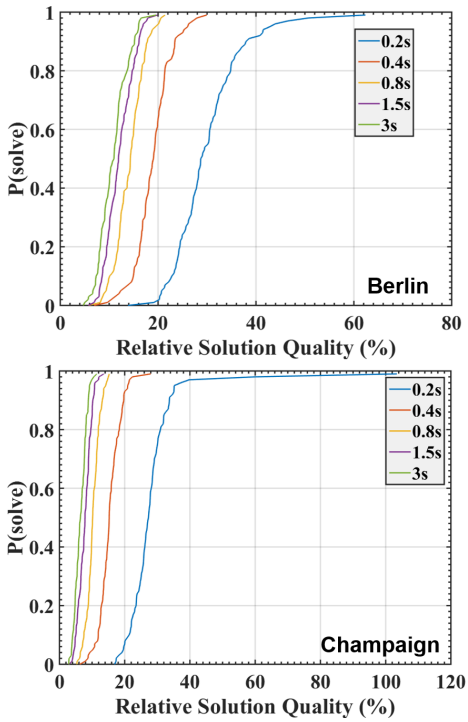

Figure 6: SQD plot using SA for Berlin and Champaign

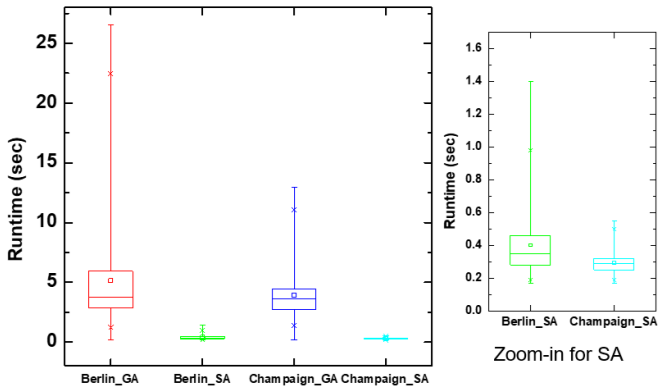

Figure 7: Box plot using GA and SA for Berlin and Champaign (right plot: zoom-in for SA)

this figure shows the minimum runtimes needed for solutions better than $(1 + 20\%)OPT$. For each case, the box plot includes maximum, minimum results, median and quartiles (25% and 75%). 99% and 1% values are also represented by the crosses, and mean value is represented by the square.

From the box plots, there are only a few large outlier data points (<3) based on the results from 100 independent numerical experiments. Therefore, the performance of these two algorithms are stable and robust. The box plots also show that the running time is varying due to random seeds. We can see that SA can quickly converge to a satisfying solution and take less time than GA.

6 DISCUSSION

6.1 Branch and Bound algorithm

A branch-and-bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded by using upper and lower estimated bounds of the quantity being optimized. For the implementation of Branch-and-Bound algorithm, weâĂŹve considered various strategies, including random selection, assignment-problem strategy, and penalty-based selection. Since penalty-based selection will select the smallest extra costs for the current reduced matrix, thus will obtain a possibly lowest lower bound for the current step, we choose this strategy to search through the branches. In principle, BnB algorithm can obtain the exact optimal solution given enough time. However, since the worst case time complexity remains the same as that of the Brute Force, which is $O(2^n)$, it may take extremely long time. As we observed in the results, the cities with large input size (>55) cannot be solved within two days, and we can only obtain solutions as good as (1 + 30%)OPT within 48h runtime.

6.2 Approximation algorithm

MST-Approximation algorithm takes $O(n^2)$ and is the fastest one among the four algorithms. However, although it can guarantee an approximation ratio of 2, its errors are often larger than those of optimized local search algorithms as tested.

6.3 Genetic Algorithm

Genetic algorithm has more capability to find the overall optimal solution. We found that the number of chromosomes in the population, crossover probability and mutation probability play important roles to improve the accuracy and efficiency of the algorithm. Based on many tries, we set the number of chromosomes in the population as 20 if the number of nodes in the city is more than 90. And the probability of mutation is set as 0.1. Otherwise, the number of chromosomes is set as 40 and the probability of mutation is set as 0.01. The probability of crossover is set as 0.8 regardless of the number of nodes in the city.

6.4 Simulated Annealing algorithm

Simulated annealing algorithm is easy to implement. However, based on the implementation, SA algorithm gets stuck to local minimum easily. In addition, the quality of the output strongly depends on the initial value. In our implementation, we normalized the probability of taking new solution over the old one in the Metropolis condition as following:

$$P(s', s) = \begin{cases} 1 & \text{if } f(s') < f(s) \\ exp\left\{\frac{f(s)-f(s')}{Tf(s)}\right\} & \text{otherwise} \end{cases}$$

where $f(s)$ and $f(s')$ is the total length of new solution and old solution respectively.
The temperature is updated using exponential relation shown below:

$$T = Te^{-kt_c}$$

Where t_c is the cutoff time. We would like the temperature to drop to 20% of the initial temperature. Thus, the constant k can be derived as:

$$k = \frac{3.912}{t_c}$$

With different cutoff time, the speed of temperature decrease is also different, which would probability improve the output quality.

7 CONCLUSIONS

In this project, we have implemented four different algorithms to solve the NP-complete problem âĂŞ Traveling Salesperson Problem (TSP). For the exact solutions, we used penalty-based selection strategy in the Branch-and-Bound algorithm to obtain good quality of lower bound. Over half of all the given cases have been solved within 1.5 hours. Then we implemented MST-approximation, which is a classical algorithm to solve NP-complete problem within short time frame. For all the given cases, it takes less than 0.1 sec to run the algorithm, and can achieve solution qualities better than 40%.

We have also implemented two local search algorithms: generic algorithm (GA) and simulated annealing algorithm (SA). Local search algorithms are generally fast and can produce satisfactory results with carefully optimized parameters. All the given cases can be

solved with qualities better than 15% within 2 minutes in our experiments.

ACKNOWLEDGMENTS

D. Han and S. Xiong conducted the algorithm design, implementation and analysis for the BnB and Approximation algorithms. C. Ye and Y. Zhou conducted the algorithm design, implementation and analysis for the local search algorithms. All of the authors contributed to the manuscript with the sections corresponding to each algorithm.

We would like to thank the NOES solver to offer Concorde to solve symmetric TSP at https://neos-server.org/neos/solvers/co:concorde/-TSP.html.

REFERENCES

[1] John D C Little, Katta G Murty, Dura W Sweeney, and Caroline Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research*, 11(6):972–989, 1963. URL https://econpapers.repec.org/RePEc:inm:oropre:v:11:y:1963:i:6:p:972-989.

[2] Maria Battarra, Artur Alves, Anand Subramanian, and Eduardo Uchoa. Exact algorithms for the traveling salesman problem with draft limits. *European Journal of Operational Research*, 235(1):115–128, 2014. ISSN 0377-2217. doi: 10.1016/j.ejor.2013.10.042. URL http://dx.doi.org/10.1016/j.ejor.2013.10.042.

[3] JÃÿrgen Glomvik, Marielle Christiansen, Kjetil Fagerholt, and Gilbert Laporte. Computers & Operations Research The Traveling Salesman Problem with Draft Limits. *Computers and Operation Research*, 39(9):2161–2167, 2012. ISSN 0305-0548. doi: 10.1016/j.cor.2011.10.025. URL http://dx.doi.org/10.1016/j.cor.2011.10.025.

[4] Sumanta Basu and Diptesh Ghosh. A Review of the Tabu Search Literature on Traveling Salesman Problems. pages 1–16, 2008.

[5] Zar Chi, Su Su, and May Aye Khine. An Ant Colony Optimization Algorithm for Solving Traveling Salesman Problem. 16:54–59, 2011.

[6] Varshika Dwivedi. Travelling Salesman Problem using Genetic Algorithm. pages 25–30, 2012.

[7] Chaitanya Pothineni. Travelling Salesman Problem using Branch and Bound Approach. pages 1–8, 2013.

[8] Jon Kleinberg Tardos and ÃĽva. *Algorithm Design - Jon Kleinberg and Éva Tardos*. ISBN 0321295358. doi: 10.1145/214748.214752.

[9] Cezary Z. Janikow and Zbigniew Michalewicz. Binary and floating point representation in GA.pdf, 1991.

[10] Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets. URL http://www.obitko.com/tutorials/genetic-algorithms/.

[11] Tutorial : Introduction to Genetic Algorithm n application on Traveling Sales Man Problem (TSP) - YouTube. URL https://www.youtube.com/watch?v=3GAfjE_ChRI.