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 or TBD), local search with simulated annealing (SA) and generic algorithm (GA). We have also evaluated their theoretical and experimental complexities on real world instances. Based on the empirical analysis, we found that (under construction).

For the checkpoint of this project, we implemented two approaches: branch and bound (BnB), and Local search (LS) with generic algorithm. The results are shown in the follow sections. **Also, we'd like to participate the project competition (Team 31)**.

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. (Full versions of results under construction)

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

$$\begin{aligned} \pi &= 3.141592; \\ deg &= \lfloor x_i \rfloor; \\ min &= \pi(deg + 5/3min)/180 \end{aligned}$$

- Step 2. Calculating the geographical distance

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

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 to be solved 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 solution 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:

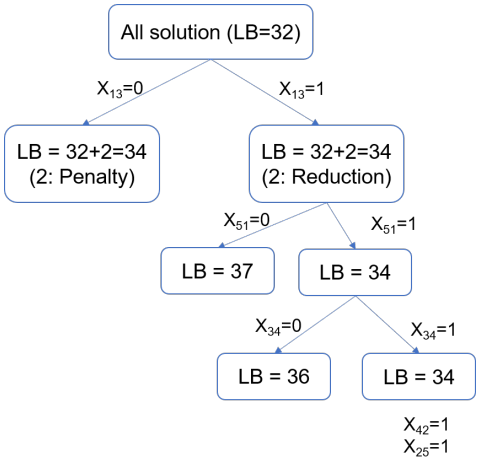

Figure 1: Decision Tree for an Example of BnB Algorithm

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**.

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)
```

4.2 Construction Heuristics with Approximation Guarantees

under construction

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[8].

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[9]

6. (Loop) Go to step 2

4.3.3 *Example.*

A specific example of Genetic Algorithm is as follows[10].

1. Generate chromosomes

We assume that there are 7 citys, 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 $O(nm)$.

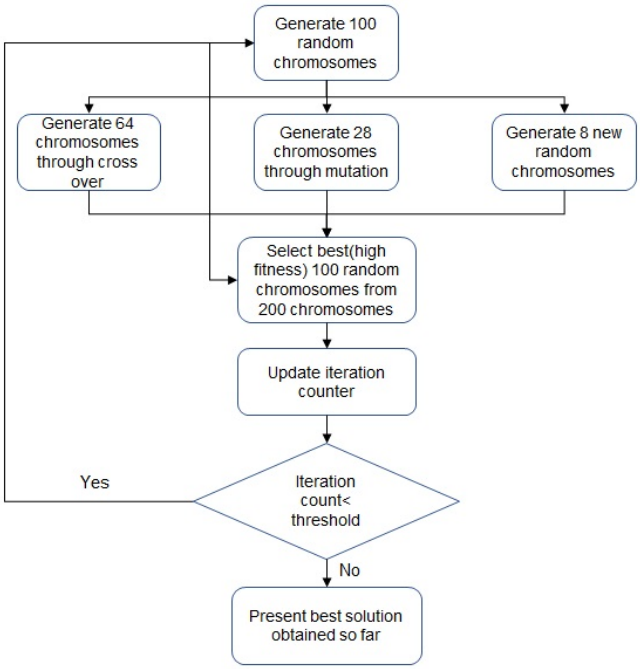

Figure 2: Flow chart of Genetic Algorithm

4.4 Local Search 2: Simulated Annealing

under construction

5 EMPIRICAL EVALUATION

5.1 Branch-and-Bound Algorithm

The current resutls of Branch-and-Bound algorithms are shown in the table shown below. The code is implemented by desktop with CPU: Intel i7-6700 8 core and RAM: 16G. The maximum implementation time is cut off to 10 minutes. Thus the results shown in the table below are the best results achieved in 10 minutes by Branch-and-Bound algorithm.

City	Elap.Time (s)	Sol.Qual.
Boston	6.70	893536
Atlanta	0.09	2003763
Champaign	600	53047
Cincinnati	0.01	277952
UKansasState	0.02	62962
ulysses16	0.01	6756
Berlin	600	7832
Philadelphia	600	1481183
Denver	600	123828
NYC	600	1632222
Roanoke	600	832259
Umissouri	600	145512
Toronto	600	1595032
SanFrancisco	600	946164

5.2 Local Search 1: Genetic Algorithm

The current results of Genetic Algorithm are shown in the table below. The code is implemented by desktop with CPU: Intel i7-6700, 8 core and RAM: 16G. The maximum implementation time is cut off to 10 minutes. Thus the results shown in the table below are the best results achieved in 10 minutes by Genetic Algorithm.

City	Elap.Time(s)	Sol.Qual.
Boston	600s	936444
Atlanta	2.2s	2003763
Champaign	600s	54061
Cincinnati	0.04s	277952
UKansasState	0.04s	62962
ulysses16	0.78s	6756
Berlin	600s	8195
Philadelphia	600s	1401831
Denver	600s	111036
NYC	600s	1615971
Roanoke	600s	1062436
Umissouri	600s	143276
Toronto	600s	1415453
SanFrancisco	600s	912124

5.3 Local Search 2: Simulated Annealing

under construction

5.4 Overall Evaluation

6 DISCUSSION

under construction

7 CONCLUSIONS

under construction

ACKNOWLEDGMENTS

The authors would like to thank xxx for providing the xxx.

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] JÃ¿rgen 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] Cezary Z. Janikow and Zbigniew Michalewicz. Binary and floating point representation in GA.pdf, 1991.

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

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