CSE 6140 Algorithms Project - TSP
Fall 2018, Group 20

HENGHAO LI, Georgia Institute of Technology, USA
WENTING TAN, Georgia Institute of Technology, USA
XIA WU, Georgia Institute of Technology, USA
YUNHAN ZOU, Georgia Institute of Technology, USA

The traveling salesman problem is a classic NP-hard problem in computer science and operations search field over roughly 70 years[9], focusing on providing optimal solutions[13]. We proposed two methods with both exact solution and heuristics with approximation guarantees in this project: Branch and Bound and MST approximation. We were provided with 14 dataset of different location points of 14 different cities in the world. One of them is of GEO format while the remaining are of Euclidean one. All the computed distances are stored in a symmetric matrix as an input for all algorithms. For Branch-and-Bound method, we obtained results without errors comparing to benchmarks[1], which proves the correctness of our implementation. For MST approximation, we got relative small or no errors comparing to the optimal solution[1]. In the future, two other local search methods will be implemented and more comparisons among all these four methods will be presented.

Key Words and Phrases: Traveling salesman problem, Minimum Spanning Tree, Branch and Bound, NP-complete

1 INTRODUCTION

The traveling salesman problem (TSP) has been served as one of the most intensively studied problem[9], which was first formulated in 1930s[14]. It is an NP-hard computational problem and used as a benchmark for several optimization problems over the years[14]. There already exist a large scale of exact solutions and approximating methods. Besides, it can be widely applied in many different fields other than computer science, such as DNA sequencing and astronomy.

In this project, we plan to solve this problem with 4 different algorithms. Branch-and-bound is a classic method and can help us find the exact solution, by sacrificing the running time. MST heuristic is an approximation method that can reduce the time complexity to "nearly" linear, but may introduce offsets to the result. In addition to these two algorithms mentioned, we will add another two local search methods in the future and perform a comprehensive analysis among them.

2 PROBLEM DEFINITION

2.1 Travelling salesman problem

The traveling salesman problem, which is a classic NP-hard problem[12] in combinatorial optimization, aims to solve the following problem: given a list of cities and the distances between these cities, find the shortest possible route that visits all cities exactly once before returning to the original city [14]. This problem can be formulated as a weighted graph problem: given a list of N nodes and weighted edges $d(u, v)$ for $u \in N$ and $v \in N$, find the shortest path that visits all the nodes only once in the graph before returning to the starting node. Note that the distances between any two of the cities, i.e. the weight of any edge, applies to both direction. That is, in this

Authors' addresses: Henghao Li, Georgia Institute of Technology, North Ave NW, Atlanta, GA, 30332, USA, hli366@gatech.edu; Wenting Tan, Georgia Institute of Technology, North Ave NW, Atlanta, GA, 30332, USA, wtan64@gatech.edu; Xia Wu, Georgia Institute of Technology, North Ave NW, Atlanta, GA, 30332, USA, xia.wu@gatech.edu; Yunhan Zou, Georgia Institute of Technology, North Ave NW, Atlanta, GA, 30332, USA, yzou46@gatech.edu.

project, the graph is undirected and the adjacency matrix is symmetric. In addition, there exists an edge between each of and the rest of the cities, making it a dense graph.[11].

More specifically, we will parse input data and save the distances between any two given locations in an n-by-n symmetric matrix $D = (d(u, v))$, where u, v are locations and $d(u, v)$ is the distance or the cost function between them. Our goal is to arrange these locations in a cyclic order such that the sum of $d(u, v)$ between any two consecutive points are minimal[3]. Note that all the edge costs here are symmetric and satisfy the triangle inequality.

2.2 Data

The input data are all stored as *.tsp* file, whose format is as used in *TSPLIB*[8]. In this project, we have information of 14 different cities in the world. Each of them contains some specific locations of $x - y$ coordinates and corresponding IDs, with data type of either Euclidean or geographical. Specifically, the $x - y$ coordinates in ulysses16.tsp file are formatted as geographical format while all the others are stored as Euclidean one. Following are the details of preprocessing input dataset. After calculation, we will store the distances in a 2-dimensional matrix, where the indices represent for the vertex ID and the values represent for the corresponding distance between them.

2.2.1 Euclidean 2D Format. Let $u = (x_u, y_u)$ and $v = (x_v, y_v)$ denote any two locations, and let $d(u, v)$ denote the distance between them, then the Euclidean distance is calculated as following:

$$\begin{aligned} distance_x &= x_u - x_v \\ distance_y &= y_u - y_v \\ distance(u, v) &= \lfloor \sqrt{{distance_x}^2 + {distance_y}^2} \rceil \end{aligned}$$

2.2.2 Geographical Format. Let $u = (x_u, y_u)$ and $v = (x_v, y_v)$ denote any two locations, where x and y represent for latitude and longitude, respectively. Then $d(u, v)$ is their distance on the idealized sphere with radius 6378.388 kilometers. The latitude and longitude are given in `DDD.MM` format, where `DDD` represents for degree and `MM` represents for minutes. Positive values of x and y represent for North and East, while negative values of x and y are assumed to be South and West, respectively. For example, $u = (43.22, -22.34)$ means for point u, its latitude and longitude are North $43°22'$ and West $22°34'$. Thus we need to convert these data points into radians before calculating $d(u, v)$:

$$\begin{aligned} rad(x) &= \pi * \frac{\lfloor x \rfloor + 5.0 * \frac{x - \lfloor x \rfloor}{3.0}}{180.0} \\ rad(y) &= \pi * \frac{\lfloor y \rfloor + 5.0 * \frac{y - \lfloor y \rfloor}{3.0}}{180.0} \end{aligned}$$

Then the Geographical distance is calculated as following:

$$\begin{aligned} RRR &= 6378.388 \\ q1 &= \cos\left(rad(y_u) - rad(y_v)\right) \\ q2 &= \cos\left(rad(x_u) - rad(x_v)\right) \\ q3 &= \cos\left(rad(x_u) + rad(x_v)\right) \\ distance(u, v) &= \lfloor (RRR * \arccos\left(0.5 * ((1.0 + q1) * q2 - (1.0 - q1) * q3)\right) + 1.0) \rfloor \end{aligned}$$

Note that, based on the methods above, the diagonal of the input matrix are values of zero for Euclidean distance, whereas the diagonal of the input matrix are values of one for geographical distance.

2.3 Output

Output data are all stored in folder named "output". There are two typical types of output.

2.3.1 Solution file. This file ends with ".sol" and contains two lines only.

- line 1: Best quality found for the specific input dataset, with regard to the chosen algorithm and cut-off seconds.
- line 2: List of vertex IDs of the TSP tour.

2.3.2 Trace file. This file ends with ".trace" and each line contains two variables.

- variable 1: Timestamp in seconds
- variable 2: Best quality found of the corresponding timestamp

3 RELATED WORKS

Traveling salesman problem is a classic NP-hard problem in computer science field. Many works and solutions have been made and raised in the past years. Kruskal et al. provided us a greedy method to find the minimum spanning tree in a graph, which can be applied to solve the traveling salesman problem[6]. S. Lin et al. proposed an effective heuristic algorithm based on general approach to heuristics, and solve the problem by joining the two end points with a small distance[7]. V Černý presented approximate solutions using Monte Carlo algorithm, obtaining results that are very close to the optimal ones. It generates the permutations of trip stations randomly, where the probabilities count on the length of the route[2]. George Dantzig et al. found a solution of large-scale TSP, showing that by a certain arrangement of 49 cities in the US, one can find the shortest road distance for one of each of the 48 states and Washington D.C.[3]. David E Goldberg et al. show a genetic algorithm that lead to a partially-mapped crossover operator. They formulated TSP as a n-permutation problem without allele values and obtained a "nearly" optimal solution[5]. Although in this project we are solving TSP with sequential methods only, there exist some distributed algorithms, for example, the ant colony system introduced by Marco Dorigo et al., which outperforms other natural algorithms significantly for both symmetric and asymmetric TSPs[4].

4 ALGORITHMS

4.1 Branch-and-Bound

4.1.1 Algorithm description.

In TSP problem, our objective is to minimize the length of a complete tour that visits every node exactly once except the starting node. For branch-and-bound, the idea is to split up the set of all solutions into two or more disjoint subsets. Each of these subsets can be further split into subsets and so on. However, the splitting must obey a constraint that the subsets have to satisfy. This process is called the branching. One can visualized this process as a tree where the root node represents the original problem with an empty solution. The internal node are partial solutions while leaf nodes are solutions. Each child node inherits the constraints of their parent and expands on the existing solution. For each node, we can calculate a lower bound that is based on the distance traveled so far. When the lower bound of a node is higher than the value of some known upper bound, then we do not need to go down that node and prune that part of the tree. This process is called the bounding. When a new solution is found at a leaf node, we compare the cost (total distance traveled) to our current upper bound and then keep the best one. In this way, we keep finding better solutions and saving time exploring promising nodes. The process ends when there is no more node to explore, at which point we can be sure that the current best solution is the optimal solution.

4.1.2 *Algorithm implementation.*

There are 3 classes related to branch-and-bound algorithm: Node.py, PriorityQueue.py, and BranchNBound.py. The Node class represents a single node in the search space tree. The key information each node contains is the path taken to that point, and the lower bound of reaching a solution. There are several ways to expand a parent node, such as depth-first order and lower-cost order. In this project, we designed a priority queue that always explores the most promising node with the lowest cost. This is called the least-cost branch-and-bound. The BranchNBound contains the body of the algorithm. The pseudocode is outlined below in Algorithm 1 and Algorithm 2.

As Algorithm 1 states, the upper bound of the best solution is initially set to infinity, and the empty solution always start from index 0. The distance matrix is preprocessed and reduced to obtain the initial cost with the steps outlined in Algorithm 2. The initial cost at this stage is not an exact cost, but it gives some idea on the cost lower bound, i.e., the actual cost cannot be lower than this cost. Then the reduced matrix, initial cost, path, and a set tracking visited indices are passed to the child node, with the key in the Priority Queue being the cost. The stopping condition for the while loop is either time runs out or all nodes have been removed from the queue. The node with the smallest is selected to expand next. If its cost is larger than the current upper bound, then we know that it's not going to lead to a promising path, and thus we prune it. Otherwise, we expand its neighbors that haven't been visited in the path so far, reduce the corresponding reduced matrix and find the new cost, which is calculated using the equation:

$$\text{New Cost} = \text{Cost So Far} + \text{Reduced Cost} + \text{Reduced Matrix[start, end]}$$

Again, we check whether the new cost is lower than the upper bound. If so, we add the new child node to the queue.

4.1.3 *Analysis.*

Compared to the benchmark solution[1], the solution found by branch-and-bound algorithm is the optimal solution with the shortest distance. This is expected because branch-and-bound is an exact algorithm, so given enough time, it's guaranteed to find the optimal solution. However, since it's an exact algorithm, it tends to evaluate every possible solution. Even with pruning, the runtime still follows the exponential growth. Because of the NP-Completeness of TSP, a solution with polynomial runtime is yet to be found.

4.1.4 *Time and space complexities.*

In this exhaustive search mentioned above, all solutions will be tested. And The space complexity using Priority Queue is upper bounded as following:

- Time complexity is $O(n!)$.
- Space Complexity is $O(n)$.

4.2 MST-Approximation

4.2.1 *Algorithm description.*

The main function of the algorithm is named as *compute*. It takes in two inputs *input* and d, which are the cost matrix and its dimension. Its output is *tsp* and c, which are the the 2-approximate tour and its total cost. First step is to compute the minimal spanning tree of the graph. Second step is to perform depth first search on the minimal spanning to obtain a pre-order of all nodes. The details of these steps are shown in the *MST* and *DFS* functions in Algorithm 3, respectively.

Our MST implementation is based on Prim's algorithm. The reason why we choose it instead of Kruskal's one here is that the graph is dense in that there is an edge between each pair of cities thus $m = n^2$. For the same reason, we choose normal list representation rather than binary heap or

ALGORITHM 1: Branch-and-Bound Algorithm

Input: Symmetric distance matrix (CostMatrix), program time limit in seconds
Output: The best solution found within the time limit

```
UpperBound = ∞; Solution = []; TimeLimit = time limit; Duration = 0
preprocess CostMatrix to make diagonal elements ∞
ReducedMatrix, cost = Reduce(CostMatrix)
Path = [0]; Visited = set(0)
RootNode = Node(ReducedMatrix, cost, Path, Visited)
Frontier = PriorityQueue(); Frontier.append(RootNode)
repeat
    TopNode = Frontier.pop()
    CostSoFar = TopNode.getCost()
    if CostSoFar < UpperBound then
        Get Path, CurrentIndex, ReducedMatrix, Visited from TopNode;
        Neighbors = Indices currently not in Visited;
        if ReducedMatrix contains only ∞ then
            if CostSoFar < UpperBond then
                Solution = Path;
                UpperBound = CostSoFar;
            end
        end
        for NextIndex in Neighbors do
            Set ReducedMatrix[CurrentIndex, :] and ReducedMatrix[:, NextIndex] to ∞;
            Set ReducedMatrix[NextIndex, 0] = ∞;
            Cost, NewReducedMatrix = Reduce(ReducedMatrix);
            NewCost = CostSoFar + Cost + ReducedMatrix[CurrentIndex, NextIndex];
            if NewCost < UpperBound then
                Path.append(NextIndex); Visited.append(NextIndex);
                ChildNode = Node(NewReduce, NewCost, Path, Visited);
                Frontier.append(ChildNode);
            end
        end
    end
    Update Duration;
until Frontier.isEmpty() or Duration > TimeLimit;
```

ALGORITHM 2: Reduce Matrix

Input: Cost matrix to reduce
Output: Cost, Reduced cost matrix

```
Cost = 0;
MinRow = minimum values for each row;
if The minimum value in a row is not ∞ or 0 then
    Subtract the minimum value from the row;
    Cost = Cost + Min;
end
MinCol = minimum values for each column;
if The minimum value in a column is not ∞ or 0 then
    Subtract the minimum value from the row;
    Cost = Cost + Min;
end
```

ALGORITHM 3: MST-Approximation

```
Function compute(d, input): (c, tsp)
    root = 0 mst = MST(d, input, root)
    tsp = DFS(d, mst, root)
    c = 0
    for i = 0 to d-1 do
        c += input[tsp[i]][tsp[i + 1]]
    end
end
Function MST(d, input, root) : (mst)
    for i = 0 to d-1 do
        mst[i] = i ; a[i] = Inf ; visited[i] = 0
    end
    visited[root] = 1
    while True do
        b = 1
        for i = 0 to d-1 do
            if not visited[i] then b = 0
        end
        if b == 1 then break
        a_min = Inf ; i_min = 0
        for i = 0 to d-1 do
            if not visited[i] and a[i] < a_min then
                a_min = a[i] ; i_min = i
            end
        end
        u = i_min
        visited[u] = 1
        for v = 0 to d-1 do
            if not visited[v] and input[u][v] < a[v] then
                a[v] = input[u][v]
                mst[v] = u
            end
        end
    end
end
Function DFS(d, mst, root): (tsp)
    tsp = []
    for i = 0 to d-1 do visited[i] = 0
    stack.push(root)
    visited[root] = 1
    while not stack.empty() do
        cur = stack.pop()
        tsp.append(cur)
        for i = 0 to d-1 do
            if not visited[i] and mst[i] == cur then
                stack.push(i)
                visited[i] = 1
            end
        end
    end
    tsp.append(root)
end
```

Fibonacci heap. Thus the time complexity is $O(n^2)$. Also referencing from the lecture slide Graph-2, we know Prim's is a greedy algorithm, in that at each step it chooses the next smallest-weighted edge connected to the partially constructed spanning tree. The correctness lies in the Cut Property, which essentially states that the smallest-weigted edge of any cut of the graph must be in the minimal spanning tree.

Our DFS implementation is stack-based. Since the *tsp* we obtained from MST is a list where the index represents one node and the value represents its parent, and the root of the tree is also the parent of itself. For each node we choose, we have to search through the list to find its children to expand. Thus a potentially better implementation is to construct a tree first, reducing time complexity from $O(n^2)$ to $O(n \log n)$.

4.2.2 *Approximation guarantee.*

Following the lecture slides, we prove that MST-approximation is a 2-approximation. First observation is that the total cost of minimal spanning tree is no greater than the optimal cost of travelling salesman problem, $cost(mst) \leq cost(OPT)$. Second observation is a Eulerian tour can be obtained through traverse through the minimal spanning tree and include each edge twice as they are visited from both directions. Thus we have the relation $cost(T) = 2 * cost(mst)$, where T is a Eulerian tour. Third observation is that traversing the minimal spanning tree in preorder and include each edge exactly once when they are first visited gives us the tour *tsp* in the algorithm. Further, we have the relationship $cost(tsp) \leq cost(T)$, by applying the triangle inequality, under the assumption that this version of the TSP problem is metric. Combing the three results we have $cost(OPT) \leq cost(tsp) \leq 2cost(OPT)$. This concludes our proof.

4.2.3 *Time and space complexities.*

- Time complexity is $O(n^2)$.
- Space Complexity is $O(n^2)$.

4.2.4 *Strength and weakness.*

First strength of this algorithm is that it gives guarantee of the solution quality, that is, the minimal cost is no larger than two times the optimal. Second strength is that the time and space complexity is very low both theoretically and empirically. Essentially, it is polynomial solution to NP-hard problem. First weakness of approximation algorithm is that it cannot be improved in solution quality even given more time and/or space resources. Second weakness of approximation algorithm is that the algorithm itself is hard to improve upon. For example in the MST-approximation, no performance tuning of any parameter is possible.

5 EMPIRICAL EVALUATION

5.1 Platform

(1) CPU: 2.7 GHz Intel Core i5
(2) RAM: 8GB 1867 MHz DDR3
(3) Language: Python 2.7

5.2 Experiments

5.2.1 *Experimental procedure.*

Now we have two algorithms (Branch-and-Bound and MST-approximation) working, and tested on provided dataset.

- Branch-and-Bound: This algorithm was tested on Cincinnati, ulysses16, Atlanta and UKansasState tsp files. The results were compared to the benchmark solution[1] and are listed in Table 1.

- MST-Approx: This algorithm was tested on all the dataset. The results were compared to the optimal solution and are listed in Table 1

5.2.2 *Evaluation criteria and results.*

For comparing purpose, we set the time limit to 10 minutes. Table 1 shows all results of all 14 cities we obtained until now from Branch and Bound and MST approximation algorithms.

Table 1. Comprehensive Table

	Branch and Bound			**MST Approx.**		
Dataset	Time(s)	Sol.Qual.	RelErr	Time(s)	Sol.Qual.	RelErr
Ulysses16	267.225	6859	0	0.000356698036194	1	7652/6859-1
Cincinnati	0.259	277952	0	0.000189781188965	1	315452/277952-1
Atlanta	164.215	2003763	0	0.000413107872009	1	0
UKansasState	0.558	62962	0	0.000260019302368	1	0
Berlin	N/A	N/A	N/A	0.00235280990601	1	10303/7542-1
Boston	N/A	N/A	N/A	0.00152168273926	1	1094649/893576-1
Champaign	N/A	N/A	N/A	0.00223729610443	1	0
Denver	N/A	N/A	N/A	0.00477349758148	1	0
NYC	N/A	N/A	N/A	0.00328822135925	1	0
Philadelphia	N/A	N/A	N/A	0.00111479759216	1	0
Roanoke	N/A	N/A	N/A	0.0321681976318	1	0
San Francisco	N/A	N/A	N/A	0.00741050243378	1	0
Toronto	N/A	N/A	N/A	0.00866539478302	1	0
UMissouri	N/A	N/A	N/A	0.00735290050507	1	0

6 DISCUSSION

Due to time limitation, we have not completed all the tests for Branch-and-Bound algorithm. However, for at least one of either Euclidean or geographical format input dataset, we obtained exactly the same results as the benchmark[1], which is a proof of correctness of our implementation. This can be told from the Table 1 that all the relative errors are zero until now. For MST heuristic method, we finished computation in a very short period of time other than the Branch-and-Bound method, and obtained results that give guarantee of the solution quality. Yet we may further improve both the computational time and space complexity of this algorithm, for instance, reducing time complexity of DFS from $O(n^2)$ to $O(n \log n)$.

7 CONCLUSIONS

We chose Branch-and-Bound and MST-approximation algorithm as our implementing choices for the traveling salesman problem. The former one performs a top-down exhaustive search through the graph[10], and the solution found is guaranteed to be optimal. For Ulysses and Cincinnati, the qualities of the best solution we obtained are 6859 and 277952, respectively, which are exactly the same as the benchmark[1]. However, this method is time-consuming and needs further improvements on lower bounding function. Since the minimum possible cost of TSP is at least cost of a MST of the graph, it is easy for us to implement TSP in polynomial (even near-linearly) time[9] with MST heuristic. In the future, we will make improvement on the existing ones and implement another two local search algorithms.

REFERENCES

[1] 1995. Best known solutions for symmetric TSPs. (10 Feb 1995). http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html

[2] V. Černý. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45, 1 (01 Jan 1985), 41–51. https://doi.org/10.1007/BF00940812

[3] George Dantzig, Ray Fulkerson, and Selmer Johnson. 1954. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America* 2, 4 (1954), 393–410.

[4] Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation* 1, 1 (1997), 53–66.

[5] David E Goldberg, Robert Lingle, et al. 1985. Alleles, loci, and the traveling salesman problem. In *Proceedings of an international conference on genetic algorithms and their applications*, Vol. 154. Lawrence Erlbaum, Hillsdale, NJ, 154–159.

[6] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[7] Shen Lin and Brian W Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations research* 21, 2 (1973), 498–516.

[8] Gerhard Reinelt. 1991. TSPLIB - A traveling salesman problem library. *ORSA journal on computing* 3, 4 (1991), 376–384.

[9] Tim Roughgarden. [n. d.]. Lecture 16: The Traveling Salesman Problem. ([n. d.]). https://theory.stanford.edu/~tim/w16/l/l16.pdf

[10] Wikipedia. [n. d.]. Wikipedia, Branch and bound. ([n. d.]). https://en.wikipedia.org/wiki/Branch_and_bound

[11] Wikipedia. [n. d.]. Wikipedia, Complete graph. ([n. d.]). https://en.wikipedia.org/wiki/Complete_graph

[12] Wikipedia. [n. d.]. Wikipedia, NP-hardness. ([n. d.]). https://en.wikipedia.org/wiki/NP-hardness

[13] Wikipedia. [n. d.]. Wikipedia, Travelling salesman problem. ([n. d.]). https://simple.wikipedia.org/wiki/Travelling_salesman_problem

[14] Wikipedia. [n. d.]. Wikipedia, Travelling salesman problem. ([n. d.]). https://en.wikipedia.org/wiki/Travelling_salesman_problem