

Travelling Salesman Problem

Project Report

Group Members:

Abhinav Agarwal 19UCS254

Raghav Agrawal 19UCS237

Pranjal Sharma 19UCC144

Aditya Pandey 19UCC043

Course Instructors:

Dr Poulami Dalapati

Dr Nitin Kumar

**Department of Computer Science and Engineering
The LNM Institute of Information Technology, Jaipur**

Report Contents

1. Task Definition
2. Infrastructure
3. Approach
 - a. Minimum Spanning Tree
 - i. Prims Algorithm
 - b. Minimum Spanning Tree-Based Heuristic
 - c. A* Search Algorithm
 - d. Sample Output
 - e. Time Complexity
 - f. Current State of Art Methods
4. Conclusion
5. Acknowledgement
6. References

Task Definition

The travelling salesman problem is a well-known optimization problem. Optimal solutions to small instances can be found in a reasonable time by linear programming. However, since the TSP is NP-hard, it will be very time consuming to solve larger instances with guaranteed optimality. Setting optimality aside, there's a bunch of algorithms offering comparably fast running time and still yielding near-optimal solutions.

The travelling salesman problem is to find the shortest Hamiltonian cycle in a graph. This problem is NP-hard and thus interesting. There are a number of algorithms used to find optimal tours, but none are feasible for large instances since they all grow exponentially.

We can get down to polynomial growth if we settle for near-optimal tours. We gain speed, speed and speed at the cost of tour quality. So the interesting properties of heuristics for the TSP is mainly speed and closeness to optimal solutions.

There are mainly two ways of finding the optimal length of a TSP instance. The first is to solve it optimally and thus finding the length. The other is to calculate the Held-Karp lower bound, which produces a lower bound to the optimal solution. This lower bound is the de facto standard when judging the performance of an approximation algorithm for the TSP.

Input:- Undirected Weighted Graph (cities are nodes and paths are edges, the distance between two cities are weights), Euclidean distance between every city and goal state.

Output:- Shortest Path that obeys the criteria described

Some **Real-World problems** solved using TSP are:-

- Routing problems on networks, where total routing costs are distance-dependent. An example is the processing sequence in PCB manufacture or VLSI fabrication.
- Scheduling problems. An example is that of minimizing overall makespan for sequence-dependent setup times on equipment such as paint mixing machines. Some analytical problems
- Many problems are very similar to the TSP, such as the Vehicle Routing Problem, which has obvious practical implications in distribution problems (milk runs, courier, post, etc.) and the Hamiltonian circuit problem in graph theory.


Infrastructure

We have used **C++** as our primary language for the code. Our code is taking cities and distance between them as input and then creating an **Adjacency matrix** for the same, then we are using MST to form a suitable **heuristic** to apply the **A* algorithm** which in the end is giving us a suitable **path** which salesmen need to follow.

Approach

Initially, we are taking cities and distance between them as input and creating an **Adjacency matrix** for the same.

Sample Input -



```
6
0 1 4
0 2 6
0 3 15
0 4 20
0 5 9
1 2 3
1 3 8
1 4 5
1 5 12
2 3 2
2 4 18
2 5 16
3 4 14
3 5 10
4 5 11
```

Then we have to create a heuristic so that we can apply the A* Algorithm, for creating heuristic we are creating Minimum Spanning Tree (MST)

Minimum Spanning Tree

Given an undirected graph $G=(V, E)$ with arc length d . A spanning tree T is a subgraph of G that is a tree and spans all nodes, Every spanning tree has $(n-1)$ arcs basically minimum spanning tree problem is to find a spanning tree of minimum length

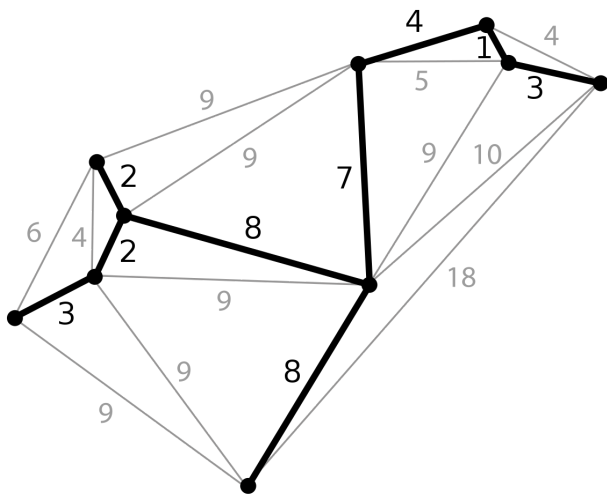
Algorithm to find MST

Step 1: find the shortest arc in the network. If there are more than one, pick anyone randomly. Highlight this arc and the nodes connected.

Step 2: Pick the next shortest arc unless it forms a cycle with the arcs already highlighted before. Highlight the arc and the nodes connected.

Step 3: If all arcs are connected, then we are done. Otherwise, repeat Step 2.

We are using prims algorithm to find MST from our graph



Prims Algorithm

The idea behind Prim's algorithm is simple: a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

Step 1: Create a set *mstSet* that keeps track of vertices already included in MST.

Step 2: Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.

Step 3: While *mstSet* doesn't include all vertices

- a) Pick a vertex u which is not there in *mstSet* and has minimum key value.
- b) Include u to *mstSet*.
- c) Update the key value of all adjacent vertices of u . To update the key values, iterate through all adjacent vertices. For every adjacent vertex v , if the weight of edge $u-v$ is less than the previous key value of v , update the key value as the weight of $u-v$

The idea of using key values is to pick the minimum weight edge from the cut. The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Code

```

void prims(int V) // To create mst of our graph and this will be our heuristic
{
    // prims algorithm

    vector<int> parent(V, -1); // parent of a vertex
    vector<int> key(V, INT_MAX); // minimum weight to reach every node from its parent
    vector<bool> mstSet(V, false); // is vertex present in mst

    key[0] = 0; // starting point

    int numberOfEdges = V - 1;

    for (int i = 0; i < numberOfEdges; i++)
    {
        /*
            mn - total_cost weight in the key array
            vertex - vertex with minimum weight in key array which is also not present in mst
        */

        int mn = INT_MAX, vertex;

        for (int j = 0; j < V; j++)
        {
            if (!mstSet[j] && key[j] < mn)
            {
                mn = key[j];
                vertex = j;
            }
        }

        mstSet[vertex] = true; // we will add this vertex into our mst

        for (int j = 0; j < V; j++)
        {
            if (!mstSet[j] && adj[vertex][j] < key[j])
            {
                parent[j] = vertex;
                key[j] = adj[vertex][j];
            }
        }
    }

    // creating heuristic

    vector<vector<int>> flag(V, vector<int>(V, 0));

    for (int k = 0; k < V; k++)
    {
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (adj[i][j] == key[k] && flag[i][j] == 0)
                {
                    mst[i][j] = 0;
                    flag[i][j] = 1;
                }
                else if (flag[i][j] == 0)
                {
                    mst[i][j] = adj[i][j];
                }
            }
        }
    }

    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (i != j)
            {
                heuristicMatrix[i][j] = adj[i][j] + mst[i][j];
            }
        }
    }

    return;
}

```


A Minimum Spanning Tree-Based Heuristic

We are iterating over our MST distance matrix and our graph (distance between graph) matrix, and if they both are the same we are adding this distance in an MST Matrix and after completion of this process, we are appending our distance matrix over it, thus doubling the distances which form MST.

After forming the heuristic we are using the A* algorithm to find the suitable path

A* Search Algorithm

- What is A* Search Algorithm?

A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals.

- Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, have “brains”. What it means is that it is really a smart algorithm that separates it from the other conventional algorithms. This fact is cleared in detail in the sections below.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Pseudo Code

Input: A graph $G(V,E)$ with source node *start* and goal node *end*.

Output: Least cost path from *start* to *end*.

Steps:

Initialise

```
open_list = { start }           /* List of nodes to be traversed */
closed_list = { }               /* List of already traversed nodes */
g(start) = 0                    /* Cost from source node to a node */
h(start) = heuristic_function(start, end) /* Estimated cost from node to goal node */
f(start) = g(start) + h(start)  /* Total cost from source to goal node */
```

while *open_list* is not empty

m = Node on top of *open_list*, with least *f*

 if *m* == *end*

 return

 remove *m* from *open_list*

 add *m* to *closed_list*

 for each *n* in *child(m)*

 if *n* in *closed_list*

 continue

cost = *g(m)* + *distance(m,n)*

 if *n* in *open_list* and *cost* < *g(n)*

 remove *n* from *open_list* as new path is better

 if *n* in *closed_list* and *cost* < *g(n)*

 remove *n* from *closed_list*

 if *n* not in *open_list* and *n* not in *closed_list*

 add *n* to *open_list*

g(n) = *cost*

h(n) = *heuristic_function(n, end)*

f(n) = *g(n)* + *h(n)*

return failure

A* search algorithm. Pseudocode of the A* search algorithm operating with open and closed lists of nodes.

Code

```
void A_Star(int V)
{
    int count = 0; // number of vertices
    int i = 0;
    long int total_cost = 0; // total cost of travel
    int min_idx = 0;

    vector<int> vertices(V, 0); // vertex visited array
    vector<vector<int>> flag(V, vector<int>(V, 0));

    vertices[0] = 1;

    cout << "Path : " << endl;

    while (count < V)
    {
        count = 0;
        int mn = INT_MAX;

        for (int j = 1; j < V; j++)
        {
            if (mn > heuristicMatrix[i][j] && heuristicMatrix[i][j] != -1 && flag[i][j] == 0)
            {
                mn = heuristicMatrix[i][j];
                min_idx = j;
            }
        }

        cout << i << " --> " << min_idx << endl;

        // so we not travel same edge again
        flag[i][min_idx] = 1;
        flag[min_idx][i] = 1;

        total_cost += adj[i][min_idx];

        i = min_idx;
        vertices[i] = 1;

        for (int k = 0; k < V; k++)
        {
            if (vertices[k])
                count++;
        }

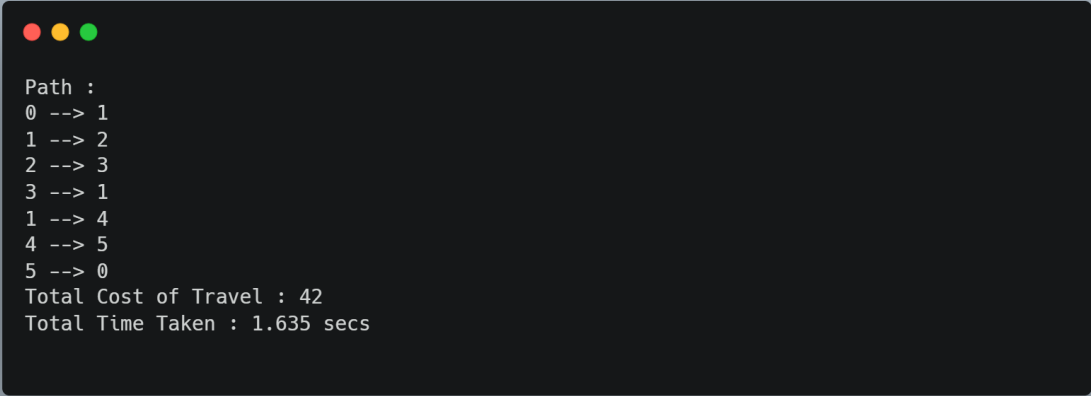
        cout << i << " --> "
             << "0" << endl;

        total_cost += adj[i][0];

        cout << "Total Cost of Travel : " << total_cost << endl;

        return;
    }
}
```

Output



```
Path :  
0 --> 1  
1 --> 2  
2 --> 3  
3 --> 1  
1 --> 4  
4 --> 5  
5 --> 0  
Total Cost of Travel : 42  
Total Time Taken : 1.635 secs
```

Time Complexity

The time complexity of the solution of travelling salesman problem with help of MST heuristic using A* Algorithm is $O(E*(V^2))$ Complexity of graph matrix method = V^2 Complexity of prim's algorithm = $E*(V^2)$ Complexity of A* algorithm = $E*(V^2) + V^2 + E*\log(V) = O(V^2)$

Current State of Art Methods

1. Nearest Neighbor:- This is perhaps the simplest and most straightforward TSP heuristic. The key to this algorithm is to always visit the nearest city.

Nearest Neighbor, $O(n^2)$

1. Select a random city.
2. Find the nearest unvisited city and go there.
3. Are there any unvisited cities left? If yes, repeat step 2.
4. Return to the first city. The Nearest Neighbor algorithm will often keep its tours within 25% of the Held-Karp lower bound.

2. Greedy:- The Greedy heuristic gradually constructs a tour by repeatedly selecting the shortest edge and adding it to the tour as long as it doesn't create a cycle with less than N edges, or increases the degree of any node to more than 2. We must not add the same edge twice of course.

Greedy, $O(n^2 \log^2(n))$.

1. Sort all edges.
2. Select the shortest edge and add it to our tour if it doesn't violate any of the above constraints.
3. Do we have N edges in our tour? If no, repeat step 2. The Greedy algorithm normally keeps within 15- 20% of the Held-Karp lower bound.

3. Insertion Heuristics:- Insertion heuristics are quite straightforward, and there are many variants to choose from. The basics of insertion heuristics are to start with a tour of a subset of all cities, and then insert the rest by some heuristic. The initial sub tour is often a triangle or the convex hull. One can also start with a single edge as a sub tour.

Nearest Insertion, $O(n^2)$.

1. Select the shortest edge, and make a sub tour of it.

2. Select a city not in the sub tour, having the shortest distance to any one of the cities in the sub tour.
3. Find an edge in the sub tour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat step 2 until no more cities remain.

Convex Hull, $O(n^2 \log^2(n))$

1. Find the convex hull of our set of cities, and make it our initial sub tour.
2. For each city not in the sub tour, find its cheapest insertion (as in step 3 of Nearest Insertion). Then choose the city with the least cost/increase ratio, and insert it.
3. Repeat step 2 until no more cities remain.

4. Christofides:- Most heuristics can only guarantee a worst-case ratio of 2 (i.e. a tour with twice the length of the optimal tour). Professor Nicos Christofides extended one of these algorithms and concluded that the worst-case ratio of that extended algorithm was $3/2$. This algorithm is commonly known as the Christofides heuristic. Original Algorithm (Double Minimum Spanning Tree), worst-case ratio 2, $O(n^2 \log^2(n))$.

1. Build a minimal spanning tree (MST) from the set of all cities.
2. Duplicating all edges, we can now easily construct an Euler cycle.
3. Traverse the cycle, but do not visit any node more than once, taking shortcuts when a node has been visited.

Christofides Algorithm, worst-case ratio $3/2$, $O(n^3)$

1. Build a minimal spanning tree from the set of all cities.

2. Create a minimum-weight matching (MWM) on the set of nodes having an odd degree. Add the MST together with the MWM.
3. Create an Euler cycle from the combined graph, and traverse it taking shortcuts to avoid visited nodes.

The main difference is the additional MWM calculation. This part is also the most time consuming one, having a time complexity of $O(n^3)$. Tests have shown that Christofides' algorithm tends to place itself around 10% above the Held-Karp lower bound.

Conclusion

We came up with a method that can find a suitable path for the travelling salesman problem so that he can travel such that he starts with city 0 and after travelling each city ends up at the starting city. We found that A* Algorithms with MST heuristics are much more efficient than traditional methods, but not as efficient as methods like Christofides Heuristic as it has worst case ratio of $3/2$ compared to 2 of MST Heuristic.

Acknowledgements

We would like to thank our professors, Dr. Poulami Dalapati and Dr. Nitin Kumar, for giving us the opportunity to make the Travelling Salesman Problem (TSP). We are overwhelmed and thankful to all those who helped put ideas into our mind and helped us research deeply on this topic. We would like to thank every person associated with us who supported us and provided all types of help on this project. Any attempt at any level can't be satisfactorily completed without the support and guidance of our parents and friends. We will be looking forward to making more such projects and learning in the process.

References

1. Christian Nilsson Linkoping University
[<http://160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf>]
2. Geeksforgeeks, Travelling Salesman Problem
[<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>]
3. Wikipedia, Travelling Salesman Problem
[https://en.wikipedia.org/wiki/Travelling_salesman_problem]
4. Florida State University, Data Set For TSP
[<https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html>]
5. Lecture Slides of Dr Poulami Dalapati and Dr Nitin Kumar.

THANK YOU