Solving TSP using Christofides Algorithm

Mohammed Al-Khrashi

438103043

Mohammed Al-Ohaydab

438101307

1. Introduction

The Traveling Salesman Problem (TSP) is one of the most famous computer science problems that has been studied intensively.

The problem description is as follows: A salesman must travel between N cities cities = $\{c_1, c_2 .., c_n\}$, such that he visits every city exactly once, and returns back to the original city. Between every two cities c_i , c_j there is an associated cost w_{ij} , the salesman wants to know the path with the least cost.

We will use Christofides Algorithm to find an approximate Solution to the TSP.

2. Christofides' Algorithm

Christofides' Algorithm is an alternative algorithm for solving the TSP, the algorithm is an approximation algorithm for finding the solution to the traveling salesman problem (TSP). The total cost of traveling that is produced by this algorithm is guaranteed to not exceed 1.5 [1] of the optimal solution cost.

The algorithm assumes an input graph G, where G is a complete graph (fully connected graph), and assumes that G is a metric graph, that is, the cost (weight) between each two vertices in G satisfies the triangle inequality.

The algorithms steps can be written as follows:

1. Find a minimum spanning tree (MST) T for G
2. Let O be the set of vertices with odd degrees in T.
3. Find the minimum weight perfect matching M for the vertices in O.
4. Combine M and T to form a multigraph H
5. Form an Eulerian circuit in H
6. Make the circuit Hamiltonian by skipping every repeated vertex. This is our path.[1]

3. Algorithm Steps Description and Implementation

In this section we will describe the algorithm steps, and how we decided to implement them.

A. Data Structures

To represent our graph G, we store our vertices in a Hash Table, where the vertex name is it's key, and it's value is an euclidean coordinate, this is done to ensure that our graph has the metric graph property.

We store our edges in a Hash Set in order to quickly access them, and see if they exist.

In order to retrieve an edge weight we store weights in a Hash Table. To get the weight between the two vertices u and v, we use the pair (u,v) as a key to get the weight associated between u and v.

We chose to value speed over space in our implementation of the graph.

B. Finding the MST

To find an MST for our graph G we used Kruskal's algorithm.

Kruskal's Algorithm is a greedy algorithm for finding an MST, it does that by adding edges to a new graph T from G based on the weights (increasingly), if adding a certain edge creates a cycle discard it, otherwise add it to the new graph T.

```
algorithm Kruskal(G) is
    A := ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            A := A ∪ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return A
```

Fig. 1 [2] Kruskal's Algorithm Pseudocode

Algorithm 1: Minimum Weight Perfect Matching

Result: Returns minimum perfect weight matching graph M

O:List of odd vertices,G: A fully metric connected graph, M: An empty graph to fill

```
while O not empty do
    v = removeLastFrom(O);
    minValue = ∞
    for each u in O do
        if weight(u,v) ¡ minValue then
            minValue = weight(u,v)
            closest = u
        end
    end
    M.addEdge(closest,v,minValue)
    O.remove(closest)
end
return M;
```

Fig .2 Pseudocode for Minimum Weight Perfect Matching algorithm.

C. Odd degree Vertices

To find the odd degree vertices, simply iterate over all vertices in T (MST for G), and check if each vertex has an odd number of neighbors, if so add it to our list O.

There should be an even number of vertices by the handshaking lemma.

D. Minimum Weight Perfect Matching of the Odd Vertices

A perfect matching of a graph is where each vertex of a graph is connected to exactly one other vertex

To find the minimum perfect matching M for the odd vertices, we use a greedy approach that takes the first vertex in O, finds the vertex closest to it then creates an edge between them in M and removes both vertices from O, until O is empty.

Note that since we have an even number of odd degrees, all vertices will have exactly one edge.

E. Forming the Multigraph

Using M and T we form a multigraph H by simply adding every edge of M into H and every edge of M into H.

Note that a multigraph allows repeated edges.

F. Eulerian Tour

From our multigraph H, we start from our root vertex V (The salesman "start" city).

We choose any neighbor of V and move to it, deleting the edge between them, if a vertex has no neighbors add it to the path, and go to the previous vertex (using a stack) [3].

G. Hamiltonian Tour

Finding our final path is simple.

Move along the Eulerian circuit, skipping every repeated vertex. This path will now be our TSP solution.

4. Experiment and Data Interpretation

To ensure that our Christofides algorithm is well tested, we started by creating the vertices and assigning a random number of euclidean

coordinates for each vertex, where the number of vertices is fixed, all vertices would have an edge with every vertex (complete graph).

We compared input size (number of vertices) vs time taken (ms) when running our algorithm, each time increasing the number of vertices by 5 in order to spot a general trend in our graph.
Each graph is executed 3 times to obtain a meaningful running time.

In Fig.3 We see a general increase in time as input size increases, with some fluctuations as some graphs happen to be easier to solve than graphs with lower size, but in general time increases with input size.
Also we can observe that time increase is much better than a standard solution! Since our solution grows polynomially rather than expenitionally.

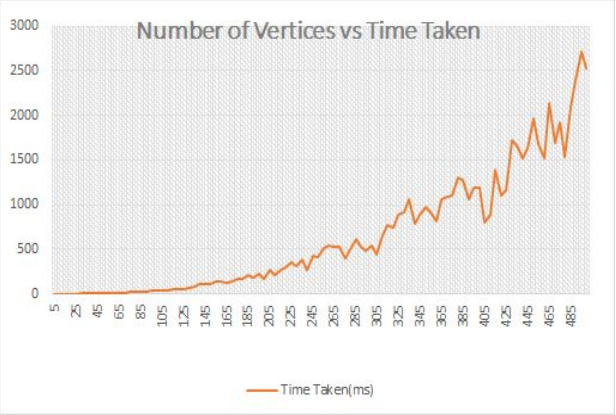

Fig .3 Time taken to run our algorithm plotted against graph size(|V|).

In Fig.4 we plotted input size against total path cost produced by our algorithm, we see a steady increase of path cost depending on the input size.
This is due to the nature of the problem where an increase of input size will increase the amount of vertices that need to be visited in the solution.

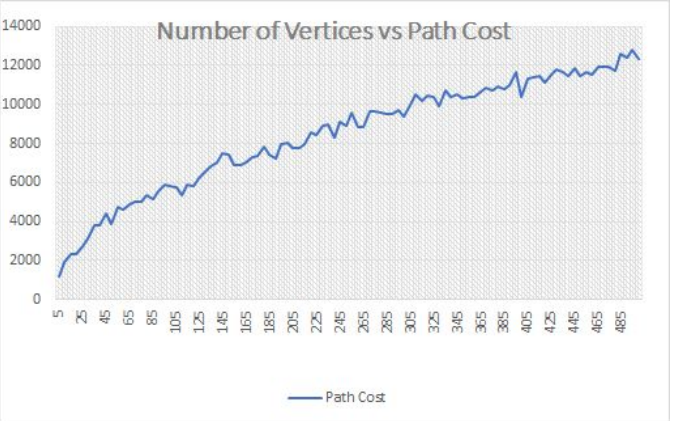

Fig .4 Path cost for our algorithm plotted against graph size(|V|).

5. Conclusions

When finding an optimal solution for a TSP is unfeasible for large input size due to the huge time complexity, approximation algorithms can be a valid alternative.

Christofides Algorithm provides us with a good approximation assuming that the graph is metric and complete.

The algorithm combines finding the MST of a graph, and finding a Minimum Weight Perfect Matching of a graph to find the final path, with both of these two algorithms having a polynomial running time.

Christofides Algorithm finds a path in polynomial time. A complexity that is way more feasible than standard algorithms for finding a TSP. With a path cost that cannot exceed 3/2 the cost of the optimal solution.

6. References

[1] Wikipedia Article Christofides Algorithm: Christofides algorithm

[2] Wikipedia Article Kruskal's Algorithm: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[3] Eulerian Path and Circuit http://www.graph-magics.com/articles/euler.php