Travelling Salesman Problem (TSP)

THORNTON Alberto-John, FIALA Vilem

1 Introduction

The Travelling Salesman Problem (TSP) is a classic optimisation challenge in computer science and operations research. The goal is to find the shortest possible route for a salesman to visit each city in a list exactly once and return to the starting city. Represented as nodes in a graph, with weighted paths that may denote distances, costs, or times, the objective is to minimize the total distance traveled or the total cost. The constraints are that every city must be visited exactly once, and the route must form a Hamiltonian cycle, returning to the origin city.

2 Algorithm Explanation

The input graph is represented in the form of a matrix $N \times N$, where N is the number of nodes in the graph. The numbers in the matrix represent the weight of edges between each node; if the weight is set to 0, it means that the edge isn't between those two nodes. We are using the matrix just for storing information about the graph, number of nodes (cities) and the weights of edges between them.

The method used is Brute Force, which tries all possible options and finds the best solution. The function takes the graph and the starting node (city). The nodes are represented by numbers from 0 to $N - 1$. Every possible permutation for the remaining cities is created, and the algorithm checks which permutation is possible and has the lowest weight (meaning the least amount of kilometers traveled).

There is a small optimisation that checks if the weight of the currently tested permutation is higher than the lowest weighted path found so far. If so, it skips checking the current permutation. The algorithm also checks if the graph does not have a Hamiltonian cycle.

The output shows the combines weight of edges of the shortest path. Followed by the path itself.

3 Algorithm in Action

3.1 Example 1: Simple Hamiltonian Graph

Input matrix:

$$\begin{bmatrix} 0 & 0 & 15 & 20 \\ 0 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}$$

Output: (95, [0, 3, 1, 2, 0])

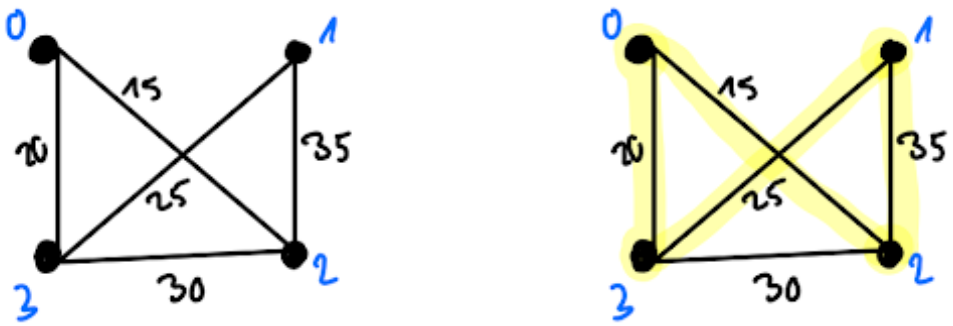


Figure 1: Simple Hamiltonian Graph

3.2 Example 2: Fully Connected Graph

Input matrix:

$$\begin{bmatrix} 0 & 8 & 25 & 28 & 20 & 15 \\ 8 & 0 & 10 & 30 & 0 & 50 \\ 25 & 10 & 0 & 12 & 44 & 51 \\ 28 & 30 & 12 & 0 & 13 & 42 \\ 20 & 0 & 44 & 13 & 0 & 14 \\ 15 & 50 & 51 & 42 & 14 & 0 \end{bmatrix}$$

Output: (72, [0, 5, 4, 3, 2, 1, 0])

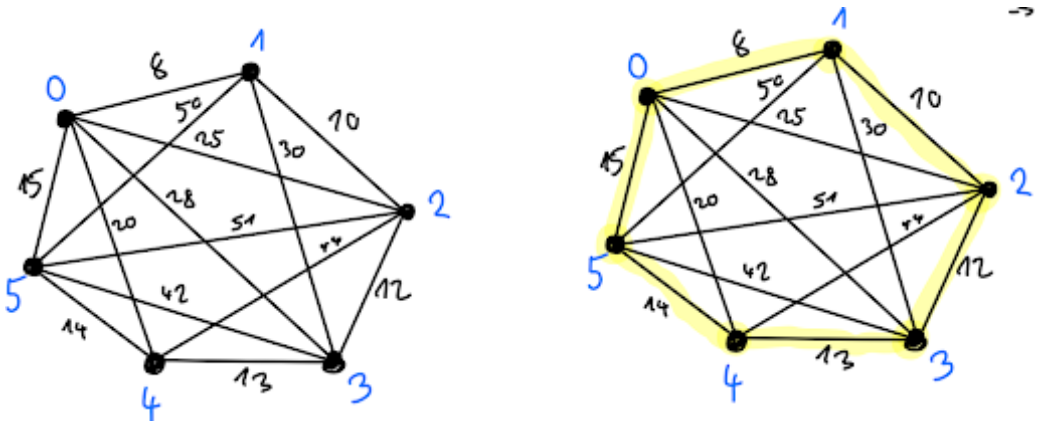


Figure 2: Fully Connected Graph

3.3 Example 3: Graph Resembling a Real-Life Map

Consider a graph resembling the historic city of York, UK. Input matrix:

$$\begin{bmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 14 & 0 & 12 & 0 & 0 & 7 \\ 0 & 14 & 0 & 11 & 4 & 0 & 0 & 0 \\ 0 & 0 & 11 & 0 & 15 & 13 & 0 & 0 \\ 0 & 12 & 4 & 15 & 0 & 9 & 7 & 8 \\ 0 & 0 & 0 & 13 & 9 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 7 & 10 & 0 & 6 \\ 5 & 7 & 0 & 0 & 8 & 0 & 6 & 0 \end{bmatrix}$$

Output: (72, [0, 5, 4, 3, 2, 1, 0])

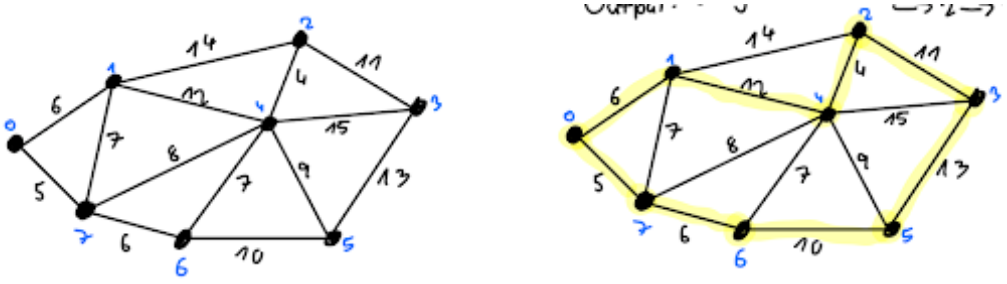


Figure 3: Graph Resembling York, UK

Figure 4: Map of York, UK. Source: Wikimedia

3.4 Example 4: Non-Hamiltonian Graph

Input matrix:

$$\begin{bmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 14 & 0 & 12 & 0 & 0 & 7 \\ 0 & 14 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 13 & 0 & 0 \\ 0 & 12 & 4 & 15 & 0 & 9 & 7 & 8 \\ 0 & 0 & 0 & 13 & 9 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 7 & 10 & 0 & 0 \\ 5 & 7 & 0 & 0 & 8 & 0 & 0 & 0 \end{bmatrix}$$

Output: “This graph is not Hamiltonian!”

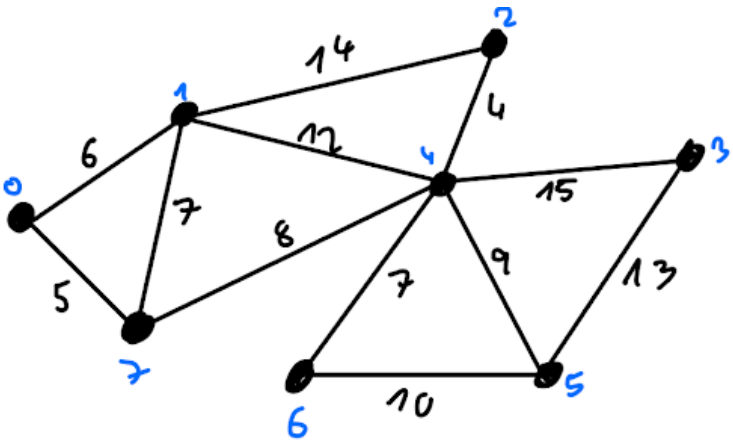


Figure 5: Non-Hamiltonian Graph

4 Problem of Non-Hamiltonian Graph

With having a Non-Hamiltonian Graph comes a problem for the salesman, as if we want to use this algorithm practically, we should be able to find a way always, maybe by taking new ways or routes. We will just present an explanation of the solution and show it as a example on one permutation in the graph from Example 4.

4.1 Explanation of the improved algorithm

The algorithm will work from the beginning the same way as our simple algorithm. Meaning we will take the matrix with the weights of edges between each nodes (cities) and number of nodes. From the number of nodes, we will create all possible permutations and start testing which permutations has the lowest combined weight. Until now it is the same as our previous algorithm.

But here comes the complicated change that we need to introduce in order to fix the problem if the Graph is Non-Hamiltonian. We can approach this from two ways, as we could include this update right from the begging of the algorithm, meaning that even if the graph is Hamiltonian, we would still look for possible changes of creating extra roads (edges) that would make the current permutation possible and it could result in better result than with just the basic graph. But maybe we would prefer to try to use the original graph if it is possible, meaning that if the graph is Hamiltonian we would never create a new edge. That would mean we first have to check if the graph is Hamiltonian, and if not, we start again going through the permutations with possibility of creating new edge.

If using our previous algorithm as a base for this improved algorithm, it will make the algorithm much slower. In every permutation, if the permutation is not possible, meaning between two nodes (cities) that should be after each other is not an edge (a road) we will find a new edge with the weight of a shortest path from the current node to the next node. That can lead to trouble that we could do this looking quite often if we have bad permutation, on the other hand, with the optimisation when we stop examining the current permutation and move to another.

We will start from the current node and implement a Dijsktra algorithm, looking for next node (city) in our current permutation. We will be calculating the cumulative distance and then use it as the weight of the new edge. And after we create this new edge, we will continue on with the normal algorithm again until we reach another dead end.

4.2 Opinion on this solution

This solution definitely solves the problem of non-Hamiltonian graph, but it is not the most optimal one. As the weight of the newly created edge is just sum of the weights of edges on the path between the nodes, it will not correspond with the layout of the cities on the map. On the other hand it also takes some time to actually build a road so we can take that as a solution.

Also as we are using pure brute force to find a solution, if we have non-Hamiltonian Graph, we will create new edge or edges every permutation, which will make the algorithm slower. With this improvement we should introduce another base algorithm, which will be more based on BST/Dijsktra and just adding few edges that would be the shortest.

4.3 Example on one permutation on non-Hamiltonian Graph

Input matrix:

$$\begin{bmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 & 5 \\ 6 & 0 & 14 & 0 & 12 & 0 & 0 & 7 \\ 0 & 14 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 13 & 0 & 0 \\ 0 & 12 & 4 & 15 & 0 & 9 & 7 & 8 \\ 0 & 0 & 0 & 13 & 9 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 & 7 & 10 & 0 & 0 \\ 5 & 7 & 0 & 0 & 8 & 0 & 0 & 0 \end{bmatrix}$$

Currently investigating permutation:

$$(0, 1, 2, 3, 4, 5, 6, 7)$$

Possible output for this permutation would be:

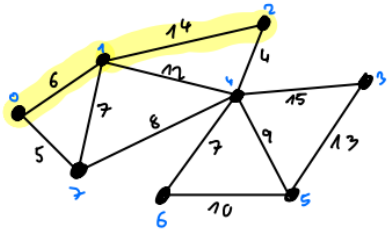
$$[93, (0, 1, 2, 3, 4, 5, 6, 7, 0)]$$

This output shows only the nodes (cities) that are actually visited, we could also output the whole walk, with the nodes that we "visit" when creating the new edges, so another possible output would be:

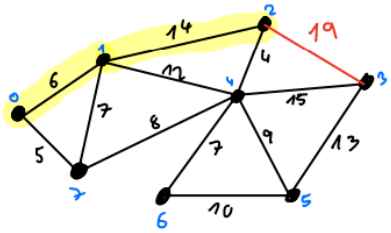
$$[93, (0, 1, 2, 4, 3, 4, 5, 6, 4, 7, 0)]$$

This output would show us the full walk and enable us to calculate better the final weight, as we can just follow the nodes in the walk.

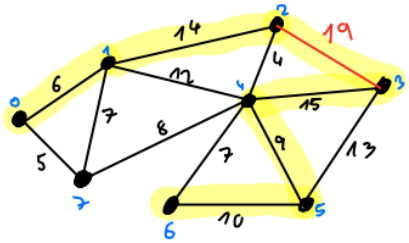
Also as this is only an example on one permutation $(0, 1, 2, 3, 4, 5, 6, 7)$ it is not necessarily the best solution for this graph, as we didn't implement this algorithm and only wanted to show how it creates new edges. Improvement could be using a permutation that only creates one edge, for example: $(0, 1, 2, 3, 5, 6, 4, 7)$, which would have weight of 82. But this one creates only one edge, and for our example we wanted to show creation of multiple edges. There are also permutations that would create new edge nearly every single time, making the walk really long, for example: $(0, 3, 7, 2, 6, 1, 5, 4)$ which would have weight of 136. This one, only going from node 5 to node 4 wouldn't create a new edge.



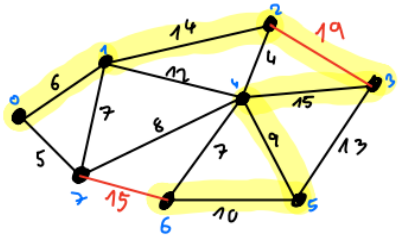
(a) Coming onto missing edge



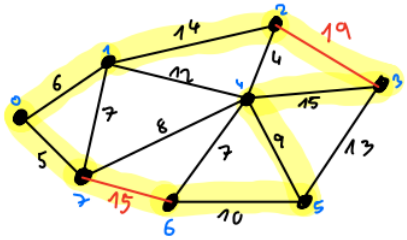
(b) Creating new edge based on shortest path



(c) Coming onto another missing edge



(d) Creating new edge based on shortest path



(e) Creating new edge based on shortest path

Figure 6: Example of the improved algorithm creating new edges