7 Traveling Problems

Activity 7.1. *Recall the definition of an Euler graph, a Hamiltonian graph, and a weighted graph described in the fourth lesson.*

7.1 Travelling Salesman Problem (TSP)

The history of the Traveling Salesman Problem (TSP) dates back to the 19th century even though its origin is unclear. The TSP was mathematically formulated by the Irish mathematician William Rowan Hamilton and by the British mathematician Thomas Kirkman.

Suppose there are several towns in an area, and a traveling salesman needs to visit each town and return to his starting position. Given the network of roads connecting the various towns on his itinerary, the traveling salesman's **problem is to find a route that minimizes his total distance traveled while ensuring that his starting and ending positions are the same**. Such a route **may visit some towns more than once.**

The network of roads can be represented by a weighted graph as follows: each town is represented by a vertex, and each road connecting two towns is represented by an edge joining the corresponding vertices, with its weight being the length of the given road. A journey that visits each town and ends back at the starting position is represented by a closed edge sequence in the graph, with its associated vertex sequence containing every vertex.

Example 1. *Consider the TSP depicted in figure 1. There are several different routes*

that will visit every town. For example, we could visit the towns in the order A, B, C, D, E, then back to A. Or we could use the route A, D, C, E, B then back to A.

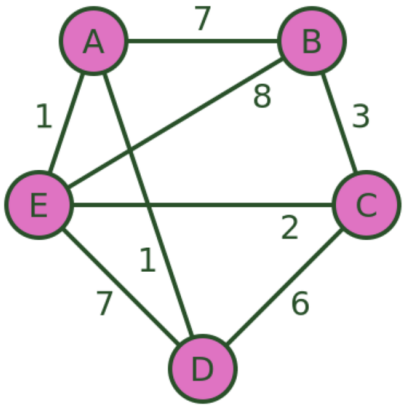

Figure 1: a road network

In graph-theoretic terms, the traveling salesman's **problem is to find a closed edge sequence of minimum total weight.**

Remark. *We assume that the graph is* ***undirected and connected****, ensuring that the salesman can indeed visit every town using the roads of the given network.*

Several approaches exist for solving the Traveling Salesman Problem (TSP), each with its own advantages and limitations. Here, we will explore three potential algorithms.

- Brute force algorithm
- Nearest neighbor algorithm
- Bellman–Held–Karp algorithm

7.1.1 Brute force algorithm

The simplest algorithm to solve the problem is the brute force approach.

Example 2. *Consider the TSP depicted in figure 1.*

- *We will assume that we start at vertex A. Since we intend to make a tour of all the vertices it doesn't matter which vertex we start at, the shortest closed loop will still be the same.*

- *Starting at A, visit vertices B, C, D, and E in some particular order, then return to A.*

- *To find the shortest route, we will try* ***every possible ordering*** *of vertices B, C, D, and E and record the cost of each one. Then we can find the shortest.*

For example, the ordering ABCDEA has a total cost of $7 + 3 + 6 + 7 + 1 = 24$.
The ordering ABCEDA has a total cost of $7 + 3 + 2 + 7 + 1 = 20$.
Some routes, such as ADEBCA are impossible because a required road doesn't exist. We can just ignore those routes.
After evaluating every possible route, we are certain to find the shortest route (or routes, as several different routes may happen to have the same length that also happens to be the shortest length). In this case, the shortest route is AECBDA with a total length of $1 + 8 + 3 + 6 + 1 = 19$.

The main problem with this algorithm is that it is very inefficient. In this example, since we have already decided that A is the start/end point, we must work out the visiting order for the 4 *towns BCDE. We have* 4 *choices of the first town,* 3 *choices for the second town,* 2 *choices for the third town, and* 1 *choice for the fourth town. So there are* 4! *combinations. That is only* 24 *combinations, which is no problem, you could even do it by hand. The brute force method is impractical for all but the most trivial scenarios.*

7.1.2 Nearest neighbour algorithm

This algorithm is sometimes called the **naive algorithm**. It is quite simple and straightforward: you start by visiting the town closest to the starting point. Then, each time you need to visit the next town, you simply choose the closest town to your current location on the map, excluding any towns you have already visited.

Example 3. *Consider the TSP depicted in figure 1.*

- *Starting at A, we visit the nearest town that we haven't visited yet. In this case, D and E are both distances* 1 *from A. There is no specific reason for choosing one*

over the other. We will prioritize the connections in alphabetical order, so we select D*. (As an aside, if we had picked* E *instead we would get a different result.)*

- *From* D*, the closest town is* C *with a distance of* 6 *(we can't pick* A *because we have already been there).*
- *From* C *the closest town is* E *with distance* 2*.*
- *From* E *we have to go to* B *because we have already visited every other town that is a distance of* 8*. And from* B *we must return to* A *with a distance of* 7*.*
- *The final path is* $ADCEBA$ *and the total distance is* $1 + 6 + 2 + 8 + 7 = 24$*.*

  The path is not the best, but it is not terrible either. This algorithm will often find a reasonable path. However, it can sometimes go badly wrong. *The main disadvantage is that the algorithm doesn't take into account the big picture.*

Exercise 1. *Suppose we apply the nearest neighbor algorithm to solve the above TSP.*

(i). What happens if you revise the graph shown in figure 1 so that the distance between A *and* B *is* 100 *units?*

(ii). What happens if you erase the edge between A *and* B*?*

7.1.3 Bellman–Held–Karp algorithm

The Bellman–Held–Karp algorithm is **a dynamic programming algorithm** for solving TSP more efficiently than brute force. The algorithm assumes the graph is complete. However, it can be used with an incomplete graph like the one we described in previous examples. To do this, we simply add the missing connections (shown below in grey in figure 2) and assign them a very large distance (for example 1000). This ensures that the missing connections will never form part of the shortest path:

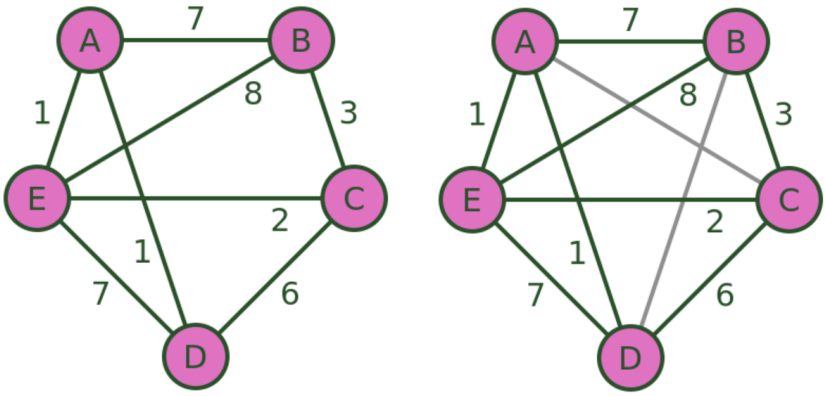

Figure 2: completion of the graph

We use the following notation in the algorithm. D_{AB} indicates the distance between towns A and B. $C_{AC(B)}$ indicates the cost (or total distance) from A to C via B.
For instance, $C_{AC(B)} = 7 + 3 = 10$, $C_{AB(C)} = 1000 + 3 = 1003$, $C_{AE(D)} = 1 + 7 = 8$.

- Fix a starting point. Here we have picked A.
- Calculate the cost of every possible combination of 2 towns starting from A. There are 12 combinations: $AB(C)$, $AB(D)$, $AB(E)$, $AC(B)$, $AC(D)$, $AC(E)$, $AD(B)$, $AD(C)$, $AD(E)$, $AE(B)$, $AE(C)$, $AE(D)$.
- Store the values for later use.
- Extend them for 3 towns starting from A. For example, $C_{AD(BC)}$ represents one of two paths $ABCD$ or $ACBD$ whichever is shorter. That is

$$C_{AD(BC)} = \min\{C_{AC(B)} + D_{CD}, C_{AB(C)} + D_{BD}\}.$$

- Repeat this for every possible combination of traveling from A to x via y and z. There are, again, 12 combinations: $AB(CD)$, $AB(CE)$, $AB(DE)$, $AC(BD)$, $AC(BE)$, $AC(DE)$, $AD(BC)$, $AD(BE)$, $AD(CE)$, $AE(BC)$, $AE(BD)$, $AE(CD)$.
- Store these values, along with the optimal path, for later use.

- Calculate the optimal route for traveling from A to any other town via 3 intermediate towns. For instance,

$$C_{AE(BCD)} = \min\{C_{AD(BC)} + D_{DE}, C_{AC(BD)} + D_{CE}, C_{AB(CD)} + D_{BE}\}.$$

- Repeat for all possible paths. This time there are only 4 combinations we need to calculate: $AB(CDE)$, $AC(BDE)$, $AD(BCE)$, and $AE(BCD)$.
- Store these values, along with the optimal path, for later use.
- Calculate the optimal route for traveling from A back to A via all 4 other towns.

$$C_{AA(BCDE)} = \min\{C_{AB(CDE)}+D_{BA}, C_{AC(BDE)}+D_{EA}, C_{AD(BCE)}+D_{DA}, C_{AE(BCD)}+D_{EA}\}.$$

- Determine the optimal root.

Remark. *Bellman–Held–Karp algorithm has poor performance for modestly large numbers of towns. The performance of the algorithm is*

$$time = O(2^n n^2) \qquad and \qquad space = O(n2^n).$$

Remark. *In all three algorithms described above, each city (vertex) is considered only once in each route. Hence they do not always provide the real optimal solution.*

Exercise 2. *What is the relationship between TSP and Hamiltonian cycles?*

Activity 7.2. *(i). Write pseudo codes for the three algorithms described above.*

(ii). Write codes for them using any programming language.

7.2 Chinese Postman Problem (CPP)

The Chinese Postman Problem (CPP), also known as the **Route Inspection Problem**. In 1962, a Chinese mathematician called Kuan Mei-Ko was interested in a postman

delivering mail to a number of streets such that the total distance walked by the postman was as short as possible.

The problem is defined on an **undirected connected graph** where each edge has a certain length or weight associated with it. The goal is to find a **closed tour that traverses each edge at least once while minimizing the total length or cost of the tour**.

Example 4. *A postman has to start at A, walk along all* 13 *streets, and return to A. See figure 3. The numbers on each edge represent the length, in meters, of each street. The problem is to find a trail that uses all the edges of a graph with minimum length.*

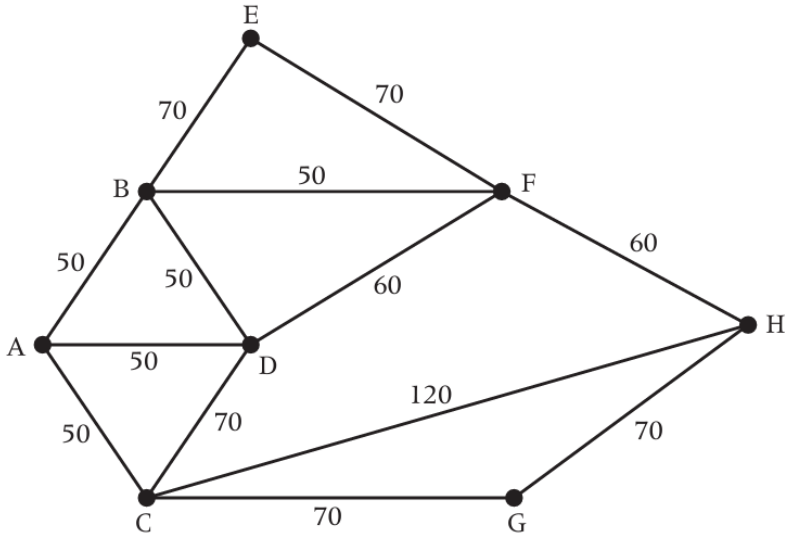

Figure 3:

Exercise 3. *Suppose there are n odd-degree vertices in a graph G. How many ways are there of pairing those odd vertices together?*

7.2.1 Chinese postman algorithm

To find a minimum Chinese postman route we must walk along each edge at least once and in addition we must also walk along the pairings of odd vertices on one extra occasion.

Step 1: List all odd vertices.

Step 2: List all possible pairings of odd vertices.

Step 3: For each pairing find the edges that connect the vertices with the minimum weight.

Step 4: Find the pairings such that the sum of the weights is minimized.

Step 5: On the original graph add the edges that have been found in Step 4.

Step 6: The length of an optimal Chinese postman route is the sum of all the edges added to the total found in Step 4.

Step 7: A route corresponding to this minimum weight can then be easily found.

Example 5. *Now let's apply the algorithm to the original problem described in figure 3.*

Step 1: The odd vertices are A and H.

Step 2: There is only one way of pairing these odd vertices, namely AH.

Step 3: The shortest way of joining A to H is using the path AB, BF, FH, a total length of 160.

Step 4: -

Step 5: Draw these edges on the original network.

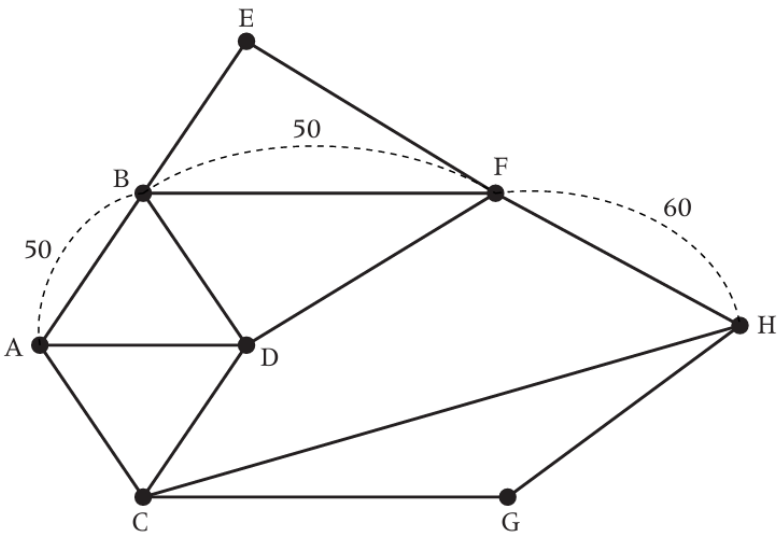

Figure 4:

Step 6: The length of the optimal Chinese postman route is the sum of all the edges in the original network, which is 840m, plus the answer found in Step 3, which is 160m. Hence the length of the optimal Chinese postman route is 1000m.

Step 7: One possible route corresponding to this length is $ADCGHCABDFBEFHFBA$, but many other possible routes of the same minimum length can be found.

Remark. *If the corresponding weighted graph is Eulerian then an Eulerian circuit provides an optimal solution.*

Homework

1. Solve the following traveling salesman problems using the Brute force algorithm, nearest neighbor algorithm, and Bellman–Held–Karp algorithm.

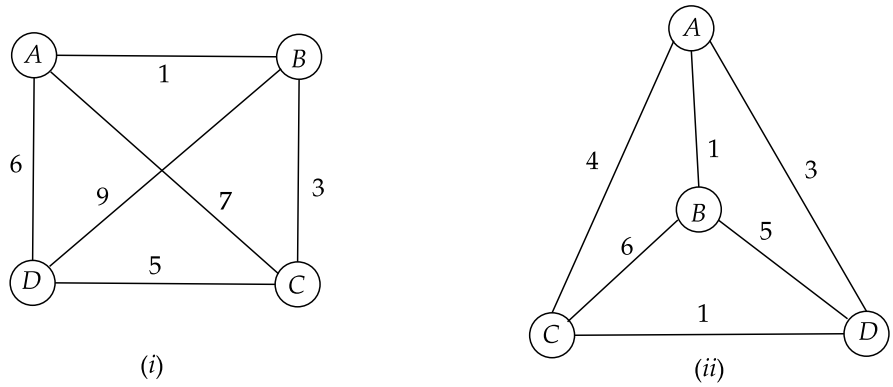

Figure 5:

2. Find the length of an optimal Chinese postman route for the network given in figure 6.

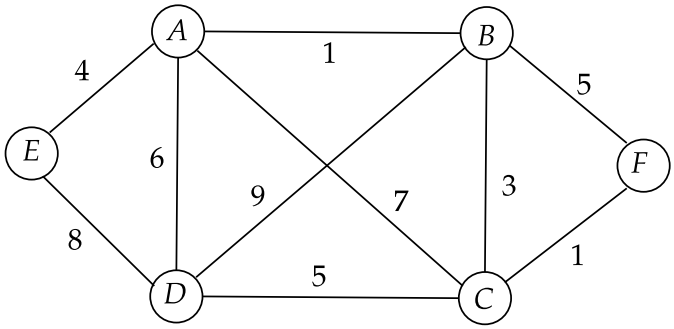

Figure 6:

3. Find the length of an optimal Chinese postman route and an optimal route for the networks below.

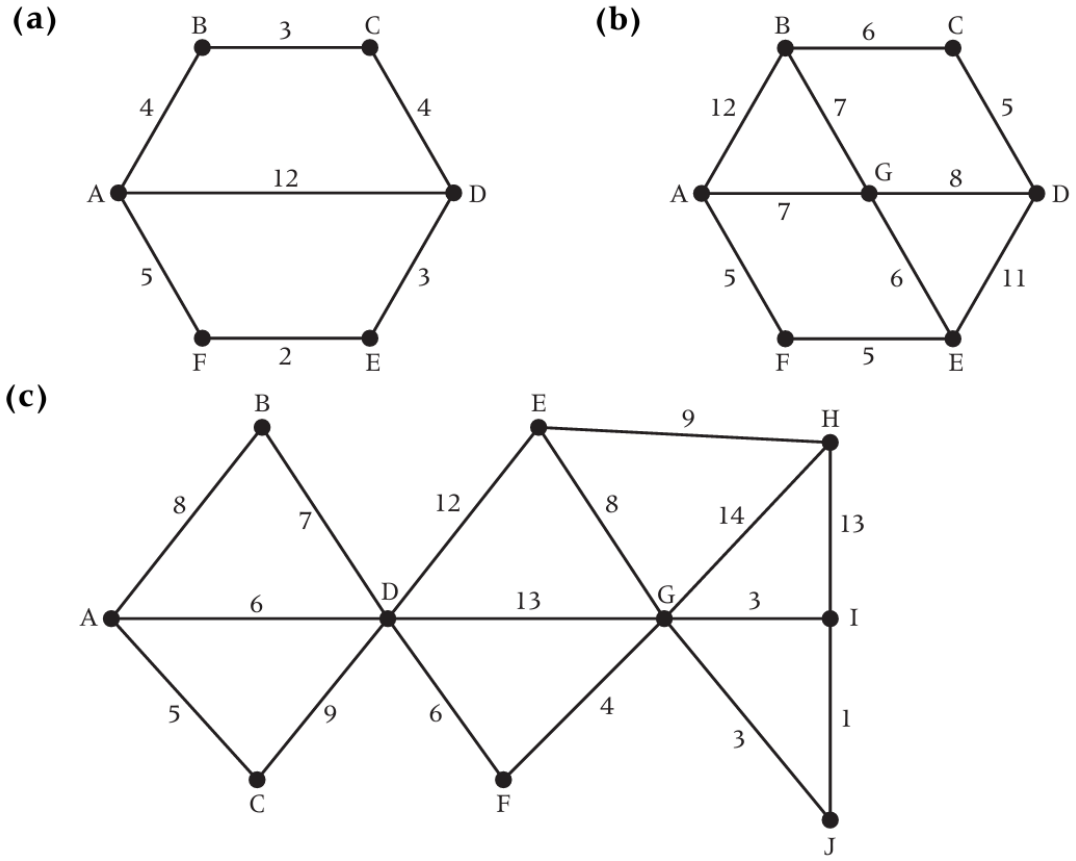

Figure 7: