- **Part A**

Miss Marple is at a train station in a foreign town. She wants to select a hotel that has the *maximum number of shortest paths* from the train station. She thinks that this should reduce the risk of getting lost. Suppose she gives you a city map represented via a graph $G = \langle V, E \rangle$ with $n = |V|$ locations and $m = |E|$ edges connecting the locations. Each edge connecting a pair of directly connected locations has a *unit distance*, say 1. Help Miss Marple to find a proper hotel by designing a $O(n + m)$ runtime algorithm that finds the number of shortest paths between the train station, located at node $s$ and every hotel on the map. You need to convince her that your algorithm is correct, and it does find all possible shortest paths. Your solution must contain an explanation and a pseudocode.

A :

We need to count the number of shortest paths from the starting point (train station) to each hotel. Since the weight of all edges in the graph is 1, the shortest path is equivalent to the path with the least number of edges. Breadth-first search (BFS) is a natural fit for this problem, because BFS traverses nodes layer by layer, ensuring that the path length recorded when a node is first visited is the shortest distance.

Initialization:

Maintain two arrays:

dist[]: Record the shortest distance from the starting point to each node, the initial value is infinity (indicating unvisited).

count []: Record the number of shortest paths from the starting point to each node, the initial value is 0.

The shortest distance of the starting point is set to 0 (dist[s]=8), and the number of shortest paths is set to 1 (count[s]=1). Use a queue to manage the nodes to be processed and initially add the starting point to the queue.

BFS traversal:

Take node u from the queue and traverse all its adjacent nodes v.

Case 1: If v's shortest distance dist[v] > dist[u] + 1, it means that a shorter path has been found. At this time, update dist[v] = dist[u] + 1, and set count[v] to count[u]. Then add v to the queue.

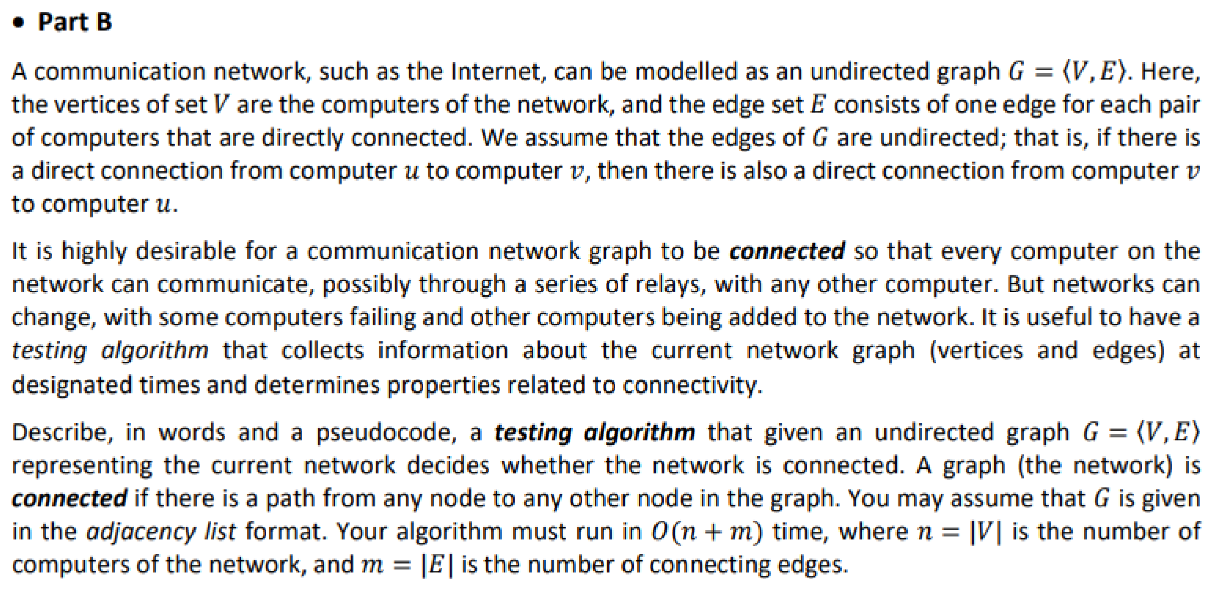
Case 2: If dist[v]== dist[u]+1, it means that there is another shortest path. At this time, the number of paths is accumulated: count[ v ] += count [ u ].

Correctness verification:

BFS layer-by-layer traversal ensures the correctness of the shortest distance. The accumulation logic of the number of paths ensures that all possible path combinations are counted. For example, if node v can be reached through two different paths with the same shortest distance, count [ v ] will correctly reflect the total number.

Time Complexity

Each node and edge is visited only once, so the time complexity is O(n + m).



- **Part B**

A communication network, such as the Internet, can be modelled as an undirected graph $G = \langle V, E \rangle$. Here, the vertices of set $V$ are the computers of the network, and the edge set $E$ consists of one edge for each pair of computers that are directly connected. We assume that the edges of $G$ are undirected; that is, if there is a direct connection from computer $u$ to computer $v$, then there is also a direct connection from computer $v$ to computer $u$.

It is highly desirable for a communication network graph to be ***connected*** so that every computer on the network can communicate, possibly through a series of relays, with any other computer. But networks can change, with some computers failing and other computers being added to the network. It is useful to have a *testing algorithm* that collects information about the current network graph (vertices and edges) at designated times and determines properties related to connectivity.

Describe, in words and a pseudocode, a ***testing algorithm*** that given an undirected graph $G = \langle V, E \rangle$ representing the current network decides whether the network is connected. A graph (the network) is ***connected*** if there is a path from any node to any other node in the graph. You may assume that $G$ is given in the *adjacency list* format. Your algorithm must run in $O(n + m)$ time, where $n = |V|$ is the number of computers of the network, and $m = |E|$ is the number of connecting edges.

A :

The connectivity of an undirected graph requires that there is a path between any two nodes. You can verify whether all nodes are reachable from a certain starting point through a traversal .

1. Initialization:

Use the Boolean array visited[] to mark whether the node has been visited, and the initial value is all false.

Start traversal from any starting point.

2. BFS traversal:

Add the starting point to the queue and mark it as visited.

Loop through the nodes in the queue:

Take out node u and traverse all its adjacent nodes v.

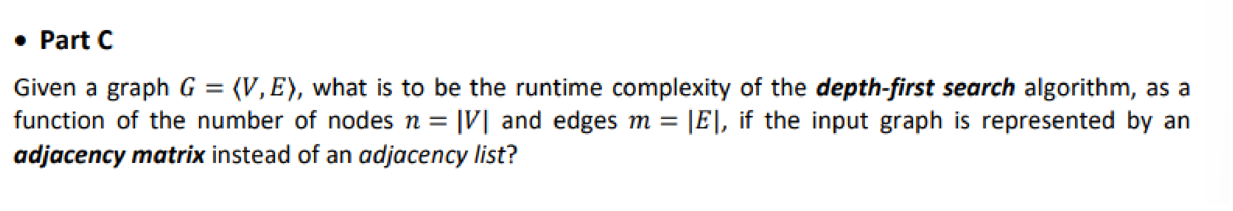
If v has not been visited, mark it as visited and add it to the queue.

3. Connectivity check:

After the traversal, check the visited[] array. If all elements are true, the graph is connected, otherwise it is not connected.

Time complexity

The time complexity of BFS traversal is O( n + m), and the final check takes O(n) time, with a total complexity of O(n + m).



- **Part C**

Given a graph $G = \langle V, E \rangle$, what is to be the runtime complexity of the ***depth-first search*** algorithm, as a function of the number of nodes $n = |V|$ and edges $m = |E|$, if the input graph is represented by an ***adjacency matrix*** instead of an *adjacency list*?

A :

When the graph is represented by an adjacency matrix, the time complexity of DFS is O(n^2).

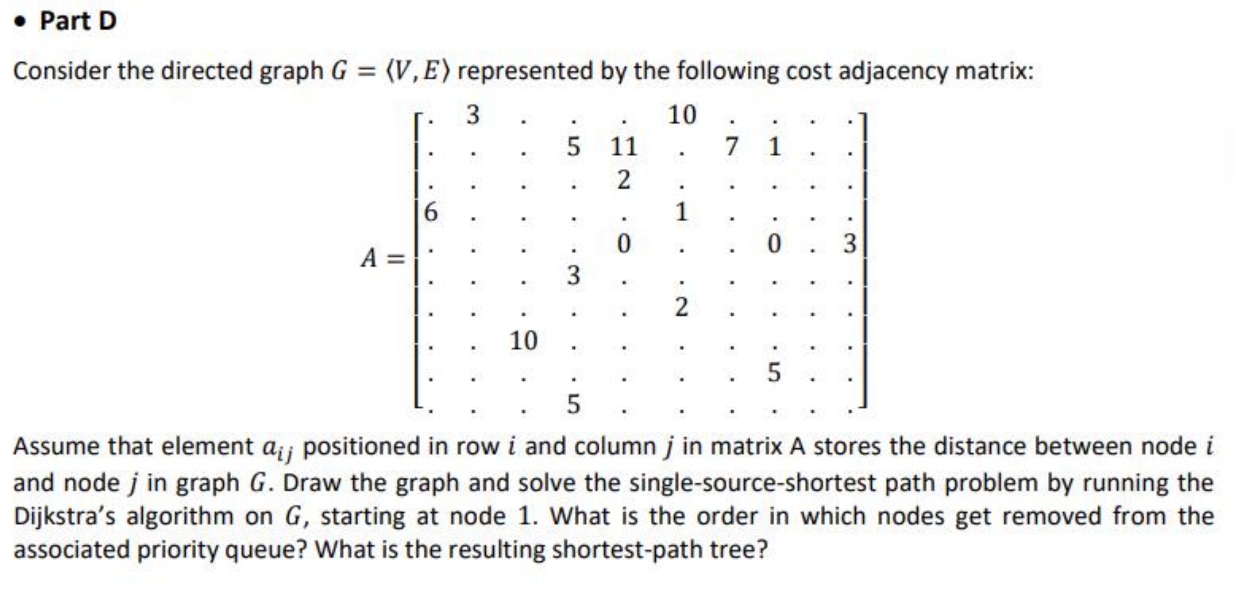
The adjacency matrix is an n * n two-dimensional array, and each node needs to traverse all possible edges.

DFS needs to visit each node once, a total of n times.

Each time a node is visited, n elements need to be traversed to find adjacent nodes, so the total time complexity is O(n × n) = O(n^2).

Compared with the adjacency list:

The adjacency list only needs to traverse the actual edges (a total of m), so the time complexity is O(n + m).



- **Part D**

Consider the directed graph $G = \langle V, E \rangle$ represented by the following cost adjacency matrix:

$$A = \begin{bmatrix} . & 3 & . & . & . & 10 & . & . & . & . \\ . & . & . & 5 & 11 & . & 7 & 1 & . & . \\ . & . & . & . & 2 & . & . & . & . & . \\ 6 & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & 0 & . & . & 0 & . & 3 \\ . & . & . & 3 & . & . & . & . & . & . \\ . & . & . & . & . & 2 & . & . & . & . \\ . & . & 10 & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 5 & . & . \\ . & . & . & 5 & . & . & . & . & . & . \end{bmatrix}$$

Assume that element $a_{ij}$ positioned in row $i$ and column $j$ in matrix A stores the distance between node $i$ and node $j$ in graph $G$. Draw the graph and solve the single-source-shortest path problem by running the Dijkstra's algorithm on $G$, starting at node 1. What is the order in which nodes get removed from the associated priority queue? What is the resulting shortest-path tree?

A：

Dijkstra algorithm execution steps

Initialization:

The dist[] array records the shortest distance from the starting point to each node. The initial values are all infinite, dist[1] = 0.

The priority queue (minimum heap) is initialized to {node 1: 0}.

Node processing order:

Step 1: Take out node 1 (distance 0) and update its adjacent nodes:

Node 2: dist[2] = 0 + 3 = 3, join the queue. [3:10]

Node 6: dist[6] = 0 +10 = 10 , join the queue. [6,10]

Prioritize Node 2

Step 2: Take out node 2 (distance 3), update its adjacent nodes:

Node 4: dist[4] = 3 + 5 = 8 → Join the queue.

Node 5: dist[5] = 3 + 11 = 14 → Join the queue.

Node 7: dist[7] = 3 + 7 = 10 → Join the queue.

Node 8: dist[8] = 3 + 1 = 4 → Join the queue.

Prioritize Node 8

Step 3: Take out node 8 (distance 4), update its adjacent nodes:

Node 3: dist[3] = 4 + 10 = 14 → added to the queue.

Step 4: Take out node 4 (distance 8) and update its adjacent nodes:

Node 6: dist[6] = min (10, 8 + 1) = 9 , join the queue.

Step 5: Take out node 6 (distance 9), update its adjacent nodes:

Node 4: dist[4] = 9 + 3 = 12 (the original distance of 8 is better)

Step 6: Take out node 7 (distance 10), update its adjacent nodes:

Node 6: dist[6] = min(9, 10 + 2) = 9

Step 7: Take out node 5 (distance 14)

Node 8: dist[8] = min(4, 14 + 0) = 4 (no update required).

Node 10: dist[10] = 14 + 3 = 17 → Join the queue.

Step 8: Take out node 3 (distance 14)

Node 5: dist[5] = min(14, 14 + 2) = 14 (no update required).

Step 9: Take out node 10 (distance 17)

Node 4: dist[4] = min(8, 17 + 5) = 8 (no update required).

Final processing order: 1 ->2 ->8 ->4 ->6 ->7 ->5 ->3 ->10.

Shortest path tree:

1 (root node).

1 -> 2 (distance 3)

1 -> 2 -> 8 (distance 4)

1 -> 2 -> 4 (distance 8)

1 -> 2 -> 4 -> 6 (distance 9)

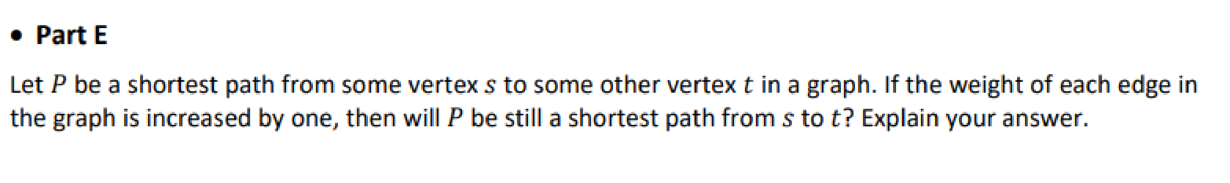
1 -> 2 -> 7 (distance 10)

1 -> 2 -> 4 -> 5 (distance 14)

1 -> 2 -> 8 -> 3 (distance 14)

1 -> 2 -> 4 -> 5 -> 10 (distance 17)

Unreachable nodes: 9



- **Part E**

Let $P$ be a shortest path from some vertex $s$ to some other vertex $t$ in a graph. If the weight of each edge in the graph is increased by one, then will $P$ be still a shortest path from $s$ to $t$? Explain your answer.

A: After increasing the weight of all edges in the graph by 1, the original shortest path may no longer be optimal. The reasons are as follows:

Nonlinearity of path weight change

The total increase in the weight of the original shortest path is the number of edges in the path. The total increase in the weight of other paths is the number of edges multiplied by 1.

The impact of the difference in the number of edges:

If the original shortest path has more edges, its total weight increase may exceed that of another path with fewer edges but slightly larger original weight.

For example: the original shortest path has 3 edges with a weight of 1 each, and the other path has 1 edge with a weight of 4. The original path is better.

After the edge weight is increased by 1, the total weight of the original path becomes 6 ($3\times2$), and the other path becomes 5 ($4 + 1$). At this time, the original path loses its optimality.

Conclusion

After the edge weight is increased, the weight of the original shortest path increases in direct proportion to its number of edges, while the weight increases of other paths may be different. Therefore, the original path may no longer be the shortest path, depending on the number of edges and the original weight distribution of each path in the graph.