CSE-2321

Data Structure

Part-8

Presented by

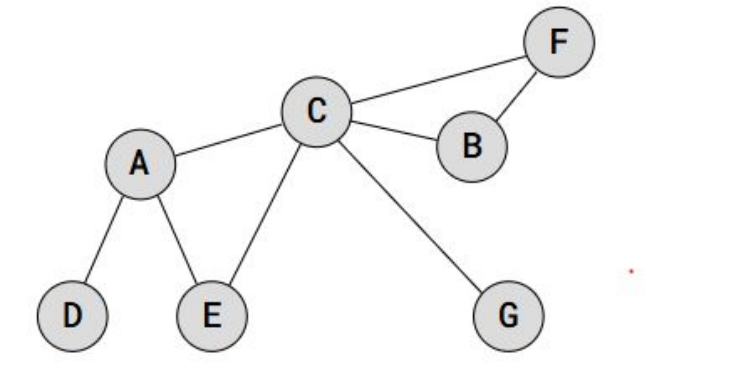
Asmaul Hosna Sadika
Adjunct Faculty
Dept of CSE, IIUC

Contents

- Introduction
- Graph terminology
- Representation of graphs – adjacency matrix, adjacency list
- Minimum Spanning Tree
- Traversing a graph
- Shortest Path algorithms

Graph

- A graph is an abstract data type (ADT) which consists of a set of objects that are connected to each other via links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.
- Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

- V = {A, B, C, D, E, F, G}
- E = {AC, AD, AE, BC, BF, CE, CF, CG}

Application of Graphs

Graphs are used to represent and solve problems where the data consists of objects and relationships between them, such as:

- **Social Networks:** Each person is a vertex, and relationships (like friendships) are the edges. Algorithms can suggest potential friends.
- **Maps and Navigation:** Locations, like a town or bus stops, are stored as vertices, and roads are stored as edges. Algorithms can find the shortest route between two locations when stored as a Graph.
- **Internet**: Can be represented as a Graph, with web pages as vertices and hyperlinks as edges.
- **Biology:** Graphs can model systems like neural networks or the spread of diseases.

Terminology

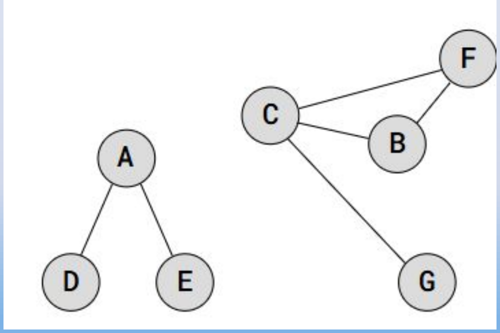
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represent edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D

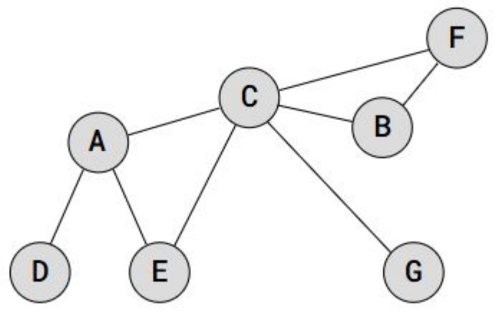
❑ **Closed path -**A path P is known as a closed path if the edge has the same end-points.

❑ **Simple path** - A path P is known as a simple path if all the nodes in the path are distinct with the exception that v0 may be equal to vn .

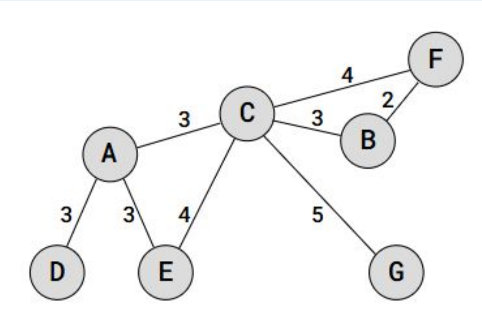
Cycle: A cycle is a closed simple path with length 3 or more. A simple cycle has no repeated edges or vertices (except the first and last vertices). A cycle of length k is called a K-cycle.



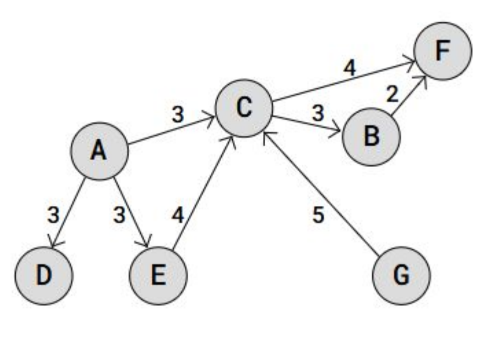
A **connected** Graph is when all the vertices are connected through edges somehow. A Graph that is not connected, is a Graph with isolated (disjoint) subgraphs, or single isolated vertices.



A **weighted** Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability.



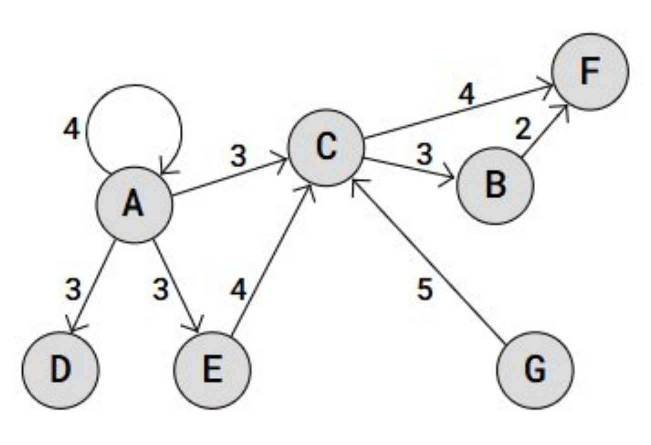
A **directed** Graph, also k
the edges between the vertex pairs have a
direction. The direction c
gs like hierarchy or flow.



A cyclic Graph is defined differently depending on whether it is directed or not:

- A **directed cyclic** Graph is when you can follow a path along the directed edges that goes in circles. Removing the directed edge from F to G in the animation above makes the directed Graph not cyclic anymore.
- An **undirected cyclic** Graph is when you can come back to the same vertex you started at without using the same edge more than once. The undirected Graph above is cyclic because we can start and end up in vertes C without using the same edge twice.

A **loop**, also called a self-loop, is an edge that begins and ends on the same vertex. A loop is a cycle that only consists of one edge. By adding the loop on vertex A in the animation above, the Graph becomes cyclic.



Complete graph

A graph G is said to be complete if every node u in G is adjacent to every other node v in G. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

Tree

A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph. If T is a finite tree with m nodes, then T will have m-1 edges.

Multi-graph

A graph with multiple edges and/or loops is called a multi-graph. Distinct edges which connect the same end-points are called multiple edges. That is, e = [u, v] and e' = [u, v] are known as multiple edges of G.

Terminology of a Directed Graph

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it.

An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge **e** = (u, v) [e also called an arc]

- The edge begins at u and terminates at v.
- u is known as the **origin or initial point** of e. Correspondingly, v is known as the **destination or terminal point** of e.
- u is the **predecessor** of v. Correspondingly, v is the **successor or neighbor** of u.
- u is adjacent to v, and v is adjacent to u.

Out-degree of a node

The out-degree of a node u, written as **outdeg(u)**, is the number of edges that originate at u.

In-degree of a node

The in-degree of a node u, written as **indeg(u)**, is the number of edges that terminate at u.

Pendant vertex (also known as leaf vertex) - A vertex with degree one.

Terminology of a Directed Graph

Source

A node u is known as a source if it has a positive out-degree but a zero in-degree.

Sink

A node u is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability

A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Strongly connected directed graph

A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally connected graph

A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

Weakly connected digraph

A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1

Simple directed graph

A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

Representation of Graphs

A Graph representation tells us how a Graph is stored in memory.

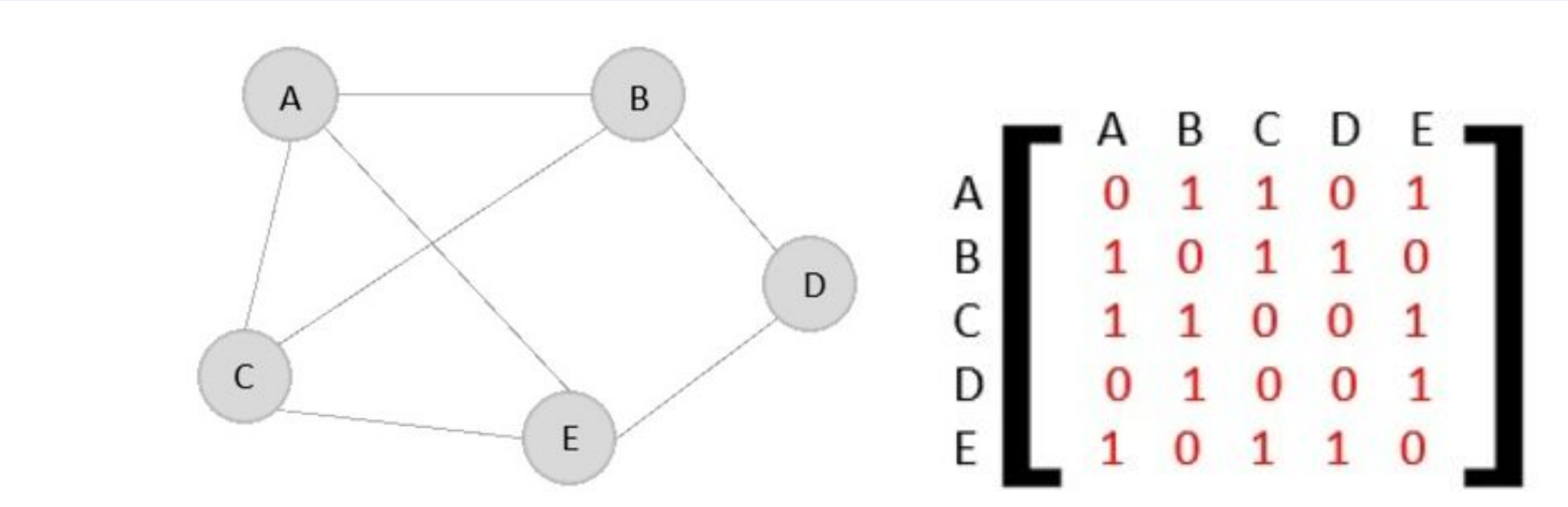
Different Graph representations can:

- take up more or less space.
- be faster or slower to search or manipulate.
- be better suited depending on what type of Graph we have (weighted, directed, etc.), and what we want to do with the Graph.
- be easier to understand and implement than others.

Graph representations store information about which vertices are adjacent, and how the edges between the vertices are. Graph representations are slightly different if the edges are directed or weighted

Adjacency Matrix Graph Representation

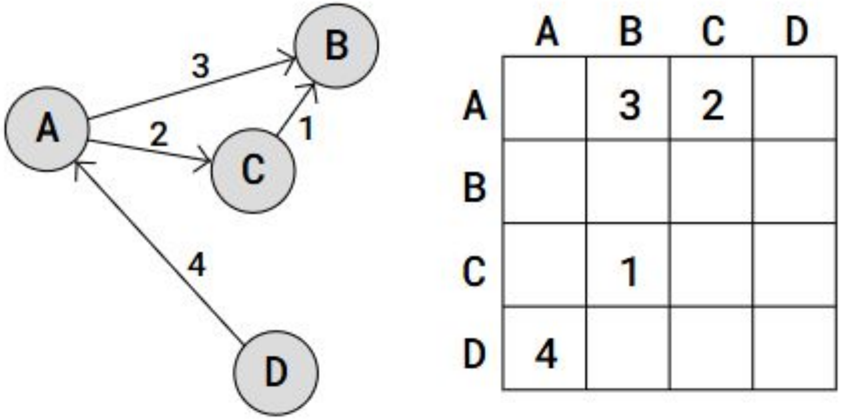
The Adjacency Matrix is a 2D array (matrix) where each cell on index (i,j) stores information about the edge from vertex i to vertex j. The Adjacency Matrix is a V x V matrix where the values are filled with either 0 or 1. If the link exists between Vi and Vj, it is recorded 1; otherwise, 0.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 1 & 1 & 0 & 0 & 1 \\ D & 0 & 1 & 0 & 0 & 1 \\ E & 1 & 0 & 1 & 1 & 0 \end{array}$$

The adjacency matrix above represents an undirected Graph, so the values '1' only tells us where the edges are. Also, the values in the adjacency matrix is symmetrical because the edges go both ways (undirected Graph).

- To create a directed Graph with an adjacency matrix, we must decide which vertices the edges go from and to, by inserting the value at the correct indexes (i,j). To represent a weighted Graph we can put other values than '1' inside the adjacency matrix.
- Below is a directed and weighted Graph with the Adjacency Matrix representation next to i

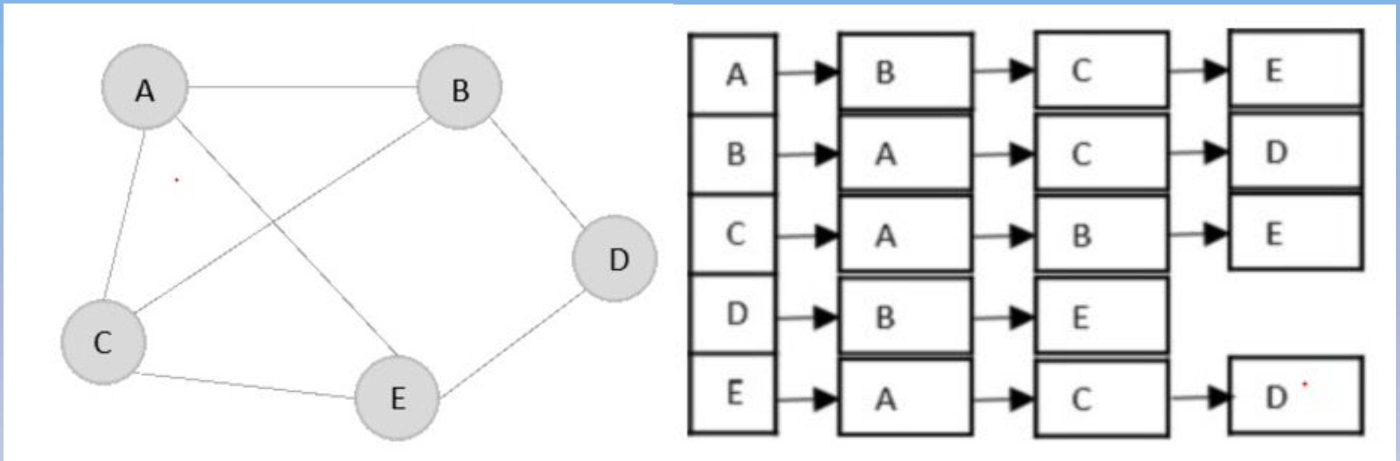


	A	B	C	D
A		3	2	
B				
C		1		
D	4			

A directed and weighted Graph, and its adjacency matrix.

Adjacency List Graph Representation

- In case we have a 'sparse' Graph with many vertices, we can save space by using an Adjacency List compared to using an Adjacency Matrix, because an Adjacency Matrix would reserve a lot of memory on empty Array elements for edges that don't exist.
- A 'sparse' Graph is a Graph where each vertex only has edges to a small portion of the other vertices in the Graph.
- An Adjacency List has an array that contains all the vertices in the Graph, and each vertex has a Linked List (or Array) with the vertex's edges.



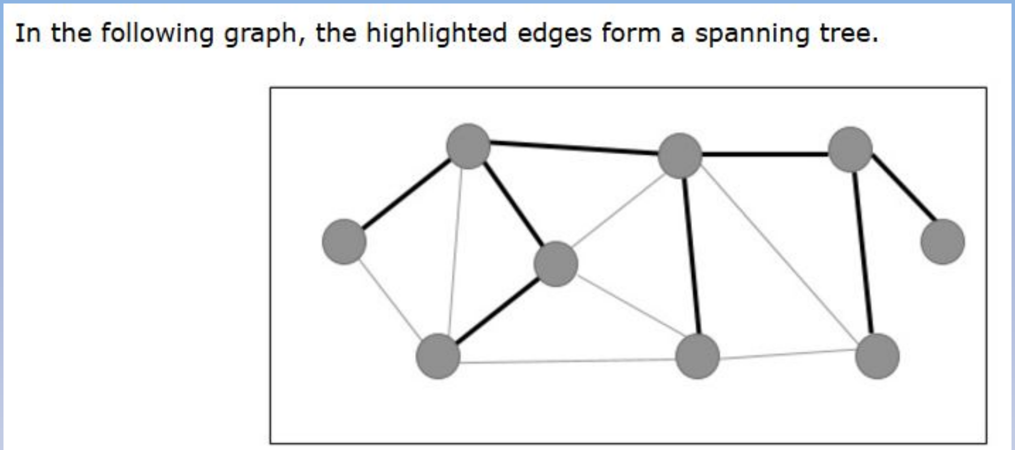
Spanning Tree

A spanning tree is a subset of an undirected graph that contains all the vertices of the graph connected with the minimum number of edges in the graph. Precisely, the edges of the spanning tree is a subset of the edges in the original graph.

If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

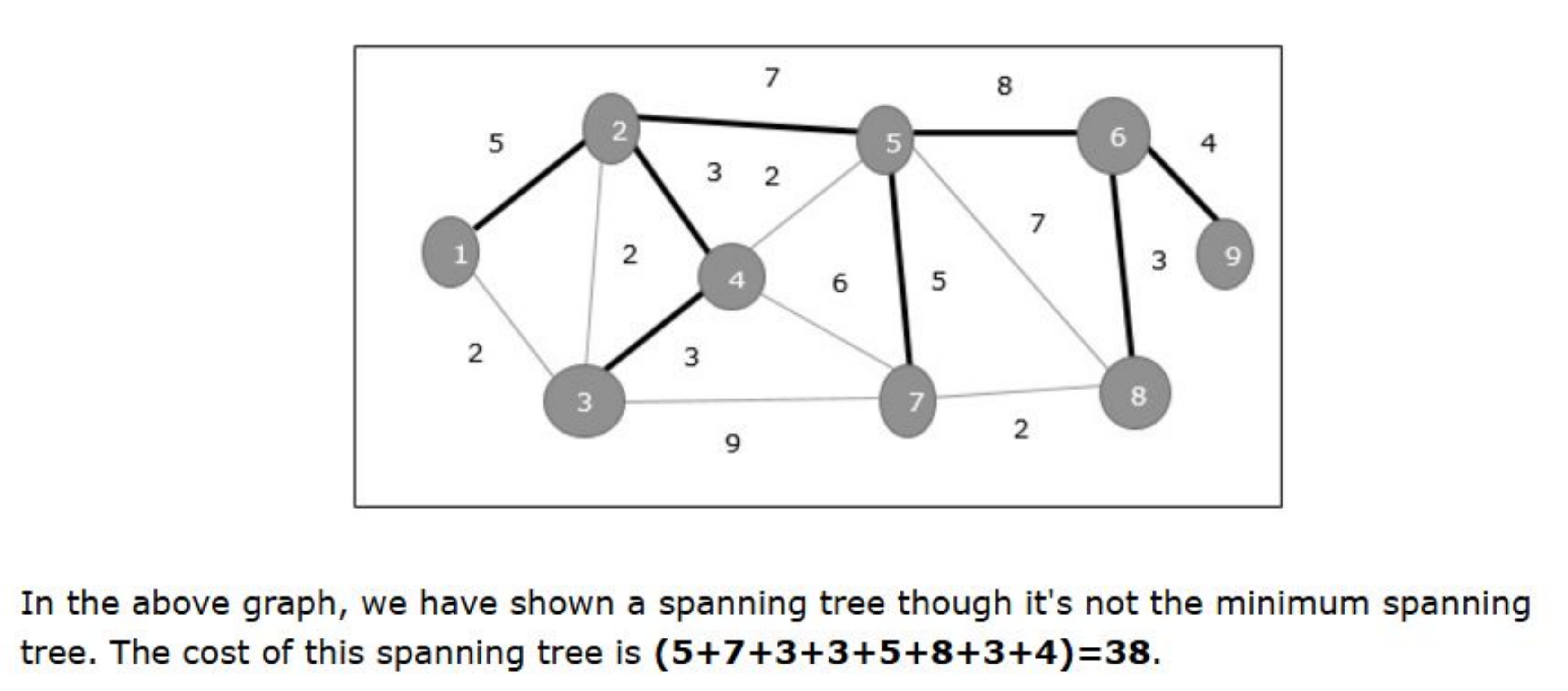


In the following graph, the highlighted edges form a spanning tree.

Minimum Spanning Tree (MST)

- A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.
- As we have discussed, one graph may have more than one spanning tree. If there are n number of vertices, the spanning tree should have n − 11 number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.
- Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.

Minimum Spanning Tree (MST)



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is **(5+7+3+3+5+8+3+4)=38**.

Traversing A graph

The primary operations of a graph include creating a graph with vertices and edges, and displaying the said graph. However, one of the most common and popular operation performed using graphs are Traversal, i.e. visiting every vertex of the graph in a specific order.

There are two types of traversals in Graphs −

- Depth First Search Traversal
- Breadth First Search Traversal

Breadth First Search Traversal

- Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes.
- While using BFS for traversal, any node in the graph can be considered as the root node.
- There are many ways to traverse the graph, but among them, BFS is the most commonly used approach.
- It is a recursive algorithm to search all the vertices of a tree or graph data structure.
- BFS puts every vertex of the graph into two categories - visited and non-visited.
- It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

BFS Algorithm

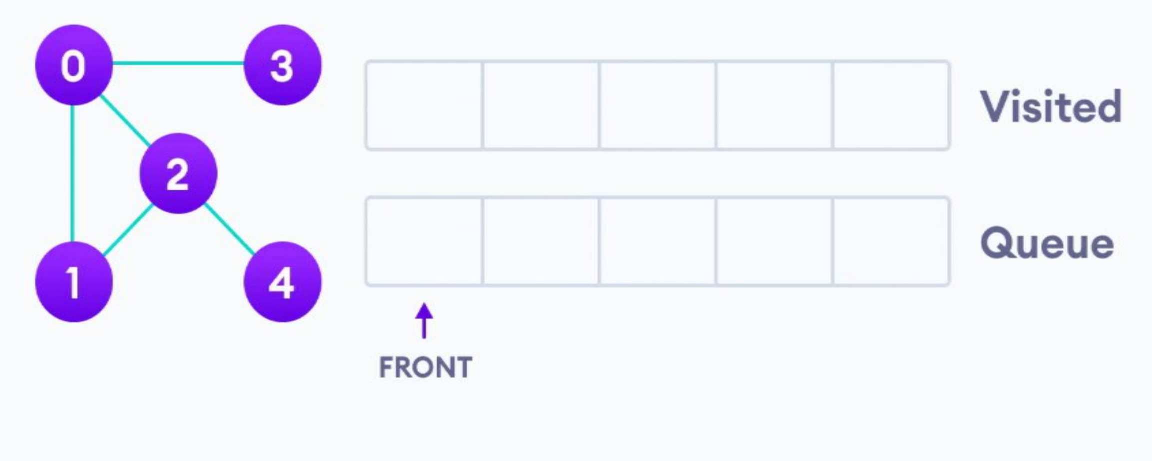
A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

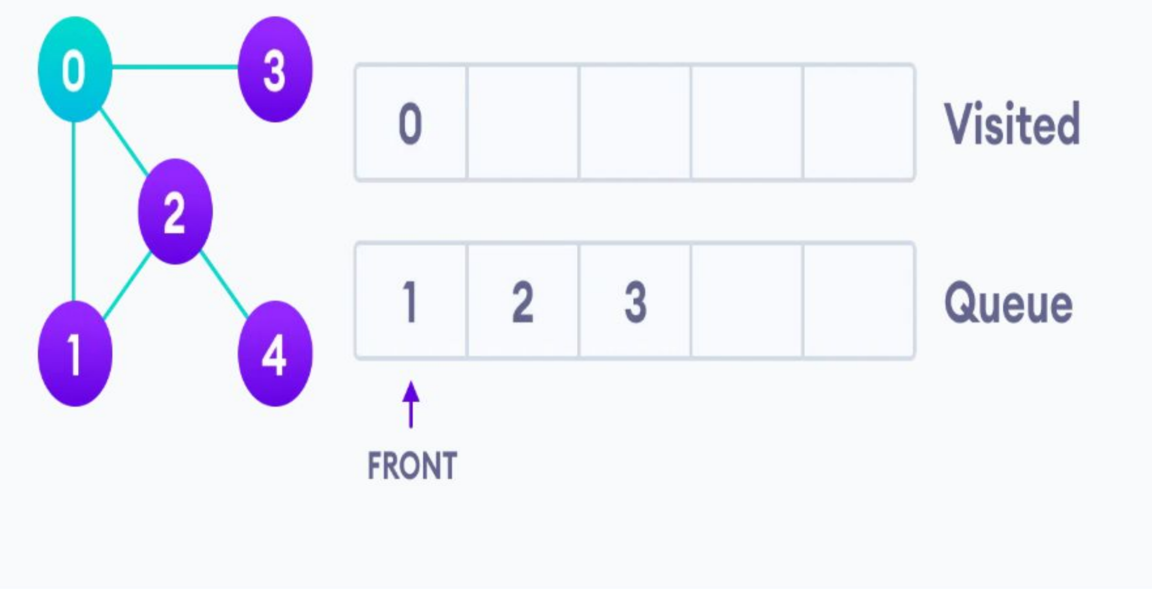
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

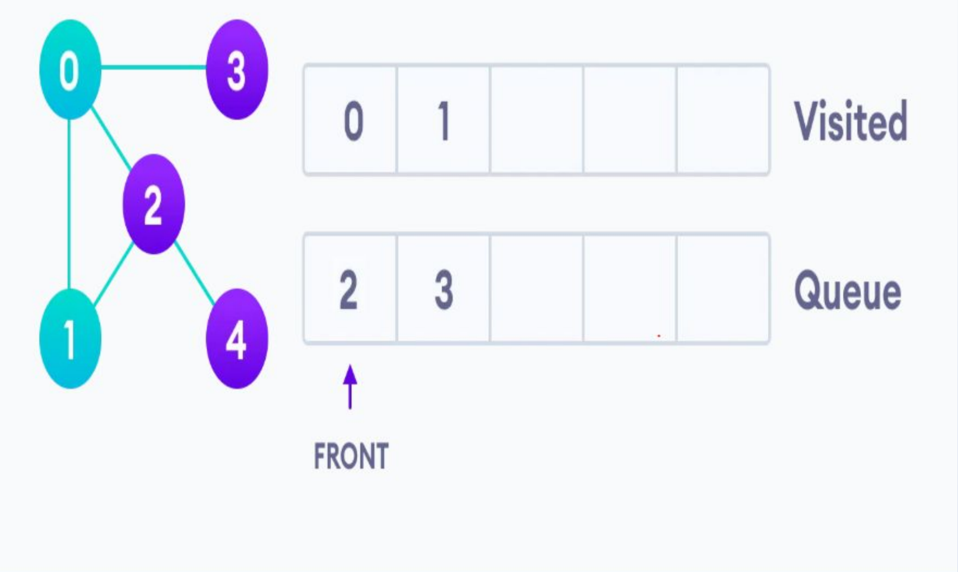
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones that aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.
5. The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node



Undirected graph with 5 vertices

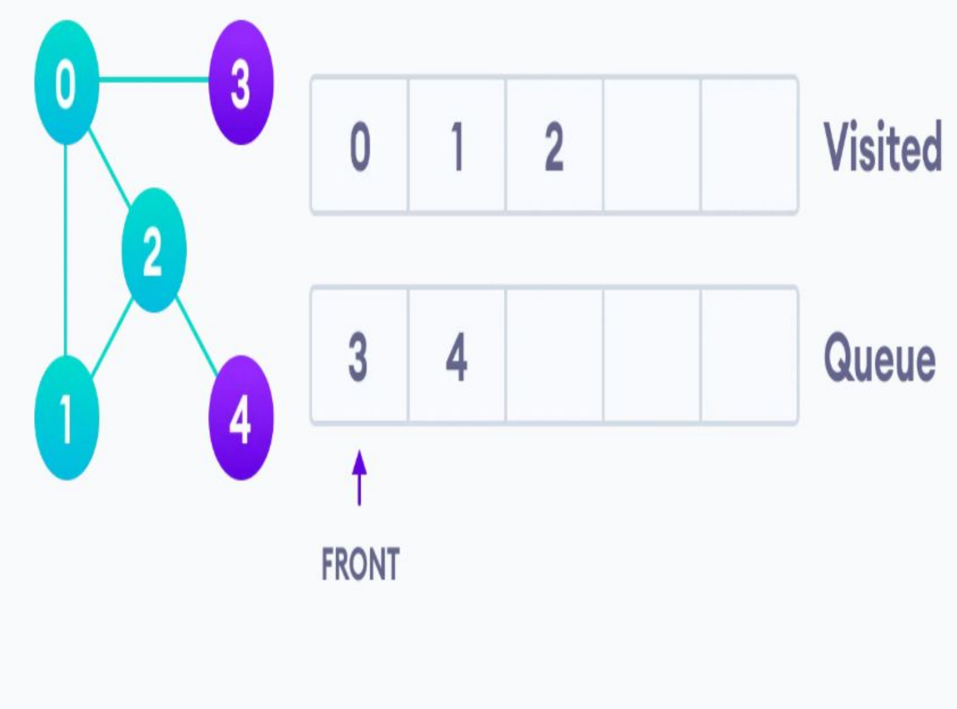


Visit start vertex and add its adjacent vertices to queue



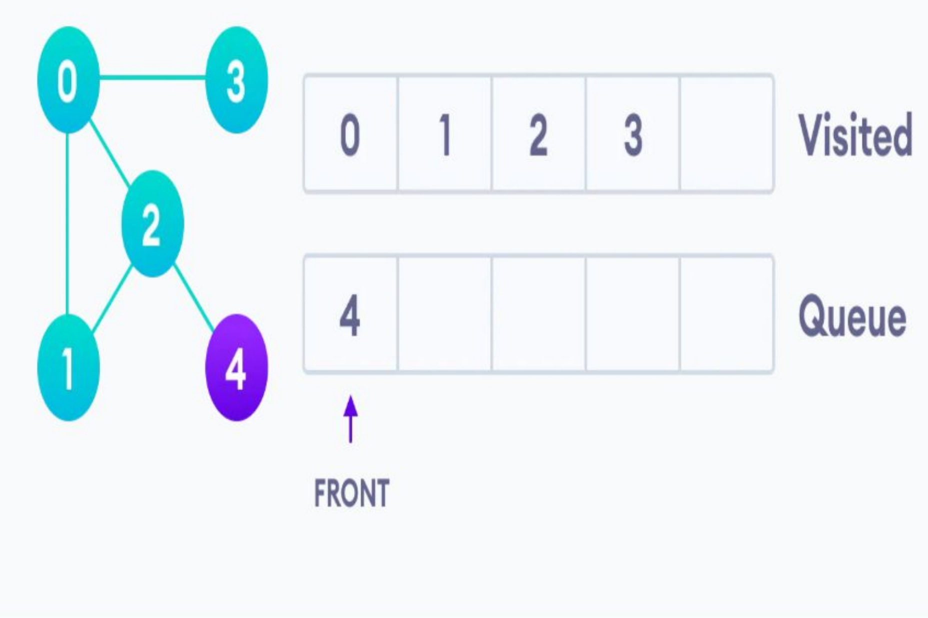
0	1				Visited
2	3				Queue

Visit the first neighbour of start node 0, which is 1



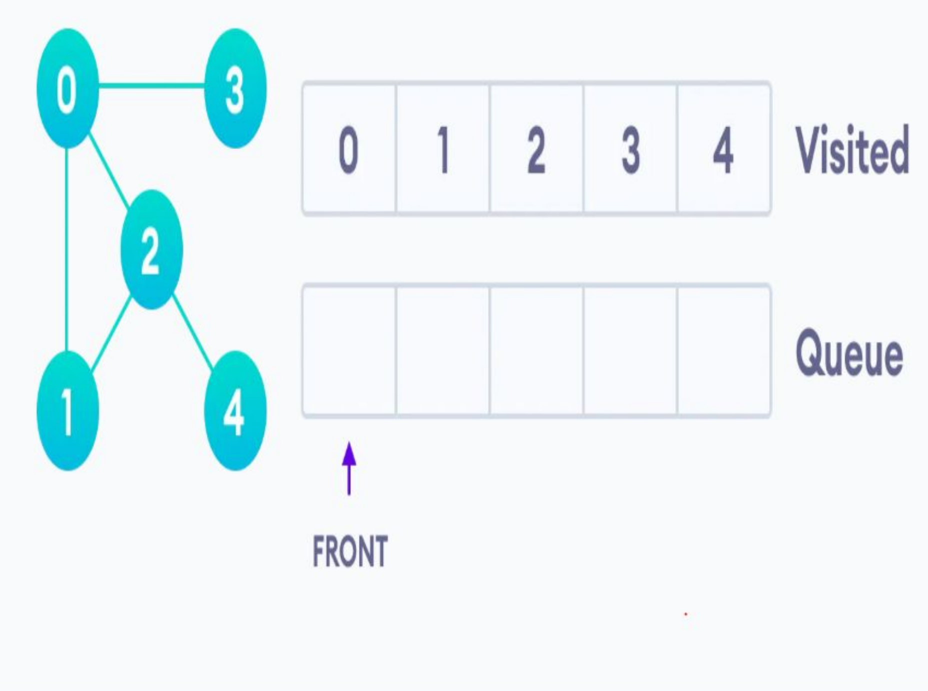
0	1	2			Visited
3	4				Queue

Visit 2 which was added to queue earlier to add its neighbours



0	1	2	3		Visited
4					Queue

4 remains in the queue



0	1	2	3	4	Visited
					Queue

Visit last remaining item in the queue to check if it has unvisited neighbors

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Depth First Search

- Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure. This algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.
- It employs the following rules.

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

DFS Algorithm

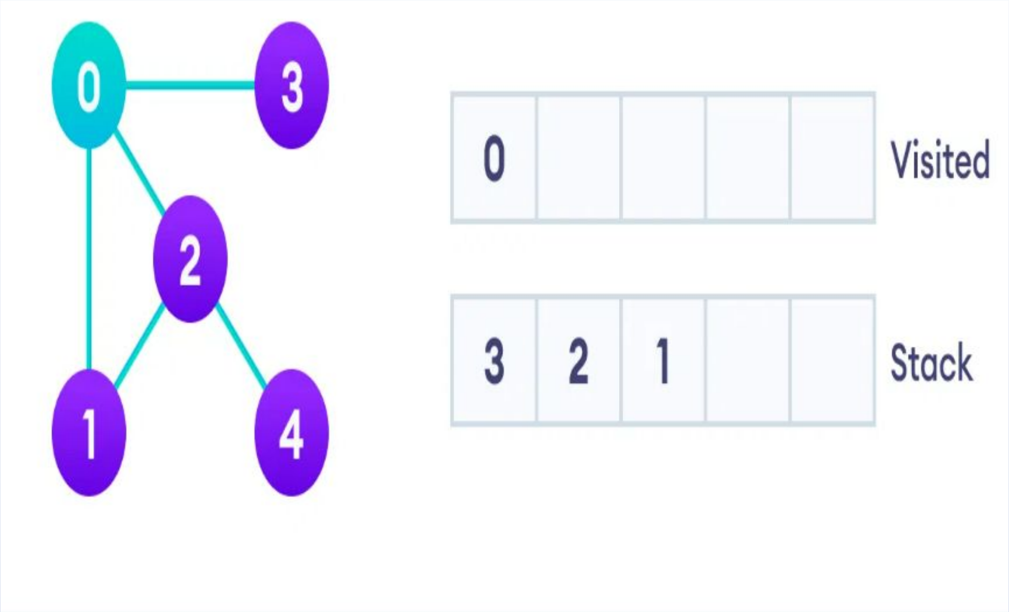
A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

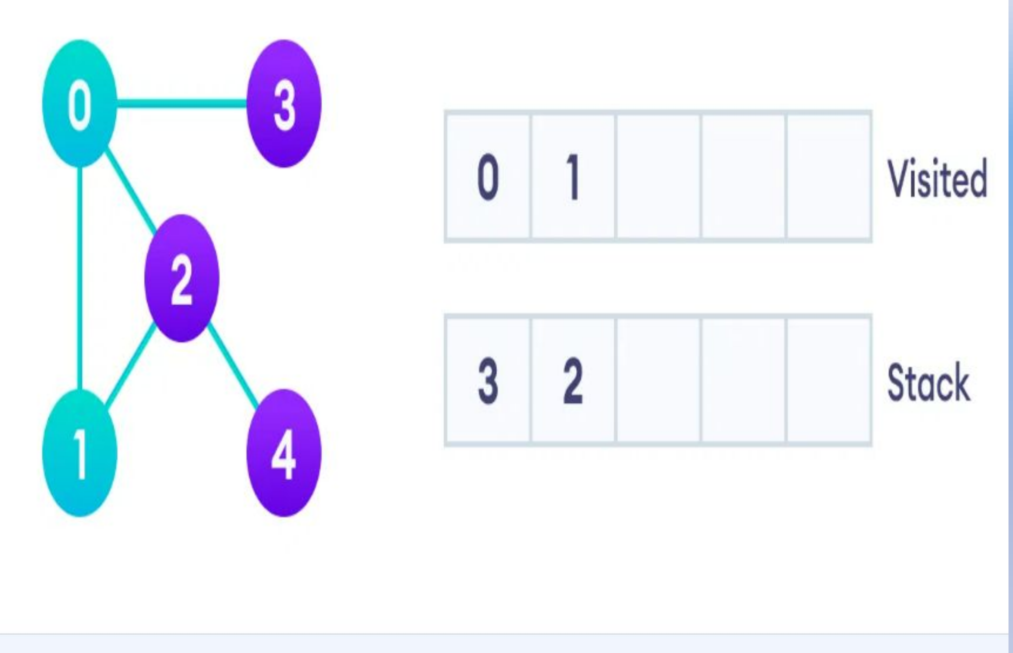
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.



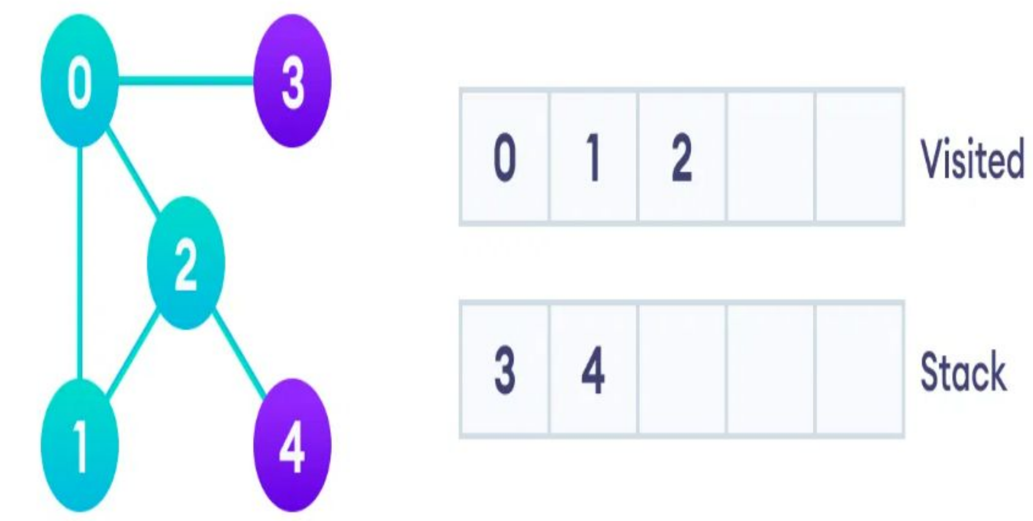
0					Visited
3	2	1			Stack

Visit the element and put it in the visited list



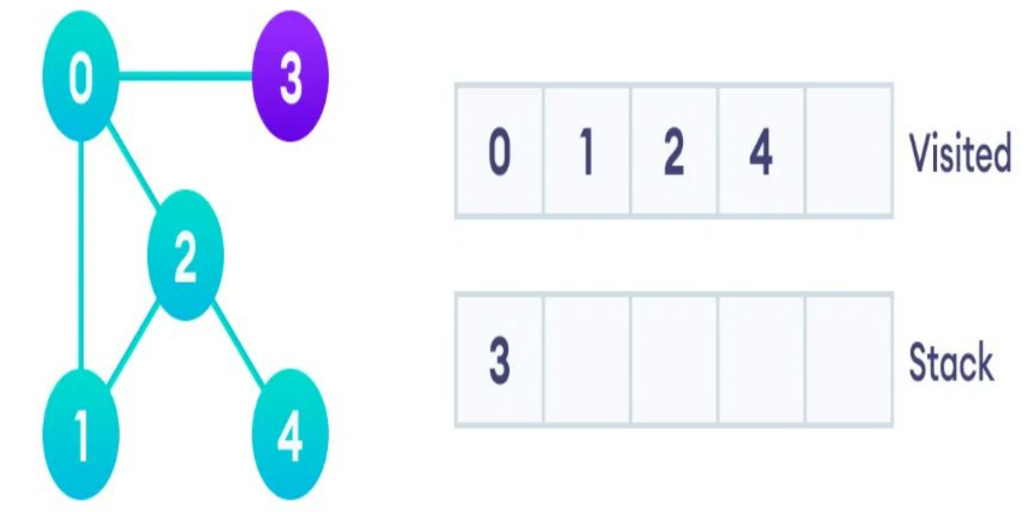
0	1				Visited
3	2				Stack

Visit the element at the top of stack



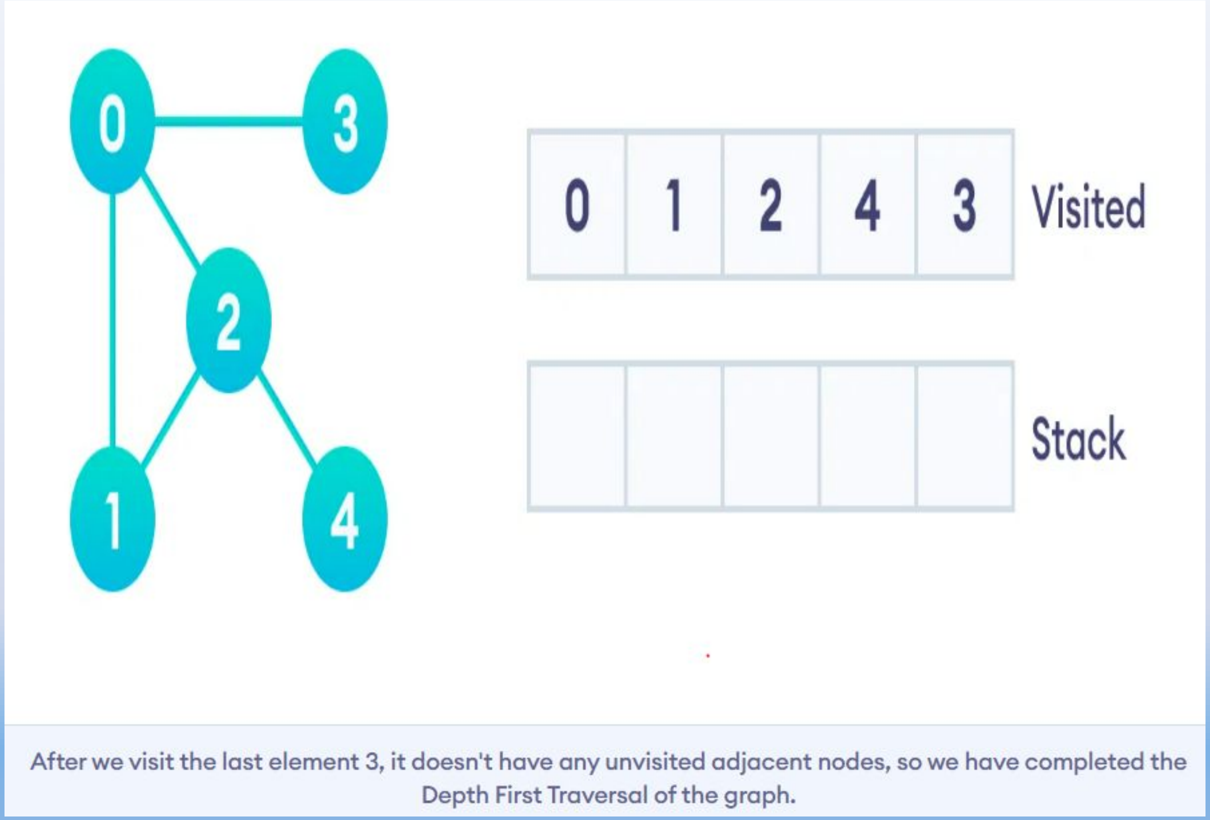
0	1	2			Visited
3	4				Stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



0	1	2	4		Visited
3					Stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



0	1	2	4	3	Visited
					Stack

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V)

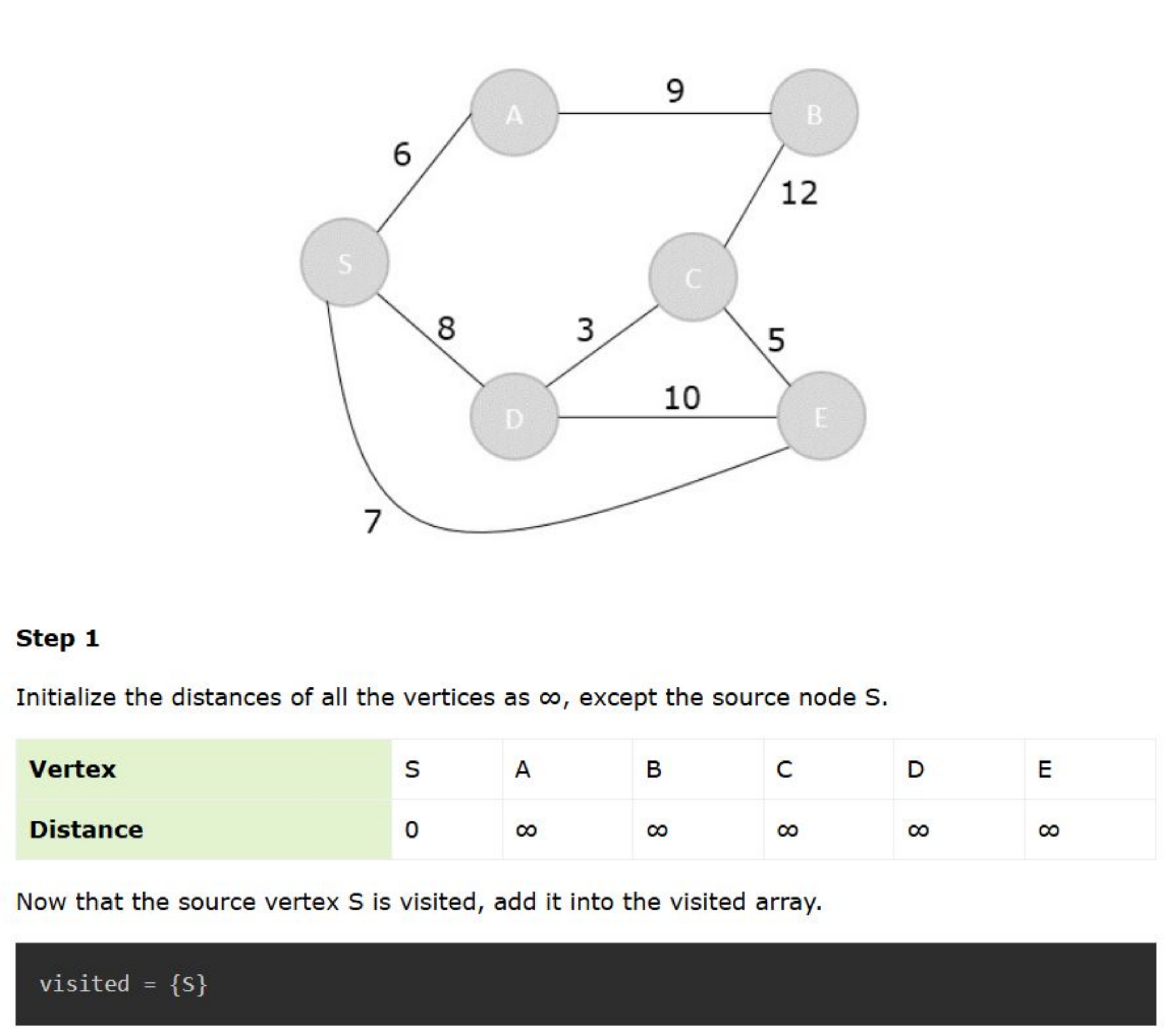
Dijkstra’s Shortest Path Algorithm

- The dijkstra’s algorithm is designed to find the shortest path between two vertices of a graph. These two vertices could either be adjacent or the farthest points in the graph. The algorithm starts from the source. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. And the output is the shortest path spanning tree.
- Dijkstra’s algorithm can be performed on both directed and undirected graphs.
- Dijkstra’s algorithm is also called **single-source shortest path algorithm**. The output obtained is called **shortest path spanning tree**.

Dijkstra’s Shortest Path Algorithm

Algorithm

- Declare two arrays − *distance*[] to store the distances from the source vertex to the other vertices in graph and *visited*[] to store the visited vertices.
- Set distance[S] to ‘0’ and distance[v] = ∞, where v represents all the other vertices in the graph.
- Add S to the visited[] array and find the adjacent vertices of S with the minimum distance.
- The adjacent vertex to S, say A, has the minimum distance and is not in the visited array yet. A is picked and added to the visited array and the distance of A is changed from ∞ to the assigned distance of A, say d_1, where $d_1 < \infty$.
- Repeat the process for the adjacent vertices of the visited vertices until the shortest path spanning tree is formed.



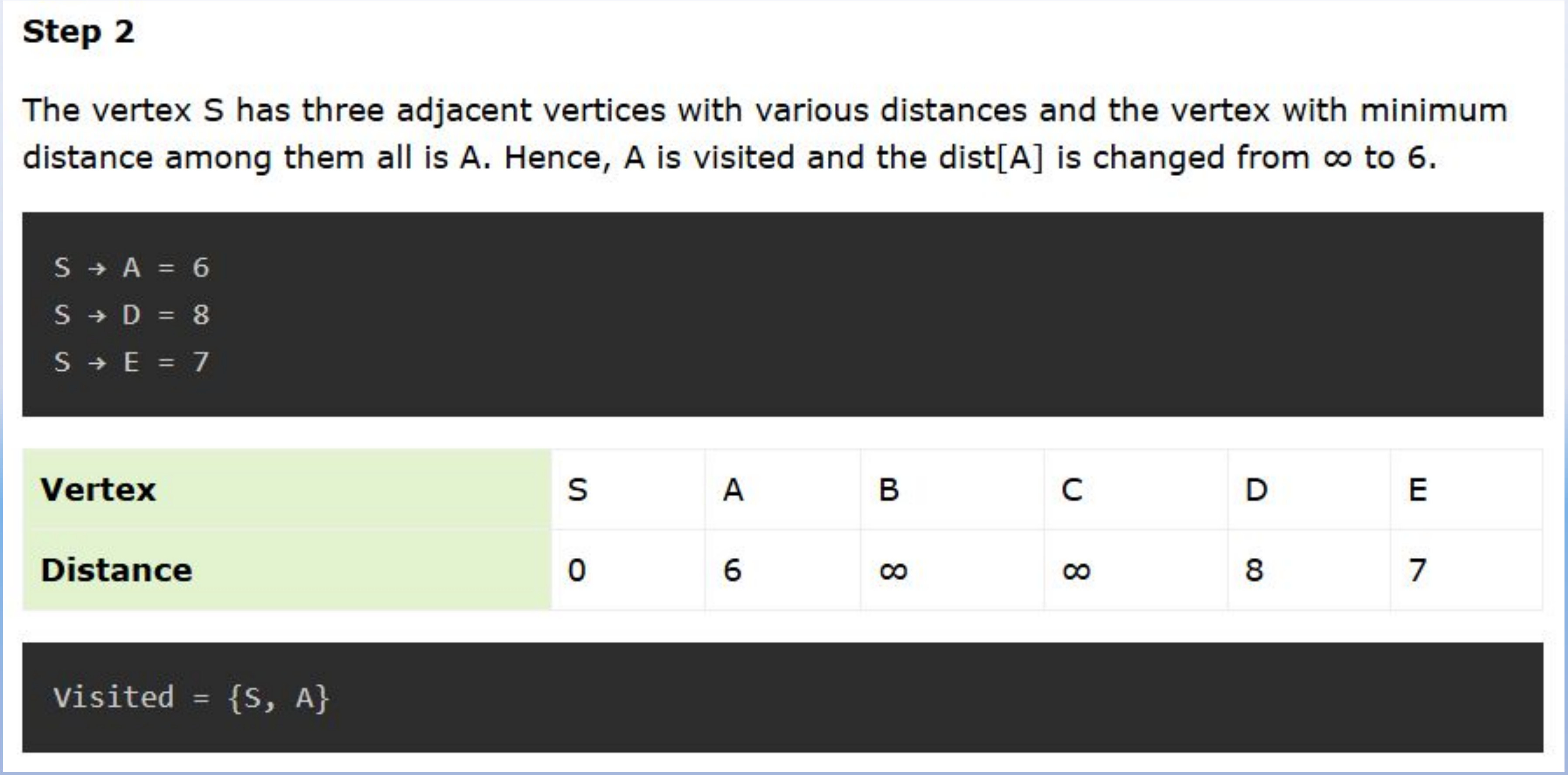
Step 1

Initialize the distances of all the vertices as ∞, except the source node S.

Vertex	S	A	B	C	D	E
Distance	0	∞	∞	∞	∞	∞

Now that the source vertex S is visited, add it into the visited array.

```
visited = {S}
```



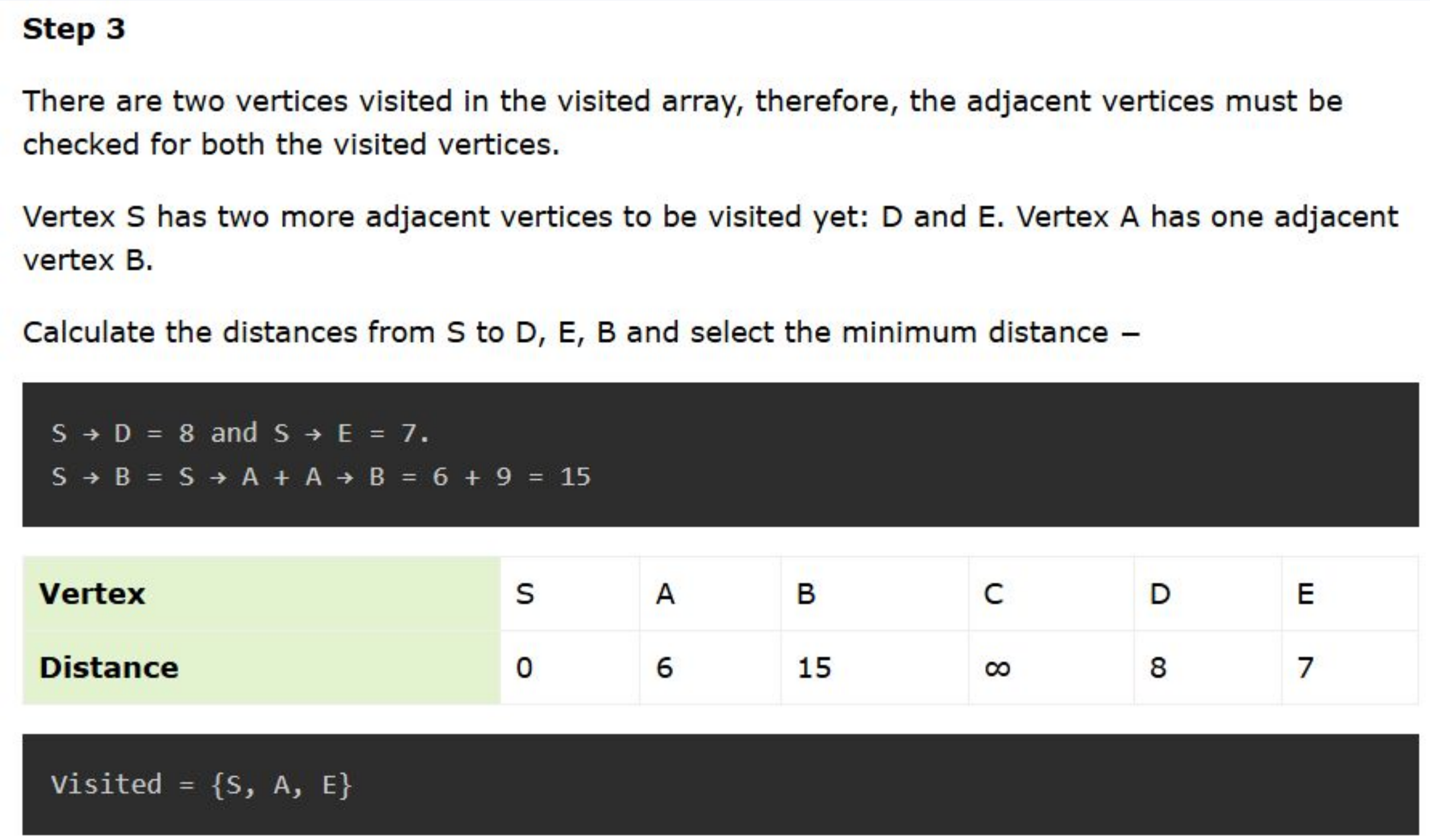
Step 2

The vertex S has three adjacent vertices with various distances and the vertex with minimum distance among them all is A. Hence, A is visited and the dist[A] is changed from ∞ to 6.

```
S → A = 6
S → D = 8
S → E = 7
```

Vertex	S	A	B	C	D	E
Distance	0	6	∞	∞	8	7

```
Visited = {S, A}
```



Step 3

There are two vertices visited in the visited array, therefore, the adjacent vertices must be checked for both the visited vertices.

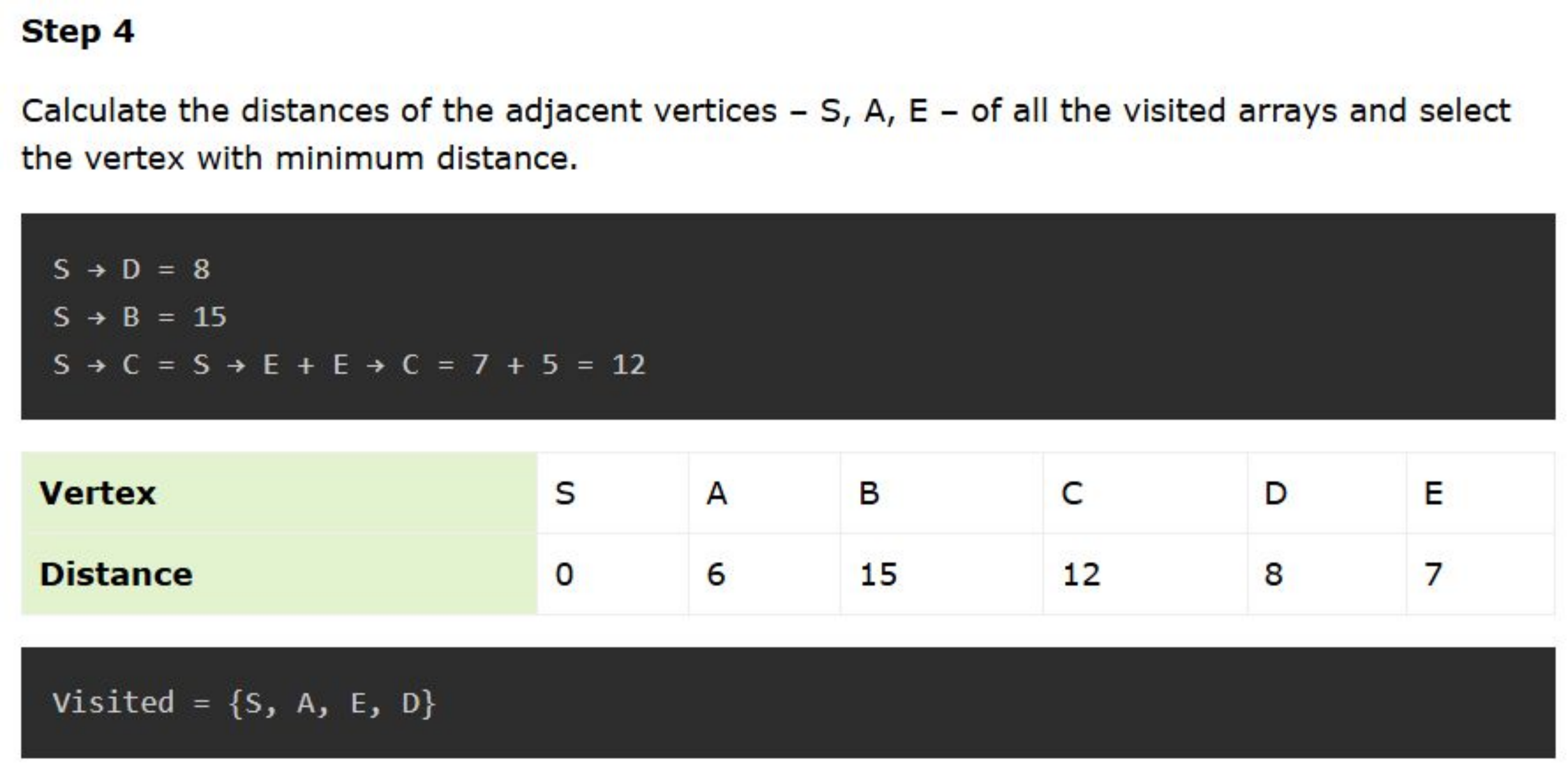
Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.

Calculate the distances from S to D, E, B and select the minimum distance –

```
S → D = 8 and S → E = 7.
S → B = S → A + A → B = 6 + 9 = 15
```

Vertex	S	A	B	C	D	E
Distance	0	6	15	∞	8	7

```
Visited = {S, A, E}
```



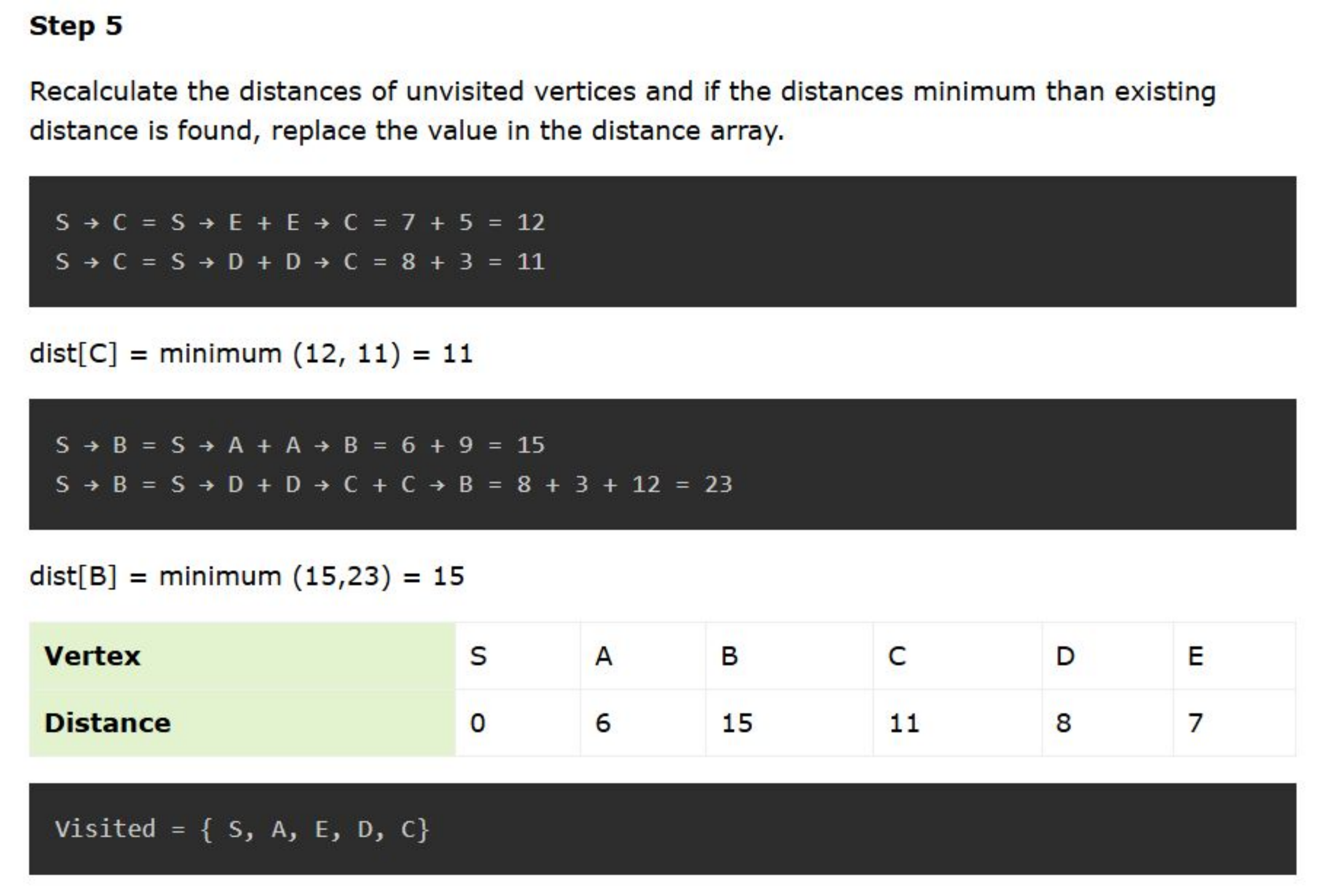
Step 4

Calculate the distances of the adjacent vertices – S, A, E – of all the visited arrays and select the vertex with minimum distance.

```
S → D = 8
S → B = 15
S → C = S → E + E → C = 7 + 5 = 12
```

Vertex	S	A	B	C	D	E
Distance	0	6	15	12	8	7

```
Visited = {S, A, E, D}
```



Step 5

Recalculate the distances of unvisited vertices and if the distances minimum than existing distance is found, replace the value in the distance array.

```
S → C = S → E + E → C = 7 + 5 = 12
S → C = S → D + D → C = 8 + 3 = 11
```

dist[C] = minimum (12, 11) = 11

```
S → B = S → A + A → B = 6 + 9 = 15
S → B = S → D + D → C + C → B = 8 + 3 + 12 = 23
```

dist[B] = minimum (15,23) = 15

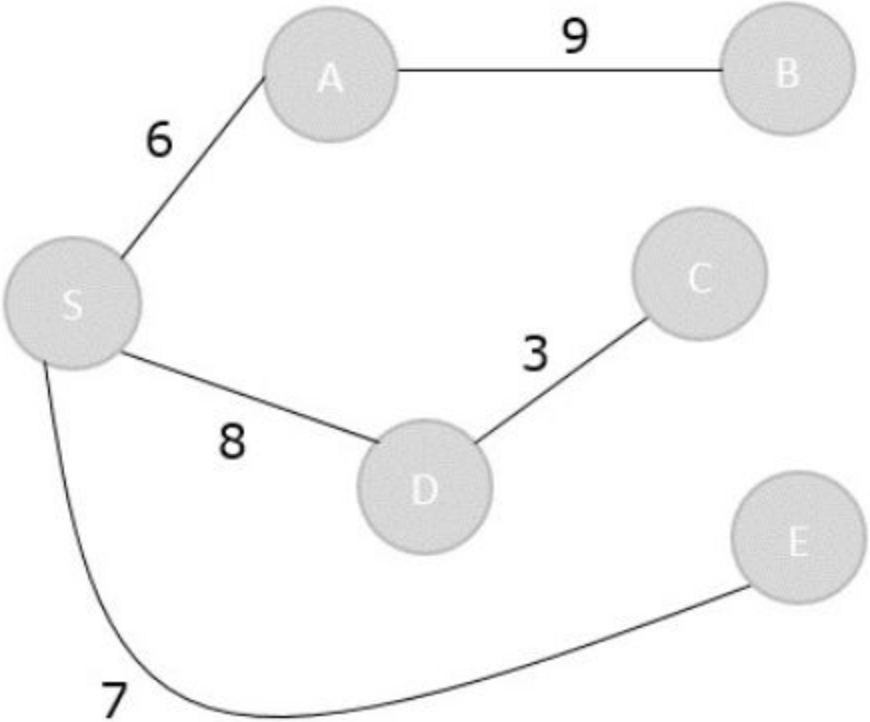
Vertex	S	A	B	C	D	E
Distance	0	6	15	11	8	7

```
Visited = { S, A, E, D, C}
```

Step 6

The remaining unvisited vertex in the graph is B with the minimum distance 15, is added to the output spanning tree.

```
Visited = {S, A, E, D, C, B}
```



The shortest path spanning tree is obtained as an output using the dijkstra's algorithm.

Floyd-Warshall Algorithm

- The Floyd-Warshall algorithm is a graph algorithm that is deployed to find the shortest path between all the vertices present in a weighted graph. This algorithm is different from other shortest path algorithms; to describe it simply, this algorithm uses each vertex in the graph as a pivot to check if it provides the shortest way to travel from one point to another.
- The Floyd-Warshall algorithm works on both directed and undirected weighted graphs unless these graphs do not contain any negative cycles in them. By negative cycles, it means that the sum of all the edges in the graph must not lead to a negative number.
- Since the algorithm deals with overlapping sub-problems – the path found by the vertices acting as pivots are stored for solving the next steps – it uses the dynamic programming approach.
- Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved using the Adjacency Matrix representation of graphs.

Floyd-Warshall Algorithm

Consider a graph, **G = {V, E}** where **V** is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, **G**, is represented in the form of an adjacency matrix, **A**, that contains all the weights of every edge connecting two vertices.

Algorithm

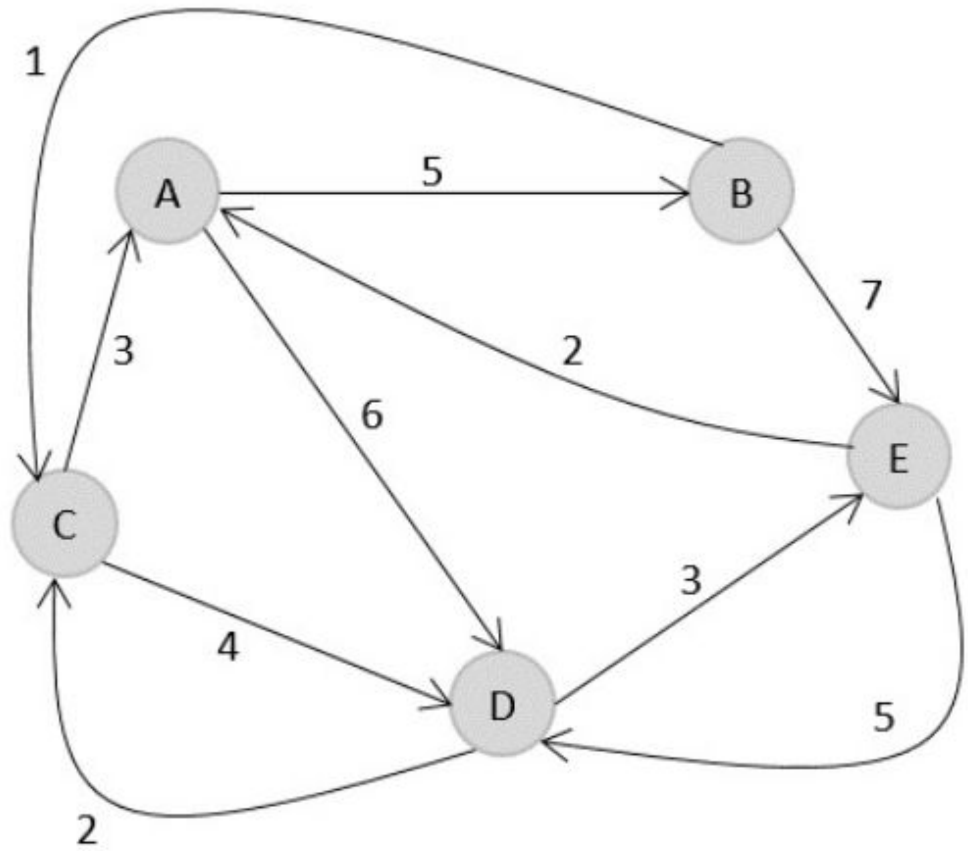
Step 1 − Construct an adjacency matrix **A** with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

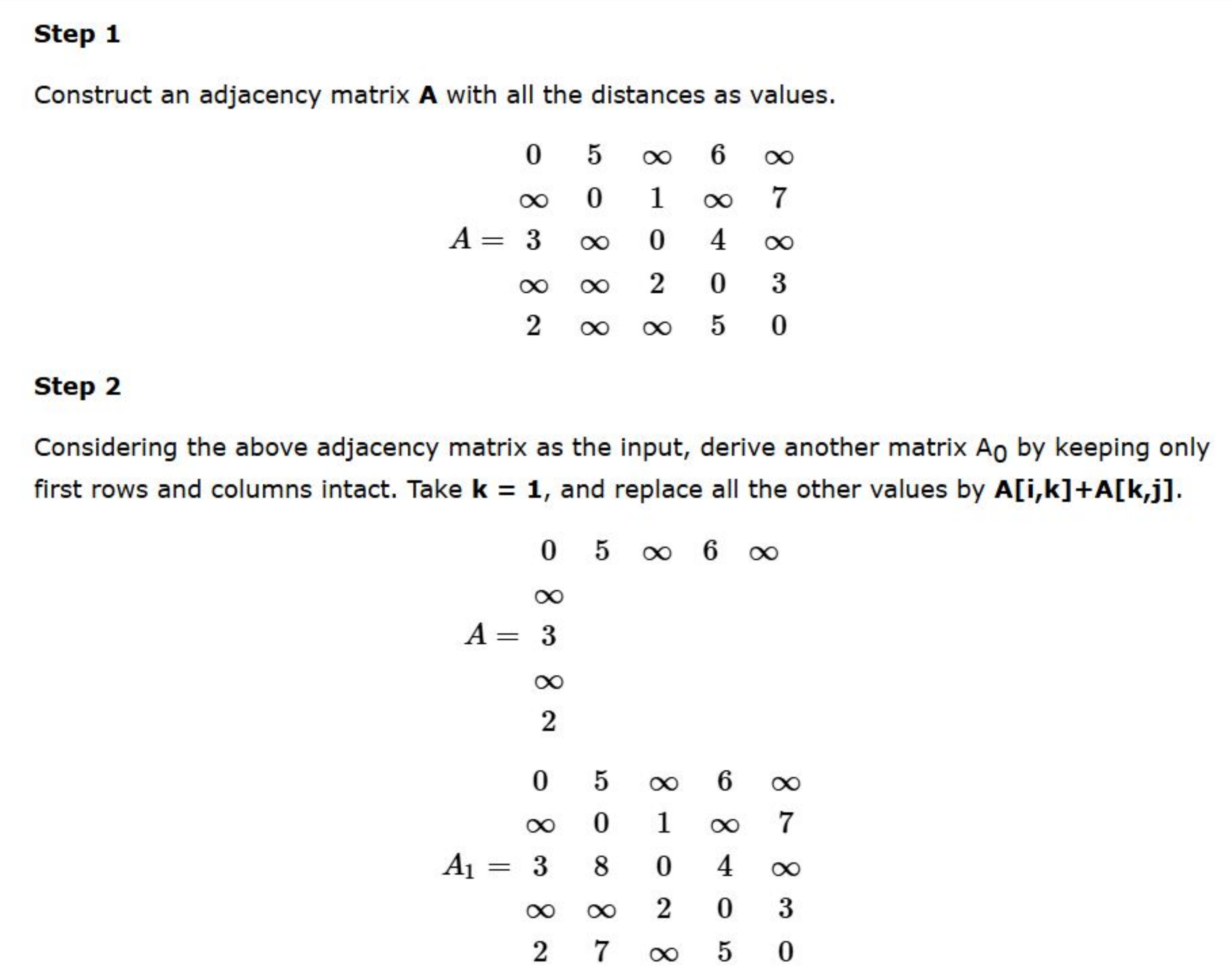
Step 2 − Derive another adjacency matrix $\mathbf{A_1}$ from **A** keeping the first row and first column of the original adjacency matrix intact in $\mathbf{A_1}$. And for the remaining values, say $\mathbf{A_1[i,j]}$, if **A[i,j]>A[i,k]+A[k,j]** then replace $\mathbf{A_1[i,j]}$ with **A[i,k]+A[k,j]**. Otherwise, do not change the values. Here, in this step, **k = 1** (first vertex acting as pivot).

Step 3 − Repeat **Step 2** for all the vertices in the graph by changing the **k** value for every pivot vertex until the final matrix is achieved.

Step 4 − The final adjacency matrix obtained is the final solution with all the shortest paths.

Consider the following directed weighted graph **G = {V, E}**. Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.





Step 1

Construct an adjacency matrix **A** with all the distances as values.

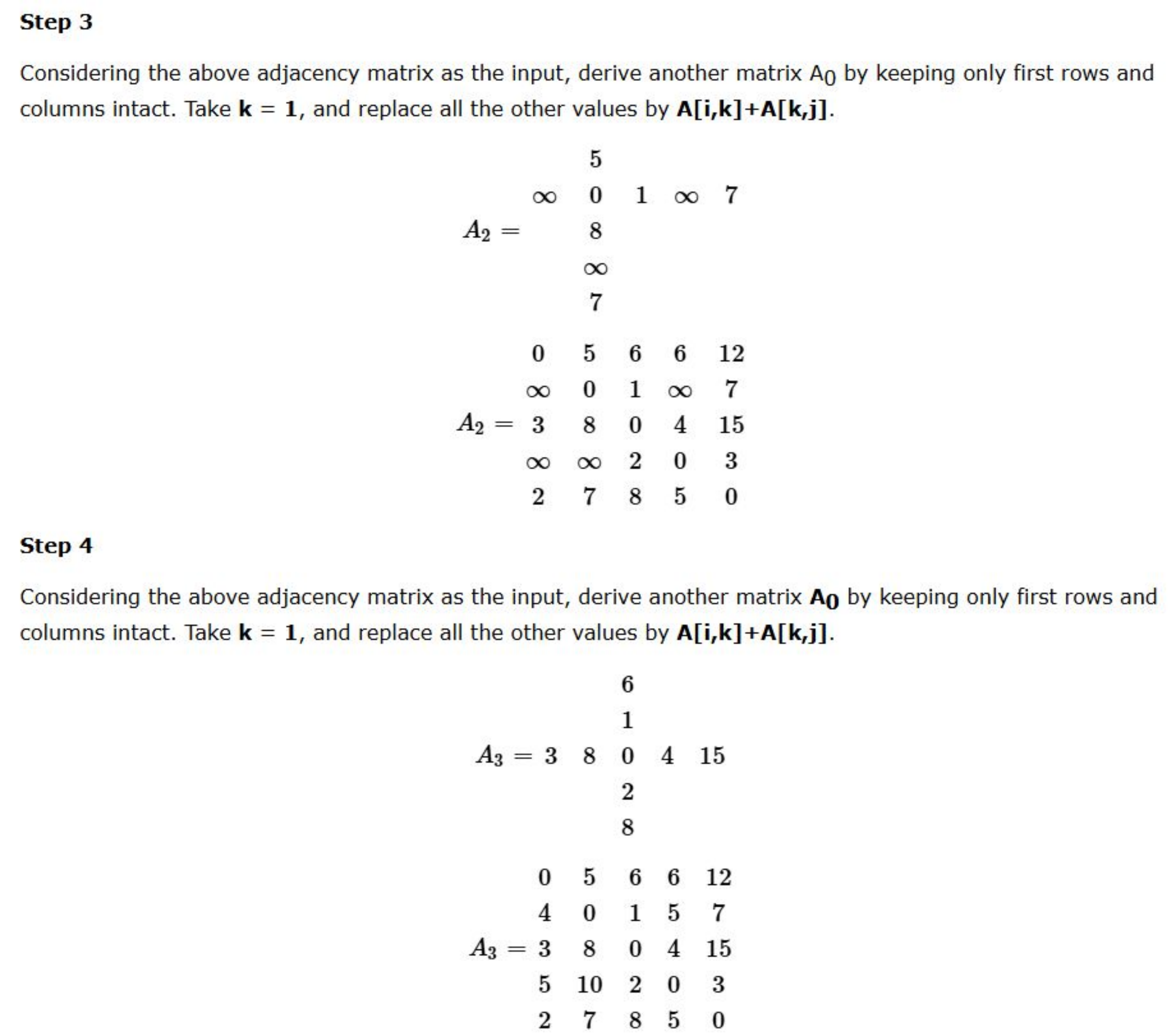
$$A = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & \infty & \infty & 5 & 0 \end{matrix}$$

Step 2

Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & & & & \\ 3 & & & & \\ \infty & & & & \\ 2 & & & & \end{matrix}$$

$$A_1 = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & \infty & 5 & 0 \end{matrix}$$



Step 3

Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k** = **1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_2 = \begin{array}{ccccc} & 5 & & & \\ \infty & 0 & 1 & \infty & 7 \\ & 8 & & & \\ & \infty & & & \\ & 7 & & & \end{array}$$

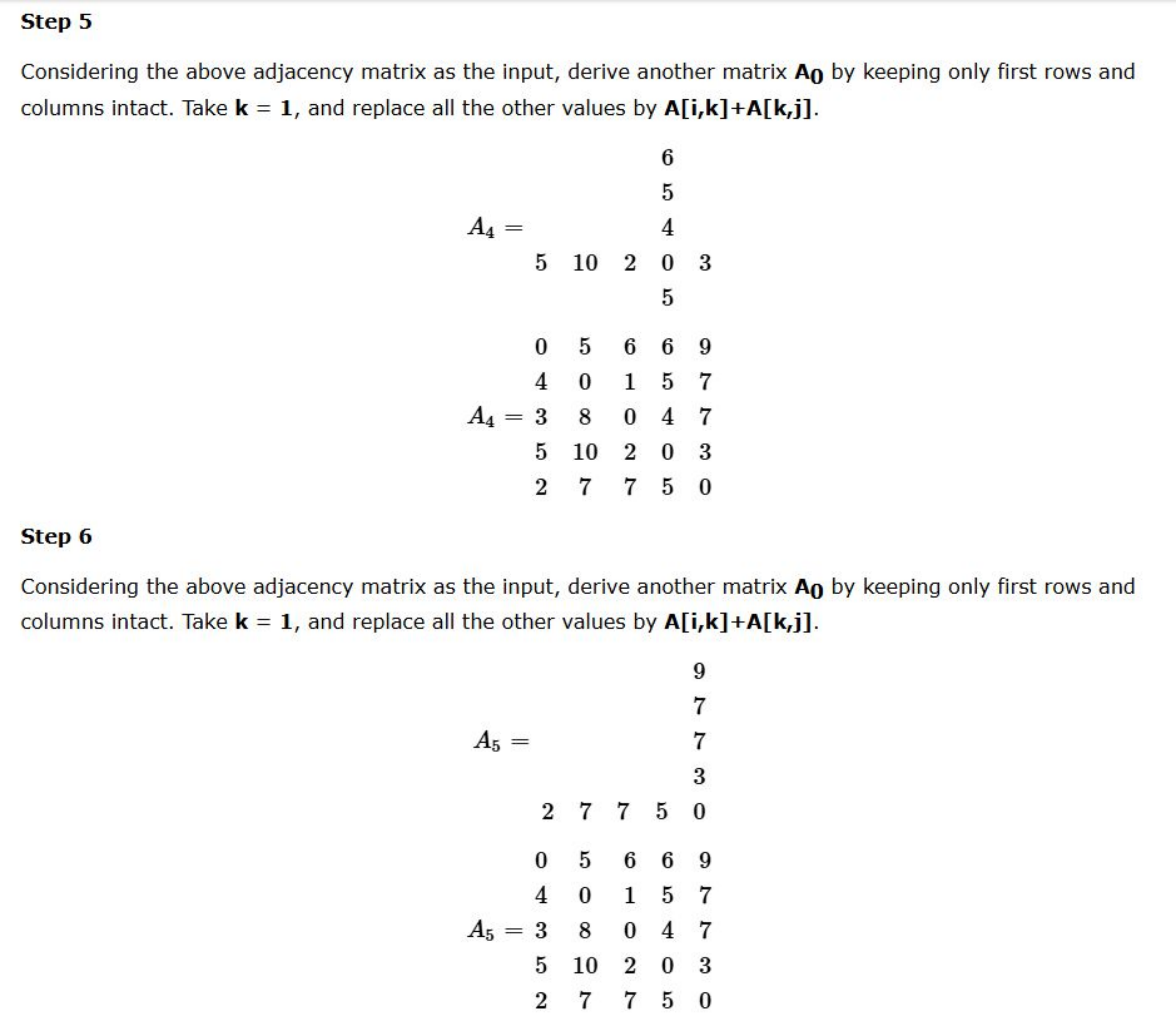
$$A_2 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 12 \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & 15 \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{array}$$

Step 4

Considering the above adjacency matrix as the input, derive another matrix **A_0** by keeping only first rows and columns intact. Take **k** = **1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_3 = \begin{array}{ccccc} & & 6 & & \\ & & 1 & & \\ 3 & 8 & 0 & 4 & 15 \\ & & 2 & & \\ & & 8 & & \end{array}$$

$$A_3 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 12 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 15 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{array}$$



Step 5

Considering the above adjacency matrix as the input, derive another matrix **A₀** by keeping only first rows and columns intact. Take **k** = **1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_4 = \begin{array}{ccccc} & & & 6 & \\ & & & 5 & \\ & & & 4 & \\ 5 & 10 & 2 & 0 & 3 \\ & & & 5 & \end{array}$$

$$A_4 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{array}$$

Step 6

Considering the above adjacency matrix as the input, derive another matrix **A₀** by keeping only first rows and columns intact. Take **k** = **1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_5 = \begin{array}{ccccc} & & & & 9 \\ & & & & 7 \\ & & & & 7 \\ & & & & 3 \\ 2 & 7 & 7 & 5 & 0 \end{array}$$

$$A_5 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{array}$$

Thank you !

ANY QUESTION?