Graph-Data Structure

Dr. Gopal Chandra Jana
Assistant Professor

What is a graph?

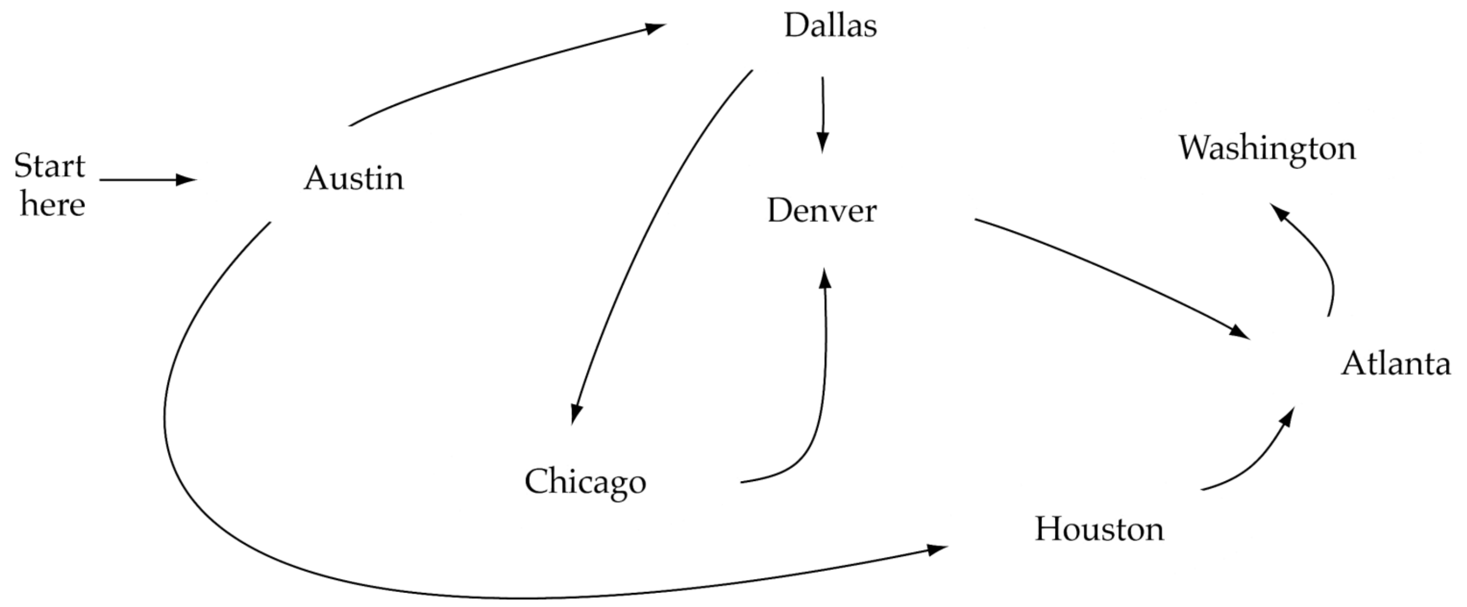
- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices

A graph *G* is defined as follows:

G=(V,E)

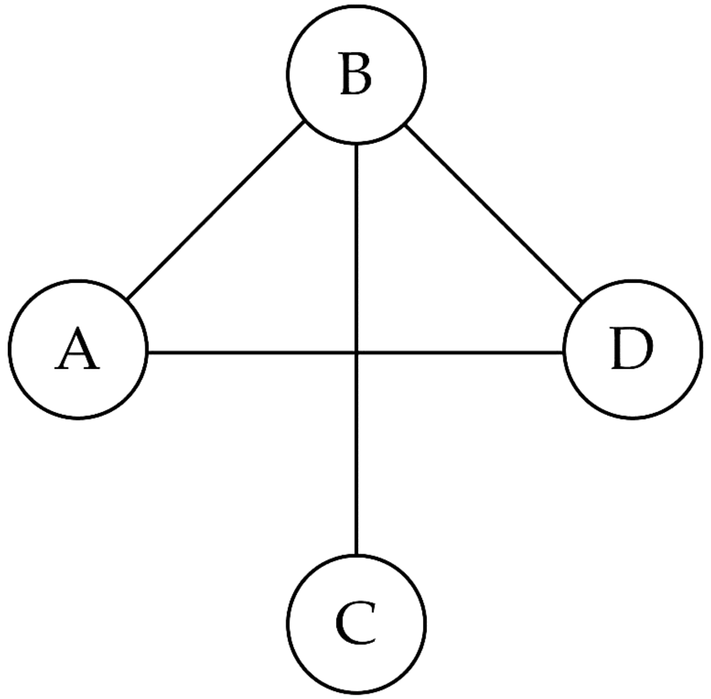
V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)



Directed vs. undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*

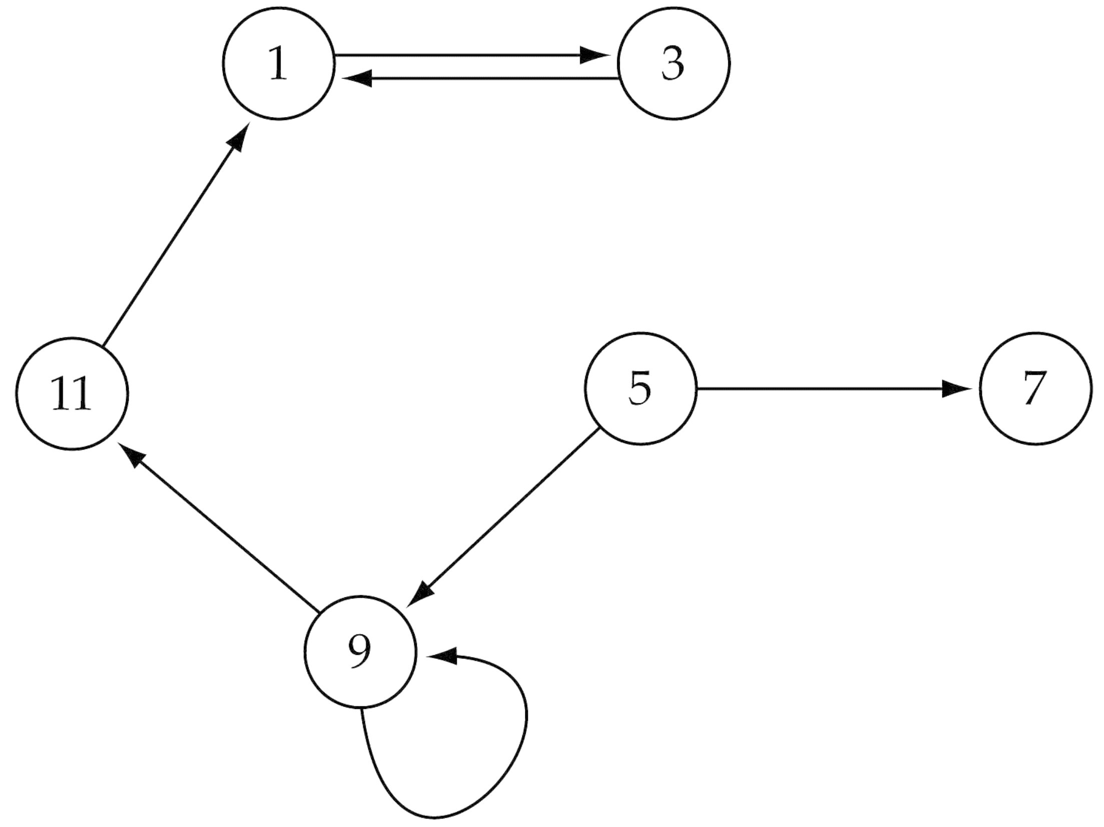


V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



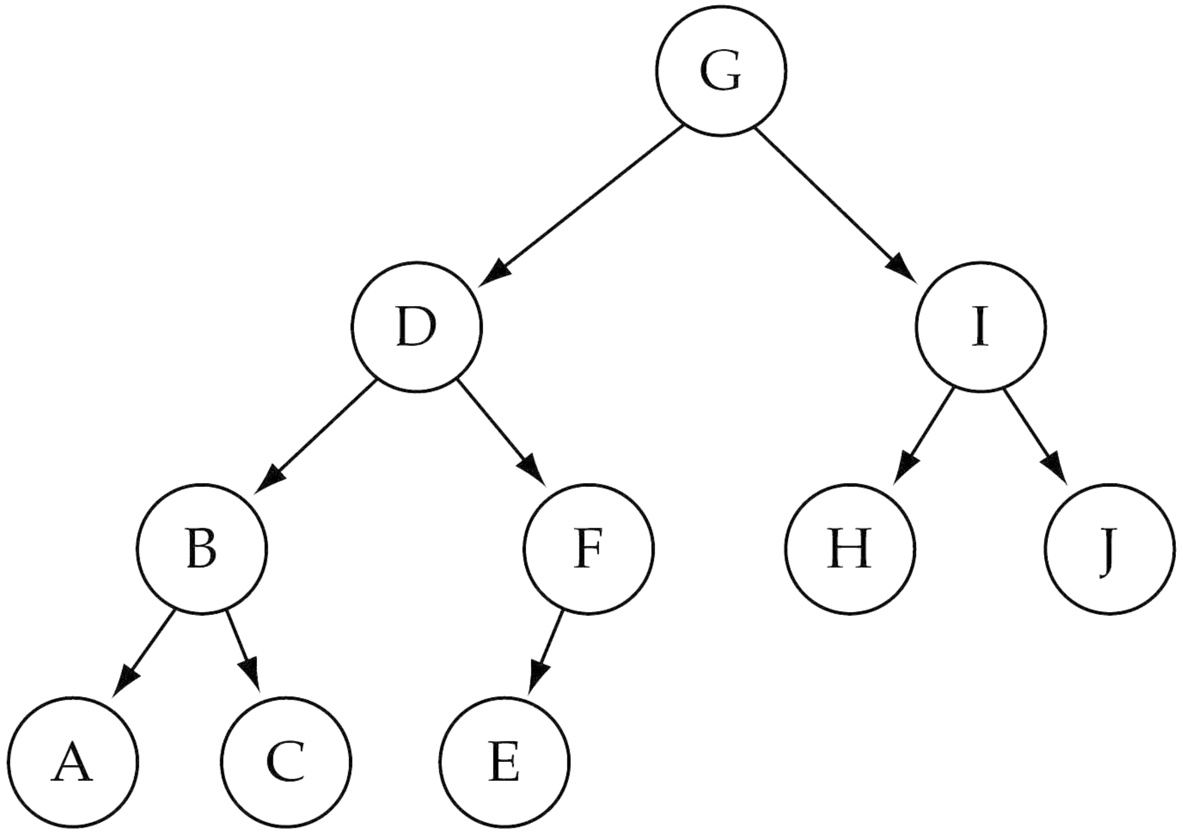
V(Graph2) = { 1, 3, 5, 7, 9, 11 }

E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7) (9,9), (11,1) 11), (9, 9), (11, 1) }

Trees vs graphs

- Trees are special cases of graphs!!

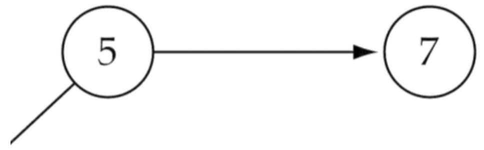
(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology

- <u>Adjacent nodes</u>: two nodes are adjacent if they are connected by an edge



5 is adjacent to 7
7 is adjacent from 5

- <u>Path</u>: a sequence of vertices that connect two nodes in a graph

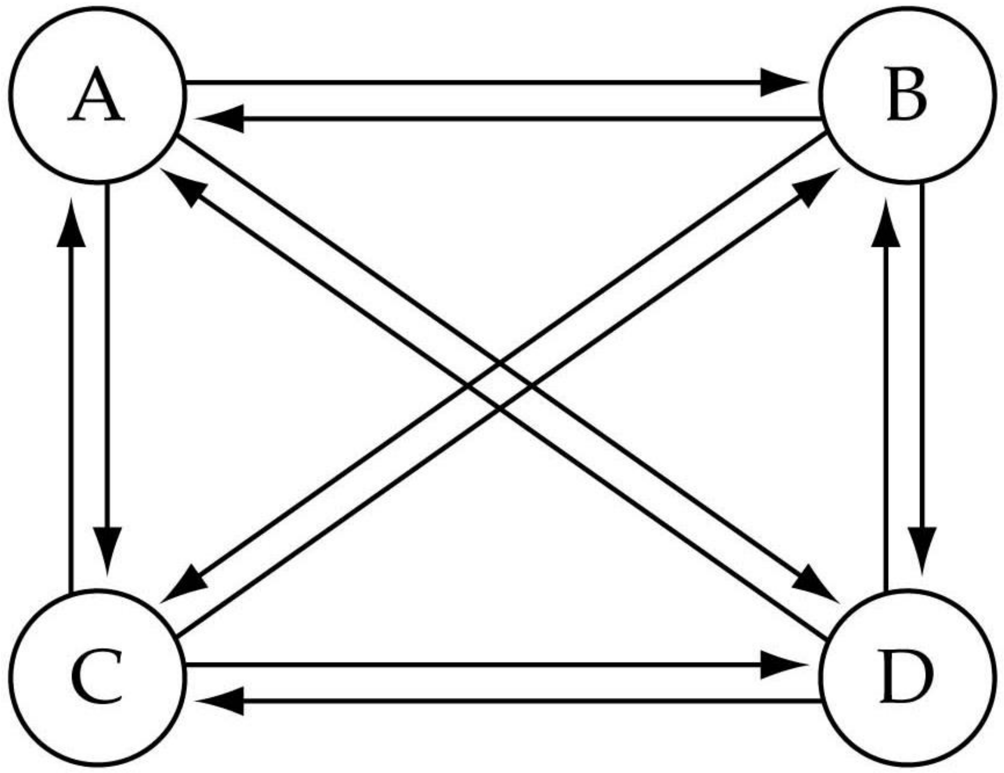
- <u>Complete graph</u>: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



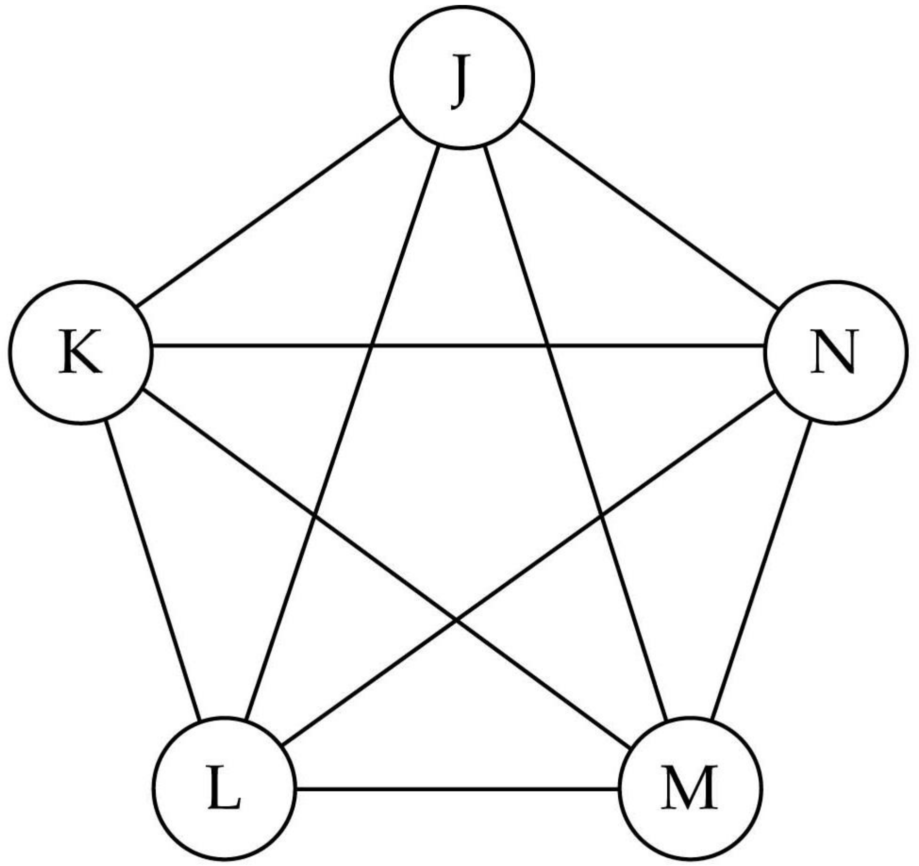
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

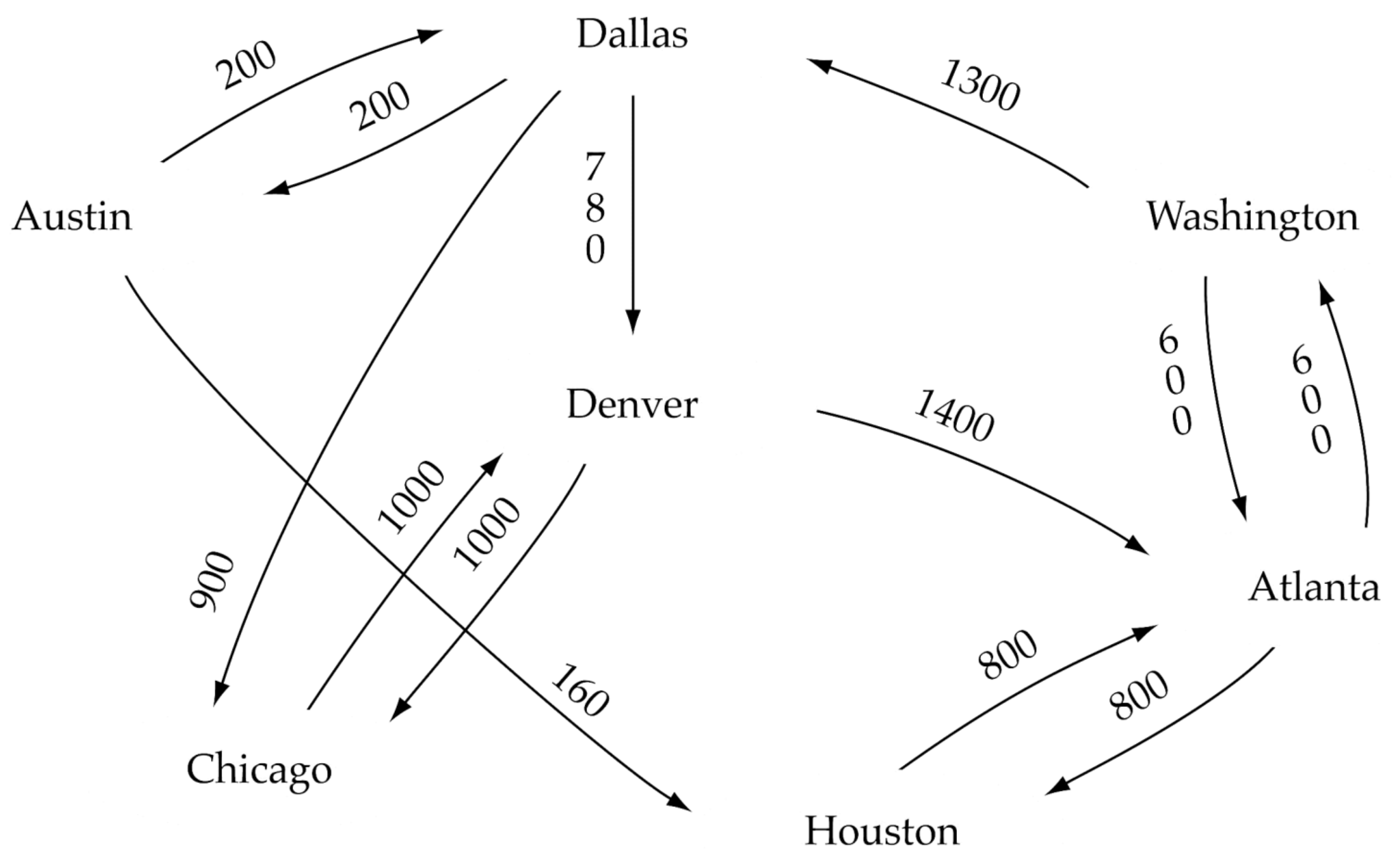
$O(N^2)$



(b) Complete undirected graph.

Graph terminology (cont.)

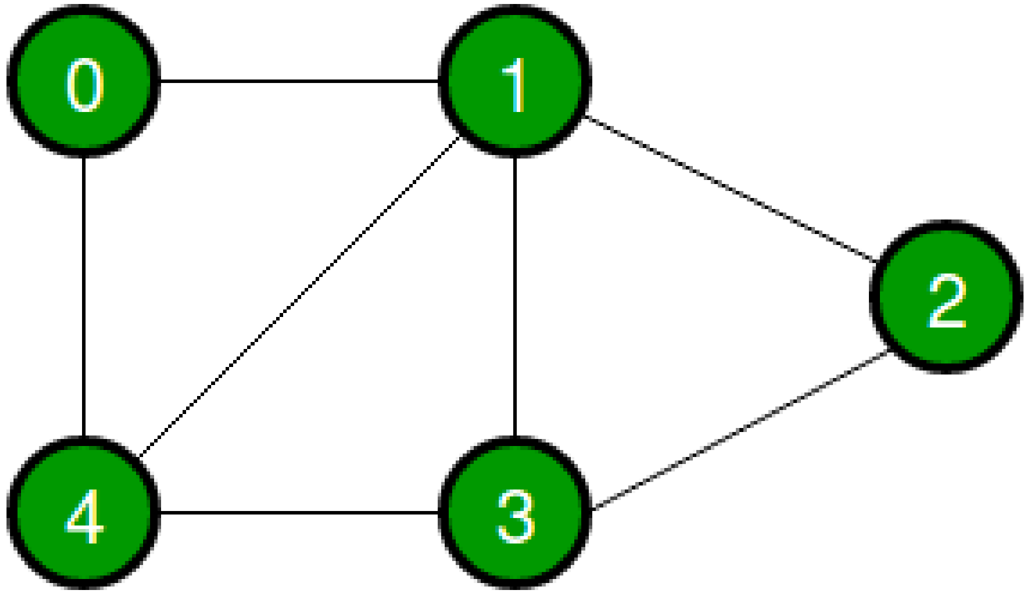
- Weighted graph: a graph in which each edge carries a value



Graph implementation

- <u>Array-based implementation (Adjacency Matrix)</u>
  - – A 1D array is used to represent the vertices
  - – A 2D array (adjacency matrix) is used to represent the edges

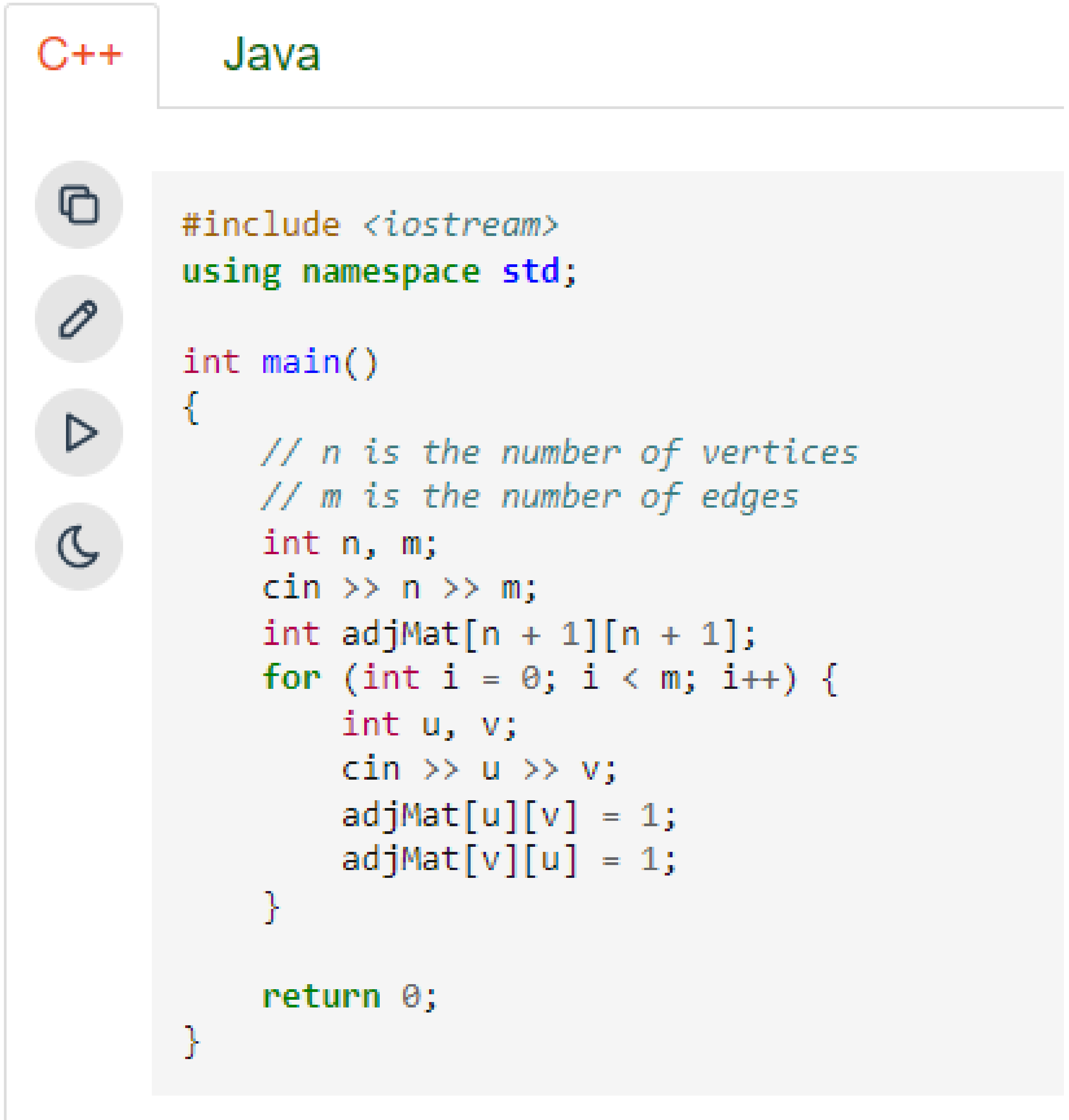
$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \end{array}$$



Pros*:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

Cons*:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

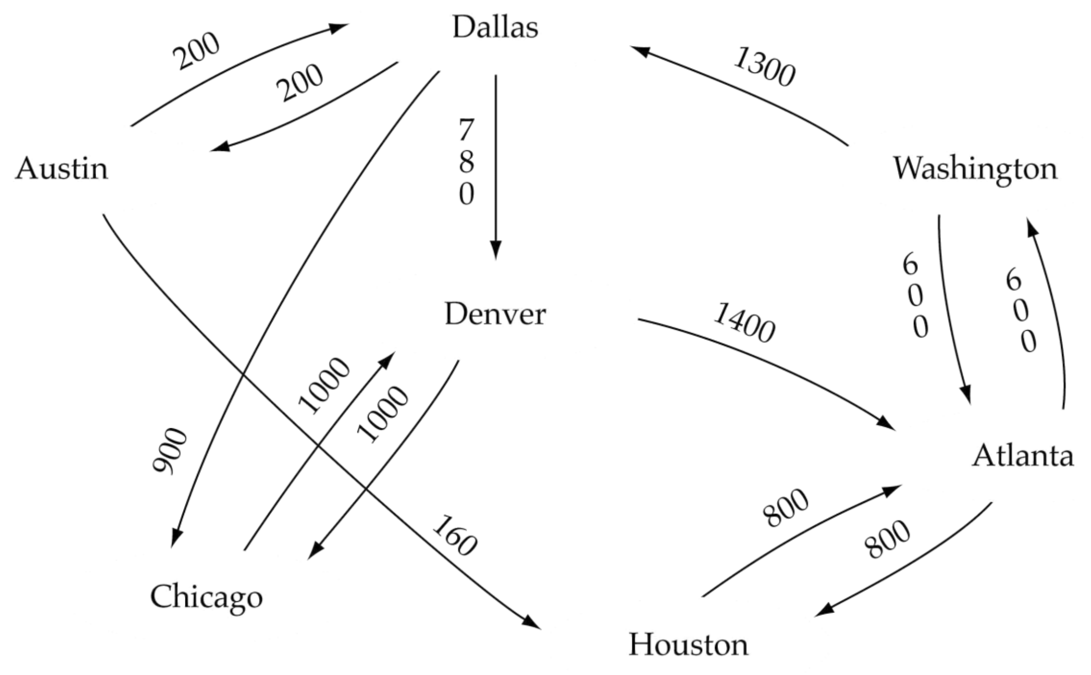
Implementation of taking input for adjacency matrix

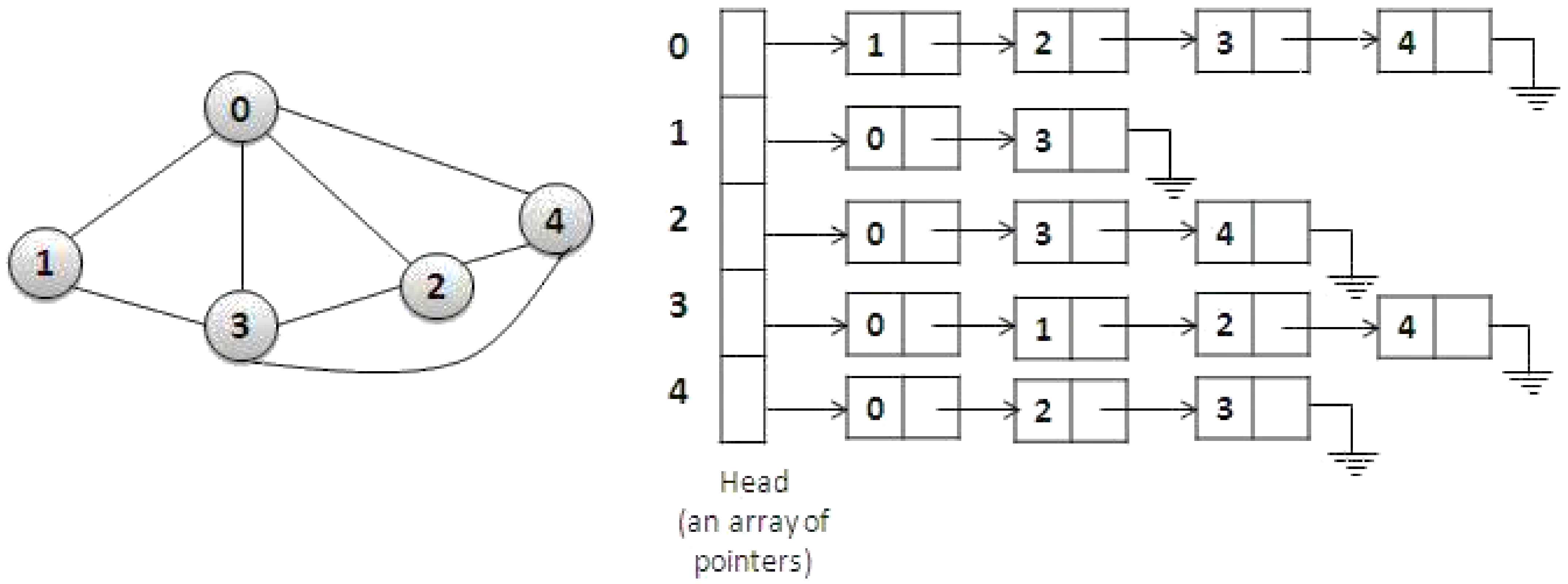


```
#include <iostream>
using namespace std;

int main()
{
    // n is the number of vertices
    // m is the number of edges
    int n, m;
    cin >> n >> m;
    int adjMat[n + 1][n + 1];
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adjMat[u][v] = 1;
        adjMat[v][u] = 1;
    }

    return 0;
}
```

Graph implementation (cont.)

- <u>Linked-list implementation (Adjency List)</u>
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)





Adjacency List Representation of Graph

Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs --$|E| \sim O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly

- **Adjacency list**
  - Good for sparse graphs -- $|E| \sim O(|V|)$
  - Memory requirements: $O(|V| + |E|) = O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of the linked list. The vector implementation has advantages of cache friendliness.

```cpp
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "\n Adjacency list of vertex "
             << v << "\n head ";
        for (auto x : adj[v])
           cout << "-> " << x;
        printf("\n");
    }
}
```

```cpp
// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
    return 0;
}
```

Output:

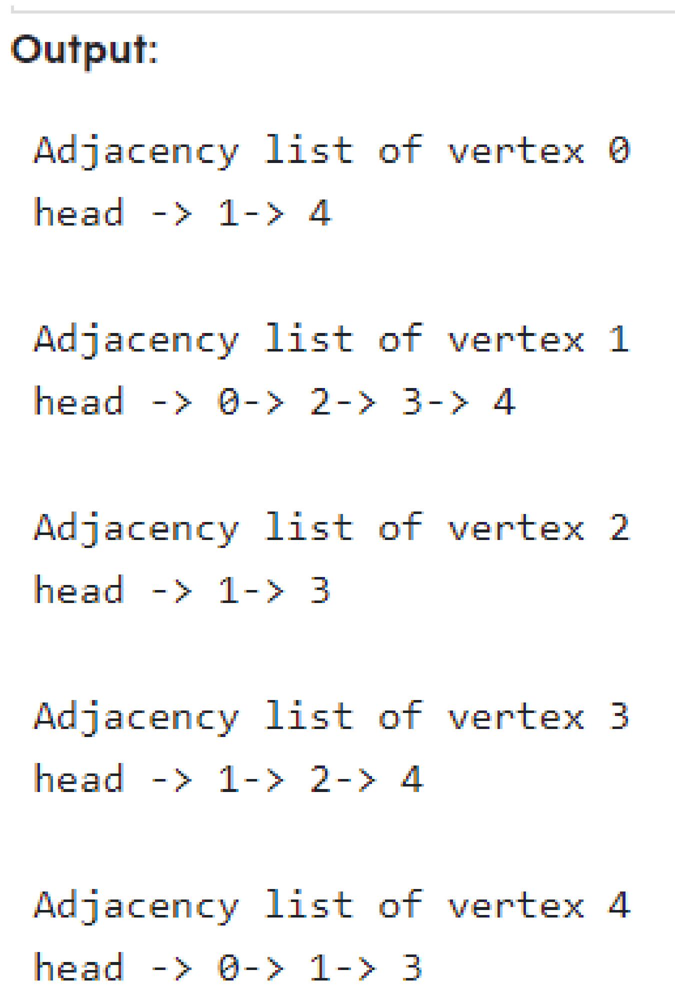
```
 Adjacency list of vertex 0
 head -> 1-> 4

 Adjacency list of vertex 1
 head -> 0-> 2-> 3-> 4

 Adjacency list of vertex 2
 head -> 1-> 3

 Adjacency list of vertex 3
 head -> 1-> 2-> 4

 Adjacency list of vertex 4
 head -> 0-> 1-> 3
```

***Pros*:** Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

***Cons*:** Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

```
// Java code to demonstrate Graph representation
//using ArrayList in Java
import java.util.*;

class Graph {

   // A utility function to add an edge in an undirected graph
   static void addEdge(ArrayList<ArrayList<Integer> > adj,
                  int u, int v)
   {
      adj.get(u).add(v);
      adj.get(v).add(u);
   }

   // A utility function to print the adjacency list
   // representation of graph
   static void printGraph(ArrayList<ArrayList<Integer> > adj)
   {
      for (int i = 0; i < adj.size(); i++) {
         System.out.println("\nAdjacency list of vertex" + i);
         for (int j = 0; j < adj.get(i).size(); j++) {
            System.out.print(" -> "+adj.get(i).get(j));
         }
         System.out.println();
      }
   }
```

```
// Driver Code
   public static void main(String[] args)
   {
      // Creating a graph with 5 vertices
      int V = 5;
      ArrayList<ArrayList<Integer> > adj
               = new ArrayList<ArrayList<Integer> >(V);

      for (int i = 0; i < V; i++)
         adj.add(new ArrayList<Integer>());

      // Adding edges one by one
      addEdge(adj, 0, 1);
      addEdge(adj, 0, 4);
      addEdge(adj, 1, 2);
      addEdge(adj, 1, 3);
      addEdge(adj, 1, 4);
      addEdge(adj, 2, 3);
      addEdge(adj, 3, 4);

      printGraph(adj);
   }
}
```



Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

Graph searching

- *Problem*: find a path between two nodes of the graph (e.g., Pune and Noida)

- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

During the execution of the nodes of a graph, a node can have one of three states i.e. Status of a Node as follows.

Status 1 (**White color):** The initial state of any node **(Ready state)**

Status 2 (**Gray color):** The node is in the queue or stack i.e. waiting status of the node. **(Waiting State)**

Status 3 (**Black color):** The node has been processed, removed from stack/queue. **(Processed state)**

BFS Algorithm:

1. Initialize all nodes to the ready state (White color).
2. Put the starting node in the Queue and change its status to the waiting state (Color Gray) and weight 0.
3. Repeat step 4 & step 5 until the Queue is empty.
4. Remove the front Node N of the queue. Process the removed node N and change its status to the processed state (Black color).
5. Add all the adjacent nodes of N that are in the ready state. Change their status to waiting for state and update their weight by (weight of N+1).
6. End.

DFS Algorithm:

1. Initialize all nodes to the ready state. i.e. color of all nodes to be white.
2. Push the starting node A into STACK and change its status to the waiting state i.e. Color Gray and update its forward time stamp to be 1.
3. Repeat step 4 and step 5 until STACK is empty.
4. POP the top Node N of STACK. Process N and change its status to the processed state i.e. Black color.
5. PUSH all the adjacent nodes of N onto the stack if they are in the ready state (white). Change their status to the waiting state (gray). If no new visit i.e. (no node having white color) Backtrack is required go to step 4.
6. End.

Breadth First Traversal of a Graph

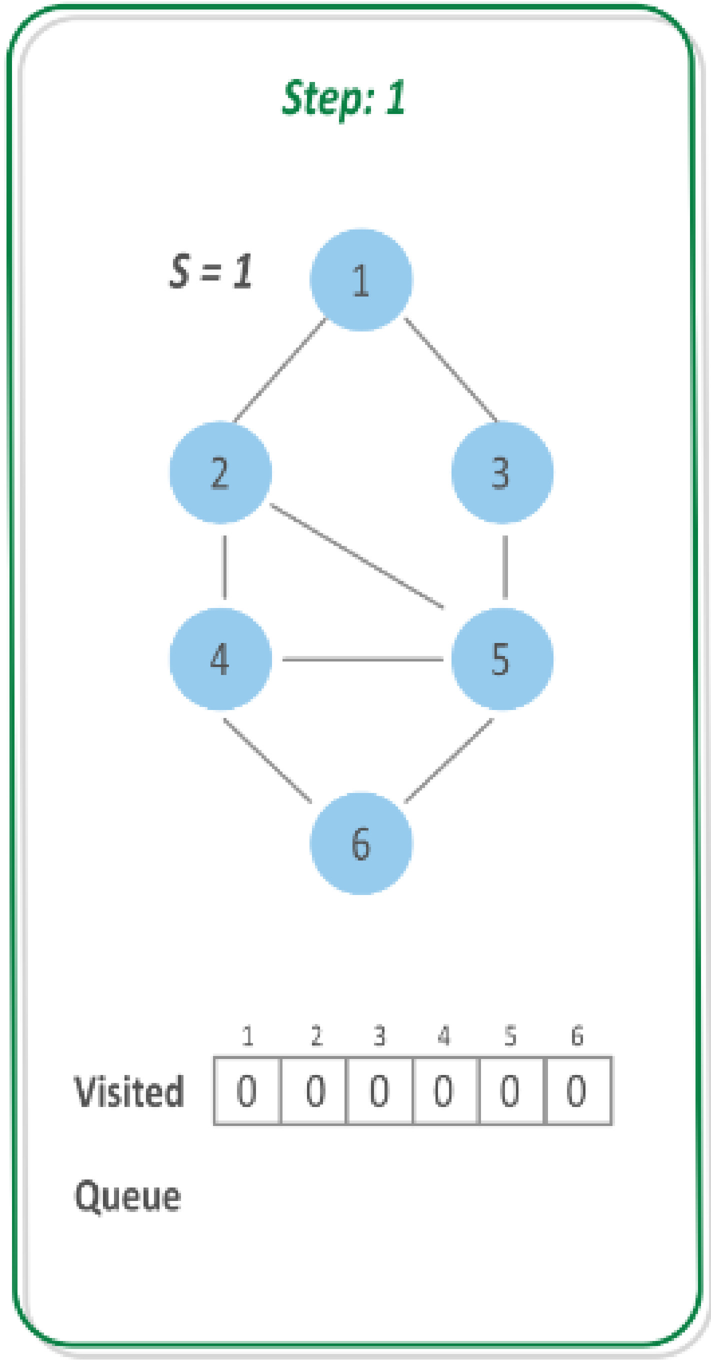
The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.

The BFS traversal of Graphs also traverses the graph in levels. It starts the traversal with a given vertex, visits all of the vertices adjacent to the initially given vertex and pushes them all to a queue in order of visiting. Then it pops an element from the front of the queue, visits all of its neighbours and pushes the neighbours which are not already visited into the queue and repeats the process until the queue is empty or all of the vertices are visited.

The BFS traversal uses an auxiliary boolean array say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

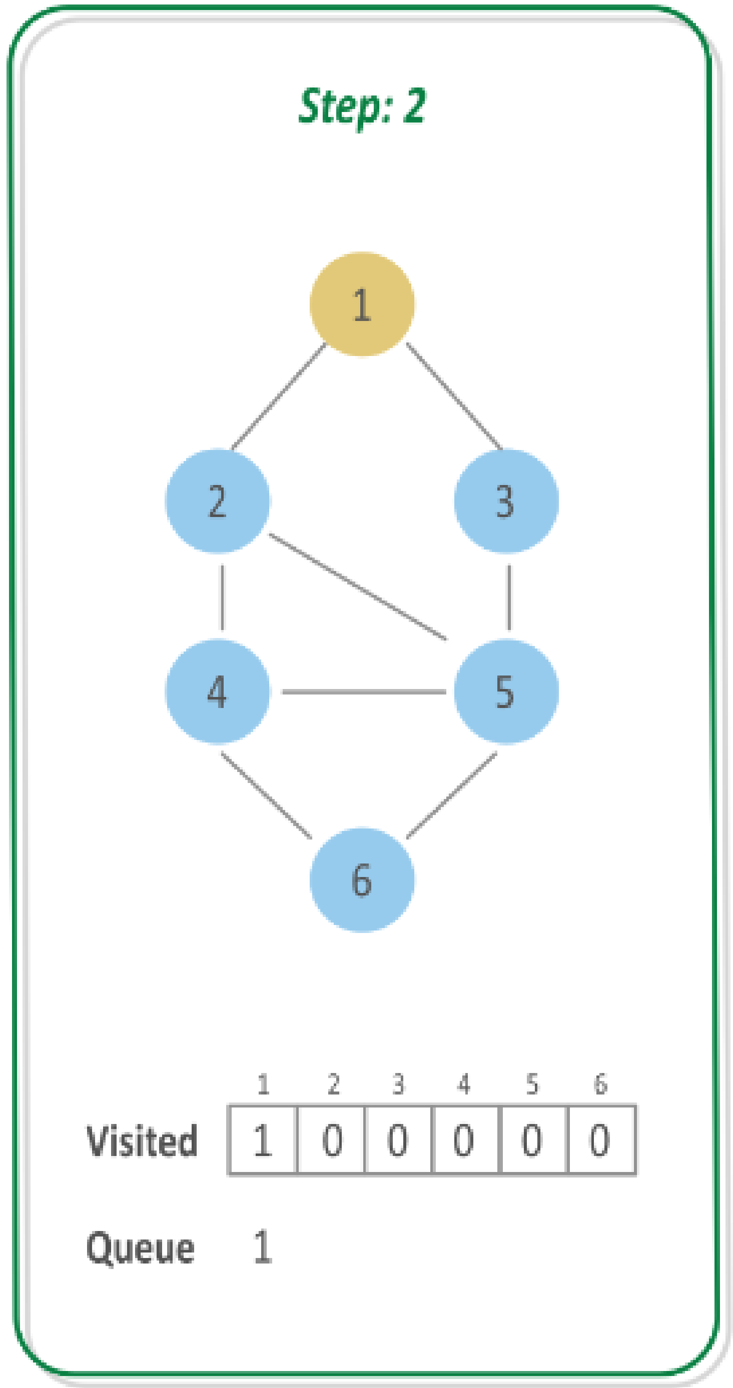
Complete Algorithm:

- Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
- Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
- Until the Queue is non-empty, repeat the below steps:
    - Pop an element from the queue and print the popped element.
    - Traverse all of the vertices adjacent to the vertex poped from the queue.
    - If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.
-



	1	2	3	4	5	6
Visited	0	0	0	0	0	0

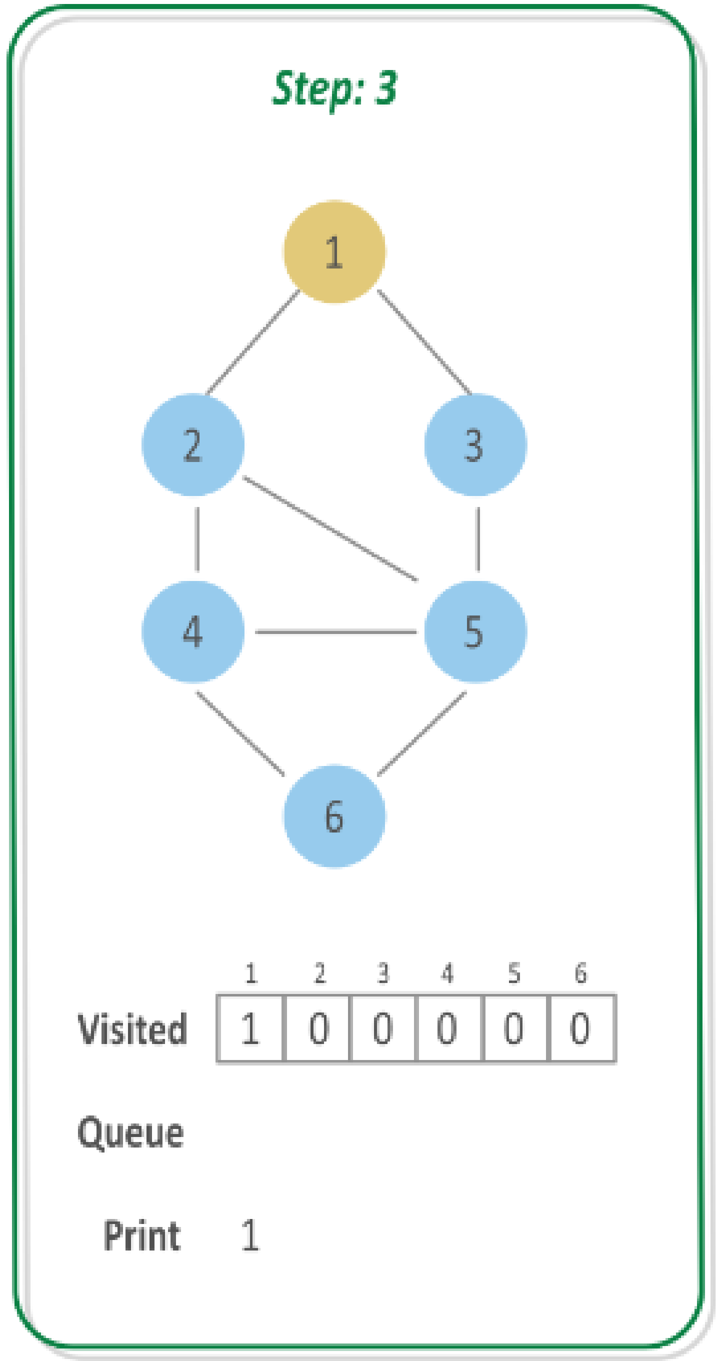
Queue



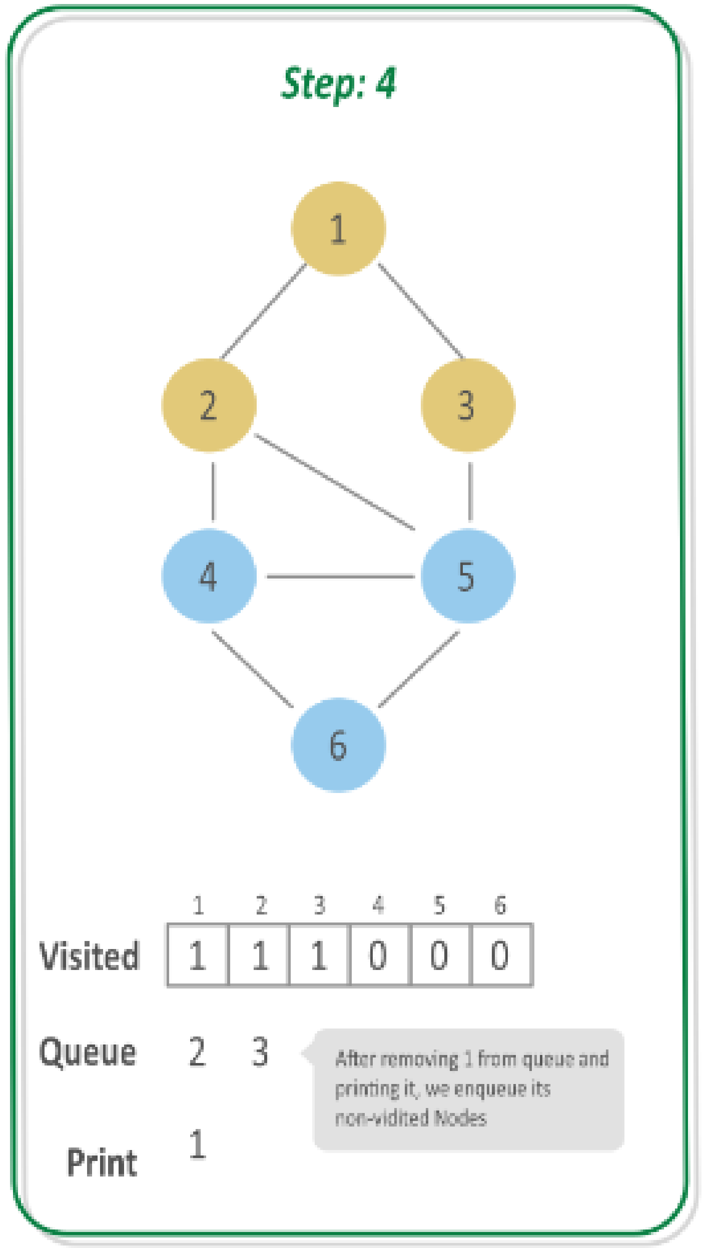
	1	2	3	4	5	6
Visited	1	0	0	0	0	0

Queue 1

Step 3: POP the vertex at the front of queue that is 1, and print it.



Step 4: Check if adjacent vertices of the vertex 1 are not already visited. If not, mark them visited and push them back to the queue.

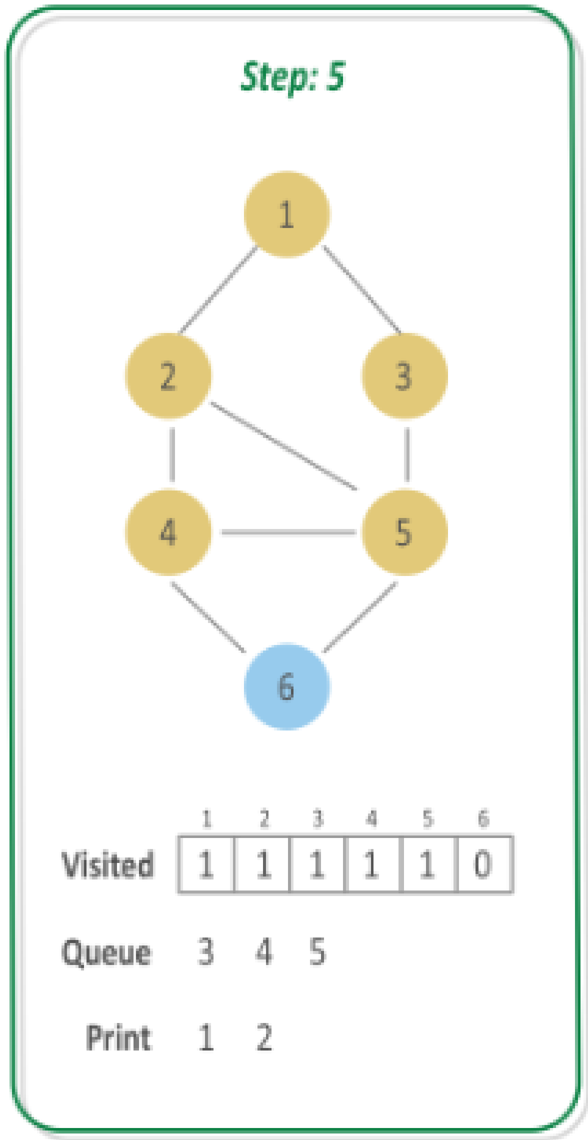

Step: 4
1
2
3
4
5
6
Visited
1 2 3 4 5 6
1 1 1 0 0 0
Queue 2 3
After removing 1 from queue and printing it, we enqueue its non-vidited Nodes
Print 1

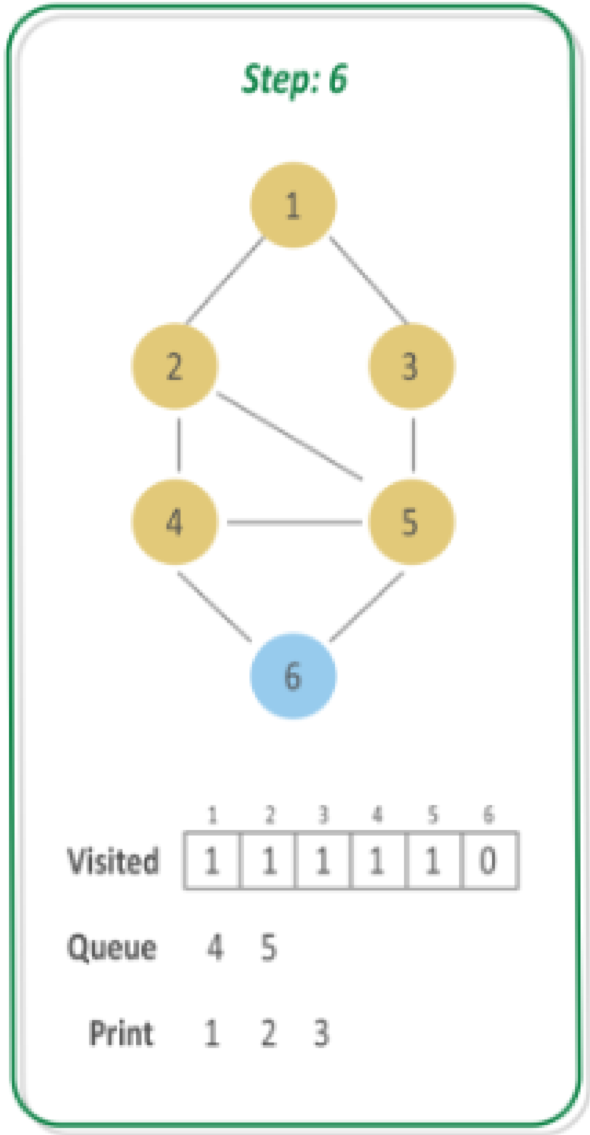
Step 5:

- POP the vertex at front that is 2 and print it.
- Check if the adjacent vertices of 2 are not already visited. If not, mark them visited and push them to queue. So, push 4 and 5.

Step 6:

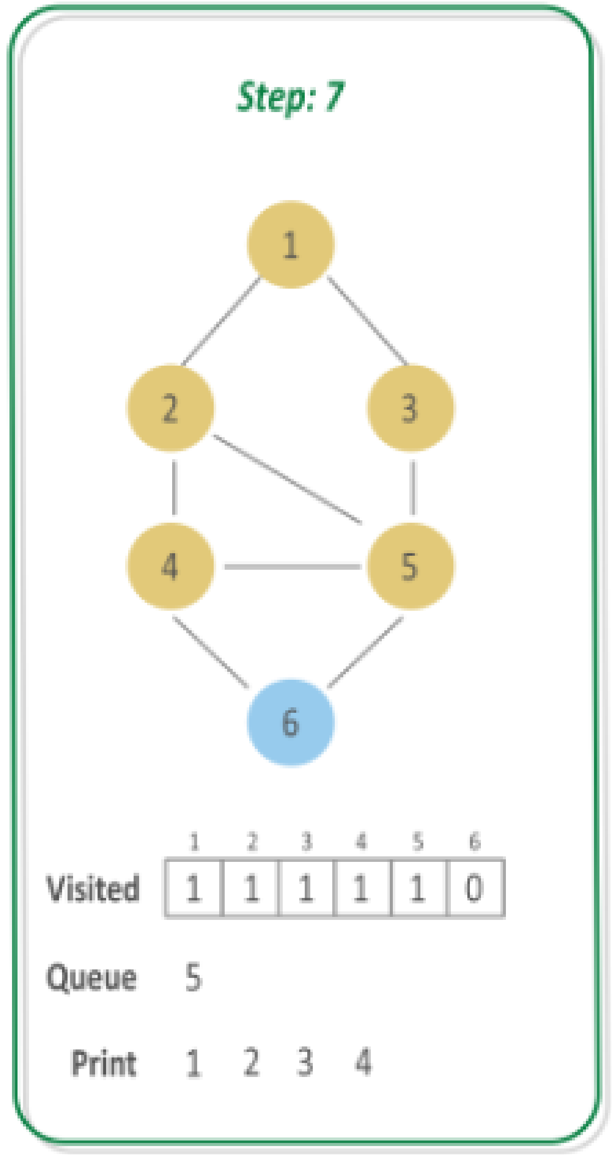
- POP the vertex at front that is 3 and print it.
- Check if the adjacent vertices of 3 are not already visited. If not, mark them visited and push them to queue. So, donot push anything.





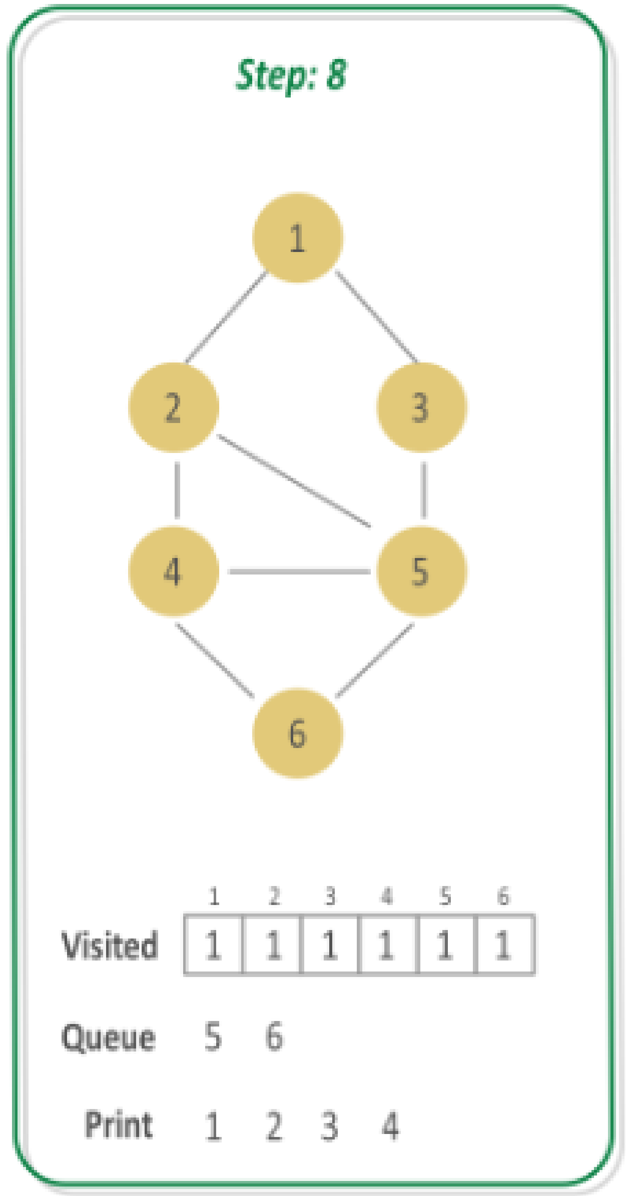
Step 7:

- POP the vertex at front that is 4 and print it.



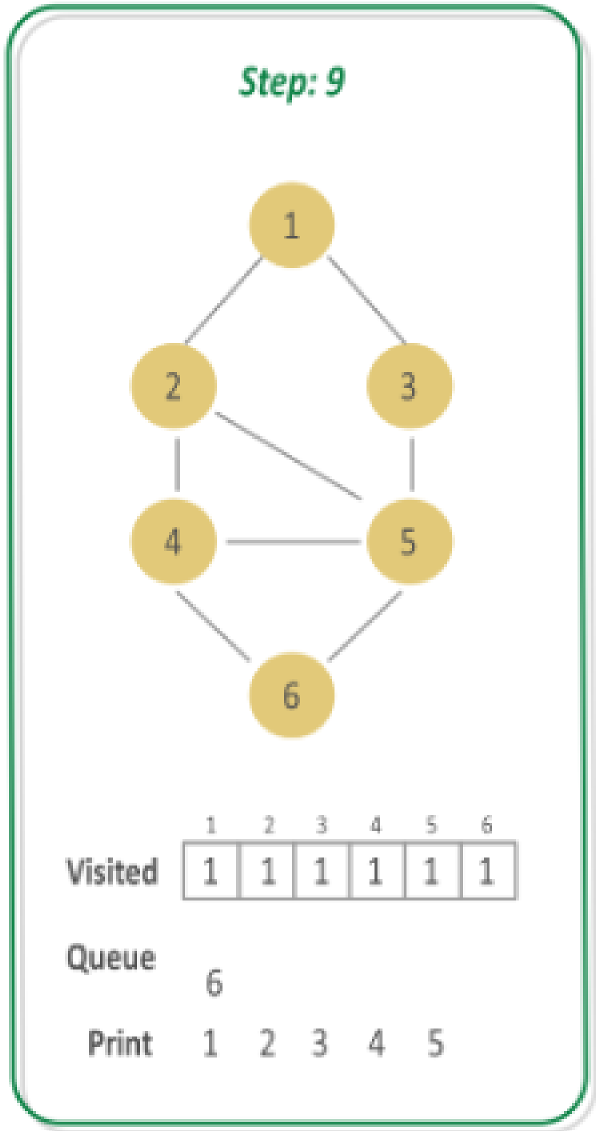
Step 8:

- Check if the adjacent vertices of 4 are not already visited. If not, mark them visited and push them to queue. So, push 6 to the queue.



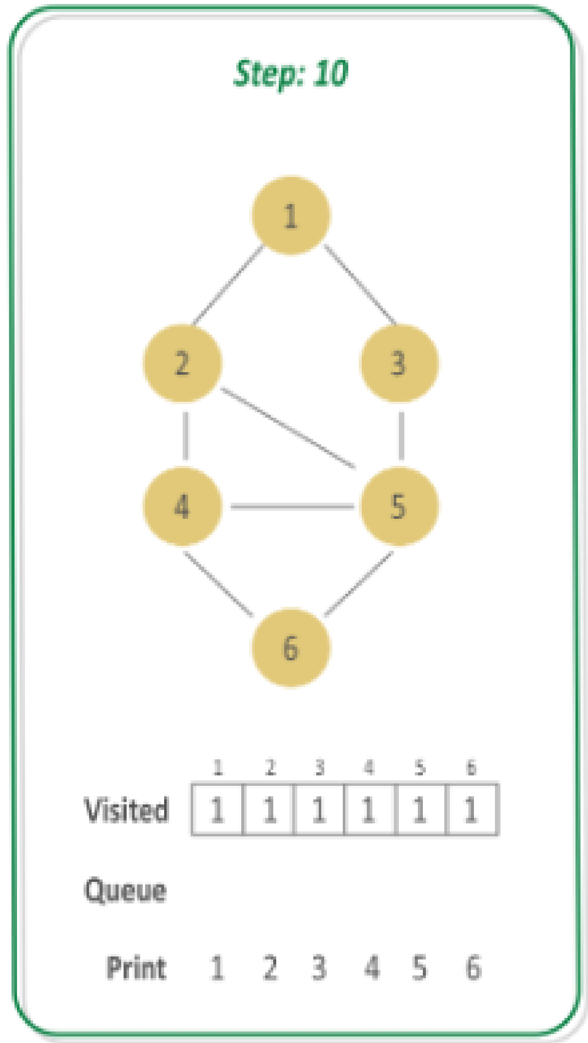
Step 9:

- POP the vertex at front, that is 5 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.



Step 10:

- POP the vertex at front, that is 6 and print it.
- Since, all of its adjacent vertices are already visited, donot push anything.



Since the Queue is empty now, it means that the complete graph is traversed.

```cpp
// C++ program to implement BFS traversal
// of a Graph

#include <bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Function to perform BFS traversal of the given Graph
void BFS(vector<int> adj[], int V)
{
    // Initialize a boolean array
    // to keep track of visited vertices
    bool visited[V + 1];

    // Mark all vertices not-visited initially
    for (int i = 1; i <= V; i++)
        visited[i] = false;

    // Create a Queue to perform BFS
    queue<int> q;

    // Our source vertex is vertex
    // numbered 1
    int s = 1;

    // Mark S visited and Push to queue
    visited[s] = true;
    q.push(s);
```

```cpp
    while (!q.empty()) {
        // Pop element at front and print
        int node = q.front();
        q.pop();

        cout << node << " ";

        // Traverse the nodes adjacent to the currently
        // poped element and push those elements to the
        // queue which are not already visited
        for (int i = 0; i < adj[node].size(); i++) {
            if (visited[adj[node][i]] == false) {
                // Mark it visited
                visited[adj[node][i]] = true;

                // Push it to the Queue
                q.push(adj[node][i]);
            }
        }
    }
}

// Driver code
int main()
{
    int V = 6;
    vector<int> adj[V + 1];
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 2, 4);
    addEdge(adj, 2, 5);
    addEdge(adj, 3, 5);
    addEdge(adj, 4, 5);
    addEdge(adj, 4, 6);
    addEdge(adj, 5, 6);

    BFS(adj, V);

    return 0;
}
```

Applications of BFS

We have earlier discussed Breadth First Traversal Algorithm for Graphs. We have also discussed Applications of Depth First Traversal. In this article, applications of Breadth First Search are discussed.

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Peer to Peer Networks.** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.
4. **Social Networking Websites:** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection:** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph:** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use BFS to detect cycle in a directed graph also,
9. **Ford–Fulkerson algorithm** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.
10. **To test if a graph is Bipartite** We can either use Breadth First or Depth First Traversal.
11. **Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

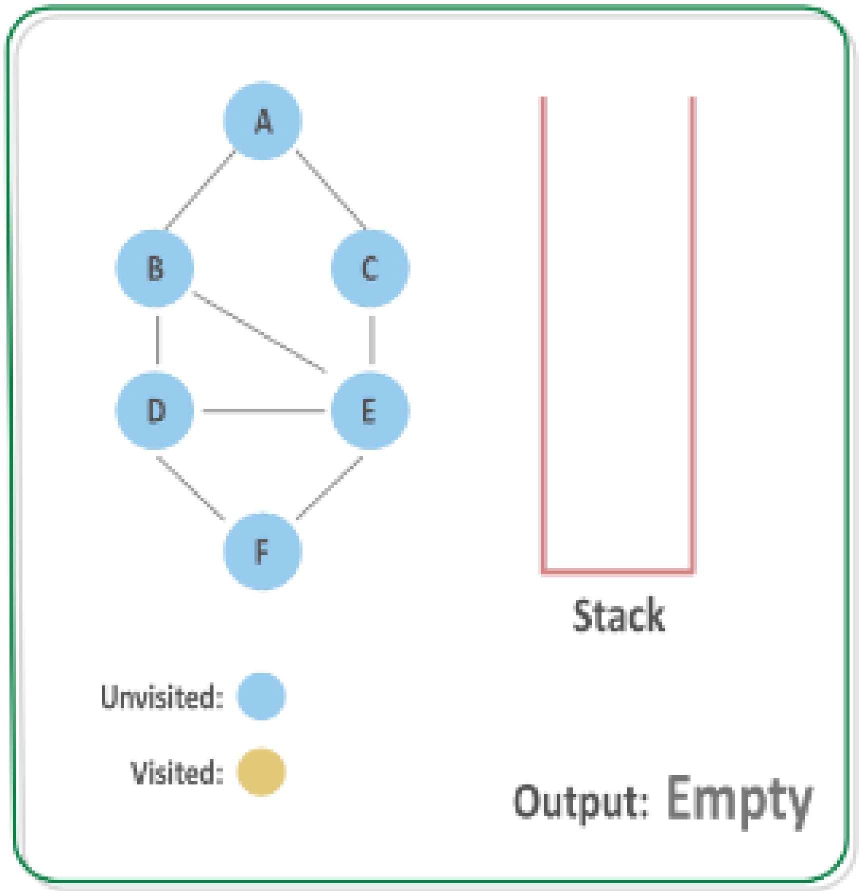
Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structure similar to Breadth First Search.
There can be many more applications as Breadth First Search is one of the core algorithms for Graphs.

Depth First Traversal of a Graph

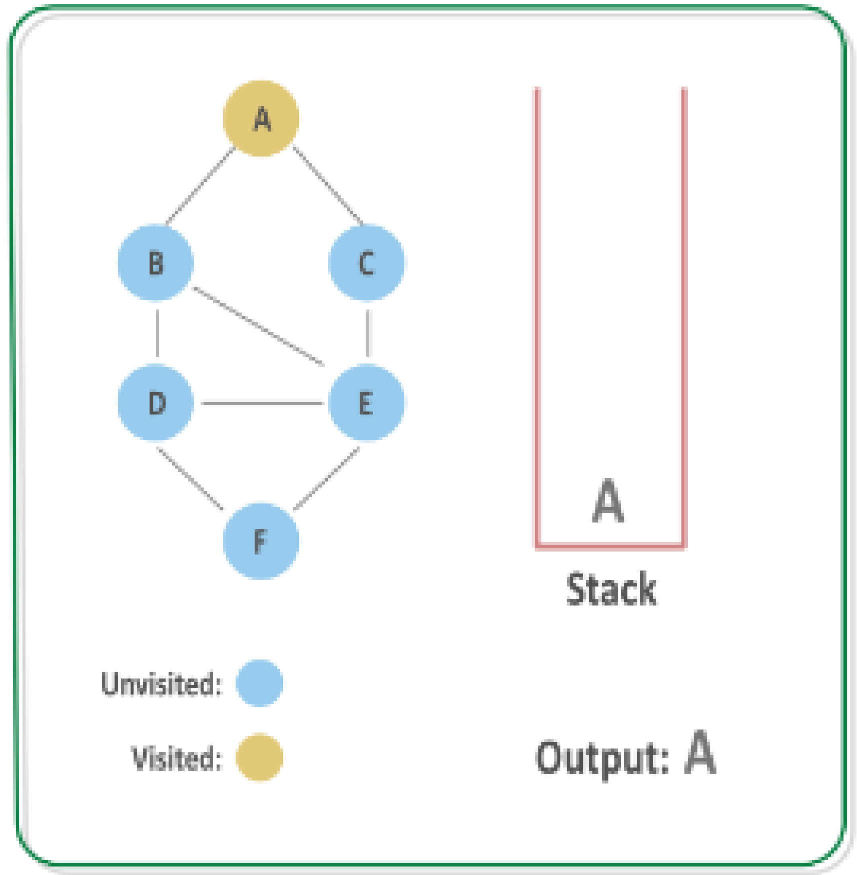
The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, it in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.

We can implement the DFS traversal algorithm using a recursive approach. While performing the DFS traversal the graph may contain a cycle and the same node can be visited again, so in order to avoid this we can keep track of visited array using an auxiliary array. On each step of the recursion mark, the current vertex visited and call the recursive function again for all the adjacent vertices.

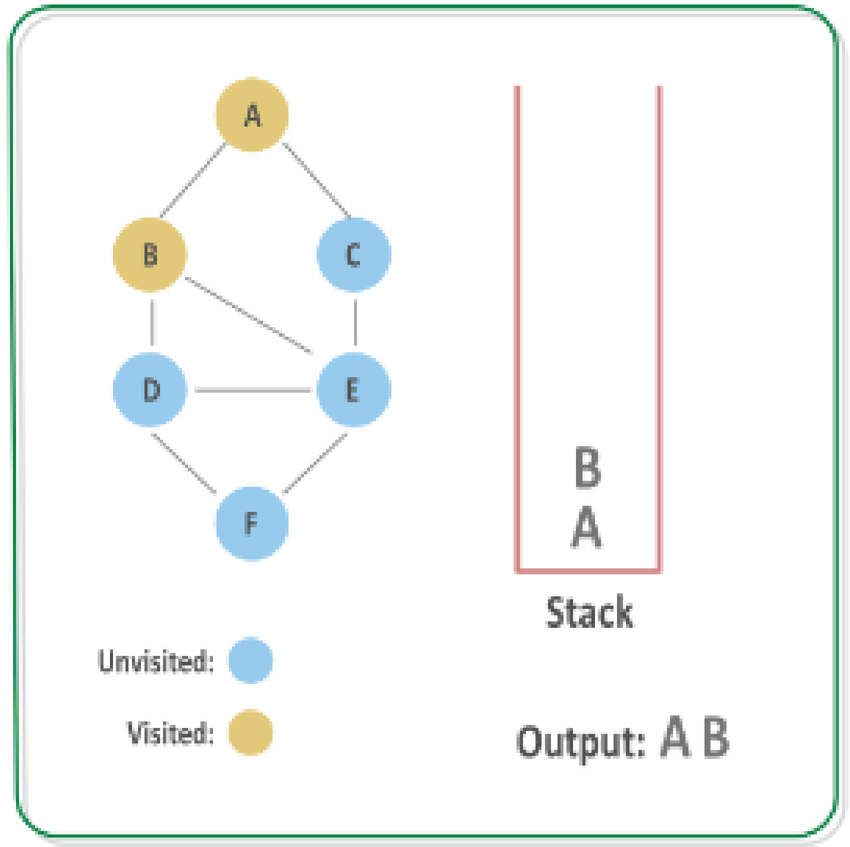
Step 1: Consider the below graph and apply the DFS algorithm recursively for every node using an auxiliary stack for recursive calls and an array to keep track of visited vertices.



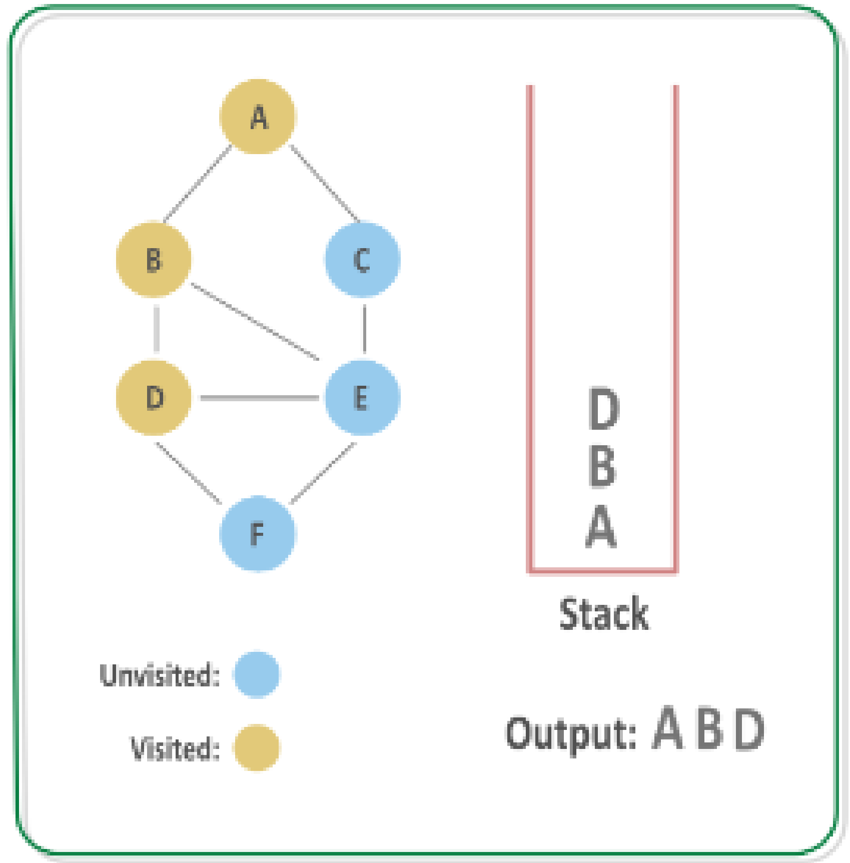
Step 2: Process the vertex A, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is B. The vertex A is put on the auxilary stack for now.



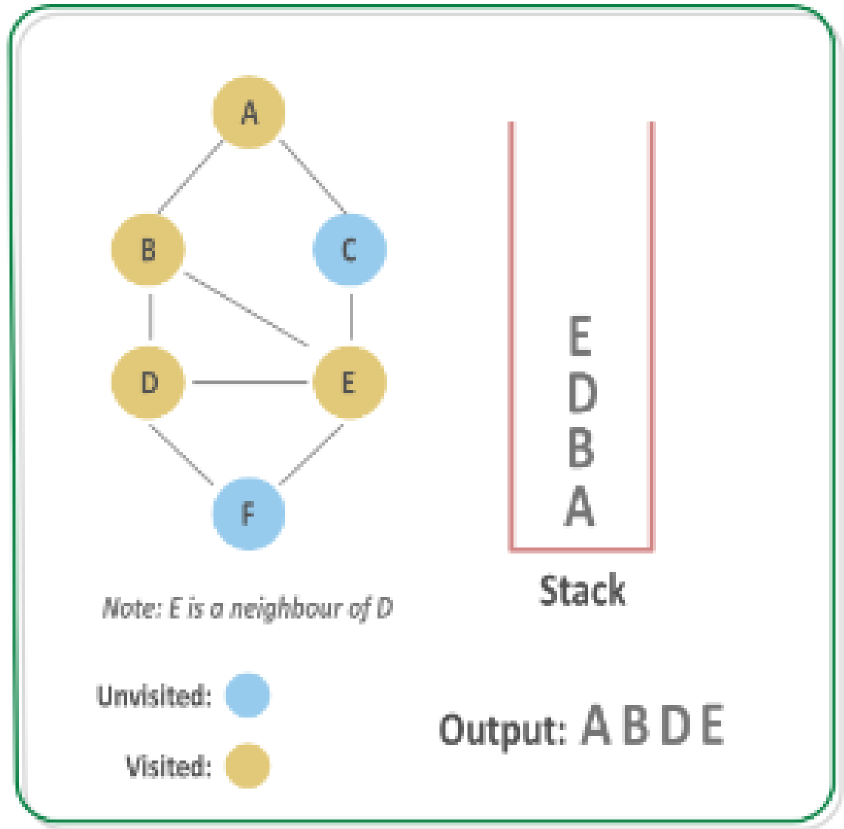
Step 3: Process the vertex B, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is D. The vertex B is put on the auxilary stack for now.



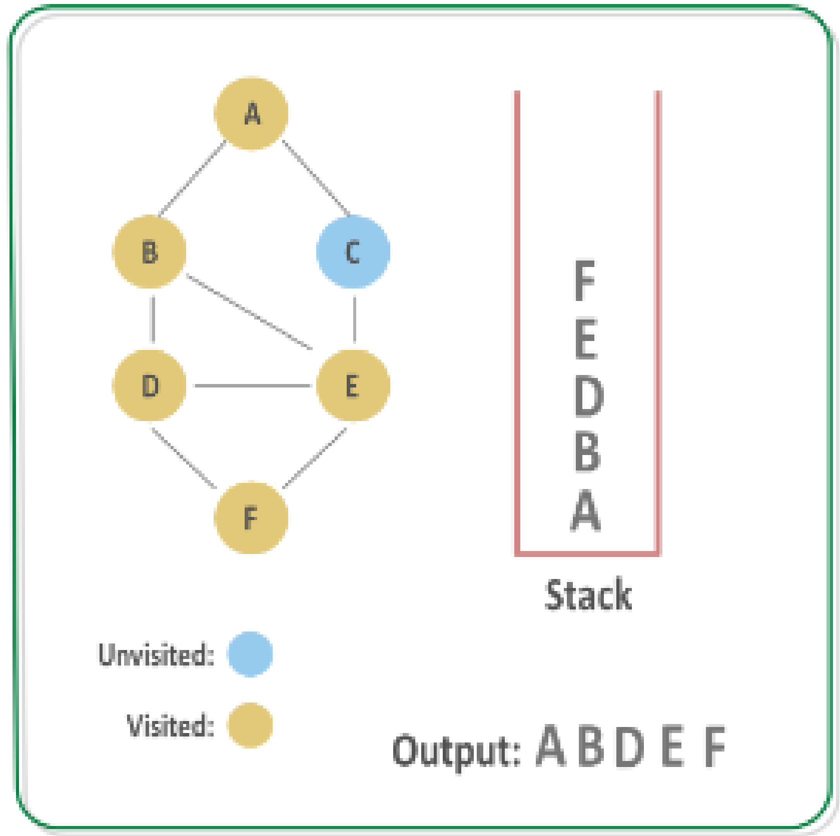
Step 4: Process the vertex D, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is E. The vertex D is put on the auxilary stack for now.



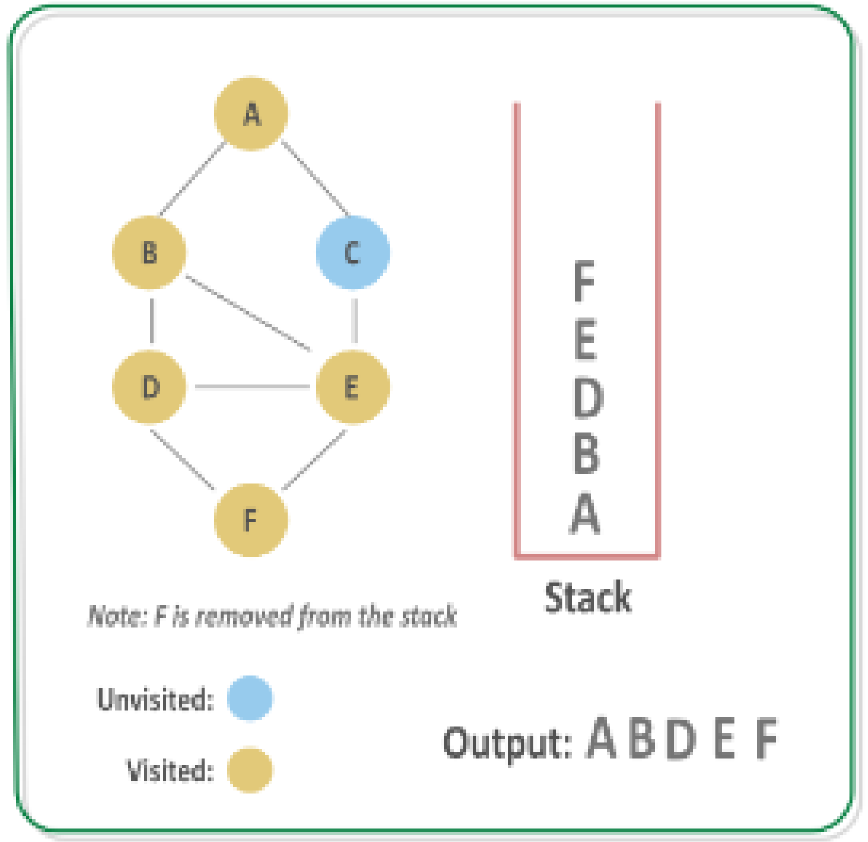
Step 5: Process the vertex E, mark it visited and call DFS for its adjacent vertices. Suppose the first adjacent vertex processed is F. The vertex E is put on the auxilary stack
for now.



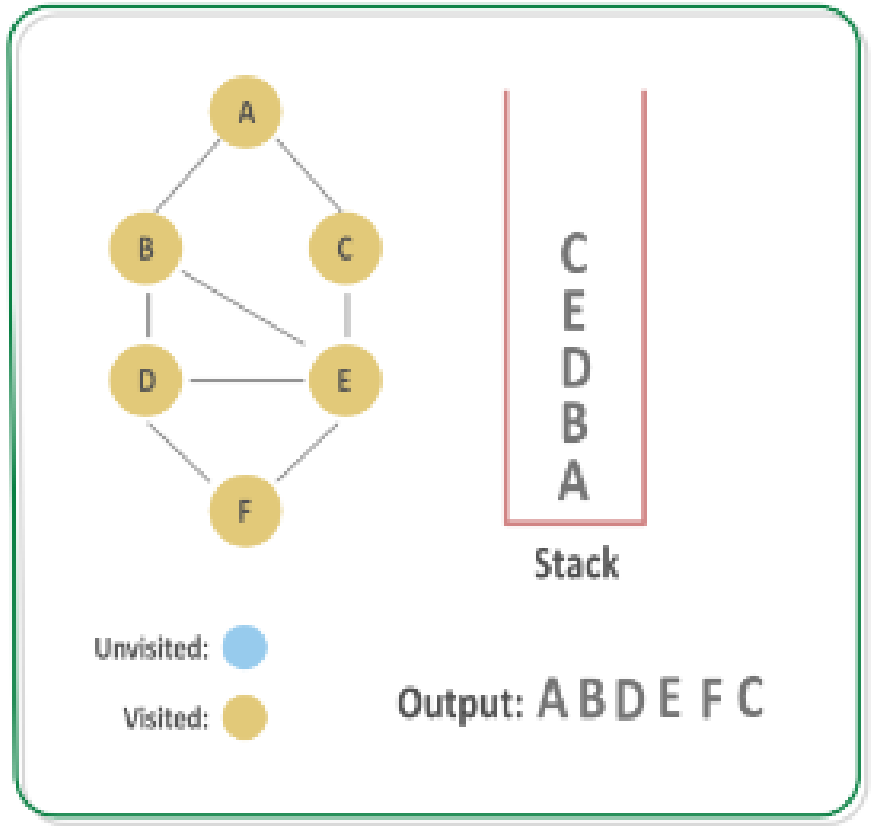
Step 6: Process the vertex F, mark it visited and call DFS for its adjacent vertices. There are no adjacent vertex of the vertex F, so backtrack to find a new path. The vertex F is put on the auxilary stack for now.



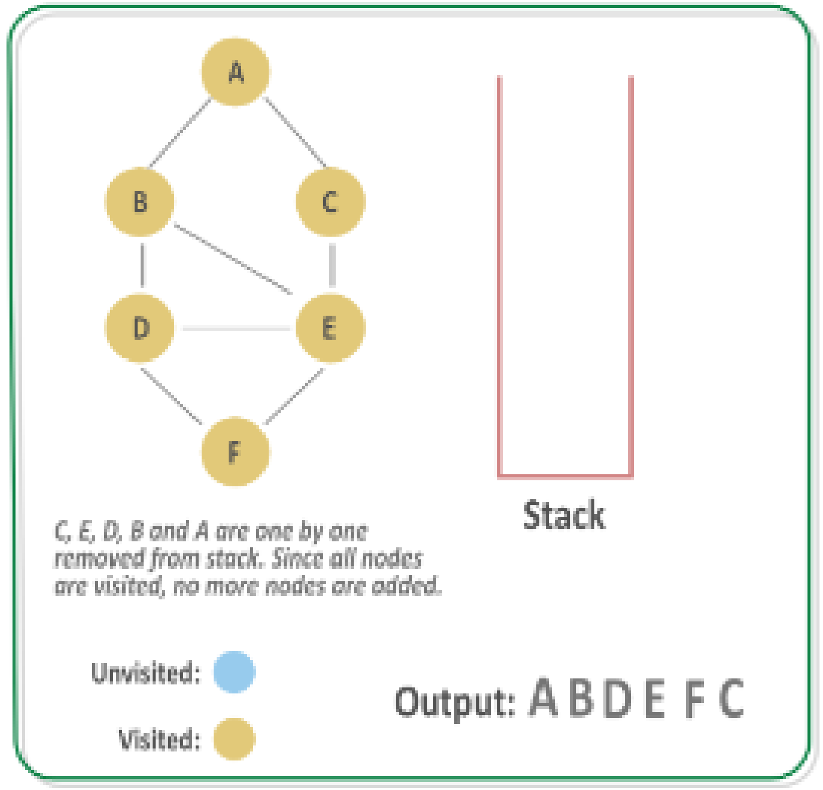
Step 7: Since the vertex F has no adjacent vertices left unvisited, backtrack and go to previous call, that is process any other adjacent vertex of node E, that is C.

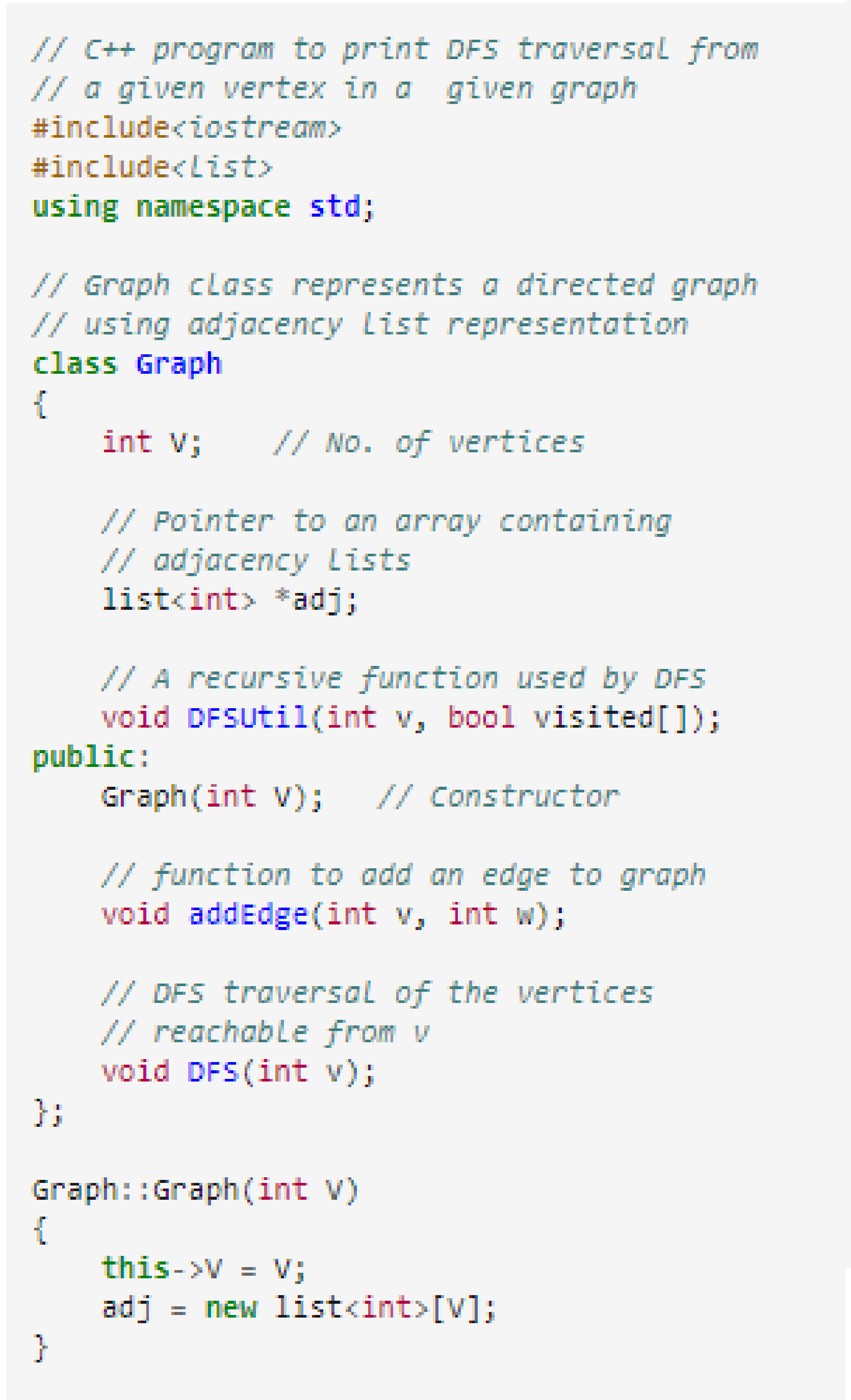


Step 8: Process the vertex C, mark it visited and call DFS for its adjacent vertices. The vertex C is put on the auxilary stack for now.

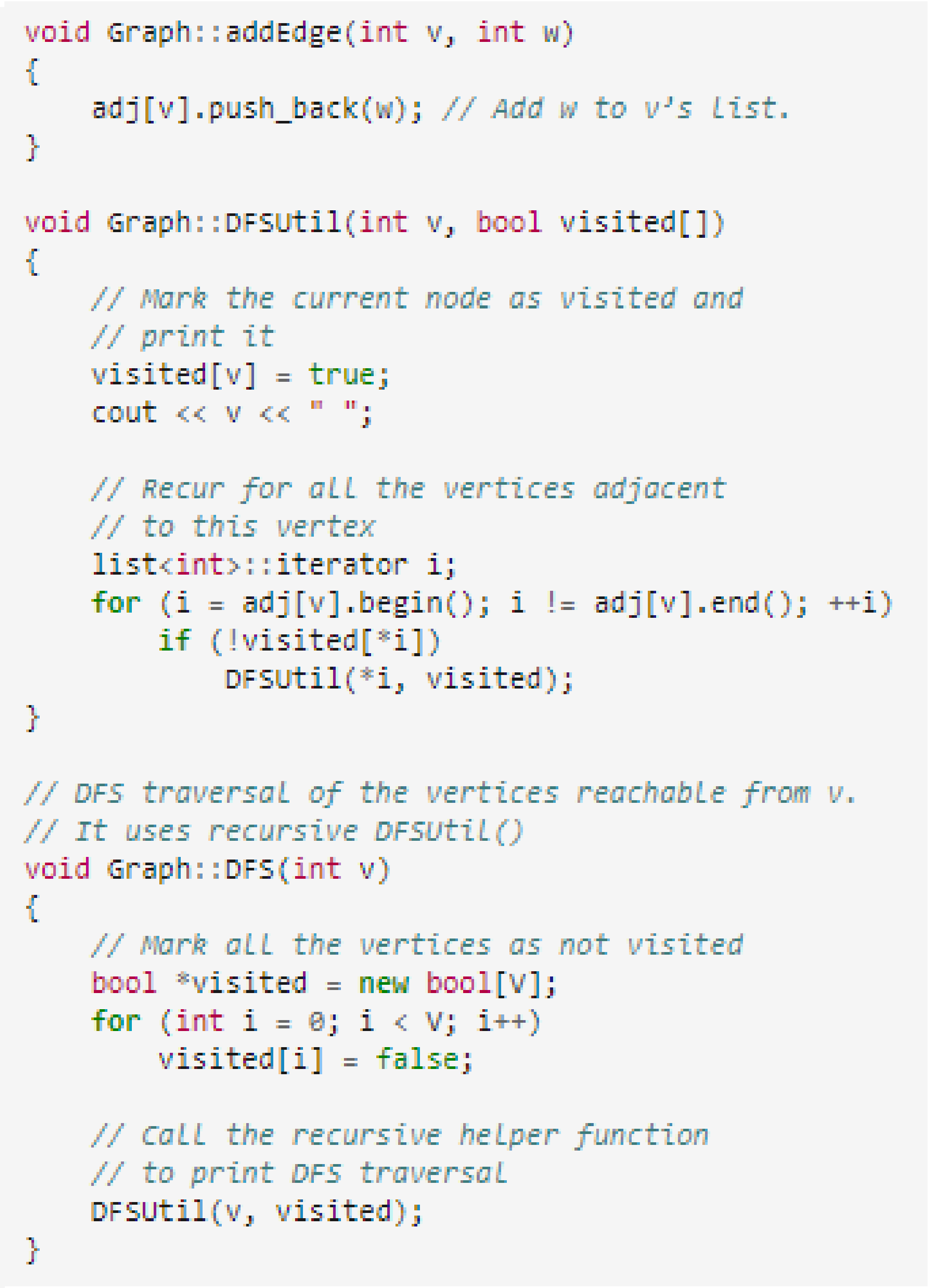


Step 9: Since there are no adjacent vertex of C, backtrack to find a new
path and keep removing nodes from stack until a new path is found. Since all of the nodes are processed so the stack will get empty.

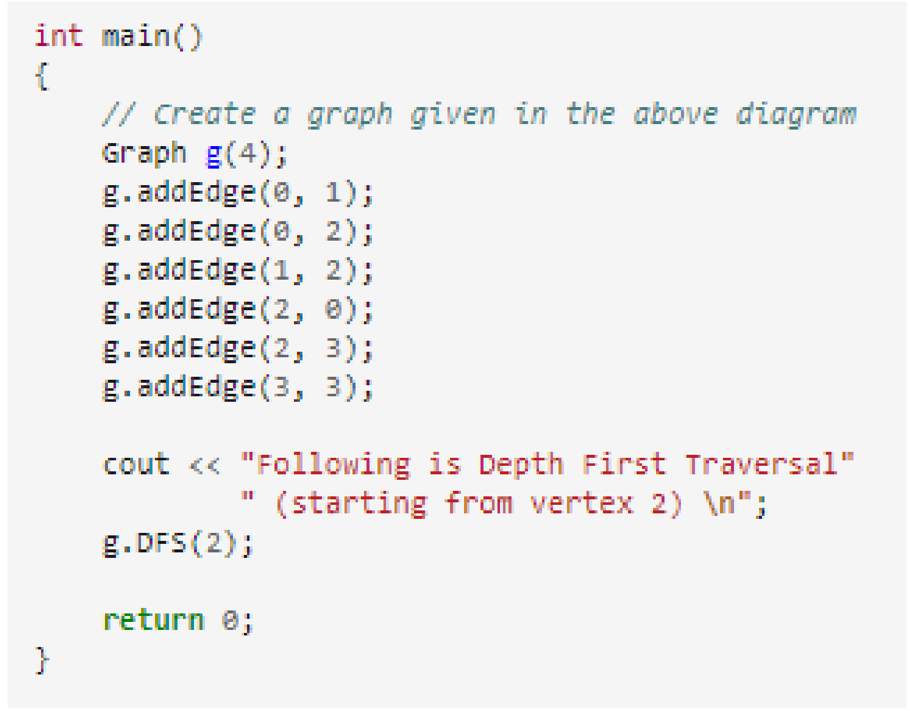

A
B
C
D
E
F
Stack
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.
Unvisited:
Visited:
Output: A B D E F C



```cpp
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
```



```cpp
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}
```



```cpp
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
         " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

Output:

```
Following is Depth First Traversal
(starting from vertex 2)

 2 0 1 3
```

Applications, Advantages and Disadvantages of DFS

Depth First Search is a widely used algorithm for traversing a graph. Here we have discussed some applications, advantages, and disadvantages of the algorithm.

Applications of Depth First Search:

1. Detecting cycle in a graph: A graph has a cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges.

2. Path Finding: We can specialize the DFS algorithm to find a path between two given vertices u and z.
Call DFS(G, u) with u as the start vertex.
Use a stack S to keep track of the path between the start vertex and the current vertex.
As soon as destination vertex z is encountered, return the path as the contents of the stack

3. Topological Sorting: Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers.

4. To test if a graph is bipartite: We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.

5. Finding Strongly Connected Components of a Graph: A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See **this** for DFS-based algo for finding Strongly Connected Components)

6. Solving puzzles with only one solution: such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.).

7. Web crawlers: Depth-first search can be used in the implementation of web crawlers to explore the links on a website.

8. Maze generation: Depth-first search can be used to generate random mazes.

9. Model checking: Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

10. Backtracking: Depth-first search can be used in backtracking algorithms.

Advantages of Depth First Search:

- Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth d and branching factor b (the number of children at each node, the outdegree) is O(bd) since it generates the same set of nodes as a breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
- If the depth-first search finds a solution without exploring much in a path then the time and space it takes will be very less.
- DFS requires less memory since only the nodes on the current path are stored. By chance, DFS may find a solution without examining much of the search space at all.

Disadvantages of Depth First Search:

- The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
- Depth-First Search is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution if more than one solution.

Key Points	BFS	DFS
1. Data Structure	It uses a queue for traversal.	It uses the stack for traversal.
2. Test	Used to test whether the graph is connected or not.	Used to test whether the graph is connected or not.
3. Computation	It is used to compute the existence of cycles in a graph.	Computes the spanning tree of a connected graph.
4. Priority Nodes	Priority is given to the nodes in the frontier i.e. breadth-first or level order.	Priority is given to the nodes in the depth.
5. Traversal order	Used to find the vertices of the same level	Used to find the path from one vertex to the deepest vertex.
6. Applied in Algorithms	Prims uses the technique. Shortest pathfinding algorithms use.	MST uses the technique. Backtracking algorithms use it.

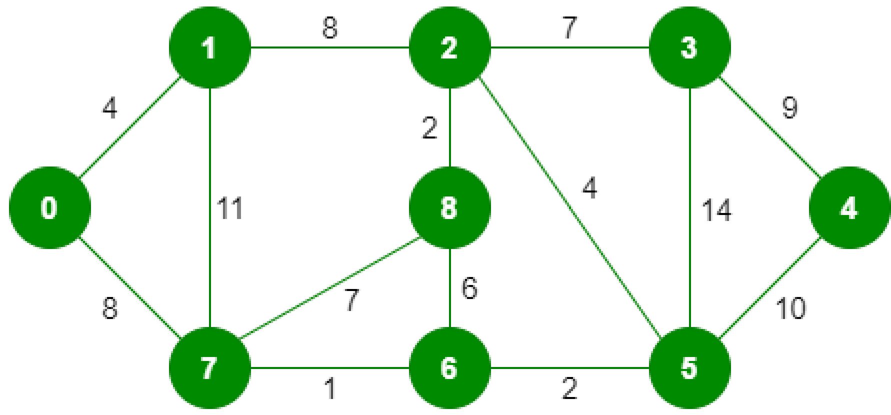
Introduction to Kruskal's Algorithm

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a **Greedy Algorithm**.

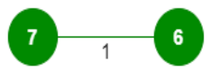
How to find MST using Kruskal's algorithm?

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

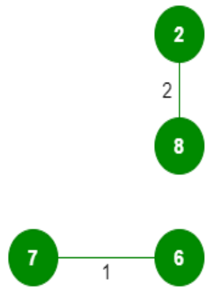


Step 1 Add edge 7-6 in the MST



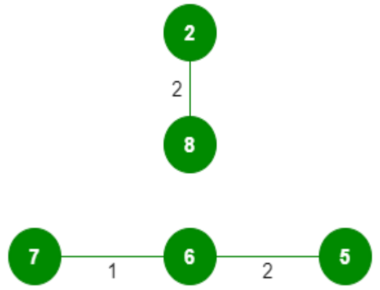
MST using Kruskal's algorithm

Step 2 Add edge 8-2 in the MST



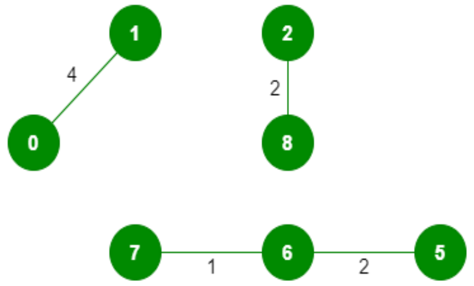
MST using Kruskal's algorithm

Step 3 Add edge 6-5 in the MST



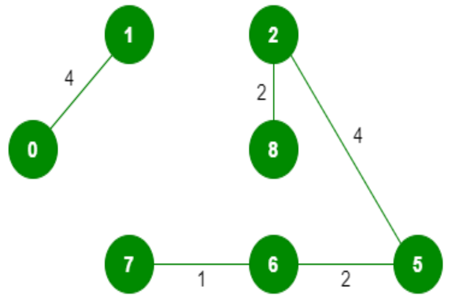
MST using Kruskal's algorithm

Step 4 Add edge 0-1 in the MST



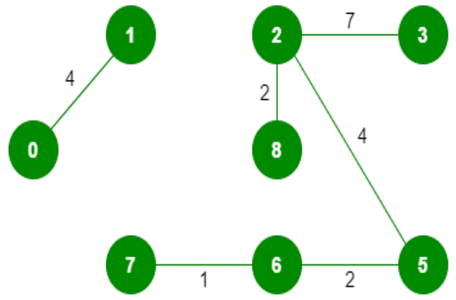
MST using Kruskal's algorithm

Step 5 Add edge 2-5 in the MST



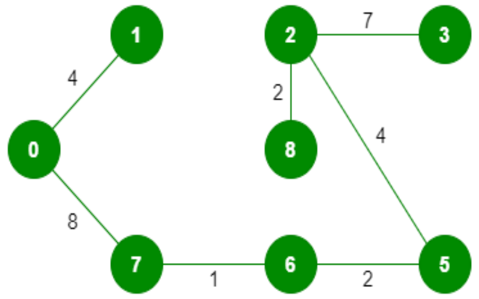
MST using Kruskal's algorithm

Step 6 Add edge 2-3 in the MST as 8-6 can't be added



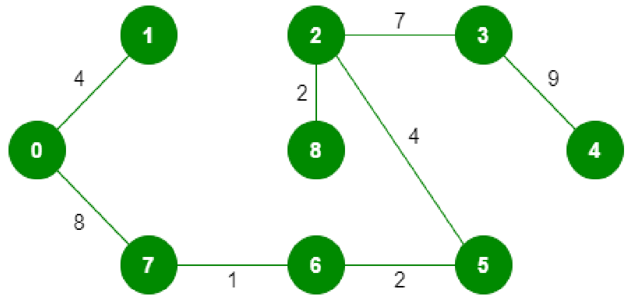
MST using Kruskal's algorithm

Step 7 Add edge 0-7 in the MST as 7-8 can't be added



MST using Kruskal's algorithm

Step 8 Add edge 3-4 in the MST. It completes the MST



MST using Kruskal's algorithm

Prim's Minimum Spanning Tree Algorithm

What is Minimum Spanning Tree?

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree (MST)** or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Number of edges in a minimum spanning tree:

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Prim's Algorithm

Prim’s algorithm is a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory

So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work?

The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

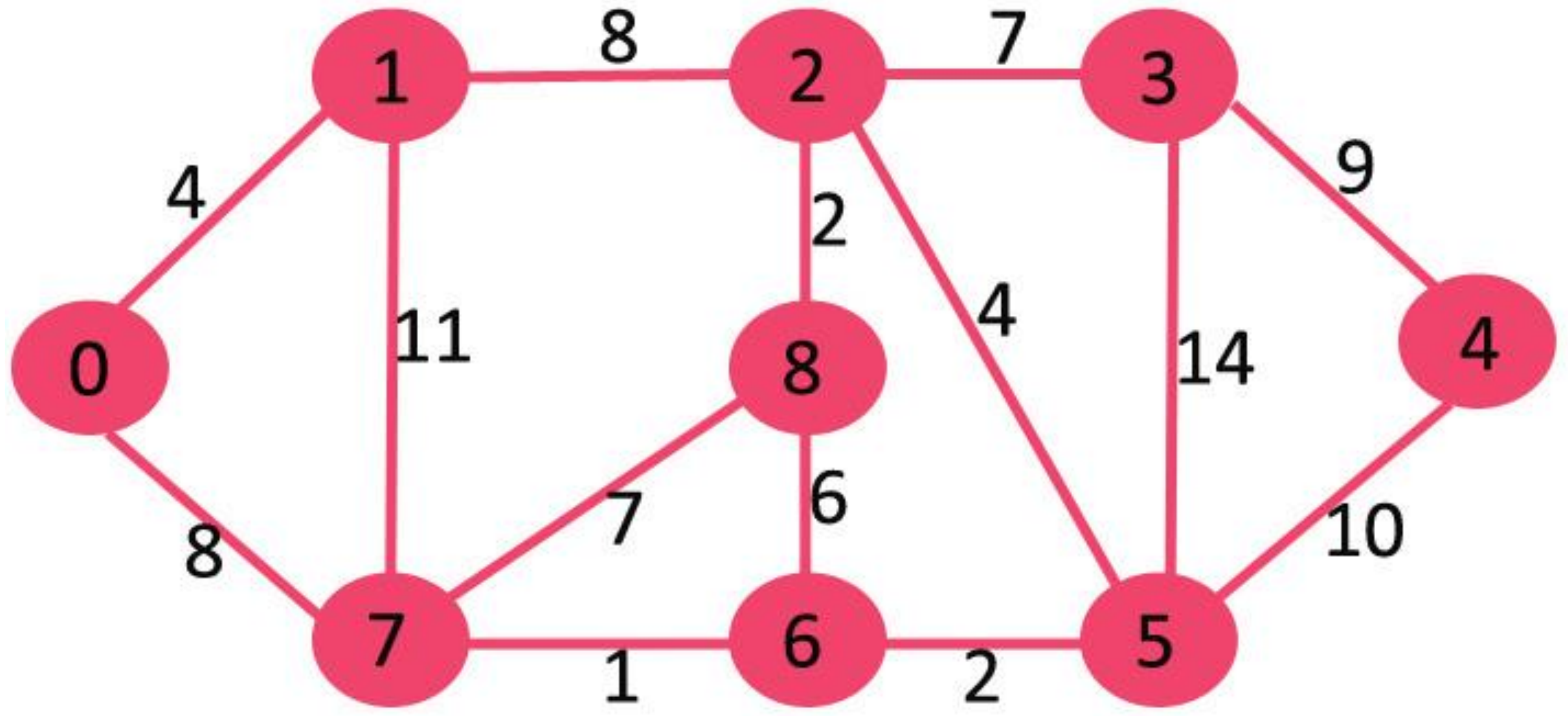
Algorithm:

1. Create a set *mstSet* that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices:
    - Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
    - Include *u* to mstSet.
    - Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*.

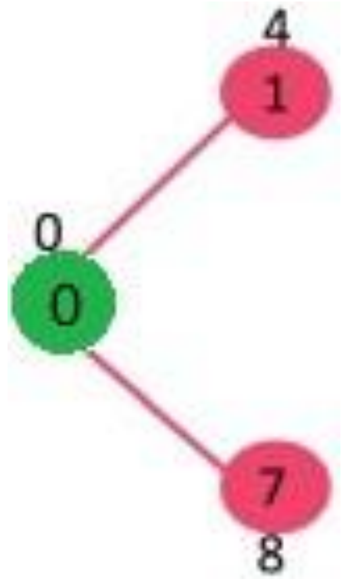
The idea of using key values is to pick the minimum weight edge from cut

The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

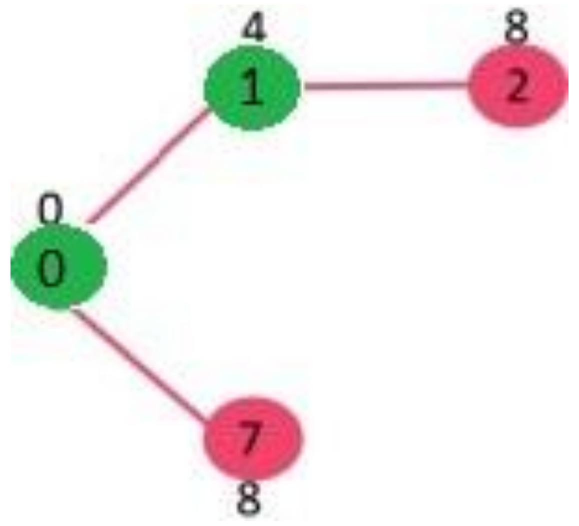
Let us understand this with the help of following example:



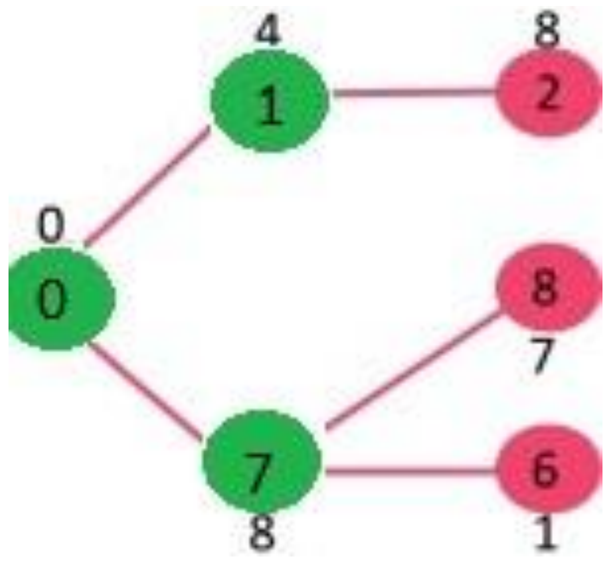
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet* , update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

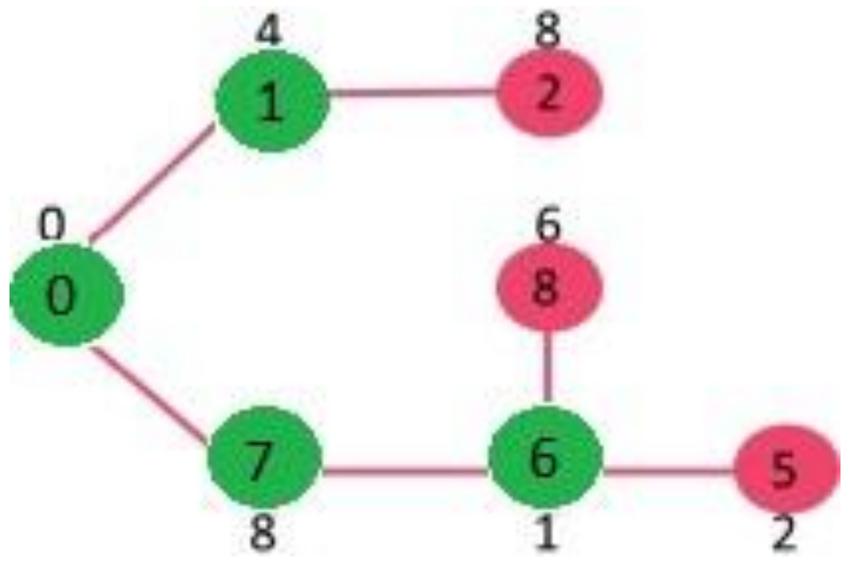


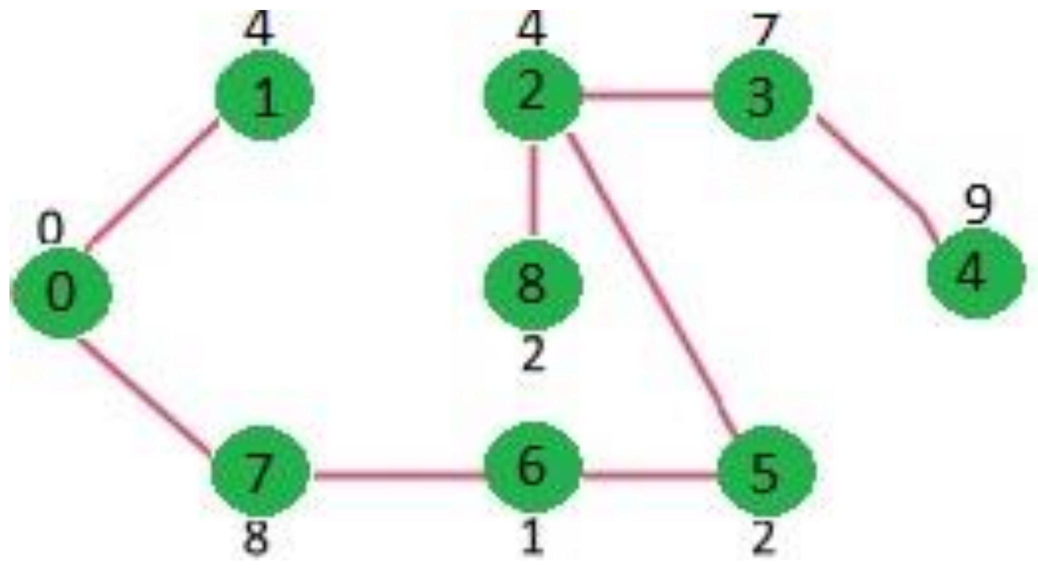
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

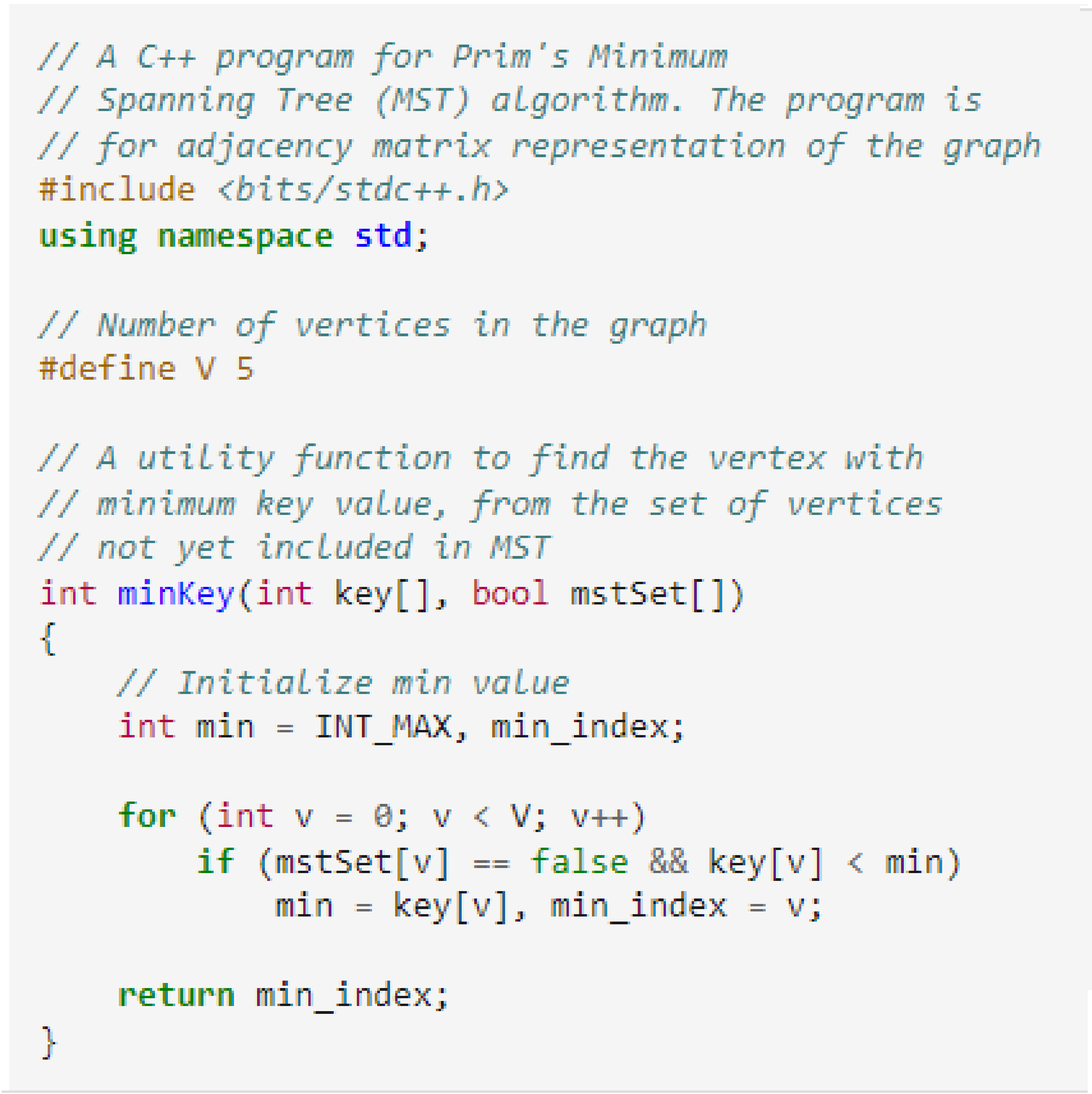


Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).

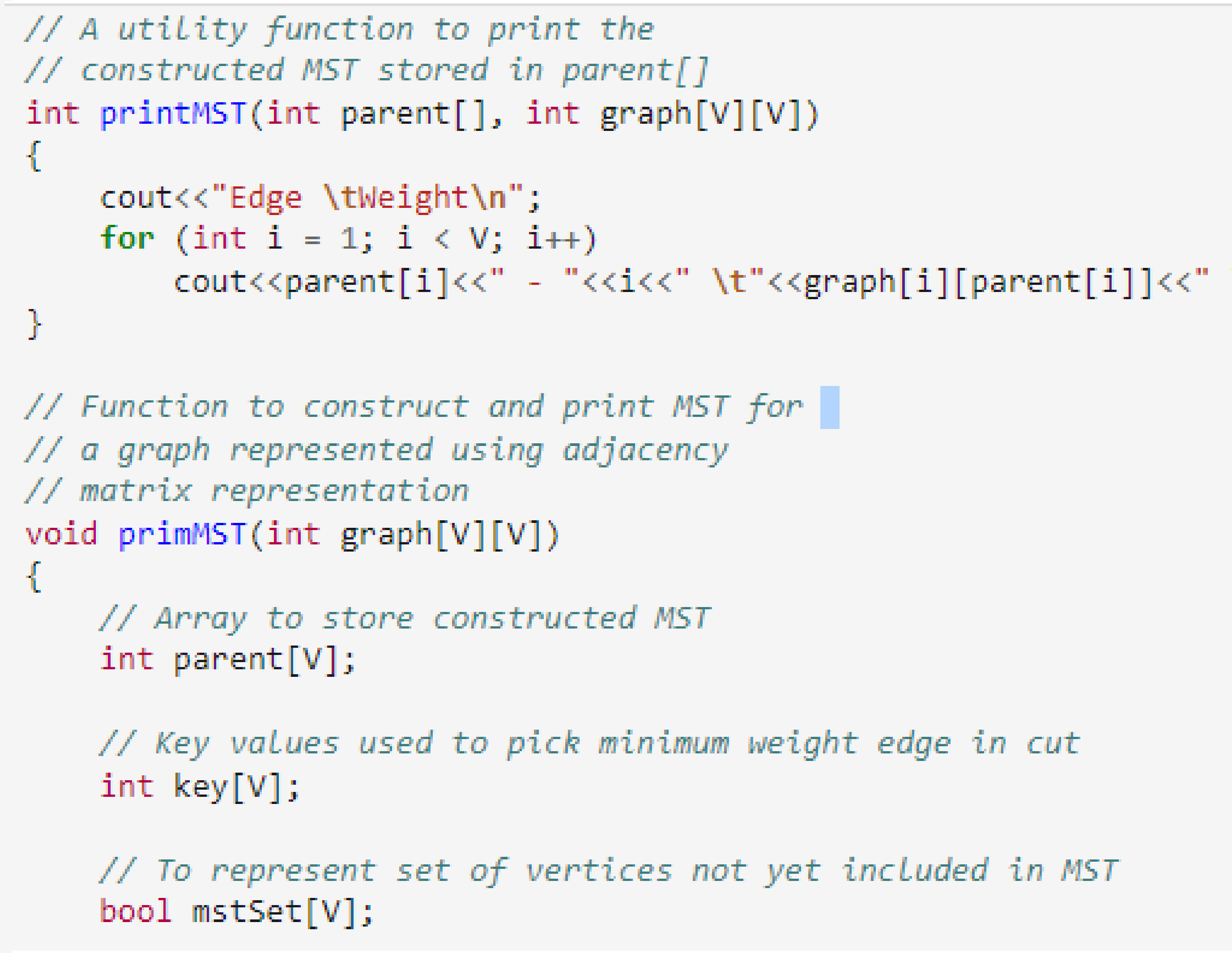




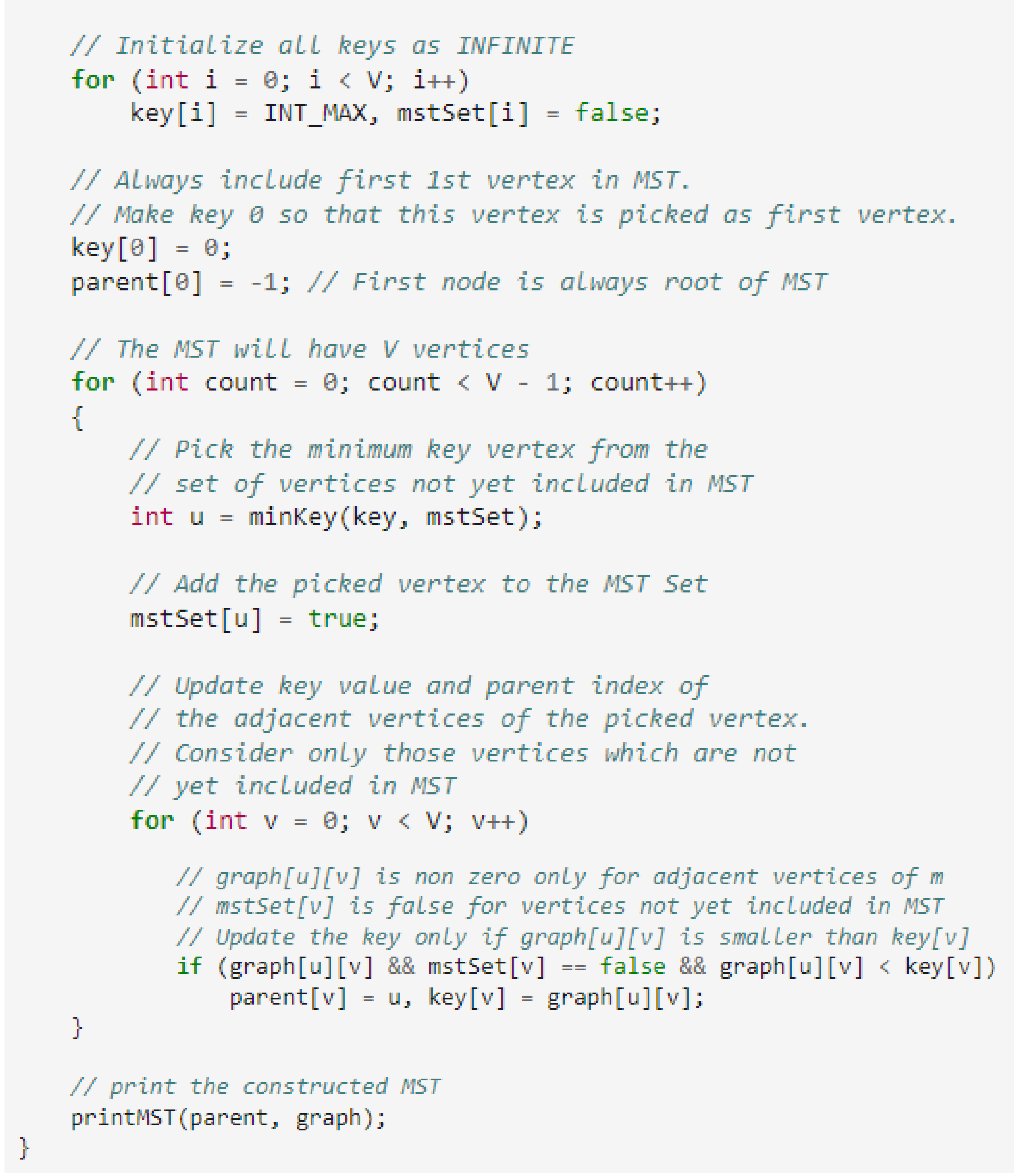




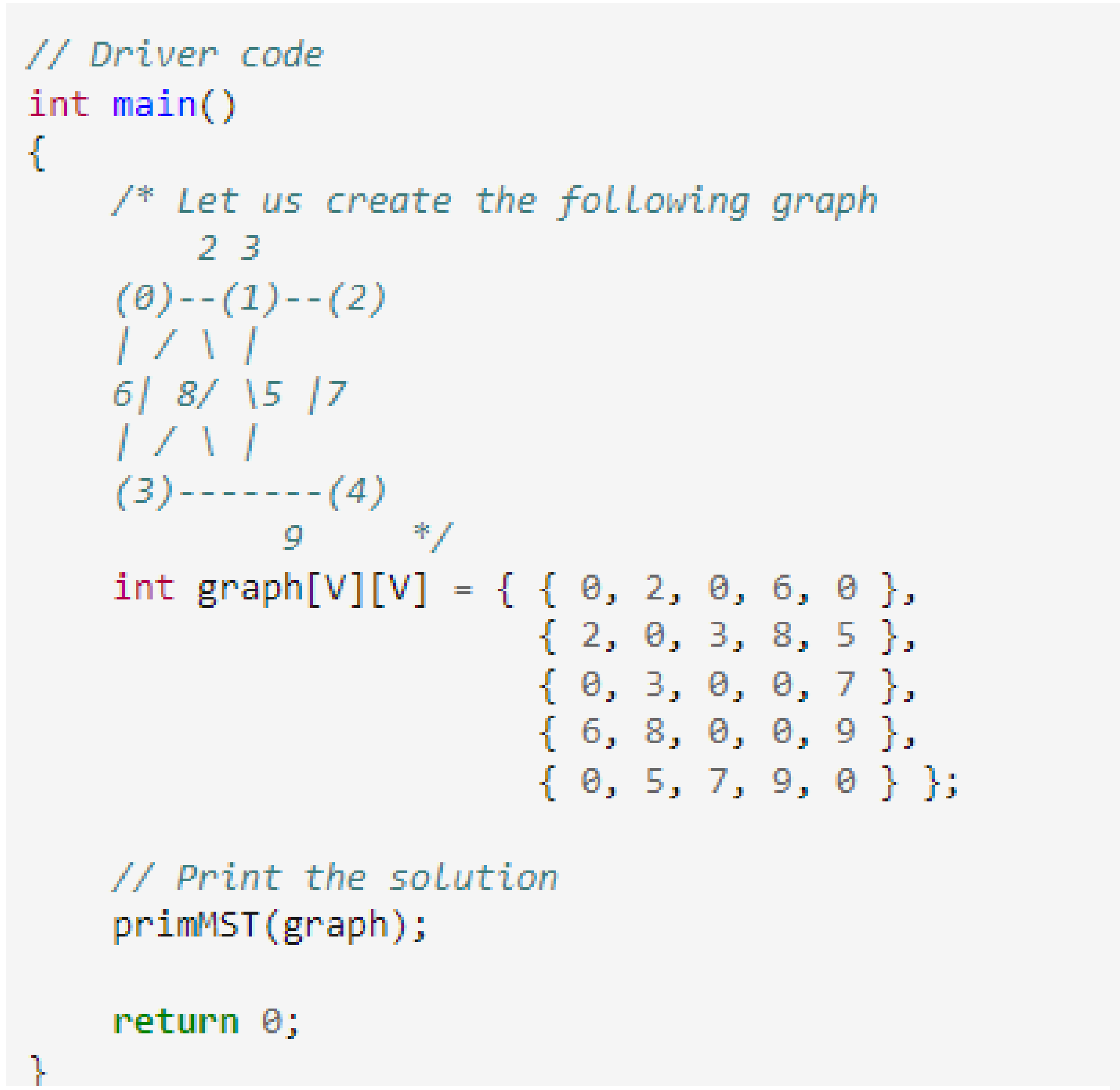
```cpp
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
```



```cpp
// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<"
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices not yet included in MST
    bool mstSet[V];
```



```
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}
```



```
// Driver code
int main()
{
    /* Let us create the following graph
          2    3
    (0)--(1)--(2)
    |   / \   |
    6| 8/   \5 |7
    | /      \ |
    (3)-------(4)
         9        */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

Output

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.
Auxiliary Space: O(V)

Dijkstra's Algorithm for Shortest Path in a Weighted Graph

Given a graph and a source vertex in the graph, ***find the shortest paths from single source to all vertices in the given graph****.*

Dijkstra's algorithm is a variation of the BFS algorithm. In Dijkstra's Algorithm, a SPT*(shortest path tree)* is generated with given source as root. Each node at this SPT stores the value of the shortest path from the source vertex to the current vertex. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

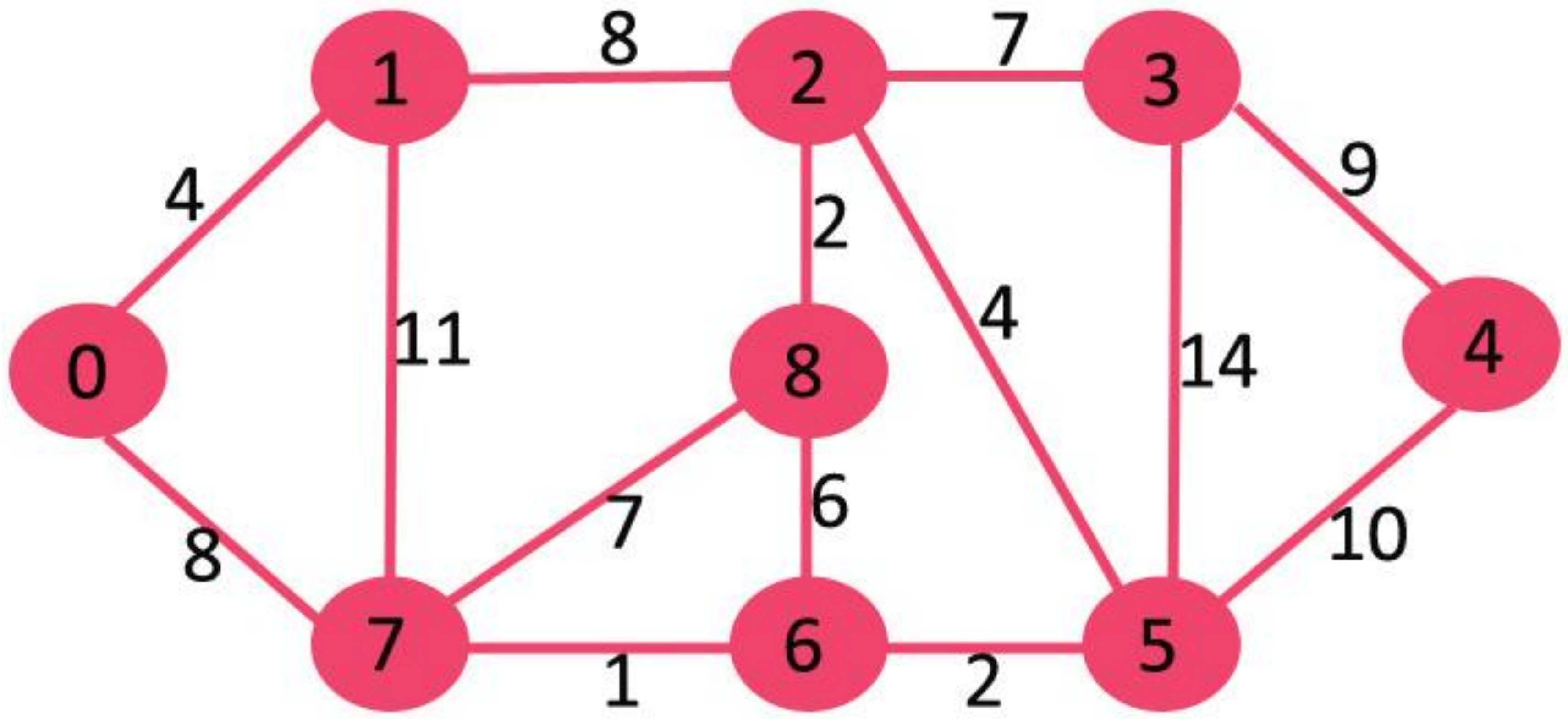
Below is the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given weighted graph.

Algorithm:

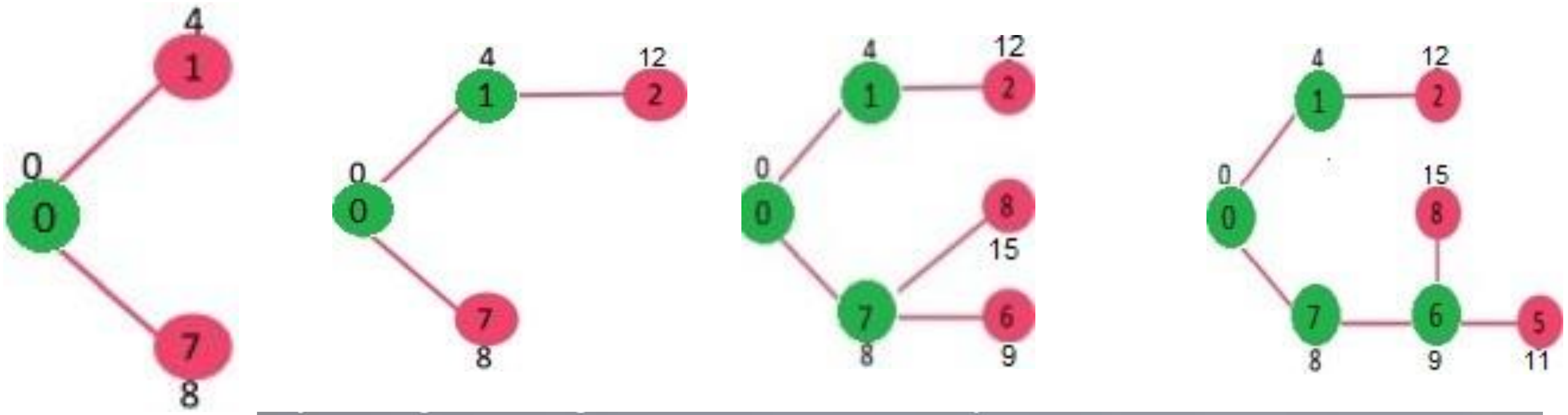
1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* doesn't include all vertices:
    - Pick a vertex u which is not there in *sptSet* and has minimum distance value.
    - Include u to *sptSet*.
    - Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

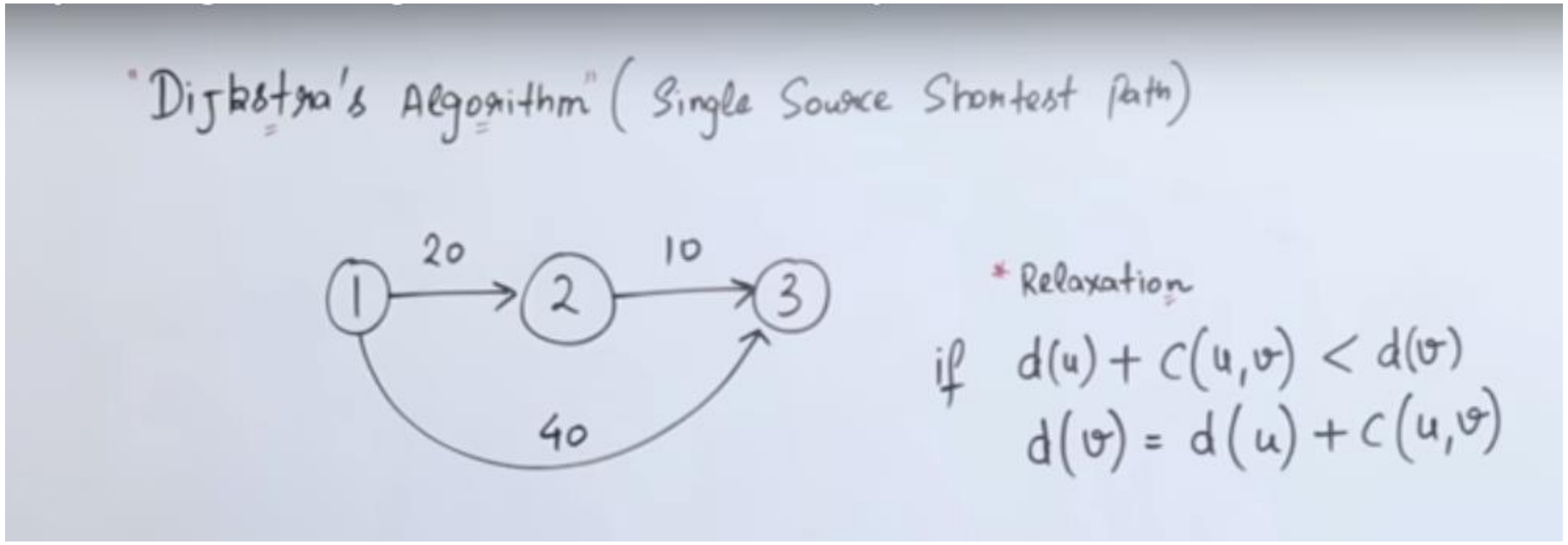
Let us understand the above algorithm with the help of an example. Consider the below given graph:

The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value.



The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



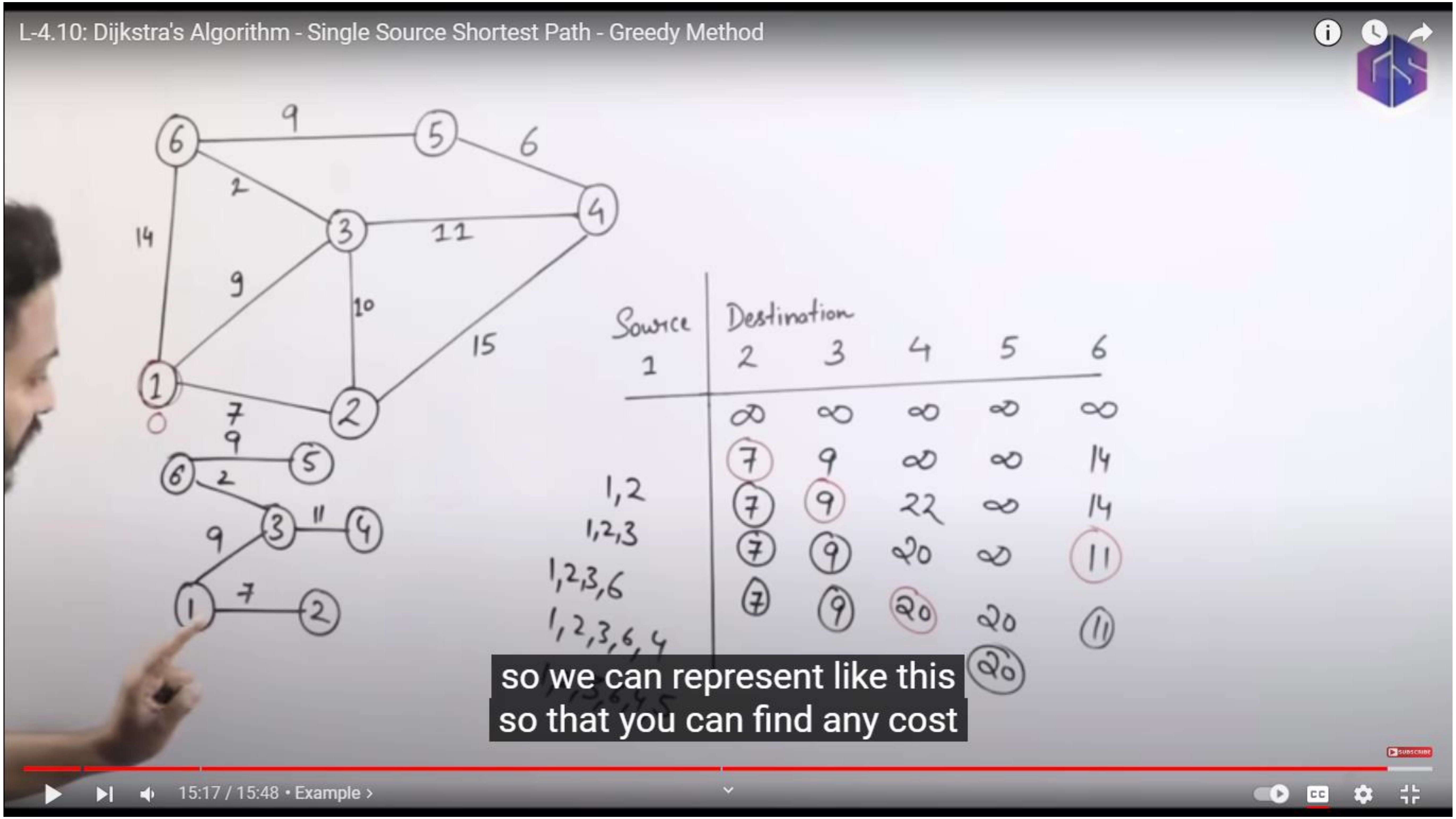


"Dijkstra's Algorithm" (Single Source Shortest Path)

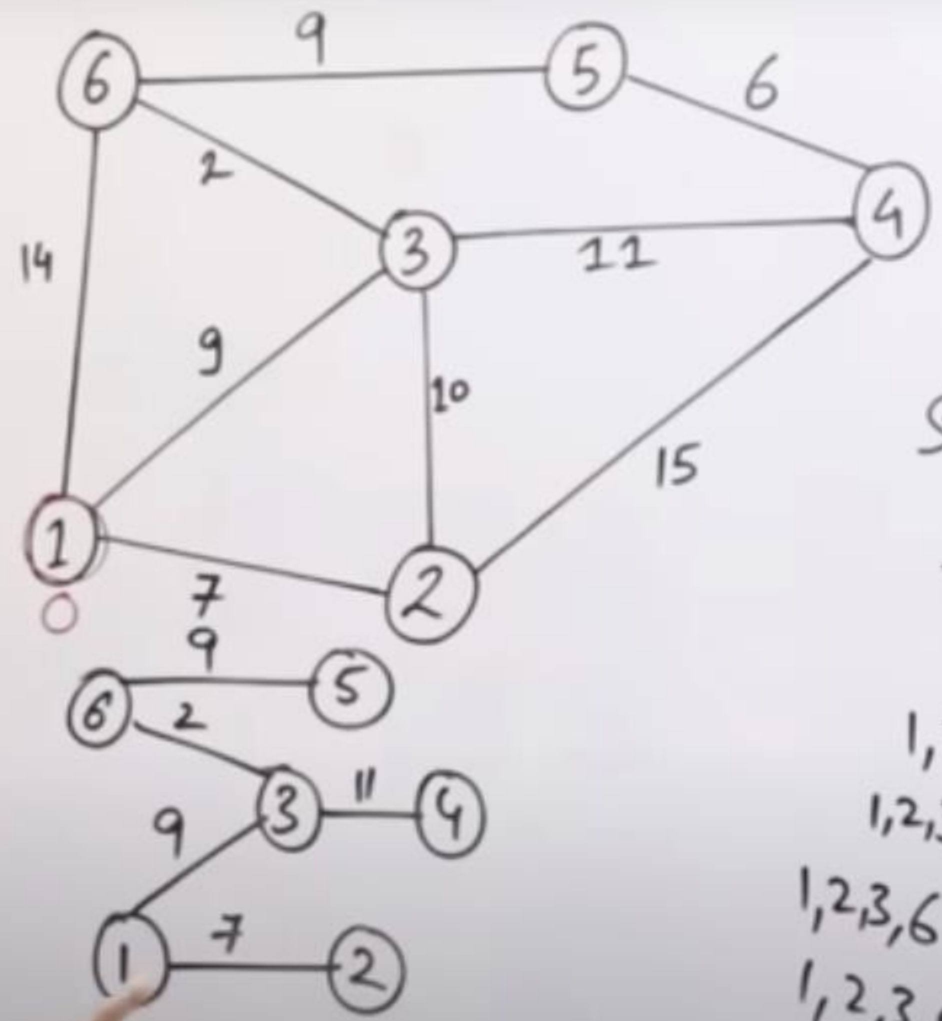
* Relaxation

if $d(u) + c(u,v) < d(v)$

$d(v) = d(u) + c(u,v)$



L-4.10: Dijkstra's Algorithm - Single Source Shortest Path - Greedy Method



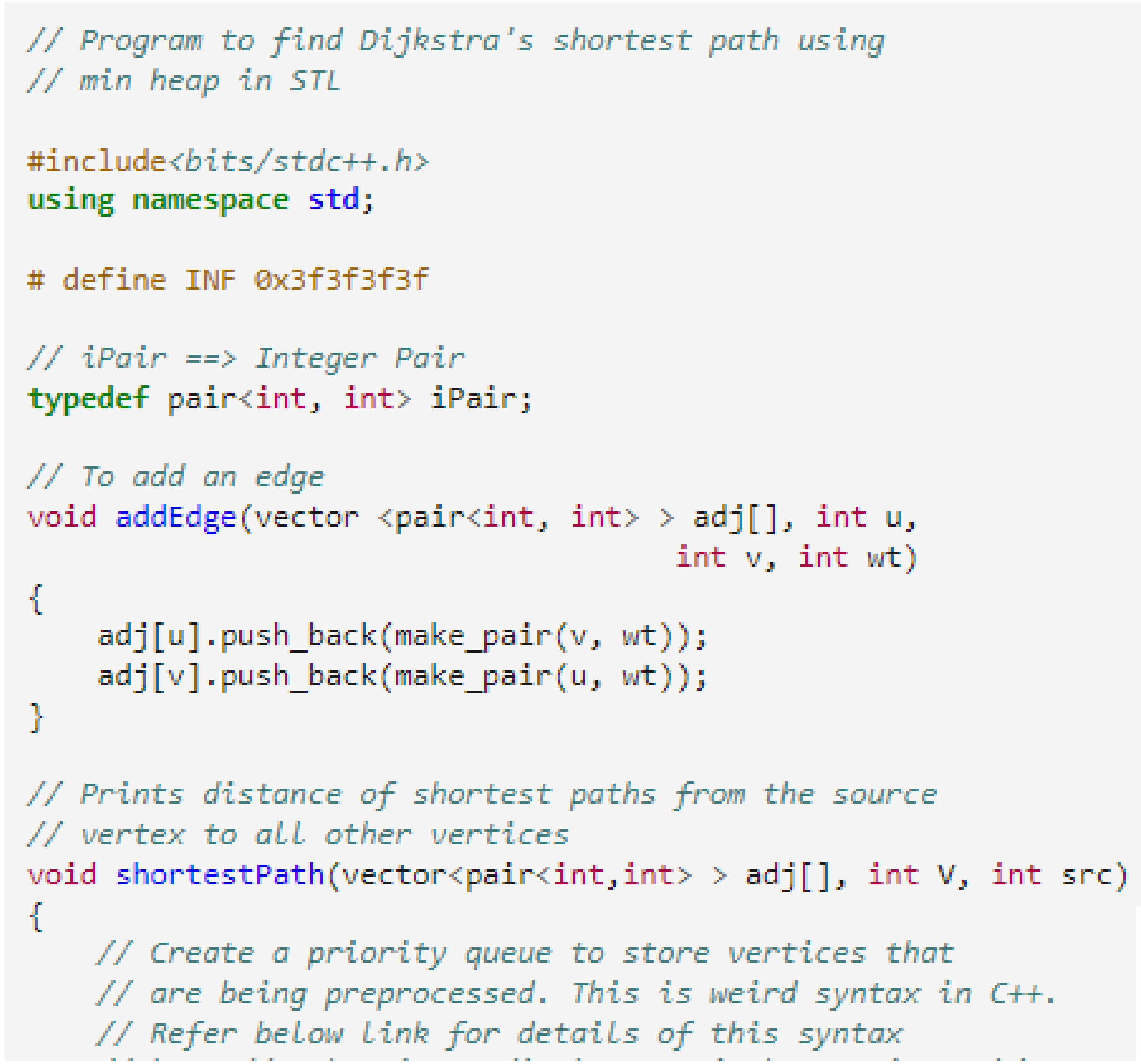
Source 1	Destination 2	3	4	5	6
	∞	∞	∞	∞	∞
	7	9	∞	∞	14
1,2	7	9	22	∞	14
1,2,3	7	9	20	∞	11
1,2,3,6	7	9	20	20	11
1,2,3,6,4				20	
[illegible]					

so we can represent like this
so that you can find any cost

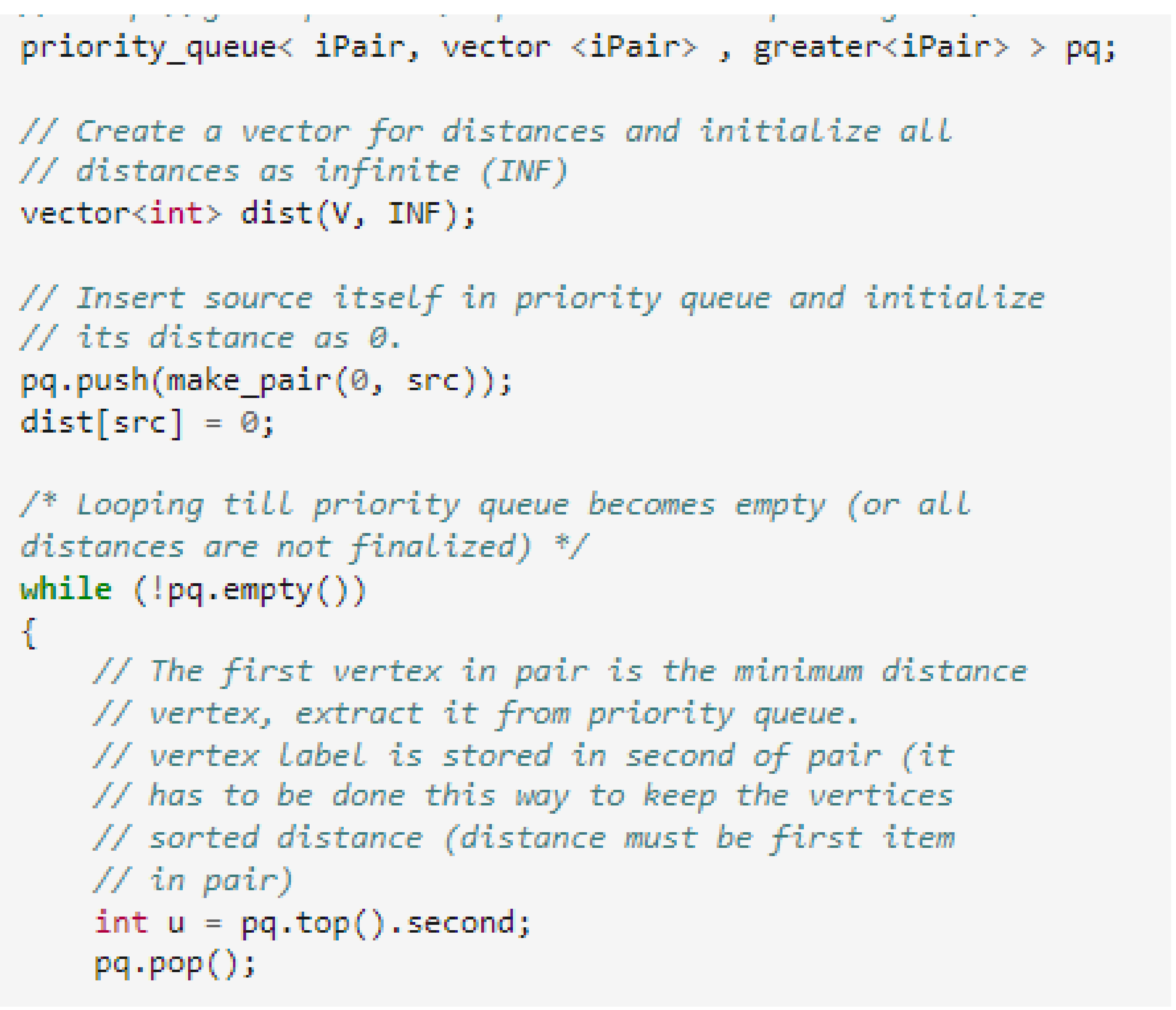
15:17 / 15:48 • Example ›

Implementation:
Since at every step we need to find the vertex with minimum distance from the source vertex from the set of vertices currently not added to the SPT, so we can use a min heap for easier and efficient implementation. Below is the complete algorithm using priority_queue(min heap) to implement Dijkstra's Algorithm:

```
1) Initialize distances of all vertices as infinite.

2) Create an empty priority_queue pq. Every item of pq is a pair (weight, vertex). Weight (or distance) is used as the first item of
pair as the first item is by default used to compare two pairs

3) Insert source vertex into pq and make its distance as 0.

4) While either pq doesn't become empty
        a) Extract minimum distance vertex from pq. Let the extracted vertex be u.
        b) Loop through all adjacent of u and do following for every vertex v.
                // If there is a shorter path to v through u.
                If $dist[v] > dist[u] + weight(u, v)$
                        (i) Update distance of v, i.e., do $dist[v] = dist[u] + weight(u, v)$
                        (ii) Insert v into the pq (Even if v is already there)

5) Print distance array dist[] to print all shortest paths.
```



```
// Program to find Dijkstra's shortest path using
// min heap in STL

#include<bits/stdc++.h>
using namespace std;

# define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
                                     int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Prints distance of shortest paths from the source
// vertex to all other vertices
void shortestPath(vector<pair<int,int> > adj[], int V, int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
```



```
priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

// Create a vector for distances and initialize all
// distances as infinite (INF)
vector<int> dist(V, INF);

// Insert source itself in priority queue and initialize
// its distance as 0.
pq.push(make_pair(0, src));
dist[src] = 0;

/* Looping till priority queue becomes empty (or all
distances are not finalized) */
while (!pq.empty())
{
    // The first vertex in pair is the minimum distance
    // vertex, extract it from priority queue.
    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted distance (distance must be first item
    // in pair)
    int u = pq.top().second;
    pq.pop();
```

```cpp
        // Get all adjacent of u.
        for (auto x : adj[u])
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = x.first;
            int weight = x.second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

Time Complexity: The time complexity of the Dijkstra's Algorithm when implemented using a min heap is O(E * logV), where E is the number of Edges and V is the number of vertices.

Note: The Dijkstra's Algorithm doesn't work in the case when the Graph has negative edge weight

```cpp
// Driver Code
int main()
{
    int V = 9;
    vector<iPair > adj[V];

    // making above shown graph
    addEdge(adj, 0, 1, 4);
    addEdge(adj, 0, 7, 8);
    addEdge(adj, 1, 2, 8);
    addEdge(adj, 1, 7, 11);
    addEdge(adj, 2, 3, 7);
    addEdge(adj, 2, 8, 2);
    addEdge(adj, 2, 5, 4);
    addEdge(adj, 3, 4, 9);
    addEdge(adj, 3, 5, 14);
    addEdge(adj, 4, 5, 10);
    addEdge(adj, 5, 6, 2);
    addEdge(adj, 6, 7, 1);
    addEdge(adj, 6, 8, 6);
    addEdge(adj, 7, 8, 7);

    shortestPath(adj, V, 0);

    return 0;
}
```

Output:

```
Vertex   Distance from Source
0                 0
1                 4
2                 12
3                 19
4                 21
5                 11
6                 9
7                 8
8                 14
```

Bellman-Ford Algorithm for Shortest Path

Problem: Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra's algorithm for this problem. **Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (**with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.* ***Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra****.*

Algorithm:
Input: Graph and a source vertex *src*.
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1. This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph. Do following for each edge u-v:
   If dist[v] > dist[u] + weight of edge uv, then update dist[v] as: *dist[v] = dist[u] + weight of edge uv*.
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v. If dist[v] > dist[u] + weight of edge **uv**, then "Graph contains negative weight cycle".

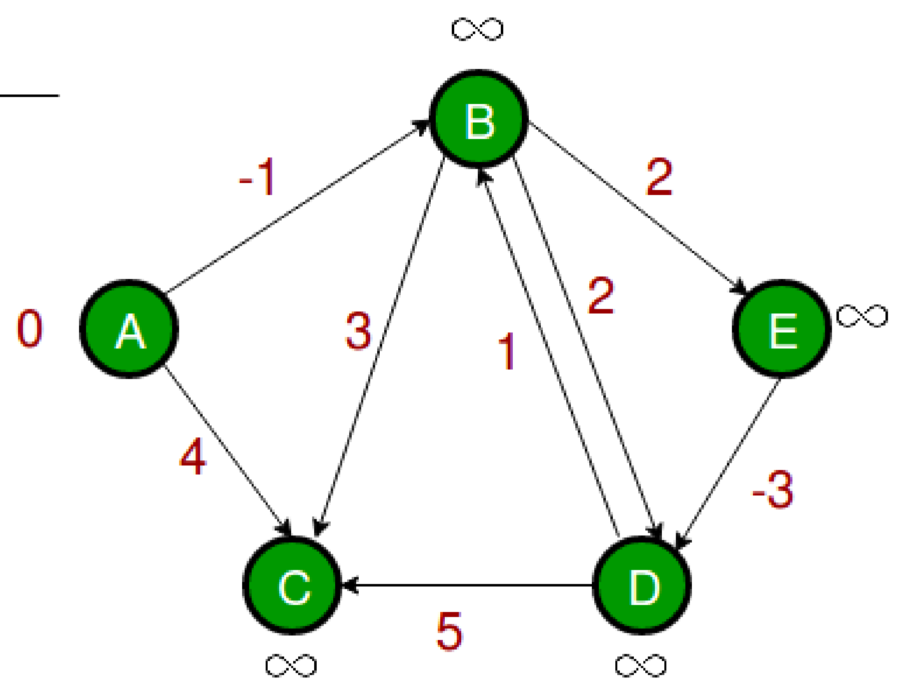
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. **If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.**

How does this work? Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most **i** edges are calculated. **There can be maximum |V| - 1 edge in any simple path, that is why the outer loop runs |v| - 1 time.** The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edge.

Example:
Let us understand the algorithm with following example graph. The images are taken from this source.

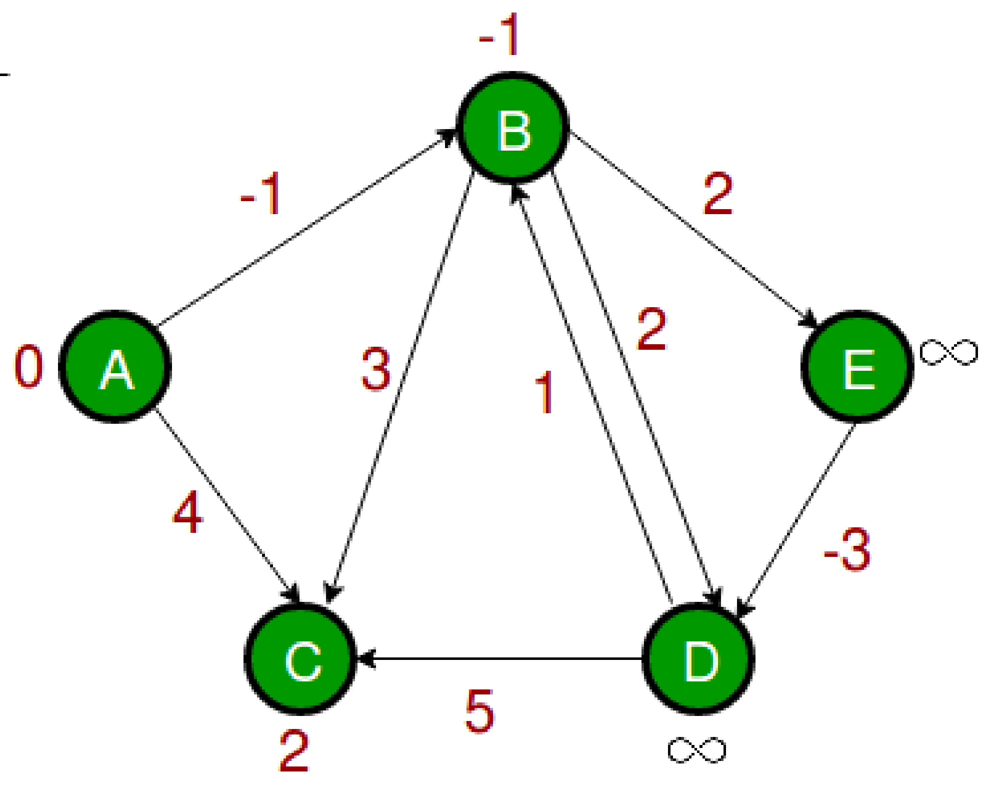
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. **Total number of vertices in the graph is 5, so *all edges must be processed 4 times.***

A	B	C	D	E
0	∞	∞	∞	∞



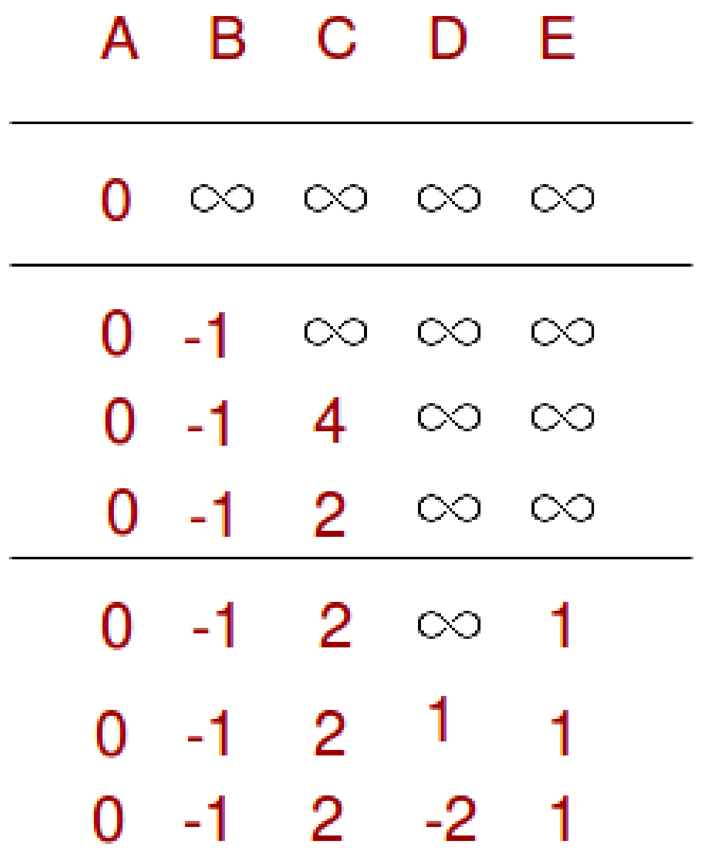
Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

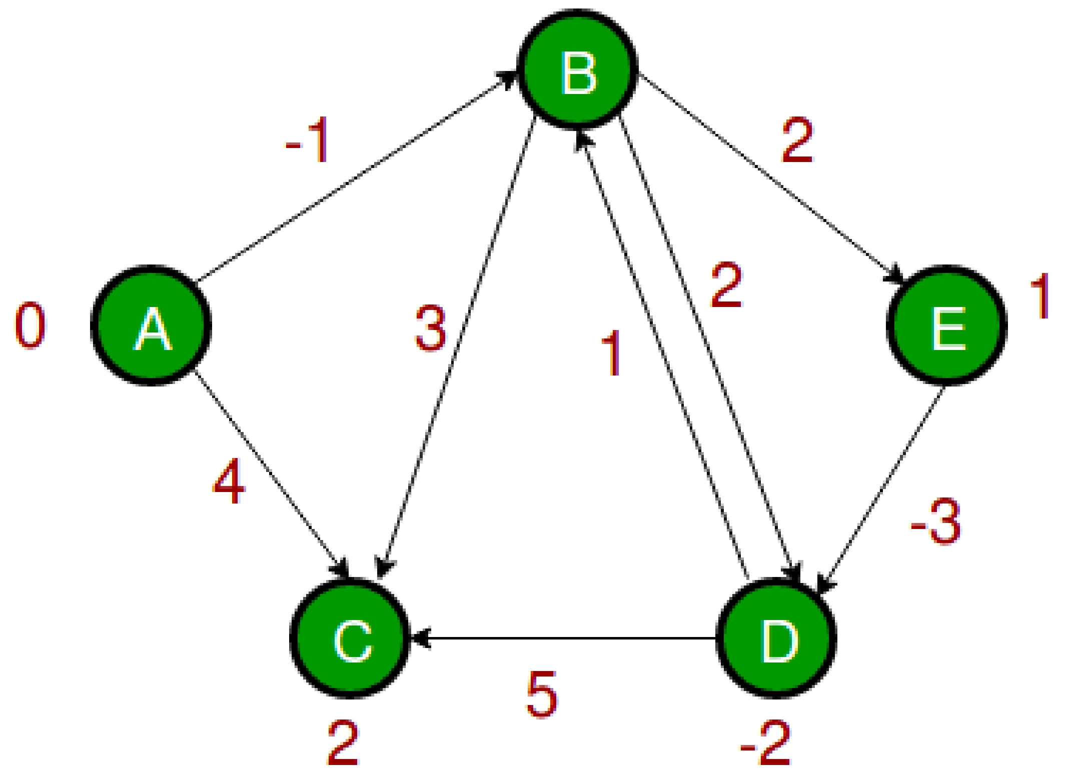


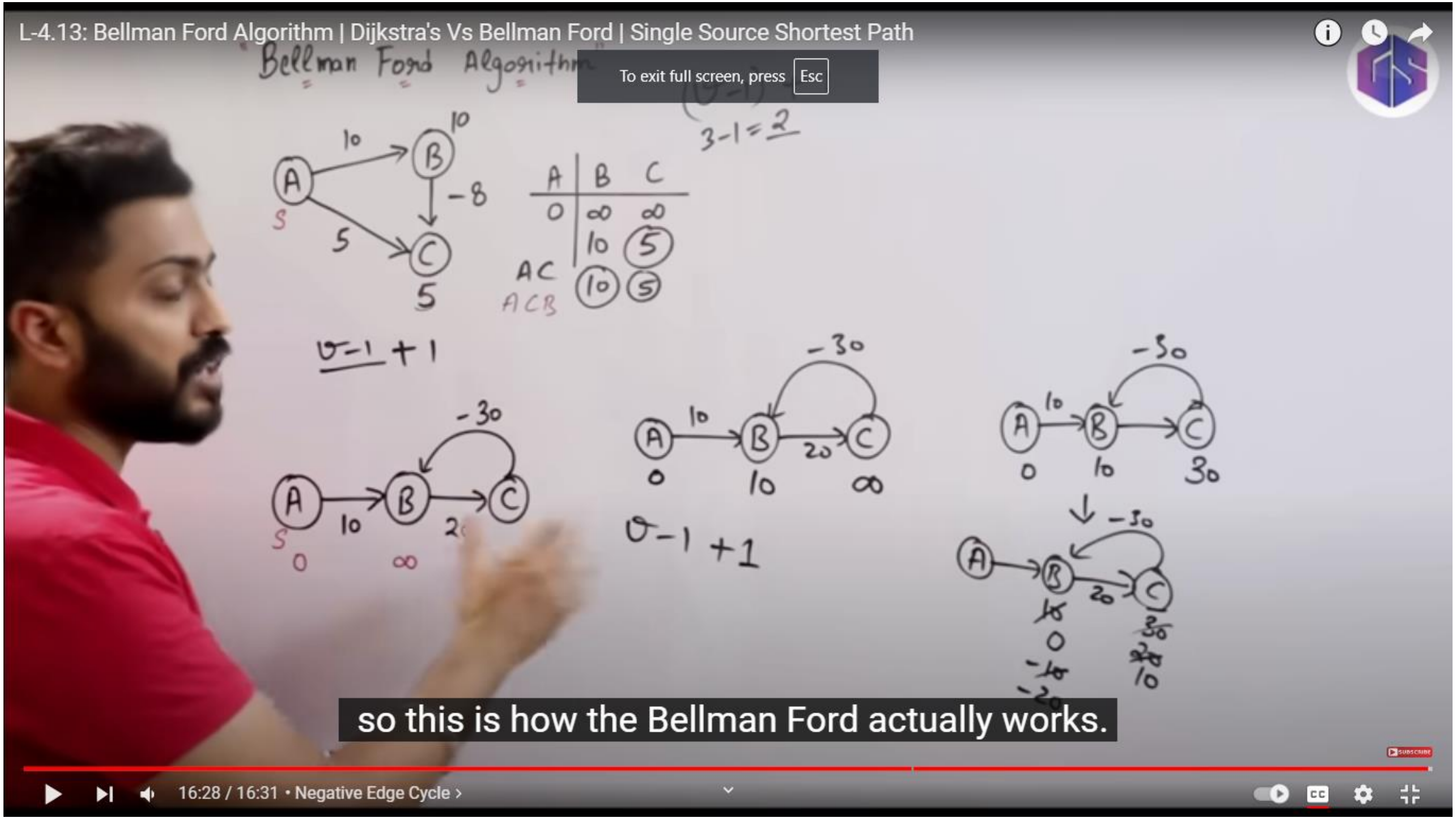
The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed a second time (The last row shows final values).

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1





L-4.13: Bellman Ford Algorithm | Dijkstra's Vs Bellman Ford | Single Source Shortest Path

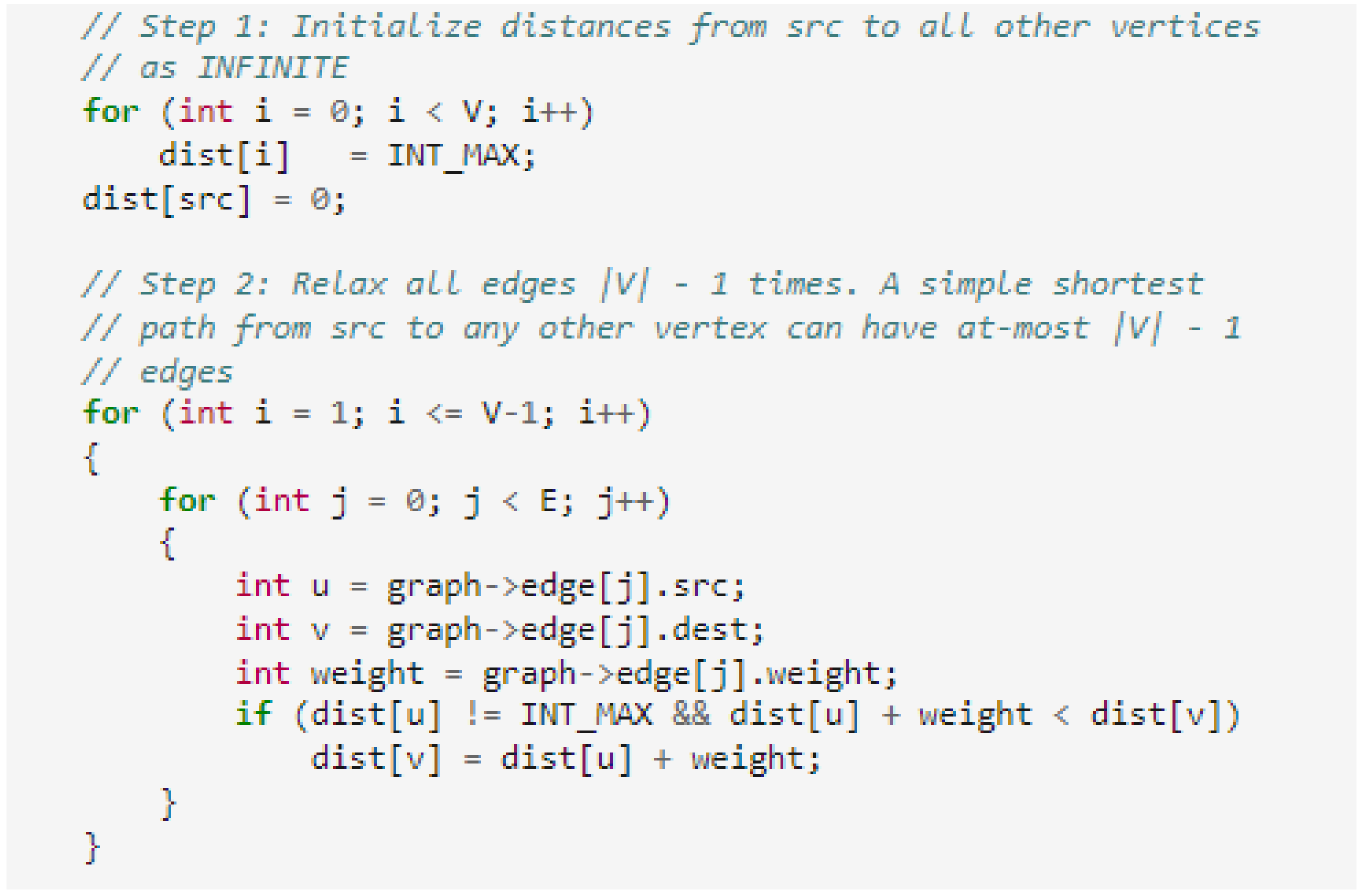
To exit full screen, press Esc

so this is how the Bellman Ford actually works.

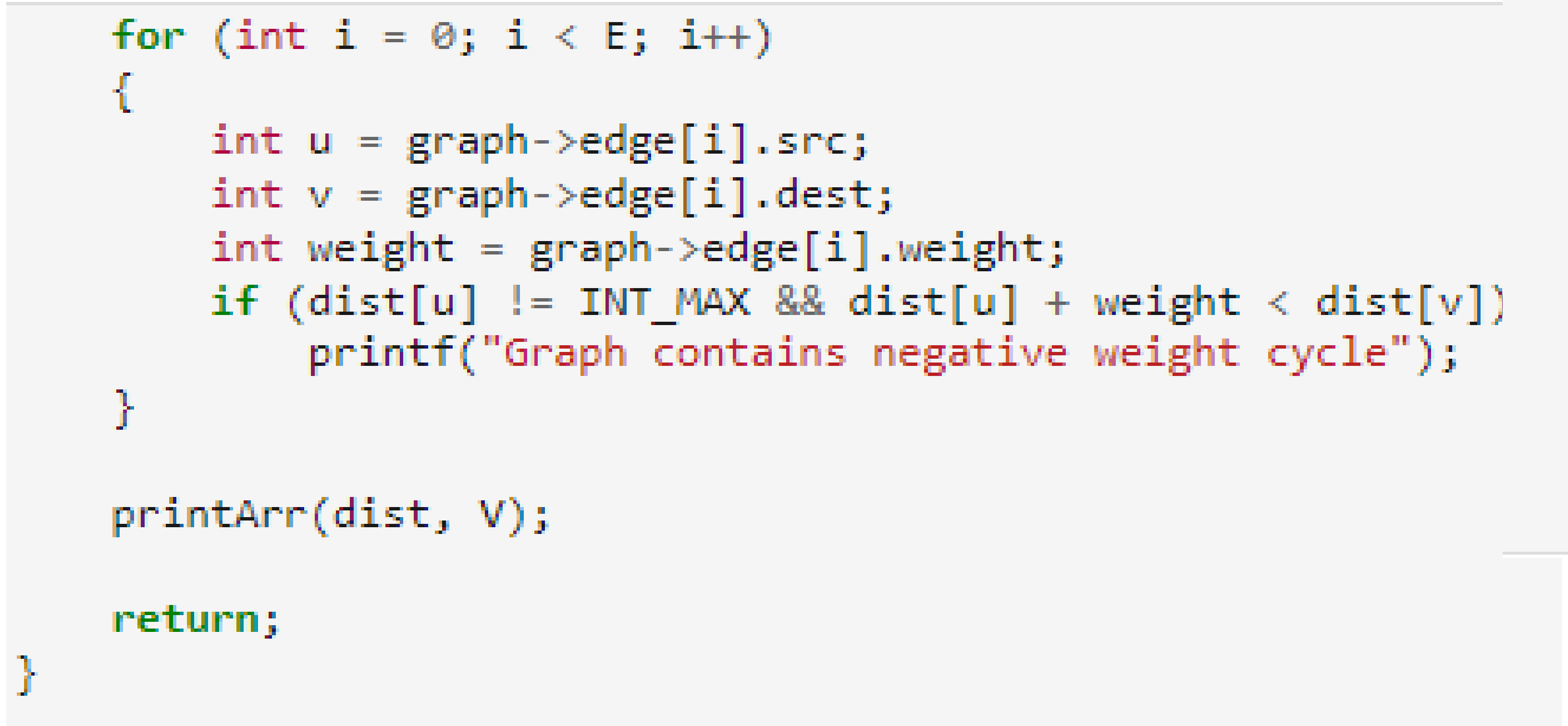
16:28 / 16:31 • Negative Edge Cycle ›

```cpp
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};
```

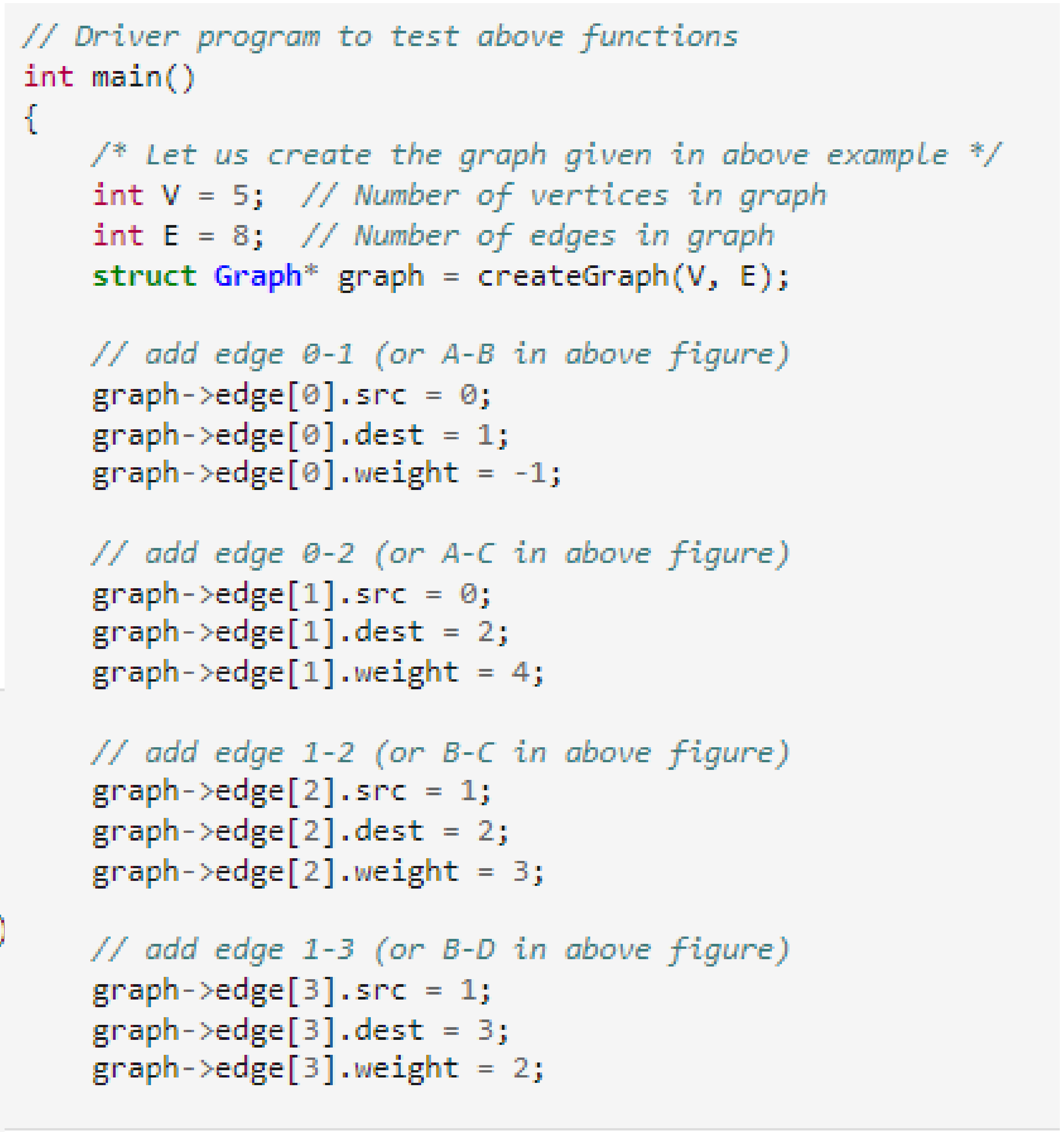
```cpp
// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm.  The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];
```



```
    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
```



```
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}
```



```
// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5;  // Number of vertices in graph
    int E = 8;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;
```

```
// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;
```

Output:

```
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1
```

Important Notes:

Negative weights are found in various applications of graphs. For example, instead of paying the cost for a path, we may get some advantage if we follow the path.

Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijksra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Floyd Warshall Algorithm

The **Floyd-Warshall algorithm**, named after its creators **Robert Floyd and Stephen Warshall**. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both **positive** and n**egative edge weights**, making it a versatile tool for solving a wide range of network and connectivity problems.

The **Floyd Warshall Algorithm** is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the **directed** and **undirected weighted** graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Floyd Warshall Algorithm Algorithm:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.
- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
  - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
  - **k** is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

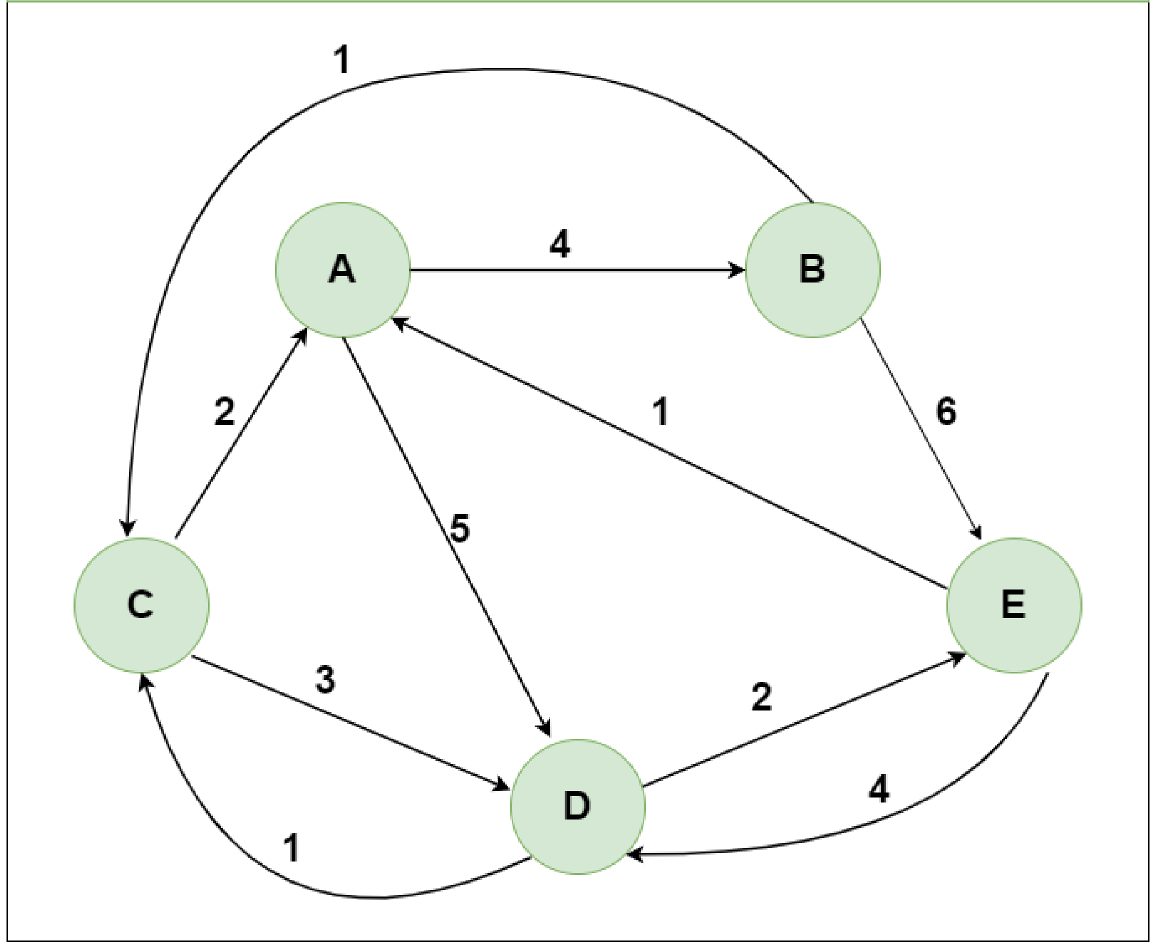
Pseudo-Code of Floyd Warshall Algorithm :

```
For k = 0 to n – 1
        For i = 0 to n – 1
                For j = 0 to n – 1
                        Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])
```

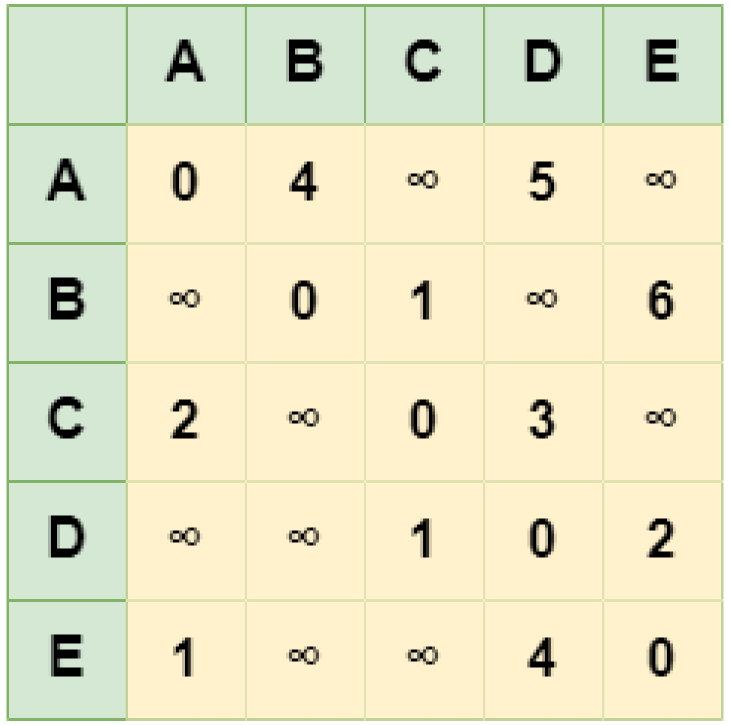
where i = source Node, j = Destination Node, k = Intermediate Node

<u>Step 1</u>: *Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from **i** to **j**, also Distance[i][j] = Infinity if there is no edge from **i** to **j.***

Example Graph



Step1: Initializing Distance[ ][ ] using the Input Graph

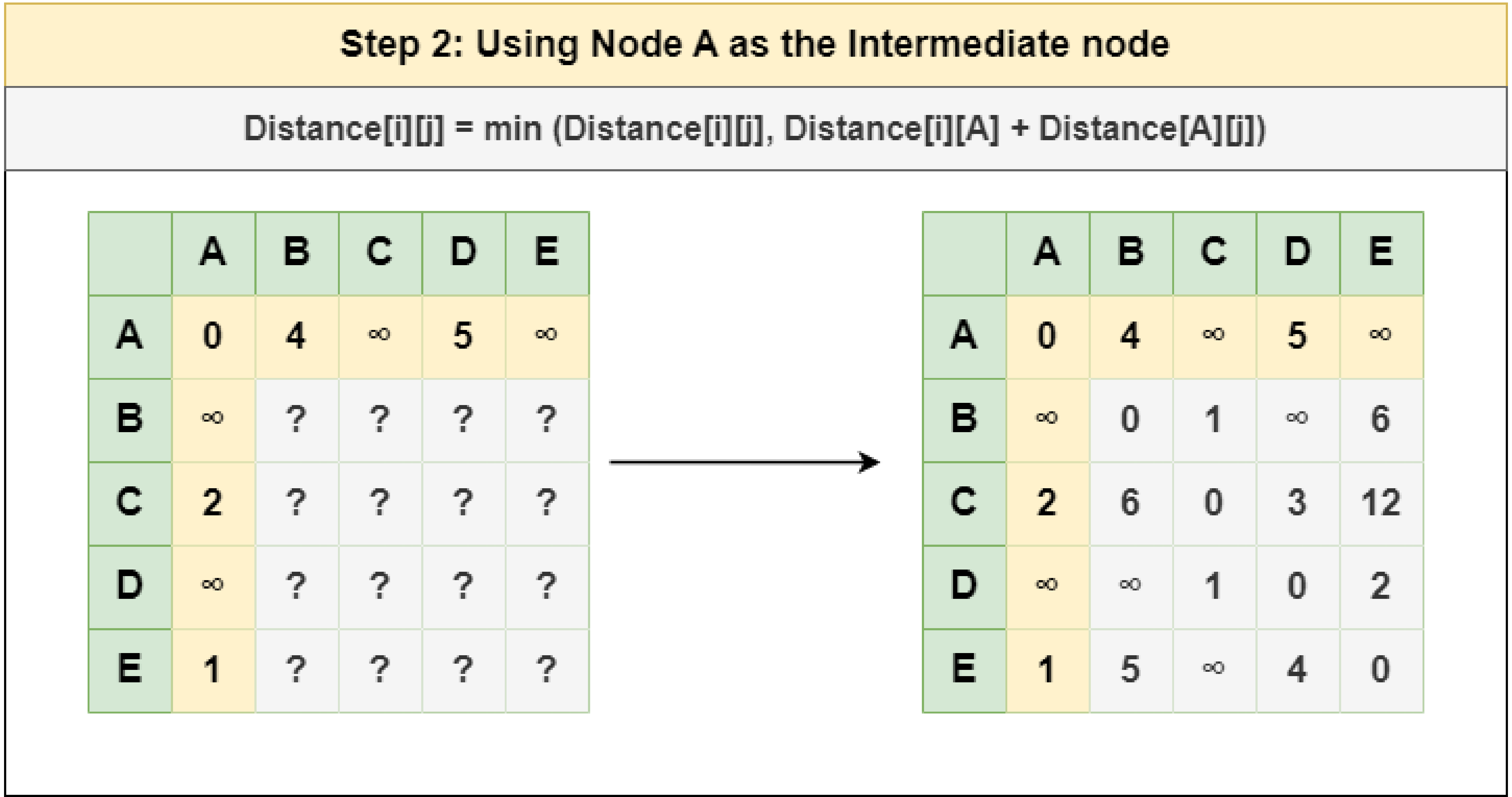


	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: *Treat node **A** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **A**) + (Distance from **A** to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**A**] + Distance[**A**][j])*



Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: *Treat node* ***B*** *as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to ***B****) + (Distance from* ***B*** *to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][****B****] + Distance[****B****][j])*

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

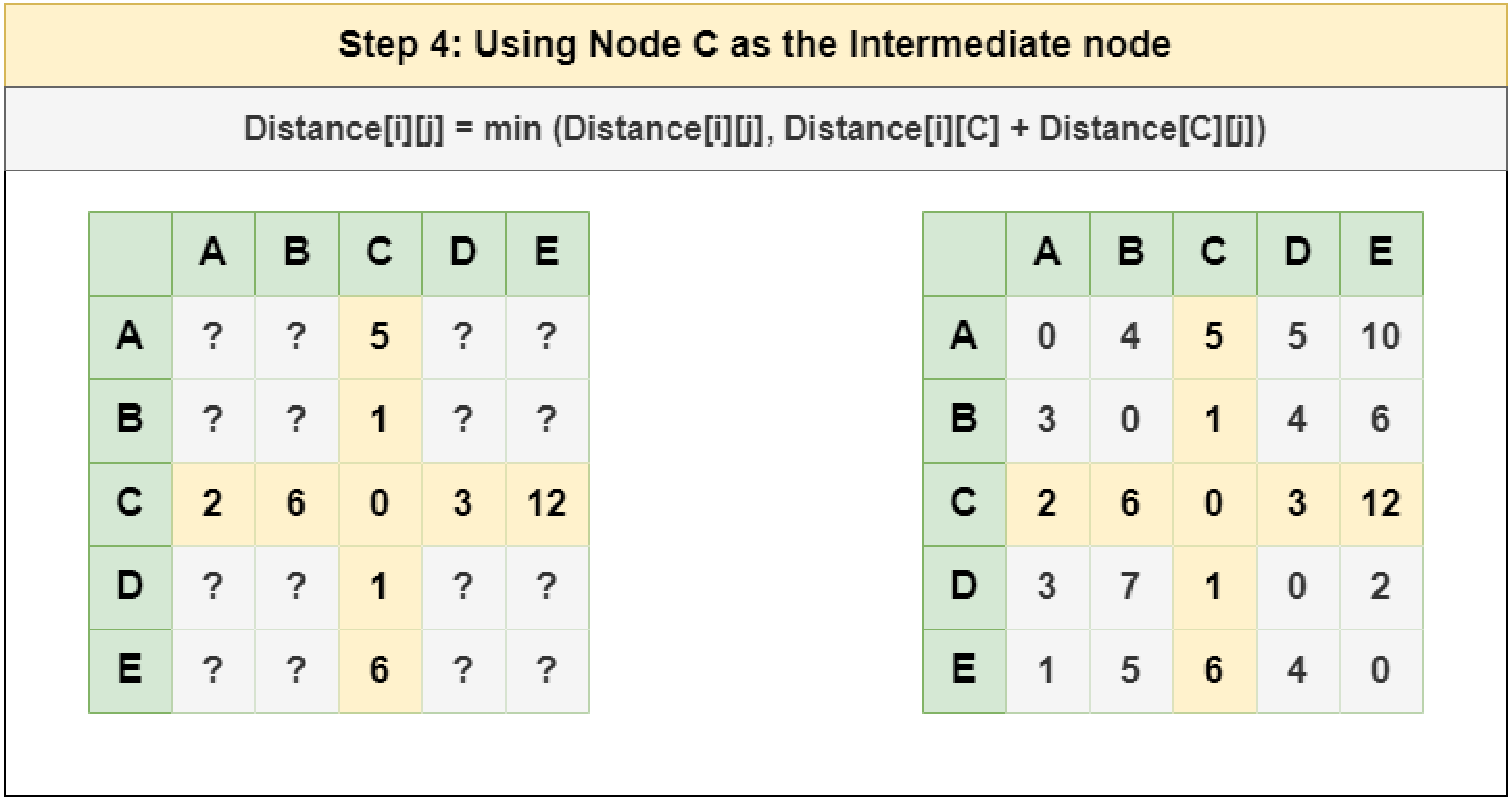
	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

***<u>Step 4</u>**: Treat node **C** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **C**) + (Distance from **C** to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**C**] + Distance[**C**][j])*



Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

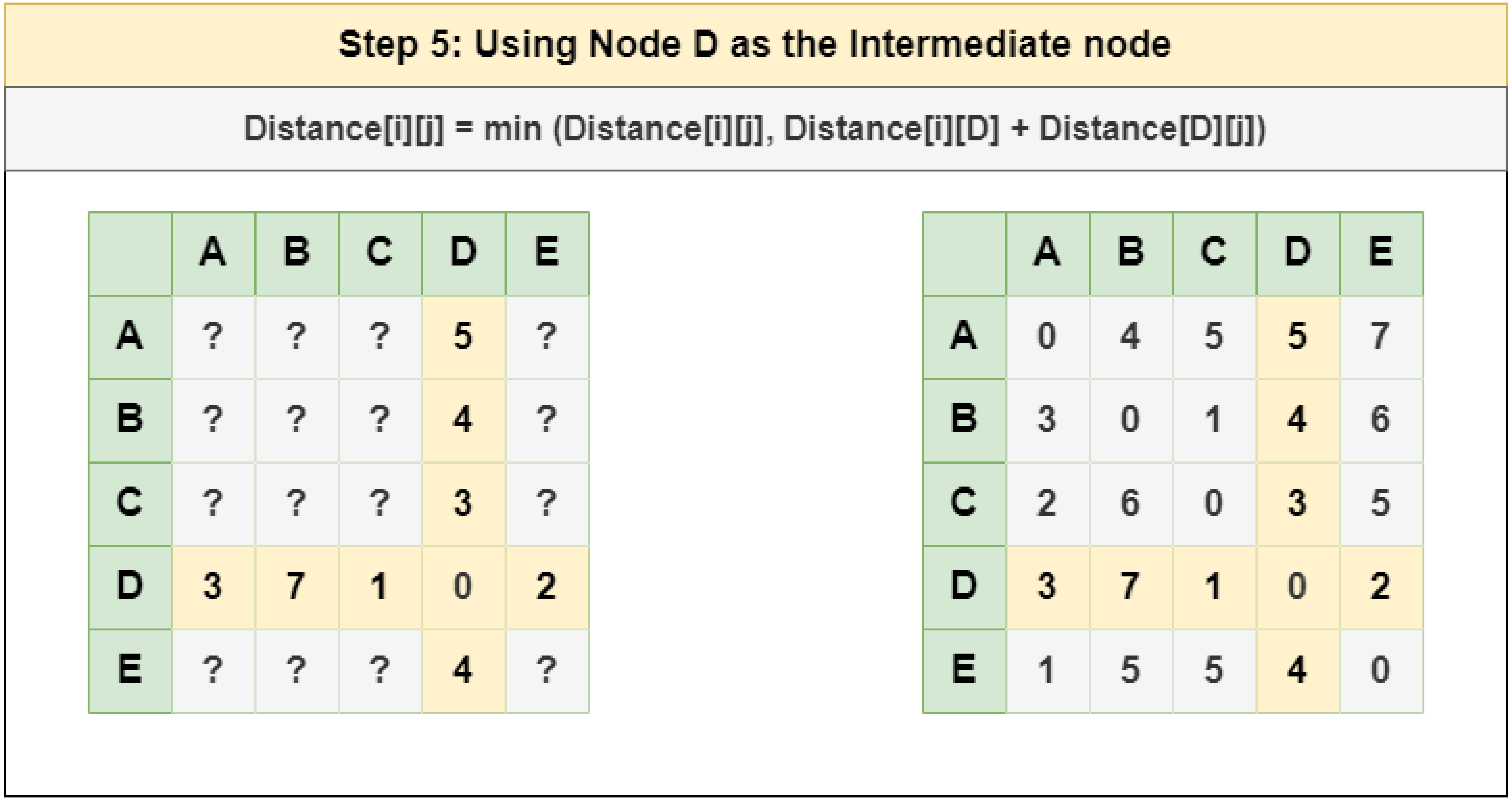
	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

***<u>Step 5</u>**: Treat node **D** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **D**) + (Distance from **D** to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])*



Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6: *Treat node* ***E*** *as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to ***E****) + (Distance from* ***E*** *to j))*

*= Distance[i][j] = minimum (Distance[i][j], Distance[i][****E****] + Distance[****E****][j])*

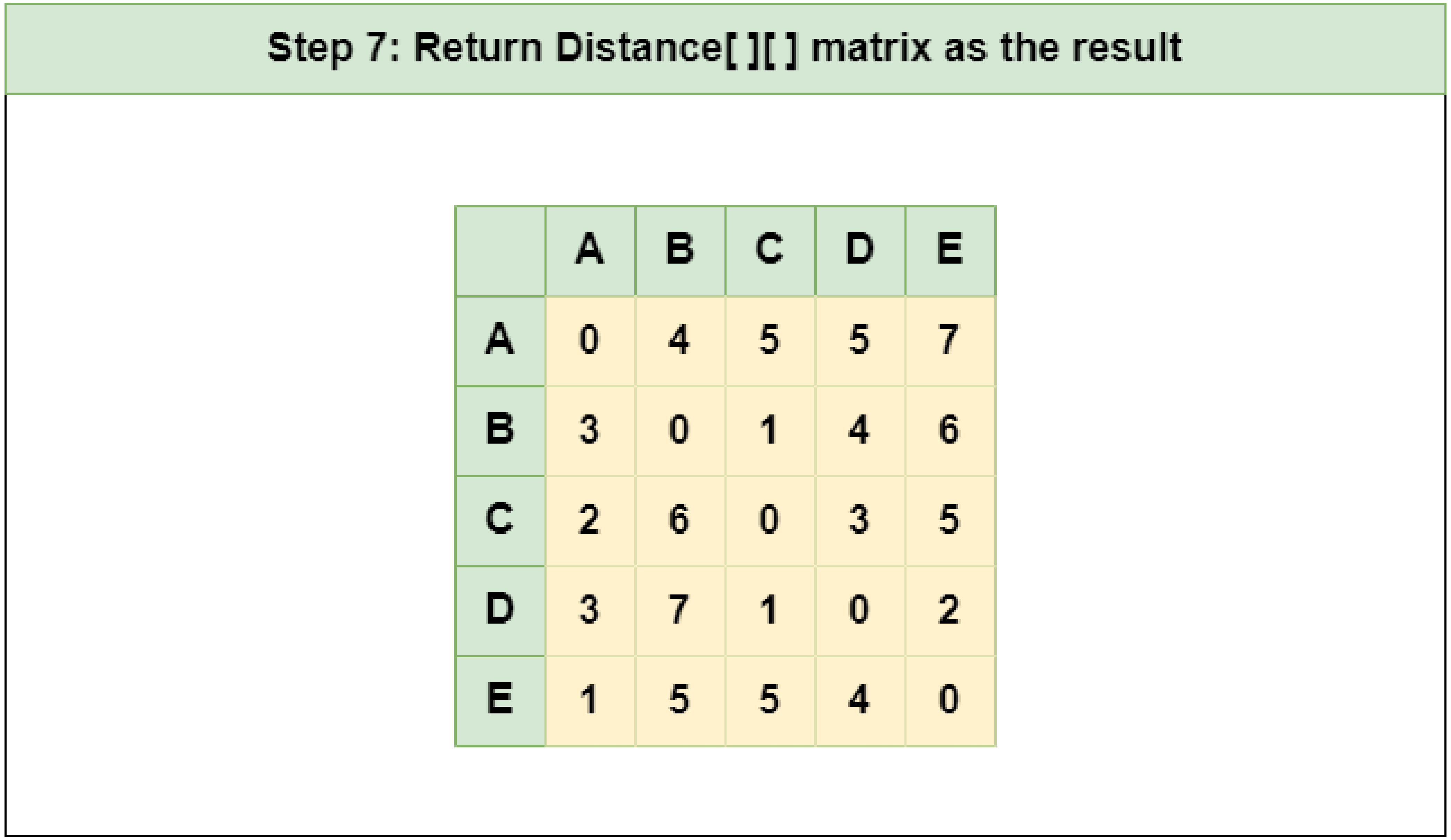
Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

***<u>Step 7</u>**: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.*



Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

```
// C++ Program for Floyd Warshall
Algorithm
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
value.This value will be used for
vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);
```

```
// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
  int i, j, k;
for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of
                // dist[i][j]
                if (dist[i][j] > (dist[i][k] + dist[k][j])
                    && (dist[k][j] != INF
                        && dist[i][k] != INF))
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
```

```
/* A utility function to print solution */
void printSolution(int dist[][V])
{
   cout << "The following matrix shows the
shortest "
         "distances"
         " between every pair of vertices \n";
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
         if (dist[i][j] == INF)
            cout << "INF"
                 << " ";
         else
            cout << dist[i][j] << "   ";
      }
      cout << endl;
   }
}
```

```
int main()
{
   /* Let us create the following weighted graph
           10
   (0)------->(3)
      |     /|\
   5 |      |
      |      | 1
   \|/      |
   (1)------->(2)
           3    */
   int graph[V][V] = { { 0, 5, INF, 10 },
                       { INF, 0, 3, INF },
                       { INF, INF, 0, 1 },
                       { INF, INF, INF, 0 } };

   // Function call
   floydWarshall(graph);
   return 0;
}
```

Graph Applications:

1. **Social Networks:**
   Graphs model relationships between entities. Social networks like Facebook and LinkedIn use graphs to represent connections between users.
2. **Networks and Routing Algorithms:**
   Graphs are used to model and analyze computer networks. Routing algorithms, such as Dijkstra's algorithm, operate on graphs to find the shortest path between nodes.
3. **Dependency Resolution:**
   Graphs are used to represent dependencies between tasks in project management or software builds. Dependency graphs help determine the order of execution.
4. **Web Page Linking:**
   The World Wide Web can be represented as a directed graph, where web pages are nodes, and hyperlinks are edges.
5. **Circuit Design:**
   Graphs model the connectivity of electronic circuits, helping engineers design and analyze complex systems.
6. **Recommendation Systems:**
   Collaborative filtering algorithms often use graphs to model the preferences and connections between users and items for personalized recommendations.
7. **Geographical Maps:**
   Graphs can represent roads, cities, and transportation networks, aiding in route planning and logistics.
8. **Game Development:**
   Graphs are used to represent game maps, character interactions, and decision trees in game development.

Conclusion and Key Takeaways

1 Tree Data Structures Are Everywhere

From your computer's file system to your bank's transaction history, trees are widely used to represent hierarchical data.

2 Choose the Right Tree for the Job

There are many types of trees, each with its own strengths and weaknesses. Selecting the right one can make a big difference in the efficiency and speed of your algorithm.

3 Think Recursively

The recursive nature of trees makes them powerful tools for solving complex problems. By thinking in terms of parent-child relationships, you can unlock the full potential of tree structure.

Thank you!