```
void Prims(struct Graph *G, int s) {
    struct PriorityQueue *PQ = CreatePriorityQueue();
    int v, w;
    EnQueue(PQ, s);
    Distance[s] = 0;                    // assume the Distance table is filled with -1
    while ((!IsEmptyQueue(PQ)) {
    v = DeleteMin(PQ);
        for all adjacent vertices w of v {
            Compute new distance d= Distance[v] + weight[v][w];
            if(Distance[w] == -1) {
                Distance[w] = weight[v][w];
                Insert w in the priority queue with priority d
                Path[w] = v;
            }
            if(Distance[w]  > new distance d) {
                Distance[w] = weight[v][w];
                Update priority of vertex w to be d;
                Path[w] = v;
            }
        }
    }
}
```

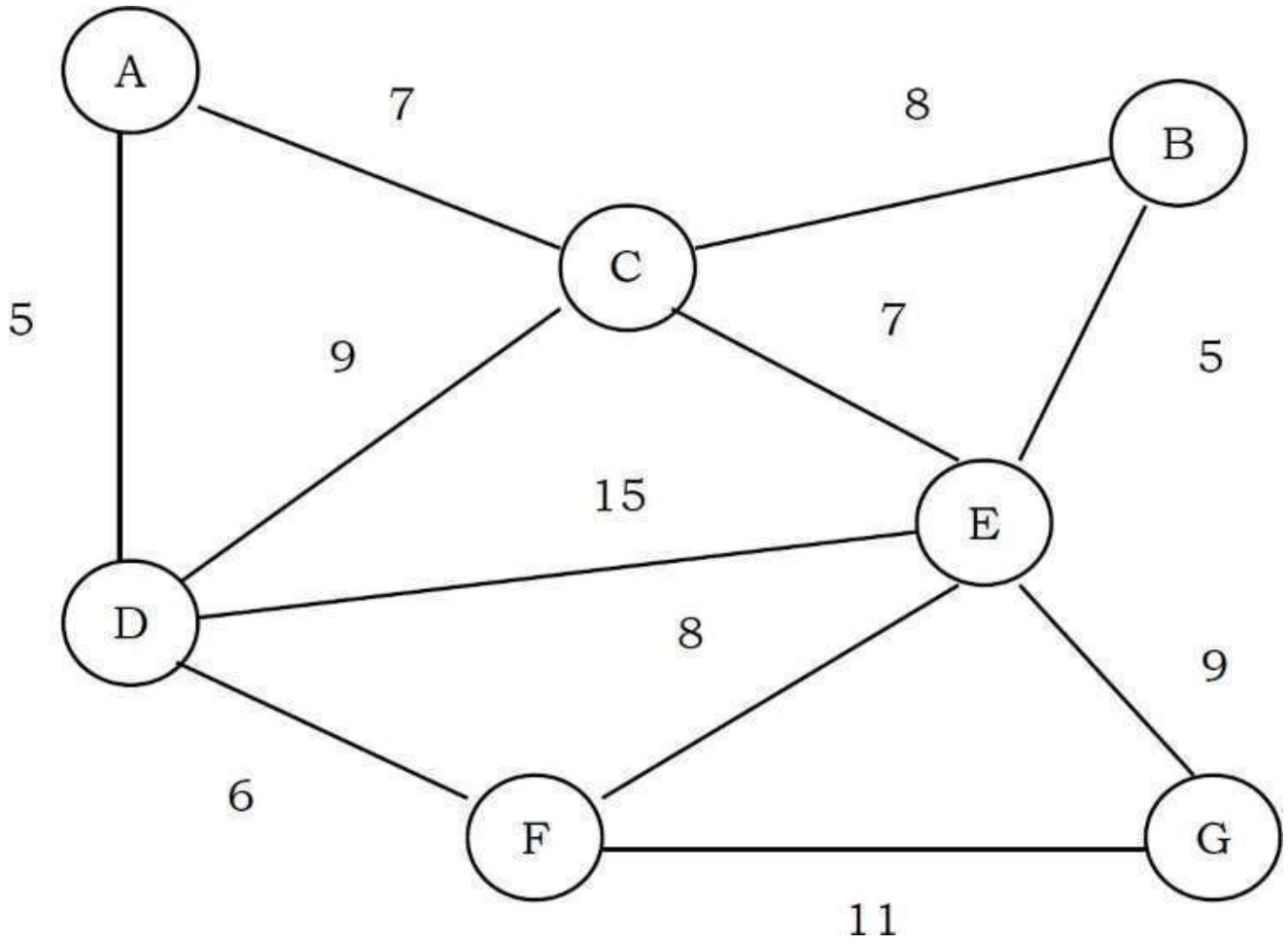
The entire implementation of this algorithm is identical to that of Dijkstra's algorithm. The running time is $O(|V|^2)$ without heaps [good for dense graphs], and O (*ElogV*) using binary heaps [good for sparse graphs].

Kruskal's Algorithm

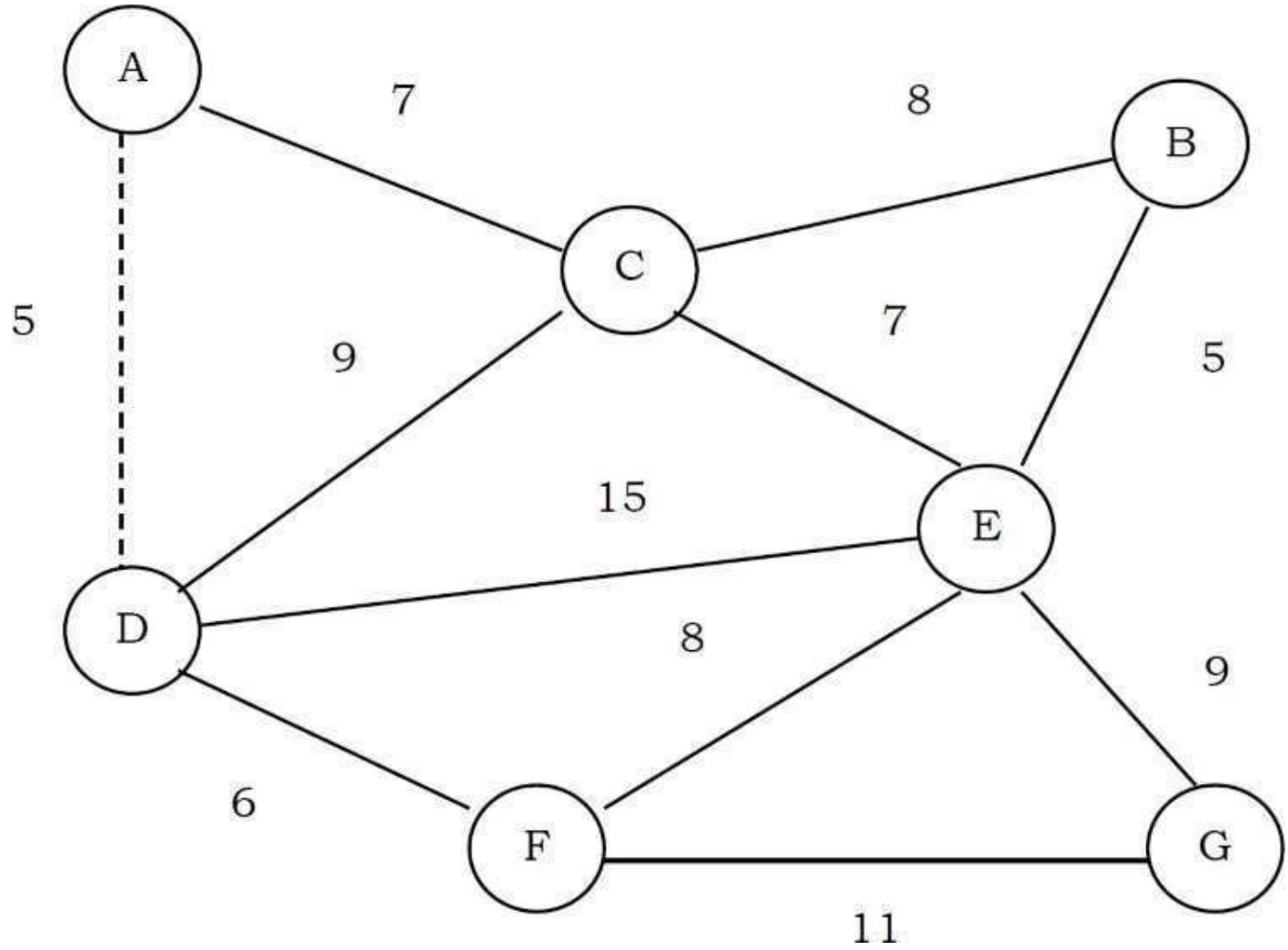
The algorithm starts with V different trees (V is the vertices in the graph). While constructing the minimum spanning tree, every time Kruskal's alorithm selects an edge that has minimum weight and then adds that edge if it doesn't create a cycle. So, initially, there are | V | single-node trees in the forest. Adding an edge merges two trees into one. When the algorithm is completed, there will be only one tree, and that is the minimum spanning tree. There are two ways of implementing Kruskal's algorithm:

- By using Disjoint Sets: Using UNION and FIND operations
- By using Priority Queues: Maintains weights in priority queue

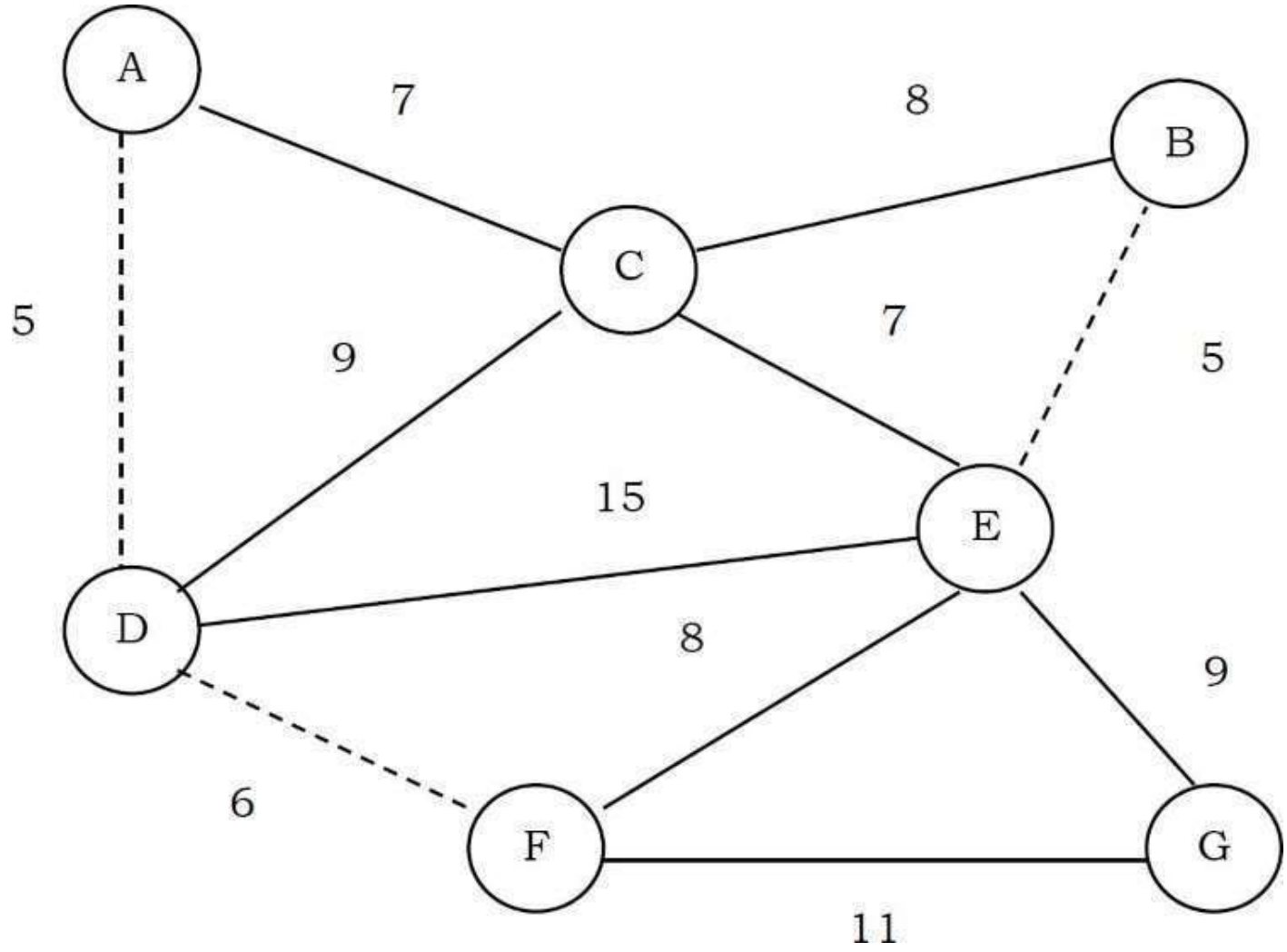
The appropriate data structure is the UNION/FIND algorithm [for implementing forests]. Two vertices belong to the same set if and only if they are connected in the current spanning forest. Each vertex is initially in its own set. If *u* and *v* are in the same set, the edge is rejected because it forms a cycle. Otherwise, the edge is accepted, and a UNION is performed on the two sets containing *u* and *v*. As an example, consider the following graph (the edges show the weights).



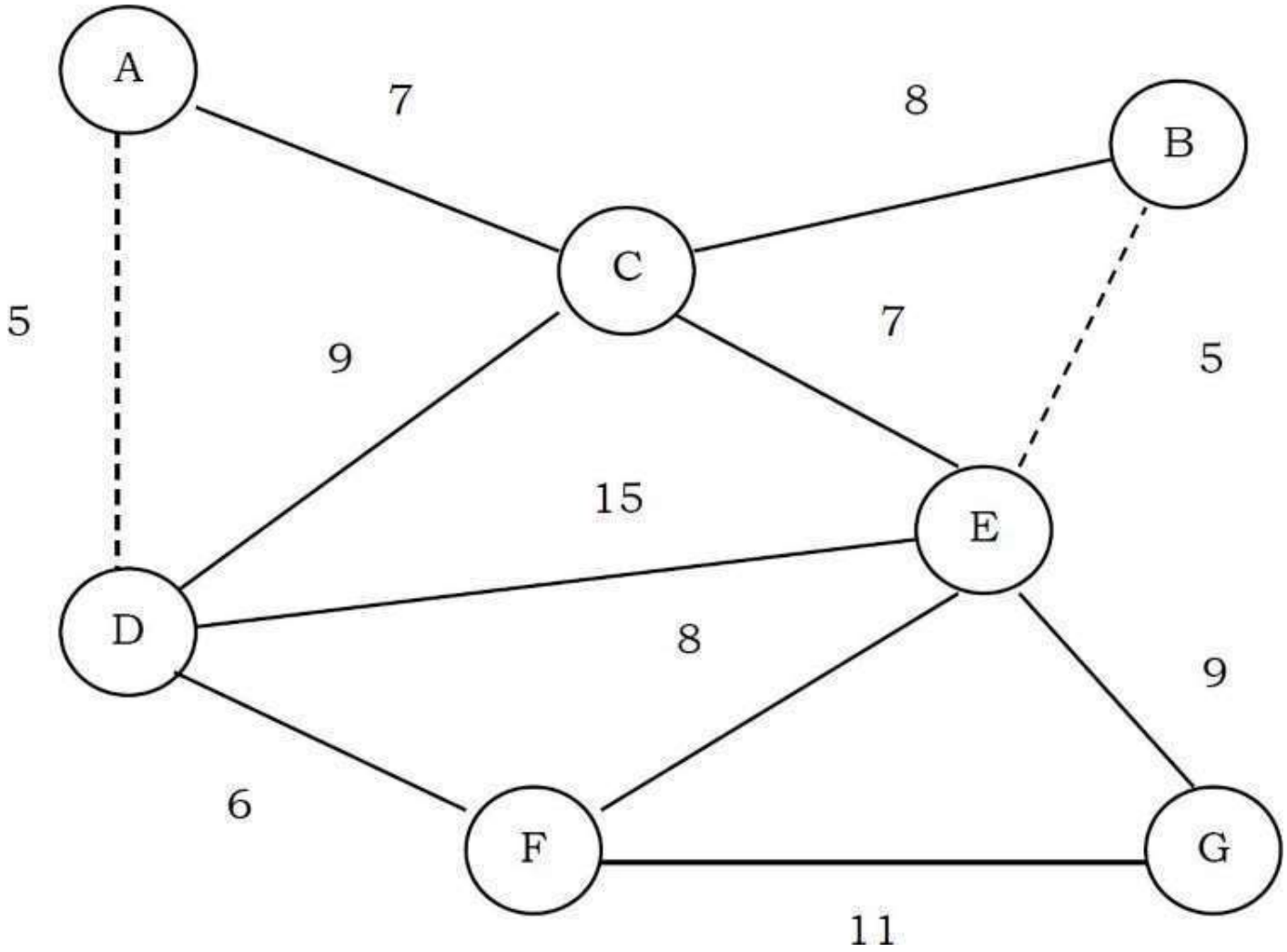
Now let us perform Kruskal's algorithm on this graph. We always select the edge which has minimum weight.



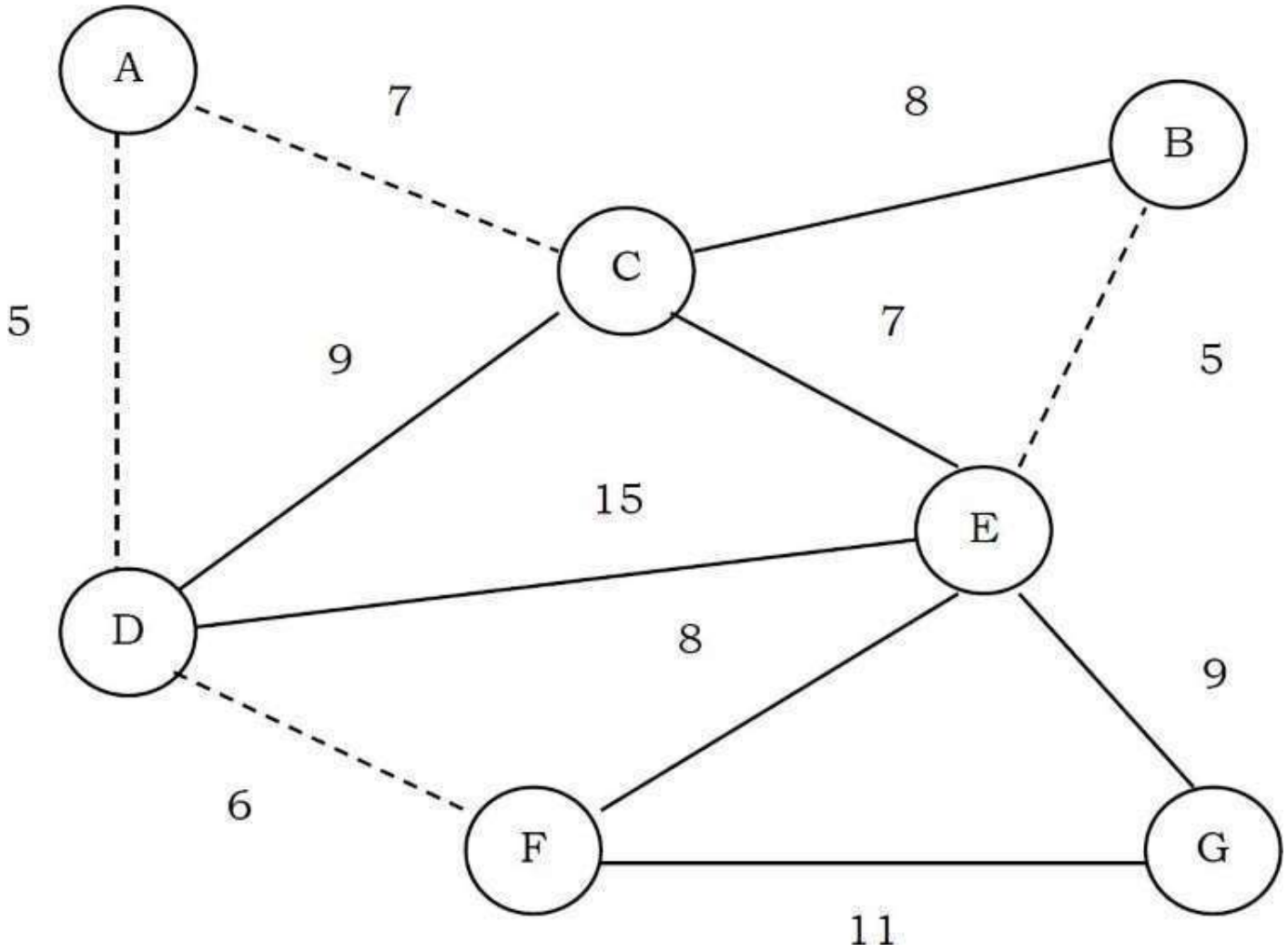
From the above graph, the edges which have minimum weight (cost) are: AD and BE. From these two we can select one of them and let us assume that we select AD (dotted line).



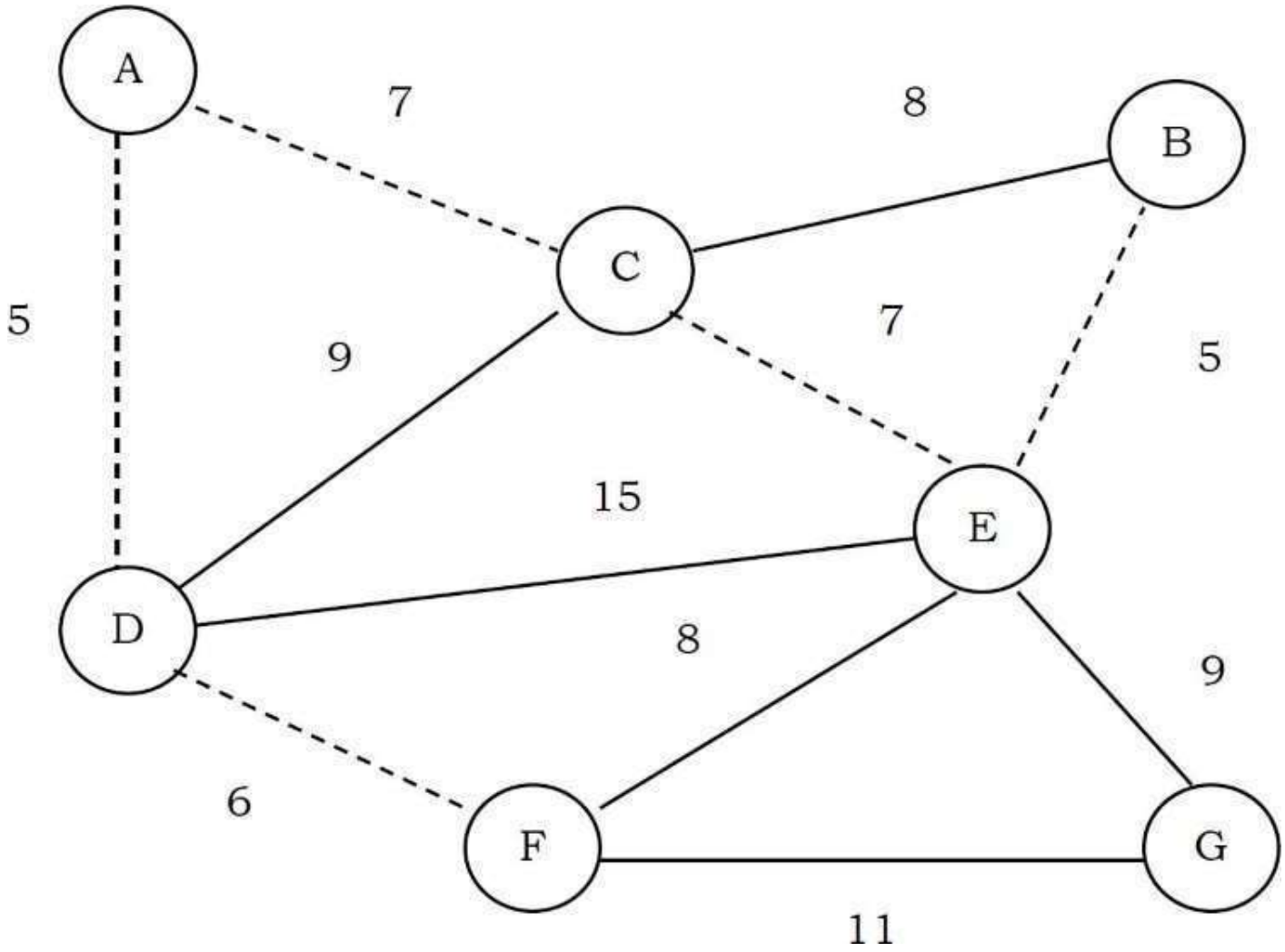
DF is the next edge that has the lowest cost (6).



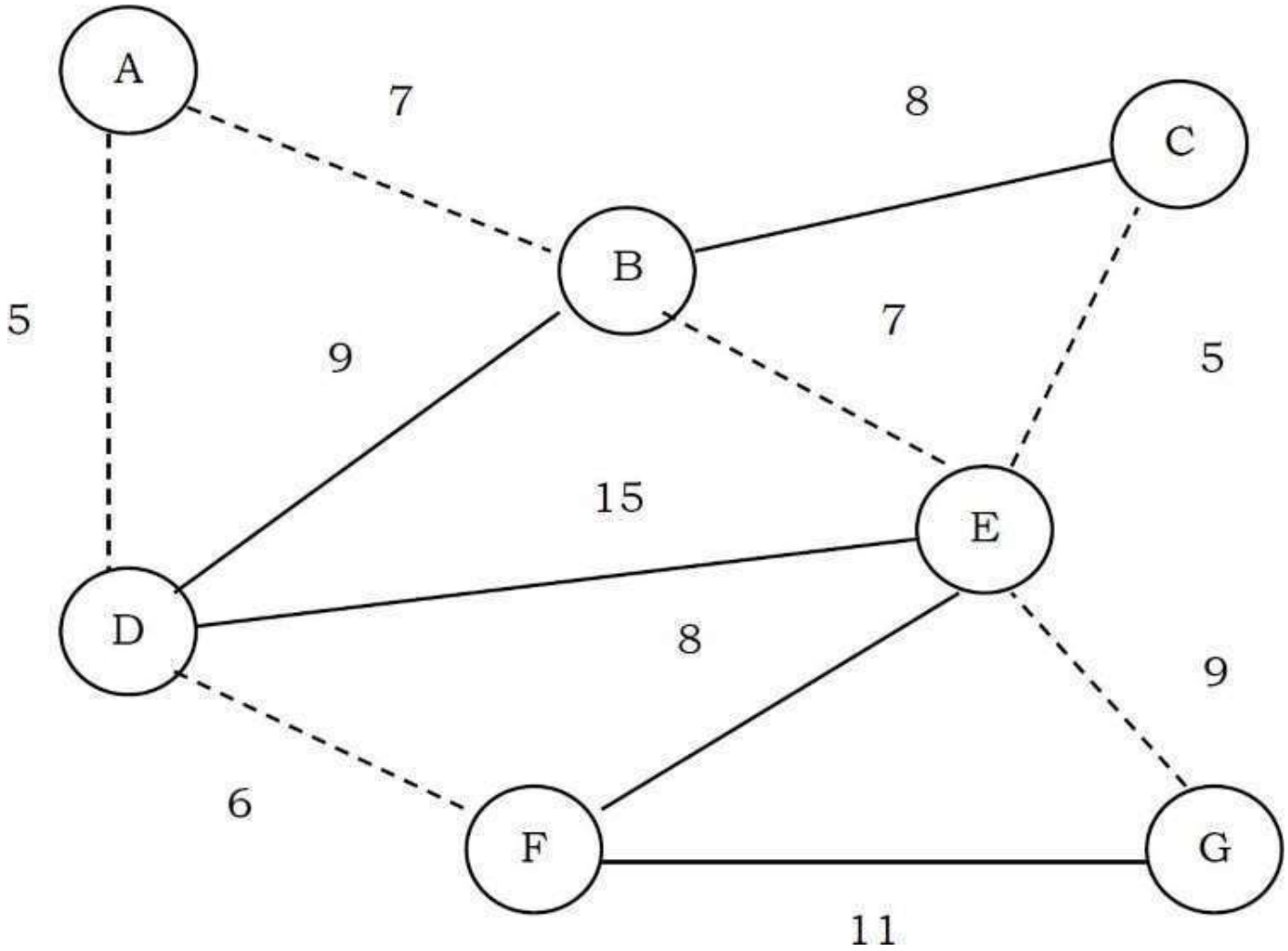
BE now has the lowest cost and we select it (dotted lines indicate selected edges).



Next, AC and CE have the low cost of 7 and we select AC.



Then we select CE as its cost is 7 and it does not form a cycle.



The next low cost edges are CB and EF. But if we select CB, then it forms a cycle. So we discard it. This is also the case with EF. So we should not select those two. And the next low cost is 9 (BD and EG). Selecting BD forms a cycle so we discard it. Adding EG will not form a cycle and therefore with this edge we complete all vertices of the graph.

```
void Kruskal(struct Graph *G) {
    S = ϕ;   // At the end S will contains the edges of minimum spanning trees
    for (int v = 0; v< G→V; v++)
        MakeSet (v);
    Sort edges of E by increasing weights w;
    for each edge (u, v) in E { //from sorted list
        if(FIND (u) ≠ FIND (v)) {
            S = S ∪ {(u, v)};
            UNION (u, v);
        }
    }
    return S;
}
```

Note: For implementation of UNION and FIND operations, refer to the *Disjoint Sets ADT* chapter.

The worst-case running time of this algorithm is O(*ElogE*), which is dominated by the heap operations. That means, since we are constructing the heap with E edges, we need O(*ElogE*) time to do that.

9.9 Graph Algorithms: Problems & Solutions

Problem-1 In an undirected simple graph with *n* vertices, what is the maximum number of edges? Self-loops are not allowed.

Solution: Since every node can connect to all other nodes, the first node can connect to $n - 1$ nodes. The second node can connect to $n - 2$ nodes [since one edge is already there from the first node]. The total number of edges is: $1 + 2 + 3 + \cdots + n - = \frac{n(n-1)}{2}$ edges.

Problem-2 How many different adjacency matrices does a graph with *n* vertices and *E* edges have?

Solution: It's equal to the number of permutations of *n* elements, i.e., *n*!.

Problem-3 How many different adjacency lists does a graph with *n* vertices have?

Solution: It's equal to the number of permutations of edges, i.e., *E*!.

Problem-4 Which undirected graph representation is most appropriate for determining whether or not a vertex is isolated (is not connected to any other vertex)?

Solution: Adjacency List. If we use the adjacency matrix, then we need to check the complete row to determine whether that vertex has edges or not. By using the adjacency list, it is very easy to check, and it can be done just by checking whether that vertex has NULL for next pointer or not [NULL indicates that the vertex is not connected to any other vertex].

Problem-5 For checking whether there is a path from source s to target *t*, which one is best between disjoint sets and DFS?

Solution: The table below shows the comparison between disjoint sets and DFS. The entries in the table represent the case for any pair of nodes (for *s* and *t*).

Method	Processing Time	Query Time	Space
Union-Find	$V + E \log V$	$\log V$	V
DFS	$E + V$	1	$E + V$

Problem-6 What is the maximum number of edges a directed graph with n vertices can have and still not contain a directed cycle?

Solution: The number is V (V – 1)/2. Any directed graph can have at most n^2 edges. However, since the graph has no cycles it cannot contain a self loop, and for any pair *x*,*y* of vertices, at most one edge from (*x*,*y*) and (*y*,*x*) can be included. Therefore the number of edges can be at most (V^2 – V)/2 as desired. It is possible to achieve V(V – 1)/2 edges. Label *n* nodes 1,2... *n* and add an edge (x, y) if and only if $x < y$. This graph has the appropriate number of edges and cannot contain a cycle (any path visits an increasing sequence of nodes).

Problem-7 How many simple directed graphs with no parallel edges and self-loops are possible in terms of *V*?

Solution: (V) × (V – 1). Since, each vertex can connect to V – 1 vertices without self-loops.

Problem-8 What are the differences between DFS and BFS?

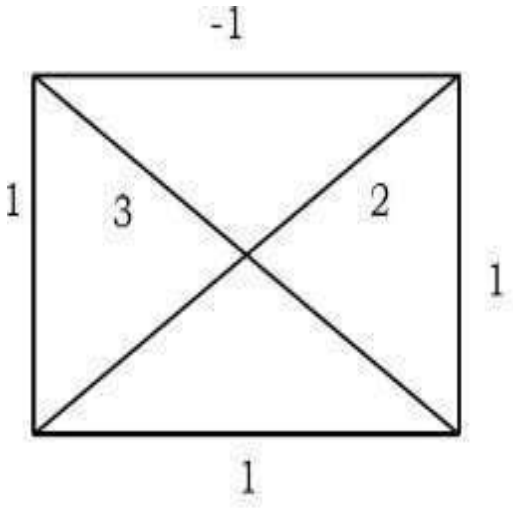
Solution:

DFS	BFS
Backtracking is possible from a dead end.	Backtracking is not possible.
Vertices from which exploration is incomplete are processed in a LIFO order	The vertices to be explored are organized as a FIFO queue.
The search is done in one particular direction	The vertices at the same level are maintained in parallel.

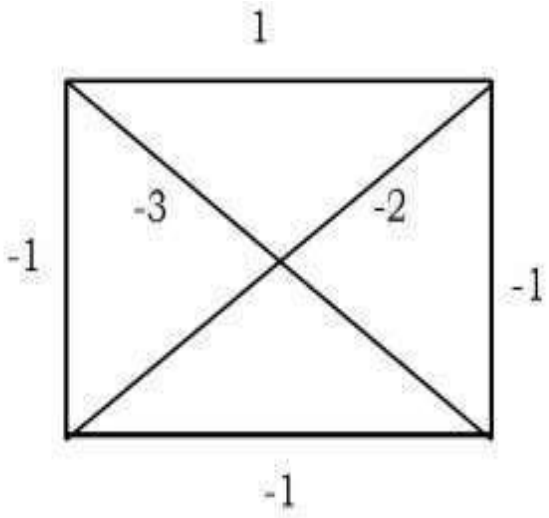
Problem-9 Earlier in this chapter, we discussed minimum spanning tree algorithms. Now,

give an algorithm for finding the maximum-weight spanning tree in a graph.

Solution:



Given graph



Transformed graph with negative edge weights

Using the given graph, construct a new graph with the same nodes and edges. But instead of using the same weights, take the negative of their weights. That means, weight of an edge = negative of weight of the corresponding edge in the given graph. Now, we can use existing *minimum spanning tree* algorithms on this new graph. As a result, we will get the maximum-weight spanning tree in the original one.

Problem-10 Give an algorithm for checking whether a given graph *G* has simple path from source s to destination *d*. Assume the graph *G* is represented using the adjacent matrix.

Solution: Let us assume that the structure for the graph is:

```
struct Graph {
    int V;                  //Number of vertices
    int E;                  //Number of edges
    int ** adjMatrix; //Two dimensional array for storing the connections
};
```

For each vertex call *DFS* and check whether the current vertex is the same as the destination vertex or not. If they are the same, then return 1. Otherwise, call the *DFS* on its unvisited neighbors. One important thing to note here is that, we are calling the DFS algorithm on vertices which are not yet visited.

```
void HasSimplePath(struct Graph *G, int s, int d) {
    int t;
    Viisited[s] = 1;
    if(s == d)
        return 1;
    for(t = 0; t < G→V; t++) {
        if(G→adjMatrix[s][t] && !Viisited[t])
            if(DFS(G, t, d))
                return 1;
    }
    return 0;
}
```

Time Complexity: O(*E*). In the above algorithm, for each node, since we are not calling *DFS* on all of its neighbors (discarding through *if* condition), Space Complexity: O(*V*).

Problem-11 Count simple paths for a given graph *G* has simple path from source s to destination d? Assume the graph is represented using the adjacent matrix.

Solution: Similar to the discussion in Problem-10, start at one node and call DFS on that node. As a result of this call, it visits all the nodes that it can reach in the given graph. That means it visits all the nodes of the connected component of that node. If there are any nodes that have not been visited, then again start at one of those nodes and call DFS.

Before the first DFS in each connected component, increment the connected components *count*. Continue this process until all of the graph nodes are visited. As a result, at the end we will get the total number of connected components. The implementation based on this logic is given below:

```
void CountSimplePaths(struct Graph * G, int s, int d) {
    int t;
    Viisited[s] = 1;
    if(s == d) {
        count++;
        Visited[s] = 0;
        return;
    }
    for(t = 0; t < G→V; t++) {
        if(G→adjMatrix[s][t] && !Viisited[t]) {
            DFS(G, t, d);
            Visited[t] = 0;
        }
    }
}
```

Problem-12 **All pairs shortest path problem:** Find the shortest graph distances between every pair of vertices in a given graph. Let us assume that the given graph does not have negative edges.

Solution: The problem can be solved using *n* applications of *Dijkstra's* algorithm. That means we apply *Dijkstra's* algorithm on each vertex of the given graph. This algorithm does not work if the graph has edges with negative weights.

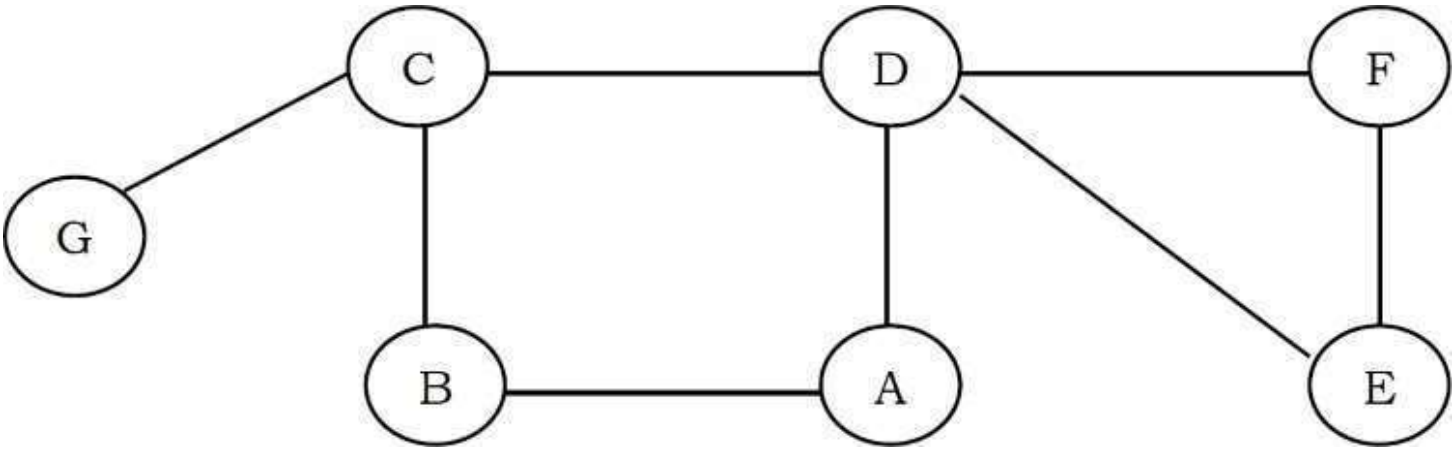
Problem-13 In Problem-12, how do we solve the all pairs shortest path problem if the graph has edges with negative weights?

Solution: This can be solved by using the *Floyd – Warshall algorithm*. This algorithm also works in the case of a weighted graph where the edges have negative weights. This algorithm is an example of Dynamic Programming -refer to the *Dynamic Programming* chapter.

Problem-14 **DFS Application:** *Cut Vertex* or *Articulation Points*

Solution: In an undirected graph, a *cut vertex* (or articulation point) is a vertex, and if we remove it, then the graph splits into two disconnected components. As an example, consider the following figure. Removal of the "*D*" vertex divides the graph into two connected components ({*E,F*} and {*A,B, C, G*}).

Similarly, removal of the "*C*" vertex divides the graph into ({*G*} and {*A, B,D,E,F*}). For this graph, *A* and *C* are the cut vertices.

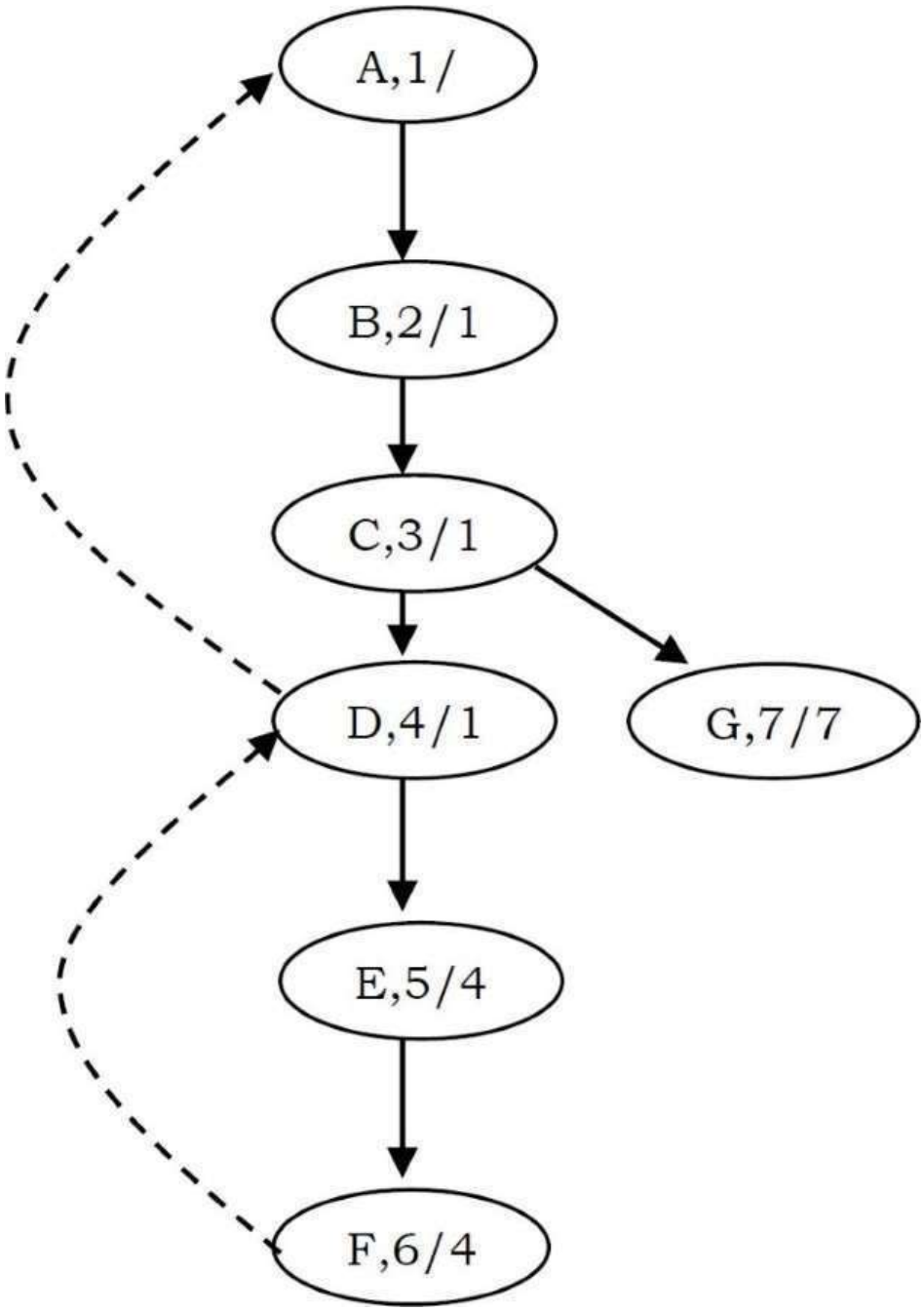


Note: A connected, undirected graph is called *bi – connected* if the graph is still connected after removing any vertex.

DFS provides a linear-time algorithm (O(*n*)) to find all cut vertices in a connected graph. Starting at any vertex, call a *DFS* and number the nodes as they are visited. For each vertex *v*, we call this DFS number *dfsnum*(v). The tree generated with DFS traversal is called *DFS spanning tree*. Then, for every vertex *v* in the *DFS* spanning tree, we compute the lowest-numbered vertex, which we call *low*(*v*), that is reachable from *v* by taking zero or more tree edges and then possibly one back edge (in that order).

Based on the above discussion, we need the following information for this algorithm: the *dfsnum* of each vertex in the *DFS* tree (once it gets visited), and for each vertex *v*, the lowest depth of neighbors of all descendants of *v* in the *DFS* tree, called the *low*.

The *dfsnum* can be computed during DFS. The low of *v* can be computed after visiting all descendants of *v* (i.e., just before *v* gets popped off the *DFS* stack) as the minimum of the *dfsnum* of all neighbors of *v* (other than the parent of *v* in the *DFS* tree) and the *low* of all children of *v* in the *DFS* tree.



The root vertex is a cut vertex if and only if it has at least two children. A non-root vertex u is a cut vertex if and only if there is a son *v* of *u* such that *low*(*v*) ≥ *dfsnum*(*u*). This property can be tested once the *DFS* is returned from every child of *u* (that means, just before u gets popped off the DFS stack), and if true, *u* separates the graph into different bi-connected components. This can be represented by computing one bi-connected component out of every such *v* (a component which contains *v* will contain the sub-tree of *v*, plus *u*), and then erasing the sub-tree of *v* from the tree.

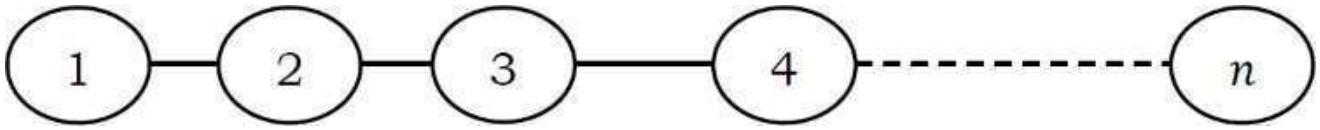
For the given graph, the *DFS* tree with *dfsnum/low* can be given as shown in the figure below. The implementation for the above discussion is:

```
int adjMatrix [256] [256] ;
int dfsnum [256], num = 0, low [256];
void CutVertices( int u ) {
      low[u] = dfsnum[u] = num++;

      for (int v = 0 ; v < 256; ++v ) {
            if(adjMatrix[u][v] && dfsnum[v] == -1) {
                  CutVertices( v ) ;
                  if(low[v] > dfsnum[u])
                        printf("Cut Vetex:%d",u);
                  low[u] = min ( low[u] , low[v] ) ;
            }
            else     // (u,v) is a back edge
                  low[u ] = min(low[u] , dfsnum[v]) ;
      }
}
```

Problem-15 Let G be a connected graph of order n. What is the maximum number of cut-vertices that G can contain?

Solution: $n - 2$. As an example, consider the following graph. In the graph below, except for the vertices 1 and n, all the remaining vertices are cut vertices. This is because removing 1 and n vertices does not split the graph into two. This is a case where we can get the maximum number of cut vertices.

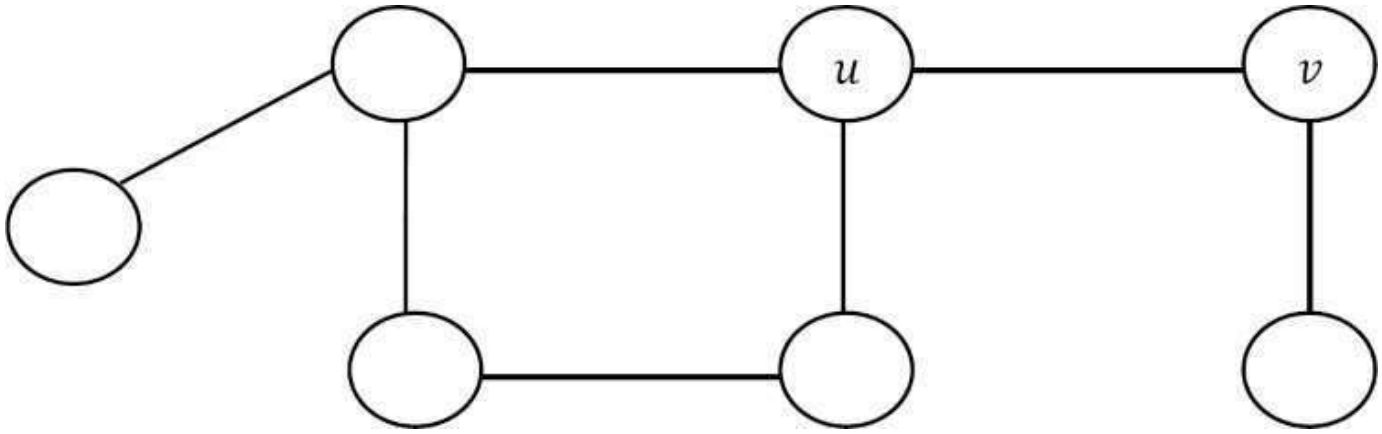


Problem-16 **DFS Application:** *Cut Bridges* or *Cut Edges*

Solution:

Definition: Let G be a connected graph. An edge uv in G is called a *bridge* of G if $G - uv$ is disconnected.

As an example, consider the following graph.



In the above graph, if we remove the edge *uv* then the graph splits into two components. For this graph, *uv* is a bridge. The discussion we had for cut vertices holds good for bridges also. The only change is, instead of printing the vertex, we give the edge. The main observation is that an edge (*u*, *v*) cannot be a bridge if it is part of a cycle. If (*u*, *v*) is not part of a cycle, then it is a bridge.

We can detect cycles in *DFS* by the presence of back edges, (*u*, *v*) is a bridge if and only if none of v or *v's* children has a back edge to *u* or any of *u's* ancestors. To detect whether any of *v's* children has a back edge to *u's* parent, we can use a similar idea as above to see what is the smallest *dfsnum* reachable from the subtree rooted at *v*.

```
int dfsnum[256], num = 0, low [256];
void Bridges( struct Graph *G, int u ) {
    low[u] = dfsnum[u] = num++;

    for (int v = 0 ; G→V; ++v ) {
        if(G→adjMatrix[u][v] && dfsnum[v] == -1) {
            cutVertices( v ) ;

            if(low[v] > dfsnum[u])
                print (u,v)  as a bridge

            low[u] = min ( low[u] , low[v] ) ;
        }
        else    // (u,v) is a back edge
            low[u ] = min(low[u] , dfsnum[v]) ;
    }
}
```

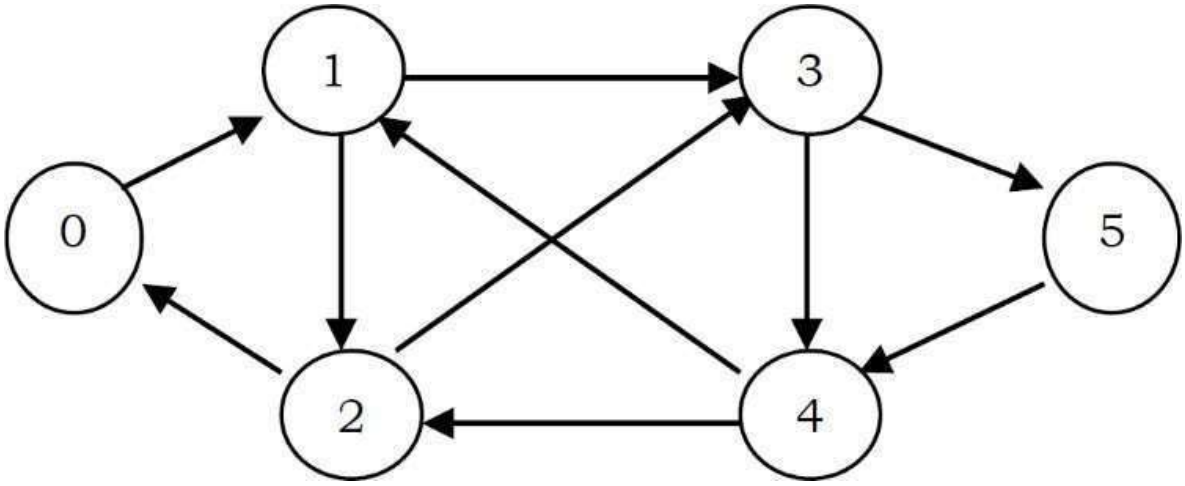
Problem-17 **DFS Application:** Discuss *Euler* Circuits

Solution: Before discussing this problem let us see the terminology:

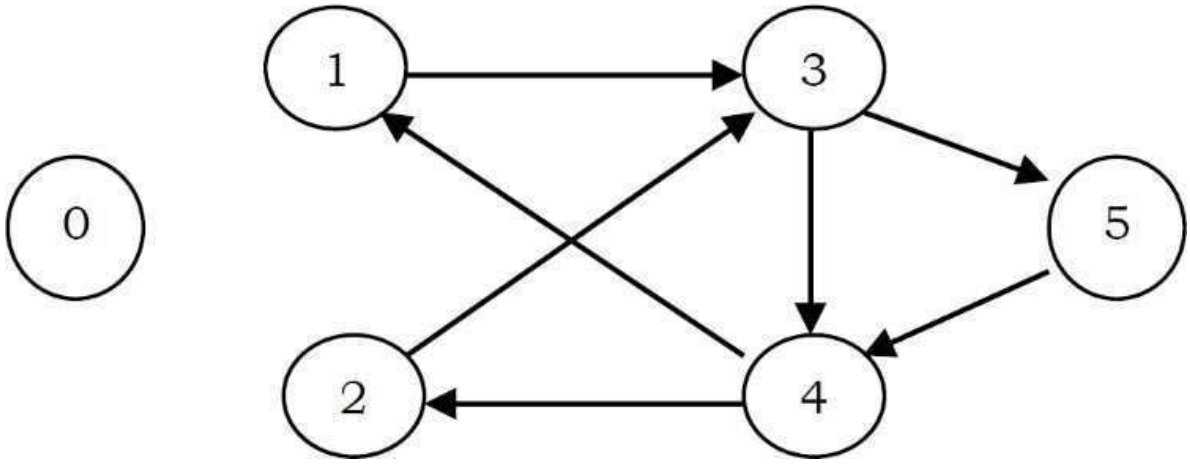
- *Eulerian tour*- a path that contains all edges without repetition.
- *Eulerian circuit* – a path that contains all edges without repetition and starts and ends in the same vertex.
- *Eulerian graph* – a graph that contains an Eulerian circuit.
- *Even vertex:* a vertex that has an even number of incident edges.
- *Odd vertex:* a vertex that has an odd number of incident edges.

Euler circuit: For a given graph we have to reconstruct the circuits using a pen, drawing each line exactly once. We should not lift the pen from the paper while drawing. That means, we must find a path in the graph that visits every edge exactly once and this problem is called an *Euler path* (also called *Euler tour*) or *Euler circuit problem*. This puzzle has a simple solution based on DFS.

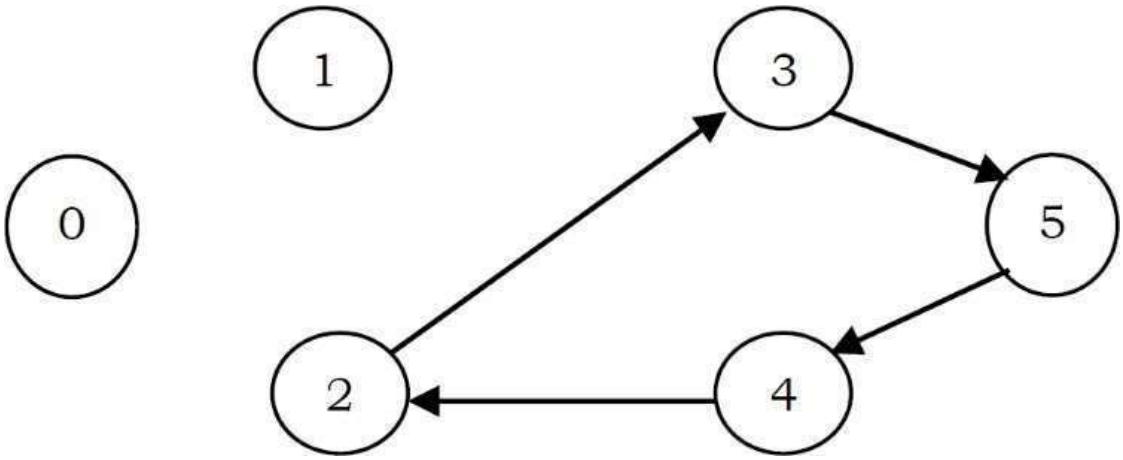
An *Euler* circuit exists if and only if the graph is connected and the number of neighbors of each vertex is even. Start with any node, select any untraversed outgoing edge, and follow it. Repeat until there are no more remaining unselected outgoing edges. For example, consider the following graph: A legal Euler Circuit of this graph is 0 1 3 4 1 2 3 5 4 2 0.



If we start at vertex 0, we can select the edge to vertex 1, then select the edge to vertex 2, then select the edge to vertex 0. There are now no remaining unchosen edges from vertex 0:



We now have a circuit 0,1,2,0 that does not traverse every edge. So, we pick some other vertex that is on that circuit, say vertex 1. We then do another depth first search of the remaining edges. Say we choose the edge to node 3, then 4, then 1. Again we are stuck. There are no more unchosen edges from node 1. We now splice this path 1,3,4,1 into the old path 0,1,2,0 to get: 0,1,3,4,1,2,0. The unchosen edges now look like this:



We can pick yet another vertex to start another DFS. If we pick vertex 2, and splice the path 2,3,5,4,2, then we get the final circuit 0,1,3,4,1,2,3,5,4,2,0.

A similar problem is to find a simple cycle in an undirected graph that visits every vertex. This is known as the *Hamiltonian cycle problem.* Although it seems almost identical to the *Euler* circuit problem, no efficient algorithm for it is known.

Notes:

- A connected undirected graph is *Eulerian* if and only if every graph vertex has an even degree, or exactly two vertices with an odd degree.
- A directed graph is *Eulerian* if it is strongly connected and every vertex has an equal *in* and *out* degree.

Application: A postman has to visit a set of streets in order to deliver mails and packages. He needs to find a path that starts and ends at the post-office, and that passes through each street

(edge) exactly once. This way the postman will deliver mails and packages to all the necessary streets, and at the same time will spend minimum time/effort on the road.

Problem-18 **DFS Application:** Finding Strongly Connected Components.

Solution: This is another application of DFS. In a directed graph, two vertices *u* and *v* are strongly connected if and only if there exists a path from *u* to *v* and there exists a path from *v* to *u*. The strong connectedness is an equivalence relation.

- A vertex is strongly connected with itself
- If a vertex *u* is strongly connected to a vertex *v*, then *v* is strongly connected to *u*
- If a vertex *u* is strongly connected to a vertex *v*, and *v* is strongly connected to a vertex *x*, then *u* is strongly connected to *x*

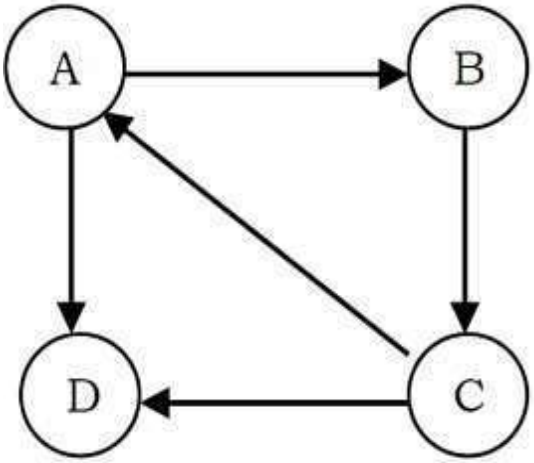
What this says is, for a given directed graph we can divide it into strongly connected components. This problem can be solved by performing two depth-first searches. With two DFS searches we can test whether a given directed graph is strongly connected or not. We can also produce the subsets of vertices that are strongly connected.

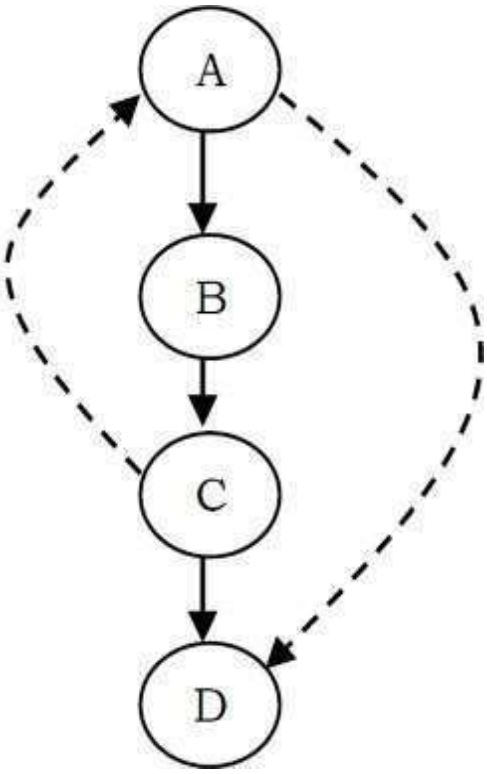
Algorithm

- Perform DFS on given graph *G*.
- Number vertices of given graph *G* according to a post-order traversal of depth-first spanning forest.
- Construct graph G_r by reversing all edges in *G*.
- Perform DFS on G_r*:* Always start a new DFS (initial call to Visit) at the highest-numbered vertex.
- Each tree in the resulting depth-first spanning forest corresponds to a strongly-connected component.

Why this algorithm works?

Let us consider two vertices, *v* and *w*. If they are in the same strongly connected component, then there are paths from *v* to *W* and from *w* to *v* in the original graph *G*, and hence also in G_r. If two vertices *v* and w are not in the same depth-first spanning tree of G_r, clearly they cannot be in the same strongly connected component. As an example, consider the graph shown below on the left. Let us assume this graph is *G*.



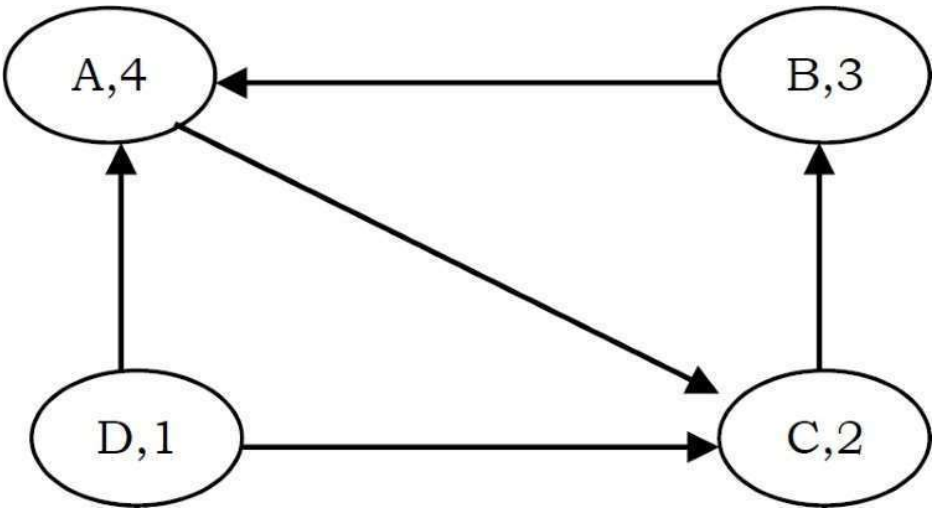


Now, as per the algorithm, performing *DFS* on this *G* graph gives the following diagram. The dotted line from *C* to *A* indicates a back edge.

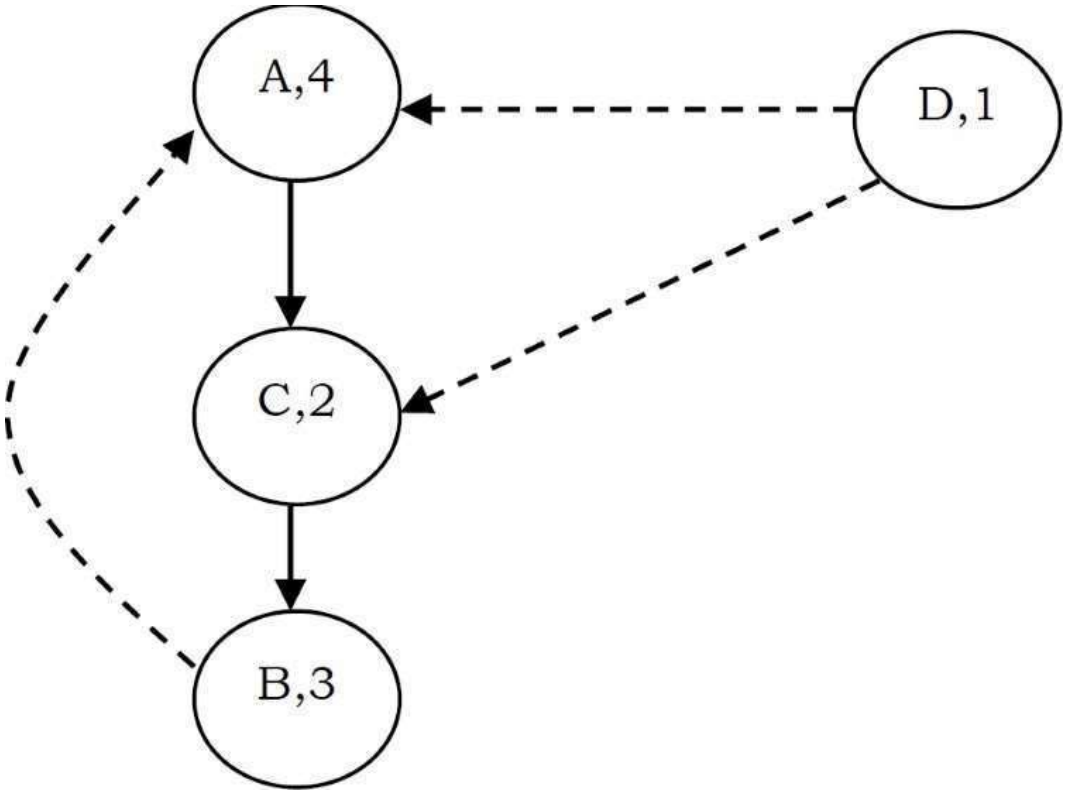
Now, performing post order traversal on this tree gives: *D,C,B* and *A*.

Vertex	Post Order Number
A	4
B	3
C	2
D	1

Now reverse the given graph *G* and call it G_r and at the same time assign postorder numbers to the vertices. The reversed graph G_r will look like:



The last step is performing DFS on this reversed graph G_r. While doing *DFS*, we need to consider the vertex which has the largest DFS number. So, first we start at *A* and with *DFS* we go to *C* and then *B*. At B, we cannot move further. This says that {*A*, *B*, *C*} is a strongly connected component. Now the only remaining element is *D* and we end our second *DFS* at *D*. So the connected components are: {*A*, *B*, *C*} and {*D*}.



The implementation based on this discussion can be shown as:

```
//Graph represented in adj matrix.
int adjMatrix [256][256], table[256];
vector <int> st ;
int counter = 0 ;
//This table contains the DFS Search number
int dfsnum [256], num = 0, low[256] ;
void StronglyConnectedComponents( int u ) {
      low[u] = dfsnum[ u ] = num++;
      Push(st, u) ;
      for( int v = 0 ; v < 256; ++v ) {
            if(graph[u][v] && table[v] == -1) {
                  if( dfsnum[v] == -1)
                     StronglyConnectedComponents(v) ;
                  low[u] = min(low[u] , low[v]) ;
            }
      }
      if(low[u] == dfsnum[u]) {
            while( table[u] != counter) {
                  table[st.back()] = counter;
                  Push(st) ;
            }
            ++ counter;
      }
}
```

Problem-19 Count the number of connected components of Graph *G* which is represented in the adjacent matrix.

Solution: This problem can be solved with one extra counter in *DFS*.