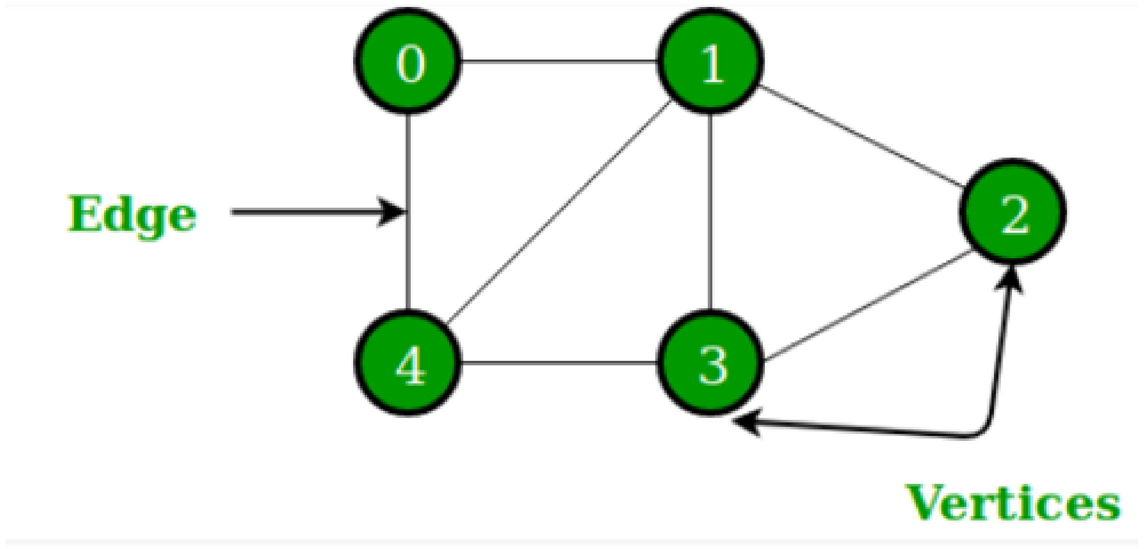
Unit 1
Graph

The Graph Abstract Data Type, Introduction, Definition, Graph Representation, Elementary Graph Operation, Depth First Search, Breadth First Search, Connected Components, Spanning Trees, Biconnected Components, Minimum Cost Spanning Trees, Kruskal’s Algorithm, Prim s Algorithm, Sollin’s Algorithm, Shortest Paths and Transitive Closure, Single Source/All Destination: Nonnegative Edge Cost, Single Source/All Destination: General Weights, All-Pairs Shortest Path, Transitive Closure.

Define Graph?

- ➢**Non- linear Data Structure**.
- ➢Consisting of set of nodes and edges.
- ➢The **Nodes** are also called as **Vertices, Objects, Things** and the **Edges** are also called as **Paths, Lines**.
- ➢A Graph G(V,E) consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

Graph Example



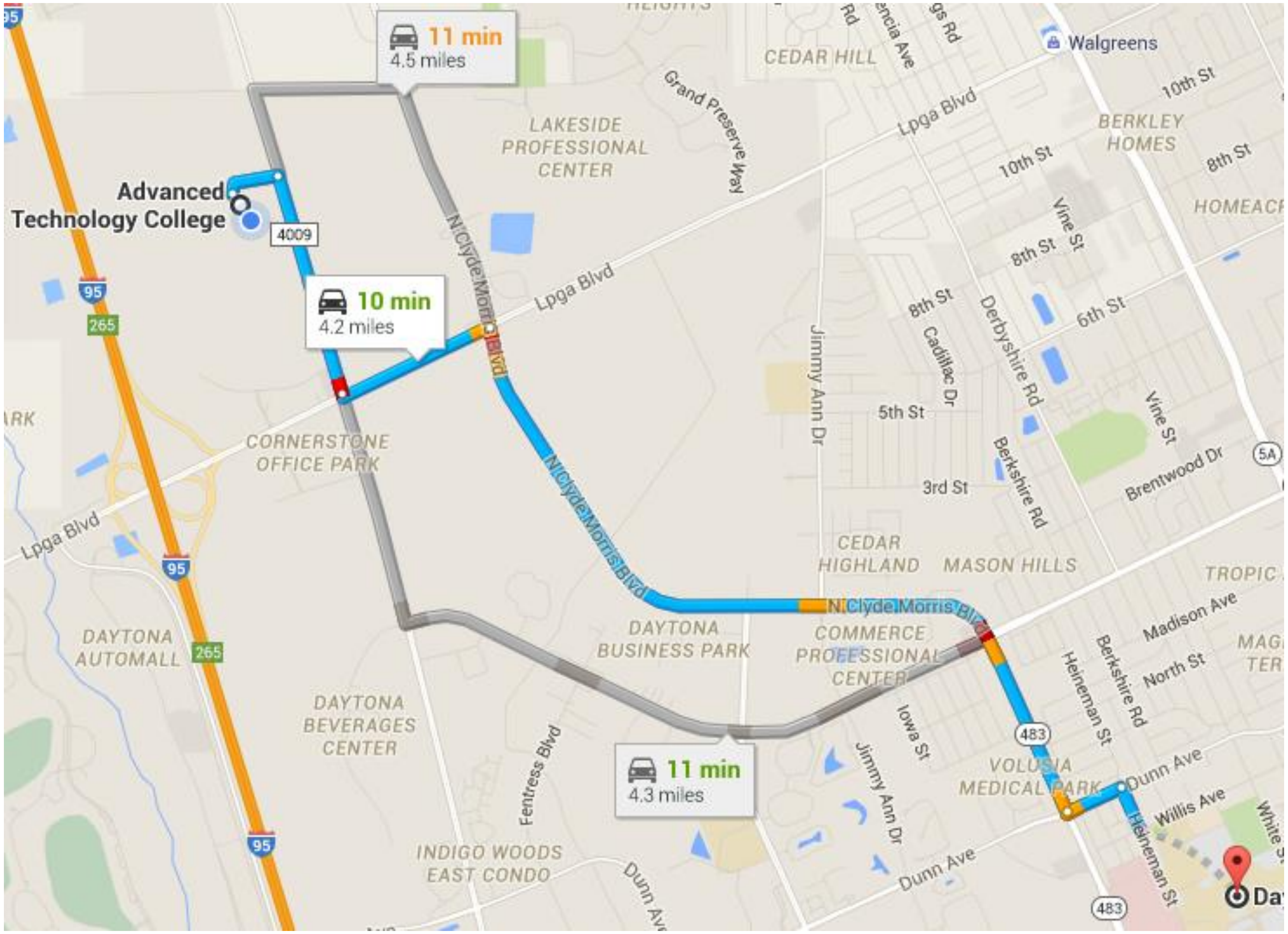
the set of vertices V = {0,1,2,3,4} and
the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Why Graphs? (or) Graph Applications

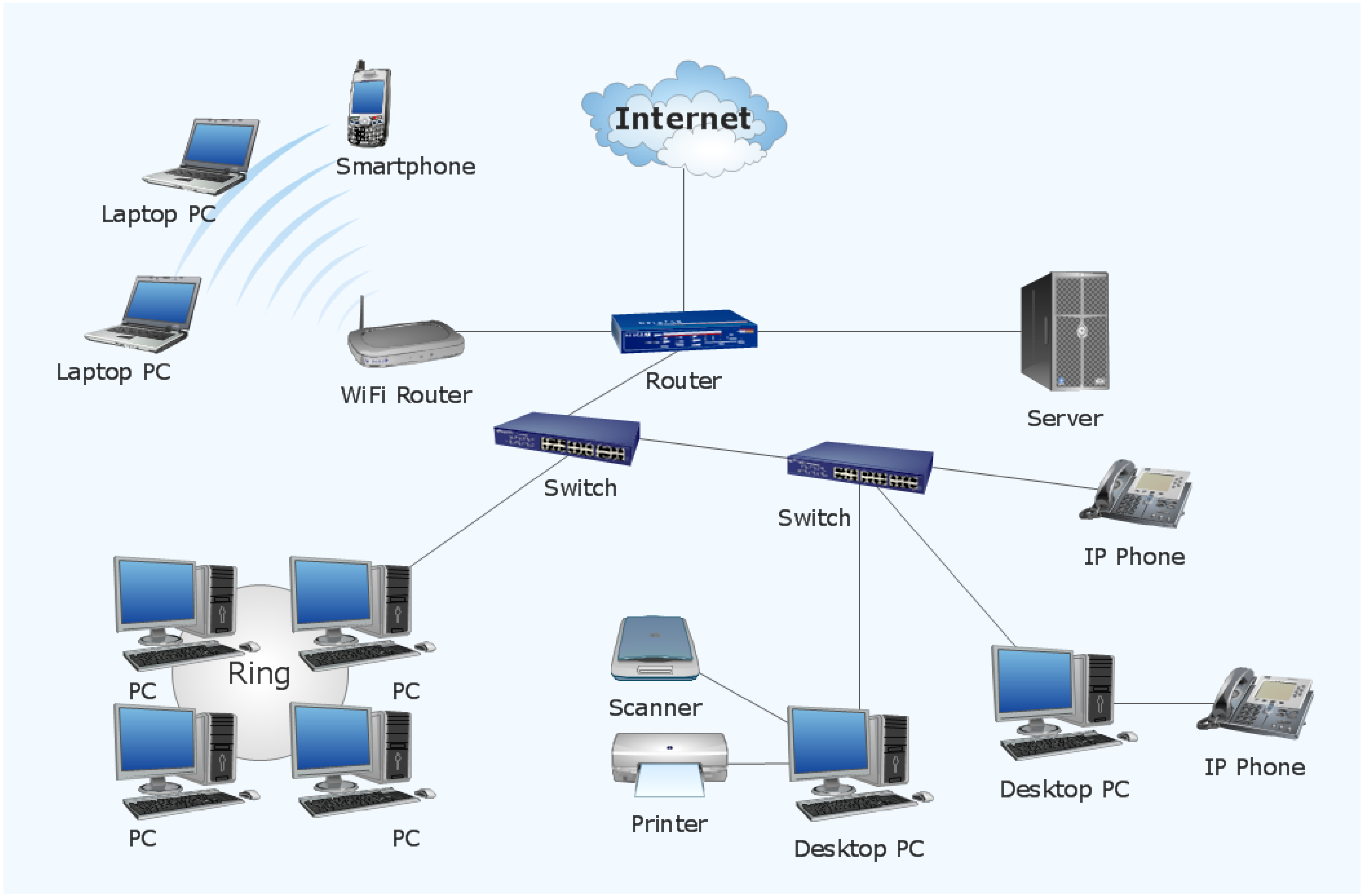
Graphs are used to solve many **real-life problems**.

- **Recommendations:** Amazon, Flipkart uses graphs to make suggest products.
- **Social networks** E.g., Facebook/ LinkedIn uses a graph for suggesting friends.
- **GPS Navigation**
- **Networks routing**
- **Driving directions**
- **Airline Traffic**
- **Telcom:** Cell tower frequency planning.

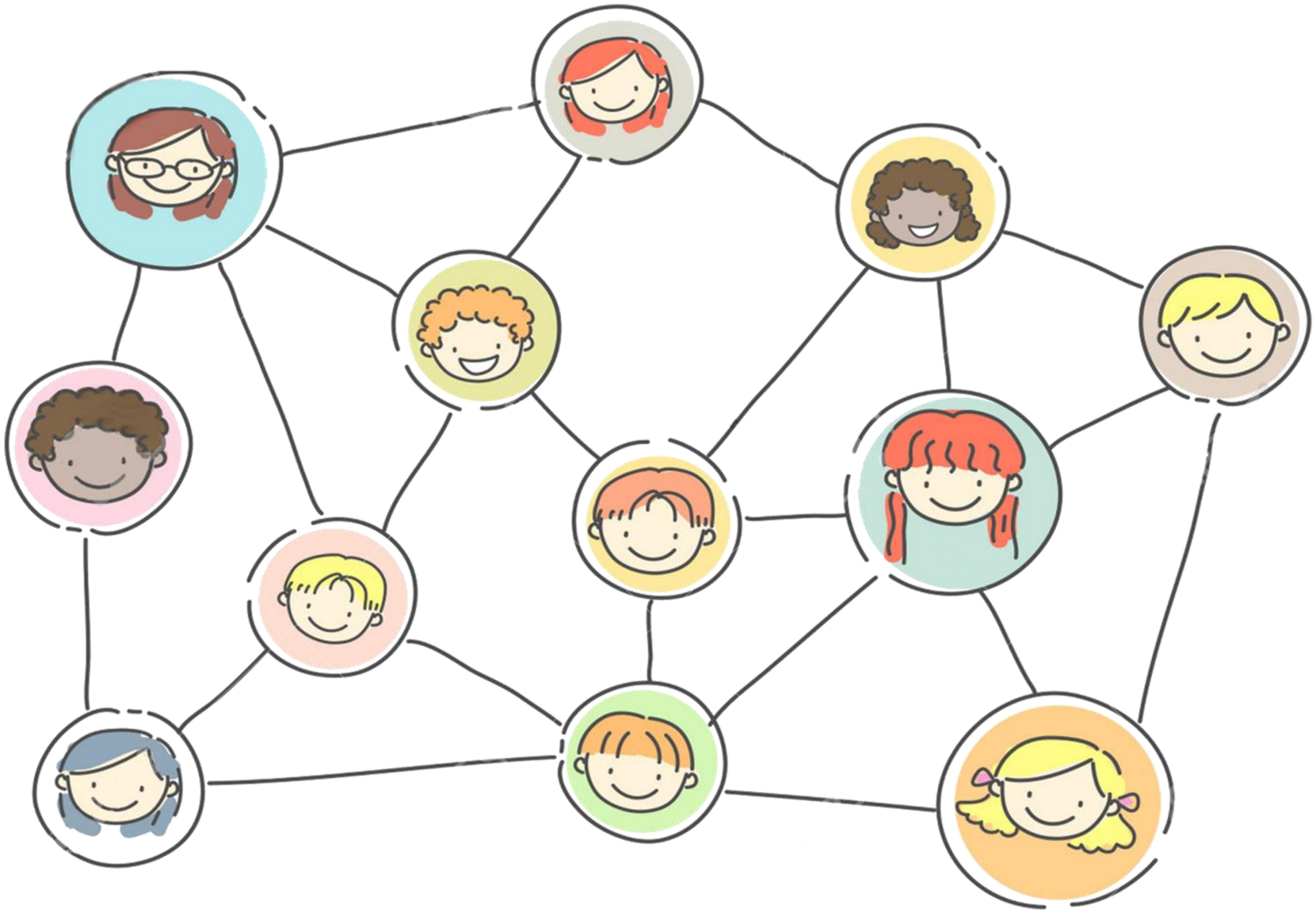
GPS Navigation



Networks routing

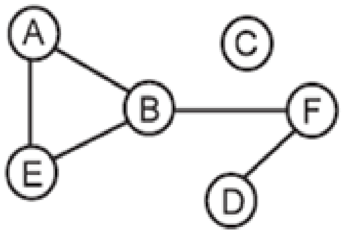


Linkedin- Suggesting friends.

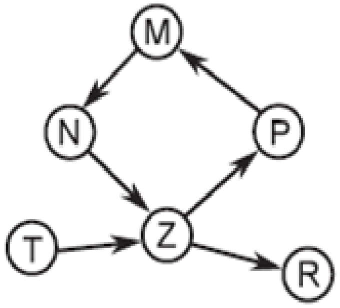


Basics of Graphs

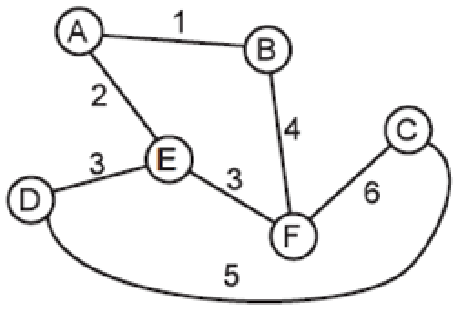
- **Directed graph-** a set of vertices that are connected together, where all the edges are directed from one vertex to another.
- Also called as digraph or a directed network.
- **Undirected graph-** a set of vertices that are connected together, where all the edges are bidirectional.



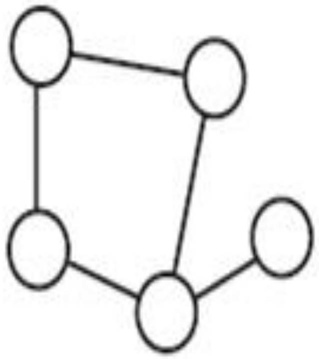
(a) Graph G_1—an undirected graph.



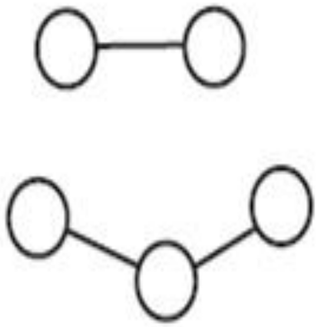
(b) Graph G_2—a directed graph.



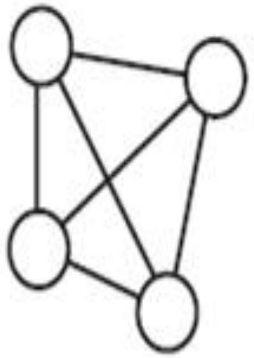
(d) Graph G_4—an undirected graph with weighted edges.



(a) A connected graph.



(b) A disconnected graph.



(c) A complete graph.

What is Graph ADT?

- **Graph()** creates a new, empty graph.
- **addVertex(vert)** adds an instance of Vertex to the graph.
- **addEdge(fromVert, toVert)** Adds a new, directed edge to the graph that connects two vertices.
- **addEdge(fromVert, toVert, weight)** Adds a new, weighted, directed edge to the graph that connects two vertices.
- **DeleteVertex(graph, v)** return a graph in which v and all edges incident to it are removed
- **DeleteEdge(graph, v1, v2)** return a graph in which the edge (v1, v2) is removed.
- **getVertex(vertKey)** finds the vertex in the graph named vertKey.
- **getVertices()** returns the list of all vertices in the graph.
- **Boolean IsEmpty(graph)** .

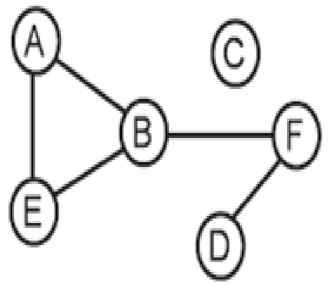
Graph Representation

- Two ways to implement or represent graphs.
- They are

1. **Adjacency Matrix Representation:** To implement a graph is to use a **two-dimensional matrix.**

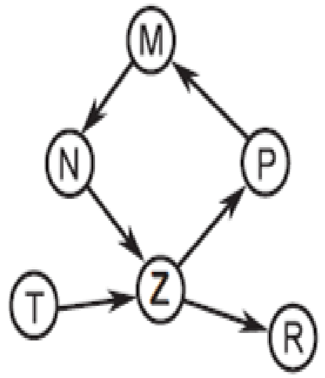
1. **Adjacency List Representation:** To implement a graph is to use list of **Linked List.**

Adjacency Matrix Representation



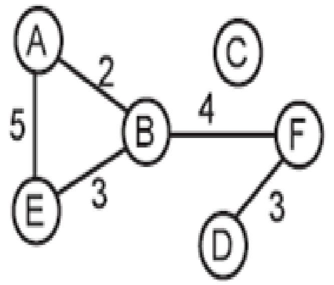
	A	B	C	D	E	F
A	0	1	0	0	1	0
B	1	0	0	0	1	1
C	0	0	0	0	0	0
D	0	0	0	0	0	1
E	1	1	0	0	0	0
F	0	1	0	1	0	0

(a) Adjacency matrix of an undirected graph.



	M	N	P	R	T	Z
M	0	1	0	0	0	0
N	0	0	0	0	0	1
P	1	0	0	0	0	0
R	0	0	0	0	0	0
T	0	0	0	0	0	1
Z	0	0	1	1	0	0

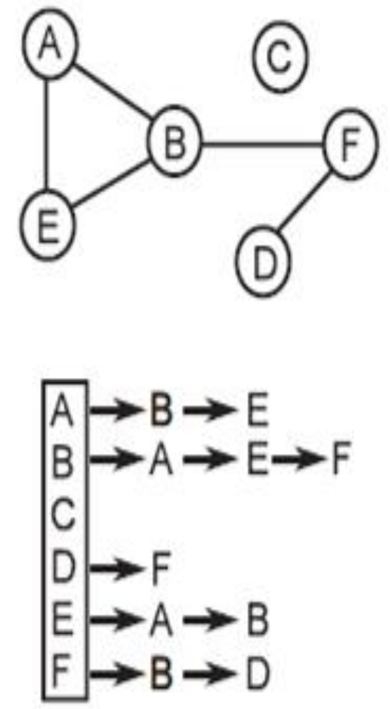
(b) Adjacency matrix of a directed graph.



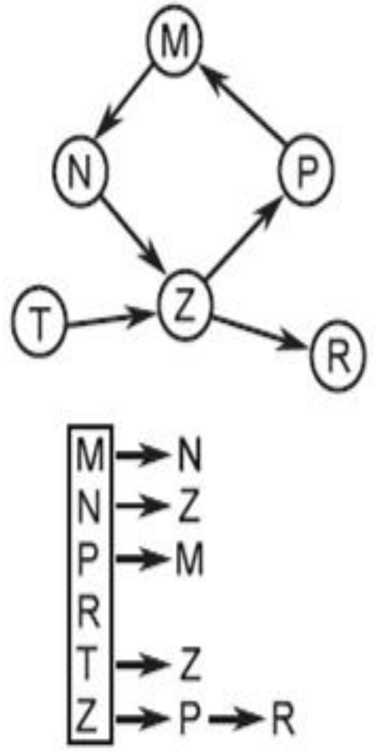
	A	B	C	D	E	F
A	0	2	0	0	5	0
B	2	0	0	0	3	4
C	0	0	0	0	0	0
D	0	0	0	0	0	3
E	5	3	0	0	0	0
F	0	4	0	3	0	0

(c) Adjacency matrix of an undirected weighted graph.

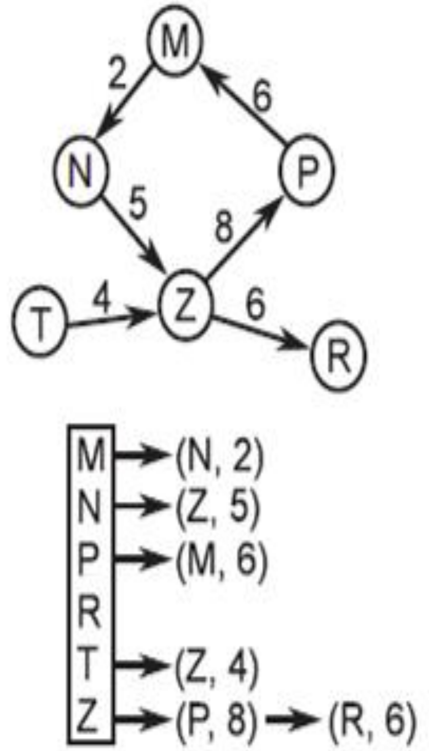
Adjacency List Representation



(a) Adjacency list of an undirected graph.



(b) Adjacency list of a directed graph.



(c) Adjacency list of a directed weighted graph.

Graph Operations

- Basic primary operations of a Graph
  - **Add Vertex** − Adds a vertex to the graph.
  - **Add Edge** − Adds an edge between the two vertices of the graph.
  - **Display Vertex** − Displays a vertex of the graph.
  - **Delete Vertex-** Removes a vertex from the graph.
  - **Delete Edge-** Removes an edge.
  - **get_vertex_value -** returns a value of vertex.
  - **set_vertex_value** - returns a value of edge.

Graph Traversals

- Graph traversal is the process of visiting each vertex at most once in a graph.

- Two Techniques:
  1. **DFS (**Depth First Search**)**
  2. **BFS (**Breadth First Search**)**

Depth First Search (DFS)

- DFS is a recursive algorithm for traversing all the vertices of a graph and finally produces spanning tree.

- Stack Data structure is used to implement DFS.

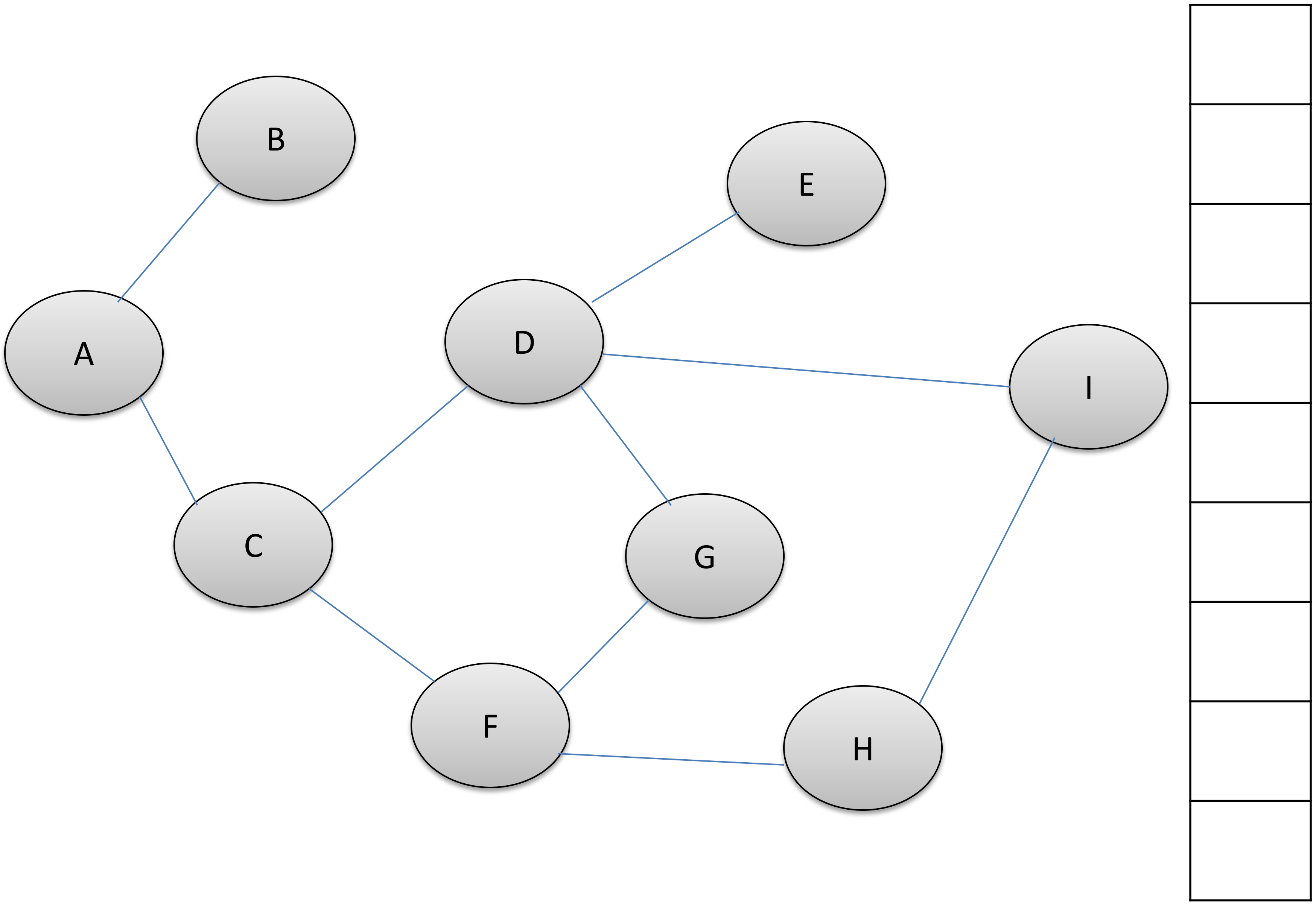
The following steps are used to implement DFS...

- Step 1 - Start traversing from any one vertex of a graph. Mark it as visited and push it on to the Stack.
- Step 2 - Traverse to any one of the unvisited adjacent vertices of top of stack. Mark it as visited and push it on to the stack.
- Step 3 - Repeat step 2 until there is no unvisited adjacent vertex of top of the stack.
- Step 4 - If there is no unvisited vertex then pop the vertex from the stack.
- Step 5 - Repeat steps 2,3 and 4 until stack becomes Empty.
- Step 6 - When stack becomes Empty, then produce final spanning tree.

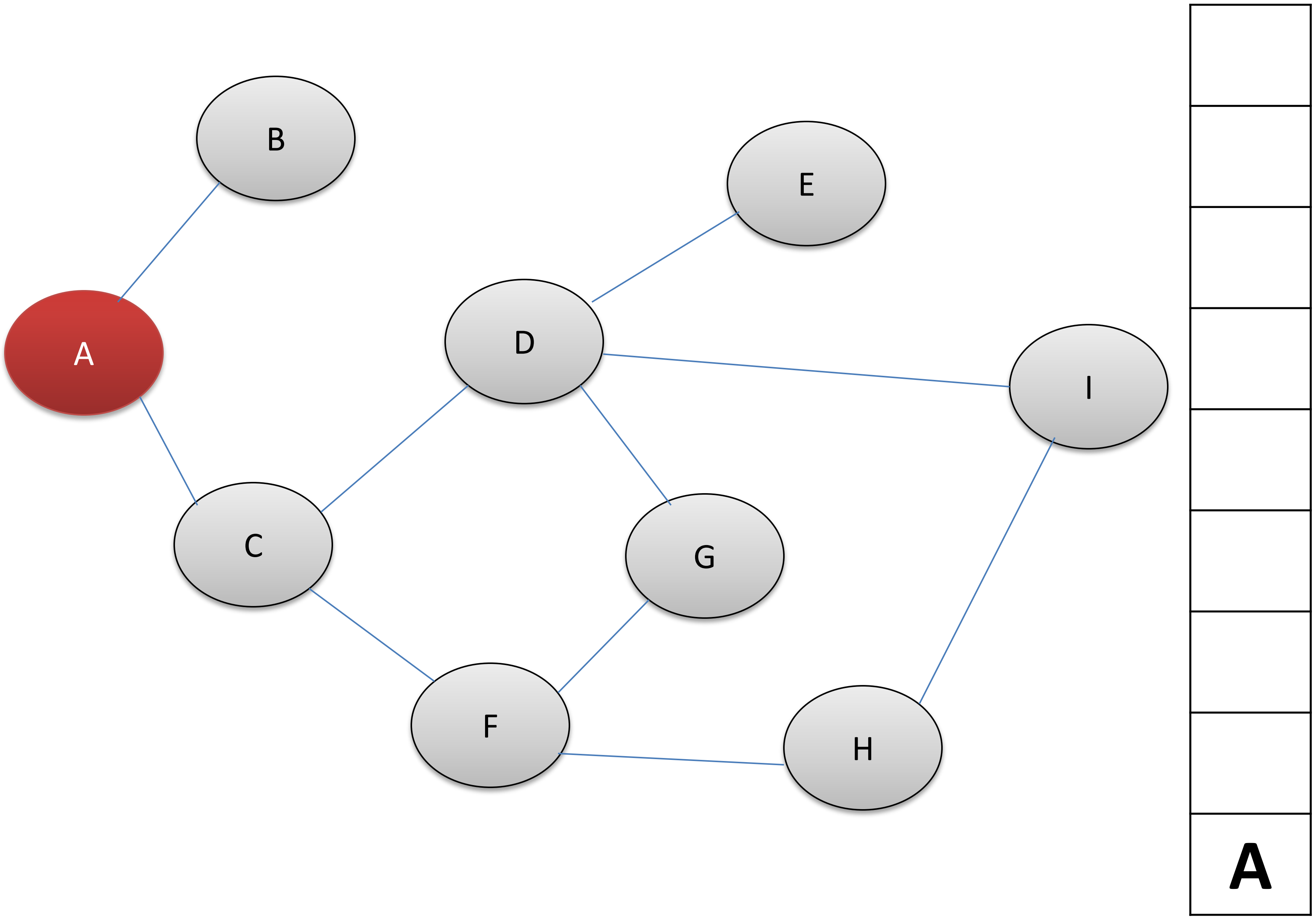
DFS Algorithm

```
DFS(V)
{
    mark V as visited
    for each adjacent vertex K of V
    {
        if K is unvisited
        {
            DFS(K)
        }
    }
}
```

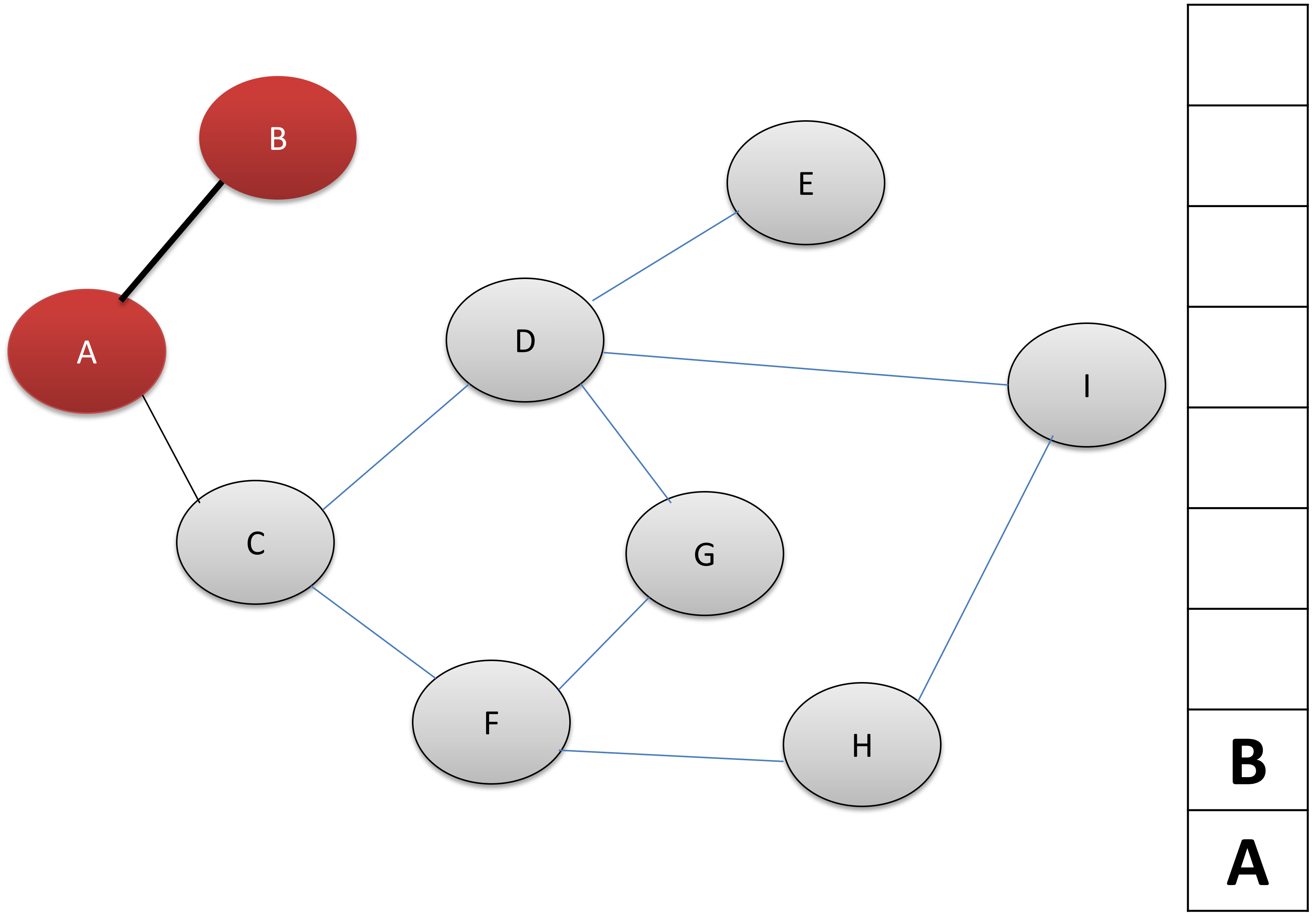
Example



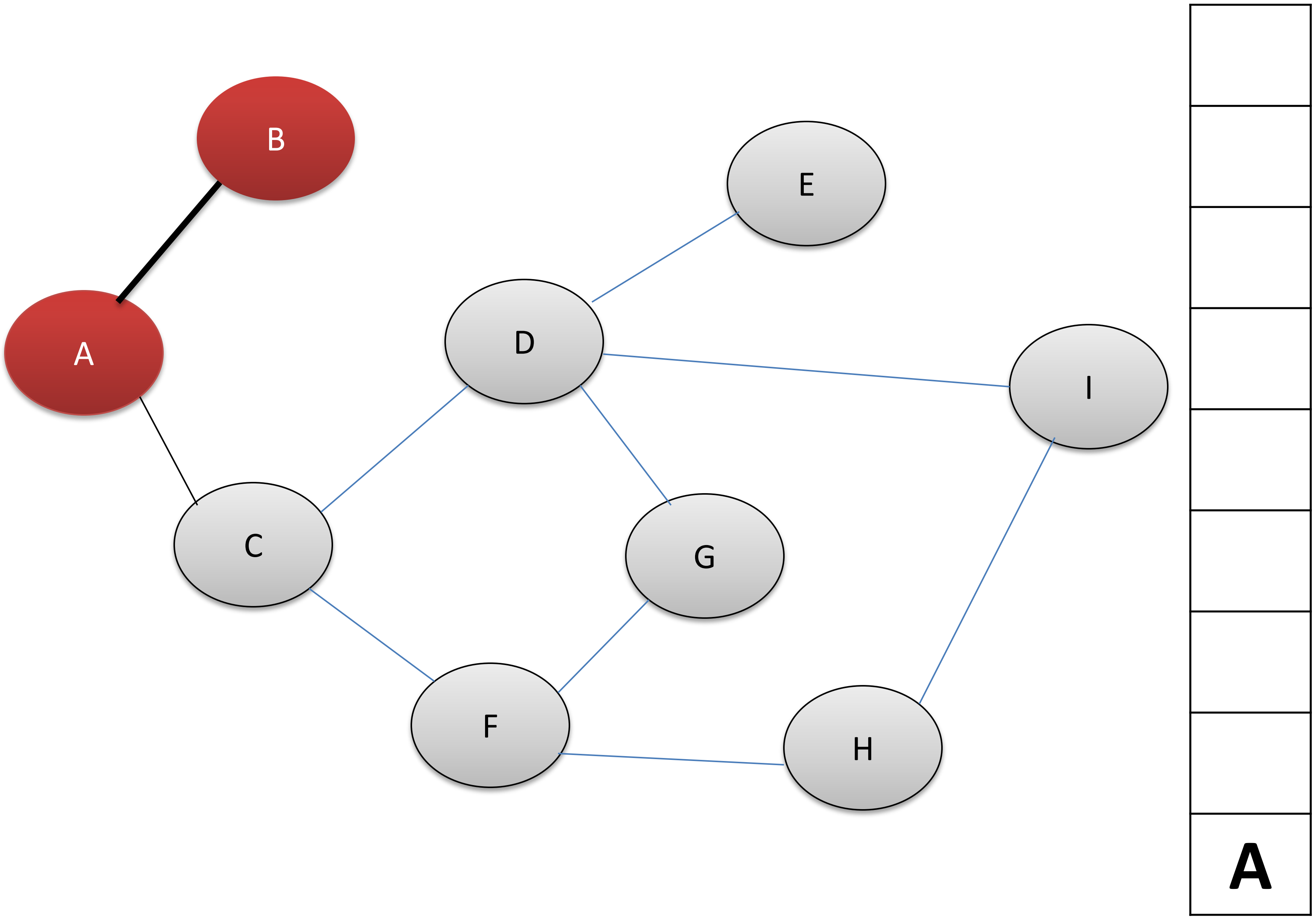
Example



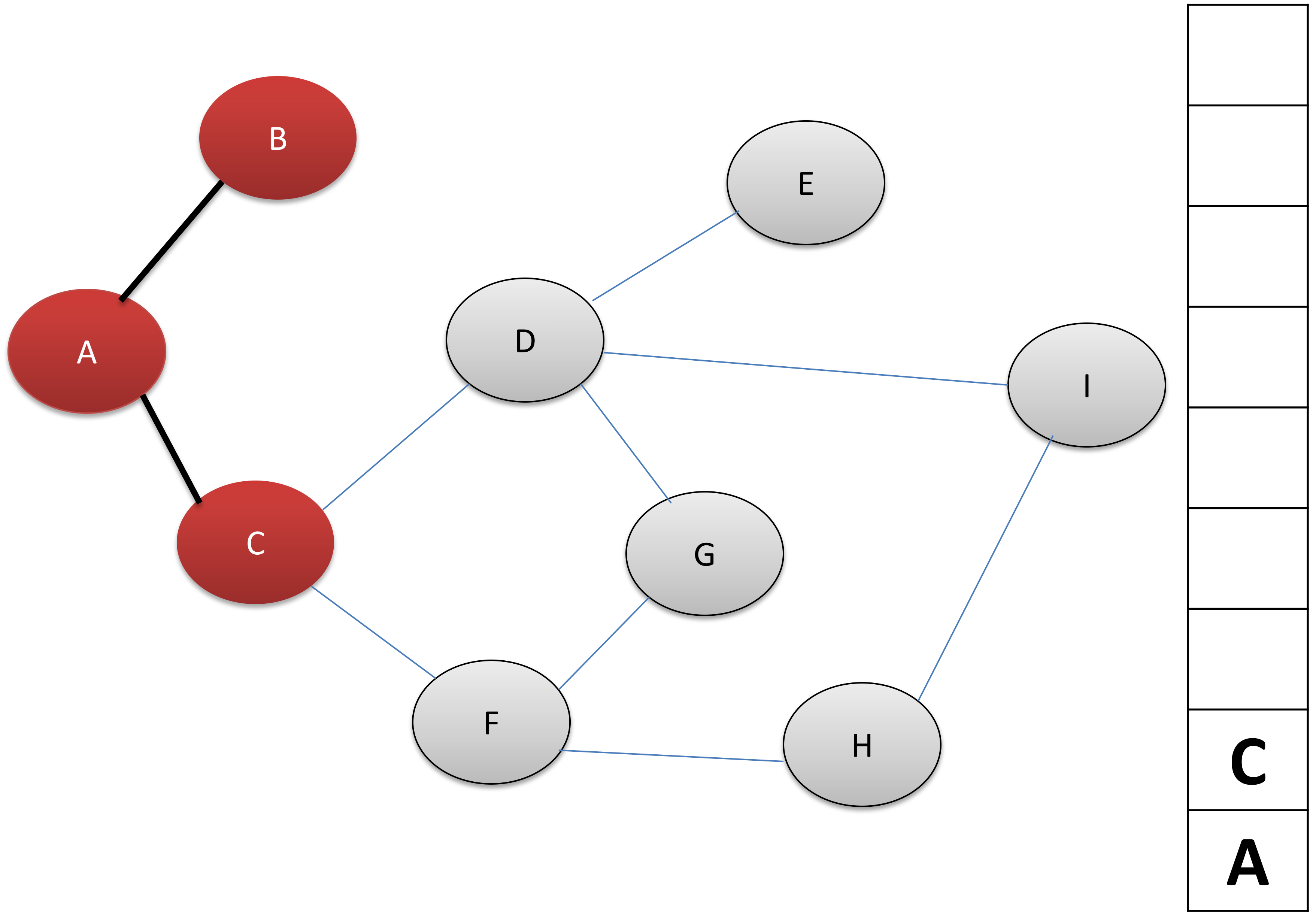
Example



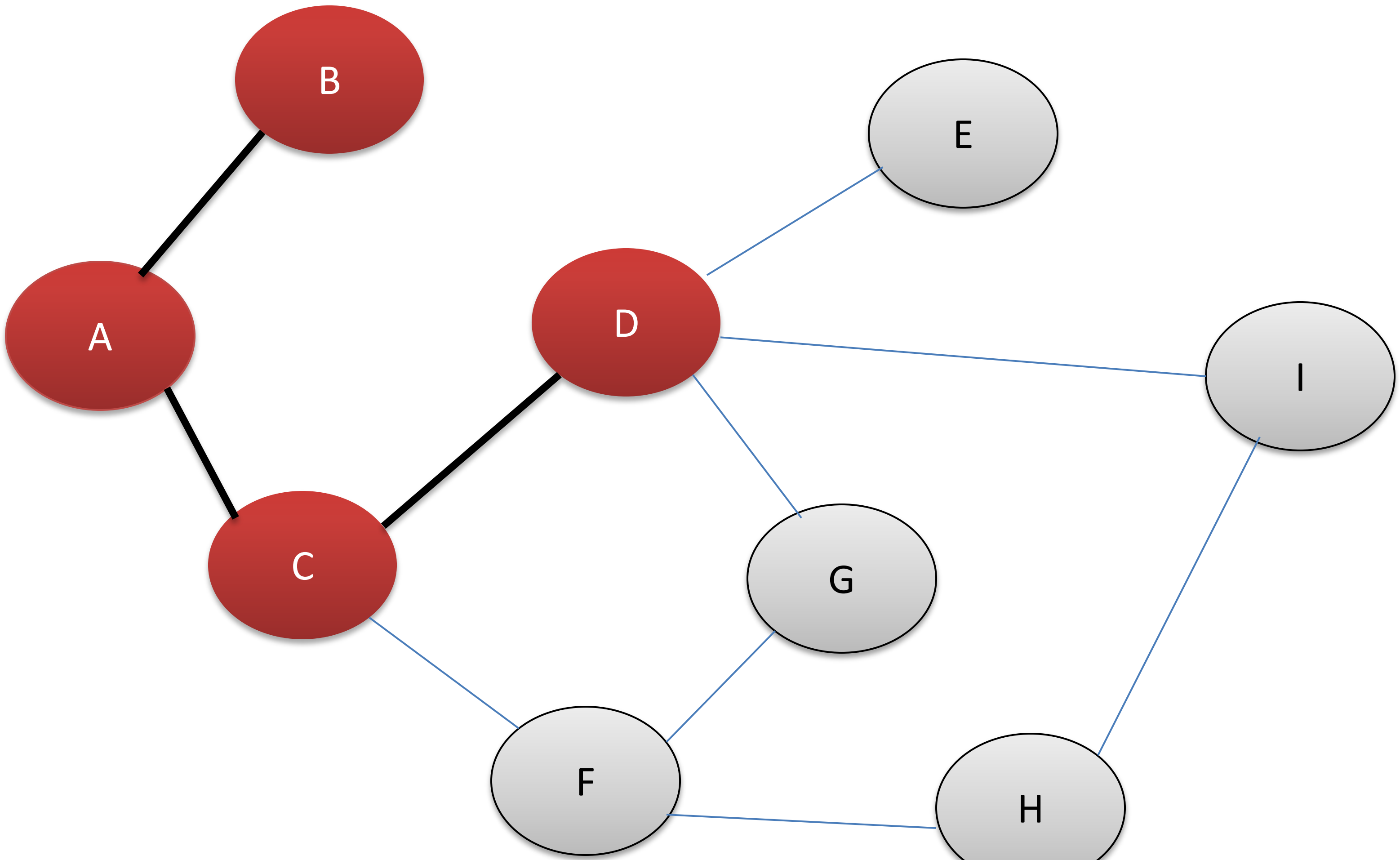
Example

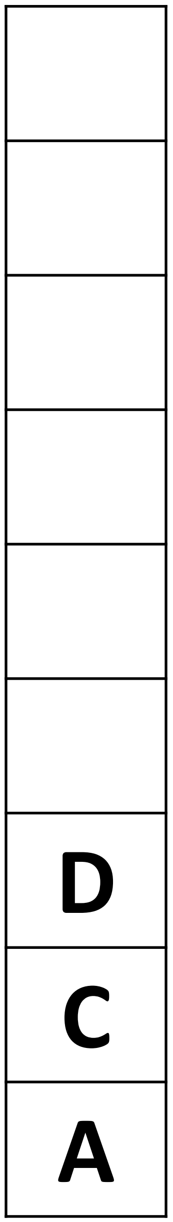


Example

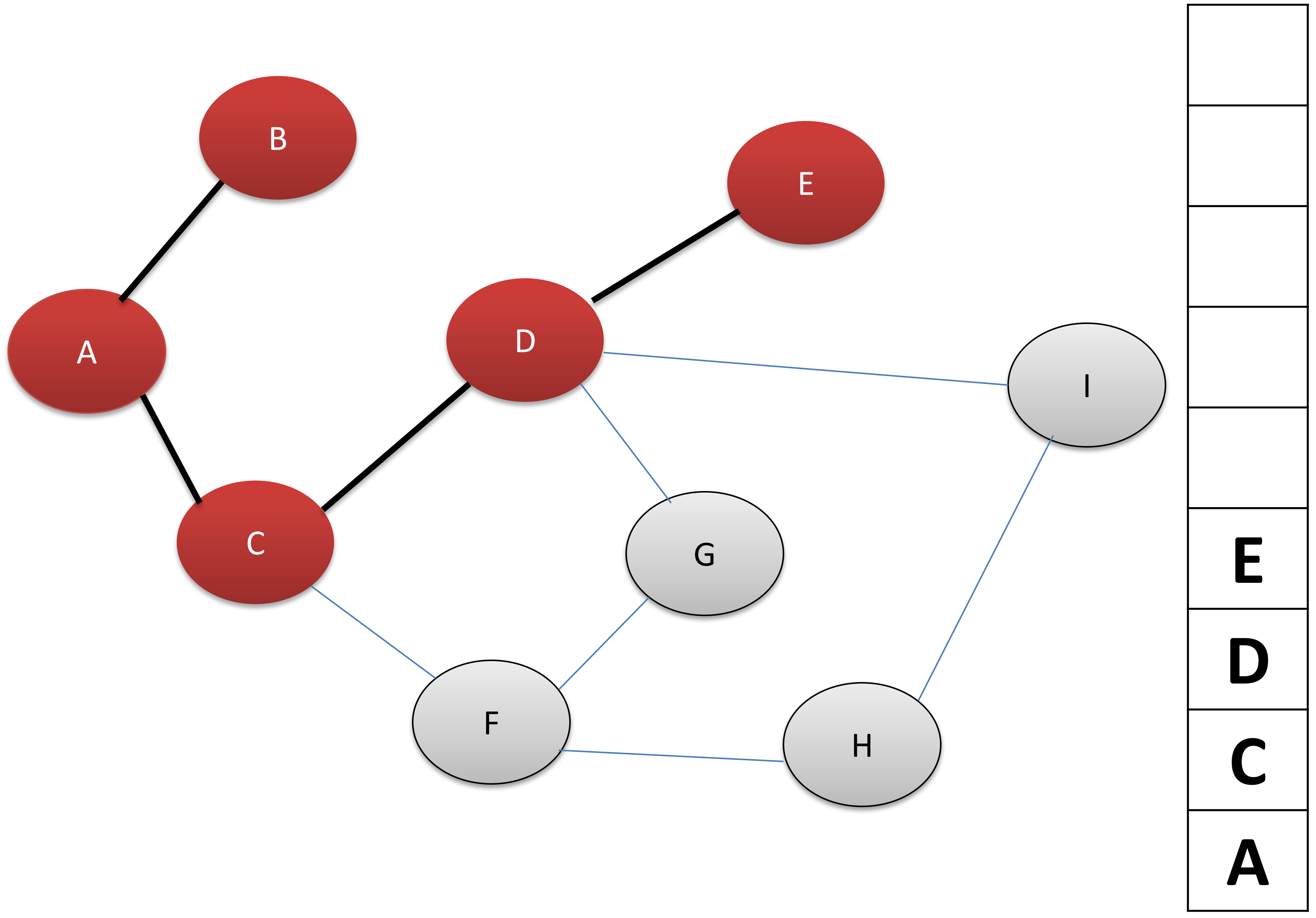


Example

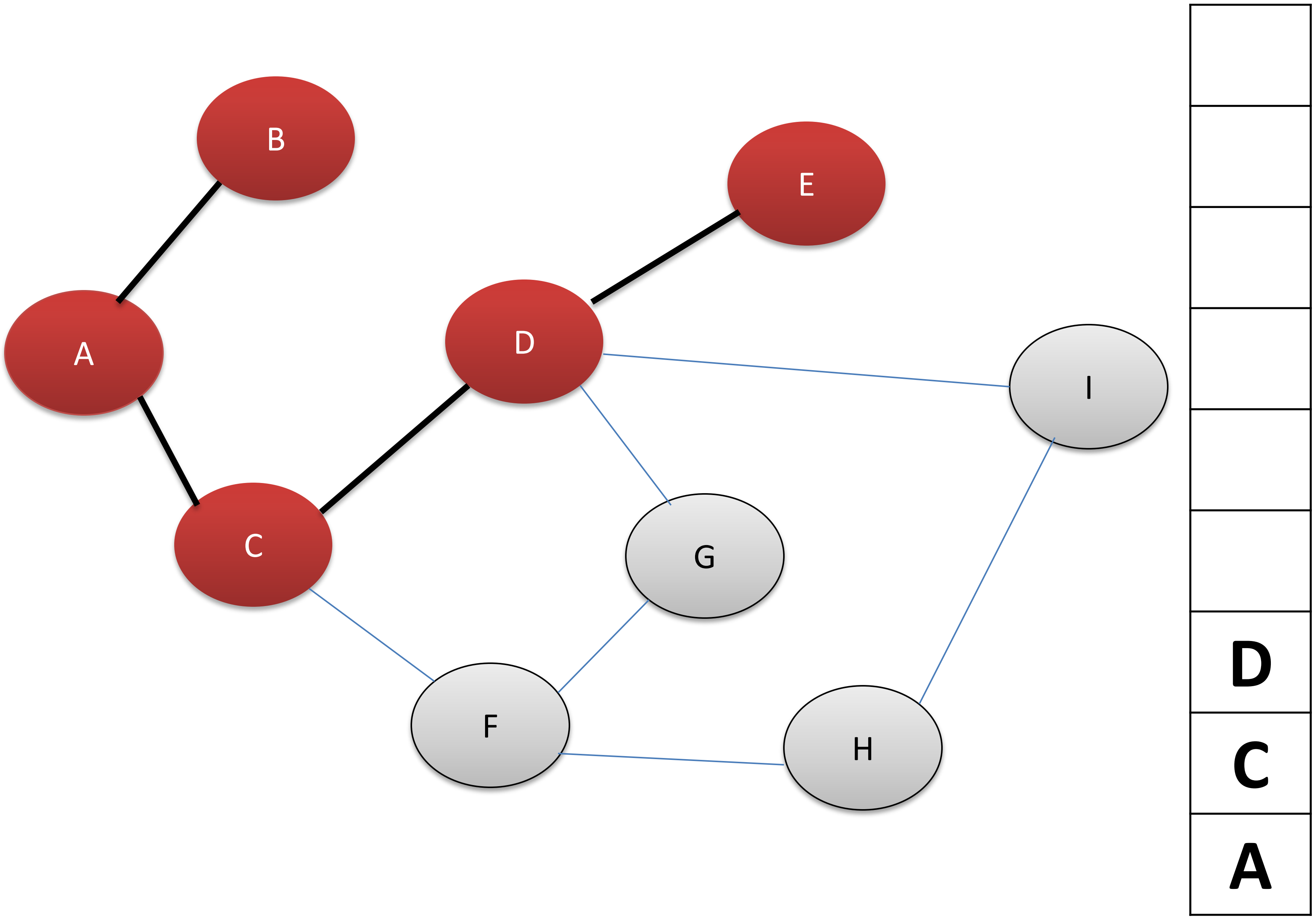




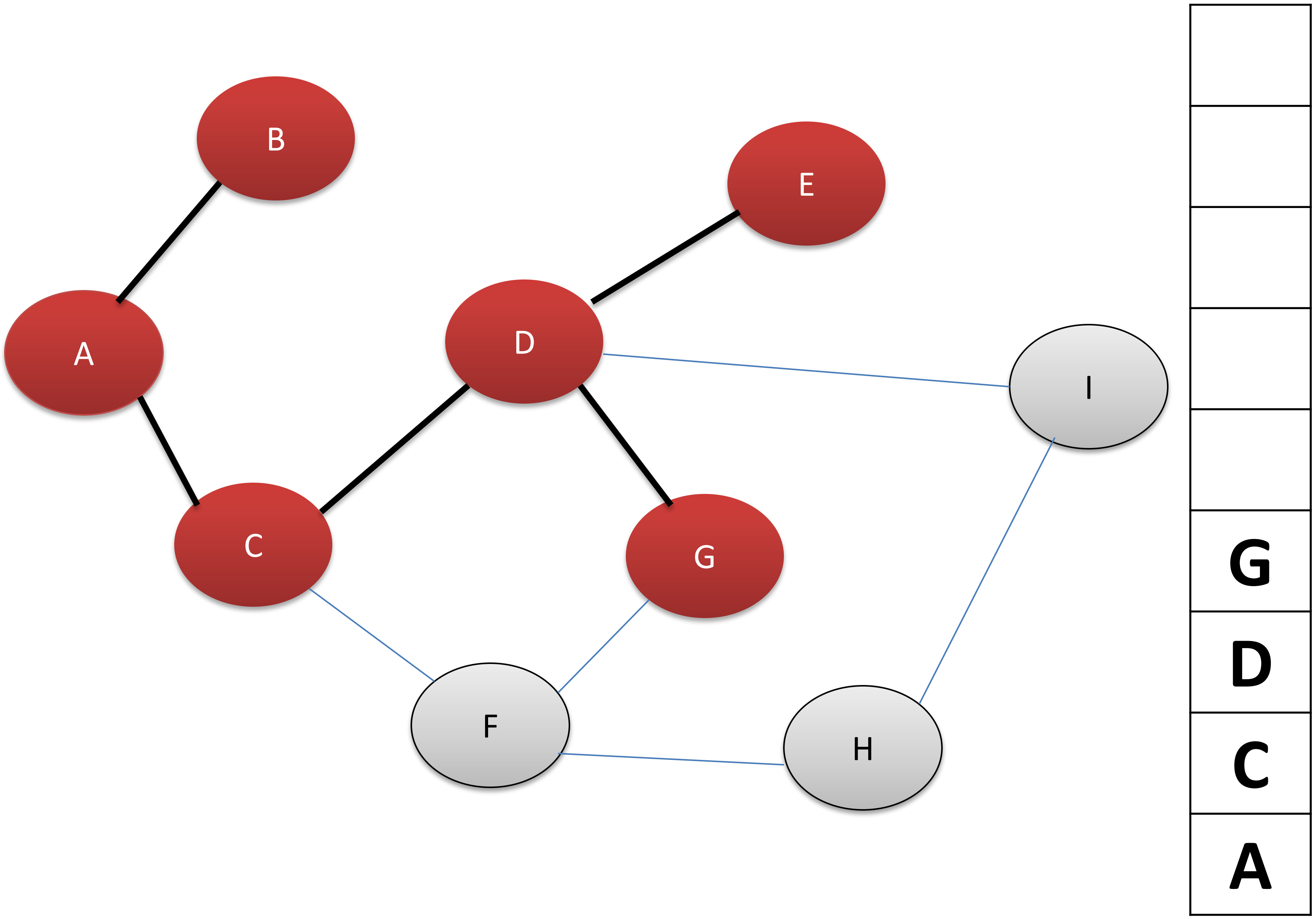
Example



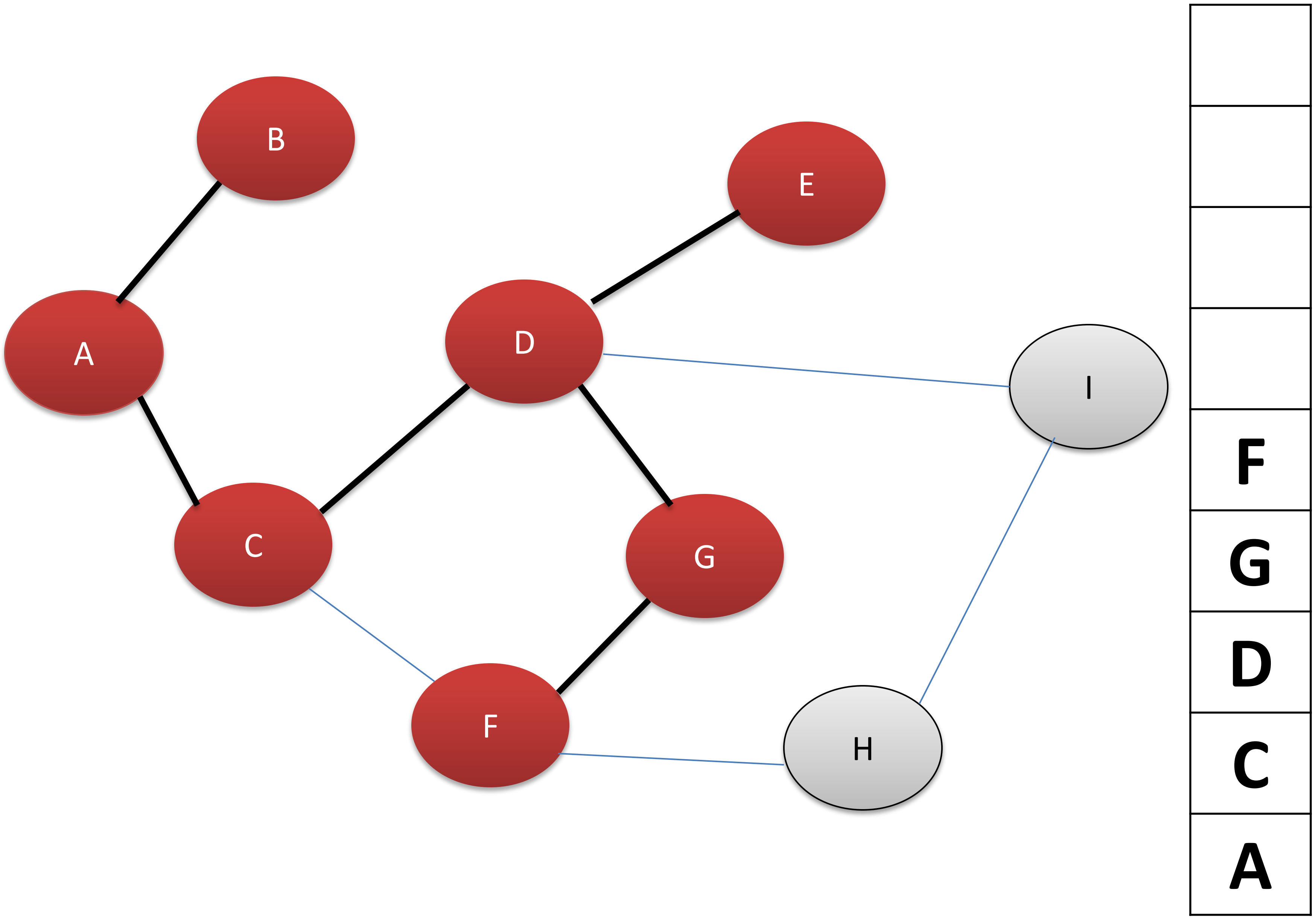
Example



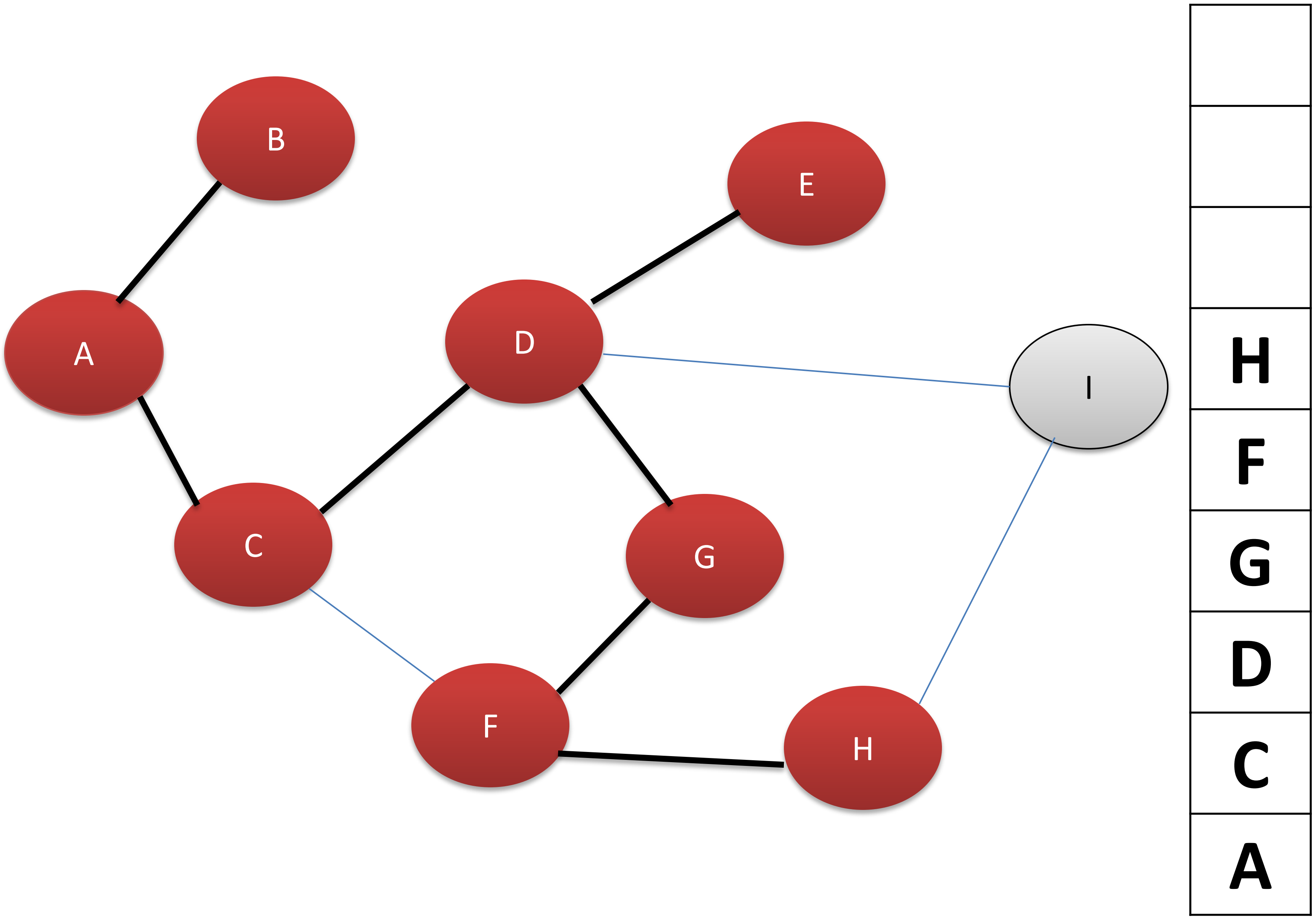
Example



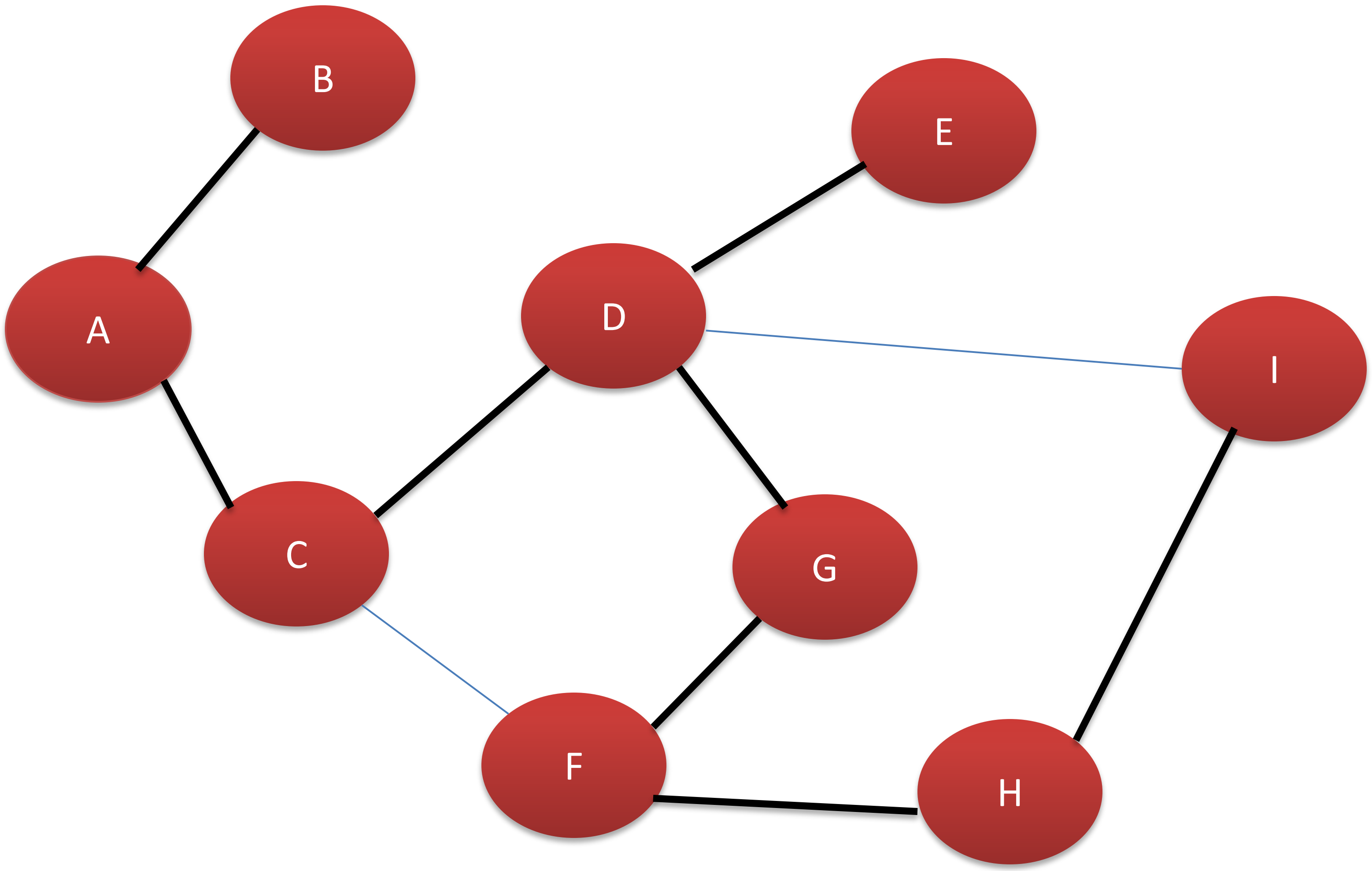
Example

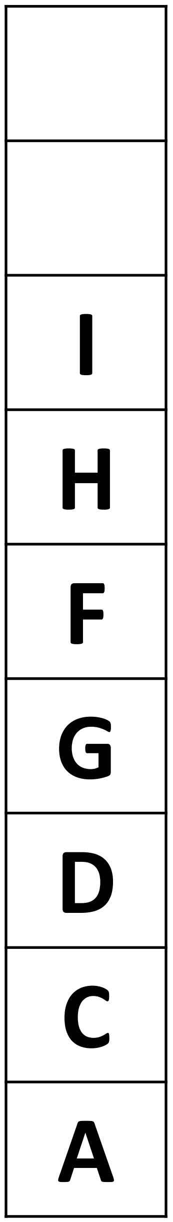


Example

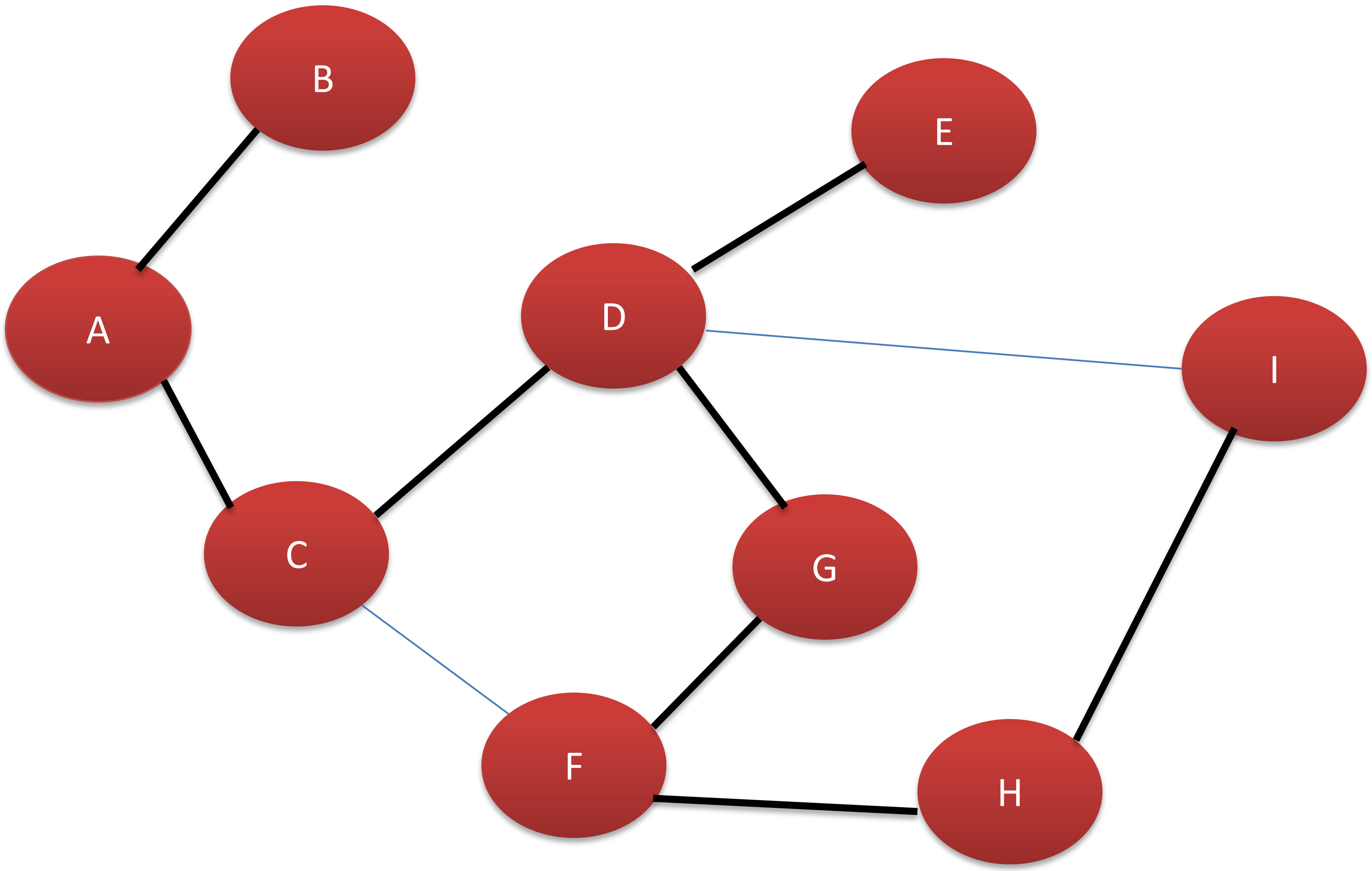


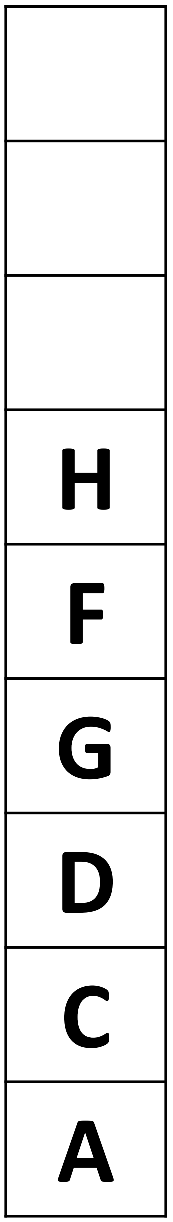
Example



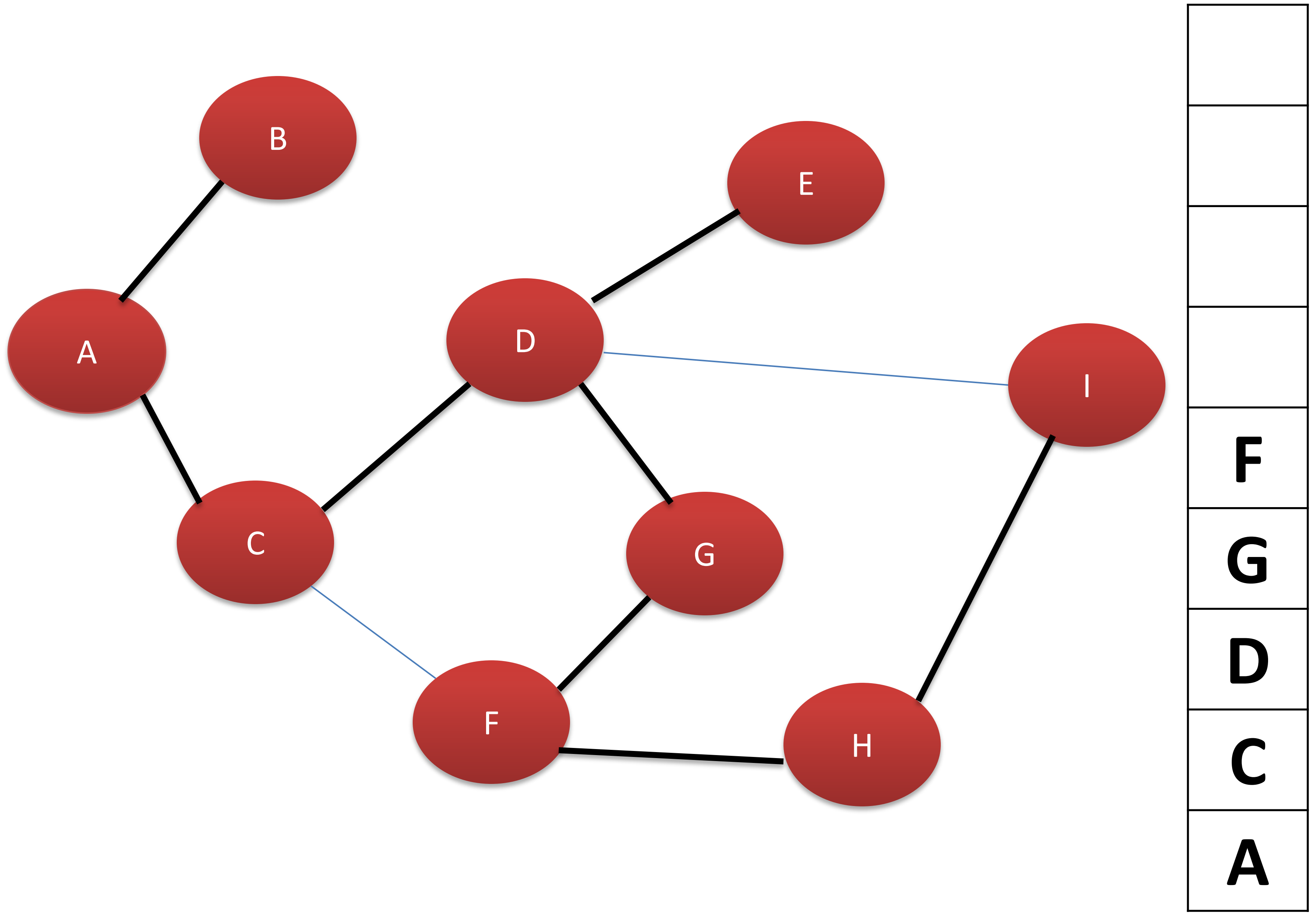


Example

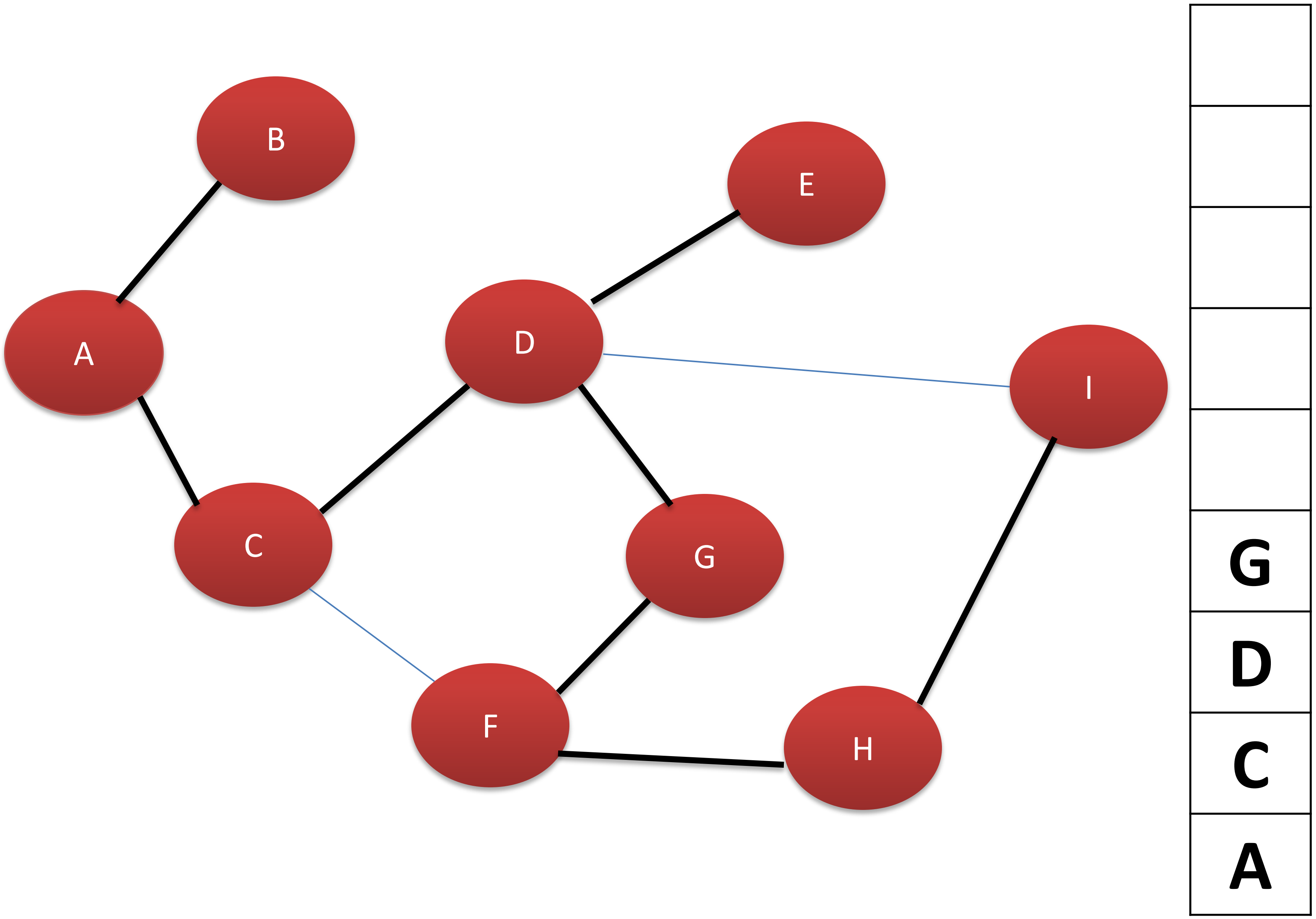




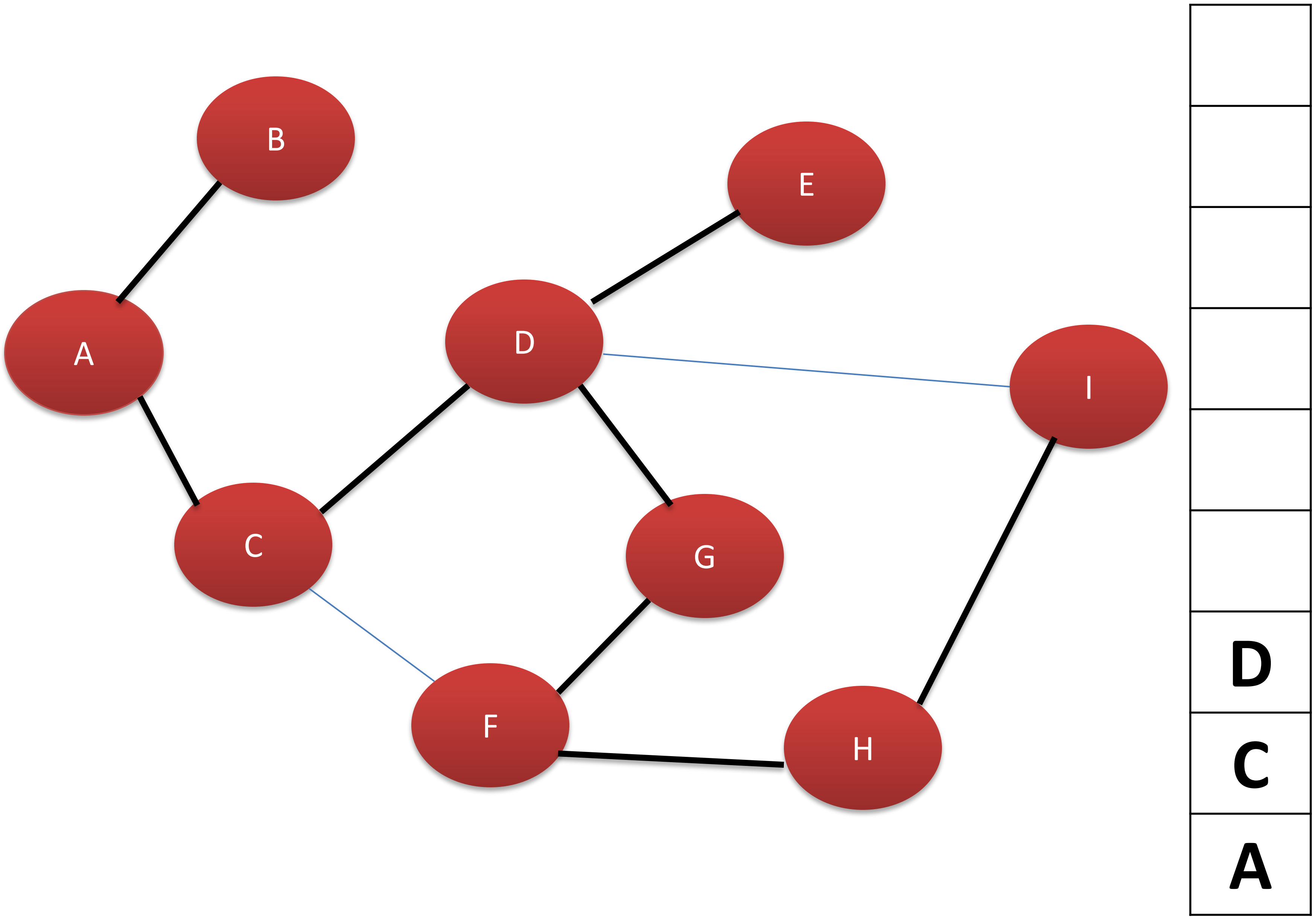
Example



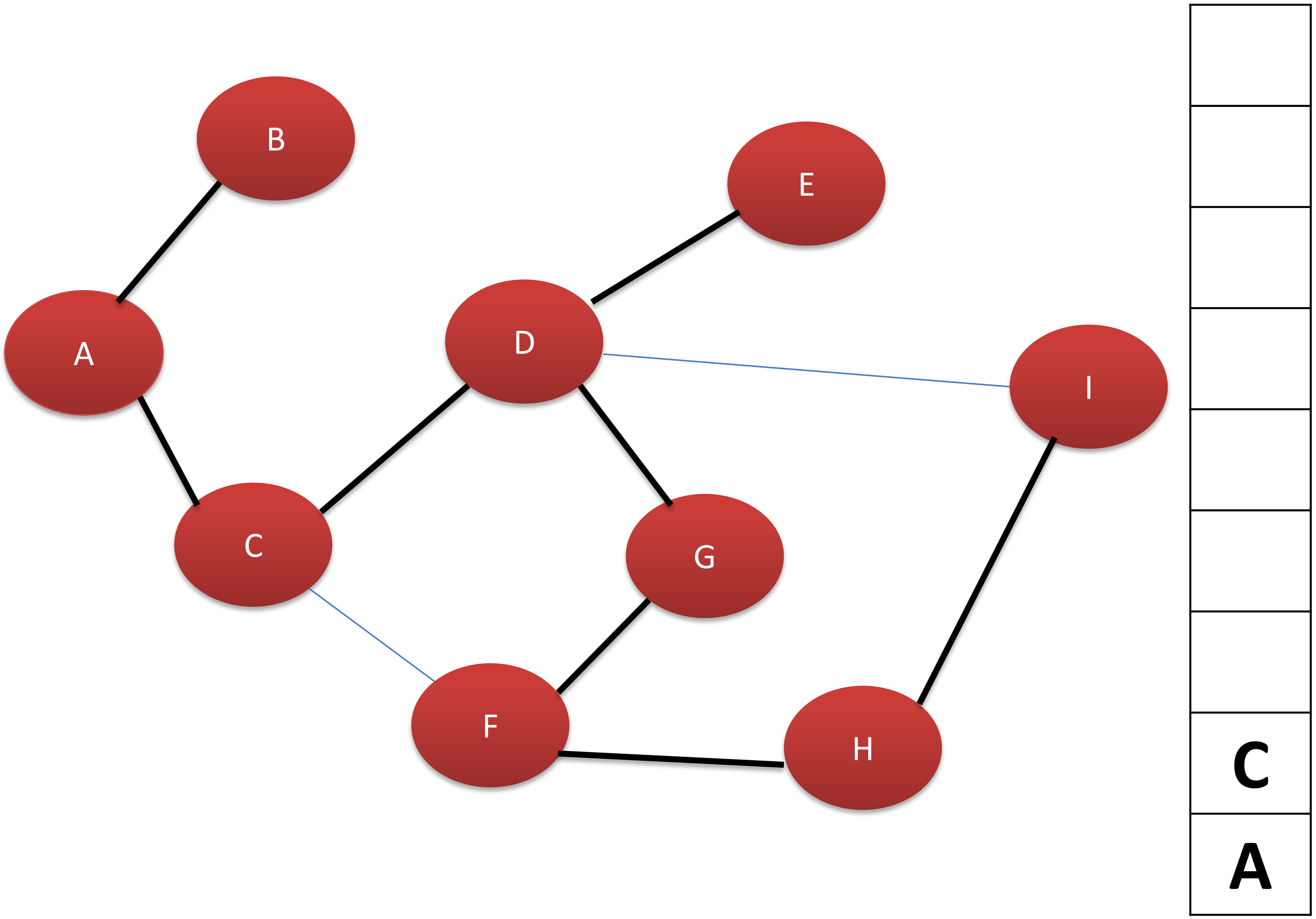
Example



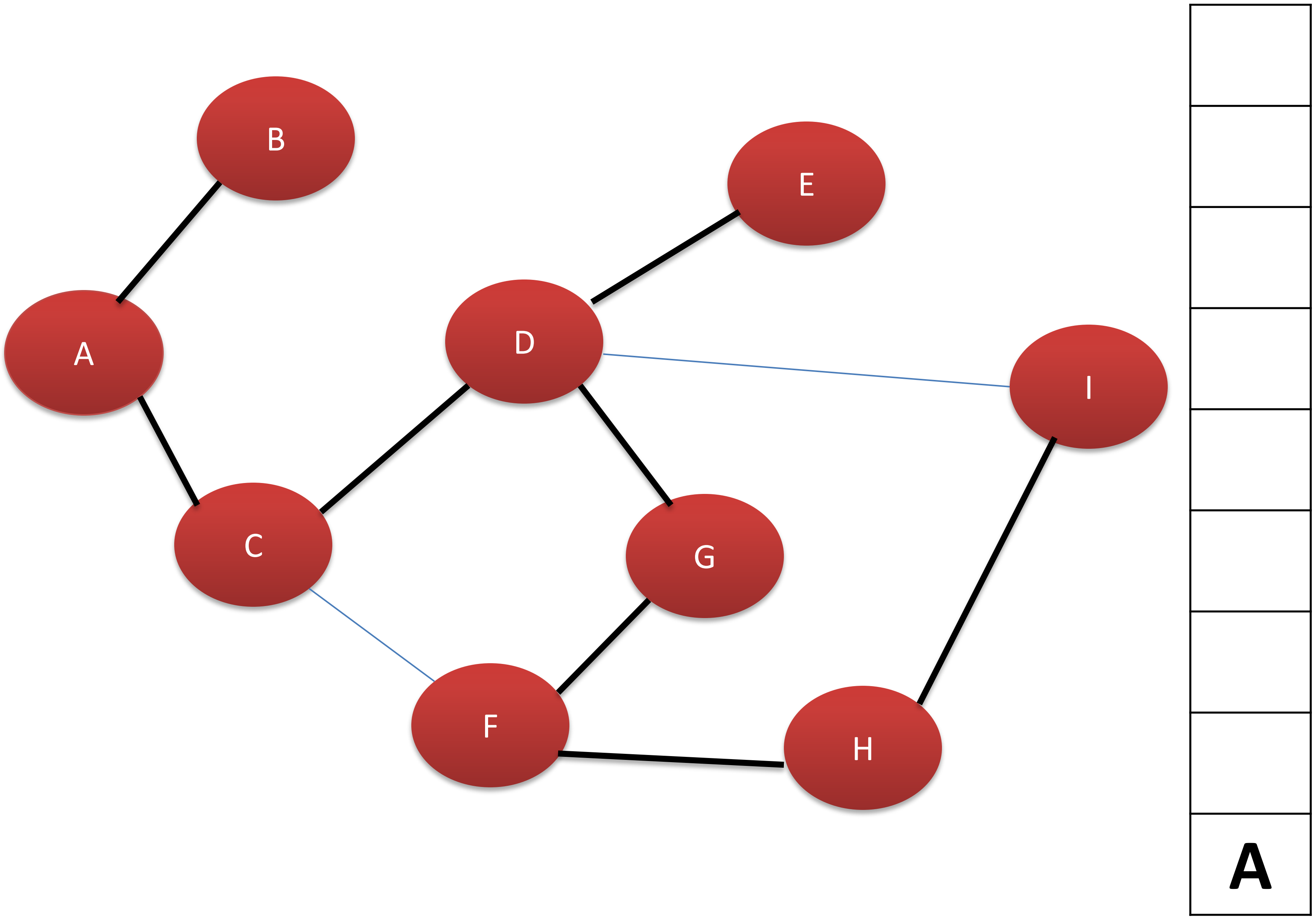
Example



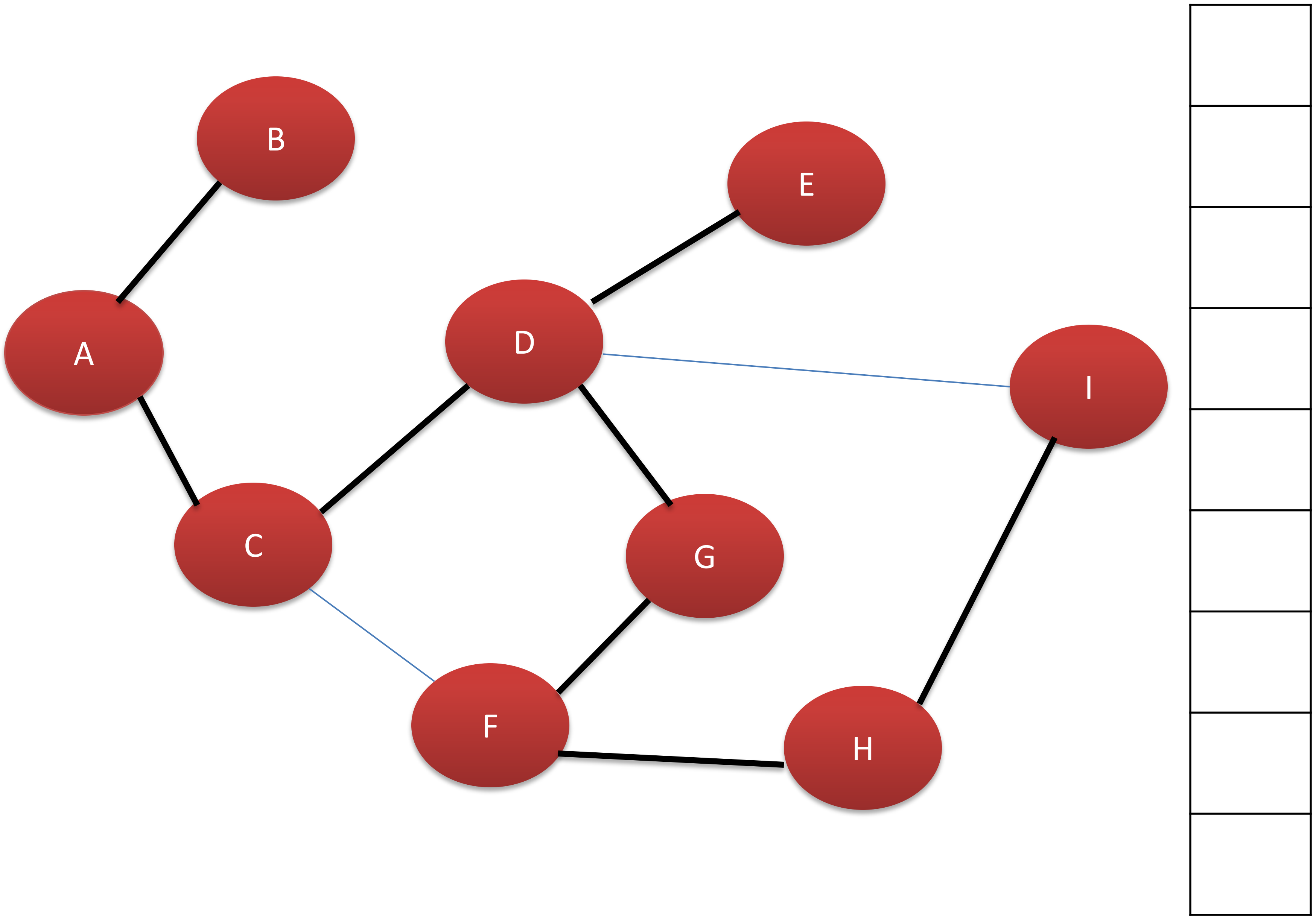
Example

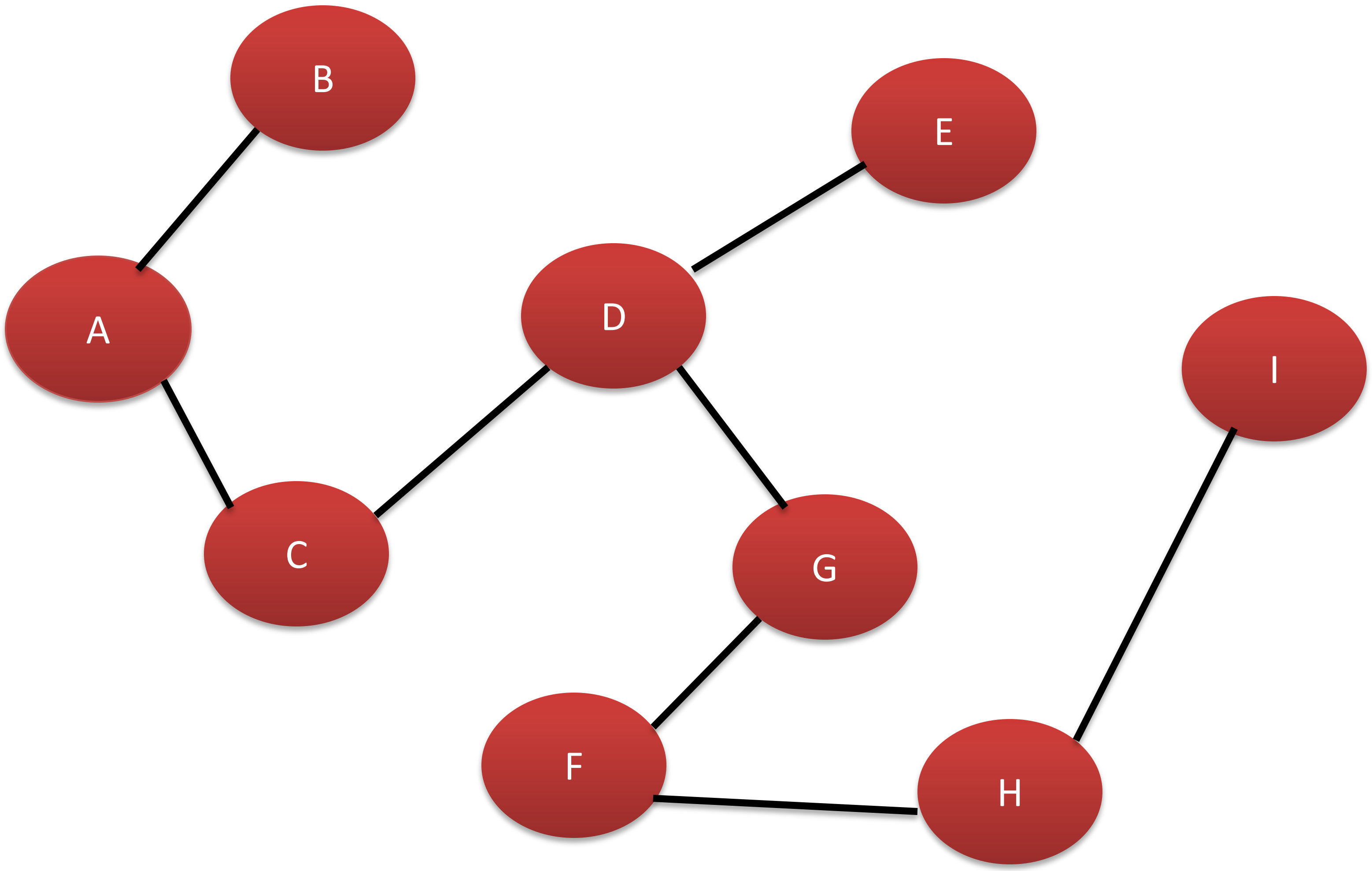


Example



Example





Minimum Spanning Tree

DFS Applications

DFS is useful for

1. Finding a path between two specified nodes u, v of un-weighted graph.
2. Finding a path between two specified nodes u, v of weighted graph.
3. Finding whether a graph is connected or not.
4. Finding spanning tree of a connected graph.

Breadth First Search (BFS)

- **Breadth First Search** (BFS) algorithm traverses a graph in level by level and finally produces a spanning tree.
- **Queue** Data Structure is used to implement BFS.

The following points are used to implement **BFS**

Step 1 - Start Traversing from any one vertex of a graph. Mark it as visited and insert it into the Queue.

Step 2 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 3 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex from queue.

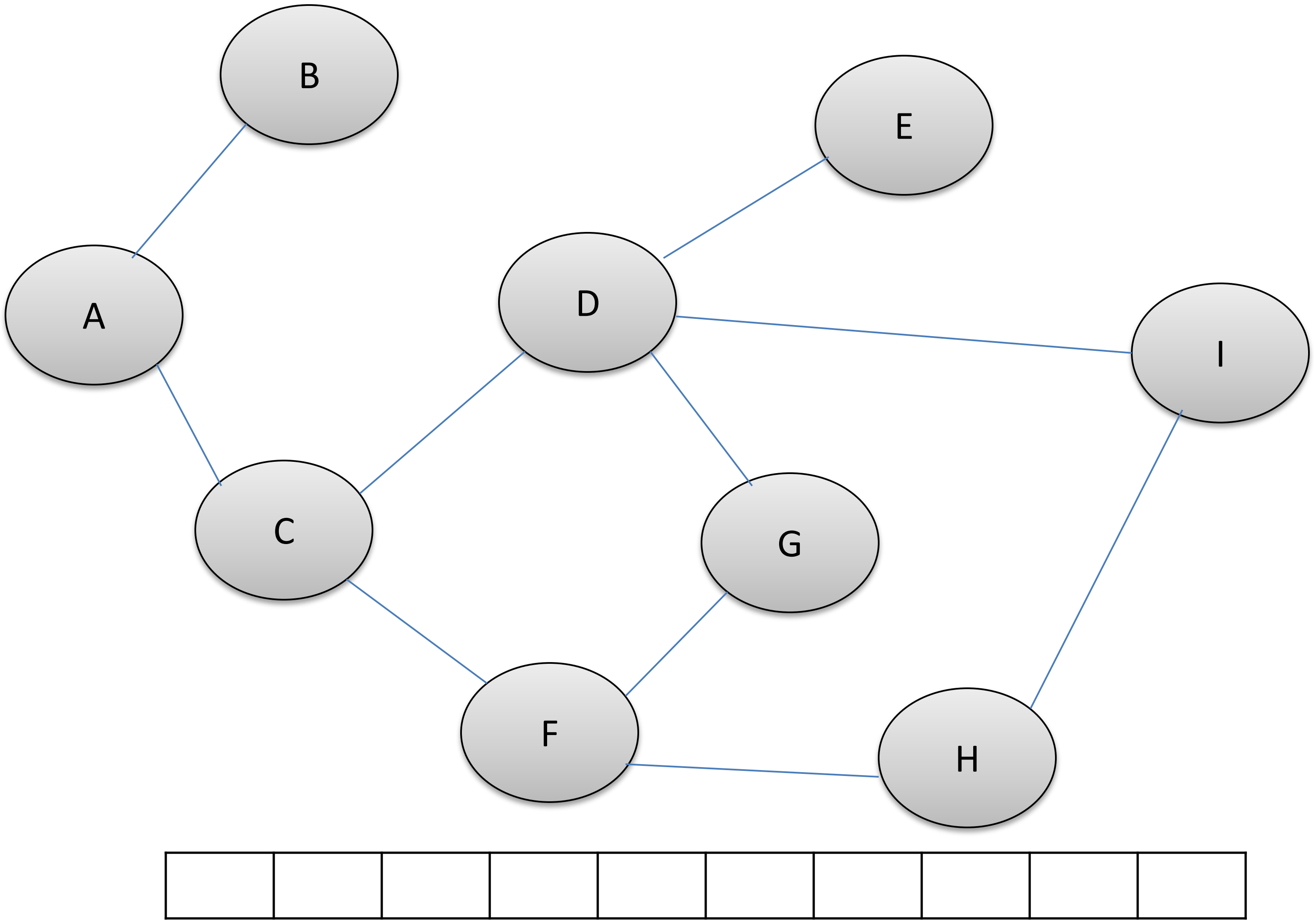
Step 4 - Repeat steps 2 and 3 until queue becomes empty.

Step 5 - When queue becomes empty, then produce final spanning tree.

BFS Algorithm

```
BFS (G, V)
{
    mark s as visited.
    Q.enqueue( V )
    while ( Q is not empty)
    {
          V = Q.dequeue( )
             for all adjacent vertices W of V
             {
                   if W is not visited
                   {
                           mark W as visited.
                       `   Q.enqueue( W )
                   }
             }
      }
}
```

Example



Empty Queue

BFS Applications

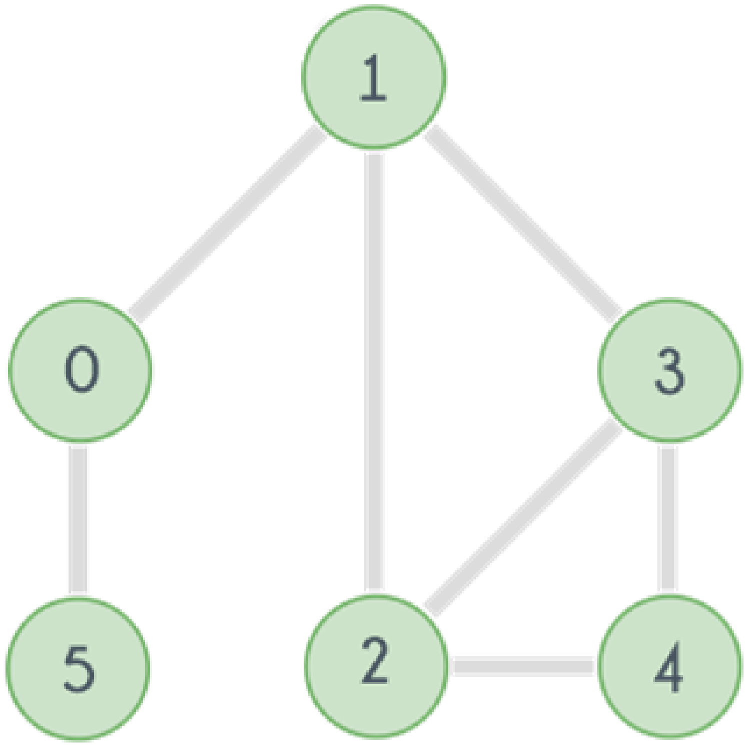
BFS is useful for

1. Finding the shortest path between two nodes u, v of un-weighted graph.
2. Finding the shortest path between two nodes u, v of weighted graph.
3. Finding all connected components in a graph.
4. Finding all nodes within an individual connected component.

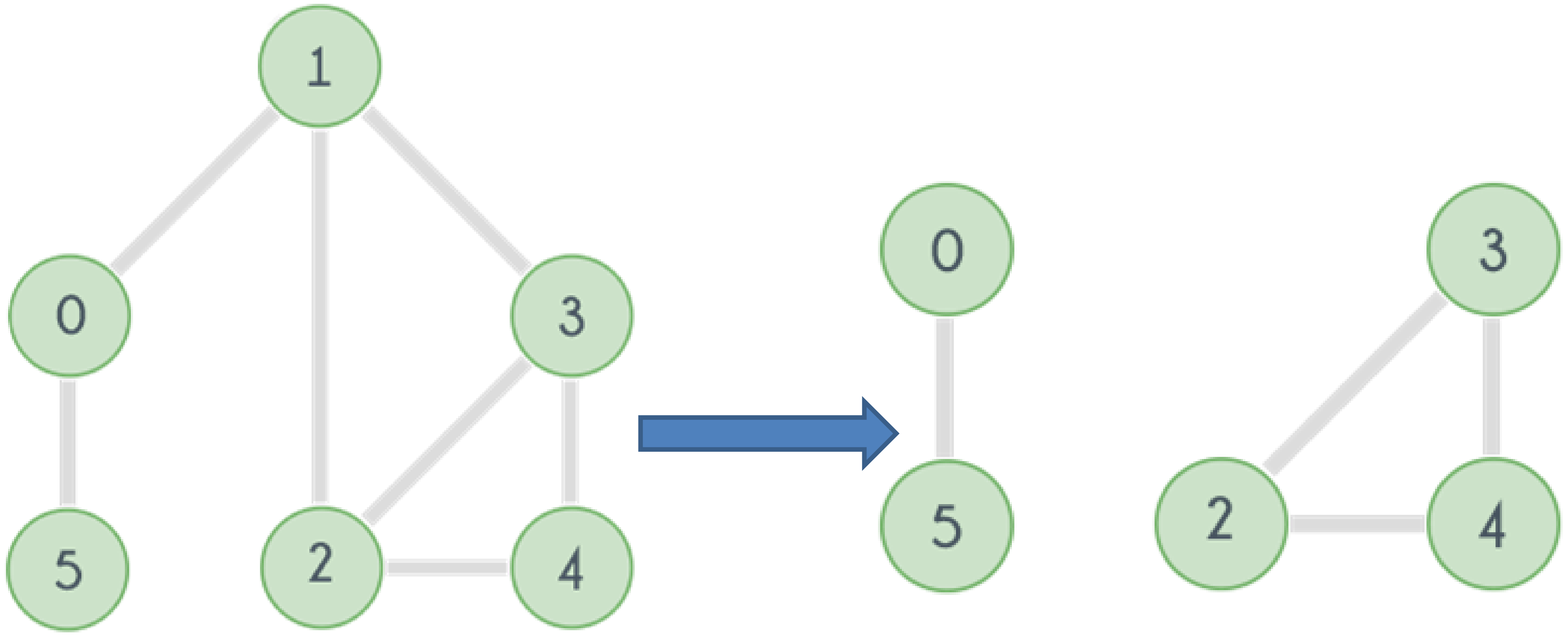
Articulation Vertex or Cut Vertex

- In a graph, a vertex is called as articulation vertex if removing it and all the edges associated with it gives more than one connected components.

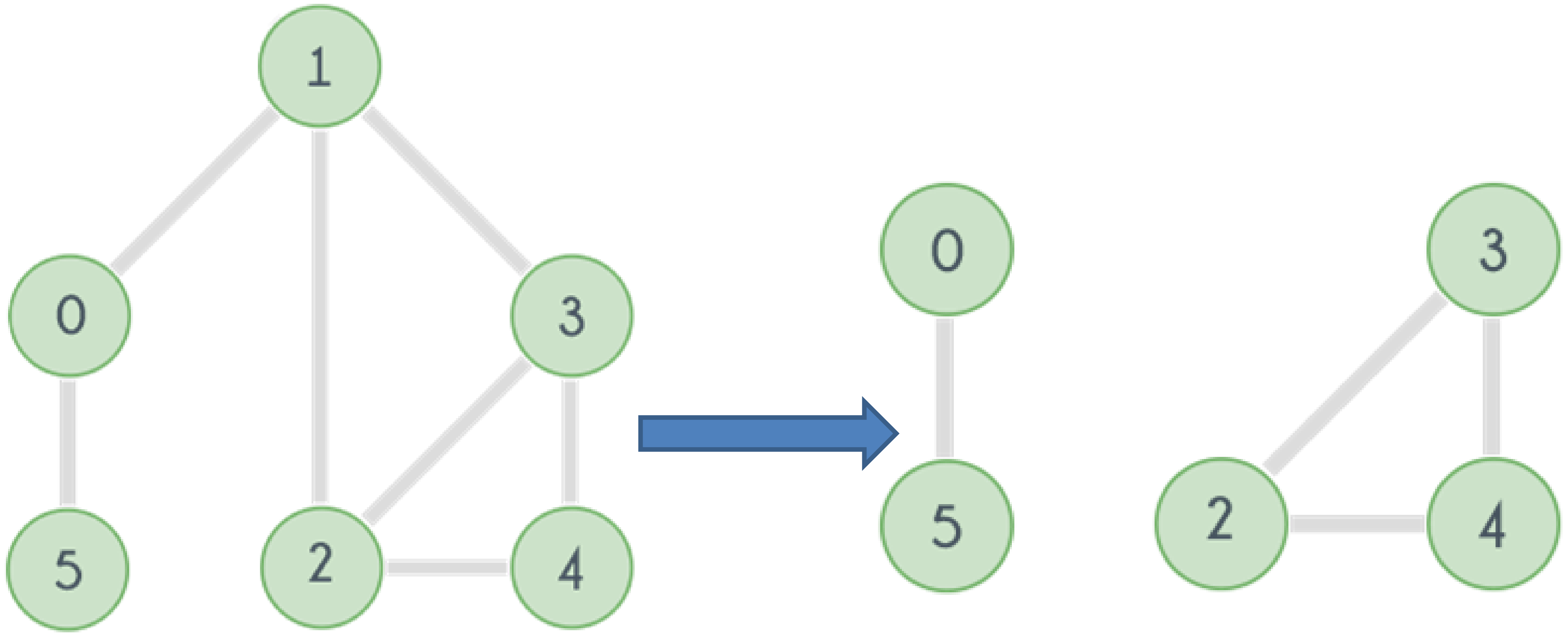
Delete vertex **1**



Now the result is… Two disconnected graphs



So, here vertex 1 is called as Articulation vertex or cut vertex

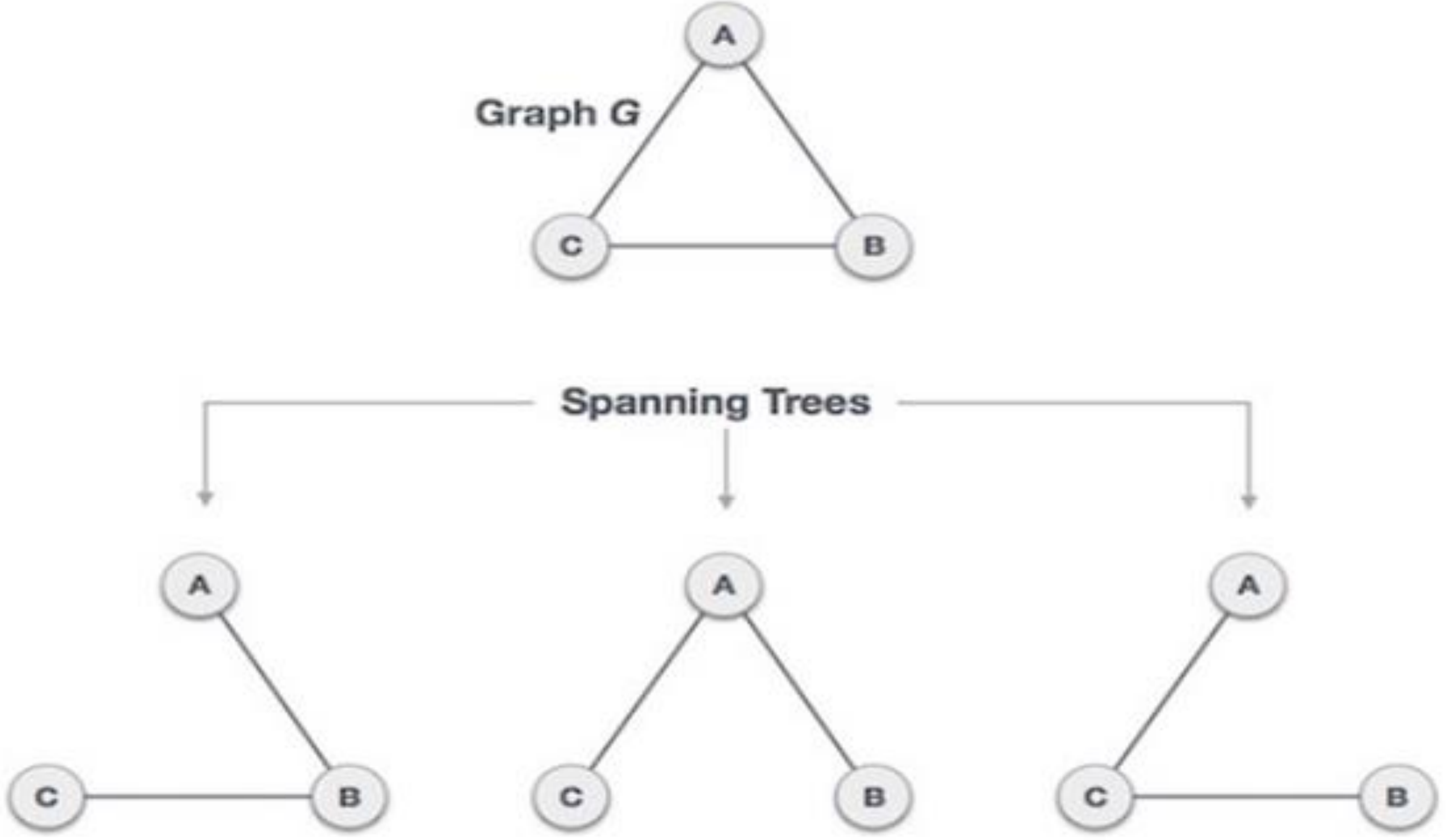


Bi-Connected Graph

- A Graph is called as Bi-connected graph when it doesn’t have articulation vertices.
- A graph with no articulation point called bi-connected.
- In other words, a graph is bi-connected if and only if any vertex is deleted, the graph remains connected.

Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices and does not have cycles.
- A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes.



n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected.
- Adding one edge to the spanning tree will create a circuit or loop.

Application of Spanning Tree

- Spanning tree is basically used to find a minimum path to connect all nodes in a graph.

  - **Network Planning**
  - **Computer Network Routing Protocol**
  - **Cluster Analysis**

Minimum cost Spanning Tree (MST)

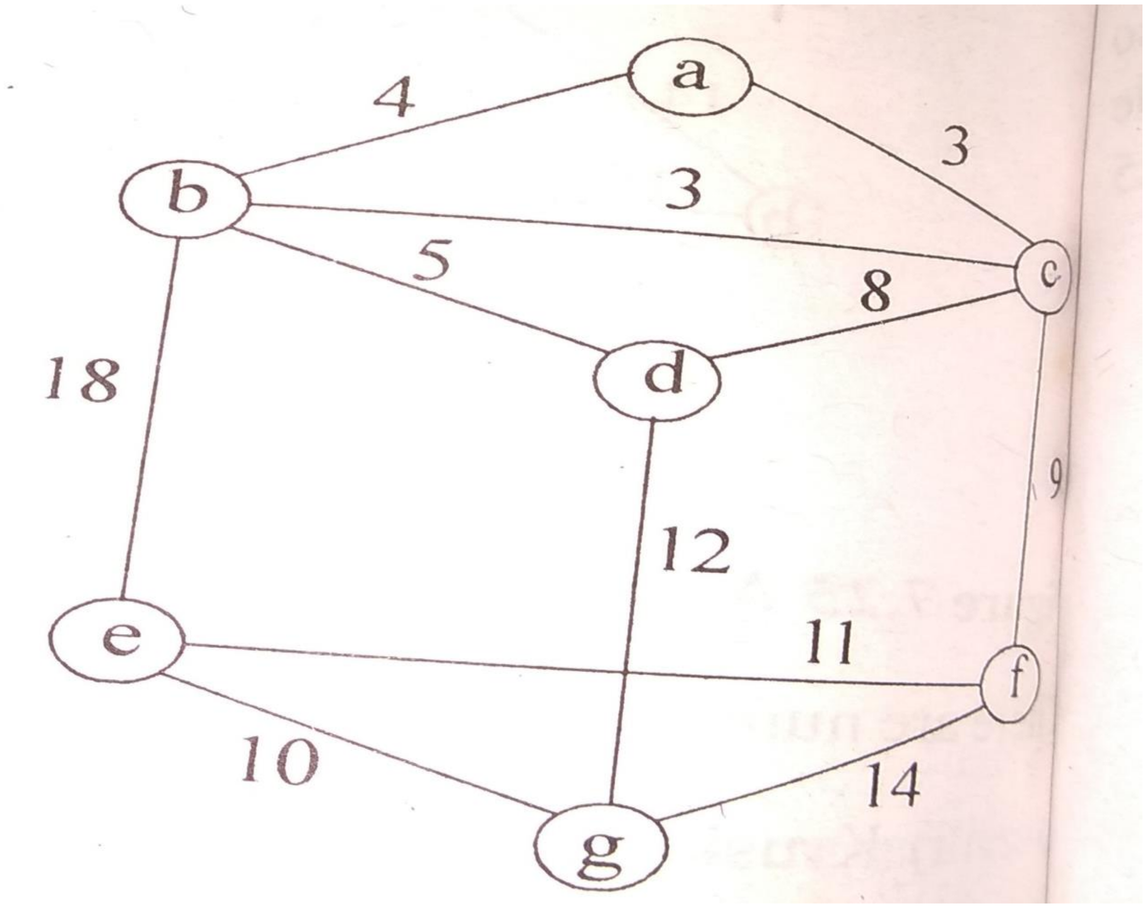
- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

- Minimum Spanning-Tree Greedy Algorithms
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Sollin’s Algorithm

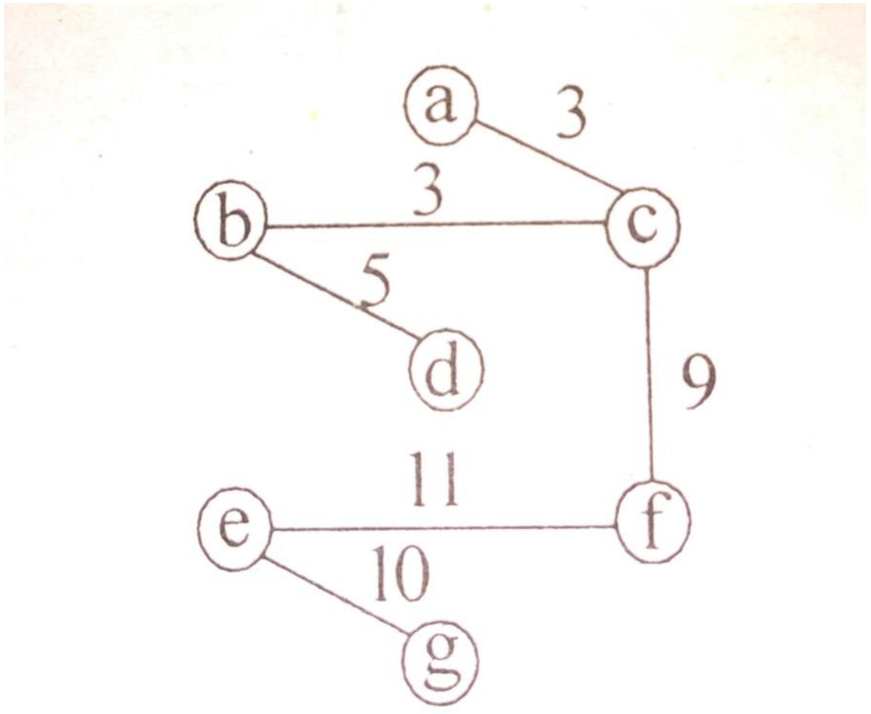
Kruskals MST

- Kruskals greedy algorithm is used to find the minimum cost spanning tree by select and inserting edge to the graph in ascending order without form a cycle.

sample weighted undirected graph



Minimum Cost Spanning Tree by Kruskals



Kruskals Algorithm

Kruskals Algorithm

Let G be a graph having N vertices and E edges.

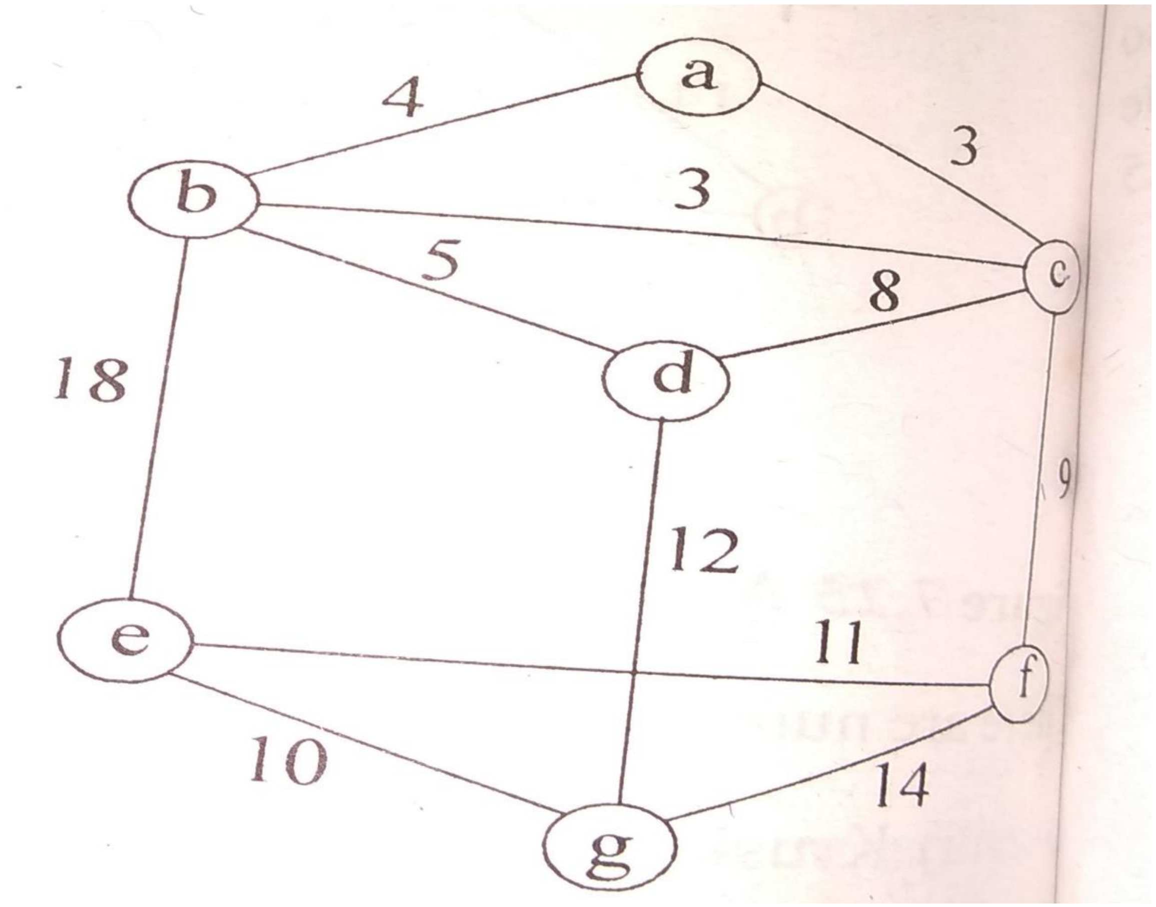
```
Kruskals()
{
    Set T={} // empty tree
    while(|T|<N and E≠ empty)
   {
      select an edge (u,v) from E  of lowest weight
      delete (u,v) from E.
      if (u,v) doesn't create a cycle in T then
      {
          set T=T union (u,v)
      }
   }
    if |T|<N-1  then
           no spanning trees
    return T
}
```

Prims MST

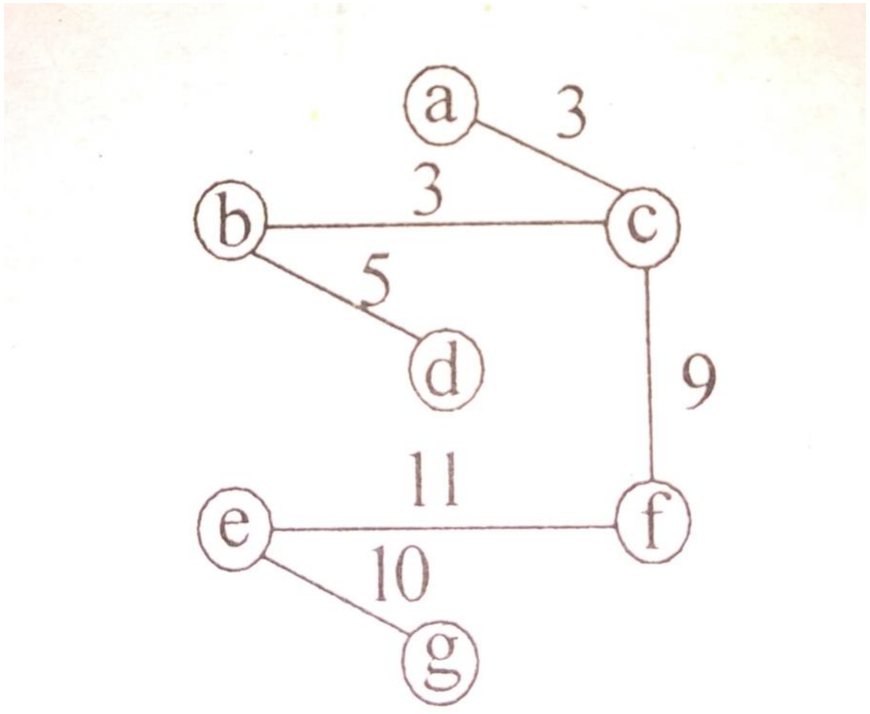
- Prims greedy algorithm is used to find the minimum cost spanning tree of an weighted , connected, undirected graph.

- In Prim's algorithm, a least-cost edge (u, v) is added to T such that T∪ {(u, v)} is also a tree. This repeats until T contains n-1 edges.

sample weighted undirected graph



Minimum Cost Spanning Tree by Prims Algorithm



Prim's Algorithm

```
//Assume that G has at least one vertex
T={};   // empty tree
TV = {};     //start with any vertex  and no edges

while (T contains less than n)
    {
       Let (u,v) be a  least-cost edge such that u in TV and v not in TV;
       If (there is no such edge)
                 break;
        add v to TV;
        add (u,v) to T)
  }

If (T contains less than n-1 edges)
    then "no spanning tree"
Return T
```

Kruskal's vs Prims

- Prim’s algorithm initializes with a node, whereas Kruskal’s algorithm initiates with an edge.
- In prim’s algorithm, graph must be a connected graph while the Kruskal’s can function on both connected and disconnected graphs too.
- Prim’s algorithm has a time complexity of $O(V^2)$, and Kruskal’s time complexity is $O(\log V)$.

Sollin’s Algorithm

- Sollin’s algorithm is also called **Boruvka’s** algorithm.
- It was the first algorithm to find the MST.
- Boruvka’s Algorithm is a greedy algorithm and is similar to Kruskal’s algorithm and Prim’s algorithm.

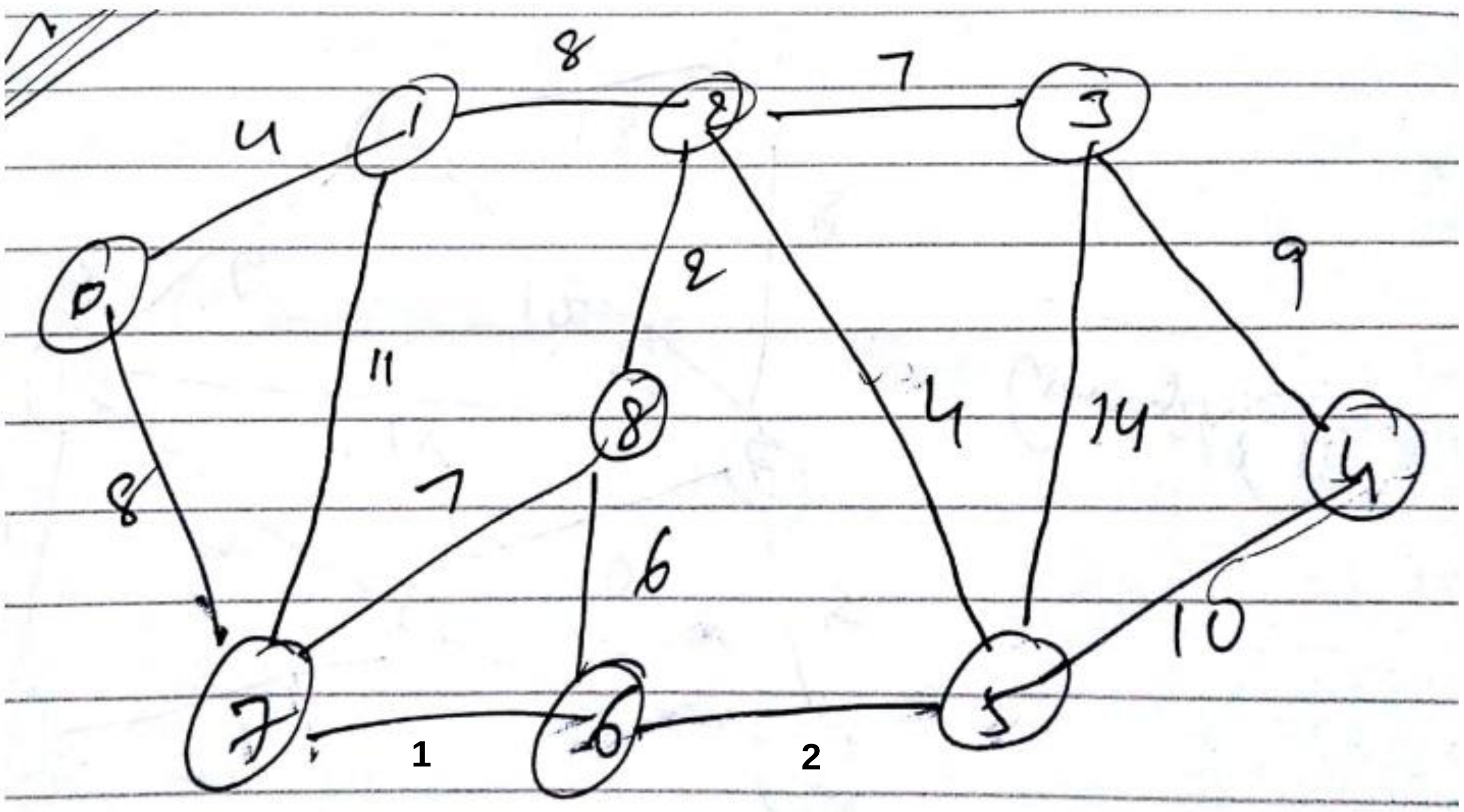
Sollin’s Algorithm

1. Write all the vertices of a connected graph.
2. Select the min cost outgoing edge of all vertices of a graph.
3. Highlight the separate sets of connected components.
4. While there are more than one components, do following for each component.

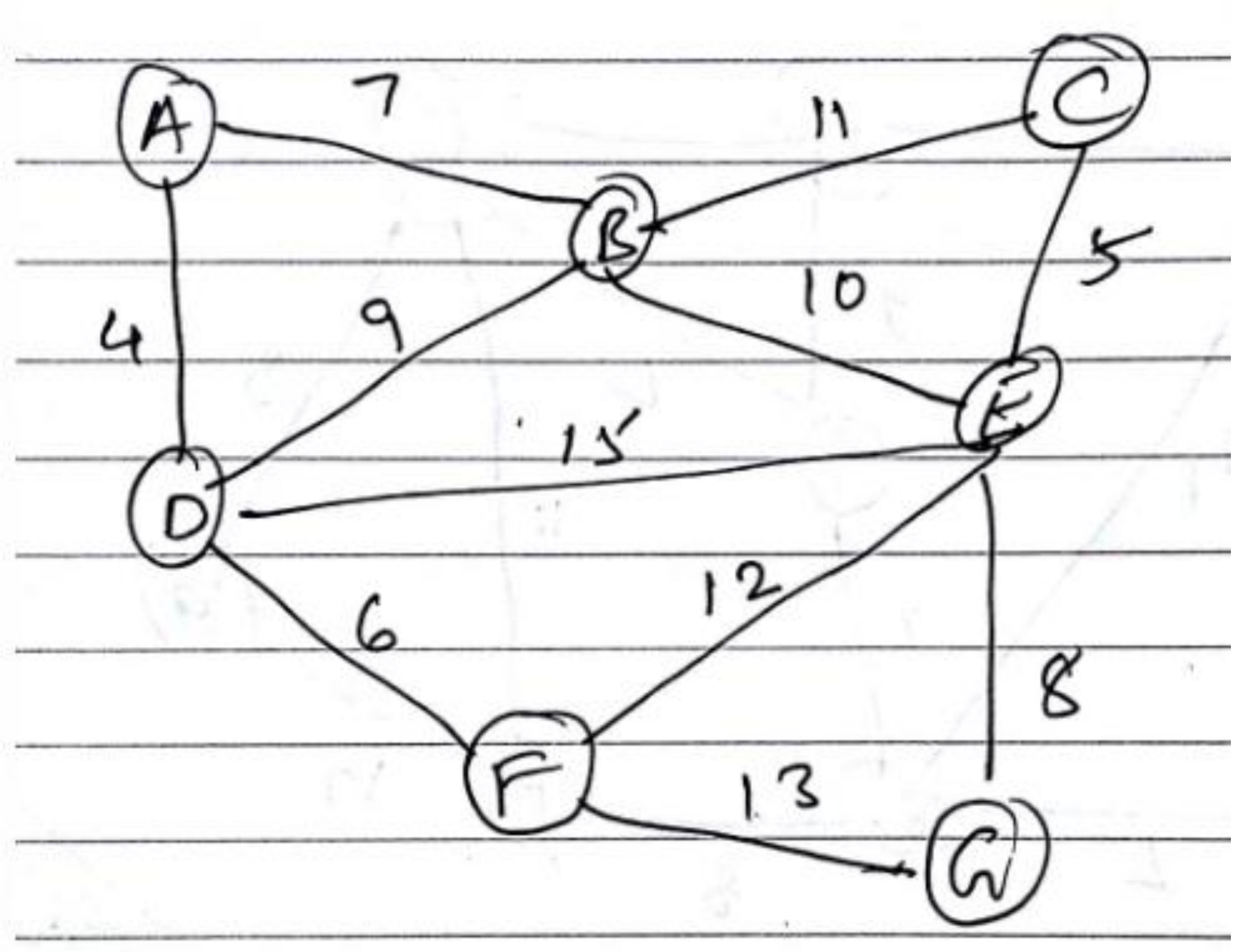
   a) Find the one component of min weight edge that connects with to any other component.

   b) Add this min cost edge to MST if not already added.
5. Repeat step 4 until all vertices are in single component.
6. Return MST.

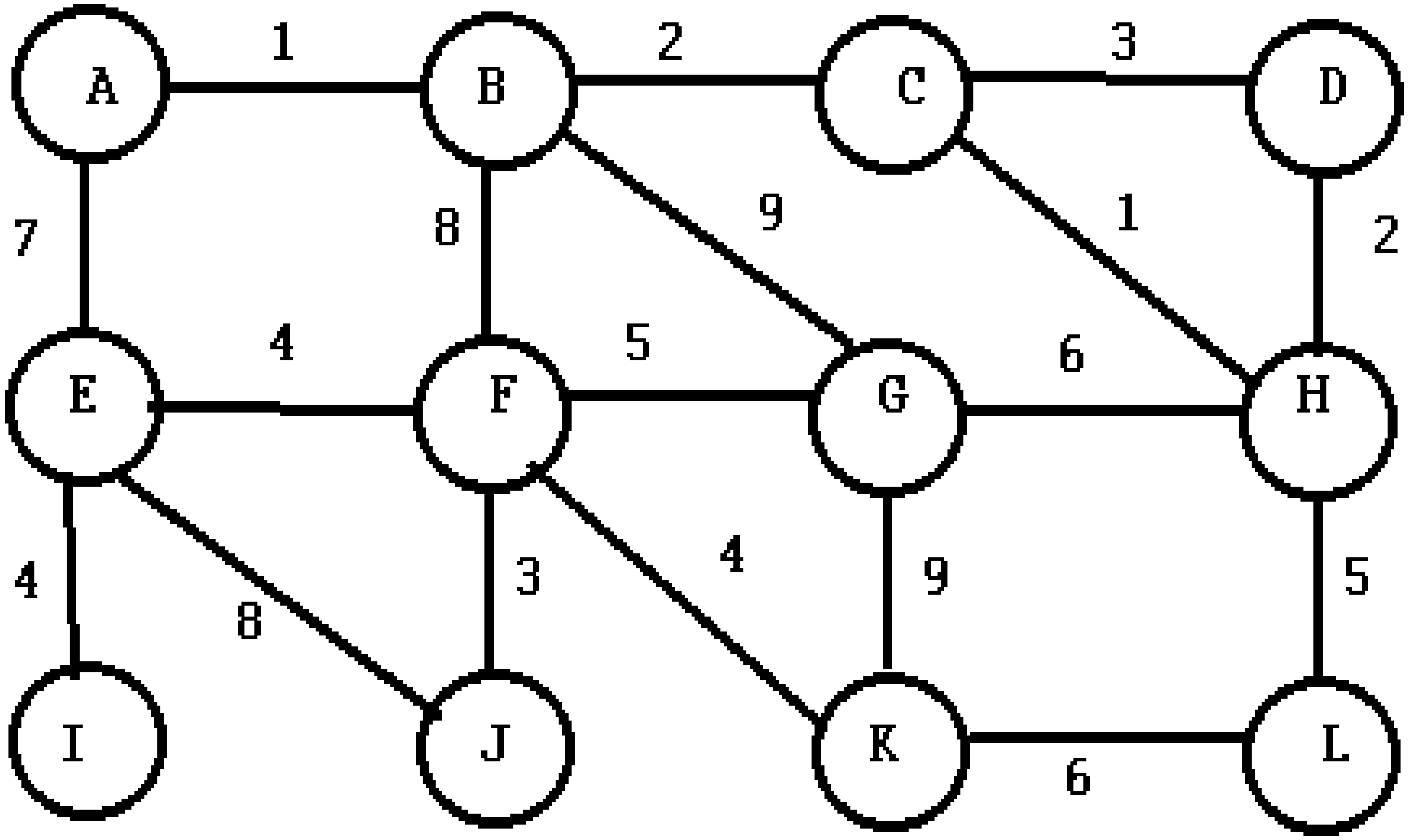
Example 1



Example 2



Example 3



Shortest Path

- It is the process of finding a **shortest path between two vertices** in a graph such that the total sum of the edge weights is minimum.
- There are two different

shortest path problems...

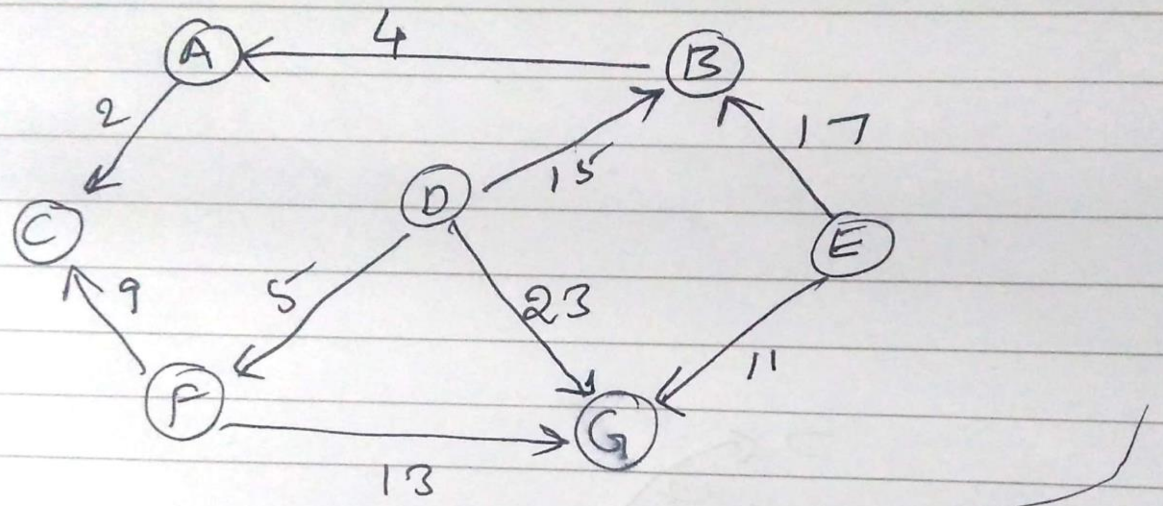
  – 1. Single Source Shortest Path
  – 2. All pairs Shortest Path

E
2 Mins
A
15 Mins
D
4 Mins
4 Mins
4 Mins
4 Mins
C
B
Hyde Park
Kensington Gardens
high street
south kensington
gloucester road
knight's bridge

Single Source Shortest Path

- Single Source Shortest Path is to **finding the shortest path from one vertex to any other vertices** of a graph.
- There are two techniques...
  - 1. Dijkstra's Algorithm
  - 2. Bellman Ford Algorithm

Single Source Shortest Path :



for Ex
finding shortest Path from (A) to all other vertices of a graph.

i.e

(A) $\longrightarrow$ (B)	cost	is min
(A) $\longrightarrow$ (C)	cost	"
(A) $\longrightarrow$ (D)	cost	"
(A) $\longrightarrow$ (E)	cost	"
(A) $\longrightarrow$ (F)	cost	"
(A) $\longrightarrow$ (G)	cost	"

All pairs Shortest Path

- All Pairs Shortest Path is to **finding the shortest path between pair of vertices** of a graph.
- The technique is...

    1. Floyd-Warshall Algorithm

All pairs shortest path

$\Big\downarrow$ find

$$\begin{tikzcd}
A \arrow[r, "3"] \arrow[d, "3"'] & B \arrow[d, "1"] \\
C \arrow[ur, "3"] & D \arrow[ul, "2"] \arrow[l, "5"]
\end{tikzcd}$$

$$\begin{tikzcd}
& B \\
\text{min cost from } A \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"'] & C \\
& D
\end{tikzcd}$$

$$\begin{tikzcd}
& A \\
\text{min cost from } B \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"'] & C \\
& D
\end{tikzcd}$$

$$\begin{tikzcd}
& A \\
\text{min cost from } C \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"'] & B \\
& D
\end{tikzcd}$$

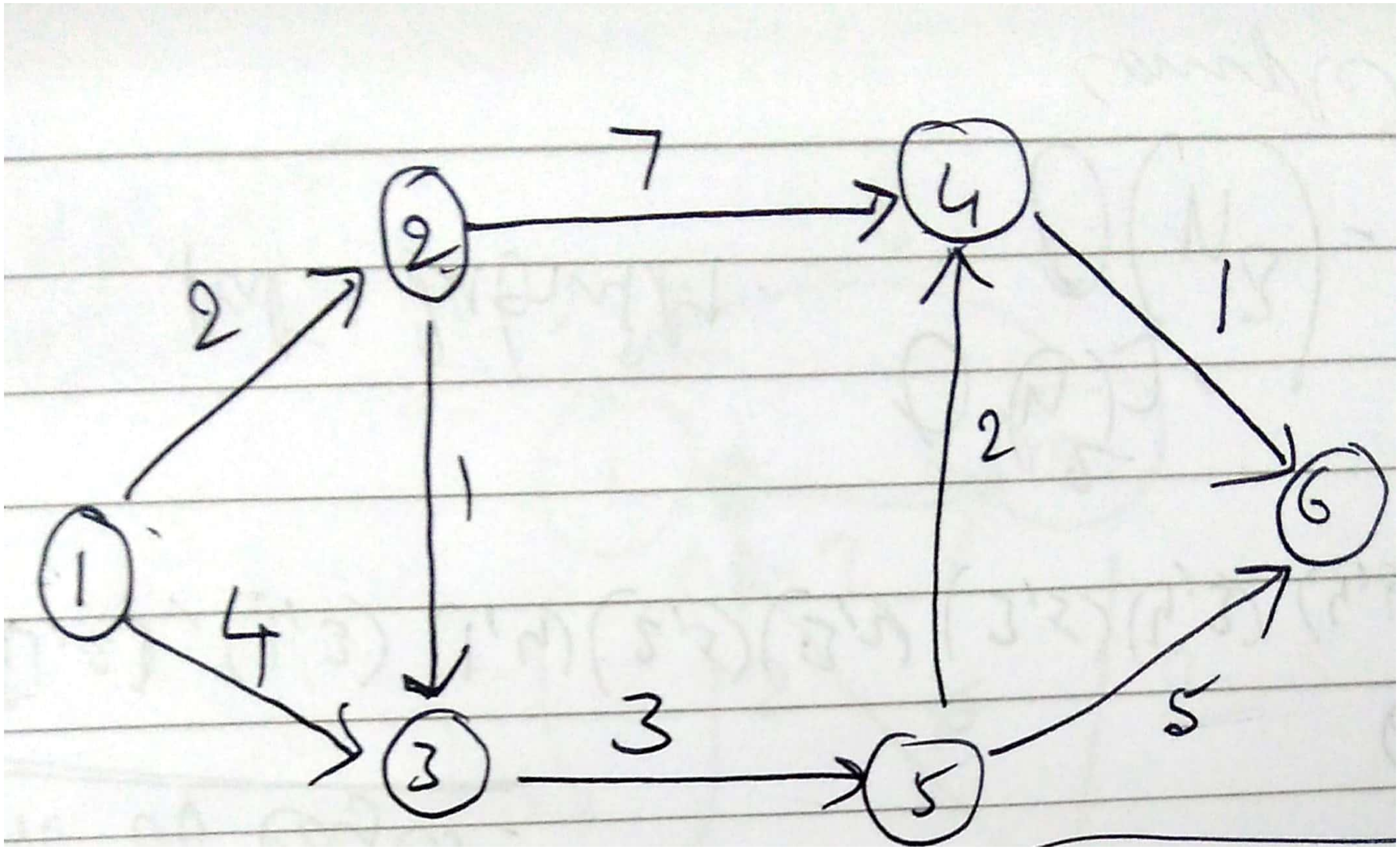
$$\begin{tikzcd}
& A \\
\text{min cost from } D \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"'] & B \\
& C
\end{tikzcd}$$

Dijkstra's algorithm

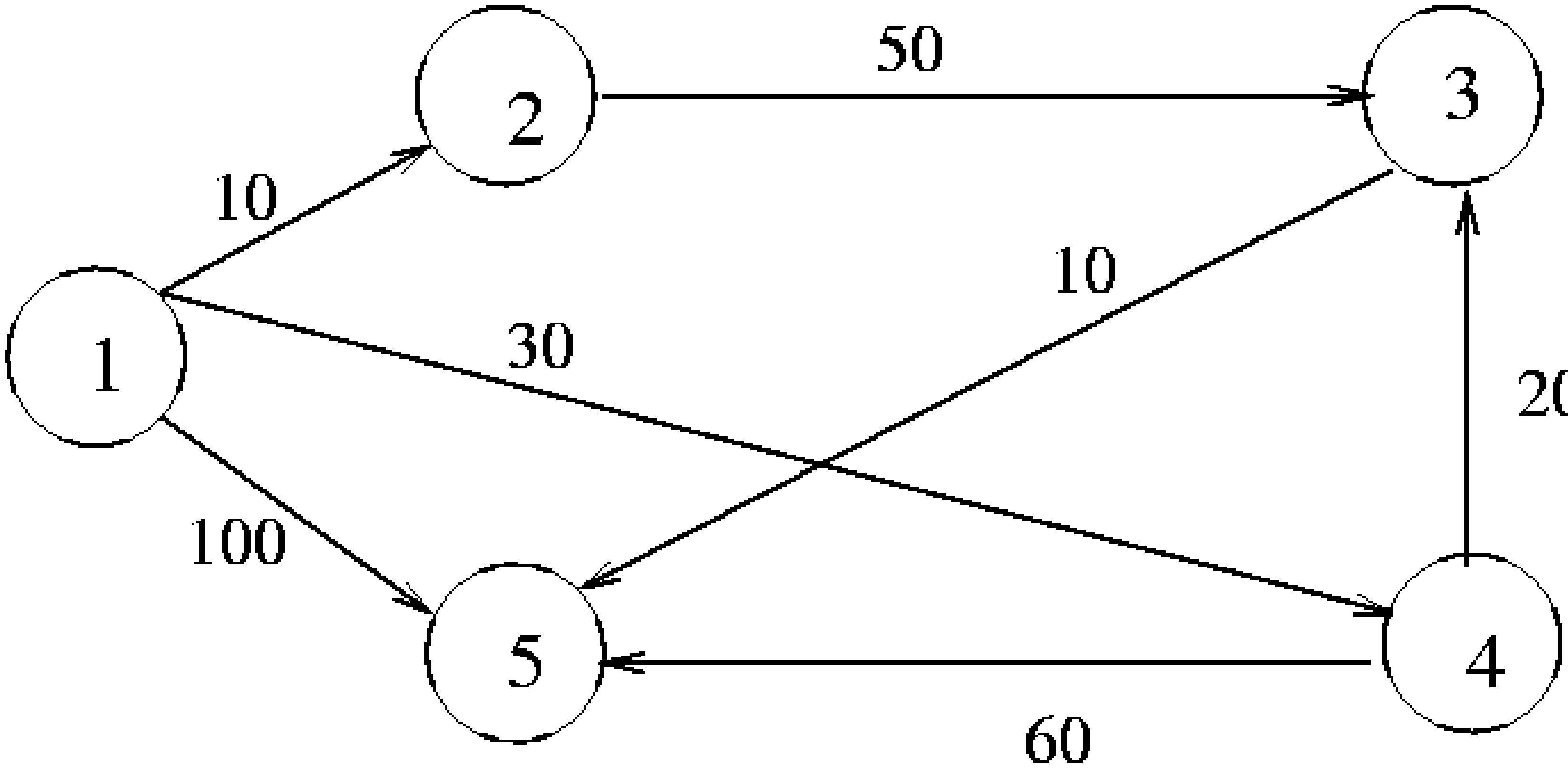
- **Dijkstra’s algorithm** is to find the shortest paths from the source vertex to all other vertices in the graph.
- **Dijkstra’s algorithm** can be used to determine the shortest path from one node in a graph to *every other node* within the same graph.
- The time complexity for the matrix representation is **O(V^2).**

https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e

Example 1



Example 2



Advantages & Disadvantages

Advantages:

- To finding shortest paths between vertices of graph.

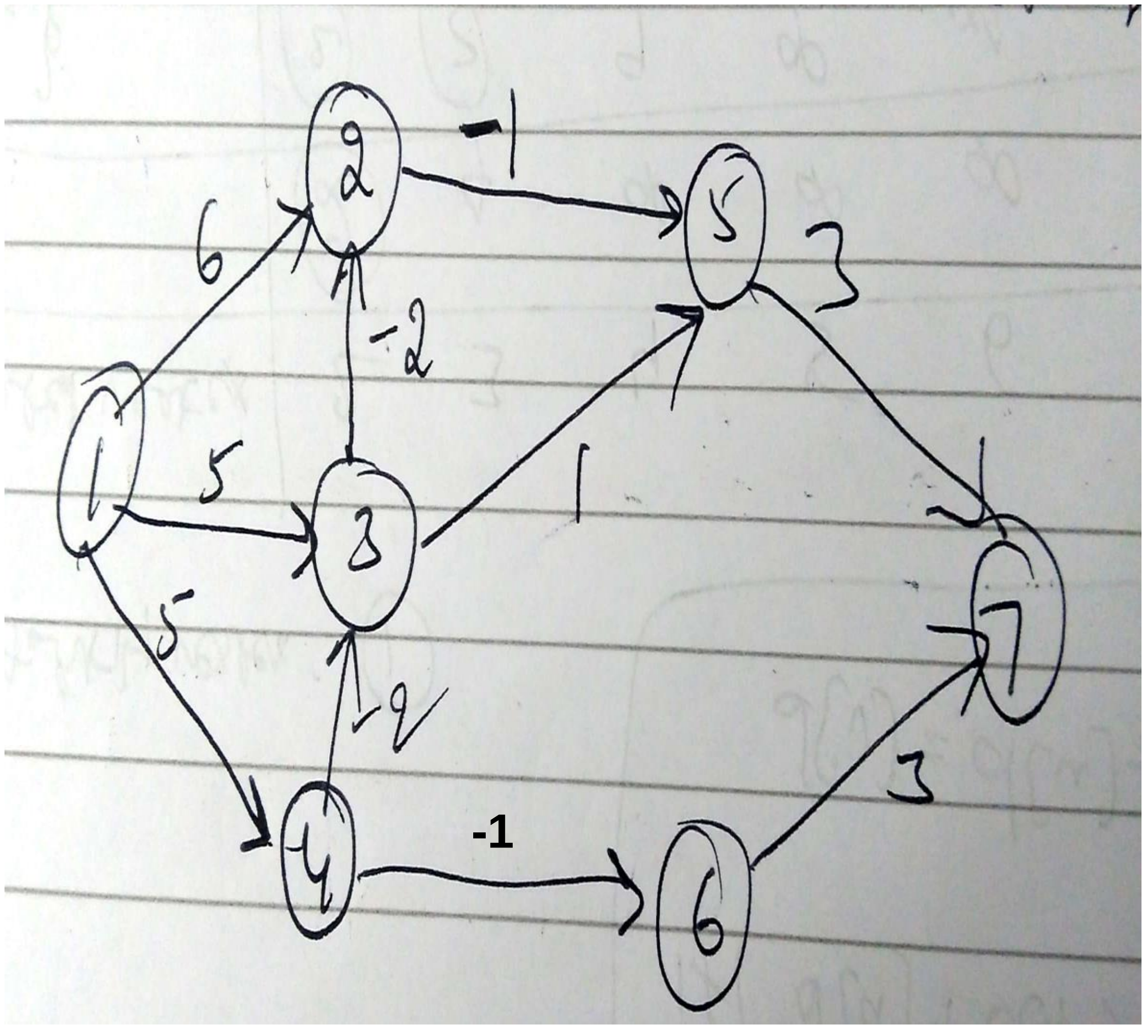
Disadvantage

It will not work on a graph which is having **Negative cost edges. (**It is implemented by Bellman Ford Algorithm**)**

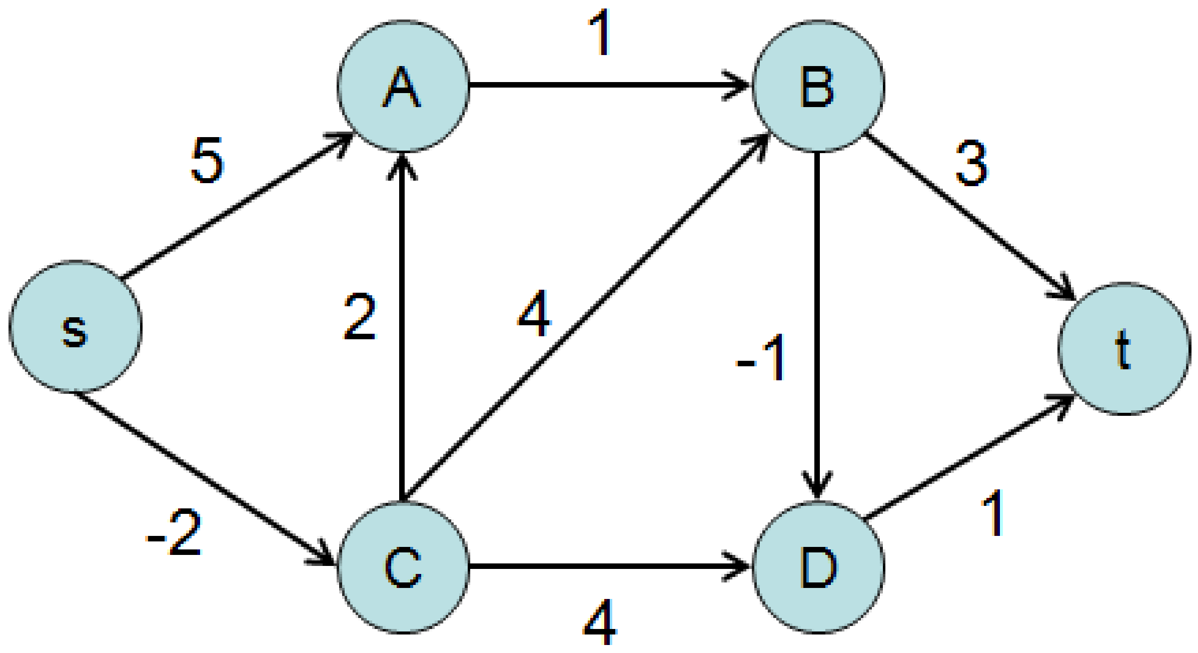
Bellman Ford's algorithm

- **Bellman Ford's algorithm** is used to find the shortest paths from the source vertex to all other vertices in a weighted graph.
- It depends on the following concept: Shortest path contains at most n−1 edges, because the shortest path couldn't have a cycle.
- Time Complexity of **Bellman-Ford** is **O(V*E)**, where V is the number of vertices and E is the number of edges in the graph.

Example 1



Example 2



Advantages & Disadvantages

Advantages:

- To finding shortest paths between vertices of graph.

Disadvantage

It will not work on a graph which is having **Cycle of Negative cost edges.**

All-Pairs Shortest Path

- **All-Pairs Shortest Path** is used to find the shortest paths between all pairs of vertices in the graph.

- The Technique is **Floyd–Warshall's Algorithm.**

All pairs shortest path

$$\begin{tikzcd}
A \arrow[r, "3"] \arrow[d, "3"'] & B \arrow[d, "1"] \\
C \arrow[ur, "3"] & D \arrow[l, "5"] \arrow[ul, "2"]
\end{tikzcd}$$

$\Big\downarrow$ find

$$\begin{tikzcd}
 & B \\
\text{min cost from } A \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"] & C \\
 & D
\end{tikzcd}$$

$$\begin{tikzcd}
 & A \\
\text{min cost from } B \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"] & C \\
 & D
\end{tikzcd}$$

$$\begin{tikzcd}
 & A \\
\text{min cost from } C \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"] & B \\
 & D
\end{tikzcd}$$

$$\begin{tikzcd}
 & A \\
\text{min cost from } D \arrow[ur, "to"] \arrow[r, "to"] \arrow[dr, "to"] & B \\
 & C
\end{tikzcd}$$

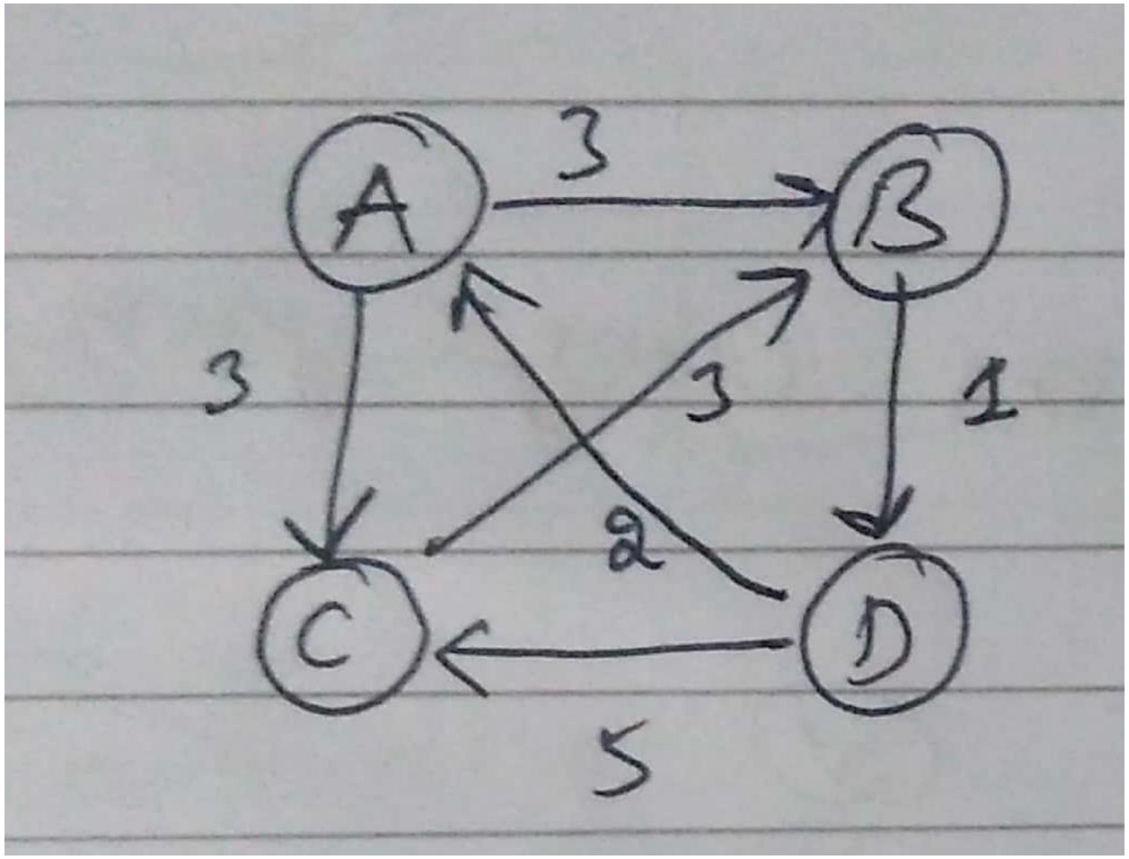
Floyd–Warshall's Algorithm

- Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative.
- Advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O(V3), where V is the number of vertices in a graph.

Algorithm Steps:

1. For a graph with N vertices:
2. Initialize the shortest paths between any 2 vertices with Infinity.
3. Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1intermediate vertex and so on.. until using all N vertices as intermediate nodes.
4. Minimize the shortest paths between any 2 pairs in the previous operation.
5. For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first Knodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

Example 1



Floyd-Warshall Algorithm

```
for(int k = 1; k <= n; k++)
{
   for(int i = 1; i <= n; i++)
   {
     for(int j = 1; j <= n; j++)
     {
       dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
     }
   }
}
```

Time Complexity is O(N3)

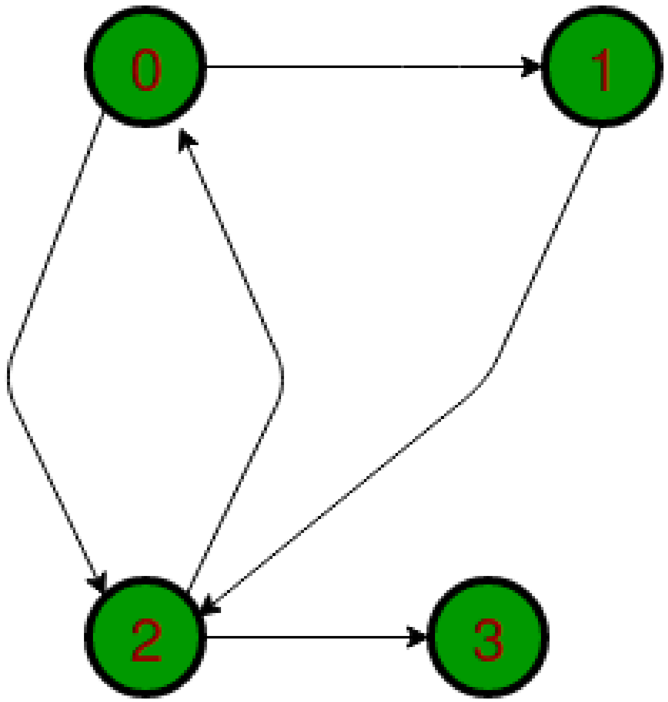
Transitive Closure of Graph
(or)
Reachability Matrix

Definition: Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph.

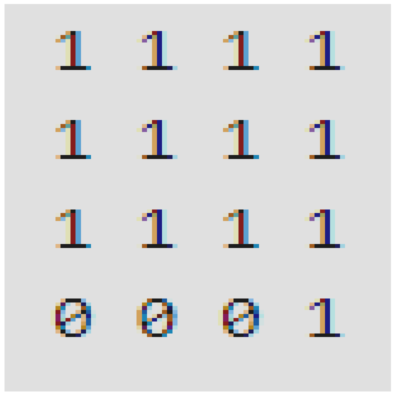
Here reachable mean that there is a path from vertex i to j.

The reach-ability matrix is called transitive closure of a graph.

Example 1

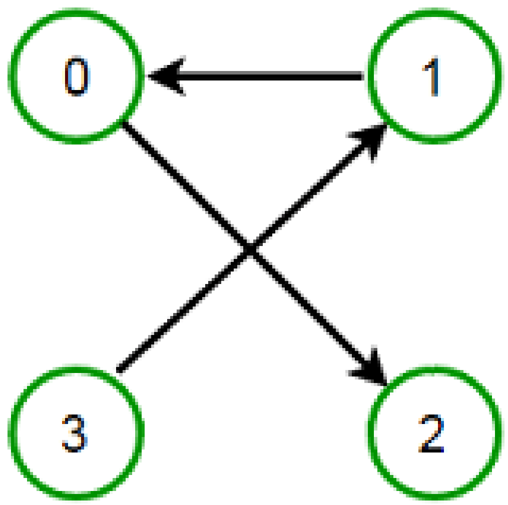


Transitive closure of above graphs is



```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

Example 2



Transitive closure of above graphs is

```
1 0 1 0
1 1 1 0
0 0 1 0
1 1 1 1
```