CSC10004 – Data Structures and Algorithms

Session 07
Graph Structure

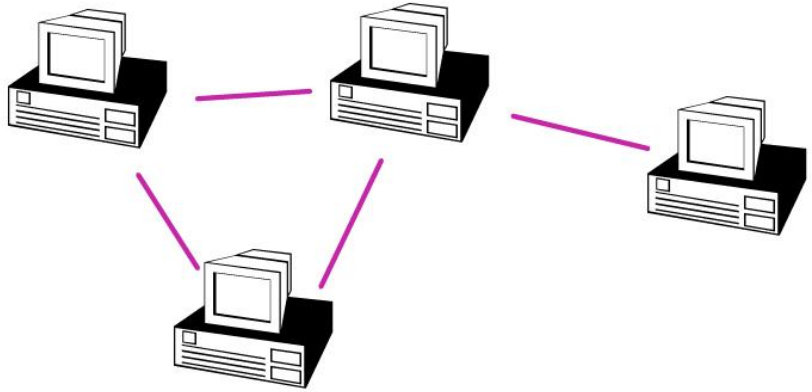
Instructors:
Dr. Lê Thanh Tùng

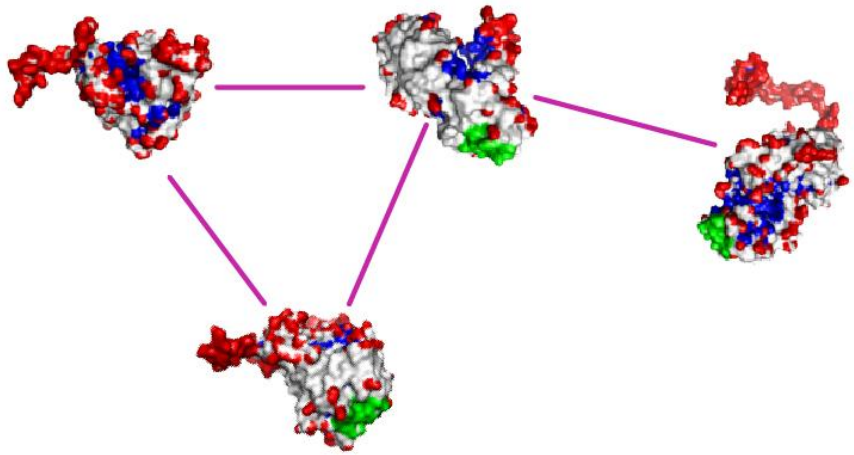
Content

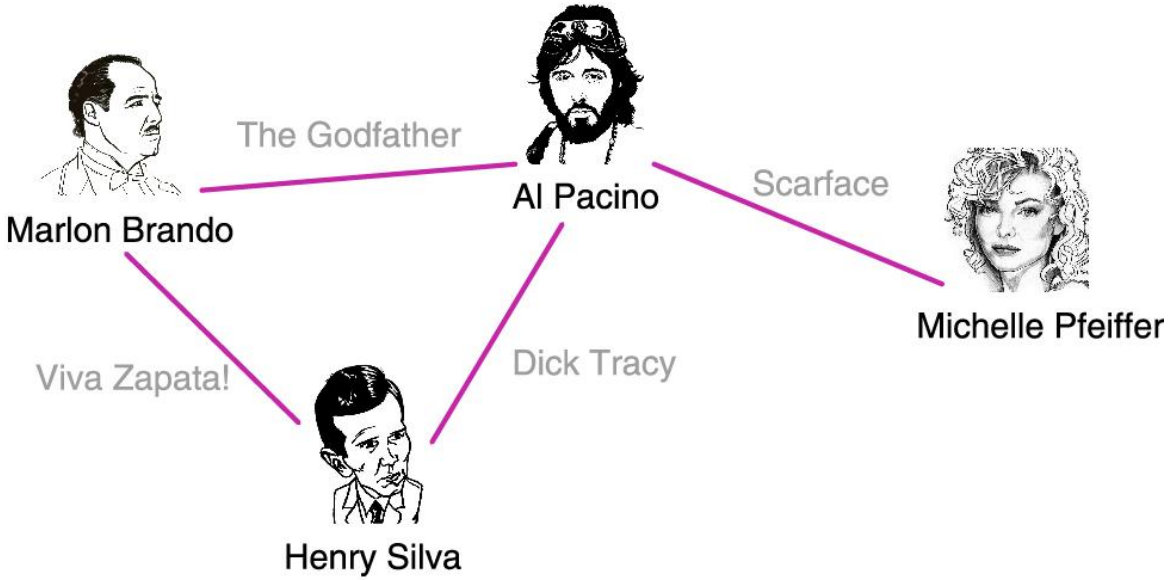
1. Terminologies
2. Graph Representation
3. Graph Traversal
4. Spanning Tree

Terminologies

Examples



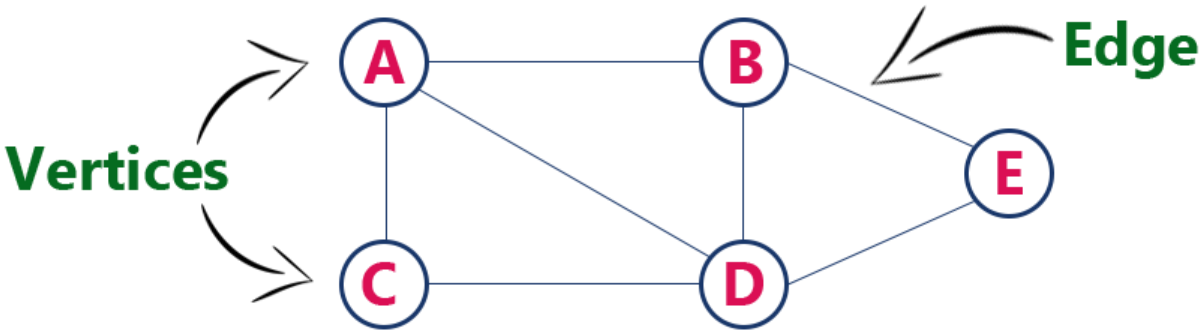



The Godfather
Marlon Brando
Al Pacino
Scarface
Michelle Pfeiffer
Viva Zapata!
Dick Tracy
Henry Silva

Networks or Graphs

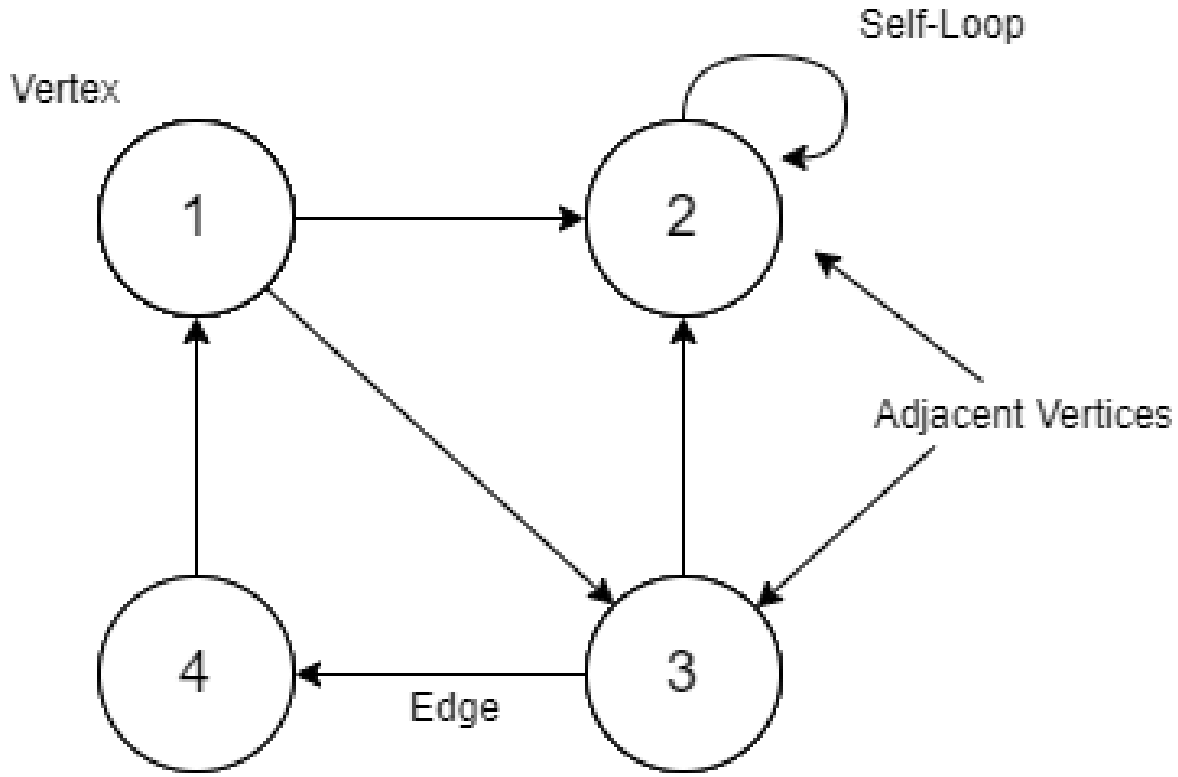
- Network often refers to real systems
  - www,
  - social network,
  - metabolic network
- Graph: mathematical representation of a network
  - web graph,
  - social graph (a Facebook term)

- A graph consists of a **finite** **set of vertices** (or nodes) and **set of edges** which connect a pair of vertices (nodes).
- $G = \{V, E\}$
  - V: set of vertices – $V = \{v_1, v_2, \ldots, v_n\}$
  - E: set of edges – $E = \{e_1, e_2, \ldots, e_m\}$
- Example:
  - V = {A, B, C, D, E}
  - E = {AB, AC, AD, BD, BE, CD, ED}



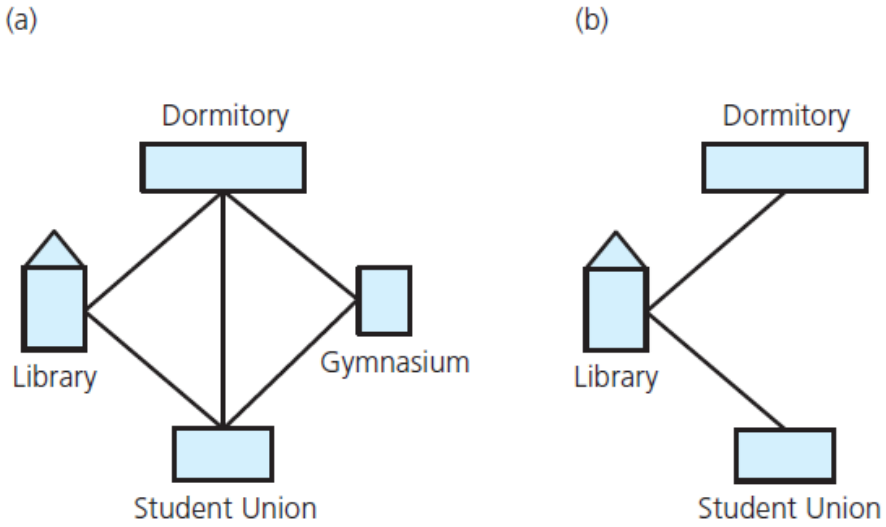
Terminologies

- Vertex: also called a node.
- Edge: connects two vertices.
- Loop (self-edge): An edge of the form (v, v).
- Adjacent: two vertices are adjacent if they are joined by an edge.



Terminologies

- A subgraph consists of a subset of a graph's vertices and a subset of its edges.
- $G' = \{V', E'\}$ is a subgraph of $G = \{V, E\}$ if $V' \subseteq V$, $E' \subseteq E$



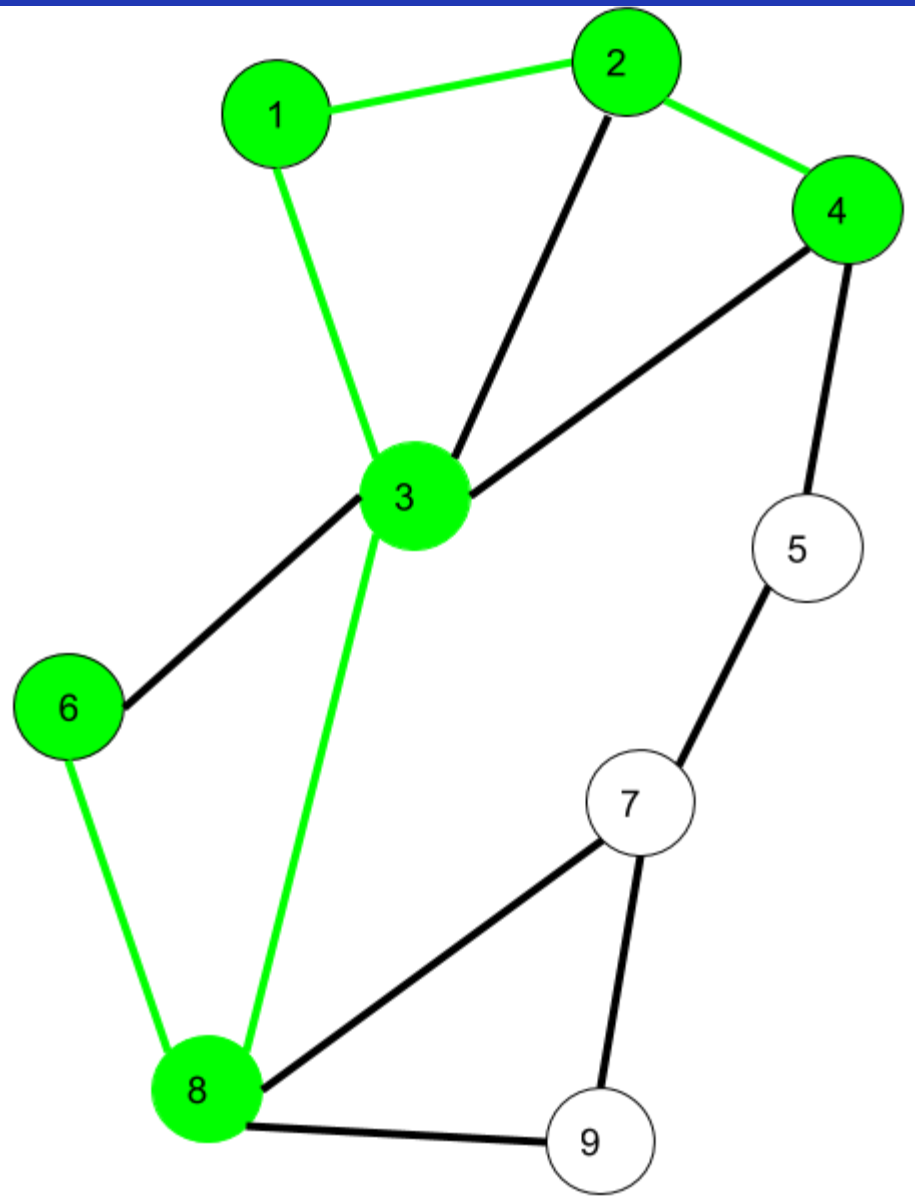
(a) A campus map as a graph;
(b) a subgraph of (a)

Terminologies

- **Walk :** A walk is a sequence of vertices and edges of a graph
  - Edge and Vertices both can be repeated
- **Trail**: is an open walk in which no edge is repeated
  - Vertex can be repeated
- **Circuit/Cycle**: Traversing a graph such that not an edge is repeated but vertex can be repeated
  - Vertex can be repeated and Edge can not be repeated
- **Path**: A sequence of edges that begins at one vertex and ends at another vertex → **no repeated vertex and edge**.
  - If all vertices of a path is distinct, the path is simple.

- **Path:**

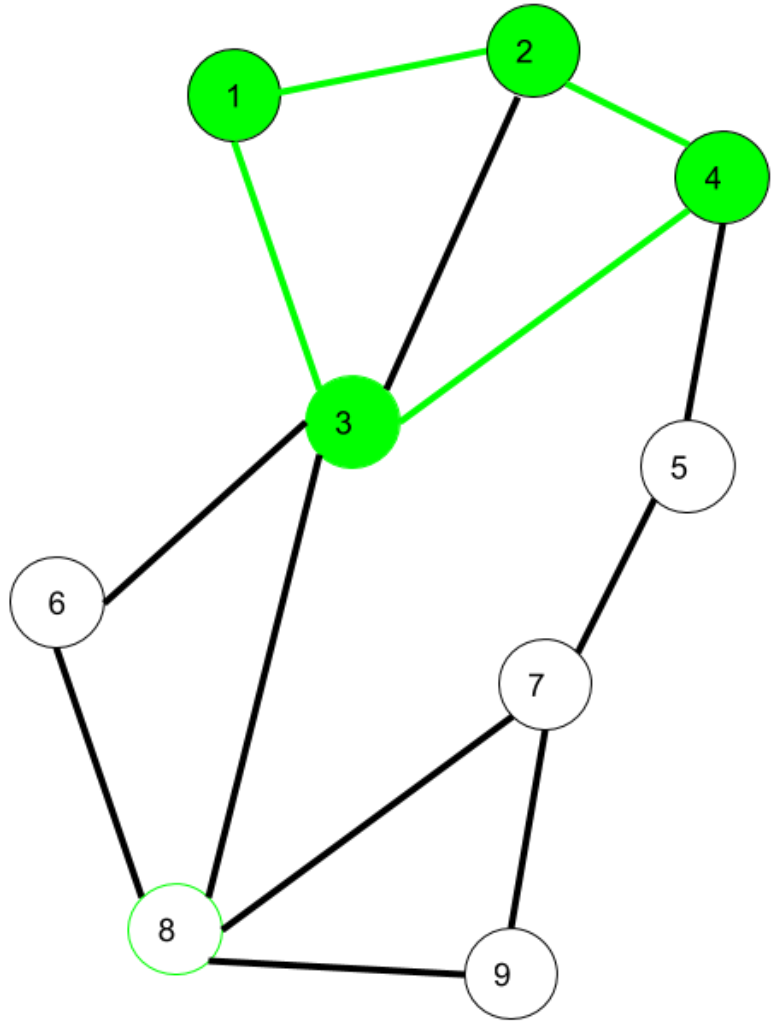
$6 \rightarrow 8 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4$



Terminologies

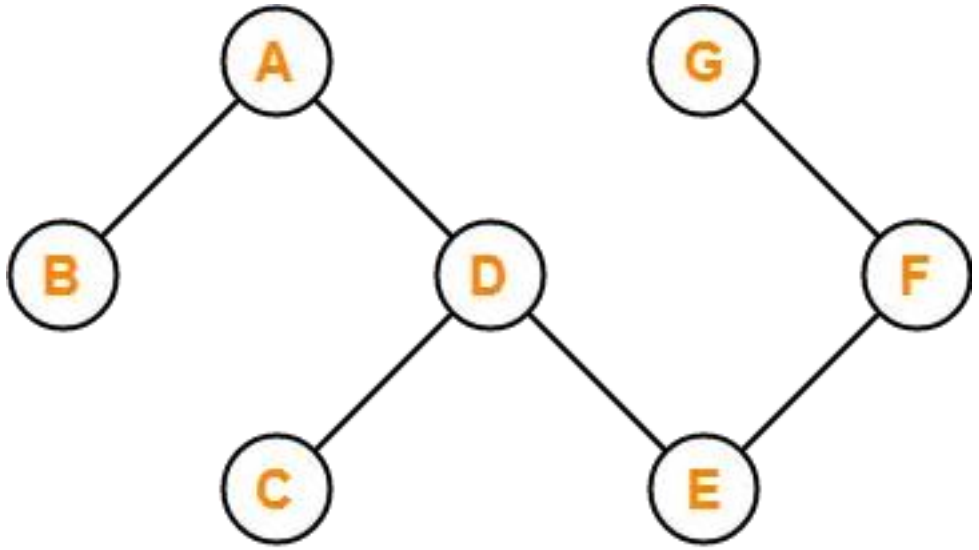
- **Cycle**: A path that starts and ends at the same vertex and does not traverse the same edge more than once.
  - *no repeated vertex and edge.*
  - *Same starting and ending vertex*

  For example, Cycle: 1 → 2 → 4 → 3 → 1

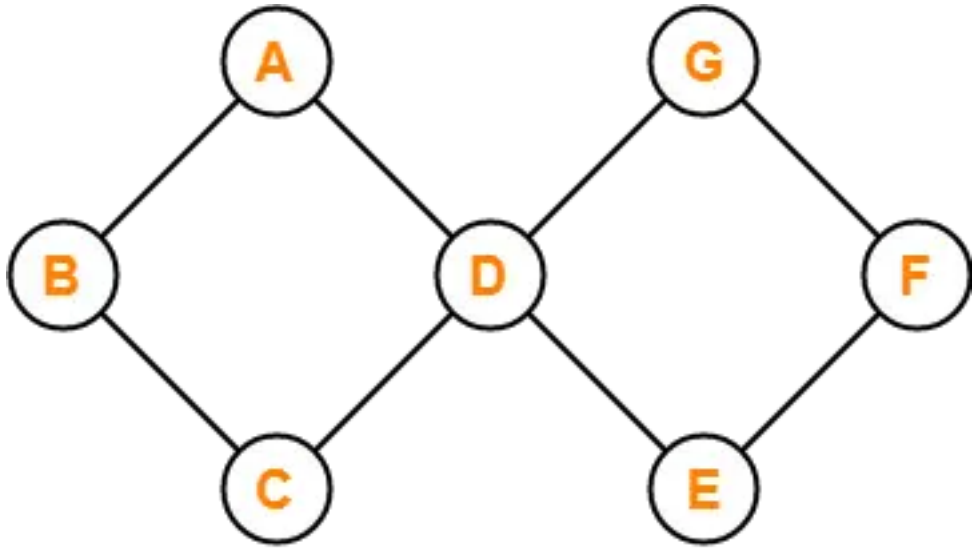


- **Acyclic graph**: A graph with no cycle.

- **Acyclic graph**: A graph with no cycle.



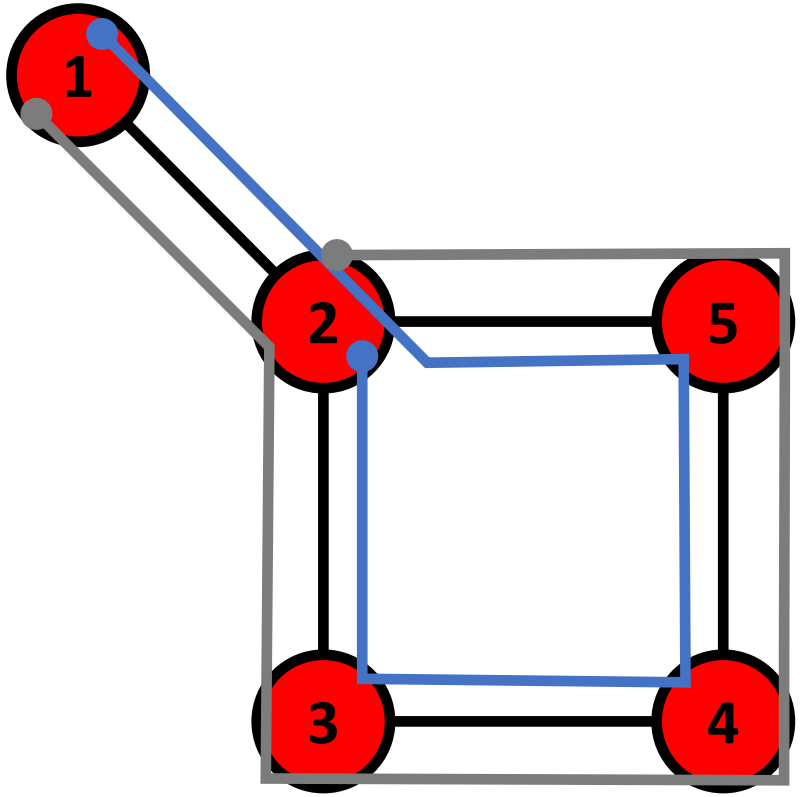
Example of Acyclic Graph



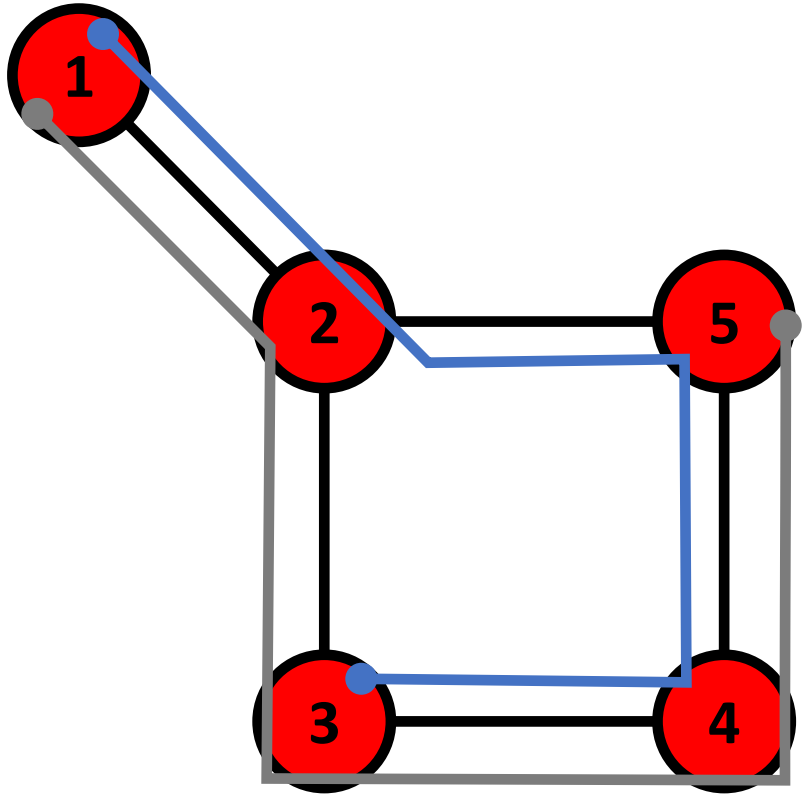
Example of Cyclic Graph

Terminologies

- **Eulerian Path:** A path that traverses **each edge** exactly once.
- **Hamiltonian Path:** A path that visits **each vertex** exactly once.



Eulerian Path

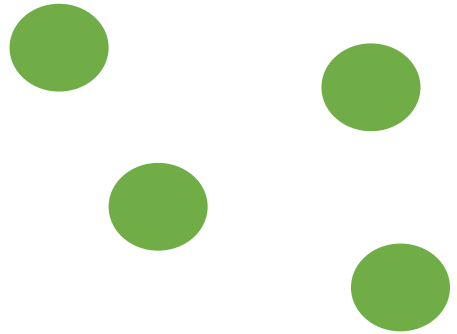


Hamiltonian Path

Terminologies

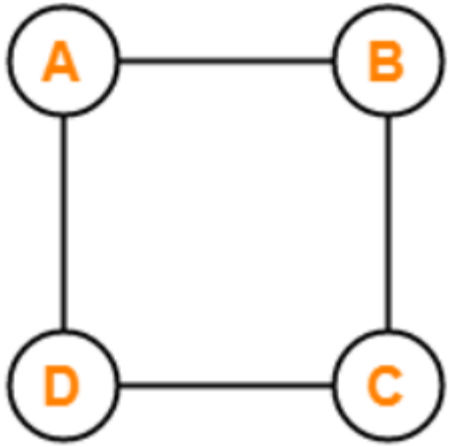
- **Null graph:** A graph having no edges
- **Trivial graph:** A graph with only one vertex

trivial graph



null graph

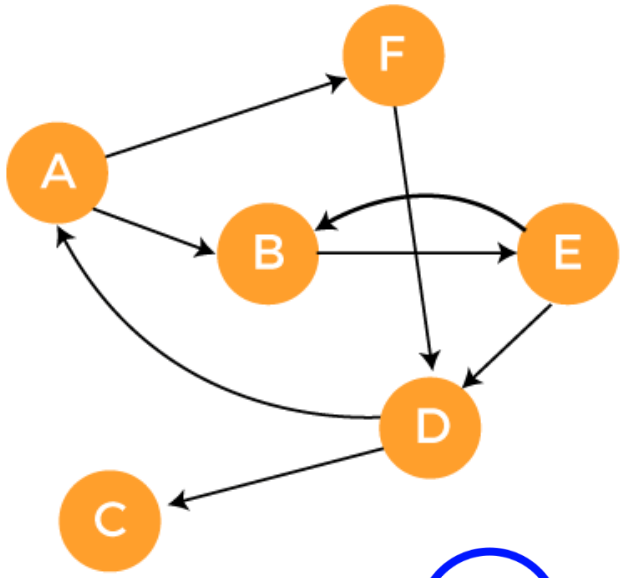
- **Undirected graph:** the graph in which edges do not indicate a direction



undirected graph

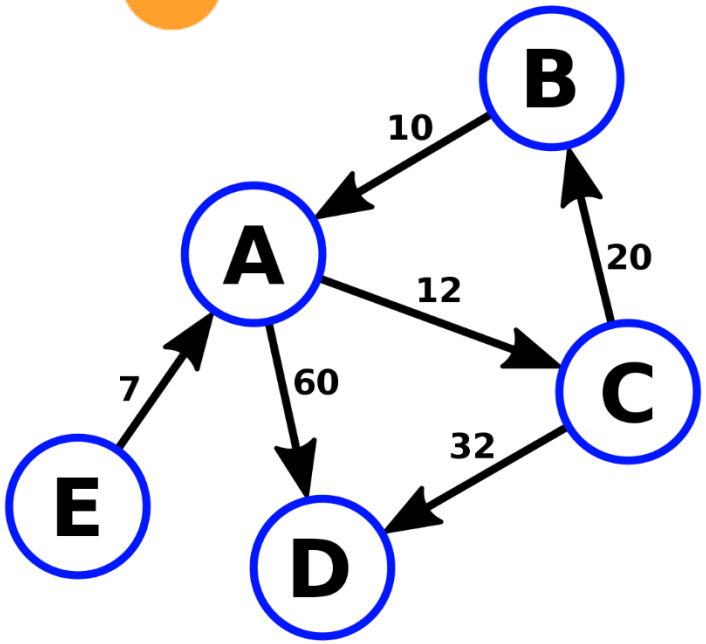
Terminologies

- **Directed graph, or digraph:** a graph in which each edge has a direction



directed graph

- **Weighted graph:** a graph with numbers (weights, costs) assigned to its edges



weighted graph

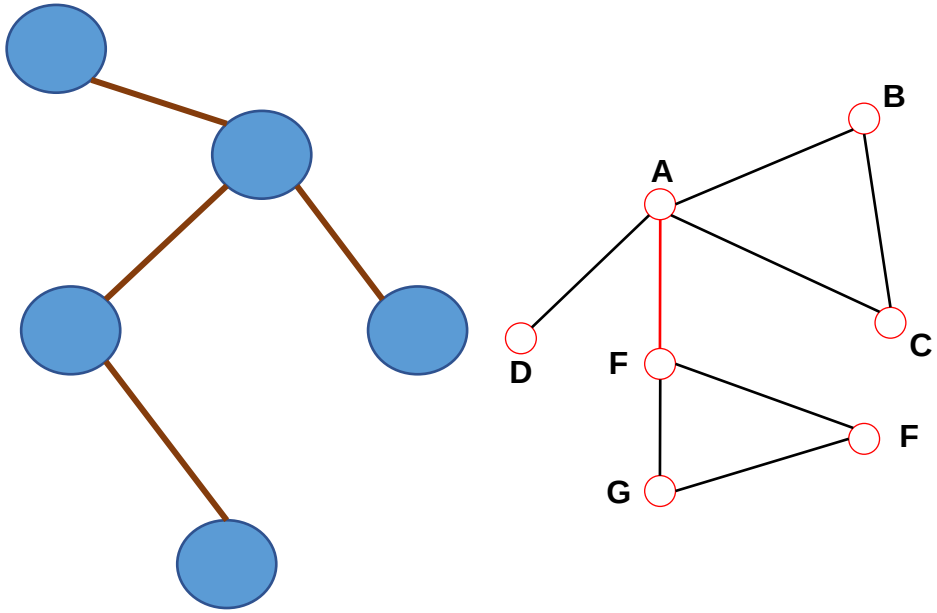
Terminologies

- **Connected graph:** A graph in which each pair of distinct vertices has a path between them

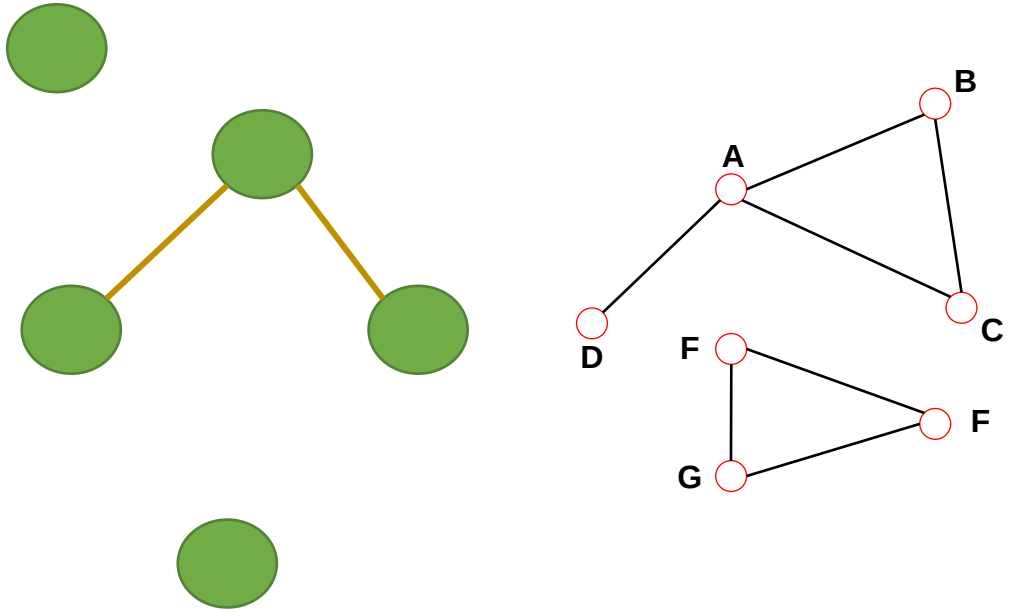
- **Disconnected graph:** A graph does not contain at least two connected vertices.
  - there does not exist any path between at least one pair of vertices

- Graph cannot have duplicate edges between vertices.
  - **Multigraph**: does allow multiple edges

Terminologies



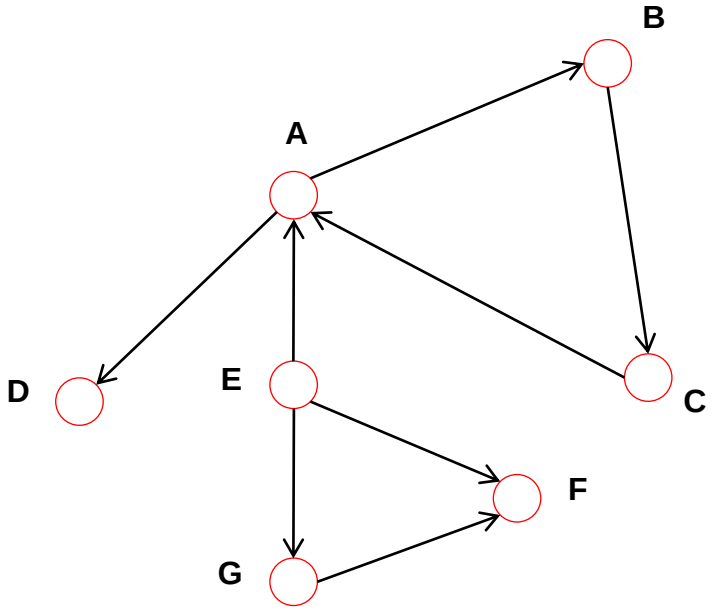
connected graph

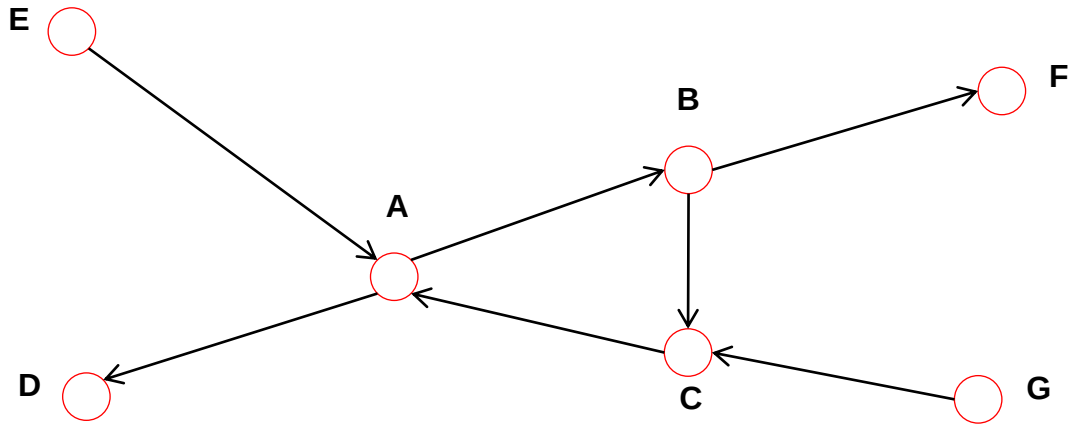


disconnected graph

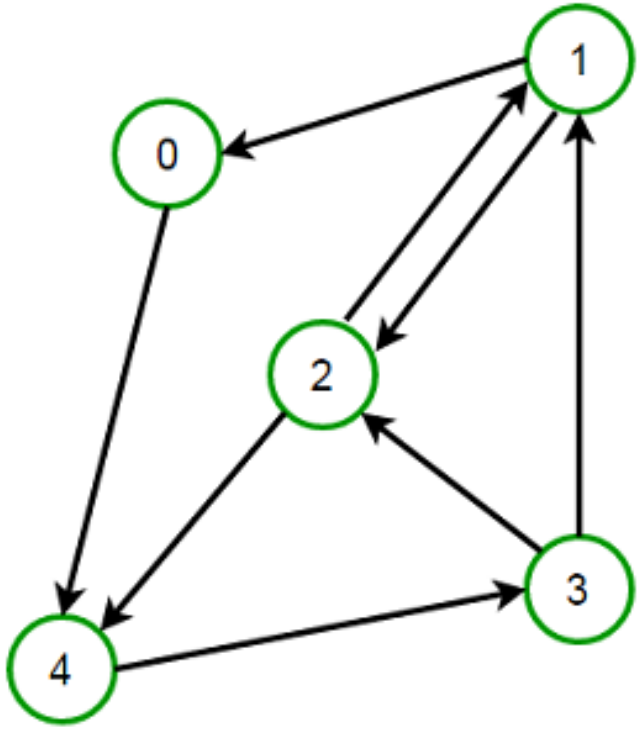
Terminologies

- **Strongly connected directed graph:** if there is a path in each direction between each pair of vertices of the graph.
- **Weakly connected directed graph:** it is connected if we disregard the edge directions.



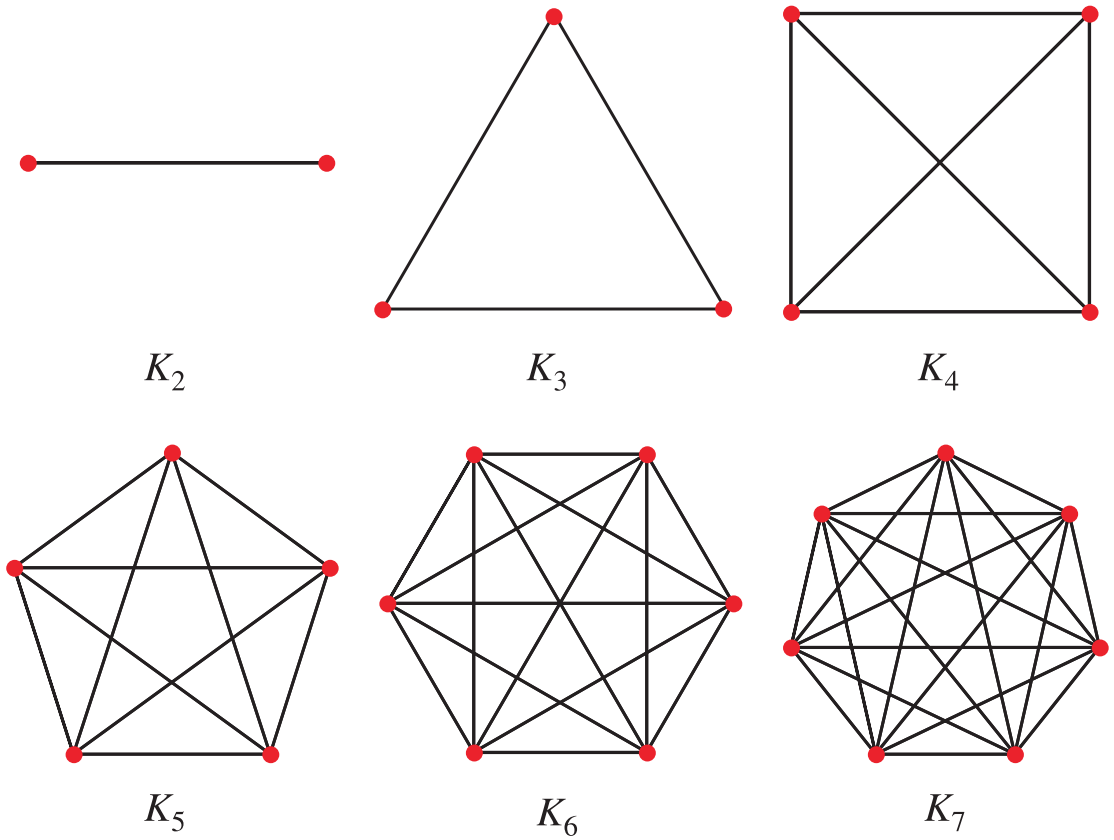


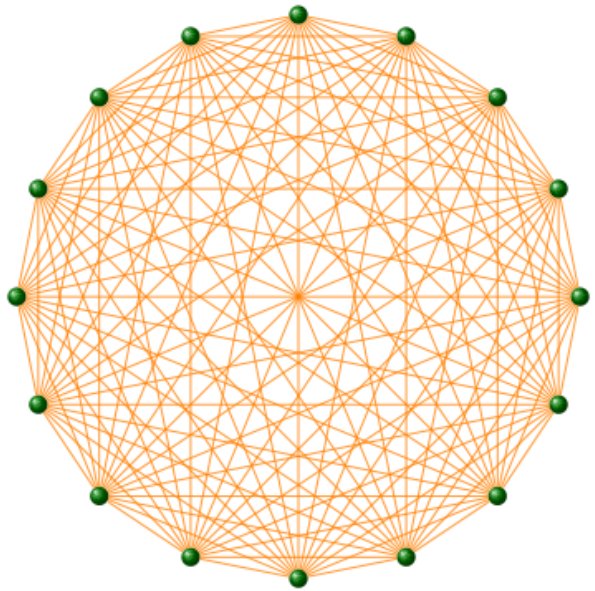
It is connected but not strongly connected (e.g., there is no way to get from F to G by following the edge directions)



Terminologies

- **Complete graph:** A graph in which each pairs of distinct vertices has an edge between them
- The maximum number of edges a graph N vertices can have?

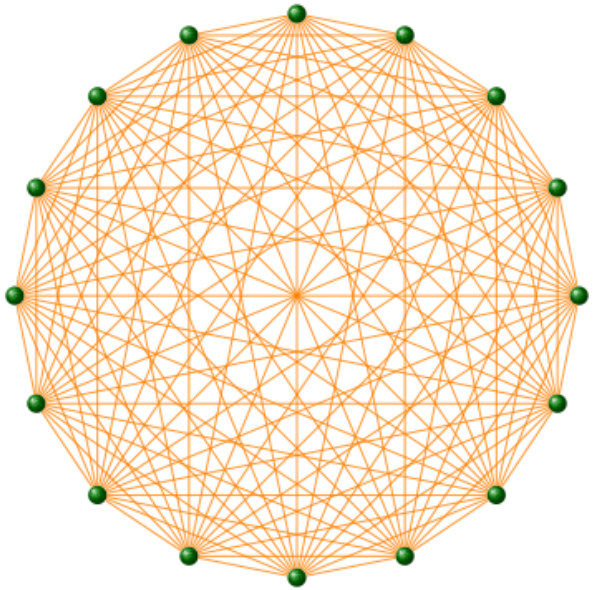




Terminologies

- **Complete graph:** A graph in which each pairs of distinct vertices has an edge between them
- The maximum number of edges a graph N vertices can have?

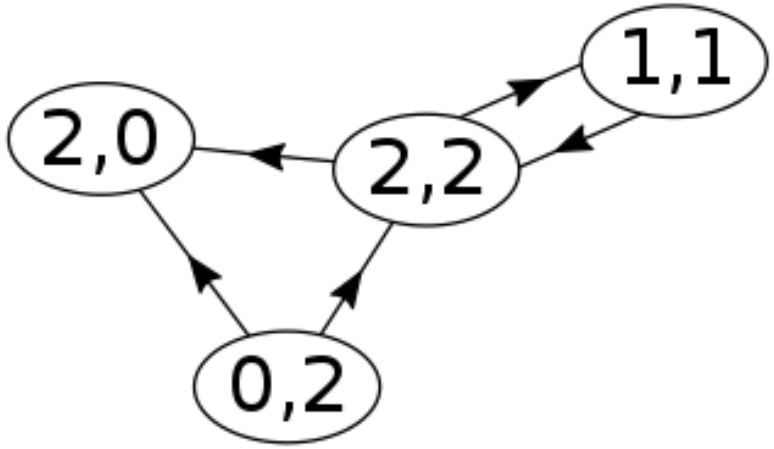
$$C(N, 2) = \frac{N * (N - 1)}{2}$$



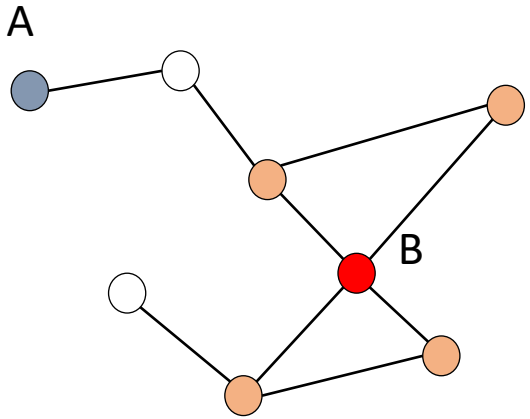
Terminologies

- **Degree** of a vertex v (denoted deg(v)): the number of edges connected to v.
- In directed graphs, we can define an in-degree and out-degree of vertex v.
  - **In-degree** of v (denoted $\deg^-(v)$): number of head ends adjacent to v.
  - **Out-degree** of v (denoted $\deg^+(v)$): number of tail ends adjacent to v.
  - $\deg(v) = \deg^-(v) + \deg^+(v)$
- Note:
  - arc(x, y): direction from x to y.
    - x is called *tail* and y is called *head* of the arc.

Terminologies



A directed graph with vertices labeled (indegree, outdegree)



deg(A) = 1; deg(B) = 4

Terminologies

- Let $G = \{V, E\}$
- If G is an undirected graph

$$\sum_{v \in V} \deg(v) = 2|E|$$

- If G is a directed graph

$$\sum_{v \in V} deg^{-}(v) = \sum_{v \in V} deg^{+}(v) = |E|$$

Graph Representation

Graph Representation

- **Adjacency Matrix**

- **Adjacency List**

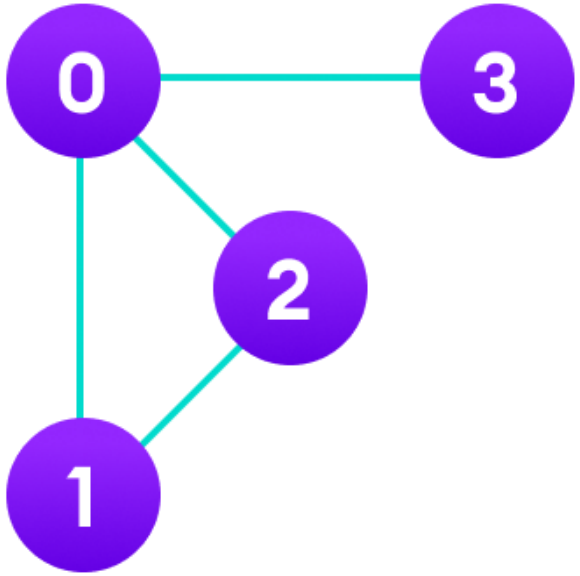
Adjacency Matrix

- An adjacency matrix is a 2D array of |V| x |V| vertices. Each row and column represent a vertex
- If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j

$$A[i][j] = \begin{cases} 1 \; if \; there \; is \; an \; edge(i,j) \\ 0 \; if \; there \; is \; no \; edge \; (i,j) \end{cases}$$

$$A[i][j] = \begin{cases} w \; with \; w \; is \; the \; weight \; of \; edge(i,j) \\ \infty \qquad\qquad if \; there \; is \; no \; edge \; (i,j) \end{cases}$$

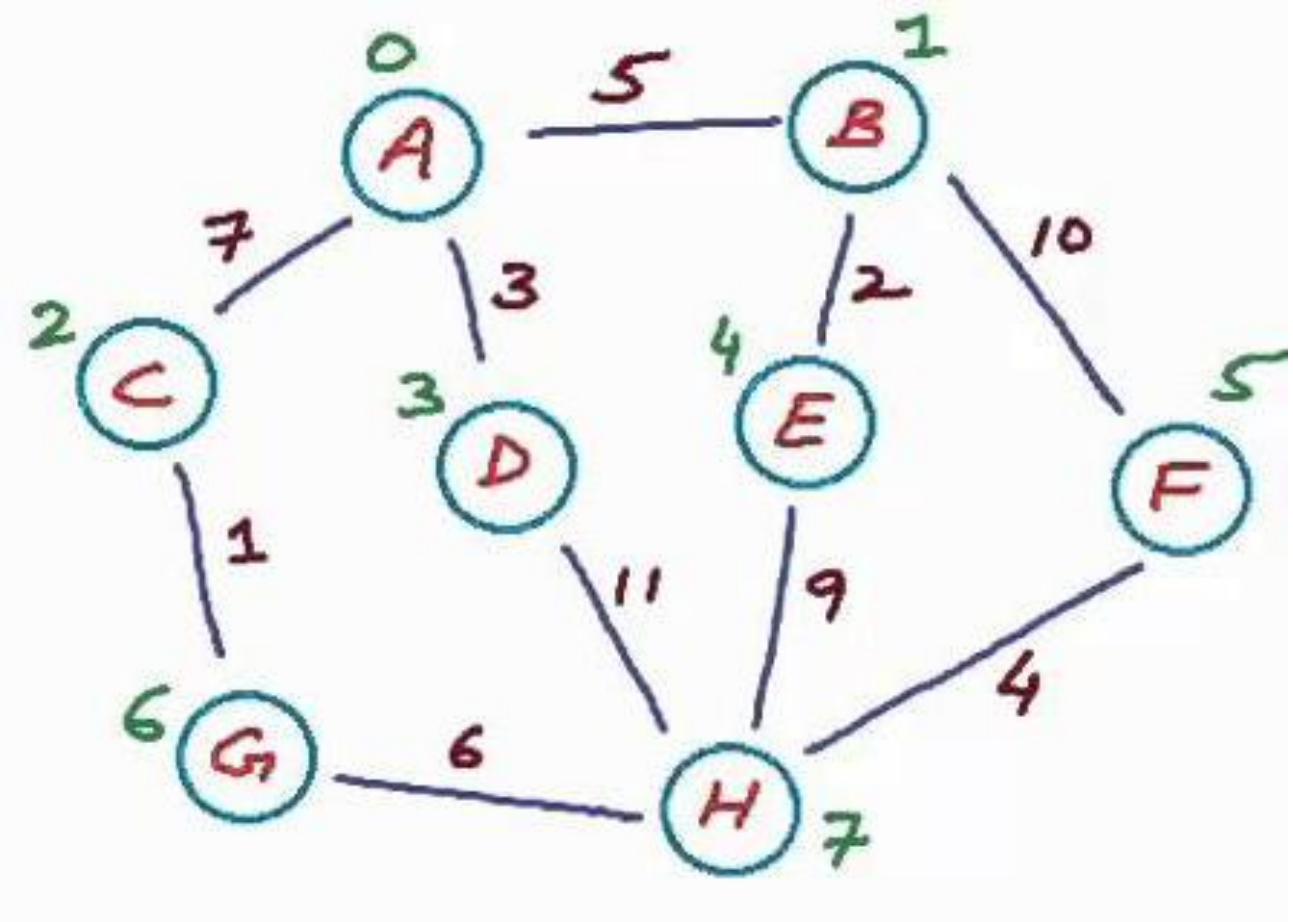
Adjacency Matrix

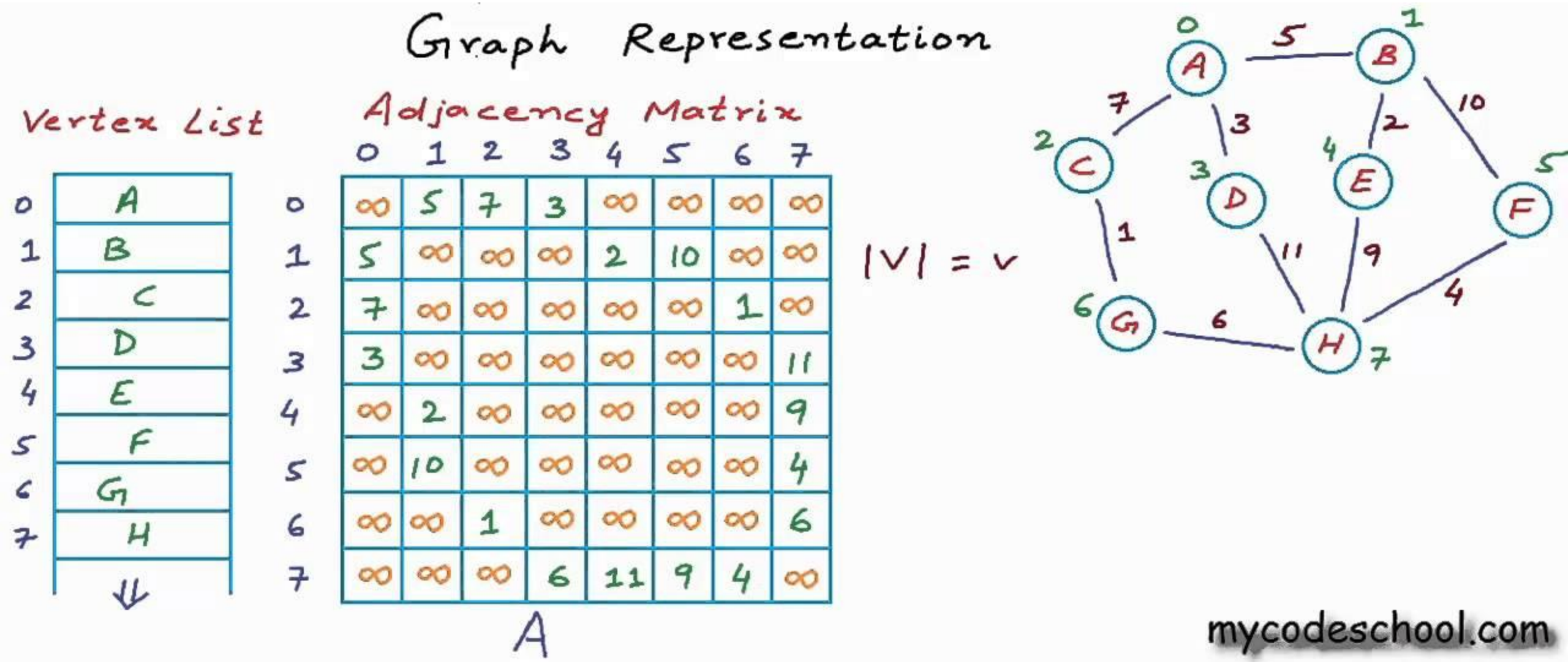


	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Adjacency Matrix

- Determine the Adjacency Matrix of the following graph



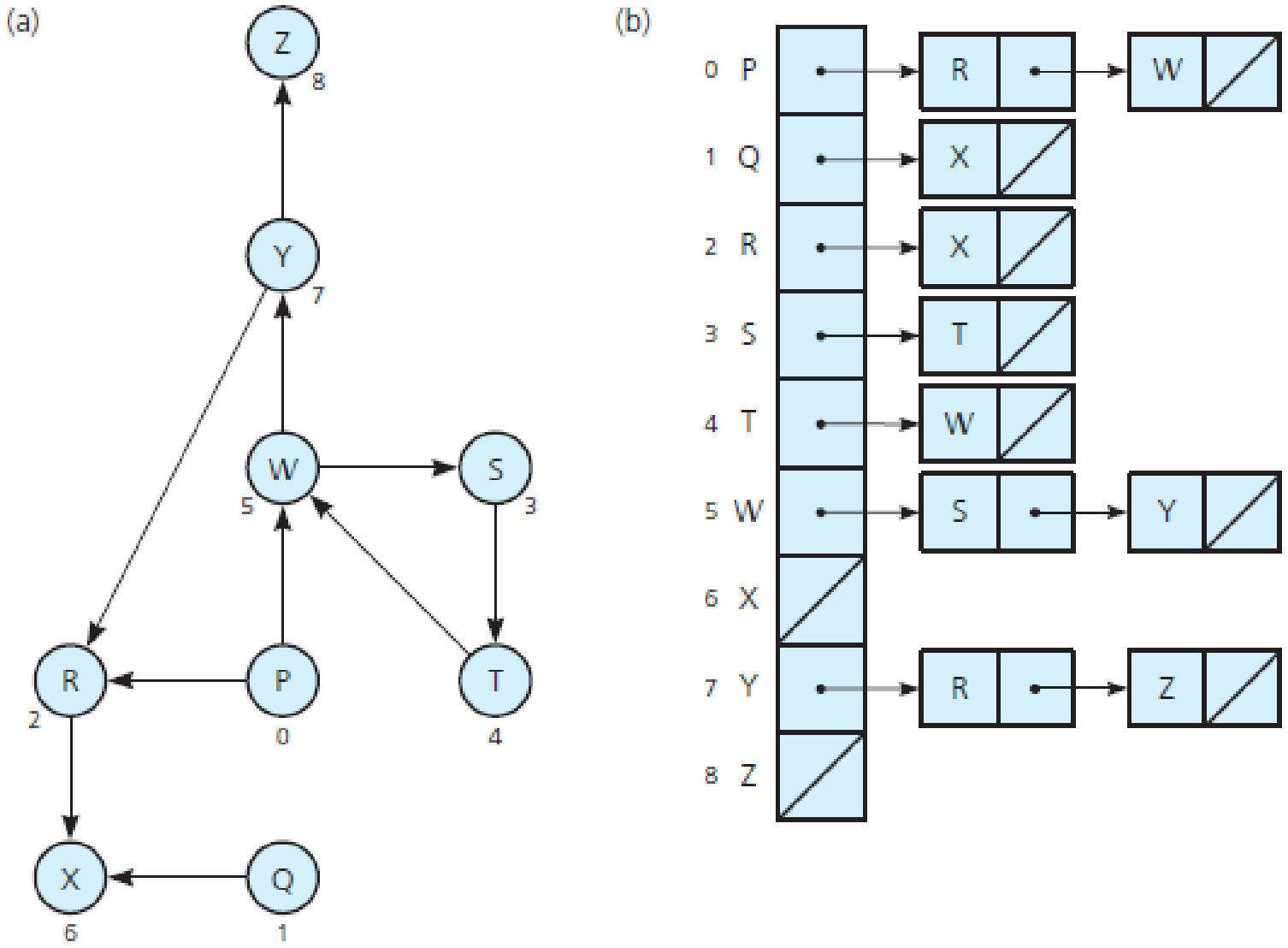

Graph Representation
Vertex List
Adjacency Matrix
0 1 2 3 4 5 6 7
0 A
1 B
2 C
3 D
4 E
5 F
6 G
7 H
0 ∞ 5 7 3 ∞ ∞ ∞ ∞
1 5 ∞ ∞ ∞ 2 10 ∞ ∞
2 7 ∞ ∞ ∞ ∞ ∞ 1 ∞
3 3 ∞ ∞ ∞ ∞ ∞ ∞ 11
4 ∞ 2 ∞ ∞ ∞ ∞ ∞ 9
5 ∞ 10 ∞ ∞ ∞ ∞ ∞ 4
6 ∞ ∞ 1 ∞ ∞ ∞ ∞ 6
7 ∞ ∞ ∞ 6 11 9 4 ∞
A
|V| = v
A B C D E F G H
5 7 3 2 10 1 11 9 4 6 6 7
mycodeschool.com

Adjacency List

- A graph with n vertices has n linked chains.

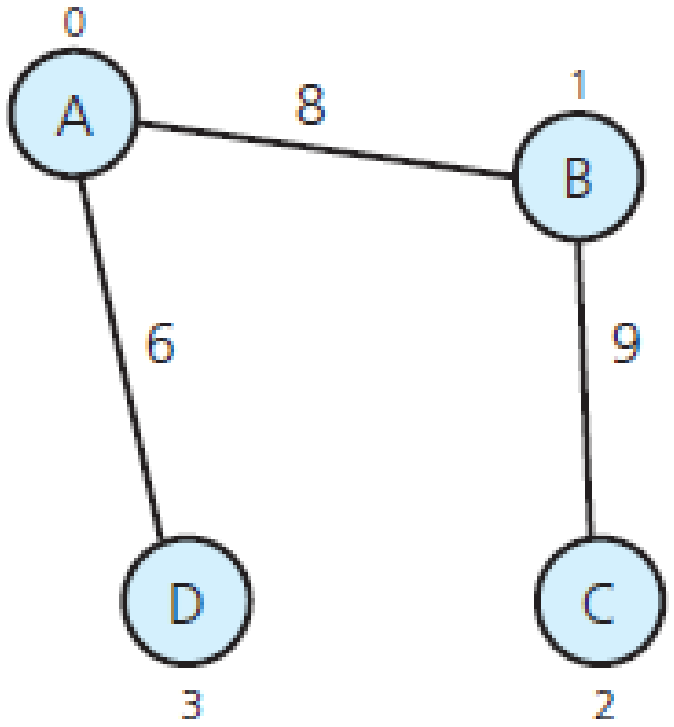
- The i^{th} linked chain has a node for vertex j if and only if having edge (i,j).

Adjacency List

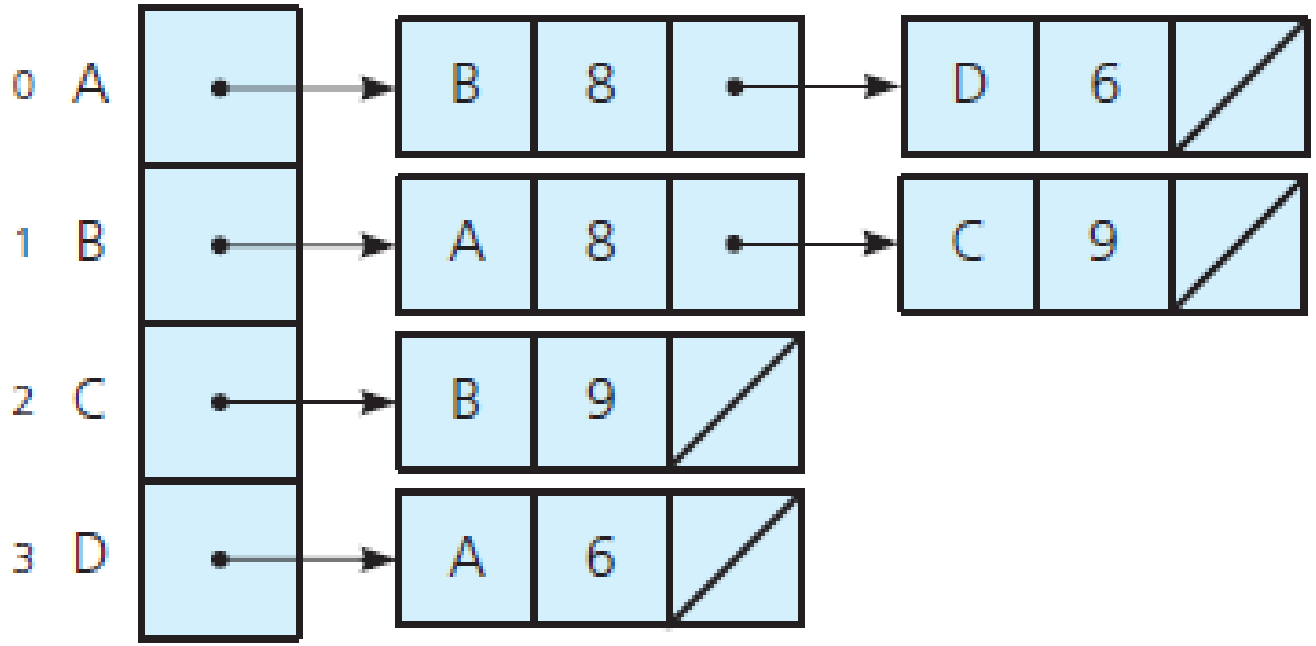


Adjacency List

(a)

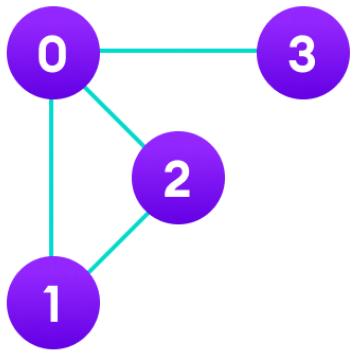


(b)

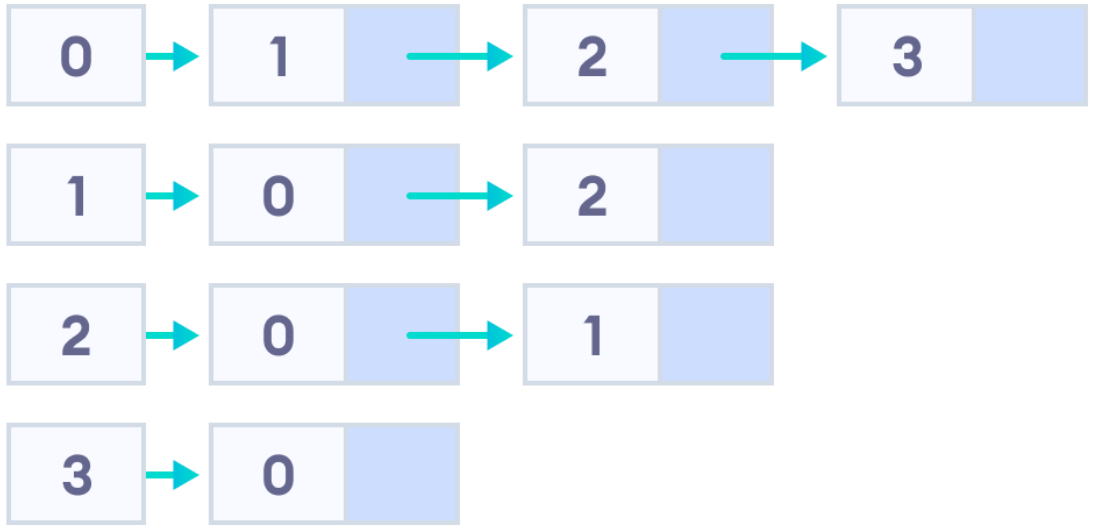


Relative Advantages

- Faster to test if (x, y) in graph?
- Faster to find the degree of a vertex?
- Less memory on small graph?
- Less memory on big graph?
- Edge insertion or deletion?
- Faster to traverse the graph?
- Better for most problems?



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0



Adjacency Matrix

- Uses $O(n^2)$ memory
- It is fast to lookup and check for presence or absence of a specific edge between any two nodes O(1)
- It is slow to iterate over all edges
- It is slow to add/delete a node; a complex operation $O(n^2)$
- It is fast to add a new edge O(1)

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Readmore: Comparision/

Adjacency List

- Memory usage depends more on the number of edges (and less on the number of nodes), which might save a lot of memory if the adjacency matrix is sparse
- Finding the presence or absence of specific edge between any two nodes is slightly slower than with the matrix O(k); where k is the number of neighbors nodes
- It is fast to iterate over all edges because you can access any node neighbors directly
- It is fast to add/delete a node; easier than the matrix representation
- It is fast to add a new edge O(1)

Graph Traversal

- Visits (all) the vertices that it can reach.

- Connected component is subset of vertices visited during traversal that begins at given vertex.

Depth-First Search

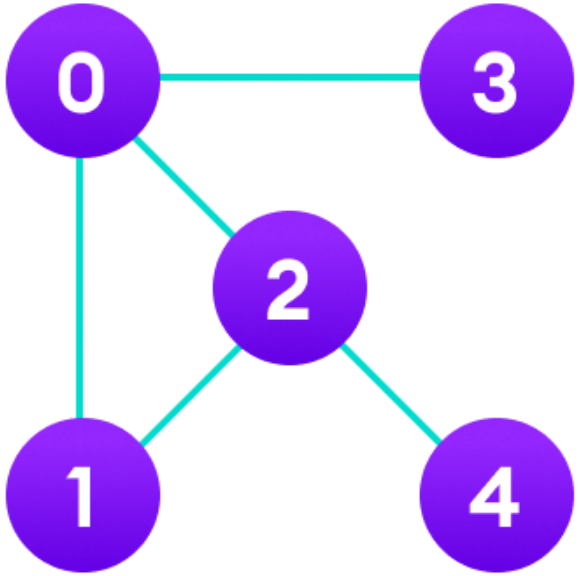
- Goes as far as possible from a vertex before backing up

```
DFS(v: vertex)
{
   Mark v as visited
   for (each unvisited vertex u adjacent to v)
      DFS(u)
}
```

Depth-First Search

```
DFS(v: vertex)
   s = new empty stack
   s.push(v)
   Mark v as visited
   while (s is not empty)  {
      if (no unvisited vertices are adjacent to
                                 the vertex on the top of the stack)
         s.pop()
      else {
         s.push(u)
         Marked u as visited
      }
   }
```

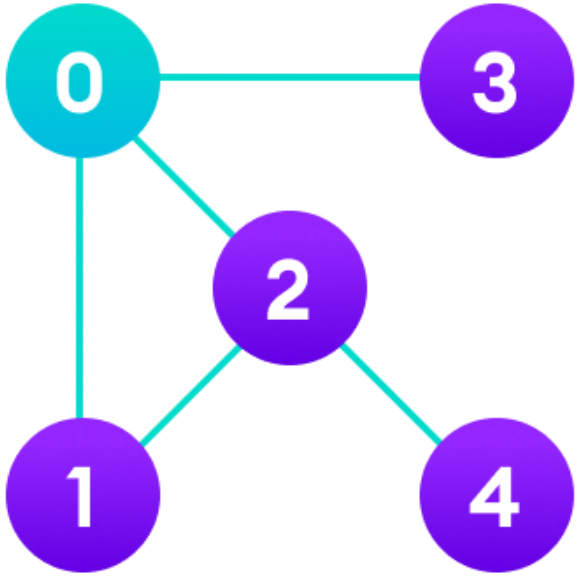
Depth-First Search



Visited

Stack

Depth-First Search



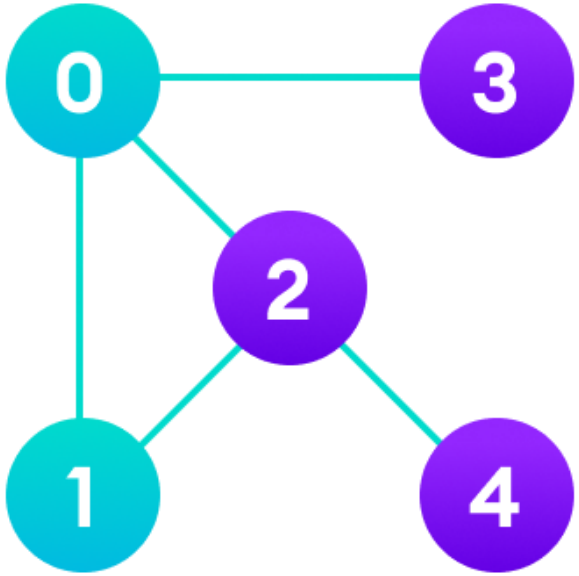
0				

Visited

1	2	3		

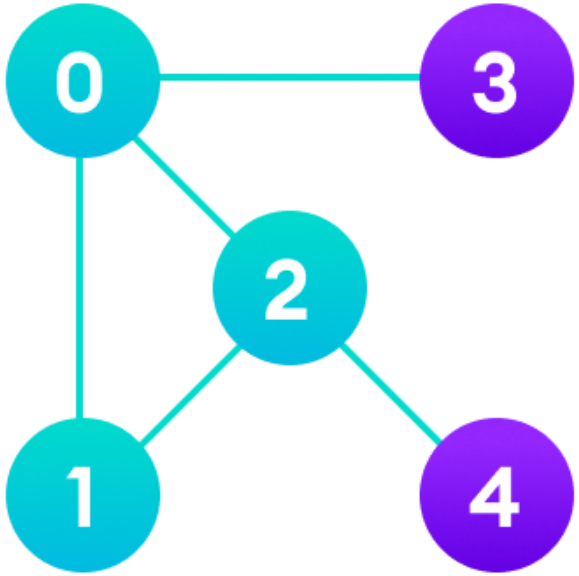
Stack

Depth-First Search



0	1				Visited
2	3				Stack

Depth-First Search

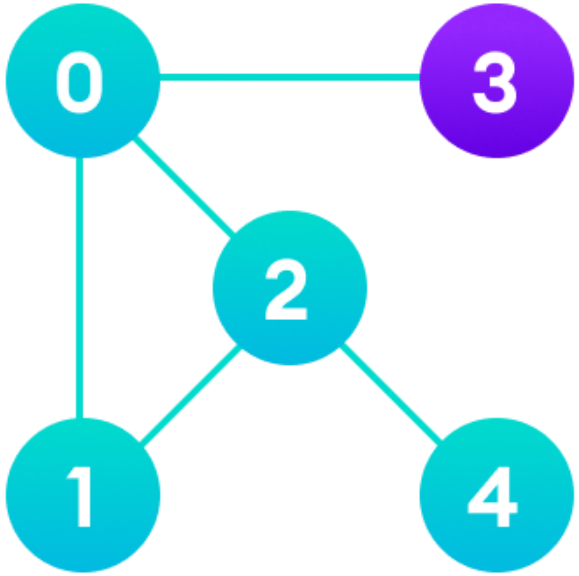


0	1	2		

Visited

4	3			

Stack



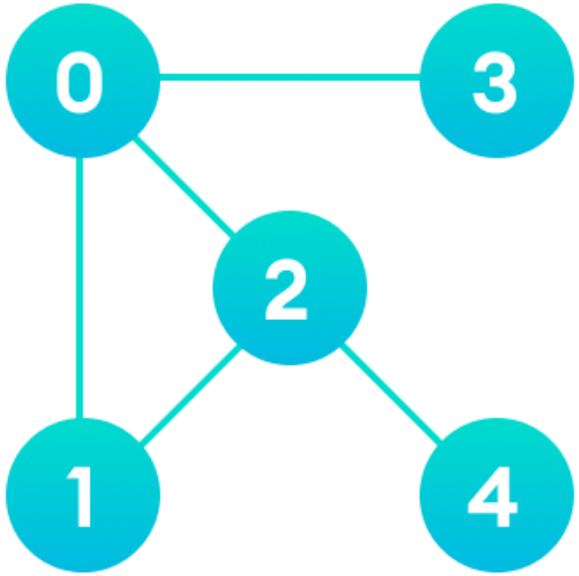
0	1	2	4	

Visited

3				

Stack

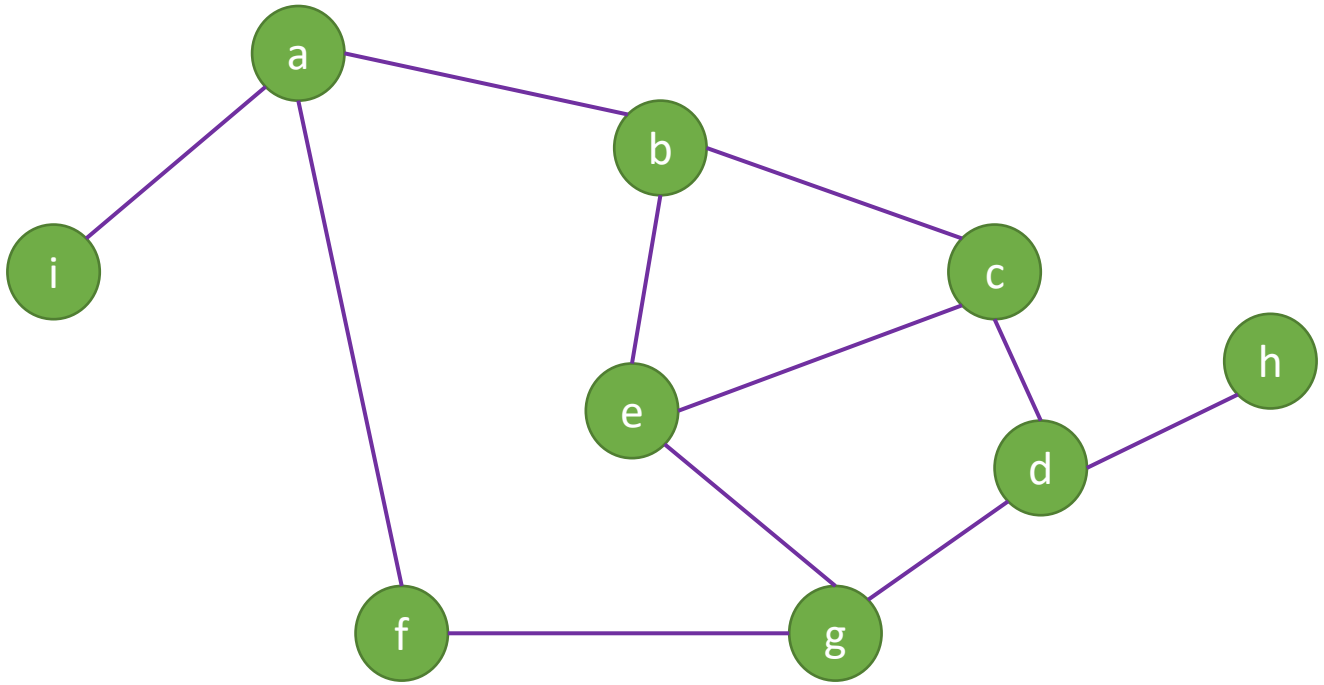
Depth-First Search



0	1	2	4	3	Visited

					Stack

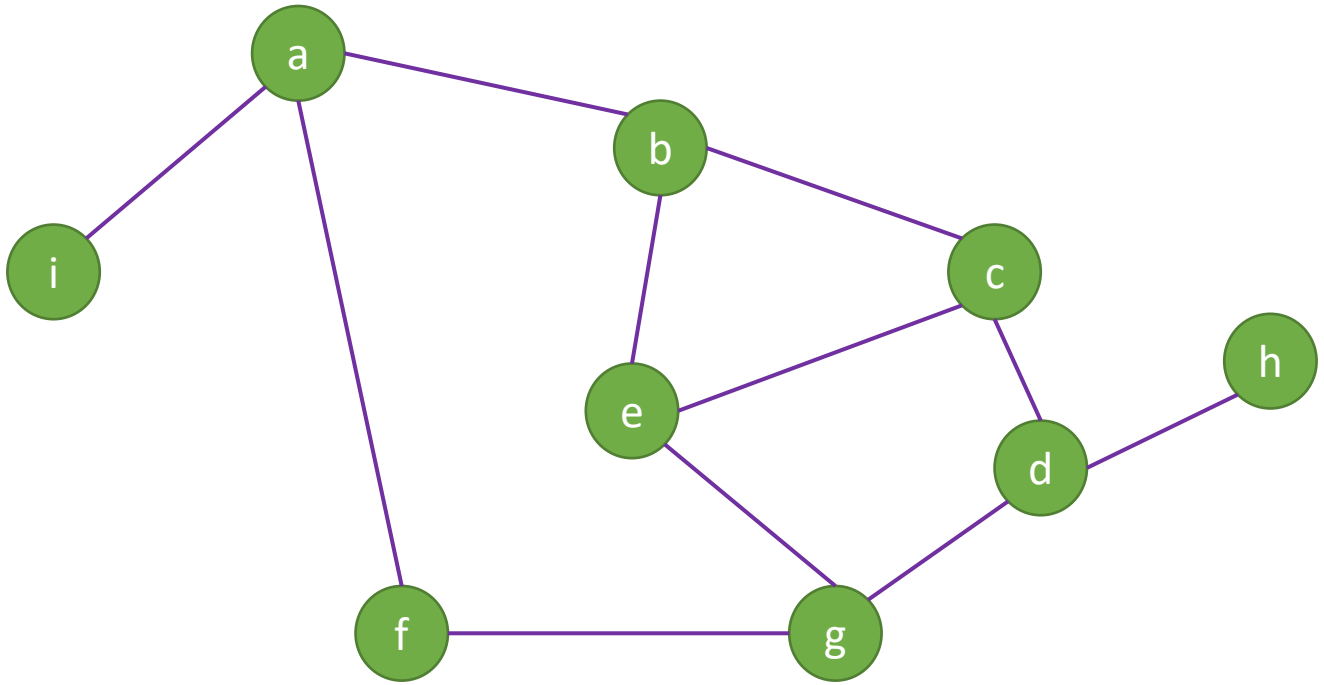
Depth-First Search



DFS starts at ***a***:

DFS starts at ***e***:

Depth-First Search



DFS starts at ***a***: a, b, c, d, g, e, f, h, i

DFS starts at ***e***: e, b, a, f, g, d, c, h, i

Breadth-First Search

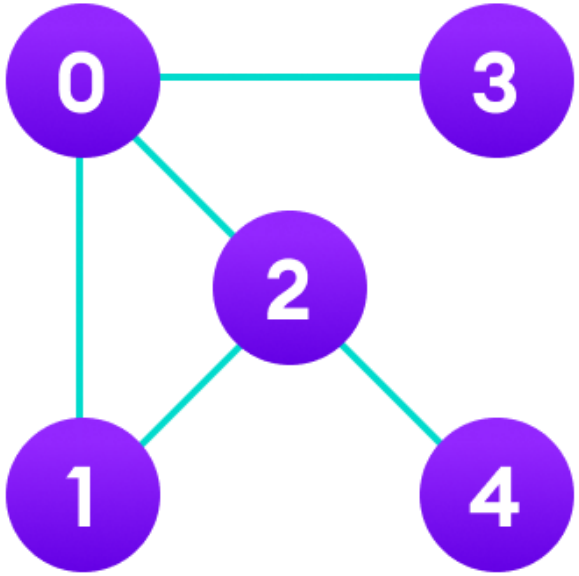
- Visits all vertices adjacent to vertex before going forward.

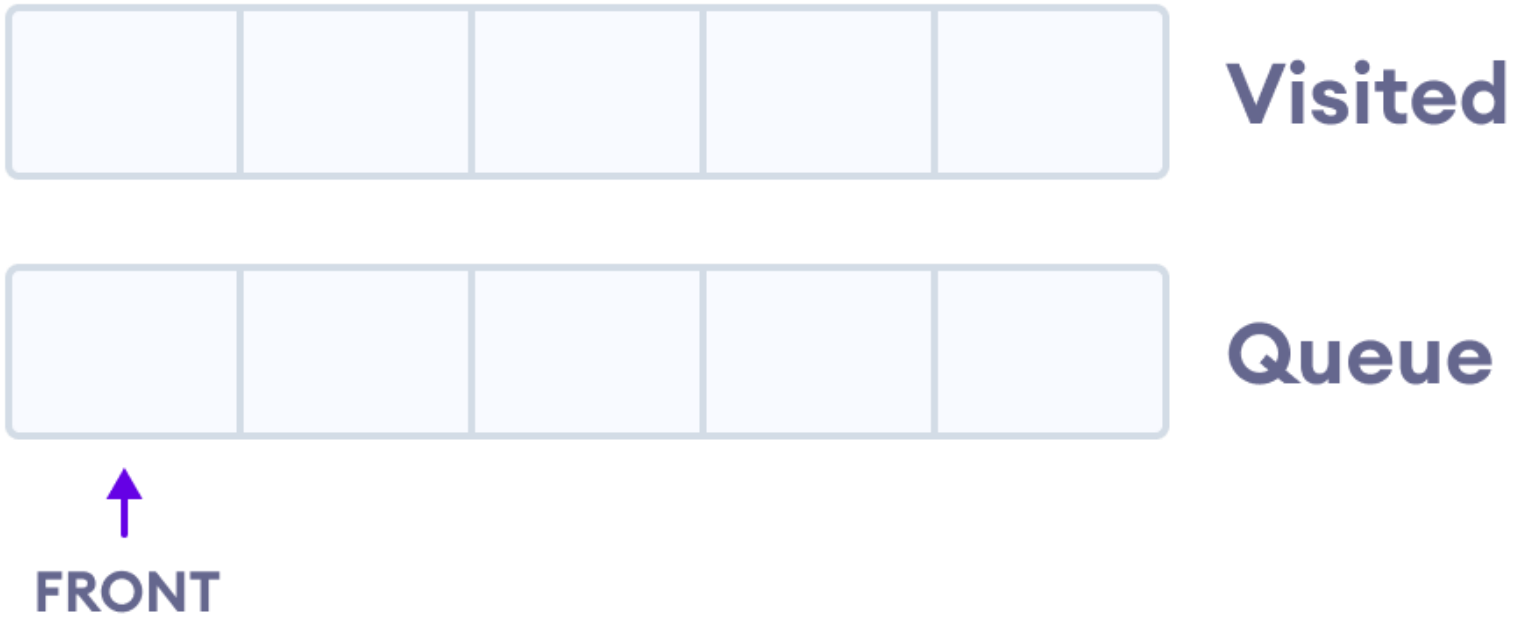
- Breadth-first search uses a **queue**.

Breadth-First Search

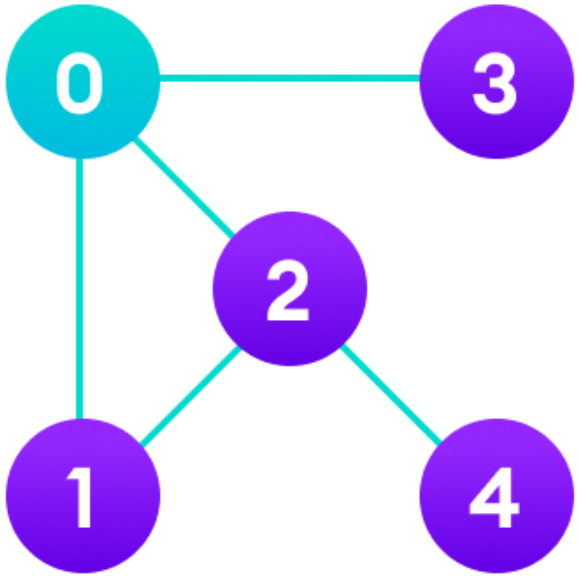
```
BFS(v: Vertex)
  q = a new empty queue
  q.enqueue(v)
  Mark v as visited
  while (q is not empty){
    w = q.dequeue()
    for (each unvisited vertex u adjacent to w){
      Mark u as visited
      q.enqueue(u)
    }
  }
```

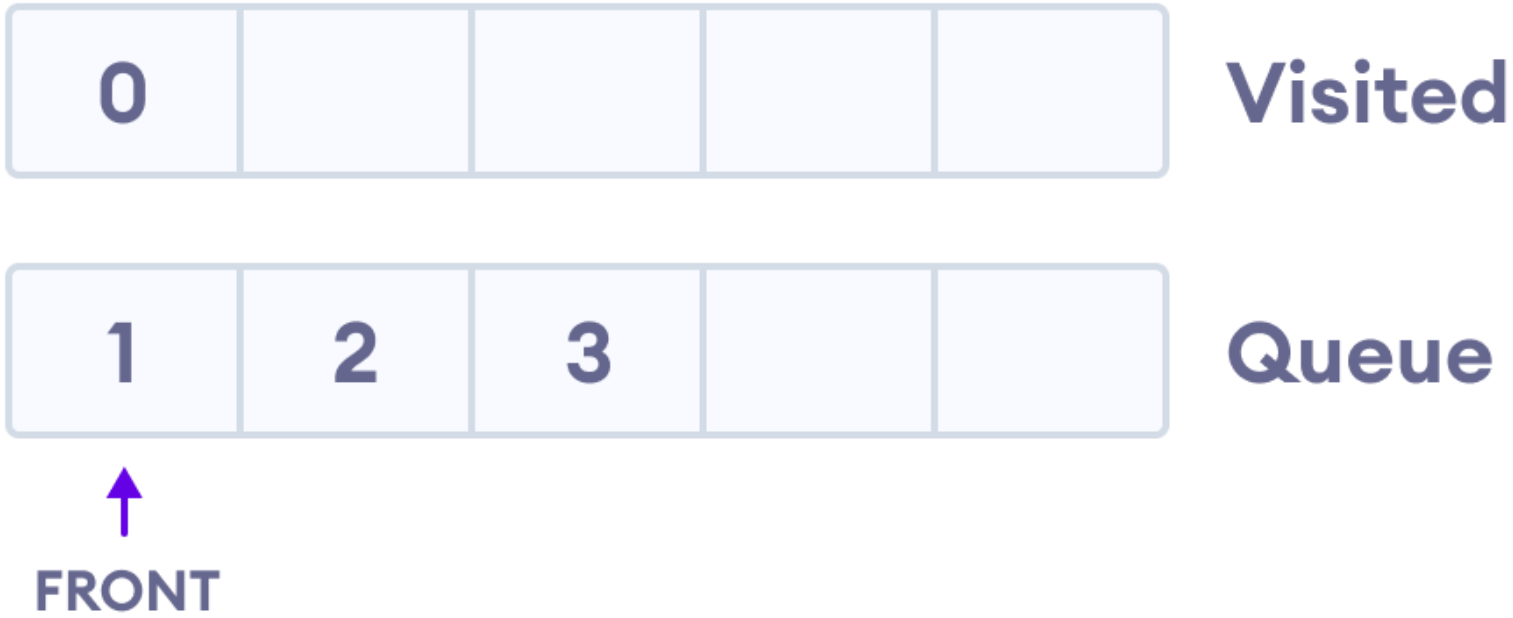
Breadth-First Search



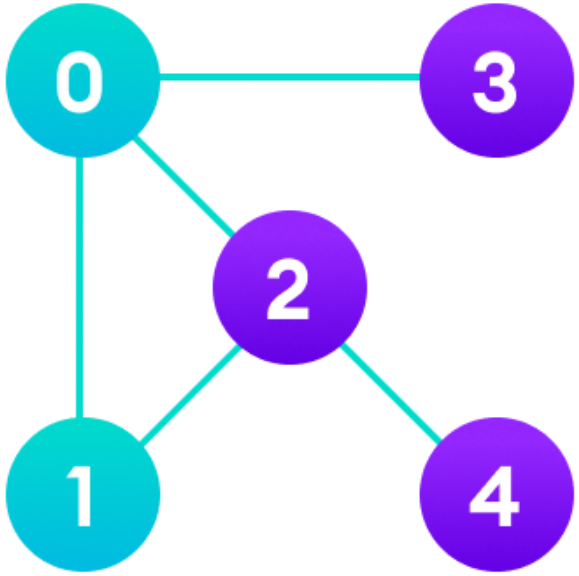


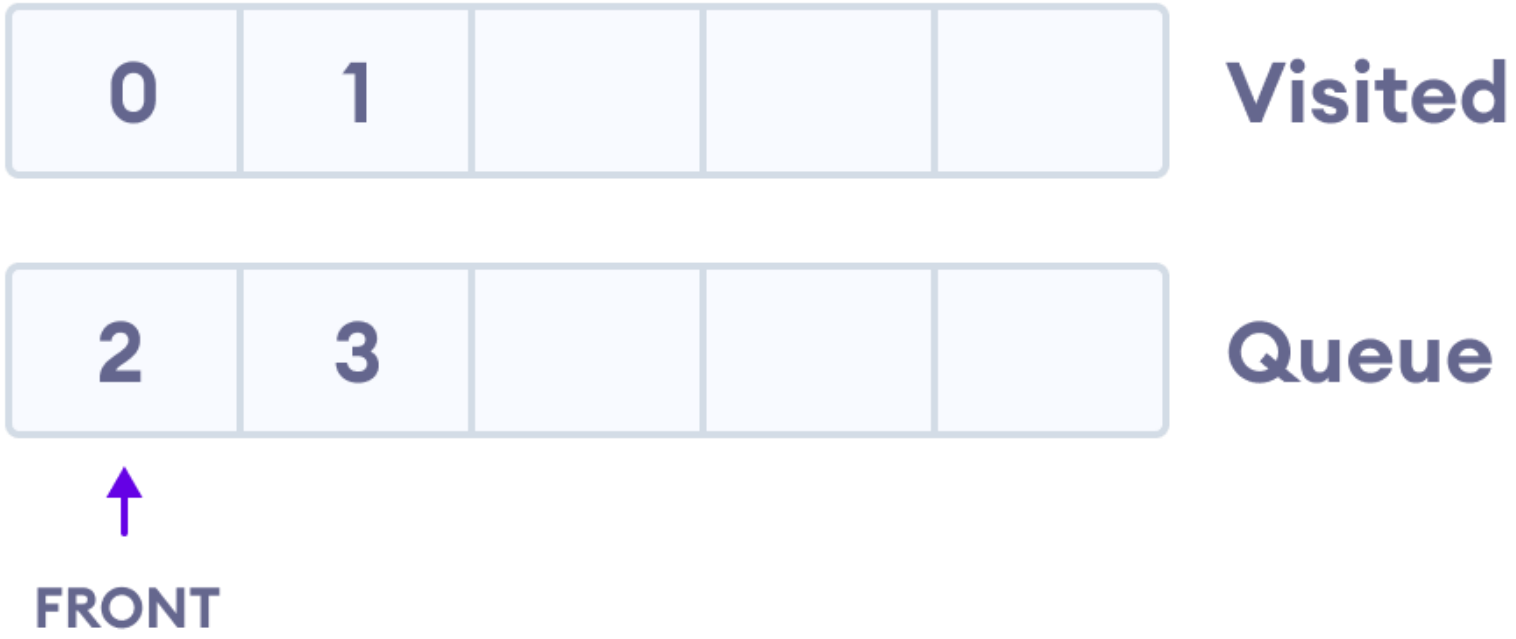
Breadth-First Search



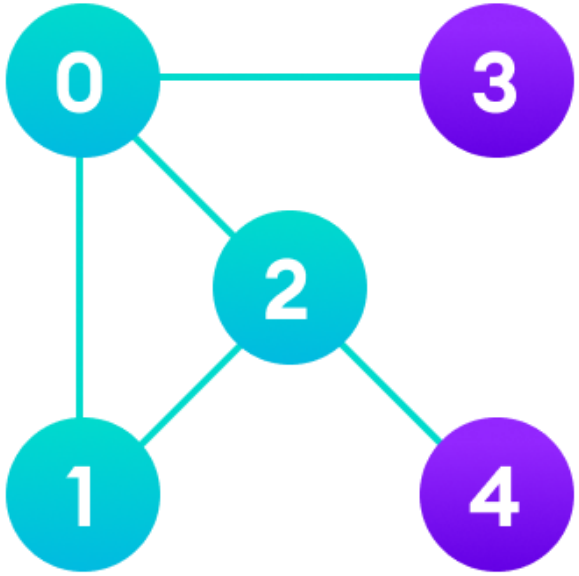


Breadth-First Search





Breadth-First Search



0	1	2		

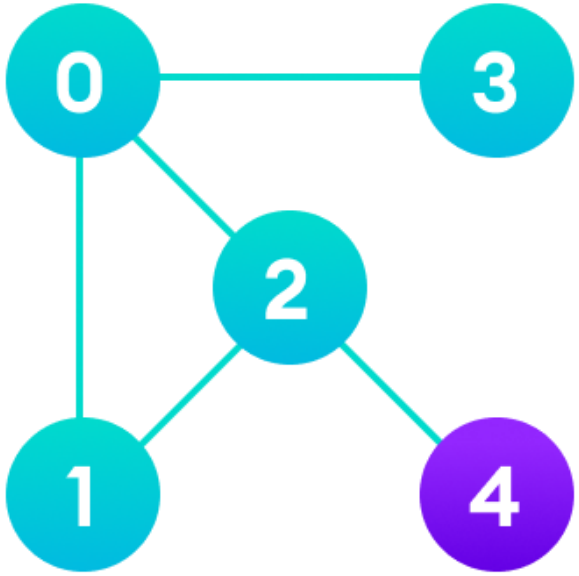
Visited

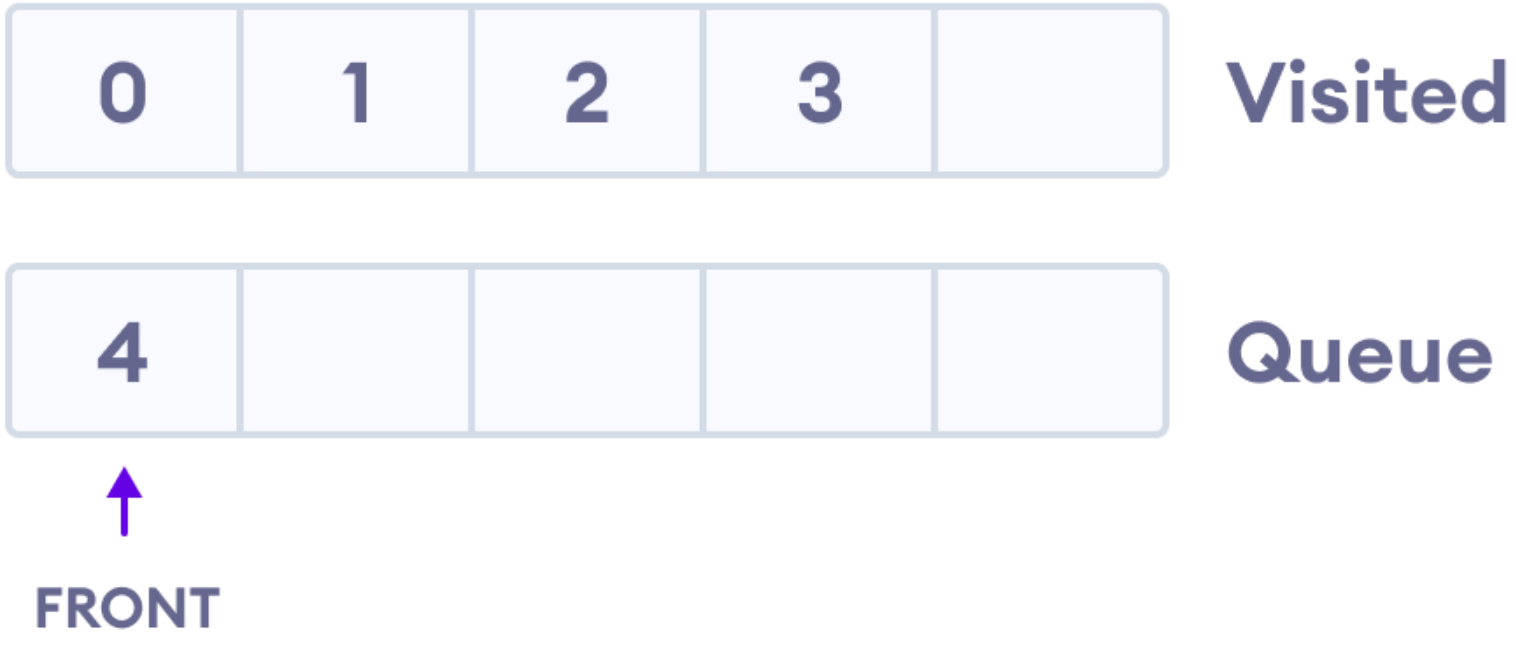
3	4			

Queue

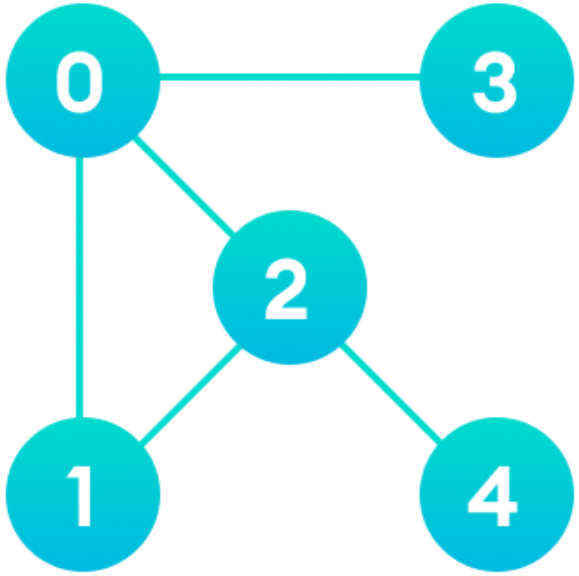
FRONT

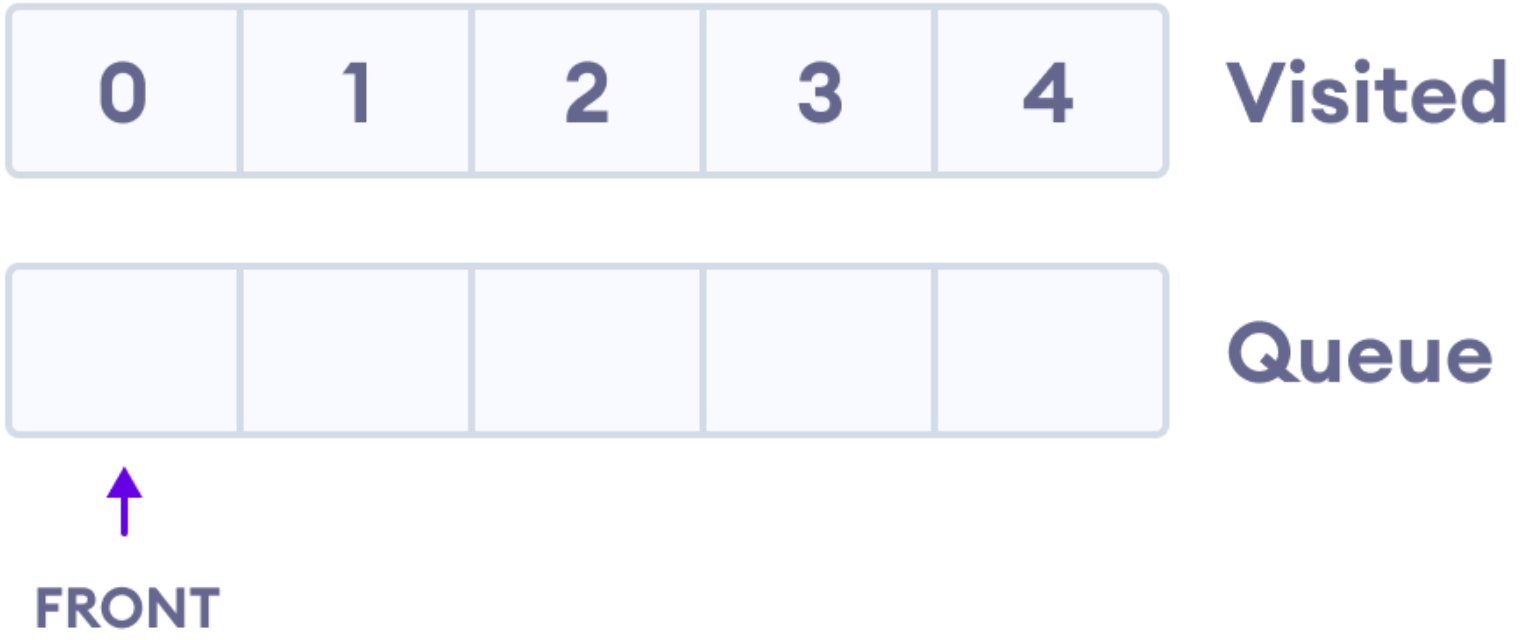
Breadth-First Search



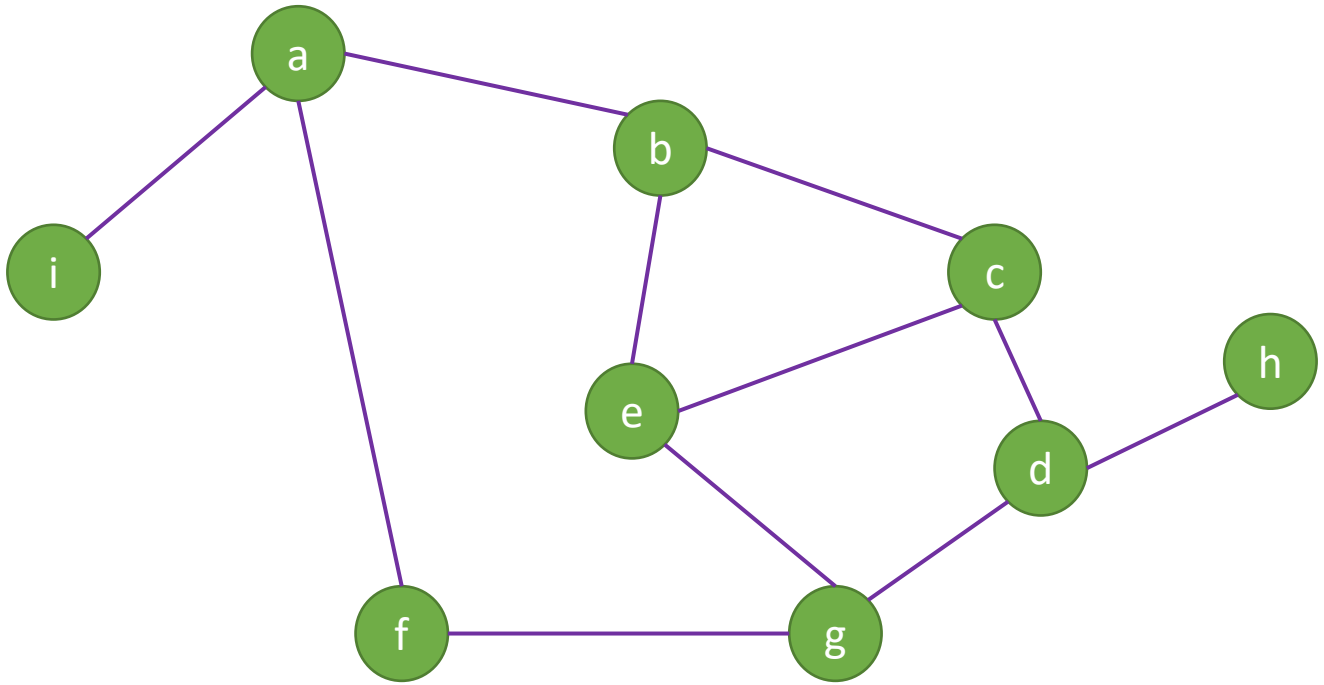


Breadth-First Search





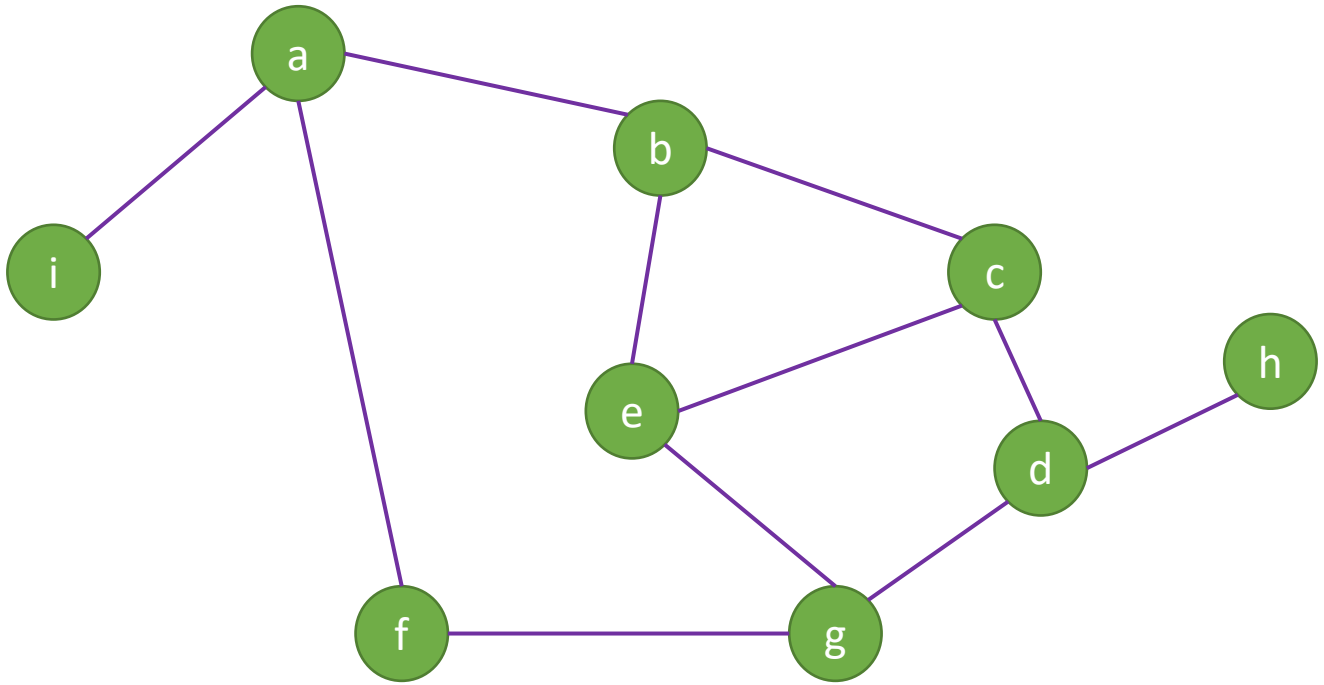
Breadth-First Search



BFS starts at ***a***:

BFS starts at ***e***:

Breadth-First Search



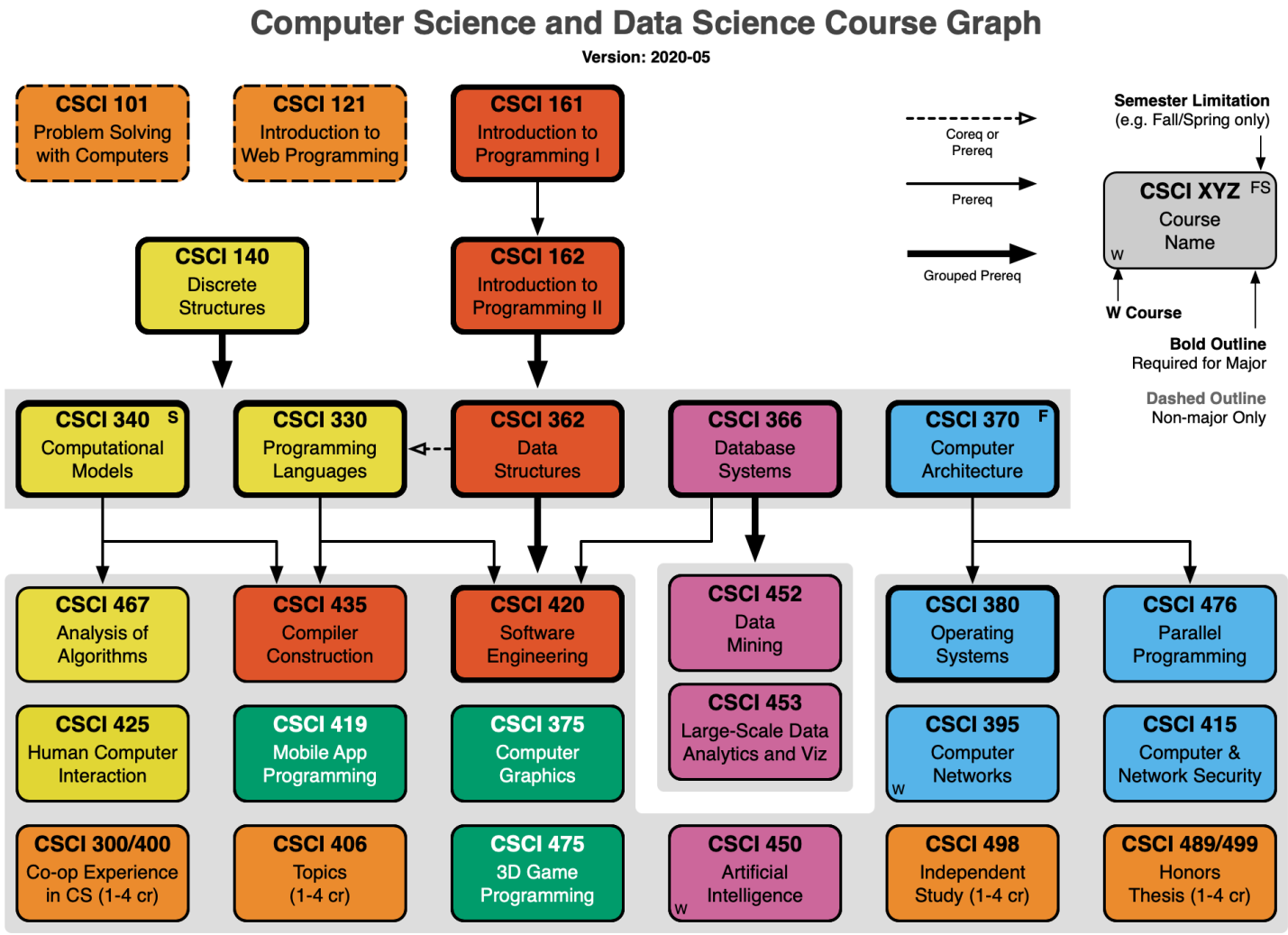
BFS starts at ***a***: a, b, f, i, c, e, g, d, h

BFS starts at ***e***: e, b, c, g, a, d, f, i, h

Topological Sorting

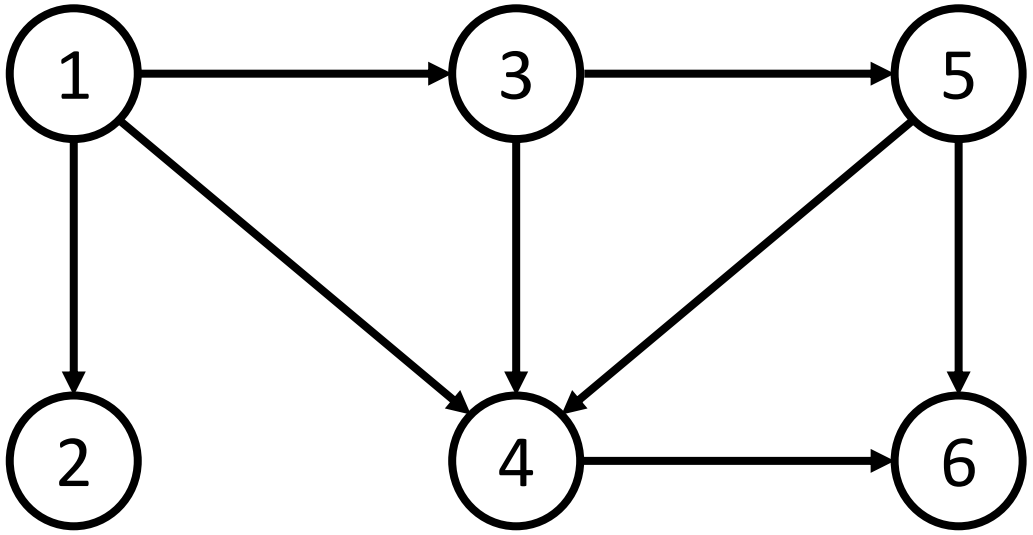
Ordering a graph

- Suppose we have a directed acyclic graph (DAG) of courses, and we want to find an order in which the courses can be taken
  - Must take all prerequisite before you can take a give course
  - How can we find a valid ordering of the vertices?



Topological sort

- Given a digraph G = (V, E), a total ordering of G's vertices such that for every edge (v, w) in E, vertex v precedes w in the ordering
- For example,
  - T = (1, 3, 2, 5, 4, 6)

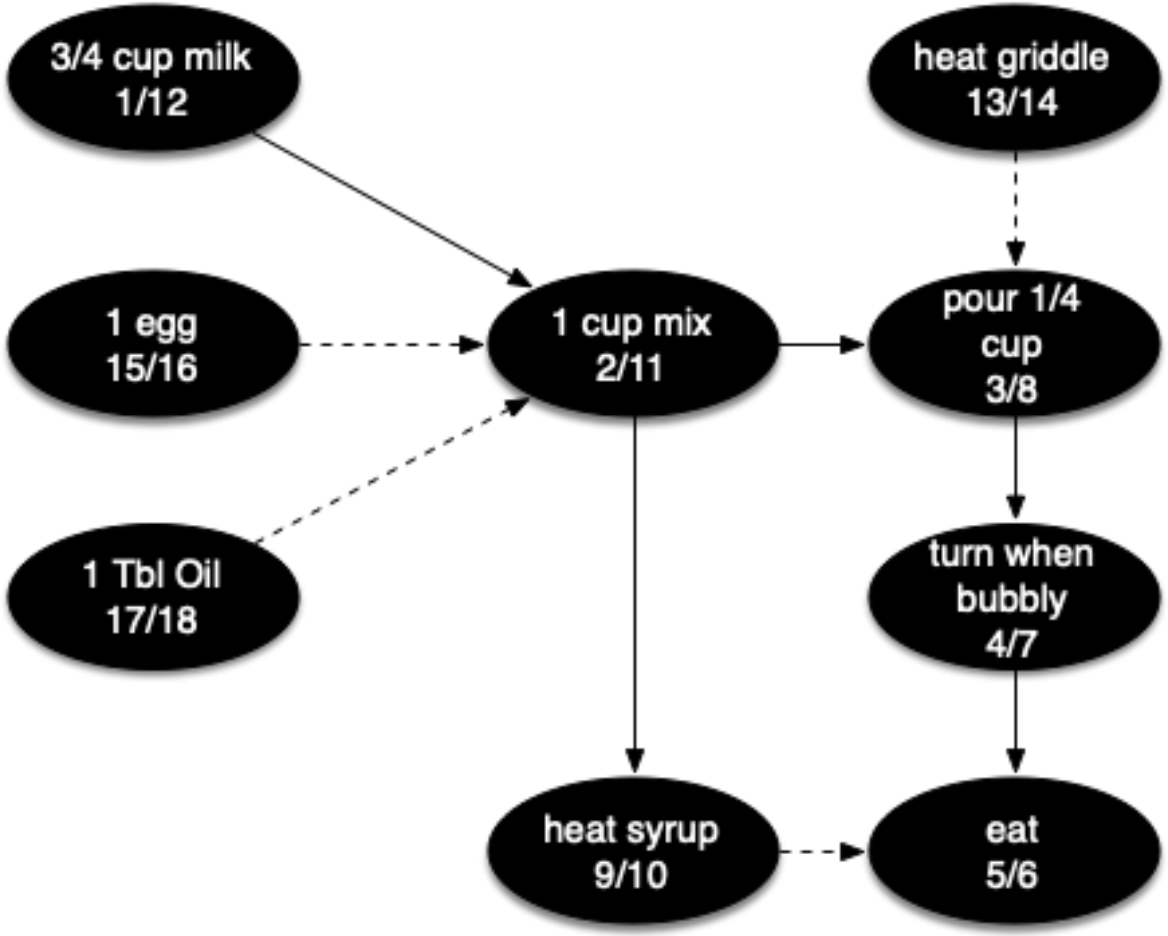


Note: there may be multiple topological orderings

E.g: T = (1, 2, 3, 5, 4, 6) is also valid

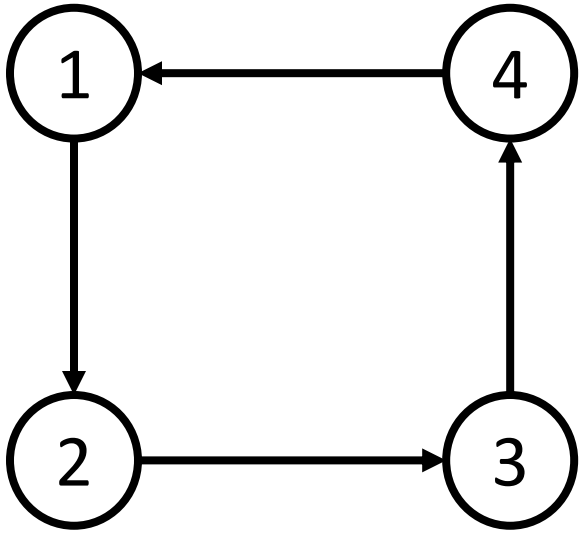
Practical Example

- Determine the order of task's list where each task needs to be completed with some tasks having to be completed first. The tasks would be nodes in a graph
- A cooking recipe



Condition

- A topological ordering of a graph is possible if and only if the graph does not contain any directed cycles.



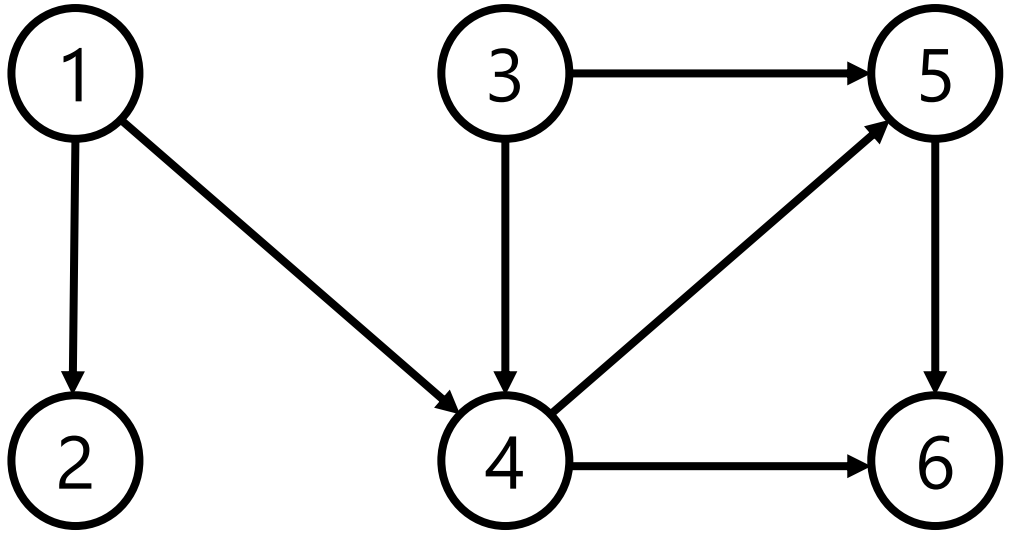
(A topological ordering of this graph **does not** exist.)

- There are 2 main algorithms for finding the topological order of a graph:
  - Kahn's Algorithm
  - Recursive DFS

DFS for Topological Sorting

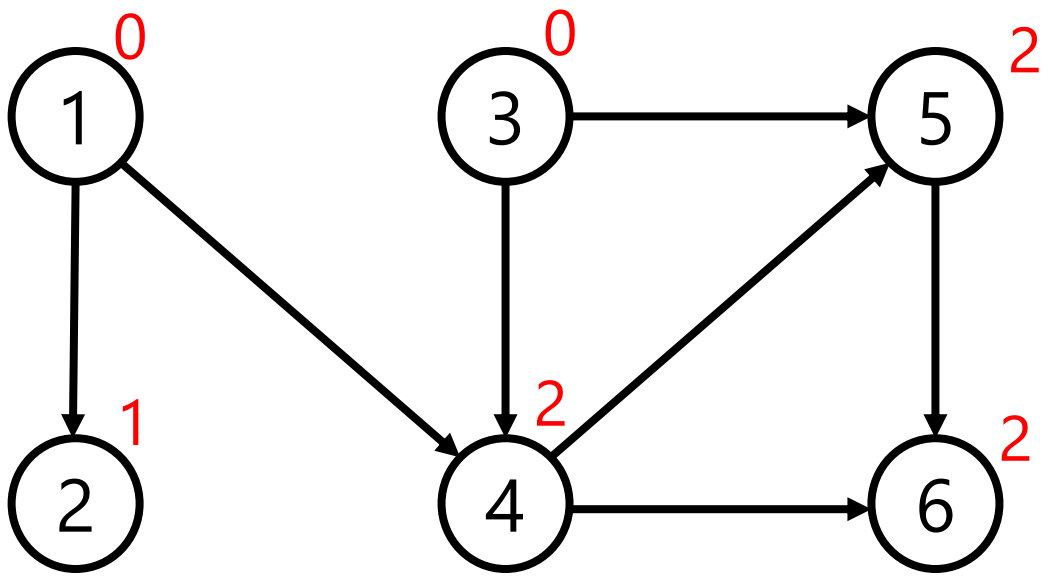
Idea: visiting node **A** means we must have visited all nodes that have **A** as a prerequisite

1. Add all vertices with an in-degree of 0 onto a list
2. Take each vertex in the list one at a time
3. Go to all its neighbors
4. If it doesn't have any neighbors that are unvisited, add it onto the front of T, mark it as visited and go back
5. Else, go to Step 3
6. When all vertices of the graph have been visited, return T


1
3
5
2
4
6

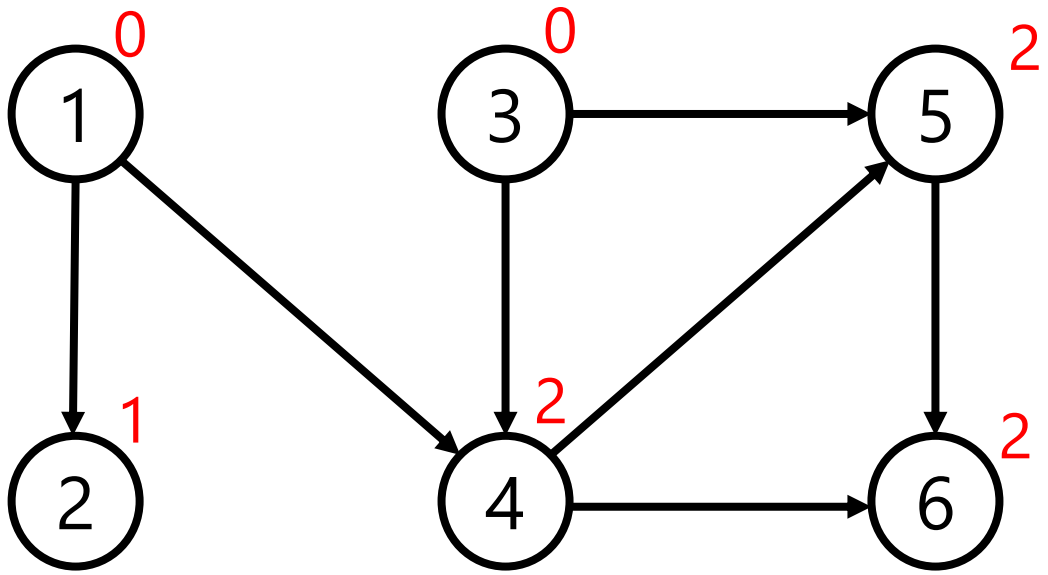
DFS for Topological Sorting

- Compute the in-degree of each vertex.



DFS for Topological Sorting

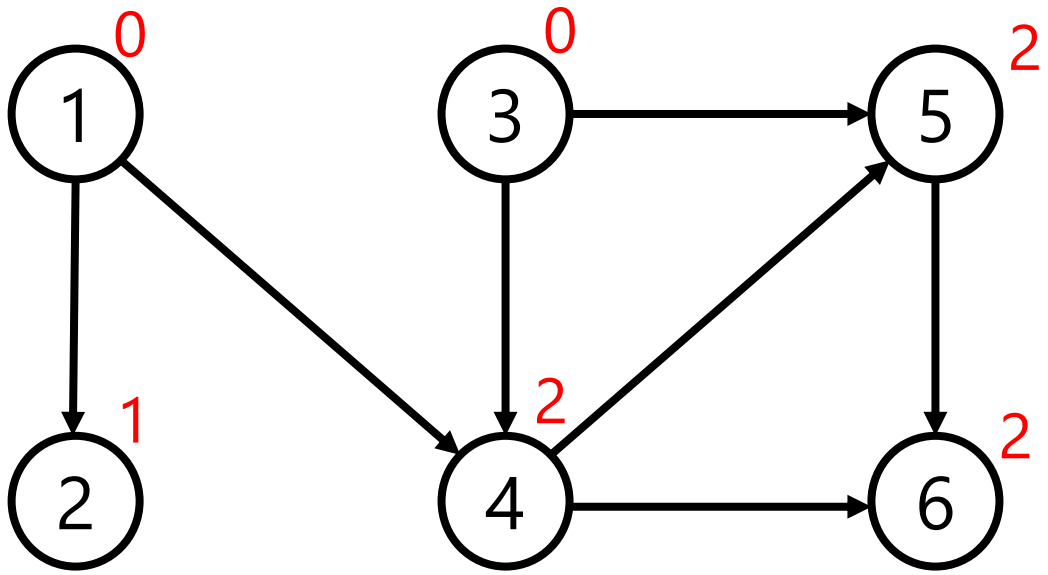
- Init the list of all vertices that has in-degree of 0
- And the result list: T



- Q = (1, 3)
- T = ()

DFS for Topological Sorting

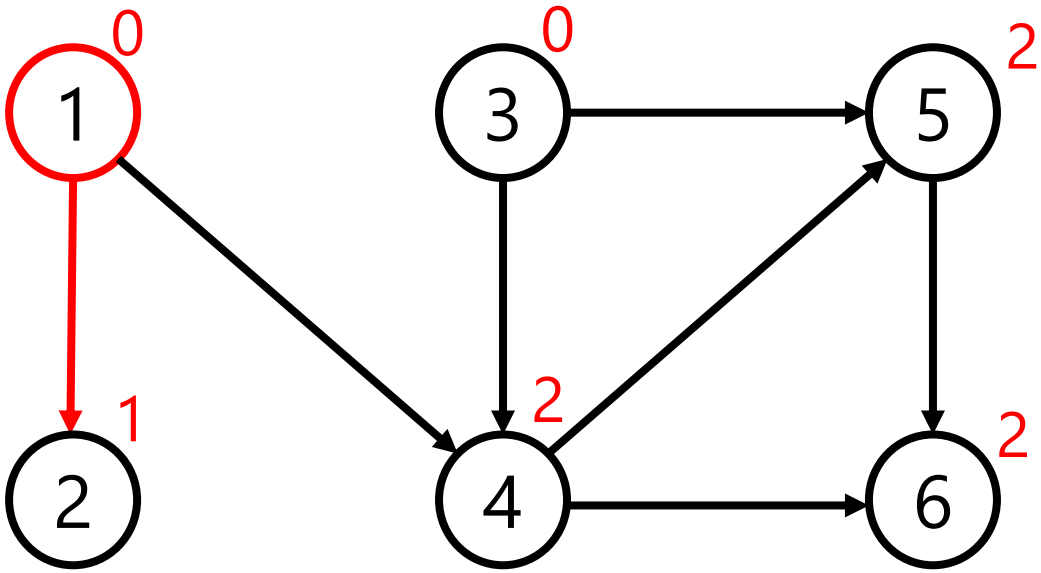
- Start from the first element in list Q
- Go to neighbors until no path



- Q = (1, 3)
- T = ()

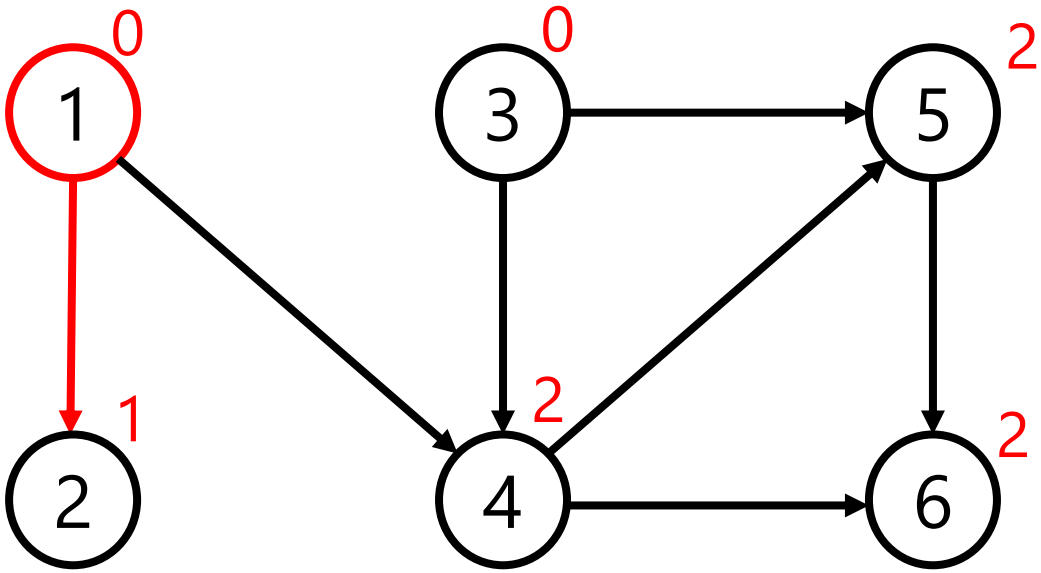
DFS for Topological Sorting

- Start from the first element in list Q
- Go to neighbors until no path



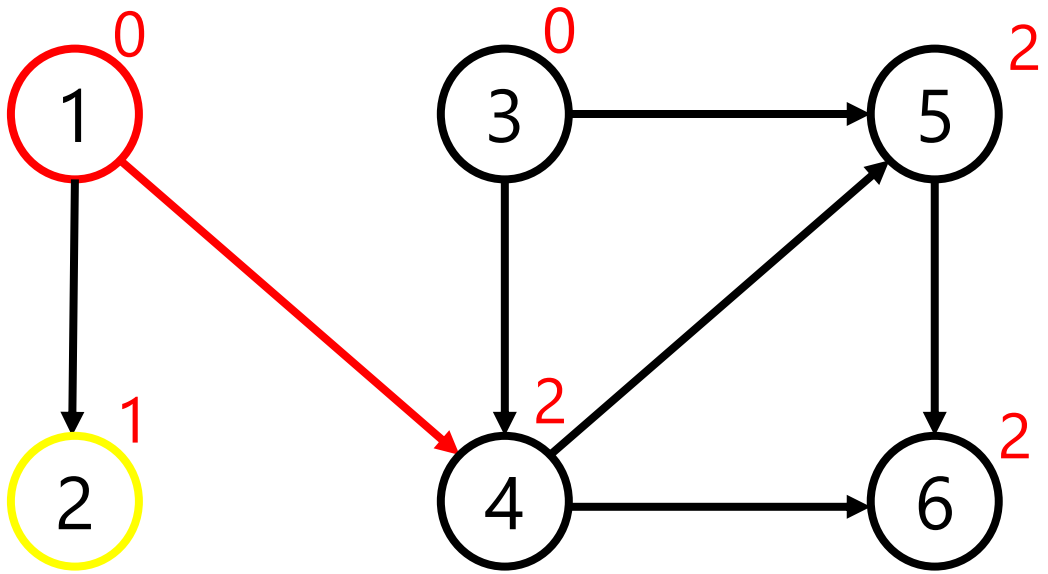
- Q = (1, 3)
- T = (2)

- Backtrack to the previous node from Node 1



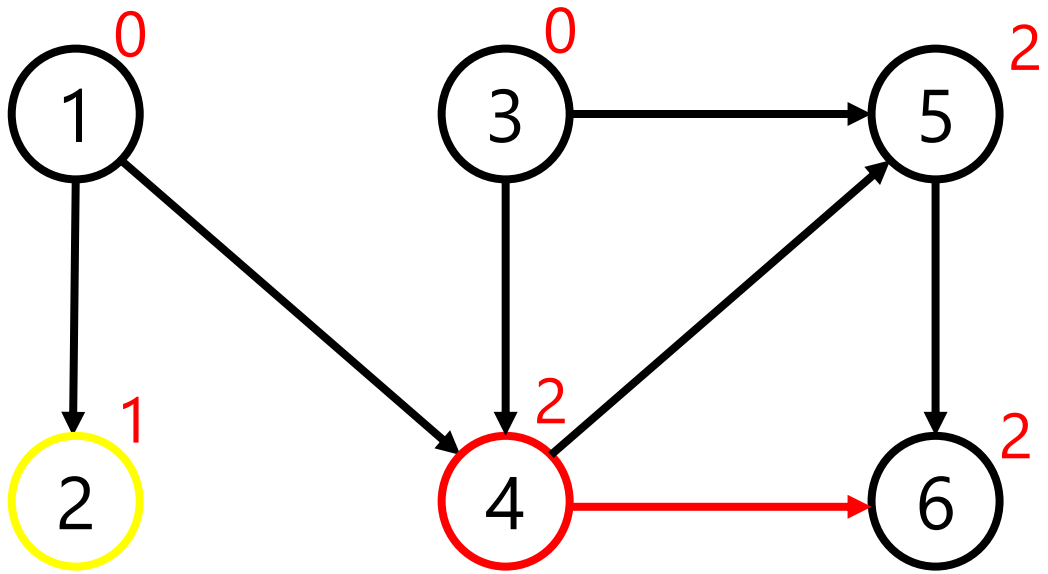
- Q = (1, 3)
- T = (2)

- Backtrack to the previous node from Node 1



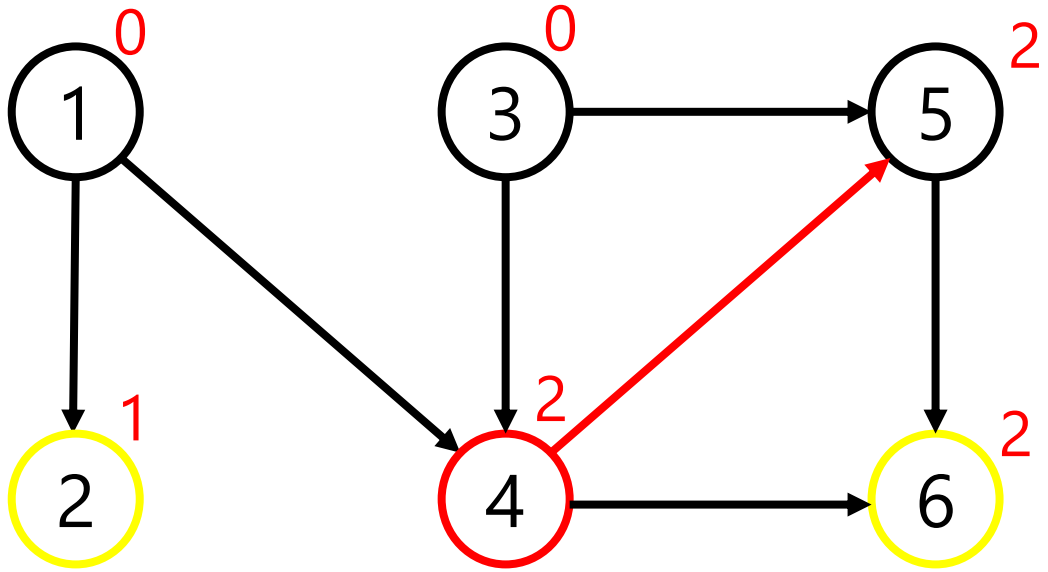
- Q = (1, 3)
- T = (2)

- Backtrack to the previous node from Node 1



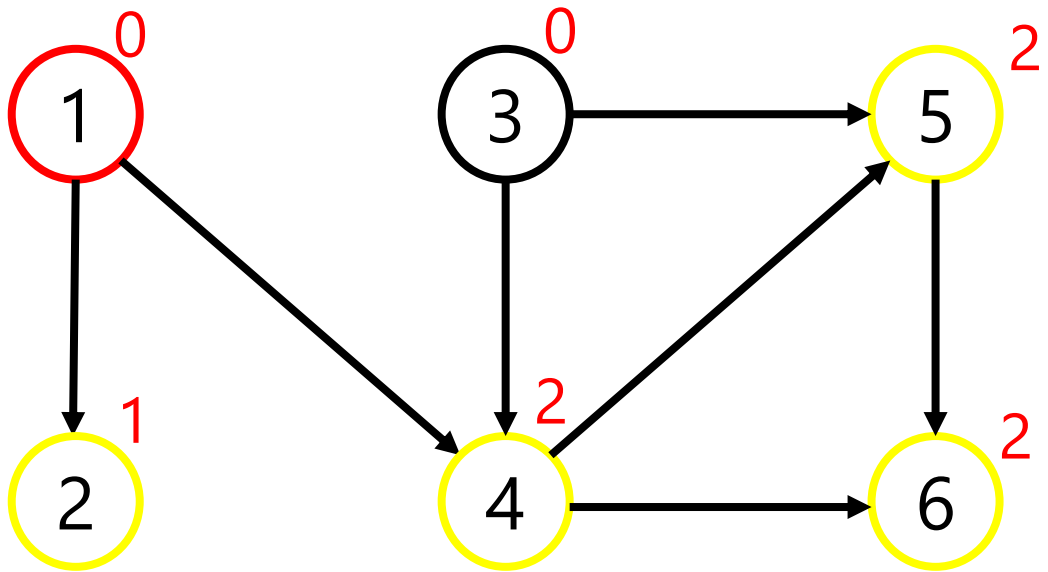
- Q = (1, 3)
- T = (6, 2)

- Backtrack to the previous node from Node 4



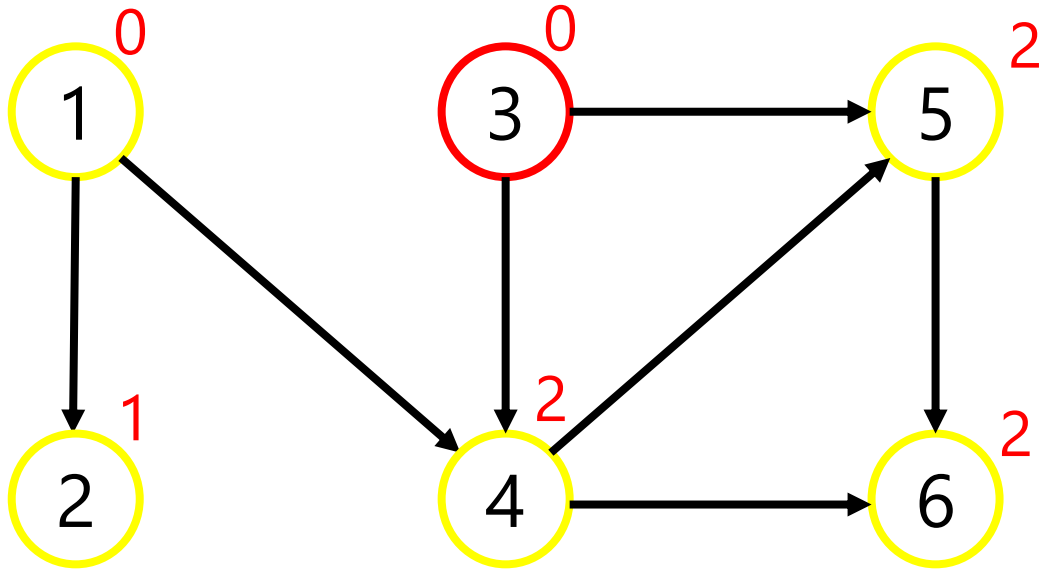
- Q = (1, 3)
- T = (5, 6, 2)

- Backtrack to the previous node from Node 1



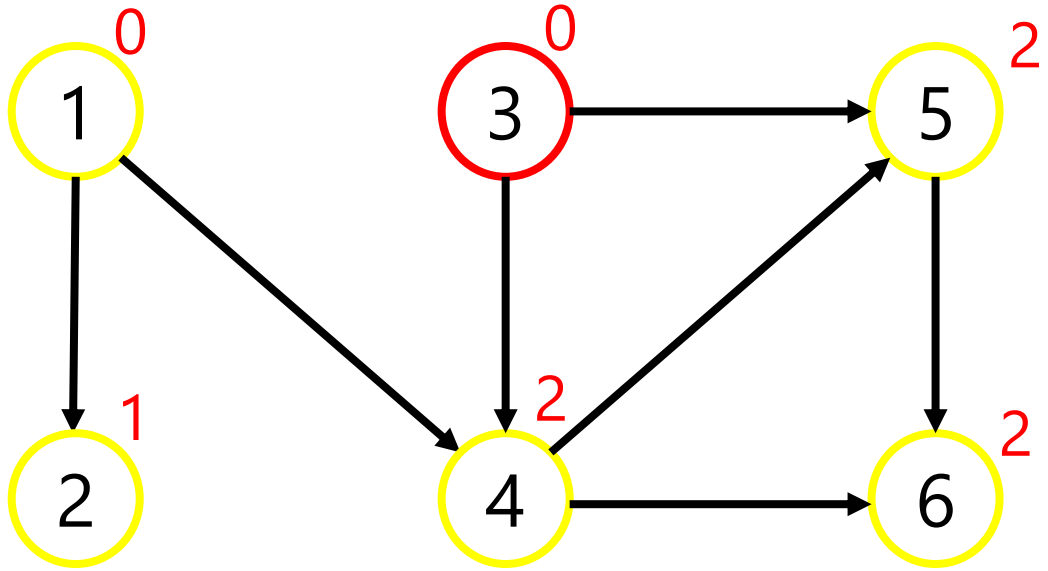
- Q = (1, 3)
- T = (4, 5, 6, 2)

- Finish the first node in Q



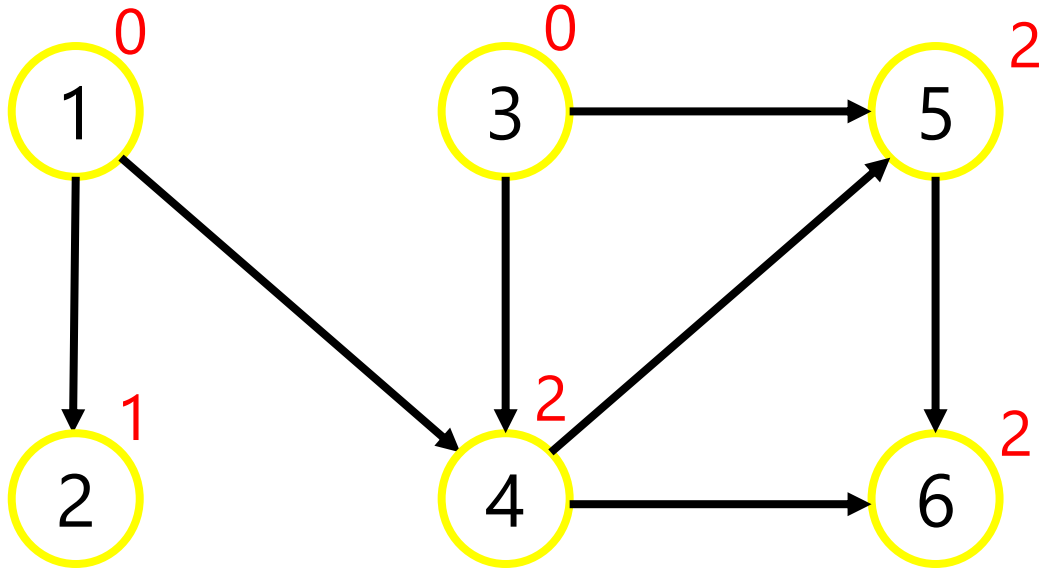
- Q = (3)
- T = (1, 4, 5, 6, 2)

- Finish the first node in Q
- Go to next starting vertex.



- Q = (3)
- T = (1, 4, 5, 6, 2)

- No unvisited neighbor's, so finish Node 3.
- Empty list Q, so this algorithm finishes.



- Q = ()
- T = (3, 1, 4, 5, 6, 2)

DFS for Topological Sorting

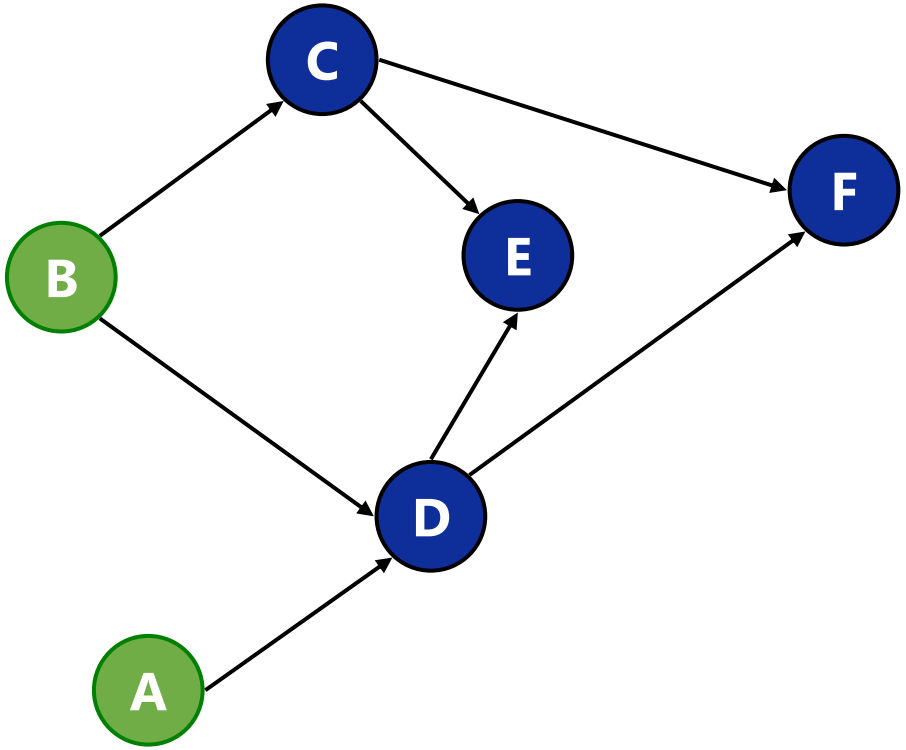
- Time complexity: O(V + E)
  - You traverse each edge once and you check each vertex once to find the in-degree of each vertex
- Space complexity: O(V + E)
  - You need a list of edges and a few lists of the vertices (technically it's 3V + E)

Kahn's Algorithm

```
function topologicalSort():
   ordering := { }.
   Repeat until graph is empty:
      Find a vertex v with in-degree of 0
         (If there is no such vertex, the graph cannot be
         sorted; stop.)
      Delete v and all of its outgoing edges from the graph.
      ordering += v
```

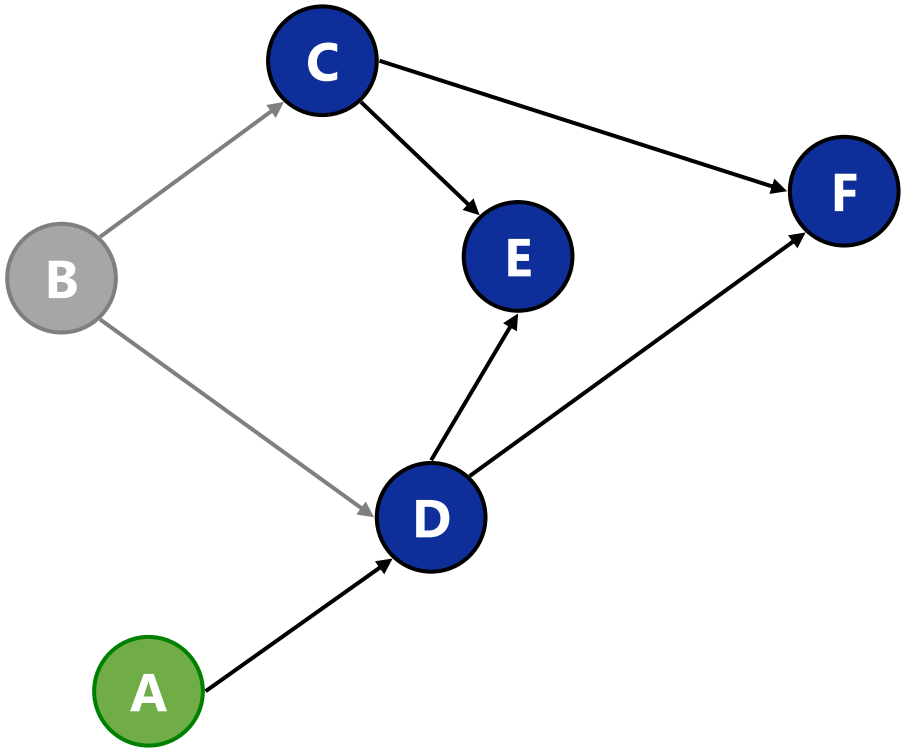
Kahn's Algorithm

- *ordering* := {}.



Expand ordering set with 0-degree vertex

- *ordering* := {B}

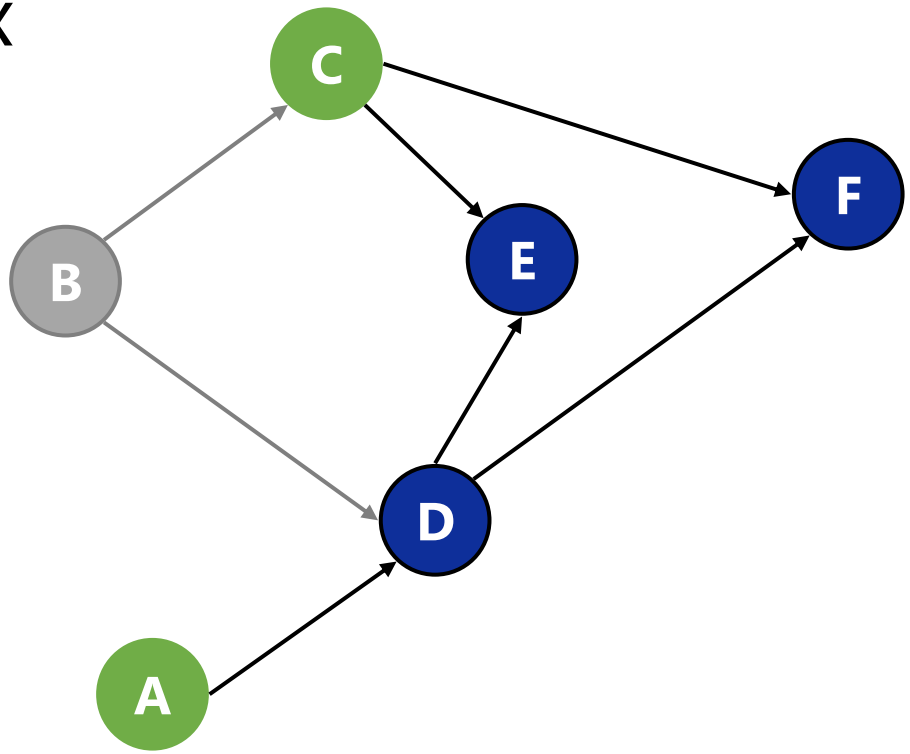


Kahn's Algorithm

- *ordering* := {B}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C}

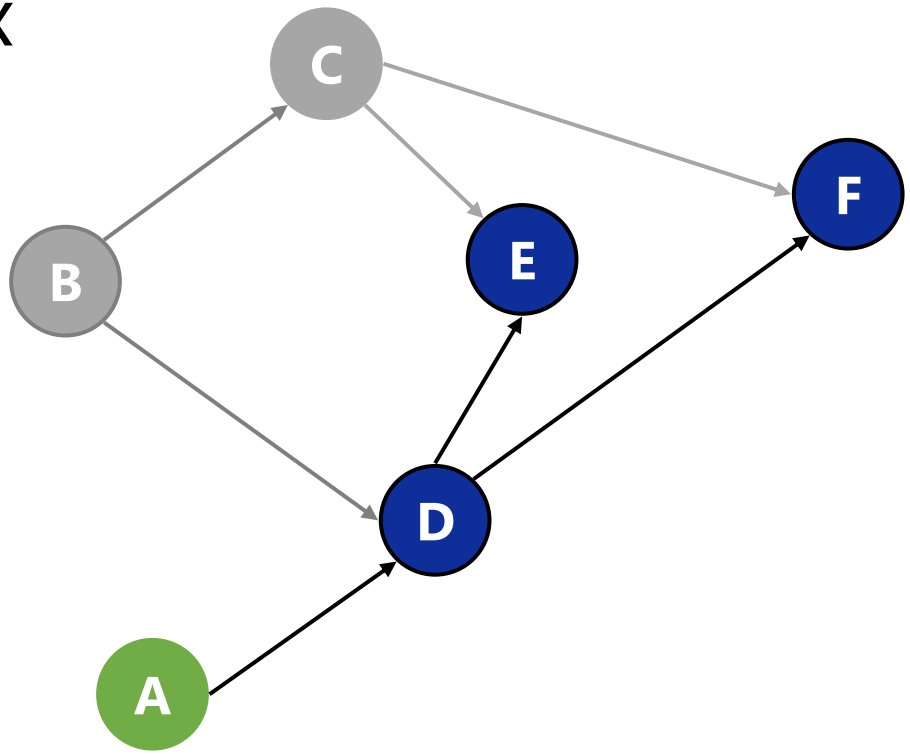


Kahn's Algorithm

- *ordering* := {B, C}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C, A}

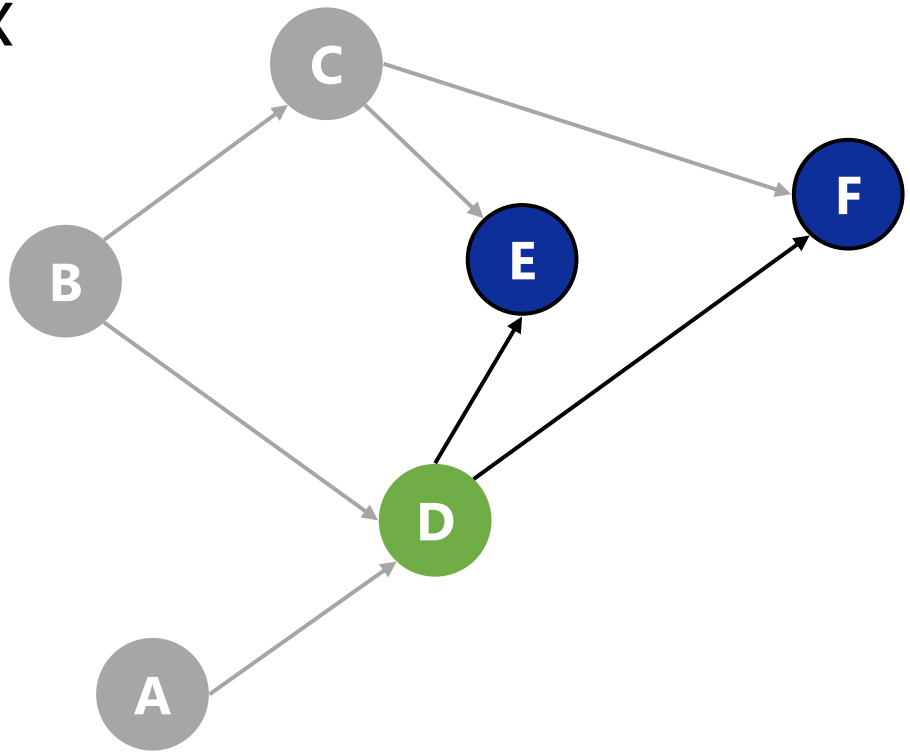


Kahn's Algorithm

- *ordering* := {B, C, A}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C, A, D}

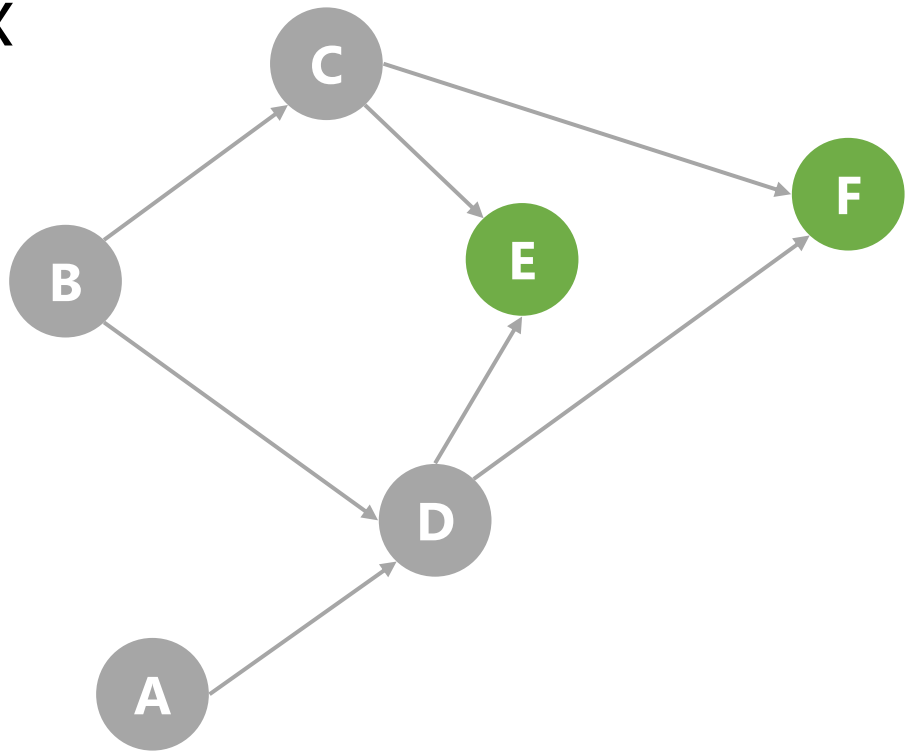


Kahn's Algorithm

- *ordering* := {B, C, A, D}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C, A, D, F}

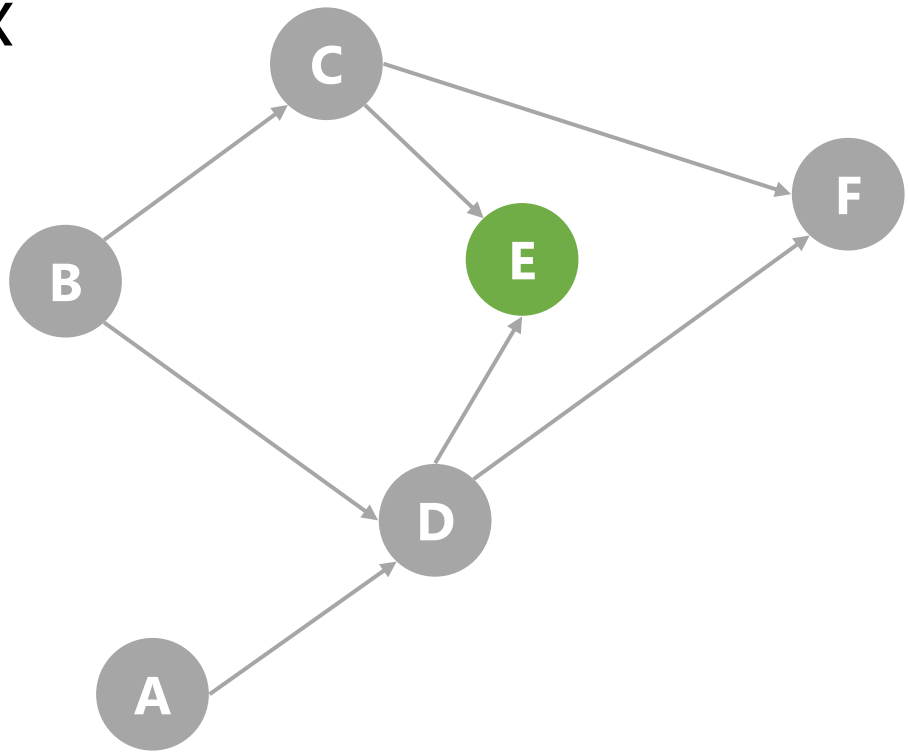


Kahn's Algorithm

- *ordering* := {B, C, A, D, F}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C, A, D, F, E}

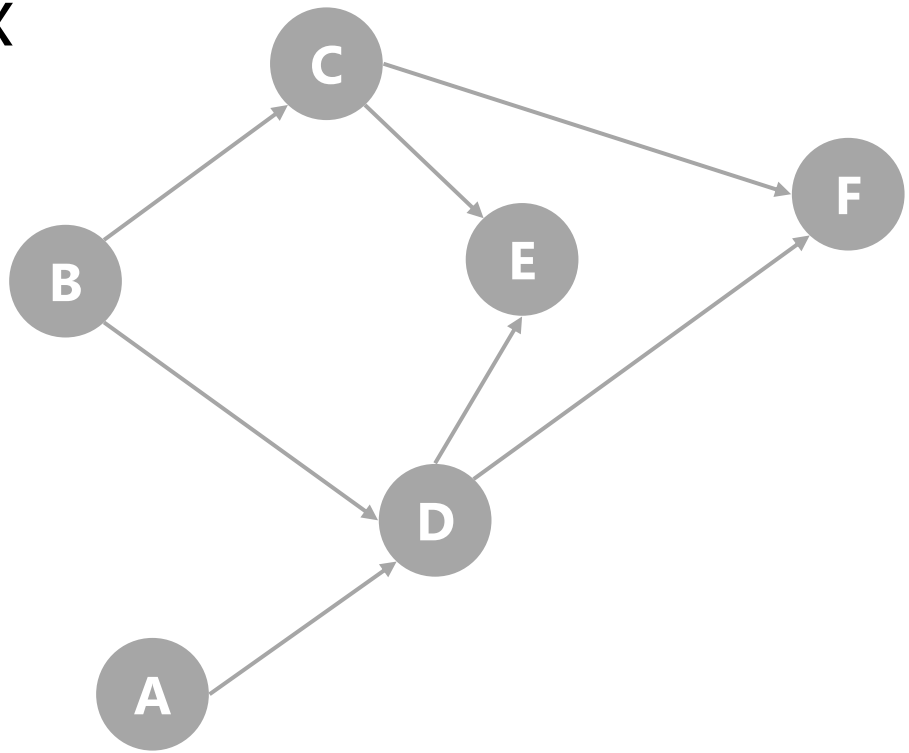


Kahn's Algorithm

- *ordering* := {B, C, A, D, F}

Expand ordering set with 0-degree vertex

- *ordering* := {B, C, A, D, F, E}



- *Revised algorithm*

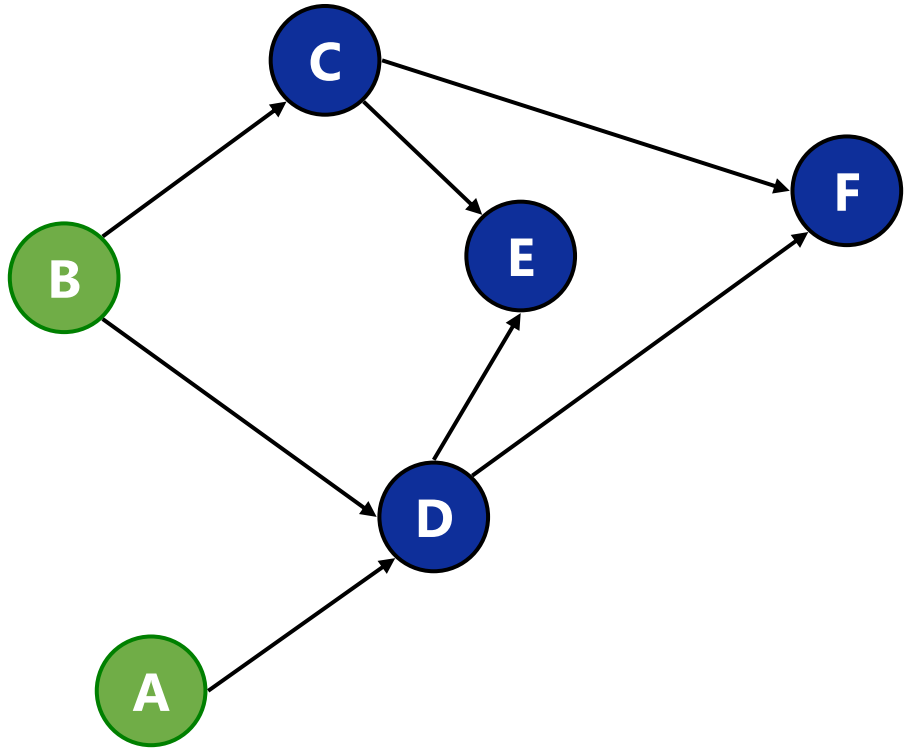
  We don't want to literally delete vertices and edges from the graph while trying to topological sort it; so let's revise the algorithm

Kahn's Algorithm

```
map := {each vertex → its in-degree}
queue := {all vertices with in-degree = 0}
ordering := { }
Repeat until queue is empty:
   Dequeue the first vertex v from the queue
   ordering += v
   Decrease the in-degree of all v's neighbors by 1 in map
   queue += {any neighbors whose in-degree is now 0}
If all vertices are processed, success
Otherwise, there is a cycle
```

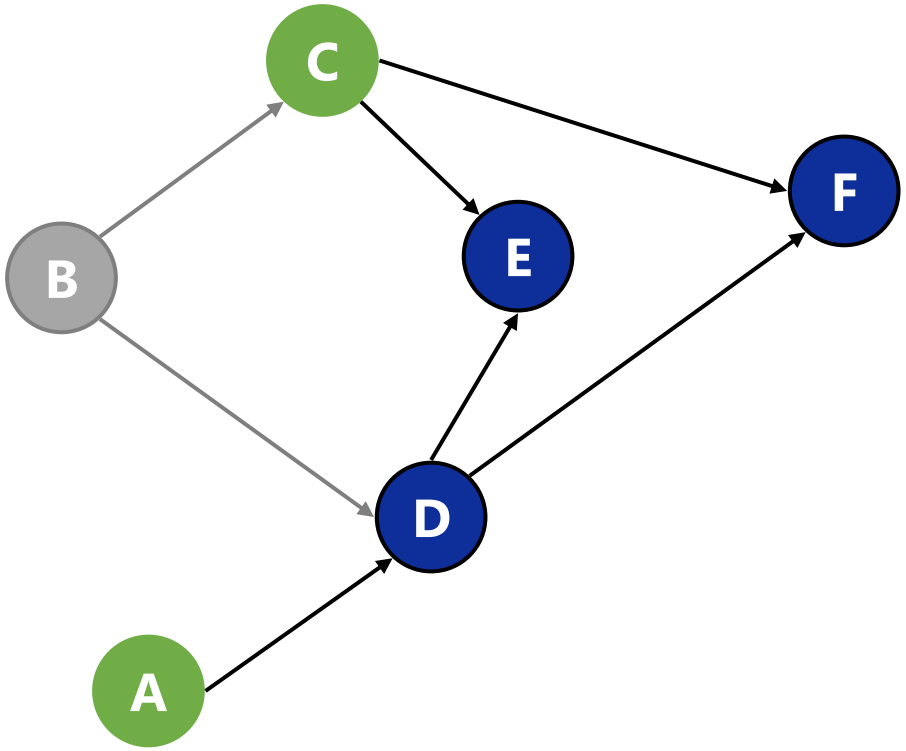
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = 1, D = 2,
  E = 2, F = 2 }
- queue:= { B, A }
- ordering:= { }



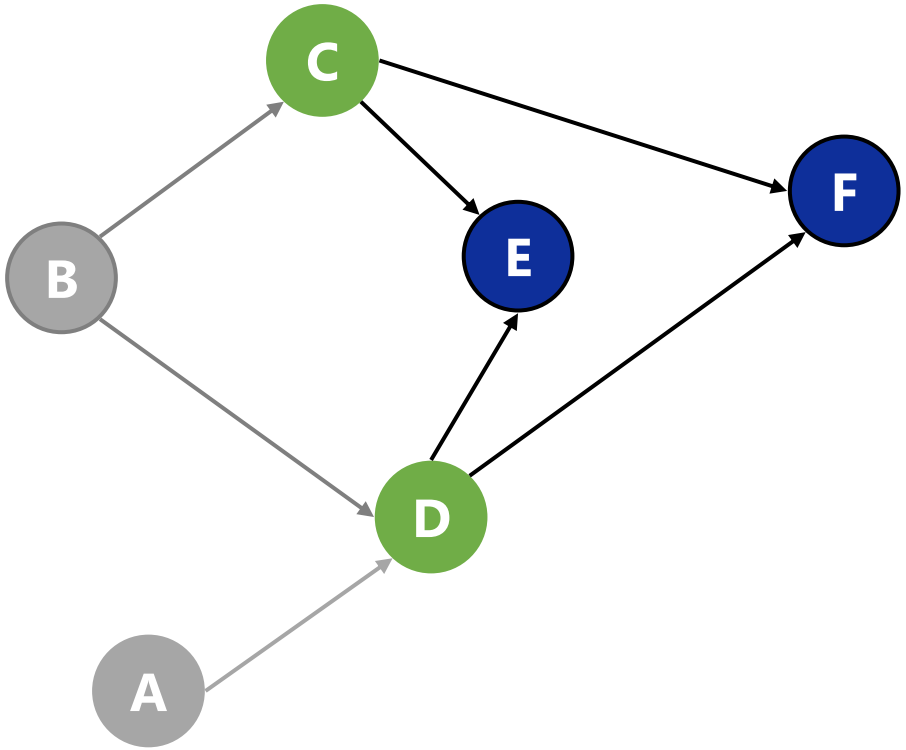
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = 0, D = 1,
  E = 2, F = 2 }
- queue:= { A , C }
- ordering:= { B }



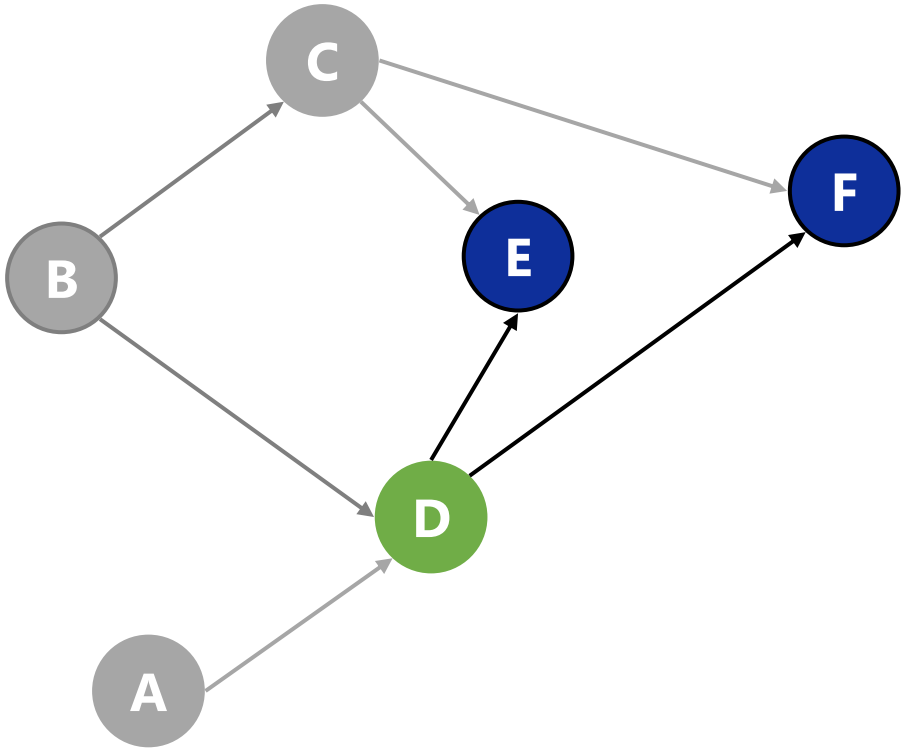
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = 0, D = 0,
  E = 2, F = 2 }
- queue:= { C , D }
- ordering:= { B, A }



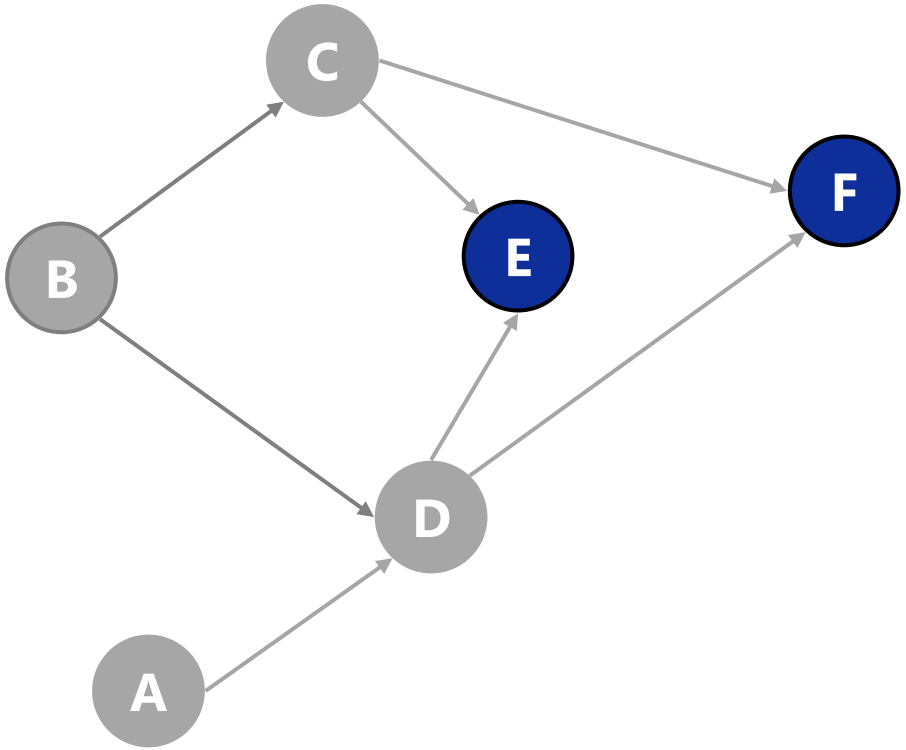
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = **0**, D = 0,
  E = 1, F = 1 }
- queue:= { **D** }
- ordering:= { B, A, C}



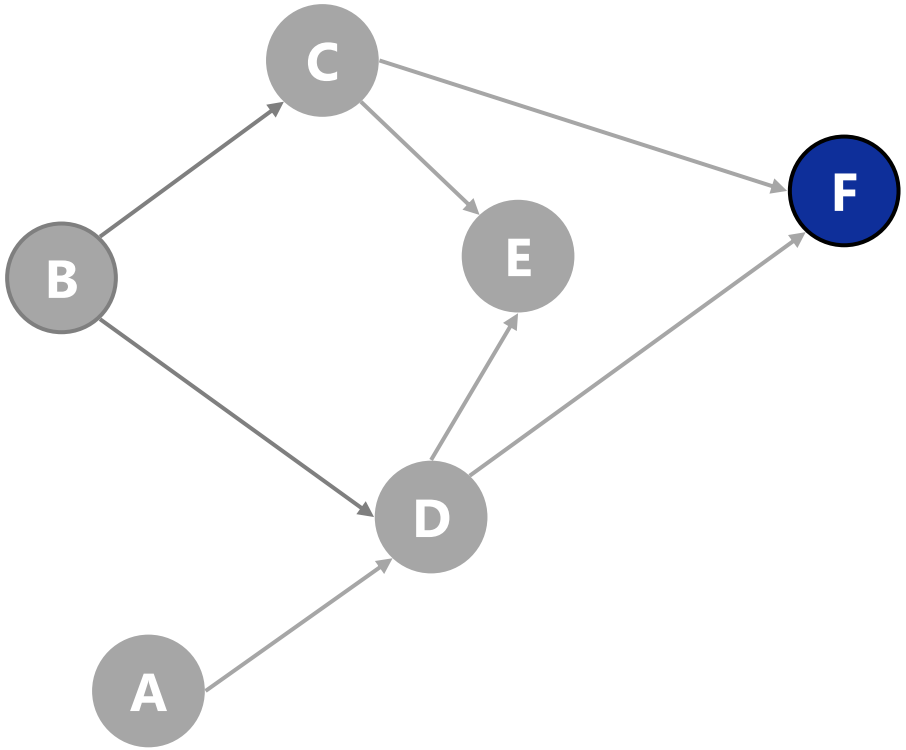
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = **0**, D = 0,
  E = 0, F = 0 }
- queue:= { **E , F**}
- ordering:= { B, A, C, D}



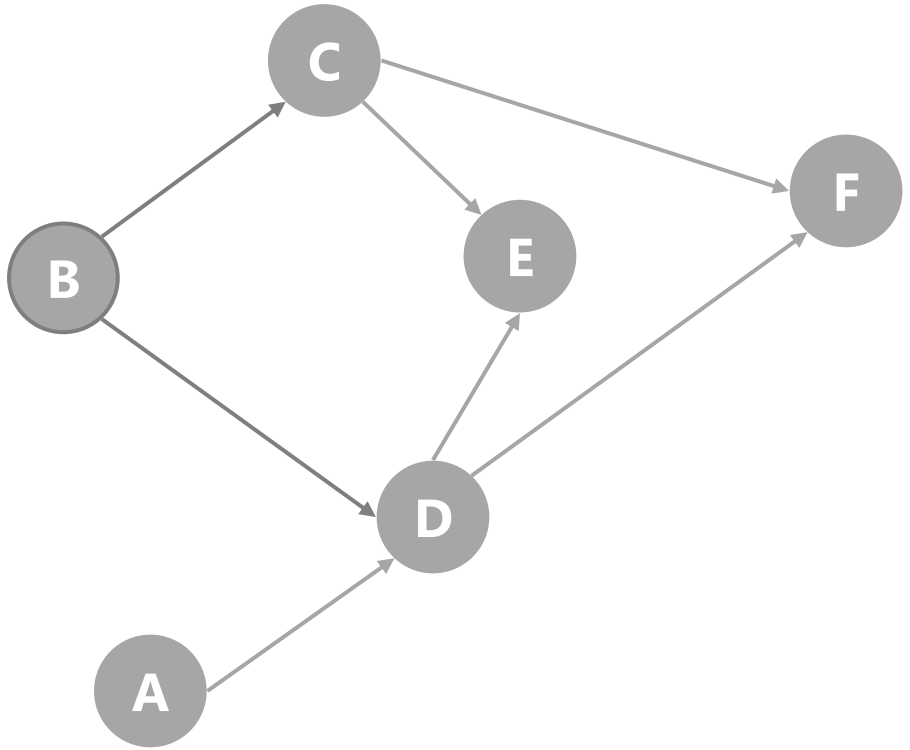
Kahn's Algorithm

- map := { A = 0, B = 0,
  C = **0**, D = 0,
  E = 0, F = 0 }
- queue:= { **F** }
- ordering:= { B, A, C, D, E}



Kahn's Algorithm

- map := { A = 0, B = 0,
  C = 0, D = 0,
  E = 0, F = 0 }
- queue:= { }
- ordering:= { B, A, C, D, E, F}



Kahn's Algorithm

function topologicalSort():

map := {each vertex → its in-degree}. // O(*V*)

queue := {all vertices with in-degree = 0}.

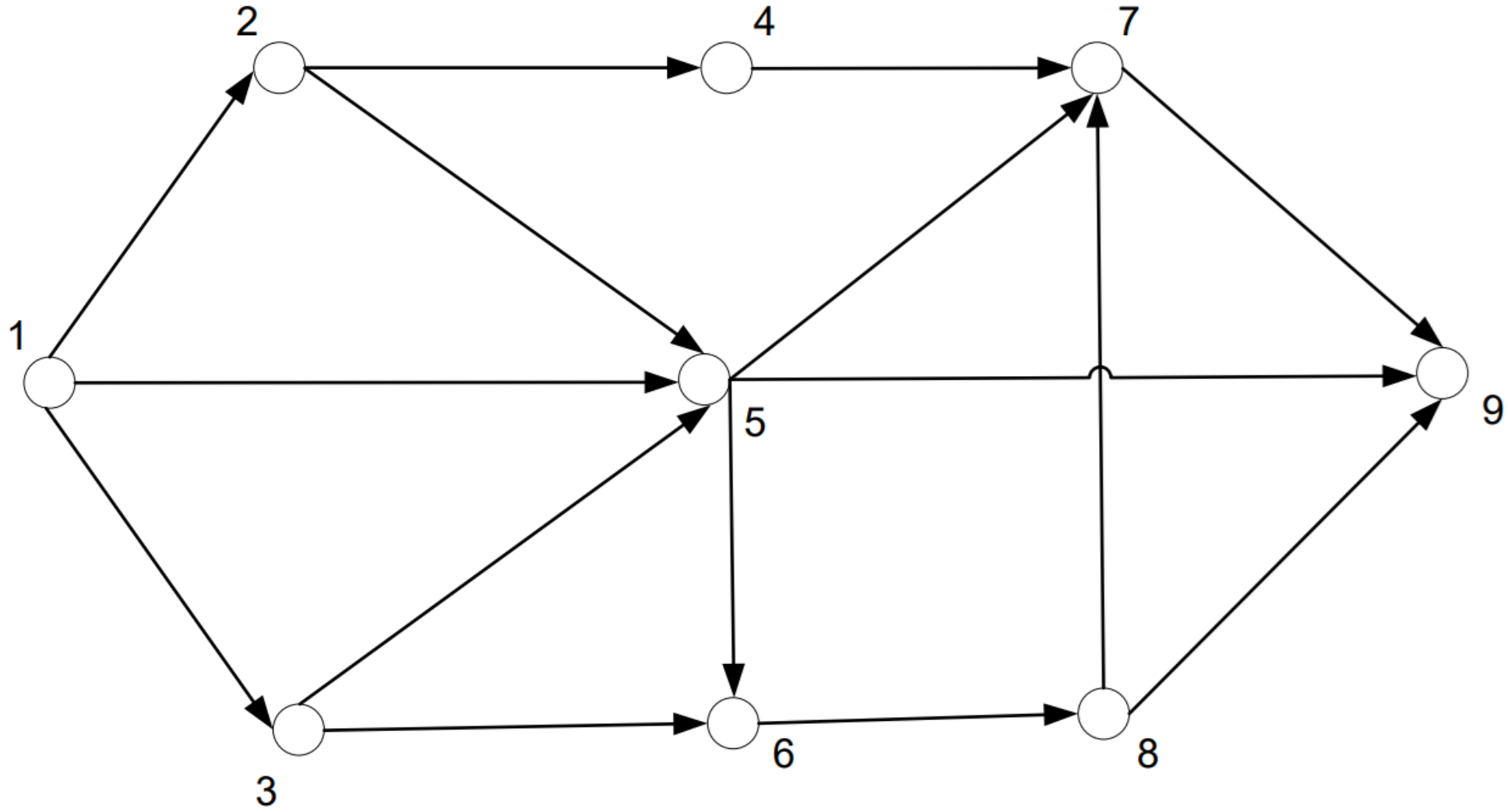
ordering := { }.

Repeat until queue is empty: // O(*V*)

- Dequeue the first vertex *v* from the queue // O(*1*)
- *ordering* += *v* // O(*1*)
- Decrease the in-degree of all *v*'s neighbors by 1 in the *map* // O(*E*) for all passes
- *queue* += {any neighbors whose in-degree is now 0}

Overall: **O(*V* + *E*)** ; essentially O(*V*) time on a sparse graph (fast!)

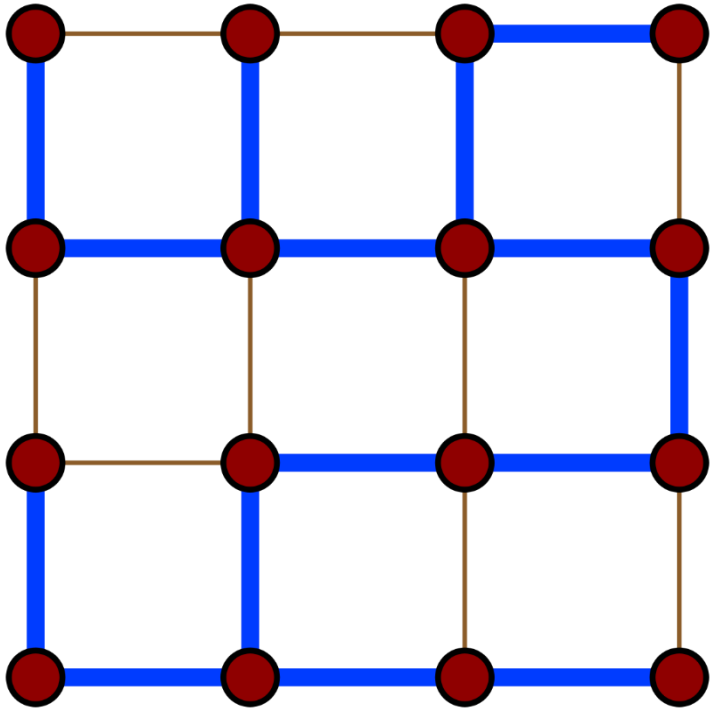
- Show a topological sort of an acyclic graph in the following figure

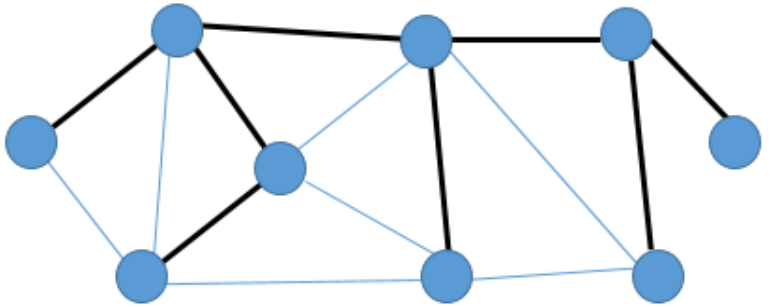


Minimum Spanning Tree

Spanning Tree

- A spanning tree
  - is a **subgraph** of undirected graph G
  - has **all the vertices** covered with **minimum** possible number of edges.
- does not have cycles
- cannot be disconnected.



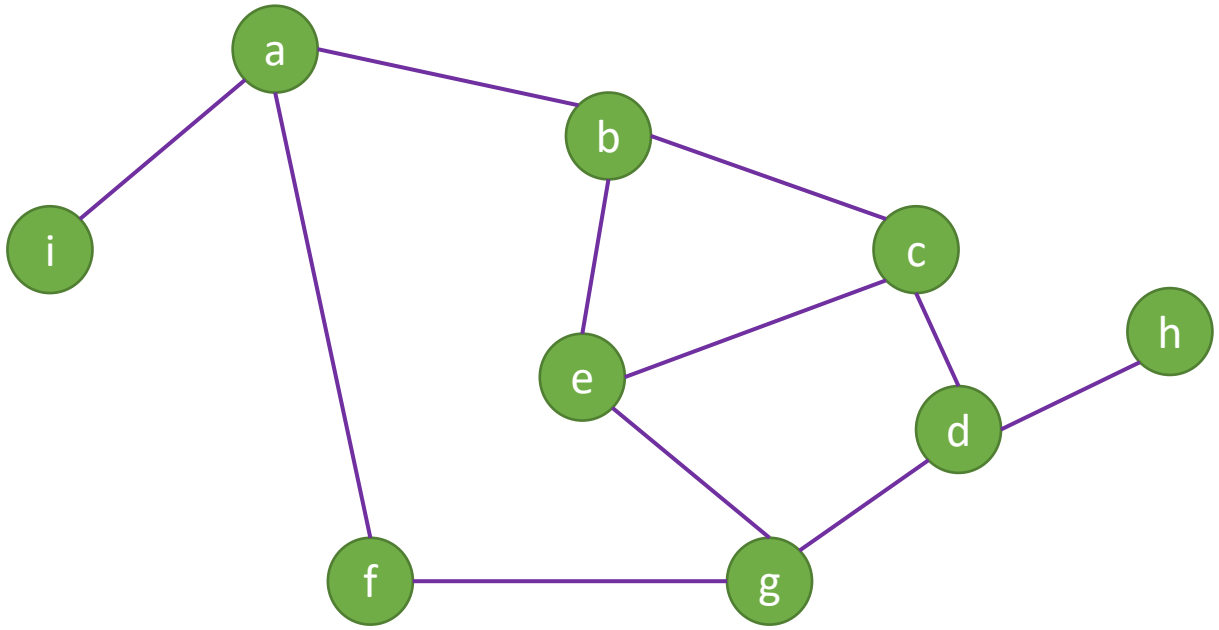


Spanning Tree

- 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).
- The spanning tree is **minimally connected**.
- The spanning tree is **maximally acyclic**.

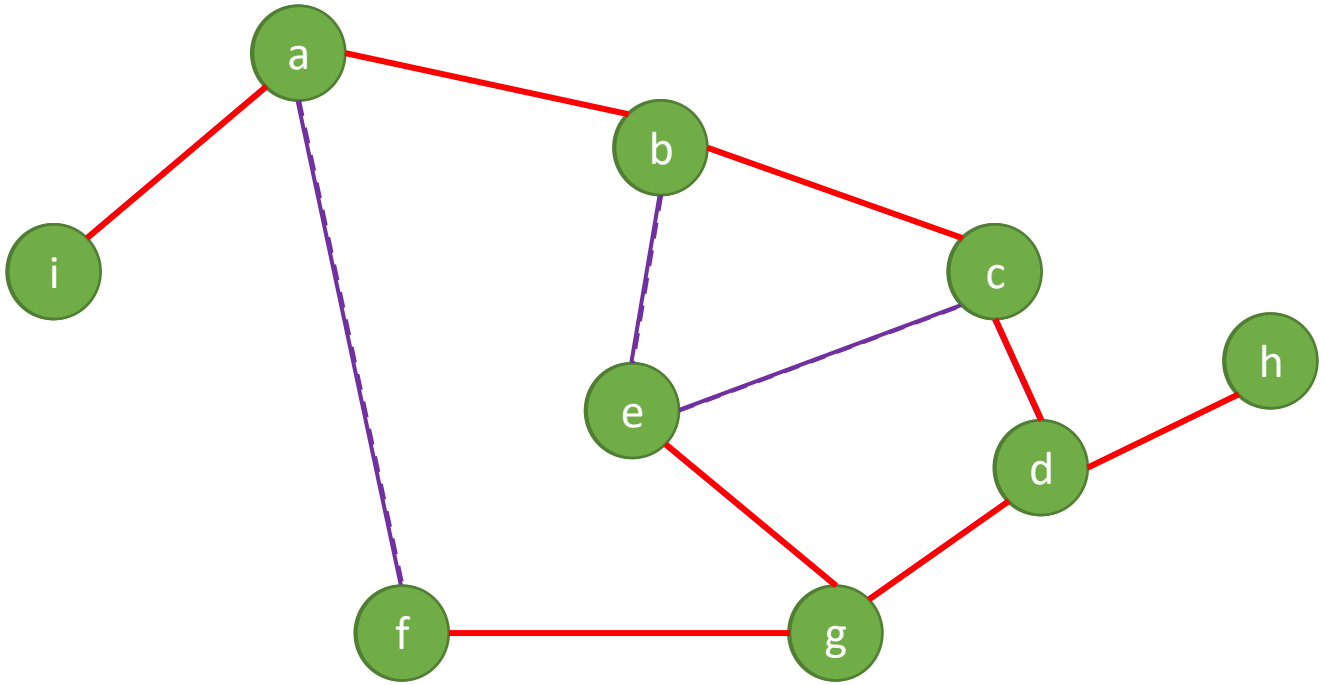
Spanning Tree

- Depth-first-search spanning tree
- Breadth-first-search spanning tree



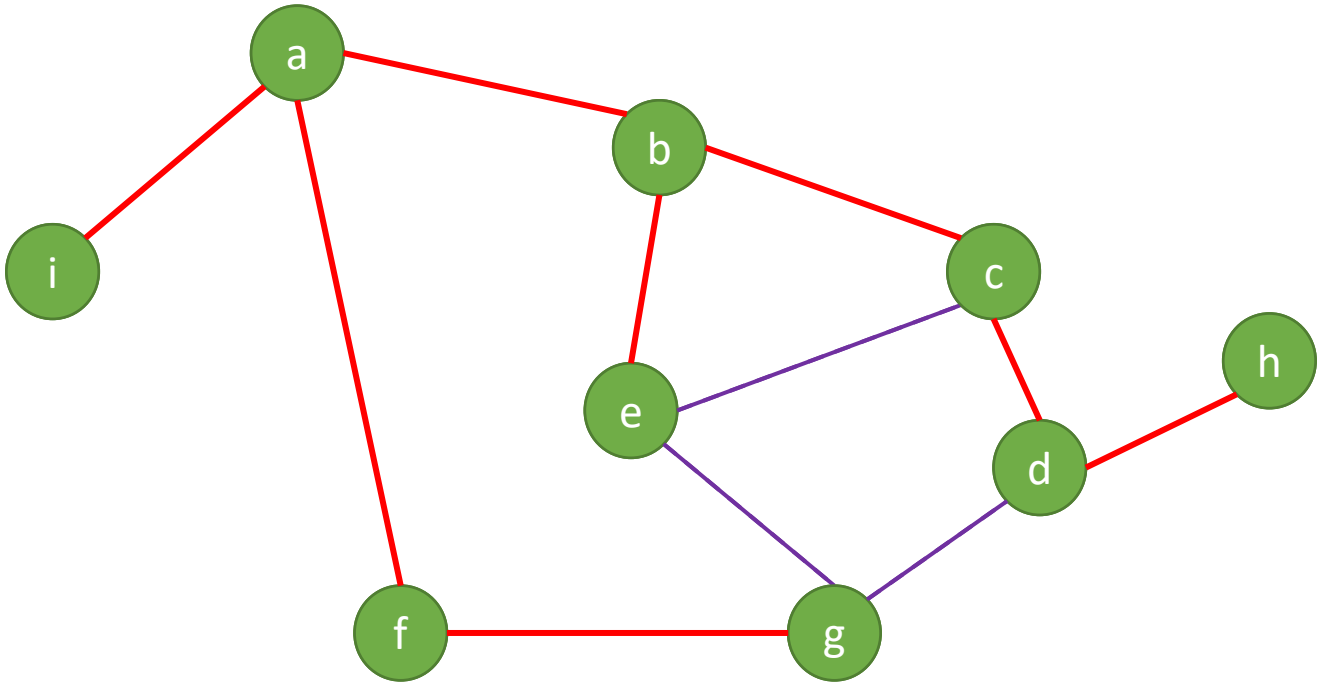
Spanning Tree

- Depth-first-search spanning tree



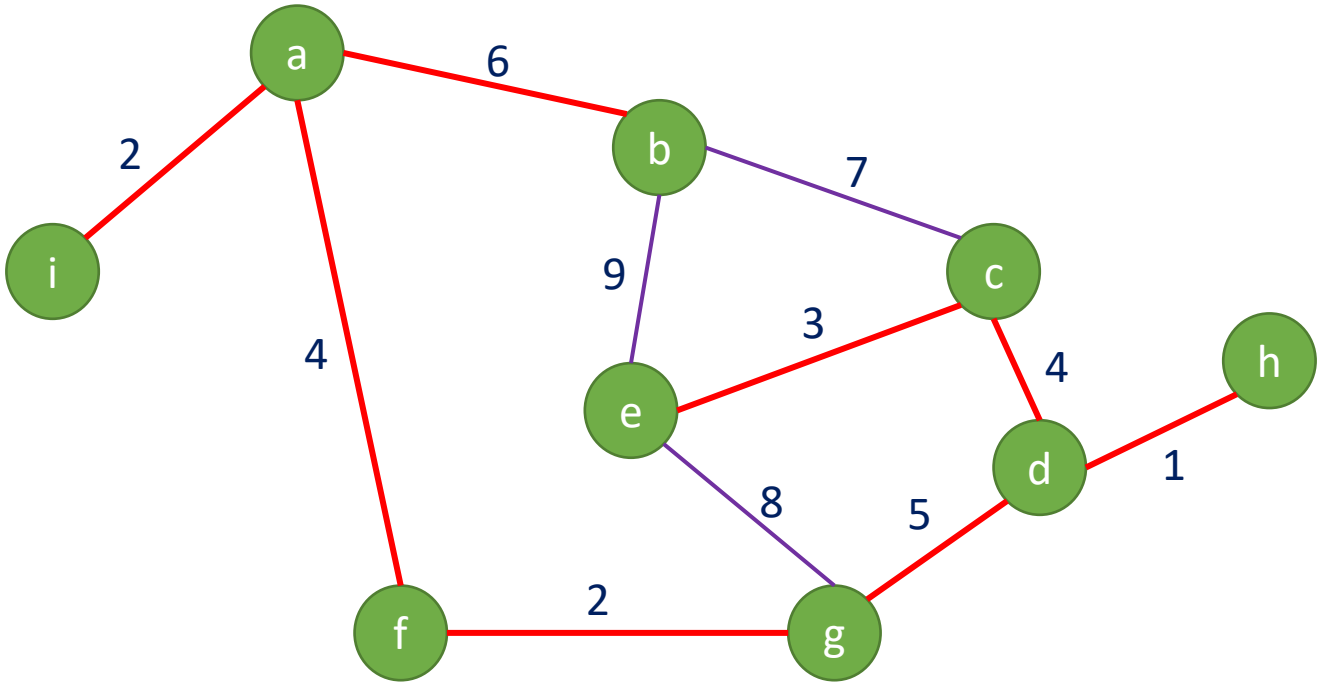
Spanning Tree

- Breadth-first-search spanning tree



Minimum Spanning Tree

- A minimum spanning tree is a spanning tree that has **minimum weight** than all other spanning trees of the same graph.



Prim's Minimum Spanning Tree

Begins with any vertex s.

Initially, the tree T contains only the starting vertex s.

At each stage,

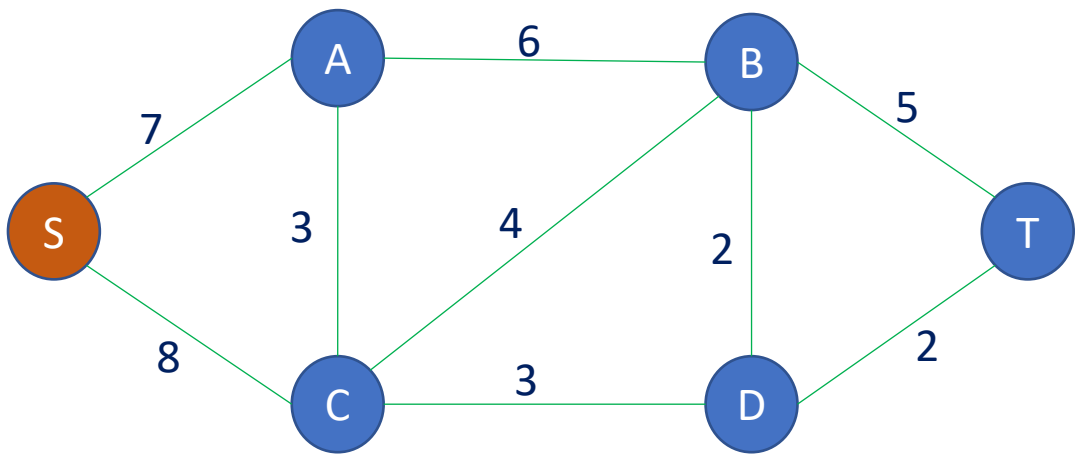
- Select the least cost edge $e(v, u)$ with v in T and u not in T.
- Add u and e to T

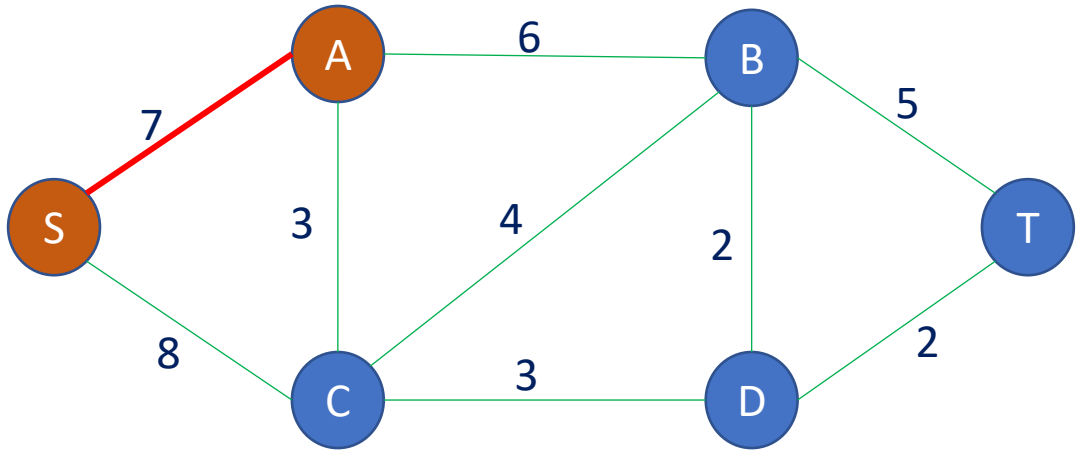
Prim's Minimum Spanning Tree

```
PrimSpanningTree(matrix[N][N], source)
{
    for v = 0 to N-1 {
        length[v] = matrix[source][v]
        parent[v] = source
    }
    Mark source //Add source to the spanning tree
    for step = 1 to N-1 {
        Find the vertex v such that length[v] is smallest and v is not in ST
        Mark v
        for all vertices u not in vertexSet
            if (length[u] > matrix[v][u]){
                length[u] = matrix[v][u]
                parent[u] = v
            }
    }
}
```

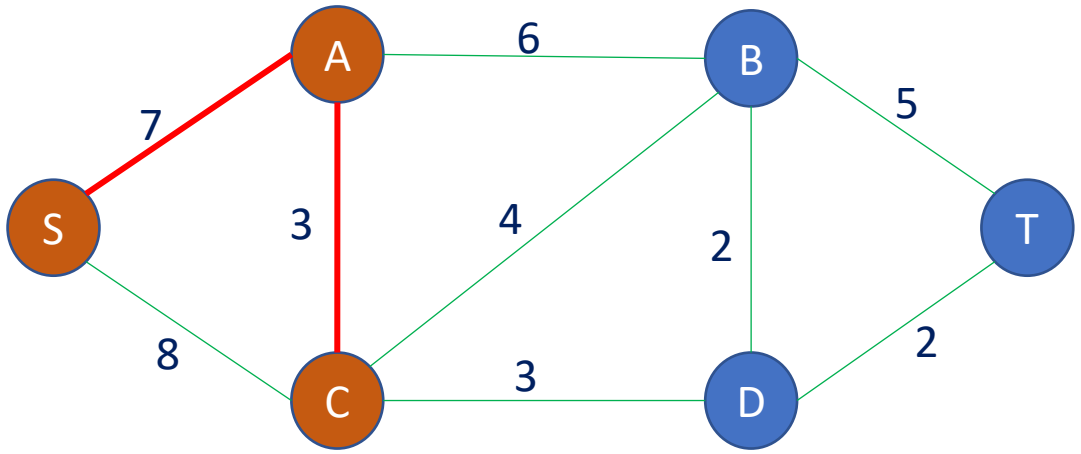
Prim's Minimum Spanning Tree

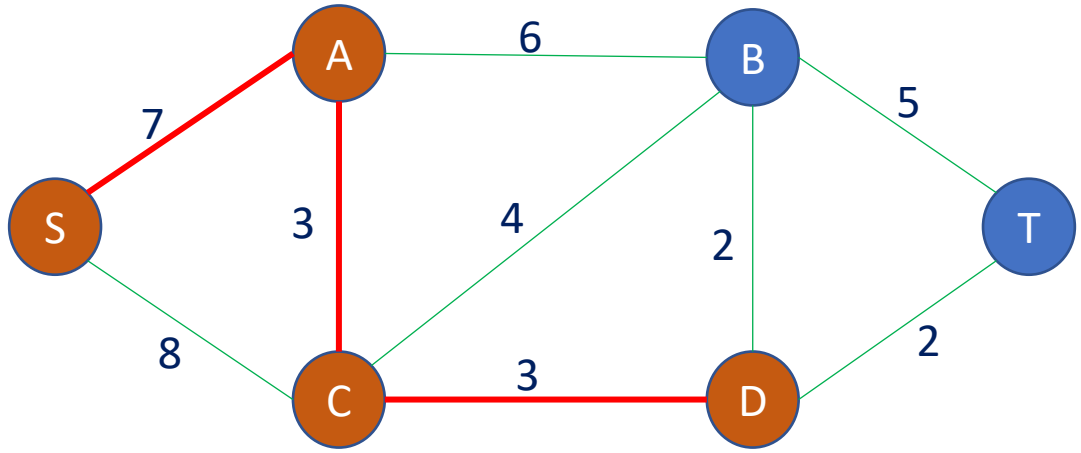
- The steps for implementing Prim's algorithm are as follows:
  - Initialize the minimum spanning tree with a vertex chosen at random.
  - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
  - Keep repeating step 2 until we get a minimum spanning tree


A
B
S
T
C
D
7
6
5
3
4
2
8
3
2

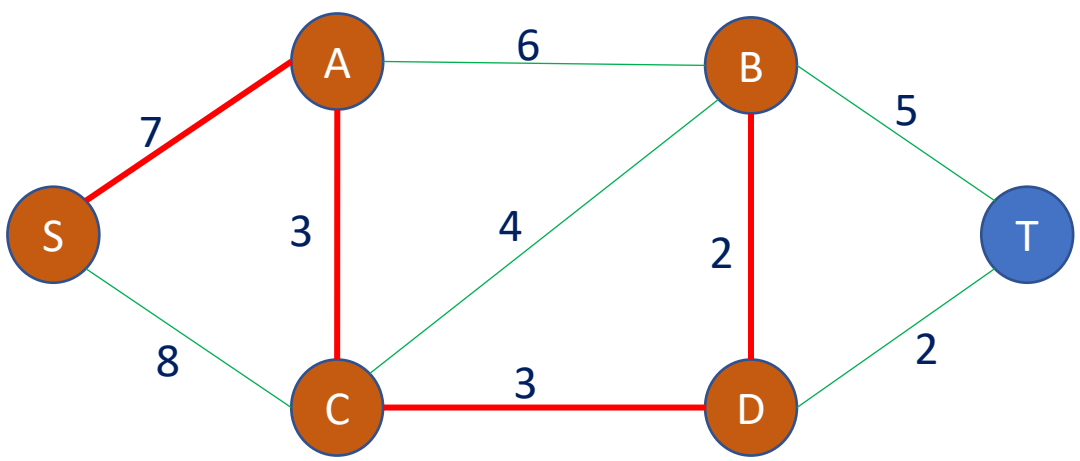

A
B
S
T
C
D
7
6
5
3
4
2
8
3
2

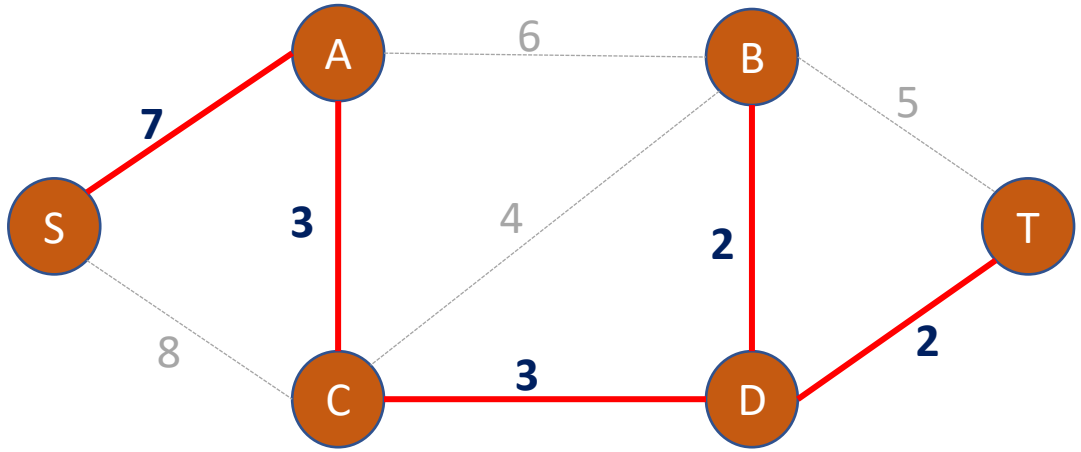
Prim's Minimum Spanning Tree





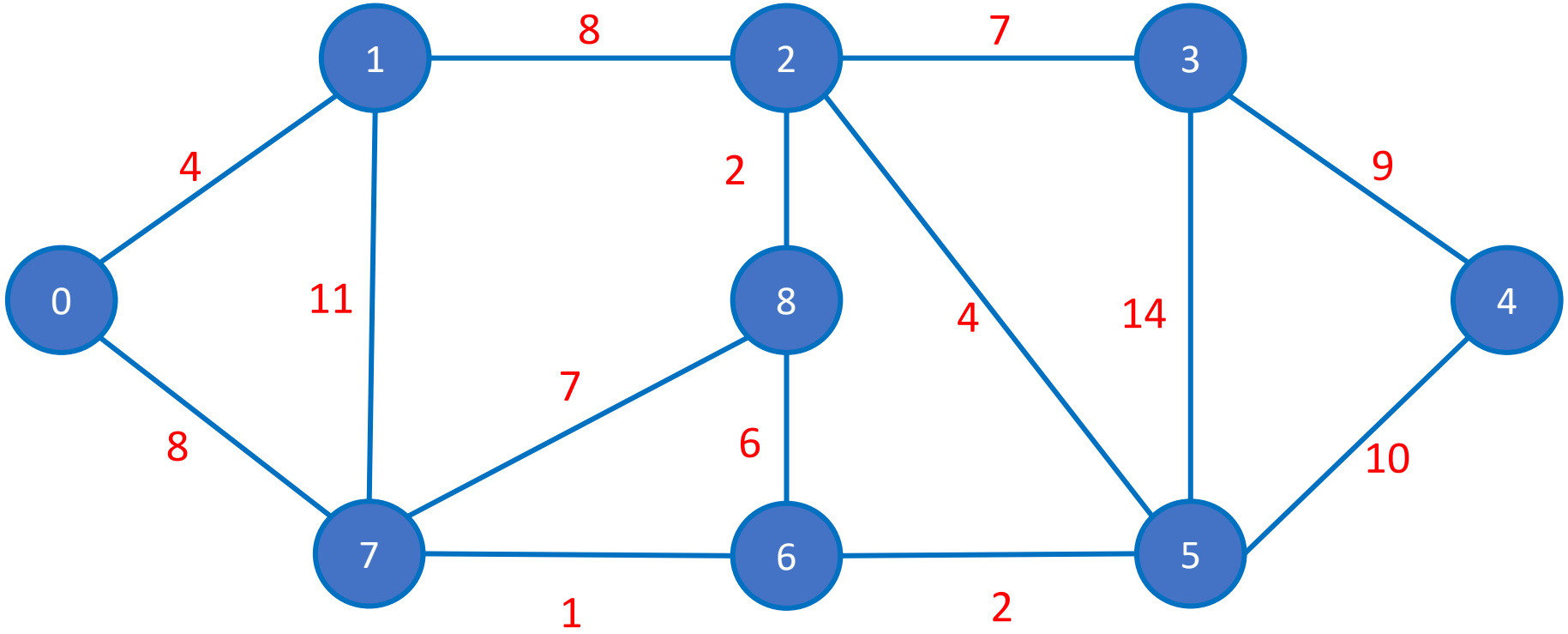
Prim's Minimum Spanning Tree





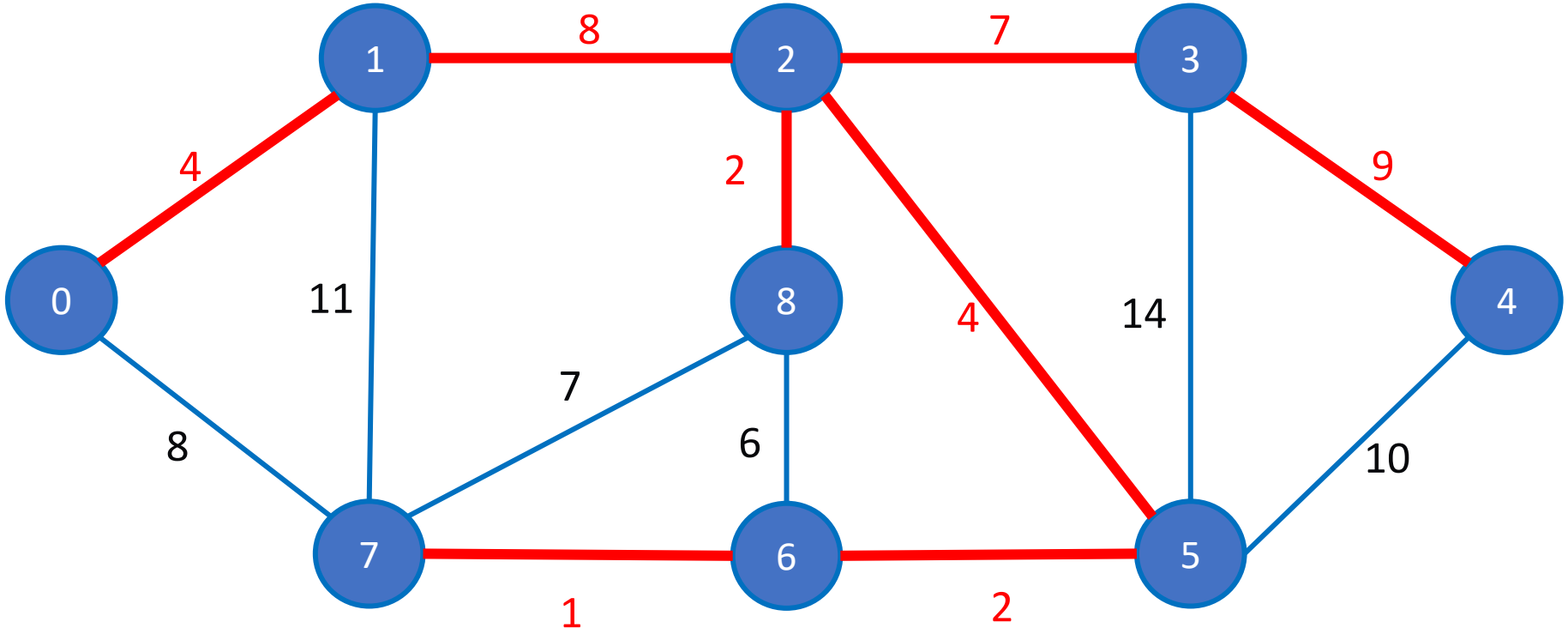
Prim's Minimum Spanning Tree

- Demonstrate the Prim algorithm to find the MST from the following graph



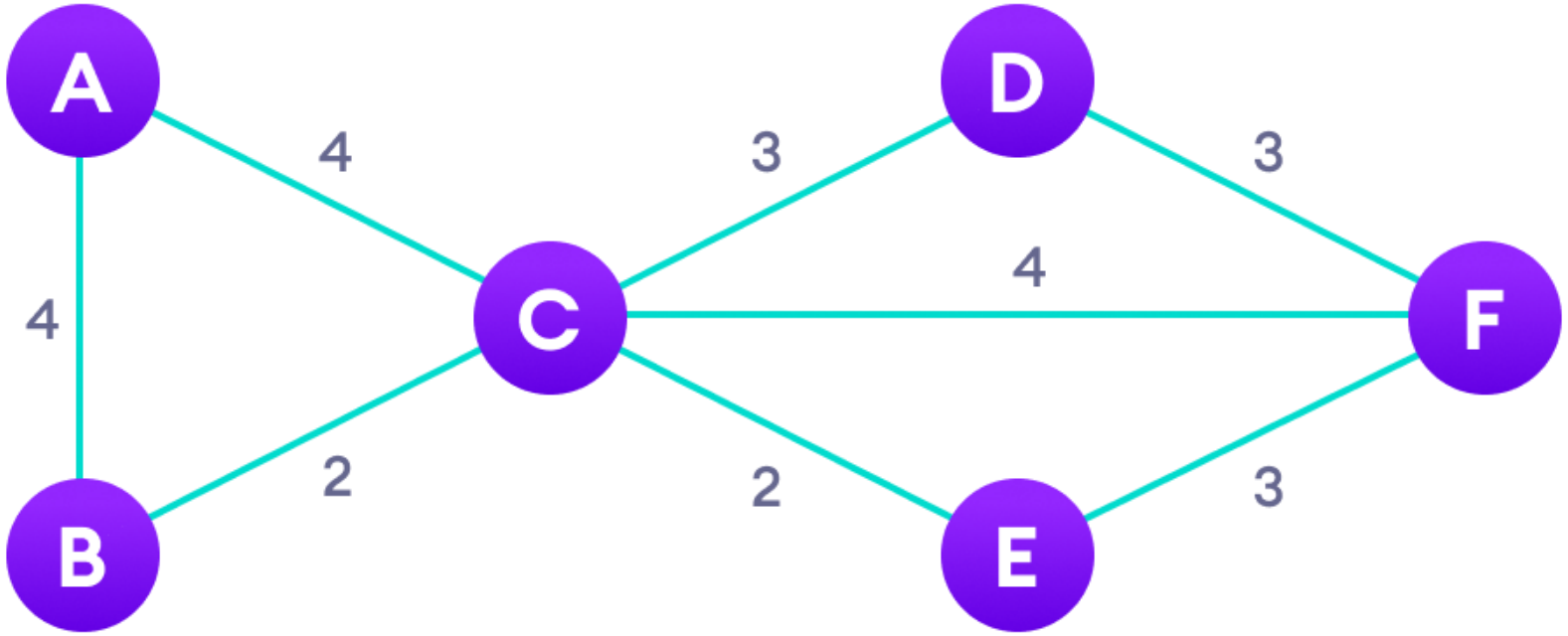
Prim's Minimum Spanning Tree

- Demonstrate the Prim algorithm to find the MST from the following graph



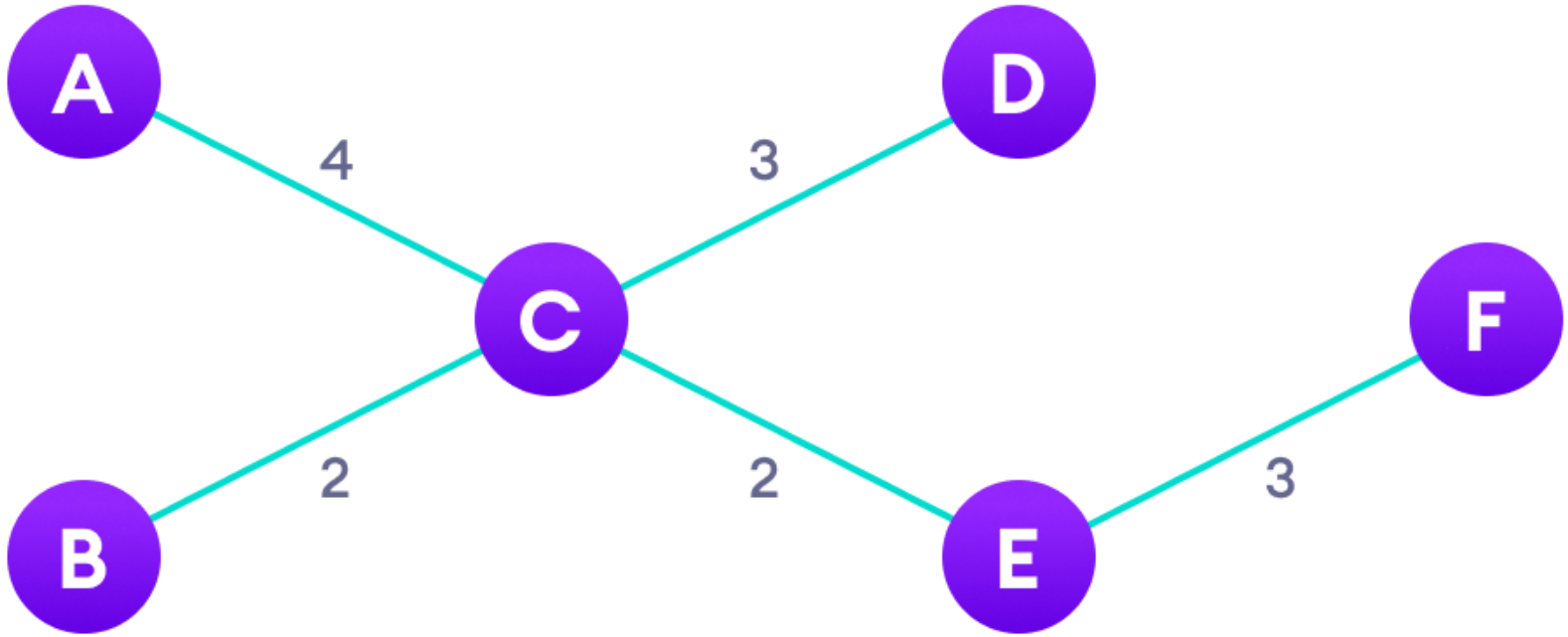
Prim's Minimum Spanning Tree

- Demonstrate the Prim algorithm to find the MST from the following graph



Prim's Minimum Spanning Tree

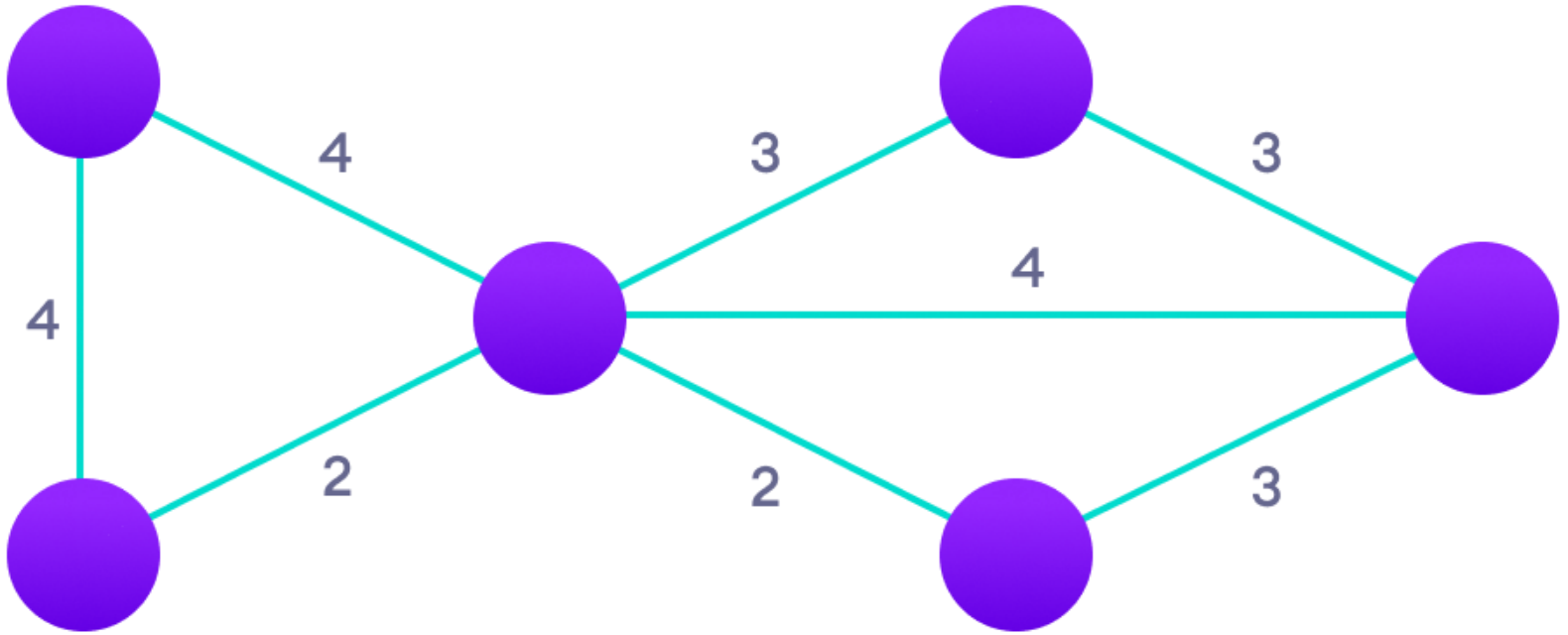
- Demonstrate the Prim algorithm to find the MST from the following graph



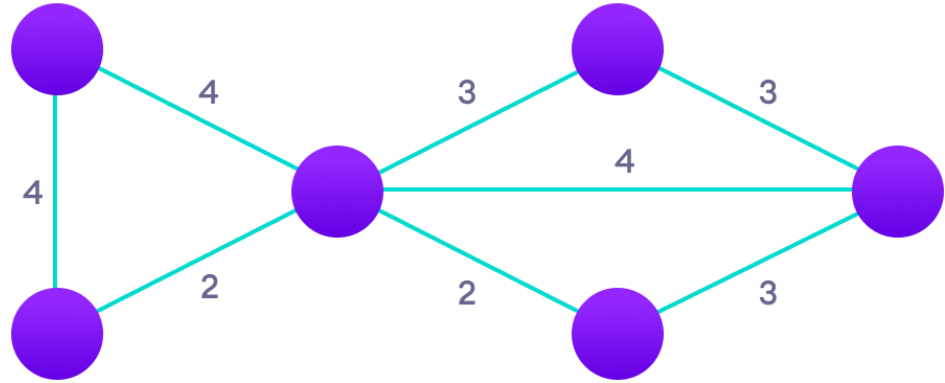
- The steps for implementing Kruskal's algorithm are as follows:
  - Sort all the edges from low weight to high
  - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
  - Keep adding edges until we reach all vertices

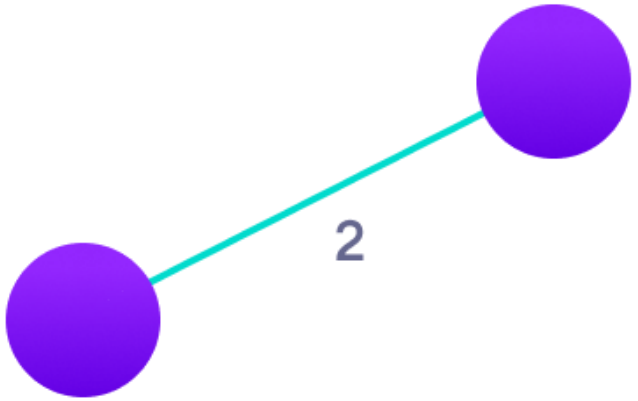
Kruskal's Minimum Spanning Tree

```
KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V:

    MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order
                          by weight(u, v):

    if FIND-SET(u) ≠ FIND-SET(v):

        A = A ∪ {(u, v)}

    UNION(u, v)

return A
```

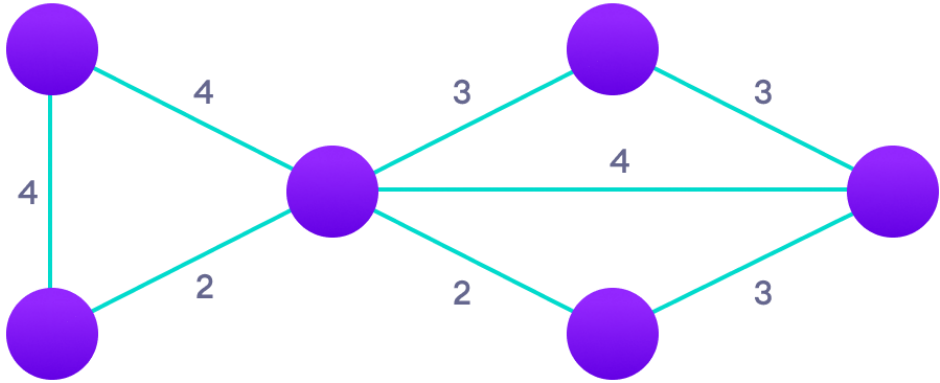


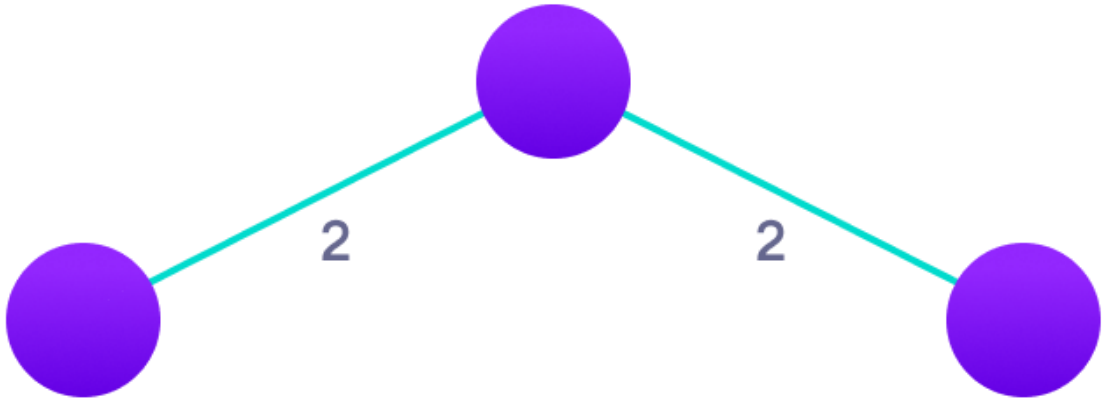
Step: 1



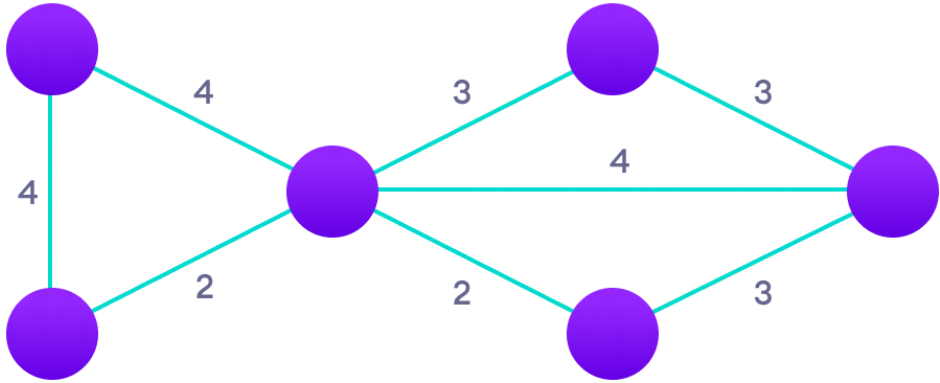


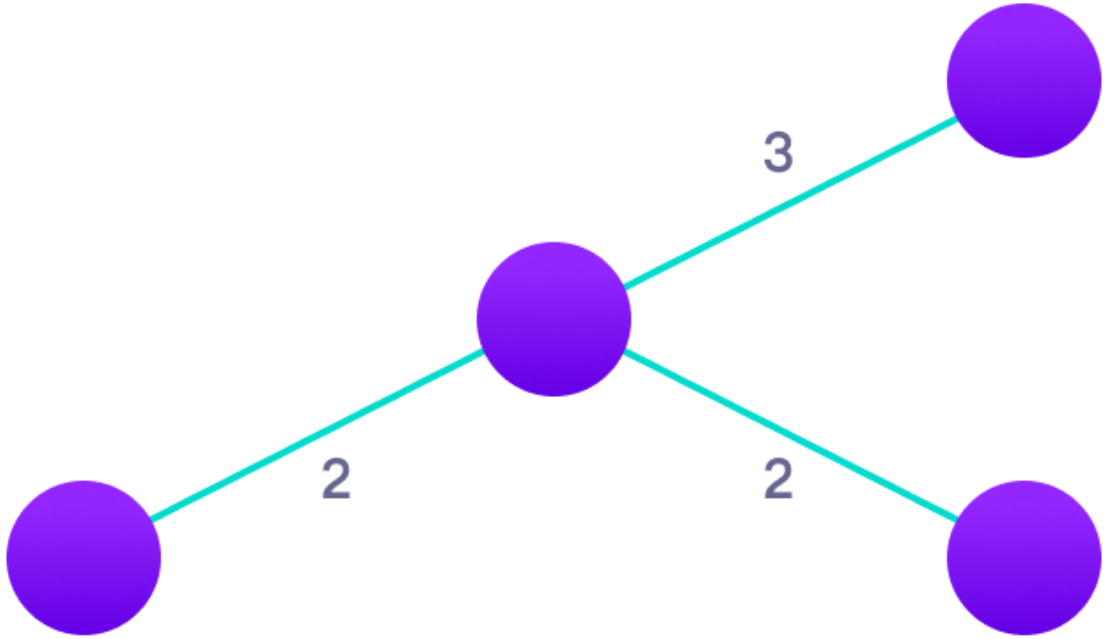
Step: 2



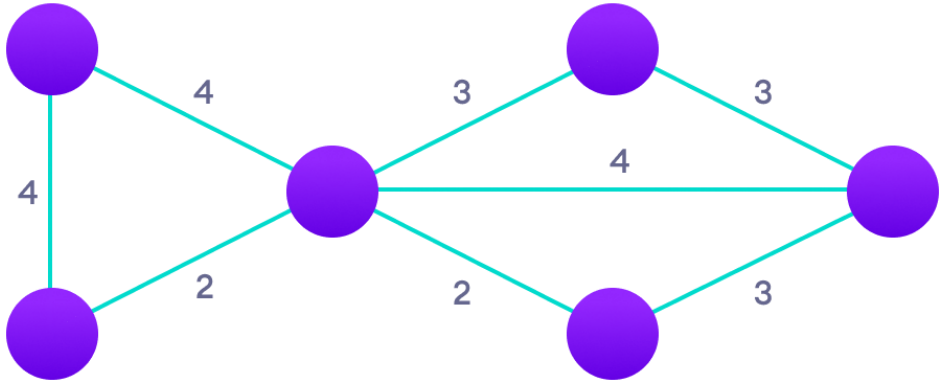


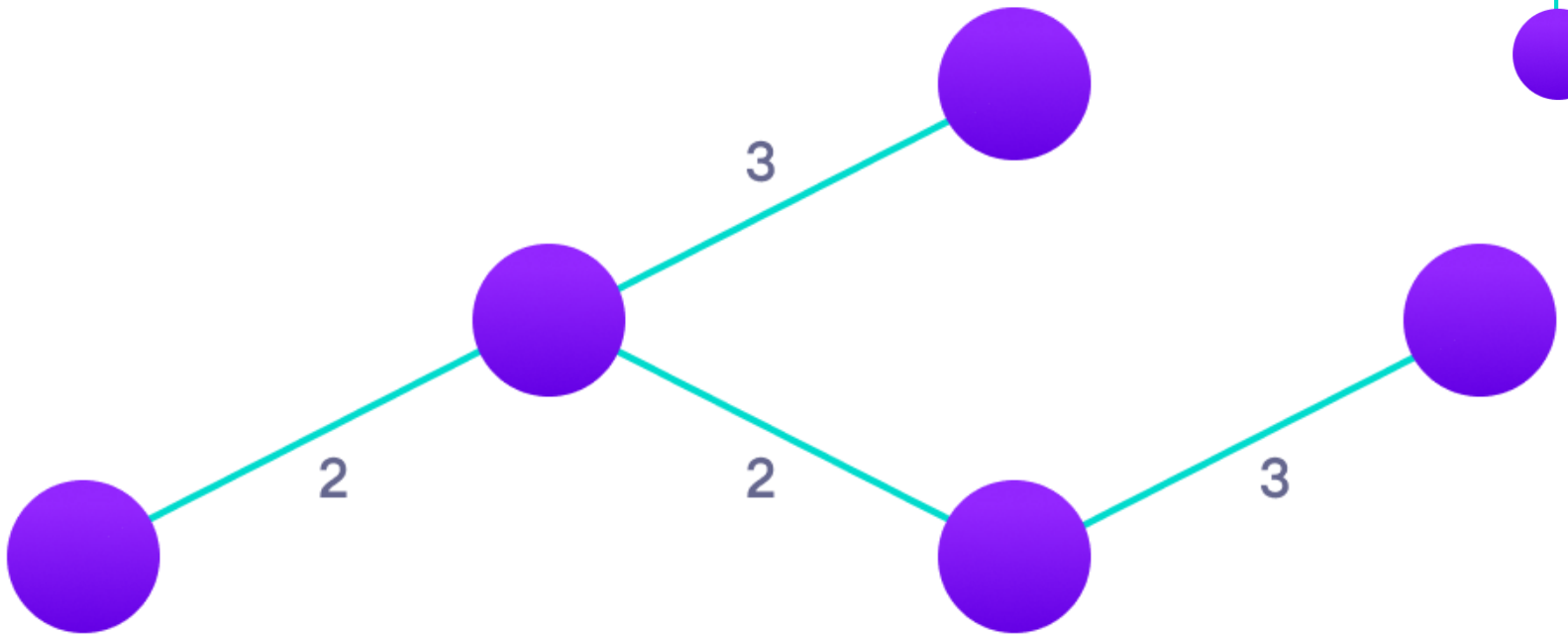
Step: 3



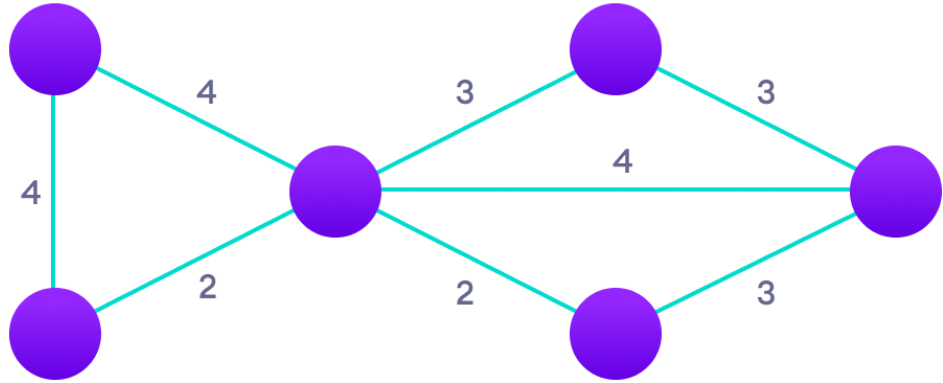


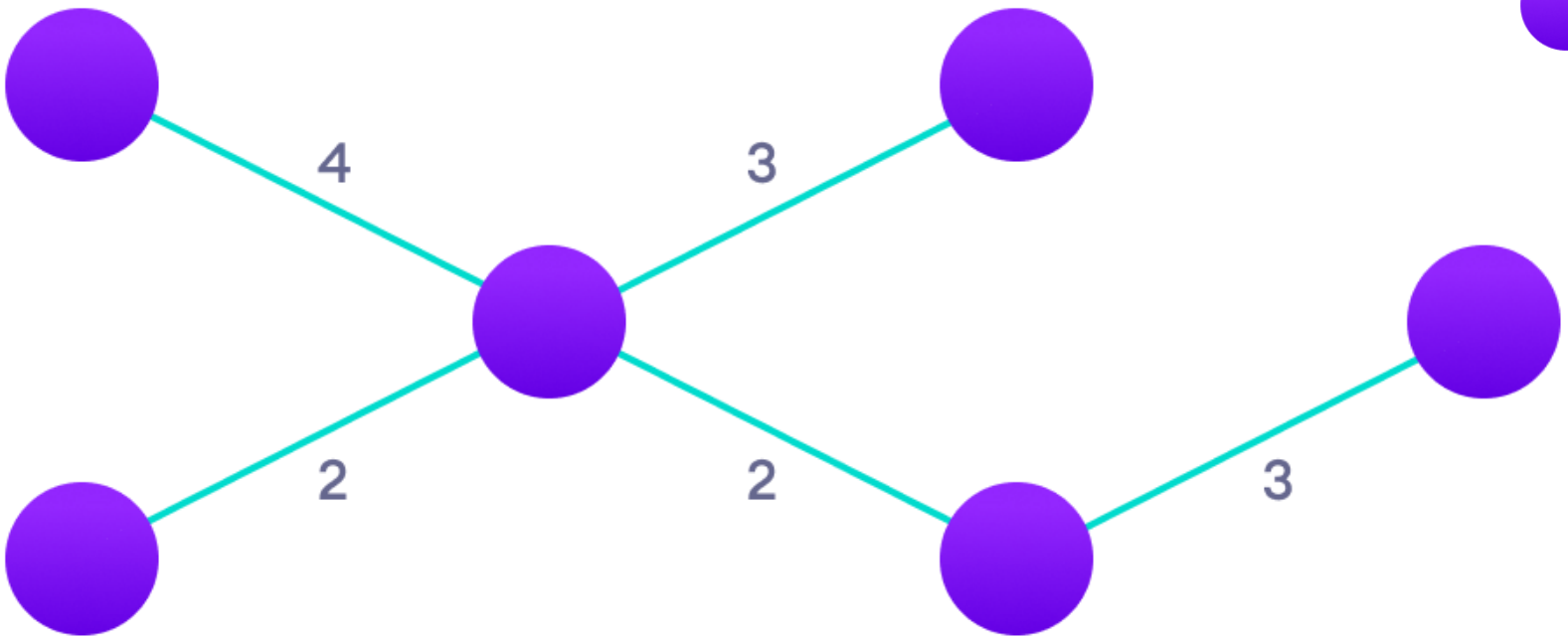
Step: 4





Step: 5



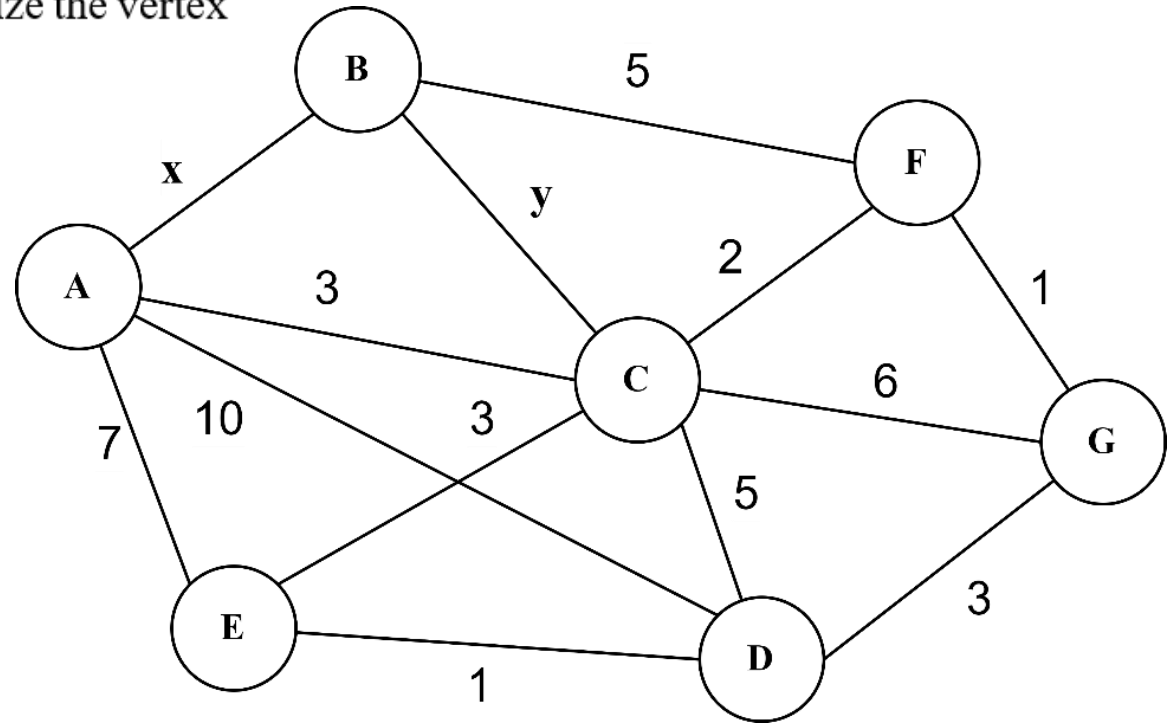


Step: 6

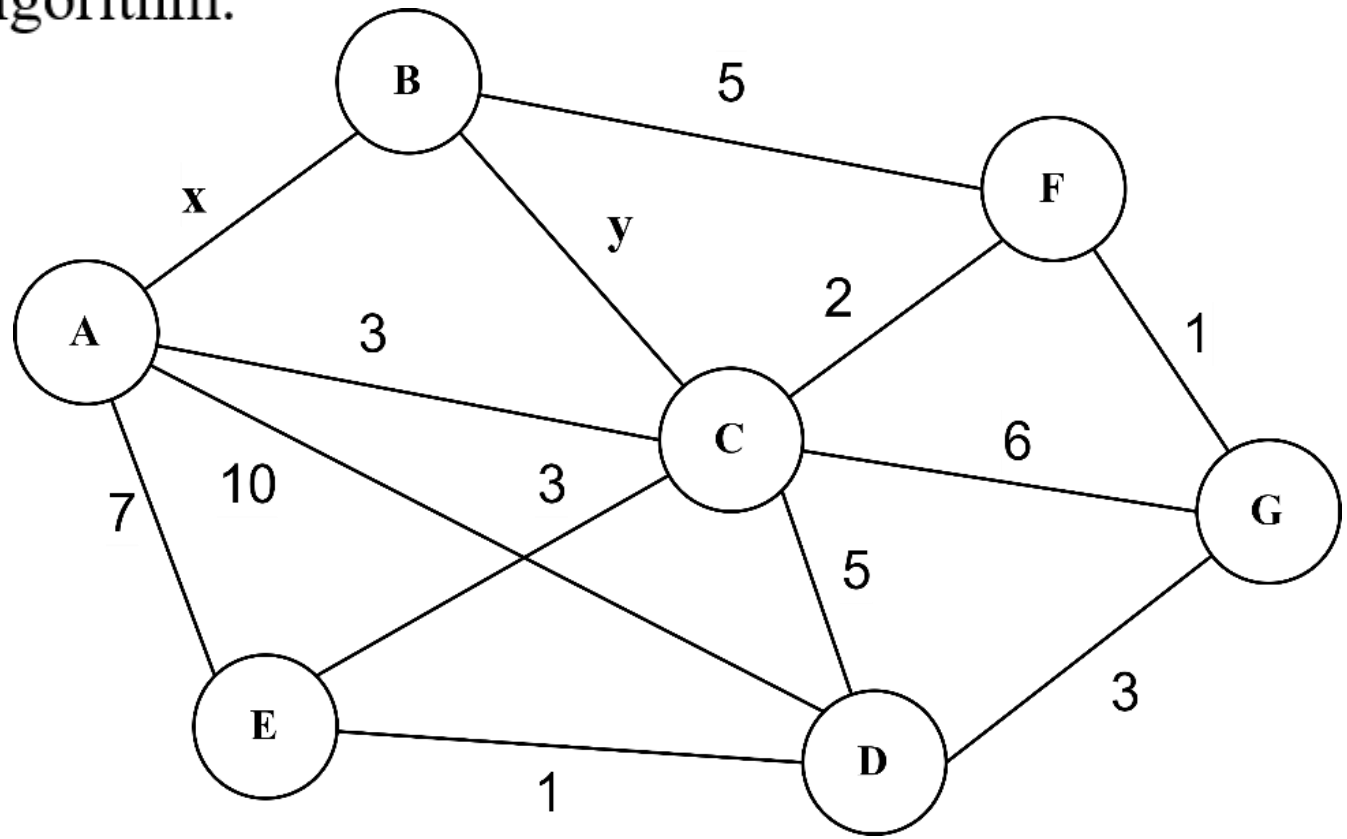
a. *(5 pts)* Vertex **S** is determined by the following table with the value of last 3 digits of your StudentID (SID) mod 4. For example, if your StudentID is 22127999 (last 3 digits is 999 and 999 mod 4 = 3) then vertex **E** is chosen as the source vertex **S**.

Last 3 digits of SID mod 4	0	1	2	3
S	C	A	F	E

Determine the order of vertices by BFS traversals on the graph, starting from vertex S. In the case of multiple vertices that can be reached simultaneously, prioritize the vertex that appears earlier in the alphabet.



d. *(5 pts)* Find the maximum values of two positive integers, **x** and **y**, within the graph **G** such that they facilitate the construction of the Minimum Spanning Tree. This tree initiates from vertex **C** and includes the edges (A, B) and (B, C). Describe your approach by employing Prim's Algorithm.



THANK YOU
for YOUR ATTENTION