Depth First Search

- ❑ **Cost Analysis**

Application of DFS

- ❑ **Topological Sort**
- ❑ **Finding Connected Component**
- ❑ **Connectivity of a Graph**

Depth First Search

A method to traverse all the vertices of a given graph(directed or undirected)

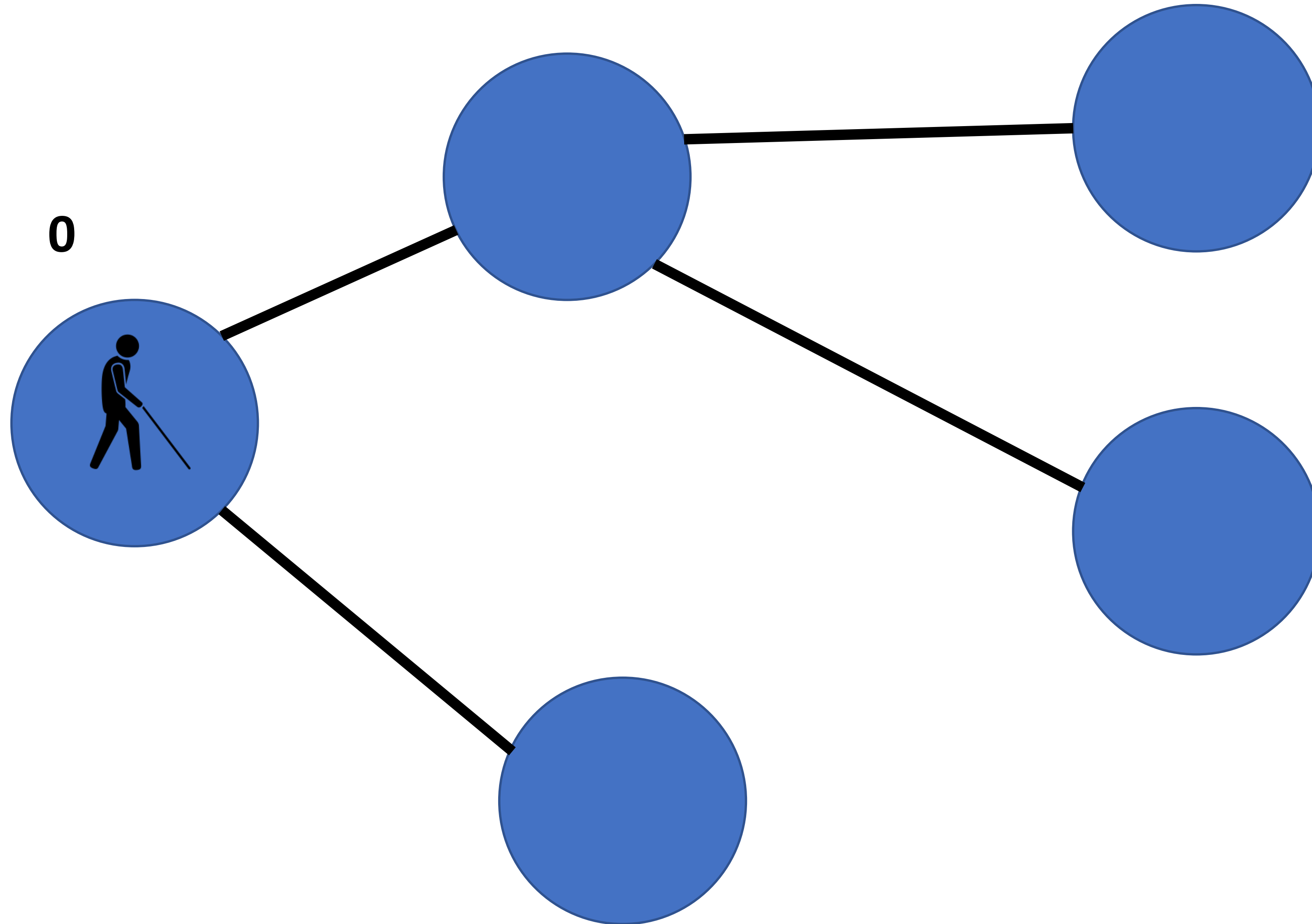
Logic:

Goes deeper and deeper in one direction until it hits the dead end

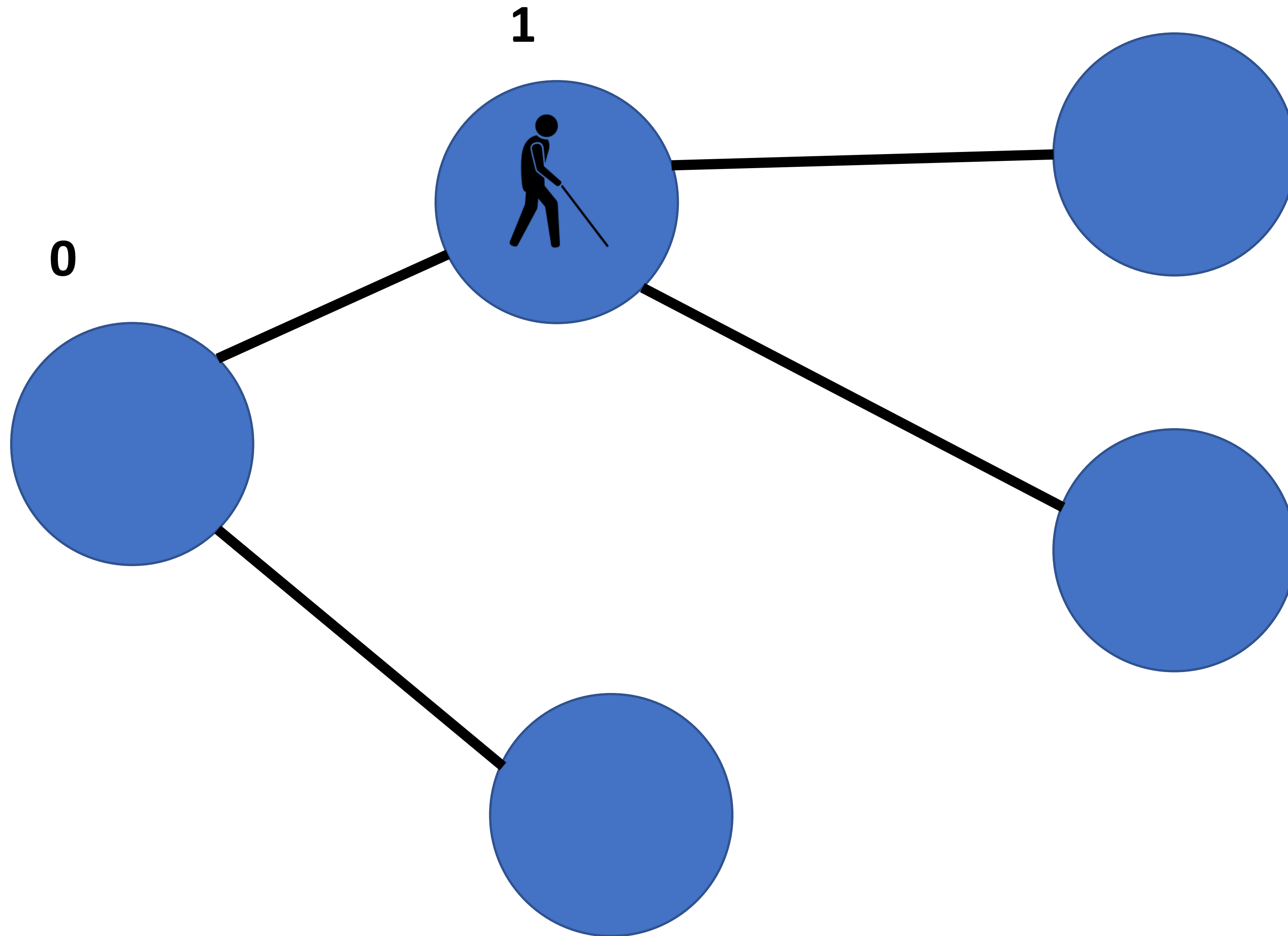
Depth First Search

Similar to the way a blind person walk

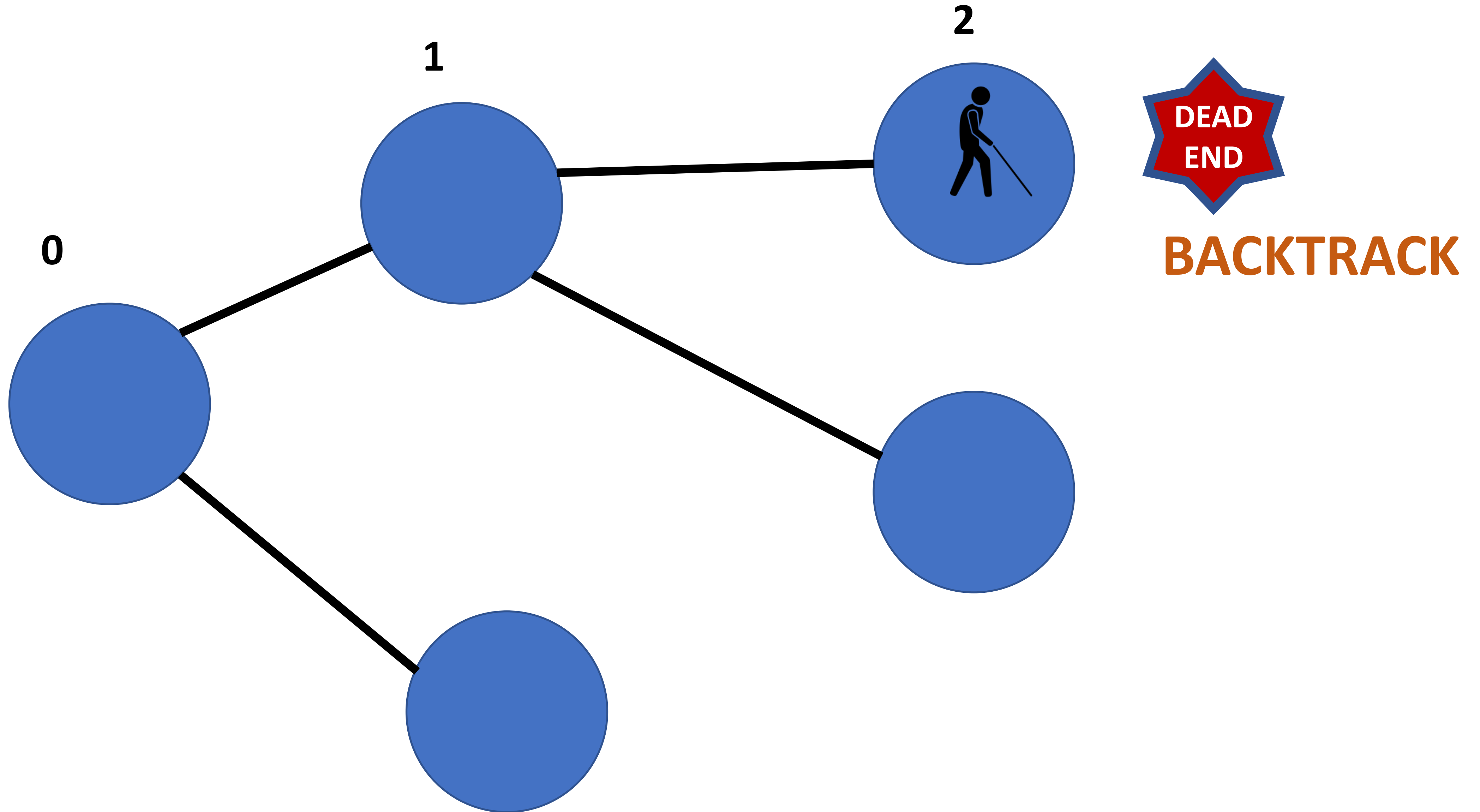
Depth First Search



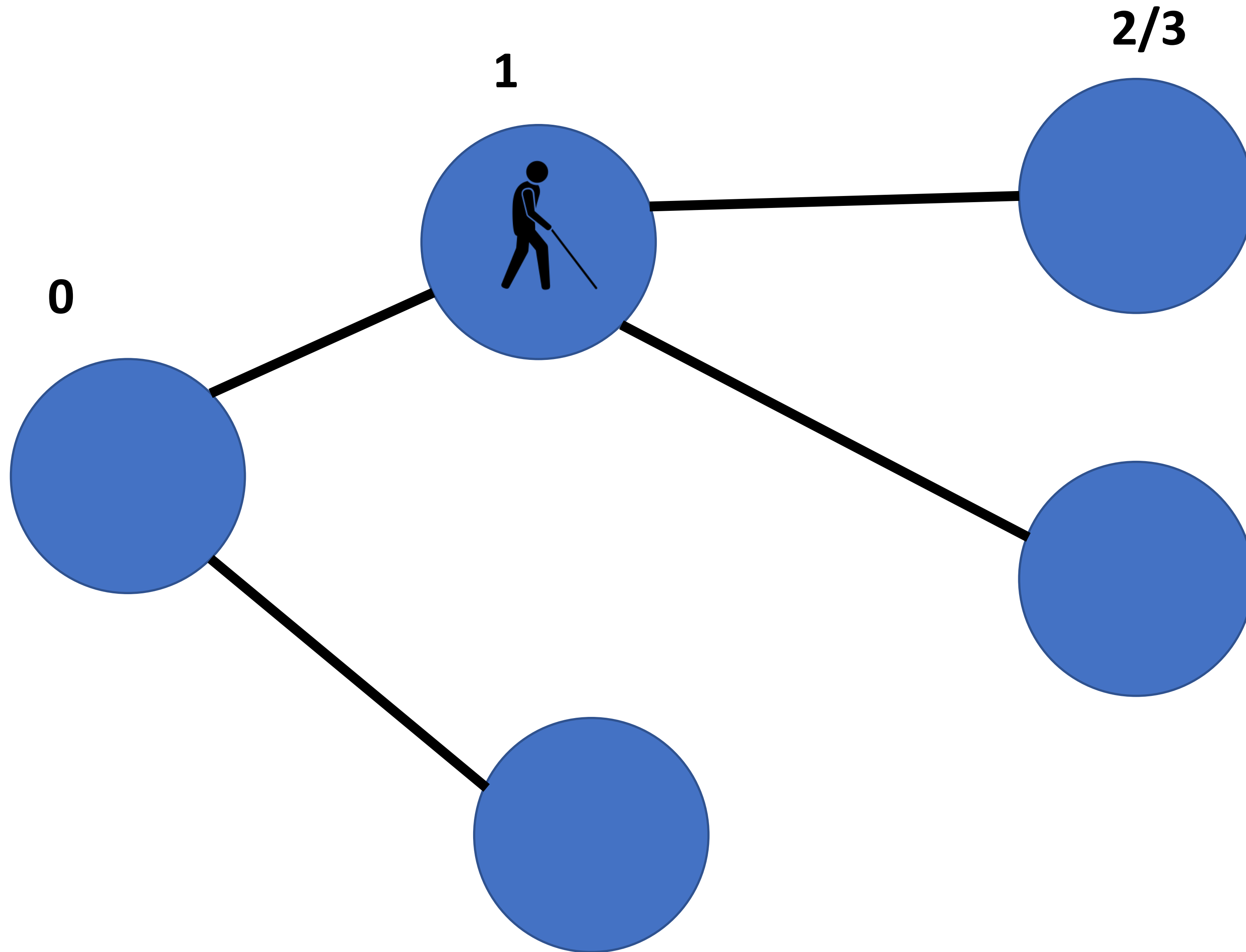
Depth First Search



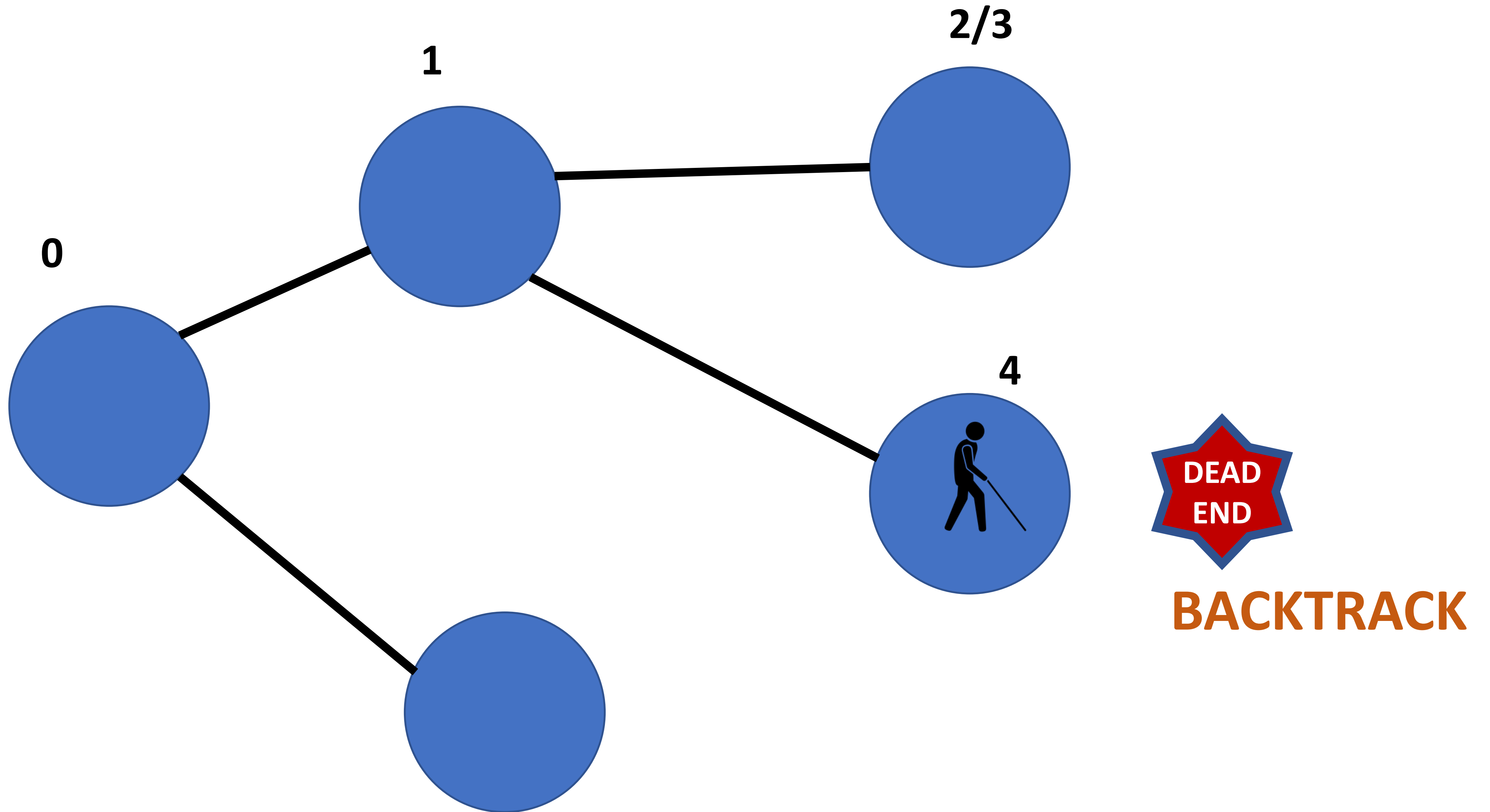
Depth First Search



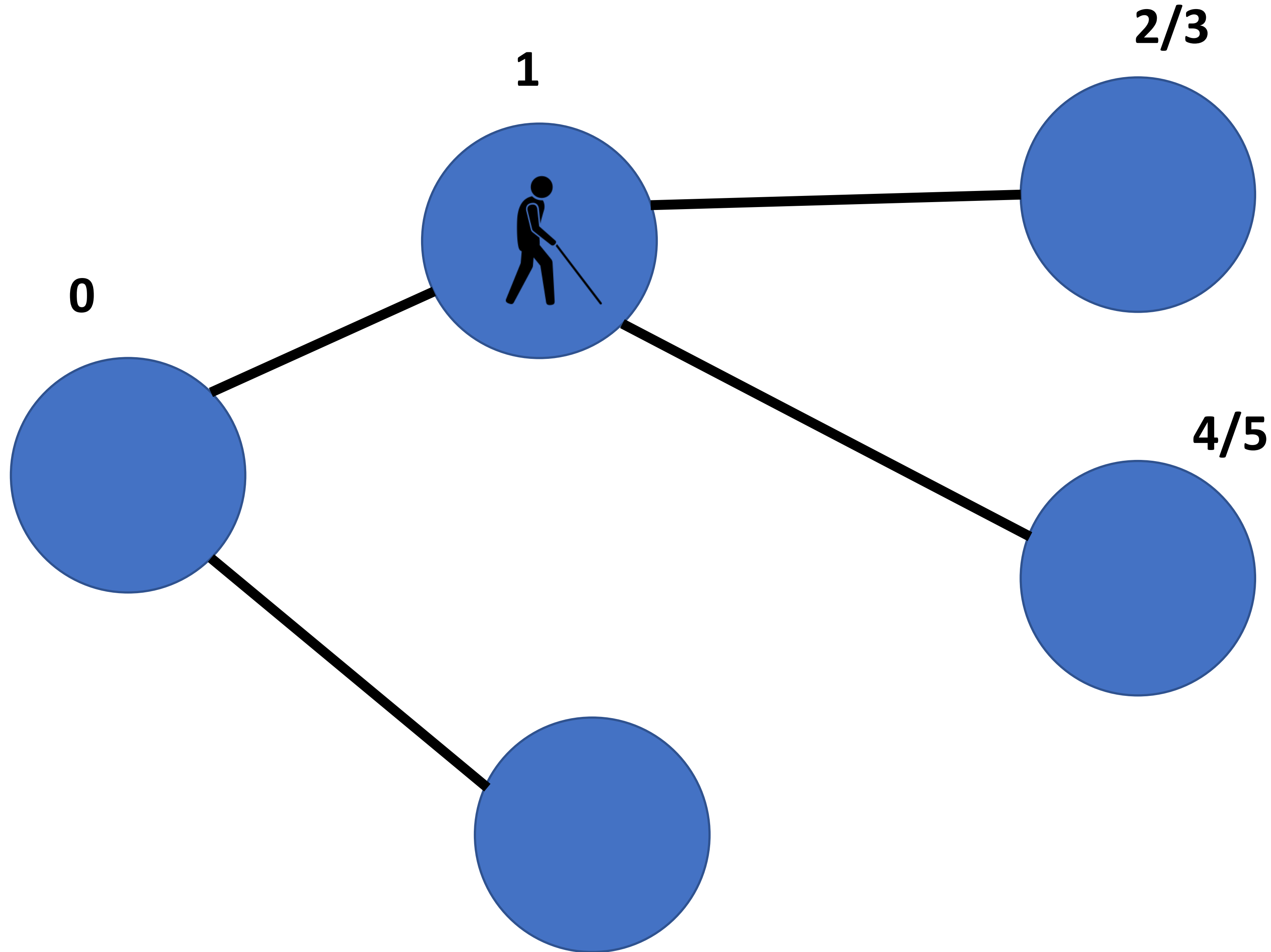
Depth First Search



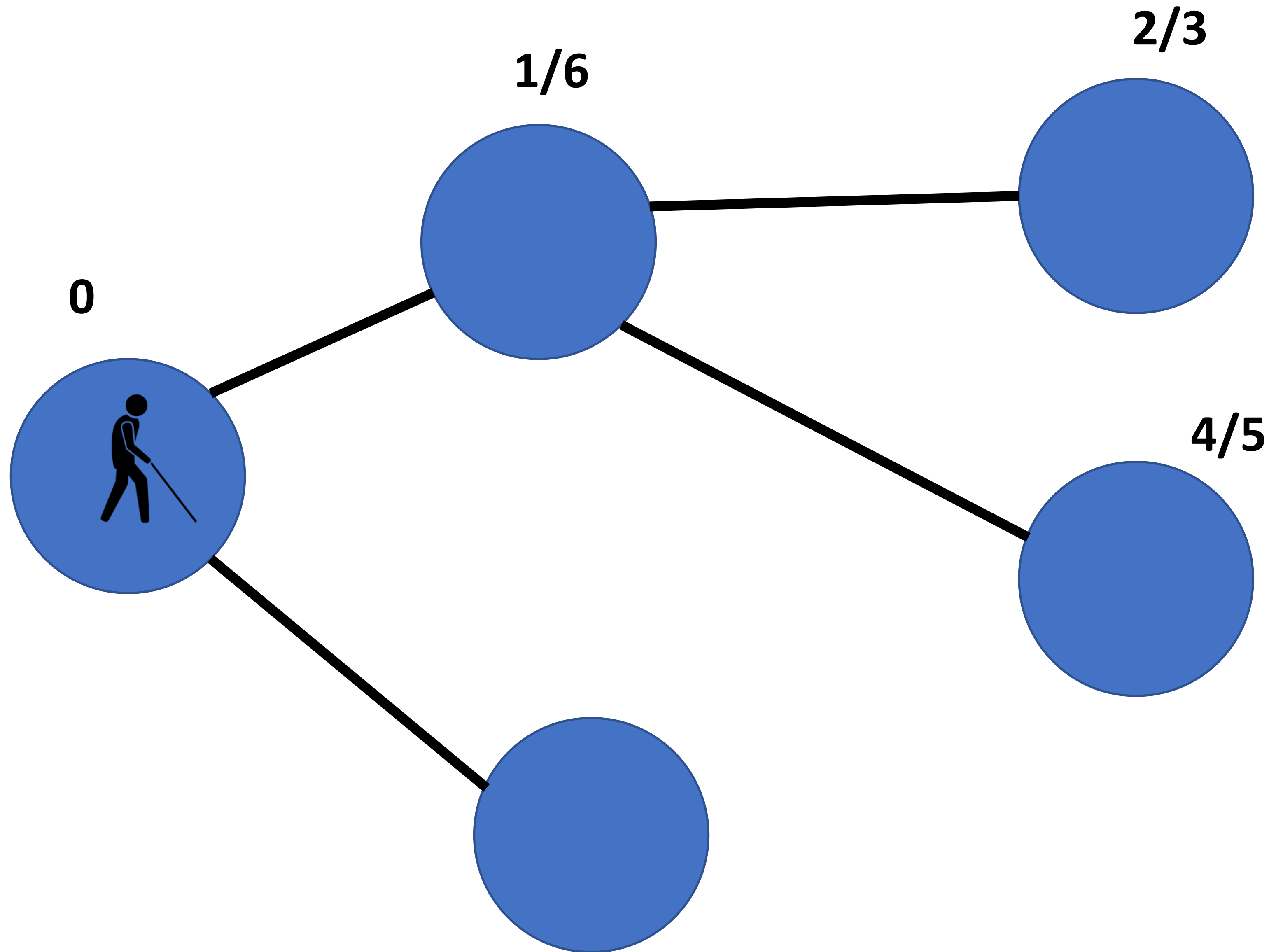
Depth First Search



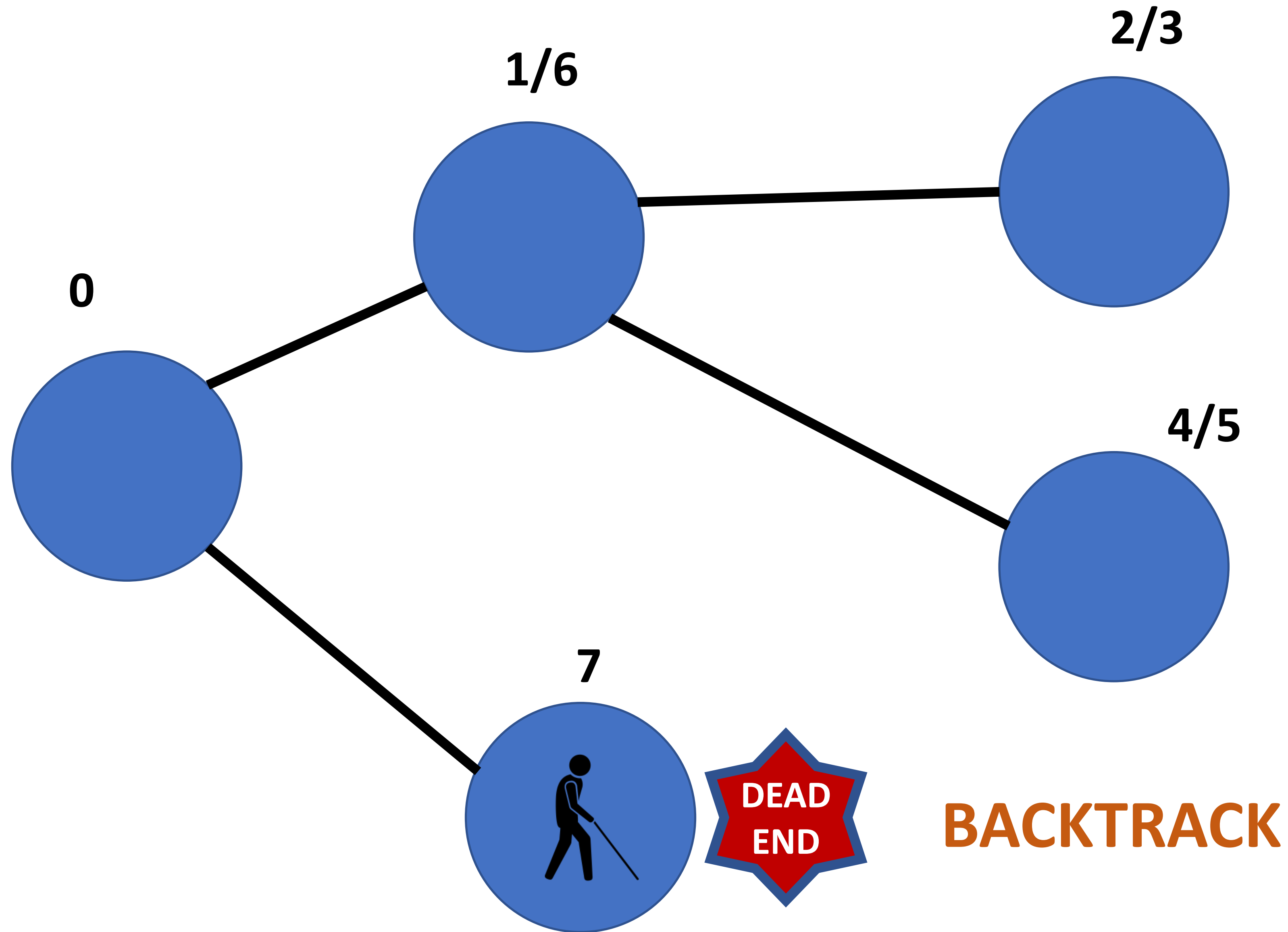
Depth First Search



Depth First Search

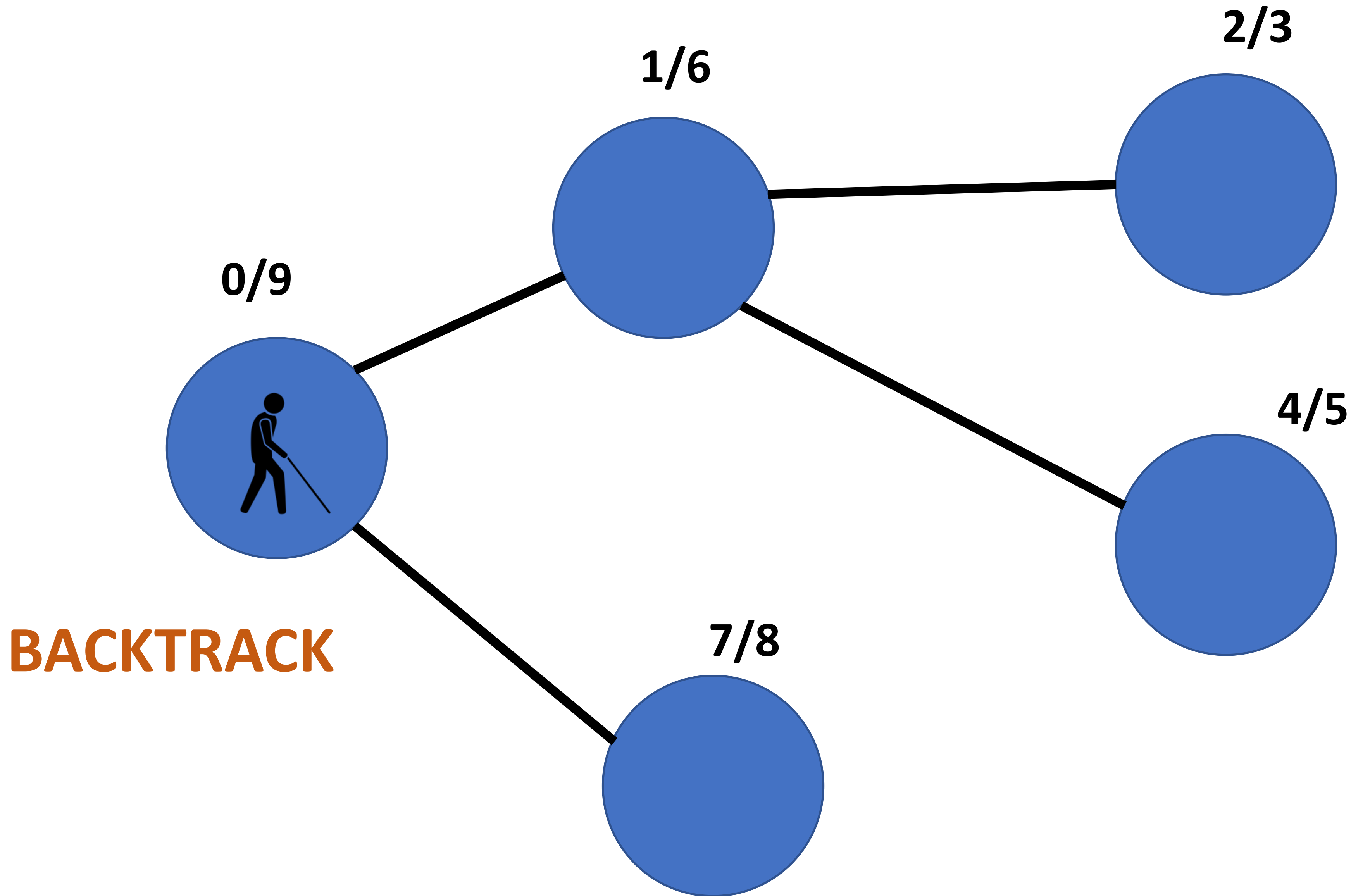


Depth First Search

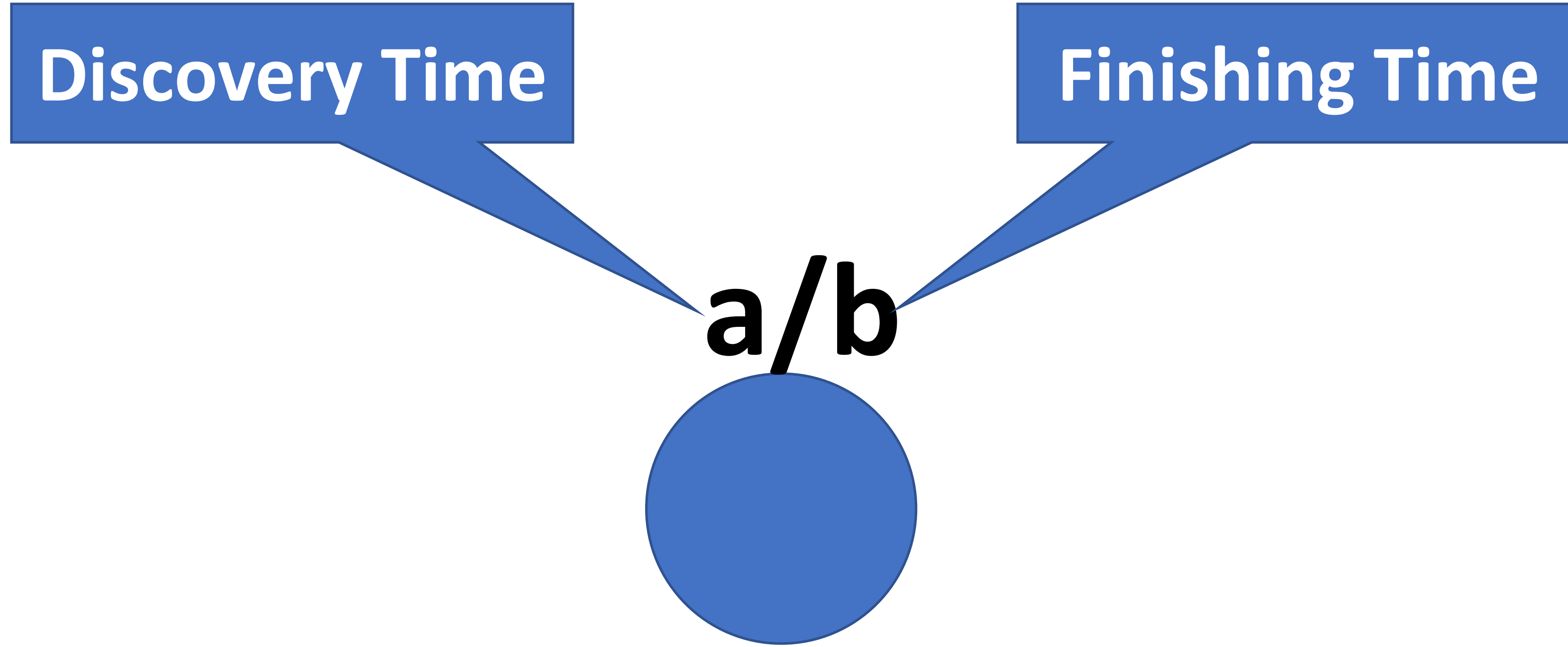


BACKTRACK

Depth First Search

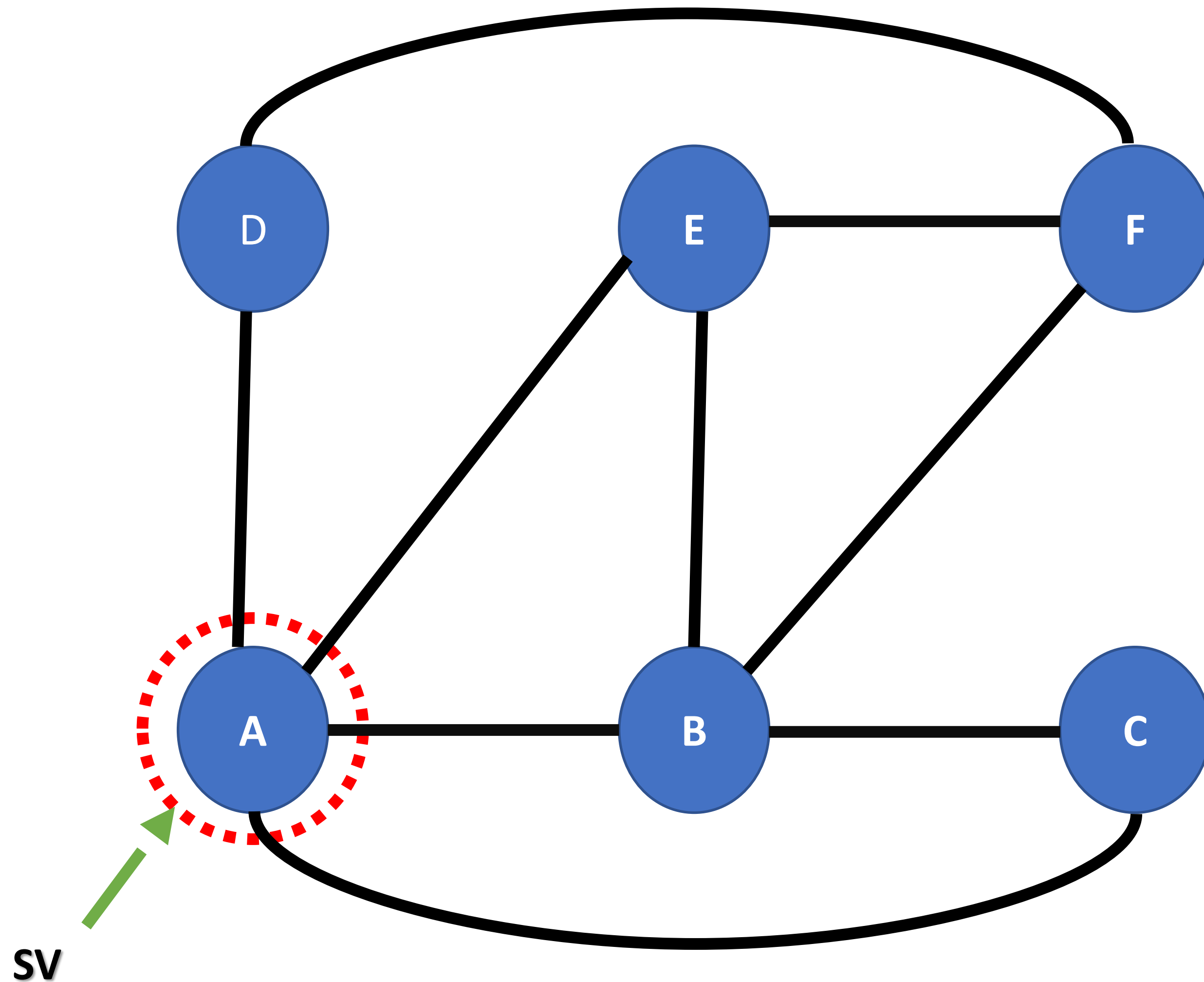


Depth First Search



Depth First Search

Let us consider the following graph

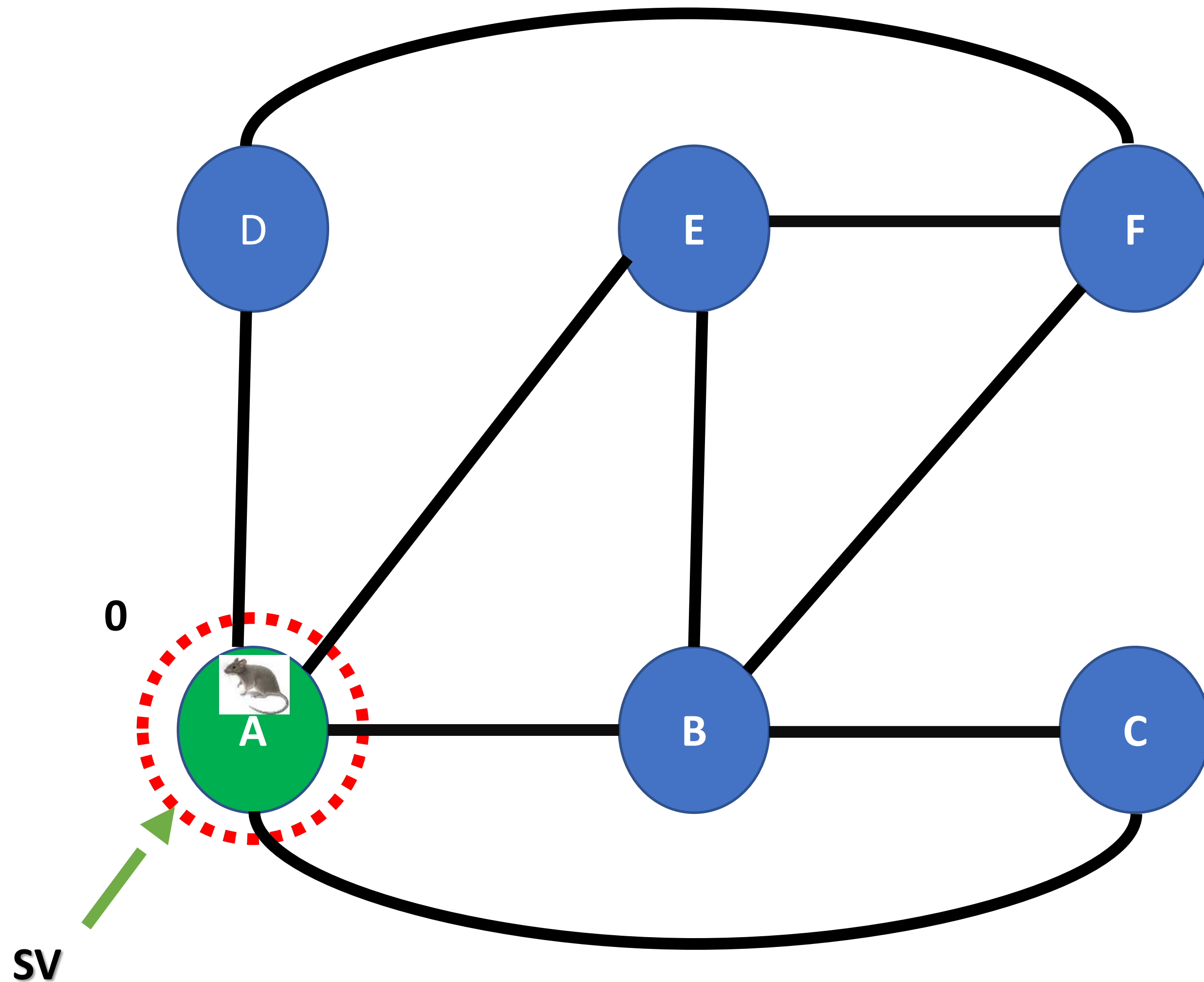


Visited

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

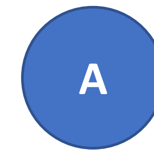
Depth First Search

Let us consider the following graph



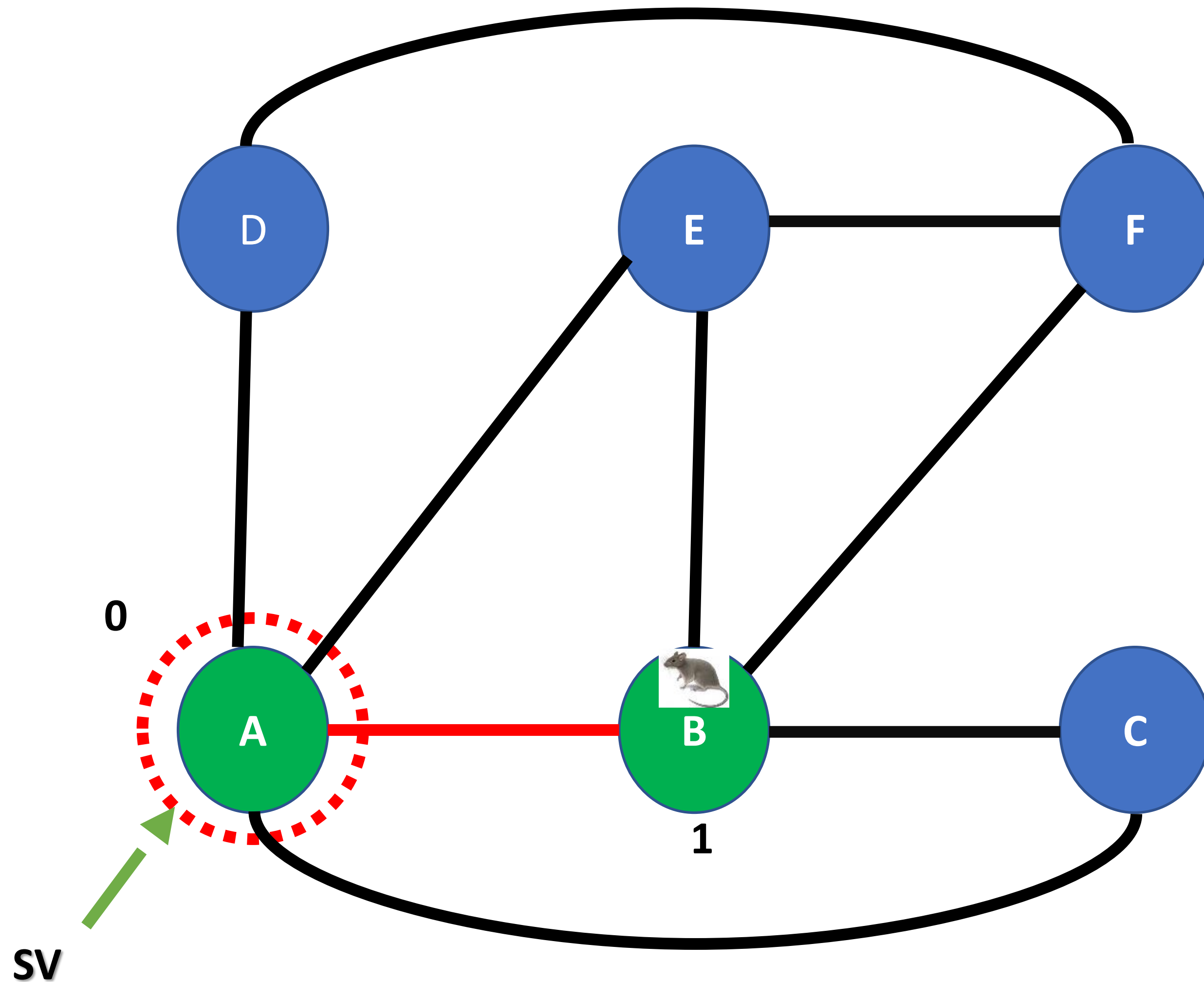
Visited

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



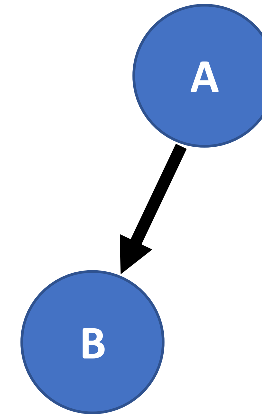
Depth First Search

Let us consider the following graph



Visited

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

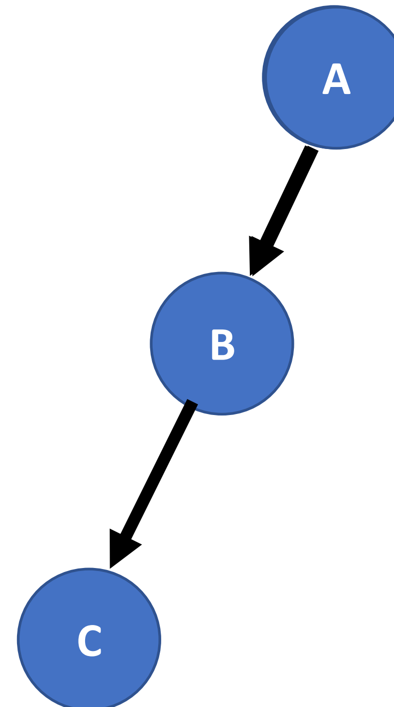


Depth First Search

Let us consider the following graph

Visited

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



Depth First Search

Let us consider the following graph

D E F

0 A B C 2/3

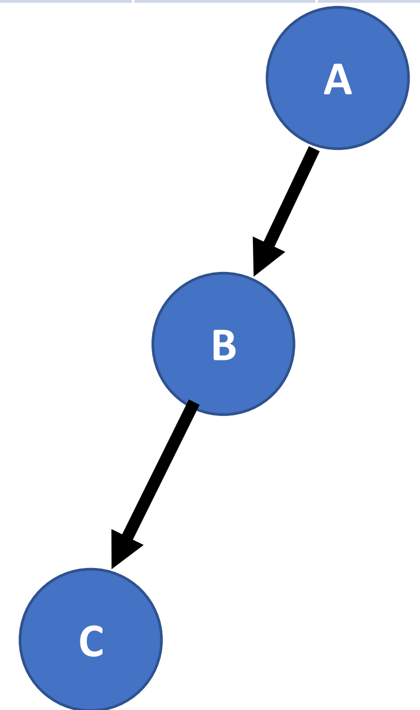
1

SV

Visited

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

BACKTRACK

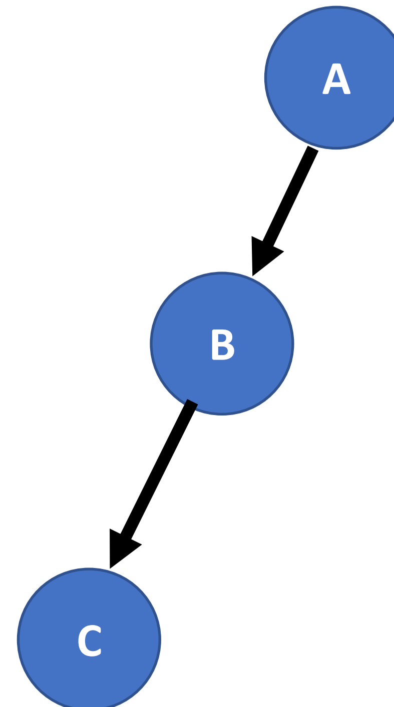


Depth First Search

Let us consider the following graph

Visited

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



Depth First Search

Let us consider the following graph

D E F 4

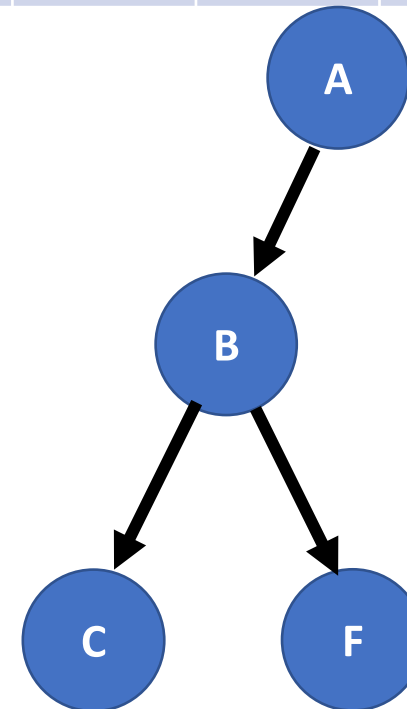
0 A B C 2/3

1

SV

Visited

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

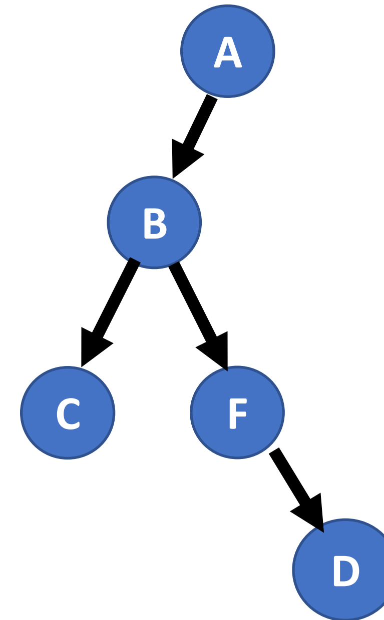


Depth First Search

BACKTRACK

Visited

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

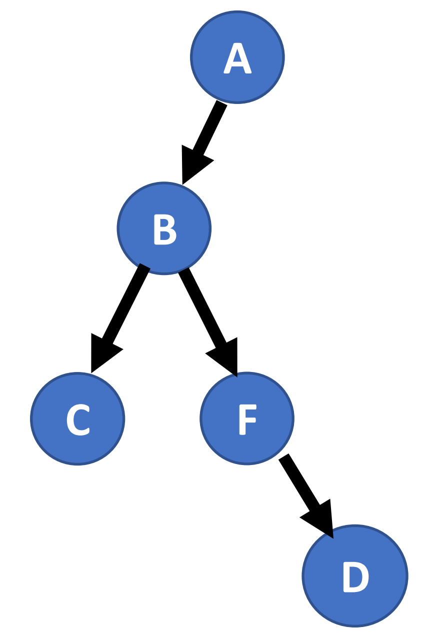


5
D
E
F
4
0
A
B
C
2/3
1
SV

Depth First Search

Visited

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



5/6 D E F 4

0 A B C 2/3

1

SV

Depth First Search

BACKTRACK

Visited

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

5/6
D

7
E

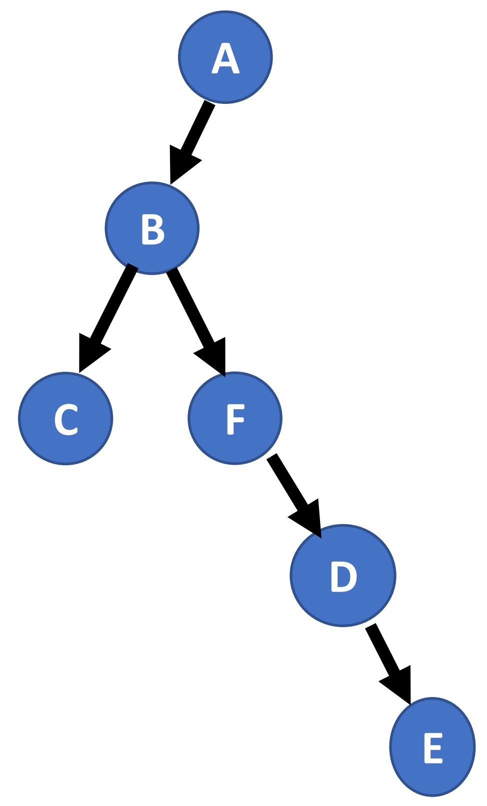
F
4

0
A

B
1

C
2/3

SV

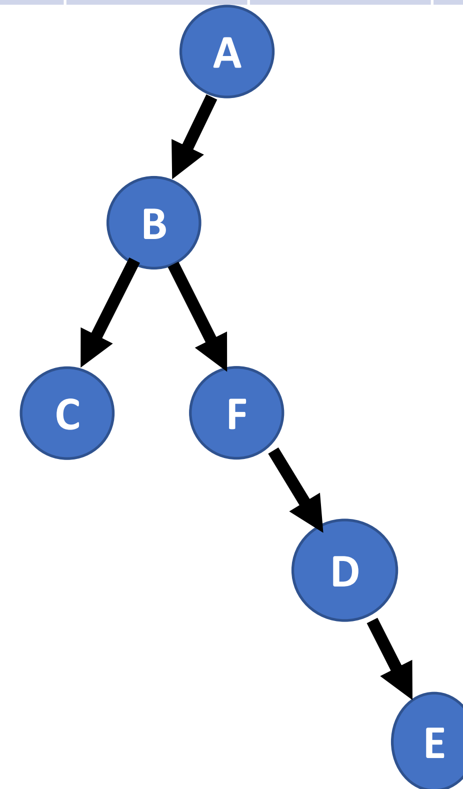


Depth First Search

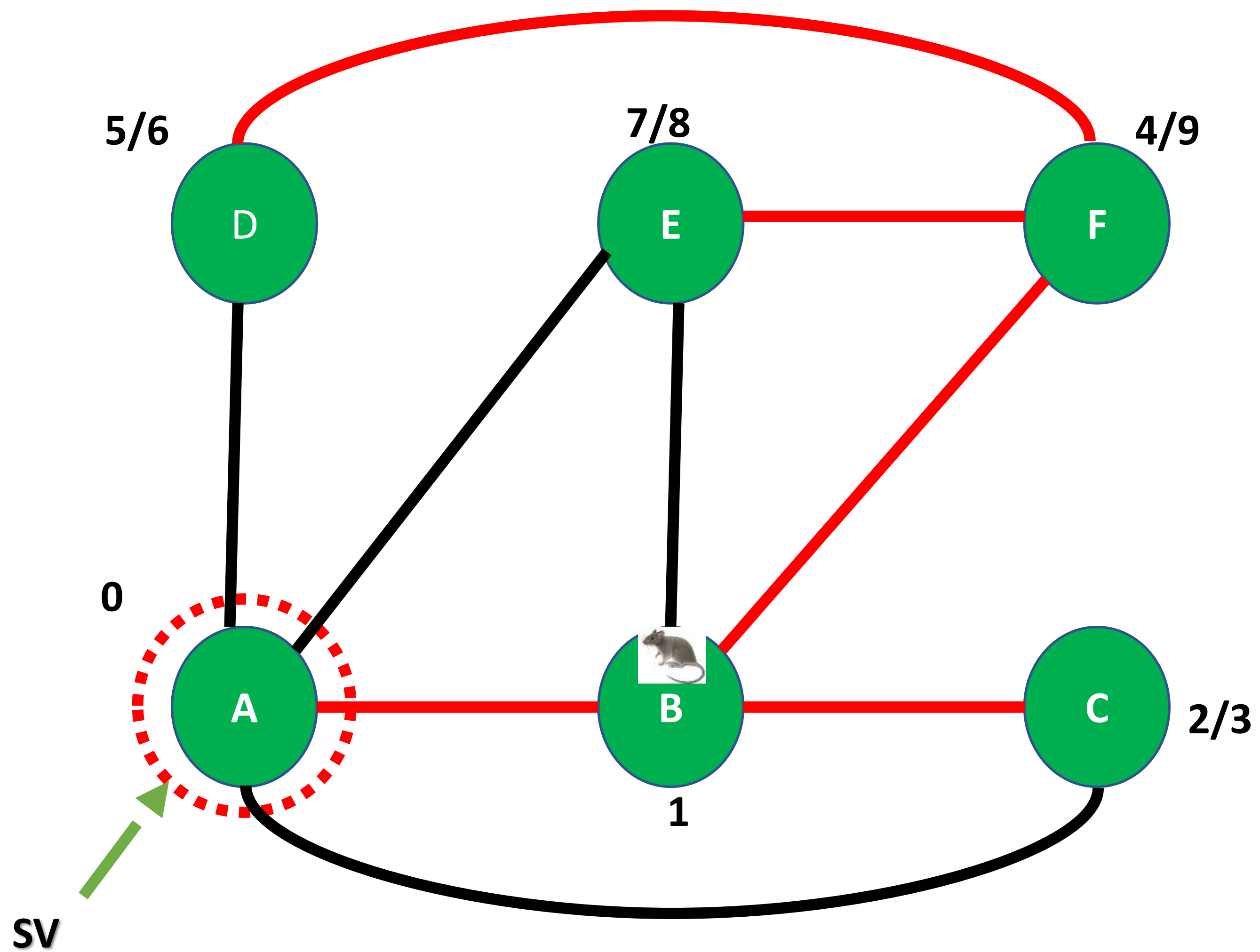
BACKTRACK

Visited

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



Depth First Search

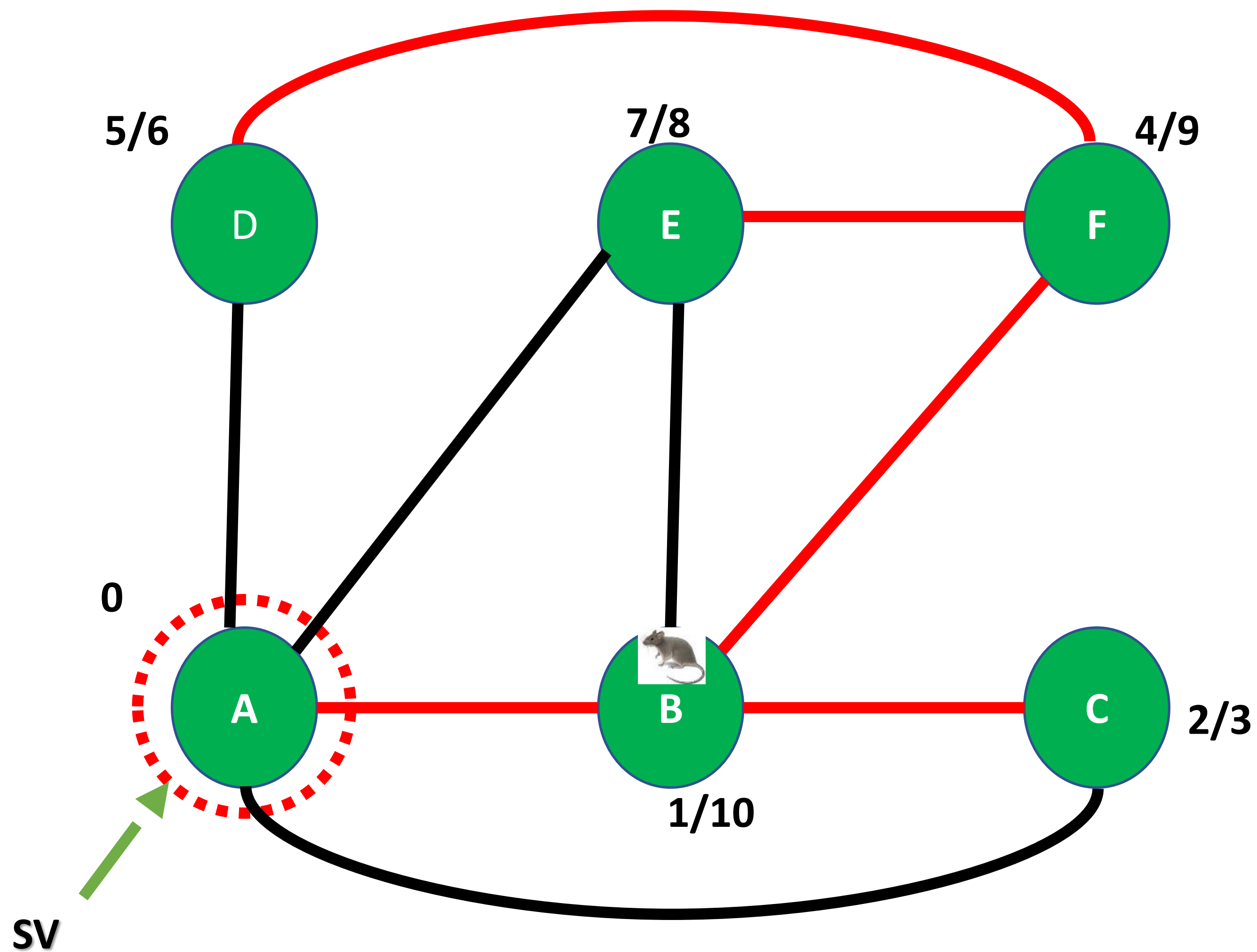


Visited

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

BACKTRACK

Depth First Search

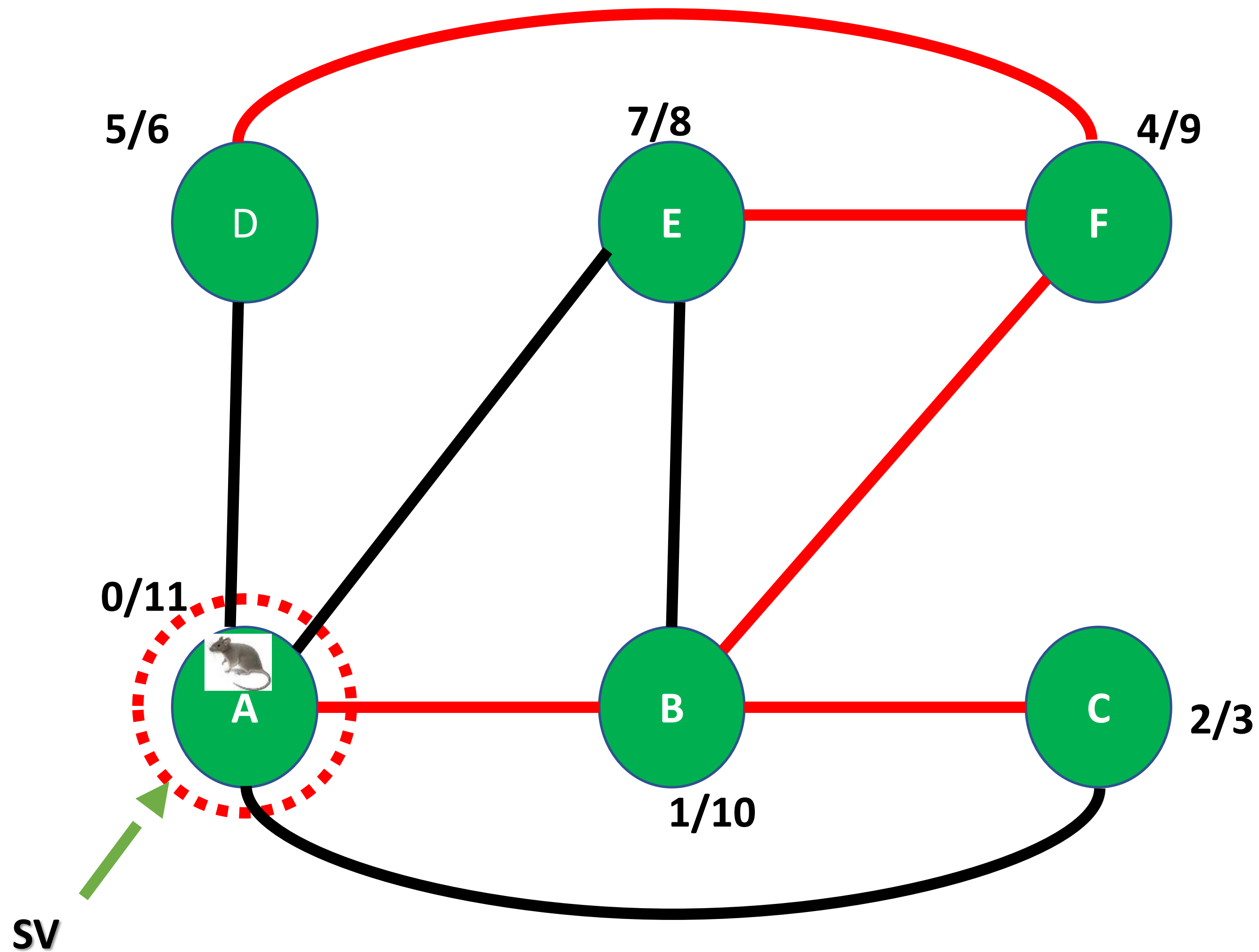


Visited

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

BACKTRACK

Depth First Search

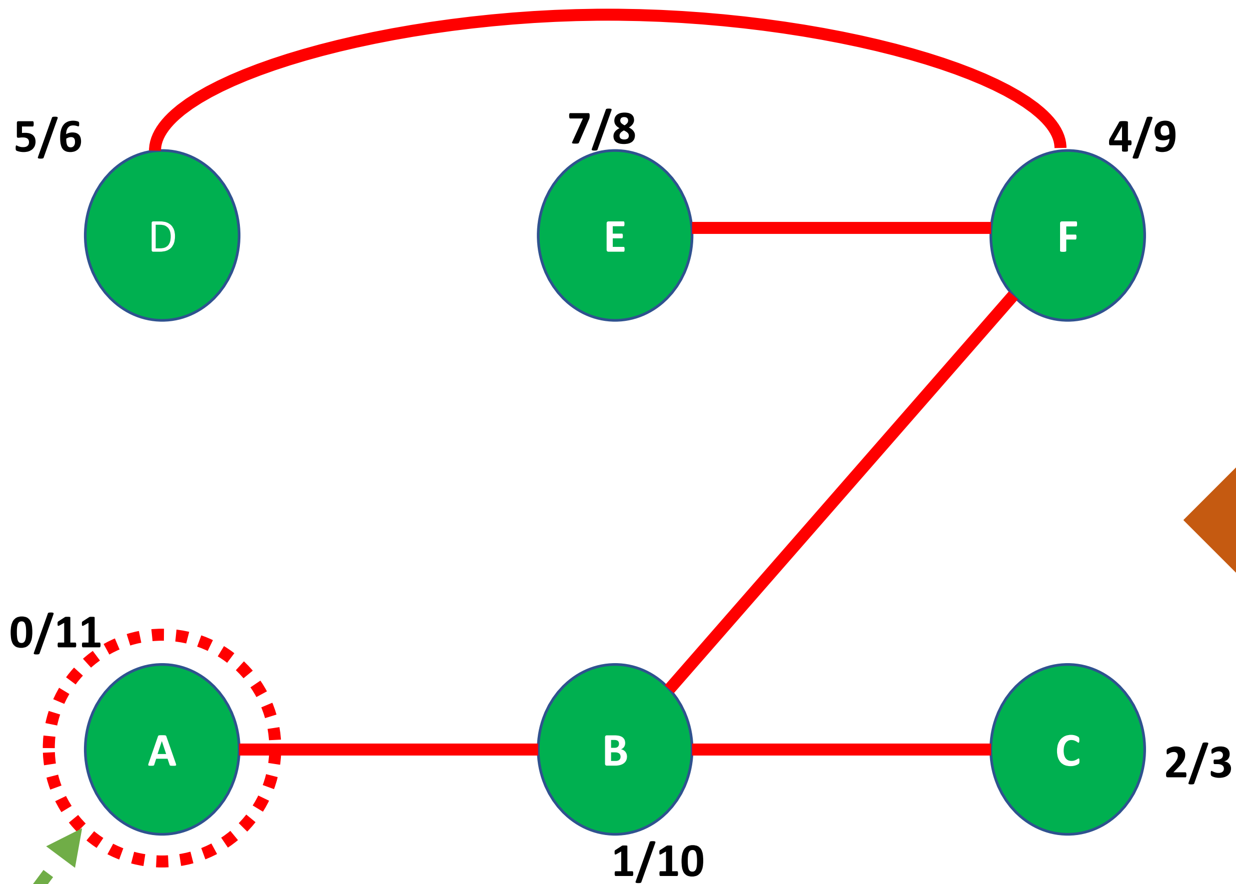


Visited

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

BACKTRACK

Depth First Search

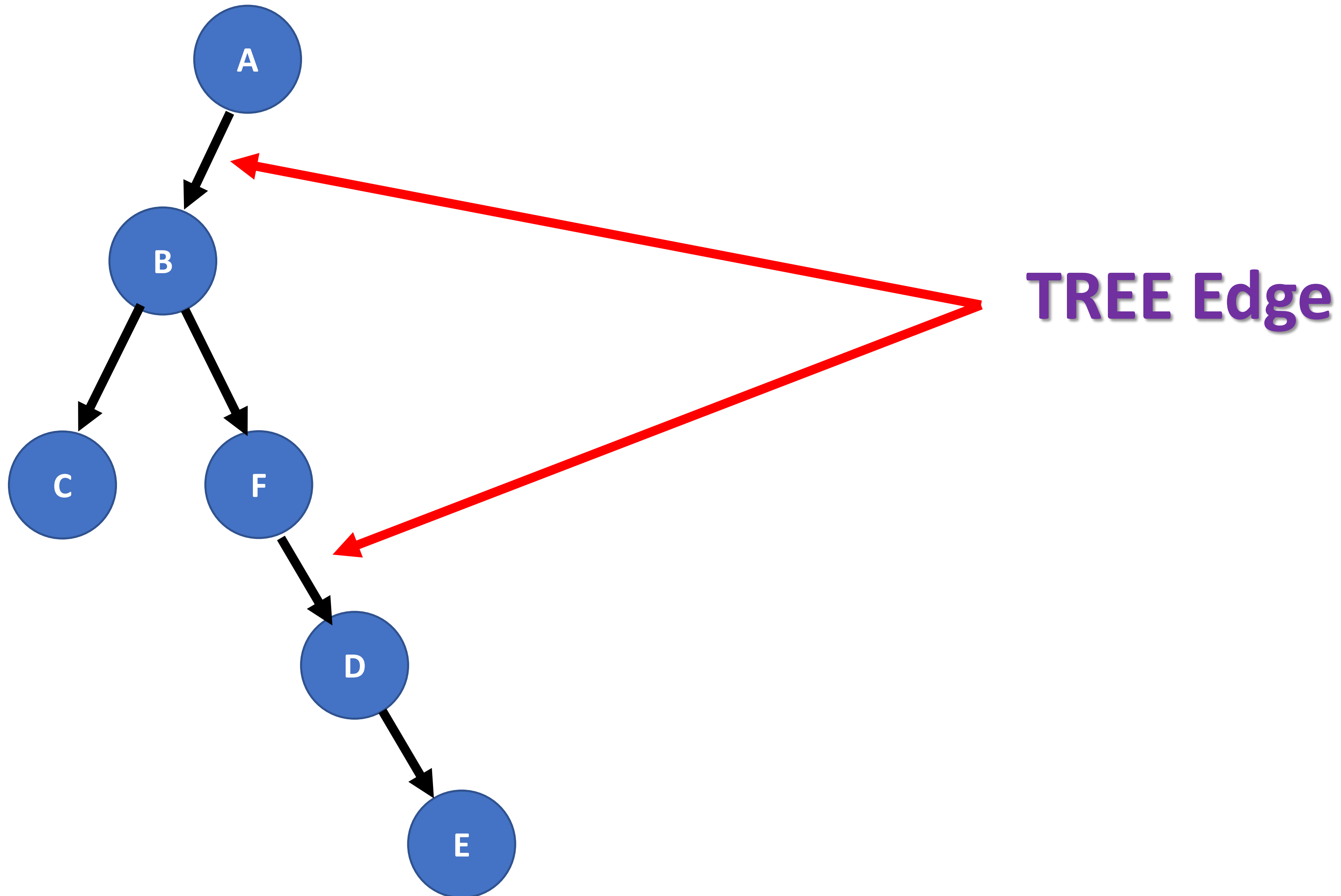


DFS traversal

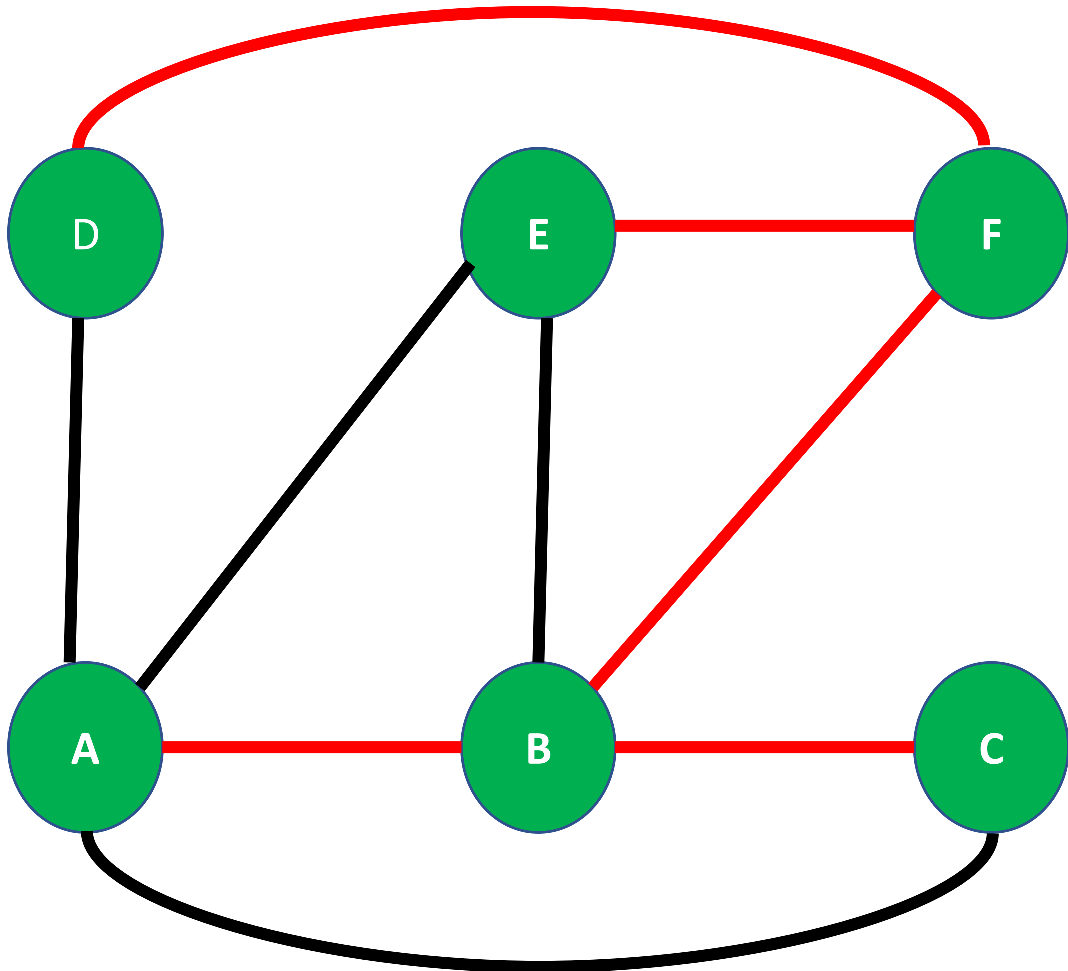
A, B, C, F, D, E

DFS Spanning TREE

Depth First Search



Depth First Search



A
B
C
F
D
E
Back Edge

Depth First Search

Algorithm DFS(v)

// Given undirected(directed) graph G=(V,E) with n vertices and an array
//visited[] initially set to 0 this algorithm visits all vertices reachable
//from v and visited[] are global.

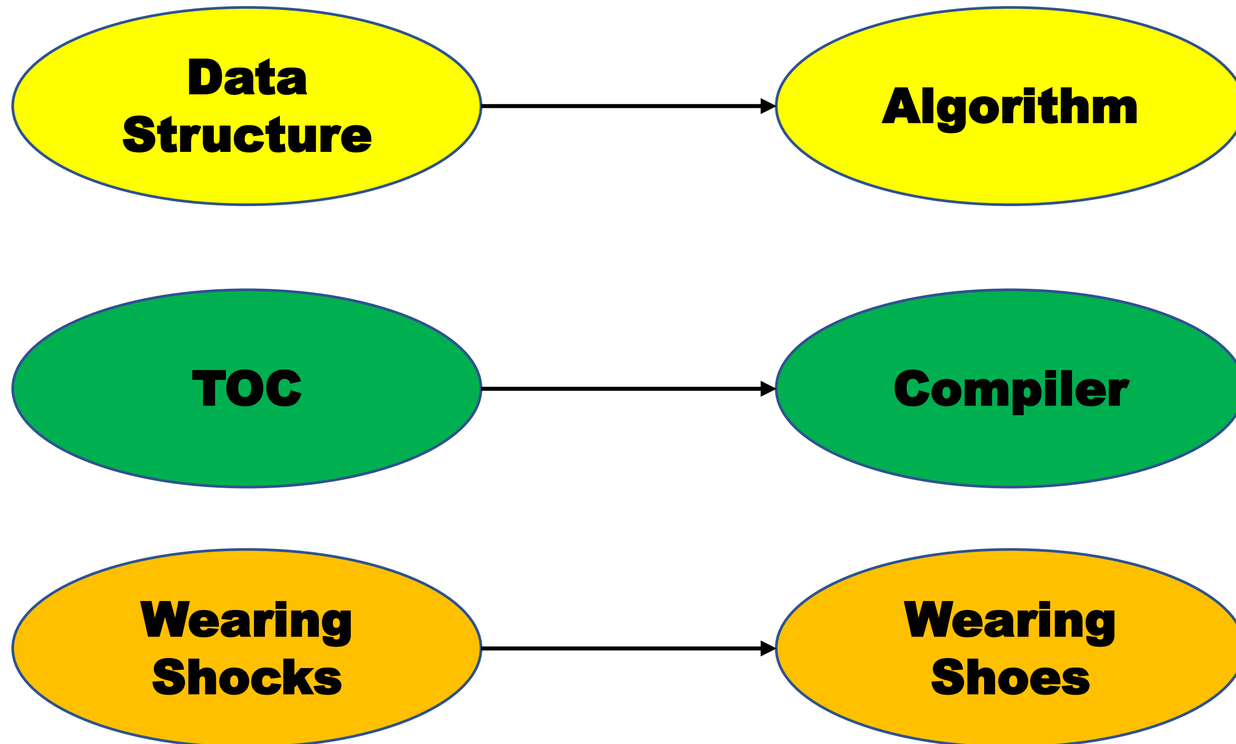
```
{
    visited[v] = 1
    for each vertex w adjacent to v do
        {
            if ( visited[w] == 0) then DFS(w)
        }
}
```

Depth First Search

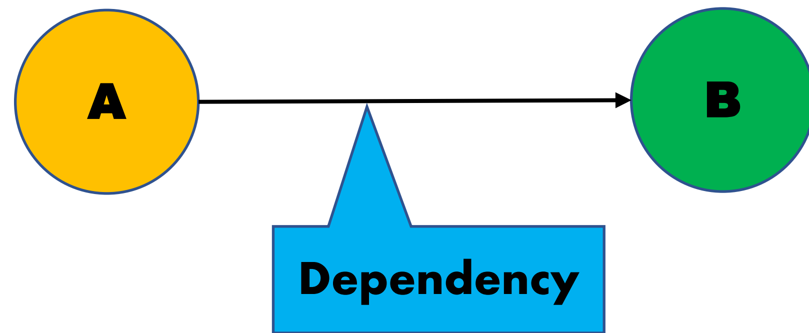
```
Algorithm DFS(v)
// Given undirected(directed) graph G=(V,E) with n vertices and an array
//visited[] initially set to 0 this algorithm visits all vertices reachable
//from v and visited[] are global.
 {
       visited[v] = 1
       for each vertex w adjacent to v do
             {
                   if ( visited[w] == 0) then DFS(w)
             }
 }
```

Time Complexity = O(V + E) = O(E)

Dependent Event



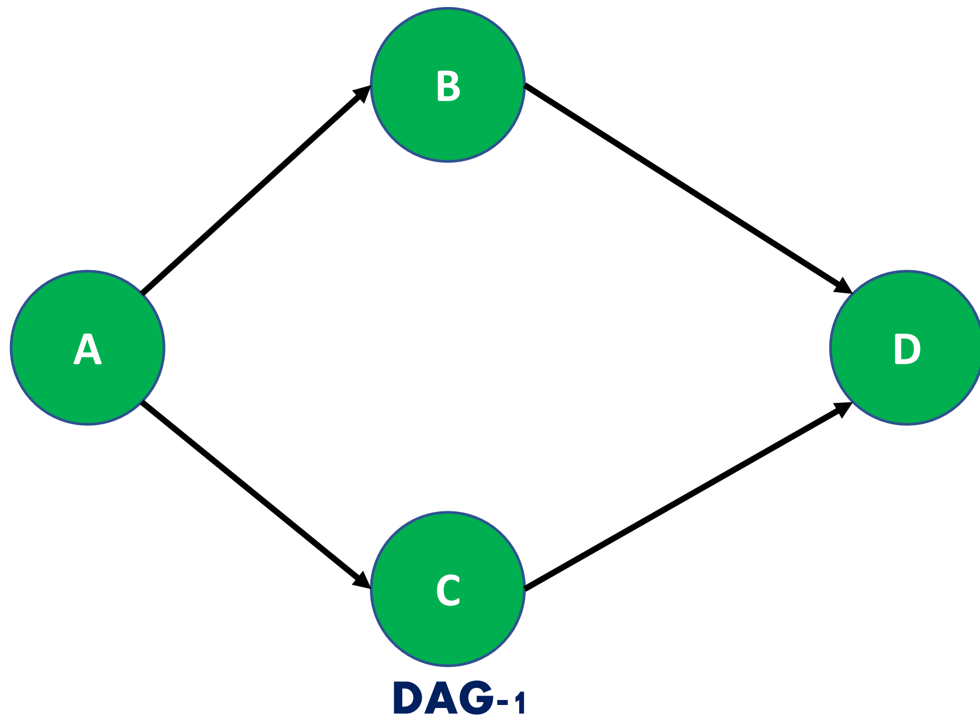
Dependent Event



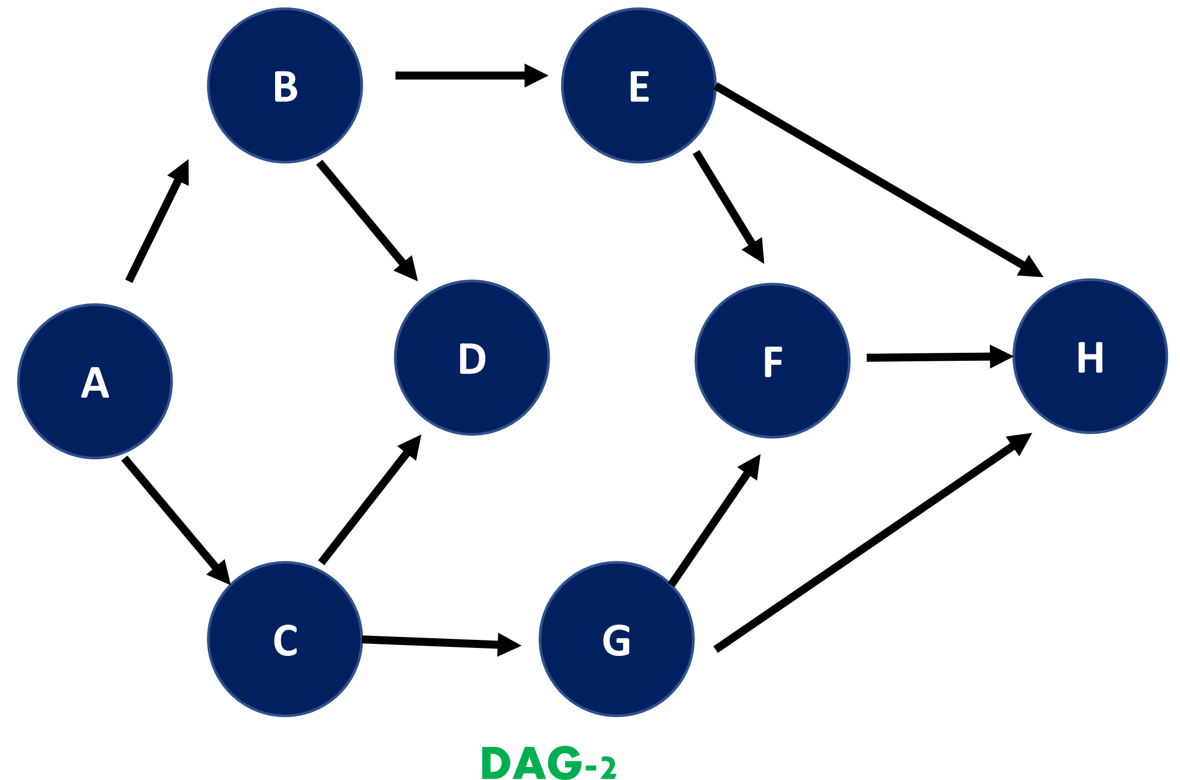
A must occur before the occurrence of B

Directed Acyclic Graph(DAG)

A graph is a DAG if it is Directed and Acyclic.



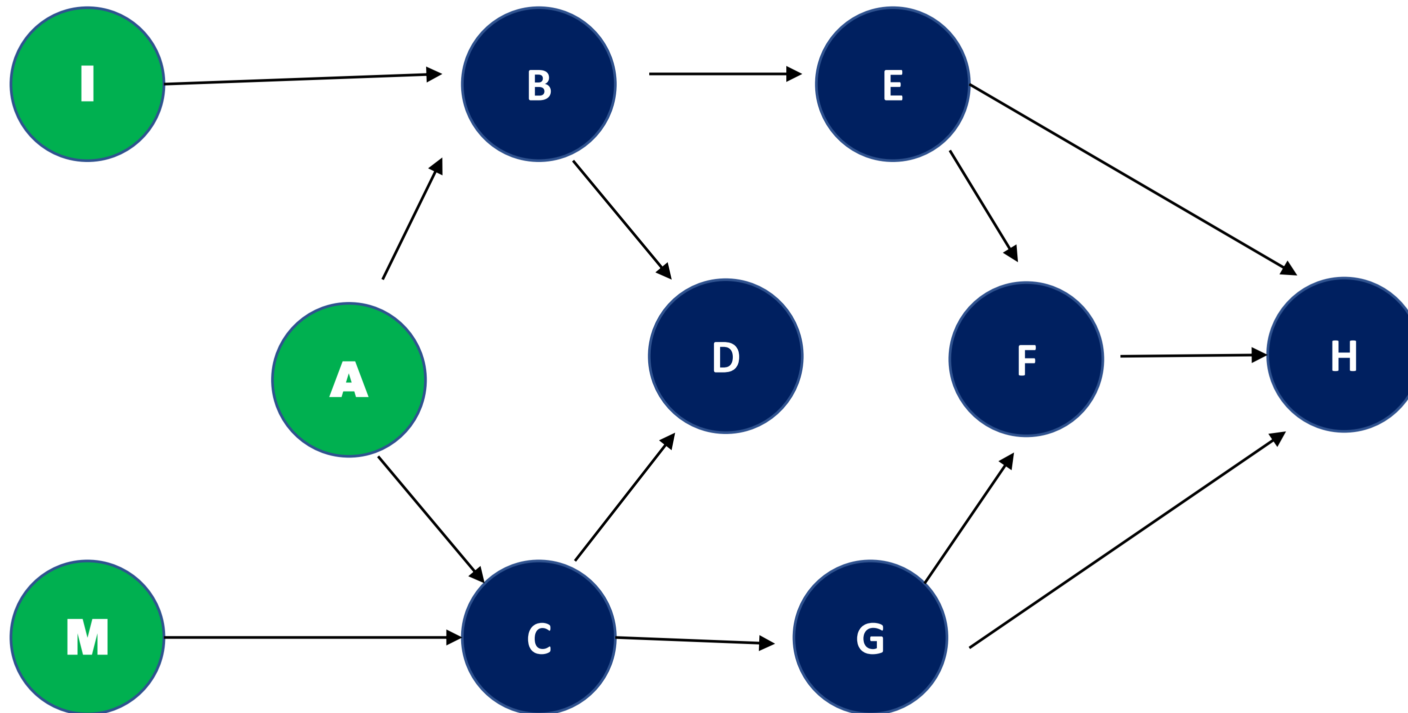
DAG-1



DAG-2

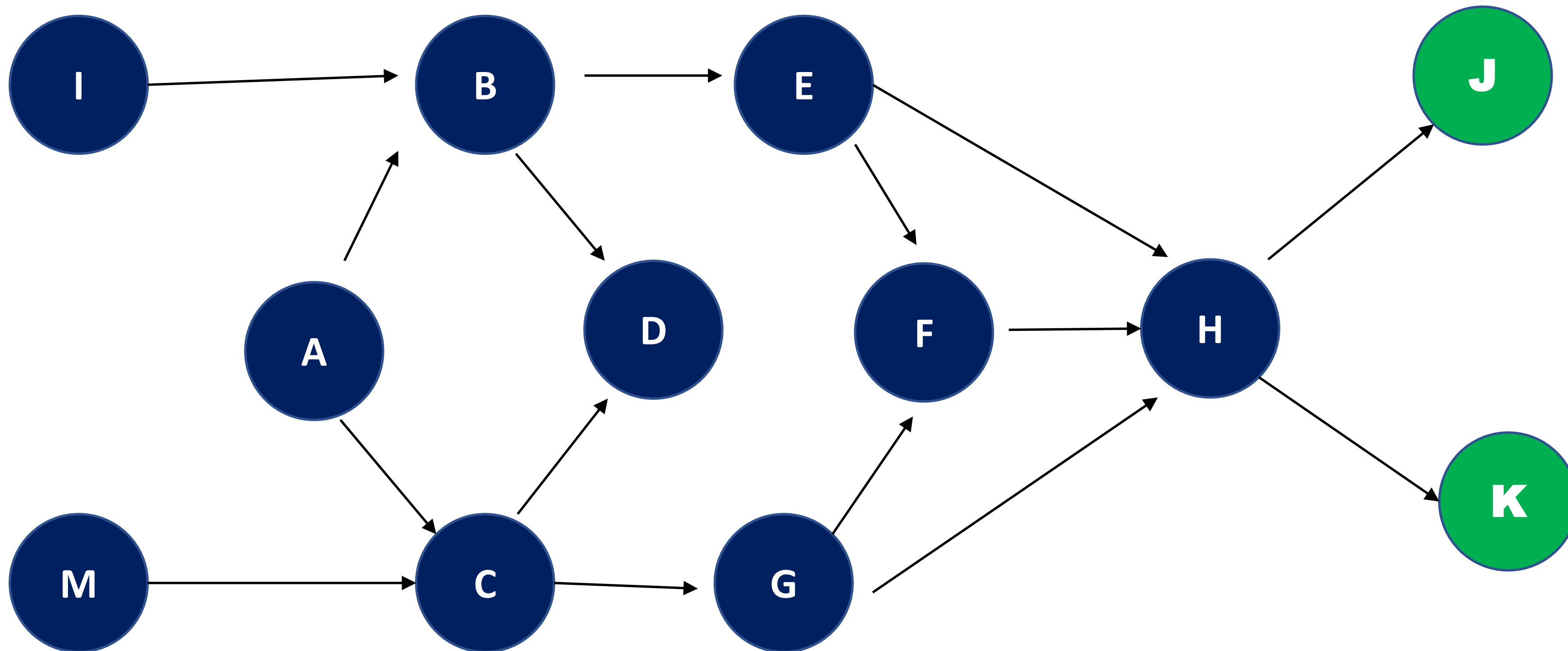
Directed Acyclic Graph(DAG)

Observation1: A DAG has at least one vertex whose indegree is 0



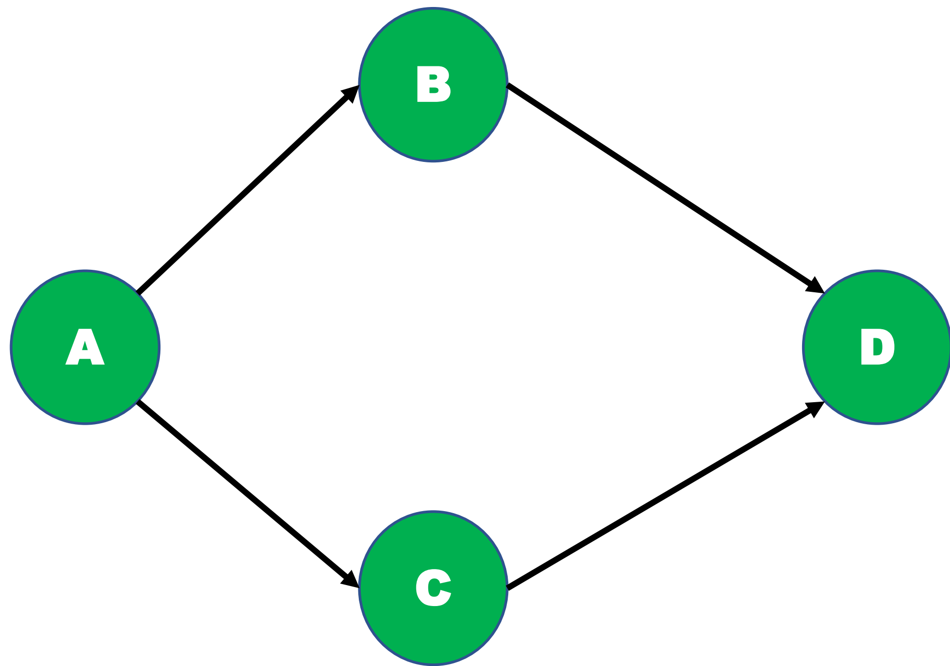
Directed Acyclic Graph(DAG)

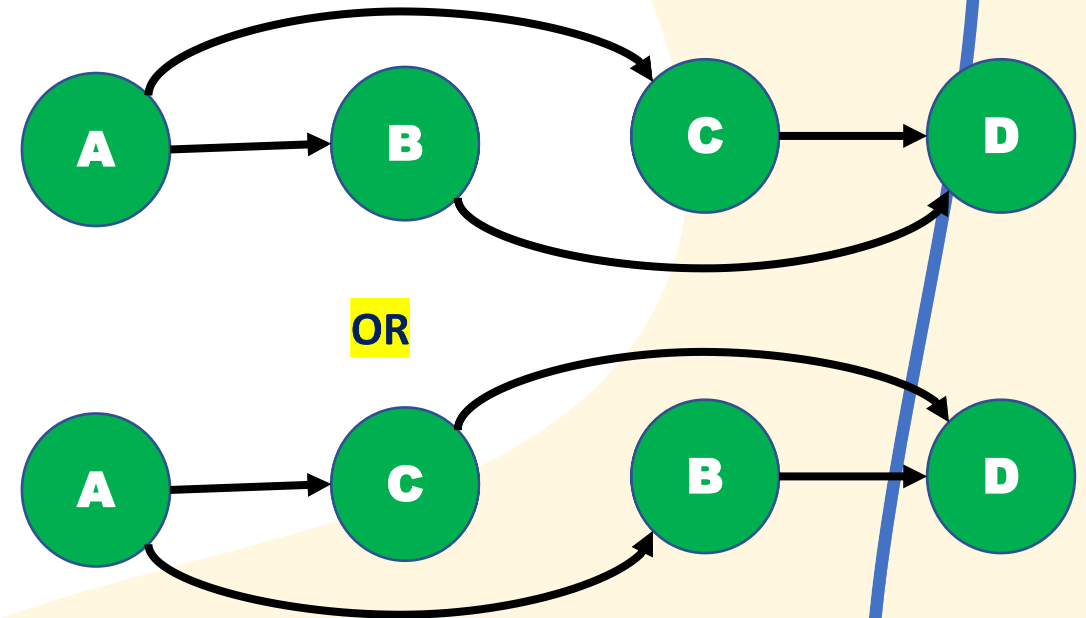
Observation2: A DAG has at least one vertex whose outdegree is 0



Topological Sorting or Ordering

Arranging the events/activities in linear/serial order by preserving their dependencies





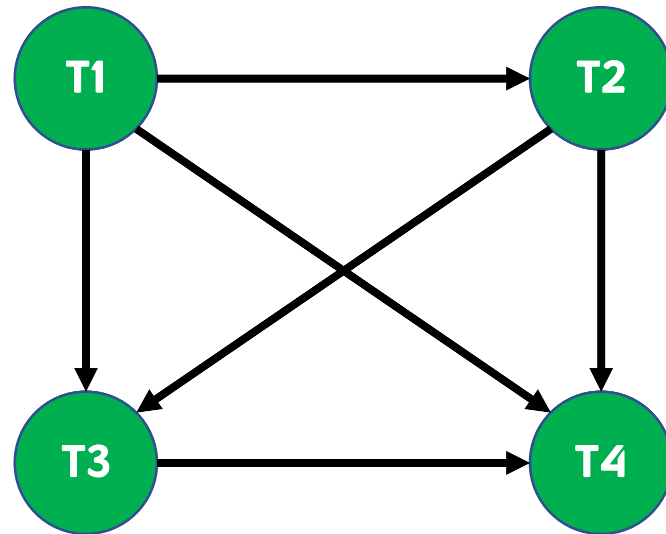
NOTE: Topological sorting is possible only for DAG

Topological Sorting or Ordering

Topological_Sorting(G)

1. Call DFS(G) to compute the finishing time f[v] for all vertices
2. Output vertices in decreasing order of their finishing time f[v]

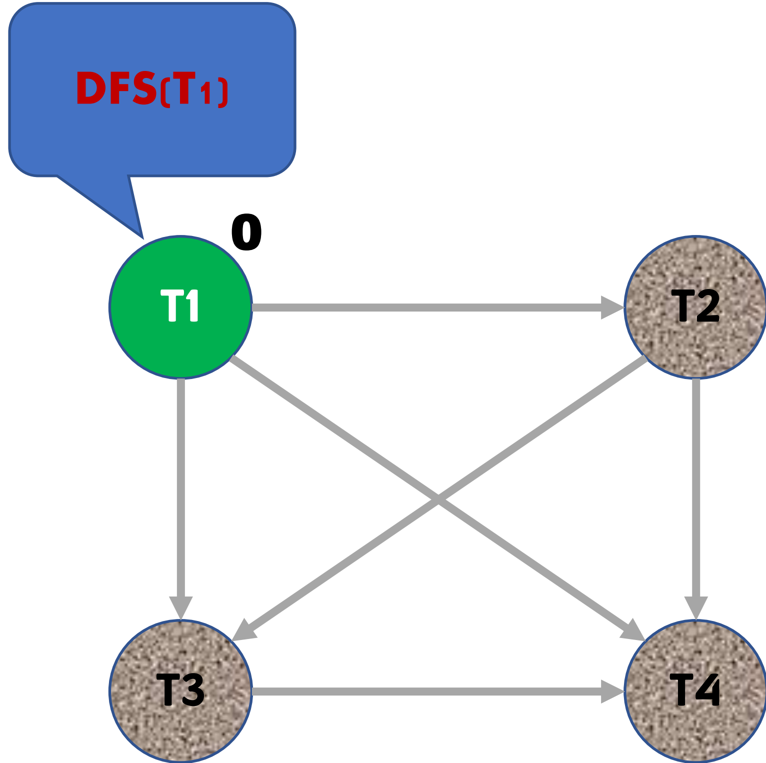
Topological Sorting using DFS



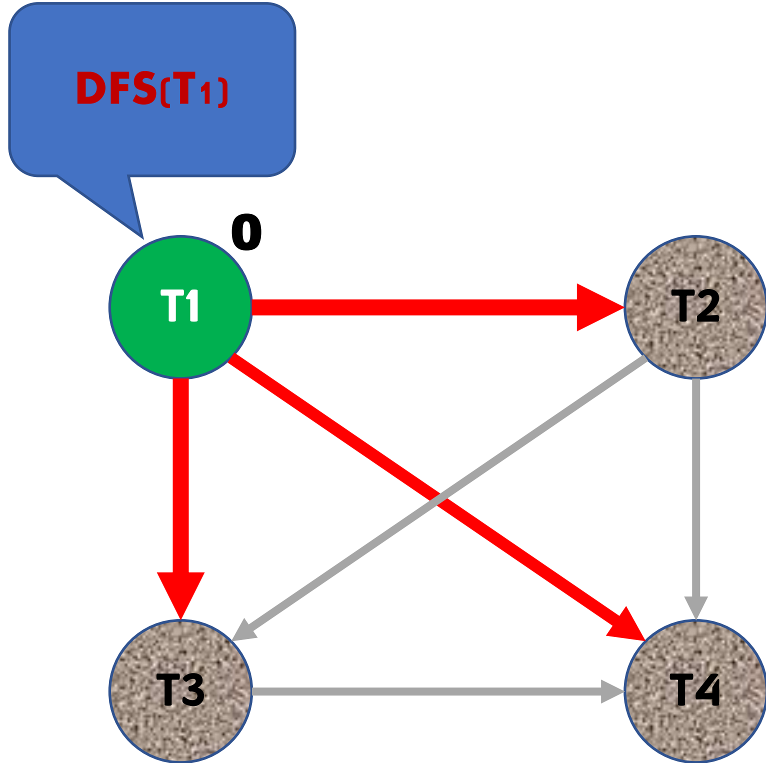
d[u] = discovery time of vertex u(time when we explore u first time)

f[u] = Finishing time of vertex u(time when we backtrack from u)

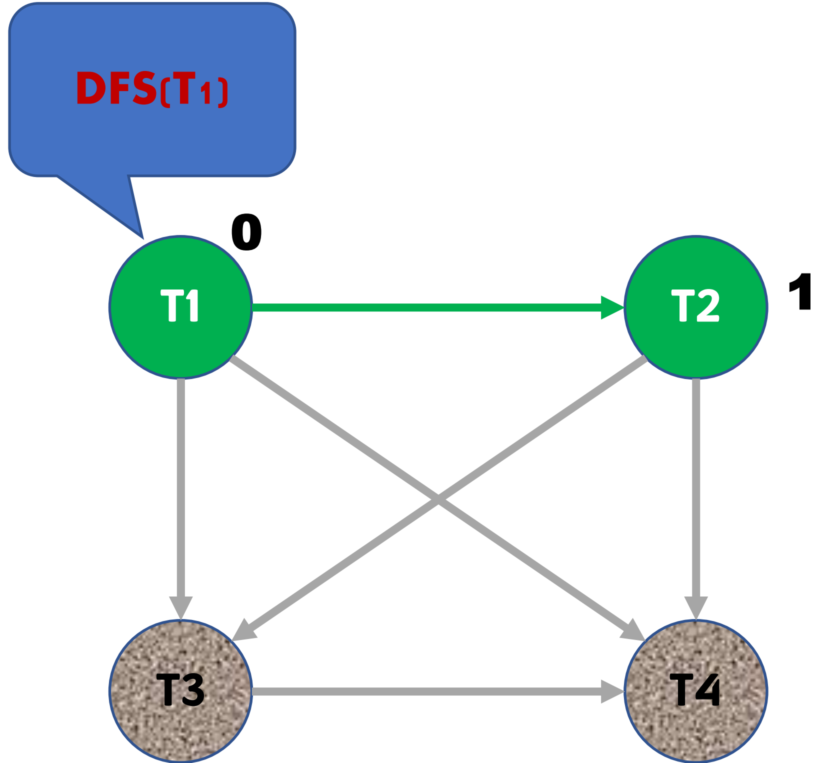
Topological Sorting using DFS



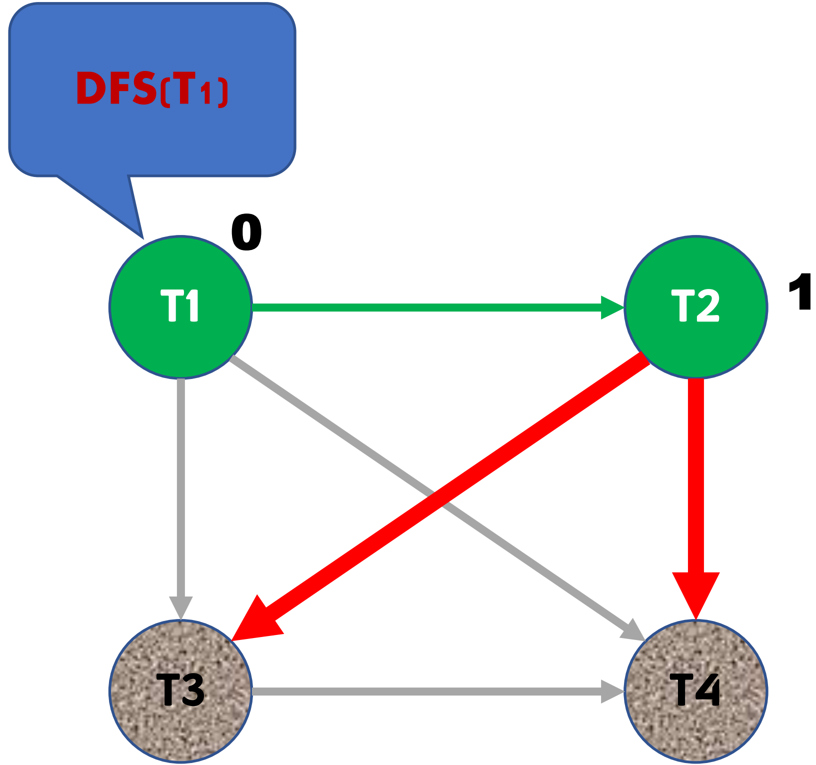
Topological Sorting using DFS



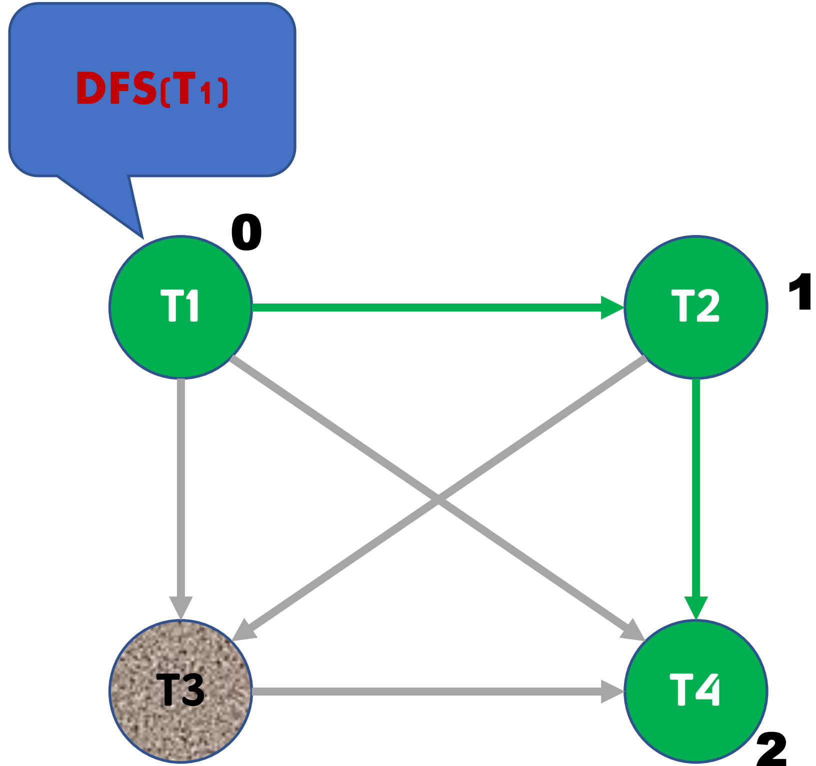
Topological Sorting using DFS



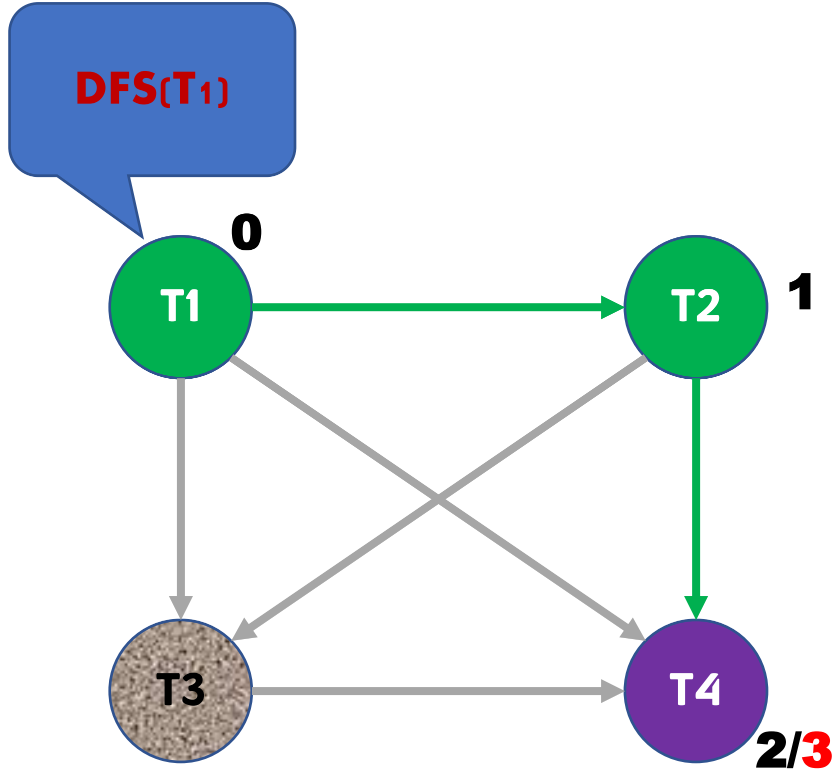
Topological Sorting using DFS



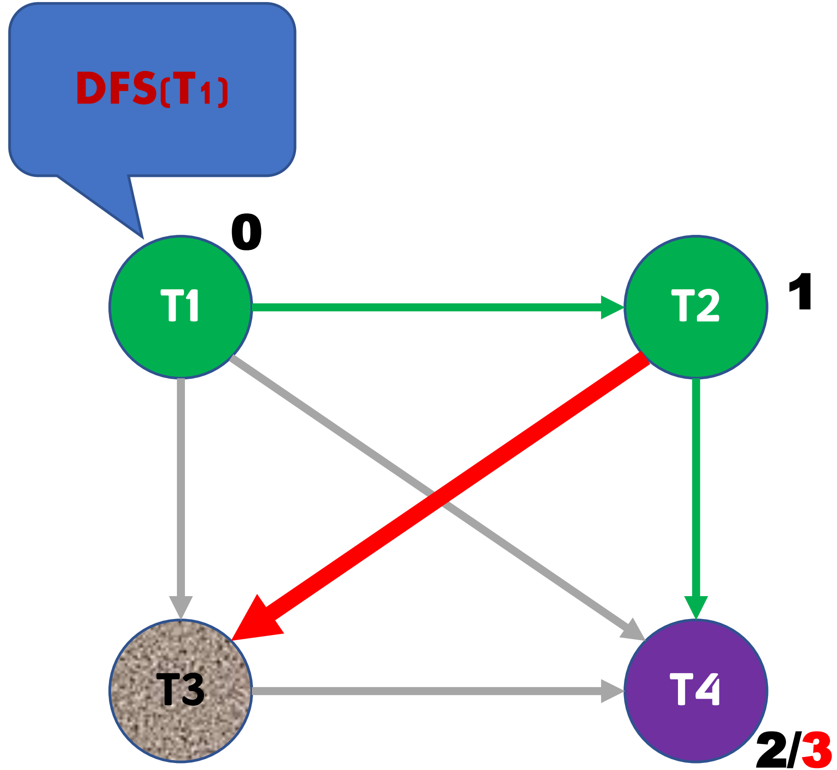
Topological Sorting using DFS



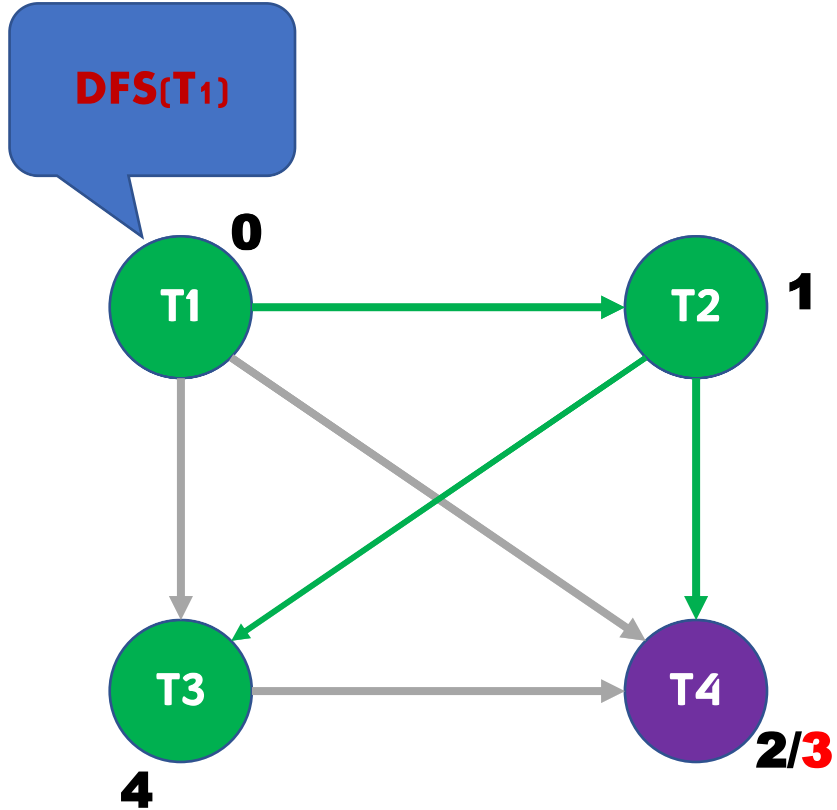
Topological Sorting using DFS



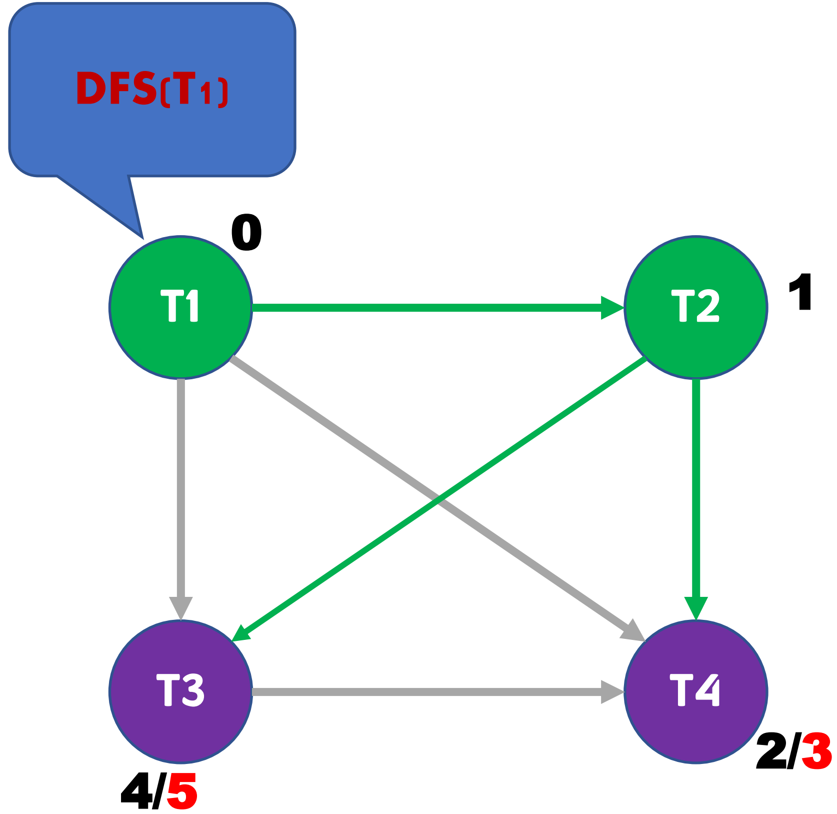
Topological Sorting using DFS



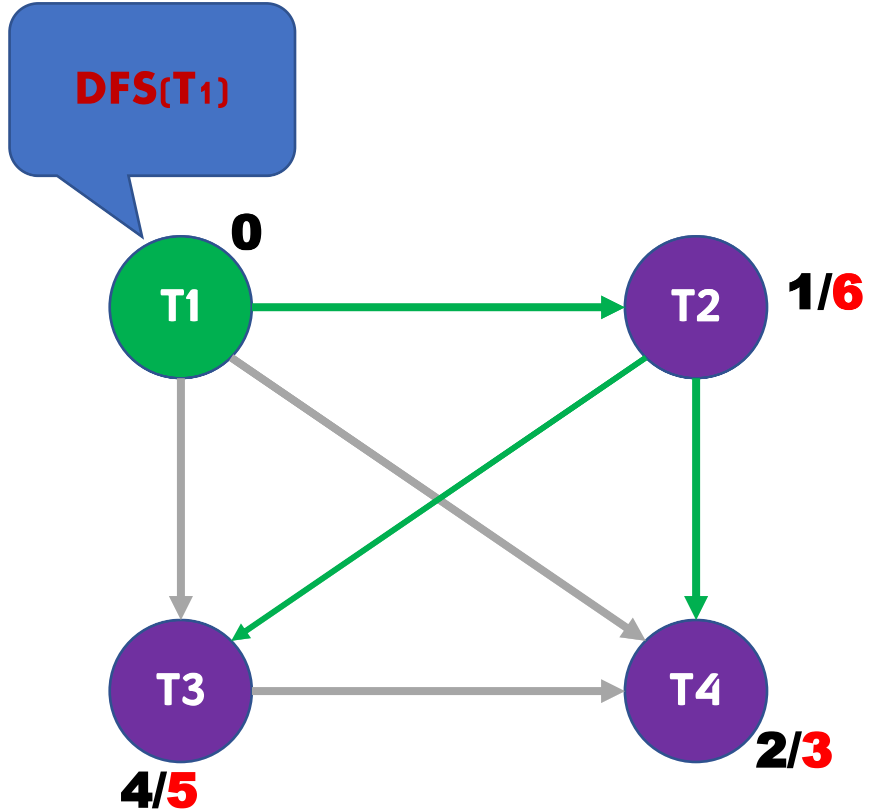
Topological Sorting using DFS



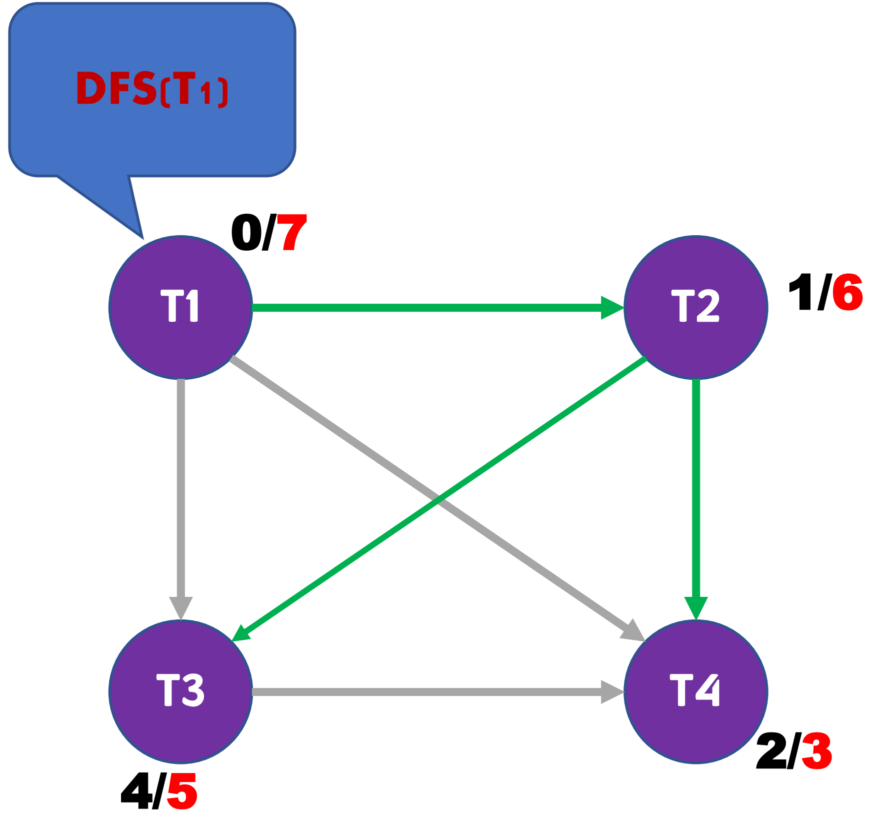
Topological Sorting using DFS



Topological Sorting using DFS



Topological Sorting using DFS



Topological Order is as follows

Topological Sorting using DFS

Cost Analysis

Topological(G)

1. Call DFS(G) to compute the finishing time f[v] for all vertices
2. Output vertices in decreasing order of their finishing time f[v]

O(V+E)

O(V)

Time Complexity = O(V+E)

Topological Sorting using DFS

Cost Analysis

Topological(G)

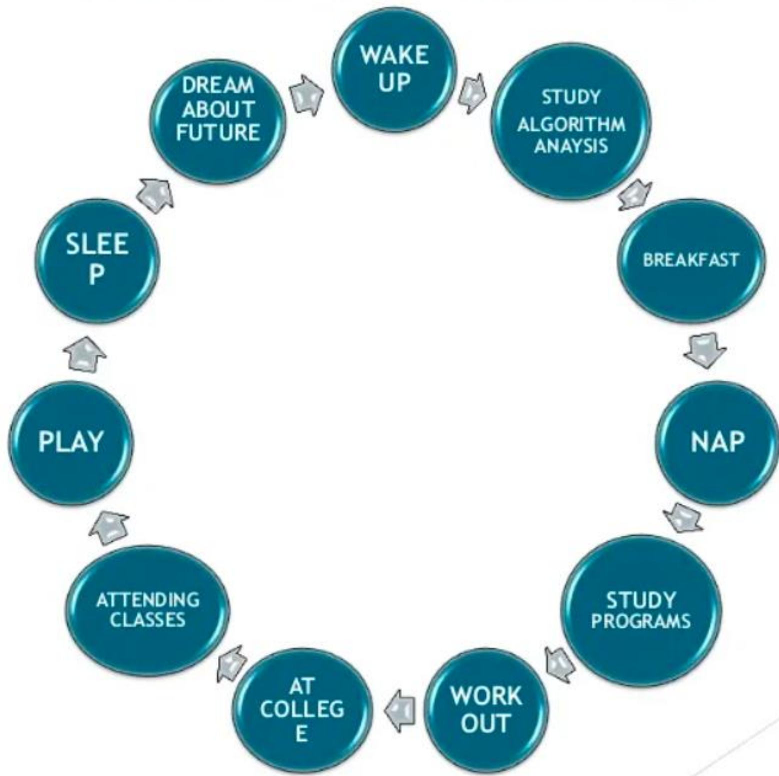
1. Call DFS(G) to compute the finishing time f[v] for all vertices
2. Output vertices in decreasing order of their finishing time f[v]

Space Complexity = O(V+E)

Topological Sort

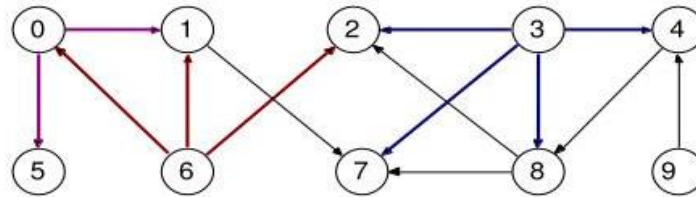
- Directed Acyclic Graph (dag): a directed graph which contains no directed cycles.
- A topological order for a dag G: a sequential listing of all the vertices of G such that if (u,v) € E(G) then u precedes v in the listing.
- Topological Sort: an algorithm that takes as input a dag G and produces a topological order of the vertices of G
- Example Applications: ·
  - Prerequisites dag for courses at a University
  - Glossary of technical terms: (t1,t2) € E(G) iff t1 is used in the definition of t2.

Graphs

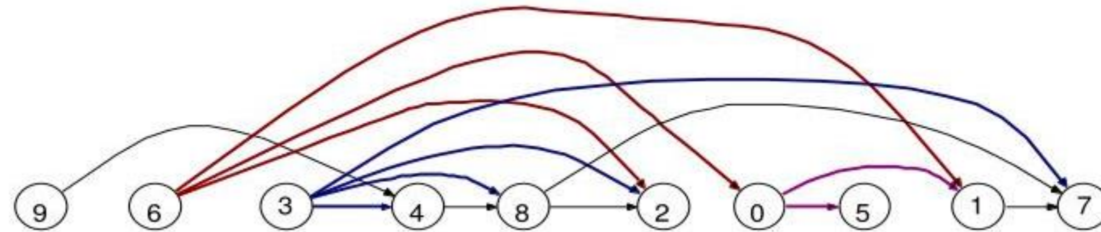


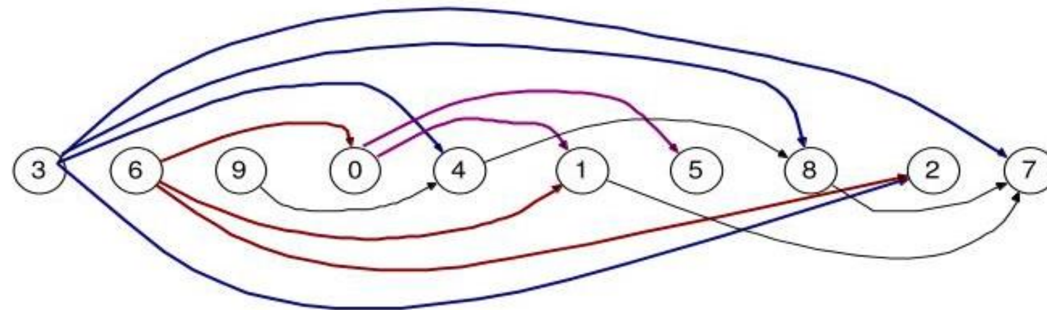
Graphs

In general, topological order of a dag is not unique.

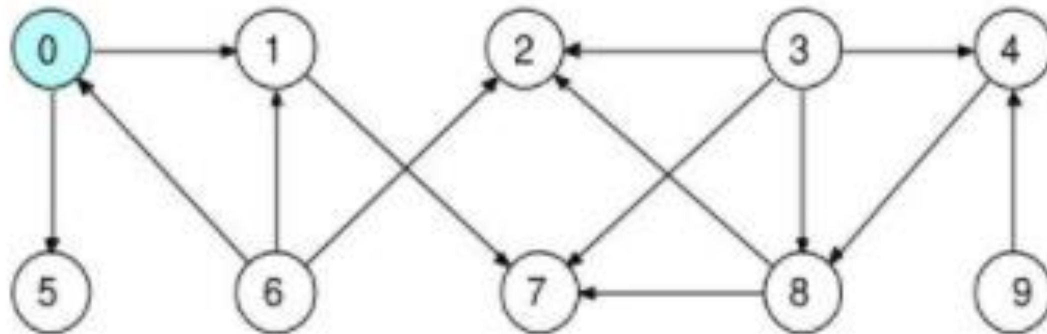


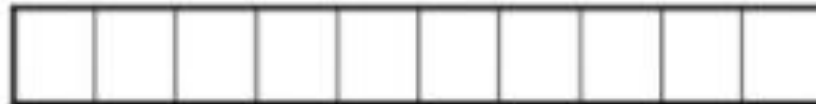
Directed Acyclic Graph G





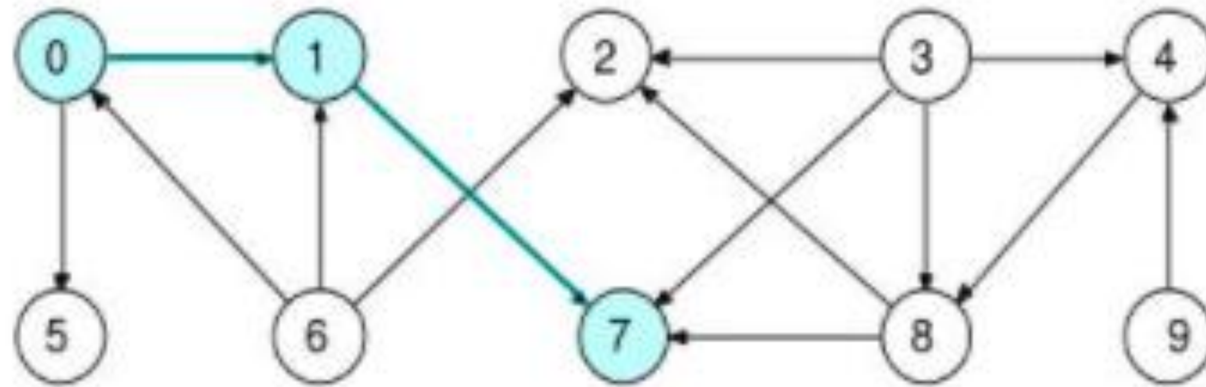
Graphs

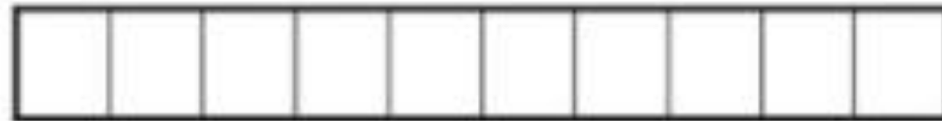




Graphs

Graphs





Graphs

									7

Graphs

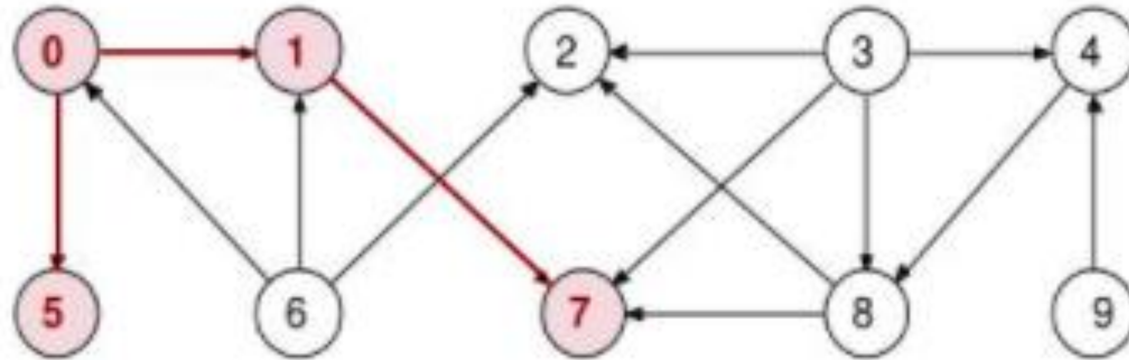
								1	7

Graphs

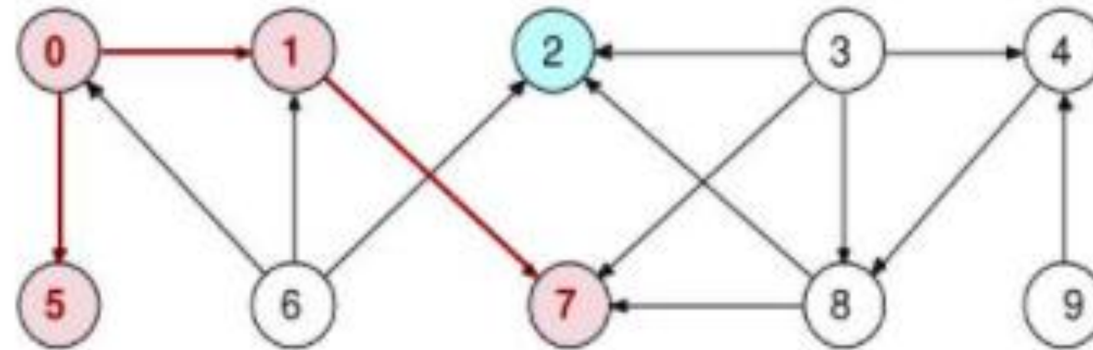
								1	7

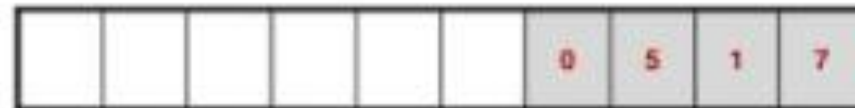
Graphs

Graphs

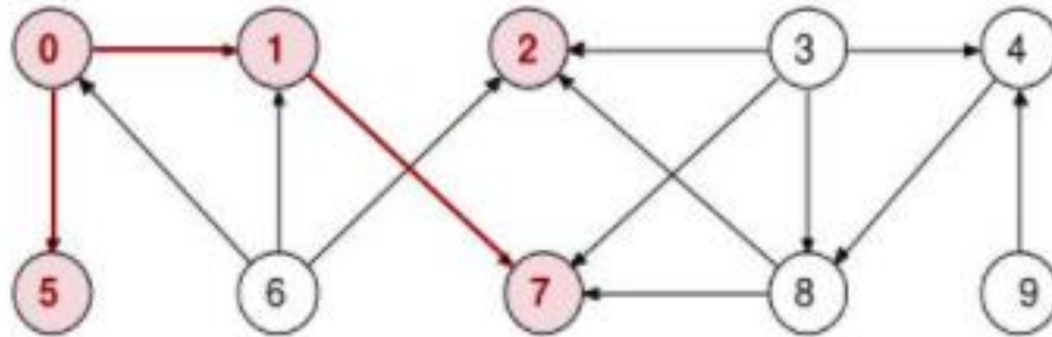


Graphs

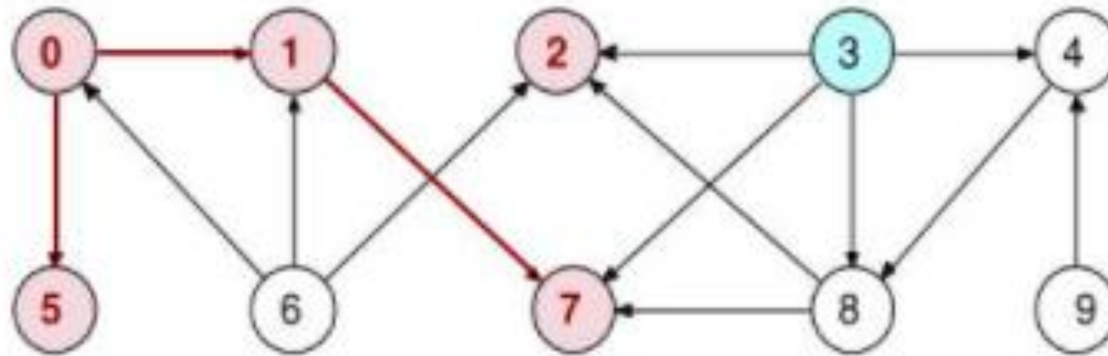




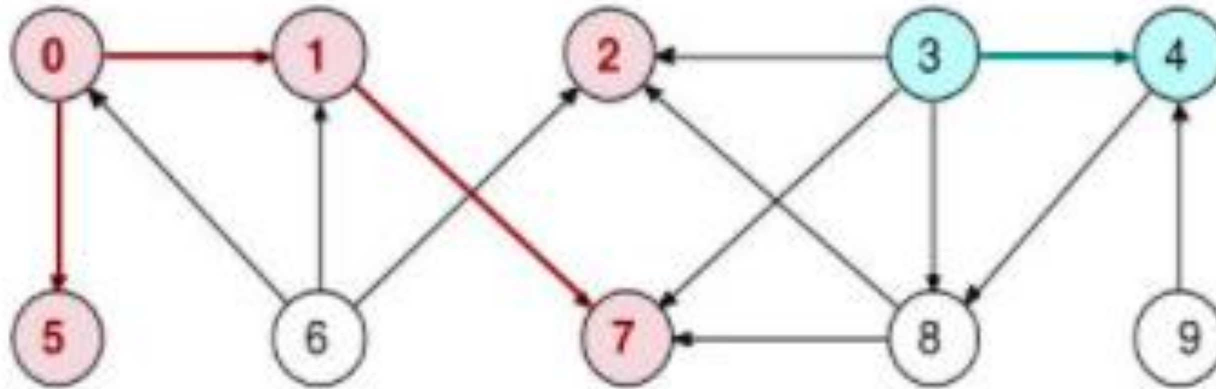
Graphs



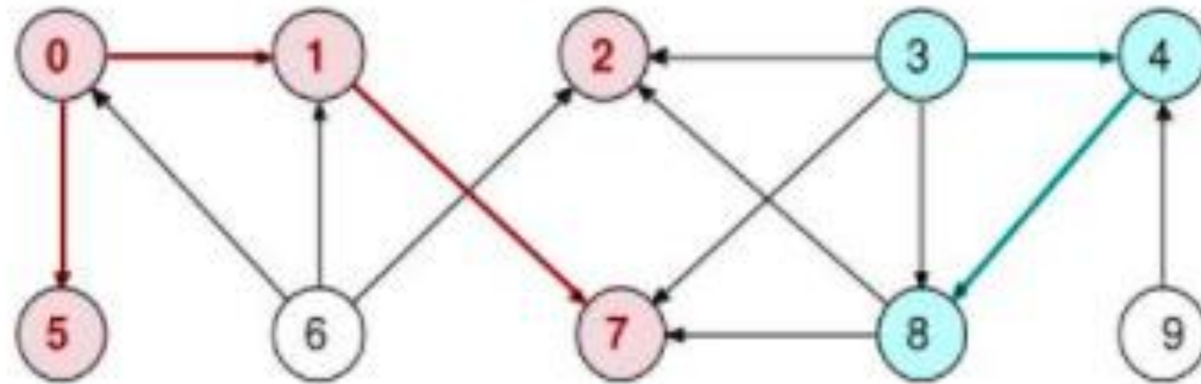
Graphs



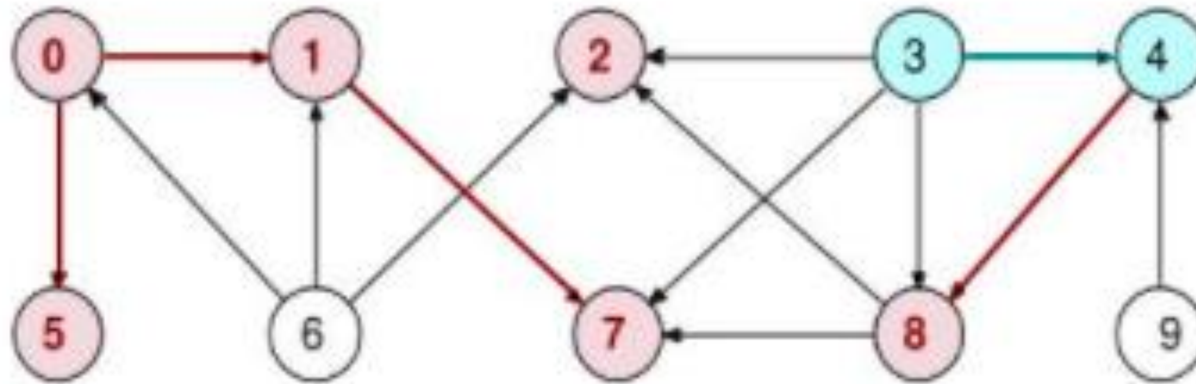
Graphs

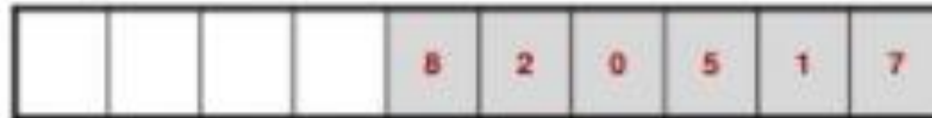


Graphs

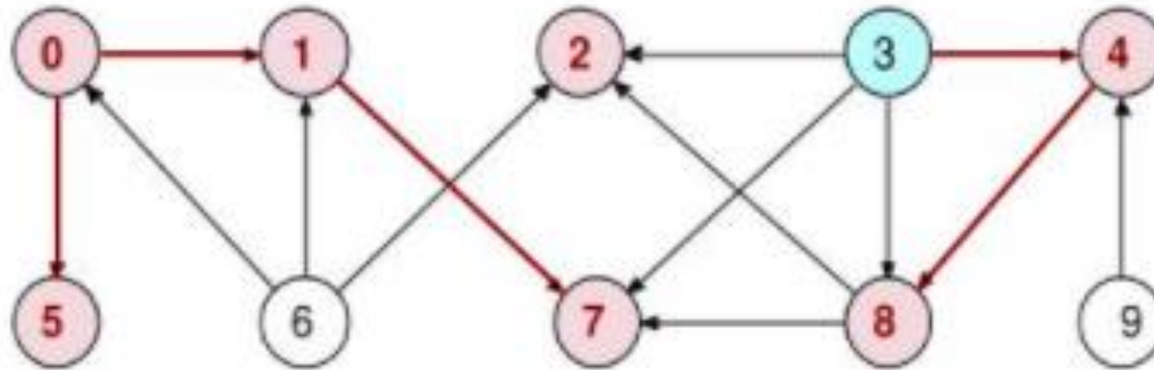


Graphs

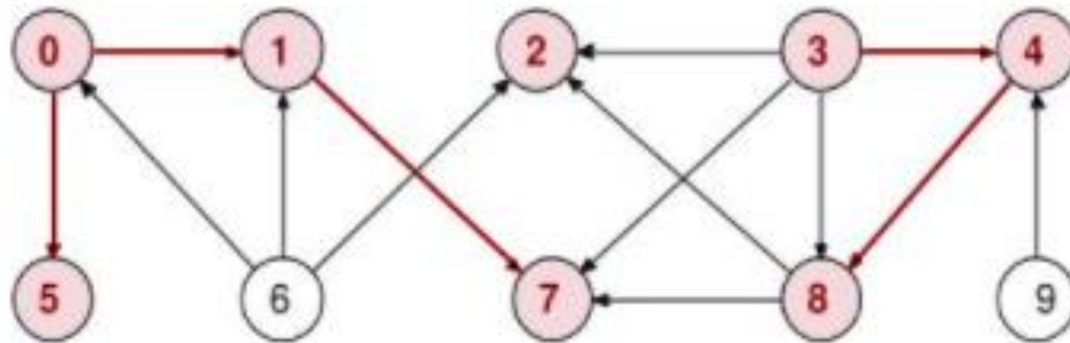


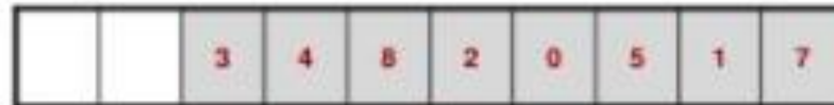


Graphs

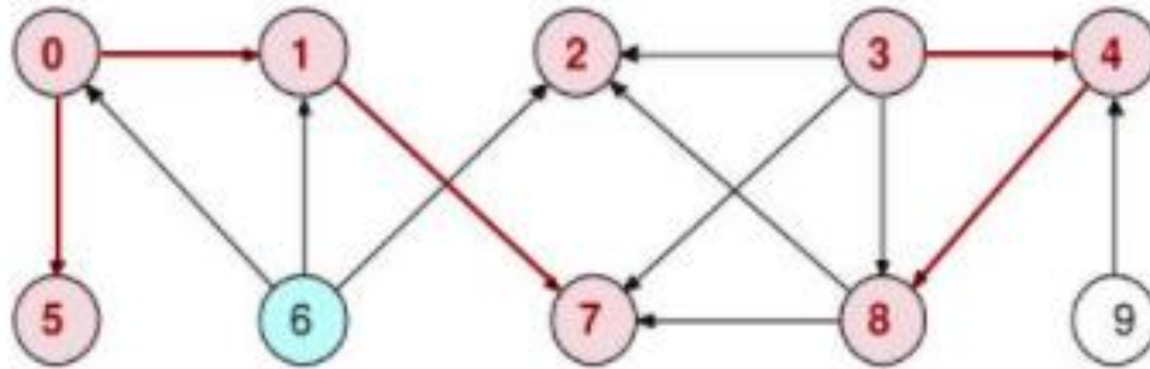


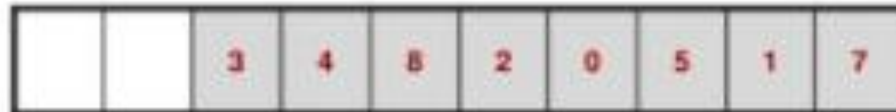
Graphs



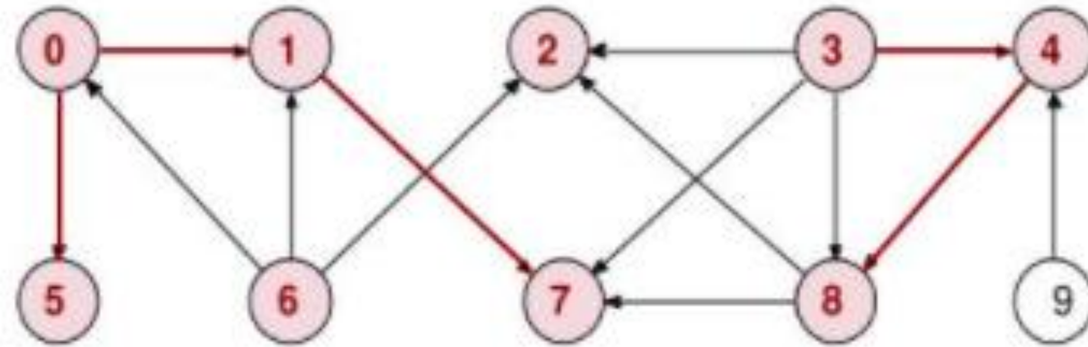


Graphs



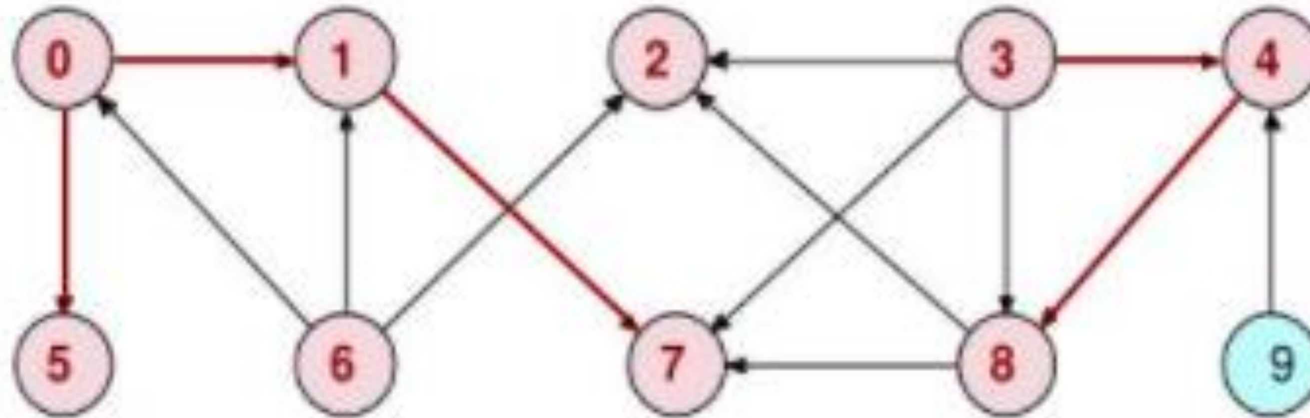


Graphs



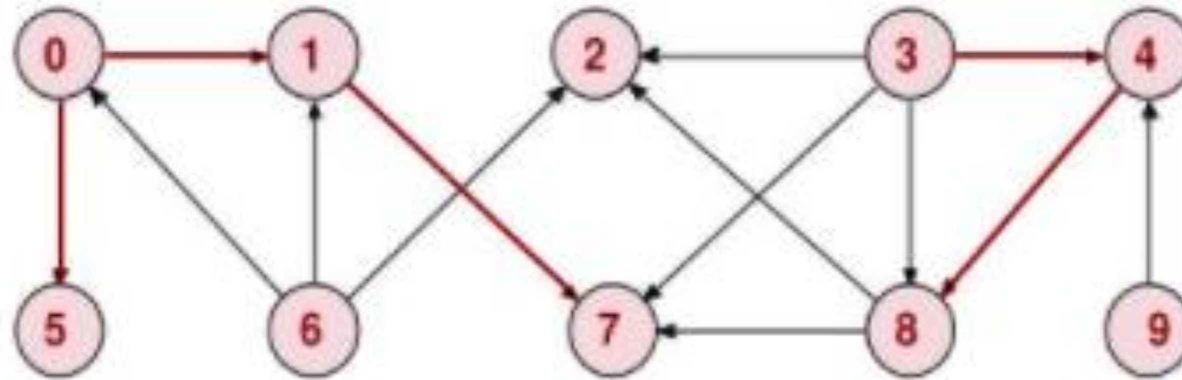
	6	3	4	8	2	0	5	1	7

Graphs



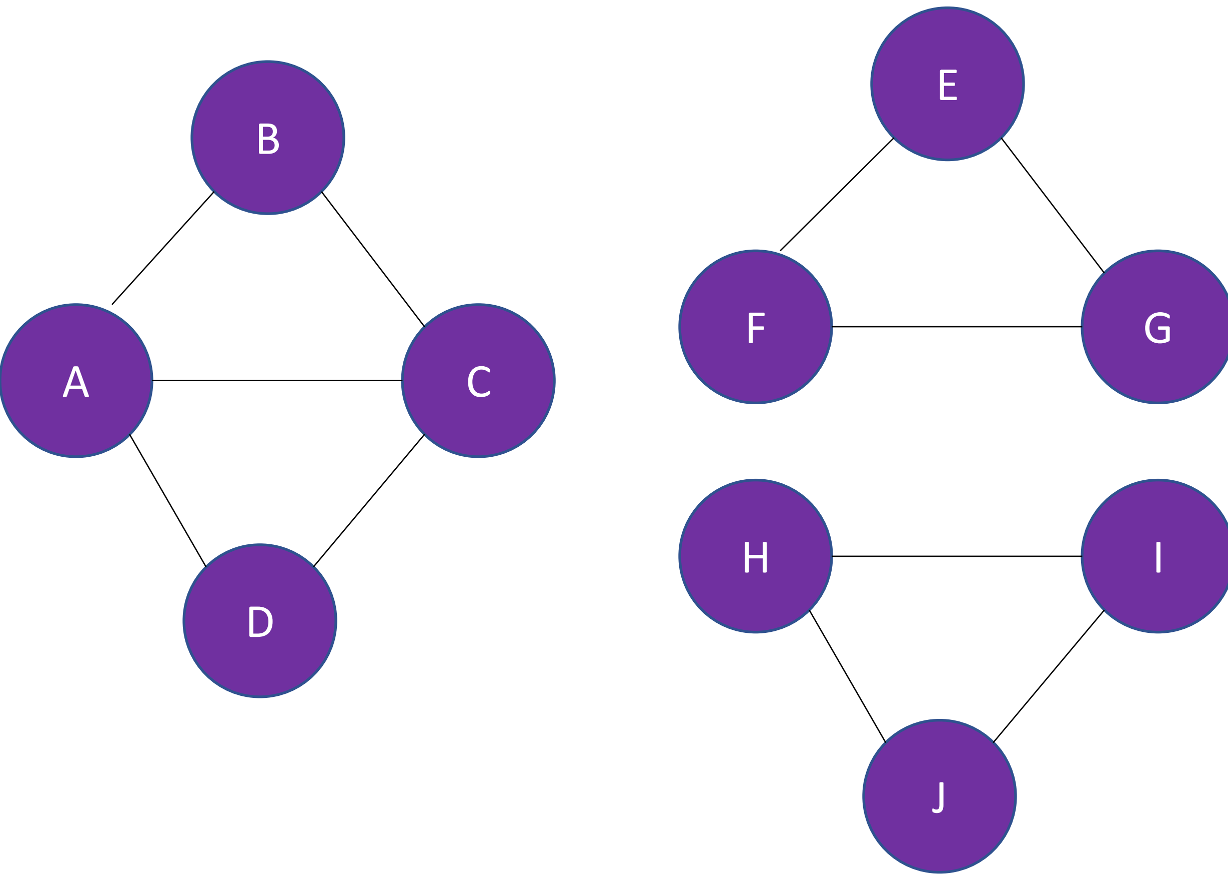
	6	3	4	8	2	0	5	1	7

Graphs



9	6	3	4	8	2	0	5	1	7

Connected Components using DFS



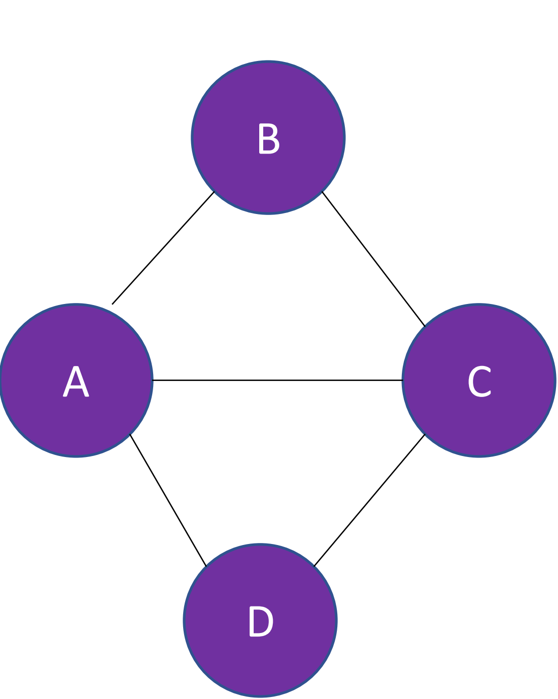
No of Connected Components = 3

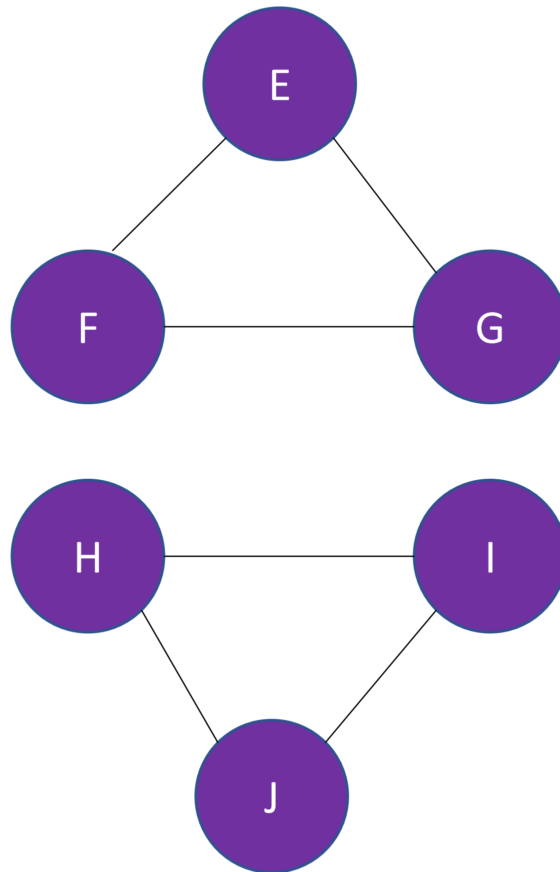
Connected Components using DFS

Algorithm 1: Finding Connected Components using DFS

```
Data: Given an undirected graph G(V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex k ∈ V do
 |  Visited[k] = False;
end
for each vertex k ∈ V do
 |  if Visited[k] == False then
 |   |  DFS(V,k);
 |   |  Component_Count = Component_Count + 1;
 |  end
end
Print Component_Count;
```

Connected Components using DFS



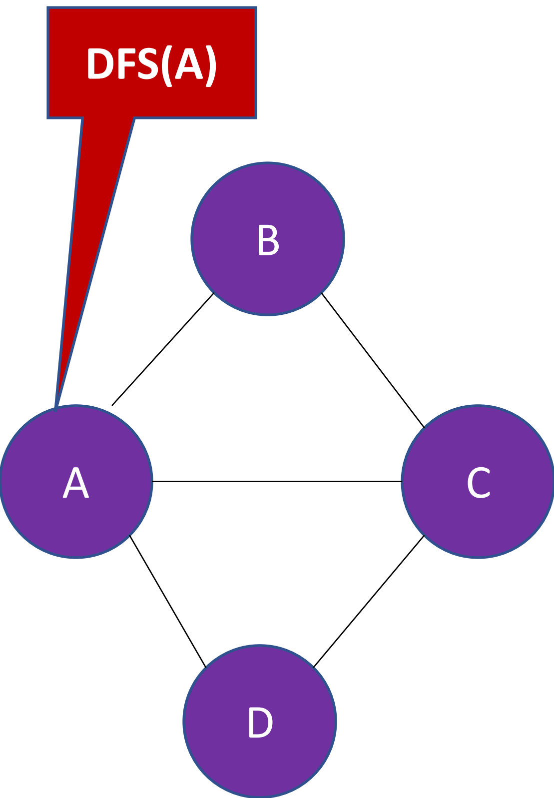


NCC=0

Visited	Vertex
0	A
0	B
0	C
0	D
0	E
0	F
0	G
0	H
0	I
0	J

DFS(A)

Connected Components using DFS

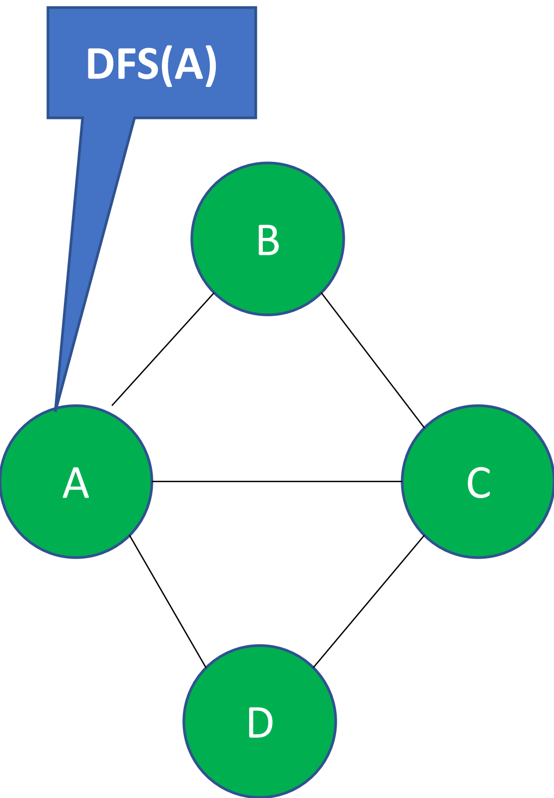


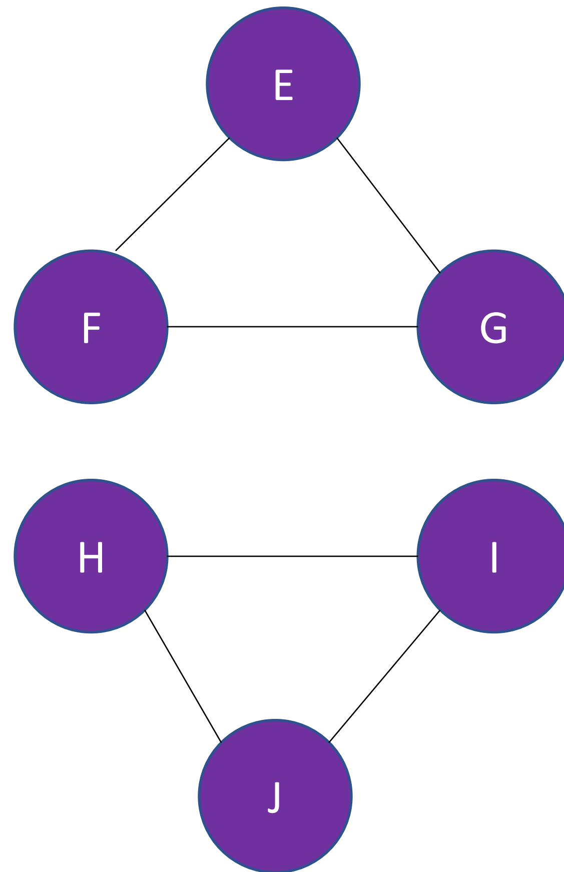
E
F
G
H
I
J

NCC=0

Visited	Vertex
0	A
0	B
0	C
0	D
0	E
0	F
0	G
0	H
0	I
0	J

Connected Components using DFS

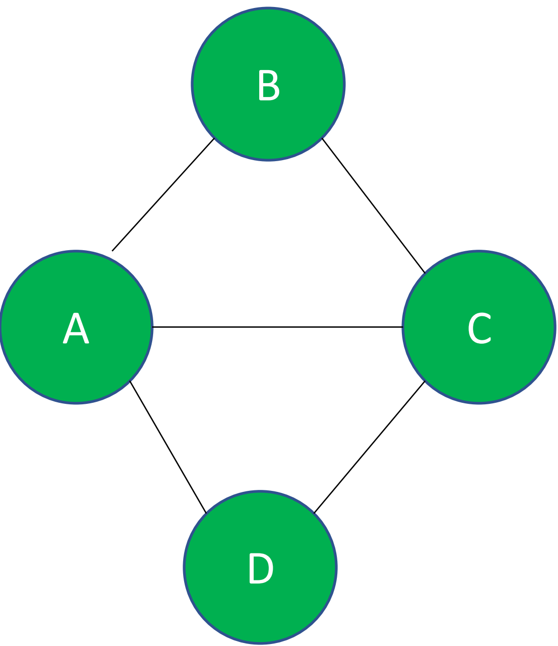


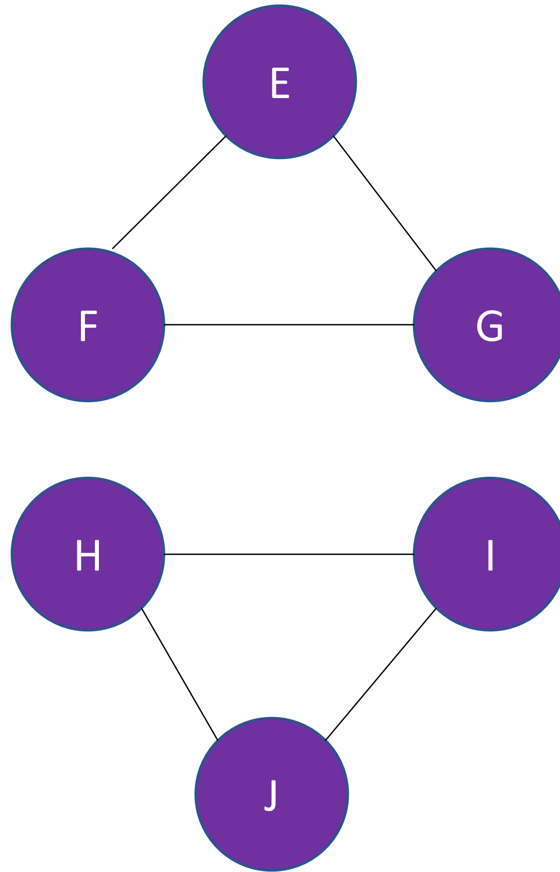


NCC=1

Visited	Vertex
1	A
1	B
1	C
1	D
0	E
0	F
0	G
0	H
0	I
0	J

Connected Components using DFS



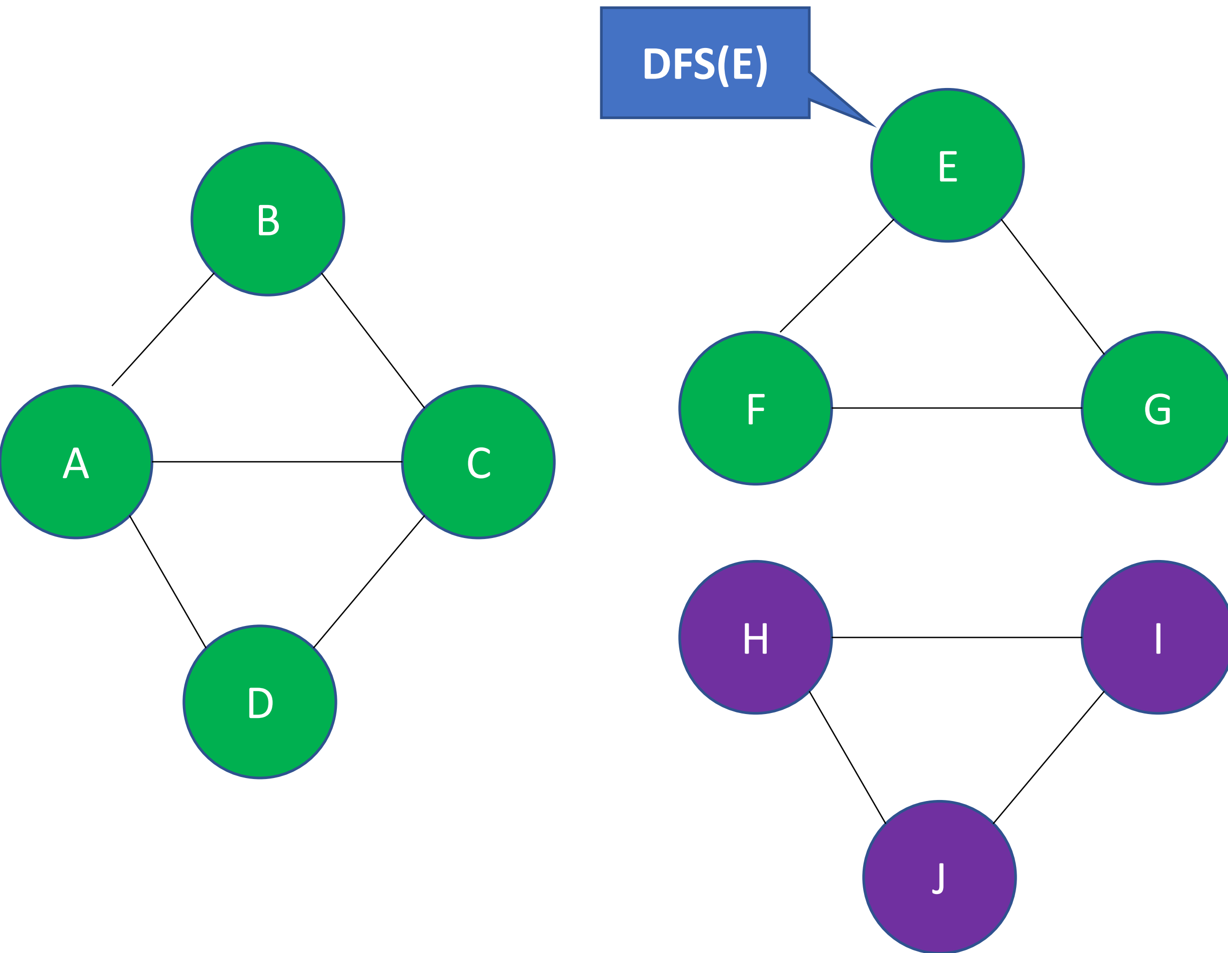


NCC=1

Visited	Vertex
1	A
1	B
1	C
1	D
0	E
0	F
0	G
0	H
0	I
0	J

DFS(E)

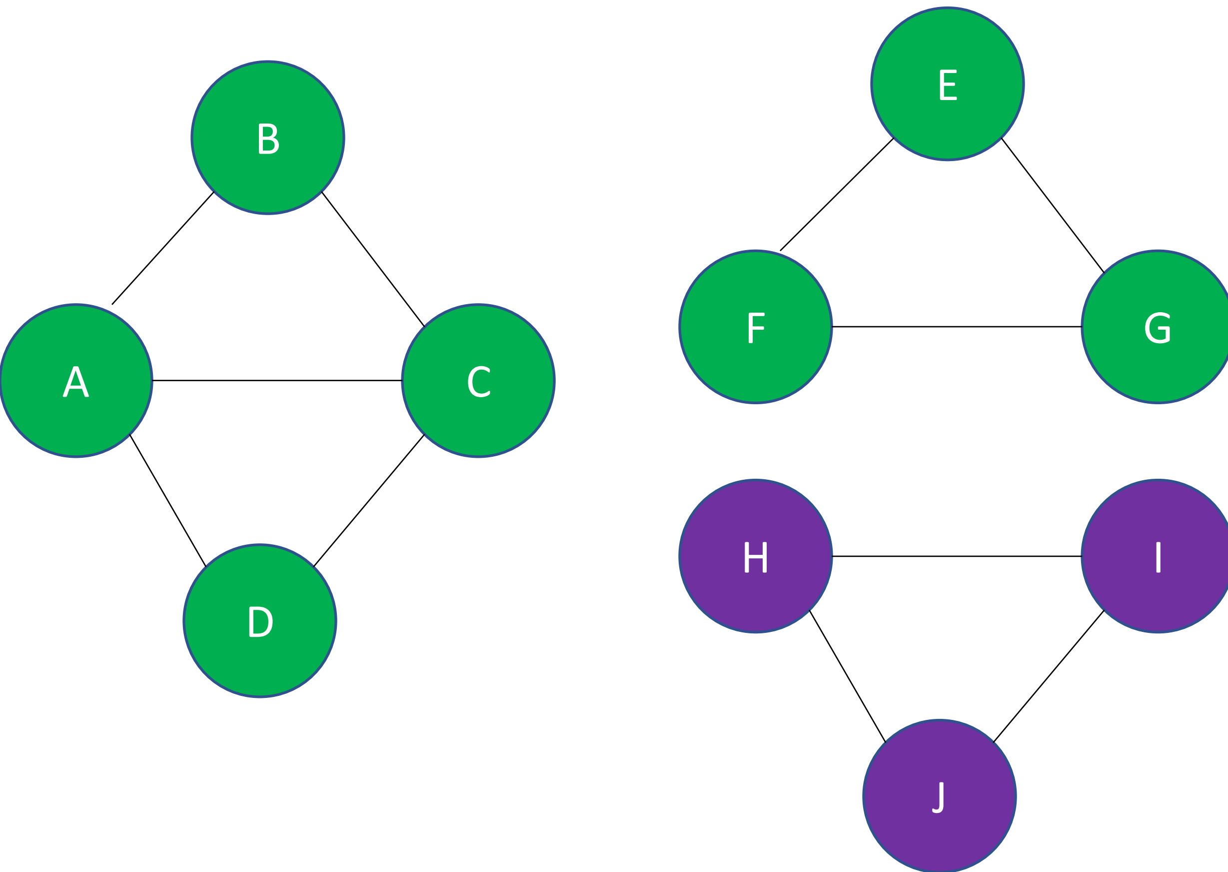
Connected Components using DFS



NCC=1

Visited	Vertex
1	A
1	B
1	C
1	D
1	E
1	F
1	G
0	H
0	I
0	J

Connected Components using DFS

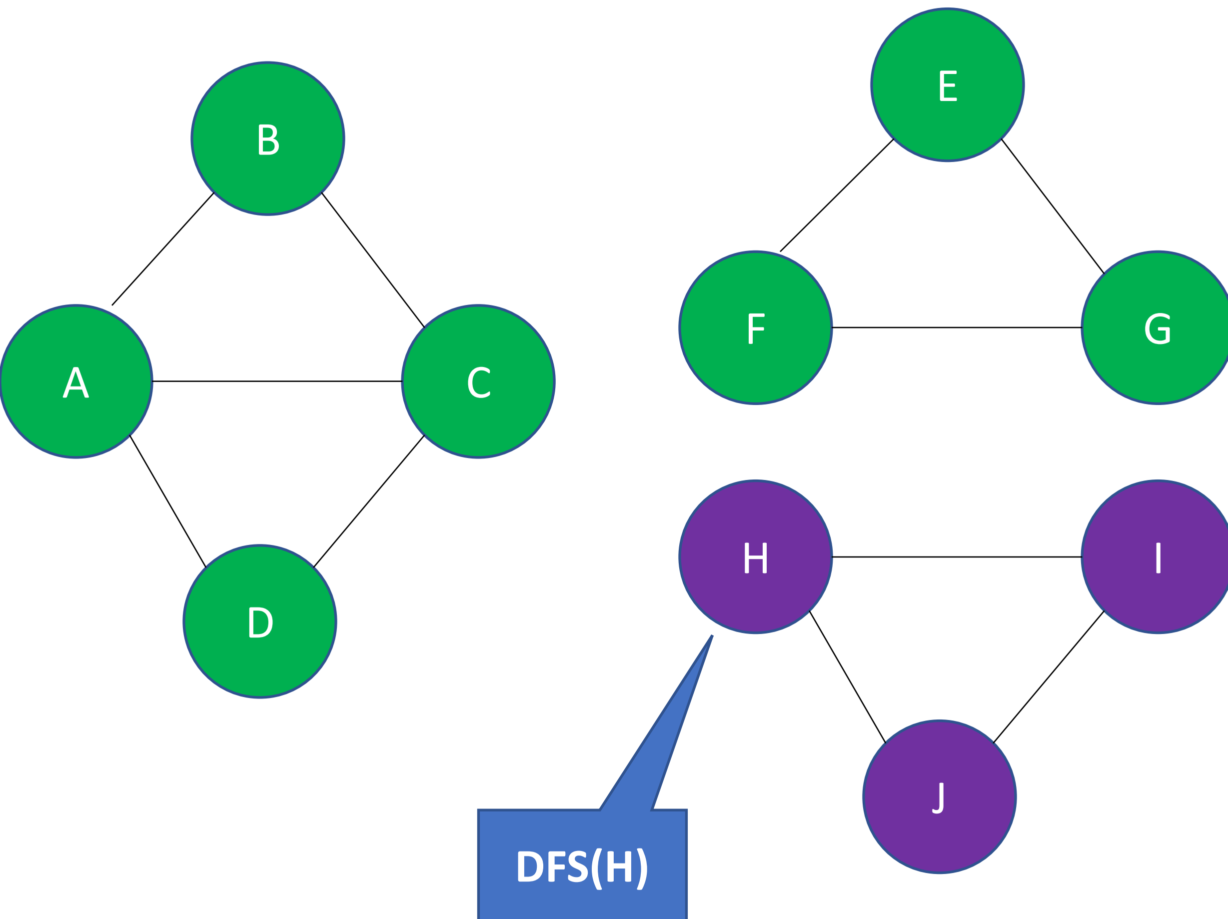


NCC=2

Visited	Vertex
1	A
1	B
1	C
1	D
1	E
1	F
1	G
0	H
0	I
0	J

DFS(H)

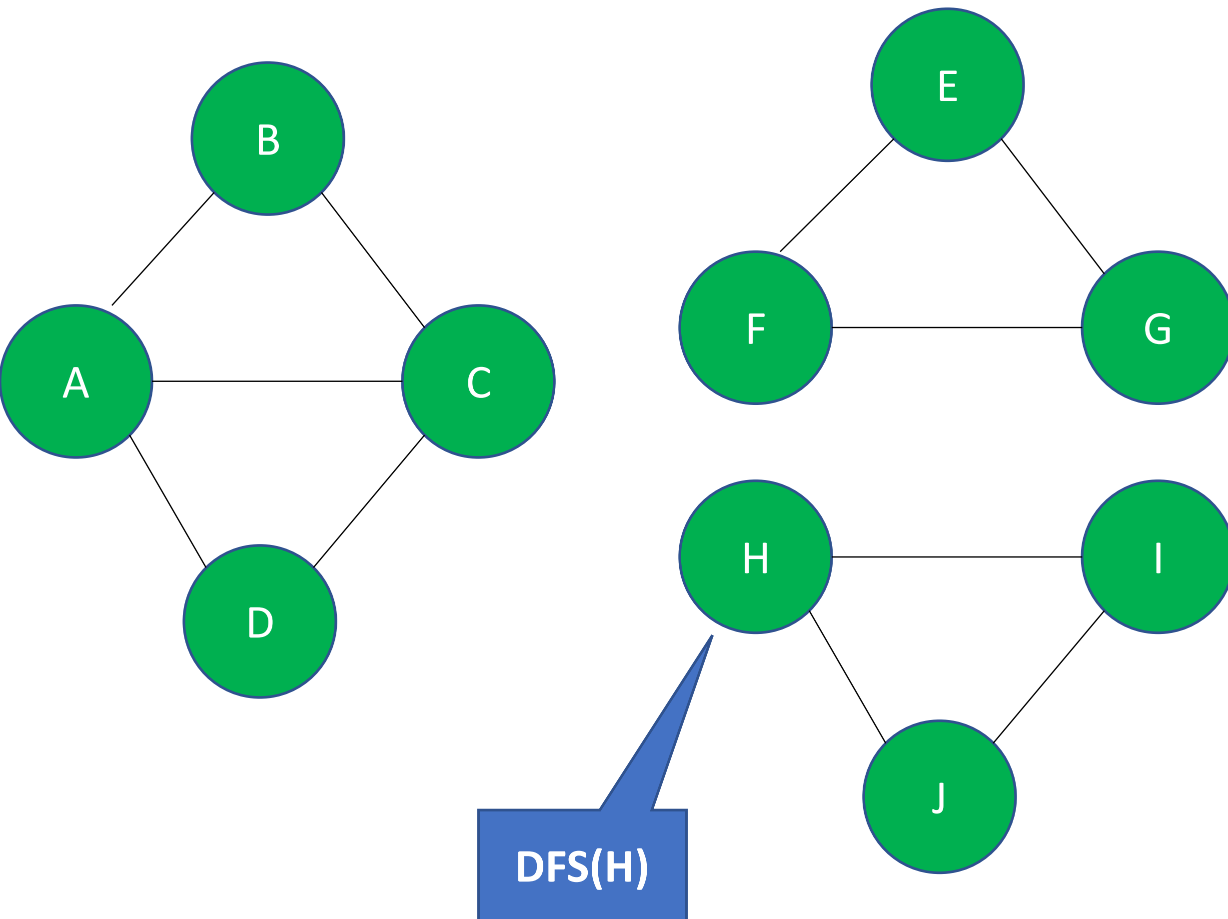
Connected Components using DFS



NCC=3

Visited	Vertex
1	A
1	B
1	C
1	D
1	E
1	F
1	G
0	H
0	I
0	J

Connected Components using DFS



NCC=3

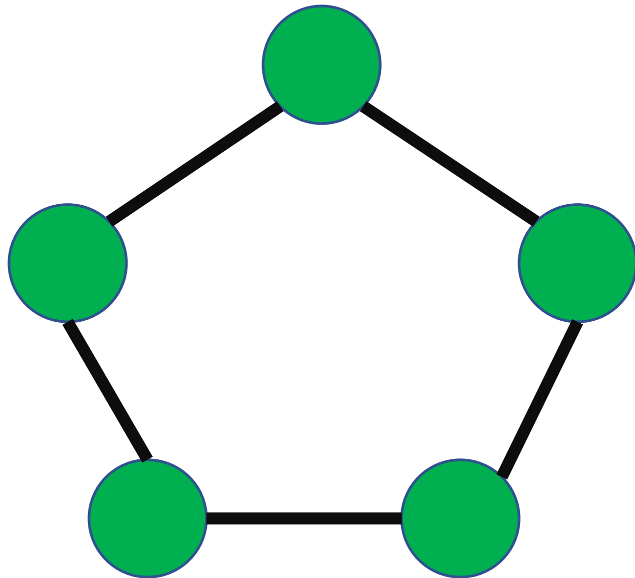
Visited	Vertex
1	A
1	B
1	C
1	D
1	E
1	F
1	G
1	H
1	I
1	J

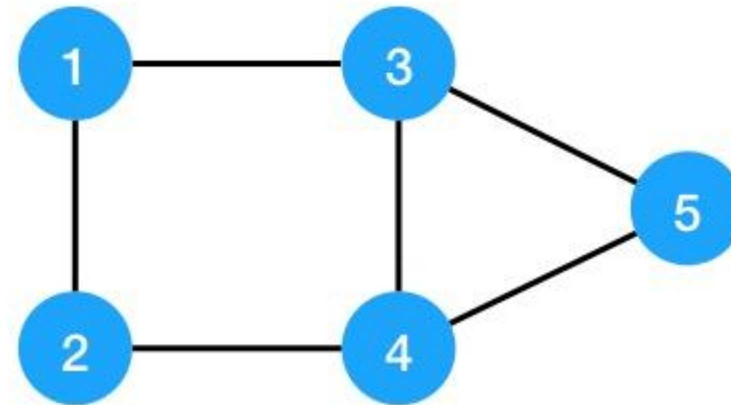
Connected Components using DFS

Time Complexity = $O(V + E)$ = $O(E)$

Bi-Connected Graph

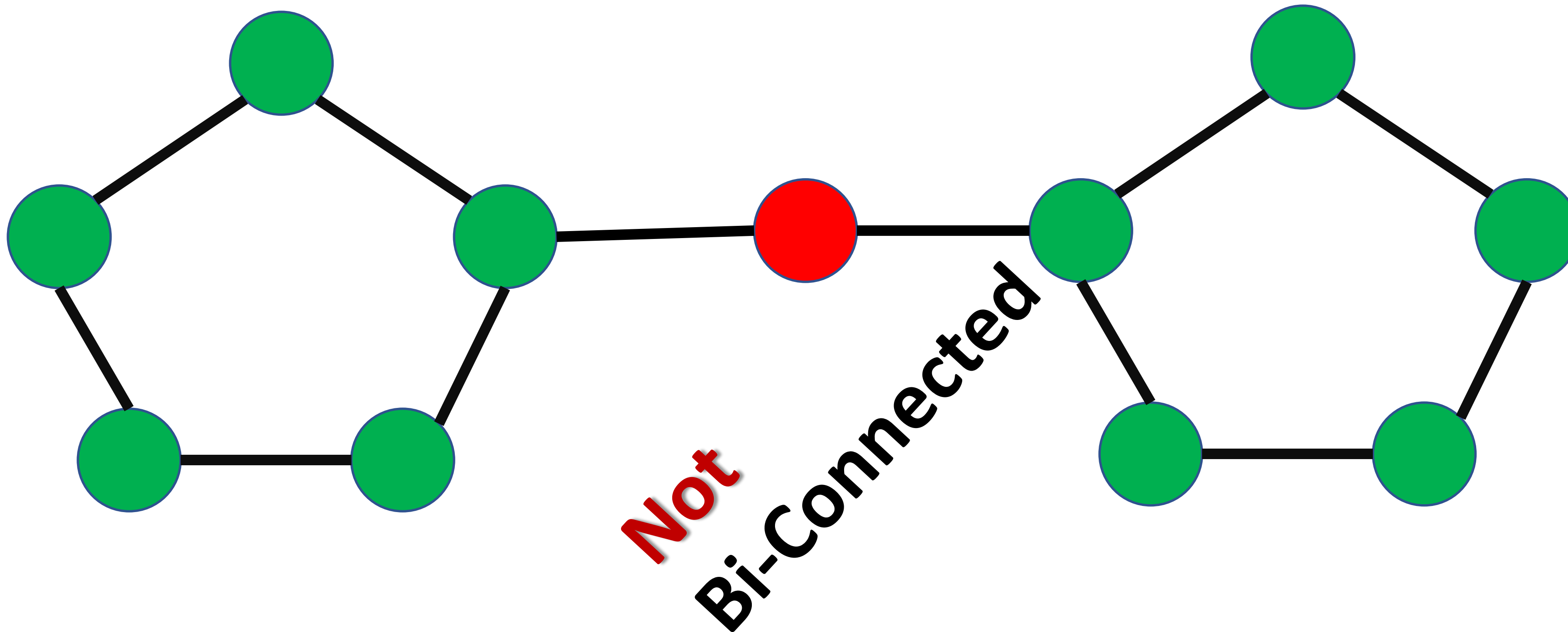
Def: A graph G is bi-connected if after removal of any vertex it remains connected





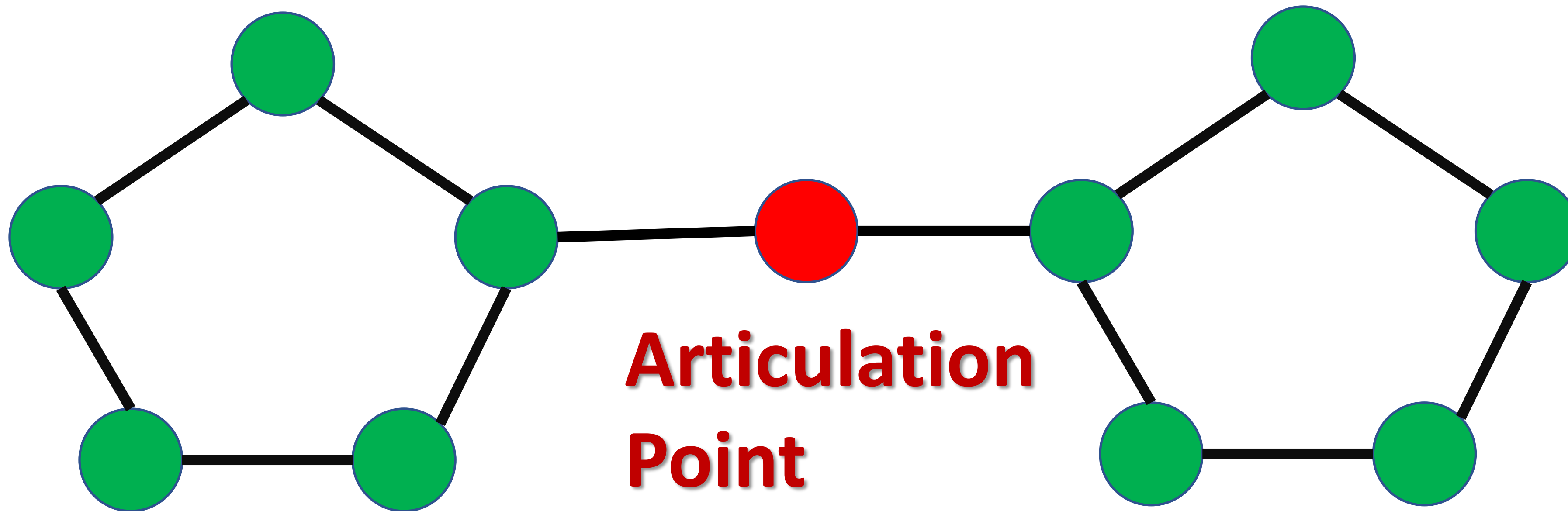
Bi-Connected Graph

Def: A graph G is bi-connected if after removal of any vertex it remains connected



Articulation Point or Cut Vertex

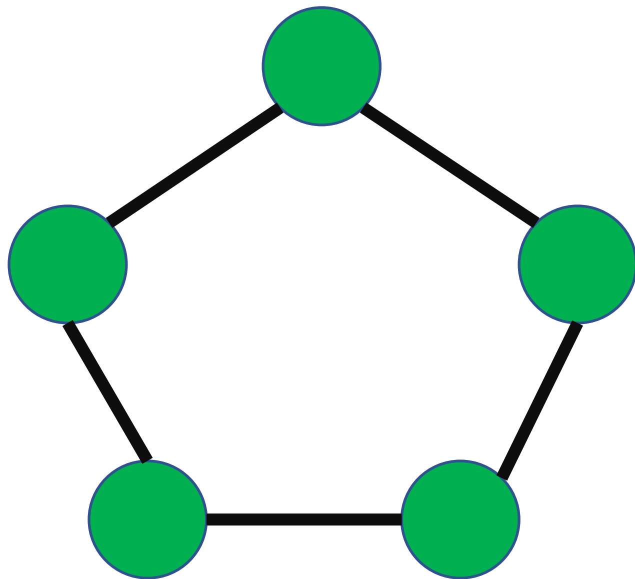
Def: A vertex in graph G whose removal disconnects the graph is known as an articulation point or cut vertex in G



Bi-Connected Graph

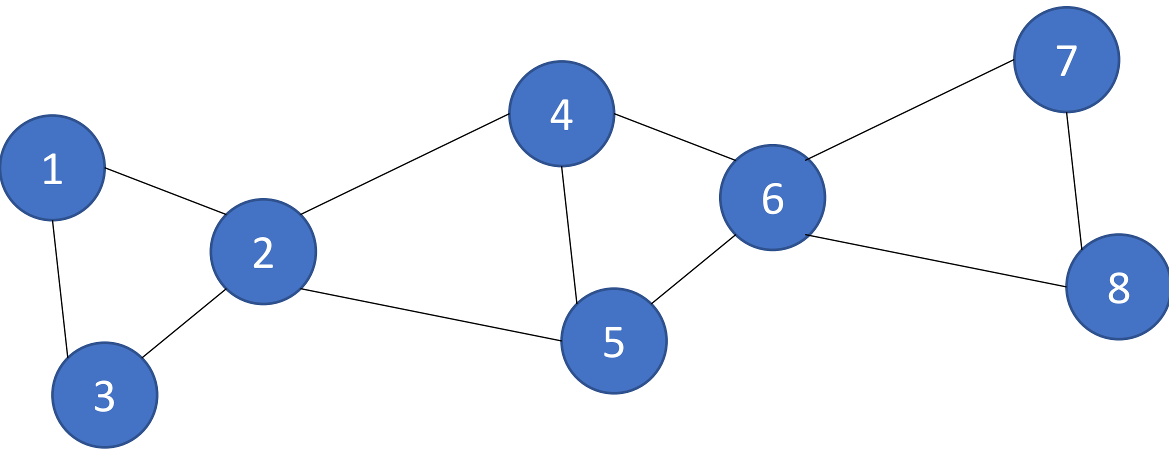
Observation

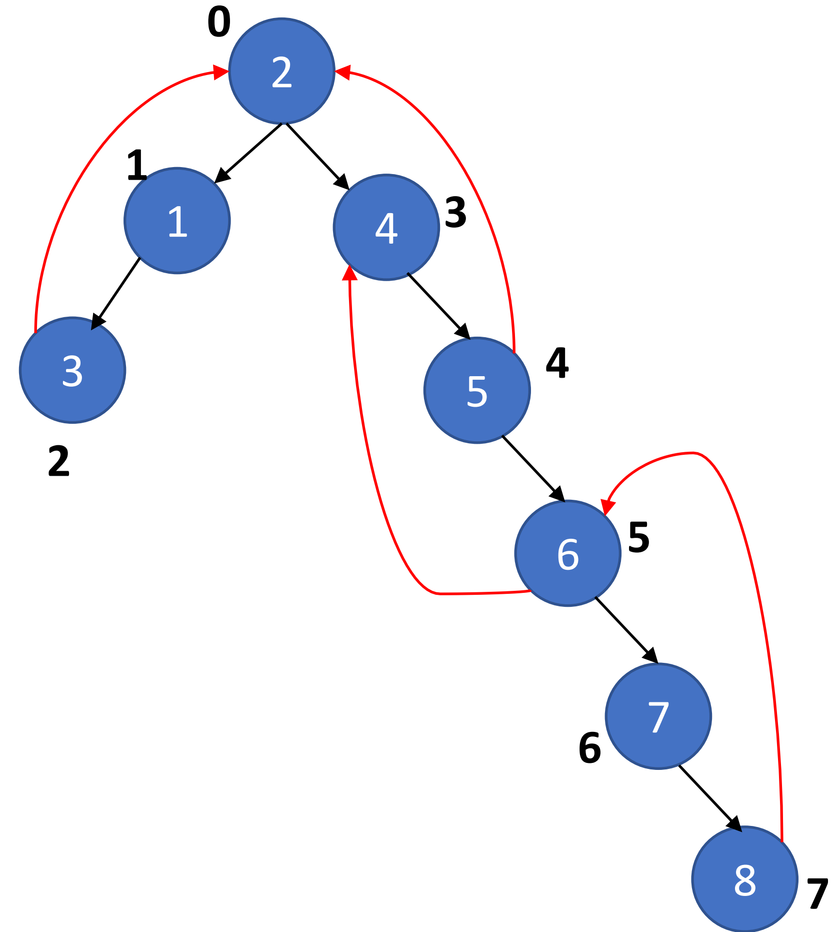
Graph G is Bi-Connected iff there is NO articulation point in a graph



No Articulation Point

Bi-Connected Graph

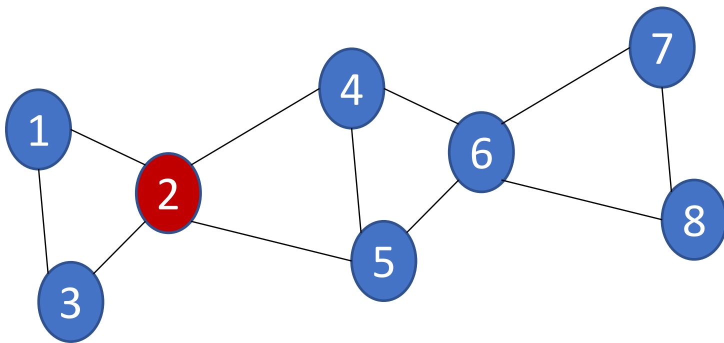


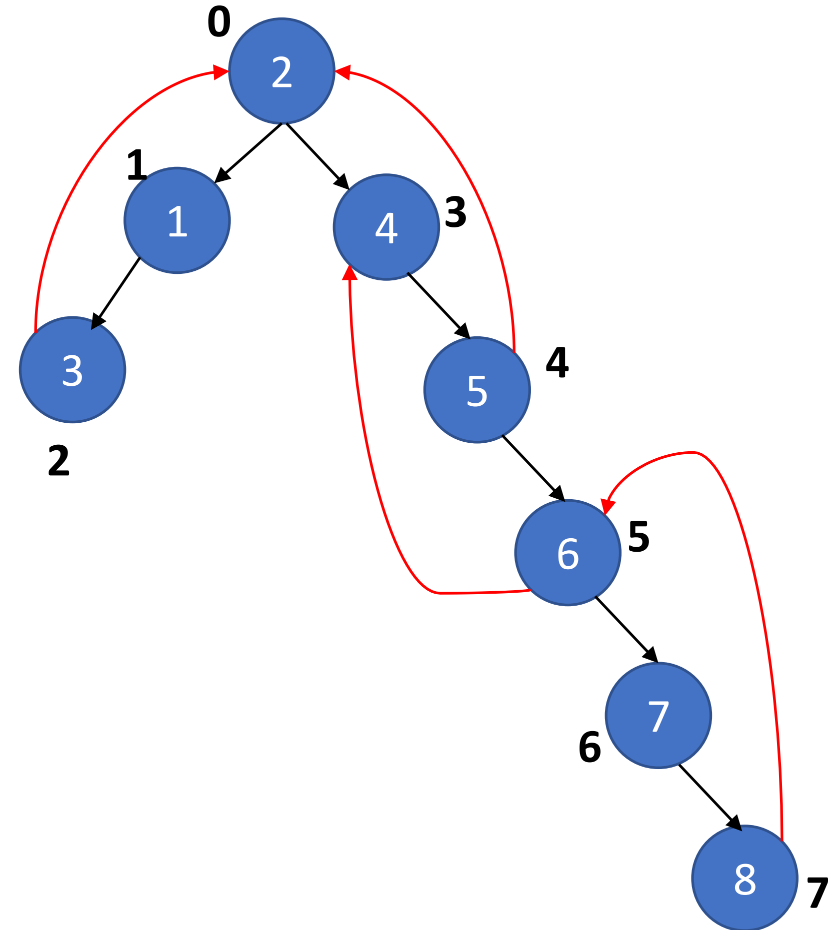


Bi-Connected Graph

Observation-1

The root of a DFS-tree is an articulation point if and only if it has at least two children.

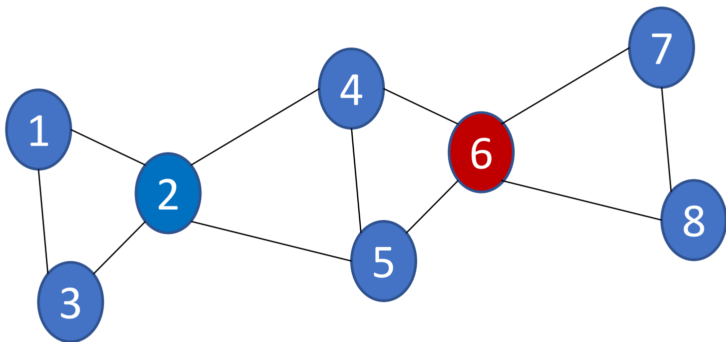


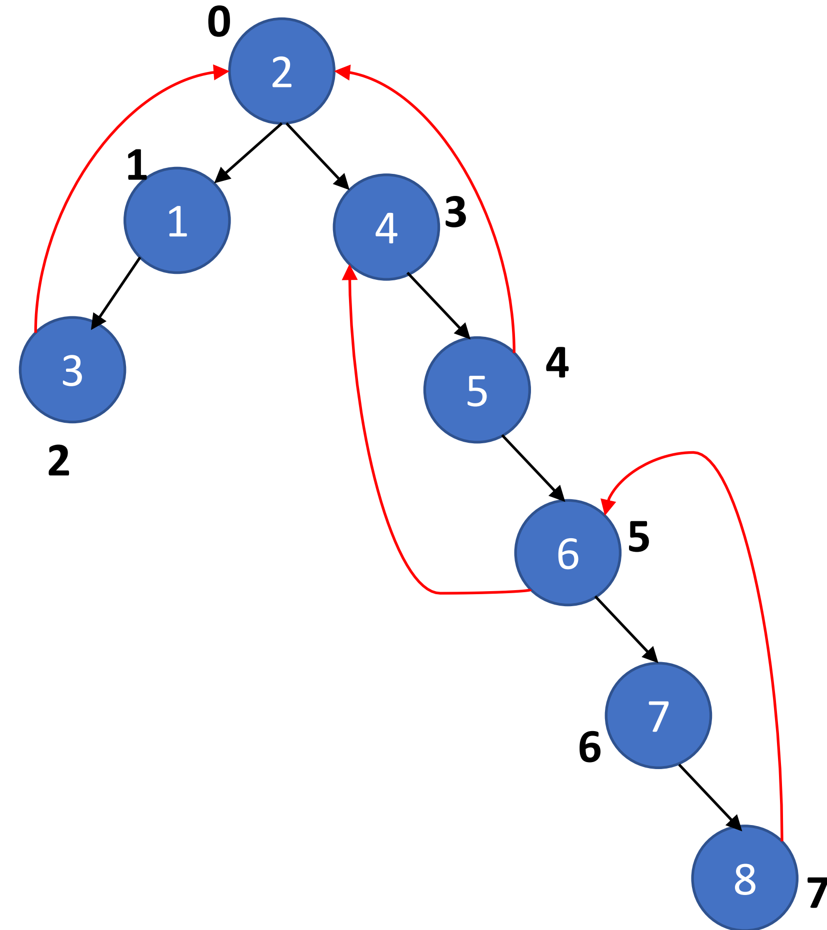


Bi-Connected Graph

Observation-2

A non-root vertex v of a DFS-tree is an articulation point of G if and only if it has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

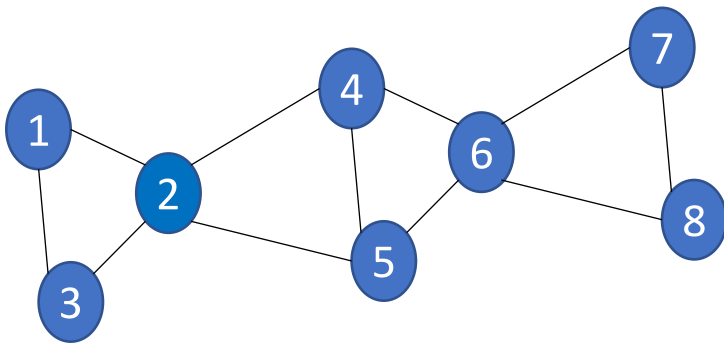


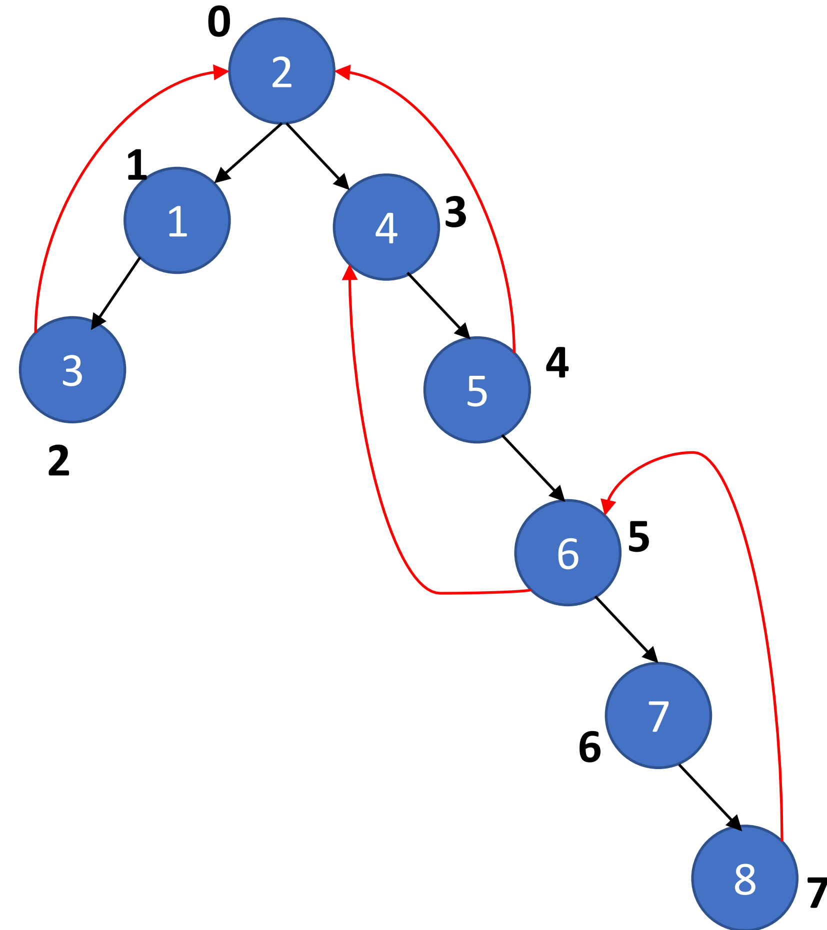


Bi-Connected Graph

Observation-2

Leaf of a DFS-tree are never articulation points



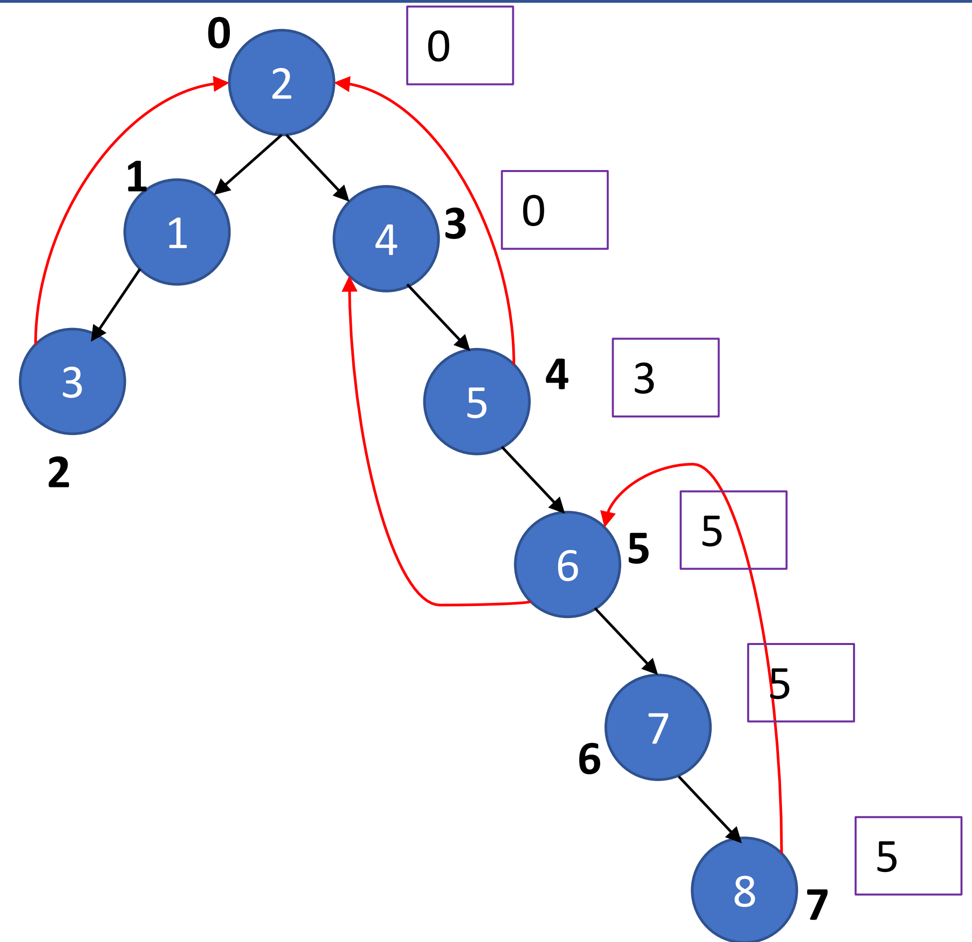


Bi-Connected Graph

Low Function

LOW(u) =

the highest ancestor (identified by its smallest discovery time) of u that can be reached from a descendant of u by using back-edges



Bi-Connected Graph

Observation

u is articulation point if it has a descendant v with LOW(v)>=u.d