CSCE 411

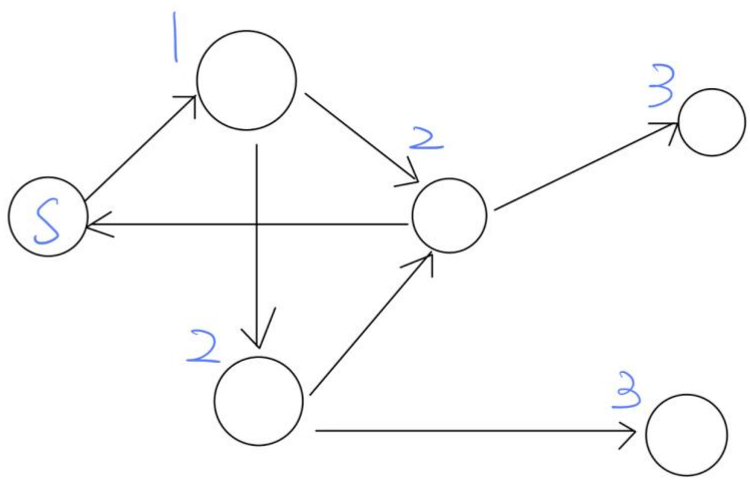
Lecturer: Nate Veldt

Fall 2025

---

1 Breadth First Search

(Unweighted) Shortest Path Problem: Given a graph $G = (V, E)$ and source node $s \in V$, find the shortest path from s to every other $v \in V$.



We will do this using the *breadth first search* algorithm.

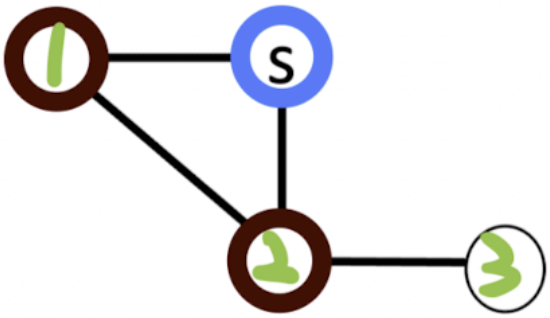
Attribute	Explanation	Initialization
u.status	tells us whether a node is explored, undiscovered, or discovered	s.status $=$ discovered u.status $=$ undiscovered, $\forall u \neq s$
u.dist	distance from s to u	s.dist $= 0$ u.dist $= \infty \ \forall u \neq s$
u.parent	predecessor or "discoverer" of node u	s.parent $= s.\pi = \varnothing$ u.parent $= u.\pi =$ NIL

We will also make use of a queue Q to keep track of the child nodes that were encountered but not yet explored.

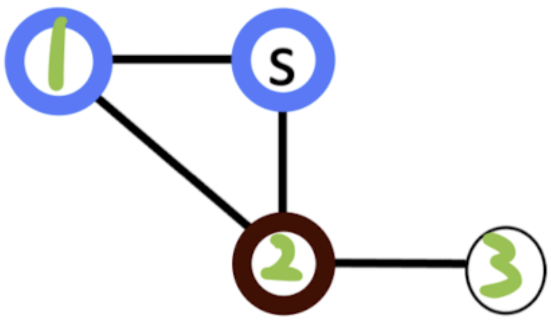
Basic Idea

- Mark s as *discovered* (blue)
- Iteratively *explore* discovered nodes to find new *discovered* nodes
- Continuously update distance from s for each discovered node

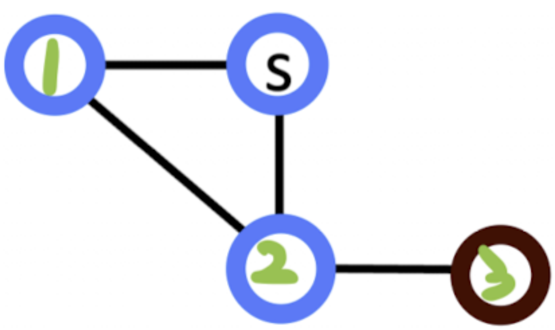
Example



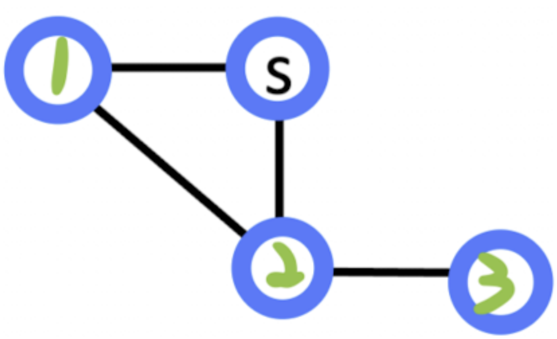
$1. \pi = s, Q = [1\ 2].$



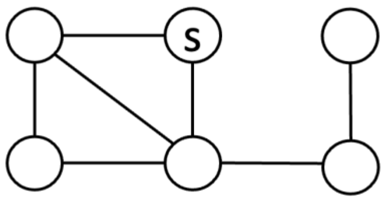
$2. \pi = s, Q = [2].$

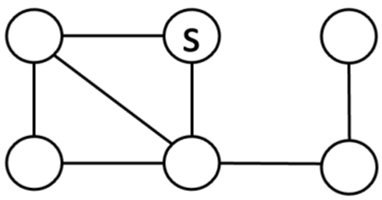


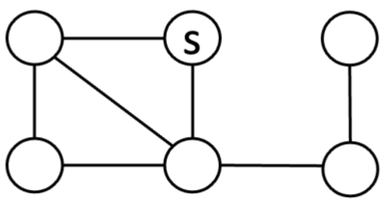
$3. \pi = 2, Q = [3].$

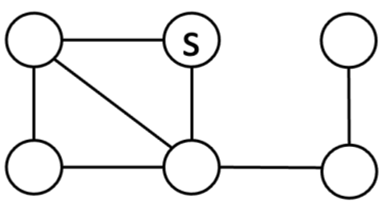


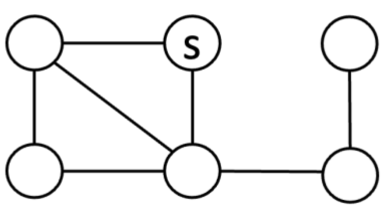
$Q = [\,].$

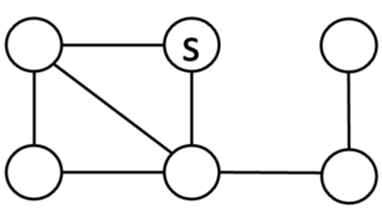

S

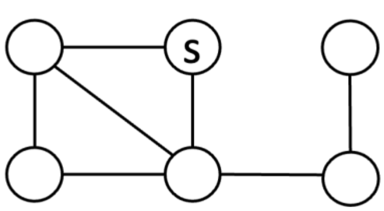

S

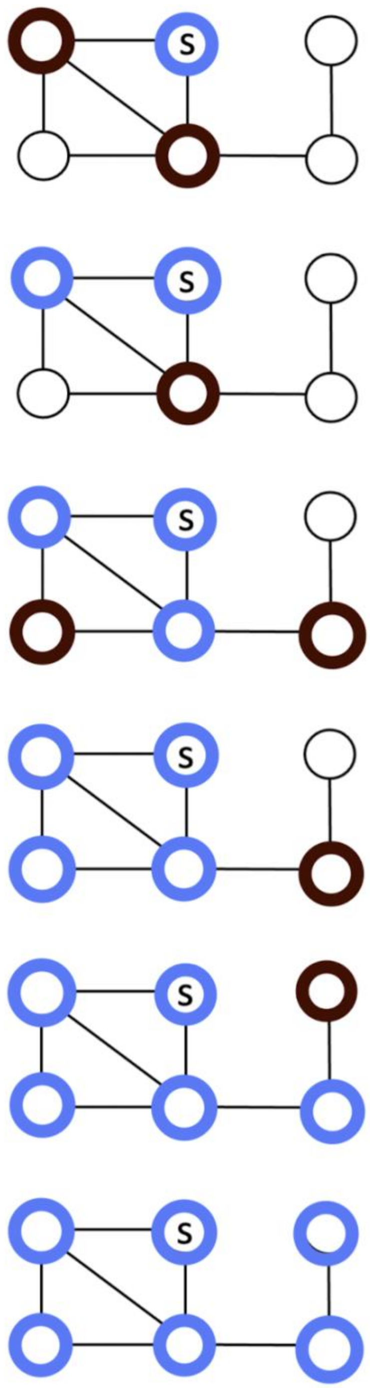

S


S


S


S


S


S
S
S
S
S
S

1.1 Shortest Paths and Breadth First Trees

Definition. Given a graph $G = (V, E)$, source node s, and a *parent* attribute for each node, a predecessor tree is a subgraph $\hat{G} = (\hat{V}, \hat{E})$ where

$$\hat{V} = \{s\} \cup \{v \in V: v.parent \neq NIL\}$$

$$\hat{E} = \{(v.parent, v): v \in \hat{V} - \{s\}\}$$

It is furthermore a breadth first search tree if it contains a unique simple path from s to v that is the shortest path from s to v in G.

Benefits of the BFS algorithm

- If G is undirected, it finds the connected component that

- It tells us the shortest path (and distance) from s

- It provides a breadth-first tree

1.2 Code and Runtime Analysis

BFS(G,s)

```
for v ∈ V do
    v.parent = NIL
    v.dist = ∞
    v.status = U
end for
s.dist = 0
s.status = D
Initialize Q
Enqueue(s)
while |Q| > 0 do
    u = Dequeue(Q)
    N(u) = Adj[u]
    for v in N(u) do
        if v.status == U then
            v.status = D
            v.parent = u
            v.dist = u.dist + 1
            Enqueue(v)
        end if
    end for
    u.status = E
end while
```

- We assume G is undirected and stored as an adjacency list.
- Initializing attributes takes $O(n)$ time
- Each node u only enters Q once, and entering/leaving Q takes $O(1)$ time
- When we *explore* u, we discover up to $d_u = |\mathrm{Adj}[u]|$

Using aggregate analysis, what is the overall runtime of this method?

(A) $O(n)$

(B) $\boldsymbol{O(m+n)}$

(C) $O(n^2)$

(D) $O(mn)$

Solution

The initialization step (from the loop over V to the part that sets v.dist and v.status) takes $O(|V|) = O(n)$ time.

Each node u enters Q only once, and entering or leaving Q takes $O(1)$ time. Therefore, the total time is $O(\sum_{v\in V} 1) = O(n)$.

When we *explore* u, we discover up to $d_u = |\mathrm{Adj}[u]|$ neighbors, so this step takes

$$O\left(\sum_{v\in V} d_v\right) = O(2m) = O(m)$$

time.

Therefore, the total running time is $O(n+n+m) = O(n+m)$.

2 Depth First Search: Background and Motivating problems

Recall that a *breadth-first* search explores nodes that are k steps away from node s before exploring any nodes that are $k+1$ steps away.

A *depth-first search* instead explores the *most recently discovered vertex* before backtracking and exploring other previously discovered nodes.

Roughly speaking, this is accomplished by <u>replacing the queue by a stack</u>.

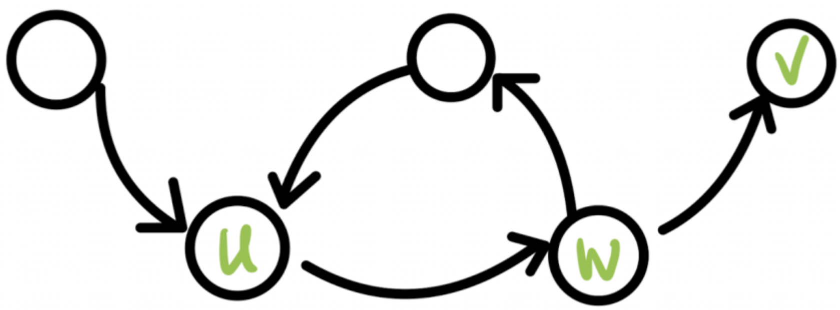
Depth first search is used in several applications for analyzing directed graphs. We will take a closer look at these applications before exploring how to solve them using DFS.

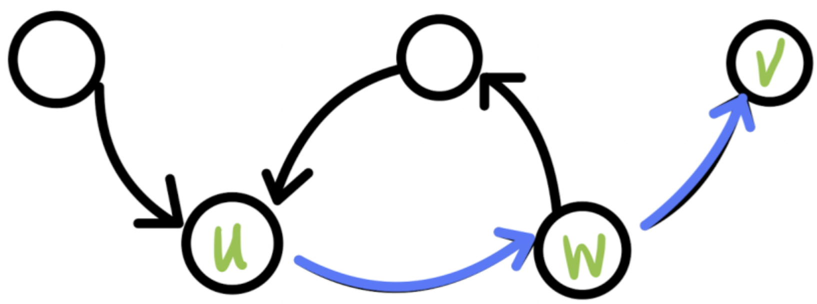
Directed graph reminders

$(i, j) \in E$ means a directed edge from i to j and (j, i) (the directed edge from j to i) may not exist.

2.1 Reachability and Connected Components

Reachability. Given a graph $G = (V, E)$ and node set $S \subseteq V$, node $v \in S$ is *reachable* from node $u \in S$ if there is a directed path from u to v in S.



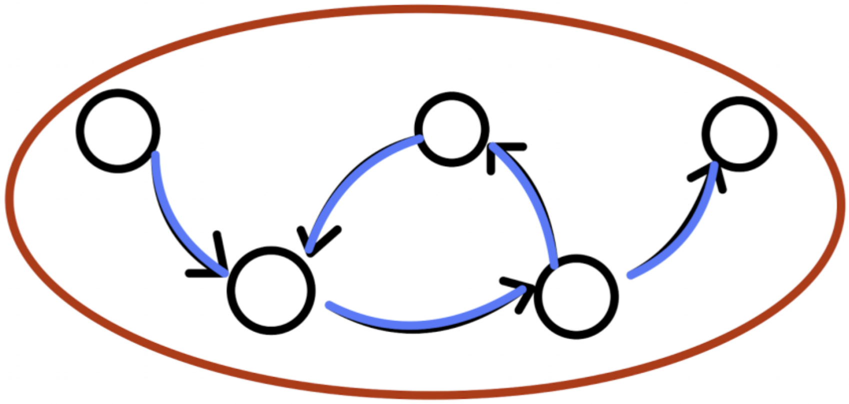


v is *reachable* from u since there is a directed path $(u, w), (w, v)$.

Connected components. For an undirected graph $G = (V, E)$ a connected component is a maximal subgraph in which every node in is reachable from every other node in S

Weakly Connected components If $G = (V, E)$ is directed, a *weakly connected component* is a

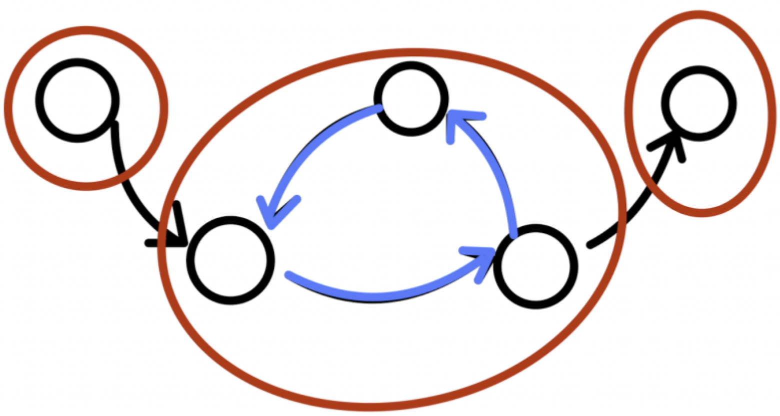
<u>connected component in the graph obtained by ignoring edge directions</u>



There is 1 *weakly connected component* in this graph.

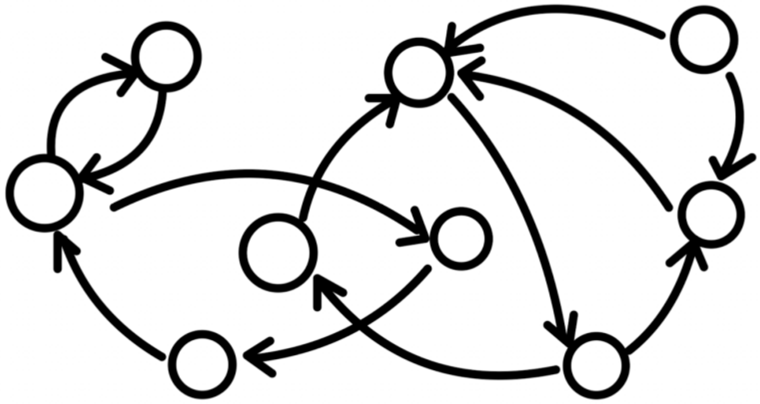
Strongly Connected components If $G = (V, E)$ is directed, a *strongly connected component* is subgraph $S \subseteq V$ in which there is <u>a directed path from each</u> $u \in S$ <u>to every other</u> $v \in S$<u>.</u>

<u>I.e. every node in</u> S <u>is reachable from every other node in</u> S<u>.</u>



There are 3 *strongly connected component* in this graph.

Question 1. *How many weakly connected components and strongly connected components are there in the following graph, respectively?*



(A) 1 and 3

(B) 1 and 2

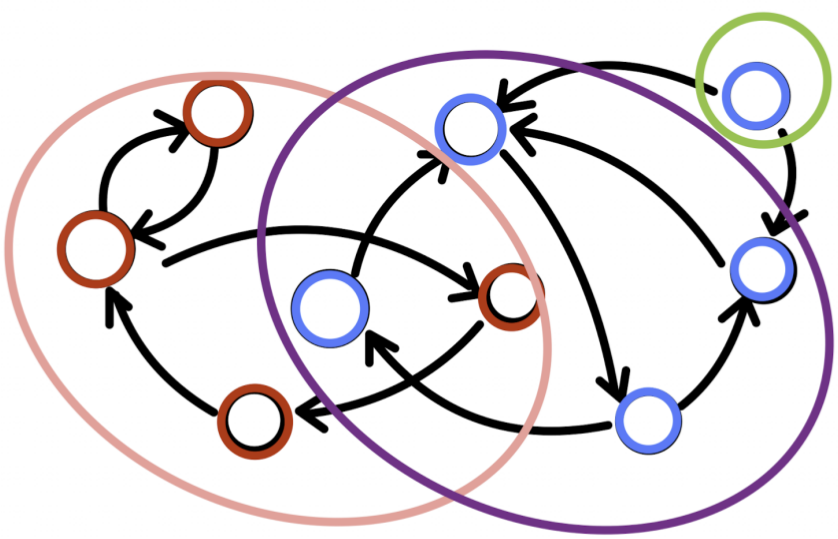
(C) 0 and 1

(D) 2 and 3

(E) 2 and 2

(F) something else

Solution

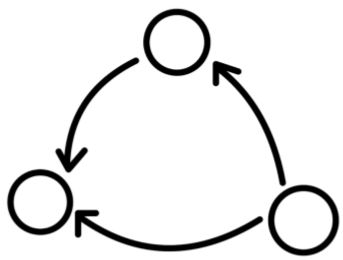


2.2 Directed Acyclic Graphs

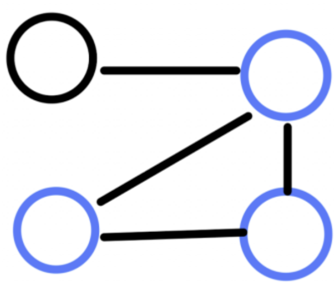
A *cycle* in a directed graph is a directed path <u>that starts and ends at the same node</u>

A *Directed acyclic* graph is a directed graph that <u>has no cycles</u>.

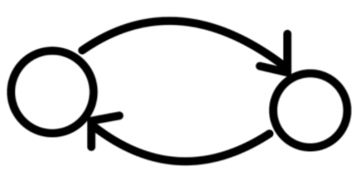
Examples



cycle in directed graph.



cycle in undirected graph (blue nodes).



cycle of size 2 in directed graph.

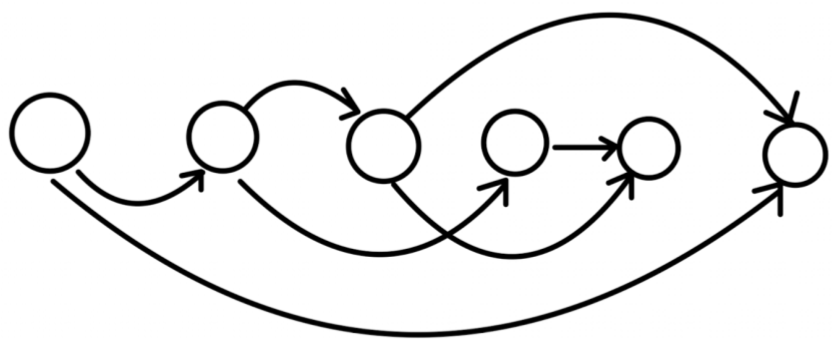
no cycle.

Note: in undirected graph, there is no such thing as a cycle on 2 nodes.

Note: cycle cannot reuse edges or nodes.

2.3 Topological Sorting

A topologically ordering of a directed acyclic graph $G = (V, E)$ is an ordering of nodes so that: if $(u, v) \in E$ then u comes before v in the ordering.



3 Depth First Search Algorithm

Unlike in a BFS, a depth-first search (DFS):

- Explores the *most recently discovered vertex* before backtracking and exploring other previously discovered vertices
- All nodes in the graph are explored (rather than just a DFS for a single node s)
- We keep track of a global *time*, and each node is associated with two timestamps for when it is *discovered* and *explored*.

Each node $u \in V$ is associated with the following attributes

Attribute	**Explanation**	**Initialization**
u.status	tells us whether a node has been *undiscovered*, *discovered*, and *explored*	u.status $= u$
u.D	timestamp when u is first discovered	NIL
u.F	timestamp when u is finished being explored	NIL
u.parent	predecessor/“discoverer" of u	NIL

DFS(G)

for $v \in V$ **do**

  v.parent $= NIL$

  v.status $=$ U

end for

time $= 0$

for $u \in V$ **do**

  if u.status $== U$ **then**

    DFS-VISIT(G, u)

    end if

end for

DFS-VISIT(G, u)

time $=$ time $+ 1$

$u.D =$ time

u.status $= D$

for $v \in$ Adj$[u]$ **do**

  if v.status $== U$ **then**

    v.parent $= u$

    DFS-VISIT(G, v)

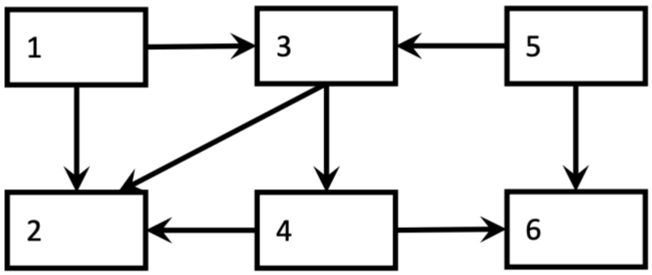
  end if

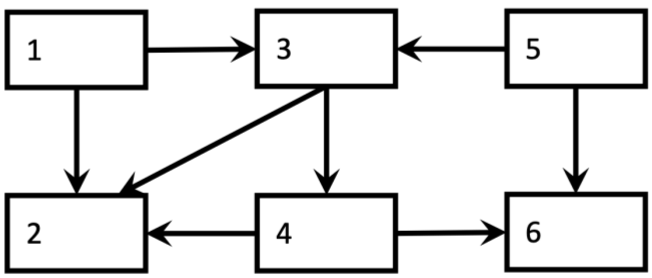
end for

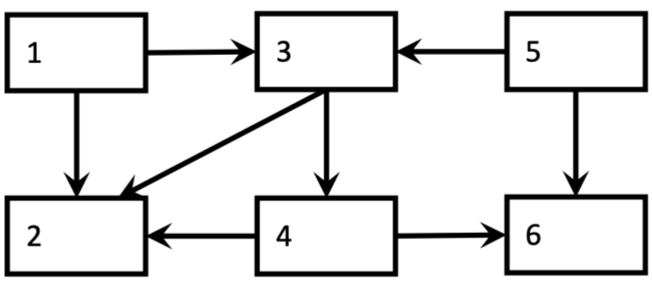
u.status $=$ E

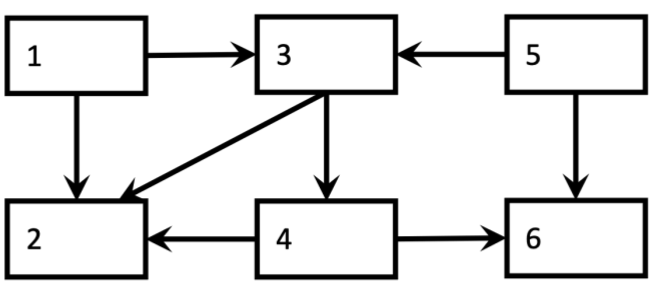
time $=$ time $+ 1$

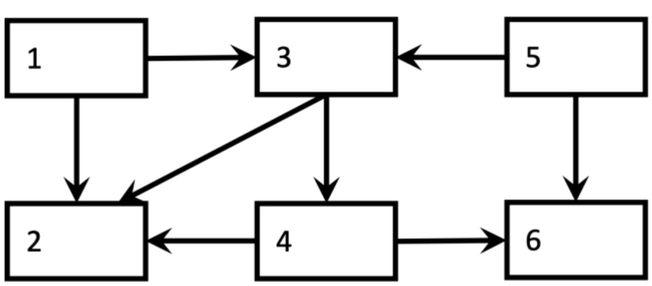
$u.F =$ time


1
3
5
2
4
6


1
3
5
2
4
6

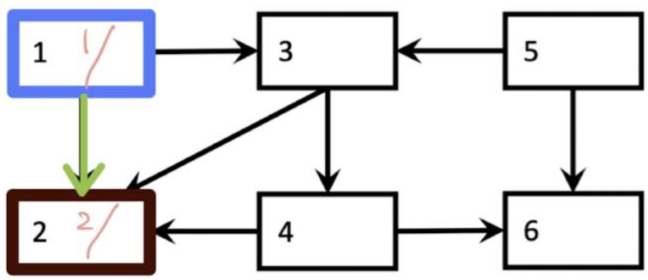

1
3
5
2
4
6

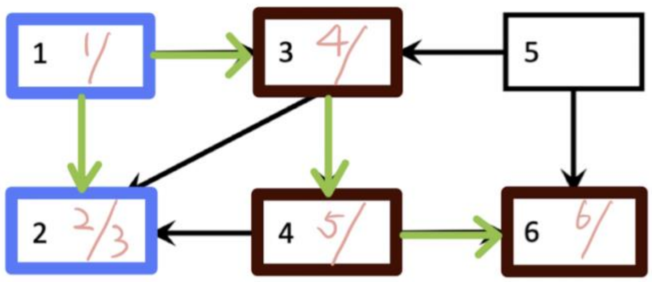

1
3
5
2
4
6

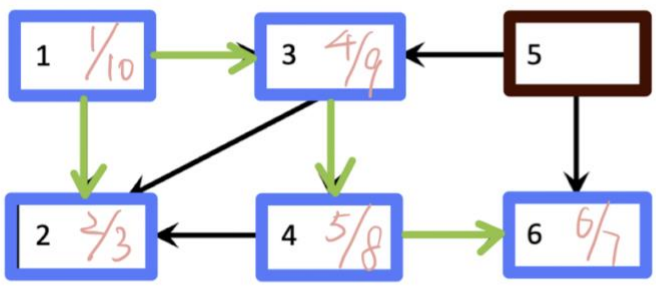

1
3
5
2
4
6

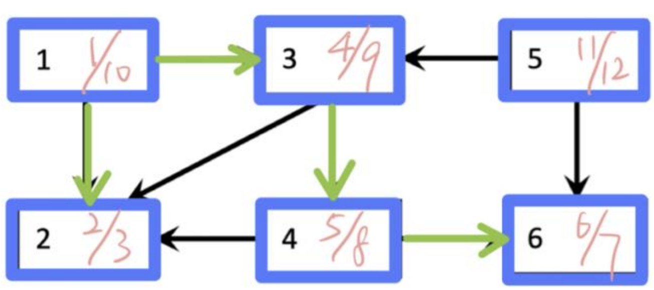
For the notation $v: 1/10$, 1 is $v.D$ and 10 is $v.F$.

We use green arrows from v to u to indicate u.parent $= v$.









1	***1***	2	3	4	5	6	7	8	9	***10***	11	12
2	1	***2***	***3***	4	5	6	7	8	9	10	11	12
3	1	2	3	***4***	5	6	7	8	***9***	10	11	12
4	1	2	3	4	***5***	6	7	***8***	9	10	11	12
5	1	2	3	4	5	6	7	8	9	10	***11***	***12***
6	1	2	3	4	5	***6***	***7***	8	9	10	11	12

3.1 Runtime Analysis

Question 2. *What is the runtime of a depth first search, assuming that we store the graph in an adjacency list, and assuming that* $|E| = \Omega(|V|)$*?*

(A) $O(|V|)$

(B) $\boldsymbol{O}(|\boldsymbol{E}|)$

(C) $O(|V| \times |E|)$

(D) $O(|V|^2)$

(E) $O(|E|^2)$

Solution

The runtime is $O(|E| + |V|)$. Since $|E| = \Omega(|V|)$, it will be $O(|E|)$.

3.2 Properties of DFS

Theorem. In any depth-first search of a graph $G = (V, E)$, for any pair of vertices u and v, exactly one of the following conditions holds:

- $[u.D, u.F]$ and $[v.D, v.F]$ are disjoint; neither u nor v is a descendant of the other.
- $[v.D, v.F]$ contains $[u.D, u.F]$ and u is a descendant of v
- $[u.D, u.F]$ contains $[v.D, v.F]$ and v is a descendant of u

We will not prove this, but we'll give a quick illustration

Corollary Descendant Property

v is a descendant of u if and only if $u.D \leq v.D \leq v.F \leq u.F$

3.3 Classification of Edges

Given a graph $G = (V, E)$ performing a DFS on G produces a graph $\hat{G} = (V, \hat{E})$ where

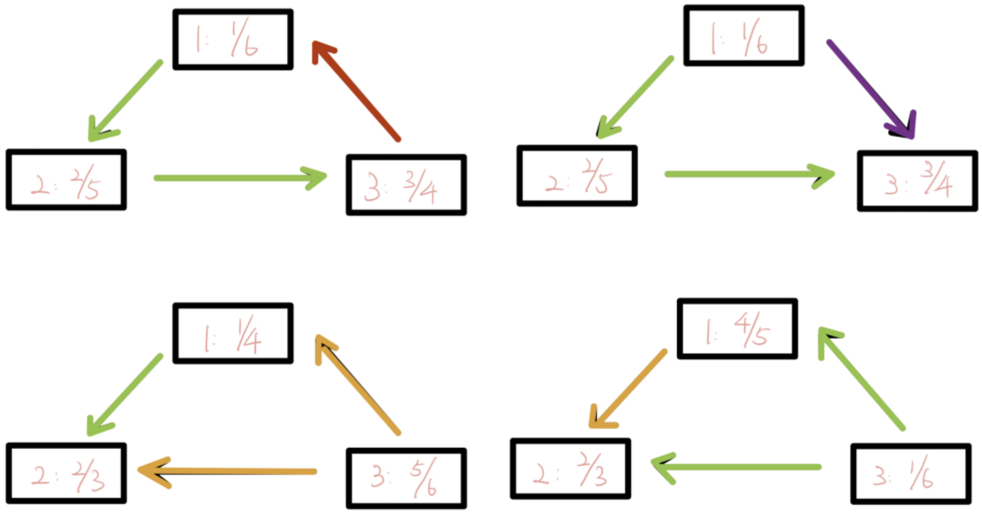
$$\hat{E} = \{(u.\text{parent}, u) : u \in V \text{ and } u.\text{parent} \neq NIL\}$$

This is called a *depth-first* forest of G.

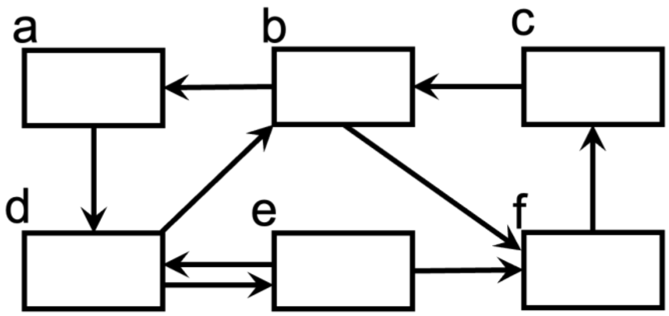
Given any edge $(u, v) \in E$, we can classify it based on the status of node v when we are performing the DFS:

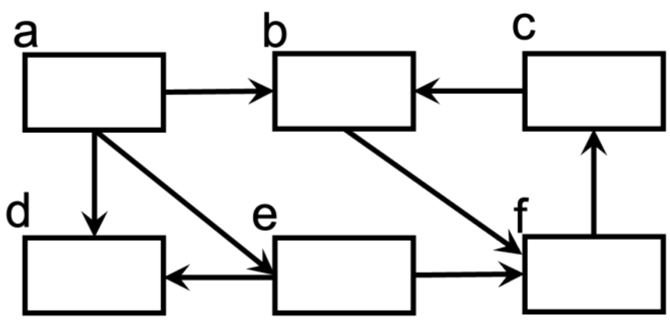
Edge	**Explanation**	**How to tell when exploring $(\boldsymbol{u}, \boldsymbol{v})$?**
Tree edge	edge in $\hat{E}$	v.status $== U$
Back edge	connects u to ancestor v	v.status $== D$
Forward edge	connects vertex u to descendant v	v.status $== E$ *and* $u.D < v.D$
Cross edge	either (a) connects two different trees or (b) crosses between siblings/cousins in same tree	v.status $== E$ *and* $u.D > v.D$

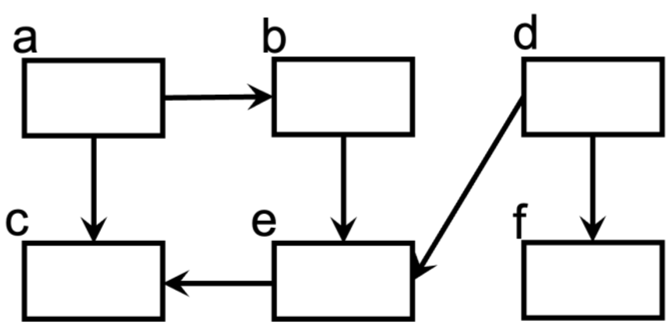
Examples

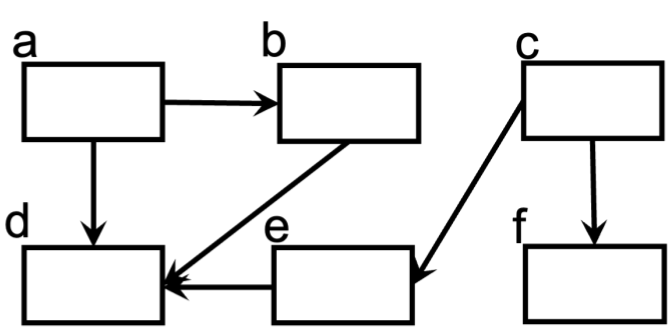


4 Practice









Question 3. *How many of the above graphs were directed acyclic graphs?*

(A) 1

(B) 2

(C) 3

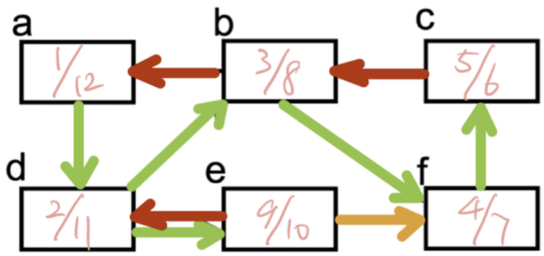
(D) 4

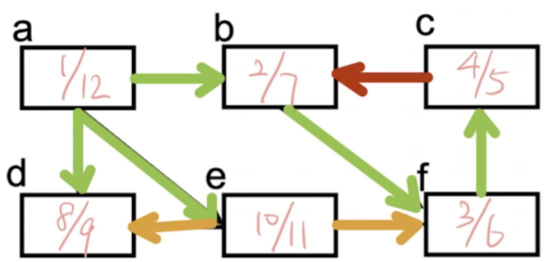
(E) none of them

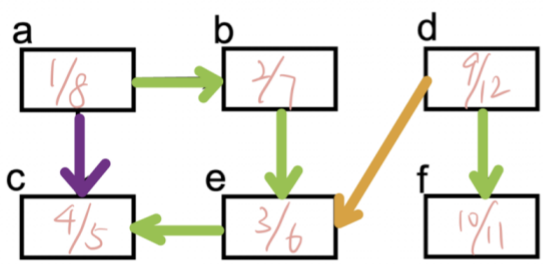
Solution

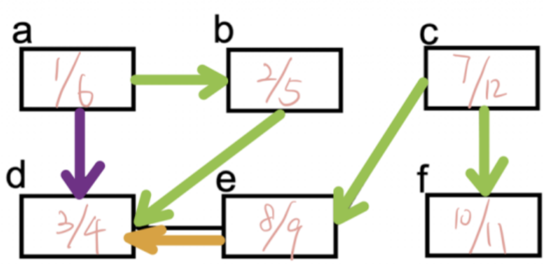
The first graph is not a directed acyclic graph since there is a cycle (a, d, b).

The second graph is not a directed acyclic graph since there is a cycle (b, f, c).


a
1/12
b
3/8
c
5/6
d
2/11
e
9/10
f
4/7


a
1/12
b
2/7
c
4/5
d
8/9
e
10/11
f
3/6


a
1/8
b
2/7
d
9/12
c
4/5
e
3/6
f
10/11


a
1/6
b
2/5
c
7/12
d
3/4
e
8/9
f
10/11