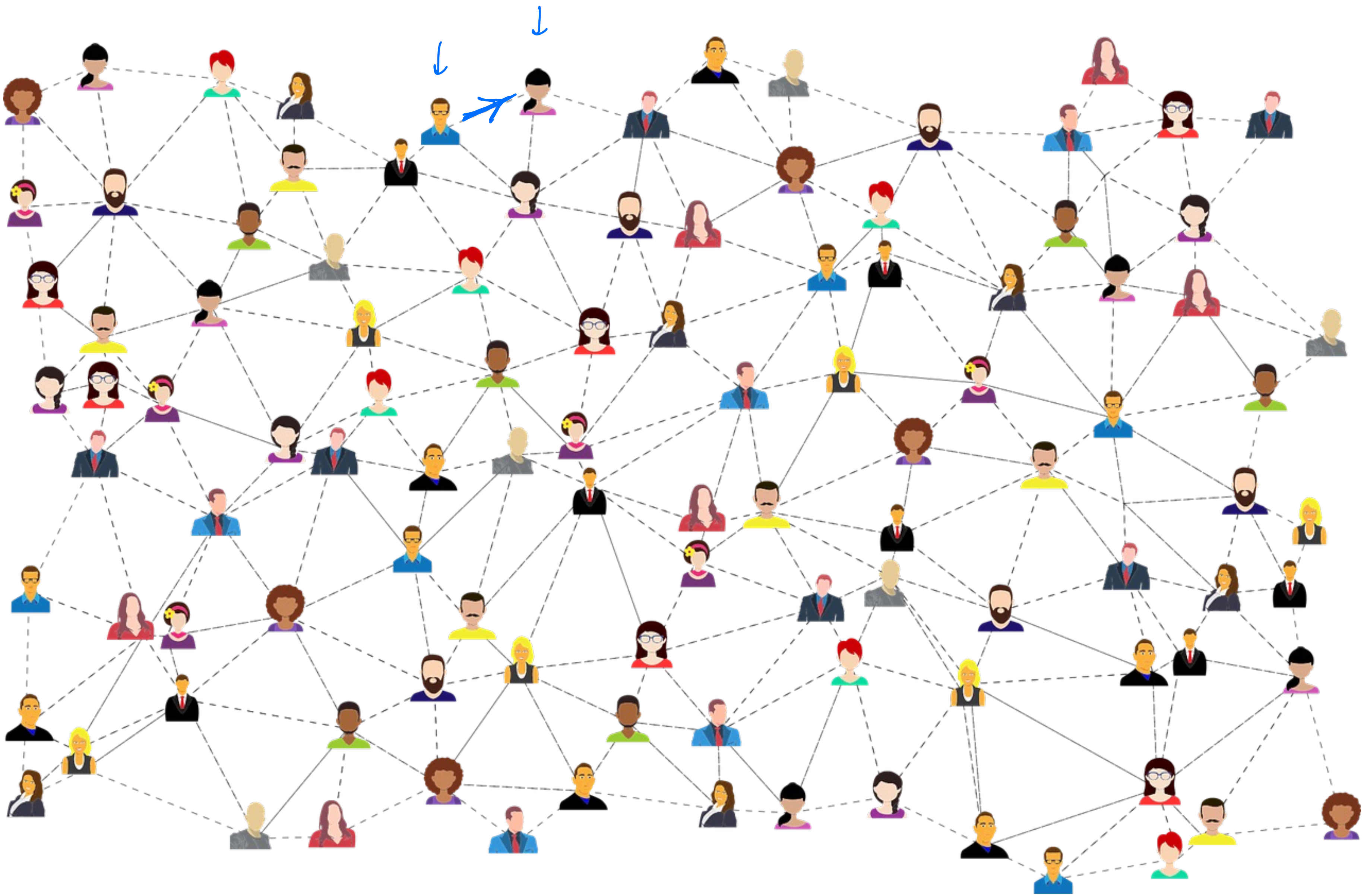
Graphs and Graph Traversals

a. Introduction to Graphs

Graphs: Key Definitions

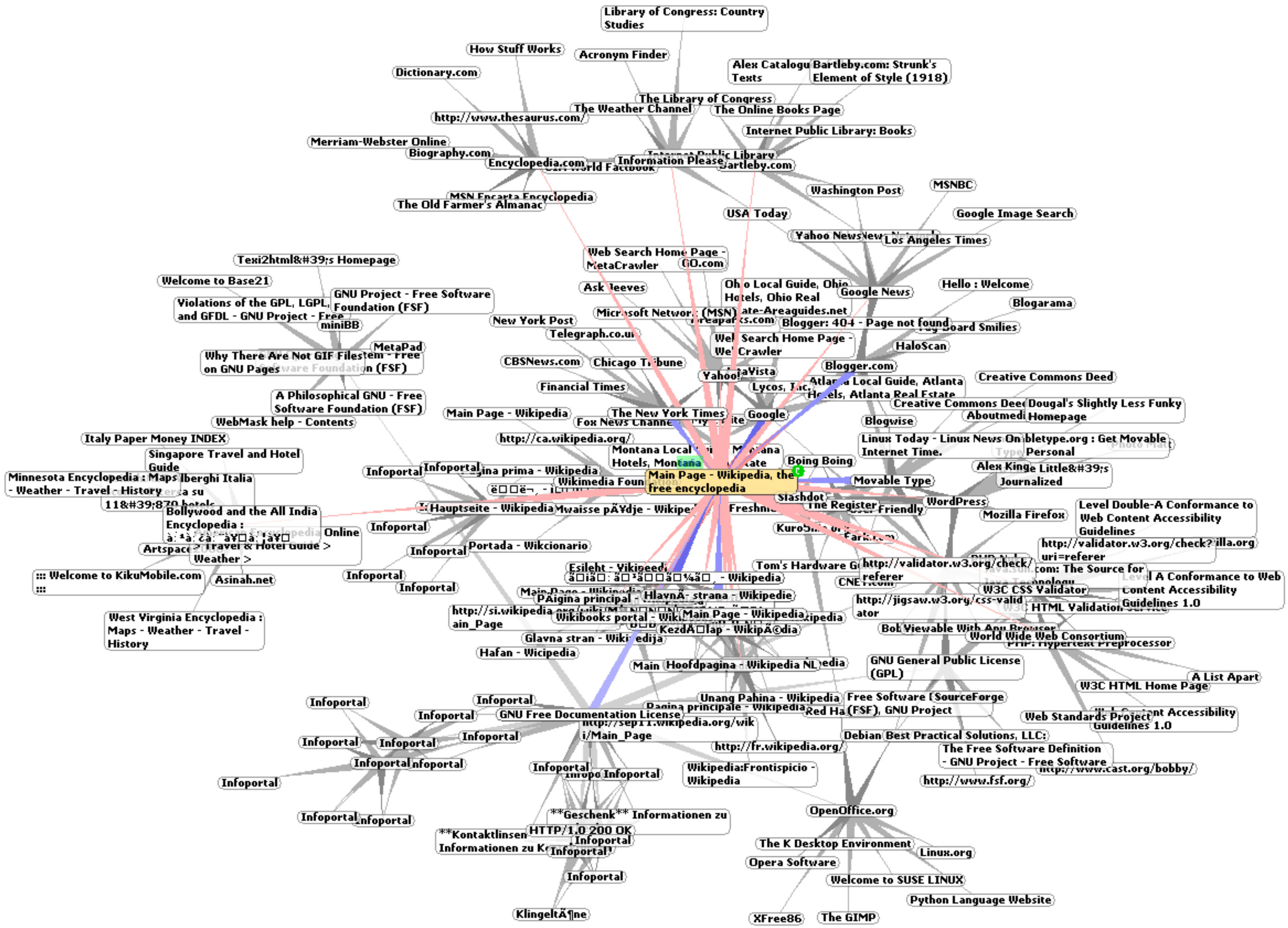
- **Vertices:** can be used to represent people, items, cities,…
- **Edges:** represent connections, roads, relations between pairs of vertices.
  - Can be directed or undirected.

Example: Social Relations



Example: Public Transport

Example: World Wide Web



Graphs: Key Definitions

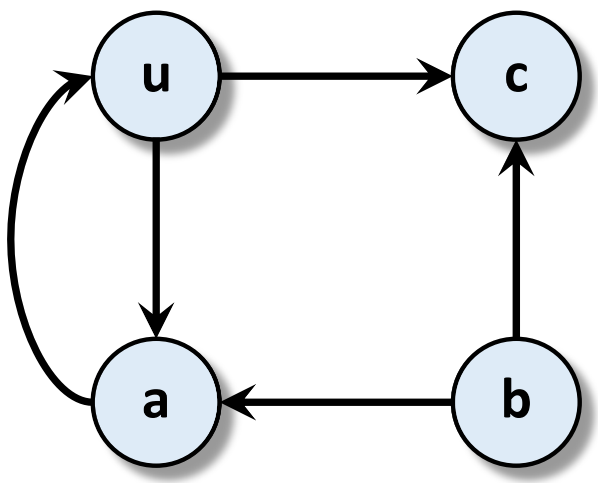
vertex Set

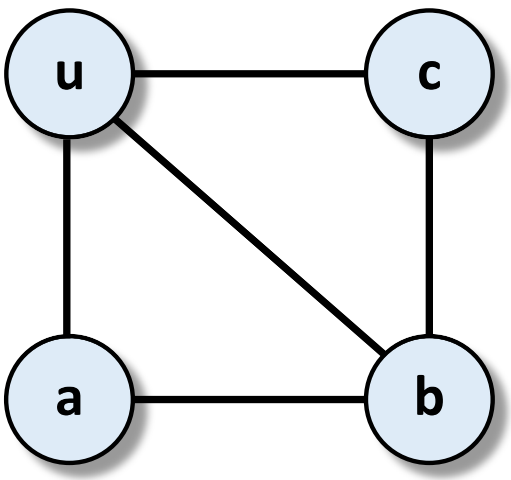
Edge Set

- We represent graphs by $G = (V, E)$
  - V is the set of nodes/vertices
  - $E \subseteq V \times V$ is the set of edges

$u \to v$

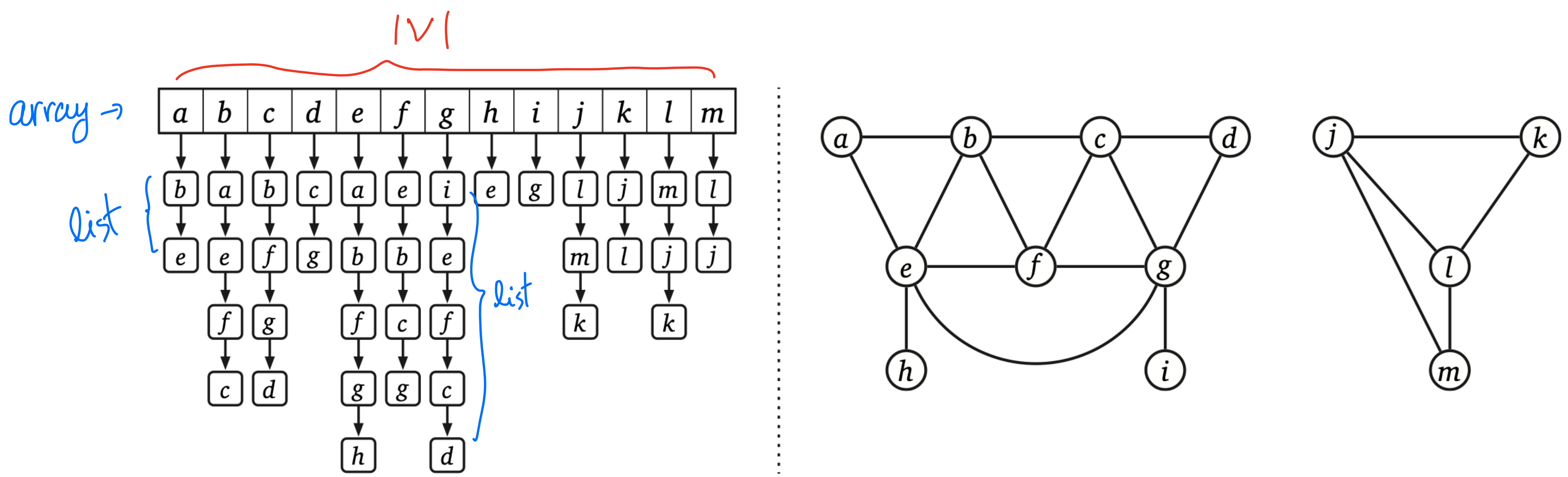
- **Directed**: Edges are ordered pairs $e = (u, v)$ "from u to v"
- **Undirected:** Edges are unordered $e = (u, v)$ "between u and v"





Data Structures: Adjacency List

- An adjacency list is an array of lists, each containing the neighbors of one of the vertices (or the out-neighbors if the graph is directed)



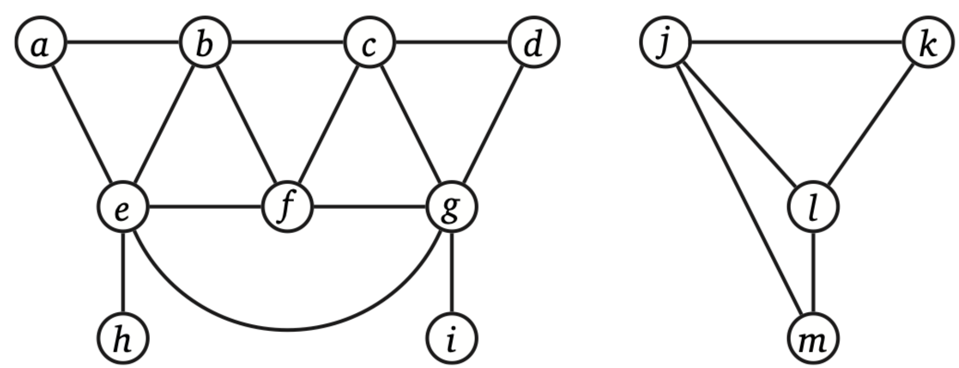
Data Structures: Adjacency Matrix

- The adjacency matrix of a graph G is a matrix of 0s and 1s, normally represented by a two-dimensional array A[1 .. V, 1 .. V], where each entry indicates whether a particular edge is present in G.

|V|

|V|

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0



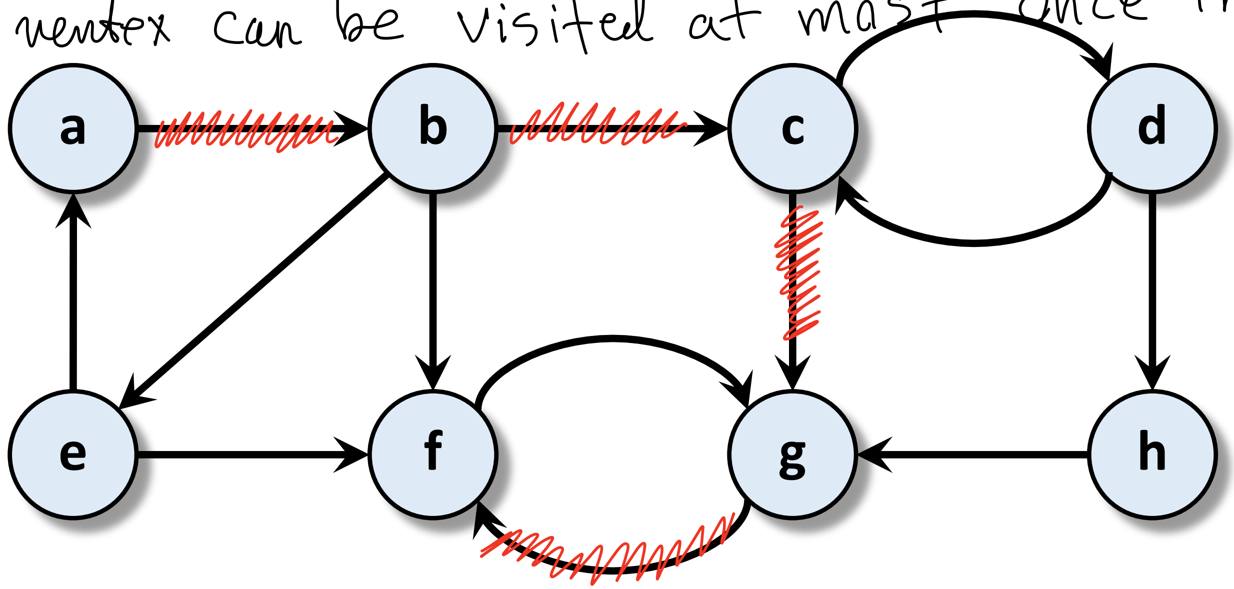
Data Structures: Comparison

deg(u) is the # of neighbors of u.

	Standard adjacency list (linked lists)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)$

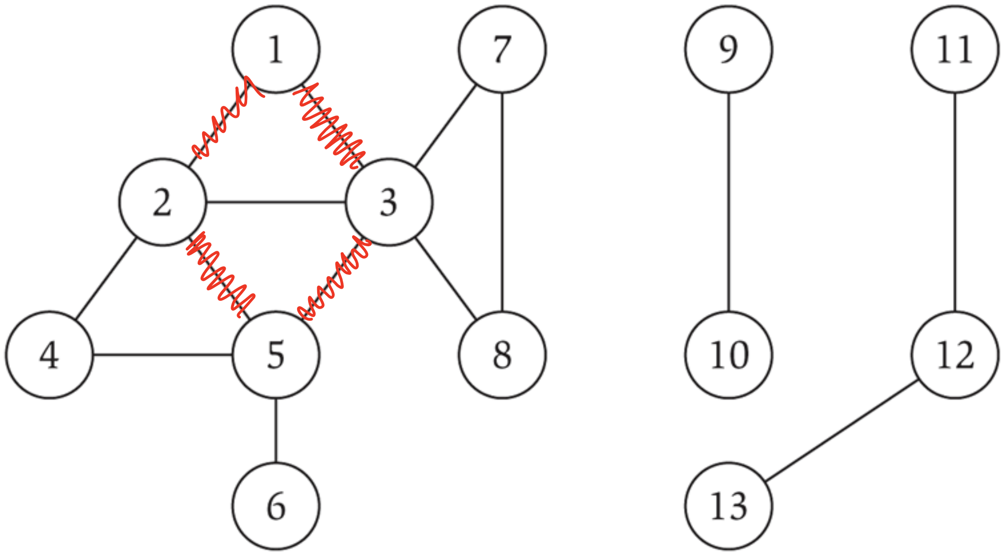
Basic Graph Theory: Paths

- A path is a sequence of consecutive edges in E
  - $P = \{(u, w_1), (w_1, w_2), (w_2, w_3), \ldots, (w_{k-1}, v)\}$
  - $P = u - w_1 - w_2 - w_3 - \cdots - w_{k-1} - v$
  - The length of the path is the # of edges
  - A vertex can be visited at most once in a path



Basic Graph Theory: Cycles

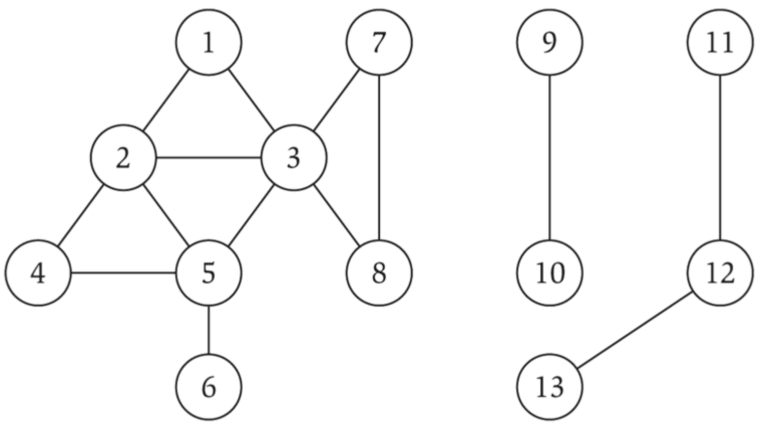
- A cycle is a path $v_1 - v_2 - \cdots - v_k - v_1$ and $v_1, \ldots, v_k$ are distinct

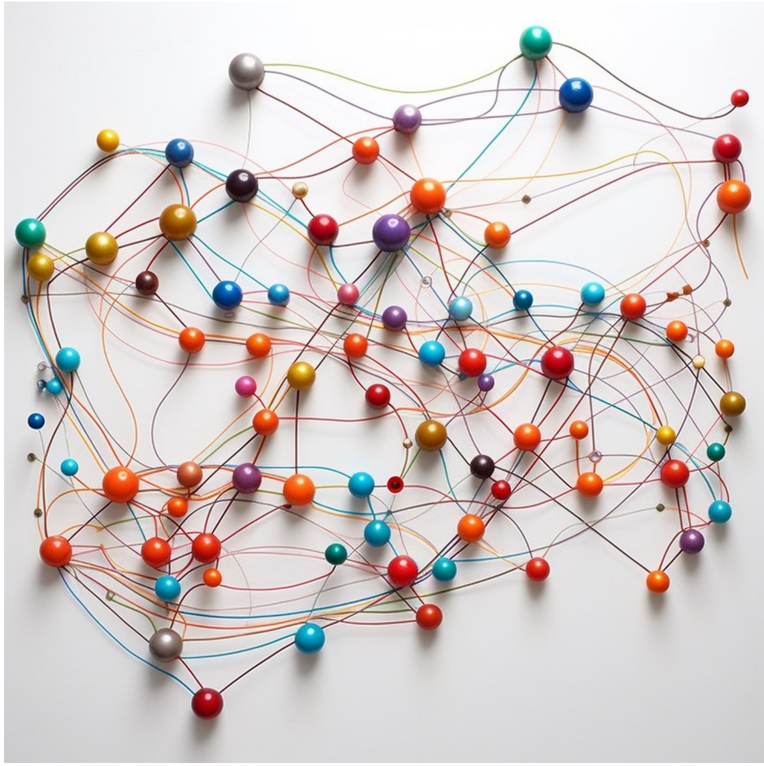


Def: A graph is "simple" if there are no parallel edges or self-loops in the graph

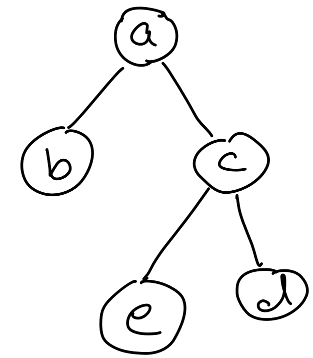
Basic Graph Theory: Connectivity

- An undirected graph is connected if there is a path between every two vertices in the graph.

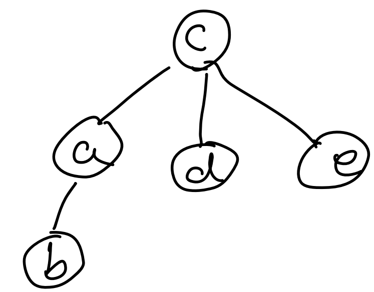




Basic Graph Theory: Trees

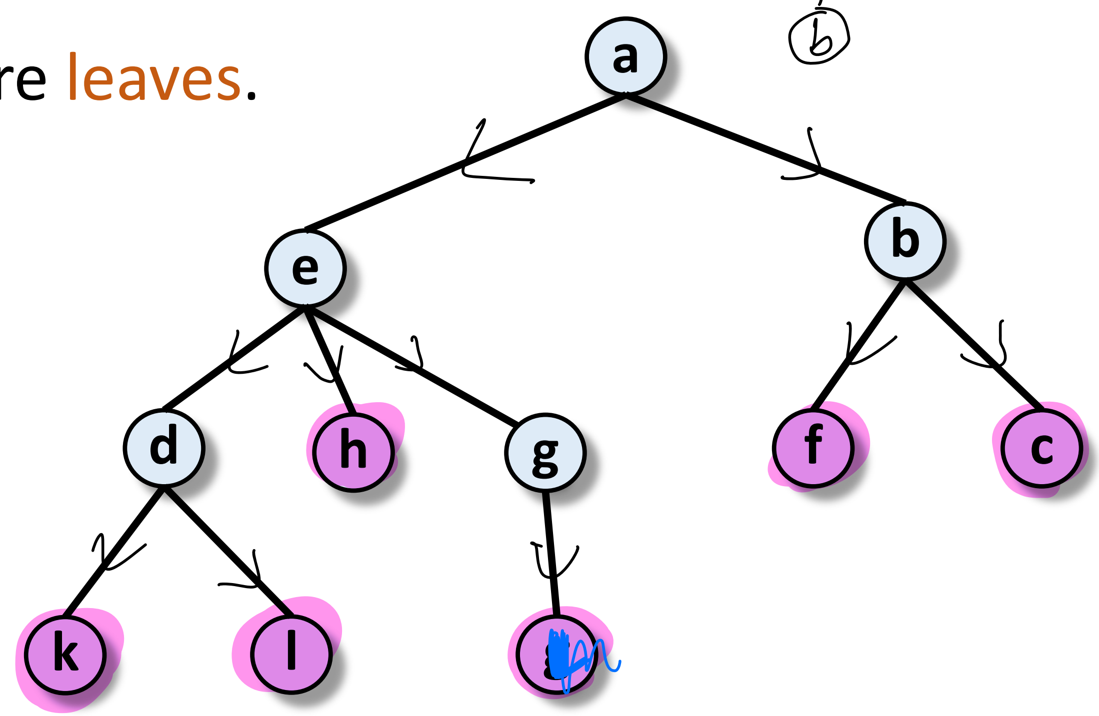


- A simple undirected graph G is a tree if:
  - G is connected
  - G contains no cycles



- Degree one vertices are leaves.
- A collection of trees is

called a forest.



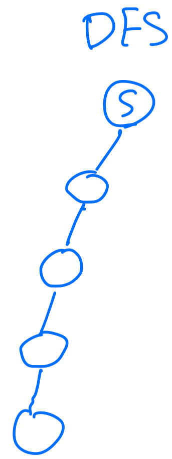
Claim: Every tree on n vertices has exactly $n-1$ edges.

Graphs and Graph Traversals

a. Introduction to Graphs

b. Graph Traversals: DFS

Exploring a Graph



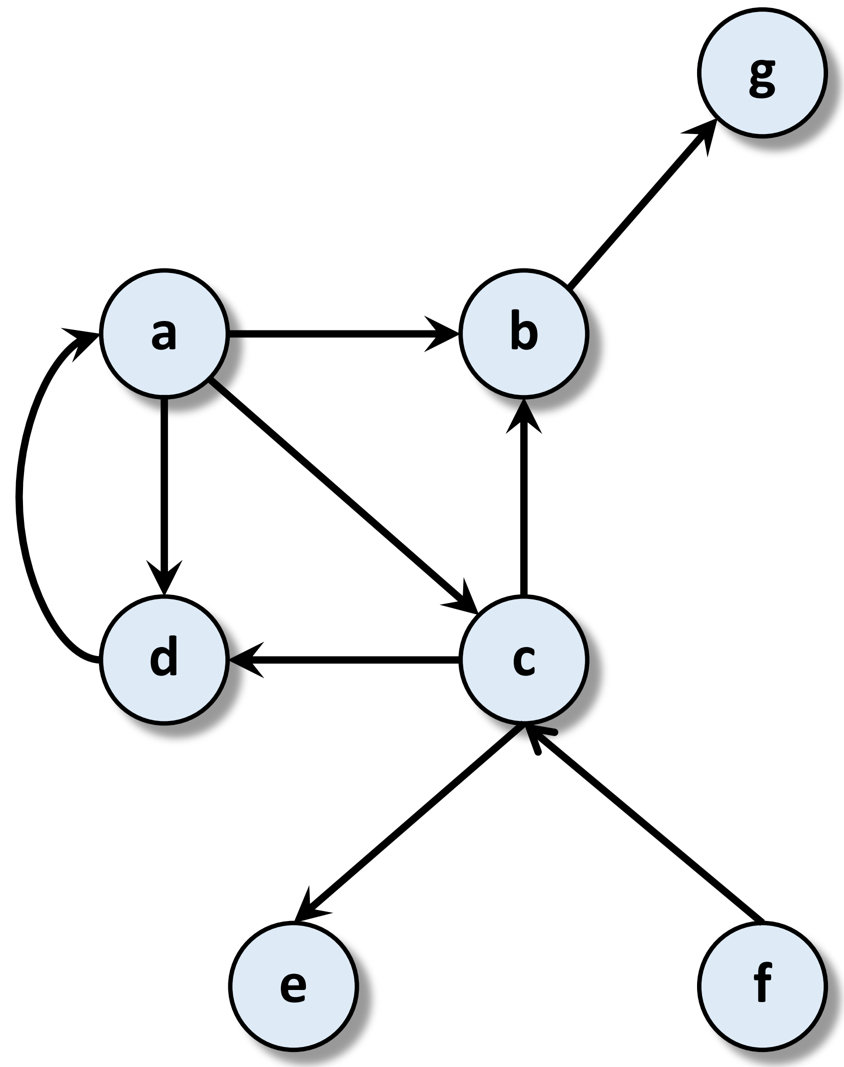
- **Problem:** Is there a path from s to t?
- **Idea:** Explore all nodes reachable from s.

- Two different search techniques:
  - **Breadth-First Search:** explore nearby nodes before moving on to farther away nodes
  - **Depth-First Search:** follow a path until you get stuck, then go back

Depth-First Search (DFS)

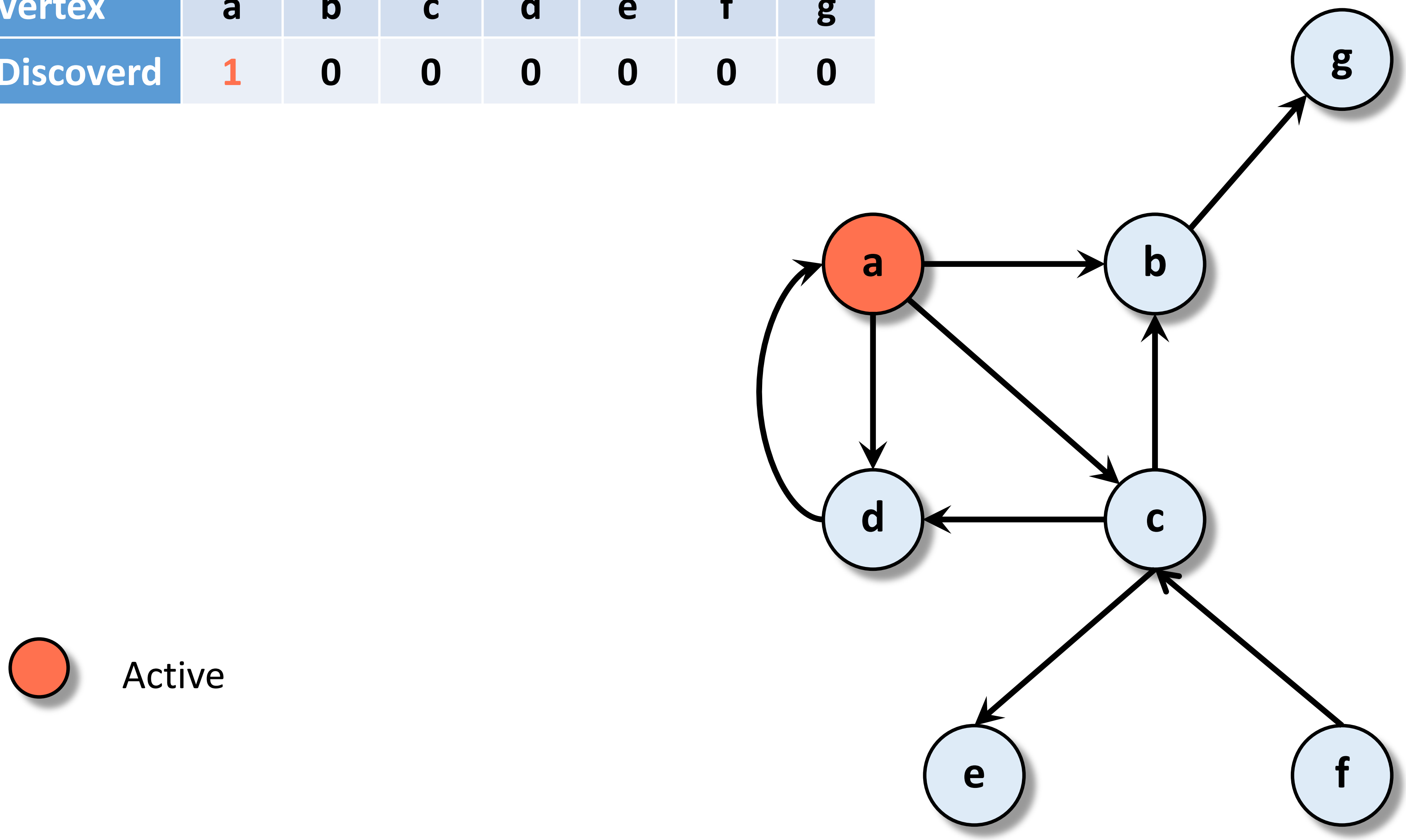
(For both directed and undirected graphs)

Depth-First Search in Directed Graphs



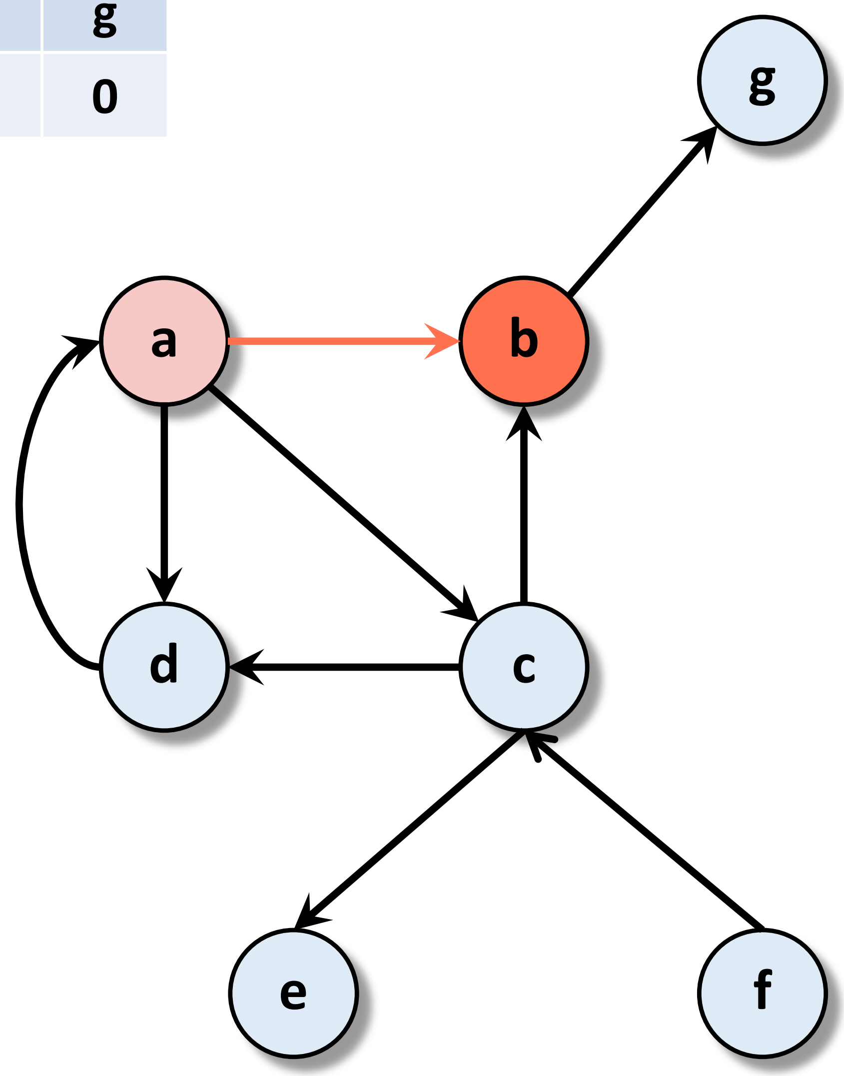
Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	0	0	0	0	0	0



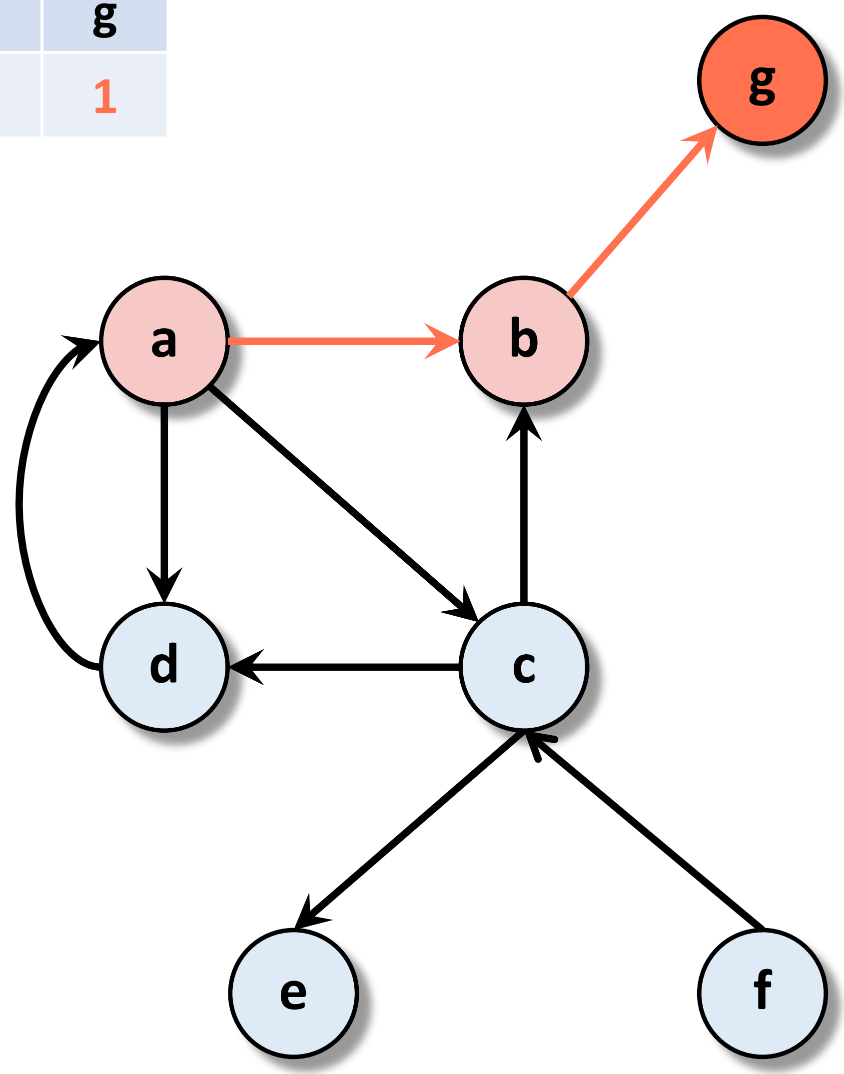
Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	0	0	0	0	0



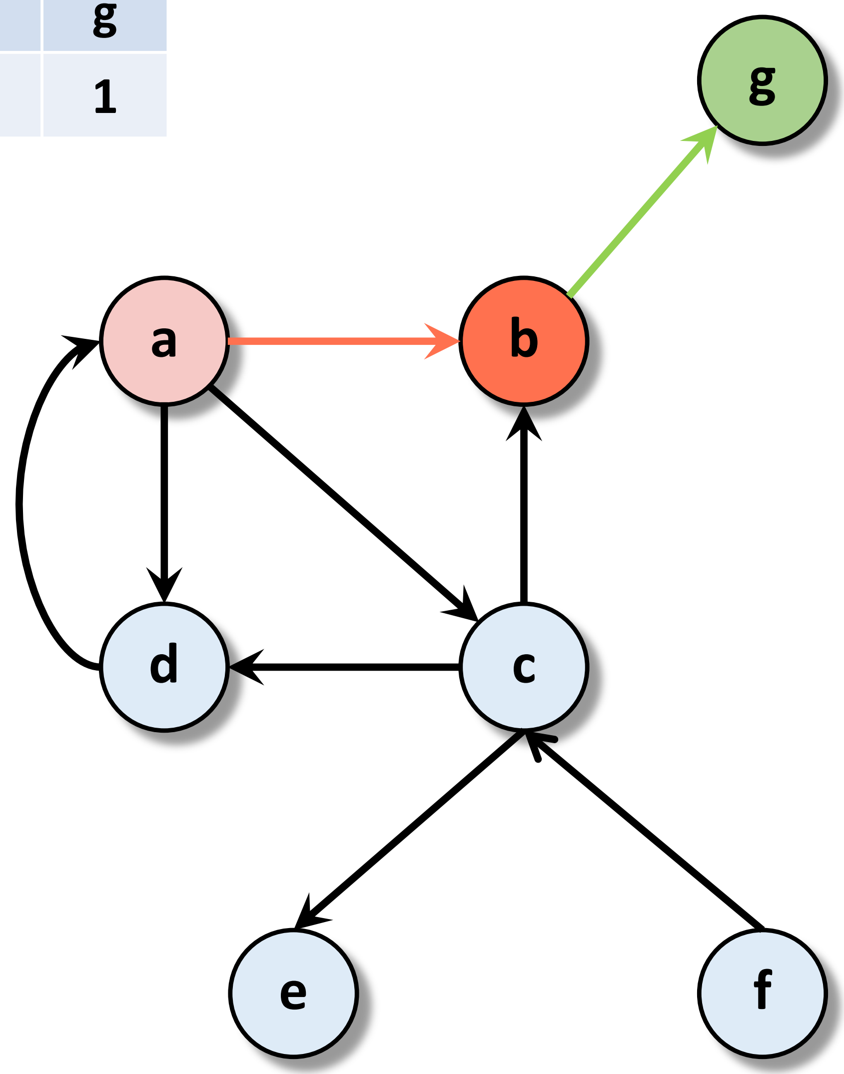
Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	0	0	0	0	1



Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	0	0	0	0	1



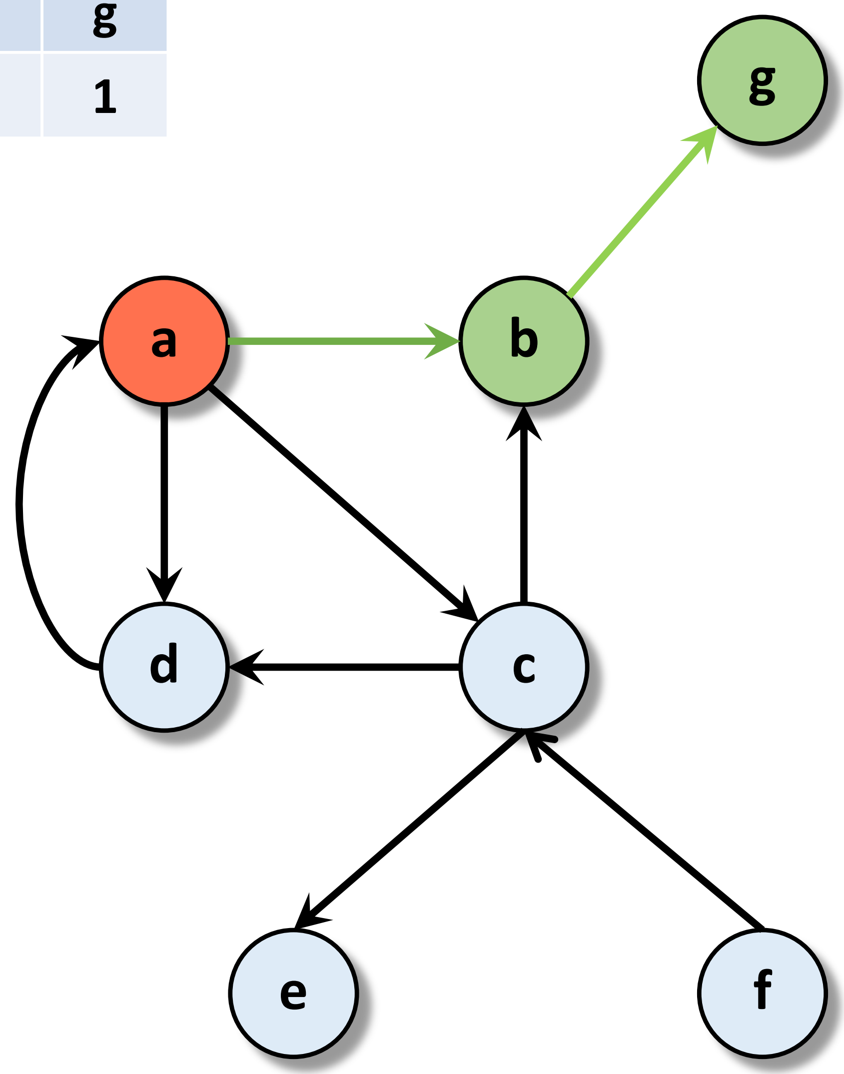
Active

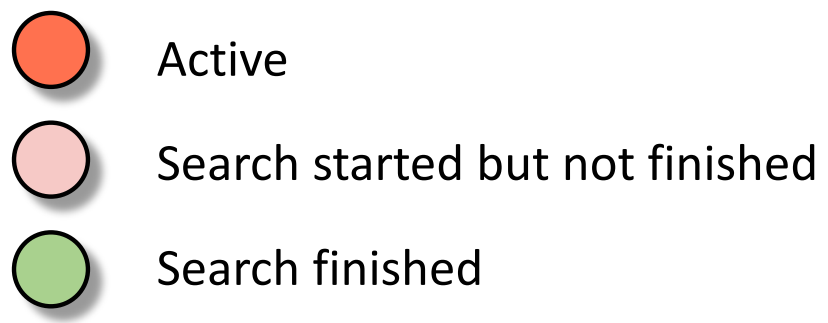
Search started but not finished

Search finished

Depth-First Search in Directed Graphs

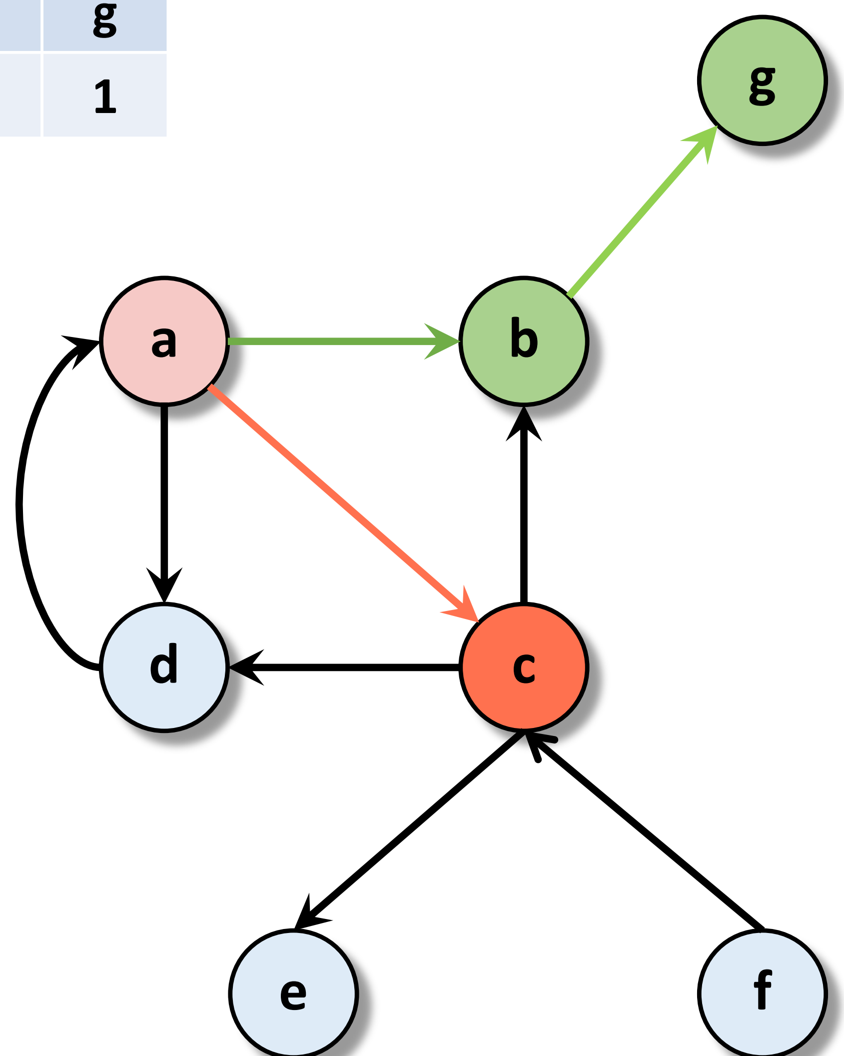
Vertex	a	b	c	d	e	f	g
Discoverd	1	1	0	0	0	0	1

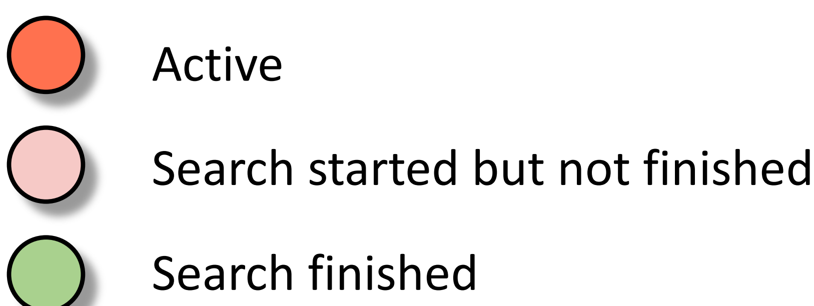




Depth-First Search in Directed Graphs

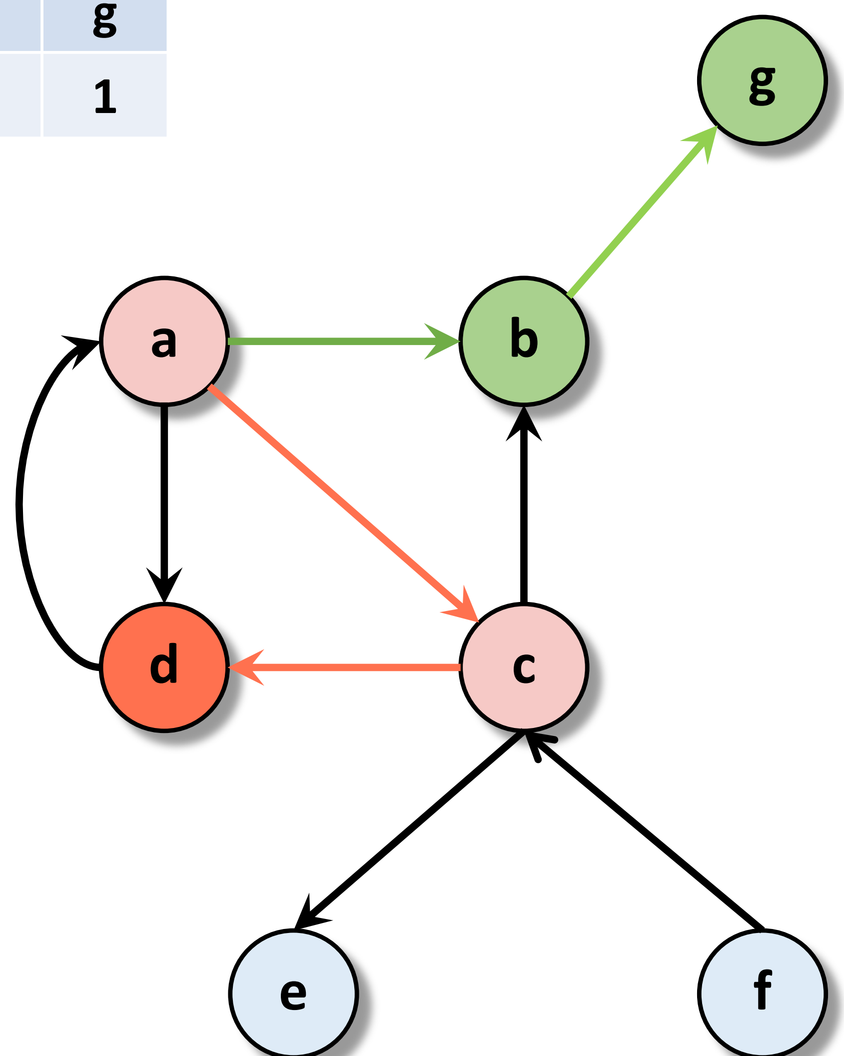
Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	0	0	0	1





Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	0	0	1



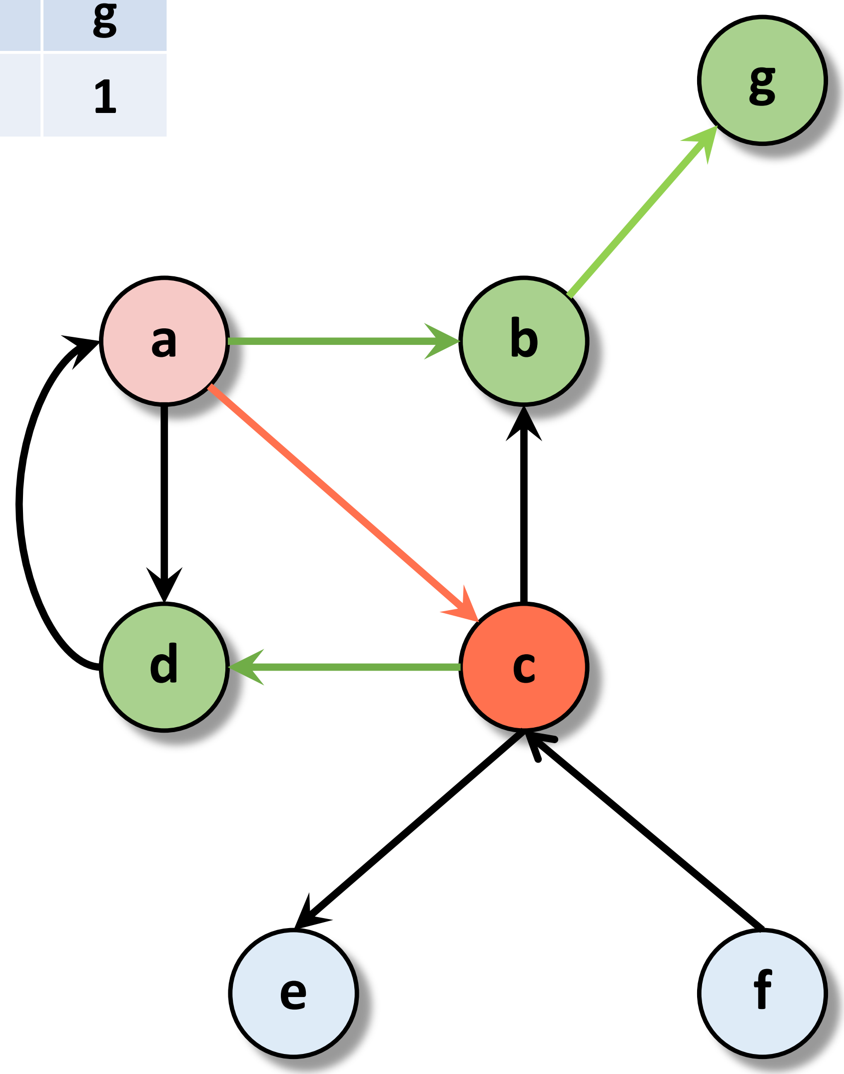
Active

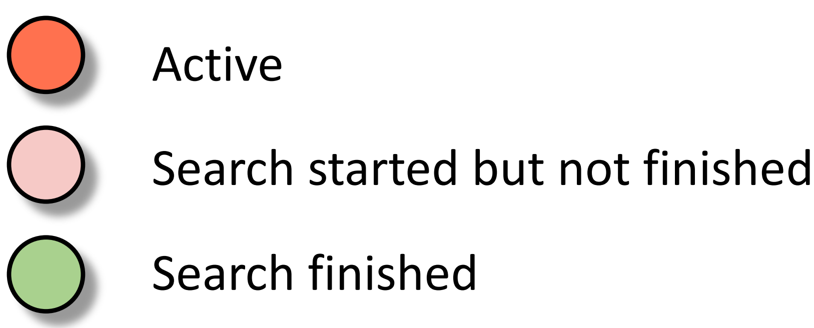
Search started but not finished

Search finished

Depth-First Search in Directed Graphs

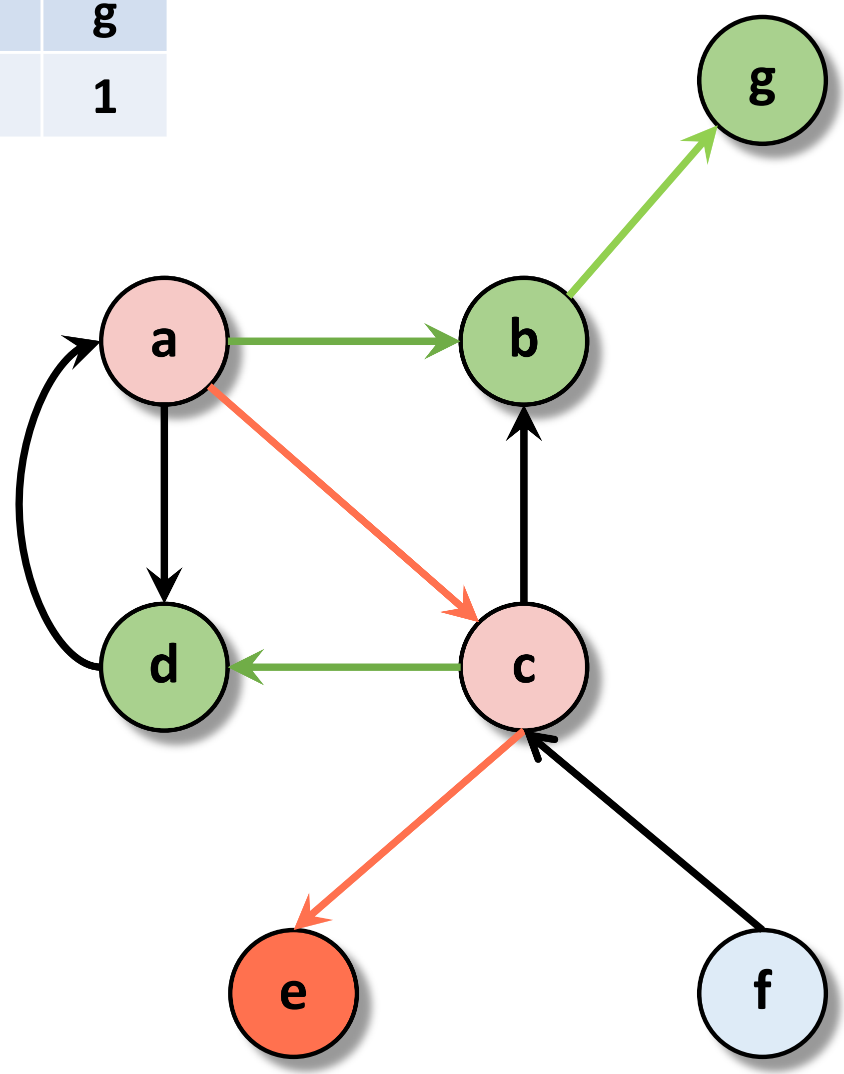
Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	0	0	1

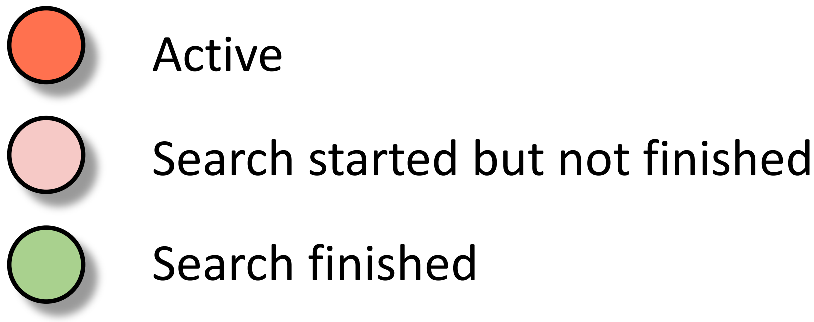




Depth-First Search in Directed Graphs

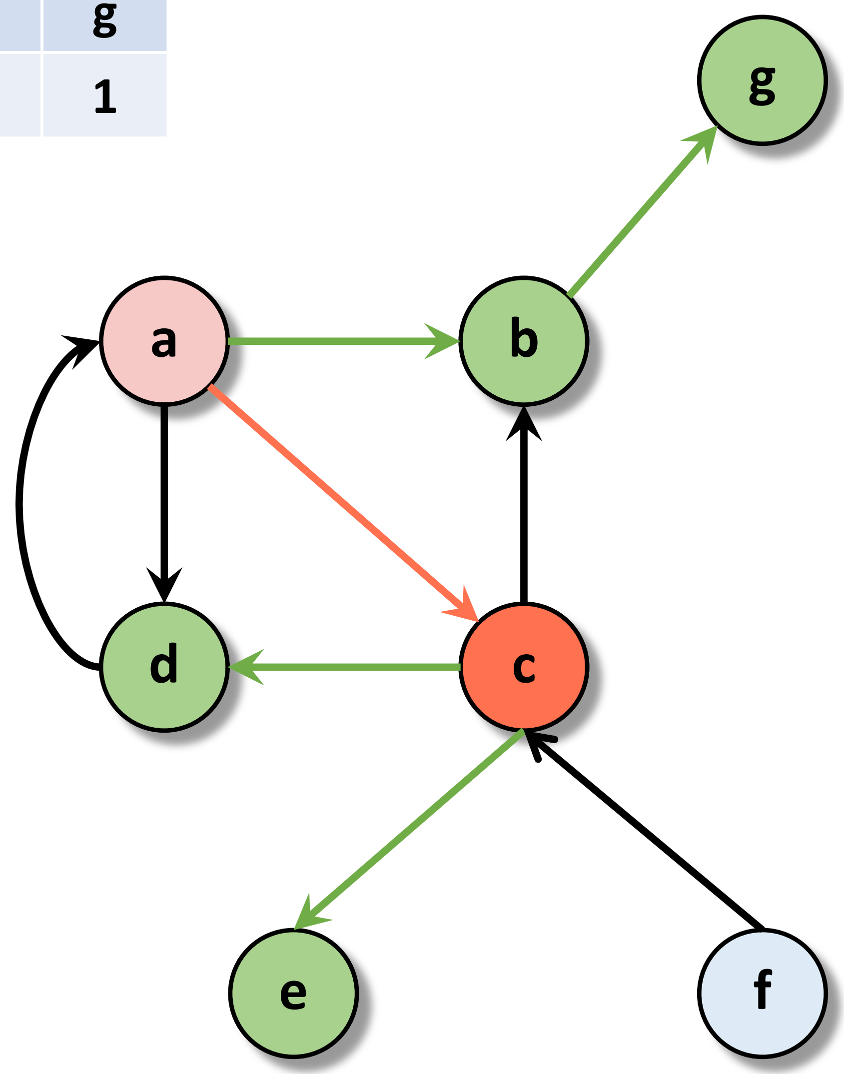
Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	1	0	1





Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	1	0	1



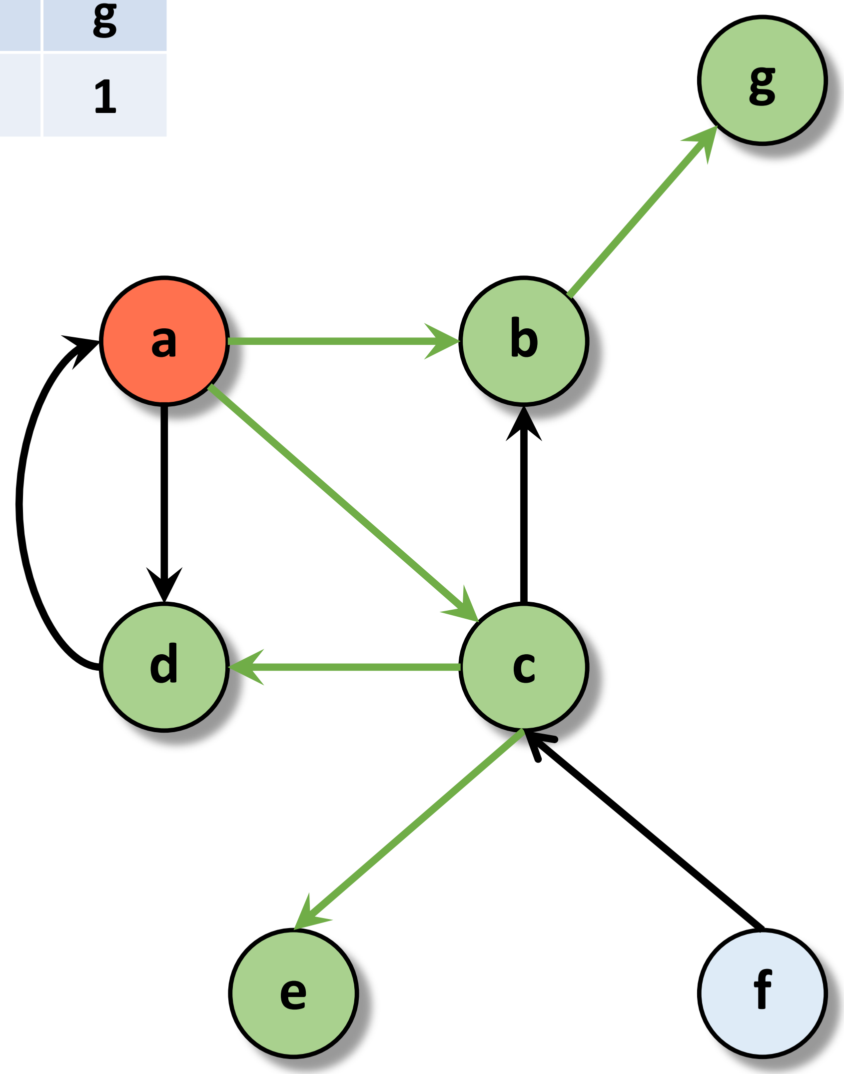
Active

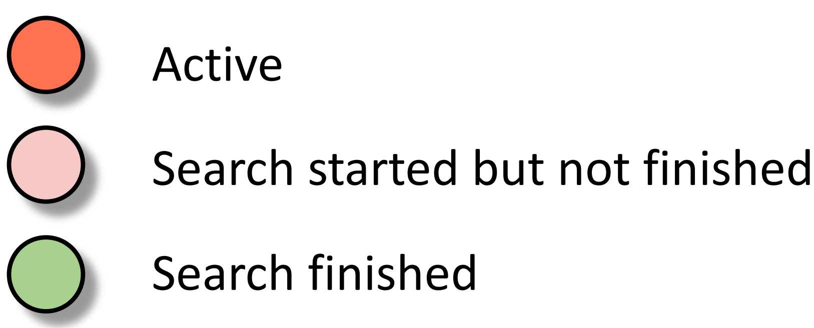
Search started but not finished

Search finished

Depth-First Search in Directed Graphs

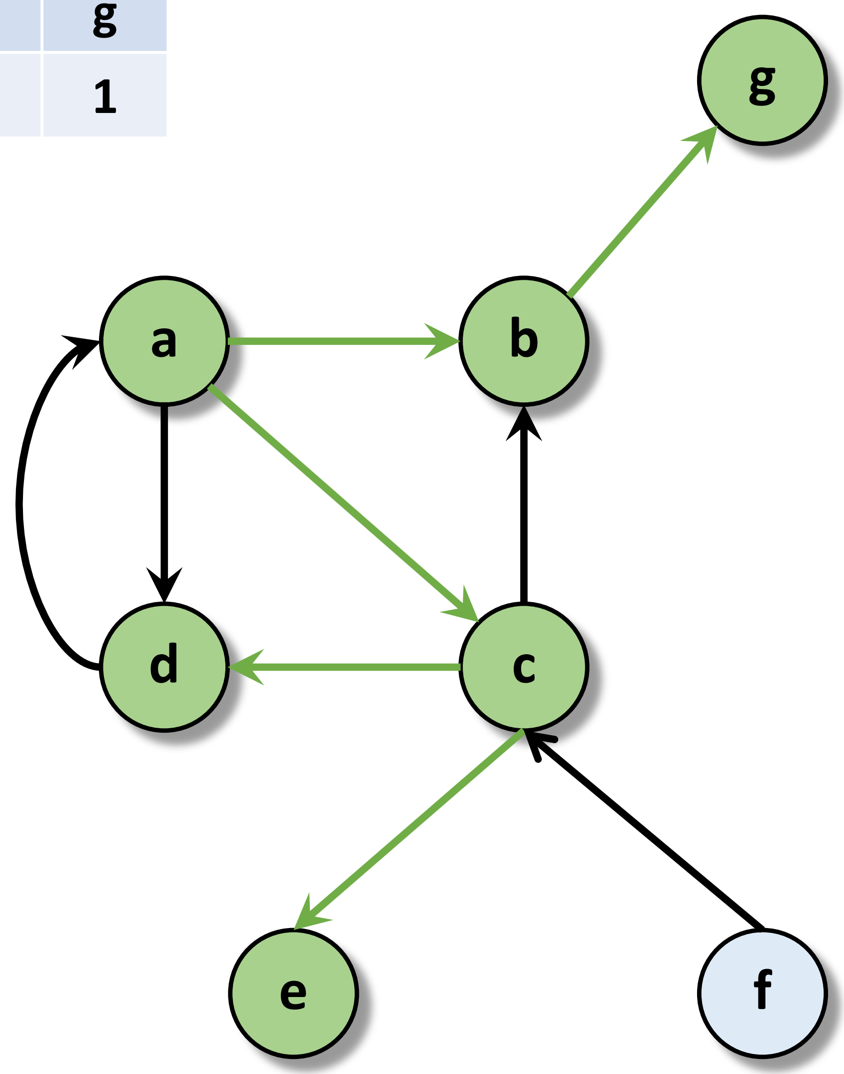
Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	1	0	1





Depth-First Search in Directed Graphs

Vertex	a	b	c	d	e	f	g
Discoverd	1	1	1	1	1	0	1



Active

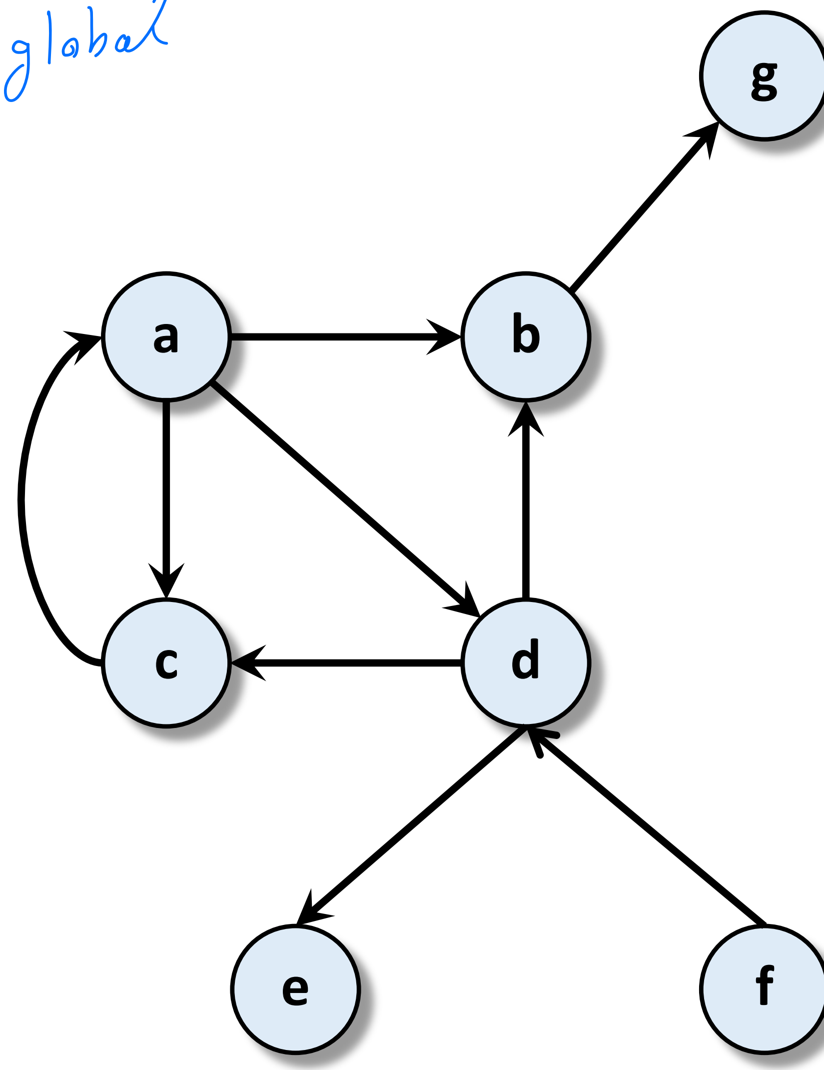
Search started but not finished

Search finished

Depth-First Search in Directed Graphs

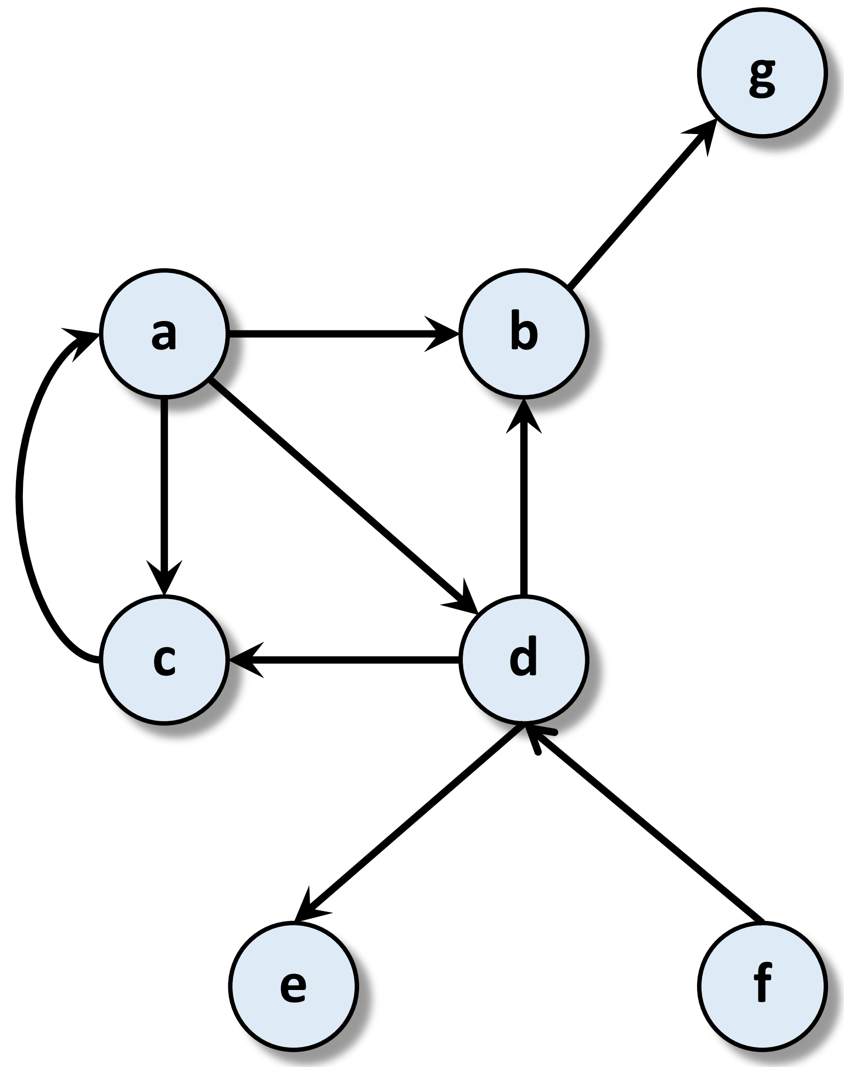
```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1

  for ((u,v) in E):
    if (discovered[v]=0):
      DFS(v)
```

global



Depth-First Search in Directed Graphs

```
G = (V,E) is a graph
discovered[u] = 0 ∀u
parent ← global array
DFS(u):
  discovered[u] = 1

  for ((u,v) in E):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)
```



Practice Problems

$n := |V| \quad m := |E|$

Use DFS to count the number of vertices reachable from a vertex u.

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1

  for ((u,v) in E):
    if (discovered[v]=0):
      DFS(v)
```

Idea: Run DFS(u).

$O(n)$ time

count # of 1's in discovered.

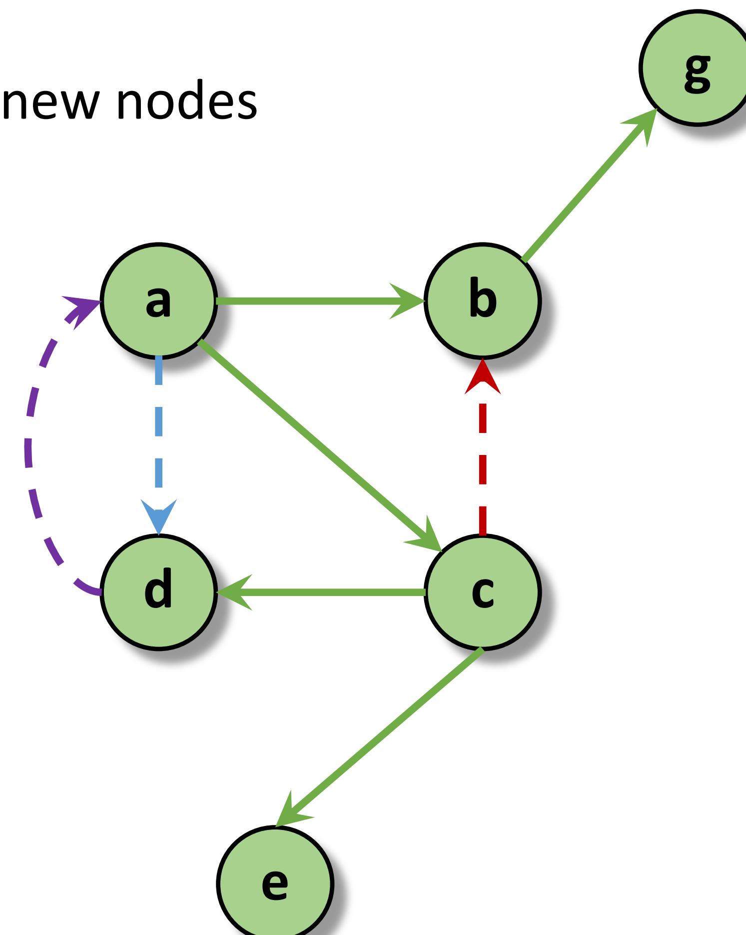
Practice Problems

Use DFS to count the number of vertices reachable from a vertex u.

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  reachable = 1

  for ((u,v) in E):
    if (discovered[v]=0):
        reachable+= DFS(v)

  return reachable
```

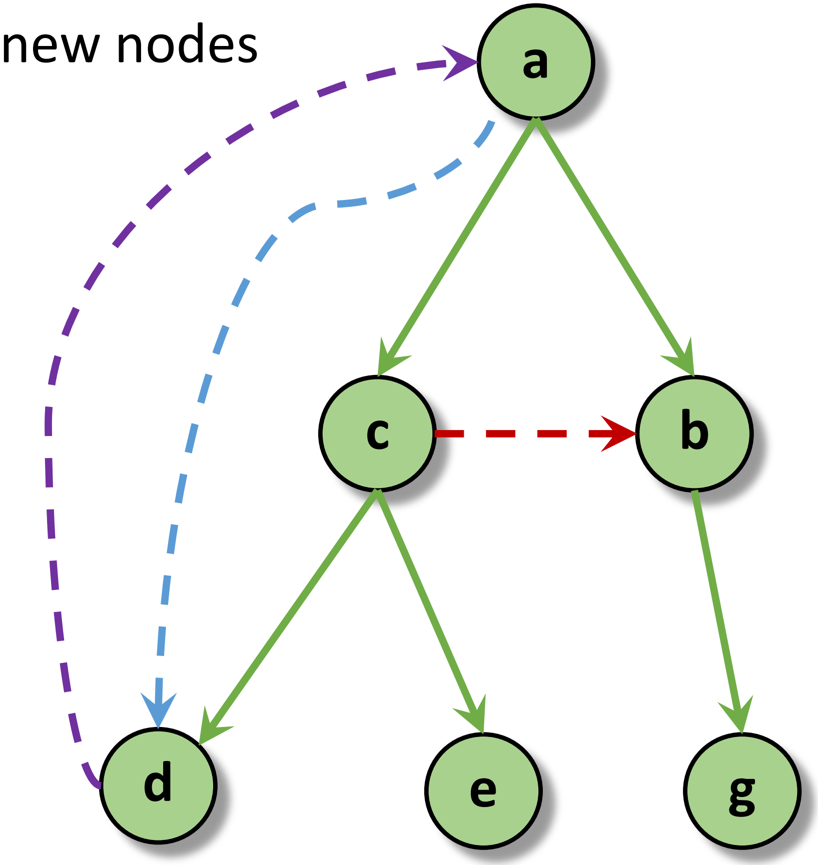
Depth-First Search in Directed Graphs

- **Fact:** The parent-child edges form a (directed) tree
- **Each edge in G has a type:**
  - Tree **edges:** $(a, b), (b, g), (c, e)$
    - These are the edges that discover new nodes
  - Forward **edges:** (a, d)
    - Ancestor to descendant
  - Back **edges:** (d, a)
    - Descendant to ancestor
    - Implies a directed cycle!
  - Cross **edges:** (c, b)
    - No ancestral relation



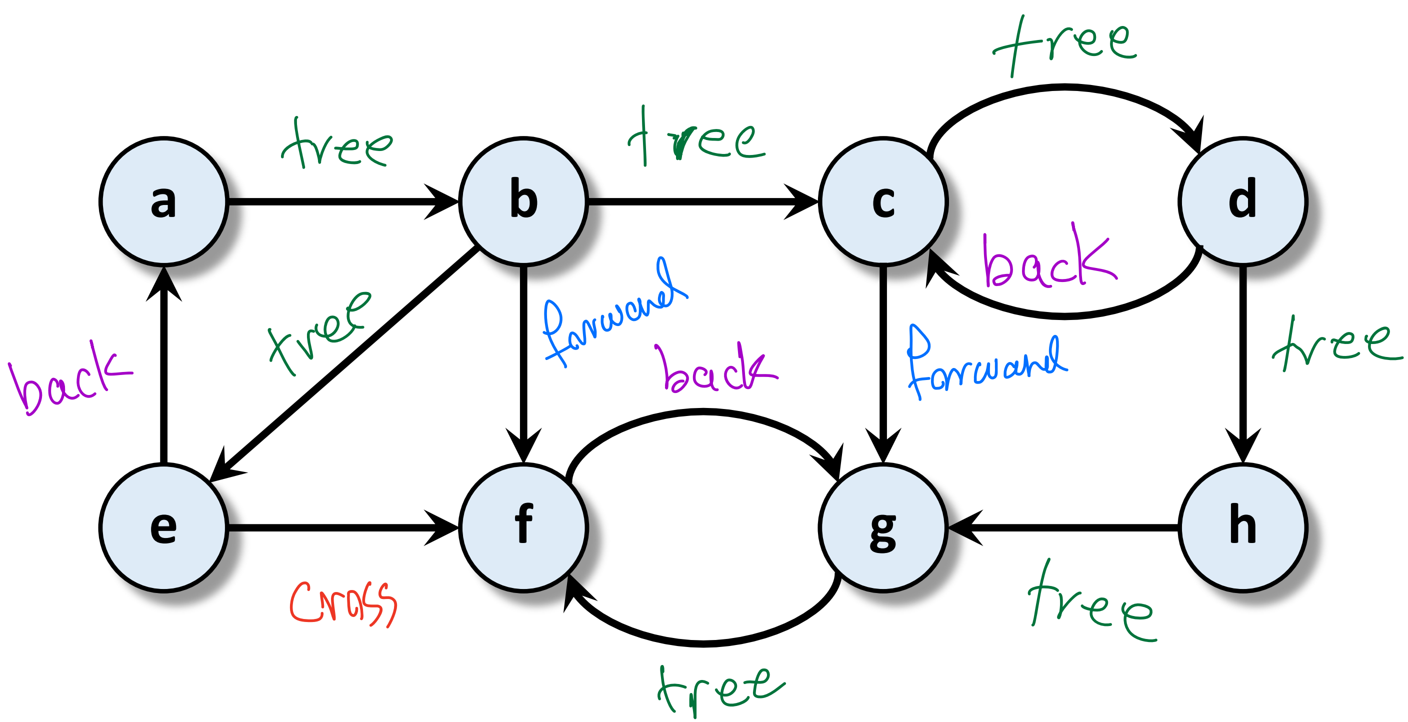
Depth-First Search in Directed Graphs

- **Fact:** The parent-child edges form a (directed) tree
- **Each edge in G has a type:**
  - **Tree edges:** $(a, b), (b, g), (c, e)$
    - These are the edges that discover new nodes
  - **Forward edges:** (a, d)
    - Ancestor to descendant
  - **Back edges:** (d, a)
    - Descendant to ancestor
    - Implies a directed cycle!
  - **Cross edges:** (c, b)
    - No ancestral relation



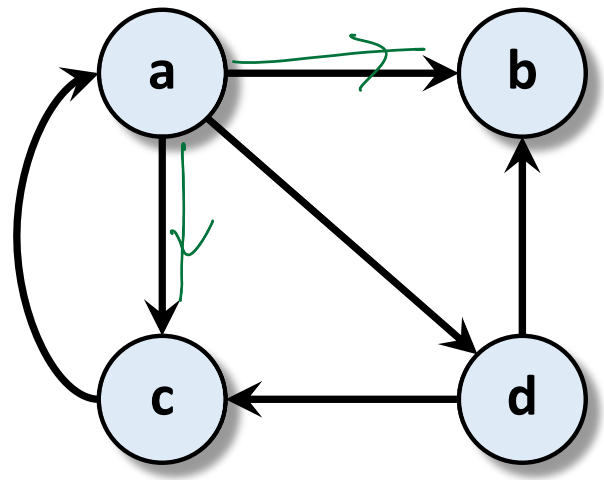
Ask the Audience

- DFS starting from node a
  - Search in alphabetical order
  - Label edges with {tree, forward, back, cross}



Discovery and Finish Times

```
G = (V,E) is a graph
discovered[u] = 0 ∀u
 clock = 1 , is a global variable
DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++        discover time

  for ((u,v) in E):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++        finish time
```

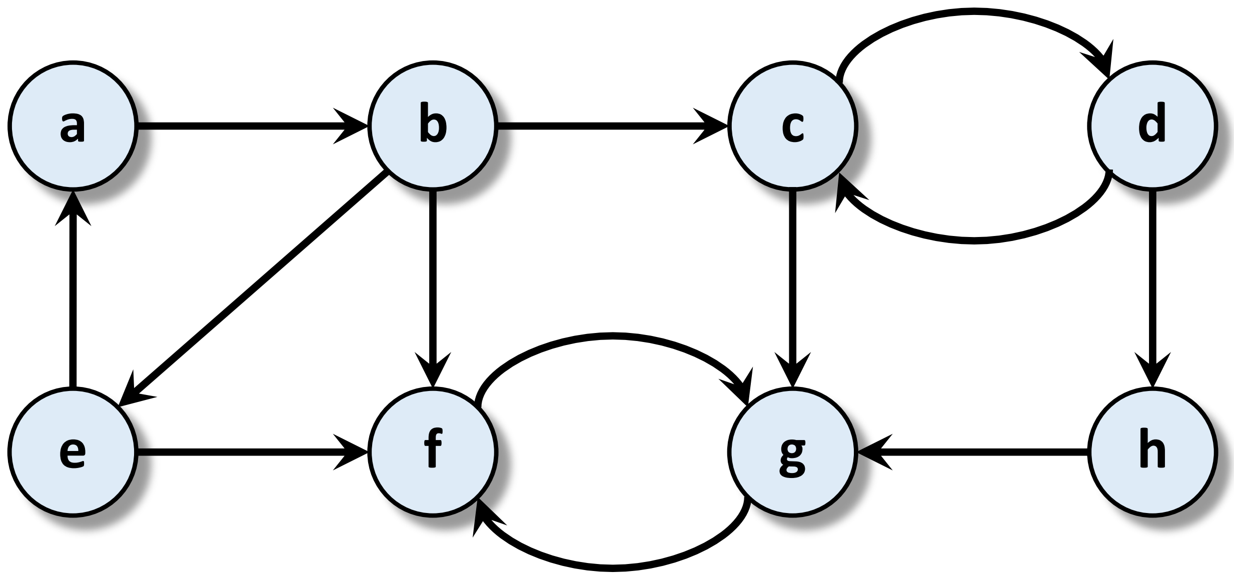


Vertex	Discovery	Finish
a	1	8
b	2	3
c	4	5
d	6	7

- Maintain a counter **clock**, initially set **clock = 1**, and ++ it when starting and finishing the search of a vertex.

Ask the Audience

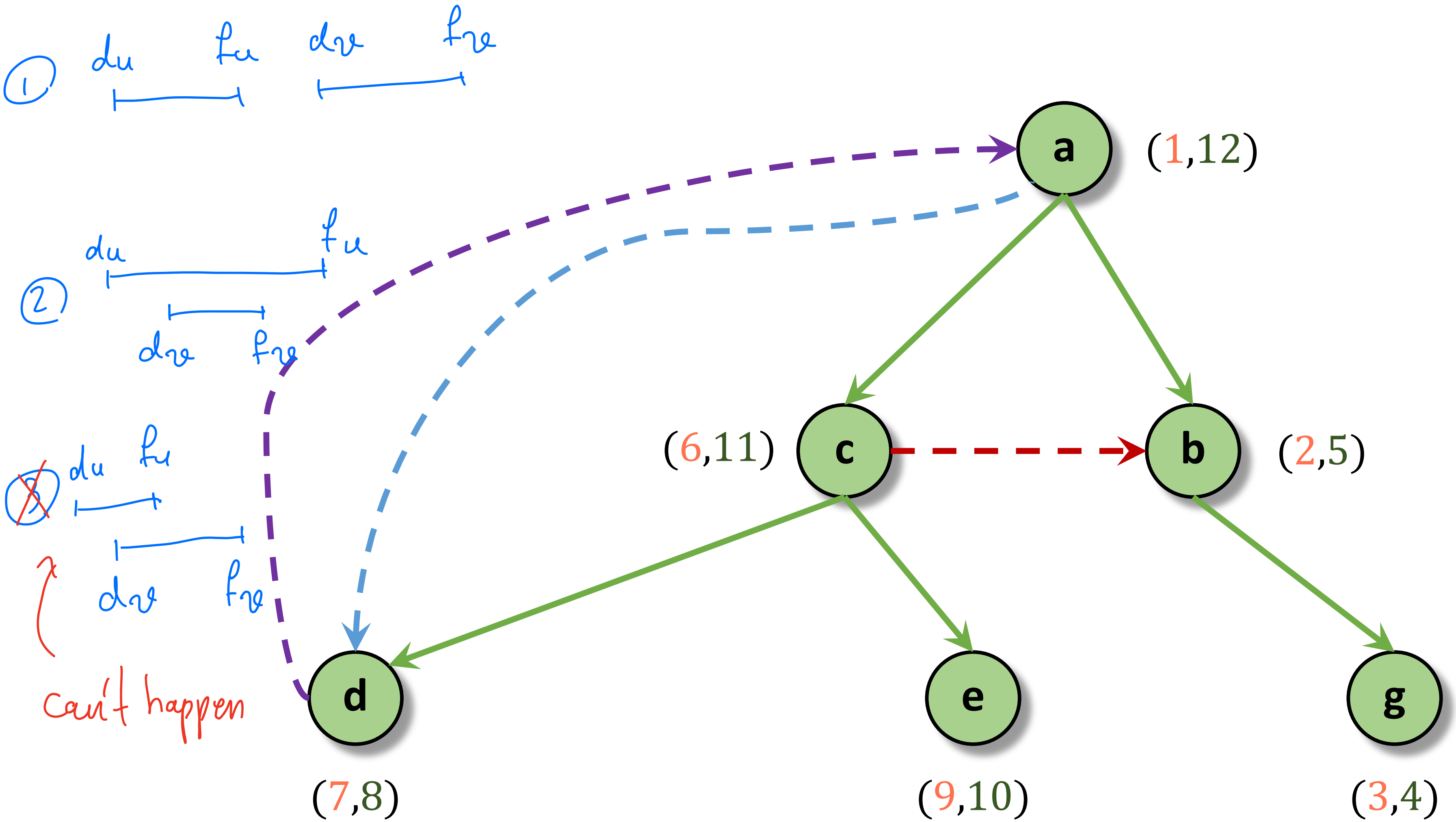
- Compute the **discovery and** finish times for this graph
  - DFS from $\boldsymbol{a}$, search in alphabetical order



Vertex	a	b	c	d	e	f	g	h
Discovery d[]	1	2	3	4	13	7	6	5
Finish f[]	16	15	12	11	14	8	9	10

Discovery and Finish Times

(d_u, f_u) and (d_v, f_v) either nest or are disjoint



Discovery and Finish Times

(d_u, f_u) and (d_v, f_v) either nest or are disjoint

Let u be the vertex that is discovered first and consider the two possible cases:

- There is a path from u to v

  v must be discovered before u finishes.

- No path from u to v

  u discovers all undiscovered vertices reachable from u and finishes. Vertex v is discovered afterwards.

Discovery & Finish Times and Edge Types

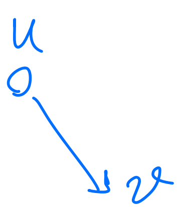
- For any (directed) edge (u, v):

  - u started earlier but finished later $\Longrightarrow$?   Tree or forward

  - v started earlier but finished later $\Longrightarrow$?   back edge

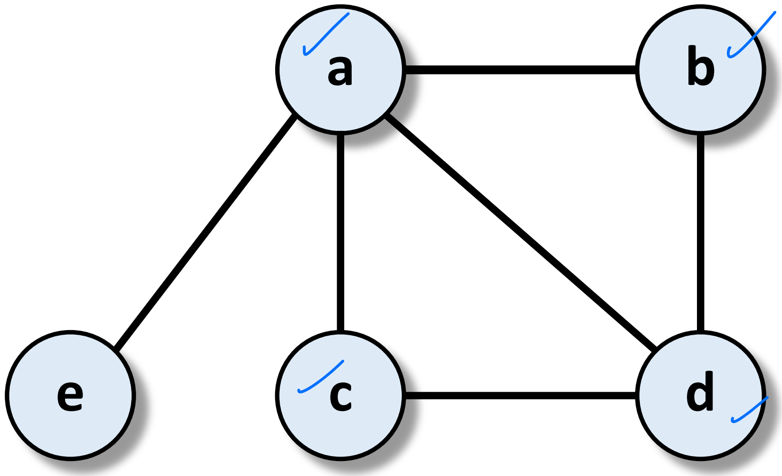
  - v started and finished earlier $\Longrightarrow$?   cross

  - u started and finished earlier $\Longrightarrow$?   Cannot happen



DFS in Undirected Graphs

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++

  for ((u,v) in E):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++
```

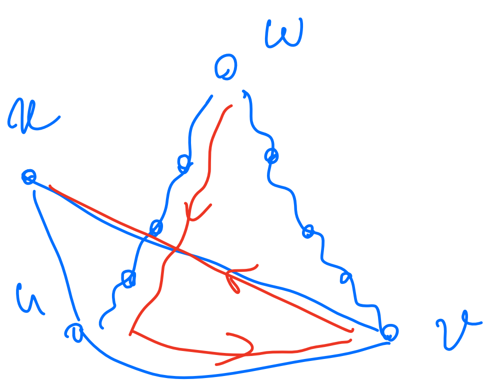


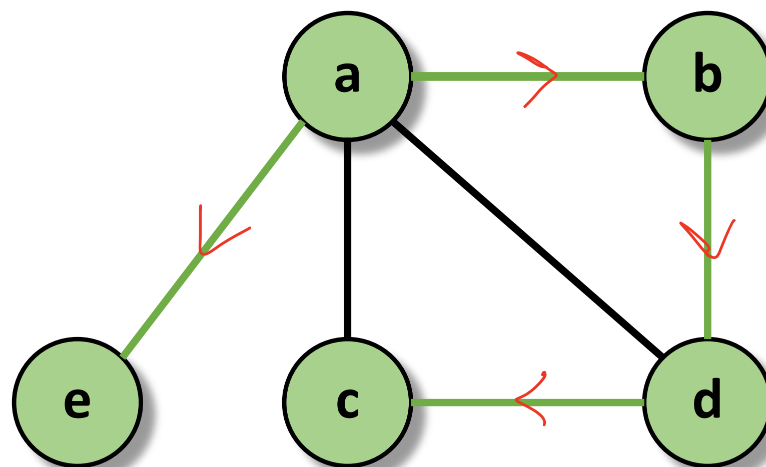
Vertex	Discovery	Finish
a	1	10
b	2	7
c	4	5
d	3	6
e	8	9

- Maintain a counter **clock**, initially set **clock = 1**, and ++ it when starting and finishing the search of a vertex.

Depth-First Search in Undirected Graphs

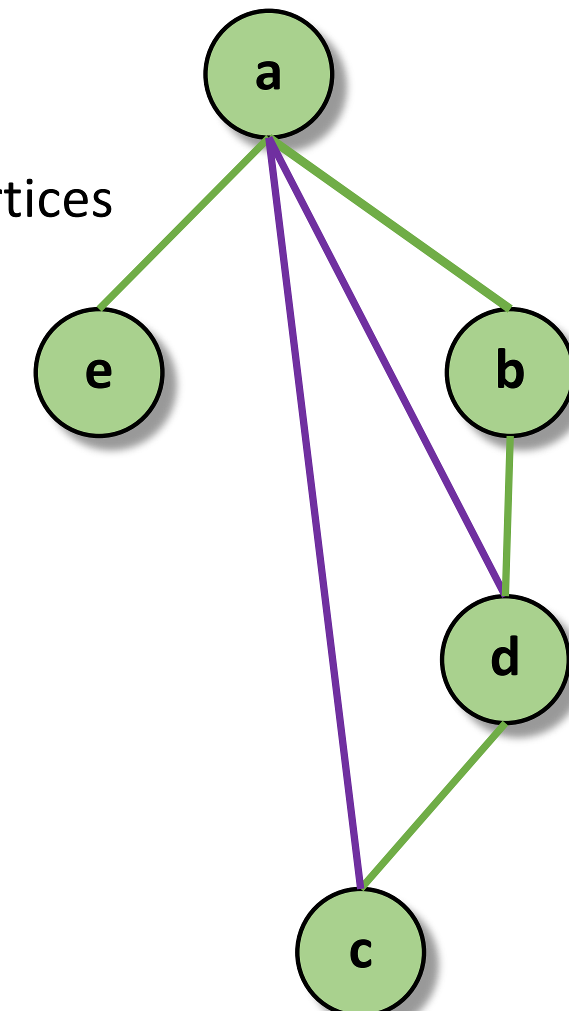
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - **Tree edges:** $(a, b), (a, e), (b, d)$
    - These are the edges that discover new vertices
  - **Back edges:** $(c, a), (d, a)$
    - between descendent and ancestor
  - **No forward or cross edges**





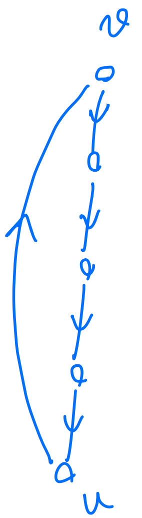
Depth-First Search in Undirected Graphs

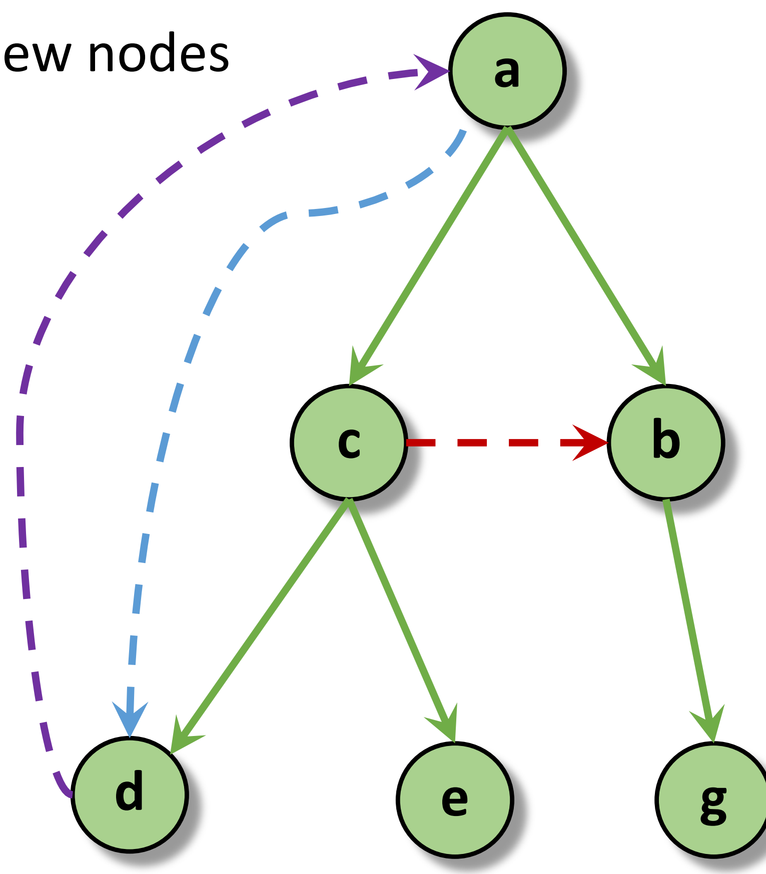
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - Tree **edges:** $(a, b), (a, e), (b, d)$
    - These are the edges that discover new vertices
  - Back **edges:** $(c, a), (d, a)$
    - between descendent and ancestor
  - **No forward or cross edges**



Depth-First Search in Directed Graphs

- **Fact:** The parent-child edges form a (directed) tree
- **Each edge in G has a type:**
  - **Tree edges:** $(a, b), (b, g), (c, e)$
    - These are the edges that discover new nodes
  - **Forward edges:** (a, d)
    - Ancestor to descendant
  - **Back edges:** (d, a)
    - Descendant to ancestor
    - Implies a directed cycle!
  - **Cross edges:** (c, b)
    - No ancestral relation





DFS Running Time (w/ adj. lists)

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++

  for ((u,v) in E):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++
```

DFS Running Time (w/ adj. lists)

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++

  for ((u,v) in E):
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++
```

- Initialization takes $\Theta(n)$
- DFS(u):
  - Processes all edges (u, v) incident from u
  - Calls DFS(v) for every undiscovered v
- Number of recursive calls equal to number of vertices reachable from $u = O(n)$
- Other work constant factor of number of edges reachable from $u = O(m)$
- $O(n + m)$ time and space

vertices

edges

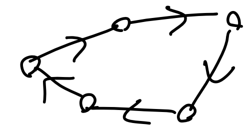
Depth First Search: Recap

- **DFS(s):** Explore all vertices and edges reachable from s.
- Builds a DFS tree, and classifies edges as tree, back, and (for directed) forward, cross
- Different order of processing edges can lead to different DFS trees and classifications
- Regardless, each DFS traversal explores the same set of vertices and edges
- Running time = $O(n + m)$
- Handy subroutine useful for many applications

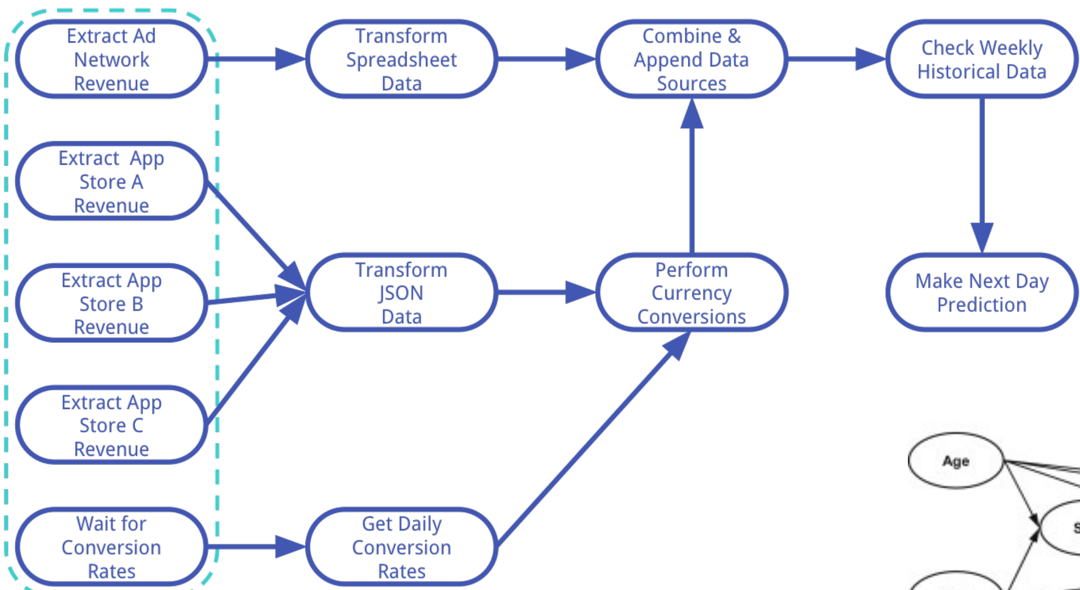
Graphs and Graph Traversals

a. Introduction to Graphs
b. Graph Traversals: DFS
c. Directed Acyclic Graphs

Directed Acyclic Graphs (DAGs)

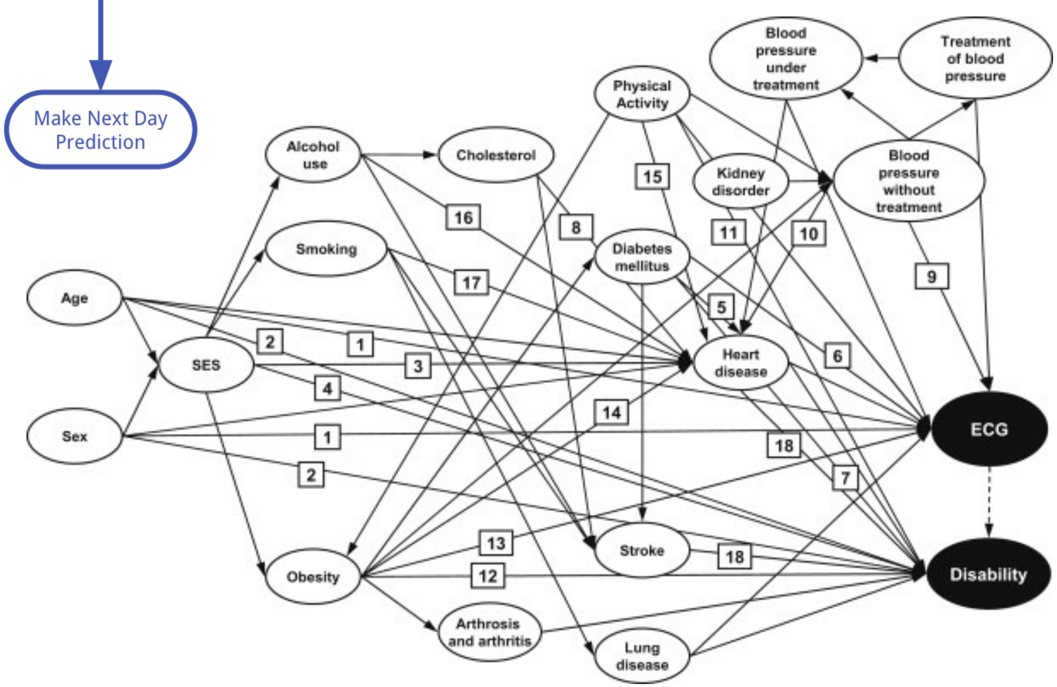


- **DAG:** A directed graph with no directed cycles
- Can be much more complex than an undirected graph without cycles (collection of trees)

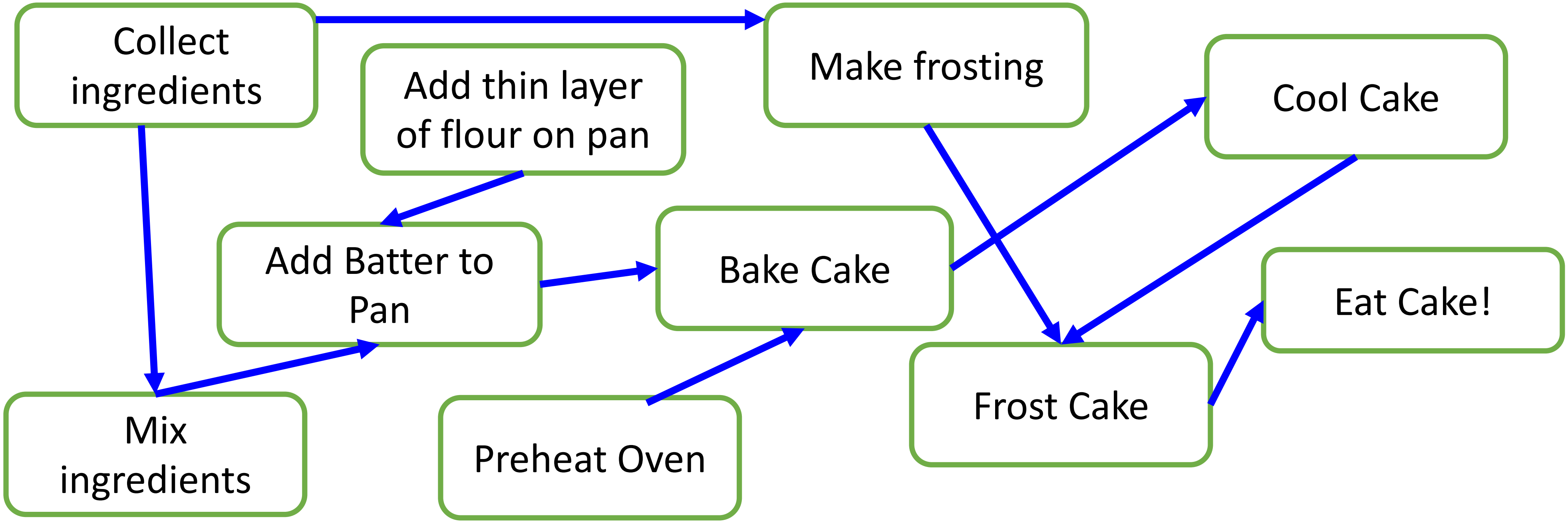


Source: Medium.com on Apache Airflow

Rohrig et al, Journal of Clinical Epidemiology, 2014



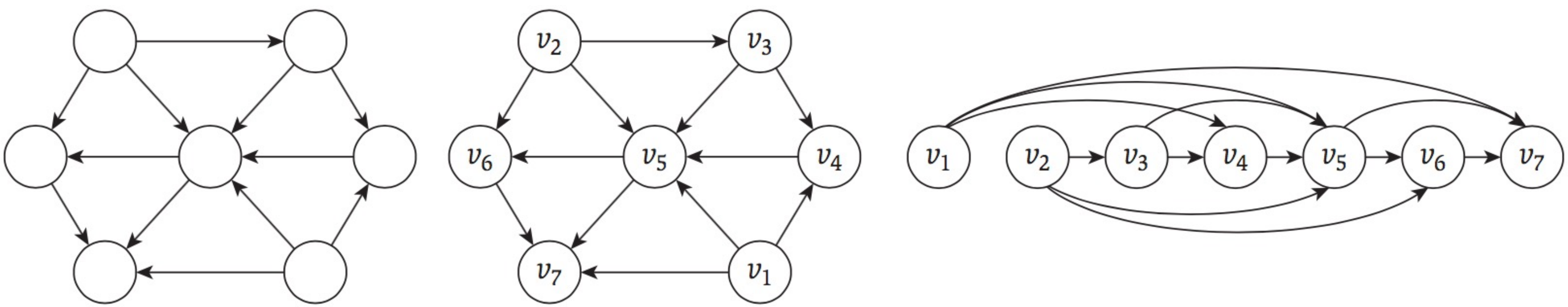
DAGs as Precedence Relationships



- Each node of the DAG is an activity
- Edge (u, v) indicates that u needs to be completed before v can be done
- In what order should the activities be completed?
    - Topological ordering

DAGs and Topological Ordering

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



- A topological ordering of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - G has a topological ordering $\Rightarrow$ G is a DAG

DAGs and Topological Ordering

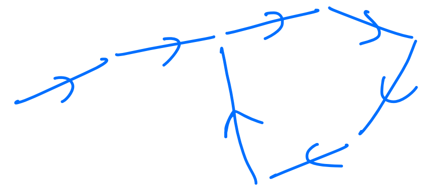
- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

DAGs and Topological Ordering

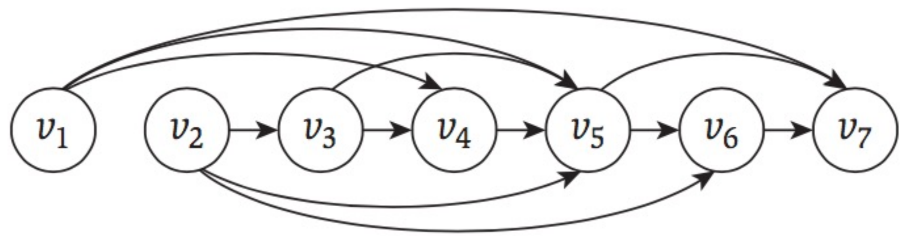
- **Problem 1:** given a digraph G, is it a DAG?
- **Problem 2:** given a digraph G, can it be topologically ordered?

- **Thm:** G has a topological ordering $\Leftrightarrow$ G is a DAG
  - We will design an algorithm that given a DAG returns a topological ordering.

DAGs and In-Degrees



- **Observation:** the last node must have no out-edges



- **Fact:** In any DAG, there is always a node with no outgoing edges (i.e., out-degree of 0)

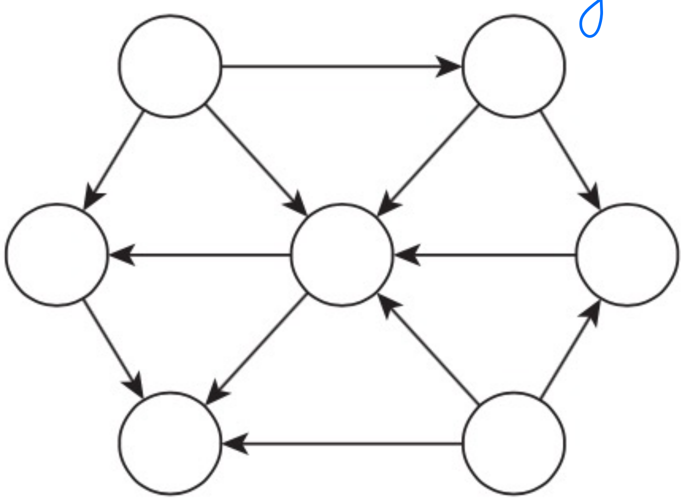
Proof by contradiction. Suppose every vertex has an outgoing edge. Start from an arbitrary vertex v. Keep going through outgoing edge. At some point, we revisit a vertex and find a directed cycle

Existence of Topological Ordering

- **Fact:** In any DAG, there is a node with out-degree 0
- **Theorem:** Every DAG has a topological ordering
- **Proof (Induction):** Induction on n. Base case $n=1$.

Take a vertex v_n with no out-going edge (exists by the fact above).

Delete v and its edges. The remaining graph G' is still a DAG.



Let $v_1, \dots, v_{n-1}$ be the topological ordering of the remaining DAG (exists by IH).

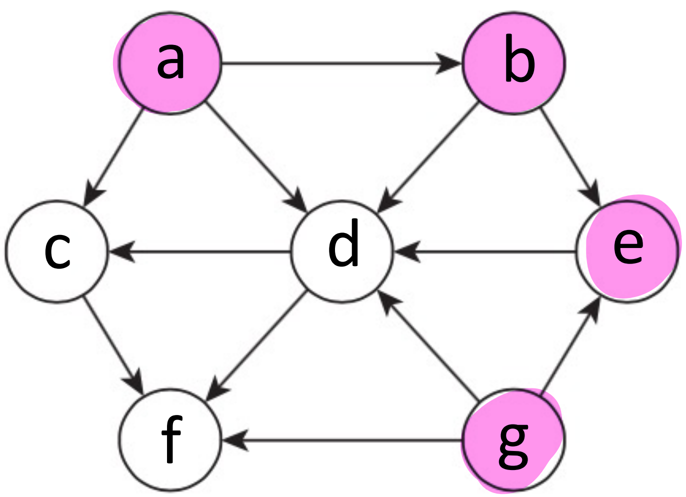
Return $v_1, \dots, v_{n-1}, v_n$.

Topological Ordering Algorithm I

- Repeatedly find node u with zero in-degree
  - Place u next in the order
  - Remove u and out-edges
  - Update in-degrees
  - $\Theta(n^2)$ time

Topological Ordering Algorithm I

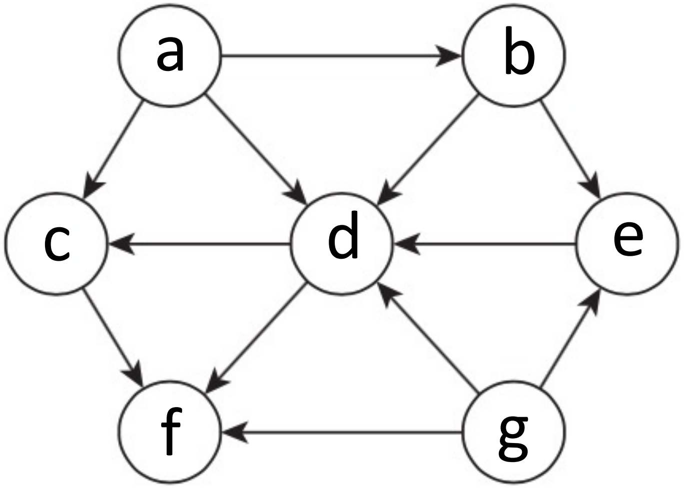
- Repeatedly find node u with zero in-degree
  - Place u next in the order
  - Remove u and out-edges
  - Update in-degrees
  - $\Theta(n^2)$ time



a b g e d c f

Topological Ordering Algorithm I

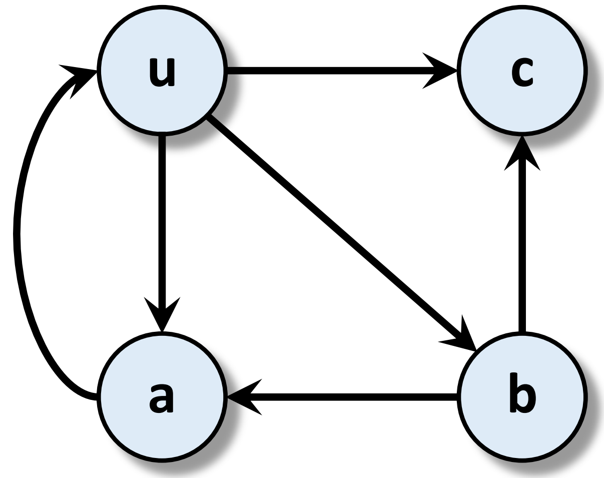
- Repeatedly find node u with zero in-degree
  - Place u next in the order
  - Remove u and out-edges
  - Update in-degrees
  - $\Theta(n^2)$ time



A faster algorithm?

Recall Finish Times

```
G = (V,E) is a graph
discovered[u] = 0 ∀u

DFS(u):
  discovered[u] = 1
  d[u] = clock, clock++

  for (u,v) in E:
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)

  f[u] = clock, clock++
```



Vertex	d	f
u	1	8
a	2	3
b	4	7
c	5	6

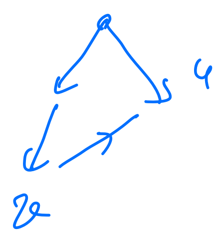
- Maintain a counter **clock**, initially set **clock = 1**

Finish Times and Back Edges

- **Observation:** In a DAG, the first vertex to finish has no outgoing edges.

Suppose v is the first vertex to finish.

If v has an outgoing edge (v,u) then either we have not discovered u yet, which contradicts v having finished, or u is a discovered vertex but this would mean that (v,u) is a backward edge. This gives a directed cycle, contradicting the graph being a DAG.



Finish Times and Back Edges

- **Observation:** In a DAG, any vertex v has only edges to the vertices with smaller finish time.

Finish Times and Back Edges

- **Observation:** In a DAG, any vertex v has only edges to the vertices with smaller finish time.
- **Proof by contradiction:** Assume (v, u) exists with $f_v < f_u$. Two possible cases when v visited:

- u is already discovered

- u is not discovered

Topological Ordering from Finish Times

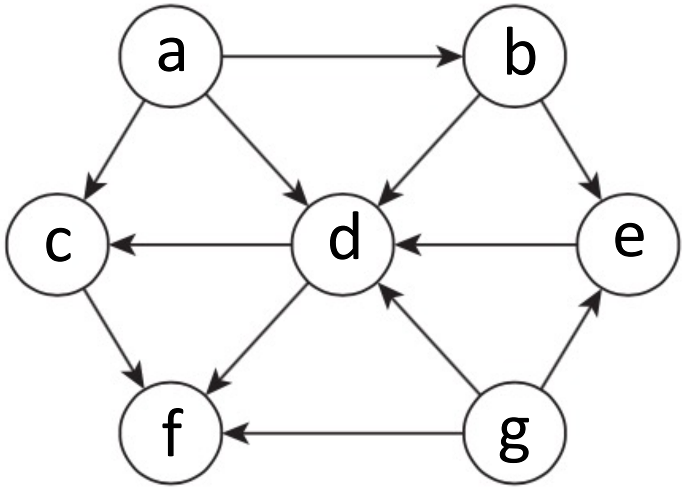
- **Claim:** Ordering nodes by decreasing finish times gives a topological ordering

Topological Ordering Algorithm II

- Initialize
- Run DFS on whole graph
- Return vertices in reverse order of finish times.

```
DFS(u):
  discovered[u] = 1
  for (u,v) in E:
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)
   push u in S


discovered[u] = 0 ∀u
S = empty stack
for u in V:
  if discovered[u] = 0:
    DFS(u)
Return reversed(s)
```

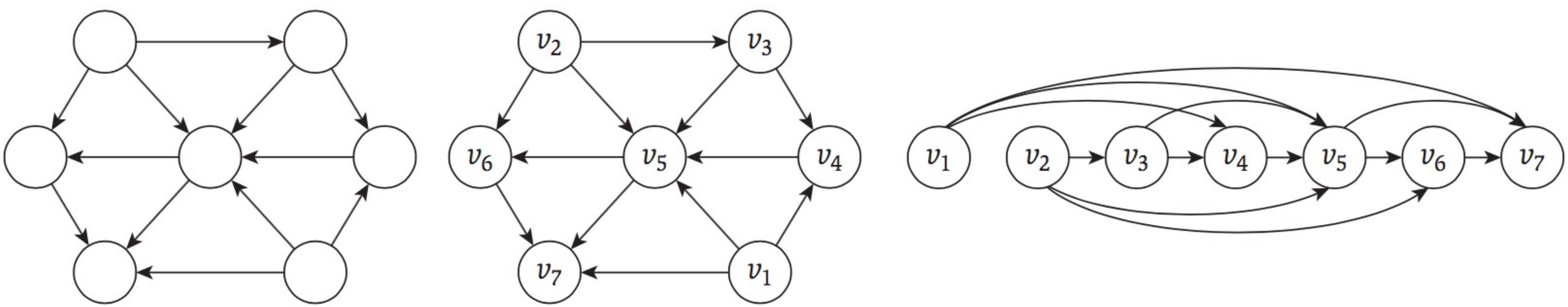
Topological Ordering Algorithm II




```
DFS(u):
  discovered[u] = 1
  for (u,v) in E:
    if (discovered[v]=0):
      parent[v] = u
      DFS(v)
   push u in S


discovered[u] = 0 ∀u
S = empty stack
for u in V:
  if discovered[u] = 0:
    DFS(u)
Return reversed(s)
```

Topological Ordering Recap

- **DAG:** A directed graph with no directed cycles
- Any DAG can be **topologically ordered**
  - Label nodes $v_1, \dots, v_n$ so that $(v_i, v_j) \in E \implies j > i$



- Can compute a TO in $\Theta(n + m)$ time using DFS
  - Reverse of finish times (post-order) is a topological order