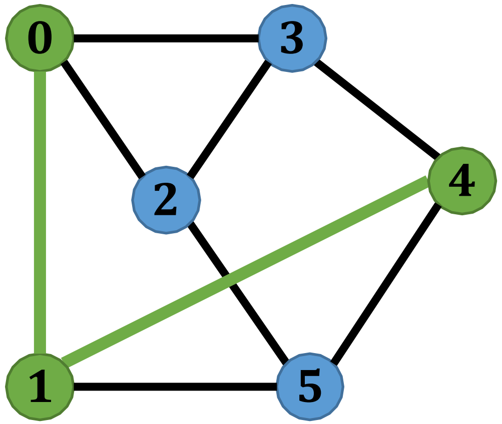


Lecture 4: DiGraphs, Strongly Connected Components

CSC 226: Algorithms and Data Structures II

University of Victoria

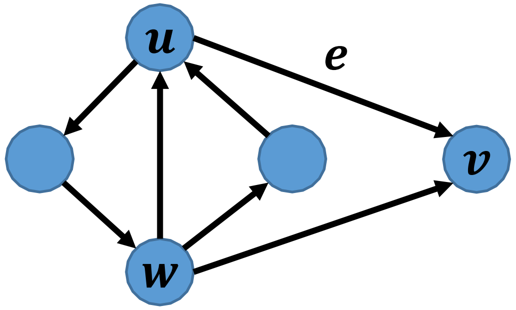
Directed Graphs (Digraphs)

A **directed graph (digraph)** is a graph whose edges are all directed

- Can implement undirected graphs using directed graphs
- Applications include one-way streets, flights, task scheduling

A **directed edge** $\boldsymbol{e}$ represents an **asymmetric** relationship between two vertices $\boldsymbol{u}$ and $\boldsymbol{v}$

- We write $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$ is an ordered pair
- $\boldsymbol{u}$ and $\boldsymbol{v}$ are the **endpoints** of the edge
- $\boldsymbol{u}$ is **adjacent** to $\boldsymbol{v}$ and vice versa
- $\boldsymbol{e}$ is **incident** to $\boldsymbol{u}$ and $\boldsymbol{v}$
- $\boldsymbol{u}$ is the **source vertex** and $\boldsymbol{v}$ is the **destination vertex**

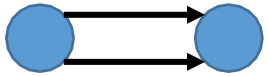


- The **indegree** of a vertex is the number of incoming edges ($\mathbf{indeg}(\boldsymbol{w}) = 1$)
- The **outdegree** of a vertex is the number of outgoing edges ($\mathbf{outdeg}(\boldsymbol{w}) = 3$)

Simple Digraphs

A **simple digraph** is a graph with **no self-loops and no parallel / multi-edges**

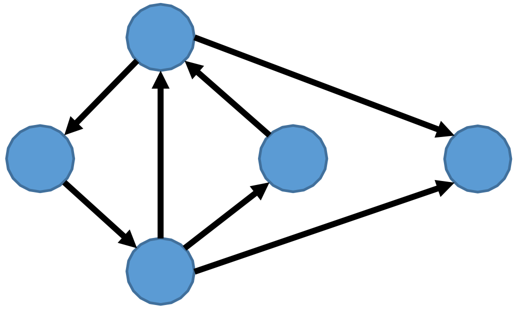
- Note that parallel edges in digraphs refer to edges pointing in the same direction



Theorem: If $G = (V, E)$ is a digraph with m edges, then

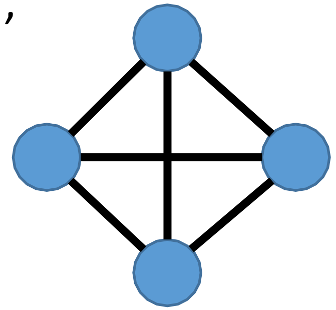
$$(\ )$$

$$\text{indeg}(v) = \text{outdeg}(v) = m \qquad v \in V$$

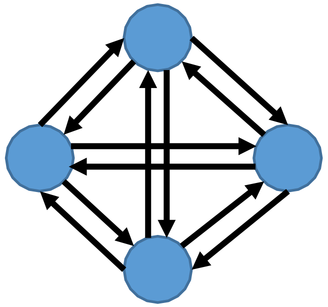


Theorem: Let G be a simple digraph with n vertices and m edges. Then,

$$m \leq n(n-1)$$



$\frac{n(n-1)}{2}$ edges

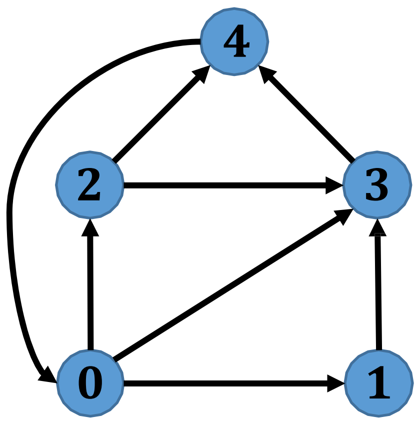


$n(n-1)$ edges

Corollary: A simple digraph with n vertices has $O(n^2)$ edges

Digraph Representation: Set of Edges

Maintain a list of directed edges (array or linked list)



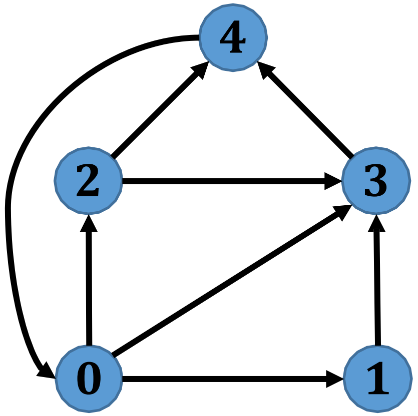
$(0, 1)$
$(0, 2)$
$(0, 3)$
$(1, 3)$
$(2, 3)$
$(2, 4)$
$(3, 4)$
$(4, 0)$

- Also have to store a separate list of vertices since some vertices have no edges

Digraph Representation: Adjacency Matrix

Maintain a **2-dimensional** $\boldsymbol{n} \times \boldsymbol{n}$ boolean array

- For each directed edge $(\boldsymbol{u}, \boldsymbol{v})$, i.e $\boldsymbol{u} \rightarrow \boldsymbol{v}$, $\mathbf{adj}[\boldsymbol{u}][\boldsymbol{v}] = \mathbf{true}\ (\mathbf{1})$

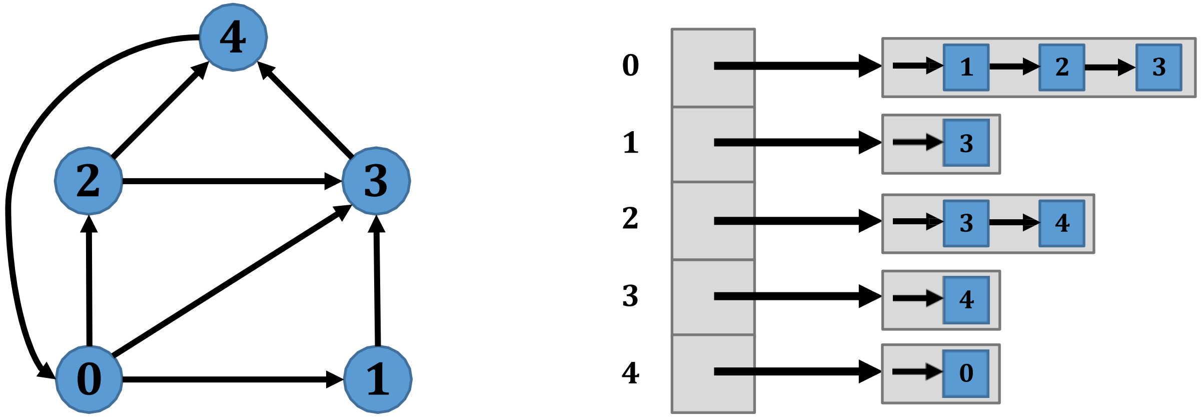


	Destination				
Source	0	1	2	3	4
0	0	1	1	1	0
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	0	0	0	0

- In undirected graphs, every edge appears twice. In directed graphs, each edge appears once.

Digraph Representation: Adjacency List

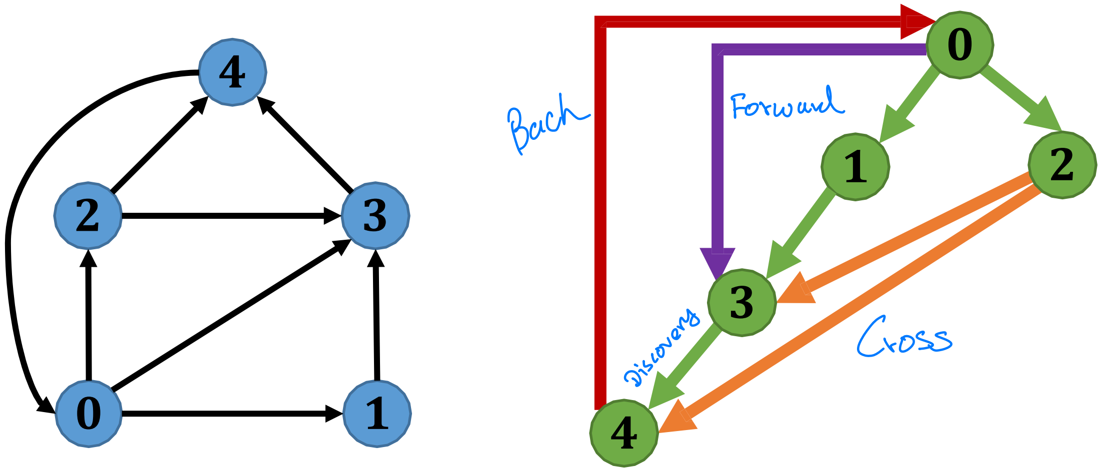
Maintain an array indexed by vertices which points to a list of adjacent vertices



- In undirected graphs, every edge appears twice. In directed graphs, each edge appears once.

Directed DFS

We can modify the traversal algorithms (DFS and BFS) for undirected graphs to directed graphs by traversing edges only along their direction.



In directed DFS, we have four types of edges:

- **Discovery edges** lead to unvisited nodes in the traversal and form a spanning tree
- **Back edges** go from nodes to one of its ancestors in the traversal discovery spanning tree
- **Forward edges** go from nodes to one of its descendants in the traversal discovery spanning tree
- **Cross edges** connect two nodes which do not have any ancestor and descendant relationship in the traversal discovery spanning tree

A directed DFS starting at a vertex s determines the vertices reachable from s

Directed DFS

$\boldsymbol{DirectedDFS}\ (\boldsymbol{G}, \boldsymbol{u})$:

Input: Directed graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery, back, forward, or cross edges

Label $\boldsymbol{u}$ as active

for each outgoing edge $\boldsymbol{e}$ **do**

    if $\boldsymbol{e}$ is unexplored **then**

        $\boldsymbol{v} \leftarrow$ destination vertex of $\boldsymbol{e}$

        if $\boldsymbol{v}$ is unexplored and not active **then**

            Label $\boldsymbol{e}$ as an explored *discovery* edge

            $\boldsymbol{DirectedDFS}(\boldsymbol{G}, \boldsymbol{v})$

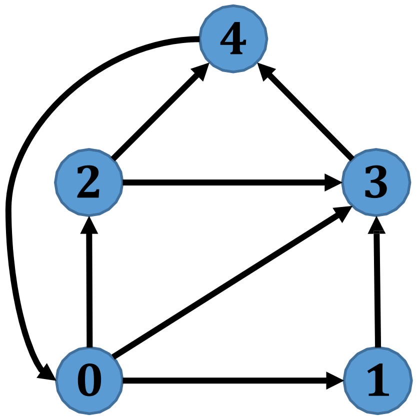
        else if $\boldsymbol{v}$ is active **then**

            Label $\boldsymbol{e}$ as an explored *back* edge

        else

            Label $\boldsymbol{e}$ as an explored *forward* / *cross* edge

Label $\boldsymbol{u}$ as explored



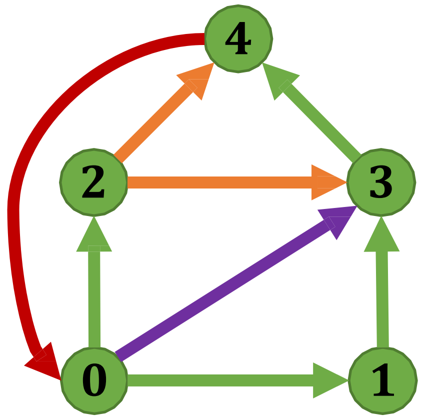
Directed DFS

$\boldsymbol{DirectedDFS}\ (\boldsymbol{G}, \boldsymbol{u})$:

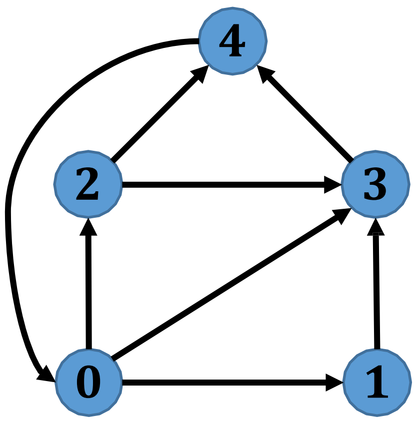
Input: Directed graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

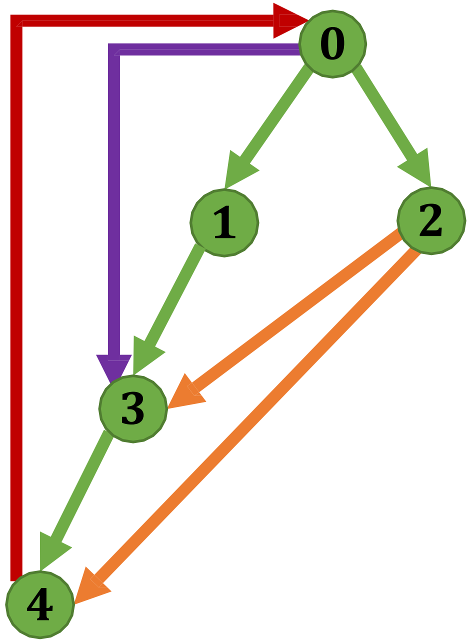
Output: Labeling of edges in the connected component as discovery, back, forward, or cross edges

Label $\boldsymbol{u}$ as active
for each outgoing edge $\boldsymbol{e}$ **do**
  if $\boldsymbol{e}$ is unexplored **then**
    $\boldsymbol{v} \leftarrow$ destination vertex of $\boldsymbol{e}$
    if $\boldsymbol{v}$ is unexplored and not active **then**
      Label $\boldsymbol{e}$ as an explored *discovery* edge
      $\boldsymbol{DirectedDFS}(\boldsymbol{G}, \boldsymbol{v})$
    else if $\boldsymbol{v}$ is active **then**
      Label $\boldsymbol{e}$ as an explored *back* edge
    else
      Label $\boldsymbol{e}$ as an explored *forward / cross* edge
Label $\boldsymbol{u}$ as explored



Directed DFS





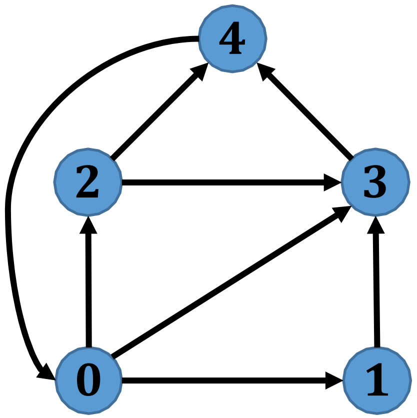
Discovery edges lead to unvisited nodes in the traversal and form a spanning tree

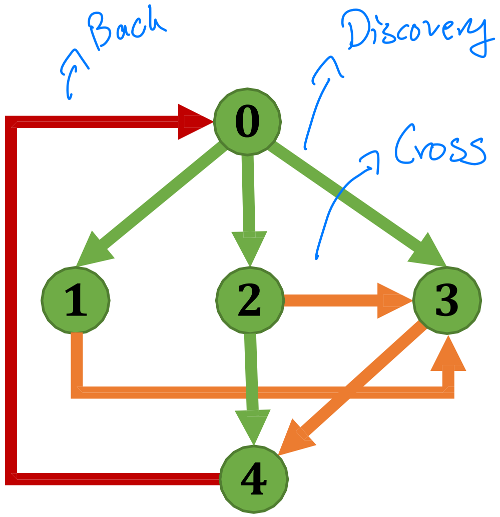
Back edges go from nodes to one of its ancestors in the traversal discovery spanning tree

Forward edges go from nodes to one of its descendants in the traversal discovery spanning tree

Cross edges connect two nodes which do not have any ancestor and descendant relationship in the traversal discovery spanning tree

Directed BFS



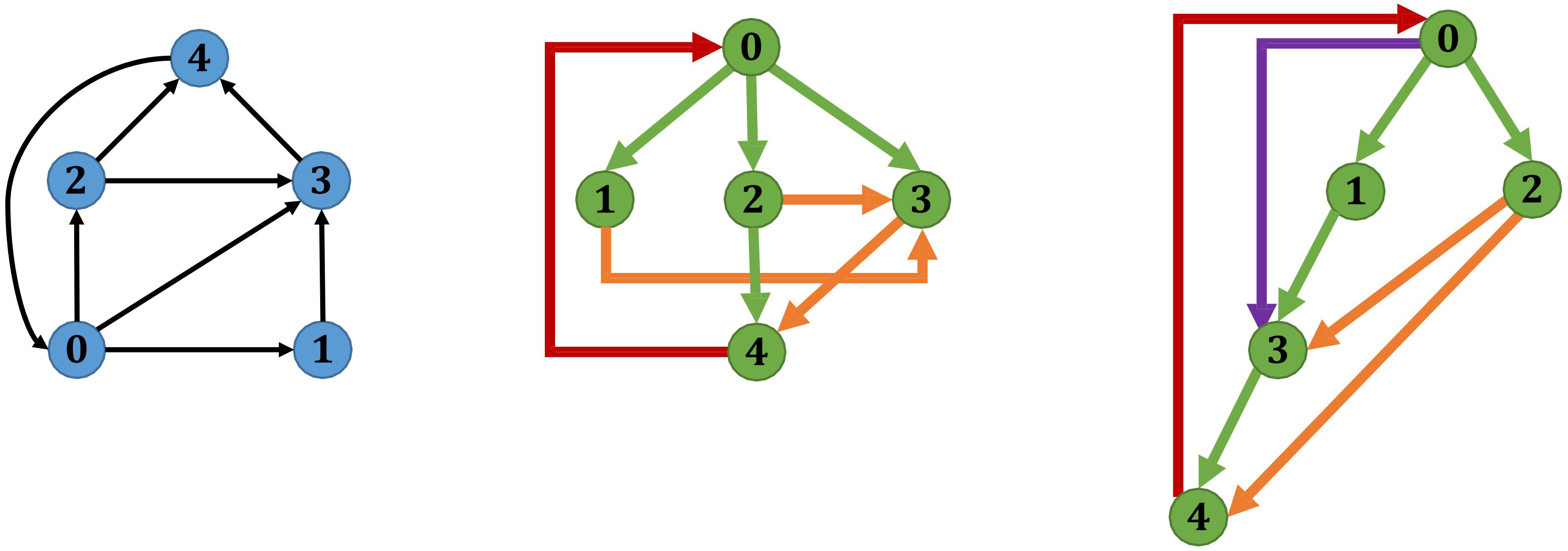


In directed BFS, we have three types of edges:

- **Discovery edges** lead to unvisited nodes in the traversal and form a spanning tree
- **Back edges** go from nodes to one of its ancestors in the traversal discovery spanning tree
- **Cross edges** connect two nodes which do not have any ancestor and descendant relationship in the traversal discovery spanning tree

A directed BFS starting at a vertex s determines the vertices reachable from s

Directed BFS



Why can we have forward edges in Directed DFS but not BFS?

Forward edges go from nodes to one of its descendants in the traversal discovery spanning tree

Directed BFS

$\boldsymbol{DirectedBFS}\ (\boldsymbol{G}, \boldsymbol{u})$:

Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery, back, or cross edges

$\boldsymbol{Q} \leftarrow$ new empty queue

Label $\boldsymbol{u}$ as explored

$\boldsymbol{Q}$.enqueue($\boldsymbol{u}$)

while $\boldsymbol{Q}$ is not empty **do**

  $\boldsymbol{u} \leftarrow \boldsymbol{Q}$.dequeue()

  for each outgoing edge $\boldsymbol{e}$ **do**

    $\boldsymbol{v} \leftarrow$ destination vertex of $\boldsymbol{e}$

    if $\boldsymbol{e}$ is unexplored **then**

      if $\boldsymbol{v}$ is unexplored **then**

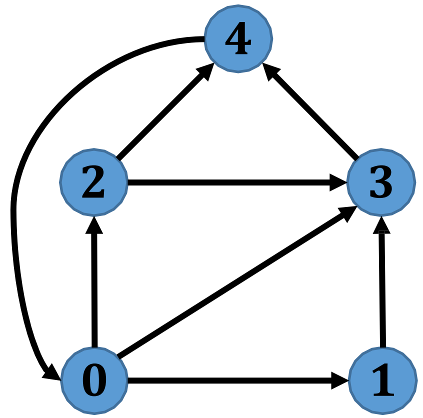
        Label $\boldsymbol{e}$ as an explored *discovery* edge

        Mark $\boldsymbol{v}$ as explored

        $\boldsymbol{Q}$.enqueue($\boldsymbol{v}$)

      else

        Label $\boldsymbol{e}$ as an explored *back* / *cross* edge



Directed BFS

$\boldsymbol{DirectedBFS}\ (\boldsymbol{G}, \boldsymbol{u})$:

Input: Graph $\boldsymbol{G}$ and vertex $\boldsymbol{u}$ of $\boldsymbol{G}$

Output: Labeling of edges in the connected component as discovery, back, or cross edges

$\boldsymbol{Q} \leftarrow$ new empty queue

Label $\boldsymbol{u}$ as explored

$\boldsymbol{Q}$.enqueue($\boldsymbol{u}$)

while $\boldsymbol{Q}$ is not empty **do**

    $\boldsymbol{u} \leftarrow \boldsymbol{Q}$.dequeue()

    for each outgoing edge $\boldsymbol{e}$ **do**

        $\boldsymbol{v} \leftarrow$ destination vertex of $\boldsymbol{e}$

        if $\boldsymbol{e}$ is unexplored **then**

            if $\boldsymbol{v}$ is unexplored **then**

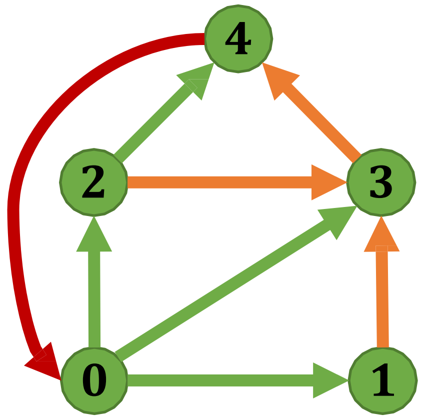
                Label $\boldsymbol{e}$ as an explored *discovery* edge

                Mark $\boldsymbol{v}$ as explored

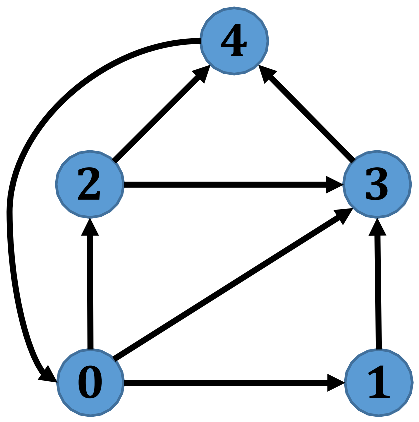
                $\boldsymbol{Q}$.enqueue($\boldsymbol{v}$)

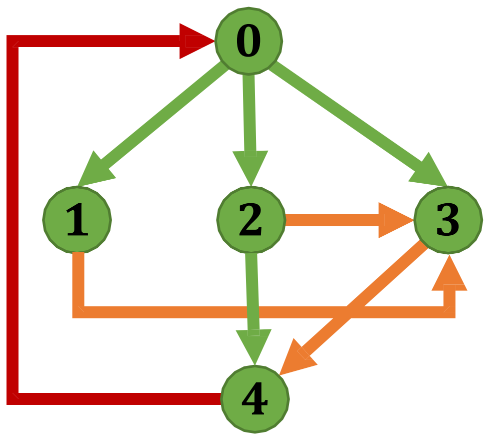
            else

                Label $\boldsymbol{e}$ as an explored *back* / *cross* edge



Directed BFS Tree





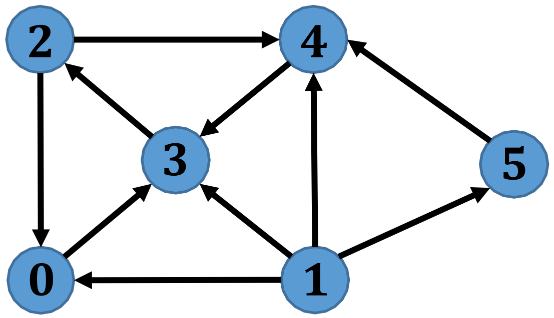
Discovery edges lead to unvisited nodes in the traversal and form a spanning tree

Back edges go from nodes to one of its ancestors in the traversal discovery spanning tree

Cross edges connect two nodes which do not have any ancestor and descendant relationship in the traversal discovery spanning tree

Reachability

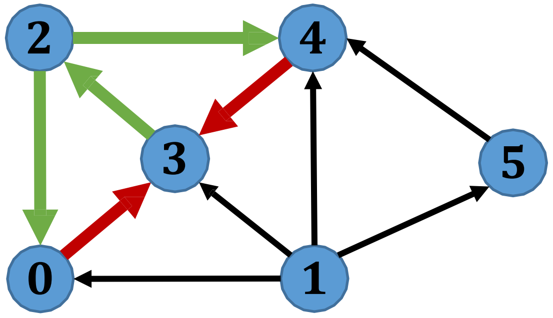
Given vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ of a digraph, we say $\boldsymbol{v}$ is **reachable** from $\boldsymbol{u}$ if $\boldsymbol{G}$ has a directed path from $\boldsymbol{u}$ to $\boldsymbol{v}$

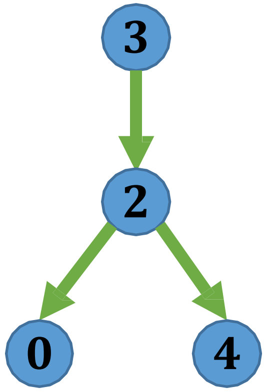


Example: $\mathbf{0}, \mathbf{2}$, and $\mathbf{4}$ are reachable from $\mathbf{3}$

Can determine the vertices reachable from a vertex $\boldsymbol{u}$ by running directed DFS or BFS starting at $\boldsymbol{u}$. If v is in this search then it is reachable.

E.g. $\boldsymbol{DirectedDFS}(\boldsymbol{G}, \mathbf{3})$:





Graph Connectivity

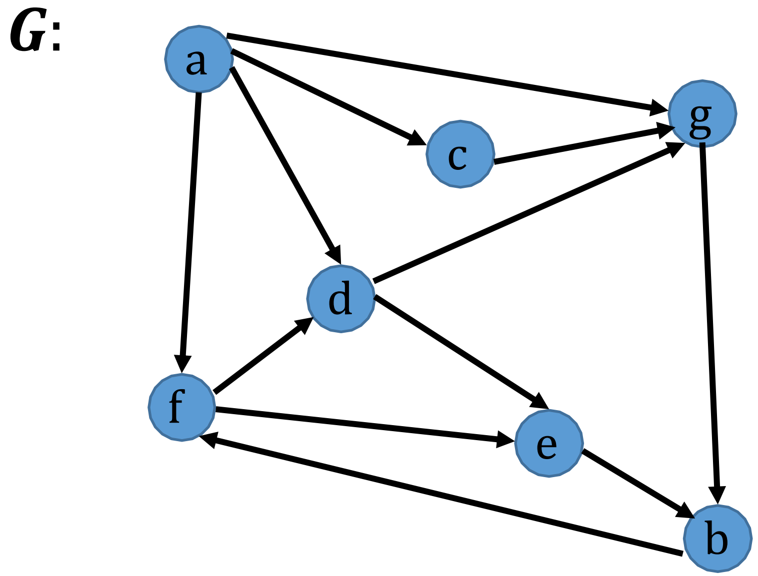
There is a connection between all nodes, but the direction need not be correct. You treat all edges as if they don't have direction

A Digraph is **connected** if every pair of vertices is connected by an undirected path

Every node is reachable by every node, taking direction into account.

A digraph G is **strongly connected** if for every pair of vertices u and v of G, $u$is reachable from v and v is reachable from u.

- How can we determine if a graph is strongly connected?



Strong Connectivity

A Graph is strongly connected if each vertex can reach all other vertices.

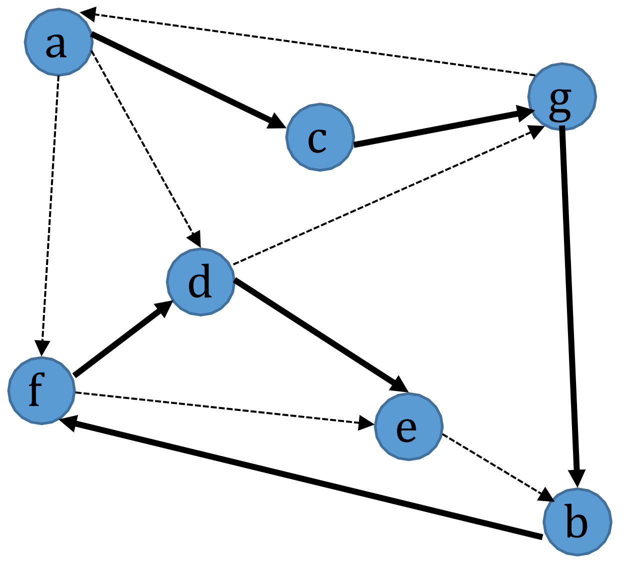
- How can we find out if a graph is strongly connected?

CheckStrongConnectivity **(G, v):**
Input: Graph G and vertex v of G
Output: "yes" if G is strongly connected, otherwise "no“

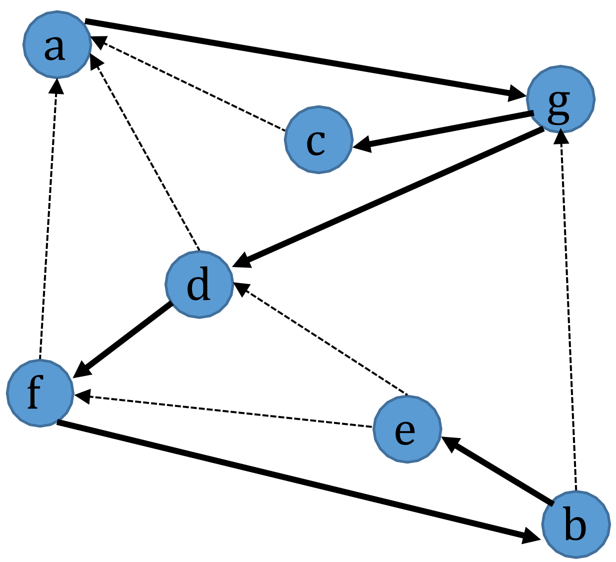
Perform a DFS from v in G
    If there exists a vertex w that is not visited during the DFS:
        Output **"no"** and terminate
Reverse the edges of G to obtain G^r
Perform a DFS from v in G^r
    If there exists a vertex w that is not visited during the DFS:
        Output **"no"** and terminate
    Otherwise:
        Output **"yes“**

- **Running time: $O(n + m)$**

G:



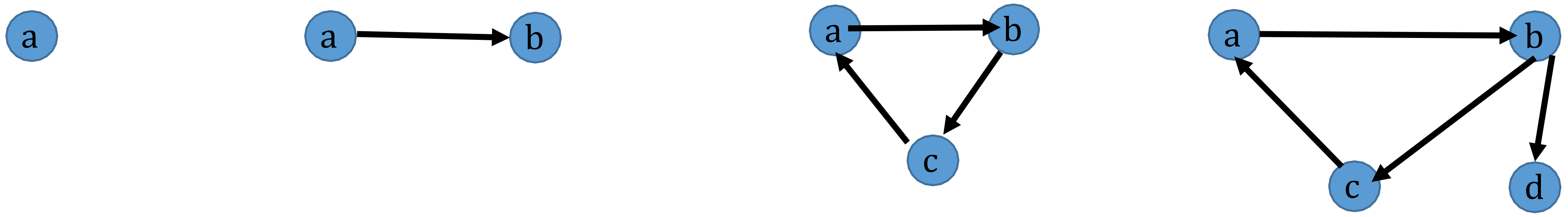
G^r :



Strongly Connected Components

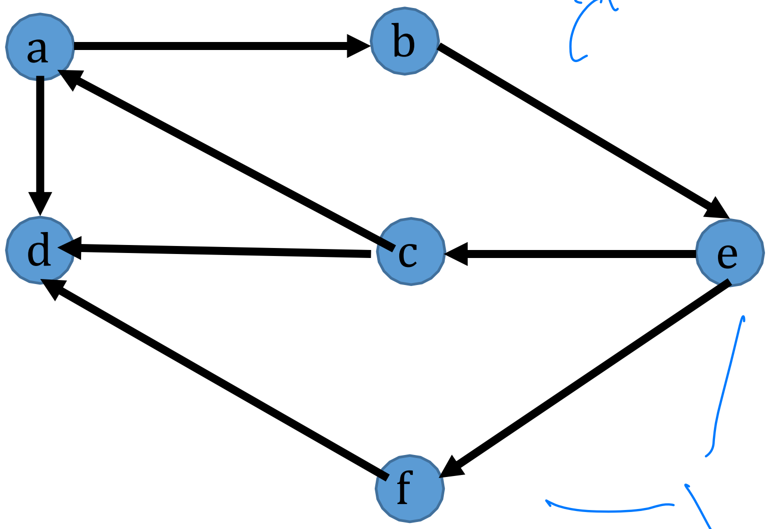
A strongly connected component (SCC) of a directed graph is a maximal set of nodes in which there is a path from any node in the set to any other node in the set. Aka a subgraph that is strongly connected.

Identifying the SCC of a graph is a very important preprocessing step for many algorithms (e.g., topological sort)



An SCC Example

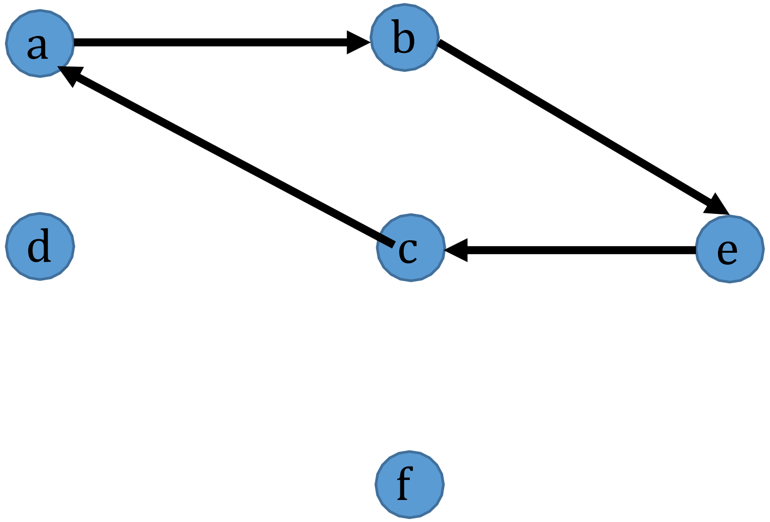
Not SCC



$G = (V, E)$

no path from f to e.

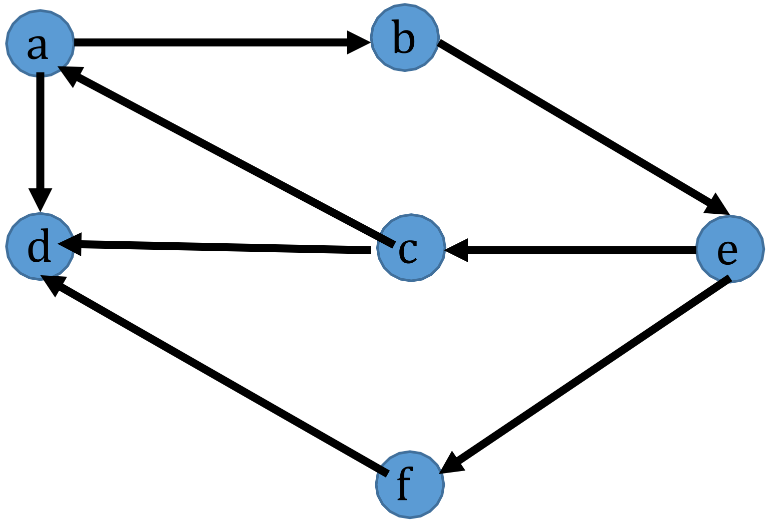
SCC



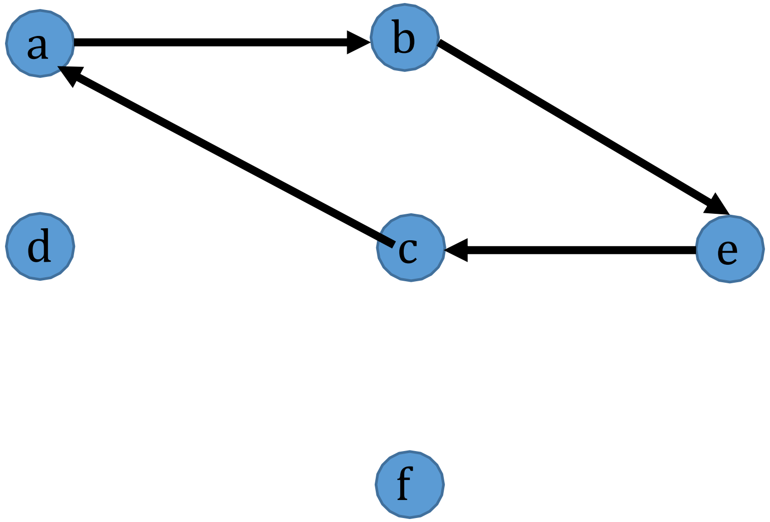
$G_K = (G_K, E_K)$

How to Find The SCCs?

- Can we use DFS?
- Where should we start from?
- What if we start a dfs from d then f and then e?
- Can we start from e?



$G = (V, E)$

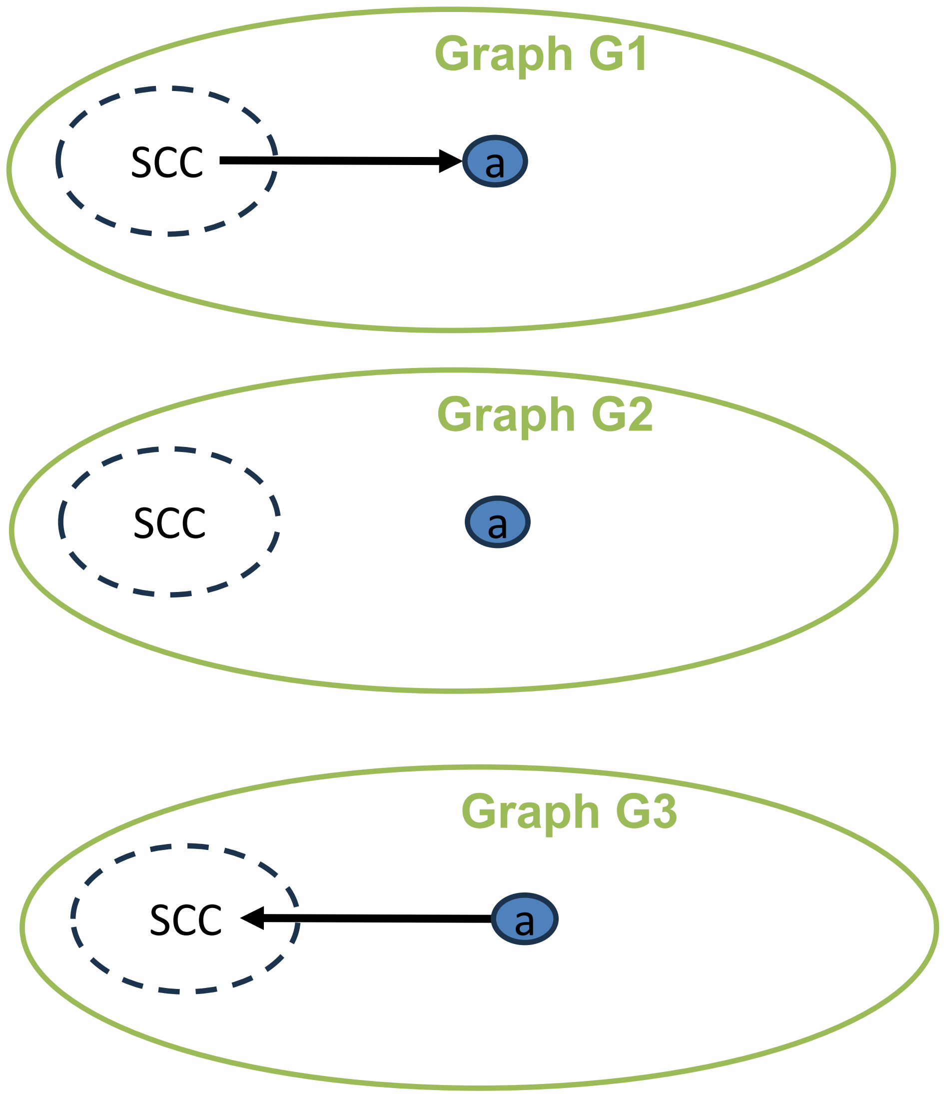


$G_K = (G_K, E_K)$

How to Find The SCCs?

Consider these cases:

- In which case starting from a node in SCC would result in a wrong answer?



Lemma

Lemma: Let *C* be a strong component in a digraph *G* and let v be any vertex not in *C*. Prove that if there is an edge *e* pointing from v to any vertex in *C*, then vertex v appears before *every* vertex in *C* in the reverse post-order of G.

Proof : If v is visited before every vertex in *C*, then every vertex in *C* will be visited and finished before v finishes (because every vertex in *C* is reachable from v via edge *e*). If some vertex in *C* is visited before v, then all vertices in *C* will be visited and finished before v is visited (because v is not reachable from any vertex in *C*—if it were, such a path when combined with edge *e* would be part of a directed cycle, implying that v is in *C*).

Lemma - Continued

- Let C be a strong component in a digraph G and let v be any vertex not in C. If there is an edge e pointing from any vertex in C to v, then vertex v appears before *every* vertex in C in the reverse post-order of G^R. Why?

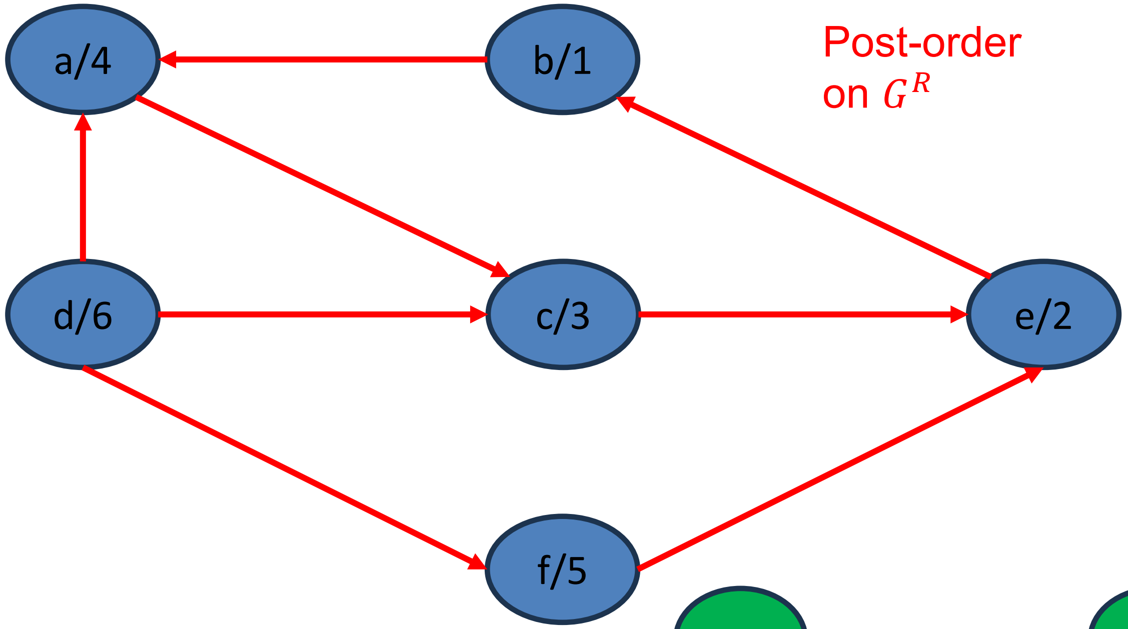
- Hint: Apply the lemma to G^R.

SCC Algorithm – Kosoraju's Algorithm

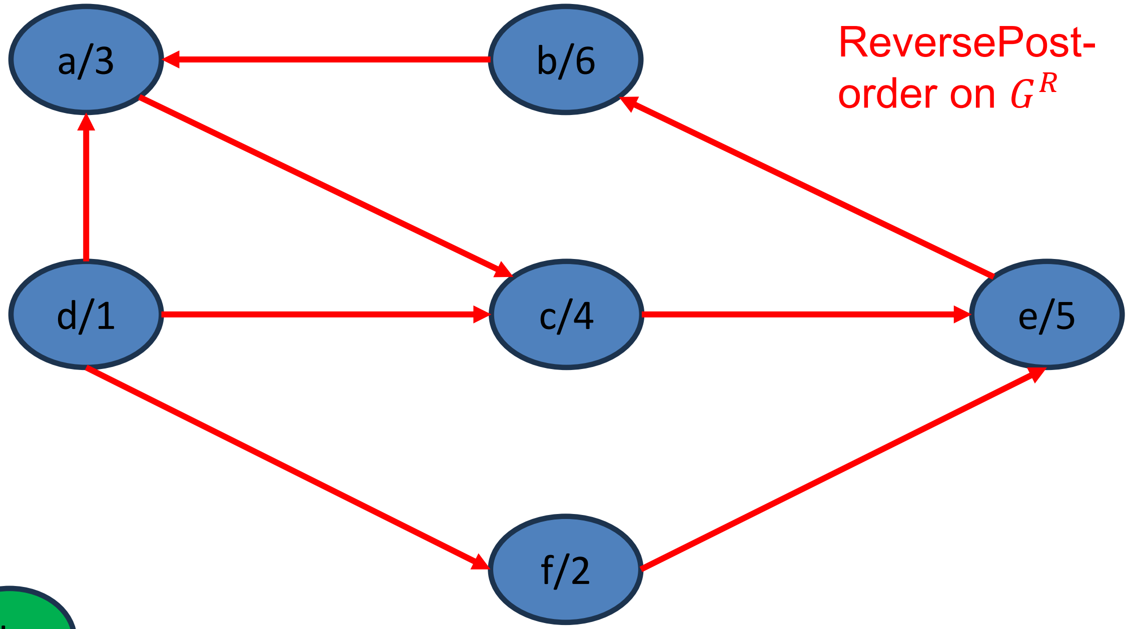
Algorithm 1 Compute Strongly Connected Components (SCCs)

Require: Directed graph $G = (V, E)$
Ensure: Strongly connected components (SCCs) of G

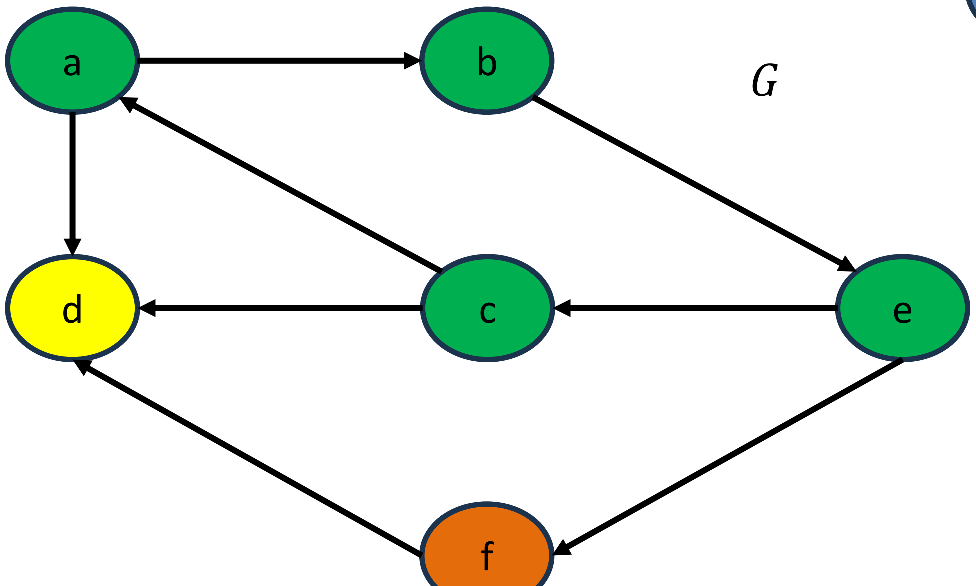
Construct the reversed graph $G_R = (V, E_R)$:
for all edges $(u, v) \in E$ **do**
    Add edge (v, u) to E_R.
end for
Use DepthFirstOrder to compute the reverse postorder of G_R.
Initialize all vertices in G as unmarked.
for all vertices v in the order of reverse postorder from G_R **do**
    if v is unmarked **then**
        Perform standard DFS on G starting from v.
        Mark all vertices visited during this DFS as part of the same strong component.
    end if
end for



Post-order on G^R



ReversePost-order on G^R



G