Algorithm Analysis and Data Structures

CSCI 7432 – Fall 2022

DFS-Based Algorithms

Dr. Yao XU

Assistant Professor

Department of Computer Science

Georgia Southern University

Email: yxu@georgiasouthern.edu

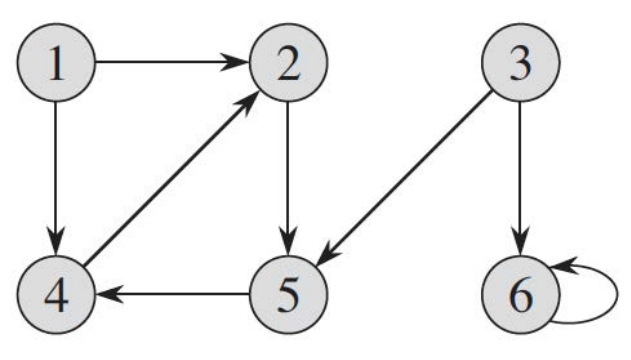
Table of Contents

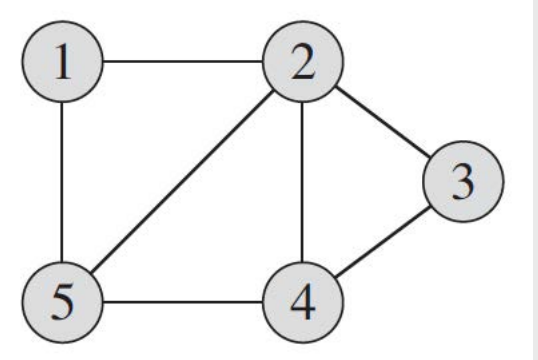
Review: Graphs

Basic Graph Definitions

Directed and Undirected Graphs (1/2)

- A ***graph*** $G = (V, E)$ consists of a ***vertex set*** V of $n = |V|$ vertices/nodes and an ***edge set*** E of $m = |E|$ edges.
  - ***Directed graph*** (***digraph***): Edge (u, v) goes from vertex u to vertex v.
    - (u, v) is an outgoing edge for u and an incoming edge for v.
    - (v, v) is a ***self-loop*** goes from v to itself.
  - ***Undirected graph***: (u, v) and (v, u) are the same edge, with $u \neq v$.
- **Examples:**





Directed and Undirected Graphs (2/2)

- In an **undirected graph**, the ***degree*** of a vertex v is:

$$\deg(v) = \text{the number of edges that touch } v$$

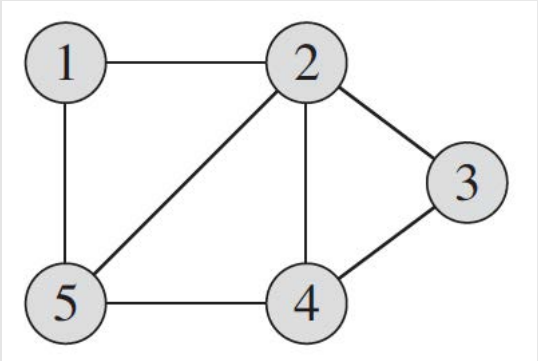
- In a **directed graph**, the ***degree*** of a vertex v is:

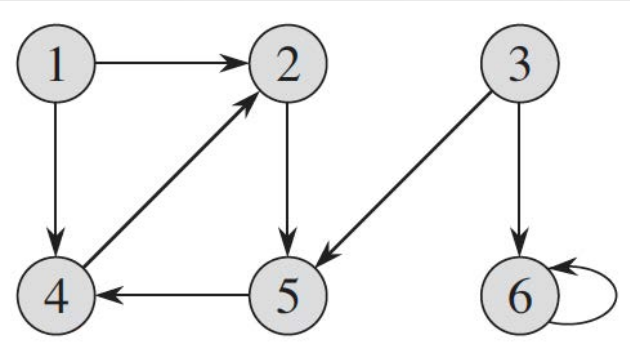
$$\deg(v) = \deg^-(v) + \deg^+(v),$$

where

  - the ***in-degree*** of v is $\deg^-(v) =$ the number of incoming edges for v
  - the ***out-degree*** of v is $\deg^+(v) =$ the number of edges outgoing from v

- **Examples:**

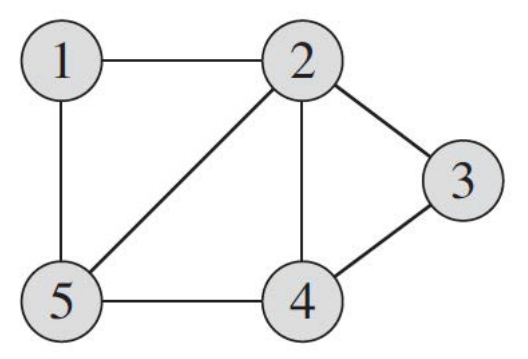




Simple Path and Cycle

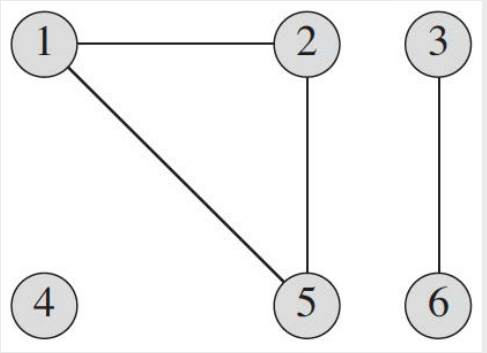
- A ***path*** is a sequence of vertices $v_0, v_1, \cdots, v_k$ such that there exists an edge (v_{i-1}, v_i) for $i = 1, 2, \cdots, k$.
- A ***simple path*** is a path where all vertices in the path are distinct.
- A ***cycle*** is a path with $v_0 = v_k$.
- A ***simple cycle*** is a cycle where all vertices $v_1, v_2, \cdots, v_k$ are distinct.
- A graph with no cycles is called an ***acyclic graph***.
- Unless specified otherwise, we assume
  - No multiple edges connecting two vertices
  - A path is a simple path
  - A cycle is a simple cycle

Example:

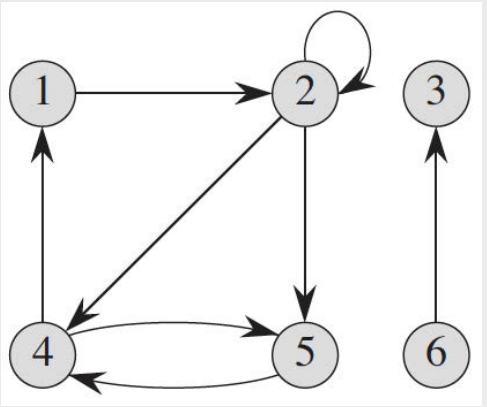


Connectivity in Graphs

- An **undirected graph** $G = (V, E)$ is called ***connected*** if for every pair of vertices $u, v \in V$, there exists a path from u to v.
  - A **tree** is a connected acyclic graph.
- A ***connected component*** of an undirected graph G is a maximal connected subgraph of G.
  - **Example**: the subgraph on vertex set $\{1, 2, 5\}$



- A **directed graph** $G = (V, E)$ is called ***strongly-connected*** if for every pair of vertices $u, v \in V$, there exist a path from u to v and a path from v to u.
- A ***strongly connected component*** of a directed graph G is a maximal strongly connected subgraph of G.
  - **Example**: the subgraph on vertex set $\{1, 2, 4, 5\}$



Review: Graphs

Representations of Graphs

Graph Representation

- Given graph $G = (V, E)$, either directed or undirected, we represent the vertex set V by $G.V$ and edge set E by $G.E$ in pseudocode.
- Two common ways to represent graphs for algorithms:
  - ***Adjacency lists***
  - ***Adjacency matrix***
- When expressing the **running time** of a graph algorithm, it's often in terms of both $n = |V|$ and $m = |E|$.
- In asymptotic notation — and only in asymptotic notation — we may drop the cardinality.
  - **Example:** $O(V + E) = O(n + m)$

Graph Representation: Adjacency Matrix

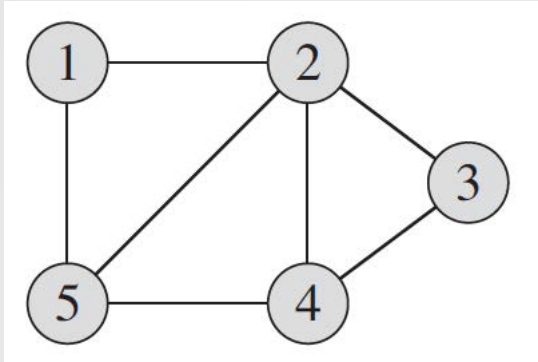
Represent $G = (V, E)$, with $n = |V|$ and $m = |E|$, by

- A $n \times n$ matrix $A = \{a_{ij}\}$
- $a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$

Advantage: can check if some edge $(u, v) \in E$ in $\Theta(1)$ time.

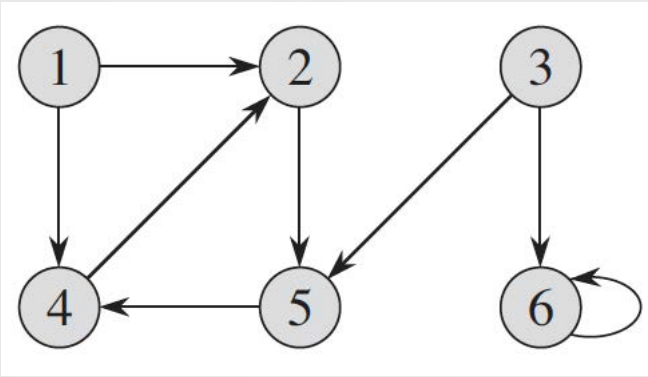
Disadvantage: requires $\Theta(n^2)$ space even when $m \ll n^2$.

Example 1:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Example 2:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Graph Representation: Adjacency Lists (1/2)

Represent $G = (V, E)$, with $n = |V|$ and $m = |E|$, by

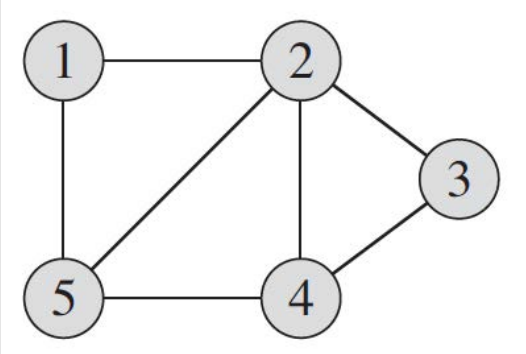
- An array Adj of n lists.
- List $Adj[u]$ has all vertices v such that $(u, v) \in E$.

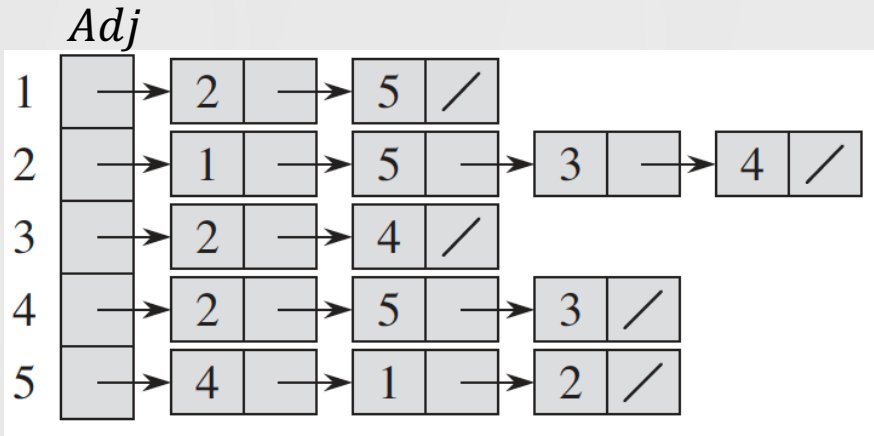
Disadvantage: requires $O(|\, Adj(u)\,|)$ time to check if $(u, v) \in E$.

In an undirected graph, total length of the n lists $= \sum_{u \in V} |Adj(u)| = 2m$.

Advantage: $\Theta(n + m)$ space, better than $\Theta(n^2)$ when $m \ll n^2$.

Example 1:





Graph Representation: Adjacency Lists (2/2)

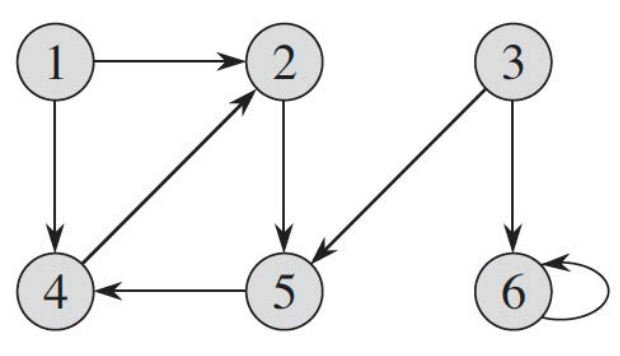
Represent $G = (V, E)$, with $n = |V|$ and $m = |E|$, by

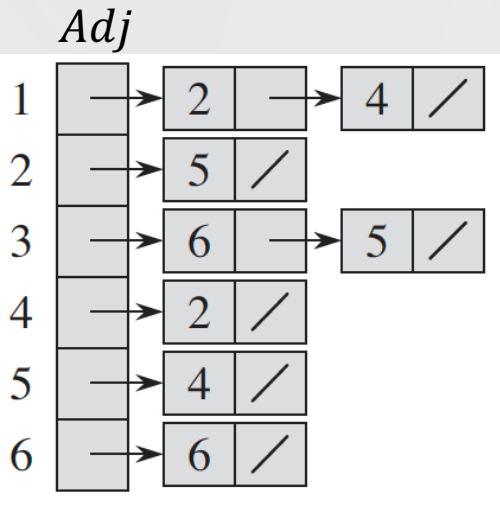
- An array Adj of n lists.
- List $Adj[u]$ has all vertices v such that $(u, v) \in E$.

In a digraph, total length of the n lists $= \sum_{u \in V} |Adj(u)| = m$.

Space is still $\Theta(n + m)$.

Example 2:





Note: *In this course, unless specified otherwise, we assume graphs are represented by adjacency lists.*

Review: Graph Traversal

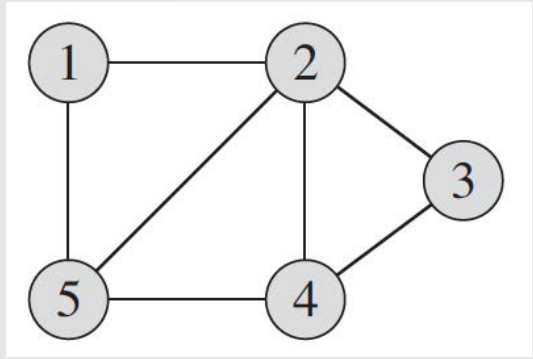
Breadth-First Search (BFS)

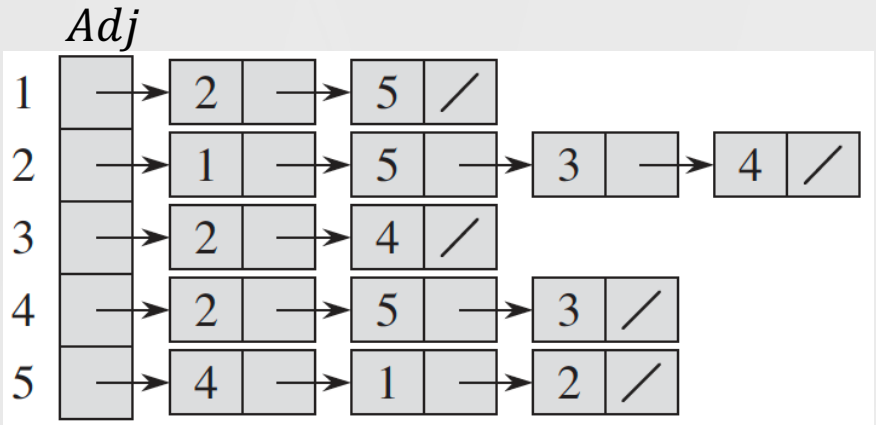
Breadth-First Search (1/2)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Three attributes for each vertex:

- $v.color$ = WHITE (unvisited) or GRAY (discovered) or BLACK (finished)
- $v.d$ = distance (smallest #edges) from s to v
- $v.\pi$ = the predecessor of v in the ***breadth-first tree***
- **Example:**





v	1	2	3	4	5
$v.\pi$	NIL	1	2	2	1
$v.d$	0	1	2	2	1

Breadth-First Search (2/2)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

BFS(G)

for each vertex $u \in G.V$
  $u.color$ = WHITE
  $u.d = \infty$
  $u.\pi$ = NIL
for each $u \in G.V$
  if $u.color$ == WHITE
    BFS-VISIT(G, u)

BFS-VISIT(G, s)

// lines 5-18 of BFS(G, s)

Running time of BFS(G):

- Using adjacency matrix:

$$\Theta\left(n + \sum_{v \in V} n = n + n^2\right)$$
$$= \Theta(n^2)$$

- Using adjacency lists:

$$\Theta\left(n + \sum_{v \in V} \deg(v) = n + 2m\right)$$
$$= \Theta(n + m)$$

Warning: By BFS(G, s), vertices in other connected components wouldn't be discovered!!!

Review: Graph Traversal

Depth-First Search (DFS)

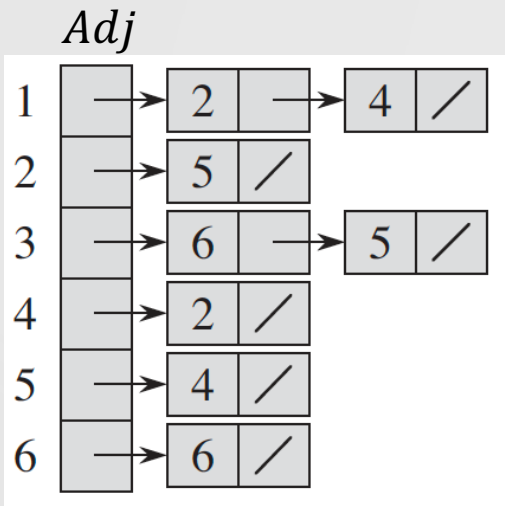
Depth-First Search (1/2)

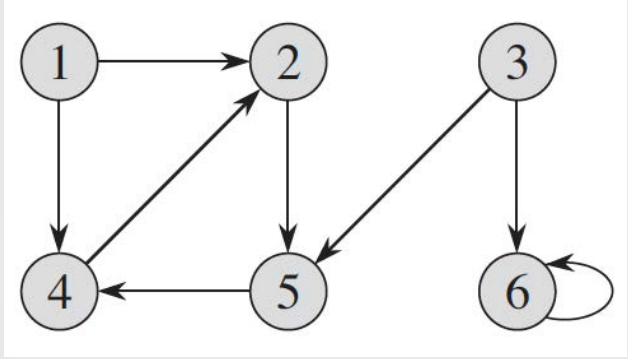
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

Three attributes for each vertex:

- $v.color$ = WHITE (unvisited) or GRAY (discovered) or BLACK (finished)
- $v.\pi$ = the predecessor of v in the ***depth-first tree/forest***
- $v.d$ = discovery time of v
- $v.f$ = finish time of v
- **Example:**





v	1	2	3	4	5	6
$v.\pi$	NIL	1	NIL	5	2	3
$v.d$	1	2	9	4	3	10
$v.f$	8	7	12	5	6	11

Depth-First Search (2/2)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

- **Running time** of $\mathrm{DFS}(G)$:
  - Using adjacency matrix:

$$\Theta\left(n + \sum_{v \in V} n = n + n^2\right) = \Theta(n^2)$$

  - Using adjacency lists:

$$\Theta\left(n + \sum_{v \in V} \deg(v) = n + 2m\right) = \Theta(n + m)$$

Review: Graph Traversal

Classification of Edges

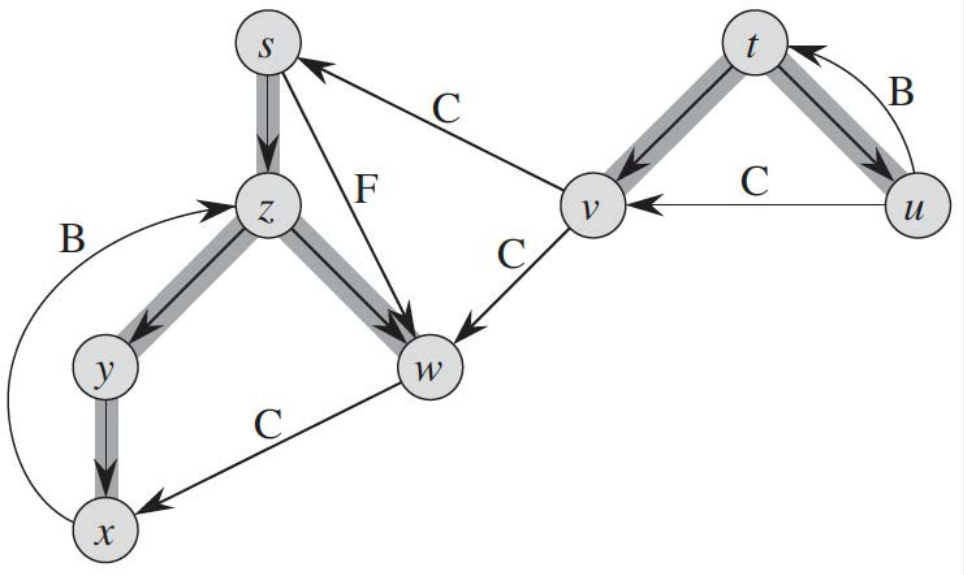
Classification of Edges

Based on the breadth-first tree/forest or depth-first tree/forest, edges in the original graph can be classified into the following four types.

1. ***Tree edge***: an edge (u, v) in the breadth-first or depth-first tree/forest
2. ***Back edge***: a non-tree edge (u, v) where v is an ancestor of u in the tree, including self-loops in digraphs
3. ***Forward edge***: a non-tree edge (u, v) where v is a descendent of u in the tree

   "back" = "forward" in undirected graphs.
4. ***Cross edge***: any other edge

Example: DFS for a digraph



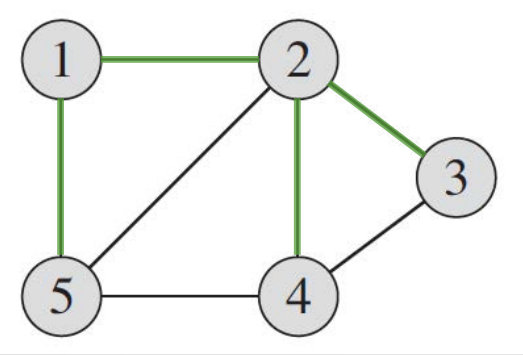
- Bold gray: Tree edge
- B: Back edge
- F: Forward edge
- C: Cross edge

BFS Edges

BFS for **undirected** graphs:

- Have only tree and cross edges
- No back/forward edges

Example:

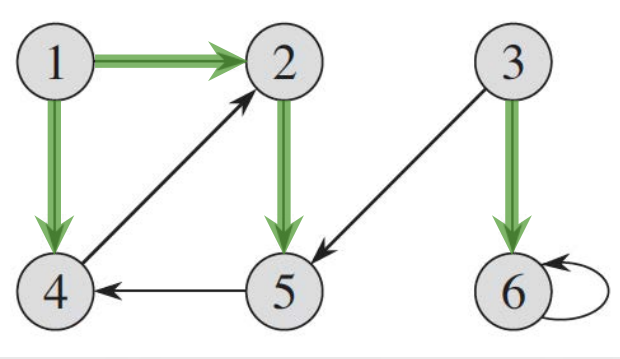


Tree edges: (1, 2), (1, 5), (2, 3), (2, 4)
Cross edges: (2, 5), (3, 4), (4, 5)

BFS for **directed** graphs:

- Have only tree, back, and cross edges
- No forward edges

Example:



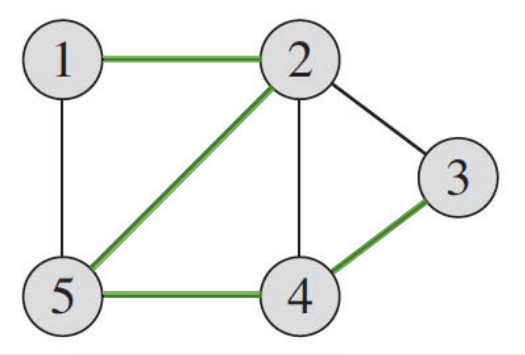
Tree edges: (1, 2), (1, 4), (2, 5), (3, 6)
Back edge: (6, 6)
Cross edge: (3, 5), (4, 2), (5, 4)

DFS Edges

DFS for **undirected** graphs:

- Have only tree and back/forward edges
- No cross edges

Example:

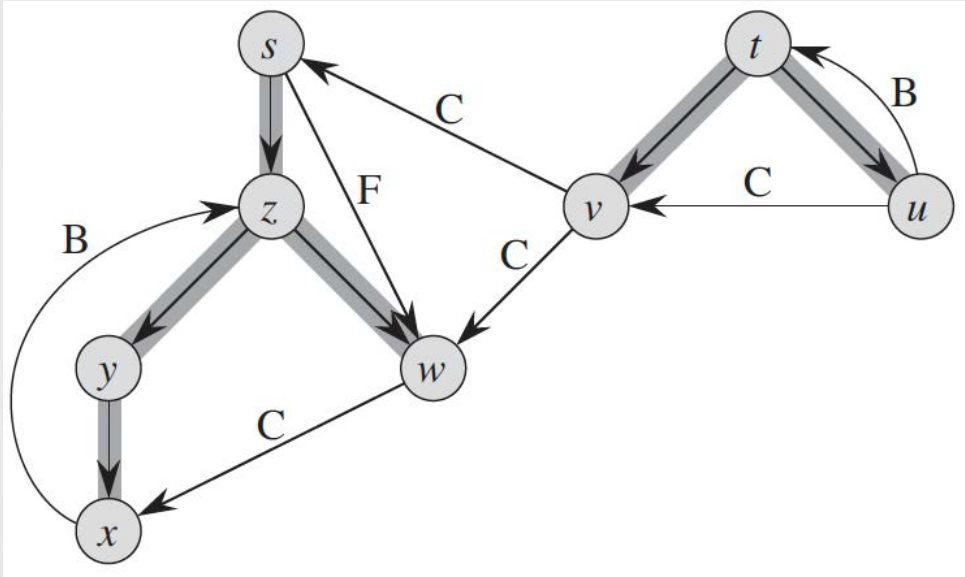


Tree edges: (1, 2), (2, 5), (3, 4), (4, 5)
Back/Forward edges: (1, 5), (2, 3), (2, 4)

DFS for **directed** graphs:

- Can have all four types of edges

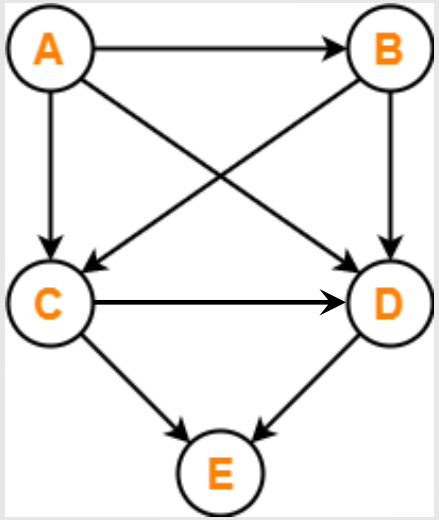
Example:



Topological Sort

Topological Sort

- ***Topological Sort*** (of a graph): A linear ordering of vertices such that if edge $(u, v) \in E$, then u appears somewhere before v.
- If a directed graph has a **topological sort**, we can find an assignment of a unique number $\{1, 2, \cdots, n\}$ to each vertex such that all edges are heading forward.
- **Example:**



Topological sort:

- A B C D E

Topological Sort of A DAG

- ***Directed Acyclic Graph (DAG)***: A directed graph with no cycles.

Theorem: G is a DAG iff G has a topological sort.

Proof.

- "$\Leftarrow$":
  - If G has a topological sort, we can find an assignment of a unique number $\{1, 2, \cdots, n\}$ to each vertex in G such that all edges are heading forward.
  - So, there must be no cycle in $G \Rightarrow G$ is a DAG.
- "$\Rightarrow$":*
  - Algorithm TOPOLOGICAL-SORT will find a topological sort of a DAG.

** For "⇒", see the complete proof of Theorem 22.12 on p.614 of the textbook.*

Find Topological Sort

TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times $v.f$ for each vertex $v \in G.V$
   - During DFS, as each vertex is finished, insert it onto the front of a linked list.
2. **return** the linked list of vertices

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

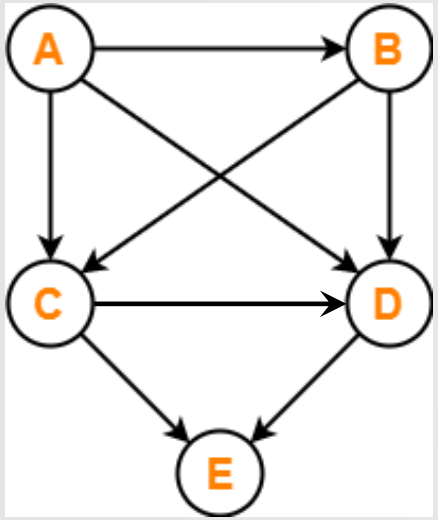
- There is no need to perform a sorting algorithm for decreasing finishing times.
- It can be done as part of DFS.
- **Running time:** $\Theta(n + m)$ (same as DFS)

Topological Sort Example

TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times $v.f$ for each vertex $v \in G.V$
   - During DFS, as each vertex is finished, insert it onto the front of a linked list.
2. **return** the linked list of vertices

Example:



Correctness of Topological Sort

Theorem. Algorithm TOPOLOGICAL-SORT produces a topological sort of the input DAG G.*

Proof. Show that for every edge $(u, v) \in E$, $u.f > v.f$.

Case 1: $u.d < v.d$

- Based on DFS, v becomes BLACK before u.
- Then, we have $u.f > v.f$

Case 2: $u.d > v.d$

- G has no cycle, so there is no path from v to u.
- u will not be visited by DFS-VISIT(G, v).
- $u.d > v.d$ implies $u.f > v.f$

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

** See Theorem 22.12 on p.614 of the textbook.*

Topological Sort

Longest Path in a DAG

Longest Path in a DAG

- Given a DAG G, define $L(G, v)$ as the length of the longest path in G starting with v.
- Then the length of the **longest path** in G will be

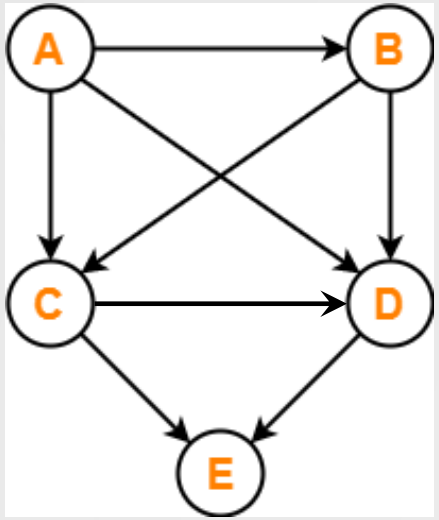
$$\max_{v \in V} \{L(G, v)\}$$

- If S is a topological sort of G and u is the first vertex in S, then

$$L(G, u) = \max_{v \in N^+(u)} \{L(G, v)\} + 1,$$

where $N^+(u)$ is the set of out-neighbors of u.

Example:



DP for Longest Path in DAG (1/3)

Step 1: Find a recurrence relation

- The length of the longest path in G starting with u is

$$L(G,u) = \begin{cases} 0, & \text{if } N^+(u) = \emptyset \\ 1 + \max\limits_{v \in N^+(u)} \{L(G,v)\}, & \text{otherwise} \end{cases}$$

Step 2: Count #distinct recursive calls — $n = |V|$

Step 3: Define an array L of size n.

- $L[k]$ will hold the value of $L(G, v_k)$, where v_k is the k-th vertex in the topological sort of G.
- Fill the array L according to the recurrence relation.
- The largest element in L will be the length of the longest path in G.

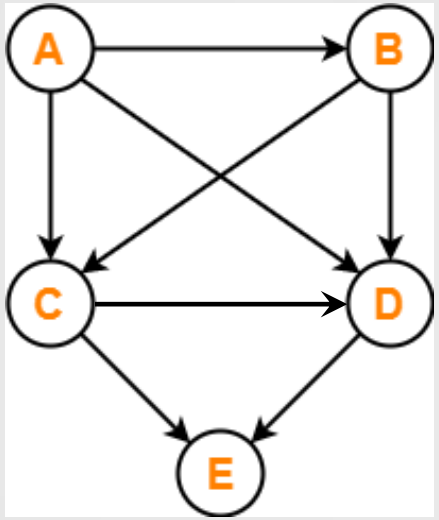
DP for Longest Path in DAG (2/3)

Step 3 (cont'd): Fill the array L according to the following recurrence

$$L[u] = \begin{cases} 0, & \text{if } N^+(u) = \emptyset \\ 1 + \max\limits_{v \in N^+(u)} L[v], & \text{otherwise} \end{cases}$$

```
LONGEST-PATH-DAG(G)
1  S = TOPOLOGICAL-SORT(G)
2  for each v in downward order of S
3      L[v] = 0
4      for each u ∈ N+[v]
5          if 1 + L[u] > L[v]
6              L[v] = 1 + L[u]
```

Example:



v	A	B	C	D	E
$L[v]$					

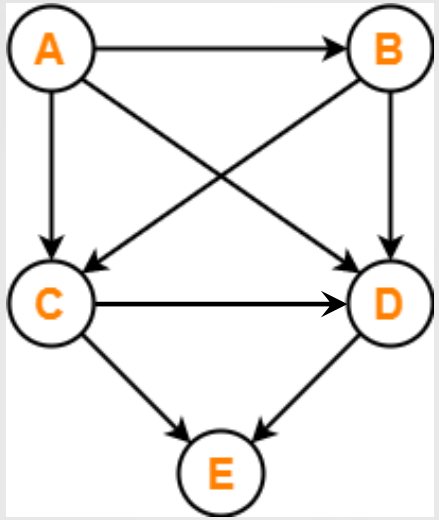
Running time: $\Theta(n+m)$

DP for Longest Path in DAG (3/3)

Step 4: Extract a longest path from array L.

```
PRINT-LONGEST-PATH-DAG(G, S, L)
1  i = FIND-MAX(L) // return the index
2  v = S[i]
3  while v ≠ NIL
4       PRINT(v)
5       nextnode = NIL
6       for each u ∈ N⁺[v]
7            if L[v] == 1 + L[u]
8                 nextnode = u
9       v = nextnode
```

Example:



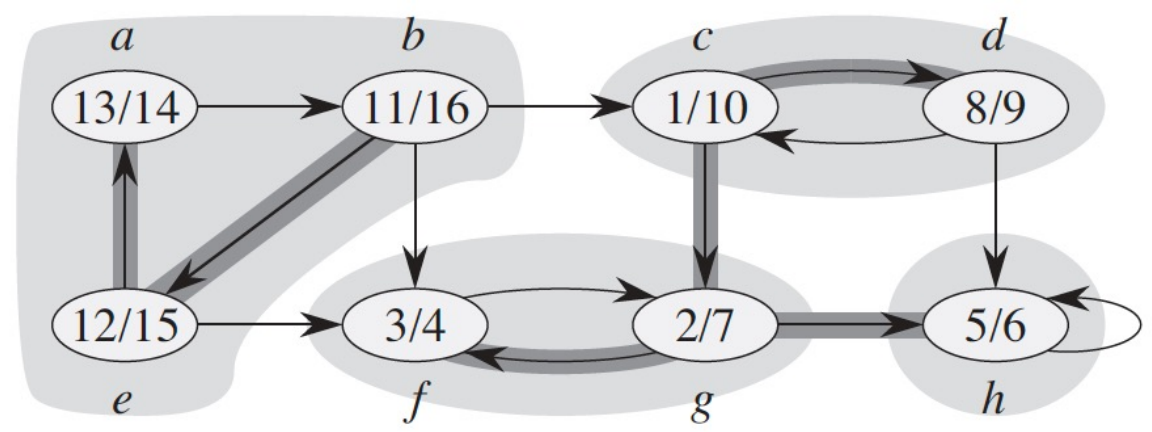
v	A	B	C	D	E
$L[v]$	4	3	2	1	0

Running time: $\Theta(n+m)$

Strongly Connected Components

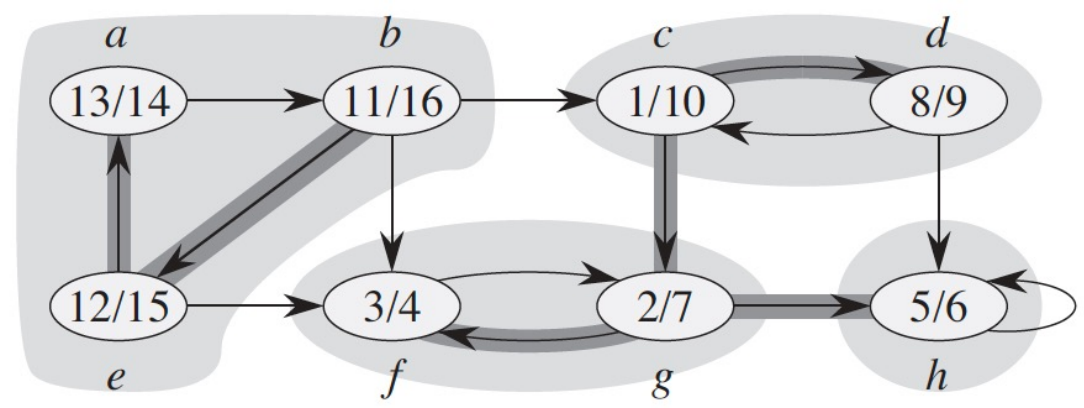
Strongly Connected Components

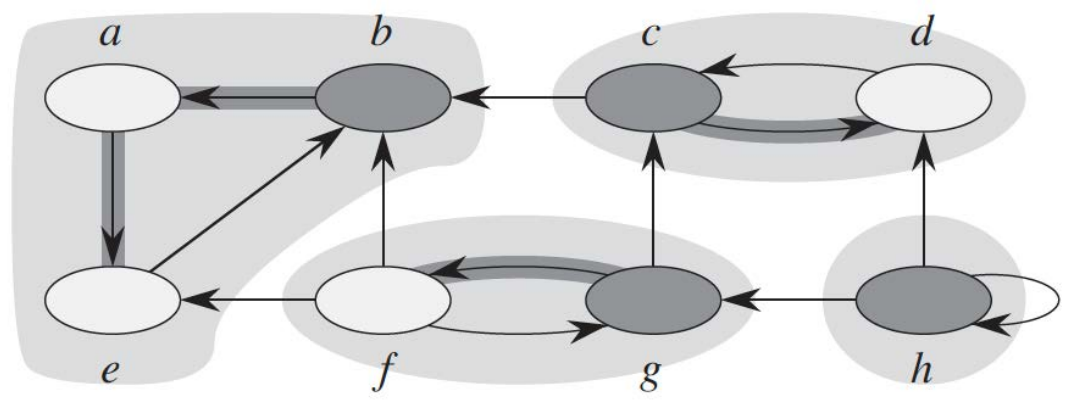
- Given a **digraph** $G = (V, E)$, a ***strongly connected component (SCC)*** of G is a maximal set of vertices $C \subseteq V$ such that for any two vertices $u, v \in C$, there is a path from u to v and also a path from v to u.
- Define $SCC(u)$ as the set of all nodes that are reachable from u and that u is also reachable from every one of them.
- **Claim 1:** If u and v are reachable from each other, then $SCC(u) = SCC(v)$.
- **Example:**



Properties of SCCs (1/3)

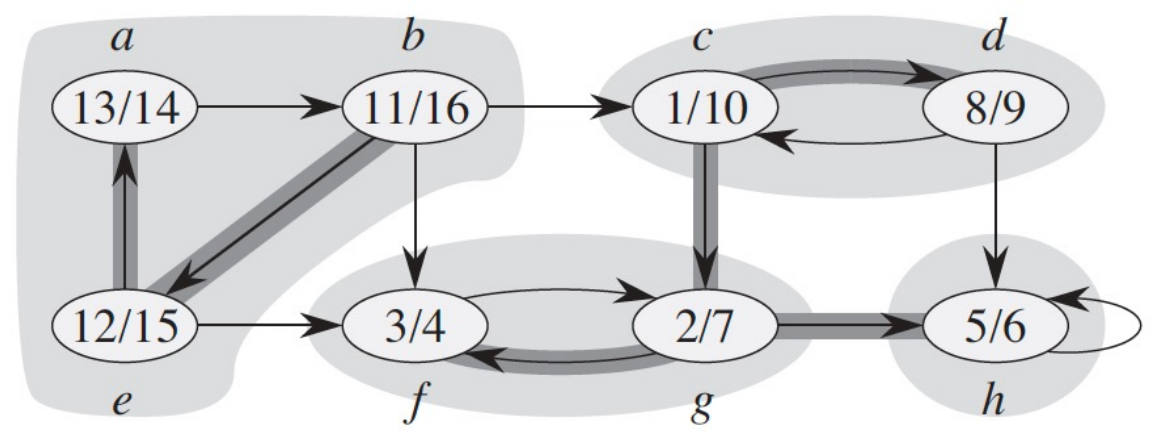
- Define $G^T = (V, E^T)$ as the ***transpose*** of G, with all edges reversed.
- **Claim 2:** G and G^T have the same **SCC**s.
- **Claim 3:** The **SCC**s of G form a **partition** of V into $\{C_1, C_2, \cdots C_k\}$. That is, $C_1 \cup C_2 \cup \cdots \cup C_k = V$ and $C_i \cap C_j = \emptyset$ for any $i \neq j$.
- **Example:**



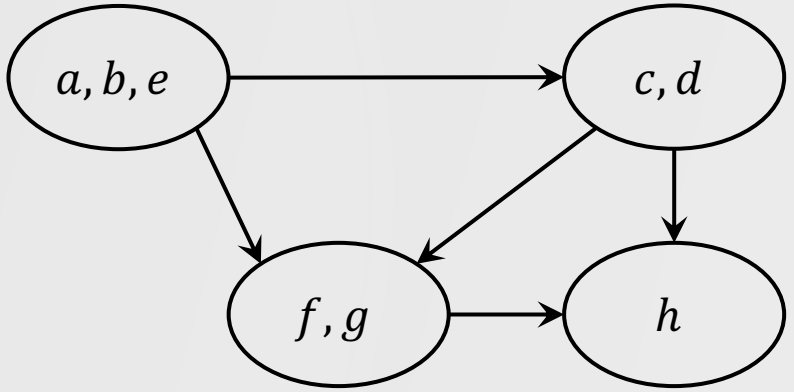


Properties of SCCs (2/3)

- Create a **component graph** $G_{SCC} = (V_{SCC}, E_{SCC})$, where
  - $V_{SCC} = \{x_1, x_2, \cdots, x_k\}$, with each vertex x_i representing a **SCC** C_i
  - E_{SCC} has an edge (x_i, x_j) iff $(u, v) \in E$ with $u \in C_i$ and $v \in C_j$.
- **Example:**



G



G_{SCC}

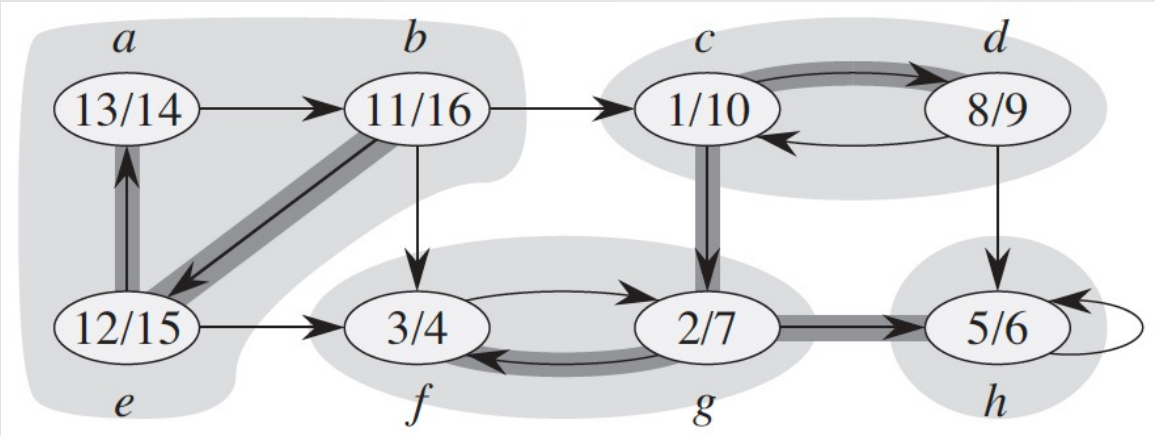
Properties of SCCs (3/3)

Claim 4: G_{SCC} is a DAG.

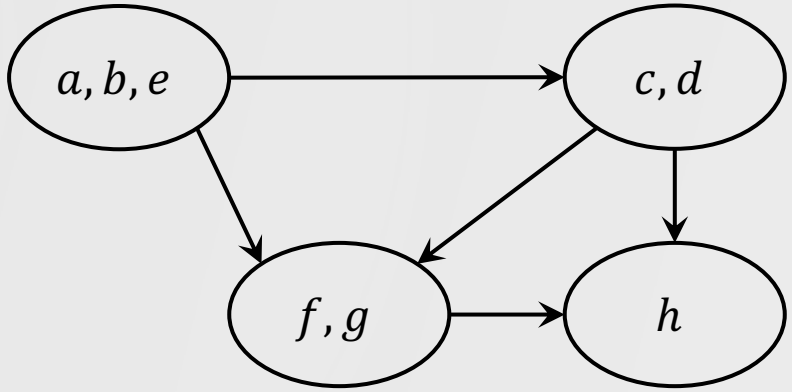
Proof. By contradiction.

- If G_{SCC} is not a DAG, then there is a cycle $\langle x_i, x_j, \cdots, x_i \rangle$ in G_{SCC}.
- Consider vertices $v_i \in C_i$ and $v_j \in C_j$. There is a path from v_i to v_j and also a path from v_j to v_i. v_i and v_j should be in the same SCC. (**Claim 1**)

A contradiction: C_i and C_j are different SCCs.



G



G_{SCC}

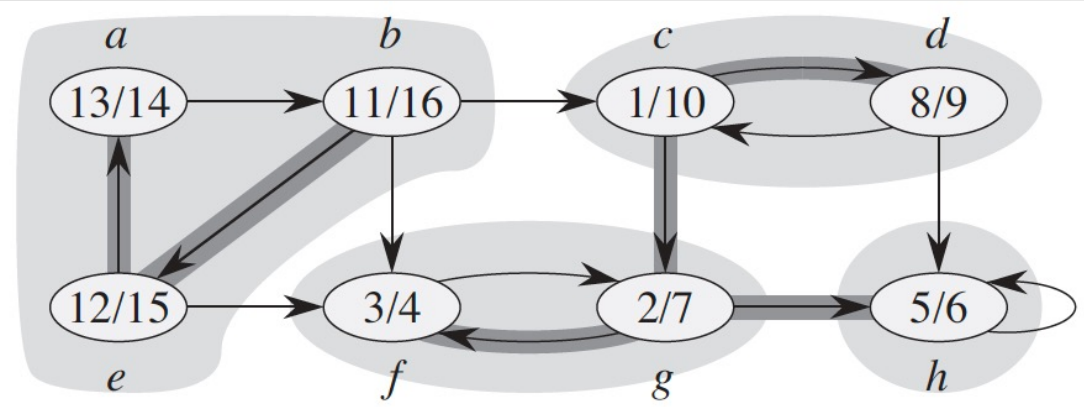
Finding SCCs

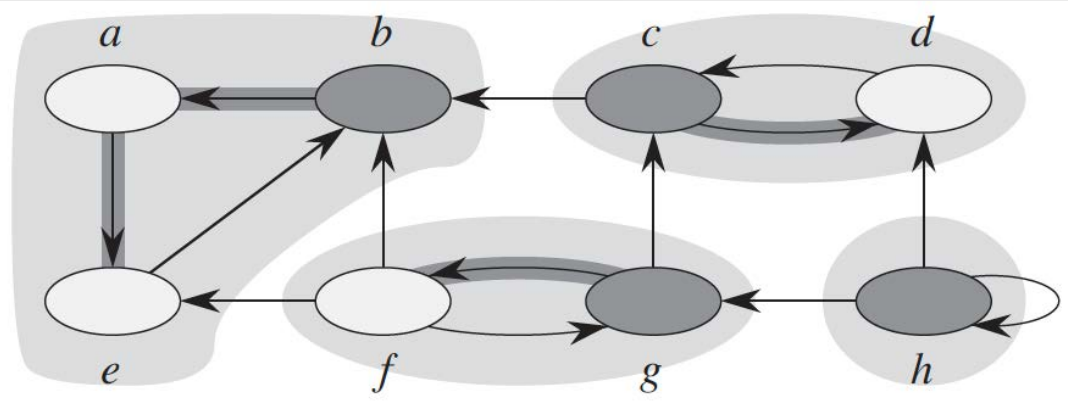
SCC(G)

1. Call DFS(G) to get a decreasing $v.f$ order
2. Create G^T
3. Call DFS(G^T) with the main DFS loop traversing nodes in a decreasing $v.f$ order (computed in step 1)
4. **return** the vertex set of each tree of the depth-first forest of G^T formed in step 3

Running time: $\Theta(n+m)$

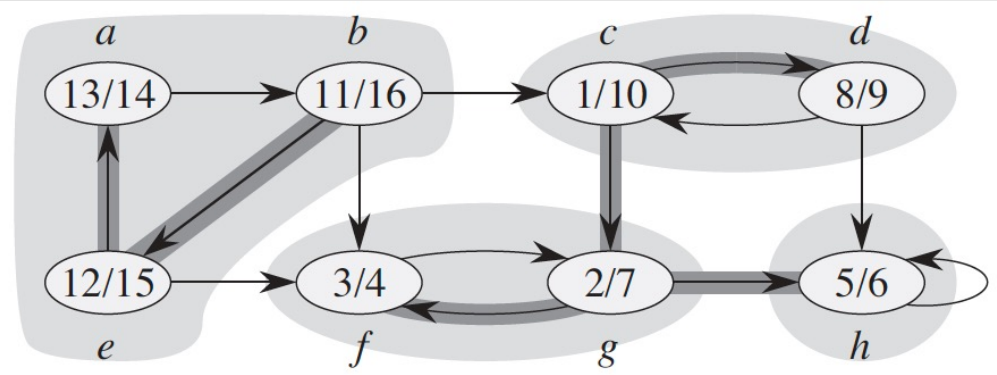
Example:

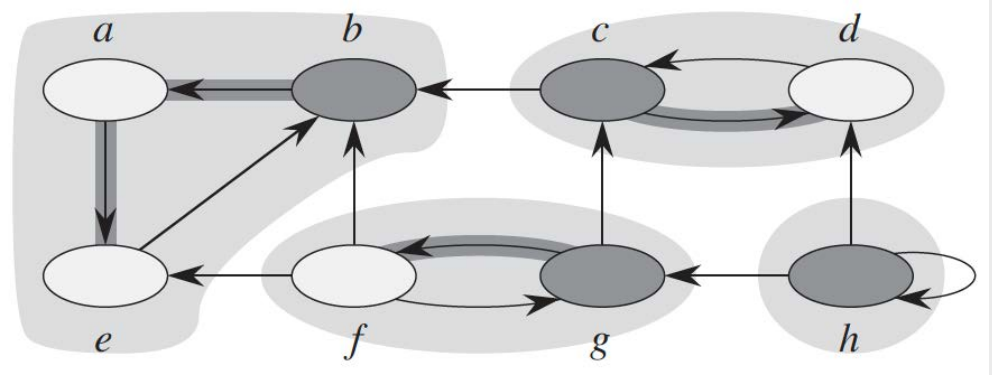


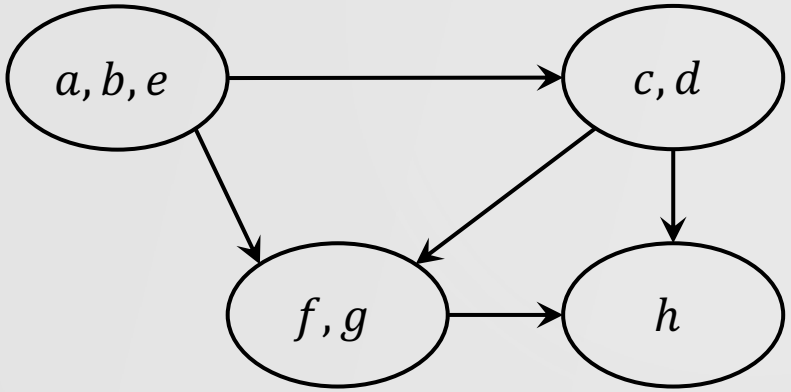


Intuition for the Correctness *(Optional)* (1/5)

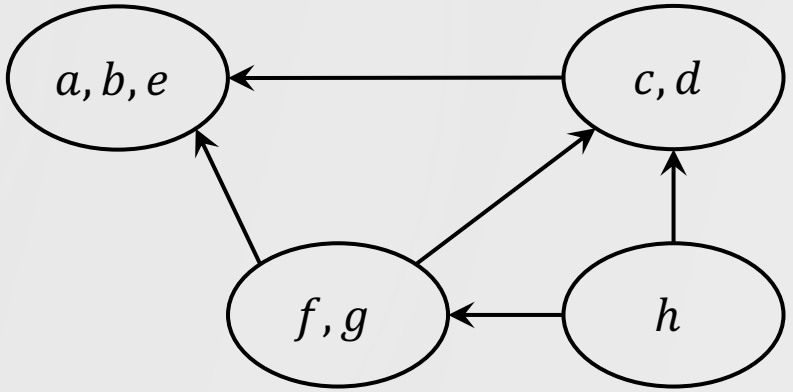
- First, observe that $(G^T)_{SCC} = (G_{SCC})^T$.
- **Example:**







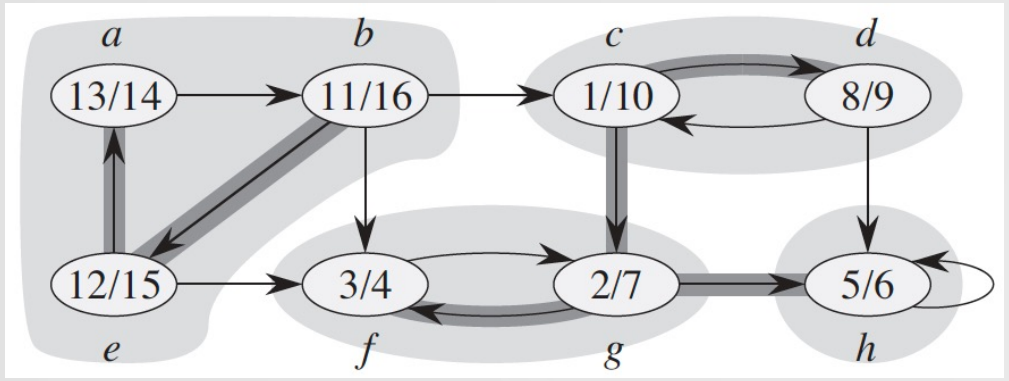
G_{SCC}



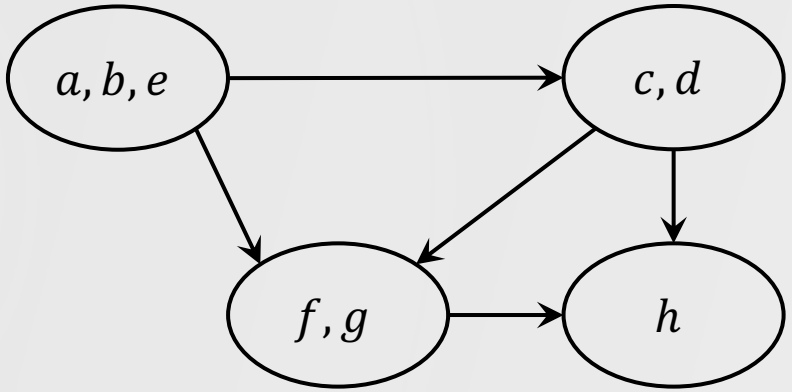
$(G^T)_{SCC}$

Intuition for the Correctness *(Optional)* (2/5)

- For any SCC C_i of G, let w_i be the node with the largest finishing time.
- Let $w_1, w_2, \cdots, w_k$ be in descending order of finishing time.
- As G_{SCC} is a DAG (**Claim 4**), $x_1, x_2, \cdots, x_k$ is a topological sort of G_{SCC}.
- **Example:**



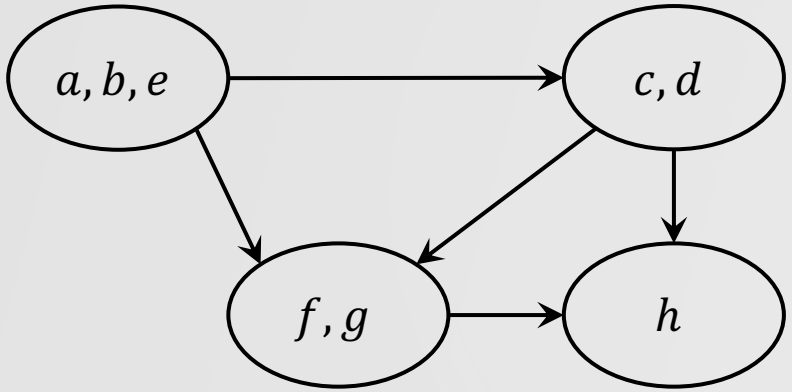
G



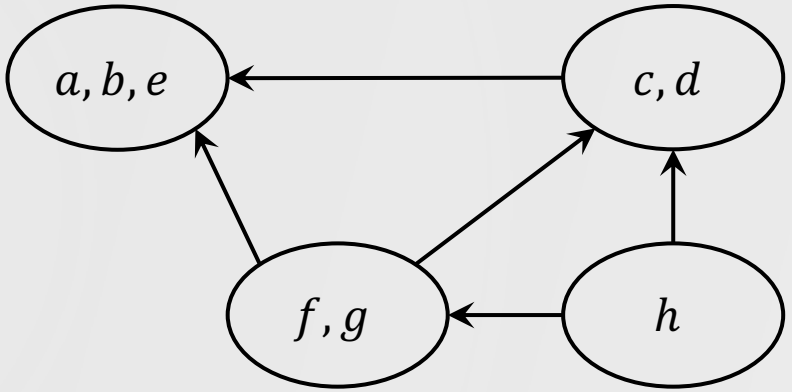
G_{SCC}

Intuition for the Correctness *(Optional)* (3/5)

- As G_{SCC} is a DAG (**Claim 4**), $x_1, x_2, \cdots, x_k$ is a topological sort of G_{SCC}.
- So, $x_1, x_2, \cdots, x_k$ is the inverse of a topological sort of $(G^T)_{SCC}$.
- This means that not a single edge leaves x_1 in $(G^T)_{SCC}$.
- **Example:**



G_{SCC}



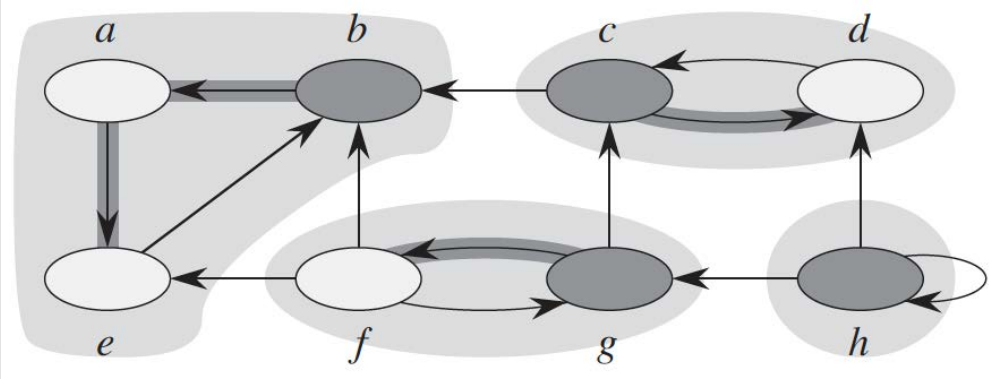
$(G^T)_{SCC}$

Intuition for the Correctness *(Optional)* (4/5)

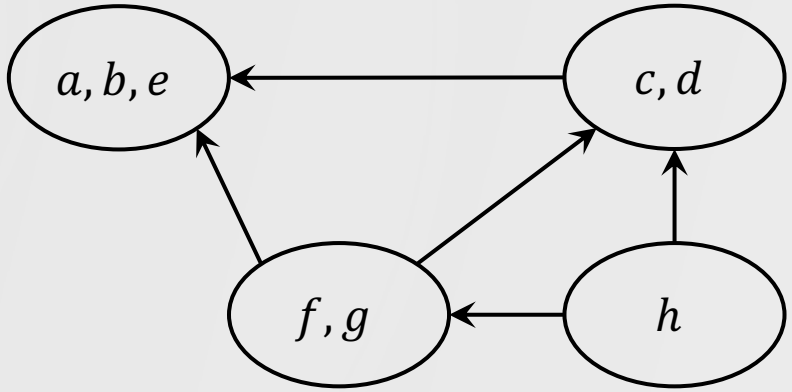
SCC(G)

1. Call DFS(G) to get a decreasing $v.f$ order
2. Create G^T
3. Call DFS(G^T) with the main DFS loop traversing nodes in a decreasing $v.f$ order (computed in step 1)
4. **return** the vertex set of each tree of the depth-first forest of G^T formed in step 3

- Not a single edge leaves x_1 in $(G^T)_{SCC}$.
- w_1 (with largest finishing time) is the first vertex in the DFS(G^T) call.
- Therefore, all vertices reachable from w_1 are the vertices in $SCC(w_1) = C_1$.



G^T



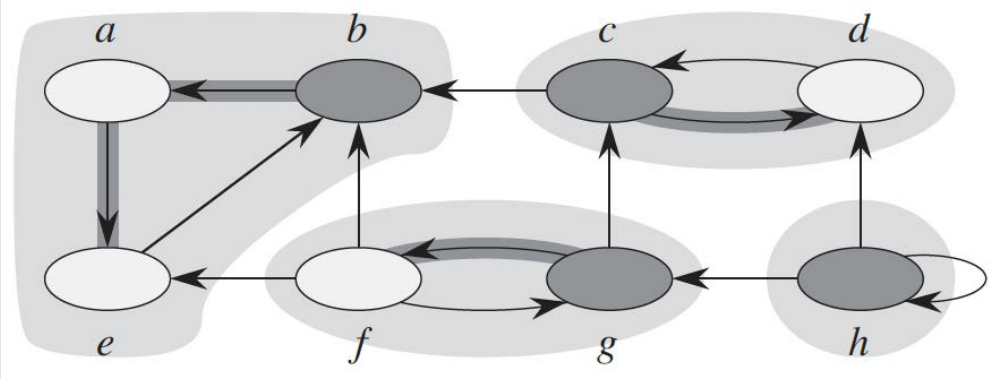
$(G^T)_{SCC}$

Intuition for the Correctness *(Optional)* (5/5)

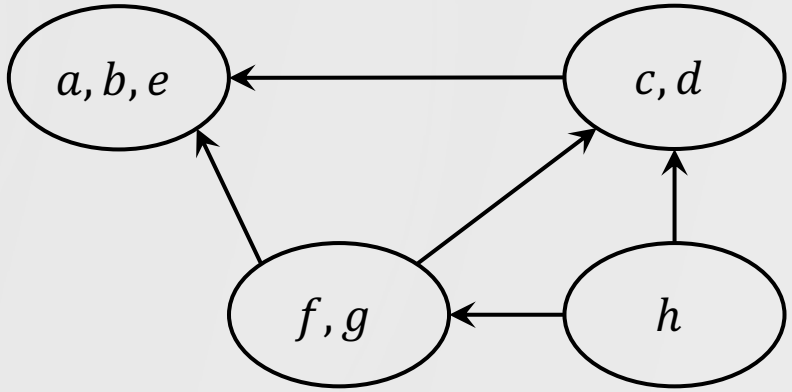
SCC(G)

1. Call DFS(G) to get a decreasing $v.f$ order
2. Create G^T
3. Call DFS(G^T) with the main DFS loop traversing nodes in a decreasing $v.f$ order (computed in step 1)
4. **return** the vertex set of each tree of the depth-first forest of G^T formed in step 3

- Continue inductively to argue that:
- By the time w_i is discovered in the DFS(G^T) call, all the vertices in SCCs $C_1, C_2, \cdots, C_{i-1}$ are already BLACK, so w_i will only reach vertices in C_i.



G^T



$(G^T)_{SCC}$

Thank you!
Questions?