CS 577 - Graphs

Manolis Vlatakis

Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)
- Trees

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)
- Trees
- Digraph

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)
- Trees
- Digraph
- Directed Acyclic Graph (DAG)

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)
- Trees
- Digraph
- Directed Acyclic Graph (DAG)
- Bipartite

Graphs

Graphs

A graph G is a pair $G = (V, E)$, where V is a set of vertices/nodes and E is a set of edges/arcs connecting a pair of vertices. That is, $E \in V \times V$.

Some Special Graphs

- Complete graph (K_4)
- Cycle (C_4)
- Path (P_4)
- Trees
- Digraph
- Directed Acyclic Graph (DAG)
- Bipartite
- Forests

Trees

Definition

- A connected graph without cycles.
- A single node may be designated as the root of the tree.
- Any node with degree 1 that is not the root is a leaf.

Trees

Definition

- A connected graph without cycles.
- A single node may be designated as the root of the tree.
- Any node with degree 1 that is not the root is a leaf.

Properties of a tree T

1. If $|V| \geq 2$, (unrooted) T has at least 2 leaves.
2. For all nodes u and v, there exists one path between them in T.
3. $|V| = |E| + 1$ for $|V| \geq 1$.

Trees

Definition

- A connected graph without cycles.
- A single node may be designated as the root of the tree.
- Any node with degree 1 that is not the root is a leaf.

Properties of a tree T

1. If $|V| \geq 2$, (unrooted) T has at least 2 leaves.
2. For all nodes u and v, there exists one path between them in T.
3. $|V| = |E| + 1$ for $|V| \geq 1$.

🤔Is P_{10} a tree?

What can be represented by graphs?

- Transportation networks
- Communication networks
- Information networks
- Social networks
- Dependency networks

Connectivity

Graph Connectivity

Problem: s-t connectivity

Given a graph $G = (V, E)$, and the vertices s and t, is there a path from s to t in G?

Graph Connectivity

Problem: s-t connectivity

Given a graph $G = (V, E)$, and the vertices s and t, is there a path from s to t in G?

Connected Graph

If all $(u, v) \in V \times V$ are connected, then G is connected.

Graph Connectivity

Problem: s-t connectivity

Given a graph $G = (V, E)$, and the vertices s and t, is there a path from s to t in G?

Connected Graph

If all $(u, v) \in V \times V$ are connected, then G is connected.

Connected Components

Let $H \subset G$ be a subgraph of G. If H is connected and there are no edges between H and $G \setminus H$. Then, H is a connected component of G.

Graph Exploration/Traversal

Determining *s*-*t* Connectivity

Requires an algorithm that explores or traverses the graph by considering the edges of the graph to find all nodes connected to s.

Graph Exploration/Traversal

Determining s-t Connectivity

Requires an algorithm that explores or traverses the graph by considering the edges of the graph to find all nodes connected to s.

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
  Add v to R
end
return R

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix			
Adjacency list			
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$		
Adjacency list			
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	
Adjacency list			
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
Adjacency list			
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$		
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	$O(1)$
Edge list			
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	$O(1)$
Edge list	$O(\|E\| + \|V\|)$		
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
Adjacency list	$O(\lvert V\rvert \cdot \min(\lvert E\rvert, \lvert V\rvert))$	$O(\min(\lvert V\rvert, \lvert E\rvert))$	$O(1)$
Edge list	$O(\lvert E\rvert + \lvert V\rvert)$	$O(\lvert E\rvert)$	
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	$O(1)$
Edge list	$O(\|E\| + \|V\|)$	$O(\|E\|)$	$O(\|E\|)$
Incidence matrix			

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$
Adjacency list	$O(\|V\| \cdot \min(\|E\|, \|V\|))$	$O(\min(\|V\|, \|E\|))$	$O(1)$
Edge list	$O(\|E\| + \|V\|)$	$O(\|E\|)$	$O(\|E\|)$
Incidence matrix	$O(\|V\|\|E\|)$		

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
Adjacency list	$O(\lvert V\rvert \cdot \min(\lvert E\rvert, \lvert V\rvert))$	$O(\min(\lvert V\rvert, \lvert E\rvert))$	$O(1)$
Edge list	$O(\lvert E\rvert + \lvert V\rvert)$	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$
Incidence matrix	$O(\lvert V\rvert\lvert E\rvert)$	$O(\lvert E\rvert)$	

Graph Encodings and Implementation

Representations

- **Adjacency matrix**: $|V|$ by $|V|$ matrix with a 1 if nodes are adjacent.
- **Adjacency list**: For each node, list adjacent nodes.
- **Edge list**: List of all node pairs representing the edges (plus list of nodes).
- **Incidence matrix**: $|V|$ by $|E|$ matrix with a 1 if node is incident to the edge.

	Space	Find (u, v)	List of neighbours
Adjacency matrix	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
Adjacency list	$O(\lvert V\rvert \cdot \min(\lvert E\rvert, \lvert V\rvert))$	$O(\min(\lvert V\rvert, \lvert E\rvert))$	$O(1)$
Edge list	$O(\lvert E\rvert + \lvert V\rvert)$	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$
Incidence matrix	$O(\lvert V\rvert\lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert V\rvert\lvert E\rvert)$

Graph Exploration/Traversal

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
  Add v to R
end
return R

🤔Which graph representation would be best suited?

Graph Exploration/Traversal

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
    Add v to R
end
return R

Rough Running Time

- At step i: $O(|E_i| \cdot (\log |R_i| + \log |R_i|) + \log |R_i|)$, assuming R is a self-balancing BST.

Graph Exploration/Traversal

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
    Add v to R
end
return R

Rough Running Time

- At step i: $O(|E_i| \cdot (\log |R_i| + \log |R_i|) + \log |R_i|)$, assuming R is a self-balancing BST.
- At most $|E|$ steps: $O(|E|^2 \log |V|)$

Graph Exploration/Traversal

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
    Add v to R
end
return R

Rough Running Time

- At step i: $O(|E_i| \cdot (\log |R_i| + \log |R_i|) + \log |R_i|)$, assuming R is a self-balancing BST.
- At most $|E|$ steps: $O(|E|^2 \log |V|)$

What is this algorithm lacking?

Graph Exploration/Traversal

Algorithm: Generalized Exploration

$R = \{s\}$
while $\exists$ an edge (u, v) where $u \in R$ and $v \notin R$ **do**
  Add v to R
end
return R

Rough Running Time

- At step i: $O(|E_i| \cdot (\log |R_i| + \log |R_i|) + \log |R_i|)$, assuming R is a self-balancing BST.
- At most $|E|$ steps: $O(|E|^2 \log |V|)$

An exploitable order of traversing the edges!!!

Breadth-First Search (BFS)

Process

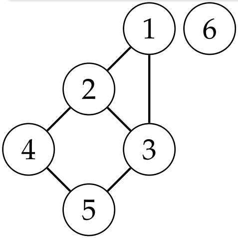
- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

Breadth-First Search (BFS)

Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

This process engenders a BFS tree. Start at 1 and draw such a tree for the following.

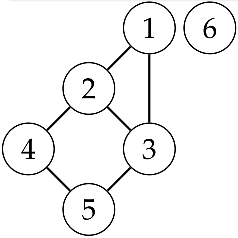


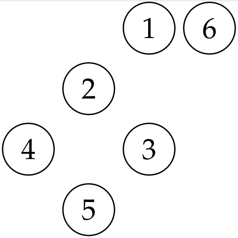
Breadth-First Search (BFS)

Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

This process engenders a BFS tree. Start at 1 and draw such a tree for the following.



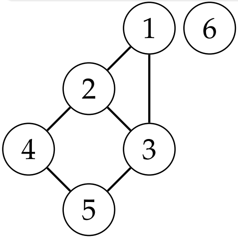


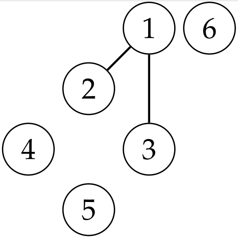
Breadth-First Search (BFS)

Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

This process engenders a BFS tree. Start at 1 and draw such a tree for the following.



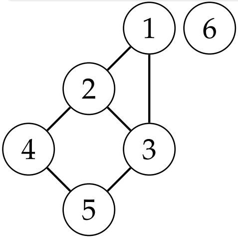


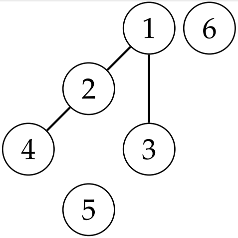
Breadth-First Search (BFS)

Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

This process engenders a BFS tree. Start at 1 and draw such a tree for the following.



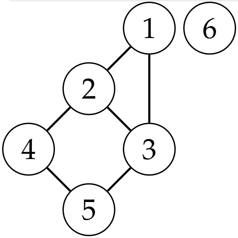


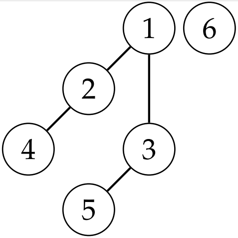
Breadth-First Search (BFS)

Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

This process engenders a BFS tree. Start at 1 and draw such a tree for the following.





Depth-First Search (DFS)

Recursive Process starting at s

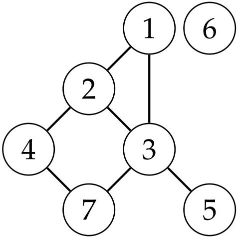
- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.

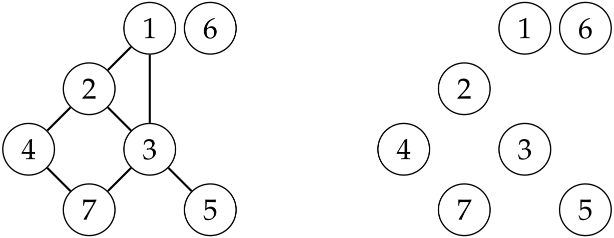


Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.

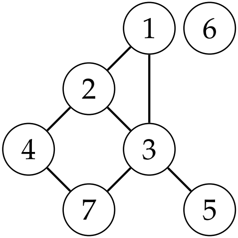


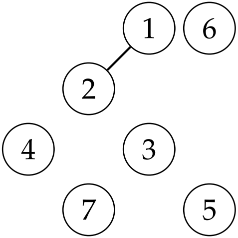
Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.



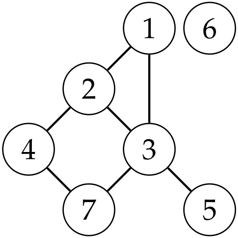


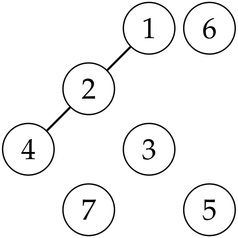
Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.



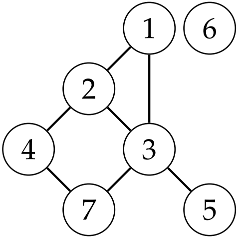


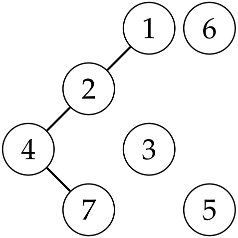
Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.



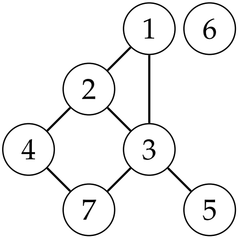


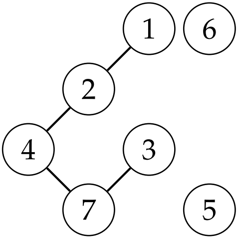
Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.



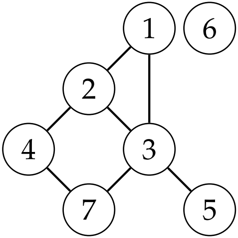


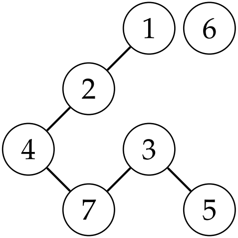
Depth-First Search (DFS)

Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

This process engenders a DFS tree. Start at 1 and draw such a tree for the following.





Implementing BFS and DFS

🤔Which graph representation would be best for BFS and DFS?

Implementing BFS and DFS

🤔Which graph representation would be best for BFS and DFS? Why?

Implementing BFS and DFS

BFS Process

- Also referred to as graph flooding.
- Let L_i be all the neighbours at a distance i from s.
- Starting from $i = 0$, visit all the nodes (not previously visited) in L_i. Increment i and repeat.

DFS Recursive Process starting at s

- Mark s as visited.
- For each $(s, u) \in E$ where u has not been visited, do DFS(u).

Implementing BFS and DFS

Algorithm: BFS(S)

```
Initialize v[u] = false for all
  nodes
Set v[s] = true
Add s to tree T
Add s to queue Q
while Q is not empty do
    u = dequeue(Q)
    foreach neighbour r of u
      do
        if !v[r] then
            Add (u, r) to T
            Set v[r] = true
            Enqueue v
        end
    end
end
return T
```

Implementing BFS and DFS

Algorithm: BFS(S)

```
Initialize v[u] = false for all nodes
Set v[s] = true
Add s to tree T
Add s to queue Q
while Q is not empty do
    u = dequeue(Q)
    foreach neighbour r of u do
        if !v[r] then
            Add (u, r) to T
            Set v[r] = true
            Enqueue v
        end
    end
end
return T
```

Algorithm: DFS(S)

```
Initialize v[u] = false and p[u] = null for all nodes
Push s to stack S
while S is not empty do
    u = pop(S)
    if !v[u] then
        Add (p[u], u) to T
        Set v[u] = true
        foreach neighbour r of u do
            Push r to stack S
            Set p[r] = u
        end
    end
end
return T
```

Implementing BFS and DFS

Algorithm: BFS(S)

```
Initialize v[u] = false for all nodes
Set v[s] = true
Add s to tree T
Add s to queue Q
while Q is not empty do
    u = dequeue(Q)
    foreach neighbour r of u do
        if !v[r] then
            Add (u, r) to T
            Set v[r] = true
            Enqueue v
        end
    end
end
return T
```

Algorithm: DFS(S)

```
Initialize v[u] = false and p[u] = null for all nodes
Push s to stack S
while S is not empty do
    u = pop(S)
    if !v[u] then
        Add (p[u], u) to T
        Set v[u] = true
        foreach neighbour r of u do
            Push r to stack S
            Set p[r] = u
        end
    end
end
return T
```

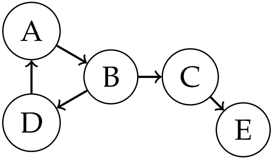
Runtime: $O(|E| + |V|)$

Strongly Connected Components

Directed Graphs

Directed Graph

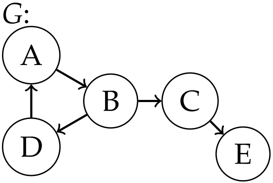
- In a directed graph, the edges have a direction and are often called arcs.
- I.e. (u, v) is different than (v, u).



Strong Connectivity

Mutually Reachable

- A pair of nodes (u, v) in a directed graph are mutually reachable if there is a path from u to v, and from v to u.
- Note: This property is transitive: if (u, v) and (v, w) are both mutually reachable, then u, w is mutually reachable.



Strong Connectivity

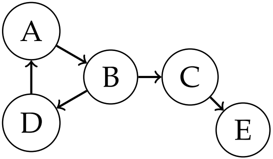
Mutually Reachable

- A pair of nodes (u, v) in a directed graph are mutually reachable if there is a path from u to v, and from v to u.
- Note: This property is transitive: if (u, v) and (v, w) are both mutually reachable, then u, w is mutually reachable.

Strongly Connected

A directed graph is strongly connected if, for every pair of nodes (u, v), u and v are mutually reachable.

G:



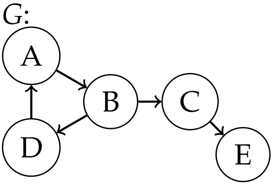
Strong Connectivity

Mutually Reachable

- A pair of nodes (u, v) in a directed graph are mutually reachable if there is a path from u to v, and from v to u.
- Note: This property is transitive: if (u, v) and (v, w) are both mutually reachable, then u, w is mutually reachable.

Testing for Mutually Reachable

How might we check if (u, v) is mutually reachable?



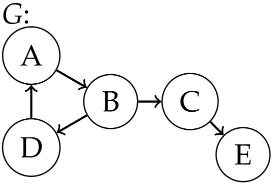
Strong Connectivity

Mutually Reachable

- A pair of nodes (u, v) in a directed graph are mutually reachable if there is a path from u to v, and from v to u.
- Note: This property is transitive: if (u, v) and (v, w) are both mutually reachable, then u, w is mutually reachable.

Testing for Mutually Reachable

Check if DFS/BFS from u reach v, and DFS/BFS from v reaches u.



Strong Connectivity

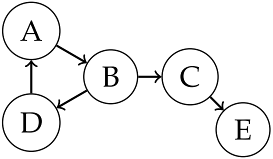
Mutually Reachable

- A pair of nodes (u, v) in a directed graph are mutually reachable if there is a path from u to v, and from v to u.
- Note: This property is transitive: if (u, v) and (v, w) are both mutually reachable, then u, w is mutually reachable.

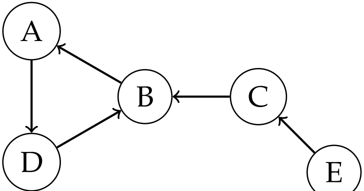
Testing for Mutually Reachable

Check if DFS/BFS from u in G reaches v, and DFS/BFS from u in G^{REV} reaches v.

G:



G^{REV}:



Strongly Connected Components

Strongly Connected Component (SCC)

A maximal strongly connected subgraph.

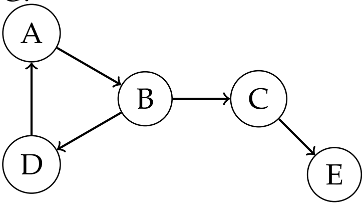
Strongly Connected Components

Strongly Connected Component (SCC)

A maximal strongly connected subgraph.

🤔How many SCC in G?

G:



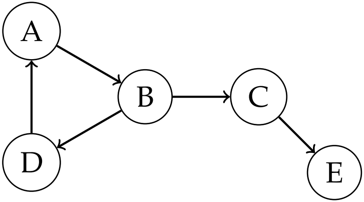
Strongly Connected Components

Strongly Connected Component (SCC)

A maximal strongly connected subgraph.

🤔How many SCC in G? 3

G:



Strongly Connected Components

Problem

Find the SCCs in a digraph G.

Strongly Connected Components

Problem

Find the SCCs in a digraph G.

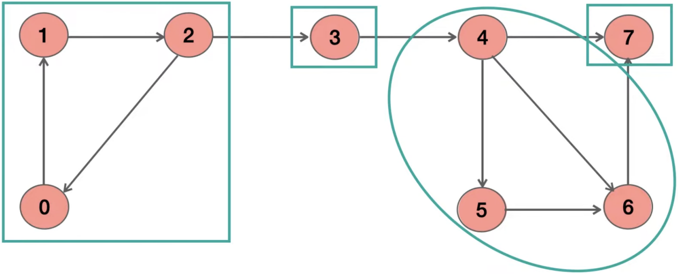
Kosaraju's Algorithm

1. Populate a stack S with a DFS on G.
2. Build G^{REV} for G, and set all nodes to unvisited.
3. While S is not empty:
   1. Pop node v from S.
   2. If v is unvisited, run DFS on G^{REV} from v to extract an SCC.

Kosaraju's Algorithm

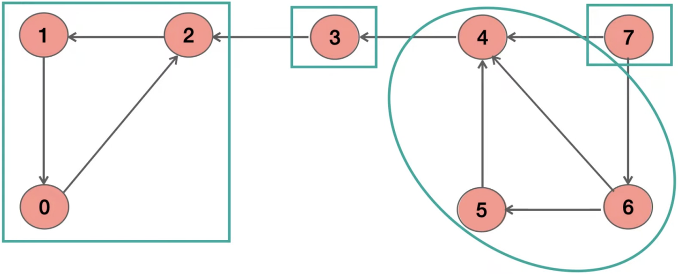
Kosaraju's Algorithm

Execution Paradigm



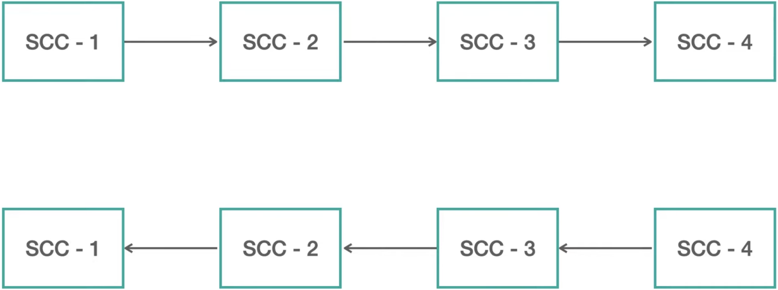
Kosaraju's Algorithm

Execution Paradigm



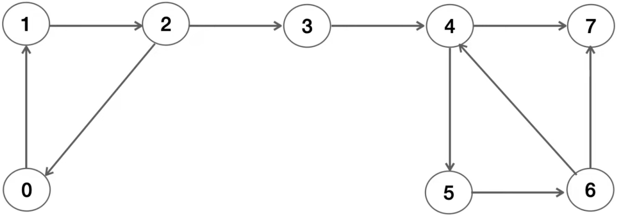
Kosaraju's Algorithm

Execution Paradigm



Kosaraju's Algorithm

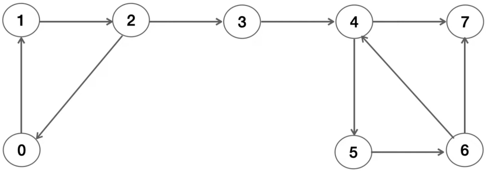
Execution Paradigm



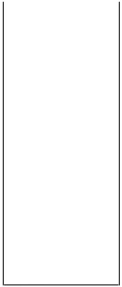
Start DFS traversal
Put vertex on stack when finished

Kosaraju's Algorithm

Execution Paradigm

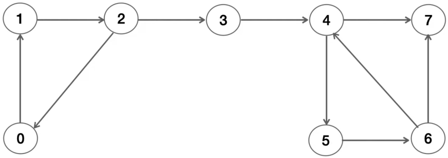


Start DFS traversal
Put vertex on stack when finished

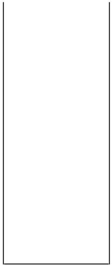


Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

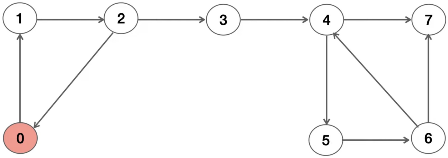


Stack

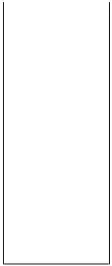
visited	F	F	F	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

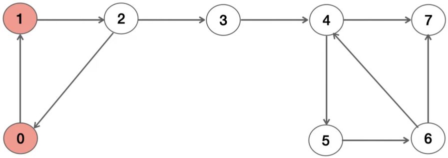


Stack

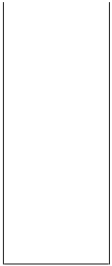
visited	T	F	F	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

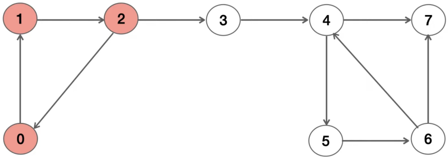


Stack

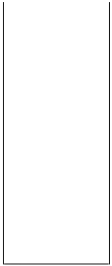
visited	T	T	F	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

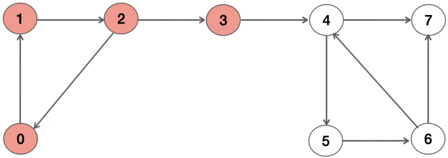


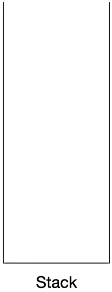
Stack

visited	T	T	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



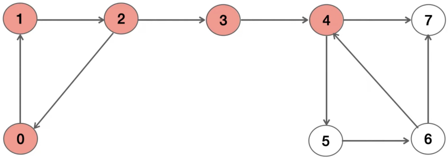


Start DFS traversal
Put vertex on stack when finished

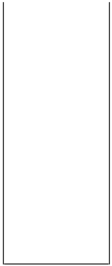
visited	T	T	T	T	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

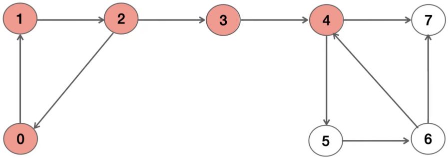


Stack

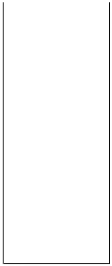
visited	T	T	T	T	T	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

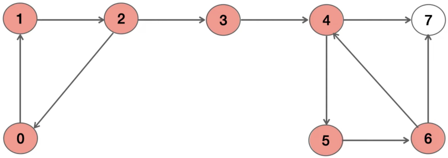


Stack

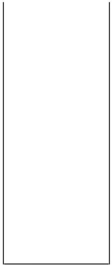
visited	T	T	T	T	T	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

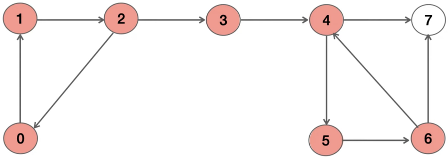


Stack

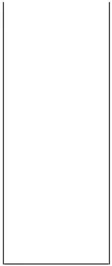
visited	T	T	T	T	T	T	T	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Start DFS traversal
Put vertex on stack when finished

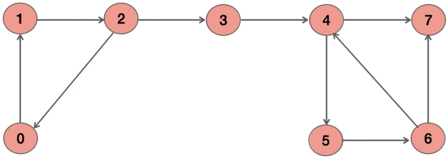


Stack

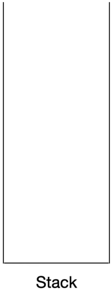
	visited							
visited	T	T	T	T	T	T	T	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



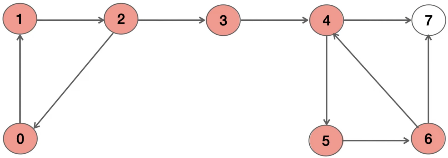
Start DFS traversal
Put vertex on stack when finished



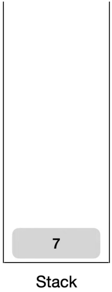
visited	T	T	T	T	T	T	T	T
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



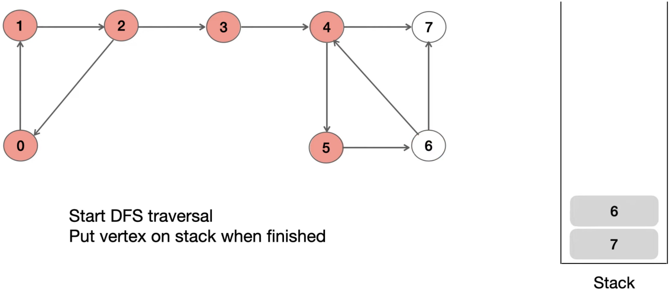
Start DFS traversal
Put vertex on stack when finished



visited	T	T	T	T	T	T	T	T
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm

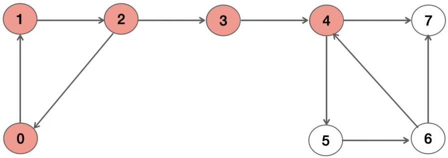


Start DFS traversal
Put vertex on stack when finished

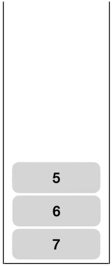
visited	T	T	T	T	T	T	T	T
	0	1	2	3	4	5	6	7

KOSARAJU'S ALGORITHM

EXECUTION PARADIGM



Start DFS traversal
Put vertex on stack when finished

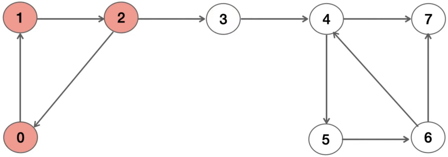


Stack

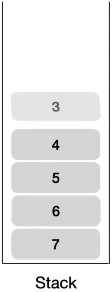
	0	1	2	3	4	5	6	7
visited	T	T	T	T	T	T	T	T

Kosaraju's Algorithm

Execution Paradigm



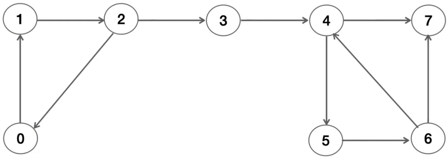
Start DFS traversal
Put vertex on stack when finished



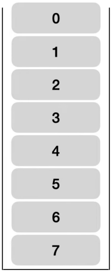
visited	T	T	T	T	T	T	T	T
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



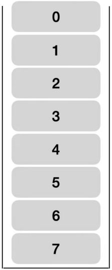
Start DFS traversal
Put vertex on stack when finished



visited	T	T	T	T	T	T	T	T
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm

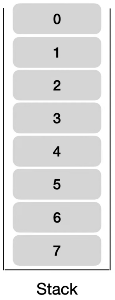


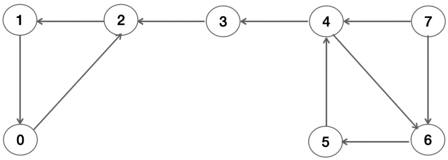
Stack

Reverse the original graph

Kosaraju's Algorithm

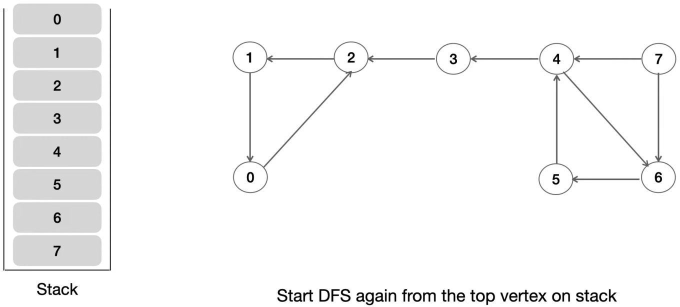
Execution Paradigm





Kosaraju's Algorithm

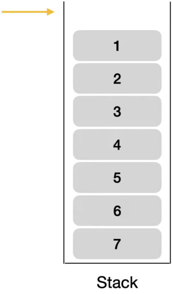
Execution Paradigm

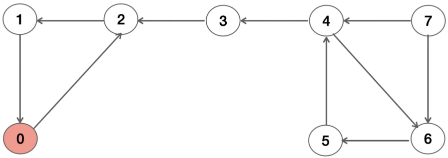


Start DFS again from the top vertex on stack

Kosaraju's Algorithm

Execution Paradigm



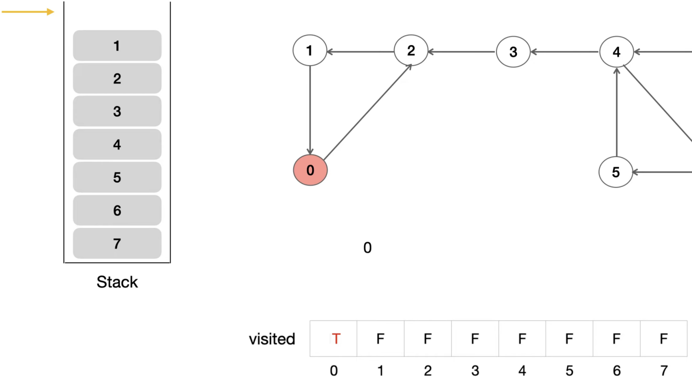


Start DFS again from the top vertex on stack

visited	0	1	2	3	4	5	6	7
	F	F	F	F	F	F	F	F

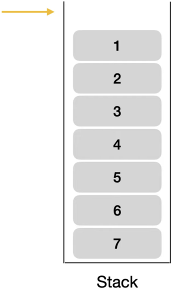
Kosaraju's Algorithm

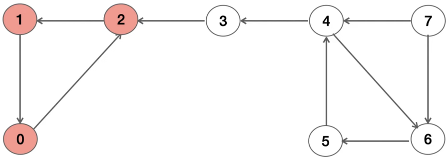
Execution Paradigm



Kosaraju's Algorithm

Execution Paradigm



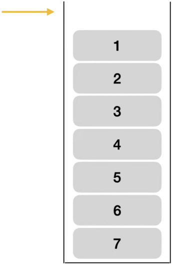


0 2

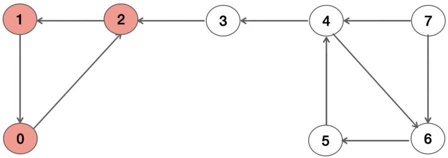
visited	T	F	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Stack

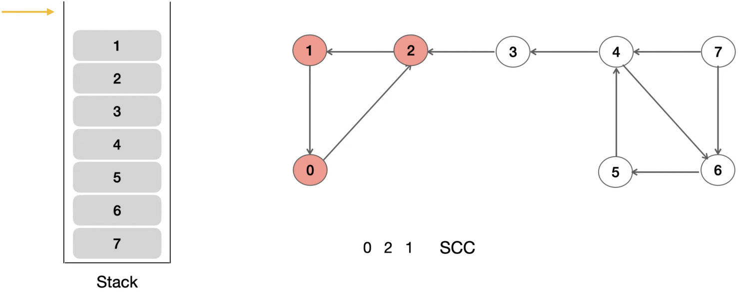


0 2 1

visited	T	T	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

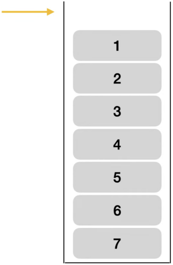
Execution Paradigm



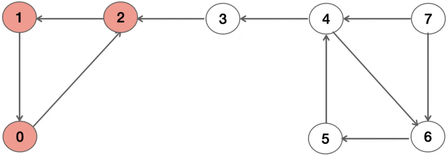
visited	T	T	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Stack



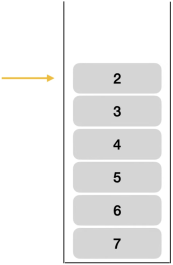
0 2 1 SCC

Keep popping the nodes until we get unvisited node

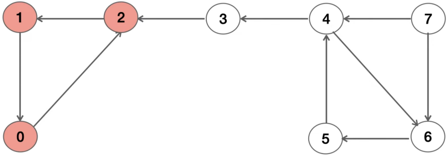
visited	T	T	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

Execution Paradigm



Stack



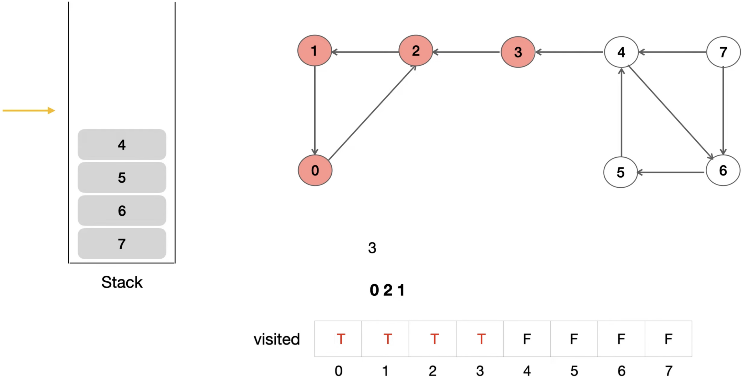
0 2 1 SCC

Keep popping the nodes until we get unvisited node

visited	T	T	T	F	F	F	F	F
	0	1	2	3	4	5	6	7

Kosaraju's Algorithm

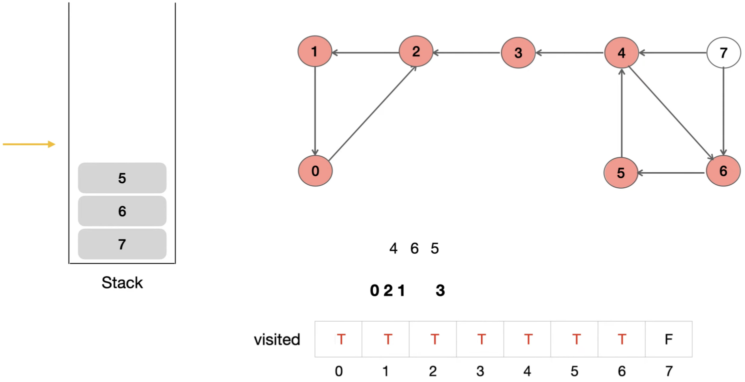
Execution Paradigm



🤔How I can remember that [0,2,1] are in a different SCC?

Kosaraju's Algorithm

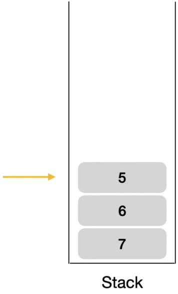
Execution Paradigm

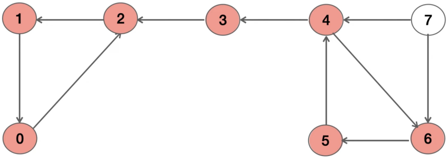


We can have an array SCC[], initialized to −1 and write the number of the corresponding SCC when it is finalized.

Kosaraju's Algorithm

Execution Paradigm





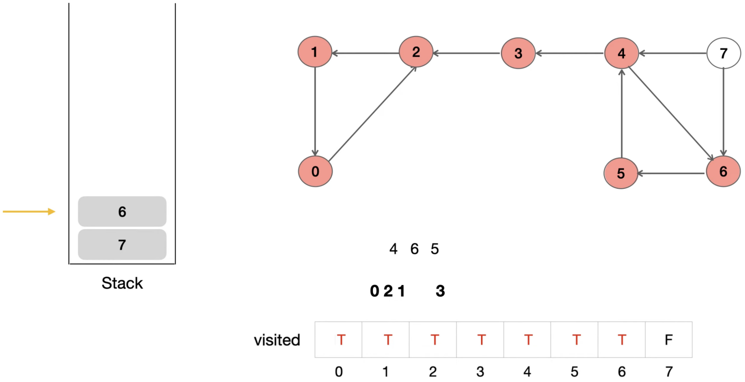
4 6 5

0 2 1 3

visited	T	T	T	T	T	T	T	F
	0	1	2	3	4	5	6	7

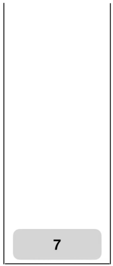
Kosaraju's Algorithm

Execution Paradigm

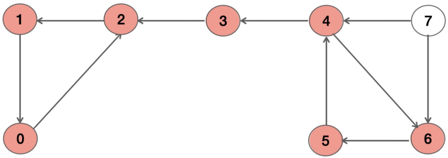


Kosaraju's Algorithm

Execution Paradigm



Stack

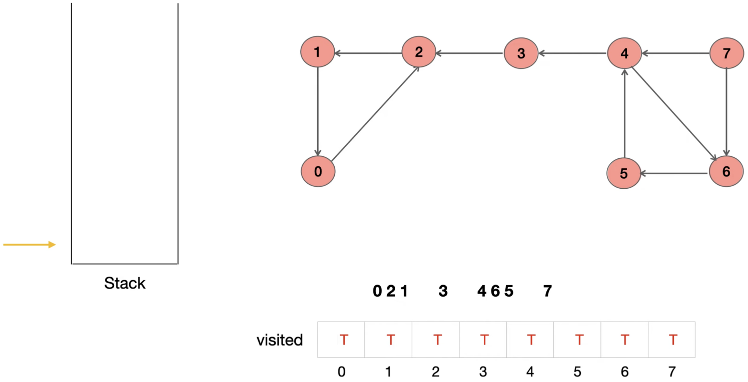


0 2 1 3 4 6 5

	0	1	2	3	4	5	6	7
visited	T	T	T	T	T	T	T	F

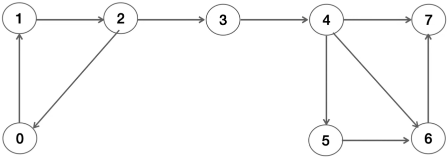
Kosaraju's Algorithm

Execution Paradigm

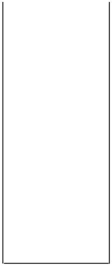


Kosaraju's Algorithm

Recap

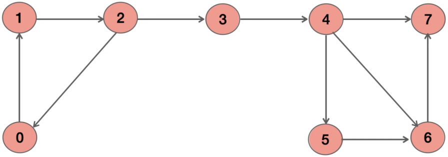


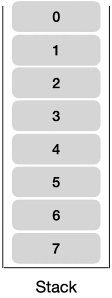
Step 1: Start DFS traversal
Put vertex on stack when finished



Kosaraju's Algorithm

Recap

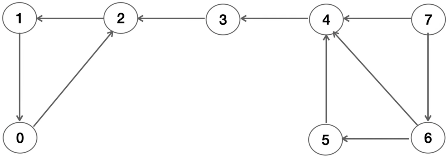




Step 1: Start DFS traversal
Put vertex on stack when finished

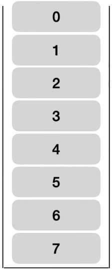
KOSARAJU'S ALGORITHM

RECAP



Step 2: Reverse the original graph

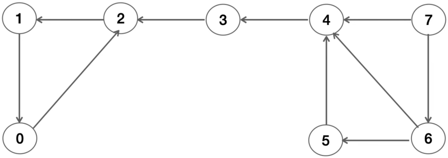
Start DFS again from top of the vertex



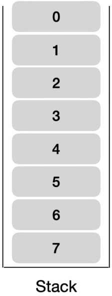
Stack

Kosaraju's Algorithm

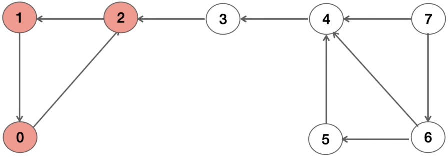
Recap



Step 3: Start DFS again from top of the vertex

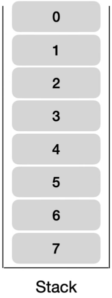


Kosaraju's Algorithm
Recap



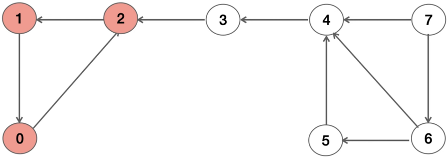
Step 3: Start DFS again from top of the vertex

When DFS finishes, all visited nodes form a SCC



Kosaraju's Algorithm

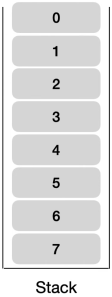
Recap



Step 3: Start DFS again from top of the vertex

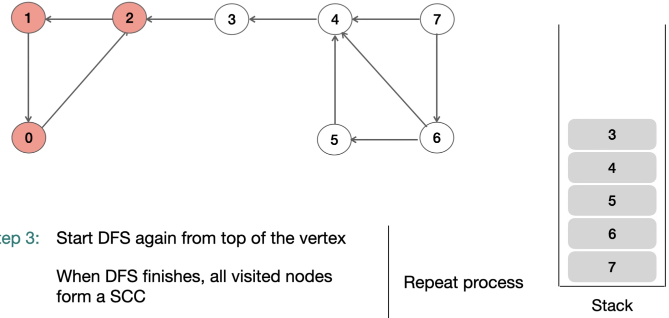
When DFS finishes, all visited nodes form a SCC

Pop nodes until unvisited node is found



Kosaraju's Algorithm

Recap



Step 3: Start DFS again from top of the vertex

When DFS finishes, all visited nodes form a SCC

Pop nodes until unvisited node is found

Repeat process

Strongly Connected Components

Problem

Find the SCCs in a digraph G.

Kosaraju's Algorithm

1. Populate a stack S with a DFS on G.
2. Build G^{REV} for G, and set all nodes to unvisited.
3. While S is not empty:
   1. Pop node v from S.
   2. If v is unvisited, run DFS on G^{REV} from v to extract an SCC.

🤔What is the time complexity of Kosaraju's Algorithm?

Strongly Connected Components

Problem

Find the SCCs in a digraph G.

Kosaraju's Algorithm

1. Populate a stack S with a DFS on G.
2. Build G^{REV} for G, and set all nodes to unvisited.
3. While S is not empty:
    1. Pop node v from S.
    2. If v is unvisited, run DFS on G^{REV} from v to extract an SCC.

🤔What is the time complexity of Kosaraju's Algorithm?
$O(|E| + |V|)$

Kosaraju's Algorithm

Correctness Proof: Key Lemma and Corollaries

Key Lemma

Let C be a strongly connected component of G, and v be a vertex not in C. Suppose that there is a path from C to v (i.e., there is a path from some vertex in C to v). Then

$$\max\{f[u] : u \in C\} > f[v].$$

$$\Downarrow$$

Kosaraju's Algorithm

Correctness Proof: Key Lemma and Corollaries

Key Lemma

Let C be a strongly connected component of G, and v be a vertex not in C. Suppose that there is a path from C to v (i.e., there is a path from some vertex in C to v). Then

$$\max\{f[u] : u \in C\} > f[v].$$

$$\Downarrow$$

Corollary 1

Let C_1, C_2 be two strongly connected components of G, and suppose that there is a path from (some vertex in) C_1 to (some vertex in) C_2. Then

$$\max\{f[u] : u \in C_1\} > \max\{f[v] : v \in C_2\}.$$

$$\Downarrow$$

Kosaraju's Algorithm

Correctness Proof: Key Lemma and Corollaries

Corollary 1

Let C_1, C_2 be two strongly connected components of G, and suppose that there is a path from (some vertex in) C_1 to (some vertex in) C_2. Then

$$\max\{f[u] : u \in C_1\} > \max\{f[v] : v \in C_2\}.$$

$$\Downarrow$$

Corollary 2

Let C_1, C_2 be two strongly connected components of G, and suppose that

$$\max\{f[u] : u \in C_1\} > \max\{f[v] : v \in C_2\}.$$

Then there is no path in G from C_2 to C_1.

Kosaraju's Algorithm

Correctness Proof: Key Lemma and Corollaries

Corollary 2

Let C_1, C_2 be two strongly connected components of G, and suppose that

$$\max\{f[u] : u \in C_1\} > \max\{f[v] : v \in C_2\}.$$

Then there is no path in G from C_2 to C_1.

$$\Downarrow$$

Corollary 3

Let C_1, C_2 be two strongly connected components of G, and suppose that

$$\max\{f[u] : u \in C_1\} > \max\{f[v] : v \in C_2\}.$$

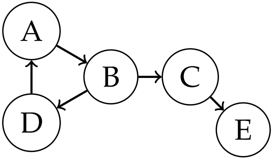
Then there is no path in G^T from C_1 to C_2. (Recall that G^T is the transpose of G, which is obtained from G by reversing all the edges of G.)

Topological Ordering

Directed Graphs

Directed Graph

- In a directed graph, the edges have a direction and are often called arcs.
- I.e. (u, v) is different than (v, u).



Directed Graphs

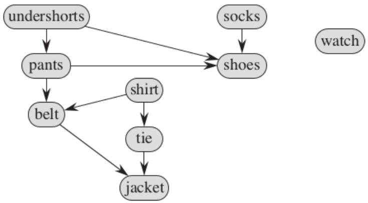
Directed Graph

- In a directed graph, the edges have a direction and are often called arcs.
- I.e. (u, v) is different than (v, u).

Directed Acyclic Graph (DAG)

- A directed graph with no directed cycles.
- Precedence relationships.

Getting dressed:



Topological Ordering

Definition

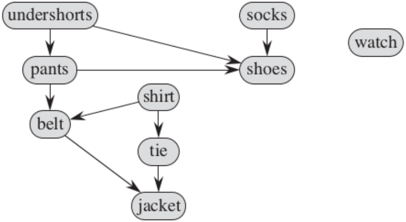
An ordering of the nodes of a DAG which respected the precedence relations.

Topological Ordering

Definition

An ordering of the nodes of a DAG which respected the precedence relations.

Getting dressed DAG:

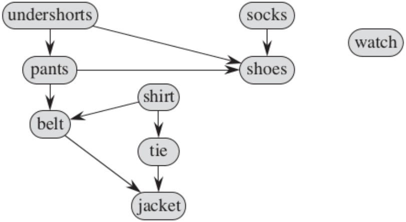


Topological Ordering

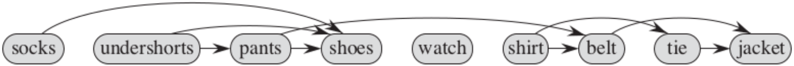
Definition

An ordering of the nodes of a DAG which respected the precedence relations.

Getting dressed DAG:



Topological ordering:



DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

Proof (Exercise)

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

Proof (Exercise)

- By way of contradiction, assume all nodes in G have an incoming edge.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

Proof (Exercise)

- By way of contradiction, assume all nodes in G have an incoming edge.
- Pick an arbitrary node u and follow the incoming node back to v. Since all nodes have an incoming edge, when can repeat this infinitely.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

Proof (Exercise)

- By way of contradiction, assume all nodes in G have an incoming edge.
- Pick an arbitrary node u and follow the incoming node back to v. Since all nodes have an incoming edge, when can repeat this infinitely.
- After visiting $|V| + 1$ nodes, by the Pigeon Hole principle, we have visited some node w twice $\implies$ G contains a cycle.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

- The Key Property allows us to show that all DAGs have a topological ordering.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

In every DAG G, there is a node v with no incoming edges.

- The Key Property allows us to show that all DAGs have a topological ordering.
- Prove it by induction.

DAGs and Topological Ordering

Observation 1

If G has a topological ordering, then G is a DAG.

Key Property

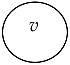
In every DAG G, there is a node v with no incoming edges.

- The Key Property allows us to show that all DAGs have a topological ordering.
- Prove it by induction.
- Does the inductive proof imply an algorithm to build a topological ordering from a DAG? If so, what is it?

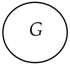
Topological Ordering

Induction

Base: DAGs with |V|=1
Topological Order: v

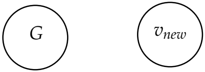


Ind. Hypothesis: DAGs with |V|=k
Topological Order: $v_1 \prec \cdots \prec v_k$



Ind. Step: DAGs with |V|=k+1

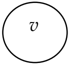
Topological Order:



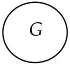
Topological Ordering

Induction

Base: DAGs with |V|=1
Topological Order: v

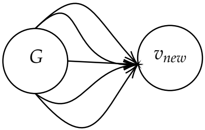


Ind. Hypothesis: DAGs with |V|=k
Topological Order: $v_1 \prec \cdots \prec v_k$



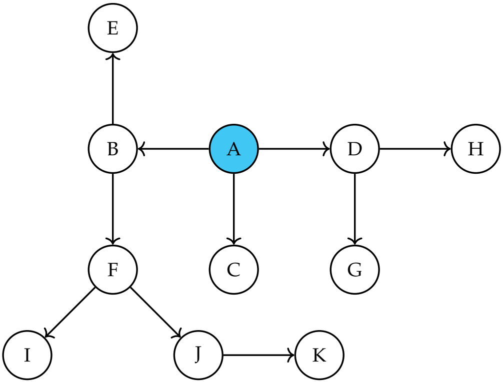
Ind. Step: DAGs with |V|=k+1

Topological Order: $v_1 \prec \cdots \prec v_k \prec v_{new}$



Topological Ordering

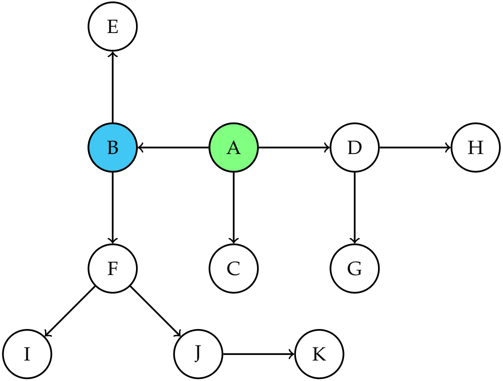
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq$

Topological Ordering

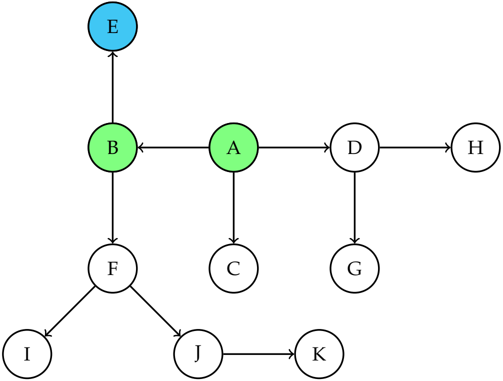
Visualization of Induction



$\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq$

Topological Ordering

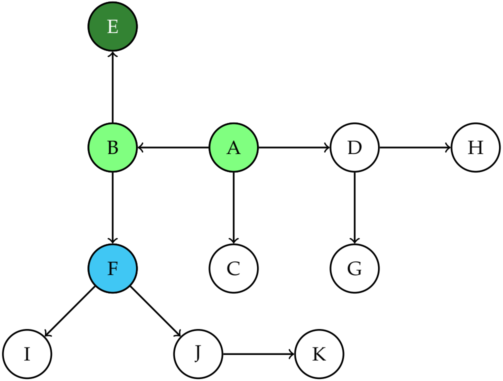
Visualization of Induction



$\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq\preceq$

Topological Ordering

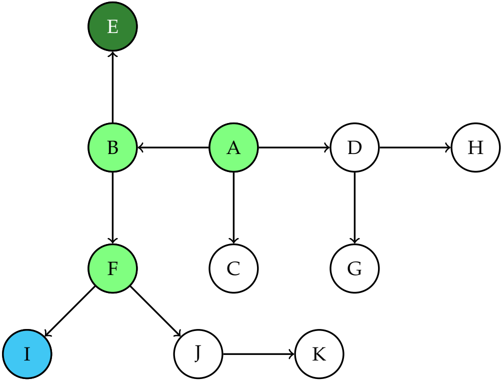
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq$ E

Topological Ordering

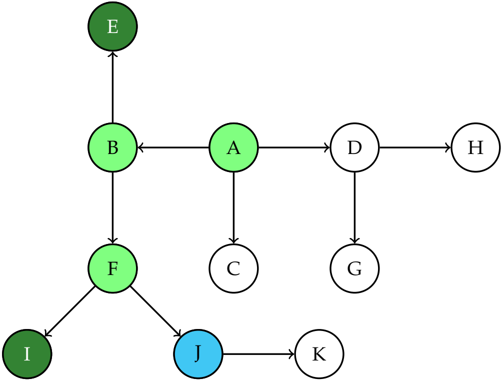
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq$ E

Topological Ordering

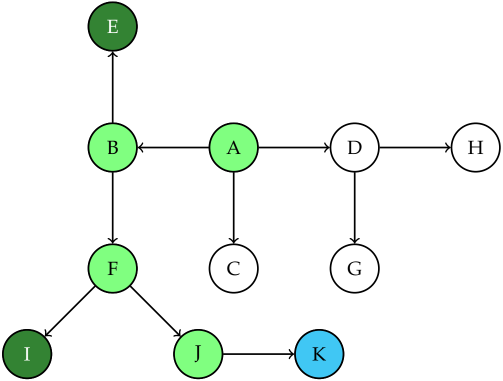
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq$ I $\preceq$ E

Topological Ordering

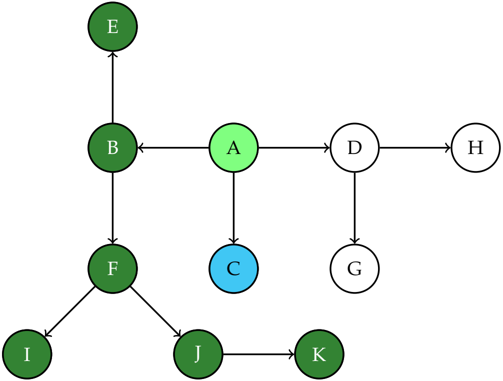
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq \preceq \preceq \preceq$ K $\preceq$ I $\preceq$ E

Topological Ordering

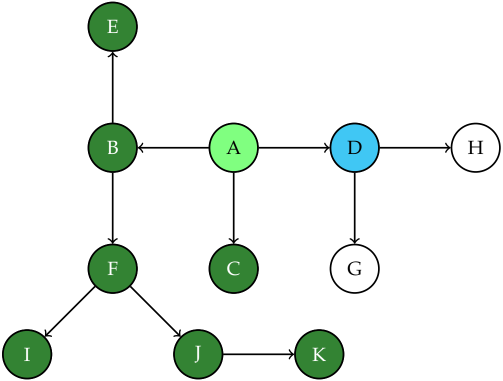
Visualization of Induction



$\preceq \preceq \preceq \preceq \preceq B \preceq F \preceq J \preceq K \preceq I \preceq E$

Topological Ordering

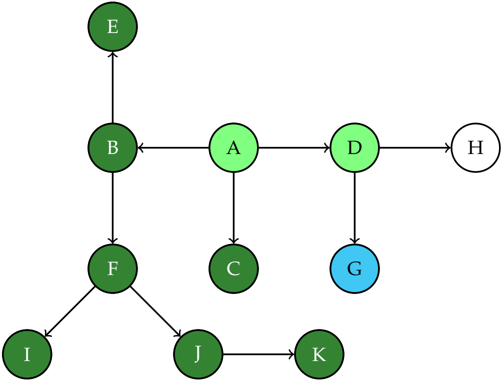
Visualization of Induction



$\preceq \preceq \preceq \preceq C \preceq B \preceq F \preceq J \preceq K \preceq I \preceq E$

Topological Ordering

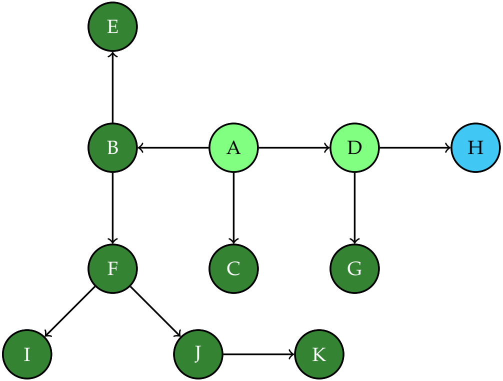
Visualization of Induction



$\leq \leq \leq \leq C \leq B \leq F \leq J \leq K \leq I \leq E$

Topological Ordering

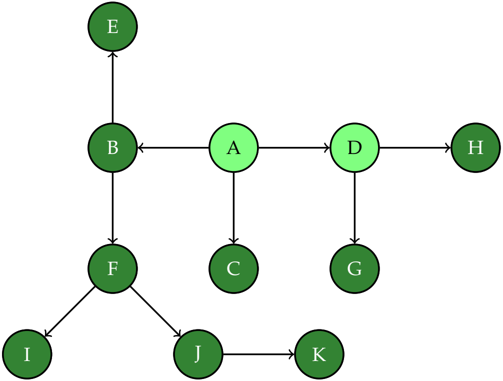
Visualization of Induction



$\leq \leq \leq G \leq C \leq B \leq F \leq J \leq K \leq I \leq E$

Topological Ordering

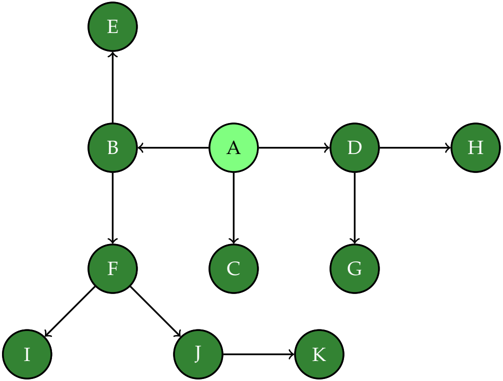
Visualization of Induction



$\leq \leq H \leq G \leq C \leq B \leq F \leq J \leq K \leq I \leq E$

Topological Ordering

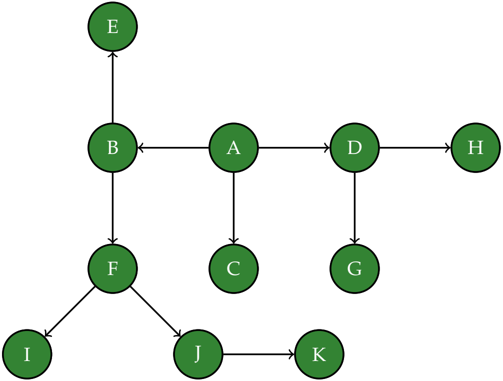
Visualization of Induction



$\leq D \leq H \leq G \leq C \leq B \leq F \leq J \leq K \leq I \leq E$

Topological Ordering

Visualization of Induction



$$A \le D \le H \le G \le C \le B \le F \le J \le K \le I \le E$$

Appendix

References

Image Sources I

WISCONSIN UNIVERSITY OF WISCONSIN–MADISON https://brand.wisc.edu/web/logos/