算法设计与分析(2024年春季学期)
Graph Basics

授课老师: 栗师

南京大学计算机科学与技术系

Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

Examples of Graphs

Figure: Road Networks

Figure: Internet

Figure: Social Networks

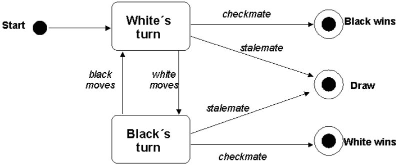
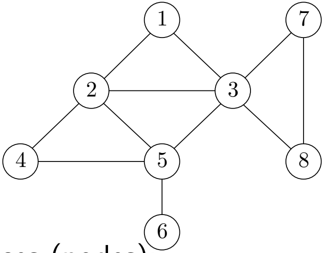


Figure: Transition Graphs

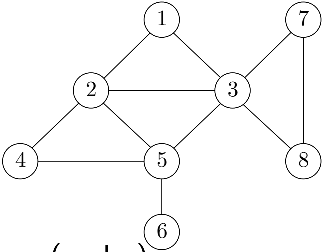
(Undirected) Graph $G = (V, E)$



- V: set of vertices (nodes);

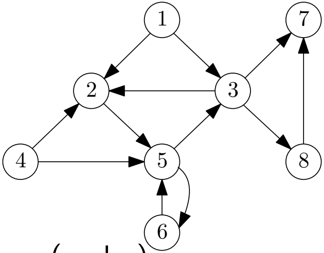
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2

(Undirected) Graph $G = (V, E)$



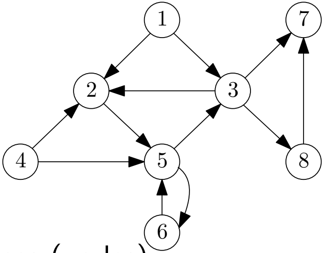
- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2
  - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$

Directed Graph $G = (V, E)$



- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - directed graphs: relationship is asymmetric, E contains ordered pairs

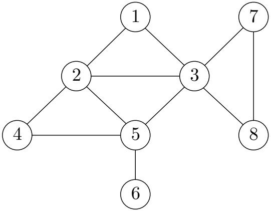
Directed Graph $G = (V, E)$



- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - directed graphs: relationship is asymmetric, E contains ordered pairs
  - $E = \{(1, 2), (1, 3), (3, 2), (4, 2), (2, 5), (5, 3), (3, 7), (3, 8), (4, 5), (5, 6), (6, 5), (8, 7)\}$

Abuse of Notations

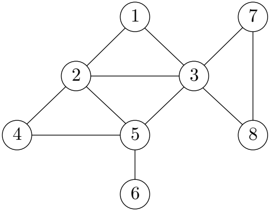
- For (undirected) graphs, we often use (i, j) to denote the set $\{i, j\}$.
- We call (i, j) an unordered pair; in this case $(i, j) = (j, i)$.



- $E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}$

- Social Network : Undirected
- Transition Graph : Directed
- Road Network : Directed or Undirected
- Internet : Directed or Undirected

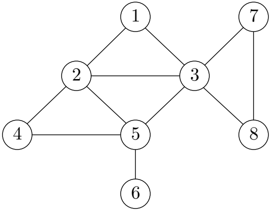
Representation of Graphs

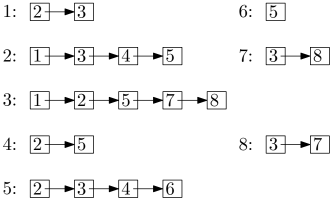


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected

Representation of Graphs





1: 2 → 3

2: 1 → 3 → 4 → 5

3: 1 → 2 → 5 → 7 → 8

4: 2 → 5

5: 2 → 3 → 4 → 6

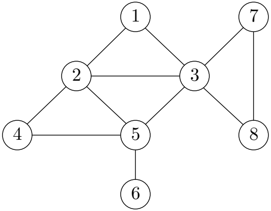
6: 5

7: 3 → 8

8: 3 → 7

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbours of v.

Representation of Graphs



1: 2 3

2: 1 3 4 5

3: 1 2 5 7 8

4: 2 5

5: 2 3 4 6

6: 5

7: 3 8

8: 3 7

$d : (2, 4, 5, 2, 4, 1, 2, 2)$

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbours of v.
- If graph is static: store neighbors of all vertices in a length-$2m$ array, where the neighbors of any vertex are consecutive.

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage		
time to check $(u,v) \in E$		
time to list all neighbours of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \leq m \leq n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	
time to check $(u, v) \in E$		
time to list all neighbours of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n - 1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$		
time to list all neighbours of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n - 1 \leq m \leq n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	
time to list all neighbours of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbours of v		

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n - 1 \le m \le n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbours of v	$O(n)$	

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \leq m \leq n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbours of v	$O(n)$	$O(d_v)$

Outline

1. Graphs

2. Connectivity and Graph Traversal
    - Testing Bipartiteness

3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
    - $O(n+m)$-Time Algorithm to Find Bridges
    - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
    - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

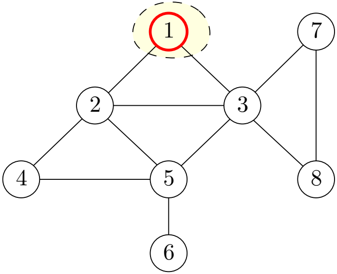
- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)

Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j

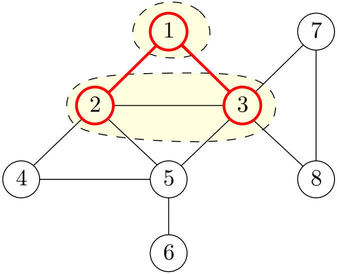
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



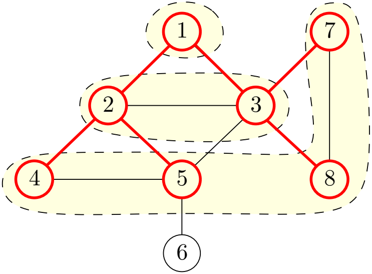
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



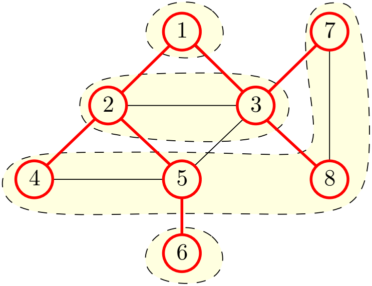
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



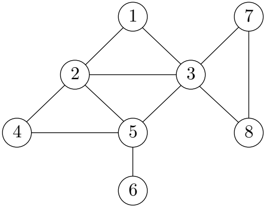
Implementing BFS using a Queue

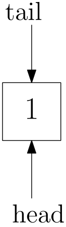
BFS(s)

```
$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as "visited" and all other vertices as "unvisited"
while $head \leq tail$ do
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbours $u$ of $v$ do
        if $u$ is "unvisited" then
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark $u$ as "visited"
```

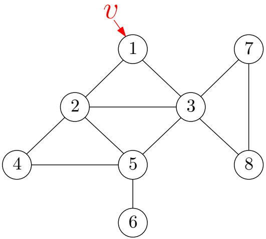
- Running time: $O(n + m)$.

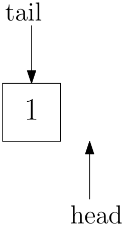
Example of BFS via Queue





Example of BFS via Queue

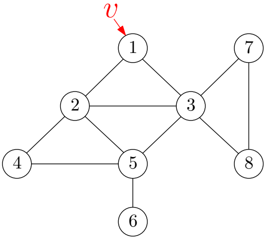


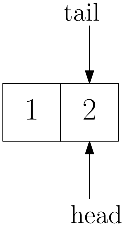

tail

1

head

Example of BFS via Queue



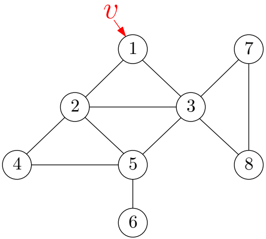


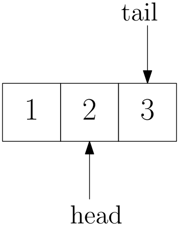
tail

1	2

head

Example of BFS via Queue



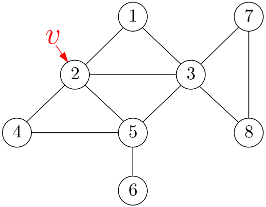


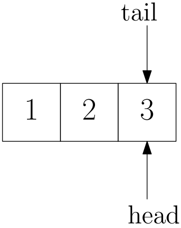
tail

1	2	3

head

Example of BFS via Queue



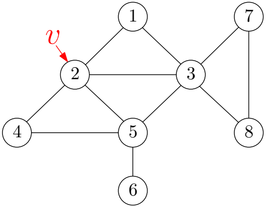


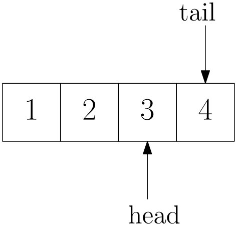
tail

1	2	3

head

Example of BFS via Queue



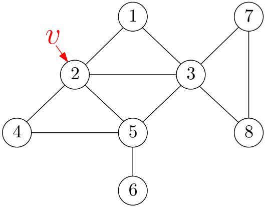


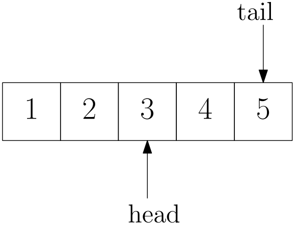
tail

1	2	3	4

head

Example of BFS via Queue



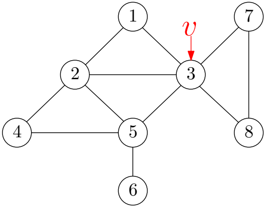


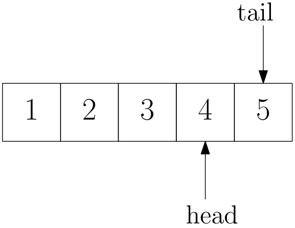
tail

1	2	3	4	5

head

Example of BFS via Queue

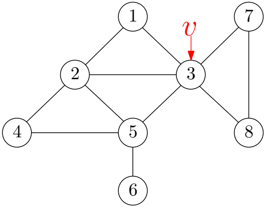


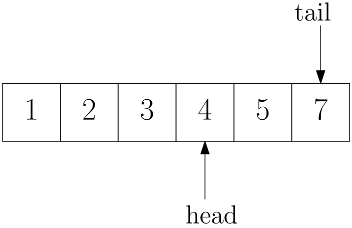

tail

1	2	3	4	5

head

Example of BFS via Queue



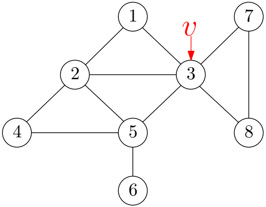


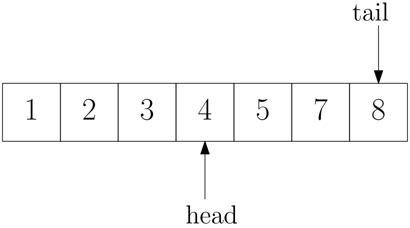
tail

1	2	3	4	5	7

head

Example of BFS via Queue



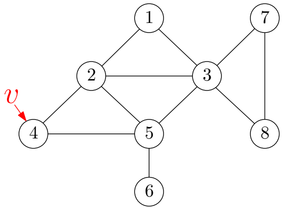


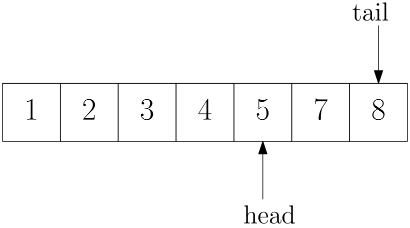
tail

1	2	3	4	5	7	8

head

Example of BFS via Queue



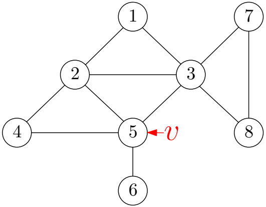


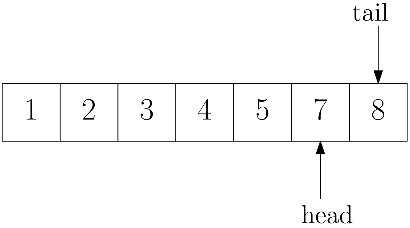
tail

1	2	3	4	5	7	8

head

Example of BFS via Queue



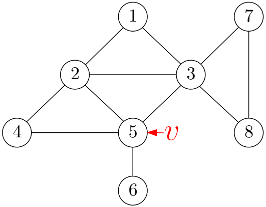


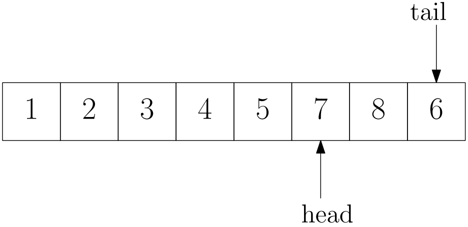
tail

1	2	3	4	5	7	8

head

Example of BFS via Queue



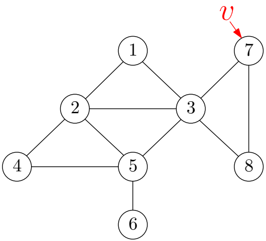


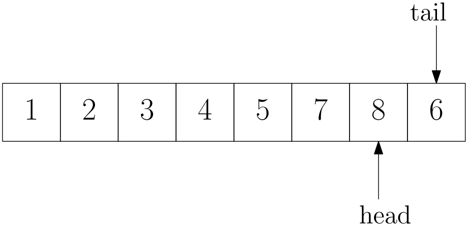
tail

1	2	3	4	5	7	8	6

head

Example of BFS via Queue

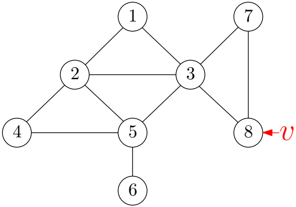


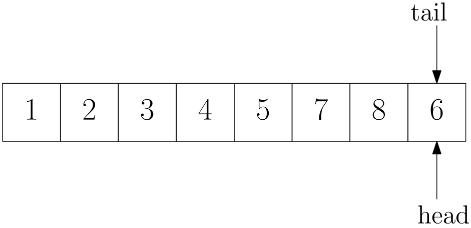

tail

1	2	3	4	5	7	8	6

head

Example of BFS via Queue

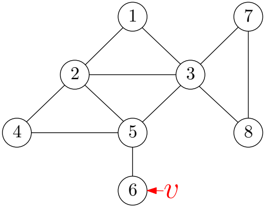


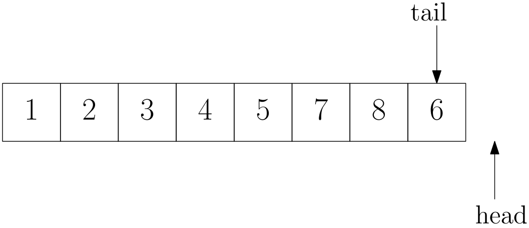

tail

1	2	3	4	5	7	8	6

head

Example of BFS via Queue




tail

1	2	3	4	5	7	8	6

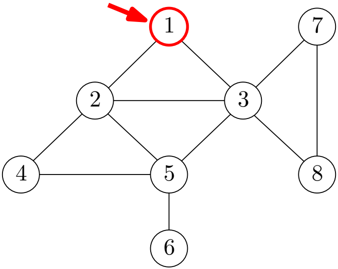
head

Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

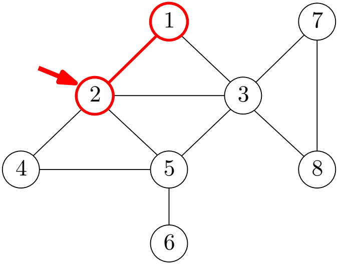
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



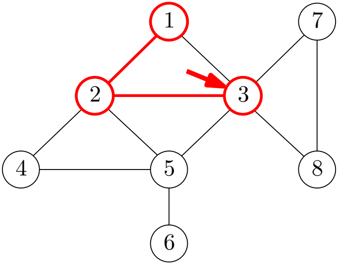
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



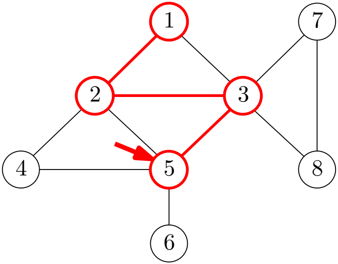
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



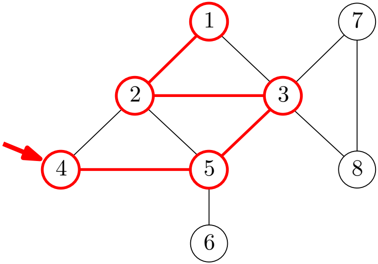
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



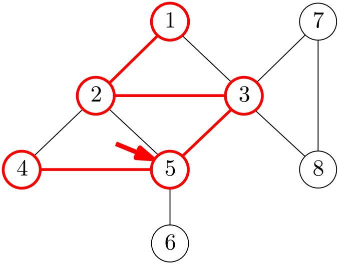
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



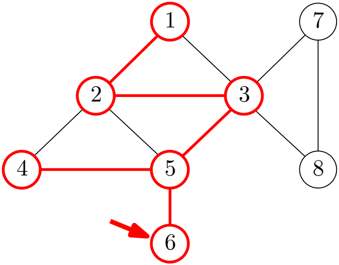
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



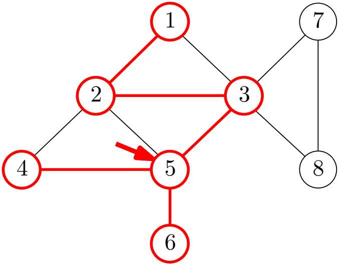
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



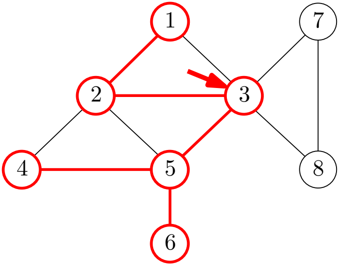
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



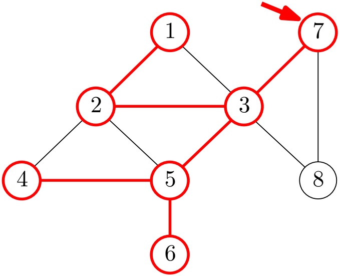
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



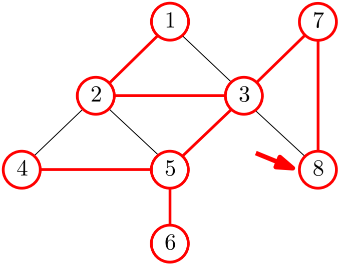
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Implementing DFS using Recurrsion

DFS(s)

```
mark all vertices as "unvisited"
recursive-DFS($s$)
```

recursive-DFS(v)

```
mark $v$ as "visited"
for all neighbours $u$ of $v$ do
    if $u$ is unvisited then recursive-DFS($u$)
```

Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

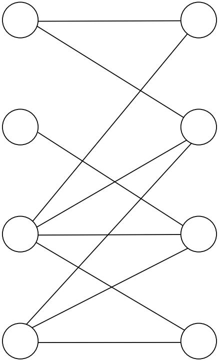
3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

Testing Bipartiteness: Applications of BFS

Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$.



Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$

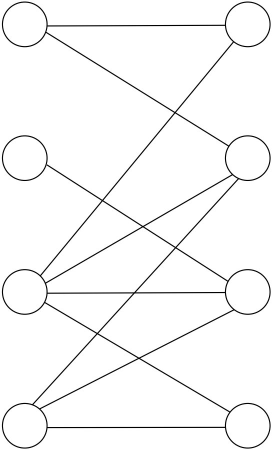
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found

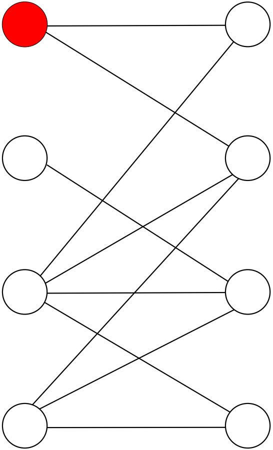
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found
- If G contains multiple connected components, repeat above algorithm for each component

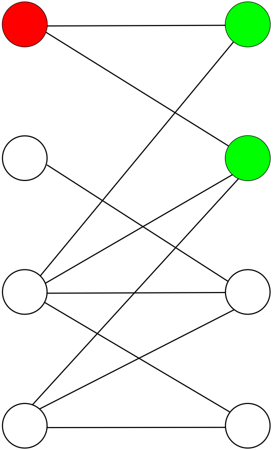
Test Bipartiteness



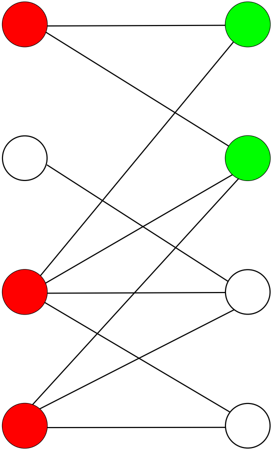
Test Bipartiteness



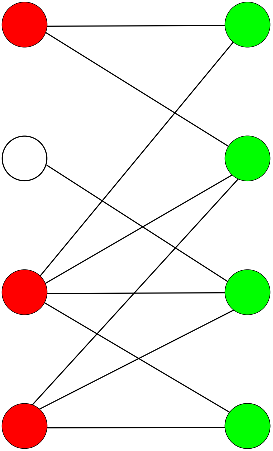
Test Bipartiteness



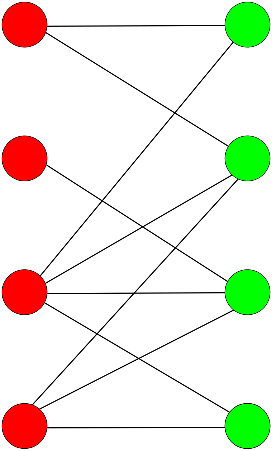
Test Bipartiteness



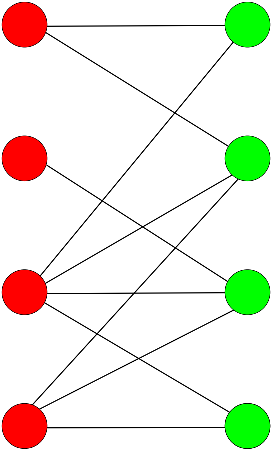
Test Bipartiteness

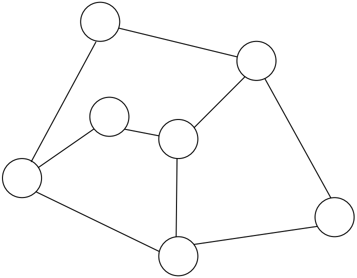


Test Bipartiteness

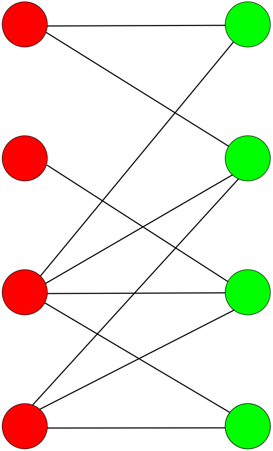


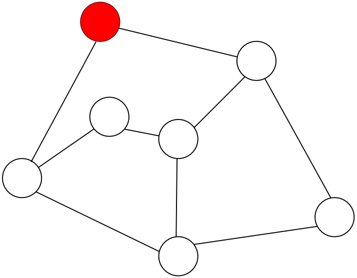
Test Bipartiteness



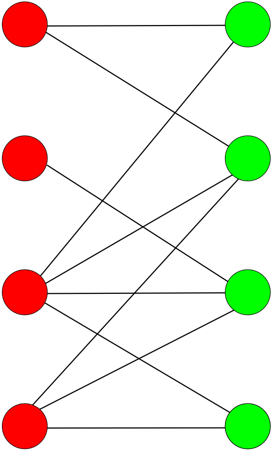


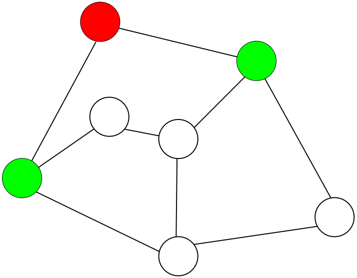
Test Bipartiteness



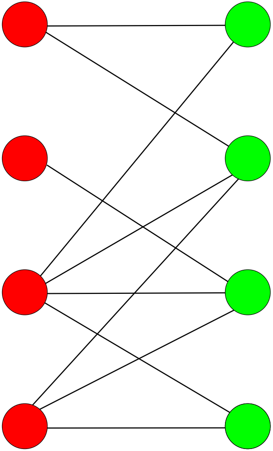


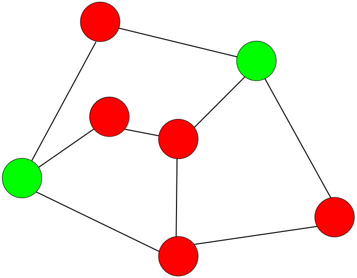
Test Bipartiteness



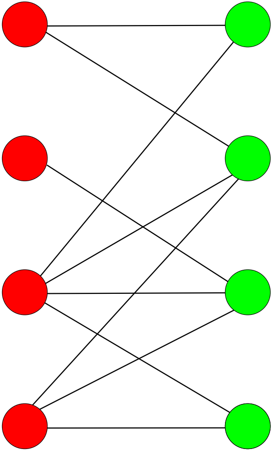


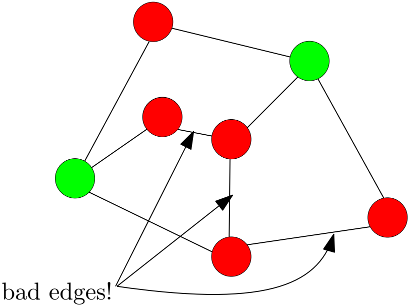
Test Bipartiteness





Test Bipartiteness





Testing Bipartiteness using BFS

BFS(s)

```
$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as “visited” and all other vertices as “unvisited”
**while** $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    **for** all neighbours $u$ of $v$ **do**
        **if** $u$ is “unvisited” **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark $u$ as “visited”
```

Testing Bipartiteness using BFS

test-bipartiteness(s)

```
$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark $s$ as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
while $head \leq tail$ do
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbours $u$ of $v$ do
        if $u$ is "unvisited" then
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark $u$ as "visited"
            $color[u] \leftarrow 1 - color[v]$
        else if $color[u] = color[v]$ then
            print("$G$ is not bipartite") and exit
```

Testing Bipartiteness using BFS

```
mark all vertices as “unvisited”
for each vertex $v \in V$ do
    if $v$ is “unvisited” then
        test-bipartiteness($v$)
print(“$G$ is bipartite”)
```

Testing Bipartiteness using BFS

```
mark all vertices as “unvisited”
for each vertex $v \in V$ do
    if $v$ is “unvisited” then
        test-bipartiteness($v$)
print(“$G$ is bipartite”)
```

Obs. Running time of algorithm $= O(n + m)$

Outline

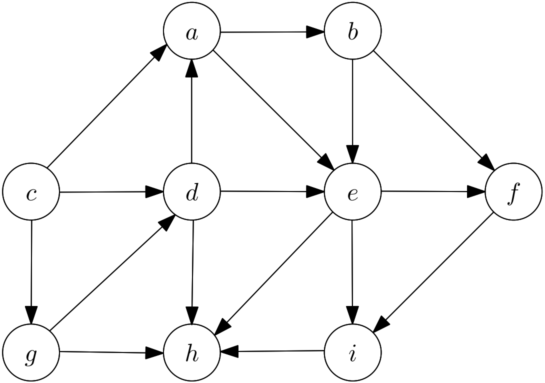
1. Graphs
2. Connectivity and Graph Traversal
   - Testing Bipartiteness
3. Topological Ordering
4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices
5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$

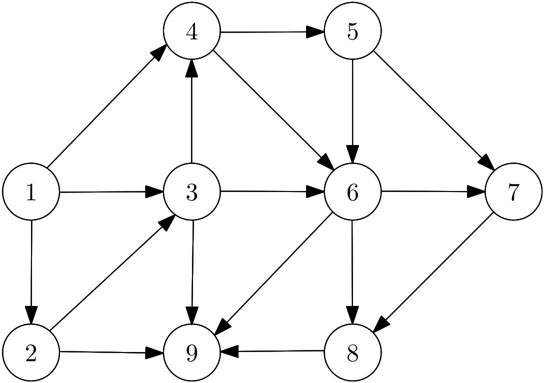


Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

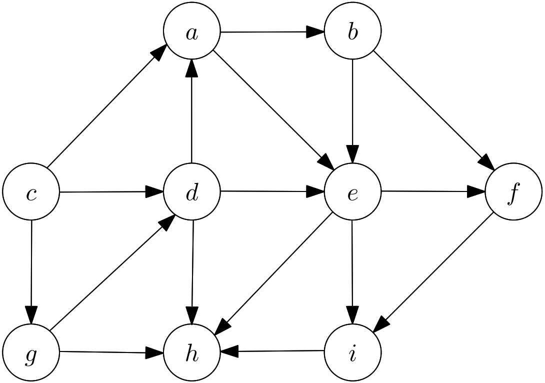
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



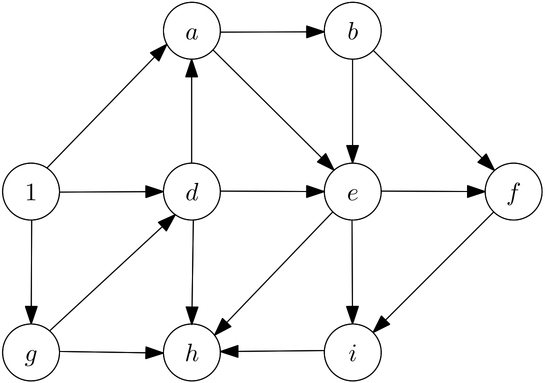
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



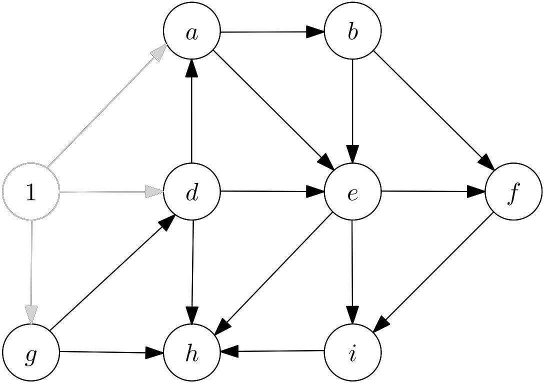
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



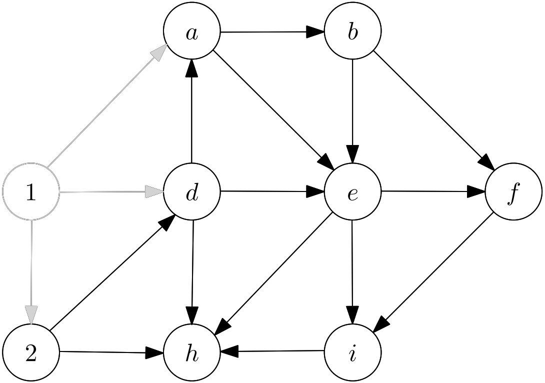
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



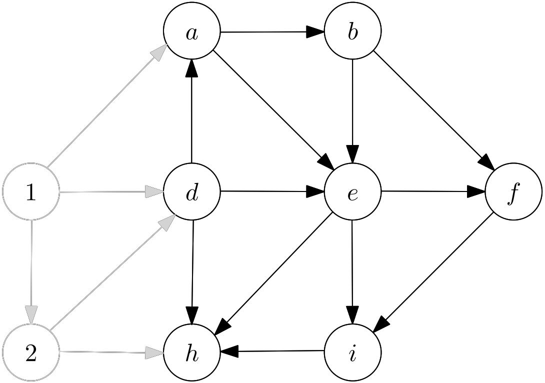
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



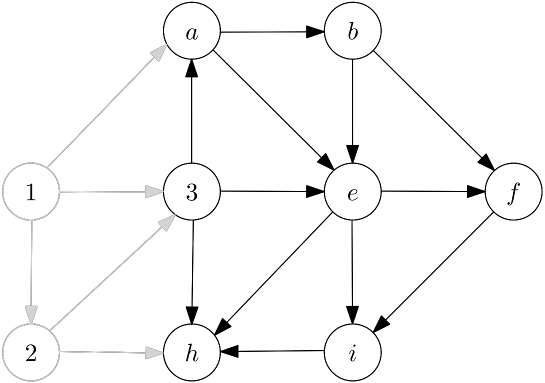
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



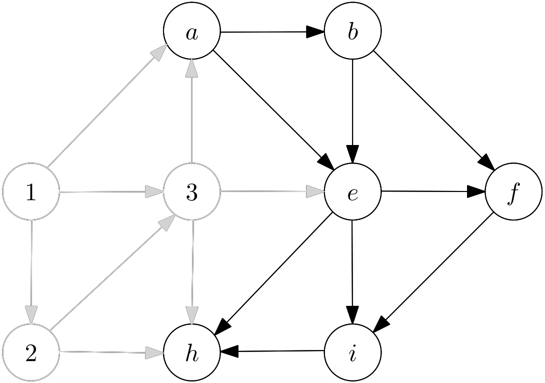
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



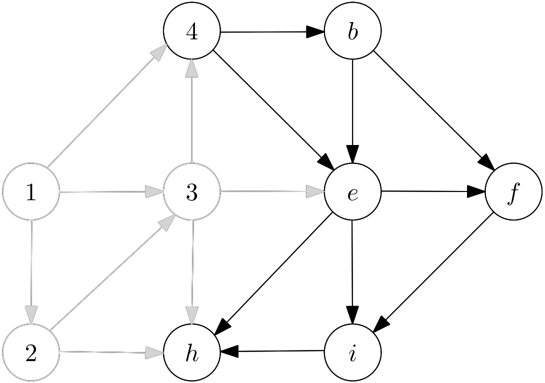
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



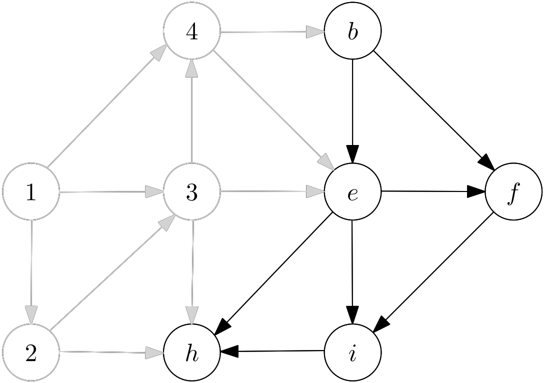
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



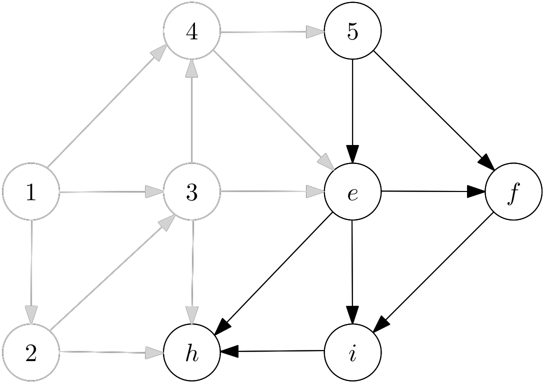
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



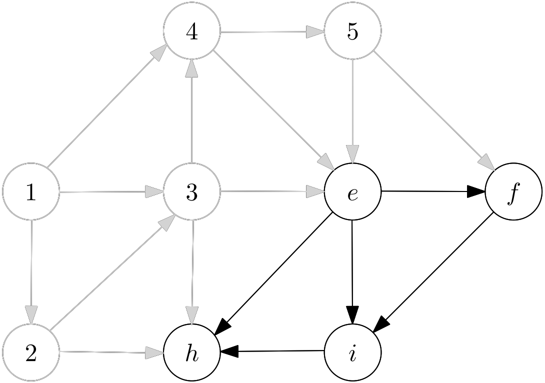
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



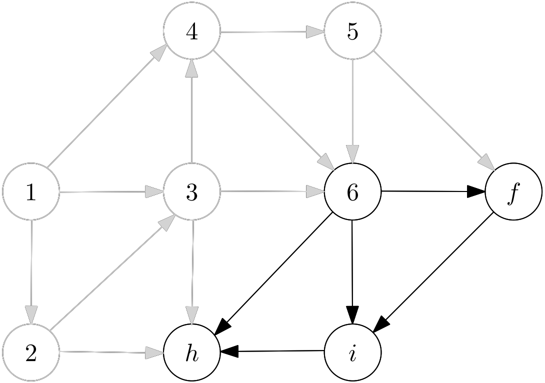
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



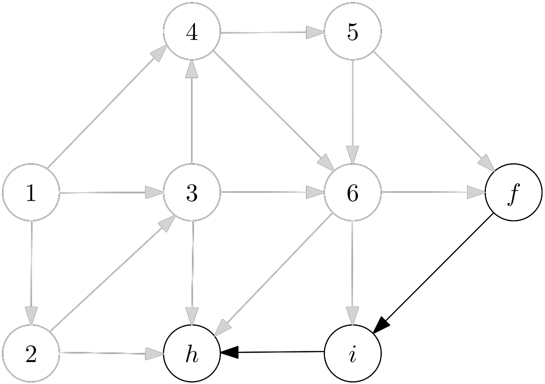
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



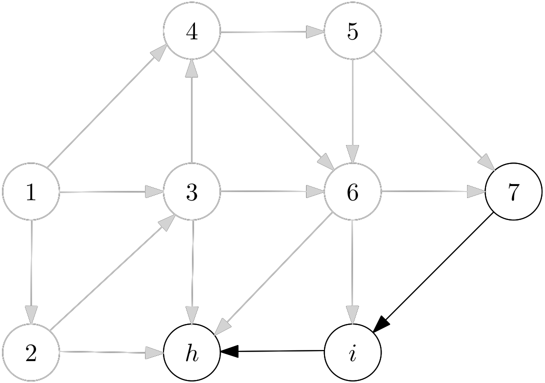
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



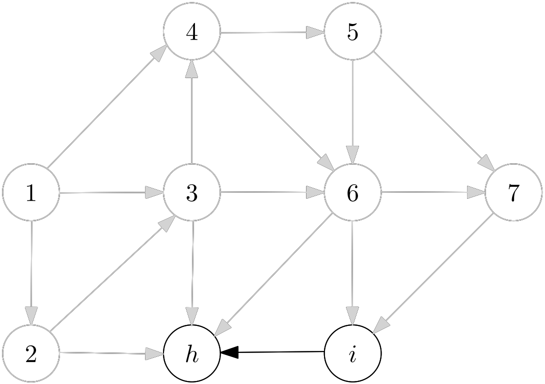
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



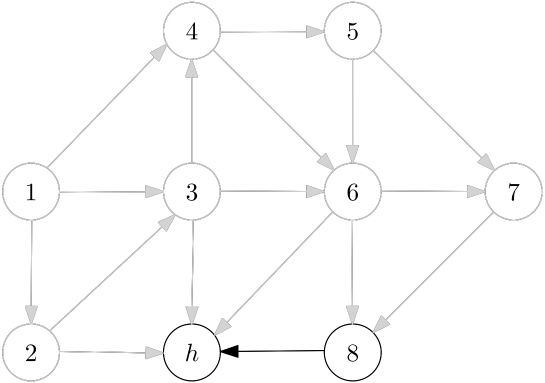
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



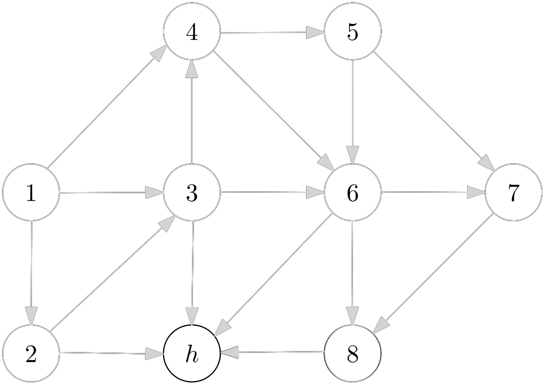
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



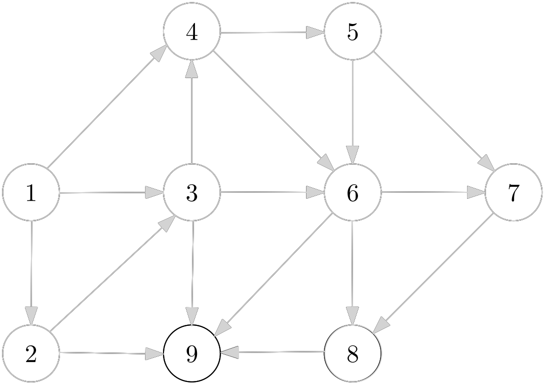
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



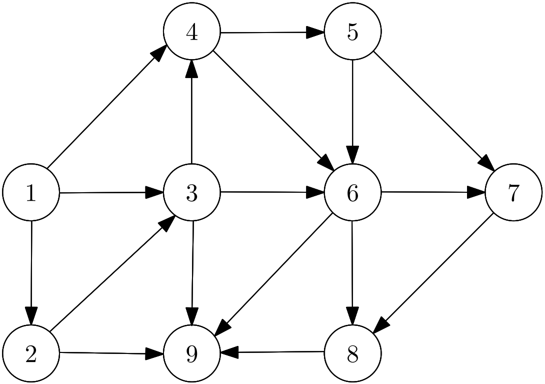
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:

- Use linked-lists of outgoing edges
- Maintain the in-degree d_v of vertices
- Maintain a queue (or stack) of vertices v with $d_v = 0$

topological-sort(G)

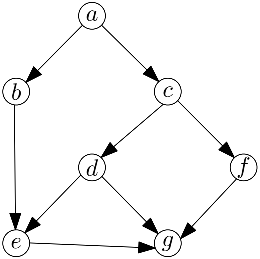
```
let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$ do
    for every $u$ such that $(v, u) \in E$ do
        $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$ do
    $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
    for every $u$ such that $(v, u) \in E$ do
        $d_u \leftarrow d_u - 1$
        if $d_u = 0$ then add $u$ to $S$
if $i < n$ then output "not a DAG"
```

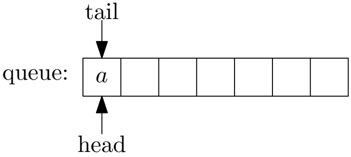
- S can be represented using a queue or a stack
- Running time $= O(n + m)$

S as a Queue or a Stack

DS	Queue	Stack
Initialization	$head \leftarrow 1$, $tail \leftarrow 0$	$top \leftarrow 0$
Non-Empty?	$head \leq tail$	$top > 0$
Add(v)	$tail \leftarrow tail + 1$ $S[tail] \leftarrow v$	$top \leftarrow top + 1$ $S[top] \leftarrow v$
Retrieve v	$v \leftarrow S[head]$ $head \leftarrow head + 1$	$v \leftarrow S[top]$ $top \leftarrow top - 1$

Example





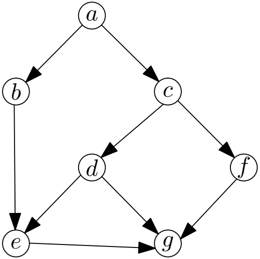
tail

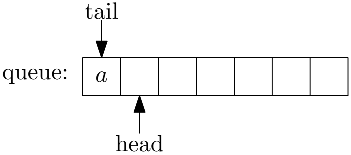
queue: a

head

	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

Example





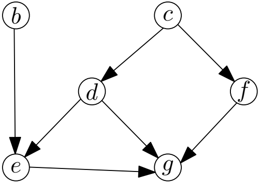
tail

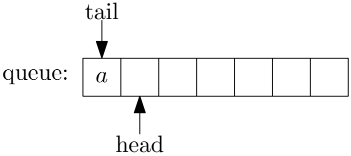
queue: a

head

	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

Example



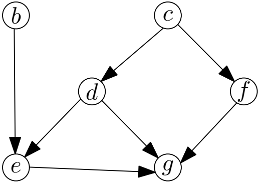

tail

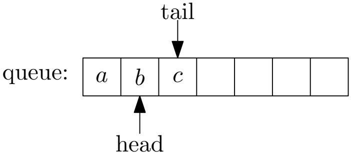
queue: a

head

	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

Example



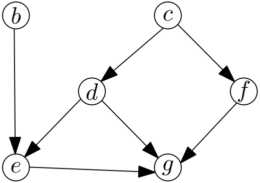

tail

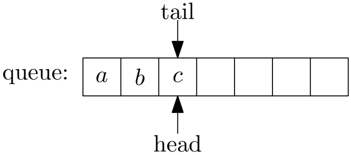
queue:	a	b	c				

head

	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

Example





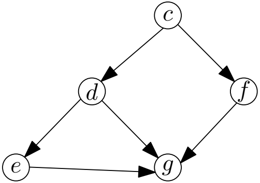
tail

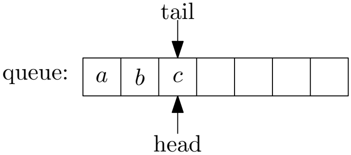
queue: | a | b | c | | | | |

head

	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

Example



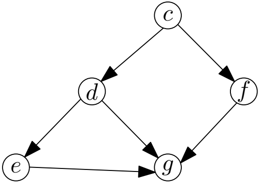

tail

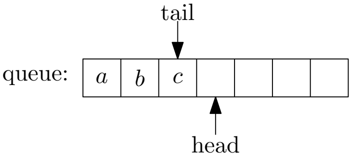
queue:	a	b	c				

head

	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

Example



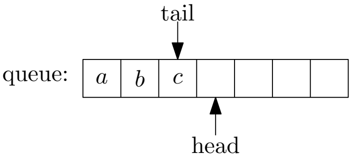

tail

queue:	a	b	c				

head

	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

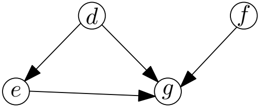
Example



tail

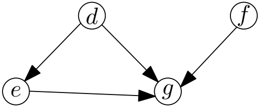
queue: | a | b | c | | | | |

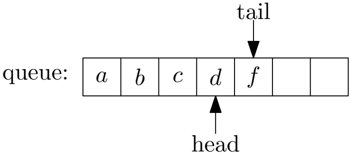
head



	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example





tail

queue:	a	b	c	d	f		

head

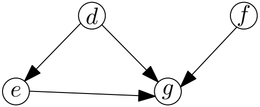
	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example

tail

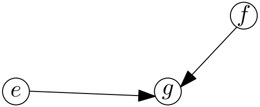
queue:	a	b	c	d	f		

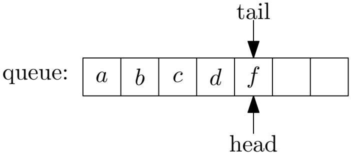
head



	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

Example





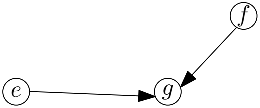
tail

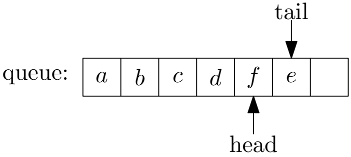
queue:	a	b	c	d	f		

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

Example



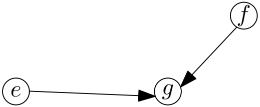

tail

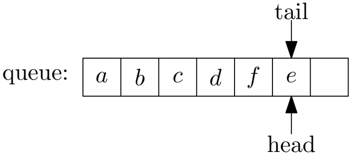
queue: | a | b | c | d | f | e | |

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

Example





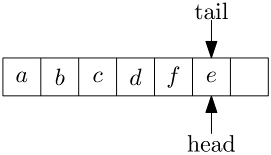
tail

queue:	a	b	c	d	f	e	

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

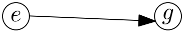
Example


tail

queue:

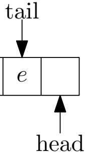
a	b	c	d	f	e	

head



	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

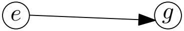
Example



tail

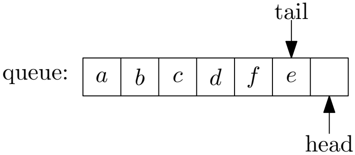
queue:	a	b	c	d	f	e	

head



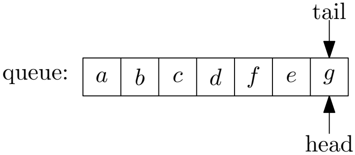
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example



	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example



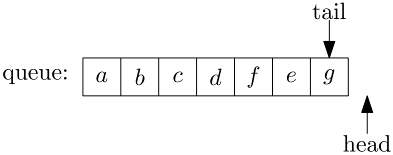
tail

queue:	a	b	c	d	f	e	g

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example



tail

queue:	a	b	c	d	f	e	g

head

	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Outline

1 Graphs

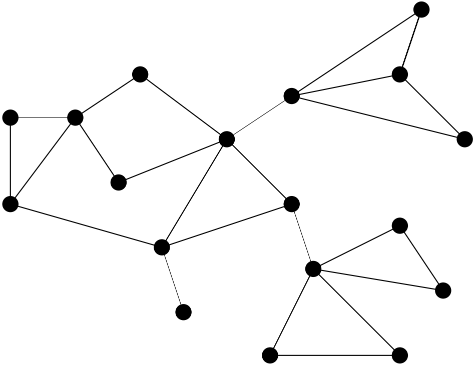
2 Connectivity and Graph Traversal
- Testing Bipartiteness

3 Topological Ordering

4 Bridges and 2-Edge-Connected Components
- $O(n+m)$-Time Algorithm to Find Bridges
- Related Concept: Cut Vertices

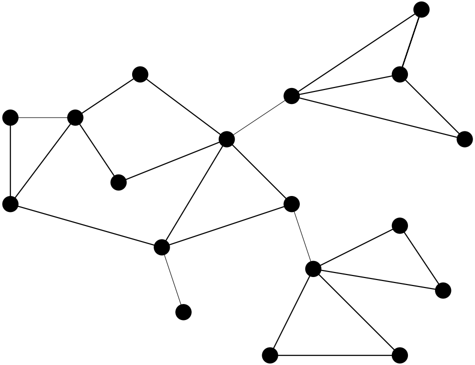
5 Strong Connectivity in Directed Graphs
- Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.

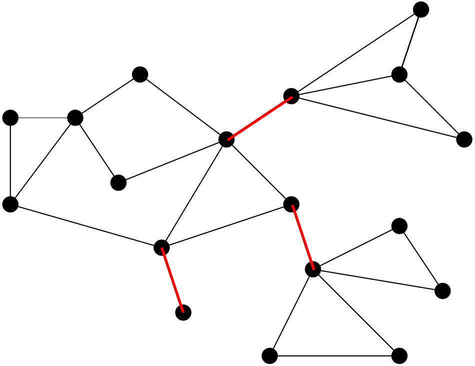


Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.

- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

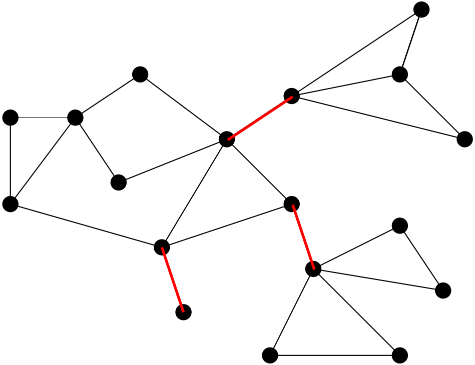


Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

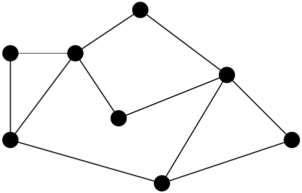
Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

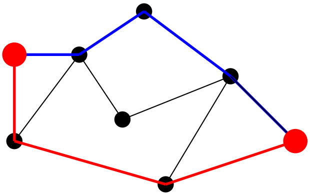
Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

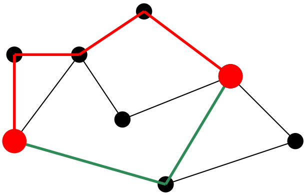
Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

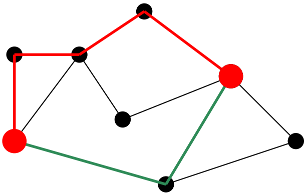
Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.



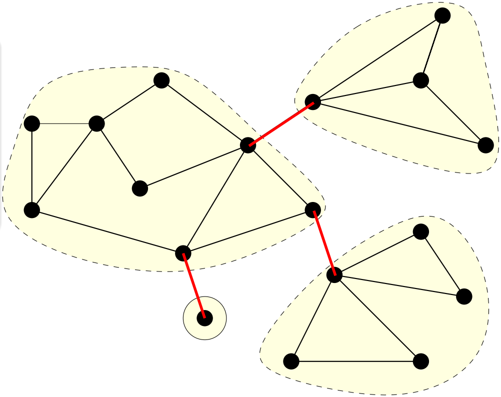
- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.

Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

Lemma Let B be the set of bridges in a graph $G = (V, E)$. Then, every connected component in $(V, E \setminus B)$ is 2-edge-connected. Every such component is called a 2-edge-connected component of G.

Def. Given $G = (V, E)$, $e \in E$ is called a bridge if the removal of e from G will increase its number of connected components.

- When G is connected, $e \in E$ is a bridge iff its removal will disconnect G.



Def. A graph $G = (V, E)$ is 2-edge-connected if for every two $u, v \in V$, there are two edge disjoint paths connecting u and v.

Lemma Let B be the set of bridges in a graph $G = (V, E)$. Then, every connected component in $(V, E \setminus B)$ is 2-edge-connected. Every such component is called a 2-edge-connected component of G.

Outline

1 Graphs

2 Connectivity and Graph Traversal
- Testing Bipartiteness

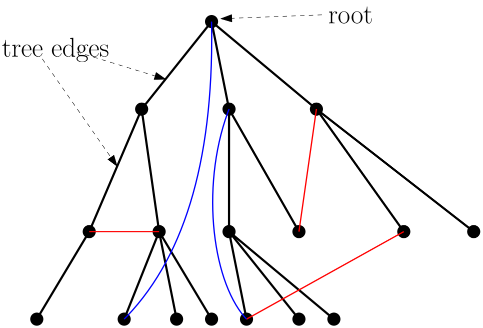
3 Topological Ordering

4 Bridges and 2-Edge-Connected Components
- $O(n+m)$-Time Algorithm to Find Bridges
- Related Concept: Cut Vertices

5 Strong Connectivity in Directed Graphs
- Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

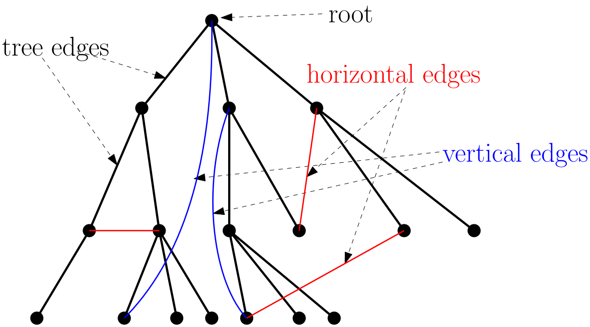
Vertical and Horizontal Edges

- $G = (V, E)$: connected graph
- $T = (V, E_T)$: rooted spanning tree of G

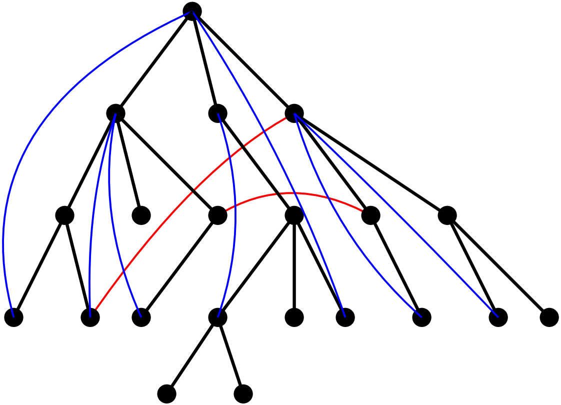


Vertical and Horizontal Edges

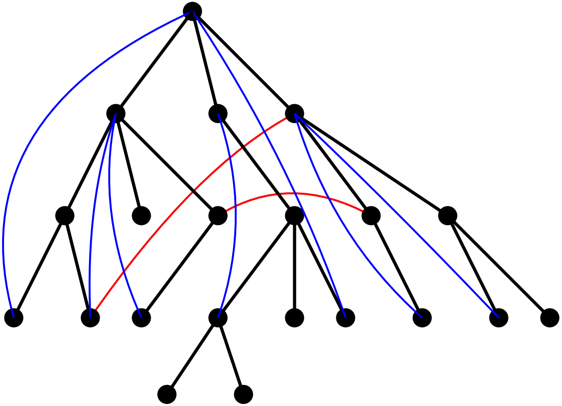
- $G = (V, E)$: connected graph
- $T = (V, E_T)$: rooted spanning tree of G
- $(u, v) \in E \setminus E_T$ is
  - vertical if one of u and v is an ancestor of the other in T,
  - horizontal otherwise.



- $G = (V, E)$: connected graph T: a DFS tree for G

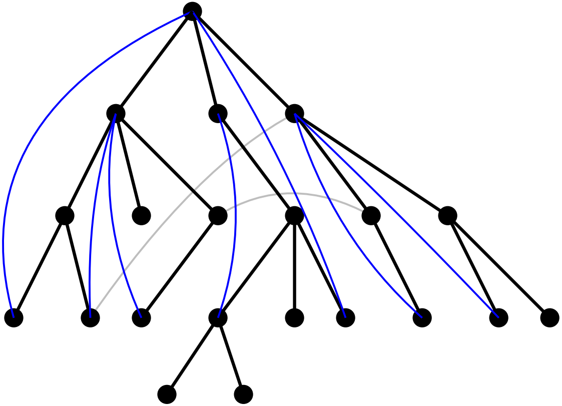


- $G = (V, E)$: connected graph T: a DFS tree for G



Q: Can there be a horizontal edges (u, v) w.r.t T?

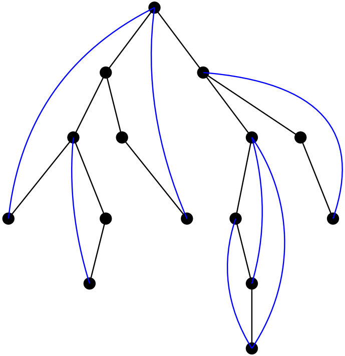
- $G = (V, E)$: connected graph T: a DFS tree for G



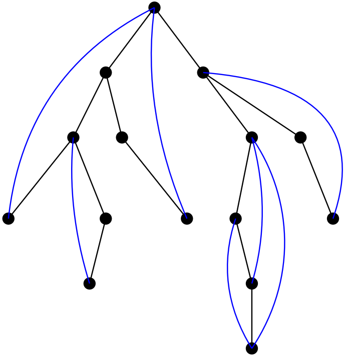
Q: Can there be a horizontal edges (u, v) w.r.t T?

A: No!

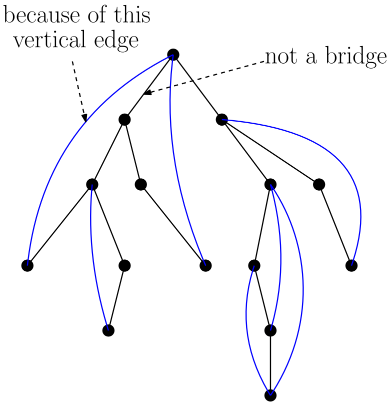
- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges



- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges
- vertical edges: not bridges



- $G = (V, E)$: connected graph
- T: a DFS tree for G
- G contains only tree and vertical edges
- vertical edges: not bridges

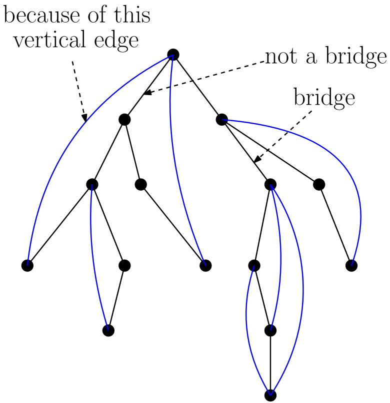


Lemma

- $(u, v) \in T$, u is parent
- (u, v) is not a bridge $\Longleftrightarrow \exists$ vertical edge connecting an (inclusive) descendant of v and an (inclusive) ancestor of u

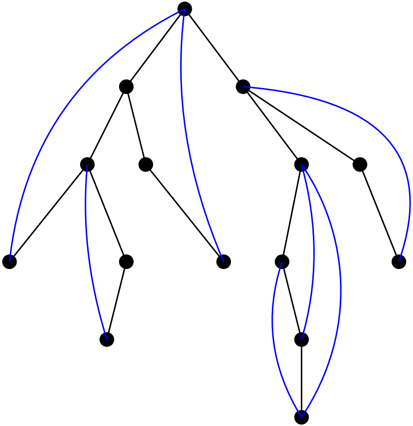
- $G = (V, E)$: connected graph
- T: a DFS tree for G

- G contains only tree and vertical edges
- vertical edges: not bridges

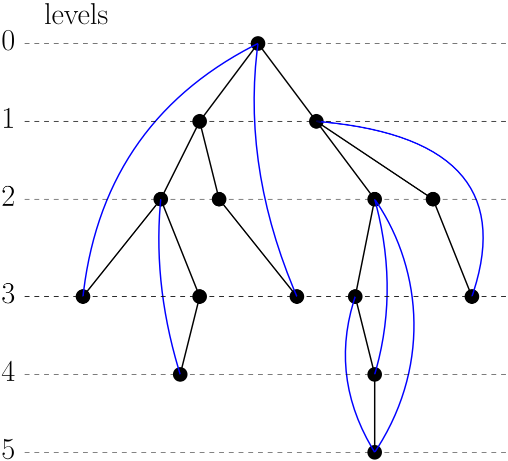


Lemma

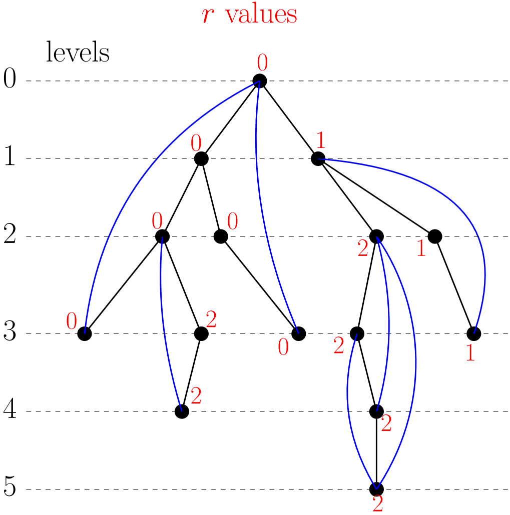
- $(u, v) \in T$, u is parent
- (u, v) is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of v and an (inclusive) ancestor of u



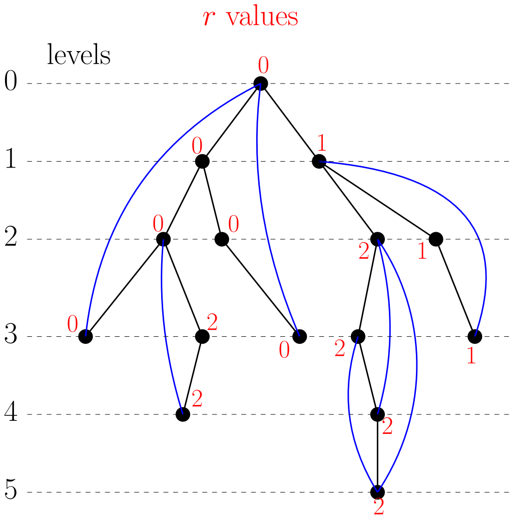
- $v.l$: the level of vertex v in DFS tree



- $v.l$: the level of vertex v in DFS tree
- T_v: subtree rooted at v
- $v.r$: the smallest level that can be reached by a vertical edge from T_v



- $v.l$: the level of vertex v in DFS tree
- T_v: subtree rooted at v
- $v.r$: the smallest level that can be reached by a vertical edge from T_v
- $(parent(u), u)$ is a bridge if and only if $u.r \geq u.l$.



recursive-DFS(v)

```
mark $v$ as “visited”
$v.r \leftarrow \infty$
for all neighbours $u$ of $v$ do
    if $u$ is unvisited then                              ▷ $u$ is a child of $v$
        $u.l \leftarrow v.l + 1$
        recursive-DFS($u$)
        if $u.r \geq u.l$ then claim $(v, u)$ is a bridge
        if $u.r < v.r$ then $v.r \leftarrow u.r$
    else if $u.l < v.l - 1$ then              ▷ $u$ is ancestor but not parent
        if $u.l < v.r$ then $v.r \leftarrow u.l$
```

finding-bridges

```
mark all vertices as “unvisited”
for every $v \in V$ do
    if $v$ is unvisited then
        $v.l \leftarrow 0$
        recursive-DFS($v$)
```

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

Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

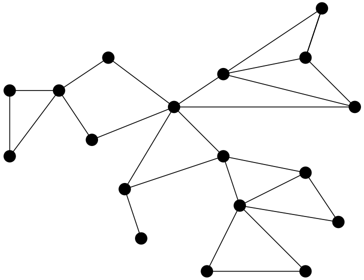
3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

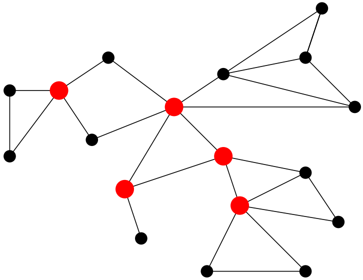
Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.



Cut vertices

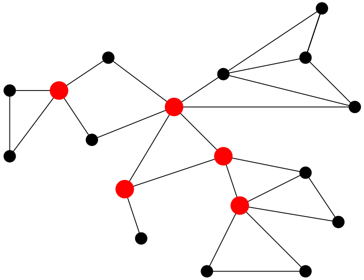
Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.



Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.

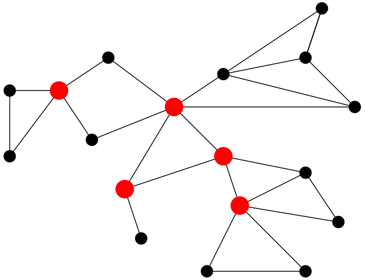
Def. A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between u and v.

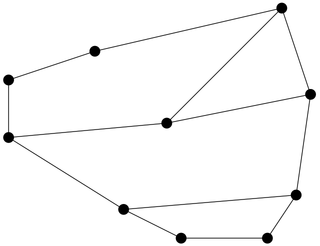


Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.

Def. A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between u and v.

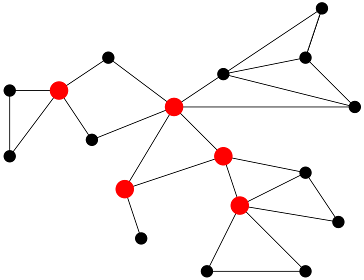


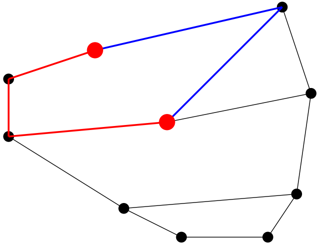


Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.

Def. A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between u and v.

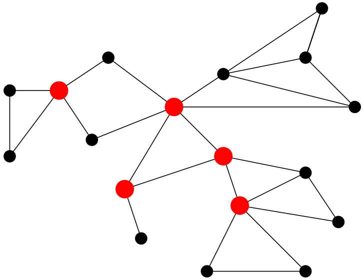


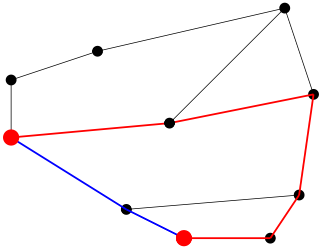


Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.

Def. A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between u and v.



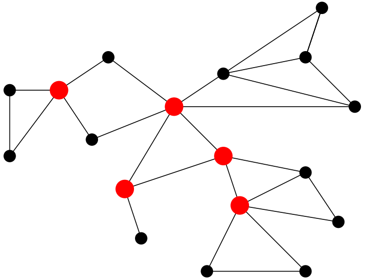


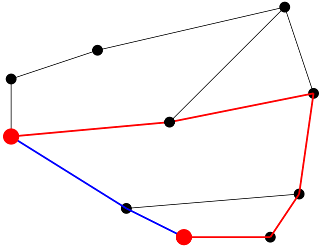
Cut vertices

Def. A vertex is a cut vertex of $G = (V, E)$ if its removal will increase the number of connected components of G.

Def. A graph $G = (V, E)$ is 2-(vertex-)connected (or biconnected) if for every $u, v \in V$, there are 2 internally-disjoint paths between u and v.

Lemma A graph $G = (V, E)$ with $|V| \geq 3$ does not contain a cut vertex, if and only if it is biconnected.





Q: How can we find the cut vertices?

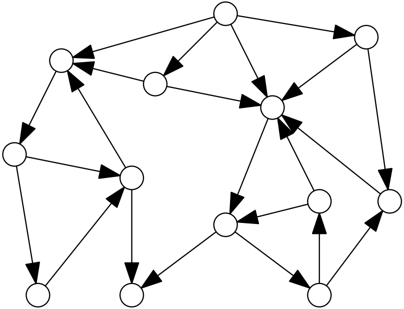
Q: How can we find the cut vertices?

A: With a small modification to the algorithm for finding bridges.

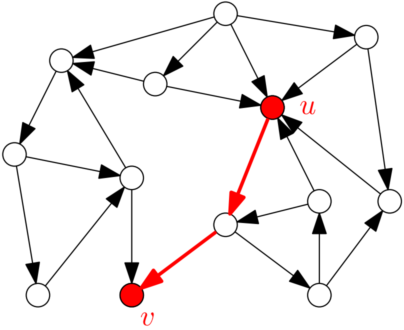
Outline

1. Graphs
2. Connectivity and Graph Traversal
   - Testing Bipartiteness
3. Topological Ordering
4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices
5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

- directed graph $G = (V, E)$.

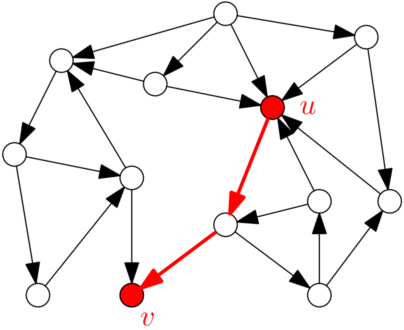


- directed graph $G = (V, E)$.
- it may happen: there is a $u \to v$ path, but no $v \to u$ path.



- directed graph $G = (V, E)$.
- it may happen: there is a $u \to v$ path, but no $v \to u$ path.

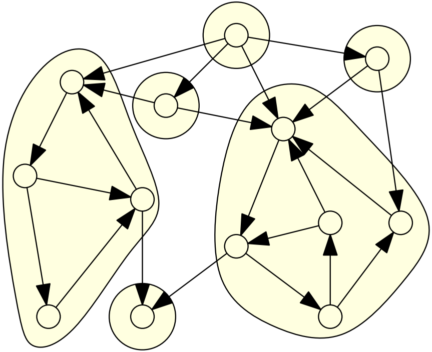
Def. A directed graph $G = (V, E)$ is strongly connected if for every $u, v \in V$, there is a path from u to v in G.



- directed graph $G = (V, E)$.
- it may happen: there is a $u \to v$ path, but no $v \to u$ path.

Def. A directed graph $G = (V, E)$ is strongly connected if for every $u, v \in V$, there is a path from u to v in G.

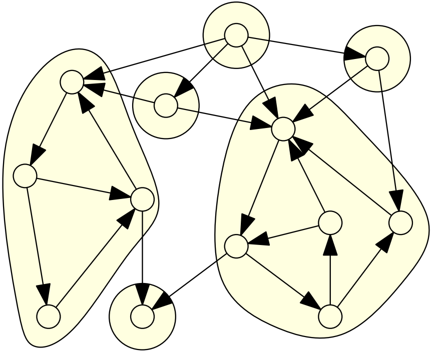
Def. A strongly connected component (SCC) of a directed graph G is a maximal strongly connected subgraph of G.



- directed graph $G = (V, E)$.
- it may happen: there is a $u \to v$ path, but no $v \to u$ path.

Def. A directed graph $G = (V, E)$ is strongly connected if for every $u, v \in V$, there is a path from u to v in G.

Def. A strongly connected component (SCC) of a directed graph G is a maximal strongly connected subgraph of G.

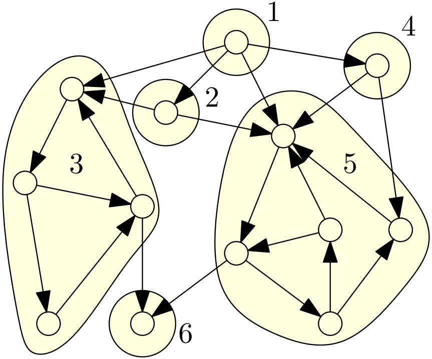


- Define equivalence relation: u and v are related if they are reachable from each other
- equivalence class $\equiv$ SCC

- directed graph $G = (V, E)$.
- it may happen: there is a $u \to v$ path, but no $v \to u$ path.

Def. A directed graph $G = (V, E)$ is strongly connected if for every $u, v \in V$, there is a path from u to v in G.

Def. A strongly connected component (SCC) of a directed graph G is a maximal strongly connected subgraph of G.



- Define equivalence relation: u and v are related if they are reachable from each other
- equivalence class $\equiv$ SCC
- After contracting each SCC, G becomes a directed-acyclic (multi-)graph (DAG).

Q: How can we check if a directed graph $G = (V, E)$ is strongly-connected?

Q: How can we check if a directed graph $G = (V, E)$ is strongly-connected?

A:

- Run a traversal algorithm (either BFS or DFS) from s twice, one on G, one on G with all directions of edges reversed
- If we reached all vertices in both algorithms, then G is strongly-connected
- Otherwise, it is not.

Q: How can we check if a directed graph $G = (V, E)$ is strongly-connected?

A:

- Run a traversal algorithm (either BFS or DFS) from s twice, one on G, one on G with all directions of edges reversed
- If we reached all vertices in both algorithms, then G is strongly-connected
- Otherwise, it is not.

Q: How can we find all strongly-connected components (SCCes) of a directed graph G?

Q: How can we check if a directed graph $G = (V, E)$ is strongly-connected?

A:

- Run a traversal algorithm (either BFS or DFS) from s twice, one on G, one on G with all directions of edges reversed
- If we reached all vertices in both algorithms, then G is strongly-connected
- Otherwise, it is not.

Q: How can we find all strongly-connected components (SCCes) of a directed graph G?

A: A much harder problem. Tarjan's $O(n+m)$-time algorithm.

Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

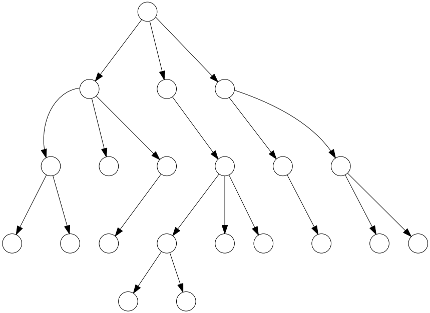
3. Topological Ordering

4. Bridges and 2-Edge-Connected Components
   - $O(n+m)$-Time Algorithm to Find Bridges
   - Related Concept: Cut Vertices

5. Strong Connectivity in Directed Graphs
   - Tarjan's $O(n+m)$-Time Algorithm for Finding SCCes

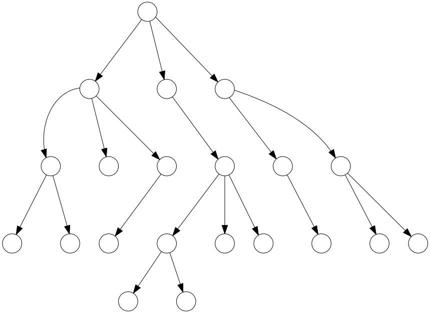
Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T



Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T

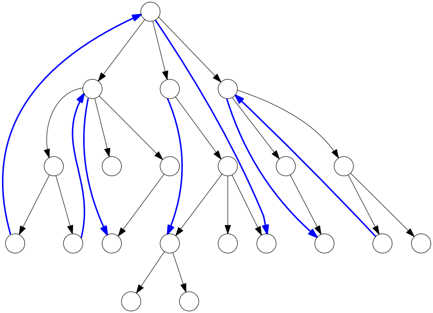


type of edges in G w.r.t T

- tree edges: edges in T

Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T

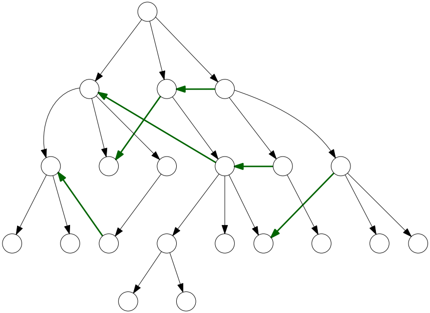


type of edges in G w.r.t T

- tree edges: edges in T
- upwards (vertical) edges
- downwards (vertical) edges

Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T

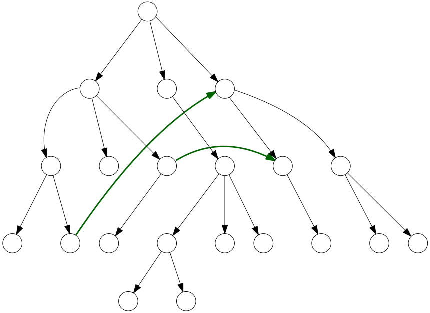


type of edges in G w.r.t T

- tree edges: edges in T
- upwards (vertical) edges
- downwards (vertical) edges
- leftwards horizontal edges

Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T



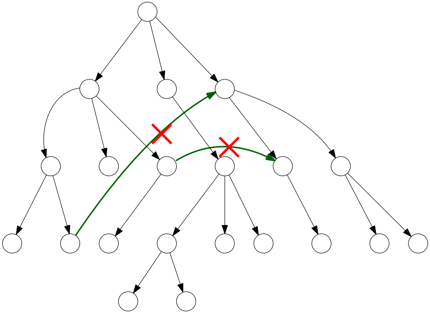
type of edges in G w.r.t T

- tree edges: edges in T
- upwards (vertical) edges
- downwards (vertical) edges
- leftwards horizontal edges

Q: Can there be rightwards horizontal edges?

Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T



type of edges in G w.r.t T

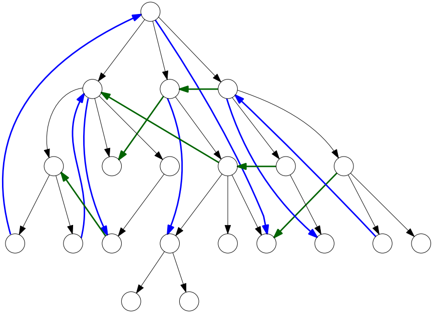
- tree edges: edges in T
- upwards (vertical) edges
- downwards (vertical) edges
- leftwards horizontal edges

Q: Can there be rightwards horizontal edges?

A: No!

Type of Edges w.r.t a Directed DFS Tree

- directed graph, $G = (V, E)$, a DFS-tree T,
- assuming every vertex is reachable from the root of T



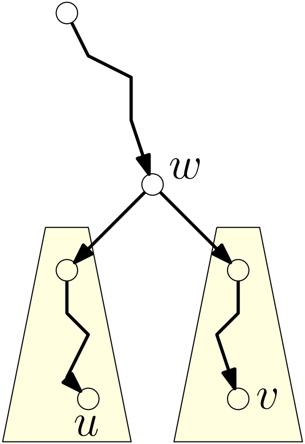
type of edges in G w.r.t T

- tree edges: edges in T
- upwards (vertical) edges
- downwards (vertical) edges
- leftwards horizontal edges

Q: Can there be rightwards horizontal edges?

A: No!

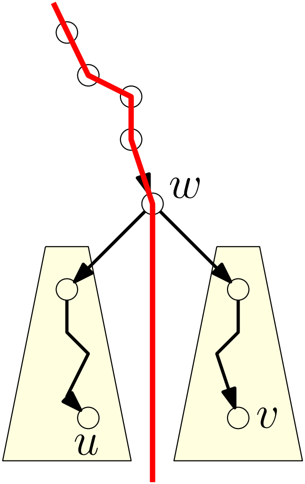
Lemma Suppose u and v are in the same SCC, and w is the lowest common ancestor (LCA) of u and v in T. Then w is the same SCC as u and v.



Lemma Suppose u and v are in the same SCC, and w is the lowest common ancestor (LCA) of u and v in T. Then w is the same SCC as u and v.

Proof.

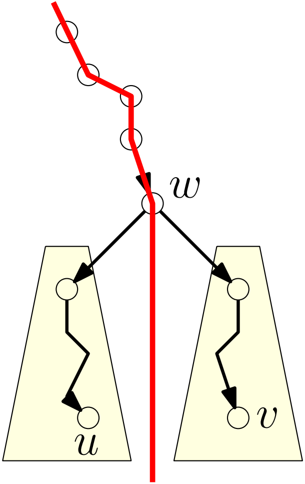
- Idea: using leftward, upwards and tree edges, u can not reach v without touching w or its ancestors. □



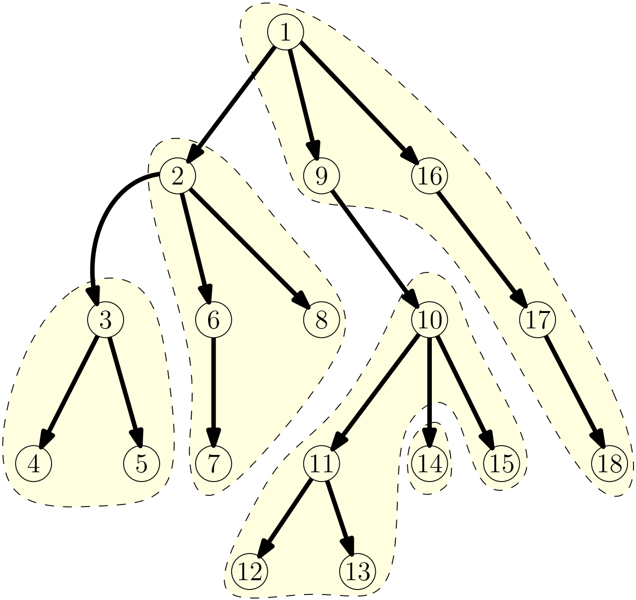
Lemma Suppose u and v are in the same SCC, and w is the lowest common ancestor (LCA) of u and v in T. Then w is the same SCC as u and v.

Proof.

- Idea: using leftward, upwards and tree edges, u can not reach v without touching w or its ancestors. □



Lemma The vertices in every SCC of G induce a sub-tree in T.


1
2
9
16
3
6
8
10
17
4
5
7
11
14
15
18
12
13

An Intermediate Algorithm to Keep in Mind

```
build the DFS tree $T$
while $T$ is not empty do
    find the first vertex $v$ in the posterior-order-traversal of $T$ satisfying the following property: there are no edges from $T_v$ to outside $T_v$
    claim vertices in $T_v$ as a SCC, remove them from $T$ and all edges incident to them from $T$ and $G$
```

Illustration of Intermediate Algorithm

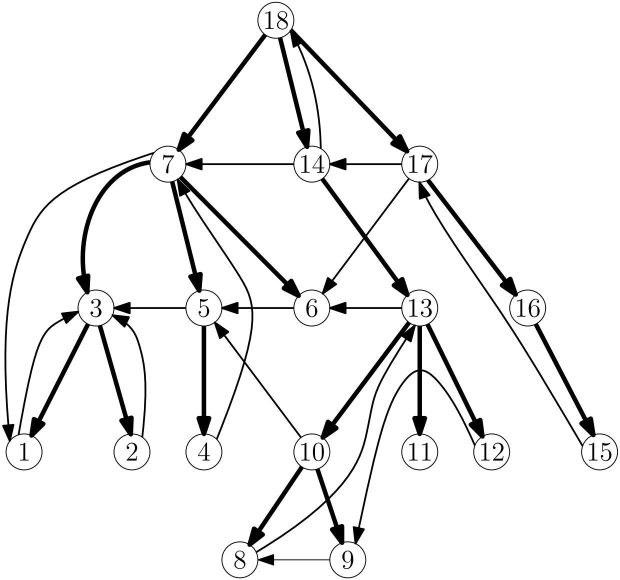


Illustration of Intermediate Algorithm

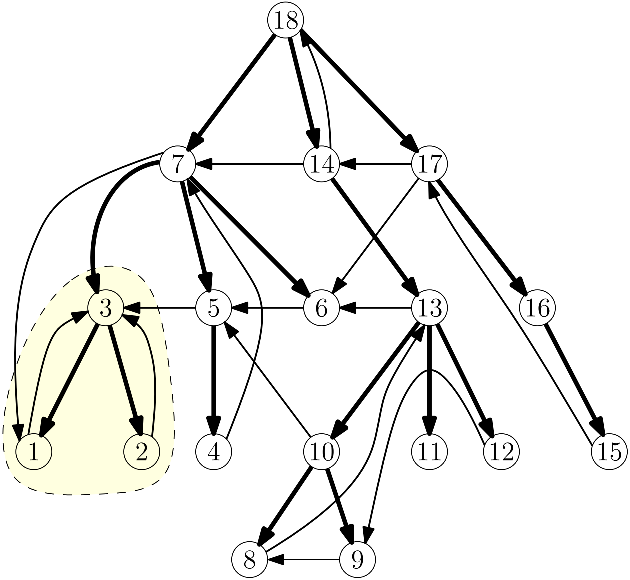


Illustration of Intermediate Algorithm

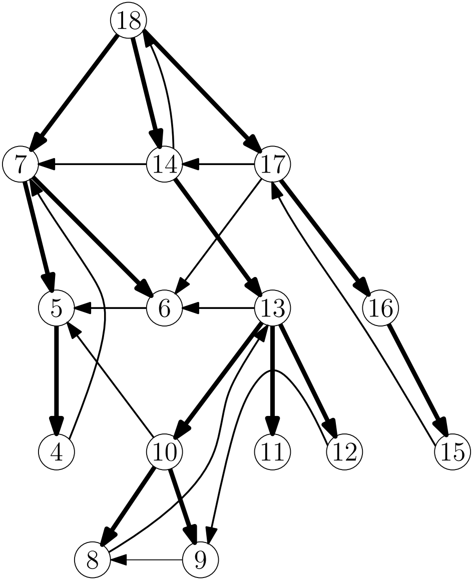


Illustration of Intermediate Algorithm

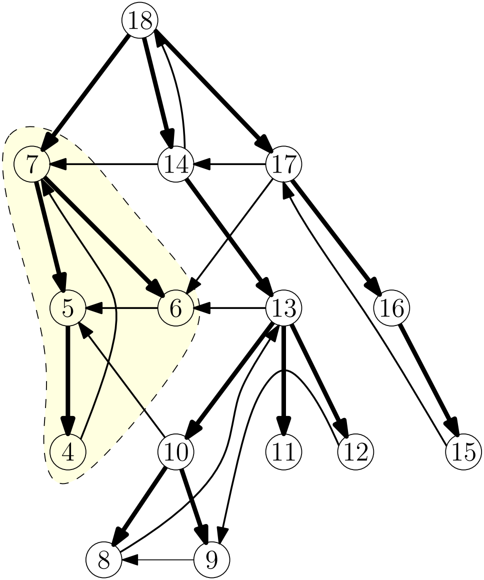


Illustration of Intermediate Algorithm

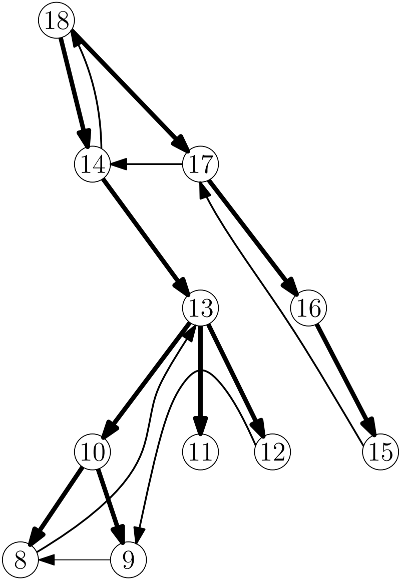


Illustration of Intermediate Algorithm

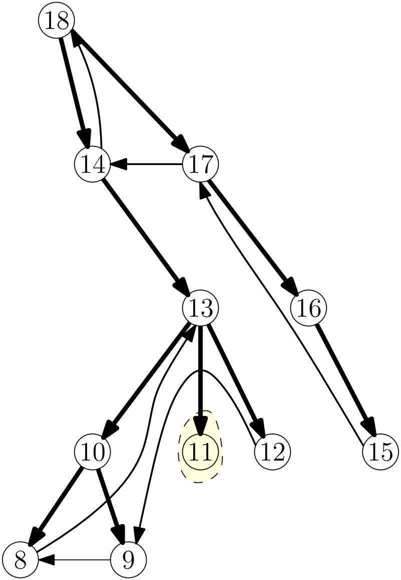


Illustration of Intermediate Algorithm

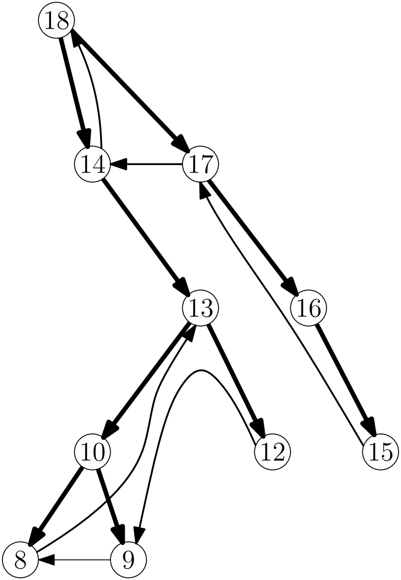


Illustration of Intermediate Algorithm

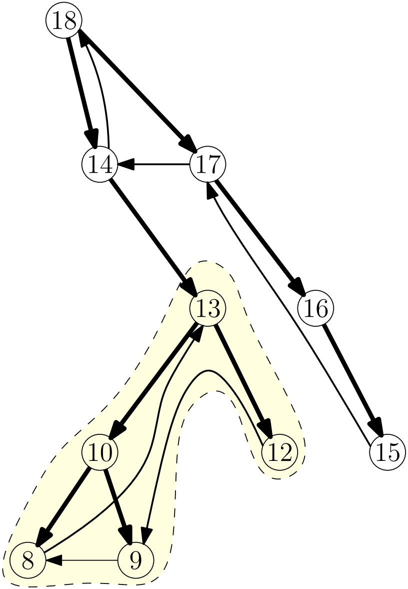


Illustration of Intermediate Algorithm

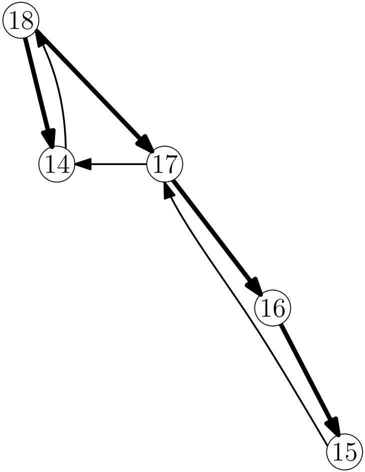


Illustration of Intermediate Algorithm

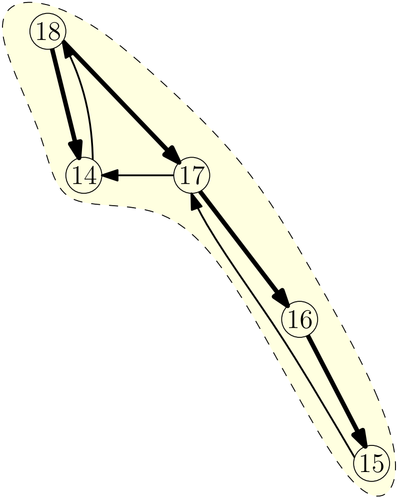


Illustration of Intermediate Algorithm

Illustration of Tarjan's Algorithm

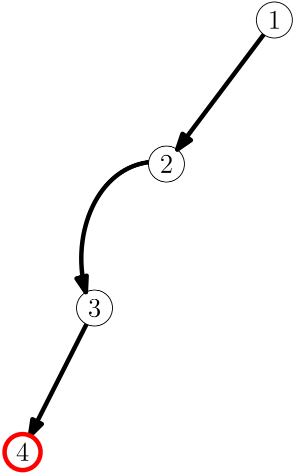


Illustration of Tarjan's Algorithm

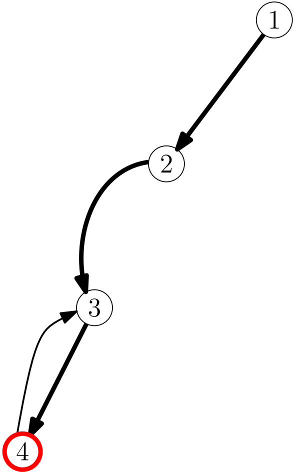


Illustration of Tarjan's Algorithm

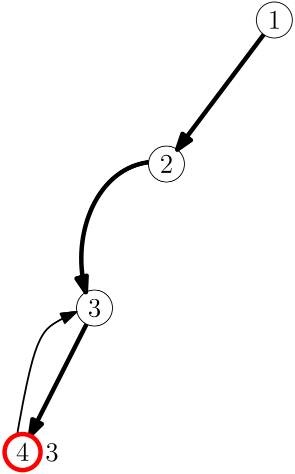


Illustration of Tarjan's Algorithm

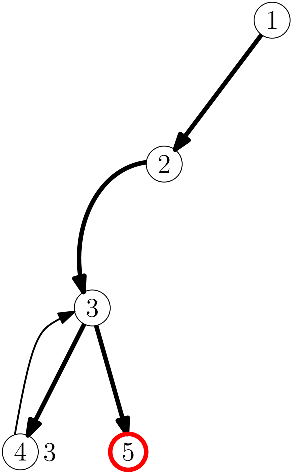


Illustration of Tarjan's Algorithm

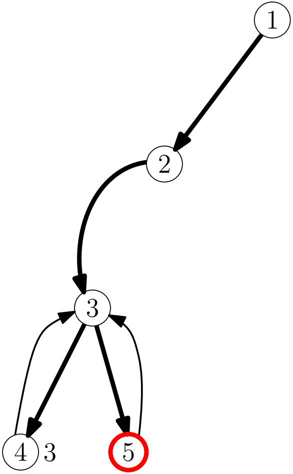


Illustration of Tarjan's Algorithm

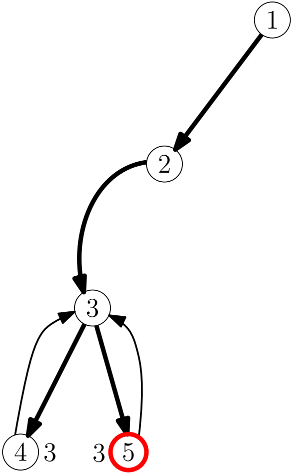


Illustration of Tarjan's Algorithm

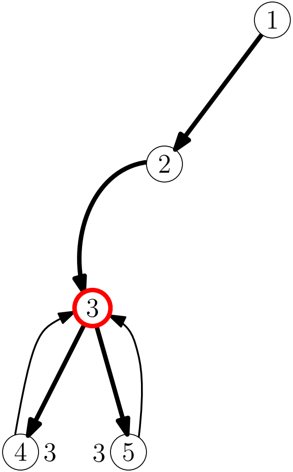


Illustration of Tarjan's Algorithm

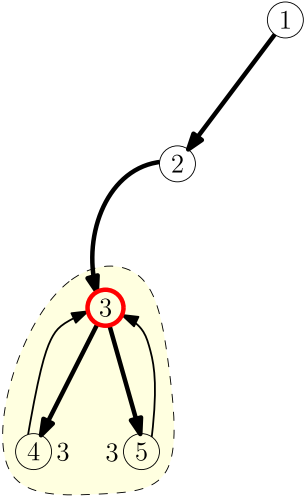


Illustration of Tarjan's Algorithm

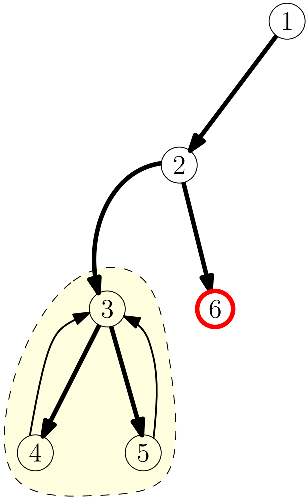


Illustration of Tarjan's Algorithm

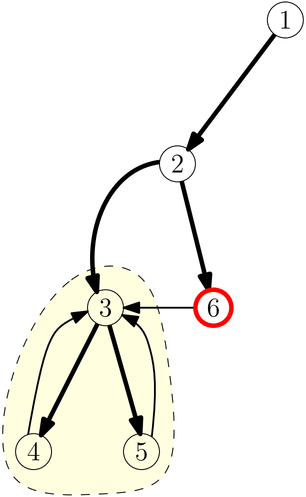


Illustration of Tarjan's Algorithm

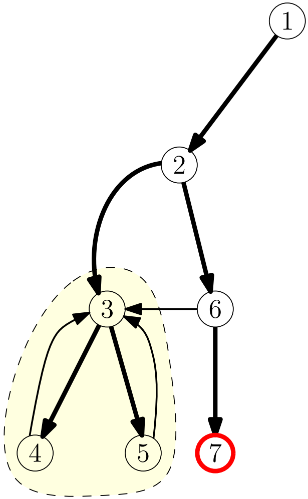


Illustration of Tarjan's Algorithm

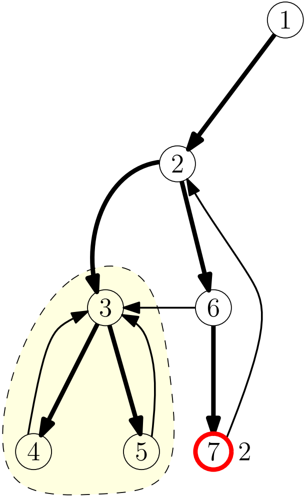


Illustration of Tarjan's Algorithm

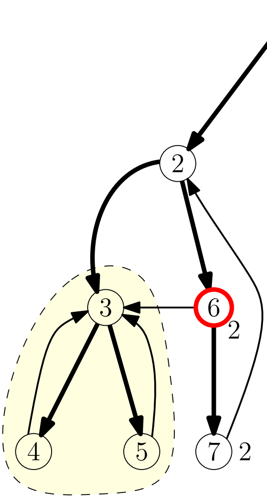


Illustration of Tarjan's Algorithm

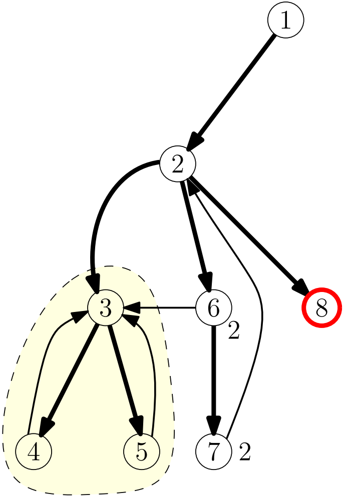


Illustration of Tarjan's Algorithm

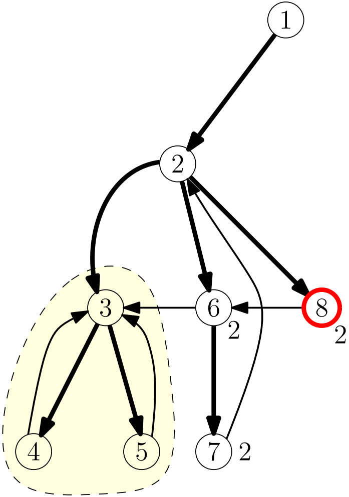


Illustration of Tarjan's Algorithm

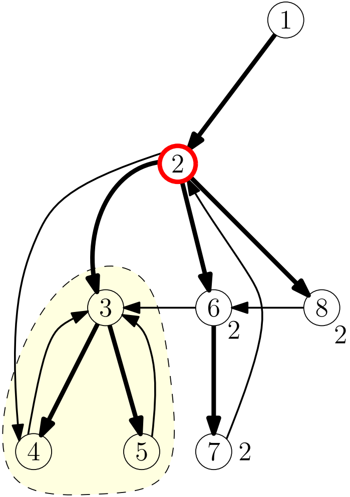


Illustration of Tarjan's Algorithm

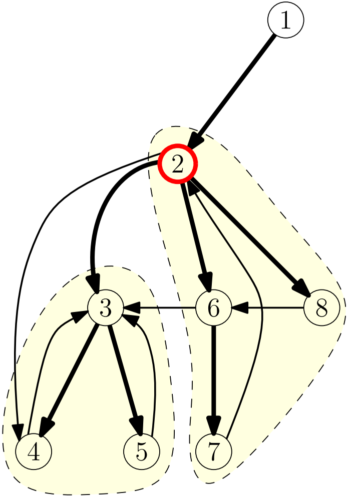


Illustration of Tarjan's Algorithm

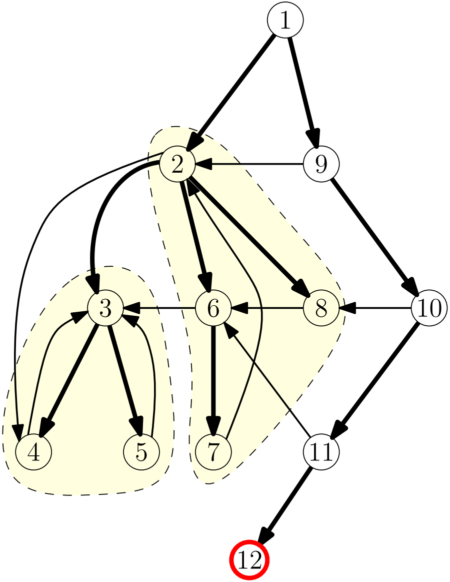


Illustration of Tarjan's Algorithm

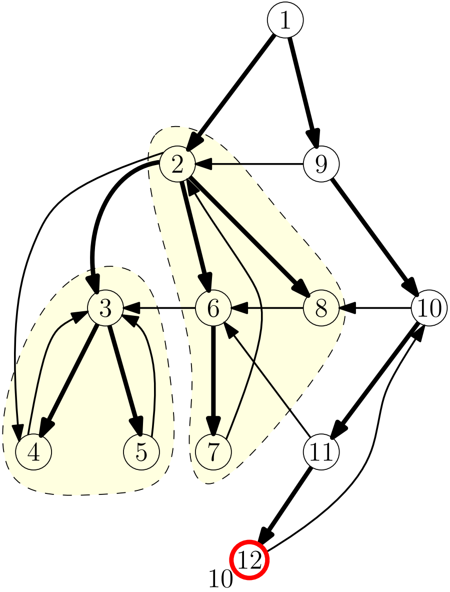


Illustration of Tarjan's Algorithm

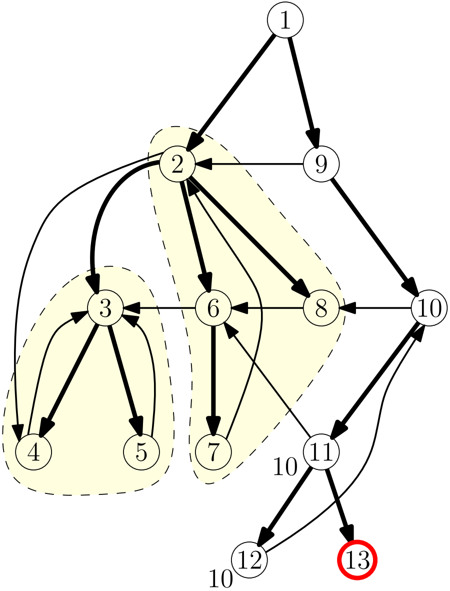


Illustration of Tarjan's Algorithm

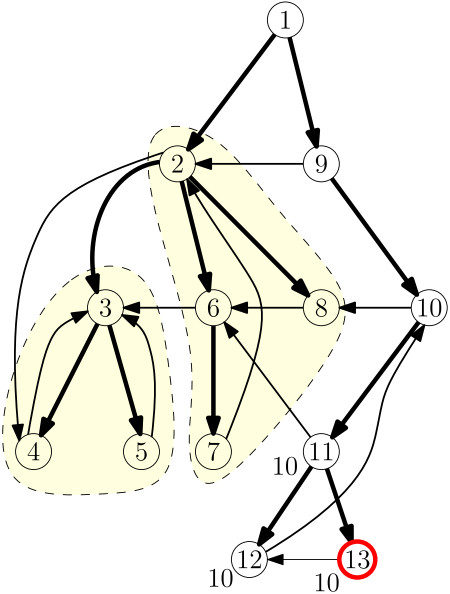


Illustration of Tarjan's Algorithm

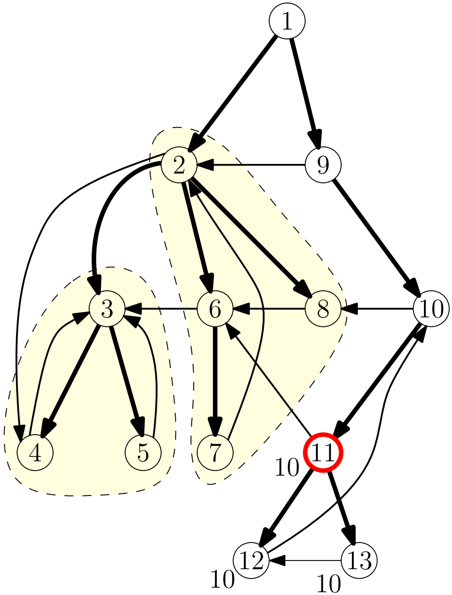


Illustration of Tarjan's Algorithm

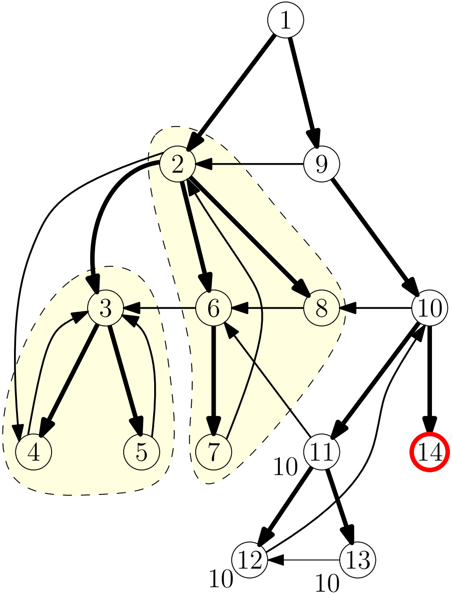


Illustration of Tarjan's Algorithm

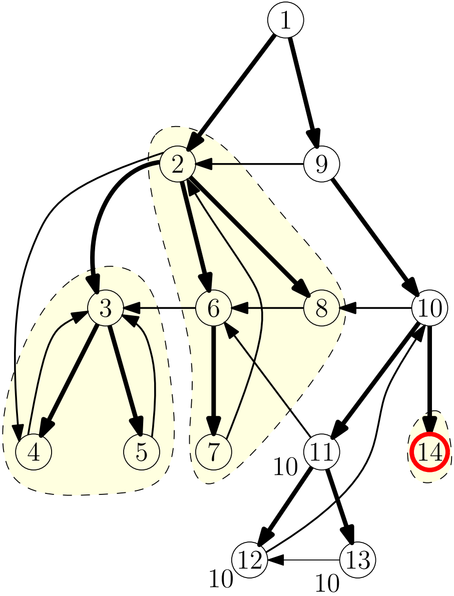


Illustration of Tarjan's Algorithm

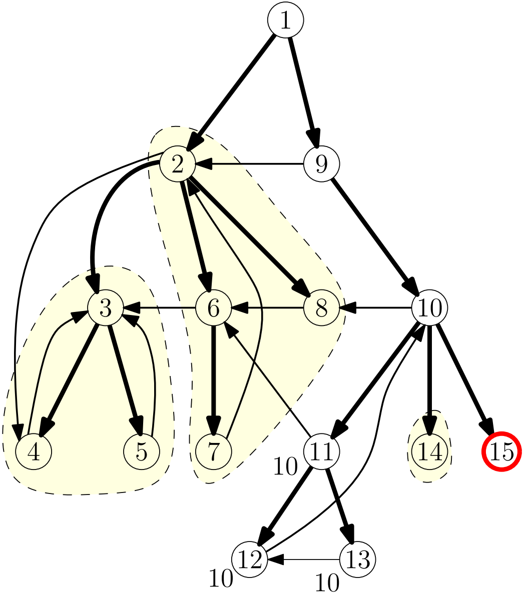


Illustration of Tarjan's Algorithm

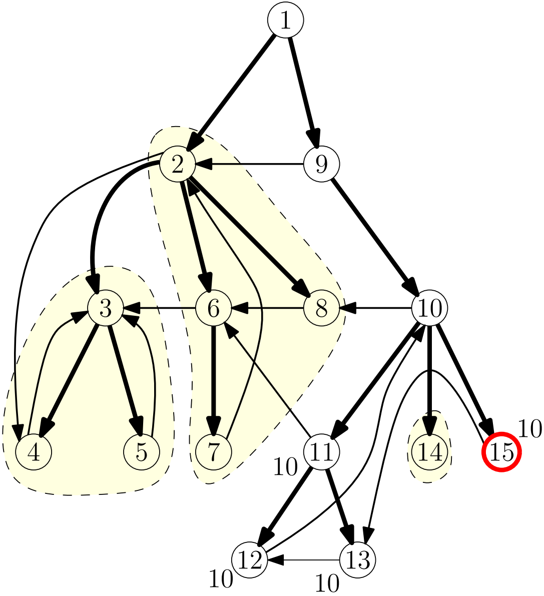


Illustration of Tarjan's Algorithm

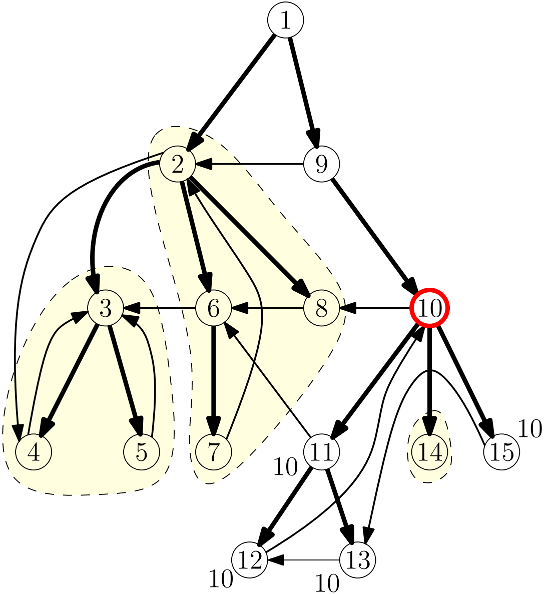


Illustration of Tarjan's Algorithm

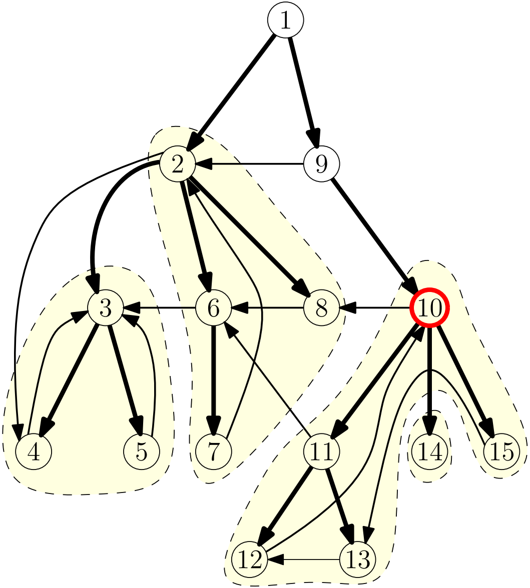


Illustration of Tarjan's Algorithm

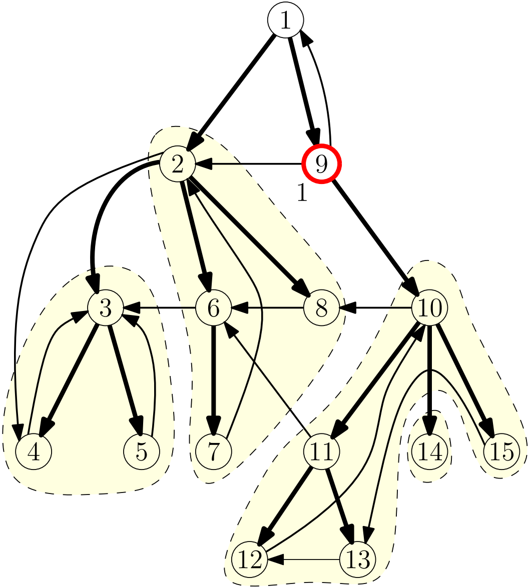


Illustration of Tarjan's Algorithm

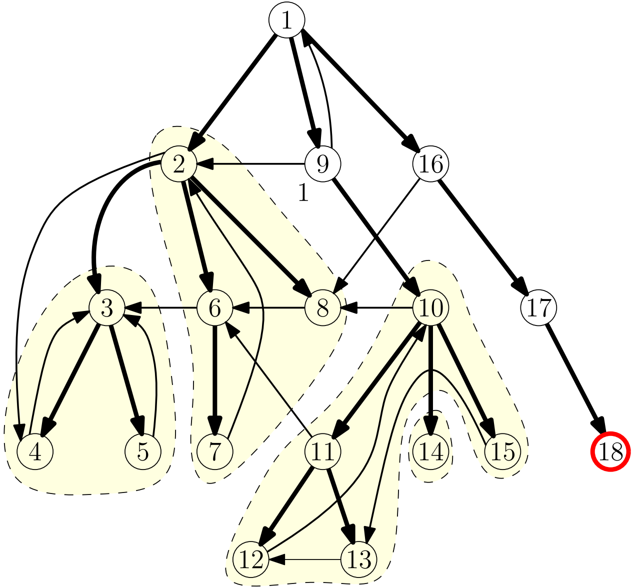


Illustration of Tarjan's Algorithm

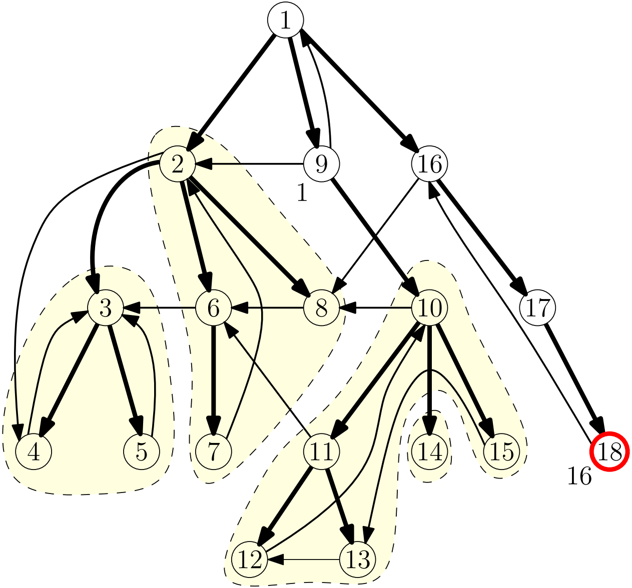


Illustration of Tarjan's Algorithm

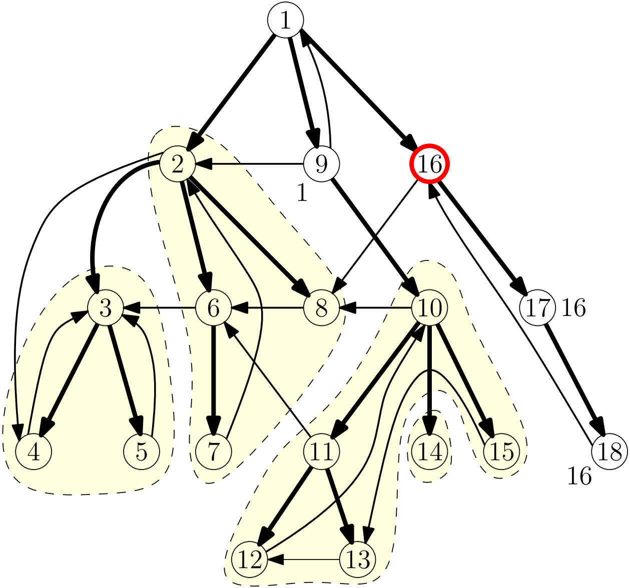


Illustration of Tarjan's Algorithm

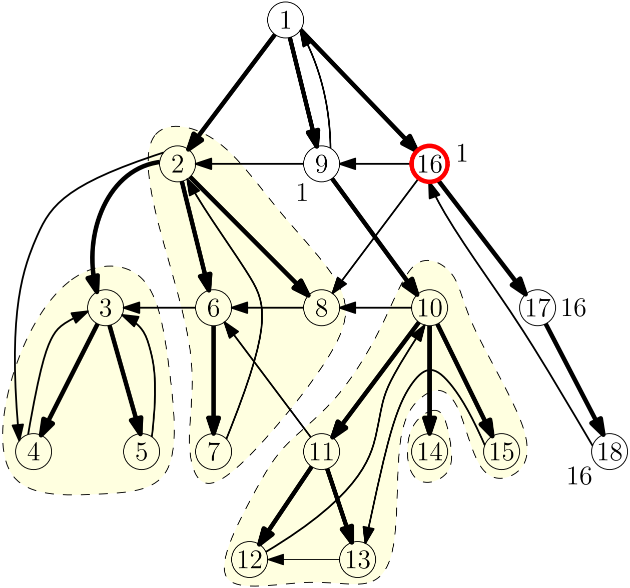


Illustration of Tarjan's Algorithm

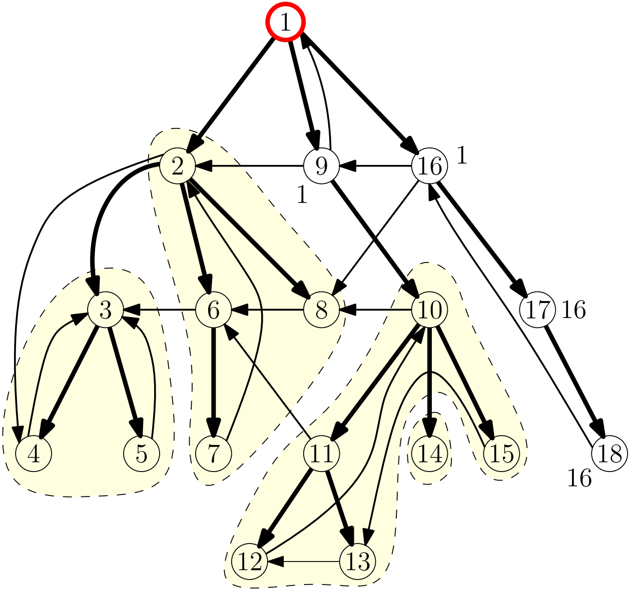
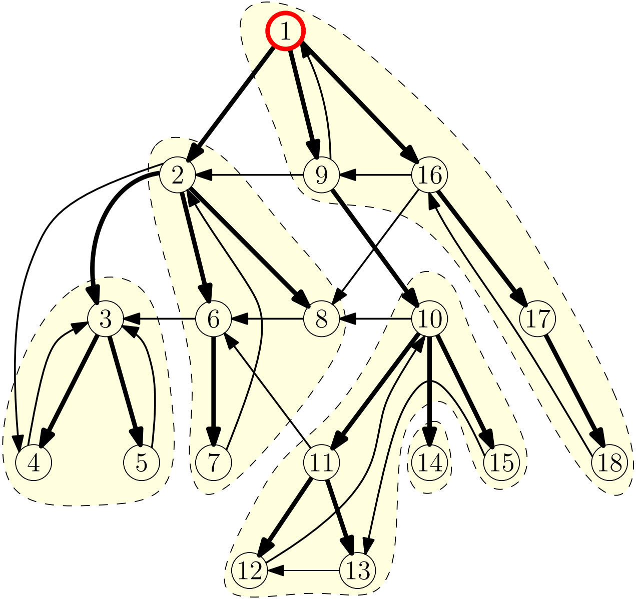


Illustration of Tarjan's Algorithm



finding strongly connected components

```
$statck \leftarrow$ empty stack, $i \leftarrow 0$
**for** every $v \in V$ **do**: $v.i \leftarrow \perp, onstack[i] \leftarrow$ **false**
**for** every $v \in V$ **do**
    **if** $v.i = \perp$ **then** recursive-DFS$(v)$
```

recursive-DFS(v)

```
$i \leftarrow i + 1, v.i \leftarrow i, v.r \leftarrow i$
$stack.push(v), onstack[v] \leftarrow$ **true**
**for** every outgoing edge $(v, u)$ of $v$ **do**
    **if** $u.i = \perp$ **then** recursive-DFS$(u)$
    **if** $onstack[u]$ and $u.r < v.r$ **then** $v.r \leftarrow u.r$
**if** $v.r = v.i$ **then**
    pop all vertices in $stack$ after $v$, including $v$ itself
    set $onstack$ of these vertices to be **false**
    declare that these vertices form an SCC
```

Running time of the algorithm is $O(n+m)$.