CSE 431/531: Algorithm Analysis and Design (Fall 2021)
Graph Basics

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Outline

Examples of Graphs

Figure: Road Networks

Figure: Internet

Figure: Social Networks

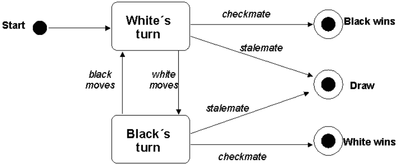
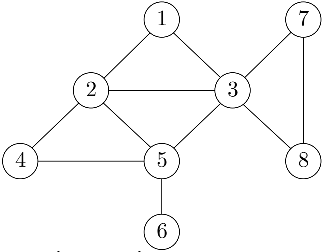


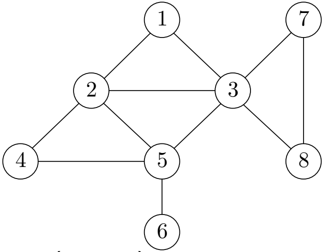
Figure: Transition Graphs

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



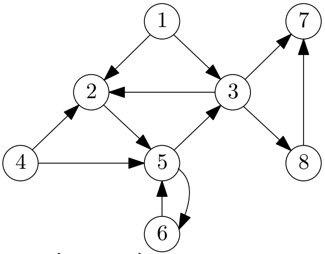
- V: set of vertices (nodes);
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2

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



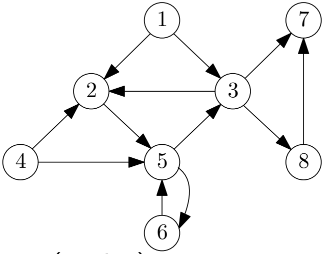
- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2
  - $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7,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

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

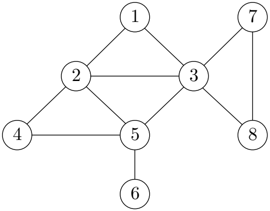
Directed Graph $G = (V, E)$



- 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

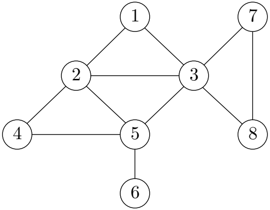
- For (undirected) graphs, we often use (i, j) to denote the set $\{i, j\}$.
- We call (i, j) an unordered pair; in this case $(i, j) = (j, i)$.



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

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

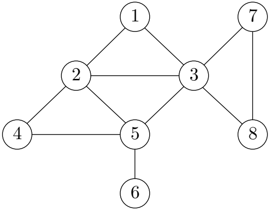
Representation of Graphs

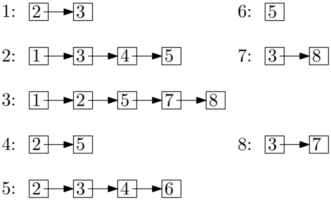


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

- 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





- 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.

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		
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	$O(n^2)$	
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 $n-1 \leq m \leq n(n-1)/2$
- 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		

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	$O(n)$	

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

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

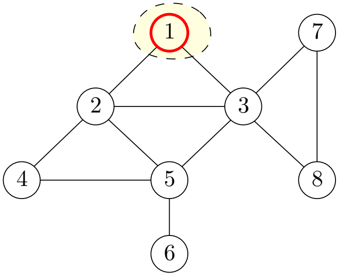
- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

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

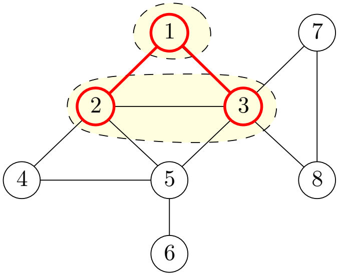
Breadth-First Search (BFS)

- 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



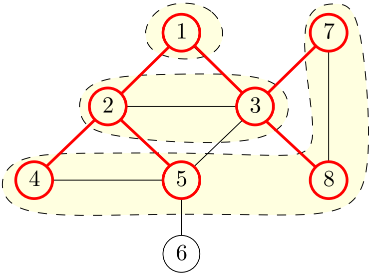
Breadth-First Search (BFS)

- 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



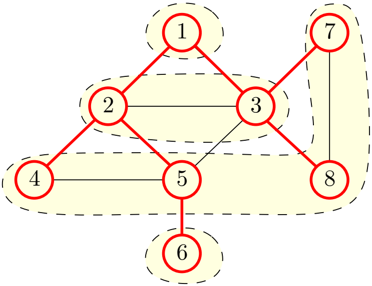
Breadth-First Search (BFS)

- 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



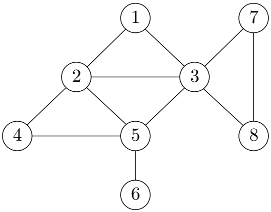
Implementing BFS using a Queue

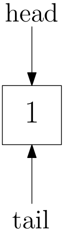
BFS(s)

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

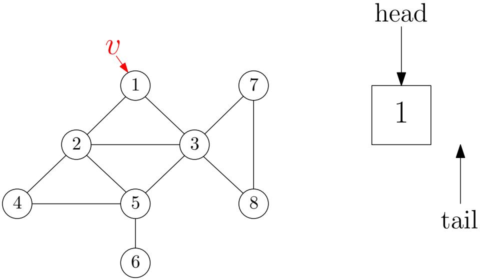
- Running time: $O(n + m)$.

Example of BFS via Queue

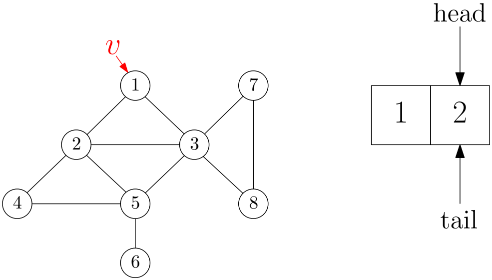




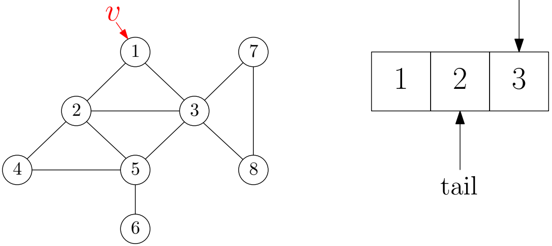
Example of BFS via Queue



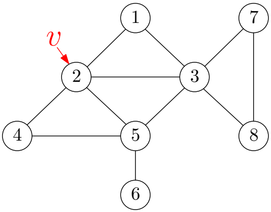
Example of BFS via Queue

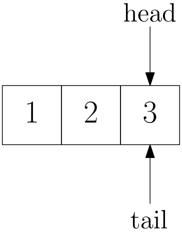


Example of BFS via Queue

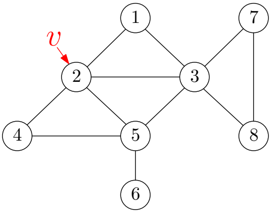


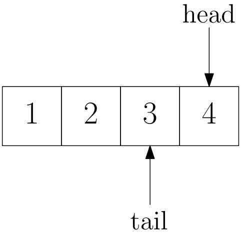
Example of BFS via Queue



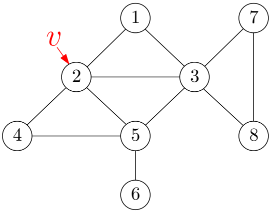


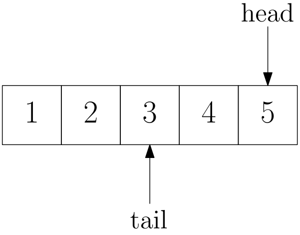
Example of BFS via Queue



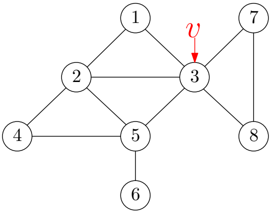


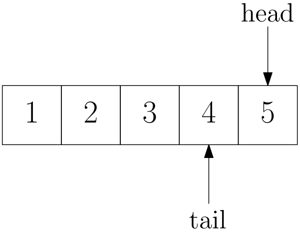
Example of BFS via Queue



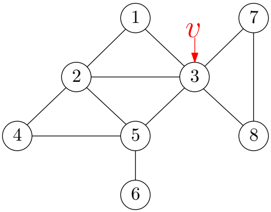


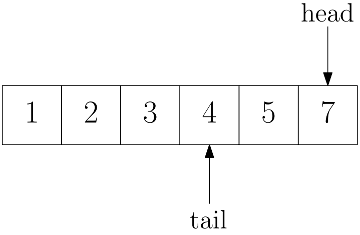
Example of BFS via Queue



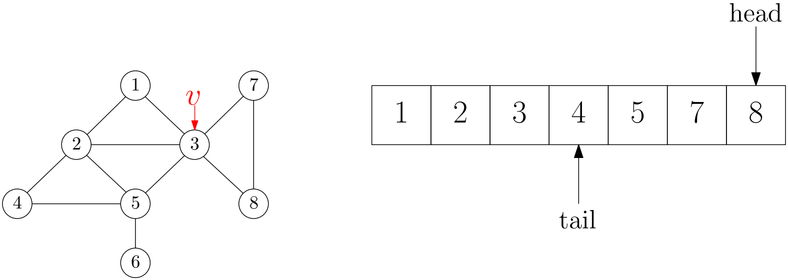


Example of BFS via Queue

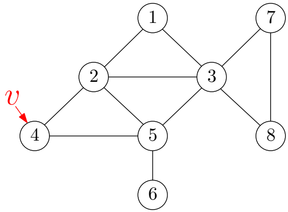


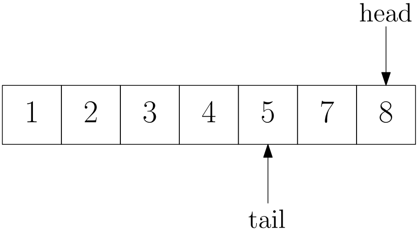


Example of BFS via Queue

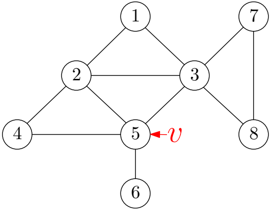


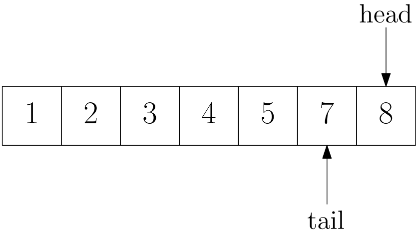
Example of BFS via Queue



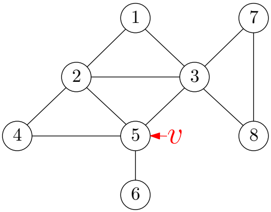


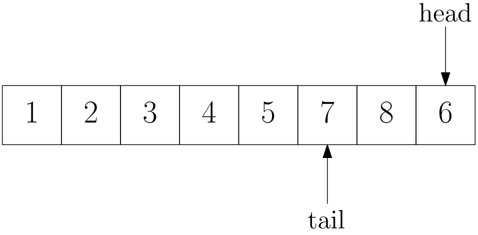
Example of BFS via Queue



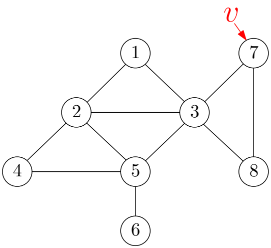


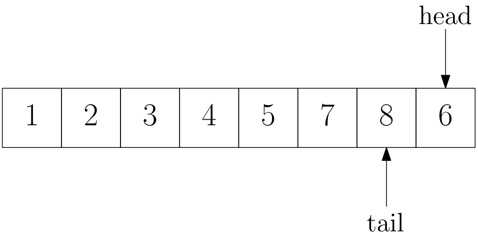
Example of BFS via Queue



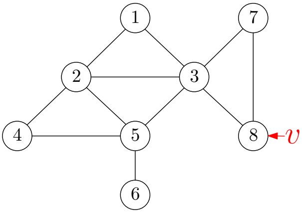


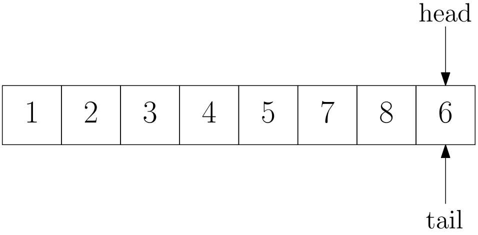
Example of BFS via Queue



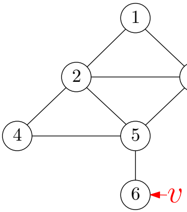


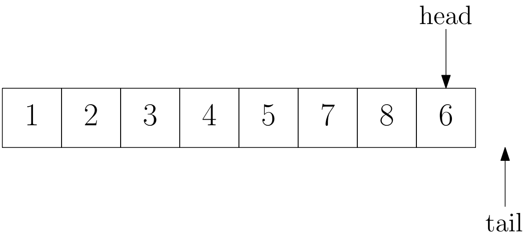
Example of BFS via Queue





Example of BFS via Queue



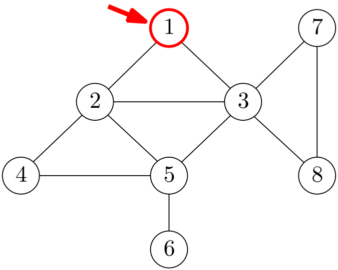


Depth-First Search (DFS)

- 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

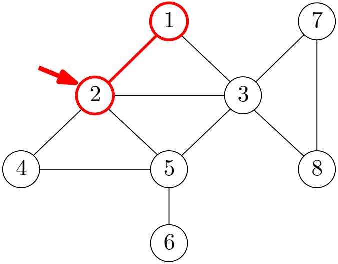
Depth-First Search (DFS)

- 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



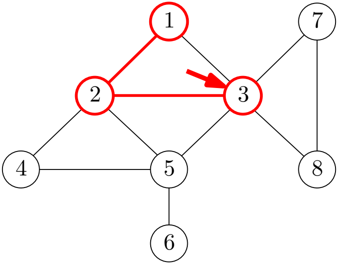
Depth-First Search (DFS)

- 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



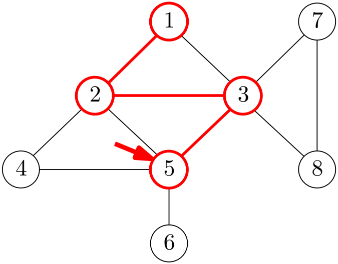
Depth-First Search (DFS)

- 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



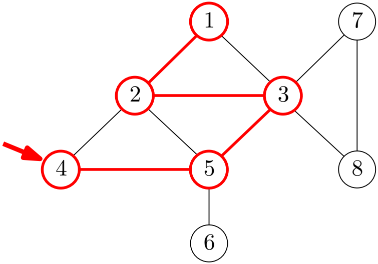
Depth-First Search (DFS)

- 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



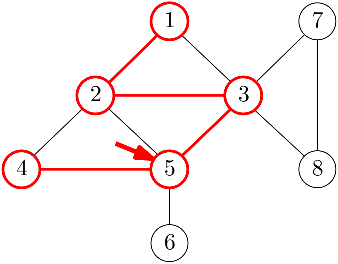
Depth-First Search (DFS)

- 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



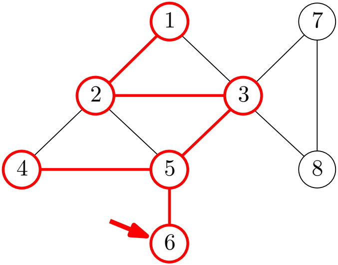
Depth-First Search (DFS)

- 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



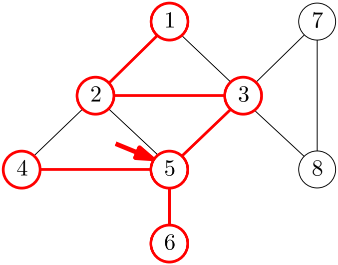
Depth-First Search (DFS)

- 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



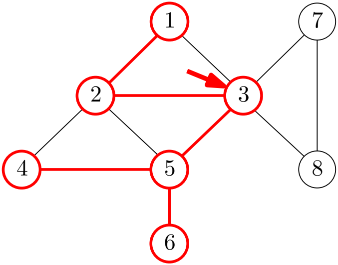
Depth-First Search (DFS)

- 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



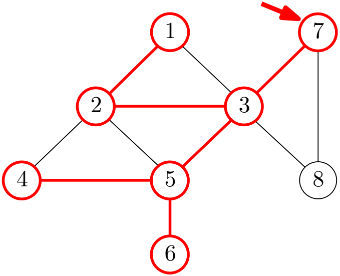
Depth-First Search (DFS)

- 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



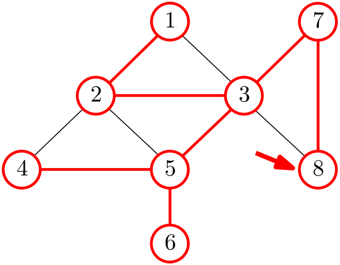
Depth-First Search (DFS)

- 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)
```

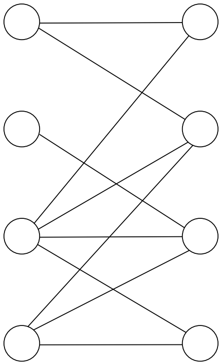
recursive-DFS(v)

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

Outline

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
- $\cdots$

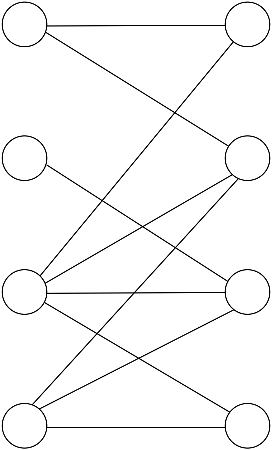
Testing Bipartiteness

- 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

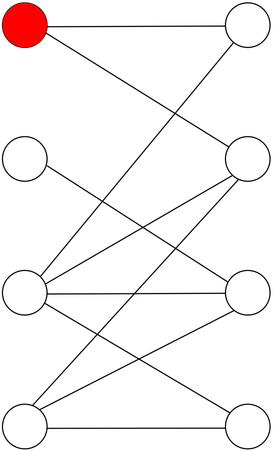
Testing Bipartiteness

- 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

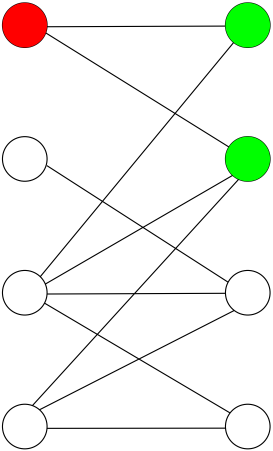
Test Bipartiteness



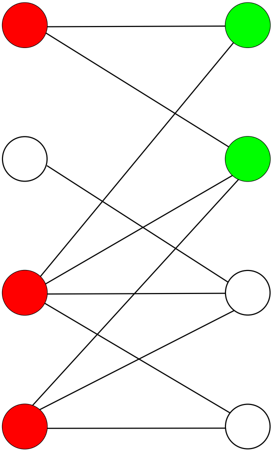
Test Bipartiteness



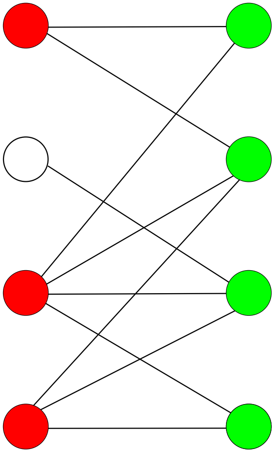
Test Bipartiteness



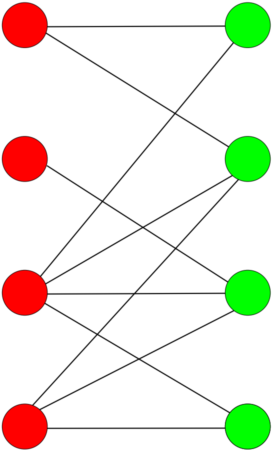
Test Bipartiteness



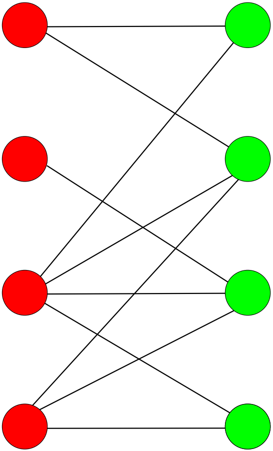
Test Bipartiteness

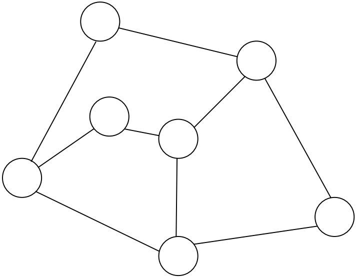


Test Bipartiteness

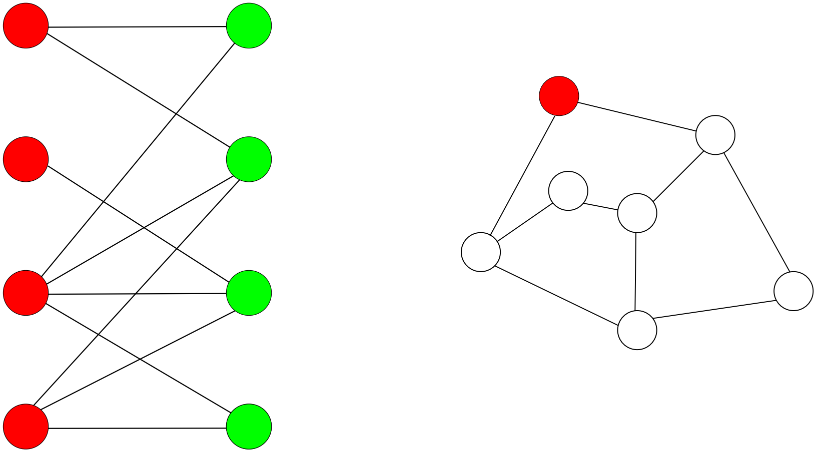


Test Bipartiteness

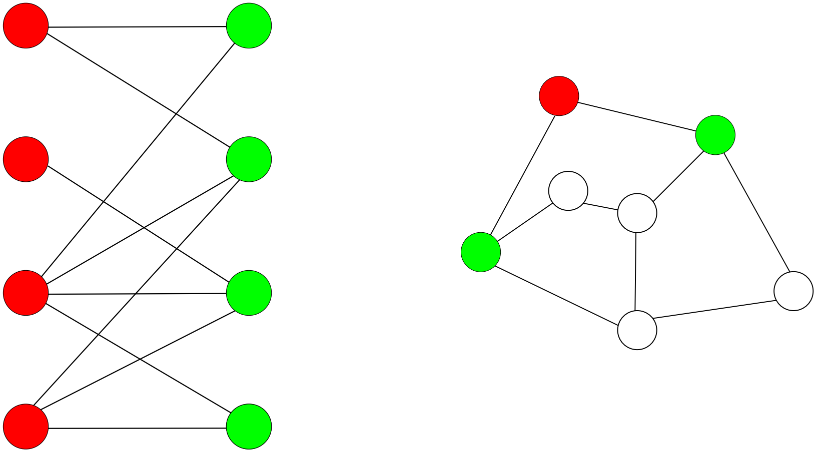




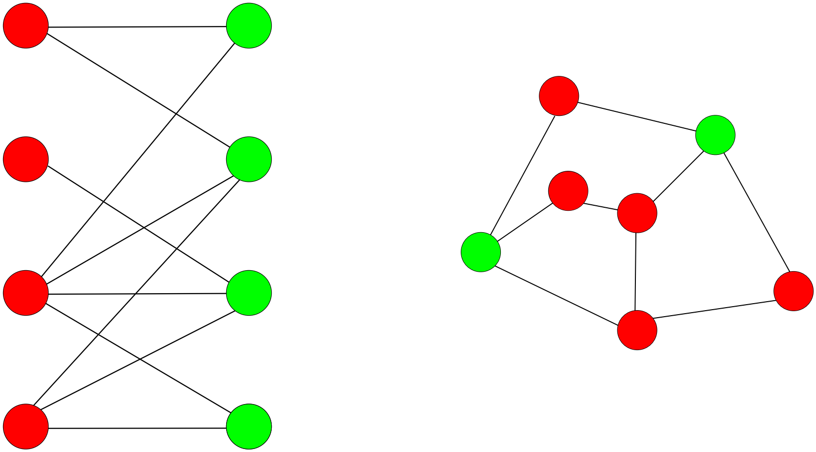
Test Bipartiteness



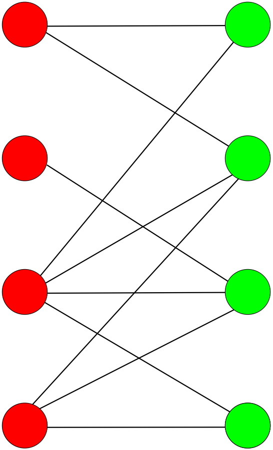
Test Bipartiteness

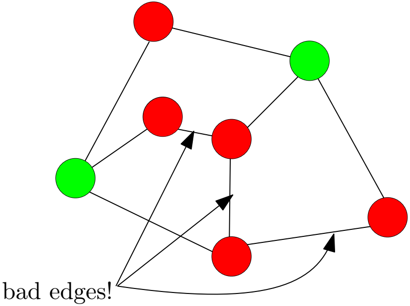


Test Bipartiteness



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 $\geq$ tail **do**
    $v \leftarrow queue[tail], tail \leftarrow tail + 1$
    for all neighbours u of v **do**
        if u is "unvisited" **then**
            $head \leftarrow head + 1, queue[head] = u$
            mark u as "visited"

Testing Bipartiteness using BFS

test-bipartiteness(s)

```
head ← 1, tail ← 1, queue[1] ← s
mark s as "visited" and all other vertices as "unvisited"
color[s] ← 0
while head ≥ tail do
    v ← queue[tail], tail ← tail + 1
    for all neighbours u of v do
        if u is "unvisited" then
            head ← head + 1, queue[head] = u
            mark u as "visited"
            color[u] ← 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 $= O(n+m)$

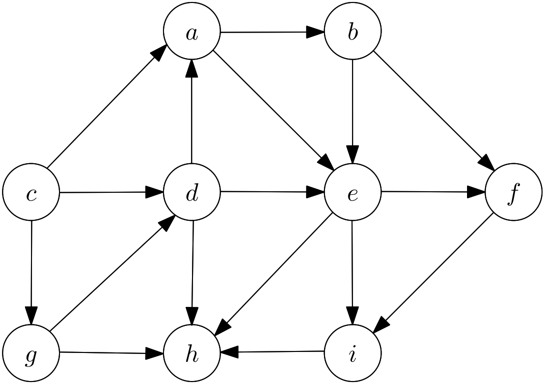
Outline

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 $\pi(u) < \pi(v)$

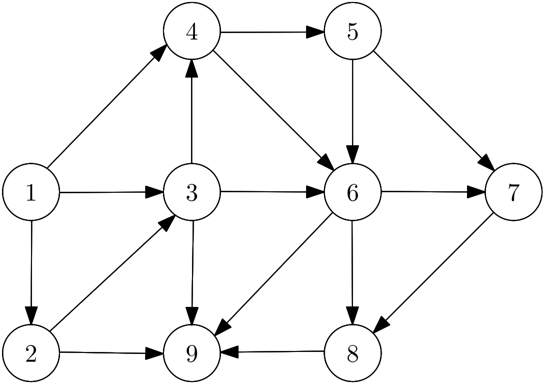


Topological Ordering Problem

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

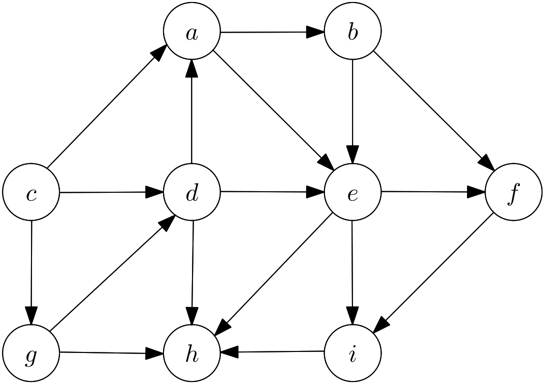
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

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



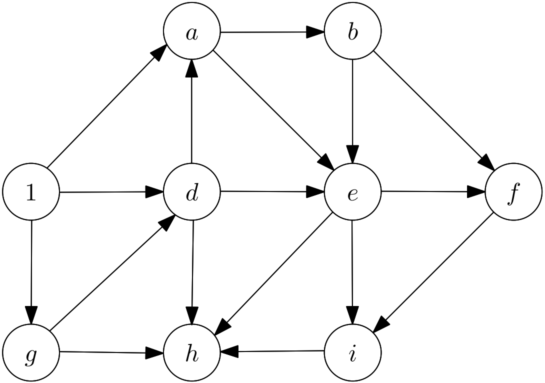
Topological Ordering

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



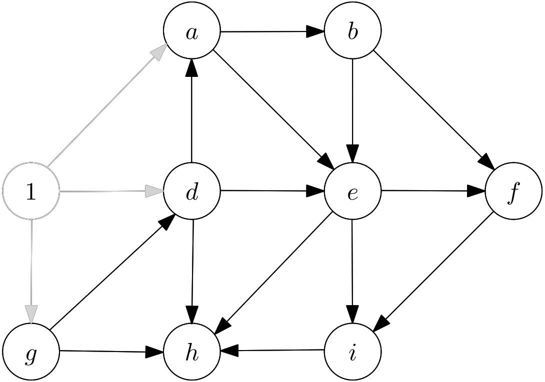
Topological Ordering

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



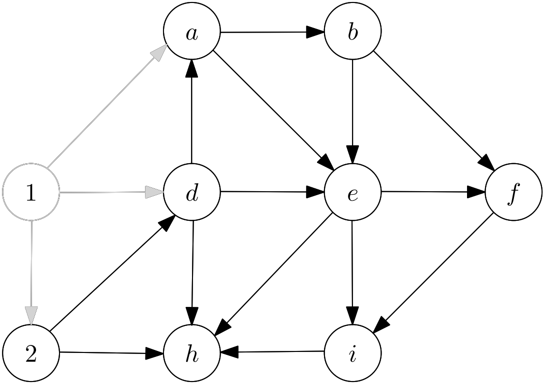
Topological Ordering

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



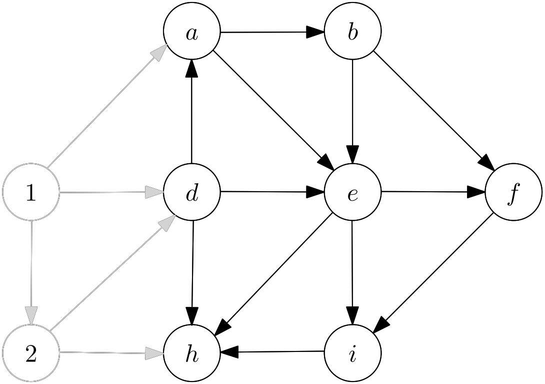
Topological Ordering

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



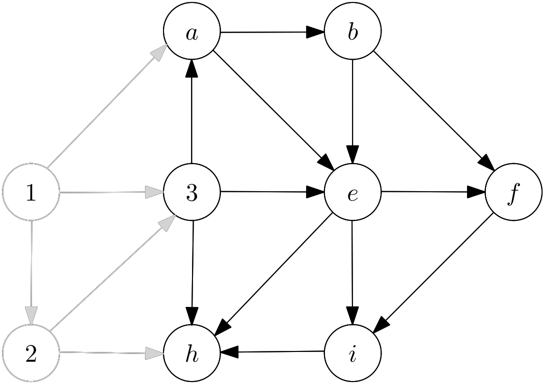
Topological Ordering

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



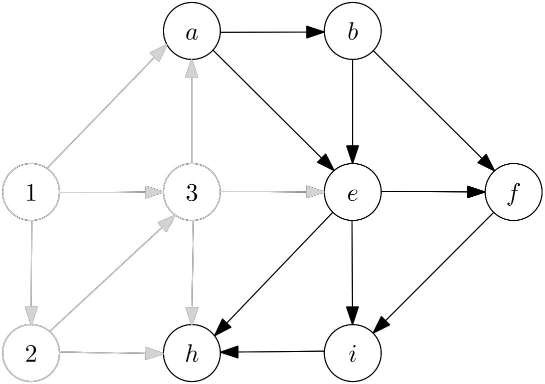
Topological Ordering

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



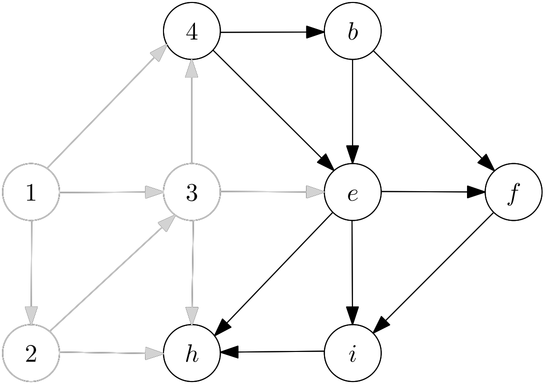
Topological Ordering

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



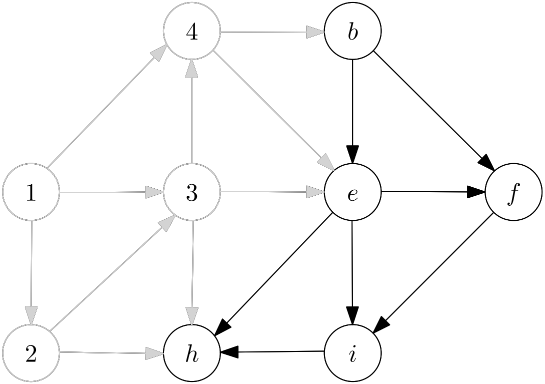
Topological Ordering

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



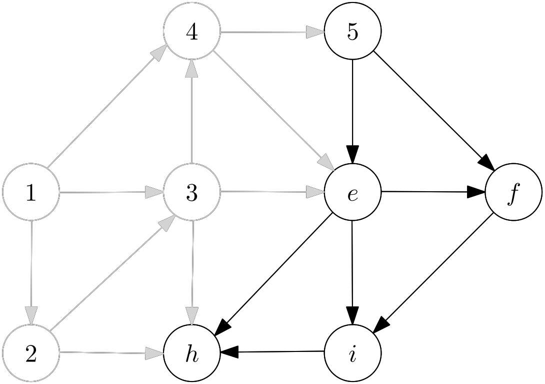
Topological Ordering

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



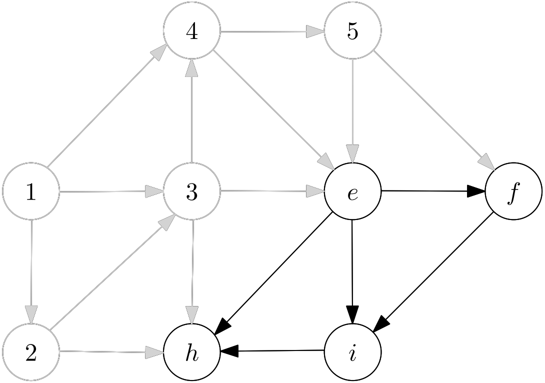
Topological Ordering

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



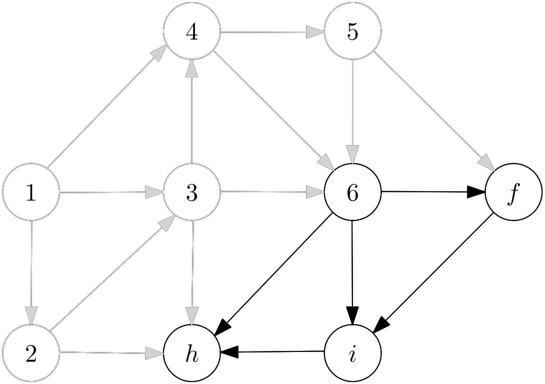
Topological Ordering

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



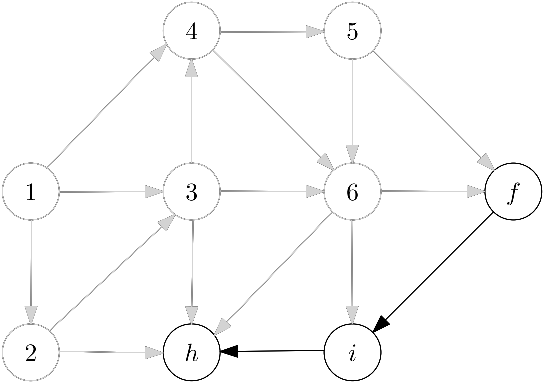
Topological Ordering

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



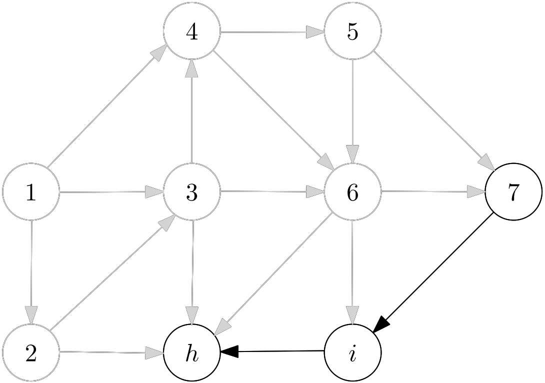
Topological Ordering

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



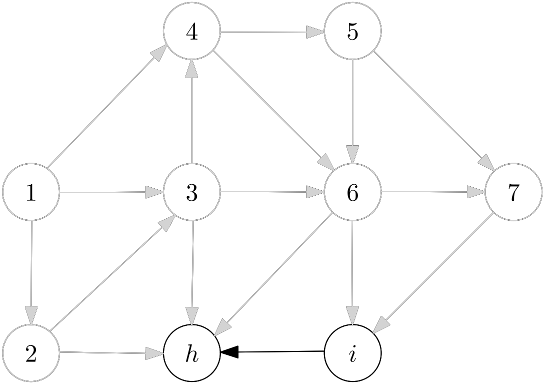
Topological Ordering

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



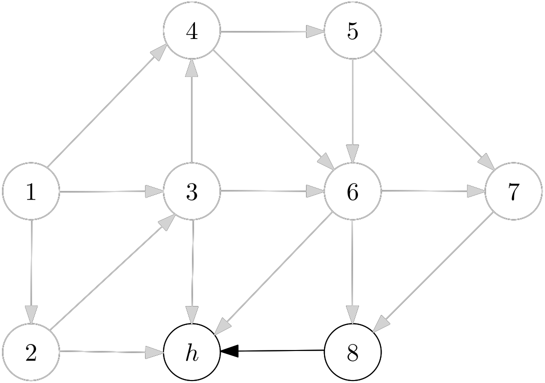
Topological Ordering

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



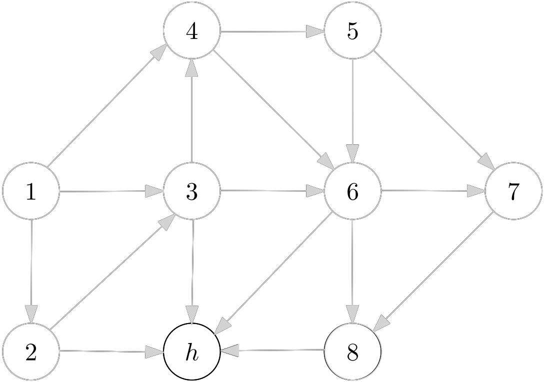
Topological Ordering

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



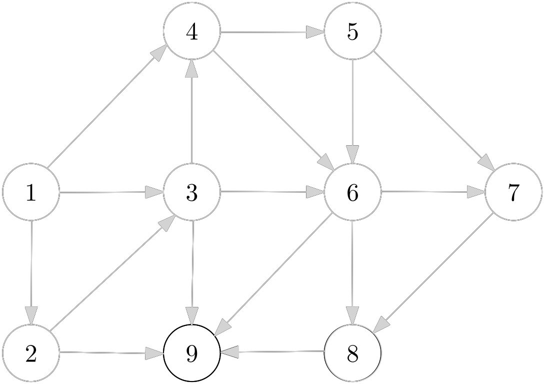
Topological Ordering

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



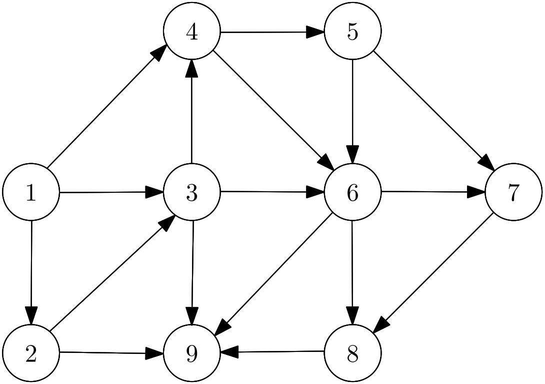
Topological Ordering

- 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)

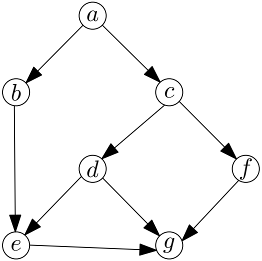
let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$ **do**
    for every u such that $(v, u) \in E$ **do**
        $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$ **do**
    $v \leftarrow$ arbitrary vertex in S, $S \leftarrow S \setminus \{v\}$
    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
    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"

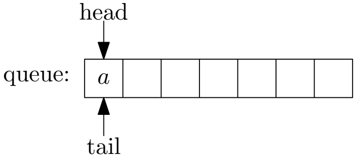
- S can be represented using a queue or a stack
- Running time $= O(n + m)$

S as a Queue or a Stack

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

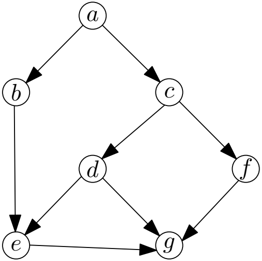
Example

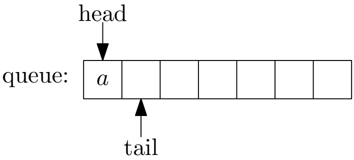




	a	b	c	d	e	f	g
degree	0	1	1	1	2	1	3

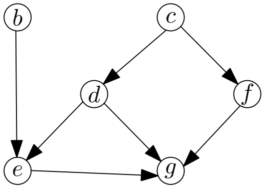
Example





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

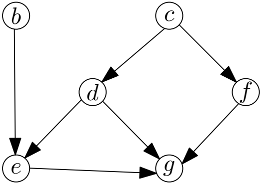
Example

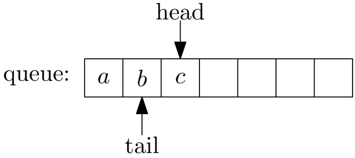


queue: a (head → a; tail → second cell)

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

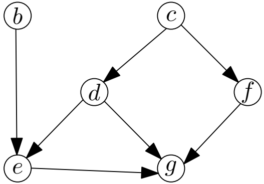
Example

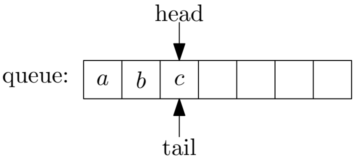




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

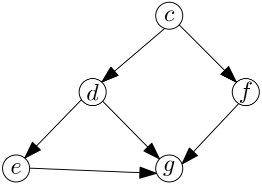
Example

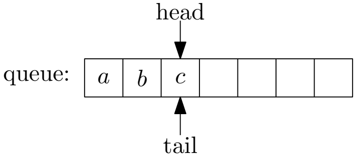




	a	b	c	d	e	f	g
degree	0	0	0	1	2	1	3

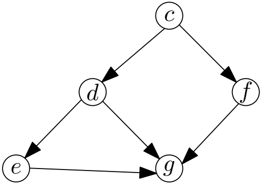
Example

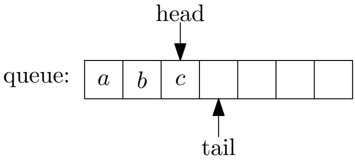




	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

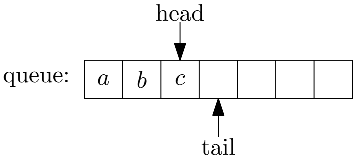
Example

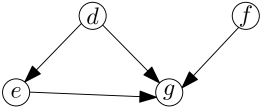




	a	b	c	d	e	f	g
degree	0	0	0	1	1	1	3

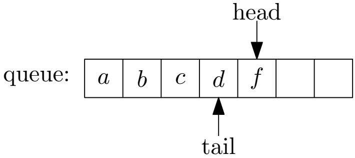
Example

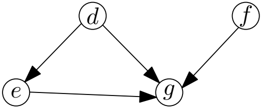




	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

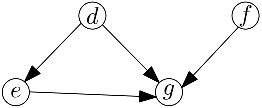
Example

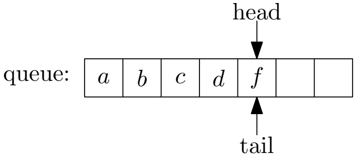




	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

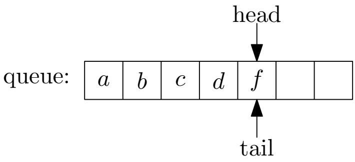
Example

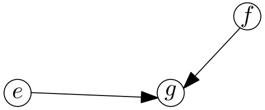




	a	b	c	d	e	f	g
degree	0	0	0	0	1	0	3

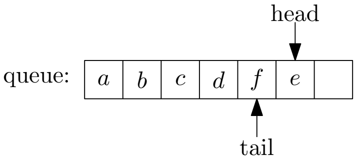
Example

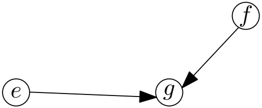




	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	2

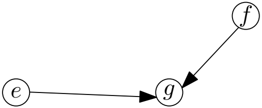
Example





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

Example



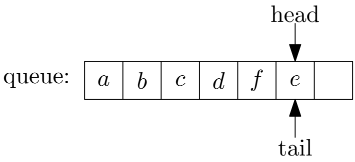
queue:

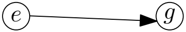
a	b	c	d	f	e	

head → e; tail → e

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

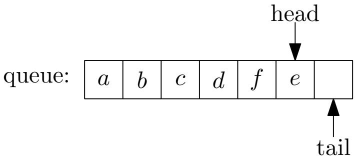
Example

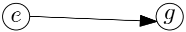




	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

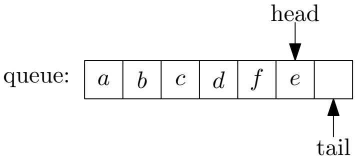
Example





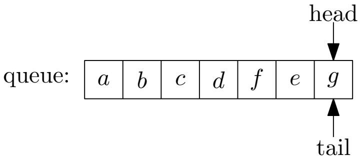
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	1

Example



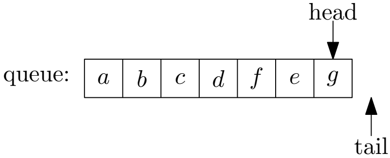
	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Example



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

Example

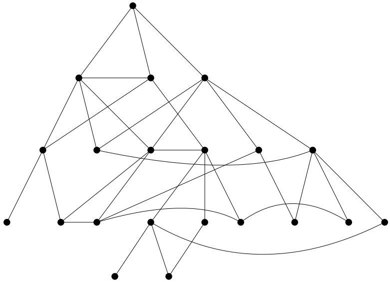


	a	b	c	d	e	f	g
degree	0	0	0	0	0	0	0

Outline

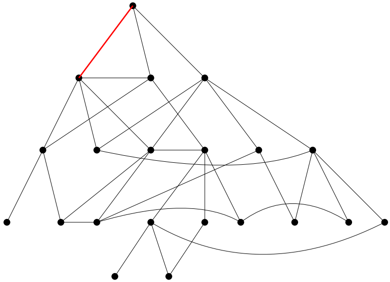
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



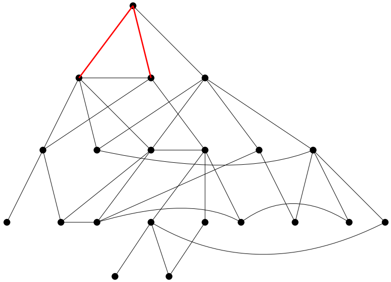
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



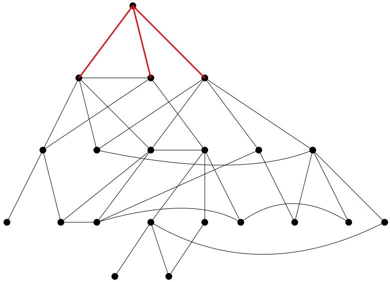
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



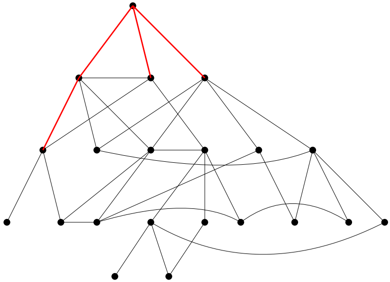
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



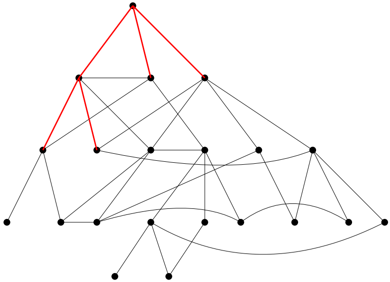
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



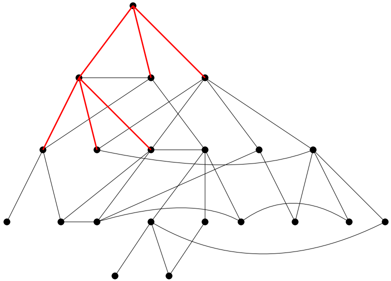
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



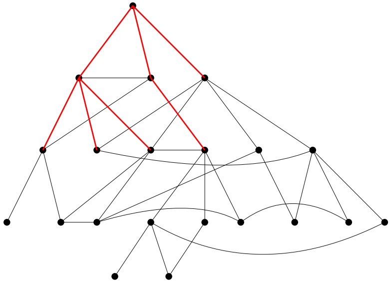
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



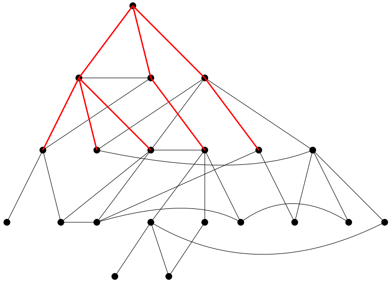
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



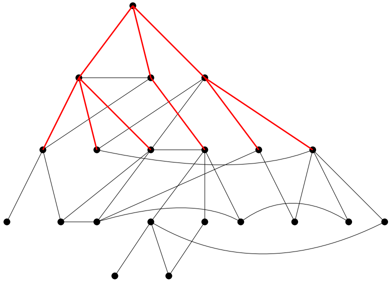
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



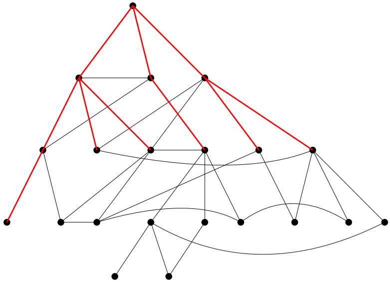
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



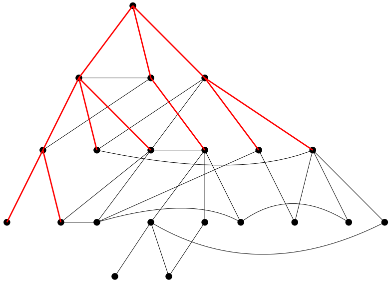
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



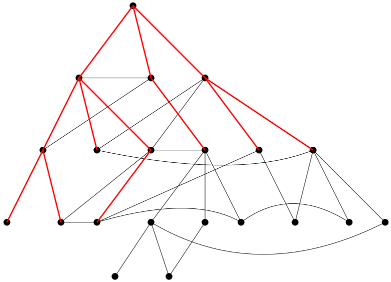
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



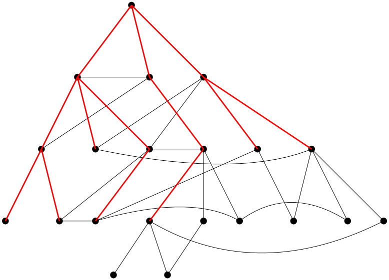
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



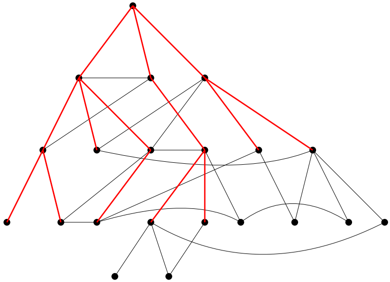
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



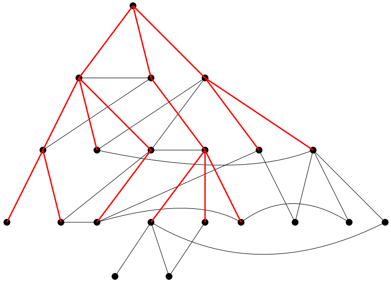
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



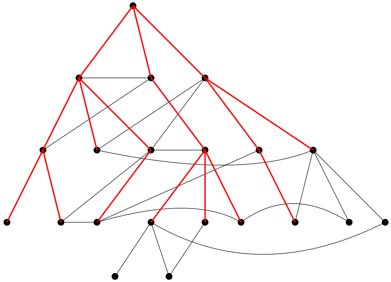
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



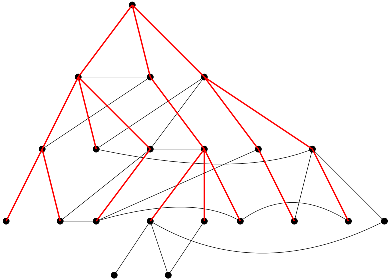
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



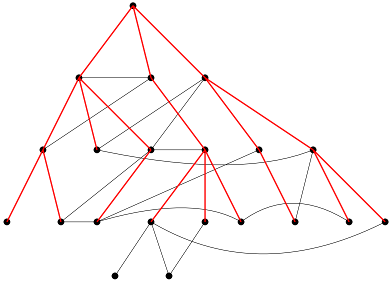
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



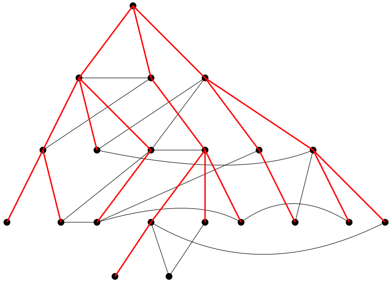
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



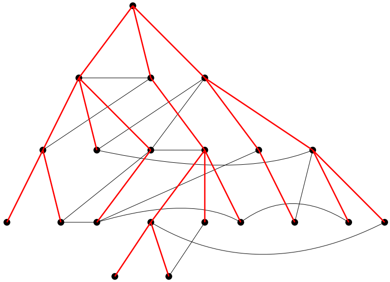
Properties of a BFS Tree

Given a BFS tree T of a connected graph G



Properties of a BFS Tree

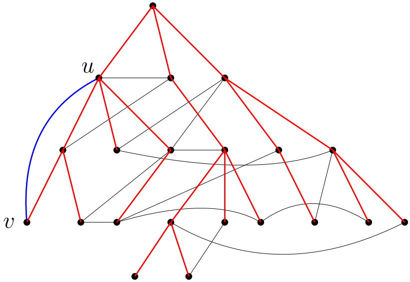
Given a BFS tree T of a connected graph G



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

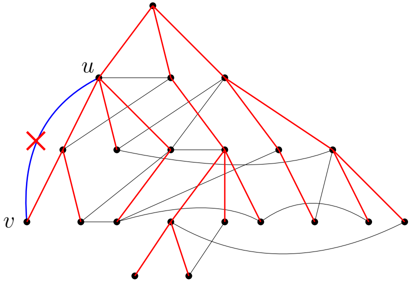
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

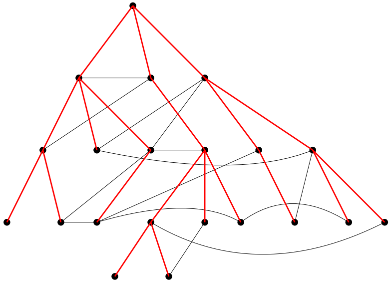
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

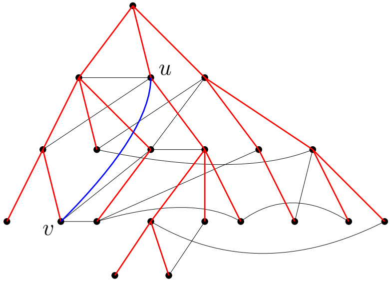
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

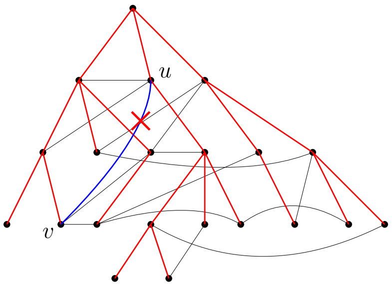
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

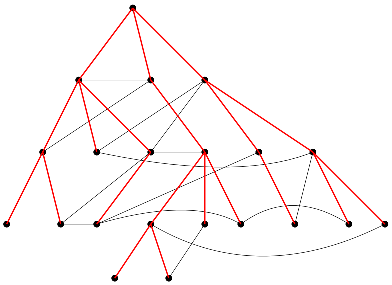
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?
- No. v should be a child of u.



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

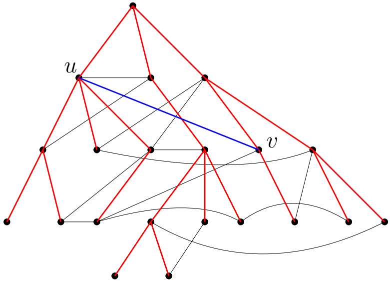
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?
- No. v should be a child of u.



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

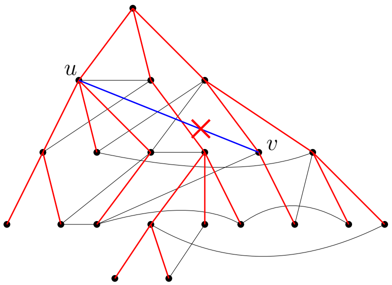
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?
- No. v should be a child of u.
- Can there be a horizontal edge (u, v), where u is 1 level above v, but v's parent is to the right of u?



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

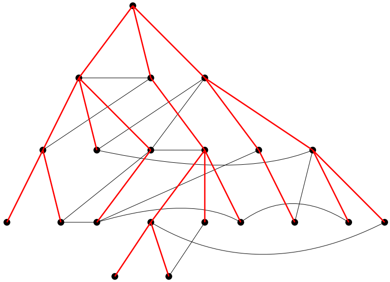
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?
- No. v should be a child of u.
- Can there be a horizontal edge (u, v), where u is 1 level above v, but v's parent is to the right of u?
- No. v should be a child of u.



Properties of a BFS Tree

Given a BFS tree T of a connected graph G

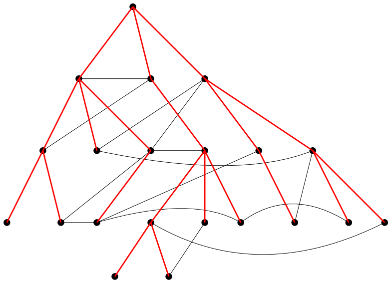
- Can there be a vertical edge (u, v), $u \geq 2$ levels above v?
- No. v should be a child of u
- Can there be a horizontal edge (u, v) $u \geq 2$ levels above v?
- No. v should be a child of u.
- Can there be a horizontal edge (u, v), where u is 1 level above v, but v's parent is to the right of u?
- No. v should be a child of u.



Properties of a BFS Tree

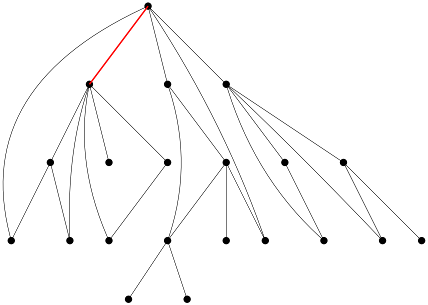
Given a BFS tree T of a connected graph G, other than the tree edges, we only have horizontal edges (u, v), where

- either u and v are at the same level
- or u is 1 level above v, and v's parent is to the left of u, (or vice versa)



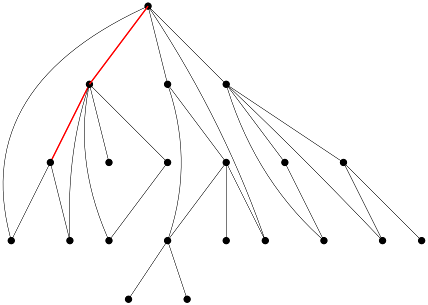
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



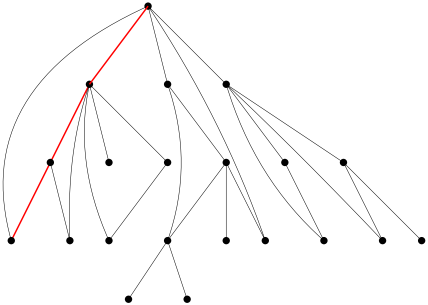
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



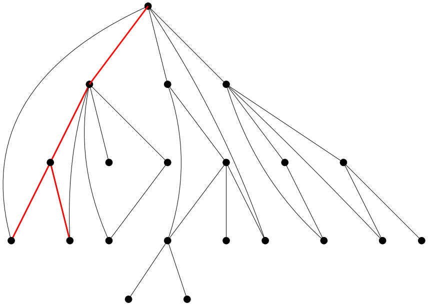
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



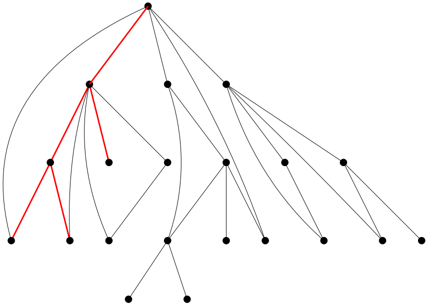
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



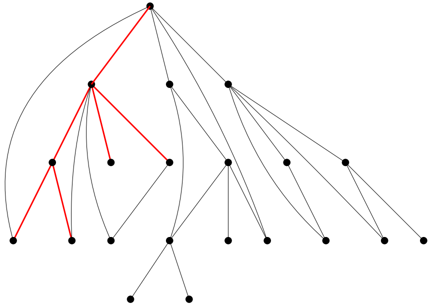
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



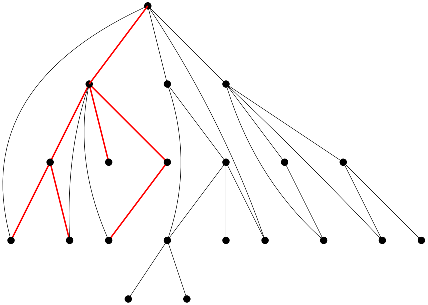
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



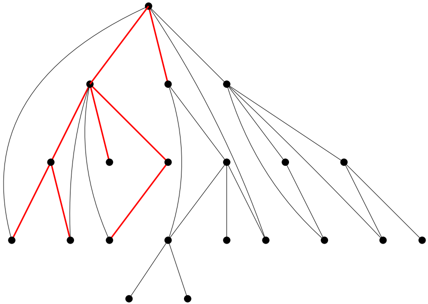
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



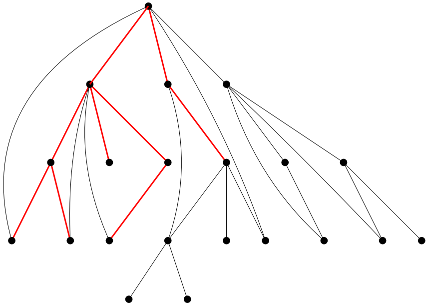
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



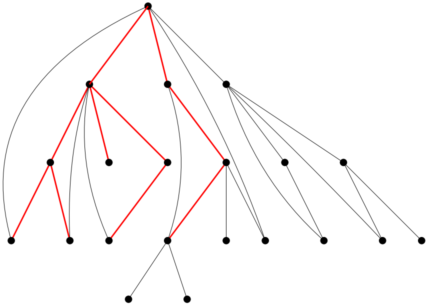
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



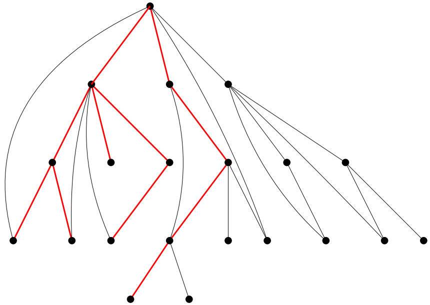
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



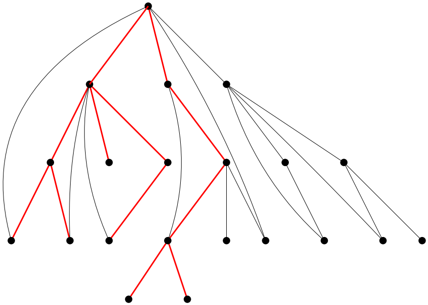
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



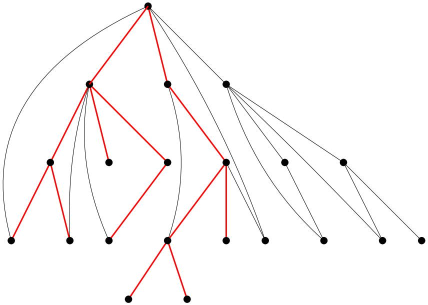
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



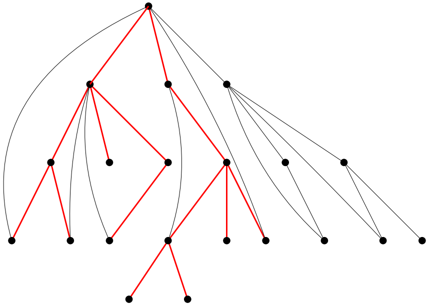
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



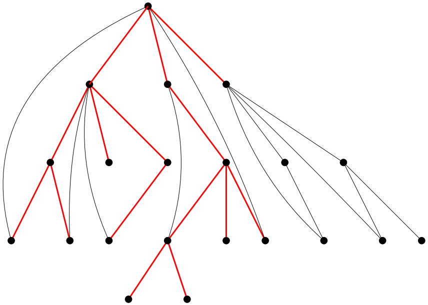
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



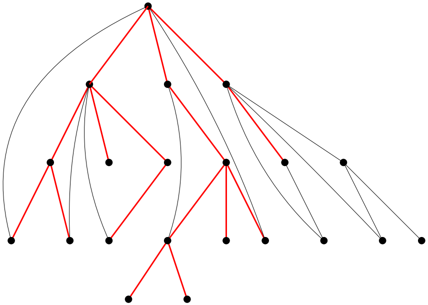
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



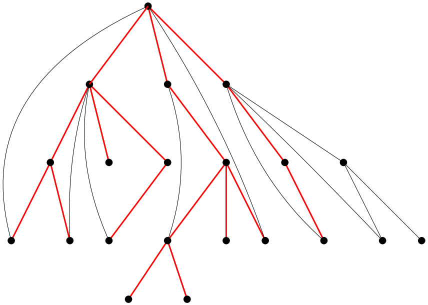
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



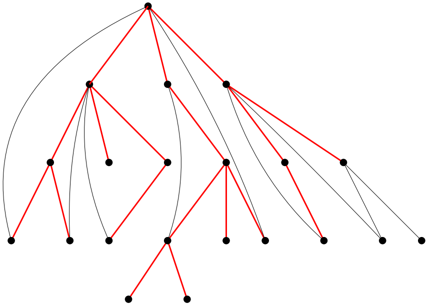
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



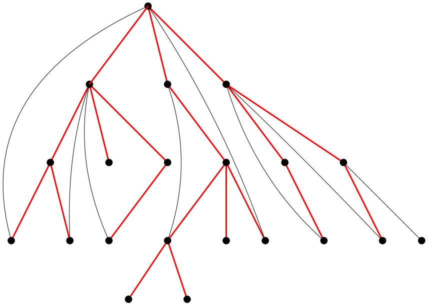
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



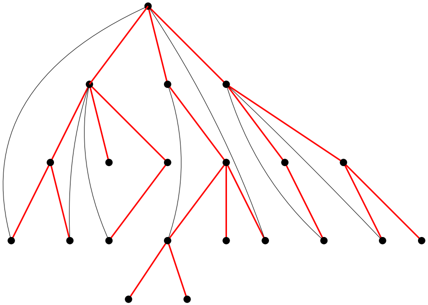
Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,



Properties of a DFS Tree

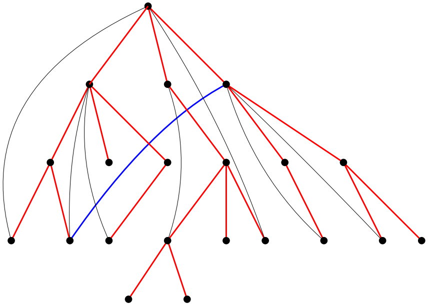
Given a tree DFS tree T of a graph (connected) G,



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

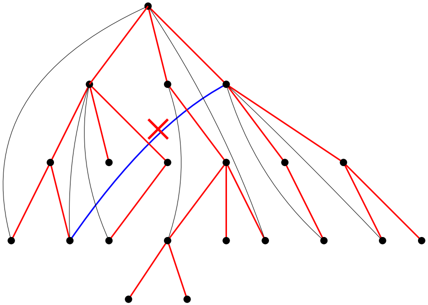
- Can there be a horizontal edge (u, v)?



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

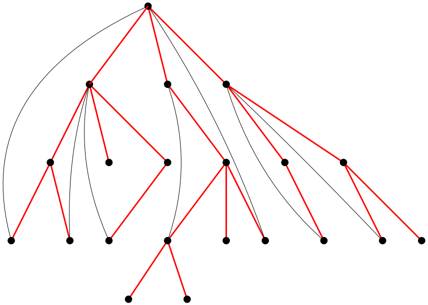
- Can there be a horizontal edge (u, v)?
- No.



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

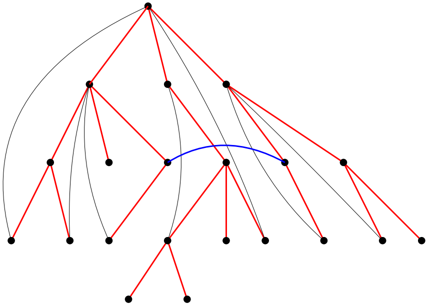
- Can there be a horizontal edge (u, v)?
- No.



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

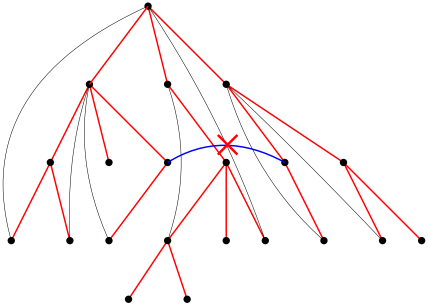
- Can there be a horizontal edge (u, v)?
- No.



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

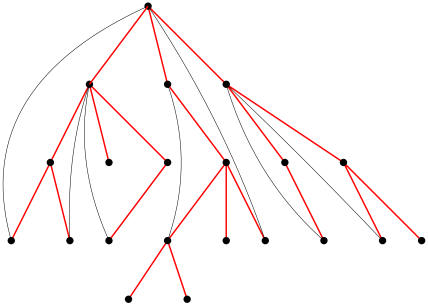
- Can there be a horizontal edge (u, v)?
- No.



Properties of a DFS Tree

Given a tree DFS tree T of a graph (connected) G,

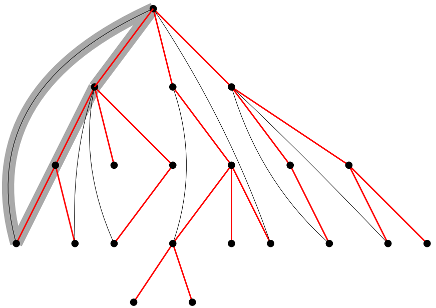
- Can there be a horizontal edge (u, v)?
- No.
- All non-tree edges are vertical edges.



Properties of a DFS Tree

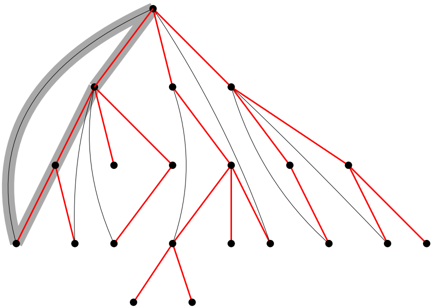
Given a tree DFS tree T of a graph (connected) G,

- Can there be a horizontal edge (u, v)?
- No.
- All non-tree edges are vertical edges.
- A vertical edge (u, v) and its the edges in the path from u to v in T form a cycle; we call it a canonical cycle.



Properties of a DFS Tree

Lemma If G contains a cycle, then it has a canonical cycle.

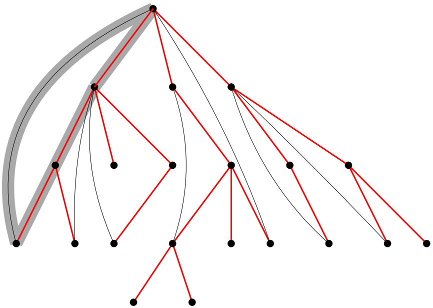


Properties of a DFS Tree

Lemma If G contains a cycle, then it has a canonical cycle.

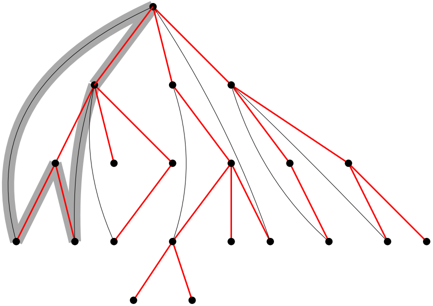
Proof.

- If G contains a cycle, then it must have at least non-tree edge.
- W.r.t DFS tree T, we can only have vertical + tree edges
- $\exists$ at least one vertical edge
- There is a canonical cycle $\square$



Properties of a DFS Tree

Lemma If G contains a cycle, then it has a canonical cycle.



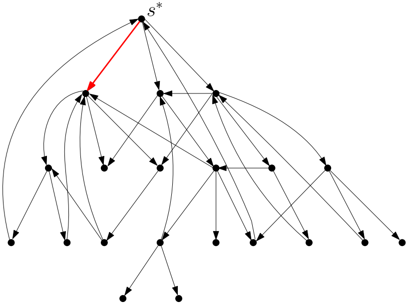
Proof.

- If G contains a cycle, then it must have at least non-tree edge.
- W.r.t DFS tree T, we can only have vertical + tree edges
- $\exists$ at least one vertical edge
- There is a canonical cycle $\square$

- There might or might not be non-canonical ones.

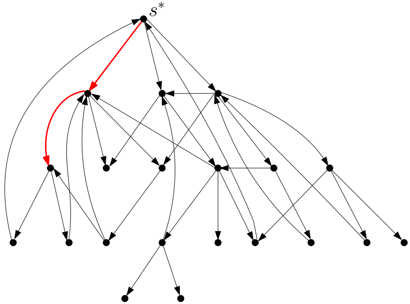
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



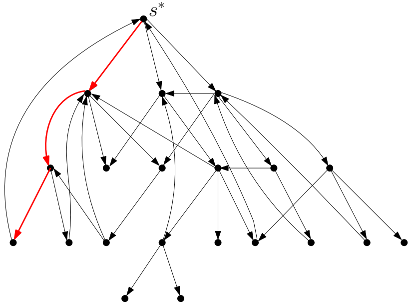
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



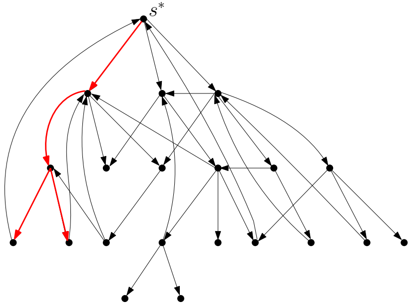
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



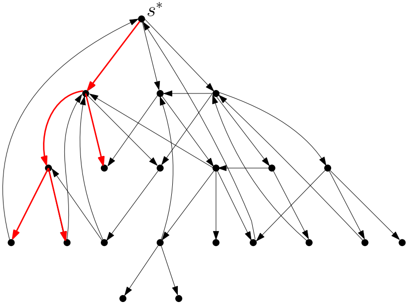
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



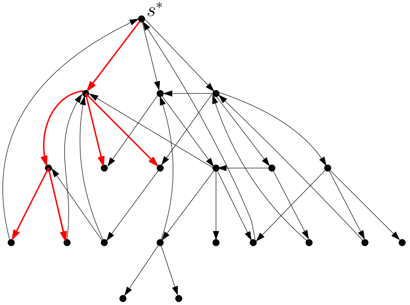
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



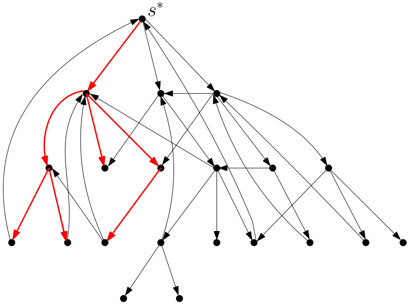
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



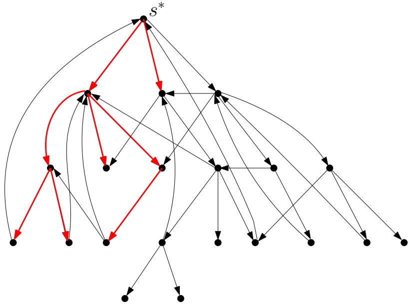
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



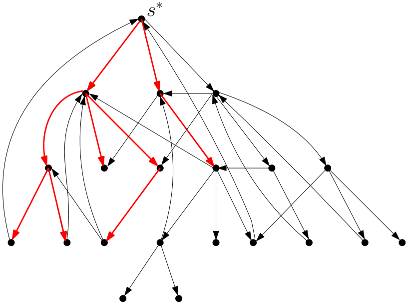
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



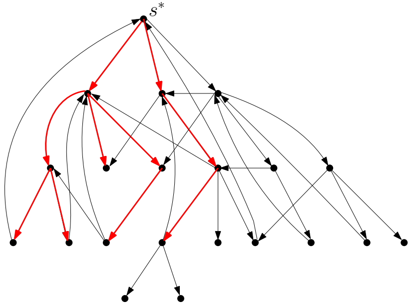
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



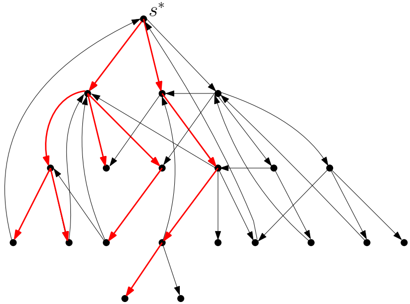
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



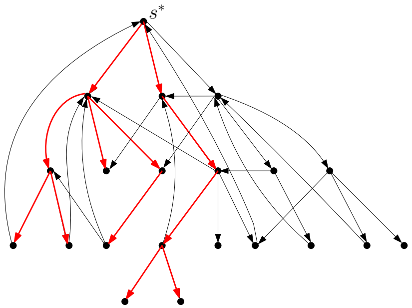
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



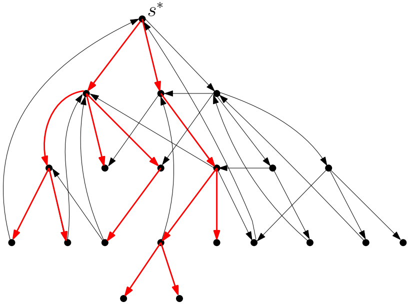
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



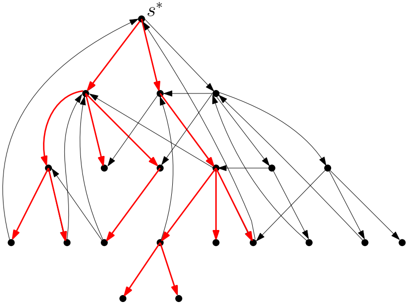
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



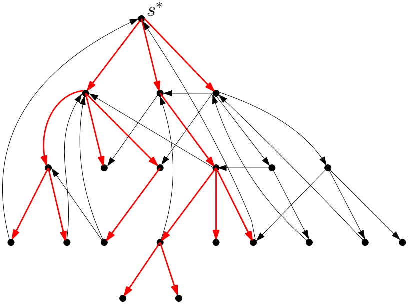
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



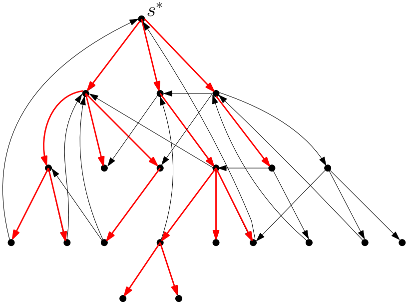
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



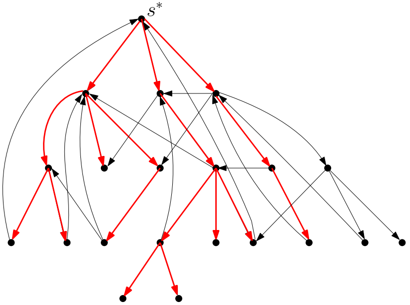
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



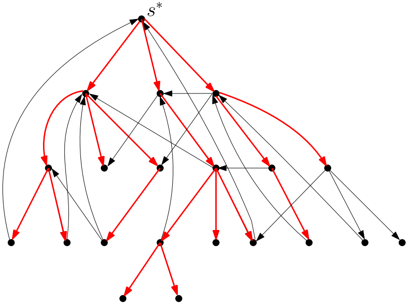
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



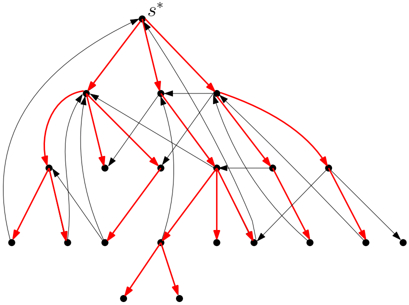
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



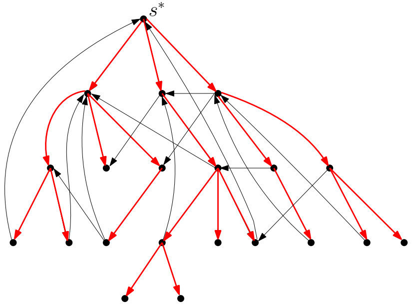
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



Properties of a DFS Tree Over a Directed Graph

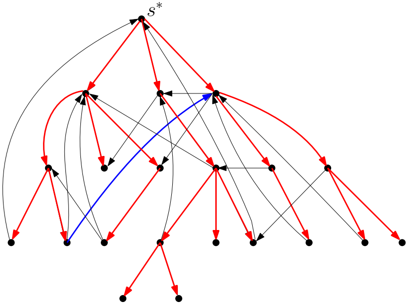
Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

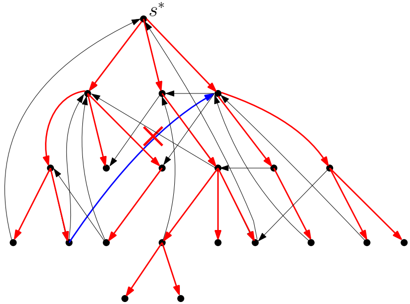
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

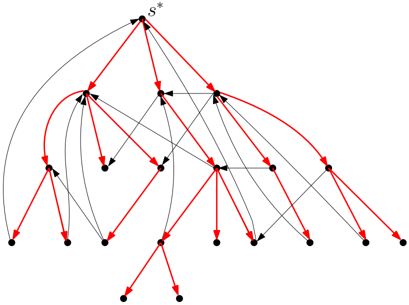
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

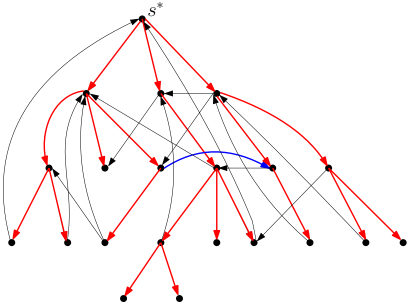
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

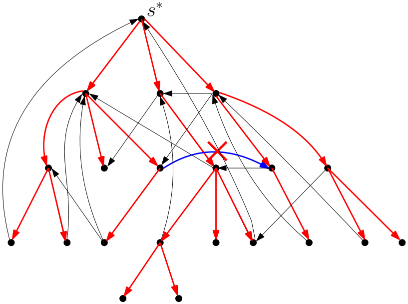
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

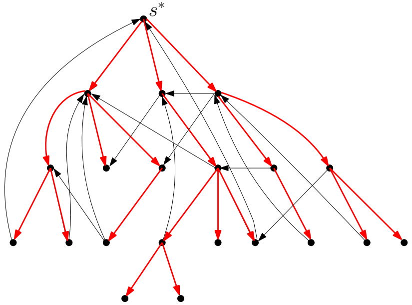
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

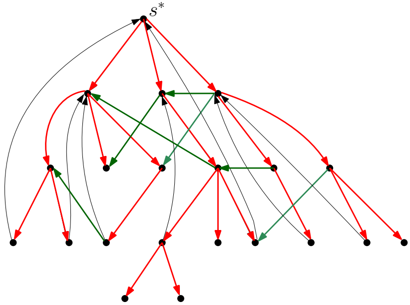
- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.



Properties of a DFS Tree Over a Directed Graph

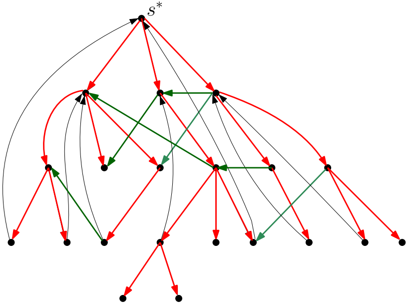
Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

- Can there be a horizontal (directed) edge (u, v) where u is visited before v?
- No.
- However, there can be horizontal edges (u, v) where u is visited after v.



Properties of a DFS Tree Over a Directed Graph

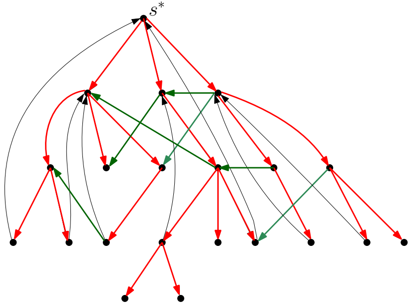
Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

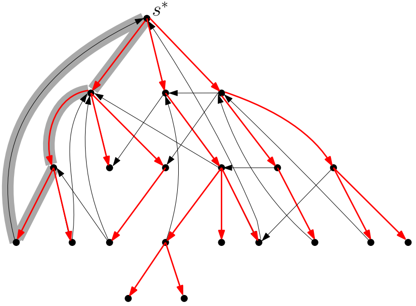
- Other than tree edges, there are two types of edges:
  - vertical edges directed to ancestors
  - horizontal edges (u, v) where u is visited after v.



Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree T of a directed graph G, assuming all vertices can be reached from the starting vertex s^*

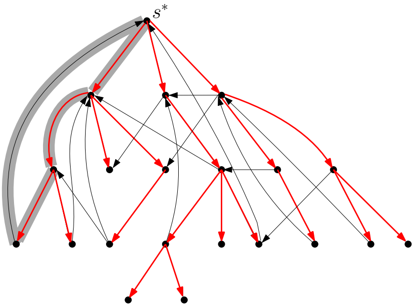
- Other than tree edges, there are two types of edges:
  - vertical edges directed to ancestors
  - horizontal edges (u, v) where u is visited after v.
- An vertical edge (u, v) and the tree edges in the tree path from v to u form a cycle, and we call it a canonical cycle.



Properties of a DFS Tree Over a Directed Graph

Lemma If there is a cycle in the directed graph G, then there must be a canonical one.

Proof.

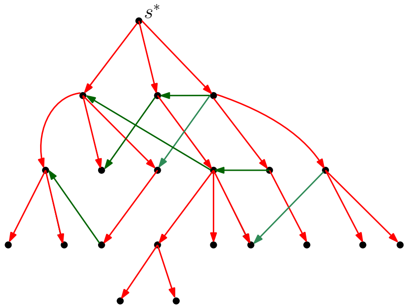


Properties of a DFS Tree Over a Directed Graph

Lemma If there is a cycle in the directed graph G, then there must be a canonical one.

Proof.

- Focus on tree edges and horizontal edges

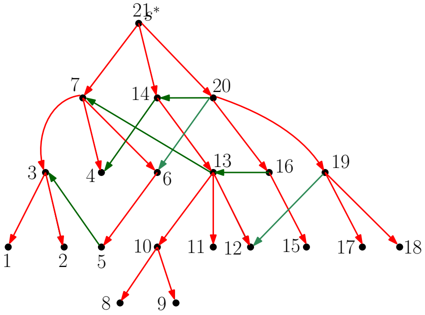


Properties of a DFS Tree Over a Directed Graph

Lemma If there is a cycle in the directed graph G, then there must be a canonical one.

Proof.

- Focus on tree edges and horizontal edges
- post-order-traversal of T gives a reversed topological ordering

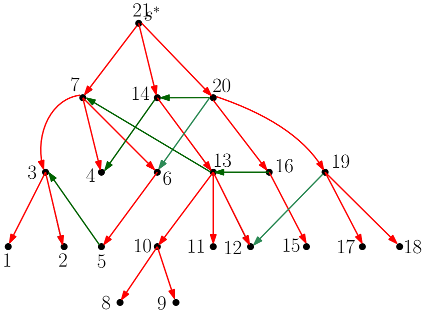


Properties of a DFS Tree Over a Directed Graph

Lemma If there is a cycle in the directed graph G, then there must be a canonical one.

Proof.

- Focus on tree edges and horizontal edges
- post-order-traversal of T gives a reversed topological ordering
- Without vertical edges, G has no cycles □

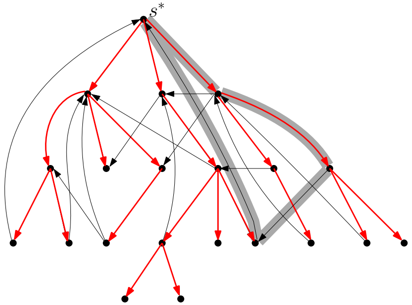


Properties of a DFS Tree Over a Directed Graph

Lemma If there is a cycle in the directed graph G, then there must be a canonical one.

Proof.

- Focus on tree edges and horizontal edges
- post-order-traversal of T gives a reversed topological ordering
- Without vertical edges, G has no cycles □



- Again, there might be non-canonical cycles.

Cycle Detection Using DFS in Directed Graphs

Algorithm 1 Check-Cycle-Directed

```
add a source s* to G and edges from s* to all other vertices.
visited ← boolean array over V, with visited[v] = false, ∀v
instack ← boolean array over V, with instack[v] = false, ∀v
DFS(s*)
return "no cycle"
```

Algorithm 2 DFS(v)

```
visited[v] ← true, instack[v] ← true
for every outgoing edge (v, u) of v do
    if inqueue[u] then                              ▷ Find a vertical edge
        exit the whole algorithm, by returning "there is a cycle"
    else if visited[u] = false then
        DFS(u)
instack[v] ← false
```