Introduction to Algorithms

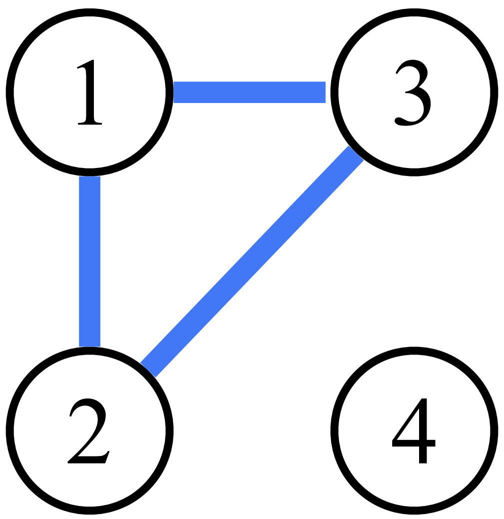
Meng-Tsung Tsai

11/19/2019

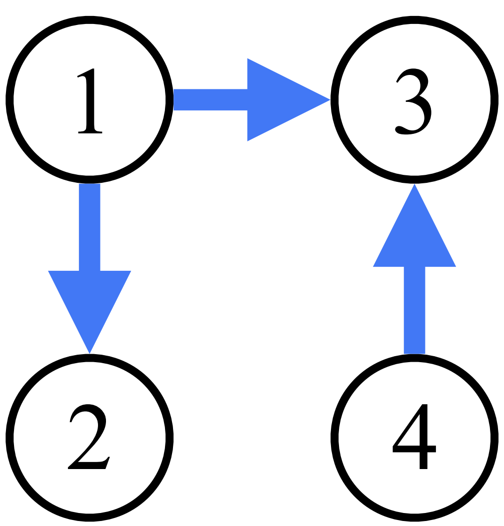
Graph Representation

What is a graph?

A structure that contains nodes and edges.



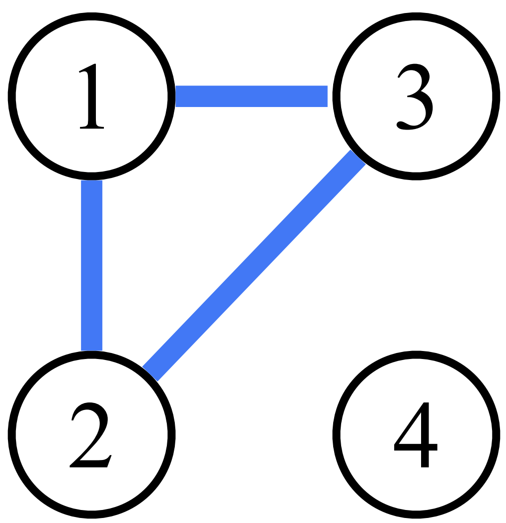
An undirected graph G = (V, E) where
the node set V = {1, 2, 3, 4}, and
the edge set E = {(1, 2), (1, 3), (2, 3)}.



A directed graph G = (V, E) where
the node set V = {1, 2, 3, 4}, and
the edge set E = {(1, 2), (1, 3), (4, 3)}.

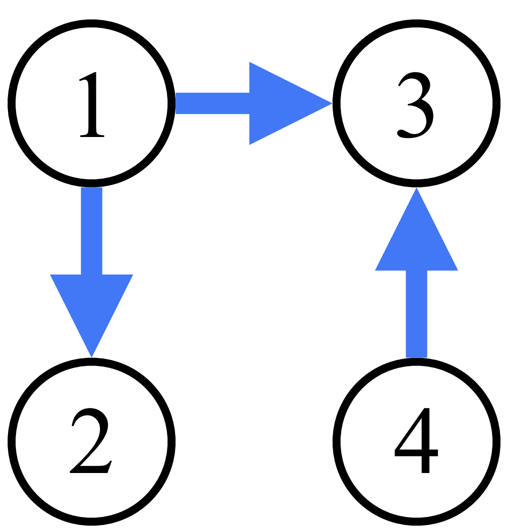
What is a graph?

A structure that contains nodes and edges.



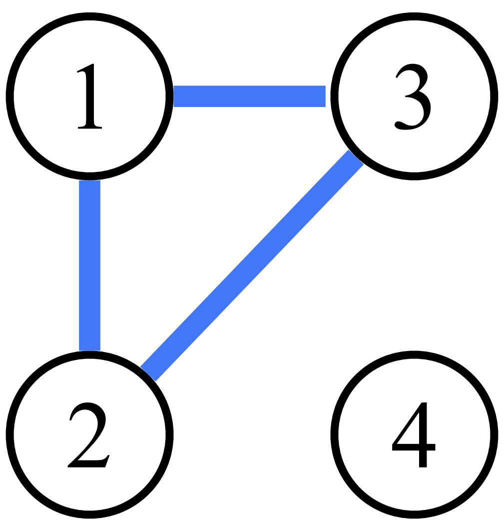
An undirected graph G = (V, E) where
the node set V = {1, 2, 3, 4}, and
the edge set E = {(1, 2), (1, 3), (2, 3)}.

(u, v) = (v, u) in an undirected graph.



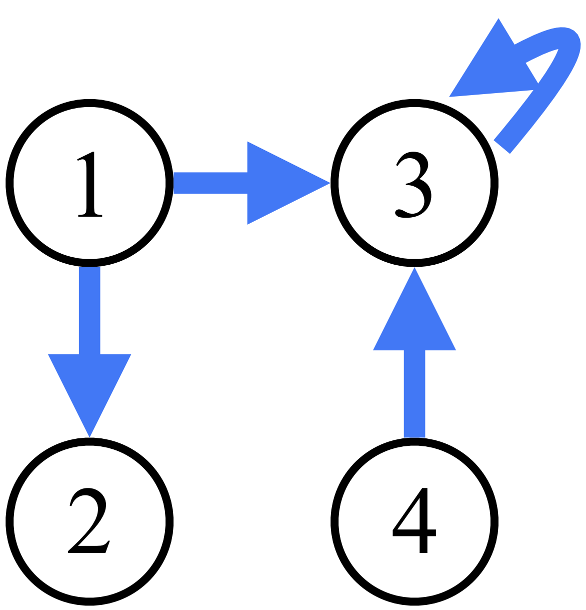
A directed graph G = (V, E) where
the node set V = {1, 2, 3, 4}, and
the edge set E = {(1, 2), (1, 3), (4, 3)}.

Representing a graph by adjacency matrix



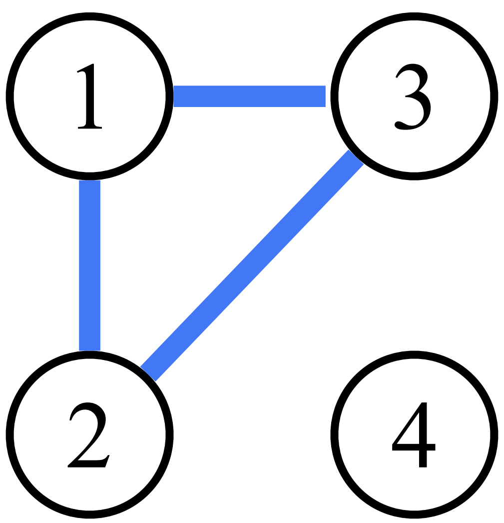
A	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	0
4	0	0	0	0

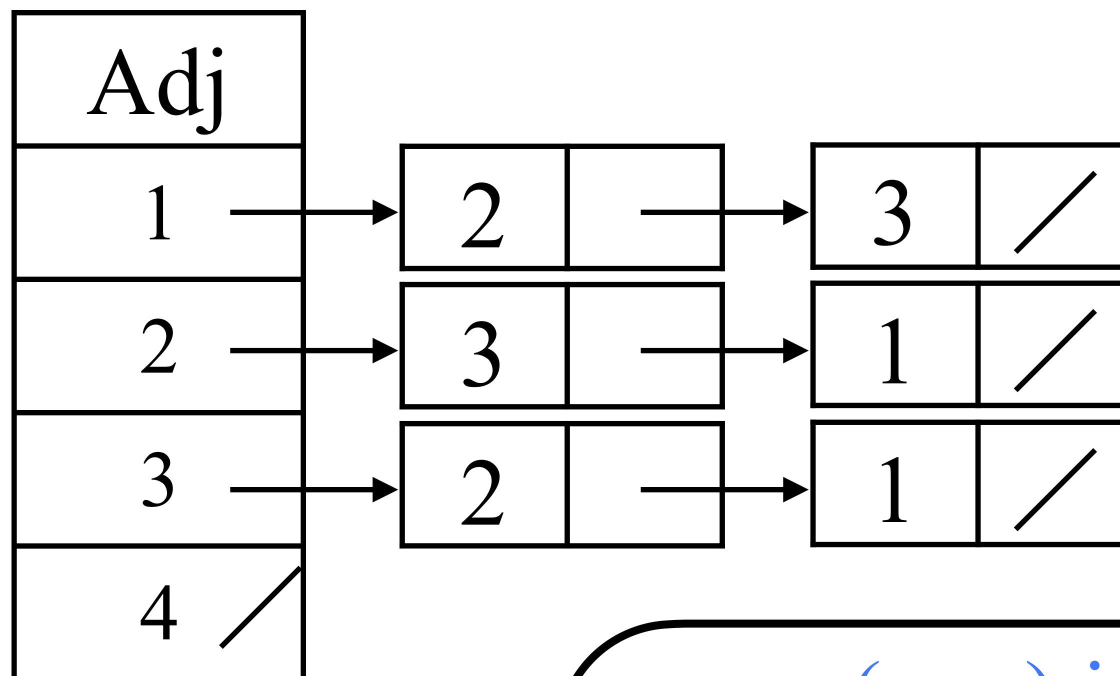
(u, v) in E iff A[u][v] = 1.



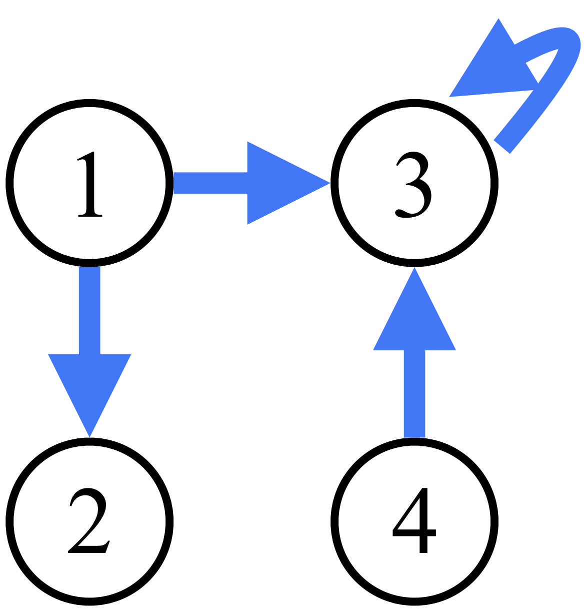
A	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	0	1	0
4	0	0	1	0

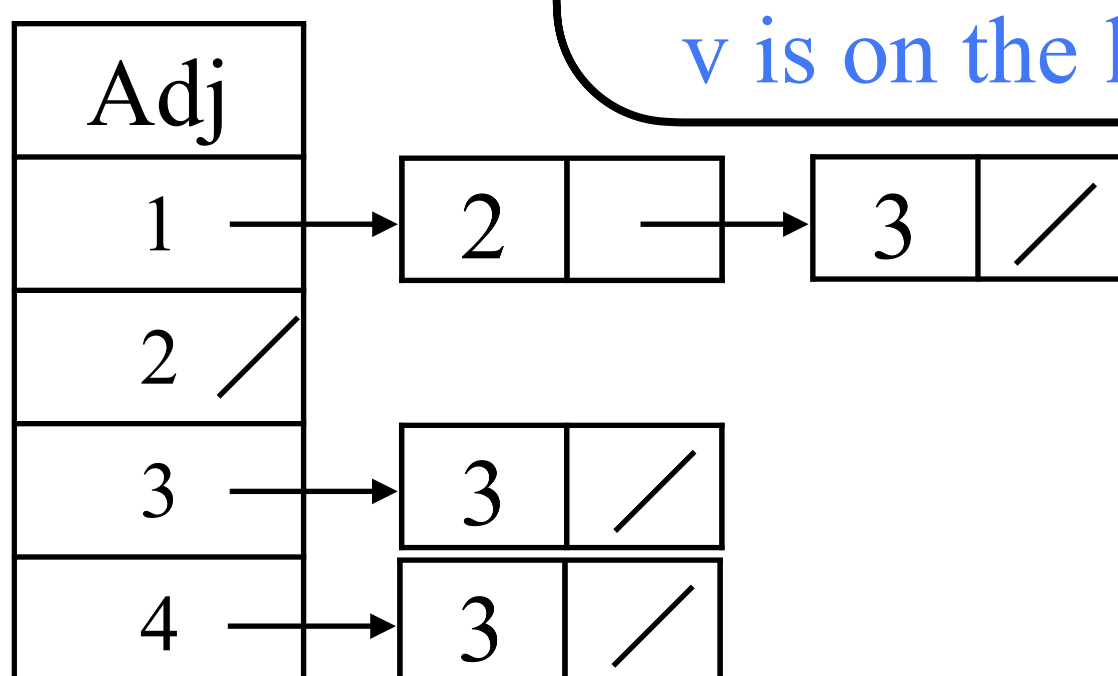
Representing a graph by adjacency list





(u, v) in E iff
v is on the list Adj[u].





Pros and Cons

In graph theory,
n = |V| and m = |E|.

	adjacency matrix	adjacency list
space usage	$O(n^2)$	O(n+m)
cost to decide if (u, v) in E	O(1)	O(deg(u))
add an edge (u, v)	O(1)	O(1)
delete an edge (u, v)	O(1)	O(deg(u))
iterate over u's neighbors	O(n)	O(deg(u))

Exercise

Let $G = (V, E)$ be a simple graph, i.e. containing no self-loops and multi-edges. Devise an $O(n+m)$-time algorithm to sort the edge set E so that

edge (u_1, v_1) precedes edge (u_2, v_2) iff either $(u_1 < u_2)$
or $(u_1 = u_2$ and $v_1 < v_2)$.

Exercise

Let G be a simple undirected graph, i.e. containing no self-loops and multi-edges. Assume that $n < m = o(n^2)$.

(a) Devise an $O(nm)$-time algorithm to enumerate all simple paths of length 2. We say a path is **simple** if all nodes on it are distinct.

(b) Devise an $O(m^{1.5})$-time algorithm to enumerate all simple cycles of length 3, i.e. triangle. We say a cycle is simple if no nodes on it repeat.

Exercise

Let G be a simple undirected graph, i.e. containing no self-loops and multi-edges. Assume that $n < m = o(n^2)$.

(a) Devise an $O(nm)$-time algorithm to enumerate all simple paths of length 2. We say a path is **simple** if all nodes on it are distinct.

(b) Devise an $O(m^{1.5})$-time algorithm to enumerate all simple cycles of length 3, i.e. triangle. We say a cycle is simple if no nodes on it repeat.

P_2 is a subgraph simpler than C_3.
Why is enumerating P_2 slower than enumerating C_3?

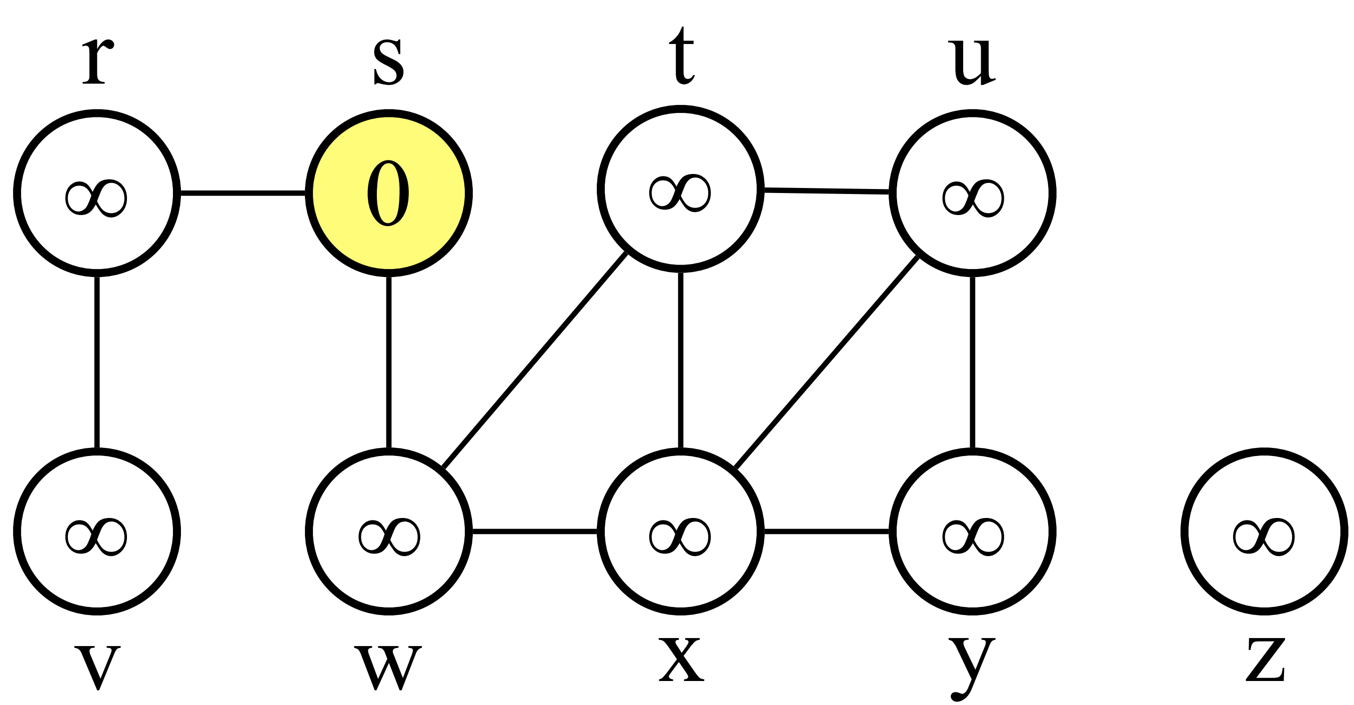
Breadth-first Search

Goal

Given a graph G and a node s, we would like to explore the edges of G to discover the nodes that are reachable from s.

For any pair of nodes u and v that are reachable from s, if dis(s, u) < dis(s, v), then u is visited earlier than v.

dis(s, v) denotes the min # edges in any path from s to v.

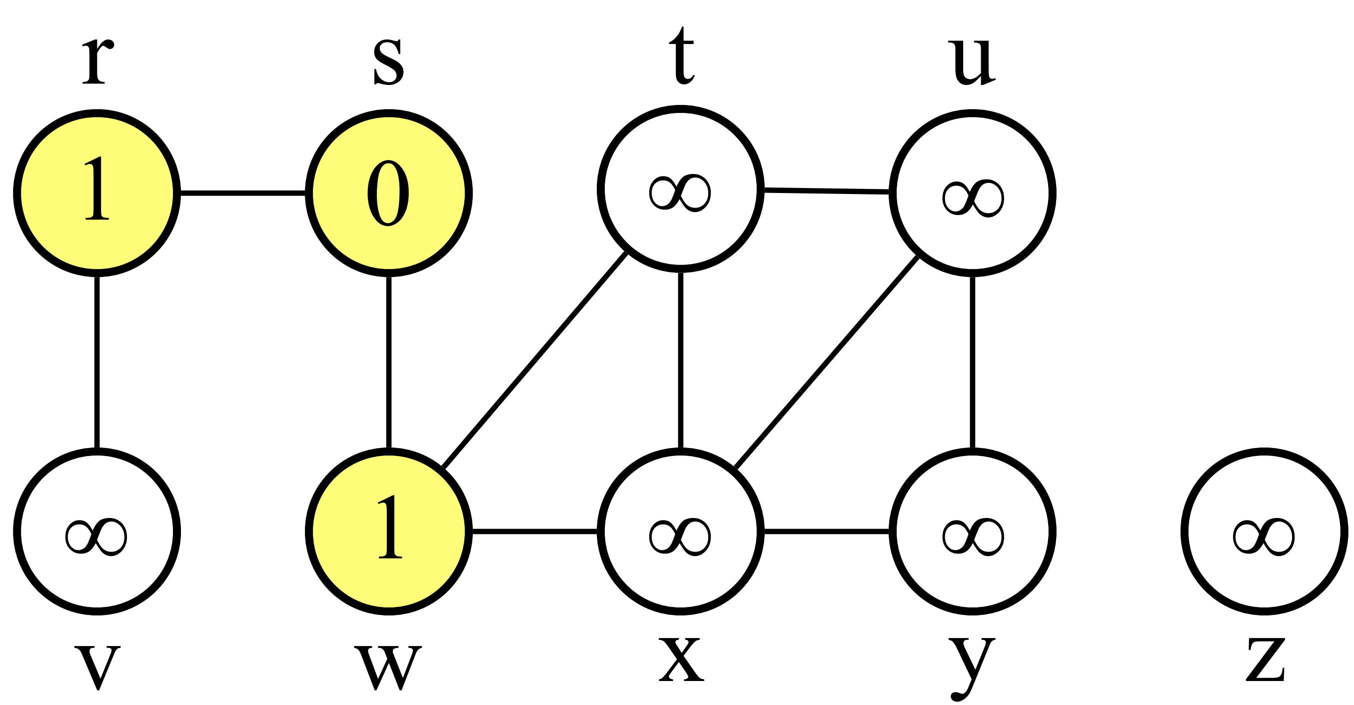


Goal

Given a graph G and a node s, we would like to explore the edges of G to discover the nodes that are reachable from s.

For any pair of nodes u and v that are reachable from s, if dis(s, u) < dis(s, v), then u is visited earlier than v.

dis(s, v) denotes the min # edges in any path from s to v.

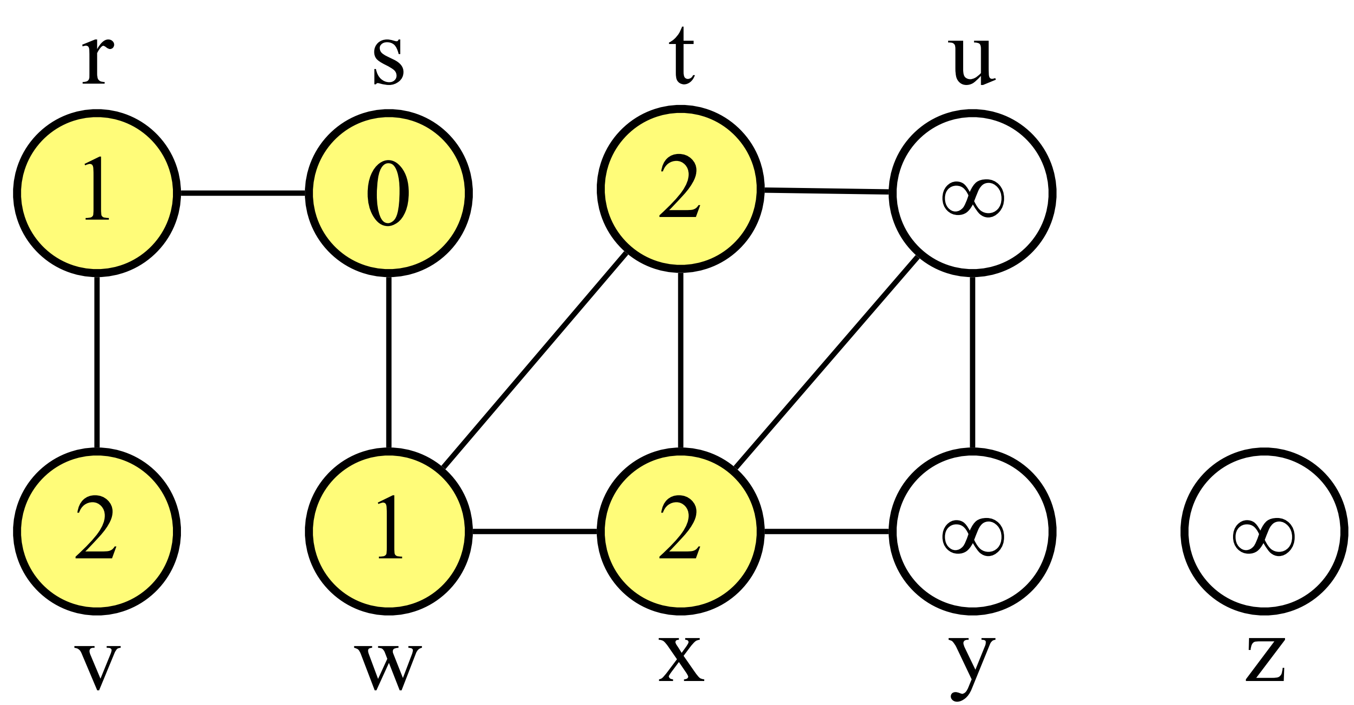


Goal

Given a graph G and a node s, we would like to explore the edges of G to discover the nodes that are reachable from s.

For any pair of nodes u and v that are reachable from s, if dis(s, u) < dis(s, v), then u is visited earlier than v.

dis(s, v) denotes the min # edges in any path from s to v.

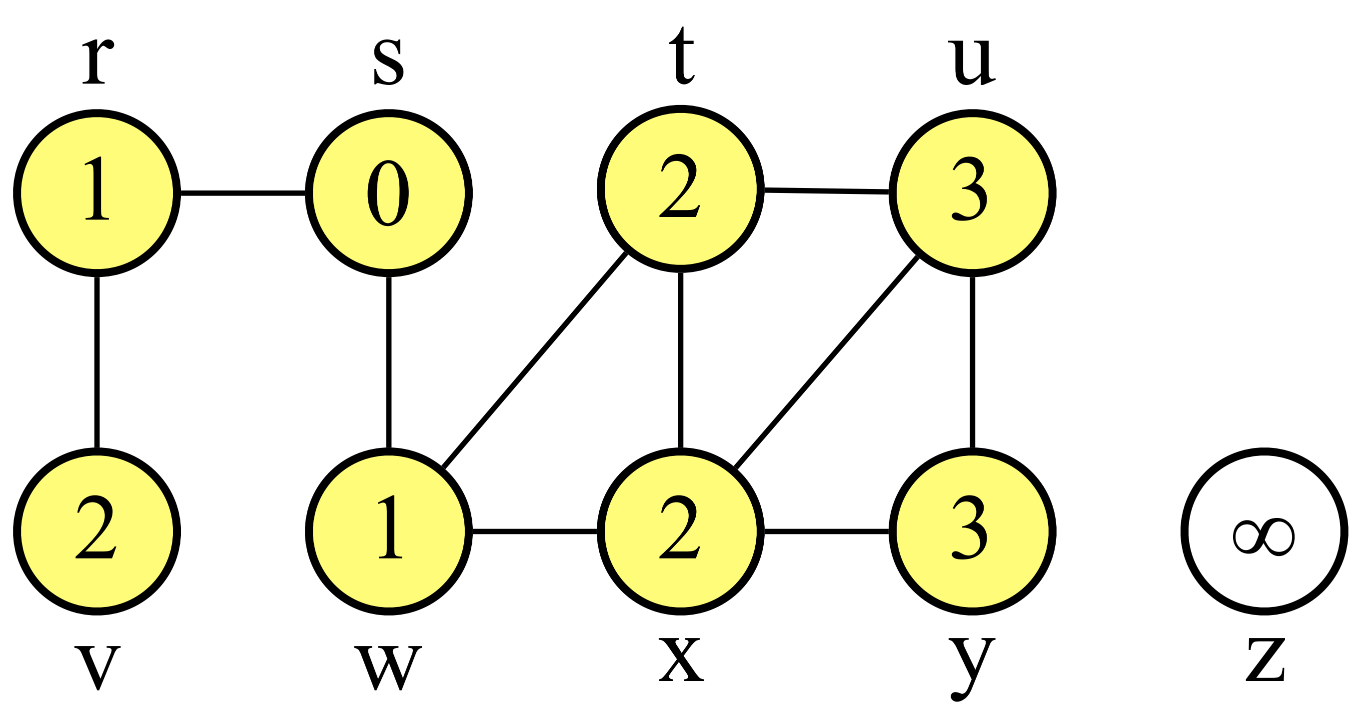


Goal

Given a graph G and a node s, we would like to explore the edges of G to discover the nodes that are reachable from s.

For any pair of nodes u and v that are reachable from s, if dis(s, u) < dis(s, v), then u is visited earlier than v.

dis(s, v) denotes the min # edges in any path from s to v.



Approach

$BFS_0(G, s)\{$

$\quad$ Initially, $S_0 = \{s\}$.

$\quad$ $\text{for}(i = 1;\ S_{i-1} \neq \emptyset;\ {++i})\{$

$\quad\quad$ $S_i = \{x : (u, x) \in E \text{ for some } u \in S_{i-1} \text{ and } x \notin S_j \text{ for } j < i\};$

$\quad$ $\}$

$\}$

Claim: $S_i = D_i$

where D_i denotes the set of nodes v whose $dis(s, v) = i$.

Approach

$BFS_0(G, s)\{$

  Initially, $S_0 = \{s\}$.

  $for(i = 1;\ S_{i-1} \neq \emptyset;\ ++i)\{$

    $S_i = \{x : (u, x) \in E \text{ for some } u \in S_{i-1} \text{ and } x \notin S_j \text{ for } j < i\};$

  $\}$

$\}$

Claim: $S_i = D_i$

where D_i denotes the set of nodes v whose $dis(s, v) = i$.

Clearly, $S_0 = D_0$. Assume that $S_i = D_i$ for every $i < k$.

Approach

$D_k \equiv \{x$: there is a path from s to x using k edges and there is no path from s to x using $< k$ edges$\}$.

Approach

$D_k \equiv$ {x : there is a path from s to x using k edges and there is no path from s to x using $<$ k edges}.

$D_k =$ {x : $(u, x) \in E$ and $u \in D_{k-1}$ and there is no path from s to x using $<$ k edges}.

Approach

$D_k = \{x : (u, x) \in E$ and $u \in D_{k-1}$ and
there is no path from s to x using $< k$ edges$\}$.

$D_k = \{x : (u, x) \in E$ and $u \in D_{k-1}$ and
$x \notin D_j$ for any $j < k\}$.

Approach

$D_k = \{x : (u, x) \in E \text{ and } u \in D_{k-1}$ and
$x \notin D_j$ for any $j < k\}$.

$D_k = \{x : (u, x) \in E \text{ and } u \in S_{k-1}$ and
$x \notin D_j$ for any $j < k\}$.

Approach

$D_k = \{x : (u, x) \in E$ and $u \in S_{k-1}$ and
$x \notin D_j$ for any $j < k\}$.

$D_k = \{x : (u, x) \in E$ and $u \in S_{k-1}$ and
$x \notin S_j$ for any $j < k\} = S_k$. $\Rightarrow$ $S_i = D_i$ for all i's.

Polishing (1/2)

```
BFS₁(G, s){
    visited[1..n] = {no};

    Initially, S₀ = {s}. visited[s] = yes;

    for(i = 1; Sᵢ₋₁ ≠ ∅; ++i){
        Sᵢ = ∅;
        foreach (u ∈ Sᵢ₋₁)
            foreach (v ∈ Adj[u])
                if (visited[v] = no){
                    Sᵢ ← Sᵢ ∪ {v}; visited[v] = yes;
                }
    }
}
```

Polishing (2/2)

```
BFS(G, s){ // merge all Si's into a single queue Q
    visited[1..n] = {no};
    dis[1..n] = {∞}; parent[1..n] = {NIL};

    EnQueue(Q, s); visited[s] = yes; dis[s] = 0;

    for(i = 1; Q ≠ ∅; ++i){
        u = DeQueue(Q);
        foreach (v ∈ Adj[u])
            if (visited[v] = no){
                EnQueue(Q, v); visited[v] = yes; dis[v] = dis[u]+1;
                parent[v] = u;
            }
    }
}
```

Summary

Every node is being placed in the queue at most once.
$\Rightarrow$ Therefore, each adjacency list is scanned by at most once.
$\Rightarrow$ The total running time is O(n+m).

Let T = {(parent[u], u) : parent[u] $\neq$ NIL}. T is called the BFS-tree rooted at node s.

T has no cycle because dis[parent[u]] < dis[u] and every node has a single parent.

Applications

Diameter

The diameter of a tree $T = (V, E)$ is defined as

$$\max_{a,\, b \in V} dis(a, b).$$

Devise an $O(n)$-time algorithm to compute the diameter.

A simple case

The diameter of a tree $T = (V, E)$ is defined as

$$\max_{a,\, b \in V} \mathrm{dis}(a, b).$$

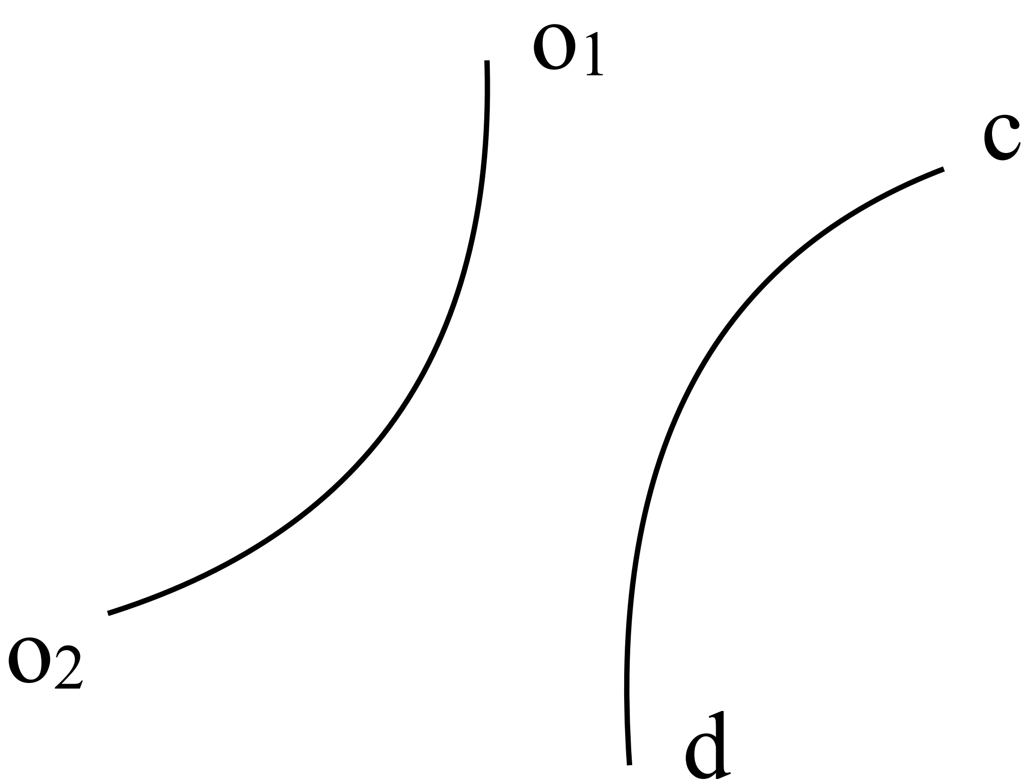
Devise an $O(n)$-time algorithm to compute the diameter.

Suppose you are given a node $s \in \{o_1, o_2\}$ such that $\mathrm{dis}(o_1, o_2) = \max_{a,b \in V} \mathrm{dis}(a, b)$. Run BFS(T, s) and record the largest dis[x] among $x \in V$, then we obtain the diameter.

To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

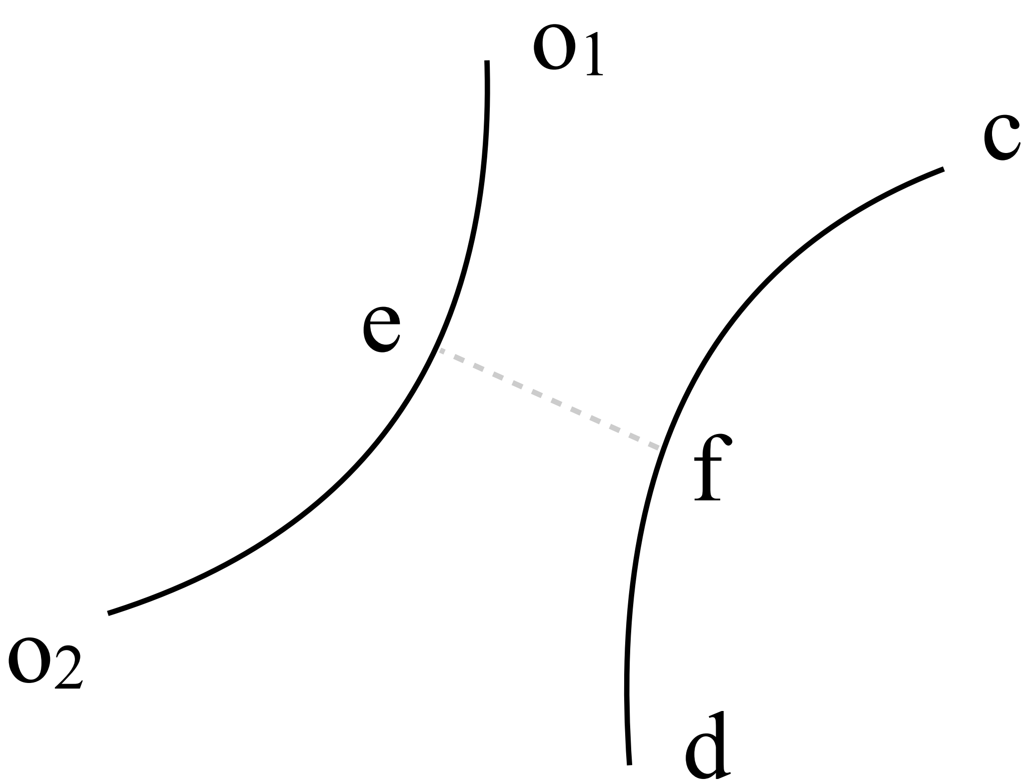
Case 1. The path connecting o_1, o_2 doesn't intersect with the path connecting c, d.



To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

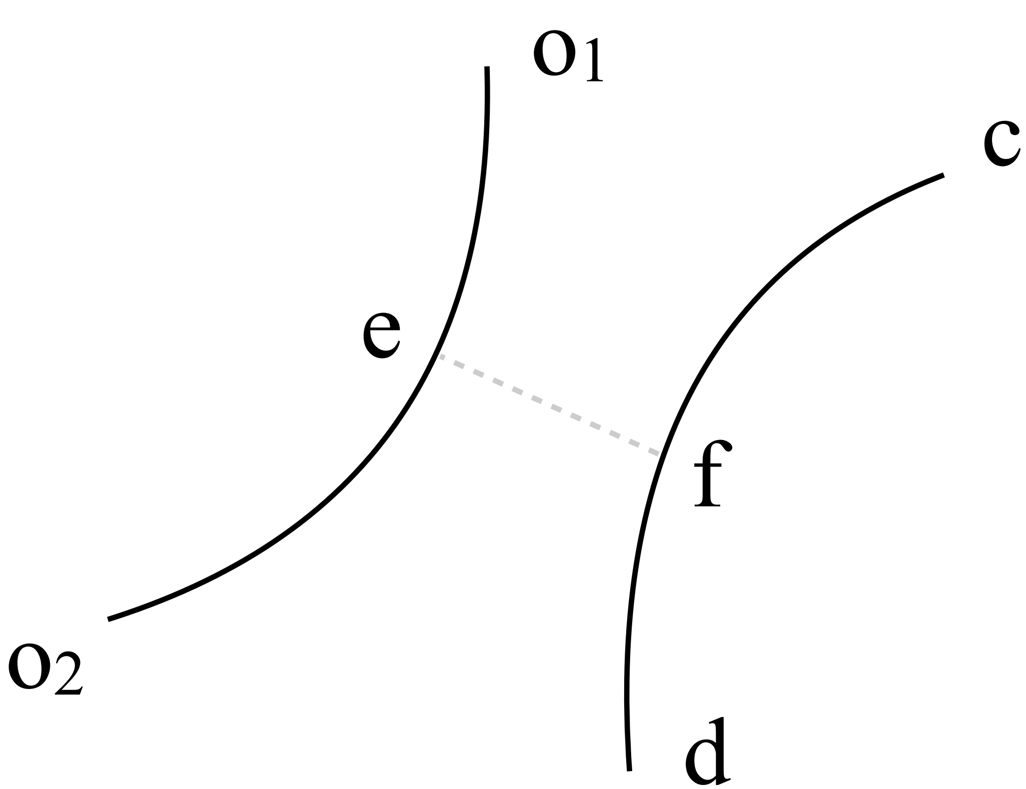
Case 1. The path connecting o_1, o_2 doesn't intersect with the path connecting c, d.



To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

Case 1. The path connecting o_1, o_2 doesn't intersect with the path connecting c, d.

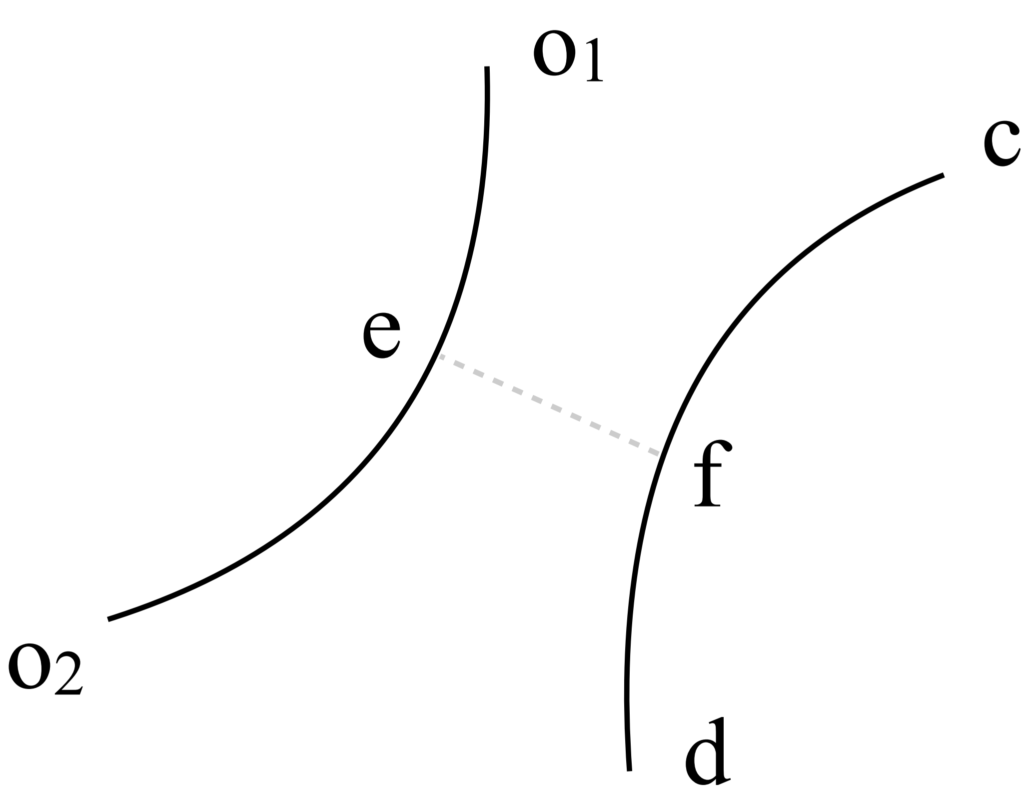


Find the path e...f that connects these two paths.

To obtain the node s

<u>Claim</u>. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

Case 1. The path connecting o_1, o_2 doesn't intersect with the path connecting c, d.

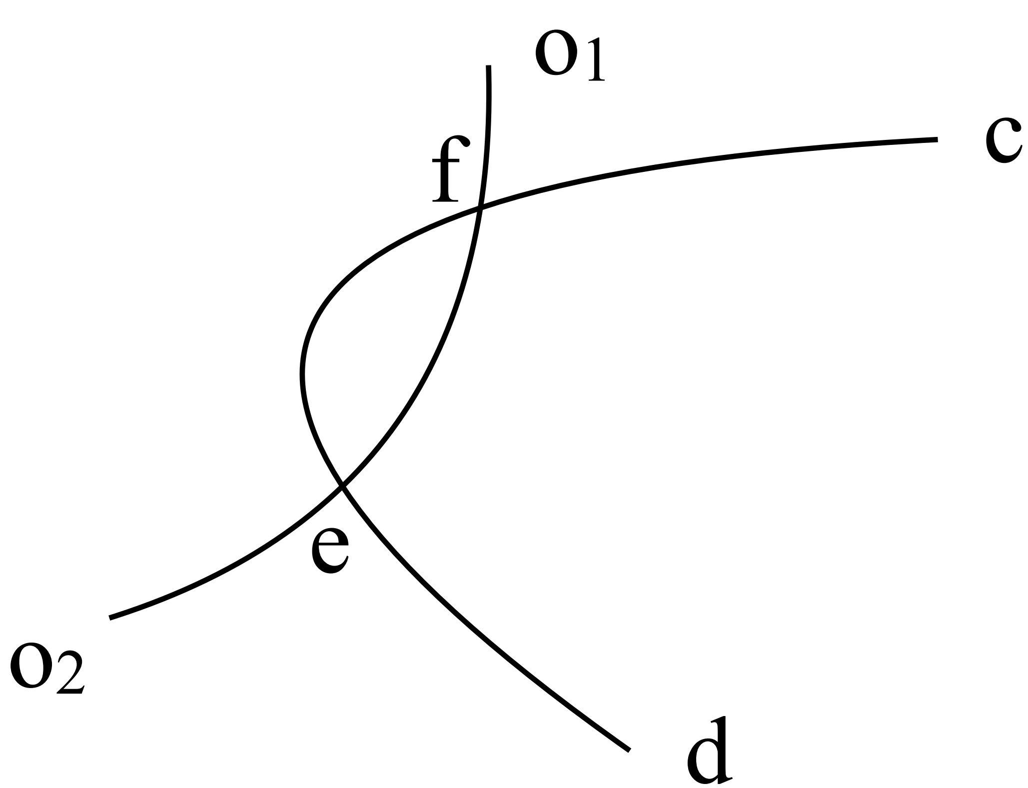


$$
\begin{aligned}
&\mathrm{dis}(o_1, o_2)\\
&\geq \mathrm{dis}(o_1, e)+\mathrm{dis}(e, f)+\mathrm{dis}(f, d)\\
&> \mathrm{dis}(o_1, e)+\mathrm{dis}(f, d)\\
&\Rightarrow \mathrm{dis}(e, o_2) > \mathrm{dis}(f, d)\\
&\Rightarrow \mathrm{dis}(c, o_2) > \mathrm{dis}(c, d) \ \rightarrow\leftarrow
\end{aligned}
$$

To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

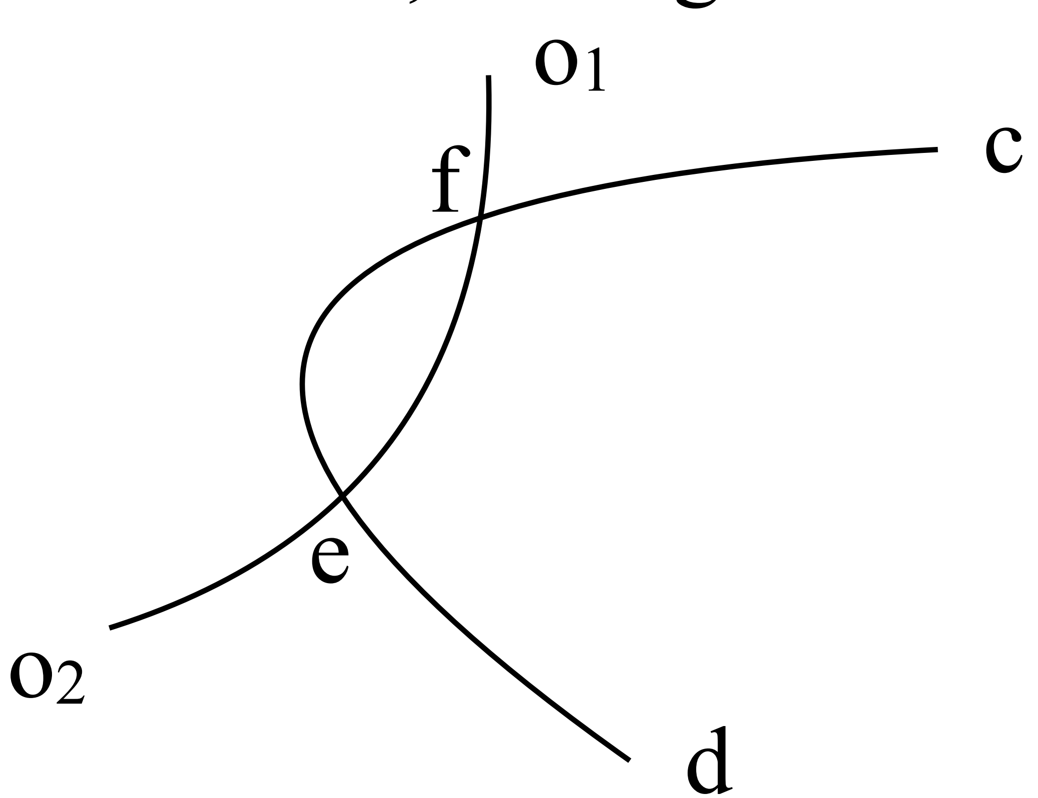
Case 2. The path connecting o_1, o_2 have multiple intersections, say f, e, ..., with the path connecting c, d. In addition, the segment f...e on two paths are different.



To obtain the node s

<u>Claim</u>. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, d = o_1 or d = o_2. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

Case 2. The path connecting o_1, o_2 have multiple intersections, say f, e, ..., with the path connecting c, d. In addition, the segment f...e on two paths are different.

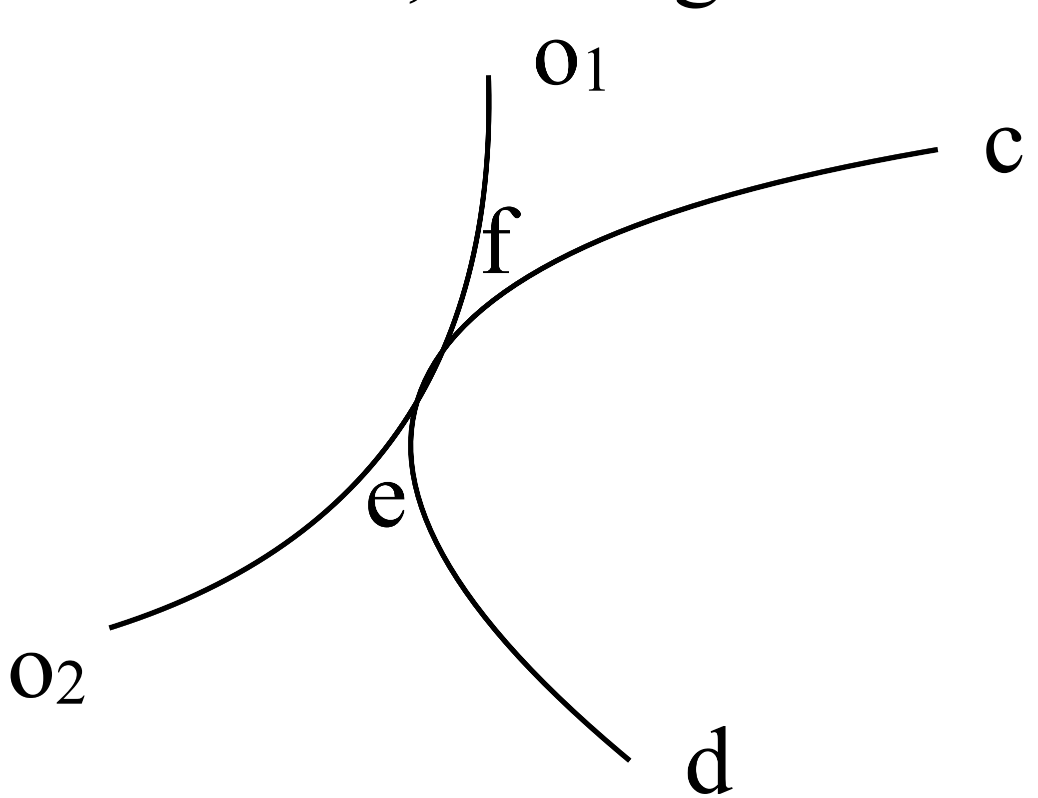


Then we have a cycle, but T is a tree. $\rightarrow\leftarrow$

To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, d = o_1 or d = o_2. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

Case 3. The path connecting o_1, o_2 have multiple intersections, say f, e, ..., with the path connecting c, d. In addition, the segment f...e on two paths are the same.

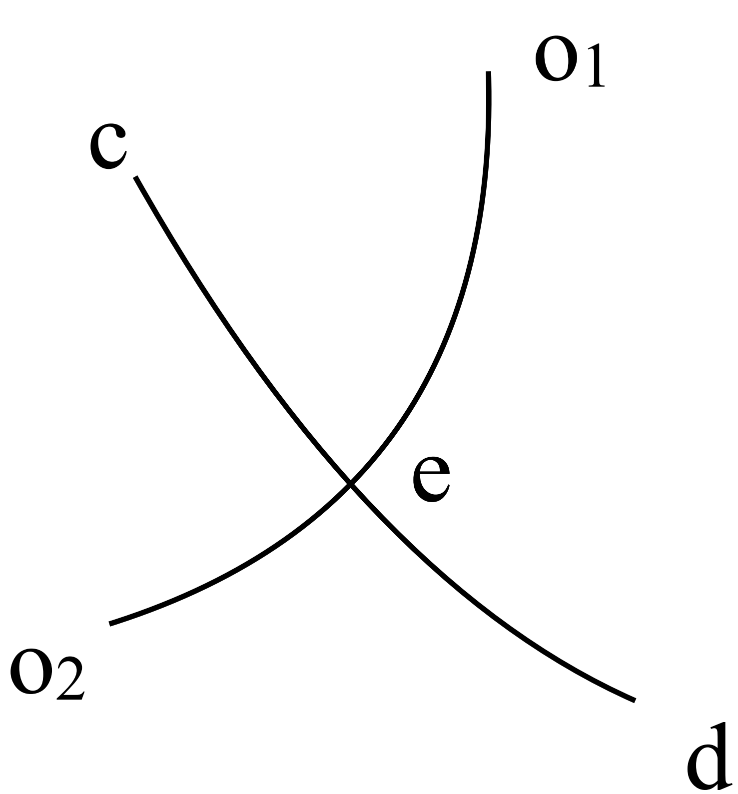


$$\mathrm{dis}(o_1, o_2)$$
$$= \mathrm{dis}(o_1, f)+\mathrm{dis}(f, e)+\mathrm{dis}(e, o_2)$$
$$> \mathrm{dis}(o_1, f)+\mathrm{dis}(f, e)+\mathrm{dis}(e, d)$$
[why not $\geq$]
$$\Rightarrow \mathrm{dis}(e, o_2) > \mathrm{dis}(e, d) \rightarrow\leftarrow$$

To obtain the node s

<u>Claim</u>. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$.
Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

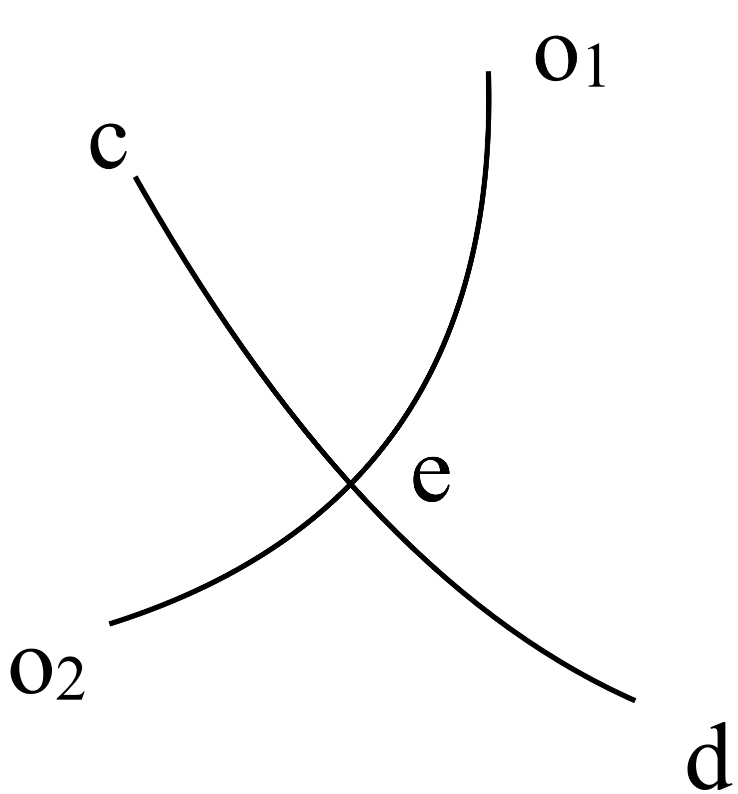
Case 4. The path connecting o_1, o_2 have exactly one intersection, say e, with the path connecting c, d.



To obtain the node s

<u>Claim</u>. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

Case 4. The path connecting o_1, o_2 have exactly one intersection, say e, with the path connecting c, d.



$$
\begin{aligned}
&\mathrm{dis}(o_1, o_2) \\
&= \mathrm{dis}(o_1, e)+\mathrm{dis}(e, o_2) \\
&> \mathrm{dis}(o_1, e)+\mathrm{dis}(e, d) \text{ [why not } \geq \text{]} \\
&\Rightarrow \mathrm{dis}(e, o_2) > \mathrm{dis}(e, d) \rightarrow\leftarrow
\end{aligned}
$$

To obtain the node s

Claim. Let c be an arbitary node. Run BFS(T, c) and record the node d whose dis[d] is largest. Then, $d = o_1$ or $d = o_2$. Note that there might be multiple (o_1, o_2) pairs and what d matches is one node in some (o_1, o_2) pair.

The claim thus holds, implying that computing the diameter of a tree can be solved by running BFS twice. The total runtime is $O(n+m) = O(n)$.

DFS-Visit

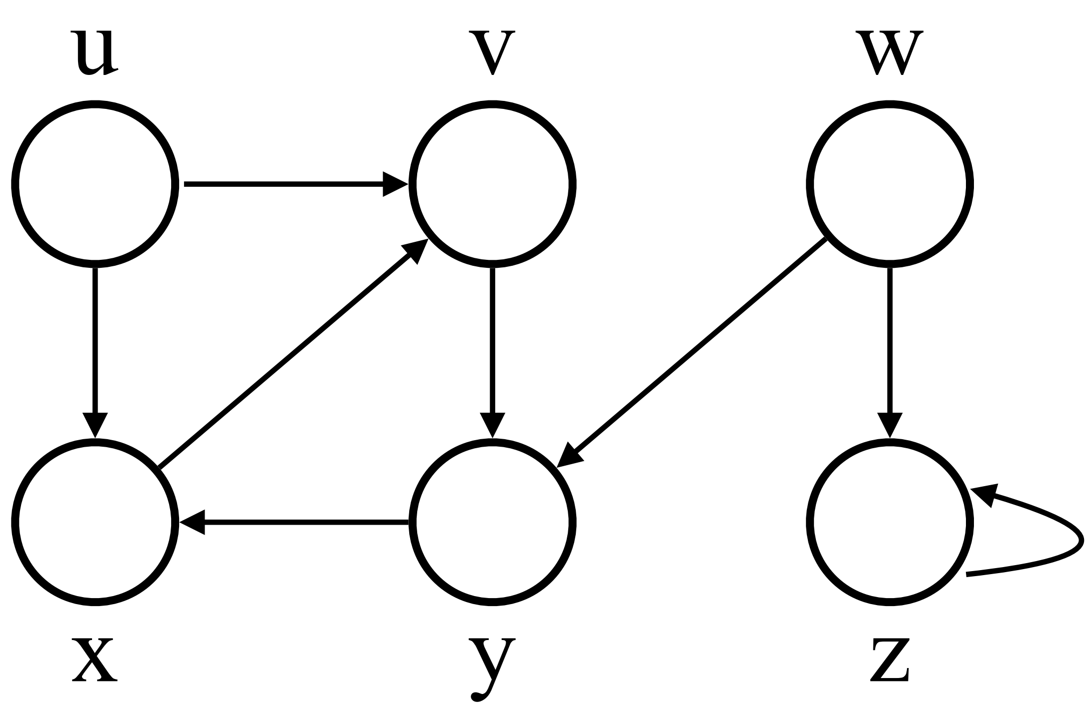
DFS-Visit(G, s)

DFS-Visit(G, s) is a building block of DFS(G).

In DFS-Visit(G, s), we are given a graph G and a node s, the purpose is to explore the edges of G to discover the nodes that are reachable from s.

DFS-Visit will explore an unexplored node from the latest discovered node.

DFS-Visit(G, s)



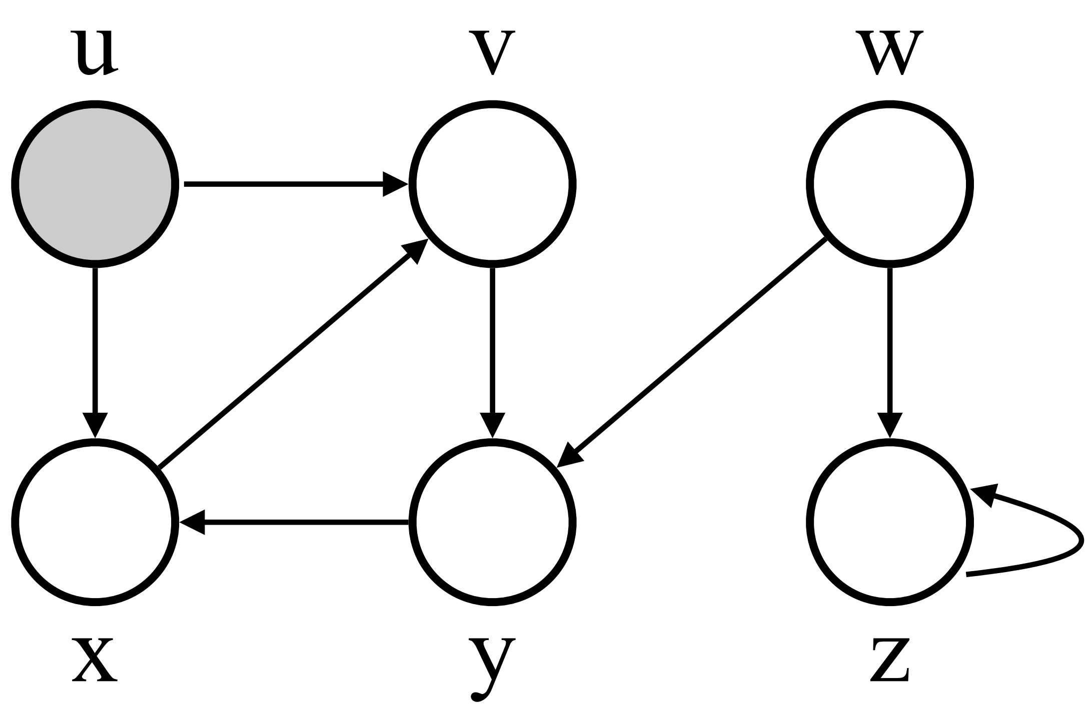
Call DFS-Visit(G, u).

Recall that we will explore the edges from the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



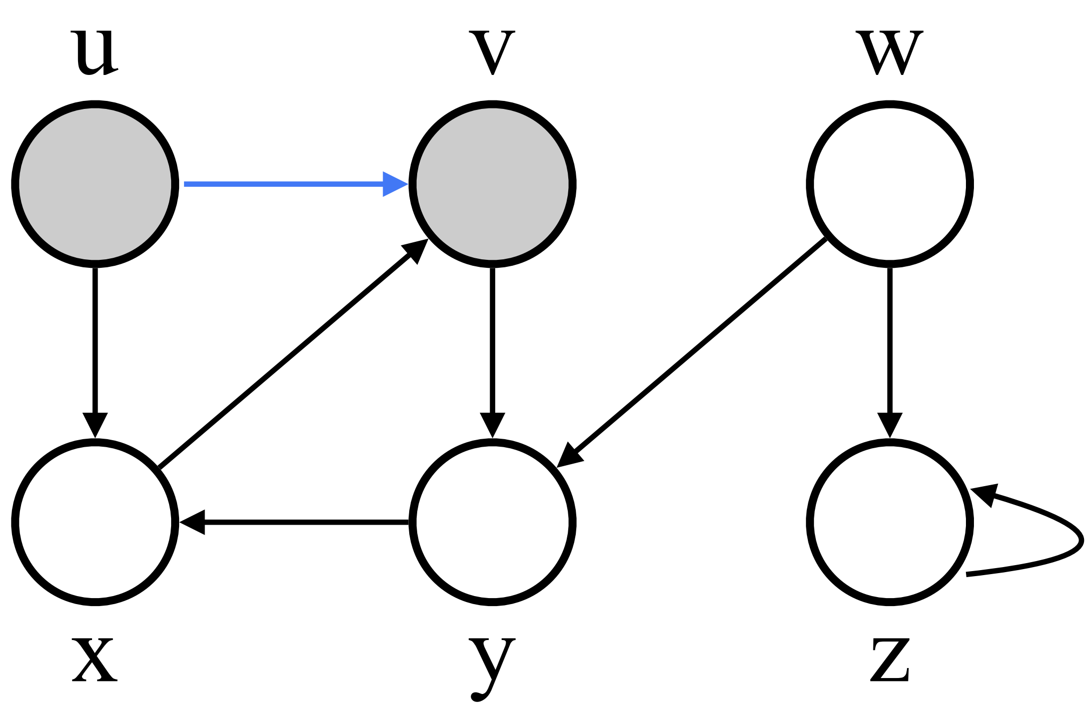
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



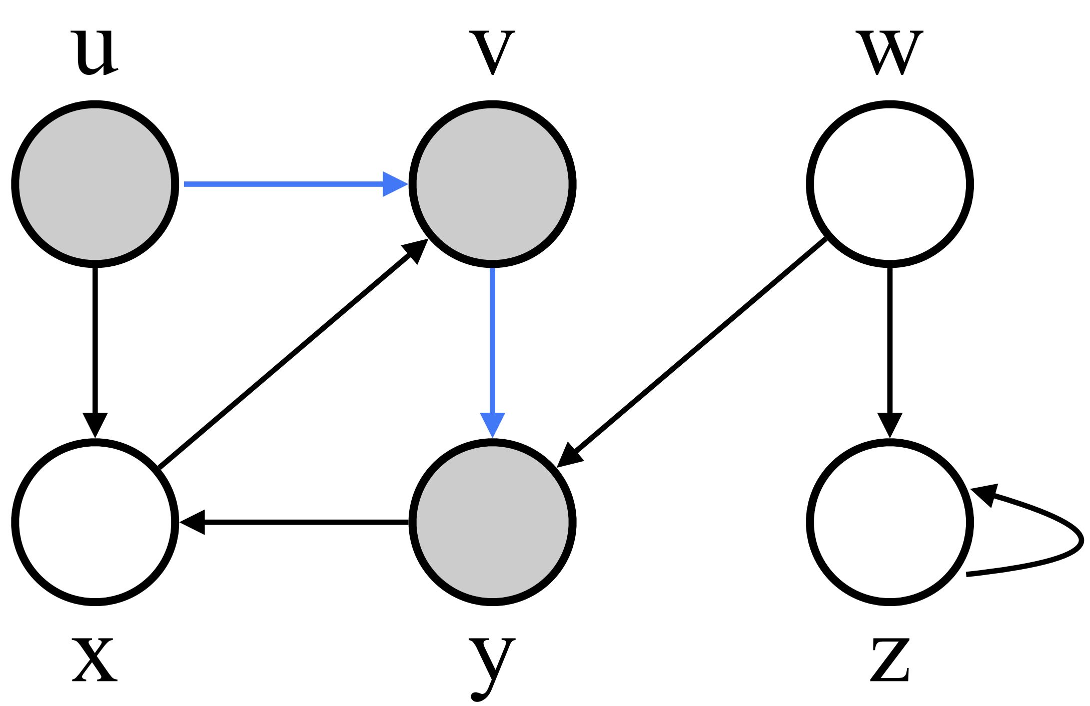
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

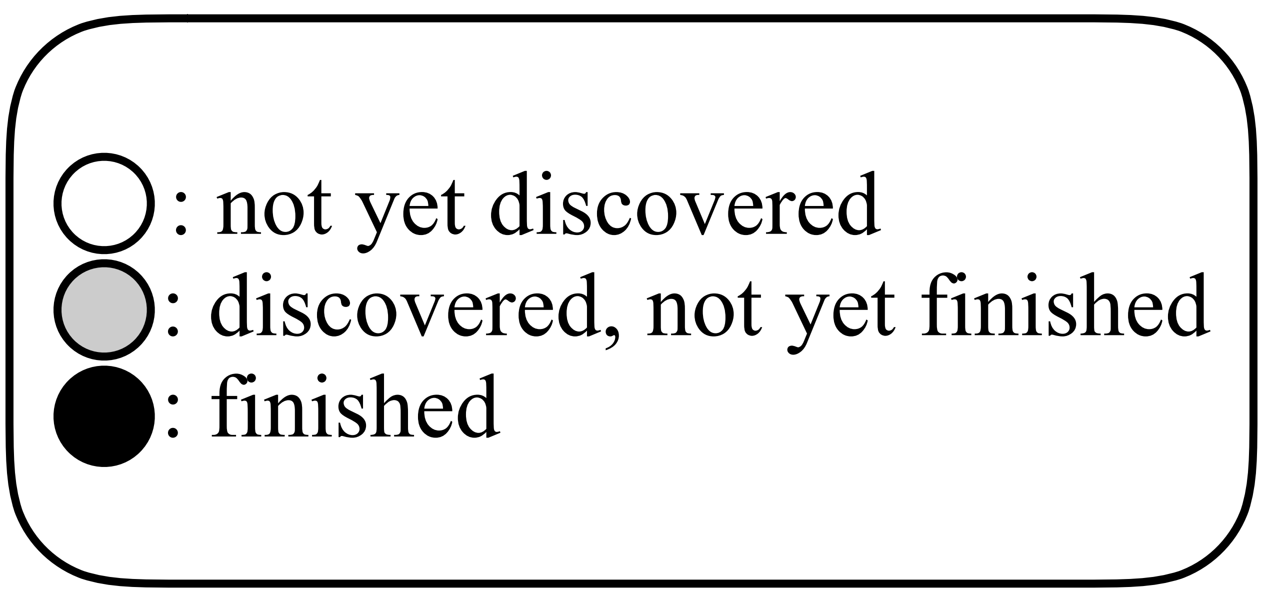
: edges in G
: edges in the DFS-tree

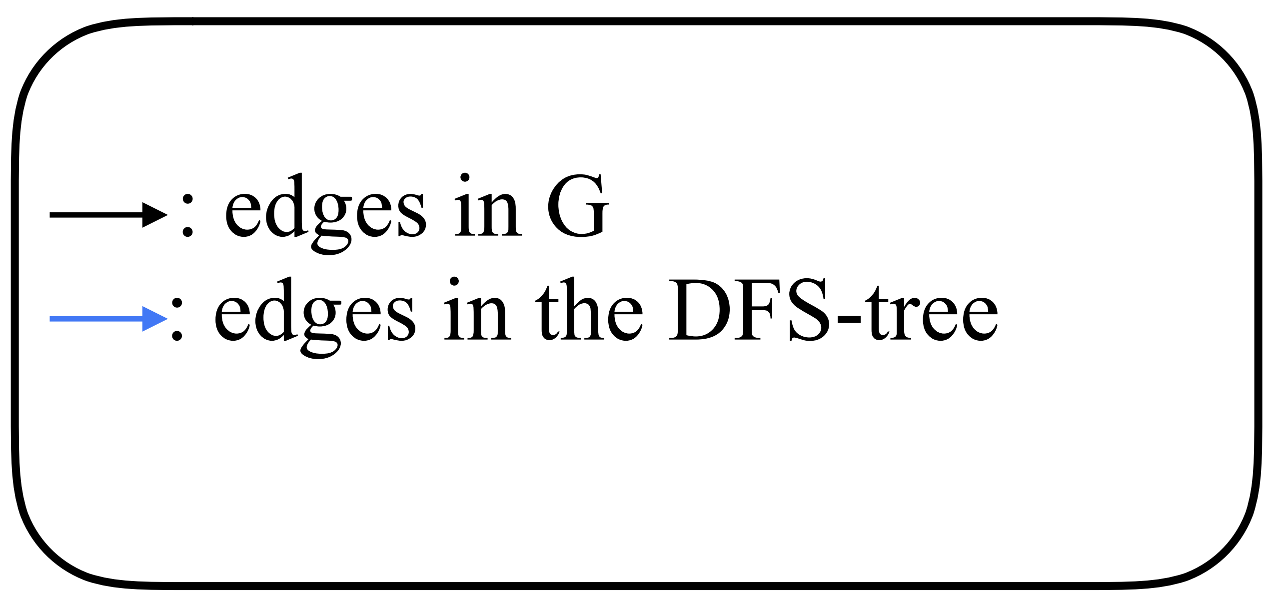
DFS-Visit(G, s)



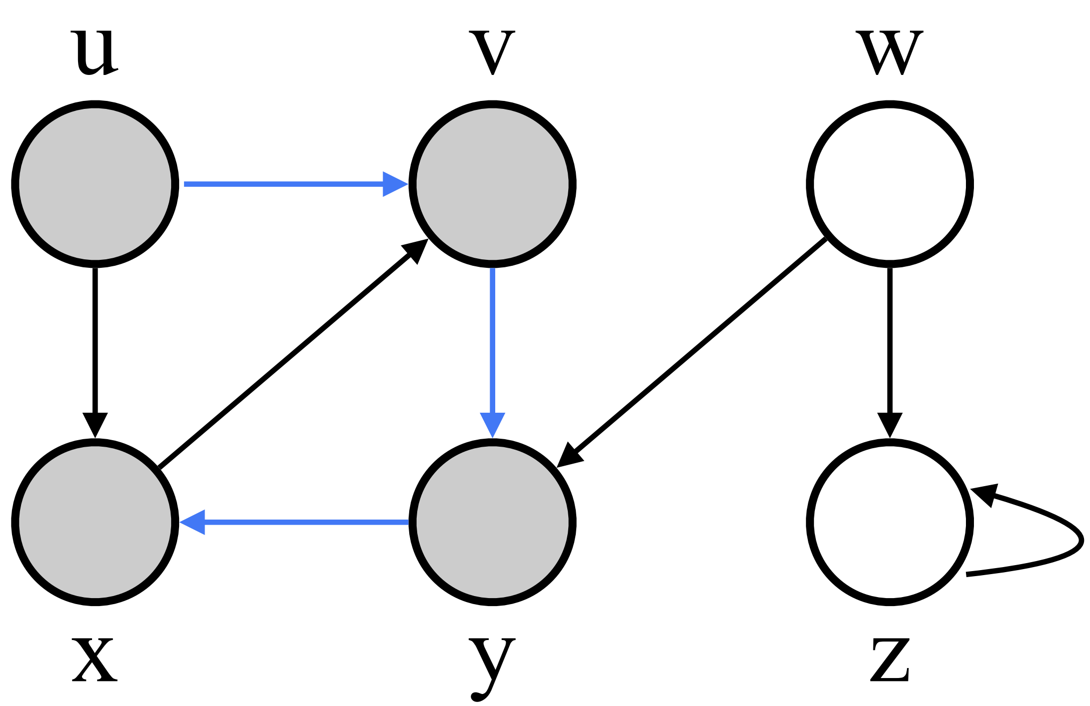
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.





DFS-Visit(G, s)



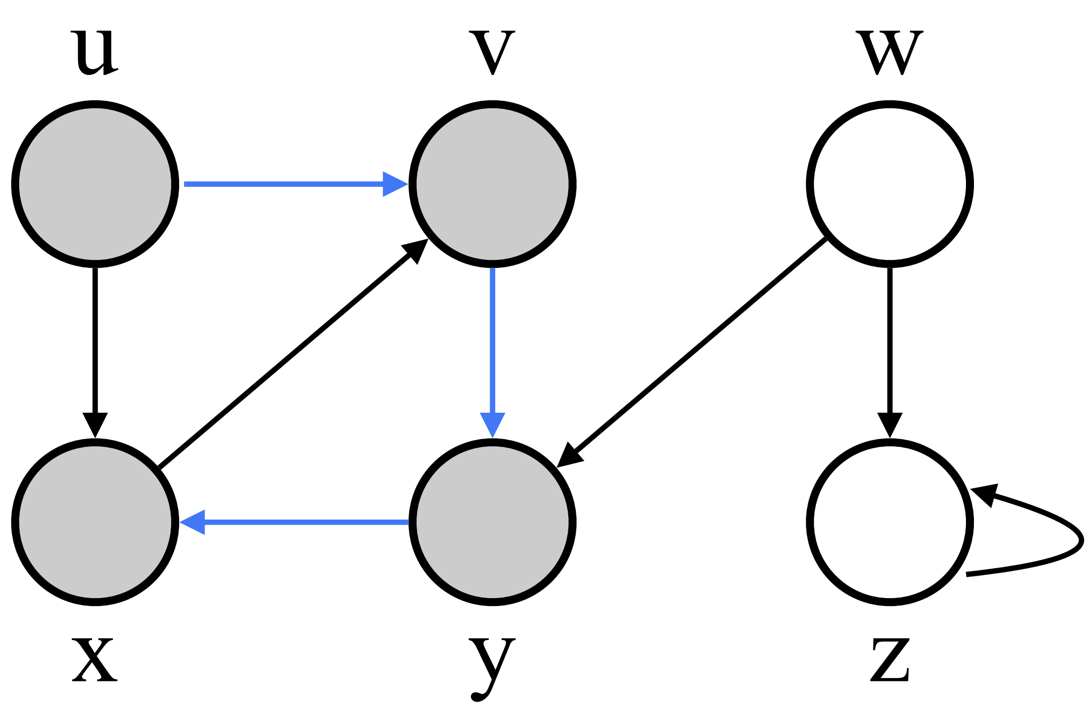
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



Call DFS-Visit(G, u).

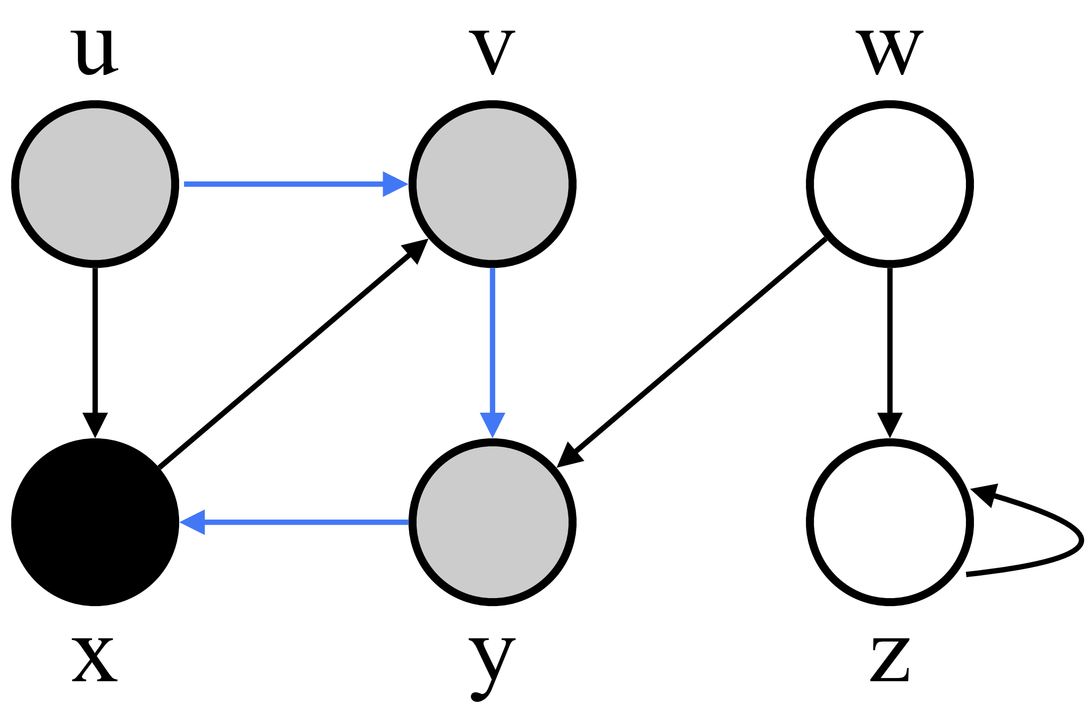
Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

We cannot visit more not-yet-discovered nodes from x. So x is finished.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



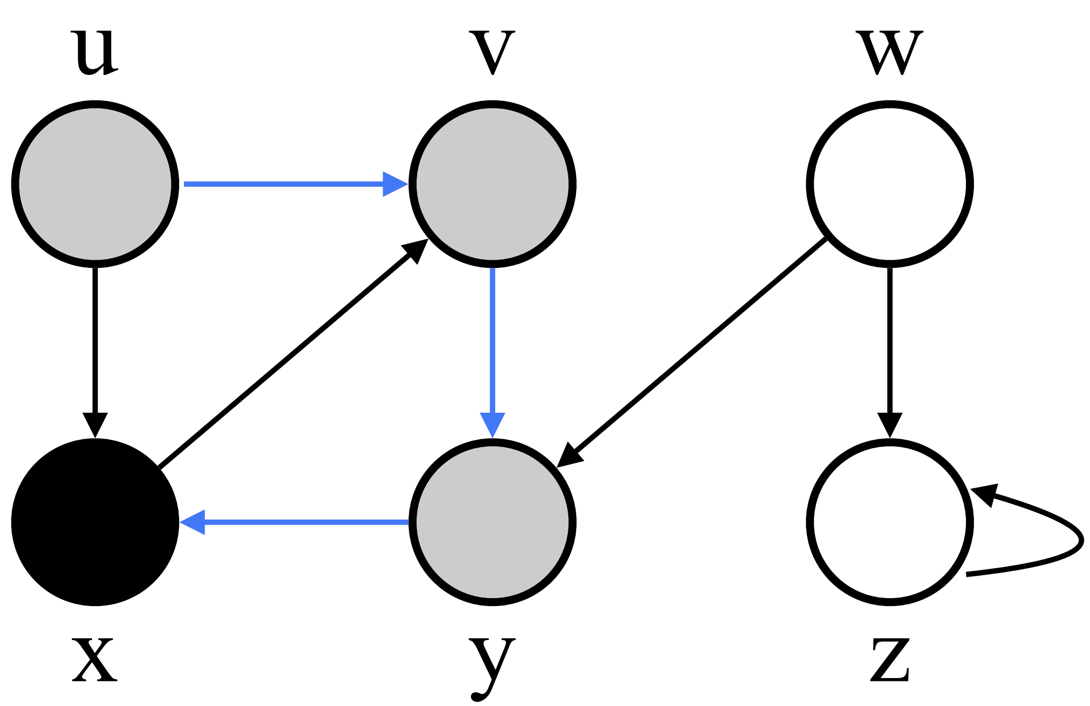
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



Call DFS-Visit(G, u).

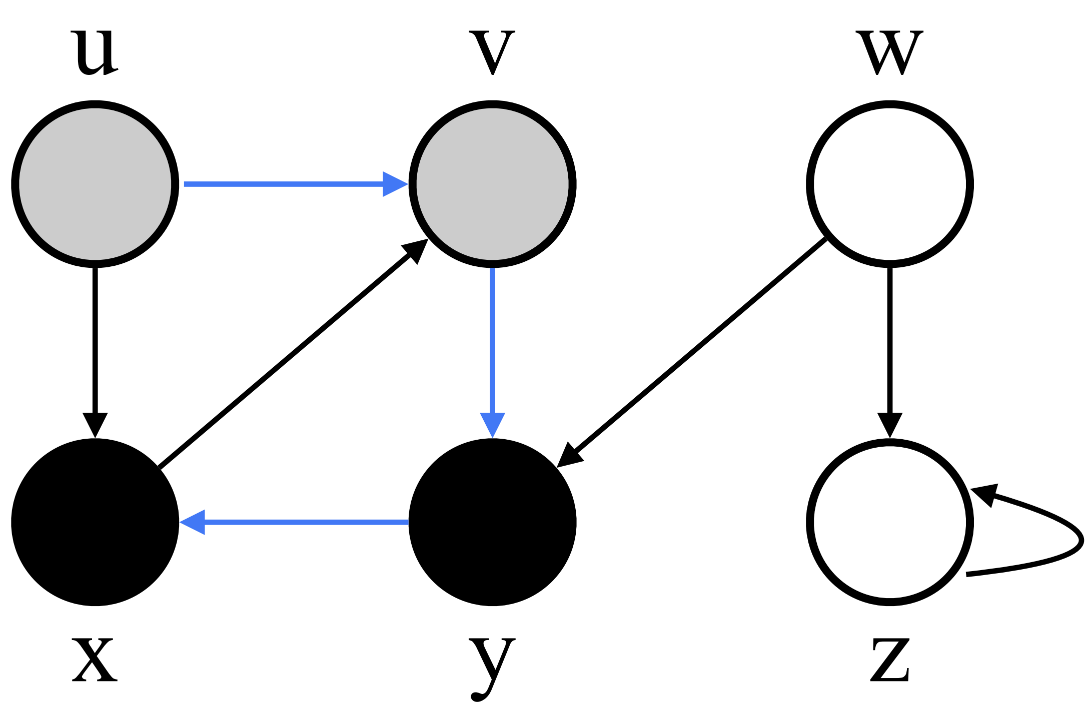
Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

We cannot visit more not-yet-discovered nodes from y. So y is finished.

: not yet discovered
: discovered, not yet finished
: finished

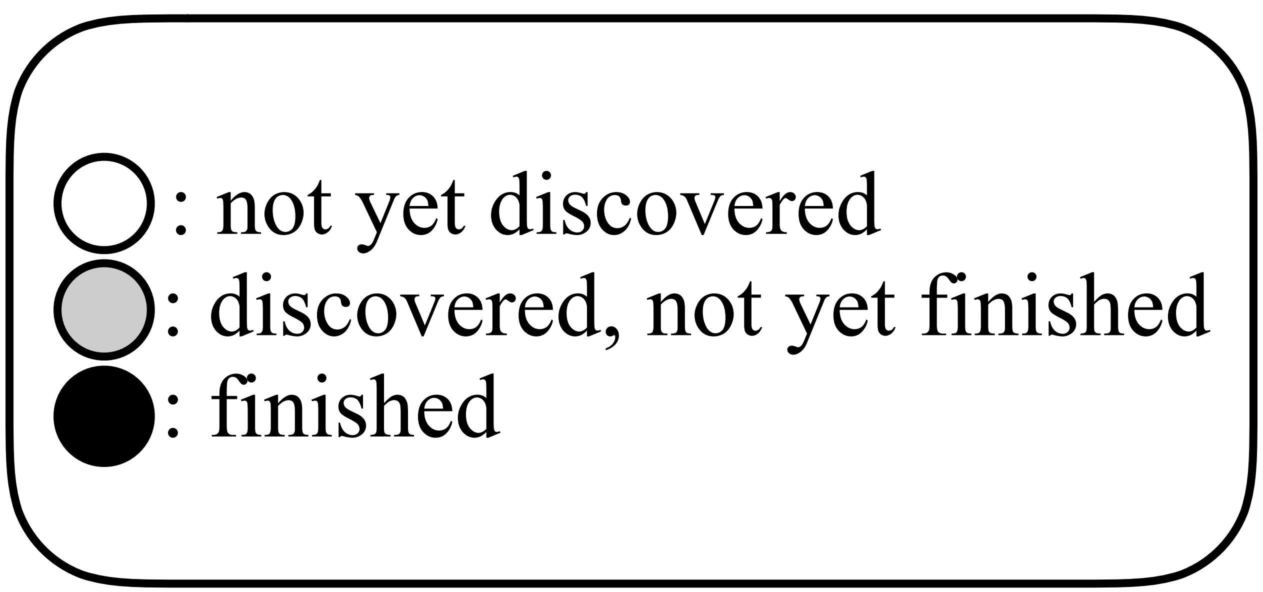
: edges in G
: edges in the DFS-tree

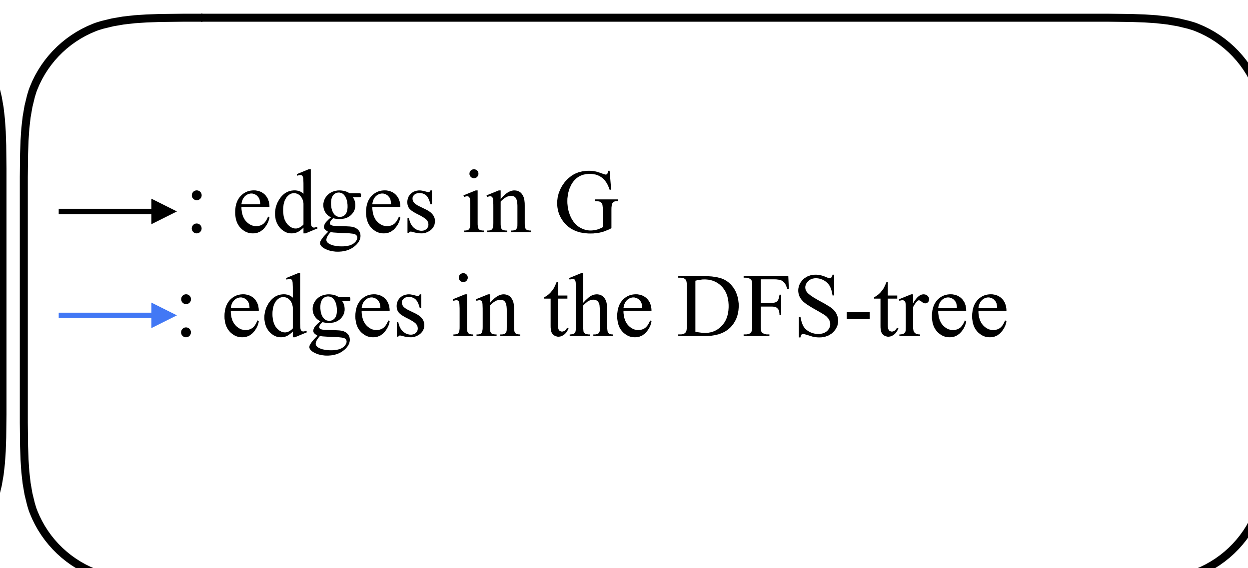
DFS-Visit(G, s)



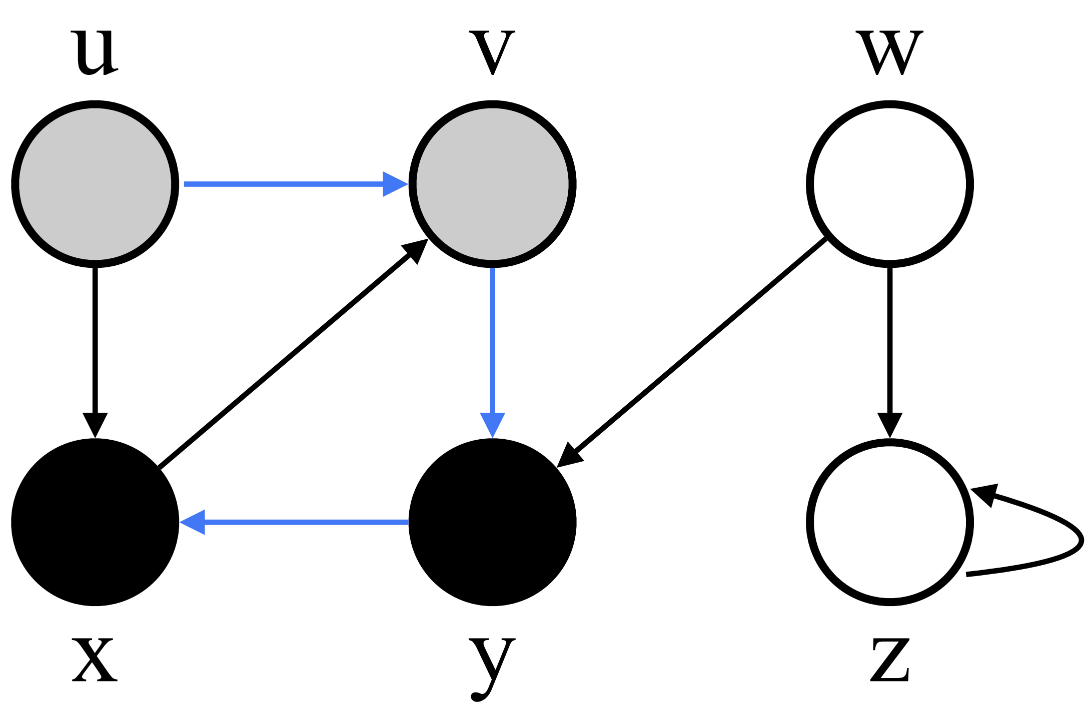
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.





DFS-Visit(G, s)



Call DFS-Visit(G, u).

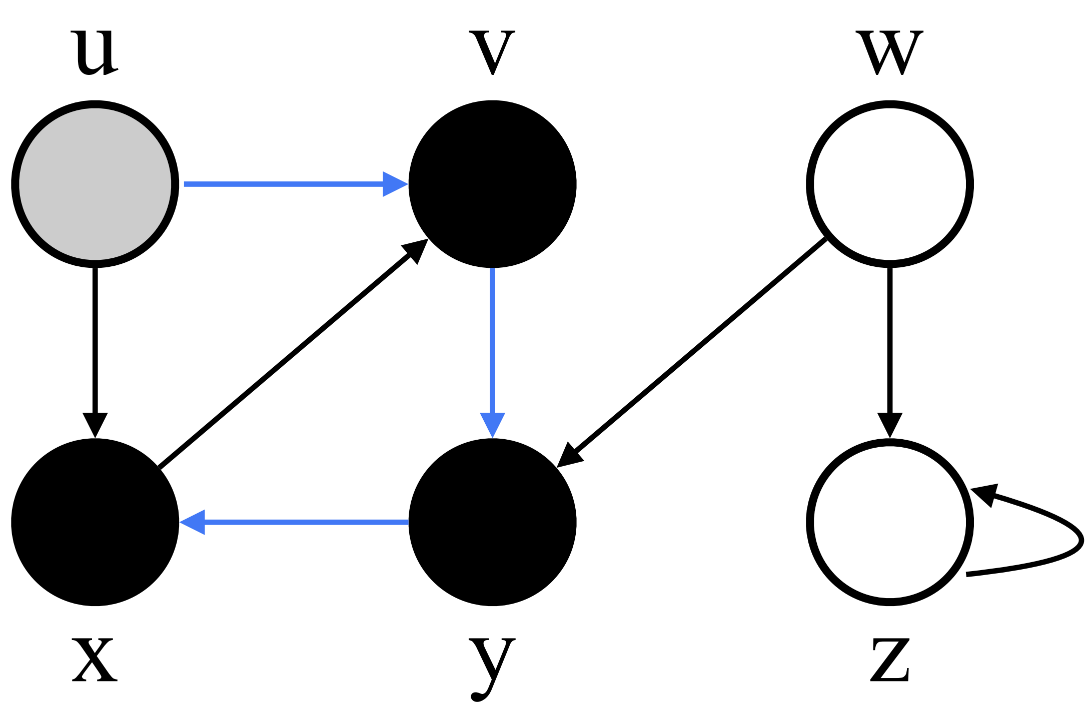
Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

We cannot visit more not-yet-discovered nodes from v. So v is finished.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



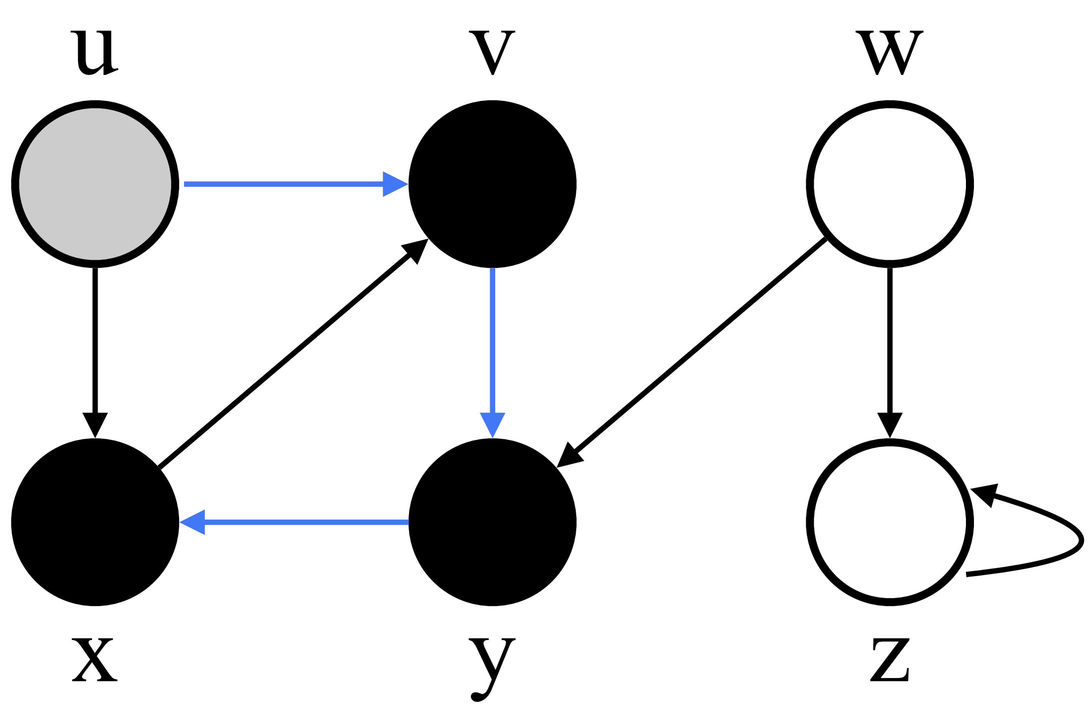
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



Call DFS-Visit(G, u).

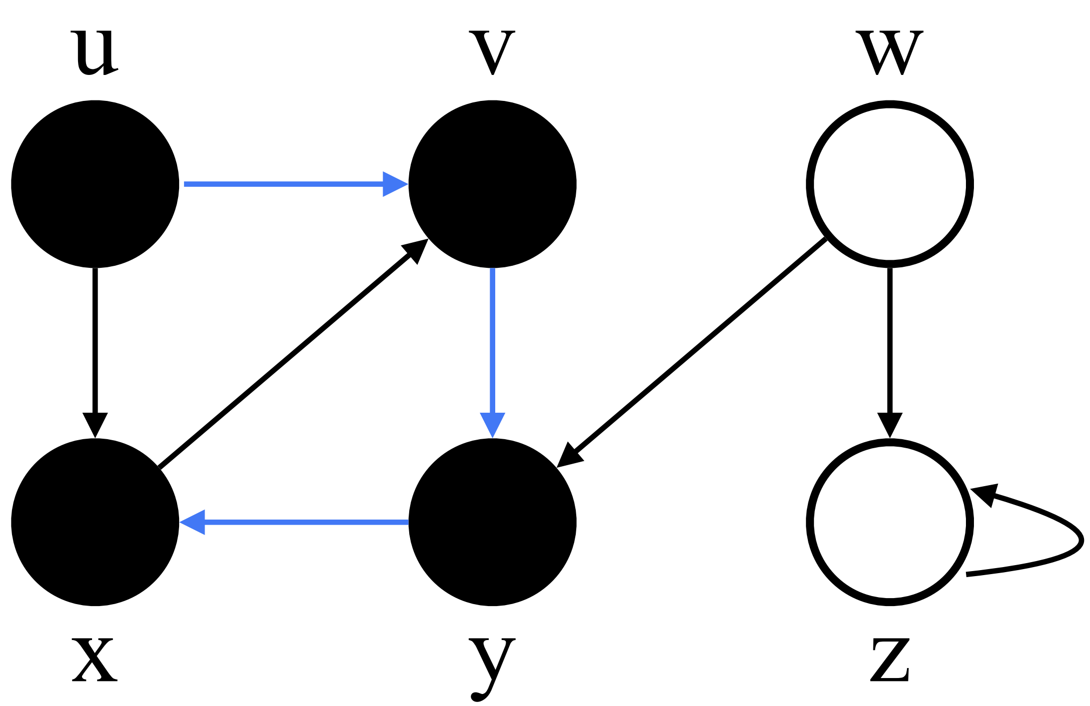
Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

We cannot visit more not-yet-discovered nodes from u. So u is finished.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)



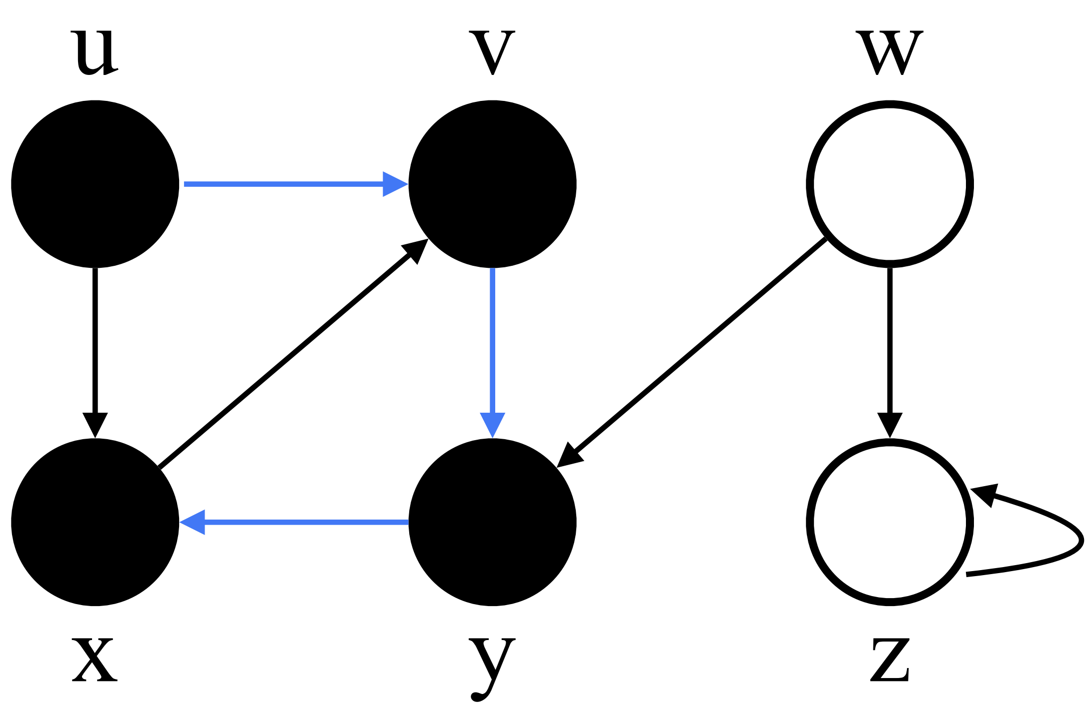
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

: not yet discovered
: discovered, not yet finished
: finished

: edges in G
: edges in the DFS-tree

DFS-Visit(G, s)

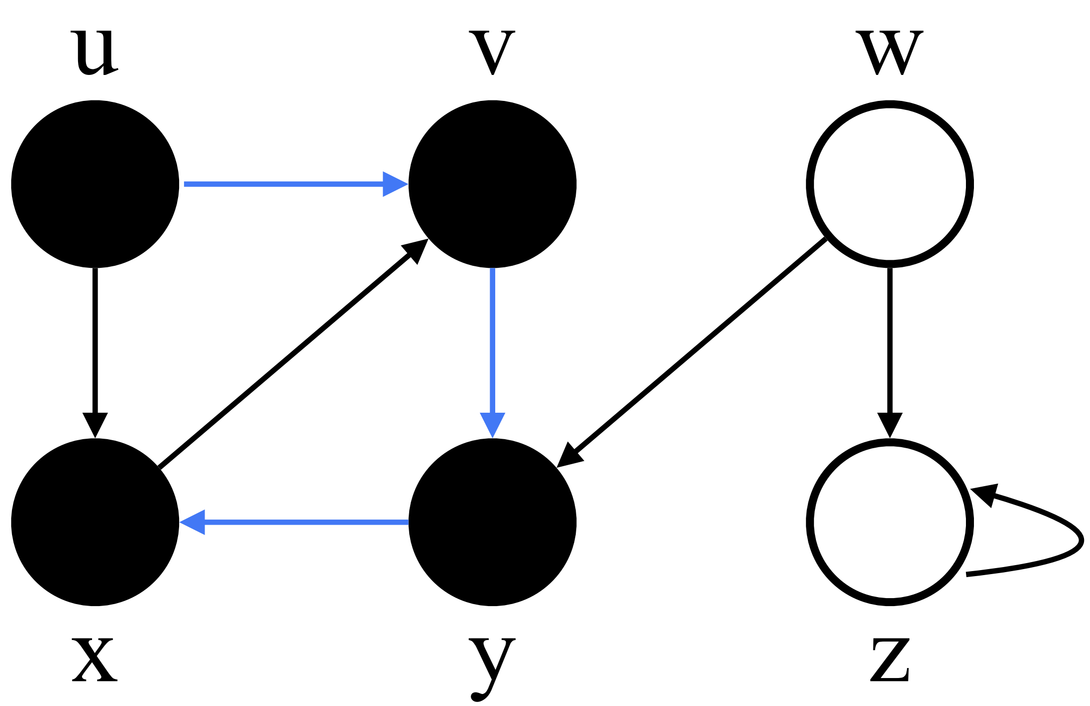


Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

Once the starting node u is finished, all the nodes that are reachable from u have been discovered.

DFS-Visit(G, s)



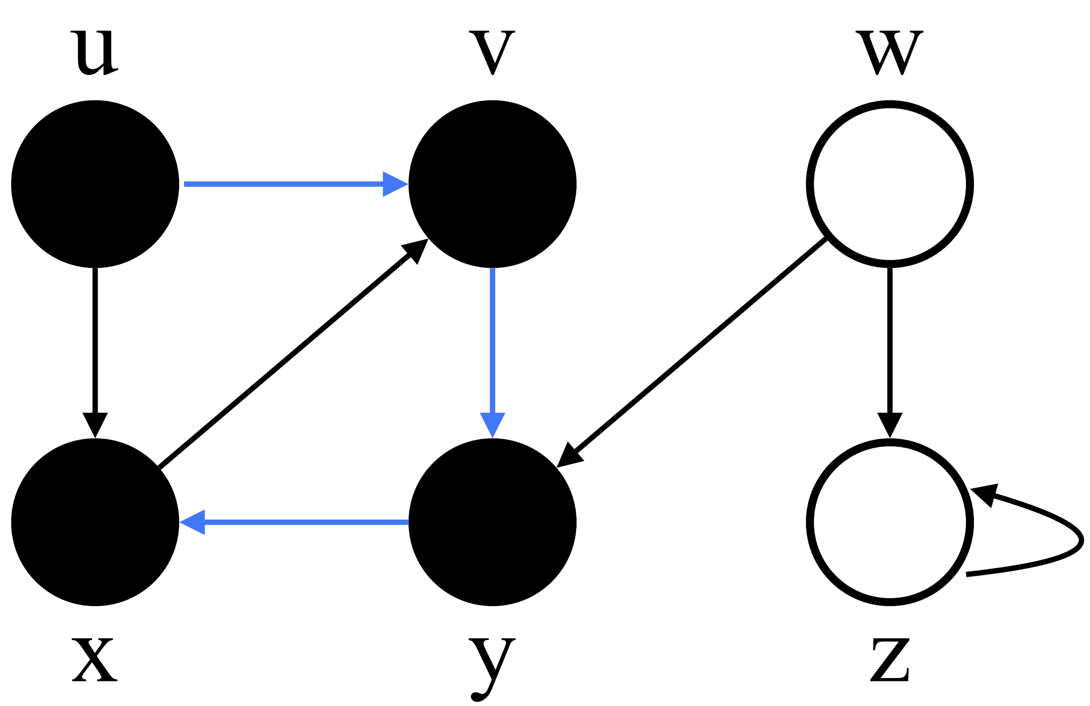
Call DFS-Visit(G, u).

Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

Once the starting node u is finished, all the nodes that are reachable from u have been discovered.

v is finished before u. After v is finished, are the nodes reachable from v **all** discovered?

DFS-Visit(G, s)



Call DFS-Visit(G, u).

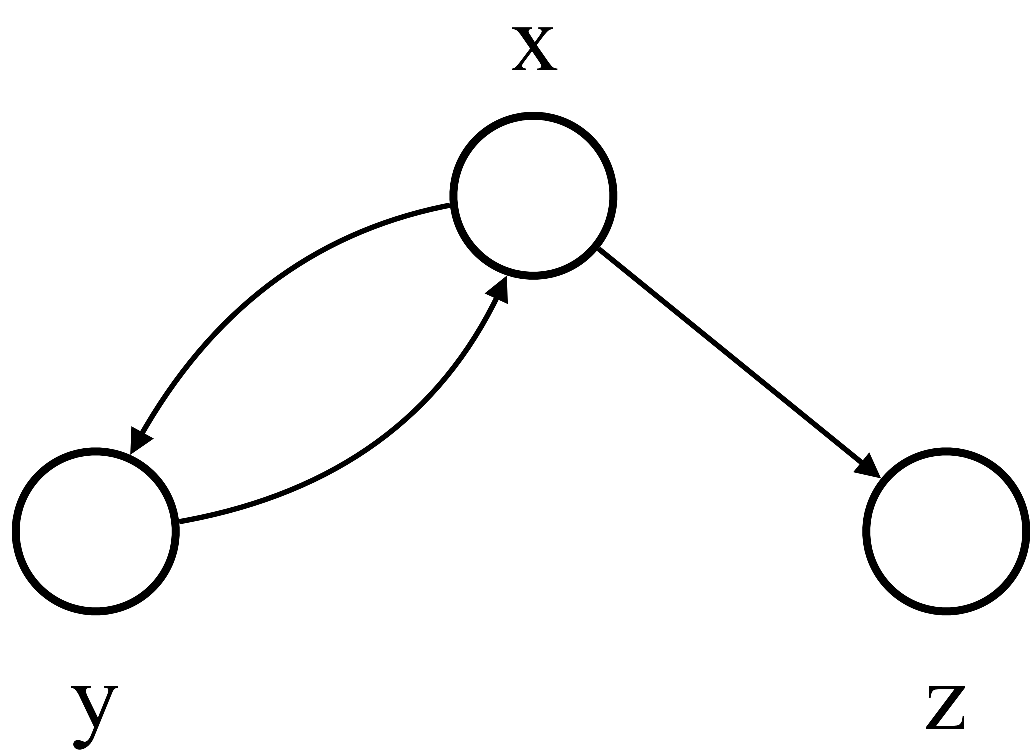
Recall that we will explore the edges of the lastest discovered node to visit not-yet-discovered nodes.

Once the starting node u is finished, all the nodes that are reachable from u have been discovered.

v is finished before u. After v is finished, are the nodes reachable from v **all** discovered?

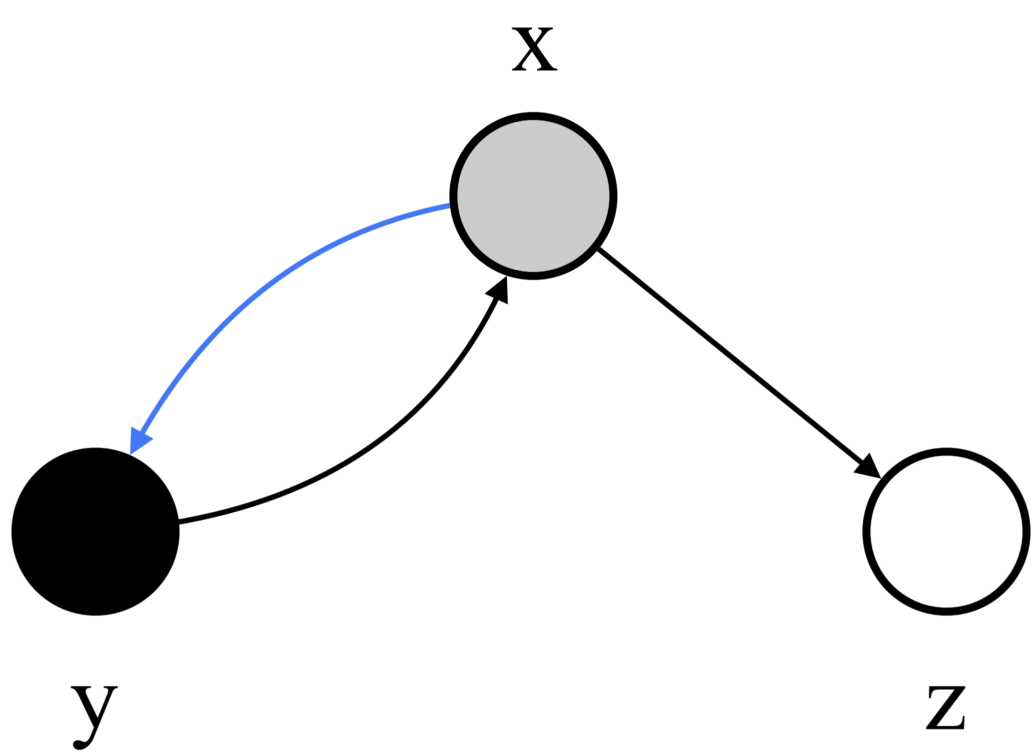
No.

For an arbitrary node, the claim may be false



The nodes reachable from y are x and z.

For an arbitrary node, the claim may be false



The nodes reachable from y are x and z.

If we call DFS-Visit(G, x), at some time step we obtain the above graph.

For node y, when y is finished, it doesn't imply that all the nodes reachable from y have been discovered.

For the starting node, the claim is true

Proof

Let s be the set of starting node, and U be the set of nodes that are reachable from s but not yet discovered after s is finished.

Let D be the set of nodes discovered by calling DFS-Visit(G, s).

There exists an edge (p, q) between D and U; otherwise, there exists no path from s to U. Why don't we visit q before p is finished? →←

Why doesn't the above proof work for an arbitrary node?

Pseudocode of DFS-Visit(G, s)

The initial call is DFS-Visit(G, s) // initially, all the nodes have color white

```
DFS-Visit(G, u){
    u.color ← Gray;
    foreach (node v in Adj[u]){
        if(v.color equals White){ // v hasn't yet been discovered
            v.parent ← u;
            DFS-Visit(G, v); // explore the latest discovered node
        }
        // then explore the current node
    }
    u.color ← Black;
    return;
}
```

Depth First Search

DFS(G)

DFS(G) has a more important task than simply identifying the node set that are reachable from some starting node.

Indeed, DFS(G) may invoke DFS-Visit(G, s_i) for multiple starting nodes s_1, s_2, ..., s_t, where s_2 is an arbitrary node in G that are not reachable from s_1, and more generally s_i is an arbitrary node in G that are not reachable from s_1, s_2, ..., s_{i-1}.

In other words, DFS(G) will explore the structure of the entire graph, while DFS-Visit(G, s) may not.

In addition, we will timestamp each node when it is discovered and when it is finised.

Pseudocode of DFS(G) and its Building Block DFS-Visit(G, s) (timestamped)

```
DFS(G){
    foreach (node u in G){
        u.color ← White;
        u.parent ← NIL;
    }
    time ← 0;
    foreach (node u in G){
        if(u.color equals White){
            DFS-Visit(G, u);
        }
    }
}
```

Pseudocode of DFS(G) and its Building Block DFS-Visit(G, s) (timestamped)

```
DFS-Visit(G, u){
    time ← time + 1; u.d ← time; // discovery time
    u.color ← Gray;
    foreach (node v in Adj[u]){
        if(v.color equals White){ // v hasn't yet been discovered
            v.parent ← u;
            DFS-Visit(G, v); // explore the latest discovered node
        }
        // then explore the current node
    }
    u.color ← Black;
    time ← time + 1; u.f ← time; // finishing time
}
```

We will see that these timestamps are useful.

Parenthesis Theorem

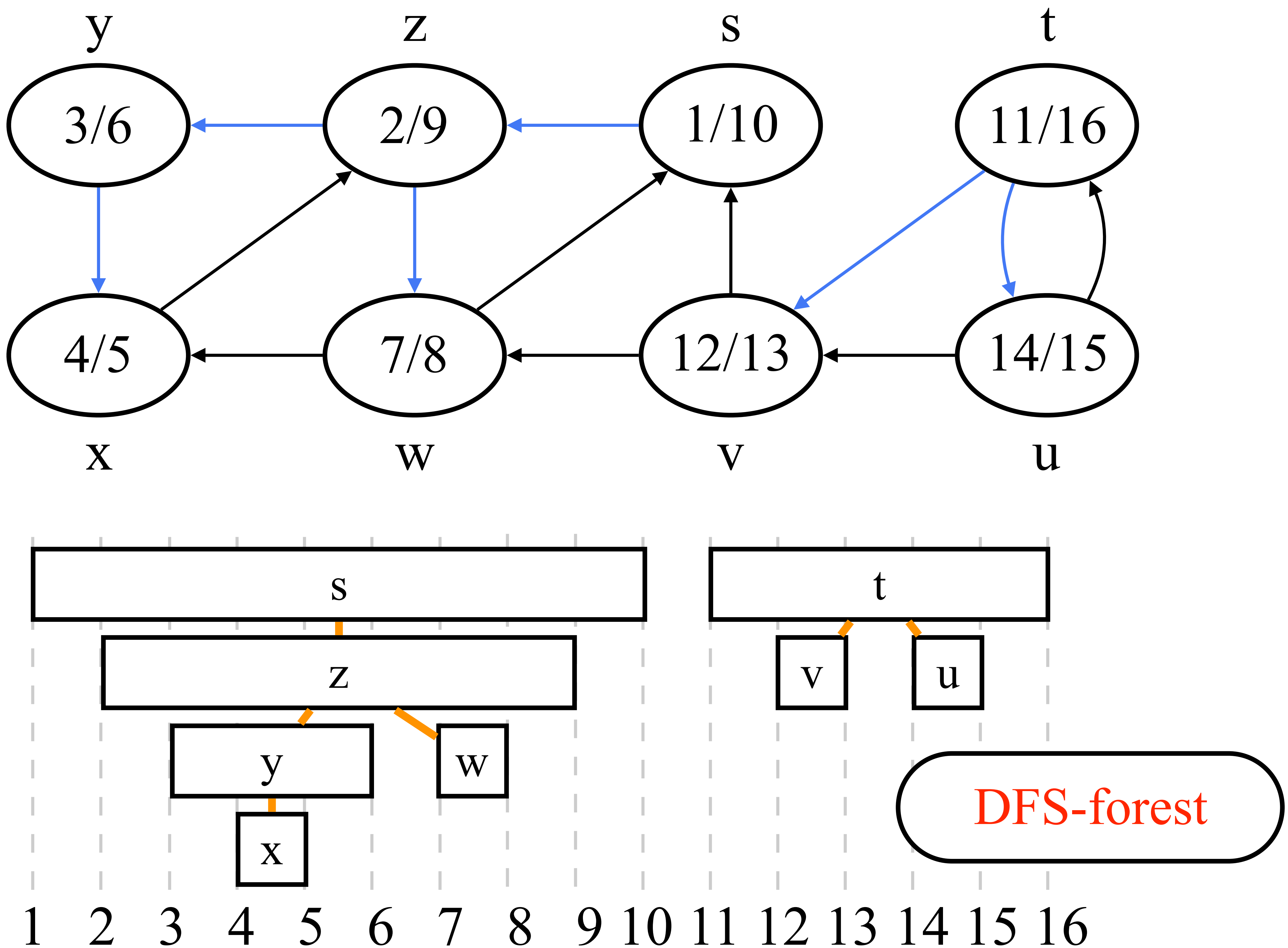
For any two nodes u and v, exactly one of the following three conditions holds:

(1) the intervals [u.d, u.f] and [v.d, v.f] are disjoint
// imply that neither u nor v is a descendant of the other in the DFS-forest (why forest?)

(2) the interval [v.d, v.f] contains the interval [u.d, u.f]
// imply that v is an ancestor of u in the DFS-forest

(3) the interval [u.d, u.f] contains the interval [v.d, v.f]
// imply that u is an ancestor of v in the DFS-forest

Parenthesis Theorem



Classification of edges

There are four types of edges w.r.t. a DFS-forest F

(1) **tree edges**: the edges (u, v) in F
// v was discovered while v.color equals White

(2) **back edges**: the edges (u, v) connecting a node u to an ancestor v in F // self-loops are back edges

(3) **forward edges**: the non-tree edges (u, v) connecting a node u to a descendant v in F

(4) **cross edges**: all the other edges. // edges between two disjoint subtrees in a tree or two disjoint trees in the forest

Classification of edges

Identifying the type of an edge by colors and timestamps.

(1) **tree edges**: while exploring (u, v), v.color equals White

(2) **back edges**: while exploring (u, v), v.color equals Gray

(3) **forward edges**: while exploring (u, v), v.color equals Black and u.d < v.d

(4) **cross edges**: while exploring (u, v), v.color equals Black and u.d > v.d

Classification of edges

Identifying the type of an edge by colors and timestamps.

(1) **tree edges**: while exploring (u, v), v.color equals White

(2) **back edges**: while exploring (u, v), v.color equals Gray

(3) **forward edges**: while exploring (u, v), v.color equals Black and u.d < v.d

(4) **cross edges**: while exploring (u, v), v.color equals Black and u.d > v.d

For an undirected graph, it is ambiguous to classify the edges because (u, v) = (v, u). Thus, we classify the edges as the first type in the classification list that applies.

Exercise

Show that for an undirected graph G, any DFS-forest of G (note that DFS forest may not be unique) has no forward edge and cross edge.

(Proof can be found on pp. 610 in I2A)

Exercise

Show that for an undirected graph G, any DFS-forest of G (note that DFS forest may not be unique) has no forward edge and cross edge.

(Proof can be found on pp. 610 in I2A)

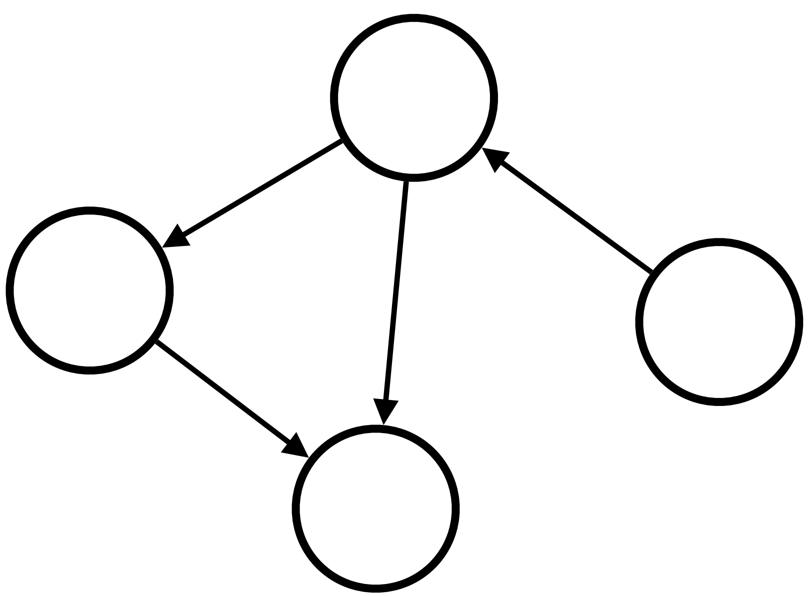
Is there any graph that has a unique DFS-forest?

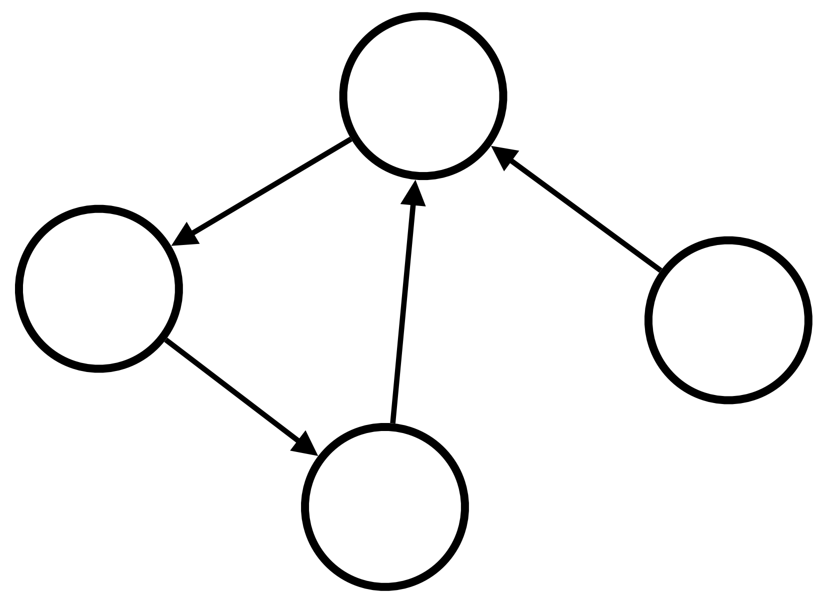
Topological Sort

Directed acyclic graph

If a directed graph G has no directed cycle, then we say G is a directed acyclic graph (DAG).

<u>Example</u>.



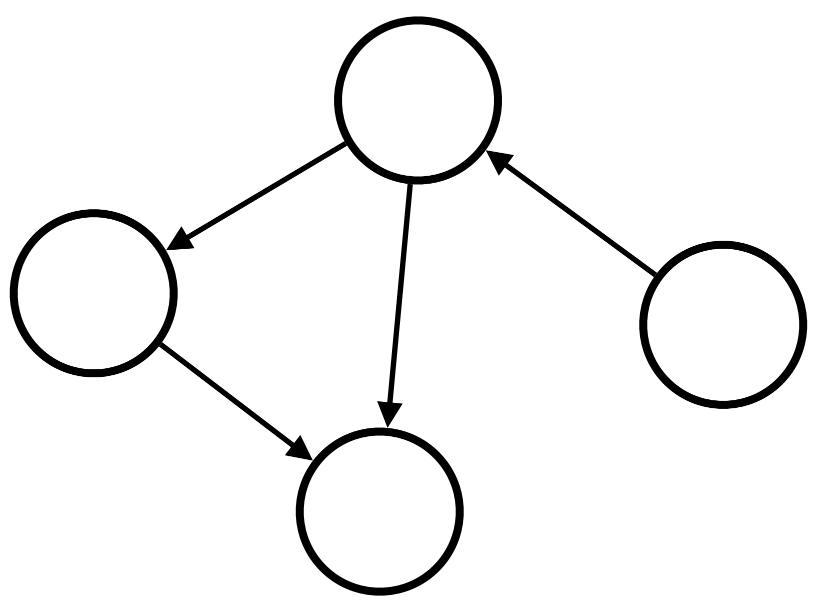


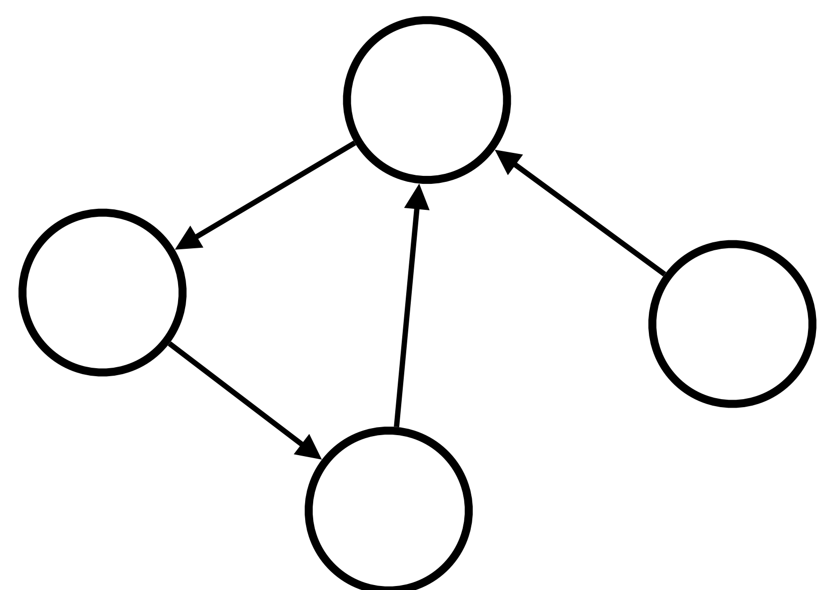
Which one is a DAG?

Directed acyclic graph

If a directed graph G has no directed cycle, then we say G is a directed acyclic graph (DAG).

<u>Example</u>.





Which one is a DAG?

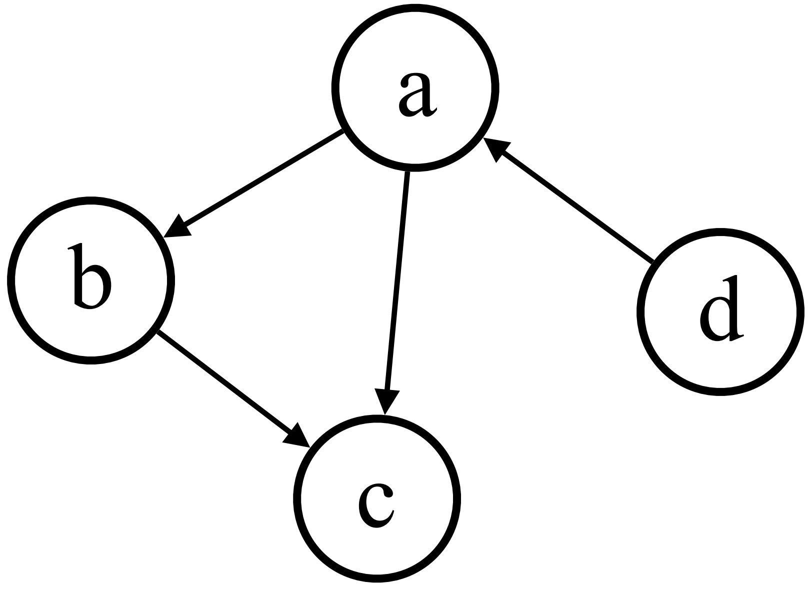
The left one.

Definition of Topological sort

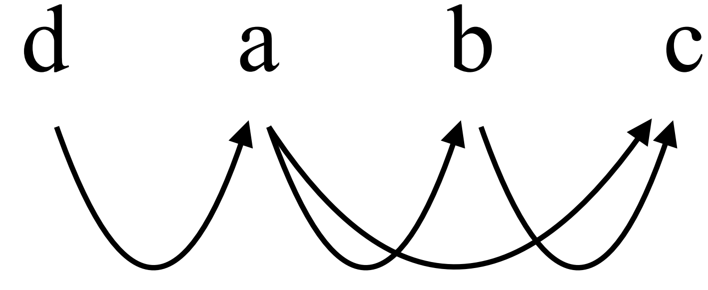
Input: a directed acyclic graph G

Output: an ordering of nodes so that for each edge (u, v) in G, node u appears earlier than node v in the ordering

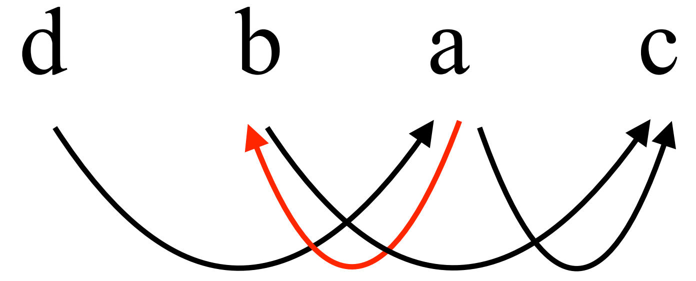
<u>Example</u>.



A feasible node ordering:



An infeasible node ordering:

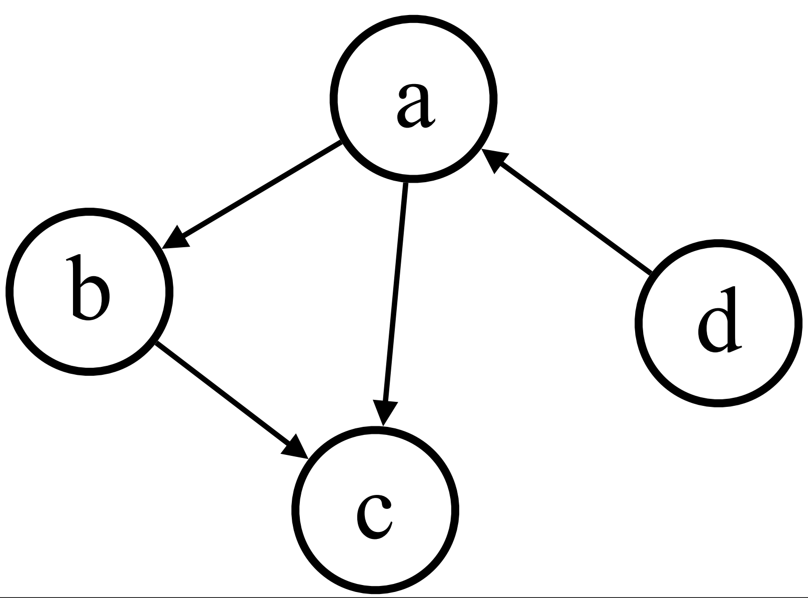


Definition of Topological sort

Input: a directed acyclic graph G

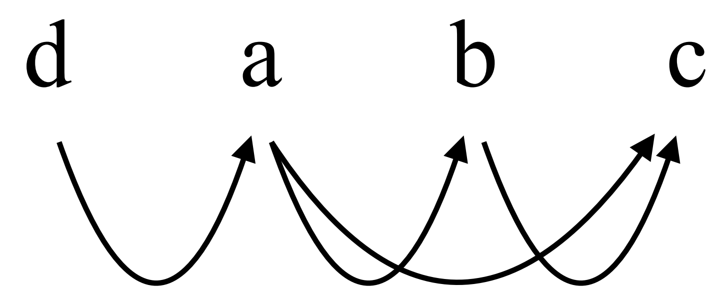
Output: an ordering of nodes so that for each edge (u, v) in G, node u appears earlier than node v in the ordering

Example.

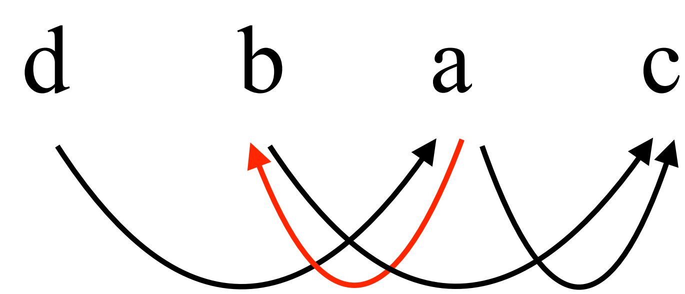


Is the node ordering unique?

A feasible node ordering:



An infeasible node ordering:



T-sort a DAG by DFS

T-sort-DFS(G){
  L $\leftarrow \varnothing$; // L will be updated and eventually becomes a list of nodes representing the T-sorted node ordering
  DFS(G);
  each time node v_i is finished, insert v_i onto the front of L;
  return L;
}

T-sort a DAG by DFS

```
T-sort-DFS(G){
    L ← ∅; // L will be updated and eventually becomes a list of nodes representing the T-sorted node ordering
    DFS(G);
    each time node v_i is finished, insert v_i onto the front of L;
    return L;
}
```

Note that the nodes on L are ordered by their finishing time. If a node has an earlier finishing time, then it appears later on L.

Correctness

It suffices to show that for every edge (u, v) in G, $v.f < u.f$.

Proof.
If (u, v) is a tree edge, then u is an ancestor of v, implying that $v.f < u.f$.

If (u, v) is a back edge, then the path from u to v plus edge (v, u) form a cycle, contradicting that G is a DAG.

If (u, v) is a forward edge, then u is an ancestor of v, implying that $v.f < u.f$.

Otherwise (u, v) is cross edge, then $v.f < u.f$.

T-sort a DAG by peeling the nodes of in-degree 0

```
T-sort-peeling(G){
   L ← ∅; // L will be updated and eventually becomes a list of nodes representing the T-sorted node ordering
   while(there exists a node v in G that has in-degree 0){
      insert v onto the back of L;
   }
}
```

Correctness

For every finite DAG, there exists a node of in-degree 0. Otherwise, let $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t$ be the longest simple path in the DAG. Since v_1 has in-degree > 0, there exists an edge (u, v_1) in G and u doesn't appear on the path (otherwise form a directed cycle). Then, $u \rightarrow v_1 \rightarrow \ldots \rightarrow v_t$ is a longer path $\rightarrow\leftarrow$.

Everytime we remove a node of in-degree 0, the resulting graph remains a DAG (you cannot create a cycle by node removal), and therefore L contains all the nodes in G.

If there is an edge (u, v), then v cannot have in-degree 0 before u is placed on L. Hence, L is a feasible node ordering.

Exercise

Show that T-sort-peeling(G) can be implemented in O(n+m) time, where n denotes the number of nodes in G and m denotes the number of edges in G.

Exercise

Show that T-sort-peeling(G) can be implemented in O(n+m) time, where n denotes the number of nodes in G and m denotes the number of edges in G.

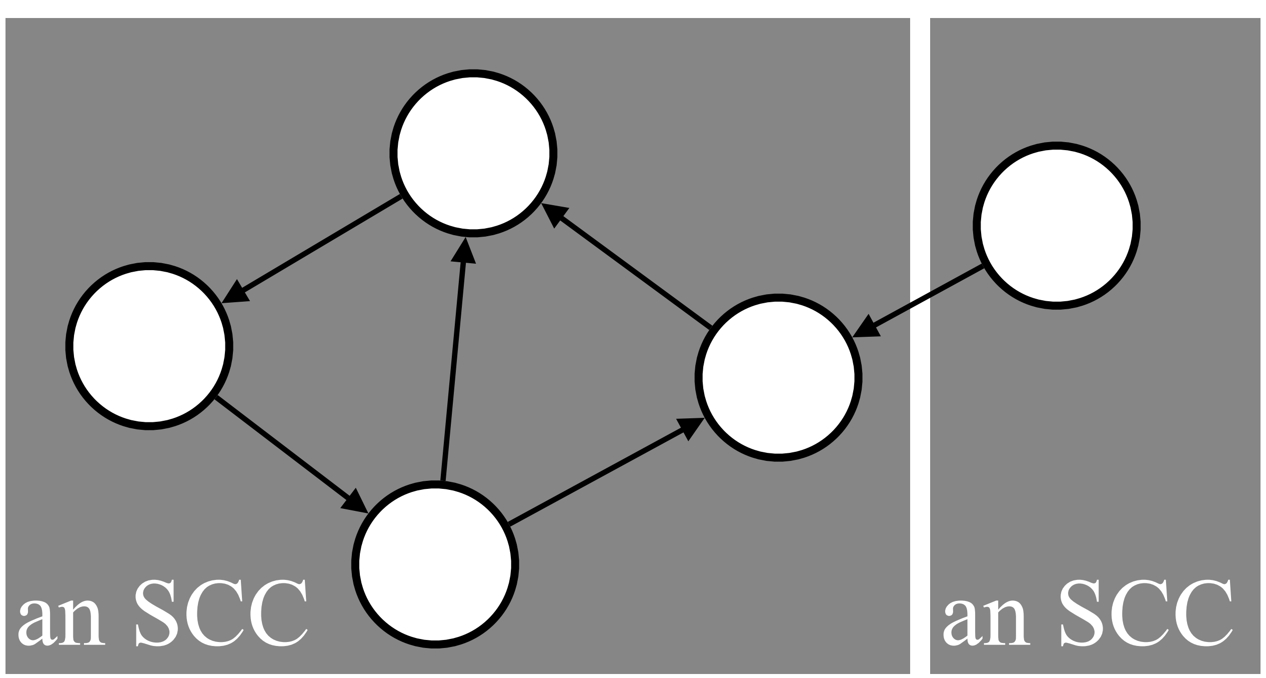
T-sort-DFS is asympotically as fast as T-sort-peeling.

Strongly Connected Components

Strongly connected components

Let G be a directed graph, we say a node set U in G is a strongly connected component if U is maximal and for every pair of nodes p, q $\in$ U, there is a directed path from p to q and from q to p. We say U is maximal if there exists no node set X in G so that U is a proper subset of X and X is a strongly connected component.

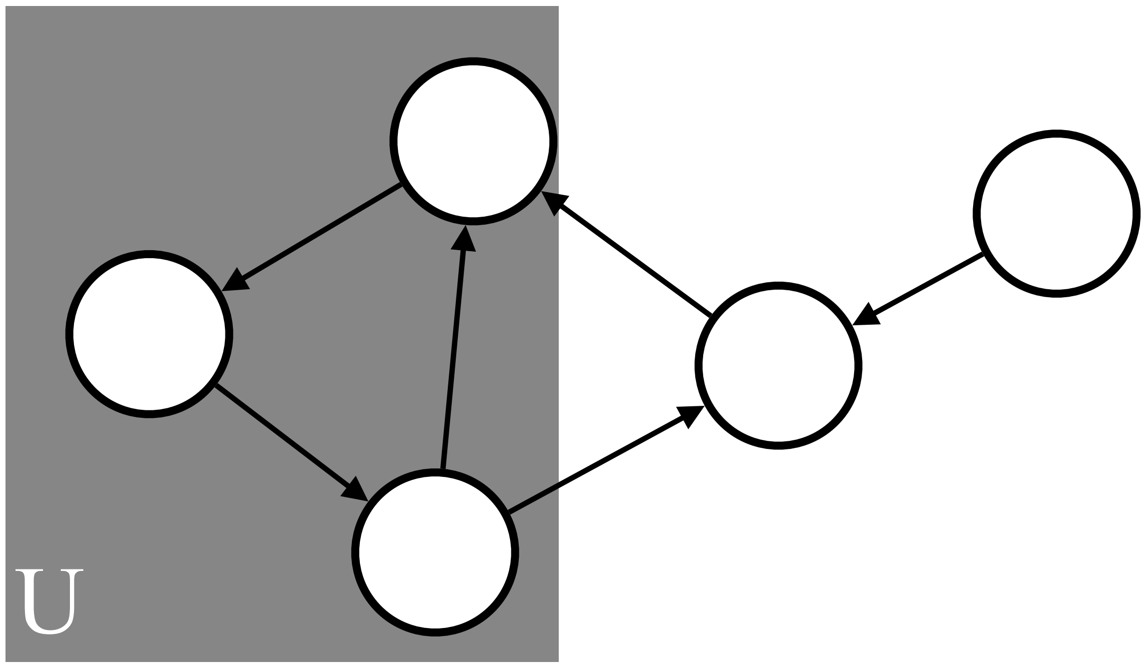
Example.



Strongly connected components

Let G be a directed graph, we say a node set U in G is a strongly connected component if U is maximal and for every pair of nodes p, q $\in$ U, there is a directed path from p to q and from q to p. We say U is maximal if there exists no node set X in G so that U is a proper subset of X and X is a strongly connected component.

Example.

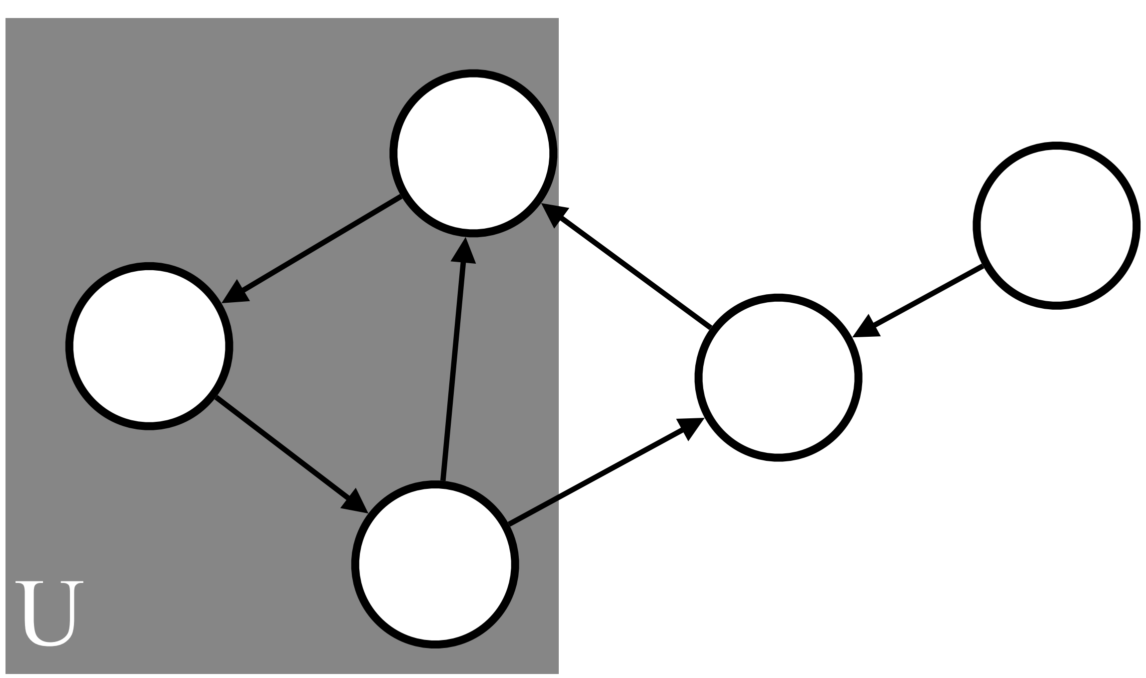


Is the node set U an SCC?

Strongly connected components

Let G be a directed graph, we say a node set U in G is a strongly connected component if U is maximal and for every pair of nodes p, q ∈ U, there is a directed path from p to q and from q to p. We say U is maximal if there exists no node set X in G so that U is a proper subset of X and X is a strongly connected component.

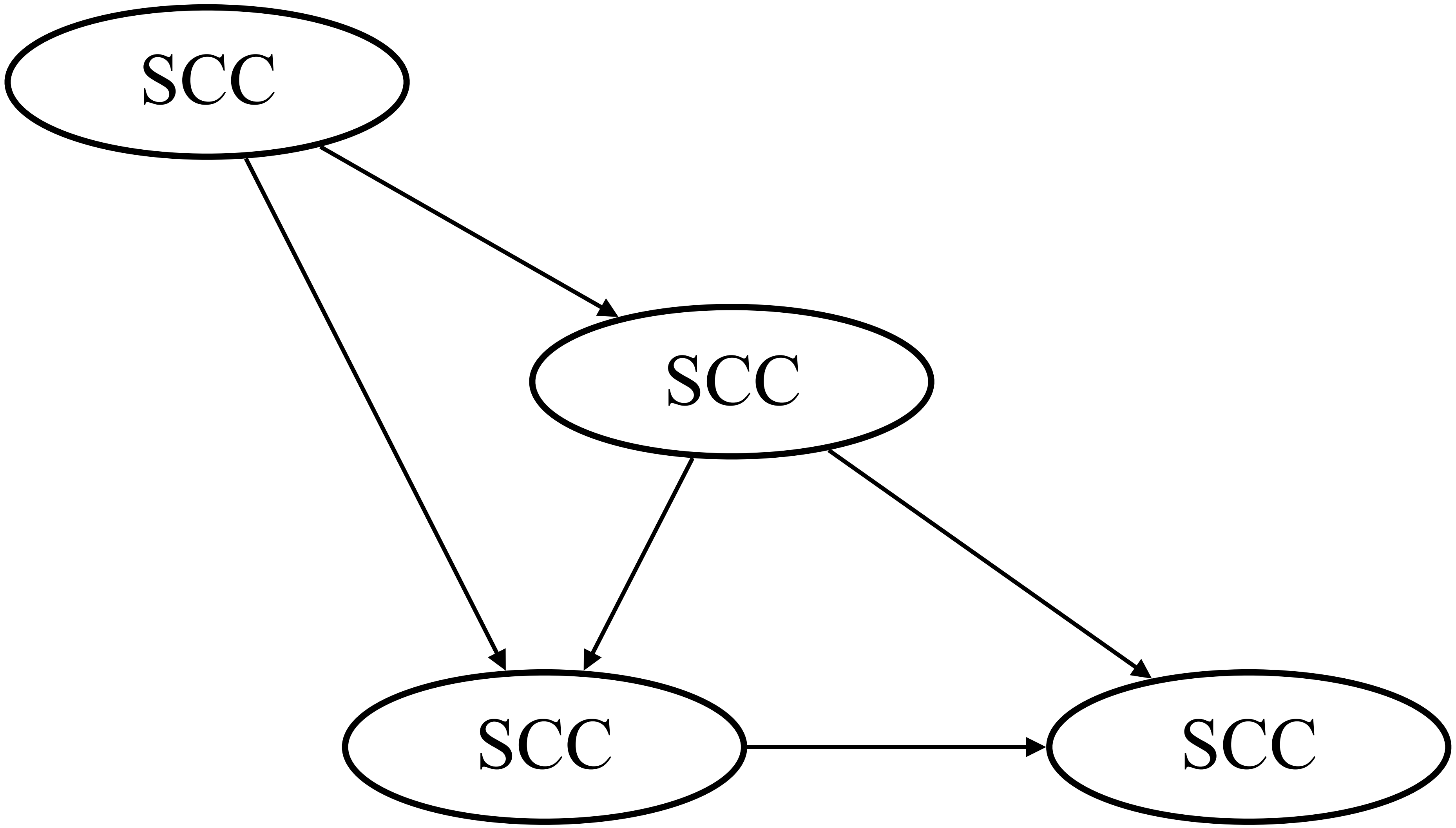
<u>Example</u>.



Is the node set U an SCC?

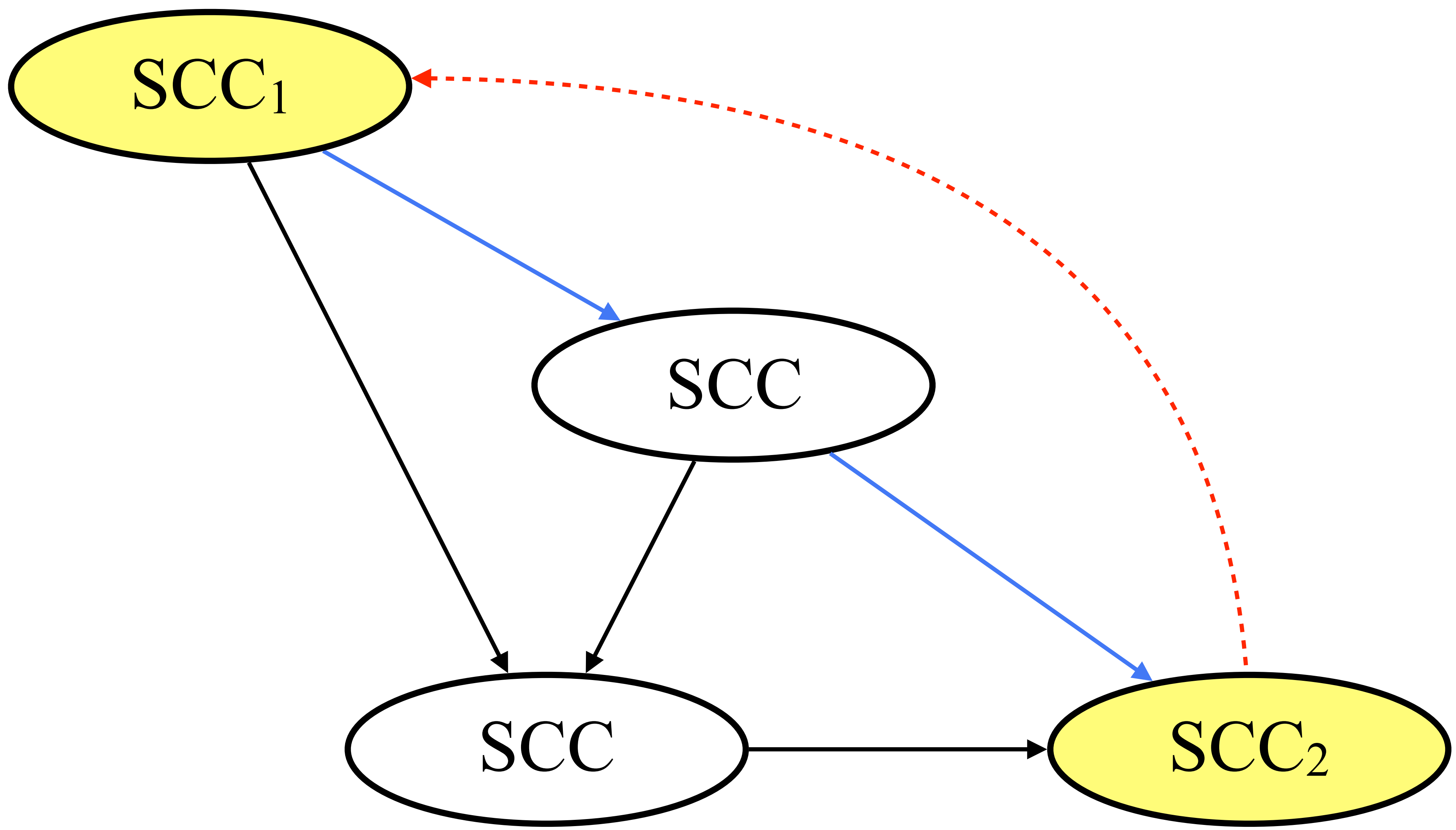
No, U is not maximal.

Decomposing a directed graph into SCCs



If we view each SCC as a supernode, then the resulting graph is a DAG. (Why?)

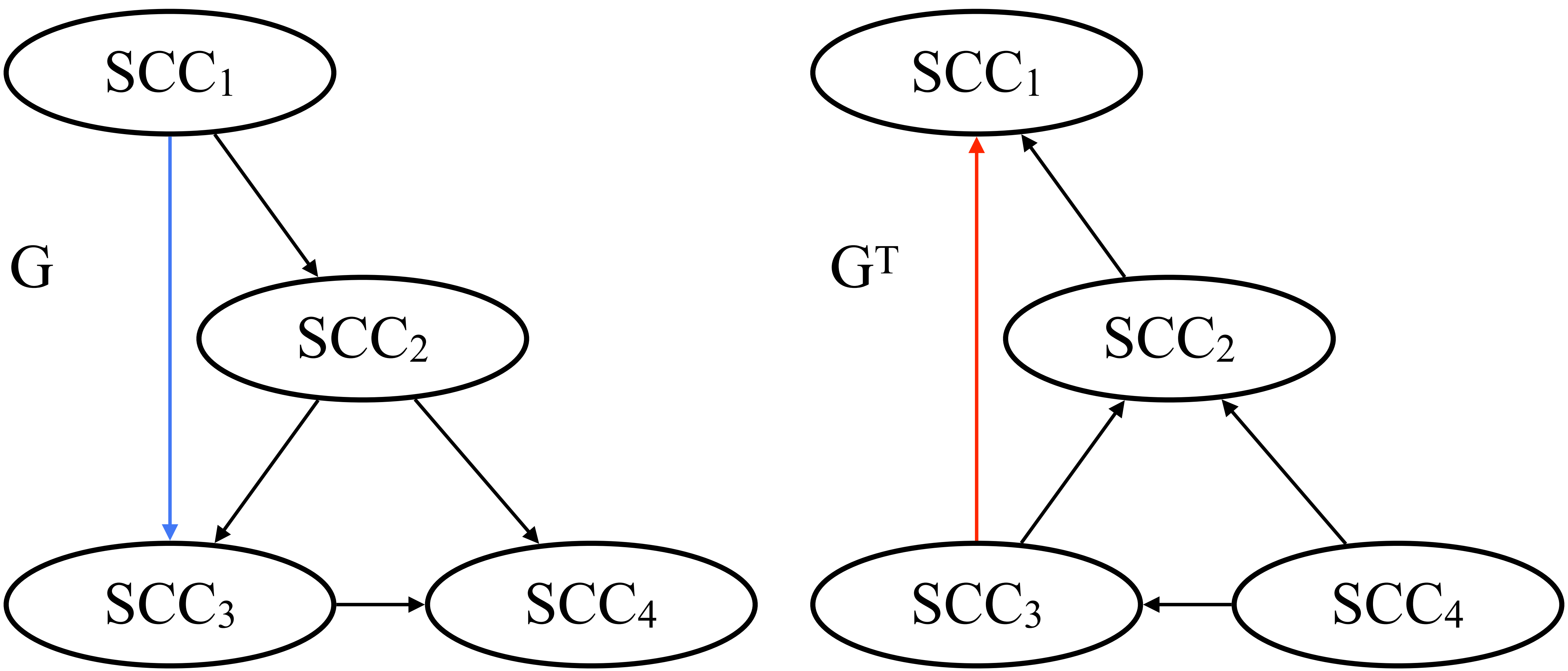
Decomposing a directed graph into SCCs



If there a path from SCC_1 to SCC_2, then there is no path in the reverse direction. Otherwise, $SCC_1 \cup SCC_2$ is a larger SCC, contradicting the maximality of SCC_1.

The transpose graph of G

Let G^T be the transpose graph of G. In other words, edge $(u, v) \in G^T$ if and only if edge $(v, u) \in G$. Then, the SCC decomposition of G and G^T are the same except the edges between SCCs have reverse direction.



SCC decomposition by DFS

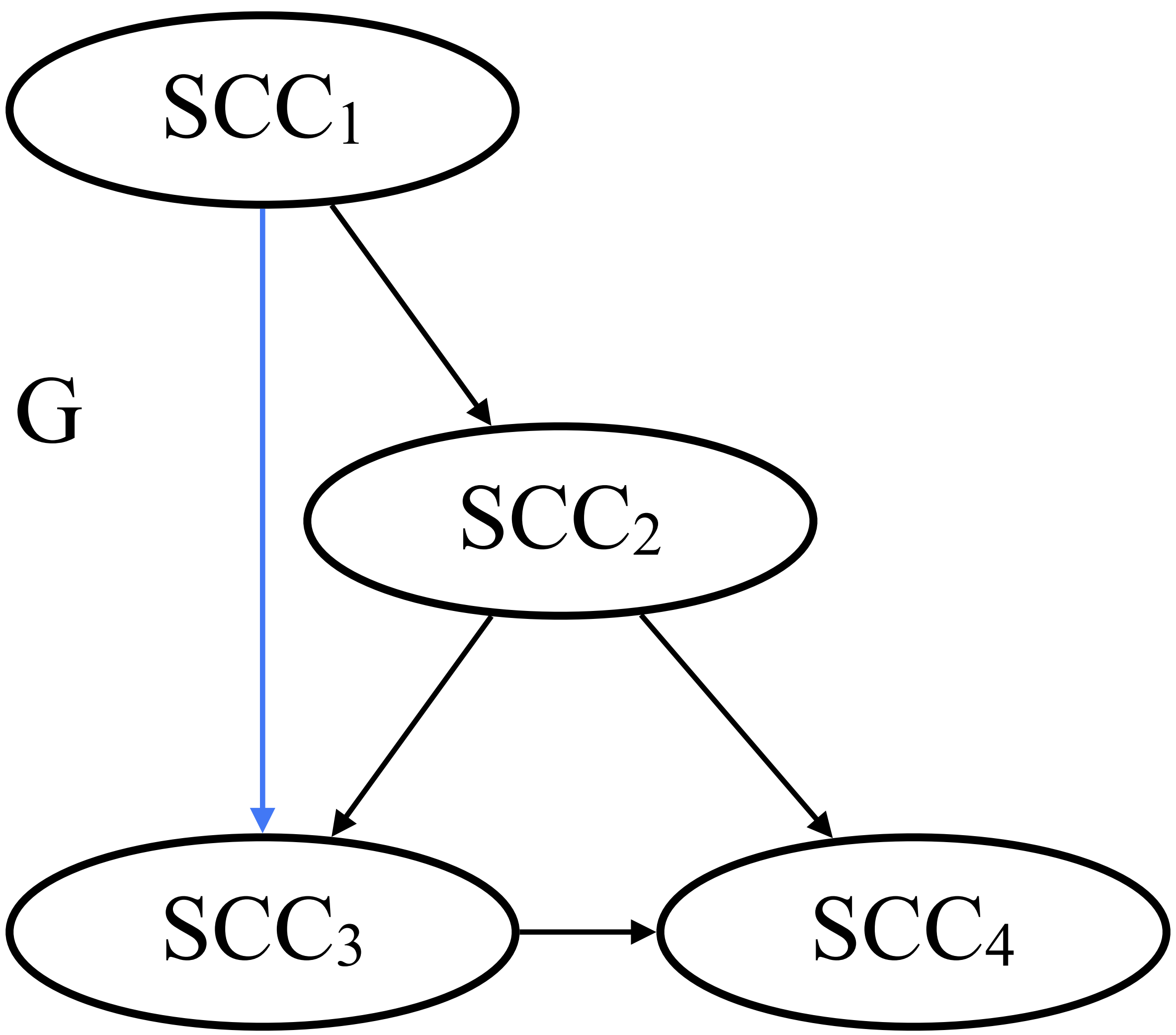
SCC-decomposition(G){
DFS(G);
Let priority[1..n] = {v_1.f, v_2.f, ..., v_n.f};
DFS(G^T); // while picking the root for a new tree, pick the node u whose priority[u] is the largest.
return each tree in the DFS-forest of G^T as an SCC;
}

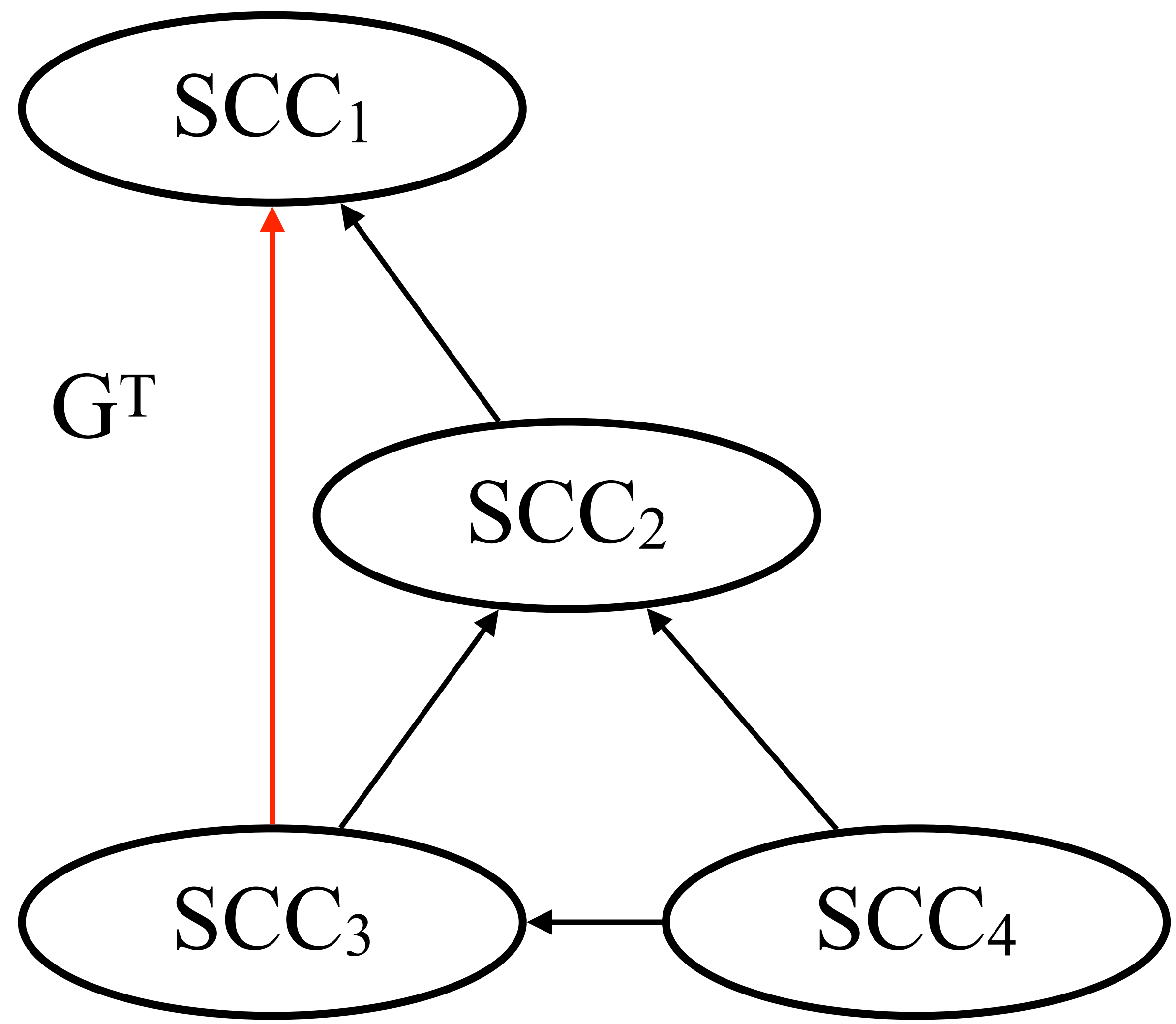
Correctness (1/2)

Claim. If there exists an edge (u, v) so that $u \in SCC_1$ and $v \in SCC_2$, then $f(SCC_1) < f(SCC_2)$, where $f(X)=\max_{v \in X} v.f$.

Proof. The proof is shown in Theorem 22.12 on pp. 608 and Lemma 22.14 on pp. 618 in I2A.

Correctness (2/2)

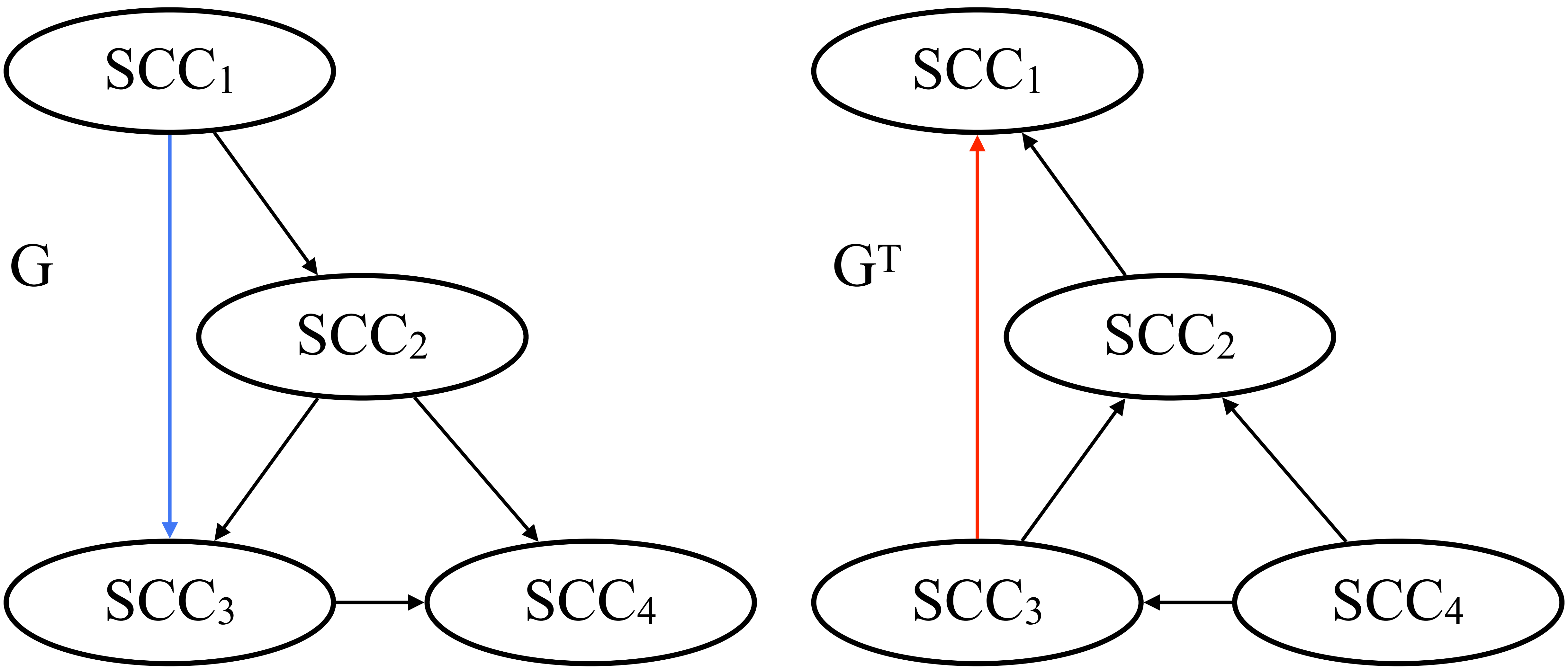




Note that $f(SCC_1) > f(SCC_2) > f(SCC_3) > f(SCC_4)$ in DFS(G).

Thus, $d(SCC_1) > d(SCC_2) > d(SCC_3) > d(SCC_4)$ in DFS(G^T).

Correctness (2/2)



Once we discover a node in SCC_1 in DFS(G^T), then every node in SCC_1 will be discovered (the reachability of DFS). Thus, the first tree return from DFS(G^T) is exactly SCC_1. Similar argument applies for subsequent trees.

Exercise (2SAT)

Input: a conjunctive normal form CNF in which each clause has exactly (or at most, they are equivalent) 2 literals.

Output: a truth assignment so that the CNF is evaluated as True.

<u>Example</u>.

$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee \neg x_2)$

a clause

a literal

In a kSAT, each clause has exactly (at most) k literals.

Let (x_1, x_2, x_3) = (True, Flase, True), then the above 2SAT is satisfied (i.e. evaluated as True).

Exercise (2SAT)

Input: a conjunctive normal form CNF in which each clause has exactly (or at most, they are equivalent) 2 literals.

Output: a truth assignment so that the CNF is evaluated as True.

<u>Example</u>.

$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee \neg x_2)$

a clause

a literal

2SAT can be solved in linear time by the SCC decomposition.
It is conjectured that 3SAT cannot be solved in polynomial-time.

In a kSAT, each clause has exactly (at most) k literals.

Let (x_1, x_2, x_3) = (True, Flase, True), then the above 2SAT is satisfied (i.e. evaluated as True).