CSC263 Winter 2020

Graphs: DFS

Lecture 8

Midterm

6pm IB110
80 minutes

TA Office Hours

- 10-11 am MN3120 (Naaz)
- 3-4 pm MN3220 (Rida)

Don't forget your T-card!

Practise previous midterm questions!

Recap

- ADT: Graph
- Data structures
  - Adjacency **matrix**
  - Adjacency **list**
- Graph operations
  - Add vertex, remove vertex, …, edge query, …
  - **Traversal**

Graph Traversals

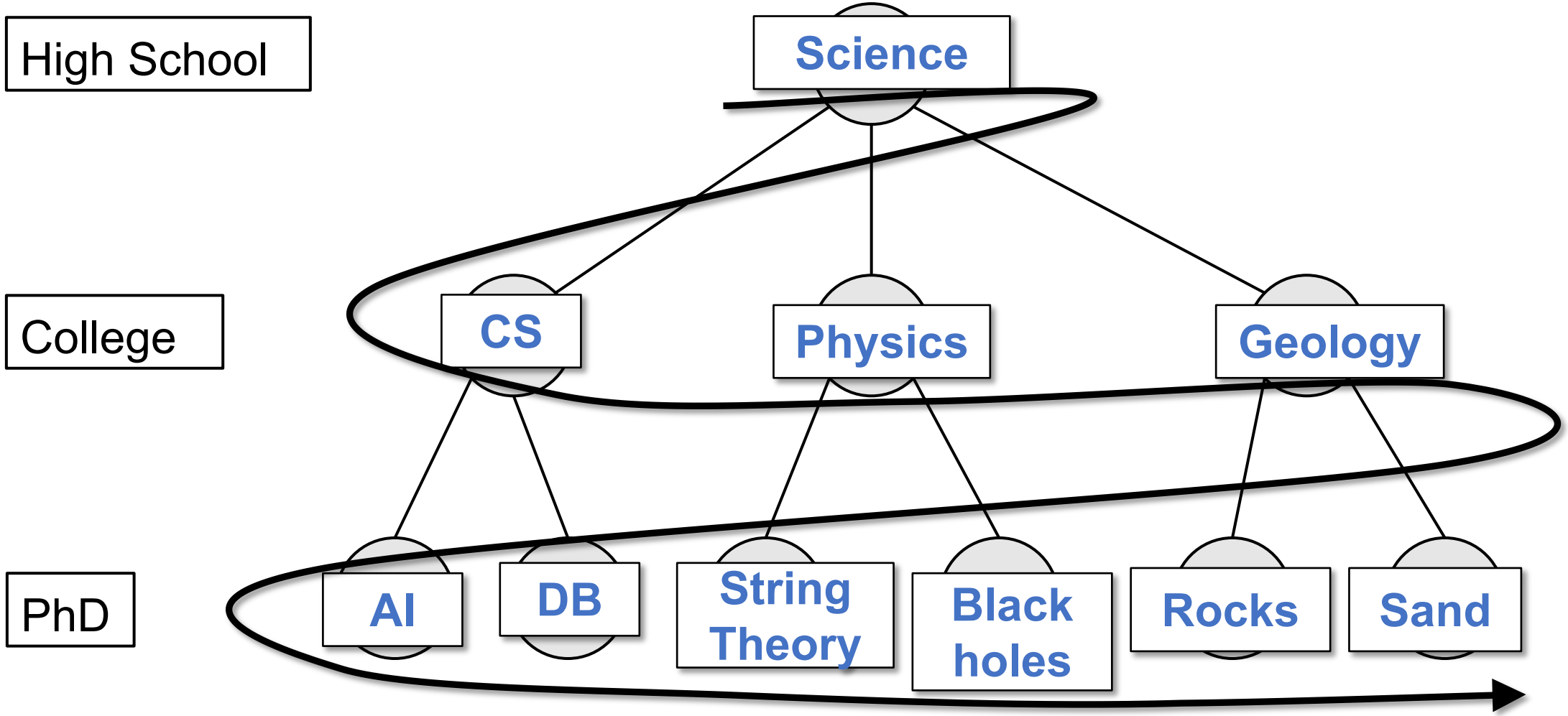
Visiting every vertex once, starting from a given vertex.

The visits can follow different orders, we will study the following two ways

- **Breadth First Search (BFS)**
- **Depth First Search (DFS)**

The Breadth-First ways of learning these subjects

➔ Level by level, finish high school, then all subjects at College level, then finish all subjects in PhD level.

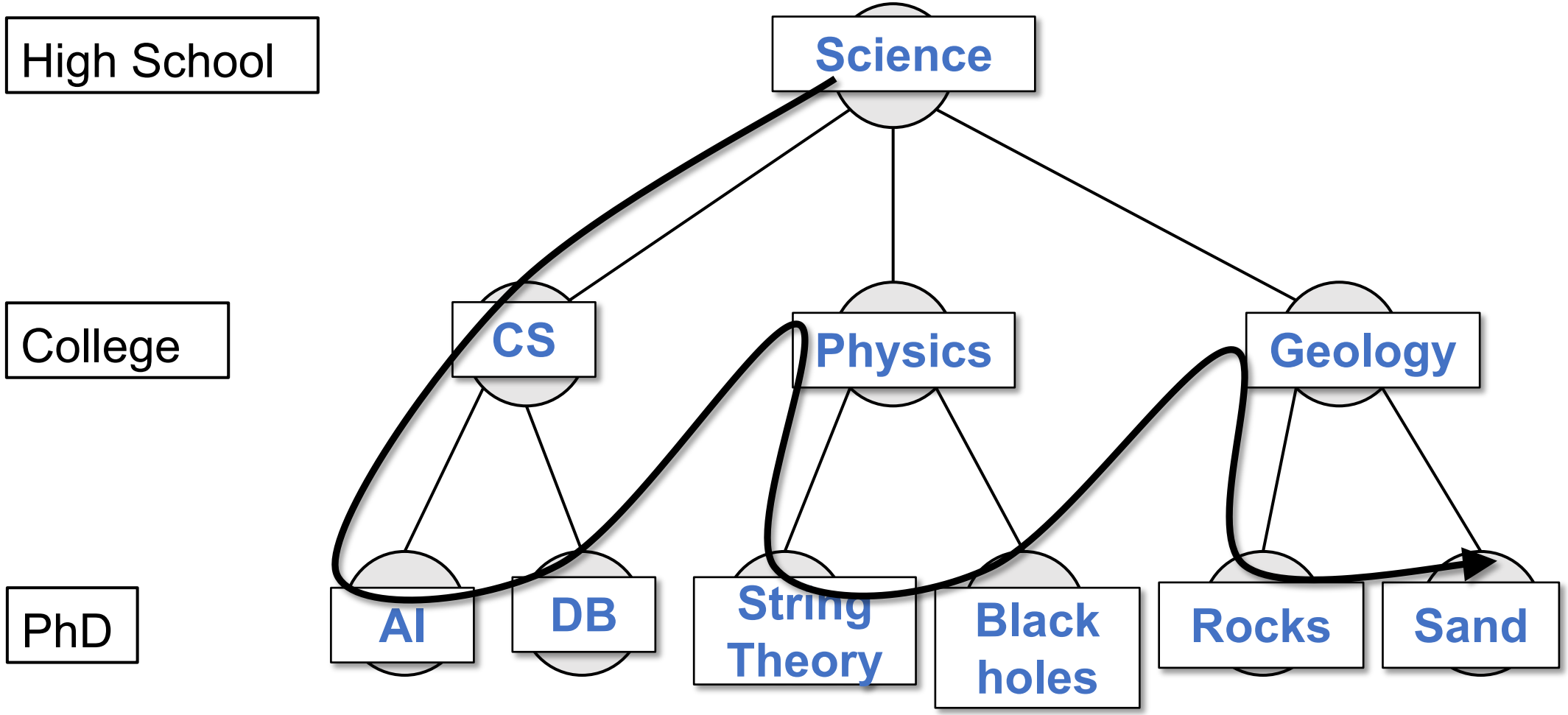


Recap of BFS

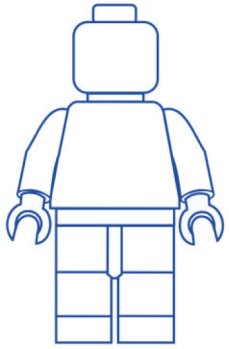
- Prefer to explore **breadth** rather than **depth**
- Useful for getting **single-source shortest paths** on **unweighted graphs**
- Useful for testing reachability
- Runtime **O(|V|+|E|)** with **adjacency list** (with adjacency **matrix** it'll be different)

The Depth-First way of learning these subjects

➔ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.

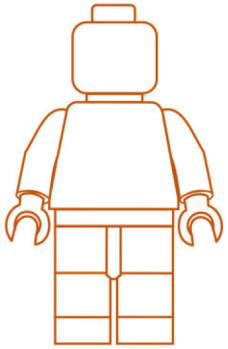


DFS and BFS



```
NOT_QUITE_BFS(root):
  Q ← Queue()
  Enqueue(Q, root)
  while Q not empty:
    x ← Dequeue(Q)
    print x
    for each child c of x:
      Enqueue(Q, c)
```

```
NOT_QUITE_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

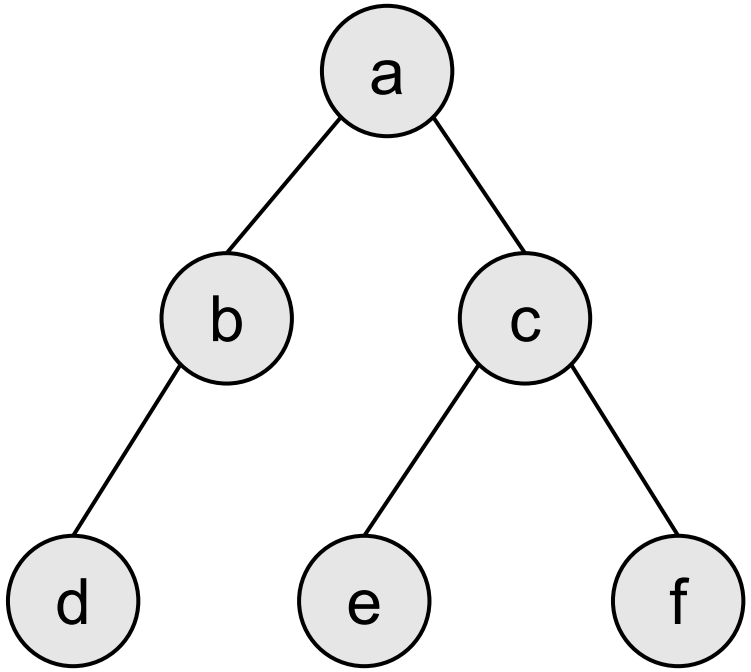


They are twins!

Sepcial Case: DFS in a Tree

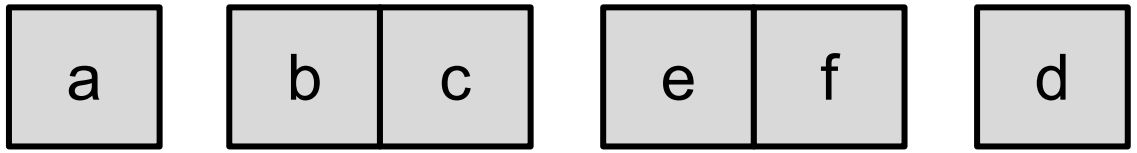
Output:

a
c
f
e
b
d




```
NOT_QUITE_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

Stack:



POP **POP** **POP** **POP** **POP** **POP**

A nicer way to write this code?

```
NOT_QUITE_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

```
NOT_QUITE_DFS(root):
  print root
  for each child c of root:
      NOT_QUITE_DFS(c)
```

The use of stack is basically implementing **recursion**.

Exercise: Trace this code on the tree in the previous slide.

DFS in Graphs

Explore edges out of the most recently discovered vertex *v* that still has unexplored edges leaving it.

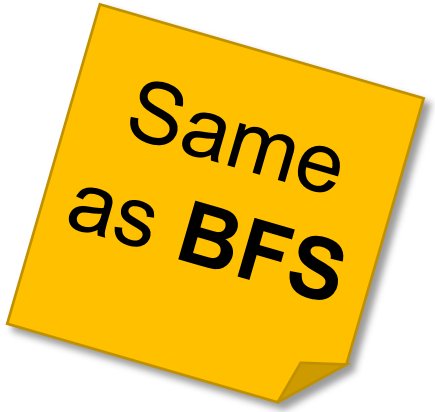
Once all of *v*'s edges have been explored, the search **backtracks** to explore edges leaving the vertex from which *v* was discovered.

Continue until we have **discovered all reachable vertices** from the original source vertex.

If any **undiscovered vertices** remain, DFS **selects** one of them as a **new source**, and repeats the search from that source.

Algorithm repeats this process until it has discovered **every vertex**.

Visiting a Vertex only once



Remember you visited it by **labelling** it using **colours**.

➔ **White**: “unvisited”

➔ **Gray**: “encountered”

➔ **Black**: “explored”

➔ Initially all vertices start off as **white**

➔ Colour a vertex **gray** the **first** time visiting it

➔ Colour a vertex **black** when **all** its **neighbours** have been encountered

➔ Don’t visit **gray** or **black** vertices

➔ In the end, all vertices are **black** (sort-of)

Additonal Values to Remember

- **pi[v]**: the vertex from which v is encountered
  - "I was introduced as **whose** neighbour?"

Same as **BFS**

Clock ticking, incremented whenever someone's colour is changed

- For each vertex v, remember two **timestamps**
  - **d[v]**: "**discovery** time", when the vertex is first encountered (when it becomes gray)
  - **f[v]**: "**finishing** time", when all the vertex's neighbours have been visited (when become black).

different from d[v] in BFS

The DFS Pseudocode (incomplete)

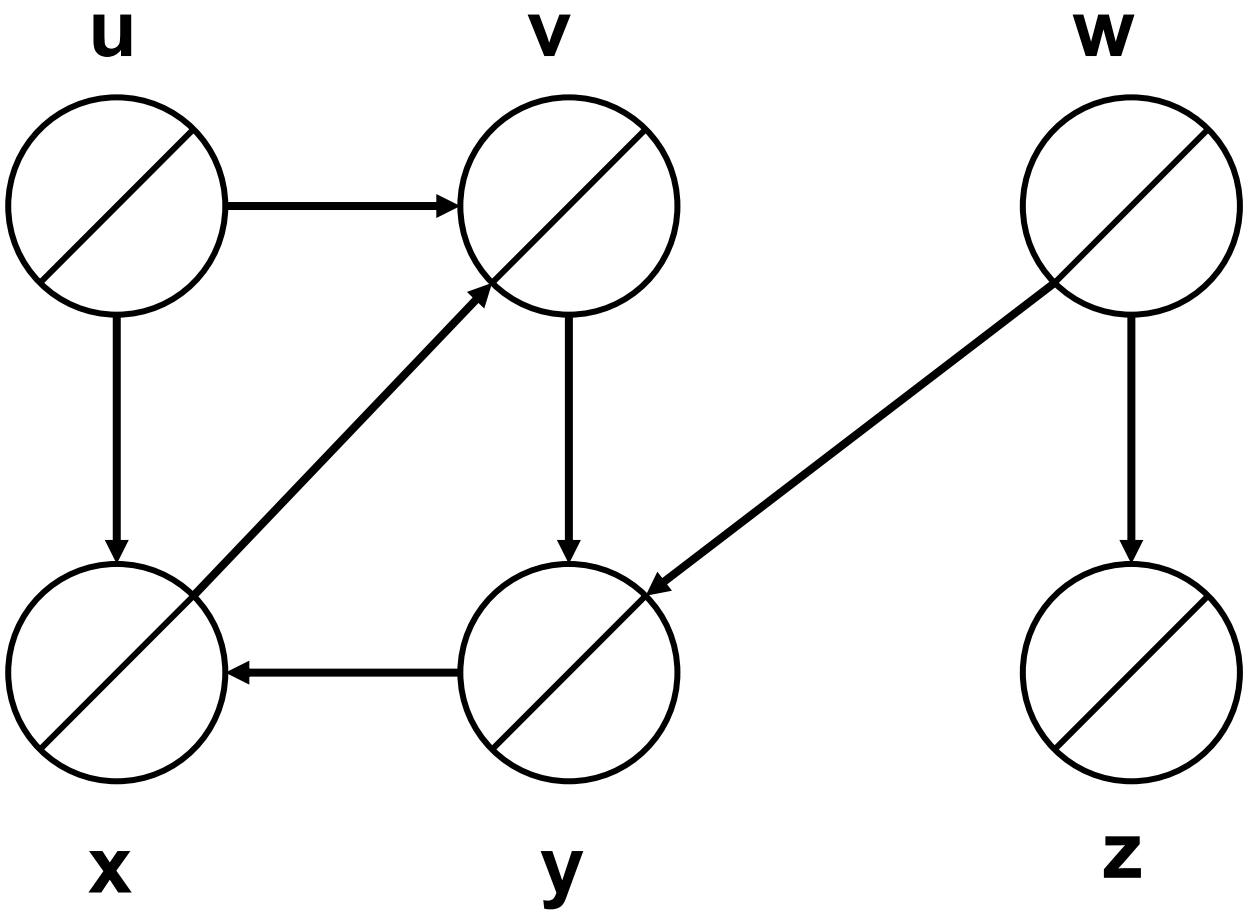
```
DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time          # keep discovery time on first encounter
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time          # keep finishing time after exploring all neighbours
```

The orange part is the same as NOT_QUITE_DFS

Why **DFS_VISIT** instead of **DFS**? We will see...

Let's run an example!

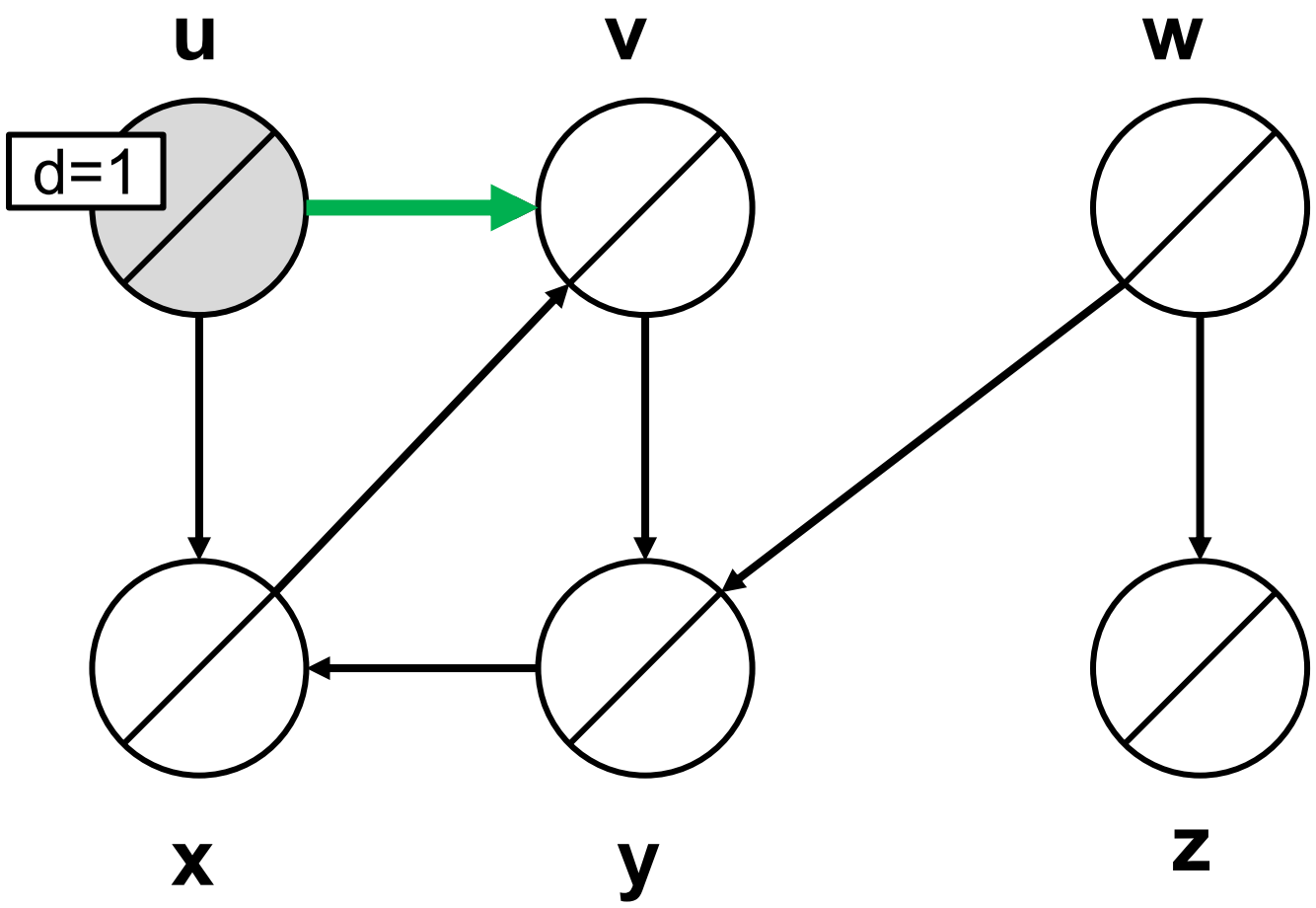
time = 0



DFS_VISIT(G, u)

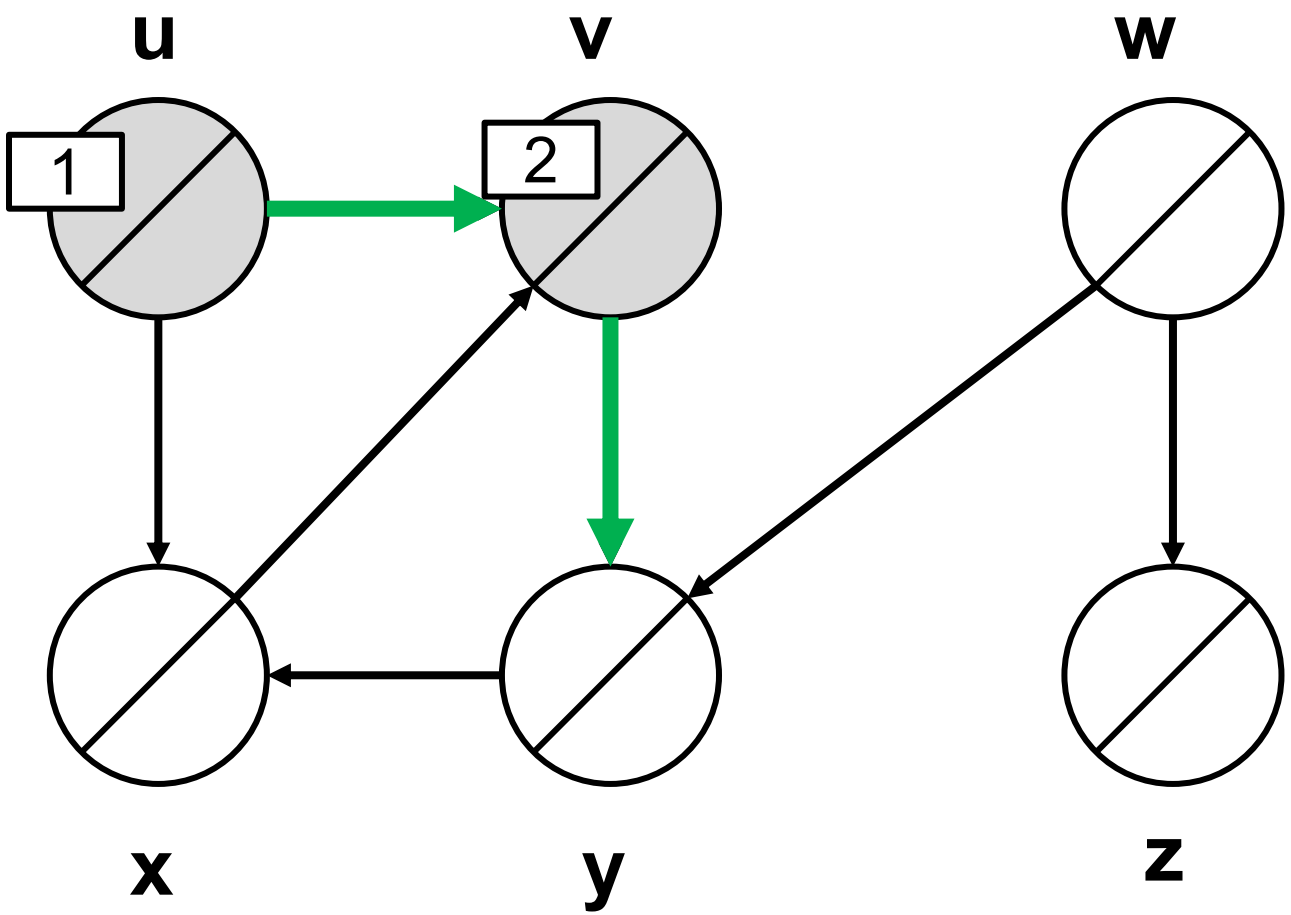
```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 1, encounter the source vertex



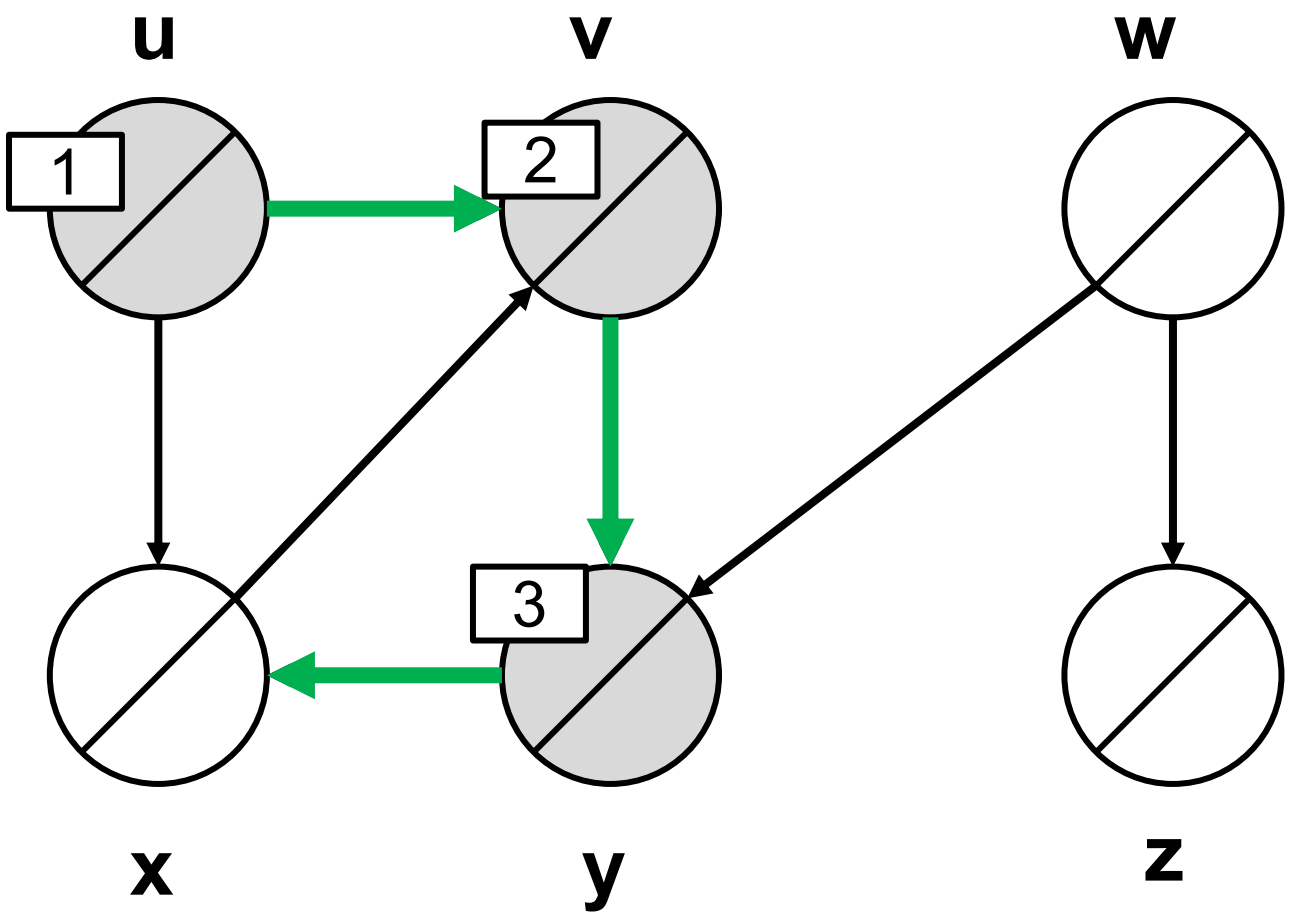

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 2, recursive call, level 2



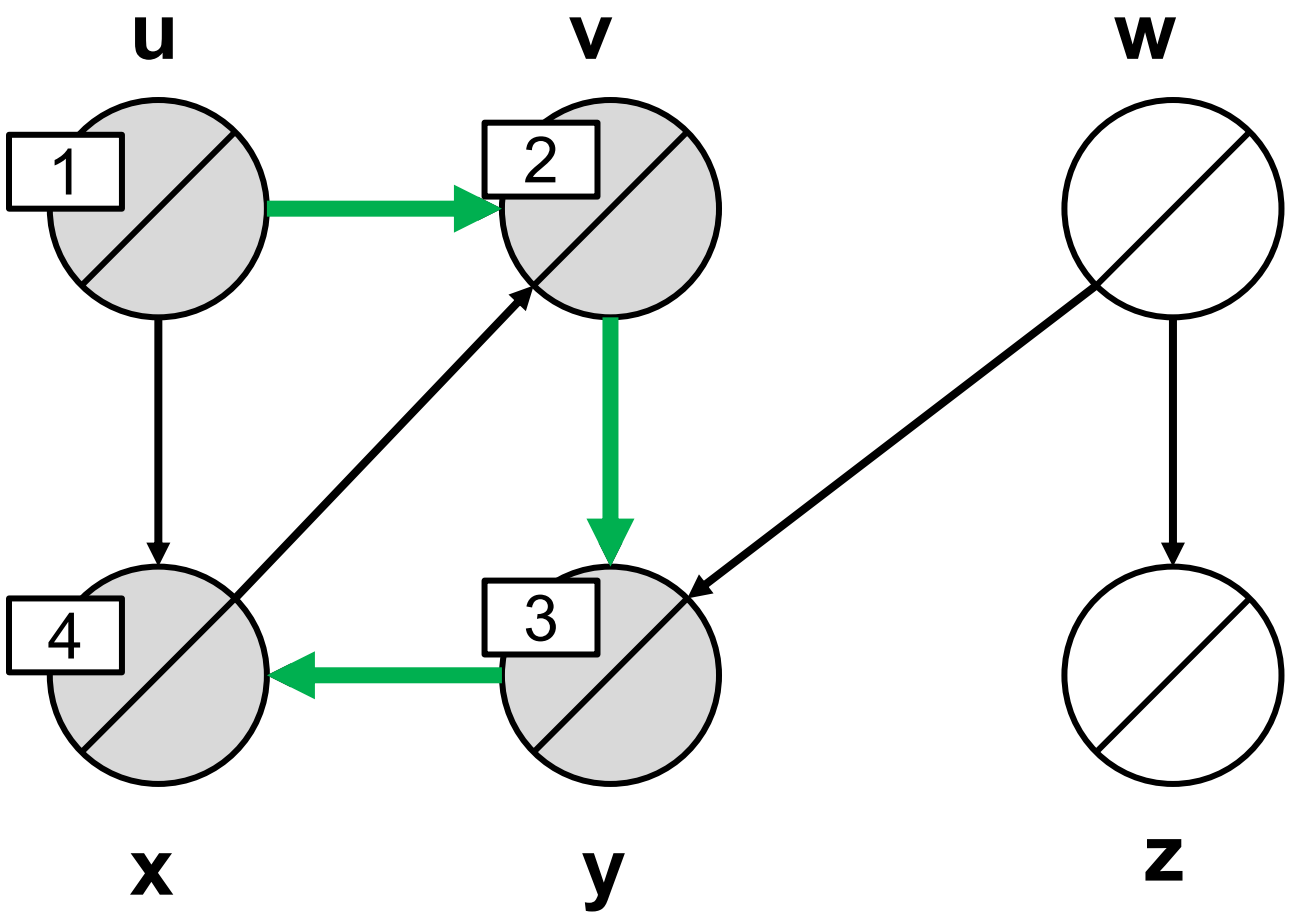

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 3, recursive call, level 3



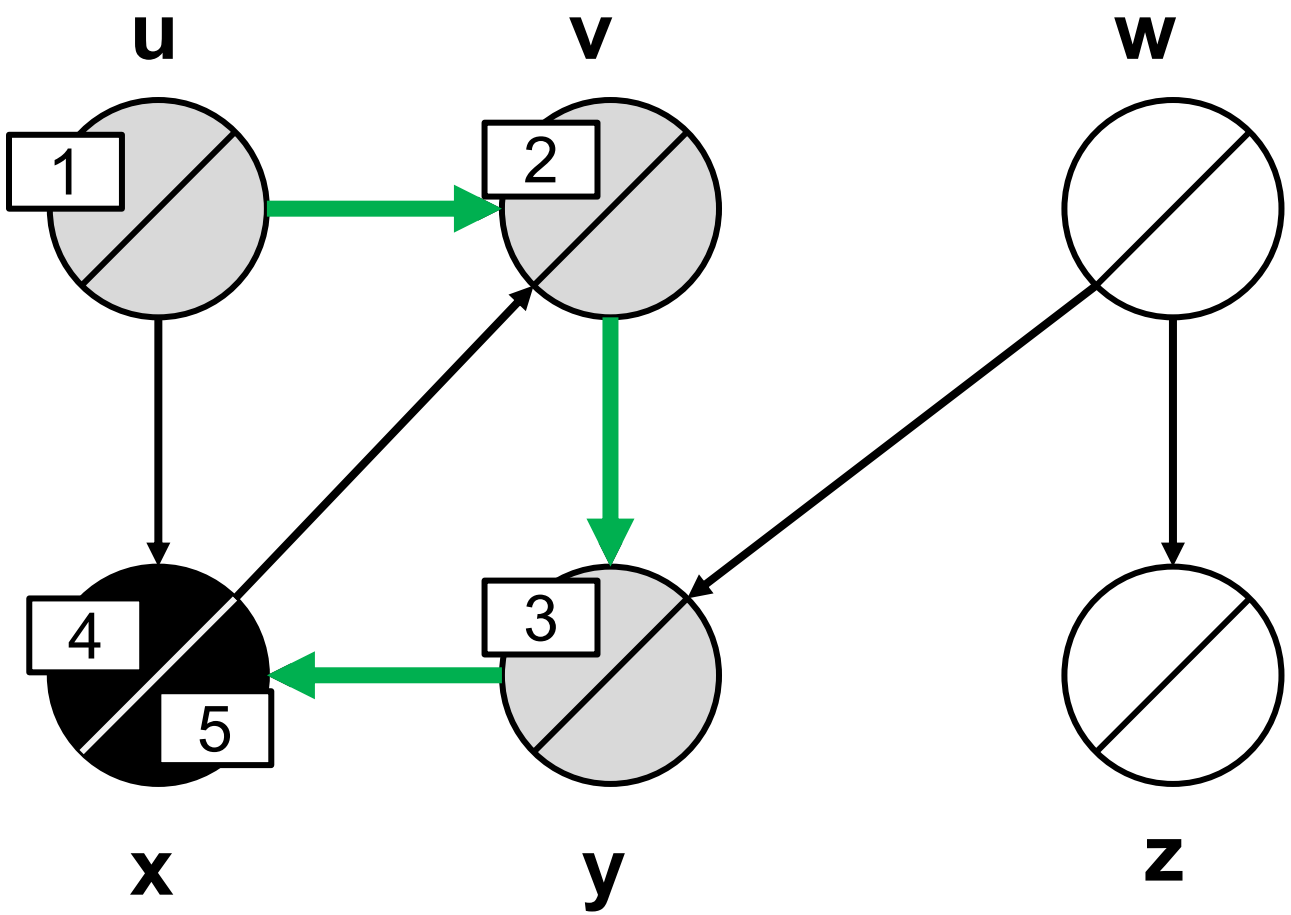

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 4, recursive call, level 4



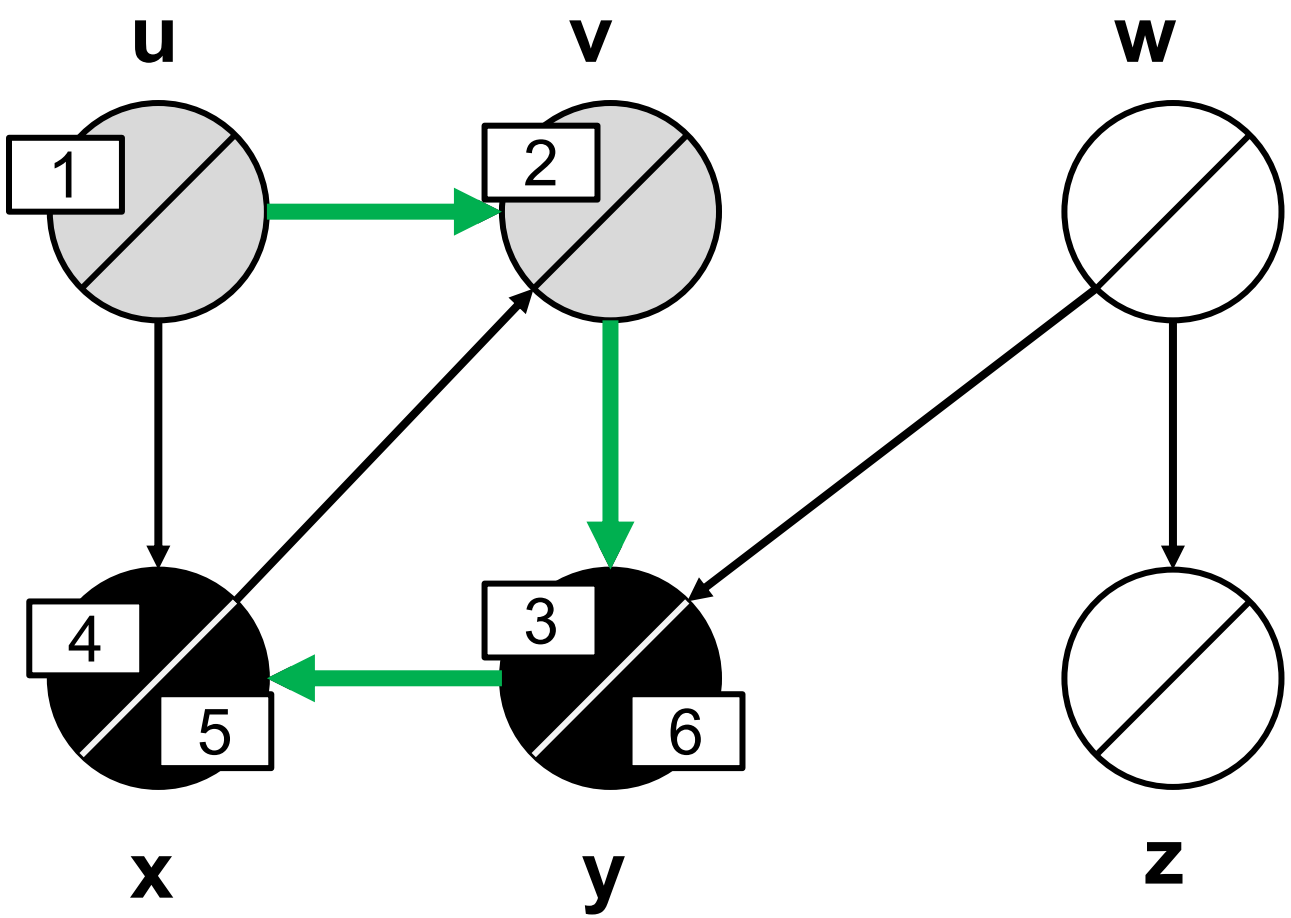

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 5, vertex x finished



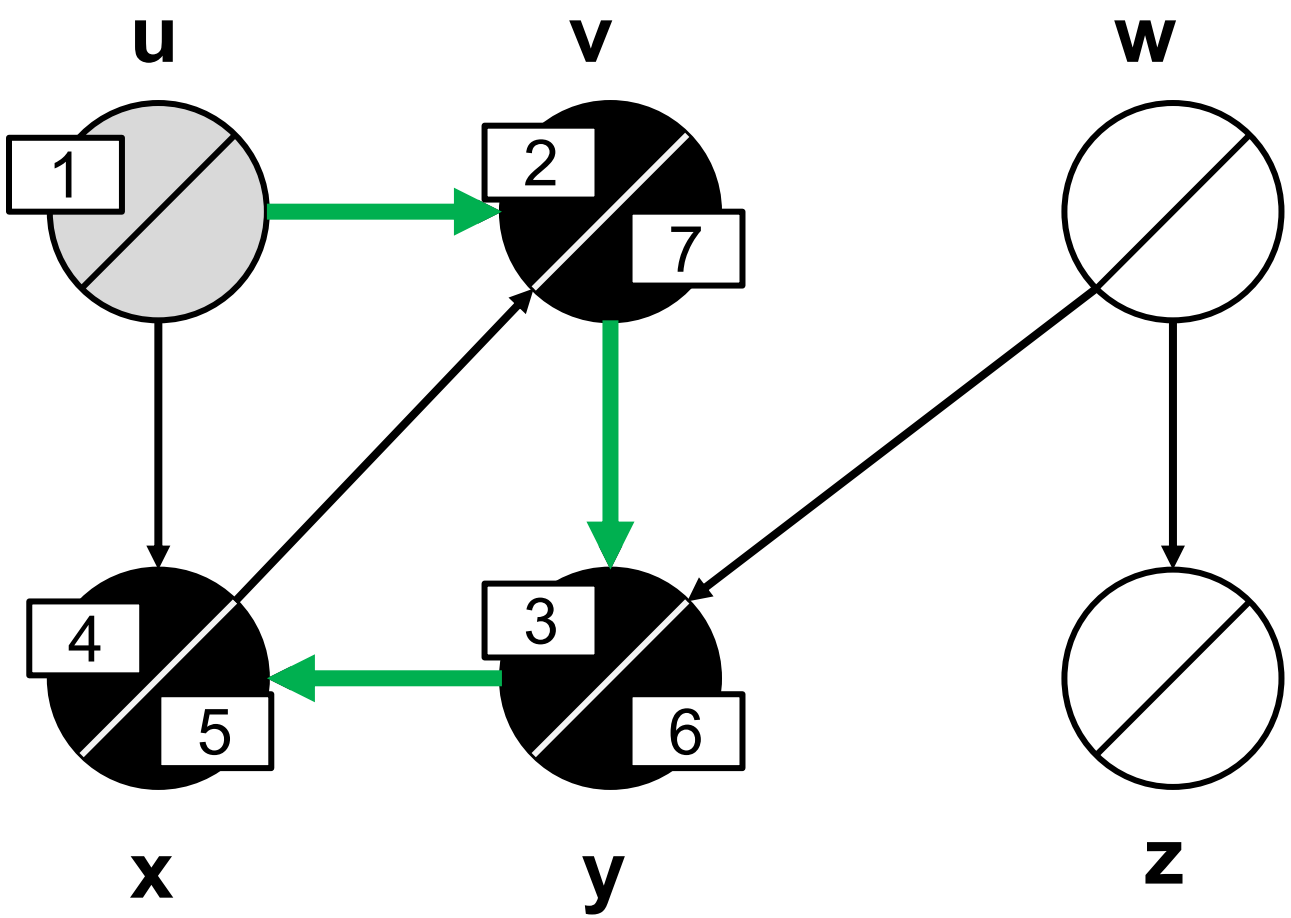

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 6, recursion back to level 3, finish y



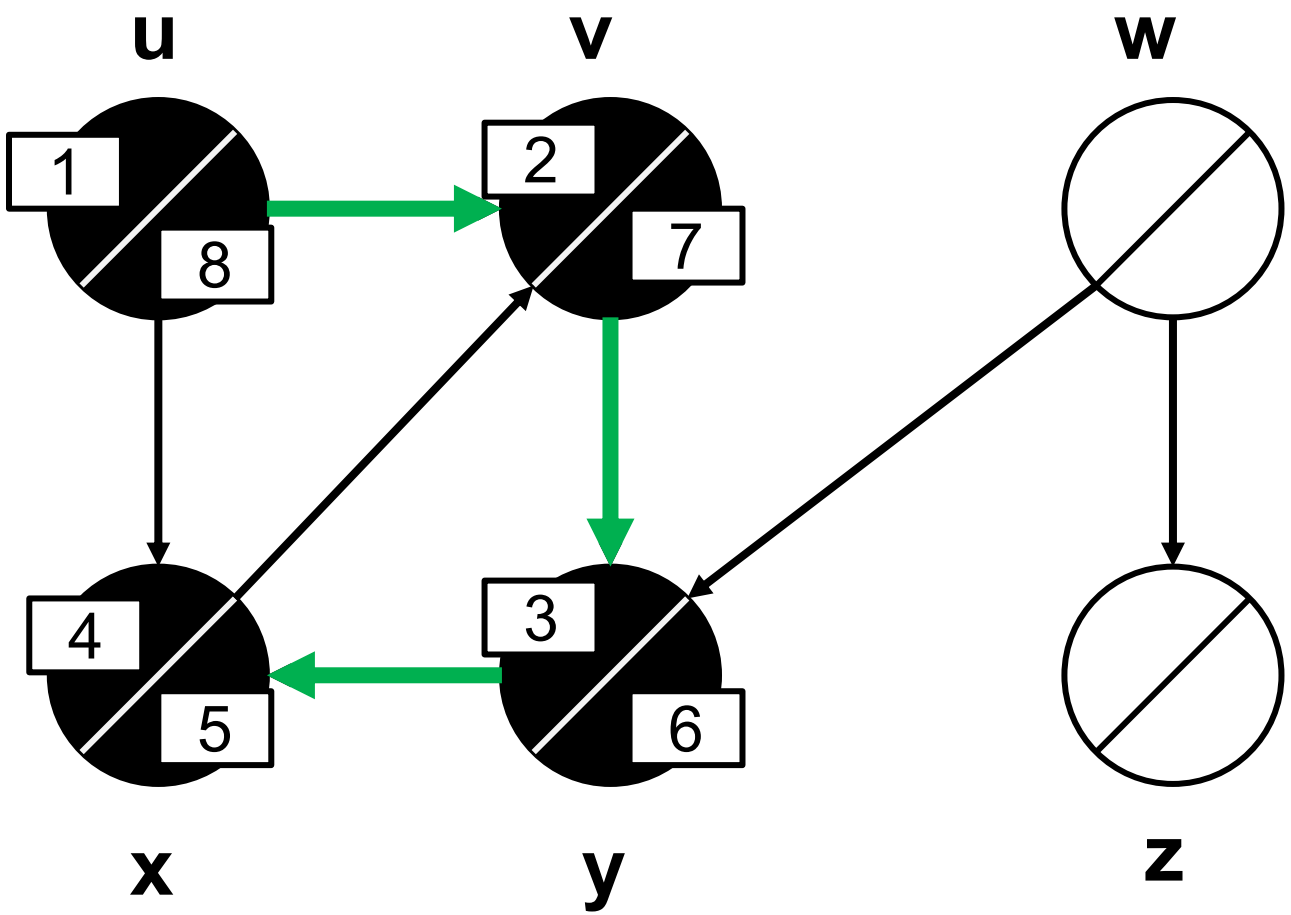

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 7, recursive back to level 2, finish v



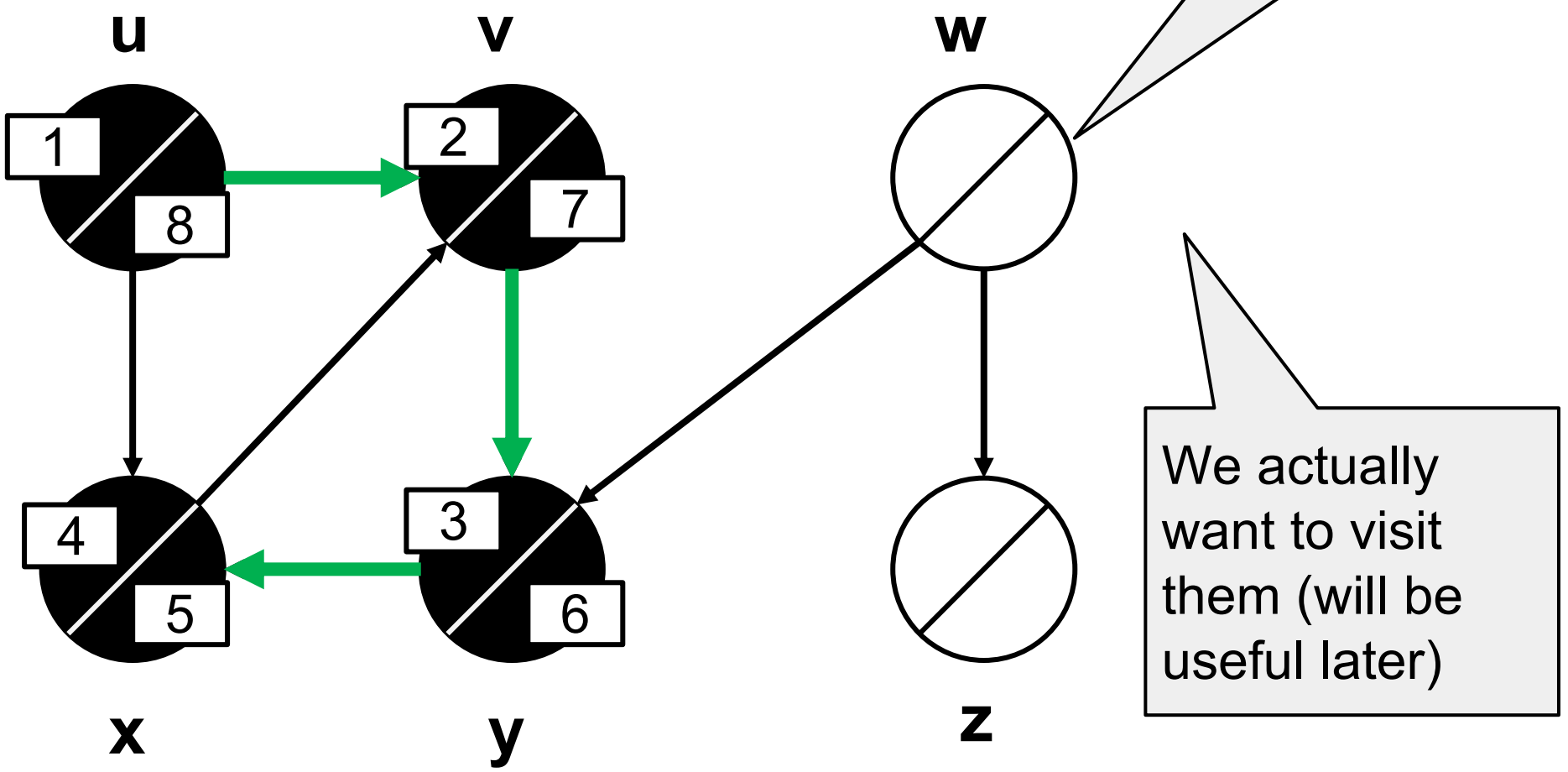

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 8, recursion back to level 1, finish u




```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

DFS_VISIT(G, u) completed



DFS Pseudocode

Strategy Call `DFS_visit` for every unvisited vertex.

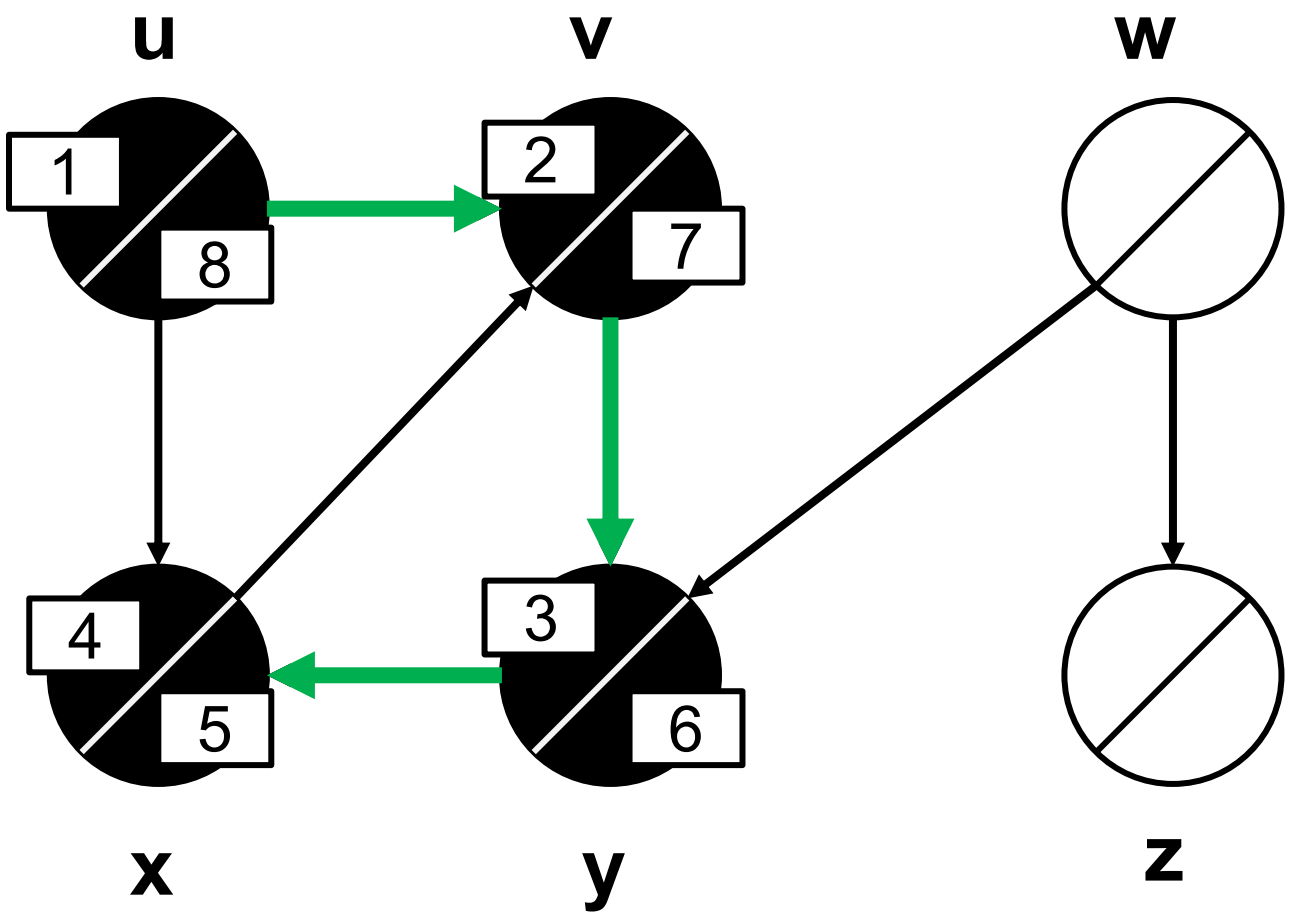
```
DFS(G):
    for each v in G.V:
        colour[v] ← white
        f[v] ← d[v] ← ∞
        pi[v] ← NIL
    time ← 0
    for each v in G.V:
        if colour[v] = white:
            DFS_VISIT(G, v)
```

Initialization

Make sure NO vertex is left with white colour.

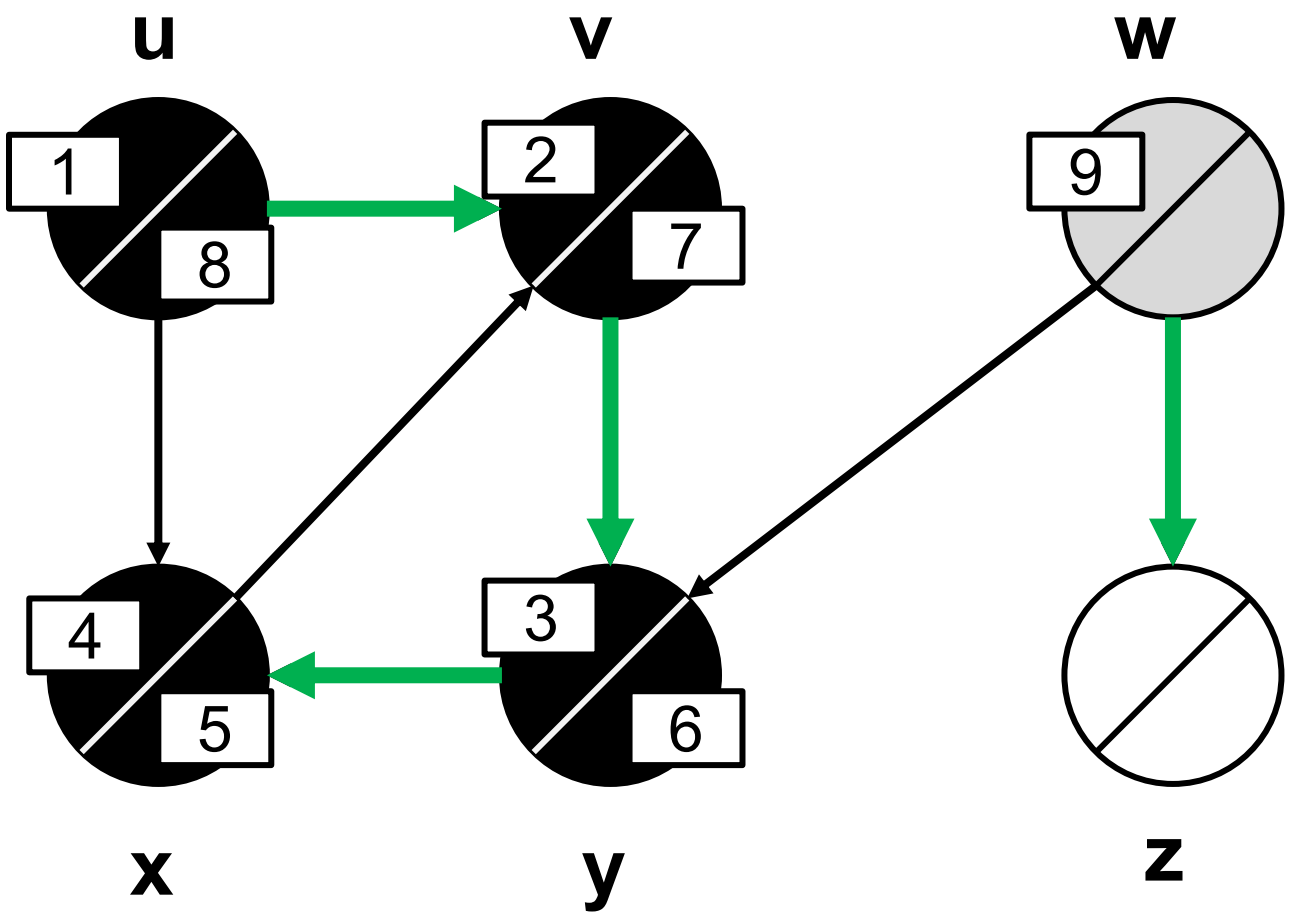
```
DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time
```

So, let's finish this DFS



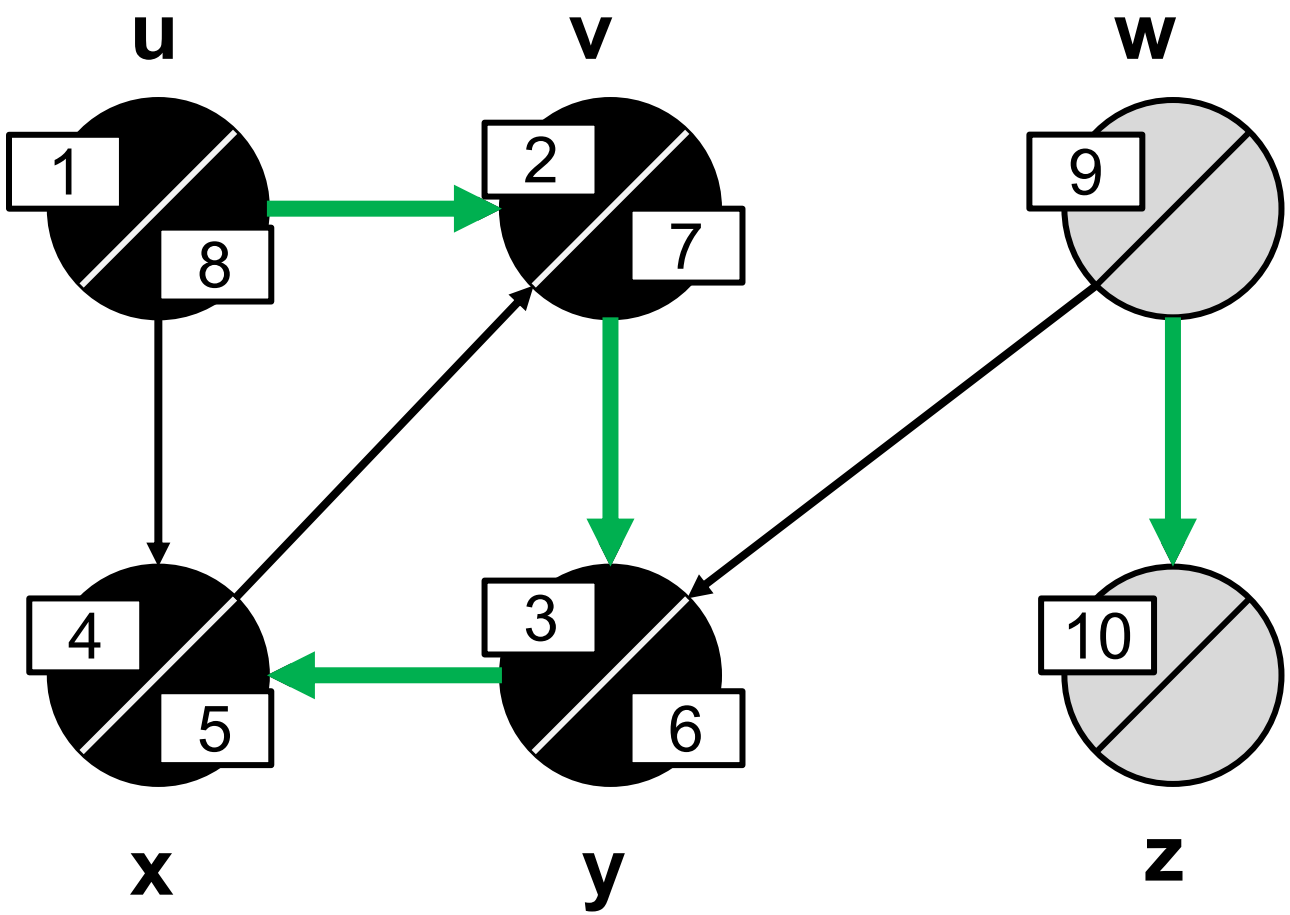

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 9, DFS_VISIT(G, w)



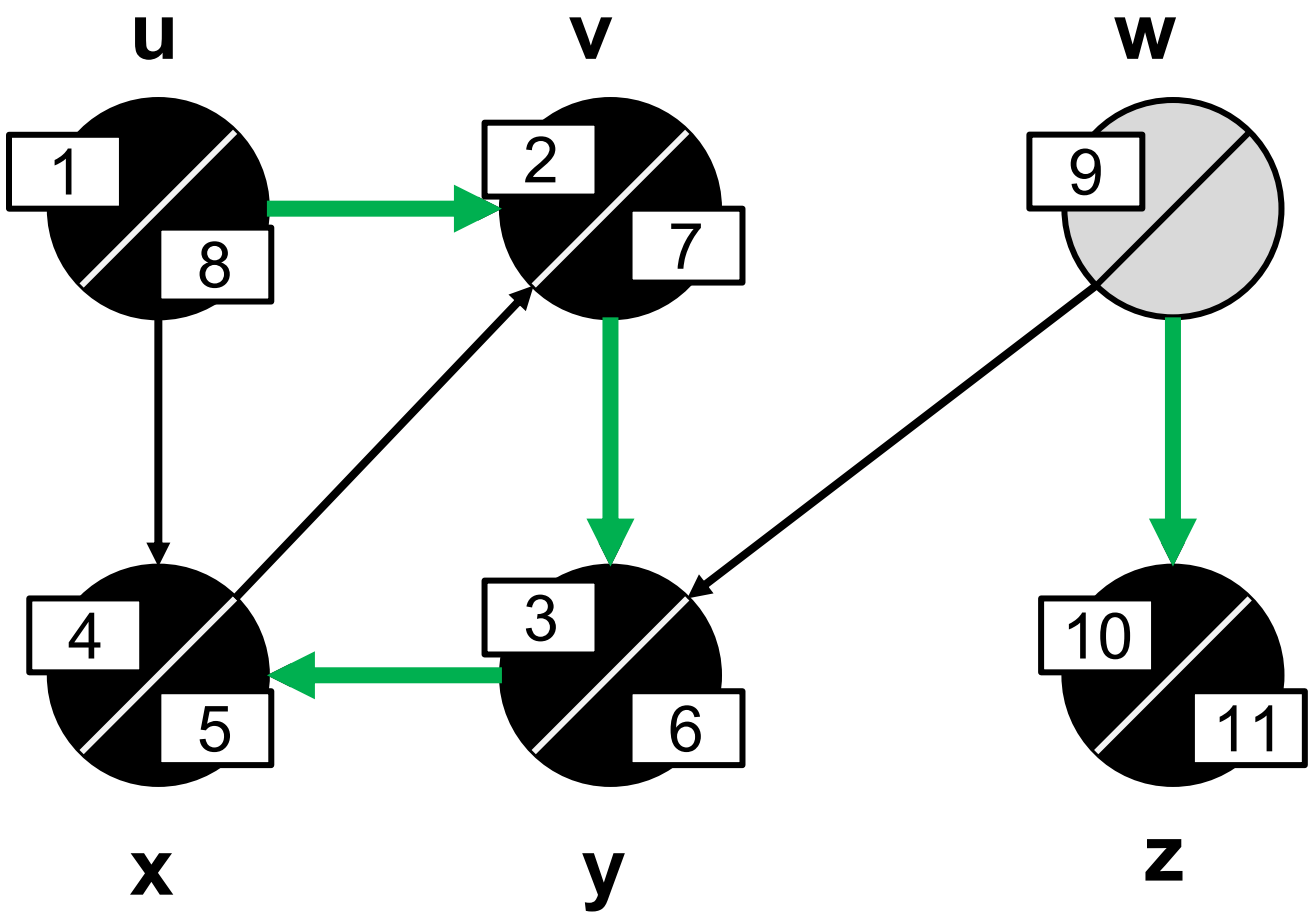

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 10



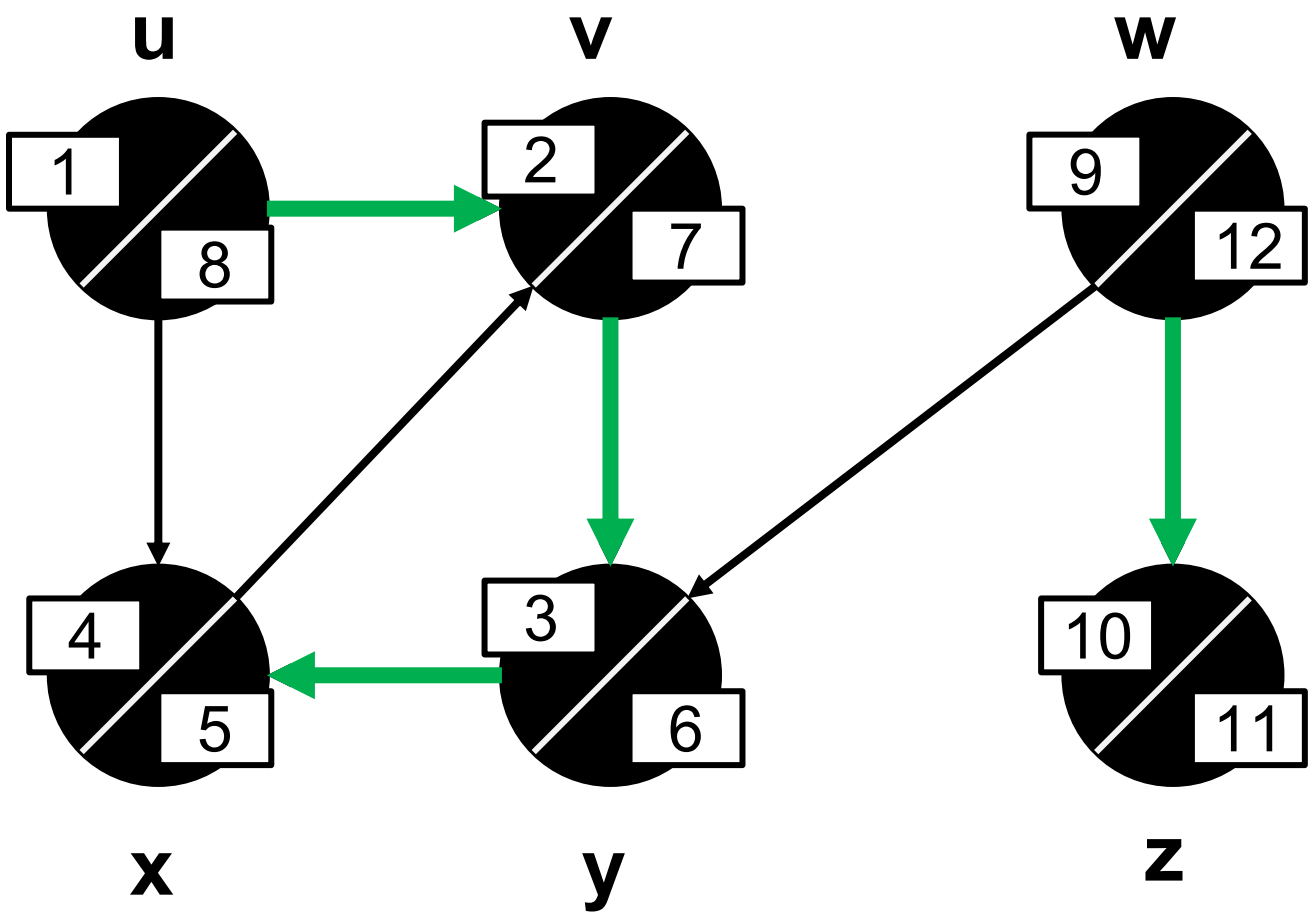

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 11



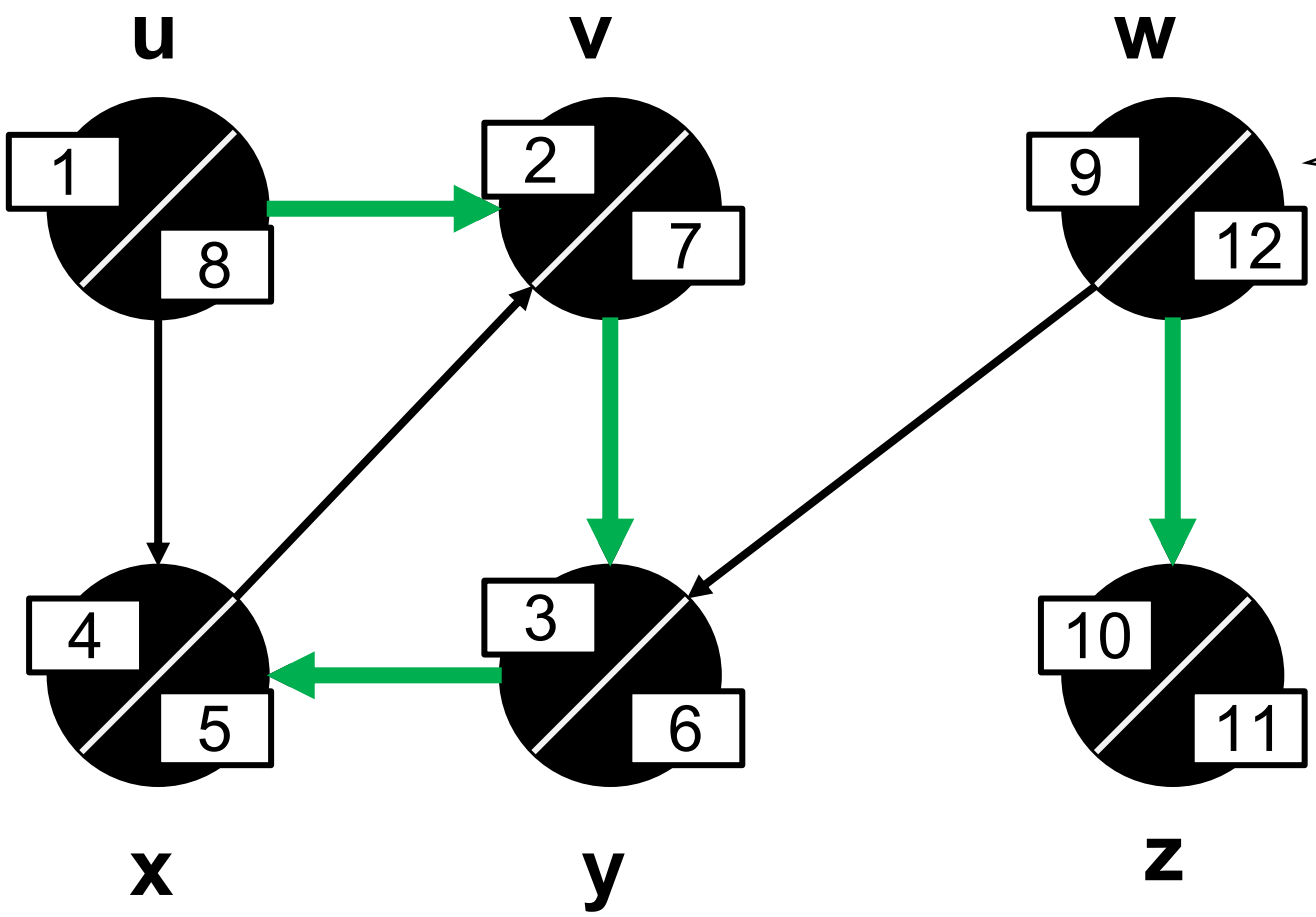

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

time = 12



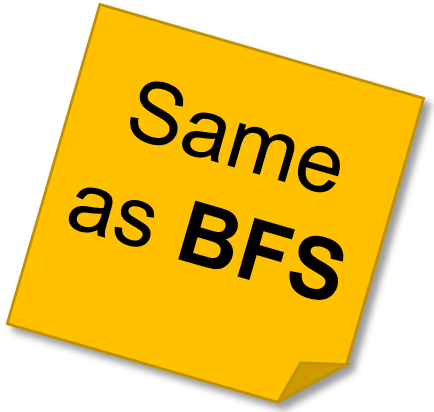

```
DFS_VISIT(G, u):
   colour[u] ← gray
   time ← time + 1
   d[u] ← time
   for each neighbour v of u:
      if colour[v] = white:
         pi[v] ← u
         DFS_VISIT(G, v)
   colour[u] ← black
   time ← time + 1
   f[u] ← time
```

DFS(G) done!



Every vertex has been assigned a **discovery time** and a **finishing time.**

Runtime Analysis



The total amount of work (use **adjacency list**):

- Visit each vertex once
  - constant work per vertex
  - in total: **O(|V|)**
- At each vertex, check all its neighbours (all its incident edges)
  - Each edge is checked once (in a directed graph)
  - in total: **O(|E|)**

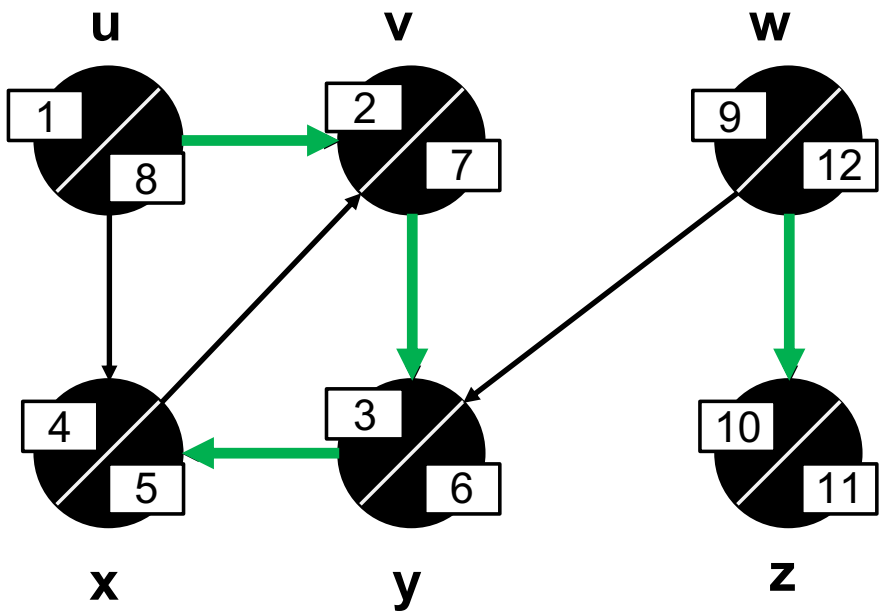
Total runtime: O(|V|+|E|)

7 min

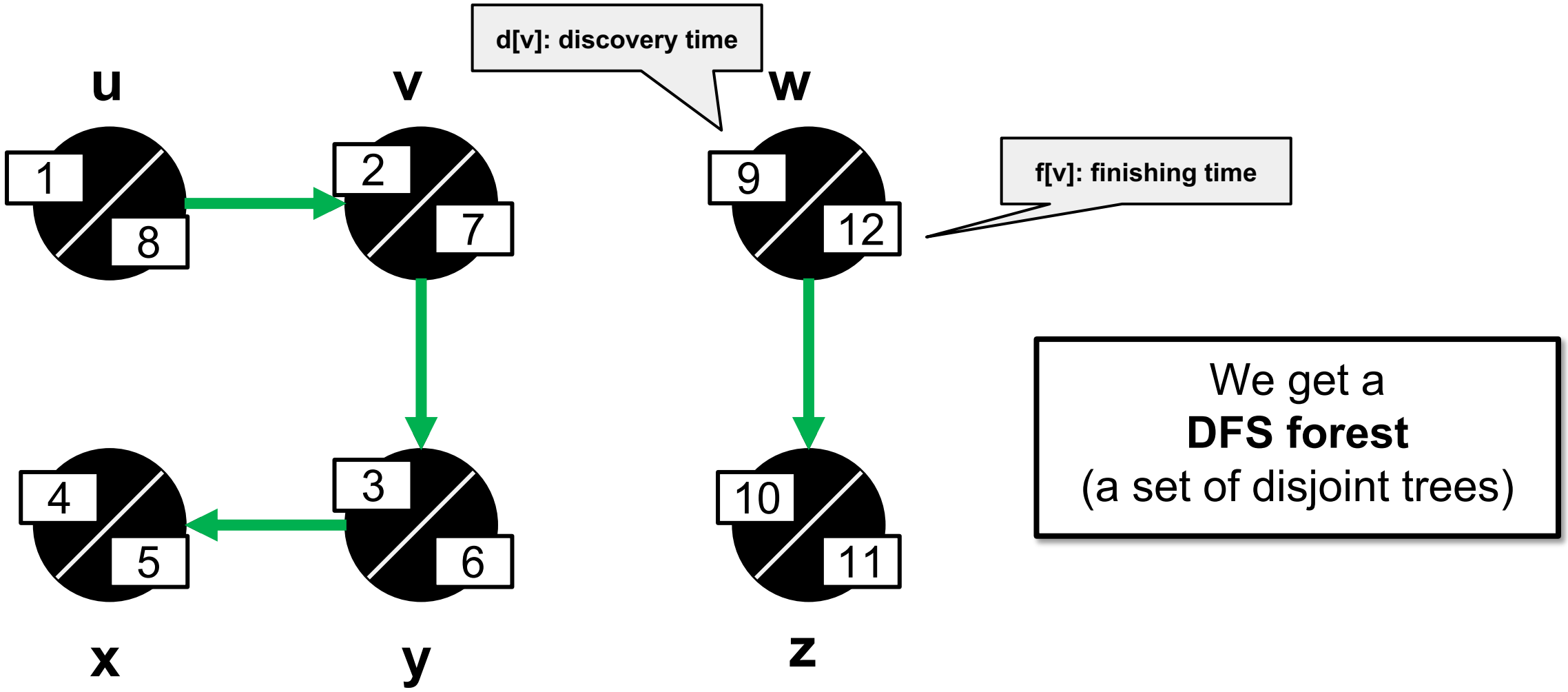
Properties of DFS

- DFS visits every node exactly once
- DFS can tell us whether a graph is connected

Same as **BFS**



Information about Graph Structure



We get a
DFS forest
(a set of disjoint trees)

Information about Graph Structure

- Detect whether a graph has a cycle.

How can we detect a cycle?

determine descendant / ancestor relationship

in the DFS forest

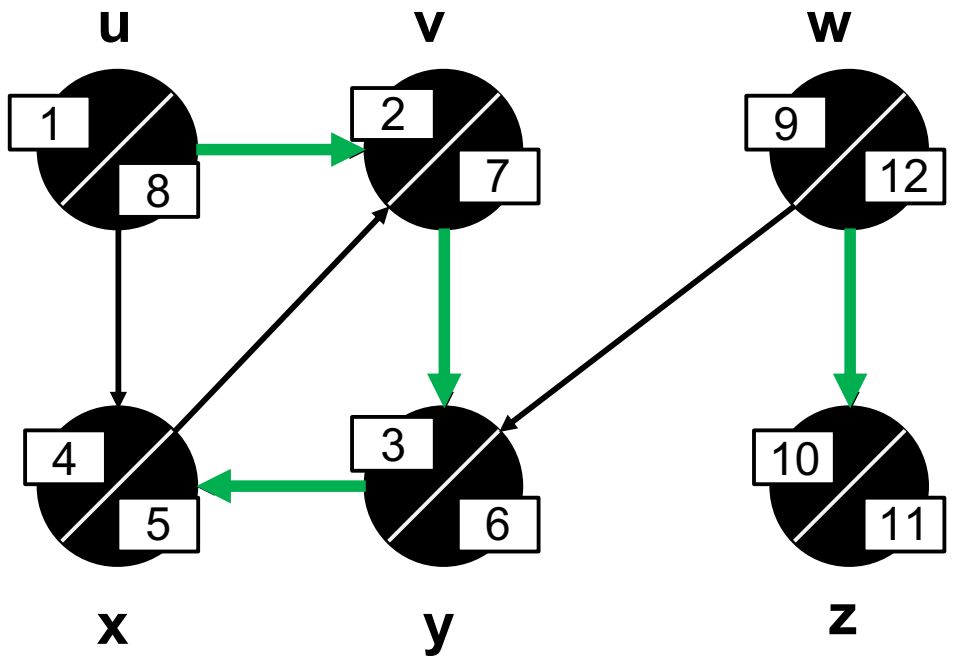
How to decide whether y is a descendant of u in the DFS forest?

Idea #1

Trace back the **pi[v]** pointers (the green edges) starting from **y**, see whether you can get to **u**.

Worst-case takes **O(n)** steps.

Can do better than this…



the parenthesis structure

((( ))) ( ) (( ))

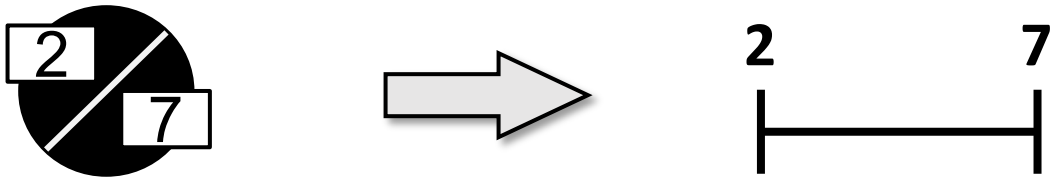
➔ Either one pair **contains** the another pair.
➔ Or one pair is **disjoint** from another

(( ))

This (overlapping) never happens!

Visualize

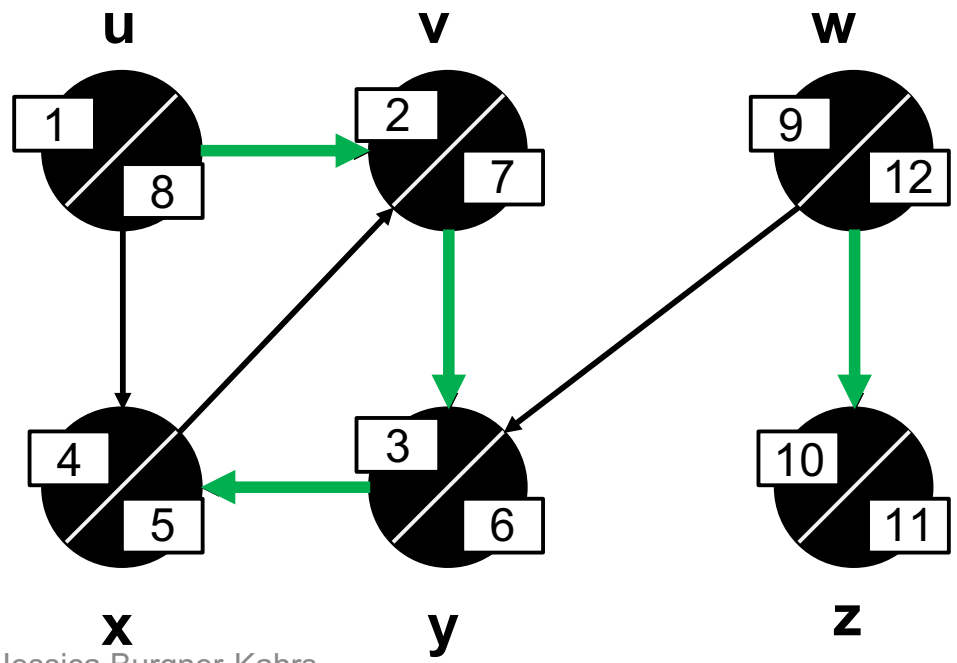
d[v], f[v] as interval [d[v], f[v]]

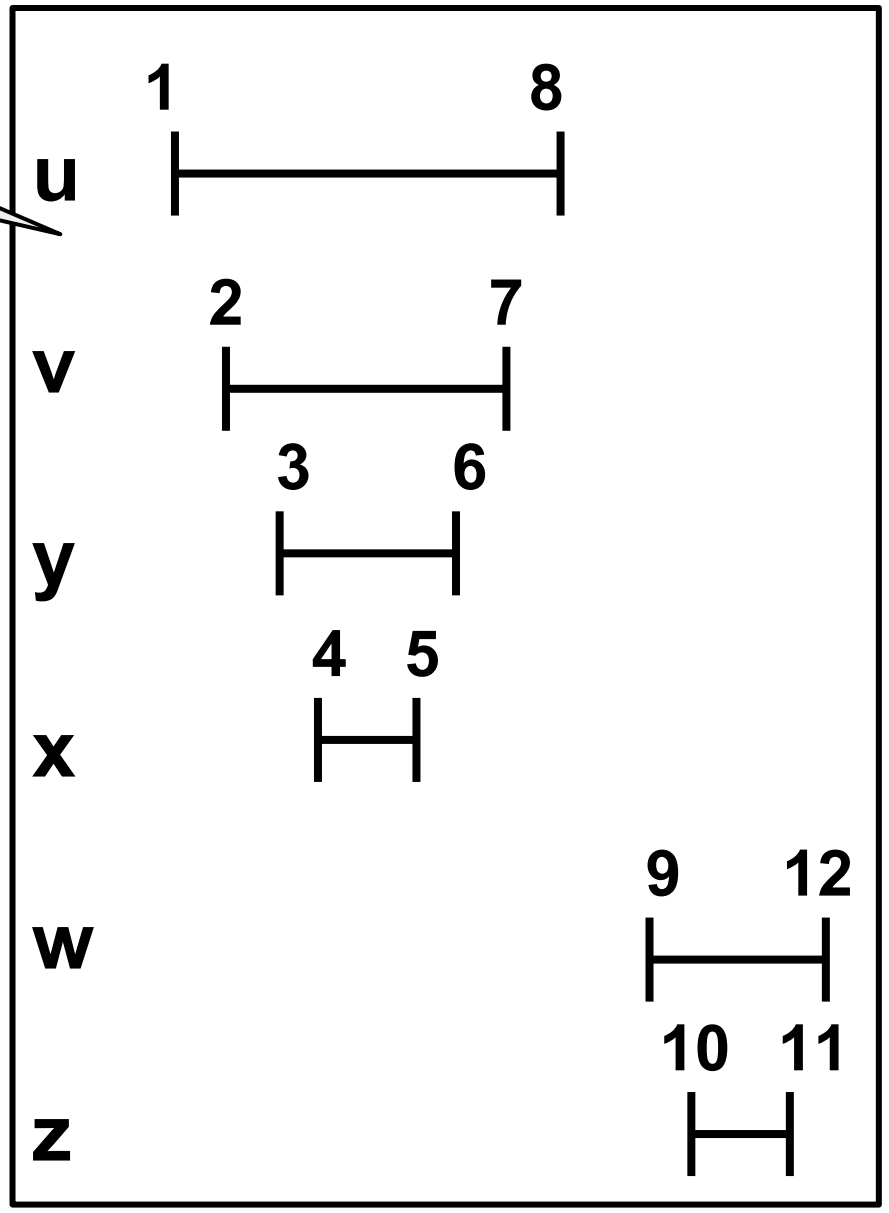


Now, visualize all the intervals!

What do you see in this?

Parenthesis structure!





Parenthesis Theorem

CLRS
Chapter 22.3

In any depth-first search of a (directed or undirected) graph G=(V,E), for any two vertices *u* and *v*, exactly one of the following three conditions holds:

- the intervals [d[u], f[u]] and [d[v],f[v]] are entirely **disjoint**,
  and **neither *u* nor *v* is a descendant of the other** in the depth-first forest,
- the interval [d[u], f[u]] is **contained entirely** within the interval [d[v],f[v]],
  and ***u* is a descendant of *v*** in a depth-first tree, or
- the interval [d[v],f[v]] is **contained entirely** within the interval [d[u], f[u]],
  and ***v* is a descendant of *u*** in a depth-first tree.

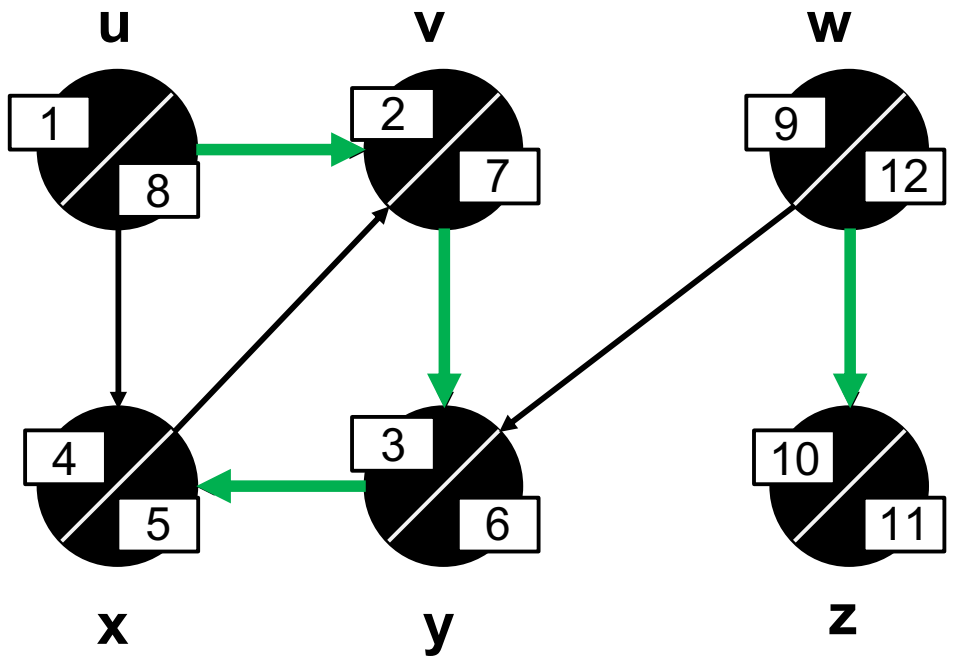
How to decide whether y is a descendant of u in the DFS forest?

Idea #1

Trace back the … green … whether … to **u**.

Worst-… takes **O(n)** steps.

FORGET THIS

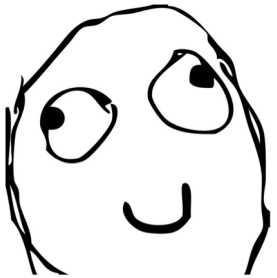


Idea #2

See if [d[u], f[u]] contains [d[y], f[y]].

Worst-case: 1 step!

WAY BETTER!

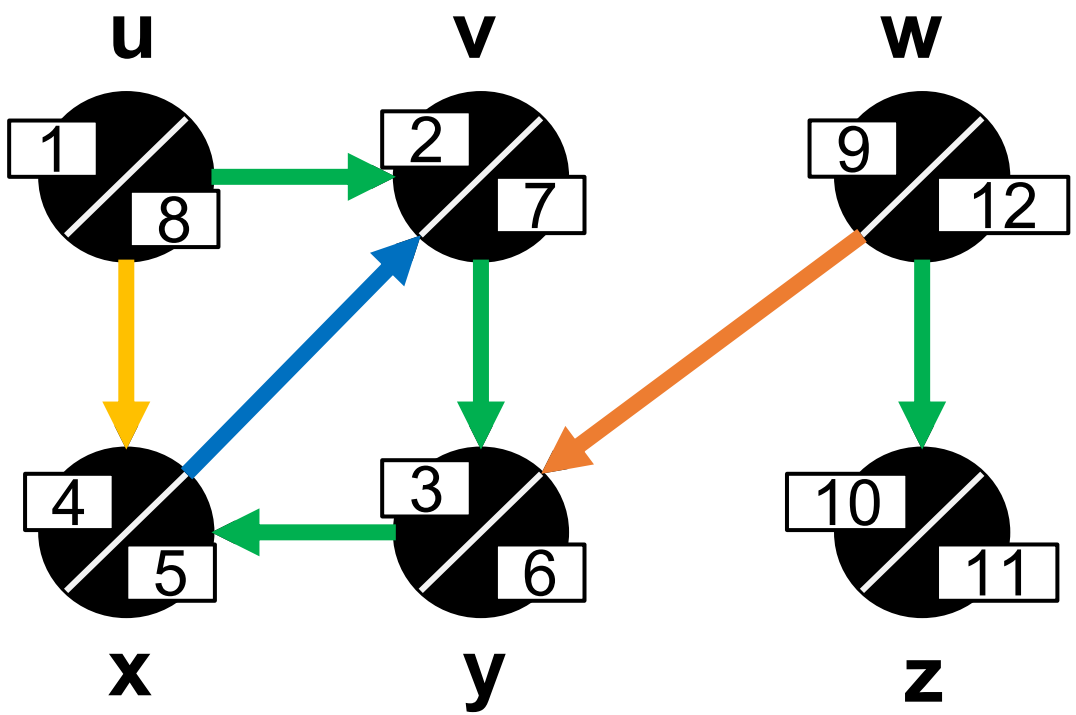


We can efficiently check whether a vertex is an ancestor of another vertex in the DFS forest.

so what...

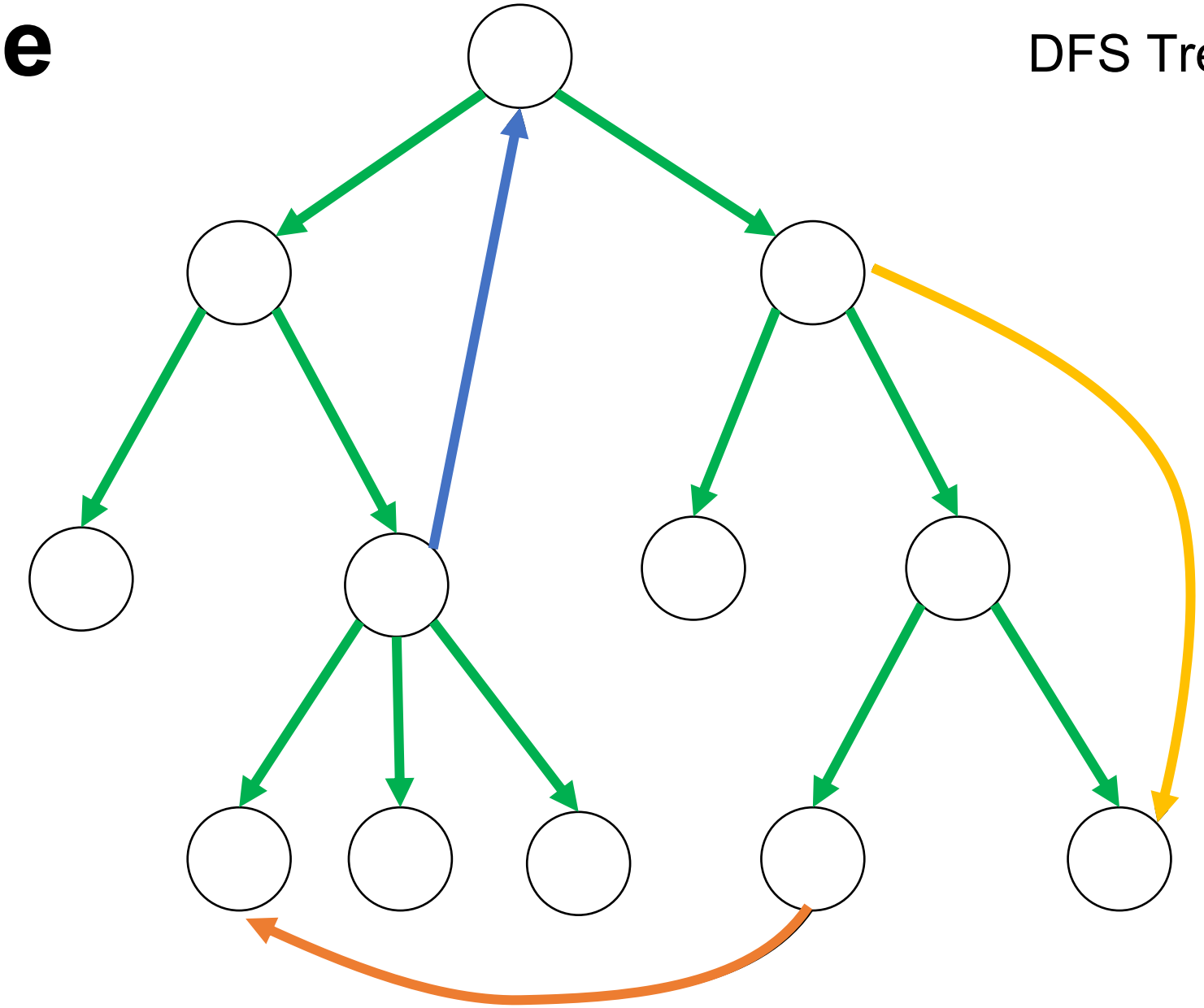
Classifying Edges

4 Types of Edges in a Graph after DFS



- **Tree edge:** an edge in the DFS-forest
- **Back edge:** a non-tree edge pointing from a vertex to its **ancestor** in the DFS forest.
- **Forward edge:** a non-tree edge pointing from a vertex to its **descendant** in the DFS forest
- **Cross edge**: all other edges

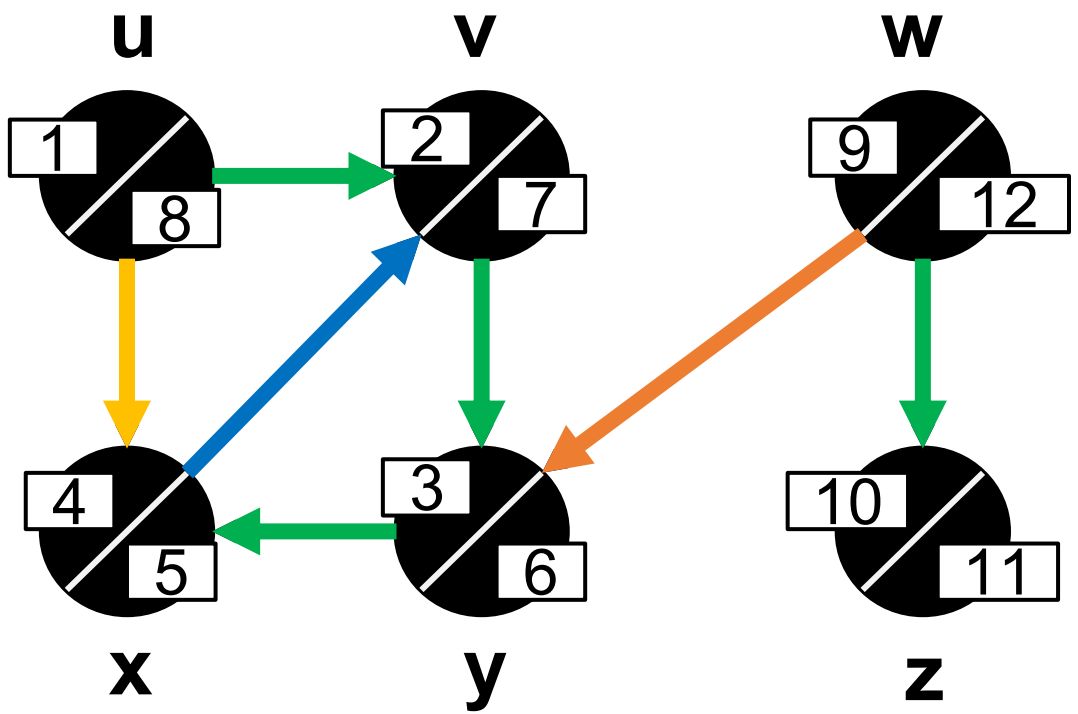
Example

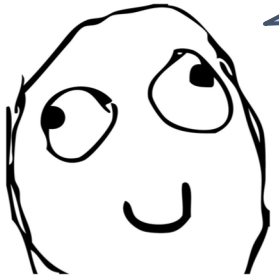


Checking Edge Types

We can efficiently check whether a vertex is an **ancestor / descendant** of another vertex using the **parenthesis structure** of **[d[v], f[v]]** intervals.

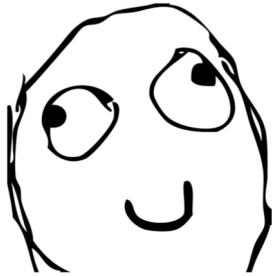
Edge Type of uv	Discovery Times	Finishing Times
Tree edge	d[u] < d[v]	f[u] > f[v]
Back edge	d[u] > d[v]	f[u] < f[v]
Forward edge	d[u] < d[v]	f[u] > f[v]
Cross edge	d[u] > d[v]	f[u] > f[v]





We can efficiently check edge types after a DFS!

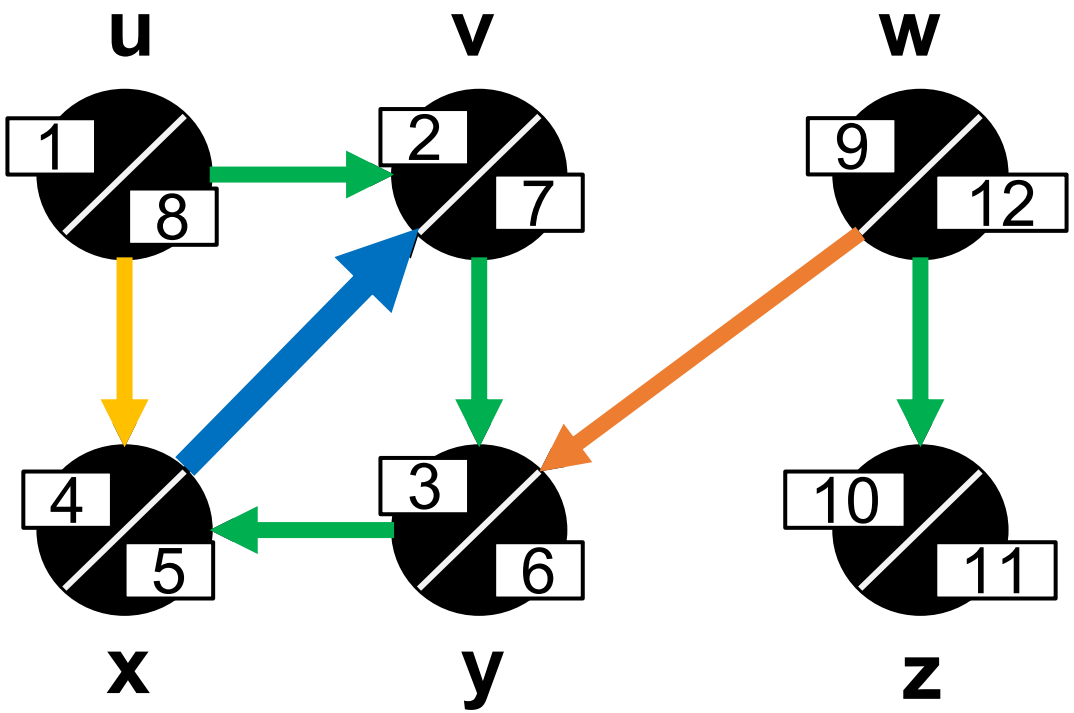
so what...



A graph is cyclic if and only if DFS yields a back edge.

That's useful!

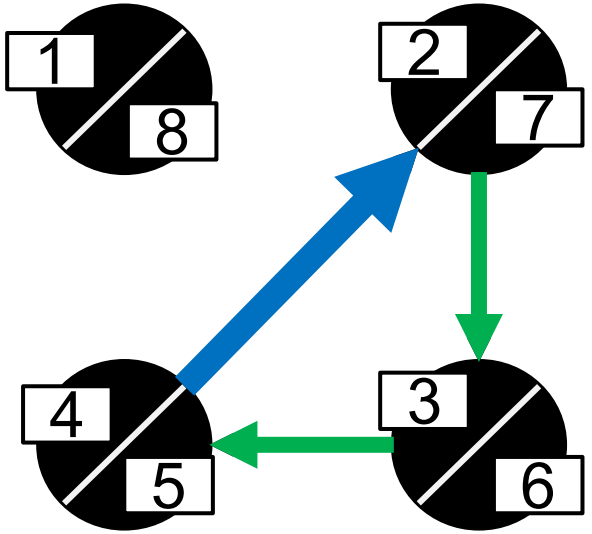
A (directed) graph contains a cycle if and only if DFS yields a back edge.



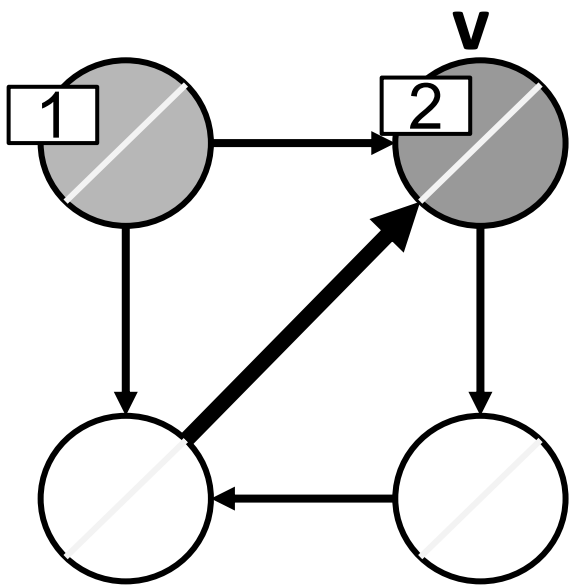
A (directed) graph contains a cycle if and only if DFS yields a back edge.

Proof of "if":

Let the edge be (u, v),
then by definition of back edge,
v is an ancestor of u in the DFS tree,
then there is a path from v to u,
i.e., v → … → u,
using DFS forest edges plus the back edge u → v,
which is a Cycle.

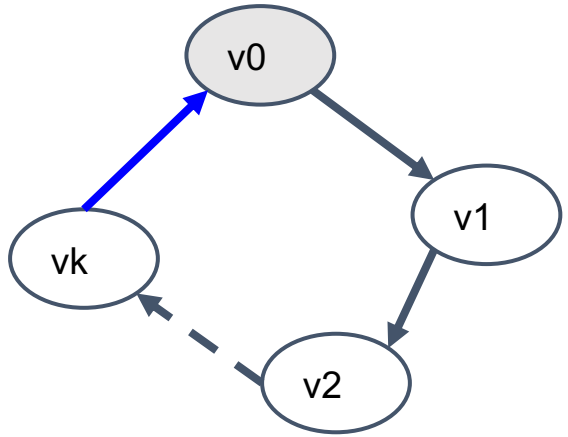


A (directed) graph contains a cycle if and only if DFS yields a back edge.



Proof of "only if":

Let the cycle be...,



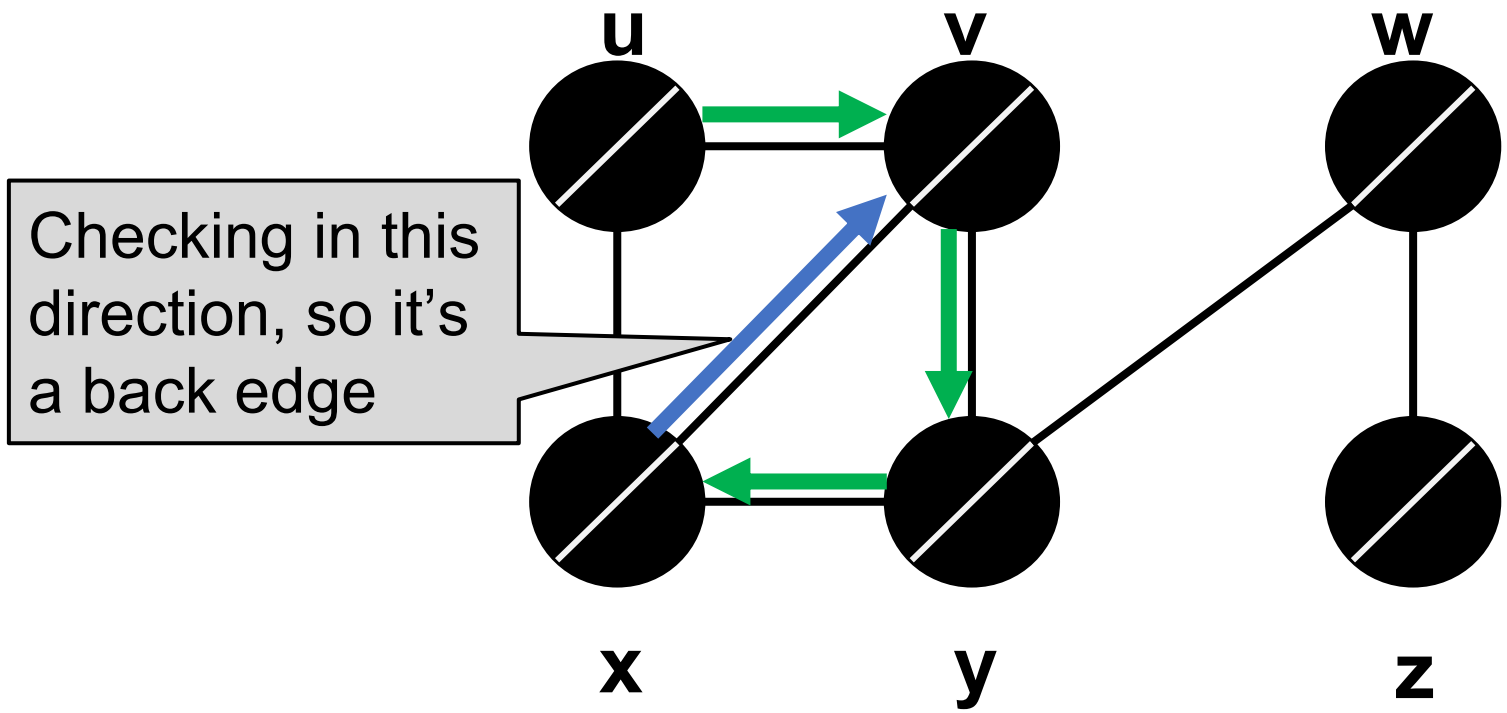
Let v0 be the first one that turns gray, when all others in the cycle are white, then vk must be a descendant of v0. (Read "White Path Theorem" in CRLS)

How about undirected graph?

Should **back** and **forward** edges be the same thing?

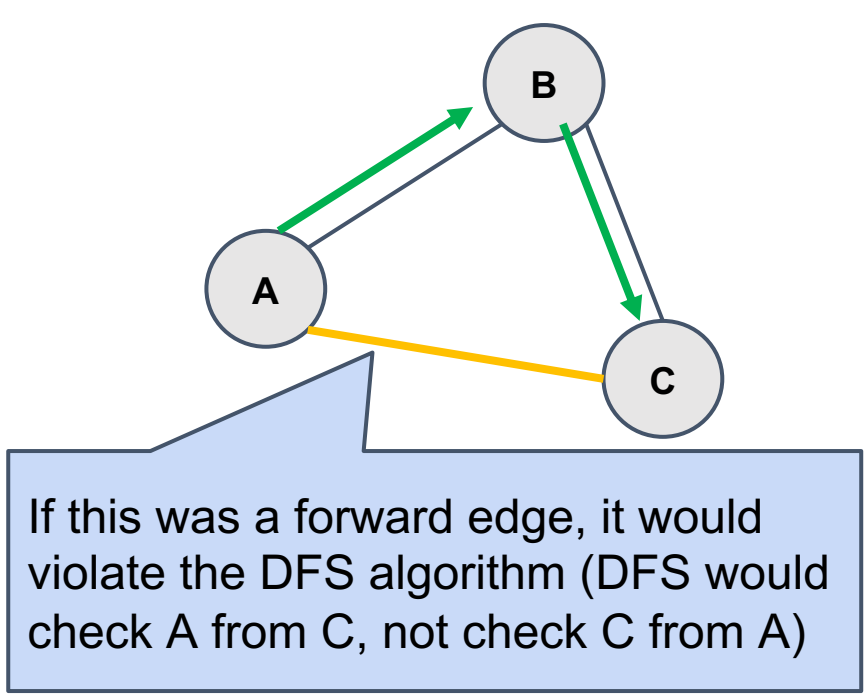
➔ No, because although the edges are undirected, **neighbour checking** still has a “direction”.

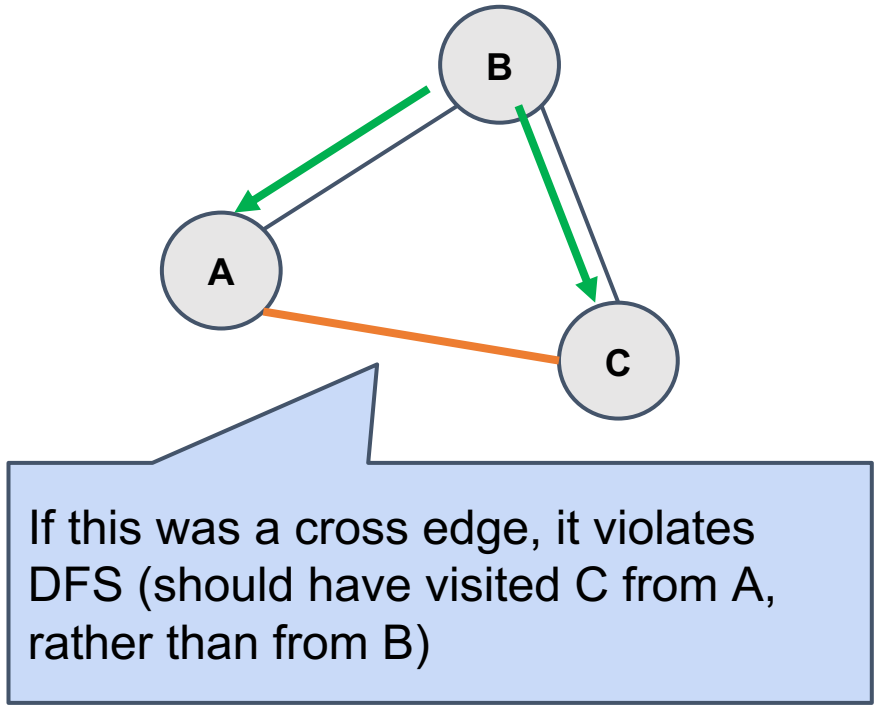
Classify an edge as the **first** type in the classification list that applies.



Undirected Graph: Edge Types

After a DFS on a undirected graph, **every edge** is either a **tree edge** or a **back edge**, i.e., **no** forward edges or cross edges exist.





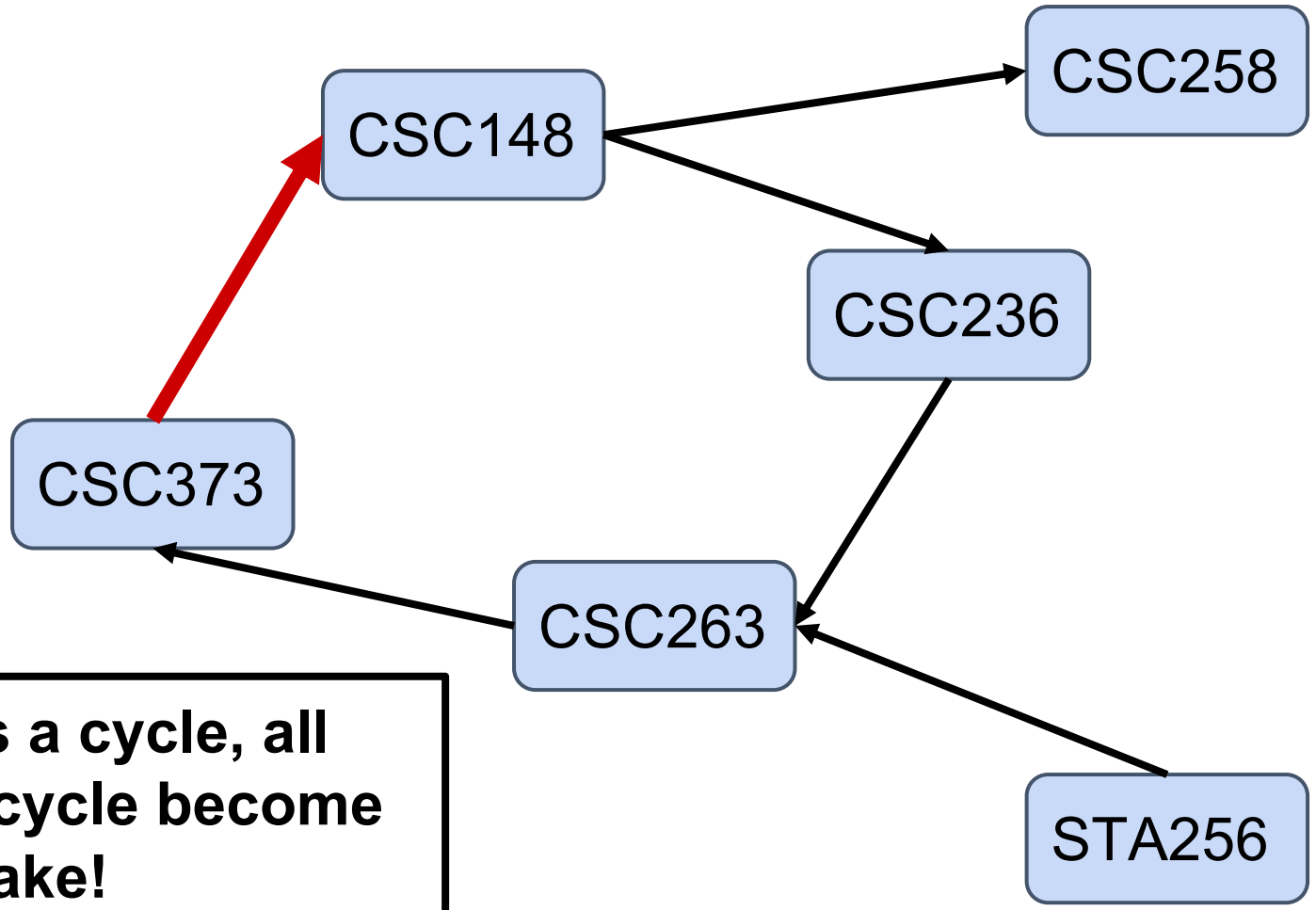
Why do we care about cycles in a graph?

Because cycles have meaningful implication in real applications.

Applications of DFS

- Detect cycles in a graph (CLRS 22.3)
- Topological sort (CLRS 22.4)
- Strongly connected components (CLRS 22.5)

Example: A Course Prerequisite Graph

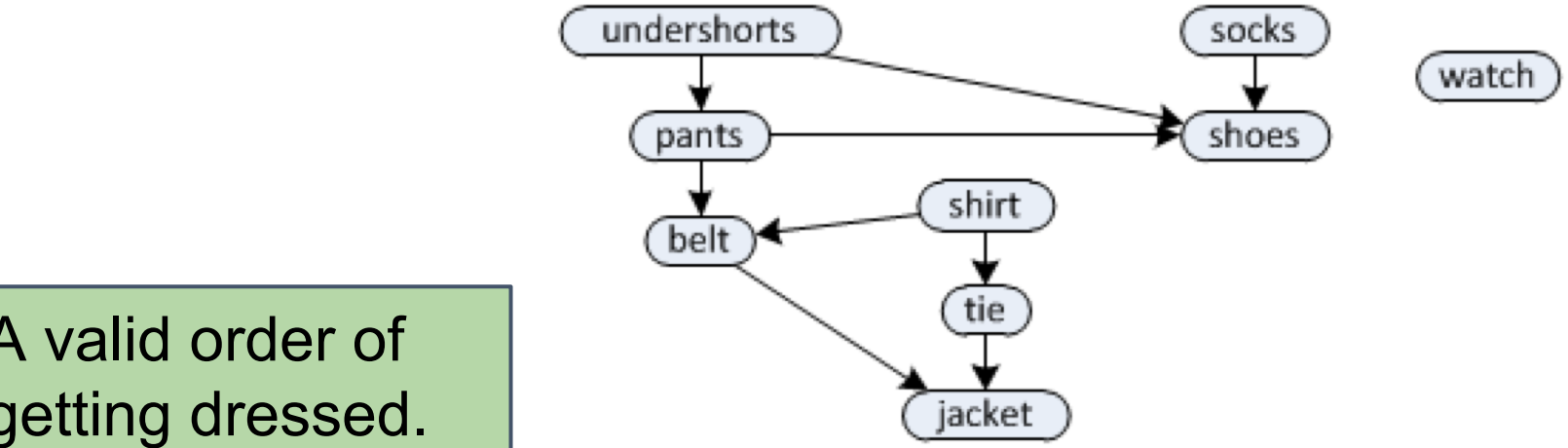


If the graph has a cycle, all courses in the cycle become impossible to take!

Topological Sort

Topological sort is useful for scheduling jobs that have dependencies between each other.

- Vertices are tasks, edges are dependencies
- **Goal**: Order the vertices so that the tasks can all be completed without violating dependencies.



How to do Topological Sorting

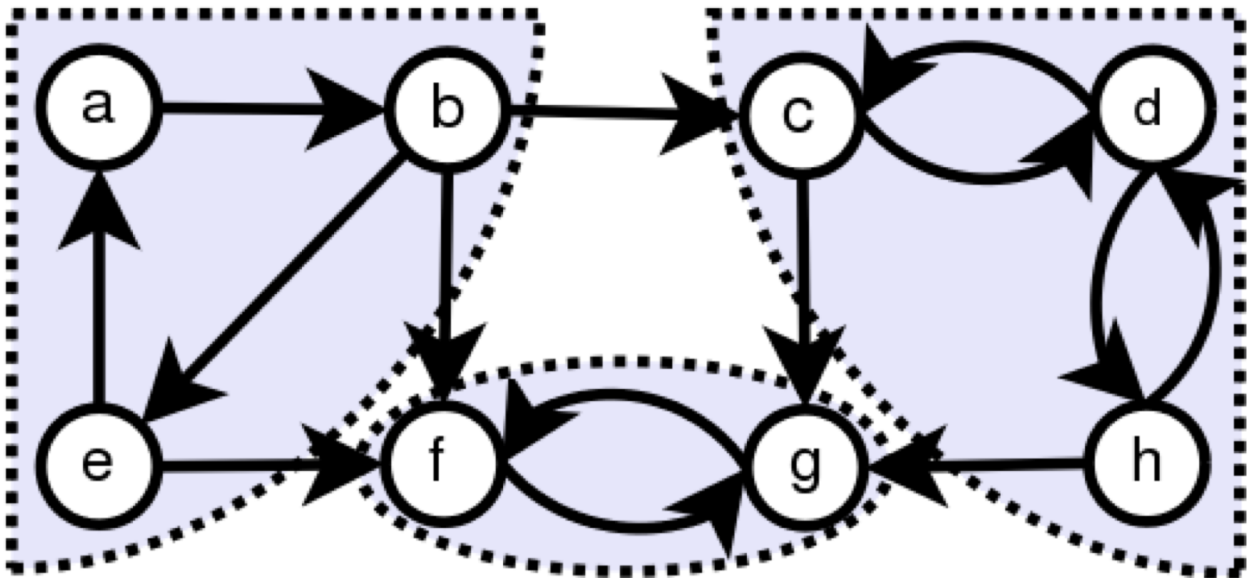
1. Do a DFS
2. Order vertices according to their finishing times f[v]

Read CLRS 22.4 for more details.

Other methods for topological sort also exist, such as Kahn's algorithm https://en.wikipedia.org/wiki/Topological_sorting

Strongly Connected Components

➔Subgraphs with strong connectivity (any pair of vertices can reach each other)



How to Compute Strongly Connected Components?

1. Call **DFS**(G) to compute finishing times for each vertex
2. Compute G^T
3. Call **DFS**(G^T), considering the vertices in decreasing order f[u] from line 1
4. Each Tree in DFS Forest from line 3 is a strongly connected component

G=(V,E)

Transpose of G

$G^T=(V,E^T)$ with

$E^T=\{(u,v): (v,u) \in E\}$

Read CLRS 22.4 for more details.

Summary of DFS

- It's the twin of BFS (Queue vs Stack)
- Keeps two timestamps: d[v] and f[v]
- Has same runtime as BFS
- Does **NOT** give us shortest-path
- Allows for cycle detection (back edge)
- Useful applications such as topological sort and strongly connected components
- For real problems, choose BFS and DFS wisely

The Show is a massive celebration of a year in CS at UTM, happening March 20th. This is a great opportunity to destress before exams from all the hard work you guys have put in with a ton of exciting games, activities, workshops, competitions, and projects!

The best part is free food and free admission, just make sure you get your ticket soon because they are going fast!

The event coordinators would love to answer any and all questions!
Come by DH2014 or message @utmmcss.