CSC263 Week 9

Larry Zhang

Announcements

- Midterm, class average 62.5%   (37.5/60)
- PS7 out soon, due next Tuesday
- A2 out, due March 31, start early!
- Don’t forget to give feedback (especially about the midterm)
    - ***http://goo.gl/forms/S9yie3597B***

Recap

- The Graph ADT
  - definition and data structures
- BFS
  - gives us single-source **shortest** path
  - Let **δ(s, v)** denote the length of shortest path from **s** to **v**...
  - then after performing a BFS starting from **s**, we have, for all vertices v

$$\mathbf{d[v] = \delta(s, v)}$$

We can totally prove it.

Idea of the proof

There is no way $d[v] < \delta(s, v)$, according to Lemma 22.2

Use contradiction: suppose there exist v s.t. $d[v] > \delta(s, v)$, let v be the one with the **minimum** $\delta(s, v)$.

Then on a shortest path between s and v, pick vertex u which is immediately before v...

then we have $\mathbf{d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1}$

Must be equal because u is on the shortest path from s to v.

Must be equal because v is the minimum $\delta(s, v)$ that violates $d[v] > \delta(s, v)$, so u must not be violating.

Think about the moment after dequeue u (checking u’s neighbours)

➔ if v is white, $d[v] = d[u] + 1$ (how BFS works), **contradiction**!

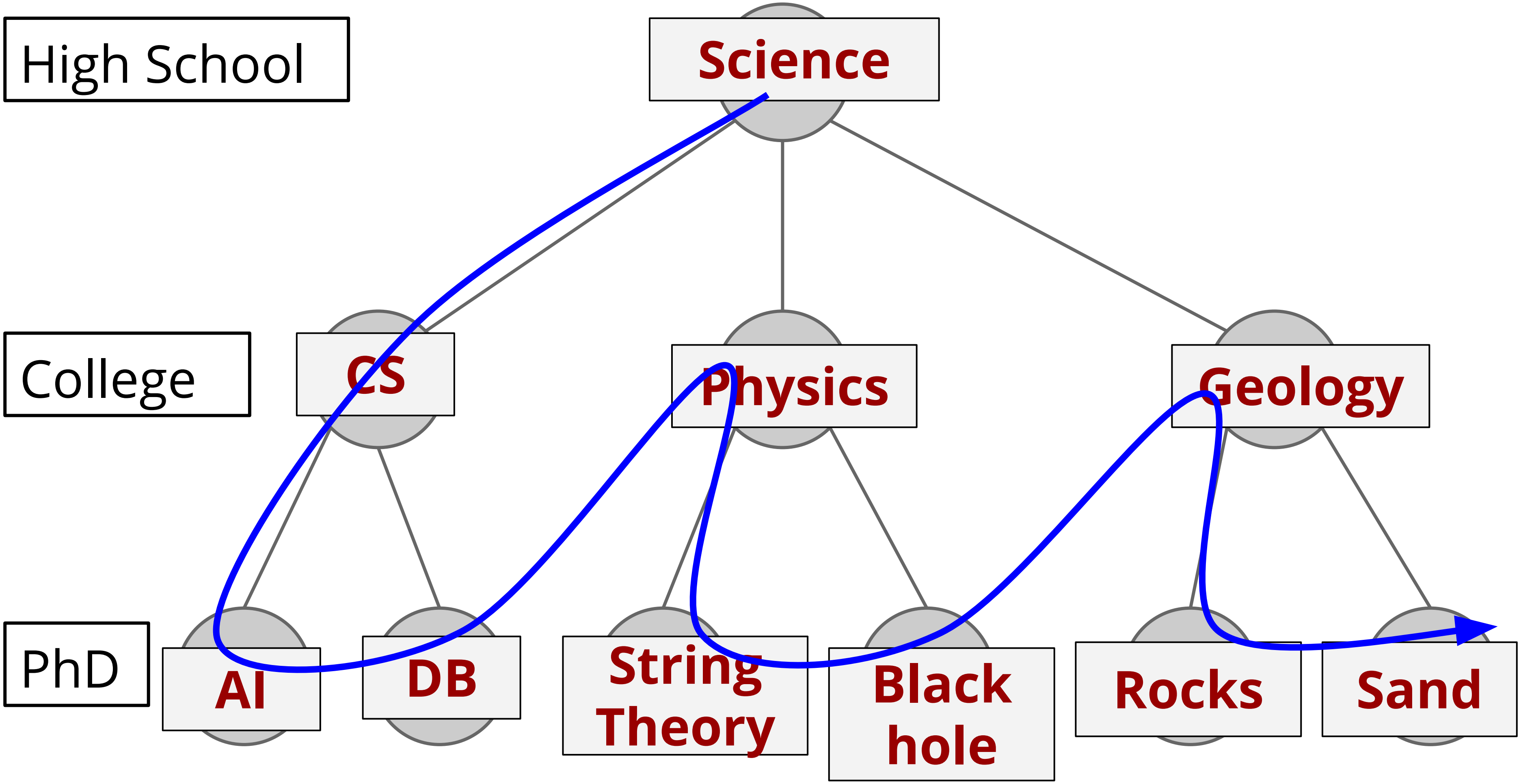
➔ if v is black, $d[v] <= d[u]$ (coz v is dequeued before u), **contradiction**!

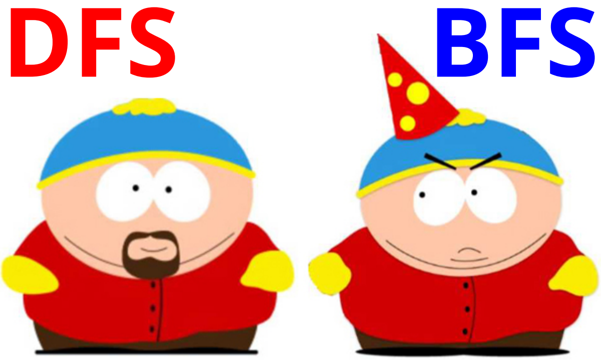
➔ if v is gray, then it is coloured gray by some other vertex w, then $d[v] = d[w] + 1$ and $d[w] <= d[u]$, therefore $d[v] <= d[u] + 1$, **contradiction**!

Depth-First Search

The Depth-First way of learning these subjects

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





```
NOT_YET_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)
```

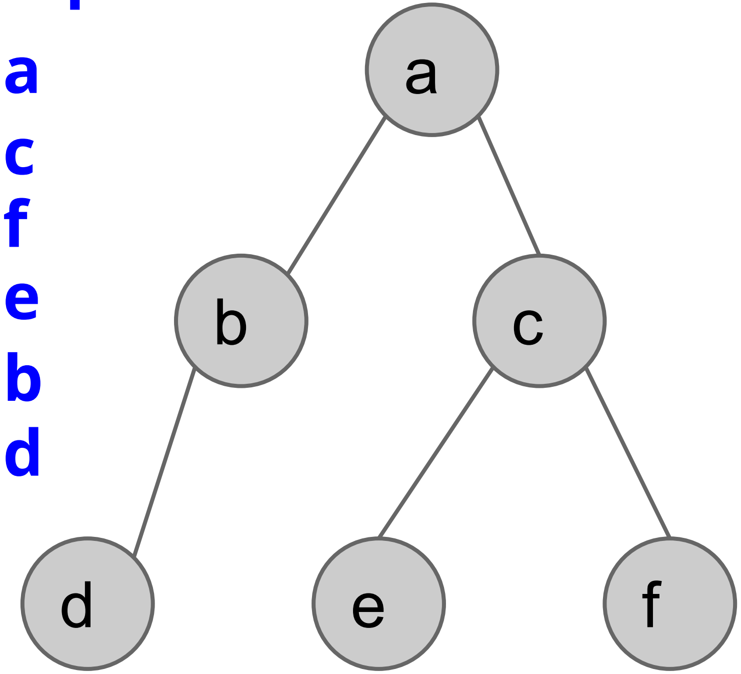
```
NOT_YET_DFS(root):
  Q ← Stack()
  Push(Q, root)
  while Q not empty:
    x ← Pop(Q)
    print x
    for each child c of x:
      Push(Q, c)
```

Why they are twins!

DFS in a tree

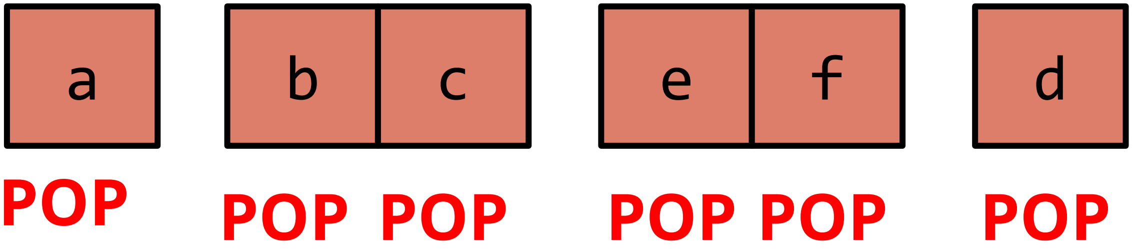
Output:

a
c
f
e
b
d



```
NOT_YET_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:



a
POP

b c
POP POP

e f
POP POP

d
POP

A nicer way to write this code?

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

```
NOT_YET_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_YET_DFS(root):
  print root
  for each child c of x:
      NOT_YET_DFS(c)
```

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

Avoid visiting a vertex twice, same as BFS

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

➔ **White**: “unvisited”

➔ **Gray**: “encountered”

➔ **Black**: “explored”

➔ Initially all vertices are **white**

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

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

➔ Avoid visiting **gray** or **black** vertices

➔ In the end, all vertices are **black**

Other values to remember, some are same as BFS

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

Other values to remember, different from BFS

- There is a **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
  - **f[v]**: "finishing time", when all the vertex's neighbours have been visited.

Note : this d[v] is totally different from that distance value d[v] in BFS!

The pseudo-code (incomplete)

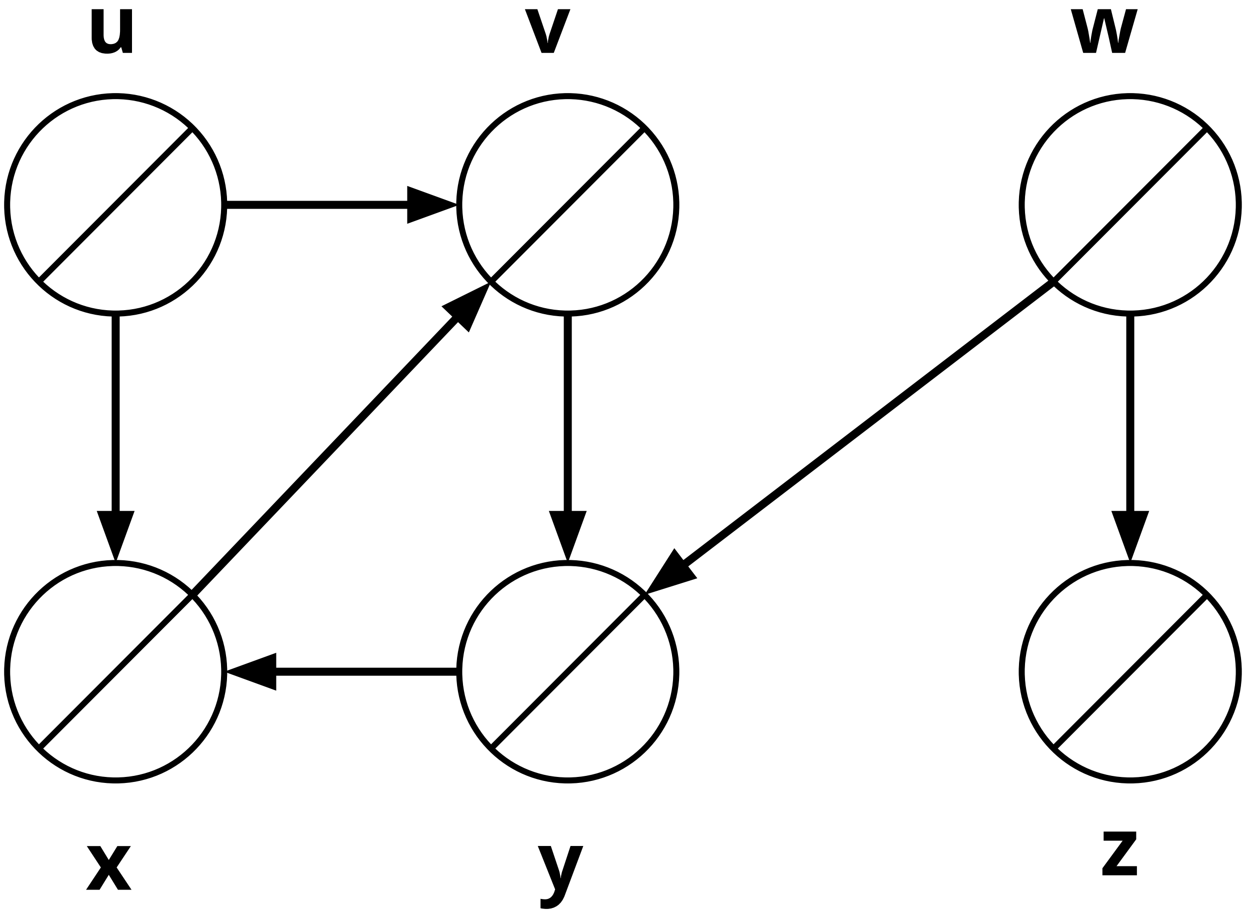
The red part is the same as NOT_YET_DFS

```
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
```

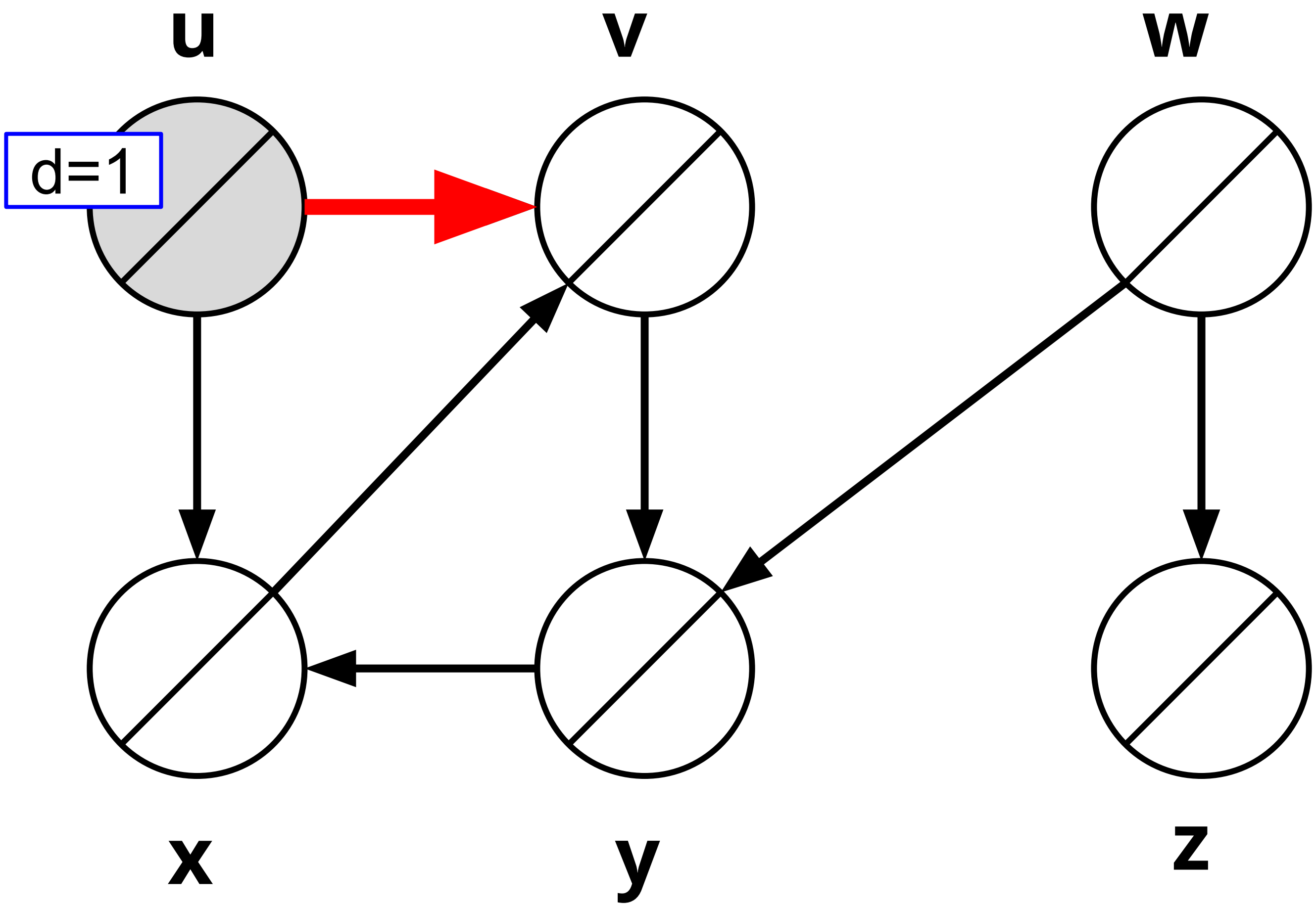
Why **DFS_VISIT** instead of **DFS**? We will see…

Let’s run an example!

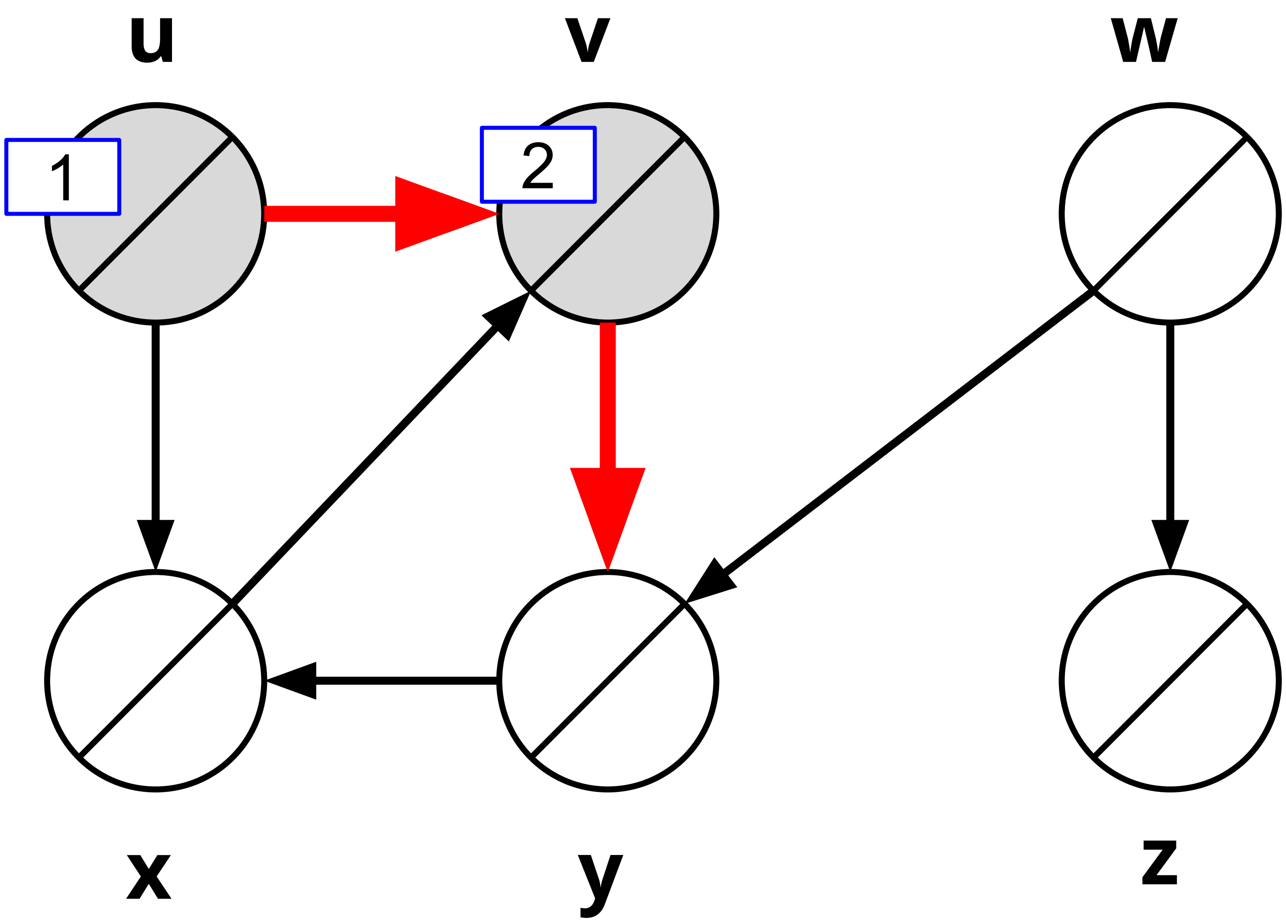
DFS_VISIT(G, u)



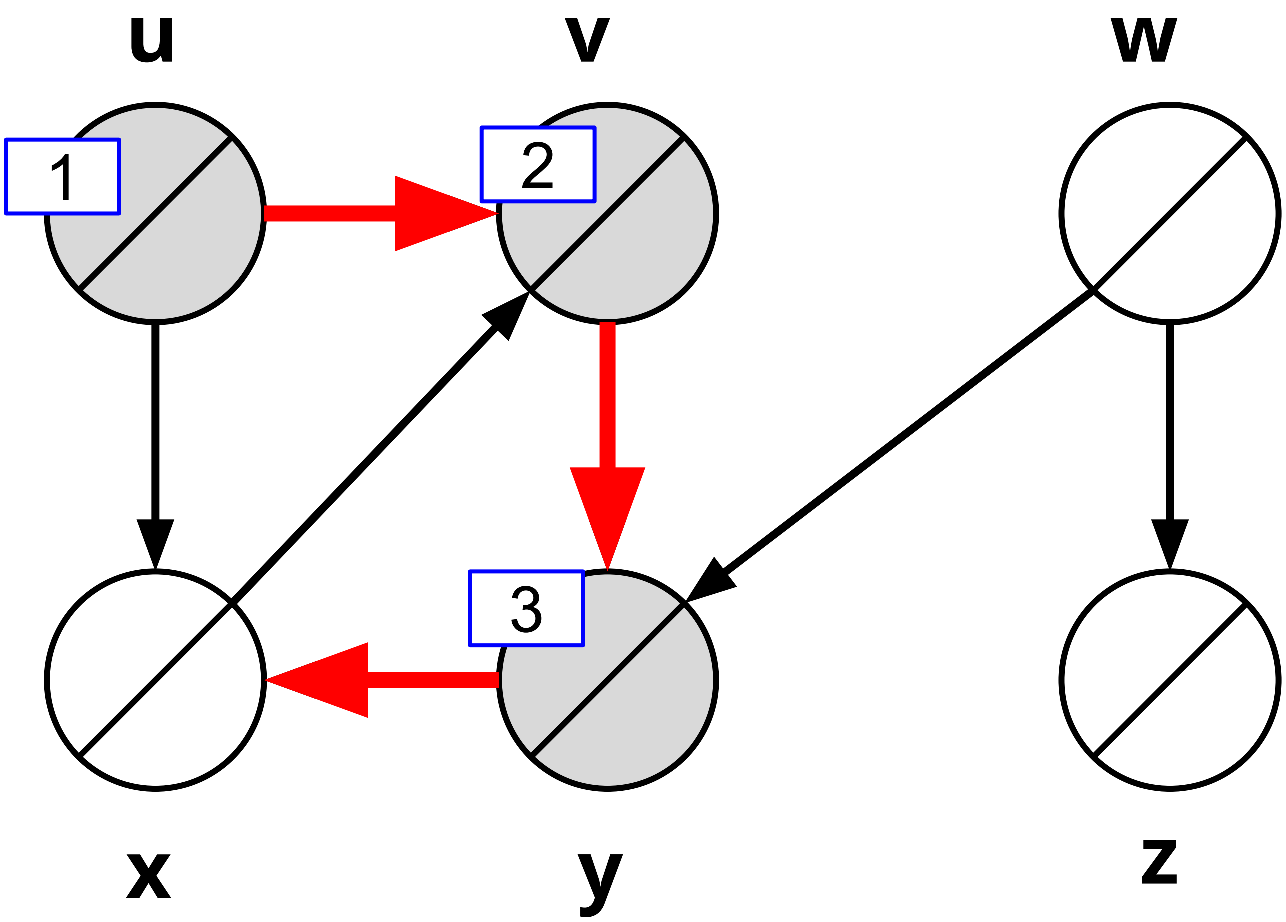
time = 1, encounter the source vertex



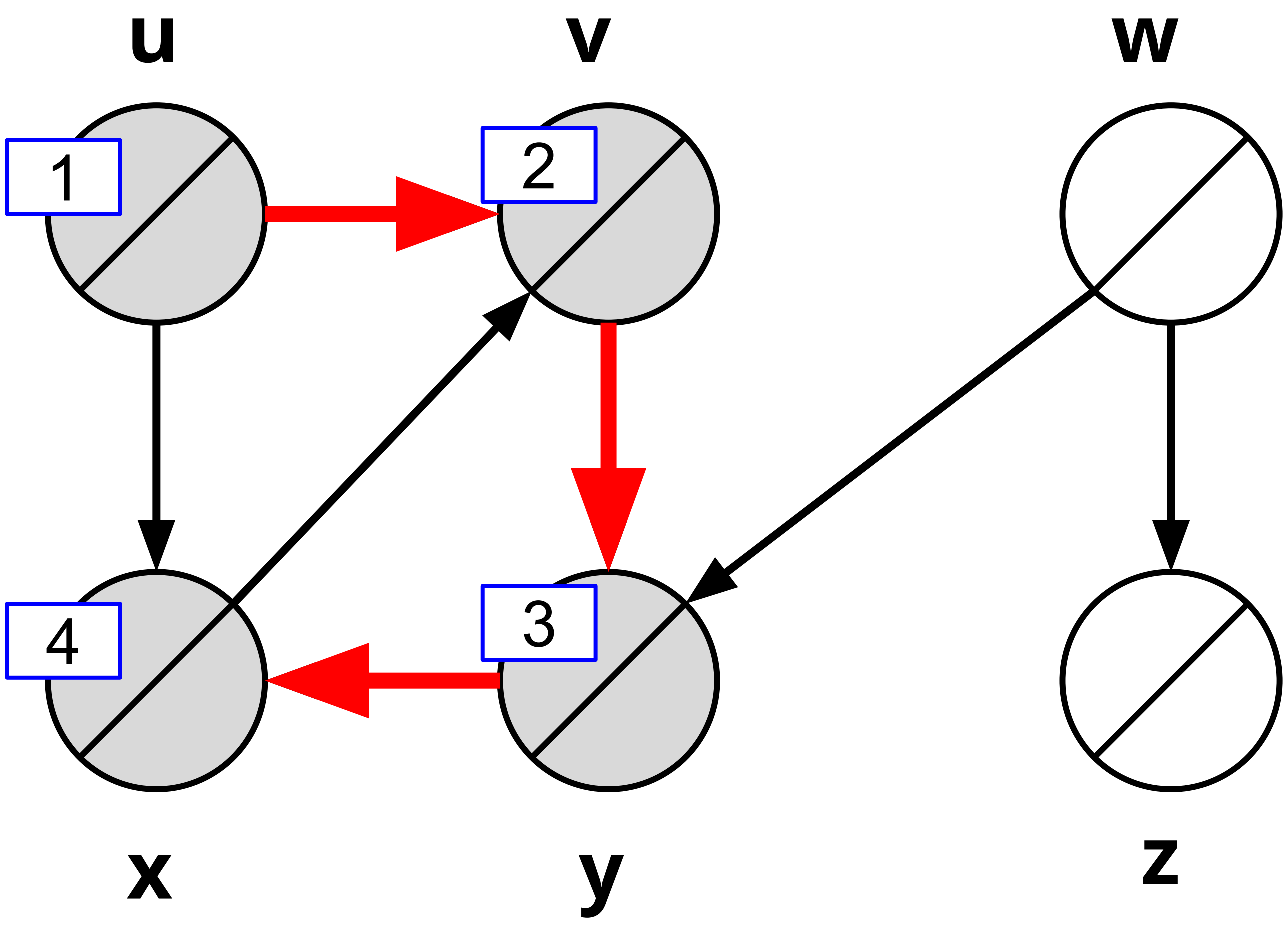
time = 2, recursive call, level 2



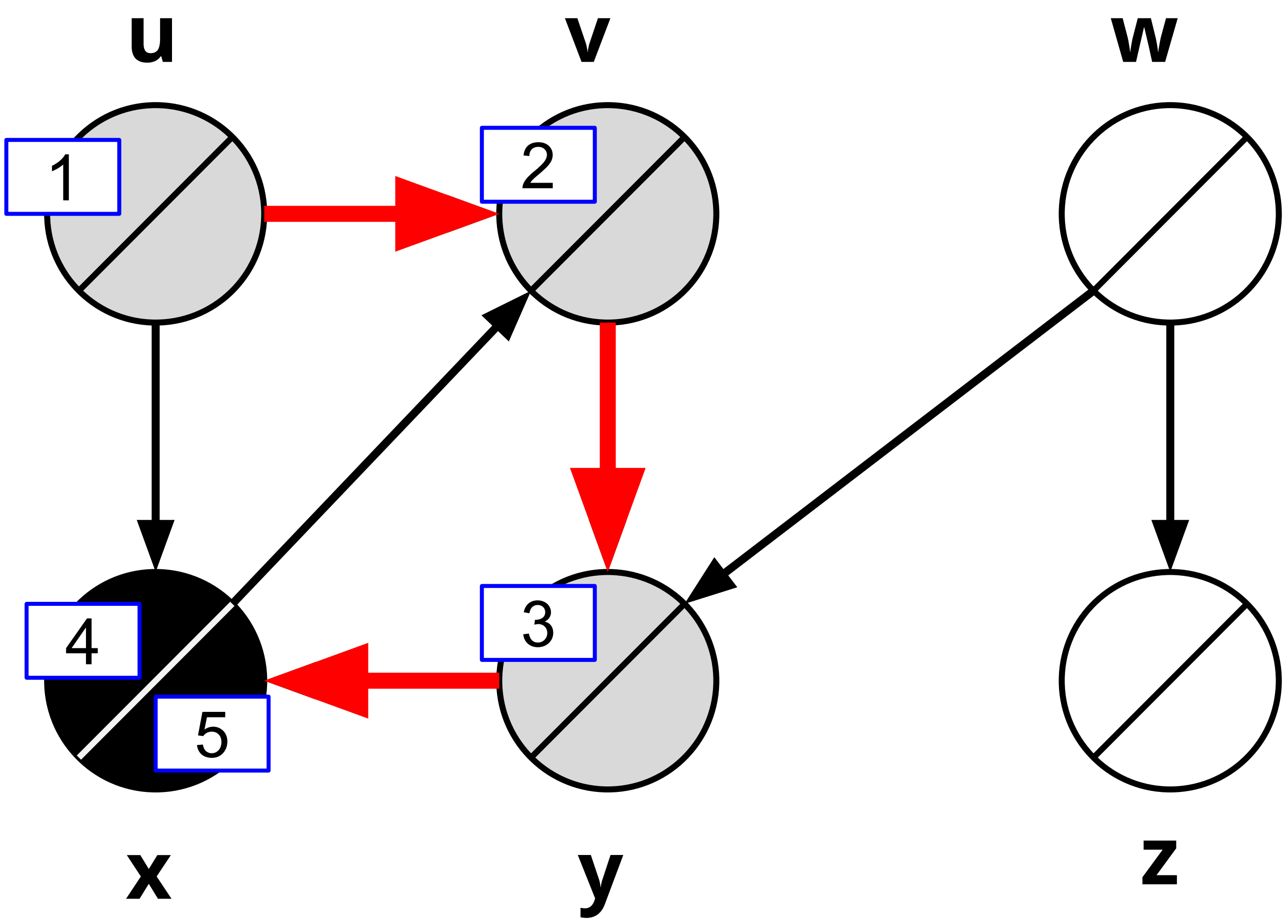
time = 3, recursive call, level 3



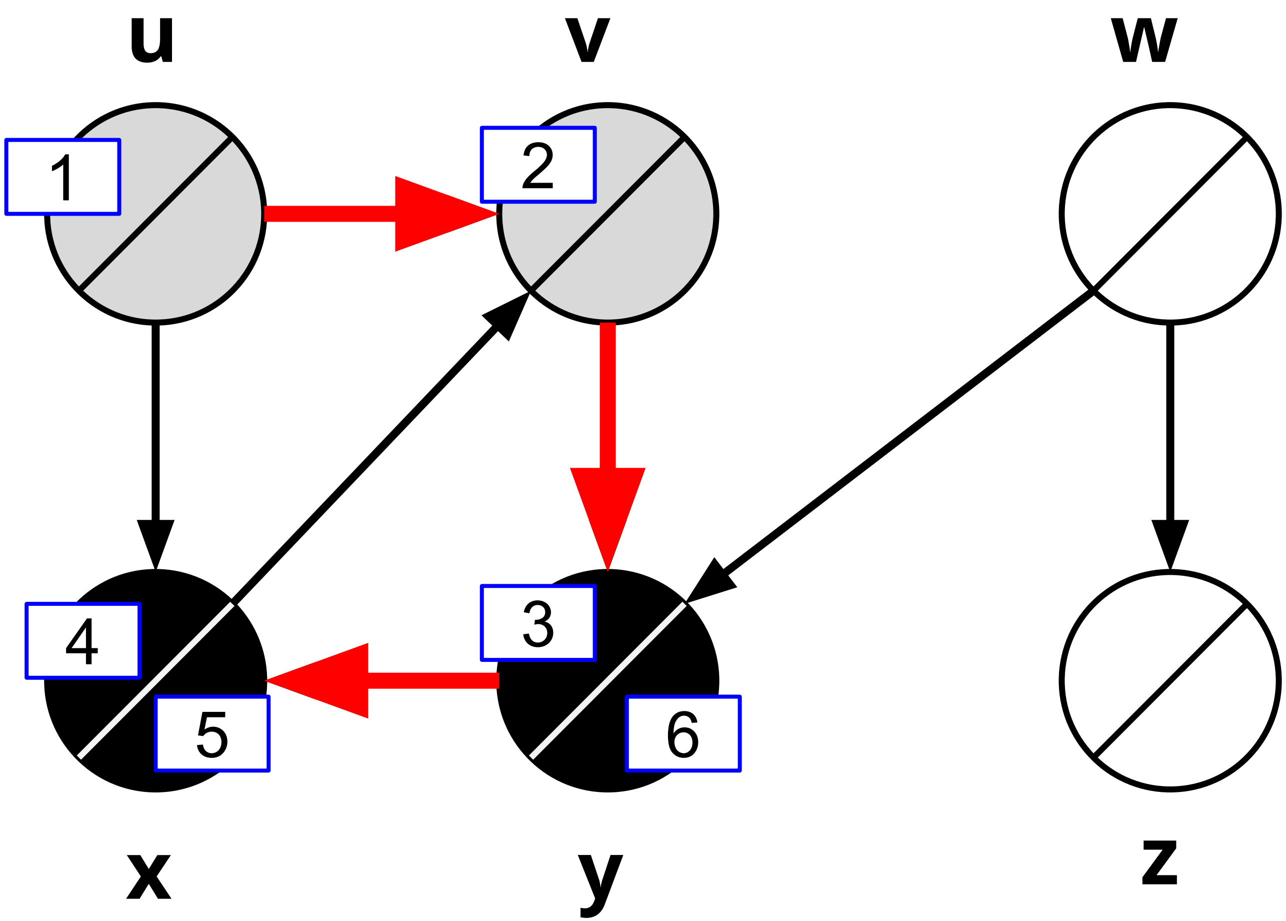
time = 4, recursive call, level 4



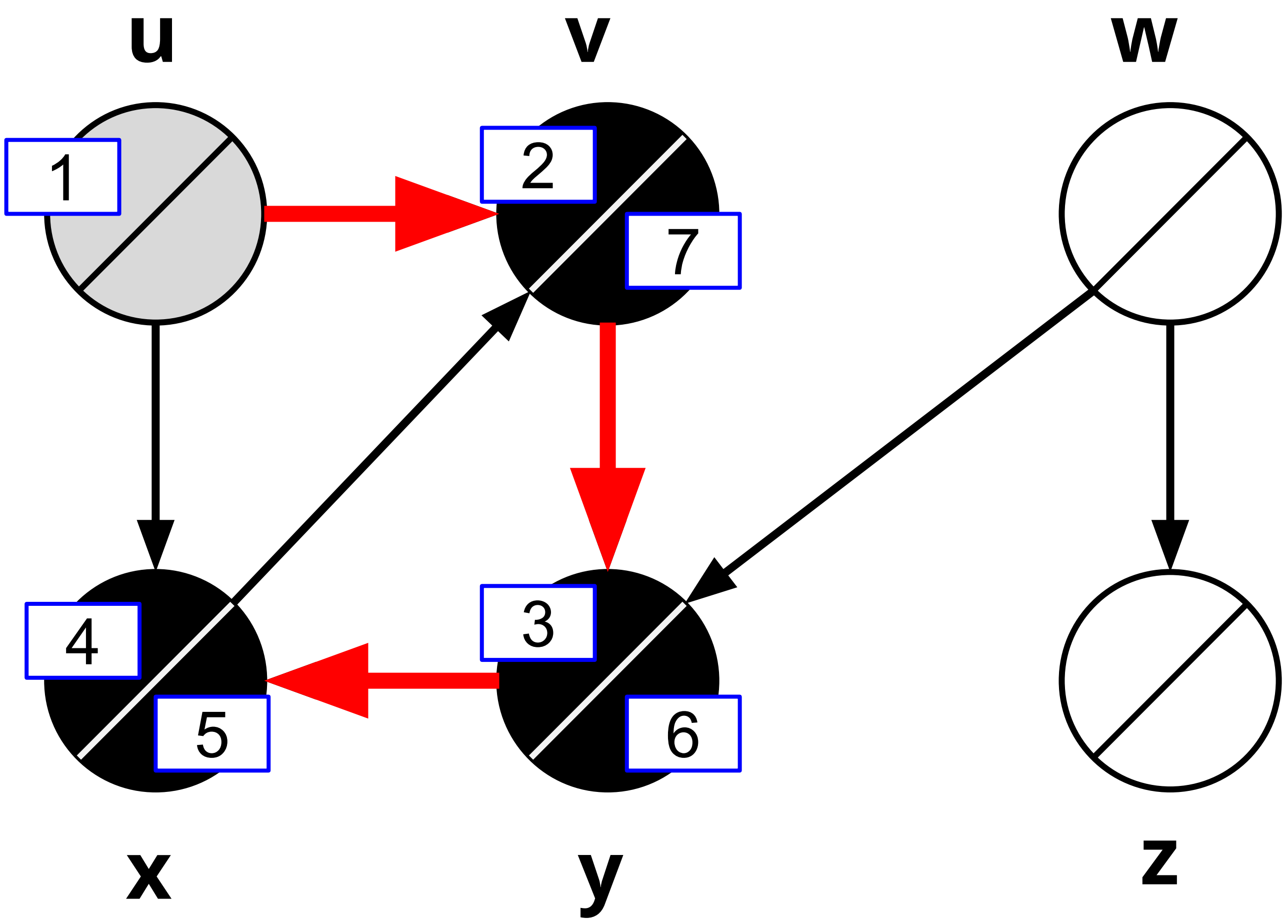
time = 5, vertex x finished



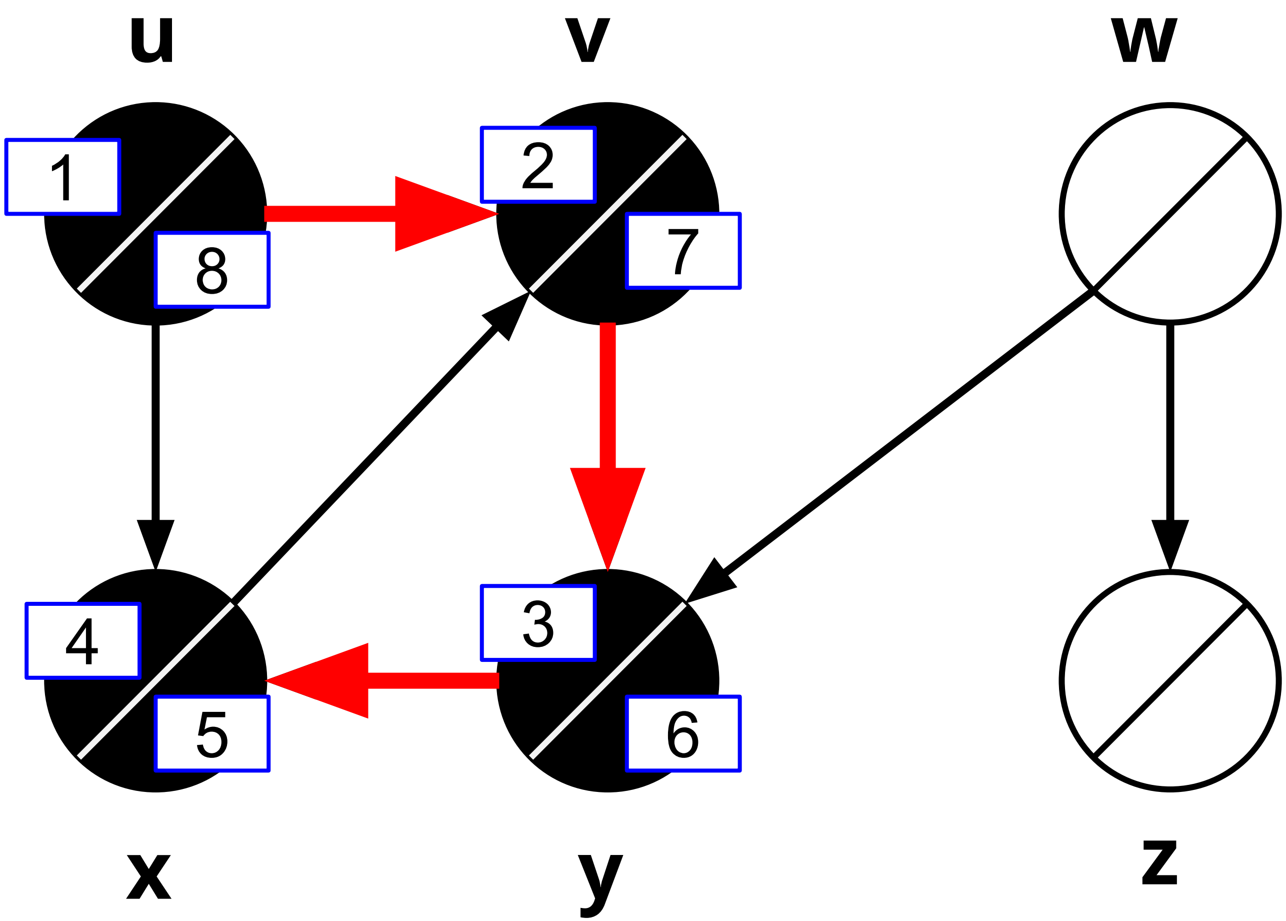
time = 6, recursion back to level 3, finish y



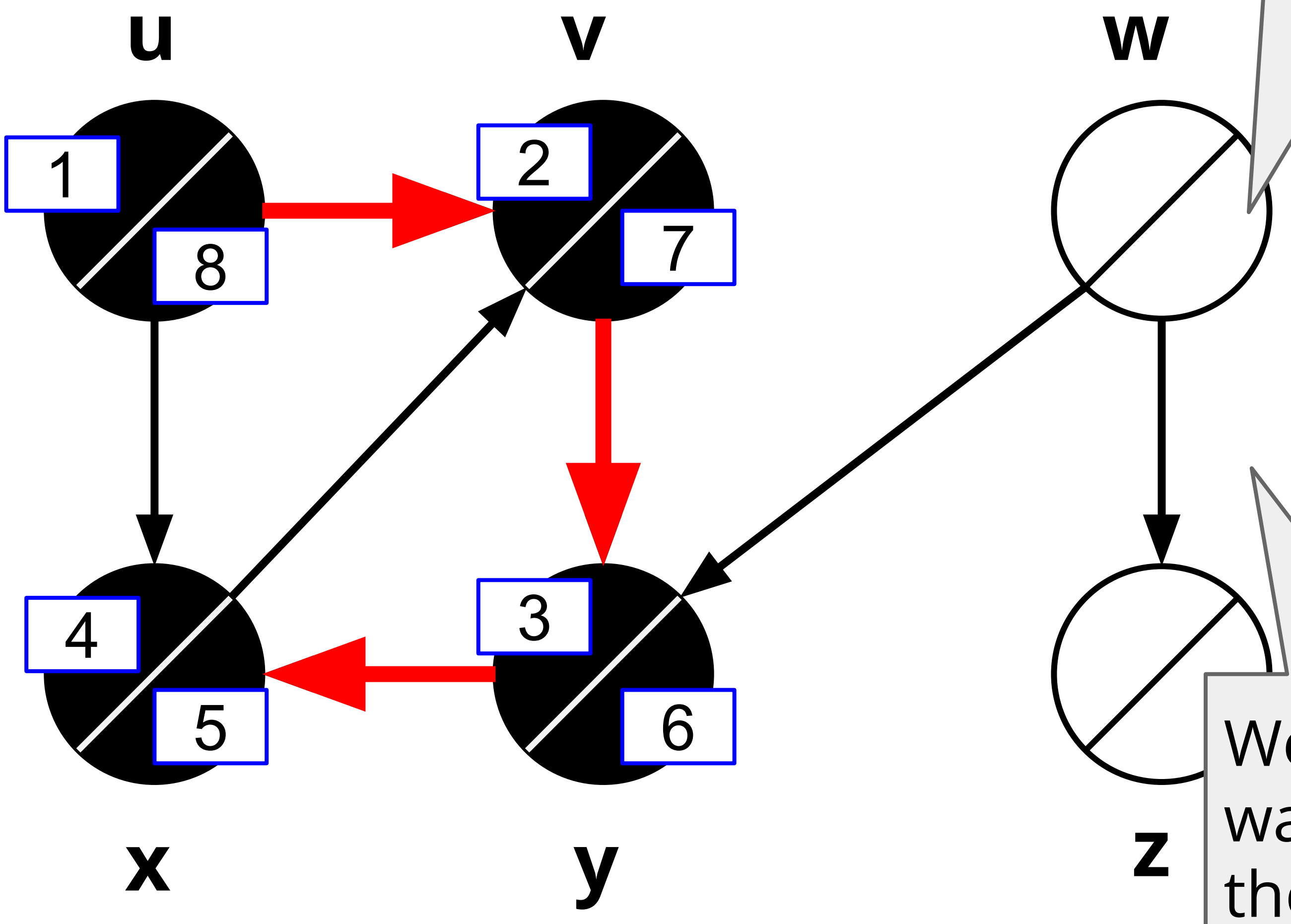
time = 7, recursive back to level 2, finish v



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



DFS_VISIT(G, u) done!



What about these two white vertices?

We actually want to visit them (for some reason)

The pseudo-code for visiting everyone

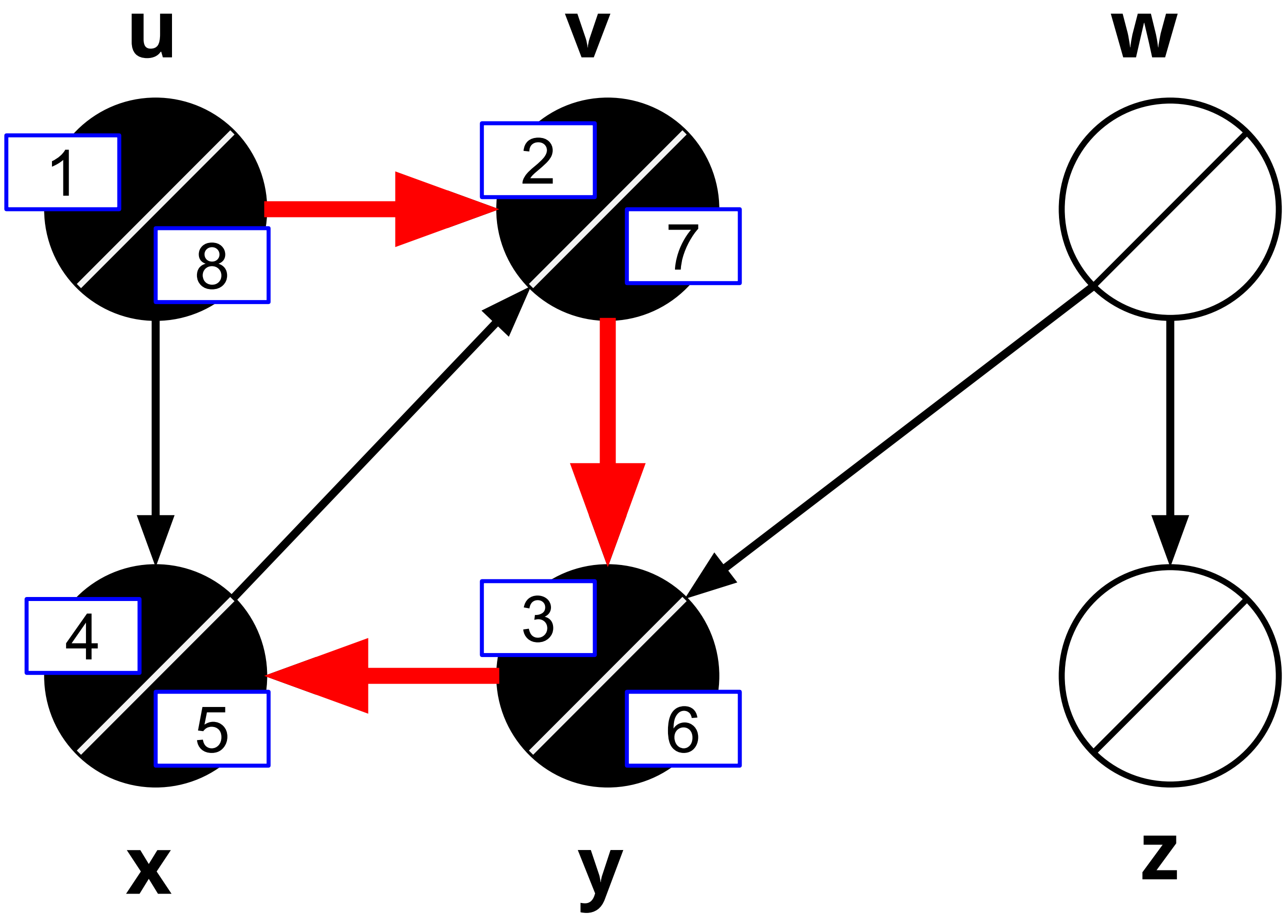
```
DFS(G):
    for each v in G.V:
        colour[v] ← white
        f[v] ← d[v] ← ∞
        pi[v] ← NIL
    time ← 0
    for each v in G.V:
        if colour[v] = white:
            DFS_VISIT(G, v)
```

Initialization

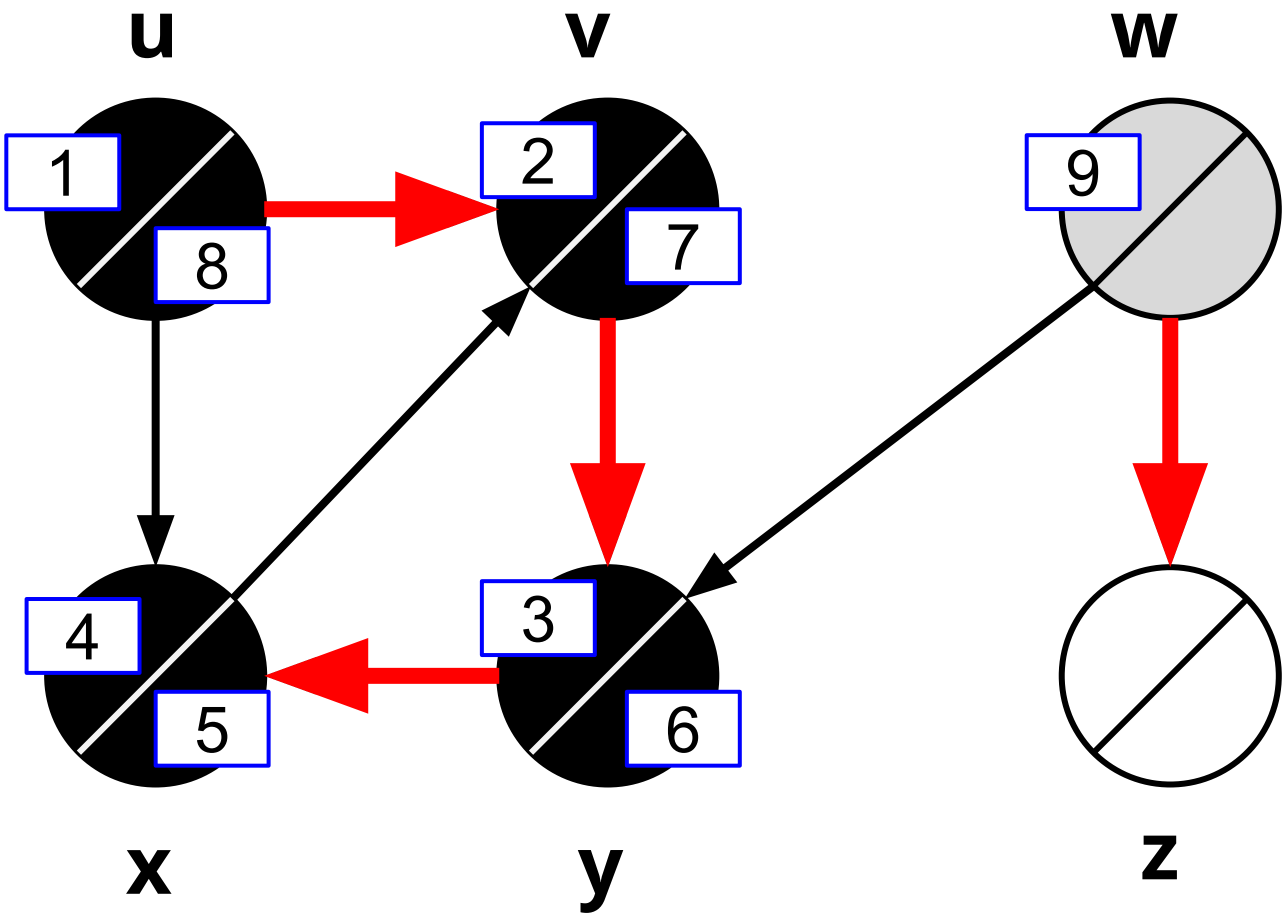
Make sure NO vertex is left with white colour.

```
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
```

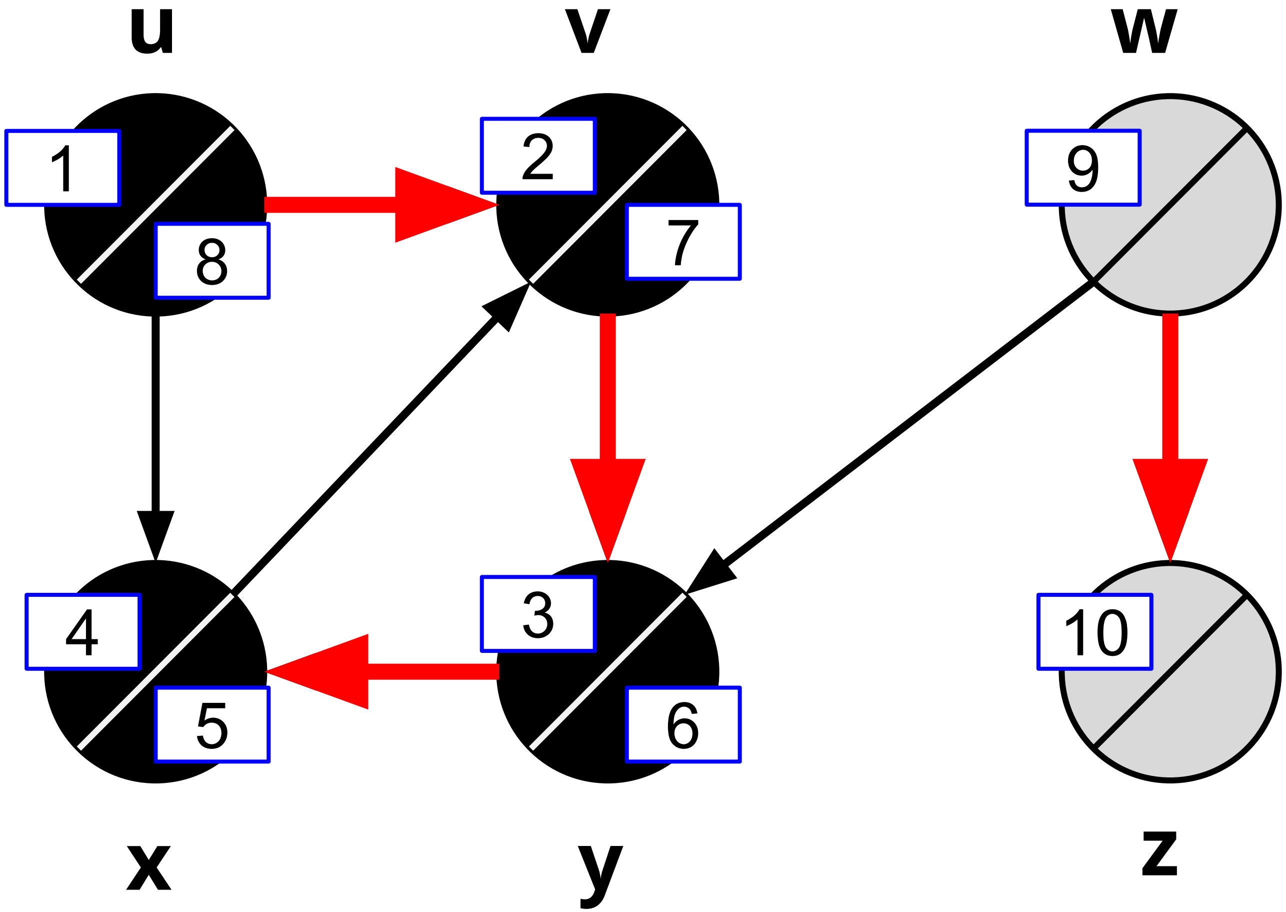
So, let’s finish this DFS



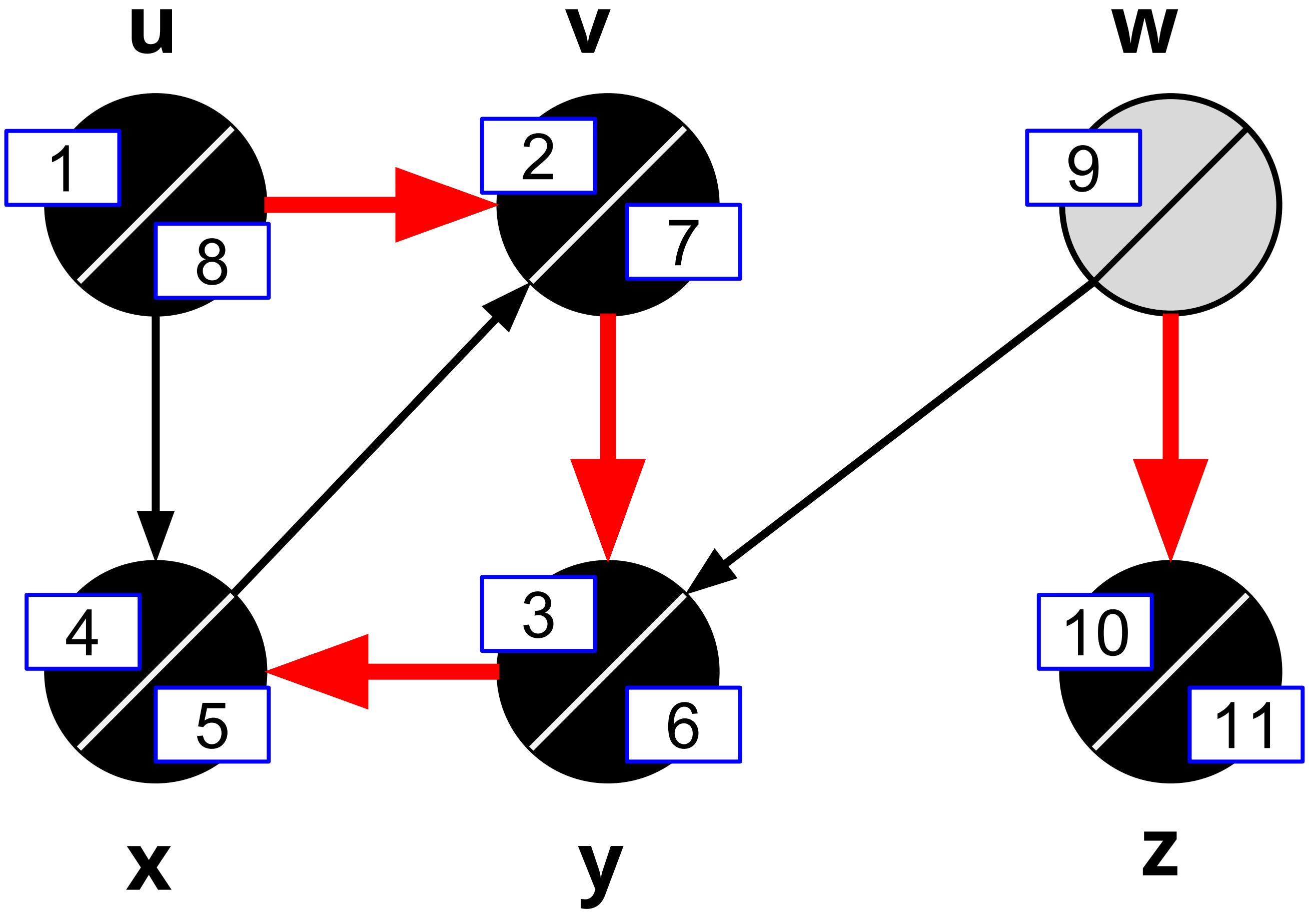
time = 9, DFS_VISIT(G, w)



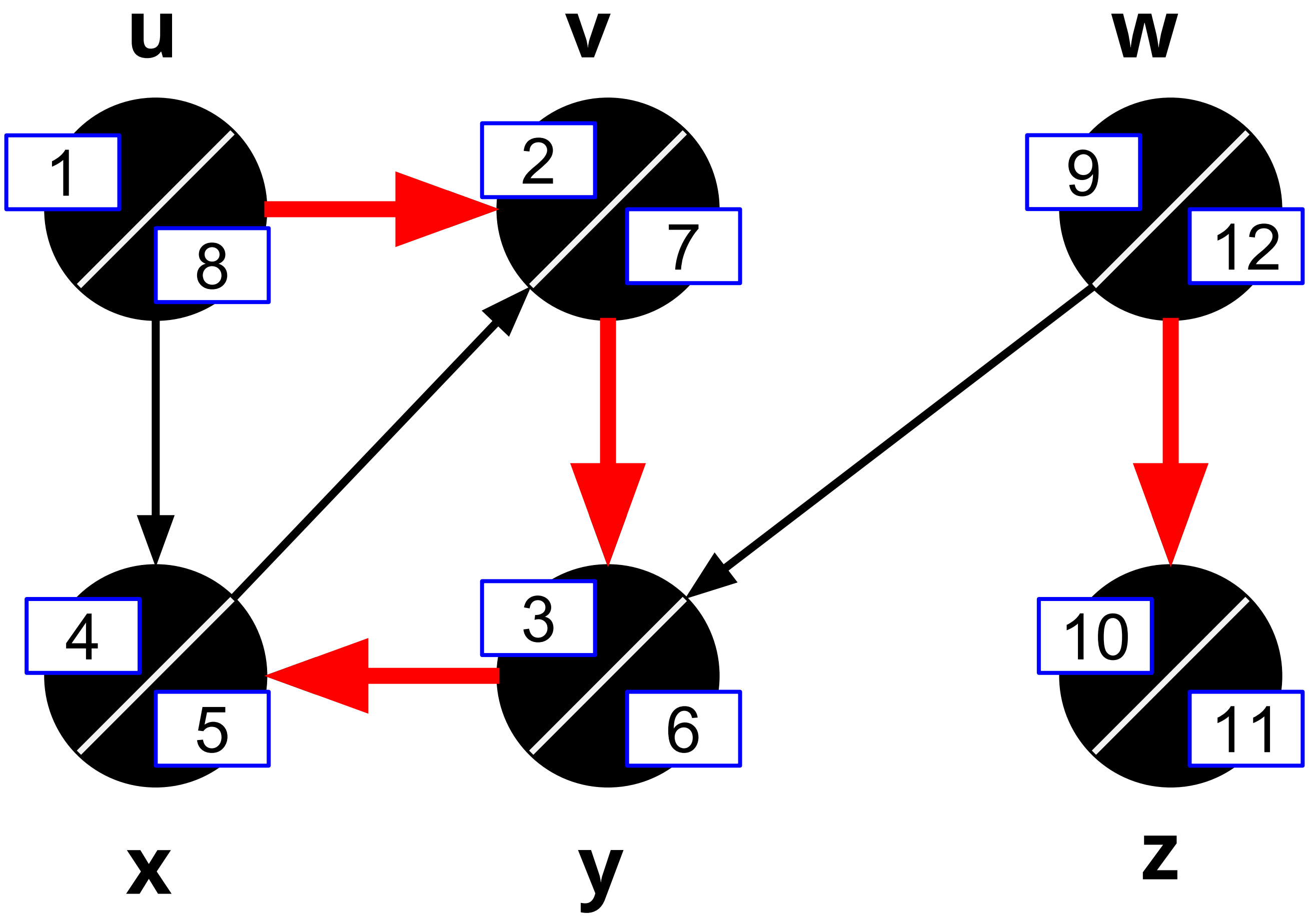
time = 10



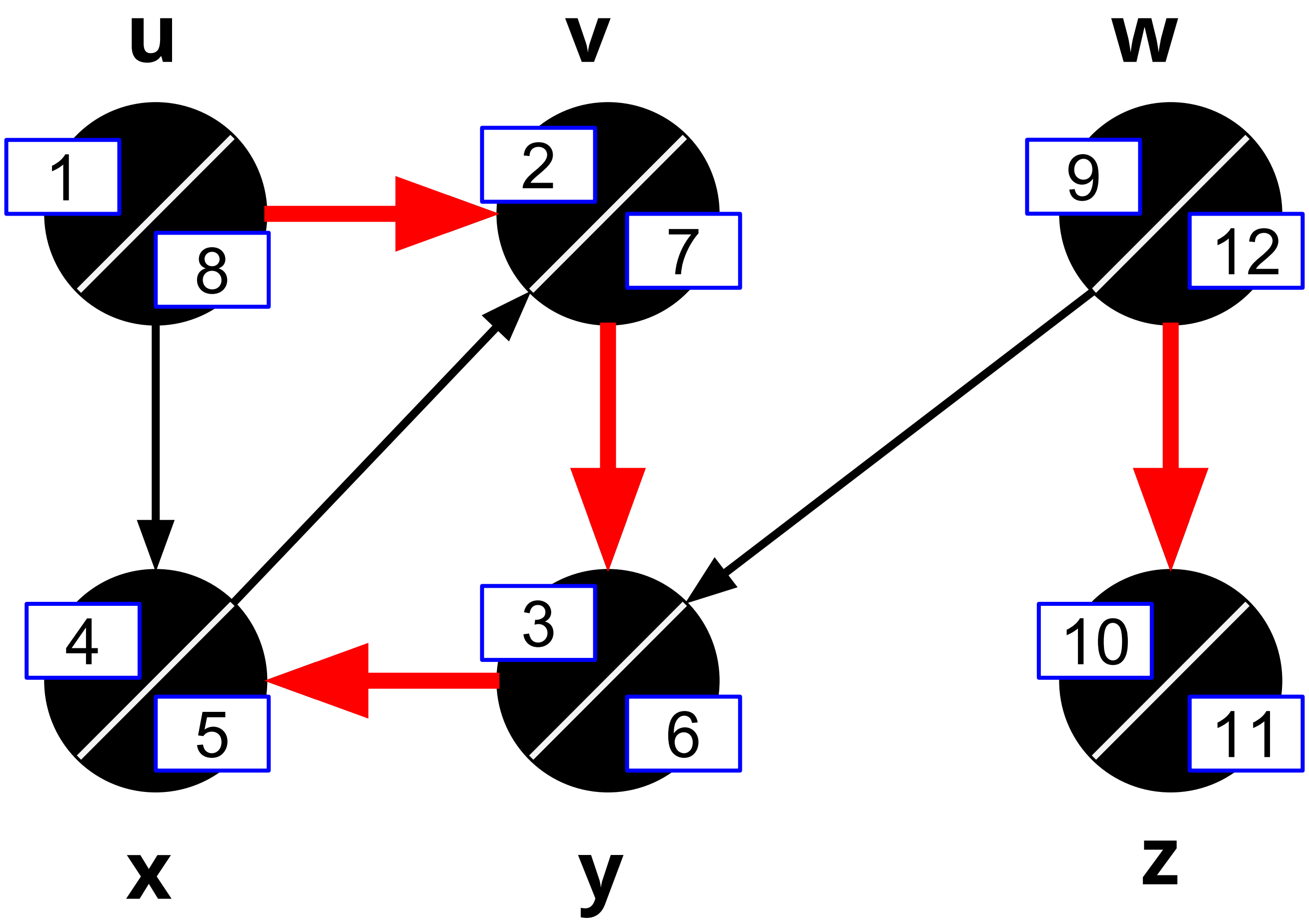
time = 11



time = 12



DFS(G) done!



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|)**

Same as BFS

Total runtime:

O(|V|+|E|)

What do we get from DFS?

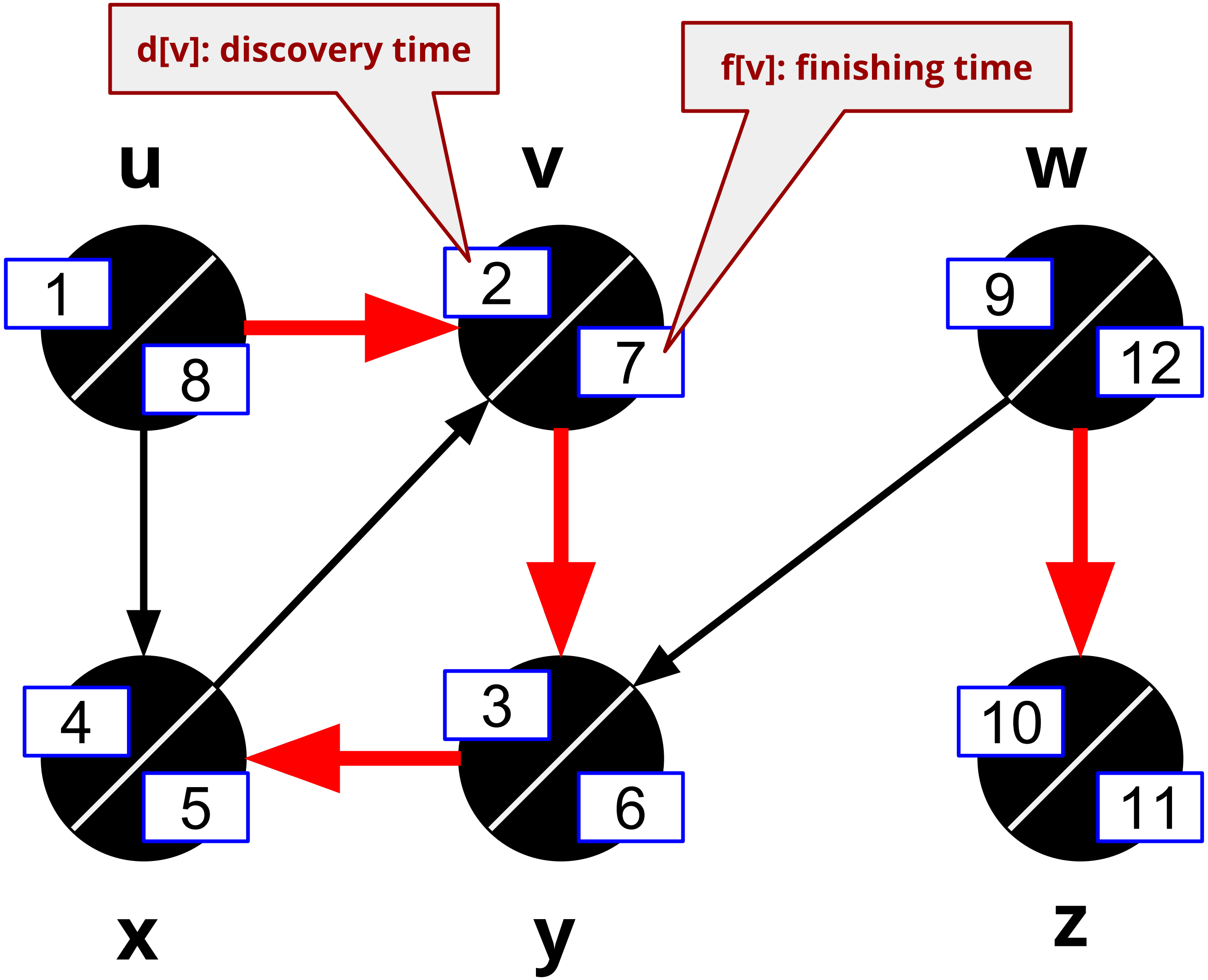
- Detect whether a graph has a cycle.
  - That’s why we wanted to visit all vertices -- if you want to be sure whether a graph has a cycle or not, you’d better check **everywhere**.
  - Why didn’t we do the similar thing for BFS?

- How exactly do we detect a cycle?

CSC263 Week 9

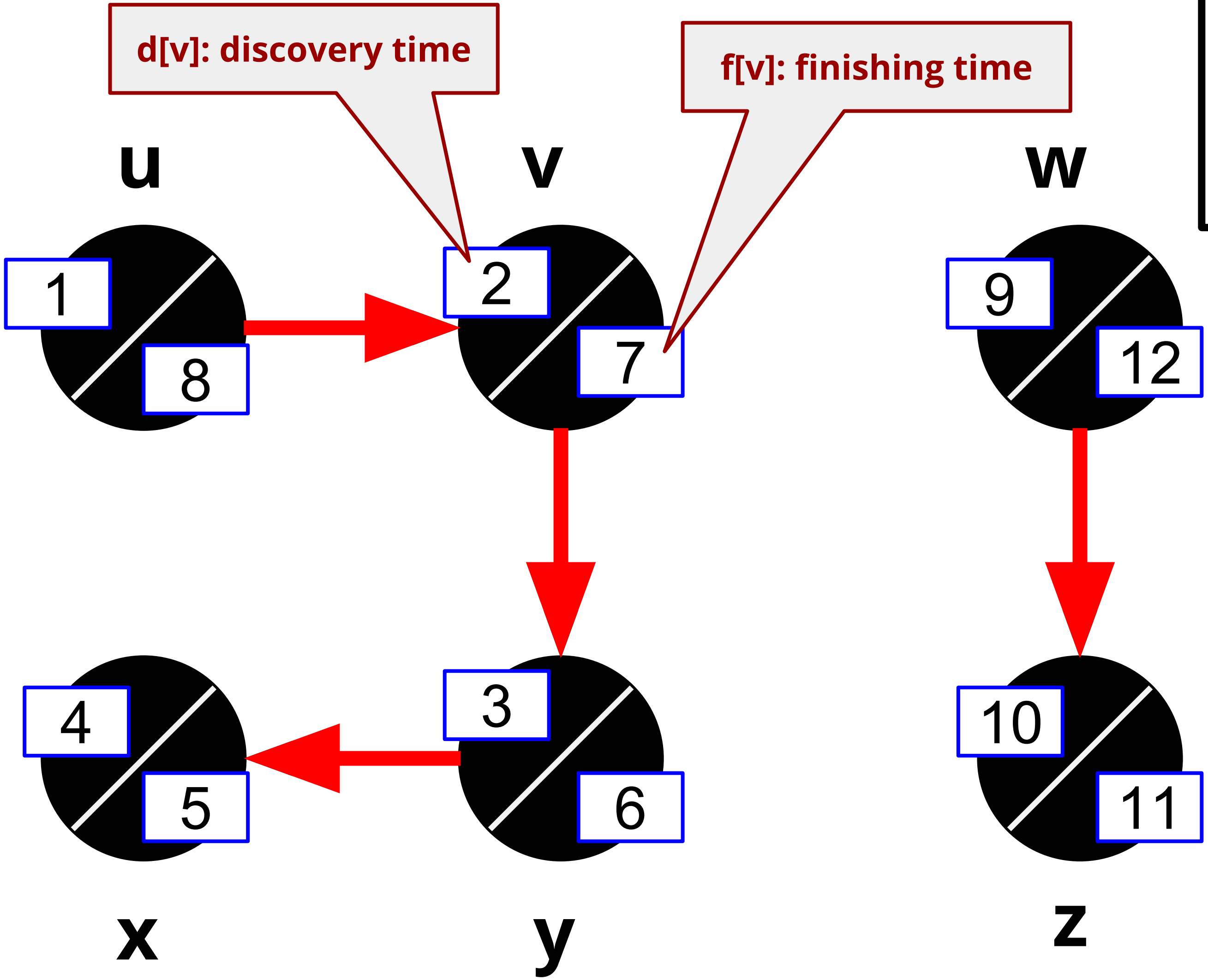
Thursday

Recap: DFS(G) done!



How do we use all the info?

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


d[v]: discovery time

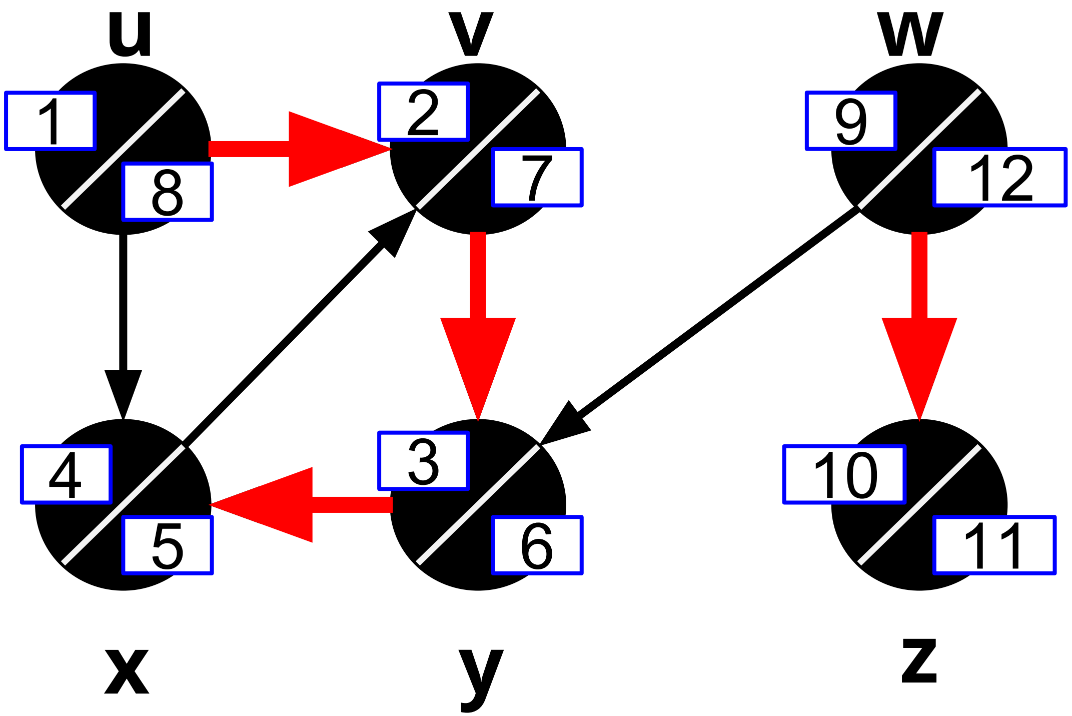
f[v]: finishing time

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 red edges) starting from **y**, see whether you can get to **u**.

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



the “parenthesis structure”

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

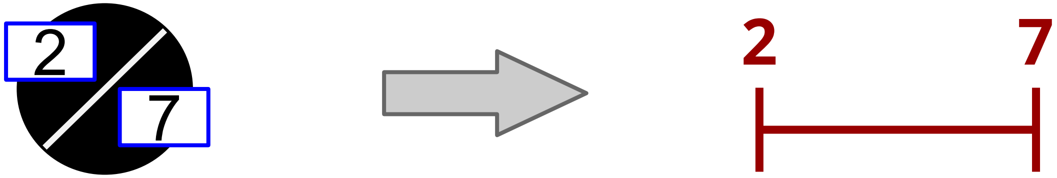
➔ Either one pair **contains** the another pair.

➔ Or one pair is **disjoint** from another

(( ))

This (overlapping) never happens!

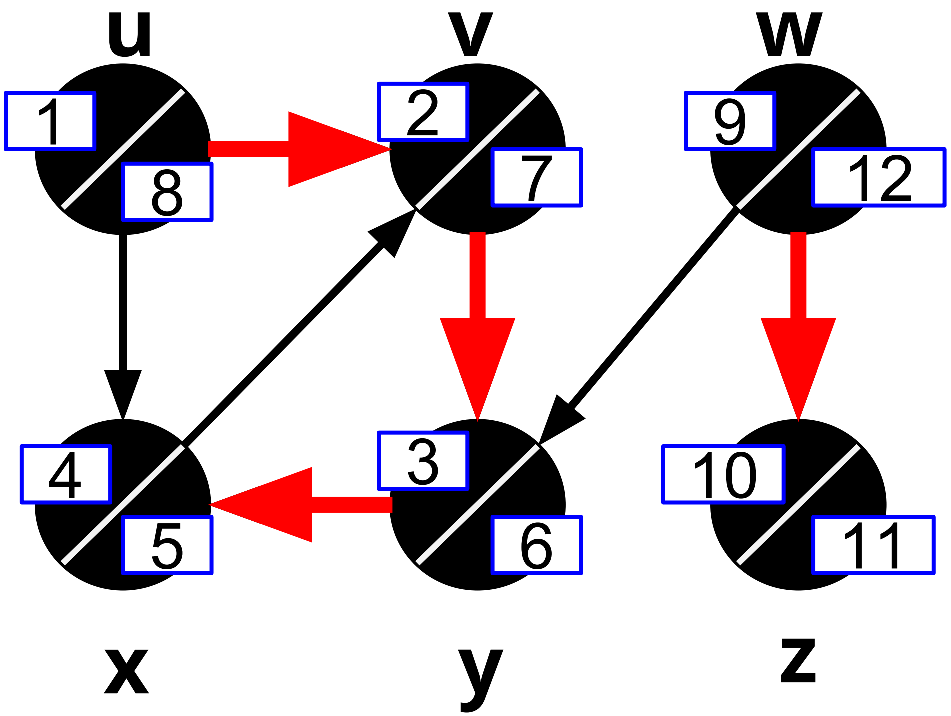
Visualize d[v], f[v] as interval [d[v], f[v]]

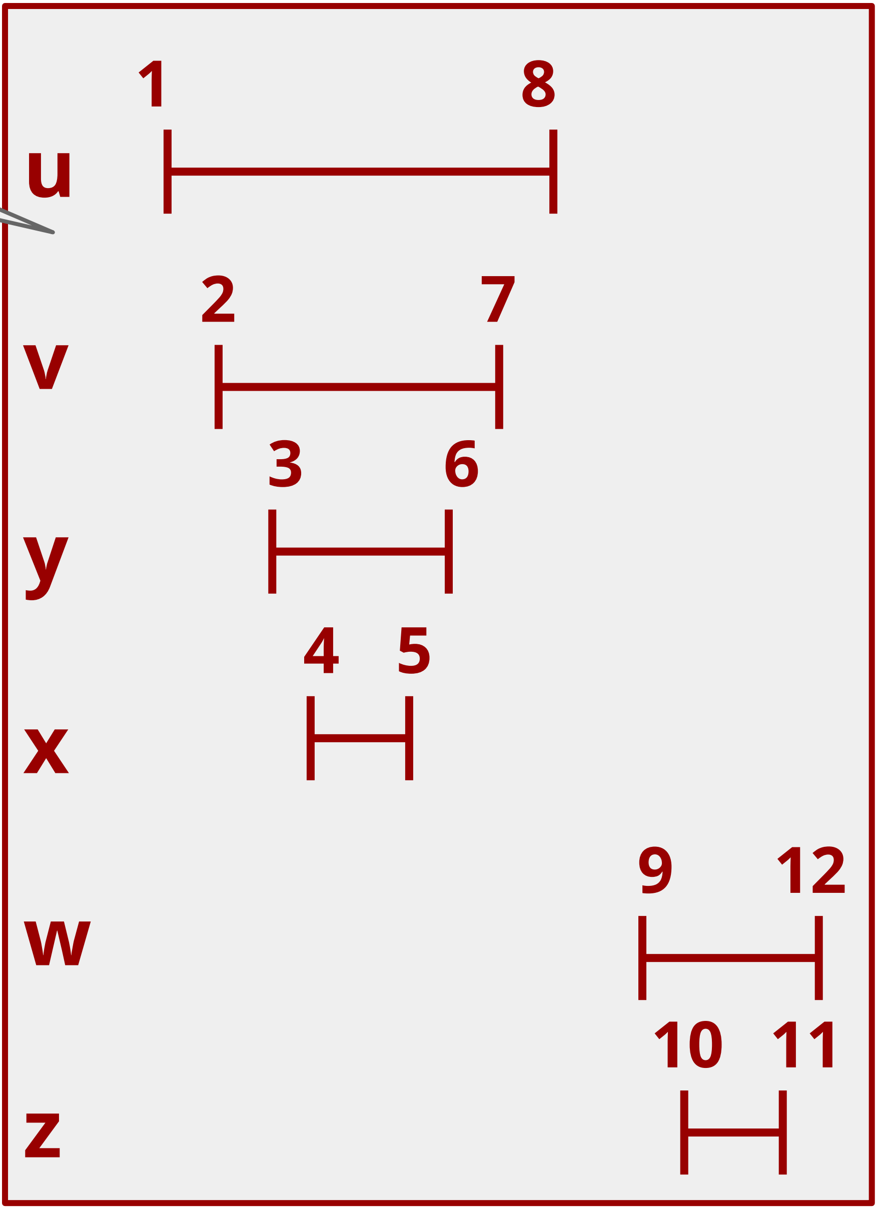


Now, visualize all the intervals!

What do you see in this?

Parenthesis structure!





The **[d[v], f[v]]** intervals that we got from DFS follow the parenthesis structure, i.e.,

- ➔ Either one interval **contains** another
- ➔ Or one is **disjoint** from another

Moreover,

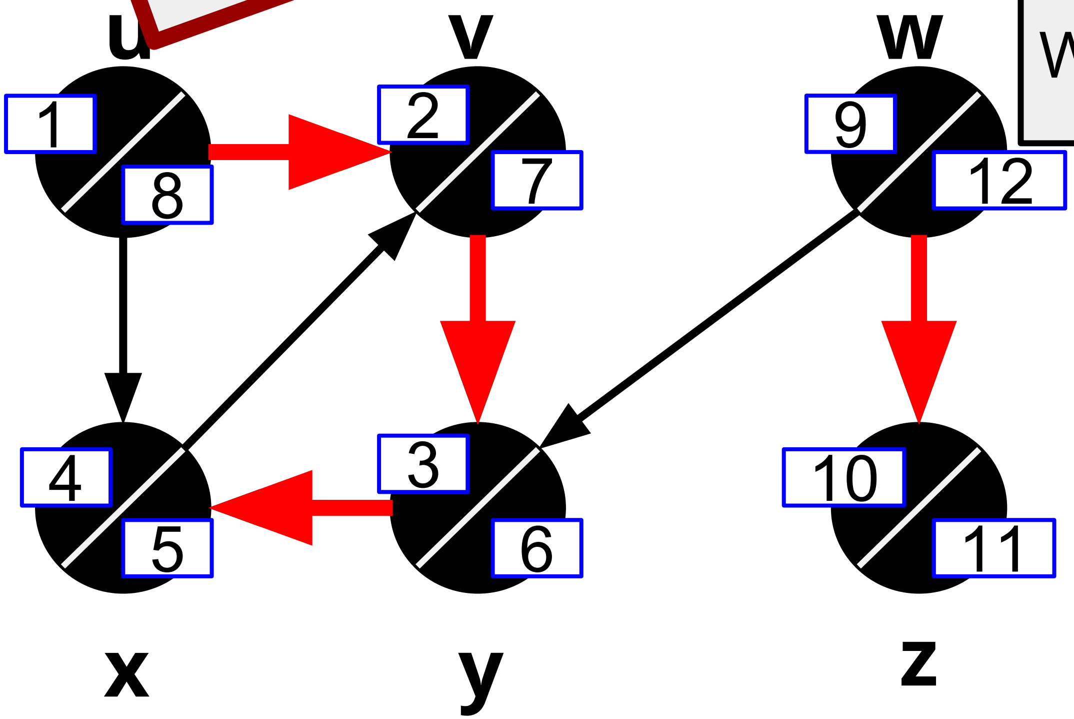
- ➔ **Iff** interval of **u** contains interval of **v**, then **u** is an **ancestor** of **v** in the DFS forest.
- ➔ If interval of **u** is disjoint from interval of **v**, then they are **not** ancestors of each other.

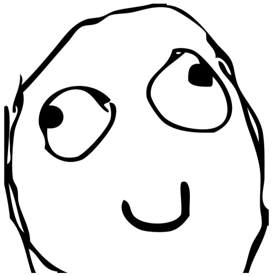
How to decide whether **y** is a **descendant** of **u** in the DFS forest?

Idea #1: trace b nters
(the re y, see
to **u**.
W akes **O(n)** steps.

FORGET ABOUT IT

Idea #2: see if **[d[u], f[u]]** contains **[d[y], f[y]].**
Worst-case: **1 step**!





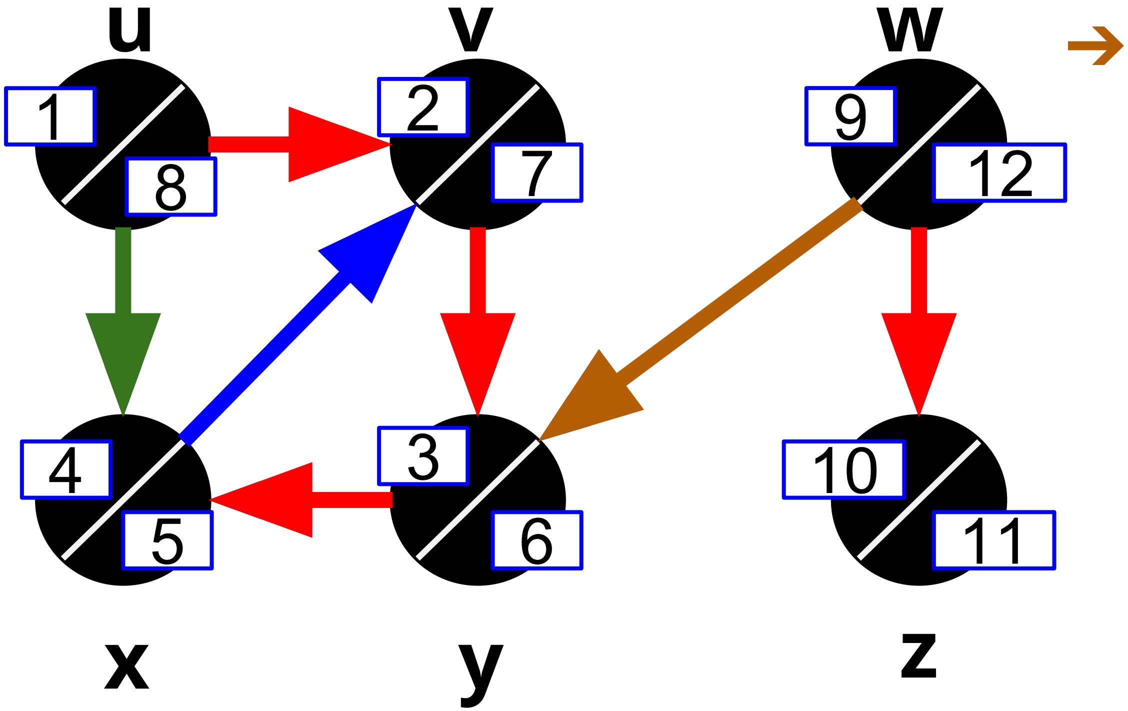
We can efficiently check whether a vertex is an ancestor of another vertex in the DFS forest.

so what...

Classifying Edges

4 types of edges in a graph after a DFS

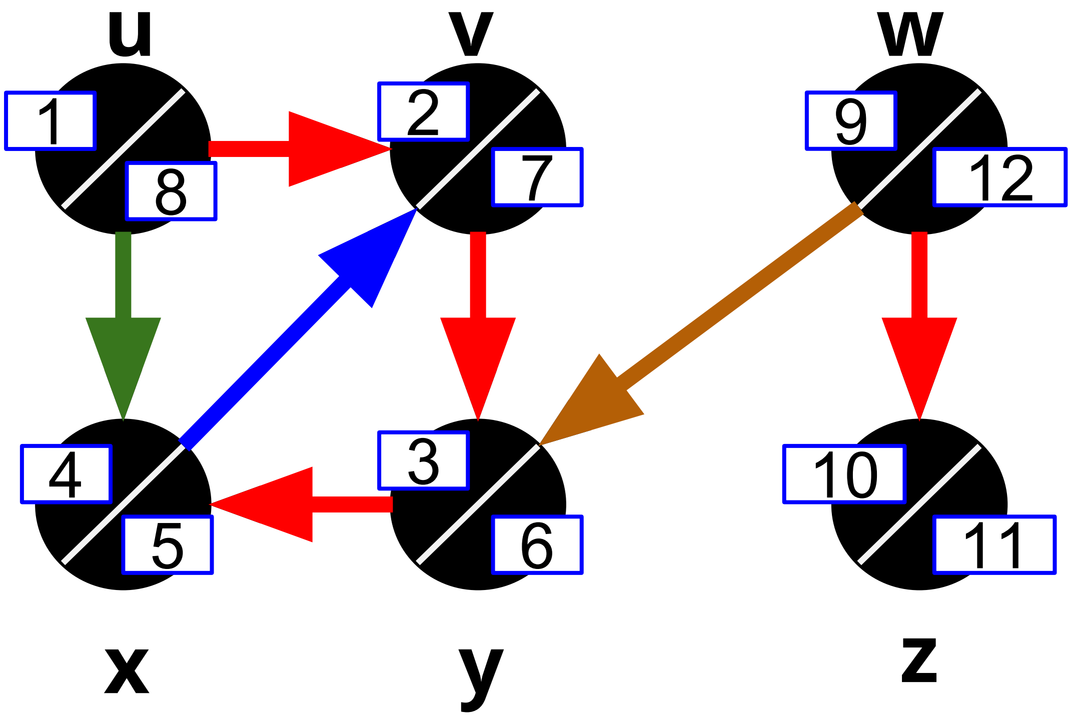
- **Tree edge:** an edge in the DFS-forest
- **Back edge:** a non-tree edge pointing from a vertex to its **ancestor** in the DFS forest.
- **Forward edge:** a non-tree edge pointing from a vertex to its **descendant** in the DFS forest

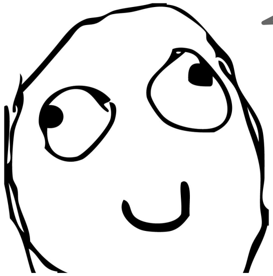


- **Cross edge**: all other edges

Checking edge types

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



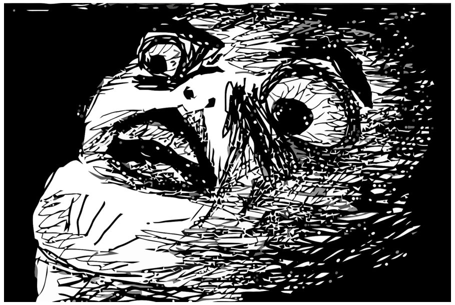


We can efficiently check edge types after a DFS!

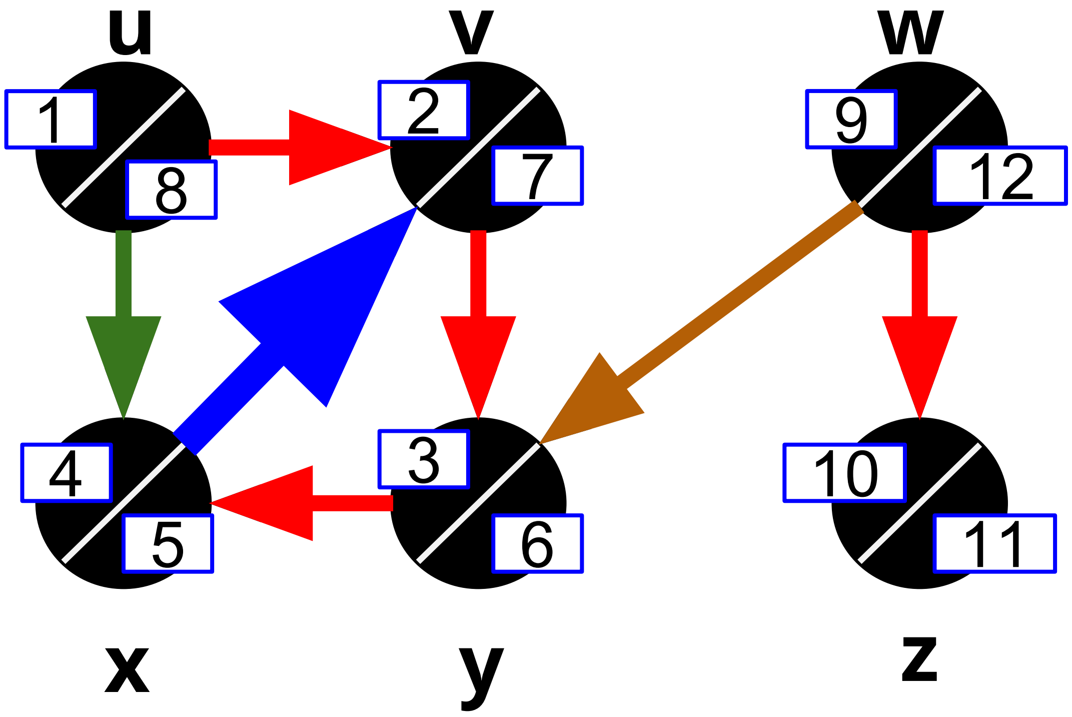
so what...

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



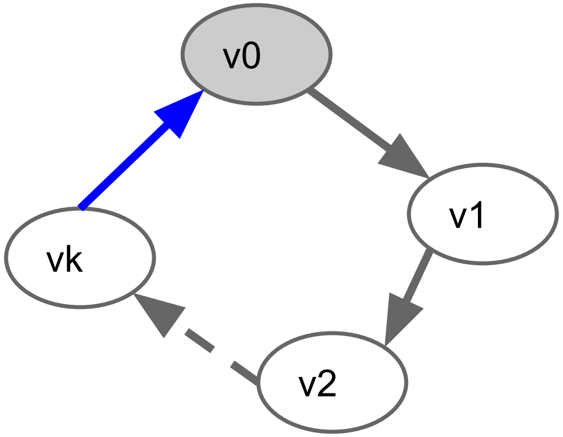
A (directed) graph contains a cycle if and only if DFS yields a back edge

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 their is a path from v to u, i.e., v → … → u,
plus the back edge u → v,
BOOM! Cycle.

Proof of "only if":

Let the cycle be…,

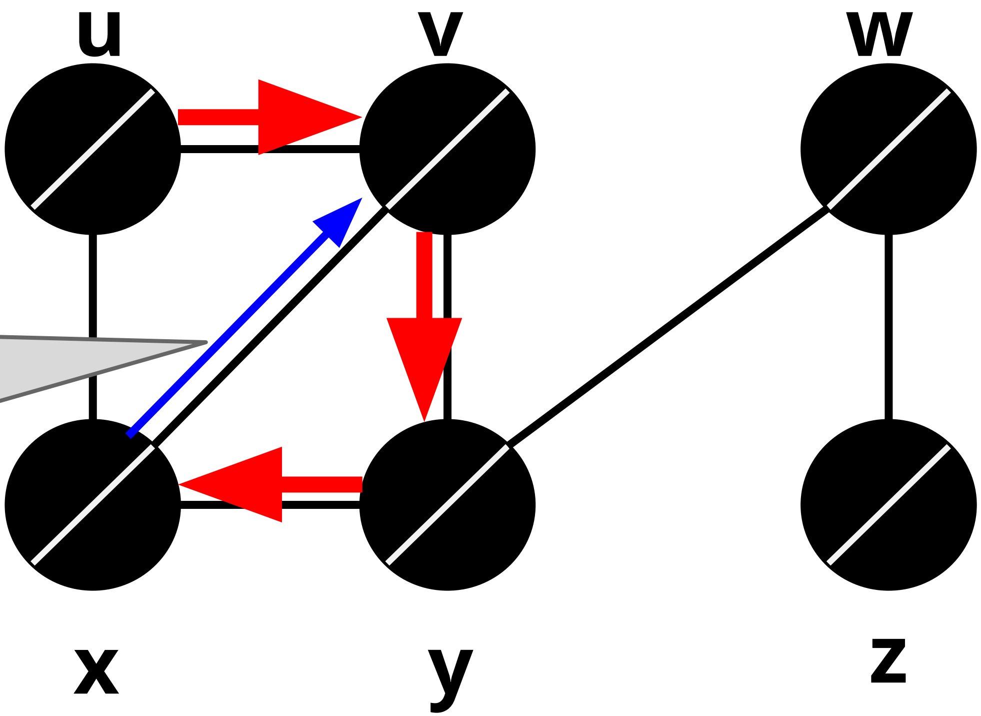


Let v0 be the first one that turns gray, when all others in the cycle are white, then vk must be a descendant of v0.
(Read "White Path Theorem" in Text)

How about undirected graph?

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

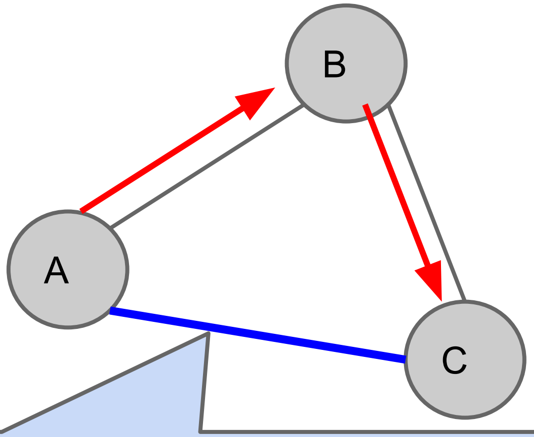
➔ No, because although the edges are undirected, **neighbour checking** still has a “direction”.



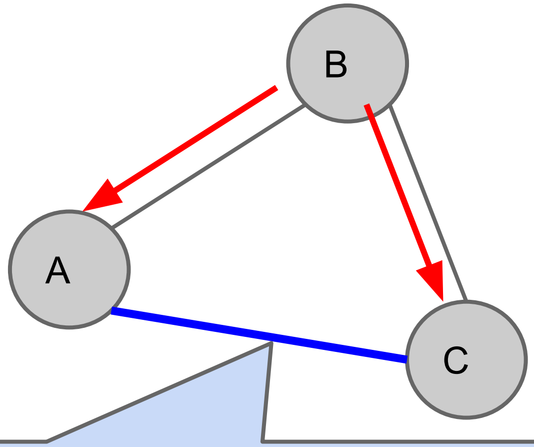
Checking in this direction, so it’s a back edge

More about undirected graph

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



If this were a forward edge, it would violate the DFS algorithm (not checking at C but tracing back and check at A)

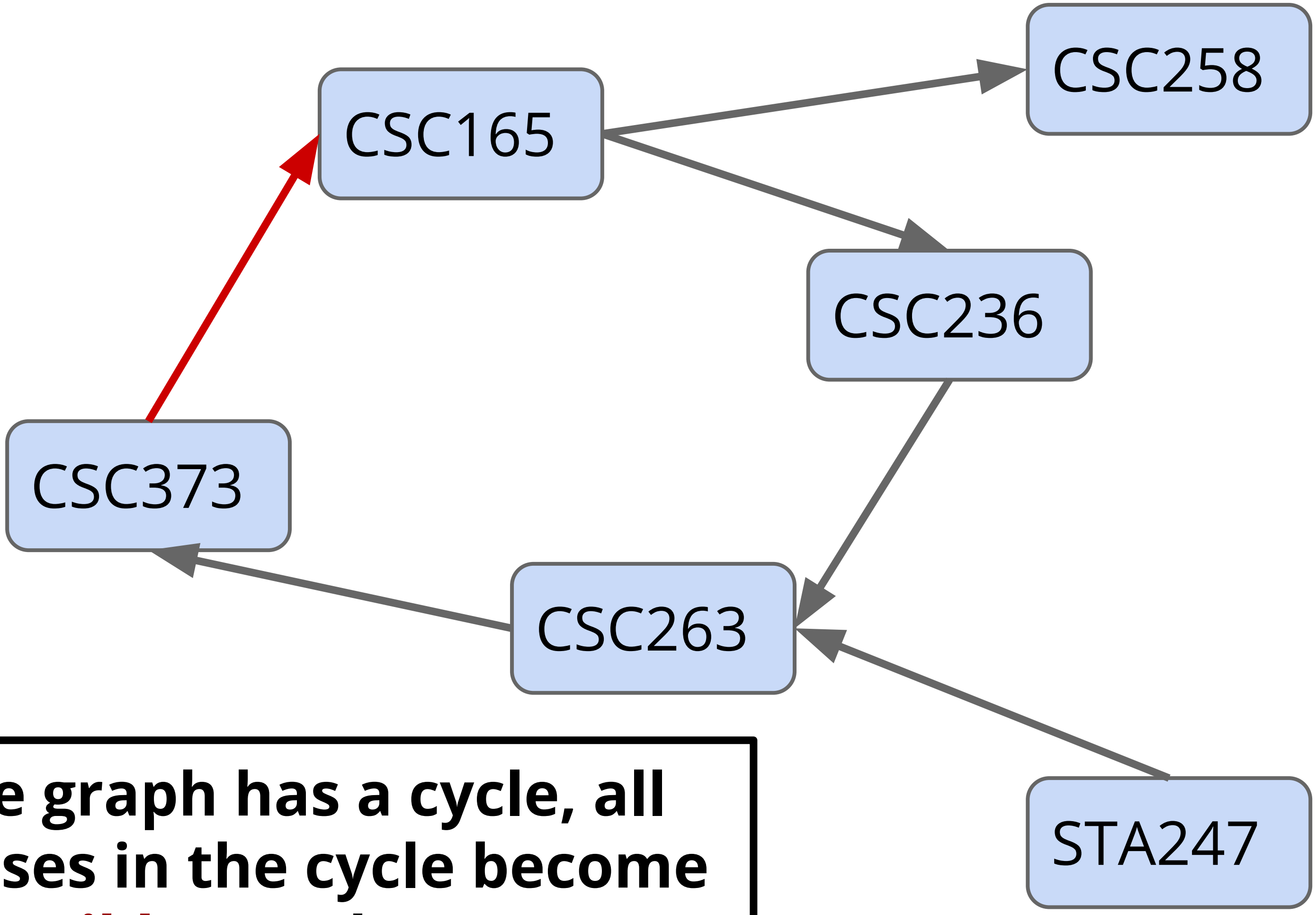


If this were a cross edge, it violets DFS again (should have checked (A, C) when reached A, but instead wait until C is visited.)

Why do we care about cycles in a graph?

Because cycles have meaningful implication in real applications.

Example:
a course prerequisite graph



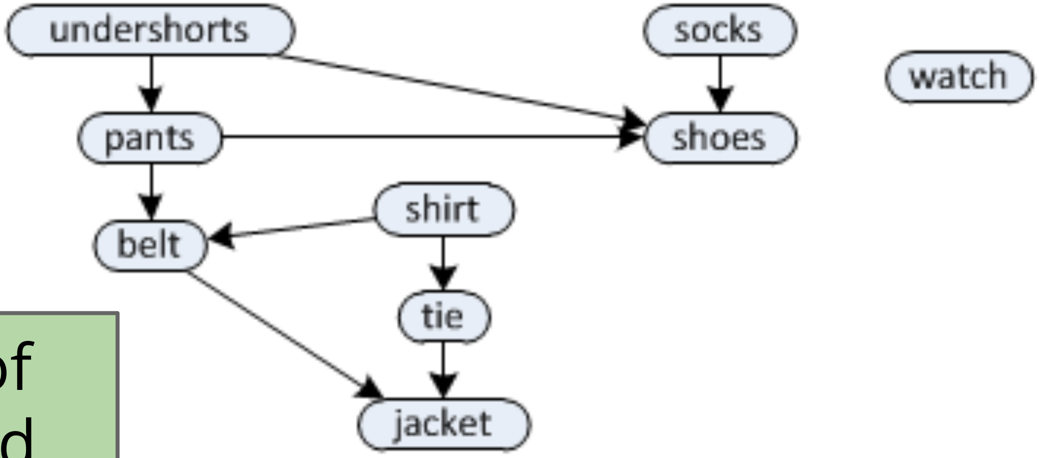
If the graph has a cycle, all courses in the cycle become impossible to take!

Applications of DFS

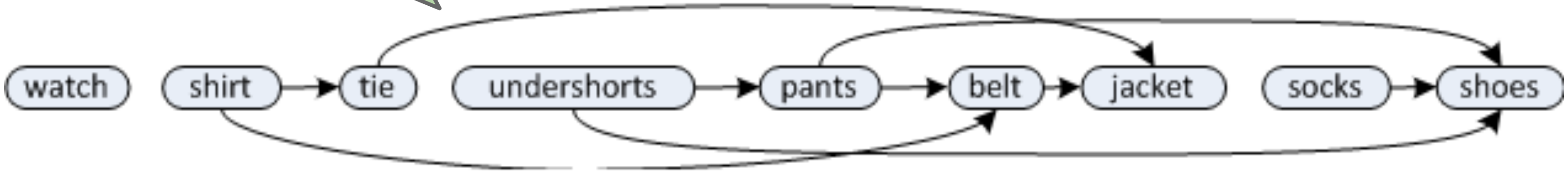
- ➔ Detect cycles in a graph
- ➔ Topological sort
- ➔ Strongly connected components

Topological Sort

- ➔ Place the vertices in such an order that all edges are pointing to the right side.



A valid order of getting dressed.



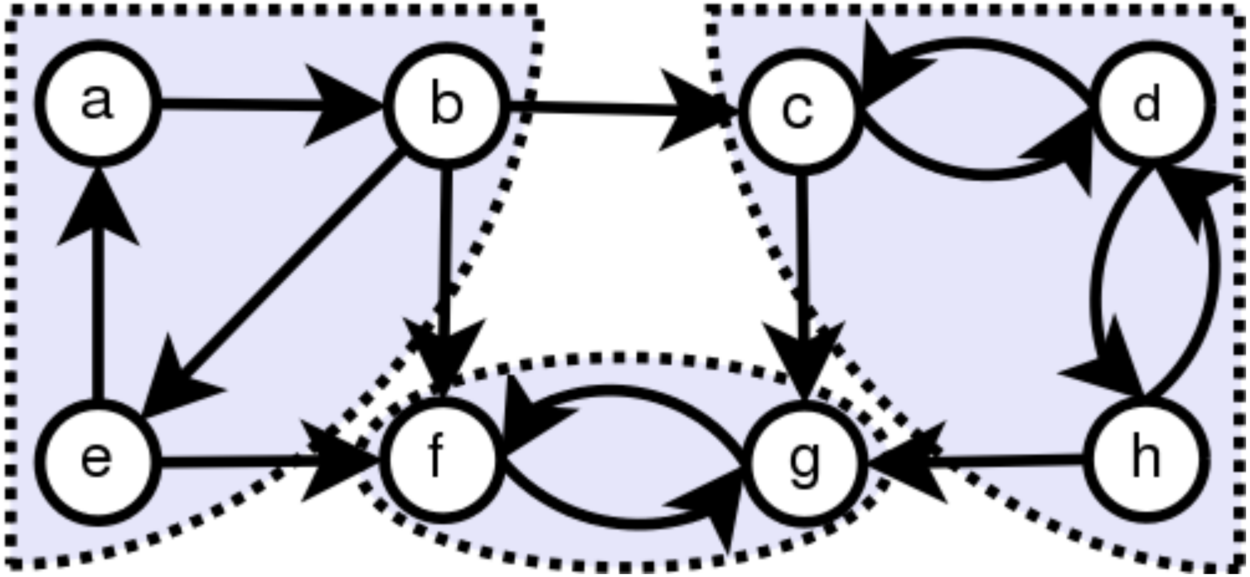
How to do topological sorting

1. Do a **DFS**

2. Order vertices according to their **finishing times f[v]**

Strongly connected components

- Subgraphs with strong connectivity (any pair of vertice can reach each other)

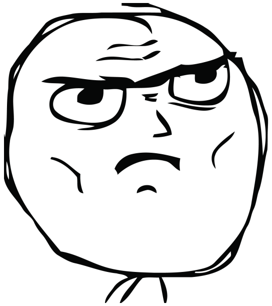


Summary of DFS

- ➔ 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
- ➔ Give us cycle detection (back edge)
- ➔ For real problems, choose BFS and DFS wisely.

Next week

- → Minimum Spanning Tree



http://goo.gl/forms/S9yie3597B