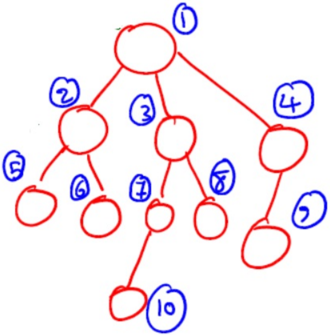
CSC263H
Data Structures and Analysis

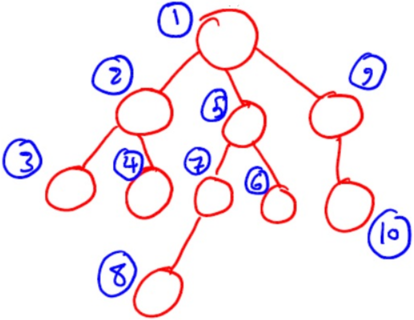
Dr. Bahar Aameri & Dr. Marsha Chechik

Winter 2024 – Week 9

BFS vs DFS: A Familiar Example

Consider performing the BFS and DFS algorithms on the **root** of a **tree**.





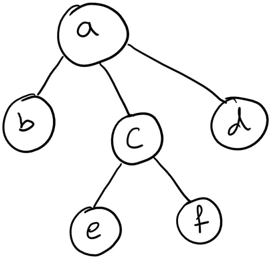
BFS in a tree (starting from root) is a level-by-level traversal.

DFS visits the child vertices before visiting the sibling vertices.

```
NotYetBFS(T, root):
    Q = ∅
    Enqueue(Q, root)
    While Q not empty:
        u = Dequeue(Q)
        print u
        for each child c of u:
            Enqueue(Q, c)
```

```
NotYetDFSIte(T, root):
    S = ∅
    Push(S, root)
    While S not empty:
        u = Pop(S)
        print u
        for each child c of u:
            Push(S, c)
```

```
NotYetDFSIte(T, root):
    S = ∅
    Push(S, root)
    While S not empty:
        u = Pop(S)
        print u
        for each child c of u:
            Push(S, c)
```



Iteration	Output	Stack
0		a
1	a	b, c, d
2	b	c, d
3	c	f, e, d
4	f	e, d
5	e	d
6	d	

Notice that NotYetDFSIte is a stack simulation of a **recursive** algorithm.

```
NotYetDFSIte(T, root):
    S = ∅
    Push(S, root)
    While S not empty:
        u = Pop(S)
        print u
        for each child c of u:
            Push(S, c)
```

```
NotYetDFSRec(T, root):
    print root
    for each child c of root:
        NotYetDFSRec(T, c)
```

Exercise: Trace NotYetDFSRec on the tree in the previous slide.

DFS Implementation

How avoid visiting a vertex **twice**?
Remember the visited vertices by labelling them using colours (Similar to BFS).

- White: **Unvisited** (undiscovered) vertices.
- Gray: **Encountered** (discovered) vertices.
- Black: **Explored** vertices.
  Have been **visited** and all of their **neighbours** are **explored**.

- Initially all vertices are **white**.
- Change a vertex's color to **gray** the first time visiting it (i.e., calling DFSVisit for the vertex).
- Change a vertex's color to **black** when all its neighbours have been explored.
- Avoid visiting (i.e., calling DFSVisit for) **gray** or **black** vertices.
- In the end, all vertices are **black**.

Other useful values to remember during the traversal (NOT exactly the same as BFS):

- The vertex from which v is encountered, stored in $v.p$

- Keep track of two *timestamps* for each vertex v:
  There is a **timer** incremented whenever a vertex's **colour is changed**:
  - **Discovery time**: the **time** when v is first **encountered**, stored in $v.d$
  - **Finishing time**: the **time** when all the neighbours of v have been **completely visited**, stored in $v.f$

DFS Implementation

```
DFS(G):
    for each t ∈ G.V:                    # Initializing
        t.colour = White
        t.p = nil
    time = 0
    for each s ∈ G.V:
        if s.colour == White             # Make sure NO vertex is left unvisited.
            DFSVisit(G, s)
```

DFS(G):

$\Theta(n)$

$|V| = n$

$|E| = m$

for each $t \in G.V$: **# Initializing**
    $t.colour$ = White
    $t.p$ = nil
$time = 0$
for each $s \in G.V$:
    if $s.colour$ == White **# Make sure NO vertex is left unvisited.**
        DFSVisit(G, s)

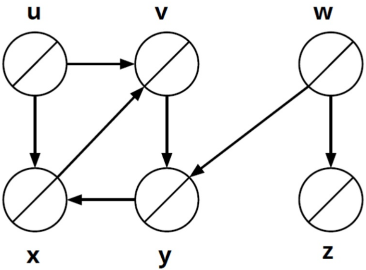
DFSVisit(G, s):

$time = time + 1$ **# time is a global variable**
$s.d = time$
$s.colour$ = Gray
for each $t \in G.adj[s]$:
    if $t.colour$ == White **# only visit unvisited vertices**
        $t.p = s$ **# t is introduced as s's neighbour**
        DFSVisit(G, t)
$s.colour$ = Black **# s is explored as all its neighbours have been encountered**
$time = time + 1$
$s.f = time$ **# Keep finishing time after exploring all neighbours**

The blue lines are the same as NotYetDFSRec.

DFSVisit(G, u)

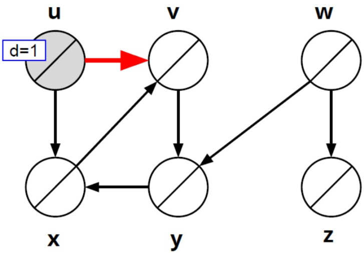
$time = 0$



DFSVisit(G, u)

$time = 1$

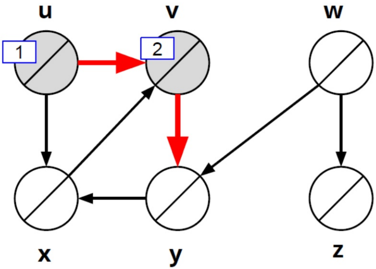
Encounter the **source** vertex of DFSVisit



DFSVisit(G, u)

$time = 2$

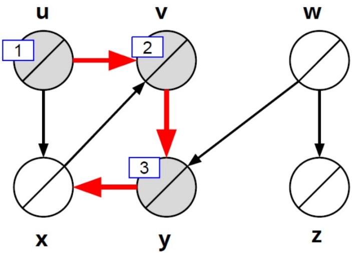
Level 2 of recursive call



DFSVisit(G, u)

$time = 3$

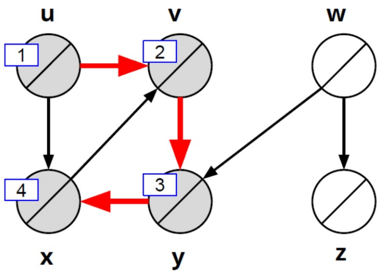
Level 3 of recursive call



DFSVisit(G, u)

$time = 4$

Level 4 of recursive call

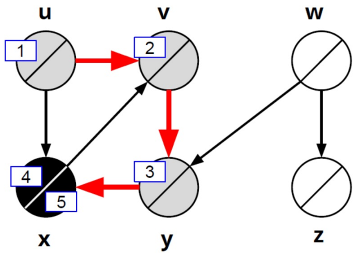


DFSVisit(G, u)

$time = 5$

Level 4 of recursive call

Vertex x finished

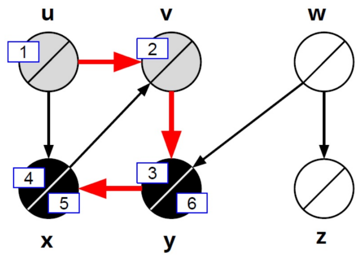


DFSVisit(G, u)

$time = 6$

Recursion back to **Level 3**

Vertex $\boldsymbol{y}$ finished

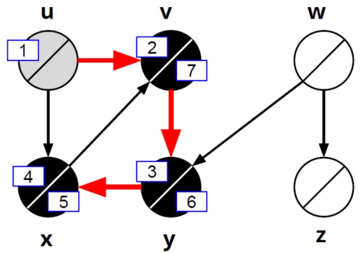


DFSVisit(G, u)

$time = 7$

Recursion back to **Level 2**

Vertex $\boldsymbol{v}$ finished

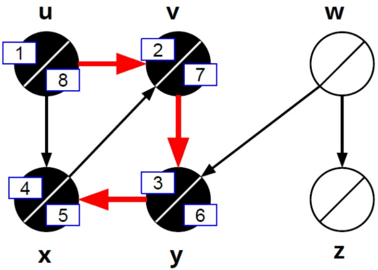


DFSVisit(G, u)

$time = 8$

Recursion back to **Level 1**

Vertex $\boldsymbol{u}$ finished



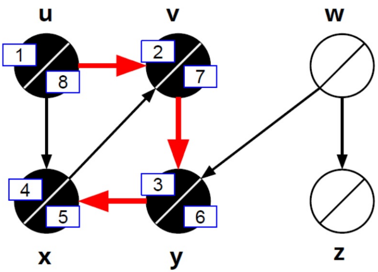
DFSVisit(G, s):

$time = time + 1$ **# time is a global variable**
    $s.d = time$
    $s.colour =$ Gray
    for each $t \in G.adj[s]$:
        if $t.colour ==$ White **# only visit unvisited vertices**
            $t.p = s$ **# t is introduced as s's neighbour**
            DFSVisit(G, t)
    $s.colour =$ Black **# s is explored as all its neighbours have been encountered**
    $time = time + 1$
    $s.f = time$ **# Keep finishing time after exploring all neighbours**

DFS(G):

for each $t \in G.V$: **# Initializing**
        $t.colour =$ White
        $t.p =$ nil
    $time = 0$
    for each $s \in G.V$:
        if $s.colour ==$ White **# Make sure NO vertex is left unvisited.**
            DFSVisit(G, s)

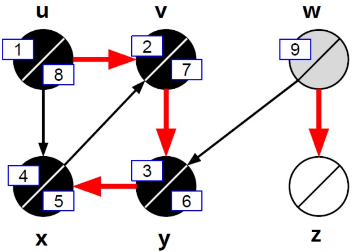
$time = 8$



DFSVisit(G, w)

$time = 9$

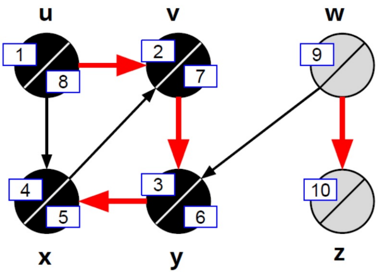
Encounter the **source** vertex of DFSVisit



DFSVisit(G, w)

$time = 10$

Level 2 of recursive call

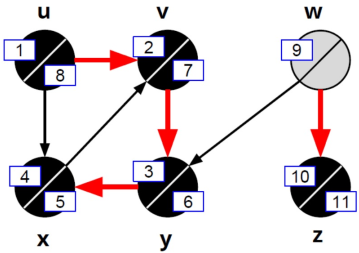


DFSVisit(G, w)

$time = 11$

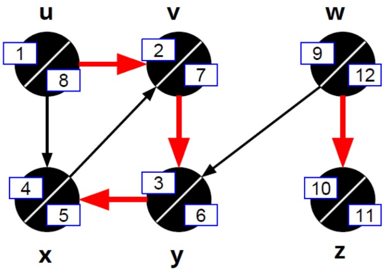
Level 2 of recursive call

Vertex z finished

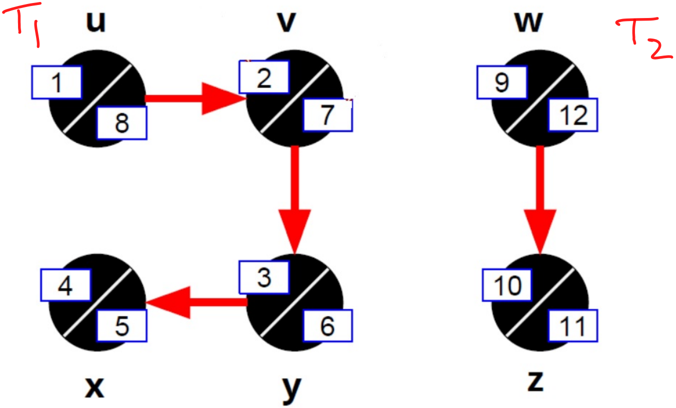


DFSVisit(G, w)

$time = 12$
Recursion back to **Level 1**
Vertex $\boldsymbol{w}$ finished



DFS Forest



DFS Running Time

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

1. Visit each vertex once: $\Theta(n)$
   Assign values to $v.colour$, $v.d$, $v.p$, etc.

2. At each vertex, check all its neighbours (i.e., all its incident edges). $\Theta(m)$
   Each edge is checked at most twice (by the two end vertices)

- Total for 1: $\Theta(n)$
- Total for 2: $\Theta(m)$

$n = |V|$

$m = |E|$

Total running time: $\Theta(n+m)$

Exercise: What is the DFS Worst-case running time when using an adjacency **matrix**? $\Theta(n^2)$

DFS Properties

- DFS can be performed on both **directed and undirected** graphs.

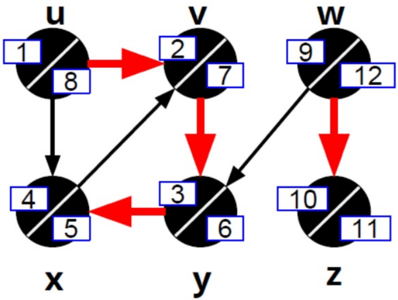
- **Timestamps** generated by the DFS algorithm have parenthesis structure.

Parenthesis Structure:
- Either one pair **contains** the another pair.
- Or one pair is **disjoint** from another

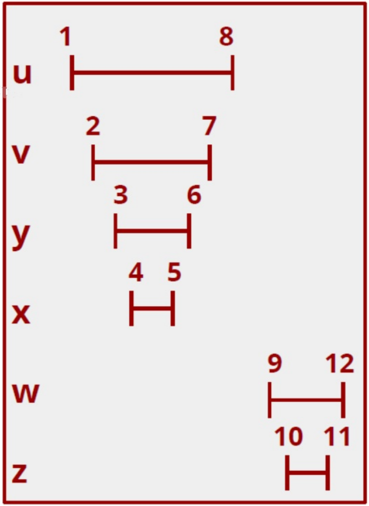
((())) () (())

Overlapping **never** happens:

(())



u
v
w
1
8
2
7
9
12
4
5
3
6
10
11
x
y
z



1
8
u
2
7
v
3
6
y
4
5
x
9
12
w
10
11
z

Parenthesis Theorem (Theorem 22.7 of CLRS):
In any depth-first search of a graph G, for any two vertices u and v

- interval $[v.d, v.f]$ contains interval $[u.d, u.f]$, or
- interval $[u.d, u.f]$ contains interval $[v.d, v.f]$, or
- $[v.d, v.f]$ and $[u.d, u.f]$ are disjoint (no overlap).

Nesting of Descendants' Intervals (Corollary 22.8 of CLRS):
In the depth-first forest for a graph G,
vertex v is a proper descendant of vertex u iff
the interval $[u.d, u.f]$ contains $[v.d, v.f]$.
That is, $u.d < v.d < v.f < u.f$.

Applications of DFS

- Detecting Cycles in Graphs.

- Topological Sort

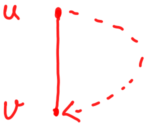
- Finding Strongly Connected Components (Section 22.5 of CLRS, ~~Optional~~)

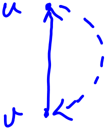
Applications of DFS: Detecting Cycles

In a graph, a cycle is path from an vertex u to **itself**.

If we know that there exists an **edge** between v and u, and also there exists **another path** between u and v (other than the edge (v, u)), we can say that there exists a path from u to itself, and therefore the graph has a cycle.

General Case (both directed and undirected graphs):
Consider a graph G.
Suppose u is an **ancestor** of v in a DFS-forest of G.
This means that there exists a **path** from u to v.
Now assume that there is an **edge** from v to u.
Then we can say that a **cycle** is detected.

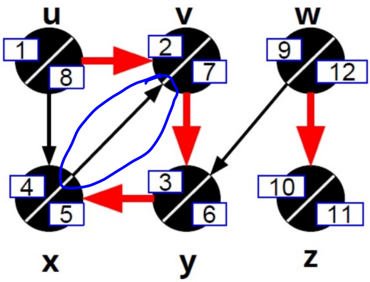




Applications of DFS: Detecting Cycles

- **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.

Lemma 22.11 of CLRS: A graph contains a cycle iff DFS yields a back edge.

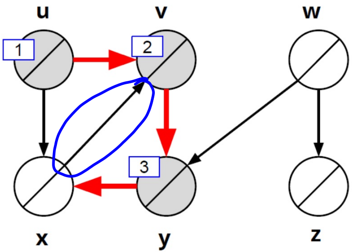


Applications of DFS: Detecting Cycles

How to identify a back edge?

- When performing DFS, look for edges to Gray vertices.
  If such an edge exists, it is a back edge.

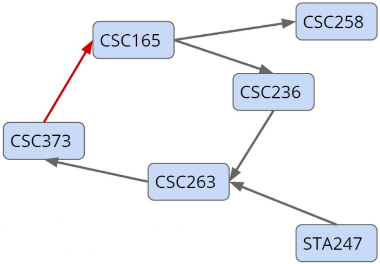
- **Reason**: In DFS, ancestors of the vertex being visited are Gray.



Why do we care about detecting cycles?

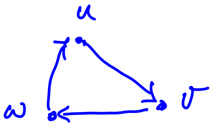
1. For Topological Sort.
2. If the edges represent dependency relations, then having a cycle implies cyclic dependency.

Example:
Course prerequisite graph: If the graph has a cycle, all courses in the cycle become impossible to take!



A **topological sort** of a directed graph $G = \langle V, E \rangle$ is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.
If the graph contains a **cycle**, then **no linear ordering** is possible.

Intuition: a topological sort of a graph is an ordering of its **vertices** along a **horizontal line** so that all directed edges go from **left to right**.

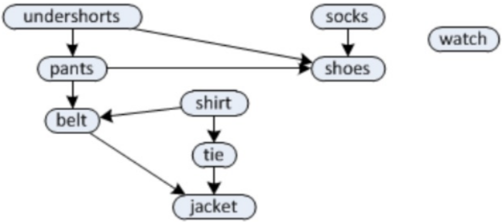


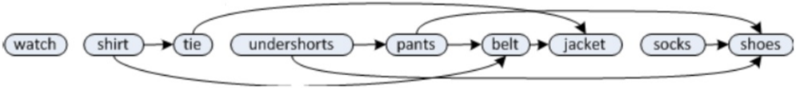
u v

Applications of DFS: Topological Sort

Dressing Order Example: Must don certain garments before others (e.g., socks before shoes).

A directed edge (u, v) indicates that garment u must be donned before garment v.





A topological sort of the graph gives an order for getting dressed.

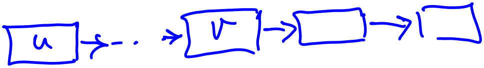
Topological Sort: High-level Description

For any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$
Proof: Left as an exercise.
You should do a proof by contradiction. To check your answer see Theorem 22.12 in CLRS.

$u \longrightarrow v$

TopologicalSort(G)

1. Call $DFS(G)$.
2. During DFS make sure that G does not contain any circles. At any points if a circle is detected return an empty list.
3. As each vertex is finished, insert it onto the front of a linked list.
4. Return the linked list of vertices.



Note: Topological sorting is **different** from the usual kind of sorting (like quick sort or heap sort).

After Lecture

- After-lecture Readings and Practice Problems: Chapter 6 of the course notes
- Optional Readings: CLRS Sections 22.1, 22.2, 22.3, 22.4
- Problems 22.1-1, 22.1-2, 22.2-1, 22.3-2. in CLRS.