THIS TIME

- Depth-first search (DFS)
  - Complexity
  - DFS forest edge types
- Application: cycle detection via DFS
- Application: topological sort via DFS (also without DFS)
DEPTH FIRST SEARCH (DFS)
BASIC DFS PROPERTIES TO RECALL

- Nodes start **white**
- A node \( v \) turns **gray** when it is **discovered**, which is when the first call to \( DFSVisit(v) \) happens
- **After** \( v \) is turned **gray**, we recurse on its neighbours
- After recursing on **all neighbours**, we turn \( v \) **black**
  - Recursive calls on neighbours end first, so neighbours turn black first

Also gets a discovery time \( d[v] \) at this point
Algorithm: $DFS(G)$
for each $v \in V(G)$
    do $\{$
        $\text{colour}[v] \leftarrow \text{white}$
        $\pi[v] \leftarrow \emptyset$
    $\}$
    time $\leftarrow 0$
for each $v \in V(G)$
    do $\{$
        if $\text{colour}[v] = \text{white}$
            then $DFS\text{visit}(v)$
        $\}$

Algorithm: $DFS\text{visit}(v)$
$\text{colour}[v] \leftarrow \text{gray}$
time $\leftarrow \text{time} + 1$
d$[v] \leftarrow \text{time}$
comment: $d[v]$ is the discovery time for vertex $v$
for each $w \in \text{Adj}[v]$
    do $\{$
        if $\text{colour}[w] = \text{white}$
            then $\{$
                $\pi[w] \leftarrow v$
                $DFS\text{visit}(w)$
            $\}$
        $\}$
$\text{colour}[v] \leftarrow \text{black}$
time $\leftarrow \text{time} + 1$
f$[v] \leftarrow \text{time}$
comment: $f[v]$ is the finishing time for vertex $v$

Complexity with adjacency lists is $O(n + m)$

Home exercise: complexity with adjacency matrix?
CLASSIFYING EDGES IN THE DFS FOREST
RECALL THE EXAMPLE:

Initial call: \textit{DFSvisit}(1), recursive calls: \textit{DFSvisit}(2), \textit{DFSvisit}(3), \textit{DFSvisit}(4).

Initial call: \textit{DFSvisit}(5), recursive call: \textit{DFSvisit}(6).

The depth-first forest consists of two trees. One tree has arcs 12, 23, 34 (initial call from \textit{DFSvisit}(1)) and the other tree has arc 56 (initial call from \textit{DFSvisit}(5)).
Classification of Edges in Depth-first Search

What are the edge types in the example graph?

$uv$ is a **tree edge** if $u = \pi[v]$

$uv$ is a **forward edge** if it is not a tree edge, and $v$ is a descendant of $u$ in a tree in the depth-first forest

$uv$ is a **back edge** if $u$ is a descendant of $v$ in a tree in the depth-first forest

any other edge is a **cross edge**.

Can we classify edges **without** inspecting the DFS forest? Perhaps using $d[...]$, $f[...]$ and/or $\text{colour}[...]$?
**Definitions**

- **Definition:** we use $I_u$ to denote $(d[u], f[u])$, which we call the **interval of $u$**

- **Definition:** $v$ is **white-reachable from $u$** if there is a path from $u$ to $v$ containing **only white nodes**
EXPLORING D[], F[] AND COLOUR[]

• **Observe:** every node \( v \) that is white-reachable from \( u \) when we call \( DFSVisit(u) \) (discover \( u \)) becomes **gray** after \( u \) and **black** before \( u \) (so \( I_v \) is nested inside \( I_u \))

- Start \( DFSVisit(u) \) and colour \( u \) grey
- Perform \( DFSVisit \) calls recursively...
- Colour \( u \) black and return from \( DFSVisit(u) \)

Note: \( v \) is discovered during the recursive \( DFSVisit \) calls, which is equivalent to \( v \) being a descendent of \( u \) in the DFS forest
MORE GENERALLY

- **Theorem**: Let $u, v$ be any nodes. The following statements are all equivalent.
  - $(v$ is discovered during $DFSVisit(u)$)
  - $(v$ is white-reachable from $u$ when we call $DFSVisit(u)$)
  - $(v$ is a descendant of $u$ in the DFS forest)
  - $(v$ turns grey after $u$ and black before $u$)
  - $(I_v$ nested inside $I_u$)
DFS inspects every edge in the graph. When DFS inspects an edge \{u, v\}, the colour of v and relationship between the intervals of u and v determine the edge type.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of v</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>Q1?</td>
<td>Q2?</td>
</tr>
<tr>
<td>forward</td>
<td>Q4?</td>
<td>Q3?</td>
</tr>
<tr>
<td>back</td>
<td>Q6?</td>
<td>Q5?</td>
</tr>
<tr>
<td>cross</td>
<td>Q8?</td>
<td>Q7?</td>
</tr>
</tbody>
</table>

Recall: (v is discovered during DFSVisit(u))
⇔ (v is white-reachable from u when we call DFSVisit(u))
⇔ (v is a descendant of u in the DFS forest)
⇔ (v turns grey after u and black before u)
⇔ (I_v nested inside I_u)

\(v\) discovered during DFSVisit(u) but not at the first level of recursion (else \{u, v\} is a tree edge). So the recursive call in DFSVisit(u) that discovered v must finish (and set v black) before DFSVisit(u) inspects \{u, v\}.
**USEFUL FACT: PARENTHESIS THEOREM**

- **Theorem:** for each pair of nodes $u, v$ the intervals of $u$ and $v$ are either **disjoint** or **nested**

- **Proof:** Suppose the intervals are **not** disjoint.
  - Then either $d[v] \in I_u$ or $d[u] \in I_v$
  - WLOG suppose $d[v] \in I_u$
  - Then $v$ is discovered during $DFSVisit(u)$, so $v$ is white-reachable from $u$ when $DFSVisit(u)$ starts
  - So, $v$ must turn black before $u$, so $f[v] < f[u]$
  - So the intervals are **nested**. QED
DFS inspects **every edge** in the graph. When DFS inspects an edge \{u, v\}, the colour of \(v\) and relationship between the intervals of \(u\) and \(v\) determine the **edge type**.

**Recall:**  

\[(v \text{ is discovered during } \text{DFSVisit}(u)) \iff (v \text{ is } \text{white-reachable} \text{ from } u \text{ when we call } \text{DFSVisit}(u)) \iff (v \text{ is a descendant of } u \text{ in the DFS forest}) \iff (I_v \text{ nested inside } I_u)\]

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of (v)</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>(d[u] &lt; d[v] &lt; f[v] &lt; f[u])</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>(d[u] &lt; d[v] &lt; f[v] &lt; f[u])</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>(d[v] &lt; d[u] &lt; f[u] &lt; f[v])</td>
</tr>
<tr>
<td>cross</td>
<td>Q8?</td>
<td>Q7?</td>
</tr>
</tbody>
</table>

If \(I_u\) were earlier, then \(v\) would be **discovered before \(u\) finishes** (because of edge \{u, v\}), so intervals would not be disjoint!  

Intervals \(I_u\) and \(I_v\) must be **disjoint**. But which is earlier?  

\(v\) is **not** a descendent, and **not** an ancestor.

So, \(I_v\) must be earlier.
DFS APPLICATION:

TESTING WHETHER A GRAPH IS A DAG

A directed graph $G$ is a directed acyclic graph, or DAG, if $G$ contains no directed cycle.
Lemma 6.7

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

Proof.

(⇒): Any back edge creates a directed cycle.
Case (↔): Suppose ∃ directed cycle. Show ∃ back edge.

- Let \( v_1, v_2, \ldots, v_k, v_1 \) be a directed cycle.
- WLOG let \( v_1 \) be earliest discovered node in the cycle.

Consider edge \( \{v_k, v_1\} \)

Since \( d[v_1] < d[v_k] \), \( \{v_k, v_1\} \) must be a back or cross edge. Why?

Discovered before \( v_2, \ldots, v_k \)

Recall: nodes become gray when discovered

Recall: every node \( v_i \) that is white-reachable from \( v_1 \) when we discover \( v_1 \) (call DFSvisit\( (v_1) \)) turns black before \( v_1 \) (\( f[v_i] < f[v_1] \)).

So \( v_k \) must turn black before \( v_1 \), and we have \( f[v_k] < f[v_1] \).

Thus, \( \{v_k, v_1\} \) must be a back edge. QED
When we observe an edge from $u$ to $v$, check if $v$ is gray.

**Lemma 6.7**

A directed graph is a DAG if and only if a depth-first search encounters no back edges.

- Search for back edges
- How to identify a back-edge?

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of $v$</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>$d[v] &lt; d[u] &lt; f[u] &lt; f[v]$</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>$d[v] &lt; f[v] &lt; d[u] &lt; f[u]$</td>
</tr>
</tbody>
</table>

When we observe an edge from $u$ to $v$, check if $v$ is gray.
DFS: TESTING WHETHER A GRAPH IS A DAG

Algorithm: $DFS(G)$

| $DAG \leftarrow \text{true}$ |

for each $v \in V(G)$

    do $\begin{cases} \text{colour}[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}$

    time $\leftarrow 0$

for each $v \in V(G)$

    do $\begin{cases} \text{if colour}[v] = \text{white} \\ \text{then } DFSvisit(v) \end{cases}$

return $(DAG)$
Algorithm: $DFSvisit(v)$

\begin{align*}
&\text{colour}[v] \leftarrow \text{gray} \\
&\text{time} \leftarrow \text{time} + 1 \\
&d[v] \leftarrow \text{time} \\
&\text{comment: } d[v] \text{ is the discovery time for vertex } v \\
&\text{for each } w \in \text{Adj}[v] \\
&\quad \text{do } \begin{cases} \\
&\quad \quad \text{if } \text{colour}[w] = \text{white} \\
&\quad \quad \text{then } \begin{cases} \\
&\quad \quad \quad \pi[w] \leftarrow v \\
&\quad \quad \quad \text{DFSvisit}(w) \\
&\quad \quad \text{if } \text{colour}[w] = \text{gray} \text{ then } \text{DAG} \leftarrow \text{false} \\
&\quad \end{cases} \\
&\quad \end{cases} \\
&\text{colour}[v] \leftarrow \text{black} \\
&\text{time} \leftarrow \text{time} + 1 \\
&f[v] \leftarrow \text{time}
\end{align*}
HOME EXERCISE

• Try running the algorithm on this graph:
TOPOLOGICAL SORT
Finding node orderings that satisfy given constraints
Example problem: getting dressed in the morning

- Pants before belt
- Socks before shoes
- Watch any time

Could do various things first. Which ones are possible? What do they have in common?

Edge \( \{u, v\} \) means \( u \) must be completed before \( v \)
Topological sort

Try to order nodes linearly so there are only pointers from left to right!

IFF there is a (directed) cycle!

Might not be possible! How can this happen?

Try to order nodes linearly so there are **only** pointers from left to right!
MORE REALISTIC USE CASES (1/2)

- Compiler design: ordering program compilation steps

  Not just compilers... Linkers, build systems (makefiles, etc.)

  Cell evaluation-order in spreadsheets...

  Manufacturing processes, chemical reaction processes, ... all sorts of planning and scheduling

  Can do parse.y, then parse.h, then parse.c, then main.c, then parse.o, then main.o, then prog...

  Can verify pointers only go left-to-right. Other orders are possible...
Goal: output a serial order of tasks that can be run without worrying about dependencies (because all dependencies are already satisfied by the time a task is run).

(Nodes are numbered according to one such order.)

Can even schedule tasks to run in parallel! Can do 1||2 then 3||13 then 4||5||15 etc.

Similar to instruction scheduling and pipelining in modern processors!
A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in $V$ such that $u < v$ whenever $uv \in E$.

Here is a topological ordering—all edges are directed from left to right:
**Lemma 6.5**

A DAG contains a vertex of indegree 0.

**Proof.**

Suppose we have a directed graph in which every vertex has positive indegree. Let $v_1$ be any vertex. For every $i \geq 1$, let $v_{i+1}v_i$ be an arc. In the sequence $v_1, v_2, v_3, \ldots$, consider the first repeated vertex, $v_i = v_j$ where $j > i$. Then $v_j, v_{j-1}, \ldots, v_i, v_j$ is a directed cycle.

One of these must be repeated. So there is a cycle!
**EXISTENCE OF TOPOLOGICAL SORT**

**Theorem 6.6**

A directed graph $D$ has a topological sort if and only if it is a DAG.

**Proof.**

$(\Rightarrow)$: Suppose $D$ has a directed cycle $v_1, v_2, \ldots, v_j, v_1$. Then $v_1 < v_2 < \cdots < v_j < v_1$, so a topological ordering does not exist.

$(\Leftarrow)$: Suppose $D$ is a DAG. Then the algorithm below constructs a topological ordering.

**Algorithm:** $\text{TopOrdering}(D)$

1. $D_1 \leftarrow D$
2. for $i \leftarrow 1$ to $n$
   - do let $v_i$ be a vertex in $D_i$ having indegree 0
     - construct $D_{i+1}$ from $D_i$ by deleting $v_i$
   - return $(v_1, v_2, \ldots, v_n)$

Not a very efficient algorithm

We can build a more efficient algorithm by using DFS
Lemma 6.8
Suppose $D$ is a DAG. Then $f[v] < f[u]$ for every arc $uv$.

Proof.
Look at the classification on slide # 13. In a DAG, there are no back edges. For any other type of arc $uv$, it holds that $f[v] < f[u]$. □

Therefore, if $D$ is a DAG and we order the vertices in reverse order of finishing time, then we get a topological ordering.

To see why we get a topological ordering, suppose $D$ is a DAG and we order nodes from largest to smallest finish time, so $f_{v_1} > f_{v_2} > \cdots > f_{v_{n-1}} > f_{v_n}$

Suppose a right-to-left edge $\{v_j, v_i\}$ exists.

Since edge $\{v_j, v_i\}$ exists, the lemma implies $f_{v_i} < f_{v_j}$.

But this contradicts the node ordering!

So edges are left-to-right, hence this is a topological order!
TOPOLOGICAL ORDERING VIA DFS

Algorithm: $DFS(G)$

1. $InitializeStack(S)$
2. $DAG \leftarrow true$
3. for each $v \in V(G)$
   - do $egin{cases} colour[v] \leftarrow white \\ \pi[v] \leftarrow \emptyset \end{cases}$
4. $time \leftarrow 0$
5. for each $v \in V(G)$
   - do $egin{cases} \text{if } colour[v] = \text{white} \text{ then } DFSvisit(v) \end{cases}$
6. if $DAG$ then return $(S)$ else return $(DAG)$
Algorithm: DFSvisit($v$)

1. $colour[v] \leftarrow \text{gray}$
2. $time \leftarrow time + 1$
3. $d[v] \leftarrow time$
4. comment: $d[v]$ is the discovery time for vertex $v$
5. for each $w \in \text{Adj}[v]$
   - if $colour[w] = \text{white}$
     - then ($\pi[w] \leftarrow v$
     - $\text{DFSvisit}(w)$
   - if $colour[w] = \text{gray}$ then $\text{DAG} \leftarrow \text{false}$
6. $colour[v] \leftarrow \text{black}$
7. $\text{Push}(S, v)$
8. $time \leftarrow time + 1$
9. $f[v] \leftarrow time$

Running time? $O(n + m)$ with adjacency lists

Save each node when it finishes

We are pushing smallest finishing times first into a stack, so when we pop them out, we will get largest finishing time first
The initial calls are $DFSvisit(1)$, $DFSvisit(2)$ and $DFSvisit(3)$.

The discovery/finish times are as follows:

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d[v]$</th>
<th>$f[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d[v]$</th>
<th>$f[v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The topological ordering is 3, 2, 5, 4, 1, 6 (reverse order of finishing time).
Algorithm: *Kahn*(D)

- compute $\text{deg}^-(v)$ for all vertices $v$
- for all $v$ such that $\text{deg}^-(v) = 0$, insert $v$ into a queue $Q$
- for $i \leftarrow 1$ to $n$
  - if $Q$ is empty
    - then return ()
  - else
    - let $v$ be the first vertex in $Q$
    - remove $v$ from $Q$ and output $v$
    - for all $w$ in $\text{Adj}(v)$
      - $\text{deg}^-(w) \leftarrow \text{deg}^-(w) - 1$
      - do
        - if $\text{deg}^-(w) = 0$
          - then insert $w$ into $Q$

Running time is $O(n + m)$ with adjacency lists
HOME EXERCISE (KAHN’S ALGORITHM)

Run on this graph...