Pre-lecture brain teaser

Last time we looked at the BasicSearch algorithm:

\[ \text{Explore}(G,u): \]
\[ \text{Visited}[1\ldots n] \leftarrow \text{FALSE} \]
\[ \text{Add } u \text{ to } S \]
\[ \text{Visited}[u] \leftarrow \text{TRUE} \]
\[ \text{ExploreStep}(G,u,\text{Visited},S) \]
\[ \text{Output } S \]

\[ \text{ExploreStep}(G,x,\text{Visited},S): \]
\[ \text{for each edge } xy \text{ in } \text{Adj}(x) \text{ do} \]
\[ \text{if } (\text{Visited}[y] = \text{FALSE}) \]
\[ \text{Visited}[y] \leftarrow \text{TRUE} \]
\[ \text{ExploreStep}(G,x,\text{Visited},S): \]
\[ \text{return} \]

We said that if ToExplore was a:
- Stack, the algorithm is equivalent to DFS
- Queue, the algorithm is equivalent to BFS

What if the algorithm was written recursively (instead of the while loop, you recursively call explore). What would the algorithm be equivalent to?
ECE-374-B: Lecture 16 - Directed Graphs (DFS, DAGs, Topological Sort)

Instructor: Nickvash Kani
March 21, 2023

University of Illinois at Urbana-Champaign
Last time we looked at the BasicSearch algorithm:

```
Explore(G, u):
    Visited[1..n] ← FALSE
    Add u to S
    Visited[u] ← TRUE
    ExploreStep(G, u, Visited, S)
Output S

ExploreStep(G, x, Visited, S):
    for each edge xy in Adj(x) do
        if (Visited[y] = FALSE)
            Visited[y] ← TRUE
            ExploreStep(G, x, Visited, S):
    return
```

We said that if ToExplore was a:

- Stack, the algorithm is equivalent to DFS
- Queue, the algorithm is equivalent to BFS

What if the algorithm was written recursively (instead of the while loop, you recursively call explore). What would the algorithm be equivalent to?
Directed Acyclic Graphs - definition and basic properties
**Definition**
A directed graph $G$ is a **directed acyclic graph (DAG)** if there is no directed cycle in $G$. 

![Diagram of a directed acyclic graph]
Is this a DAG?
Is this a DAG?
Sources and Sinks

Definition

• A vertex $u$ is a **source** if it has no in-coming edges.
• A vertex $u$ is a **sink** if it has no out-going edges.
Proposition
Every DAG $G$ has at least one source and at least one sink.
Proposition
Every DAG $G$ has at least one source and at least one sink.

Proof.
Let $P = v_1, v_2, \ldots, v_k$ be a longest path in $G$. Claim that $v_1$ is a source and $v_k$ is a sink. Suppose not. Then $v_1$ has an incoming edge which either creates a cycle or a longer path both of which are contradictions. Similarly if $v_k$ has an outgoing edge. \qed
Topological ordering
Order or strict total order on a set $X$ is a binary relation $\prec$ on $X$, such that

- Transitivity: $\forall x, y, z \in X \quad x \prec y$ and $y \prec z \implies x \prec z$.
- For any $x, y \in X$, exactly one of the following holds: $x \prec y$, $y \prec x$ or $x = y$. 
Convention about writing edges

- **Undirected graph edges:**

  \[ uv = \{u, v\} = vu \in E \]

- **Directed graph edges:**

  \[ u \to v \equiv (u, v) \equiv (u \to v) \]
**Definition**

A topological ordering/topological sorting of $G = (V, E)$ is an ordering $≺$ on $V$ such that if $(u \to v) \in E$ then $u ≺ v$.

Informal equivalent definition: One can order the vertices of the graph along a line (say the $x$-axis) such that all edges are from left to right.
Exercise: show algorithm can be implemented in $O(m + n)$ time.
Exercise: show algorithm can be implemented in $O(m + n)$ time.

Simple Algorithm:

1. Count the in-degree of each vertex
2. For each vertex that is source ($\text{deg}_{in}(v) = 0)$:
   2.1 Add $v$ to the topological sort
   2.2 Lower degree of vertices $v$ is connected to.  

\footnote{1}
Topological Sort: Example

Adjacency List:

<table>
<thead>
<tr>
<th>Node</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d, e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>h, g</td>
</tr>
<tr>
<td>f</td>
<td>h</td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

Generate $\text{deg}_{in}(v)$:

<table>
<thead>
<tr>
<th>Degree</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b, c</td>
</tr>
<tr>
<td>1</td>
<td>d, f, g</td>
</tr>
<tr>
<td>2</td>
<td>e, h</td>
</tr>
</tbody>
</table>
Topological Sort: Example

Adjacency List:

<table>
<thead>
<tr>
<th>Node</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d, e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>h, g</td>
</tr>
<tr>
<td>f</td>
<td>h</td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

Generate $deg_{in}(v)$:

<table>
<thead>
<tr>
<th>Degree</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b, c</td>
</tr>
<tr>
<td>1</td>
<td>d, f, g</td>
</tr>
<tr>
<td>2</td>
<td>e, h</td>
</tr>
</tbody>
</table>

Topological Ordering:
Multiple possible topological orderings
DAGs and Topological Sort

• **Note:** A DAG $G$ may have many different topological sorts.

• Exercise: What is a DAG with the most number of distinct topological sorts for a given number $n$ of vertices?

• Exercise: What is a DAG with the least number of distinct topological sorts for a given number $n$ of vertices?
Direct Topological ordering - code

```
TopSort(G):
    Sorted ← NULL
    deg_in[1..n] ← −1
    Tdeg_in[1..n] ← NULL
    Generate in-degree for each vertex
    for each edge xy in G do
        deg_in[y] ++
    for each vertex v in G do
        Tdeg_in[deg_in[v]].append(v)
    Next we recursively add vertices
    with in-degree = 0 to the sort list
    while (Tdeg_in[0] is non-empty) do
        Remove node x from Tdeg_in[0]
        Sorted.append(x)
        for each edge xy in Adj(x) do
            deg_in[y] --
            move y to Tdeg_in[deg_in[y]]
    Output Sorted
```
Lemma
A directed graph $G$ can be topologically ordered $\iff G$ is a DAG.

Proof.
Proof by contradiction. Suppose $G$ is not a DAG and has a topological ordering $\prec$. $G$ has a cycle

$$C = u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow u_1.$$ 

Then $u_1 \prec u_2 \prec \cdots \prec u_k \prec u_1$
Lemma
A directed graph $G$ can be topologically ordered $\implies G$ is a DAG.

Proof.
Proof by contradiction. Suppose $G$ is not a DAG and has a topological ordering $\prec$. $G$ has a cycle

$$C = u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \rightarrow u_1.$$

Then $u_1 \prec u_2 \prec \cdots \prec u_k \prec u_1$

$\implies u_1 \prec u_1.$

A contradiction (to $\prec$ being an order). Not possible to topologically order the vertices. $\square$
An explicit definition of what topological ordering of a graph is

For a graph $G = (V, E)$ a **topological ordering** of a graph is a numbering $\pi : V \rightarrow \{1, 2, \ldots, n\}$, such that

$$\forall (u \rightarrow v) \in E(G) \implies \pi(u) < \pi(v).$$

(That is, $\pi$ is one-to-one, and $n = |V|$)
Example...
Example...

Assuming:

\[ V = \{a, \ldots w\} \]
\[ \pi = \{1, \ldots 23\} \]
Depth First Search (DFS)
Depth First Search (DFS) in Undirected Graphs
• **DFS** special case of Basic Search.
• **DFS** is useful in understanding graph structure.
• **DFS** used to obtain linear time ($O(m + n)$) algorithms for
  • Finding cut-edges and cut-vertices of undirected graphs
  • Finding strong connected components of directed graphs
• ...many other applications as well.
DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

**DFS(G)**

for all $u \in V(G)$ do

Mark $u$ as unvisited
Set pred($u$) to null

$T$ is set to $\emptyset$

while $\exists$ unvisited $u$ do

DFS($u$)

Output $T$

**DFS($u$)**

Mark $u$ as visited

for each $uv$ in $\text{Out}(u)$ do

if $v$ is not visited then

add edge $uv$ to $T$

set pred($v$) to $u$

DFS($v$)

Implemented using a global array $\text{Visited}$ for all recursive calls.

$T$ is the search tree/forest.
Edges classified into two types: \( uv \in E \) is a

- **tree edge**: belongs to \( T \)
- **non-tree edge**: does not belong to \( T \)
Edges classified into two types: \( uv \in E \) is a

- **tree edge**: belongs to \( T \)
- **non-tree edge**: does not belong to \( T \)
DFS with pre-post numbering
Keep track of when nodes are visited.

**DFS(G)**

```plaintext
for all \( u \in V(G) \) do
  Mark \( u \) as unvisited
  \( T \) is set to \( \emptyset \)
  \( time = 0 \)
while \( \exists \) unvisited \( u \) do
  DFS(\( u \))
Output \( T \)
```

**DFS(\( u \))**

```plaintext
Mark \( u \) as visited
\( pre(u) = ++time \)
for each \( uv \) in \( Out(u) \) do
  if \( v \) is not marked then
    add edge \( uv \) to \( T \)
    DFS(\( v \))
post(\( u \)) = ++\( time \)
```
Animation

\[ time = 0 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
$time = 1$

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([1, )</td>
</tr>
</tbody>
</table>
$time = 1$

<table>
<thead>
<tr>
<th>vertex</th>
<th>$[pre, post]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
</tbody>
</table>
Animation

\[ \text{time} = 2 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
</tbody>
</table>

![Diagram showing a graph with vertices labeled 1 to 10, and edges illustrating the pre and post orders at time 2.](image-url)
$time = 2$

<table>
<thead>
<tr>
<th>vertex</th>
<th>$[pre, post]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
</tbody>
</table>
**Animation**

\[ \text{time} = 3 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([\text{pre, post}])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
</tbody>
</table>

![Graph Diagram]

---
**Animation**

\[ \text{time} = 4 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Animation

\[ time = 5 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
<tr>
<td>6</td>
<td>[5, ]</td>
</tr>
</tbody>
</table>

\[ \]
Animation

\[
\text{time} = 6
\]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([1, ])</td>
</tr>
<tr>
<td>2</td>
<td>([2, ])</td>
</tr>
<tr>
<td>4</td>
<td>([3, ])</td>
</tr>
<tr>
<td>5</td>
<td>([4, ])</td>
</tr>
<tr>
<td>6</td>
<td>([5, 6])</td>
</tr>
</tbody>
</table>

Diagram showing a graph with vertices 1 to 6 and edges connecting them.
**Animation**

\[
\text{time} = 7
\]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([1,\ ])</td>
</tr>
<tr>
<td>2</td>
<td>([2,\ ])</td>
</tr>
<tr>
<td>4</td>
<td>([3,\ ])</td>
</tr>
<tr>
<td>5</td>
<td>([4,\ ])</td>
</tr>
<tr>
<td>6</td>
<td>([5,\ 6])</td>
</tr>
<tr>
<td>3</td>
<td>([7,\ ])</td>
</tr>
</tbody>
</table>
**time** = 8

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, ]</td>
</tr>
<tr>
<td>7</td>
<td>[8, ]</td>
</tr>
</tbody>
</table>
**Animation**

\[ \text{time} = 9 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([\text{pre}, \text{post}])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, ]</td>
</tr>
<tr>
<td>7</td>
<td>[8, ]</td>
</tr>
<tr>
<td>8</td>
<td>[9, ]</td>
</tr>
</tbody>
</table>

![Graph with vertices and edges](image)
**Animation**

\[ \text{time} = 10 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, ]</td>
</tr>
<tr>
<td>7</td>
<td>[8, ]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
### Animation

**time = 11**

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, ]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, ]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>

![Graph Diagram](image-url)
Animation

\[ \text{time} = 12 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([1, )</td>
</tr>
<tr>
<td>2</td>
<td>([2, )</td>
</tr>
<tr>
<td>4</td>
<td>([3, )</td>
</tr>
<tr>
<td>5</td>
<td>([4, )</td>
</tr>
<tr>
<td>6</td>
<td>([5, 6] )</td>
</tr>
<tr>
<td>3</td>
<td>([7, 12] )</td>
</tr>
<tr>
<td>7</td>
<td>([8, 11] )</td>
</tr>
<tr>
<td>8</td>
<td>([9, 10] )</td>
</tr>
</tbody>
</table>
Animation

\[ \text{time} = 13 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, ]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
Animation

\[ time = 14 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, ]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
**Animation**

\[ \text{time} = 15 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([\text{pre}, \text{post}])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, ]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
Animation

\[ \text{time} = 16 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([pre, post])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
time = 17

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>9</td>
<td>[17, ]</td>
</tr>
</tbody>
</table>
**time** = 18

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>9</td>
<td>[17, ]</td>
</tr>
<tr>
<td>10</td>
<td>[18, ]</td>
</tr>
</tbody>
</table>
Animation

time = 19

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>9</td>
<td>[17, ]</td>
</tr>
<tr>
<td>10</td>
<td>[18, 19]</td>
</tr>
</tbody>
</table>
**Animation**

### time = 20

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>9</td>
<td>[17, 20]</td>
</tr>
<tr>
<td>10</td>
<td>[18, 19]</td>
</tr>
</tbody>
</table>
### Animation

<table>
<thead>
<tr>
<th>vertex</th>
<th>[pre, post]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, 16]</td>
</tr>
<tr>
<td>2</td>
<td>[2, 15]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 14]</td>
</tr>
<tr>
<td>5</td>
<td>[4, 13]</td>
</tr>
<tr>
<td>6</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>3</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>7</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>9</td>
<td>[17, 20]</td>
</tr>
<tr>
<td>10</td>
<td>[18, 19]</td>
</tr>
</tbody>
</table>
Node $u$ is active in time interval $[\text{pre}(u), \text{post}(u)]$

**Proposition**

*For any two nodes $u$ and $v$, the two intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are disjoint or one is contained in the other.*

pre and post numbers useful in several applications of DFS
DFS in Directed Graphs
DFS in Directed Graphs

\[ \text{DFS}(G) \]

Mark all nodes \( u \) as unvisited
\( T \) is set to \( \emptyset \)
\( time = 0 \)

while there is an unvisited node \( u \) do
  \[ \text{DFS}(u) \]
Output \( T \)

\[ \text{DFS}(u) \]
Mark \( u \) as visited
\( \text{pre}(u) = ++time \)

for each edge \( (u, v) \) in \( \text{Out}(u) \) do
  if \( v \) is not visited
    add edge \( (u, v) \) to \( T \)
  \[ \text{DFS}(v) \]
\( \text{post}(u) = ++time \)
Example of **DFS** in directed graph
Example of DFS in directed graph
DFS Properties

Generalizing ideas from undirected graphs:

- **DFS**\((G)\) takes **\(O(m + n)\)** time.
DFS Properties

Generalizing ideas from undirected graphs:

- **DFS**\((G)\) takes \(O(m + n)\) time.
- Edges added form a branching: a forest of out-trees. 
  
  Output of **DFS**\((G)\) depends on the order in which vertices are considered.
Generalizing ideas from undirected graphs:

- **DFS**\((G)\) takes \(O(m + n)\) time.
- Edges added form a **branching**: a forest of out-trees. 
  **Output of DFS**\((G)\) **depends on the order in which vertices are considered.**
- If \(u\) is the first vertex considered by **DFS**\((G)\) then **DFS**\((u)\) outputs a directed out-tree \(T\) rooted at \(u\) and a vertex \(v\) is in \(T\) if and only if \(v \in rch(u)\)
DFS Properties

Generalizing ideas from undirected graphs:

- $\text{DFS}(G)$ takes $O(m + n)$ time.
- Edges added form a branching: a forest of out-trees. **Output of $\text{DFS}(G)$ depends on the order in which vertices are considered.**
- If $u$ is the first vertex considered by $\text{DFS}(G)$ then $\text{DFS}(u)$ outputs a directed out-tree $T$ rooted at $u$ and a vertex $v$ is in $T$ if and only if $v \in \text{rch}(u)$
- For any two vertices $x, y$ the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint or one is contained in the other.
Generalizing ideas from undirected graphs:

- **DFS**\((G)\) takes \(O(m + n)\) time.
- Edges added form a branching: a forest of out-trees. **Output of** **DFS**\((G)\) **depends on the order in which vertices are considered.**
- If \(u\) is the first vertex considered by **DFS**\((G)\) then **DFS**\((u)\) outputs a directed out-tree \(T\) rooted at \(u\) and a vertex \(v\) is in \(T\) if and only if \(v \in \text{rch}(u)\)
- For any two vertices \(x, y\) the intervals \([\text{pre}(x), \text{post}(x)]\) and \([\text{pre}(y), \text{post}(y)]\) are either disjoint or one is contained in the other.
Edges of $G$ can be classified with respect to the **DFS** tree $T$ as:

- **Tree edges** that belong to $T$
- A **forward edge** is a non-tree edges $(x,y)$ such that $y$ is a descendant of $x$.
- A **backward edge** is a non-tree edge $(x,y)$ such that $y$ is an ancestor of $x$.
- A **cross edge** is a non-tree edges $(x,y)$ such that they don’t have a ancestor/descendant relationship between them.
DFS tree and related edges

Edges of $G$ can be classified with respect to the DFS tree $T$ as:

- **Tree edges** that belong to $T$
- A **forward** edge is a non-tree edges $(x, y)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A **backward** edge is a non-tree edge $(x, y)$ such that .
- A **cross** edge is a non-tree edges $(x, y)$ such that
Types of Edges

- Back edges:
- Forward edges:
- Cross edges:

![Diagram showing different types of edges with labeled arrows and numbers between nodes]
Types of Edges

- Back edges:
- Forward edges:
- Cross edges:
DFS and cycle detection: Topological sorting using DFS
Cycles in graphs

Given an undirected graph how do we check whether it has a cycle and output one if it has one?
Cycles in graphs

Given an undirected graph how do we check whether it has a cycle and output one if it has one?

**Question:** Given an directed graph how do we check whether it has a cycle and output one if it has one?
Cycle detection in directed graph using topological sorting

**Question**
Given G, is it a **DAG**?

If it is, compute a topological sort.

If it fails, then output the cycle C.
Topological sort a graph using DFS

**DFS based algorithm:**

- Compute **DFS**(G)
- If there is a back edge $e = (v, u)$ then G is not a **DAG**. Output cycle $C$ formed by path from $u$ to $v$ in $T$ plus edge $(v, u)$.
- Otherwise output nodes in decreasing post-visit order. **Note:** no need to sort, **DFS**(G) can output nodes in this order.
Topological sort a graph using DFS

**DFS based algorithm:**

- Compute \( \text{DFS}(G) \)
- If there is a back edge \( e = (v, u) \) then \( G \) is not a \textbf{DAG}. Output cycle \( C \) formed by path from \( u \) to \( v \) in \( T \) plus edge \( (v, u) \).
- Otherwise output nodes in decreasing post-visit order.

\textbf{Note:} no need to sort, \( \text{DFS}(G) \) can output nodes in this order.

Computes topological ordering of the vertices.

Algorithm runs in \( O(n + m) \) time.
Topological sort a graph using \textbf{DFS}

\textbf{DFS} based algorithm:

\begin{itemize}
  \item Compute \textbf{DFS}(G)
  \item If there is a back edge \(e = (v, u)\) then G is not a \textbf{DAG}. Output cycle \(C\) formed by path from \(u\) to \(v\) in \(T\) plus edge \((v, u)\).
  \item Otherwise output nodes in decreasing post-visit order. \textbf{Note}: no need to sort, \textbf{DFS}(G) can output nodes in this order.
\end{itemize}

Computes topological ordering of the vertices.

Algorithm runs in \(O(n + m)\) time. Correctness is not so obvious.

See next two propositions.
Listing out the vertices in post-number decreasing gives: c, b, a, e, g, d, f, h

Remind you of anything?
Listing out the vertices in post-number decreasing gives:

c, b, a, e, g, d, f, h

Remind you of anything?
Listing out the vertices in post-number decreasing gives:

c, b, a, e, g, d, f, h

Remind you of anything?
Proposition

$G$ has a cycle $\iff$ there is a back-edge in $\text{DFS}(G)$.

Proof.

If: $(u, v)$ is a back edge implies there is a cycle $C$ consisting of the path from $v$ to $u$ in $\text{DFS}$ search tree and the edge $(u, v)$.

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.

Let $v_i$ be first node in $C$ visited in $\text{DFS}$.

All other nodes in $C$ are descendants of $v_i$ since they are reachable from $v_i$.

Therefore, $(v_{i-1}, v_i)$ (or $(v_k, v_1)$ if $i = 1$) is a back edge. $\square$
Proposition
If $G$ is a DAG and $\text{post}(v) > \text{post}(u)$, then $(u \rightarrow v)$ is not in $G$.

Proof.
Assume $\text{post}(u) < \text{post}(v)$ and $(u \rightarrow v)$ is an edge in $G$. 
Proposition
If $G$ is a DAG and $\text{post}(v) > \text{post}(u)$, then $(u \to v)$ is not in $G$.

Proof.
Assume $\text{post}(u) < \text{post}(v)$ and $(u \to v)$ is an edge in $G$. One of two holds:

- Case 1: $[\text{pre}(u), \text{post}(u)]$ is contained in $[\text{pre}(v), \text{post}(v)]$.
- Case 2: $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$. 

$\blacksquare$
Decreasing post numbering is valid

Proposition
If $G$ is a DAG and $\text{post}(v) > \text{post}(u)$, then $(u \rightarrow v)$ is not in $G$.

Proof.
Assume $\text{post}(u) < \text{post}(v)$ and $(u \rightarrow v)$ is an edge in $G$. One of two holds:

- **Case 1:** $[\text{pre}(u), \text{post}(u)]$ is contained in $[\text{pre}(v), \text{post}(v)]$. Implies that $u$ is explored during $\text{DFS}(v)$ and hence is a descendent of $v$. Edge $(u, v)$ implies a cycle in $G$ but $G$ is assumed to be DAG!

- **Case 2:** $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$. This cannot happen since $v$ would be explored from $u$. 


We just proved:

**Proposition**

*If \( G \) is a DAG and \( \text{post}(v) > \text{post}(u) \), then \((u \rightarrow v)\) is not in \( G \).*

\[\Rightarrow\] sort the vertices of a DAG by decreasing post-numbering in decreasing order, then this numbering is valid.
Topological sorting

**Theorem**

$G = (V, E)$: Graph with n vertices and m edges.

Compute a topological sorting of $G$ using **DFS** in $O(n + m)$ time.

That is, compute a numbering $\pi : V \to \{1, 2, \ldots, n\}$, such that

$$(u \to v) \in E(G) \implies \pi(u) < \pi(v).$$
The meta graph of strong connected components
Algorithmic Problem
Find all SCCs of a given directed graph.
Previous lecture:
Saw an $O(n \cdot (n + m))$ time algorithm.
This lecture: sketch of a $O(n + m)$ time algorithm.
Graph of SCCs

Meta-graph of SCCs
Let $S_1, S_2, \ldots, S_k$ be the strong connected components (i.e., SCCs) of $G$. The graph of SCCs is $G^{\text{SCC}}$

- Vertices are $S_1, S_2, \ldots, S_k$
- There is an edge $(S_i, S_j)$ if there is some $u \in S_i$ and $v \in S_j$ such that $(u, v)$ is an edge in $G$. 
The meta graph of SCCs is a DAG...

**Proposition**
*For any graph $G$, the graph $G^{SCC}$ has no directed cycle.*

**Proof.**
If $G^{SCC}$ has a cycle $S_1, S_2, \ldots, S_k$ then $S_1 \cup S_2 \cup \cdots \cup S_k$ should be in the same SCC in $G$.  □
**To Remember: Structure of Graphs**

**Undirected graph:** connected components of $G = (V, E)$ partition $V$ and can be computed in $O(m + n)$ time.

**Directed graph:** the meta-graph $G^{SCC}$ of $G$ can be computed in $O(m + n)$ time. $G^{SCC}$ gives information on the partition of $V$ into strong connected components and how they form a DAG structure.

Above structural decomposition will be useful in several algorithms
Linear time algorithm for finding all SCCs
Finding all **SCCs** of a Directed Graph

**Problem**
Given a directed graph \( G = (V, E) \), output all its strong connected components.
Finding all SCCs of a Directed Graph

Problem
Given a directed graph $G = (V, E)$, output all its strong connected components.

Straightforward algorithm:

- Mark all vertices in $V$ as not visited.
- For each vertex $u \in V$ not visited yet do
  - Find $SCC(G, u)$ the strong component of $u$:
    - Compute $rch(G, u)$ using $DFS(G, u)$
    - Compute $rch(G^{rev}, u)$ using $DFS(G^{rev}, u)$
    - $SCC(G, u) \leftarrow rch(G, u) \cap rch(G^{rev}, u)$
  - $\forall u \in SCC(G, u)$: Mark $u$ as visited.

Running time: $O(n(n + m))$
Problem
Given a directed graph $G = (V, E)$, output all its strong connected components.

Straightforward algorithm:

Mark all vertices in $V$ as not visited.
for each vertex $u \in V$ not visited yet do
find $SCC(G, u)$ the strong component of $u$:
Compute $rch(G, u)$ using $DFS(G, u)$
Compute $rch(G^{rev}, u)$ using $DFS(G^{rev}, u)$
$SCC(G, u) \leftarrow rch(G, u) \cap rch(G^{rev}, u)$
$\forall u \in SCC(G, u)$: Mark $u$ as visited.

Running time: $O(n(n + m))$ Is there an $O(n + m)$ time algorithm?
Structure of a Directed Graph

Graph G

Reminder $G^{SCC}$ is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph $G$, its meta-graph $G^{SCC}$ is a DAG.
Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

- Let $u$ be a vertex in a sink SCC of $G^{\text{SCC}}$
- Do $\text{DFS}(u)$ to compute $\text{SCC}(u)$
- Remove $\text{SCC}(u)$ and repeat
Wishful Thinking Algorithm

- Let \( u \) be a vertex in a sink SCC of \( G^{SCC} \)
- Do \( \text{DFS}(u) \) to compute \( SCC(u) \)
- Remove \( SCC(u) \) and repeat

Justification

- \( \text{DFS}(u) \) only visits vertices (and edges) in \( SCC(u) \)
Linear-time Algorithm for SCCs: Ideas

Wishful Thinking Algorithm

- Let $u$ be a vertex in a sink SCC of $G^{SCC}$
- Do DFS($u$) to compute SCC($u$)
- Remove SCC($u$) and repeat

Justification

- DFS($u$) only visits vertices (and edges) in SCC($u$)
- ... since there are no edges coming out a sink!
Wishful Thinking Algorithm

- Let $u$ be a vertex in a sink SCC of $G^{SCC}$
- Do $\text{DFS}(u)$ to compute $\text{SCC}(u)$
- Remove $\text{SCC}(u)$ and repeat

Justification

- $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
- ... since there are no edges coming out a sink!
- $\text{DFS}(u)$ takes time proportional to size of $\text{SCC}(u)$
Wishful Thinking Algorithm

- Let $u$ be a vertex in a sink SCC of $G^{SCC}$
- Do $\text{DFS}(u)$ to compute $\text{SCC}(u)$
- Remove $\text{SCC}(u)$ and repeat

Justification

- $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
- ... since there are no edges coming out a sink!
- $\text{DFS}(u)$ takes time proportional to size of $\text{SCC}(u)$
- Therefore, total time $O(n + m)!$
Big Challenge(s)

How do we find a vertex in a sink $SCC$ of $G^{SCC}$?
Big Challenge(s)

How do we find a vertex in a sink $SCC$ of $G^{SCC}$?

Can we obtain an implicit topological sort of $G^{SCC}$ without computing $G^{SCC}$?
How do we find a vertex in a sink $SCC$ of $G^{SCC}$?

Can we obtain an implicit topological sort of $G^{SCC}$ without computing $G^{SCC}$?

Answer: $DFS(G)$ gives some information!
Maximum post numbering and the source of the meta-graph
Claim

Let $v$ be the vertex with maximum post numbering in $\text{DFS}(G)$. Then $v$ is in a $\text{SCC}$ $S$, such that $S$ is a source of $G^{\text{SCC}}$. 
Claim
Let \( v \) be the vertex with maximum post numbering in \( \text{DFS}(G^{rev}) \). Then \( v \) is in a \( \text{SCC} \) \( S \), such that \( S \) is a sink of \( G^{\text{SCC}} \).
Claim
Let \( v \) be the vertex with maximum post numbering in \( \text{DFS}(G^{rev}) \). Then \( v \) is in a SCC \( S \), such that \( S \) is a sink of \( G^{SCC} \).

Holds even after we delete the vertices of \( S \) (i.e., the vertex with the maximum post numbering, is in a sink of the meta graph).
The linear-time SCC algorithm itself
do \text{DFS}(G^{rev}) \text{ and output vertices in decreasing post order.}
Mark all nodes as unvisited
for each u in the computed order do
  if u is not visited then
    \text{DFS}(u)
    Let $S_u$ be the nodes reached by $u$
    Output $S_u$ as a strong connected component
    Remove $S_u$ from $G$

\textbf{Theorem}
\textit{Algorithm runs in time $O(m + n)$ and correctly outputs all the SCCs of $G$.}
Linear Time Algorithm: An Example - Initial steps

Graph $G$:

Reverse graph $G^{rev}$:

DFS of reverse graph:

Pre/Post DFS numbering of reverse graph:

1. B
2. E
3. A
4. C
5. F
6. D
7. G
8. H
Linear Time Algorithm: An Example

Original graph $G$ with rev post numbers:

$$
\begin{array}{c}
G \\
F \\
E \\
B \\
C \\
D \\
H \\
A \\
\end{array}
\Rightarrow
\begin{array}{c}
F \\
E \\
B \\
C \\
D \\
H \\
A \\
\end{array}
$$

Do DFS from vertex $G$ remove it.

$\text{SCC computed: } \{G\}$
**Linear Time Algorithm: An Example**

Do **DFS** from vertex G, remove it.

SCC computed: 
\{G\}

Do **DFS** from vertex H, remove it.

SCC computed: 
\{G\}, \{H\}
Linear Time Algorithm: An Example

Do **DFS** from vertex $H$, remove it.

SCC computed:
\{G\}, \{H\}

Do **DFS** from vertex $B$
Remove visited vertices: \{F, B, E\}.

SCC computed:
\{G\}, \{H\}, \{F, B, E\}
Linear Time Algorithm: An Example

Do **DFS** from vertex *F*
Remove visited vertices: 
\{*F*, *B*, *E*\}.

SCC computed:
\{*G*\}, \{*H*\}, \{*F*, *B*, *E*\}

---

Do **DFS** from vertex *A*
Remove visited vertices: 
\{*A*, *C*, *D*\}.

SCC computed:
\{*G*\}, \{*H*\}, \{*F*, *B*, *E*\}, \{*A*, *C*, *D*\}
SCC computed:
\{G\}, \{H\}, \{F, B, E\}, \{A, C, D\}
Which is the correct answer!
Exercise:
Given all the strong connected components of a directed graph $G = (V, E)$ show that the meta-graph $G^{\text{SCC}}$ can be obtained in $O(m + n)$ time.
A template for a class of problems on directed graphs:

- Is the problem solvable when $G$ is strongly connected?
- Is the problem solvable when $G$ is a DAG?
- If the above two are feasible then is the problem solvable in a general directed graph $G$ by considering the meta graph $G^{SCC}$?
Summary
Take away Points

- **DAGs**
- Topological orderings.
- **DFS**: pre/post numbering.
- Given a directed graph $G$, its **SCCs** and the associated acyclic meta-graph $G^{SCC}$ give a structural decomposition of $G$ that should be kept in mind.
- There is a **DFS** based linear time algorithm to compute all the **SCCs** and the meta-graph. Properties of **DFS** crucial for the algorithm.
- **DAGs** arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).