DAGs, DFS, topological sorting, linear time algorithm for SCC

Lecture 16
Thursday, October 22, 2020
16.1 Overview: Depth First Search and SCC
Overview

Topics:
- Structure of directed graphs
- **DAGs**: Directed acyclic graphs.
- Topological ordering.
- **DFS** pre/post number, and its properties.
- Linear time algorithm for **SCCs**.
THE END

... 

(for now)
16.2
Directed Acyclic Graphs
16.2.1

DAGs definition and basic properties
Directed Acyclic Graphs

Definition
A directed graph $G$ is a **directed acyclic graph** (DAG) if there is no directed cycle in $G$. 

![Directed Acyclic Graph Diagram]
Is this a DAG?
Sources and Sinks

Definition

1. A vertex $u$ is a **source** if it has no in-coming edges.
2. A vertex $u$ is a **sink** if it has no out-going edges.
Simple DAG Properties

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.
Simple DAG Properties

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.
DAG properties

1. $G$ is a DAG if and only if $G^{\text{rev}}$ is a DAG.
2. $G$ is a DAG if and only if each node is in its own strong connected component.

Formal proofs: exercise.
THE END
...
(for now)
16.2.2 Topological ordering
Total recall: Order on a set

Order or strict total order on a set $X$ is a binary relation $\prec$ on $X$, such that

1. **Transitivity:** $\forall x, y, z \in X \quad x \prec y$ and $y \prec z \implies x \prec z$.

2. For any $x, y \in X$, exactly one of the following holds:
   - $x \prec y$,
   - $y \prec x$ or
   - $x = y$.

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec x_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set $X$: listing the elements of $X$ from smallest to largest.
Total recall: Order on a set

**Order** or **strict total order** on a set $X$ is a binary relation $\prec$ on $X$, such that

1. **Transitivity:** $\forall x, y, z \in X \quad x \prec y$ and $y \prec z \implies x \prec z$.

2. For any $x, y \in X$, exactly one of the following holds:
   
   - $x \prec y$, $y \prec x$ or $x = y$.

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec x_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set $X$: listing the elements of $X$ from smallest to largest.
Total recall: Order on a set

Order or strict total order on a set $X$ is a binary relation $\prec$ on $X$, such that:

1. **Transitivity:** $\forall x, y, z \in X \quad x \prec y$ and $y \prec z \implies x \prec z$.

2. For any $x, y \in X$, exactly one of the following holds: $x \prec y$, $y \prec x$ or $x = y$.

Cannot have $x_1, \ldots, x_m \in X$, such that $x_1 \prec x_2, \ldots, x_{m-1} \prec x_m, x_m \prec x_1$, because...

Order on a (finite) set $X$: listing the elements of $X$ from smallest to largest.
Convention about writing edges

1. Undirected graph edges:
   \[ uv = \{u, v\} = vu \in E \]

2. Directed graph edges:
   \[ u \rightarrow v \equiv (u, v) \equiv (u \rightarrow v) \]
**Topological Ordering/Sorting**

A [topological ordering/topological sorting](#) of \( G = (V, E) \) is an ordering \( \prec \) on \( V \) such that if \((u \rightarrow v) \in E\) then \( u \prec v \).

**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.
DAGs and Topological Sort

Lemma

A directed graph $G$ can be topologically ordered $\iff$ $G$ is a DAG.

Need to show both directions.
DAGs and Topological Sort

Lemma

A directed graph $G$ is a DAG $\implies G$ can be topologically ordered.

Proof.

Consider the following algorithm:

1. Pick a source $u$, output it.
2. Remove $u$ and all edges out of $u$.
3. Repeat until graph is empty.

Exercise: prove this gives topological sort.
Topological ordering in linear time

Exercise: show algorithm can be implemented in $O(m + n)$ time.
Topological Sort: Example

```
 a b c
d e
f g
h
```
DAGs and Topological Sort

**Lemma**

A directed graph $G$ can be topologically ordered $\Rightarrow G$ 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 $u_1 \prec u_2 \prec \cdots \prec u_k \prec u_1$

$\Rightarrow u_1 \prec u_1.$

A contradiction (to $\prec$ being an order). Not possible to topologically order the vertices.
# 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

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

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.
Regular sorting and DAGs
DAGs and Topological Sort

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

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

3. **Exercise:** What is a DAG with the least number of distinct topological sorts for a given number $n$ of vertices?
THE END

...

(for now)
16.2.2.1

Explicit definition of what topological ordering
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...
THE END

... 

(for now)
16.3
Depth First Search (DFS)
16.3.1 Depth First Search (DFS) in Undirected Graphs
Depth First Search

1. **DFS** special case of Basic Search.
2. **DFS** is useful in understanding graph structure.
3. **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
4. ...many other applications as well.
DFS in Undirected Graphs

Recursive version. Easier to understand some properties.

\[
\text{DFS}(G) \\
\text{for all } u \in V(G) \text{ do} \\
\quad \text{Mark } u \text{ as unvisited} \\
\quad \text{Set } \text{pred}(u) \text{ to null} \\
\quad T \text{ is set to } \emptyset \\
\quad \text{while } \exists \text{ unvisited } u \text{ do} \\
\quad \quad \text{DFS}(u) \\
\quad \text{Output } T
\]

\[
\text{DFS}(u) \\
\quad \text{Mark } u \text{ as visited} \\
\quad \text{for each } uv \text{ in } \text{Out}(u) \text{ do} \\
\quad \quad \text{if } v \text{ is not visited then} \\
\quad \quad \quad \text{add edge } uv \text{ to } T \\
\quad \quad \quad \text{set } \text{pred}(v) \text{ to } u \\
\quad \quad \text{DFS}(v)
\]

Implemented using a global array \textit{Visited} for all recursive calls. 
\( T \) is the search tree/forest.
Edges classified into two types: $uv \in E$ is a

1. tree edge: belongs to $T$
2. non-tree edge: does not belong to $T$
Properties of DFS tree

Proposition

1. $T$ is a forest
2. Connected components of $T$ are same as those of $G$.
3. If $uv \in E$ is a non-tree edge then, in $T$, either:
   1. $u$ is an ancestor of $v$, or
   2. $v$ is an ancestor of $u$.

Question: Why are there no cross-edges?
Exercise

Prove that DFS of a graph $G$ with $n$ vertices and $m$ edges takes $O(n + m)$ time.
THE END

...

(for now)
16.3.2 DFS with pre-post numbering
**DFS with Visit Times**

Keep track of when nodes are visited.

- **DFS(G)**
  - 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)**
  - 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

\[ \text{time} = 0 \]

\begin{tabular}{l|l}
vertex & [pre, post] \\
\hline
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{tabular}
Animation

\[ time = 1 \]

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

\[
\begin{align*}
\text{vertex} & \\
1 & \quad [\text{pre}, \text{post}] \\
& \quad [1, ]
\end{align*}
\]
Animation

\[ \text{time} = 1 \]

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

![Graph Diagram]

1. Edge 1-2
2. Edge 2-3
3. Edge 3-4
4. Edge 3-5
5. Edge 5-6
6. Edge 4-7
7. Edge 7-8
8. Edge 8-9
9. Edge 9-10
10. Edge 10-7
Animation

\textbf{time} = 2

<table>
<thead>
<tr>
<th>vertex</th>
<th>\textit{pre, post}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,]</td>
</tr>
<tr>
<td>2</td>
<td>[2,]</td>
</tr>
</tbody>
</table>

![Graph diagram with vertices numbered 1 to 10 and edges connecting them.]
Animation

\[ \text{time} = 2 \]

<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>
</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>

Diagram: A graph with vertices numbered 1 to 10 and edges connecting them in a specific structure.
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>

\[ \text{1} \]
\[ \text{2} \]
\[ \text{3} \]
\[ \text{4} \]
\[ \text{5} \]
\[ \text{6} \]
\[ \text{7} \]
\[ \text{8} \]
\[ \text{9} \]
\[ \text{10} \]
Animation

$\text{time} = 5$

<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

\textit{time} = 6

<table>
<thead>
<tr>
<th>vertex</th>
<th>\textit{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>
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>
Animation

*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, 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>
# Animation

**time** = 10

| vertex |  
|--------|--
| 1      | [1, ]  
| 2      | [2, ]  
| 4      | [3, ]  
| 5      | [4, ]  
| 6      | [5, 6]  
| 3      | [7, ]  
| 7      | [8, ]  
| 8      | [9, 10]  

![Graph Diagram](image-url)
Animation

\[ \text{time} = 11 \]

<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>
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

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

\[ \text{time} = 14 \]

<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, 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>

![Graph Diagram]

---

1
2
3
4
5
6
7
8
9
10
Animation

\[ \text{time} = 15 \]

<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, 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

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>
**Animation**

\[ time = 17 \]

<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>

![Graph diagram with vertices numbered and ranges for pre and post times]
Animation

\[ \text{time} = 18 \]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([\text{pre}, \text{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**

\[
\text{time} = 19
\]

<table>
<thead>
<tr>
<th>vertex</th>
<th>([\text{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>7</td>
<td>[7, 12]</td>
</tr>
<tr>
<td>8</td>
<td>[8, 11]</td>
</tr>
<tr>
<td>9</td>
<td>[9, 10]</td>
</tr>
<tr>
<td>10</td>
<td>[18, 19]</td>
</tr>
</tbody>
</table>
Animation

\textit{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>
<tr>
<td>vertex</td>
<td>[pre, post]</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
</tr>
<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>
**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.

**Proof.**

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

**pre** and **post** numbers useful in several applications of **DFS**
**pre and post numbers**

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.

**Proof.**

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

**pre and post numbers useful in several applications of DFS**
**pre** and **post** numbers

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.

Proof.

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

**pre** and **post** numbers useful in several applications of **DFS**
pre and post numbers

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.

**Proof.**

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

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

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.

Proof.

- Assume without loss of generality that \( \text{pre}(u) < \text{pre}(v) \). Then \( v \) visited after \( u \).
- If \( \text{DFS}(v) \) invoked before \( \text{DFS}(u) \) finished, \( \text{post}(v) < \text{post}(u) \).
- If \( \text{DFS}(v) \) invoked after \( \text{DFS}(u) \) finished, \( \text{pre}(v) > \text{post}(u) \).

pre and post numbers useful in several applications of DFS
**pre** and **post** numbers

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.

**Proof.**

- Assume without loss of generality that $\text{pre}(u) < \text{pre}(v)$. Then $v$ visited after $u$.
- If $\text{DFS}(v)$ invoked before $\text{DFS}(u)$ finished, $\text{post}(v) < \text{post}(u)$.
- If $\text{DFS}(v)$ invoked after $\text{DFS}(u)$ finished, $\text{pre}(v) > \text{post}(u)$.

**pre** and **post** numbers useful in several applications of **DFS**
THE END

... (for now)
16.4

DFS in Directed Graphs

DFS
16.4.1

DFS in Directed Graphs: Pre/Post numbering
DFS in Directed Graphs

**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 $T$

**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:

1. **DFS**$(G)$ takes $O(m + n)$ time.

2. Edges added form a branching: a forest of out-trees. Output of **DFS**$(G)$ depends on the order in which vertices are considered.

3. 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 $v \in rch(u)$.

4. 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.

**Note:** Not obvious whether **DFS**$(G)$ is useful in directed graphs but it is.
DFS Properties

Generalizing ideas from undirected graphs:

1. \( \text{DFS}(G) \) takes \( O(m + n) \) time.

2. Edges added form a branching: a forest of out-trees. Output of \( \text{DFS}(G) \) depends on the order in which vertices are considered.

3. 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 \( v \in \text{rch}(u) \)

4. 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.

Note: Not obvious whether \( \text{DFS}(G) \) is useful in directed graphs but it is.
DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS(G)** takes $O(m + n)$ time.

2. Edges added form a branching: a forest of out-trees. Output of **DFS(G)** depends on the order in which vertices are considered.

3. 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 $v \in rch(u)$

4. 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.

Note: Not obvious whether **DFS(G)** is useful in directed graphs but it is.
DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS**\(_G\) takes \(O(m + n)\) time.

2. Edges added form a branching: a forest of out-trees. Output of **DFS**\(_G\) depends on the order in which vertices are considered.

3. 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 \(v \in \text{rch}(u)\)

4. 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.

Note: Not obvious whether **DFS**\(_G\) is useful in directed graphs but it is.
**DFS Properties**

Generalizing ideas from undirected graphs:

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

2. Edges added form a branching: a forest of out-trees. Output of **DFS** \((G)\) depends on the order in which vertices are considered.

3. 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 \(v \in rch(u)\)

4. 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.

*Note:* Not obvious whether **DFS** \((G)\) is useful in directed graphs but it is.
DFS tree and related edges

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

1. **Tree edges** that belong to $T$

2. A **forward edge** is a non-tree edges $(x, y)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

3. A **backward edge** is a non-tree edge $(y, x)$ such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

4. A **cross edge** is a non-tree edges $(x, y)$ such that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.
Types of Edges

[2, 11]

[3, 10]

[4, 5]

[6, 7]

[1, 16]

[12, 15]

[13, 14]

[8, 9]
THE END

...(for now)
16.4.2
DFS and cycle detection:
Topological sorting using DFS
Cycles in graphs

**Question:** 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?
Cycles in graphs

**Question:** 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...

And detect a cycle in the process:

**DFS** based algorithm:

1. Compute \( \text{DFS}(G) \)
2. 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 \( (v, u) \).
3. Otherwise output nodes in decreasing post-visit order. **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.
Correctness is not so obvious. See next two propositions.
Topological sort a graph using DFS...

And detect a cycle in the process

**DFS** based algorithm:

1. Compute $\text{DFS}(G)$
2. 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 $(v, u)$.
3. Otherwise output nodes in decreasing post-visit order. **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.

Correctness is not so obvious. See next two propositions.
Topological sort a graph using DFS...

And detect a cycle in the process

**DFS** based algorithm:

1. Compute $\text{DFS}(G)$
2. 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 $(v, u)$.
3. Otherwise output nodes in decreasing post-visit order. **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.

Correctness is not so obvious. See next two propositions.
Back edge and Cycles

**Proposition**

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

**Proof.**

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

Only if: Suppose there is a cycle \(C = v_1 \to v_2 \to \ldots \to v_k \to v_1\).

Let \(v_i\) be first node in \(C\) visited in 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.

\[ \Box \]
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 $[\text{pre}(v), \text{post}(v)]$.
- Case 2: $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$. 
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 $[\text{pre}(v), \text{post}(v)]$.
- **Case 2:** $[\text{pre}(u), \text{post}(u)]$ is disjoint from $[\text{pre}(v), \text{post}(v)]$.
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 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 $G$.

$\implies$ 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 $\pi : V \rightarrow \{1, 2, \ldots, n\}$, such that

$$(u \rightarrow v) \in E(G) \implies \pi(u) < \pi(v).$$
Example
THE END

... (for now)
16.5
The meta graph of strong connected components
Strong Connected Components (SCCs)

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

\[ G: \]

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 \( G^{SCC} \)

1. Vertices are \( S_1, S_2, \ldots, S_k \)
2. 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 \( G \).
Reversal and SCCs

Proposition

For any graph $G$, the graph of SCCs of $G^{\text{rev}}$ is the same as the reversal of $G^{\text{SCC}}$.

Proof.

Exercise.
The meta graph of SCCs is a DAG...

**Proposition**

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

**Proof.**

If $G^{\text{SCC}}$ 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 $G$. Formal details: exercise.
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 $G^{\text{SCC}}$ of $G$ can be computed in $O(m + n)$ time. $G^{\text{SCC}}$ 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.
THE END

... (for now)
16.6
Linear time algorithm for finding all strong connected components of a directed graph
16.6.1
Wishful thinking linear-time SCC algorithm
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:** \( O(n(n + m)) \)

Is there an \( O(n + m) \) time algorithm?
Finding all SCCs of a Directed Graph

Problem

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

Straightforward algorithm:

1. Mark all vertices in $V$ as not visited.
2. 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 $DFS(G^{rev}, u)$
   - $SCC(G, u) \iff rch(G, u) \cap rch(G^{rev}, u)$
   - $\forall u \in SCC(G, u)$: Mark $u$ as visited.

Running time: $O(n(n + m))$

Is there an $O(n + m)$ time algorithm?
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 $\text{SCC}(G, u)$ the strong component of $u$:

Compute $\text{rch}(G, u)$ using $\text{DFS}(G, u)$

Compute $\text{rch}(G^{rev}, u)$ using $\text{DFS}(G^{rev}, u)$

$\text{SCC}(G, u) \leftarrow \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$

$\forall u \in \text{SCC}(G, u)$: Mark $u$ as visited.

**Running time:** $O(n(n + m))$

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

Graph $G$

Graph of $\text{SCC}s$ $G^{\text{SCC}}$

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

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

Exploit structure of meta-graph...

### Wishful Thinking Algorithm

1. Let \( u \) be a vertex in a sink SCC of \( G^{SCC} \)
2. Do \( DFS(u) \) to compute \( SCC(u) \)
3. Remove \( SCC(u) \) and repeat

### Justification

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

Exploit structure of meta-graph...

### Wishful Thinking Algorithm

1. Let \( u \) be a vertex in a sink SCC of \( G^{SCC} \)
2. Do \( DFS(u) \) to compute \( SCC(u) \)
3. Remove \( SCC(u) \) and repeat

### Justification

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

Exploit structure of meta-graph...

### Wishful Thinking Algorithm

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

### Justification

1. $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
2. ... since there are no edges coming out a sink!
Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

### Wishful Thinking Algorithm

1. Let $u$ be a vertex in a sink SCC of $G_{SCC}$
2. Do $\text{DFS}(u)$ to compute $\text{SCC}(u)$
3. Remove $\text{SCC}(u)$ and repeat

### Justification

1. $\text{DFS}(u)$ only visits vertices (and edges) in $\text{SCC}(u)$
2. ... since there are no edges coming out a sink!
3. $\text{DFS}(u)$ takes time proportional to size of $\text{SCC}(u)$
Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let \( u \) be a vertex in a sink SCC of \( G^{SCC} \)
2. Do \( DFS(u) \) to compute \( SCC(u) \)
3. Remove \( SCC(u) \) and repeat

Justification

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

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

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

Answer: $\text{DFS}(G)$ gives some information!
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 $G^{SCC}$?

Answer: $DFS(G)$ gives some information!
Big Challenge(s)

How do we find a vertex in a sink \( \text{SCC} \) of \( G^{\text{SCC}} \)?

Can we obtain an implicit topological sort of \( G^{\text{SCC}} \) without computing \( G^{\text{SCC}} \)?

Answer: \( \text{DFS}(G) \) gives some information!
THE END

...  

(for now)
16.6.2
Maximum post numbering and the source of the meta-graph
Post numbering and the meta graph

Claim

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

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

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).
Reverse post numbering and the meta graph

Claim

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

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 END

...

(for now)
16.6.3
The linear-time SCC algorithm itself
Linear Time Algorithm

...for computing the strong connected components in $G$

```
do DFS($G^{rev}$) 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
    DFS($u$)
    Let $S_u$ be the nodes reached by $u$
    Output $S_u$ as a strong connected component
    Remove $S_u$ from $G$
```

**Theorem**

*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:
Reverse graph $G_{rev}$:

DFS of reverse graph:

Pre/Post DFS numbering of reverse graph:
Linear Time Algorithm: An Example

Removing connected components: 1

Original graph \( G \) with rev post numbers:

Do **DFS** from vertex \( G \) remove it.

SCC computed: \( \{ G \} \)
Linear Time Algorithm: An Example

Removing connected components: 2

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

Removing connected components: 3

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

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

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

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

Removing connected components: 4

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\}
Linear Time Algorithm: An Example

Final result

SCC computed:
{G}, {H}, {F, B, E}, {A, C, D}
Which is the correct answer!
Obtaining the meta-graph...

Once the strong connected components are computed.

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.
Solving Problems on Directed Graphs

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 $G^{SCC}$?
THE END
...
(for now)
16.7
An Application of directed graphs to make
Make/Makefile

A  I know what make/makefile is.
B  I do NOT know what make/makefile is.
make Utility [Feldman]

1. Unix utility for automatically building large software applications
2. A makefile specifies
   1. Object files to be created,
   2. Source/object files to be used in creation, and
   3. How to create them
An Example makefile

project: main.o utils.o command.o
        cc -o project main.o utils.o command.o

main.o: main.c defs.h
        cc -c main.c
utils.o: utils.c defs.h command.h
        cc -c utils.c
command.o: command.c defs.h command.h
        cc -c command.c
makefile as a Digraph

main.c  main.o
|       |
utils.c  |
|       |
defs.h  utils.o  project
|       |
command.h
|       |
command.c  command.o
Computational Problems for make

1. Is the makefile reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.
Algorithms for make

1. Is the makefile reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
   1. Find all vertices reachable (using DFS/BFS) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.
THE END
...
(for now)
16.8

Summary
Take away Points

1. **DAGs**
2. Topological orderings.
3. **DFS**: pre/post numbering.
4. Given a directed graph $G$, its **SCC**s and the associated acyclic meta-graph $G^{SCC}$ give a structural decomposition of $G$ that should be kept in mind.
5. There is a **DFS** based linear time algorithm to compute all the **SCC**s and the meta-graph. Properties of **DFS** crucial for the algorithm.
6. **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).