Last time we looked at the BasicSearch algorithm:

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

\[ \text{ExploreStep}(G, u, \text{Visited}, S) \]
- Output \( S \)

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

We said that if \( \text{ToExplore} \) was a:
- Stack, the algorithm is equivalent to \text{DFS}
- Queue, the algorithm is equivalent to \text{BFS}

What if the algorithm was written recursively (instead of the while loop, you recursively call \text{explore}). What would the algorithm be equivalent to?
Pre-lecture brain teaser

Last time we looked at the BasicSearch algorithm:

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

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

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?
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 $G$. 

![Directed Acyclic Graph Diagram]


3
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 $≺$ on $X$, such that

- **Transitivity:** $\forall x, y, z \in X \quad x ≺ y$ and $y ≺ z \implies x ≺ z$.
- For any $x, y \in X$, exactly one of the following holds: $x ≺ y$, $y ≺ x$ or $x = y$. 


Convention about writing edges

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

- Directed graph edges:
  \[ u \rightarrow v \equiv (u, v) \equiv (u \rightarrow v) \]
**Definition**

A topological ordering/topological sorting of $G = (V, E)$ is an ordering $≺$ on $V$ such that if $(u \rightarrow v) \in E$ 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. Calculate the in-degree of each vertex
2. For each vertex that is source ($deg_{in}(v) = 0$):
   2.1 Add $v$ to the topological sort
   2.2 Lower the in-degree of vertices $v$ is connected to. 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>In-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>In-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

![Diagram showing multiple possible topological orderings of a graph with nodes labeled a, b, c, d, e, f, g, and h.]
**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**($G$):

```plaintext
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. \qed
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 $n = |V|$)
Assuming:

\[ V = \{a, \ldots, w\}\]

\[ \pi = \{1, \ldots, 23\} \]
Assuming:

\[ V = \{ a, \ldots, w \} \]
\[ \pi = \{ 1, \ldots, 23 \} \]
Depth First Search (DFS)
Depth First Search (DFS) in Undirected Graphs
Depth First Search

- **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.

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

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

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

- **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
DFS with Visit Times

Keep track of when nodes are visited.

\[
\text{DFS}(G) \\
\text{for all } u \in V(G) \text{ do} \\
\quad \text{Mark } u \text{ as unvisited} \\
\quad T \text{ is set to } \emptyset \\
\quad \text{time} = 0 \\
\text{while } \exists \text{ unvisited } u \text{ do} \\
\quad \text{DFS}(u) \\
\text{Output } T
\]

\[
\text{DFS}(u) \\
\quad \text{Mark } u \text{ as visited} \\
\quad \text{pre}(u) = ++time \\
\quad \text{for each } uv \text{ in } \text{Out}(u) \text{ do} \\
\quad \quad \text{if } v \text{ is not marked then} \\
\quad \quad \quad \text{add edge } uv \text{ to } T \\
\quad \quad \text{DFS}(v) \\
\quad \text{post}(u) = ++time
\]
Animation

\[ \text{time} = 0 \]

<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>
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

\[ \text{time} = 2 \]

<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>
</tbody>
</table>

\[
\begin{array}{c|c}
\text{vertex} & \text{[pre, post]} \\
1 & [1, ] \\
2 & [2, ] \\
\end{array}
\]
$$time = 3$$

<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>
</tbody>
</table>

Diagram showing a graph with vertices and edges.
Animation

\(time = 4\)

<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>

\[\begin{array}{c|c}
\hline
\text{vertex} & \text{[pre, post]} \\
\hline
1 & [1, ] \\
2 & [2, ] \\
4 & [3, ] \\
5 & [4, ] \\
\hline
\end{array}\]
### 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

**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 of a graph with vertices and time annotations]
**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

**time** = 9

<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, ]</td>
</tr>
</tbody>
</table>
Animation

\[ \text{time} = 10 \]

<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, 10]</td>
</tr>
</tbody>
</table>

![Graph representation of the order of vertices](image)
**Animation**

\[ \text{time} = 11 \]

<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, 11]</td>
</tr>
<tr>
<td>8</td>
<td>[9, 10]</td>
</tr>
</tbody>
</table>
**Animation**

\[time = 12\]

<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>

Diagram: A directed graph with vertices 1 to 10 and edges connecting them in a specific order.
\( \text{time} = 13 \)

<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>
**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>
### 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>([\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>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>
**Animation**

**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>

![Graph Diagram](image)
$time = 20$

<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

DFS(G)
- Mark all nodes $u$ as unvisited
- $T$ is set to $\emptyset$
- $time = 0$
- while there is an unvisited node $u$ do
  - DFS($u$)
- Output $T$

DFS($u$)
- Mark $u$ as visited
- pre($u$) = ++$time$
- for each edge $(u, v)$ in $Out(u)$ do
  - if $v$ is not visited
    - add edge $(u, v)$ to $T$
    - DFS($v$)
- post($u$) = ++$time$
Example of **DFS** in directed graph
Example of DFS in directed graph

[6, 7] F ———>[13, 14] D

A ——— B
C ——— D
E ——— F
G ——— H

DFS colors:
- [2, 11] B: Blue
- [1, 16] A: Red
- [12, 15] C: Blue
- [3, 10] E: Red
- [6, 7] F: Red
- [4, 5] G: Red
- [8, 9] H: Blue
- [13, 14] D: Blue
Generalizing ideas from undirected graphs:

- \textbf{DFS}(G) takes } O(m + n) \text{ time.
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:

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

- **Back edges:** (F,B), (D,A)
- **Forward edges:** (B,H)
- **Cross edges:** (F,G), (H,G), (D,H)
• **Back edges**: (F,B), (D,A)
• **Forward edges**: (B,H)
• **Cross edges**: (F,G), (H,G), (D,H)
DFS and cycle detection: Topological sorting using DFS
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 **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, $\text{DFS}(G)$ can output nodes in this order.

Computes topological ordering of the vertices.

Algorithm runs in $O(n + m)$ time.
Topological sort a graph using **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.

Computes topological ordering of the vertices.

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

See next two propositions.
Example

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 \( C = v_1 \to v_2 \to \ldots \to v_k \to v_1 \).

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 $G$. 

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 \([\text{pre}(v), \text{post}(v)]\).
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 $u$. 

\[\square\]
We just proved:

**Proposition**
*If G is a DAG and post(v) > post(u), then (u → v) is not in G.*

⇒ sort the vertices of a DAG by decreasing post numbering in decreasing order, then this numbering is valid.
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.
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 $G^{SCC}$

- 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.
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 $G$. \qed
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.

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:

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: $O(n(n + m))$
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.

\[
\text{for each vertex } u \in V \text{ not visited yet do}
\]

find \( SCC(G, u) \) the strong component of \( u \):

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

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

\( \forall u \in SCC(G, u) \): 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.
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
Linear-time Algorithm for SCCs: Ideas

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 $\text{SCC}(u)$
Linear-time Algorithm for SCCs: Ideas

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!
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

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 $\text{SCC}(u)$
Linear-time Algorithm for SCCs: Ideas

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 $G^{SCC}$?
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 $\text{DFS}(G^{\text{rev}})$. 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 \( \text{DFS}(G^{\text{rev}}) \).

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 linear-time SCC algorithm itself
Linear Time Algorithm

**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 $G$.*
Graph $G$:  

Reverse graph $G^{rev}$:  

DFS of reverse graph:  

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

Original graph $G$ with rev post numbers:

$$
\begin{array}{c}
G \\
F \\
E \\
B \\
C \\
D \\
H \\
A \\
16 \\
11 \\
6 \\
12 \\
10 \\
15 \\
5 \\
4
\end{array}
$$

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

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 it.

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 $G^{SCC}$ 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
• **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).
Scratch Figures

The diagram shows a network of nodes labeled with letters: a, b, c, d, e, f, g, h. The network has directed edges connecting the nodes in various ways, indicating a flow or relationship between the elements.