# DAGs and Graph Search

<table>
<thead>
<tr>
<th>Lecture A</th>
<th>Tiefenbruck</th>
<th>Tu, Th 11am-12:20pm</th>
<th>Center 119</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture B</td>
<td>Tiefenbruck</td>
<td>Tu, Th 9:30am-10:50am</td>
<td>Center 119</td>
</tr>
<tr>
<td>Lecture C</td>
<td>Minnes</td>
<td>Tu, Th 3:30pm-4:50pm</td>
<td>WLH 2005</td>
</tr>
</tbody>
</table>

[http://cseweb.ucsd.edu/classes/fa15/cse21-abc/](http://cseweb.ucsd.edu/classes/fa15/cse21-abc/)

Oct 27, 2015
1. Definition of DAG.
2. Ordering algorithm on a DAG.
3. Graph search and reachability.

In the textbook: Sections 10.4 and 10.5
(some) Prerequisites for (some) CSE classes

2-year plan:

Take classes in some order.

If course A is prerequisite for course B, must take course A before we take course B.
Prerequisites for CSE classes

Which of the following orderings are ok?

A. 30, 145, 151, 100.
B. 110, 105, 21, 101.
C. 21, 105, 130.
D. More than one of the above.
E. None of the above.
Prerequisites for CSE classes

What if we want to include all vertices (i.e. courses)?

Is this possible for any graph?
What if we want to include all vertices (i.e. courses)?

Is this possible for any graph?

1. Classify graphs for which it is.
2. For those, find a good ordering.
Barriers to ordering

Which of the following graphs have good orderings?

A. 

B. 

C. 

D. 

E. None of the above.
A can't be first (because B is before it).
B can't be first (because C is before it).
C can't be first (because D is before it).
D can't be first (because A is before it).

Whenever there is a cycle, can't find a "good" ordering.
Directed graphs with no cycles are called \textit{directed acyclic graphs} (DAGs).
Directed graphs with no cycles are called **directed acyclic graphs** (DAGs).

A **topological ordering** of a graph is an (ordered) list of all its vertices such that, for each edge \((v,w)\) in the graph, \(v\) comes before \(w\) in the list.
Two algorithmic questions:

1. Given an (ordered) list of all vertices in the graph, is it a topological ordering?
2. Given a graph, produce a topological ordering.
1. Given an (ordered) list of all vertices in the graph, is it a topological ordering?

*How would you do it?*
2. Given a graph, produce a topological ordering.

At what vertex should we start?

A. Any vertex is okay.
B. We must start at A.
C. Choose any vertex with at least one outgoing edge.
D. Choose any vertex with no incoming edges.
E. None of the above.
In a DAG, vertices with no incoming edges are called **sources**.

Which of these vertices are sources?

A. Only A and G.
B. Only A.
C. Only I.
D. Only I and F.
E. None of the above.
Lemma 1: Every DAG has a (at least one) source

*How would you prove this?*
Sources of a DAG

Lemma 1: Every DAG has a (at least one) source

How would you prove this?

Not a source … look at incoming edges
Lemma 1: Every DAG has a (at least one) source

How would you prove this?

Not a source … look at incoming edges
Lemma 1: Every DAG has a (at least one) source

How would you prove this?

Found a source!
Lemma 1: Every DAG has a (at least one) source

Let $G$ be a DAG. We want to show that $G$ has a source vertex.

**In a proof by contradiction (aka indirect proof), what should we assume?**

A. $G$ has a source vertex.
B. All the vertices in $G$ are sources.
C. No vertex in $G$ is a source.
D. $G$ has at least one source vertex and at least one vertex that's not a source.
E. None of the above.
Lemma 1: Every DAG has a (at least one) source

Proof of Lemma 1: Let $G$ be a DAG with $n$ ($n>1$) vertices. We want to show that $G$ has a source vertex.

Assume towards a contradiction that no vertex in $G$ is a source.

Let $v_0$ be a vertex in $G$. Since $v$ is not a source (by assumption), it has an incoming edge. Let $v_1$ be a vertex in $G$ so that $(v_1, v_0)$ is an edge in $G$. Since $v_1$ is also not a source, let $v_2$ be a vertex in $G$ so that $(v_2, v_1)$ is an edge in $G$. Keep going to find vertices. There must be a repeated vertex in this list (Pigeonhole Principle). Contradiction with $G$ being acyclic.
Sources of a DAG

Notation: $G-v$ is the graph that results when remove $v$ and all of its incident edges from $G$.

Lemma 2: If $v$ is a source vertex of $G$, then $G$ is a DAG if and only if $G-v$ is a DAG.
Sources of a DAG

Notation: $G-v$ is the graph that results when remove $v$ and all of its incident edges from $G$.

Lemma 2: If $v$ is a source vertex of $G$, then $G$ is a DAG if and only if $G-v$ is a DAG.

Proof of Lemma 2: Let $G$ be a DAG and assume $v$ is a vertex in $G$.

Assume $G$ is a DAG. WTS $G-v$ is a DAG.

Assume $G-v$ is a DAG. WTS $G$ is a DAG.
Sources of a DAG

Notation: $G - v$ is the graph that results when remove $v$ and all of its incident edges from $G$.

Lemma 2: If $v$ is a source vertex of $G$, then $G$ is a DAG if and only if $G - v$ is a DAG.

Proof of Lemma 2: Let $G$ be a DAG and assume $v$ is a vertex in $G$.

Assume $G$ is a DAG. WTS $G - v$ is a DAG. Can't introduce any cycles by removing edges.

Assume $G - v$ is a DAG. WTS $G$ is a DAG.
Sources of a DAG

*Notation:* $G-v$ is the graph that results when remove $v$ and all of its incident edges from $G$.

Lemma 2: If $v$ is a **source vertex** of $G$, then $G$ is a DAG if and only if $G-v$ is a DAG

**Proof of Lemma 2:** Let $G$ be a DAG and assume $v$ is a vertex in $G$.

Assume $G$ is a DAG. **WTS** $G-v$ is a DAG. Can't introduce any cycles by removing edges.

Assume $G-v$ is a DAG. **WTS** $G$ is a DAG. ?? Contrapositive …
Sources of a DAG

Notation: $G-v$ is the graph that results when remove $v$ and all of its incident edges from $G$.

Lemma 2: If $v$ is a source vertex of $G$, then $G$ is a DAG if and only if $G-v$ is a DAG

Proof of Lemma 2: Let $G$ be a DAG and assume $v$ is a vertex in $G$.

Assume $G$ is a DAG. WTS $G-v$ is a DAG. Can't introduce any cycles by removing edges.

Assume $G$ is not a DAG. WTS $G-v$ is not a DAG. A cycle in $G$ can't include a source (because no incoming edges). So this cycle will also be in $G-v$. 
Find Topological Ordering (if possible)

While G has at least one vertex
  If G has some source,
    Choose one source and output it.
    Delete the source and all its outgoing edges from G.
  Else
    Return that G is not a DAG.
Find Topological Ordering (if possible)

While G has at least one vertex
  If G has some source,
    Choose one source and output it.
    Delete the source and all its outgoing edges from G.
  Else
    Return that G is not a DAG.

Implementation details:
Maintain integer array, InDegree[], of length n
Maintain collection of sources, S, as list, stack, or queue.

Choose first x in S.
For each y adjacent to x,
  Decrement InDegree[y]
If InDegree[y]=0, add y to S.
Example

InDegree[]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: \( S = A, G \)

Output:

Choose first \( x \) in \( S \).
For each \( y \) adjacent to \( x \),
  - Decrement \( \text{InDegree}[y] \)
  - If \( \text{InDegree}[y]=0 \), add \( y \) to \( S \).
Choose first $x$ in $S$.
For each $y$ adjacent to $x$,
Decrement $\text{InDegree}[y]$
If $\text{InDegree}[y]=0$, add $y$ to $S$. 

**Example**

**InDegree[]**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: $S = G, B, C$

**Output:** A
Example

**InDegree[]**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: **S = B, C, E**

**Output:** A, G

Choose first x in S.

For each y adjacent to x,
Decrement InDegree[y]
If InDegree[y]=0, add y to S.
**Example**

**InDegree[]**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: \( S = C, E \)

**Output:** A, G, B

Choose first \( x \) in \( S \).
For each \( y \) adjacent to \( x \),
Decrement \( \text{InDegree}[y] \)
If \( \text{InDegree}[y] = 0 \), add \( y \) to \( S \).
### Example

**InDegree[]**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: \( S = E, D, H \)

**Output:** A, G, B, C

Choose first \( x \) in \( S \).
For each \( y \) adjacent to \( x \),
  - Decrement \( \text{InDegree}[y] \)
  - If \( \text{InDegree}[y] = 0 \), add \( y \) to \( S \).
Example

InDegree[]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Collection of sources: $S = \text{D, H}$


Choose first $x$ in $S$.
For each $y$ adjacent to $x$,
Decrement $\text{InDegree}[y]$
If $\text{InDegree}[y] = 0$, add $y$ to $S$. 
Example

**InDegree[]**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Collection of sources: \( S = H, F \)

**Output:** A, G, B, C, E, D

Choose first \( x \) in \( S \).
For each \( y \) adjacent to \( x \),
  - Decrement \( \text{InDegree}[y] \)
  - If \( \text{InDegree}[y] = 0 \), add \( y \) to \( S \).
Example

\textbf{InDegree[]}

\begin{tabular}{cccccccc}
\hline
A & B & C & D & E & F & G & H & I \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}

Collection of sources: $S = F, I$

\textbf{Output}: A, G, B, C, E, D, H

Choose first $x$ in $S$.
For each $y$ adjacent to $x$,
Decrement InDegree[$y$]
If InDegree[$y$] = 0, add $y$ to $S$. 
Example

**InDegree[]**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Collection of sources: $S = I$

**Output:** A, G, B, C, E, D, H, F

Choose first $x$ in $S$.

For each $y$ adjacent to $x$,
- Decrement $\text{InDegree}[y]$
- If $\text{InDegree}[y]=0$, add $y$ to $S$. 
**Example**

InDegree[]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Collection of sources: \( S = \)

**Output:** A, G, B, C, E, D, H, F, I

Choose first \( x \) in \( S \).

For each \( y \) adjacent to \( x \),

- Decrement InDegree[\( y \)]
- If InDegree[\( y \)] = 0, add \( y \) to \( S \).
Prerequisites for CSE classes

2-year plan:

Take classes in some order.

If course A is prerequisite for course B, must take course A before we take course B.

How many quarters?
First layer: all the sources

Next layer: nodes that are sources (once we remove previous layer and its outgoing edges)

Repeat…
Prerequisites for some CSE classes

How many quarters (layers) before take all classes?

A. 1.
B. 2.
C. 3.
D. 4.
E. More than four.
Recall: Tartaglia's Pouring Problem

Large cup: contains 8 ounces, can hold more.
Medium cup: is empty, has 5 ounce capacity.
Small cup: is empty, has 3 ounce capacity

You can pour from one cup to another until the first is empty or the second is full.
Rephrasing the problem:

(1) Is there a path from (8,0,0) to (4,4,0)?
(2) If so, what's the best path?

"Best" means "shortest length"
Tartaglia's Pouring Problem

Rephrasing the problem: using configurations

(1) Is there a path from (8,0,0) to (4,4,0)?
(2) If so, what's the best path?

"Best" means "shortest length"

(l, m, s) means
l ounces in large cup
m ounces in medium cup
s ounces in small cup
Tartaglia's Pouring Problem

How many configurations are possible?

A. Infinitely many
B. $4 \times 6 \times 9 = 216$
C. 24
D. 16
E. None of the above.

(l, m, s)

means

l ounces in large cup
m ounces in medium cup
s ounces in small cup
How many configurations are possible?

Small cup: 0, 1, 2, or 3
Medium cup: 0, 1, 2, 3, 4, or 5
Large cup: 0, 1, 2, 3, 4, 5, 6, 7, or 8

(l, m, s) ** means

l ounces in large cup
m ounces in medium cup
s ounces in small cup

**integer values
Tartaglia's Pouring Problem

How many configurations are possible?

Small cup: 0, 1, 2, or 3
Medium cup: 0, 1, 2, 3, 4, or 5
Large cup: 0, 1, 2, 3, 4, 5, 6, 7, or 8

But can't have 3 in small AND 5 in medium AND 8 in large: **Total must be 8.**
Tartaglia's Pouring Problem

(l, m, s) ** means
l ounces in large cup
m ounces in medium cup
s ounces in small cup

**integer values

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

The three columns total 8 in each row. 24 rows.
You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?

- (8,0,0)
- (3,5,0) → Pour from medium cup → (3,2,3)
- (5,0,3) → Unto previous → (0,5,3)

Pour from large cup
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?
You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

Which configurations are actually possible?
Tartaglia's Pouring Problem

You can pour from one cup to another until the first is empty or the second is full: so either (or both) of the small cup and the medium cup must always be either full or empty.

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>l</th>
<th>m</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
Are all 16 configurations reachable from (8,0,0)?

A. Yes
B. All except for (8,0,0)
C. No
Given a directed graph $G$ and a start vertex $s$,

produce a list of all vertices $v$ reachable from $s$ by a directed path in $G$. 
Given a directed graph $G$ and a start vertex $s$, produce a list of all vertices $v$ reachable from $s$ by a directed path in $G$.

At each point in a graph search algorithm, the vertices are partitioned into

- **X**: eXplored
- **F**: frontier (reached but haven't yet explored)
- **U**: unreached
Graph reachability: HOW

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
    Move v from F to X.

Return X.
procedure GraphSearch (G: directed graph, s: vertex)

Initialize $X =$ empty, $F = \{s\}$, $U = V - F$.
While $F$ is not empty:
    Pick $v$ in $F$.
    For each neighbor $u$ of $v$:
        If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
    Move $v$ from $F$ to $X$.

Return $X$. 
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V − F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

After first iteration of while loop…

v = (8,0,0)
X = {(8,0,0)}  F = {(3,5,0), (5,0,3)}  U = green nodes
procedure GraphSearch \((G: \text{directed graph}, s: \text{vertex})\)

Initialize \(X = \text{empty}, F = \{s\}, U = V - F.\)
While \(F\) is not empty:
  Pick \(v\) in \(F.\)
  For each neighbor \(u\) of \(v:\)
    If \(u\) is not in \(X\) or \(F\), then move \(u\) from \(U\) to \(F.\)
  Move \(v\) from \(F\) to \(X.\)

Return \(X.\)

\(v = (8,0,0)\)
\(X = \{(8,0,0), (3,5,0)\}\)
\(F = \{(3,2,3), (0,5,3) (5,0,3)\}\)

**Different methods of choosing \(v\) give breadth-first search vs. depth-first search**

**After second iteration of while loop…**

Return \(X.\)
procedure GraphSearch \((G: \text{ directed graph, } s: \text{ vertex})\)

Initialize \(X = \text{ empty}, F = \{s\}, U = V - F\).
While \(F\) is not empty:
  Pick \(v\) in \(F\).
  For each neighbor \(u\) of \(v\):
    If \(u\) is not in \(X\) or \(F\), then move \(u\) from \(U\) to \(F\).
  Move \(v\) from \(F\) to \(X\).

Return \(X\).

*Does this algorithm output the collection of vertices \(v\) reachable from \(s\) by a directed path in \(G\)?*
Graph reachability: WHY

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize \(X=\) empty, \(F = \{s\}, U = V - F\).
While \(F\) is not empty:
  Pick \(v\) in \(F\).
  For each neighbor \(u\) of \(v\):
    If \(u\) is not in \(X\) or \(F\), then move \(u\) from \(U\) to \(F\).
  Move \(v\) from \(F\) to \(X\).

Return \(X\).

Does this algorithm output the collection of vertices \(v\) reachable from \(s\) by a directed path in \(G\)?

**Goal:**
1. Every element of output \(X\) is reachable from \(s\) in \(G\).
2. Every reachable vertex is in \(X\) (by end of algorithm).
Graph reachability: WHY

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F.
While F is not empty:
   Pick v in F.
   For each neighbor u of v:
      If u is not in X or F, then move u from U to F.
   Move v from F to X.

Return X.

Claim: After t\textsuperscript{th} iteration through while loop, every element of (current version of) X or F is reachable from s in G.

Proof by induction on t.
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = \{s\}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

Claim: After \( t^{th} \) iteration through while loop, every element of (current version of) X or F is reachable from s in G.

Base case (t=0):
Before any iterations of loop, X is initialized as empty and F is initialized as \{s\}. WTS s is reachable from s in G. 😊
**Graph reachability: WHY**

**procedure GraphSearch** (G: directed graph, s: vertex)

- Initialize X= empty, F = {s}, U = V – F.
- While F is not empty:
  - Pick v in F.
  - For each neighbor u of v:
    - If u is not in X or F, then move u from U to F.
  - Move v from F to X.
- Return X.

**Claim**: After $t^{th}$ iteration through while loop, every element of (current version of) X or F is reachable from s in G.

**Induction step**: Suppose after $t^{th}$ iteration, every element of X or F is reachable from s in G.

*What happens in $t+1^{st}$ iteration?*
**Graph reachability: WHY**

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize $X = \emptyset$, $F = \{s\}$, $U = V - F$.

While $F$ is not empty:

Pick $v$ in $F$.

For each neighbor $u$ of $v$:

- If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.

Move $v$ from $F$ to $X$.

Return $X$.

**Claim**: After $t$th iteration through while loop, every element of (current version of) $X$ or $F$ is reachable from $s$ in $G$.

**Induction step**: Suppose that after $t$th iteration, every element of $X$ or $F$ is reachable from $s$ in $G$.

*What happens in $t+1$st iteration?*
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

Claim: After $t^{th}$ iteration through while loop, every element of (current version of) X or F is reachable from s in G.

Using Claim to prove Goal 1: After the final iteration, output X, which by claim, only contains vertices that are reachable from s.
Graph reachability: WHY

procedure GraphSearch (G: directed graph, s: vertex)

Initialize $X = \text{empty}$, $F = \{s\}$, $U = V - F$.
While $F$ is not empty:
   Pick $v$ in $F$.
   For each neighbor $u$ of $v$:
      If $u$ is not in $X$ or $F$, then move $u$ from $U$ to $F$.
   Move $v$ from $F$ to $X$.

Return $X$.

Does this algorithm output the collection of vertices $v$ reachable from $s$ by a directed path in $G$?

Goal:
1. Every element of output $X$ is reachable from $s$ in $G$. 😊
2. Every reachable vertex is in $X$ (by end of algorithm).
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

**WTS Goal 2:** Every reachable vertex is in X.

*Hint: assume, towards a contradiction that some vertex is reachable from s but not in X. Look for first vertex on the path between s that is not in X.*
Graph reachability: WHEN

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F, then move u from U to F.
  Move v from F to X.

Return X.

How long does it take to pick v in F?
How long does it take to iterate over neighbors of v?

Need to know some implementation decisions.
Graph reachability: WHEN

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X= empty, F = \{s\}, U = V – F.
While F is not empty:
Pick v in F.
For each neighbor u of v:
  If u is not in X or F, then move u from U to F.
Move v from F to X.
Return X.

Assume G stored as adjacency list.
Assume have array Status[]
* length n array
* each entry either F, X, U

What's an upper bound on the time it takes to do one iteration of the body of the for loop?

A. O( n² )
B. O(n)
C. O( degree (v) )
D. O( |F| )
E. None of the above.
procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
    Pick v in F.
    For each neighbor u of v:
        If u is not in X or F, then move u from U to F.
    Move v from F to X.
Return X.

Assume G stored as adjacency list.
Assume have array Status[]
* length n array
* each entry either F, X, U

What's an upper bound on the time it takes to go through the whole for loop for a given v?

A. O( n^2 )
B. O(n)
C. O( degree (v) )
D. O( |F| )
E. None of the above.
Graph reachability: WHEN

\[\text{procedure GraphSearch} \ (G: \text{directed graph}, \ s: \text{vertex})\]

Initialize \(X\) = empty, \(F = \{s\}\), \(U = V - F\).
While \(F\) is not empty:
  Pick \(v\) in \(F\).
  For each neighbor \(u\) of \(v\):
    If \(u\) is not in \(X\) or \(F\), then move \(u\) from \(U\) to \(F\).
  Move \(v\) from \(F\) to \(X\).

Return \(X\).

Assume \(G\) stored as adjacency list.
Assume have array \(\text{Status}[]\)
* length \(n\) array
* each entry either \(F\), \(X\), \(U\)

What's an upper bound on the time spent on the for loop throughout the whole algorithm?

A. \(O( n )\)
B. \(O( |V| )\)
C. \(O( |E| )\)
D. \(O( |F| )\)
E. None of the above.
Graph reachability: WHEN

**procedure GraphSearch** (G: directed graph, s: vertex)

Initialize X = empty, F = \{s\}, U = V – F.
While F is not empty:
   Pick v in F.
   For each neighbor u of v:
      If u is not in X or F, then move u from U to F.
   Move v from F to X.

Return X.

Assume G stored as adjacency list.
Assume have array Status[]
   * length n array
   * each entry either F, X, U

Total time is asymptotically upper bounded by sum of degrees of all vertices

i.e. $O(2 |E|)$
i.e. $O( |E|)$
Reminders

HW 5 due Friday 11:59pm via Gradescope.