• HW5 is due on Thursday, February 16, 11:59pm

• MIDTERM REVIEW SESSION POLL – CLOSES TONIGHT 11:59PM

• Today: Days 14+15 of posted slides – graph search, trees
  • Topological ordering from Day 13
  • Reachability
  • ...

• Any logistic, other issues?
Sources of a DAG

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

How would you prove this?

(Pigeonhole Principle, given that $G$ is a DAG.)

(We did this three slides ago...)
Topological ordering

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

How would you do this?

“Source” vertex has no incoming edges

(pigeonhole argument that a DAG must have at least one source vertex)
Sources of a DAG

*Notation:* $G-v$ is the graph that results when we remove $v$ and all of its outgoing 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.

Look at the proof (following slides) – we’ll pick up with this on Monday…!
Sources of a DAG

Notation: \( G-v \) is the graph that results when we remove \( v \) and all of its outgoing 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 \( v \) be a source vertex in directed graph \( G \).

Assume \( G \) is a DAG. We want to show (“WTS”) that \( G-v \) is a DAG.

Assume \( G-v \) is a DAG. We want to show (“WTS”) that \( G \) is a DAG.
Sources of a DAG

Notation: \( G-v \) is the graph that results when remove \( v \) and all of its outgoing 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 \( v \) be a source vertex in directed graph \( G \).

Prove: \( G \) is a DAG \( \Rightarrow \) \( G-v \) is a DAG. But, can’t introduce any cycles by removing edges!

Prove: \( G-v \) is a DAG \( \Rightarrow \) \( G \) is a DAG.
Sources of a DAG

*Notation:* $G-v$ is the graph that results when remove $v$ and all of its outgoing 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 $v$ be a source vertex in directed graph $G$.

Prove: $G$ is a DAG $\implies G-v$ is a DAG. But, can’t introduce any cycles by removing edges!

Prove: $G-v$ is a DAG $\implies G$ is a DAG. ?? Contrapositive ...

\[ \neg p \implies \neg q \implies (G \text{ not a DAG}) \]
Sources of a DAG

**Notation:** $G-v$ is the graph that results when remove $v$ and all of its outgoing 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 $v$ be a source vertex in directed graph $G$.

**Prove:** $G$ is a DAG $\implies G-v$ is a DAG. But, can’t introduce any cycles by removing edges!

**Prove:** $G$ is not a DAG $\implies G-v$ is not a DAG. But, any cycle in $G$ can't include a source (because a source has no incoming edges). So any cycle in $G$ 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 *(can't find a source...!)*
    Return that G is not a DAG.

*(G has a cycle...)*
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:
Choose first x in S.
For each y adjacent to x,
  Decrement InDegree[y] and
  If 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**

<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>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>
<td>2</td>
</tr>
</tbody>
</table>

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

Output: $A, \_,$

**Diagram**
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: \( S = \{ G, B, C, E \} \)

Output: \( A, C \)

Choose first \( x \) in \( S \).
For each \( y \) adjacent to \( x \),
  
  Decrement \( \text{InDegree}[y] \)
  
  If \( \text{InDegree}[y] = 0 \), add \( y \) to \( S \).
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>0</td>
<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: $A, G, B$

Choose first $x$ in $S$.
For each $y$ adjacent to $x$,
Decrement InDegree[$y$]
If InDegree[$y$] = 0, add $y$ to $S$. (ETC.)
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>0</td>
<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></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>0</td>
<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 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>0</td>
<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: \( S = D, H \)


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></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

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: \( S = \{F, I\} \)

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></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: $S = \{I\}$

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

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>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.
(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.

How many quarters? 3
Layers of a DAG

First layer

all nodes that are sources

Next layer

all nodes that are now sources

(once we remove previous layer and its outgoing edges)

Repeat…
Prerequisites for CSE classes

How many quarters (layers) before take all classes?

A. 1.
B. 2.
C. 3.
D. 4.
E. More than four.
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

\[
\begin{align*}
\{0, 1, 2, 3\} & \times \{0, 1, \ldots, 5\} \\
\times \{0, 1, \ldots, 8\} & \times \{0, 1, \ldots, 9\} = 216
\end{align*}
\]
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

(l, m, s)

means

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

(not possible)
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.**

(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

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

**integer values

The three columns total 8 in each row. 24 rows.
Which configurations are actually possible?

Construct a graph by considering all possible moves (pours) from each configuration.
Tartaglia's Pouring Problem

Which configurations are actually possible?

Construct a graph by considering all possible moves (pours) from each configuration.
Tartaglia's Pouring Problem

Which configurations are actually possible?

Construct a graph by considering all possible moves (pours) from each configuration.
Which configurations are actually possible?

Construct a graph by considering all possible moves (pours) from each configuration.
Tartaglia's Pouring Problem

Which configurations are actually possible?

Possible in blue.  Impossible in red.

<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>
Graph reachability: WHAT

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 $G$.

Q: "Given a graph $G=(V,E)$, where... is there a path in $G$ from $s$ to $t$?"
Graph reachability: HOW

Given a directed graph $G = (V, E)$ 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

"goal" = all vertices reachable from $s$ via a directed path in $G$.
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 \( X. \)
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.

Before any iterations of while loop…
X = empty      F = {(8,0,0)}      U = green nodes
“s”
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
Different methods of choosing v give breadth-first search vs. depth-first search

**Graph reachability: HOW**

```plaintext
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 second iteration of while loop…

v = (8,0,0)
X = {(8,0,0), (3,5,0)}
F = {(3,2,3), (0,5,3) (5,0,3)}
**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 $G$?
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).
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.

Proof by induction on \(t\).

(Q: What if we are bad programmers and this takes a long time?)
Graph reachability: WHY

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 \( X. \)

**Claim:** After \( t^{\text{th}} \) 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 the while loop, \( X \) is initialized as empty and \( F \) is initialized as \( \{s\}. \ And, \ s \ is reachable from \( s \) in \( G. \)

☺
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 that after the \( t \)th iteration, every element of X or F is reachable from s in G.

What happens in the \( t+1 \)st iteration?
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 that after the t\(^{th}\) iteration, every element of X or F is reachable from s in G.

**What happens in the t+1\(^{st}\) iteration?**

ADD neighbors of v to F
MOVE v from F to X
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 $G$.

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

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

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

**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 $X$.

*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. \(O(n^2)\)
B. \(O(n)\)
C. \(O(\text{degree}(v))\)
D. \(O(|F|)\)
E. None of the above.

A look at, rewrite, \(O(1)\) entries of Status[]
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 go through the whole for loop for a given v?

A. \( O(n^2) \)
B. \( O(n) \)
C. \( O(\text{degree}(v)) \)
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

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

\[ i.e. \ O(2 \ |E| ) \]
\[ i.e. \ O( \ |E| ) \]
Announcements

HW5 Due Thursday 11:59pm

Office Hours and 1-1 sessions
Are you using these well?
Lots on the course calendar!

MT2 scope = through “Counting II” (no “encoding”)
Another Special Type of Graph: Trees
Trees

1. Definitions of trees

2. Properties of trees

3. Revisiting uses of trees

A (fun) SPIS 2013 talk, “Trees, Trees and More Trees”:
A **rooted tree** is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.
A **rooted tree** is a connected directed acyclic graph in which one vertex has been designated the root, which has no incoming edges, and every other vertex has exactly one incoming edge.

Special case of DAGs from last class. Note that each vertex in middle has *exactly one* incoming edge from layer above. Edges are directed *away from* the root.
Which of the following directed graphs are trees (with root indicated in green)?

A. ×
   DAG, but not tree

B. ×
   DAG, but not tree

C. rooted tree

D. ×
   has a cycle → not a DAG
(Rooted) Trees: definitions

Rosen p. 747-749
If vertex $v$ is not the root, it has exactly one incoming edge, which is from its parent, $p(v)$. The height of vertex $v$ is given by the recurrence:

$$h(v) = h(p(v)) + 1 \quad \text{if } v \text{ is not the root, and } h(r) = 0$$
(Rooted) Trees: definitions

**Height** of vertex $v$: $h(v) = h(p(v)) + 1$  
if $v$ is not the root, and  
h(r) = 0

What is the height of the red vertex?
A. 0  
B. 1  
C. 2  
D. 3  
E. None of the above.
**Height** of vertex \( v \): 
\[
\text{h}(v) = \text{h}( \text{p}(v) ) + 1 \quad \text{if } v \text{ is not the root, and} \quad \text{h}(r) = 0
\]

*Height* of tree is maximum height of a vertex in the tree.

\[ \text{height} (T) = 3 \]
A **binary tree** is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?
A. 2  ✓
B. 3  ✓
C. 6  ✓
D. 8  ✓
E. None of the above.
A **binary tree** is a rooted tree where every (internal) vertex has no more than 2 children.

How many leaves does a binary tree of height 3 have?

A. 2
B. 3
C. 6
D. 8
E. None of the above.

*See Theorem 5 for proof of upper bound*
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

Which of the following are full binary trees?

A.

B.

C.

D.

C, D ✔
A **full** binary tree is a rooted tree where every internal vertex has exactly 2 children.

**At most** how many vertices are there in a full binary tree of height $h$?

A. $\Theta(h)$

B. $\Theta(2^h)$

C. $\Theta(h^2)$

D. $\Theta(\log h)$

Max number of vertices when tree is balanced
A full binary tree is a rooted tree where every internal vertex has exactly 2 children.

**Key insight:** number of vertices doubles on each level.

\[
\text{Total \# vertices} \quad n = 1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1 \quad \text{i.e.} \quad \Theta(2^h)
\]

If \( n \) is number of vertices:

\[
n = 2^{h+1} - 1
\]

so

\[
h = \log(n+1) - 1 \quad \text{i.e.} \quad \Theta(\log n)
\]

Max number of vertices when tree is balanced
Relating height and number of vertices:

$$\log(n+1) - 1 \leq h \leq \_\_\_$$

This is what we just proved.

How do we prove?

What tree with n vertices has the greatest possible height?
Relating height and number of vertices:

\[ \log(n+1) - 1 \leq h \leq n - 1 \]

This is what we just proved.

Rosen p. 749

Adding another vertex \( \Rightarrow h \) can increase by at most 1

What tree with \( n \) vertices has the greatest possible height?

How do we prove?
Trees

1. Definitions of trees
2. Properties of trees
3. Revisiting uses of trees

In data structures:
Binary search trees

We'll start with this next time
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

Implementation

Each vertex is an object with the fields

\[ p = \text{parent} \quad \]
\[ lc = \text{left child} \quad \]
\[ rc = \text{right child} \quad \]
\[ \text{value} \quad \]

When is \( p \) null?

A. If we have an error in our implementation.
B. When the value is 0.
C. When the vertex is a leaf node.
D. When the vertex is the root node.
E. None of the above.

Rosen, pages 757-759.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

**Implementation**

Each vertex is an object with the fields

\[ p = \text{parent} \]
\[ lc = \text{left child} \]
\[ rc = \text{right child} \]
\[ \text{value} \]

When is \( lc \) null?

A. If we have an error in our implementation.
B. When the value is 0.
C. When the vertex is a leaf node.
D. When the vertex is the root node.
E. None of the above.
Binary Search Trees

- Facilitate binary search (must maintain sorted order of data)
- Dynamic

For each vertex v
- If x is in subtree rooted at lc(v),
  \[ \text{value}(x) \leq \text{value}(v). \]
- If x is in the subtree rooted at rc(v),
  \[ \text{value}(x) \geq \text{value}(v). \]

Compare with p. 757 of Rosen: “both larger than … and smaller than …”