Next exams:

**Thursday 10/31**: Graph algorithms

HWs due **Monday 10/21** or **Wed. 10/23**
& **Monday 10/28**

**Thursday 11/21**: NP-completeness

No class—**Tuesday 11/26**

See class website.
Graphs:

\[ G = (V, E) \]
\[ V = \text{vertices} \]
\[ E = \text{edges} \]
\[ n = |V| \]
\[ m = |E| \]

Undirected graphs: edge \((i, j) \in E\) means \(i \& j\) connected by an edge.

Directed graphs: \(i \rightarrow j \in E\) means edge from \(i\) to \(j\).

Representing graphs:

1) Adjacency matrix \(A\) size \(O(n^2)\)
2) Adjacency list size \(O(n+m)\)

Exploring graphs:

DFS = depth first search
BFS = breadth first search.
**DFS:**

Stack = LIFO  
= last-in, first-out

push  

pop

**BFS:**

Queue = FIFO  
= first-in, first-out

enqueue  

dequeue

**Example:**

![Graph Diagram]

tree edges  
back edges
**DFS:** can implement a stack using recursion.

**Pseudocode:**

\[
\text{DFS}(G): \\
\text{for all } v \in V, \text{ set } \text{visited}(v) = \text{FALSE} \\
\text{Explore}(v) \\
\text{for all } w \in V, \text{ if not } \text{visited}(v) \text{ then } \text{Explore}(v) \\
\]

\[
\text{Explore}(w): \\
\text{visited}(w) = \text{TRUE} \\
\text{for all } (w, z) \in E: \\
\text{if not } \text{visited}(z) \text{ then } \text{Explore}(z) \\
\]

**Explore:** finds all vertices reachable from \( z \). Can use to find connected components.
For undirected $G$, vertices $v$ & $w$ are connected if there is a path between $v$ & $w$.

Connected components: maximal set of connected vertices.

Example:

![Graph](image)

3 components: $\{A, B, E, I, J\}$, $\{F, I, G, D, H, K, L\}$

To find connected components:

1) Choose arbitrary start vertex $z$
2) Run $\text{Explore}(z)$ to find component containing $z$
3) Choose an unexplored $z$ & repeat.

Same pseudocode as $\text{DFS}$ just keep track of connected component #.
Here's the pseudocode:

**DFS(G):**

for all \( v \in V \), \( \text{visited}(v) = \text{FALSE} \)

\( cc = 0 \)

for all \( v \in V \),

if not \( \text{visited}(v) \)

then \( [cc++] \)

\[ \text{Explore}(v) \]

**Explore(z):**

\( \text{visited}(z) = \text{TRUE} \)

\( \text{ccnum}(z) = cc \)

for each \( (z, w) \in E \):

if not \( \text{visited}(w) \)

then \( \text{Explore}(w) \)

Running time: \( O(n+m) \), for \( G \) in adjacency list representation.
How about connectivity in directed graphs?

- Need more info from DFS.
- Add a clock
- Keep track of preorder # & postorder #

\[\text{Pre}(v) = \text{time start exploring } v\]
\[\text{Post}(v) = \text{time finish exploring all neighbors of } v\]

**DFS(G):**

- For all \(v \in V\), \(\text{visited}(v) = \text{FALSE}\)
- \(\text{clock} = 1\)
- For all \(v \in V\), if not \(\text{visited}(v)\)
  - then Explore\((v)\)

**Explore\((v)\):**

- \(\text{visited}(v) = \text{TRUE}\)
- \(\text{Pre}(v) = \text{clock}\)
- \(\text{clock}++\)
- For every \((v,w) \in E\)
  - if not \(\text{visited}(w)\) then Explore\((w)\)
- \(\text{Post}(w) = \text{clock}\)
- \(\text{clock}++\)
Example:

Let's run DFS starting at B
3 types of non-tree (unexplored) edges

**Forward:** $D \rightarrow G$: vertex to descendant

$\text{Pre}(D) < \text{pre}(G) < \text{Post}(G) < \text{Post}(D)$

$b/c \ G \ is \ in \ subtree \ of \ G$

**Back:** $E \rightarrow A, F \rightarrow B$: vertex to ancestor

$\text{Pre}(B) < \text{pre}(F) < \text{Post}(F) < \text{Post}(B)$

$F \ is \ in \ B's \ subtree$

**Cross:** $F \rightarrow H, H \rightarrow G$

$\text{Pre}(H) < \text{post}(H) < \text{pre}(F) < \text{Post}(F)$

$\text{Start & finish } H \ before \ start/finish \ F.$
Property: Directed $G$ has a cycle if its DFS tree contains a back edge.

Proof:

$\Rightarrow$ if $G$ has a cycle $C$, let $v$ be 1st vertex in $C$ visited in the DFS. Then $C \setminus v$ is reachable from $v$ so all are in $v$'s subtree of the DFS tree.

$\Rightarrow$ $\geq 1$ edge from $C \setminus v$ to $v$ which is a back edge.

$\Leftarrow$ Say the DFS tree has a back edge $w \rightarrow v$.
Then $w$ is a descendant of $v$ so there is a path $P$ from $v$ to $w$.

$\&$ taking $P \cup (w, v)$ gives a cycle.
So if no cycles in directed $G$ then no back edges.

For back edge $w \rightarrow v$, $\text{Post}(w) < \text{Post}(v)$

For all other edges $w \rightarrow v$, $\text{Post}(v) < \text{Post}(w)$.

So can detect if $G$ has a cycle by checking Post #\'s to see if we find a back edge.
DAG = Directed acyclic graph. 
  no cycles hence no back edges

Topologically sorting a DAG 
= order the vertices so that 
  all edges go left to right 
  (lower #) (higher #)

Easy: sort by decreasing Post #
  highest Post # → lowest Post #

Since no back edges,
  every edge w → v 
  has Post(w) > Post(v) 
  so left → right.
Example:

Run DFS:

Topologically sorted as:

(There are other topological orderings)