Depth-First Search and Directed Graphs
Announcements/ Reminders

• Review. **Problem Set Advice handout**
• Can we use results proved in class in assignment solutions?
  • Yes
• **Homework 0 Feedback**: check for annotated comments in PDF along with text box comments, preview of future grading
• Look at **Homework 0 Sample Solutions** posted on **GLOW**
• Pay close attention to feedback: some proofs were not proofs
• **Discussion**:
  • Geometric series question
  • Induction question
Story So Far

• Breadth-first search
• Using breadth-first search for connectivity
• Using breadth-first search for testing bipartiteness

**BFS (G, s):**

Put s in the queue Q

While Q is not empty
   Extract v from Q
   If v is unmarked
      Mark v
      For each edge (v, w):
         Put w into the queue Q
Generalizing BFS: Whatever-First

If we change how we store the explored vertices (the data structure we use), it changes how we traverse

Whatever-First-Search \((G, s)\):

1. Put \(s\) in the **bag**
2. While **bag** is not empty
   - Extract \(v\) from **bag**
   - If \(v\) is unmarked
     - Mark \(v\)
     - For each edge \((v, w)\):
       - Put \(w\) into the **bag**

Depth-first search: when **bag** is a **stack**, not queue
Depth-First Search: Recursive

• Perhaps the most natural traversal algorithm
• Can be written *recursively* as well
• Both versions are the same; can actually see the “recursion stack” in the iterative version

**Recursive-DFS(u):**
Set status of u to marked # discovered u
for each edges (u, v):
  if v's status is unmarked:
    DFS(v)
# done exploring neighbors of u
Depth-first Search Example
DFS Running Time

- Inserts and extracts to a stack: $O(1)$ time
- For every node $v$, explore degree($v$) edges
  \[
  \sum_{v} \text{degree}(v) = 2m
  \]
- Connected graphs have $m \geq n - 1$ and thus is $O(m)$ and for general graphs, it is $O(n + m)$

**IterativeDFS($s$):**

1. **Push($s$)**
2. while the stack is not empty
   1. $v \leftarrow \text{Pop}$
   2. if $v$ is unmarked
      1. mark $v$
      2. for each edge $vw$
         1. **Push($w$)**
Depth-First Search Tree

- DFS returns a spanning tree, similar to BFS

\[
\text{DFS-Tree}(G, s): \\
\text{Put } (\emptyset, s) \text{ in the stack } S \\
\text{While } S \text{ is not empty} \\
\text{Extract } (p, v) \text{ from } S \\
\text{If } v \text{ is unmarked} \\
\quad \text{Mark } v \\
\quad \text{parent}(v) = p \\
\quad \text{For each edge } (v, w): \\
\quad \quad \text{Put } (v, w) \text{ into the stack } S
\]

- The spanning tree formed by parent edges in a DFS are usually long and skinny
**Lemma.** For every edge $e = (u, v)$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

**Proof.** Obvious if edge $e$ is in $T$.

Suppose edge $e$ is not in $T$. Without loss of generality, suppose DFS is called on $u$ before $v$.

- When the edge $u, v$ is inspected $v$ must have been already marked visited (why?)
  - Or else $(u, v) \in T$ and we assumed otherwise
- Since $(u, v) \notin T$, $v$ is not marked visited during the DFS call on $u$
- Must have been marked during a recursive call within DFS($u$)
  - Thus $v$ is a descendant of $u$ ■
In-Class Exercise

**Question.** Given an undirected connected graph $G$, how can you detect (in linear time) that contains a cycle?

[Hint. Use DFS]
In-Class Exercise

**Question.** Given an undirected connected graph $G$, how can you detect (in linear time) that contains a cycle?

**Idea.** When we encounter a back edge $(u, v)$ during DFS, that edge is necessarily part of a cycle (cycle formed by following tree edges from $u$ to $v$ and then the back edge from $v$ to $u$).

**Cycle-Detection-DFS**($u$):
- Set status of $u$ to marked  
  
  # discovered $u$
- for each edges $(u, v)$:
  - if $v$'s status is unmarked:
    - DFS($v$)
  - else  
    - # found an edge to a marked node
    - found a back edge, report a cycle!
- # done exploring neighbors of $u$
Directed Graphs

Notation. $G = (V, E)$.

- Edges have “orientation”
- Edge $(u, v)$ or sometimes denoted $u \rightarrow v$, leaves node $u$ and enters node $v$
- Nodes have “in-degree” and “out-degree”
- No loops or multi-edges (why?)

Terminology of graphs extend to directed graphs: directed paths, cycles, etc.
Directed Graphs in Practice

Web graph:
- Webpages are nodes, hyperlinks are edges
- Orientation of edges is crucial
- Search engines use hyperlink structure to rank web pages

Road network
- Road: nodes
- Edge: one-way street
Directed reachability. Given a node $s$ find all nodes reachable from $s$.

- Can use both BFS and DFS. Both visit exactly the set of nodes reachable from start node $s$.

- **Strong connectivity.** Connected components in directed graphs defined based on mutual reachability. Two vertices $u$, $v$ in a directed graph $G$ are mutually reachable if there is a directed path from $u$ to $v$ and from $v$ to $u$. A graph $G$ is **strongly connected** if every pair of vertices are mutually reachable.

- The mutual reachability relation decomposes the graph into strongly-connected components.

- **Strongly-connected components.** For each $v \in V$, the set of vertices mutually reachable from $v$, defines the strongly-connected component of $G$ containing $v$. 


Strongly Connected Components
Deciding Strongly Connected

**First idea.** How can we use BFS/DFS to determine strong connectivity? Recall: BFS/DFS on graph $G$ starting at $v$ will identifies all vertices reachable from $v$ by directed paths

- Pick a vertex $v$. Check to see whether every other vertex is reachable from $v$;
- Now see whether $v$ is reachable from every other vertex

**Analysis**

- First step: one call to BFS: $O(n + m)$ time
- Second step: $n - 1$ calls to BFS: $O(n(n + m))$ time
- Can we do better?
Testing Strong Connectivity

Idea. Flip the edges of G and do a BFS on the new graph

- Build $G_{rev} = (V, E_{rev})$ where $(u, v) \in E_{rev}$ iff $(v, u) \in E$
- There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$
- Call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis (Performance)

- $BFS(G, v)$: $O(n + m)$ time
- Build $G_{rev}$: $O(n + m)$ time. [Do you believe this?]
- $BFS(G_{rev}, v)$: $O(n + m)$ time
- Overall, linear time algorithm!

Kosaraju’s Algorithm
Testing Strong Connectivity

Idea. Flip the edges of $G$ and do a BFS on the new graph

- Build $G_{\text{rev}} = (V, E_{\text{rev}})$ where $(u, v) \in E_{\text{rev}}$ iff $(v, u) \in E$
- There is a directed path from $v$ to $u$ in $G_{\text{rev}}$ iff there is a directed path from $u$ to $v$ in $G$
- Call $\text{BFS}(G_{\text{rev}}, v)$: Every vertex is reachable from $v$ (in $G_{\text{rev}}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis (Correctness)

- **Claim.** If $v$ is reachable from every node in $G$ and every node in $G$ is reachable from $v$ then $G$ must be strongly connected
- **Proof.** For any two nodes $x, y \in V$, they are mutually reachable through $v$, that is, $x \leadsto v \leadsto y$ and $y \leadsto v \leadsto z \blacksquare$
Directed Acyclic Graphs (DAGs)

**Definition.** A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

**Question.** Given a directed graph $G$, can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?
Directed Acyclic Graphs (DAGs)

**Definition.** A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

**Question.** Given a directed graph $G$, can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?
Directed Acyclic Graphs (DAGs)

**Definition.** A directed graph is acyclic (or a DAG) if it contains no (directed) cycles.

**Question.** Given a directed graph \( G \), can you detect if it has a cycle in linear time? Can we apply the same strategy (DFS) as we did for undirected graphs?

```
Cycle-Detection-Directed-DFS(u):
    Set status of u to marked # discovered u
    for each edges (u, v):
        if v's status is unmarked:
            DFS(v)
        else if v is marked but not finished
            report a cycle!
    mark u finished # done exploring neighbors of u
```