Depth-first Search and Directed Graphs

Story So Far

- Breadth-first search
- Using breadth-first search for connectivity
- Using bread-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
```

The BFS Tree

- Can remember parent nodes (the node at level i that lead us to a given node at level $i + 1$)

```
BFS-Tree(G, s):
  Put (∅, s) in the queue Q
  While Q is not empty
      Extract (p, v) from Q
          If v is unmarked
              Mark v
              parent(v) = p
              For each edge (v, w):
                  Put (v, w) into the queue Q
```

Spanning Trees

- **Definition.** A spanning tree of an undirected graph G is a connected acyclic subgraph of G that contains every node of G.
- The tree produced by the BFS algorithm (with $(u, \text{parent}(u))$ as edges) is a spanning tree of the component containing s.

Spanning Trees

- **Definition.** A spanning tree of an undirected graph G is a connected acyclic subgraph of G that contains every node of G.
- The tree produced by the BFS algorithm (with $(u, \text{parent}(u))$ as edges) is a spanning tree of the component containing s.
- The BFS spanning tree gives the shortest path from s to every other vertex in its component (we will revisit shortest path in a couple of lectures)
- BFS trees in general are short and bushy

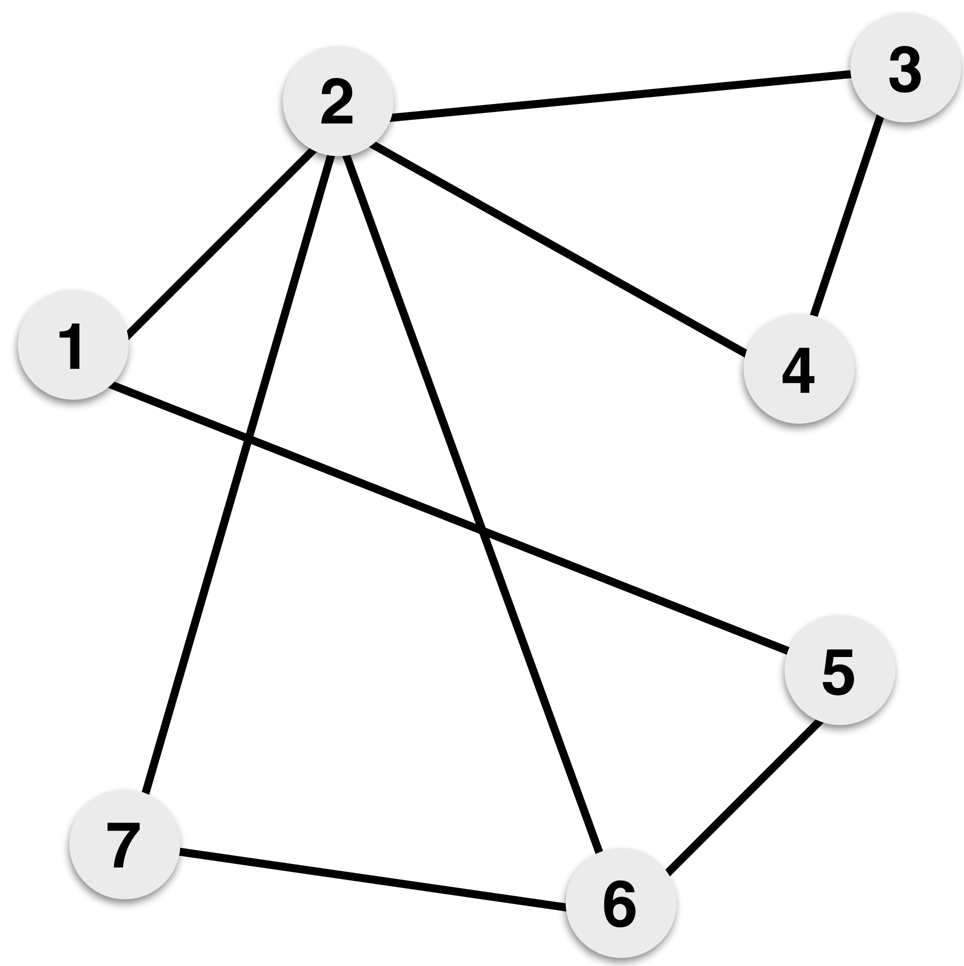
Generalizing BFS: Whatever-First

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

```
Whatever-First-Search (G, s):
  Put s in the bag
  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 Example



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

```
RECURSIVEDFS(v):
    if v is unmarked
        mark v
        for each edge vw
            RECURSIVEDFS(w)
```

```
ITERATIVEDFS(s):
    PUSH(s)
    while the stack is not empty
        v ← POP
        if v is unmarked
            mark v
            for each edge vw
                PUSH(w)
```

Depth-first Search: Stack

- Inserts and extracts to a stack take $O(1)$ time
- Thus, overall running time is $O(n + m)$

```
ITERATIVEDFS(s):
    PUSH(s)
    while the stack is not empty
        v ← POP
        if v is unmarked
            mark v
            for each edge vw
                PUSH(w)
```

Depth-First Search Tree

- DFS returns a spanning tree, similar to BFS

```
DFS-Tree(G, s):
  Put (∅, s) in the stack S
  While S is not empty
      Extract (p, v) from S
          If v is unmarked
              Mark v
              parent(v) = p
              For each edge (v, w):
                  Put (v, w) into the stack S
```

- The spanning tree formed by parent edges in a DFS are usually long and skinny

Depth-First Search Tree

Lemma. For every edge $e = (u, v)$ in G, one of u or v is an ancestor of the other in T.

```
RecursiveDFS(p, v):
    If v is unmarked
    Mark v
    parent(v) = p   # (p, v) is a tree edge
        For each edge (v, w):
           RecursiveDFS(v, w)
```

Easier to think in terms of recursive definition

Depth-First Search Tree

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.

Depth-First Search Tree

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.

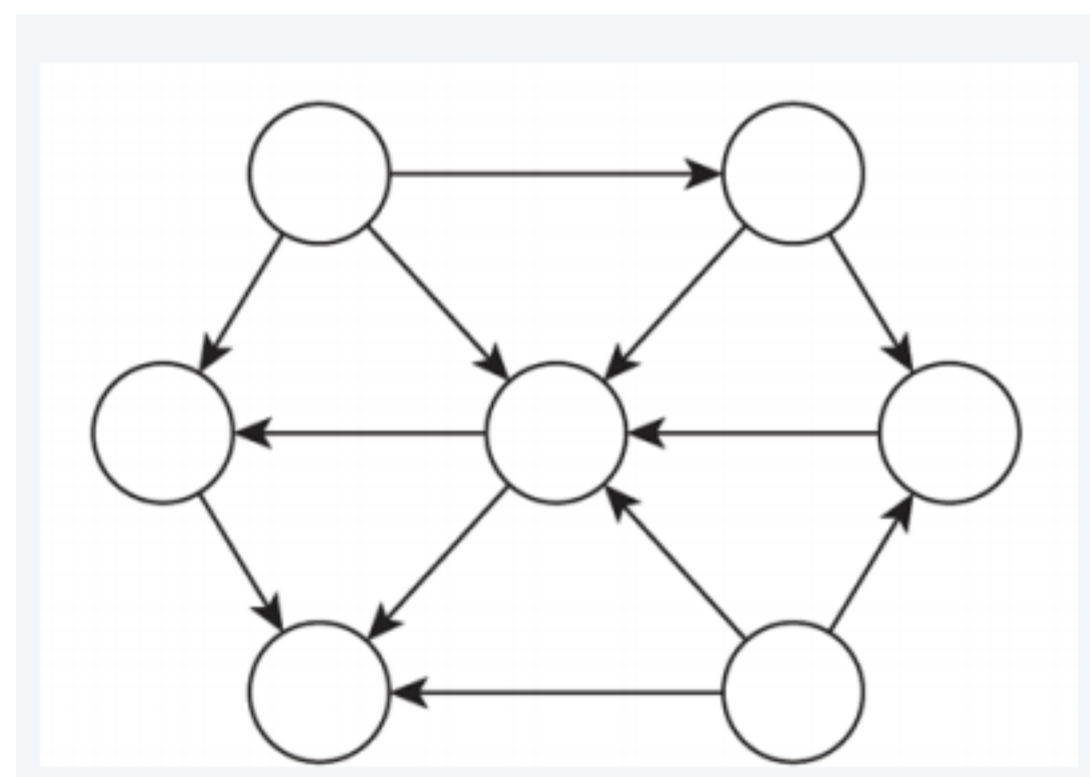
- When the edge u, v is inspected v must have been already marked visited; Or else $(u, v) \in T$
- v is not marked visited during the DFS call on u
- Must have been marked during a recursive call within DFS(u), thus v is a descendant of u

Directed Graphs

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

- Edges have "orientation"
- Edge (u, v) leaves node u and enters node v
- Nodes have "in-degree" and "out-degree"
- No loops or multi-edges (why?)

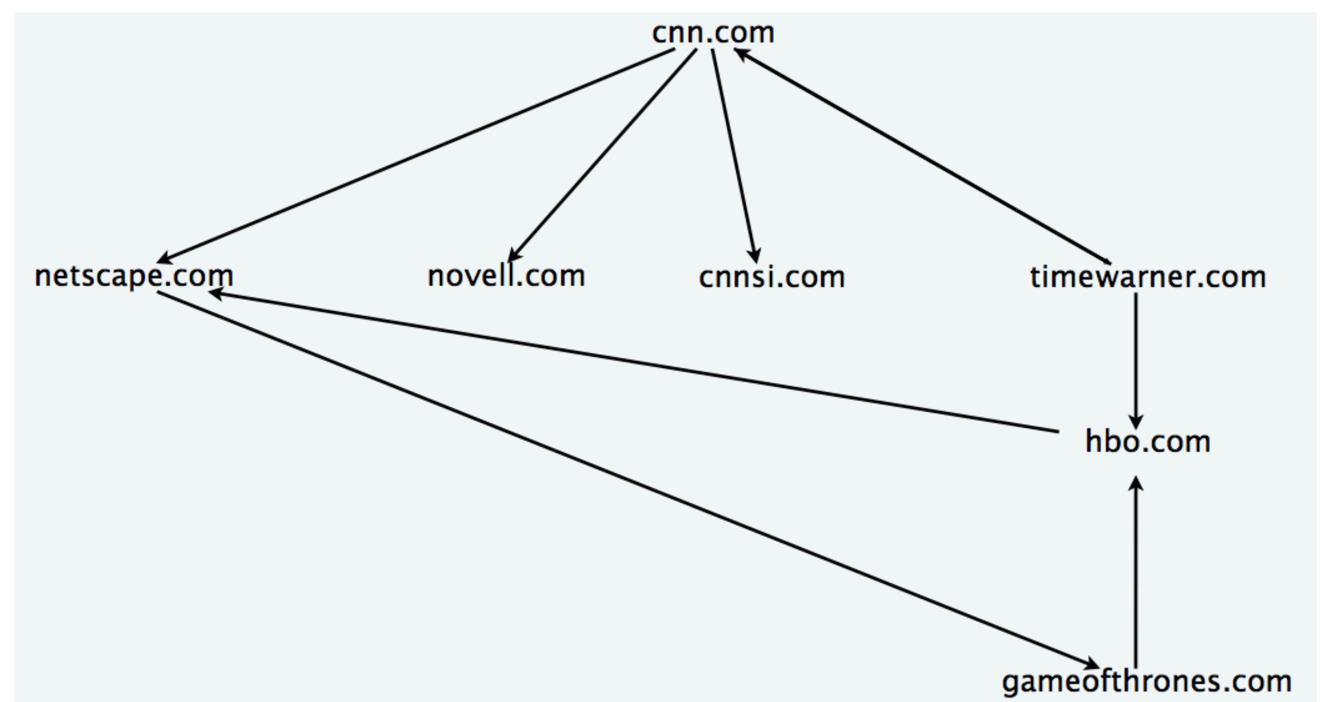
Terminology of graphs extend to directed graphs: directed paths, cycles, etc.



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 Graph Search

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 starting node s

Review: Equivalence Relation

Definition. A binary relation $\succeq$ on a set S is an equivalence relation on S if $\succeq$ has the following properties

- Reflexive: $\forall x \in S,\ x \succeq x$
- Symmetric: $\forall x, y \in S,\ x \succeq y \implies y \succeq x$
- Transitive: $\forall x, y, z \in S, x \succeq y$ and $y \succeq x \implies x \succeq z$

Question. Identify the properties in these relations:

(a) Lives-in-the-same-city-as, (b) Is-an-ancestor of

Reachability & Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- Each node is reachable from itself **(reflexive)**
- If v is reachable from u, then u is reachable from v **(symmetric)**
- If v is reachable from u, and u is reachable from w, then then v is reachable from w **(transitive)**

Connectivity & Equivalence Classes

An equivalence relation $\simeq$ on a set S gives rise to equivalence classes $S_x = \{y \mid y \simeq x\}$, also written as $[x]$

These equivalence classes have the following properties

- For every $x \in S$, $x \in S_x$
- For every $x, y \in S$, $S_x = S_y$ or $S_x \cap S_y = \emptyset$

That is, the equivalence classes partition S!

Definition (Connected component.) For each $v \in V$, $[v]$, the set of vertices reachable from v, defines the connected component of G containing v.

Connectivity in Directed Graphs

- In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric
- Can we define a related equivalence relation on the vertices of a directed graph?

Connectivity in Directed Graphs

- In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric
- Can we define a related equivalence relation on the vertices of a directed graph?
- Two vertices u, v in a directed graph G are mutually reachable if there is a directed path from u to v and from from v to u
- Mutually reachable is an equivalence relation
    - Why?

Strongly Connected

- 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

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

Deciding Strongly Connected

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

Deciding Strongly Connected

First idea. How can we use BFS to determine strong connectivity? Recall: BFS 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

Deciding Strongly Connected

First idea. How can we use BFS to determine strong connectivity? Recall: BFS 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?**

Deciding Strongly Connected

Improved 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_{rev} iff there is a directed path from u to v in G

Second step: 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).

Deciding Strongly Connected

Improved 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_{rev} iff there is a directed path from u to v in G

Second step: Call $\text{BFS}(G_{\text{rev}}, v)$: Every vertex is reachable from v (in G_{rev}) if and only if v is reachable from every vertex (in G).

Analysis

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