Last Time

• Introduction To Graphs
  • Definitions and Properties: Undirected Graphs
Today’s Outline

• More on Graphs
  • Applications and Problems
    • Testing connectedness
    • Counting connected components
      – Breadth-first and Depth-first search
  • Directed Graphs
    • Definition and Properties
  • Reachability and (Strong) Connectedness
• Graph Data Structures: Preliminaries
  • Graph Interface
Reachability and Connectedness

- Def’n: A vertex v in G is *reachable* from a vertex u in G if there is a path from u to v
- v is reachable from u *iff* u is reachable from v
- Def’n: An undirected graph G is *connected* if for every pair of vertices u, v in G, v is reachable from u (and vice versa)
- The set of all vertices reachable from v, along with all edges of G connecting any two of them, is called the *connected component of v*
Basic Graph Algorithms

• We’ll look at a number of graph algorithms
  • Connectedness: Is G connected?
    • If not, how many connected components does G have?
  • Cycle testing: Does G contain a cycle?
    • Does G contain a cycle through a given vertex?
  • If the edges of G have costs:
    • What is the cheapest subgraph connecting all vertices
      – Called a connected, spanning subgraph
    • What is a cheapest path from u to v?
  • And more....
Operations on Graphs

- What are the basic operations we need to describe algorithms on graphs?
  - Given vertices u and v: are they adjacent?
  - Given vertex v and edge e, are they incident?
  - Given an edge e, get its incident vertices (ends)
  - How many vertices are adjacent to v? (degree of v)
    - The vertices adjacent to v are called its neighbors
  - Get a list of the vertices adjacent to v
    - From which we can get the edges incident with v
Testing Connectedness

• How can we determine whether $G$ is connected?
  • Pick a vertex $v$; see if every vertex $u$ is reachable from $v$

• How could we do this?
  • Visit the neighbors of $v$, then visit their neighbors, etc. See if we reach all vertices
    • Assume we can mark a vertex as “visited”

• How do we manage all of this visiting?
  • Let’s try an example…
**Reachability: Breadth-First Search**

\[ \text{BFS}(G, v) \]  // Do a breadth-first search of \( G \) starting at \( v \)
\[ \text{// pre: all vertices are marked as unvisited} \]
\[ \text{count } \leftarrow 0; \]
\[ \text{Create empty queue } Q; \text{ enqueue } v; \text{ mark } v \text{ as visited; count}++; \]
\[ \text{While } Q \text{ isn’t empty} \]
\[ \quad \text{current } \leftarrow Q.\text{dequeue}(); \]
\[ \quad \text{for each unvisited neighbor } u \text{ of current:} \]
\[ \quad \quad \text{add } u \text{ to } Q; \text{ mark } u \text{ as visited; count}++; \]
\[ \text{return count;} \]

Now compare value returned from \( \text{BFS}(G,v) \) to size of V
BFS Reflections

• The BFS algorithm traced out a tree $T_v$: the edges connecting a visited vertex to (as yet) unvisited neighbors
• $T_v$ is called a BFS tree of $G$ with root $v$ (or from $v$)
• The vertices of $T_v$ are visited in level-order
• This reveals a natural measure of distance between vertices: the length of (any) shortest path between the vertices
Distance in Undirected Graphs

Def: The distance between two vertices $u$ and $v$ in an undirected graph $G=(V,E)$ is the minimum of the path lengths over all $u$-$v$ paths.

- It is the depth of $u$ in $T_v$: a BFS tree from $v$
- We write it as $d(u,v)$. It satisfies the properties
  - $d(u,u) = 0$, for all $u \in V$
  - $d(u,v) = d(v,u)$, for all $u,v \in V$
  - $d(u,v) \leq d(u,w) + d(w,v)$, for all $u,v,w \in V$
- This last property is call the triangle inequality
Reachability: Depth-First Search

\[ \text{DFS}(G, v) \]  // Do a depth-first search of G starting at v
\[ \text{pre: all vertices are marked as unvisited} \]
\[ \text{count} \leftarrow 0; \]
Create empty stack \( S \); push \( v \); mark \( v \) as visited; count++;
While \( S \) isn’t empty
\[ \text{current} \leftarrow S.\text{pop}(); \]
for each unvisited neighbor \( u \) of current:
\[ \text{add} \ u \ \text{to} \ S; \ \text{mark} \ u \ \text{as visited}; \ \text{count}++ \]
return count;

Now compare value returned from \( \text{DFS}(G, v) \) to size of V
DFS Reflections

• The DFS algorithm traced out a tree different from that produced by BFS
  • It still consists of the edges connecting a visited vertex to (as yet) unvisited neighbors
• It is called a DFS tree of $G$ with root $v$ (or from $v$)
• Vertices are processed in pre-order w.r.t. the tree
• By manipulating the stack differently, we could produce a post-order version of DFS
• And perhaps write DFS recursively…. 
Recursive Depth-First Search

// Before first call to DFS, set all vertices to unvisited
// Then call DFS(G, v)

DFS(G, v)

Mark v as visited; count = 1;
for each unvisited neighbor u of v:
    count += DFS(G, u);
return count;

Is it even clear that this method does what we want?!

Let’s prove some facts about it....
Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

• Proof: Induction on length d of shortest path from v to w
  • Base case: d = 0: Then v = w ✓
  • Ind. Hyp.: Assume DFS visits all vertices w of distance at most d from v (for some d ≥ 0).
  • Ind. Step: Suppose now that w is distance d+1 from v. Consider a path of length d+1 from v to w and let u be the next-to-last vertex on the path...
Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

• Proof: Induction on length d of shortest path from v to w
  • The path is v = v₀, v₁, v₂, ..., vᵣ = u, vᵣ₊₁ = w
    • The edges are implied so not explicitly written!
  • By Ind. Hyp., u is visited. At this point, if w has not yet been visited, it will be one of the unvisited vertices on which DFS() is recursively called, so it will then be visited.
Recursive Depth-First Search

Claim: DFS visits only vertices reachable from v
• Idea: Prove by induction on number of times DFS is called that DFS is only called on vertices w reachable from v

Claim: DFS counts correctly the number of vertices reachable from v
• Idea: Induction on number of unvisited vertices reachable from v
  • DFS will never be called on same vertex twice
Claim: DFS(G,v) returns the number of unvisited nodes reachable from v

Proof: Uses previous two observations

- DFS visits every node reachable from v
- DFS doesn’t visit any node not reachable from v