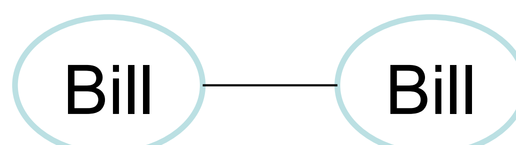
CSCI 136
Data Structures & Advanced Programming

Lecture 27

Fall 2017

Instructors:



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

A Basic Graph Fact

- Denote the degree of a vertex *v* by *deg(v)*.
- Thm: For any graph *G* = *(V,E)*

$$\sum_{v \in V} \deg(v) = 2|E|$$

where |*E*| is the number of edges in *G*

- Proof Hint: Induction on |*E*|: How does removing an edge change the equation?
  - Instead: Count pairs *(v,e)* where *v* is incident with e

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 you 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

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

Now compare value returned from 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

DFS(G, v)   // Do a depth-first search of G starting at v

// pre: all vertices are marked as unvisited

count ← *0;*

Create empty stack S; push v; mark v as visited; count++;

While S isn't empty

```
        current ← S.pop();
        for each unvisited neighbor u  of current :
                add u to S; mark u as visited; count++
```

return count;

Now compare value returned from 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 visited 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 $v = v_0, v_1, v_2, \ldots, v_d = u, v_{d+1} = w$
    - 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

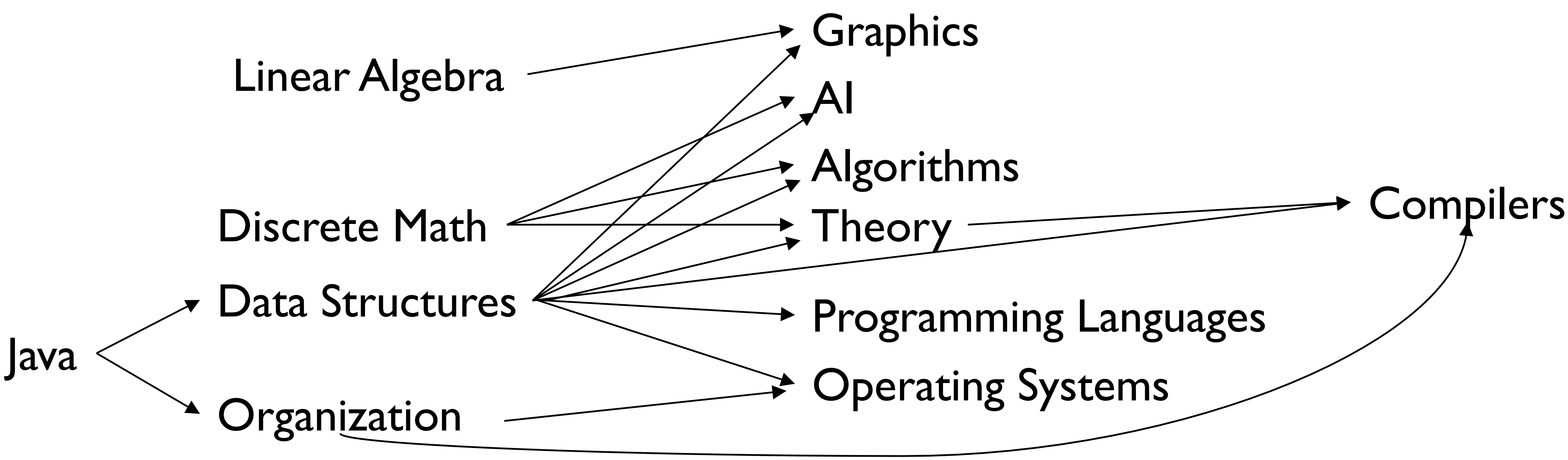
Recursive Depth-First Search

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

Directed Graphs

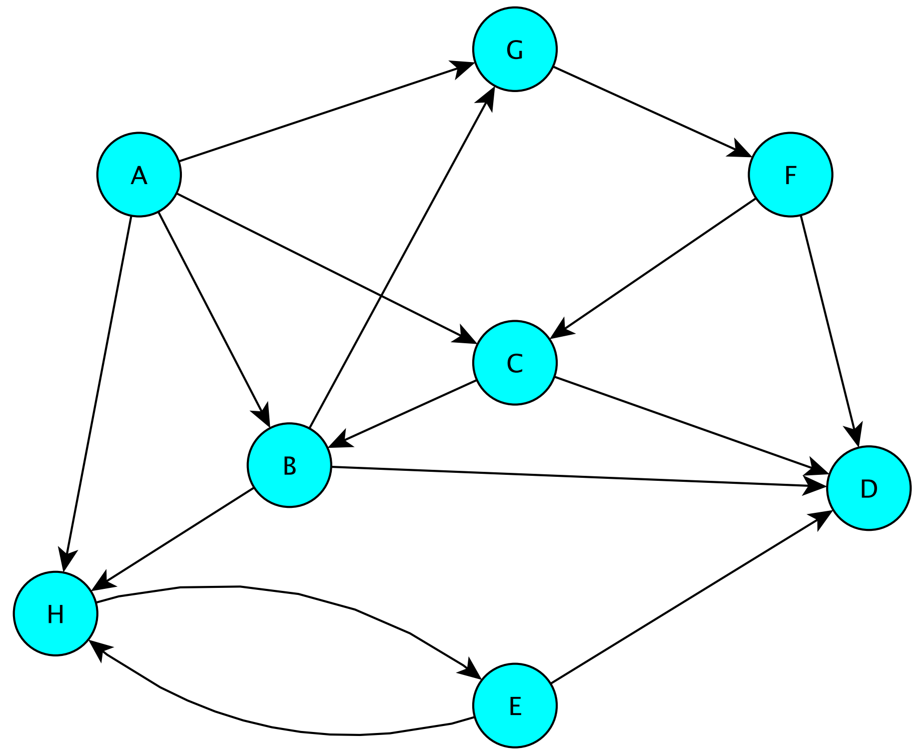


Def'n: In a *directed graph G = (V,E)*, each edge e in E is an *ordered* pair: e = (u,v) vertices: its *incident vertices*. The *source* of e is u; the *destination/target* is v.

Note: (u,v) ≠ (v,u)

Directed Graphs

- The (out) neighbors of B are D, G, H: B has out-degree 3
- The in neighbors of B are A, C: B has in-degree 2
- A has in-degree 0: it is a *source* in G; D has out-degree 0: it is a *sink* in G



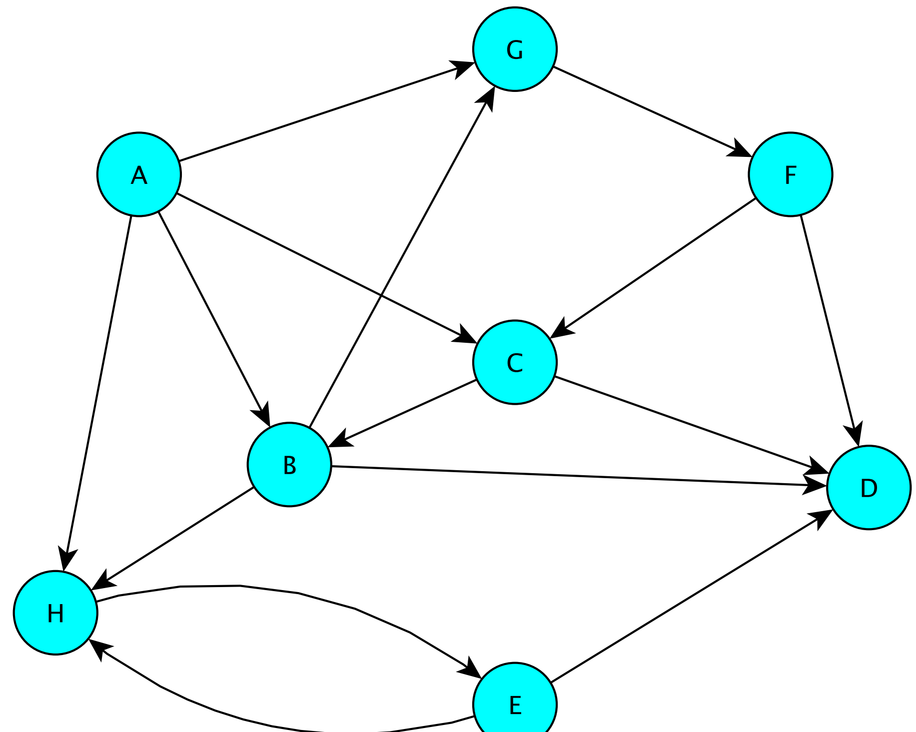
A walk is still an alternating sequence of vertices and edges

$$u = v_0, e_1, v_1, e_2, v_2, \dots, v_{k-1}, e_k, v_k = v$$

but now $e_i = (v_{i-1}, v_i)$: all edges *point along direction* of walk

Directed Graphs

- A, B, H, E, D is a walk from A to D
- It's also a (simple) path
- D, E, H, B, A is *not* a walk from D to A
- B, G, F, C, B is a (directed) cycle (it's a 4-cycle)
- So is H, E, H (a 2-cycle)



- D is reachable from A (via path A, B, D), but A is not reachable from D
- In fact, every vertex is reachable from A