CSCI 136
Data Structures &
Advanced Programming

Lecture 28
Fall 2017
Instructors: Bill Bill
Last Time

• More on Graphs
  • Built up a vocabulary to talk about graphs
  • Proved some things about graphs
  • Introduced:
    • Connectedness
    • Reachability
This Time

- More on Graphs
  - Applications and Problems
    - Testing connectedness
    - Counting connected components
    - Breadth-first
  - Depth-first search
    - And recursive depth-first search
- Directed Graphs : Introduction
Next Time?

- More Directed Graphs
  - Reachability and (Strong) Connectedness
- Graph Data Structures: Implementation
  - Graph Interface
  - Adjacency Array Implementation Basic Concept
  - Adjacency List Implementation Basic Concept
  - Adjacency Array Implementation Details
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

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

Now compare value returned from \( \text{BFS}(G, v) \) to \(|V|\)
**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 $v$
- 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
Definition: The \textit{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.

- Distance is the depth of $u$ in $T_v$ (a BFS tree from $v$)
  - We write distance as $d(u,v)$
Distance in Undirected Graphs

- Distance satisfies the following 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 \( u, v, w \in V \)

- The last property is called the *triangle inequality*
**Reachability: Depth-First Search**

\[ \text{DFS}(G, v) \quad // \text{Do a depth-first search of } G \text{ starting at } v \]

// pre: all vertices are marked as unvisited

\[ \text{count} \leftarrow 0; \]

Create empty stack \( S \); push \( v \); mark \( v \) as visited; \( \text{count}++; \)

While \( S \) isn’t empty

\[ \text{current} \leftarrow S.\text{pop}(); \]

for each unvisited neighbor \( u \) of \( \text{current} \):

\[ \text{add } u \text{ to } S; \text{mark } u \text{ as visited}; \text{count}++; \]

return \( \text{count}; \)

Now compare value returned from \( \text{DFS}(G, v) \) to \( |V| \)
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
• 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 \( d \geq 0 \)).
  - 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, ..., 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: \( \text{DFS}(G, v) \) 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 \( v \)
**Definition:** 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: $(u, v) \neq (v, u)$
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 is a source in G: A has in-degree 0
• D is sink in G: D has out-degree 0

A walk is still an alternating sequence of vertices and edges
\[ u = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k = v \]
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
Directed Graphs

• A BFS of $G$ from A visits every vertex
• A BFS of $G$ from F visits all vertices but A
• A BFS of $G$ from E visits only E, H, D

• Connectivity in directed graphs is more subtle than in undirected graphs!
Directed Graphs

- Vertices $u$ and $v$ are *mutually reachable* vertices if there are paths from $u$ to $v$ and $v$ to $u$.
- *Maximal* sets of mutually reachable vertices form the *strongly connected components* of $G$.
Implementing Graphs

• Involves a number of implementation decisions, depending on intended uses
  • What kinds of graphs will be available?
    • Undirected, directed, mixed
  • What underlying data structures will be used?
  • What functionality will be provided
  • What aspects will be public/protected/private
• We’ll focus on popular implementations for undirected and directed graphs (separately)