CSCI 136
Data Structures &
Advanced Programming

Lecture 28
Fall 2018
Instructors: Bill Bill
Announcements

- I have office hours today from 1:00-2:00pm
Last Time

- More on Graphs
  - Applications and Problems
    - Testing connectedness
    - Counting connected components
    - Breadth-first search
    - Depth-first search
      - And recursive depth-first search
Today’s Outline

• Recursive Depth First Search
  • Why it works

• Directed Graphs
  • Definition and Properties
  • Reachability and (Strong) Connectedness

• Graph Data Structures: Implementation
  • Graph Interface
  • Adjacency Array Implementation Basic Concepts
  • Adjacency List Implementation Basic Concepts
  • Adjacency Array Implementation Details
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....
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 the following by induction on number of times DFS is called:
  • 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
What *Exactly* Does DFS Do?

• Given a graph $G = (V, E)$, a vertex $v$, let $X \subseteq V$, where $v \notin X$.

• Assume $X$ are exactly the vertices of $V$ that have been marked as visited.

• Claim: $\text{DFS}(G,v)$ will visit exactly those vertices that are in the connected component of $G – X$ that contains $v$.
  
  • $G – X$ is the graph obtained by deleting the vertices of $X$–and edges using $X$–from $G$.
  
  • Prove by induction on $|V – X|$
Implementing Breadth-First Search

BFS(G, v) // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
// post: return number of visited vertices

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;
int BFS(Graph<V,E> g, V src) {
    Queue<V> todo = new QueueList<V>();
    int count = 0;
    g.visit(src); count++;
    todo.enqueue(src);
    while (!todo.isEmpty()) {
        V node = todo.dequeue();
        Iterator<V> neighbors = g.neighbors(node);
        while (neighbors.hasNext()) {
            V next = neighbors.next();
            if (!g.isVisited(next)) {
                g.visit(next); count++;
                todo.enqueue(next);
            }
        }
    }
    return count;
}
int BFS(Graph<V,E> g, V src) {
    Queue<V> todo = new QueueList<V>(); int count = 0;
    g.visit(src); count++;
    todo.enqueue(src);
    while (!todo.isEmpty()) {
        V node = todo.dequeue();
        Iterator<V> neighbors = g.neighbors(node);
        while (neighbors.hasNext()) {
            V next = neighbors.next();
            if (!g.isVisitedEdge(node,next)) g.visitEdge(next,node);
            if (!g.isVisited(next)) {
                g.visit(next); count++;
                todo.enqueue(next);
            }
        }
    }
    return count;
}
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: $(u,v) \neq (v,u)$
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 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
\[ u = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k = v \]
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)
Graphs in structure

• We want to store information at vertices and at edges, but we favor vertices
  • Let V and E represent the types of information held by vertices and edges respectively
  • Interface Graph<V,E> extends Structure<V>
    • Vertices are the building blocks; edges depend on them
• Type V holds a label for a (hidden) vertex type
• Type E holds a label for an (available) edge type
  • Label: Application-specific data for a vertex/edge
Graphs in structure

- The methods described in the Structure interface deal with vertices
  - but also impact edges: e.g., clear()
- We’ll want to add a number of similar methods to provide information about edges, and the graph itself