CSCI 136
Data Structures &
Advanced Programming

Lecture 28
Fall 2017
Instructors: Bill Bill
Last 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
Today

• Graph Data Structures: Implementation
  • Using the Graph Interface
  • Implementing the Graph Interface
    • Adjacency Array
    • Adjacency List
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 \( E \) holds a label for an (available) edge
  • label: Application-specific data for a vertex/edge
Graphs in structure

• So, the methods described in the Structure interface are about 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
Recall: Desired Functionality

• What are the basic operations we need in order to describe algorithms on graphs?
  • Given vertices \( u \) and \( v \): are they \textit{adjacent}?  
  • Given vertex \( v \) and edge \( e \), are they \textit{incident}?  
  • Given an edge \( e \), get its incident vertices (\textit{ends})  
  • How many vertices are adjacent to \( v \)? (\textit{deg}(v))  
    • The vertices adjacent to \( v \) are called its \textit{neighbors}  
  • Get a list of the neighbors of \( v \) (or the edges incident with \( v \))
Graph Interface Methods

• void add(V vLabel), V remove(V vLabel)
  • Add/remove vertex to graph

• void addEdge(V vLabel1, V vLabel2, E edgeLabel), E removeEdge(V vLabel1, V vLabel2)
  • Add/remove edge between vLabel1 and vLabel2

• boolean containsEdge(V vLabel1, V vLabel2)
  • Returns true iff there is an edge between vLabel1 and vLabel2

• Edge<V,E> getEdge(V vLabel1, V vLabel2)
  • Returns edge between vLabel1 and vLabel2

• void clear()
  • Remove all nodes (and edges) from graph
Graph Interface Methods

- boolean visit(V vLabel)
  - Mark vertex as “visited” and return previous value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as “visited”
- boolean isVisited(V vLabel), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vLabel)
  - Get iterator for all neighbors of vLabel
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges
Edge Class

• Graph edges are defined in their own public class
  • `Edge<V,E>(V vLabel1, V vLabel2, E label, boolean directed)`
  • Construct a (possibly directed) edge between two labeled vertices (`vLabel1 → vLabel2`)
  • `vLabel1 : here; vLabel2 : there`

• Useful methods (getters and setters):
  - `label()`, `here()`, `there()`
  - `setLabel()`, `isVisited()`, `isDirected()`
Reachability: 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;
add v to Q, mark v as visited, add ‘v’ to count

While Q isn’t empty
    current ← Q.dequeue();
    for each unvisited neighbor u of current:
        add u to Q, mark u as visited, add ‘u’ to count

return count;

How does this translate to code?
Breadth-First Search

```java
int BFS(Graph<V,E> g, V src) {
    int count = 0; Queue<V> todo = new QueueList<V>();
    todo.enqueue(src);
    g.visit(src); count++;
    while (!todo.isEmpty()) {
        V vertex = todo.dequeue();
        Iterator<V> neighbors = g.neighbors(vertex);
        while (neighbors.hasNext()) {
            V next = neighbors.next();
            if (!g.isVisited(next)) {
                todo.enqueue(next);
                g.visit(next); count++;
            }
        }
    }
    return count;
}
```
Breadth-First Search of Edges

```java
int BFS(Graph<V,E> g, V src) {
    int count = 0; Queue<V> todo = new QueueList<V>();
    todo.enqueue(src);
    g.visit(src); count++;
    while (!todo.isEmpty()) {
        V vertex = todo.dequeue();
        Iterator<V> neighbors = g.neighbors(vertex);
        while (neighbors.hasNext()) {
            V next = neighbors.next();
            if (!g.isVisitedEdge(vertex, next))
                g.visitEdge(vertex, next);
            if (!g.isVisited(next)) {
                todo.enqueue(next);
                g.visit(next); count++;
            }
        }
    }
    return count;
}
```
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;

How does this translate to code?
Recursive Depth-First Search

```java
int depthFirstSearch(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
        V next = neighbors.next();
        if (!g.isVisited(next))
            count += depthFirstSearch(g, next);
    }
    return count;
}
```
Beyond the API

• So far we have *used* the structure5 graph interface methods in graph traversal algorithms

• How would we design classes that *implement* the interface?
  • What data structures should store the vertices?
  • What data structures should store the edges?
Representing Graphs

• Two standard approaches
  • Option 1: Array-based (directed and undirected)
  • Option 2: List-based (directed and undirected)

• We’ll look at both
  • Array-based graphs store the edge information in a 2-dimensional array indexed by the vertices
  • List-based graphs store the edge information in a (1-dimensional) array of lists
    • The array is indexed by the vertices
    • Each array element is a list of edges incident with that vertex
Adjacency Array: Directed Graph

Entry \((i, j)\) stores 1 if there is an edge from \(i\) to \(j\); 0 otherwise.

E.G.: \(\text{edges}(B, C) = 1\) but \(\text{edges}(C, B) = 0\)
**Adjacency Array: Undirected Graph**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Entry (i,j) store 1 if there is an edge between i and j; else 0

E.G.: edges(B,C) = 1 = edges(C,B)
The vertices are stored in an array $V[]$. $V[]$ contains a linked list of edges incident to a given vertex.
Adjacency List: Directed Graph

The vertices are stored in an array $V[]$. $V[]$ contains a linked list of edges having a given source.
Graph Classes in structure5
Why so many?! 

- There are two types of graphs: undirected & directed 
- There are two implementations: arrays and lists 
- We want to be able to avoid large amounts of identical code in multiple classes 
- We abstract out features of implementation common to both directed and undirected graphs 

We’ll tackle array-based graphs first....