CSCI 136
Data Structures &
Advanced Programming

Lecture 28
Fall 2017
Instructors: Bill

Bill ➔ Bill
Announcements

• Dzung will be moving his TA hours from 2-4pm Saturday to 2-4pm Sunday this week.
Last Time

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

• 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
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 E holds a label for an (available) edge
  • Label: Application-specific data for a vertex/edge
Graphs in structure5

- 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
Recall: Desired Functionality

- 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 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 \rightarrow vLabel2`)
  - `vLabel1 : here; vLabel2 : there`

- Useful methods:
  - `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 \leftarrow 0;
Create empty queue Q; enqueue v; mark v as visited; count++
While Q isn’t empty
    current \leftarrow Q.dequeue();
    for each unvisited neighbor u of current :
        add u to Q; mark u as visited; count++
return count;
Breadth-First Search

```java
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;
}
```
Breadth-First Search of Edges

```java
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;
}
```
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;
Recursive Depth-First Search

```java
int DFS(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 += DFS(g, next);
    }
    return count;
}
```
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 \(\text{edges}(C,B) = 0\)

<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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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.: \( \text{edges}(B,C) = 1 = \text{edges}(C,B) \)
Adjacency List: Directed Graph

The vertices are stored in an array $V[]$
$V[]$ contains a linked list of edges having a given source
Adjacency List: Undirected Graph

The vertices are stored in an array V[]. V[] contains a linked list of edges incident to a given vertex.
Graph Classes in structure 5

- **Interface**
- **Abstract Class**
- **Class**

**Structure**

- **Graph**
  - **GraphMatrix**
    - **GraphMatrixDirected**
    - **GraphMatrixUndirected**
  - **GraphList**
    - **GraphListDirected**
    - **GraphListUndirected**

- **Vertex**
  - **GraphMatrixVertex**
  - **GraphListVertex**

- **Edge**
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....