CSCI 136
Data Structures &
Advanced Programming

Lecture 29
Fall 2017

Instructors:  

Bill J \rightarrow Bill L
Bill L \rightarrow Bill J
Last Time

- Graph Data Structures: Implementation
  - Graph Interface
  - Adjacency Array Implementation Basic Concepts
  - Adjacency List Implementation Basic Concepts
Today’s Outline

• Graph Data Structures: Implementation
  • Adjacency Array Implementation Details
  • Adjacency List Implementation Details
    • Featuring many Iterators!
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.: \(\text{edges}(B,C) = 1 = \text{edges}(C,B)\)
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

- Interface
- Abstract Class
- Class

Structure

Graph

- GraphMatrix
  - GraphMatrixDirected
  - GraphMatrixUndirected

- GraphList
  - GraphListDirected
  - GraphListUndirected

Vertex

- GraphMatrixVertex
- GraphListVertex

Edge
Graph Classes in structure

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....
## Adjacency Array: Directed 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>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>

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

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)
### Adjacency Array: Undirected Graph

**Halving the Space (not in structure5)**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>
<tr>
<td>2</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>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</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>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(i,j) maps to i*7+j
Vertex and GraphMatrixVertex

• We need to define a Vertex class
  • Unlike the Edge class, Vertex class is not public
  • Useful Vertex methods:
    V label(), boolean visit(),
    boolean isVisited(), void reset()
• GraphMatrixVertex class adds one more useful attribute to Vertex class
  • Index of node (int) in adjacency matrix
    int index()
  • Why do we only need one int to represent index?
• In these slides, we write GMV for GraphMatrixVertex
Choosing a Dictionary Structure

• We need a structure that will let us retrieve the index of a vertex given the vertex label (a dictionary)

• Many choices
  • Vector of associations:
    • Vector<Association<V, GraphMatrixVertex<V>>>
  • Ordered Vector of Associations
  • BinarySearchTree of Associations

• Problem: We don’t want to allow multiple vertices with same label.... [Why?]

• We’ll use the Map Interface [Chapter 15]
  • Maps require a unique key for each entry
Digression: Map Interface

• Methods for Map<K, VAL>
  • int size() - returns number of entries in map
  • boolean isEmpty() - true iff there are no entries
  • boolean containsKey(K key) - true iff key exists in map
  • boolean containsValue(VAL val) - true iff val exists at least once in map
  • VAL get(K key) - get value associated with key
  • VAL put(K key, VAL val) - insert mapping from key to val, returns value replaced (old value) or null
  • VAL remove(K key) - remove mapping from key to val
  • void clear() - remove all entries from map
• We’ll study this more in a week or so....
Implementing the Matrix Model

• Abstract class – partially implements Graph
  
  public abstract class GraphMatrix<V,E> implements Graph<V,E>

• This class will implement features common to directed and undirected graphs

• Instance variables
  
  protected int size; //max size of matrix
  protected Object data[][][]; //matrix of edges
  protected Map<V, GMV<V>> dict; //labels -> vertices
  // This is structure5.Map, NOT java.util.Map!
  protected List<Integer> freeList; //avail indices
  protected boolean directed;
GraphMatrix Constructor
(Yes, abstract classes can have constructors!)

protected GraphMatrix(int size, boolean dir) {
    this.size = size; // set maximum size
directed = dir; // fix direction of edges

    // the following constructs a size x size matrix
    // (the “Objects” will be “Edges”)
    // (can’t use generics with arrays!)
data = new Object[size][size];

    // label→index translation table
dict = new Hashtable<V,GraphMatrixVertex<V>>(size);

    // put all indices in the free list
    freeList = new SinglyLinkedList<Integer>();
    for (int row = size-1; row >= 0; row--)
        freeList.add(new Integer(row));
}
public void add(V label) {
    // if there already, do nothing
    if (dict.containsKey(label)) return;

    Assert.pre(!freeList.isEmpty(), "Matrix not full");
    // allocate a free row and column
    int row = freeList.removeFirst().intValue();
    // add vertex to dictionary
    dict.put(label, new GraphMatrixVertex<V>(label, row));
}
public V remove(V label) {
    // find and extract vertex
    GraphMatrixVertex<V> vert;
    vert = dict.remove(label);
    if (vert == null) return null;
    // remove vertex from matrix
    int index = vert.index();
    // clear row and column entries
    for (int row=0; row<size; row++) {
        data[row][index] = null;
        data[index][row] = null;
    }
    // add node index to free list
    freeList.add(new Integer(index));
    return vert.label();
}
neighbors Iterator

public Iterator<V> neighbors(V label) {
    GraphMatrixVertex<V> vert = dict.get(label);
    List<V> list = new SinglyLinkedList<V>();
    for (int row=size-1; row>=0; row--) {
        Edge<V,E> e = (Edge<V,E>)data[vert.index()][row];
        if (e != null) {
            if (e.here().equals(vert.label())) {
                list.add(e.there());
            } else list.add(e.here());
        }
    }
    return list.iterator();
}
GraphMatrixDirected

• Completes the implementation of GraphMatrix to ensure graph is directed
• GraphMatrixUndirected is very similar…
• How do we implement GraphMatrixDirected?
  • We’ll discuss some methods
  • Read Ch 16 for complete details…
GraphMatrixDirected

• Constructor

public GraphMatrixDirected(int size) {
    // pre: size > 0
    // post: constructs an empty graph that may be expanded to at most size vertices. Graph is directed if dir true and undirected otherwise
    // call GraphMatrix constructor
    super(size,true);
}
GraphMatrixDirected

- **addEdge**
  
  // pre: vLabel1 and vLabel2 are labels of existing vertices
  
  public void addEdge(V vLabel1, V vLabel2, E label) {
      GraphMatrixVertex<V> vtx1,vtx2;
      vtx1 = dict.get(vLabel1);
      vtx2 = dict.get(vLabel2);
      Edge<V,E> e = new Edge<V,E>(vtx1.label(), vtx2.label(),
                                  label, true);
      data[vtx1.index()][vtx2.index()] = e;
  }
GraphMatrixDirected

• removeEdge

  // pre: vLabel1 and vLabel2 are labels of existing vertices
  public E removeEdge(V vLabel1, V vLabel2) {
      // get indices
      int row = dict.get(vLabel1).index();
      int col = dict.get(vLabel2).index();
      // cache old value
      Edge<V,E> e = (Edge<V,E>)data[row][col];
      // update matrix
      data[row][col] = null;
      if (e == null) return null;
      else return e.label(); // return old value
  }
GraphMatrix Efficiency

- Assume Map operations are $O(1)$ (for now)
  - $|E|$ = number of edges
  - $|V|$ = number of vertices
- Runtime of add, addEdge, getEdge, removeEdge, remove?
- Space usage?
- Conclusions
  - Matrix is good for dense graphs
  - Have to commit to maximum # of vertices in advance
### Efficiency: Assuming Fast Map

<table>
<thead>
<tr>
<th>Operation</th>
<th>GraphMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>addEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>getEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeEdge</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(</td>
</tr>
<tr>
<td>space</td>
<td>$O(</td>
</tr>
</tbody>
</table>