Last Time

- Graph Algorithms
  - Reachability: BFS
  - MCST: Kruskal and Prim
  - Single Source Shortest Path: Dijkstra
Today’s Outline

• Graph Implementation Details
  • Adjacency Matrix
  • Adjacency List

• Time/Space Complexity

• Last Graph day, so may go fast! Focus on the high level ideas and the *tradeoffs*
Graph Classes in structure5

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

**Structure**

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

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

- **Edge**
### Adjacency Array: Directed Graph

The table below represents the adjacency array of a 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>

Challenges to having our rows/columns be “vertices”:

- Can’t use Objects as array indices
- How does adding/deleting a vertex work?!
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 adds one more useful attribute to
  the abstract Vertex class:
  • Index of node (int) in adjacency matrix
    int index()
  • Why do we only need one int to represent index?
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>>>
  - OrderedVector 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

- Maps *unique* keys to values (*V* is value not vertex!!!)
- Methods for Map<*K*, *V*>
  - *int size()* - returns number of entries in map
  - *boolean isEmpty()* - true iff there are no entries
  - *boolean containsKey(*K* key)* - true iff key present
  - *boolean containsValue(*V* val)* - true iff val exists at least once in map
  - *V get(*K* key)* - get value associated with key
  - *V put(*K* key, *V* val)* - insert mapping from key to val, returns value replaced (old value) or null
  - *V remove(*K* key)* - remove mapping from key to val
  - *void clear()* - remove all entries from map
- We’ll study this more on Monday....
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 boolean directed;
GraphMatrix Constructor
(Yes, abstract classes can have constructors!)

```java
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 = 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: Make a list and return its iterator

```java
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, 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)
  - $|E| = \text{number of edges}$
  - $|V| = \text{number of vertices}$
- 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></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>
Graph Classes in structure5

Graph

- GraphMatrix
  - GraphMatrixDirected
  - GraphMatrixUndirected

- GraphList
  - GraphListDirected
  - GraphListUndirected

Interface

Abstract Class

Class

Structure

Vertex

- GraphMatrixVertex
- GraphListVertex

Edge
Adjacency List : Directed Graph

The vertices are stored in an array $V[]$
$V[i]$ contains a linked list of all edges with a given source
The vertices are stored in an array $V[]$
$V[i]$ contains a linked list of all edges incident to a given vertex
GraphList: Big Picture

- Maintain an *adjacency list* of *edges* at each vertex (no adjacency matrix)
  - Keep only outgoing edges for directed graphs
- Support both directed and undirected graphs (GraphListDirected, GraphListUndirected)
**Vertex and GraphListVertex**

- We use the same `Edge` class for all graph types
- We extend `Vertex` to include an `Edge` list
- `GraphListVertex` class adds to `Vertex` class:
  - A Structure to store edges adjacent to the vertex
    ```java
    protected Structure<Edge<V,E>> adjacencies; // adjacent edges
    - adjacencies is created as a SinglyLinkedList of edges
    ```
  - Several methods
    ```java
    public void addEdge(Edge<V,E> e)
    public boolean containsEdge(Edge<V,E> e)
    public Edge<V,E> removeEdge(Edge<V,E> e)
    public Edge<V,E> getEdge(Edge<V,E> e)
    public int degree()
    // and methods to produce Iterators...
    ```
public GraphListVertex(V label) {
    super(label); // init Vertex fields
    adjacencies = new SinglyLinkedList<Edge<V,E>>();
}

public boolean containsEdge(Edge<V,E> e) {
    return adjacencies.contains(e);
}

public void addEdge(Edge<V,E> e) {
    if (!containsEdge(e)) adjacencies.add(e);
}

public Edge<V,E> removeEdge(Edge<V,E> e) {
    return adjacencies.remove(e);
}
GraphListVertex Iterators

// Iterator for incident edges
public Iterator<Edge<V,E>> adjacentEdges() {
    return adjacencies.iterator();
}

// Iterator for adjacent vertices
public Iterator<V> adjacentVertices() {
    return new GraphListAIterator<V,E>(
        adjacentEdges(), label());
}

GraphListAIterator creates an Iterator over vertices based on
the Iterator over edges produced by adjacentEdges()

(Details in the book and on posted slides)
GraphList (Abstract base class)

• To implement GraphList, what data structures do we need?
  • (Maintain an adjacency list of edges at each vertex)

• GraphListVertex class
  • Instance vars: label, visited flag, linked list of edges

• “Array V[]” of GraphListVertex
  • I Lied! We actually use a Map from V to GraphListVertex:
    Map<V,GraphListVertex<V,E>> dict; // label -> vertex

• Do we need a free list like GraphMatrix?
• Do we need to know |V| ahead of time?
protected Map<V,GraphListVertex<V,E>> dict;
protected boolean directed;

protected GraphList(boolean dir){
    dict = new Hashtable<V,GraphListVertex<V,E>>(());
    directed = dir;
}

public void add(V label) {
    if (dict.containsKey(label)) return;
    GraphListVertex<V,E> v = new GraphListVertex<V,E>(label);
    dict.put(label,v);
}
public Edge<V,E> getEdge(V label1, V label2) {
    Edge<V,E> e = new Edge<V,E>(get(label1),
                                get(label2), null, directed);
    return dict.get(label1).getEdge(e);
}

(in GraphListVertex)

public Edge<V,E> getEdge(Edge<V,E> e) {
    Iterator<Edge<V,E>> edges = adjacencies.iterator();
    while (edges.hasNext()) {
        Edge<V,E> adjE = edges.next();
        if (e.equals(adjE))
            return adjE;
    }
    return null;
}
GraphListDirected

- GraphListDirected (GraphListUndirected) implements the methods requiring different treatment due to (un)directedness of edges
  - addEdge, remove, removeEdge, …

- (We will only look at GraphListDirected in class)
// addEdge in GraphListDirected.java
// first vertex is source, second is destination
public void addEdge(V vLabel1, V vLabel2, E label) {
    // first get the vertices
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);
    // create the new edge
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), label, true);
    // add edge only to source vertex linked list (aka adjacency list)
    v1.addEdge(e);
}
public V remove(V label) {
  //Get vertex out of map/dictionary
  GraphListVertex<V,E> v = dict.get(label);

  //Iterate over all vertex labels (called the map “keyset”)
  Iterator<V> vi = iterator();
  while (vi.hasNext()) {
    //Get next vertex label in iterator
    V v2 = vi.next();

    //Skip over the vertex label we're removing
    //(Nodes don't have edges to themselves...)
    if (!label.equals(v2)) {
      //Remove all edges to "label"
      //If edge does not exist, removeEdge returns null
      removeEdge(v2,label);
    }
  }
  //Remove vertex from map
  dict.remove(label);
  return v.label();
}
public E removeEdge(V vLabel1, V vLabel2) {
    //Get vertices out of map
    GraphListVertex<V,E> v1 = dict.get(vLabel1);
    GraphListVertex<V,E> v2 = dict.get(vLabel2);

    //Create a “temporary” edge connecting two vertices
    Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), null, true);

    //Remove edge from source vertex linked list
    e = v1.removeEdge(e);
    if (e == null) return null;
    else return e.label();
}
Efficiency Revisited

• 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 better for dense graphs
  • List is better for sparse graphs
  • For graphs “in the middle” there is no clear winner
### Efficiency: Assuming Fast Map

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