CSE 373: Data Structures and Algorithms

Lecture 20: Graphs II
Implementing a graph

• If we wanted to program an actual data structure to represent a graph, what information would we need to store?
  – for each vertex?
  – for each edge?

• What kinds of questions would we want to be able to answer quickly:
  – about a vertex?
  – about its edges / neighbors?
  – about paths?
  – about what edges exist in the graph?

• We'll explore three common graph implementation strategies:
  – edge list, adjacency list, adjacency matrix
Edge list

• **edge list**: an unordered list of all edges in the graph

• **advantages**
  – easy to loop/iterate over all edges

• **disadvantages**
  – hard to tell if an edge exists from A to B
  – hard to tell how many edges a vertex touches (its degree)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Adjacency matrix

- **adjacency matrix**: an $n \times n$ matrix where:
  - the nondiagonal entry $a_{ij}$ is the number of edges joining vertex $i$ and vertex $j$ (or the weight of the edge joining vertex $i$ and vertex $j$)
  - the diagonal entry $a_{ii}$ corresponds to the number of loops (self-connecting edges) at vertex $i$
Pros/cons of Adj. matrix

- **advantage**: fast to tell whether edge exists between any two vertices $i$ and $j$ (and to get its weight)
- **disadvantage**: consumes a lot of memory on sparse graphs (ones with few edges)
Adjacency matrix example

• The graph at right has the following adjacency matrix:
  – How do we figure out the degree of a given vertex?
  – How do we find out whether an edge exists from A to B?
  – How could we look for loops in the graph?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</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>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency lists

• **adjacency list**: stores edges as individual linked lists of references to each vertex's neighbors
  
  – generally, no information needs to be stored in the edges, only in nodes, these arrays can simply be pointers to other nodes and thus represent edges with little memory requirement

![Adjacency list diagram]
Pros/cons of adjacency list

- **advantage**: new nodes can be added to the graph easily, and they can be connected with existing nodes simply by adding elements to the appropriate arrays; "who are my neighbors" easily answered

- **disadvantage**: determining whether an edge exists between two nodes requires $O(n)$ time, where $n$ is the average number of incident edges per node

![Diagram of adjacency list]

- Each node is represented by a box with its value and the number of incident edges in parentheses.
- Arrows represent edges pointing from the source node to the target node.
- The diagram shows a simple graph with nodes 0 to 6.
### Adjacency list example

- The graph at right has the following adjacency list:
  - How do we figure out the degree of a given vertex?
  - How do we find out whether an edge exists from A to B?
  - How could we look for loops in the graph?

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacent Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 5</td>
</tr>
<tr>
<td>2</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3, 7, 5</td>
</tr>
<tr>
<td>5</td>
<td>6, 7</td>
</tr>
<tr>
<td>6</td>
<td>1, 7</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

![Graph visualization](attachment:image.png)
# Runtime table

- **n** vertices, **m** edges
- no parallel edges
- no self-loops

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>n</strong> + <strong>m</strong></td>
<td><strong>n</strong> + <strong>m</strong></td>
<td><strong>n</strong>^2</td>
<td></td>
</tr>
<tr>
<td>Finding all adjacent vertices to ( v )</td>
<td><strong>m</strong></td>
<td>\text{deg}(v)</td>
<td><strong>n</strong></td>
</tr>
<tr>
<td>Determining if ( v ) is adjacent to ( w )</td>
<td><strong>m</strong></td>
<td>\text{deg}(v)</td>
<td>1</td>
</tr>
<tr>
<td>inserting a vertex</td>
<td>1</td>
<td>1</td>
<td><strong>n</strong>^2</td>
</tr>
<tr>
<td>inserting an edge</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removing vertex ( v )</td>
<td><strong>m</strong></td>
<td>1</td>
<td><strong>n</strong>^2</td>
</tr>
<tr>
<td>removing an edge</td>
<td><strong>m</strong></td>
<td>\text{deg}(v)</td>
<td>1</td>
</tr>
</tbody>
</table>
Practical implementation

- Not all graphs have vertices/edges that are easily "numbered"
  - how do we actually represent 'lists' or 'matrices' of vertex/edge relationships? How do we quickly look up the edges and/or vertices adjacent to a given vertex?
    - Adjacency list: Map<V, List<V>>
    - Adjacency matrix: Map<V, Map<V, E>>
Maps and sets within graphs

since not all vertices can be numbered, we can use:

1. adjacency list
   - each Vertex maps to a List of edges
   - Vertex --> List of Edges
   - to get all edges adjacent to $V_1$, look up
     $List<Edge> neighbors = map.get(V_1)$

2. adjacency map (adjacency matrix for objects)
   - each Vertex maps to a hashtable of adjacent vertices
   - Vertex --> (Vertex --> Edge)
   - to find out whether there's an edge from $V_1$ to $V_2$, call
     $map.get(V1).containsKey(V2)$
   - to get the edge from $V_1$ to $V_2$, call $map.get(V1).get(V2)$
Implementing Graph with Adjacency List

public interface Graph<V> {
    public void addVertex(V v);

    public void addEdge(V v1, V v2, int weight);

    public boolean hasEdge(V v1, V v2);

    public Edge<V> getEdge(V v1, V v2);

    public boolean hasPath(V v1, V v2);

    public List<V> getDFSPath(V v1, V v2);

    public String toString();
}

public class Edge<V> {  
    public V from, to;  
    public int weight;  

    public Edge(V from, V to, int weight) {  
        if (from == null || to == null) {  
            throw new IllegalArgumentException("null");  
        }  
        this.from = from;  
        this.to = to;  
        this.weight = weight;  
    }  

    public String toString() {  
        return "<" + from + ", " + to + ", " + weight + ">";  
    }  
}