CS171 Introduction to Computer Science II

Graphs
Graphs

• Examples
• Definitions
• Implementation/Representation of graphs
Graphs

• Graphs: set of vertices connected pairwise by edges
• Interesting and useful structure
• Many practical applications
  – Maps
  – Web content
  – Schedules
  – Social networks
  – ...
Delta Airlines Domestic Routes

Delta Airlines domestic routes

From Atlanta

From Memphis

4/12/2012
Bow Tie Theory

disconnected pages
origination 24%
core 30%
24% termination
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.

Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.
Course Prerequisite Graph

Mathematics Major Requirements

Notes
- Underlined courses are required only for the BS.
- BA requires 332 or 321.
- 210 is required for 331 but not 337.

123 Calculus I
110 or 111 Foundations of Computing

124 Calculus II

210 Proof Techniques

231 Linear Algebra & Diff Eq

332 Abstract Algebra I
321 Advanced Analysis I
331 Combinatorics OR 337 Operations Research
329 Complex Analysis OR 357 Methods of Applied Math
232 Mathematical Modeling

Electives (2)
242 329 331 334 337
341 357 400
Graphs

• Undirected graphs
  – simple connections

• Digraphs
  – each connection has a direction

• Edge-weighted graphs
  – each connection has an associated weight

• Edge-weighted digraphs
  – each connection has both a direction and a weight
Undirected Graphs

• A graph is a set of vertices and a collection of edges that each connect a pair of vertices
Graph representation

Graph drawing. Provides intuition about the structure of the graph.

Caveat. Intuition can be misleading.

two drawings of the same graph
Glossary

• When there is an edge connecting two vertices, the vertices are **adjacent to** one another and the edge is **incident to** both vertices
• A **self-loop** is an edge that connects a vertex to itself
• Two edges that connect the same pair of vertices are **parallel**
• The **degree** of a vertex is the number of edges incident to the vertex, with loops counted twice
• A **subgraph** is a subset of a graph’s edges (and associated vertices) that constitutes a graph
Glossary

• A path in a graph is a sequence of vertices connected by edges
  – A simple path is one with no repeated vertices
  – A cycle is a path with at least one edge whose first and last vertices are the same
  – A simple cycle is a cycle with no repeated edges or vertices (except the first and last vertices)
  – The length of a path is its number of edges
• One vertex is connected to another if there exists a path that contains both of them
• A graph is connected if there is a path from every vertex to every other vertex in the graph
  – A graph that is not connected consists of a set of connected components
• An acyclic graph is a graph with no cycles.
Graphs

• Examples
• Definitions
• Implementation/Representation of graphs
### Graph API

**public class Graph**

- **Graph(int V)**  
  create an empty graph with V vertices
- **Graph(In in)**  
  create a graph from input stream
- **void addEdge(int v, int w)**  
  add an edge v-w
- **Iterable<Integer> adj(int v)**  
  vertices adjacent to v
- **int V()**  
  number of vertices
- **int E()**  
  number of edges
- **String toString()**  
  string representation

```java
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
```

- read graph from input stream
- print out each edge (twice)
How to represent/implement a graph?

• Space-efficient
  – Accommodate types of graphs that likely to encounter

• Time-efficient
  – Add an edge
  – If there is edge between v and w
  – Iterate over vertices adjacent to v
  – ...

Real-world graphs

- Real-world graphs tend to be “sparse”
  - Huge number of vertices, small average vertex

Two graphs (V = 50)
Representation Options

• Edge list
• Adjacency matrix
• Adjacency lists
Set-of-edges graph representation

Maintain a list of the edges (linked list or array).
Adjacency-matrix graph representation

Maintain a two-dimensional $V$-by-$V$ boolean array; for each edge $v \rightarrow w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}.$
Adjacency-list graph representation

Maintain vertex-indexed array of lists.
Graph representations

In practice. Use adjacency-lists representation.
- Algorithms based on iterating over vertices adjacent to \( v \).
- Real-world graphs tend to be “sparse.”

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between ( v ) and ( w )?</th>
<th>iterate over vertices adjacent to ( v )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>1</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>1 *</td>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>degree(( v ))</td>
<td>degree(( v ))</td>
</tr>
</tbody>
</table>

* disallows parallel edges
Adjacency-list graph representation: Java implementation

```java
public class Graph {
    private final int V;
    private Bag<Integer>[][] adj;

    public Graph(int V) {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
Bag API

**Main application.** Adding items to a collection and iterating (when order doesn’t matter).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Bag()</code></td>
<td>create an empty bag</td>
</tr>
<tr>
<td><code>void add(Item x)</code></td>
<td>insert a new item onto bag</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of items in bag</td>
</tr>
<tr>
<td><code>Iterable&lt;Item&gt; iterator()</code></td>
<td>iterator for all items in bag</td>
</tr>
</tbody>
</table>

Implementation. Stack (without pop) or queue (without dequeue).
Full implementation

- http://algs4.cs.princeton.edu/41undirected/Graph.java.html
Graph API: sample client

Graph input format.

In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + " - " + w);

% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
...
12-11
12-9

read graph from input stream
print out each edge (twice)
### Typical graph-processing code

<table>
<thead>
<tr>
<th>Method</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute the degree of $v$</td>
<td>public static int degree(Graph $G$, int $v$)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int degree = 0;</td>
</tr>
<tr>
<td></td>
<td>for (int $w : G.adj(v)) degree++;</td>
</tr>
<tr>
<td></td>
<td>return degree;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>compute maximum degree</td>
<td>public static int maxDegree(Graph $G$)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int max = 0;</td>
</tr>
<tr>
<td></td>
<td>for (int $v = 0; $v &lt; G.V(); $v++)</td>
</tr>
<tr>
<td></td>
<td>if (degree($G, $v) &gt; max)</td>
</tr>
<tr>
<td></td>
<td>max = degree($G, $v);</td>
</tr>
<tr>
<td></td>
<td>return max;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>compute average degree</td>
<td>public static int avgDegree(Graph $G$)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>return 2 * G.E() / G.V();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>count self-loops</td>
<td>public static int numberOfSelfLoops(Graph $G$)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int count = 0;</td>
</tr>
<tr>
<td></td>
<td>for (int $v = 0; $v &lt; G.V(); $v++)</td>
</tr>
<tr>
<td></td>
<td>for (int $w : G.adj($v))</td>
</tr>
<tr>
<td></td>
<td>if ($v == $w) count++;</td>
</tr>
<tr>
<td></td>
<td>return count/2;</td>
</tr>
</tbody>
</table>