1. **Motivation**

1.1 **Up To Today:**
- graph theory, which leads to …
- implicit graphs and help with homework, then…
- explicit graphs to help build generic graph classes…
- build graphs

1.2 **What To Do With Graphs?**
- next two lectures…
- generalize traversal: BFS, DFS
- use traversal for searching
- sorting
- shortest path to something
- more…?

2. **Representations**

2.1 **Implicit**
- rules/model creates a network of nodes/edges
- ex) puzzle moves
  - each move makes a new puzzle
  - treat each state as a node
  - so, rules implicit define a graph
- common for games!

2.2 **Explicit**
- define all nodes $V$ and edges $E$ ahead of time
- want system to represent edges
- why? it’s the “biggest problem”:
  - $G = (V,E)$ and each edge $e$ in $E$ is a pair $(v1,v2)$
  - most edges possible? $|V|^2$
    (form pairs from all nodes)
  - most sets of edges possible? $2^{|V|^2}$
- so, use container to represent edges
  - adjacency matrix
  - adjacency list
2.3 Adjacency Matrix

- **adjacency matrix**
  \[ A_{ij} = \begin{cases} w_{ij} & \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases} \]

- terms
  \( v_i \): node i; \( v_j \) node j
  \( \{v_i, v_j\} \in E \): edge between nodes i (\( v_i \)) and j (\( v_j \)) belongs to set of edges \( E \)
  \( w_{ij} \): weight of edge between nodes i and j

- \( A_{ij} \): the matrix (rectangular 2x2 array) as rows (i) and cols (j); coords correspond to nodes i and j

2.4 Adjacency List

- adjacency list: linked list of nodes adjacent to a node
- need \( |V| \) lists

2.5 graph types to develop:

- undirected
- directed
- weighted
2.8 Weighted

- assuming also weighted
- \( w_{ij} \): cost or weight of edge from node i to node j
- sometimes use sentinel to represent “no edge” between i and j

\[
A_{ij} = \begin{cases} 
    w_{ij} & (v_i, v_j) \in E \\
    0 & \text{otherwise}
\end{cases}
\]

Use array \( A \) of lists: include weights
List for i contains j, w for edge \((i, j)\)

![Graph Representation]

2.9 Choice of AM or AL?

- Adjacency Matrix
  - uses \( O(|V|^2) \) space
  - can answer “is there an edge from i to j?” in \( O(1) \) time
  - enumerating all nodes adjacent to i: \( O(|V|) \) (find all nodes j in row i)
  - could be sparse because of wasted space (0s)
  - better for dense graphs (lots of edges)!

- Adjacency List
  - uses \( O(|V| + |E|) \) space (\( |V| \) for i nodes, \( |E| \) for j nodes emanating from each i node)
  - can answer question “is there an edge from i to j?” in \( O(|E|) \) time
  - enumerating all nodes adjacent to i: \( O(1) \) per adjacent node in linked list
  - better for sparse graphs (few edges)!

3. Implementation

3.1 Implicit

- can use containers to store node and edge info
- a bit too problem specific, though effective

3.2 Explicit

- Adjacency Matrix - left as exercise
- Adjacency List
  - using linked list to allow for flexible building
  - kind of gives implicit building by allowing for node/edge creation “on the fly”
- focus on digraph, but could be weighted
  - Sections 3, 4, 5, 6
  - many methods left out – will see for graph problems

4. Verticies

4.1 Fields

- \textbf{label}: we like to have names, numbers, …
- \textbf{edges}: collection of all emanating edges from the current vertex
- \textbf{visited}: need later to tag vertex for searching…
- sometimes includes \textbf{cost} (cost to get \textit{here} from somewhere)

4.2 Constructor

- set \textbf{label}
- create \textbf{edges} adjacency list (AL)

4.3 Methods

- \textbf{addEdge}: add to AL
- \textbf{equals}: need for path checking
- more?
import java.util.*;

public class Vertex {
    private Object label;
    private LinkedList edges; // adjacent edges
    private boolean visited; // tag

    public Vertex(Object o) {
        label = o;
        edges = new LinkedList();
    }

    public void addEdge(Edge e, int weight) {
        Vertex source = this;
        Vertex dest = e.getDest();
        edges.add(new Edge(source, dest, weight));
    }

    public void addEdge(Edge e) {
        addEdge(e, 0);
    }

    public boolean equals(Vertex other) {
        return label.equals( ((Vertex)other).label );
    }

    public String toString() {
        return label.toString();
    }

    public Collection getEdges() { return edges; }
} // Class Vertex

5. Edges
5.1 Fields
• source: s->d, the node from which edge emanates
• dest: actually, all you need is this since Vertex keeps track of adjacent edges of source
• weight: could make double (sometimes called cost)

5.2 Constructors
• build edge from s->d
• can default to weight of 0 to handle unweighted graphs

5.3 Methods
• equals and compareTo:
  - many algorithms want to know shortest path
  - need to compare costs of going in different directions
• toString: "source-weight->dest"
• more?

6. Directed Graphs
6.1 Fields
• vertices dictionary:
  - key-val pairs of (VertexName,Vertex)
  - each Vertex points to its adjacency list!
• edgeCount

6.2 Constructors
• set vertices to LinkedHashMap
• maintains order of nodes in order created
• nodes must be created before edges this way!

6.3 Methods
• use vertex names/labels!
• addVertex: put Vertex in Map: (name, Vertex)
• addEdge: connect s and d nodes (they must exist!)
import java.util.*;
public class Digraph {

  private Map verticies; // dictionary of nodes
  private int edgeCount; // number of edges

  public Digraph() {
    verticies = new LinkedHashMap();
  }

  // Add vertex to map
  public void addVertex(Object name) {
    verticies.put(name, new Vertex(name));
  }

  // Adds edge (source and dest node must exist!):
  public void addEdge(Object s, Object d, int weight) {
    // Key is NAME of Vertex
    // Val is THE Vertex
    // So, get keys of s and d and use them to
    // retrieve their vals (their Verticies):
    Vertex source = (Vertex)verticies.get(s);
    Vertex dest = (Vertex)verticies.get(d);

    // Create edge between source and dest:
    s.addEdge(new Edge(source, dest, weight));
    edgeCount++;
  }

  public void addEdge(Object source, Object dest) {
    addEdge(source, dest, 0);
  }

  public String toString() {
    String s = "";
    Iterator it = verticies.keySet().iterator();
    while (it.hasNext()) {
      // current node label:
      Object key = it.next();

      // current Vertex:
      Vertex val = (Vertex) verticies.get(key);

      // build string for current vertex in Map:
      s += "[" + val + "]" + "-->" + "\n";
    }
    return s;
  }
}

7. Demonstration

7.1 Code

public class TestDigraph {
  public static void main(String[] args) {
    Digraph g = new Digraph();
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addEdge("A", "B");
    g.addEdge("A", "C");
    g.addEdge("B", "C");
    System.out.println(g);
  }
}

7.2 Output

[A] --> [A-0->B], [A-0->C]
[B] --> [B-0->C]
[C] --> []

8. Exercises

- Demonstrate why we use edges for explicit representations of graphs.
- Develop Vertex, Edge, Digraph, and TestDigraph classes for the adjacency matrix approach. You should develop methods to handle I/O in reading in a grid of adjacencies to help build a graph.
- Remove the source node field from class Edge and modify the remaining classes as necessary. This design is a bit more common than the examples given to you.
- Rewrite Digraph’s addEdge such that it does not assume that the nodes exist. You may either throw an exception or perhaps create more nodes….
- Graphical graph: This was once a final project long ago…develop a GUI tool that draws a graph that a user creates, either via the GUI or as a translation from the collection that contains the vertices and edges. A rudimentary application would naively draw each vertex according to a pre-determined grid and then draw the edges using the given vertex geometry.