CMPSCI 187: Programming With Data Structures

Lecture #31: Graphs
David Mix Barrington
26 November 2012

Graphs

- Graph Vocabulary
- Applications of Graphs
- Paths in Graphs
- The Weighted Graph Interface
- Implementing Graphs With Arrays
- Implementing Graphs With Linked Lists

Graph Vocabulary

- Linear lists and trees are two ways to make objects out of **nodes** and **connections** from one node to another. There are many other ways to do it.
- A graph consists of a set of nodes called **vertices** (one of them is a **vertex**) and a set of **edges** that connect the nodes. In an **undirected graph**, each edge connects two distinct vertices. In a **directed graph**, each edge goes *from* one vertex *to* another vertex.
- Two vertices are **adjacent** if there is an undirected edge from one to the other.
- A **path** (in either kind of graph) is a sequence of edges where each edge goes to the vertex that the next edge comes from. A **simple path** is one that never reuses a vertex (DJW blur the distinction). In a tree, there is exactly one simple path from any one vertex to any other vertex.
- A **complete graph** is one with every possible edge among its vertices.

Applications of Graphs

- Any situation where there are entities and **binary connection relationships** can be described with a graph.

- **Transportation networks** consist of points and connections from one point to another. We often want to know whether these connections can be formed into paths, which is asking whether particular paths exist in the corresponding graph. We also often attach **weights** to the edges, representing distance or cost. A natural problem is then to find the **shortest path** from one vertex to another.

- We might use a vertex to represent every species of animal in some ecosystem, and place an edge between every pair of vertices representing species that compete for resources. Paths in this graph would then represent more complicated relationships among the species.

Paths in Graphs

- We can give a recursive definition of paths in a graph. There is a 0-step path from any vertex to itself. If α is an i-step path from vertex x to vertex y, and e is an edge from y to some vertex z, then there is an (i+1)-step path β from x to z, given by appending the edge e to the path α.

- This suggests a recursive algorithm for exploring all paths out of some original vertex x -- explore the 0-edge path, then recursively explore all the paths out of all the neighbors (adjacent vertices) of x.

- This should strike you as familiar -- we've used this technique to mark all the squares on a continent in Projects 3 and 4. A continent is a **connected component** of the graph where vertices are land squares and there are edges between the vertices for squares that are adjacent (not diagonally). A connected component of an undirected graph is the set of all vertices that have a path to some particular vertex.

The Weighted Graph Interface

- Here's an interface that represents a directed graph with integer edge weights. The vertices are T objects, which we can add to the graph as we wish. We add an edges by giving its from and to vertices and its weight. We can mark vertices, get an unmarked vertex if one exists, and get a queue containing the vertices adjacent from any given vertex. Note that vertices may be equal according to T's equals method.

```
public interface WeightedGraphInterface<T> {
   boolean isEmpty( );
   boolean isFull( );
   void addVertex(T vertex);
   boolean hasVertex(T vertex);
   void addEdge(T fromVertex, T toVertex, int weight);
   int weightIs(T fromVertex, T toVertex);
   UnboundedQueueInterface<T> getToVertices(T vertex);
   void clearMarks( );
   void markVertex(T vertex);
   boolean isMarked(T vertex);
   T getUnmarked( );
```

Implementing Graphs With Arrays

- Here are the field declarations and constructors for an array-based implementation of our weighted graph interface. Note the casting business. As usual, I've saved some code length over DJW by having the zero-parameter constructor call the one-parameter constructor.

```
public class WeightedGraph<T>
             implements WeightedGraphInterface<T> {
   public static final int NULL_EDGE = 0;
   public static final int DEFCAP = 50;
   private int numVertices, maxVertices;
   private T[ ] vertices;
   private int[ ][ ] edges;
   private boolean[ ] marks;
   public WeightedGraph(int maxV) {
      numVertices = 0; maxVertices = maxV;
      vertices = (T[ ]) new Object[maxV];
      marks = new boolean[maxV];
      edges = new int[maxV][maxV];}
   public WeightedGraph( ) {this(DEFCAP);}
```

Some Methods of `WeightedGraph`

```
public void addVertex(T vertex) {
   vertices(numVertices) = vertex;
   for (int index = 0; index < numVertices; index++) {
      edges[numVertices][index] = NULL_EDGE;
      edges[index][numVertices] = NULL_EDGE;}
   numVertices++;}

private int indexIs(T vertex) {
   int index = 0;
   while (!vertex.equals(vertices[index])) index++;
   return index;}

public void addEdge(T fromVertex, T toVertex, int weight) {
   int row = indexIs(fromVertex), column = IndexIs(toVertex);
   edges[row][column] = weight;}

public int weightIs(T fromVertex, T toVertex) {
   int row = indexIs(fromVertex), column = IndexIs(toVertex);
   return edges[row][column];}
```

Building a Queue of Neighbors

- The other methods of `WeightedGraphInterface` are fairly simple to implement. Here we look at the method that takes a vertex and gives back a queue containing its neighbors. Remember that when we add a new vertex we set all the edge weights to and from it to `NULL_EDGE`. We assume that this special value may not be the weight of an actual edge, so that we can cycle through all possible edges out of `vertex` and enqueue the vertices corresponding to real edges.

```
public UnboundedQueueInterface<T> getToVertices(T vertex) {
   UnboundedQueueInterface<T> adjVertices =
                                        new LinkedUndndQueue<T>( );
   int fromIndex = indexIs(vertex), toIndex;
   for (toIndex = 0; toIndex < numVertices; toIndex++)
      if (edges[fromIndex][toIndex] != NULL_EDGE)
         adjVertices.enqueue(vertices[toIndex]);
   return adjVertices;}
```

Implementing Graphs With Linked Lists

- The array-based implementation, called an **adjacency matrix**, is the simplest way to represent graphs on a computer. But on a graph with n vertices, it always takes $O(n^2)$ memory locations, one for each entry in the edge array.

- Graphs in applications are usually **sparse**, meaning that they have relatively few neighbors for each vertex. A sparse graph of n vertices might have only O(n) edges, and its matrix would be mostly null edges. Another representation called an **adjacency list** saves space, and can save time if the algorithms are designed to use it efficiently.

- An adjacency list has an array of linked lists of vertices, one for each vertex. The list for vertex v has all of v's neighbors, so there is only one entry for each edge of a directed graph, or two for each edge of an undirected graph. We can implement the methods of the weighted graph interface fairly easily. For example, the queue of neighbors just takes its entries from the list for the given vertex.