CmpSci 187: Programming with Data Structures Spring 2015

Lecture #20, Introduction to Graphs

John Ridgway

April 14, 2015

1 Introduction to Graphs

Graphs and their 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** (or **di-graph**), each edge goes from one vertex to another.
- 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 leaves the vertex that the last edge led to. 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 vertex to any other vertex.
- A complete graph is one with every possible edge among its vertices.

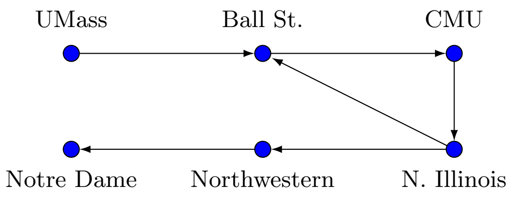
Applications of Graphs

- 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, e.g., can we get from Puffer's Pond to Northampton?

- We also often attach **weights** to the edges, representing distance or cost.
- A natural problem is to find the shortest (or cheapest) 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.

A Directed Graph Example

Consider a graph where the vertices represent BCS college football teams, and there is an edge from x to y if team x defeated team y during the 2014 season. Only six of the 128 teams are shown.



Clicker Question #1

In the above graph, what does the number of edges into the vertex for team x represent? (Teams do not play one another more than once.)

A. x's number of wins

B. **x's number of losses**

C. x's total games

D. x's wins minus losses

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 used this technique to mark all the squares on a continent in a square grid.
- 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.

2 Implementing the Graph Abstraction

`UnweightedGraphInterface`

- Here's an interface that represents a directed graph. The vertices are `V` objects, which we can add to the graph as we wish. We add an edge by giving its from and to vertices.
- Note that a vertex may not be added to the graph more than once, though two vertices in the graph may be equal according to `V`'s `equals` method.

```
import java.util.Iterator;
import java.util.List;
public interface UnweightedGraphInterface<V>
{
  int getNumVertices();
  boolean isEmpty();
  void addVertex(V vertex);
  boolean hasVertex(V vertex);
  List<V> getVertices();
  Iterator<V> vertexIterator();
  void addEdge(V fromVertex, V toVertex);
  boolean hasEdge(V from, V to);
  List<V> getNeighbors(V vertex);
  GraphMarker<V> getMarker();
}
```

Implementing Graphs With Arrays

- On the next slide are the field declarations and constructors for an array-based implementation of our unweighted graph interface.
- We actually use an `ArrayList` to hold the vertices.

```
import java.util.ArrayList;
import java.util.List;

public class UnweightedDenseGraph<V>
  implements UnweightedGraphInterface<V>
{
  protected ArrayList<V> vertices;
  private boolean[][] edges;

  public UnweightedDenseGraph(int maxV) {
    vertices = new ArrayList<V>(maxV);
    edges = new boolean[maxV][maxV];
  }
```

- The ith entry in the `vertices ArrayList` stores a label for vertex i.
- The array entry `edges[i,j]` is `true` if there is an edge from `i` to `j` if there is one, or `false` if there is not.

```
  public void addVertex(V vertex) {
    vertices.add(vertex);
  }
  protected int getIndexOf(V vertex) {
    int result = vertices.indexOf(vertex);
    if (result < 0) {
      throw new GraphException(
          "Vertex not in graph: " + vertex); }
    return result;
  }
  public void addEdge(V from, V to) {
    int fromIndex = getIndexOf(from);
    int toIndex = getIndexOf(to);
    edges[fromIndex][toIndex] = true;
  }
```

The `hasEdge` Methods

The rest of `UnweightedGraphInterface` is fairly simple to implement. First we look at the `hasEdge` methods. There are two of them; one takes the indices of the vertices while the other takes the vertices themselves. The latter uses the former.

```
public boolean hasEdge(V from, V to) {
  return hasEdge(getIndexOf(from),
                 getIndexOf(to));
}
protected boolean hasEdge(int fromI, int toI) {
  return edges[fromI][toI];
}
```

Building a List of Neighbors

Here we look at the method that takes a vertex and gives back a list containing its neighbors.

```
public List<V> getNeighbors(V from) {
  int fromIndex = getIndexOf(from);
  List<V> neighbors = new ArrayList<V>();
  for (int i = 0; i < vertices.size(); i += 1)
    {
      if (hasEdge(fromIndex, i)) {
        neighbors.add(vertices.get(i));
      } }
  return neighbors;
}
```

Clicker Question #2

Suppose we have a graph of n vertices, and each vertex has at most d edges out of it. What is the worst-case running time of the `getNeighbors` method?

A. $O(d)$

B. $O(nd)$

C. $O(1)$

D. $O(n)$

3 Paths in Graphs

The `isPath` Problem

- The simplest question about paths in a graph is just whether a path *exists* with a given source and destination. We'll define a `boolean` method `isPath` that takes two vertices as parameters and answers this question.
- Once we can answer this, we know how to approach more complicated questions, like whether the whole set of vertices is **connected** (there is a path from any vertex to any other) or how to list the vertices in the same **connected component** as a given vertex.
- Eventually we'll see three approaches to this problem. Each works in the same way — it keeps a set of reachable vertices in some data structure. It advances its search by taking a vertex out of the structure, then putting any unseen neighbors it might have into the structure, until the goal node comes out of the structure or the structure becomes empty with no goal node found.
- The choice of which data structure we use determines the type of search.
- It turns out that all the approaches require some way of marking the vertices. We will examine that first.

A Vertex-Marking Interface

- Just as we keep the state of an iteration out of a collection class by using an iterator, we will keep the marking state out of the graph classes by using a `GraphMarker`.
- The `GraphInterface` class has a method `getMarker` that gets a marking object associated with the graph.

```
public interface GraphMarker<V> {
  void mark(V vertex);
  void unmark(V vertex);
  boolean isMarked(V vertex);
}
```

The BaseGraphMarker Class

```
public class BaseGraphMarker<V>
  implements GraphMarker<V>
{
  BaseGraph<V> graph;
  protected boolean[] marks;
  public BaseGraphMarker(BaseGraph<V> graph) {
    this.graph = graph;
    marks=new boolean[graph.getNumVertices()]; }
  public void mark(V vertex) {
    marks[graph.getIndexOf(vertex)] = true; }
  public void unmark(V vertex) {
    marks[graph.getIndexOf(vertex)] = false; }
  public boolean isMarked(V vertex) {
    return marks[graph.getIndexOf(vertex)]; }
}
```

Depth-First Search on a Graph

Note that we mark vertices on the way down, so we check only simple paths.

```
  public boolean isPath(String from, String to) {
    if (from == to) { return true; }
    marker.mark(from);
    List<String> vertices =
                 graph.getNeighbors(from);
    for (String vertex : vertices) {
      if (! marker.isMarked(vertex)) {
        if (isPath(vertex, to)) {
          return true;
        }
      }
    }
    return false;
  }
```