Introduction to Graphs

Kathi Fisler
adapted for CS200 by Milda Zizyte

March 7, 2022

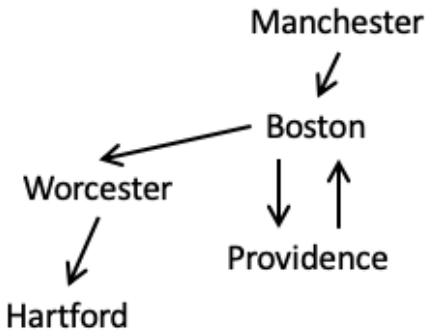
Objectives

By the end of these notes, you will know:

- What a graph is
- How to represent a graph in data structures
- How to check for a route in a graph

1 Introducing Graphs

The following diagram shows bus routes available on a New England regional bus line.



This picture has two kinds of information: cities and bus routes. Data that have some sort of item and connections between them are called *graphs*. By this definition, trees are graphs. But in this example, we see that there is a *cycle*, in which one can go from Boston to Providence and back again. Graphs may feature cycles (which is what makes them interesting). We refer to the items as *Vertices* (sometimes you may see *Nodes*), and the connections as *Edges*.

2 Data Structures for Graphs

What options might we have for data structures for graphs?

1. A list of Edge objects, where an Edge contains two Strings
2. A list of Edge objects, where an Edge contains two Vertex objects

3. A Vertex object, which contains a list of other Vertices (the ones to which there are edges)

4. A 2D array in which cells represent edges

We shall work with option 3. The array version requires you to search through the array to follow chains of edges between vertices, which we need for finding routes. The lists of edges have similar issues. Option 3 makes it easy to get from one vertex to its neighbors by following edges.

3 Classes for Graphs

Here's a graph class in which each vertex has a list of edges to other vertices.

Note: this code is different from lecture, in that we use the generic `T` to show how the code from lecture could be used to express any graph in this way.

```
1  class Graph<T> {
2
3    LinkedList<Vertex<T>> vertices;
4
5    public Graph() {
6      this.vertices = new LinkedList<Vertex<T>>();
7    }
8
9    public void addVertex(Vertex<T> v) {
10     this.vertices.add(v);
11   }
12
13   public void addEdge(Vertex<T> from, Vertex<T> to) {
14     if (this.vertices.contains(to) && this.vertices.contains(from)) {
15       from.addEdge(to);
16     }
17     // throw exception otherwise (code elided)
18   }
19 }
20
21 class Vertex<T> {
22   T val;
23   LinkedList<Vertex<T>> toVertices;
24
25   public Vertex(T v) {
26     this.val = v;
27   }
28
29   public void addEdge(Vertex<T> to) {
30     this.toVertices.add(to);
31   }
32 }
```

And here is our sample graph using our graph data structure (for example, in a `Main` class:

```
1 Graph<String> g = new Graph<String>();
2 Vertex<String> man = new Vertex<String>(``Manchester'');
3 g.addvertex(man);
4 Vertex<String> bos = new Vertex<String>(``Boston'');
5 g.addvertex(bos);
6 Vertex<String> pvd = new Vertex<String>(``Providence'');
7 g.addvertex(pdv);
```

```
Vertex<String> wos = new Vertex<String>(``Worcester'');
g.addvertex(wos);
Vertex<String> har = new Vertex<String>(``Hartford'');
g.addvertex(har);

g.addEdge(man, bos);
g.addEdge(bos, pvd);
g.addEdge(bos, wos);
g.addEdge(pvd, bos);
g.addEdge(wos, har);
```

4 Checking for Routes

One common question to ask about a graph is whether there is a path from one vertex to another. In the case of our example, we'd be asking whether there is a route from one city to another. Before we write the code, let's work out some examples (sample tests). Rather than write the tests with `Assert` in a testing class object, we're going to write these less formally as print statements in the `Main` class, to make interactive testing easier.

```
System.out.println(g.canReach(bos, pvd)); // should be true
System.out.println(g.canReach(bos, har)); // should be true
System.out.println(g.canReach(har, bos)); // should be false
System.out.println(g.canReach(pvd, pvd)); // should be true
```

The one potentially controversial test here is the one from a city to itself. Should we consider it a route if we don't actually go anywhere? Depending on your application, either answer might make sense. For our purposes, we will treat this case as true, taking the interpretation that being at your destination is more important than whether you needed to travel to get there.

Now, let's write a `canReach` method in the `Graph` class and `Vertex` class.

```
// Graph class

// determine whether one can reach the dest from the source by following edges of the graph
public boolean canReach(Vertex<T> source, Vertex<T> dest) {
  return source.canReach(dest);
}
```

```
// Vertex class

public boolean canReach(Vertex<T> dest) {
  if (this.equals(dest)) {
    return true;
   }

  for (Vertex<T> neighbor : this.toVertices) {
    if (neighbor.canReach(dest)) {
      return true;
     }
   }
   return false;
}
```

This code is a straightforward recursive traversal – we check whether the two vertices are equal (the base case). If not, then we see whether we can get to the `dest` from any of the vertices that `source` has an edge to. If so, then we can get to `dest` by taking the successful edge between `source` and that vertex, so we return true.

Our first two tests both return true. The third (Boston to Hartford) goes into an infinite loop – what happened? We will discuss and solve this problem in the next lecture.