Introduction to Graphs

Kathi Fisler and Milda Zizyte

Oct 17, 2022

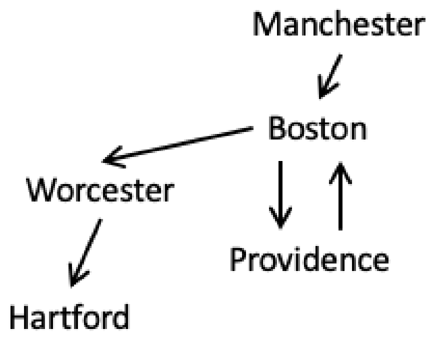
Objectives

By the end of these notes, you will know:

- What a graph is
- How to represent a graph in data structures
- How to check if a route exists between two vertices 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

Unlike trees, graphs have no hierarchy (there is no "top vertex" of a graph using which we might refer to the rest of the graph). This indicates that we need to represent a graph as some collection of graphs, vertices, or both.

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 list of Vertex objects, which contains a list of other Vertices (the ones to which there are edges)
4. A 2D array (called an "adjacency matrix") in which cells are boolean values that are true if an edge exists between the vertices represented by the row and column of that cell

We shall work with option 3. The array version requires you to search through the array to identify the outgoing edges of a vertex, instead of having a way to access them directly. The lists of edges have similar issues. Option 3 makes it easy to get from one vertex to its neighbors by following edges. Since we are not concerned about imposing an order on the collection of vertices, we will use a HashSet instead of a list to hold all of them so that we can check if a Graph contains a Vertex in constant time.

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.

```
class Graph<T> {

  HashSet<Vertex<T>> vertices;

  public Graph() {
    this.vertices = new HashSet<Vertex<T>>();
  }

  public void addVertex(Vertex<T> v) {
    this.vertices.add(v);
  }

  public void addEdge(Vertex<T> from, Vertex<T> to) {
    if (this.vertices.contains(to) && this.vertices.contains(from)) {
      from.addEdge(to);
    }
    // throw exception otherwise (code elided)
  }
}

class Vertex<T> {
  T val;
  LinkedList<Vertex<T>> toVertices;

  public Vertex(T v) {
    this.val = v;
  }

  public void addEdge(Vertex<T> to) {
    this.toVertices.add(to);
  }
}
```

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

```
Graph<String> g = new Graph<String>();
Vertex<String> man = new Vertex<String>(``Manchester'');
```

```java
g.addvertex(man);
Vertex<String> bos = new Vertex<String>(``Boston'');
g.addvertex(bos);
Vertex<String> pvd = new Vertex<String>(``Providence'');
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).

```java
assertTrue(g.canReach(bos, pvd));
assertTrue(g.canReach(bos, har));
assertFalse(g.canReach(har, bos));
assertTrue(g.canReach(pvd, pvd));
```

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.

```java
// 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);
}
```

```java
// 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. In the terms of our example graph, this is saying “How do we know if we can get from Manchester to Worcester? Well, we know there’s a bus route from Manchester to Boston. If we can get from Boston to Worcester, then we know we can get from Manchester to Boston using that bus route.”

In the next lecture, we will try to run these tests and discuss a problem we run into.