Lecture 28: Representing and Traversing Graphs

11:00 AM, Mar 26, 2021

Contents

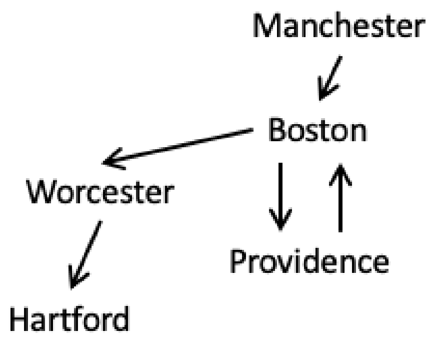
Objectives

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 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 Edges, where an edge contains two Strings
2. A list of Edges, where an edge contains two Nodes
3. A Node object, which contains a list of other Nodes (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 nodes, which we need for finding routes. The lists of edges have similar issues. Option 3 makes it easy to get from one node to its neighbors by following edges.

3 Classes for Graphs

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

```
class Graph[T] {

  class Node(val contents: T, var getsTo : List[Node]) {

      // inserts outbound edge to given node
      def addEdge(toNode : Node) =
            getsTo = toNode :: getsTo

      override def toString = contents.toString
  }

  // a list of all the nodes in the graph
  private var nodes = List[Node]()

  def createNode(contents: T) = {
      val newNode = new Node(contents, List())
      nodes = newNode :: nodes
      newNode
  }

  def addEdge(fromNode: Node, toNode: Node) =
      fromNode.addEdge(toNode)
  def show =
      for (node <- nodes) {
          println(node.contents + " gets to " + node.getsTo.toString)
}
```

And here is our sample graph using our graph data structure:

```
object BusRoute {
   val G = new Graph[String]()

   val bos = G.createNode("Boston")
   val pvd = G.createNode("Providence")
   val man = G.createNode("Manchester")
   val wor = G.createNode("Worcester")
   val har = G.createNode("Hartford")

   G.addEdge(man,bos)
   G.addEdge(bos,wor)
   G.addEdge(wor,har)
   G.addEdge(bos,pvd)
   G.addEdge(pvd,bos)
}
```

4 Checking for Routes

One common question to ask about a graph is whether there is a path from one node 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 `checkExpect` in a `Main` object, we're going to write these less formally as methods in the BusRoute class, to make interactive testing easier.

```
def checkmm() = G.hasRoute(man, man) // should be true
def checkmb() = G.hasRoute(man, bos) // should be true
def checkbh() = G.hasRoute(bos, har) // should be true
def checkhp() = G.hasRoute(har, pvd) // should be false
```

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 `hasRoute` method in the `Graph` class.

```
// determine whether one can reach the toNode from the fromNode
// by following directed edges in the graph
def hasRoute(fromNode: Node, toNode: Node): Boolean = {
   if (fromNode.equals(toNode))
     true
   else {
      for (next <- fromNode.getsTo) {
            if (this.hasRoute(next, toNode))
                return true
        }
        false
    }
}
```

This method is a straightforward recursive traversal – we check whether the two nodes are equal (the base case). If not, then we see whether we can get to the `toNode` from any of the nodes that `fromNode` has an edge to. If so, then we can get to `toNode` by taking the successful edge out of `fromNode`, so we could return true.

Our first two tests both return true. The third (Boston to Hartford) goes into an infinite loop – what happened?

4.1 Traversing Data with Cycles

Consider the sequence of computations if we try to compute a route from Boston to Hartford. Since these aren't the same city, we'll try the edges out of Boston. There are two, Providence and Worcester. So the `for` loop will try them in order. The following listing shows the two calls pending, as we start the first of them.

```
G.hasRoute(bos, har)
  // for loop generates two calls:
  G.hasRoute(pvd, har)   <--- try this one first
  G.hasRoute(wor, har)
```

When we check for a route from Providence to Hartford, we follow the edges out of Providence to look for a route. Given the arrangement of the recursive calls and the for-loop, we will try the edges out of Providence before we try the route out of Worcester (still pending from the original `hasRoute` call:

```
G.hasRoute(bos, har)
  // for loop generates two calls:
  G.hasRoute(pvd, har)
    G.hasRoute(bos, har)  <--- try this one next
  G.hasRoute(wor, har)
```

Since Boston and Hartford aren't the same city, we will expand the edges out of Boston (following the for-loop in the current recursive call):

```
G.hasRoute(bos, har)
  // for loop generates two calls:
  G.hasRoute(pvd, har)
    G.hasRoute(bos, har)
      G.hasRoute(pvd, har)  <--- now try this
      G.hasRoute(wor, har)
  G.hasRoute(wor, har)
```

Hopefully, you see that this will not end well (or, frankly, at all). The execution never gets to try a route out of Worcester (which would have worked) because it gets stuck in the cycle between Boston and Providence.

You might ask whether we could solve this problem by creating the initial graph differently, such that Worcester appears before Providence in the list of edges out of Boston. That would avoid the issue for this specific graph, but it wouldn't solve the cyclic data problem in the general case.

So what do we do? We need some way to track which nodes we've already tried, so that we don't try them again.

4.2 Tracking Previously Visited Nodes

One way to track nodes we've already seen would be to add a field to the Node class to record this:

```
class Node(val contents: T, var getsTo : List[Node]) {
  var visited = false
  ...
}
```

This won't end up being a good idea in practice. This assumes that only one route check will be running over the graph at the same time. In real scale systems (like your favorite map application), there can be dozens of searches happening over the same data at the same time. We therefore need to maintain the visited information outside the nodes.

We will augment our `hasRoute` implementation with a separate List containing the Nodes that we've already searched from. We check the list before expanding out new calls from the `fromNode`, and we add to the list of checked nodes before making a new recursive call. Here's the code (and an updated initial call to `hasRoute`.:

```
def hasRoute(fromNode: Node, toNode: Node, visited: List[Node]): Boolean = {
    // return a boolean to say "is there a route" (we'll compute it on
      Wedesnday)
    if (fromNode.equals(toNode)) {
      true
    } else if (visited.contains(fromNode)) {
      return false
    }else {
      for (next <- fromNode.getsTo) {
        if (this.hasRoute(next, toNode, fromNode :: visited))
          return true
      }
      false
    }
}

def checkbh() = G.hasRoute(bos,har, List()) //should be true
```

With this modification, the infinite loop no longer occurs.

One way to see why is to write out the sequence of calls again.

```
G.hasRoute(bos, har, List())
  // for loop generates two calls:
  G.hasRoute(pvd, har, List(bos)
    G.hasRoute(bos, har, List{bos,pvd))
      // bos is in visited list, so for-loop doesn't happen
  G.hasRoute(wor, har, ???)
```

Stop and Think: What fills in the ??? in the above call expansion?

Stop and Think: What is the runtime of `hasRoute` with the `visited` list?

5 Study Questions

1. Why does the Graph class have a separate method for addEdge, rather than take the list of edge nodes only as a constructor variable?
2. Is the version of `hasRoute` with the `visited` list functional or imperative?
3. We noted that the problem of doing computations multiple times, which led to excessive runtime, was a motivation for dynamic programming. Does this suggest that we could use dynamic programming to handle the infinite loop problem with `hasRoute`?

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.