Lecture 28: Representing and Traversing Graphs

11:00 AM, Mar 26, 2021

Contents

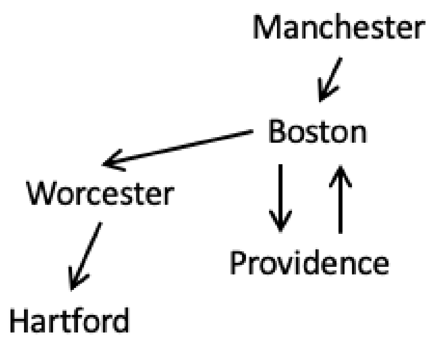
Objectives

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 Scala set containing the Nodes that we've already searched from. Before we make that change, however, we're going to rewrite our current `hasRoute` code to let us play with a critical traversal option.

4.3 Maintaining a List of Nodes to Check

The for-loop in `hasRoute` implicitly sets up a stack of recursive calls. For sake of clarity, rather than leave the sequence of calls implicit in the unrolling of the for-loop, let's rewrite this code to maintain an explicit list of the cities that we need to check in order. We'll add cities to this list as we look for routes, and we'll use a `while` loop to process cities from the list until we've run out of cities:

```
def hasRouteChecklist(fromNode: Node, toNode: Node): Boolean = {
   var check = List(fromNode)

   while(!check.isEmpty) {
      val next = check.head
      check = check.tail

      if (next.equals(toNode))
           return true
      else {
           check = newNode :: check
      }
   }
   false
}
```

If you hand-trace this code, you'll find that we visit the cities in exactly the same order (with the same infinite loop!) as in our original. The only difference lies in maintaining the list of cities to visit, rather than having that sequence implicit in the recursive calls that get made.

4.4 Adding a Set of Visited Nodes

Let's augment this new version with a set of nodes that have already been visited. We create a mutable set, add nodes to it as we visit them, and only add nodes to the `check` list if we haven't visited them previously:

```
def hasRouteCheckListVisited(fromNode: Node, toNode: Node): Boolean = {
    var check = List(fromNode)
    val visited = scala.collection.mutable.Set[Node]()

    while(!check.isEmpty) {
       val next = check.head
       check = check.tail

       if (next.equals(toNode))
          return true
       else {
          visited.add(next)
          for (newNode <- next.getsTo)
             if (!visited.contains(newNode))
                 check = newNode :: check
       }
    }
    false
}
```

Once you add a data structure to track progress of a computation, you should write down an invariant to capture what you expect of that data structure. Here, we expect the following invariant:

```
def hasRouteCheckListVisited(fromNode: Node, toNode: Node): Boolean = {
    var check = List(fromNode)
    val visited = scala.collection.mutable.Set[Node]()
    // INVARIANT: Every node in visited has already been checked
```

Ask Yourself: What about the converse (opposite) assertion? Is it true that every node in checked has never been visited?

5 Depth-First Search (DFS)

Our `hasRouteCheckListVisited` function implements a classic graph algorithm known as *depth-first search* (DFS). With depth first search, when a node has more than one outbound edge, we explore all of the paths out of the first edge (in our example, Providence) before we explore paths out of the other edges (Worcester).

Where in the code does this "deep" decision get made? It's in the expression where we augment the `check` list. By putting the nodes reachable from the current node at the *front* of the `check` list, we guarantee we will visit them before we visit nodes that are already pending in the *check* list.

6 Breadth-First Search (BFS)

So what if we made a different decision there, and put the new nodes at the end of the *check* list instead of at the front? In other words, what if our code looked like:

```
          for (newNode <- next.getsTo)
             if (!visited.contains(newNode))
                 // the change is in the next line
                 check = check ::: List(newNode)
```

Now which order would we visit the nodes in when searching for a Boston-Hartford route?

We'd still start at Boston, putting Providence and Worcester in the *check* list (in that order). When processing Providence, we would put any next-nodes that we hadn't visited *after* Worcester in the list (in this case, there aren't any Providence next-nodes we haven't visited, but that's besides the point). In this way, we wouldn't get stuck in a deep infinite cycle without also making progress on the other paths.

This version is called *breadth-first search* (BFS), because it explores the breadth of next nodes before moving onto their next nodes. This implies that you first check all nodes reachable in one step from the starting point, then all nodes reachable in two steps from the starting point, then all nodes reachable in three steps, and so on.

If we were using breadth-first search, would we still need a visited list? Wouldn't we eventually find the route, before getting stuck in an infinite loop? If there is a route, you would find it without going into an infinite loop. But what if there is no route (as with Hartford to Boston)? Then, you would get stuck in an infinite loop if you didn't have a visited list.

Furthermore, the visited list is still useful for time-efficiency. When your graph has multiple paths of different length to the same node from your starting point, you'd end up exploring that same node for each path, rather than only once.

7 Differences between Depth-First and Breadth-First Search

At core, the difference between DFS and BFS is that the former maintains the check list as a *stack*, while the latter maintains it as a *queue*. As we have seen, in code this difference amounts to using `addFirst` versus `addLast` when adding nodes to the list.

Which algorithm (BFS or DFS) should you use in practice? It depends on context.

- If your goal is to find the shortest path length, use BFS. Since BFS checks all nodes at each distance from the starting node before looking at any node at distance + 1, if there are two paths of different lengths to the same node, BFS detects the shortest one first.
- If your goal is detect cycles, use DFS. As soon as DFS tries to process a node that is already in the visited list, you know you have a cycle in the graph. Cycle-detection ends up being a key component of some advanced computing applications – we will study one of these in the next lecture.
- Also, given the shape of your particular graph and characteristics of your route queries, one of BFS or DFS might perform better in practice.

We'll see a couple of interesting applications of these algorithms before the semester ends. First one comes up next class ...

8 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. When we made the set of nodes that we had previously visited, why did we make it mutable rather than immutable? Can you find a graph that illustrates why we made it immutable (assume DFS)?
3. Create some other sample graphs and make sure you understand the differences in the orders that depth-first and breadth-first search would consider when looking for paths.
4. The fundamental problem with looking for routes with depth-first search is that we might go into an infinite loop if we end up making the same function call a second time. This sounds familiar to what we looked at in dynamic programming. There, we made an array of all the possible function inputs and would have tracked the computed answer. How does the list of visited nodes here compare to the array of stored outputs in dynamic programming?
5. If you were going to search for paths, not just their existence, would you rather build on depth-first or breadth-first search?
