Lecture 29: Exploring Graphs

11:00 AM, Mar 29, 2021

Contents

Objectives

By the end of these notes, you will know:

- An alternative code outline for traversing graphs
- How to compute paths between nodes
- How to compute shortest paths between nodes

1 Refining our Previous Graph Traversal Code

We ended last class with the following function to check whether there is a route from one node to another:

```
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
    }
  }
```

This is a clean recursive solution. However, in order to highlight the differences between different approaches to traversing graphs (for different purposes), we're going to rewrite this code so that the order in which nodes are explored is made explicit.

1.1 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 = next.getsTo :: 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.

1.2 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
}
```

The interesting part of this new version is the `for` loop that adds new nodes to the `check` list. Here, we add the nodes to the front of the `check` list. But we could make other choices. We will explore one of those shortly. But first, let's compute routes.

2 Tracking Routes

Right now, `hasRouteChecklist` will tell us whether there is a route, but not which nodes to take to get between nodes. To produce the route, we also need to record the node through which each node entered the `check` list.

When we first discussed dynamic programming, we created an additional data structure (a hashmap or an array) to store information about how the computation had progressed so far. We will use the same idea here. In this case, we will create a hashmap that maps each node to the node that added it to the checklist. The following code calls that hashmap `cameFrom`.

```
  def getRouteChecklist(fromNode: Node, toNode: Node): Option[List[Node]] =
     {
    var check = List(fromNode)
    var visited = List[Node]()
    val cameFrom = scala.collection.mutable.Map[Node,Node]()

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

      if (curr.equals(toNode)) {
        return Some(getPath(fromNode, toNode, cameFrom)
      } else {
        visited = curr :: visited
        for(n <- curr.getsTo) {
          if (!visited.contains(n)) {
            check = n :: check
            cameFrom(n) = curr
          }
        }
      }
    }
    None
  }
```

Note that the method should now return a list of nodes that constitute the path, not just a boolean. But since there might not be a path, we'll have the return be an `Option` type.

The `getPath` method, which is not included here, builds the path from the contents of `cameFrom`. On our sample graph, for example, after a search from Boston to Hartford, `cameFrom` would have the following key-value pairs:

```
pvd -> bos
wor -> bos
har -> wor
```

`getPath` would retrieve the `toNode` (`har`), retrieve which city got to `har`, then iterate that process until reaching back to the `fromNode`.

3 Finding Shortest Paths: Dijkstra's Algorithm

Now assume that we care about not just finding some path, but about finding a good path. There are many definitions of good paths. One of them might be a path that cost the least, or that had the shortest total distance. For this, we want to use an algorithm called Dijkstra's Algorithm.

Dijkstra's algorithm assumes that our graphs have costs or weights on the edges. Without these, there's no useful notion of a "shortest" path. Here's a different graph, this times with edge weights. In this graph, we will also assume that edges are bidirectional (meaning that you could travel them in either direction).

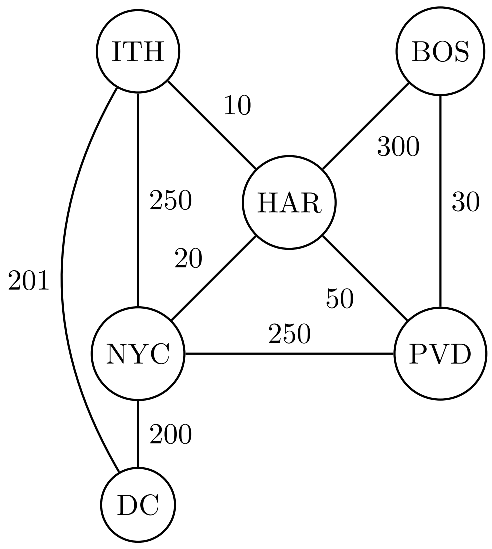

Figure 1: An example weighted graph. Edges are undirected; travel is possible in both directions at the same cost.

How do we think about finding short paths. Think back to our discussion of how we might add items to the `check` list. Shortest paths involve some sort of optimization, or priority of which nodes to consider. In this case, we want to check nodes in an order based on the cheapest way to make progress in our search for a path.

For today, we'll give an overview of the algorithm. You'll implement it in lab, then we'll come back and talk about it in more detail on Friday.

The idea of the algorithm is to maintain a hashmap that stores, for each node, the lowest cost to reach that node from the `fromNode`. The lecture capture shows how this works (the remaining notes are for reference, but the lecture capture is likely clearer).

Initially, all nodes are infinitely far from the starting node (except the start node itself).

After initialization, we should see the following estimates in `dist`:

`[BOS:0, PVD:inf, HAR:inf, ITH:inf, NYC:inf, DC:inf]`

These values form the priorities that will determine the order of elements in the priority queue. The algorithm removes `BOS` from the priority queue; our estimate of 0 for reaching Boston is optimal (we mark this with `[X]`). Now, for each of Boston's neighbors (`HAR` and `PVD`) we check to see if we can do better than the current estimate. Since the current estimates are both infinity, and Boston has finite-cost edges to both cities, we can improve both estimates.

1. `[BOS:0 [X], PVD:30, HAR:300, ITH:inf, NYC:inf, DC:inf]`

Now the algorithm removes `PVD`: 30 is the optimal path length from Boston to Providence. We can improve estimates for 2 of Providence's neighbors: Hartford and New York City. Both now equal the optimal cost to reach Providence (30) plus the length of the direct edge from Providence (50 and 250 respectively):

2. `[BOS:0 [X], PVD:30 [X], HAR:80, ITH:inf, NYC:280, DC:inf]`

The algorithm continues until it runs out of nodes on the priority queue, updating estimates as follows:

3. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90, NYC:100, DC:inf]`
4. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100, DC:291]`
5. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100 [X], DC:291`
6. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100 [X], DC:291[X]`
