DFS and BFS

Kathi Fisler
adapted for CS200 by Milda Zizyte

March 9, 2022

Objectives

By the end of these notes, you will know:

- How to keep track of information to avoid infinite loops when traversing graphs
- An alternative code outline for traversing graphs
- How order of vertices considered matters when traversing graphs

1 Solving the infinite loop

In the previous lecture, we saw that the following attempt at trying to answer if a path exists between two vertices might encounter an infinite loop:

```
// 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;
}
```

Why is this? It turns out that the culprit is the cycle that exists between Boston and Providence.

1.1 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 `foreach` loop will try them in order. The following listing shows the two calls pending, as we start the first of them.

```
bos.canReach(har)
  pvd.canReach(har) // <-- try this one first

  wos.canReach(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 `foreach`, we will try the edges out of Providence before we try the route out of Worcester (still pending from the original `canReach` call:

```
bos.canReach(har)
  pvd.canReach(har) // <-- try this one first
    bos.canReach(har)

  wos.canReach(har) // still haven't gotten here
```

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

```
bos.canReach(har)
  pvd.canReach(har)
    bos.canReach(har)
     pvd.canReach(har)
       // ...
     wos.canReach(har) // <-- never get here

  wos.canReach(har) // <-- nor here!
```

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 vertices we've already tried, so that we don't try them again.

1.2 Tracking Previously Visited Vertices

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

```java
class Vertex<T> {
  boolean visited; // initialize to 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 vertices.

We will augment our `canReach` (renamed `canReachMemory` implementation with a separate `HashSet` containing the vertices that we've already searched from. We check the list before expanding out new calls from the `canReachMemory`, and we add to the list of checked vertices before making a new recursive call. Here's the code:

Stop and Think: Why use a `HashSet` here? *(this is discussed in the lecture)*

```java
public boolean canReach(Vertex<T> dest) {
  return this.canReachMemory(dest, new HashSet<Vertex<T>>());
}

public boolean canReachMemory(Vertex<T> dest, HashSet<Vertex<T>> visited) {
  if (this.equals(dest)) {
    return true;
  }
  HashSet<Vertex<T>> newVisited = new HashSet<Vertex<T>>(visited);
  newVisited.add(this);
  for (Vertex<T> neighbor : this.toVertices) {
    if (! visited.contains(neighbor)) {
      if (neighbor.canReachMemory(dest, newVisited)) {
        return true;
      }
    }
  }
  return false;
}
```

(In the lecture, we discuss why we copied over visited to newVisited in this code so that we could more easily draw out its contents. In practice, we would just pass along the same HashSet.)

With this modification, the infinite loop no longer occurs.

One way to see why is to write out the sequence of calls again. We use a shorthand here to denote the contents of the `HashSet`.

```
bos.canReach(har, {})
  pvd.canReach(har, {bos})
  // do not call bos.canReach(har, ...) because it has been marked visited; return false
  wos.canReach(har, {bos})
    har.canReach(har, {bos, wos}) // return true!
```

2 Refining our Graph Traversal Code

The previous code shows 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 vertices are explored is made explicit.

2.1 Maintaining a List of Vertices to Check

The `foreach` loop in `canReach` implicitly sets up an order in which vertices are considered, using recursive method calls. For sake of clarity, rather than leave the sequence of calls implicit in the unrolling of the `foreach` 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 (we'll discuss why we tacked on a "DFS" to the method name in the next section):

```
  // in the Graph class
public boolean canReachDFS(Vertex<T> source, Vertex<T> dest) {
  LinkedList<Vertex<T>> toCheck = new LinkedList<Vertex<T>>();
  HashSet<Vertex<T>> visited = new HashSet<Vertex<T>>();

  toCheck.addLast(source);

  while (! toCheck.isEmpty()) {
    Vertex<T> checkingVertex = toCheck.removeLast();
    if (dest.equals(checkingVertex)) {
      return true;
    }
    visited.add(checkingVertex);
    for (Vertex<T> neighbor : checkingVertex.toVertices) {
      if (!visited.contains(neighbor)) {
        toCheck.addLast(neighbor);
      }
    }
  }
  return false;
}
```

If you hand-trace this code, you'll find that we queue up visits to vertices in the same order as in our original recursive code. 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.

2.2 Exploring in a different order

Our `canReachDFS` function (from last class) implements a classic graph algorithm known as *depth-first search* (DFS). With depth-first search, when a vertex 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). That is, we go deep into the graph before exploring other paths!

Where in the code does this "deep" decision get made? It's in the lines where we `addLast` to `toCheck` and `removeLast` from `toCheck`. By putting the vertices reachable from the current vertex at the *back* of the `toCheck` list, and by removing also from the back of this list, we guarantee that we will visit them before we visit vertices that are already pending in the list.

2.3 Breadth-First Search

So what if we made a different decision there, and took the new vertices off the *front* of the list rather than the back? In other words, what if our **while** loop looked like:

```
while (! toCheck.isEmpty()) {
    Vertex<T> checkingVertex = toCheck.removeFirst(); // this line changes!
    if (dest.equals(checkingVertex)) {
      return true;
    }
    visited.add(checkingVertex);
    for (Vertex<T> neighbor : checkingVertex.toVertices) {
      if (!visited.contains(neighbor)) {
        toCheck.addLast(neighbor);
      }
    }
```

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

We'd still start at Boston, putting Providence and Worcester in the `toCheck` list (in that order). When processing Providence, we would put any neighbor Vertices that we hadn't visited *after* Worcester in the list (in this case, there aren't any Providence neighbors 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 vertices before moving onto their next vertices. This implies that you first check all vertices reachable in one step from the starting point, then all vertices reachable in two steps from the starting point, then all vertices 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 vertex from your starting point, you'd end up exploring that same vertex for each path, rather than only once.

In the next class, we will discuss where it might be more appropriate to use one search over the other.