DFS - from recursion to data structure

Kathi Fisler and Milda Zizyte

October 19, 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

1 Solving the infinite loop

In the previous lecture, we implemented a `canReach` method:

```
1   // Vertex class
2
3   public boolean canReach(Vertex<T> dest) {
4     if (this.equals(dest)) {
5       return true;
6      }
7
8     for (Vertex<T> neighbor : this.toVertices) {
9       if (neighbor.canReach(dest)) {
10        return true;
11       }
12     }
13     return false;
14  }
```

When we run our tests from the previous lecture, the first two pass. The third (Boston to Hartford) goes into an infinite loop – what happened?

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 (line 4 of `canReach`), we'll try the edges out of Boston. There are two, Providence and Worcester. So the `foreach` loop (line 8) will try them in order. The following listing shows the two recursive calls that `canReach` would make, as we start the first of them.

```
bos.canReach(har)
  pvd.canReach(har) // <-- run and return from 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 that we are executing `pvd.canReach` now, the code will try the edges out of Providence before trying the route out of Worcester (still pending from the original `canReach` call):

```
bos.canReach(har)
  pvd.canReach(har) // <-- run and return from 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:

```
1 class Vertex<T> {
2   boolean visited; // initialize to false
3   ...
4 }
```

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 `canReachHelper`) implementation with a separate `HashSet` containing the vertices that we've already searched from. We check the set before expanding out new calls from the `canReachMemory`, and we add to the set 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)*

```
1 public boolean canReach(Vertex<T> dest) {
2   return this.canReachHelper(dest, new HashSet<Vertex<T>>());
3 }
4
5 public boolean canReachHelper(Vertex<T> dest, HashSet<Vertex<T>> visited) {
```

```
6   if (this.equals(dest)) {
7     return true;
8   }
9   visited.add(this);
10  for (Vertex<T> neighbor : this.toVertices) {
11    if (! visited.contains(neighbor)) {
12      if (neighbor.canReachHelper(dest, visited)) {
13        return true;
14      }
15    }
16  }
17  return false;
18 }
```

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 `visited`.

```
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, pvd}) // pvd is in visited because pvd.canReach
                                // added it to visited, which is the same
                                // HashSet in memory for all recursive calls
    har.canReach(har, {bos, pvd, 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):

Note: this code was updated on October 30 to remove the redundant add to visited that used to be between lines 13 and 14 of the updated code.

```
1   // in the Graph class
2  public boolean canReachDFS(Vertex<T> source, Vertex<T> dest) {
3    LinkedList<Vertex<T>> toCheck = new LinkedList<Vertex<T>>();
4    HashSet<Vertex<T>> visited = new HashSet<Vertex<T>>();
5
6    toCheck.addLast(source);
7    visited.add(source);
8
9    while (! toCheck.isEmpty()) {
10     Vertex<T> checkingVertex = toCheck.removeLast();
11     if (dest.equals(checkingVertex)) {
```

```
12        return true;
13      }
14      for (Vertex<T> neighbor : checkingVertex.getOutgoing()) {
15        if (!visited.contains(neighbor)) {
16          visited.add(neighbor);
17          toCheck.addLast(neighbor);
18        }
19      }
20    }
21    return false;
22  }
```

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.

When we hand-traced this code in lecture, you may have noticed that we add a vertex to the `visited` at same time as when we add it to the `toCheck` list. This resulted in a picture where a vertex was marked "visited" before we even took it off the list as the `checkingVertex` (line 10). Why mark a vertex as "visited" before we perform a computation on it? Because we are using `visited` to denote that we have marked the vertex for computation (i.e. put it in `toCheck`), so that it does not get added twice.

Our `canReachDFS` function 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! In the next lecture, we'll see where in the code this decision gets made and see an alternative order in which to examine vertices.