Lecture 27: Graphs: Traversal (Part 2)

10:00 AM, Apr 4, 2018

Contents

1 Traversing Arbitrary Graphs

Objectives

By the end of this lecture, you will be able to:

- traverse arbitrary directed graphs in both depth-first and breadth-first order

1 Traversing Arbitrary Graphs

In the previous lecture, we saw breadth- and depth-first traversal algorithms for trees—a very specific kind of directed graph. But it turns out that the algorithms we wrote before don’t quite work on all graphs. Let’s go back to the non-tree graph example from last time:

![Figure 1: A directed graph](image)

For simplicity, let’s focus just on breadth-first search. Here’s where we left off (in pseudocode):

```plaintext
BFT(G, v) {
    Q = new Queue()
    Q.add(v)
    while(!s.empty()) {
        v = Q.getFirst()
        Q.removeFirst()
        for(w in G.adjs(v)) {
            Q.add(w)
        }
    }
}
```

What happens if we run this algorithm on the above graph, starting from node 2? We see the following sequence of queue states (assuming a left-to-right ordering on adjacencies): [2], [1, 4], [5], [2, 4, 3].
[4, 2, 5], [2, 5, 4], and so on. The algorithm keeps re-visiting node 2, and never terminates! If we want to traverse graphs with cycles (like, say, the Facebook friendship graph or a street map) we’d better refine our algorithm.

The trouble is that, at any point in our traversal, there are 3 different types of node:

1. nodes that have yet to be visited at all;
2. nodes that have been seen and added to the work-list, but whose children have not yet been added; and
3. nodes that have been fully visited and expanded.

Our to-do list only lets us tell the difference between two of these!

To prevent getting caught in cycles, we’ll add an additional data structure to hold the set of nodes we’ve visited already. We’ll only add a node’s adjacencies to the work-list if it’s the first time we’ve seen that node. The actual addition to the pseudocode is fairly small—just the red lines in the following:

```java
BFT(G, v) {
  Q = new Queue()
  visited = new Set()
  Q.add(v)
  while(!s.isEmpty()) {
    v = Q.getFirst()
    Q.removeFirst()
    if(!visited.contains(v)) {
      visited.add(v)
      for(w in G.adjs(v)) {
        if(!visited.contains(w))
          Q.add(w)
      }
    }
  }
}
```

**A Design Question** Since the whole point of the visited set is to keep track of which nodes we’ve seen before, it’s worth asking why we didn’t just add a new field to the nodes (to record a “check mark”) rather than keeping the state in the function. There are at least two reasons.

First, there’s no guarantee that we—the authors of the BFT function—have any control over the nodes at all. Indeed, it’s common to see graph representations where nodes are just numeric indices into an adjacency matrix! In such cases, there’s no accessible node “class” to add a field to.

Second, we often write code that’s meant to operate on shared data. Imagine Facebook’s friendship graph. It’d be hugely inefficient to make a fresh copy of the entire graph every time someone wanted to search it. But if multiple searches shared the same graph, and the nodes had a field that indicated they’d been visited before, each node could only be visited once across all simultaneous searches! So storing the set locally to the function is the right call, here.