Graphs and Trees - And (Non-)Recursive Algorithms, D&C, and Sorting (Ch. 15 into 16)

From page 479 onward, the text can be read apart from the particular 2D grid example that was begun in earlier chapters.

Representing Graphs

Abstractly, a graph is an object $G = (V, E)$ where $V$ is a set of vertices, also called nodes, and $E$ is a subset of $V \times V$ that consists of pairs $(u, v)$ called edges. If the graph satisfies

$$(u, v) \in E \iff (v, u) \in E$$

for all nodes $u$ and $v$, then it is undirected, else it is (properly) directed. An undirected graph "Is-A" directed graph in which every edge $(u, v)$ is accompanied by its reversal $(v, u)$, but these classes are most often thought of as separate concepts. Some standard notational conventions that go with graphs:

- Vertices are numbered $i = 1$ to $n$, where $n = |V|$.
- Edges are numbered 1 up to $m = |E|$. In a directed graph without self-loops, $m$ can go as high as $n(n-1) \sim n^2$. In an undirected graph, the max # of edges is half that, i.e.

$$\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2).$$

- The size of the graph is called $n$ or $m$ according to context. Sometimes we use $N$ to mean the "true data size."

Often a graph is represented by another structure, such as a matrix or grid. Here is the 4-cycle graph and its adjacency matrix:

A rectangular grid graph has $V \subseteq \{1, \ldots, r\} \times \{1, \ldots, s\}$ minus some cells that are "blacked out." Its edges are pairs of cells that are adjacent horizontally or vertically [or diagonally]. The text uses this kind of implicit representation of nodes and edges early on. (Networked computations on various kinds of grids is a major research topic of our department associated with Professor Russ Miller and CSE429.)

When some other data point(s) are associated with each node or edge, it pays to represent nodes and/or edges as explicit objects. There are several conventions for this.
1. Use a separate Node class and/or Edge class. This is common in older texts. **Problem:** the class has public visibility and global scope by default. Different "Node" classes might name-clash. Even if not, the "too many little classes" problem.

2. Nest the Node class and/or Edge class inside the graph class. The text does this on pages 478-479 (though the graph class is less-abstractly called DrawGraph). This is standard, but in Scala there is the technicality that inner classes are subsidiary to the enclosing object, not the enclosing class.

3. Special kinds of graphs have recursive definitions that allow "flattening" the concepts of node and graph.

We have already seen an example of 3. A linked list is a graph whose nodes form a single path (or in the case of a circularly liked list, a cycle like the one above). In ISR we have followed the standard representation 2. But the native List type has no notion of "node"—just the data element directly and recursion back into List. The text shows how this is synthesized back in section 12.6 (which I skipped then—but the point will reappear at the end of section 16.3).

Within representation style 2, the choice is whether to represent edges separately or just use tuples. The text on p479 associates data with edges, so it has a separate GEdge class from GNode. Each node holds a set edges of its neighbors as a Set[GEdge]. One technicality is that an undirected graph needs to recognize that (v, u) is the same edge as (u, v).

GEdge has fields from and to.

Even though representation 2 is usually considered non-recursive, algorithms based on it are often recursive. This can be done with the basic reachability algorithm. But it is also possible to view it non-recursively as a form of breadth-first search (BFS).

**Graph Algorithm: Reachability and Search**

To understand what the text is saying on page 480, it pays to insert the idea of "freshly" reaching into the name of the algorithm. The motivation is to eliminate going-around-in-cycles. Here is the pure logic:

A goal end-node e can be freshly reached from a given node u if:
- u = e OR
- (u is fresh, i.e. not an already-visited node, AND
- there is an edge (u, v) to a neighbor v such that—upon declaring u to have been visited, i.e. done-with—we can freshly reach e from v.)

Now we can read the code, using special Scala features (that aren't necessarily recommended):
def canFreshlyReach(u: GNode, visited: mutable.Set[GNode] = new Set[GNode]()) = {
  if (u.equals(endNode)) true
  else if (visited.contains(u)) false
  else {
    visited += u
    u.edges.exists(edge => canFreshlyReach(edge.to, visited))
  }
}

[ Show animations at https://visualgo.net/en --- under Graph Traversals ]

Those animations are better described as showing the classic procedural rather than recursive reachability algorithms: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

The idea of BFS is that nodes \( u \) are first visited and then (later) expanded. Expanding \( u \) causes all of the fresh neighbors of \( u \)—those not previously visited—to be visited. The node to expand is always the earliest-visited node that hasn’t been expanded yet. **Earliest** means we have a first-in, first-out situation, which means using a queue to manage the visited list. Since we’re not putting weights on the edges, we can use a simpler node representation where the out-neighbors of \( u \), meaning those \( v \) such that \( (u, v) \) is an edge, are kept as a list:

class Graph[A] {
  class Node(var item: A, val nbhrs: List[Node]) //mutable data, fixed connections
  ...
}

The code—*if placed inside the Graph[A] class*—is:

def classicBFS(start: Node, goal: Node): Boolean = {
  if (start == goal) { return true }
  //else
  var visited = new Queue[Node]() //Scala Queue is mutable by default
  var expanded = new scala.collection.mutable.Set[Node]()
  visited.enqueue(start) //so queue is nonempty
  while (!visited.isEmpty) {
    val u = visited.pop()
    expanded += u
    for (v <- u.nbhrs) {
      if (v == goal) {
        return true
      }
      visited.enqueue(v)
    }
  }
  return false
}
return true
} else if (!(expanded.contains(v) || visited.contains(v))) {
    visited.enqueue(v)
}
} //exit of while loop means no more fresh nodes to visit/expand
return false
}
The algorithm is sound because if it returns true, it means there is a way to get from start to the goal node by traversing edges, so goal is reachable from start. It is comprehensive, and hence correct, because whenever goal is reachable from start, in k steps, say, then the BFS process will eventually find such a path. [This actually requires an inductive proof on k.]

If this code were outside Graph[A], then we would have the issue of accessing the nested Node class. We could write Graph[...].Node (with whatever concrete type is used in place of A), and since we're only receiving nodes from the object via pop(), we should not have to type-cast them to the graph object. This presumes that Node were made public or package-visible. But it is more proper anyway that the BFS code belong to the class---and Scala's general treatment of the inner Node class is hinting that it belongs to the graph object itself.

**Runtime of BFS**

The runtime analysis is highly instructive. We can do the analysis in terms of the number n of nodes and/or the number m of edges. The first thing to notice is that there aren't any simple "for i = 1 to n" type loops. But we can make these two observations:

- Every iteration of the outer while loop pops a node from the queue and puts it in the expanded set---so it can never get back onto the queue. Thus the outer while look can run for at most n iterations, since there are n nodes total.
- The inner for-loop iterates once for each out-neighbor of u, so it runs the degree of u number of times. The average degree deg(u) in a directed graph is just the number of edges divided by n, i.e. \( \frac{m}{n} \). (In an undirected graph, where you count each (u, v) just once, you get \( \frac{2m}{n} \).)

Thus the total number of iterations of both loops combined is at most \( n \cdot \frac{m}{n} = m \). Another way to view this fact is that in BFS, no edge ever gets traversed twice, so the total amount of "traversing" is at most the number of edges. If every individual line of code inside the loops were \( O(1) \) time, then you could conclude that BFS runs in \( O(m) \) time, which in the worst case of \( m = \Theta(n^2) \) edges makes \( O(n^2) \) time.
However, the test line `(!(expanded.contains(v) || visited.contains(v)))` is not elementary. Both containers can fill up to $\Theta(n)$ nodes. Now because `expanded` is a `Set`, the `contains(.)` method runs in $O(\log n)$ time at worst [we will see that a hash-table implementation claims $O(1)$ time, but this comes with some fine-print]. But lookup in a `Queue` doesn't promise better than $O(n)$ time.

There is a simple way we could avoid that time sink. We can expand the `Node` class to include a writable flag field, called "touched" say. Any visit would set the flag true, and popping the node from the queue would not unset it. So---assuming that nodes were all constructed with the flag false---the test lines would simply be

```scala
} else if (!v.touched) {
  visited.enqueue(v)
  v.touched = true
}
```

Whether we'd need `start.touched = true` earlier in the code too is a good study question. Then the running time for BFS becomes a clean $O(m)$. But---note that the infrastructure of the algorithm still requires random access of individual elements; it is not streamable.

**Depth-First Search**

What if we use a `Stack` instead of a `Queue`? We can run exactly the same code (with the "touched" update) but just changing the auxiliary data structure:

```scala
def classicDFS(start: Node, goal: Node): Boolean = {
  if (start == goal) { return true }
  //else
  var visited = new Stack[Node]()  //Scala Stack is mutable by default
  visited.enqueue(start)  //so queue is nonempty
  start.touched = true
  while(!visited.isEmpty) {
    val u = visited.pop()
    for (v <- u.nbhrs) {
      if (v == goal) {
        return true
      } else if (!v.touched) {
        visited.push(v)
        v.touched = true
      }
    }
  }  //exit of while loop means no more fresh nodes to visit/expand
```
The only difference is that now the visited nodes are treated in LIFO order. The running time analysis is entirely similar, because the count of popping from the data structure is the same.

The difference between DFS and BFS is IMHO best appreciated by sketching how each works when started from the root of a binary tree, with the goal node being one of the leaves.

BFS works in a slow-but-uniform way from top to bottom and left to right by row. DFS, however, right away jumps down the rightmost path, "barking down the wrong tree" so to speak. After popping the leaves 31 and 30 gives no new out-neighbors, the stack gets down to where it can pop and expand 14, which explores 28 and 29---again with no goal. Eventually the stack gets all the way back down to 2, expanded via the root, and after some more zigs and zags it finds the goal.