Space-efficient Iterative depth-first search

David Gries

Assume a graph with n nodes. In the worst case, the recursive dfs takes space O(n) because the depth of recursion could be O(n). That happens when the graph is just a doubly linked list.

To the right is the iterative depth-first search algorithm developed earlier, with a slight change. Earlier, we said, "visit u"; here, we say, "Mark u as visited." Somehow, we have to be able to mark nodes; we don't say how. It could be done using a boolean field in each Node object. If the nodes are numbered 0..n-1, it could be done using a boolean array. There are many ways to mark nodes.

```
/** Visit every node reachable along a path of un-
      visited nodes from node u in depth-first order.
      Precondition: u has not been visited */
public static void dfsIterative(Node u) {
   Stack s= (u) // not Java
   /* inv: All nodes (and only those nodes) that have
           to be visited are reachable along a path of
           unvisited nodes from some node in s. */
   while (s is not empty) {
      u= s.pop(); // Remove top stack node, put it in u.
      if (u is not visited) {
         Mark u as visited;
         for each neighbor w of u:  s.push(w);
      }
   }
}
```

This iterative dfs takes space $O(n^2)$ in the worst case. To see this, consider a complete undirected graph, such as K_5. In a complete undirected graph with n nodes, each node has n−1 neighbors.

When node u is first popped from the stack, its n−1 neighbors will be pushed onto the stack. The next time u is popped from the stack, n−2 of its neighbors will be pushed onto the stack (one neighbor has been visited). The next time, n−3 of its neighbors will be pushed onto the stack, and so on.

Thus (n−1) + (n−2) + ... + 1 nodes will have been pushed onto the stack, n−1 will have been popped, and the stack still contains $O(n^2)$ elements. The problem, of course, is that the stack contains many duplicates. We solve this problem by ensuring that each node is placed on the stack at most once.

We use a second way of marking a node. We mark it *discovered* when we first put it on the stack, and we put a node on the stack only if it has not been marked discovered. At this point look to the right at the addition to the invariant:

> (2) A node is marked "discovered" iff it is either on the stack or marked visited (not both).

The method begins by initializing the stack to contain only u and by marking u as discovered. This truthifies both parts of the loop invariant.

We discuss the repetend. By the invariant, the node that is popped and stored in u is marked discovered but has not been marked visited (since it was on the stack). Therefore, it is marked visited. Part (2) of the invariant is still true.

To truthify part (1) of the invariant, its undiscovered neighbors (those that are not on the stack and are not marked visited) have to be pushed onto the stack and marked discovered. That is the purpose of the loop over the neighbors of u.

Now, since each node is pushed on the stack only once, in the worst case, space O(n) is used.

```
/** Visit every node reachable along a path of un-
      visited nodes from node u in depth-first order.
      Precondition: u has not been visited */
public static void dfsIterative(Node u) {
   Stack s= (u) // not Java
   Mark u discovered;
   /* inv: (1) All nodes (and only those nodes) that have
           to be visited are reachable along a path of
           unvisited nodes from some node in s.
           (2) A node is marked "discovered" iff it is
           either on the stack or visited (not both). */
   while (s is not empty) {
      u= s.pop(); // Remove top stack node, put it in u.
      Mark u visited;
      for each neighbor w of u:
         if (w is not marked discovered) {
            s.push(w);
            Mark w discovered;
         }
      }
}
```