**Recursive DFS[[1]](#footnote-1)**

Assume we are using an adjacency-list representation of an undirected connected graph. In particular, there is a class Node, an object of which contains a linked list of its neighbors, as shown to the right. Let’s work with a complete graph: It has n nodes, and each node has n-1 neighbors. Letting m be the number of edges, we have: e = n(n-1) = n^2 – n, which is O(n^2).

**public** **class** Node {  
 LinkedList<Node> neighbors;

…

}

A recursive DFS procedure is shown to the right. Assuming no nodes have been visited, a call

/\*\* Visit all nodes reachable along a path of unvisited nodes from node u. Precondition: u is not visited. \*/

**public** **static** **void** dfs(Node u) {

Visit u;

**for** each neighbor w of u:

**if** (w is not visited) dfs(w);

}

dfs(u);

with u being any node of the graph will visit all nodes.

The depth of recursion is at most n. It cannot be more because only unvisited nodes are candidates for the recursive call dfs(w), and when that call occurs, w is immediately visited.

Because the graph is complete, the depth of recursion is n —from any node one can get to any other node. This means that the worst-case complexity for recursive DFS is O(n).

**Naïve iterative DFS**

/\*\* Visit all nodes reachable along a path  
 \* of unvisited nodes from node u. \*/

**public** **static** **void** dfs(Node u) {

Stack<Node> s;

s.push(u);

**while** (s.size() != 0) {

Node n= s.pop();

**if** (n is not visited) {

Visit n;

**for** each neighbor w of u:

s.push(w);

}

}

}

We also discussed an iterative DFS, in which the call stack is replaced by a stack that the method has to maintain. An iterative DFS appears to the right.

Since the graph is complete, each node has n-1 neighbors. Therefore when one node is popped from the stack, n-1 nodes are pushed onto it. So, after one iteration of the while loop, stack s contains n-1 nodes. After two iterations, it contains (n-2) + (n-1) = 2n-3 nodes. After three iterations, 2n-2 + (n-1) = 3n-3. You can see that the maximum size of the stack is O(n^2), which is O(e).

Thus, the recursive DFS takes space O(n) in the worst case while the naïve iterative DFS takes space O(n^2).

Problem with iterative DFS: It pushes all the neighbors of a node onto the stack. There is no need for this.

Instead push an Iterator (an object that takes constant space) for the neighbors onto the stack, with the meaning that all nodes enumerated by that Iterator have to be processed. Thus, stack s has type

/\*\* Return the next node from s (null if none) \*/

**public** **static** Node nextNode(  
 Stack<Iterator<Node>) {

**while** (s.size() != **null**) {

Iterator<Node> it= s.peek();

if (it.hasNext()) {

return it.next();

}

s.pop(); // no more nodes to enumerate

}

**return** **null**;

}

Stack<Iterator<Node>>

To keep the main algorithm simple, we write a method nextNode, which removes the next node from stack s and returns it. We present this method to the right. A loop is needed because one or more Iterators on the stack could have no more elements to enumerate.

Now we can write the iterative DFS as shown to the right. Each iteration of the loop processes the next node from stack s.

/\*\* Visit all nodes reachable along a path  
 \* of unvisited nodes from node u. \*/

**public** **static** **void** dfs(Node u) {

Stack<Iterator<Node>> s= **new** Stack<>();

s.push(u.neighbors.iterator());

Node no= nextNode(s);

// inv: no is the next node to process —if null, no more

**while** (no != **null**) {

if (no is not visited) {

Visit no;

s.push(no.neighbors.iterator();

}

no= nextNode(s);

}

}

This version has the same worst-case complexity as the recursion version: O(n) where n is the number of nodes.

**The space requirements of BFS**

BFS is written iteratively, like the iterative DFS. The difference is that is uses a queue instead of a stack.

Implement BFS using a queue of nodes, like the naïve DFS, and its space requirement in the worst case is O(M), or O(n^2) for a dense graph.

Implement BFS using a queue of Iterator<Node>, and it space requirement O(n), like the second iterative DFS.

1. This note is written fairly hurriedly in order to let everyone know about this topic. It may therefore have lots of errors. The code has not been checked. This will all be fixed later. [↑](#footnote-ref-1)