COS 226 Precept 8 Class Notes Fall 2023

Robert E. Tarjan

revised November 6 2023

1 November 3: Non-Recursive Depth-First Search

In lecture, Kevin presented depth-first search as a recursive algorithm but breadth-first search as an iterative one. The recursive definition of DFS, including numbering vertices in both preorder and postorder, is the following: DFS(to visit a vertex v): (1) Mark vertex v, and number v in preorder. (2) Recursively visit each unmarked vertex w adjacent from v. (3) Number v in postorder. The iterative definition of BFS is the following: BFS(to visit all vertices reachable from a vertex u): (1) Mark u, and initialize a queue Q to contain u. (2) Repeat until Q is empty: Deque the first vertex on Q, say v. For each unmarked vertex w adjacent from v, mark w and enqueue it on Q.

The question we considered is how to implement DFS without using recursion. In particular, suppose we replace the queue in BFS with a stack. Does this give a non-recursive implementation of DFS?

To help answer this question, let's consider what DFS does. At a minimum, we would like a numbering of the vertices in preorder and a numbering in postorder. The latter is what we need, for example, to produce a topological order of the vertices of a DAG (directed acyclic graph); namely, an order such that every arc is from a lower vertex to a higher one in the order. As discussed in lecture, for any DAG, the reverse of a DFS postorder is a topological order.

We call the algorithm obtained by replacing the queue in BFS with a stack *quick search*. Quick search visits each vertex twice, once when it adds the vertex to the stack and once when it removes the vertex from the stack. Does either order of visits give us either a DFS preorder or a DFS postorder? One immediate observation is that *neither* order is a DFS postorder, since x, the starting vertex of the search, is both added to and deleted from the stack first, but in any DFS postorder it is last.

One useful example is the graph with five vertices $1, 2, 3, 4, 5$ and arcs $1 \to 2, 1 \to 3, 2 \to 4, 3 \to 5$. Suppose we start a quick search from vertex 1. Depending on the order of 2 and 3 in the adjacent from list of vertex 1, the order of stacking will be $1, 2, 3, 4, 5$ or $1, 3, 2, 4, 5$, but the two possible DFS preorders on this graph are $1, 2, 4, 3, 5$ and $1, 3, 5, 2, 4$, and the two possible DFS postorders are $4, 2, 5, 3, 1$ and $5, 3, 4, 2, 1$. The stacking order does not give a DFS preorder, nor a DFS postorder, even in reverse.

Let's consider another example: the graph with vertices $1, 2, 3, 4$ and arcs $1 \to 2$, $1 \to 3$, $1 \to 4$, $2 \to 4$, $3 \to 4$. Depending on the order of $2, 3, 4$ on the list of vertices adjacent from 1, a quick search starting from 1 can stack the vertices $2, 3, 4$ in any order. Suppose it stacks them $4, 3, 2$. Then the unstacking order is $1, 2, 3, 4$. Neither this order nor its reverse is a possible DFS preorder: The only possible DFS preorders on this graph are $(1, 2, 4, 3)$, $(1, 3, 4, 2)$, $(1, 4, 2, 3)$, and $(1, 4, 3, 2)$. If on the other hand it stacks them $2, 3, 4$, then the unstacking order is $1, 4, 3, 2$. Neither this order nor its reverse is a possible DFS postorder: The only possible DFS postorders are $4, 2, 3, 1$ and $4, 3, 2, 1$.

We conclude that quick search as specified cannot mimic depth-first search. But we can modify it to do so. The first step is to mark a vertex only when it is popped from the stack. The algorithm becomes: To search from vertex u,do the following: (1) Initialize a stack S to contain u. (2) Repeat until S is empty: Pop the top vertex on S, say v. If v is unmarked, mark it, and for each unmarked vertex w adjacent from v, push w onto S.

This algorithm can push a vertex onto the stack many times, once for each arc to it. To implement the algorithm, we need to allow the stack to contain up to E vertices, rather than V, and we need to use a stack representation that allows a vertex to be on the stack in many places rather than in just one. But this algorithm does in fact mark vertices in DFS preorder.

Exercise: Prove this.

This is not yet enough, because it does not provide a DFS postorder, which is what we need to compute a topological order on a DAG. To obtain a DFS postorder, we make a further change to the algorithm: We delay popping a vertex until the entire depth-first search from it is finished. Specifically, a vertex is in one of three states: unmarked, premarked (previsited but not postvisited), or postmarked. Initially all vertices are unmarked. The algorithm is: To search from vertex u,do the following: (1) Initialize a stack S to contain u. (2) Repeat until S is empty: Let v be the top vertex on S. If v is unmarked, premark it, number it in preorder, and for each unmarked vertex w adjacent from v, push w onto S. If on the other hand v is premarked, pop it from S, postmark it, and number it in postorder.

We call this algorithm the *vertex-based implementation* of DFS.

Exercise: Prove that the vertex-based implementation of DFS correctly simulates the recursive version.

As already noted, this implementation of DFS has the disadvantage that the size of the stack can grow

to E. With a sufficiently powerful stack implementation (a doubly linked list), one can keep the stack size bounded by V. The idea is to avoid duplicate copies of a vertex on the stack by *moving* a vertex to the top of the stack instead of creating a new copy. This requires the ability to tell whether a vertex is on the stack and to find its position on the stack if it is.

Depth-first search has some other important properties. One is that a DFS is a local search that advances along and later retreats along each arc from a visited vertex. We can find the order of these advances and retreats using the vertex stack implementation, but doing so requires adding additional code. This is much more straightforward in the recursive version.

A final point is that a better way to implement DFS non-recursively is to more directly mimic what the recursive version is doing. A recursive call suspends an iteration over an adjacent from list, returning to the iteration once the recursive call is finished. We can simulate this non-recursively by maintaining a stack of the current positions in the adjacent from lists at which such iterations have been suspended. For details see R. E. Tarjan and U. Zwick, "Finding Strong Components Using Depth-First Search" on arXiv (also European J. Combinatorics, to appear).