CSci 4041: Algorithms and Data Structures
Graph Traversals
Part 2

Last revision April 19, 2020

Here's the agenda for this lecture.

Analyzing GRAPH-BREADTH-FIRST.
Fun with GRAPH-BREADTH-FIRST: finding shortest paths.
Traversing a graph depth-first.
Example of GRAPH-DEPTH-FIRST.
Analyzing GRAPH-DEPTH-FIRST.

Analyzing GRAPH-BREADTH-FIRST. Cormen analyzes what I called GRAPH-BREADTH-FIRST on page 597; this is a summary of what he says. Suppose we're traversing a graph G whose edges are in the set E and whose vertexes are in the set V.

The initialization loop at lines 02–04 runs $|V - 1|$ times. Since the loop's body runs in constant time, the entire loop runs in $\Theta(|V|)$ time. The remaining steps at lines 05–09 also run in constant time, so GRAPH-BREADTH-FIRST needs $\Theta(|V|)$ time for initialization.

Lines 10–19 do all the work. We'll assume that VISIT takes $O(1)$ time, so we'll ignore it. If we assume that DEQUEUE and ENQUEUE, each take $O(1)$ time, then all the queue operations together take $O(|V|)$ time, since each vertex is added to (and removed from) Q exactly once. Also, since the inner loop at lines 13–18 scans the adjacency list of each vertex u exactly once, and the sum of the lengths of all adjacency lists is $\Theta(|E|)$, the total time spent by the inner loop is $O(|E|)$.

When we add all these up, we get $O(|E| + |V|)$ as the total run time—so GRAPH-BREADTH-FIRST runs in time proportional to the size of the adjacency structure *G.Adj*. This is what we'd expect from any kind of traversal algorithm, which must visit each relevant part of whatever data structure it traverses.

Fun with GRAPH-BREADTH-FIRST. What can we do with a breadth-first graph traversal procedure? Suppose that u and v are vertexes in a graph G. We'll define the *length* of a *path* between u and v as the number of edges that we must follow to get from u to v. Some paths will be longer than others. If G has cycles, then a path may even have an infinite length, if we enter a cycle and go round and round without ever leaving it.

A procedure like GRAPH-BREADTH-FIRST can find a *shortest path* between u and v. (If there's more than one shortest path, then GRAPH-BREADTH-FIRST will find only one of them.) It's easy to see why this works. A breadth-first traversal never goes deeply into a graph if it can avoid doing so. As a result, a path found by a breadth-first traversal must be the least deep possible path—the shortest path. Of course this isn't a proof; it's hardly even an argument. Skeptics may want to look at Cormen pages 597–599, where he proves that a breadth-first traversal really does find a shortest path.

Now suppose we want a shortest path from the source vertex s to some other vertex v. We first call GRAPH-BREADTH-FIRST(G, s). If v is a vertex in G, then this defines $v.d$, the length of a shortest path from v back to s. It also defines $v.\pi$, an edge that takes us closer to s if we follow it. To see the shortest path from s to v, we can call the procedure PRINT-PATH which appears on page 601 of Cormen.

```
00 PRINT-PATH($s$, $v$)
01   if $v == s$
02     print $s$
03   else if $v.\pi ==$ NIL
04     print "No path from" $s$ "to" $v$ "exists."
05   else
06     PRINT-PATH($s$, $v.\pi$)
07     print $v$
```

Cormen's version of PRINT-PATH takes the graph G as an extra parameter. I've omitted G because PRINT-PATH never uses it. I guess Cormen thinks s and v will get lonely if they don't stay close to G.

If there is a path from s to v, then PRINT-PATH will print a series of vertexes s, u_1, u_2 ..., u_n, v, where the subscripted u's are zero or more vertexes that appear along the path from *s* to *v*. If there's no path from *s* to *v*, then PRINT-PATH goes as far as it can, then prints an error message at line 04, and stops.

PRINT-PATH works by recursively following π-links starting from *s*, until it gets to *v* at line 01, or until it runs out of vertexes at line 03. However, this visits vertexes in reverse, from *v* to *s*, which isn't what we want. To see the vertexes in the right order, it prints each vertex at line 07, after the recursive call at line 08 has returned. This is a common trick in recursive procedures for visiting the elements of a linked list in opposite order.

So what can we do with a shortest path from *s* to *v*? Here's an example. Suppose that each vertex in *G* represents a stage or state in the solution of a problem. The edges represent things we might do to help solve that problem. The source vertex *s* is the initial state, before we've tried to solve the problem. If a vertex *u* is reachable from *s* by following an edge, then *u* is a state that's closer to being a solution. We can solve the problem by starting from *s* and following edges until we get to a final vertex *v* in which the problem is solved. And to solve the problem most efficiently, we'll to follow edges along a shortest path from *s* to *v*. Some classic AI algorithms work this way.

Traversing a graph depth-first. We'll now look at a way to traverse a graph *depth-first.* Like a breadth-first traversal, a depth-first traversal visits vertexes in a graph. But where a breadth-first traversal was *cowardly,* never going deeply into the graph unless it had no choice, a depth-first traversal is *brave,* always going in as deeply as it can. Also, where a breadth-first traversal uses a queue to keep track of the vertexes it visits, a depth-first traversal uses a stack: in this case, the same stack that keeps track of recursive calls. That means a depth-first traversal is most easily written as a recursive procedure.

Cormen defines a procedure called DFS, which stands for *Depth First Search,* on page 604. However, like his earlier procedure BFS (*Breadth First Search*) it doesn't really search a graph, but rather traverses it. Things like that bother me more than they should, so I'll call my version GRAPH-DEPTH-FIRST.

```
00 GRAPH-DEPTH-FIRST(G)
01   for each u ∈ G.V
02     u.color = WHITE
03     u.π = NIL
04   time = 0
05   for each u ∈ G.V
06     if u.color == WHITE
07       GRAPH-DEPTH-FIRSTING(G, u)
```

It takes a graph *G* as its parameter at line 00. The loop at lines 01–03 initializes vertexes, in much the same way that GRAPH-BREADTH-FIRST did. It assigns the color WHITE to each vertex *u* at line 02, and sets *u*'s predecessor link π to NIL at line 03. The color attribute is used in the same way as GRAPH-BREADTH-FIRST used it—to keep from being trapped in cycles.

Unlike GRAPH-BREADTH-FIRST however, it doesn't begin traversing *G* from a single source vertex *s.* Instead, it uses a loop in lines 05–07 to traverse *G* from every vertex *u* that it hasn't encountered before. Line 06 tests if *u.color* is WHITE, meaning that *u* hasn't been encountered, and line 07 actually begins the traversal by calling the helper GRAPH-DEPTH-FIRSTING. (If I have a procedure *xxx* then I like to call its helper *xxxing.*) GRAPH-DEPTH-FIRSTING actually does all the work, traversing *G* with *u* as a source vertex.

I haven't mentioned yet that Cormen uses a global variable called *time,* initalized at line 04, to compute what he calls *timestamps.* (The only way we know it's global is that Cormen says it is in the text.) Imagine that as *G* is being traversed, a clock ''ticks'' each time some significant event occurs. The variable *time* holds the number of ticks that have occurred so far.

The helper GRAPH-DEPTH-FIRSTING looks like this. As stated before, it takes a graph *G* and a source vertex *u* as its parameters in line 00. It performs a depth-first traversal starting from *u.*

```
00 GRAPH-DEPTH-FIRSTING(G, u)
01   time = time + 1
02   u.d = time
03   u.color = GRAY
04   VISIT(u)
05   for each v ∈ G.Adj[u]
06     if v.color == WHITE
```

```
07         v.π = u
08         GRAPH-DEPTH-FIRSTING(G, v)
09   u.color = BLACK
10   time = time + 1
11   u.f = time
```

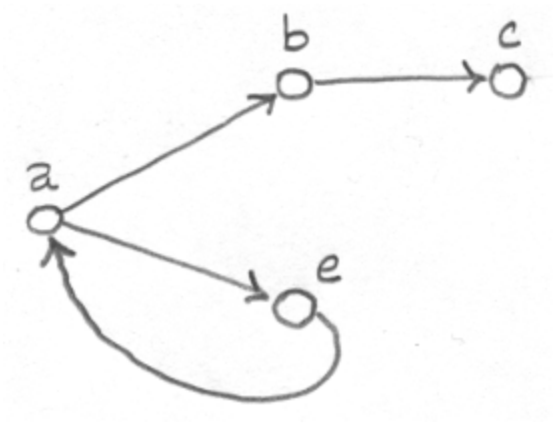
The first thing GRAPH-DEPTH-FIRSTING does is change some attributes of vertex *u.* Each vertex *u* has an attribute *u.d* that is its *discovery time,* the number of clock ticks that have occurred when we first encounter *u.* It also has an attribute *u.f* that is its *finish time,* the number of clock ticks that have occurred when we've finished with *u.* At line 01, we increment *time* to get the next clock tick, and at line 02, we set *u.d* to *time,* recording its discovery time.

When GRAPH-DEPTH-FIRSTING is called, *u.color* is WHITE. At line 03, we set *u.color* to GRAY, meaning that we've encountered *u,* but are not yet finished with it. This keeps us from being trapped in a cycle that involves *u,* so we won't call GRAPH-DEPTH-FIRSTING on *u* again. We also call VISIT on *u* at line 04, to show that this is where *u* is first visited.

The loop at lines 05–08 uses the adjacency structure *G.Adj* to find all vertexes *v* that are connected to *u* by a single edge. Inside the loop, we test at line 06 if *v.color* is WHITE, meaning that we've never encountered *v* before. If we haven't, then at line 07 we set *v.*π to point from *v* back to *u,* just as we did in GRAPH-BREADTH-FIRST. We then call GRAPH-DEPTH-FIRSTING at line 08 to continue traversing *G,* but this time starting from *v.* Because we're doing this by a recursive call, it means we're descending deeply and bravely into *G,* just as we'd expect.

When the loop finishes, we set *u.color* to BLACK at line 09, meaning that we're finished with *u.* We also get the next clock tick at line 10 by incrementing *time,* and set *u.f* to *time* at line 11 to record the time that we finished with *u.*

Example of GRAPH-DEPTH-FIRST. Here's a simple example of how GRAPH-DEPTH-FIRST works. Cormen has a more complex example on page 605. I've tried to design this example to how depth-first traversal differs from breadth-first traversal, and to show how GRAPH-DEPTH-FIRST avoids being trapped in cycles. Suppose we want to traverse the directed graph *G* depth-first.

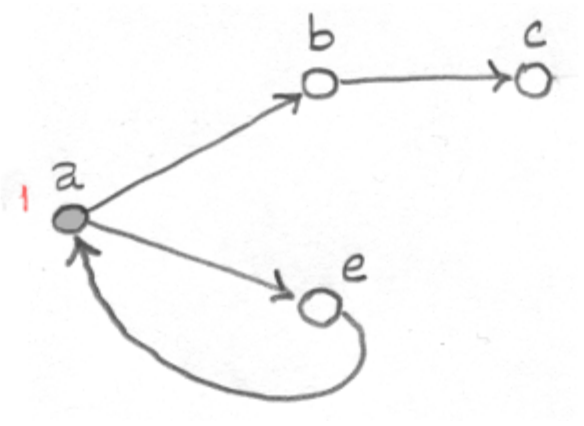


The graph *G*.

We start by calling GRAPH-DEPTH-FIRST, which runs the loop at lines 01–03 to initialize *G.* For each vertex *u*, we set *u.color* to WHITE, and *u.*π to NIL. We won't show π links yet. We'll also set the global variable *time* to 0.

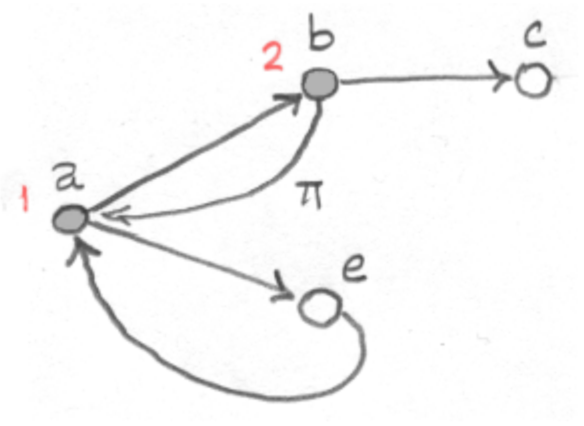
We now enter the loop at lines 05–07, which picks a start vertex *u* from the set *G.V.* Since sets are unordered, we don't know which vertex it might choose. To simplify the example, let's say it chooses vertex *a.* (The procedure will still work even if it chose a different vertex.) Since *a.color* is WHITE, meaning that we've never encountered *a* before, we call GRAPH-DEPTH-FIRSTING(*G, a*) to begin traversing *G* from *a.* (We're now one recursive call deep.)

At lines 01–02 of GRAPH-DEPTH-FIRSTING, we reset *time* to 1, and set *a.d* to 1. At line 03, we change *a.color* to GRAY, which will keep us from being trapped in a cycle involving *a.* At line 04, we call VISIT on *u.* Showing *d* attributes in red, the graph now looks like this.



We now enter the loop at lines 05–08, which finds each vertex *v* that's accessible by an edge from *a.* This works because *G.Adj*[*a*] is a linked list containing *b* and *e.* However, since the vertexes in the list aren't in any specific order, we don't know which one will be found. For simplicity, let's suppose we find vertex *b.* Since *b.color* is WHITE, meaning we haven't encountered *b* before, we set the back pointer *b.*π to *a* at line 07, and then recursively call GRAPH-DEPTH-FIRSTING(*G*, *b*) at line 08 to continue traversing from *b.* (We're now two recursive calls deep.)

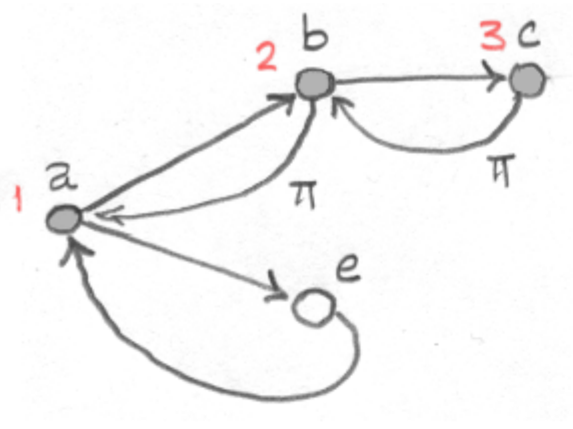
The recursive call sends us back to lines 01–04. We reset *time* to 2, set *b.d* to 2, set *b.color* to GRAY, and call VISIT(*b*). The graph now looks like this.



We enter the loop at lines 05–08 again, trying to find a vertex that's accessible by an edge from *b.* There's only one, the vertex *c.* Since *c.color* is WHITE, we set the back pointer *c.*π to *b* at line 07, and then recursively call GRAPH-DEPTH-FIRSTING(*G*, *c*) at line 08 to continue traversing from *c.* (We're now three recursive calls deep.)

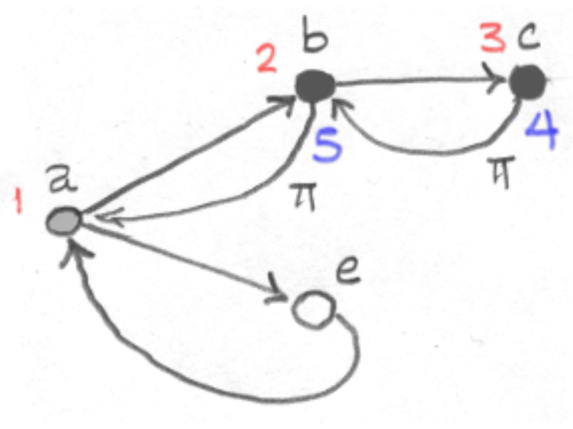
By the way, what just happened shows the difference between breadth-first and depth-first traversals. If we were traversing *G* breadth-first, then we would have chosen *e* instead of *c.* This is because breadth-first traversal, the coward, won't go deeply into the graph unless it has to. However, depth-first traversal, bravely descends as deeply as it can, going from *b* to *c.*

The recursive call sends us back to lines 01–04. We reset *time* to 3, set *c.d* to 3, set *c.color* to GRAY, and call VISIT(*b*). The graph now looks like this.



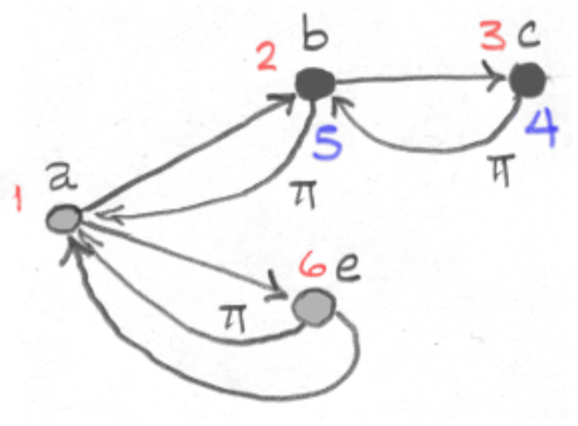
We enter the loop at lines 05–08 again, trying to find a vertex that's accessible by an edge from *c,* but there isn't one, so we fall through the loop, and our recursion terminates. GRAPH-DEPTH-FIRSTING sets *c.color* to BLACK at line 09, resets *time* to 3 at line 10, sets *c*'s finish time *c.f* to 3 at line 11, then returns. (We're now two recursive calls deep.)

We return from the call to GRAPH-DEPTH-FIRSTING at line 08, inside the loop in lines 05–08 that's looking for vertexes which are accessible by an edge from *b.* But there are no more—the only one was *c*—so we exit the loop, ending up at line 09, where we set *b.color* to BLACK. At line 10, we reset *time* to 5, set *b.f* to 5, and return again. (We're now one recursive call deep.) The graph looks like this after the two calls have returned, showing the finish times in blue.

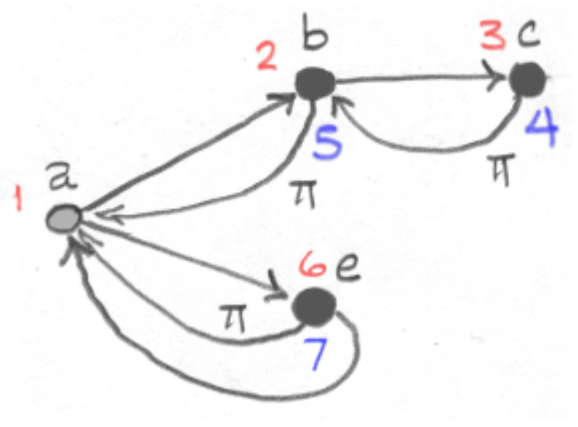


That last return was from line 08 in the loop of lines 05–08 in the first call to GRAPH-DEPTH-FIRSTING. In that loop, we were trying to find vertexes that could be reached by following edges from *a.* We'd already found *b,* and now the second time through the loop, we find *e.* Since *e.color* is WHITE at line 06, we set *e.*π to *a* at line 07, and call GRAPH-DEPTH-FIRSTING(*G*, *e*) recursively. (We're two recursive calls deep again.)

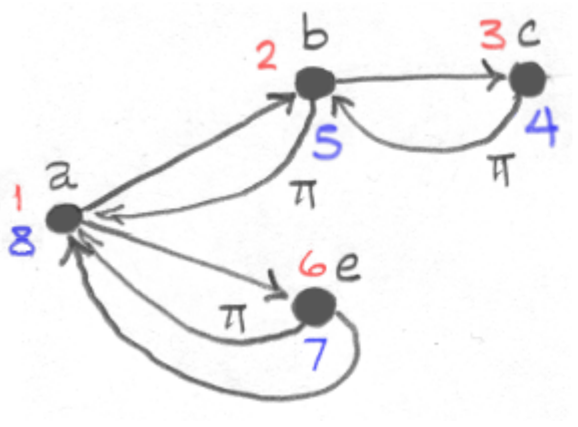
That recursive call sends us back to lines 01–04. We reset *time* to 6, set *e*'s discovery time *e.d* to 6, and set *e.color* to GRAY. The graph now looks like this.



We enter the loop in lines 05–08, trying to find vertexes that are accessible from *e* by an edge, but there aren't any, so we fall through the loop to lines 09–11. There, we set *e.color* to BLACK, reset *time* to 7, set *e.f* to 7, and return. (We're one recursive call deep.) The graph looks like this.



When we return, we end up back in the loop of lines 05–08 in the first call to GRAPH-DEPTH-FIRSTING, where we were trying to find vertexes *v* accessible from *a* by an edge. We try to go around that loop again, but there are no more vertexes that can be reached from *a*, so the loop stops. At lines 09–11, we set *a*'s color to BLACK, reset *time* to 8, set *a.f* to 8, and return. (We're zero recursive calls deep.) We leave the graph looking like this.



We've now traversed the entire graph, turning all its vertexes BLACK, but GRAPH-DEPTH-FIRST doesn't know that yet. It knows only that it's traversed the part of the graph accessible from vertex *a*, but there may be other vertexes that need to be investigated. The return takes us back inside the loop of lines 05–07 in GRAPH-DEPTH-FIRST, which is trying to find those vertexes. It look at vertexes *b*, *c*, *d*, and *e*, but it finds at line 07 that none of them are WHITE. As a result, it can make no more recursive calls to GRAPH-DEPTH-FIRSTING, so GRAPH-DEPTH-FIRST terminates, and we're done.

There was a cycle involving the vertexes *a* and *e* in the graph *G*, but GRAPH-DEPTH-FIRST didn't get caught in it. It used the same coloring mechanism that GRAPH-BREADTH-FIRST did, with WHITE vertexes that have never been encountered, GRAY vertexes that have been encountered but are still being investigated, and BLACK vertexes that are completed.

Analyzing GRAPH-DEPTH-FIRST. We'll conclude by doing a quick analysis of GRAPH-DEPTH-FIRST, a summary of Cormen's analysis on page 606. Suppose *G* is a graph whose vertexes are in the set *V* and whose edges are in the set *E*. The procedure GRAPH-DEPTH-FIRST has two loops, each of which look at each vertex in *V*, so together they take $\Theta(|V|)$ time. However, the interesting part of the algorithm occurs in GRAPH-DEPTH-FIRSTING.

We call GRAPH-DEPTH-FIRSTING once for each vertex *u* in *V*, either directly from GRAPH-DEPTH-FIRST, or indirectly during a recursive call to GRAPH-DEPTH-FIRSTING itself. During each call to GRAPH-DEPTH-FIRSTING, the loop in lines 05–08 executes $|G.Adj[u]|$ times. If we add up all these times, the total must be $\Theta(|E|)$, since all *G*'s edges are represented in *G.Adj*.

As a result, the total run time for GRAPH-DEPTH-FIRST is $\Theta(|V| + |E|)$. Just as with GRAPH-BREADTH-FIRST, this is what we would expect from a traversal algorithm that visits all relevant parts of a data structure.