CSCI 4041: Algorithms and Data Structures
Graph Traversals
Part 1

Last revision April 13, 2020

Here's the agenda.

Terminology.
Traversing a tree depth-first.
Traversing a tree breadth-first.
Dealing with cycles and paths.
Traversing a graph breadth-first.
Example.

Both this lecture and the next will deal with graph traversal algorithms. We'll first discuss how tree traversals and graph traversals are alike, and how they're different. We'll also discuss a procedure that traverses a graph breadth-first. Most of this (except for the part about trees) can be found on pages 594–597 of Cormen.

Terminology. We'll start by reviewing some terminology. We say that we *visit* an object if we access that object somehow, and maybe perform some computation with it. For example, we might say that we visit an element of an array, or that we visit a vertex (node) in a tree. If we say we visit an object, then we don't have to say why we're visiting it, or what computation we'll perform with it.

We say that we *traverse* an object if we visit all parts of it in some systematic order. For example, if we traverse an array, it might mean that we visit each of its elements in order of their indexes. If we traverse a tree, it might mean that we visit each of its vertexes (nodes) in order of their depths.

In this lecture, we'll construct a procedure that traverses a graph. It starts with an initial vertex, called a *source vertex,* then follows edges from that vertex to other vertexes, and follows edges from those to still other vertexes, etc. The procedure continues in this way until all possible vertexes are visited. Note, however, that some vertexes may not be reachable from the source vertex—we'll see what to do about that later.

Each time our procedure visits a vertex v, it calls a procedure VISIT on v. We don't know what VISIT does, and we don't care. All we know is that the call to VISIT tells where v is visited.

Traversing a tree depth-first. Before constructing a procedure to traverse a graph, let's look at a simpler problem —constructing procedures to traverse a *tree.* You've seen similar procedures in whatever data structures course you took before this one. Recall that a tree is just a special case of a graph—it's a graph with no cycles. That means some things about traversing trees will be relevant to traversing graphs.

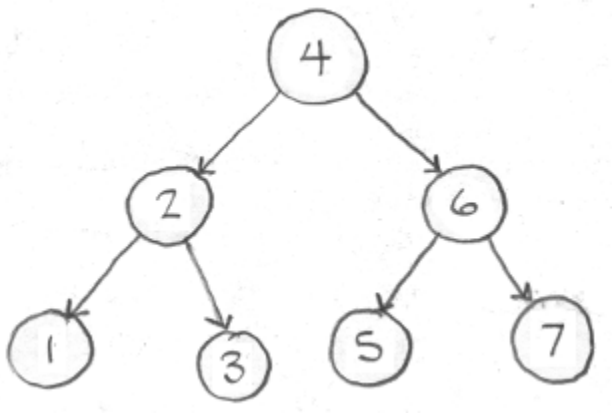
We suppose that our trees are binary trees, built from nodes that have two slots. If p points to a node, then the slot $p.left$ points to p's left subtree, and the slot $p.right$ points to p's right subtree. An empty tree is represented as NIL.

There are at least two different ways to traverse a binary tree. One is a *depth-first traversal* that uses a recursive backtracking algorithm. Depth-first traversal starts at the root of the tree, then moves down, following *left* and *right* edges, as deeply as it can go. When it reaches a vertex from which it can't descend any deeper, because the vertex's *left* and *right* slots are NIL, then it visits that vertex. It then backtracks, returning to the next most deeply nested vertex, and repeats the process. It continues in this way until all vertexes are visited.

The procedure TREE-DEPTH-FIRST shows one way to do depth-first traversal on a binary tree. (It's not in Cormen, probably.) It takes a parameter r that is either NIL, or that points to the root vertex of the tree. You may recognize it as an *inorder traversal* of r.

```
00 TREE-DEPTH-FIRST(r)
01   if r ≠ NIL
02     TREE-DEPTH-FIRST(r.left)
03     VISIT(r)
04     TREE-DEPTH-FIRST(r.right)
```

Suppose r points to the root of a complete binary tree with seven nodes, like the one shown below. The numbers in the nodes tell us the order in which TREE-DEPTH-FIRST visits its vertexes. Node 1 is visited first, then node 2, then node 3, etc., with node 7 visited last.



One way to think about depth-first traversal is that it's *brave.* It fearlessly descends as deeply as it can, and stops only when it can't descend any farther. It then backtracks and tries to descend as deeply as it can again, but in a different way.

Traversing a tree breadth-first. Another way to traverse a binary tree is called *breadth-first traversal.* Like depth-first traversal, it visits all the vertexes in the tree. But unlike depth-first traversal, breadth-first traversal is *cowardly.* It's too scared to descend deeply into the tree, so it does that only when there's no alternative.

Depth-first traversal uses a *stack* to keep track of how it visits vertexes. You didn't see the stack explicitly in TREE-DEPTH-FIRST because it's the same stack that's used to implement recursion. Breadth-first traversal uses a *queue* instead of a stack. (It's not a priority queue, just an ordinary queue.) Recall that a queue is an finite ordered sequence in which elements are added at the rear and removed from the front. When an element is added, we say that the element is *enqueued,* and when an element is removed, we say that the element is *dequeued.*

The procedure TREE-BREADTH-FIRST performs breadth-first traversal on a binary tree. (I don't think it's in Cormen either.) Its parameter r points either to NIL or to the root of the tree, and it uses a queue Q. Since TREE-BREADTH-FIRST is a little more complicated than TREE-DEPTH-FIRST, we'll talk about it in a little more detail.

```
00 TREE-BREADTH-FIRST(r)
01   if r ≠ NIL
02     Q = ∅
03     ENQUEUE(Q, r)
04     while Q ≠ ∅
05       v = DEQUEUE(Q)
06       VISIT(v)
07       if v.left ≠ NIL
08         ENQUEUE(Q, v.left)
09       if v.right ≠ NIL
10         ENQUEUE(Q, v.right)
```

At line 01, TREE-BREADTH-FIRST tests if r is the empty tree NIL. If the tree is empty, then it does nothing. The rest of the discussion assumes r isn't empty.

At line 02, it creates a new empty queue, called (what else?) Q. Following Cormen's conventions for pseudocode, you can tell it's a queue because it's called Q, a pun. Cormen also writes an empty queue as ∅.

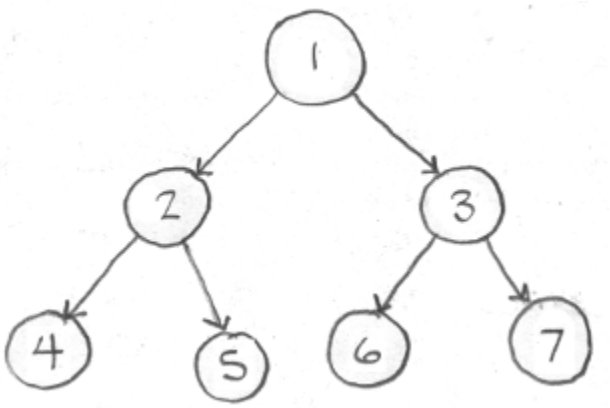
At line 03, it adds r to Q. We can now see what Q is used for: it contains the vertexes of r that have been seen, but have not yet been visited.

Lines 04–10 are a loop that runs while Q is not empty. We know it runs at least once, because Q starts out with one element r in it. Inside the loop, at line 05, we remove a vertex v from Q, and at line 06, we visit v.

There may be more vertexes to be visited. At line 07, we test if v's left subtree is empty. If it's not, then at line 08 we add

the root vertex of the left subtree, *v.left*, to the rear of *Q*. Similarly, at line 09, we test if *v*'s right subtree is empty. If it's not, then at line 10 we add the root vertex of the right subtree, *v.right*, to the rear of *Q* also. In either case, if we add a new vertex to *Q*, then the loop will keep running. It will continue until we can't add any more new vertexes to *Q*.

Now suppose *r* points to the root of a complete binary tree with seven nodes, the same as the one we used with TREE-DEPTH-FIRST. The numbers in the nodes tell us the order in which TREE-BREADTH-FIRST visits its vertexes. Node 1 is visited first, then node 2, then node 3, etc., with node 7 visited last.



Just as we'd expect, TREE-BREADTH-FIRST visits nodes starting with the root, moving left to right, and descending into the tree as shallowly as it can.

Dealing with cycles and paths. One reason that TREE-DEPTH-FIRST and TREE-BREADTH-FIRST work is that trees don't have cycles. That means if we start at a node *p*, then we can't get back to *p* again by following *left* and *right* slots. If we could, then either procedure wouldn't terminate if it entered a cycle containing *p*. So if we want to write a procedure that traverses a graph, we must have a way to escape from cycles. How can we do that?

Let's think of an analogy. Suppose you're trapped in an underground maze, in the dungeon of a mysterious castle. You need to find your way out. Since every room of the maze looks like every other room, you could end up walking in circles from room to room. One way to avoid this is to spray-paint a big red *X* on the wall every time you enter a new room. If you enter a room, and you see an *X* on the wall already, then you know you've been there before, so you should go somewhere else.

You might also carry a big ball of string. You anchor one end of the string to the room you start from (somehow). Then you reel the string out behind you as you walk from room to room. If you ever escape from the maze, then you could follow the string to find the path you took to get out. That doesn't help you avoid cycles, but it might help other people avoid being trapped in the same maze.

Algorithms that traverse graphs do similar things. They mark vertexes they've visited in the past, so they don't visit them again. They also record the sequence of vertexes they visit, to produce a path through the graph. They don't use paint and strings though.

Traversing a graph breadth-first. Our goal for this lecture is to design a procedure I'll call GRAPH-BREADTH-FIRST. It traverses a graph breadth-first, visiting all its vertexes. Cormen has a very similar procedure on page 595, but he calls his procedure BFS, which is short for Breadth-First Search.

He shouldn't have called it that. Cormen's BFS procedure doesn't *search* the graph, it just *traverses* it! Normally when we talk about a search procedure, we mean that it searches *for* something. For example, we might use linear search to look for an element in an array and stop when we find the element we want. But that's not what a traversal does! It visits every element, without looking for anything.

We should call things by their right names, so my version of Cormen's procedure is called GRAPH-BREADTH-FIRST instead of BFS. It traverses a graph *G* breadth-first, using the adjacency structure *G.Adj* that we discussed last time. (We could also traverse a graph using the adjacency matrix *G.M*, but we're not doing that now.) That means that vertexes are represented as small positive integers.

Before we discuss how GRAPH-BREADTH-FIRST works, you should know that it depends on having extra attributes in each vertex. If *v* is a vertex, then it has attributes *v.color*, *v.d*, and *v.*π (the Greek letter *pi*). Cormen doesn't say how these attributes are somehow attached to integers. He doesn't have to, he just assumes that they are. That's one of the fun things

about pseudocode.

The attribute *v.color* is used like the *X*'s on the walls of the maze—it keeps us from being trapped in cycles. It has one of three values: WHITE, GRAY, or BLACK. If *v.color* is WHITE, then it means we've never seen *v* before. If *v.color* is GRAY, then we've seen *v* before, but not yet visited it—we will in the future. If *v.color* is BLACK, then *v* has been visited, and will never be visited again. The color names suggest that all vertexes start out WHITE, then gradually darken to GRAY, and finally turn BLACK.

In the attribute *v.d*, the *d* is short for *distance.* It isn't like anything we used in the maze (the analogy isn't perfect). As we said before, when we traverse a graph, we start at a source vertex *s.* The value of *v.d* tells how many edges we had to follow in order to get from *s* to *v*, so it's the *distance* from *v* to *s.*

In the attribute *v*.π, the π is short for *predecessor* (or maybe *πredecessor?*). It's another vertex, and it's used something like the ball of string in the maze. Suppose that we're at vertex *v,* and we want to get back to the start vertex *s.* If we go from vertex *v* to vertex *v*.π, and from there to vertex (*v*.π).π, and then to ((*v*.π).π).π, etc., then eventually we'll find our way back to *s.*

By the way, we don't really need *v.color*, *v.d* and *v*.π to traverse a graph. All we really need is some way to mark a vertex *v* that we've already visited, so we don't visit it again. We could do that with a simple Boolean attribute like *v.mark,* which is TRUE if we've visited *v* and FALSE otherwise. (This will be the subject of a future homework assignment.) Cormen has all these vertex attributes because he'll use them for other things later—and we might, too. With that warning in mind, here's GRAPH-BREADTH-FIRST.

```
00 GRAPH-BREADTH-FIRST(G, s)
01   for each u ∈ G.V − { s }
02     u.color = WHITE
03     u.d = ∞
04     u.π = NIL
05   s.color = GRAY
06   s.d = 0
07   s.π = NIL
08   Q = ∅
09   ENQUEUE(Q, s)
10   while Q ≠ ∅
11     u = DEQUEUE(Q)
12     VISIT(u)
13     for each v ∈ G.Adj[u]
14       if v.color == WHITE
15         v.color = GRAY
16         v.d = u.d + 1
17         v.π = u
18         ENQUEUE(Q, v)
19     u.color = BLACK
```

In line 00, *G* is the graph we want to traverse. It has a set of vertexes *G.V* and a set of edges *G.E*, although we won't use *G.E* here. It also has an adjacency structure *G.Adj*, which we discussed in the previous lecture. The vertex *s* is the source vertex we'll start traversing from, so *s* ∈ *G.V.*

In lines 01–04, we have a loop where *u* is set to each vertex in *G.V* except *s,* one at a time, in an unspecified order. This loop initializes all vertexes except *s.* Cormen often uses a **for-each** loop when he wants to iterate over a sequence of objects, but he doesn't care how he does it. Recall that the programming language Python has a similar loop.

In line 02, we set *u*'s color to WHITE, which means we've never encountered *u* before. In line 03, we set *u*'s distance from *s* to ∞ (recall that we could do this without ∞ if we were clever enough), which means we don't know its distance from *s.* And in line 04, we set *u*'s predecessor to NIL, which means it has no predecessor yet. We may change these attributes later.

In lines 05–07, we initialize the source vertex *s.* At line 05, we set *s*'s color to GRAY, which means that we've encountered *s,* but have not yet visited it. At line 06, we set *s*'s distance to 0, since it's at zero distance from itself. And at line 07, we set

s's predecessor to NIL, since it has no predecessor. We'll change *s*'s color later.

We're still initializing. At line 08, we let *Q* be an empty queue, as we did in TREE-BREADTH-FIRST. At line 09, we add the source vertex *s* to *Q*, so we can visit it later. By convention, whenever we add a vertex to *Q*, its color will always be GRAY, which means we've encountered it, but have not yet visited it. Now initialization is complete, and we can begin traversal.

The outer loop at lines 10–18 does all the work. It's like the loop at lines 04–10 in TREE-BREADTH-FIRST, because it removes vertexes from *Q*, visits them, and maybe adds other vertexes back to *Q* again, until *Q* is empty. At line 11, we remove a GRAY vertex *u* from *Q* and visit it at line 12.

The vertex *u* may be connected to other vertexes *v* by edges, so we add them to *Q* so we can visit them later. The inner loop at lines 13–18 does this. Cormen uses a **for-each** loop that sets *v* to each vertex in *G*.Adj[*u*], a list of vertexes to which *u* is connected by an edge. This happens in arbitrary order.

At line 14 within the inner loop, we test if *v*'s color is WHITE. That keeps us from being trapped in a cycle. If *v.color* isn't WHITE, then we do nothing with *v.* Why? If *v.color* is GRAY, then it's already in *Q* so we can visit it later, and we must not add it to *Q* again. If *v.color* is BLACK, then we're done with it, so we also must not add it to *Q* again.

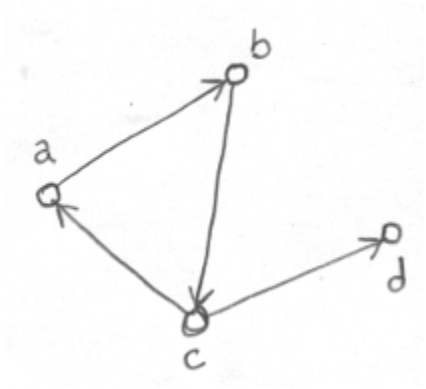
Let's assume *v.color* is WHITE. At line 15 within the inner loop, we change *v.color* to GRAY. Since we followed an edge from *u* to get to *v,* we set *v*.π to *u* at line 17. For the same reason, the number of edges from *v* to *s* must be one more than the number of edges from from *u* to *s,* so at line 16 we set *v.d* to *u.d* + 1. Finally, we add *v* to *Q* and go around the inner loop again.

When the inner loop of lines 13–18 finishes, we've done everything we can with the vertex *v,* so we never want to see it again. We make sure of that by setting *u.color* to BLACK. This works because only WHITE vertexes can have their color changed to GRAY and be added to *Q.*

Once we've done that, we go around the outer loop of lines 10–19 again, getting the next vertex *u* from *Q*. We continue this way until *Q* becomes empty, the outer loop stops, and GRAPH-BREADTH-FIRST terminates.

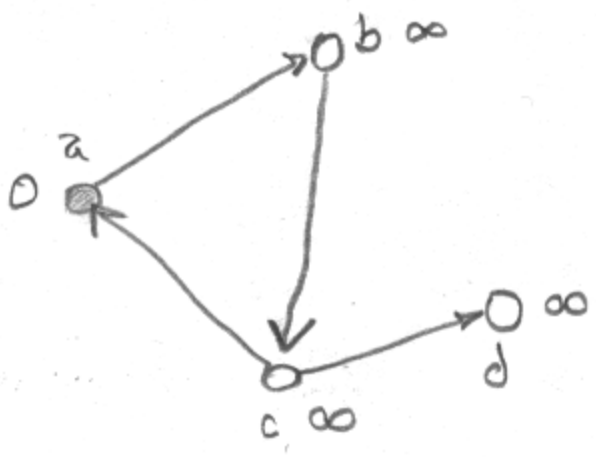
An example of breadth-first traversal. Of course all that was perfectly clear! Maybe an example will make it even clearer. Cormen has an example on top of page 596, but we'll do a simpler one here.

We'll use this graph (call it *G*) from a previous lecture. It has a cycle from *a* to *b* to *c,* so we can show that GRAPH-BREADTH-FIRST isn't bothered by cycles. And *G* is a directed graph, but GRAPH-BREADTH-FIRST also works on undirected graphs.

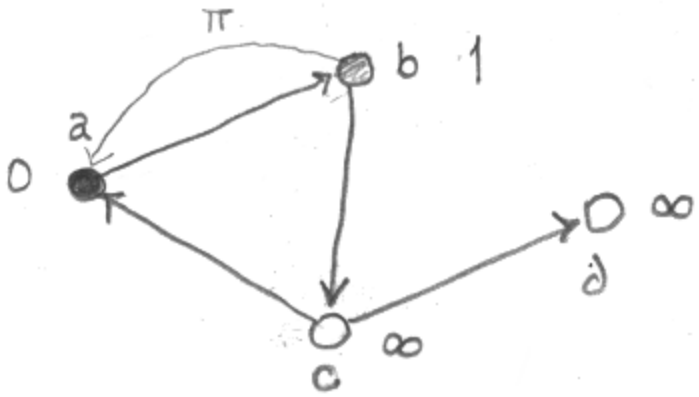


The graph *G*.

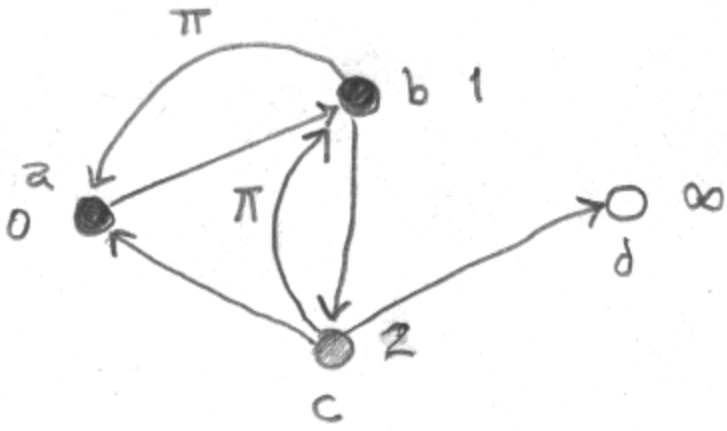
We'll call GRAPH-BREADTH-FIRST(*G, a*) to traverse *G* starting from the source vertex *a.* After initialization, *a.color* is GRAY, *a.d* is 0, and *a*.π is NIL. Every other vertex has its *color* attribute set to WHITE, its *d* attribute set to ∞, and its π attribute set to NIL. The queue *Q* contains one vertex *a.*



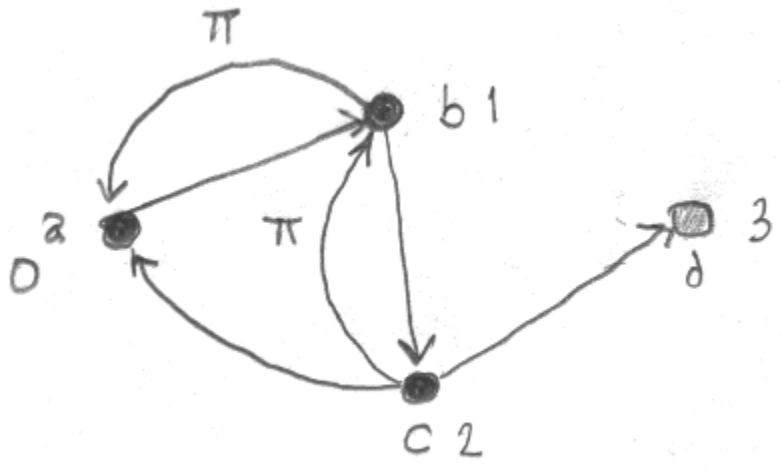
The first time through the outer loop, we remove *a* from *Q* and visit *a.* The vertex *a* is connected only to the vertex *b,* so the inner loop runs once with *v* set to *b.* Since *b.color* is WHITE, we change *b.color* to GRAY, *b.d* to 1, and *b.*π to *a.* We also add *b* to *Q.* At that point the inner loop finishes, so we set *a.color* to BLACK, meaning that we're done with *a.*



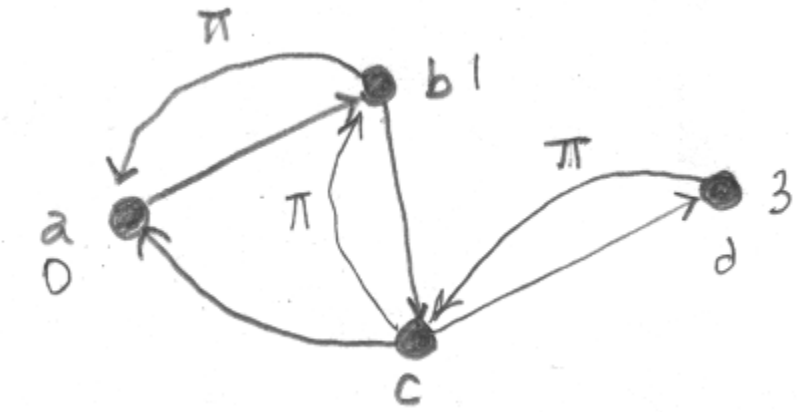
We now go through the outer loop a second time, removing *b* from *Q* and visiting *b.* The vertex *b* is connected only to the vertex *c,* so the inner loop runs once with *v* set to *c.* Since *c.color* is WHITE, we change *c.color* to GRAY, *c.d* to 2, and *c.*π to *b.* We also add *b* to *Q.* The inner loop then finishes, so we set *b.color* to BLACK. We're done with *b.*



The third time through the outer loop, we remove *c* from *Q*, and visit *c*. The vertex *c* is connected to vertexes *a* and *d*, so the inner loop runs twice, first with *v* set to *a*. However, *a.color* is BLACK, so nothing happens. When the inner loop runs again, *v* is set to *d*. Since *d.color* is WHITE, we change *d.color* to GRAY, *d.d* to 3, and *d*.π to *c*. We also add *c* to *Q*. The inner loop then finishes, so we set *c.color* to BLACK, and we're done with *c*.



We're almost done, but GRAPH-BREADTH-FIRST doesn't know it yet. The fourth time through the outer loop, we remove *d* from *Q*, and visit *d*. The vertex *d* isn't connected to any other vertex, so we fall through the inner loop without it doing anything. We then set *d.color* to BLACK.



When we try to execute the outer loop again, we find that *Q* is ∅, so it terminates. We've now visited each vertex in *G*.

We can conclude a few things from this example. GRAPH-BREADTH-FIRST didn't become trapped in the *a*–*b*–*c* cycle because of the way vertexes were colored, and the way their colors were checked before adding them to *Q*. We have a path from *d* to *c* to *b* to *a* by following the π attributes, which records how vertexes were visited. Sadly, however, the example is too small to show that we visited vertexes breadth-first. Unbelievers are invited to look at the much more complex example on page 596 of Cormen.

Next time, we'll see what can be done with all that extra information stored in the vertex attributes. We'll also start looking at a procedure to traverse graphs depth-first.