ICS 46 Spring 2022
Notes and Examples: Graph Traversals

Traversing a graph

Previously, we learned about tree traversals, of which we saw two main variants: *depth-first traversals* and *breadth-first traversals*. These kinds of traversals vary in terms of the order in which they visit the nodes in the tree:

- Depth-first traversals visit nodes by following one path in the tree as far as it will go, then backtracking and trying a different path.
- Breadth-first traversals visit nodes in the order of how far they are away from the root.

Graphs resemble trees in at least some ways, so it stands to reason that we might be able to use similar approaches to traverse them. Of course, we've also seen that graphs differ from trees in some fairly significant ways. Let's consider how each kind of traversal might work on a graph, and what we might have to do to tweak the tree traversal algorithms so that they would work on a graph instead.

Depth-first graph traversals

Let's consider again how we do a depth-first traversal of a tree. A sketch of the basic algorithm looks like this.

```
DFT(Tree t):
    visit(root of t)
    for each subtree s of t:
        DFT(s)
```

If we want to do something similar in a graph, we'll need to make some adjustments. There are three things that will need to change:

- One obvious adjustment is that we begin a depth-first tree traversal at a specially-designated node called the root. But where is the root of a graph? It turns out that graphs, in general, have no such notion; no one vertex is considered any more "special" than any other. But if no vertex is particularly special, perhaps it doesn't matter where we start.
- At each step, we'll need two parameters: the whole graph and the vertex where we currently are. (Why we don't need both in a tree is because trees have a root and subtrees that are trees, so passing a tree is enough to keep track of our current position; at any given time, we're at the root of some subtree.)
- Instead of looping over the subtrees of some tree, we'll instead loop over the vertices that we can reach via one outgoing edge from the current vertex.

Our first attempt

Putting those ideas together, we arrive at the following sketch of a depth-first traversal algorithm for a graph.

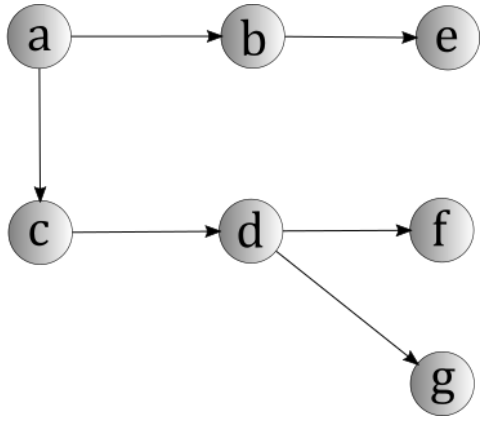
```
DFT(Graph g):
    let startVertex be any vertex in g
```

```
        DFTr(g, startVertex)

    DFTr(Graph g, Vertex v):
        visit(v)
        for each vertex w such that the edge v → w exists:
            DFTr(g, w)
```

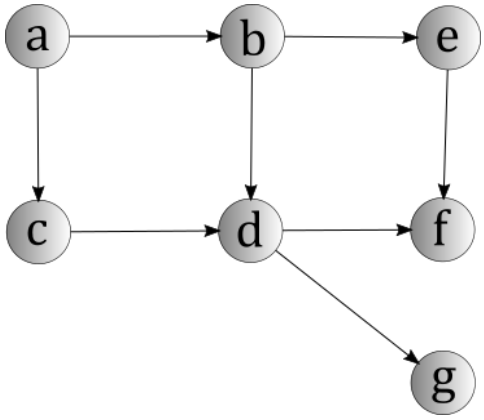
DFT is the complete algorithm; it's job is to traverse an entire graph. **DFTr** can be seen as a recursive algorithm to do a partial traversal of all of the vertices that can be reached starting from the given vertex **v**. (If you were implementing this in a C++ class, **DFT** would likely be a public member function, while **DFTr** might instead be a private member function that's used mainly as a helper.)

Try running through this algorithm on paper, using the graph below. What result do you get if you use *a* as the start vertex?



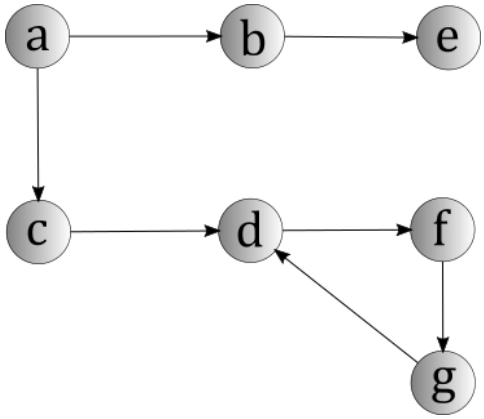
So far so good. One possible outcome is *abecdfg*. Your result may differ depending on which order you iterated through the available edges from each vertex, but you should have visited every vertex exactly once.

Now try the same algorithm on this graph instead. What result do you get if you use *a* as the start vertex?



This time, if you followed the algorithm faithfully as we described it above, you may have noticed that the result was more problematic. Because this graph has what we've called splits-and-joins in it, we end up visiting some of the vertices more than once. (For example, one possible outcome would be *abefdfgcdfg*.) That's an issue we'll need to rectify.

Finally, try the same algorithm on this graph. What result do you get if you use *a* as the start vertex?



This is even more problematic. If you're running through the algorithm the way we wrote it, you may have noticed that you'd slip into infinite recursion, due to the fact that this graph contains a cycle; once you reach the vertex *d*, you'll forever recurse through the cycle involving the vertices *d*, *f*, and *g*.

So we have some work to do to clean this up a bit.

Tracking where we've been

We can solve both the splits-and-joins and cycle problems simultaneously by introducing an additional trick. If we keep track of the vertices we've visited and never allow ourselves to recurse forward to a vertex we've already visited, then neither splits-and-joins nor cycles will be a problem for us; our algorithm will naturally avoid both situations. Here's an updated version of our algorithm that handles that problem.

```
DFT(Graph g):
    for each vertex v in g:
        v.visited = false

    let startVertex be any vertex in g
    DFTr(g, startVertex)

DFTr(Graph g, Vertex v):
    visit(v)
    v.visited = true

    for each vertex w such that the edge v → w exists:
        if not w.visited:
            DFTr(g, w)
```

We've introduced the notion of a **visited** value associated with each vertex. In practice, this may not be implemented as a member variable or member function of each vertex; instead, this might be tracked as a separate collection (e.g., an array of **bool** values). But the key here is that we have to keep track of which vertices we've visited, so we can avoid revisiting ones that we've already seen.

Try our new algorithm on the graphs above, again using *a* as the start vertex each time. Verify that this has solved the problem.

Now try our new algorithm again on any of the graphs above, but using *b* as the start vertex instead. Unfortunately, we've been met with an additional problem; if you start with a vertex from which all other vertices can't be reached, then there are vertices you'll never visit during your traversal. Why we never had this problem with trees is because we always started at the root, and all nodes are always reachable from the root; that's guaranteed, by definition, in a tree. Graphs, on the other hand, don't make that kind of promise, so we'll need to work around the issue ourselves.

Completing our algorithm design

Ensuring that we visit every vertex is a simple matter of allowing ourselves to use **DFTr** repeatedly. A simple definition of **DFTr(g, v)** is this: *Visit every vertex in the graph g that is reachable from the vertex v, except for the ones we've visited before*. So, if we don't visit every vertex the first time, we should try it again, starting from some vertex that we didn't reach previously. This continues until there are no more vertices that we haven't visited.

The final design of our algorithm looks like this.

```
DFT(Graph g):
    for each vertex v in g:
        v.visited = false

    for each vertex v in g:
        if not v.visted:
            DFTr(g, v)

DFTr(Graph g, Vertex v):
    visit(v)
    v.visited = true

    for each vertex w such that the edge v → w exists:
        if not w.visited:
            DFTr(g, w)
```

In other words, we first mark all vertices as unvisited. Next, we call **DFTr** on some unvisited vertex and let it work its way through all of the vertices it's able to reach. When we return from the first call to **DFTr**, we go back into the second loop in **DFT** and look for a vertex we haven't visited. When we find one, we call **DFTr** again. This process repeats until every vertex is considered, at which point every vertex will have been visited.

Try this complete version of **DFT** on the graphs above; try each one a couple of times using a different start vertex each time. You should see that this algorithm will visit every vertex exactly once in a generally depth-first order, i.e., by walking forward along one path until a dead end is reached, then backtracking and trying a different path, and so on, until every vertex has been reached.

Asymptotic analysis

Now that we've got a complete depth-first graph traversal algorithm, we should consider how long this algorithm might take to run. Even though we've sketched pseudocode, this will be a little more complicated than a simple trick like using multiplication every time we see a loop. What we need to understand is the totality of what the algorithm does. There are a few important facts that underlie the analysis:

- A complete traversal visits every vertex exactly once.
- When a vertex is visited, it's necessary to find all of its outgoing edges.
- Overall, the algorithm enumerates all of the edges in the graph exactly once.

Let's break down the various parts of what this algorithm does, and separately analyze each. We can then add these analyses together to determine a final, overall result.

Part	*Adj. Matrix*	*Adj. Lists*	*Commentary*
Marking all vertices unvisited	$\Theta(v)$	$\Theta(v)$	This is the loop that initially sets **visited** to false for each vertex.
Deciding whether to call **DFTr** on each vertex	$\Theta(v)$	$\Theta(v)$	This is the loop that optionally calls **DFTr** on each vertex, if it hasn't been visited yet. Not counted here is the time it takes to do the

			actual work of **DFTr**; we'll count that separately.
Visiting all vertices	$\Theta(v)$	$\Theta(v)$	This makes the reasonable assumption that visiting each vertex takes $\Theta(1)$ time. (That's not to say that visits are necessarily fast, but that the time it takes to visit one vertex isn't a function of the number of vertices or the number of edges.)
Enumerating all edges	$\Theta(v^2)$	$\Theta(v + e)$	This is where the implementation makes a difference. An adjacency matrix would require us to enumerate all *possible* edges, whether they're in the graph or not; adjacency lists would require us only enumerate the edges that are actually there.
TOTAL	**$\Theta(v^2)$**	**$\Theta(v + e)$**	

To analyze how much memory we need, we'll need to think about what we would be spending the memory on.

- $\Theta(v)$ memory to store the **visited** flags for each vertex. (It's probably a small constant factor of v, but a constant factor of v nonetheless.)
- The stack associated with the recursion could potentially get as deep as the number of vertices, if there's one path that leads through every vertex from the start vertex. So the stack will require $O(v)$ memory.

So, in total, we'll use $\Theta(v)$ memory — above and beyond the memory used by the graph — to perform the traversal.

Using a depth-first traversal to find cycles

Depth-first graph traversals have application besides just visiting every vertex; they also form the basis of a number of seemingly unrelated graph algorithms. One example is determining whether a directed graph has any cycles (i.e., whether it qualifies as a DAG).

There is a key insight that can help us to define a cycle-finding algorithm. In a depth-first traversal, there's always the *current path*, a path leading from the place you started to the place you are. As you recurse forward, that path gets longer; as you backtrack, that path gets shorter. But there is always such a path.

So how do you know when you've found a cycle? What if you find an edge leading to a vertex that's already on the current path? Necessarily, that means there's a cycle. And note, additionally, that the presence of a cycle will always lead to this circumstance; the first time you reach one of the vertices in the cycle, you'll ultimately find that cycle.

Now the only question is how to efficiently check whether a vertex is in the current path. We'll do that by keeping a separate flag called **onPath** for each vertex. The flag starts out **false**, then we'll set it to **true** when we move forward to some vertex and then back to **false** when we backtrack away from it. This way, the set of all vertices for which **onPath** is **true** would be our current path.

The algorithm, then, would be a modification to our **DFT** algorithm. We no longer need the **visit** step, if all we're interested in is whether there is a cycle.

```
DFT(Graph g):
    for each vertex v in g:
        v.visited = false
        v.onPath = false
```

```
        for each vertex v in g:
            if not v.visted:
                DFTr(g, v)

    DFTr(Graph g, Vertex v):
        v.visited = true
        v.onPath = true

        for each vertex w such that the edge v → w exists:
            if not w.visited:
                DFTr(g, w)
            if w.onPath:
                CYCLE FOUND

        v.onPath = false
```

If, in this algorithm, we reach the line that says **CYCLE FOUND**, we've found a cycle; if we never reach that line, there is no cycle in the graph.

Nothing we did here changes the asymptotic analysis for this algorithm — instead of changing one flag each time we reach a vertex, we change two (and then eventually change one back). So we'd say that this takes $\Theta(v^2)$ time if the graph is implemented using an adjacency matrix, or $\Theta(v + e)$ if implemented using adjacency lists.

Breadth-first graph traversals

As we saw when we were learning about tree traversals, another option is a breadth-first traversal. And, indeed, we can do a *breadth-first graph traversal*, as well, based on the same principle: Visit the vertices in the order of how far they are away from the place you started. The algorithm, like its tree-traversing counterpart, uses a queue to keep track of the next vertices it should visit. Each time it visits a vertex, it enqueues the vertices that can be reached from that vertex — importantly skipping vertices that have already been visited or enqueued, so we don't visit the same vertex twice.

A sketch of the algorithm follows:

```
    BFT(Graph g, Vertex startVertex):
        for each vertex v in g:
            v.visited = false

        let q be an empty queue
        q.enqueue(startVertex)

        while not q.isEmpty():
            v = q.dequeue()
            visit(v)
            v.visited = true

            for each vertex w such that the edge v → w exists:
                if not w.visited and not q.contains(w):
                    q.enqueue(w)
```

It's important to note that this algorithm will only reach vertices that are connected to the vertex you start from, though this is quite often a feature rather than a bug. A typical application of a breadth-first graph

traversal is to find other vertices that are nearby, or to find the smallest number of edges you'd need to follow in order to get from one vertex to another. In these kinds of circumstances, disconnected vertices — those you can't reach from the start vertex — are not something you're generally interested in. (If you really do want a breadth-first traversal of the entire graph, you can use a similar trick to the one we used in our depth-first algorithm, by continuing the traversal from some other unvisited vertex when you have nowhere left to go.)

Asymptotic analysis

Ultimately, the time it takes to run this algorithm will be the same as the time required to run a depth-first traversal, as long as it's not expensive to ask the question of whether the queue contains some vertex. (One way to make that into a $\Theta(1)$ operation would be to track this as an additional **enqueued** flag on each vertex.) If this can be done in $\Theta(1)$ time, the total time would be:

- $O(v^2)$ if the graph is implemented as an adjacency matrix
- $O(v + e)$ if the graph is implemented as adjacency lists

The reasons are fairly similar to the analysis of our depth-first traversal. The reason we say O instead of Θ is that not all of the vertices may be connected to the start vertex, so we may not end up traversing the whole graph.

Considering the memory usage of breadth-first traversal leads to a similar result as a depth-first traversal.

- We would need $\Theta(v)$ memory to store the **visited** flag (and possibly also an **enqueued** flag) for every vertex.
- Additionally, we'd need the memory for the queue, which might quickly have almost every vertex in it if the graph is quite dense. The queue, generally, would use $O(v)$ memory.

So, in total, we'd need $\Theta(v)$ memory, though the constant factor on the v might be quite small if the queue never gets large. And, in practice, we would tend to use this algorithm when two things are true:

- We're not necessarily interested in reaching every vertex; instead, we're primarily concerned with vertices that are relatively close (e.g., LinkedIn finding all of the people that are within two or three connections from you).
- The graph is relatively sparse (i.e., most vertices don't have edges to most others).

In those circumstances, the behavior will be reasonable enough. And even an extra $\Theta(v)$ memory isn't that big of a deal alongside a data structure that's already bigger than that to begin with.

How else might we traverse graphs?

Because graphs are organized differently from trees, it stands to reason that there might be other ways to traverse them that are neither depth-first nor breadth-first. Depth-first and breadth-first traversals have their uses, forming the basis of a variety of graph algorithms.

But if our only goal is to simply visit every vertex, for example, we'll generally have a data structure that gives us that information directly. In an adjacency matrix implementation, there is generally a one-dimensional array, ancillary to the matrix itself, that stores vertex information; that array could simply be traversed linearly. If an adjacency lists implementation, the one-dimensional array that forms the basis of the implementation could be traversed linearly.

If our goal is to visit every edge in no particular order, we could similarly traverse the underlying data structure — adjacency matrix or adjacency lists — directly.

How you choose from among these options depends on the problem you're trying to solve. The best thing is to understand what the choices are — depth-first, breadth-first, or something more direct — and which

scenarios might be appropriate for each.