CMSC 351: Depth-First Traverse

Justin Wyss-Gallifent

November 2, 2024

1 Introduction:

Suppose we are given a graph G and a starting node s. Suppose we wish to simply traverse the graph in some way looking for a particular value associated with a node. We're not interested in minimizing distance or cost or any such thing, we're just interested in the traversal process.

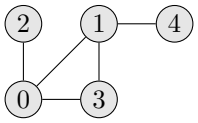
2 Intuition

One classic way to go about this is a depth-first traverse. The intuitive idea is that starting with a starting node s we follow one path as far as possible before backtracking. When we backtrack we only do so as little as possible until we can go deeper again.

Observe that this description does not lead to a unique traversal because there may be multiple paths that we can follow from a given vertex.

3 Visualization

Before writing down some explicit pseudocode let's look at an easy graph and look at how the above intuition might pan out. Consider this graph:



Suppose we start at the vertex $s = 0$. We have three edges we can follow, let's suppose we follow the edge to the vertex 3 first. From 3 we can only go to 1 (we can't go back to 0 since we've visited it already) and then to 4 (we can't go back to 0 or 3). At that point we have gone as deep as we can along that branch so we go back to the most recent branch for which there are other paths available. We have to go back to 0 and from there we go to 2. Thus our depth-first traverse follows the vertices in order $0, 3, 1, 4, 2$.

4 Algorithm Implementation

There are several classic approaches to constructing an algorithm for depth-first traverse:

- Using recursion.
- Using a stack and allowing elements to be on that stack more than once and just ignoring later encounters. We'll call this the Simple Stack.
- Using a stack and only allowing elements to be on the stack once. We'll call this the Fancy Stack.

5 Recursive Implementation

5.1 Pseudocode

The pseudocode for the recursive implementation is as follows:

```
// These are global.
VORDER = []
VISITED = list of length V full of FALSE
function dft(G,x):
    VORDER.append(x)
    VISITED[x] = TRUE
    for all adjacent nodes y (in some order)
        if VISITED[y] == FALSE
            dft(G,y)
        end if
    end for
end function
dft(G,s)
```

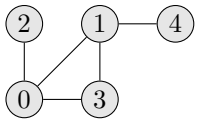
In this pseudocode the list `VORDER` will record the order in which we visit the vertices while the list `VISITED` will indicate whether or not a vertex has been visited.

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in decreasing order.

Example 5.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. Before the function is called we have:

index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Our first call is `dft(G,0)` and before the `for` loop we then have:

index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

The `for` loop cycles through the vertices 3,2,1 (call this depth 1) and the first call is `dft(G,3)` which yields, before its own `for` loop:

index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

From `dft(G,3)` the `for` loop cycles through the vertices 1,0 (call this depth 2) and the first call is `dft(G,1)` which yields, before its own `for` loop:

index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

From `dft(G,1)` the `for` loop cycles through the vertices 4,0 (call this depth 3) and the first call is `dft(G,4)` which yields, before its own `for` loop:

index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

From `dft(G,4)` the `for` loop cycles through the vertices 1 (call this depth 4) but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 3 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 2 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 2 so we call `dft(G,2)` which yields, before its own `for` loop:

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

From `dft(G,2)` the `for` loop cycles through the vertices 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 1 but since that vertex has been visited, `dft` is not called on it again.

Then we are done. Observe that the order in which we visited the nodes is $0, 3, 1, 4, 2$.

5.2 Pseudocode Time Complexity

Suppose V is the number of nodes and E is the number of edges. What follows is exactly the same as breadth-first traverse so if that made sense you can possibly skip this.

- The initialization takes $\mathcal{O}(V)$. This could in fact take $\Theta(1)$ depending on the architecture but the choice has no effect on the result.
- Each node gets processed once so this is $V\Theta(1)$ each for a total of $\Theta(V)$.

- Since each edge is attached to two nodes the `for` loop will iterate a total of $2E$ times over the course of the entire algorithm. This gives a total of $\Theta(2E) = \Theta(E)$.

The time complexity is therefore $\mathcal{O}(V) + \Theta(V) + \Theta(E) = \mathcal{O}(V + E)$. If initialization is actually $\Theta(1)$ then this becomes $\Theta(V + E)$.

Note 5.2.1. Note that our pseudocode and analysis assumes we have direct access to a node's edges using something like an adjacency list. If we use an adjacency matrix then the inner loop becomes $\Theta(V)$ and the entire pseucode becomes $\Theta(V^2)$.

6 Simple Stack Implementation

6.1 Pseudocode

The pseudocode for the simple stack implementation is as follows:

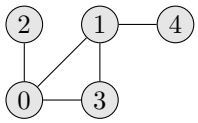
```
VORDER = []
VISITED = list of length V full of FALSE
STACK = [s]
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE:
        VISITED[x] = TRUE
        VORDER.append(x)
        for all nodes y adjacent to x:
            if VISITED[y] == FALSE:
                STACK.push(y)
            end if
        end for
    end if
end while
```

The line:

```
for all nodes y adjacent to x
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 6.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. We start with the following

before the `while` loop:

S = [0]

index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices $1, 2, 3$ and push them all onto the stack:

S = [1,2,3]

index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices $0, 1$ and push 1 (but not 0) onto the stack:

S = [1,2,1]

index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices $0, 3, 4$ and push 4 (but not $0, 3$) onto the stack:

S = [1,2,4]

index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

S = [1,2]

index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

S = [1]

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Iterate! We pop $x = 1$ off the stack. Since it's visited we do nothing with it. We then iterate over the vertices $0, 3, 4$ but nothing is pushed onto the stack:

S = []

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is $0, 3, 1, 4, 2$.

6.2 Pseudocode Time Complexity

The time complexity of this pseudocode is not immediately easy to see since we are not sure exactly how many times the `while` loop will run, given that we are allowing vertices on the stack more than once. However the following facts are certain:

- The body of the `while` loop iterates once for each vertex popped off the stack. Since vertices may be on the stack more than once we don't know how many times this will be without knowing more about the structure of the graph. Suppose we say that the while loop runs W times with $W \geq V$. Since the `if` statement passes exactly V times we can break the `while` loop into two categories - the V times where the `if` statement passes and the $W - V$ times when it fails.
- Focusing only on each of the the $W - V$ times when the `if` statement fails. the body of the `while` loop takes time $\Theta(1)$. This gives a total time $(W - V)\Theta(1) = \Theta(W - V)$ for all `if` statement failures together.
- If we are using an adjacency matrix then focusing only on each of the V times when the `if` statement passes, the `for` loop iterates V times taking time $\Theta(1)$ for a total of $V\Theta(1) = \Theta(V)$. This gives a total time $\Theta(V^2)$ for all `if` statement passes together.

  Together then if we are using an adjacency matrix then the total time for the pseudocode is $\Theta(W - V) + \Theta(V^2)$.
- If we are using an adjacency list let's ignore the `for` loop for a minute. Focusing only on each of the the V times when the `if` statement passes, the body of the `if` statement takes time $\Theta(1)$. This gives a total time $\Theta(V)$ for all `if` statement passes together.

  But then the body of the `for` loop, over the course of the entire algorithm, iterates $2E$ times for a total time $\Theta(2E)$.

Together then if we are using an adjacency list then the total time is $\Theta(W - V) + \Theta(V) + \Theta(V + E)$.

So now we have:

- In the best-case $W = V$ (there are no stack repeats) which can happen for a star graph starting at the star point. In that case the adjacency matrix has time:

  $$\Theta(W - V) + \Theta(V^2) = \Theta(V - V) + \Theta(V^2) = \Theta(V^2)$$

  And the adjacency list has time:

  $$\Theta(W-V)+\Theta(V)+\Theta(V+E) = \Theta(V-V)+\Theta(V)+\Theta(V+E) = \Theta(V+E)$$

- In the worst case every vertex is connected to every other vertex in which case the algorithm proceeds as follows:

  We start by pushing the starting vertex onto the stack. When we pop this vertex there is 1 visited vertex and the `for` loop will push all of the remaining $V - 1$ (unvisited) vertices onto the stack.

  When we pop the next vertex there are 2 visited vertices and the `for` loop will push $V - 2$ (unvisited) vertices onto the stack, all of which will be repeats.

  This will repeat through the entire process and all together the number of vertices which get pushed onto the stack and hence the number of iterations of the while loop will be:

  $$W = 1 + (V - 1) + (V - 2) + ... + 2 + 1 + 0 = 1 + 0.5V(V - 1)$$

  In this case then the adjacency matrix has time:

  $$\Theta(W - V) + \Theta(V^2) = \Theta(1 + 0.5V(V - 1) - V) + \Theta(V^2) = \Theta(V^2)$$

  And the adjacency list has time:

  $$\Theta(W-V)+\Theta(V)+\Theta(V+E) = \Theta(1+0.5V(V-1)-V)+\Theta(V)+\Theta(V+E) = \Theta(V^2+E)$$

Note 6.2.1. I've seen places where people say that this implementation is $\Theta(V + E)$ in all cases when an adjacency list is used but clearly this is false.

7 Fancy Stack Implementation

7.1 Introduction

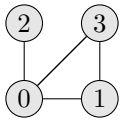
It is possible to modify the stack version to bring it down to $\Theta(V + E)$ in all cases provided an adjacency list is used.. The trick is to find a way to ensure

that an item is only ever popped off the stack once, but we can't be sloppy about it. In theory there are two ways to go about this. When we are about to push something onto the stack, other than checking if it has been visited:

(a) We check whether it has been or still is on the stack and if so, we don't push it.

(b) We check whether it has been on the stack, we don't push it, and if it is on the stack, we remove the earlier occurrence.

While (a) seems easier to do it does not work, as is easily demonstrated by this graph:



Examine approach (a). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and don't push anything, so $S = [3, 2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = [3]$. We then pop $x = 3$ (marking it as visited), and don't push anything, so $S = []$, and then we are done. However we have now visited the vertices in the order $0, 1, 2, 3$ which is not a DFT since after visiting 1 we should go to 3.

On the other hand examine approach (b). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and we push 3, replacing the earlier occurrence, so $S = [2, 3]$. We then pop $x = 3$ (marking it as visited) and we push nothing, so $S = [2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = []$, and then we are done. Now we have visited the vertices in the order $0, 1, 3, 2$.

Before digging into (b) recall that for (b) we need to be able to detect if an element is on the stack and remove an element from the stack both as quickly as possible. It turns out that both can be done by setting the stack up as a doubly-linked list with a helper array which is indexed by the elements and for each element points to either the element on the stack or NULL if it is not on the stack. In this way we can use the pointers to know if an element is on the stack in $\Theta(1)$ time and the doubly-linked list allows us to remove that element from the stack in $\Theta(1)$ time.

7.2 Pseudocode

In this pseudocode we won't fiddle with the details of pointers or removal:

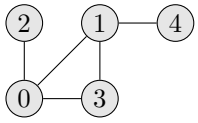
```
VORDER = []
VISITED = list of length V full of FALSE
STACK = [s]
while STACK is not empty:
    x = STACK.pop()
    VISITED[x] = TRUE
    VORDER.append(x)
    for all nodes y adjacent to x:
        if VISITED[y] == FALSE:
            if y on STACK:
                STACK.remove(y)
            end if
            STACK.push(y)
        end if
    end for
end while
```

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 7.1. Let's return to our example from earlier:



Illustrating the pointers is a bit of a pain so we will avoid doing so. Instead the critical difference between this implementation and the stack implementation is that whenever a vertex is pushed onto the stack we check if it is already on the stack and if so we delete its earlier occurence. In what follows this is the only significant difference.

We'll start our traversal at the vertex $x = 0$. We start with the following before the `while` loop:

S = [0]

index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such

and append it to the visiting order. We then iterate over the vertices $1, 2, 3$ and push them all onto the stack:

S = [1,2,3]

index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices $0, 1$ and push 1 (but not 0) onto the stack which deletes the earlier 1:

S = [2,1]

index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices $0, 3, 4$ and push 4 (but not $0, 3$) onto the stack:

S = [2,4]

index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

S = [2]

index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

S = []

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is $0, 3, 1, 4, 2$.

7.3 Pseudocode Time Complexity

With this modification the `while` loop iterates exactly V times and the time complexity is exactly as for the Shortest Path Algorithm and Breadth-First Search.

Thus f the graph is stored as an adjacency matrix then the time is $\Theta(V^2)$ and if the graph is stored as an adjacency list then the time is $\Theta(V + E)$.