CMSC 351: Depth-First Traverse

Justin Wyss-Gallifent

April 21, 2022

1 Introduction: .......................................................... 2
2 Intuition ................................................................. 2
3 Working Through an Example .................................. 2
4 Pseudocode ............................................................ 11
5 Pseudocode Time Complexity ............................... 12
6 Thoughts, Problems, Ideas ..................................... 13
7 Python Test and Output ........................................... 14
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 traverse process.

2 Intuition

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

3 Working Through an Example

Example 3.1. Consider the following graph

Suppose we wish to traverse the graph starting at the node $s = 0$. We set up a few things:

- A current node $x$.
- An array $D$ of length $n$ which indicates whether each node has been discovered or not. Initially it is all FALSE except for $D[s] = TRUE$.
- An array $DORDER$ which will contain the nodes in the order we visit them.
Thus we have:

\[ x = 0 \]


\[ DORDER = [0] \]

We notice that \( x = 0 \) is attached to undiscovered \( \{1, 4, 5\} \) so we follow 1.
Thus we have:

\[ x = 1 \]


\[ DORDER = [0, 1] \]
We notice that $x = 1$ is attached to undiscovered $\{2, 6\}$ so we follow 2. Thus we have:

$x = 2$

DORDER = [0, 1, 2]

We notice that $x = 2$ is attached to undiscovered $\{3, 6, 7\}$ so we follow 3. Thus we have:

$x = 3$

DORDER = [0, 1, 2, 3]
We notice that $x = 3$ is attached to undiscovered \{\} so we go back as few steps as possible to reach a node where an alternate node may be chosen. This is node 2 which allows us to follow 6.
Thus we have:

$x = 6$


$DORDER = [0, 1, 2, 3, 6]$

We notice that $x = 6$ is attached to undiscovered \{11\} so we follow 11.
Thus we have:

$x = 11$


$DORDER = [0, 1, 2, 3, 6, 11]"
We notice that \( x = 11 \) is attached to undiscovered \( \{15\} \) so we follow 15. Thus we have:

\[ x = 15 \]


\[ DORDER = [0, 1, 2, 3, 6, 11, 15] \]

We notice that \( x = 15 \) is attached to undiscovered \( \{} \) so we go back as few steps as possible to reach a node where an alternate node may be chosen. This is node 2 which allows us to follow 7. Thus we have:

\[ x = 7 \]


\[ DORDER = [0, 1, 2, 3, 6, 11, 15, 7] \]
We notice that $x = 7$ is attached to undiscovered $\{\}$ so we go back as few steps as possible to reach a node where an alternate node may be chosen. This is node 0 which allows us to follow 4.
Thus we have:

$$x = 4$$


$$DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4]$$

We notice that $x = 4$ is attached to undiscovered $\{8\}$ so we follow 8.
Thus we have:

$$x = 8$$


$$DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8]$$
We notice that $x = 8$ is attached to undiscovered \{5, 13\} so we follow 5. Thus we have:

\[
x = 5
\]

\[
\]

\[
DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5]
\]

We notice that $x = 5$ is attached to undiscovered \{9\} so we follow 9. Thus we have:

\[
x = 9
\]

\[
\]

\[
DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9]
\]
We notice that $x = 9$ is attached to undiscovered \{10\} so we follow 10. Thus we have:

$x = 10$


$DORDER = [0, 1, 2, 3, 6, 11, 7, 4, 8, 5, 9, 10]$

We notice that $x = 10$ is attached to undiscovered \{14\} so we follow 14. Thus we have:

$x = 14$

$D = [T, T, T, T, T, T, T, T, T, T, T, T, T, F, T, T]$

$DORDER = [0, 1, 2, 3, 6, 11, 7, 4, 8, 5, 9, 10, 14]$
We notice that $x = 14$ is attached to undiscovered $\{\}$ so we go back as few steps as possible to reach a node where an alternate node may be chosen. This is node 8 which allows us to follow 13.

Thus we have:

$x = 13$

$D = [T, T, T, T, T, T, T, T, T, T, T, T, F, T, T, T]$

$DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13]$

We notice that $x = 13$ is attached to undiscovered $\{12\}$ so we follow 12. Thus we have:

$x = 12$

$D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T]$

$DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13, 12]$

There are no undiscovered nodes.

We stop with $DORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13, 12]$. 
4 Pseudocode

The following algorithm simply visits all nodes. It doesn’t do anything with them except assigns a global list in the order in which they are visited.

Here we present the recursive version. It’s fairly straightforward to write a non-recursive version but there are some nuances which need to be dealt with. Here the phrasing of the pseudocode assumes that $G$ is stored as an adjacency list.

```plaintext
// These are global.
DORDER = []
D = list of FALSE of length V
function depthfirsttraverse(G,x):
    DORDER.append(x)
    D[x] = TRUE
    for all edges from x
        y = the node at the other end
        if D[y] == FALSE
            depthfirsttraverse(G,y)
    end
end
depthfirsttraverse(G,s)
```

Consider that each recursive call is on an undiscovered node. It follows that the maximum depth of the recursion will be equal to the longest path extending from the starting node. Along with the fact that the recursion is only called on a node which is undiscovered gives insight into the operation of the algorithm and why the recursive process actually terminates.

Note 4.0.1. Note that for loop will always encounter at least one previously discovered node, the algorithm-parent of $x$, except when $x$ is the root. If at any point the algorithm encounters a previously discovered node which is not the algorithm-parent of $x$ then the algorithm has discovered a cycle.

Note 4.0.2. If we wish to exit when we find a node then we’ll need to manage the recursion properly so that when the node is found we progressively drop out of all the recursive layers.

Note 4.0.3. Depth-first traversing is more useful when we suspect that the target is far from the starting node.

Note 4.0.4. Depth-first traversing is more useful for puzzle-like problems which involve making a decision and carrying it through to completion (this is a recursive process).
5 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 $O(V)$. 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 $O(V) + \Theta(V) + \Theta(E) = O(V + E)$. If initialization is actually $\Theta(1)$ then this becomes $\Theta(V + E)$.

**Note 5.0.1.** 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 pseudocode becomes $\Theta(V^2)$. 
6 Thoughts, Problems, Ideas

1. Suppose $G$ is stored by its adjacency matrix $AM$. Adjust the depth-first pseudocode to make this clear and calculate the resulting $\Theta$ time complexity. Call this new function $\text{depthfirsttraverse}(AM,x)$.

2. Modify the depth-first pseudocode to add a second list $\text{PARENT}$ which keeps track of the parent of each node visited. That is, $\text{PARENT}(z)$ should contain the parent of node $z$. The root node should have NULL assigned. How does this affect the time complexity?

3. Modify the depth-first traverse pseudocode to detect and return TRUE if there is a cycle in the graph and FALSE if not. How does this affect the time complexity?
7 Python Test and Output

The following code is applied to the graph above. This follows the model of the pseudocode and in addition creates and returns a list of the nodes in the order in which they were visited.

Code:

```python
def dfs(EL,n,x,depth):
    print('  ' * depth + ' Recursive depth = ' + str(depth))
    D[x] = True
    V.append(x)
    print('  ' * depth + 'V = ' + str(V))
    print('  ' * depth + 'D = ' + str(D).replace('True','T').replace('False','F'))
    for y in EL[x]:
        if not D[y]:
            dfs(EL,n,y,depth+1)
EL = [
    [1, 4, 5],
    [0, 2, 6],
    [1, 3, 6, 7],
    [2],
    [0, 8],
    [0, 8, 9],
    [1, 2, 11],
    [2],
    [4, 5, 13],
    [5, 10],
    [9, 14],
    [6, 15],
    [13],
    [8, 12],
    [10],
    [11]]
n = 16
s = 0
D = [False] * n
V = []
dfs(EL,n,s,0)
```
Output:

\[ V = [0] \]
_Recursive depth = 1
\[ V = [0, 1] \]
__Recursive depth = 2
\[ V = [0, 1, 2] \]
___Recursive depth = 3
\[ V = [0, 1, 2, 3] \]
___Recursive depth = 3
\[ V = [0, 1, 2, 3, 4] \]
____Recursive depth = 4
\[ V = [0, 1, 2, 3, 4, 5] \]
___Recursive depth = 5
\[ V = [0, 1, 2, 3, 4, 5, 6] \]
__Recursive depth = 6
\[ V = [0, 1, 2, 3, 4, 5, 6, 7] \]
_Recursive depth = 7
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8] \]
___Recursive depth = 8
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \]
____Recursive depth = 9
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, F] \]
_____Recursive depth = 10
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 11
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 12
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 13
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 14
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 15
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 16
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 17
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 18
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 19
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 20
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 21
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 22
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 23
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] \]
\[ D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T] \]
______Recursive depth = 24
\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25] \]
Recursive depth = 4

V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13, 12]

D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T]