CMSC 351: Depth-First Traverse

Justin Wyss-Gallifent

May 8, 2021

1 Introduction: ................................................................. 2
2 Intuition ................................................................. 2
3 Pseudocode ............................................................ 3
4 Pseudocode Time Complexity ........................................... 10
5 Thoughts, Problems, Ideas ................................................ 11
6 Python Test and Output ................................................ 12
1 Introduction:

Suppose we are given a graph $G$ and a starting vertex $s$. 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 vertex $s$ we follow one branch (recursively) as far as possible before backtracking. When we backtrack we only do so as little as possible until we can go deeper again.

If this concept is unclear we’ll see it unfold with an example after we give the pseudocode.
3 Pseudocode

The following algorithm simply visits all vertices. 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.
VISITORORDER = []
VISITED = list of FALSE of length V
function depthfirsttraverse(G,x):
    VISITORORDER.append(x)
    VISITED[x] = TRUE
    for all edges from x
        y = the vertex at the other end
        if VISITED[y] == FALSE
            depthfirsttraverse(G,y)
    end
end
depthfirsttraverse(G,s)
```

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

Note 3.0.1. Note that for loop will always encounter at least one previously discovered vertex, the algorithm-parent of $x$, except when $x$ is the root. If at any point the algorithm encounters a previously discovered vertex which is not the algorithm-parent of $x$ then the algorithm has discovered a cycle.

Note 3.0.2. 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 3.0.3. Depth-first traversing is more useful when we suspect that the target is far from the starting vertex.

Note 3.0.4. 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).
Example 3.1. Consider the following graph. This is almost identical to the graph in the breadth-first traverse except one edge has been removed because it makes the progress more clear.

\[ x = 0 \]


\[ VISITEDORDER = [0] \]

There are undiscovered adjacent vertices \{1, 4, 5\}.

\[ x = 1 \]


\[ VISITEDORDER = [0, 1] \]

There are undiscovered adjacent vertices \{2, 6\}.

\[ x = 2 \]


\[ VISITEDORDER = [0, 1, 2] \]
There are undiscovered adjacent vertices \{3, 6, 7\}.

\( x = 3 \)


\( VISITEDORDER = [0, 1, 2, 3] \)

\[
\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{0} & 4 & 8 & 12 & 15 & 11 & 7 & 3 \\
\text{1} & 5 & 9 & 13 & 10 & 14 & 6 & 2 \\
\text{2} & 6 & 10 & 14 & & & & \\
\text{3} & 7 & 11 & 15 & & & &
\end{array}
\]

There are undiscovered adjacent vertices \{\}.

\( x = 6 \)


\( VISITEDORDER = [0, 1, 2, 3, 6] \)

\[
\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{0} & 4 & 8 & 12 & 15 & 11 & 7 & 3 \\
\text{1} & 5 & 9 & 13 & 10 & 14 & 6 & 2 \\
\text{2} & 6 & 10 & 14 & & & & \\
\text{3} & 7 & 11 & 15 & & & &
\end{array}
\]

There are undiscovered adjacent vertices \{11\}.

\( x = 11 \)


\( VISITEDORDER = [0, 1, 2, 3, 6, 11] \)

\[
\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3}
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{0} & 4 & 8 & 12 & 15 & 11 & 7 & 3 \\
\text{1} & 5 & 9 & 13 & 10 & 14 & 6 & 2 \\
\text{2} & 6 & 10 & 14 & & & & \\
\text{3} & 7 & 11 & 15 & & & &
\end{array}
\]
There are undiscovered adjacent vertices \{15\}.

\(x = 15\)
\[VISITEDORDER = [0, 1, 2, 3, 6, 11, 15]\]

There are undiscovered adjacent vertices \{\}\.

\(x = 7\)
\[VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7]\]

There are undiscovered adjacent vertices \{\}\.

\(x = 4\)
\[VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4]\]
There are undiscovered adjacent vertices \{8\}.

\[ x = 8 \]
\[ VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8] \]

There are undiscovered adjacent vertices \{5,13\}.

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

There are undiscovered adjacent vertices \{9\}.

\[ x = 9 \]
\[ VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9] \]
There are undiscovered adjacent vertices $\{10\}$.

$x = 10$


$VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10]$
There are undiscovered adjacent vertices \( \{12\} \).

\[ x = 12 \]

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

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

There are no undiscovered vertices.

We stop with \( VISITEDORDER = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13, 12] \).
4 Pseudocode Time Complexity

Suppose $V$ is the number of vertices and $E$ is the number of edges. What follows is exactly the same as breadth-first traverse.

- 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 vertex gets processed once so this is $V\Theta(1)$ each for a total of $\Theta(V)$.

- Since each edge is attached to two vertices 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 4.0.1.** Note that our pseudocode and analysis assumes we have direct access to a vertex’s edges using something like an adjacency list. If we use an adjacency matrix then the inner loop becomes $\Theta(V)$ and the entire psuedocode becomes $\Theta(v^2)$. 

5 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 vertex $z$. 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 $\text{TRUE}$ if there is a cycle in the graph and $\text{FALSE}$ if not. How does this affect the time complexity?
6 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 vertices 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)
```
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 = 4
  _V = [0, 1, 2, 3, 6]
  _Recursive depth = 5
  _V = [0, 1, 2, 3, 6, 11]
  _Recursive depth = 6
  _V = [0, 1, 2, 3, 6, 11, 15]
  _Recursive depth = 7
  _V = [0, 1, 2, 3, 6, 11, 15, 7]
  _Recursive depth = 8
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4]
  _Recursive depth = 9
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8]
  _Recursive depth = 10
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5]
  _Recursive depth = 11
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9]
  _Recursive depth = 12
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10]
  _Recursive depth = 13
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14]
  _Recursive depth = 14
  _V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14, 13]
  _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, 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]