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 brance (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 Algorithm

The algorithm for depth-first traverse starting at a vertex $s$ proceeds as follows:

We first set up:

- A stack with just $s$ on it, so $S = [s]$.
- A boolean list $D$ of length $V$ called the visited array which indicates whether a vertex has been visited or not and fill it full of $F$, or 0.

We then repeat the following steps until the stack is empty:

1. $x = S.pop$
2. $D[x] = T$
3. Find all vertices adjacent to $x$ which have not been visited and are not on the stack. For each, push it on the stack.

By convention and consistency when we “find all vertices” we’ll do it in increasing numerical order.

4 Working Through an Example

Example 4.1. Consider the following graph
Suppose we wish to traverse the graph starting at the node \( s = 0 \).


**Iterate!** We pop \( x = 0 \). We mark it as visited and note that it is adjacent to unvisited \( 1, 4, 5 \). We push them onto the stack.

\( S = [1, 4, 5] \)

**Iterate!** We pop \( x = 5 \). We mark it as visited and note that it is adjacent to unvisited \( 8, 9 \). We push them onto the stack.

\( S = [1, 4, 8, 9] \)

**Iterate!** We pop \( x = 9 \). We mark it as visited and note that it is adjacent to unvisited \( 10 \). We push that onto the stack.

\( S = [1, 4, 8, 10] \)

**Iterate!** We pop \( x = 10 \). We mark it as visited and note that it is adjacent to unvisited \( 14 \). We push that onto the stack.

\( S = [1, 4, 8, 14] \)

**Iterate!** We pop \( x = 14 \). We mark it as visited and note that it is not adjacent to any unvisited vertices.

\( S = [1, 4, 8] \)

**Iterate!** We pop \( x = 8 \). We mark it as visited and note that it is adjacent to unvisited \( 4, 13 \). We push 13 onto the stack but not 4 since it already is on the stack.

\( S = [1, 4, 13] \)

**Iterate!** We pop \( x = 13 \). We mark it as visited and note that it is adjacent to unvisited \( 12 \). We push that onto the stack.

\( S = [1, 4, 12] \)

**Iterate!** We pop \( x = 12 \). We mark it as visited and note that it is not adjacent to any unvisited vertices.
$S = [1, 4]$  

Iterate! We pop $x = 4$. We mark it as visited and note that it is not adjacent to any unvisited vertices.  
$S = [1]$  

Iterate! We pop $x = 1$. We mark it as visited and note that it is adjacent to unvisited $2, 6$. We push them onto the stack.  
$S = [2, 6]$  

Iterate! We pop $x = 6$. We mark it as visited and note that it is adjacent to unvisited $2, 11$. We push $11$ onto the stack but not $2$ since it already is on the stack.  
$S = [2, 11]$  

Iterate! We pop $x = 11$. We mark it as visited and note that it is adjacent to unvisited $15$. We push that onto the stack.  
$S = [2, 15]$  

Iterate! We pop $x = 15$. We mark it as visited and note that it is not adjacent to any unvisited vertices.  
$S = [2]$  

Iterate! We pop $x = 2$. We mark it as visited and note that it is adjacent to unvisited $3, 7$. We push them onto the stack.  
$S = [3, 7]$  
$D = [T, T, T, F, T, T, T, T, T, T, T, T, T]$  

Iterate! We pop $x = 7$. We mark it as visited and note that it is not adjacent to any unvisited vertices.  
$S = [3]$  
$D = [T, T, T, F, T, T, T, T, T, T, T, T, T]$  

Iterate! We pop $x = 3$. We mark it as visited and note that it is not adjacent to any unvisited vertices.  
$S = []$  
$D = [T, T, T, T, T, T, T, T, T, T, T, T]$  

Now we are done because the stack is empty.  

The order in which we traverse is the order in which the nodes were popped and marked as visited:  

$$0, 5, 9, 10, 14, 8, 13, 12, 4, 1, 6, 11, 15, 2, 7, 3$$
Note that for a small graph we can do this in a nice tabular fashion:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$x$</th>
<th>Adj</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>1,4,5</td>
<td>T F F F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>[1,4,5]</td>
<td>5</td>
<td>8,9</td>
<td>T F F F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>[1,4,8,9]</td>
<td>9</td>
<td>10</td>
<td>T F F F F T F F F F T F F F F F F F F</td>
</tr>
<tr>
<td>[1,4,8,10]</td>
<td>10</td>
<td>14</td>
<td>T F F F F T F F F F T F F F F F F F F</td>
</tr>
<tr>
<td>[1,4,8,14]</td>
<td>14</td>
<td></td>
<td>T F F F F T F F F F T F F F F F F F F</td>
</tr>
<tr>
<td>[1,4,8]</td>
<td>8</td>
<td>13</td>
<td>T F F F F T F F T T T F F F F F T F</td>
</tr>
<tr>
<td>[1,4,13]</td>
<td>13</td>
<td>12</td>
<td>T F F F F T F F T T T F F F T T F</td>
</tr>
<tr>
<td>[1,4,12]</td>
<td>12</td>
<td></td>
<td>T F F F F T F F T T T F F T T T F</td>
</tr>
<tr>
<td>[1,4]</td>
<td>4</td>
<td></td>
<td>T F F F T T T F F T T T F T T T F</td>
</tr>
<tr>
<td>[1]</td>
<td>1</td>
<td>2,6</td>
<td>T T F F T T F F T T T F F T T T F</td>
</tr>
<tr>
<td>[2,6]</td>
<td>6</td>
<td>11</td>
<td>T T F F T T T F T T T F T T T F</td>
</tr>
<tr>
<td>[2,11]</td>
<td>11</td>
<td>15</td>
<td>T T F F T T T F T T T F T T T F</td>
</tr>
<tr>
<td>[2,15]</td>
<td>15</td>
<td></td>
<td>T T F F T T T F T T T F T T T F</td>
</tr>
<tr>
<td>[2]</td>
<td>2</td>
<td>3,7</td>
<td>T T T F T T T F T T T F T T T T</td>
</tr>
<tr>
<td>[3,7]</td>
<td>7</td>
<td></td>
<td>T T T F T T T F T T T F T T T T</td>
</tr>
<tr>
<td>[3]</td>
<td>3</td>
<td></td>
<td>T T T T T T T T T T T T T T T T</td>
</tr>
</tbody>
</table>
5 Pseudocode

Here is the pseudocode for the above implementation. We’ve also added a second boolean array $OS$ to quickly check if something is on the stack.

```python
function dftstack(G,x):
    D = list of FALSE of length V
    OS = list of FALSE of length V
    S = [x]
    OS[x] = TRUE
    while S is nonempty:
        x = S.pop
        D[x] = TRUE
        OS[x] = FALSE
        for all y adjacent to x
            if D[y] == FALSE and OS[y] == FALSE:
                S.push(y)
                OS[y] = TRUE
        end if
    end for
    end while
end function
```

**Note 5.0.1.** Depth-first traversing is more useful when we suspect that the target is far from the starting node.

**Note 5.0.2.** 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).
6 Pseudocode Time Complexity

Suppose $V$ is the number of nodes and $E$ is the number of edges.

- The initialization takes $\Theta(V)$ to create the two lists of length $V$.
- Each vertex gets popped once so the while loop iterates $V$ times.
- The body of the while loop, excluding the for loop, takes $\Theta(1)$ time, so that is $\Theta(V)$ total.
- The body of the for loop iterates $2E$ times over the course of the entire algorithm, once for each vertex at each end of the edge. The body takes constant time $\Theta(1)$. so overall this is $\Theta(2E) = \Theta(E)$.

Together then the time is $O(E + V)$.

Note that this assumes we have direct access to the edges for each vertex, for example if we have an adjacency list. What would the situation be if we had an adjacency matrix?

7 Modifying to Search

Depth-first traverse can be tweaked if there is a target node in mind. How would you tweak the pseudocode to exit as soon as the target was found and how would that change the time complexity?
8 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 `depthfirsttraverse(AM, x)`.

2. Modify the depth-first pseudocode to add a second list `PARENT` which keeps track of the parent of each node visited. That is, `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?
9 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

_\[ V = [0, 1, 2, 3, 6] \]
_Recursive depth = 4

_\[ V = [0, 1, 2, 3, 6, 11] \]
_Recursive depth = 5

_\[ V = [0, 1, 2, 3, 6, 11, 15] \]
_Recursive depth = 3

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7] \]
_Recursive depth = 1

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4] \]
_Recursive depth = 2

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8] \]
_Recursive depth = 3

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5] \]
_Recursive depth = 4

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9] \]
_Recursive depth = 5

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10] \]
_Recursive depth = 6

_\[ V = [0, 1, 2, 3, 6, 11, 15, 7, 4, 8, 5, 9, 10, 14] \]
_Recursive depth = 3

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

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

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