CMSC 351: Breadth-First Traverse

Justin Wyss-Gallifent

April 19, 2022

1. Introduction .......................................................... 2
2. Intuition ................................................................. 2
3. Working Through an Example ..................................... 3
4. Pseudocode ............................................................. 9
5. Pseudocode Time Complexity ................................... 10
   5.1 Sandwiches ....................................................... 10
6. Recursive Implementation ......................................... 11
7. Thoughts, Problems, Ideas ......................................... 11
8. 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 node. We’re not interested in minimizing distance or cost or any such thing, we’re just interested in the traverse.

2 Intuition

One classic way to go about this is a breadth-first traverse. The idea is that starting with $s$ we check all vertices connected to $s$ first, and then all vertices connected to those, and so on. In this sense we’re covering the graph in “layers of increasing distance from $s$”. This is the idea of a breadth-first traverse.

Note 2.0.1. The shortest path algorithm basically basically follows a breadth-first approach.

Note 2.0.2. Breadth-first traversing is more useful if there’s a target and we suspect that the target is close to the starting vertex.

Note 2.0.3. Breadth-first traversing is more useful for things like web-crawling when we might want to find all of the closer vertices first and the algorithm may truncate early.

Note 2.0.4. Breadth-first traversing is more useful when we are trying to explore a strongly connected part of a graph, the idea being that we want to explore close to home before venturing too far away.
3 Working Through an Example

In this example we won’t show every single iteration. Instead we’ll only show those as noted.

Example 3.1. Consider the following graph.

![Graph](image)

We’ll start with $s = 0$.

We set up a few things:

- A queue $Q = \{s\} = \{0\}$ which will contain the nodes in the order we process them.
- 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 vertices in the order we visit them.

Thus we have:

$Q = \{0\}$


$DORDER = [0]$
Iterate: We pop off $u = 0$, we find all connected and undiscovered vertices \{1, 4, 5\} those get put onto $Q$ and we update $D$.

$Q = \{1, 4, 5\}$


$DORDER = [0, 1, 4, 5]$
To save some time, let’s list the results of the next three pops, meaning popping all the things on $Q$ off:

Iterate: We pop off $u = 1$, we find all connected and undiscovered vertices \{2, 6\} so those get put onto $Q$ and we update $D$.

Iterate: We pop off $u = 4$, we find all connected and undiscovered vertices \{8\} so those get put onto $Q$ and we update $D$.

Iterate: We pop off $u = 5$, we find all connected and undiscovered vertices \{9\} so those get put onto $Q$ and we update $D$.

$Q = \{2, 6, 8, 9\}$


$DORDER = [0, 1, 4, 5, 2, 6, 8, 9]$
To save some time, let’s list the results of the next four pops, meaning popping all the things on \( Q \) off:

Iterate: We pop off \( u = 2 \), it gives us undiscovered vertices \( \{3, 7\} \) so those get put onto \( Q \) and we update \( D \).

Iterate: We pop off \( u = 6 \), we find all connected and undiscovered vertices \( \{11\} \) so those get put onto \( Q \) and we update \( D \).

Iterate: We pop off \( u = 8 \), we find all connected and undiscovered vertices \( \{13\} \) so those get put onto \( Q \) and we update \( D \).

Iterate: We pop off \( u = 9 \), we find all connected and undiscovered vertices \( \{10\} \) so those get put onto \( Q \) and we update \( D \).

\( Q = \{3, 7, 11, 13, 10\} \)


\( DORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10] \)
To save some time, let’s list the results of the next five pops, meaning popping all the things on $Q$ off:

Iterate: We pop off $u = 3$, we find all connected and undiscovered vertices $\emptyset$.

Iterate: We pop off $u = 7$, we find all connected and undiscovered vertices $\emptyset$.

Iterate: We pop off $u = 11$, we find all connected and undiscovered vertices $\{15\}$ so those get put onto $Q$ and we update $D$.

Iterate: We pop off $u = 13$, we find all connected and undiscovered vertices $\{12\}$ so those get put onto $Q$ and we update $D$.

Iterate: We pop off $u = 10$, we find all connected and undiscovered vertices $\{14\}$ so those get put onto $Q$ and we update $D$.

$Q = \{15, 12, 14\}$

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

$DORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]$
To save some time, let’s list the results of the next three pops, meaning popping all the things on $Q$ off:

Iterate: We pop off $u = 15$, we find all connected and undiscovered vertices $\{\}$. 
Iterate: We pop off $u = 12$, we find all connected and undiscovered vertices $\{\}$. 
Iterate: We pop off $u = 14$, we find all connected and undiscovered vertices $\{\}$. 

$Q = \{\}$ 

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

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

Since $Q$ is empty we’re done. 

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

---

8
4 Pseudocode

Here is the pseudocode:

```plaintext
function breadthfirsttraverse(G,s)
    n = number of vertices in G
    // Queue to keep track of what’s next to be dealt with.
    QUEUE = [s]
    // Array to keep track of whether vertices
    // have been discovered.
    D = array with n elements all FALSE
    D[s] = TRUE
    // Array to keep the order in which vertices
    // have been discovered.
    DORDER = [s]
    while QUEUE is not empty
        u = QUEUE.pop
        // If we’re looking for something,
        // put the code to return it here.
        for every edge attached to u
            v = vertex at the other end
            if D[v] == FALSE
                QUEUE.push(v)
                D[v] = TRUE
                DORDER.append(v)
        end
    end
    return (DORDER)
end
```
5 Pseudocode Time Complexity

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

- 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 pushed onto and popped off the queue exactly once so this is $V \Theta(1)$ each for a total of $\Theta(V)$ for just the `pop`, not the `for` loop.

- 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 $\Theta(2E) = \Theta(E)$ for the body of the `for` loop. Note this is over the entire algorithm, even though the code sits inside the `while` loop. If this is confusing, have a sandwich shortly.

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 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 pseudocode becomes $\Theta(V^2)$.

**Note 5.0.2.** There are breadth-first traverses which run in $O(E \log V)$ but they requires a radically different pseudocode based upon a heap structure instead of a simple list $S$. This heap structure is what leads to the $\log V$ factor.

5.1 Sandwiches

The $2E$ issue is often confusing so here’s an analogy which may help. Suppose there were $V$ people in a room and a table nearby containing $E$ sandwiches. Each sandwich is cut in half and each half-sandwich has someone’s name on it.

The following pseudo(sandwich)code gets executed:

```plaintext
for each $P =$ person
  $P$ yells "Delicious"
  $P$ goes to the table
  for each half-sandwich with $P$’s name on it
    $P$ yells "Delicious"
    $P$ eats the half-sandwich
  end
end
```

How many times would you hear Delicious?

The answer is that you’d hear it once for each person and twice for each sandwich. So $V + 2E$. 

10
6 Recursive Implementation

Breadth-first traverses may also be implemented recursively.

7 Thoughts, Problems, Ideas

1. Suppose node $i$ is a structure with properties $i.height$, $i.weight$ and $i.volume$. Modify the pseudocode to return the weight of the first node encountered whose weight is more than 100. You may assume such a node exists.

2. Same as above but no such assumption. Return $NULL$ if no such node is found.

3. When $s$ is popped from the queue all of the vertices connected to $s$ will be newly discovered. This is not true for any other vertex $x$ because the algorithm-parent of $x$ will already be discovered. Under what circumstances, for every other vertex $x$, would the algorithm-parent be the only previously discovered vertex?

4. Let $q_i$ be the length of $Q$ after the $i$th iteration of the while loop. We’ll say $q_0 = 1$ to be comprehensive since $Q = [s]$ when no iterations have completed. So in the example in the notes $q_1 = 3$, $q_3 = 4$, and so on. Of course there is some $k$ such that $q_k = 0$ as this is when the algorithm ends. Moreover in the example $q_i$ initially (nonstrictly) increases and then (nonstrictly) decreases. Must the $q_i$ always follow this pattern? Explain.

5. Building off the previous problem what is the maximum that $k$ might be? How about the minimum? What would a graph look like (qualitatively) if its $k$-value were somewhere in the middle? Explain using specific examples of graphs.

6. Describe the impact if the graph were given with the adjacency matrix rather than the adjacency list. How would that impact the pseudocode and the time complexity?

7. Modify the pseudocode so that we may pass a $maxdepth$ and so that the algorithm will go no further than that depth from the starting vertex.
8 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 discovered.

Code:

```python
def bfs(EL, n, s):
    Q = [s]
    D = [False] * n
    D[s] = True
    V = [s]
    print ('Q = ' + str(Q))
    print ('V = ' + str(V))
    #print ('D = ' + str(D).replace('True','T').replace('False','F'))
    while len(Q) != 0:
        x = Q.pop(0)
        for y in EL[x]:
            if not D[y]:
                D[y] = True
                V.append(y)
                Q.append(y)
                print ('Q = ' + str(Q))
                print ('V = ' + str(V))
                #print ('D = ' + str(D).replace('True','T').replace('False','F'))
    return (V)

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

n = 16
s = 0
```
visited = bfs(EL,n,s)
print('Discovered = ' + str(visited))
Output:

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