CMSC 351: Breadth-First Search

Justin Wyss-Gallifent

April 27, 2021

1 Introduction

Suppose we are given a graph G and a starting vertex s. Suppose we wish to simply search 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 search.

2 Intuition

One classic way to go about this is a breadth-first search. 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 search.

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 returns a list in the order in which they are visited.

```
function breadthfirstsearch(G,s)
    n = number of vertices in G
    QUEUE = [s]
    DISCOVERED = array with n elements all FALSE
    DISCOVERED[S] = TRUE
    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 DISCOVERED[v] == FALSE
                QUEUE.push(v)
                DISCOVERED[v] = TRUE
            end
        end
    end
end
```

Note 3.0.1. The shortest path algorithm basically basically follows a breadth-first approach.

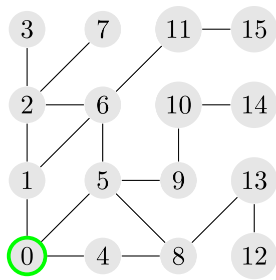
Note 3.0.2. Breadth-first searching is more useful when we suspect that the target is close to the starting vertex.

Note 3.0.3. Breadth-first searching 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.

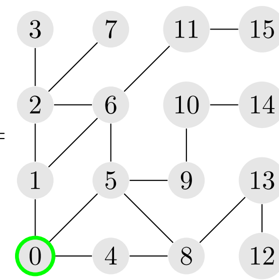
Note 3.0.4. Breadth-first searching is more useful when we are trying to explore a strongly connected part of a graph, the idea being that we want to explore close to home before venturing too far away.

Example 3.1. Consider the following graph. We're going to start with $s = 0$ so that goes into Q first. This is right before the `while` loop starts.
Here $Q = \{0\}$ and $D = [T, F, F, F, F, F, F, F, F, F, F, F, F, F, F, F]$:

Q =

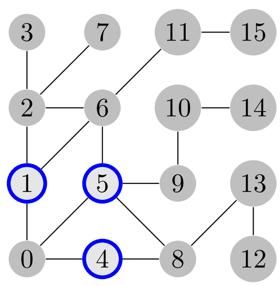


and D =

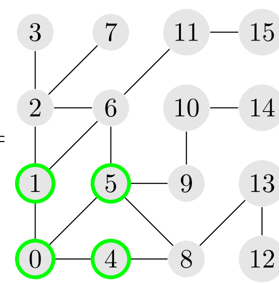


We pop off $x = 0$, it gives us unvisited vertices $\{1, 4, 5\}$ so those get put onto Q and we update D.
Here $Q = \{1, 4, 5\}$ and $D = [T, T, F, F, T, T, F, F, F, F, F, F, F, F, F, F]$:

Q =

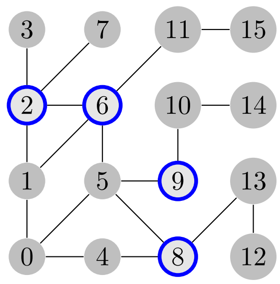


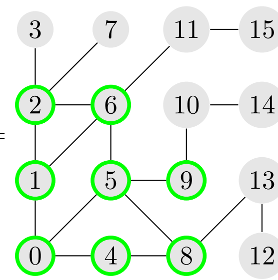
and D =



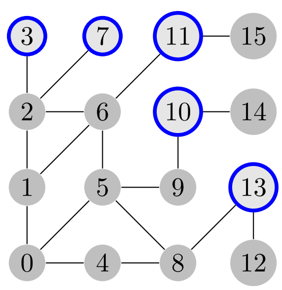
We pop off $x = 1$, it gives us unvisited vertices $\{2, 6\}$ so those get put onto Q and we update D.
We pop off $x = 4$, it gives us unvisited vertices $\{8\}$ so those get put onto Q and we update D.
We pop off $x = 5$, it gives us unvisited vertices $\{9\}$ so those get put onto Q and we update D.

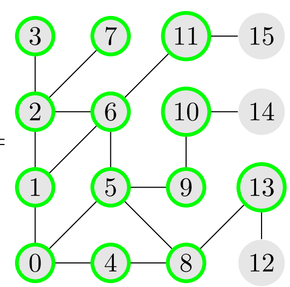
Here $Q = \{2, 6, 8, 9\}$ and $D = [T, T, T, F, T, T, T, F, T, T, F, F, F, F, F, F]$:

Q = 

and D = 

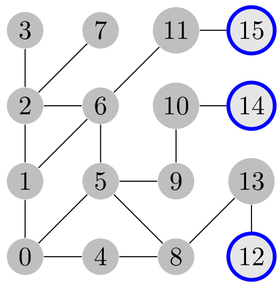
We pop off $x = 2$, it gives us unvisited vertices $\{3, 7\}$ so those get put onto Q and we update D.
We pop off $x = 6$, it gives us unvisited vertices $\{11\}$ so those get put onto Q and we update D.
We pop off $x = 8$, it gives us unvisited vertices $\{13\}$ so those get put onto Q and we update D.
We pop off $x = 9$, it gives us unvisited vertices $\{10\}$ so those get put onto Q and we update D.
Here $Q = \{3, 7, 11, 13, 10\}$ and $D = [T, T, T, T, T, T, T, T, T, T, T, T, T, F, T, F, F]$.

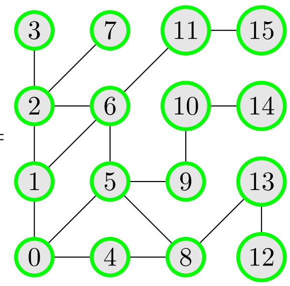
Q = 

and D = 

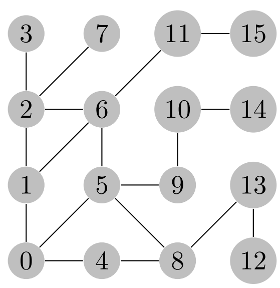
We pop off $x = 3$, it gives us unvisited vertices $\{\}$ so those get put onto Q and we update D.
We pop off $x = 7$, it gives us unvisited vertices $\{\}$ so those get put onto Q and we update D.
We pop off $x = 11$, it gives us unvisited vertices $\{15\}$ so those get put onto Q and we update D.
We pop off $x = 13$, it gives us unvisited vertices $\{12\}$ so those get put onto Q and we update D.
We pop off $x = 10$, it gives us unvisited vertices $\{14\}$ so those get put onto Q and we update D.

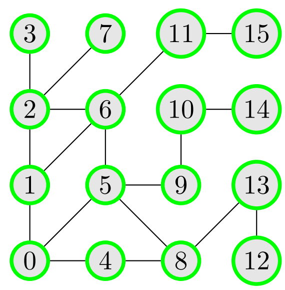
Here $Q = \{15, 12, 14\}$ and $D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T]$:

Q = 

and D = 

We pop off $x = 15$, it gives us unvisited vertices $\{\}$.
We pop off $x = 12$, it gives us unvisited vertices $\{\}$.
We pop off $x = 14$, it gives us unvisited vertices $\{\}$.
Here $Q = \{\}$ and $D = [T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T]$:

Q = 

and D = 

Since Q is empty we're done.

4 Pseudocode Time Complexity

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

- 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 vertex gets pushed onto and popped off the queue exactly once so this is $2\Theta(1)$ 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 $\mathcal{O}(V) + \Theta(V) + \Theta(E) = \mathcal{O}(V + E)$. 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 pseucode becomes $\Theta(v^2)$.

Note 4.0.2. There are breadth-first searches which run in $\mathcal{O}(E \lg V)$ 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 $\lg V$ factor.

5 Recursive Implemention

Breadth-first searches may also be implemented recursively.

6 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 ith 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.

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 vertices in the order in which they were visited.
Code:

```
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('Visited = ' + 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]
Visited = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12,
     14]
```