CMSC 351: Shortest Path Algorithm

Justin Wyss-Gallifent

April 26, 2021

1 Introduction ........................................ 2
2 Intuition .............................................. 2
3 Pseudocode ........................................... 5
4 Pseudocode Time Complexity ....................... 6
5 Thoughts, Problems, Ideas .......................... 7
6 Python Test and Output ............................. 8
1 Introduction

Consider the following graph:

Suppose we wish to find the shortest path between two vertices $s$ and $t$. How can we go about this?

2 Intuition

Intuitively the idea is this: First label all vertices with a (tentative distance) value of $\infty$ and with no predecessor.

Then we label the starting vertex $s$ as having distance 0 from itself and we put 0 on a queue of vertices which need to be dealt with.

Next we repeat the following until $t$ gets a distance label:

- Pop a vertex $v$ off the queue, go through all of the $\infty$ vertices $v$ is connected to, label each with $v$’s distance plus 1 and label its predecessor as $v$ and put each on the queue.

Once we label $t$ we have found its distance and we can use the predecessor labels to find the shortest path back to the origin.

In truth the distance labels are not necessary since we can use the length of the shortest path to calculate the distance.

Example 2.1. Let’s see how this works on a really easy graph. In the following suppose we wish to find the shortest path path from vertex $s = 0$ to vertex $t = 7$:

The first thing we do is label all the vertices with a value of $\infty$ except for $s = 0$ itself which we label with a value of 0. We also assign the queue.
We then pop 0 off the queue and we go over all the $\infty$ vertices connected to 0 by an edge (those are 1,3) and we label those with the value $0 + 1 = 1$ and their predecessor as 0. We note that vertex 7 is not amongst them. We also push them onto the queue.

We then pop 1 off the queue and we go over all the $\infty$ vertices connected to 1 by an edge (those are 2,4) and we label those with the value $1 + 1 = 2$ and their predecessor as 1. We note that 7 is not amongst them. We also push them onto the queue.

We then pop 3 off the queue and we go over all the $\infty$ vertices connected to 3 by an edge (that is 6) and we label that with the value $1 + 1 = 2$ and its predecessor as 3. We note that 7 is not amongst them. We also push it onto the queue.
We then pop 2 off the queue and we go over all the \( \infty \) vertices connected to 2 by an edge (that is 5) and we label that with the value \( 2 + 1 = 2 \) and its predecessor as 2. We note that 7 is not amongst them. We also push it onto the queue.

\[
\begin{aligned}
&\text{Queue: } [4, 6, 5] \\
&\text{We then pop 4 off the queue and we go over all the } \infty \text{ vertices connected to 4 by an edge (that is 7) and we label that with the value } 2 + 1 = 3 \text{ and its predecessor as 4. We note that 7 is amongst them so we return it and are done. The pseudocode does not push it onto the queue here because the push happens after the return conditional.}
\end{aligned}
\]

We then pop 4 off the queue and we go over all the \( \infty \) vertices connected to 4 by an edge (that is 7) and we label that with the value \( 2 + 1 = 3 \) and its predecessor as 4. We note that 7 is amongst them so we return it and are done. The pseudocode does not push it onto the queue here because the push happens after the \textbf{return} conditional.

\[
\begin{aligned}
&\text{Queue: } [6, 5] \\
&\text{Since we’ve located vertex 7 we can backtrack using the predecessors. The predecessors go 4, 1, 0 so the path is } (0, 1, 4, 7) \text{ and has length 3.}
\end{aligned}
\]

Note that the predecessor array in this case is:

\[
P = [\text{NULL}, 0, 1, 0, 1, 2, 3, \text{NULL}]
\]

We extract the path by observing that:

\[
\]
3 Pseudocode

Here is the pseudocode. The distance and predecessor data is stored in lists (if each vertex is an object these can instead just be properties of the object). The list of vertices to deal with next is stored in a queue.

```
function shortestpath(G, s, t)
    dist = distance array of size n full of inf
    pred = predecessor array of size n full of null
    Q = empty queue
    dist[s] = 0
    \\ Push s onto Q so it's the next (first) vertex we deal with.
    Q.push(s)
    \\ While there are vertices to deal with,
    \\ pop one off and analyze its adjacent nodes.
    \\ If unlabeled, label and push onto the queue.
    while Q is nonempty
        x = Q.pop
        \\ The next line presumes we can access the edge list.
        for each edge in the edge set of x
            let y be the connected vertex
            if dist[y] == inf
                dist[y] = dist[x] + 1
                pred[y] = x
                if y == t
                    return(pred)
                end
                Q.push(y)
            end
        end
    end
    \\ POST: Returns pred, from which the path can be extracted.
```

Note that some pseudocode you’ll see for this include an array which keeps track of whether each vertex has been visited or not. This isn’t necessary since this can be determined based upon whether or not a vertex has a finite distance assigned.
4 Pseudocode Time Complexity

The best-case is easy - this can be $\Theta(1)$ if $s = t$

The worse-case is not as obvious. It’s tempting to look at the pseudocode for this but we need to be careful. Let’s see why.

Let’s assume for now that the initialization step before the while loop is $\Theta(1)$.

Since no vertex ever gets put on the queue more than once the while loop iterates at most $n$ times.

It’s tempting to think that if each vertex is connected to every other vertex that the inner for each edge loop might also run $n$ times, giving a time complexity of $O(n^2)$. However while not wrong, this goes a little overboard because we can see that if each vertex were connected to every other vertex that $t$ would be found immediately.

Instead, observe that it’s certainly the case that in a worst-case we may have to visit all vertices and in a worst-case we may have to scan all edges. It’s also true that if we have scanned all vertices and all edges that we have certainly found our target.

Since the two iterative processes together take care of a vertex and an edge the total number of iterations necessary will certainly be no more than $|E| + n$, where $|E|$ is the number of edges.

Thus we can say for certain that in the worst-case $T(n, |E|) = O(n + |E|)$.
5 Thoughts, Problems, Ideas

1. Is it more advantageous to have a graph $G$ represented by an adjacency matrix or an adjacency list in order to implement this pseudocode algorithm in actual code? Explain.

2. Modify the pseudocode so that it returns the length of the shortest path from $s$ to $t$.

3. This algorithm may be modified to find a shortest path tree by not targeting a specific vertex but rather proceeding until all vertices have been accounted for. In this case the predecessor list which is returned can be used to reconstruct the entire tree. Give the pseudocode for this algorithm.

4. Suppose the graph were weighted instead of unweighted. If the line

   \[ \text{dist}[y] = \text{dist}[x] + 1 \]

were replaced by

   \[ \text{dist}[y] = \text{dist}[x] + \text{edgeweight}(x \text{ to } y) \]

would this effectively find the shortest total weighted distance from $s$ to $t$? If so, explain why. If not, explain why with a graph.

5. Modify the shortest path algorithm so that it finds the shortest path from $s$ to itself (a cycle) and returns FALSE if no such cycle exists.
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.

```python
def shortestpath(AL, n, u, up):
    dist = [float('inf')] * n
    pred = [None] * n
    Q = []
    dist[u] = 0
    print('Dist: ' + str(dist))
    Q.append(u)
    print('Queue: ' + str(Q))
    while len(Q) != 0:
        x = Q.pop(0)
        print('Pop ' + str(x) + ' and process vertices ' + str(AL[x]))
        for y in AL[x]:
            if dist[y] == float('inf'):
                print('__Process: ' + str(y))
                dist[y] = dist[x] + 1
                print('__Dist: ' + str(dist))
                pred[y] = x
                if y == up:
                    return (dist, pred)
                Q.append(y)
                print('__Queue: ' + str(Q))
            else:
                print('__Already done: ' + str(y))

AL = [
    [1, 3],
    [0, 2, 4],
    [1, 5],
    [0, 4, 6],
    [1, 3, 7],
    [2, 8],
    [3, 8],
    [4],
    [5, 6]
]
n = 9
u = 0
up = 7
[dist, pred] = shortestpath(AL, n, u, up)

path = []
x = up
while x != None:
    path.append(x)
    x = pred[x]
path.reverse()
print('Path: ' + str(path))
print('Length: ' + str(len(path) - 1))
```

AL =
    [[1, 3],
     [0, 2, 4],
     [1, 5],
     [0, 4, 6],
     [1, 3, 7],
     [2, 8],
     [3, 8],
     [4],
     [5, 6]]

n = 9
u = 0
up = 7
[dist, pred] = shortestpath(AL, n, u, up)

path = []
x = up
while x != None:
    path.append(x)
    x = pred[x]
path.reverse()
print('Path: ' + str(path))
print('Length: ' + str(len(path) - 1))
Output:

```
Dist: [0, inf, inf, inf, inf, inf, inf, inf, inf]
Queue: [0]
Pop 0 and process vertices [1, 3]
__Process: 1
__Dist: [0, 1, inf, inf, inf, inf, inf, inf, inf]
__Queue: [1]
__Process: 3
__Dist: [0, 1, inf, 1, inf, inf, inf, inf, inf]
__Queue: [1, 3]
Pop 1 and process vertices [0, 2, 4]
__Already done: 0
__Process: 2
__Dist: [0, 1, 2, 1, inf, inf, inf, inf, inf]
__Queue: [3, 2]
__Process: 4
__Dist: [0, 1, 2, 1, 2, inf, inf, inf, inf]
__Queue: [3, 2, 4]
Pop 3 and process vertices [0, 4, 6]
__Already done: 0
__Already done: 4
__Process: 6
__Dist: [0, 1, 2, 1, 2, inf, 2, inf, inf]
__Queue: [2, 4, 6]
Pop 2 and process vertices [1, 5]
__Already done: 1
__Process: 5
__Dist: [0, 1, 2, 1, 2, 3, 2, inf, inf]
__Queue: [4, 6, 5]
Pop 4 and process vertices [1, 3, 7]
__Already done: 1
__Already done: 3
__Process: 7
__Dist: [0, 1, 2, 1, 2, 3, 2, 3, inf]
Path: [0, 1, 4, 7]
Length: 3
```