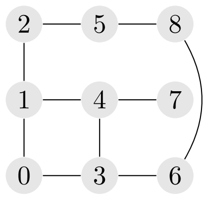
CMSC 351: Shortest Path Algorithm

Justin Wyss-Gallifent

July 30, 2024

1 Introduction

Consider the following graph:



Suppose we wish to find the shortest path between two vertices s and t, say $s = 0$ and $t = 7$. How can we go about this?

2 Algorithm Outline

Intuitively the idea is this: First assign all vertices with a (tentative distance) value of ∞ and with no predecessor. Technically we could assign them `NULL` instead but ∞ allows us to think of it as a distance that will be computed as the algorithm progresses.

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

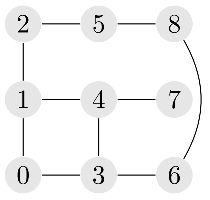
Next we repeat the following until t gets a distance assigned:

- Dequeue a vertex x, for each ∞ vertex y that x is connected to, assign y with x's distance plus 1 and additionally label it as having x as a predecessor, then put y on the queue.

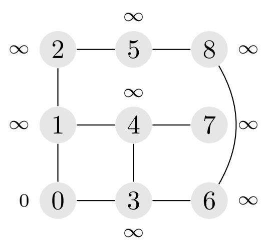
Once we finally assign t a distance we can use the predecessor labels to find the shortest path back to the origin.

3 Detailed Example

Example 3.1. 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 $t = 7$:

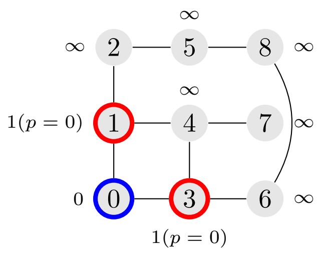


The first thing we do is assign all the vertices with a distance of ∞ except for $s = 0$ itself which we assign a distance of 0. We also initialize the queue.



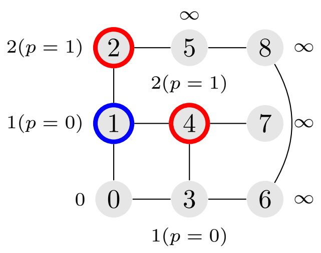
Queue: $[0]$

We then dequeue 0 and we go over all the ∞ vertices adjacent to 0 (those are 1,3) and we assign those distance $0 + 1 = 1$ and label them has having predecessor 0. We note that vertex 7 is not amongst them. We also enqueue them.



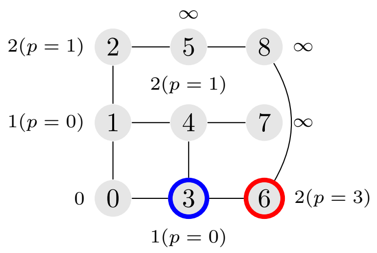
Queue: $[1, 3]$

We then dequeue 1 and we go over all the ∞ vertices adjacent to 1 (those are 2,4) and we assign those distance $1 + 1 = 2$ and label them has having predecessor 1. We note that 7 is not amongst them. We also enqueue them.



Queue: $[3, 2, 4]$

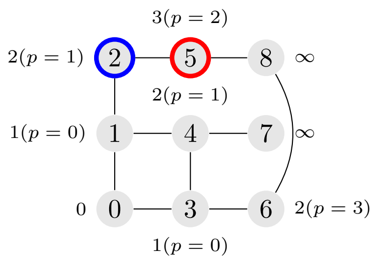
We then dequeue 3 and we go over all the ∞ vertices adjacent to 3 (that is 6) and we assign that distance $1 + 1 = 2$ and label it as having predecessor 3. We note that 7 is not amongst them. We also enqueue it.



Queue: $[2, 4, 6]$

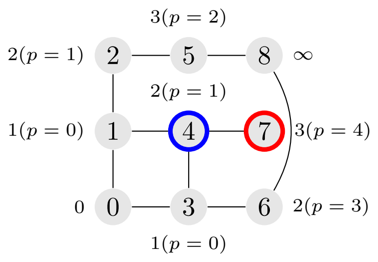
We then dequeue 2 and we go over all the ∞ vertices adjacent to 2 (that is

5) and we assign that distance $2 + 1 = 2$ and label it as having predecessor 2. We note that 7 is not amongst them. We also enqueue it.



Queue: $[4, 6, 5]$

We then dequeue 4 and we go over all the ∞ vertices adjacent to 4 (that is 7) and we assign that distance $2 + 1 = 3$ and label it as having predecessor 4. We note that 7 is amongst them so we return it and are done. The pseudocode does not enqueue it onto the queue here because the `enqueue` happens after the `return` conditional.



Queue: $[6, 5]$

Since we've located vertex 7 we can backtrack using the predecessors. The predecessors go $4, 1, 0$ so the path is $\langle 0, 1, 4, 7\rangle$ and has length 3.

Note that the predecessors can be stored as a list as:

$$P = [NULL, 0, 1, 0, 1, 2, 3, 4, NULL]$$

We extract the path by observing that:

$$P[7] = 4,\ P[4] = 1,\ P[1] = 0$$

4 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 V full of inf
    pred = predecessor array of size V full of NULL
    Q = empty queue
    dist[s] = 0
    Q.enqueue(s)
    while Q is nonempty
        x = Q.dequeue
        for each infinity vertex y adjacent to x
            dist[y] = dist[x] + 1
            pred[y] = x
            if y == t
                return(pred)
            end
            Q.enqueue(y)
        end
    end
end
```

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.

5 Pseudocode Time Complexity

The best-case is easy, if s and t are connected by an edge and if t is the first vertex visited from s then assuming that it takes $\Theta(V)$ to initialize `dist` and `pred` then the total time is $\Theta(V)$

The worst-case is not as obvious.

- There is $\Theta(V)$ time required inside the function but excluding the while loop.
- The while loop could iterate as many $V-1$ times. and the body of the while loop, excluding the for loop, takes constant time. This then a total of $\Theta(V)$.
- Over the course of the entire algorithm the body of the for loop iterates twice for each edge, taking constant time for each, so that's $\Theta(2E) = \Theta(E)$.

Thus we can say for certain that in the worst-case:

$$T(V,E) = \Theta(V+E)$$

In addition note that for an undirected simple graph we have $E \leq C(V,2) = \frac{1}{2}V(V-1)$ since at most each pair of vertices may be connected by an edge.

Thus we could also say that $T(V) = \mathcal{O}(V^2)$.

Again, just to really reiterate, this is assuming that we have the graph stored as an adjacency list.

6 Pseudocode Auxiliary Space

The auxiliary space is $\Theta(V)$ to store dist and pred, each of which have length V, to store Q, which has length less than V, and to store a couple of extra variables.

7 Removing the Target

A classic modification of this is to remove the target t and allow the code to simply run until the queue is empty and return the pred list afterwards. Here is the pseudocode in brief, with no comments:

```
function shortestpath(G,s)
    dist = distance array of size n full of inf
    pred = predecessor array of size n full of null
    Q = empty queue
    dist[s] = 0
    Q.enqueue(s)
    while Q is nonempty
        x = Q.dequeue
        for each vertex y connected to x
            if y has a label of infinity
                dist[y] = dist[x] + 1
                pred[y] = x
                Q.enqueue(y)
            end
        end
    end
    return(pred)
end
```

The pred list will then return a list of predecessors which implicitly yields the shortest paths from s to each and every vertex in the graph.

8 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

   ```
   dist[y] = dist[x] + 1
   ```

   were replaced by

   ```
   dist[y] = dist[x] + edgeweight(x 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.

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 vertices in the order in which they were visited.

```
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))
```

Output:

```
Dist: [0, inf, inf, inf, inf, inf, inf, inf, inf]
Queue: [0]
Dequeue 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]
Dequeue 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]
Dequeue 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]
Dequeue 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]
Dequeue 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
```