HW 3

Yunhao Lin

October 8, 2018

1 a:

```
1:  procedure NumShortestPath
2:      for each vertecx V do
3:          initialize V's visited mark = 0
4:          initialize V's value = 0
5:          initialize V's count = 0
6:
7:      q = New Queue()
8:      Mark v as visited = 1
9:      Mark the v.value = 0 for length from the begin vertex to v
10:     Mark the v.count =1 for the number of possible ways to get here
11:     Add v to q
12:
13:     while q is not empty do
14:         mark z as the next vertex in the q (dequeue)
15:         for each successor c of z do
16:             if c is unvisited, visited = 0 then
17:                 c.value = z.value + 1
18:                 c.count = z.count
19:                 Mark c as visited, visited = 1
20:                 Add c to q
21:             else
22:                 if c.value == z.value + 1 then
23:                     c.value  stay unchanged
24:                     c.count+ = z.count
25:                 else if c.value > z.value + 1 then
26:                     c.value = w.value + 1
27:                     count = z.count
28:                 else if c.value < z.value + 1 then
29:                     do nothing;
30:     Return the count value in w.
```

2 b:

Claim: After every time a vertex is dequeued from the queue, which means at each iteration of the while loop, the count is correctly set to every visited vertex.
Base Case: At the begining vertex s, each successor of s has its count set to 1 by the algorithm as they are added to the queue. Which is correct and therefore base case hold.
Inductive hypothesis: After k times of the iteration of the while loop, each vertex x that has been inspected holds the correct count of the number of shortest path to x.

Inductive step: Prove it worked for $k+1$ iteration, when the $(k+1)$th vertex is dequeued and update all the successor of $(k+1)$ vertex.
Name the (k+1)th vertex as α. Every one of the neighbours of α has three conditions.
Case 1: The neighbour is never visited before, we name this neighbour β. Since the value of α is already the shortest from v to α, in this case, the value of $(\alpha+1)$ would the shortest length from v to β. And the number of possible ways to get from α to β, is the same as the possible from v to α, cause there is only one way to get from α to β. This indicates if α is never visited before, algorithm would give the correct output for the number of shortest path at each vertex.
Case 2: The neighbour is visited before we name it γ. So it already has a value and a count.

1. When the path length from v to α to γ, which is $\alpha+1$ is shorter than the path length already in γ. Update the shortest length in γ. And the number of possible ways to get from α to γ, is the same as the possible from v to α, cause there is only one way to get from α to γ.

2. When the path length from v to α to γ is the same as the the path length that is already in γ. The value γ holds stay the same, since it represents the length from v to γ. But there is now also path from α to γ which is also the shortest path, the number of possible shortest path is now the number of possible path to α plus the original possible path. Which our algorithm correctly states from line 23 to 24.This would correctly change γ's value and count to represent the shortest path length and number of shortest path to γ.

3. Whent he path length from v to α to γ is longer than the path already in γ. There is no need to update the shortest path length and shortest path count in γ. Which is right.

   So for the neighbour that has been visited before, algorithm would create the correct output.

In general, in (k+1)th iteration, all successors of the (k+1)th vertex is updated correctly. Therefore, the algorithm works for (k+1)th iteration. By induction, The algorithm holds for number of n vertexs, n is a positive integer.

3 c:

Bascially, we use the implanmentation of BFS. At first we actually iterate through list to set up the value and count which means there would be at most (m+n) time here too. The operation of queuing and dequeue takes O(1) time, and we did that for each vertex just once, which would give us the O(n). Since adjecency list list only checked when the parent is dequeued, for each list there is only one time that they can be checked. But we have three extra if statement here which is O(1) and still makes the list checking in O(m) time. In total, the time is still O(m+n).