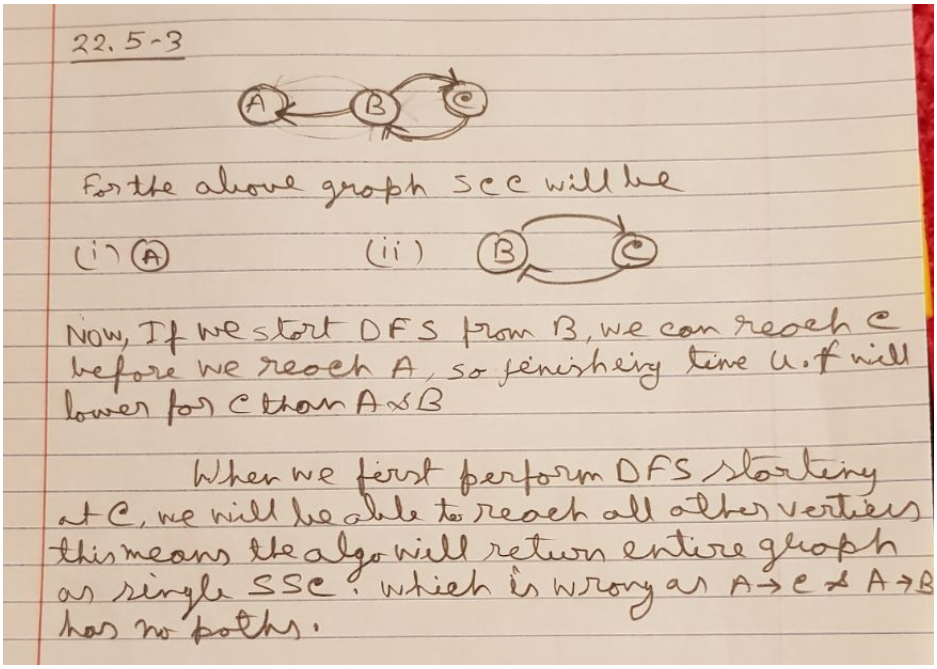
Solution to Homework Assignment 7(CS 430)

Saptarshi Chatterjee
CWID: A20413922

April 9, 2018

1 Q. 1. Exercise 22.5-3 on page 620 of CLRS3



2 Problem 22-1 (a) on page 621 of CLRS3

- 1 Suppose (u, v) is a back edge or a forward edge in a BFS of an undirected graph. Then one of u and , say u, is a proper ancestor of the other (v) in the breadth-first tree. Since we explore all edges of u before exploring any edges of any of u?s descendants, we must explore the edge (u, v) at

the time we explore u. But then (u, v) must be a tree edge.

- 2 In BFS, an edge (u, v) is a tree edge when we set v.? = u. But we only do so when we set v.d = u.d + 1. Since neither u.d nor v.d ever changes thereafter, we have v.d = u.d + 1 when BFS completes.

- 3 Consider a cross edge (u, v) where, without loss of generality, u is visited before v. At the time we visit u, vertex must already be on the queue, for otherwise (u, v) would be a tree edge. Because v is on the queue, we have v.d ? u.d + 1 by Lemma 22.3. By Corollary 22.4, we have v.d ? u.d. Thus, either v.d = u.d or v.d = u.d + 1. [2]

3 Given a graph G with weighted edges and a minimum spanning tree T of G, give and analyze an algorithm to update the minimum spanning tree when the weight of an edge e is G decreased

- Consider the case that edge e is decreased: if e is an edge in T, no need to change anything, since e still belongs to the MST after we decrease its weight.

- If deleting e from T, there obtains an intermediate spanning forest F. Then e is a safe edge with respect to F even before we decrease its weight. Thus decreasing the weight of e only makes it safer.

- If e is not an edge of T, the graph $U = T \cup \{e\}$ must contain a cycle, since there is a unique path in T between the endpoints of e. Let emax be the maximum-weight edge in this cycle. The new MST is $\hat{T} = U\{emax\}$. (e and emax could be the same edge.) We can use BFS/DFS to find emax in U in O(V + E) time. [3]

4 Given a directed graph G with positive-weight edges, a starting vertex s, and an ending vertex t, there may be more than one possible shortest path from s to t. The best shortest path is the path with the fewest edges.

(a) We can add a very small value (let say ϵ) to each edge . Now the path that will have higher number of edges will have higher cost than the path that have lesser number of edges where as both path had equal minimum path length. Now we need to be careful about choosing such small value , as lets say there was a path that had a higher cost previously , but lesser number of edges , then after this value modification a previously shorter path can end up having higher value as the number of ϵ added to it is higher .

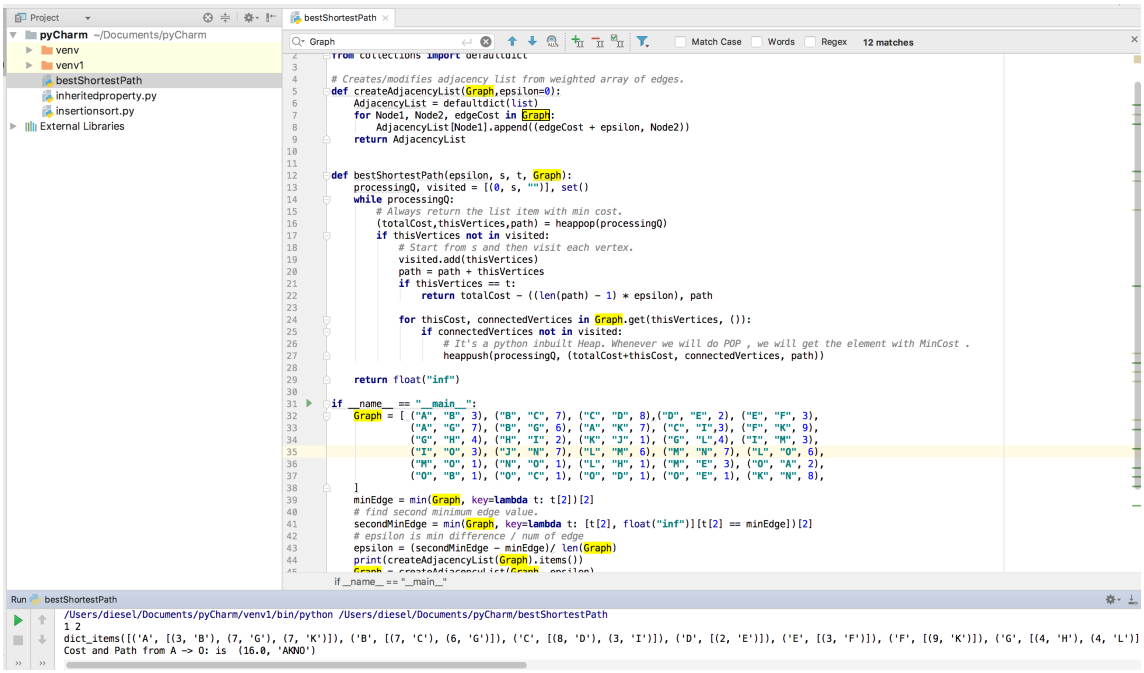
To solve this problem we take the difference between 2 smallest edges . and divide that value with number of edges to get ϵ.

Now lets say the graph has n edges . And it's shortest path contains (n-1) edges which is slightly high (this value will atleat be the difference between 2 minimum edge) than another path having just 1 edge. So after adding ϵ to all the edges , the minimum path is still minimum as atlest (n+1) epsilon to make it longer than the other path , but it just has (n-1) ϵ .

After value modification apply Dijkstra?s algo to find the shortest path between s and t.

$Complexity = Complexity to modify value + Complexity of Dijkstra?s = O(E + E \lg V) = O(E \lg V)$

b)



sourcecode

```
from heapq import heappop, heappush
from collections import defaultdict

# Creates/modifies adjacency list from weighted array of edges.
def createAdjacencyList(Graph,epsilon=0):
    AdjacencyList = defaultdict(list)
    for Node1, Node2, edgeCost in Graph:
        AdjacencyList[Node1].append((edgeCost + epsilon, Node2))
    return AdjacencyList


def bestShortestPath(epsilon, s, t, Graph):
    processingQ, visited = [(0, s, "")], set()
    while processingQ:
        # Always return the list item with min cost.
        (totalCost,thisVertices,path) = heappop(processingQ)
        if thisVertices not in visited:
            # Start from s and then visit each vertex.
            visited.add(thisVertices)
            path = path + thisVertices
            if thisVertices == t:
                return totalCost - ((len(path) - 1) * epsilon), path

            for thisCost, connectedVertices in Graph.get(thisVertices, ()):
                if connectedVertices not in visited:
                    # It's a python inbuilt Heap. Whenever we will do POP , we will get
                    # the element with MinCost .
                    heappush(processingQ, (totalCost+thisCost, connectedVertices, path))

    return float("inf")

if __name__ == "__main__":
    Graph = [ ("A", "B", 3), ("B", "C", 7), ("C", "D", 8),("D", "E", 2), ("E", "F", 3),
              ("A", "G", 7), ("B", "G", 6), ("A", "K", 7), ("C", "I",3), ("F", "K", 9),
              ("G", "H", 4), ("H", "I", 2), ("K", "J", 1), ("G", "L",4), ("I", "M", 3),
              ("I", "O", 3), ("J", "N", 7), ("L", "M", 6), ("M", "N", 7), ("L", "O", 6),
              ("M", "O", 1), ("N", "O", 1), ("L", "H", 1), ("M", "E", 3), ("O", "A", 2),
              ("O", "B", 1), ("O", "C", 1), ("O", "D", 1), ("O", "E", 1), ("K", "N", 8),
    ]
    minEdge = min(Graph, key=lambda t: t[2])[2]
    # find second minimum edge value.
    secondMinEdge = min(Graph, key=lambda t: [t[2], float("inf")][t[2] == minEdge])[2]
    # epsilon is min difference / num of edge
    epsilon = (secondMinEdge - minEdge)/ len(Graph)
    print(createAdjacencyList(Graph).items())
```

```
Graph = createAdjacencyList(Graph, epsilon)
# (16.0, 'AKNO')   . Though multiple path exists with value 16.0 like ABCIO,AKJNO
print("Cost and Path from A -> O: is ", bestShortestPath(epsilon, "A", "O", Graph))

\includegraphics[scale=.4]{4301}
```

References

[1] CLSR Solution

texttthttp://sites.math.rutgers.edu/ ajl213/CLRS/Ch16.pdf

[2] Prev Work Solution
http : *//ranger.uta.edu/ huang/teaching/CSE*5311/*HW*4*Solution.pdf*

[3] Graph Work Solution
http : *//www.cs.jhu.edu/ zliu*39/*teaching/hw*6 − *sol.pdf*