Algorithms (6470) HW05

Alex Darwiche

July 26, 2024

Answers

Q1

(a) Prove that if a sequence of relaxation steps ever sets $s.\pi$ to non-NIL value, then G contains a negative weight cycle.

(b) The distance at the source node is 0, so s.d = 0.

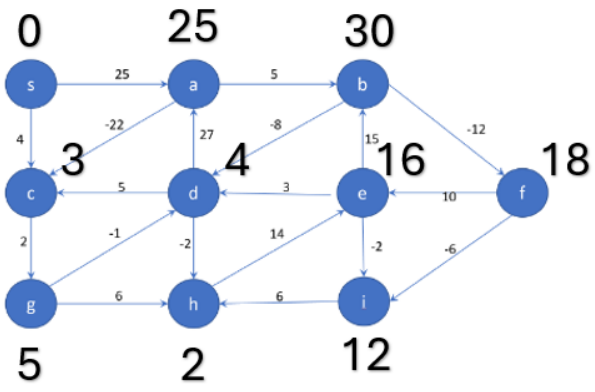
(c) For relaxation to ever make it back to s and change its π, there would need to be a vertex u, such that 0 > u.d + w(u,v).

(d) For this to be true, the distance at vertex u or the edge from u must be negative.

(e) This would mean that there is a negative weight cycle, as s.d would be going below 0.

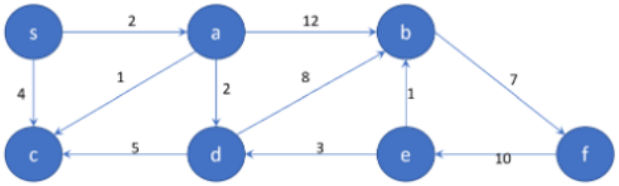
Q2

(a) Graph G - Bellman Ford Algorithm Results



Q3

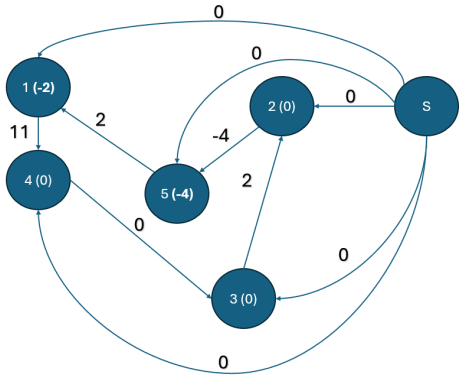
(a) Order of Addition of Vertices to Stack: S, A, C, C(lower cost), D, B, B(lower cost), F



1	2	3	4
SA (2)	SAC(3)	SADB(12)	SADBF(19)
SC (4)	~~SC(4)~~		
	SAD(4)		
	SAB(14)		

Q4

(a) Solving System of Difference Constraints



X1 = -2
X2 = 0
X3 = 0
X4 = 0
X5 = -4

Q5

(a) Prove that EXTEND is associative. Essentially show that if EXTEND(EXTEND(A,B),C) = EXTEND(A,EXTEND(B,C)).

(1) We will start by computing the left and right sides of the above equation with the given information.

(2) EXTEND(EXTEND(A,B),C) = $min(min(a_{ij} + b_{jk}) + c_{ij})$

(3) $min(a_{ij}) + min(b_{jk}) + min(c_{ij})$

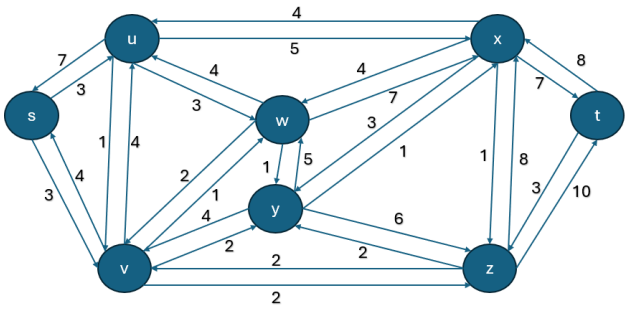
(4) EXTEND(A, EXTEND(B,C)) $= min(a_{ij} + min(b_{ij} + c_{jk}))$

(5) $min(a_{ij}) + min(b_{ij}) + min(c_{jk})$

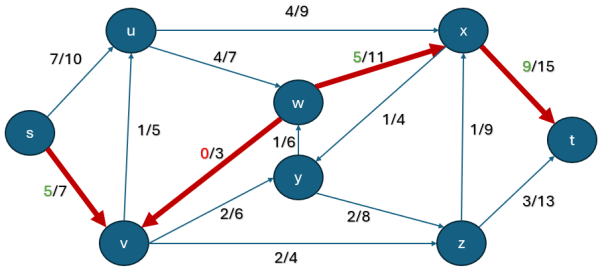
(6) As we can see, these expressions are equivalent and thus proves the associativity of the EXTEND algorithm.

Q6

(a) Residual Chart



(b) Augmenting path that reduces flow on one edge, increases on others.



(c) (s,v,w,x,t). The flow that goes through this path is 1.

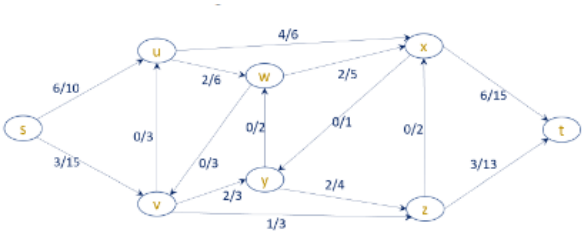
(d) See above for new net flow chart.

Q7

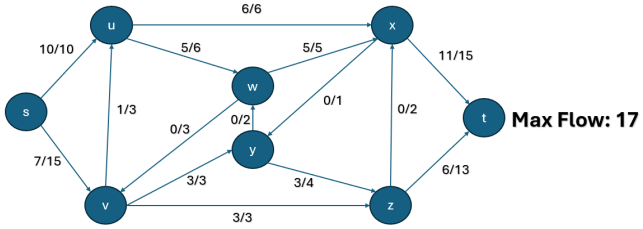
(a) $f(S,T) = 3 + 2 + 6 - 2 - 0 - 0 = 9$

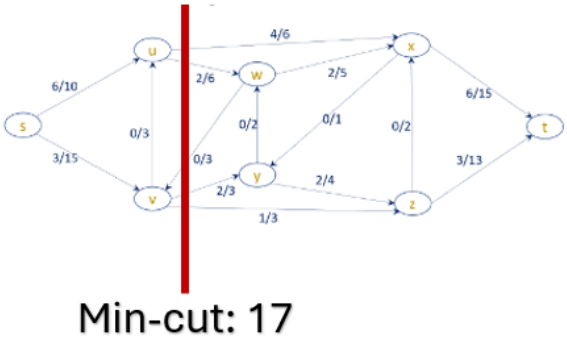
(b) $c(S,T) = 15 + 6 + 15 = 36$

(c) The following is the min-cut and max-flow of Graph G.



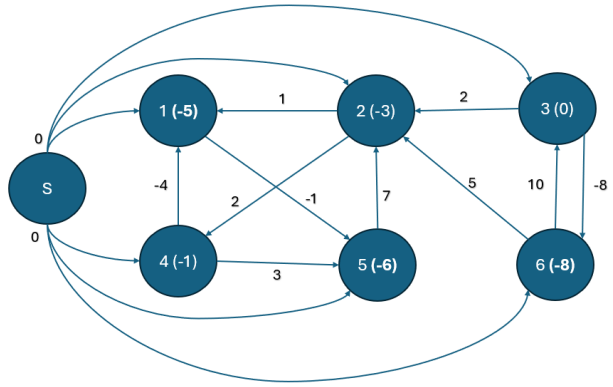
Let $S = \{s, u, x\}$ and $T = \{v, w, y, z, t\}$

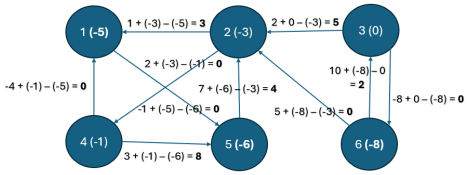


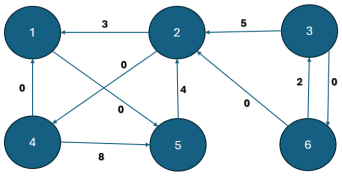


Q8

(a) Johnson's Algorithm, combining Bellman-Ford and Djikstra's Algorithm.







	1	2	3	4	5	6
1	-	4	INF	4	0	INF
2	0	-	INF	0	3	INF
3	0	0	-	0	0	
4	0	4	INF	-	0	INF
5	4	4	INF	4	-	
6	0	0	2	0	0	-

Q1 (Graduate students only)

(a) Psuedocode to count all paths. Basically we're running a DFS style search from each vertex in the graph, and counting each time we hit the goal vertex. This works as each branch of the DFS is a unique path with a distinct start point. To start the DFS from different vertices each time, we will simply reorder the adjaceny matrix each time, until all vertices have been the starting node.

```
def COUNT-PATHS(Graph G, Vertex v):

for vertex in G.V:
    \\ We will want to complete a DFS
    \\ for all vertices, and count each
    \\ time they end in the target Vertex v.
    G = reorder-adjaceny(G)
    path_count = DFS(G)

def DFS(G,v)
    for each vertex in G.V:
```

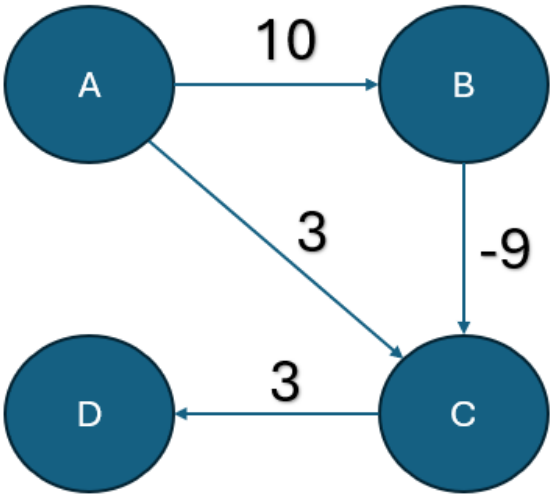
```
            count += DFS-visit(G,vertex,count)
        return count

    def DFS-visit(G,v,count)
        for each vertex in G.Adj[u]
            if vertex != v:
                DFS-visit(G,vertex,count)
            else:
                count = count + 1
        return count
```

(b) We also do not care about going over a path that has already been gone over, so the coloring shouldn't matter in this algorithm. We are simply looking to extend each neighbor of each child node until it reaches the goal node.

Q2 (Graduate students only)

(a) The following chart shows when Djikstra's algorithm will not work with negative edge weights. You can see that the goal state is achieved while all the AB(10) is still in the stack. This remains in the stack, unexplored, because its value is higher than the goal state. Djikstra's Algorithm makes the assumption that the goal node is reached when the lowest possible path to expand is includes the finishing node. You can clearly see that the shortest path actually involves going from B to C, but this algorithm will not explore that.



AC(3)	ACD(6)
AB(10)	AB(10)