Q1 18

1) 20 pts
Mark the following statements as **TRUE** or **FALSE**. No need to provide any justification.

[TRUE/FALSE]
For any edge e that is part of a minimum cut in a flow network G, if we increase the capacity of that edge by any integer k>1, then that edge will no longer be part of a minimum cut.

[TRUE/FALSE]
The sequence alignment algorithm described in class can be used to find the longest common subsequence between two given sequences.

[TRUE/FALSE]
The scaled version of the Ford-Fulkerson algorithm can compute the maximum flow in a flow network in polynomial time.

[TRUE/FALSE]
Given a set of demands D = {dv} on a circulation network G(V,E), if the total demand over V is zero, then G has a feasible circulation with respect to D.

[TRUE/FALSE]
In a flow network, the maximum value of an $s-t$ flow could be less than the capacity of a given $s-t$ cut in that network.

[TRUE/FALSE]
If f is a max $s-t$ flow of a flow network G with source s and sink t, then the capacity of the min $s-t$ cut in the residual graph G_f is 0.

[TRUE/FALSE]
In a graph with negative weight cycles, one such cycle can be found in O(nm) time where n is the number of vertices and m is the number of edges in the graph.

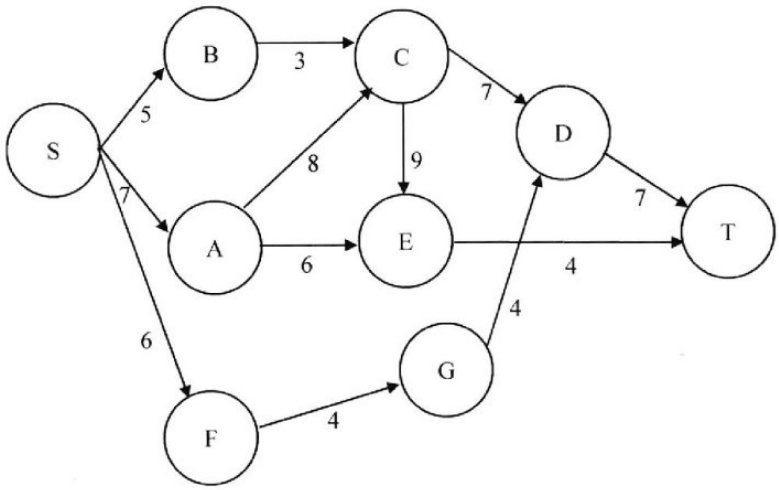
[TRUE/FALSE]
An algorithm runs in weakly polynomial time if the number of operations is bounded by a polynomial in the number of bits in the input, but not in the number of integers in the input.

[TRUE/FALSE]
Let G(V,E) be an arbitrary flow network, with a source s, a sink t. Given a flow f of maximum value in G, we can compute an s-t cut of minimum capacity in O(|E|) time.

[TRUE/FALSE]
The basic Ford-Fulkerson algorithm can be used to compute a maximum matching in a given bipartite graph in strongly polynomial time.

Q2 14

2) 20 pts
Perform two iterations (i.e. two augmentation steps) of the scaled version of the Ford-Fulkerson algorithm on the flow network given below. You need to show the value of Δ and the augmentation path for each iteration, and the flow f and $G_f(\Delta)$ after each iteration. (Note: iterations may or may not belong to the same scaling phase)



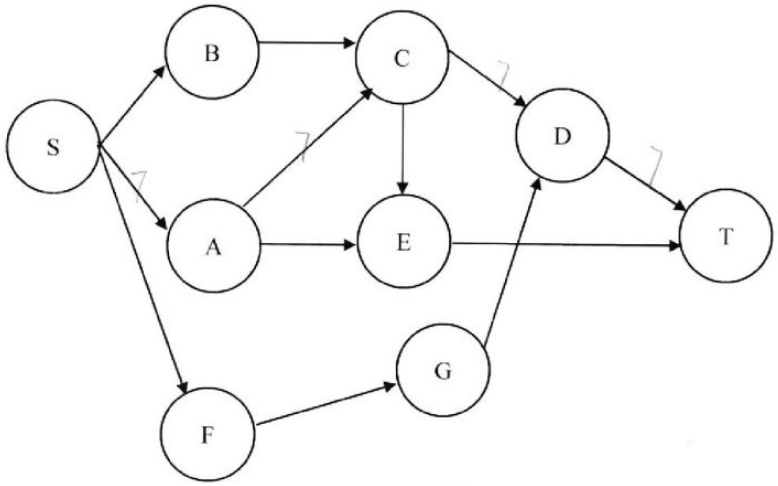
See next page…

(a) Iteration 1: (8 pts)

Δ is 4

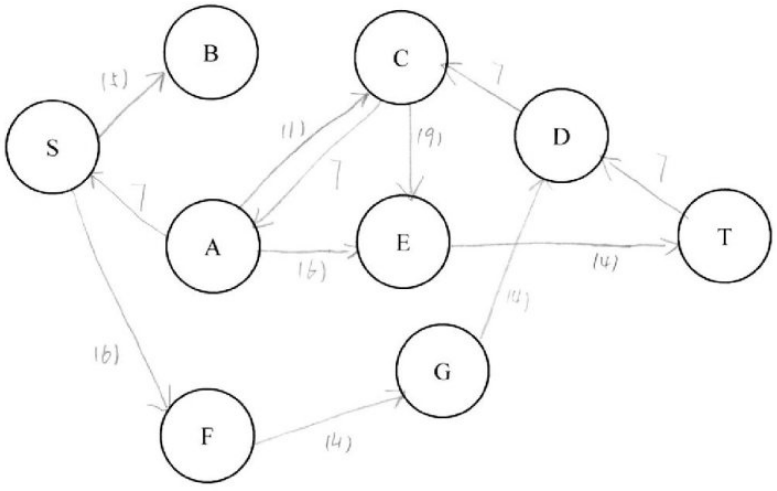
Augmentation Path: S->A->C->D->T

Flow after the first iteration (you can write flow values over each edge carrying flow):



the number in parenthesis is the residual capacity of that edge

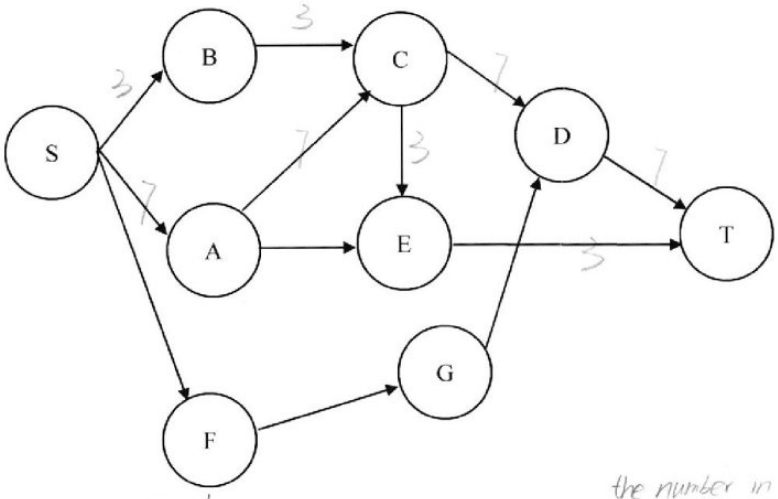
$G_f(\Delta)$ after the first iteration (you can write flow values over each edge carrying flow):



(b) Iteration 2: (8 pts)
Δ is 3

Augmentation Path: S->B->C->E->T

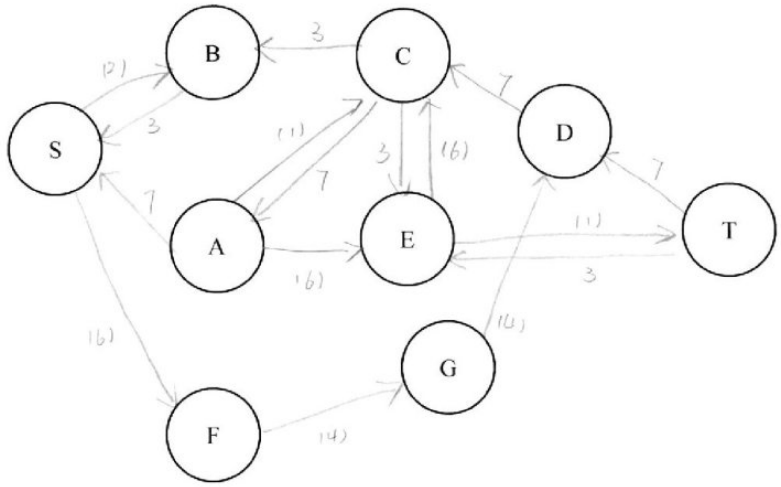
Flow after the second iteration (you can write flow values over each edge carrying flow):



second

the number in parenthesis is the residual capacity of that edge

$G_f(\Delta)$ after the first iteration (you can write flow values over each edge carrying flow):



(c) Can the choice of augmentation paths in the scaled version of Ford-Fulkerson affect the number of iterations? Explain why. (4 pts)

Yes, because the choice of augmentation paths will affect the number of iterations in the inner loop during one scaling phase.

3

Q3 20

3) 20 pts
The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. Each of these operations has unit cost.

For example, the Levenshtein distance between "kitten" and "sitting" is 3. A minimal edit script that transforms the former into the latter is:

kitten -> sitten
sitten -> sittin
sittin -> sitting

We want to design a dynamic programming algorithm that calculates the Levenshtein distance between a string X of length m and another string Y of length n. An edit can be adding, removing, or changing a character in X.

a) Define (in plain English) subproblems to be solved. (4 pts)

OPT(i,j) is the Levenshtein distance between the substring $(x_1 \cdots x_i)$ of string X and the substring $(y_1 \cdots y_j)$ of string Y

b) Write the recurrence relation for subproblems. (6 pts)

More explanation on Page 11

$$OPT(i,j) = \begin{cases} OPT(i-1,j-1) & \text{if } x_i = y_j \\ \min(OPT(i-1,j-1)+1,\ OPT(i-1,j)+1,\ OPT(i,j-1)+1) & \text{if } x_i \neq y_i \end{cases}$$

base case: $OPT(0,j) = j$ for $j = 0$ to n, $OPT(i,0) = i$ for $i = 0$ to m

c) Using the recurrence formula in part b, write pseudocode (using iteration) to compute the Levenshtein distance between strings X and Y. (6 pts)
Make sure you have initial values properly assigned. (2 pts)

```
Assign initial values
FOR i=0 to m, set OPT[i,0] = i
FOR j=0 to n, set OPT[0,j] = j
FOR i=1 to m
    FOR j=1 to n
        IF X.charAt(i) = X.charAt(j) OPT[i,j] = OPT[i-1,j-1]
            OPT[i,j] = OPT[i-1,j-1]
        ELSE OPT[i,j] = min(OPT[i-1,j-1]+1, OPT[i-1,j]+1, OPT[i,j-1]+1
    ENDFOR
ENDFOR
Return OPT[m,n]
```

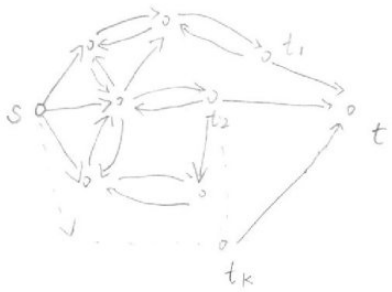
d) Compute the runtime of the algorithm described in part c and state whether your solution runs in polynomial time or not (2 pts)

:D
Since there are $m \times n$ entries of the array OPT[·,·] and each entry only take constant time to compute. the runtime of the above algorithm is $O(mn)$
It is polynomial since m and n are the length of the input strings X and Y.

Q4 20

4) 20 pts

Suppose a concert has just ended and C cars are parked at the event. We would like to determine how long it takes for all of them to leave the area. For this problem, we are given a graph representing the road network where all cars start at a particular vertex s (the parking lot) and several vertices ($t_1, t_2, \ldots, t_k$) are designated as exits. We are also given capacities (in cars per minute) for each road (directed edges). Give a polynomial-time algorithm to determine the amount of time necessary to get all cars out of the area.



$|V| = k$

Build a flow network $N(G(V,E), s, t)$ based on the given graph.

Set s as the source node and delete all edges going into s since no one will go back to the parking lot

Add a node as the sink node and add an edge going from each exit vertex to t with infinity capacity.

Set all other edges with the given capacities for each road.

Run Edmond-Karp algorithm to get the max flow on flow network N

Define v as the value of the max flow

Return $\frac{C}{v}$ as the amount of time neressary to get all cars out of the area.

Transform the given graph to the flow netwoik only takes $O(|V|)$ time

Since Edmond-Karp algorithm is a polynomial one to compute the max flow

This algorithm is polynomial.

Q5 15

5) 20 pts

There are n companies participating in a trade show. Company i gives away goodies worth g_i at their booth to each person visiting their booth. But you might have to wait in a line to get to the booth.

a) Knowing that the wait time at booth i is $w_i \geq 0$, formulate a solution that will earn you a minimum of G dollars' worth of goodies without spending more than a total of H hours waiting in lines. Your solution should also indicate, given G and H, if this objective is not possible to achieve. (18 pts)

Firstly I will use dynamic programming to calculate the maximum dollars' worth of goodies I will get spending no more than H total hours waiting in lines.

Define $OPT(i,t)$ as the max. dollars' worth to get from the first i companies with no more than t waiting time.

recurrence formula: $OPT(i,t) = \begin{cases} \max(OPT(i-1,t), OPT(i-1,t-w_i)+g_i) & \text{if } t \geq w_i \\ OPT(i-1,t) & \text{if } t < w_i \end{cases}$

base case: $OPT(0,t) = 0$ for $t = 0$ to H,
$OPT(i,0) = 0$ for $i = 1$ to n

no pseudocode

Then I can get $OPT(n,H)$ by using the recurrence formula above.

If $OPT(n,H) < G$, then it is not possible to achieve that objective.

Else, that objective is attainable and I can compute the optimal solution.

Define array $S[1..n]$ as $S[i]$ indicate whether I should wait at company i to get goodies or not.

More on Page 10!

b) Analyze the complexity of your solution and determine if it is an efficient solution. (2 pts)

Since there are $n \times H$ entries in the $OPT(\cdot)$ array, and each entry only need constant time to compute, it takes $O(n \cdot H)$ to get $OPT(n,H)$.

More on Page 10!

Additional Space

5) a) For $i = n$ to 1, initially $h = H$

if $OPT(i,h) = OPT(i-1,h)$, set $S[i]$ = false.

else set $S[i]$ = true, and $h = h - W_i$

End for

b) It only take $O(n)$ time to compute the optimal solution if $OPT(n,H) \geq G$.

Therefore, the overall time complexity of my solution is $O(Hn)$

This is not an efficient solution since it is pseudo-polynomial

Additional Space

3) b) When computing OPT(i, j), if the ith letter at String X is the same as the jth letter at String Y, then we do not need to do any operation.

If they are not same, there will be 3 ways to make them same

One is add a letter if $(x_1 \cdots x_{i-1})$ and $(y_1 \cdots y_{j-1})$ are direly same

One is remove

change