CSCI570 Homework 4

Name: Saunak Sahoo **USC ID: 3896400217**

Ans 1

Step 1: Defining subproblems to be solved

Let OPT[l,r] be the maximum coins we can collect by bursting balloons between indexes l, r $(1 \leq l, r \leq n)$

Step 2: Defining recurrence relation

We will iterate over nums and check what is the maximum number of coins that can be obtained when the current balloon is the last balloon to be burst

Cost to burst last balloon at index i = nums[l-1] X nums[i] X nums[r+1]

Cost to burst all balloons before the balloon at index i was burst = burstBalloons(l, index -1) + burstBalloons(index + 1, r)

Thus total cost = nums[l-1] X nums[i] X nums[r+1] + burstBalloons(l, index -1) + burstBalloons(index + 1, r)

Finally our recurrence relation:

OPT[l][r] = max{OPT[l][r] , nums[l-1] X nums[i] X nums[r+1] + burstBalloons(l, index -1) + burstBalloons(index + 1, r) }

These is no need to define base cases here as our recursive function will take care of filling OPT[n][n]

Step 3: Pseudocode

1. Insert 1 at the front and the back of array nums ie. nums[0] = nums[n+1] = 1
2. Initialize OPT[n][n] with all elements = -1
3. Define function burstBalloons(l, r):
    a. If l > r:
        i. return 0

b. if OPT[l][r] ≠ -1:
    i. return OPT[l][r]
c. Assign OPT[l][r] = 0
d. Iterate index from l to r+1:
    i. coins = nums[l-1] X nums[i] X nums[r+1]
    ii. coins = coins + burstBalloons(l, index -1) + burstBalloons(index + 1, r)
    iii. OPT[l][r] = max(OPT[l][r], coins)
e. return OPT[l][r]

4. Return OPT[1][n -2]

Step 4: Time complexity

From the above pseudocode we can see

Indexes l and r go over the nums array which has ~n elements. Hence this runs n^2 times

At step 4d we have our index iterator that also iterates over the nums array. Hence this loop runs n times

Thus total time complexity = $n^2 X n = O(n^3)$

Ans 2

Step 1: Defining subproblems to be solved

Let OPT[i,j] be the maximum money we can win where i, j is the index of the n coins (1≤i,j≤n)

Step 2: Defining recurrence relation

Case 1: If we chose the i^{th} coin with value vi, the opponent can either chose the next coin from the front i.e. (i+1)th coin or the coin at the back i.e. j^{th} coin. The opponent will try to minimize our output

OPT[i,j] = vi + min(OPT[i+2, j], OPT[i+1, j-1])
where (i+2,j) is the range of coins available to us if opponent choses the (i+1)th coin and where (i+1,j-1) is the range of coins available to us if opponent choses the jth coin

Case 2: If we chose the j^{th} coin with value vi, the opponent can either chose the next coin from the front i.e. i^{th} coin or the coin at the back i.e. (j-1)th coin. The opponent will try to minimize our output

OPT[i,j] = vj + min(OPT[i+1, j-1], OPT[i, j-2])
where (i+1, j-1) is the range of coins available to us if opponent choses the i^{th} coin and where (i, j-2) is the range of coins available to us if opponent choses the (j-1)th coin

Combining the cases we would like to get the max number of coins. So we have

OPT[i,j] = max{ vi + min(OPT[i+2, j], OPT[i+1, j-1]) , vj + min(OPT[i+1, j-1], OPT[i, j-2]) }

Step 3: Pseudocode

1. Initialize OPT[n][n]
2. Define function coin(i , j):
    a. If i>j OR i ≥n OR j<0:
        i. return 0
    b. If OPT[i][j] exists (i.e. if it has already been computed):
        i. return OPT[i][j]
    c. OPT[i,j] = max(vi + min(OPT[i+2, j], OPT[i+1, j-1]) , vj + min(OPT[i+1, j-1], OPT[i, j-2]))
    d. return OPT[i,j]
3. return coin(0, n-1)

Step 4: Time complexity

We can see from the above problem that i, j can take values up to n

Hence time complexity = $nXn = O(n^2)$

Ans 3

Step 1: Defining subproblems to be solved

Let OPT[j,i] be the number of ways to reach the tile (j,i) where 0<j<2 and 0<i<n-1

Step 2: Defining recurrence relation

Case 1: Stepping on a weak tile is not a valid way to reach tile
If BadTile[j][i] ==1 : OPT[j][i]=0

Case 2a: If the tile is strong and we are on the bottom row i.e. j=0:
OPT[j][i] = OPT[j][i-1] + OPT[j+1][i-1]

Case 2b: If the tile is strong and we are in the middle row i.e. j=1:
OPT[j][i] = OPT[j][i-1] + OPT[j+1][i-1] + OPT[j-1][i-1]

Case 2c: If the tile is strong and we are in the top row i.e. j=2:
OPT[j][i] = OPT[j][i-1] + OPT[j-1][i-1]

Base cases:

If BadTile[j][0] = 0: OPT[j][0] = 1

If BadTile[j][0] = 1: OPT[j][0] = 0

Step 3: Pseudocode

1. Initialize OPT[3][n]
2. Iterate index j from 0 to 2:
    a. if BadTile[j][0] = 0:
        i. OPT[j][0] = 1
    b. else if BadTile[j][0] = 1
        i. OPT[j][0] = 0
3. Iterate index j from 0 to 2:
    a. Iterate index i from 1 to n-1:
        i. if BadTile[j][i] =1:
            1. OPT[j][i]=0
        ii. Else:
            1. if j=0:
                a. OPT[j][i] = OPT[j][i-1] + OPT[j+1][i-1]
            2. else if j=1:
                a. OPT[j][i] = OPT[j][i-1] + OPT[j+1][i-1] + OPT[j-1][i-1]

3. else if j=2:

    a. OPT[j][i] = OPT[j][i-1] + OPT[j-1][i-1]

4. Return OPT[0][n-1] + OPT[1][n-1] + OPT[2][n-1]

Step 4: Time complexity

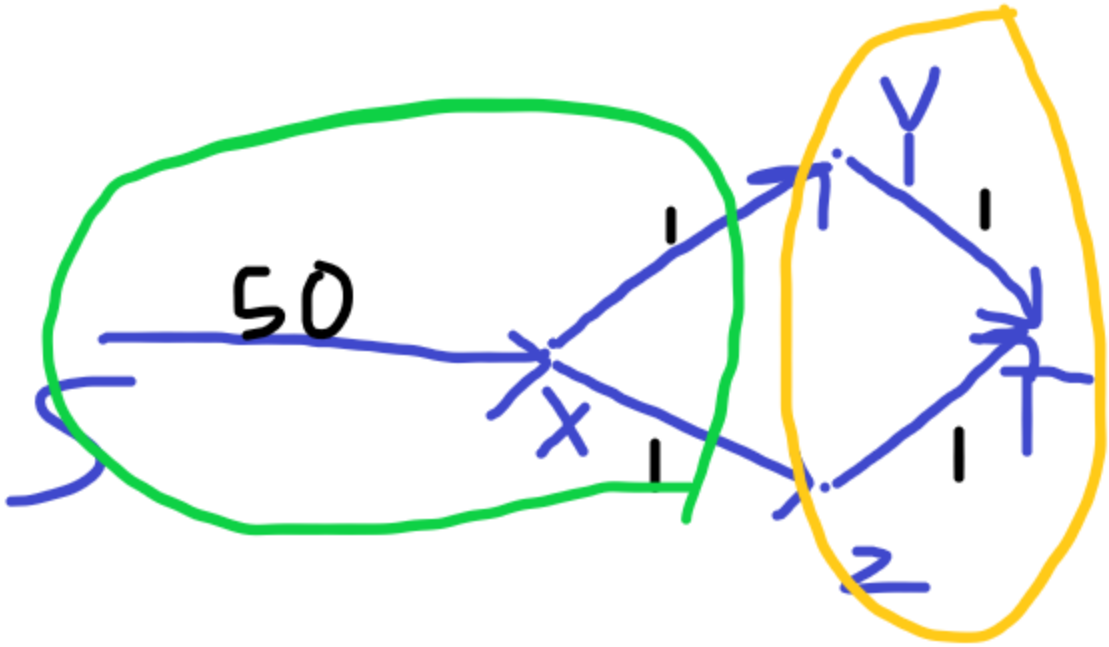
Our most expensive operation in the above pseudocode takes place at step 3a where we loop n times

Thus time complexity = $O(n)$

Ans 4

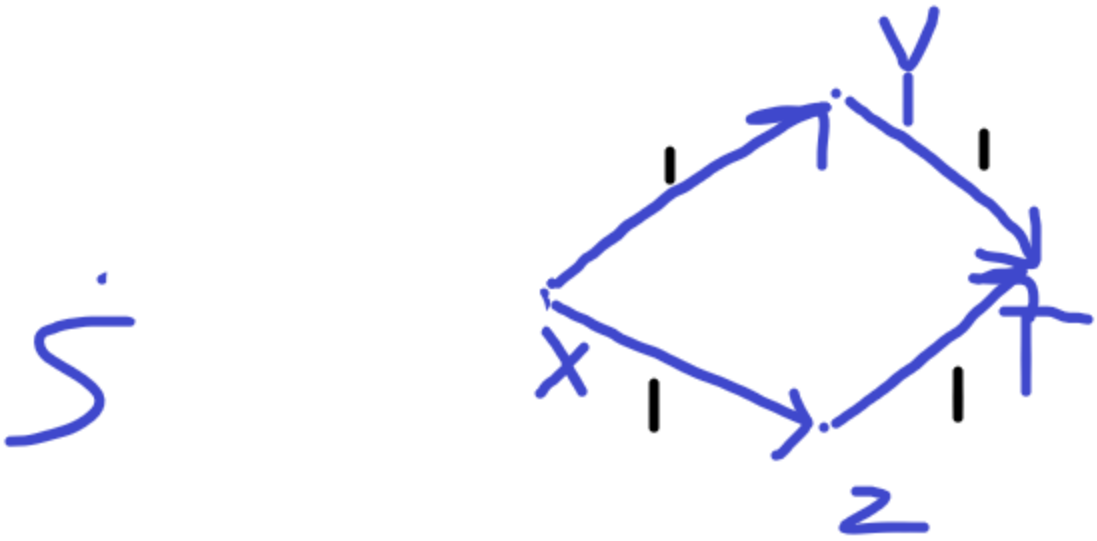
False

Proof by counter example:



Consider the above graph with min-cut A={S,X} and B = {Y, Z, T}. Here max-flow = 2

Let us consider edge e to be S→ X having capacity = 50. Let us delete this edge:



In this case the graph gets disconnected and the max-flow reduces to 0. We can see that deleting the edge e in this case reduces the max-flow the most and edge e was not a part of the min-cut. This is a contradiction to the statement: if deleting edge e reduces the original maximum flow more than deleting any other edge does, then edge e must be part of a minimum s−t cut in the original graph.

Ans 5

The statement “For a flow network with source s and the sink t, and positive integer edge capacities c - if we increase the capacity of every edge by 1, then s − t still be a minimum cut” will have 2 cases:

Case 1: All edge capacities are the same c:

Statement is **True**

Proof by contradiction:

The cut capacity of any cut in a graph with all edges having the same capacity c would be = nc where n is the number of edges cutting across a cut. If we increase capacity by 1 for each edge, we would have cut capacity for the same cut as = $n(c+1)$

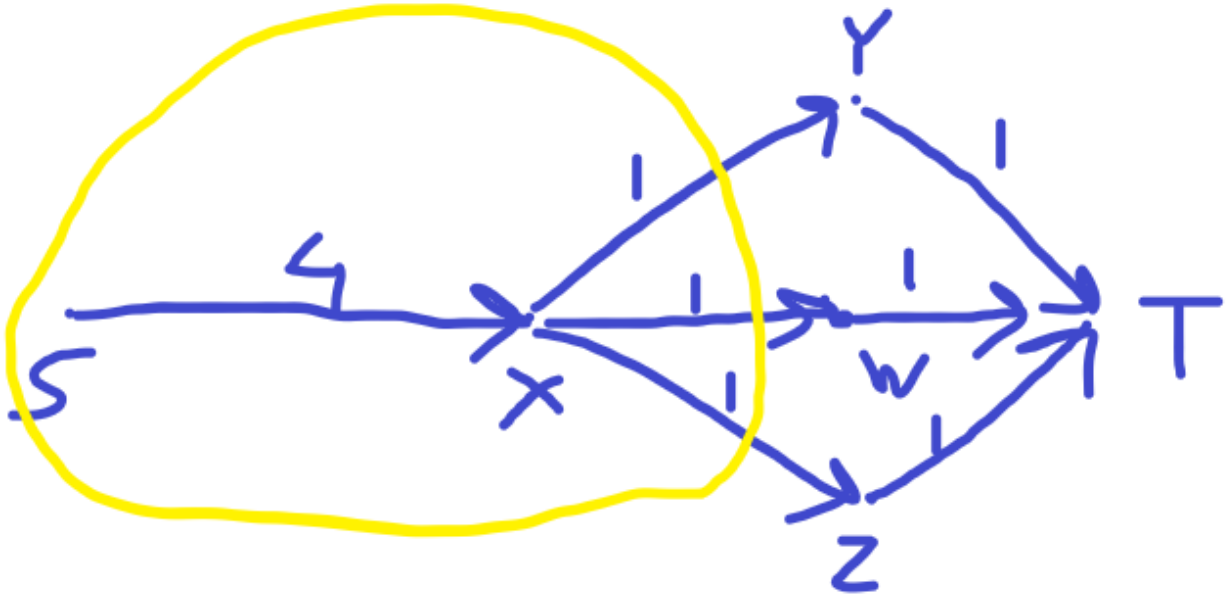
Say for our original graph min cut capacity = kc for k number of minimum edges. After modification this same cut capacity would be = $k(c+1)$

However, say this cut doesn't remain our min cut. Say we have a new min cut with capacity = $m(c+1)$ for m number of edges for some $m > k$. But this would mean that in our original graph this new cut would have capacity $mc > kc$ which contradicts the assumption that the $k-$edge cut is the min cut in the original graph. Hence we conclude that no such $m-$edge cut can exist, so our $k-$edge cut with modified cut capacity $k(c+1)$ is the new min cut

Case 2: All edge capacities are NOT the same:

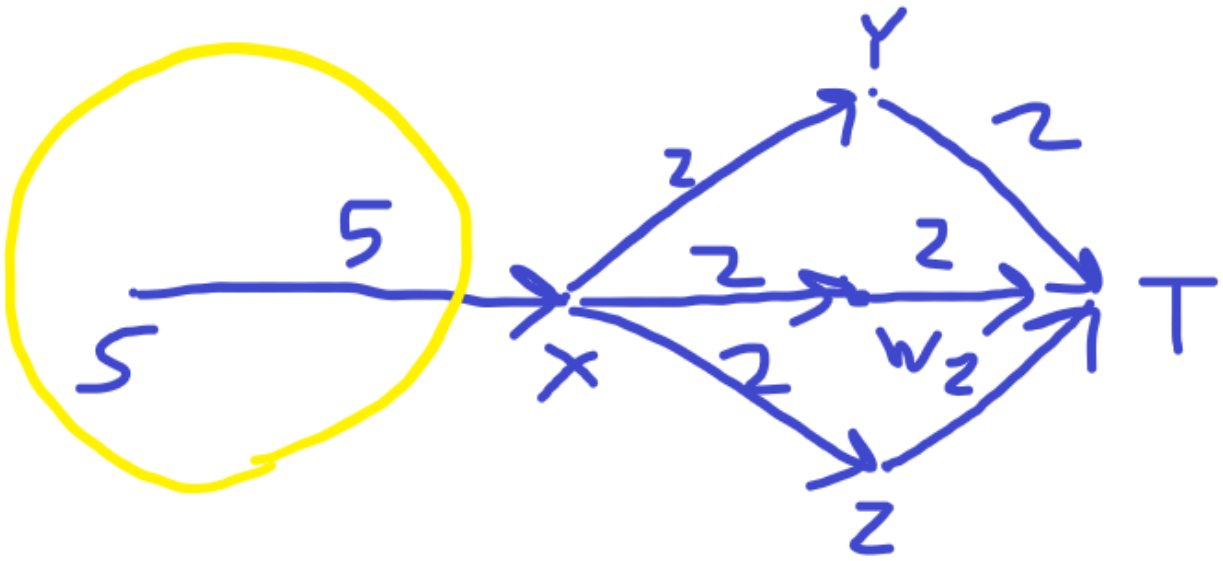
Statement is **False**

Proof by counterexample:



Consider the above graph with min-cut A={S,X} and B = {W, Y, Z, T}.

On increasing the edge capacities of each edge by 1 we get the following graph:



We can see that our min-cut has changed A={S} and B={W,X,Y,Z,T}. This is a contradiction to the above statement

Ans 6

a)

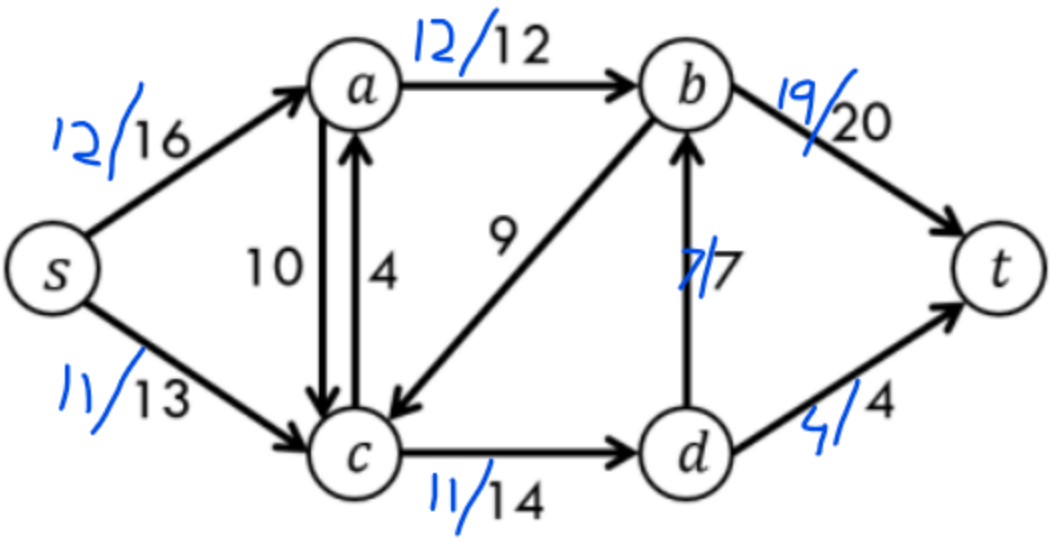
Running Ford-Fulkerson algorithm on the network to find out max flow

Steps:

1. Push f=7 through s-c-d-b-t
2. Push f=4 through s-c-d-t
3. Push f=12 through s-a-b-t

Hence max flow = 7 + 4 + 12 = 23

The residual graph with flow/capacity notation is as follows:



b)

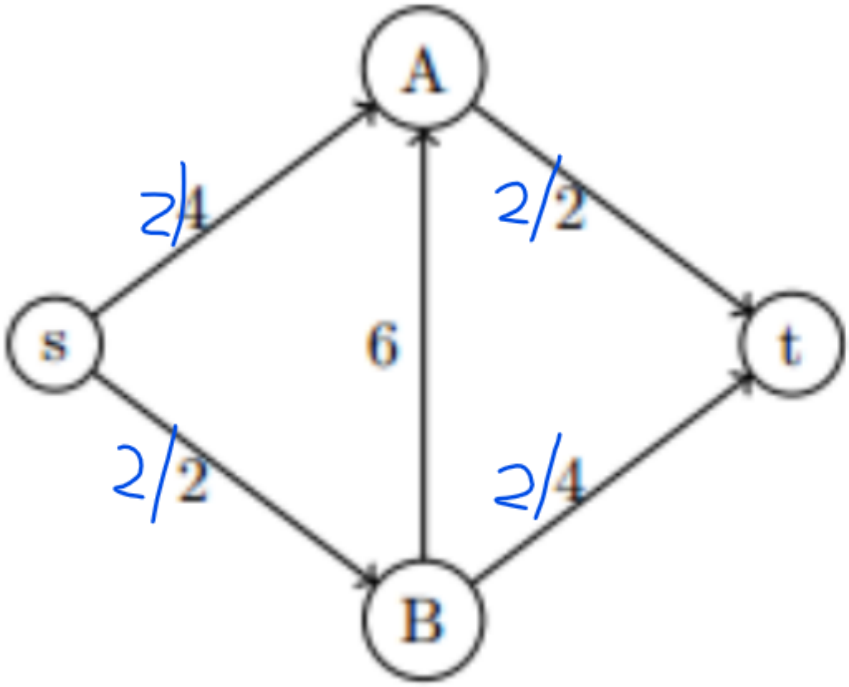
We know that the max flow = min cut value. Hence, we can use Ford-Fulkerson algorithm to find max flow which will also give us the value of the min cut

Steps:

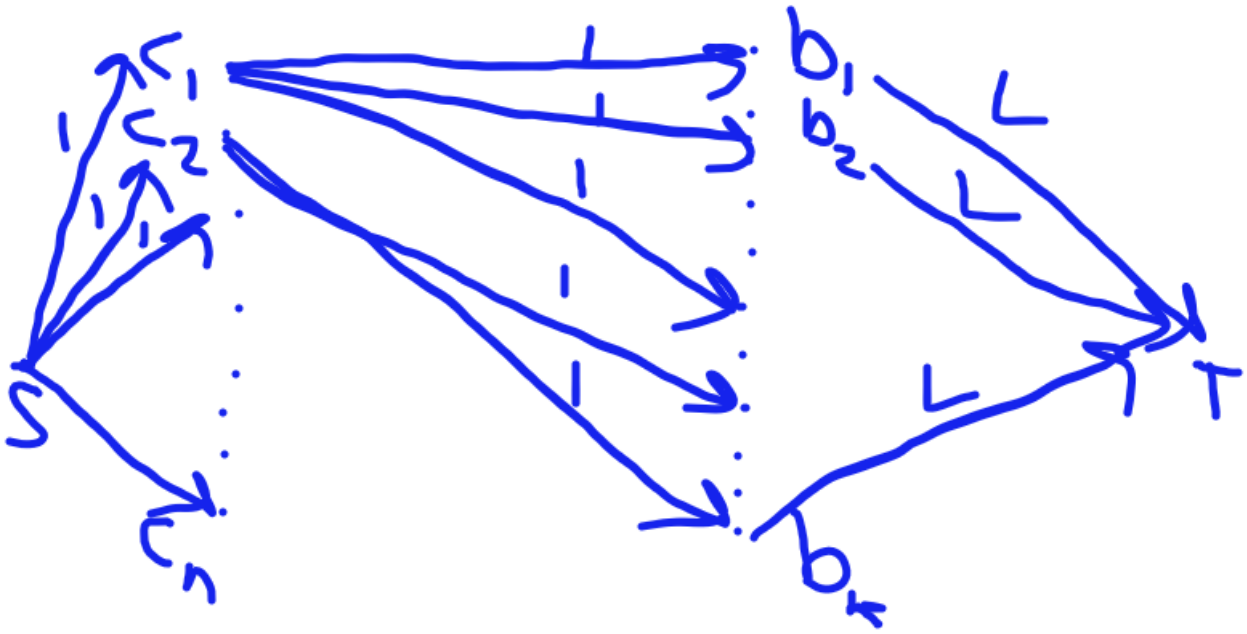
1. Push flow = 2 through s-B-t
2. Push flow = 2 through s-A-t

Hence max flow = 2 + 2 = 4 = value of min cut

The residual graph with flow/capacity notation is as follows:



Ans 7



Step 1: Creating a Network Flow

Create a network flow with clients represented as $c_1, c_2 ... c_n$ and base stations represented as $b_1, b_2 ... b_k$. A given client will be connected to a base station **only if it is**

within distance R (range parameter). Connect all the clients to a source node and all the base stations to a target node

- Assign capacity of 1 for each client that is connected to a base station to represent the fact that a client can be connected to only one base station
- Source node is connected to each client with capacity 1 to represent the fact that each client needs to be connected to a base station
- Each base station is connected to target with capacity L (load parameter) to represent the fact that a base station can be connected to at most L number of clients

Step 2: Making claim

Every client can be connected simultaneously to a base station if and only if the max flow = n (number of clients)

Step 3: Proving claim in both directions

In forward direction →
To prove: If every client is connected to a base station then max flow = n

We know that if every client $c_1, c_2 \ldots c_n$ is connected to a base station, then all the edges from source s to clients are saturated. Since the edge capacities is 1 for every such edge, the flow cannot be bigger than the number of clients i.e. n

In backward direction ←
To prove: Given the max flow = n, to find the actual assignments of clients to base stations

To find the base stations that the clients are connected to, we shall look at the saturated edges between the clients and the base stations

Ans 8

The algorithm is as follows:

1. We find the min-cut of the flow network by running Edmund Karp algorithm
2. We will then delete k edges that are going across the min-cut. 2 cases might arise here:

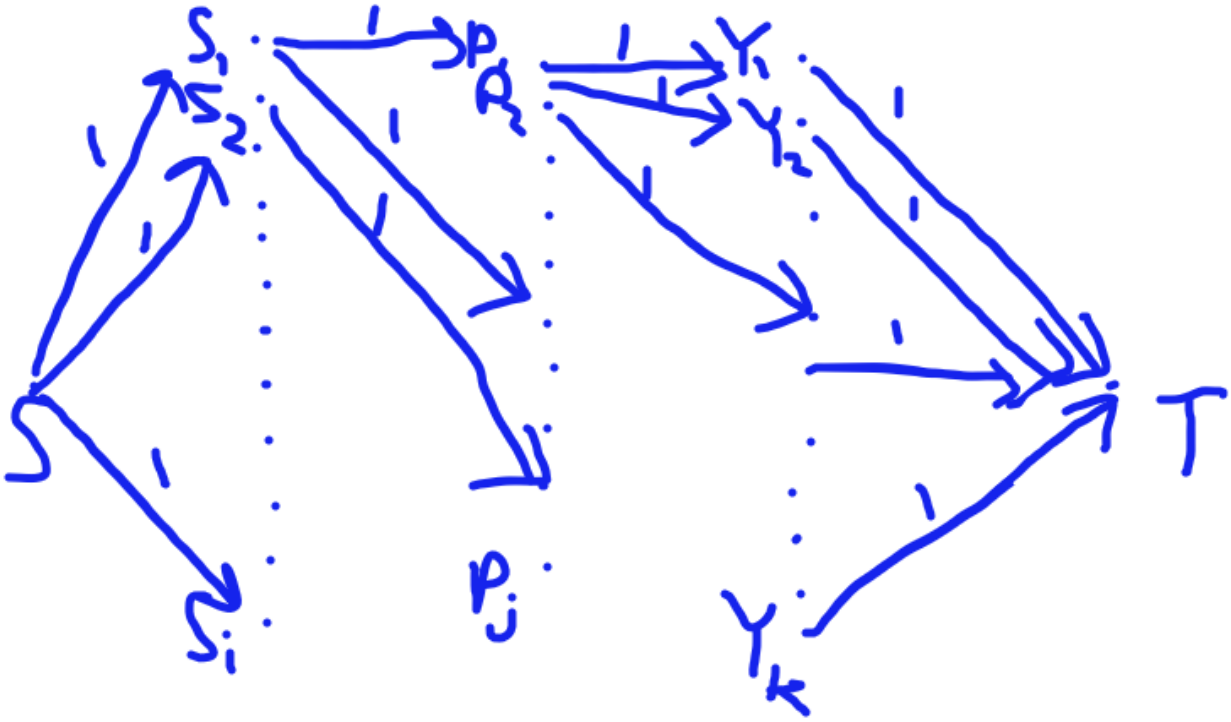
a. **Case 1**: k ≤ edges going across the min-cut
   Since each edge has a capacity of 1, deleting any of the k edges going across the min-cut would reduce the max flow by k
   Hence our new max flow = |f| - k where |f| is the original max flow

b. **Case 2:** k> edges going across the min-cut
   In this scenario we would delete all of the edges going across the min-cut as well as some other edges in the graph. Our network flow graph would get disconnected and hence
   New max flow = 0

Time Complexity: Running the Edmund Karp algorithm to find max-flow/min-cut is polynomial. Removing k edges is linear.

Hence our time complexity of this algorithm is polynomial

Ans 9



Step 1: Creating a Network Flow

Create a network flow with all instances of letter S as $s_1, s_2 \ldots s_i$, of letter P as $p_1, p_2 \ldots p_j$ and of letter Y as $y_1, y_2 \ldots y_k$ with all letters in the grid being indexed. Any letter S is connected to the letter P only if P is its neighbor (ie. if P is to the north, south, west or east of S). Similarly letter P is connected to the letter Y only if Y is its neighbor (ie. if Y is to the north, south, west or east of P). Connect all letter S to a source node and all letters Y to a target node

- Assign edge capacities 1 to all edges between the letters to represent the fact that each letter can only be part of 1 SPY
- Assign edge capacities 1 to all edges between source and letter S to represent the fact that each SPY pattern has to start with one S letter
- Assign edge capacities 1 to all edges between letters Y and target T to represent the fact that each SPY pattern has to finish with one Y letter

Step 2: Making claim

Largest number of non-overlapping SPYs formed in the grid = Max flow of the network flow

Step 3: Proving claim in both directions

In forward direction →
To prove: If largest number of non-overlapping SPYs is given, then this is equal to the max flow of the network flow

Largest number of non-overlapping SPYs would mean that each letter of the grid is used in at most 1 SPY. This is the same as the max-flow of the above network flow where each letter will only be chosen at most once in a saturated flow

In backward direction ←
To prove: Given the max flow of the network flow, prove that this is equal to the largest number of non-overlapping SPYs

For each flow that is contributing to the max flow of the network flow, it would have saturated edges. This signifies that each of the letters were chosen once and hence are non-overlapping. Hence, the max flow represents the maximum no of non-overlapping SPYs

Ans 10

a)

Step 1: Creating a Network Flow

Create a network flow with all students as $s_1, s_2 \ldots s_n$ and all classes as $c_1, c_2 \ldots c_k$. Connect a given student s_j to a class **only if it is part of the subset p_j that this student wants to sign up for**. Connect all students to a source node and all classes to a target node.

- Assign edge capacities 1 to all edges between students and classes to represent the fact that each student can be a part of any class, and each class can have any students
- Assign edge capacities m to all edges between source and students to represent the fact that each student has to register for minimum m classes
- Assign edge capacities $q_1, q_2, \ldots q_k$ to edges between classes and target to represent the fact that each class has a capacity of q_i

Step 2: Making claim

Every student can sign up for classes if and only if max flow = nXm

Step 3: Proving claim in both directions

In forward direction →
To prove: If every student has signed up for classes then max flow = nXm

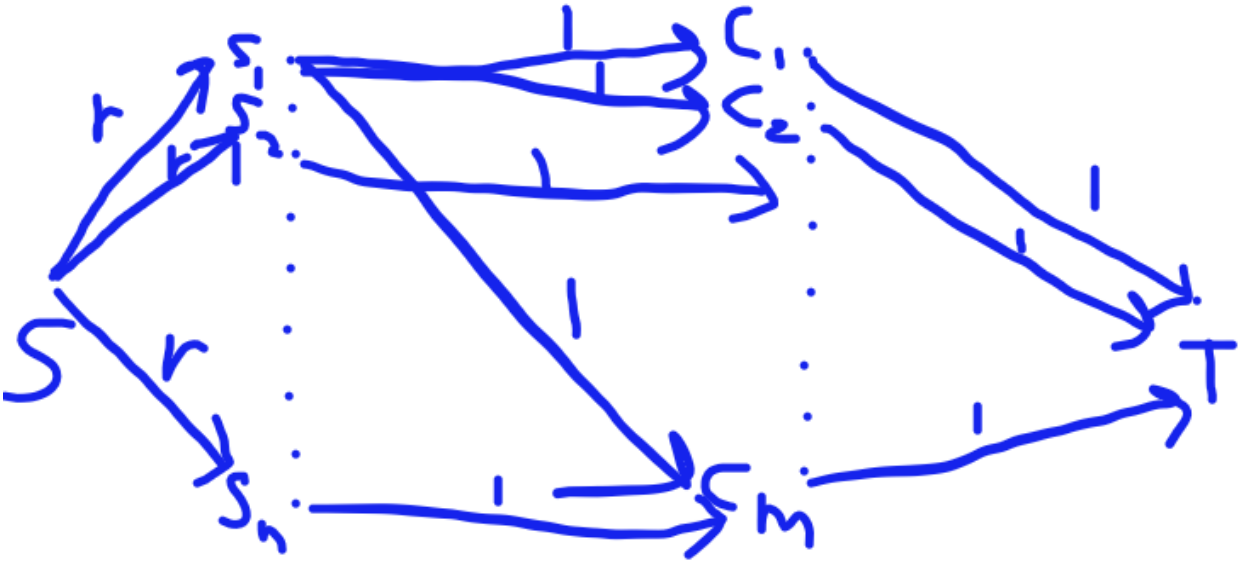
We know that if every student $s1, s2 ... s_n$ has signed up for at least m classes, then all the edges from source to students is saturated. This means that the flow cannot be bigger that $m + m + m ...$ (n times) i.e nXm

In backward direction ←

To prove: Given the max flow of the network flow, to find the actual assignment of students to classes

To find the actual assignment of students to classes, we will look at all saturated edges between students and classes

b)



Step 1: Creating a Network Flow

Create a network flow with all students as $s_1, s_2 \ldots s_n$ and all classes enrolled by them as $c_1, c_2 \ldots y_k$. Connect a given student s_j to a class **according to the solution obtained from (a) i.e. only if a student s_j has signed up for class c_k**. Connect all students to a source node and all classes to a target node.

- Assign edge capacities 1 to all edges between students and classes to represent the fact that each student can be a class representative for any of his/her enrolled classes
- Assign edge capacities r to all edges between source and students to represent the fact that each student can be a class representative for at most r classes

- Assign edge capacities 1 to all edges between classes and target to represent the fact that each class can have only 1 class representative

Step 2: Making claim

Every class will have a class representative if and only if max flow = m

Step 3: Proving claim in both directions

In forward direction →
To prove: If every class has a class representative then max flow = m

We know that if every class has a class representative, then all the edges from classes to target are saturated. This means that the flow cannot be bigger than 1+1+1… m times (for m total classes)

In backward direction ←
To prove: Given the max flow of the network flow, to find the actual assignment of student representative to classes

To find the actual assignment of which student is class representative of which class, we will look at all saturated edges between students and classes