Name: Nikhil Sathyanarayana Shastry
USC_ID: 8464475401
Email: nsathyan@usc.edu

# CSCI570 - Analysis of Algorithms (HW4)

---

**Q1.** (Dynamic Programming) Given n balloons, indexed from 0 to n−1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst the balloon i you will get nums[left] · nums[i] · nums[right] coins, where left and right are adjacent indices of i. After the bursting the balloon, the left and right then becomes adjacent. Assume, nums[−1] = nums[n] = 1 and they are not real therefore you cannot burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

**Solution:**

Subproblem: Given a range of balloons from $i$ to $j$, where $i$ is the left limit and $j$ is the right limit, our goal is to get the maximum coins by bursting the balloons within this given range. We assume that balloons outside this range have already been burst. Solving this subproblem for all the subarrays in the given $nums$ array will give us the maximum coins that we can obtain by bursting all the given balloons.

Recurrence Relation:

$OPT[i][j] = MAX(OPT\ [i][k-1] + OPT\ [k+1][j] + nums\ [i-1] * nums\ [k] * nums\ [j+1])$
where k belong to $i,\ j+1$

In this recurrence relation the maximum coins in the range from $i\ to\ j$ is given by $OPT\ [i][j]$. $OPT\ [i][k-1]$ and $OPT\ [i][k-1]$ represent the maximum coins to the left and the right of balloon k respectively. Finally, $nums[i-1] * nums[k] * nums[j+1]$ is the product of the balloon' values when you burst balloon k.

Pseudocode using Iteration:

```
function maxCoins(nums):
    n = length(nums)
    nums = [1] + nums + [1]  # handling the edge case for first balloon and the last balloon

    # A 2D array opt to store the maximum coins for subproblems and initialize the array with zeros
    opt = (n+2) x (n+2)

    for i from 1 to n:
        opt [i][i] = nums[i-1] * nums[i] * nums[i+1]
```

```
  # Start building the opt array from smaller subproblems to larger subproblems
for len from 2 to n:
    for left from 1 to n - len + 1:
      right = left + len - 1
      for k from left to right:
        opt [left][right] = max(opt [left][right],
          nums[left-1] * nums[k] * nums[right+1] + opt [left][k-1] + opt [k+1][right])

  # The final result is stored in opt [1][n]
  return opt [1][n]
```

Base Cases:

For $i == j$, there's just one balloon and there are no adjacent balloons to burst, hence the max coins you can get here is the value of the one balloon. So, $OPT\ [i][i] = nums\ [i]$ for all $i$ in the range.

For $i > j$, there are no balloons and hence $OPT\ [i][j] = 0$.

Final Answer:

After we completely compute the OPT table the final answer can be found in OPT[1][n].

Runtime Complexity:

The runtime complexity of the algorithm is $\boldsymbol{O(n^3)}$. The outer loop iterates over the balloon ranges can go up to $O(n)$. The middle loop iterates over the starting index i for each range, which can be $O(n)$. The inner loop iterates over the last balloon to burst k within each range at the worst case of $O(n)$.

**Q2.** (Dynamic Programming) Suppose you are in Casino with your friend, and you are interested in playing a game against your friend by alternating turns. The game contains a row of n coins of values $v_i$, where n is even. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money you can definitely win if you move first. Analyse the running time of your algorithm.

**Solution:**

Subproblem:

Here the subproblem for us to find is the max money that can be won in the subarray coins[i...j]. We need to consider a portion of the original coin row, that spans from the i-th coin to the j-th coin. For each subarray of coins[i...j], we need to determine the maximum amount of money that can be obtained by making optimal choices within that subarray. In each subproblem, the two cases that we consider are as follows:

1. When the user picks ($coins[i]$), the opponent picks the from i+1th or jth coin.

2. When the user picks the last coin ($coins[j]$). the opponent picks the from ith or j-1th coin.

Recurrence Relation:

$OPT[i][j] = max(coins[i] + \min(OPT[i+2][j], OPT[i+1][j-1]), coins[j] + \min(OPT[i+1][j-1], OPT[i][j-2]))$,

The above recurrence relation states that the maximum amount that can be obtained in a subarray $OPT[i][j]$ is given by the maximum between the user choosing the first coin ($coins[i]$) plus the minimum of optimal move by the opponent. When the user chooses ith coin, the opponent can choose from i+1th or jth coin(min ($OPT[i+2][j], OPT[i+1][j-1]$)). When the user chooses the last coin ($coins[j]$), $OPT[i][j]$ is given by adding $coins[j]$ with the minimum in the situation where the opponent picks the ith coin or j-1th coin. (min ($OPT[i+1][j-1], OPT[i][j-2]$)))

Pseudocode:

```
function maxCoins(coins):
    n = length of the coins array
    opt = 2D array of size n x n
    for len in range(n):
        for j in range(len, n):
            i = j - len
            x = 0, y = 0, z = 0
            if (i + 2) <= j:
                x = opt[i + 2][j]
            if (i + 1) <= (j - 1):
                y = opt[i + 1][j - 1]
            if i <= (j - 2):
                z = opt[i][j - 2]
            opt[i][j] = max(coins[i] + min(x, y), coins[j] + min(y, z))
    return opt[0][n - 1] # The final result is stored in opt [0][n-1]
```

Base Cases:

For $i == j$, it means that our subproblem has subarray only of size 1. This means that single coins = value of that coin Hence, for all i, $opt[i][i] = coins[i]$.

Final Answer:

After we completely compute the OPT table the final answer can be found in OPT[0][n-1].

Runtime:

The runtime complexity of this dynamic programming algorithm is $\boldsymbol{O(n^2)}$**.** Here we iterate through the array once for each subarray size, and later for each subarray size, we iterate through all possible starting indices.

**Q3.** (Dynamic Programming) Jack has gotten himself involved in a very dangerous game called the octopus game where he needs to pass a bridge which has some unreliable sections. The bridge consists of 3n tiles as shown below. Some tiles are strong and can withstand Jack's weight, but some tiles are weak and will break if Jack lands on them. Jack has no clue which tiles are strong or weak but we have been given that information in an array called BadTile(3,n) where BadTile(j, i) = 1 if the tile is weak and 0 if the tile is strong. At any step Jack can move either to the tile right in front of him (i.e. from tile (j, i) to (j, i + 1)), or diagonally to the left or right (if they exist). (No sideways or backward moves are allowed and one cannot go from tile (1, i) to (3, i + 1) or from (3, i) to (1, i + 1)). Using dynamic programming find out how many ways (if any) there are for Jack to pass this deadly bridge. Analyze the running time of your algorithm.

Figure below shows bad tiles in gray and one of the possible ways for Jack to safely cross the bridge alive (See Fig. 1).

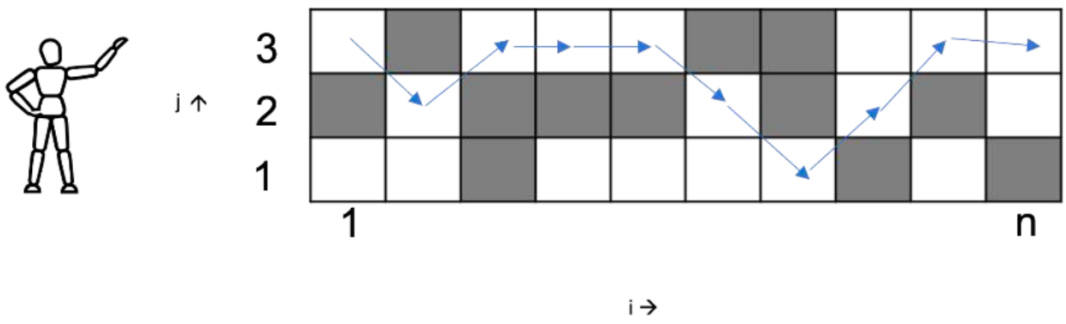



Figure 1

**Solution:**

Subproblem:

Let OPT[j, i] be the number of ways jack can reach the tile(j,i) safely from the start of the bridge where $1 \leq j \leq 3$, $1 \leq i \leq n$

Recurrence Relation:

$$OPT[j,i] = \begin{cases} OPT[j,i-1] + OPT[j+1,i-1] & BadTile(j,i) = 0, \quad j = 1 \\ OPT[j,i-1] + OPT[j-1,i-1] + OPT[j+1,i-1] & BadTile(j,i) = 0, \quad j = 2 \\ OPT[j,i-1] + OPT[j-1,i-1] & BadTile(j,i) = 0, \quad j = 3 \\ 0 & BadTile(j,i) = 1 \end{cases}$$

Pseudocode: In the pseudocode below, the array has its index starting from 0 and hence the values of j will range from 0 to 2 and the I value will range from 0 to n-1. Hence the base cases are written accordingly.

```
Def ways(badtile,n):
    OPT = [[0 for x in range(n)] for x in range(3)] #initialize 3,n 2d array
    For j in range(3): #Base case
        OPT[j][0] = 1 if badtile[j][0] ==0 else 0
    For I in range(1, n): # looping from 1 to n-1
        For j in range(3): # looping from 0 to 2
            If badtile[j][i] == 1:
                OPT[j][i] = 0
            Elif j == 0:
                OPT[j][i] = OPT[j][i-1] + OPT[j+1][i-1]
            Elif j == 1:
                OPT[j][i] = OPT[j][i-1] + OPT[j-1][i-1] + OPT[j+1][i-1]
            Elif j == 2:
                OPT[j][i] = OPT[j][i-1] + OPT[j-1][i-1]
    Return  OPT[0][n-1] + OPT[1][n-1] + OPT[2][n-1]
```

Basecases:

For j = 0, 1, 2

1. If BadTile(j,0) = 0, then OPT[j,0] = 1
2. If BadTile(j,0) = 1, then OPT[j,0] = 0

Final Answer:

After we completely compute the OPT table the summation of all the tiles of column n-1 will be the final answer i.e OPT[0][n-1] + OPT[1][n-1] + OPT[2][n-1].

Time Complexity:

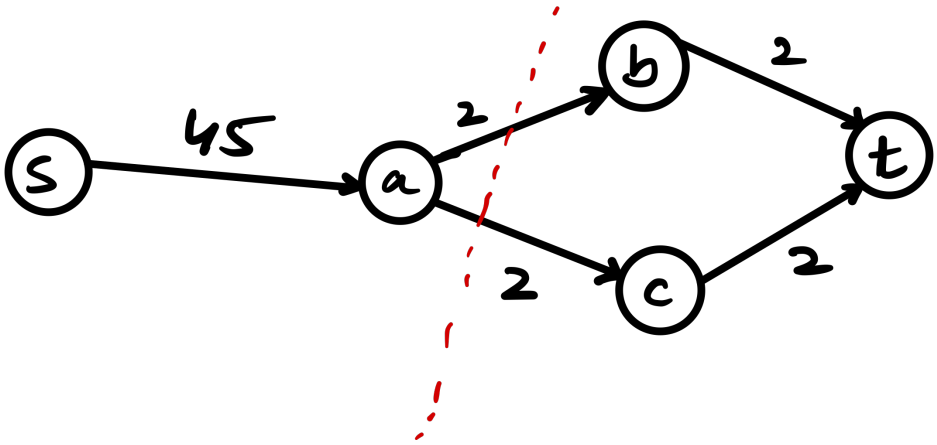
The time complexity for this algorithm will be the time taken to populate the OPT table which is going to be O(3n). Further, since n is the size of the input, the above algorithm is linear to the input size. Therefore, the overall time complexity of the above algorithm is $\boldsymbol{O(n)}$.

**Q4.** Given a flow network with the source s and the sink t, and positive integer edge capacities c. Prove or disprove the following 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.

**Solution:**

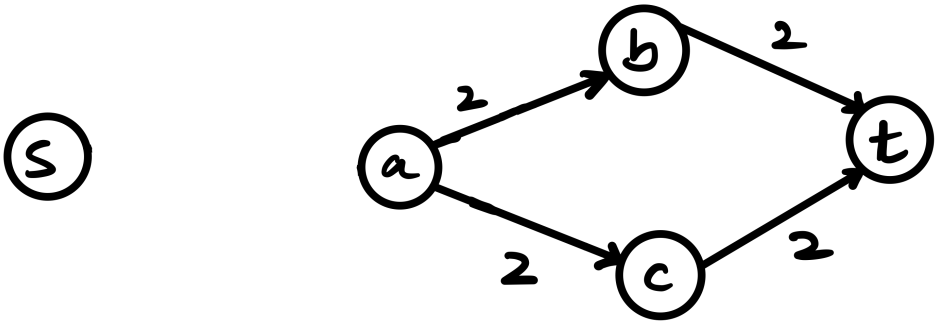
The statement given above is **False** and this can be proved by the following counter example.

**Proof by Counterexample:**




In the example graph given above, min cut is represented by the red dotted line.

In this graph let us consider the edge e to be 45 (Edge from S -> A). If we delete this edge from our graph we get a graph as shown below.




Here we can see that in such an instance, on deleting an edge that is not a part of the min-cut can make the graph disconnected and hence the overall flow in the graph become zero (more impact on the graph than deleting an edge from a min-cut could have done), Hence this proves that the above given statement – "if deleting an 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" as **FALSE**.

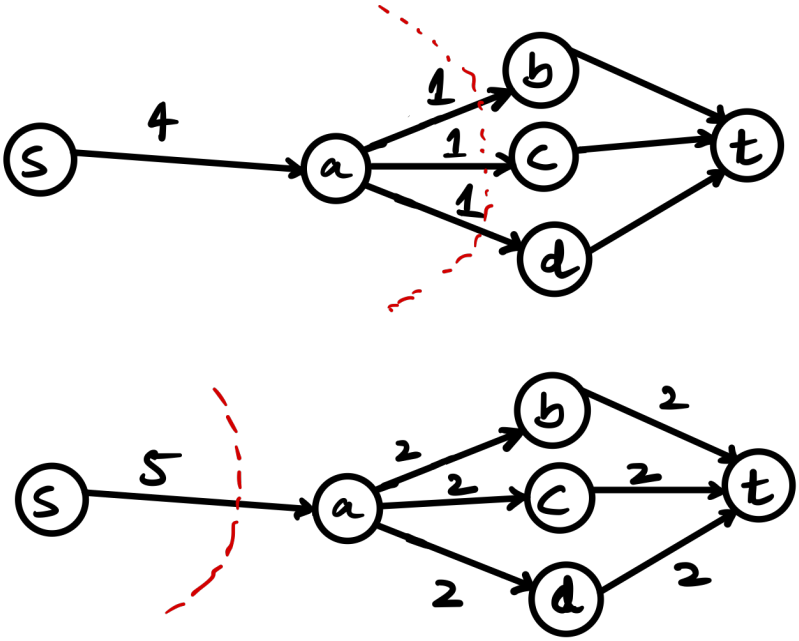
**Q5.** Given a flow network with the source s and the sink t, and positive integer edge capacities c. Let s − t be a minimum cut. Prove or disprove the following statement: If we increase the capacity of every edge by 1, then s − t still be a minimum cut.

**Solution:**

The given statement cannot be true in all cases, hence **FALSE:**

**Proof by Counterexample:**

Consider the graph below as an example:




In these graphs we can see that initially the min-cut when the S->a was 4 is not the same when we incremented the capacity of every edge by 1. Hence this counterexample disproves the statement that “if we increase the capacity of every edge by 1, then s – t still be a minimum cut”.

**Q6.** (Network Flow)

(a) For the given graph $G_1$ (see Fig. 2), find the value of the max flow. Edge capacities are mentioned on the edges. (You don't have to show each and every step of your algorithm).

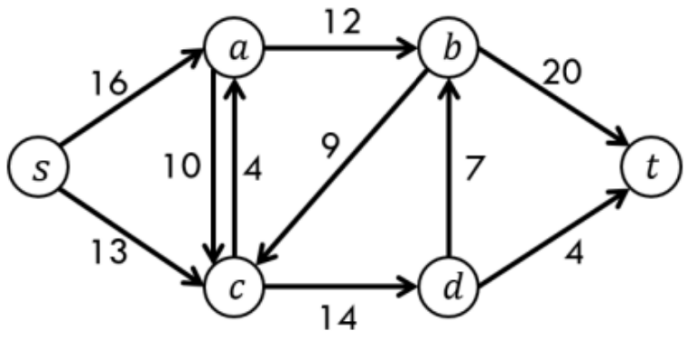



Figure 2: G1

(b) For the given graph, find the value of the min-cut. (You don't have to show each and every step of your algorithm).

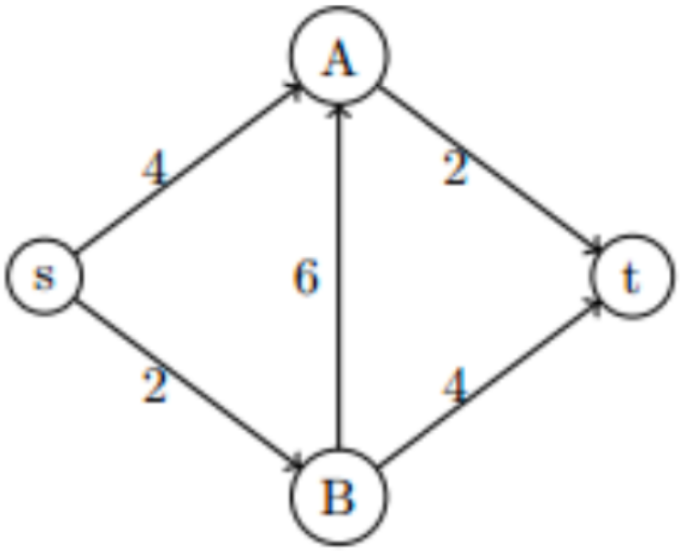
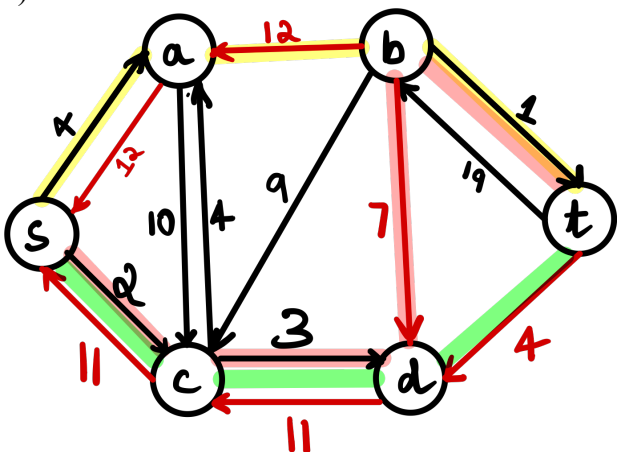



Figure 3

**Solution:**

In the solutions below, the backward edge is marked in red colour and if two nodes have only a backward edge and there is no forward edge present then it means that that particular forward edge between those 2 nodes has been saturated.

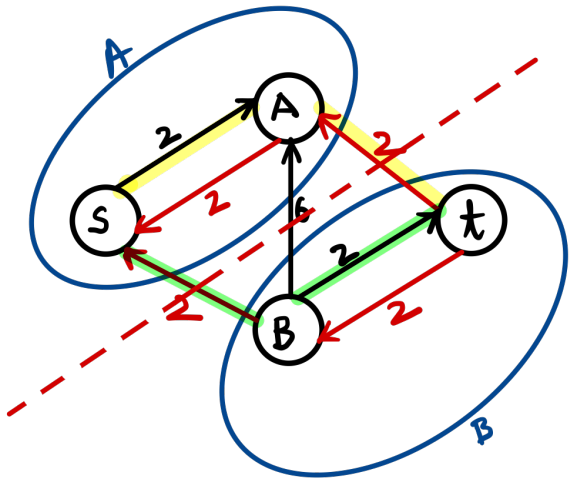
a)




| FLOW PATH CHOSEN | Residual Flow |
|---|---|
| s→a→b→t | 12 |
| s→c→d→b→t | 7 |
| s→c→d→t | 4 |
| Max Flow | 23 |

The above residual graph of G1 represents the paths followed to find the maximum flow in the given graph G1. The table next to the graph lists the specific paths chosen and the residual flows calculated for those paths. Finally, we get the maximum flow of the graph by summing up the residual flows of each of these paths. The ***Max Flow*** for the given graph G1 is **23**.

b)



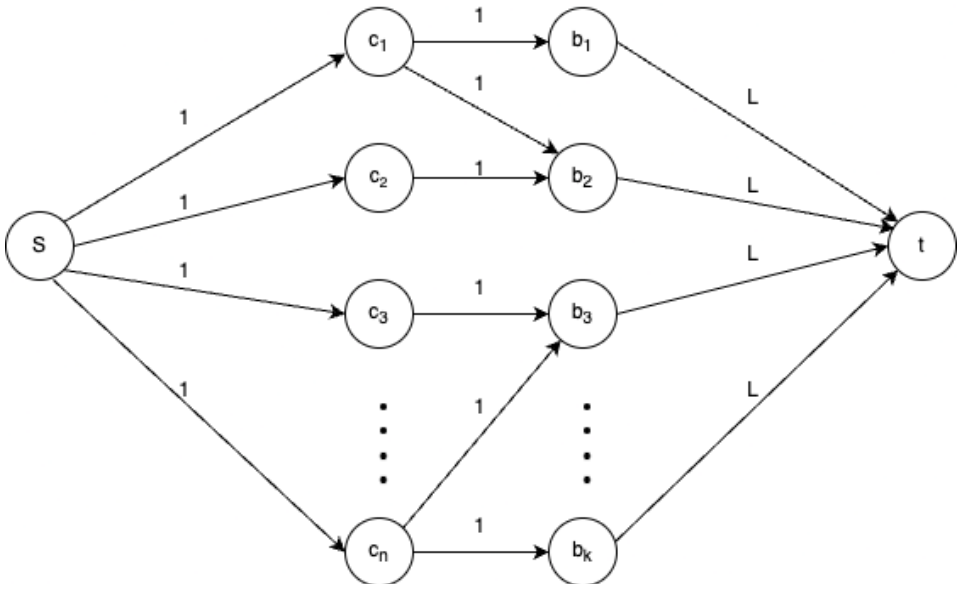

| FLOW PATH CHOSEN | Residual Flow |
|---|---|
| s→A→t | 2 |
| s→B→t | 2 |
| Max Flow | 4 |

The above residual graph given in Figure 3 represents the paths followed to find the Min Cut in the given graph. The table next to the graph lists the specific paths chosen and the residual flows calculated for those paths. Finally, we get the Min Cut of the graph by partitioning the

modified graph into two subsets such that s ∈ A and t ∈ B. And the ***Min Cut*** for the given graph is **4**.

**Q7.** (Network Flow) Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible base stations. We'll suppose there are n clients, $c_1, c_2, \ldots, c_n$, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations, $b_1, b_2, \ldots, b_k$; the position of each of these is specified by (x, y) coordinates as well. For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter R which means that a client can only be connected to a base station that is within distance R. There is also a load parameter L which means that no more than L clients can be connected to any single base station. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station. Prove the correctness of the algorithm.

**Solution:**




1. Find the distance between each client and base station and if the distance between them is than or equal to R we need to store them in a dictionary.
2. Create a network flow with client vertices: $c_1, c_2, c_3, \ldots, c_n$ and base station vertices: $b_1, b_2, b_3, \ldots, b_k$ where $n$ and $k$ are the number of clients and base stations given in the problem statement.
3. Next we need to create a source vertex (s) before the client vertices and then a sink/target (t) vertex after the base station vertices.
4. The connections are as follows:
   - Vertex S is connected to the client vertices with edge capacity of 1.
   - The base station vertices are connected to the sink/target (t) vertex with edge capacity of L.

- $c_i$ vertex is connected to $b_j$ vertices only if the distance between them is less than or equal to R which is stored in the dictionary. The edge capacity for connected edges is 1.

5. After the creation of the network flow we need to run Ford Fulkerson algorithm on it.

**Claim**: All the clients can be connected to the base stations *if and only if* there exists a flow (s-t) with a value of n which is the total number of clients. This means that every client is connected to one base station. Here the maxflow = n.

**Proof of correctness-**

Proof:
=> If every client is connected to a base station, then max flow = n

- If every client can be connected to a base station simultaneously, then all the $s{-}ci$ edges are saturated. Hence the flow cannot be greater than $n$ and if all clients can be connected then our maxflow = n (because there are n clients).

Conversely:
<= Given the max flow = n, then every client is connected to a base station

- If maxflow = n, there are exactly a total of $n$ edges between $ci - bj$ that are saturated. We make use of all the saturated edges to make a combination where all clients can connect to base stations simultaneously.

**Q8.** (Network Flow) You are given a flow network with unit-capacity edges: it consists of a directed graph G = (V, E) with source s and sink t, and $u_e = 1$ for every edge e. You are also given a positive integer parameter k. The goal is delete k edges so as to reduce the maximum s − t flow in G by as much as possible. Give a polynomial-time algorithm to solve this problem. In other words, you should find a set of edges F ⊆ E so that |F|=k and the maximum s−t flow in the graph $G' =(V,E-F)$ is as small as possible. Give a polynomial-time algorithm to solve this problem.

**Solution:**

Let us consider $f$ to be the max-flow of graph G. Similarly, we can also find the min-cut C (A, B) (by separating s and t into two partitions) for the network flow G.

Let us now consider the set of edges E' that goes across the cut (A, B), that is, all the forward edges that start from A and end in B.

Now, we must delete $k$ edges with each edge having a capacity of 1. Removing $k$ edges would never result in a flow with a smaller capacity than $f - k$ (remove k edges with capacity 1, the total capacity reduced becomes k).

There are 2 cases that can occur-

1. Number of edges E' that cross the cut(A,B) is greater than k, here we remove those edges from the cut, and then the value of max flow reduces by $k$. (it becomes $f - k$)

2. Number of edges E' is equal to or less than value k. This means that by removing all the edges, our network flow of the given graph would be disconnected, which means that the flow value reduces to 0.

From the above two conditions, we can see that the min-cut in the new graph has the value of G' is max (0, f-k), and the max value cannot be reduced any further, which is what we wanted.

**Time Complexity:** We use Edmund Karp algorithm to find the min-cut/max-flow of G value in polynomial time O(V*E) this is because we have all unit capacities. Removing at most k edges and reducing the network flow capacity (creating G') is done in linear time O(k). Hence the total time complexity will be $O(V * E)$ which is polynomial.

**Q9.** (Network Flow) Counter Espionage Academy instructors have designed the following problem to see how well trainees can detect SPY's in an n × n grid of letters S, P , and Y . Trainees are instructed to detect as many disjoint copies of the word SPY as possible in the given grid. To form the word SPY in the grid they can start at any S, move to a neighbouring P , then move to a neighbouring Y . (They can move north, east, south or west to get to a neighbour.) The following figure shows one such problem on the left, along with two possible optimal solutions with three SPY's each on the right (See Fig. 4). Give an efficient network flow-based algorithm to find the largest number of SPY's.

Note: We are only looking for the largest number of SPY's, not the actual location of the words. No proof is necessary.

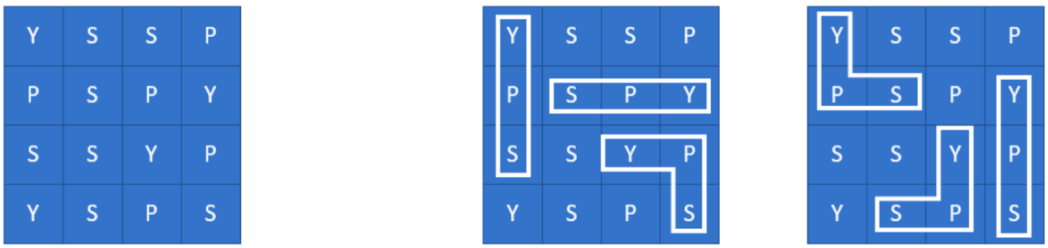
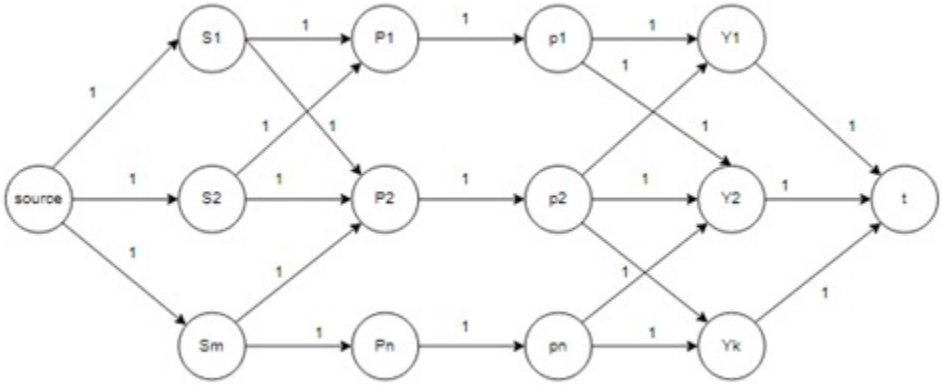



Figure 4

**Solution:**




1. First we need to traverse through the matrix and create a network flow:
   a. Create a source node s (source) and target node/sink node t.
   b. Each time you reach the letter S in the matrix, add a node for $S_{row, col}$ and connect the source node s to the node $S_{row, col}$ with an edge capacity of 1.
   c. Further, as, and when we encounter an S, we need to go to all four directions (if possible), and if we encounter a letter P, we need to create a node $P_{row, col}$. Then, draw an edge from S to its corresponding P, with an edge capacity of 1.
   d. We then need to put another matching set of $p_{row, col}$ to get distinct occurrences of the word SPY. The first set of $P_{row, col}$ and second set of $p_{row, col}$ would have an edge capacity 1. We use this second layer of p nodes to ensure that for every S node we use only one distinct P node from the input matrix. This ensures that no duplicates are being introduced to the output.
   e. Then, from the second set of node $p_{row, col}$ we again traverse to all four directions to see if we get a Y. If yes, then we create a node $Y_{row, col}$ and draw an edge from P to its corresponding Y, with edge capacity as 1.
   f. Finally, all $Y_{row. col}$ nodes are connected to the sink node t, with edge capacity 1. We do this to ensure that the Y is selected by only one "SPY" and the usage of Y nodes for each "SPY" is unique.

2. Run Ford Fulkerson or Edmund Karp's algorithm to the above network to find the max flow value.

3. The max flow min cut value we find will be the largest number of distinct (non-overlapping) occurrences of SPY present in the matrix.

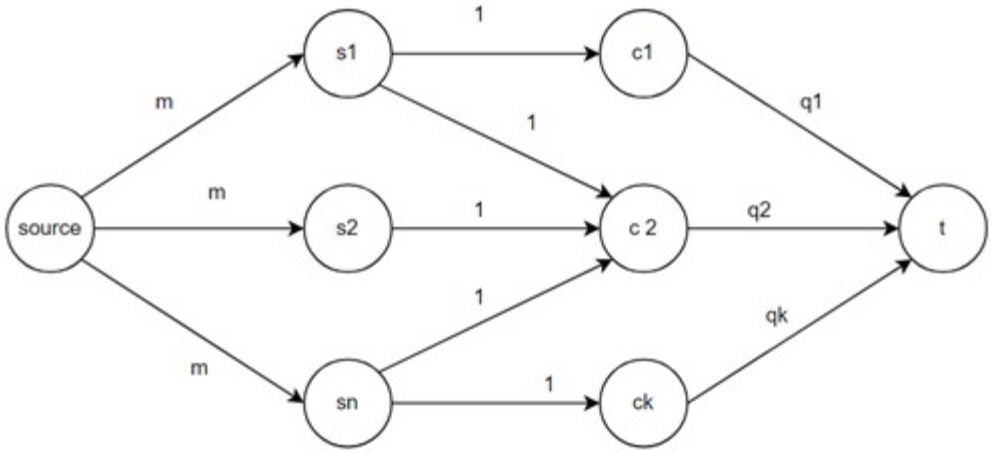
**Q10.** (Network Flow) USC students return to in person classes after a yearlong interval. There are k in-person classes happening this semester, $c_1$, $c_2$, ..., $c_k$. Also there are n students, $s_1$, $s_2$, ..., $s_n$ attending these k classes. A student can be enrolled in more than one in-person class and each in-person class consists of several students.

a) Each student $s_j$ wants to sign up for a subset $p_j$ of the k classes. Also, a student needs to sign up for at least m classes to be considered as a full time student. (Given: $p_j \geq m$) Each class $c_i$ has capacity for at most $q_i$ students. We as school administration want to find out if this is possible. Design an algorithm to determine whether or not all students can be enrolled as full time students. Prove the correctness of the algorithm.

b) If there exists a feasible solution to part (a) and all students register in exactly m classes, the student body needs a student representative from each class. But a given student cannot be a class representative for more than r (where r < m) classes which s/he is enrolled in. Design an algorithm to determine whether or not such a selection exists. Prove the correctness of the algorithm. (Hint: Use part (a) solution as starting point).

**Solution:**

**a)**




To solve the problem first we need to construct a network flow graph and then we need to run the Ford Fulkerson algorithm to get the max flow. If we get the max flow to be equal to nm, then all students can be enrolled as full time students. The construction of the graph is as follows:

- **a.** Arrange the $n$ students layer and $k$ classes layer similar to how vertices would be represented in a bipartite graph. Connect each student $s_j$ with all the classes in $p_j$ using edges with capacity of 1.
- **b.** Next we need to create a source vertex (s) before the students vertices and then a sink/target (t) vertex after the classes vertices.

**c.** Then, we need to connect all the $s_j$ student vertices to the source vertex with capacity of m.

**d.** Finally, connect all the $c_i$ class vertices to the sink vertex with capacity of $q_i$.

**Claim**: The enrolment of a student is feasible if and only if there exists a max flow of $nm$.

**Proof of correctness-**

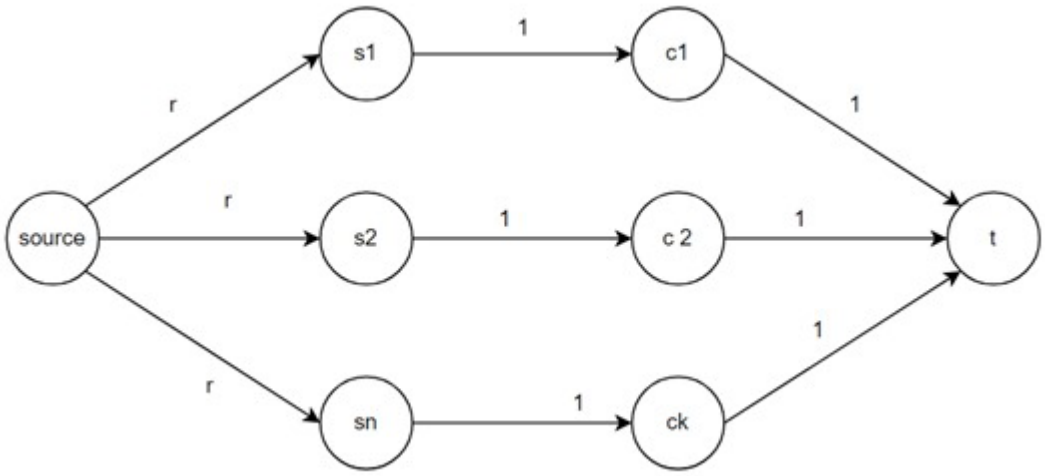
=> The max flow is nm if the problem has a feasible solution.

- Proof:
  - Suppose a viable solution exists for the problem. In that case, it signifies that all in-person classes have a maximum of qi students, and each student has successfully registered for at least m classes. Following the conservation constraint, the outward flow at the student vertex must equal the incoming flow. Consequently, all edges from the source to the student vertices will be saturated with a value of m, as each student has enrolled in their classes. Likewise, the incoming flow at the student vertex will be evenly distributed among the outgoing edges with a value of 1. The total flow from student vertices to the class vertex will amount to nm as an effect of the conservation constraint. This flow will ultimately reach the sink vertices with edges of maximum capacity qi. Hence, if the problem has a feasible solution, the maxflow will be $nm$.

<= The problem has a feasible solution if the max flow is nm.

- Proof:
  - If the problem has a maximum flow of nm, it implies that on any path from the source to the sink vertex, there exists at least one student who has successfully registered for a specific class. Given that the flow is at its maximum, it guarantees that each student vertex receives a flow of at least m, and the outward flow at each student vertex is precisely m. Consequently, each student can register for a minimum of m classes, meeting the problem's requirements. Furthermore, from our NF construction, we know that each class can have a maximum of qi students. This satisfies the other constraint in the problem statement because the flow along the edge will not exceed qi (it is always lesser than or equal to the capacity). Hence, the problem has a viable solution if the maximum flow equals $nm$.

**b)**




To solve this part of the problem we can make use of the above solution **a.** as the starting point. We need to slightly modify the above constructed graph and then we need to run the Ford Fulkerson algorithm to get the max flow. If the max flow is equal to the number of classes k, then a selection exists otherwise no valid selection exists. The construction of the graph is as follows:

- **a.** Make use of the same nodes used for the construction of solution **a.**
- **b.** We need to remove a few edges between students and classes vertices where the student did not get an opportunity to sign up for.
- **c.** The capacity on the edges from student to class edges will be equal to 1.
- **d.** The capacity of the edges from classes to sink vertex must be set to 1. This is because there can be only one student representative for each class
- **e.** The capacity of the edges between source and student vertices must be set to r.

**Claim**: The selection is feasible if and only if there exists a max flow of k.

**Proof of correctness-**

=> The max flow is k if the problem has a feasible selection.

- Proof:
  - If there is a feasible selection for the problem, it implies that all in-person classes have received valid selections, and each student has been selected for a maximum of r choices. Abiding by the conservation constraint, the outward flow at the class vertex must match the incoming flow. Consequently, all edges from the sink to the class vertices will be fully utilized with a value of 1, as each class has received exactly one selection. This ensures that k represents the inward flow at class vertices following conservation constraint. Likewise, the number of selections per student will reach the maximum value of r classes. The total flow from the source vertex to the student vertices will amount to k, which again follows the conservation constraint. Therefore, if the problem offers a feasible solution, the maximum flow will equal k.

<= The problem has a feasible selection if the max flow is k.

- Proof:
  - If the problem attains a maximum flow of k, it signifies that along any path from the source to the sink vertex, a student is chosen from a class. Because the flow is at its maximum, all class vertices receive a flow of 1, and the outward flow at the class vertices is precisely 1 (thus achieving a total flow of k). Consequently, only one student is selected for each class, meeting the problem's constraint of one selection per class. Therefore, each class can only have a single student representative selected. Additionally, based on the structure of our NF, each student can be selected for a maximum of r classes, adhering to another constraint since the flow along an edge will be at most r, which is less than or equal to the capacity of that edge. Thus, the problem has a viable selection if the maximum flow is k.