# CSCI 570 – ANALYSIS OF ALGORITHMS HOMEWORK – 4

1. (**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 $nums[left] \cdot nums[i] \cdot nums[right]$ 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 can not 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.

**ANS:**

**Given:**

1. Number of balloons = $n$
2. $nums$ is the array of $'n'$ balloons indexed from $'0'$ to $'n-1'$.
3. Each balloon is marked with a number on it.

**Constraints:**

- $nums[-1] = nums[n] = 1$
- If $i^{th}$ balloon is burst, we get $nums[left] * nums[i] * nums[right]$ coins.
- If $i^{th}$ balloon is burst, $(i-1)^{th}$ balloon and $(i+1)^{th}$ balloon become the adjacent balloons.

Here, $(i-1)$ is considered as $'left'$ and $(i+1)$ is considered as $'right'$ for the index $'i'$.

**Aim:** To find maximum number of coins we can collect by bursting the balloons wisely.

Based on the above constraints, we create a new array $numsNew$ of size $'n+2'$.

We store the 'n' elements from the array $nums$, along the indexes '1' to 'n' in $numsNew$.

And $numsNew[0] = numsNew[n+1] = 1$.

**Subproblem Definition:**

Let $OPT[i,j]$ be the maximum number of coins one can collect after bursting the balloons from index $'i'$ to index $'j'$, inclusive of balloons at $'i'$ and '$j$'. The goal is to calculate $OPT(1,n)$ to find the maximum coins that can be collected by bursting all the balloons.

**Recurrence Relation:**

Let $'k'$ be the last element that is burst from the list of balloons between index $'i'$and index $'j'$, and $i \leq k \leq j$.

$'k'$ can be any index between $'i'$ and $'j'$, inclusive of $'i'$ and $'j'$.

$$=> OPT[i,j]_{1\leq i,j\leq n} = \max_{i\leq k\leq j}( OPT[i,k-1] + OPT[k+1,j] + numsNew[i-1] * numsNew[k] * numsNew[j+1] )$$

**Base Cases:**

- Create a matrix $OPT[n+2][n+2]$ and initialize all the values to $'0'$.

Here elements of the matrix are filled diagonally. We will get an upper diagonal matrix (all elements below the diagonal elements are 0 covered in the initialization), where $OPT[1,n]$ gives the maximum number of coins one can collect after bursting all balloons wisely.

**Algorithm:**

1. Create a new array $numsNew[n+2]$.
2. Initialize $numsNew[0] = 1$ and $numsNew[n+1] = 1$.
3. Iterate the variable $'i'$ from $'1'$ to $'n'$ inclusive of $'1'$ and $'n'$ with an increment of $'1'$. For each iteration of $'i'$ perform Step 4.
4. $numsNew[i] = nums[i-1]$
5. Create $OPT[n+2][n+2]$ and initialize all values to $'0'$.
6. Iterate the variable $'length'$ from $'1'$ to $'n'$inclusive of $'1'$ and $'n'$ with an increment of $'1'$. For each iteration of $'length'$ go to Step 7.
7. Iterate the variable $'left'$ from $'1'$ to $'n - length + 1'$ inclusive of both values with an increment of $'1'$. For each iteration of $'left'$ perform Step 8 to Step 9.
8. $right = left + length - 1$ .
9. Iterate the variable $'last'$, from $'left'$ to $'right'$ inclusive of both values with an increment of $'1'$. For each iteration of $'last'$ perform Step10.

10. $OPT[left, right] = MAX(\ OPT[left, right],\ OPT[left, last-1] + OPT[last+1, right]\ +\ numsNew[left-1] * numsNew[last] * numsNew[right+1]\ )$

Maximum number of coins one can collect by bursting the balloons wisely will be present at $OPT[1, n]$.

**Pseudocode:**

```
def maxCoins(nums):
    n = len(nums)
    numsNew = [0] * (n + 2)
    numsNew[0] = 1
    numsNew[n + 1] = 1
    for i in range(1, n + 1):
        numsNew[i] = nums[i - 1]

    OPT = [[0] * (n + 2) for _ in range(n + 2)]

    for length in range(1, n + 1):
        for left in range(1, n - length + 2):
            right = left + length - 1
            for last in range(left, right + 1):
                OPT[left][right] = max(OPT[left][right], OPT[left][last -
 1] + OPT[last + 1][right] + numsNew[left - 1] * numsNew[last] * numsNew[right + 1])

    return OPT[1][n]
```

**Run-Time Complexity:**

Here the elements are filled diagonally. Finally, we get an upper diagonal matrix.

According to the algorithm, we iterate over $\frac{(n+2)^2}{2}$ elements to fill the matrix. Time complexity to iterate $\approx O(n^2)$.

In order to fill each value of $OPT[i, j]$, $i \leq last \leq j$. In the worst case it is of order $'n'$.

Time complexity to fill each value = $O(n)$.

Therefore, overall time complexity to fill $n^2$ elements $\approx O(n^3)$

2. (**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. Analyze the running time of your algorithm.

**ANS:**

**Constraints:**

- Number of coins = $n$
- $'n'$ is an even number.
- Player has to select either first or last coin from the row.
- After selecting the coin, it has to be removed permanently from the row.
- Each player gets an alternating turn.

**Aim:** To find maximum possible amount of money the player, who takes the first move, can definitely win.

=> Let the player who takes the first move be $Player_1(we)$ and the player who takes the next move be $Player_2$.

Let $OPT[i,j]$ be the maximum amount of money $Player_1$ can definitely win, when the subarray of coins from indexes $'i'$ to $'j'$.

**Subproblem Definition:**

Define $OPT(i,j)$ as the maximum possible amount of money you can win if you are playing with the coins from the i$^{th}$ coin to the j$^{th}$ coin.

The goal is to calculate $OPT(0,n-1)$ for the entire row of coins.

**Recurrence Relation:**

The amount $Player_1$ can definitely win is achieved when both $Player_1$ and $Player_2$ perform their best at each step.

$Player_1$ can choose either $i^{th}$ coin or $j^{th}$ coin. We assume $Player_2$ also plays his best.

If $Player_1$ chooses the $i^{th}$ coin, then $Player_2$ can choose either $(i+1)^{th}$ coin or $j^{th}$ coin. In that case $Player_1$ should consider the minimum of $Player_2$'s choices.

Similarly, if $Player_1$ chooses the $j^{th}$ coin, then $Player_2$ can choose either $i^{th}$ coin or $(j-1)^{th}$ coin. In that case $Player_1$ should consider the minimum of $Player_2$'s choices.

$$OPT[i,j] = \max\left( v_i + \min(OPT[i+2,j], OPT[i+1,j-1]), v_j + \min(OPT[i,j-2]), OPT[i+1,j-1]\right)$$

**Base Cases:**

When $i > j$, where $OPT(i,j) = 0$ because there are no coins to pick in this scenario (this is covered in the initialization phase).

1. When there is only a single coin, $Player_1$ can choose only that coin and the maximum possible amount is the value of that coin.

$=> OPT[i,i] = v_i$

2. When there are 2 coins, the maximum possible amount the $Player_1$ can definitely win is maximum value between those 2 coins.

$=> OPT[i,i+1] = \max(v_i, v_{i+1})$ {Provided $(i+1)$ is not out of the range index}

**Algorithm:**

1. Let $'V'$ be the values array of $'n'$ coins.
2. Initialize the $OPT$ array of size $(n,n)$ with $'0'$.
3. Iterate the variable $'i'$ from $'0'$ to $'n-1'$ and Update diagonals of array $'OPT'$ as per step 4.
4. $OPT[i,i] = V_i$
5. Iterate the variable $'i'$ from $'0'$ to $'n-2'$ and Update diagonals of array $'OPT'$ as per step 6.
6. $OPT[i.i+1] = \max(V_i, V_{i+1})$.
7. Iterate the variable $'length'$ from $'3'$ to $'n'$. For each iteration go to Step 8.
8. Iterate the variable $'i'$ from $'0'$ to $'n - length'$. For each iteration perform actions from Step 9 to Step 10.
9. $j = i + length - 1$

10. $OPT[i,j] = \max\ (\ v_i + \min(OPT[i+2,j], OPT[i+1,j-1]),\quad v_j + \min\ (OPT[i+1,j-1], OPT[i,j-2])\ )$
11. After all the iterations are done, the required answer is found at $OPT[0,n-1]$.

**Pseudocode:**

```
def maxMoneyCoins(V):
    n = len(V)
    OPT = [[0] * n for _ in range(n)]

    # Step 3: Initialize diagonals of OPT array
    for i in range(n):
        OPT[i][i] = V[i]

    # Step 4: Update diagonals of OPT array
    for i in range(n - 1):
        OPT[i][i + 1] = max(V[i], V[i + 1])

    # Step 5: Iterate over the variable 'length'
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1

            # Step 10: Calculate OPT[i][j]
            OPT[i][j] = max(
                V[i] + min(OPT[i + 2][j], OPT[i + 1][j - 1]),
                V[j] + min(OPT[i + 1][j - 1], OPT[i][j - 2])
            )

    # Step 11: Required answer is found at OPT[0][n-1]
    return OPT[0][n - 1]
```

**Run-Time Complexity:**

1. We are filling the upper diagonal matrix of size $(n,n)$. We iterate over $\frac{n^2}{2}$ elements. Based on the recurrence relation, the $'max'$ and $'min'$ values are already updated. We assume that to time complexity to lookup in the table is $O(1)$.
2. The time complexity to fill the entries of the table = $O(n^2)$.

Therefore, run time complexity of the algorithm is $O(n^2)$.

3. (**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).

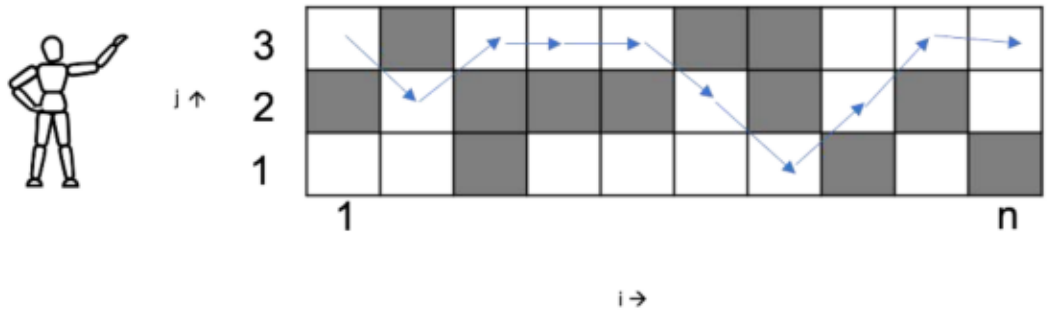



Figure 1

**ANS:**

**Given:**

- An array $'BadTile(3, n)'$ of size $'3'$ rows and $'n'$ columns.
- If $BadTile(j, i) = 0$, then the tile is strong. If $BadTile(j, i) = 1$, then the tile is weak.

**Aim:** To find the number of ways to pass the deadly bridge.

**Constraints:**

From tile $(j, i)$, we can move forward, diagonally right or left if path exists.

The above array can be represented as below.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

**Subproblem Definition:**

Let $OPT(j,i)$ as the number of ways to reach tile $(j,i)$ while avoiding weak tiles.

**Recurrence Relations:**

Let $OPT[j,i]$ be the number of ways to reach the tile $(j,i)$. We have the following **recurrence relations**,

$$OPT[3,i] = OPT[3,i-1] + OPT[2,i-1]$$
$$OPT[2,i] = OPT[3,i-1] + OPT[2,i-1] + OPT[1,i-1]$$
$$OPT[1,i] = OPT[2,i-1] + OPT[1,i-1]$$

**Base Cases:**

- When $i = 0$ (the first column), where ways $(j,0) = 1$ for the starting position. This is because there's only one way to start at the first column.

**Algorithm:**

1. Iterate over the $BadTile(3,n)$ array and change the values of $'1'$ to $'0'$ and $'0'$ to $'1'$.
2. The above array will change as below

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

3. According to the given constraints, any cell $BadTile(j,i)$ can be reached by $BadTile(j,i-1)$ , $BadTile(j-1,i-1)$\{if it exists\} and $BadTile(j+1,i-1)$\{if it exists\}.

4. Total number of ways to pass the given bridge = (Number of ways to reach $(1,n)$ + Number of ways to reach $(2,n)$ + Number of ways to reach $(3,n)$).

5. Number of ways to pass the bridge successfully = $(OPT[3,n] + OPT[2,n] + OPT[1,n])$.

6. Instead of creating a new array $OPT$ of $size\,(3,n)$ we can **use the same array $\boldsymbol{BadTile}\,(\mathbf{3},\boldsymbol{n})$ to save the space complexity**.

7. So, while iterating through the BadTile array, **if the $\boldsymbol{BadTile}\,(\boldsymbol{j},\boldsymbol{i}) = \mathbf{0}$, we skip the tile and proceed to the next tile**. If not, we compute the value of the tile using the above recurrence relation.

8. Now the recurrence relation variables change as follows
   $BadTile[3,i] = BadTile[3,i-1] + BadTile[2,i-1]$
   $BadTile[2,i] = BadTile[3,i-1] + BadTile[2,i-1] + BadTile[1,i-1]$
   $BadTile[1,i] = BadTile[2,i-1] + BadTile[1,i-1]$

9. Column $'i'$ can be computed only after the previous column $'i-1'$ is iterated and computed.

10. The **Base case** for the array is **1st column of modified BadTile** array itself.

11. We start iterating from column 2 and once all the elements of column 2 are computed and iterated, we proceed to column 3.
12. After all the iterations are done, the above array would look as below

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 2 | 2 | 2 | 0 | 0 | 0 | 2 | 2 |
| 0 | 2 | 0 | 0 | 0 | 2 | 0 | 2 | 0 | 4 |
| 1 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 0 |

13. The number of ways for above example = $Badtile[3,n] + Badtile[2,n] + Badtile[1,n] = (2+4+0) = 6$.

**Run-Time complexity:**

1. Time complexity to iterate all the elements to flip the numbers from $'0'$ to $'1'$ and vice-versa = $O(3n)$.

2. We perform one more iteration to compute the values of tiles $(j, i)$.
3. Assuming the time complexity to perform add operation is $O(1)$, the time complexity $\approx O(3n)$.
4. The total time complexity in terms of input size $\approx O(3n)$.

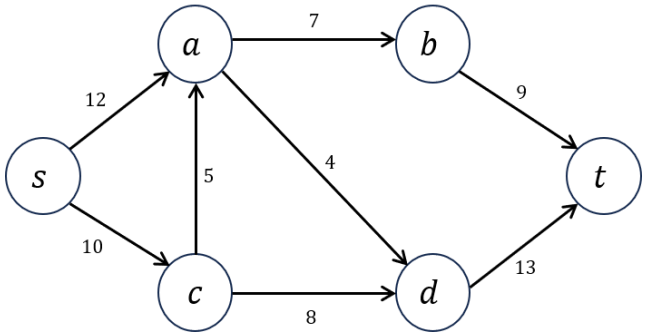
4. Given a flow network with the source $s$ and the sink $t$, and positive integer edge capacities $c$. Prove or disprove the following statement: if deleting edge $e$ reduces the original maximum flow more than deleting any other edge does, then edge $e$ must be part of a minimum $s-t$ cut in the original graph.

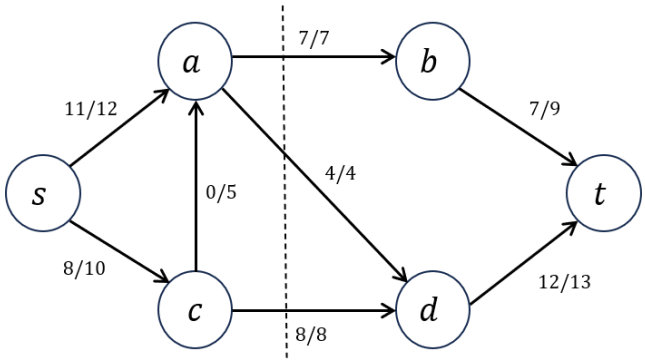
**ANS:**

**Given Statement:** If deleting edge e reduces the original maximum flow more than deleting any other edge does, then edge e must be a part of a minimum s-t cut in the original graph.

The above statement is False and the below example illustrates an example where this statement does not hold true.

**Proof by Counter Example:**




$MAX\ FLOW = 19$

$MIN - CUT$
$Partition\ 1 = \{s, a, c\}$
$Partition\ 2 = \{b, d, t\}$

**ANALYSIS:**

If we remove '$\boldsymbol{a-b}$' edge then the Max Flow will be **12**

If we remove '$\boldsymbol{a-d}$' edge then the Max Flow will be **15**

If we remove '$\boldsymbol{c-d}$' edge then the Max Flow will be **11**

If we remove '$\boldsymbol{c-a}$' edge then the Max Flow will be **19**

If we remove '$\boldsymbol{s-a}$' edge then the Max Flow will be **10 (2/5 for $'c-a'$ edge)**

If we remove '$\boldsymbol{s-c}$' edge then the Max Flow will be **11**

If we remove '$\boldsymbol{b-t}$' edge then the Max Flow will be **12**

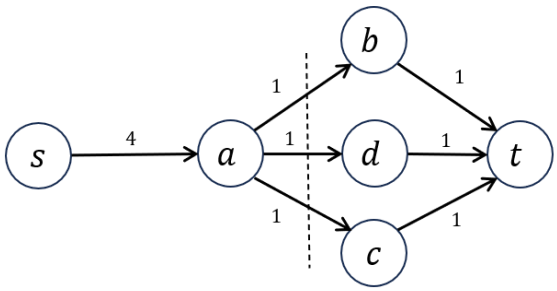
**If we remove '$\boldsymbol{d-t}$' edge then the Max Flow will be 7**

**OBSERVATION / CONCLUSION:** Max Flow of our original graph is reduced maximum by removing the $d-t$ edge more than removing any other edge in the graph. But the edge $d-t$ is not a part of min-cut for our network flow as depicted in the diagram above. Hence proved that the edge e that reduces the maximum flow of original graph when removed need not be a part of $s-t$ min-cut.

5. 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.

**ANS:** If we increase the capacity of every edge by 1, then $s-t$ might not remain the minimum cut for our updated graph in all the cases. The below example illustrates an example where this statement does not hold true.

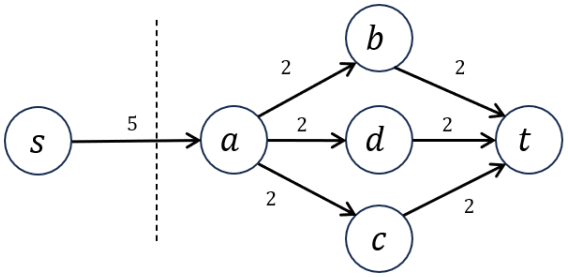
**Proof by Counter Example:**




$MAX\ FLOW = 3$

$MIN-CUT$
$Partition\ 1 = \{s, a\}$
$Partition\ 2 = \{b, c, d, t\}$

$\downarrow$ $Increasing\ Capacity\ of\ all\ edges\ by + 1$




$MAX\ FLOW = 5$

$MIN - CUT$
$Partition\ 1 = \{s\}$
$Partition\ 2 = \{a, b, c, d, t\}$

Hence, proved that the min-cut will not be same if we increase the capacity of all edges by +1 where all the capacities $\in +ve\ Integers$

6. (**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).

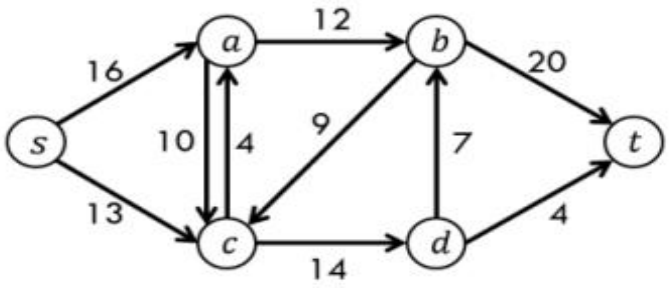
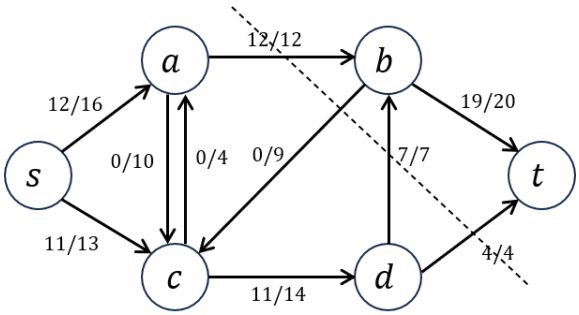



Figure 2: G1

a) **ANS:** By applying Ford-Fulkerson Algorithm we can find the max-flow as below




$MAX\ FLOW = 23$

$MIN - CUT$
$Partition\ 1 = \{s, a, c, d\}$
$Partition\ 2 = \{b, t\}$

(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).

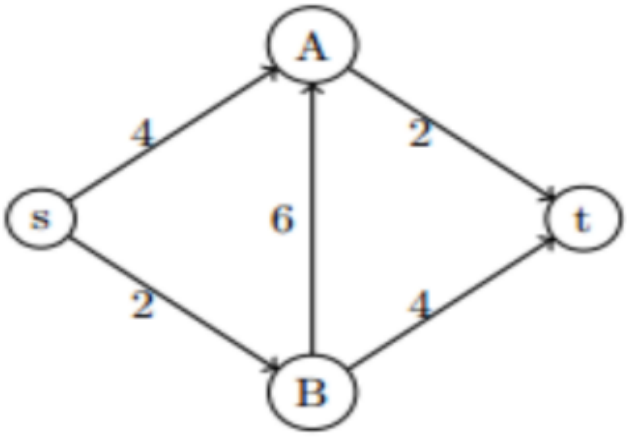
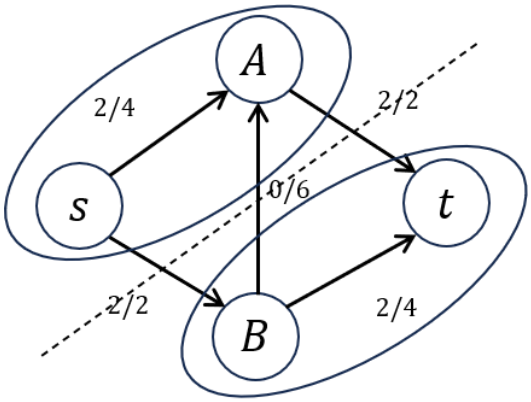



Figure 3

**b)** **ANS:** By applying Ford-Fulkerson Algorithm we can find the min-cut as below




$MAX\ FLOW = 4$
$MIN - CUT$
$Partition\ 1 = \{s, A\}$
$Partition\ 2 = \{B, t\}$

7. (**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.

**ANS:**

**STEP 1:** Reduce the problem statement to Network Flow and Create a flow with Source $s$, Sink (Target) $t$, Vertices V (Clients $c_1, c_2, \ldots . c_n$ and Base Stations $b_1, b_2, \ldots . b_k$), Edges E (capacities)

**Create Network Flow:**

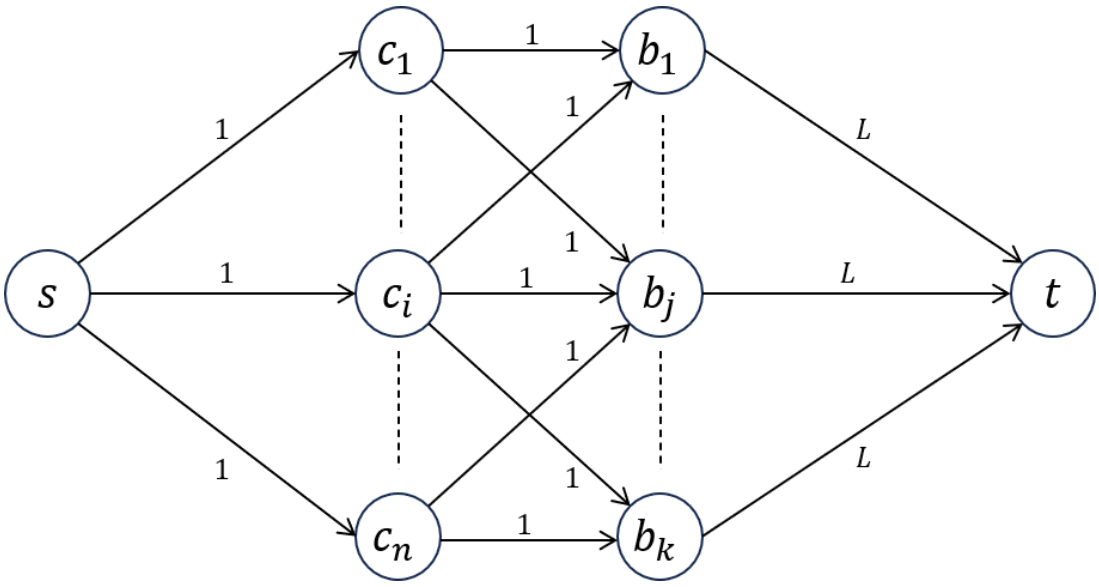
1) Source node $'s'$
2) Sink node $'t'$
3) A node for each client $'c_i'$
4) A node for each base station $'b_j'$

Connect each client node $'c_i'$ to the source node $'s'$ with an edge of capacity 1. This represents the requirement that each client must be connected to exactly one base station.

Connect each base station node $'b_j'$ to the sink node $'t'$ with an edge of capacity $'L'$. This represents the load parameter, which limits the number of clients that can be connected to each base station.

For each pair (client $'c_i'$, base station $'b_j'$), check if the distance (given (x, y) co-ordinates for client locations and base stations location find euclidean distance) between $'c_i'$ and $'b_j'$. If the distance is less than R, create an edge from $'c_i'$ and $'b_j'$ with a capacity of 1. This represents the range constraint, ensuring that a client can only be connected to a base station if the base station is within distance $'R'$ .

Following the above constraints and metrics we will end up with a network flow (Bi-Partite Graph) as depicted below



**STEP 2:** Every client can be served i.e., ∃ solution that meets all the client's requirements if and only if the Max Flow is $n$ i.e. (Sum of Flows coming out of $b_1, b_2, \ldots . b_k$) must be equal to $n$ reaching sink (target $t$).

By applying the Ford-Fulkerson algorithm, we can find the maximum flow in this network.

If the maximum flow in the network is equal to $'n'$ (the number of clients), it means that every client can be connected simultaneously to a base station while satisfying the range $'R'$ and load $'L'$ constraints. If the maximum flow is less than $'n'$, it indicates that there is no valid assignment of clients to base stations that satisfies the constraints.

**PROOF OF CORRECTNESS:**

Outward flow from $s$ should be $n$ (1 for each client) if each client is connected to atleast one valid base station. i.e., all the edges from $s - c_i$ must be saturated. Note that Max flow can't be greater than $n$ as we have only $n$ clients.

If the final flow at the sink $t$ is equal to $n$ then it means that all the clients have been connected to a valid base station following the range and load constraints. The saturated edges between $c_i - b_j$ will denote the valid connection between client and base station (i.e., which client is connected to which base station).

Hence, we have solved this problem by reducing the problem to a network flow.

8. (**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 $F \subseteq E$ 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.

**ANS:**

**Given:** Directed Graph $G$ with $V$ vertices, $E$ edges, source $s$ and sink $t$, and the all edges have unit capacities.

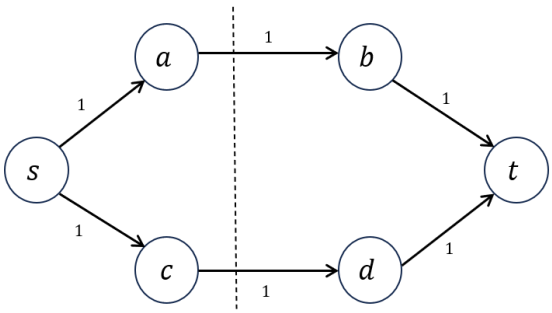
**Aim:** To find a polynomial-time algorithm for deleting $k$ edges so as to reduce the maximum $s - t$ flow in $G$ as much as possible.

**STEP 1:** Copy and replicate the graph $G$ *to* $G'$ so that the original graph is not altered. Run Ford-Fulkerson Algorithm (Optimal Versions like Edmond Karp implementation) on $G'$ to find the min $s - t$ cut (i.e., the Max-Flow of the Graph $G'$).

**STEP 2:** Here, we can encounter 2 cases. Let us consider set of edges present in the min-cut as $F$ (which is subset of set Edges $E$ of Graph $G'$) i.e., $F \subseteq E$

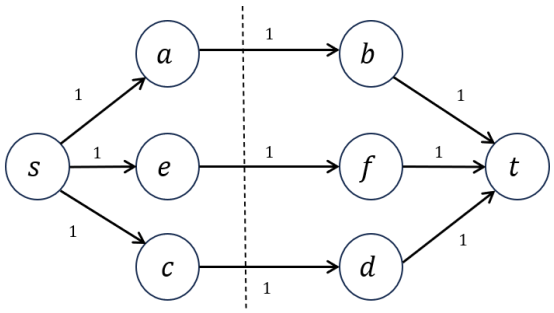
- Case 1: $Number\ of\ edges\ in\ F \leq k$
- Case 2: $Number\ of\ edges\ in\ F > k$

**Ex:** If $k = 2$, the below example graphs denote case 1 and case 2 respectively




$MAX\ FLOW = 2$

$MIN - CUT$
$Partition\ 1 = \{s, a, c\}$
$Partition\ 2 = \{b, d, t\}$




$MAX\ FLOW = 3$

$MIN - CUT$
$Partition\ 1 = \{s, a, e, c\}$
$Partition\ 2 = \{b, f, d, t\}$

**STEP 3:**

**Case 1:** If the number of edges in $F \leq k$, it means that we can delete all the edges in the set $F$ (i.e., all the edges in the min-cut of Graph $G'$) which makes the max-flow of the graph $'0'$.

**Case 2:** If the number of edges in $F > k$, it means that we can delete $'k'$edges in the set $F$ (i.e., $'k'$ edges in the min-cut of Graph $G'$) which makes the max-flow of the graph $'maxflow\ (G) - k'$.

**STEP 4:** Hence, the max-flow of the graph $G'$ after removing $k$ edges with unit capacity can be written as $max(0, maxflow(G) - k)$.

**RUN-TIME COMPLEXITY:**

The above algorithm is a polynomial run-time algorithm as we need polynomial run-time $O(V.E^2)$ to run the Ford-Fulkerson Algorithm - Edmond Karp's implementation and linear time to parse through $F$ set of edges and remove $k$ edges to reduce the flow.

9. (**Network Flow**) Counter Espionage Academy instructors have designed the following problem to see how well trainees can detect $SPY$'s in an $n \times n$ 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 neighboring $P$, then move to a neighboring $Y$. (They can move north, east, south or west to get to a neighbor.) 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.

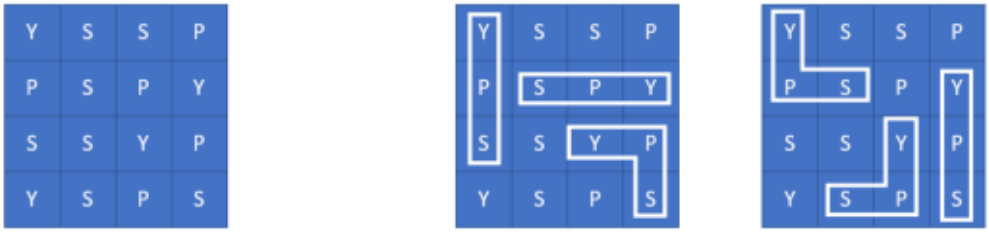



Figure 4

**ANS:**

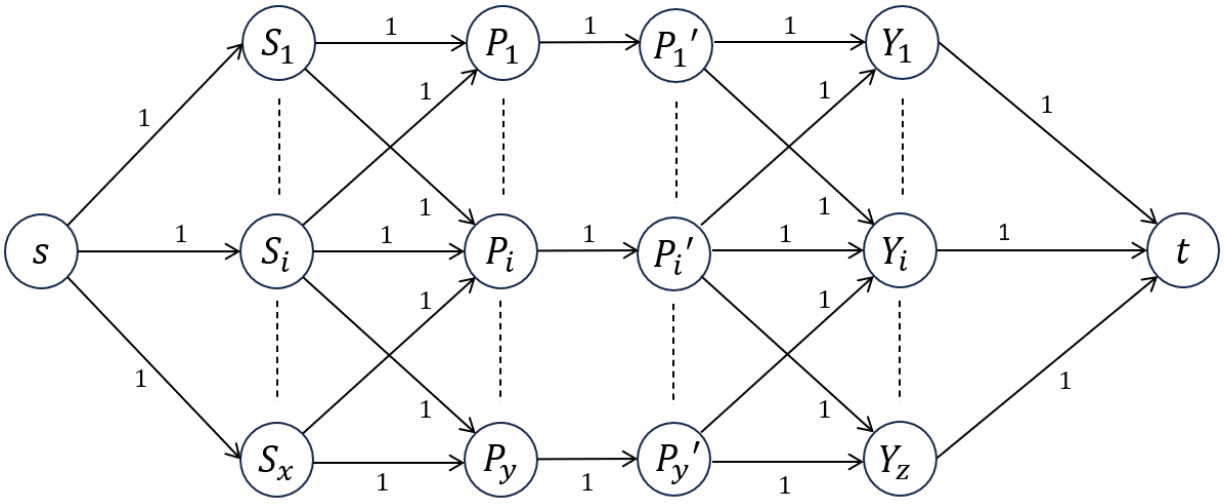
To find the largest number of disjoint copies of the word SPY in an n × n grid, we can use a network flow-based algorithm. In this problem, we can treat the grid as a flow network, where we can find the maximum number of disjoint paths representing the word SPY.

**Create a flow network:**

1. Create a source node $s$ and a sink node $t$.
2. Create a node for each cell in the $n \times n$ grid.
3. Connect the source node $s$ to all $'S'$ nodes in the grid i.e., $S_1, \ldots S_i, \ldots \ldots S_x$ (If there are $'x'$ $S$ cells in the grid) with unit edge capacities (i.e., each edge has a capacity of 1) enforcing that each $'S'$ have to be used only once, as we are finding the disjoint sets of word $'SPY'$.
4. Connect $x$ $'S'$ nodes to $y$ $'P'$ nodes (i.e., if the grid consists of $y$ $'P'$ cells) by adding edges from each $'S'$ node to its adjacent $'P'$ nodes (North or South or East or West), allowing movement from $'S'$ to $'P'$. This ensures that each $'S'$ can connect to any $'P'$ that is adjacent to it in the grid.
5. Replicate $'P'$ nodes to create $y$ $'P''$nodes and connect them with the original counterparts with unit edge capacity i.e., $P_1 \rightarrow P_1' \ldots . P_x \rightarrow P_x'$ . This is to enforce a constraint that each $'P'$ cell can only be present in one $SPY$sequence and not counted in the multiple sequences if it has another adjacent valid pair of $'S'$ and $'Y'$ cells.

6. Connect $y$ $'P''$ nodes to $z$ $'Y'$ nodes (i.e., if the grid consists of $z$ $'Y'$ cells) by adding edges from each $'P''$ node to its adjacent $'Y'$ nodes (North or South or East or West of the actual counterpart $P_1$, ..... $P_i$, ... ... $P_y$ cells in the grid), allowing movement from $'P'$ to $'Y'$ in the grid ensuring that each $'P'$ can connect to any $'Y'$ that is adjacent to it.
7. Connect all the $z$ $'Y'$ nodes to the sink node $t$ with unit edge capacity (i.e., each edge has a capacity of 1) enforcing that each $'Y'$cell have to be used only once, as we are finding the disjoint sets of word $'SPY'$ which will not allow any overlap.

After following the above 7 steps, we will obtain a network flow as follows:




Run the Ford-Fulkerson Algorithm (Best Possible Variant like Edmond Karp's etc) to find the Max-Flow in the above Network Flow. The resultant (i.e., the Max-Flow of this Network Flow) will be our maximum number of disjoint copies of word SPY in the given $n \times n$ grid.

10. (**Network Flow**) USC students return to in person classes after a year long 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.

**ANS:**

(a)

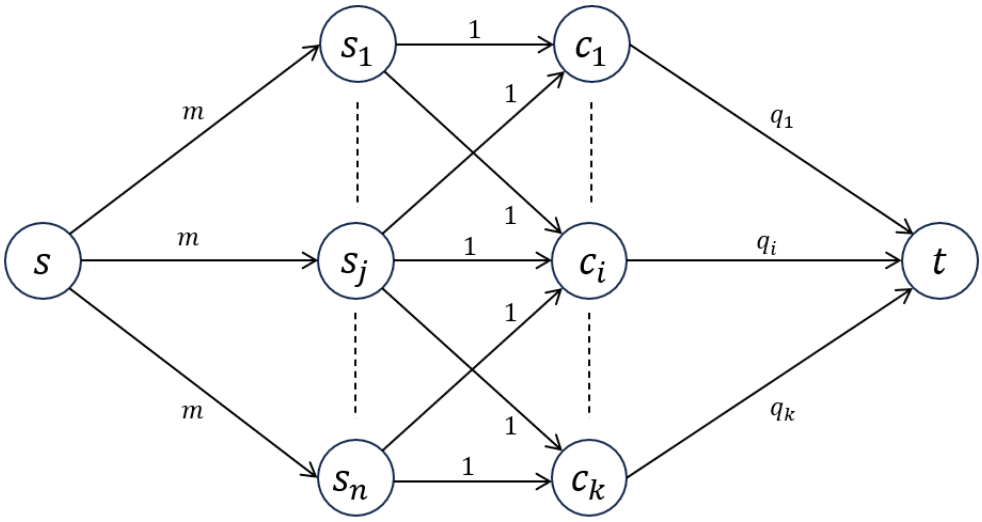
**STEP 1:** Create a flow network where there is a source node $s$, a sink node $t$, and intermediate nodes representing students $s_1$, ..... $s_j$, ... ... $s_n$ and classes $c_1$, ..... $c_i$, ... ... $c_k$ .

**STEP 2:** Connect the source $s$ to each student node with an edge of capacity $m$ for student $s_j$. This ensures that each student can sign up for at least $m$ classes as given that student's preferences $p_j \geq m$.

**STEP 3:** Connect each student node $s_j$ to the maximum of $p_j$ classes $c_i$ they want to sign up for, with edges of capacity 1. Student's preferences are represented by these edges.

**STEP 4:** Connect each class node to the sink $t$ with an edge of capacity $q_i$, representing the class's capacity constraint.

By following the above 4 steps we will end up creating the Network Flow as follows:




**STEP 5:** Apply the Ford-Fulkerson algorithm to find the maximum flow in the network.

**STEP 6:** If the maximum flow is equal to the sum of the required flow from the source to each student (which is $n \times m$), then all students can be enrolled as full-time students. Otherwise, there is no way to satisfy the requirements for all students.

**PROOF OF CORRECTNESS:**

We can use the Max-Flow Min-Cut theorem, which states that the maximum flow in a network is equal to the minimum capacity of a cut that separates the source $s$ from the sink $t$.

The outward flow from the source $s$ should be $n \times m$ as $n$ students must enroll in atleast $m$ classes to become full time students. The Max-Flow incoming at sink $t$ must also be $n \times m$ if it meets at the constraints and record valid student enrolment meeting all the capacity constraints.

a. If the maximum flow is less than $n \times m$, there must be a cut with a capacity less than $n \times m$. This cut represents a division between students and classes, indicating that some students cannot be enrolled in $m$ classes. In this case, it is impossible for all students to be enrolled as full-time students following the given constraints.

b. If the maximum flow is equal to $n \times m$, then there is no cut with capacity less than $n \times m$. This implies that there is no way to separate the students from the classes without cutting off more than $n \times m$ units of flow. Thus, all students can be enrolled as full-time students.

This algorithm runs in polynomial time, and it correctly determines whether or not all students can be enrolled as full-time students while respecting class capacities $q_i$ .

(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).

**ANS:**

(b)

To determine whether there exists a feasible solution where each class has a student representative, but no student is a representative for more than $r$ classes (where $r < m$), we can modify the algorithm from part (a)

**STEP 1:** Start with the solution obtained in part (a), where it is confirmed that each student is enrolled in $m$ classes.

**STEP 2:** Create a flow network where there is a source node $s$, a sink node $t$, and intermediate nodes representing classes $c_1, \ldots. c_i, \ldots \ldots c_k$ and students $s_1, \ldots. s_j, \ldots \ldots s_n$.

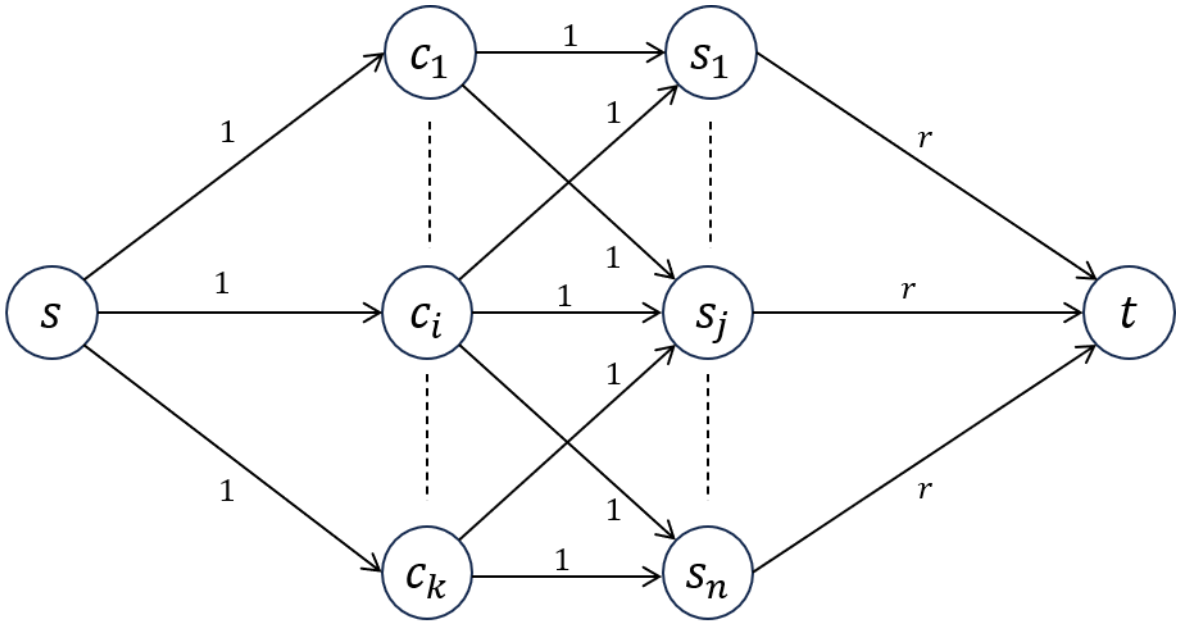
**STEP 3:** Connect the source $s$ to each student node with an edge of capacity 1 for class $c_i$. This ensures that each class can have one and only one student representative.

**STEP 4:** Connect each class node $c_i$ to the students $s_j$ based on the residual graph from part (a) solution, with edges of capacity 1. Student's enrolled in a specific class are represented by these edges.

Saturated Edge between $s_j - c_i$ (Reverse Edge in Residual Graph of part (a)) indicates that particular student $s_j$ has enrolled in a specific class $c_i$.

**STEP 5:** Connect each student node to the sink $t$ with an edge capacity of $r$, representing the capacity constraint that each student can be a student representative for a maximum of $r$ classes.

By following the above 4 steps we will end up creating the Network Flow as follows:




**STEP 6:** Apply the Ford-Fulkerson algorithm to find the maximum flow in the network.

**STEP 7:** If the maximum flow is equal to the sum of the required flow from the source to each class (which is $k$), then all classes have a student representative assigned. Otherwise, there is no way to satisfy the requirements.

## PROOF OF CORRECTNESS:

We can use the Max-Flow Min-Cut theorem, which states that the maximum flow in a network is equal to the minimum capacity of a cut that separates the source $s$ from the sink $t$.

a. If the maximum flow is less than $k$, there must be a cut with a capacity less than $k$. This cut represents a division between classes and students, indicating that some classes cannot have a student representative. In this case, it is impossible for all the classes to have a student representative following the given constraints.

b. If the maximum flow is equal to $k$, then there is no cut with capacity less than $k$. This implies that there is no way to separate the classes from the students without cutting off more than $k$ units of flow. Thus, all classes must have atleast one student representative.

This algorithm runs in polynomial time, and it correctly determines whether or not all classes can be allocated with a student representative while respecting restriction constraints $r$.

********** **THE END** **********