



CSCI 570

Exam 2 Review Slides



Dynamic Programming

Xubin Zhang

Max Possible Comfort with Seating Arrangement

Suppose you are arranging the seats for 570 Midterm 2.

There are N adjacent seats numbered 1 to N in the first row. If you assign someone to the i -th seat, this student will get $w(i)$ comfort ($w(i) \geq 0$).

However, if you place a student in the i -th seat, you can't place the other students in seat $i-1$ or $i+1$.

What is the maximum possible comfort you can get?

Seat	1	2	3	4	5	6	7
w	1	9	10	9	1	5	4

First Step: Play with examples

- **Question 1:** Give an example that starting from either the first or second seat and then picking every other seat (e.g. taking the seats 1, 3, 5, ... or 2, 4, 6, ...) does not produce an optimal solution.
- Example: [2, 1, 1, 2], the optimal arrangement is picking seat 1 and 4.

Seat	1	2	3	4
w	2	1	1	2

First Step: Play with examples

- **Question 2:** Consider the following algorithm: from the seat of the largest $w(i)$ to that of the lowest, pick the seat as long as it is allowed ($i-1$ and $i+1$ are not taken). Give an example that this algorithm fails to achieve the optimal solution.
- Example: $[9,10,9]$, the optimal solution is taking seat 1 and 3. **Greedy algorithm does not work.**

Seat	1	2	3
w	9	10	9

Second Step: Define the State/Subproblem

- **Question 3:** Consider the subproblem: “What is the maximum possible comfort that can be achieved using only seats 1 to i ?”. Let $\text{OPT}(i)$ denote the answer to this question for any i . **Is this a proper definition?**
- **Question 4:** Do we need an extra dimension for the number of arranged students like $\text{OPT}(i,k)$, or not?
- **Question 5:** Define $\text{OPT}(0)=0$, as when we have no seats, we can't arrange any student. What is $\text{OPT}(1)$? What is $\text{OPT}(2)$?
 - $\text{OPT}(1) = w(1)$, $\text{OPT}(2) = \max\{ w(1), w(2) \}$

Third Step: Find the Recurrence for $\text{OPT}(i)$

- **Question 6:** Suppose you know that the best arrangement using only seats 1 to i **does not** place a student in seat i . In this case, express $\text{OPT}(i)$ in terms of $w(i)$ and $\text{OPT}(j)$ for any $j < i$.
- $\text{OPT}(i) = \text{OPT}(i-1)$

Seat	...	$i - 2$	$i - 1$	i
w	...	8	10	?

Third Step: Find the Recurrence for $\text{OPT}(i)$

- **Question 7:** Suppose you know that the best arrangement using only seats 1 to i **does** place a student in seat i . In this case, express $\text{OPT}(i)$ in terms of $w(i)$ and $\text{OPT}(j)$ for any $j < i$.
- $\text{OPT}(i) = \text{OPT}(i-2) + w(i)$

Seat	...	$i - 2$	$i - 1$	i
w	...	8	10	?

Third Step: Find the Recurrence for $\text{OPT}(i)$

- **Question 8:** Put these two cases together, you will get the recurrence function
- $\text{OPT}(i) = \max\{ \text{OPT}(i-1), \text{OPT}(i-2) + w(i) \}.$

Fourth Step: Write down pseudo code

- Initialization and boundary condition:
 - initialize OPT as a length $N+1$ array indexed from 0 to N
 - set $OPT(0) = 0$ and $OPT(1) = w(1)$.
- The order of solving the subproblems:
 - We compute $OPT(i)$ for every i from 2 to N by the formula $OPT(i) = \max\{ OPT(i-1), OPT(i-2) + w(i) \}$.
 - For $i = 2$ to N :
$$OPT(i) = \max\{ OPT(i-1), OPT(i-2) + w(i) \}$$
- Return:
 - Return $OPT(N)$ from the array as the maximum possible comfort.

Fifth Step: Time Complexity Analysis

- What is the complexity of your solution? Is this an efficient solution?
- Here, the runtime complexity is $O(n)$,
- This algorithm is efficient, since the time complexity is a polynomial function w.r.t. input size n .

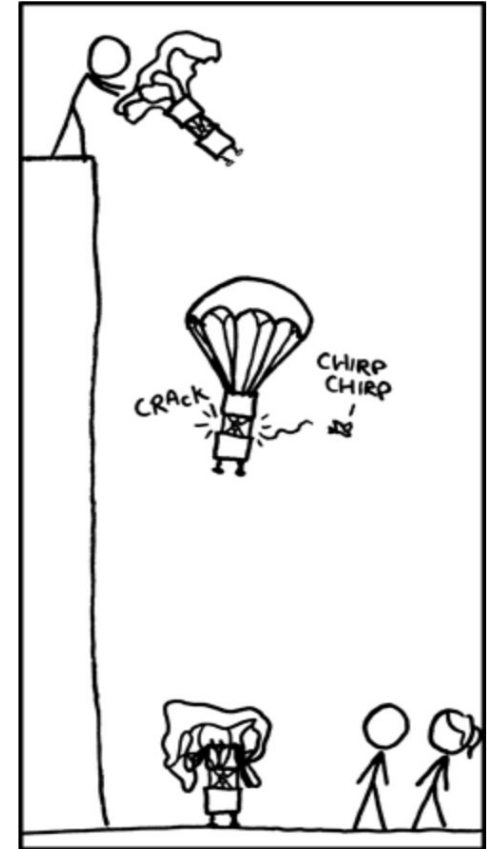
Egg dropping puzzle

Suppose you are in an n -story building and you have k eggs in your bag. You want to find out the lowest floor from which dropping an egg will break it. What is the minimum number of egg-dropping trials that is sufficient for finding out the answer in all possible cases?



Egg dropping puzzle: Example

- *For example, if the building has 100 floors ($n = 100$)*
 - *we have only 1 egg, then we must do 100 trials to find the threshold*
 - *we have 2 eggs, then we can find the threshold in 14 trials!*



[XKCD 510](#), alt text: "I hear my brother Ricky won his school's egg drop by leaving the egg inside the hen."

Egg dropping puzzle: Constraints

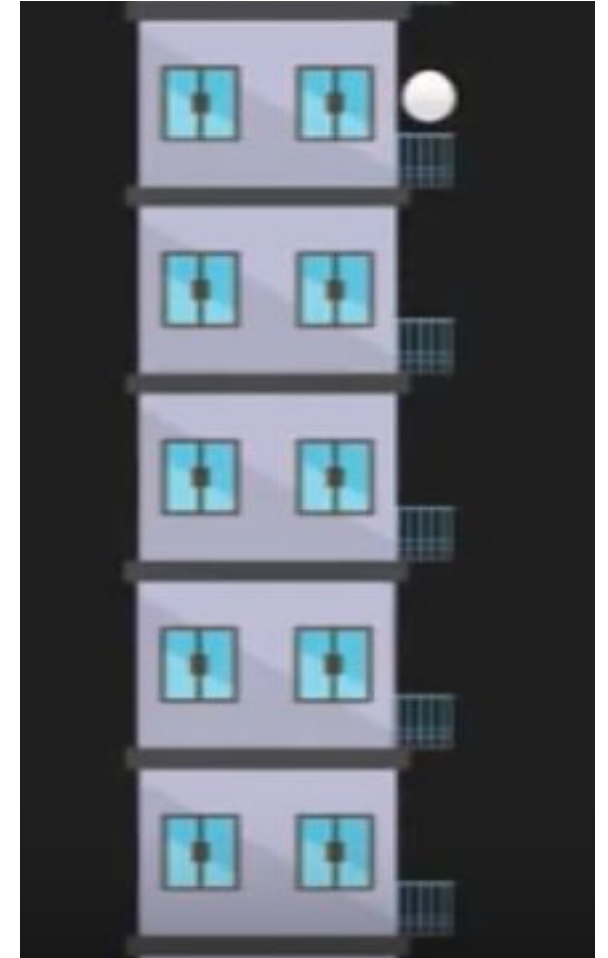
Following are the constraints:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks from a fall, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.

Remember, We are finding the least number of drops to find the threshold floor and not the threshold floor itself!

Egg dropping puzzle: Base Cases

- If there are n floors and 1 egg, we need to do n trials worst case
- If there are k eggs and 0 floors, we would need to do 0 trials
- If there are k eggs and 1 floor, we need to do 1 trial



Define the subproblem

- Let $\text{OPT}[i, j]$ denote the minimum number of trials required for an i story building and j eggs

Find the Recurrence for $\text{OPT}[i, j]$

- Obtain $\text{OPT}[i, j]$ given all the results of the solved subproblems
- Suppose we first drop an egg from floor $x \leq i$.
 - If the egg breaks, then we learn that threshold $\leq x$, but we are left with $j - 1$ eggs. To find out the threshold from this state we would need extra $\text{OPT}[x - 1, j - 1]$ steps.
 - If the egg doesn't break, then we learn that $x < \text{threshold}$ and we still have k eggs. All the above $(i - x)$ floors are the possible candidates and we have j eggs. Therefore, we would need to do $\text{OPT}[i - x, j]$ extra steps.
- we must traverse for each floor x from 1 to i to find the minimum
- $\text{OPT}[i, j] = \min_{1 \leq x \leq i} \{1 + \max\{\text{OPT}[x - 1, j - 1], \text{OPT}[i - x, j]\}\}$

Example



PseudoCode

- Base Case:
 - $\text{OPT}[0, j] = 0$; // 0 floor
 - $\text{OPT}[1, j] = 1$; // 1 floor
 - $\text{OPT}[i, 1] = i$; // 1 egg
 - $\text{OPT}[i, j] = \text{MAX_INTEGER}$, otherwise
- Iteration:
 - for $j = 2$ to k :
 - for $i = 2$ to n :
 - for $x = 1$ to i :
 - $\text{OPT}[i, j] = \min(\text{OPT}[i, j], 1 + \max\{\text{OPT}[x - 1, j - 1], \text{OPT}[i - x, j]\})$
- Return $\text{OPT}[n, k]$

DP Table

Eggs\Floors	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1	2	2	3	3	3
3	0	1	2	2	3	3	3

Complexity Analysis

- Time complexity is $O(n^2k)$
- Space complexity is $O(nk)$
- Not efficient, pseudo-polynomial time

100 floors 2 eggs

- We start trying from 14'th floor.
 - If Egg breaks on 14th floor we one by one try remaining 13 floors, starting from 1st floor.
 - If egg doesn't break we go to 27th floor.
- If egg breaks on 27'th floor,
 - we try floors from 15 to 26.
 - If egg doesn't break on 27'th floor, we go to 39'th floor.
- And so on...

Another approach

- $OPT[i, j]$ means that, given i trials, j eggs what is the maximum number of floor that we can check.
- The recurrence equation is:
- $OPT[i, j] = OPT[i - 1, j - 1] + OPT[i - 1, j] + 1$,
which means we take 1 trial to a floor,
if egg breaks, then we can check $OPT[i - 1, j - 1]$ floors.
if egg doesn't break, then we can check $OPT[i - 1, j]$ floors.
- $OPT[i, j]$ is the number of combinations and it increase exponentially to n
- Return i for solution

Pseudo Code

- Base case:
 - $\text{OPT}[i, j] = 0$
- Iteration:
 - $i = 0$
 - while ($\text{OPT}[i, k] < n$) {
 - $i++$;
 - for $j = 1$ to k :
 - $\text{OPT}[i, j] = \text{OPT}[i - 1, j - 1] + \text{OPT}[i - 1, j] + 1$}
- Return i

Complexity

- Time complexity is $O(k \log n)$
- Space complexity is $O(nk)$
- Not efficient, pseudo-polynomial time

Optimization to 1-Dimensional DP

- Base case:
 - $OPT[i] = 0$ $i=0$ to k
- Iteration:
 - $i = 0$
 - while ($OPT[k] < n$) {
 - $i++$;
 - for $j = k$ to 1 :
 - $OPT[j] += OPT[j - 1] + 1$}
- Return i
- Time complexity $O(k \log n)$
- Space complexity is $O(k)$
- Not efficient, pseudo-polynomial time

Take Home Tips

- If you are not certain whether this problem can be solved by DP (or DFS/BFS), construct some examples that the solutions can be computed easily.
 - Seek the definition of sub-problems,
 - 1-Dimensional or 2-Dimensional or ?
 - Satisfy Optimal Substructure property ?
- Find the solution of sub-problems (connecting),
 - Decide the order of sub-problems. (Topological sort on a DAG)
 - Think about how to achieve the solution of this sub-problem.
- Tip: get familiar with common models and try them one by one. Use some simple examples to verify.



Dynamic Programming

Mengxi Wu

Number of Valid Seating Arrangements

You are asked by the city mayor to organize a COVID-19 panel discussion regarding the reopening of your town. He told you that panel members include two types of people: those who wear a face covering (F) and those who do not wear any protection (P). He also told you that to reduce the spread of the virus, those who do not wear any protection must not be sitting next to each other in the panel. Suppose that there is a row of n empty seats. The city mayor wants to know the number of valid seating arrangements for the panel members you can do.

To help you see the problem better, suppose you have $n=3$ seats for the panel members.

- Some valid seating arrangements you can do are: F-F-F, F-P-F, P-F-P.
- Some invalid seating arrangements are: F-**P-P**, **P-P-P**.

Describe a dynamic programming solution to solve this problem.

Number of Valid Seating Arrangements

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

Solution: Let $f(n)$ be the number of valid seating arrangement for the panel members when you have n seats.

Number of Valid Seating Arrangements

b). Write a recurrence relation for $f(n)$. Be sure to state base cases

Solution:

- First, we can take for each valid seating of $n-1$ members and put F at the n -th seat.
- What about P at the n -th seat? We need to take a look at valid seating arrangements of $n-2$ members. Here, we can take for each valid seating of $n-2$ members, put F at the $(n-1)$ -th seat and then put P at the n -th seat.
- This exhausts all the ways of getting a valid seating. Hence, the recurrence is $f(n) = f(n - 1) + f(n - 2)$.

Number of Valid Seating Arrangements

c). Use the recurrence part from a and write pseudocode

Solution:

Initialize array f.

Set base cases $f[0] = 0$, $f[1] = 2$, $f[2] = 3$.

For $i > 2$ to n :

$$f[i] = f[i-1] + f[i-2]$$

return $f[n]$

Number of Valid Seating Arrangements

d). What is the run time of the algorithm?

Solution: Looking up the pre-computed value takes $O(1)$ and we only need to store n unique sub-problem we encounter. Hence, this will take $O(n)$ time.

Number of Valid Seating Arrangements

e) Is the algorithm presented in part (c) an efficient algorithm?

Solution: No, the solution is not efficient. The reason is that n is the numerical value on input, and that our complexity which is $O(n)$ depends on the numerical value of the input. The solution therefore has a pseudopolynomial run time and is not efficient.

Predict the Winner

You are given an integer array `nums`. Two players are playing a game with this array: player 1 and player 2.

Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., `nums[0]` or `nums[nums.length - 1]`) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array.

Return `true` if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return `true`. You may assume that both players are playing optimally.

Predict the Winner

Example

Input: nums = [1,5,2]

Output: false

Explanation:

- Initially, player 1 can choose between 1 and 2.
- If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If player 2 chooses 5, then player 1 will be left with 1 (or 2).
- So, final score of player 1 is $1 + 2 = 3$, and player 2 is 5.
- Hence, player 1 will never be the winner and you need to return false.

Predict the Winner

Knowing that this is an optimization problem (score maximization), should be a hint that we need a greedy or DP algorithm. To formulate a DP algorithm, we must think about what the sub-problems should be and how they can be recursively computed.

From experience, the **sub-problems for arrays are subsets of the array**.

The recursive relationship can be figured out by thinking about the choices we make to construct our solution. In this case, we are picking numbers, so the choice could be **‘Which number was picked — the one at the start or the one at the end?’**.

Predict the Winner

Let $dp[i][j]$ store the maximum effective score won by player 1 for the subarray $nums[i:j]$

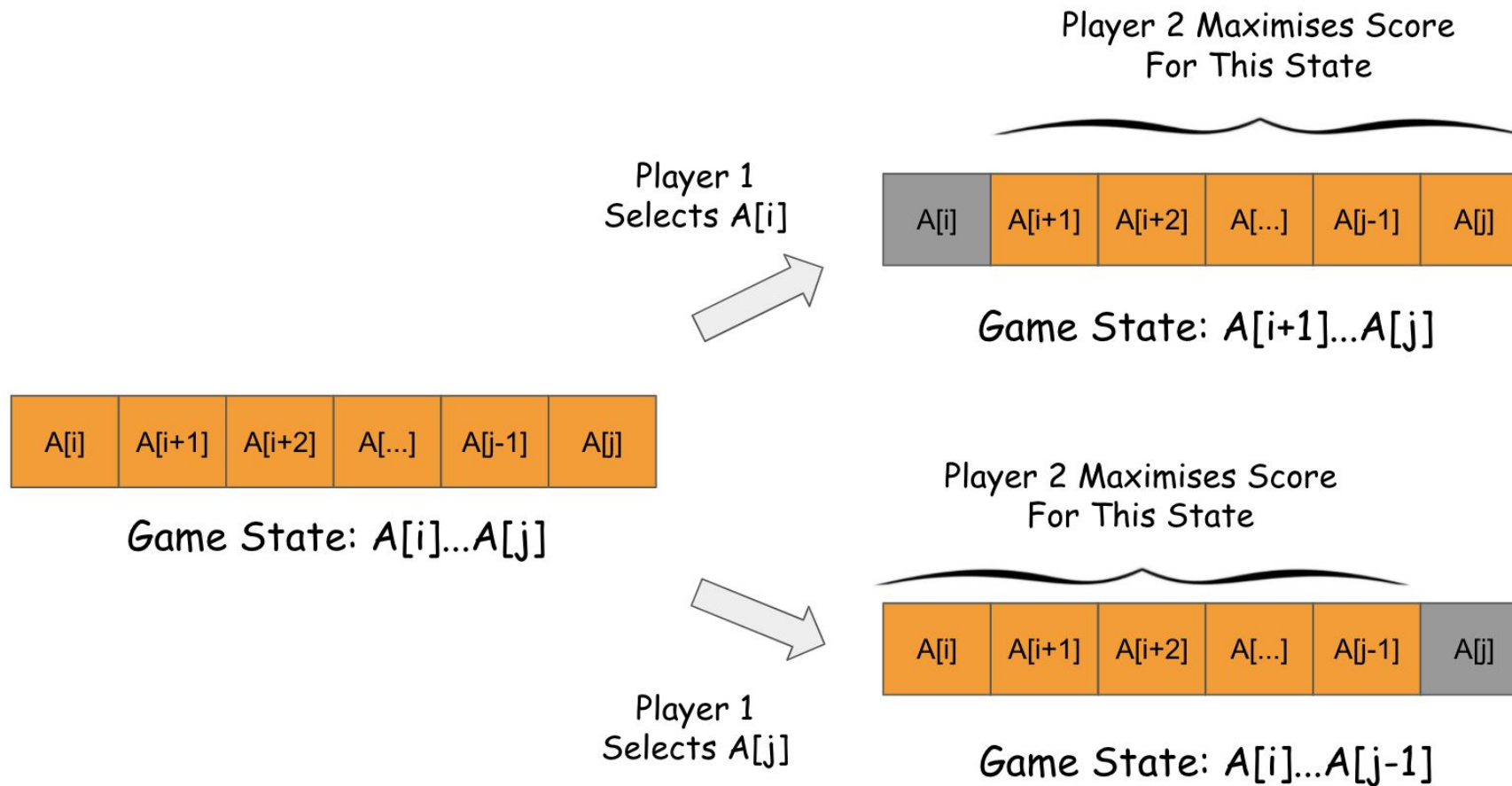
Choice 1:

If Player 1 picks $nums[i]$, $nums[i]$ is added to the score and the game state $nums[(i+1)...j]$ is given to Player 2. We know that Player 2 plays optimally as well so it will maximize its score for the game state presented which is $dp[i+1][j]$. This leaves the remaining numbers for Player 1: $(nums[i+1] + nums[i+2] + \dots + nums[j]) - dp[i+1][j]$.

Choice 2:

If Player 1 picks $nums[j]$, $nums[j]$ is added to the score and the game state $nums[(i)...(j-1)]$ is given to Player 2. We know that Player 2 plays optimally as well so it will maximize its score for the game state presented which is $dp[i][j-1]$. This leaves the remaining numbers for Player 1: $(nums[i] + nums[i+1] + \dots + nums[j-1]) - dp[i][j-1]$.

Predict the Winner



Predict the Winner

Let cumSum be a 1D array such that $\text{cumSum}[i] = \text{nums}[0] + \text{nums}[1] + \dots + \text{nums}[i]$.

Player 1 picks $\text{nums}[i]$:

- Player 2 gains $\text{dp}[i+1][j]$
 - The remaining score for Player 1 is $(\text{cumSum}[j] - \text{cumSum}[i+1] + \text{nums}[i+1]) - \text{dp}[i+1][j]$
- $$v1 = \text{nums}[i] + (\text{cumSum}[j] - \text{cumSum}[i+1] + \text{nums}[i+1]) - \text{dp}[i+1][j]$$

Player 1 picks $\text{nums}[j]$:

- Player 2 gains $\text{dp}[i][j-1]$
 - The remaining score for Player 1 is $(\text{cumSum}[j-1] - \text{cumSum}[i] + \text{nums}[i]) - \text{dp}[i][j-1]$
- $$v2 = \text{nums}[j] + (\text{cumSum}[j-1] - \text{cumSum}[i] + \text{nums}[i]) - \text{dp}[i][j-1]$$

Thus, $\text{dp}[i][j] = \max\{v1, v2\}$

Predict the Winner

To summarise the discussion above:

- Define $dp[n][n]$ such that $dp[i][j]$ ($j > i$) represents the maximum effective scores won by the first player playing with the game state $nums[i...j]$.
- Define $cumSum[]$ such that $cumSum[i] = nums[0] + nums[1] + \dots + nums[i]$.
- (Base Case) $dp[i][i] = nums[i]$ and $dp[i][i+1] = \max(nums[i], nums[i+1])$ because the first player will pick the bigger number.
- $v1 = nums[i] + cumSum[j] - cumSum[i+1] + nums[i+1] - dp[i+1][j]$
 $v2 = nums[j] + cumSum[j-1] - cumSum[i] + nums[i] - dp[i][j-1]$
 $dp[i][j] = \max(v1, v2)$
- Final answer is $dp[0][n-1] \geq cumSum[n-1] - dp[0][n-1]$.

Predict the Winner

Complexity Analysis

Time Complexity

Computing the cumulative sums array takes $O(n)$ time. To compute each entry of DP table, we are looking at two possible choices that Player 1 can make. For each choice, computing the maximum score takes $O(1)$ time. Thus, computing each entry of $dp[][]$ takes $O(1)$ time. The overall runtime is $O(n^2) \times O(1) = O(n^2)$.

Space Complexity

Our 2D DP table has $O(n^2)$ entries, $O(n^2)$ in total

Longest Increasing Path in a Matrix

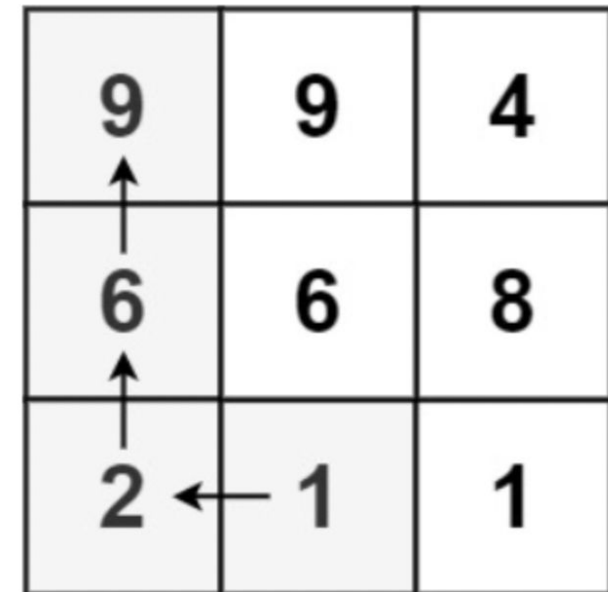
Given an $m \times n$ integers matrix, return the length of the longest increasing path in matrix.

From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

Example: matrix = $[[9,9,4],[6,6,8],[2,1,1]]$, output = 4

The longest increasing path is [1, 2, 6, 9]

9	9	4
6	6	8
2	1	1



Longest Increasing Path in a Matrix

The idea is to use dynamic programming. Maintain the 2D matrix, $dp[i][j]$, where $dp[i][j]$ stores the value of the length of the longest increasing sequence for submatrix starting from the cell $[i][j]$ (ith row and jth column).

We are allowed to go up,down,left,right. For all the four directions, we check if the cell in that direction is valid and the value in it is greater than the value in the current (i,j) cell. If it is greater, then we find the answer for that cell and add 1 to it.

Longest Increasing Path in a Matrix

if ($j < n - 1$ and ($\text{cell}[i][j] < \text{cell}[i][j + 1]$))
 $v1 = 1 + \text{Longest}(i, j + 1, \text{cell}, \text{dp}, n, m);$

if ($j > 0$ and ($\text{cell}[i][j] < \text{cell}[i][j-1]$))
 $v2 = 1 + \text{Longest}(i, j - 1, \text{cell}, \text{dp}, n, m);$

if ($i > 0$ and ($\text{cell}[i][j] < \text{cell}[i - 1][j]$))
 $v3 = 1 + \text{Longest}(i - 1, j, \text{cell}, \text{dp}, n, m);$

if ($i < n - 1$ and ($\text{cell}[i][j] < \text{cell}[i + 1][j]$))
 $v4 = 1 + \text{Longest}(i + 1, j, \text{cell}, \text{dp}, n, m);$

Longest Increasing Path in a Matrix

If cells in all the four directions are not valid or not greater than this cell then we just take this cell itself, so the length is 1. If other directions are valid, we pick the maximum of them:

```
dp[i][j] = max({v1,v2,v3,v4,1});
```

Then, we find the longest path beginning from all cells:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        if (dp[i][j] == -1)  
            Longest(i, j, matrix, dp,n,m);  
  
        ans = max(ans, dp[i][j]);  
    }  
}
```

Longest Increasing Path in a Matrix

The steps of this approach involve:

1. Input the n and m values and the matrix.
2. Create a 2-D dp vector and initialize the values in it to -1.
3. Find the $dp[i][j]$ value for each $cell[i][j]$ where the dp value is still -1.
4. For doing step 3, we have made another function that calculates the value for $cell[i][j]$ by looking at the $dp[i][j]$ value.
 1. If the $dp[i][j] \neq -1$, then we've already calculated the answer for this cell and we'll just return it.
 2. Otherwise, we call the same function for adjacent cells in the directions which are valid, i.e; inside the matrix and the value is greater than the value in the current cell. Then we add 1 to the returned answers for these cells.
 3. If no cell is valid, then the answer is 1 because the $cell[i][j]$ will itself form a path.
 4. The maximum of these will give the answer for $cell[i][j]$.
5. While calculating the $dp[i][j]$ for each cell, we keep updating the ans to the maximum of ans and $dp[i][j]$.
6. Once we're done with all the cells, our ans variable contains the required length so we return it.

Longest Increasing Path in a Matrix

Complexity Analysis

Time complexity

The time complexity $O(n*m)$, where n is the number of rows and m is the number of columns.

Reason: Because we're calculating the values of all the cells once and there are a total of $n*m$ cells.

Space complexity

$O(n*m)$, where n is the number of rows and m is the number of columns.

Reason: We're storing the values of all cells which will take $O(n*m)$ space and the matrix itself will take another $O(m*n)$ space. Thus the total space complexity is $O(2(m*n))$, which is nearly equal to the $O(m*n)$.



Network Flow

Xiyang Zhang

Warmup - True or False?

1. Let $(S, V-S)$ be a minimum (s,t) -cut in the network flow graph G . Let (u, v) be an edge that crosses the cut in the forward direction, i.e., $u \in S$ and $v \in V-S$. Then increasing the capacity of the edge (u, v) will also increase the maximum flow of G .

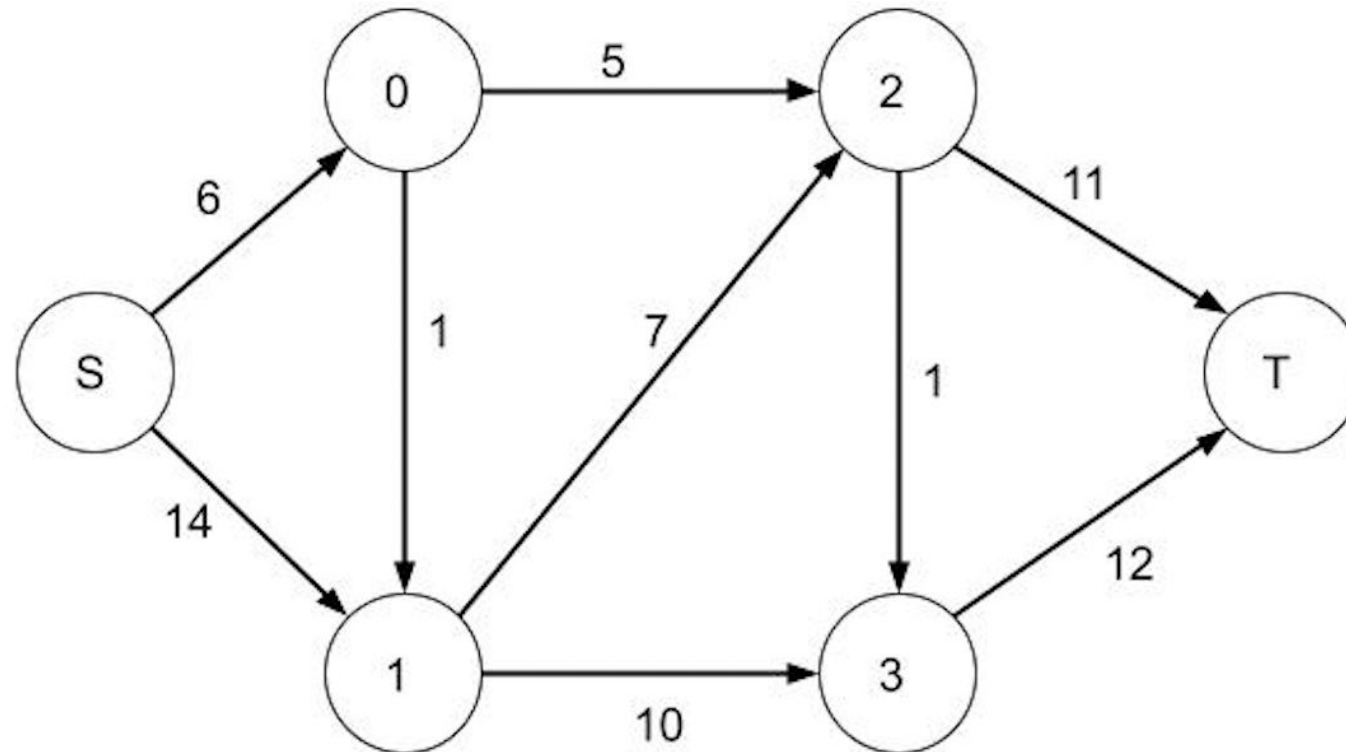
False!

2. If all of the edge capacities in a network flow graph are an integer multiple of 42, then the value of the maximum flow will be a multiple of 42.

True!

Question 1 - Scaled Ford-Fulkerson

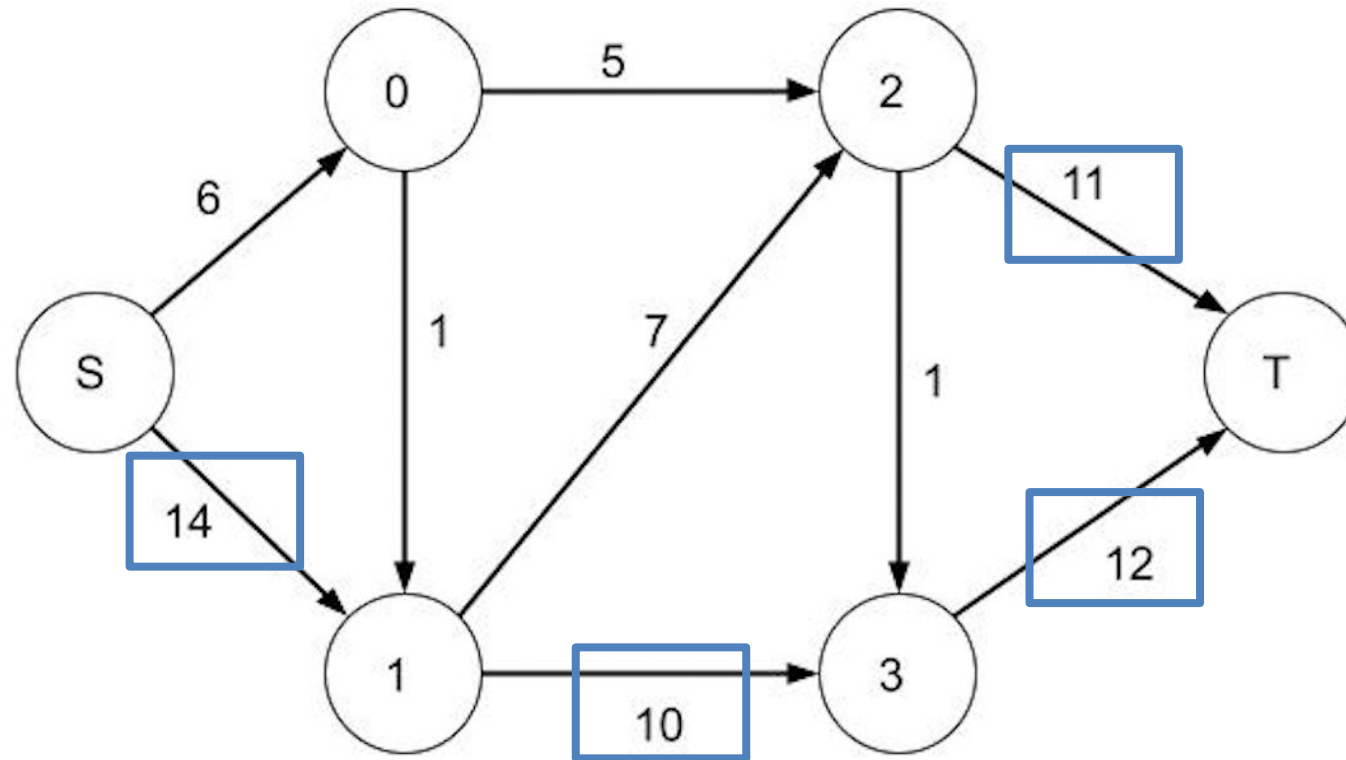
Using capacity scaling, calculate the max-flow of the following given network:



Solution

max capacity out of S = 14

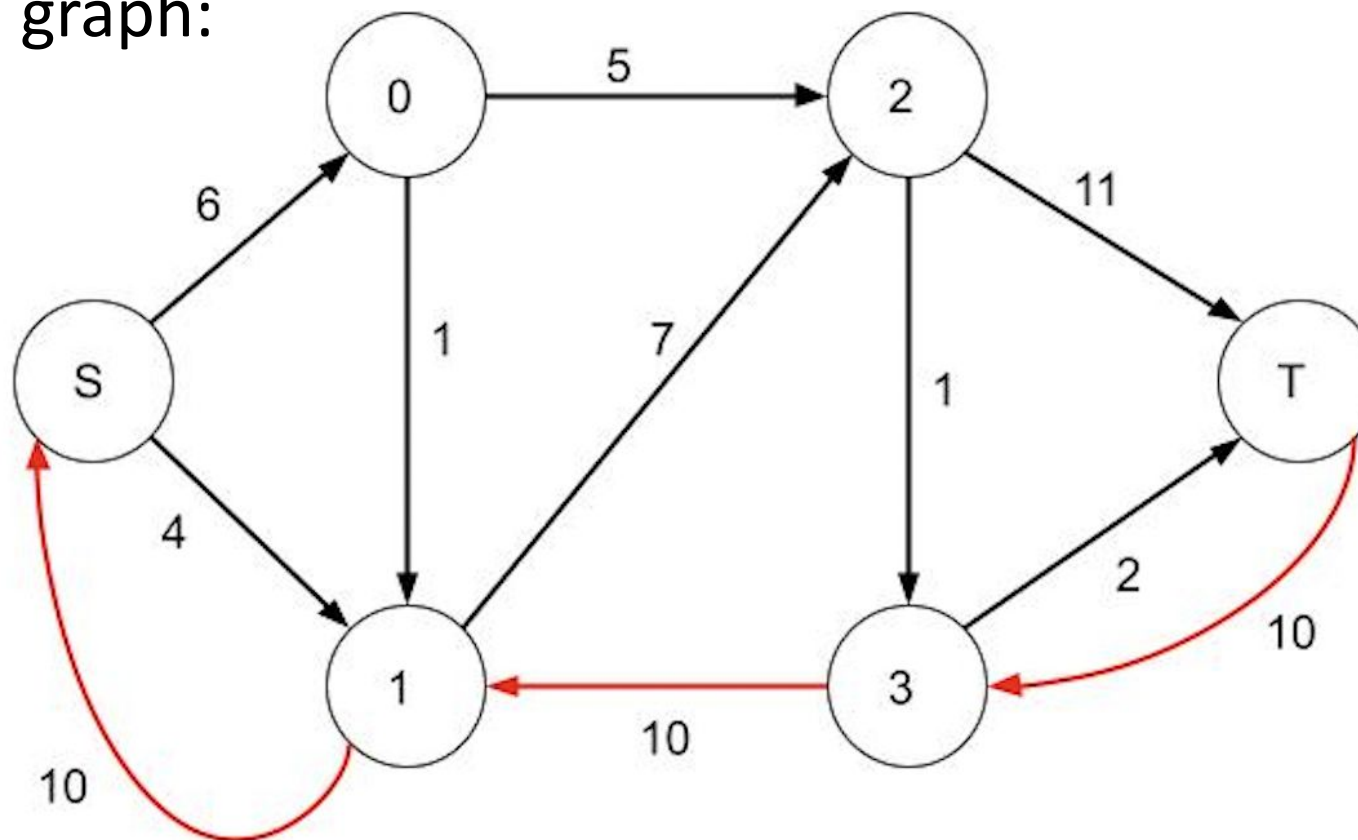
At $\Delta = 8$ phase, available edges:



Solution cont'd

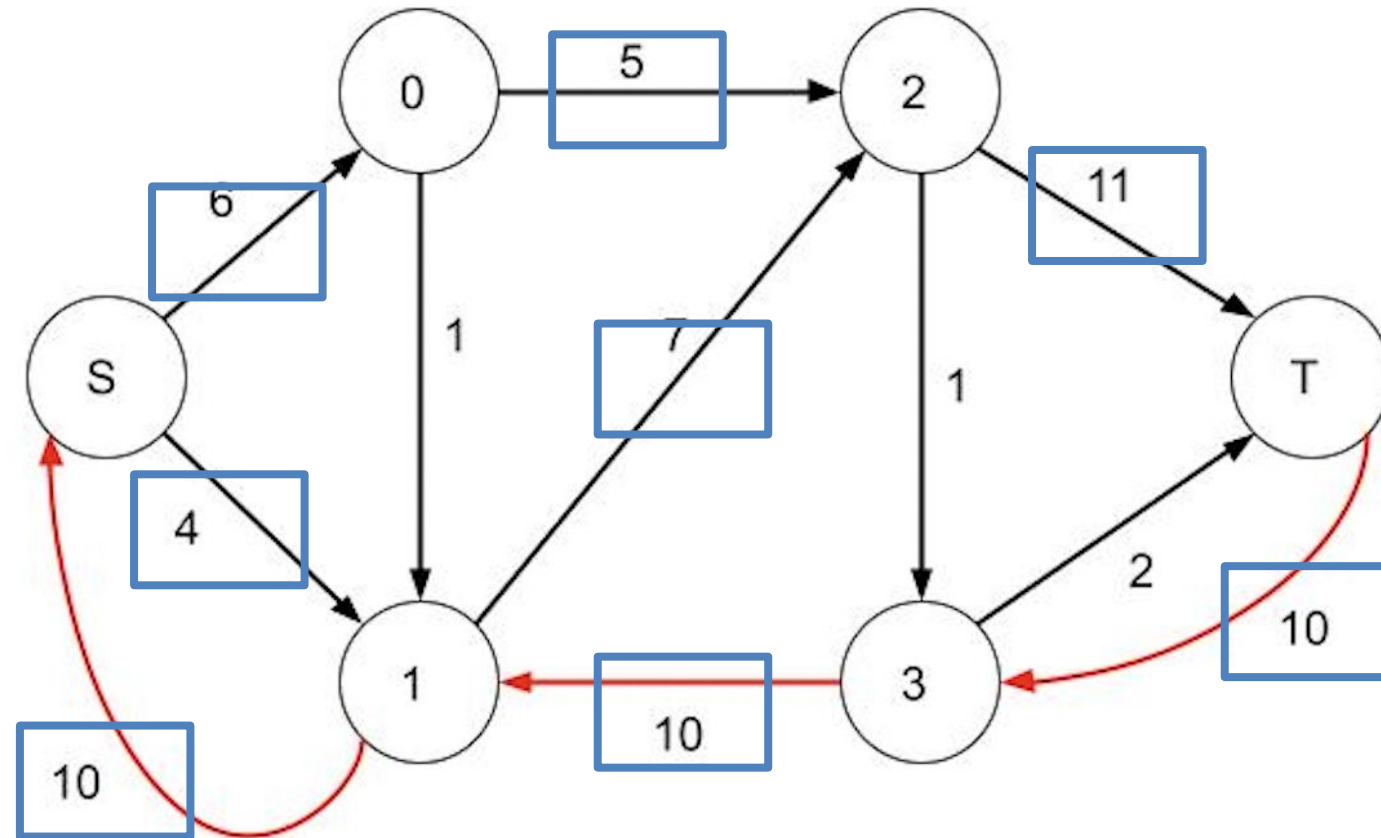
At $\Delta = 8$ phase, we have 1 augmenting path (**S-1-3-T**, bottleneck=10).

Updated residual graph:



Solution cont'd

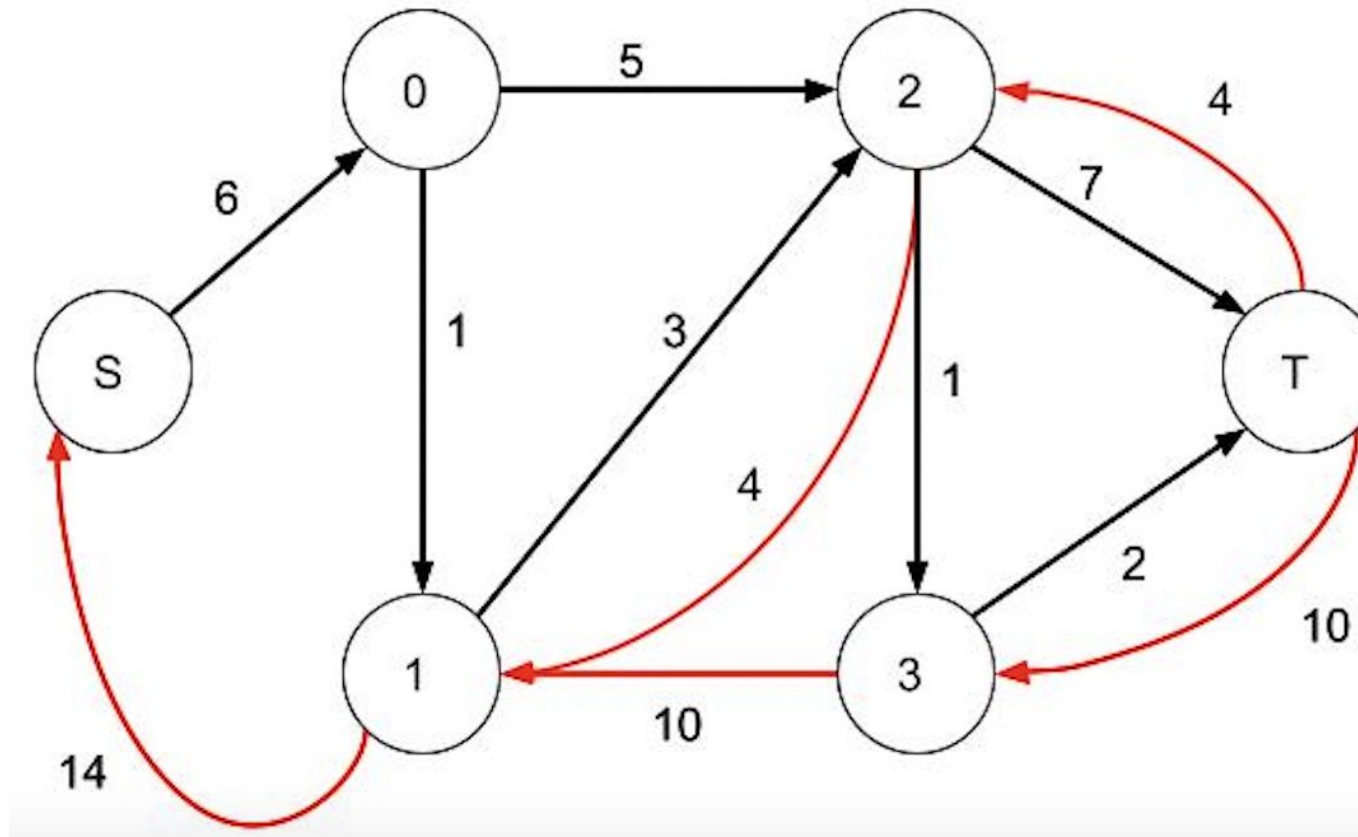
At $\Delta = 4$ phase, available edges:



Solution cont'd

At $\Delta = 4$ phase, we have 2 Augmenting Paths

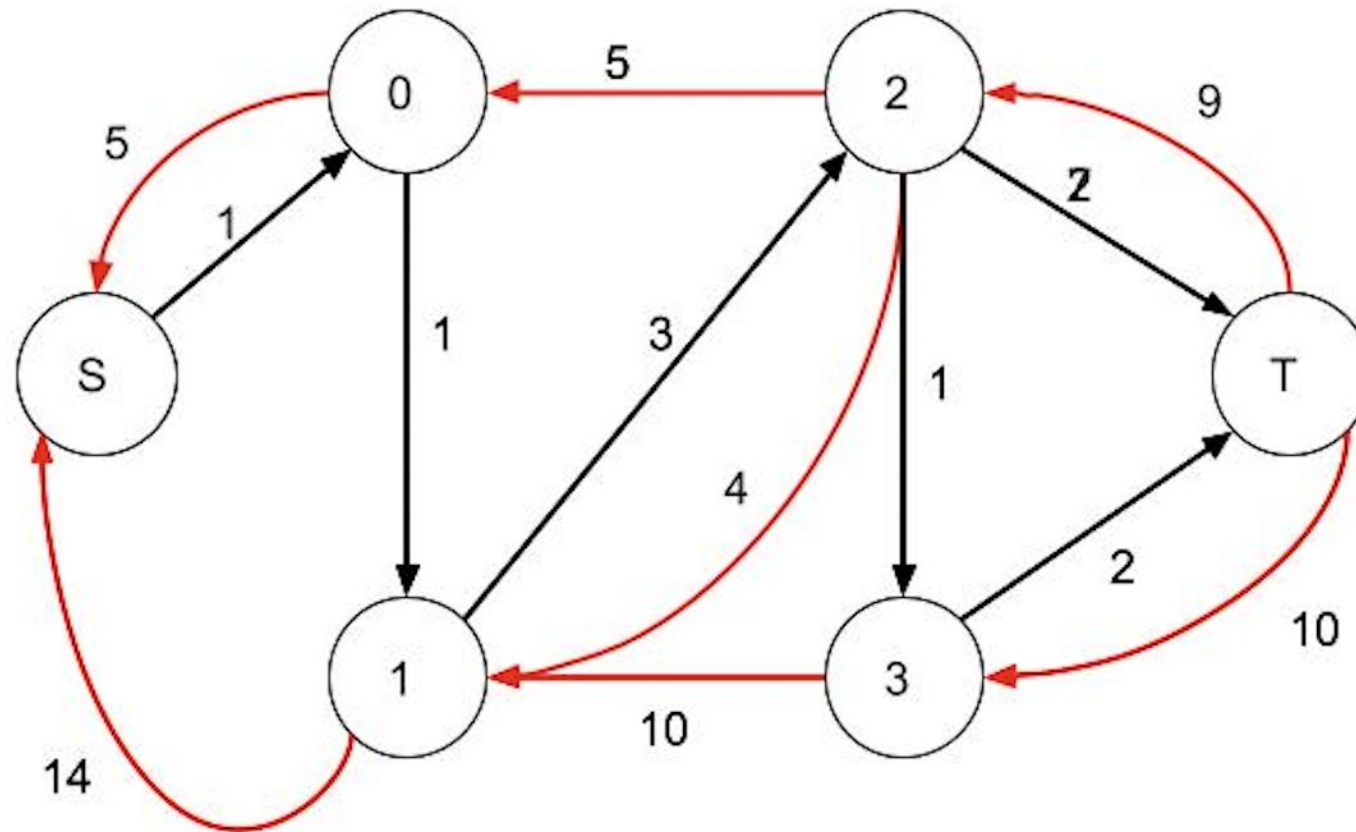
Residual graph after 1st path (**S-1-2-T**, bottleneck=4):



Solution cont'd

At $\Delta = 4$ phase, we have 2 Augmenting Paths

Residual graph after 2nd path (**S-0-2-T**, bottleneck=5):

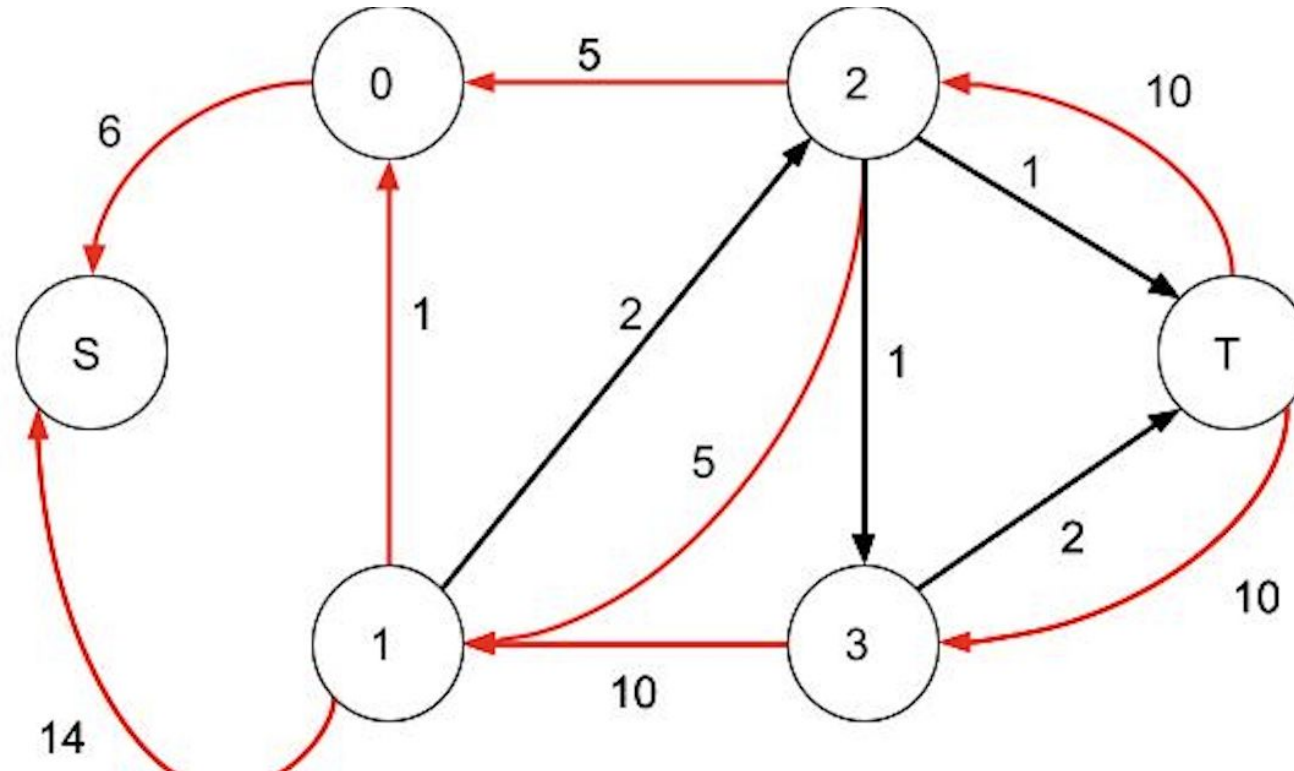


Solution cont'd

At $\Delta = 2$ phase, No possible Augmenting Path

At $\Delta = 1$ phase, 1 Augmenting path possible (**S-0-1-2-T**, bottleneck=1)

Final residual graph:



Algorithm Terminates, Max-Flow = 20

Question 2 - SPY Detection

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, and then move to a neighboring Y. (They can move north, east, south or west to get to a neighbor) Give an efficient network flow-based algorithm to find the largest number of disjoint SPY's.

*Note: We are only looking for the largest **number** of SPY's, not the actual location of the words. Proof of correctness is left as an exercise.*

Example:

Y	S	S	P
P	S	P	Y
S	S	Y	P
Y	S	P	S

Possible
optimal sols:

Y	S	S	P
P	S	P	Y
S	S	Y	P
Y	S	P	S

Y	S	S	P
P	S	P	Y
S	S	Y	P
Y	S	P	S

Solution: SPY Detection

We construct a Flow Network as follows, with **all edges having capacity 1**:

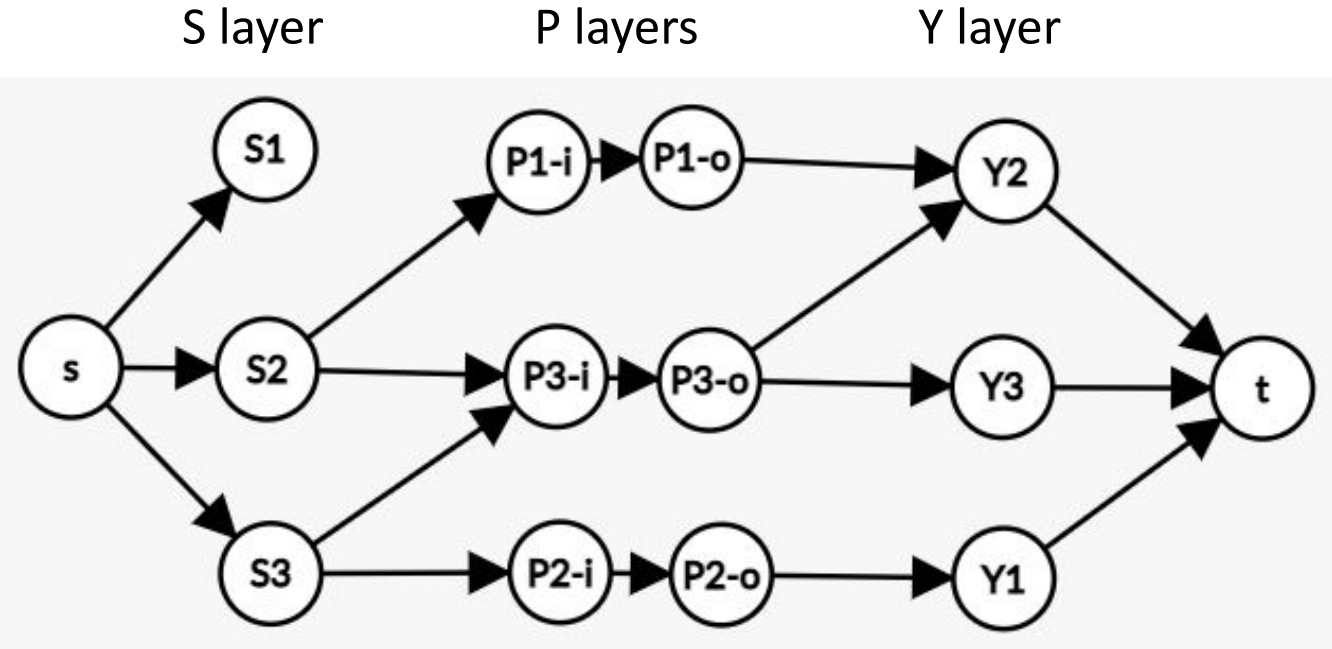
1. Create one layer of nodes for all the S's in the grid. Connect this layer to a source node s , directing edges from s to each S node.
2. Create **two layers of nodes** for all the P's in the grid. Connect the first layer to the S's based on whether they are adjacent in the grid, directing edges from S's to P's. Also, connect the first layer to the second layer by connecting the nodes that correspond to the same location. *(In other words, we are representing **each P as an edge with capacity 1**. This is similar to how we solved the node disjoint paths problem.)*
3. Create a layer of nodes for all the Y's in the grid. Connect these to a sink node t , directing edges from each Y node to t . Also, connect them to the second layer of P's based on whether the P and Y are adjacent in the grid, directing edges from P's to Y's.

*Note that we don't need to represent nodes S or Y with edges, since there are edges of capacity 1 going into S's and edges of capacity 1 leaving Y's **which will force these letters to be picked only once**.*

Claim: Value of Max-Flow in this Flow Network will give us the maximum number of disjoint SPYs.

Solution: SPY Detection (example)

Y ₁	S ₁	S ₂	P ₁
P ₂	S ₃	P ₃	Y ₂
S ₄	S ₅	Y ₃	P ₄
Y ₄	S ₆	P ₅	S ₇



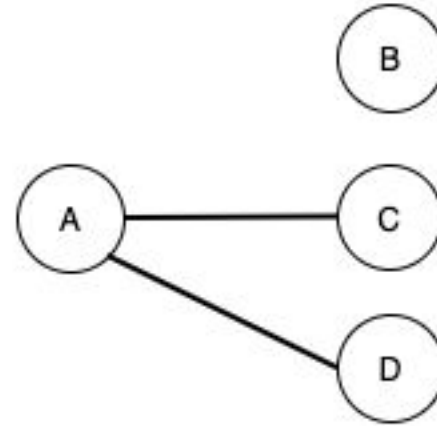
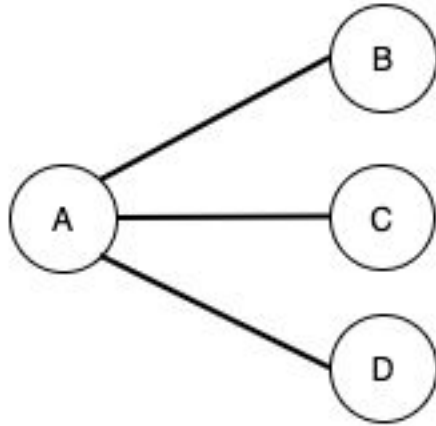
Question 3 - Edge Connectivity

The edge connectivity k of an undirected graph $G = (V, E)$ is the minimum number of edges that must be removed to disconnect the graph (or minimum number of edges that must be removed to split graph into 2 sub graphs). For example, the edge connectivity of a tree is 1.

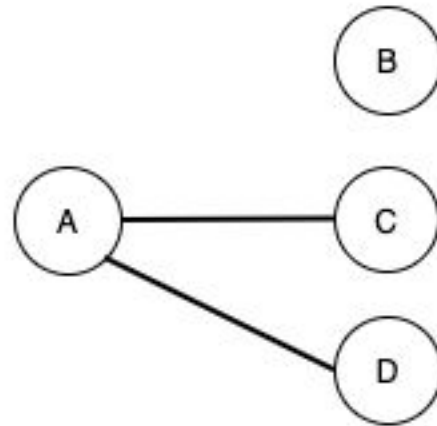
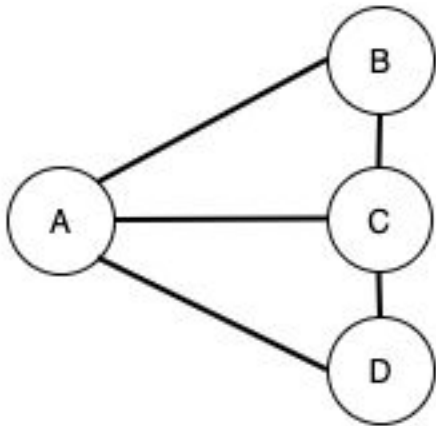
- (a) Show how the edge connectivity k of an undirected graph G with n vertices and m edges can be determined efficiently by running a max-flow algorithm.
- (b) Describe how many max-flow computations will be required to solve edge connectivity.

Example: Edge Connectivity

$k = 1$



$k = 2$



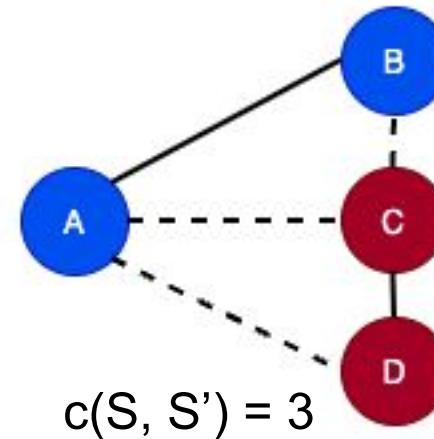
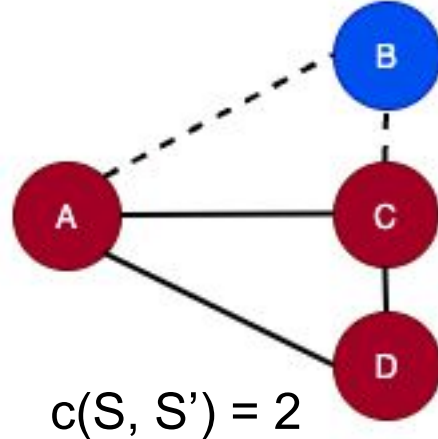
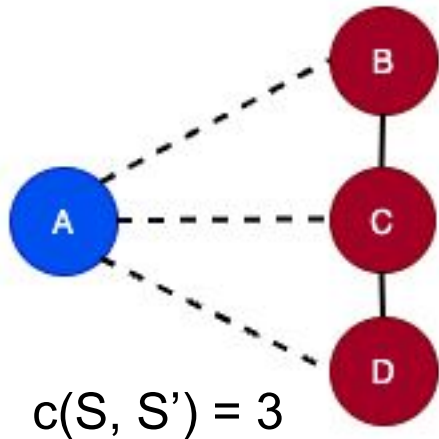
Solution: Edge Connectivity

Let V be the set of n vertices.

For a cut (S, S') , let its capacity $c(S, S')$ denote the number of edges crossing the cut.

By definition, the edge connectivity, $k = \min_{S \subset V} \{ c(S, S') \}$.

(Meaning that the minimum number of edges required to disconnect the graph is equal to the minimum cut capacity)



$$k = \min \{ 3, 2, 3, \dots \} \\ = 2$$

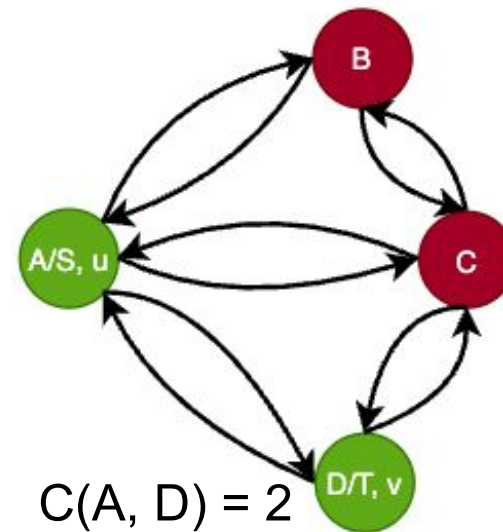
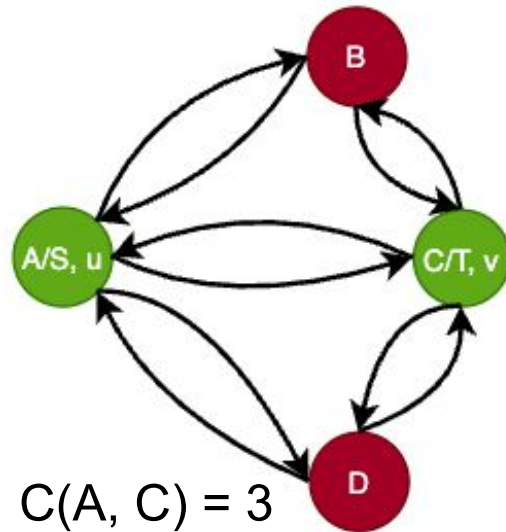
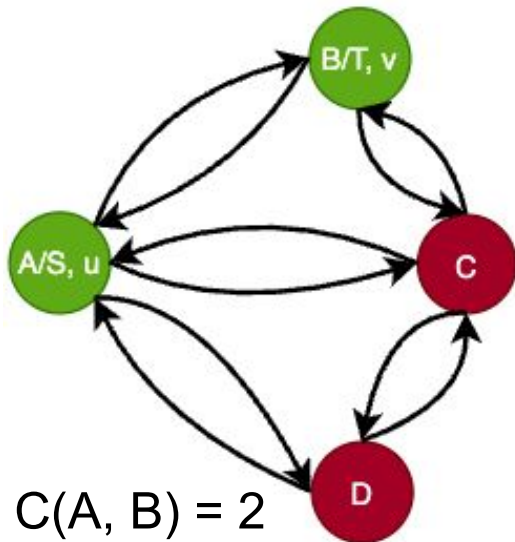
Solution: Edge Connectivity (cont'd)

Construct a directed graph G^* , from G by replacing each edge (u, v) in G by two directed edge (u, v) and (v, u) in G^* with capacity 1.

Now in G^* , fix a vertex $u \in V$. For every cut (S, S') in G^* , there is a vertex $v \in V$ such that u and v are on either side of the cut.

Let $C(u, v)$ denote the value of the min u - v cut.

Thus, $k = \min\{C(u, v)\}$ for $v \in V, v \neq u$



$$k = \min\{2, 3, 2\} = 2$$

Solution: Edge Connectivity (cont'd)

To compute $C(u, v)$ where $u \neq v$, simply compute the max flow from u to v . (Meaning - To compute the value of min cut of u - v , calculate the max flow taking u as the source and v as the sink vertex in G^* . This follows from the max-flow min-cut theorem.)

(b) Also, there are a total of n vertices, and we fix a vertex u . Hence, we need to run max-flow for all the remaining vertices (except u) as the sink. Therefore, there will be $(n-1)$ max-flow computations.



Circulation

Parth Goel

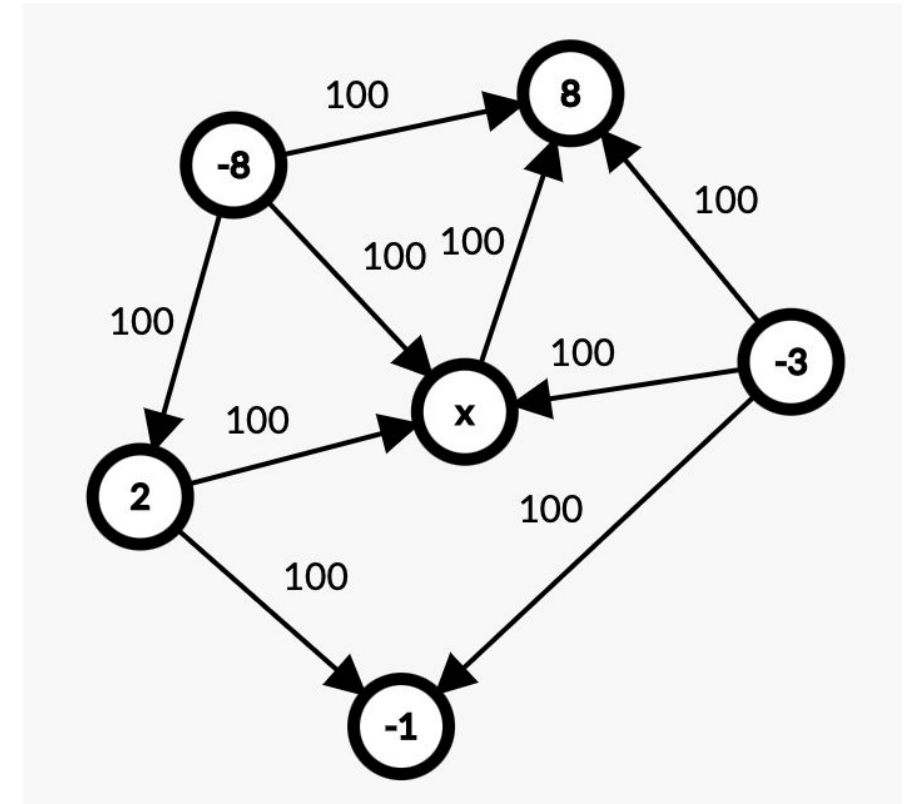
MCQ

What should be the value of x to ensure that a feasible circulation exists?

- a. 2
- b. 3
- c. 1
- d. No such value of x exists
- e. Cannot be determined

Answer: d

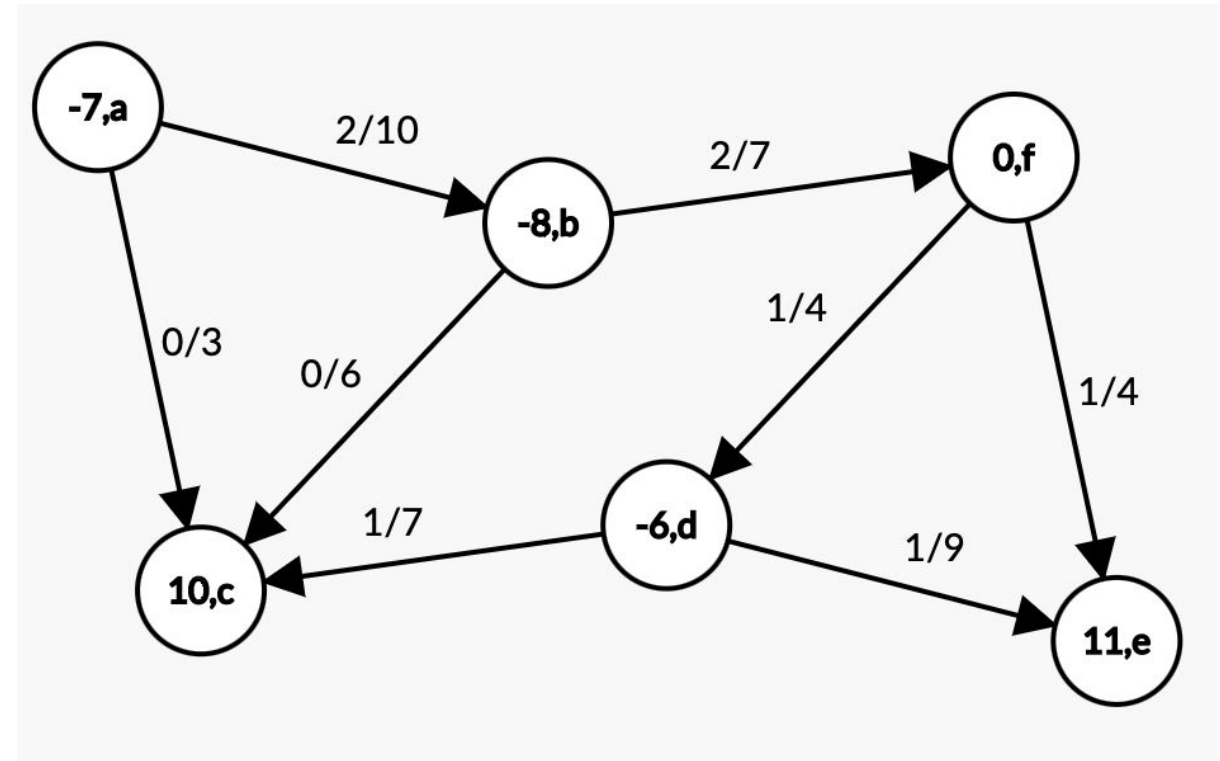
Reason: The node with demand value -1 has no outgoing edge, so can ever be satisfied



Circulation with Demands & Lower bounds

The adjacent graph G is an instance of a circulation problem with lower bounds and demands. The edge weights represent lower bounds/capacities and the node weights represent demands. A negative demand implies source.

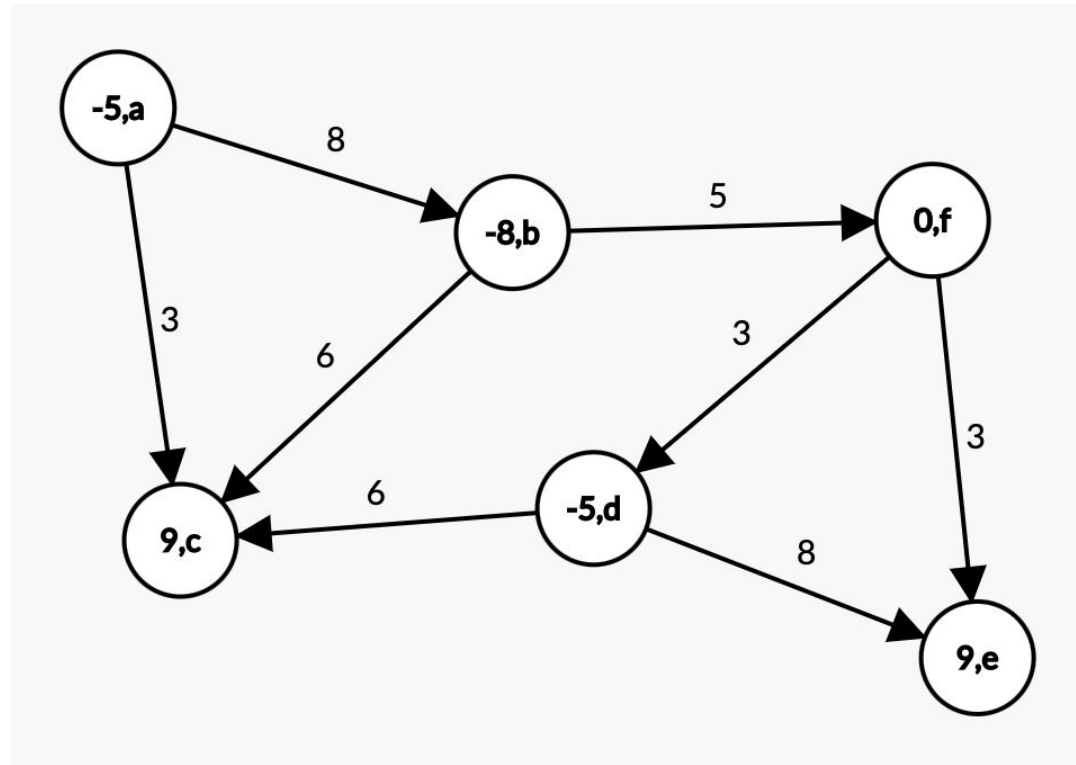
Check if there is a feasible circulation.



Step 1

Push flow f_0 through G where $f_0(e) = l_e$

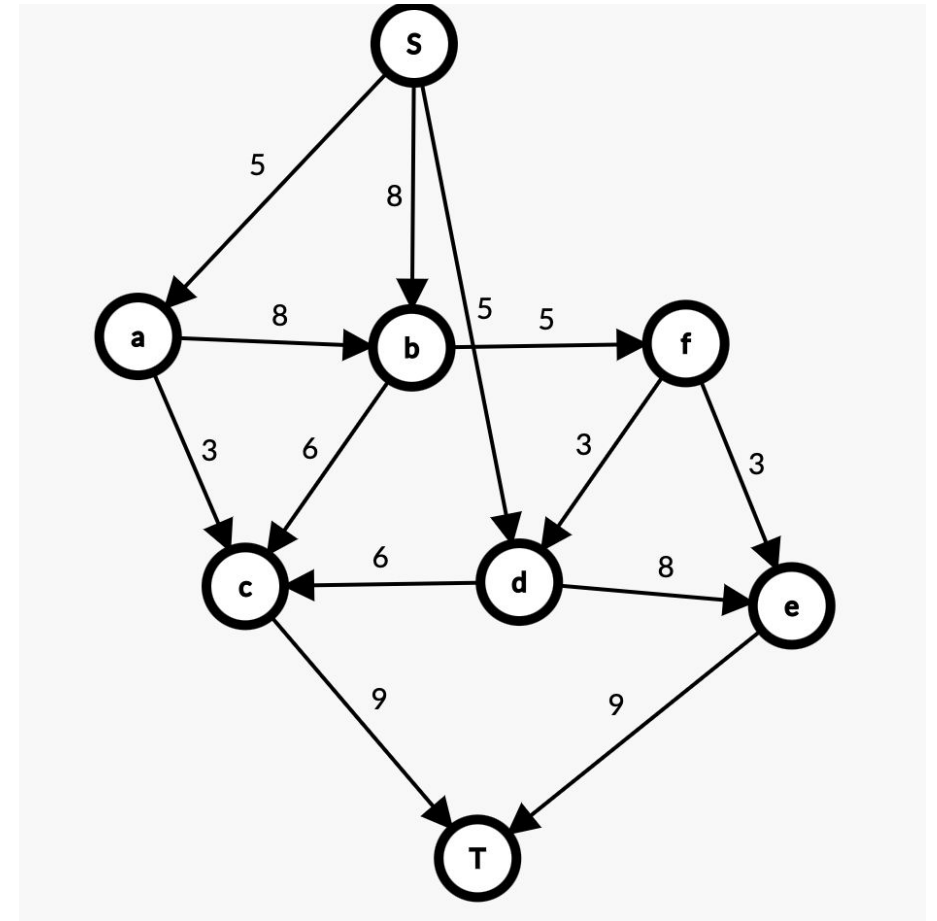
G' is constructed with $c'_e = c_e - l_e$ & $d'_v = d_v - L_v$ where L_v is the imbalance at node v due to flow f_0



Step 2

Create Sink and Source.

Connect source to nodes with negative demand values. Connect nodes with positive demand values to sink.



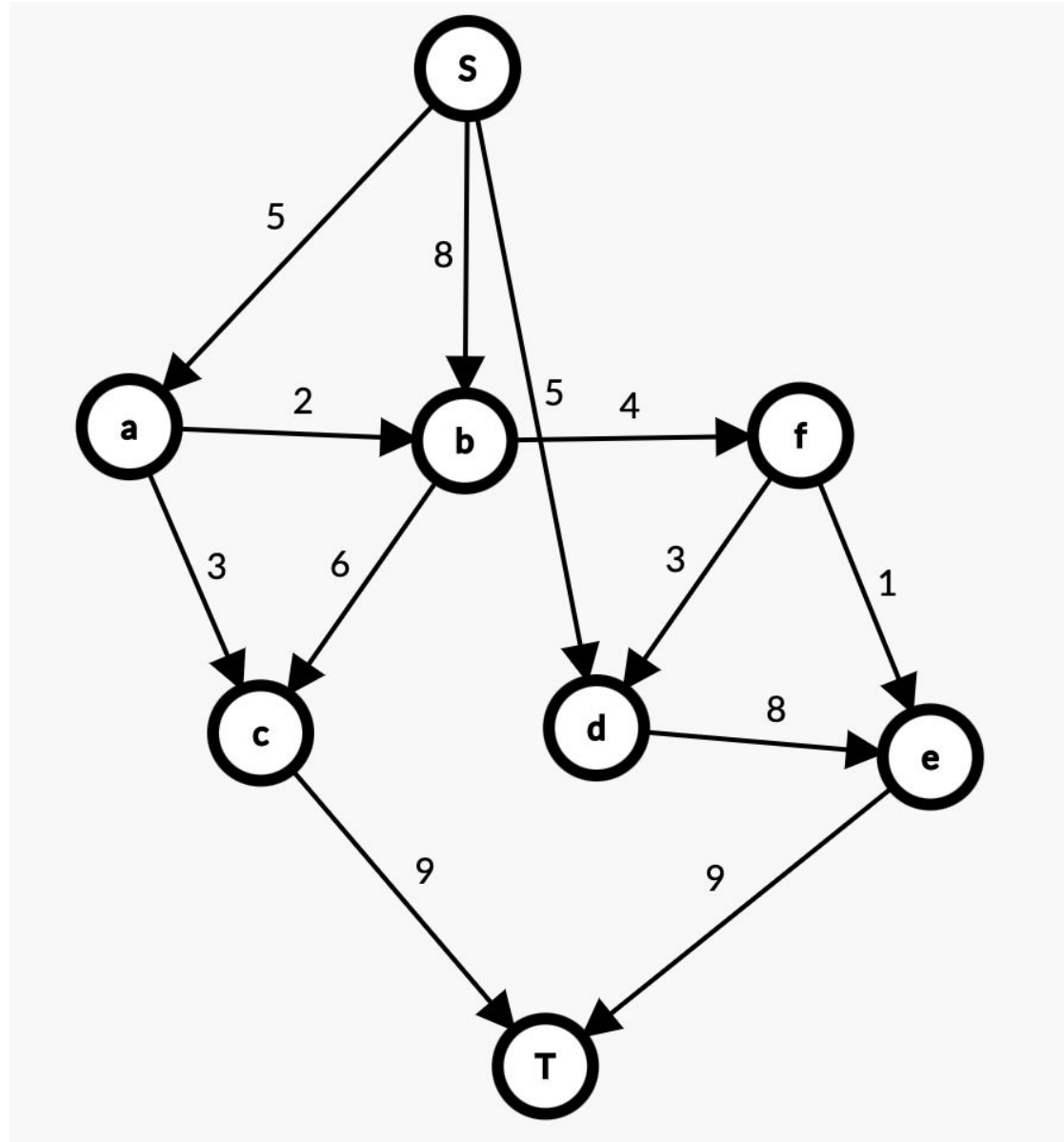
Step 3

Find max flow (f1)

Value of Max flow = 18

D after step 1 = 18

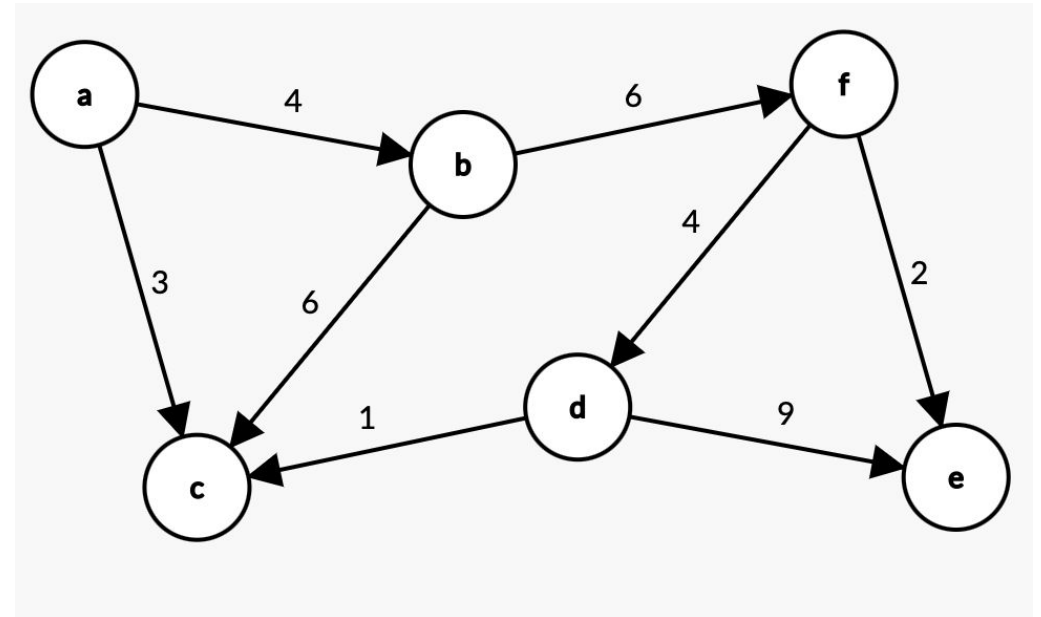
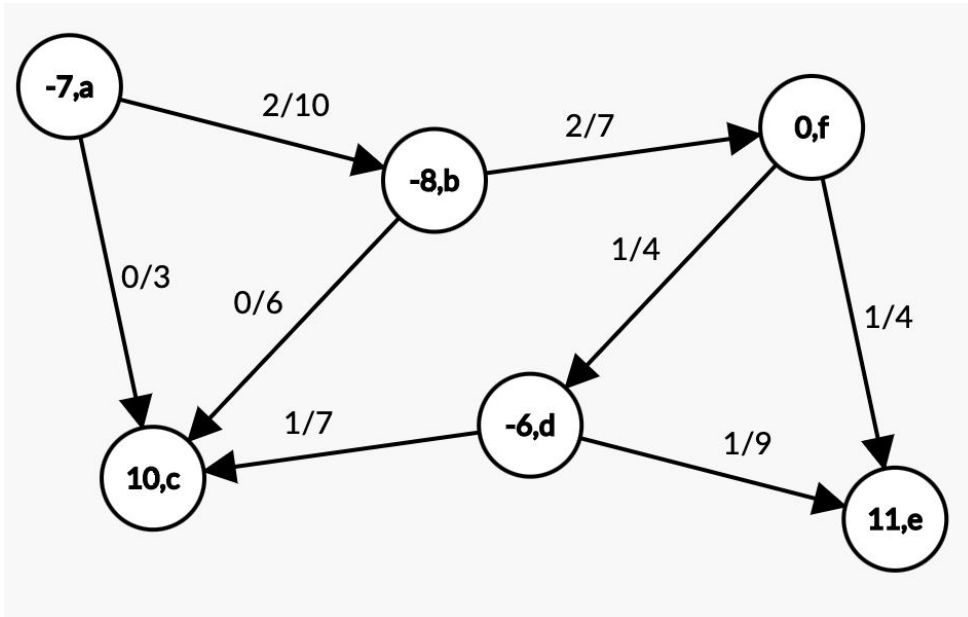
(Sum of positive demand values)



Step 4

We can see in our previous example that feasible circulation exists.

Final flow ($f_0 + f_1$):



Advertisement Policy

You are the manager of a Social media site. The site has contracts with m different advertisers, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i th advertiser looks like:

- For a subset $X_i \subseteq \{G_1, \dots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i .
- For a number r_i , advertiser i wants its ads shown to at least r_i users each minute.

There are a total of k demographic groups and these groups can overlap. Example, a group with people over the age of 30 can intersect with people living in LA. Let us say there are n users that visit the site at a given minute. The problem is: is there a way to show at most a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute?

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

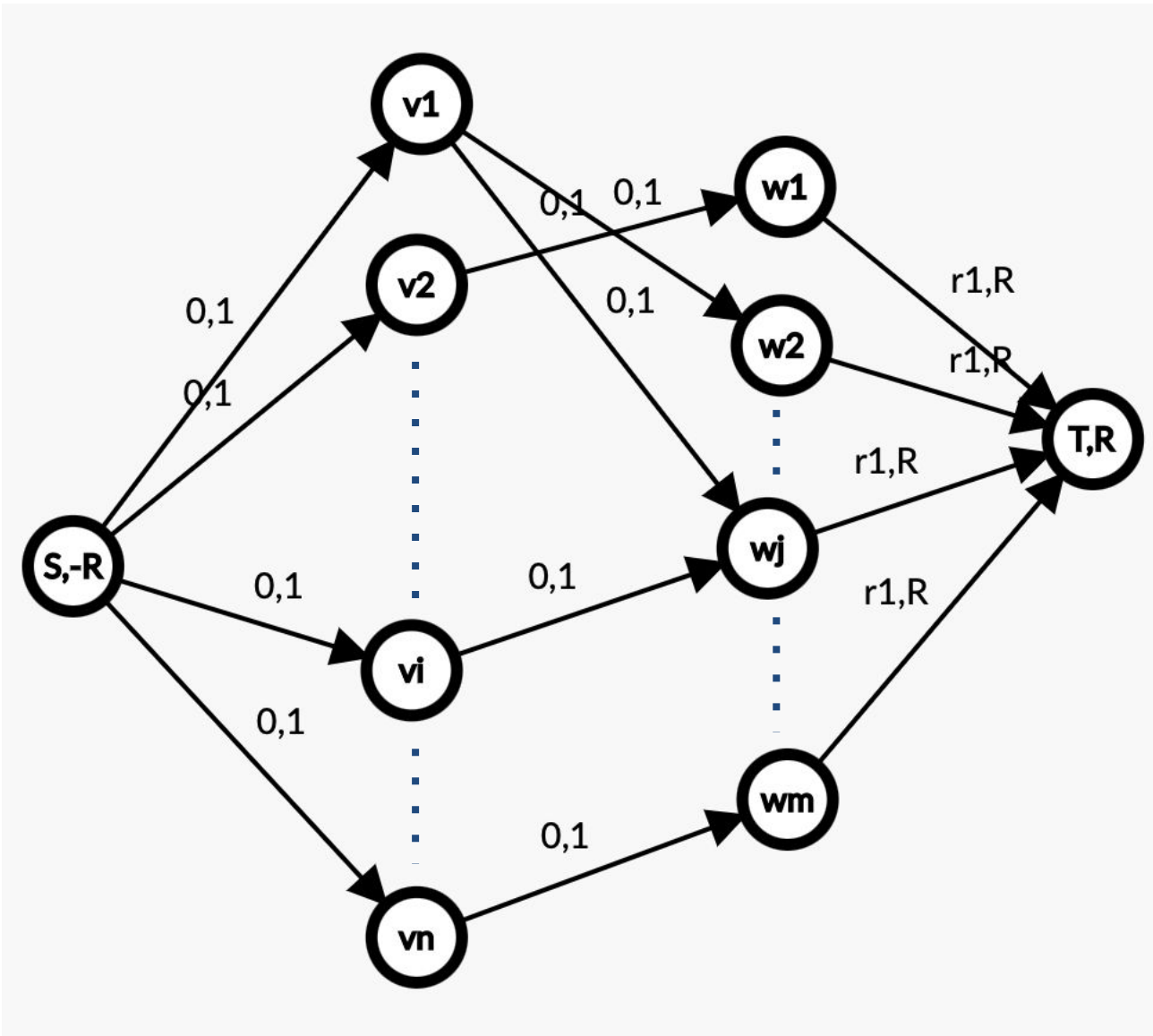
Solution

We reduce this to a circulation with demands and lower bounds problem.

We construct the following graph G :

- There is a source s , vertices v_1, \dots, v_n for each person vertices w_1, \dots, w_m for each advertiser, and sink t .
- There is an edge of capacity 1 from v_i to each w_j for which person i belongs to a demographic group that advertiser j wants to target.
- There is an edge with a capacity of 1 from s to each v_i ; and for each j , there is an edge with lower bound r_j from w_j to t .
- Finally, the source has a demand of $-\sum_j r_j$, and the sink has a demand of $\sum_j r_j$ (i.e. R).
All other nodes have demand 0.

If we have a feasible solution, we have a good advertisement policy.



Proof

Claim: There is a valid circulation in this graph if and only if there is a way to satisfy all the advertising contracts

Forward Proof: First let's say there is a way to satisfy all the advertising contracts. then we can construct a valid circulation as follows. We place a unit of flow on each edge (v_i, w_j) for which i is shown an ad from advertiser j ; we put a flow on the edge (w_j, t) equal to the number of ads shown from advertiser j ; and we put a unit of flow on each edge (s, v_i) for which person i sees an ad.

Backward Proof: If there is a valid circulation in this graph, then there is an integer circulation. In such a circulation, one unit of flow on the edge (v_i, w_j) means that we show an ad from advertiser j to person i . With this meaning, each advertiser shows their required number of ads to the appropriate people.

THANK YOU