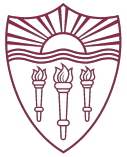


CSCI 570

Exam 2 Review Slides

By: Sajjad Shahabi, Omkar Thakoor

A string is palindromic if it is equal to its reverse, e.g., “radar” and “level” are both palindromic. Design a dynamic programming algorithm for finding the length of the longest contiguous palindromic substring of a given input string of length n.

Example: S = “cabad”, output = 3 (The longest palindromic substring is “aba”)

a) Define the subproblems to be solved

A string is palindromic if it is equal to its reverse, e.g., “radar” and “level” are both palindromic. Design a dynamic programming algorithm for finding the length of the longest contiguous palindromic substring of a given input string of length n.

Example: S = “cabad”, output = 3 (The longest palindromic substring is “aba”)

a) Define the subproblems to be solved

dp(i, j) = if substring S[i, j] (including both i and j indices) is palindromic

b) Write a recurrence relation for the subproblem.

b) Write a recurrence relation for the subproblem.

```
if(i == j):                                          //Substring of length 1 is palindromic
    dp(i,j) = True
Else If (j - i == 1):                                //Substring of length 2 is palindromic if both
     dp(i, j) = s[i] == s[j]                          chars are the same
Else:
    if(s[i] == s[j] && dp(i+1, j-1) = true):    //Substring of length > 2 is palindromic if
                dp(i, j) = true                  the first char of the dp(i, j) equals to the last
        Else:                                    character and the middle part is palindromic
                dp(i, j) = false
```

c) Using the recurrence formula in b, write pseudocode using iteration to compute the longest palindromic substring.

Make sure you specify

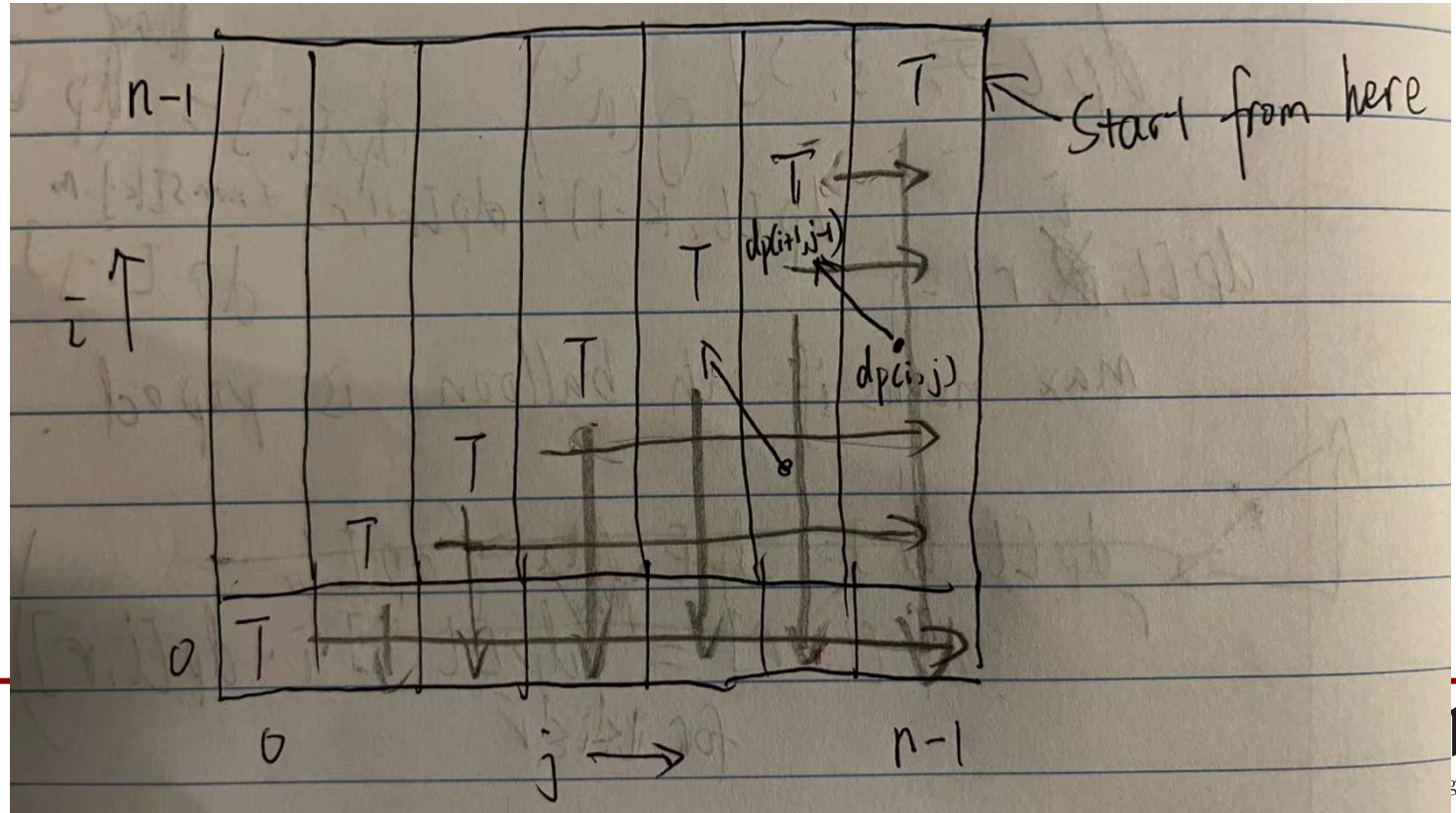
- base cases and their values

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.)

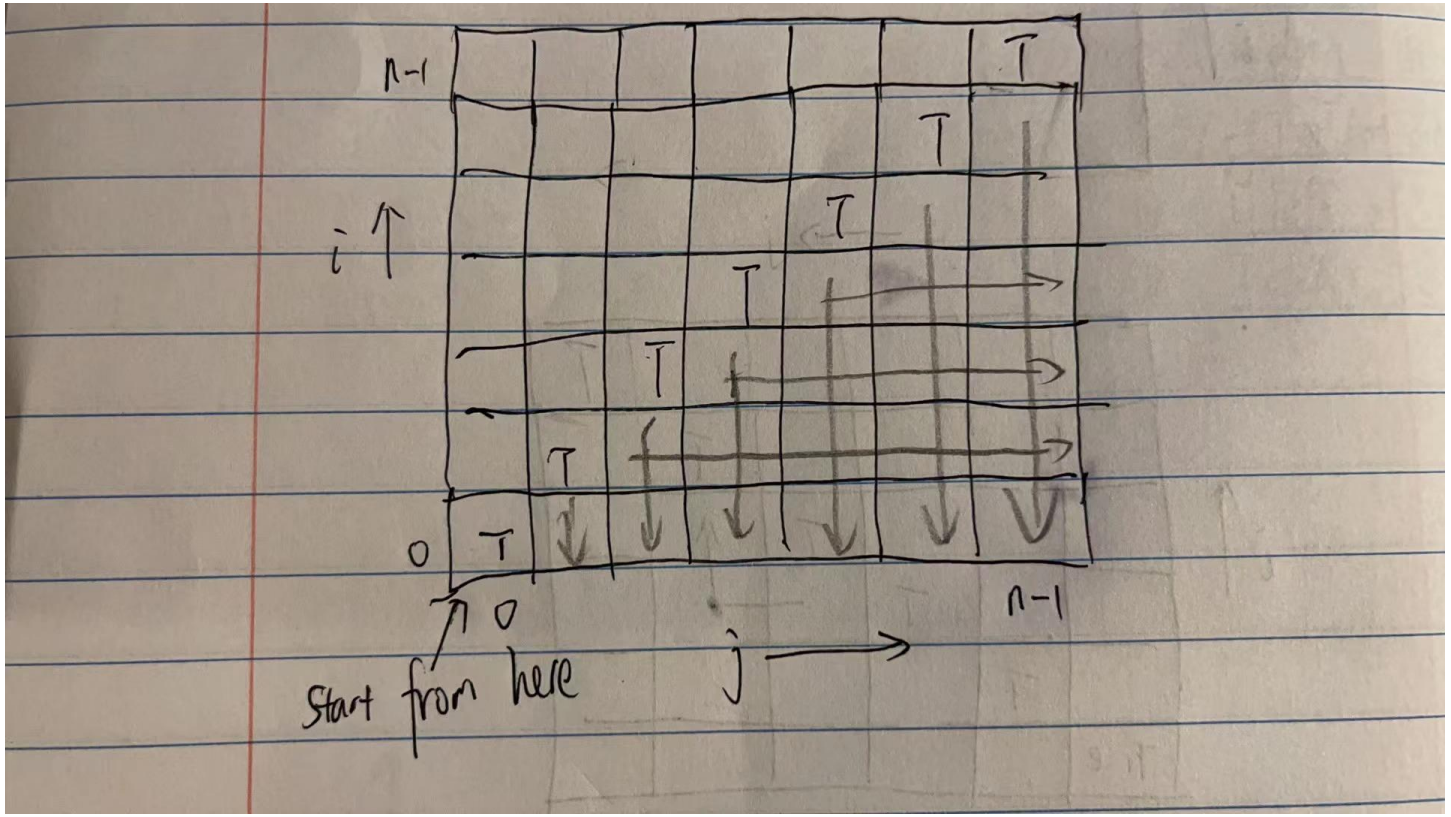
c) Using the recurrence formula in b, write pseudocode using iteration to compute the longest palindromic substring.

Since dp(i , j) depends on dp(i+1, j-1), the i outer loop needs to go from higher to lower and the j outer loop need to go from lower to higher, j cannot be smaller than i

(1) We can start from i = n -1, filling the table from left to right in each row (increasing j) and work our way down (decreasing i)



Another way is to start from j = 0 and filling the table downward in each column (decreasing i) and move from left to right (increasing j)



c) Using the recurrence formula in b, write pseudocode using iteration to compute the longest palindromic substring.

Make sure you specify

- base cases and their values

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.)

c) Using the recurrence formula in b, write pseudocode using iteration to compute the longest palindromic substring.

```
Initialization:
Let dp[0, …, n-1][0, …, n-1] be array where all values are set to false

For i = 0 to n-1 do
      dp[i][i] = true;

result = “”
For i = n-1 to 0 do
      For j = i+1 to n-1 do
            if(s[i] == s[j]) then
                  If j - i == 1 or opt(i+1, j-1) then
                        opt[i][j] = true;
                        if(j - i + 1 > result.length) then
                              result = s[i, j]
      End for
End for
Return result   //result will hold the optimal solution
```

d) What is the run time complexity of your solution? Is it efficient?

d) What is the run time complexity of your solution? Is it efficient?

O(n^2), it’s polynomial therefore efficient!

Fixing Lights

After a snowstorm, all the traffic lights went out on Main Street. Main Street is a linear street with n traffic lights. Because traffic lights are interconnected, fixing one traffic light will automatically fix its adjacent traffic lights (traffic lights adjacent to light i are lights i-1 and i+1 if they exist). However, fixing an already fixed light will do nothing. You are given a list of the times needed (in hours) to fix each light: $T = t_1, t_2, t_3, ..., t_n$. You are also given a list of pays for each light that you work on: $P = p_1, p_2, p_3, ..., p_n$. Note that you are only given pay for the lights that you have worked on, not the adjacent lights that are fixed automatically. You are given k hours to work on the traffic lights. Find the maximum pay possible.

Fixing Lights

An example to this question:

Total 9 lights: L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_9

Time needed to fix: T = 1, 4, 1, 4, 1, 1, 5, 2, 2

Pays: P = 250, 130, 100, 60, 50, 70, 270, 160, 140

You are given k = 6 hours

Solution: You should fix L_1, L_3, L_6, L_8 to earn a maximum pay of 580.

Fixing Lights

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

Fixing Lights

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

Solution: OPT[i][j] = Max pay value for lights 1...i given a total of j hours to work on traffic lights

- alternate solution can consider lights i to n (recurrence and pseudo-code changes accordingly)

Fixing Lights

b) Write a recurrence relation for the subproblems

Fixing Lights

b) Write a recurrence relation for the subproblems

Solution: Very similar to the 0-1 knapsack problem:

if we can fix light i within the j hours left (i.e. $t_i \leq j$):

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

else:

$OPT[i][j] = OPT[i - 1][j]$ (Okay to exclude this case here IF correctly covered in the pseudo-code via base cases or otherwise)

Fixing Lights

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum possible pay.

Make sure you specify

- base cases and their values

- where the final answer can be found (e.g. opt(n), or opt(0,n), etc.)

Fixing Lights

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum possible pay.

Solution:

- OPT[0][j] = 0 for all j= 0 ... k (can skip this by adding base cases for OPT[2][j] instead)

- OPT[i][0] = 0 for all i= 0 ... n (this can be excluded as it's handled by the $t_i > j$ case in the loop)

- OPT[1][j] = 0 if $t_1 > j$ otherwise p_1 for all j= 1 ... k

Fixing Lights

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the maximum possible pay.

Solution:

For i from 2 to n:

    For j from 1 to k:

        if t_i <= remaining time j:

            OPT[i][j] = max(OPT[i - 1][j], OPT[i - 2][j - t_i] + p_i)

        else:

            OPT[i][j] = OPT[i - 1][j]

Total_pay = OPT[n][k]

Fixing Lights

d) What is the complexity of your solution? Is this an efficient solution?

Fixing Lights

d) What is the complexity of your solution? Is this an efficient solution?

Solution:

Runs in O(n*k).

No, this is not an efficient algorithm. This is a pseudo-polynomial time solution because the polynomial function n*k contains a numerical value of an input term k

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: Constraints

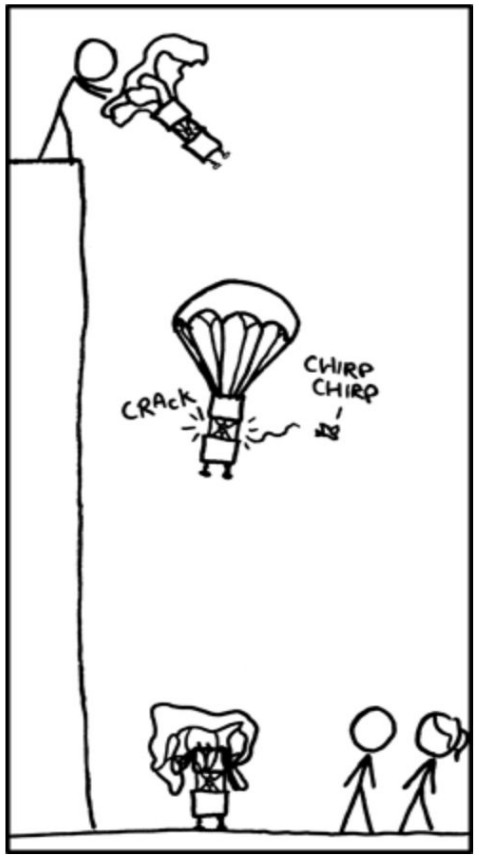
Following are the constraints:

- 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: Example

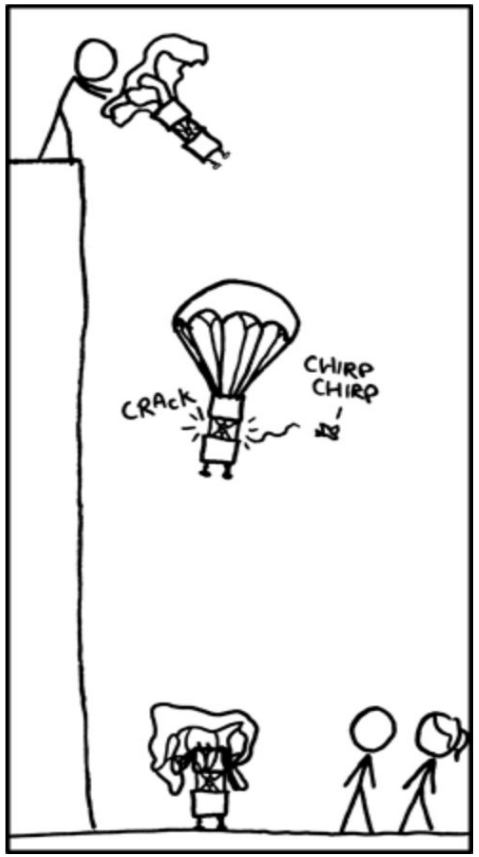
- *For example, if the building has 100 floors (n = 100)*
  - *we have only 1 egg, then we must do 100 trials to find the threshold*
  - *What about 2 eggs?*



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: 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."

Define the subproblem

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

Define the subproblem

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

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

Find the Recurrence for OPT[i, j]

b) Write a recurrence relation for the subproblems

Find the Recurrence for OPT[i, j]

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

Example


eggDrop(9, 3)
minimum of
1 + max(eggdrop(8,2), eggdrop(0,3))
1 + max(eggdrop(7,2), eggdrop(1,3))
1 + max(eggdrop(6,2), eggdrop(2,3))
1 + max(eggdrop(5,2), eggdrop(3,3))
1 + max(eggdrop(4,2), eggdrop(4,3))
1 + max(eggdrop(3,2), eggdrop(5,3))
1 + max(eggdrop(2,2), eggdrop(6,3))
1 + max(eggdrop(1,2), eggdrop(7,3))
1 + max(eggdrop(0,2), eggdrop(8,3))

PseudoCode

c) Using the recurrence formula in part b, write pseudocode using iteration to compute the minimum number of trials.

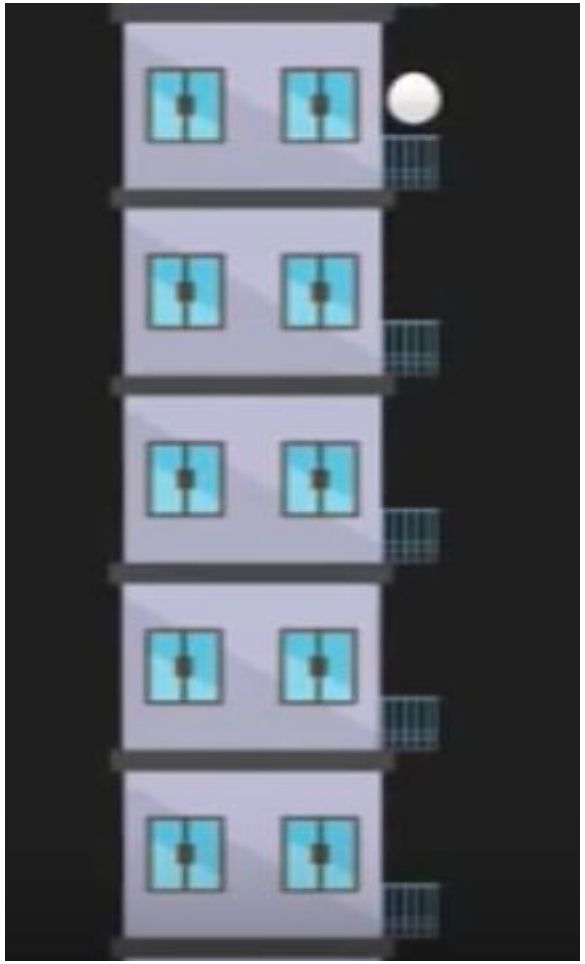
Make sure you specify

- base cases and their values

- where the final answer can be found

Base Cases

- If we 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



PseudoCode

- Base Case:
  - OPT[0, j] = 0; // 0 floor
  - OPT[1, j] = 1; // 1 floor
  - OPT[i, 1] = i; // 1 egg
  - OPT[i, j] = MAX_INTEGER, otherwise
- Iteration:

```
for j = 2 to k:
    for i = 2 to n:
        for x = 1 to i:
            OPT[i, j] = min(OPT[i, j], 1 + max{OPT[x – 1, j - 1], OPT[i – x, j]})
```

- Return 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

d) What is the complexity of your solution? Is this an efficient solution?

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...

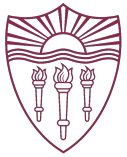
DP: tips and common mistakes

Subproblems: 1) Explain all parameters, 2) no reference to new variables, 3) clearly describe what is computed 4) Use variables different from given input

Recurrence: 1) No self-reference (unlike pseudocode) 2) when min/max/sum over multiple cases, explain the range/constraint properly

Base cases: Wherever recurrence cannot be applied! Decide which and how many accordingly. (no harm in more)

Runtime: Efficient if poly-time (in input **size**), not otherwise!



Flow and Circulation

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 ∈ S and v ∈ V−S. Then increasing the capacity of the edge (u, v) will also increase the maximum flow of G.

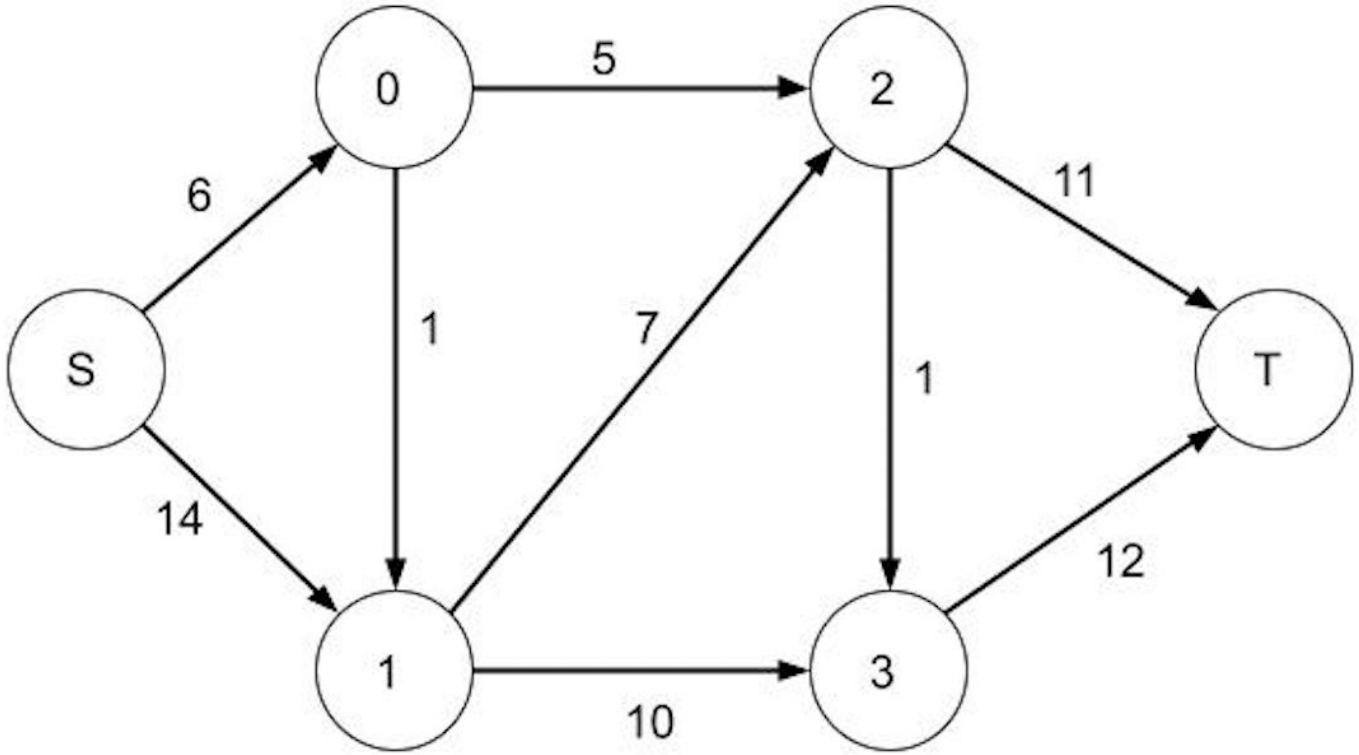
   False!

1. If all of the edge capacities in a network flow graph are an integer multiple of 81, then the value of the maximum flow will be a multiple of 81.

   True!

Question 1 - Scaled Ford-Fulkerson

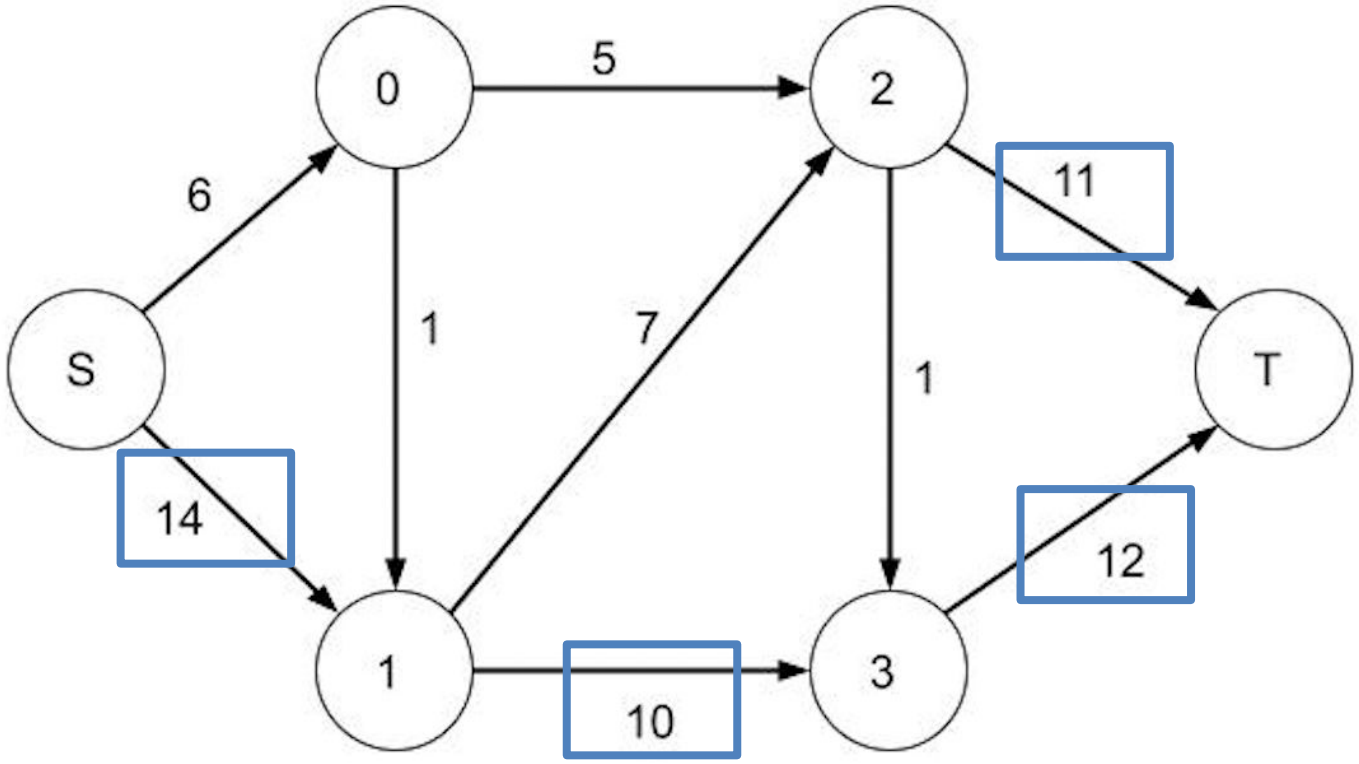
Using capacity scaling, calculate the max-flow of the following given network:



Solution

max capacity out of S = 14

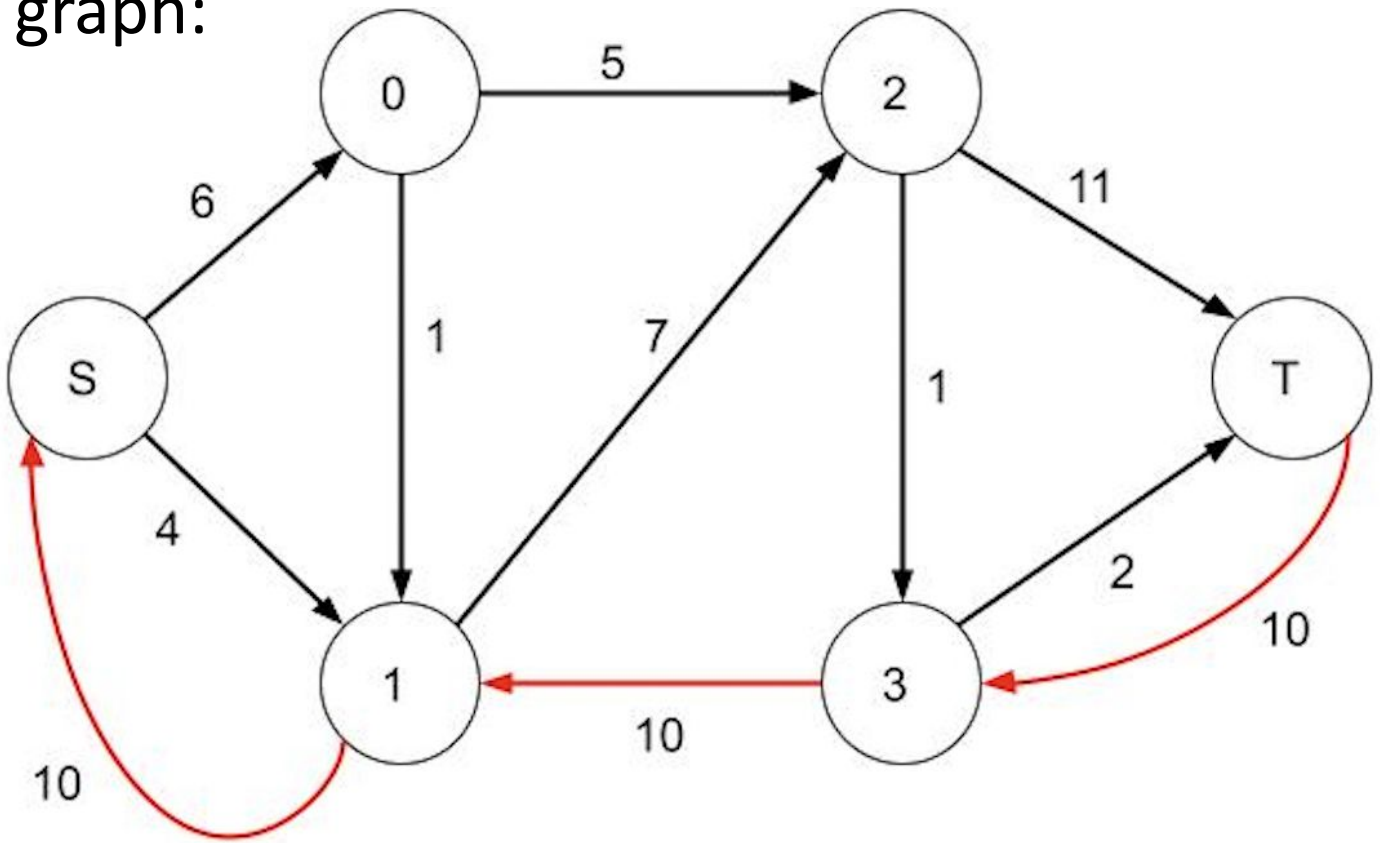
At Δ = 8 phase, available edges:



Solution cont'd

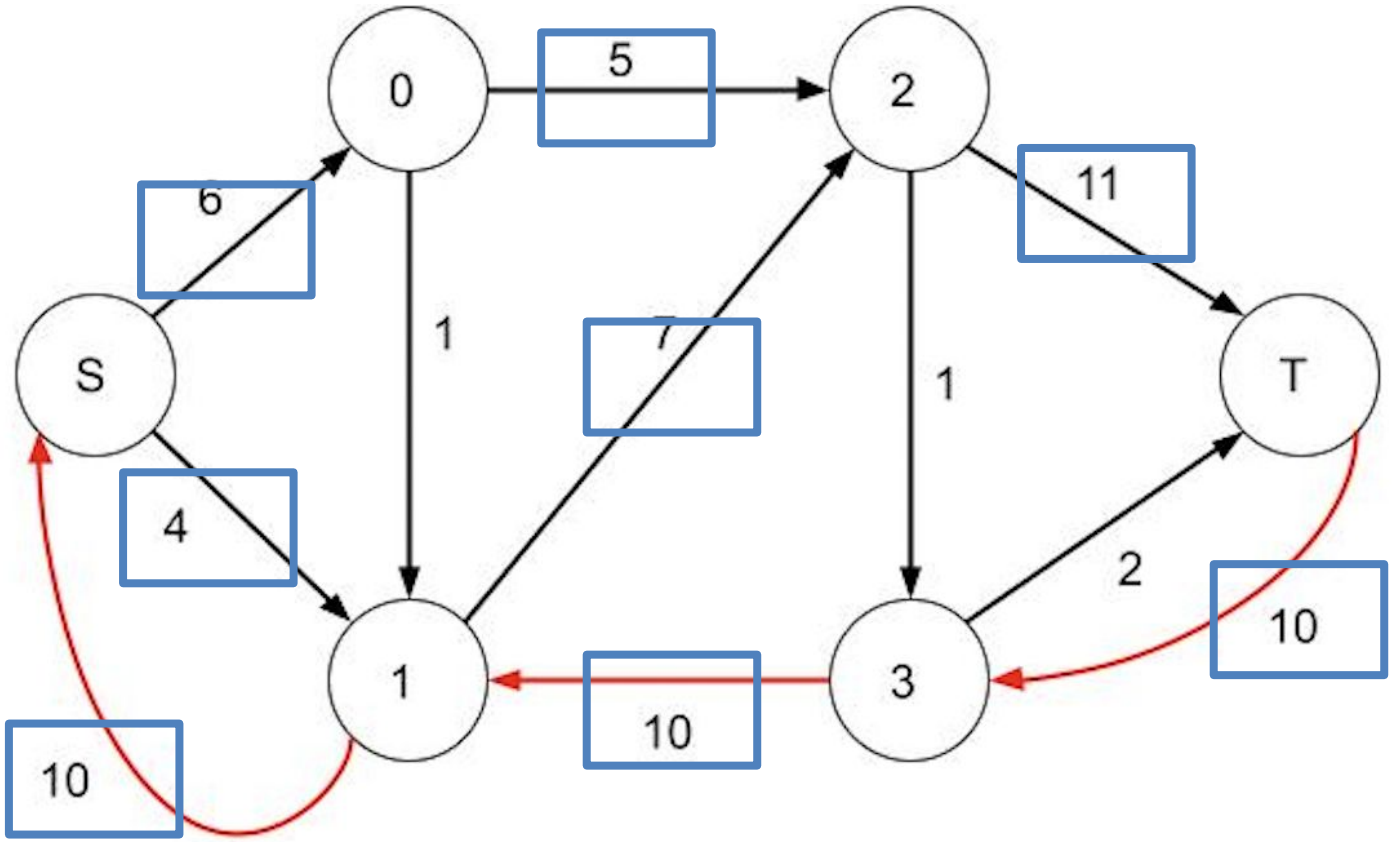
At $\Delta = 8$ phase, we have 1 augmenting path (**S-1-3-T**, bottleneck=10).

Updated residual graph:



Solution cont'd

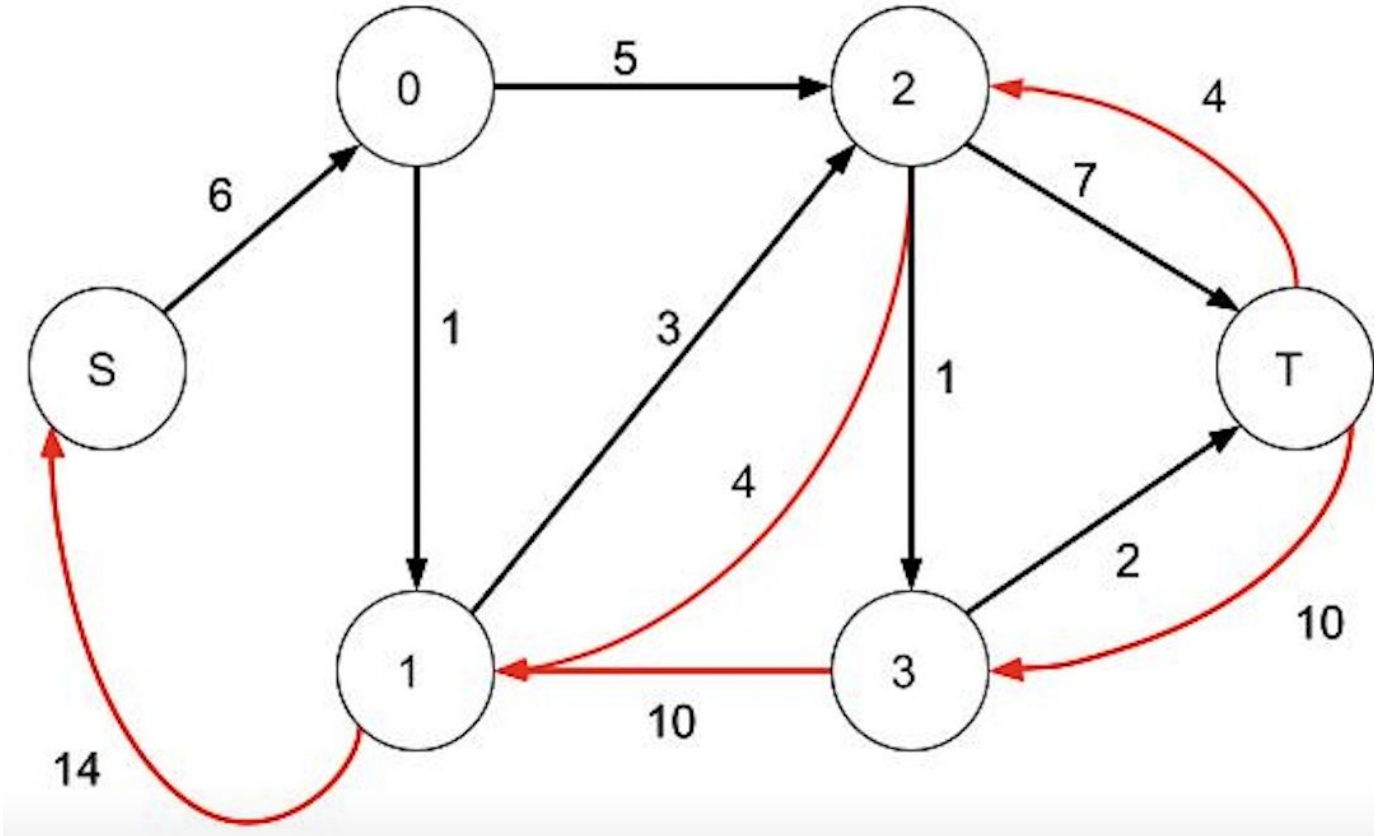
At $\Delta = 4$ phase, available edges:



Solution cont'd

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

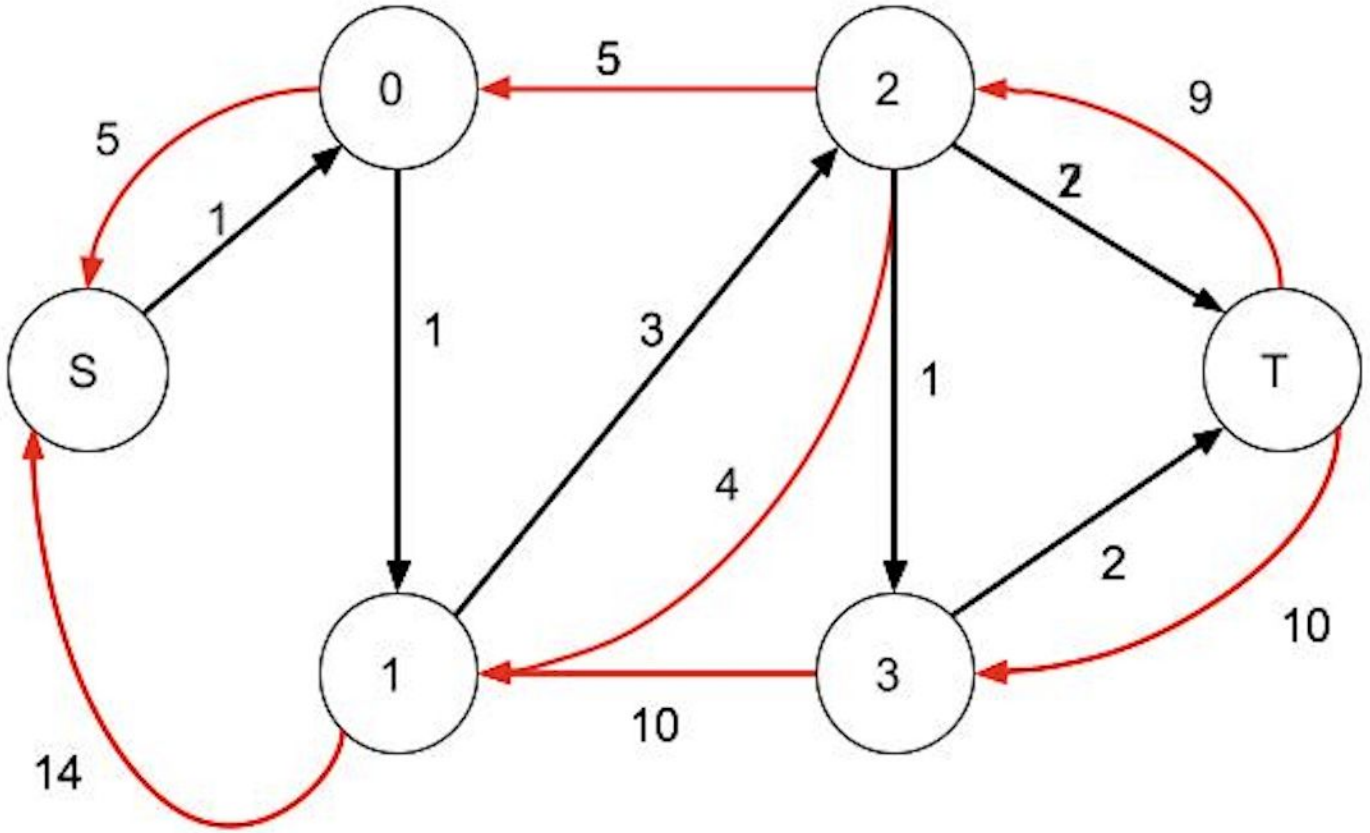
Residual graph after 1st path (**S-1-2-T**, bottleneck=4):



Solution cont'd

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

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

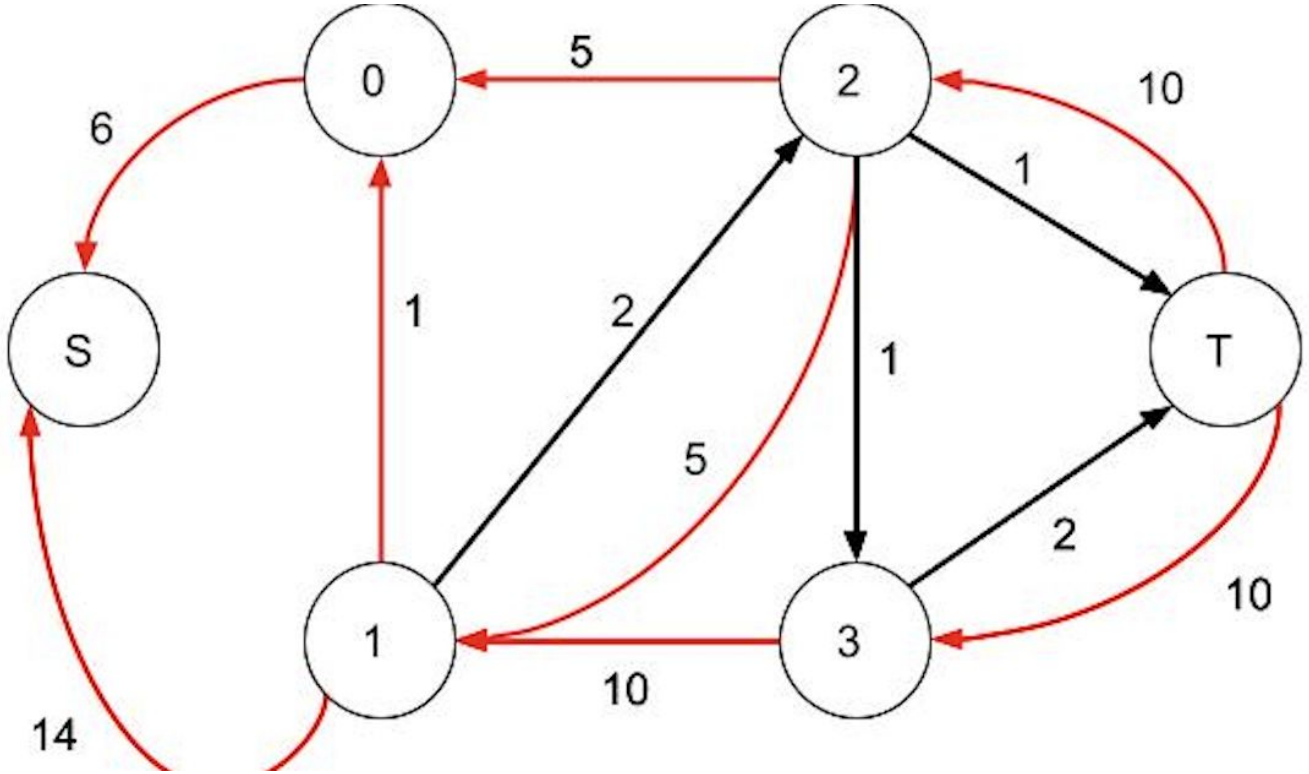


Solution cont'd

At Δ = 2 phase, No possible Augmenting Path

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

Final residual graph:

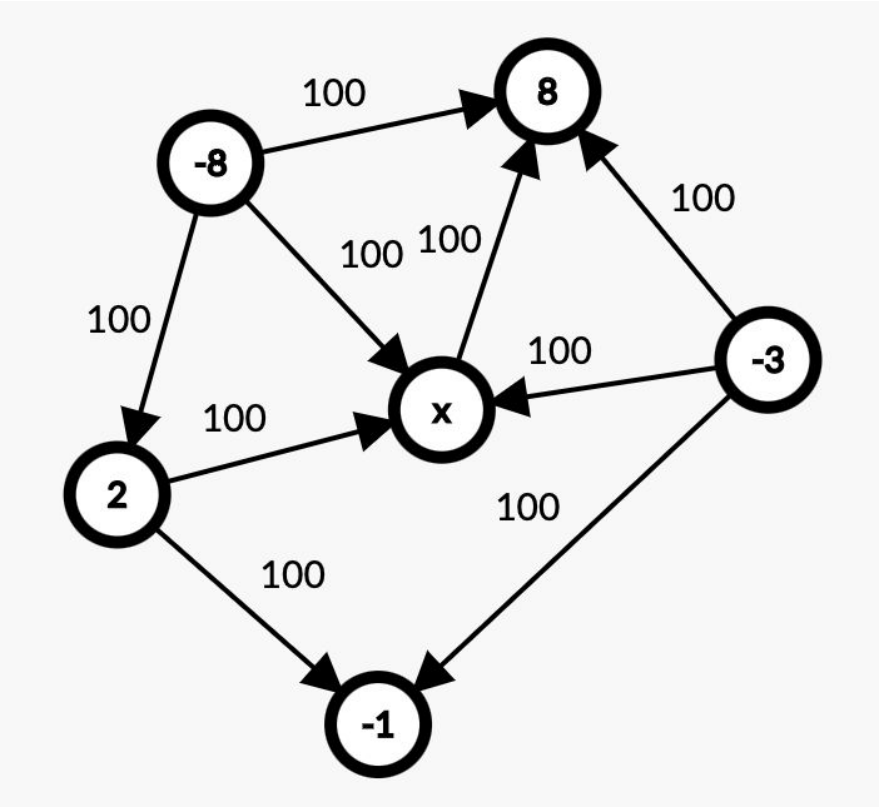


Algorithm Terminates, Max-Flow = 20

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



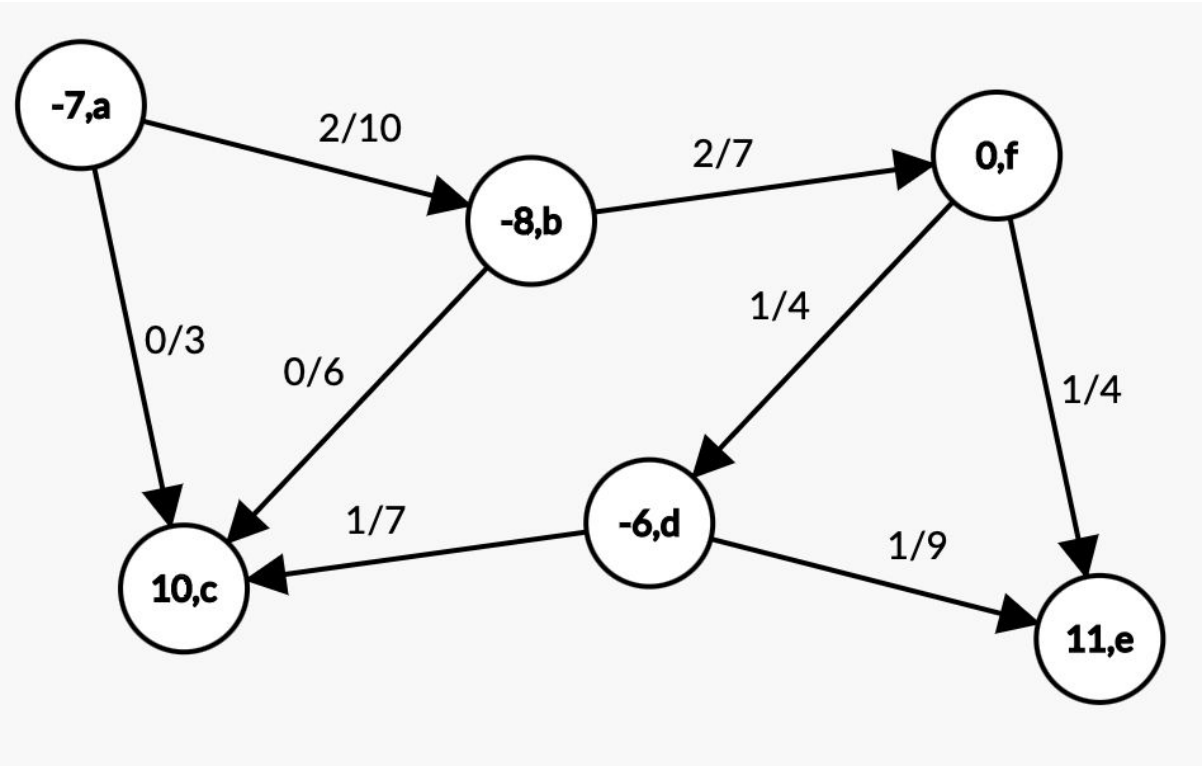
Answer: d

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.

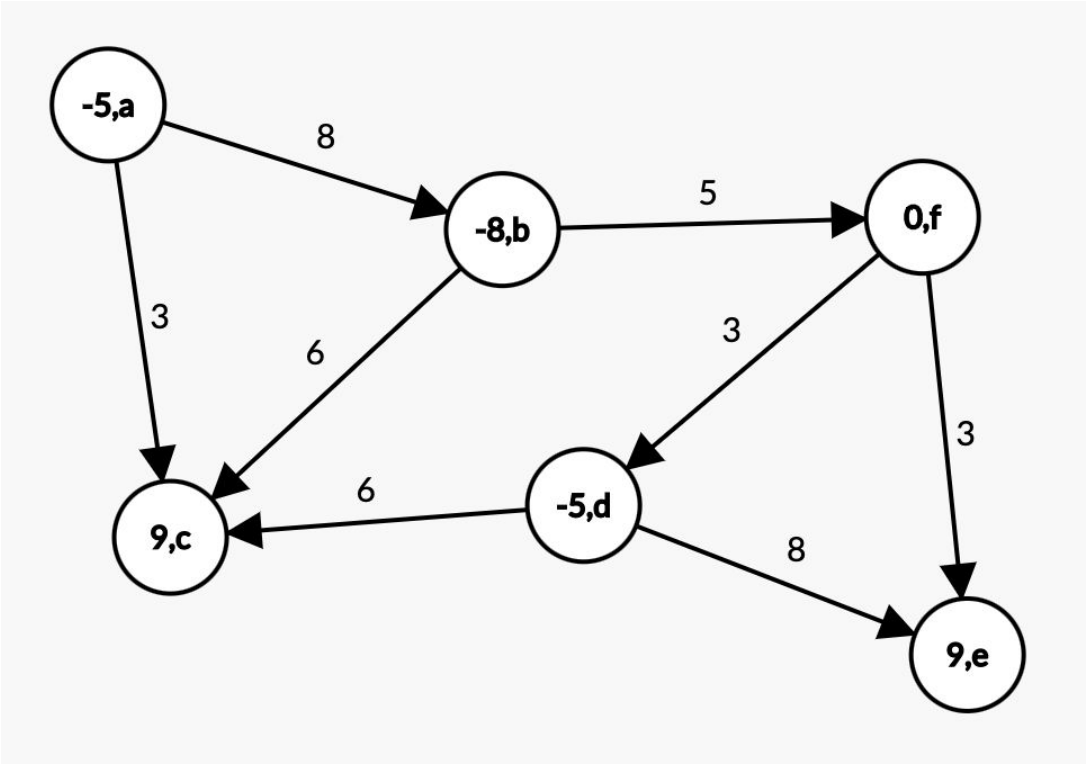
Check if there is a feasible circulation.



Step 1

Push flow f0 through G where f0(e) = le

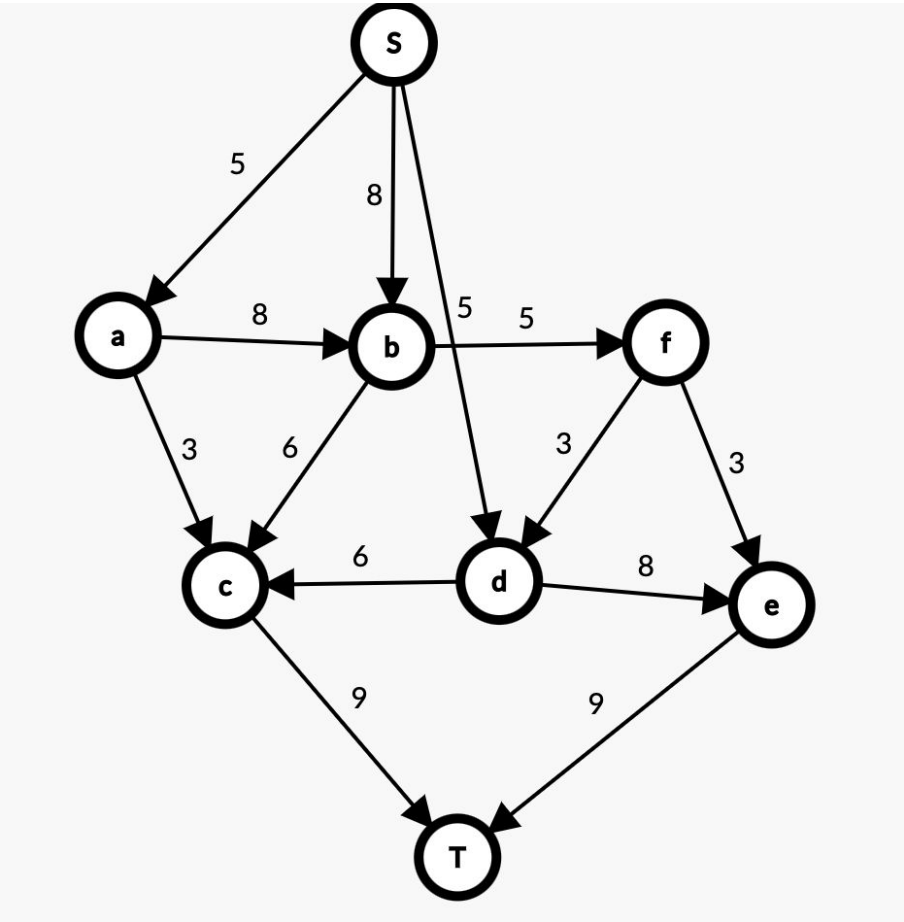
G’ is constructed with c’e = ce - le & d’v = dv - Lv where Lv is the imbalance at node v due to flow f0



Step 2

Create Sink and Source.

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



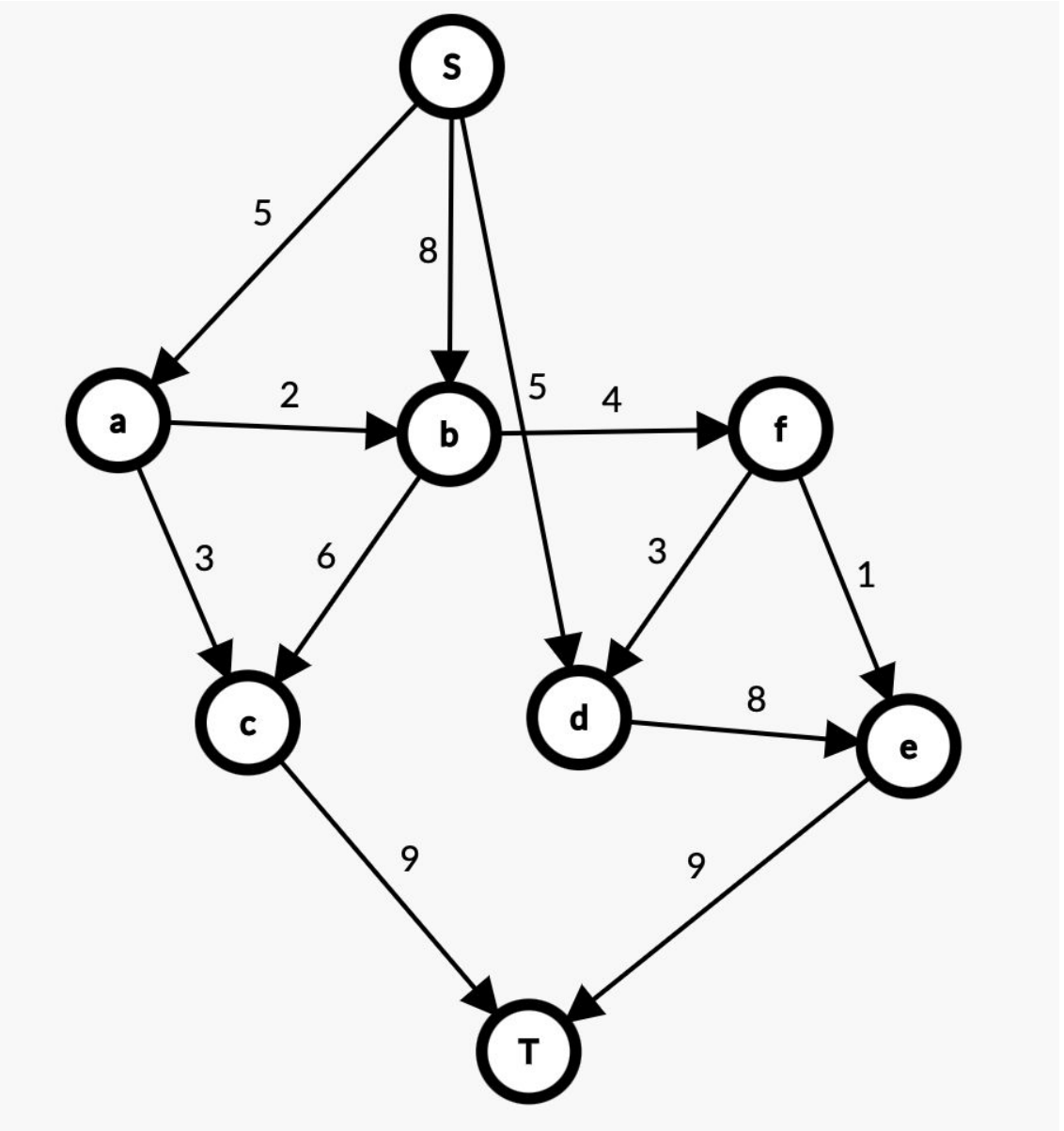
Step 3

Find max flow (f1)

Value of Max flow = 18

D after step 1 = 18

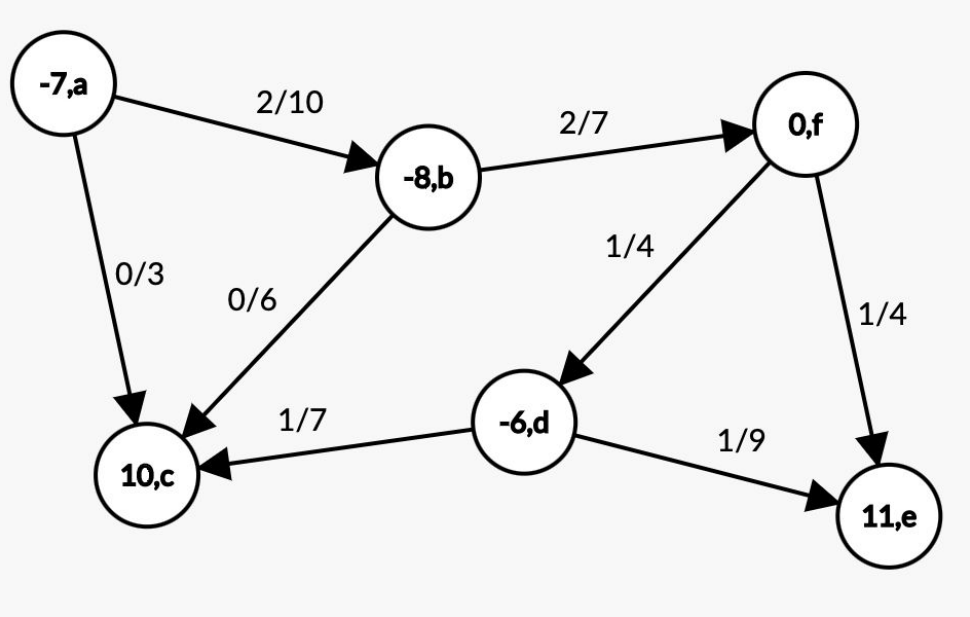
(Sum of positive demand values)

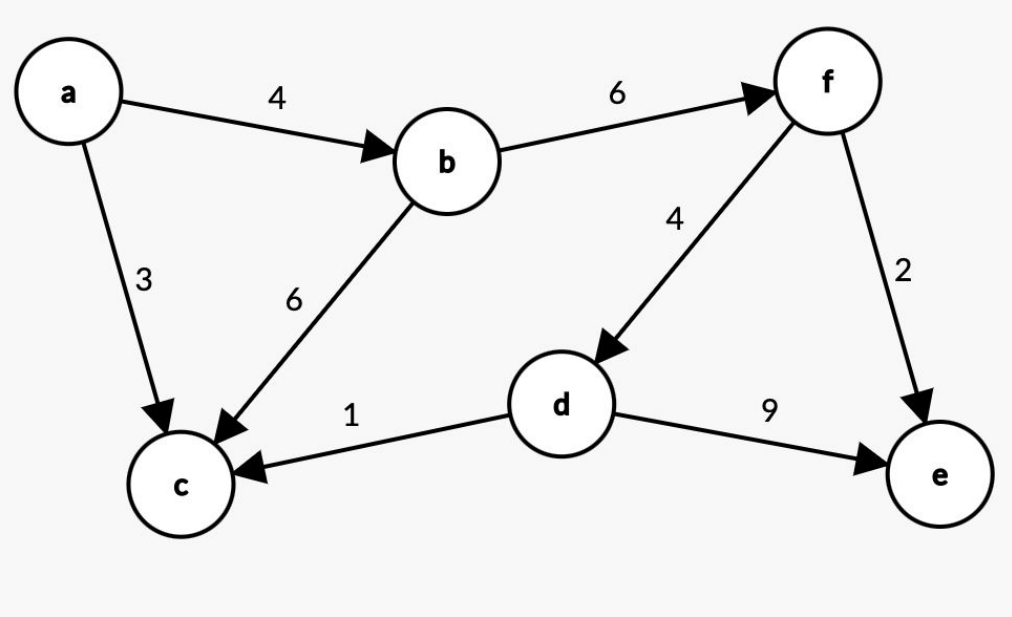


Step 4

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

Final flow (f0 + f1):





Advertisement Policy

You are the manager of a Social media site. There are n users that visit the site at a given minute. There are a total of k demographic groups of the users G1, . . . , Gk and these groups may overlap. E.g., the groups ‘people over the age of 30’, ‘people living in LA’, ‘people of xyz ethnicity’ can all intersect with each other.

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 ith advertiser looks like:

- Advertiser i wants its ads shown only to users that are in at least one of the demographic groups in the set Xi, where $X_i \subseteq \{G_1, \ldots, G_k\}$.
- Advertiser i wants its ads shown to at least ri users each minute.

As the site manager, you want to show at most a single ad to each user while fulfilling the site’s contracts with the m advertisers as above. Give an efficient network flow algorithm to decide if this is possible, and prove correctness.

Solution

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

We construct the following graph G:

- Vertex S, vertices v1,...,vn for each person vertices w1,...,wm for each advertiser
- There is an edge of capacity 1 from vi to each wj for which person i belongs to some demographic group in Xj
- There is an edge with a capacity of 1 from s to each vi; and for each j, there is an edge with lower bound rj from wj to t.
- Finally, S has a demand of $-\sum_j rj$, and each wj has a demand +rj. 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. Suppose that solution shows user i ad j. Then, we put a unit flow on edges (S, vi) and (vi, wj). Doing so gives us a feasible circulation because…

Backward Proof: Suppose there is a valid circulation in this graph. We can always assume there is an integer circulation when all the parameters are integers. Then, given such a circulation, whenever there is 1 unit flow on edge (vi,wj), we show an ad from advertiser j to person i. This satisfies all the advertiser contracts because…

Database connections

A company has n software applications & m databases that support these applications.

- Each application i has F(i) features and they are unique i.e., features are not shared across different applications.
- Each feature j of any application requires one database (DB) connection and we are given a subset of databases D(j) available for feature j.
- Each database k can only accommodate at most C(k) connections in total.
- For an application to run successfully, at most one of its features can fail (i.e., left without a required database connection).

Design a network flow solution to determine if there is an assignment of features to databases such that all applications are running as described above.

Solution 1 (Max flow based)

- Create source S and sink T
- Create n nodes for n applications & Connect S to each app node i with capacity **F(i)-1**
- Create F(i) nodes for each application i representing its features and Connect the application i to each of its feature nodes with capacity of 1
- Create m nodes for m databases (DB).
- Connect each feature node j to the subset D(j) of the DB nodes.
- Connect an edge from each database k to T with capacity C(k)

Claim: We can have all applications running subject to problem constraints if and only if value of max flow = $(\sum_i (F(i)-1))$ in the network above.

Solution 2 (Circulation)

- Create source node S and sink node T. Create a node for each database.
- Create n nodes for the n applications.
- Create F(i) nodes for each application i representing its features and Connect the application i to each of its feature nodes with capacity of 1.
- Connect S to each node application node i with lb F(i)-1 and cap F(i).
- Connect the sink to source node with unlimited capacity.
- Connect each feature node j to the subset D(j) of the DB nodes.
- Connect an edge from each database k to T with capacity C(k)

Claim: We can have all applications running subject to problem constraints if and only if a feasible circulation can be exists in the network above.