CSCI 570

Exam 2 Review Slides

Dynamic Programming

Number of ways - problem type

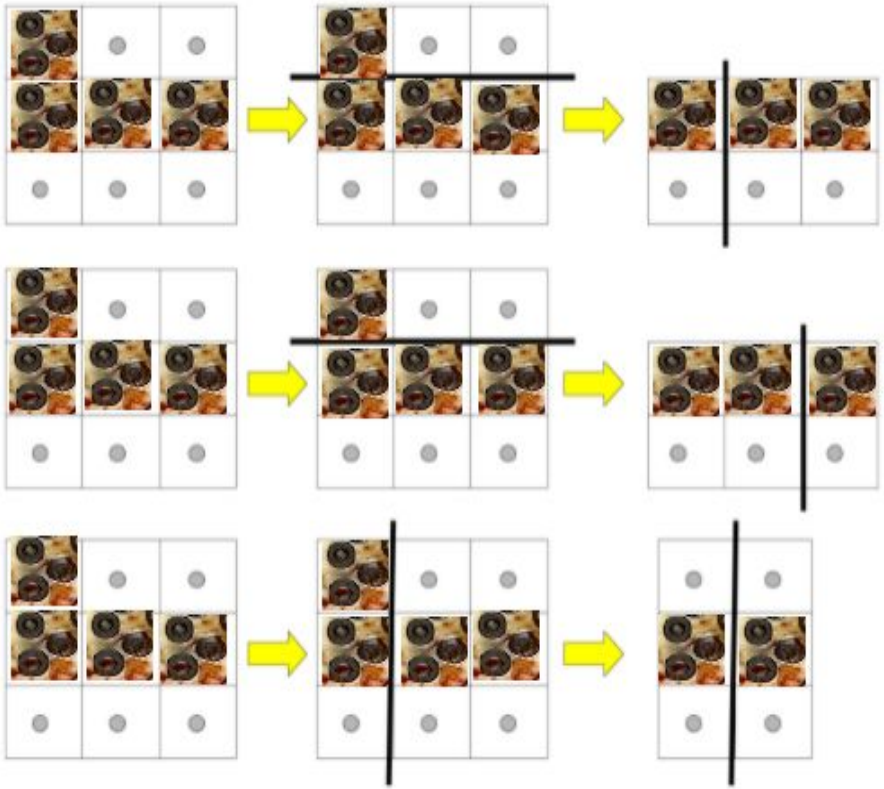
Problem Description

Given a rectangular pizza represented as a rows x cols matrix containing the following characters: 'T' (a topping) and '.' (empty cell) and given the integer k, you have to cut the pizza into k pieces using k-1 cuts.

For each cut you choose the direction: vertical or horizontal, then you choose a cut position at the cell boundary and cut the pizza into two pieces. If you cut the pizza vertically, give the left part of the pizza to a person. If you cut the pizza horizontally, give the upper part of the pizza to a person. Give the last piece of pizza to the last person.

Return the number of ways of cutting the pizza such that each piece contains at least one topping. Your algorithm must run in $\Theta(m * n * (m + n) * k)$ time, where m is the number of rows and n is the number of columns.

Example for a 3x3 pizza



The figure to the left shows how many ways we can cut the given pizza (leftmost columns) to satisfy the condition that there should be *k* pieces and each piece of pizza must have one or more toppings on it.

Evidently, there are 3 ways to cut this pizza to satisfy the constraints.

USC

What are the repeated subproblems?

Evidently, the repeated subproblems are the smaller pieces of pizza. Cutting off the first row, and then the first column gives you the same subproblem as cutting off the first column and then the first row.

This should give us the correct answer, right?

Well… it actually isn't the right representation for a subproblem. You might be able to reach the same sized pizza piece with a *different number of cuts*.

A correct subproblem representation

Instead of just considering a pizza piece as a subproblem, we should instead consider the smaller pizza piece when there are *k_remaining* cuts remaining.

This is the true repeated subproblem.

The search space for this subproblem will be O(m * n * k) because there are *m* rows, *n* columns and *k* number of pieces possible.

Recurrence relation

The choice at each instance of the subproblem boils down to:

1) How should I cut the pizza (horizontally or vertically)?
2) Once I've decided how I should cut it, where should I cut it?

Evidently, we need to exhaust our search space. For this, we should try to cut it at any point possible, on the **condition that the pieces of pizza after the cut both have toppings**. The variables *m* & *n* represent the size of the original pizza.

$$\mathrm{OPT}_{r,c,k} = \sum_{nr=r+1}^{m-1} \mathrm{OPT}_{nr,c,k-1} + \sum_{nc=c+1}^{n-1} \mathrm{OPT}_{r,nc,k-1}$$

Base cases

Evidently, for any *OPT(r, c, k)*, if there are no toppings left on the pizza piece, we should return 0 since we should not consider possible cuts if there are no toppings (question definition).

On the other hand, if we have no more cuts remaining (we've done k - 1 cuts), that means we've found our end goal of cutting the pizza into pieces while ensuring that each piece has at least one topping on it. In this case, we return the value: 1

Summing it all up

Each base case will return a 1 when we've correctly cut up the pizza.

Therefore, our recurrence relation which sums up the values returned by the future states will give us the total number of ways of cutting the pizza.

This is a top down approach to the problem, but this problem can be solved using a bottom up approach as well (iterate r & c from m to 0 and n to 0, and also iterate over k from 1 to the input value *k*).

The time complexity of our solution is O(m * n * k * (m + n))... or is it?

One last step

One important step in the recurrence relation is determining if the two pizza pieces we just obtained by performing a cut both have a topping.

The brute-force method of doing this is to just iterate over all the rows and columns of the newly cut piece and then determine if there is a topping. However, this would make the time complexity of our solution to be: O(m*n*k*(m+n)*m*n) which is far higher than the required runtime complexity as per the problem definition.

To reduce the runtime complexity to the requirement, we'll basically have to perform that step in O(1) time. Can we use a technique from Homework 6 to accomplish this?

The final touch

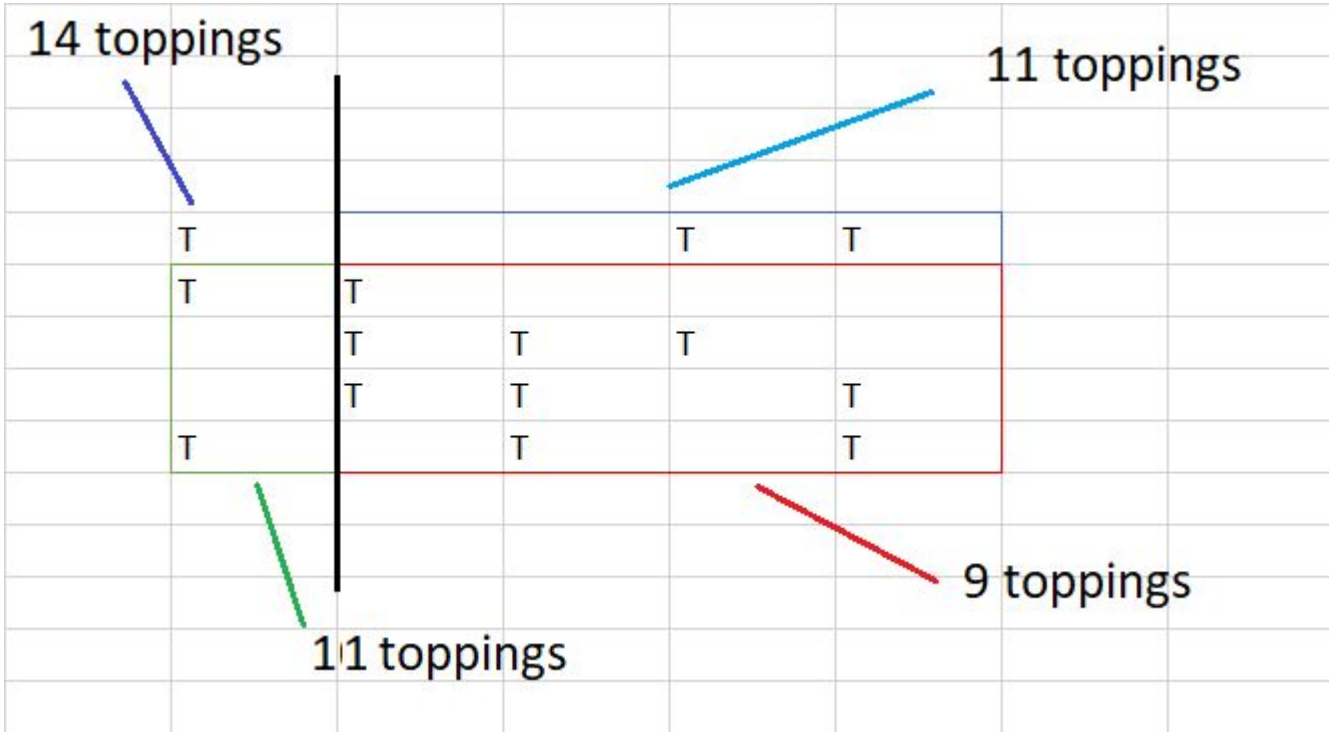
We can use a 2-D prefix sum to perform queries of the type: does the range(r1, c1) to (r2, c2) contain a topping?

It takes O(r * c) to generate this 2-D prefix sum matrix and we can find whether a piece contains a topping in O(1) time after that.

```
for r in range(m - 1, -1, -1):
            for c in range(n - 1, -1, -1):
                preSum[r][c] = preSum[r][c + 1] + preSum[r + 1][c] -
preSum[r + 1][c + 1] + (pizza[r][c] == 'T')
```

With this, we obtain our required time complexity.

Visual explanation of a 2D prefix sum



Analysing the runtime complexity

The runtime complexity of our solution is given as:

O(m * n * k * (m + n)).

Since our runtime complexity depends on the **value** of one of the inputs (k), we can say that our algorithm has a **pseudo-polynomial** runtime complexity.

However, if we analyze it a bit further, we see that we cannot make more than (m + n) cuts on the pizza. Therefore, *k* is bounded by O(m + n). Therefore, our solution has polynomial runtime complexity.

Dynamic Programming

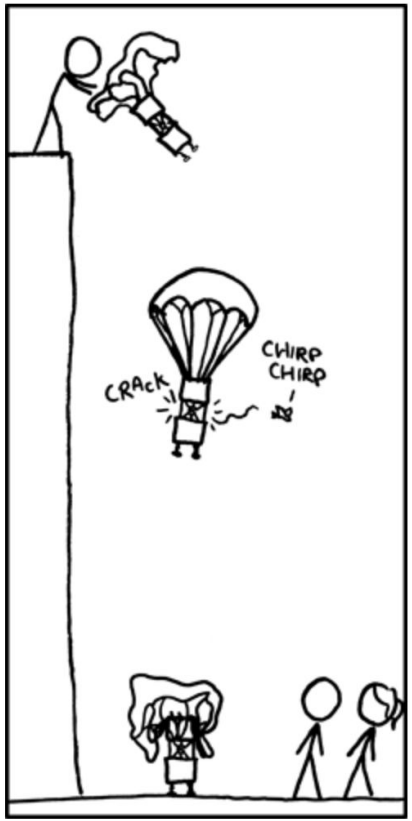
Kiran Lekkala

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 and we have 2 eggs, then we can find the answer 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.
- All eggs break when they are dropped from the n-th floor.

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

Egg dropping puzzle: Base Cases

- If we there are 5 floors and 1 egg, we need to do k trials worst case $k = f = 5$
- 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

Egg dropping puzzle: Understanding the solution

Let $O(n, k)$ denote the number of trials required for an n story building and k eggs, knowing that dropping an egg from the last floor will break it.

Suppose the first move we do is to drop an egg from floor $i < n$.

1. If the egg breaks, then we learn that $ans \leq i$ but we are left with k-1 eggs. To find out the answer from this state we would need extra $O(i-1, k-1)$ steps.

2. If the egg doesn't break, then we learn that $ans > i$ and we still have k eggs. All the above (n - i) floors are the possible candidates and we have k eggs. Therefore, we would need to do $O(n-i, k)$ extra steps.

Egg dropping puzzle: Understanding the solution

Egg dropping puzzle: Understanding the solution

Egg dropping puzzle: Understanding the solution

Egg dropping puzzle: Understanding the solution

$$O(n,k) = \min_{1 \le i < n} [1 + \max\{O(i-1, k-1), O(n-i, k)\}]$$

Base cases:

1. $O(n,k) = 0 \quad \text{if } n = 0$
2. $O(n,k) = 1 \quad \text{if } n = 1$
3. $O(n,k) = n \quad \text{if } k = 1$

Complexity: $O(n^2 k)$

Egg dropping puzzle: 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

Egg dropping puzzle: Complexity analysis

- Time Complexity: $O(n^2 * k)$
  - n is the number of floors
  - k is the number of eggs

- Memory Complexity: $O(n * k)$

Egg dropping puzzle: 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...

Dynamic Programming

Guanyang Luo

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

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

Solution: Base cases:

- f(0) = 0 (0 seat, 0 valid arrangement)
- f(1) = 2 (1 seat, 1 valid arrangement: either F or P)
- f(2) = 3 (2 seat, 3 valid arrangement: either F-F, F-P, or P-F)
- for n > 2: use the recurrence above.

Number of Valid Seating Arrangements

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

Solution: To solve the problem in O(n) time, we can use dynamic programming (through **Memoization**). By solving a sub-problem only once, for all subsequent occurrences of the sub-problems, we can use the precomputed result to solve further queries.

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.

Ford-Fulkerson and Capacity Scaling

Swetha Sivakumar

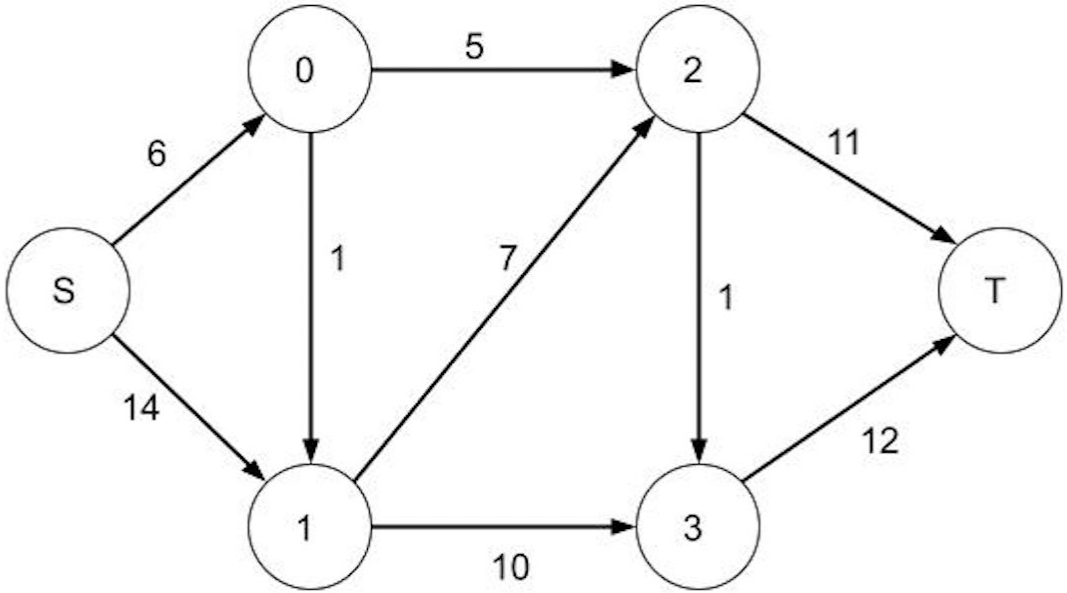
Question 1:

For the given Network Flow graph below:

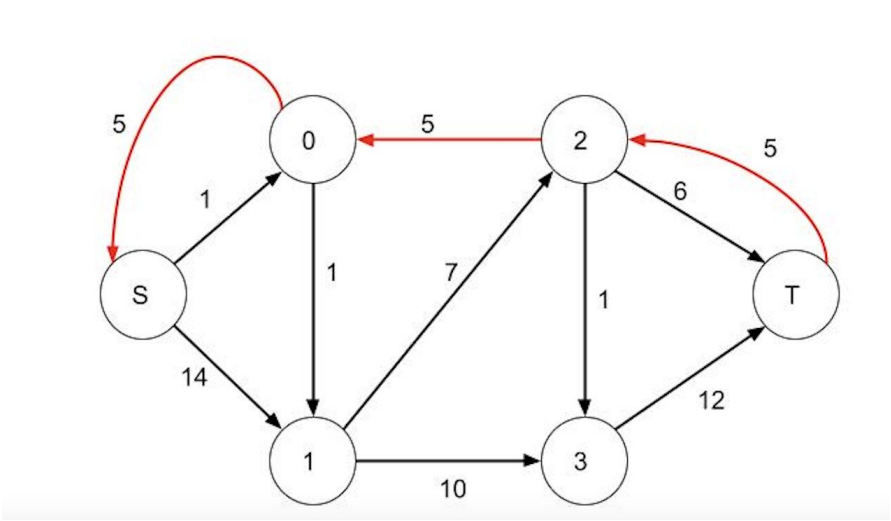
(i) Give the Final Residual graph.

(ii) What is the max-flow of the network?

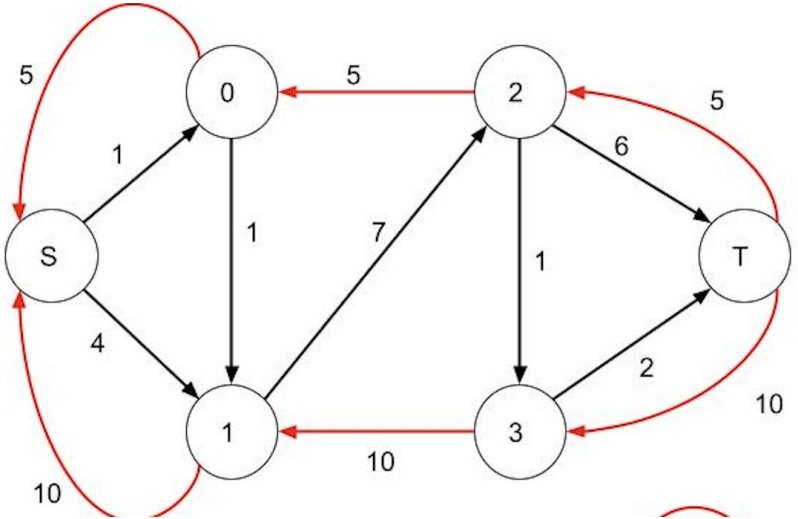
(iii) Identify the min-cut.



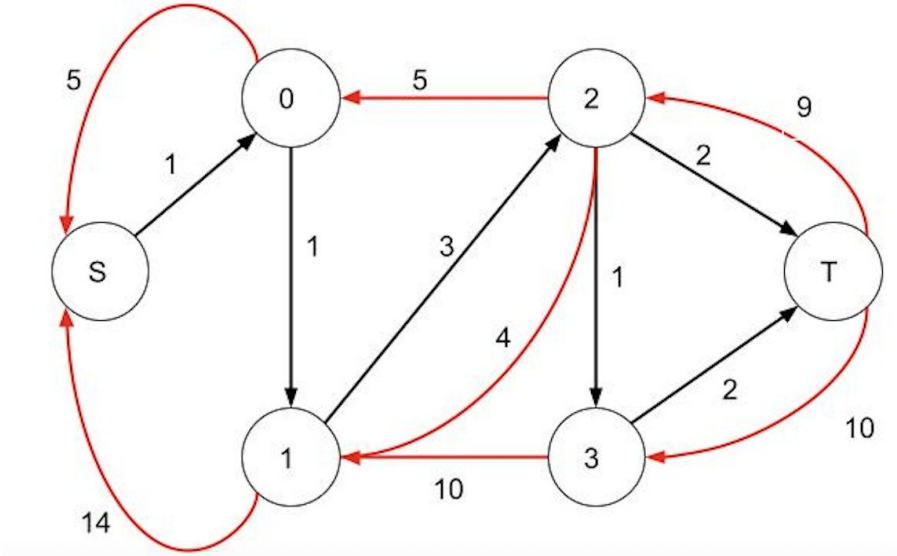
Solution:



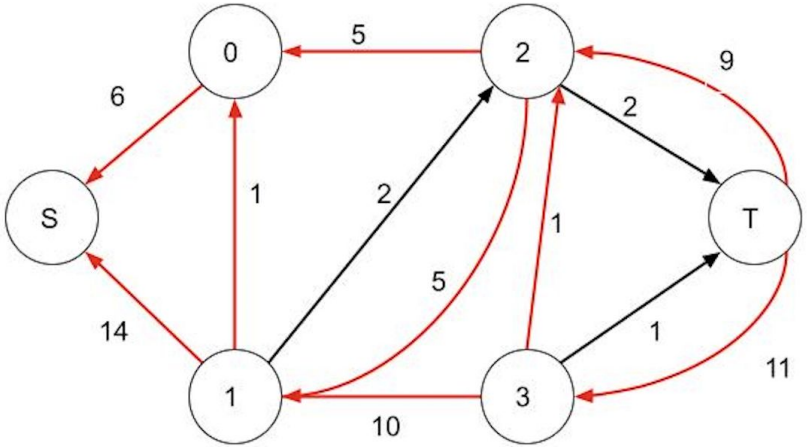
Residual Graph 1



Residual Graph 2



Residual Graph 3

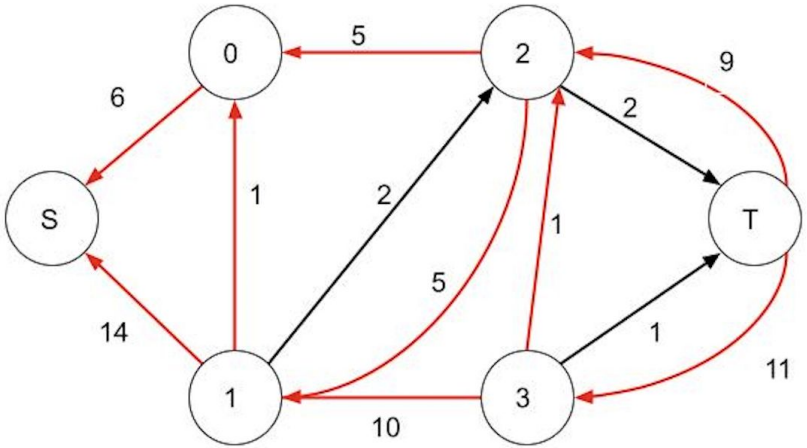


Residual Graph 4 - Final Residual Graph

Max-Flow = 20

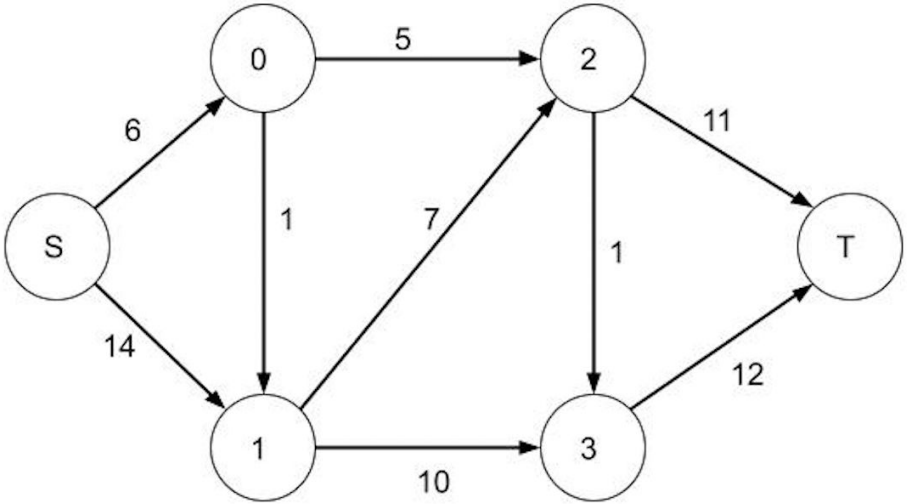
Min-Cut = {S}, {0,1,2,3,T}

Final Residual Graph:



Question 2

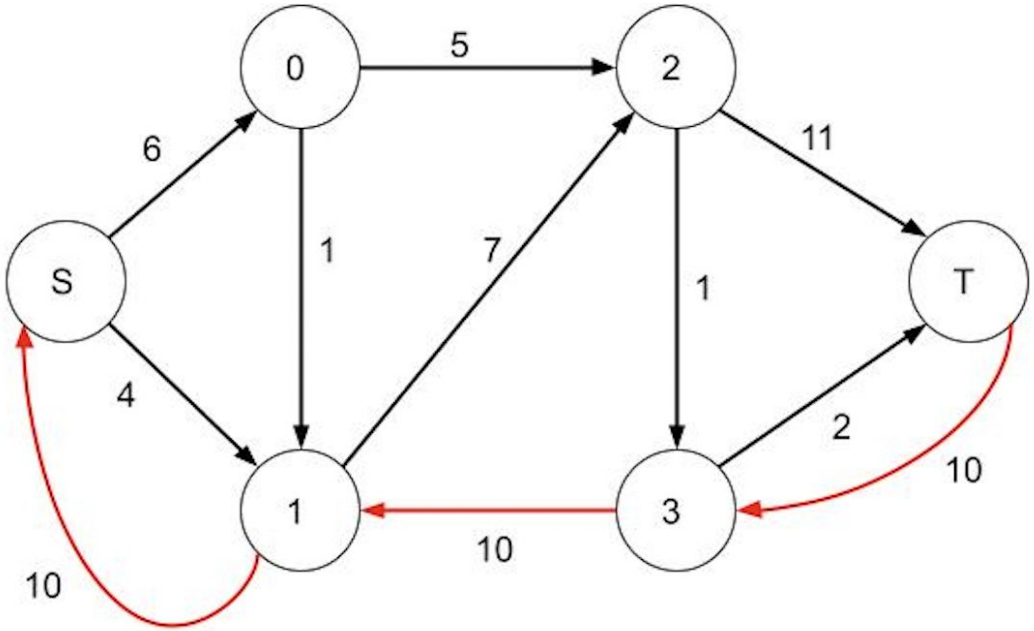
Using Capacity scaling calculate the max-flow of the following given network:



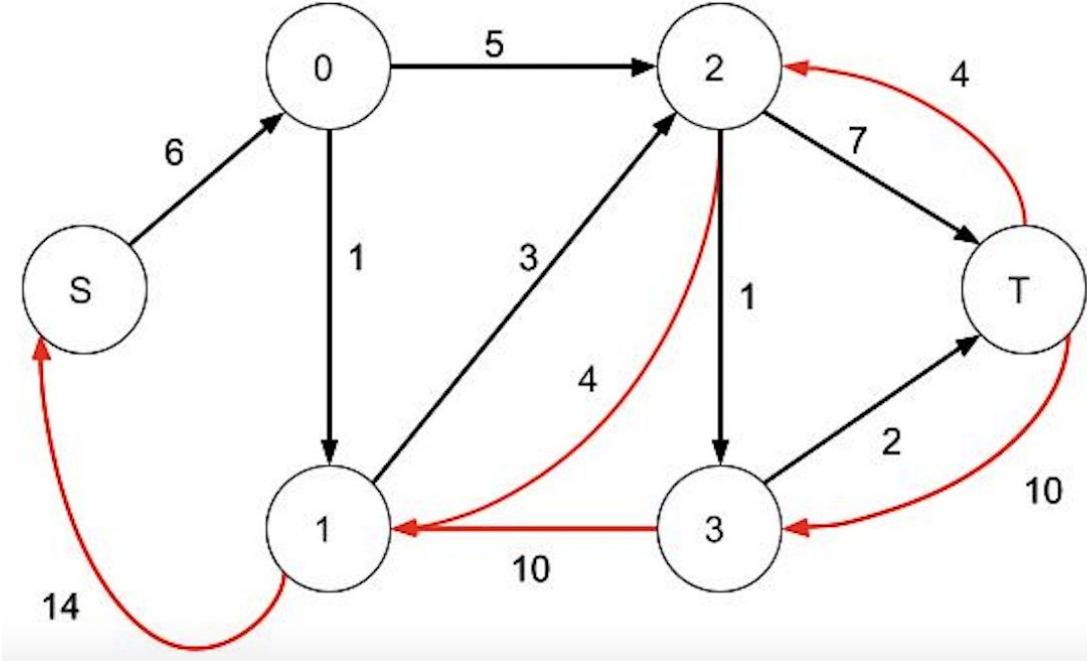
Solution

U = 14

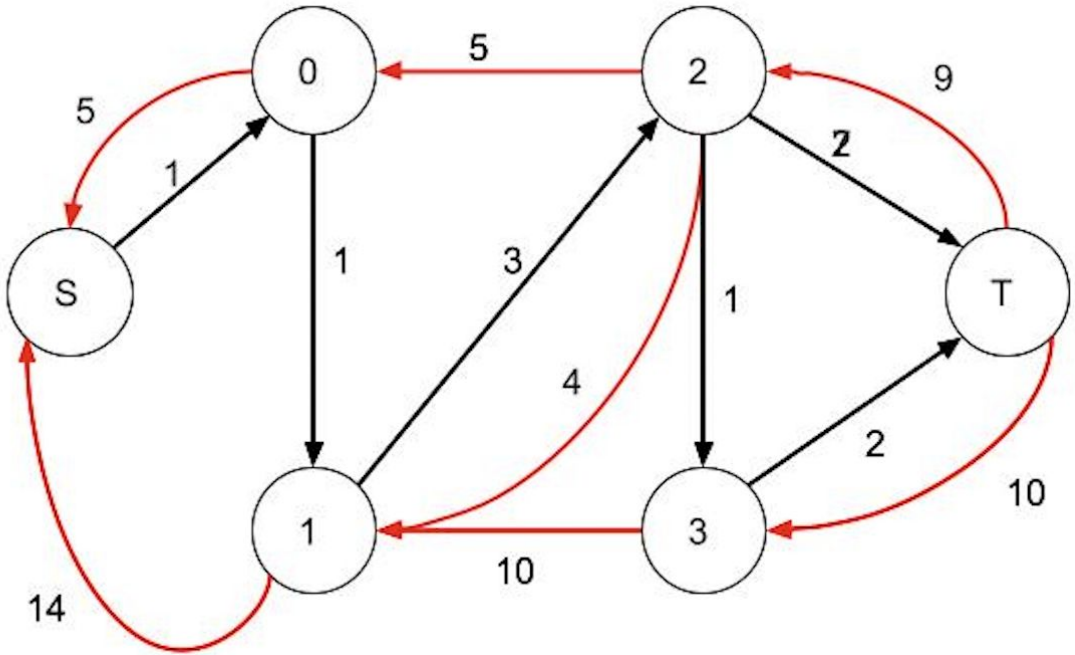
At Δ = 8 phase we have 1 augmenting path.



At Δ = 4 phase we have 2 Augmenting Paths

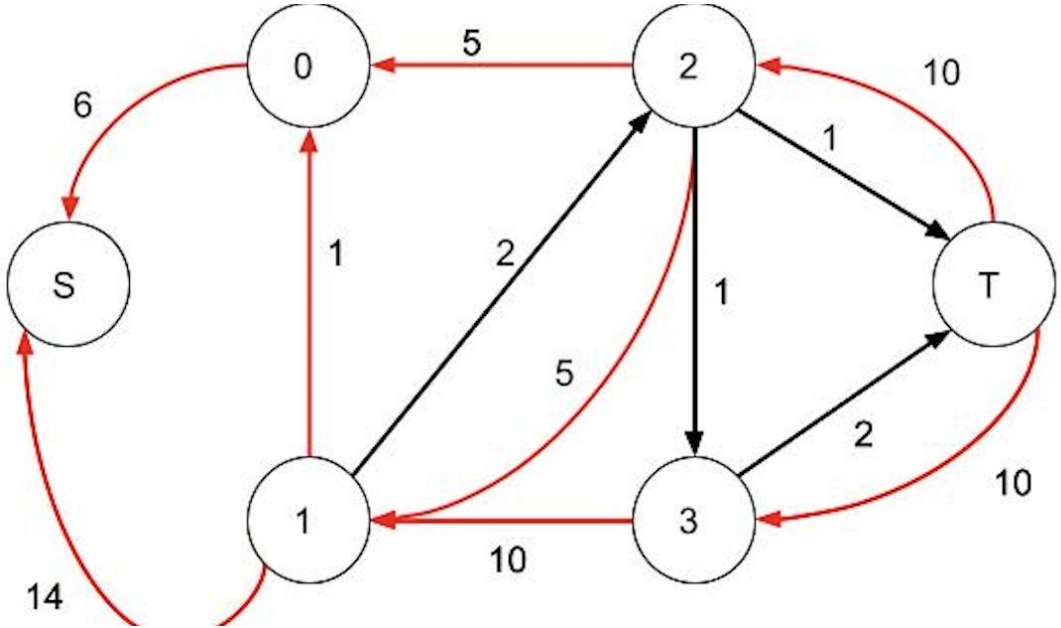


At Δ = 4



Δ = 2, No possible Augmenting Paths

At Δ = 1, 1 Augmenting path possible



Δ = 0, Algorithm Terminates , Max-Flow = 20

Application of Network Flow

Bhaskar Goyal

T/F Question

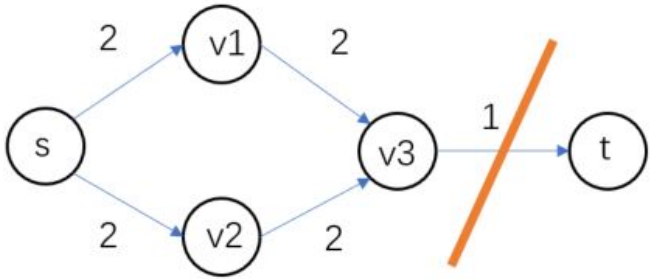
For any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer $k > 1$, then that edge will no longer be part of the minimum cut.

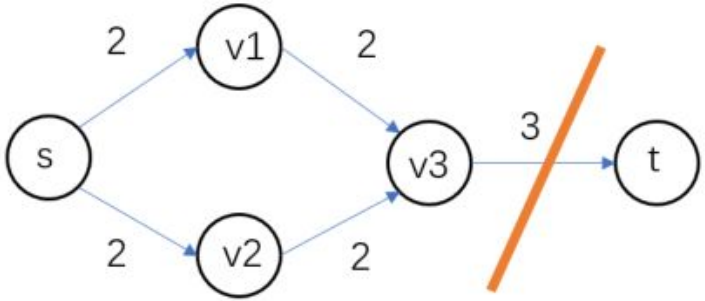
T/F Question

For any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer k > 1, then that edge will no longer be part of the minimum cut.

False.

Counter Example for k = 2





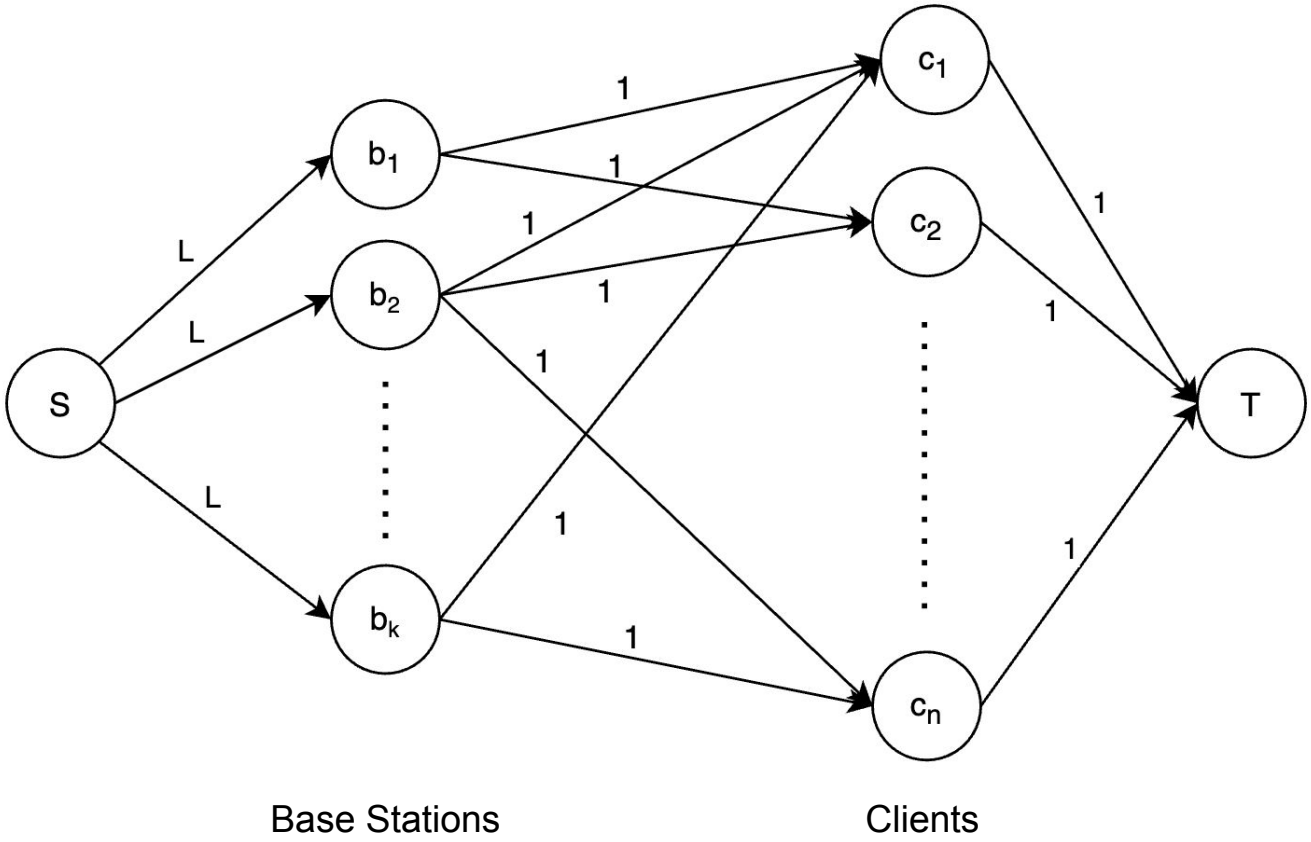
Mobile Client Connection

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, with the position of each client specified by its (x, y) coordinates in the plane. There are also *k* base stations; 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* - a client can only be connected to a base station that is within distance *r*. There is also a load parameter *L* - no more than *L* clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. 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, subject to the range and load conditions in the previous paragraph.

Solution: Mobile Client Connection



Solution: Mobile Client Connection

Let s be a source vertex and t a sink vertex. Introduce a vertex for every base station and introduce a vertex for every client.

For every base station vertex b, add an edge (s, b) of capacity L. For every client vertex c, add an edge (c, t) of unit capacity. For every base station vertex b, add a unit capacity edge from b to every client c within its range.

Run a polynomial time max flow algorithm (like Edmonds Karp) on the constructed network graph to find max flow. Output YES if and only if the max flow is n.

Solution: Mobile Client Connection

Claim - We claim that every client can be connected to a base station subject to load and range conditions if and only if the max flow of the constructed network is n.

Forward Claim: The max flow of the constructed network is n if every client can be connected to a base station subject to load and range conditions.

Backward Claim: Every client can be connected to a base station subject to load and range conditions if the max flow of the constructed network is n.

Solution: Mobile Client Connection

Forward Claim: The max flow of the constructed network is n if every client can be connected to a base station subject to load and range conditions.

Proof: If every client can be connected to a base station subject to load and range conditions, then if a client c is served by base station b, assign a unit flow to the edge (b, c). Assign the flows to the edges leaving the source (respectively the edges leaving the sink) so that the conservation constraints are satisfied at every base station (respectively client) vertex is satisfied. (Note that such an assignment is unique). Further, since every client can be connected to a base station subject to load conditions, the capacity constraints at the edges leaving the source are also satisfied. Hence we are left with a valid flow assignment for the network. Since every client is connected, the flow entering the sink is exactly n.

Solution: Mobile Client Connection

Backward Claim: Every client can be connected to a base station subject to load and range conditions if the max flow of the constructed network is n.

Proof: If the max flow of the network is n then there exists a integer flow of flow value n (since all capacities are integral). We will work with this integral flow. For every edge (b, c) with a flow of 1, assign c to the base station b. Since a client vertex can at most contribute one unit of flow entering t, and the total flow entering t is n, every client vertex has flow entering it. This implies that every client is serviced by exactly one station. Since the flow entering a base station vertex b is at most L, a base station is assigned to at most L clients. Thus every client is connected to a base station subject to load and range conditions.

Edge Connectivity

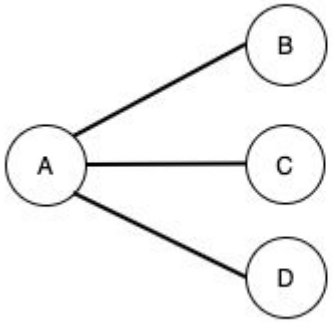
The edge connectivity of an undirected graph G = (V, E) is the minimum number of edges that must be removed to disconnect the graph (or minimum number of edges that must be removed to split graph into 2 sub graphs). For example, the edge connectivity of a tree is 1.

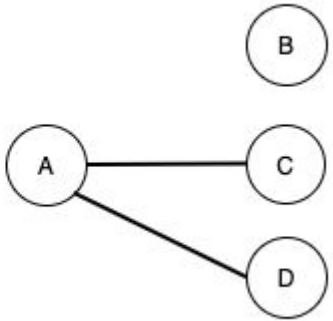
(a) Show how the edge connectivity of an undirected graph G with n vertices and m edges can be determined by running a maximum-flow algorithm.

(b) Describe how many max-flow computations will be required to solve edge connectivity.

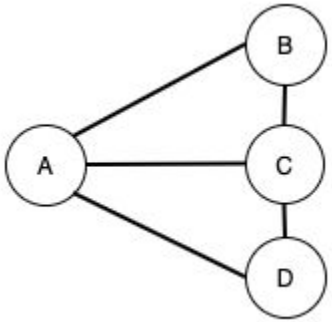
Example: Edge Connectivity

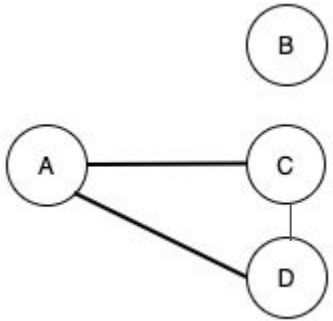
k = 1





k = 2



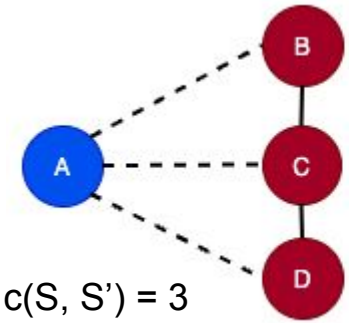


Solution: Edge Connectivity

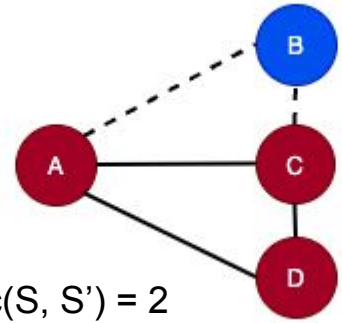
Let V be the set of n vertices.

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

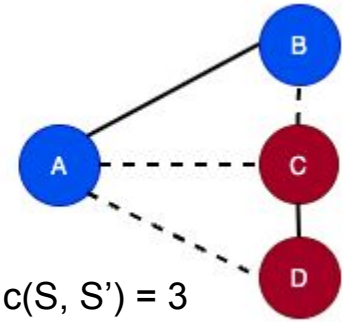
By definition, the edge connectivity, $k = min \{ c(S, S') \}$ *where* $S \subset V$.
(Meaning that for any 2 mutually exclusive set of vertices formed by a cut, the minimum number of edges required to disconnect graph is the minimum number of edges crossing that cut.)



c(S, S') = 3



c(S, S') = 2



c(S, S') = 3

k = min { 3, 2, 3, ... }

= 2

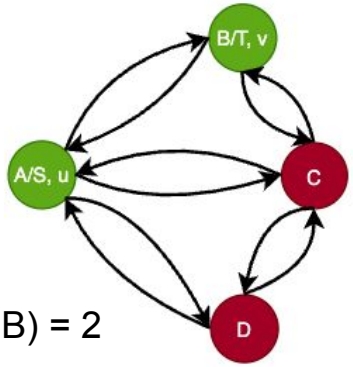
Solution: Edge Connectivity

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* graph,

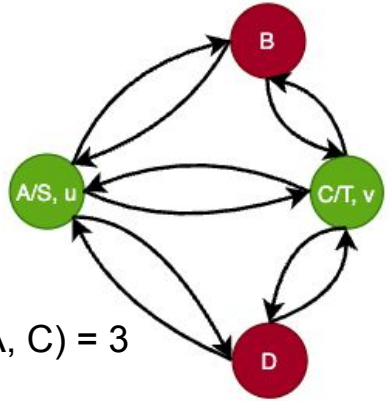
Fix a vertex u ∈ V. For every cut (S, S') in G*, there is a vertex v ∈ V such that u and v are on either side of the cut.

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

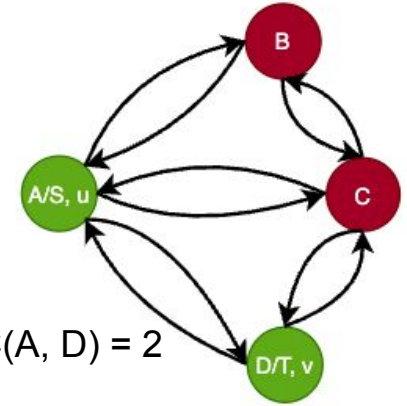
Thus, *K = min C(u, v) where v∈V, v ≠ u*



C(A, B) = 2



C(A, C) = 3



C(A, D) = 2

K = min{2, 3, 2}
= 2

Solution: Edge Connectivity

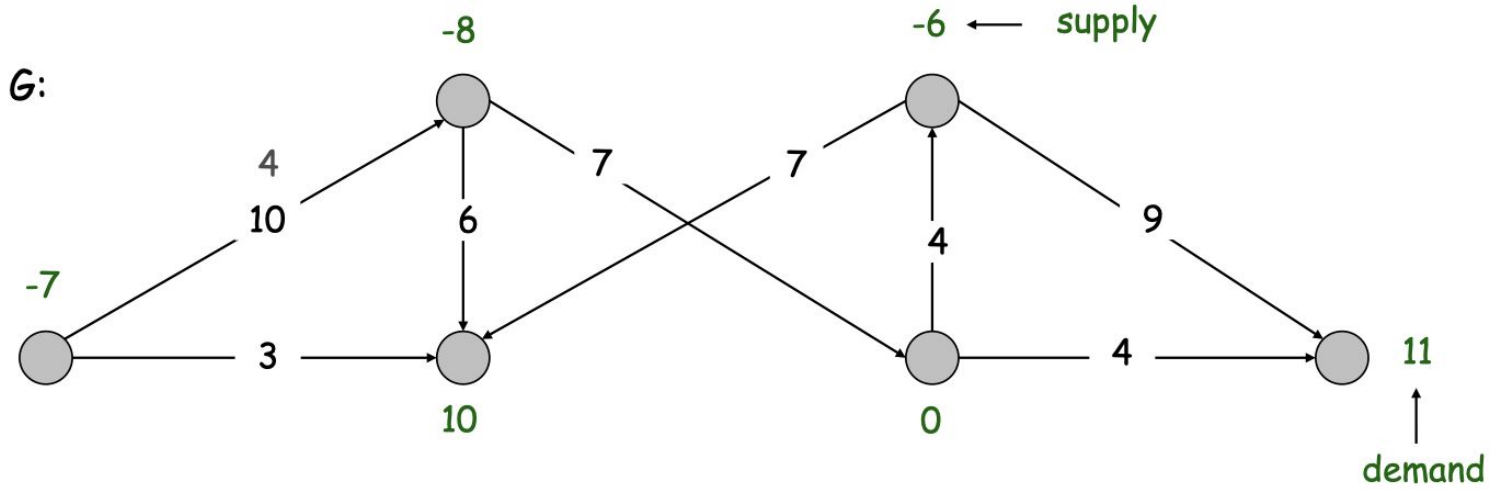
To compute C(u, v) where u ≠ v. Simply, compute the max flow from u to v. (Meaning - To compute the value of mincut of u-v, calculate the max flow taking u as the source and v as the sink vertex in G*. Because of the min-cut max-flow theorem.)

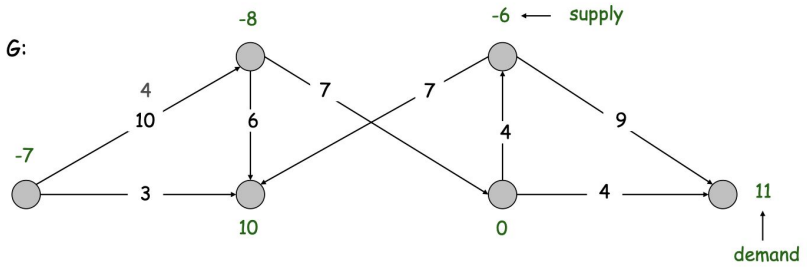
(b) Also, there are in total of n vertices, and we fix a vertex u. Hence, we need to run max flow for all the remaining vertices except u.

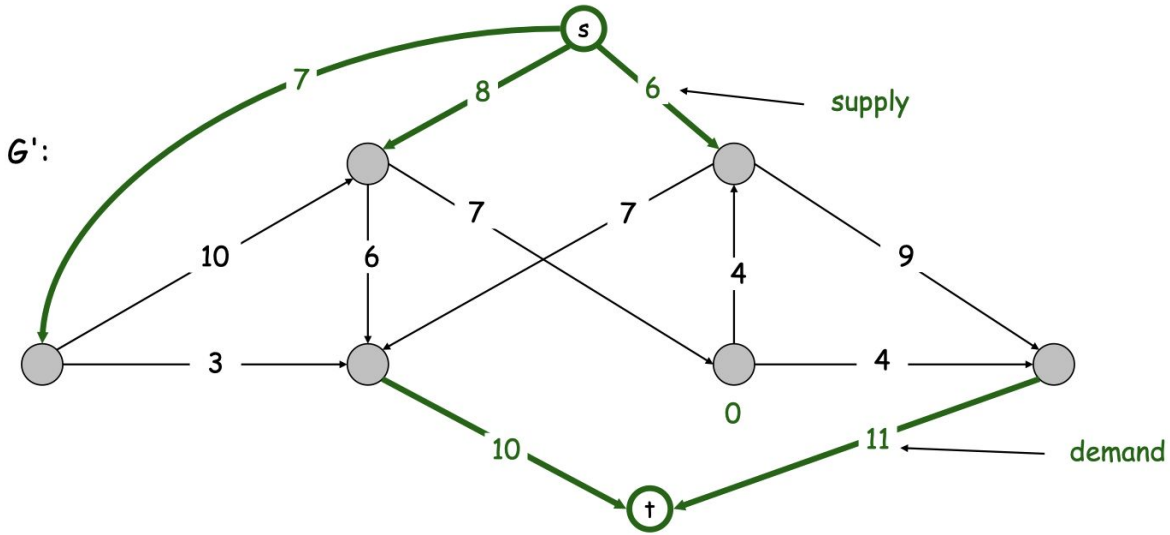
Therefore, there will be n-1 max flow computations.

Circulation &
Circulation with lower bounds

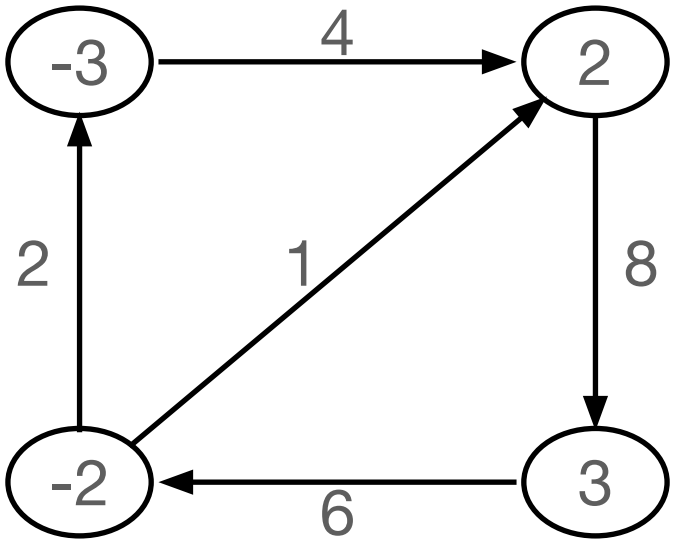
Jiang Feng


G:
-8
-6
supply
4
10
6
7
7
4
9
-7
3
4
11
10
0
demand

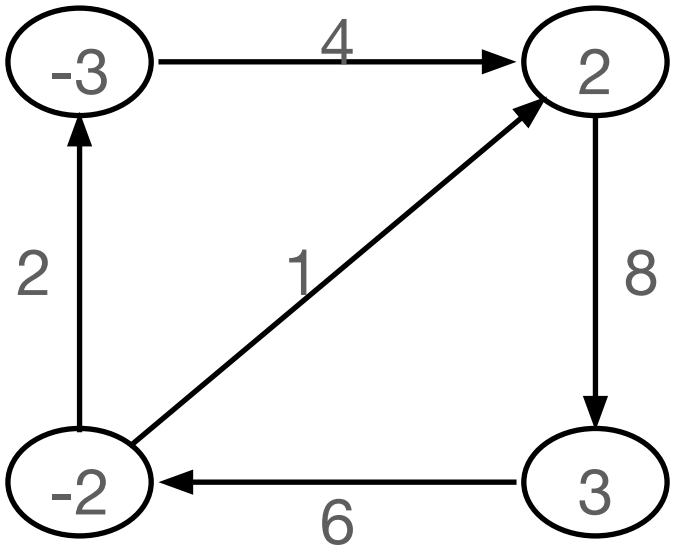

G:
-8
-6
supply
4
10
7
7
6
4
9
-7
3
10
0
4
11
demand


G':
s
7
8
6
supply
10
6
7
7
4
9
3
0
4
10
11
demand
t

Circulation



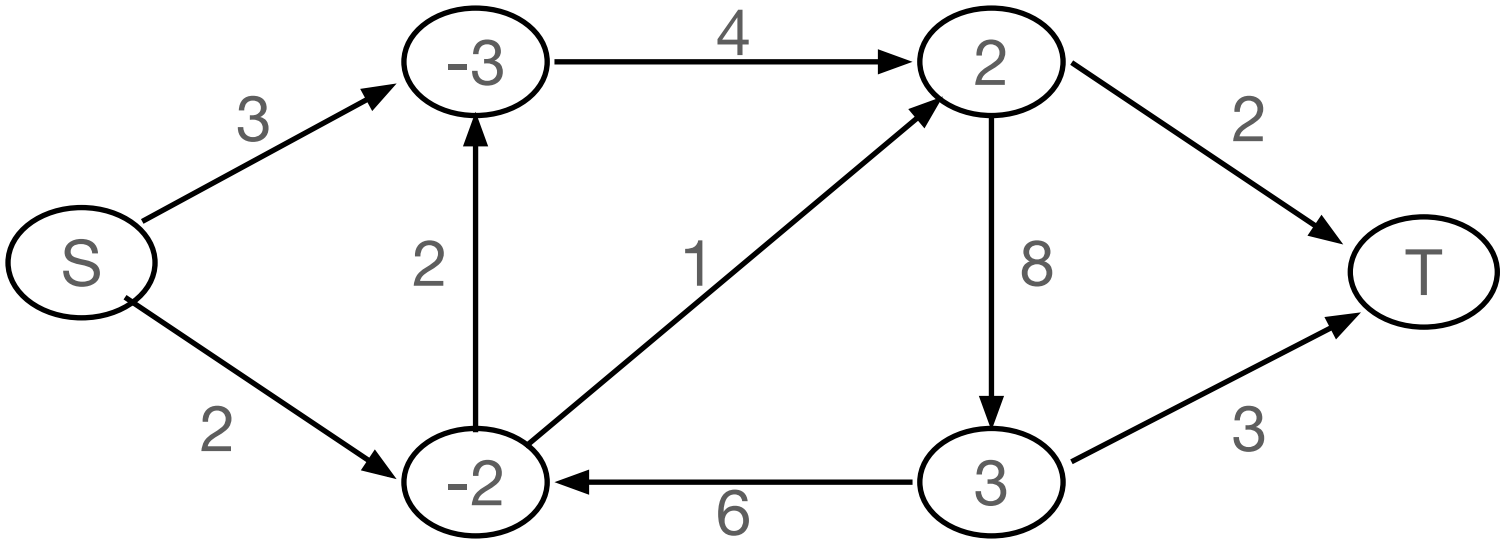
Circulation



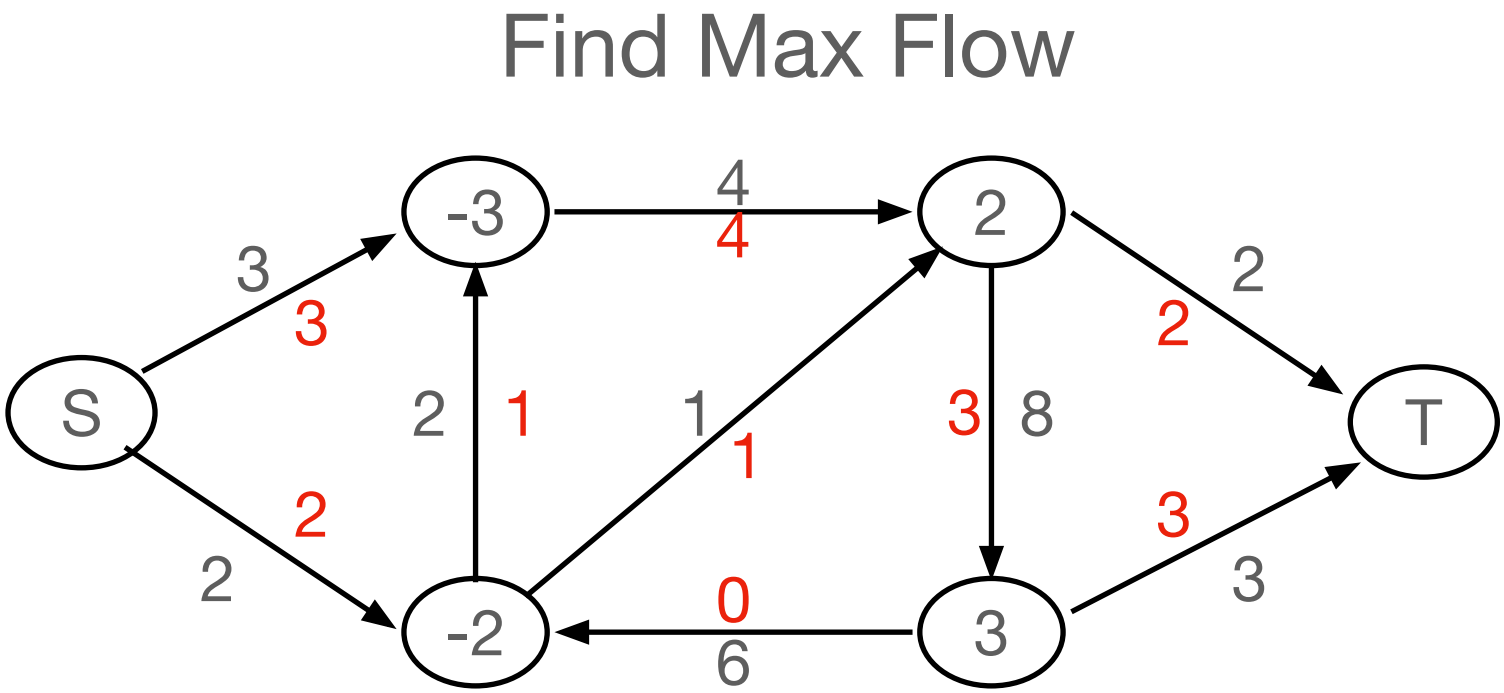
Check the demand value.
Sum of all demand values = 0

Circulation

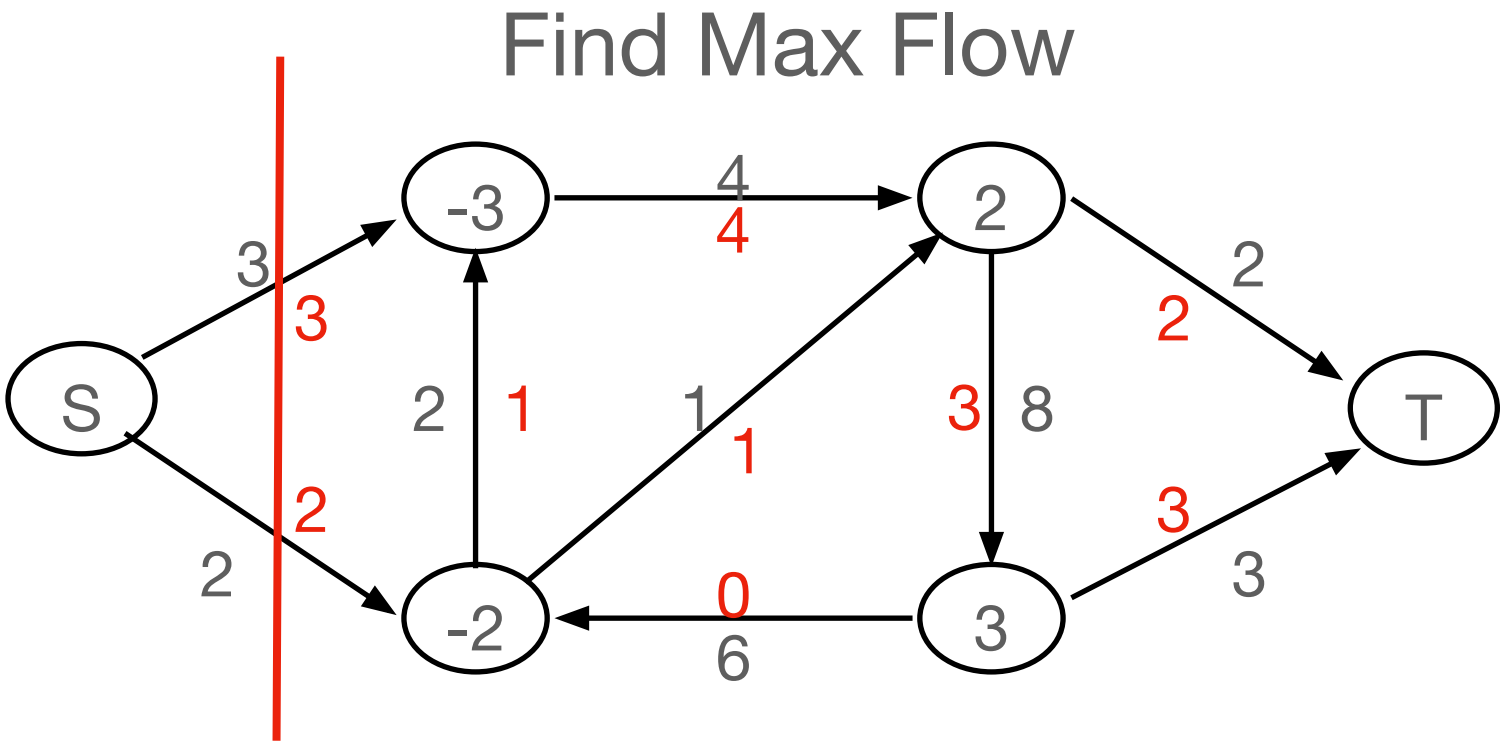
Add S and T



Circulation



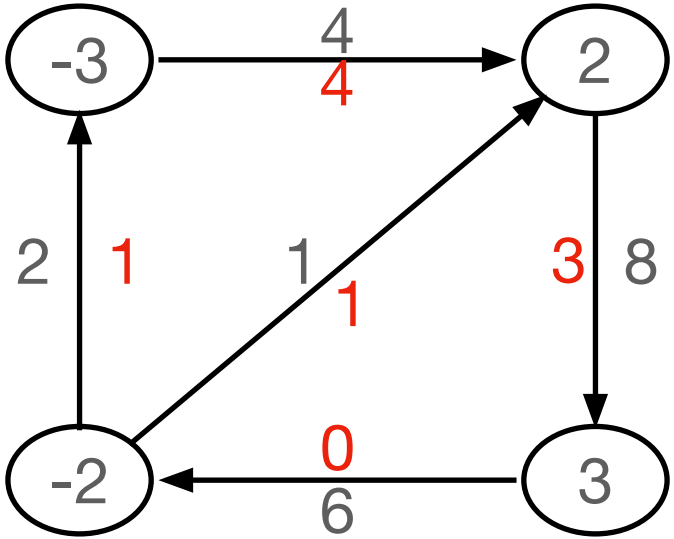
Circulation



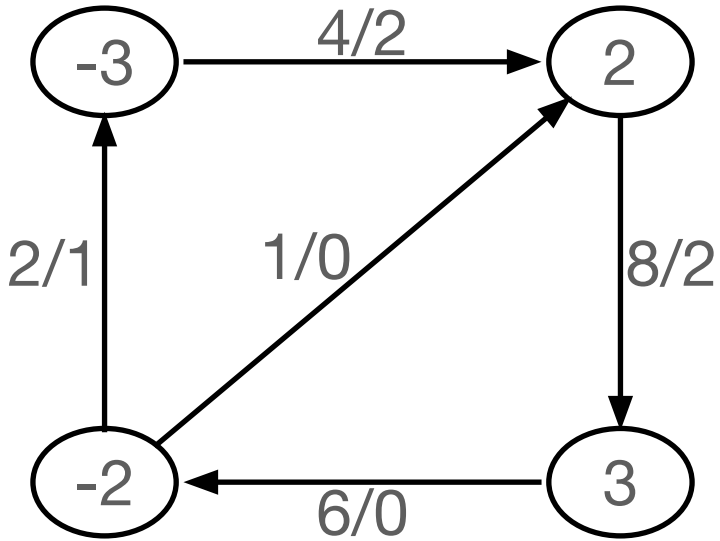
Check if all edges out from S are fully used (Capacity = flow)

Circulation

Remove S and T
Get answer



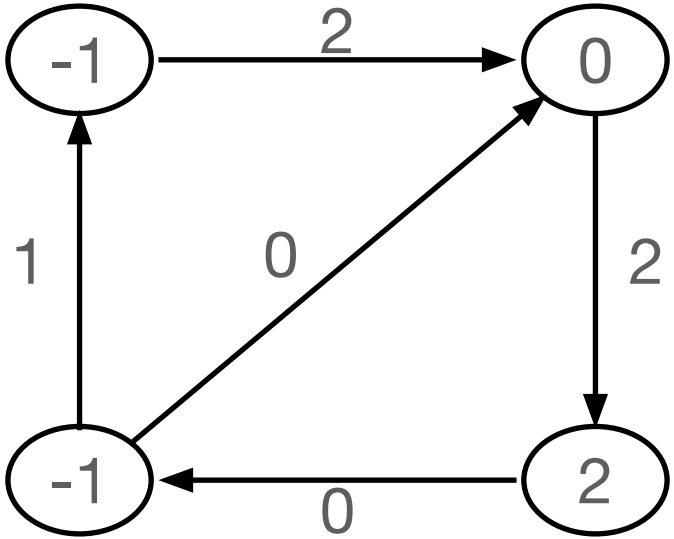
Circulation with Lower Bounds



G

Capacity/Lower bound

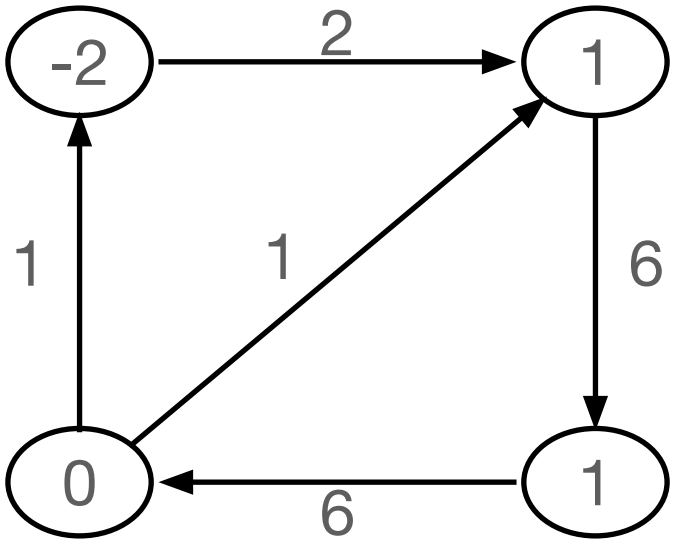
Circulation with Lower Bounds



f0 = smallest flows satisfy the lower bounds

Don’t forget change demand in node

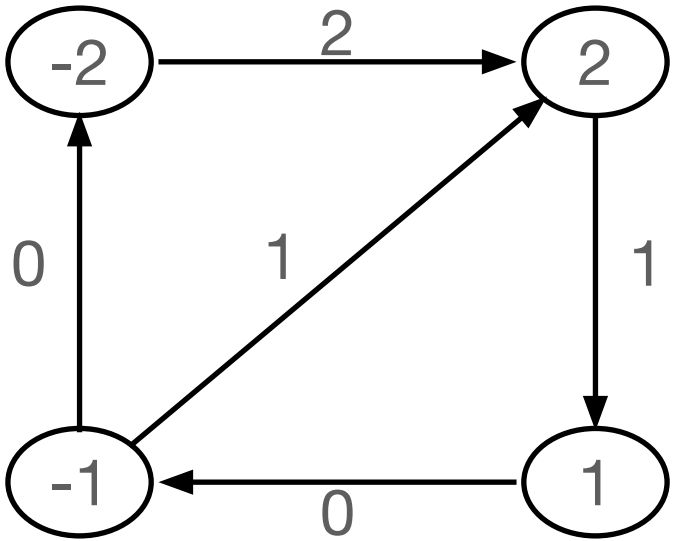
Circulation with Lower Bounds



G’ = G - f0

Both in edge and node

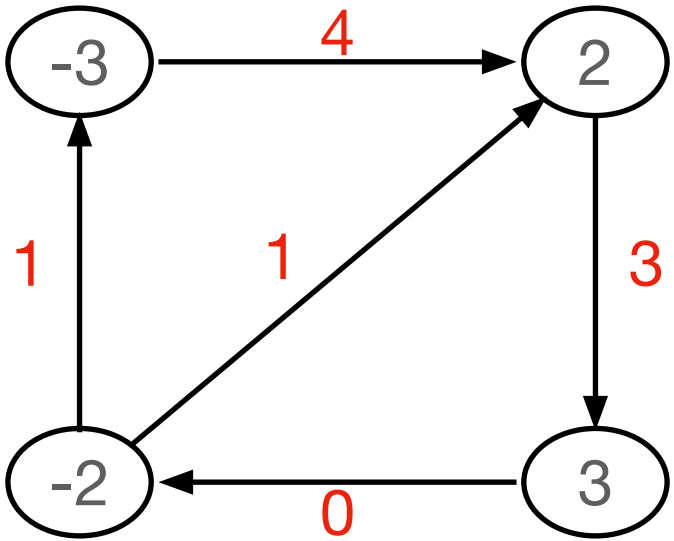
Circulation with Lower Bounds



f1 = find max flow in G’

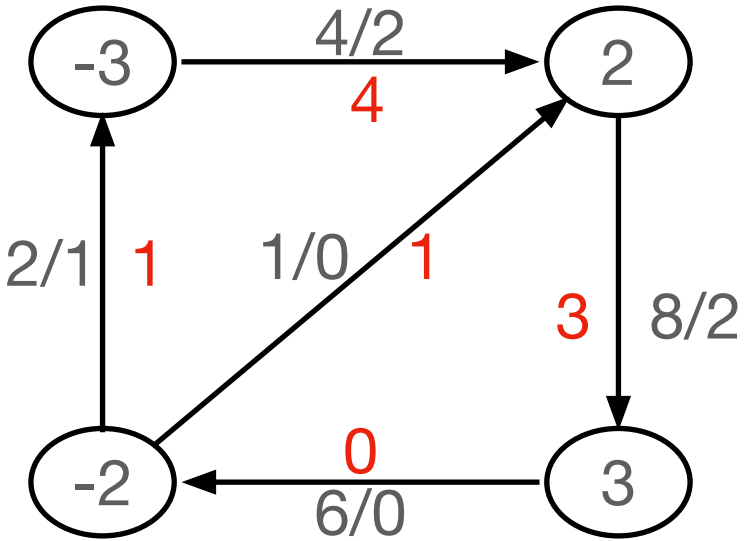
Use the same method mentioned above

Circulation with Lower Bounds



Answer = f0 + f1

Circulation with Lower Bounds



Answer in G

Capacity/Lower bound

Problem

For class 2022, students want to graduate. But they need to meet following conditions:

1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times,
   Repeat attendance can regard as different people

Please design an algorithm to determine if all students can graduate.

Problem

For class 2022, students want to graduate. But they need to meet following conditions:

1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
   Repeat attendance can regard as different people

Students node demand:
The class count requirement for students
[S1, S2, S3…]

Lessons node demand:
The particular people number requirement
[L1, L2, L3…]

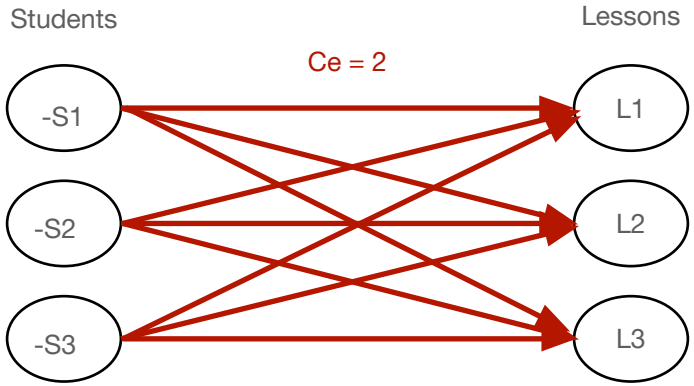
Problem

For class 2022, students want to graduate. But they need to meet following conditions:

1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
   Repeat attendance can regard as different people

Students node demand:
The class count requirement for students

Lessons node demand:
The particular people number requirement



Link all student nodes and lesson nodes,
Capacities are 2

Problem

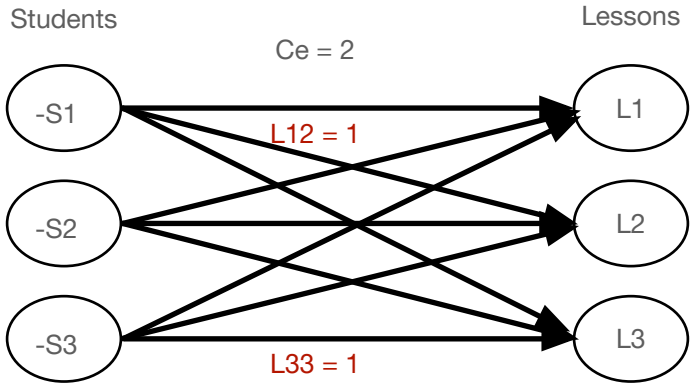
For class 2022, students want to graduate. But they need to meet following conditions:

1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
   Repeat attendance can regard as different people

Students node demand:
The class count requirement for students

Lessons node demand:
The particular people number requirement

Link all student nodes and lesson nodes,
Capacities are 2



Set lower bound of edge as 1
If the course is compulsory for the student
Other lower bounds are 0

Problem

For class 2022, students want to graduate. But they need to meet following conditions:

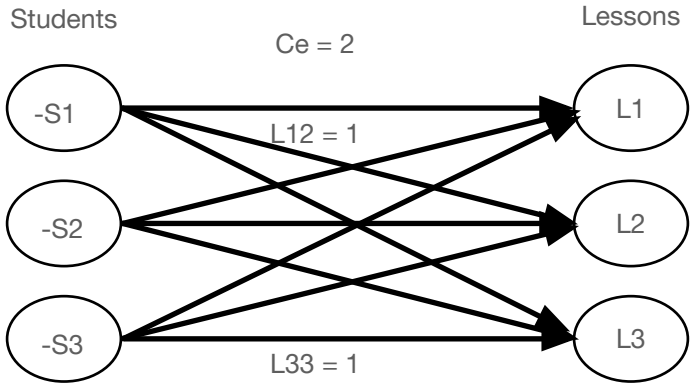
1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
   Repeat attendance can regard as different people

Students node demand:
The class count requirement for students

Lessons node demand:
The particular people number requirement

Link all student nodes and lesson nodes,
Capacities are 2

Set lower bound of edge as 1
If the course is compulsory for the student
Other lower bounds are 0



Then solve the circulation with lower bounds problem using the previous method.
If there is a valid flow for it, all student can graduate.