CSCI 570

Exam 2 Review Slides

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.

Dynamic Programming

Guanyang Luo

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.

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

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?

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

Dynamic Programming

Tiancheng Jin

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)>=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 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 OPT(i,k), or not?
- **Question 5:** Define OPT(0)=0, as when we have no seats, we can’t arrange any student. What is OPT(1)? What is OPT(2)?
    - OPT(1) = w(1), OPT(2) = max{ w(1), w(2) }

Third Step: Find the Solution for 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 OPT(i) in terms of w(i) and OPT(j) for any j < i.
  - OPT(i) = OPT(i-1)

Seat	…	i-2	i-1	i
OPT		8	10	?

Third Step: Find the Solution for 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 OPT(i) in terms of w(i) and OPT(j) for any j < i.
  - OPT(i) = OPT(i-2) + w(i)

Seat	…	i-2	i-1	i
OPT		8	10	?

Third Step: Find the Solution for OPT(i)

- **Question 8:** Put these two cases together, you will get the formula OPT(i) = max{ OPT(i-1), OPT(i-2) + w(i)}. Describe an efficient algorithm to compute the maximum possible comfort you can achieve.
- Do not forget:
  - **Initialization and boundary condition:** “We 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) }.”
  - **Return**: “Return OPT(N) from the array as the maximum possible comfort.”
- According to the requirements, you may have to provide pseudo code or analyze the time/space complexity.

Tabulation Models for Linear DP

- O(i) sub-problems involved to compute **OPT(i)**
  - Recurrence example: OPT(i) = min{ OPT(j) + w(j,i): 1<=j<i }
  - Problem example: P1 of HW6, O(n^2) solution of Longest increasing subsequence (LIS), Rod Cutting Problem.
  - w(j,i) may be difficult to compute, e.g., takes O((j-i)^2) to compute
- O(1) sub-problems
  - OPT(i) = min{ OPT(j) + w(j,i): i-k<=j<i } like this problem. P1 of HW7.
- O(log(i)) sub-problems?
  - Via binary search: O(nlogn) solution of LIS.
  - Via heap/segment-tree/red-black-tree or other data structures. (e.g. Dijkstra Algorithm)

Tabulation Models for 2-Dimensional DP

- O(i) sub-problems involved to compute **OPT(i,j)**
  - Recurrence example 1: OPT(i,j) = min{ OPT(i-1,k) + w(i,j,k): 1<=k<j }.
  - Recurrence example 2: OPT(i,j) = min{ OPT(k,j) + OPT(i,k+1) + w(i,j,k): i<=k<i } like P4 of HW6 and P3/5 of HW7. Be careful of the order to solve the sub-problems.
- O(1) sub-problems
  - Example: P3 of HW6 and P4 of HW7. The Edit distance Problem, OPT(i,j) = min{OPT(i-1,j) + a(i), OPT(i,j-1) +b(j), OPT(i-1,j-1)+w(i,j)}.

OPT	**j-2**	**j-1**	**j**
i-1	8	11	10
i	6	5	?

Take-home Message

- 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.

Application of Network Flow

Bhaskar Goyal

T/F Question

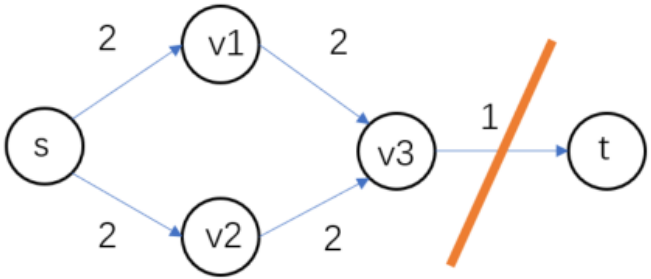
For any edge e that is part of the minimum cut in G, if we increase the capacity of that edge by any integer $k > 1$, then that edge will no longer be part of the minimum cut.

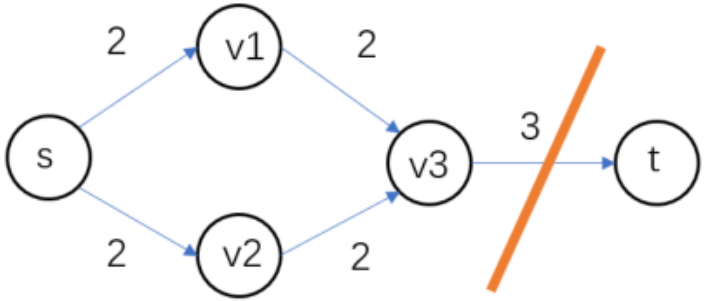
T/F Question

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.

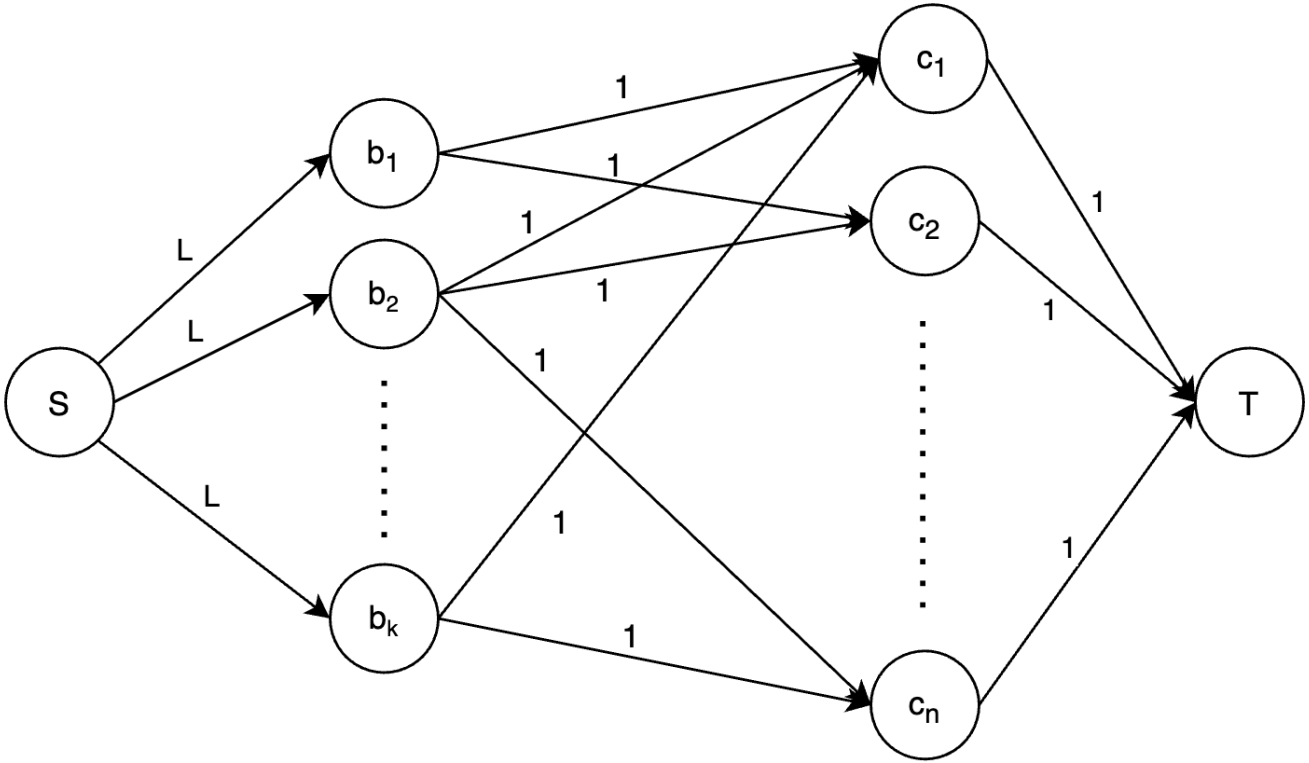
For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter *r* - a client can only be connected to a base station that is within distance *r*. There is also a load parameter *L* - no more than *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.

Additional Variations

1. What happens when there are 2 kinds of base stations, 4G and 5G?
2. 5G mobiles can only connect to 5G stations.
3. While 4G mobiles can connect to both 4G and 5G stations.
4. In addition because of some technical issues, no more than K 4G phones can be connected to any 5G base station.

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

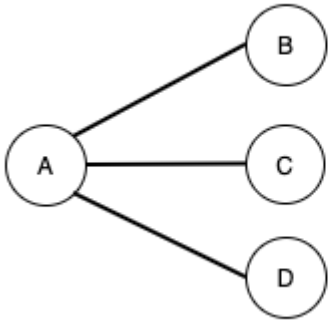
The edge connectivity of an undirected graph G = (V, E) is the minimum number of edges that must be removed to disconnect the graph (or minimum number of edges that must be removed to split graph into 2 sub graphs). For example, the edge connectivity of a tree is 1.

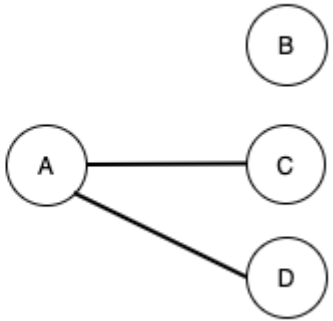
(a) Show how the edge connectivity of an undirected graph G with n vertices and m edges can be determined by running a maximum-flow algorithm.

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

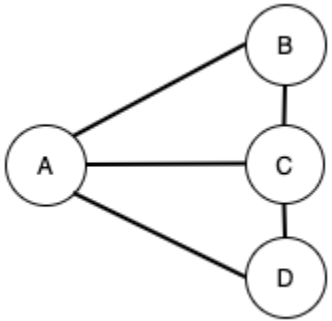
Example: Edge Connectivity

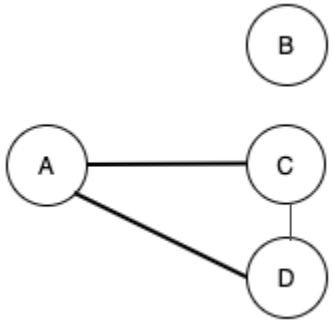
k = 1





k = 2



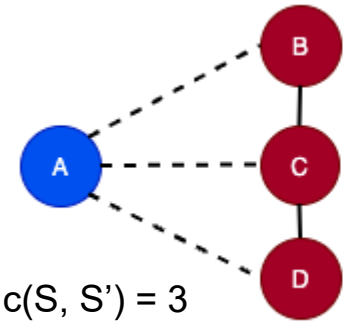


Solution: Edge Connectivity

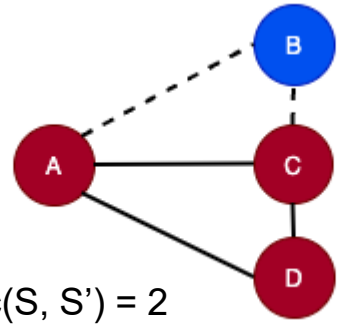
Let V be the set of n vertices.

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

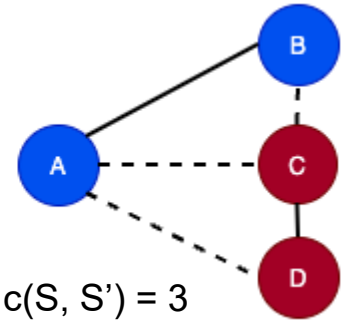
By definition, the edge connectivity, $k = min \{ c(S, S') \}$ *where* $S \subset V$.
(Meaning that for any 2 mutually exclusive set of vertices formed by a cut, the minimum number of edges required to disconnect graph is the minimum number of edges crossing that cut.)



c(S, S') = 3



c(S, S') = 2



c(S, S') = 3

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

= 2

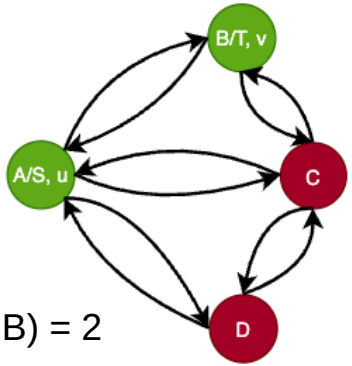
Solution: Edge Connectivity

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,

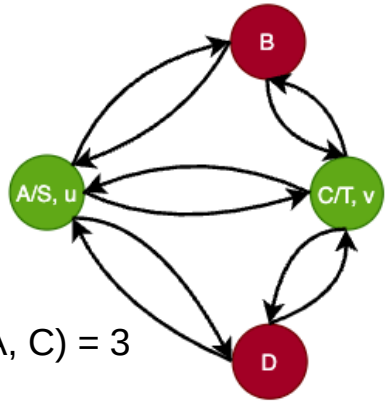
Fix a vertex u ∈ V. For every cut (S, S') in G*, there is a vertex v ∈ V such that u and v are on either side of the cut.

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

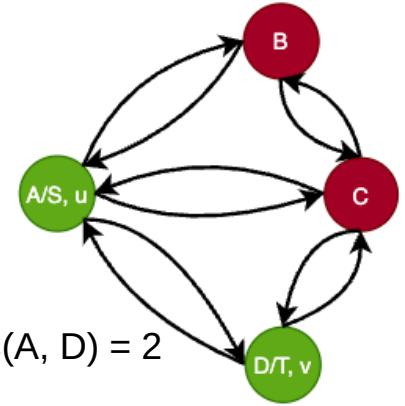
Thus, *K = min C(u, v) where v∈V, v ≠ u*



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.

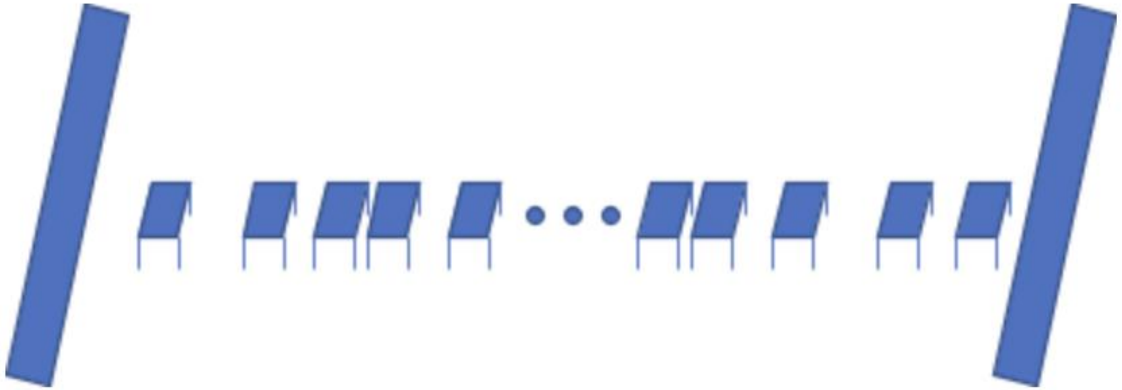
Therefore, there will be n-1 max flow computations.

Ford-Fulkerson and Capacity Scaling

Prabudh Gupta

Question 1:

A number of people have gotten themselves involved in a very dangerous game called the octopus game. At this stage of the game, they need to pass a river. The river is 100 feet wide, but contestants can only jump k feet at most, where k is obviously less than 100. To help contestants cross the bridge there are platforms placed along a straight line at integer distances from one end of the river to the other end at various distances where the distance between any two consecutive platform is less than k. But the problem is that each platform can only be used once. If another contestant tries to use it for the second time it will break.



Solution:

a) Design a network flow based solution to determine the maximum number of contestants that can safely cross this river and live to play in the next stage of the game.

The construction of the network can be done as follows:

1. Create a source ‘S’ and sink node ‘T’ and create nodes that would represent the platforms (lets assume we call these nodes {A…….N})
2. For every platform node {A….N} create a dummy node {A’,B’.....N’} respectively. Connect all the platform nodes {A…….N} to the corresponding dummy nodes {A’…….N’} so that the edge A -> A’ represents platform A and so on. Set the capacity to **1** for all these edges. This part is the most crucial in the construction to make sure that a platform can not be used by more than one contestants.
3. Connect the source ‘S’ to the platform nodes that are at max **k** feet distance from it. The capacity of any such edge could be set to anything >= 1. This is to make sure that all the inflow to the nodes is at platform node {A,B...N}.
4. Connect the dummy nodes {A’,B’...N’} to **all** the platform nodes (or the sink) that are at max **k** feet distance from it. HOWEVER, it suffices to add such edges in only the forward direction. This also makes sure the outflow at each platform is from the dummy nodes {A’,B’,....N’}

Final answer: Once the network is constructed, run a max flow algorithm on it (say FF) and the max flow value of the network obtained will be the maximum number of contestants that can cross the river.

Solution:

b) Prove that your solution is correct

Claim : The max flow of the network will give the number of contestants that can get across the river

Forward Claim : If there is a flow of value V, we can send V people across

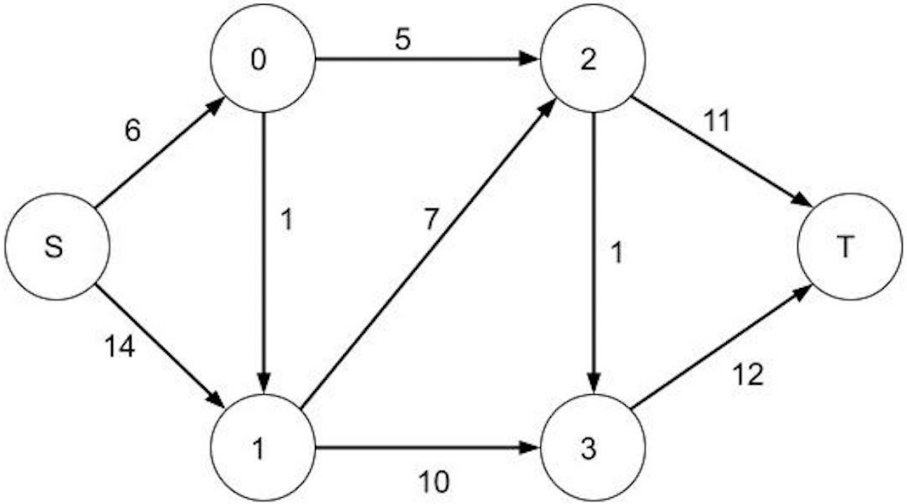
Proof : Since k<100, any S-T path must pass via some platform (say p), and thereby has a bottleneck of 1 due to the edge p - > p'. Thus any s-t path can have a flow of at most 1. Thus, if the flow value is V, there will be V paths each having a flow of 1. Further, these cannot share any edge p -> p' due to its capacity of 1. Thus, we can assign each such path to a contestant and none of them will use the same platform, thus allowing us to send V people safely.

Backward Claim : If we can send V people across, we can have a flow of value V

Proof : For 1 person crossing the river, we get a path going from s to t such that the person jumps at most k feet at a time. These V paths must be node-disjoint, since no platforms can be repeated. If we send a flow of 1 unit down each corresponding path in the network, we won't exceed any capacity constraints and this flow will have a value of V.

Question 2

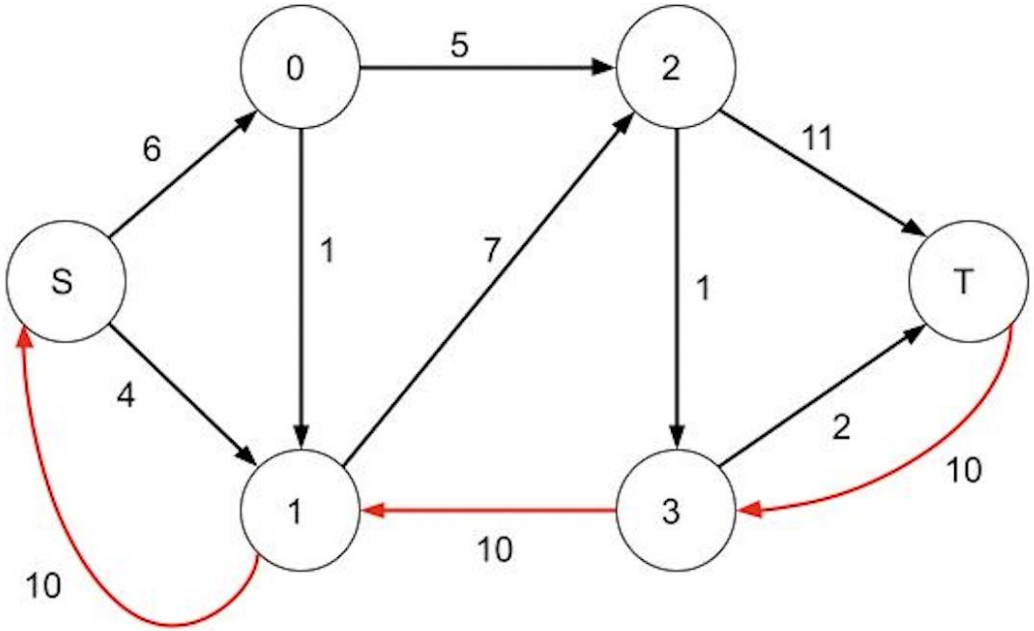
Using Capacity scaling calculate the max-flow of the following given network:



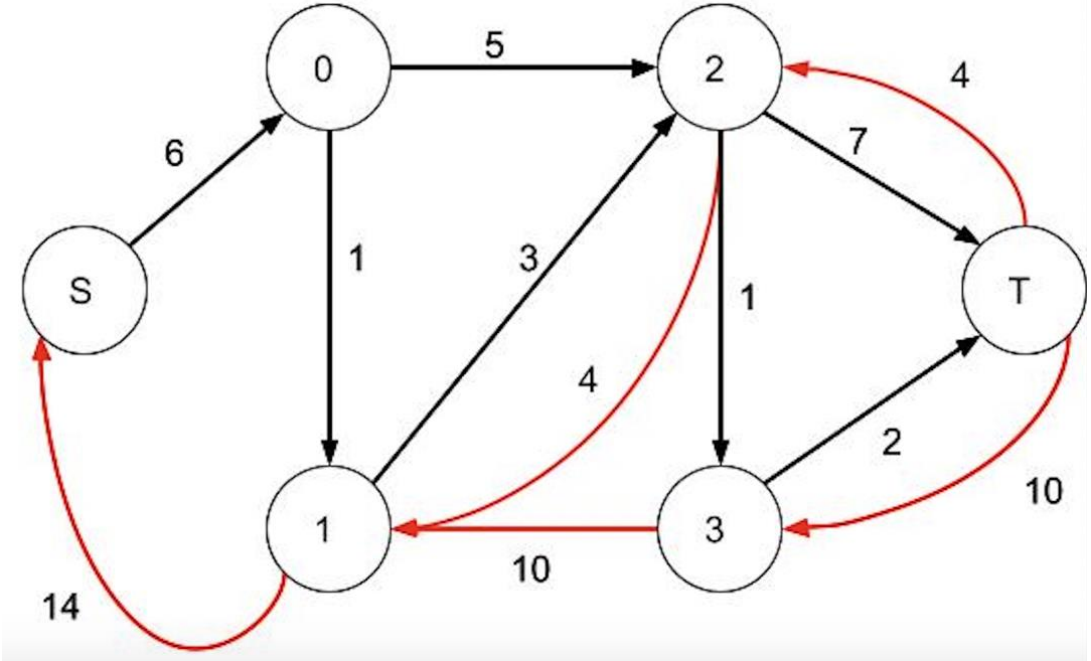
Solution

U = 14

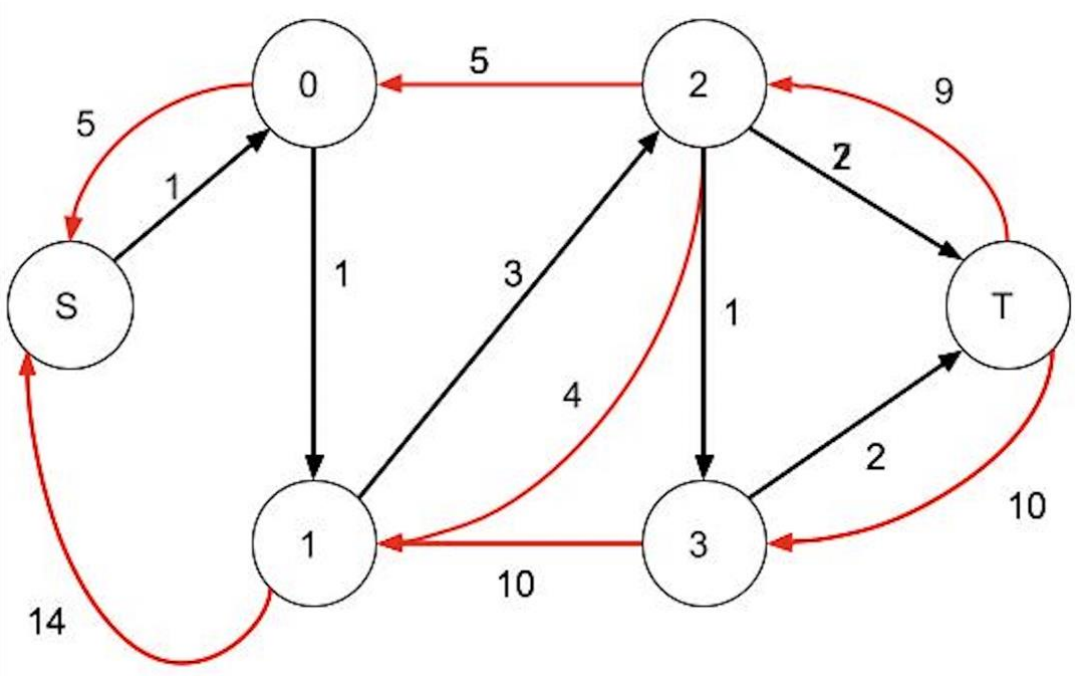
At Δ = 8 phase we have 1 augmenting path.



At Δ = 4 phase we have 2 Augmenting Paths

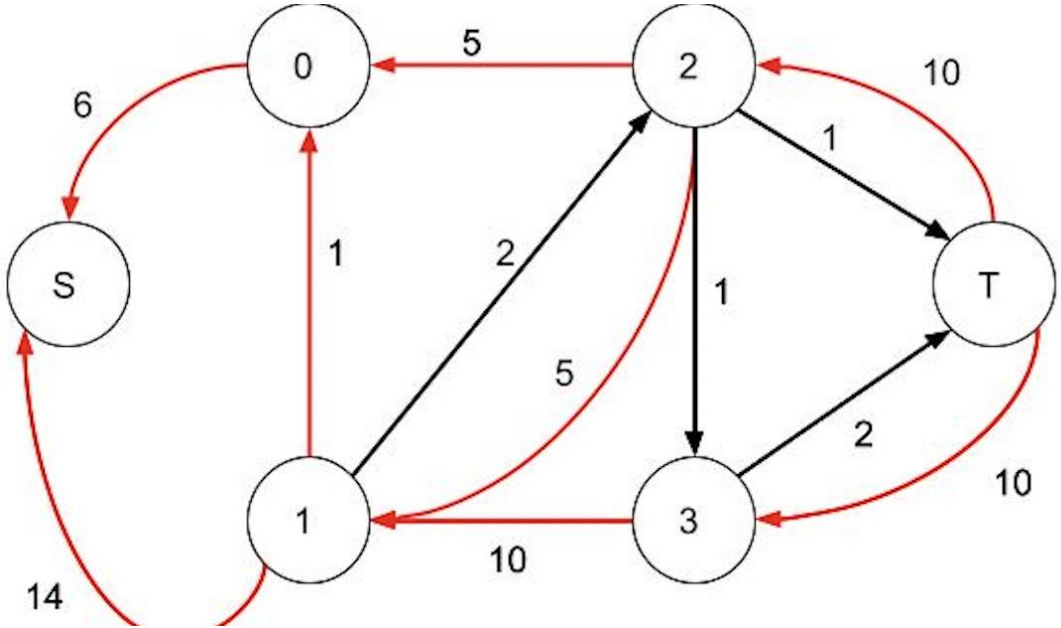


At Δ = 4



Δ = 2, No possible Augmenting Paths

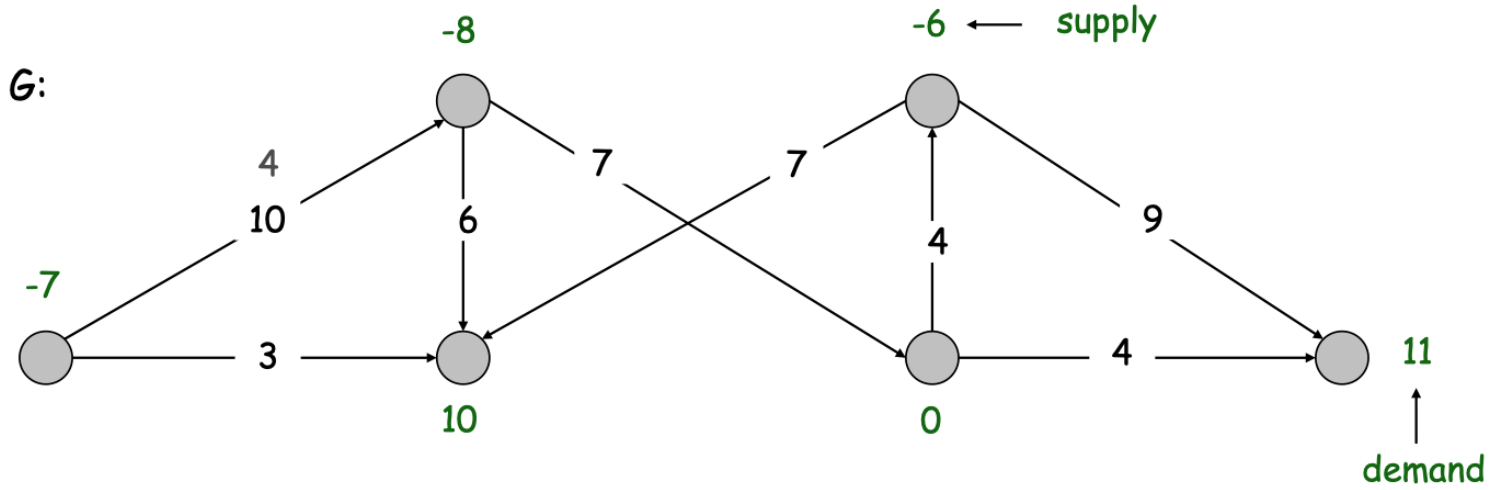
At Δ = 1, 1 Augmenting path possible

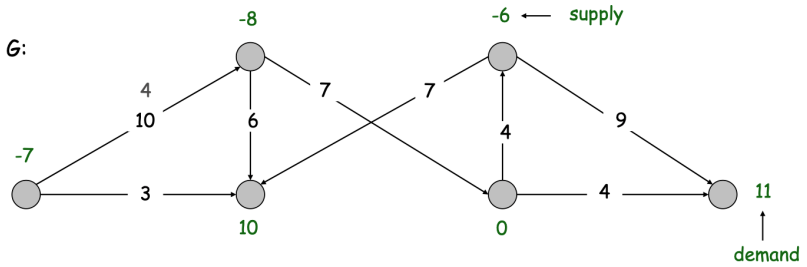


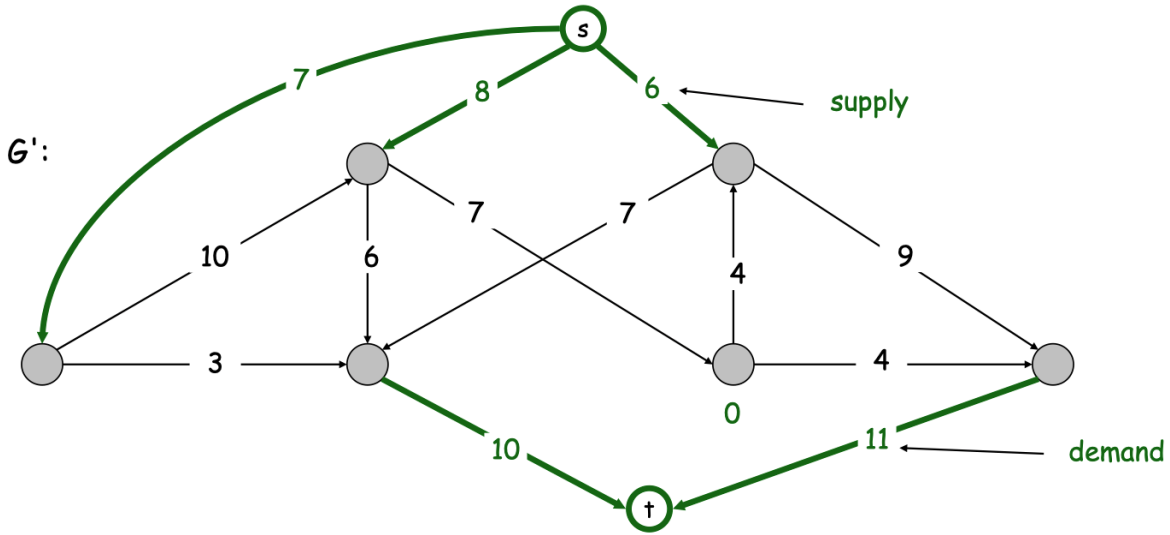
Δ = 0, Algorithm Terminates , Max-Flow = 20

Circulation &
Circulation with lower bounds

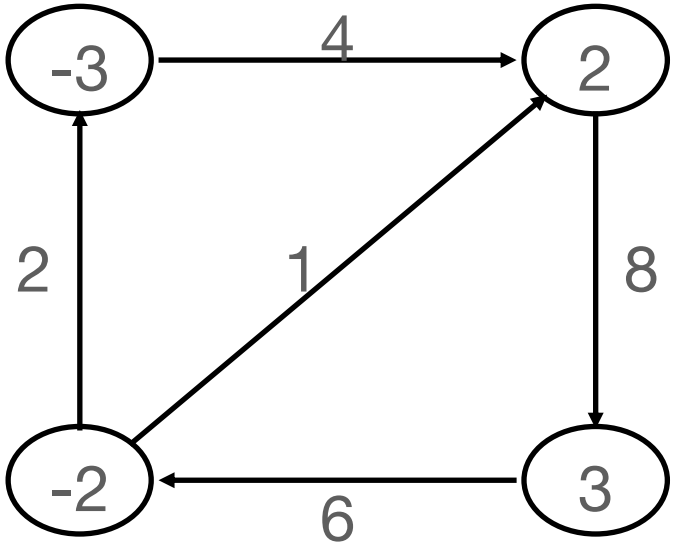
Mehrnoosh Mirtaheri


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

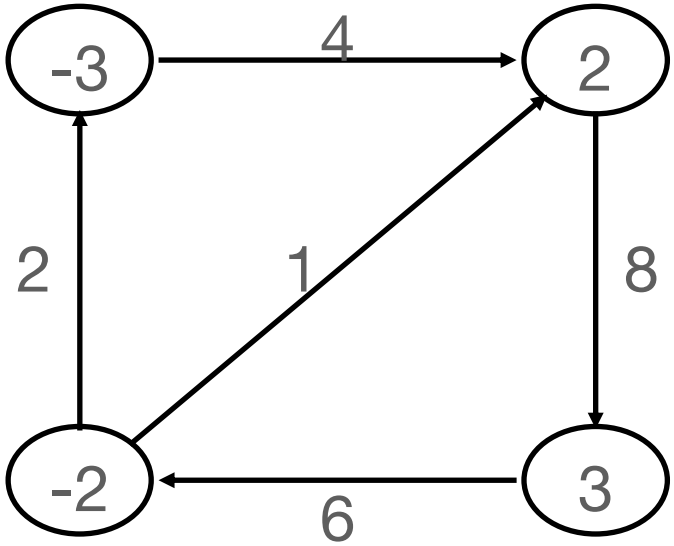

G:
-8
-6
supply
4
10
7
7
6
4
9
-7
3
10
0
4
11
demand


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

Circulation



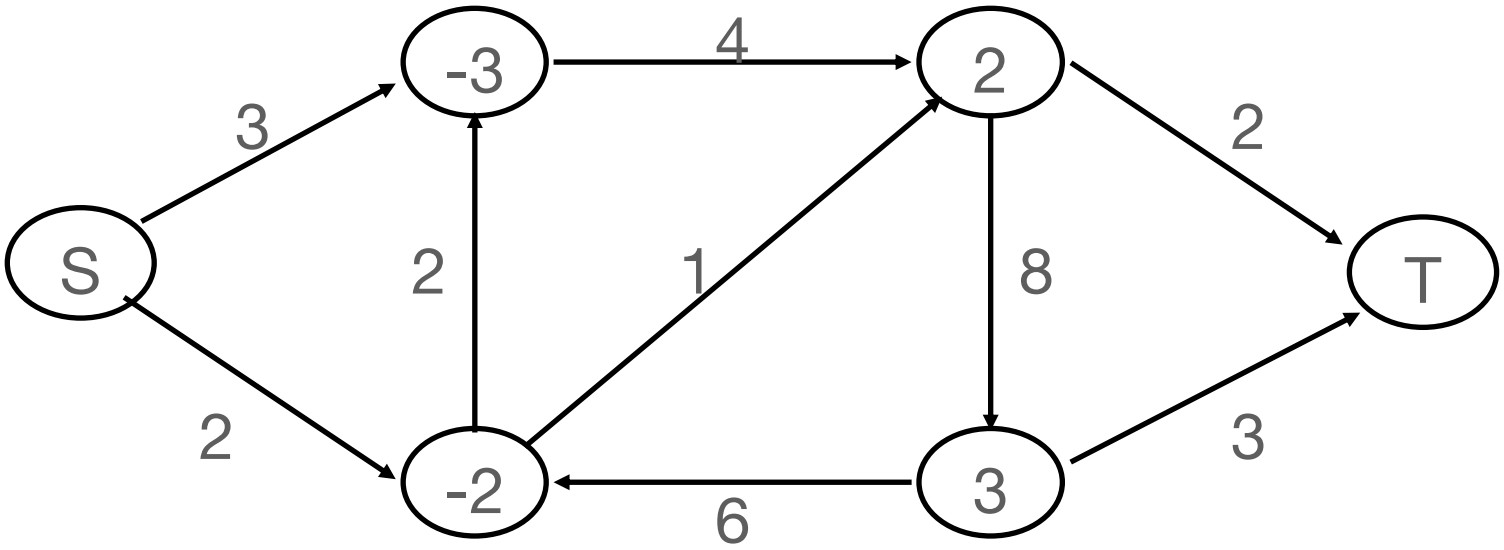
Circulation



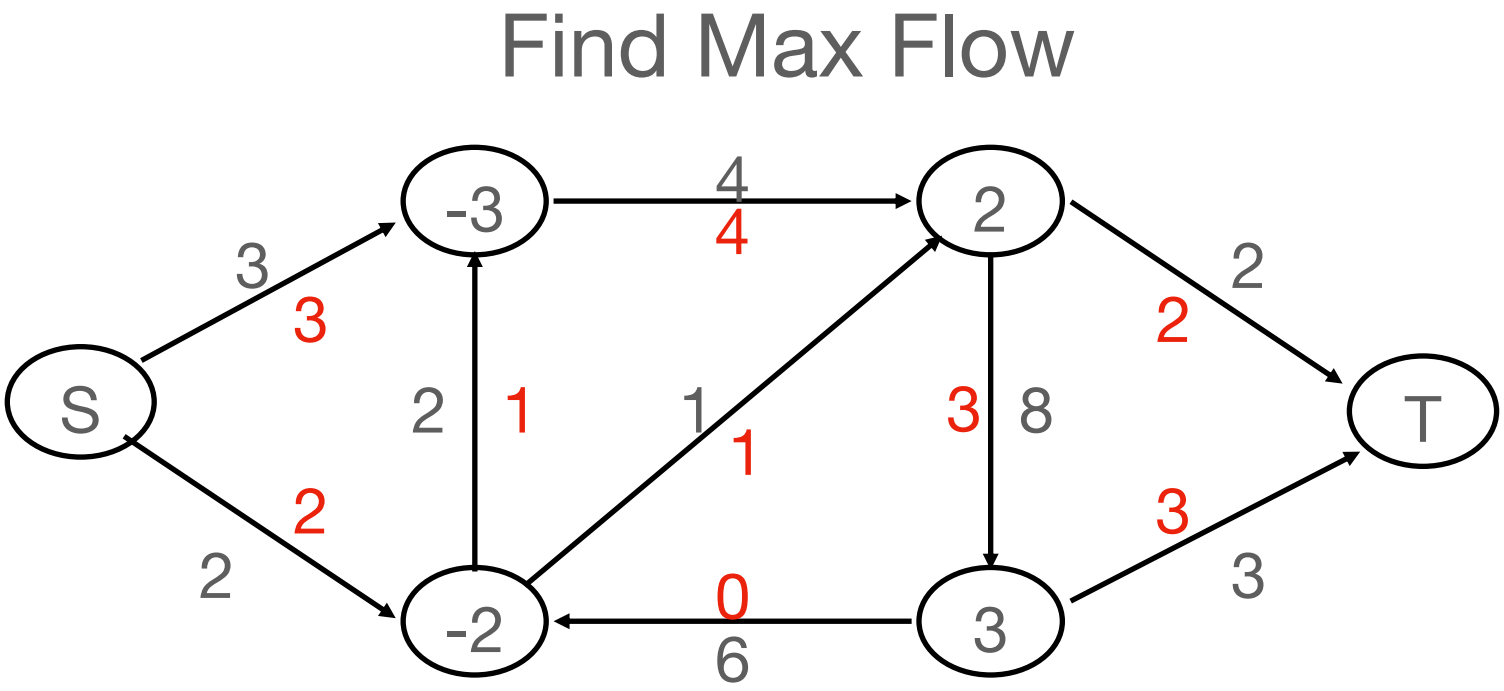
Check the demand value.
Sum of all demand values = 0

Circulation

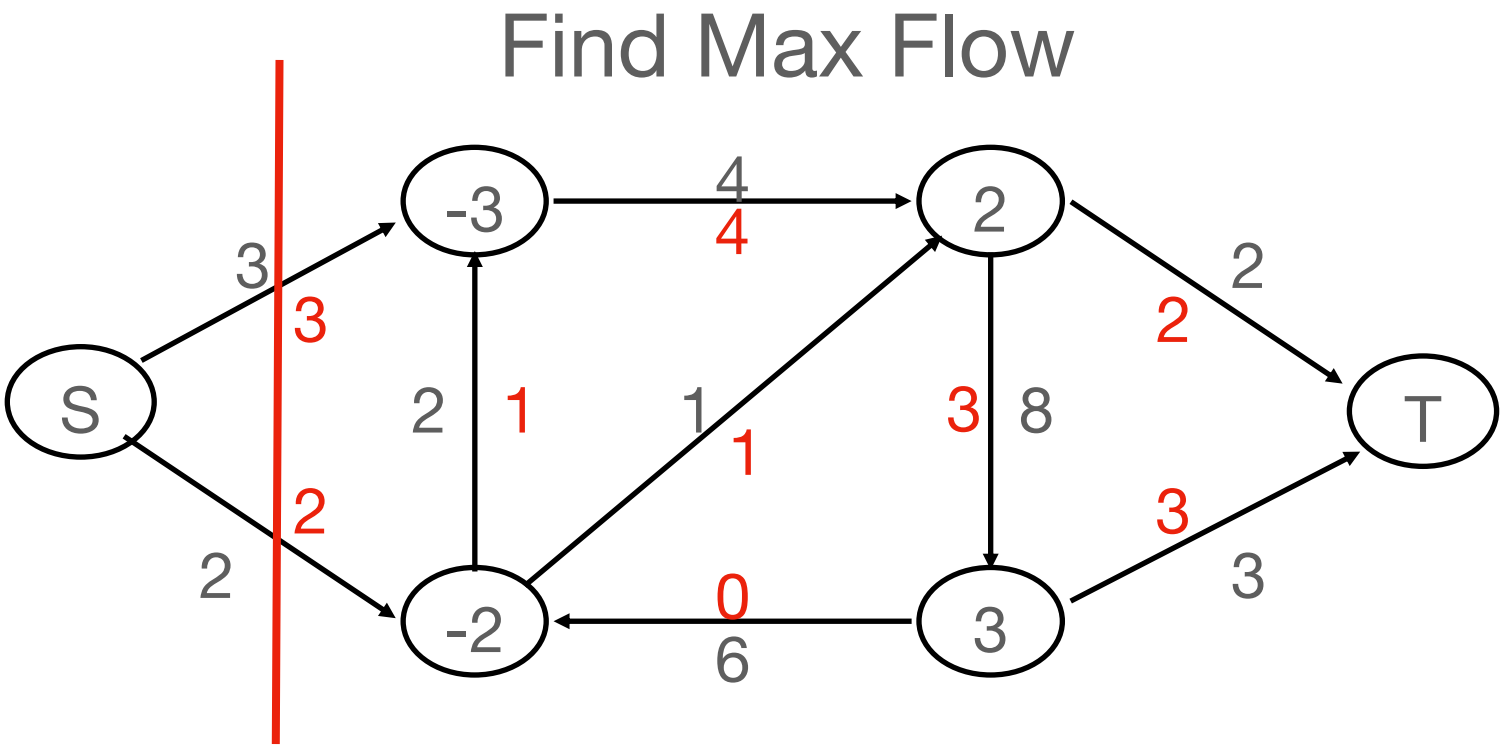
Add S and T



Circulation



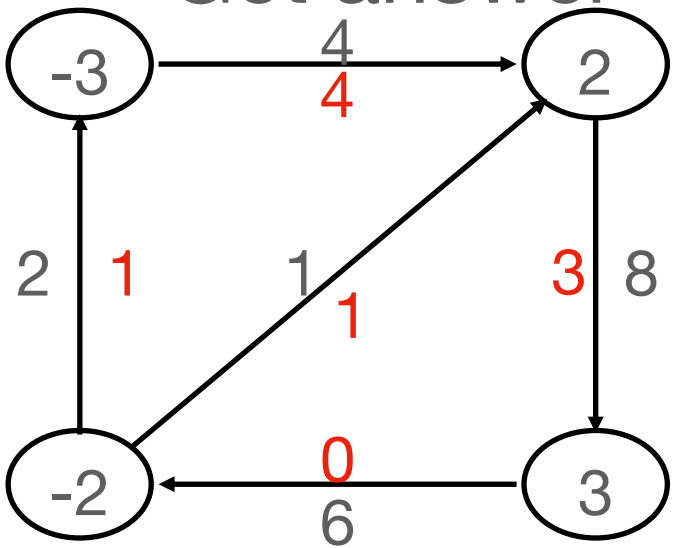
Circulation



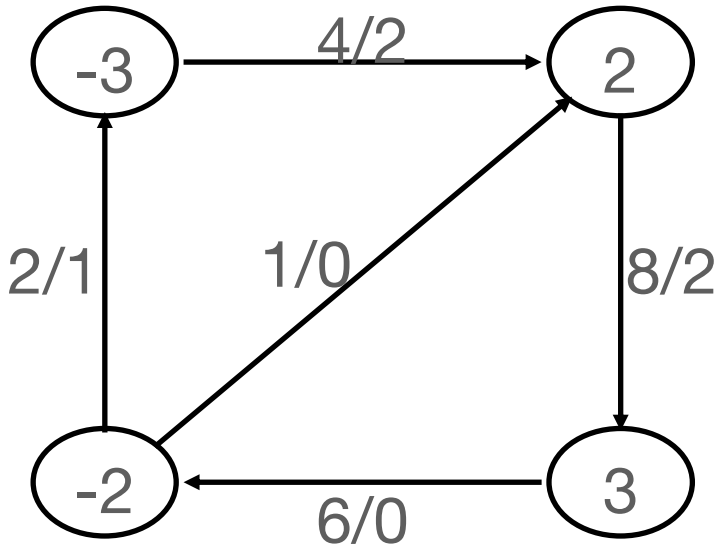
Check if all edges out from S are fully used (Capacity = flow)

Circulation

Remove S and T
Get answer



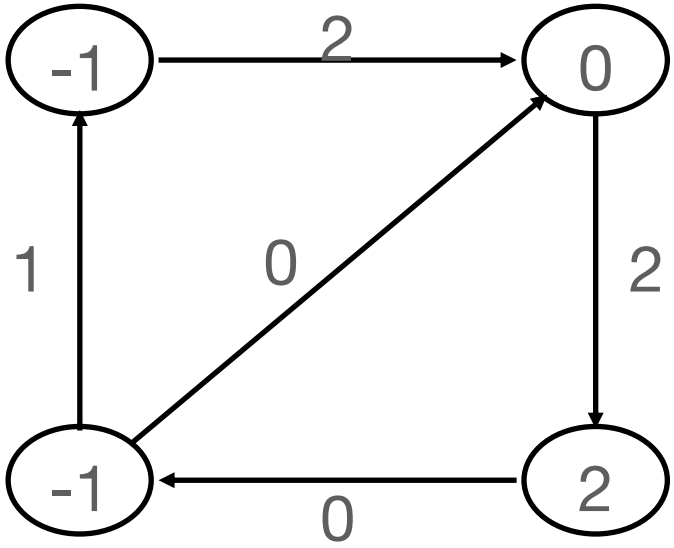
Circulation with Lower Bounds



G

Capacity/Lower bound

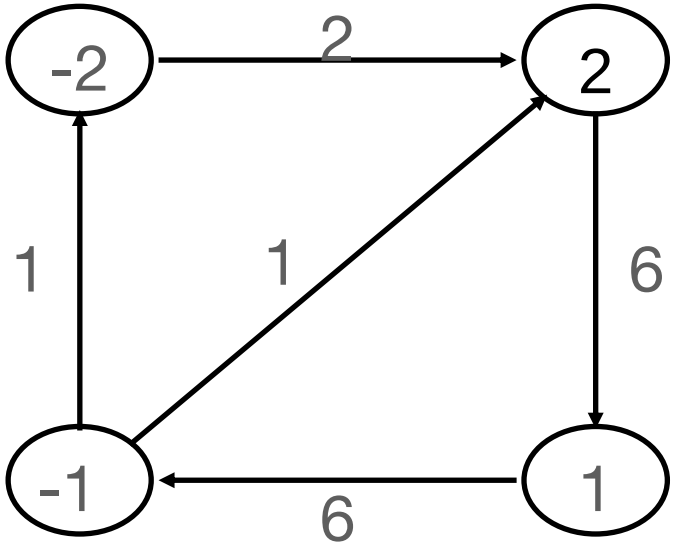
Circulation with Lower Bounds



f0 = smallest flows satisfy the lower bounds

Don’t forget change demand in node

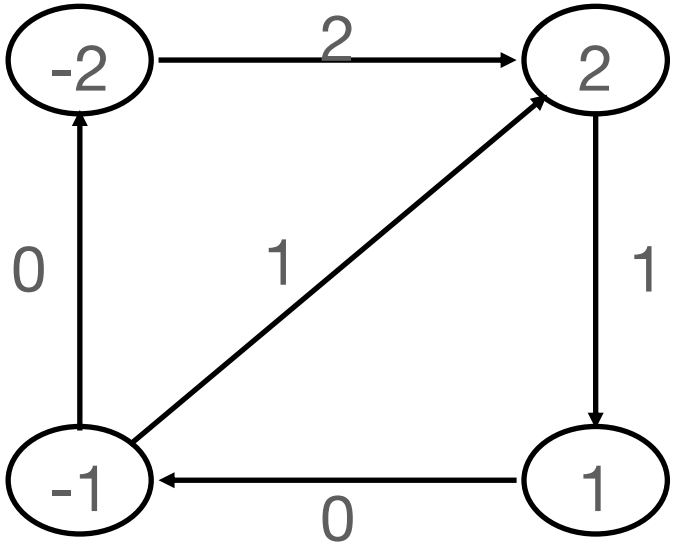
Circulation with Lower Bounds



G’ = G - f0

Both in edge and node

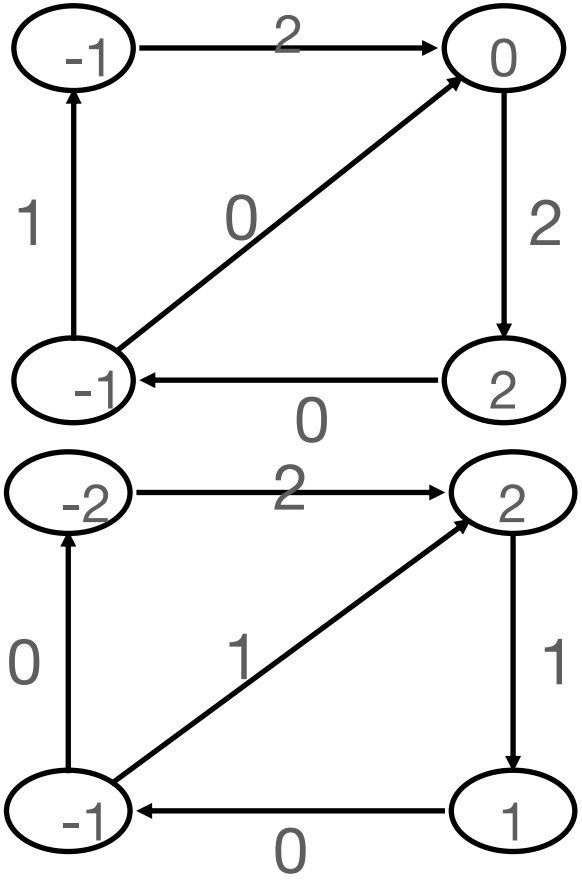
Circulation with Lower Bounds

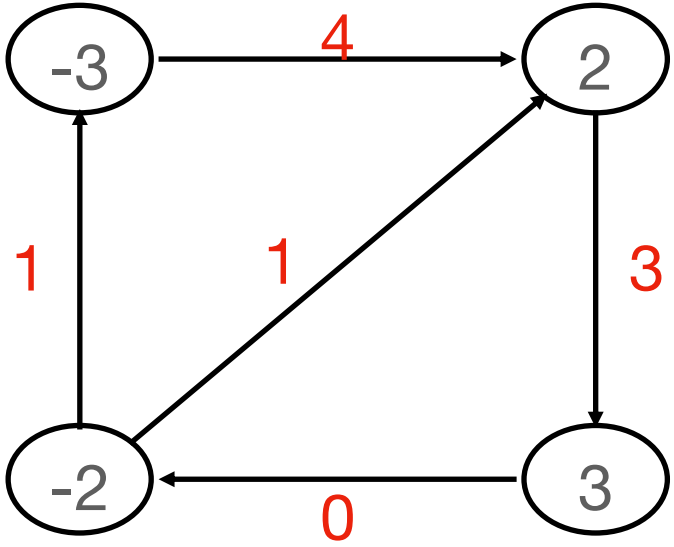


f1 = find max flow in G’

Use the same method mentioned above

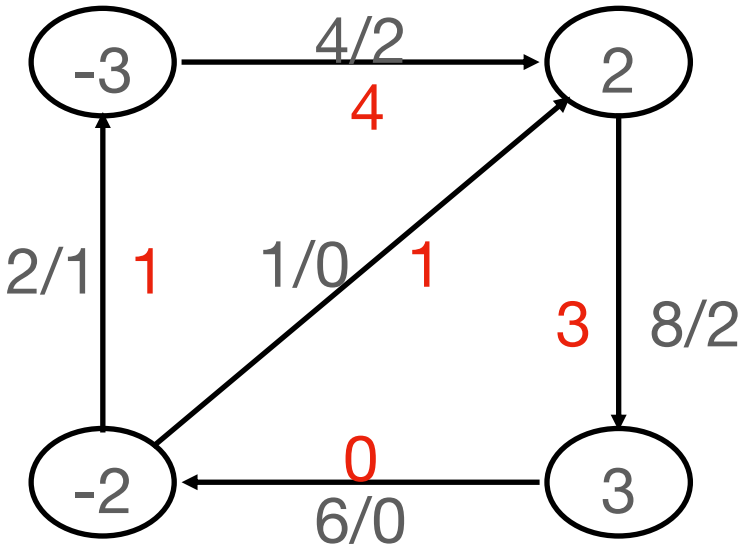
Circulation with Lower Bounds





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…]

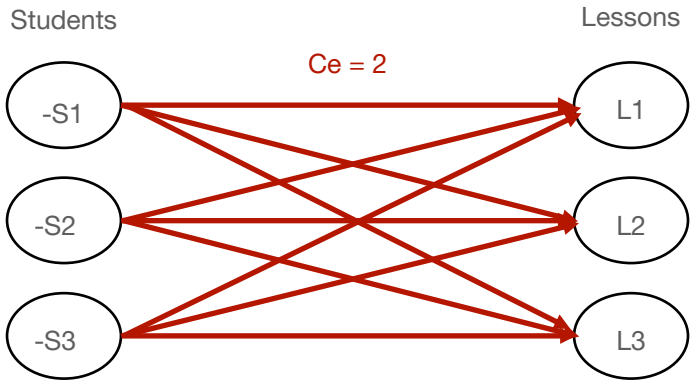
Problem

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:

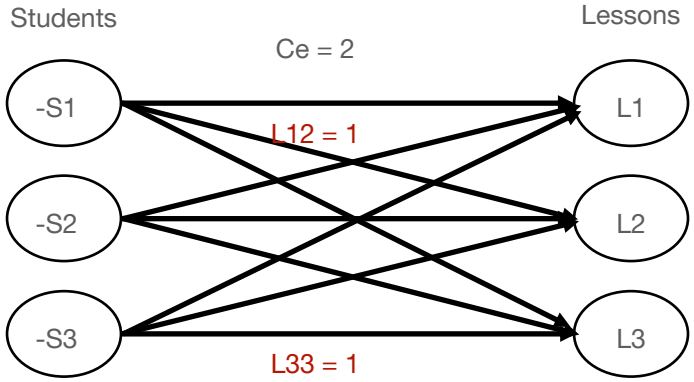
1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
 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

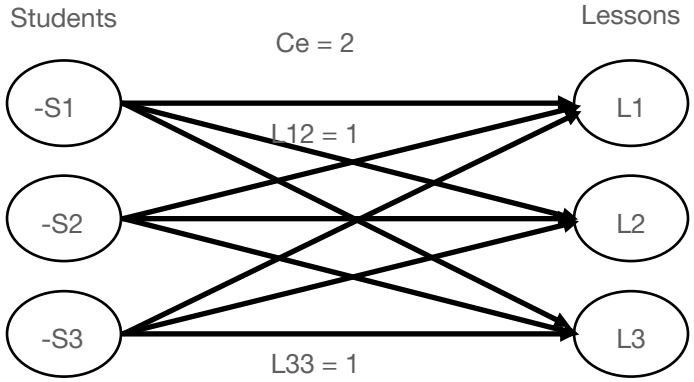
1. For each student, he/she must take a certain number of courses.
2. For each student, he/she must take specific compulsory courses.
3. For each course, it needs a certain number of people to start the course, no more or less
4. Each course can be taken by one student repetitively at most 2 times
 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