Exam 2 review

Dynamic Programming 1

Bruno Segovia

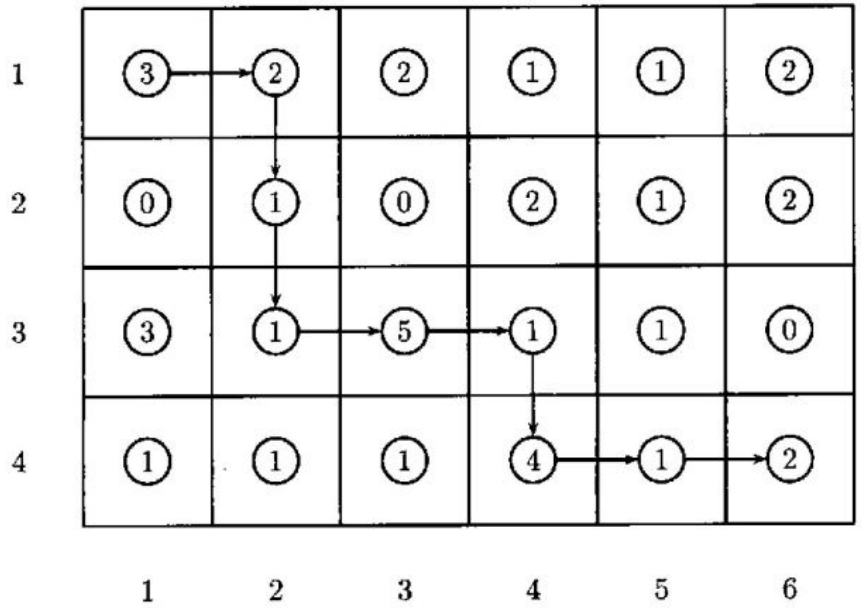
OH: MW 12–2pm

DP.1 Problem #1

You are given an n x m grid. You start in the top-left corner, cell (1,1), and in each step you move either down or right. Eventually, you end up in the bottom-right corner, cell (n, m).

Each cell (i, j) also has a set number of gold coins x_{ij} on it.

Find the down-right path from (1,1) to (n, m) that maximizes the total number of gold coins collected.



DP.1 Solution #1

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

Let OPT(i, j) be the optimal number of gold coins collected on a down-right path from (1,1) to (i,j).

DP.1 Solution #1

b). Write a recurrence relation for OPT(i, j). Be sure to state base cases.

“If we have the optimal solution at this step, what must have been the previous step?”

Consider OPT(*n*, *m*). We may say it equals OPT(optimal predecessor tile) + x_{nm}.

What are the possible predecessor tiles? Since only down-right moves are allowed, the predecessor was either above it or to its left: either (*n*-1,*m*) or (*n*,*m*-1).

Which predecessor is optimal? Clearly whichever has the higher OPT.

Thus OPT(*n*,*m*) = max{OPT(n,m-1), OPT(n-1,m)} + x_{nm}.

DP.1 Solution #1

b). Write a recurrence relation for OPT(i, j). Be sure to state base cases.

We can extend this logic: usually, OPT(*i*, *j*) = max{OPT(i,j-1), OPT(j-1,1)} + x_{ij}.

We are almost done, but what if *i* or *j* equal 1? These are your base cases.

Now write an algorithm that implements your overall recurrence relation to fill in an OPT array and returns OPT(*n*, *m*).

DP.1 Solution #1

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

$$\mathrm{OPT}(i,j) = x_{ij} + \begin{cases} 0, & \text{if } (i,j) = (1,1) \\ \mathrm{OPT}(i,j-1), & \text{if } i = 1 \\ \mathrm{OPT}(i-1,j), & \text{if } j = 1 \\ \max(\mathrm{OPT}(i,j-1), \mathrm{OPT}(i-1,j)), & \text{else} \end{cases}$$

Require: real array $x[1,\ldots,n][1,\ldots,m]$ of coins on tiles
Ensure: return best path from $(1,1)$ to (n,m) and its reward

```
a(1,1) ← x_11
pred(1,1) ← None
for j ← 2 to m do
    a(1,j) ← a(1,j−1) + x_1j
    pred(1,j) ← (1,j−1)
end for
for i ← 2 to n do
    a(i,1) ← x_i1 + a(i−1,1)
    pred(i,1) ← (i−1,1)
    for j ← 2 to m do
        a(i,j) ← x_ij + min{a(i−1,j), a(i,j−1)}
        pred(i,j) ← (i−1,j) or (i,j−1) depending on the above minimum
    end for
    a(i,m) ← a(i,m−1) + x_im
    pred(i,m) ← (i,m−1)
end for
this ← (n,m)
path ← []
while this ≠ None do
    path.add(this)
    this ← pred(this)
end while
return path.reverse() and a(n,m)
```

Another way to think about it…

“If we have the optimal solution at this step, what must have been the previous step?”

In the abstract, every DP problem is a “best-path problem.” You are given some grid where each tile has a certain reward (or cost), told there are only certain moves you can use to move from one tile to another, and asked to find the best path (most rewarding/least costly) between the start tile and end tile.

DP.1 Problem #2

You are going on a road trip. You start from position 1 and will end at position n. At each integer position along the road (so positions {1, 2, …, n}), there is a gas station. For each gas station i, you are given a price per gallon $p_i \geq 0$. One gallon of gas is exactly enough to make it one position on the line.

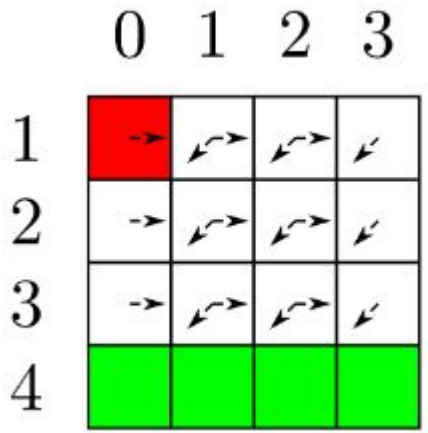
Your car has a gas tank size of $s \leq n$ gallons, which starts out empty with you at the gas station at position 1. You can stop at as many gas stations as you want. Every time you stop, you can decide how many gallons to put in your tank, though the total in your tank can never exceed s. The amount of gas you buy at any station will always be an integer. Your goal is to compute the minimum amount of money you can spend to get to location n.

DP.1 Solution #2

We may also think of this problem as a state grid with a start tile, an end tile, and a prescribed set of possible moves between each tile.

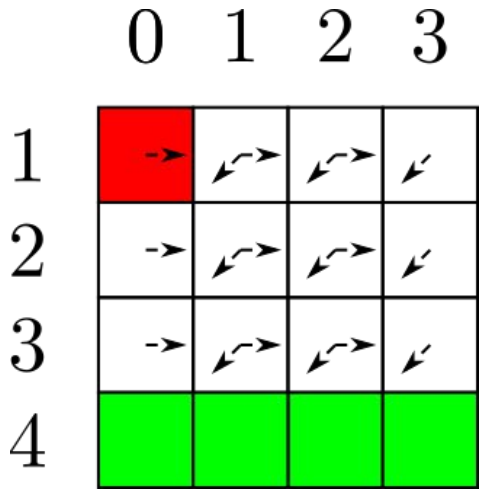
Imagine a grid [1, …, *n*] x [0, …, *s*]. Each tile (*i*, *j*) represents the state “at gas station *i* with *j* gallons of gas left in the tank.”

- We start at tile (1, 0).
- We want to end at tile (*n*, 0) as we do not want to buy any unnecessary gas.
- Given any tile (*i*, *j*), we usually have two “moves” available:
  - Move to the right, i.e. buy a gallon of gas at station *i*.
    - Cost is p_i. Move cannot be taken if you have *s* gallons of gas.
  - Move to the bottom-left, i.e. burn a gallon to get to station *i+1*.
    - Cost is 0. Move cannot be taken if you have 0 gallons of gas.

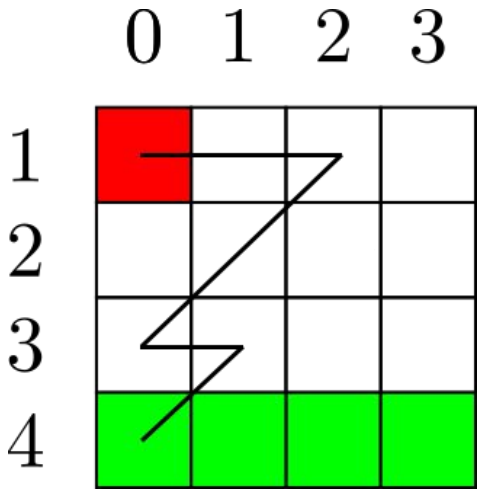


The grid for *n* = 4, *s* = 3. A right move has cost; a diagonal move has no cost.

DP.1 Solution #2



The grid for $n = 4$, $s = 3$. A right move has cost depending on the row; a diagonal has no cost.



An optimal path given $(p_1, p_2, p_3) = (1, 2, 0)$. What is the cost? What sequence of actions does this represent?

DP.1 Solution #2

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

Let OPT(i, j) be the cost of the optimal path from (1,0) to (i, j):
that is to say, the minimum cost necessary to, starting at station 1 with 0 gallons of gas, wind up at station i with exactly j gallons of gas left in the tank.

DP.1 Solution #2

b). Write a recurrence relation for OPT(i, j). Be sure to state base cases.

“If we have the optimal solution at this step, what must have been the previous step?”

We have OPT(i, j) = OPT(optimal predecessor) + cost(transition from opt. pred.).

Since there are only two possible moves, right or left-down, you can generally say that for any (i, j), the two predecessors are “came from the left” or “came from the top-right”. Clearly whichever minimizes the above expression is optimal.

DP.1 Solution #2

b). Write a recurrence relation for OPT(i, j). Be sure to state base cases.

Coming from the left means coming from (i, j-1) with cost p_j. Coming from the top-right means coming from (i-1, j+1) with cost 0. Thus we have

OPT(i, j) = min (OPT(i, j-1) + p_j, OPT(i-1, j+1) + 0)

We are almost done, but what if i or j equal 1? These are your base cases.

Now write an algorithm that implements your overall recurrence relation to fill in an OPT array and returns OPT(n, 0). (Left as an exercise for the reader.)

DP.1 Solution #2

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

$$\mathrm{OPT}(i,j) = \begin{cases} 0, & \text{if } (i,j) = (1,0) \\ \mathrm{OPT}(i,j-1) + p_i, & \text{else if } i = 1 \text{ or } j = s \\ \mathrm{OPT}(i-1,j+1), & \text{else if } j = 0 \\ \min\{\mathrm{OPT}(i-1,j+1), \mathrm{OPT}(i,j-1) + p_i\}, & \text{else} \end{cases}$$

Require: real array $p[1,\ldots,n]$ of gas prices per gallon, tank capacity $s \leq n$
Ensure: return best path to get from station 1 with no gas to station n

```
a(1,0) ← 0
pred(1,0) ← None
for j ← 1 to s do
    a(1,j) ← a(1,j−1) + p_1
    pred(1,j) ← (1,j−1)
end for
for i ← 2 to n do
    a(i,0) ← a(i−1,1)
    pred(i,0) ← (i−1,1)
    for j ← 1 to s−1 do
        a(i,j) ← min{a(i−1,j+1), a(i,j−1) + p_i}
        pred(i,j) ← (i−1,j+1) or (i,j−1) depending on the above minimum
    end for
    a(i,s) ← a(i,s−1) + p_i
    pred(i,s) ← (i,s−1)
end for
this ← (n,0)
path ← []
while this ≠ None do
    path.add(this)
    this ← pred(this)
end while
return path.reverse() and a(n,0)
```

Dynamic Programming 2

Parth Goel

Number of Valid Seating Arrangements

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

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

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

Describe a dynamic programming solution to solve this problem.

Number of Valid Seating Arrangements

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

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

Number of Valid Seating Arrangements

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

Solution:

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

Number of Valid Seating Arrangements

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

Solution:

Initialize array f.

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

For i>2 to n :

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

return f[n]

Number of Valid Seating Arrangements

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

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

Number of Valid Seating Arrangements

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

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

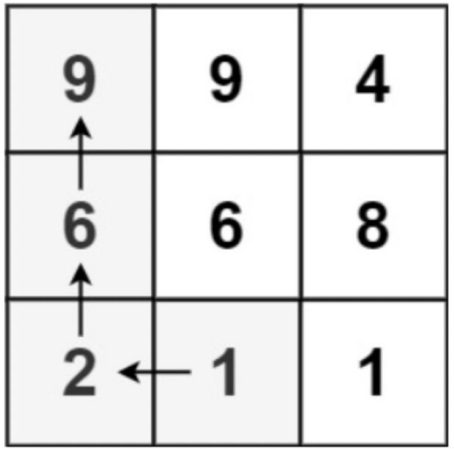
Longest Increasing Path in a Matrix

Given an m x n integers matrix, return *t*he length of the longest increasing path in matrix.

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

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

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



Longest Increasing Path in a Matrix

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

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

Longest Increasing Path in a Matrix

```
if (j < m - 1 and (cell[i][j] < cell[i][j + 1])
      v1 = 1 + Longest(i, j + 1, cell, dp,n,m);

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

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

   if (i < n - 1 and (cell[i][j] < cell[i + 1][j])
      v4 = 1 + Longest(i + 1, j, cell, dp,n,m);
```

Longest Increasing Path in a Matrix

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

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

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

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

            ans = max(ans, dp[i][j]);
        }
    }
```

Longest Increasing Path in a Matrix

The steps of this approach involve:

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

Longest Increasing Path in a Matrix

Complexity Analysis

Time complexity

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

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

Space complexity

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

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

Network Flow

Hikaru Ibayashi

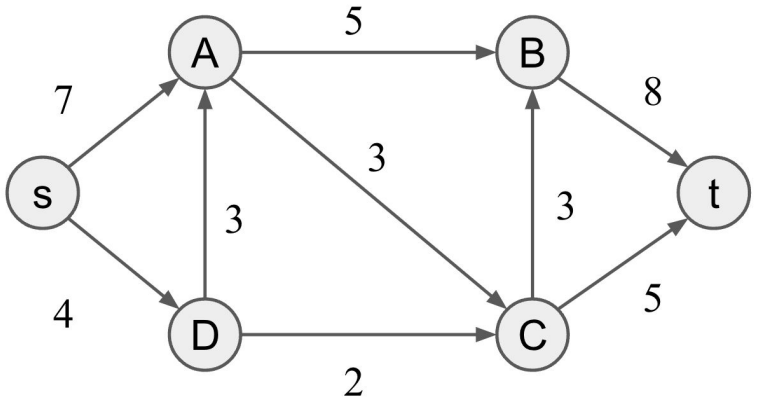
Max-Flow Problem

- A problem to find maximum flow from s to t in a given network
- Ford–Fulkerson algorithm is the most basic approach
  - Iteratively **augment flow**

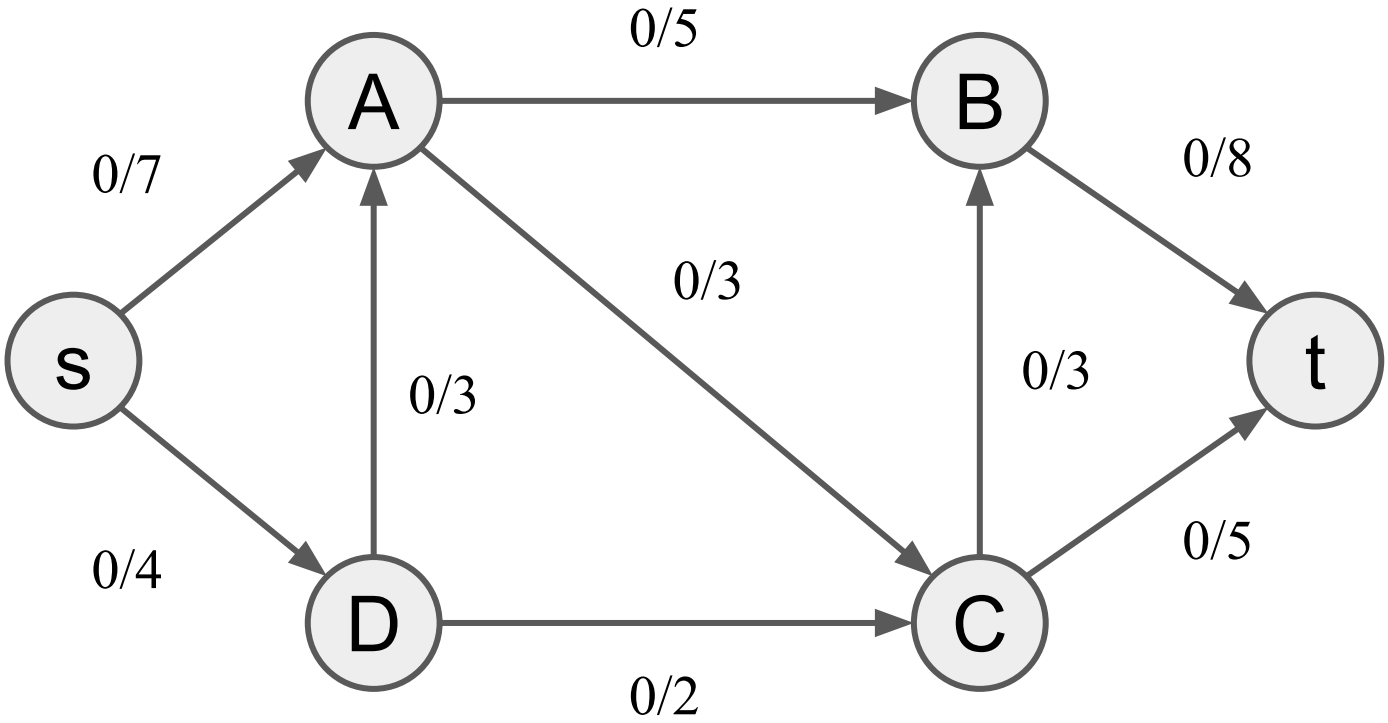
    and construct **a residual graph**
  - Time-complexity: O(C*|E|)

    Note: This is **exponential**.
- Easy to modify Ford-Fulkerson efficient

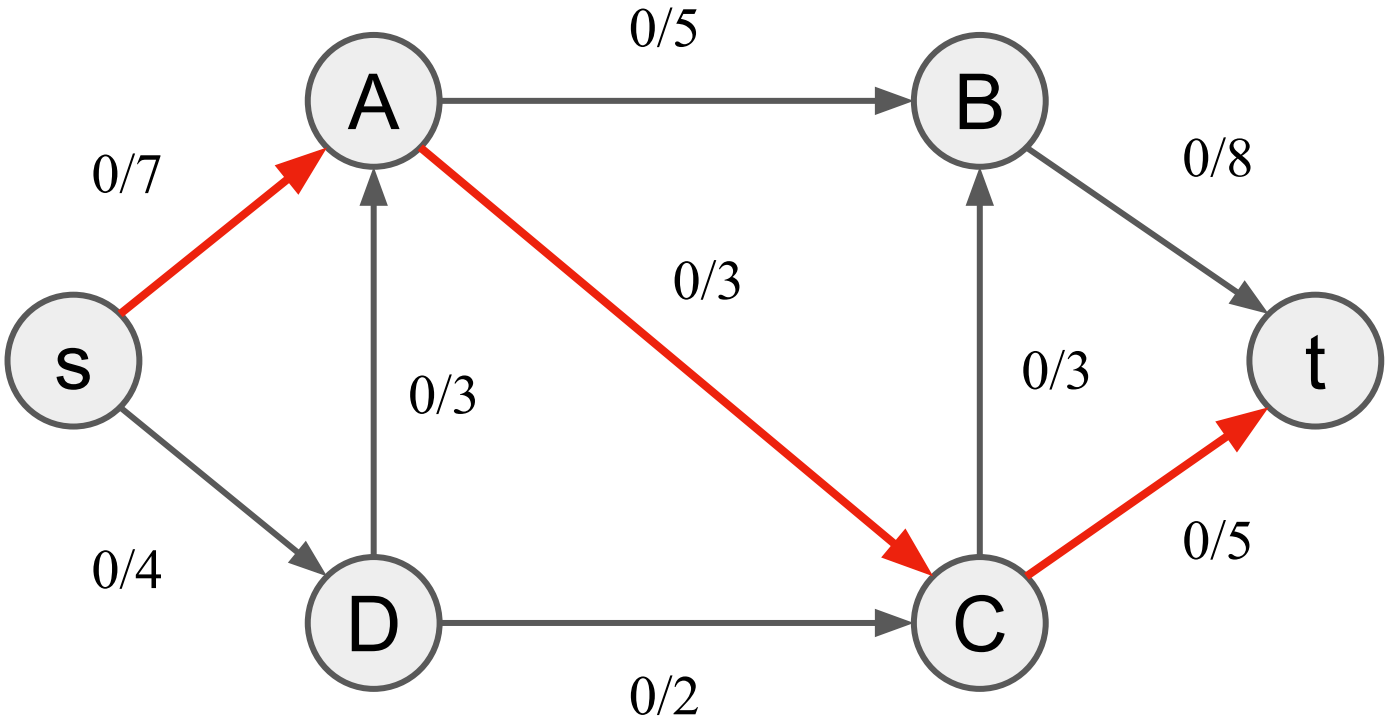
  (e.g. Capacity-Scaling, Edmonds-Karp, etc)



Network Flow

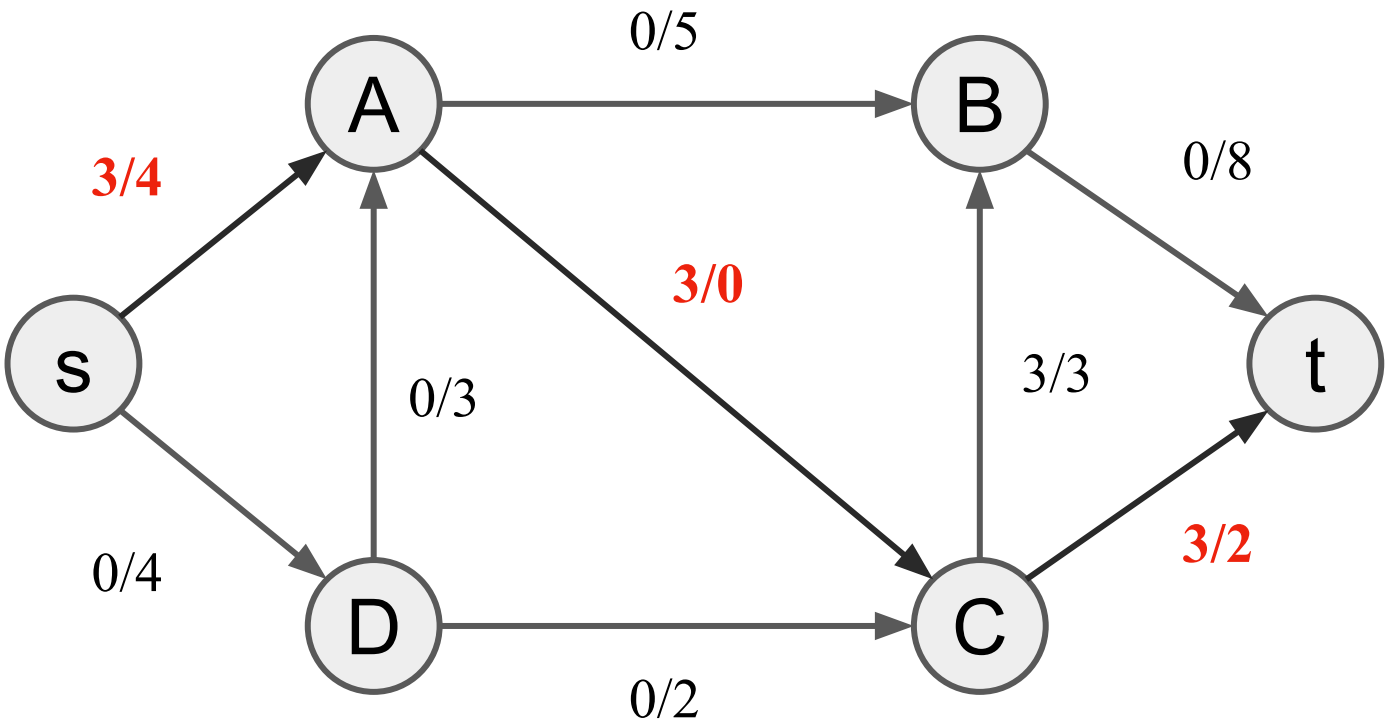


Network Flow

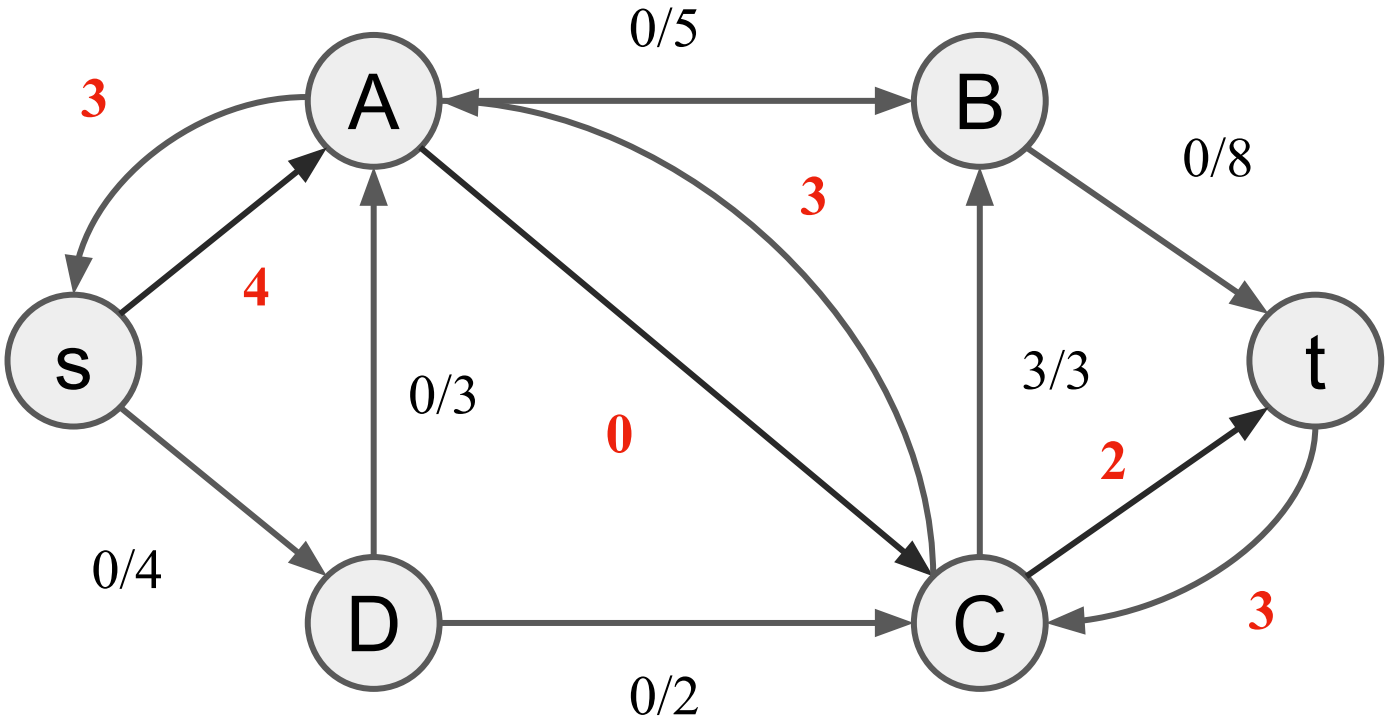


Augment flow with amount 3

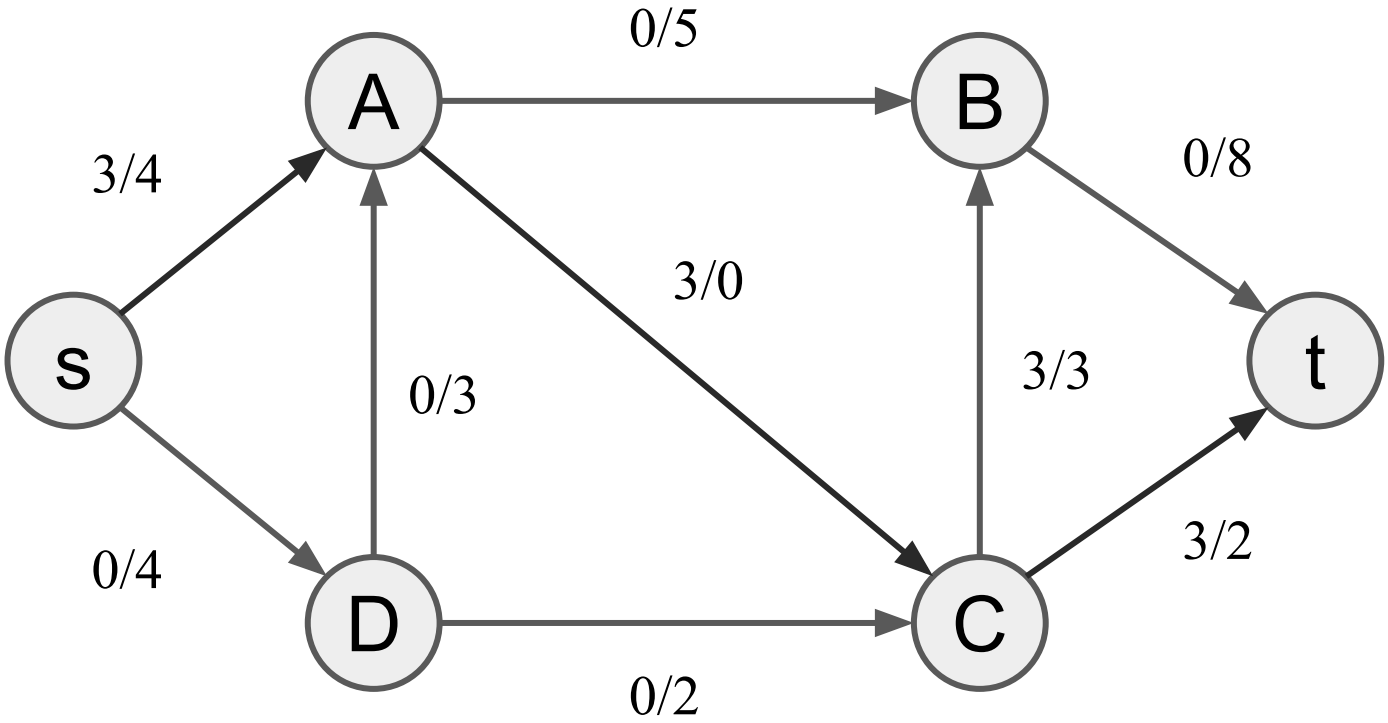
Network Flow



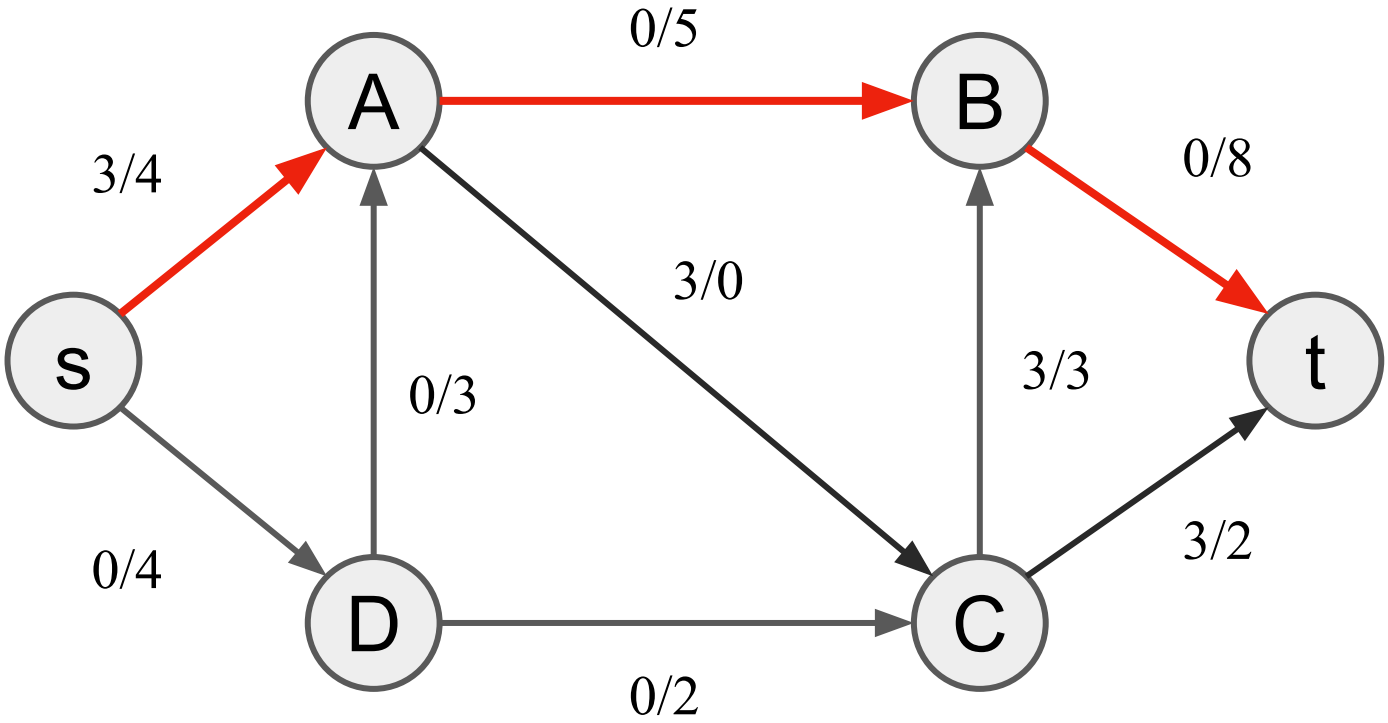
Network Flow



Network Flow

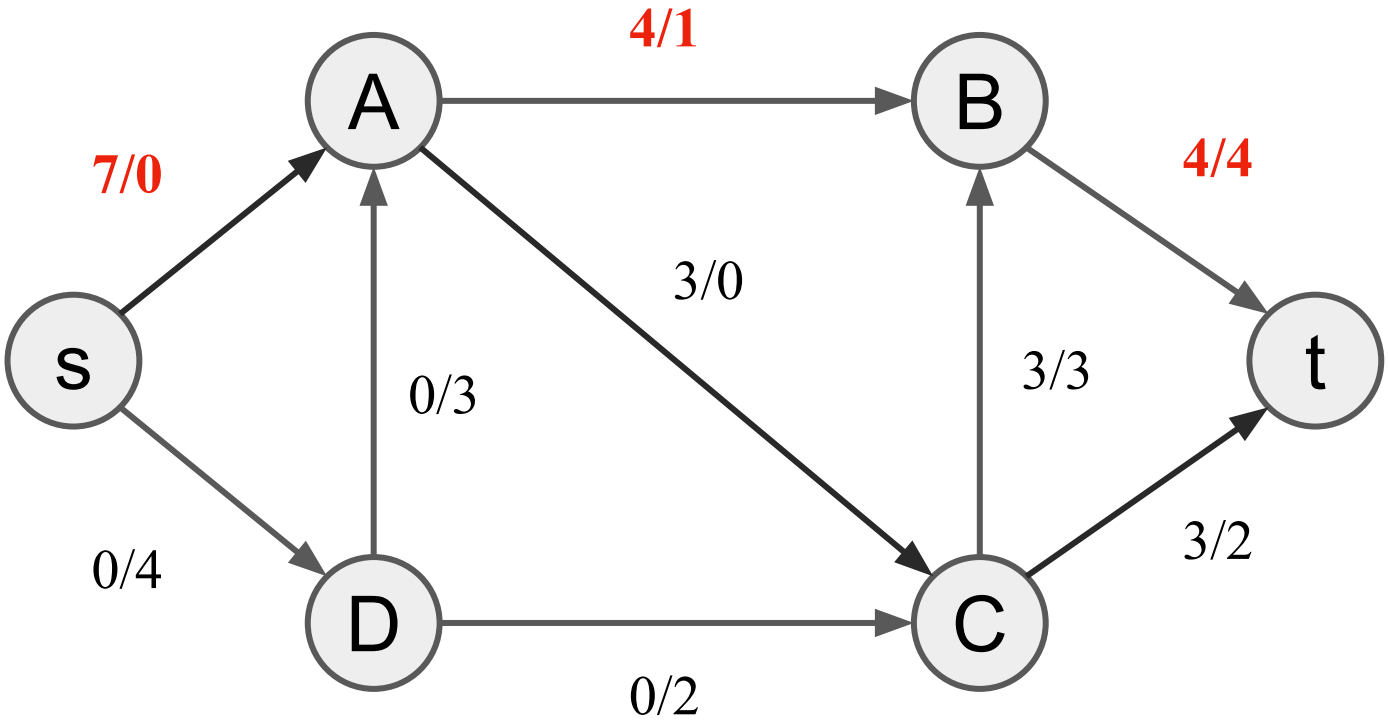


Network Flow

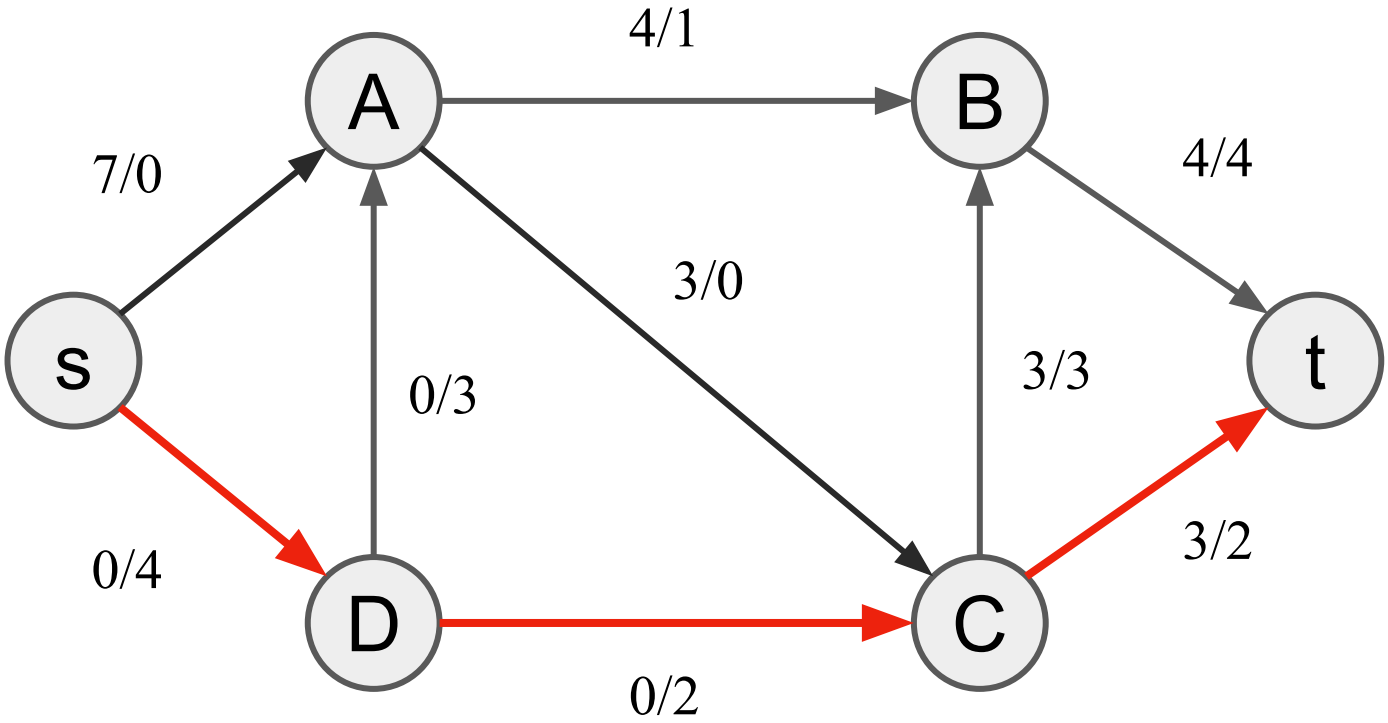


Augment flow with amount 4

Network Flow

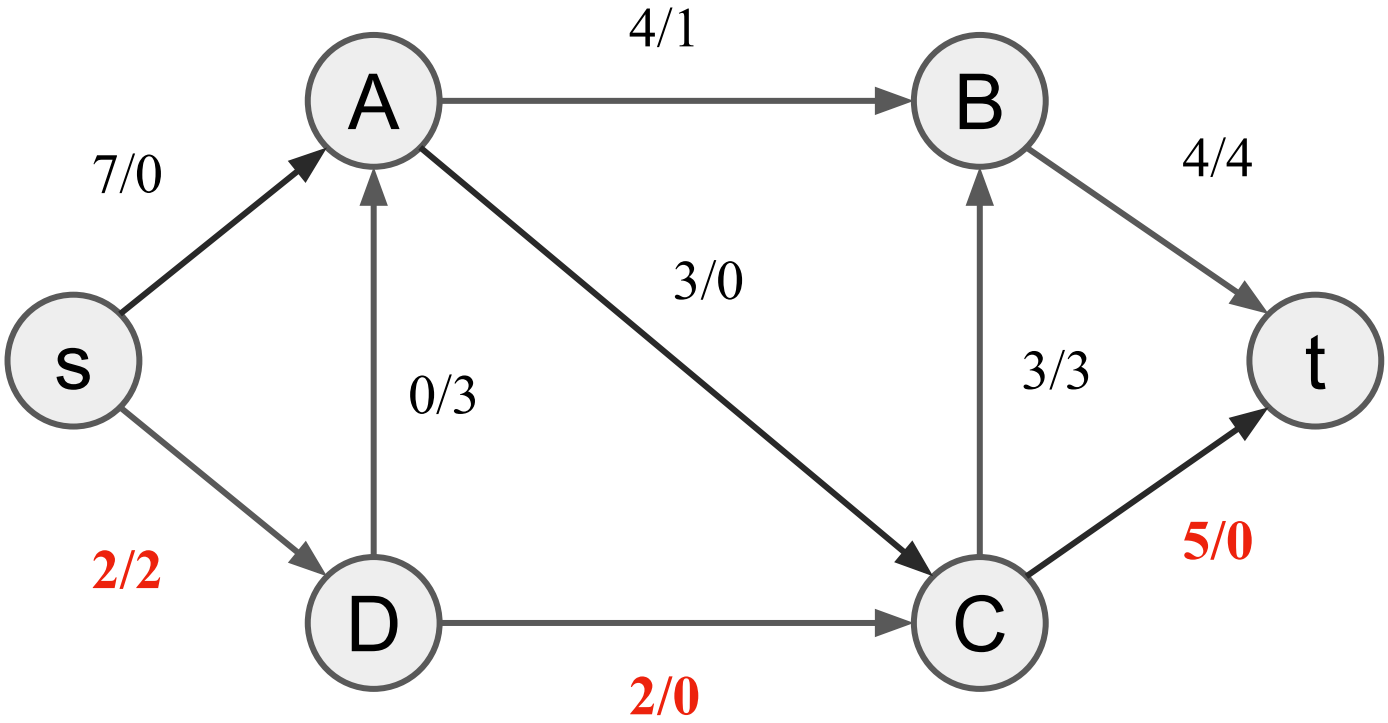


Network Flow

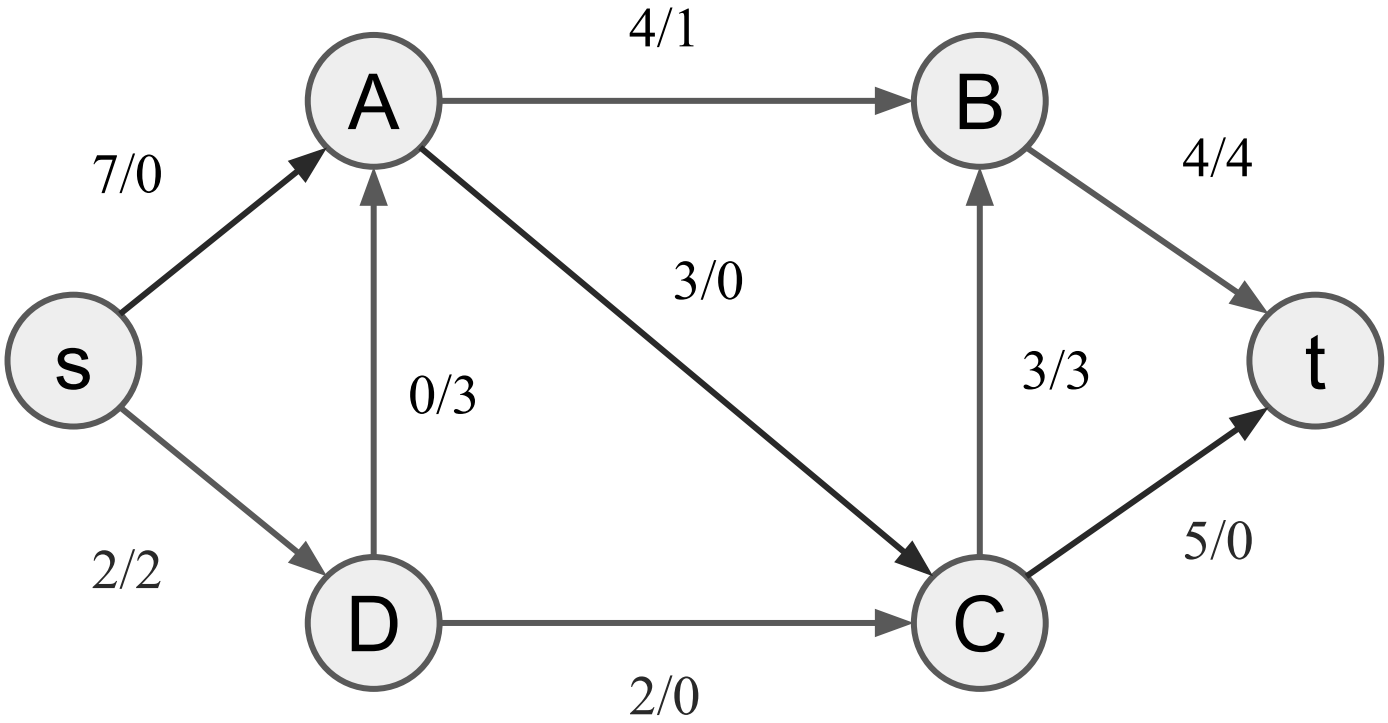


Augment flow with amount 2

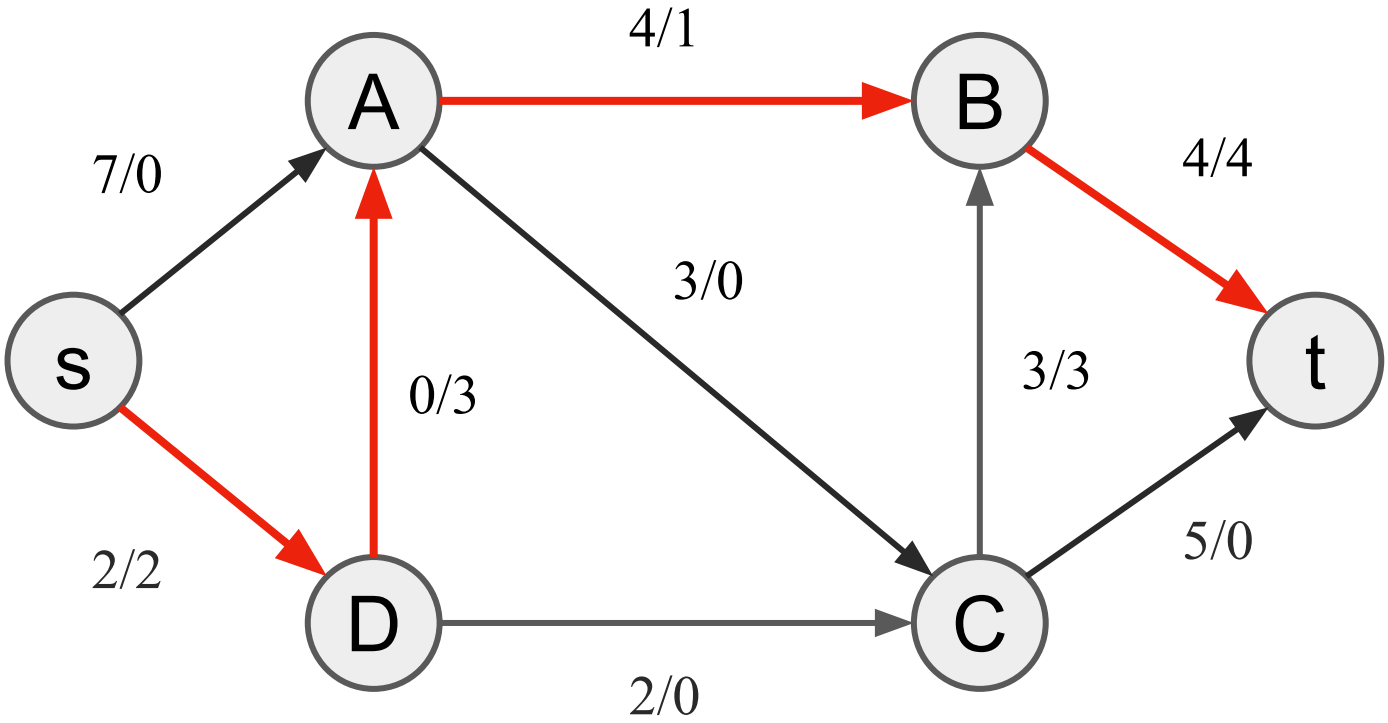
Network Flow



Network Flow

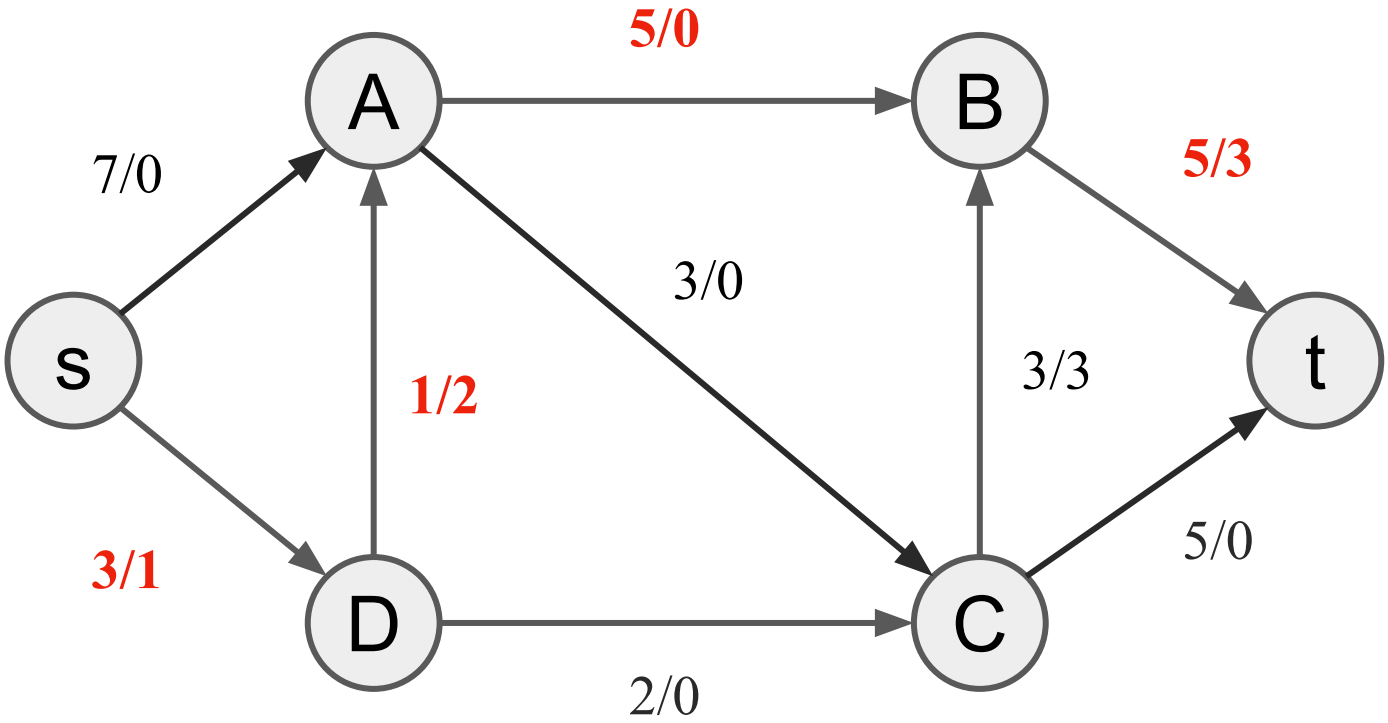


Network Flow



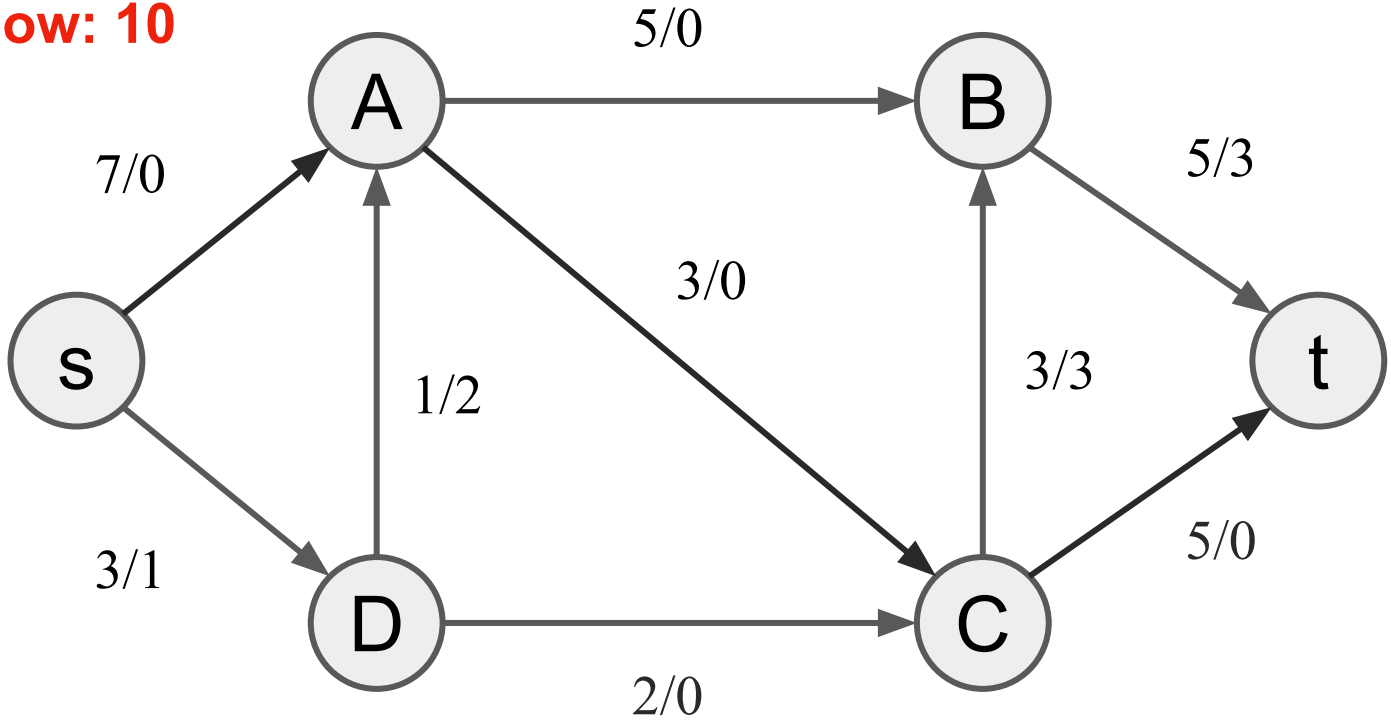
Augment flow with amount 1

Network Flow



Network Flow

Max flow: 10

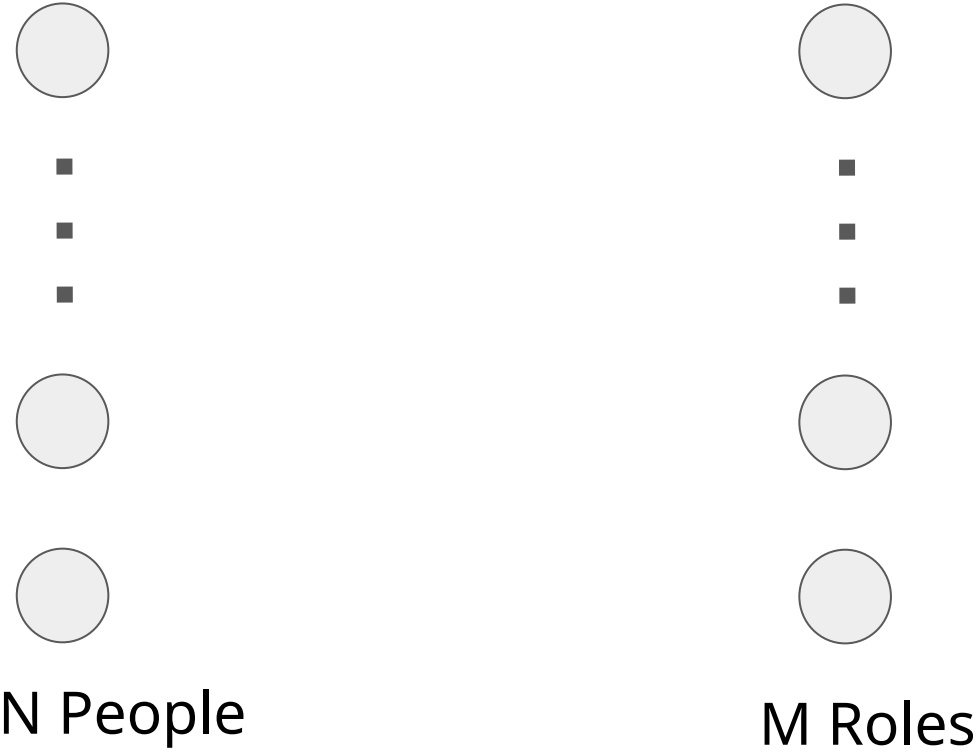


Team Building

- There are N people who want to have a role in a team
- The team has M roles and i^{th} role can have at most r_i people
- Each person is qualified for several roles
- Q: Can the team have all the N people?
- This is a typical **matching problem**
- Can be solved as a max flow problem.

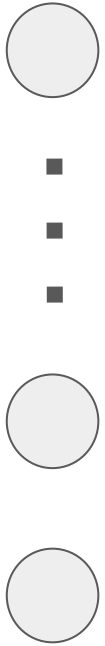
Network Flow

Network Construction

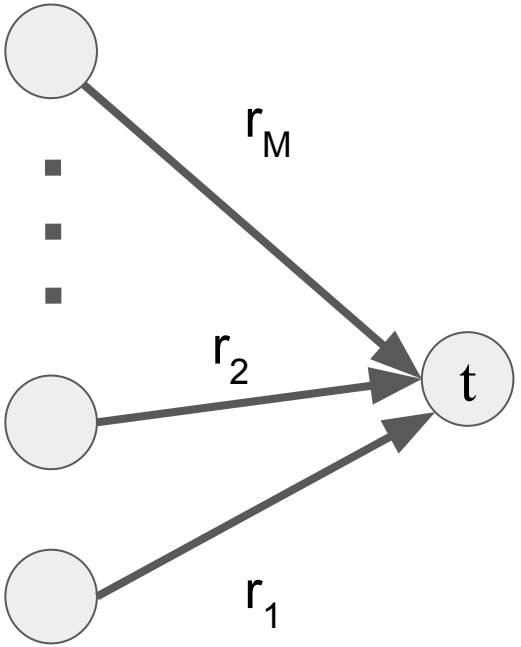


Network Flow

Network Construction



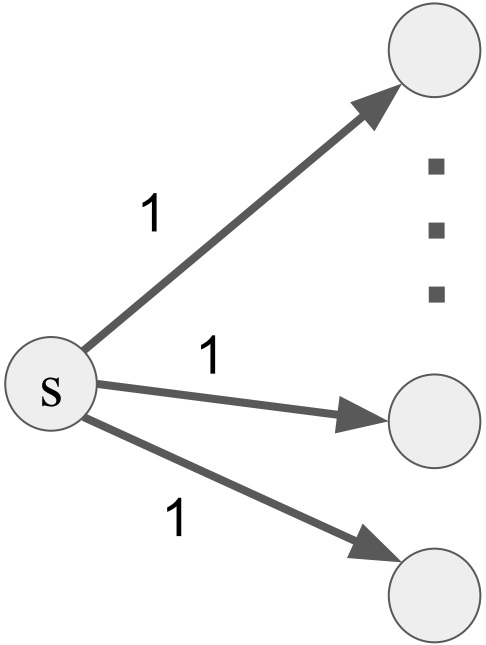
N People



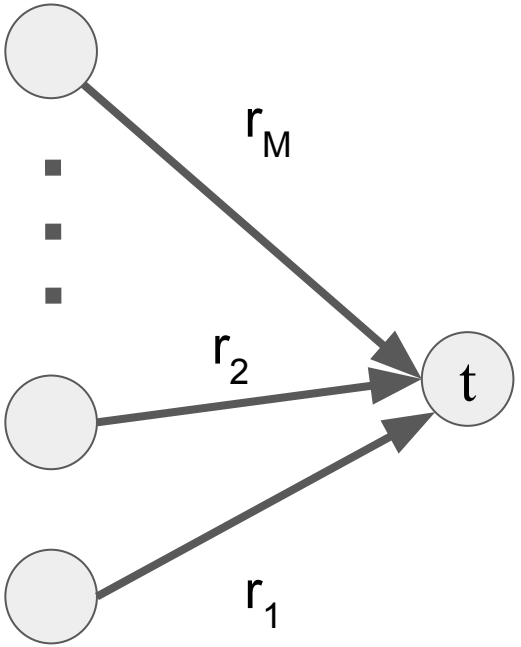
M Roles

Network Flow

Network Construction



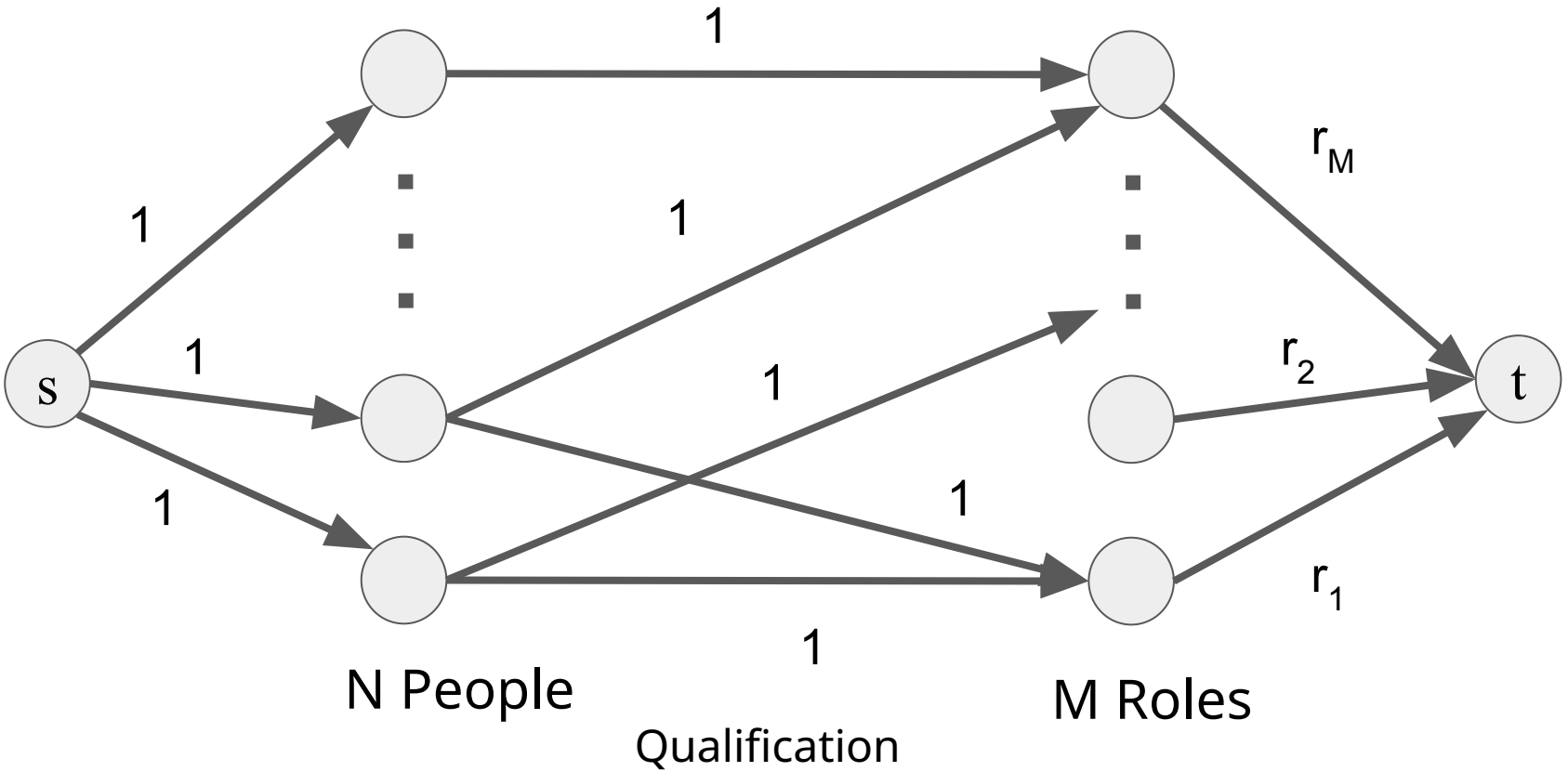
N People



M Roles

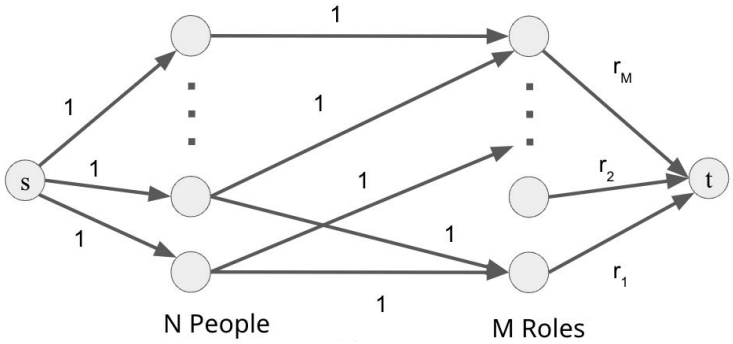
Network Flow

Network Construction



Network Flow

Given the constructed graph

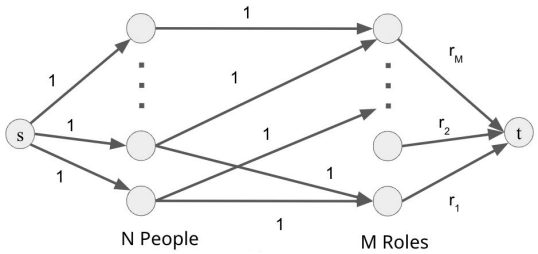


- Describe “how to use max-flow algorithm”
  - Run maxflow algorithm
  - If the maxflow is N, return true. (The team can have all the N people.)
- Why is this correct?

  Prove “The team can have all the N people if and only if the max-flow is N”
  - **Forward direction:** If the team can have all the N people, the max-flow is N.
  - **Backward direction:** If the max-flow is N, the team can have all the N people.

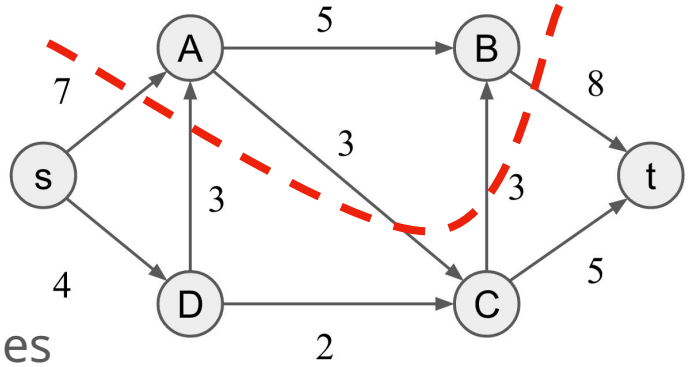
Proof in more detail

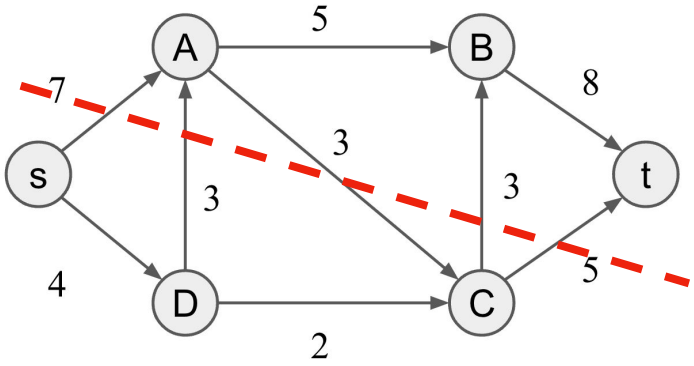
- **Forward direction:** If the team can have all the N people, the max-flow is N.
  - If the team can have all the N people,
  - Role nodes can accept all the N flow without exceeding the capacities of r_1, r_2, ..., $r_{M.}$
  - Thus the maxflow is at least N.
  - Also maxflow cannot be more than N because there are only N edges from s. Thus **maxflow = N**
- **Backward direction:** If the max-flow is N, the team can have all the N people.
  - If the maxflow is N, each person has a unit flow to a role
  - Since this max flow is a valid flow, every role has valid number of people.
  - Thus **the team can have all the N people**.



Min-Cut

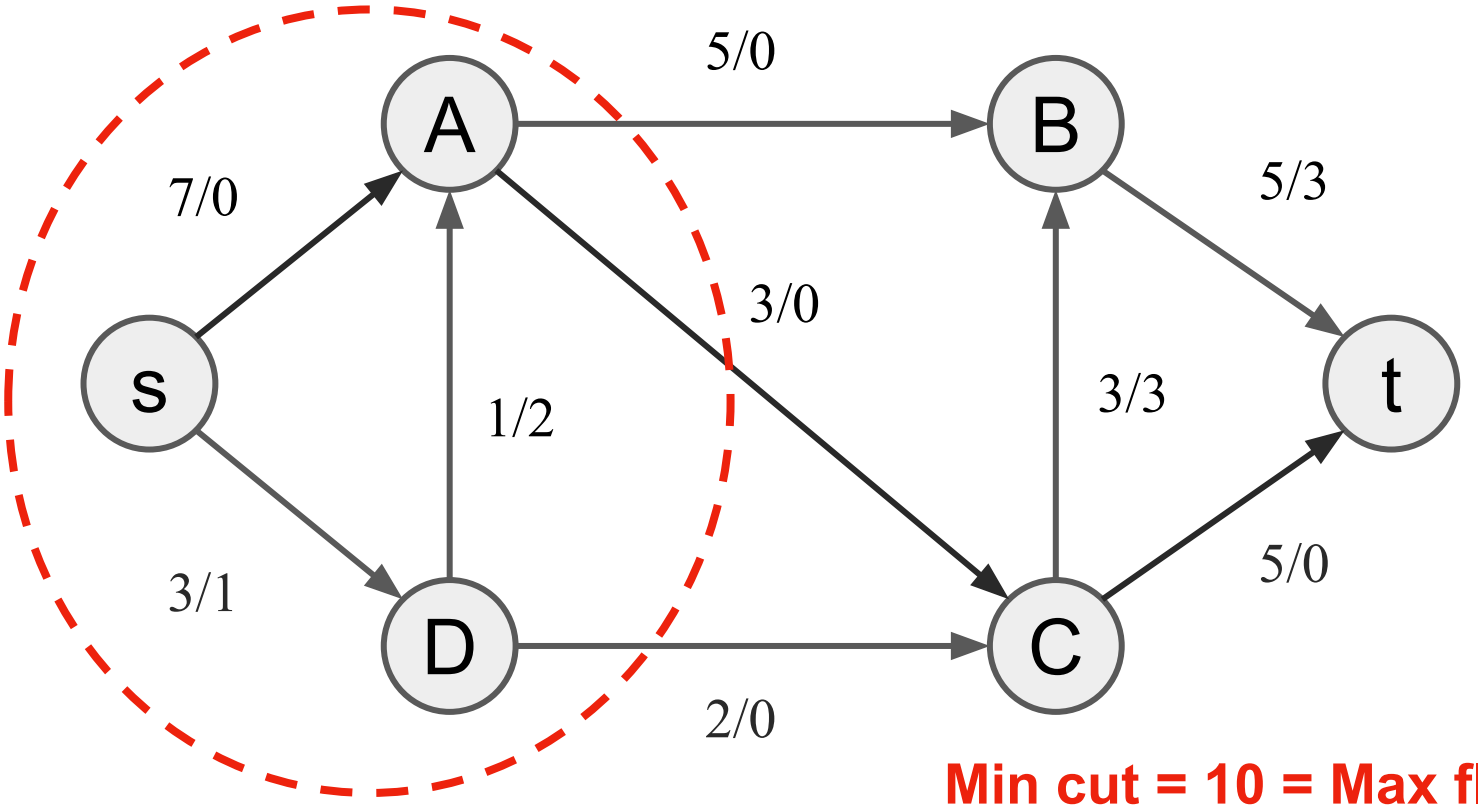
- Cut of a graph: Disjoint partition of nodes
- s-t cut: A cut that split s and t
- Value of s-t cut: total weights of outgoing edges
- What is the value of the cut on the bottom?
- Max-Flow solver can solve Min-Cut problem
- Min-Cut value = Max-Flow value





Network Flow

Nodes reachable from s



Min cut = 10 = Max flow

Divide and Conquer

Shao-Hung Chan

OH: Friday 9:00-11:00 on Zoom

Recap

Given a problem of size n

1. Divide it into a subproblems of size n/b, $b > 1$
2. Solve each subproblem recursively
3. **Merge** the solutions from the subproblems

Time analysis

1. Recursion tree
2. Master Theorem

One way of thinking

1. How to solve a problem with **brute force**?

   (What is the **extra work** in brute force?)

2. How to divide the problem to avoid extra work?

   (**How to merge the subproblems to avoid...**)

   (a) We only need to do half of the problem

   (e.g., Nearest pairs of points)

   (b) Avoid all pair comparisons

   (e.g., MergeSort)

Max Subsequence Sum

- Input: a (unsorted) sequence $a_1, a_2, ..., a_n$
- Output: $\max_{i,j} (a_i + a_{i+1} + ... + a_j)$
- Examples

19

-5, 12, -4, -3, 14, -9

-5, 12, -10, -11, 14, -9

Max Subsequence Sum

- Input: a (unsorted) sequence $a_1, a_2, ..., a_n$
- Output: $\max_{i,j} (a_i + a_{i+1} + ... + a_j)$
- Examples

19

-5, 12, -4, -3, 14, -9

-5, 12, -10, -11, 14, -9

- Algorithm 1: **Brute Force**
  - Compute the sum $a_i + a_{i+1} + ... + a_j$ **for all i,j**
  - for i=1,..., n

        for j=i+1,...,n

            ...

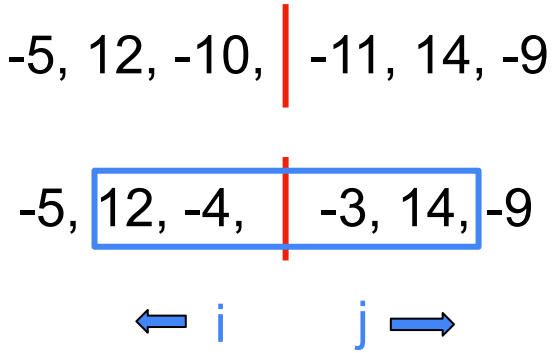
Time complexity: $O(n^2)$

What is the extra work?

Max Subsequence Sum

- Input: a (unsorted) sequence $a_1, a_2, ..., a_n$
- Output: $\max_{i,j} (a_i + a_{i+1} + ... + a_j)$
- Algorithm 2: D&C

MaxSubseqSum$(a_1, a_2, ..., a_n)$:	T(n)
if n=1, **return** a_1	Θ(1)
$m_1 = MaxSubseqSum(a_1, a_2, ..., a_{n/2})$	T(n/2)
$m_2 = MaxSubseqSum(a_{n/2+1}, ..., a_n)$	T(n/2)
$m_3 = \max_{1 \le i \le n/2}(a_i, ..., a_{n/2})$	Θ(n)
$m_4 = \max_{n/2+1 \le j \le n}(a_{n/2+1}, ..., a_j)$	Θ(n)
return $\max(m_1, m_2, m_3+m_4)$	Θ(1)



$T(n) = 2\,T(n/2) + \Theta(n)$

Which case in Master Theorem?

Integer Multiplication

e.g., $1234 \cdot 4321$

- Assumption:
  - 1 digit multiplication takes 1 time unit
  - Any addition takes O(1)
- Input: 2 integer x and y with n digits, Output: xy

Integer Multiplication

e.g., $1234 \cdot 4321$

- Assumption:
  - 1 digit multiplication takes 1 time unit
  - Any addition takes O(1)
- Input: 2 integer x and y with n digits, Output: xy
- Algorithm 1: Brute Force $\rightarrow$ Time complexity?

Integer Multiplication

e.g., $1234 \cdot 4321$

- Assumption:
  - 1 digit multiplication takes 1 time unit
  - Any addition takes O(1)
- Input: 2 integer x and y with n digits, Output: xy
- Algorithm 1: Brute Force → Time complexity?
- Algorithm 2: Divide and Conquer
  - $x = a \cdot 10^{n/2} + b,\ y = c \cdot 10^{n/2} + d$
  - $xy = ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$
  - Time complexity?

Integer Multiplication

e.g., $1234 \cdot 4321$

- Assumption:
  - 1 digit multiplication takes 1 time unit
  - Any addition takes O(1)
- Input: 2 integer x and y with n digits, Output: xy
- Algorithm 1: Brute Force → Time complexity?
- Algorithm 2: Divide and Conquer
  - $x = a \cdot 10^{n/2} + b,\ y = c \cdot 10^{n/2} + d$
  - $xy = ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$
  - Time complexity?
    - $T(n) = 4 \cdot T(n/2) + O(1),\ T(1)=1$
    - Which case in Master Theorem?

Integer Multiplication

e.g., 1234 · 4321

- Algorithm 2: Divide and Conquer
  - $x = a \cdot 10^{n/2} + b,\ y = c \cdot 10^{n/2} + d$
  - $xy = ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd$
  - Time complexity? $T(n) = 4 \cdot T(n/2) + O(1) \rightarrow T(n) = \Theta(n^2)$
- Algorithm 3: Divide and Conquer (Karatsuba, 1962)
  - $xy = ac \cdot 10^{n} + \mathbf{[(a+b)\,(c+d) - ac - bd]} \cdot 10^{n/2} + bd$
  - Time complexity?

    $T(n) = \mathbf{3} \cdot T(n/2) + O(1)$

    $T(n) = \Theta(n^{\log_2 3})$

Master Theorem

- $T(n) = a\,T(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a} \cdot n^{-\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$

  $f(n) \cdot n^{\varepsilon} < n^{\log_b a} \rightarrow$ leaf dominant $\quad \varepsilon > 0$
- Case 2: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$

  $f(n) = n^{\log_b a} \cdot \log^k n \rightarrow$ branch dominant $\quad k \geq 0$
- Case 3: $f(n) = \Omega(n^{\log_b a} \cdot n^{\varepsilon}) \Rightarrow T(n) = \Theta(f(n))$

  $f(n) > n^{\log_b a} \cdot n^{\varepsilon} \rightarrow$ root dominant $\quad \varepsilon > 0$

  $a \cdot f(n/b) \leq c \cdot f(n)$, for some $c < 1$

  (descending geometric series...)

Master Theorem

- Problem
- $T(n) = 2 \cdot T(n/2) + n^4$
- $T(n) = 2 \cdot T(n/4) + n^{1/2}$
- $T(n) = 7 \cdot T(n/2) + n^2$

- $T(n) = a\ T(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a} \cdot n^{-\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$
- Case 3: $f(n) = \Omega(n^{\log_b a} \cdot n^{\varepsilon}) \Rightarrow T(n) = \Theta(f(n))$

Master Theorem

- Problem
- $T(n) = 2 \cdot T(n/2) + n^4$
  - Case 3, $T(n) = \Theta(n^4)$
- $T(n) = 2 \cdot T(n/4) + n^{1/2}$
  - Case 2, $T(n) = \Theta(n^{1/2} \log n)$
- $T(n) = 7 \cdot T(n/2) + n^2$
  - Case 1, $T(n) = \Theta(n^{\log_2 7})$

- $T(n) = a\,T(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a} \cdot n^{-\varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$
- Case 3: $f(n) = \Omega(n^{\log_b a} \cdot n^{\varepsilon}) \Rightarrow T(n) = \Theta(f(n))$

Beyond the Exam

Problem: $T(n) = 2 \cdot T(n^{1/2}) + \log_2 n$, $T(2)=0$

1. Try to fit into Master Theorem: $T(\mathbf{m}) = 2 \cdot T(\mathbf{½\ m}) + f(m)$, $T(\mathbf{1})=0$

   How to move ½ from power to product?

2. Change the variable

   Let $m=\log_2 n \rightarrow n=2^m \rightarrow \mathbf{T(2^m)} = 2 \cdot T(2^{m/2}) + m$, $T(2)=0$

   Define a new function $g(m) = 2^m \rightarrow g(m/2)=2^{m/2}$

   $T(\mathbf{g(m)}) = 2 \cdot T(\mathbf{g(m/2)}) + m$, $T(2)=0$

   Define a new function $S(m) = T(g(m))$

   $S(m) = 2 \cdot S(m/2) + m$, $S(1)=T(g(1))=T(2)=0$

$S(m)=m \log_2 m$
$T(g(m)) = m \log_2 m$
$T(2^m) = m \log_2 m$
$T(n) = \log_2 n \cdot \log_2(\log_2 n)$