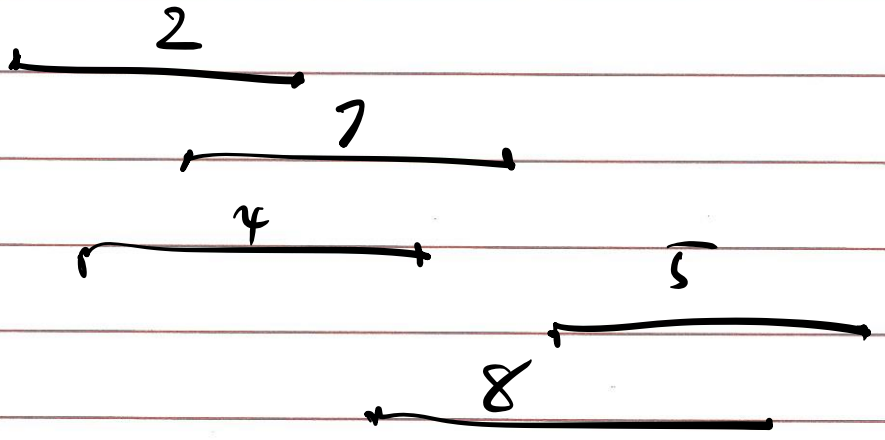
General Approach to Solving optimization problems using Dynamic Programming

1. Characterize the structure of an opt. solution
2. Recursively define the value of an opt. solution
3. Compute the value of an opt. solution in a bottom up fashion
4. Construct an opt. sol. from computed information

Problem Statement

- We have 1 resource
- " " n requests labeled 1 to n
- Each request has start time s_i, finish time f_i, and weight w_i

Goal: Select a subset $S \subseteq \{1..n\}$ of mutually compatible intervals so as to Maximize $\sum_{i \in S} w_i$



Observation: Either job i is part of the opt sol. or it isn't

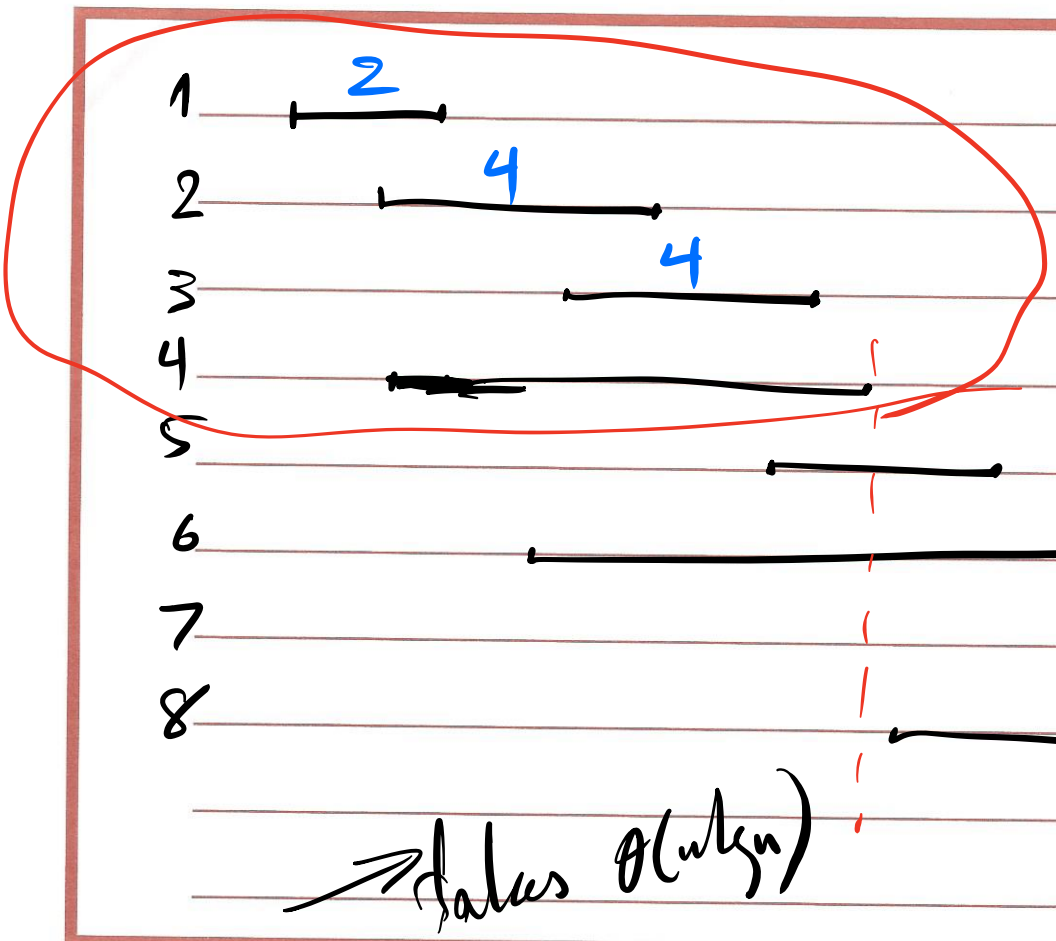
Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

Case 2 - if it isn't, value of the opt. sol. = value of the opt. sol. without job i

Sort requests in order of non-decreasing finish time.

$$f_1 \leq f_2 \leq \dots \leq f_n$$

Define $P(j)$ for an interval j to be the largest index $i < j$ such that interval i & j are disjoint.



P( )

P(1) = 0

P(2) = 0

P(3) = 1

P(4) = 0

P(5) = 2

P(6) = 1

P(7) = 5

P(8) = 4

→ Solves $\theta(n \log n)$

Def Let O_j denote the opt. solutions to the problem consisting of requests $\{1..j\}$

Let $OPT(j)$ denote the value of O_j

$O_3 = \{1, 3\}$ $OPT(3) = 6$

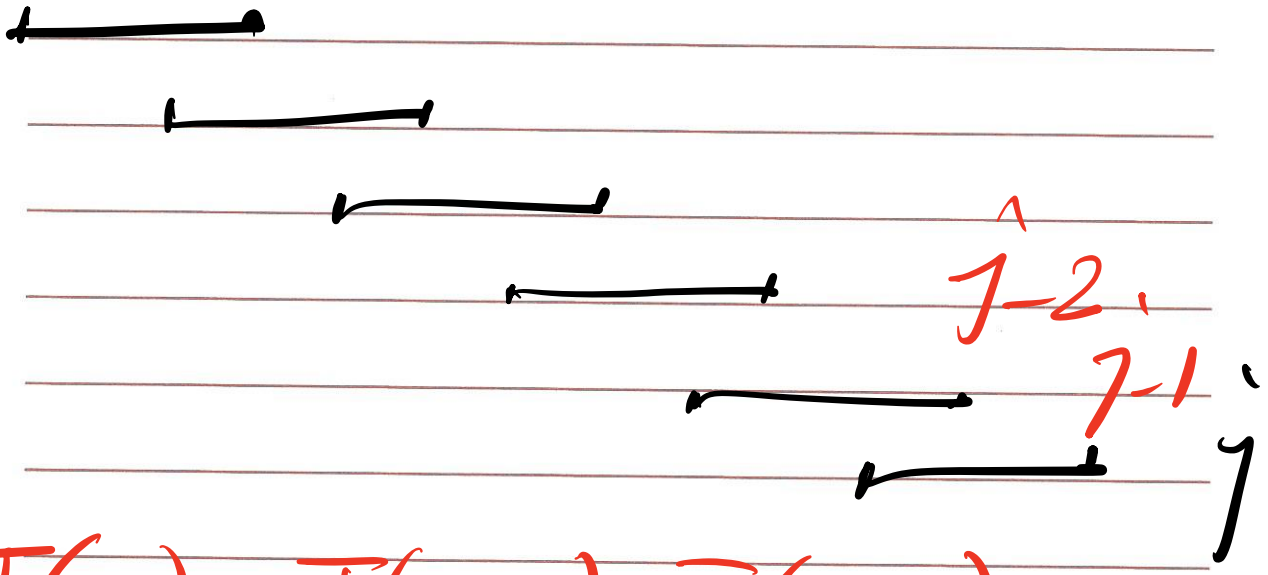
Recurrence formula

Case 1: $j \in O_j \Rightarrow OPT(j) = w_j + OPT(P(j))$

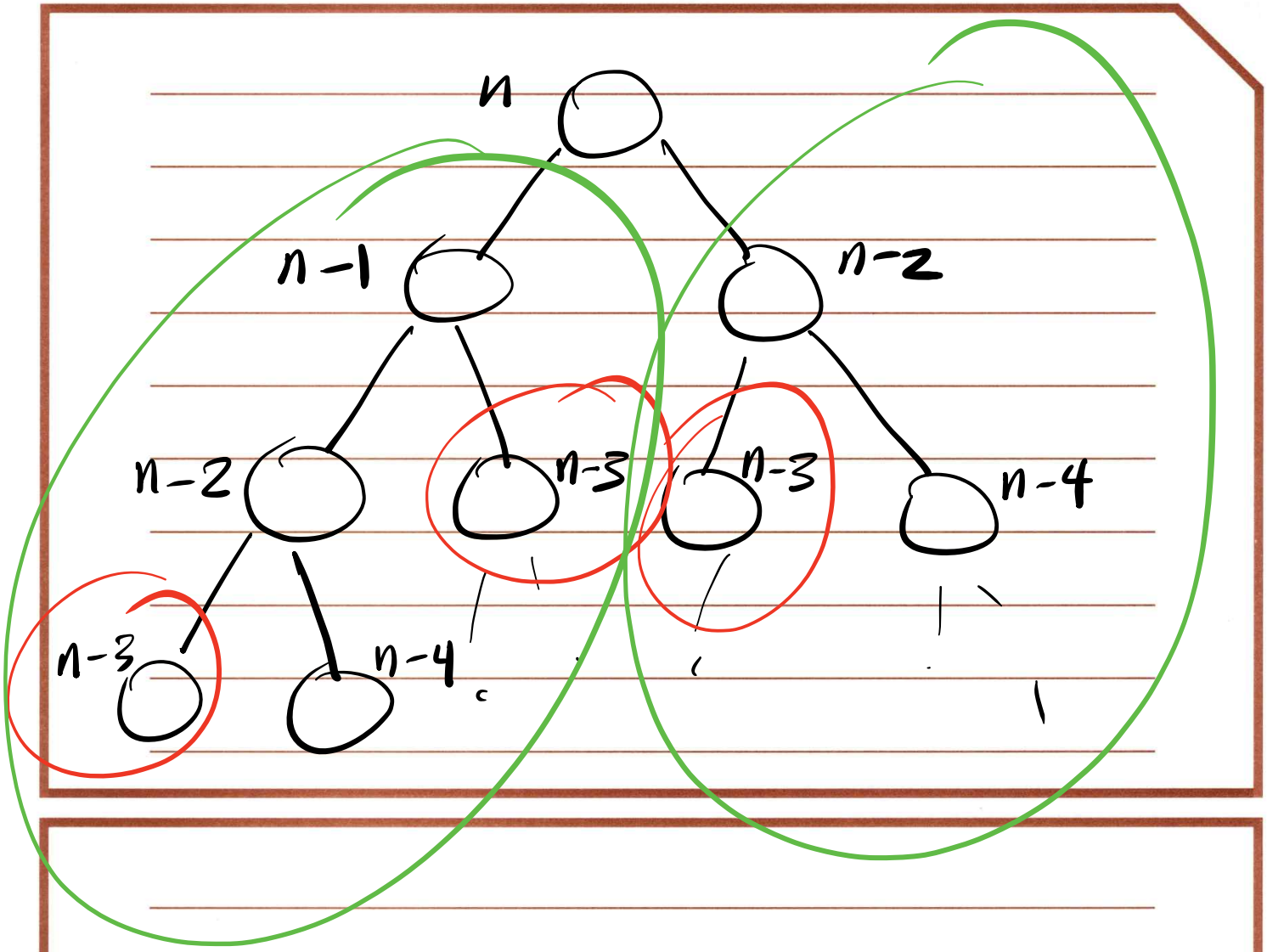
Case 2: $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

Solution:

```
Compute-opt(j)
    if j = 0 then
        return 0
    else
        return Max(
            wj + Compute-opt(P(j)),
            Compute-opt(j-1))
    endif
```



$$T(n) = T(n-1) + T(n-2)$$


n
n-1
n-2
n-2
n-3
n-3
n-4
n-3
n-4

Memoization

Store the value of Compute-opt. in a globally accessible place the first time we compute it. Then simply use this precomputed value in place of all future recursive calls.

```
M-Compute-opt(j)
    if j=0 then
        return 0
    else if M[j] is not empty then
        return M[j]
    else define M[j] = Max(wj + M-Compute-opt(p(j)), M-Compute-opt(j-1))
        return M[j]
    endif
```

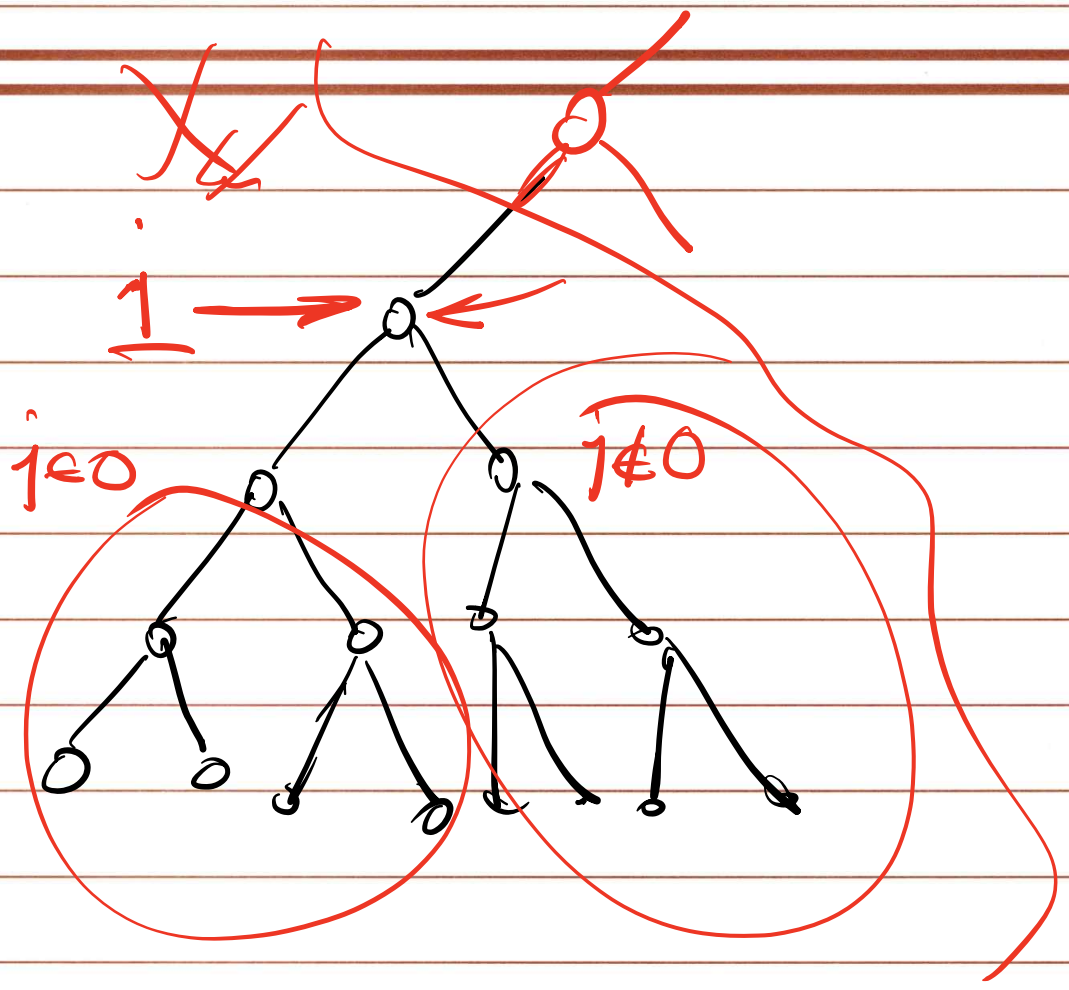
O(n)

initial Sorting : $O(n \log n)$

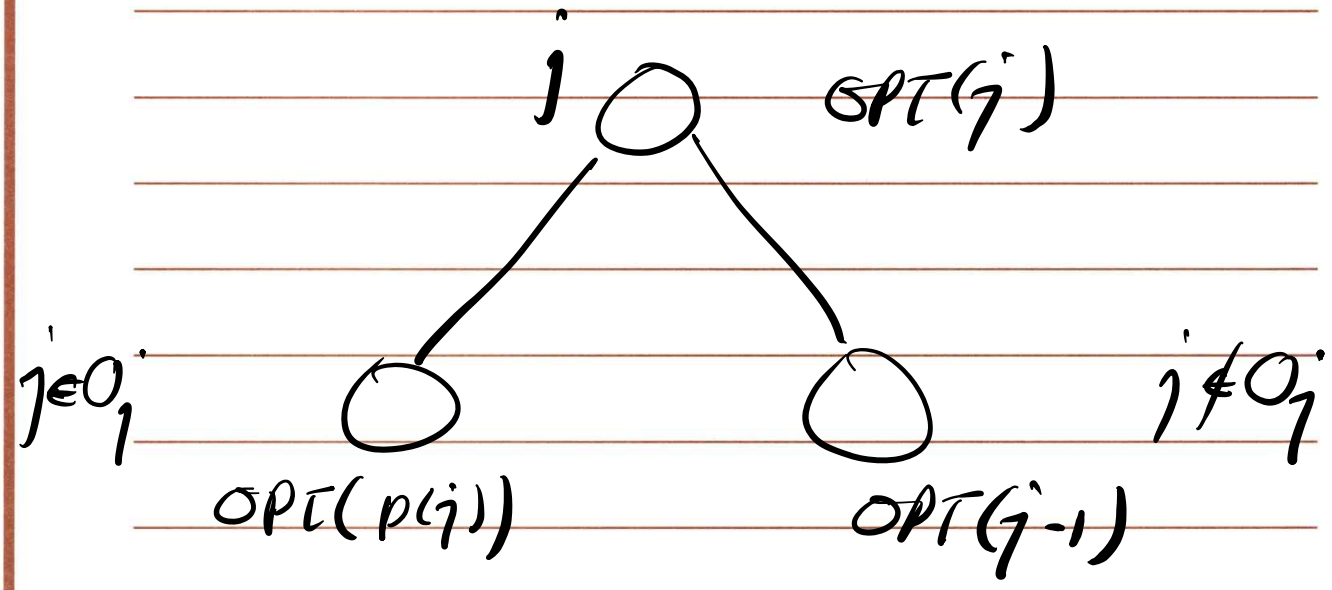
Build P() : $O(n \log n)$

M_Compute_opt : $O(n)$

overall Complexity = $O(n \log n)$



Compute an opt. Sol.



j belongs to O_j iff

$$w_j + OPT(p(j)) \geq OPT(j-1)$$

Find_Solution

if $j > 0$ then

if $w_j + M[p(j)] \geq M[j-1]$ then

output j together w/ the results of Find_Solution ($p(j)$)

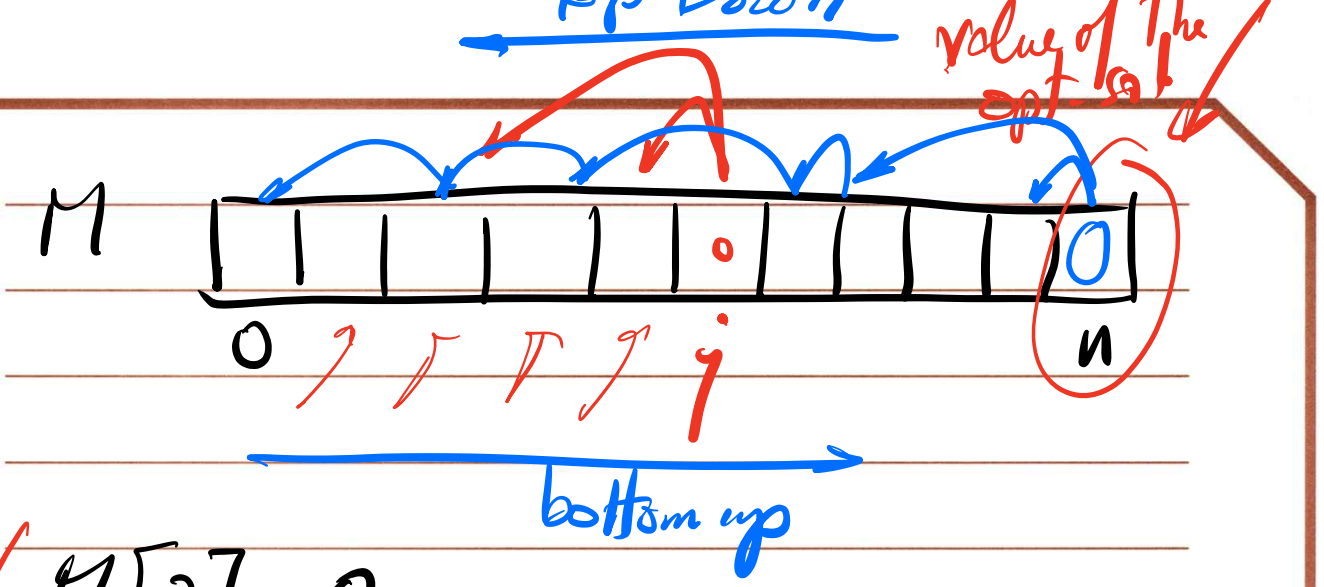
else

output the results of Find_solution ($j-1$)

end if

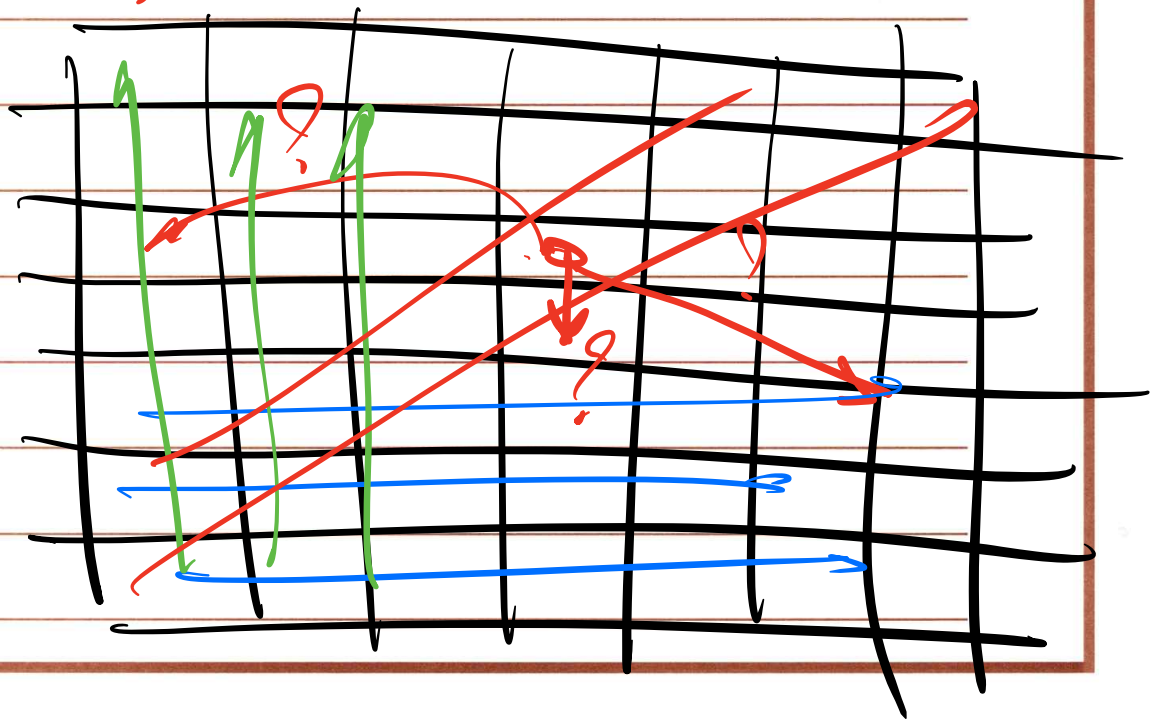
end if

$O(n)$

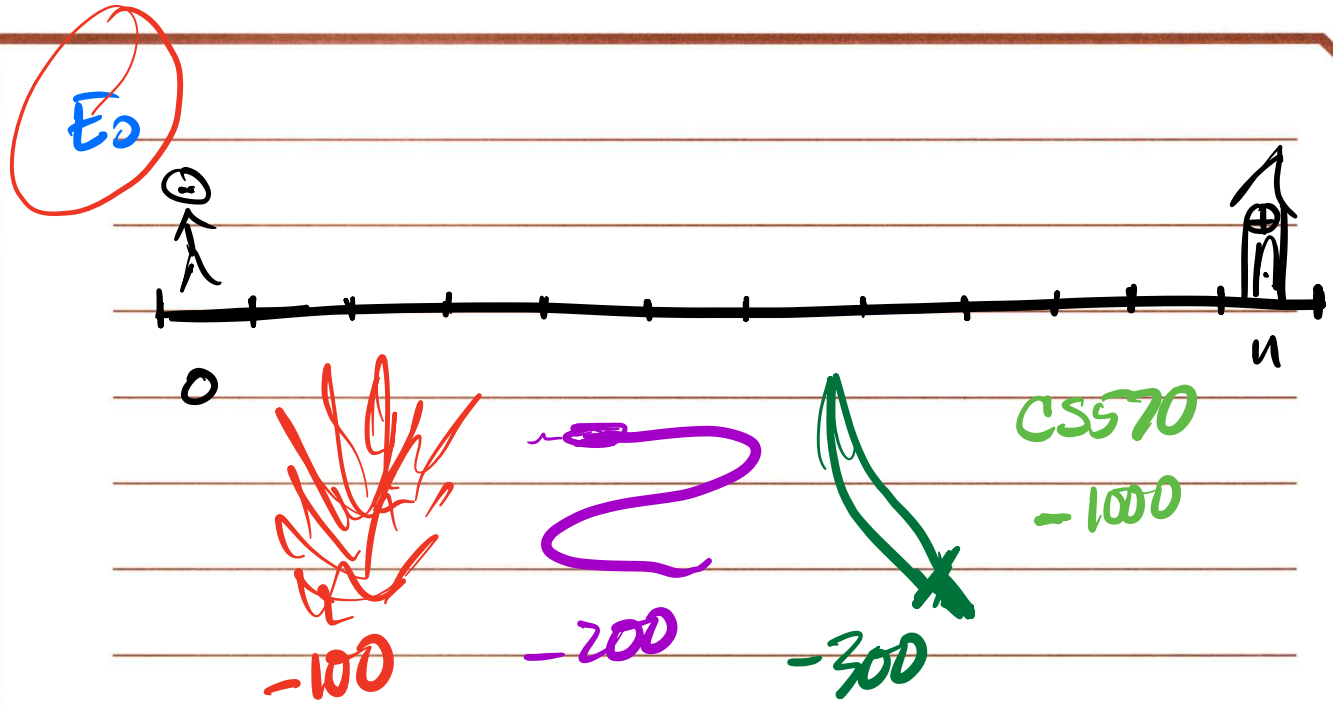



```
M[0] = 0
for i = 1 to n
    M[i] = Max (M[i-1],
                w_i + M[p(i)])
endfor
```

$\Theta(n)$



Videogame Problem



in general, we lose e_i units of energy when landing on stage i

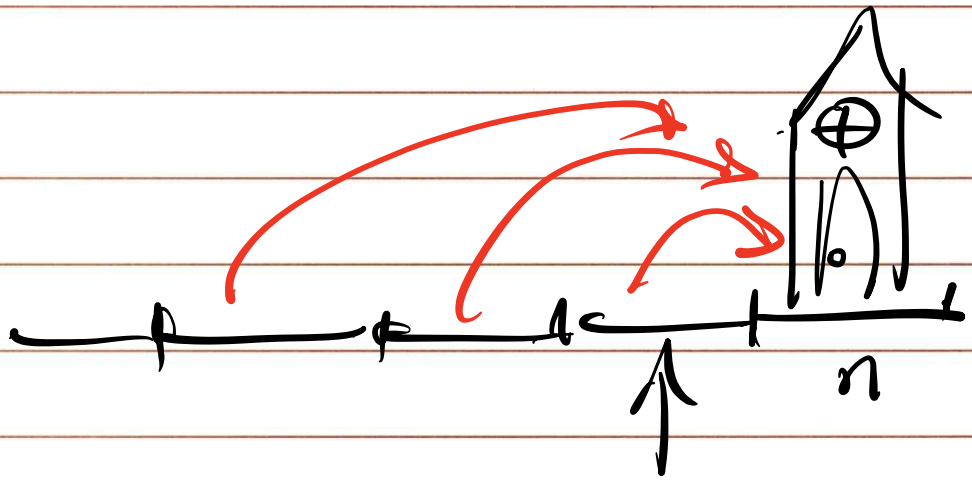
Choice: 1- walk into next stage costs 50 units

2- jump over one stage " 150

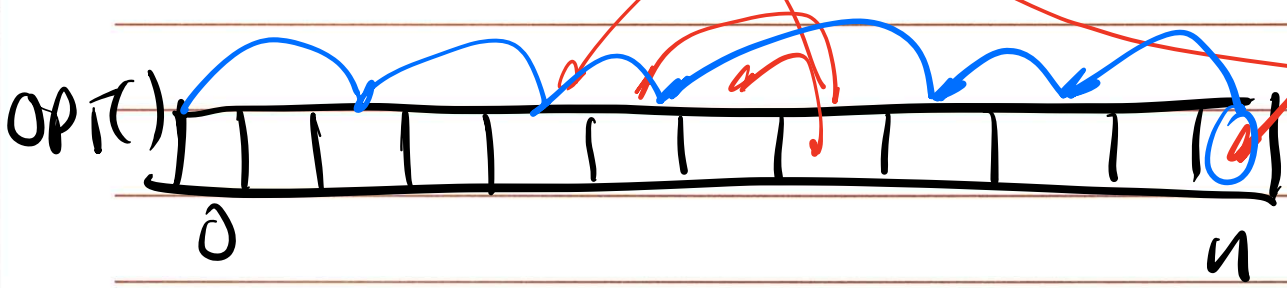
3- " " two stages 350

Question: How do you go home such that you lose as little energy as possible?

OPT(i) = opt. level of energy when we reach stage 'i'.



$$OPT(n) = Max(OPT(n-1) - 50 - e_n,$$
$$OPT(n-2) - 150 - e_n,$$
$$OPT(n-3) - 350 - e_n)$$



Value of opt. sol.

Top down

$OPT(2) = Max(\ldots)$

$OPT(0) = E_0$, $OPT(1) = E_0 - 50 \cdot e_1$

for $i = 3$ to n

end for

$\rightarrow$ takes $\Theta(n)$

Coin Problems

Austrian Schillings denominations.

1
5
10
20
25

Before Euro currency

How to pay for n Schillings w/ the min. # of coins?

$OPT(i)$ = Min # of coins to pay i Schillings w/.

$$OPT(i) = \text{Min}\,(OPT(i-1)+1,\ OPT(i-5)+1,\ OPT(i-10)+1,\ OPT(i-20)+1,\ OPT(i-25)+1)$$

$OPT(0) = 0$, $OPT(1) = \cdots$, $OPT(24) = \cdots$

for $i = 25$ to n

end for

↖ takes $\Theta(n)$

0-1 knapsack &

Subset Sum

Problem Statement

- A single resource
- Requests $\{1..n\}$ each take time w_i to process
- Can schedule jobs at any time between 0 to W

Objective: To schedule jobs such that we maximize the machine's utilization

---

$OPT(i)$ = value of the opt. sol. for requests $1-i$.

$$\begin{cases} \text{if } n \notin O, \text{ then } OPT(n) = OPT(n-1) \\ \text{if } n \in O, \text{ then } OPT(n) = w_n + OPT(n-1) \end{cases}$$

?

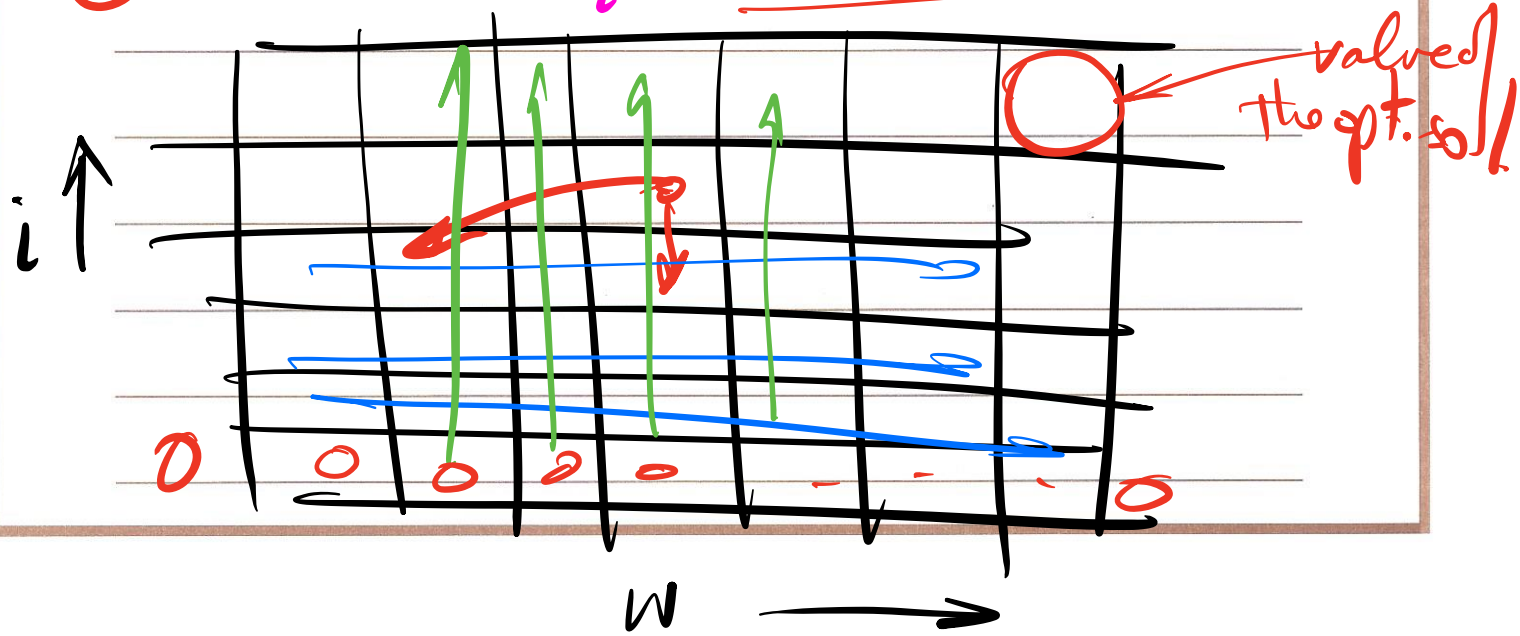
$OPT(i,w)$ = value of the opt. solution using a subset of the items $\{1..i\}$ with Max. allowed weight w.

if $n \notin O$, Then $OPT(n,w) = OPT(n-1, w)$

if $n \in O$, Then $OPT(n,w) = w_n + OPT(n-1, w-w_n)$

If $w < w_i$, Then $OPT(i,w) = OPT(i-1, w)$

else, $OPT(i,w) = Max(OPT(i-1, w), v_i + OPT(i-1, w-w_i))$



```
Subset-sum (n, W)
  array M[0,w] = 0  for each w = 0 to W
  for i = 1 to n
    for w = 0 to W
      use recurrence formula (1)
        to compute M[i, w]
    end for
  end for
  Return M[n, W]
```

n, W

Complexity = $O(nW)$

input:

w_1 | w_2 | -1 | -1 | 1 | 1 | ... | run |

n

W

01010111011

$\log_2 W$ bits

pseudo polynomial run time.

$n_W = n_Z$

$\log_2 W$

Pseudo-polynomial time

An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input

Polynomial time

An algorithm runs in polynomial time if its running time is a polynomial in the length of the input (or output).

Discussion 6

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order:$1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

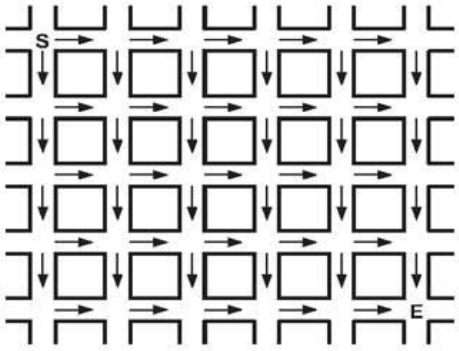


Figure A.

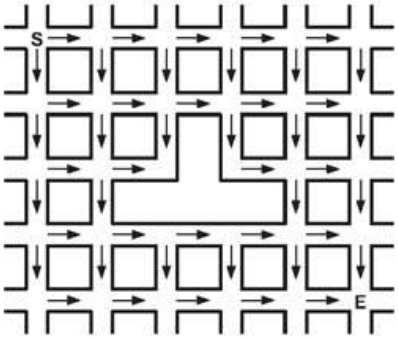
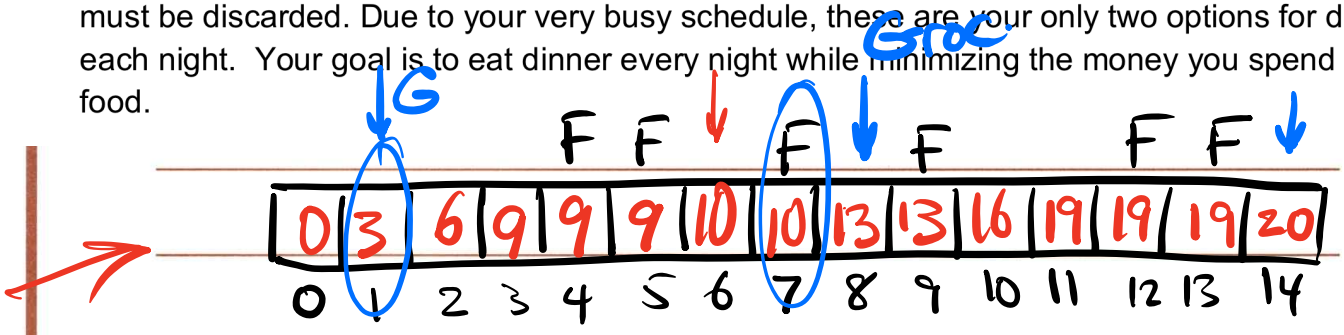


Figure B.

b) Repeat this process with Figure B; be wary of dead ends.

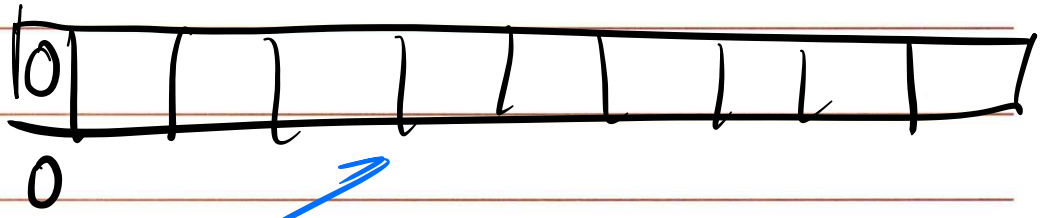
programming problem.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.



$OPT(i)$ = Min Cost to get dinner for days $1..i$

$$OPT(i) = \begin{cases} OPT(i-1) \text{ if we have free food on } i \\ \text{otherwise, } Min(OPT(i-1)+3, \\ \qquad OPT(i-7)+10) \end{cases}$$



Filling this out takes $O(n)$

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

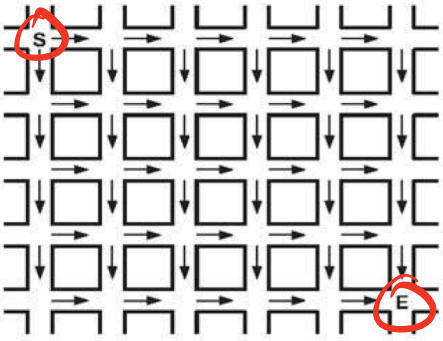


Figure A.

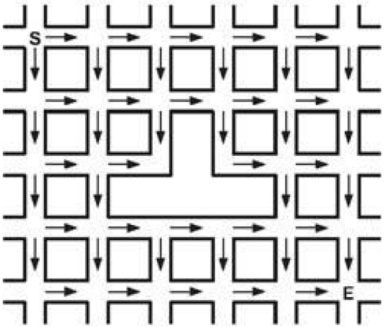


Figure B.

b) Repeat this process with Figure B; be wary of dead ends.

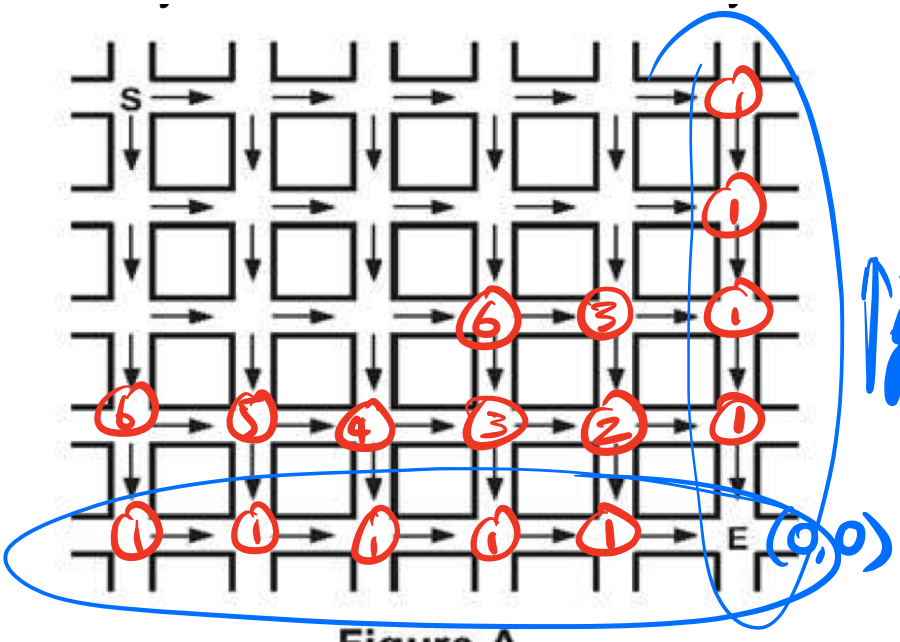


Figure A.

$OPT(i,j)$ = # of ways to go from (i,j) to E.

$OPT(i,j) = OPT(i-1,j) + OPT(i,j-1)$

This takes $O(nm)$

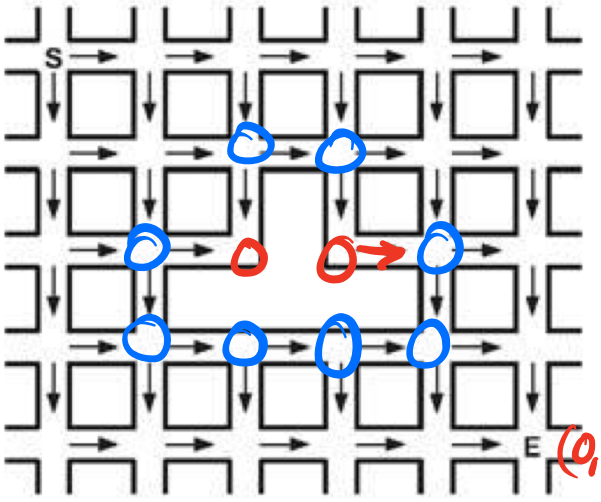


Figure B.

i

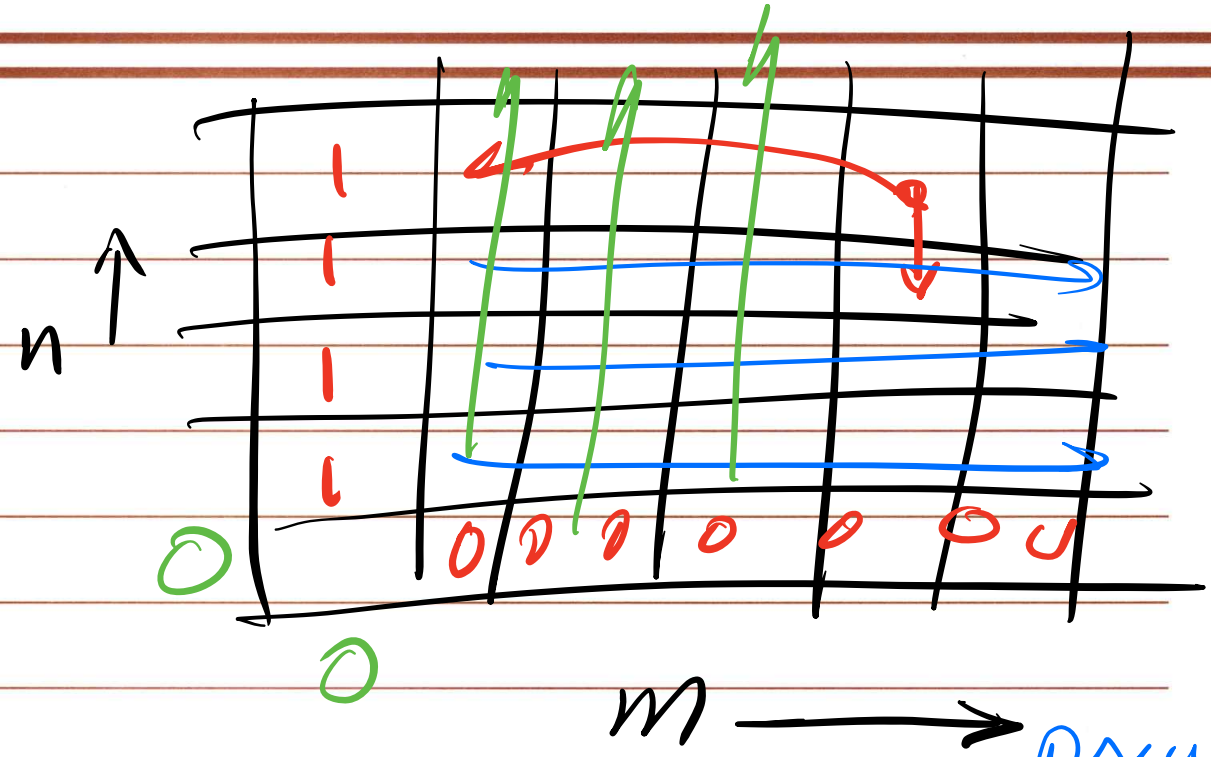
$(2,2) \rightarrow OPT(i,j) = OPT(i-1,j)$

$(3,2) \rightarrow OPT(i,j) = 0$

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order: $1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

Count(n, m) = no. of ways to pay for amount m using coins 1..n.

$$Count(n, m) = Count(n-1, m) + Count(n, m - d_n)$$



takes $O(nm)$ ← pseudo polynomial!

Assume you want to ski down the mountain. You want the total length of your run to be as long as possible, but you can only go down, i.e. you can only ski from a higher position to a lower position. The height of the mountain is represented by an *n* X *n* matrix A. A[i][j] is the height of the mountain at position (i,j). At position (i,j), you can potentially ski to four adjacent positions (i-1,j) (i,j-1), (i,j+1), and (i+1,j) (only if the adjacent position is lower than current position). Movements in any of the four directions will add 1 unit to the length of your run. Provide a dynamic programming solution to find the longest possible downhill ski path starting at any location within the given *n* by *n* grid.

1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

$OPT(i,j)$ = length of the longest downhill path starting from (i,j)

$$OPT(i,j) = \max(OPT(i',j')+1, \text{ where } (i',j') \text{ is a neighbor } \& A(i',j') < A(i,j))$$

(Sort points in increasing order of elevation

initialize all local minima to 0

loop over all subproblems in the order given by the sort.

Sort $O(n^2 \lg n^2) = O(n^2 \lg n)$

iteration $\longrightarrow O(n^2)$

overall complexity $= O(n^2 \lg n)$

Can be improved using topological ordering instead of sorting

topological ordering can be done in linear time w/ respect to the size of the graph which has $O(n^2)$ nodes & edges, so topological sort will take $O(n^2)$

Imagine starting with the given decimal number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence

32492 → **3249** → **324** → 24 → **4**

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

32492 → **3249** → 249 → **49** → **9**

also gives you 3 perfect squares, viz. 3249, 49, and 9.

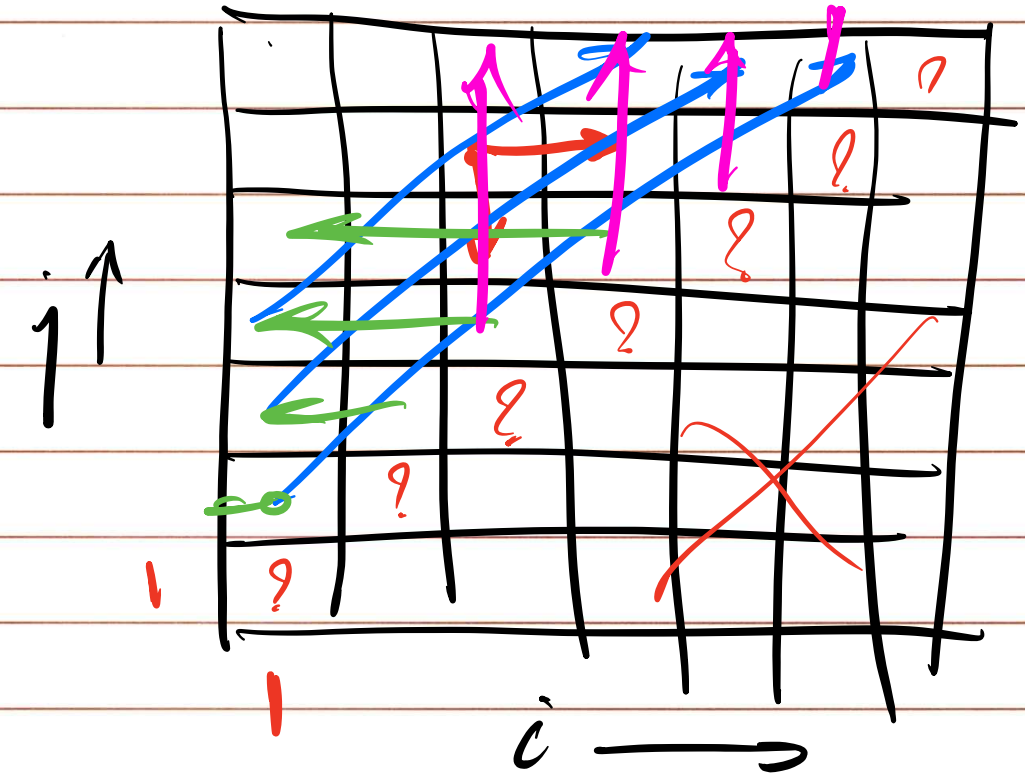
Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

$OPT(i,j)$ = Max. no of sq's from digit a_i to digit a_j

$n_{ij} = a_i \ldots a_j$

$OPT(i,j) = Max(OPT(i+1,j), OPT(i,j-1)) + IS\_Square(n_{ij})$

IS_Square() is a function that returns a 1 if the number passed to it is a perfect square and returns 0 otherwise.



Dynamic Programming

Sequence Alignment Problem

A DNA strand consists of a string of molecules called bases!

4 Types of bases:

- Adenine A
- Cytosine C
- Guanine G
- Thymine T

$S_1 = A C C G G T C G$

$S_2 = C C A G G T G G C$

```
A CC _ GGTC G _
_ CC A GGTG GC
```

3 gaps

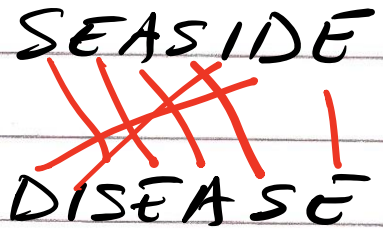
mismatch

Suppose we have 2 strings X & Y.

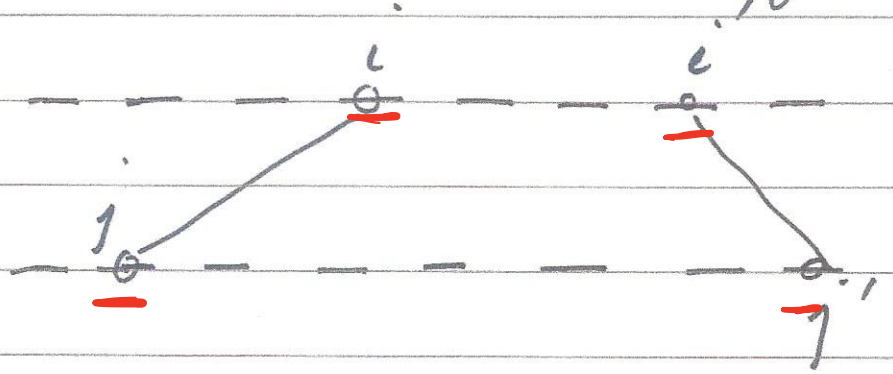
$X = \{x_1 x_2 \ldots x_m\}$

$Y = \{y_1 y_2 \ldots y_n\}$

Def. A matching is a set of ordered pairs with property that each item occurs at most once.



Def. A matching is an alignment if there are no crossing pairs.



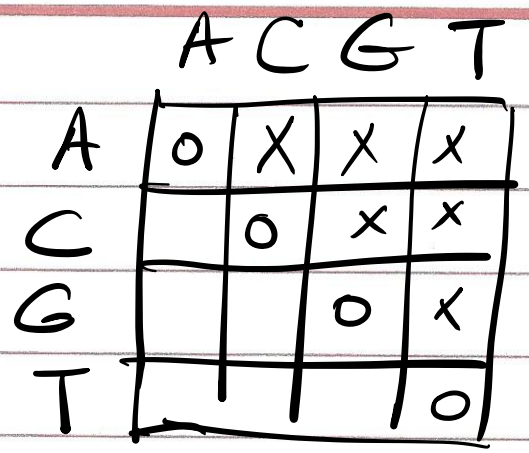
$(i, j), (i', j') \in M$

& $i < i' \rightarrow j < j'$

For an alignment M between X & Y

1- We incur a "gap penalty" of δ for each gap.

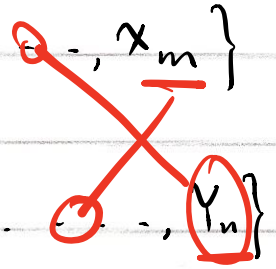
2- For each mismatch (of letters p & q) we incur a mismatch cost α_{pq}



Def Similarity between strings X & Y is the minimum cost of an alignment between X & Y.

$X = \{x_1, \ldots, x_m\}$

$Y = \{y_1, \ldots, y_n\}$



Say M is an opt. solution.

either $(x_m, y_n) \in M$ or $(x_m, y_n) \notin M$

Define $OPT(i, j)$ as the min cost of an alignment between $x_1 \ldots x_i$ & $y_1 \ldots y_j$

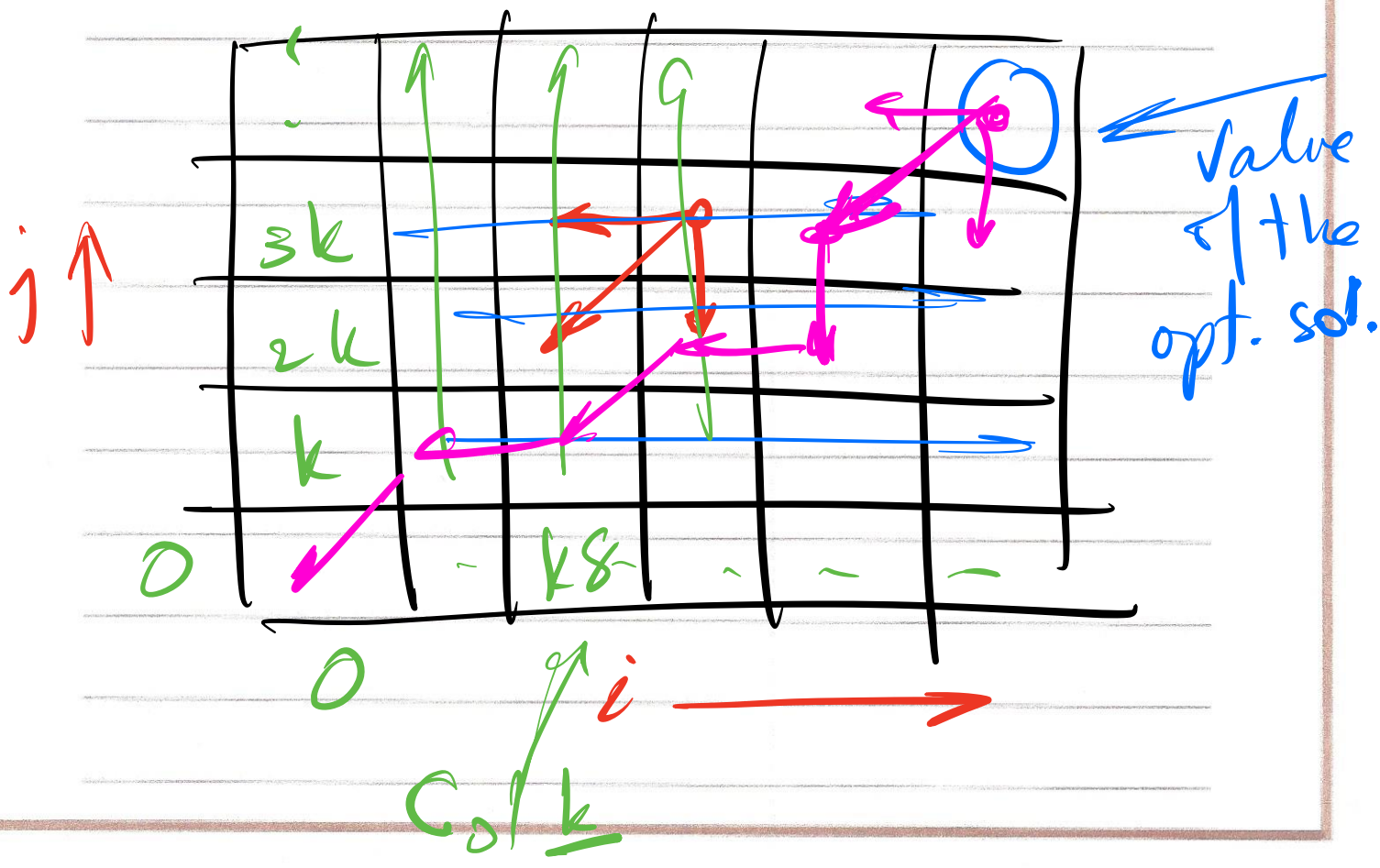
In an optimal alignment M, at least one of the following is true:

1) $(x_m, y_n) \in M \Rightarrow OPT(m,n) = OPT(m-1, n-1) + \alpha_{x_m y_n}$

2) x_m is not matched $\Rightarrow OPT(m,n) = OPT(m-1, n) + \delta$

3) y_n is not matched $\Rightarrow OPT(m,n) = OPT(m, n-1) + \delta$

1



Alignment (X, Y)

Initialize $A[i,0] = i$ & for each i

$A[0,j] = j$ & " " j

for $j = 1$ to n — $O(n)$

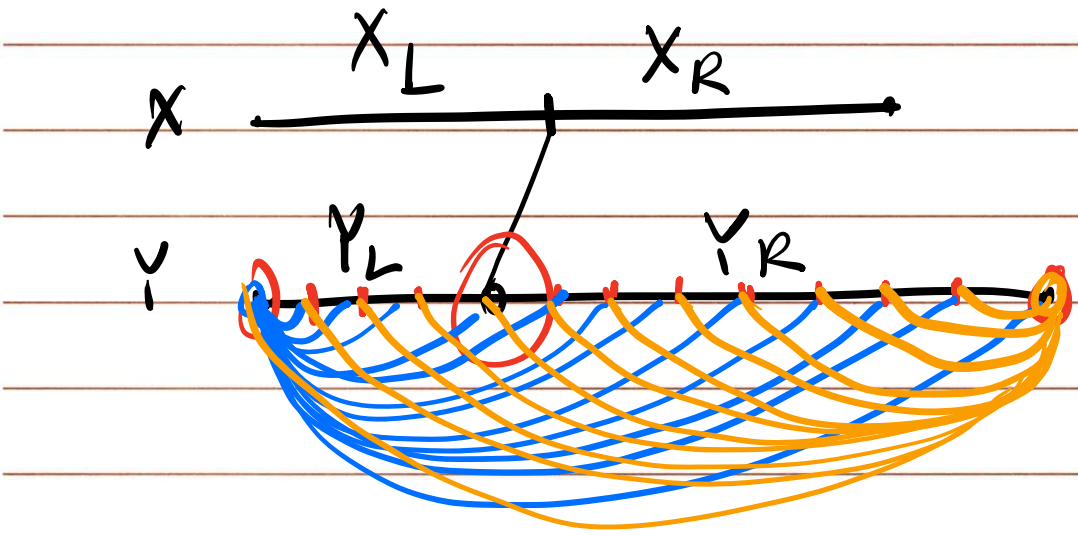
  for $i = 1$ to m — $O(m)$

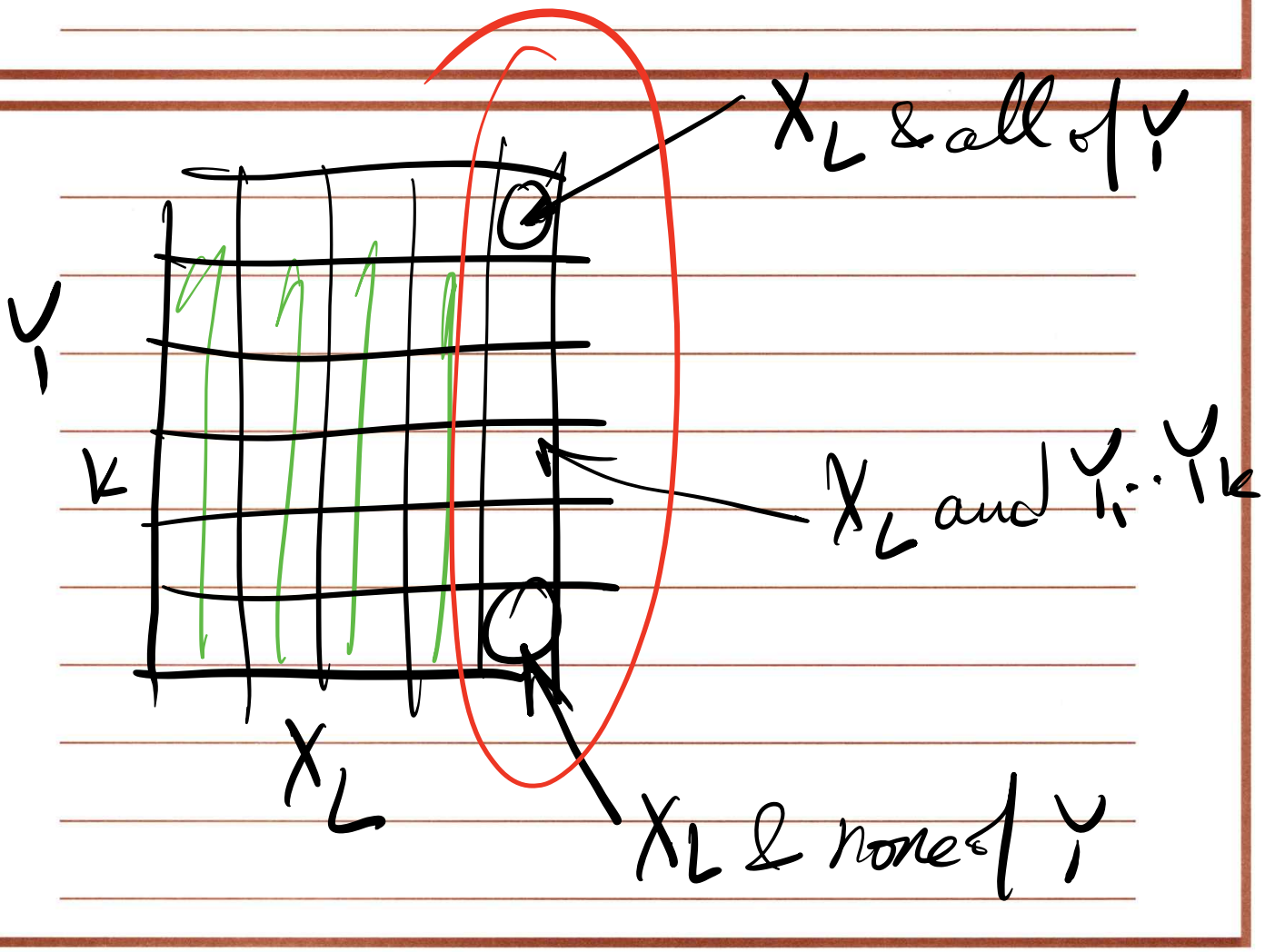
    use rec. form. (1)

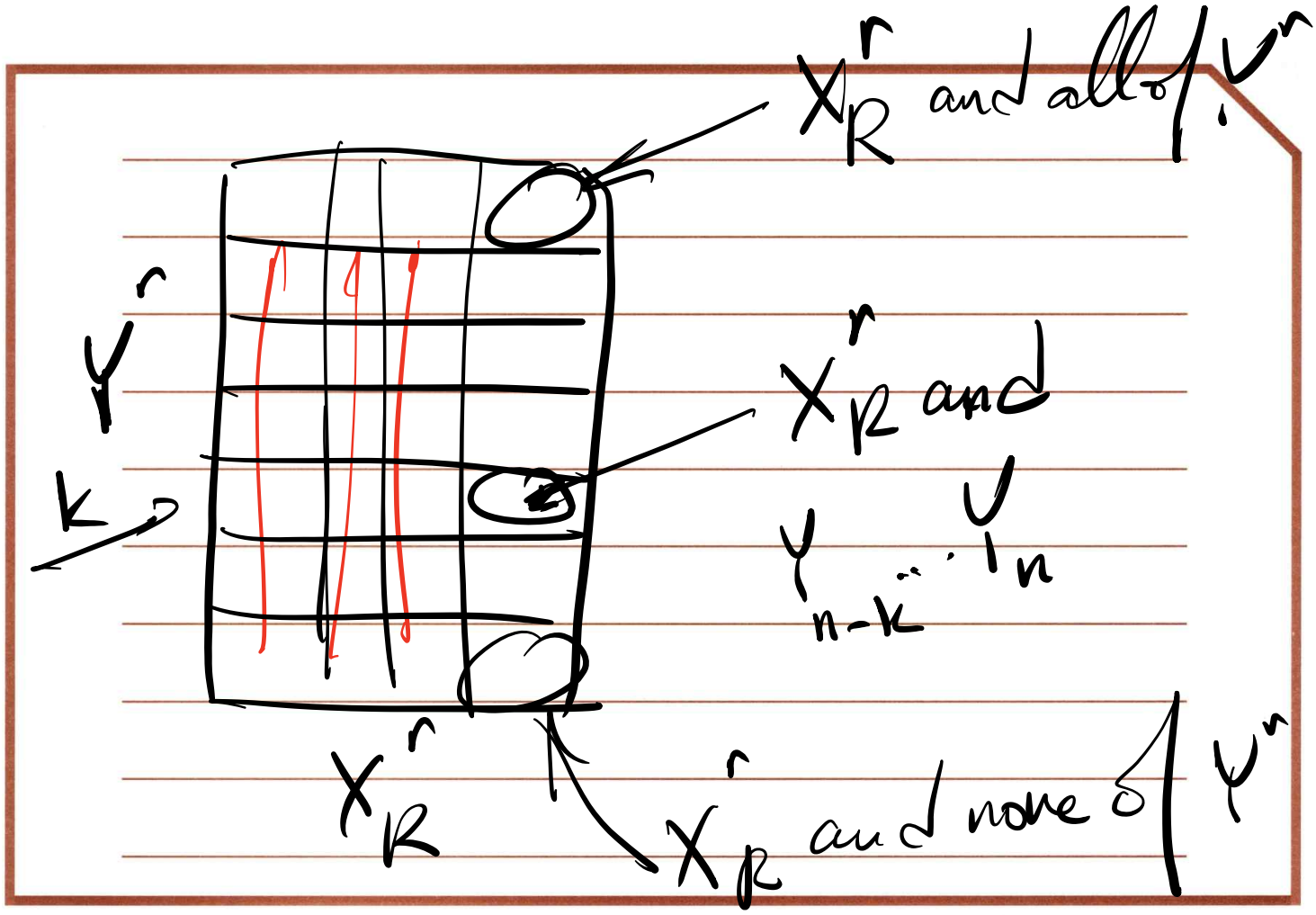
  endfor

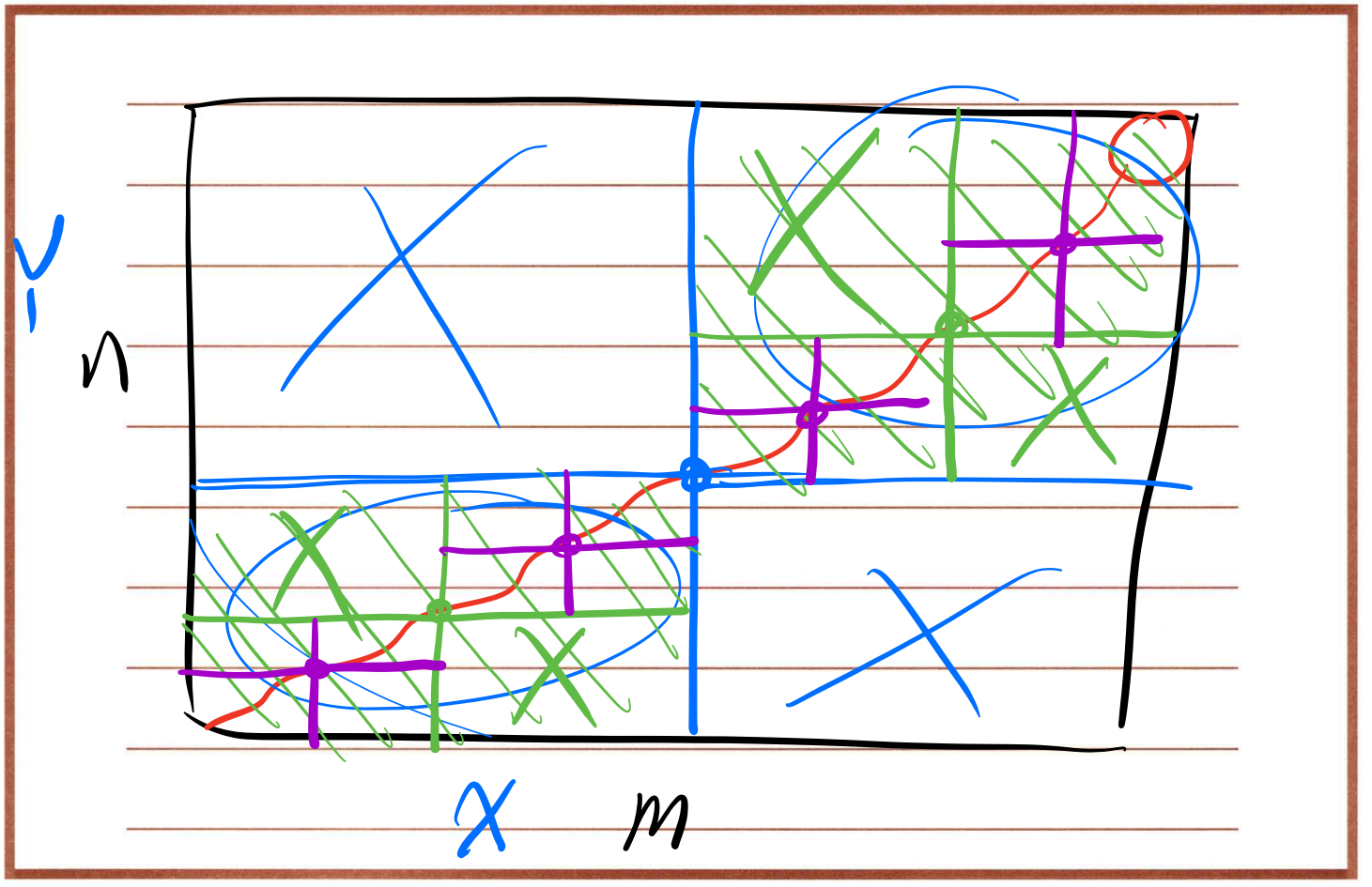
endfor

This takes $O(mn)$


X
X_L
X_R
Y
Y_L
Y_R


X_L & all of Y
Y
k
X_L and $Y_1 \ldots Y_k$
X_L
X_L & none of Y


X_R^r and all of Y^r
Y^r
X_R^r and
$Y_{n-k} \cdots Y_n$
k
X_R^r
X_R^r and none of Y^r


Y
n
X
m

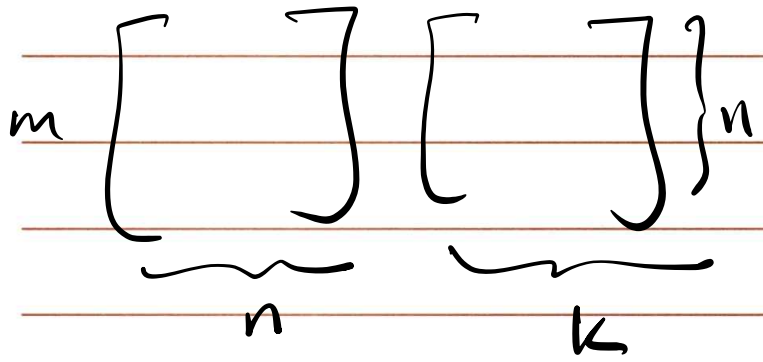
$$
\begin{aligned}
& cmn \\
& \tfrac{1}{2}\, cmn \\
& \tfrac{1}{4}\, cmn \\
& \vdots \\
\hline
& 2cmn = O(mn)
\end{aligned}
$$

Matrix Chain Multiplication

$A = A_1 \cdot A_2 \cdot \ldots \cdot A_n$

$B \cdot C \cdot D$

B is 2×10
C " 10×50
D " 50×20



total # of op's =
$m \cdot n \cdot k$

$B \cdot (C \cdot D) \Rightarrow 10,400$

$(B \cdot C) \cdot D \Rightarrow 3,000$

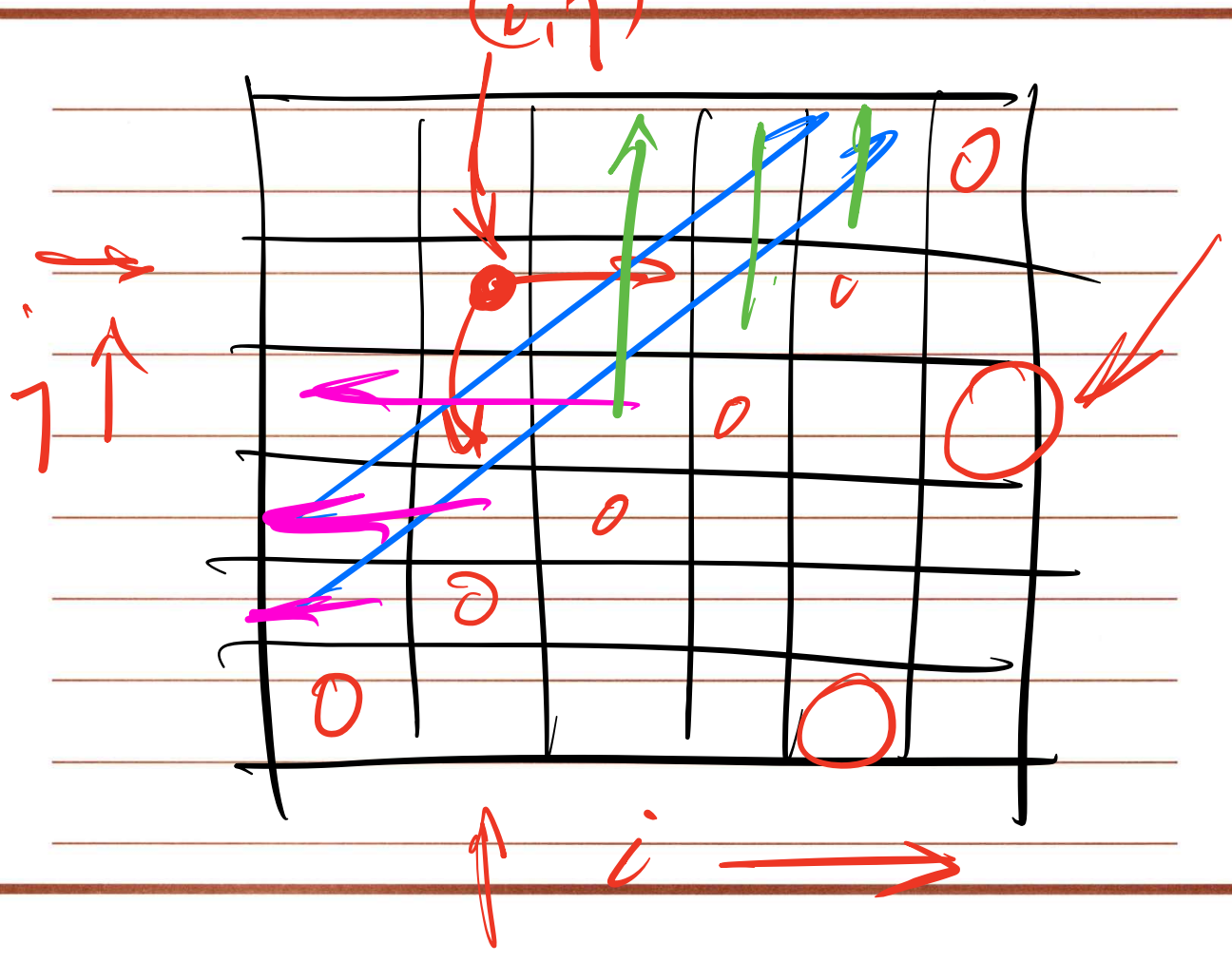
$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

$OPT(i,j)$ = opt. cost of multiplying matrices i thru j

$$OPT(i,j) = \min_{i \le k < j} \{ OPT(i,k) + OPT(k+1,j) + R_i C_k C_j \}$$

2

$O(n)$

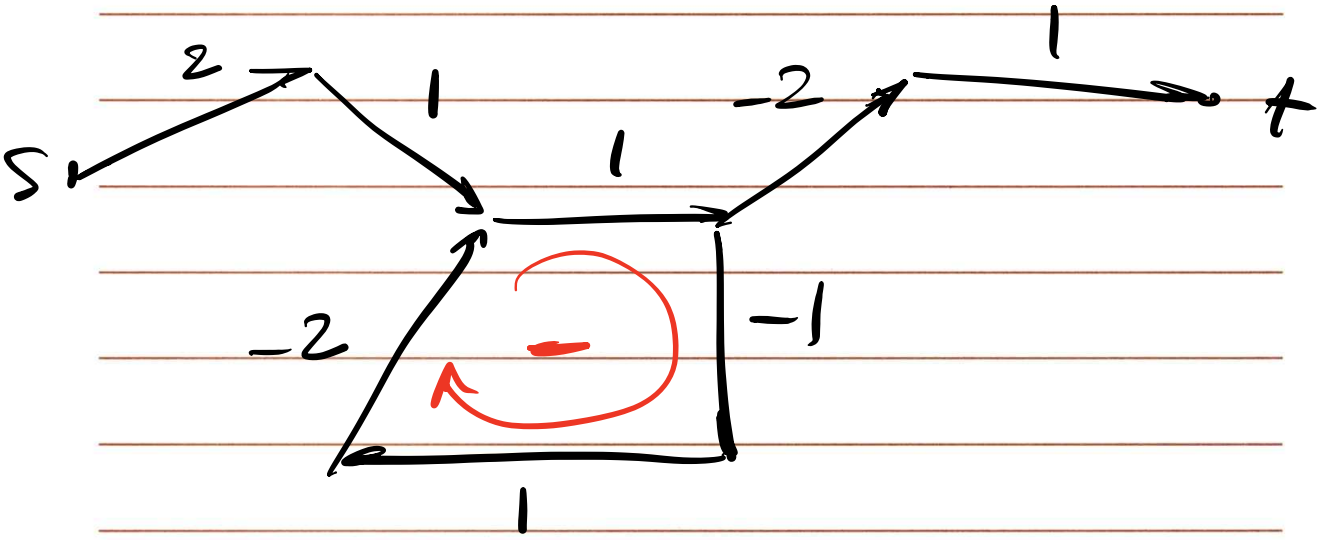


```
for i=1 to n
    opt(i,i) = 0
endfor
for j = 2 to n
    for i = j-1 to 1 by -1
        use rec. form. (2)
    endfor
endfor
```

This takes $O(n^3)$

Shortest Path Problem

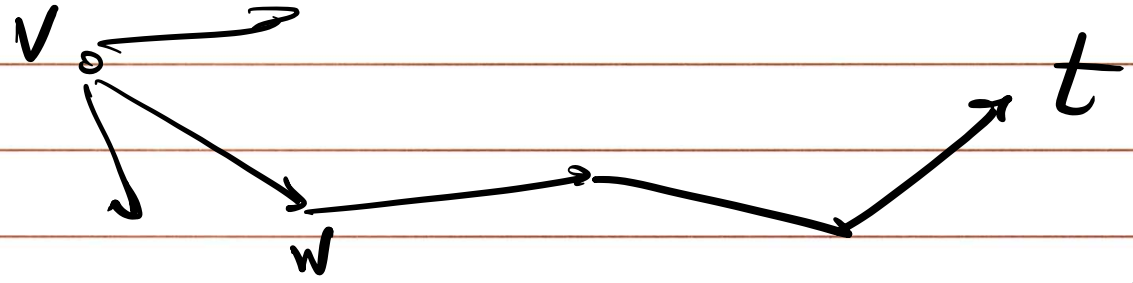
Dynamic Programming



If G has no negative cycles. Then there is a shortest path from s to t that is simple and hence has at most $n-1$ edges.

$OPT(i, v)$ denote the min cost of a v-t path using at most i edges.

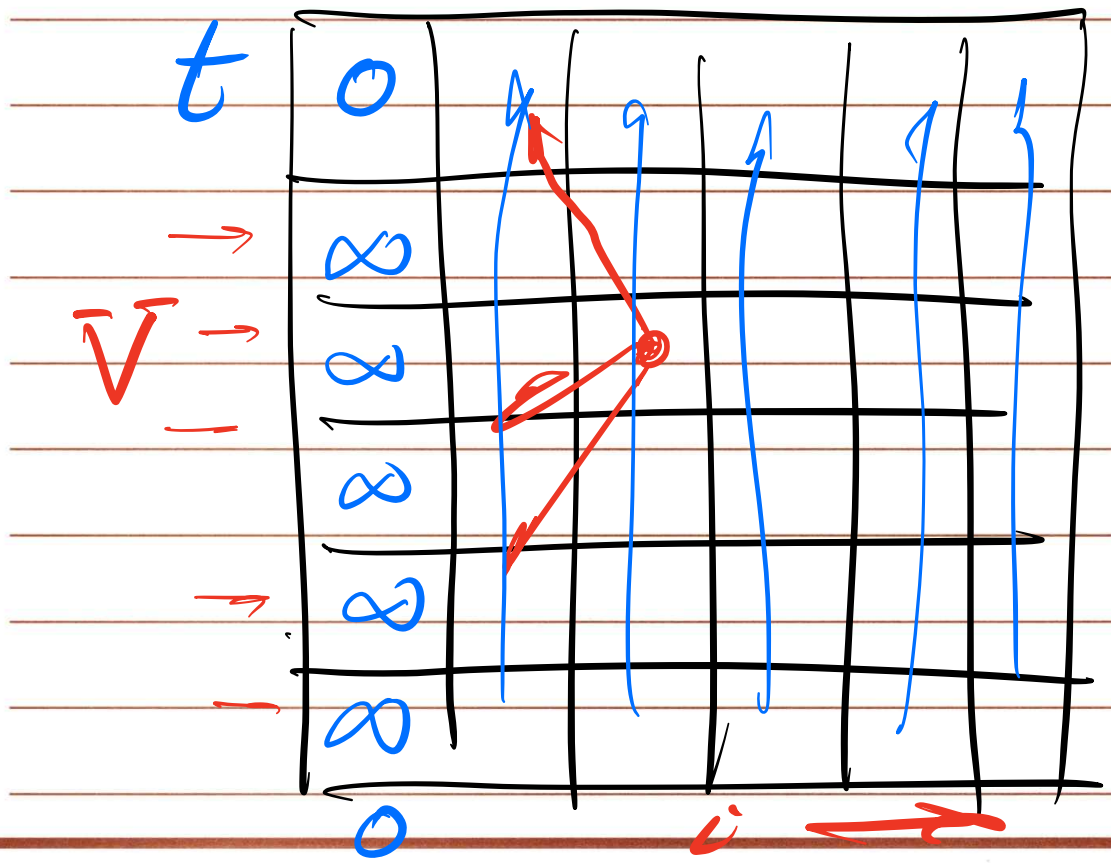
we want to find $OPT(n-1, s)$



$$OPT(i,v) = \min_{w \in Adj_v} \left(c_{vw} + OPT(i-1, w) \right)$$

Rec. formula:

$$OPT(i,v) = \min \left[OPT(i-1, v), \min_{w \in Adj_v} \left(c_{vw} + OPT(i-1, w) \right) \right]$$


t
0
∞
∞
∞
∞
∞
V
0
i

Bellman-Ford Alg.

Shortest-path (G, s, t)

n = no. of nodes in G

define $M[0,t] = 0$, $M[0,v] = \infty$

for $i = 1$ to $n-1$

  for $v \in V$ in any order

    $M[i,v] = \min(M[i-1,v], \min_{w \in Adj(v)}(M[i-1,w] + c_{vw}))$

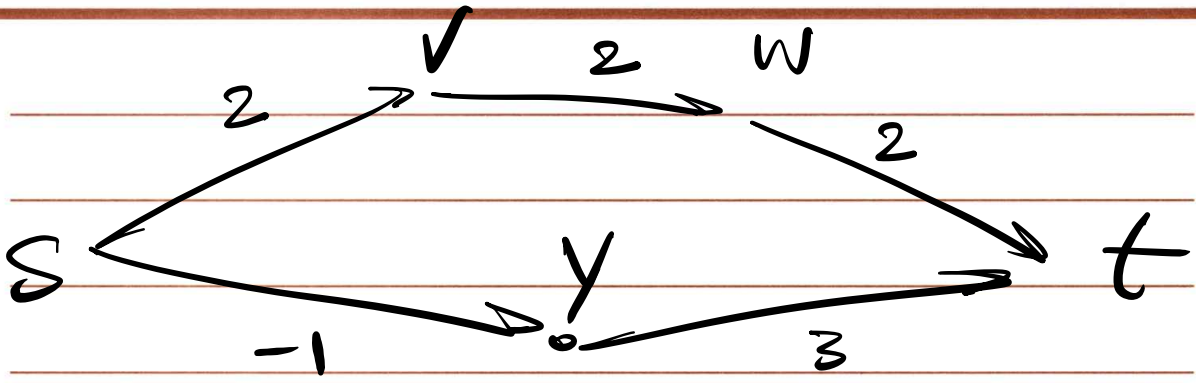
  end for

end for

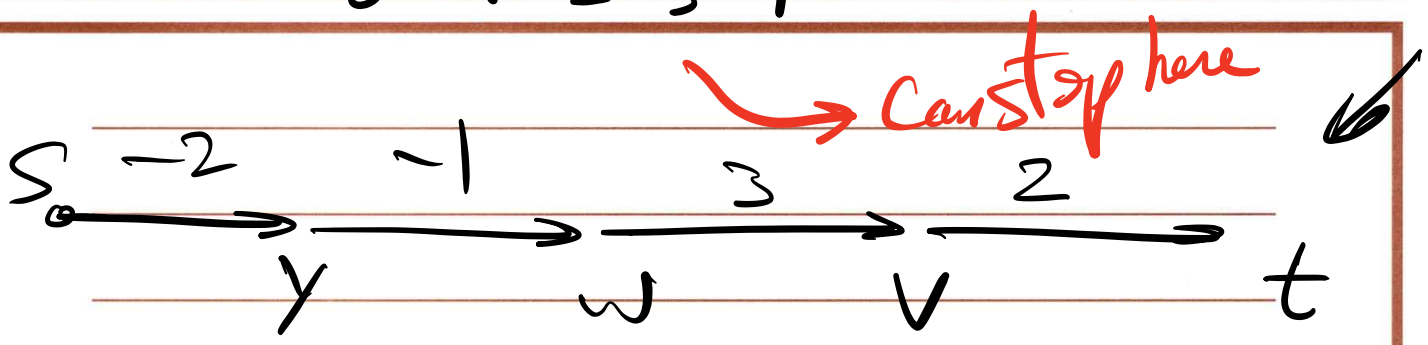
$O(n)$ $O(n)$ $O(n)$

Takes $O(n^3)$

$\hookrightarrow O(mn)$

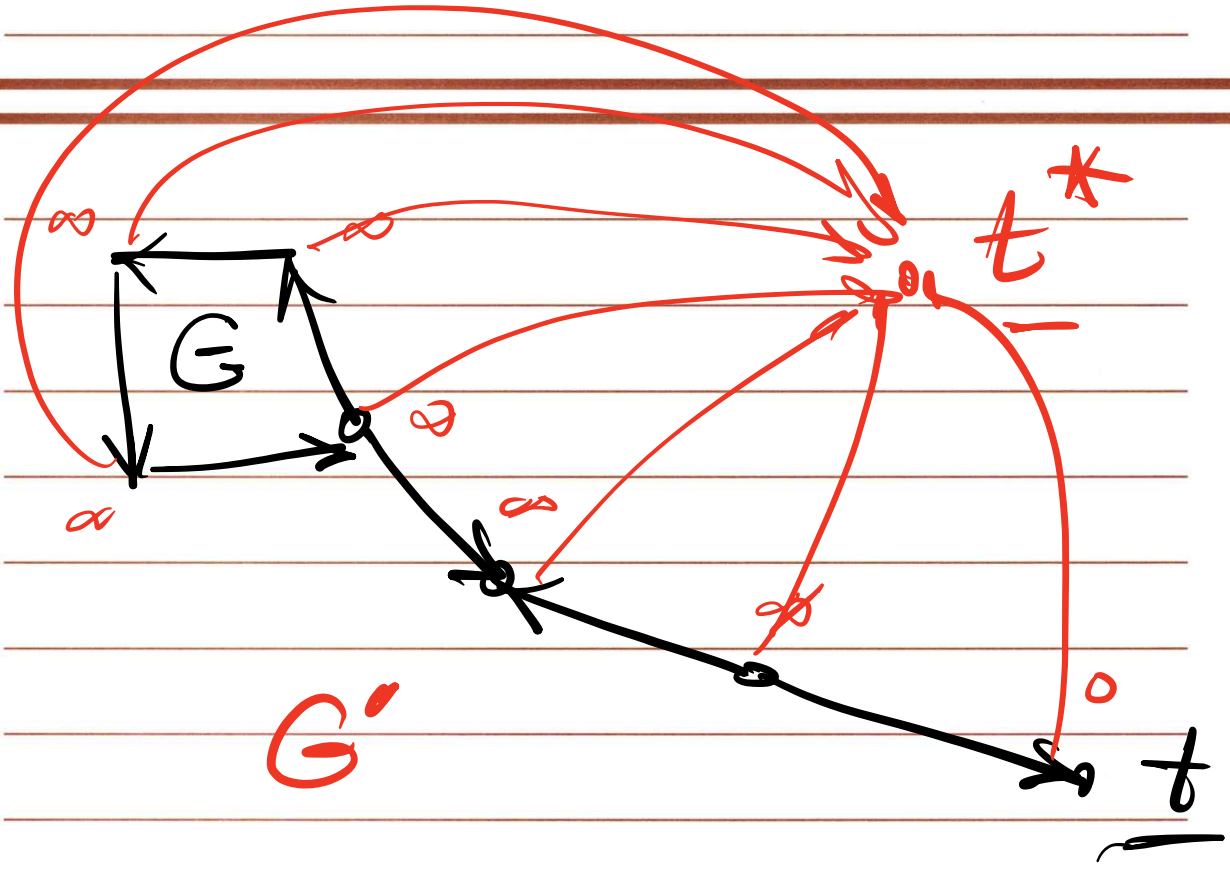


	0	1	2	3	4
t	0	0	0	0	
y	∞	3	3	3	
w	∞	2	2	2	
v	∞	∞	4	4	
s	∞	∞	2	2	



	0	1	2
t	0	0	0
v	∞	2	2
w	∞	5	5
y	∞	4	4
s	∞	2	2

	0	1	2	3	4
t	0	0	0	0	0
v	∞	2	2	2	2
w	∞	∞	5	5	5
y	∞	∞	∞	4	4
s	∞	∞	∞	∞	2



Bellman Ford

$O(mn)$

CPU Cost
FIO Cost
Comm. Cost

Dijkstra's

$O(m \log n)$

fastest
↕
Slowest

Discussion 7

1. When their respective sport is not in season, USC's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per semester, so the athletic department is not always able to help every deserving charity. For the upcoming semester, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student-athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university. Our goal is to maximize the goodwill generated for the university subject to these constraints. Note that each project must be undertaken entirely or not done at all -- we cannot choose, for example, to do half of project i to get half of g_i goodwill.

2. Suppose you are organizing a company party. The corporation has a hierarchical ranking structure; that is, the CEO is the root node of the hierarchy tree, and the CEO's immediate subordinates are the children of the root node, and so on in this fashion. To keep the party fun for all involved, you will not invite any employee whose immediate superior is invited. Each employee *j* has a value v_j (a positive integer), representing how enjoyable their presence would be at the party. Our goal is to determine which employees to invite, subject to these constraints, to maximize the total value of invitees.

3. You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

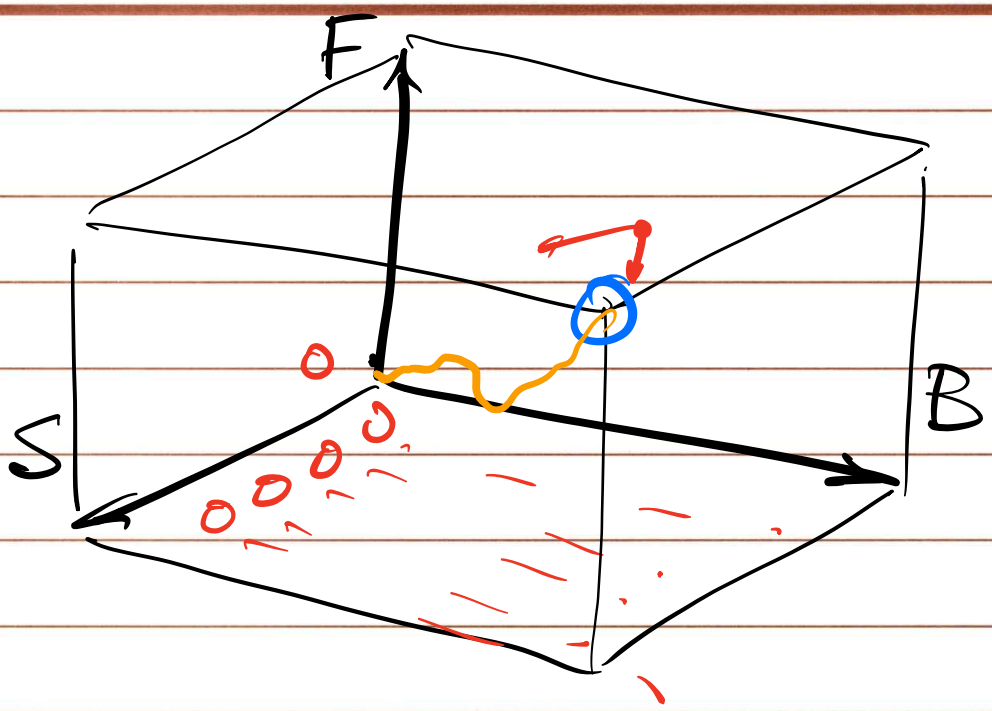
1. When their respective sport is not in season, USC's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per semester, so the athletic department is not always able to help every deserving charity. For the upcoming semester, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student-athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university. Our goal is to maximize the goodwill generated for the university subject to these constraints. Note that each project must be undertaken entirely or not done at all -- we cannot choose, for example, to do half of project i to get half of g_i goodwill.

0-1 knapsack

OPT(i, w) = opt. value of the sol. w/ req's 1..i & cap w.

OPT(i, S, B) = opt. value of the sol. for proj's 1..i w/ S students & B buses

$$OPT(i,S,B) = \max(g_i + OPT(i-1, S-s_i, B-b_i), OPT(i-1, S, B))$$

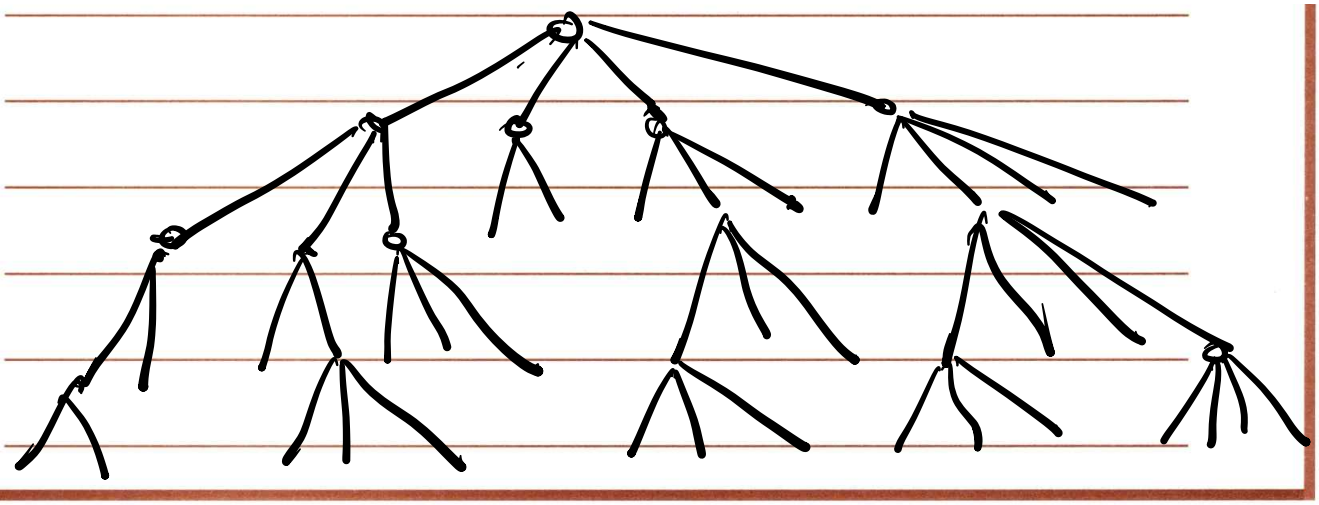


This takes $O(BFS)$

$BFS = F\, 2^{\log_2 B \cdot \log_2 S}$

Top down pass will take $O(F)$

2. Suppose you are organizing a company party. The corporation has a hierarchical ranking structure; that is, the CEO is the root node of the hierarchy tree, and the CEO's immediate subordinates are the children of the root node, and so on in this fashion. To keep the party fun for all involved, you will not invite any employee whose immediate superior is invited. Each employee j has a value v_j (a positive integer), representing how enjoyable their presence would be at the party. Our goal is to determine which employees to invite, subject to these constraints, to maximize the total value of invitees.



$OPT(i)$ = Max. fun factor for subtree rooted at node i

$$OPT(i) = \max\left(v_i + \sum_{g \in g_i} OPT(g),\ \sum_{c \in c_i} OPT(c)\right)$$

Can be done in $O(n)$

3. You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

box type: $h_i = 2$, $w_i = 3$, $d_i = 5$

$2 \times (3 \times 5)$

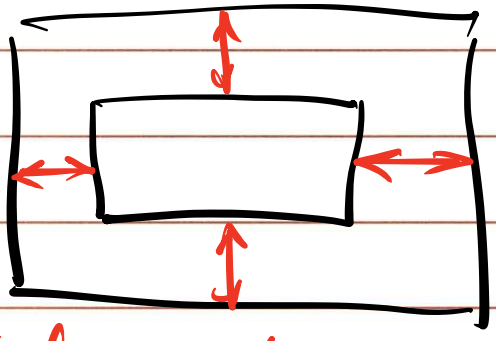
$3 \times (2 \times 5)$

$5 \times (2 \times 3)$

3 boxes

depth ↑ width

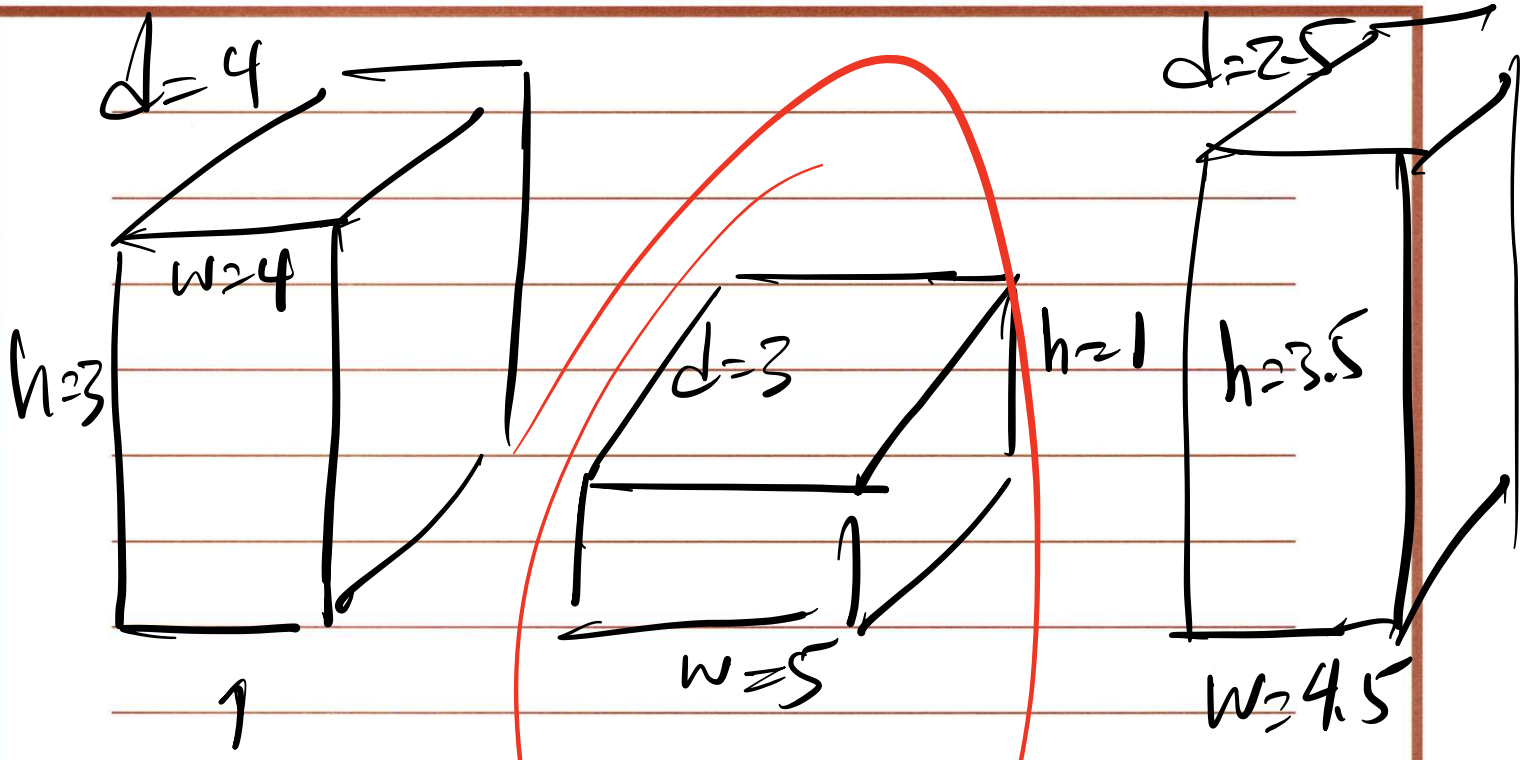
n box types → 3n boxes



can sort boxes by base area.

$H(j)$ = height of the tallest stack of boxes $1 \ldots j$

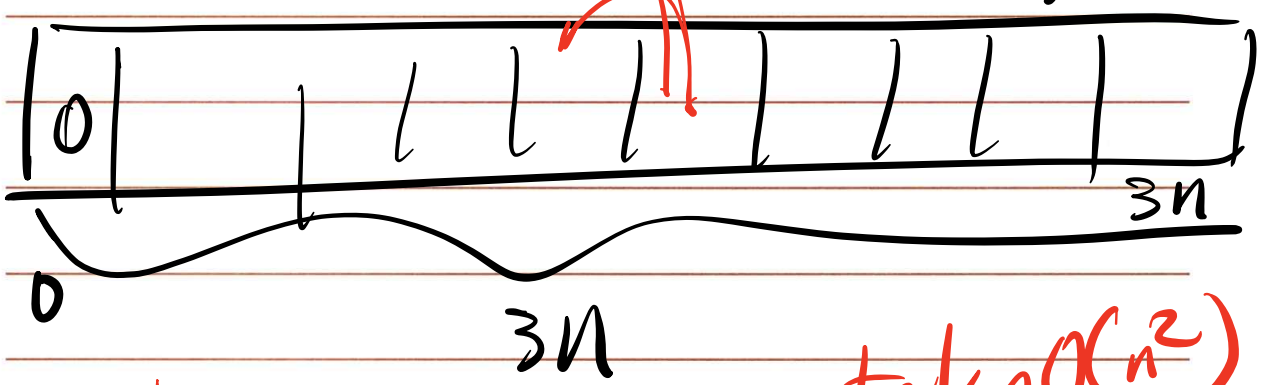
$$H(j) = \max\Big[\, H(j-1),\ h_j + \max_{1 \le k < j \,\&\, w_k > w_j \,\&\, d_k > d_j} (H(k)) \,\Big]$$



$H(1) = 3$, $H(2) = 3$, $H(3) = 6.5$

H(j) = Height of the tallest stack of boxes 1..j w/ j at the top of the stack.

$$H(j) = h_j + \max_{i<j \,\&\, w_i < w_j \,\&\, d_i < d_j} [H(i)]$$



need to search for the highest value in H() takes $O(n^2)$

Imagine starting with the given decimal number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence

32492 → **3249** → **324** → 24 → **4**

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

32492 → **3249** → 249 → **49** → **9**

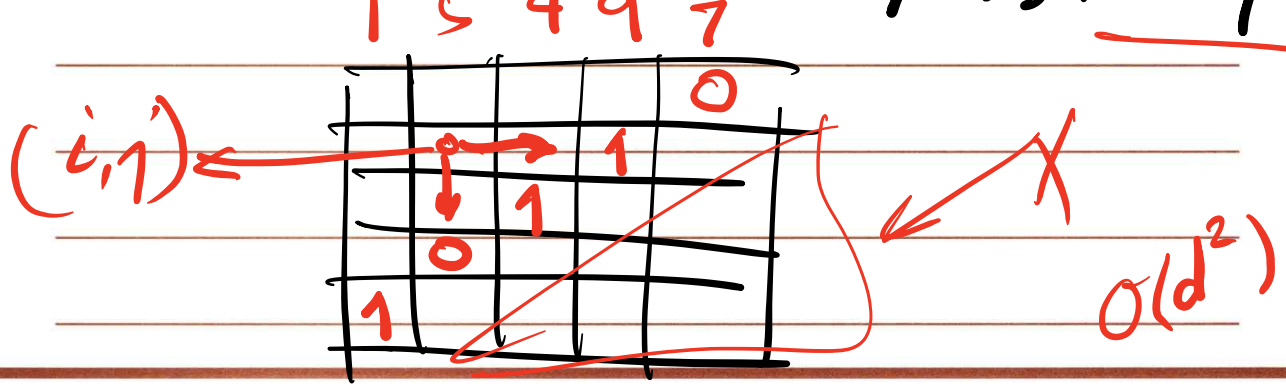
also gives you 3 perfect squares, viz. 3249, 49, and 9.

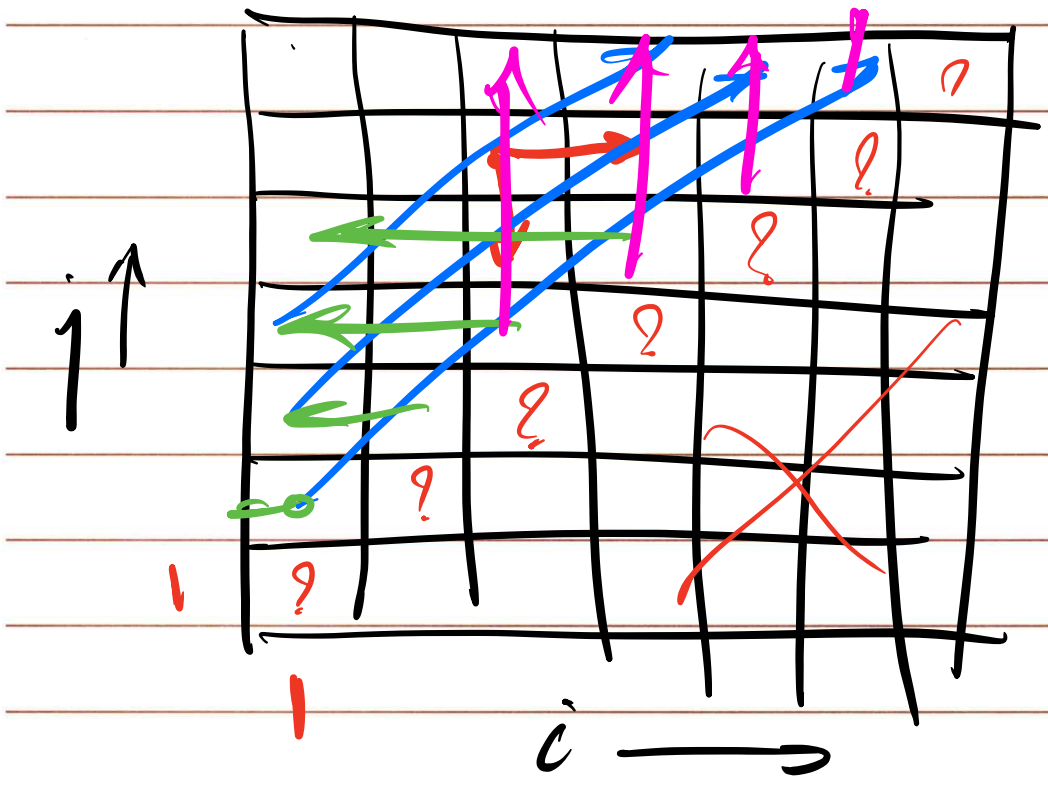
Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

$OPT(i,j)$ = Max. no of sq's from digit a_i to digit a_j

$n_{ij} = a_i \ldots a_j$

$OPT(i,j) = Max(OPT(i+1,j), OPT(i,j-1)) + IS\_Square(n_{ij})$





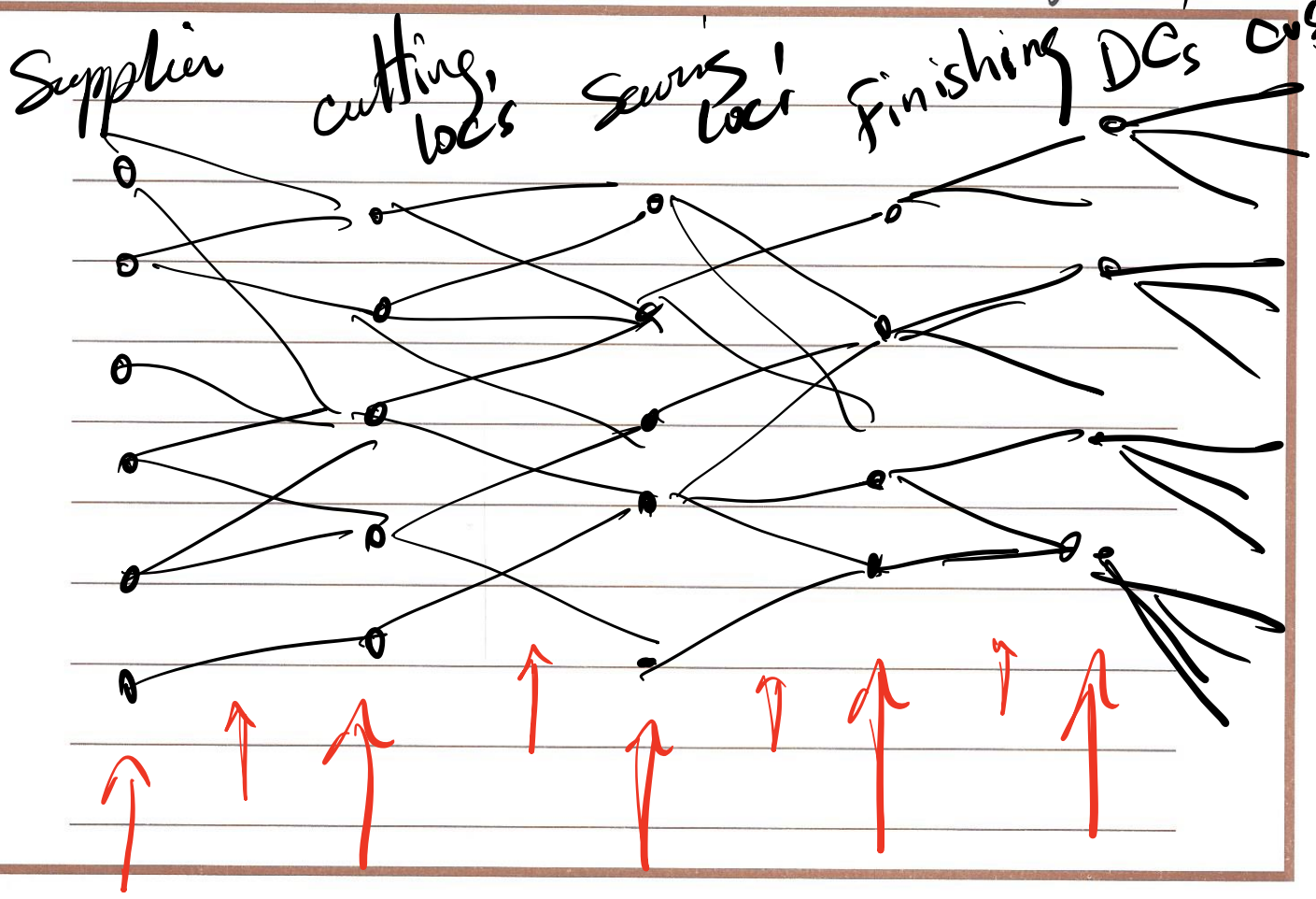
Network Flow

Examples of network flow problems:

- How much data can we send from one point in the network to another point?

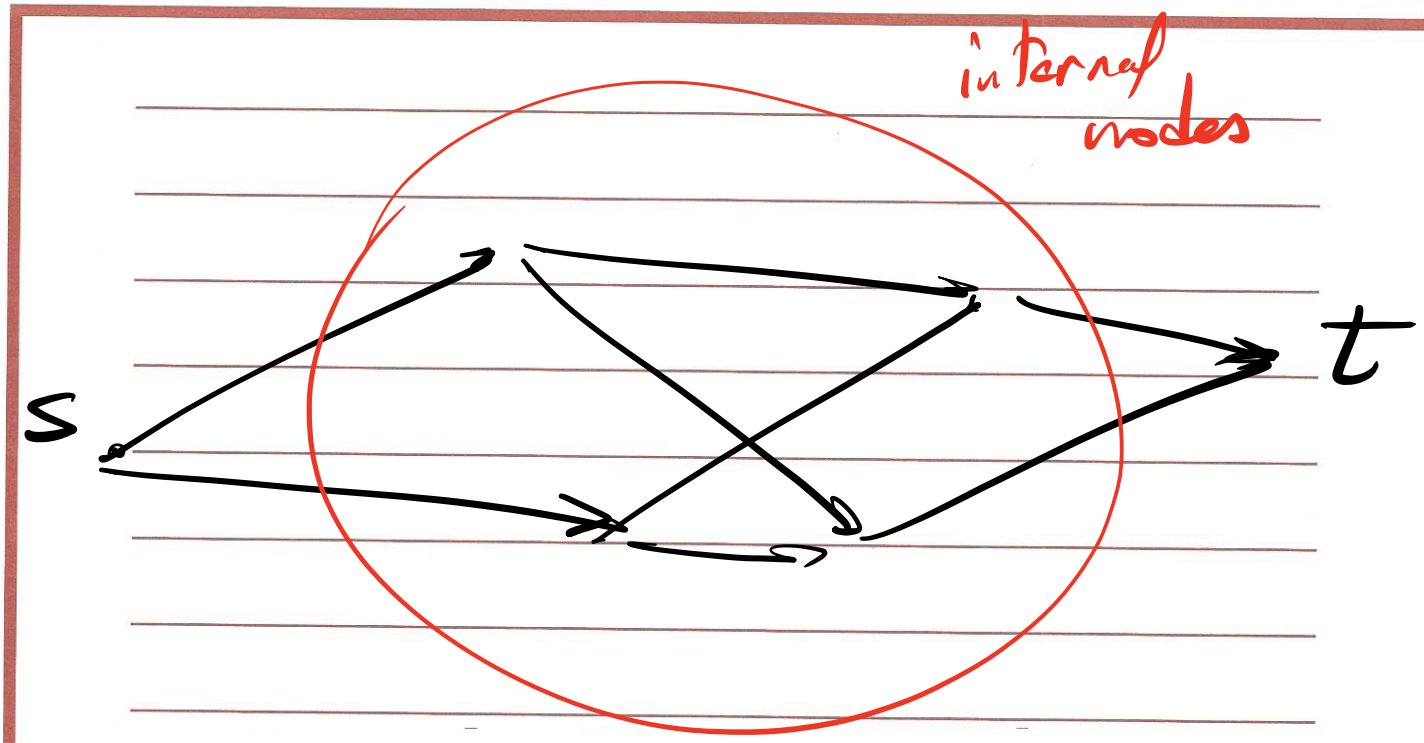
- How much traffic does the freeway system sustain for travel between two cities?

- How profitable could a manufacturing supply chain be?



Def. A flow network is a directed graph $G = (V, E)$ with following features:

- Each edge e has a non-negative capacity c_e
- Has a single source node $s \in V$
- Has a single sink node $t \in V$



Assumptions:

- no edges enter s or leave t
- at least one edge is connected to each node $O(m+n) \rightarrow O(m)$
- all capacities are integers

Notation

We will call $f(e)$ flow through edge e. $f(e)$ has the following properties:

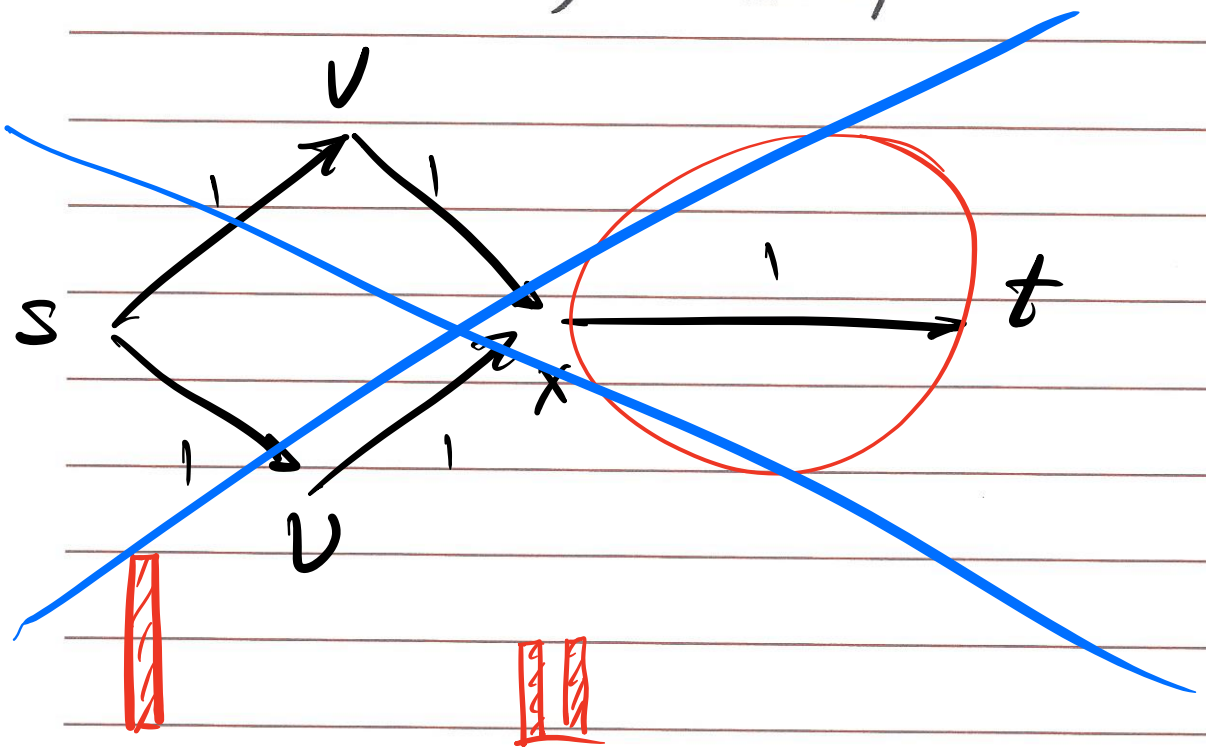
1- Capacity Constraint:

for each edge $e \in E$, $0 \leq f(e) \leq c_e$

2- Conservation of flow:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e), \text{ except for } s \& t$$

We are looking for steady state flow.

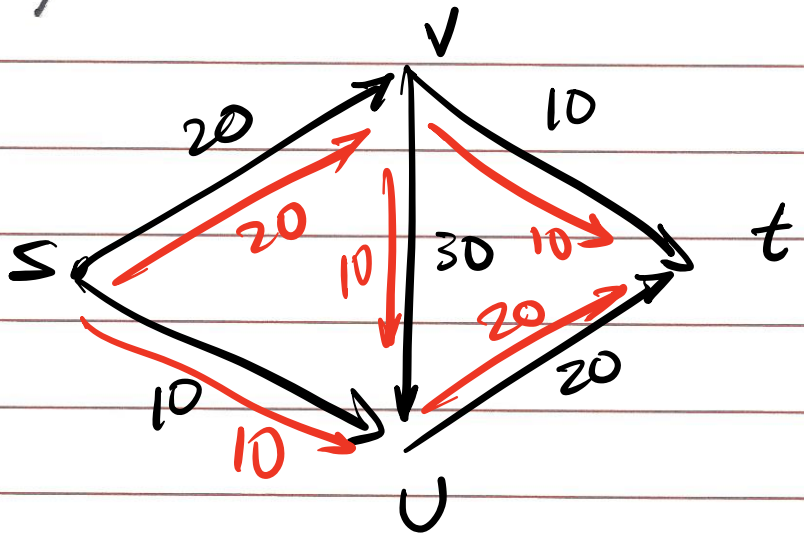


Def. For a steady state flow, the value of flow $v(f)$ is defined as follows:

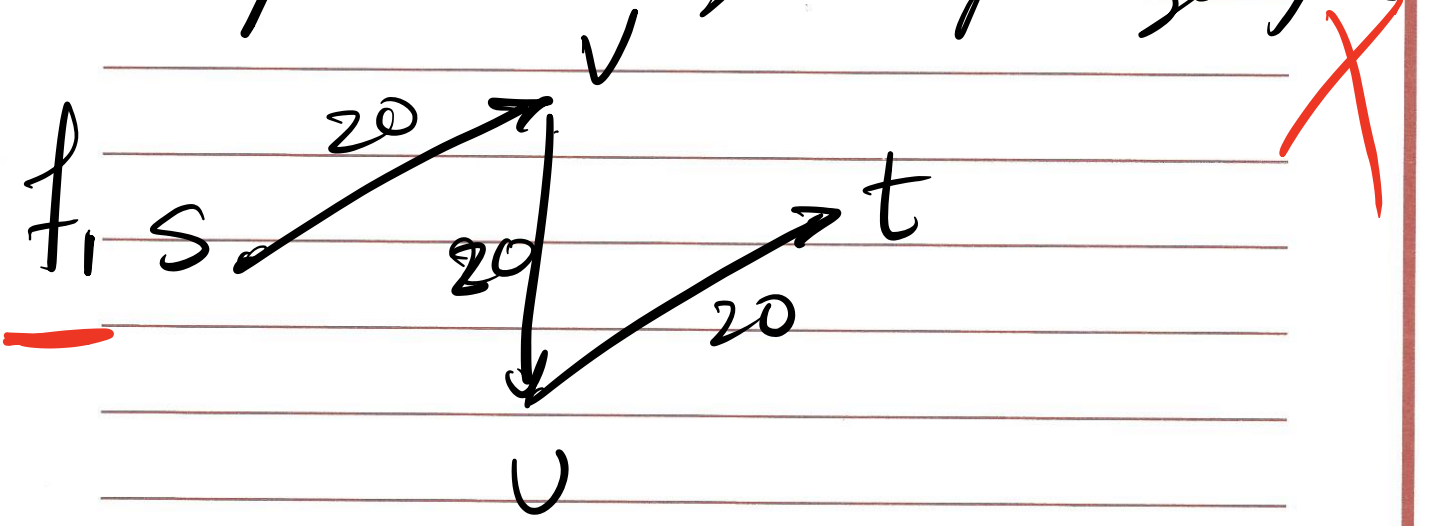
$$v(f) = \sum_{e \text{ out of } s} f(e)$$

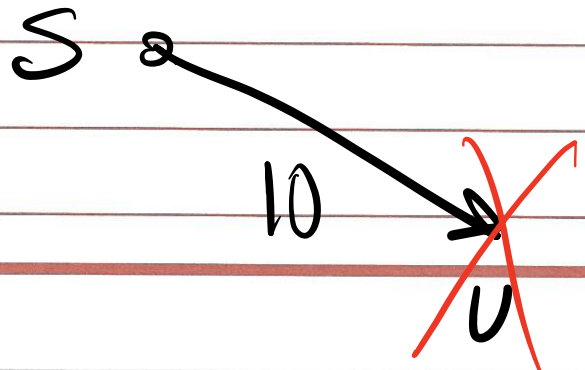
Max Flow problem

Given a flow network G, find an s-t flow with max. value.

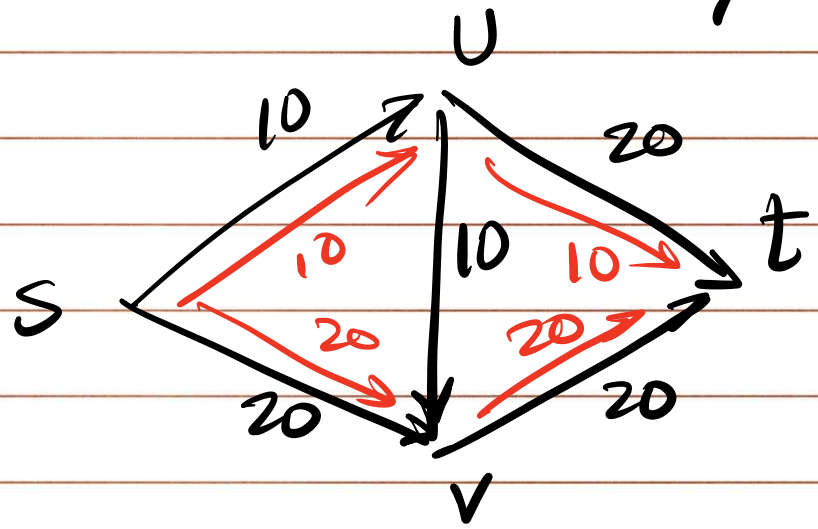


try #1 use highest cap. edges first.

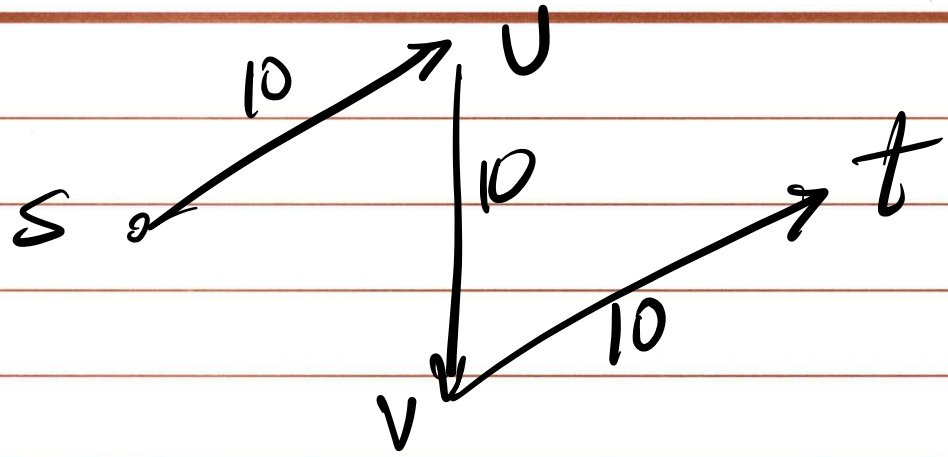




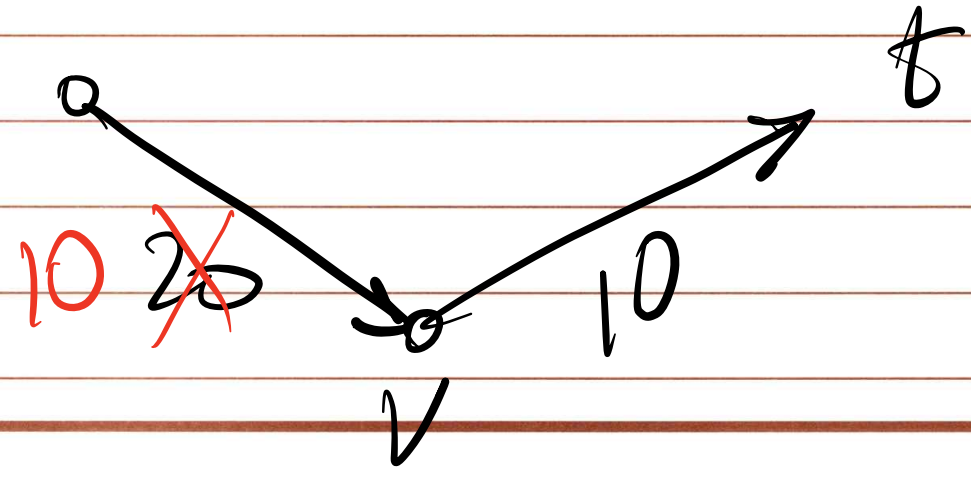
try #2 use smallest cap. edges first



f_1



f_2

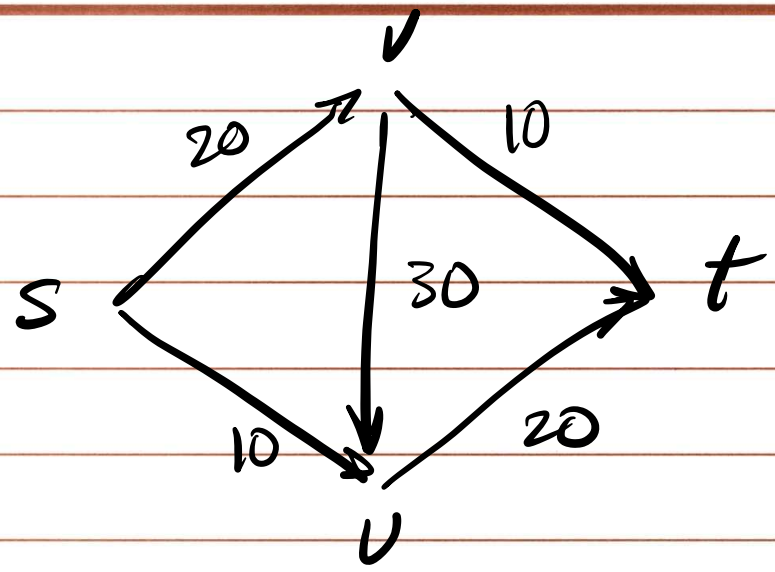


Def. G_f is the residual graph of G with the following definitions:

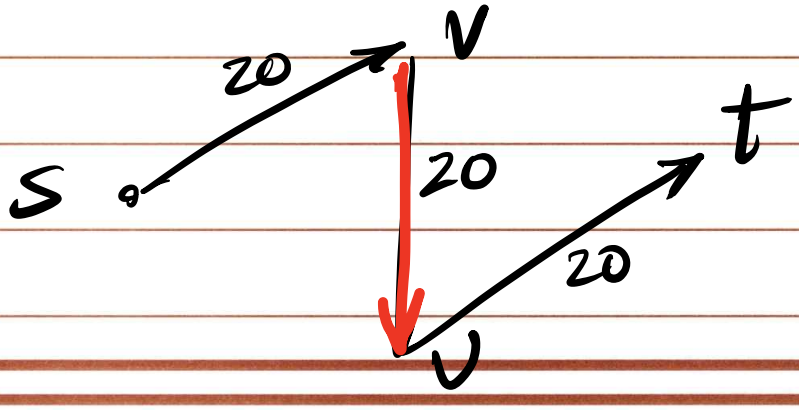
- G_f has the same set of nodes as G
- forward edge → for each edge e w/ $f(e) < c_e$, we include e in G_f with capacity $c_e - f(e)$
- backward edge → for each edge e w/ $f(e) > 0$, we include edge e' (opposite direction to e) in G_f with $f(e)$ units of capacity

To create G_f:

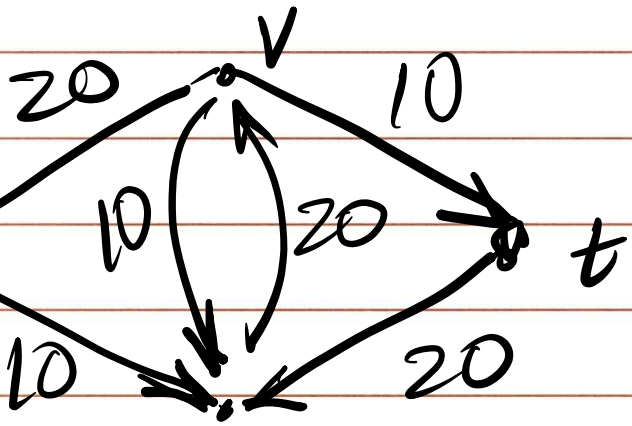
- if $f(e) = 0$
  - one forward edge w/ Cap. c_e
- if $f(e) = c_e$
  - one backward edge w/ Cap. c_e
- if $0 < f(e) < c_e$
  - one forward edge w/ Cap. $c_e - f(e)$
  - one " " w/ Cap. $f(e)$


v
20
10
s
30
t
10
20
u

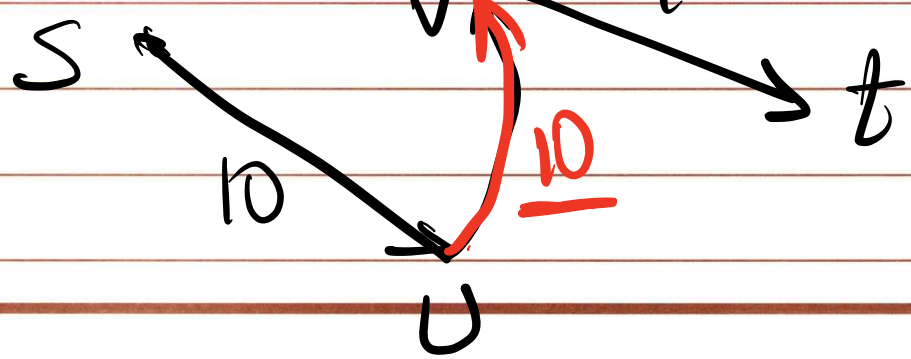
f_1


20
v
s
20
t
20
u

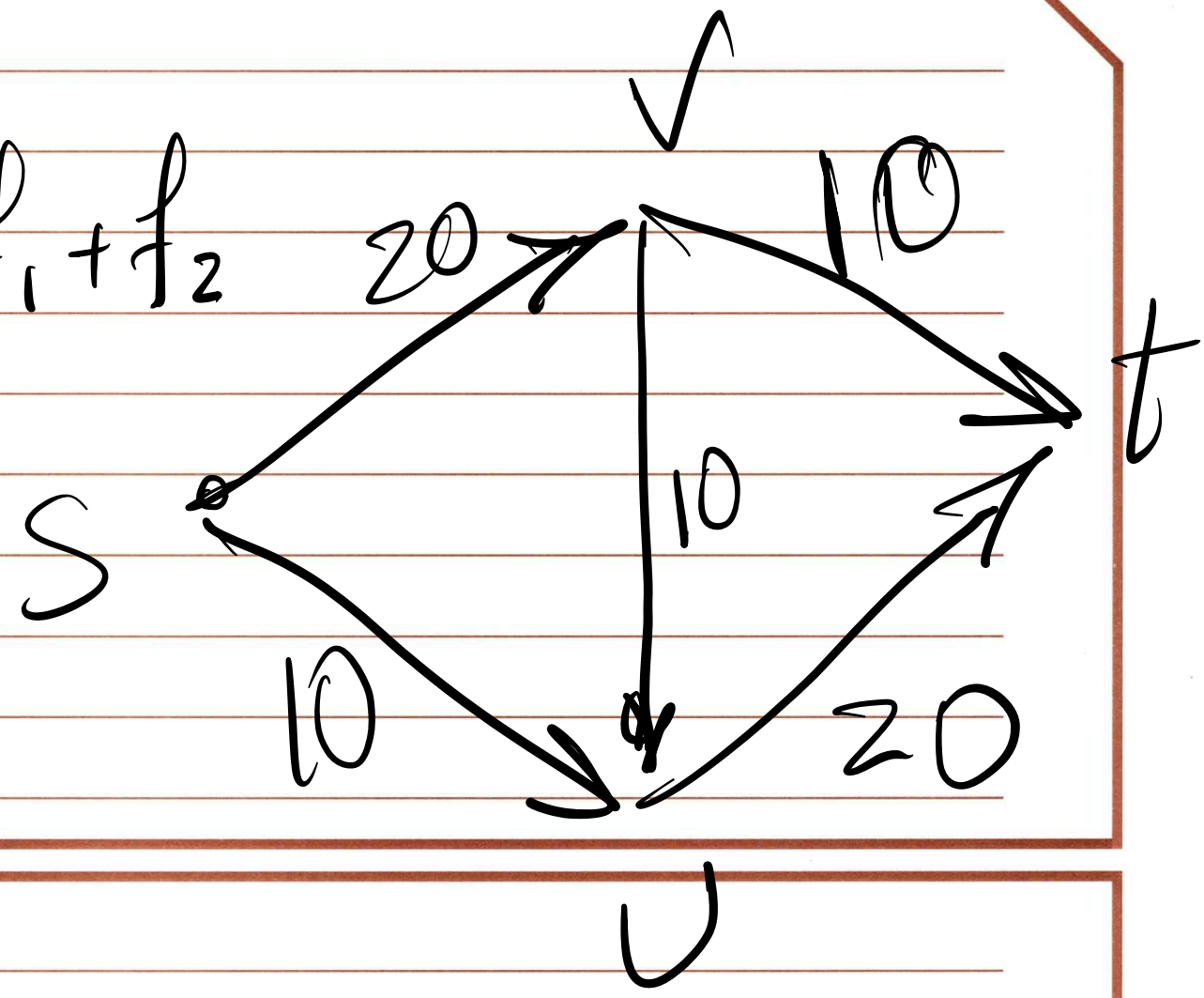
G_{f_1}


20
v
10
10
20
s
t
10
20
u

f_2


v
10
s
t
10
10
u

$f = f_1 + f_2$



Def. If P is a simple path from s to t in G_f, then bottleneck(P) is the minimum residual capacity of any edge on P.

Overall strategy to find Max Flow

- Find a path from s to t
- Find the bottleneck value for this path
- Push flow through this path with value equal to bottleneck value
- Repeat

```
Augment (f, c, P)
  Let b = bottleneck(P)
  for each edge (v,u) ∈ P
    if e=(v,u) is a forward edge,
      then increase f(e) in G by b
    else (v,u) is a backward edge
      and let e=(u,v)
      then decrease f(e) in G by b
    end if
  endfor
```

Return (f')

If f is flow before augmentation, and f' is flow after augmentation, we need to show that if f is a valid flow, then f' will also be a valid flow.

Proof: 1- Check capacity condition

Need to show that for each edge $e \in E$, we have $0 \leq f'(e) \leq c_e$

A- If e is a forward edge,

$$\text{bottleneck}(P) \leq c_e - f(e)$$

$$\underbrace{f(e) + \text{bottleneck}(P)}_{0 \leq f'(e) \leq c_e} \leq c_e - f(e) + f(e)$$

B- If e is a backward edge,

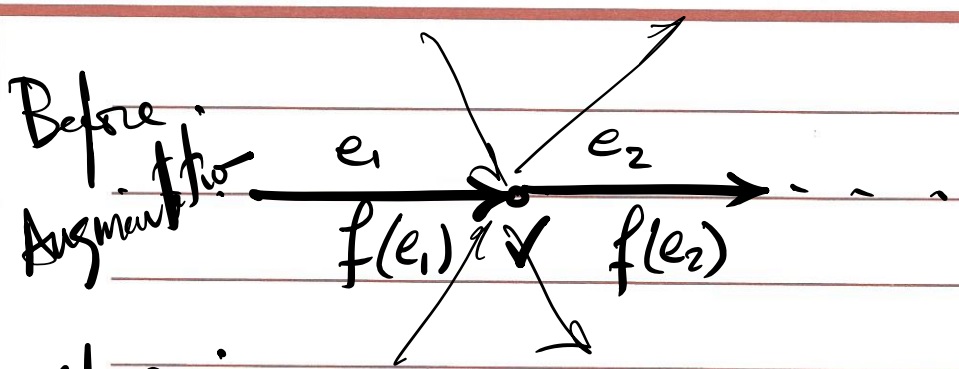
$$\text{bottleneck}(P) \leq f(e)$$

$$\underbrace{f(e) - \text{bottleneck}(P)}_{c_e \geq f'(e) \geq 0} \geq f(e) - f(e)$$

2- Check conservation of flow

Since f is a valid flow, for each node v other than s & t we have:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$



Before Aug.

Case 1 both e_1 & e_2 correspond to forward edges of G_f

Three Other Cases to examine

Ford-Fulkerson algorithm for Max Flow.

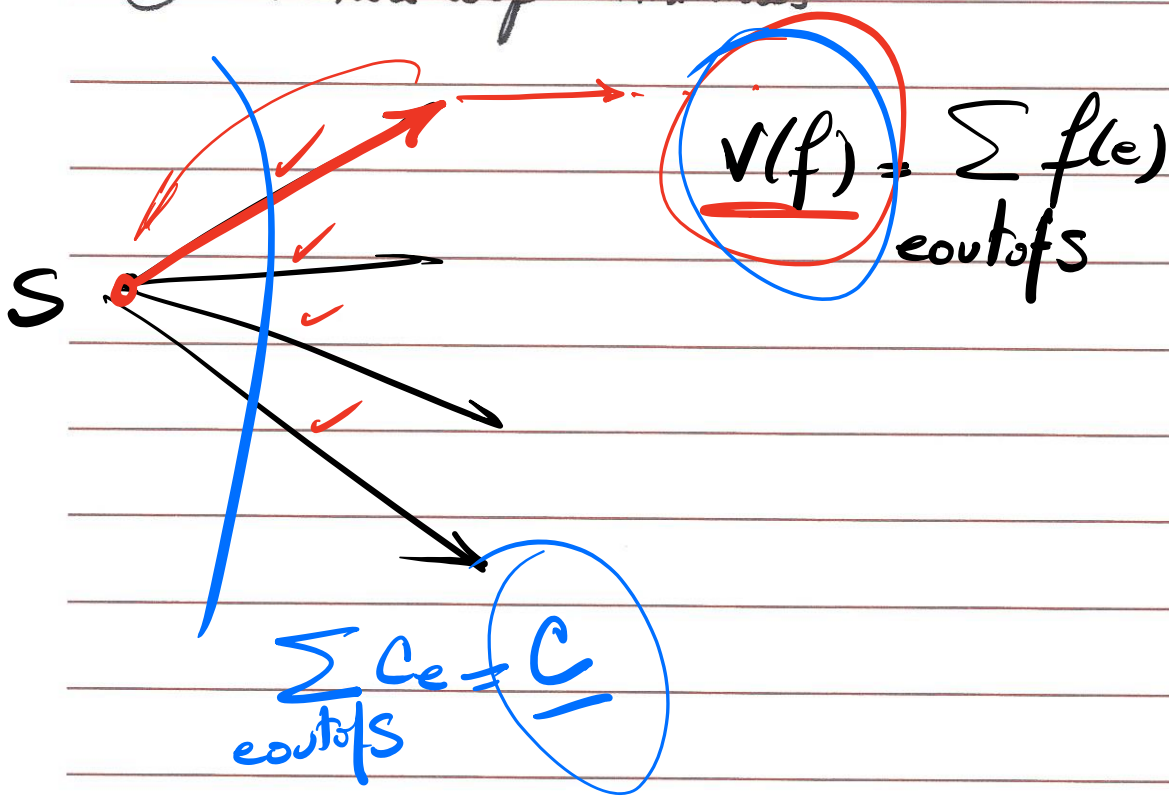
```
MaxFlow (G, s, t, c)
    Initially f(e) = 0 for all e in G
    While there is an s-t path in Gf
        let P be a simple s-t path in Gf
        f' = augment(f, c, P)
        f = f'
        update Gf
    endwhile
```

Return f.

Proof of correctness should include:

- Proof of termination
- Proof that f is a Max Flow.

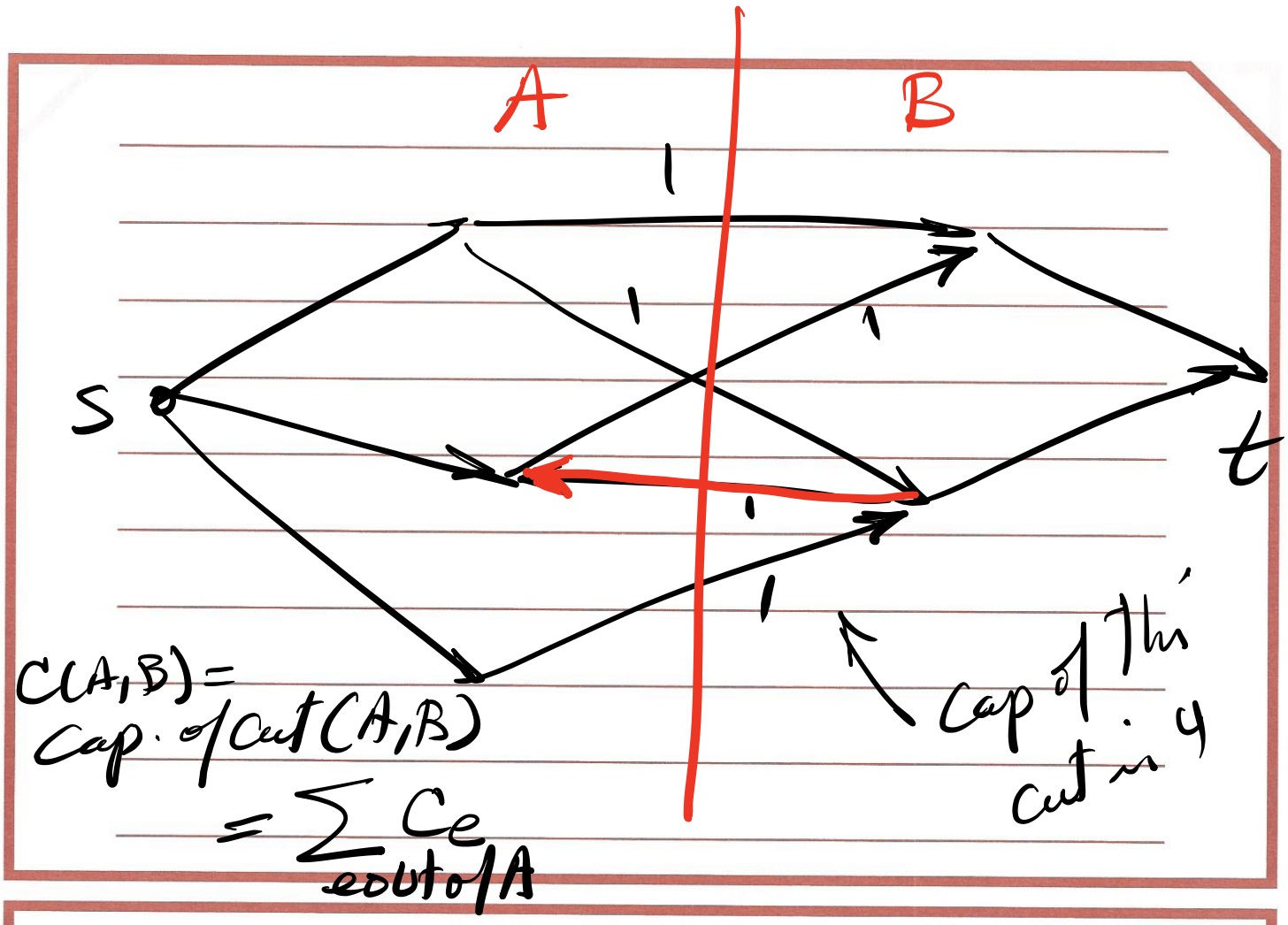
① while loop terminates



② f is a Max Flow

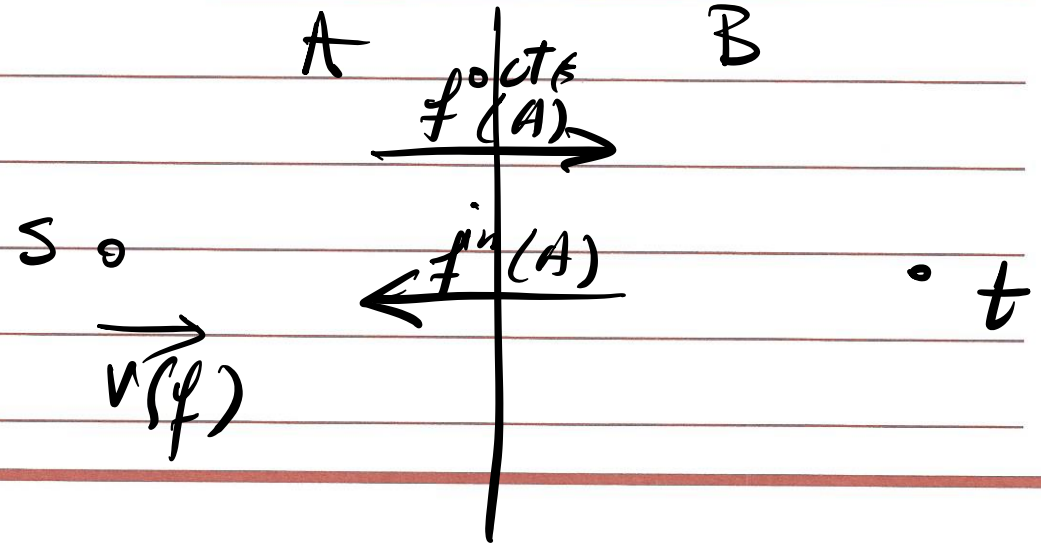
We first need some definitions

Define a cut: A cut divides nodes in the graph into 2 sets A & B such that $s \in A$ and $t \in B$



FACT: Let f be any s-t flow and (A,B) any s-t cut, then

$$v(f) = f^{out}(A) - f^{in}(A)$$



Max. value of the flow $\leq$ Cap. of the (A,B) cut

Proof: $v(f) = f^{out}(A) - f^{in}(A)$ ≥ 0

$$v(f) \leq f^{out}(A) \leq \sum_{\text{edges out of } A} c_e$$

$$v(f) \leq c(A,B)$$

So, the MaxFlow is bounded by the cap. of every s-t cut.

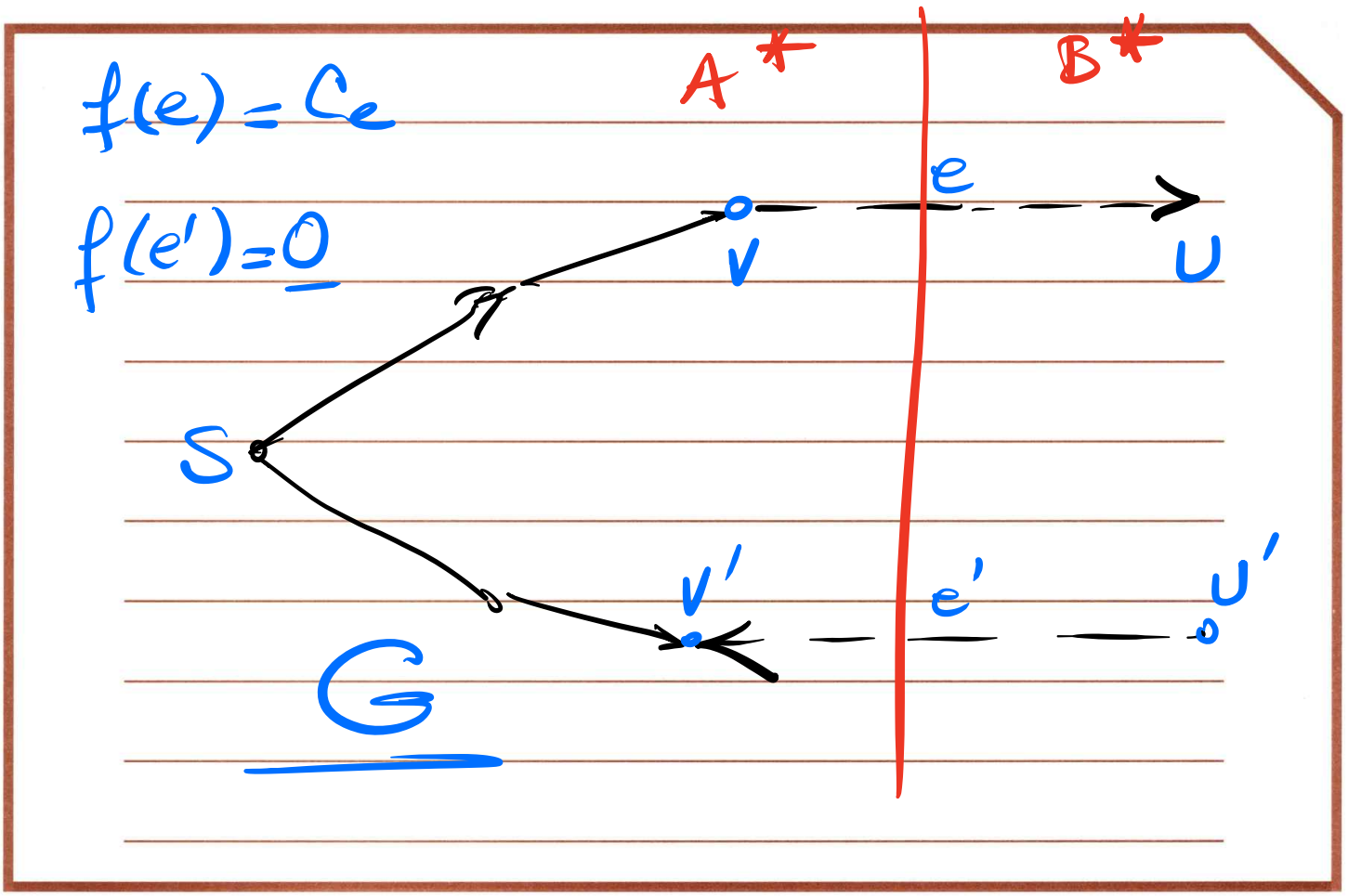
Ford-Fulkerson terminates when the flow f has no s-t paths in G_f.

Claim: If there is no s-t path in G_f, then there is an s-t cut (A^*, B^*) where

$$v(f) = c(A^*, B^*)$$

Proof: Create sets A^* & B^* such that A^* includes all nodes v where there is an s-v path in G_f.

$$B^* = V - A^*$$



$$v(f) = f^{out}(A^*) - f^{in}(A^*)$$

$$= \sum_{e \text{ out of } A^*} c_e - \emptyset$$

$$v(f) = c(A^*, B^*)$$

(A^*, B^*) has the Min Cap. of any s-t cut in G.

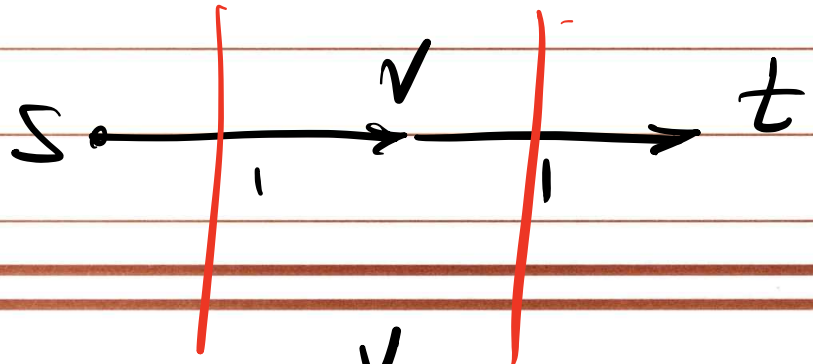
- How do we find min cut if Max Flow is given to us?

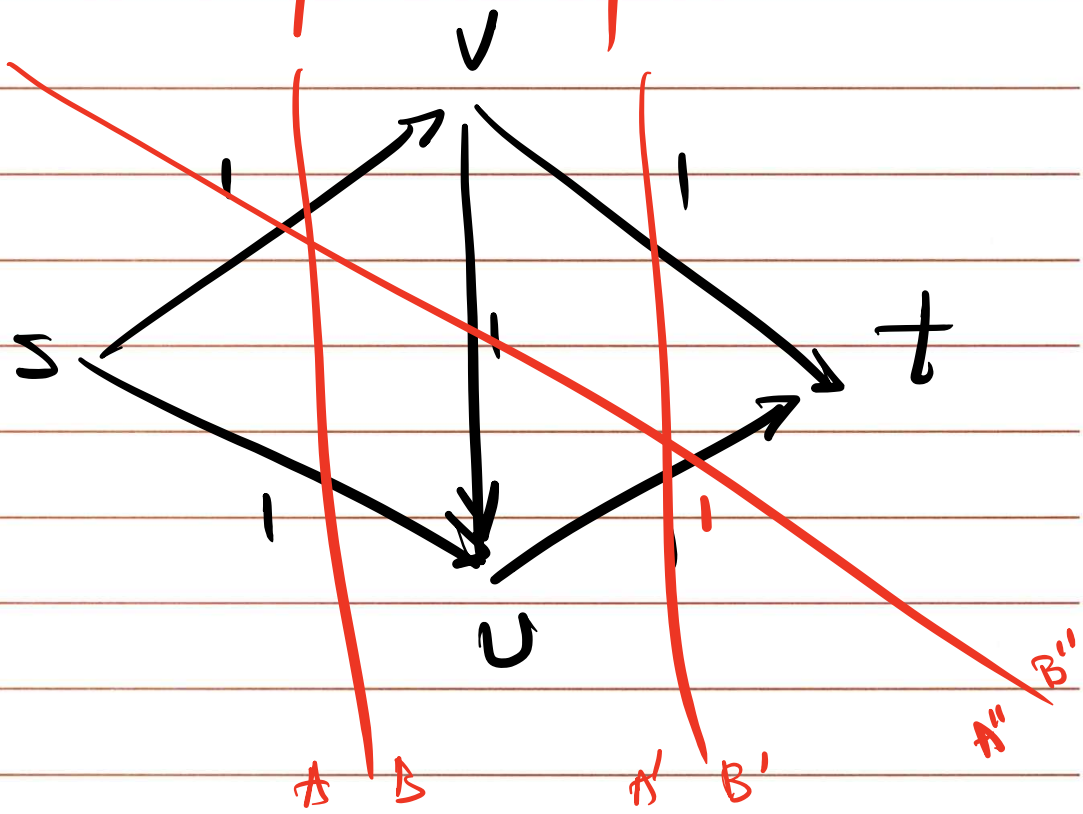
Run BFS/DFS in G_f from s

all nodes reachable from $s \longrightarrow A^*$

other nodes $\longrightarrow B^*$

(A^*, B^*) is our min cut.





How to find out if Min cut is unique?

Find Min cut closest to S
Reverse edges & find Min cut closest to t. If they are the same cut → unique min cut.

Ford-Fulkerson algorithm for Max Flow.

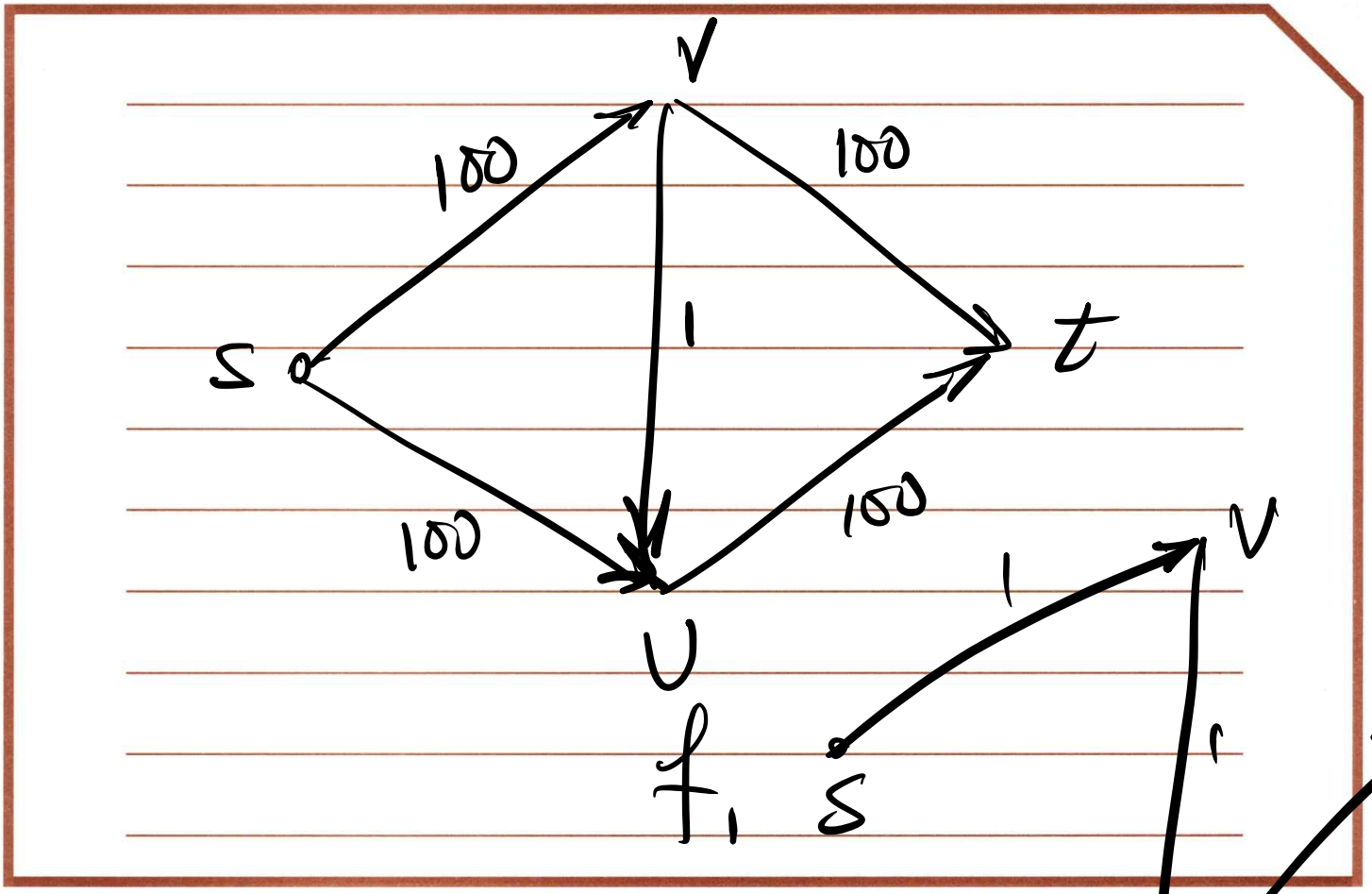
```
MaxFlow(G, s, t, C)
    Initially f(e) = 0 for all e in G
    While there is an s-t path in Gf
        let P be a simple s-t path in Gf
        f' = augment(f, c, P)
        f = f'
        update Gf
    endwhile
```

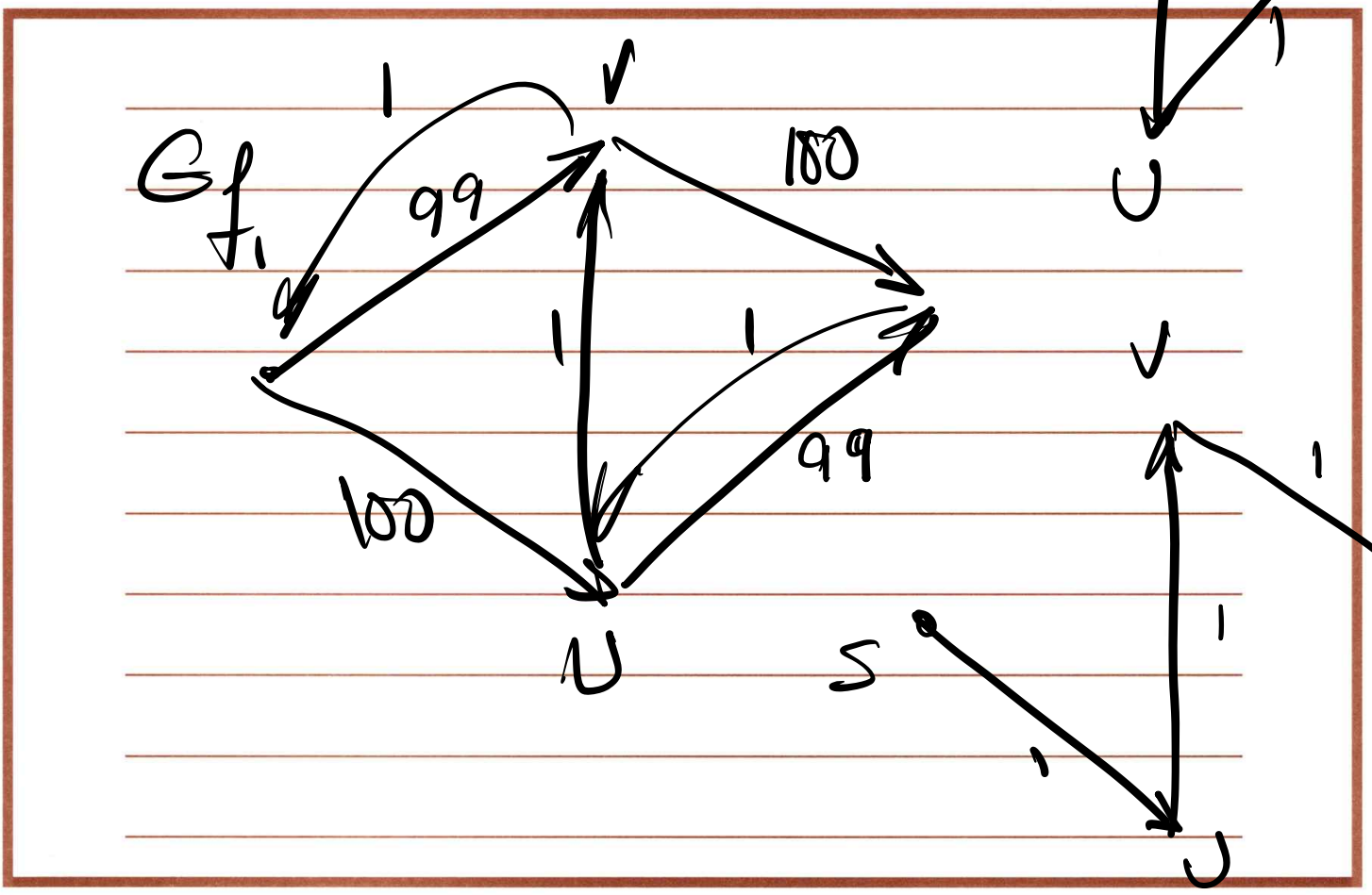
C(O(m) + O(m) + O(m))

Return f.

pseudo polynomial complexity

takes O(Cm)


v
100
100
s
1
t
100
100
u
f1
s
1
v
t
1
1
u


Gf1
1
v
99
100
1
1
99
100
u
v
1
1
s
1
t
1
u

Scaled version of Ford-Fulkerson

Initially $f(e) = 0$ for all e in G

Set Δ to be the largest power of 2 that is no larger than the Max. cap. out of s.

While $\Delta \geq 1$

  While there is an s-t path in $G_f(\Delta)$

    $O(m)$ Let P be a simple s-t path in $G_f(\Delta)$

    $O(m)$ $f' = \text{augment}(f, P)$

    $f = f'$

    update $G_f(\Delta)$

  endwhile

  $\Delta = \Delta / 2$

endwhile

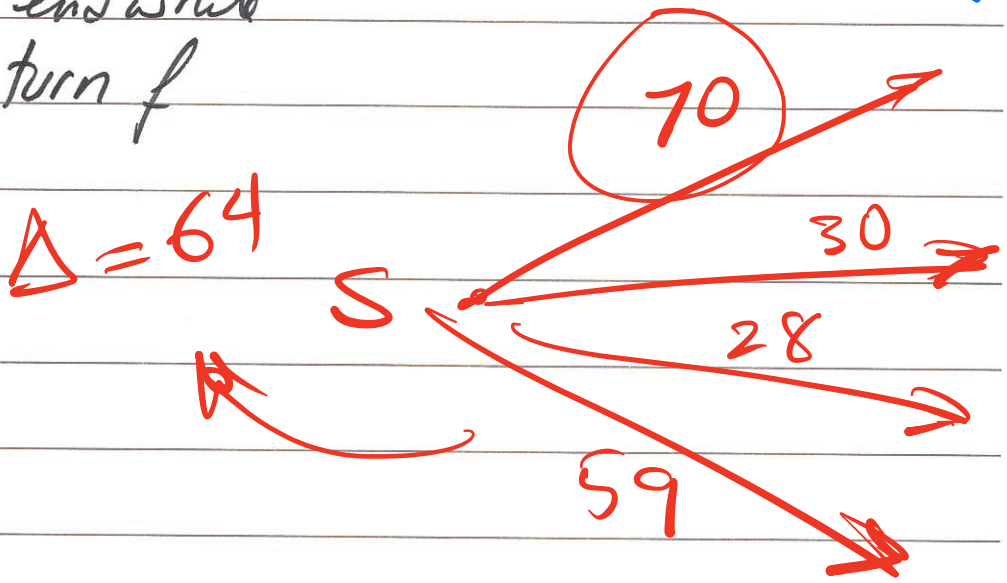
Return f

$O(mC)$ or $O(m^2)$

end of a Δ scaling phase

Δ scaling phase

$\Delta = 64$

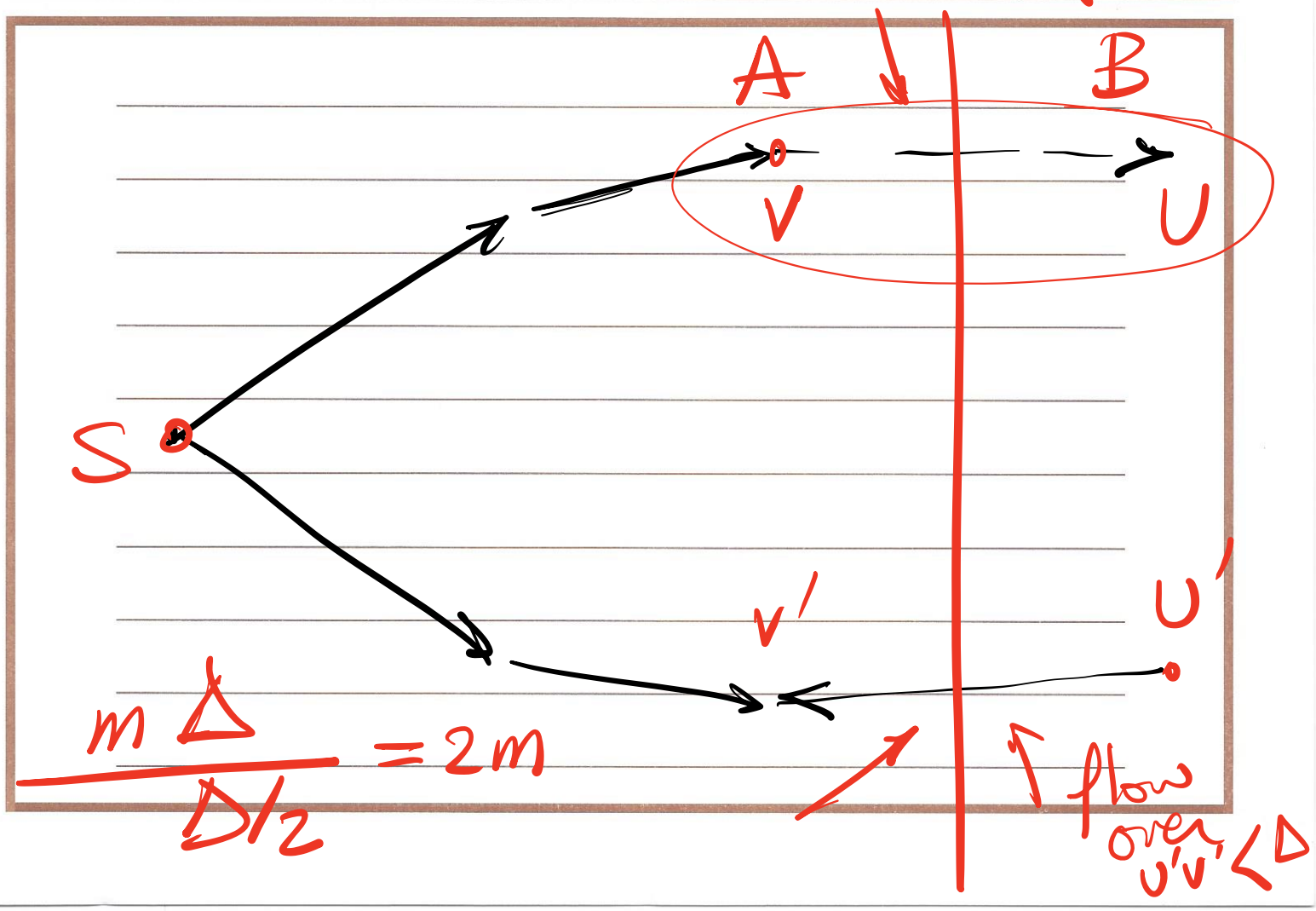


Background:

- During the Δ-scaling phase, each augmentations increases the flow value by at least Δ.

- Let f be the flow at the end of the Δ-scaling phase.

Claim: There is an s-t cut (A, B) in G for which $c(A,B) \leq v(f) + m\Delta$



Claim: The number of augmentations in a scaling phase is at most $2m$.

First scaling phase

How many times can we use each edge going out of s?

Other scaling phases

At the end of the Δ-scaling phase, we have already shown that $c(A,B) \leq v(f) + m\Delta$

If f^* is Max flow, Then $v(f^*) \leq c(A,B)$

$\Rightarrow v(f^*) \leq v(f) + m\Delta$

So, in the next scaling phase, where $\Delta' = \Delta/2$ how many iterations can we have?

Scaled version of Ford-Fulkerson

Initially $f(e) = 0$ for all e in G

Set Δ to be the largest power of 2 that is no larger than the Max. cap. out of s.

While $\Delta \geq 1$

  While there is an s-t path in $G_f(\Delta)$ — $O(m)$

    Let P be a simple s-t path in $G_f(\Delta)$ — $O(m)$

    $f' = \text{augment}(f, P)$

    $f = f'$

    update $G_f(\Delta)$

  endwhile

  $\Delta = \Delta / 2$ — $O(\log_2 C)$

endwhile

Return f

overall complexity = $O(m^2 \log_2 C)$

weakly polynomial Time Sol.

Strongly versus weakly polynomial
(relevant if input consists of integers)

An algorithm runs in strongly polynomial time if the no. of operations is bounded by a polynomial in the number of integers in the input.

An algorithm runs in weakly polynomial time if the no. of operations is bounded by a polynomial in the number of bits in the input, but not in the number of integers in the input.

Edmonds-Karp

Same as Ford-Fulkerson, except that each augmenting path must be a shortest path with available capacity.

Can be shown to have running time $O(nm^2)$

Ford-Fulkerson	$O(Cm)$ pseudo-polynomial
Scaled version of FF	$O(m^2 \lg C)$ weakly polynomial
Edmonds-Karp	$O(nm^2)$ strongly polynomial
Orlin + KTR	$O(nm)$ " "

Recently developed methods solve max flow in close to linear time WRT m.

approximation

Discussion 8

1. You have successfully computed a maximum s-t flow f for a network G = (V; E) with integer edge capacities. Your boss now gives you another network G' that is identical to G except that the capacity of exactly one edge is decreased by one. You are also explicitly given the edge whose capacity was changed. Describe how you can compute a maximum flow for G' in O(|V| + |E|) time.

2. You need to transport iron-ore from the mine to the factory. We would like to determine how long it takes to transport. For this problem, you are given a graph representing the road network of cities, with a list of k of its vertices ($t_1, t_2, \ldots, t_k$) which are designated as factories, and one vertex S (the iron-ore mine) where all the ore is present.
We are also given the following:

- Road Capacities (amount of iron that can be transported per minute) for each road (edges) between the cities (vertices).
- Factory Capacities (amount of iron that can be received per minute) for each factory (at $t_1, t_2, \ldots, t_k$)
- The amount of ore to be transported from the mine, C

Give a polynomial-time algorithm to determine the minimum amount of time necessary to transport and receive all the iron-ore at factories.

3. In a daring burglary, someone attempted to steal all the candy bars from the CS department. Luckily, he was quickly detected, and now, the course staff and students will have to keep him from escaping from campus. In order to do so, they can be deployed to monitor strategic routes.

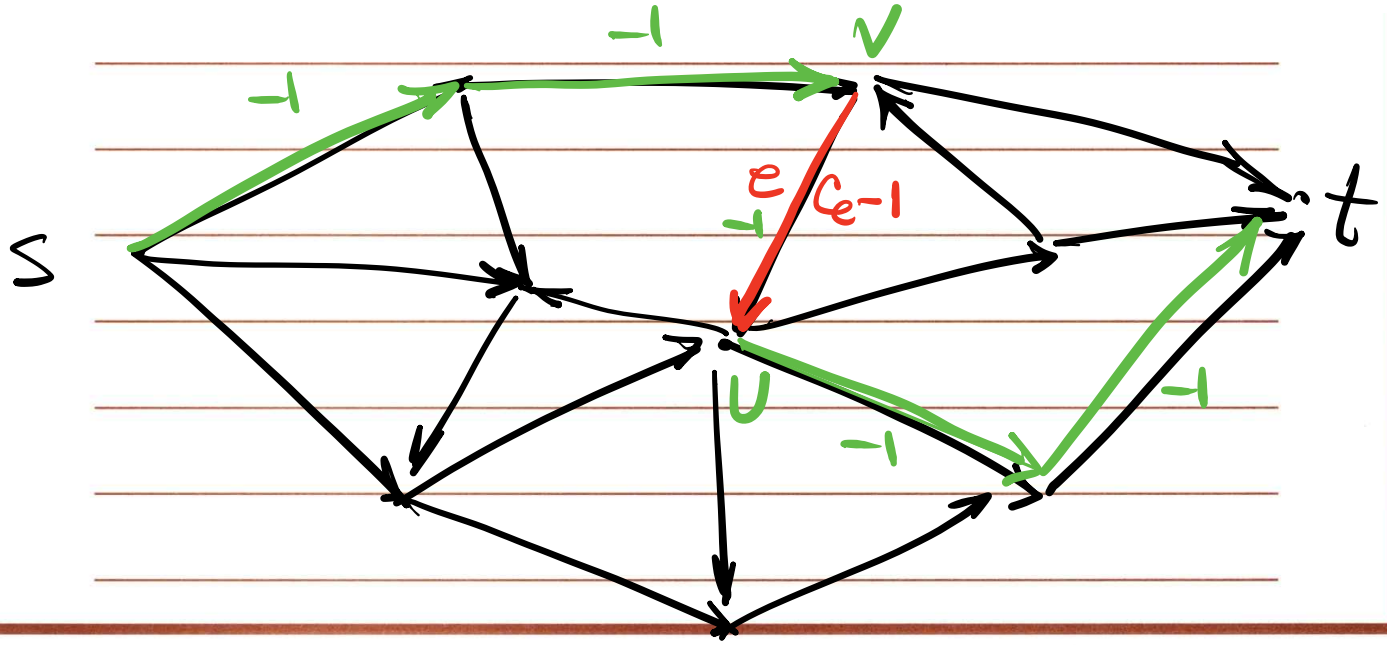
More formally, we can think of the USC campus as a graph, in which the nodes are locations, and edges are pathways or corridors. One of the nodes (the instructor's office) is the burglar's starting point, and several nodes (the USC gates) are the escape points — if the burglar reaches any one of those, the candy bars will be gone forever. Students and staff can be placed to monitor the edges. As it is hard to hide that many candy bars, the burglar cannot pass by a monitored edge undetected.

Give an algorithm to compute the minimum number of students/staff needed to ensure that the burglar cannot reach any escape points undetected (you don't need to output the corresponding assignment for students — the number is enough). As input, the algorithm takes the graph G = (V,E) representing the USC campus, the starting point s, and a set of escape points $P \subseteq V$. Prove that your algorithm is correct and runs in polynomial time.

4. We define a most vital edge of a network as an edge whose deletion causes the largest decrease in the maximum s-t-flow value. Let f be an arbitrary maximum s-t-flow. Either prove the following claims or show through counterexamples that they are false:

(a) A most vital edge is an edge e with the maximum value of c(e).
(b) A most vital edge is an edge e with the maximum value of f (e).
(c) A most vital edge is an edge e with the maximum value of f (e) among edges belonging to some minimum cut.
(d) An edge that does not belong to any minimum cut cannot be a most vital edge.
(e) A network can contain only one most vital edge.

1. You have successfully computed a maximum s-t flow f for a network G = (V; E) with integer edge capacities. Your boss now gives you another network G' that is identical to G except that the capacity of exactly one edge is decreased by one. You are also explicitly given the edge whose capacity was changed. Describe how you can compute a maximum flow for G' in O(|V| + |E|) time.



Case 1 → $f(e) < c_e$

→ Max Flow stays as it was.

Case 2 → $f(e) = c_e$

O(m) Find a path from s to v on edges carrying flow

O(m) Find . . . . u to t

O(m) Subtract 1 unit of flow on path

$s \to v \to u \to t$

$\Rightarrow$ flow f'

$O(m)$ Construct $G_{f'}$

$O(m)$ find a path from s to t in $G_{f'}$

if no path $\rightarrow$ f' is max flow

$O(m)$ else augment one more unit of flow to f'

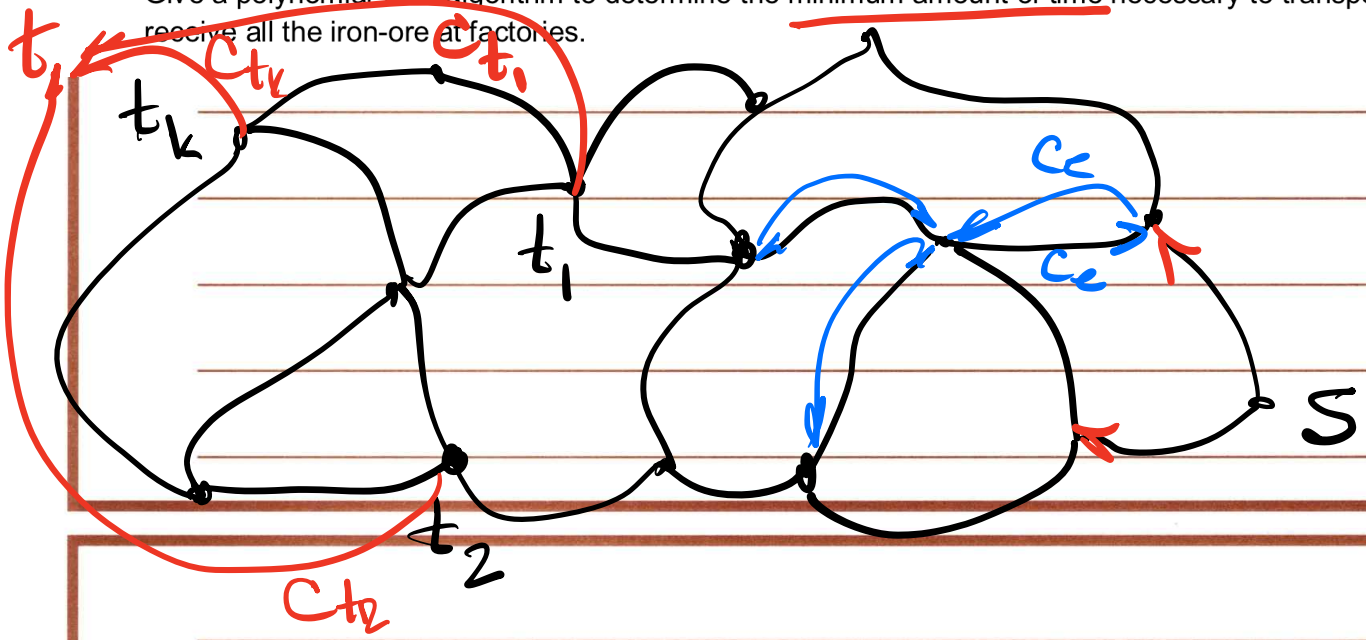
$\Rightarrow f''$

2. You need to transport iron-ore from the mine to the factory. We would like to determine how long it takes to transport. For this problem, you are given a graph representing the road network of cities, with a list of k of its vertices ($t_1, t_2, \ldots, t_k$) which are designated as factories, and one vertex S (the iron-ore mine) where all the ore is present.

We are also given the following:

- Road Capacities (amount of iron that can be transported per minute) for each road (edges) between the cities (vertices).
- Factory Capacities (amount of iron that can be received per minute) for each factory (at $t_1, t_2, \ldots, t_k$)
- The amount of ore to be transported from the mine, C → tons

Give a polynomial-time algorithm to determine the minimum amount of time necessary to transport and receive all the iron-ore at factories.



Run Max Flow

$\xrightarrow{F}$ $v(F) = X$ ton/min

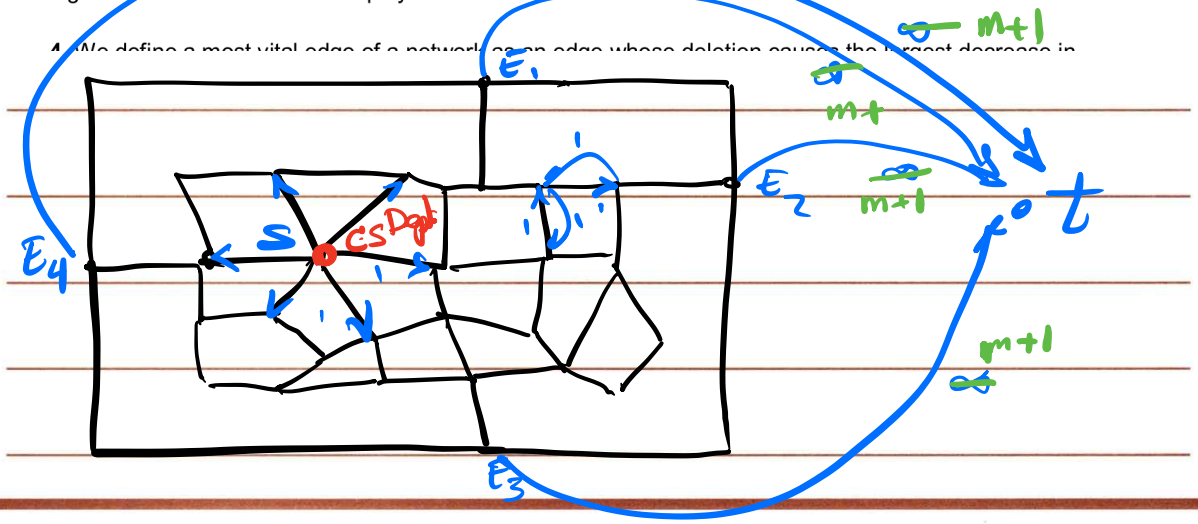
$t = \frac{C}{X}$ mins

3. In a daring burglary, someone attempted to steal all the candy bars from the CS department. Luckily, he was quickly detected, and now, the course staff and students will have to keep him from escaping from campus. In order to do so, they can be deployed to monitor strategic routes.

More formally, we can think of the USC campus as a graph, in which the nodes are locations, and edges are pathways or corridors. One of the nodes (the instructor's office) is the burglar's starting point, and several nodes (the USC gates) are the escape points — if the burglar reaches any one of those, the candy bars will be gone forever. Students and staff can be placed to monitor the edges. As it is hard to hide that many candy bars, the burglar cannot pass by a monitored edge undetected.

Give an algorithm to compute the minimum number of students/staff needed to ensure that the burglar cannot reach any escape points undetected (you don't need to output the corresponding assignment for students — the number is enough). As input, the algorithm takes the graph G = (V,E) representing the USC campus, the starting point s, and a set of escape points P ⊆ V . Prove that your algorithm is correct and runs in polynomial time.

4. We define a most vital edge of a network as an edge whose deletion causes the largest decrease in



Ford-Fulkerson → $O(Cm)$

$O(nm)$ ←

4. We define a most vital edge of a network as an edge whose deletion causes the largest decrease in the maximum s-t-flow value. Let f be an arbitrary maximum s-t-flow. Either prove the following claims or show through counterexamples that they are false:

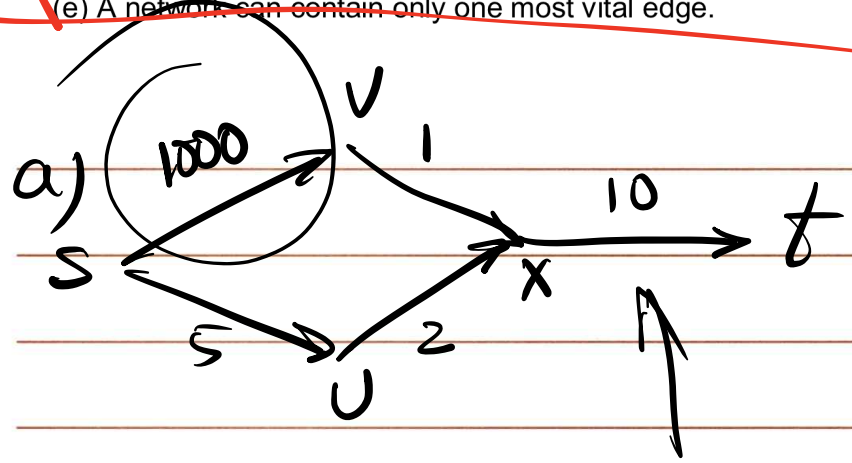
(a) A most vital edge is an edge e with the maximum value of c(e).

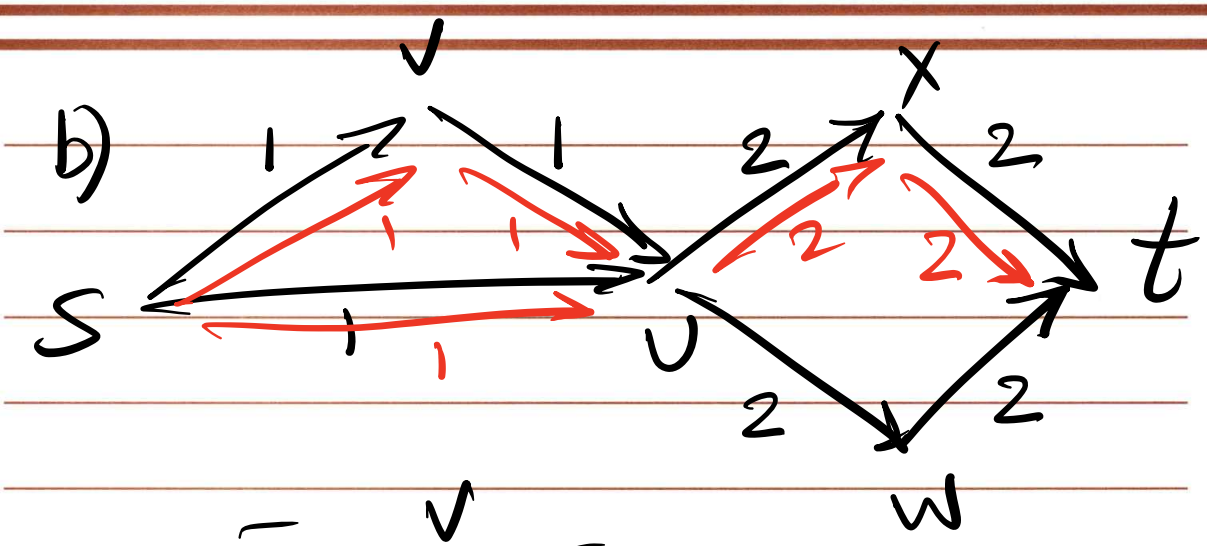
(b) A most vital edge is an edge e with the maximum value of f (e).

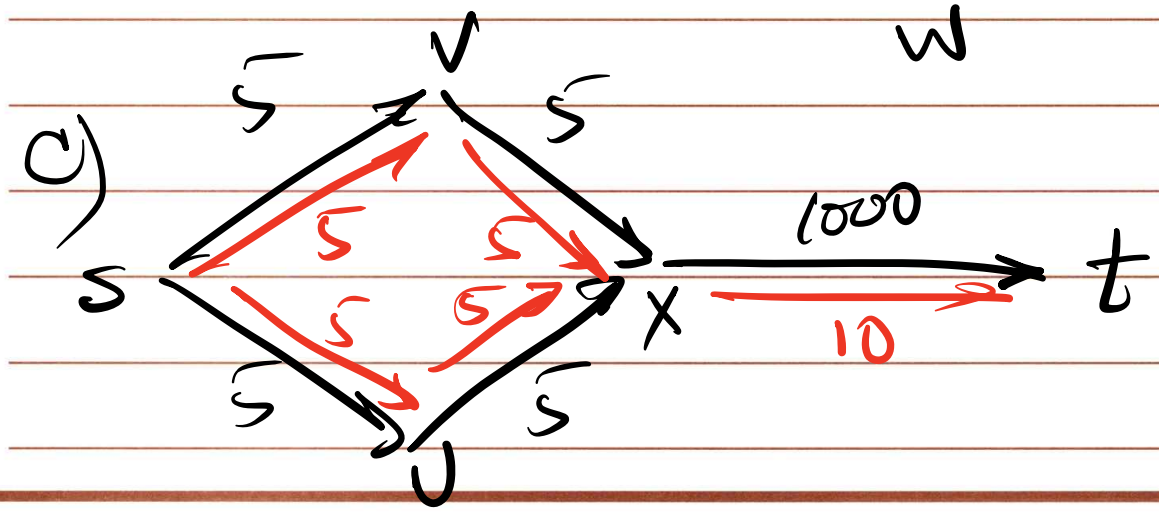
(c) A most vital edge is an edge e with the maximum value of f (e) among edges belonging to some minimum cut.

(d) An edge that does not belong to any minimum cut cannot be a most vital edge.

(e) A network can contain only one most vital edge.





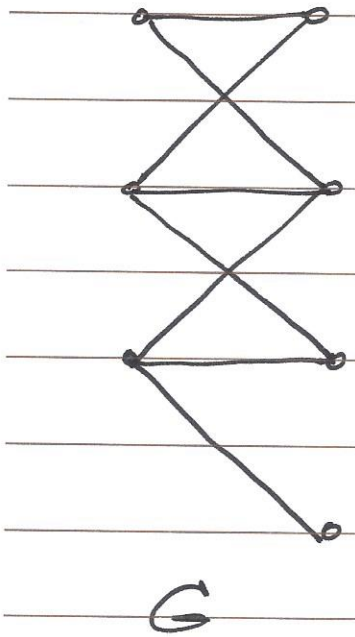


Network Flow

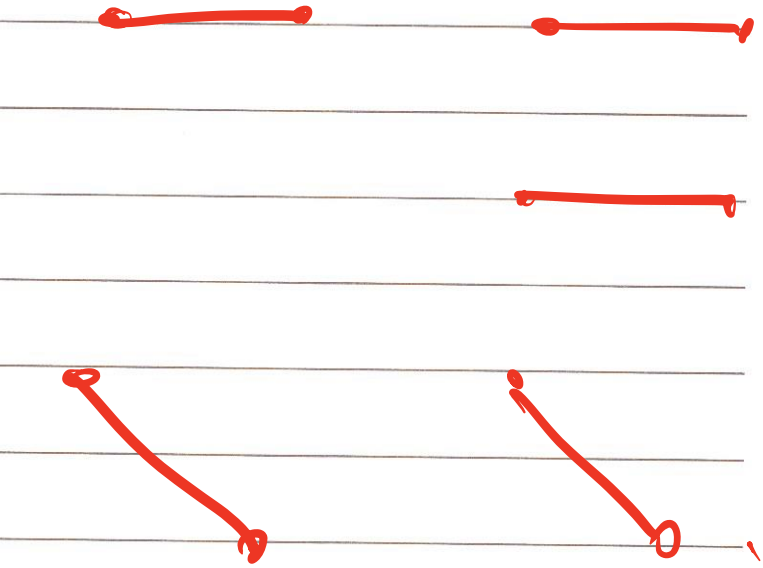
Bipartite Matching Problem

Def A bipartite graph $G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$ with property that every edge $e \in E$ has one end in X and the other in Y.

Def A matching M in G is a subset of the edges $M \subseteq E$ such that each node appears in at most one edge in M.



G



A matching

A max size matching

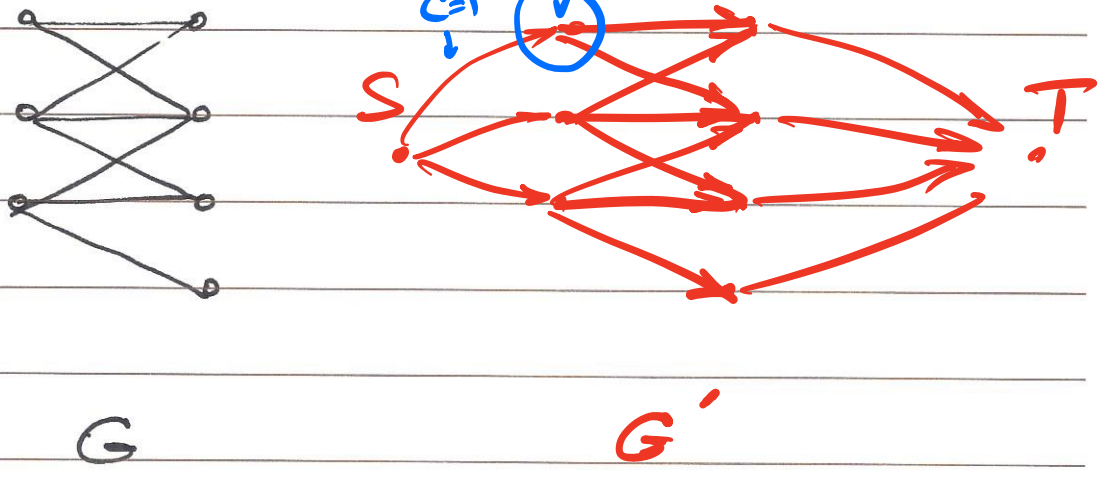
Problem Statement:

Find a matching M of largest possible size in G.

General Plan:

Design a flow network G' that will have a flow value $v(f) = k$ iff there is a ~~max.~~ ~~Size~~ matching of size k in G. Moreover, flow f in G' should identify the matching M in G.

Construction of G'



G G'

Solution

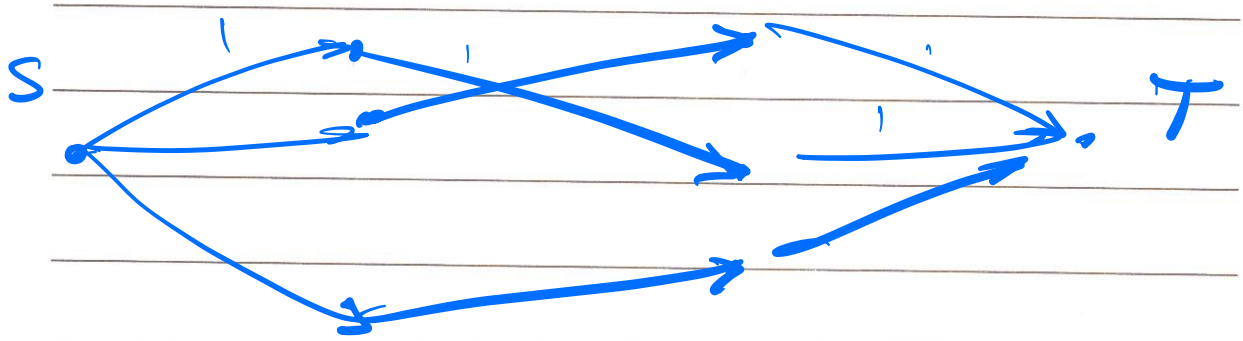
Run Max Flow on G'. Say max. flow is f.

Edges carrying flow between sets X & Y will correspond to our max. size matching in G.

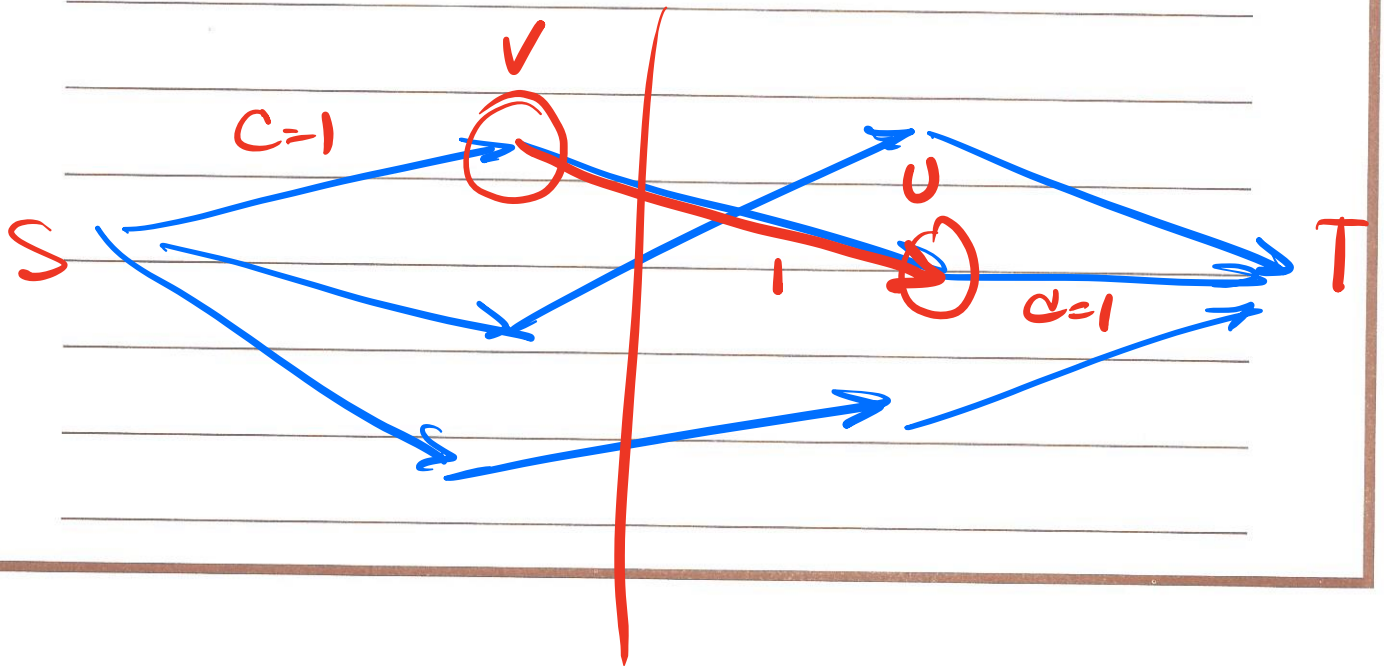
To prove this, we will show that G' will have a max. flow of value k iff G has a max. size matching of size k.

Proof:

A- If we have a matching of size k in G, we can find an s-t flow f of value k in G'.



B- If we have an s-t flow of value k in G', we can find a matching of size k in G.



If we used Ford-Fulkerson

Complexity $\longrightarrow O(Cm)$

$\downarrow$

$O(nm)$

Network Flow

Edge-Disjoint Paths

Def. A set of paths is edge-disjoint if their edge sets are disjoint

Problem Statement

Given a directed graph G with $s \& t \in V$, find max. number of edge-disjoint s-t paths in G.

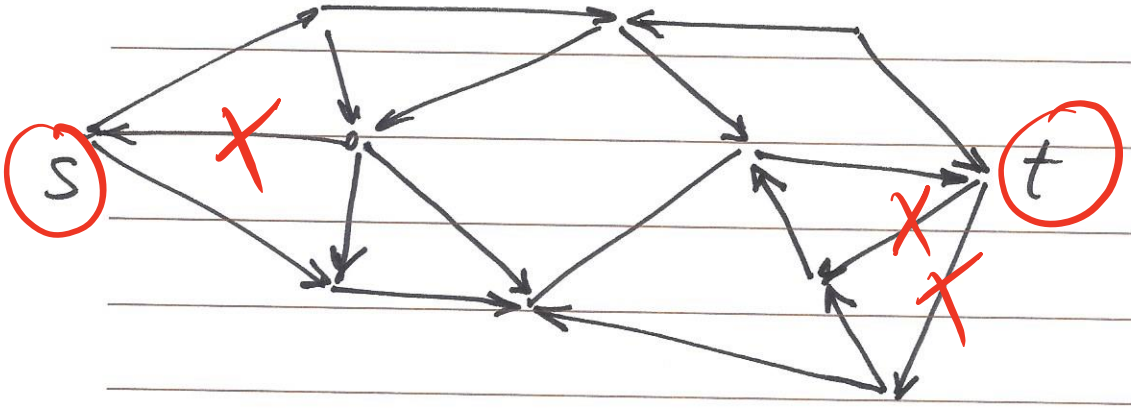
General Plan:

Design a flow network G' that will have a flow value $v(f) = k$ iff there are k edge-disjoint s-t paths in G.

Moreover, flow f in G' should identify the set of edge-disjoint paths in G.

Construction of G'

c = 1 for all edges



G

Solution:

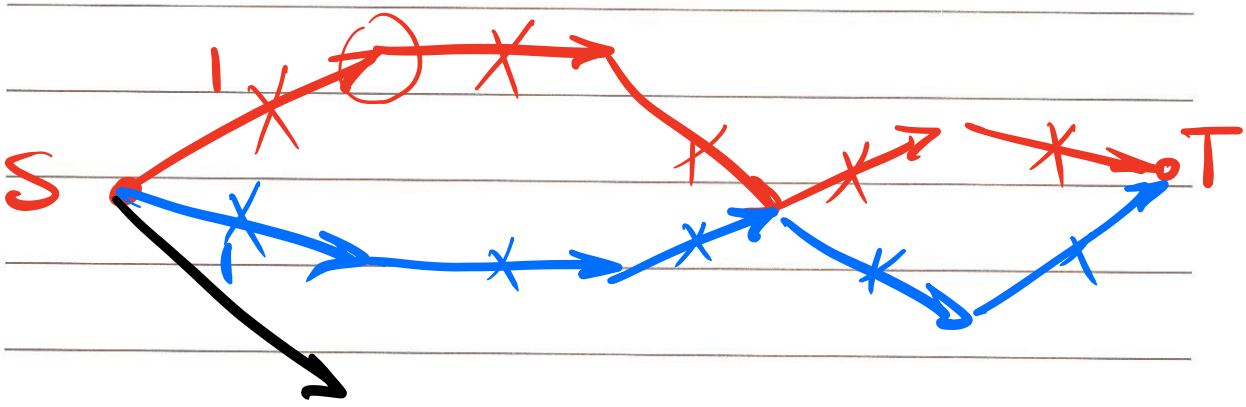
- Run Max flow in G'
- v(f) will equal the max. number of edge-disjoint s-t paths
- f will identify edges on these paths

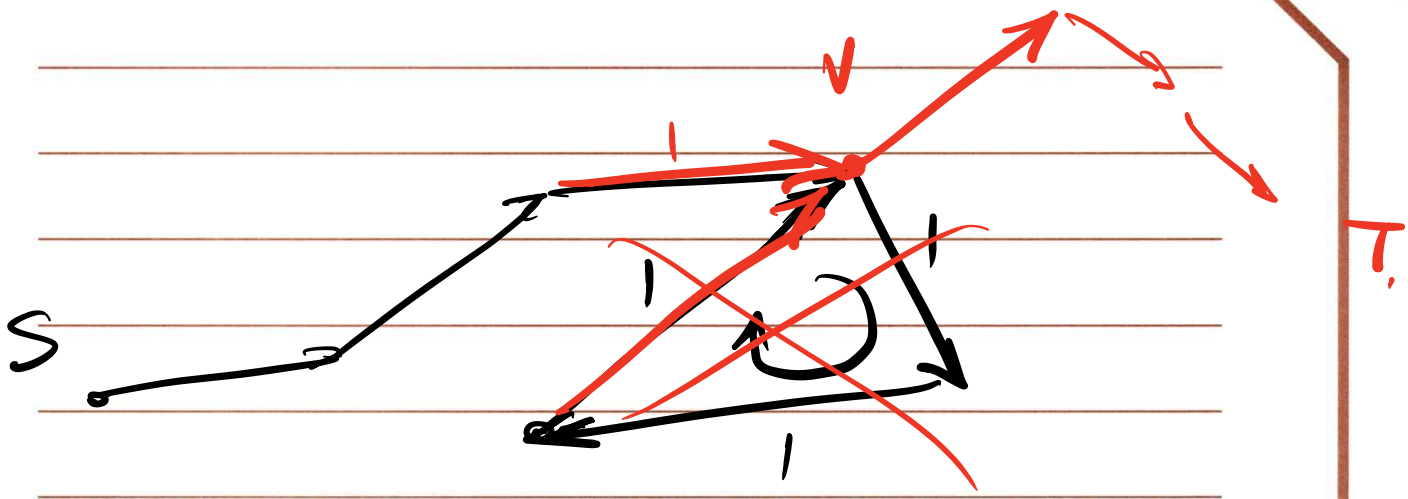
To prove this, we will show that there are k edge disjoint paths in G iff there is a flow of value k in G'.

Proof:

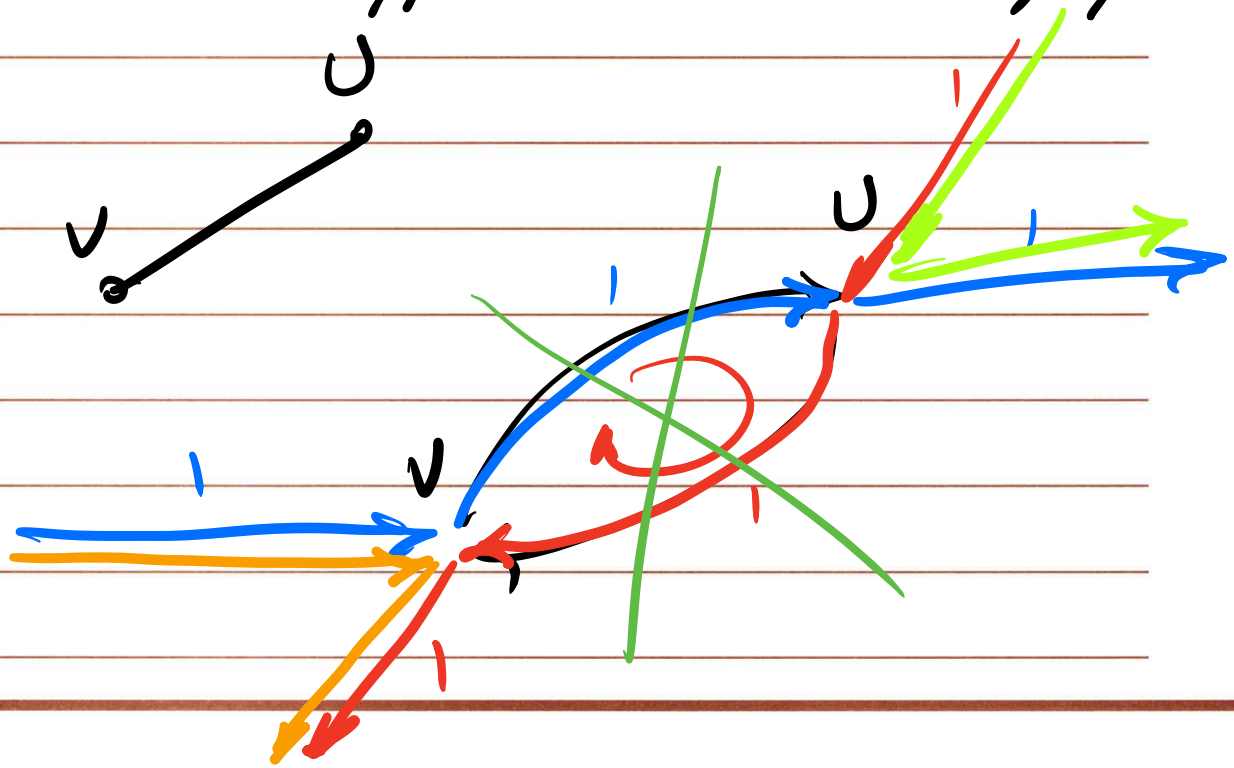
A) If we have k edge disjoint s-t paths in G, we can find a flow of value k in G'.

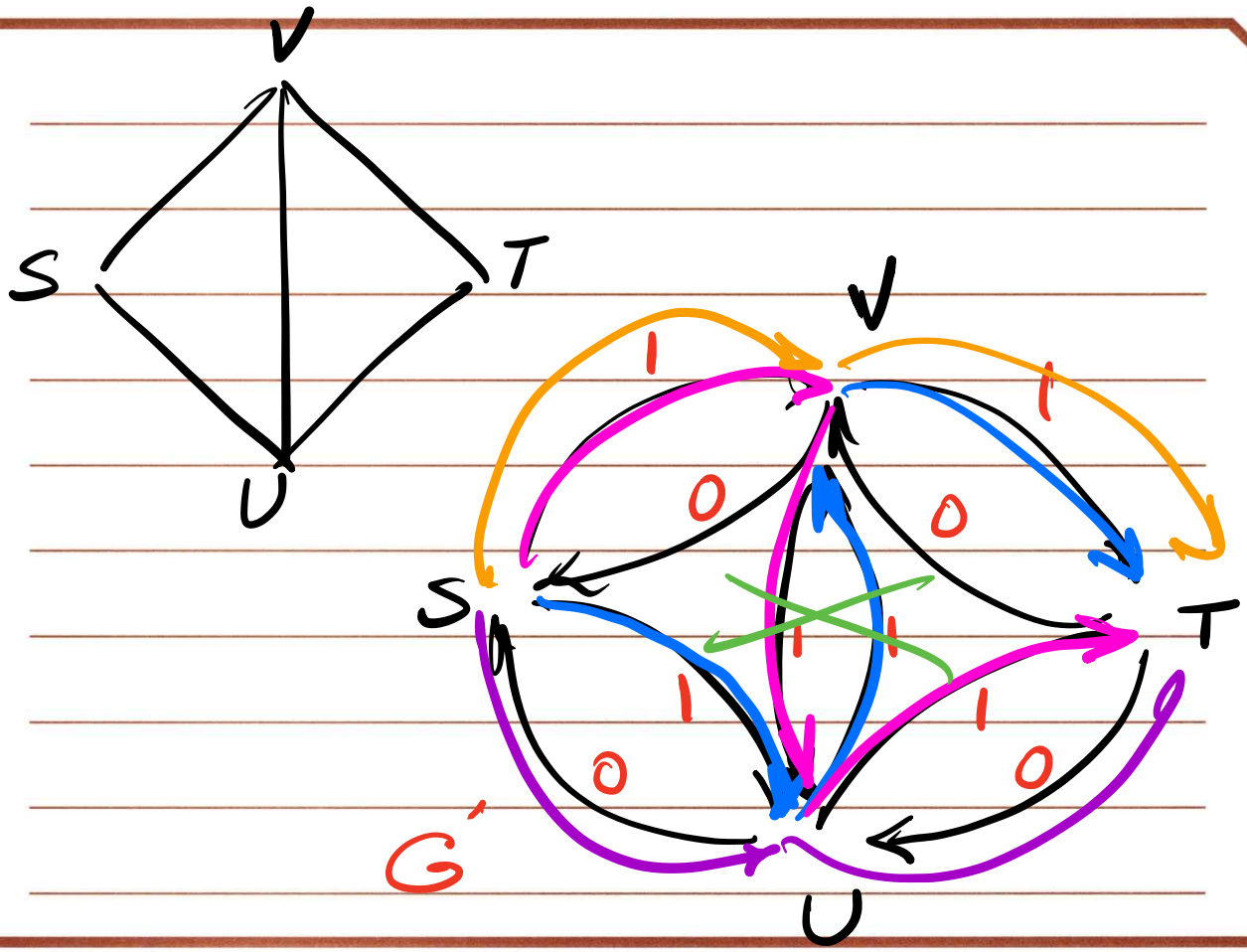
B) If we have a flow of value k in G', we can find k edge-disjoint s-t paths in G.





How do we modify this solution
So it applies to undirected graphs!




V
S
T
U
V
1
1
0
0
S
T
1
1
1
1
0
0
G´
U

Network Flow

Node-disjoint Paths

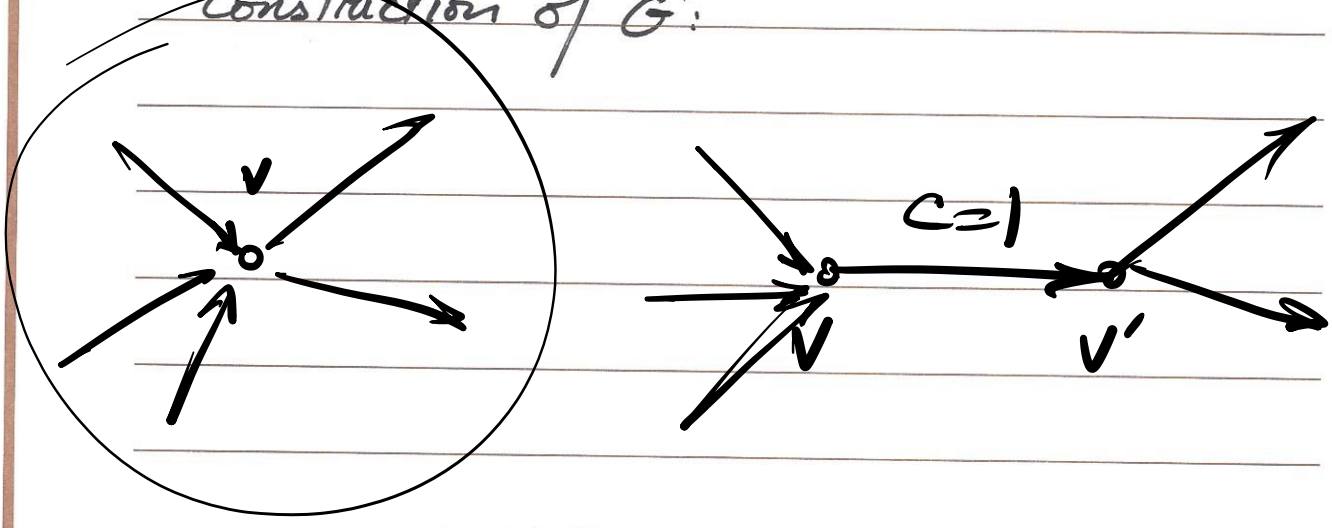
Def. A set of paths is node-disjoint if their node sets (except for starting & ending nodes) are disjoint

Problem Statement

Given a directected graph G with $s \& t \in V$, find the max. number of node-disjoint s-t paths in G.

Plan: As usual...

Construction of G':



Circulation &

Circulation with Lower Bounds

Circulation Network

- We are given a directed graph $G=(V,E)$ with capacities on the edges.
- Associated with each node $v \in V$ is a demand d_v
  - if $d_v > 0$, node v has demand of d_v for flow (sink)
  - if $d_v < 0$, node v has a supply of $|d_v|$ for flow (source)
  - if $d_v = 0$ v is neither a sink nor a source

Def. A circulation with demand $\{d_v\}$ is a function f that assigns non-negative real numbers to each edge and satisfies:

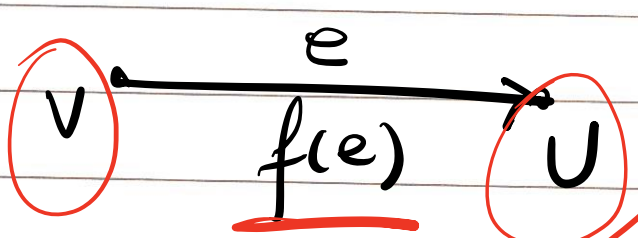
1) Capacity condition
   for each edge $e \in E$ $0 \leq f(e) \leq c_e$

2) Demand condition
   for each node $v \in V$, $f^{in}(v) - f^{out}(v) = d_v$

FACT: If there is a feasible circulation w/ demands $\{d_v\}$ then $\sum_v d_v = 0$

Proof: $f^{in}(v) - f^{out}(v) = d_v$

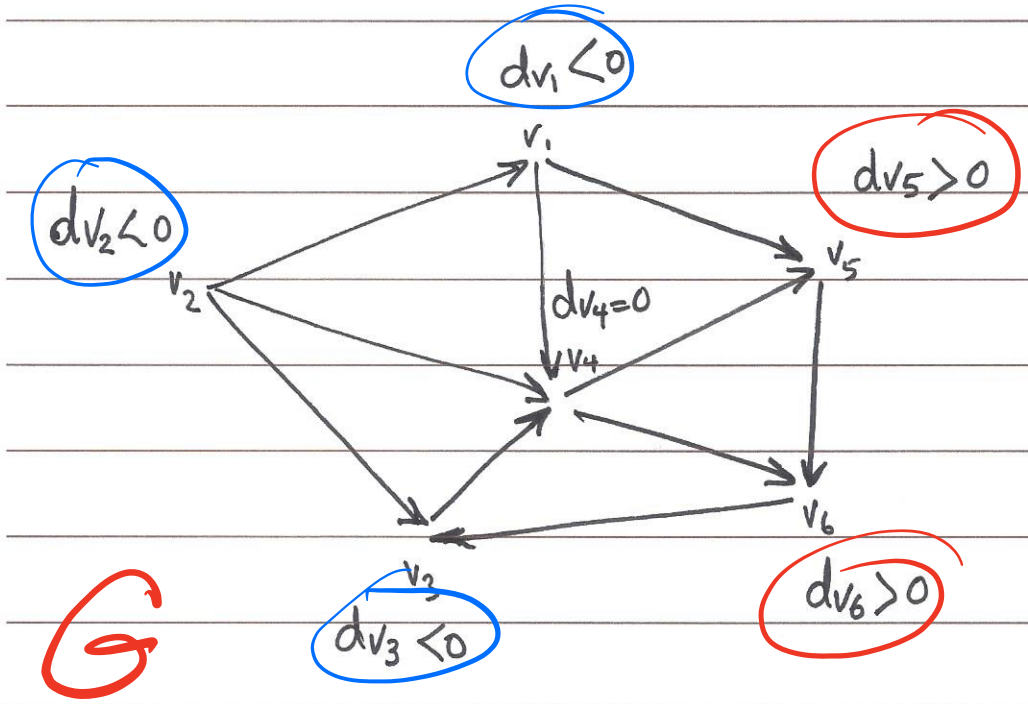
$$\sum_v d_v = \sum f^{in}(v) - f^{out}(v) = 0$$

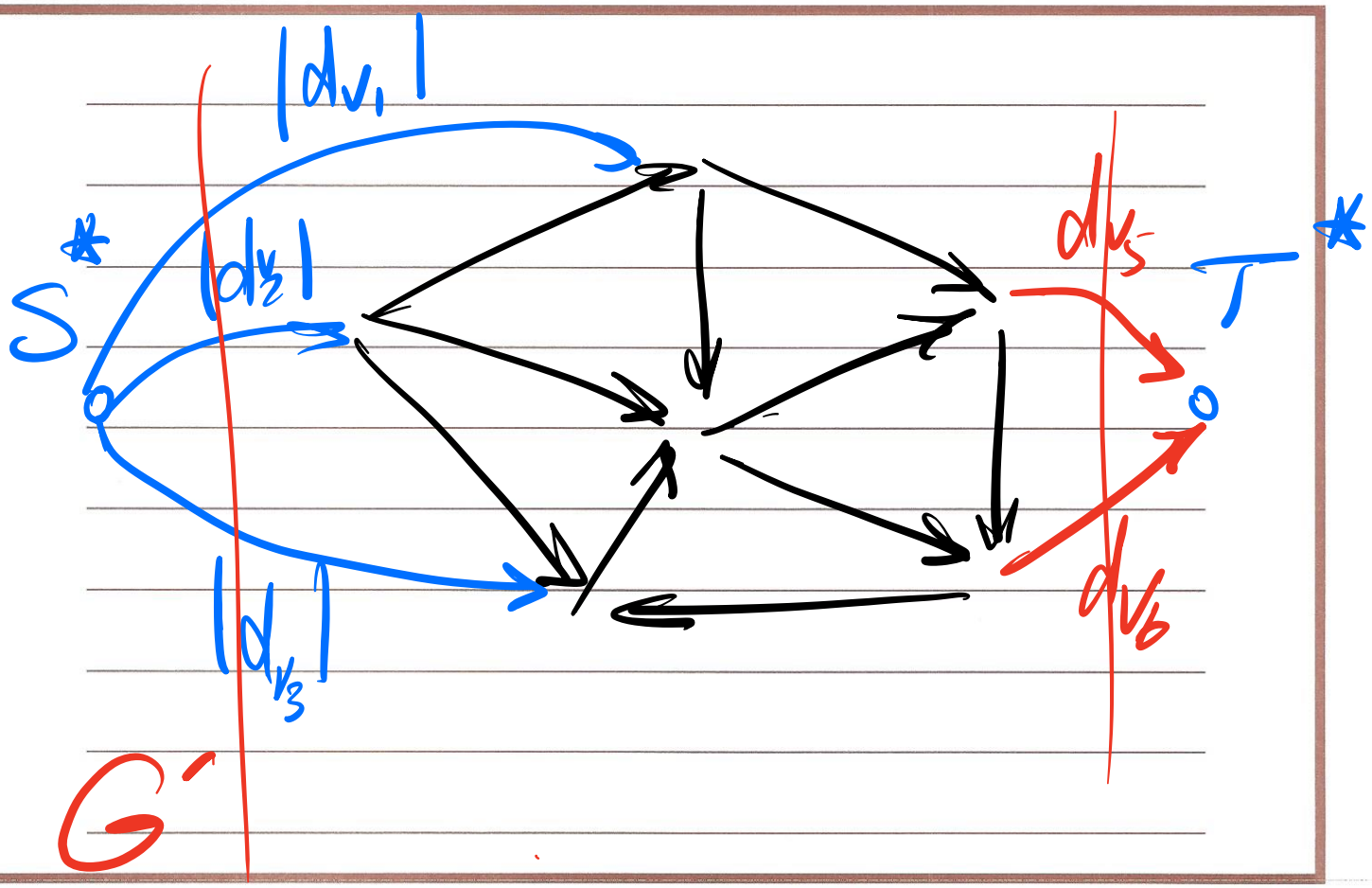


$\sum_v d_v = 0$

$$\sum_{v: d_v > 0} d_v = -\sum_{v: d_v < 0} d_v = D$$

total Demand value


$dv_1 < 0$
v_1
$dv_2 < 0$
v_2
$dv_5 > 0$
v_5
$dv_4 = 0$
v_4
v_6
v_3
$dv_3 < 0$
$dv_6 > 0$
G


$|dv_1|$
S
$|dv_2|$
$|dv_3|$
dv_5
T
dv_6
G'

Run MaxFlow on $G' \rightarrow f$

Say $v(f) = X$

$X < D$? $\rightarrow$ no feasible circ. in G

$X > D$? $\rightarrow$ Not possible

$X = D$ $\rightarrow$ we have a feasible circ. in G.

Proof:

A) If there is a feasible circulation f w/ demand values $\{d_v\}$ in G, we can find a Max Flow in G' of value D.

B) If there is a Max Flow in G' of value D, we can find a feasible circulation in G.

Circulation with Demands & Lower bounds

Conditions:

1) Capacity conditions
   for each edge $e \in E$, $l_e \leq f(e) \leq c_e$

2) Demand conditions
   for every node $v \in V$, $f^{in}(v) - f^{out}(v) = d_v$

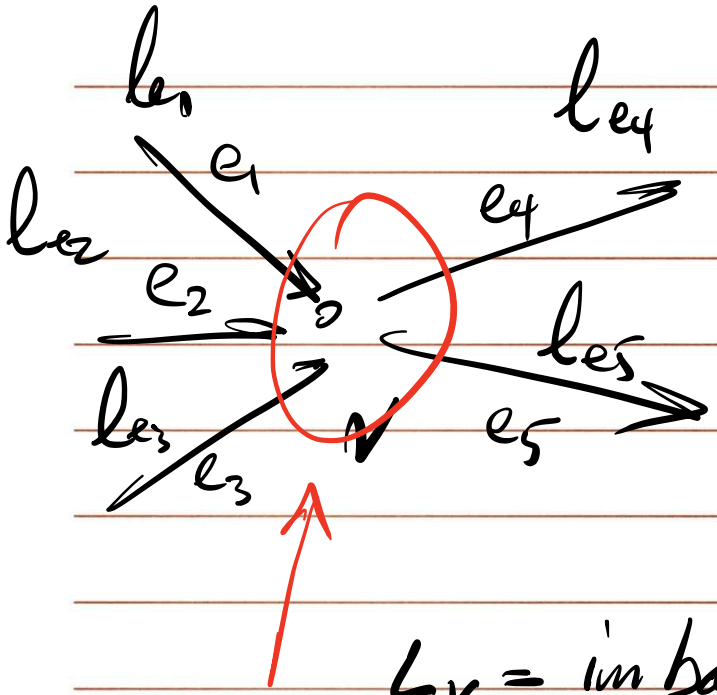
Solution:

Find feasible circulation (if it exists) in two passes.

Pass #1. find f_0 to satisfy all l_e's

Pass #2. Use remaining capacity of the network to find a feasible circulation f_1 (if it exists)

Combine the two flows: $f = f_0 + f_1$



$f_0(e_1) = l_{e_1}$

$f_0(e_2) = l_{e_2}$

L_v = imbalance at node v due to f_0

$$f_0^{in}(v) - f_0^{out}(v) = \sum_{e \text{ into } v} l_e - \sum_{e \text{ out of } v} l_e$$

$$= L_v$$

flow imbalance at node v.

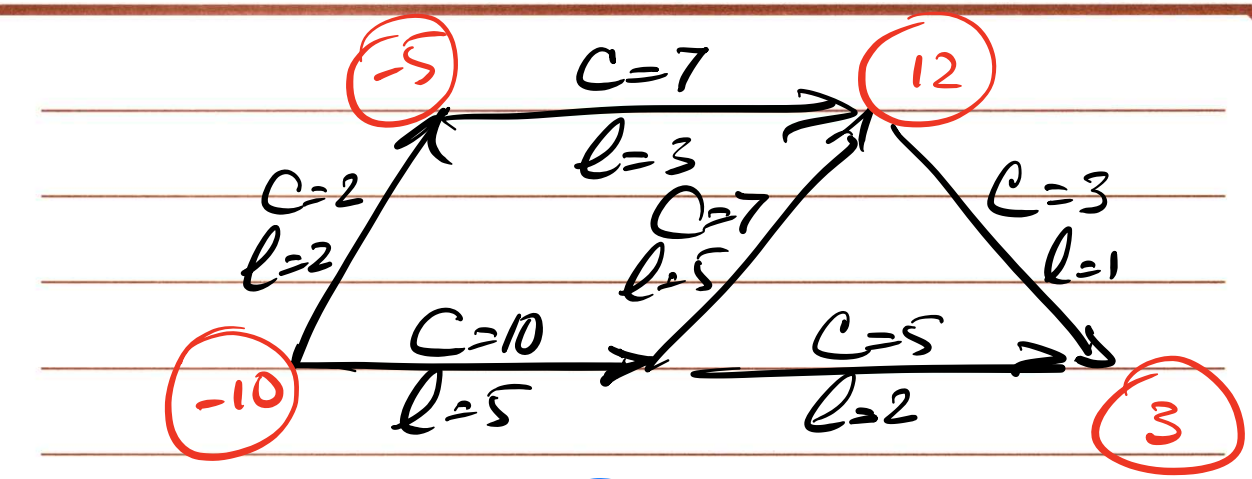
1- Push flow f thru G where $f_0(e) = l_e$

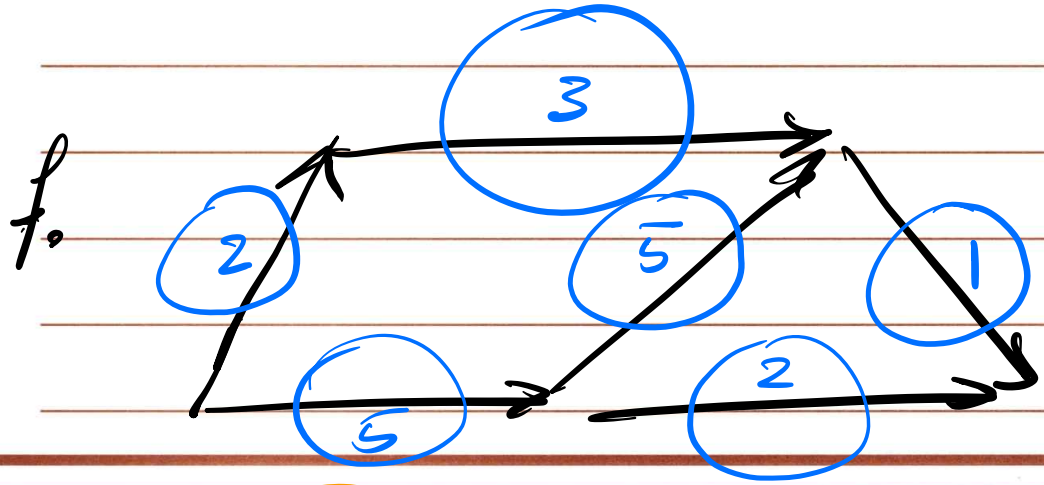
2- Construct G' where $c'_e = c_e - l_e$ & $d'_v = d_v - L_v$

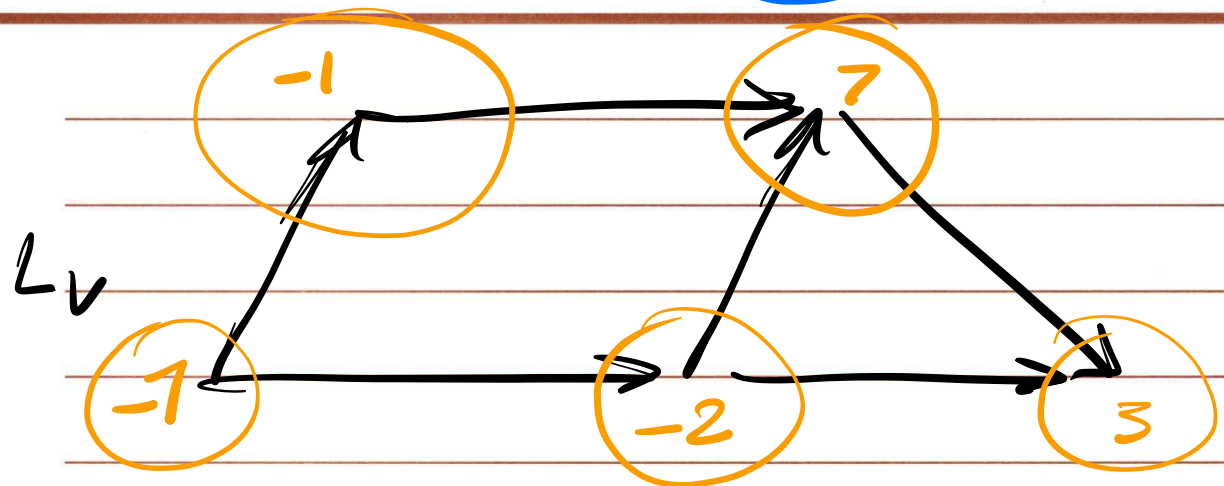
3- Find feasible circ in G' call this f_1.

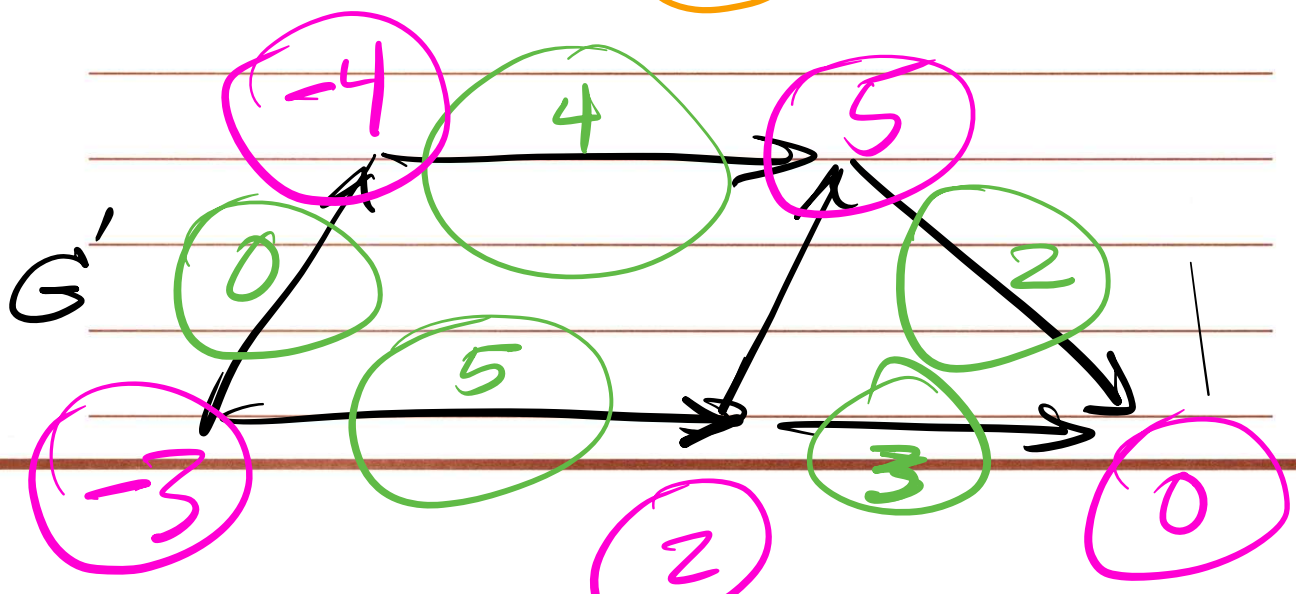
4- If there is no feasible circ in G' $\Rightarrow$ no feasible circ. in G

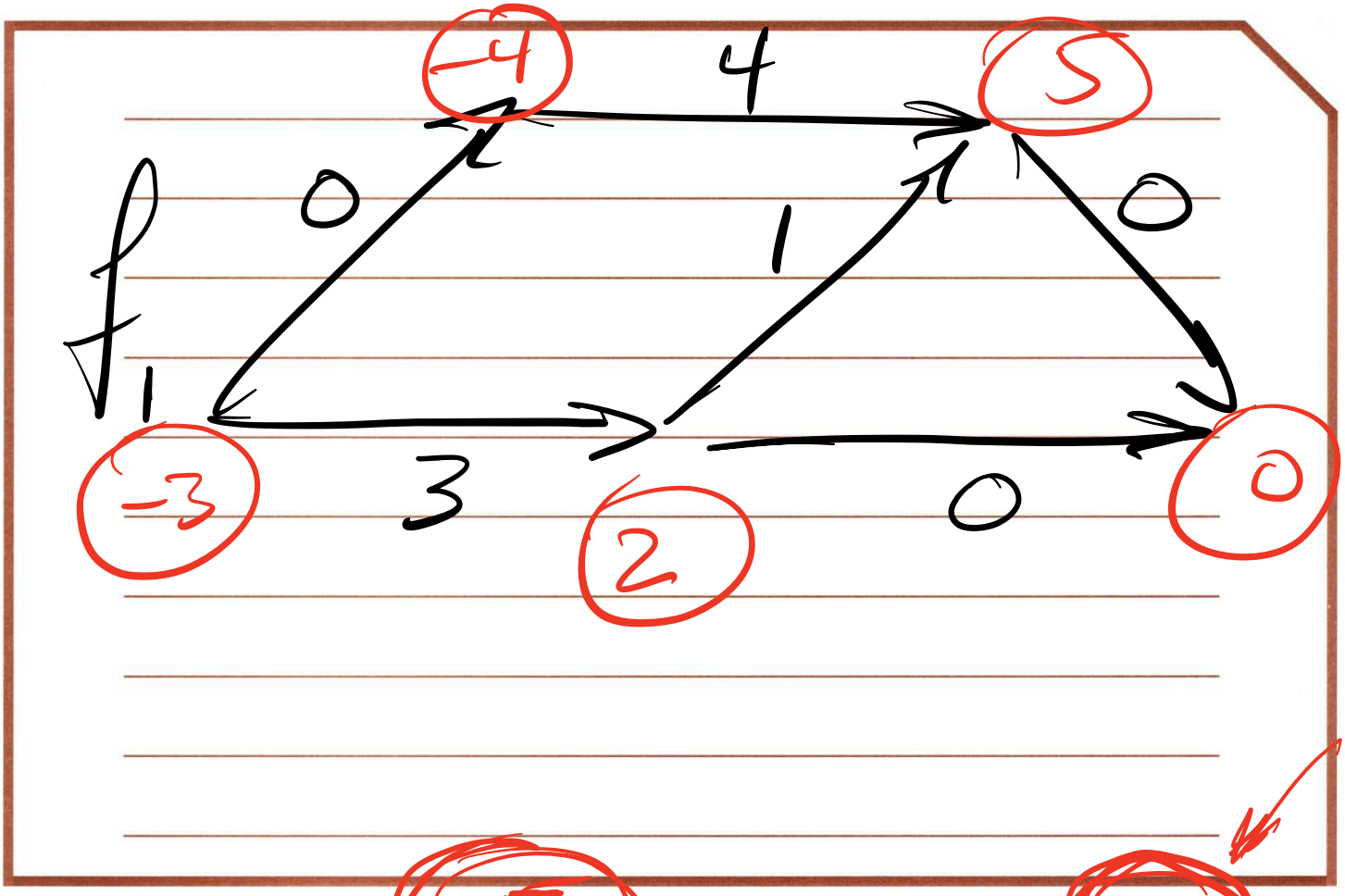
otherwise $\rightarrow$ feasible circ. in $G = f_0 + f_1$

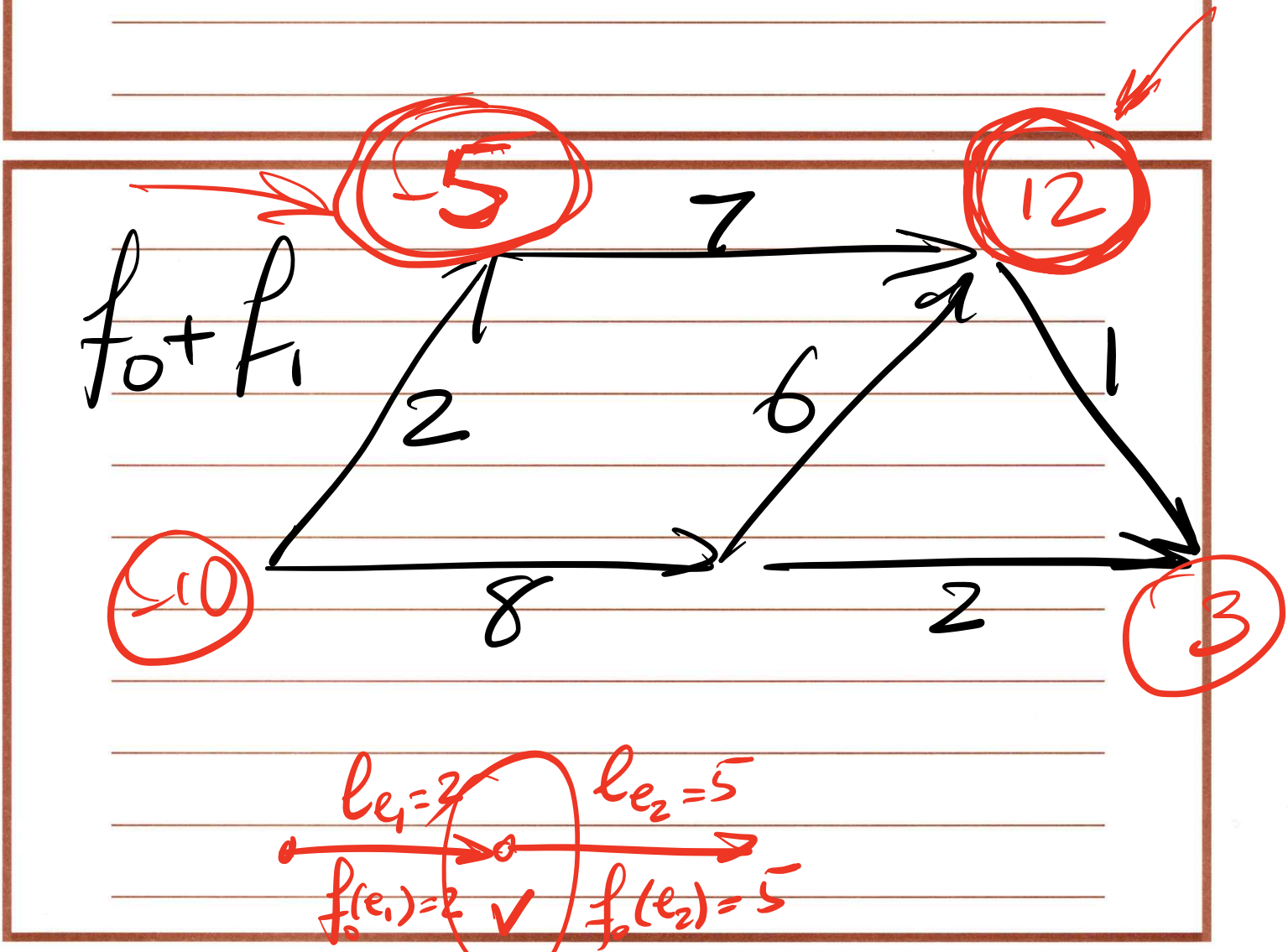

-5
C=7
12
ℓ=3
C=2
ℓ=2
C=7
ℓ=5
C=3
ℓ=1
C=10
ℓ=5
C=5
ℓ=2
-10
3


f₀
3
2
5
1
5
2


L_v
-1
7
-1
-2
3


G'
-4
4
5
0
2
5
-3
3
0
2





$l_{e_1}=2$ $l_{e_2}=5$

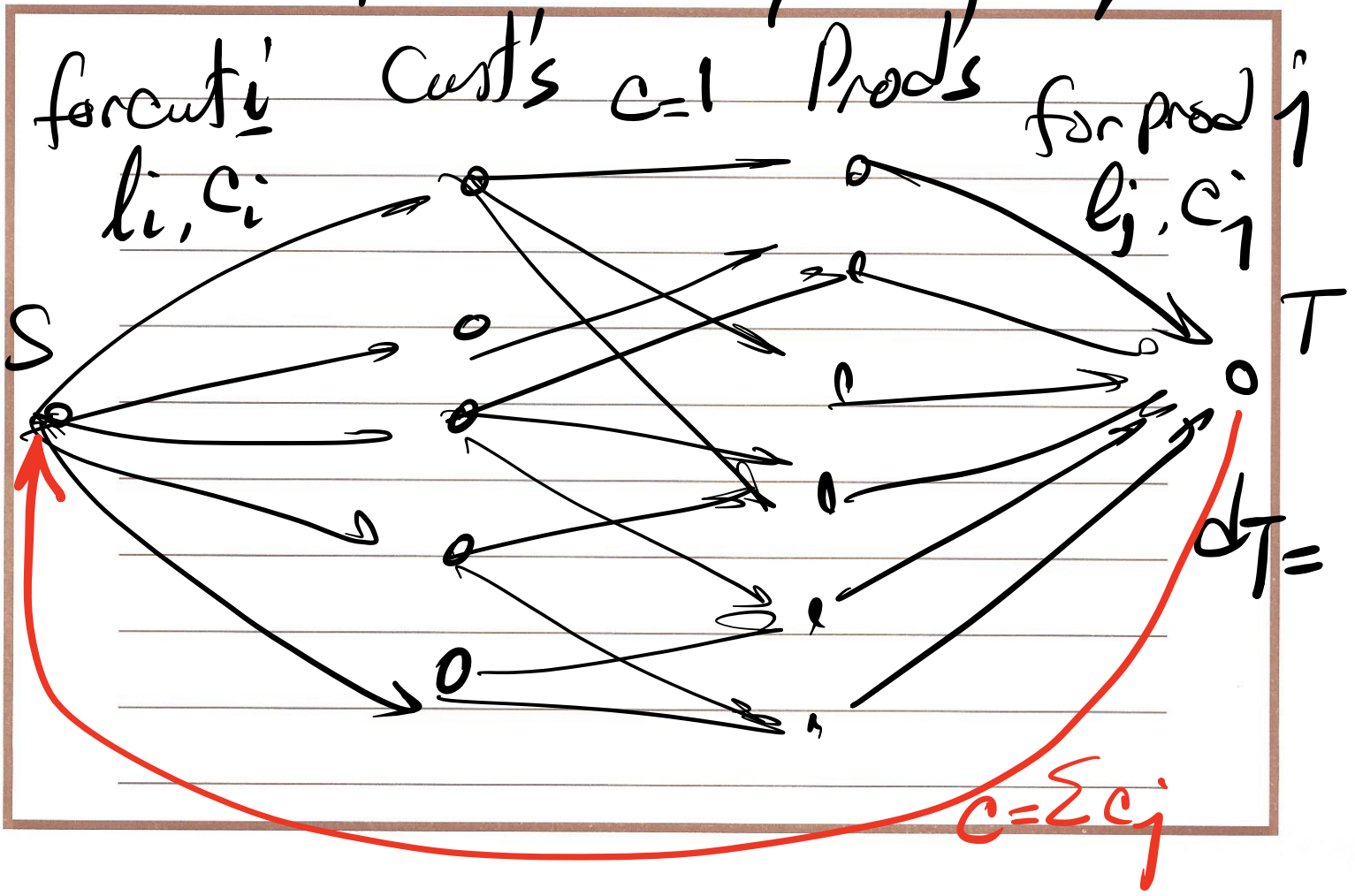
$f_0(e_1)=2$ ✓ $f_0(e_2)=5$

Survey Design Problem

Survey Design

Input: - Information on who purchased which products
- Maximum and Minimum number of questions to send to customer i
- Maximum and minimum number of questions to ask about product j

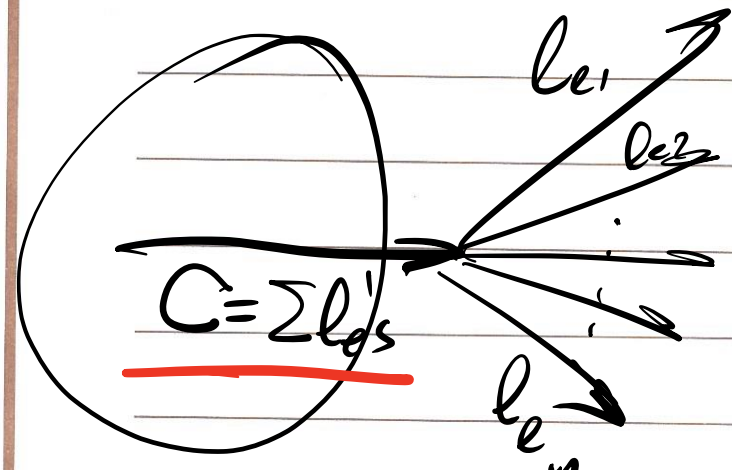
1 unit of flow will represent one question.



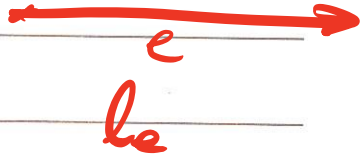
Min Flow Problems

Input: Directed graph $G=(V,E)$
Source $s \in V$
Sink $t \in V$
l_e for each edge $e \in E$
There are no c_e's

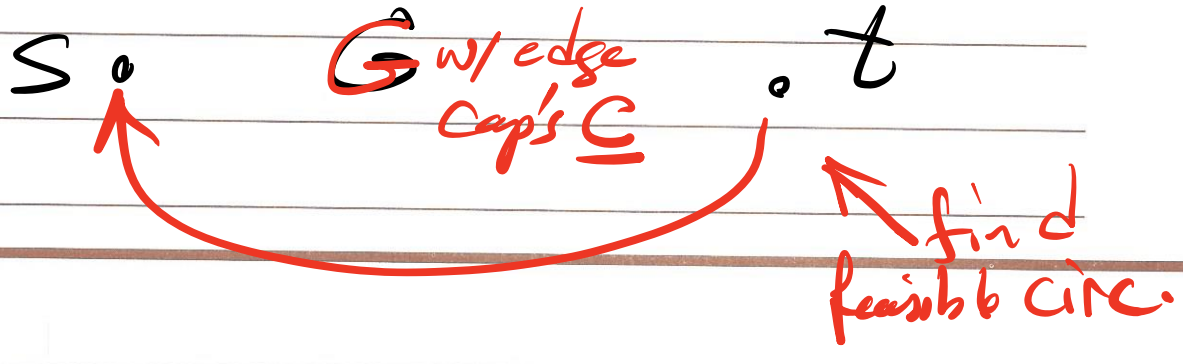
objective: Find a feasible flow of minimum possible value



excess flow $(f(e)-l_e)$



Construct a feasible circ. prob.



Solution:

- Assign "large" capacities to all edges and find a feasible flow f

- Construct G', where all the edges are reversed and the reversed edge e has capacity $= f_e - l_e$

- Find maximum flow from t to s in G'

- Min flow $= f - f'$

Discussion 9

1. We're asked to help the captain of the USC tennis team to arrange a series of matches against UCLA's team. Both teams have *n* players; the tennis rating (a positive number, where a higher number can be interpreted to mean a better player) of the *i*-th member of USC's team is t_i and the tennis rating for the *k*-th member of UCLA's team is b_k. We would like to set up a competition in which each person plays one match against a player from the opposite school. Our goal is to make *as many matches as possible* in which the USC player has a higher tennis rating than his or her opponent. Use network flow to give an algorithm to decide which matches to arrange to achieve this objective.

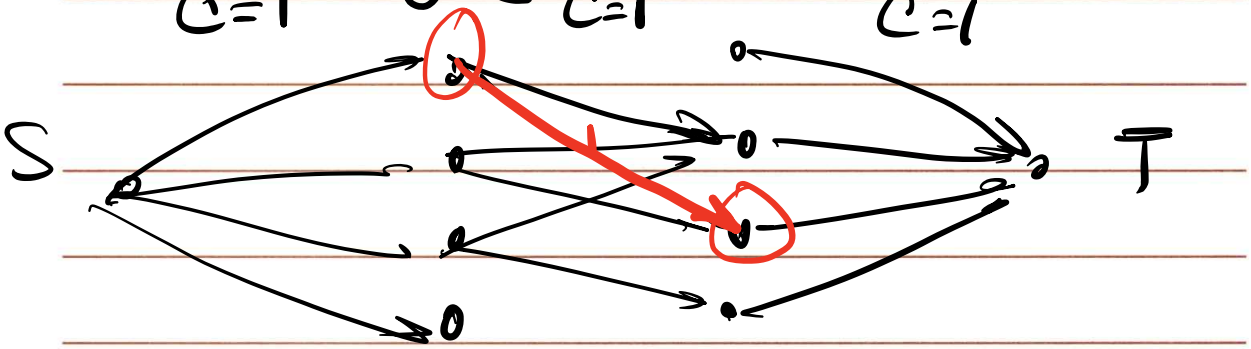
2. CSCI 570 is a large class with *n* TAs. Each week TAs must hold office hours in the TA office room. There is a set of *k* hour-long time intervals I_1, I_2, … I_k in which the office room is available. The room can accommodate up to 3 TAs at any time. Each TA provides a subset of the time intervals he or she can hold office hours with the minimum requirement of l_j hours per week, and the maximum m_j hours per week. Lastly, the total number of office hours held during the week must be *H*. Design an algorithm to determine if there is a valid way to schedule the TA's office hours with respect to these constraints.

3. There are n students in a class. We want to choose a subset of k students as a committee. There has to be m1 number of freshmen, m2 number of sophomores, m3 number of juniors, and m4 number of seniors in the committee. Each student is from one of k departments, where k = m1 +m2 +m3 +m4. Exactly one student from each department has to be chosen for the committee. We are given a list of students, their home departments, and their class (freshman, sophomore, junior, senior). Describe an efficient algorithm based on network flow techniques to select who should be on the committee such that the above constraints are all satisfied.

4. Given a directed graph G=(V,E) a source node s ϵ V, a sink node t ϵ V, and lower bound ℓ_e for flow on each edge e ϵ E, find a feasible s-t flow of minimum possible value.
Note: there are no capacity limits for flow on edges in G.

1. We're asked to help the captain of the USC tennis team to arrange a series of matches against UCLA's team. Both teams have n players; the tennis rating (a positive number, where a higher number can be interpreted to mean a better player) of the i-th member of USC's team is t_i and the tennis rating for the k-th member of UCLA's team is b_k. We would like to set up a competition in which each person plays one match against a player from the opposite school. Our goal is to make *as many matches as possible* in which the USC player has a higher tennis rating than his or her opponent. Use network flow to give an algorithm to decide which matches to arrange to achieve this objective.

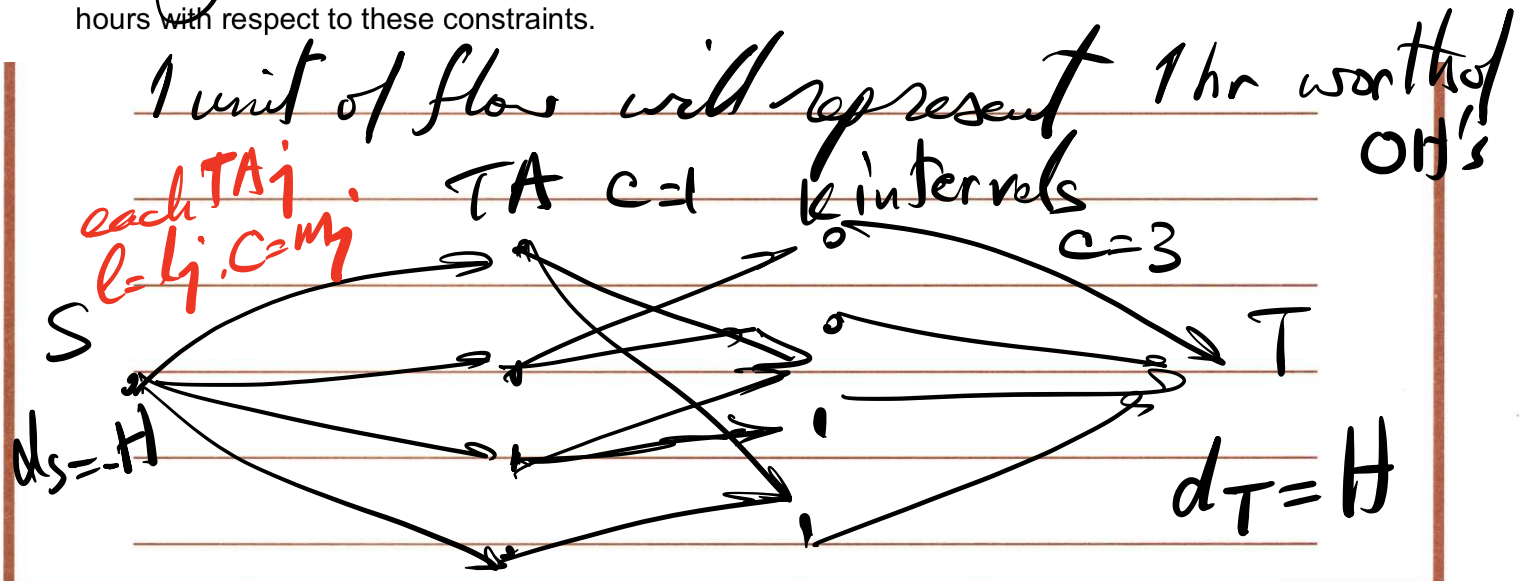
one unit of flow will represent 1 match.

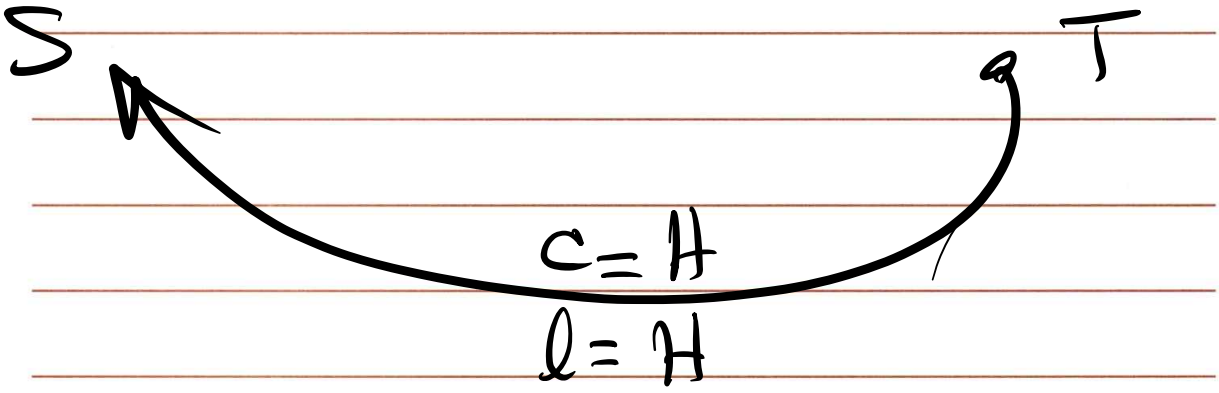


Run Max Flow $\longrightarrow f$

$v(f) = X$

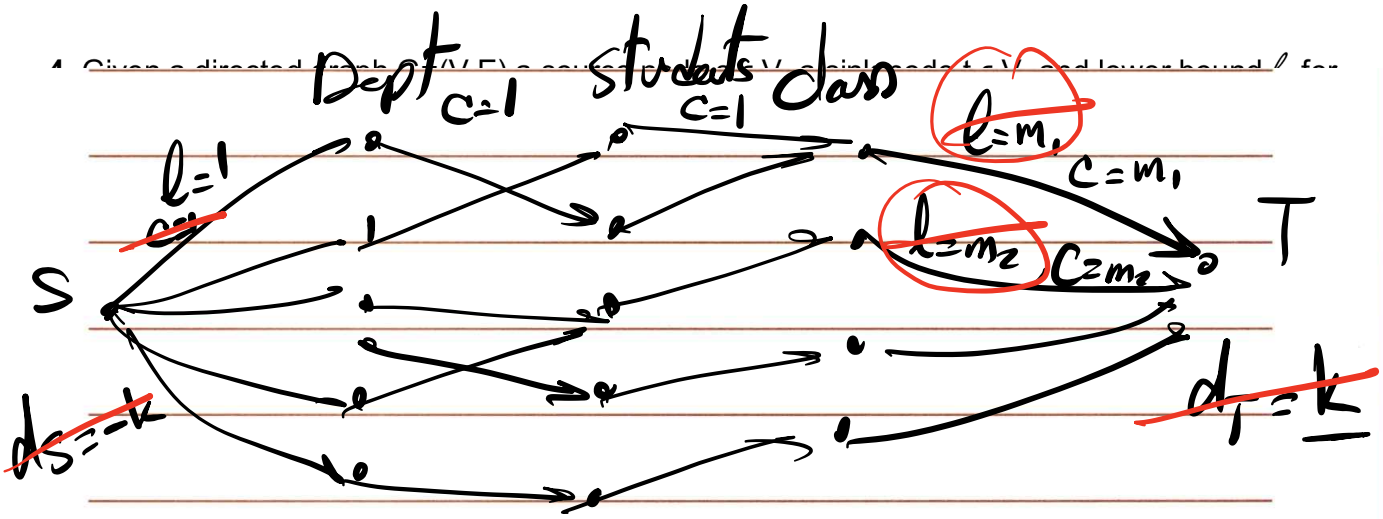
2. CSCI 570 is a large class with n TAs. Each week TAs must hold office hours in the TA office room. There is a set of k hour-long time intervals $I_1, I_2, \ldots I_k$ in which the office room is available. The room can accommodate up to 3 TAs at any time. Each TA provides a subset of the time intervals he or she can hold office hours with the minimum requirement of l_j hours per week, and the maximum m_j hours per week. Lastly, the total number of office hours held during the week must be H. Design an algorithm to determine if there is a valid way to schedule the TA's office hours with respect to these constraints.

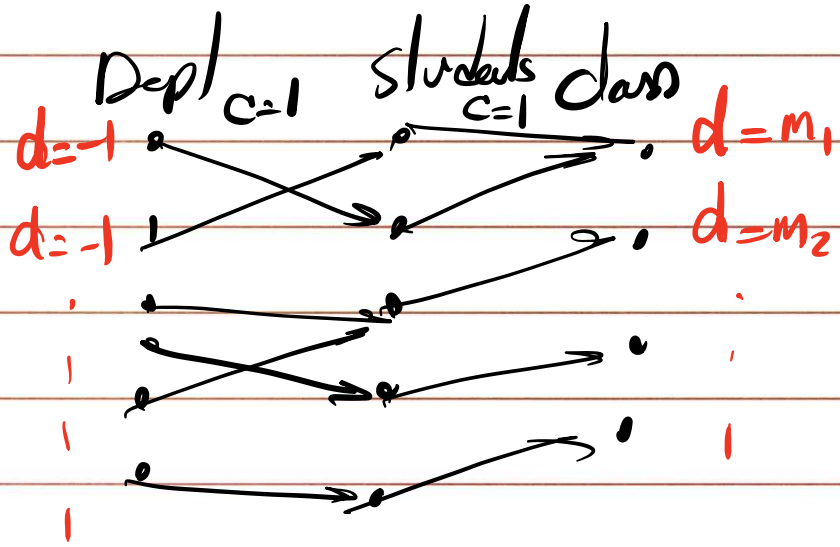




hours with respect to these constraints.

3. There are n students in a class. We want to choose a subset of k students as a committee. There has to be m1 number of freshmen, m2 number of sophomores, m3 number of juniors, and m4 number of seniors in the committee. Each student is from one of k departments, where k = m1 +m2 +m3 +m4. Exactly one student from each department has to be chosen for the committee. We are given a list of students, their home departments, and their class (freshman, sophomore, junior, senior). Describe an efficient algorithm based on network flow techniques to select who should be on the committee such that the above constraints are all satisfied.

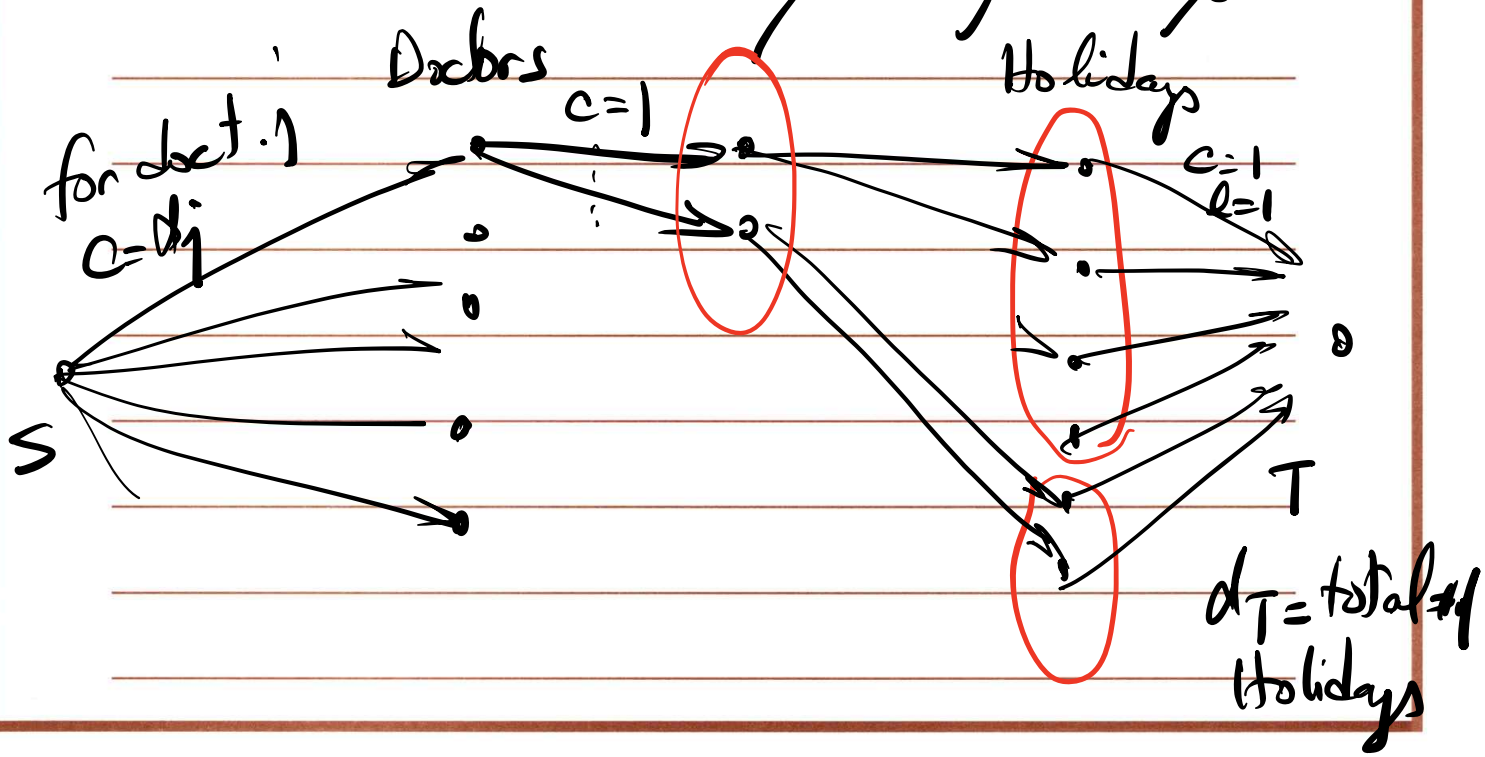




Doctors w/o weekends - Using gadgets

- Each Holiday i consists of h_i consecutive days.
- Each doctor specifies which of the days within each holiday (0 or more) they are available to be on call.
- Cannot have a doctor assigned to more than one day in each holiday
- Cannot assign a doctor j to more than d_j days total across all holidays

Objective: To have one doctor assigned to be on call on each day during holidays



Discussion 6

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order:$1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

Solution:
We will define COUNT(n, m) as the total number of ways to make change for amount m using coin denominations 1 to n.

The recurrence formula will be:
$COUNT(n,m) = COUNT(n-1, m) + COUNT(n, m- d_n)$

Note: COUNT(n-1, m) is the number of ways to make change for amount m without using coin denomination n. And $COUNT(n, m- d_n)$ represents the number of ways to make change for amount m using at least one coin of denomination n.

Initialization:
COUNT (i,0) = 1 for $0 < i \leq n$ since there is a way to pay the amount 0 (by paying 0 of all coin types)
COUNT(0,j) = 0 for $0 < j \leq m$ since there is no way to pay a non-zero amount without any coins

Bottom up pass:
For i = 1 to n
    For j = 1 to m
        COUNT(n,m) = COUNT(n-1, m)
        If ($m - d_n \geq 0$) COUNT(n,m) = COUNT(n,m) + $COUNT(n, m- d_n)$
    Endfor
Endfor

The total count will be at COUNT(n,m)

This will take O(mn) which is pseudopolynomial since m is the numerical value of an input term.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers

must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.

Solution:
We will define OPT(i) as the minimum cost of dinner for days 1 to i.

The recurrence formula will be:

$$OPT(i) = \begin{cases} OPT(i-1) \text{ if there is free food on day } i \\ \text{Otherwise, } Min(OPT(i-1) + 3 , OPT(i-7) + 10) \end{cases}$$

Initialization:
We need to initialize OPT(0..6). These are trivial problems to solve.

Bottom up pass:

```
For i = 7 to n
        If Free(i) then
                OPT(i) = OPT(i-1)
        Else
                OPT(i) = Min( OPT(i-1) + 3 , OPT(i-7) + 10 )

        Endif
Endfor
```

The minimum cost of dinner will be at OPT(n).

This will take O(n) which is polynomial if we assume that the input consists of an array of size n called Free() where Free(i) is true when there is free food on day i, and false otherwise.

To be able to determine the dinner schedule, we will go top down:

```
i = n
While i > 0
        If then
                Free(i) Mark up the calendar with “free food” on day i
                i = i - 1
        Else if OPT(i-1) + 3 < OPT(i-7) + 10 then
                Mark up the calendar with “cafeteria” on day i
                i = i - 1
        Else
                Mark up the calendar with “go grocery shopping” on day i-6
                Mark up the calendar with “eat at home” on days i-5 to i
```

i = i - 7
Endif
Endwhile

Note: on the day we “go grocery shopping” we also “eat at home”.

The top down pass will also take O(n) time. So the whole solution runs in O(n) time.

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

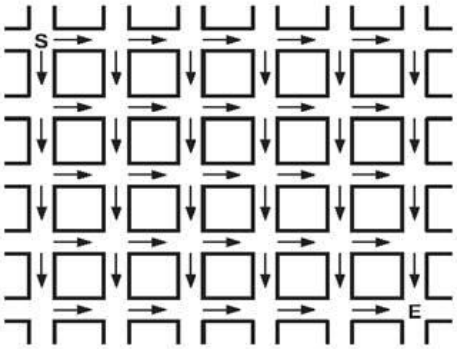


Figure A.

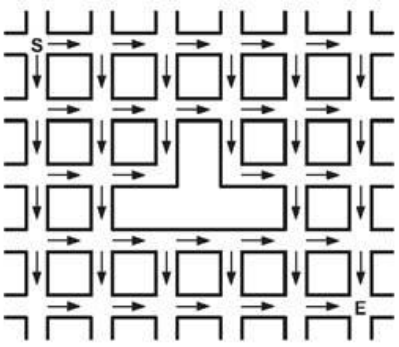


Figure B.

Solution:
Let’s say E is at coordinates (0,0) and s is at coordinates (n,m).
We will define COUNT(n, m) as the total number of ways to go from coordinates (n,m) to (0,0).

The recurrence formula will be:
COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)

Initialization:
COUNT (i,0) = 1 for $0 < i \leq n$ since there is only one way to go (horizontally) from (i,0) to (0,0)
COUNT (0,j) = 1 for $0 < j \leq n$ since there is only one way to go (vertically) from (0,j) to (0,0)

Bottom up pass:
For i = 1 to n
For j = 1 to m

```
                COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)
        Endfor
Endfor
```

The total count will be at COUNT(n,m)

This will take O(mn) which is pseudo-polynomial if we assume that the input only consists of the coordinates n and m. If the input consisted of the 2D array of all intersections, then the run time will be polynomial.

b) Repeat this process with Figure B; be wary of dead ends.

We can use the same approach and recurrence formula as in part a, except that we need to apply special recurrence formulae to the intersections that are affected namely (2, 2) and (3, 2). So, the implementation will be modified like this:

```
Bottom up pass:
For i = 1 to n
        For j = 1 to m
                If (n=2 and m=2) then
                        COUNT(i,j) = COUNT(i-1, j)
                Else if (n=3 and m=2) then
                        COUNT(i,j) = 0
                Else
                        COUNT(i,j) = COUNT(i, j-1) + COUNT(i-1, j)
                Endif
        Endfor
Endfor
```

Discussion 7

1. When their respective sport is not in season, USC's student-athletes are very involved in their community, helping people and spreading goodwill for the school. Unfortunately, NCAA regulations limit each student-athlete to at most one community service project per semester, so the athletic department is not always able to help every deserving charity. For the upcoming semester, we have S student-athletes who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. There are F projects under consideration; project i requires s_i student-athletes and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university. Our goal is to maximize the goodwill generated for the university subject to these constraints. Note that each project must be undertaken entirely or not done at all -- we cannot choose, for example, to do half of project i to get half of g_i goodwill.

Solution:
This is very similar to the 0/1 knapsack problem except that we have two types of resources (students and buses) instead of one resource.

Value of the optimal solution for unique subproblems can be described as:
OPT(i, S, B) = Maximum goodwill that can be generated using projects 1 to i, with S students and B buses.

The recurrence formula will be:
$OPT(i, S, B) = Max (g_i + OPT(i-1, S- s_i, B- b_i) , OPT(i-1, S, B))$

Initialization: OPT(0, S, B) and OPT(i, 0, B) and OPT(i, S, 0) can be set to zero since there will no goodwill if there are no projects or students or buses.

Find the value of the optimal solution for all unique subproblems (bottom up):

```
For i = 1 to F
        For j= 1 to S
                For k= 1 to B
                        If ( si > j or bi > k) then
                           OPT(i, j, k) = OPT(i-1, j, k)
                        Else
                          OPT(i, j, k) = Max ( gi + OPT(i-1, j- si, k- bi) , OPT(i-1, j, k))
                        Endif
                Endfor
        Endfor
```

Endfor

OPT(F, S, B) will hold the value of the optimal solution for the whole problem.

Complexity of this solution is O(FSB), which is pseudo-polynomial because S and B represent numerical values of the input terms rather than the size of the input.

To find the set of projects that we should select, we can then go top down (and print the list of projects selected):

```
j = S
k = B
For i= F to 1 by -1
        If g_i + OPT(i-1, j- s_i, k- b_i) > OPT(i-1, j, k) then
                Print i
                j = j- s_i
                k = k- b_i
        Endif
Endfor
```

The top down pass takes O(F) time. So, the whole solution will take O(FSB) time.

2. Suppose you are organizing a company party. The corporation has a hierarchical ranking structure; that is, the CEO is the root node of the hierarchy tree, and the CEO's immediate subordinates are the children of the root node, and so on in this fashion. To keep the party fun for all involved, you will not invite any employee whose immediate superior is invited. Each employee *j* has a value v_j (a positive integer), representing how enjoyable their presence would be at the party. Our goal is to determine which employees to invite, subject to these constraints, to maximize the total value of invitees.

We define the value of an optimal solution for a unique subproblem as:
OPT(i) = maximum “enjoyability value”/”fun value” of the subtree rooted at node i

Recurrence formula:
$OPT(i) = \text{Max} (v_i + \Sigma_{g \in g_i} OPT(g) , \Sigma_{c \in c_i} OPT(c))$
Where g_i are the grandchildren of i and c_i are the children of i.

We store the OPT values at the corresponding nodes in the company hierarchy tree. So there will be no need to create any external arrays to hold OPT.

Bottom up pass:
At every node i we compute and send up (to the parent node) the following two values:

- OPT(i) = Max ($v_i + \Sigma_{g \in gi}$ OPT(g) , $\Sigma_{c \in ci}$ OPT(c))
- $\Sigma_{c \in ci}$ OPT(c) (or zero if i has no children)
  Note: we send up the value $\Sigma_{c \in ci}$ OPT(c), so that the parent node can easily compute the sum $\Sigma_{g \in gi}$ OPT(g) since the children of i are the grandchildren of I's parent node.

This pass takes linear time with respect to the size of the tree since each node is examined only once and the total number of additions performed is O(n).

To find out who will be invited to the party we go top down. We call the **invite** function with the root node of the tree.

Invite (i)

```
If v_i + Σ_{g ∈ gi} OPT( g ) > Σ_{c ∈ ci} OPT( c )  Then
        Print i
        Call Invite with all g ∈ g_i
Else
        Call Invite with all c ∈ c_i

Endif
```

The top down pass will take O(n) time since it will only be called a maximum of n times and the work inside the function is constant time (since the two sums $v_i + \Sigma_{g \in gi}$ OPT(g) and $\Sigma_{c \in ci}$ OPT(c) can be stored at each node during the bottom up pass.)

3. You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

We will first remove the complexity related to the rotation of the boxes by creating 3n box types as follows.
Given a box with sides measuring X < Y < Z, we will create 3 box types:

Width	Depth	Height
X	Y	Z
Y	Z	X
X	Z	Y

Now each of the 3n box types can only be used once in the stack. We then sort the 3n boxes based on decreasing base area (Width X Depth).

We define the value of an optimal solution for a unique subproblem as:
OPT(i) = Maximum height of a stack of boxes with box i at the top.

Recurrence formula will be:
$OPT(i) = h(i) + \text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$

Note: we assume that Max will return a value of zero if we don't find a compatible box that i can fit on top of.

For i = 1 to 3n
    $OPT(i) = h(i) + \text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$
    Base(i) = box no. corresponding to the value $\text{Max}_{0<j<i \text{ and } i \text{ can fit on top of } j} (OPT(j))$
Endfor
Note: The maximum value will not be necessarily at OP(3n). So we need to perform a linear search on OPT to find where the maximum value appears. We are also saving the id of the box we are putting i on, to simplify the top down pass. If box i does not fit on top of any other boxes, then Base(i) = 0

This will take $O(n^2)$ since at each of the 3n iterations we need to find the maximum of up to 3n terms.

To find the boxes that will be going into the highest stack we can start with the box that has the maximum OPT value. Let's call that box i.

Print i
While Base(i) > 0
    i = Base (i)
    Print i
Endwhile

The top down pass will only take O(n) time because we had saved the Base array in the bottom up pass. Otherwise, if this were not saved the top down pass would also take $O(n^2)$ time.

Discussion 8

1. You have successfully computed a maximum s-t flow f for a network G = (V; E) with integer edge capacities. Your boss now gives you another network G' that is identical to G except that the capacity of exactly one edge is decreased by one. You are also explicitly given the edge whose capacity was changed. Describe how you can compute a maximum flow for G' in O(|V| + |E|) time.

Solution:
Say the edge vu the edge that has lost capacity, then

- Find a path in G from s to v on edges that are carrying some flow
- Find a path in G from u to t on edges that are carrying some flow
- Reduce flow by unit on edges that are on the path s -> v – u -> t. Since all these edges are carrying some flow, then the results will be a new valid flow f'
- Construct $G_{f'}$
- Try to find a path in $G_{f'}$ from s to t. If there is such a path then push (augment) one more unit of flow on that path. The resulting flow will be our max flow. If there is no such path then f'
  Is our max flow.

Complexity: all the above steps can be done in linear time, therefore the whole process takes linear time.

2. You need to transport iron-ore from the mine to the factory. We would like to determine how long it takes to transport. For this problem, you are given a graph representing the road network of cities, with a list of *k* of its vertices ($t_1, t_2, \ldots, t_k$) which are designated as factories, and one vertex *S* (the iron-ore mine) where all the ore is present.
We are also given the following:

- Road Capacities (amount of iron that can be transported per minute) for each road (edges) between the cities (vertices).
- Factory Capacities (amount of iron that can be received per minute) for each factory (at $t_1, t_2, \ldots, t_k$)
- The amount of ore to be transported from the mine, *C*

Give a polynomial-time algorithm to determine the minimum amount of time necessary to transport and receive all the iron-ore at factories.

Solution:
Construct a flow network G as follows:

- Choose S as the source
- Create a new sink node called t. Add directed edges from each t_i to t with edge capacity associated with factory I's receiving capacity.
- Replace each undirected edge (roads) with two directed edges with the same capacity as the original undirected edge

Find max flow f in G using a polynomial time max flow algorithm. The time required to move all C units of iron-ore to the factories will be T = C/v(f)

3. In a daring burglary, someone attempted to steal all the candy bars from the CS department. Luckily, he was quickly detected, and now, the course staff and students will have to keep him from escaping from campus. In order to do so, they can be deployed to monitor strategic routes.

More formally, we can think of the USC campus as a graph, in which the nodes are locations, and edges are pathways or corridors. One of the nodes (the instructor's office) is the burglar's starting point, and several nodes (the USC gates) are the escape points — if the burglar reaches any one of those, the candy bars will be gone forever. Students and staff can be placed to monitor the edges. As it is hard to hide that many candy bars, the burglar cannot pass by a monitored edge undetected.

Give an algorithm to compute the minimum number of students/staff needed to ensure that the burglar cannot reach any escape points undetected (you don't need to output the corresponding assignment for students — the number is enough). As input, the algorithm takes the graph G = (V,E) representing the USC campus, the starting point s, and a set of escape points $P \subseteq V$. Prove that your algorithm is correct and runs in polynomial time.

Solution:

This is a min-cut problem. I.e. we need to find the minimum number of edges in the network whose removal will disconnect source (CS dept) from the sink (all exit points). Here are the steps to create the flow network G:

- CS Dept will be the source s
- Create a new sink node t and connected all exit points to t with a directed edge having a very large capacity (at least m). The reason for the large capacity is that we don't want any of these new edges to appear on the min cut since these are not really representing pathways or corridors that could be monitored.
- Replace each undirected edges (pathways) with two directed edges each with capacity 1

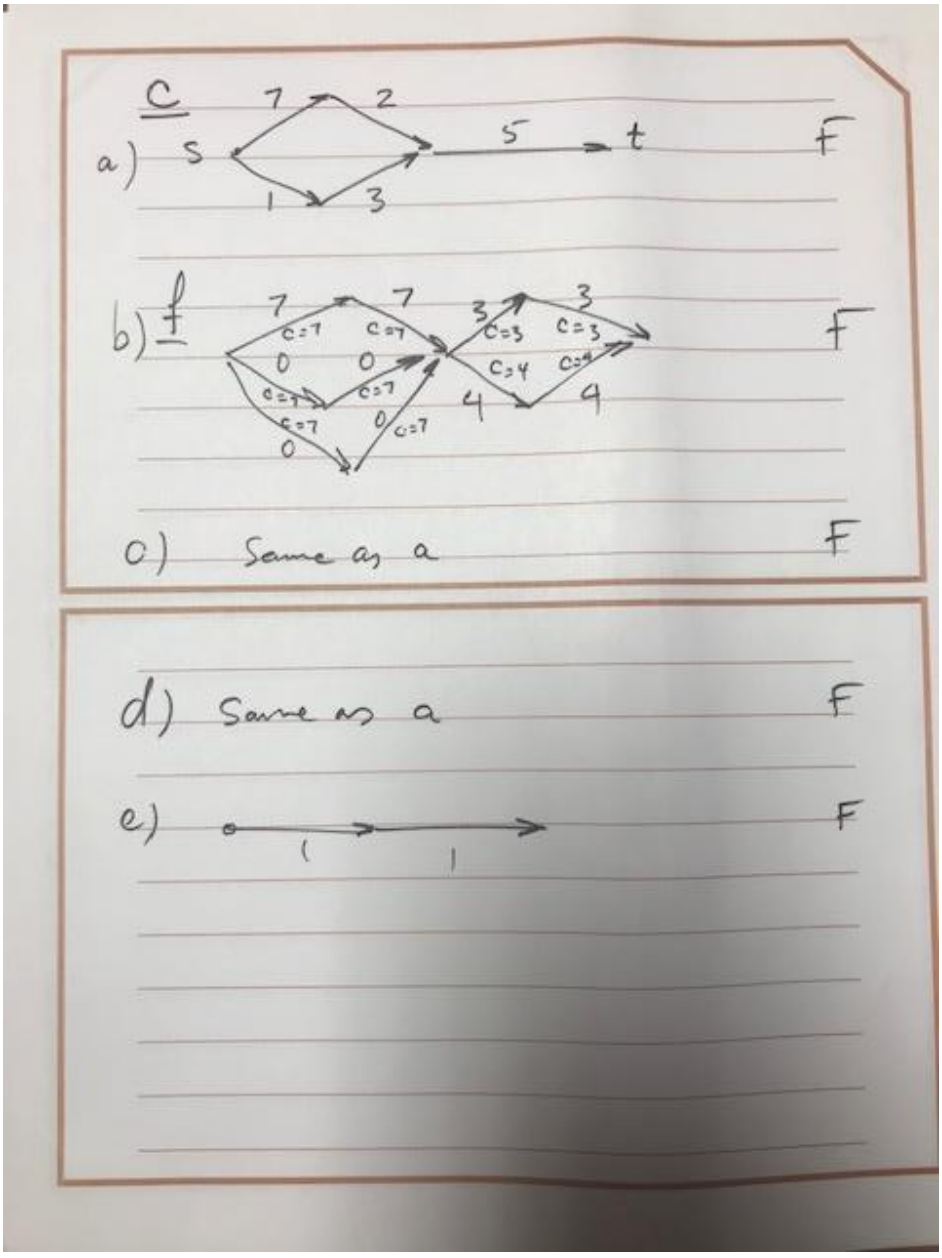
We then find max flow f in G. v(f) will be the minimum number of students/staff needed to ensure that the burglar cannot reach any escape points undetected.
Complexity: even if we use Ford-Fulkerson to solve the max flow problem in G, our solution will be polynomial time since edge capacities for edges leaving S are all 1 and there are only O(n) of them. So the complexity of the FF algorithm will be O(nm) for this particular problem.

If we are also interested in finding the exact pathways to assign the staff to, (having max flow f) we will follow the process described in lecture to find a min cut in G.

4. We define a most vital edge of a network as an edge whose deletion causes the largest decrease in the maximum s-t-flow value. Let f be an arbitrary maximum s-t-flow. Either prove the following claims or show through counterexamples that they are false:

(a) A most vital edge is an edge e with the maximum value of c(e).
(b) A most vital edge is an edge e with the maximum value of f (e).
(c) A most vital edge is an edge e with the maximum value of f (e) among edges belonging to some minimum cut.
(d) An edge that does not belong to any minimum cut cannot be a most vital edge.
(e) A network can contain only one most vital edge.

All are false. See counterexamples below.



Discussion 9

1. We're asked to help the captain of the USC tennis team to arrange a series of matches against UCLA's team. Both teams have *n* players; the tennis rating (a positive number, where a higher number can be interpreted to mean a better player) of the *i*-th member of USC's team is t_i and the tennis rating for the *k*-th member of UCLA's team is b_k. We would like to set up a competition in which each person plays one match against a player from the opposite school. Our goal is to make *as many matches as possible* in which the USC player has a higher tennis rating than his or her opponent. Use network flow to give an algorithm to decide which matches to arrange to achieve this objective.

Solution: Reduce the tournament scheduling problem to Max Flow as follows.
We will construct a flow network G such that G can have a flow of value k iff we can set up k tournament matches such that a USC player has a higher ranking than the UCLA player they play against. The construction will involve the following sets of nodes and edges:

- n nodes representing USC players and n nodes representing UCLA players + a sink node and a source node
- We will connect edges with capacity one from the source to each USC player. We will connect edges from each UCLA player to the sink with capacity one. We will connect an edge from a USC player to a UCLA player if the USC player has a higher ranking than the UCLA player

We will find Max Flow *f* in G. v(*f*) will be the number of games in which the USC player has a higher ranking than the UCLA player. The edges carrying flow from a USC player to a UCLA player identify such matches.

If v(*f*) < n, then the rest of the players can be matched randomly.

Proof: If we were asked to prove that our solution is correct, we would need to show the following:
A – If I find k matches in which USC has an advantage, I can then find a flow of value k in G.
B – If we can find a flow of value k in G, we can then find k matches in which USC has an advantage.

To prove A, we can say that if we are given k matches in which USC has an advantage, then we can find k edge disjoint s-t paths (since a given player only participates in one match) in G each one capable of carrying 1 unit of flow. Therefore, we can find a flow of value k in G.

To prove B, we can say that if we have a flow of value k in G, we must have k edges that carrying one unit of flow from a USC player to a UCLA player. Since there is only one unit of flow coming into each node on the USC side and only one unit of flow that can leave a node on the UCLA side, then this edge cannot share any nodes with other edges carrying flow from the USC side to the UCLA side. And since each of these edges identify a pair in which the USC

player has a higher ranking that the UCLA player, we can form k independent matches where USC has an advantage.

2. CSCI 570 is a large class with *n* TAs. Each week TAs must hold office hours in the TA office room. There is a set of *k* hour-long time intervals I_1, I_2, … I_k in which the office room is available. The room can accommodate up to 3 TAs at any time. Each TA provides a subset of the time intervals he or she can hold office hours with the minimum requirement of l_j hours per week, and the maximum m_j hours per week. Lastly, the total number of office hours held during the week must be *H*. Design an algorithm to determine if there is a valid way to schedule the TA's office hours with respect to these constraints.

Solution: Reduce the TA scheduling problem to Circulation with Lower Bounds as follows.
We will construct a circulation network G such that G can have a feasible flow iff we can schedule TA office hours with all the given constraints. The construction will involve the following sets of nodes and edges:

- n nodes representing TAs and k nodes representing the hou-long time intervals, a node called s and a node called t
- We will have directed edges from s to each of the n TA nodes, the edge connecting to TA j will have capacity m_j and lower bound l_j. We will also have edges from each time interval to t with capacity of 3. We will also draw a directed edge from each TA to the set of time intervals in which that TA is available.
- We will place a demand value of H at t and a supply value of -H at s.

We will then solve the feasible circulation problem. If there is a feasible circulation in G, then we can find a feasible assignment of TAs to office hours. Otherwise, there will not be a feasible assignment of TAs to office hours.

3. There are n students in a class. We want to choose a subset of k students as a committee. There has to be m1 number of freshmen, m2 number of sophomores, m3 number of juniors, and m4 number of seniors in the committee. Each student is from one of k departments, where k = m1 +m2 +m3 +m4. Exactly one student from each department has to be chosen for the committee. We are given a list of students, their home departments, and their class (freshman, sophomore, junior, senior). Describe an efficient algorithm based on network flow techniques to select who should be on the committee such that the above constraints are all satisfied.

Solution: Reduce the committee assignment problem to Max Flow as follows.
We will construct a flow network G such that G can have a max flow of value k iff there is a feasible assignment of students to the committee. The construction will involve the following sets of nodes and edges:

- 4 nodes representing freshman, sophomore, junior, and senior classes. n nodes representing students, k nodes representing departments, a node called s and a node called t.

- We will add directed edges from s to the four nodes representing freshman, sophomore, junior, and senior classes with capacities of m1, m2, m3, and m4 respectively. We will connect edges from each node representing a class to all students belonging to that class with capacity of 1. We will add edges from each student to the department they belong to with a capacity of 1. We will add edges from each department to t with a capacity of 1.

We will find max flow *f* in G. If v(*f*) = k, then there is a feasible assignment of students to the committee, otherwise there is no such feasible solution.
Note: if If v(*f*) = k, it means that each edge from a department to t is saturated and also each edge from s to the four classes is saturated (since m1+m2+m3+m4=k). Therefore, the constraints related taking one students from each department and selecting an exact number of freshmen, sophomores, etc. are all met.

4. Given a directed graph G=(V,E) a source node s ϵ V, a sink node t ϵ V, and lower bound ℓ_e for flow on each edge e ϵ E, find a feasible s-t flow of minimum possible value.
Note: there are no capacity limits for flow on edges in G.

We will solve this problem in two passes. In pass 1 we will find a feasible circulation that meets all lower bound constraints. In the second pass we will take away as much flow from the feasible flow found in the first pass without violating the minimum flow constraints on each edge. Here are the details:

Pass 1: Assign a capacity of $C = \Sigma_{(e \epsilon E)} \ell_e$ to all edges in G. Add an edge from t to s with the same capacity and find a feasible circulation f_1 in G. Note that C units of capacity will be sufficient to accommodate all minimum flow requirements on all edges, so there will always be a feasible circulation in G. But this feasible circulation may not have the minimum value of flow leaving the source. So in the next pass we will remove the excess flow.

Pass 2: Construct G' by reversing the edge directions in G (same nodes and edges but edges are in the opposite direction). Then for each edge e we assign it a capacity of $f_1(e) - \ell_e$. Note that this amount ($f_1(e) - \ell_e$) is the excess flow that can be potentially removed from edge e. G' will have no lower bound constraints on flow and will simply be a flow network.
We then find max flow f_2 in G'.

Finally, we will find min flow f_{min} by subtracting f_2 from f_1. For example, if the flow f_1 over edge e is 10 units and f_2 over edge e (but in the opposite direction) is 3 units, then $f_{min}(e)$ will be 7 units.

CSCI 570 - Fall 2021 - HW 6 - Solution

Graded Problems

For the grade problems, you must provide the solution in the form of pseudo-code and also give an analysis on the running time complexity.

Problem 1

[**10 points**] Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a **Dynamic Programming** algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

Solution: At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and recursively compute the maximum points we can get for the other piece. It is also possible to recursively attempt to obtain the maximum value for both pieces after the cut, but that requires adding an extra check to see if the value obtained by selling a rod of this length is more than recursively cutting it up.

The bottom-up pseudo-code to obtain the maximum amount of money is:

Algorithm 1 Bottom-Up-Cut-Rod(p, n)

```
Let r[0, ..., n] be a new array
r[0] = 0
for j = 1 to n do
    q = -∞
    for i = 1 to j do
        q = max(q, p[i] + r[j − 1])
    end for
end for
return r[n]
```

The time complexity of this algorithm is $\theta(n^2)$ because of the double nested for loop.

Rubric:

- 5 points for a correct dynamic programming solution

- 3 points if the solution runs in $\theta(n^2)$
- 2 points for providing analysis of runtime complexity

Problem 2

[**10 points**] Tommy and Bruiny are playing a turn-based game together. This game involves N marbles placed in a row. The marbles are numbered 1 to N from the left to the right. Marble i has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive points equal to the sum of the remaining marbles' values in the row. The winner is the one with the higher score when there are no marbles left to remove.

Tommy always goes first in this game. Both players wish to maximize their score by the end of the game.

Assuming that both players play optimally, devise a **Dynamic Programming** algorithm to return the difference in Tommy and Bruiny's score once the game has been played for any given input.

Your algorithm **must** run in $O(N^2)$ time.

Solution:
We first calculate a prefix sum for the marbles array. This enables us to find the sum of a continuous range of values in $O(1)$ time.
If we have an array like this: $[5, 3, 1, 4, 2]$, then our prefix sum array would be $[0, 5, 8, 9, 13, 15]$.
Once we've done that, we can define OPT(i, j) as the maximum difference in score achievable by the player whose turn it is to play, given that the marbles from index i to j (inclusive) remain.

The pseudo-code for this algorithm, assuming 0-indexed arrays, is:

Algorithm 2 Max-Difference-Scores(*marbles*)

Let n be the length of the marbles array
Let $prefix\_sum$ be the calculated prefix sum array for the array marbles [takes $\theta(n)$ time]
Let $OPT[[0, ..., 0], ..., [0, ..., 0]]$ be a new $n*n$ array, with values initialized to 0

for $i = n - 2$ to 0 **do**
  for $j = i + 1$ to $n - 1$ **do**
    $score\_if\_take\_i = prefix\_sum_{j+1} - prefix\_sum_{i+1} - OPT_{i+1,j}$
    $score\_if\_take\_j = prefix\_sum_{j} - prefix\_sum_{i} - OPT_{i,j-1}$
    $OPT_{i,j} = max(score\_if\_take\_i, score\_if\_take\_j)$
  end for
end for
return $OPT_{0,n-1}$

The time complexity of this algorithm is $\theta(n^2)$ if a prefix sum array is calculated initially due to there being $n * n$ subproblems to calculate.

Rubric:

- 8 points for a correct dynamic programming solution
- 2 points for providing analysis of runtime complexity

Problem 3

[**10 points**] The Trojan Band consisting of n band members hurries to lined up in a straight line to start a march. But since band members are not positioned by height the line is looking very messy. The band leader wants to pull out the minimum number of band members that will cause the line to be in a *formation* (the remaining band members will stay in the line in the same order as they were before). The formation refers to an ordering of band members such that their heights satisfy $r_1 < r_2 < ... < r_i > ... > r_n$, where $1 \leq i \leq n$.

For example, if the heights (in inches) are given as

$$R = (67, 65, 72, 75, 73, 70, 70, 68)$$

the minimum number of band members to pull out to make a formation will be 2, resulting in the following formation:

$$(67, 72, 75, 73, 70, 68)$$

Give an algorithm to find the minimum number of band members to pull out of the line.

Note: you do not need to find the actual formation. You only need to find the minimum number of band members to pull out of the line, but you need to find this minimum number in $O(n^2)$ time.

For this question, you must write your algorithm using pseudo-code.

Solution: This problem performs a $Longest - Increasing - Subsequence$ operation twice. Once from left to right and once again, but from right to left. Once we have the values for the longest subsequence we can make after we "pull out" a few band members, we can iterate over the array and determine what minimum number of pull-outs will cause our line of band members to satisfy the height order requirements.

Let $OPT_{left}(i)$ be maximum length of the line to the left of band member i (including i) which can be put in order of increasing height (by pulling out some members) Note: the problem is symmetric, so we can flip the array R and find the same values from the other direction. Let's call those values $OPT_{right}(i)$.

The recurrence relations are:

$$OPT_{left}(i) = max(OPT_{left}(i), OPT_{left}(j) + 1) \text{ such that } r_i > r_j \quad \forall \quad 1 \leq j < i$$
$$OPT_{right}(i) = \text{same once the array is flipped}$$

Pseudo code:

for $i = 1$ to n **do**
  $OPT_{left}(i) = OPT_{right}(i) = 1$ ▷ only choose itself
end for
for $i = 2$ to $n - 1$ **do**
  for $j = 1$ to $i - 1$ **do**
    if $r_i > r_j$ **then**
      $OPT_{left}(i) = max(OPT_{left}(i), OPT_{left}(j) + 1)$
    end if
  end for
end for
for $i = 2$ to $n - 1$ **do**
  for $j = 1$ to $i - 1$ **do**
    if $r_{n-i+1} > r_{n-j+1}$ **then**
      $OPT_{right}(i) = max(OPT_{right}(i), OPT_{right}(j) + 1)$
    end if
  end for
end for
result $= -\infty$
for $i = 1$ to n **do**
  result $= min(\text{result}, n - (OPT_{left}(i) + OPT_{right}(i) + 1))$
end for
return result

The runtime of this algorithm is dominated by the nested for loops used to calculate the LIS both ways. This takes $\theta(n^2)$ each time. Therefore, the time complexity of the solution is $\theta(n^2)$.

Rubric:

- 8 points for a correct dynamic programming solution in pseudo-code
- 2 points for providing analysis of runtime complexity

Problem 4

[**15 points**] You've started a hobby of retail investing into stocks using a mobile app, RogerGood. You magically gained the power to see N days into the future and you can see the prices of one particular stock. Given an array of prices of this particular stock, where $prices[i]$ is the price of a given stock on the ith day, find the maximum profit you can achieve through various buy/sell

actions. RogerGood also has a fixed *fee* per transaction. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Solution:

There are many ways to do this. If a greedy solution is given, the student must prove that it is correct. The dynamic programming way of solving this can be done in many ways.

One way to do it is:

Consider $buy(i)$ to be the maximum profit you can make if you start at day i, assuming you currently do not own a unit of stock.
Consider $sell(i)$ to be the maximum profit you can make starting at day i, assuming you already own a unit of the stock.

We will apply the transaction fee during the sale of a stock.

Algorithm 3 Bottom-Up-Max-Profit(*prices*, *fee*)

Let n be the length of the prices array
Let $buy[0, ..., n+1]$ be a new array, with values initialized to 0
Let $sell[0, ..., n+1]$ be a new array, with values initialized to 0

for $i = n$ to 0 **do**
  $buy[i] = max(sell[i+1] - prices[i], buy[i+1])$
  $sell[i] = max(buy[i+1] + prices[i] - fee, sell[i+1])$
end for

return buy[0]

The time complexity of this solution is $\theta(n)$. We use a single for-loop to compute the maximum profits at each stage.

Rubric:

- 10 points for a correct dynamic programming solution
- 3 points if the solution runs in $\theta(n)$
- 2 points for providing analysis of runtime complexity

Practice Problems

Problem 5

[**10 points**] From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have infinitely many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size w_i and value v_i. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

Solution:

Similar to what is taught in the lecture, let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$. We find the recurrence relation of $OPT(k, w)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

- We include another item of type k and solve the sub-problem $OPT(k, w - v_k)$.
- We do not include any item of type k and move to consider next type of item this solving the sub-problem $OPT(k-1, w)$.

Therefore, we have

$$OPT(k, w) = \max\{OPT(k-1, w), OPT(k, w - w_k) + v_k * i\}.$$

Moreover, we have the initial condition $OPT(0, 0) = 0$.

Problem 6

Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get $nums[left] * nums[i] * nums[right]$ coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design an dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

Here is an example. If you have the nums arrays equals $[3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5 3 and 8. The left balloons after each step is:

$$[3, 1, 5, 8] \rightarrow [3, 5, 8] \rightarrow [3, 8] \rightarrow [8] \rightarrow []$$

And the coins you get equals:

$$167 = 3 * 1 * 5 + 3 * 5 * 8 + 1 * 3 * 8 + 1 * 8 * 1.$$

Solution:

Let $OPT(l, r)$ be the maximum number of coins you can obtain from balloons $l, l+1, \ldots, r-1, r$. The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to $k-1$ and all the balloons from $k+1$ to r which are two sub problems. Therefore, we have the following recurrence relation:

$$OPT(l, r) = \max_{l \leq k \leq r} \{OPT(l, k-1) + OPT(l, k+1) + nums[k] * nums[l-1] * nums[r+1]\}$$

We have initial condition $OPT(l, r) = 0$ if $r < l$. For running time analysis, we in total have $O(n^2)$ and computation of each state takes $O(n)$ time. Therefore, the total time is $O(n^3)$.

Problem 7

Solve Kleinberg and Tardos, Chapter 6, Exercise 5.

Solution:

Let $Y_{i,k}$ denote the substring $y_i y_{i+1} \ldots y_k$. Let $Opt(k)$ denote the quality of an optimal segmentation of the substring $Y_{1,k}$. An optimal segmentation of this substring $Y_{1,k}$ will have quality equalling the quality last word (say $y_i \ldots y_k$) in the segmentation plus the quality of an optimal solution to the substring $Y_{1,i}$. Otherwise we could use an optimal solution to $Y_{1,i}$ to improve $Opt(k)$ which would lead to a contradiction.

$$Opt(k) = \max_{0 < i < k} Opt(i) + quality(Y_{i+1,k})$$

We can begin solving the above recurrence with the initial condition that $Opt(0) = 0$ and then go on to compute $Opt(k)$ for $k = 1, 2, \ldots, n$ keeping track of where the segmentation is done in each case. The segmentation corresponding to $Opt(n)$ is the solution and can be computed in $\Theta(n^2)$ time.

Problem 8

Solve Kleinberg and Tardos, Chapter 6, Exercise 6.

Solution:

Let $W = \{w_1, w_2, \ldots, w_n\}$ be the set of ordered words which we wish to print. In the optimal solution, if the first line contains k words, then the rest of the lines constitute an optimal solution for the sub problem with the set $\{w_{k+1}, \ldots, w_n\}$. Otherwise, by replacing with an optimal solution for the rest of the lines, we would get a solution that contradicts the optimality of the solution for the set $\{w_1, w_2, \ldots, w_n\}$.

Let $Opt(i)$ denote the sum of squares of slacks for the optimal solution with the words $\{w_i, \ldots, w_n\}$. Say we can put at most the first p words from w_i to w_n in a line, that is, $\sum_{t=i}^{p+i-1} c_t + p - 1 \leq L$ and $\sum_{t=1}^{p+i} w_t + p > L$. Suppose the first k words are put in the first line, then the number of extra space characters is

$$s(i,k) := L - k + 1 - \sum_{t=i}^{i+k-1} c_t$$

So we have the recurrence

$$Opt(i) = \begin{cases} 0 & \text{if } p \geq n - i + 1 \\ \min_{1 \leq k \leq p}\{(s(i,k))^2 + Opt(i+k)\} & \text{if } p < n - i + 1 \end{cases}$$

Trace back the value of k for which $Opt(i)$ is minimized to get the number of words to be printed on each line. We need to compute $Opt(i)$ for n different values of i. At each step p may be asymptotically as big as L. Thus the total running time is $O(nL)$.

Homework 7

Due Oct. 21st, 2021

Note This homework assignment covers dynamic programming from Kleinberg and Tardos.

1. Given a non-empty string s and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if s can be segmented into a space-separated sequence of one or more dictionary words. If s="algorithmdesign" and your dictionary contains "algorithm" and "design". Your algorithm should answer Yes as s can be segmented as "algorithmdesign".

   Let $s_{i,k}$ denote the substring $s_i s_{i+1} \dots s_k$. Let $Opt(k)$ denote whether the substring $s_{1,k}$ can be segmented using the words in the dictionary, namely $OPT(k) = 1$ if the segmentation is possible and 0 otherwise. A segmentation of this substring $s_{1,k}$ is possible if only the last word (say $s_i \dots s_k$) is in the dictionary the remaining substring $s_{1,i}$ can be segmented. Therefore,we have equation:

$$Opt(k) = \max_{0<i<k \text{ and } s_{i+1,k} \text{ is a word in the dictionary}} Opt(i)$$

   We can begin solving the above recurrence with the initial condition that $Opt(0) = 1$ and then go on to compute $Opt(k)$ for $k = 1, 2, \dots, n$. The answer corresponding to $Opt(n)$ is the solution and can be computed in $\Theta(n^2)$ time.

2. Solve Kleinberg and Tardos, Chapter 6, Exercise 10.

   (a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4 minutes are listed as follows (in time order):

- Machine A: 2, 1, 1, 200
- Machine B: 1, 1, 20, 100

The given algorithm will choose A then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

(b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i, you should not move in minute i, because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i, consider that if you are on machine A in minute i, you either (i) stay on machine A in minute $i-1$ or (ii) are in the process of moving from machine B to A in minute $i-1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute $i-1$, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\} :$$

Similarly, we get the recursive relation for $OPT_B(i)$:

$$OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\} :$$

The algorithm initializes $OPT_A(0) = 0, OPT_B(0) = 0, OPT_A(1) = a_1$ and $OPT_B(1) = b_1$. Then the algorithm can be written as follows:

$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;
for $i = 2, \cdots, n$ **do**
  $OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$;
  Record the action (either stay or move) in minute $i-1$ that achieves the maximum.
  $OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$;
  Record the action in minute $i-1$ that achieves the maximum.
end for
Return $\max\{OPT_A(n), OPT_B(n)\}$;
Track back through the arrays OPT_A and OPT_B by checking the action records from minute $n-1$ to minute 1 to recover the optimal solution.

It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.

3. Solve Kleinberg and Tardos, Chapter 6, Exercise 24.
   The basic idea is to ask: How should we gerrymander precincts 1 through j, for each j? To make this work, though, we have to keep track of a few extra things, by adding some variables. For brevity, we say that A-votes in a precinct are the voters for part A and B-voter are the votes for part B. We keep track of the following information about a partial solution.

   - How many precincts have been assigned to district 1 so far?
   - How many A-votes are in district 1 so far?
   - How many A-votes are in district 2 so far?

   So let $M[j, p, x, y] = true$ if it is possible to achieve at least x A-votes in distance 1 and y A-votes in district 2, while allocating p of the first j precincts to district 1. Now suppose precinct $j + 1$ has z A-votes. To compute $M[j+1, p, x, y]$, you either put precinct $j+1$ in district 1 (in which case you check the results of sub-problem $M[j, p-1, x-z, y]$) or in district 2 (in which case you check the results of sub-problem $M[j, p, x, y - z]$). Now to decide if there's a solution to the while problem, you scan the entire table at the end, looking for a value of $true$ in any entry from $M[n, n/2, x, y]$ where each of x and y is greater than $mn/4$. (Since each district gets $mn/2$ votes total).

   We can build this up in the order of increasing j, and each sub-problem takes constant time to compute, using the values of smaller sub-problems. Since there are n^2, m^2 sub-problems, the running time is $O(n^2m^2)$.

4. We initialize another matrix (dp) with the same dimensions as the original one initialized with all 0's.

   dp(i,j) represents the side length of the maximum square whose bottom right corner is the cell with index (i,j) in the original matrix.

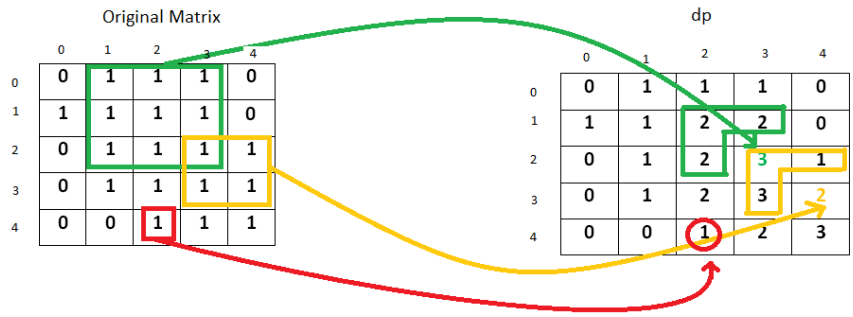
   Starting from index (0,0), for every 1 found in the original matrix, we update the value of the current element as

   $$dp(i, j) = \min\big(dp(i-1, j), dp(i-1, j-1), dp(i, j-1)\big) + 1.$$

   We also remember the size of the largest square found so far. In this way, we traverse the original matrix once and find out the required maximum size. This gives the side length of the square (say maxsqlen). The required result is the area $maxsqlen^2$

   An entry 2 at (1, 3) implies that we have a square of side 2 up to that index in the original matrix. Similarly, a 2 at (1, 2) and (2, 2) implies that a square of

side 2 exists up to that index in the original matrix. Now to make a square of side 3, only a single entry of 1 is pending at (2, 3). So, we enter a 3 corresponding to that position in the dp array.

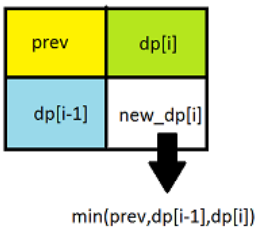


Now consider the case for the index $(3, 4)$. Here, the entries at index $(3, 3)$ and $(2, 3)$ imply that a square of side 3 is possible up to their indices. But, the entry 1 at index $(2, 4)$ indicates that a square of side 1 only can be formed up to its index. Therefore, while making an entry at the index $(3, 4)$, this element obstructs the formation of a square having a side larger than 2. Thus, the maximum sized square that can be formed up to this index is of size 2×2.

To reduce space complexity:

for calculating dp of i^{th} row we are using only the previous element and the $(i-1)^{th}$ row. Therefore, we don't need 2D dp matrix as 1D dp array will be sufficient for this.

Initially the dp array contains all 0's. As we scan the elements of the original matrix across a row, we keep on updating the dp array as per the equation $dp[j] = min(dp[j-1], dp[j], prev)$, where prev refers to the old dp[j-1]. For every row, we repeat the same process and update in the same dp array.



CSCI 570 Fall 2021
HW7 P5 Solution

1.4 Valid Palindrome if you can remove at most K characters

Given a string s and an integer k, find out if it can be transformed into a palindrome by removing at most k characters from it.

A) Let's first try to come up with a recursive solution *validPalindrome()*. Consider the bite sized question, "if I can remove $\leq$ K characters, can I make the string between indices i and j a palindrome?" Create a recursive solution that answers this bite sized question.

Solution.

bool validPalindrome(String s, int i, int j, int k):
if (k $<$ 0) return false // Return false if we removed too many characters.
if (i $\geq$ j) return true// Return true if we are looking at one character or an empty string.
if (s[i] == s[j]) return validPalindrome(s, i+1, j-1, k) // Look at next two characters if they match.
return helper(s, i+1, j, k-1) OR helper(s, i, j-1, k-1) // "remove" one of the chars and try again (dec. k)

B) Your *validPalindrome()* function likely takes 3 numbers as parameters: the starting index, ending index, and the remaining amount of characters you are allowed to remove. Think about how this solution would convert to a dynamic programming solution. You may be tempted to use a 3D array to save each result:

```
int answers[/*start index*/][/*end index*/][/*# of removable chars*/]
```
(2)

If you iterate over all three indices in order to fill up the array, the runtime and space complexity of your solution becomes O($n^2 \cdot k$). This is very not great. Let's use a different bite sized question: "HOW MANY characters do I have to remove in order to make the string between indices i and j a palindrome?" Create a recursive solution that answers this bite sized question.

Solution.

int validPalindrome(String s, int i, int j):

if (i $\geq$ j) return 0

// Return 0 if we are looking at one char or an empty string.

// Current answer = same answer as removing the two end characters.

if (s[i] == s[j]) return validPalindrome(s, i+1, j-1)

// Return 1 + minimum number of characters to make a palindrome with one of the end characters removed.

return 1 + MIN(helper(s, i+1, j), helper(s, i, j-1))

C) If you were to save the solutions in an array, the array would look like:

```
int answers[/*start index*/][/*end index*/]
```
(3)

This drastically cuts down on the space and time used by your solution. Now consider the order that you would need to fill up this array based on your recursive solution. Once you figure that out, write out your final solution. What is the new runtime and space complexity?

Solution.

ITERATIVE SOLUTION:

bool isValidPalindrome(String s, int k):

int answers[len(s)][len(s)]// Save answers here.

for i from len(s) - 2 to 0:// Iterate from end of the string to the beginning.

for j from i + 1 to len(s) - 1:// Start with smaller strings and end with larger ones.

// This is essentially just cut and paste from the recursive solution in part B.

if (s[i] == s[j]):

answers[i][j] = answers[i + 1][j - 1]

else:

answers[i][j] = 1 + MIN(answers[i][j - 1], answers[i + 1][j])

// Answer lies at the index that represents the string starting at 0 and ending at the end of the string.

return arr[0][len(s)] $\leq$ k

TIME AND SPACE COMPLEXITY

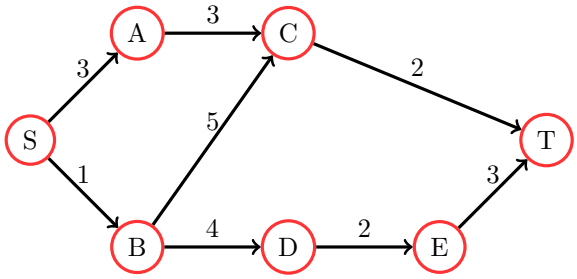
Time complexity is $O(n^2)$. Space complexity is $O(n^2)$ NOTE: There is a solution with a space complexity of $O(n)$, but we do not expect that answer.

CSCI 570 - Fall 2021 - HW 8 Solution

Due October 28th

Graded

1. You are given the following graph G. Each edge is labeled with the capacity of that edge.

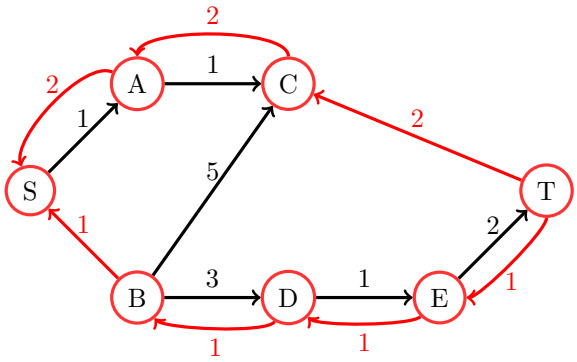


   (a) Draw the residual graph G_f using the Ford-Fulkerson algorithm corresponding to the max flow. You do not need to show all intermediate steps.

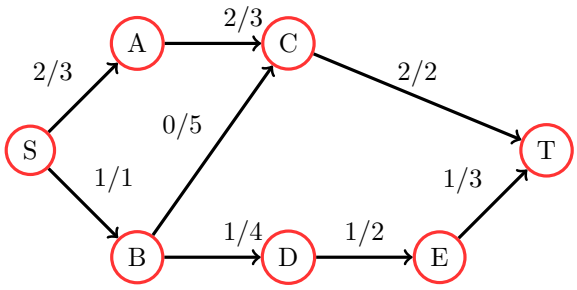
   (b) Find the max-flow value and a min-cut.

Solution:

Final Residual Graph



Final Flow graph (not required to be drawn)



The max flow is 2+1=3.

The min cut is {S, A, C} and {B, D, E, T}.

Rubric (15 pts)

- 10 pts: Correct Final Residual graph.
- 2 pts: Correct max flow value.
- 3 pts Correct min cut sets.

2. USC students return to in person classes after a year long interval. There are k in-person classes happening this semester, c_1, c_2, ..., c_k. Also there are n students, s_1, s_2, ..., s_n attending these k classes. A student can be enrolled in more than one in-person class and each in-person class consists of several students.

   (a) Each student s_j wants to sign up for a subset p_j of the k classes. Also, a student needs to sign up for at least m classes to be considered as a full time student. (Given: $p_j \geq m$) Each class c_i has capacity for at most q_i students. We as school administration want to find out if this is possible. Design an algorithm to determine whether or not all students can be enrolled as full time students. Prove the correctness of the algorithm.

   (b) If there exists a feasible solution to part (a) and all students register in exactly m classes, the student body needs a student representative from each class. But a given student cannot be a class representative for more than r (where $r < m$) classes which s/he is enrolled in. Design an algorithm to determine whether or not such a selection exists. Prove the correctness of the algorithm. (Hint: Use part (a) solution as starting point)

Solution: Part (a)

We can solve the problem by constructing a network flow graph and then running Ford–Fulkerson algorithm to get the max flow. If the max flow is equal to nm, then all students can be enrolled as full time students.

The setup of the graph is described below.

(a) Place the n students and k classes as vertices in a bipartite style graph. Connect each student s_j with all the classes in p_j using edges with capacity of 1.

(b) Place a sink vertex which is connected to all the classes (or alternatively students). Also, place a source vertex which is connected to all the students (or alternatively classes).

(c) Connect all s_j student vertices to the source (or sink) vertex with capacity of m.

(d) Connect all c_i class vertices to the sink (or source) vertex with capacity of q_i.

Claim: The enrollment is feasible if and only if there exists a max flow of nm.

Proof of Correctness

Forward Claim: The max flow is nm if the problem has a feasible solution.

Proof: If there exists a feasible solution for the problem, this means that all the in-person classes has at max of q_i students and each student was

able to sign up for at least m classes. By the law of conservation, the outward flow at the student vertex must be equal to the incoming flow to student vertex. Therefore, all the edges from source to the student vertices will be saturated with value of m, as each of the student was able to sign up for his/her classes. Similarly all the incoming flow at student vertex will be distributed to some of the out going edge with value of 1. This total flow from student vertices to the class vertex will be nm (again, by the law of conservation). And that is will eventually flow to the sink vertices with edges of max capacity q_i. Therefore, if the problem has a feasible solution the the max flow will be nm.

Backward Claim: The problem has a feasible solution if the max flow is nm.

Proof: If the problem has a max flow of nm. This means that along any path from source to sink vertex there is a student who was able to sign up for a particular class. Simply, we just have to follow the flow along a path.

Because the flow is max flow, this means that each of the student vertices receive a flow of m and the outward flow at each of the student vertices will be exactly m. Thus, the student will be able to sign up for at least m classes, which satisfies the constraint of the problem. Also, by the construction of network flow each class can accommodate at most q_i, which satisfies another constraint as the flow along the edge will be at most q_i (that is less than or equal to the capacity). Hence, the problem has a feasible selection if the max flow is nm.

Solution: Part (b)

We can solve the problem by starting from the solution from part (a). We will modify the constructed network flow graph slightly and then will run Ford–Fulkerson algorithm to get the max flow. If the max flow is equal to the number of classes k, then a selection exists otherwise no such selection exists.

The setup of the graph is described below.

(a) Use the same nodes as constructed in Part (a) of the solution.

(b) The course selections (edges between students and courses) for each student will be reduced to what is available in the solution from part (a) (our feasible solution). i.e. we will remove some edges between students and courses that they did not get to sign up for.

(c) The capacity on student to class edges will same remain same (i.e. 1).

(d) Assign 1 as the capacity from classes to sink (or alternatively source) vertex. (As there can only be 1 student representative from each class.)

(e) Assign r capacity for edges between source (or alternatively sink) and student vertices.

Claim: The selection is feasible if and only if there exists a max flow of k.

Proof of Correctness

Forward Claim: The max flow is k if the problem has a feasible selection.

Proof: If there exists a feasible selection for the problem, this means that all the in-person classes had a valid selection and each student was at max selected for r selections. By the law of conservation, the outward flow at the class vertex must be equal to the incoming flow to class vertex. Therefore, all the edges from sink to the class vertices will be saturated with value of 1, as each of the class had exactly 1 selection. This means that, k was the inward flow at class vertices. (law of conservation). Similarly, the number of selections per student will have the maximum value of r classes. And the total flow from source vertex to the student vertices will be k (again, by the law of conservation). Therefore, if the problem has a feasible solution the the max flow will be k.

Backward Claim: The problem has a feasible selection if the max flow is k.

Proof: If the problem has a max flow of k. This means that along any path from source to sink vertex there is a selection of student from a class. Simply, we just have to follow the flow along a path.

Because the flow is max flow, this means that all the class vertices receive a flow of 1 and the outward flow at the class vertices will be exactly 1 (and hence the flow is k). Thus, only 1 selection of student per class is made, which satisfies the constraint of the problem. Therefore, each class has selection of 1 student only. Also, by the construction of network flow each student can only be selected for at max r classes, which satisfies another constraint as the flow along the edge will at most r (that is less than or equal to the capacity). Hence, the problem has a feasible selection if the max flow is k.

Rubric Part A (20 pts)

- 12 pts: Correct construction of the the network flow graph.
- -4 pts: For each incorrect edge weight.
- 8 pts Accurate Proof of correctness.

Rubric Part B (20 pts)

- 12 pts: Correct construction of the the network flow graph.

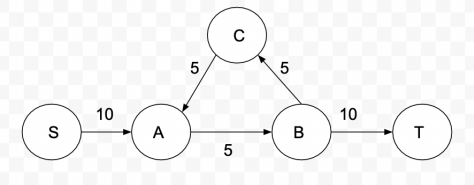
- -4 pts: For each incorrect edge weight.
- 8 pts Accurate Proof of correctness.

3. The following statements may or may not be correct. For each statement, if it is correct, provide a short proof of correctness. If it is incorrect, provide a counter-example to show that it is incorrect.

   (a) If each directed edge in a network is replaced with two directed edges in opposite directions with the same capacity and connecting the same vertices, then the value of the maximum flow remains unchanged.

   Solution:

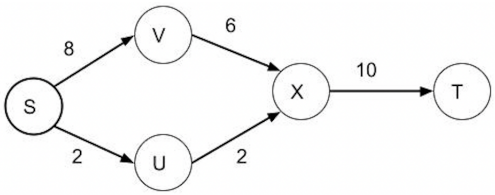
   False.
   Counter Example:



   (b) In the capacity-scaling algorithm, an edge e can be present in the residual graph $G_f(D)$ in at most one D-scaling phase.

   Solution:

   False.
   Counter Example:
   In the following example graph G, the edge (x, t) is present in two D-scaling phases.



   (c) If all edges are multiplied by a positive number 'k' then the min-cut remains unchanged.

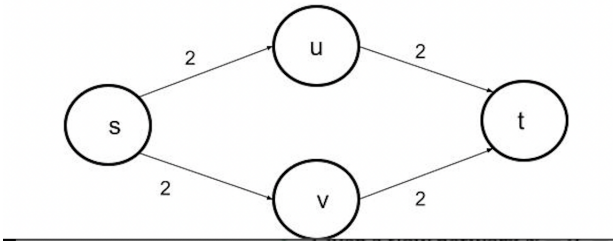
   Solution:

   True.
   The value of every cut gets multiplied by 'k', thus the relative-order of min-cut remains the same.

   (d) Suppose the maximum (s,t)-flow of some graph has value f. Now we increase the capacity of every edge by 1. Then the maximum (s,t)-flow in this modified graph will have value at most f + 1.

Solution:

False.
Counter-Example:
In the following graph, the maximum flow has value f = 4. Increasing the capacity of every edge by 1 causes the maximum flow in the modified graph to have value 6.



Rubric (12 pts)

- 3 For each sub-question (1pt: Correct True/False, 2Pts: Correct Proof or counter-example)

4. You are given a flow network with unit-capacity edges: it consists of a directed graph G = (V, E) with source s and sink t, and $u_e = 1$ for every edge e. You are also given a positive integer parameter k. The goal is delete k edges so as to reduce the maximum s − t flow in G by as much as possible. Give a polynomial-time algorithm to solve this problem. In other words, you should find a set of edges F ⊆ E so that $|F| = k$ and the maximum st flow in the graph G' = (V, E-F) is as small as possible. Give a polynomial-time algorithm to solve this problem.

   Follow up: If the edges have more than unit capacity, will your algorithm guarantee to have minimum possible max-flow?

   Solution:

   Algorithm:

   - Assume the value of max-flow of given flow network G (V, E) is g. By removing k edges, the resulting max-flow can never be less than g-k when $|E| \geq k$, since each edge has a capacity 1.
   - According to max-flow min-cut theorem, there is an s-t cut with g edges. If $g \leq k$, then remove all edges in that s-t cut, decreasing max-flow to 0 and disconnecting 's' and 't'. Else if, g > k, then remove 'k' edges from g, and create a new cut with $g - k$ edges.
   - In both the cases, whether the max-flow is 0 or $g - k$, the max-flow cannot be decreased any further thus giving us the maximum reduction in flow.
   - The algorithm has polynomial run-time. It takes polynomial time to compute the minimal-cut and linear time in 'k' to remove 'k' edges.
   - If all the edges don't have unit capacity, removing 'k' edges from min-cut in the above mentioned way does not guarantee to have the minimum possible max-flow.

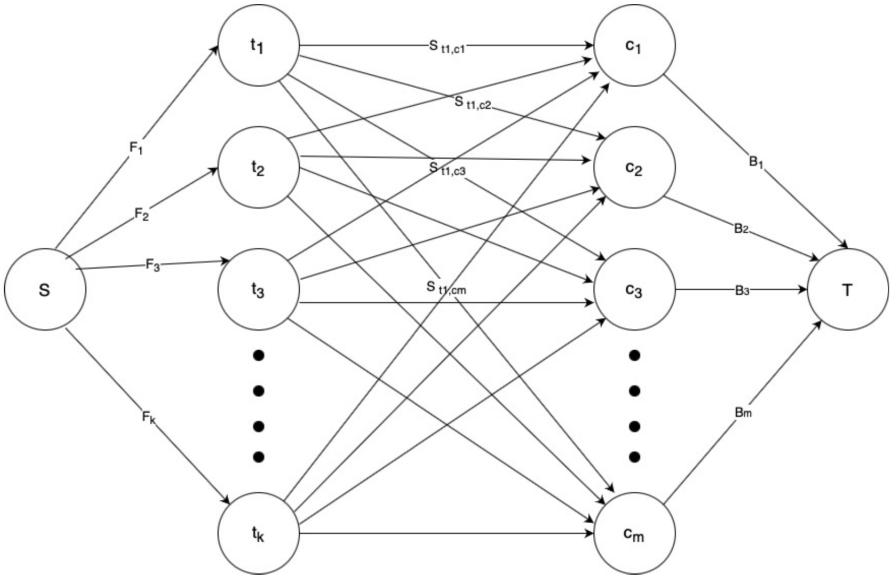
   Rubric (13 pts)

   - 10 pts: Correct Algorithm proposal with polynomial time.
   - 3 pts Correct answer for non-unit edges.

Ungraded

5. A tourist group needs to convert their USD into various international currencies. There are n tourists t_1, t_2, ..., t_n and m currencies c_1, c_2, ..., c_m. Tourist t_k has F_k Dollars to convert. For each currency c_j, the bank can convert at most B_j Dollars to c_j. tourist t_k is willing to trade as much as S_{kj} of his Dollars for currency c_j.(For example, a tourist with 1000 dollars might be willing to convert up to 200 of his USD for Rupees, up to 500 of his USD for Japanese's Yen, and up to 200 of his USD for Euros). Assuming that all tourists give their requests to the bank at the same time, design an algorithm that the bank can use to satisfy the requests (if it is possible).

Solution:



Flow Network Construction:

- Insert two new vertices, Source Node 'S' and Sink Node 'T'.
- Connect all the tourists t_i with Source Node 'S' assigning edges weight equal to F_i. Here, F_i stands for the maximum USD tourist t_i can exchange.
- Then, connect all tourists t_i to all the currencies available ie. c_j with each edge having weight S_{ij} which is the t_i tourist's limit to exchange his USD for a particular currency c_j.

- Connect currencies c_j with Sink Node 'T', with edge weights as B_j, which is the maximum limit of that particular currency c_j that the bank can convert to USD.
- In the graph constructed this way run Ford-Fulkerson or any Network Flow Algorithm from source S to sink T, the algorithm will return assignments, if it exists, that can solve the requests while following all the proposed constraints.

Claim:

The problem has a solution if and only if the max flow through the constructed graph is $\sum_k F_k$, ie. All the tourists are able to exchange their specified USD while following all the constraints.

Proof:

We have to prove the claim in Both Directions:

Proof for Forward Direction:

Let us assume there is a valid assignment and prove that max-flow = $\sum_k F_k$ exists.

Assume there is a valid assignment such that the bank can satisfy all the tourist's requests for conversion while following all the constraints. This means that all the outgoing edges from Source S to each tourist t_i is saturated. Thus, we get the max-flow in the graph from S to T is $\sum_k F_k$.

Proof in Backward Direction

Let us assume max-flow exists and thus prove that a valid assignment exists.

Let us assume we have max-flow in the above constructed graph. The max-flow = $\sum_k F_k$. From the above constructed Network flow graph we can see that, if max-flow = $\sum_k F_k$ exists, then it means all the edges are saturated, which in turn means all the tourist's were able to convert their currencies. Hence it is proved that a valid assignment exists if a max-flow = $\sum_k F_k$ exists.

6. Trip Trillionaire is planning to give potential buyers private showings of his estate, which can be modeled as a weighted, directed graph G containing locations V connected by one-way roads E. To save time, he decides to do k of these showings at the same time, but because they were supposed to be private, he doesn't want any of his clients to see each other as they are being driven through the estate.

   Trip has k grand entrances to his estate, $A = \{a_1, a_2, ..., a_k\} \subset V$. He wants to take each of his buyers on a path through G from their starting location a_i to some ending location in $B = \{b_1, b_2, ..., b_k\} \subset V$, where there are spectacular views of his private mountain range.

   Because of your prowess with algorithms, he hires you to help him plan his buyers' visits. His goal is if there exists a path for each buyer i from the entrance they take, a_i, to any ending location b_j such that no two paths share neither any edges nor any of the locations, and no two buyers end in the same location b_j. Prove the correctness of the algorithm.

   Solution

   We can solve the problem by constructing a network flow graph and then running Ford–Fulkerson algorithm to get the max flow. If the max flow is equal to k, then Trip Trillionaire can showcase his estate to his private buyers.

   Construct the network flow graph as below:

   (a) Create a graph with all the directed edges and vertices of his estate.

   (b) Split each vertex v into two vertices v_{in} and v_{out}, with a directed edge between them. All edges (u, v) now become (u, v_{in}) and all edges (v, w) now become (v_{out}, w). Assign the edge (v_{in}, v_{out}) capacity 1.

   (c) Assign a value of 1 to each of his road (or edge).

   (d) Create a source and a sink node.

   (e) Connect the source node to all the k entrances a_i. Assign these edges with a capacity of 1.

   (f) Connect the sink node to all ending locations b_j. Assign these edges with a capacity of 1.

   Claim: A path exists for each of buyer i where no two paths share any edges or any of the locations, if and only if the max flow is k.

   Proof of Correctness

   Forward Claim The max flow is k if a path exists for each of buyer i where no two paths share any edges or any of the locations.

   Proof: If there are k paths, this means that all the paths are distinct with distinct vertices. The max flow for our network will be k by construction. This is because, there are exactly k edges leaving the source vertex, with

all the weights as 1. This means that the max flow will be k from source to any starting location's incoming vertex. Now by the conservation of flow all the flow at a_i incoming vertices will eventually lead to b_j outgoing vertices that lead to sink vertex t.

Backward Claim There exists a path for each of buyer i where no two paths share any edges or any of the locations if the max flow is k.

Proof: If the problem has a max flow of k. To prove there are k distinct paths with distinct vertices, we just have to follow the flow along the path a_i to b_j. Along any path from source to sink vertex there is a unique path for that flow that starts from a_i's incoming vertex and eventually lead to b_j's outgoing vertex. Also, we split each vertex into 2 vertices and segregated the incoming and outgoing edges, which ensures that no two paths share any of the edge or any of the vertex. All of these flow will be a unique path where a buyer can be driven.

7. A vertex cover of an undirected graph $G = (V, E)$ is a subset of the vertices $A \subseteq V$ such that for every edge $e \in E$, at least one of its incident vertices is in A (that is every edge has at least one of its ends in A). Design an algorithm that takes a bipartite graph G' and a positive integer k, and decides if G' has a vertex cover of size at most k.

Solution

Partition the vertices of the graph into left vertices and right vertices, that is $V = L \cup R, L \cap R = \phi$ and every edge has one end in L and the other in R.

We construct a network $\bar{G'}$ as follows. Add a source s, a sink t and the following edges;

(i) directed edges of capacity 1 from s to every vertex in L ,

(ii) directed edges of capacity 1 from every vertex in R to t

(iii)make every edge in $\bar{G'}$ directed from L to R and of infinite capacity.

Let $(S, \bar{S'})$ be a cut of $\bar{G'}$ of finite capacity. Without loss of generality assume that $s \in S$. We claim that we can construct a vertex cover of G' of size equal to the capacity of the cut.

Define $A := (L - S) \cup (R \cap S)$.

Edges that are incident on $L - S$ are covered by $L - S$. The only edges left to check are the ones with an end in $L \cap S$. The edges that are incident on $L \cap S$ have their other end in S and are hence covered by $R \cap S$. Thus A is indeed a vertex cover.

Edges between L and R have infinite capacity and hence cannot be in the cut. The only edges crossing the cut are either incident on s or incident on t. The number of edges of the form (s, u) where $u \in \bar{S}$ equals $|L - S|$. The number of edges of the form (v, t) where $v \in S$ is $R \cap S$. Hence the size of A equals the capacity of the cut $(S, \bar{S})$ and our claim is true.

We now claim the converse. That is, given a vertex cover A of G', we can construct a cut of $\bar{G'}$ of capacity equal to the size of A.

Define the cut $S := \{s\} \cup (L - A) \cup (R \cap A)$.

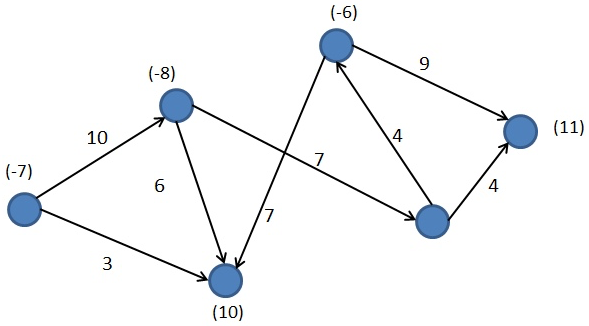
For an edge $(u, v) \in E$, if $u \in S$ then $v \in S$ (since A is a vertex cover). Thus edges in E cannot cross the cut.

The number of edges of the form (s, u) where $u \in \bar{S'}$ equals $|L \cap A|$. The number of edges of the form (v, t) where $v \in S$ equals $|R \cap A|$. Hence the capacity of the cut is $|L \cap A| + |R \cap A|$ which is exactly the size of A. Hence, the converse follows.

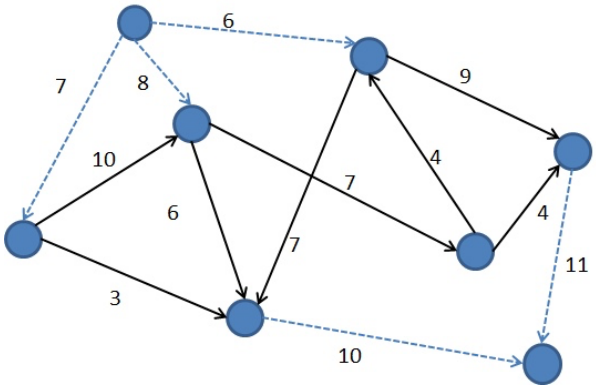
We have thus proven that the size of the min-vertex cover of G' equals the capacity of the min-cut of $\bar{G'}$. The capacity of the min-cut of $\bar{G'}$ can be computed using the Ford-Fulkerson algorithm (say the capacity is m). If $k < m$, output NO, otherwise output YES.

Homework 9

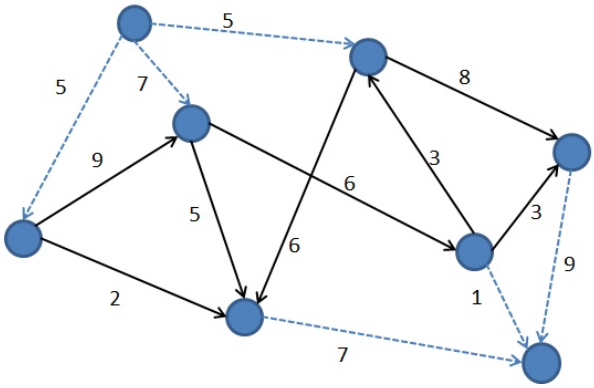
Problem 1. (10pts)
The following graph G is an instance of a circulation problem with demands. The edge weights represent capacities and the node weights (in parentheses) represent demands. A negative demand implies source.



a) Transform this graph into an instance of max-flow problem.
Solution: (5pts, no partial credits)

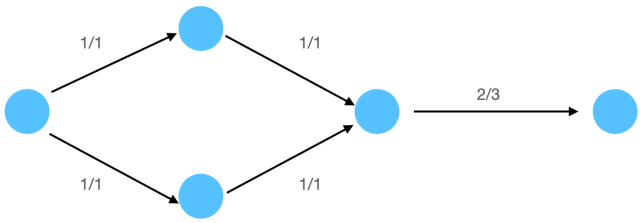


b) Now, assume that each edge of G has a constraint of lower bound of 1 unit, i.e., one unit must flow along all edges. Find the new instance of max-flow problem that includes the lower bound information. (Find G' in lecture slides)
Solution: (5pts, no partial credits)

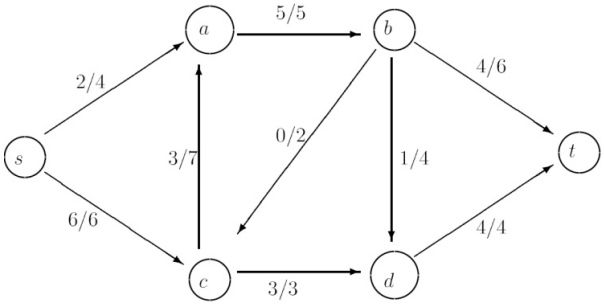


Problem 2. (9pts)
Determine if these statements are correct.

a) In a flow network, if the capacity of every edge is odd, then there is a maximum flow in which the flow on each edge is odd. (3pts)
Solution: False
Counter example:



b) The following flow is a maximal flow. (3pts)
Note: The notation a/b describes a units of flow on an edge of capacity b.



Solution: True

c) Given the min-cut, we can find the value of max flow in $O(|E|)$. (3pts)
Solution: True, Value of max flow = capacity of min-cut (iterate over all edges of the min-cut)

Problem 3. (11pts)
A company sells k different products, and it maintains a database which stores which customers have bought which products recently. We want to send a survey to a subset of n customers. We will tailor each survey so it is appropriate for the particular customer it is sent to. Here are some guidelines that we want to satisfy:

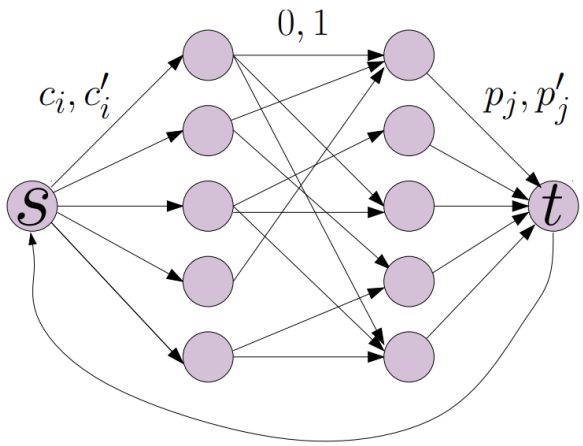
- The survey sent to a customer will ask questions only about the products this customer has purchased.
- We want to get as much information as possible, but do not want to annoy the customer by asking too many questions. (Otherwise, they will simply not respond.) Based on our knowledge of how many products customer i has purchased, and easily they are annoyed, our marketing people have come up with two bounds $0 \leq c_i \leq c'_i$. We will ask the ith customer about at least c_i products they bought, but (to avoid annoying them) at most c'_i products.
- Again, our marketing people know that we want more information about some products (e.g., new releases) and less about others. To get a balanced amount of information about each product, for the jth product we have two bounds $0 \leq p_j \leq p'_j$, and we will ask at least p_j customers about this product and at most p'_j customers.

The problem is how to assign questions to consumers, so that we get all the information we want to get, and every consumer is being asked a valid number of questions.

Solution:

First, we build a bipartite graph having consumers on one side, and products on the other side. Next, we insert the edge between consumer i and product j if the product was used by this consumer. The capacity of this edge is going to be 1. Intuitively, we are going to compute a flow in this network which is going to be an integer number. As such, every edge would be assigned either 0 or 1, where 1 is interpreted as asking the consumer about this product.

The next step, is to connect a source s to all the consumers, where the edge (s, i) has lower bound c_i and upper bound c'_i. Similarly, we connect all the products to the destination t, where (j, t) has lower bound p_j and upper bound p'_j. We would like to compute a flow from s to t in this network that comply with the constraints. However, we only know how to compute a circulation on such a network. To overcome this, we create an edge with infinite capacity between t and s. Now, we are only looking for a valid circulation in the resulting graph G which complies with the aforementioned constraints. See figure below for an example of G.



Given a circulation f in G it is straightforward to interpret it as a survey design (i.e., all middle edges with flow 1 are questions to be asked in the survey). Similarly, one can verify that given a valid survey, it can be interpreted as a valid circulation in G. Thus, computing circulation in G indeed solves our problem.

Rubric:

- Build the correct graph (6pts), -1 for each missing node/edge, or incorrect capacity

- Reduce to circulation problem and make correct conclusion (5pts). -3 if only computing max-flow. Deduct points for other incorrect statements.

Problem 4. (10pts)
Kleinberg and Tardos, Chapter 7, Exercise 7
Solution:

We build the following flow network. There is a node v_i for each client i, a node w_j for each base station j, and an edge (v_i, w_j) of capacity 1 if client i is within range of base station j. We then connect a super-source s to each of the client nodes by an edge of capacity 1, and we connect each of the base station nodes to a super-sink t by an edge of capacity L.

We claim that there is a feasible way to connect all clients to base stations if and only if there is an $s-t$ flow of value n. If there is a feasible connection, then we send one unit of flow from s to t along each of the paths s, v_i, w_j, t, where client i is connected to base station j. This does not violate the capacity conditions, in particular on the edges (w_j, t), due to the load constraints. Conversely, if there is a flow of value n, then there is one with integer values. We connect client i to base station j if the edge (v_i, w_j) carries one unit of flow, and we observe that the capacity condition ensures that no base station is overloaded.

The running is the time required to solve a max-flow problem on a graph with $O(n+k)$ nodes and $O(nk)$ edges.

Rubric:

- 4pts for constructing correct graph.
- 4pts for proving that a feasible connection is corresponding to a $s-t$ flow.
- 2pts for polynomial time complexity.

Problem 5. (practice)
Kleinberg and Tardos, Chapter 7, Exercise 9
Solution:

We build the following flow network. There is a node v_i for each patient i, a node w_j for each hospital j, and an edge (v_i, w_j) of capacity 1 if patient i is within a half hour drive of hospital j. We then connect a super-source s to each of the patient nodes by an edge of capacity 1, and we connect each of the hospital nodes to a super-sink t by an edge of capacity $\lceil n/k \rceil$.

We claim that there is a feasible way to send all patients to hospitals if and only if there is an $s-t$ flow of value n. If there is a feasible way to send patients, then we send one unit of flow from s to t along each of the paths (s, v_i, w_j, t), where patient i is sent to hospital j. This does not violate the capacity conditions on the edges (w_j, t), due to the load constraints. Conversely, if there is a flow of value n, then there is one with integer values. We send patient i to hospital j if the edge (v_i, w_j) carries one unit of flow, and we observe that the capacity condition ensures that no hospital is overloaded. The running is the time required to solve a max-flow problem on a graph with $O(n+k)$ nodes and $O(nk)$ edges.