1) we are going to use kadane's Algorithm for combination of all of the columns taken in pairs.

Kadane's Algorithm:

Input : Array A[1...n].

Output : • start & finish indices of max_sum subarray
• max_sum of these values.

Algorithm: set sum := 0
max_sum := INF
temp_start := 0 , temp_sum := 0
loop counter := 0
while (loop counter is less than length(A))
temp_sum = temp_sum + A[loop counter].
if (temp_sum < 0)
set the value of temp_sum := 0
" " " " temp_start := loop counter + 1.
if (temp sum > max_sum).
max_sum = temp_sum
start = temp_start
finish = loop counter.
loop counter ++
if finish is set
END.
else
set max_sum := A[0].
set start & finish := 0
while (loop counter < length of A).
if (A[loop_counter] > max_sum)
max_sum = A[loop counter]
start = finish = loop counter
loop counter ++
END

complexity of kadane's Algo := O(n)
as only 2 pass of ~~elems~~ Array is done

Algorithm: input : 2D array of size n×n (A)
output : i, i' & j, j' (as defined in ques^n

~~Set~~ define left, right, ~~co~~ row_counter, ~~variables~~ max_sum = 0 → -∞
define array temp of size [1×n].
left := 0
while (left < n)
    initialize temp with 0s.
    right = left
    while (right < n)
        ~~for~~ row counter = 0
        while (row counter < n)
            temp [row counter] = temp[row counter] + A[row counter, right].
            row counter = row counter + 1
        call kadane Algorithm of temp that returns start & finish values for max_sum subarray indices
        ~~kadane~~ temp
        if (max_sum - ~~kadane~~ temp > max_sum)
            Set max_sum as max_sum temp.
            Set i as start
            Set j as right
            Set i' as left
            Set j' as finish
        ~~row counter = row counter + 1~~
        right = right + 1
    left = left + 1

called by n^2 times

O(n)

O(n)

Time complexity :- kadane's algorithm is called in ~~loop~~ 2 nested loop of size n ⇒ n^2 times.

⇒ Algorithm's time complexity = $n^2 O(n)$
⇒ $O(n^3)$

Proof of correctness:

Solution of problem is iterative in nature hence we can give proof of correctness by using loop invariant- loop invariant proof involves i) initialization cond is T on start
ii) maintanance middle
iii) termination. end of loop

i) Initialization: (trivial soln) when loop starts it select i = 1st row and 1st column and max_sum ~~ele~~ returned as 1st element ~~value~~

ii) Maintainance: when loop move forward. we got the temp sum of rectangular subarray from [i, ..i' & j, ..j'] which is value obtained by start, left, right and finish of main Algorithm. which used kadane's subroutine to set- start & finish values if the subarray in process has sum more than max-sum boundary of soln is updated.

iii) Termination: same as maintainance the condition of having max-sum correct and proper boundary of subarray having that sum will be correct too.

all three phases of loop invariant proof are consistent.

2) Sequence of n days. $S_1, S_2 \ldots S_n$

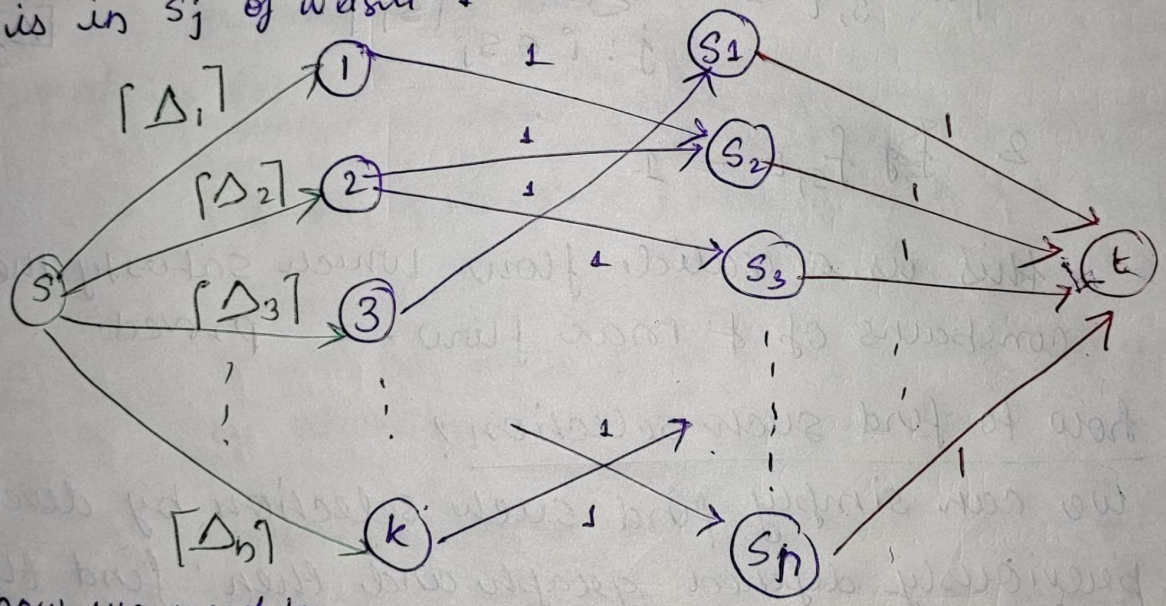
of $\{1, 2, \ldots k\}$

need to pick exactly one person to S_i st-

for a person j $\Delta_j = \sum_{i : j \in S_i} \frac{1}{|S_i|}$

good selection ⇒ person picked at most $\lceil \Delta_j \rceil$ times

we can convert this problem to network flow problem. ~~with person 1 to k~~ with person i st $i \in [1, k]$. in left & set S_j such that $\forall j \in [1, n]$ & draw edge from left to right if i is in S_j of weight 1



now we need to ~~this step will~~ ensure that ~~each~~ only one person is selected at a given day. which can be easily done by connecting S_i to sink t with capacity 1 (done in red.

next step is to ensure that each person could only be selected $\lceil \Delta_i \rceil$ times which can easily be done by limiting flow value of i^{th} vertex as $\lceil \Delta_i \rceil$.

i) such selection is always possible:

Claim: such select^{n} is possible ~~o~~ if we can find the max-flow of value d.

- since encoming edges to t are n with flow value of atmost 1 $\Rightarrow$ max flow $<= n$.
- we can easily see when this value coned be acheived.

$$\text{flow}_{i,s_j} = \frac{1}{|S_j|}$$

$$\text{flow}_{s,i} = \sum_{j : i \in S_j} \frac{1}{|S_j|}$$ which is less than $[\Delta_j]$

& ~~Σ~~ $f_{s_j t} = 1$.

this is a valid flow which satesty the constains of ~~&~~ max flow. proved.

ii) how to find such selection?

we can simply find such selection by developing previously defened graph and then find the max flow using ford-fulkerson algorithm as discussed in class.

Proof of correctness:

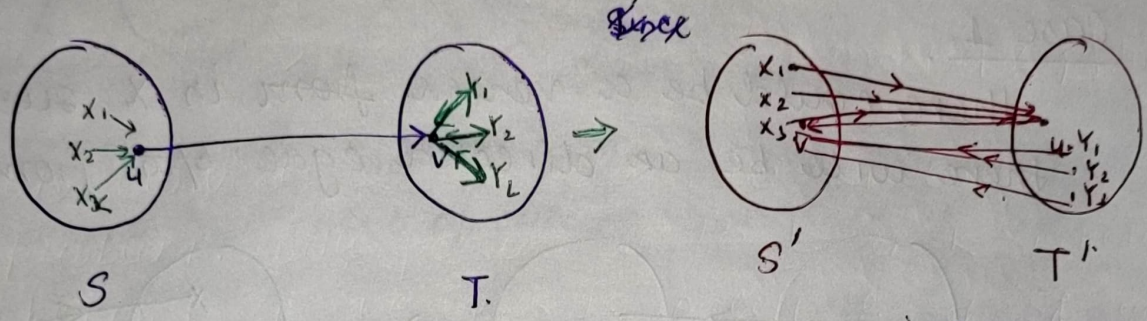
we have given the value of flow for each edge and such flow will always possible of all values of i & j and S.

3) let u, v be directed edge in G.

Given: There is a min s-t cut st $u \in S$ & $v \in T$.

To prove: no cut S' T' is possible where $u \in T'$ & $v \in S'$

let's have S-T cut which has u in S & v in T



Claim: moving u to T and v to S will increase the value of cut

proof: • let's have some vertices in S. $X_1, X_2, \dots X_k$ such that edges x to u are coming
Since $u \to v$ is in cut ⇒ there will be atleast 1 ~~+pos~~ such x is possible ⇒ $k \geq 1$.

• Similarly we have vertices in T $Y_1, Y_2, \dots Y_L$ which has edge from v to Y are going.
by above mension reason $L \geq 1$.

• when we move v to S the value of cut will be L (as now all these edges will be in between S & T.

• Similarly when we move u to v cut-value will be increased by k.

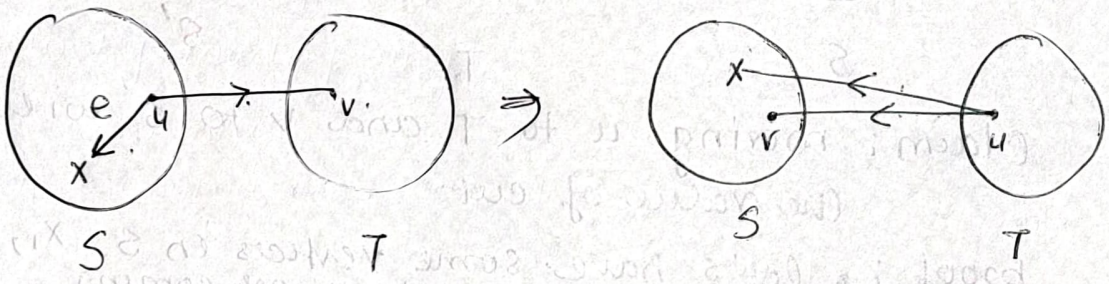
• Since u,v edge is now going from t to S it will leave no effect on new cut-

⇒ value of new cut will be $L+k$.
as we know L & $k \geq 1$ ⇒ new cut value is increased by atleast 2

Since we have not given any constraint over S' & T' we need to prove that moving any vertices between s & t will never able to undo the increase ~~&~~ caused by moving u & v to T & S.

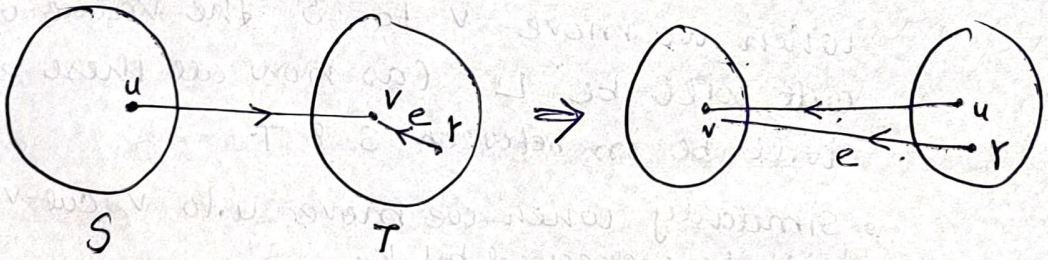
Case 1.

There could be a vertex ~~from~~ in X such that there will be a directed edge b/w from u to X.



When we move u to T this edge e will be going from T to S which ~~again~~ will never change the value of cut.

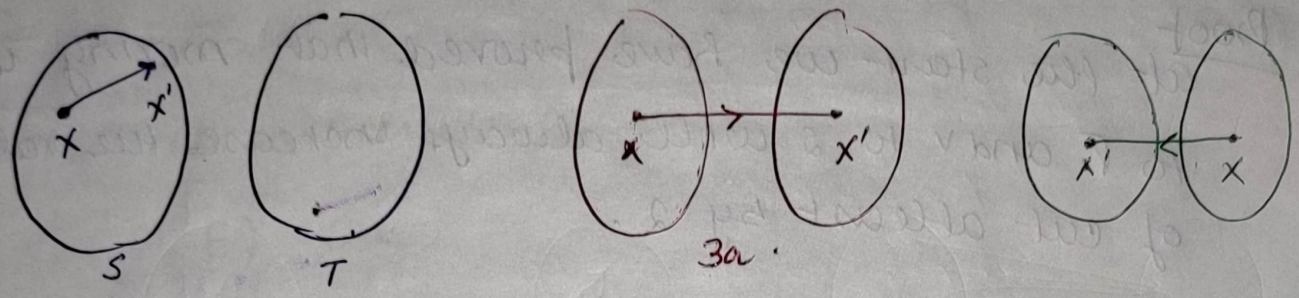
Case 2 There could be a vertex r in T such that there will be a directed edge from r to v.



by similar argument from case 1 we can say that this edge also not change the value of cut.

Now we are going to look at edges ~~between~~ in same set.

Case-3 : we have two vertex in S which is x & x' & we have a directed edge b/w them.

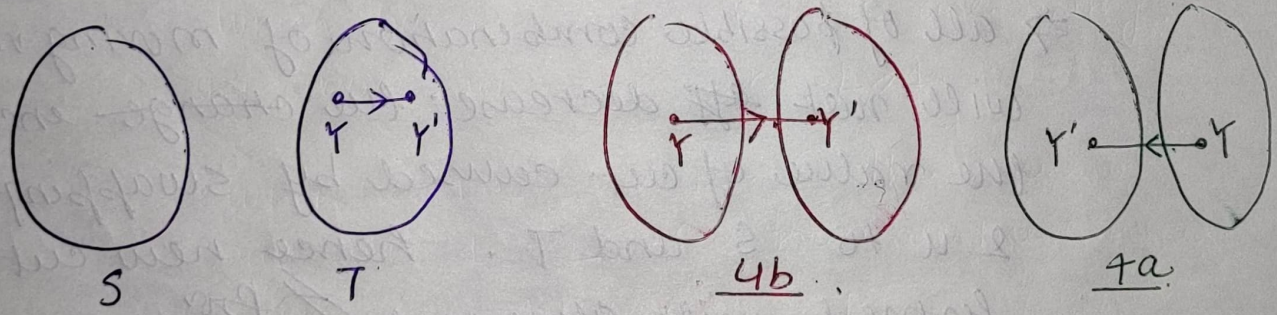


Case3a if we try to move x' to T this operation will cause the value of cut to increase by one.

Case 3b if we move x to T ⇒ no change in value of cut as this edge is going from T to S.

~~Case 3c if we move both x → x' to T we can~~

Case 4: we have two vertex in T which are Y Y' & we have directed edge b/w them.



Case 4a: if we move Y' to S ⇒ this operation include one more edge from T to S leaving no effect on cut value.

Case 4b: if we move Y to S ⇒ this operation include this edge and increase the value of cut by atleast 1.

Claim: no ^{min} cut is possible if we move u to T & v to S.

Proof

At the start we have proved that moving u to T and v to S will always increase the value of cut atleast by 2.

We have proved case by case that this ~~eff~~ increase in cut will not be able to undo by moving any other vertex of S & T to each other

to summarize. case 1, case 2, case 3b and case 4a will not affect the value of cut at all.

cases 3a & 4b are always increase the value of cut atleast by one

⇒ all of possible combination of moving vertices will not ~~eff~~ decrease the ~~change~~ increased the value of cut caused by swapping v & u to S and T. hence new cut formed will always has ~~Pro~~ value greater than min cut

Proved.

4) G- undirect graph w/ s & t two special vertices

~~Given~~ To find: effecient Algorithm to find max number of node disjoint paths in G.

our problem is a little bit complex when we compare it with edge disjoint path problem as discussed in class. so we are going to change graph G to H and solve the edge connectivity on H and number of edge disjoint path in H will be same as number of vertex disjoint path in G.

Algorithm. input-: G (V, E).
output : number of path (node disjoint)

procedure:
- for each edge E in G. make another graph G' with V and 2 directed edges in opposite dirn. in place of one edge in G. (except edges belong to s & t)
- construct H from G' by splitting V to Vin and Vout (except s & t) and put a directed edge from Vin to Vout. all incoming edges to V in G' will be incident in Vin in H and all outgoing edges will leave from Vout.
- calculate the edge disjoint-path in H as discussed in class. (take weight of each edge as 1 and find the max flow using ford algorithm)

end.

Complexity : complexity of ~~ed~~ calculating edge disjoint paths is $O(VE)$

in H we have ~~vertices~~ $|V|_H = |V|_G \times 2 = O(V)$

$$|E|_H = (|E|_G \times 2) + |V|_G$$

$$\approx |E|_G + O(E)$$

hence changing the G to H will only affect the complexity in constant time increase and the complexity of node disjoint paths will be $O(VE)$

Proof of correctness :

Claim : node disjoint path count in G is same as ~~node~~ edge disjoint path count in H.

Proof : G is undirected hence we can move in both direction for any edge but not in same time.

G $\longrightarrow$ G' $\longrightarrow$ H

(V, E) undirected. — directed edges 2|E| — direct edge b/w Vin and Vout

by taking intution from flow problem that can solve the problem of edge disjoint path count we modified the G such that at a time only one ~~edge~~ path can ~~travel~~ have a node

this is acheived by having edge b/w Vin & Vout

~~to~~ since max flow can't be more than 1 hence if ~~only~~ one path ~~can have~~. ~~an~~ have a vertex (V) in G $\Rightarrow$ ($V_{in} \rightarrow V_{out}$ in H) $\Rightarrow$ no other path can have this node.

Hence we can say that node disjoint paths in G are same as node disjoint paths in H.

proof of correctness of edge disjoint path is given in class using flow problem.

Alternative algorithm.

Menger's Theorem:

by vertex connectivity statement of menger's theorem let G be a finite undirected graph, x & y are 2 special vertices (non adjacent) then the size of the min vertex cut of x & y (the min ~~#~~ number of vertices removed to disconnect x & y) is equal to the maximal number of pairwise internally vertex disjoint paths.

we can directly find out the min vertex cut by ~~karger's min cut~~ and number of vertices removed will be our Answer.

5)

G: undirected graph.

S: subset of vertices $e(S)$: # edges has both endpoints

α: rational number.

To find: if there exist a subset S s.t. $\frac{e(S)}{|S|} > \alpha$.

$G(V, E)$ Given.

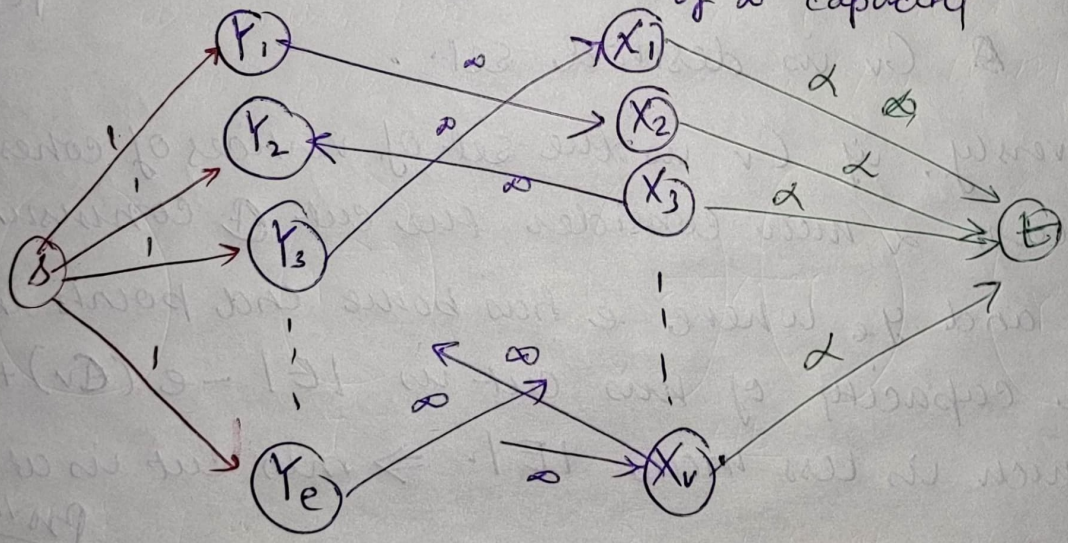
we need to relate edges to vertices in such a way that it could model a flow problem.

we will going to make graph H with two special set of vertices called X_v and Y_e

X_v: vertex for every vertex in G.

Y_e: vertex for every ~~ver~~edge in G.

we put the edges y_e to x_u & y_e to x_v in $(e = u, v)$ is of ∞ capacity



we then insert special source s to bound the value of flow. and connect it to Y_e given in red.

after that we connect the X_i to special vertex t with capacity α (given in green)

Claim: There is a set S of cohesiveness at least α iff there is an s-t cut in H of capacity at most $|E|$.

Proof: Let C be an s-t cut in H, ~~and A_e denote the vertices of type y_e in A and A_v denote the~~

A_e : vertices of type y_e in C

A_v : vertices of type x_v in C

if y_e is C $\Rightarrow$ $e(v,v)$ in C $\Rightarrow$ x_v & x_u in C

$\Rightarrow e(C_v) \geq |C_e|$.

Now capacity of this cut is $|E| - |C_e| + |C_v|\alpha$ which is greater than $|E| - e(C_v) + |C_v|\alpha$.

$\Rightarrow$ if min cut is atmost $|E|$ then $\frac{e(C_v)}{|C_v|} > \alpha$

$\Rightarrow$ C_v is desired set.

conversly, if C_v is the set of vertices of cohesiveness atleast α then consider the cut C consisting of C_v and y_e where e has bothe end points in C_v. the capacity of this cut is $|E| - e(C_v) + |C_v|\alpha$ which is less than $|E|$. $\Rightarrow$ min cut is at most $|E|$

Proved.

finally we can find the set of max cohesiveness by binary search on α. note that cohesiveness of any set is a factor of the form $\frac{B}{n!}$ where $B \in [0, n^{n^2}]$

$\Rightarrow$ we can perform binary search on B

time complexity binary search of B $\Rightarrow$ $\log(B) \Rightarrow \log(n!)$

$\Rightarrow O(n \log n)$