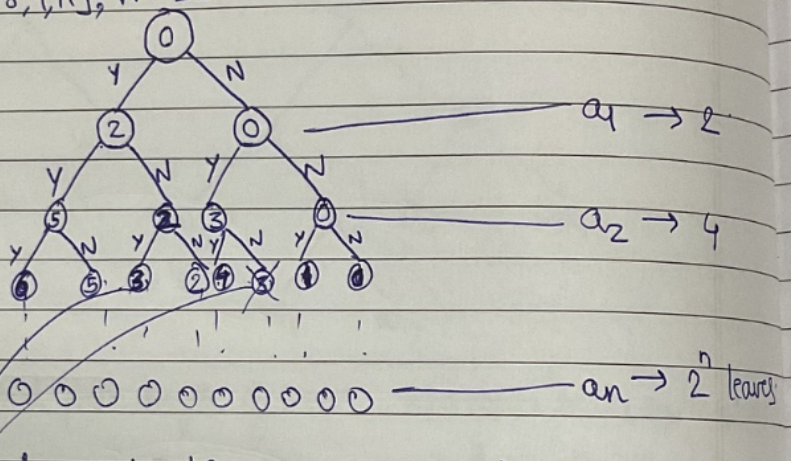
Dynamic Programming & Shortest Paths

→ Subset Sum:

$U = \{a_1, a_2, \ldots, a_n\}$ $a \in \mathbb{Z}^+$, $W \in \mathbb{Z}^+$

Is there a subset $S \subseteq U$ s.t. $\sum_{a_i \in S} a_i = W$

Ex: $U = \{2, 3, 1, 6, 8, 9, 11\}$, $W = 22$

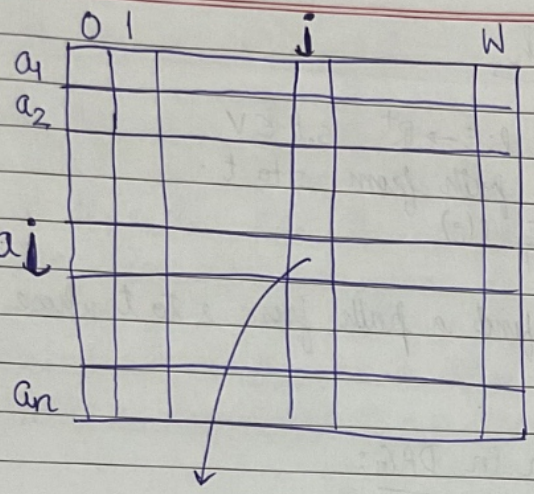


Same value so, branching only one of them is enough + We will also prune the branches that get value > W

Now we can say that

no. of nodes at any level ≤ W

n levels ⇒ O(nW)



$X(i,j) = 1$ if there exists a subset of $\{a_1 \cdots a_i\}$ whose sum is j
else it is 0.

Now; $X(i,j) = 1$

$\therefore X(i,j) = X(i-1, j) \vee X(i-1, j-a_i)$

$X(i-1, j)$: if a_i is not included in the subset

$X(i-1, j-a_i)$: if a_i is included in the subset.

This is how we can fill the table by either row by row or column by column.

And by looking at the value $X(n, W)$ we can know that a soln exists or not.

Then if $X(n, W) = 1$ then we can just backtrack by tracing the path formed by $X(i-1, j)$ or $X(i-1, j-a_i)$ or a combination of both.

→ Shortest Path

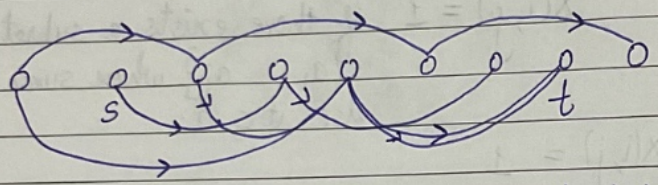
$G = (V, E)$ $l: E \to \mathbb{R}^+$ $s, t \in V$

Let P be a path from s to t.

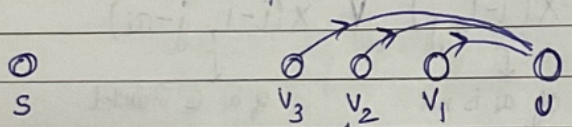
$l(P) = \sum_{e \in P} l(e)$

We want to find a path from s to t whose $l(P)$ is minimum.

* Shortest Path in DAG:



~~So, shortest path from s to v = shortest path from s to v_1~~



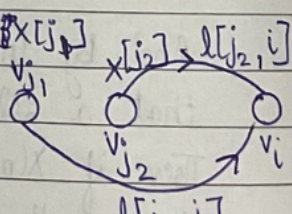
$$sp(s,v) = \min\left(\begin{bmatrix}sp(s,v_1)\\ +l(v_1,v)\end{bmatrix}, \begin{bmatrix}sp(s,v_2)\\ +l(v_2,v)\end{bmatrix}, \begin{bmatrix}sp(s,v_3)\\ +l(v_3,v)\end{bmatrix}\right)$$

Now lets consider

$v_1, v_2, \dots, v_n$

$\|$

s



$X[i] = l(sp(s, v_i))$

$$\therefore \boxed{X[i] = \min_{j:(j,i)\in E} \{X[j] + l(j,i)\}}$$

If no path from s to v_k the $X[k] = \infty$.
~~Initialize~~ with this.

So, Time Complexity = Time for Topo sort + Time for path

$O(m+n)$ + $O(m)$

$$\boxed{T.C. = O(m+n)}$$

* Shortest Pa

If we kno

So; Claim:

Thus, we

This is

s →

Now;

For

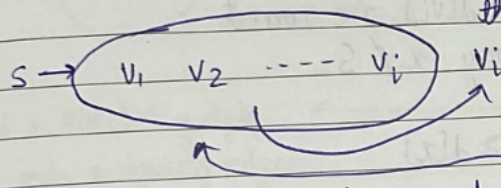
* Shortest Path in General Graph:

If we know the order of distance of vertices from s; ~~then~~
$d[v_1] \leq d[v_2] \leq \cdots \leq d[v_n]$ & $d[s] = 0$.

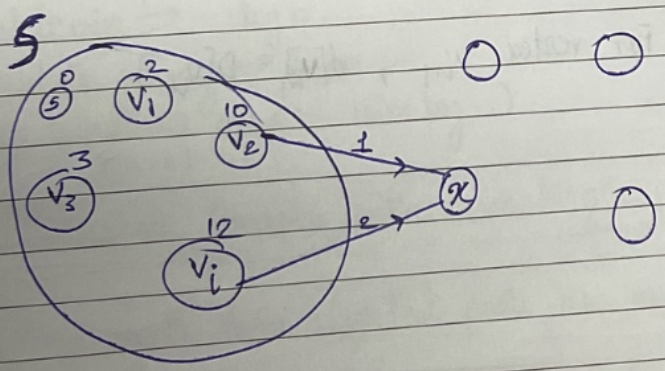
So; Claim: If there are edges that go from v_j to v_i s.t. $i \leq j$ then those edges will never be used in finding shortest path from s to and other vertex.

Thus, we can now treat this as a DAG.

This is **Djikstra's Algorithm**: If we know shortest paths from s to vertices $v_1, v_2, \cdots, v_i$ such that $d[v_1] \leq d[v_2] \leq \cdots \leq d[v_i] \leq d[v_{i+1}]$ then we can find $sp[(s, v_{i+1})]$.



Now; how do we find v_{i+1} to add it to this?



For every vertex outside ~~we will find~~; $x \notin S$

$$D[x] = \min_{w \in S} \left(d[w] + l(w, x) \right)$$

And the vertex with the smallest $D[x]$.

So,

$$v_{i+1} = \min_{x \notin S} D[x] \Rightarrow$$ add v_{i+1} to S and set $d[v_{i+1}] = D[v_{i+1}]$ and repeat until no vertex rem".

Algo: Initialize $S \leftarrow \{s\}$, $d[s] = 0$.
repeat $n-1$ times {
  for $\forall\, x \notin S$ {
    $D[x] = \min_{w \in S} (d[w] + l(w,x))$ }
  $v_{i+1} = \min_{x \notin S} D[x]$

  $S += v_{i+1}$; $d[v_{i+1}] = D[v_{i+1}]$.
}

Proof of Correctness:

Suppose $d[v_1] \leq d[v_2] \leq \dots \leq d[v_i]$ are correct.
and $d[x] \geq d[v_i]$ for any $x \notin S$.

Obs^n : ① For $x \notin S$, $D[x] \geq d[x]$

length of a certain $s \to x$ path.

② For vertex v_{i+1} ; $d[v_{i+1}] = D[v_{i+1}]$

→ Djikstra's

G = (V,
Find th
lo

S

This a dir
so only
are cor

So, n
m
So

→ Shortes

If

Ass

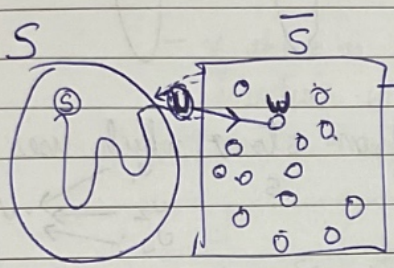
→ Djikstra's Algorithm for Single Source Shortest Path

$G = (V, E)$; $s \in V$, $l(e) \in R^+$

Find the shortest path from s to all vertices in V.

label[v] = $sp(s,v)$ if $v \in S$

$\geq sp(s,v)$ if $v \notin S$.



→ Vertices of $\bar{S}$ sit in a min-Heap H.

$\forall v \in V$ label[v] = ∞ ; label[s] = 0.

H.create(label)

while !H.empty do {

u = H.deletemin()

$\forall w \in$ adjacent(u) do {

label[w] = min(label[w], label[v] $\overset{+}{}$ l(u,w))

}

}

This a directed graph so only outgoing edges are considered adj.

So, n deletemin → log n

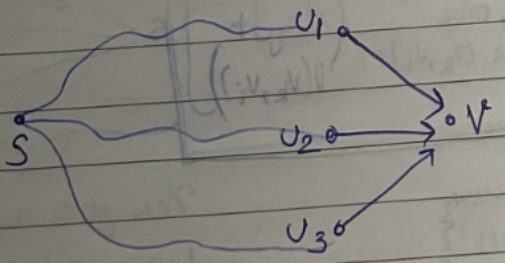
m update priority → log n

So for Djikstra's algo :- $O(m \log n)$ $\forall (m > n)$

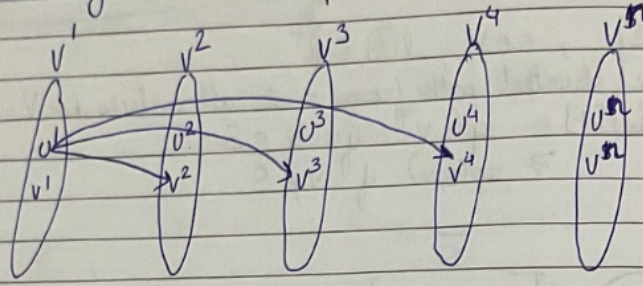
→ Shortest Paths in Graphs with -ve length paths - Bellman Ford

If ∃ a -ve cycle then shortest path are not defined.

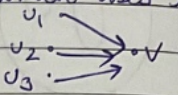
Assuming G has -ve edges but has no -ve cycle.



Lets say we make multiple copies of the graph.

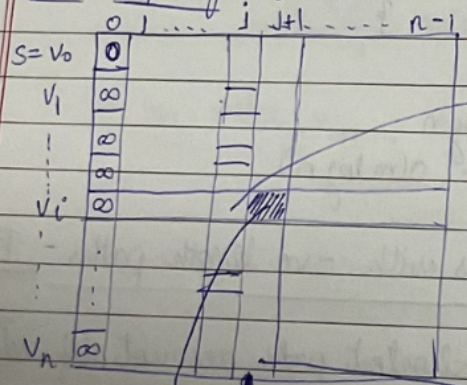


Let $d(v, l)$ = shortest path from s to v which uses almost l edges.

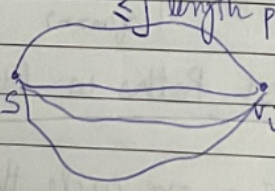


So, now $d(v, i+1) = \min\left([d(v,i)], [d(v_1,i)+l(v_1,v)], [d(v_2,i)+l(v_2,v)], [d(v_3,i)+l(v_3,v)]\right)$

Time Complexity:



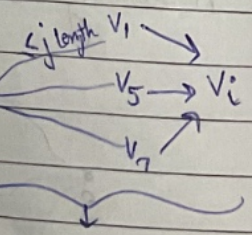
X_{ij} = shortest path from s to v_i containing almost j edges.



~~One time for one column~~ → One column requires $O(m)$ time

want to
If we know X_{ij+1} then we can just do:

So, $$X_{ij+1} = \min\left\{X[i,j], \min_{k:(v_k,v_i)\in E}\left(X[k,j] + l(v_k,v_i)\right)\right\}$$



This gives a path of ≤j+1 edges from s to v_i. Now find a path of length ≤j to a vertex which has an edge to v_i.

Time Complexity = $O(mn)$

→ Bellman-Ford

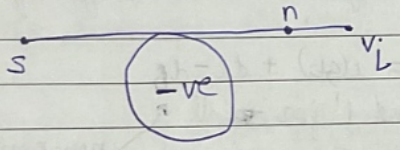
Ex:- Let n=

So, if th
20 edge
a redu

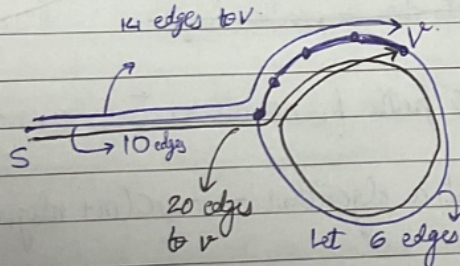
→ Bellman-Ford - single source shortest path where edges may have -ve lengths but no negative cycles.

- If no -ve cycle then any path of length n edges has length ≥ the shortest path of length at most $(n-1)$ edges.

- If there is a -ve cycle of length then we will see a reduction in the length of path having n edges too.



Ex:- Let $n = 20$



So, if there is a -ve length cycle then length of path having 20 edges will be deducted by the -ve cycle i.e. there will be a reduction as compared to the 14 edges path.

→ All Pairs Shortest Paths with -ve edges:

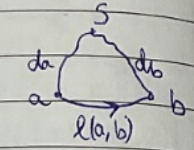
BF gives shortest path from s to v $\forall v \in V$.
APSP - find shortest path bet^n every pair of vertices.
If we run BF n times - $O(mn^2)$ time → Not good.

① Run BF once from a vertex S.
let d_v = shortest path from s to v.

② Modify edge lengths to make every edge non-negative.

$$l'(a,b) = l(a,b) + d_a - d_b$$

We are ~~only~~ find l' for ~~so~~ all edges because the paths are changed by a constant amount = $d_a - d_b$.
~~So we don't need to change positive lengths~~



non-negative length edge.

$d_a + l(a,b) \geq d_b$
$\therefore d_a + l(a,b) - d_b \geq 0$

③ Compute shortest path from all vertices using Dijkstra and lengths as l'.

n-1 Dijkstra algorithm → $n \times O(m + n\log n)$ using Fibonacci heaps.

$\therefore$ Total time $mn + mn + n^2\log n = \boxed{O(mn + n^2\log n)}$

→ Floyd Warshall

A_{ij}^k = shortest path from i to j which uses ~~intermediate~~ vertices numbered $\{1, 2, \ldots, k\}$.

So, A_{ij}^0 = Only a direct edge b/w i and j so no ~~intermediate~~ vertex in the path from i to j; only an edge.
$\therefore A_{ij}^0 = l(i,j)$

So, A_{ij}^{k+1}

k+1 → j should also use only k vertices numbered 1, 2, ..., k same as i → k+1 because only then we can ~~just~~ i → k+1 → j using k+1 vertices

Uses ve

$A_{i,k+1}^k + A$

$\therefore A_{ij}^{k+1}$

Each

by 3

$\therefore A^{k+1}$

$\therefore$ To

→ Longest Mon

with

Find the l
given in

Let X(i) =
~~So, X(i+~~

So, X(i

$\therefore$

So; $A_{ij}^{k+1} = \min(\ ,\ )$

$k+1 \to j$ should also use only k vertices numbered $1, 2, \ldots, k$ same as $i \to k+1$ because only then we can ~~get~~ $i \to k+1 \to j$ using $k+1$ vertices

Uses vertex $k+1$: $A_{i,k+1}^{k} + A_{k+1,j}^{k}$

Doesn't use vertex $k+1$: A_{ij}^{k}

$$\therefore \boxed{A_{ij}^{k+1} = \min\left(A_{i,k+1}^{k} + A_{k+1,j}^{k},\ A_{ij}^{k}\right)}$$

Each entry of A^{k+1} can be computed in constant $O(1)$ time by 3 lookups from the A^k table.

$\therefore$ A^{k+1} can be fully computed in $O(n^2)$ time.

$\therefore$ To compute A^n we need $\boxed{O(n^3) \text{ time}}$

→ Longest Monotone Subsequence:

(let's just consider inc.)

Find the longest inc. subsequence in a given sequence of integers given in an array A.

Let $X(i)$ = length of longest increasing subsequence in $A[1 \ldots i]$ ending at $A(i)$.

~~So, $X(i+1)$~~

So; $X(i+1) = \max_{k=1 \to i} X(k)+1$

only for $A(k) < A(i+1)$

$$\therefore \boxed{X(i+1) = \max_{k:\, k<i+1 \ \&\ A(k)<A(i+1)} X(k)+1}$$

→ For each $x(i) \to O(i)$ time

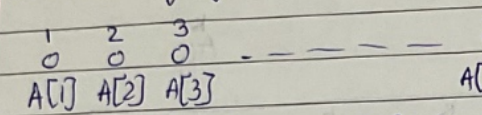
→ Order of computation: $X(1) \ldots X(n)$

⇓

Total time $= \sum_{i=1}^{n} i \Rightarrow \boxed{O(n^2)}$

Method 2:

① Draw a graph G with nodes as A[i] numbered 1 to n.



② Draw an edge from i to j if $A[i] < A[j]$ && $i<j$.

∴ The graph is acyclic.

So the longest path in the graph is the ~~answer~~ longest inc. subsequence.

③ Find the longest path in the DAG.

So; Time req^d = $O(m+n)$ and in worse case $m=n^2$

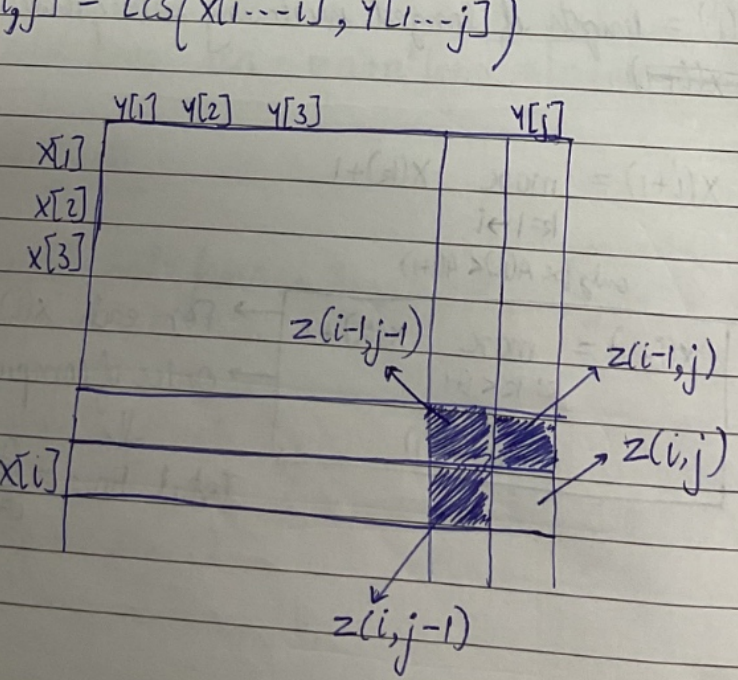
∴ $\boxed{\text{Time} = O(n^2)}$

→ Longest common Subsequence

X: abababbabbab } → LCS = 8.
Y: babbabbbaabbb

Find the longest subsequence that is common in X and Y.

Let $Z[i,j] = LCS(X[1\cdots i], Y[1\cdots j])$



Case 1:

Case 2:

LCS(X

So;

We can
or colu

Each e
So, [

The l
of [O
while

Case 1 : $x[i] = y[j]$

So, $LCS(x[1 \dots i], y[1 \dots j]) = LCS(x[1 \dots i-1], y[1 \dots j-1]) + 1$

Case 2 : $x[i] \neq y[j]$

So, either $x[i]$ or $y[j]$ is not part of LCS.

if $x[i]$ is not part of LCS

$LCS(x[1 \dots i-1], y[1 \dots j])$

if $y[j]$ is not part of LCS

$LCS(x[1 \dots i], y[1 \dots j-1])$

So,

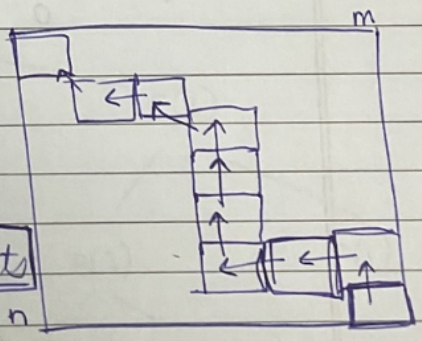
$$Z(i,j) = \begin{cases} 1 + Z(i-1, j-1) & \text{if } x[i] = y[j] \\ \max(Z(i-1,j), Z(i,j-1)) & \text{otherwise} \end{cases}$$

We can fill the table starting from (1,1) either along the rows or columns or diagonally.

Each entry takes $O(1)$ time.

So, **Time complexity = $O(mn)$**

The LCS will be the one consisting of only the diagonal moves while backtracking.



we can backtrack using pointers to reach the top.

→ Edit Distance (X, Y) → A notion of how different X & Y are based on the Edit function.

→ Knapsack Problem without Object Splitting:

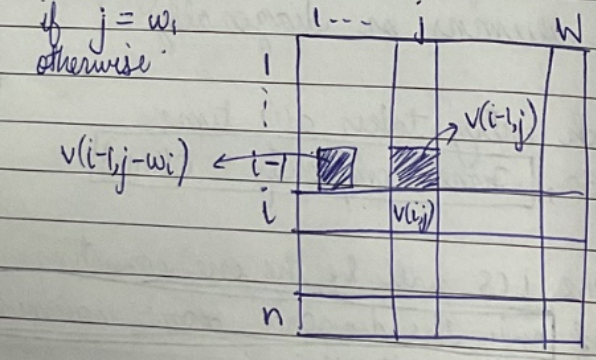
n objects: 1 ... n → value v_i
→ weight w_i

W is the max weight knapsack can hold.
Find the max value set of objects that can fit into the knapsack without splitting i.e. you can't take a fraction of any object.

* Only works if weights are integers.

Let V[i, j] = maximum value ~~subset~~ of any subset of {1 ... i} which has weight j. (we can also use almost j weight)

So, $V[1,j] = v_1$ if $j = w_1$
0 otherwise



So either best subset includes i^{th} object or doesn't include i^{th} object

$V(i,j) = v_i + V(i-1, j-w_i)$

$V(i,j) = V(i-1, j)$

So; $V(i,j) = \max\left(V(i-1,j),\ v_i + V(i-1, j-w_i)\right)$

The final answer will be the $\max_{1 \le j \le W} V[n,j]$ i.e. the max element from the last row.

→ n^{th} Fibon

$F_n = F$

fib(n) {
if
else
}
So, for

F(3)
F(2)
F(1) F(0)
To redu
can sto
F[0] = F[
for
}

the order of computation will be along the row.
For each entry time = O(1).
~~to for total find the last row elements~~

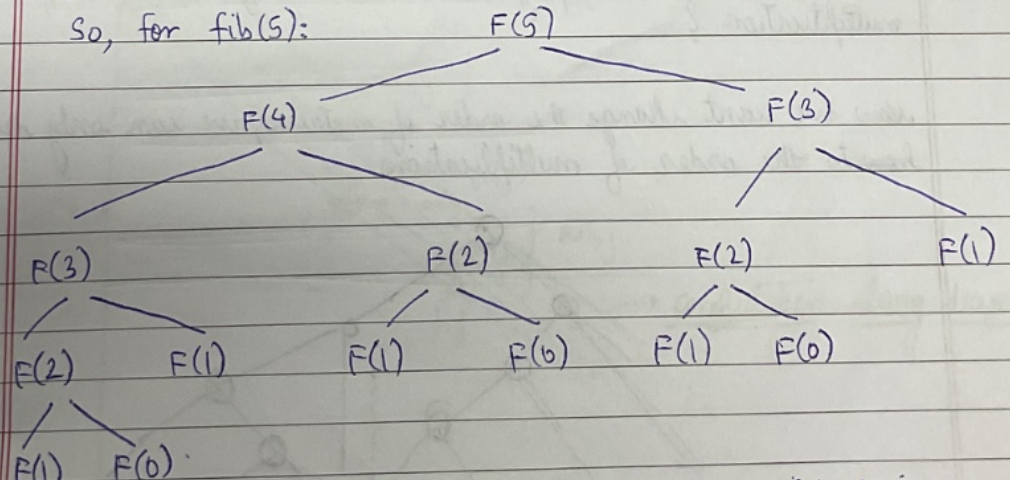
so, **Time complexity = O(nW)** → this can be reduced to O(W) by appropriate divide & conquer or pruning.

→ **n^{th} Fibonacci no:**

$F_n = F_{n-1} + F_{n-2}$ given $F_1 = 1$, $F_0 = 1$.

```
fib(n){
    if n > 2 return fib(n-1)+fib(n-2)
    else return 1
}
```

So, for fib(5):



To reduce recomputations of same value multiple times we can store the values in an array.

```
F[0] = F[1] = 1
for i=2 to n do{
        F[i] = F[i-1] + F[i-2]
}.
```

→ O(n) time

→ Matrix Chain Multiplication

Let M_1, M_2, M_3, M_4, M_5 be 5 matrices.

$M_1 \times M_2 \times M_3 \times M_4 \times M_5$

Let M_i have dimensions $(d_i \times d_{i+1})$.

For $M_1 \times M_2 \rightarrow$ time $= O(d_1 \times d_2 \times d_3)$ because final matrix will have $d_1 \times d_3$ ~~dimension~~ elements and each element is calculated by multiplying d_2 elements of M_1 with d_2 elements of M_2

We know matrix mult. is associative

i.e. $(M_1 \times M_2) \times M_3 = M_1 \times (M_2 \times M_3)$.

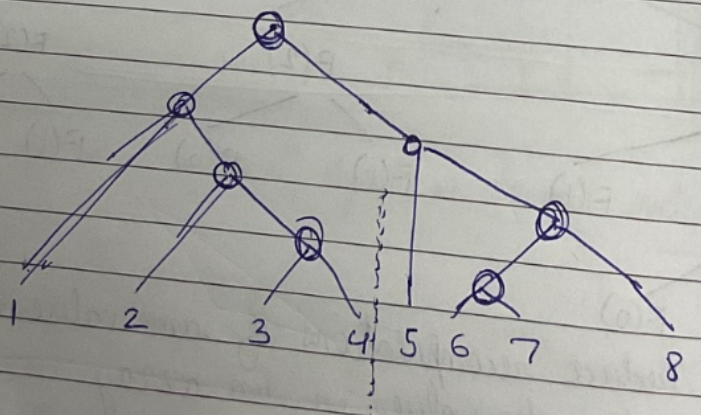
$\downarrow$ $\qquad$ $\downarrow$

$(d_1 \times d_2 \times d_3) \times (d_1 \times d_3 \times d_4)$ $\qquad$ $(d_1 \times d_2 \times d_4)(d_2 \times d_3 \times d_4)$

So, order of multiplication affects the time req^d.

Given $M_1, \dots, M_n$ with dimensions $d_i \times d_{i+1}$, in what order should we multiply so as to minimize the time for multiplication ?

Now we can't change the order of matrices; we can only decide ~~how to~~ the order of multiplications.



Split at half.

Let for $j \geq i$, $X(i,j)$ = be the min. computation req^d to multiply matrices $M_i, M_{i+1}, \ldots, M_j$.

So, $X(1,2)$ = # of computations req^d to compute $M_1 \times M_2 = d_1 \times d_2 \times d_3$

$X(2,3), X(3,4) \ldots\ldots X(n-1,n)$ are simple to compute.

For $X(1,3) = \min\left(X(1,2) + d_1 \times d_2 \times d_4 \;,\; d_1 \times d_2 \times d_4 + X(2,3)\right)$

$\downarrow$ $\qquad\qquad$ $\downarrow$

$(M_1 \times M_2) \times M_3 \qquad M_1 \times (M_2 \times M_3)$

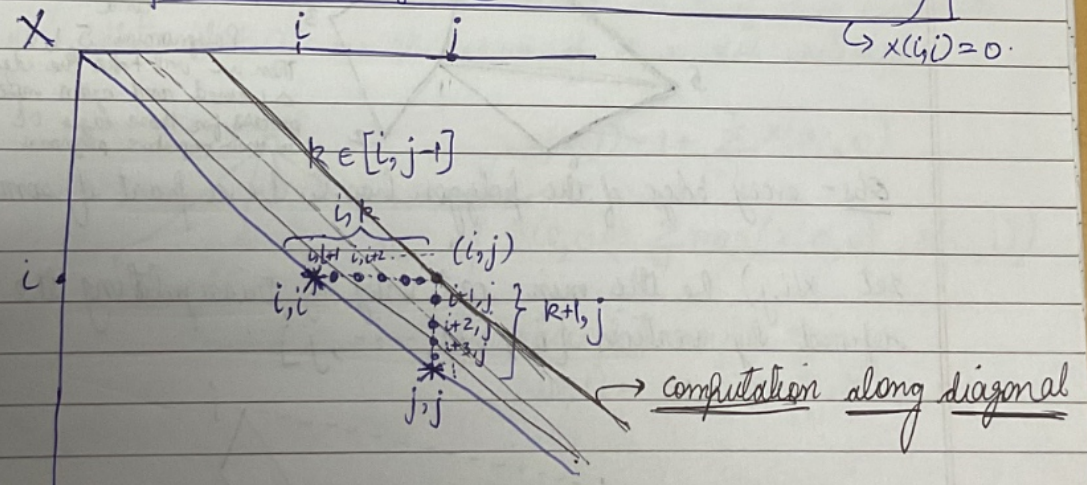
Now for $X(i,j)$: $\underline{M_i \times M_{i+1} \times \ldots\ldots \times M_j}$

$\downarrow$

$\underline{M_i \times M_{i+1} \ldots\ldots \times M_k} \times \underline{M_{k+1} \times M_{k+2} \times \ldots\ldots \times M_j}$

$$\therefore X(i,j) = \min_{i \leq k \leq j-1} \left(X(i,k) + X(k+1,j) + d_i \times d_{k+1} \times d_{j+1}\right)$$

$\hookrightarrow X(i,i) = 0.$



→ computation along diagonal

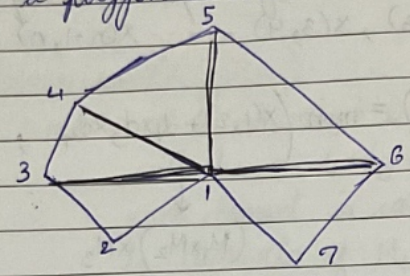
Time for each value → $O(n)$

Time for complete ~~calculation~~ order determination → $O(n^3)$

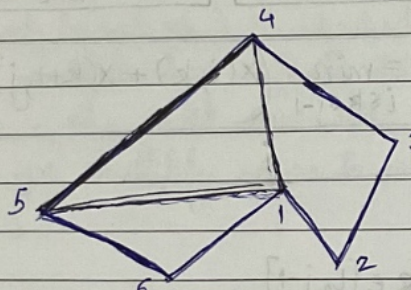
After this we will actually have to multiply the matrices in the determined order.

→ Triangulating a Polygon

Adding lines to a polygon so that is becomes full of △.



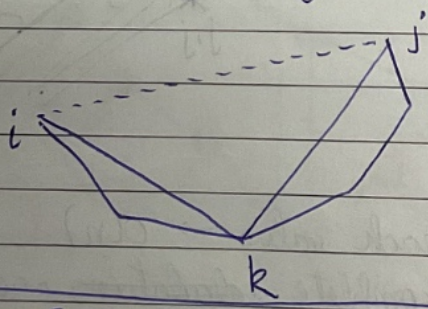
Triangulating a polygon by minimizing the total length of the line segments added.



For edge 4,5 if we select vertex 1 to be the 3rd vertex in the △ cont^n edge 4,5 then now we have 2 subproblems →
Polynomial 1,4,3,2
and
Polynomial 5,1,6.
Then we will take the edges of the △ formed and again repeat the process for these edges i.e. 1,4 and 1,5 in their respective polynomial.

Obs - every edge of the polygon has to be a part of some △.

Let $X(i,j)$ be the min. cost way of triangulating the polygon defined by vertices $\{i, i+1, \cdots, j\}$.



So, $$X(i,j) = \min_{i+1 \le k \le j-1} \{ X(i,k) + X(k,j) + l(i,k) + l(k,j) \}$$

→ Independe

In a gr
no edge
Find the

Suppose

~~So we co~~
If a lin
we can
sets of

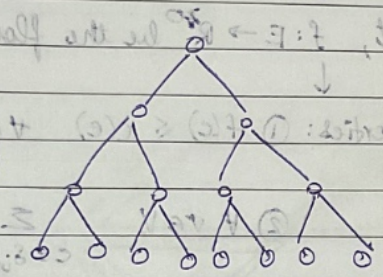
In an
indepen
and

Independent set

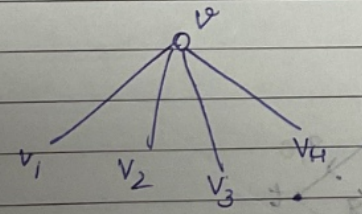
In a graph: a set of vertices $S \subseteq V$ is independent if there is no edge bet^n any two vertices of S.
Find the largest independent set in G.

Suppose G is a tree.

If a binary tree then we can add independent sets of alternate levels.



In an arbitary tree, let $X(v,1)$ be the size of the largest independent set in subtree rooted at v if v is included and let $X(v,0)$ be the same if v is not included



$$X(v,1) = 1 + \sum_{i=1}^{4} X(v_i, 0)$$

$$X(v,0) = \sum_{i=1}^{4} \max\left(X(v_i,0), X(v_i,1)\right)$$

If vertices has weights then

$$X(v,1) = w_v + \sum_{i=1}^{4} X(v_i, 0)$$

FLOWS

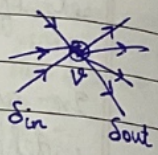
Directed graph $G = (V, E)$, $s, t \in V$ (source, sink/target), $c : E \to \mathbb{R}^+$

what's the max amount of flow that can go through the network from $s \to t$??

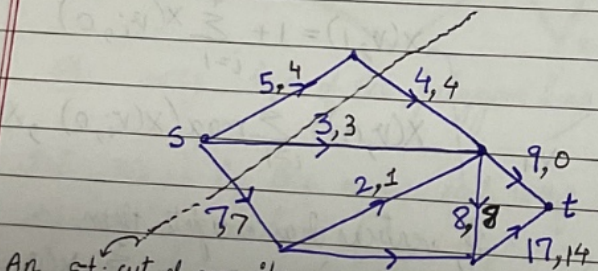
Now let, $f : E \to \mathbb{R}^{\geq 0}$ be the flow func. i.e. max flow through an edge.

Properties: ① $f(e) \leq c(e) \quad \forall e \in E$ ⟶ Capacity Constraint.

② $\forall v \in V \quad \sum_{e \in \delta_{in}(v)} f(e) = \sum_{e \in \delta_{out}(v)} f(e)$



Objective: Maximise flow going out of s. i.e. maximize $\sum_{e \in \delta_{out}(s)} f(e)$.



An s-t cut of capacity 14

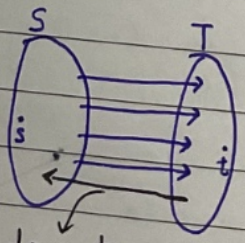
An 's-t cut' is a partition of V into sets S, T such that

① $S \cap T = \phi \quad S \cup T = V$

② $s \in S, \ t \in T$.

Capacity of a s-t cut is

$c(S, T) = \sum_{e=(u,v), \ u \in S, \ v \in T} c(e)$



We do not count edges that are in reverse dir^n.

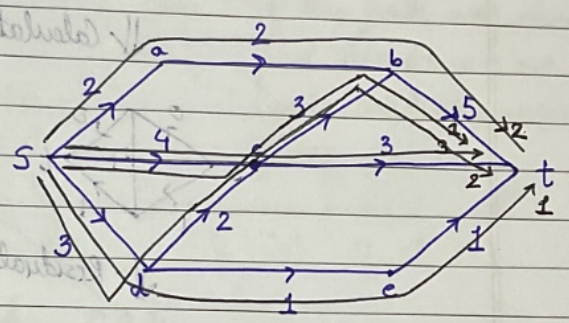
In any graph with n vertices there can be 2^{n-2} s-t cuts because s has to be in S and t in T but we have to partition rem^{n} $n-2$ vertices into S & T sets.

Maximum s-t flow ≤ minimum capacity s-t cut.

So, **Max. s-t flow = Min. s-t cut**

→ **Flow Decomposition**

We find various flows paths possible from $s \to t$.
So, for any edge the total flow from it is the sum of flows from all paths involving that edge.



s-a-b-t ⇒ 2
s-c-t ⇒ 3
s-d-e-t ⇒ 1
s-c-b-t ⇒ 1
s-d-c-b-t ⇒ 2
→ Total = 9

→ **Ford-Fulkerson Algorithm**

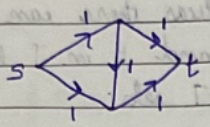
① Find a path from s to t say, P. in (I_R) → Residual graph (I_R) is obtained by removing edges of capacity 0.
② Send as much flow as you can along P
③ Repeat until no more flow can be sent.

→ ② Compute the residual capacity of all the edges in P.

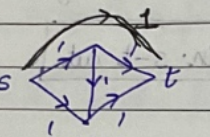
$c'(e) = c(e) - f(P)$.

$f(P) = \min_{e \in P} c'(e)$.

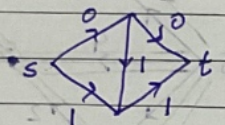
ex:-



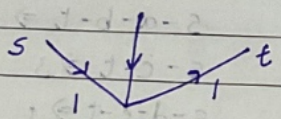
⇓ ① Find a Path.



⇓ Calculate residual capacities

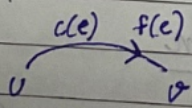


⇓ Residual graph.

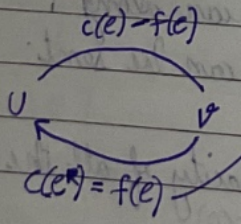


⇓ Find another path and Repeat until no more paths exists.

Now suppose,

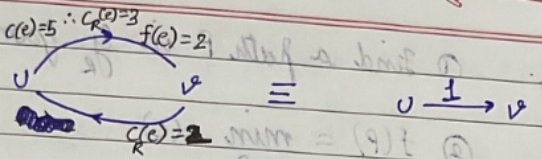


⇓

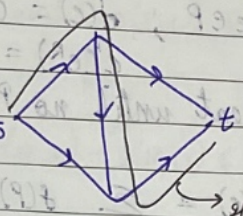


Suppose we shouldn't have sent f(e) from u → v using edge e, so to correct our mistake we add a reverse flow.

for ex:

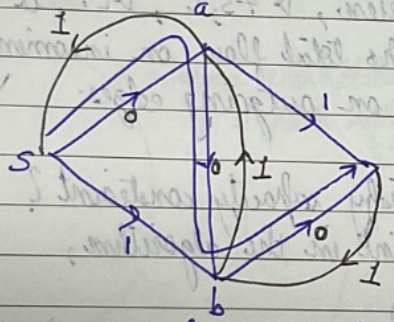


So, in previous example:



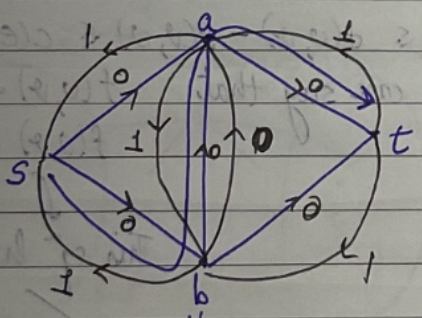
if this path was chosen first then we can no longer send any more flow ⇐

But now;

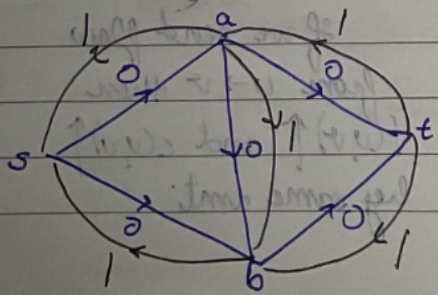


Now we have another path so we use that path

⇓



⇓



So, ① Find a path P in G_R from s to t.

② $f(P) = \min_{e \in P} c(e)$

③ $\forall e \in P$, $c'(e) = c(e) - f(P)$
$c'(e^R) = c'(e^R) + f(P)$.

④ Repeat until no P can be found.

So; $f(e) = \sum_{P : e \in P} f(P)$ will be the total flow for any edge

Does $f: E \to R^+$ satisfy conservation?
⇒ Yes, for any vertex, $v \neq s, t$, $v \in P$, a path P contributes $f(P)$ units of flow to the total flow on incoming edges as well as to the total flow on outgoing edges.

Does $f: E \to R^+$ satisfy capacity constraint?
⇒ yes, at any point in the algorithm,

$u \xrightarrow{c(e)} v$ $\quad c(e) = c(u,v) + c(v,u) =$ initial capacity of e

and $f(u,v) - f(v,u)$ is the net flow on edge (u,v)

So;

$$0 \leq f(u,v) - f(v,u) \leq c(e).$$

Proof: we can say that $f(u,v) - f(v,u) = c(e,u)$

$\therefore f(u,v) = f(v,u) + c(v,u)$.

This eqⁿ holds for every case

If we send flow from $u \to v$ then $f(u,v) \uparrow$ and $c(v,u) \uparrow$ by same amt.

If we send flow from $v \to u$ then $f(v,u) \uparrow$ and $c(u,u) \downarrow$ by same amt. so, $f(u,v)$ remains same.

Thus, we can say the eqⁿ $f(v,v) = f(v,v) + c(v,v)$ holds.

$\Downarrow$

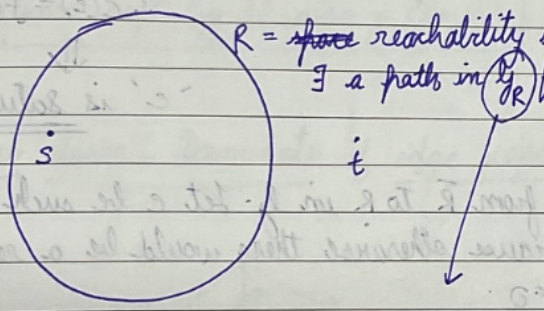
$f(v,v) - f(v,v) = c(v,v)$ holds.
and $0 \leq c(v,v) \leq c(e)$

$\Downarrow$

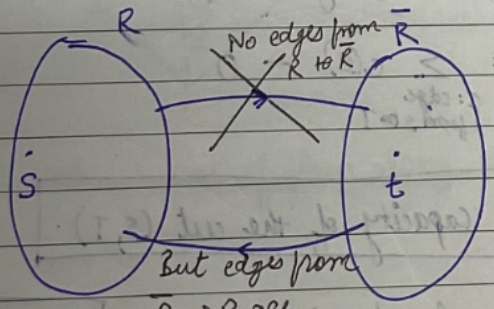
$0 \leq f(v,v) - f(v,v) \leq c(e)$ holds.

Hence, Proved.

Why is FF algorithm correct? why does it compute max flow?
⇒ when FF stops,



Residual graph does not contain edges with residual capacity = 0.

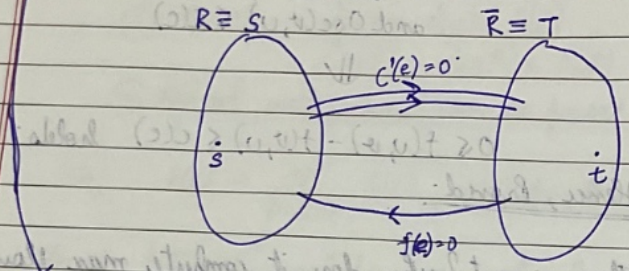


In G_R there are no edges from R to $\bar{R}$.

~~We said that~~

Now consider edges from R to $\bar{R}$ in G_f. These edges have 0 residual capacity.

So, if we take the same vertex sets in the original graph:



Let e be such an edge. then, $c'(e) = c(e) - f(e) = 0$

$\therefore c(e) = f(e)$

$\Downarrow$

'e' is saturated

Consider the edges from $\bar{R}$ to R in G_f. Let e be such an edge. Then, $f(e) = 0$ because otherwise there would be a reverse edge with residual capacity $\neq 0$.

So, Netflow for S = flow going out of S − flow going into S

$$= \sum_{e:\ \text{edge from } S \text{ to } T} c(e) - 0$$

$\therefore$ **Netflow for S = capacity of the cut (S, T).**

and we have already proved that **Max flow = min. s-t cut.**

Hence Proved that FF algorithm is correct and gives maximum flow.

Max-Flow-Min-Cut Theorem:

The max. s-t flow = the min. capacity s-t cut.

If $c: E \to \mathbb{Z}^+$ and let f^* is the max-flow then $f^*: E \to \mathbb{Z}^+$.
i.e. if capacities are integers then the max-flow is also integer.

Time req^d by FF algo: In each iteration we require a DFS to compute a path from s to t — $O(m)$

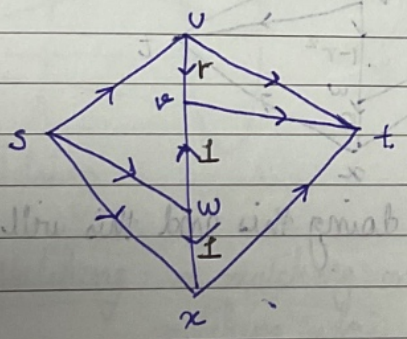
# of iterations: in each iteration we send ≥ 1 unit of flow.
↳ only when capacities are integers.

So, if max-flow = F
then # of iterations $\leq F$

$\therefore$ **Time req^d by FF algo = $O(mF)$**

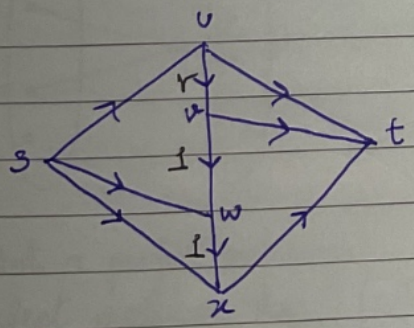
FF algo doesn't Terminate if edge capacities are Irrational.

eg:

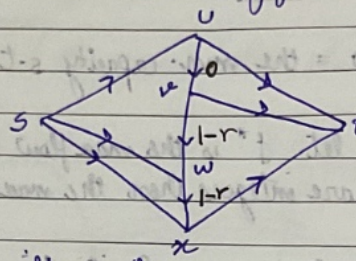


r is such that $1-r=r^2$

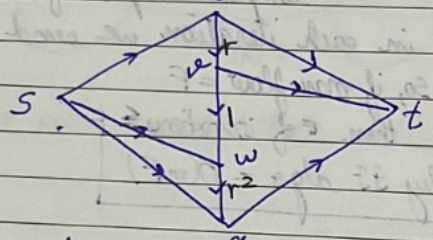
We first send 1 unit of flow along s-w-v-t so, edge v-w gets reversed with ~~residual~~ reverse capacity 1.



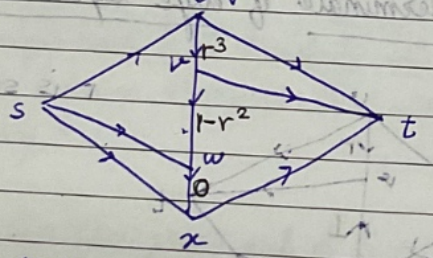
Now we send r units of flow through s-u-v-w-x-t.



Now we will use the reverse edges and send r along s-w-v-u-t



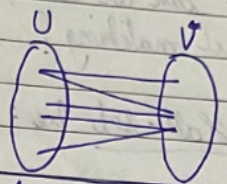
Now we send r^2 along path s-u-v-w-x-t



And we just keep doing this and this will never end because $1 > r > r^2 > r^3 \ldots\ldots$

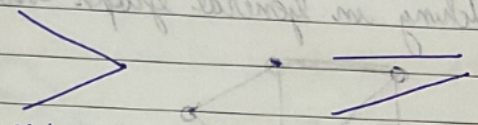
→ Bipartite Matching:

$G: (U, V, E) \quad E \subseteq U \times V$

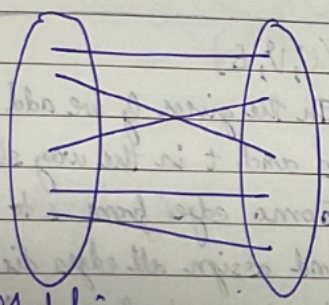


Matching: a matching is an **independent set of edges**

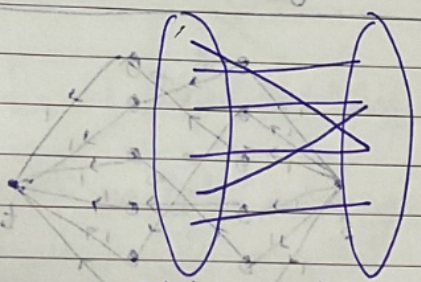
We say 2 edges are independent if they do not have a mom common vertex.



Not Independent Independent.

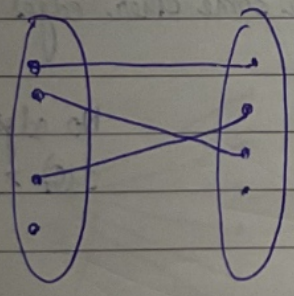


Matching

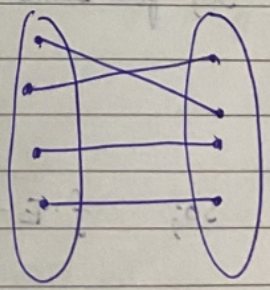


Not a matching.

Perfect Matching: a matching in which every vertex has a matching edge incident to it



Not a perfect matching



Perfect matching.

A perfect matching has $|U|$ edges.

How can we determine if a given graph $G = (U, V, E)$ has a perfect matching?

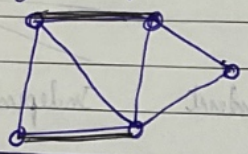
Let Hall-set be: For $S \subseteq V$.

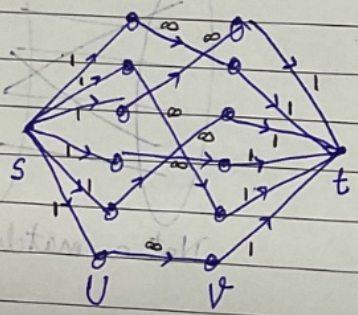
$N(S) \subseteq V$ is the set of vertices which are adjacent to S.

$S \subseteq U$ is a Hall set if $|N(S)| < |S|$

** Hall's theorem: G has a perfect matching iff it has no Hall set.

→ Matching in General Graph (Non-Bipartite)





$G = (U, V, E)$

In the given G we add 2 more vertices s and t in the way shown and add some edge from s to U and t to V and assign all edges dir^n from $s \to t$.

Edge from s to & on t have capacity = 1 and the edges $e \in E$ have capacity = ∞.

Now, compute max flow from s to t

Max. flow is integral $f: E \to \mathbb{Z}^+$

So, for some edge $f(e) = 0$ and for some other edges $f(e) = 1$

↓

No edge can have $f(e) > 1$

So; $f: E \to \{0, 1\}$.

Let M be the edges of E that have $f(e) = 1$.

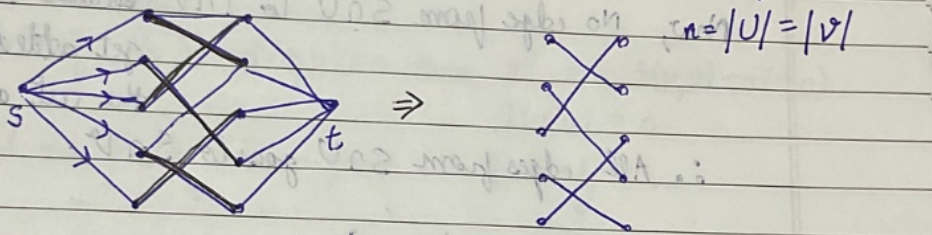
$$M = \{ e \in E \mid f(e) = 1 \}.$$

The edges of M form a Matching. b/c at any vertex $v \in U$ or V cannot have more than 1 incoming edge with flow 1 and 1 outgoing edge with flow 1.

~~The amount of flow sent~~ =

Claim: The amount of flow sent = size of M.

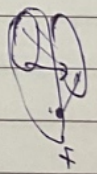
If $G = (U, V, E)$ had a perfect matching then I can send n units of flow from s to t.



Suppose maxflow from s to t $< n \Rightarrow$ no perfect matching

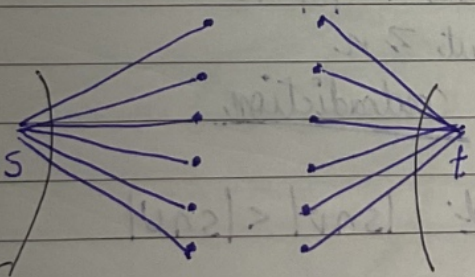
$\Downarrow$

There should exist a Hall set.



To prove: $\exists$ a Hall set.

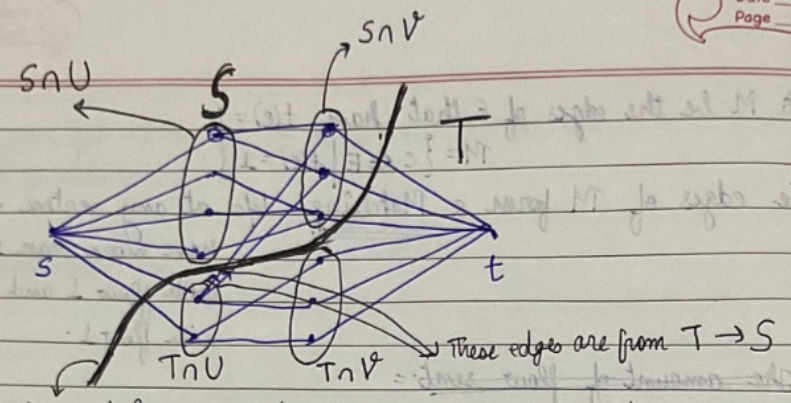
~~Suppose~~ Since maxflow $< n \Rightarrow \exists$ a s-t cut of capacity $< n$.



This cut has capacity = n

$|U| = n$ $|V| = n$

This ~~capacity~~ cut has capacity = n.



This cut has capacity $< n$ c/c the edges it is cutting are not all from $S \to T$, some are from $T \to S$ and those edges' capacity would not be counted.

Now, No edge from $S \cap U$ to $T \cap V$ because their capacity would get added in the capacity of the cut and ~~becomes~~ become ∞.

$\therefore$ All edges from $S \cap U$ goes to $S \cap V$.

Claim: $|S \cap V| < |S \cap U|$

Proof: Let $|S \cap V| \geq |S \cap U|$

$(|S \cap V| + |T \cap U|) \geq |S \cap U| + |T \cap U|$

This is the size of cut because $|S \cap V|$ edges goes into the vertex t and $|T \cap U|$ is the edges that are coming from vertex s.

$\therefore$ Size of cut $\geq |U| = n$

Size of cut $\geq n$

~~Not~~ Contradiction

Hence Proved: $|S \cap V| < |S \cap U|$

Ford-Fulk

① Find a

② $\lambda(P) =$

③ Send

④ Update

⑤ Update

⑥ Repeat

→ Sligh

~~Since~~

Let

So, t

Thus,

Ford-Fulkerson Algorithm: Works only when capacities are Integers or Rational only.

① Find a path from s to t in G_f^R.

② $\lambda(P) = \min_{e \in P} c'(e)$

③ Send $\lambda(P)$ units of flow along P. $\forall e \in P \quad f(e) \leftarrow f(e) + \lambda(P)$

④ Update residual capacities

$\forall e \in P$ $\quad c'(e) = c'(e) - \lambda(P)$

$\quad c'(e^R) = c'(e^R) + \lambda(e)$

⑤ Update flow variables.

$\forall (u,v) \in P$ if $f(u,v) > f(v,u) > 0$ then $f(u,v) = f(u,v) - f(v,u)$

$\quad f(v,u) = 0$.

⑥ Repeat until no path remain from s to t in G_f^R.

→ **Slight change**: Find the **shortest path** P from s to t in G_f^R.

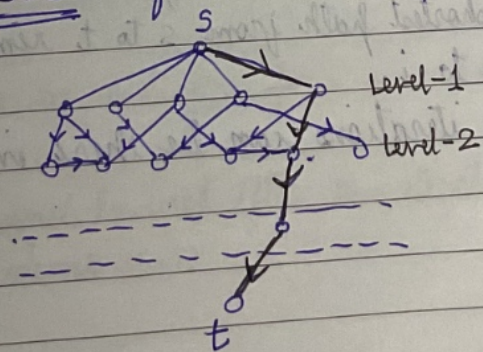
This will allow us to use FF algo. for irrational capacity edges.

~~Sorce~~

Let lengths of all edges be 1.

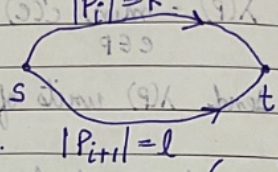
So, the shortest path is just the path with ~~min~~ least ~~vertex~~ edges.

Thus, we use **BFS** to find the shortest path.



Claim: The lengths of the shortest paths in monotonically non-decreasing.

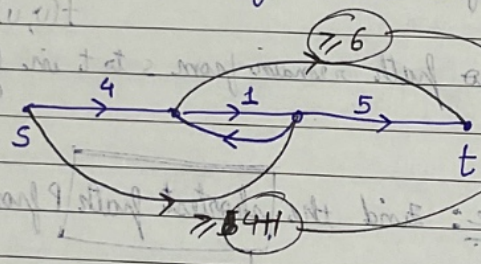
Proof: Suppose, shortest path P_i was length k at iteration i & P_{i+1} has length ℓ $(\ell < k)$ at iteration $i+1$



In iteration i we could have either reduced capacity of some edge in P_i to 0 or added some reverse edges to G^R.

Now; removing edges of 0 capacity from G^R at iteration i would not generate a shorter path at iteration $i+1$

So; Adding some reverse edge created as shorter path



Because otherwise we would have already picked this shorter path in prev. iterations

Hence Proved by contradiction

→ Iteration 1 2 3 4 5 6 7 8 9

Shortest path length: 5 5 6 6 6 7 8 8 8

Phase 5 | Phase 6 | Phase 7 | Phase 8

Phase i - Phase is a collection of iterations in which the length of the shortest path from s to t remains unchanged and is equal to i.

How many iterations can be there in a Phase?

With every iteration we lose one edge (the one whose capacity is min.) which cannot be used again (as a reverse edge) in that same phase because then the graph length of the path using that reverse edge will be > the shortest path length of current phase.

$\Rightarrow$ No. of iterations in a phase $\leq m$.
No. of Phases $\leq n-1$
Time for each ~~Phase~~ iteration $= O(m)$

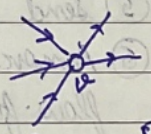
$\Rightarrow$ **Total time $= O(m^2 n)$**

↓

This doesn't depend on the flow or the capacity of edges unlike Ford-Fulkerson
$\Rightarrow$ We can use this modified algorithm for edges with irrational capacities.

→ **We can Implement a Phase in $O(n^2)$ time**

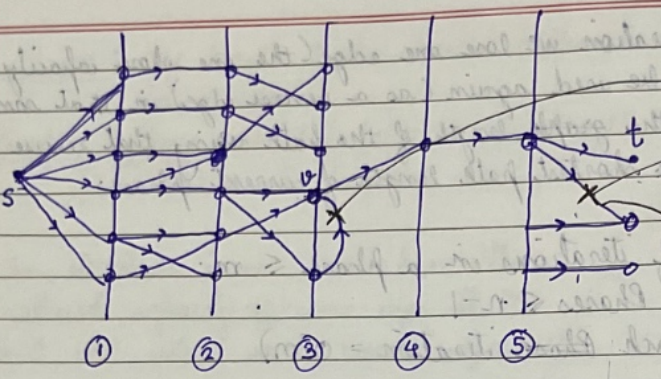
① Find a vertex with min. Throughput.



Throughput(v) = min { Total res. capacity of incoming edges, Total res. capacity of outgoing edges going to next layer }

② Send a flow from v to t and full. ~~the~~ from s of Throughput(v) units.
~~No. of saturating pushes $\leq n_v$~~
— No. of non-saturating pushes $\leq n$.

In each iteration we send flow from a vertex v (with min. throughput) to t and s.
Now, vertex v cannot be used after this iteration.
Hence, no. of iterations $\leq n$.



Cannot count these edges as throughput of the vertex.

We are sending a flow of Throughput(v) layer by layer, so it should all go to t from the prev. layer. Thus we can't count this edge in throughput calculation.

① Do a BFS in f^R, t is the sink at level i.
② Remove all vertices from level i and later except t because we can't use these in the calculation of min. throughput vertex.

Iteration:

③ Compute throughput of each vertex.
④ Find vertex v^* with min. throughput = h.
⑤ Send h units of flow from v to t and from v to s.
⑥ Remove vertex v and edges through which you sent saturating flow from other vertices.
So, we remove vertex v and all edges incoming and outgoing on v & edges on other vertices from which you sent saturating ~~push~~ flow.

Repeat until no vertex of non-zero throughput.

⑦ Recomputes f^R.

Repeat until no path from s to t in f^R.

- In one Iteration : ① # of non-saturating pushes $\leq n$
Because each vertex can have only 1 unsaturating push because after that the flow which was coming into that vertex was 0 over.

② # of saturating pushes can't be said in each iteration.

- In one Phase : ① # of iterations $\leq n$
Because in each iteration atleast 1 vertex will be removed from the graph.

② # of saturating pushes in 1 Phase $\leq m$
Because in each phase 1 edge can be saturated only once.

③ # of non-saturating pushes $\leq n^2$

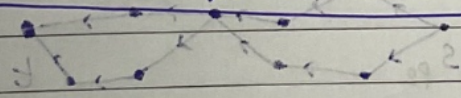
$\Rightarrow$ Total Time $= O(m+n^2) = O(n^2)$ → Thus, we can implement 1 Phase in $O(n^2)$ time.

⇓

Max. no. of phases $\leq n-1$

⇓

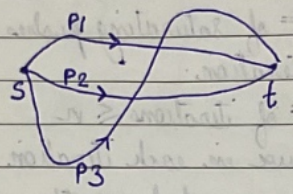
$\Rightarrow$ Total time of ∃f modified $= O(n^3)$



→ Edge Disjoint Paths

Find the maximum no. of (edge-disjoint) paths bet^n s & t.

↓ do not share edges.



Algo:

① Give every edge a unit capacity
② Find max. flow from s to t

To prove: The algo gives the max. no. of disjoint paths.

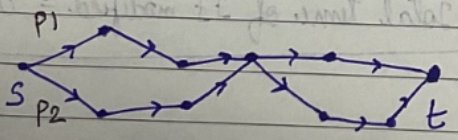
Suppose F is the max. flow computed and D is the max. no. of edge disjoint paths from s to t.

Our claim is F = D

① Let proove F ≤ D: if we sent 10 units of flow then there are atleast 10 disjoint paths.

Consider the flow func^n. f.

Decompose f into F paths.



$f: E \to \{0,1\}$: consider edges which carry 1 unit of flow from s and decompose into paths.

# of paths = # of edges with unit flow going out of s.

$\Rightarrow D \geq F$. ——— (1)

② Lets prove $D \leq F$: if I have D edges disjoint paths from s to t then I can send atleast D units of flow.

From each path max 1 unit of flow can be sent because all edges have capacity = 1.

$\Rightarrow$ flow through all D paths = D

$\Rightarrow F \geq D$ ——— ②

From ① and ②, $F = D$

Hence Proved.

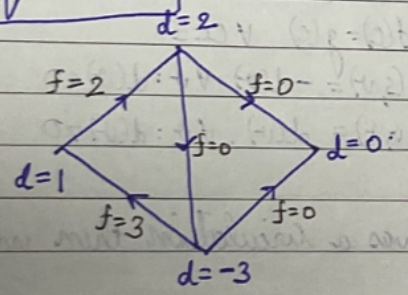
→ Circulation

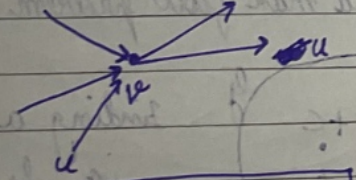
Define demand $d : V \to \mathbb{R}$

$d(v)$ is demand of vertex v. If $d(v) > 0$ then net flow entering $v = |d(v)|$

If $d(v) < 0$ then net flow outgoing $v = |d(v)|$

and $\sum_{v \in V} d(v) = 0$





$$\sum_{(v,u) \in E} f(v) - \sum_{(u,w) \in E} f(v) = d(v)$$

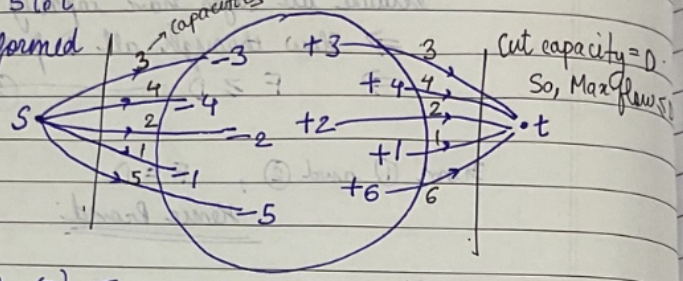
Incoming flow

Outgoing flow.

circulation is f such that it satisfies: ① $0 \le f(e) \le c(e)$ $\forall e \in E$

② $\sum_{v \in V} d(v) = 0$

Find max flow from s to t after this. in the G^R formed after this.



If max flow has value $D = \sum_{v: d(v) > 0} d(v)$ then there is a feasible circulation in the original graph.

To proof: ⇒ If there was a circulation $g: E \to R^+$ in the original graph then there is a flow from s to t of value D such that

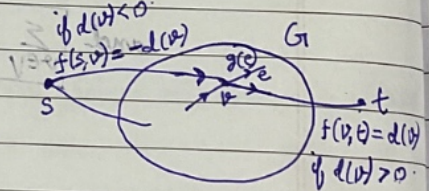
the flow f is:

f is defined as:

$f(e) = g(e)$ $\forall e \in E$

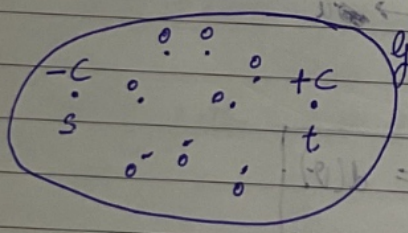
$f(s, v) = -d(v)$ $\forall v : d(v) < 0$

$f(v, t) = d(v)$ $\forall v : d(v) > 0$



So, if there was a circulation then we can build this flow function.

- We can convert a max flow problem into a circulation problem:



Finding a circulation in G = Find a max s-t flow

set the demand for s as -C and for t as +C and 0 for all others.

→ Then, find the maximum C for which this is a feasible circulation possible using binary search on C.

→ Max Flow

∀ e ∈ E

∀ v ∈ V

Suppose,

suppose

Clair

Proof

since

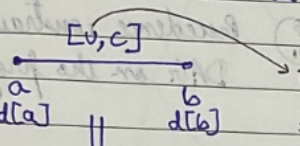
∴ val

→ Max Flow with Lower Bounds

$\forall e \in E \quad \underset{\text{lower bound}}{\underline{u(e)}} \leq f(e) \leq c(e)$

$\forall v \in V$ incoming flow at v − outgoing flow at v = $d(v)$.

Suppose, I

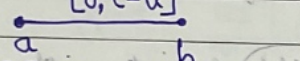


This means that atleast u units of flow should be on this edge.

⇓

Atleast u units of flow will come in through a and going out b. So, the demands of a and b will change.

II



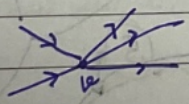
Suppose g is a feasible circulation for I.

Now, define $g'(e) = g(e) - u(e)$

Claim: g' is a feasible solution for II.

Proof: ① $\forall e \quad g'(e) \leq c(e) - u(e)$ since $g(e) + u(e) = g(e) \leq c(e)$.

② $\forall v$ net incoming − net outgoing = $d(v)$



i] Net incoming at v in g' = net-incoming at v in g − $\sum_{e:\text{incoming at } v} u(e)$

ii] Net outgoing at v in g' = net outgoing at v in g − $\sum_{e:\text{outgoing at } v} u(e)$

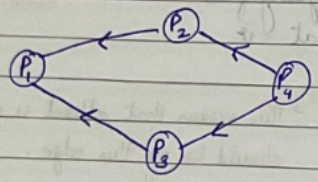
Since conservation of flow at v; $\sum_{e:\text{incoming at } v} u(e) = \sum_{e:\text{outgoing at } v} u(e)$

∴ net incoming in g' at v − net outgoing in g' at v = net incoming in g at v − net outgoing in g at v = $d(v)$.

→ Project Selection

Projects: $P_1, P_2, \cdots, P_n$

- $a \to b$ means 'b' can be done only if 'a' is completed.



Precedence constraints form a DAG on the projects.

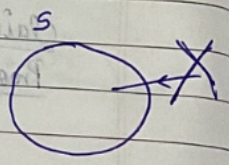
- Each project has a value $v_i \in R$.

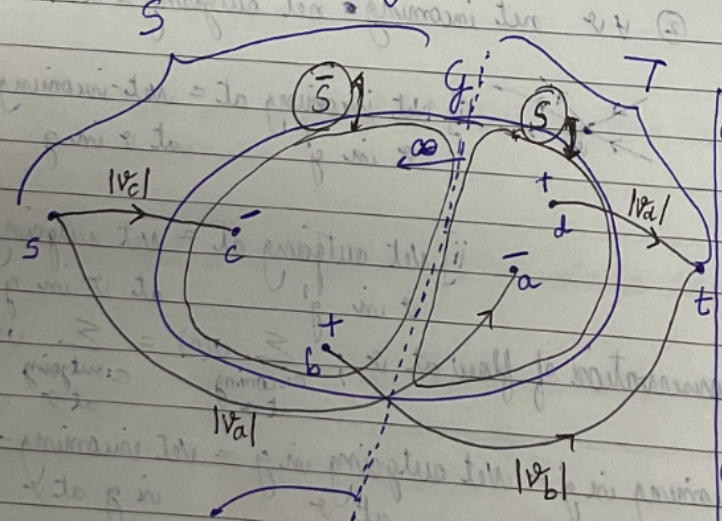
Identify a set S of (feasible) projects with maximum total value.

Precedence constraints should be satisfied for all.

⇓

We need to find S s.t. there is no edge going into S.





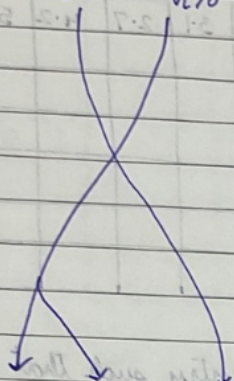
① Give all the edges in original G a capacity of ∞

② Draw edges from s to vertices with value < 0 of capacity same as |value|.

③ Draw edges from vertices with +ve value to t of capacity same as |value|.

Capacity of cut = $\sum_{\substack{i \in S \\ v_i < 0}} |v_i| + \sum_{\substack{i \in \bar{S} \\ v_i \geq 0}} |v_i|$ + edges from $\bar{S}$ to S in G having ∞ capacities.

So; Capacity of cut = $\sum_{\substack{i \in S \\ v_i < 0}} |v_i| + \sum_{\substack{i \in \bar{S} \\ v_i > 0}} |v_i|$ + ~~edges from $\bar{S}$ to $\bar{\bar{S}}$ with ∞ capacity.~~



for a feasible solution there can not be ∞ cut capacity because it would mean that there is some project in S whose precedence is not satisfied and thus an incoming edge from $\bar{S}$ to S is not allowed.

$$= \left(\sum_{v_i > 0} v_i - \sum_{\substack{i \in S \\ v_i > 0}} v_i \right) + \sum_{\substack{i \in S \\ v_i < 0}} (-v_i)$$

$$= \underbrace{\sum_{v_i > 0} v_i}_{\text{Constant}} - \left[\sum_{\substack{i \in S \\ v_i > 0}} v_i + \sum_{\substack{i \in S \\ v_i < 0}} v_i \right]$$

$\therefore$ **Capacity of cut = Constant − Value of set S**

Thus, **Capacity of cut $\propto \frac{1}{\text{Value of set S}}$**

$\because$ We wanted to maximize value of set S,

$\therefore$ now we can just find the minimum cut

Privacy

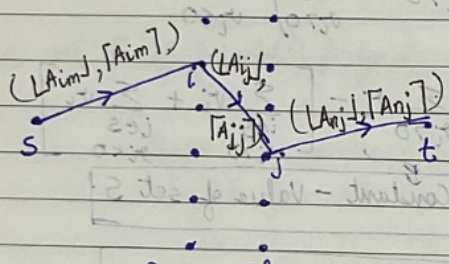
1.2	2.3	3.1	2.7	4.2	5.1	18.5
	2.4					
	2.5					
	3.7					
	3.2					
	3.9					
	18.0					

Sum Column ↓ m

Sum Row → n

→ Sum of all entries

Round up/off each entry such that sum row/column remains maintained.



Lower bound = ⌊Aij⌋ ⇒ Demand

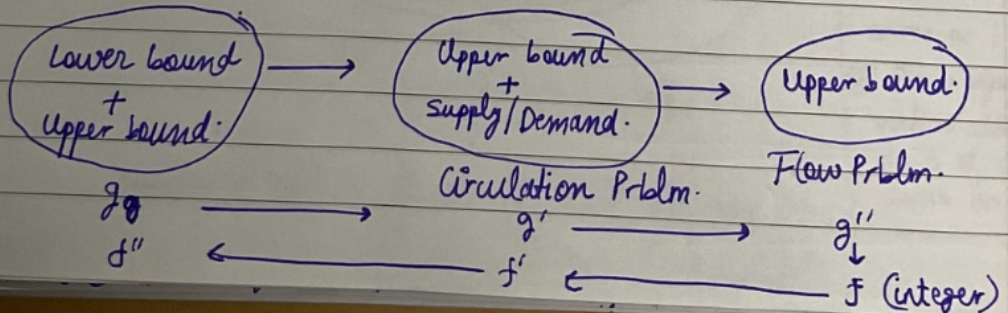
Upper bound = ⌈Aij⌉ ⇒ Capacity

Claim: There exists a flow f s.t. for every edge $v(e) \leq f(e) \leq c(e)$.

Because for every edge the lower bound is ~~the~~ ≤ the entry in the table and we know for sure that the entries in the table are valid flow values because their sum is the value of row and column ~~entries~~ vertices.

∴ Conservation will hold.

If there exists a circulation & capacities are integral the max flow is integral.



Baseball Elimination

	A	B	C	D	E	F
D	1	2	1	0	0	1

You are given the current scores of the teams & the table.

A – 62
B – 64
C – 68
D – 65
E – 67
F – 68

X_{ij} = # of games remⁿ betⁿ i & j.
So, $X_{ii} = 0$ i.e. Diagonal elements = 0
and $X_{ij} = X_{ji}$ i.e. Symmetrical Matrix.

Team with the max. score wins.
In each match the winner gets 2 points & loser gets 0 points and in case of tie both get 1 point.
We have to see if there exists a way s.t. D wins the tournament.
It might not be possible always.
for eg. D 65, D-F → D wins
F 68, D-J → D wins
J 68, F-J → In any case score of F ≥ D or J ≥ D.
So in this example D can't win.

Because in every game 2 points will be distributed b/w the 2 teams

: a random & check : attempt
Assuming k that D can score atleast k points
∴ lower bound for D = k – current points.
upper bound for D = ∞.

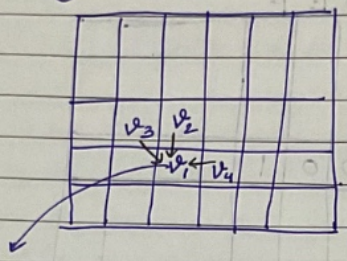
all other teams ≤ k-1 points b/c only then D can win.

One vertex for each remⁿ game

One vertex for each team

We pick a value k and check if ∃ a way s.t. D gets atleast k points and all other teams get ≤ k-1 points.

→ Open Pit Mining



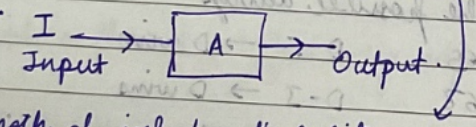
Each block has some value & precedence:
A block can be removed only if all its pre-req. blocks are removed:

|||

Project selection problem.

Polynomial Time

- Algo. A has a polynomial running time if running time of A is bounded by a polynomial in the length of the input.



Length of input = # of bits req^d to encode the input.
(III)

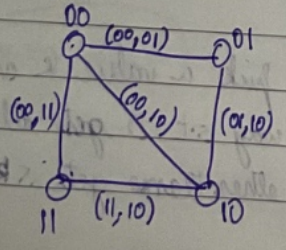
for ex: n numbers : # of bits $\leq$ $n \cdot \log U$ bits if U is the largest integer among them.

for ex: Graph $G=(V,E)$ → For vertices, $\log n$ bits for each vertex and for each edge we need $2 \times \log n$ bits

So, # of bits = ~~$O(m \log n)$~~ $2m \log n + n \log n$

∴ For ~~Graph $(V,E) = O(m \log n)$~~

∴ For Graph $(V,E) = O((m+n) \log n)$



for ex: Max flow instance : $O((m+n)\log n + m\log U)$ (Graph; Capacities of edges)

for ex: Subset Sum: $0 \le a_i \le W$ & $1 \le i \le n$.
$\therefore$ # of bits $= O(n \log W)$.

Now, what does 'polynomial in length of input' mean?
Let $|I|$ is the length of input.
$\therefore$ $|I|^k$ is polynomial in length of input. (for fixed k).

** $2^{|I|}$ is not a polynomial in $|I|$ because $\nexists$ k s.t.
$2^{|I|} \le |I|^k \quad \forall |I| \ge 0$.

Now, for topological sort : Running time is $O(m+n)$
Is it a polynomial running time algo?
$m+n \le (m+n)\log n)^k$
For $k=1$ this is true
$\therefore O(m+n)$ is Polynomial Running Time.

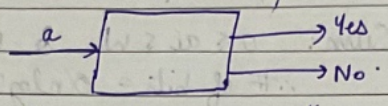
for Max flow : $n^3 \le (O((m+n)\log n + m\log U)^k$ → true for $k=3$
$\therefore O(n^3)$ is Polynomial Running Time.

for Subset sum : $nW \le (n \log W)^k$
$\log nW \le k(\log n + \log\log W)$

$$k \ge \frac{\log n + \log W}{\log n + \log\log W}$$

So, as we increase W, RHS will keep increasing and k will go upto ∞.
$\therefore$ No fixed k exists.
$\therefore$ $O(nW)$ is not a Polynomial Running Time.

For Primality Testing: determining whether the number is prime.



$|I| = \log n$ for representing.

and Trivial algorithm has a running time of $O(\sqrt{n})$

$$\sqrt{n} \leq (\log n)^k$$

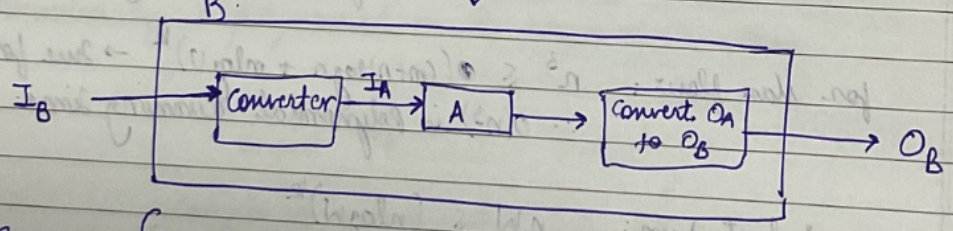
$$\frac{1}{2}\log n \leq k \log\log n$$

$$k \geq \frac{\log n}{2 \log\log n}$$

as n increases k will go to ∞

$\therefore O(\sqrt{n})$ is not a Polynomial Running time.

→ Polynomial Running time Reductions



For ex;

B = Project selection.

I_B = $G(V,E)$ and value given for each vertex.

A = Max flow problm.

I_A = $G'(V', E')$; source, sink, $c: E \to R^+$.

O_A = s-t cut of min capacity.

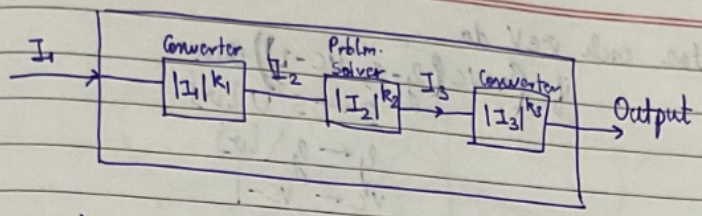
O_B = Projects corresponding to vertices in T of the min. capacity s-t cut.

We can now say that since for A we had a Polynomial time algo therfore, we can say that B also has a Polynomial time algo because B can be reduced to A. i.e. ∃ a reduction from B to A.

Optimization Problm

Decision Problm

So,



We can say that $|I_2| \leq |I_1|^{k_1}$

$|I_3| \leq |I_2|^{k_2} \leq |I_1|^{k_1 k_2}$

$\therefore$ Total time taken $= |I_1|^{k_1} + |I_2|^{k_2} + |I_3|^{k_3}$

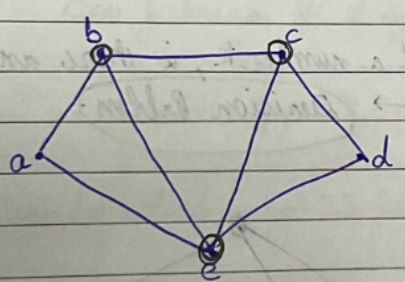
$\leq |I_1|^{k_1} + |I_1|^{k_1 k_2} + |I_1|^{k_1 k_2 k_3}$

Total time $\leq O(|I_1|^{k_1 k_2 k_3})$

$\therefore$ we can say that the procedure is Polynomial time by using it reduction.

→ **Vertex Cover Problem** (Optimization prblm)

Given $G(V, E)$; A set $S \subseteq V$ is a Vertex Cover if all edges of E are incident to a vertex in S.



In the ex., $\{b, c, e\}$ is a vertex cover.

Optimization Prblm – Given $G = (V, E)$ find a Vertex Cover of smallest size → Min-VC prblm.

Decision Prblm – K-VC prblm: given $G = (V, E)$ and an integer k, does the graph has a VC of size $\leq k$.

Lets say we have a routine to solve the k-VC prblm.
How can we use k-VC routine to compute the min-VC set?
Firstly we will do a Binary search from 1 to n to find the size of the min-VC by checking for each value whether a VC exist or not using k-VC routine.
Lets say we found the size of min-VC and say it is (VC).

Now, for each $v \in V$ do

if $(k\text{-VC}(G-v, VC-1))$ then

$S \leftarrow S \cup \{v\}$

$G \leftarrow G \setminus \{v\}$

$VC \leftarrow VC-1$

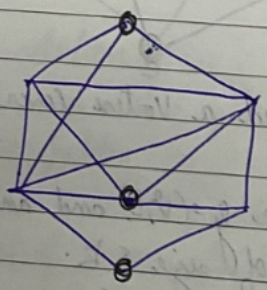
For each vertex we remove it and remove the edges incident to it ~~and that means the resⁿ graph will have x~~ and then we check whether the resⁿ graph has a vertex cover of size VC-1 ~~i.e.~~ and if there exists such a VC in the residual graph then it means that the selected vertex was part of the ~~min~~ VC set otherwise ~~not~~ it wasn't.

→ **Independent Set.**

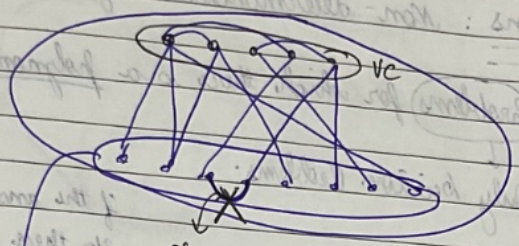
$G = (V, E)$; $S \subseteq V$ is an independent set if $\forall\, u, v \in S$, $(u, v) \notin E$.

Find the Maximum Independent Set in G → (Optimization Problem)

Given $G = (V, E)$ and a number k, is there an independent set of size $\geq k$ in G → (Decision Problem)

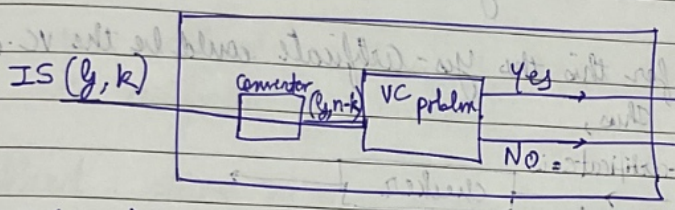


We can use the VC problem to solve the MIS problem and vice versa.



These can't exist because ~~other~~ if they do then one of these vertices should be in VC.

Independent set = Complement of Min. VC.



IS (G, k) ⇒ Does G has an independent set of size ≥ k

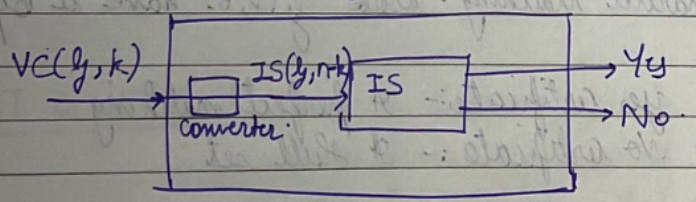
⇓

Does G has a VC of size ≤ n−k

⇓

VC(G, n−k).

similarly, VC can be solved using IS prblm.



VC(G, k) ⇒ Does G has a VC set of size ≤ k

⇓

Does G has an IS of size ≥ n−k

⇓

IS(G, n−k).

NP Problems : Non-deterministic polynomial time problem.

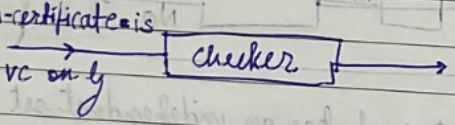
- class of (problems) for which there is a polynomial time checkable Yes certificate.
  - Only Decision Problems.

if the answer for the problem is yes then can it be ~~shown~~ verified in polynomial time by checking the yes-certificate.

ex: Vertex Cover: Does G have a VC of size $\leq k$?

So, for this the yes-certificate could be the VC on the graph G.

Thus,



ex: Independent set: Does G have an IS of size $\geq k$?

So, for this the yes-certificate would be the IS only of size $\geq k$ which will be then checked in polynomial time.

ex: Bipartite Matching: Does G(V, E) have a Bipartite matching?

yes certificate:- A perfect matching
No certificate :- A Hall set

→ NP

- A

- CN

∴

time Problem.

time checkable
Yes certificate.

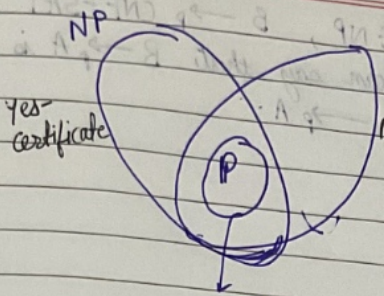
for the problem is
verified
it be ~~chosen~~ in
ne. by checking the
e.

IC. on the graph G.

k?

only if $c_{ij} \geq k$
al time.

e matching?



set of Polynomial time algorithms.

ex: Primality : Is n a prime?

yes-certificate:- Not easy to think. but there is one.
No-certificate:- Give a divisor.

→ NP Hard : atleast as hard as any problem in NP.

- A problem ~~is~~ A is NP Hard if ∀ problems $B \in NP$, $B \leq_p A$

i.e. B can be reduced to A in polynomial time.

so, if A can be solved in Polynomial time then all problems $B \in NP$ can also be.

$\therefore \forall B \in NP, \; B \xrightarrow{p} A$

- CNF-SAT is NP Hard: given a boolean formula in conjuctive normal form (CNF) check if ∃ an assignment of values to variables so that the formula is satisfiable (SAT).

∴ If we have to show that a prblm A is NP-hard then we have to show that CNF-SAT can be reduced to A.

$CNF\text{-}SAT \longrightarrow_p A$.

∵ We know $\forall B \in NP$, $B \rightarrow_p$ CNF-SAT is true

∴ Now we can say that $B \rightarrow_p A$ is true.

∴ $B \rightarrow_p$ CNF-SAT $\rightarrow_p A$.