Tel. : 91-11-27024524
: 91-11-27023128
Fax : 91-11-27023154

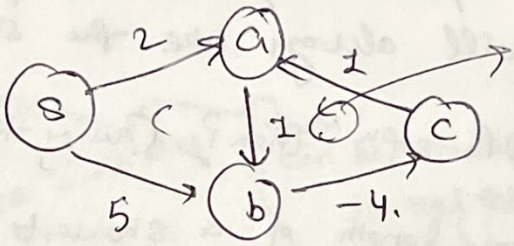
→ Djkstra's does not work with negative edge weights! (even with cycles with -ve edge wts, it will be problematic).

When cycles: -

→ dist(v) goes to $-\infty$ for every v on the cycle (a, b, c, a)

⟷ no soln to shortest paths when -ve cycles.

⟹ need to detect -ve cycles!



cycle with -ve edge wts!
(dist will keep on decreasing!)

So, we first detect this, if not found, then we find shortest path!

Properties of shortest paths :-

Suppose the problem has a soln for an input graph.

→ Can there be -ve cycle in the graph? (No)

→ Can there be +ve " " - " ? (Yes)

→ Can the shortest paths contain +ve cycles? (No)

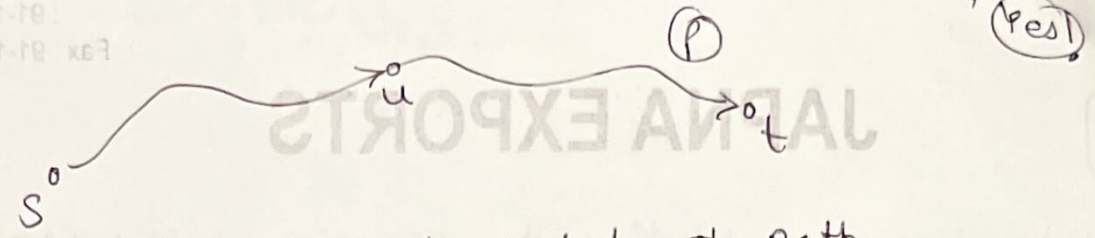
→ Consider a simple shortest-st path; are its subpaths shortest? → (Yes)

for a simple shortest path, can we have more than n-1 edges if n is the no. of vertices from $s \to v$? (No).

$\leq n-1$ edges in shortest path!

REGD. OFFICE : 186, SAINIK VIHAR, PITAM PURA, DELHI-110034
E-MAIL : japna@del3.vsnl.net.in
Visit our web site : www.japnas.com

Does problem exhibit optimal substructure? → Yes!



Let P be simple shortest st. path, that goes thru. 'u':

$$OPT(s,t) = OPT(s,u) + OPT(u,t)$$

↳ for optimality of full path, all the sub-paths must be optimal.

$\gamma > OPT(s,u)$

(*) any dist of any path b/w s & u, it will always have shorter than γ, then, we will always take the shortest path.

(*) Can we come up with a DP soln then? (using the fact above about OPT).

from s-v → path.

Let $OPT(i,v)$ = length of a shortest s-v path that was $\leq i$ edges.

(try & use more & more edges to see whether we can find the shortest path).

{ Subproblem that expresses this! }

$OPT(n-1,v)$ → as we can to at-most $(n-1)$ edges to see what can be the shortest path.

→ We wants $OPT(n-1,v)$, $\forall v \in V$ → "all set of vertices".

↙ max.

$(n-1)$ → number of edges in complete graph G.

Boundary cond^n

$$OPT(0,v) = \begin{cases} 0 & \text{(as no. edges are being considered), if } v = s \\ \infty, & \text{if } v \neq s \end{cases}$$

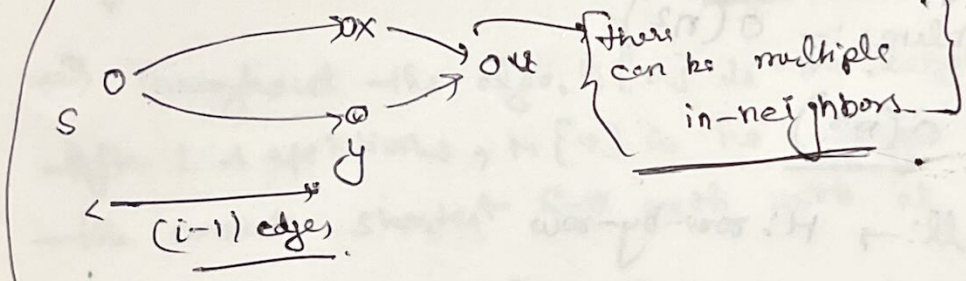
$$OPT(i,v) = \begin{cases} 0, & \text{if } v = s \\ w_{sv}, & \text{if } \exists (s,v) \in E \\ \infty, & \text{otherwise.} \end{cases}$$

Progressively we move forward ...

Tel. : 91-11-27024524
: 91-11-27023128
Fax : 91-11-27023154

Now, we want to find a recurrence,

$OPT(i,v) = \begin{cases} OPT(i-1,v) & \text{if shortest } s\text{-}v \text{ path that uses } \le i \text{ edges that actually uses } \le i-1 \text{ edges.} \\ \min_{x \in (x,v) \in E} \{ OPT(i-1,x) + w_{xv} \}, & \text{if shortest } s\text{-}v \text{ path that uses } \le i \text{ edges actually uses } i \text{ edges.} \end{cases}$



bcoz we don't know which one to take, we will consider minimum of those two!

Recurrence

$$OPT(i,v) = \min \begin{cases} OPT(i-1,v) \\ \min_{x=(x,v) \in E} \{ OPT(i-1,x) + w_{xv} \} \end{cases}$$

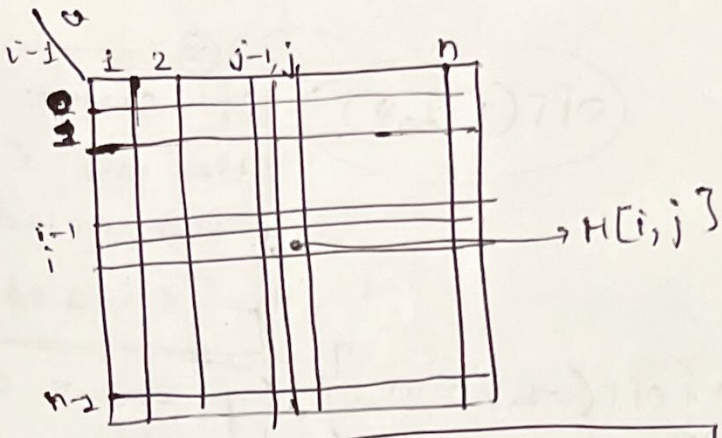
$0 \le i \le n-1$, $1 \le v \le n$ $\Rightarrow i * v \Rightarrow$ {$O(n^2)$ subproblems.}

$\Rightarrow$ As max^m no. of in-neighbors of v can possibly be n (no. of vertices)

$\therefore$ Time for each subproblem $\Rightarrow$ $O(n)$ [$\because OPT(i-1,v)$ is constant time, we already computed that]!

(*) Time to fill in all the sub-problems ⇒ $O(n^3)$ time (dense graphs)

for filling a matrix ⇒ $M[i,v] = OPT[i,v]$.



We want: $M[n-1, v]$, $\forall v \in V$

Time per subproblem :- $O(n)$

subproblems :- $O(n^2)$

Total time: $O(n^3)$

→ Order to fill :→ M : row-by-row

→ Total space :- $O(n^2)$. (we are only considering 2 rows at a time)
↳ $O(n)$ time

∵ we only consider in-degree of j,

∴ for $\min_x \{OPT(i-1, x) + w_{xv}\}$ → $O(\deg(v))$

∴ time :- $O(\deg(j))$

(for tighter analysis)

∴ for entire row (i) ⇒ $O(\sum \deg(j))$
$= O(m)$ all the edges.

Time to fill in a row ⇒ $O(m)$

∄ n rows in M ⇒ Total time :- $O(nm)$ Ans

Tel. : 91-11-27024524
: 91-11-27023128
Fax : 91-11-27023154

→ To reconstruct actual shortest paths, also keep array prev of size n such that,

{ prev[v] = predecessor of v in current shortest s-v path. }

Improving space reqs:-

Only need two rows of M at all times.

→ Actually, only need one. Thus, we can drop the index i from M[i,v] and only use it as a counter for # repetitions.

(∵ 1-D array)

⇒ Throughout the algo, M[v] is the length of some s-v path. After i repetitions, M[v] is no larger than the length of the current shortest s-v path with at most i edges.

Early termination cond^n:- If at some iteration i, no value in M changed, then stop! (This can be used to find if there are any -ve cycles in the graph) ⇒ after completed the flow and we have found the shortest path (n-1) iterations. If we run it for 1 more iteration, technically it shouldn't update! If anything is updated, this would show presence of -ve cycles (∵ path size can then grow upto $-\infty$), → would never end in terms of no. of iterations.

Thus, single loop of bellman-ford should work!

Pseudocode:-

$n \times n$ dynamic prog. table M s.t.

$M[i,v] = OPT(i,v)$

$BF(G=(V,E,w), s \in V)$

for $v \in V$ do

$M[0,v] = \infty$

end for

$M[0,s] = 0$

for $i = 1, \ldots, n-1$ do

for $v \in V$ (in any order) do

$$M[i,v] = \min \left\{ M[i-1,v], \min_{x:(x,v)\in E} \{ M[i-1,x] + w(x,v) \} \right\}$$

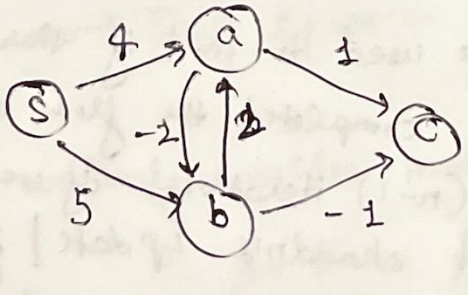
end for

end for.

order should not change!

we can remove this for space conditions!

because this would only consider prev. array formed!



M	i \ v	s	a	b	c
	0	0	∞	∞	∞
	1	0	4	5	∞
	2	0	4	3	4
	3	0	4	3	2

for using i as just repetitions, we form 1-D array!

(converts into this!)

	s	a	b	c
i=0)	0	∞	∞	∞
i=1)	0	4	3	2

this is the same as 1-D array for distance

Tel. : 91-11-27024524
: 91-11-27023128
Fax : 91-11-27023154

Pseudocode becomes:-

```
Initialize (G, s)
    for v ∈ V do
        dist[v] = ∞
        prev[v] = NIL
    end for
    dist[s] = 0
```

```
Update(u,v)
    if dist[v] > dist[u] + w(u,v) then:-
        dist[v] = dist[u] + w(u,v)
        prev[v] = u
    end if
```

s ⇒ source node

```
Bellman-ford (G=(V,E,w), s)
    Initialize (G, s)                          → O(n)
    for i = 1, ..., n-1 do                     } → O(m)   } → O(nm).
        for (u,v) ∈ E do                       }
            Update (u,v)
        end for
    end for.
```

∴ Time ⇒ O(nm)
∴ Space ⇒ O(n)

⇒ Floyd-warshall algo ⇒ (All pair shortest paths) with -ve edges.

Runs in $O(n^3)$

Ans.

REGD. OFFICE : 186, SAINIK VIHAR, PITAM PURA, DELHI-110034
E-MAIL : japna@del3.vsnl.net.in
Visit our web site : www.japnas.com

Discrete Random variables :-

takes finite no. of values with same probability :-

$$E[X] = \sum_j j \cdot Pr[X=j]$$

Bernoulli trial :-

→ flip a biased coin :-
- → heads with prob. (p.)
- → tails with prob. (1-p)

Qn :- what is the expected no. of heads?

Let $X \Rightarrow \begin{cases} 1, & \text{if coin flip comes up } (H) \\ 0, & \text{if } \sim\sim\sim\ (T) \end{cases}$

$$\Rightarrow E[X] \Rightarrow \sum_j j \cdot Pr[X=j] \Rightarrow 0 \cdot Pr[X=0] + 1 \cdot Pr[X=1]$$

$$= 0 \cdot (1-p) + 1 \cdot p = p \quad \text{Ans}$$

Indicator random variable :- a discrete rv. that only takes on values 0 & 1.

(denote occurence (or not) of an event).

FACT 1 ⇒ If X is an indicator rv., then $E[X] = Pr[X=1]$.

Exp- flip coin n times.

what is exp. no. of heads?

Let X be a rv that counts the # heads in n coin flips.

$$E[X] = \sum_{j=0}^{n} j \cdot \binom{n}{j} p^j \cdot (1-p)^{n-j}$$

{ value of that event } { Probability of that event }

o/p :- yes, iff. x has a perfect matching.

→ Encoding of a set S of abstract objects :-

Mapping of elements from S to binary strings.

Should we be concerned about the particular encoding used?

→ Yes!

Reasonable encodings :- (not unary encodings x).

→ Encoding of an integer is polynomially related to its binary representation (✓) 1024 → in binary but not (11 ... 1) (x). → not reasonable!
1024 bits

→ Encoding of a finite set is polynomially related to its "standard" encoding as a list of its elements.

{ —— , —— , }

Example of encodings :-

Input to sorting → → n elements

Standard encoding $\langle x \rangle$:- $\{x_1, x_2, \ldots x_n\}$

Length of standard encoding :- $|\langle x \rangle| = O(n)$

poly-time algo runs in :- $O(n^k)$ (in length of bit string that encodes the I/P).

Input to s-t connectivity →

Standard encoding → $\langle G, s, t \rangle$ →

$\{v_1 : \{v_{11}, \ldots, v_{1k_1}\}, v_2 : \{v_{21}, \ldots v_{2k_2}\} \ldots$

$v_n : \{v_{n_1} \ldots v_{nk_n}\}, s, t\}$

Length of standard encoding → $|\langle G, s, t \rangle| : O(n+m)$

poly-time algo runs in :- $O((n+m)^k)$ Any

Input to max flow :-

standard encoding ⇒ $\langle G, s, t, c \rangle$ ⇒ $U = \max_{e \in E} c_e$.

$O(nmU)$ & graph rep. as before!

Length ⇒ $|\langle G, s, t, c \rangle|$ ⇒ $O(n + m + m \log_2 U)$

for G for capacities.

poly-time algo runs in :-

$O((nm \log_2 U)^k)$

↳ for FF algo.

Now, coming to the decision problem ⇒

---

we have already designed algo that solves BPM ⇒

(1.) given :- bipartite graph G (I/P to BPM), we construct a flow N/W G' (I/P to max flow).

→ we transform the I/P instance x of BPM to an I/P instance y of max flow.

we can use this inely to answer the decision, basically if it equals (flow) the number of vertices in X & Y, the $|X| = |Y| = |f|$, then, return "yes", otherwise "no".

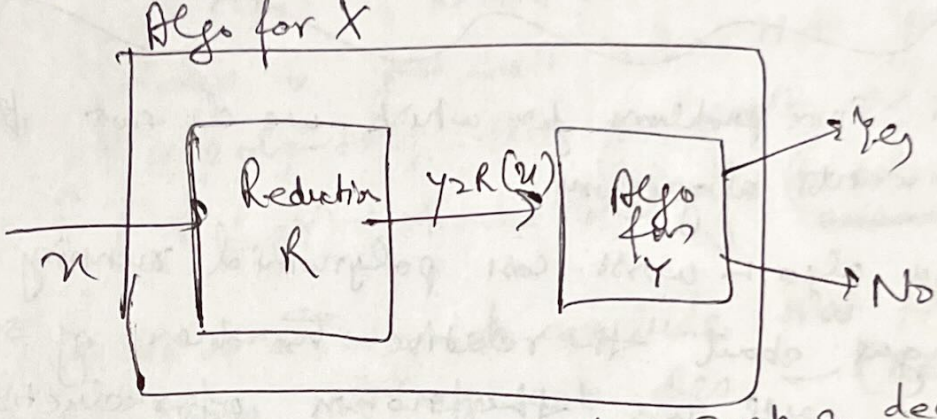
About Reductions ⇒

→ a transformation for every I/P x of X produces an equivalent input $y = R(x)$ of Y.

→ By equivalent → we mean that answer to $y = R(x)$ considered as an I/P for Y is a correct yes/no to x, considered as an I/P for X.

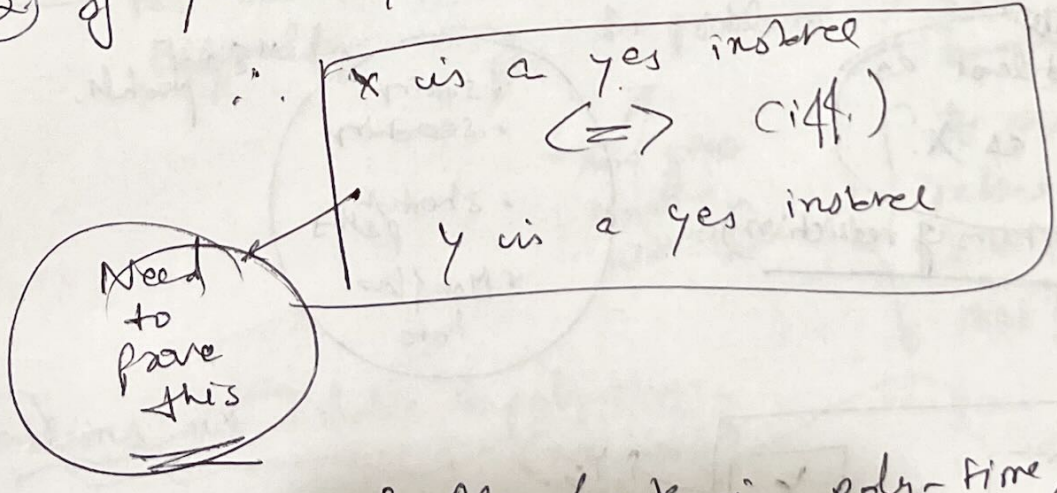
we will require that the reduction is completed in a polynomial time in $|x|$ number of computational steps.

If X & Y are computational problems,
x, y are I/Ps to X, Y respectively.



To prove equivalence of instances for such a decision problem :—

(1) If x is a yes instance, then y is a yes instance.
(2) If y is a yes instance, then x is a Y instance.



If both Red^n & Algo for Y is poly-time, then algo for X is considered to be an efficient algo.

Thus, we say that X reduces polynomially to Y;

$$X \leq_p Y$$

If Y is solvable in poly-time, then X is solvable in poly-time (∵ red^n has to poly-time)

NOTE: To solve BPM we made exactly one call to the black box for max flow. This is mostly same for our cases!

(Reduction provide a powerful technique for designing efficient algos.)

→ There are some problems for which we do not know of any efficient algorithms.

→ Effic algos → worst-case polynomial running time.

→ To argue about the relative hardness of such probs, we will use the notion of reductions!

Questions:

Suppose $X \leq_p Y$ (prob X reduces polynomially to prob. Y).

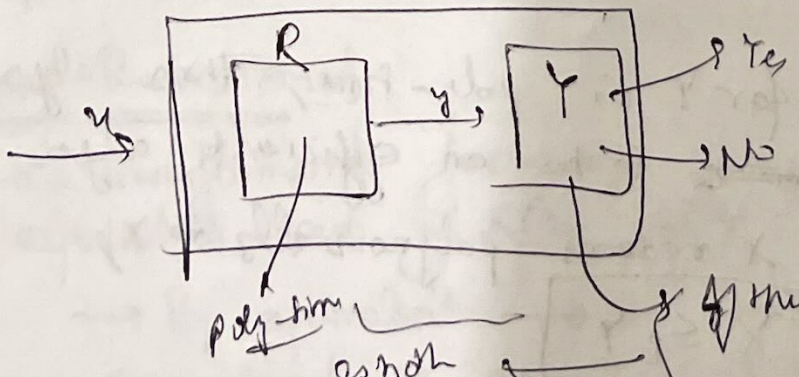
Y is at least as "hard" as X.

(instead of reduction)

Why?

"Easy" probs.

- sorting
- searching
- shortest paths
- Max flow
- etc



poly-time

as both give us time for X, this above fact holds!

If this is poly-time, then X is also poly-time,

if this is super polynomial then X is the same

∴ Suppose $X \leq_p Y$. If Y is solvable in polynomial time, then X is solvable in polynomial time.

Contrapositive of fact above:

If $X \leq_p Y$ & X is not solvable in poly-time, then Y is not solvable in poly-time.

Thus, if you have a known problem X that is non-solvable & you can't reduce it to your problem Y in poly-time, then you can't solve Y as well.

{ $(X \leq_p Y) \rightarrow$ should hold. } → to solve Y

(i) This provides us a way to conclude that problem Y does not have an efficient algo. → {Red^n helps us to establish this relatively}

(ii) For problems (discussed), we have no proof that they do not have efficient algo

→ fact above can be used to establish relative level of difficulty among the problems (thru. Reductions)

This gives us the answer!

{ Is there a way to reduce the prob. $X \rightarrow Y$ & $Y \rightarrow X$ or not?

Problems not solvable in poly-time :- (not proved)

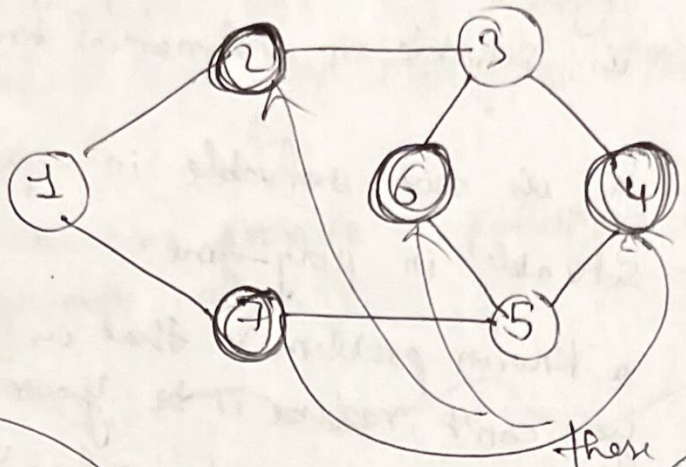
Independent Set :-

→ An independent set in $G = (V, E)$ is a subset $S \subseteq V$ of nodes s.t there is no edge b/w any pair of nodes in the set.

That is, for all $u, v \in S$, $(u, v) \notin E$.

→ easy to find small IS but difficult to find a larger one!

Eg:-



{Max^m Independent set problem}

these form the max^m independent set,

∴ we have to go over each diff. pair of vertices to check, this takes, 2^n time ⇒ $\Omega(2^n)$

(lower bound of 2^n)

(exponential time!).

Another problem:-

Vertex Cover ⇒

A vertex cover in $G = (V, E)$ is a subset $S \subseteq V$ of nodes s.t every edge $e \in E$ has at least one endpoint in S.

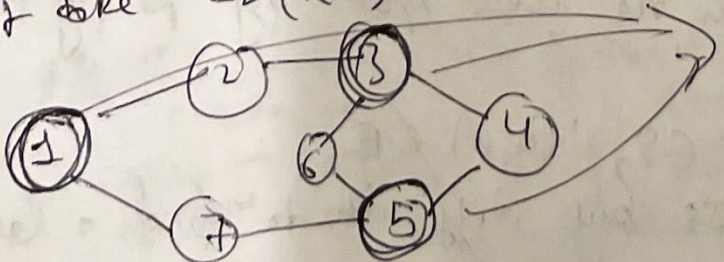
(sets of vertices that covers up all the edges)

↳ another way of looking at this.

→ Easy to find a large vertex cover (∵ it always equals set of all the vertices V).

But it is really hard to find a small one.

⇒ It take $\Omega(2^n)$ time as well.



this is the min^m vertex cover for this graph G_1.

(hard to find!)

Relationship b/w vertex cover & independent set ⇒
If S is a vertex cover, then $V - S$ is the independent set.
(complement of each other)

Optimization versions of IS & VC ⇒

Max^m IS ⇒ Given G, find an IS of max^n size.
Min^m VC ⇒ Given G, find a VC of min^m size.

(Requires exponential time :- 2^n candidate subsets
∴ every vertex may/may not belong to such a subset)

Now, we can transform these ~~decision version of~~ optimization problem into decision versions (to try & solve these) ⇒

→ An optimization problem may be transformed into a roughly equivalent problem with a yes/no answer, called the "decision version" of the optimization problem by ⇒

(1) supplying a target value for the quantity to be optimized,
(2) asking whether this value can be attained.

→ denoted by $X(D)$ if prob. was X.
↳ decision version of prob X.

Examples of decision versions of optim. probs ⇒

→ Max flow (D) :- Given a flow N/w G and an integer K (target flow), does G have a flow of value at least K?

→ Max Bipartite Matching (D):- Given a bipartite graph G & an integer k, does G have a matching of size at least k?

→ IS(D):- Given a graph G & an integer k, does G have an independent set of size at least k?

→ VC(D):- Given a graph G & an integer k, does G have a vertex cover of size at most k?

So, in a max. prob. the target value is a lower bound, while in a min. prob, it is an upper bound.

→ Shortest s-t path (D):-

Given $(G = (V, E, w), s, t)$ and k, is there an s-t path in G of length $\leq k$?

→ Min cost sequence alignment (D):-

Given (X, Y, δ, α) costs & k, is there an alignment of X, Y that costs $\leq k$?

→ Knapsack(D):-

Given $(n, v_1, \ldots, v_n, w_1, \ldots, w_n, W)$ max w possible

& k, is there a subset of the item with weight $\leq w$ & value $\geq k$?

(at least k ∵ this was a max. prob.)

⟹

Now, lets try and convert an optimization problem to its decision version:-

Max^m Independent set problem (max IS):-

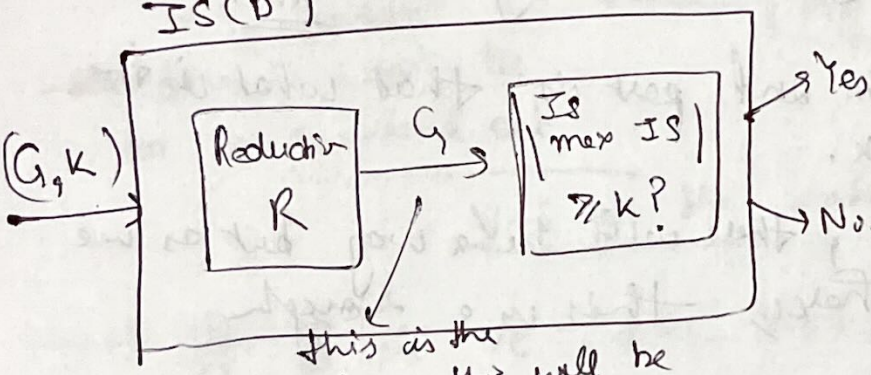
Its decision version denoted by IS(D).

convert this, we first need to prove that these 2 instances are equivalent, i.e, both IS(D) and max IS follow the fact $X \leq_p Y$ (whether they can converted in an equivalent manner) →

(1) → $IS(D) \leq_p$ max IS

(using reduction, we can convert)

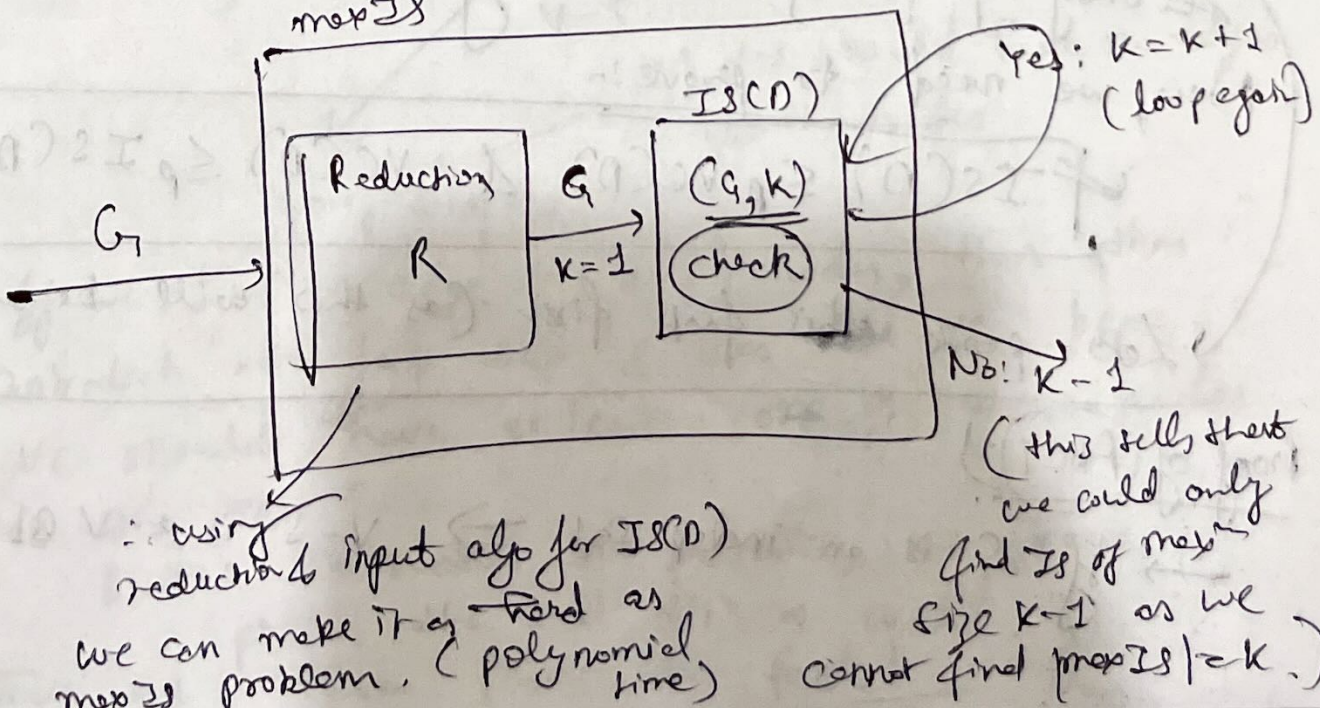
→ Intuitively, it makes sense that max the value of independent set and finding k is at least as hard (actually harder) than when we just need to tell whether an IS of size k is present or not in the I/P graph G.



(2) → max IS $\leq_p$ IS(D)

(using reduction)

(In this, calls are made again & again to IS)



NOTE: for max flow problem → nU → reduction will be of this size.

what we can do ← (this is not poly-time).

is search the search-space from $k \to$ ~~max~~ max k using binary search →

$$\log_2(nU) \to \log n + \log U$$

this becomes polynomial time after this. ═══ Ans.

What we are missing in 2nd part is that what is the set, we only return k.

To reconstruct the set, there will be a way but as we don't prove that here, this is a rough approximation!

We have proven the equivalence!

Reduction from Independent Set to VC:-

FACT: Let $G=(V,E)$ be a graph. Then $\{S\}$ is an IS of G if & only if $\{V-S\}$ is a VC of G.

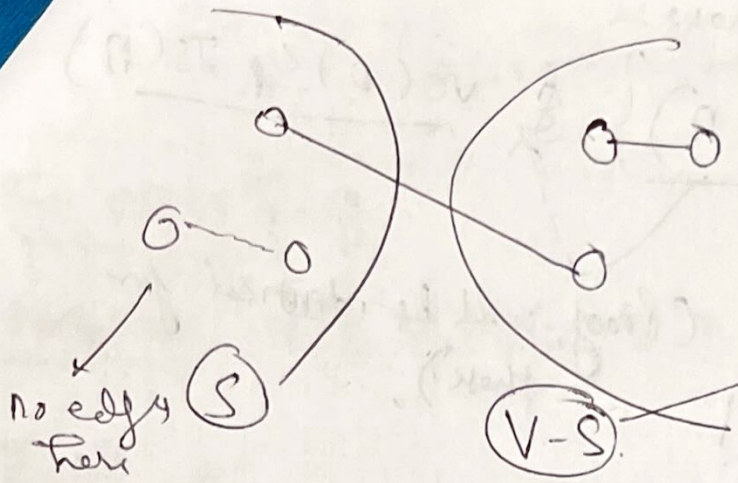
Now, we need to prove:-

$$IS(D) \leq_p VC(D) \;\&\; VC(D) \leq_p IS(D)$$

→ Lets prove this fact first (as this will be used)

Proof of FACT:

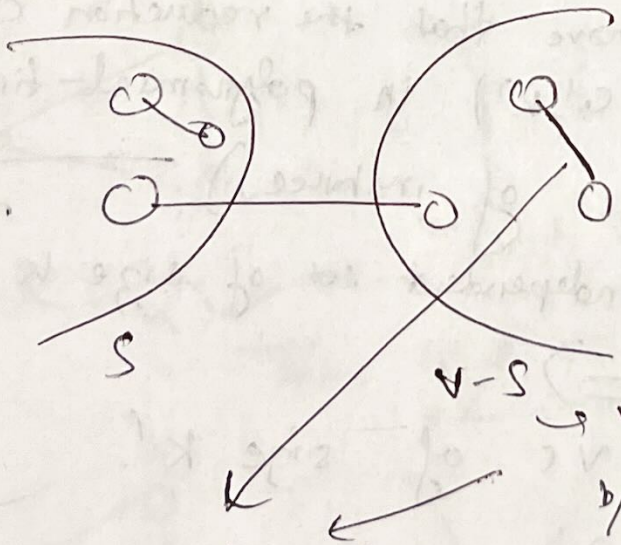
→ If S is an indep. set $\Rightarrow$ $V-S$ is a VC.



If we find an edge in nodes in S, then S cannot be an independent set.

∴ V-S is the vertex covers.

→ If S is VC, then V-S is an IS.



going contrary,

If we find an edge in {V-S} nodes, then this contradicts the fact that S was the VC bcoz, VC should have at least one of those nodes in the set S.

As this contradicts the premise, we have proven the 2nd dir" as well.

Thus, we have proven the fact! ∎

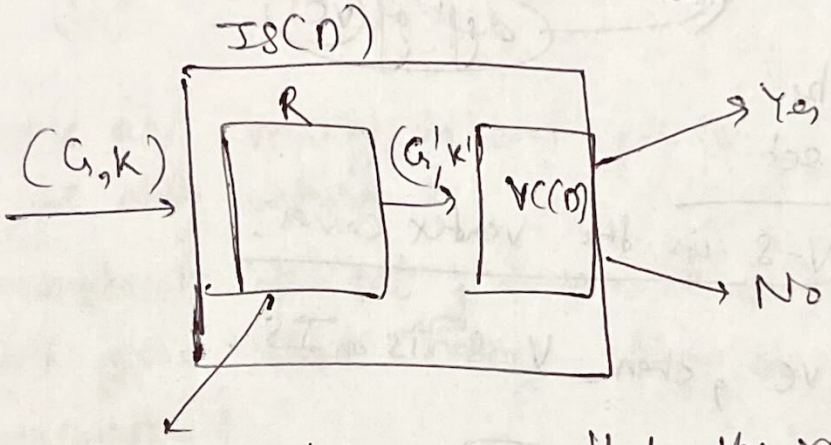
Let's prove the claim now:-

$\{IS(D) \leq_p VC(D)\}$ & $VC(D) \leq_p IS(D)$

(Proof will be identical for these).

we will try & prove this!

for IS problem $\Rightarrow$ $IS(D) \leq_p VC(D)$



Thus, we need to prove that the reduction can convert $(G, K) \longrightarrow (G', K')$ in polynomial-time.

s.t. (equivalence of instances)

G has an independent set of size K
$\Leftrightarrow$
G' has a VC of size K'. (meaning)

(Need to prove this!)

we know by the fact proven before that :-

$G = G'$ → same

$k' = n - k$ (proven before)

Thus, the instances are equivalent!

Proof completed

(As we are not changing G & const time for finding k')

thus, Poly time reduction.

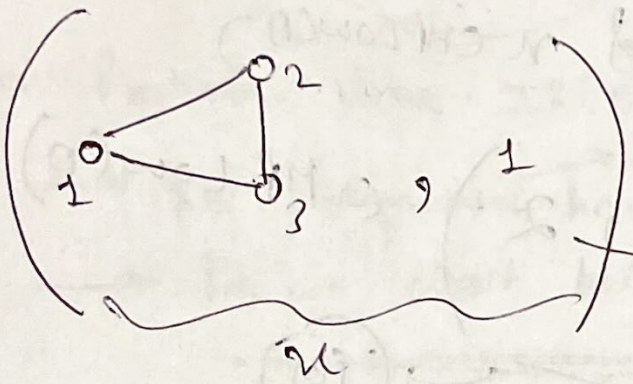
Ans

∴ IS & VC are of the same hardness.

The class P :-

Notation :- $x \in X(D) \iff x$ is a "yes" instance of $X(D)$.

Eg :-

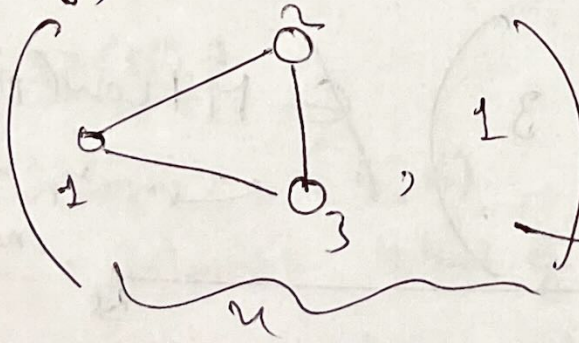


$\in IS(D)$

→ can we get an IS of size 1 on graph G
↳ "yes"

thus $x \in IS(D)$.

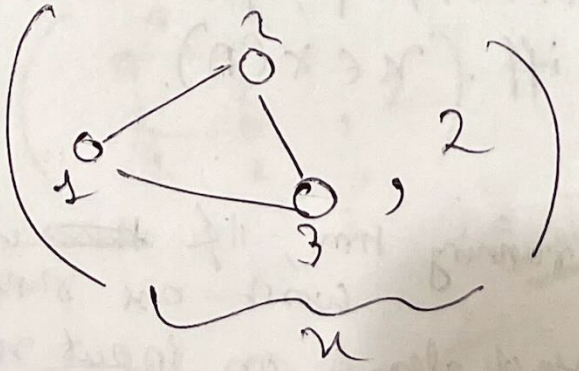
Similarly,



$\in VC(D)$

→ This is a "no" instance.

Thus $x \notin VC(D)$.



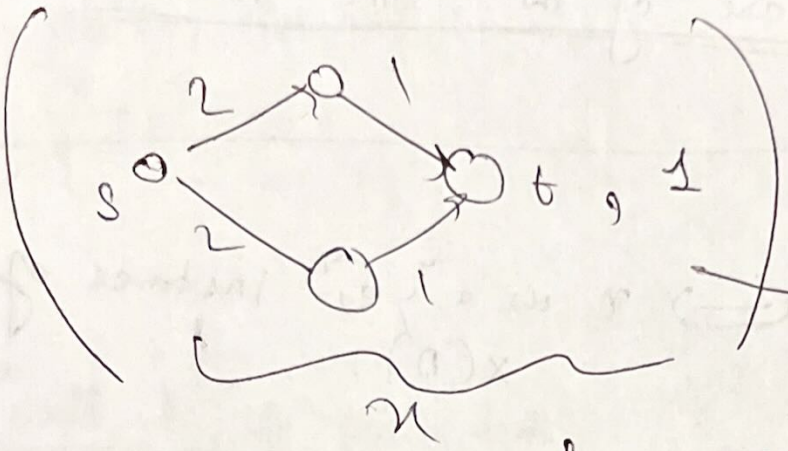
$\in VC(D)$

→ (Yes)

$x \in VC(D)$.

Similarly,

for max-flow



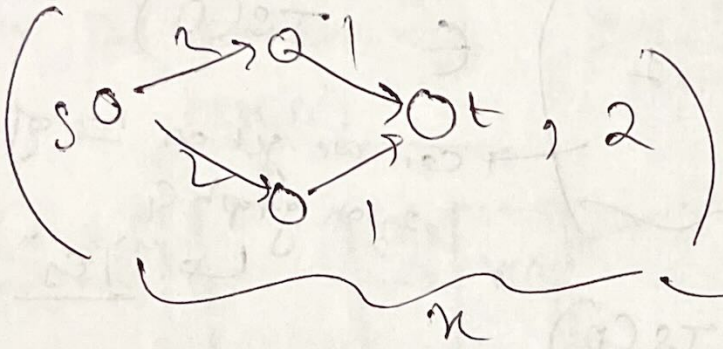
$\in$ MFlow(D)

→ flow of 1 present?

Yes

x

⇒ $x \in$ MFlow(D)

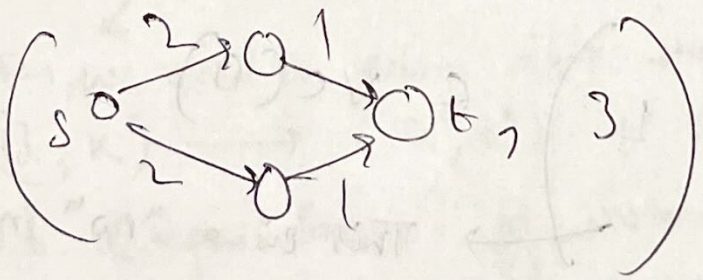


$\in$ MFlow(D)

x → Yes

⇒ $x \in$ MFlow(D)

But



$\notin$ MFlow(D)

→ "no" instance

For correctness ⇒

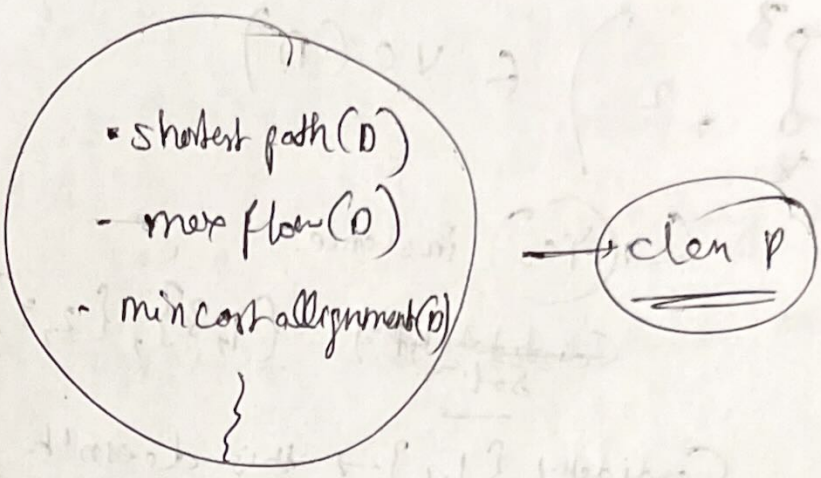
→ An algo A solves X(D), if, for all input x; A(x) = yes iff. $x \in X(D)$.

For efficiency ⇒

→ A has a poly running time, if worst-case running time of algo A on input x is $O(|x|)$

Then, we define P to be the set of decision problems that can be solved by poly-time algor.

P →



⇒ for Problems like IS(D) & VC(D) :-
(worst-case instances).

→ No polynomial time algo has been found!

→ So, we don't believe they are in P.

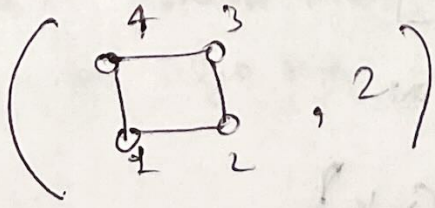
Q → Is there anything +ve we can say about such problems?

⇒ IS(D) →

→ Input :- (G, K)

→ Solution :- subset of nodes that form an IS of size $\geq k$.

→ Candidate solution :- subset of nodes.



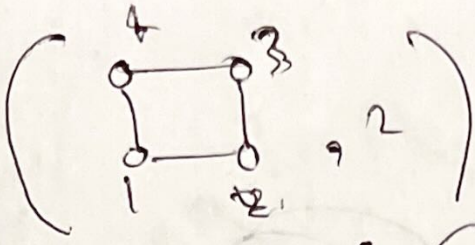
$\in$ IS(D) → Yes

a Candidate → {1,3}, {2,4}

Is {1,2} a sol^n? → "No" instance

(b) Just on basis of a "no" candidate sol^n, you can't say where it is a "no" instance.

⇒



$\in VC(D)$

↳ (Yes) instance.

~~Candidate Sol~~ $\{1,3\}, \{2,4\}$.

Candidate $\{1,2\}$ → this doesn't give us "Yes".

for both these problems :-

CERTIFICATE t for input x :-

Short Sol^n that CERTIFIES that $x \in X(D)$.

Certificates are PROPERTIES of "Yes" instance ∀

$x \in X(D) \iff \exists$ short t for X.

↳ poly-time

(*) Can define a complexity class!

NP ∀ { set of decision problem where Yes instances possess short certificates. }

∀ Verification algo. or Certifier :-

→ Input to $IS(D)$:- $x = \langle G, k \rangle$

→ Input to certifier for $IS(D)$:- (x, t) ← certificate

→ What does the certifier do with its I/P ?

(1) $\geq k$ nodes in t (2) t forms an independent set.

Network flows:—

→ edges carry traffic; nodes → switches where traffic gets diverted!

(source → sink)

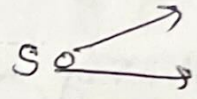
flow networks ⇒

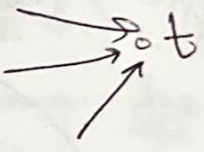
A flow network $G = (V, E)$ is a directed graph s.t.

1→ every edge has a capacity $c_e \geq 0$

2→ There is a single source $s \in V$.

3→ There is a single sink $t \in V$





Assumptions

A1:— integer capacities.

A2:— no edge enters s.

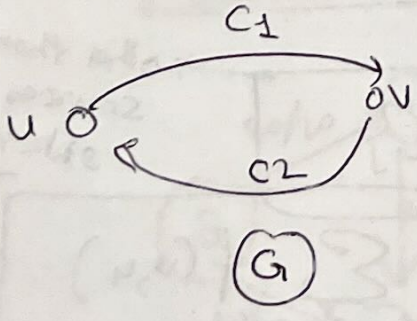
A3:— no edge leaves t.

A4:— If $(u,v) \in E$ then $(v,u) \notin E$.

A5:— Every $v \in V - \{s,t\}$ is on some s-t path.

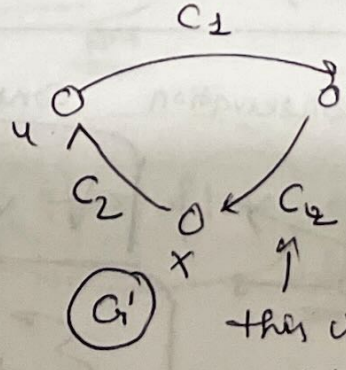
Hence, G has $m \geq n-1$ edges.

If we do have $\boxed{(u,v) \text{ \& } (v,u) \in E}$



G

⇒



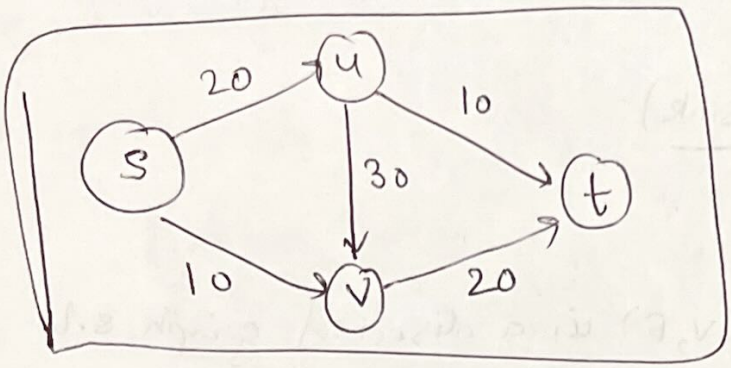
G'

(we would have to convert this).

this is upper bound of the flow thru an edge / path.

This transformation will be polynomial time!

→ An efficient transformation!



Flows:- given a flow network G, an s-t flow f in G is a func^{n}

$$f : E \longrightarrow R^{+}$$

Intuitively, the flow f(e) on edge e is the amount of traffic that edge e carries.

$$\vec{f} = \begin{bmatrix} \\ 2 \\ \vdots \\ m \end{bmatrix}$$

→ flow vector (for all the edges in the graph).

Constraints to satisfy:-

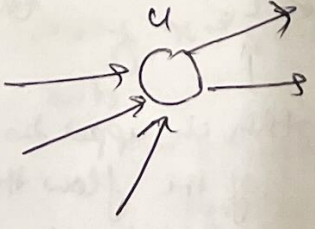
(1.) Capacity constraints.

$$\forall e \in E, \quad 0 \le f(e) \le c_e$$

(2) flow conservation constraints,

$$\forall v \in V - \{s, t\}$$

→ other than sources and sink nodes



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

$$f^{in}(u) \triangleq \sum_{e \text{ into } v} f(e) \; ; \; f^{out}(u) \triangleq \sum_{e \text{ out of } v} f(e)$$

Now, flow conservation constraints =>

$$\forall\ u \in V - \{s, t\},$$

$$f^{in}(u) = f^{out}(u)$$

Ans

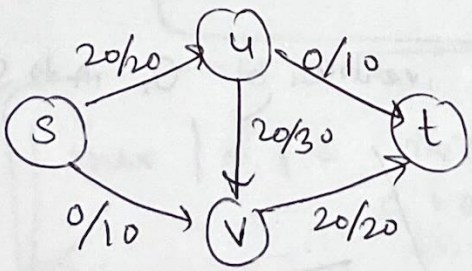
$$|f| \triangleq f^{out}(s) = \sum_{(s,u) \in E} f(s,u)$$

→ only out of the source.

Value of flow f => $|f| = f^{in}(t)$ (2nd part of flow-conservation)

(flow into the sink t)

Ans



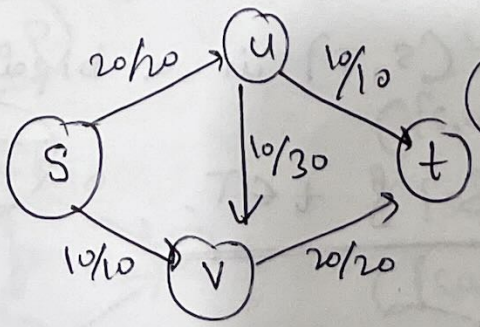
$$\vec{f} = \begin{bmatrix} 20 \\ 0 \\ 20 \\ 0 \\ 20 \end{bmatrix} \begin{matrix} su \\ sv \\ uv \\ ut \\ vt \end{matrix}$$

flow vector

(5 edges)

$$|f| = f^{out}(s) = f^{in}(t) = 20$$

Ans.



=> (Max) flow of value = 30.

(satisfying constraints)

$|f| = 30$

Max flow prob

Inputs: (G, s, t, c) s.t.

→ $G = (V, E)$ is a flow N/w.

→ $s, t \in V$ are source & sink.

→ c is the (integer-valued) capacity func^n.

O/P → a flow of maximum possible value.

(what prevents flow to be very large value).

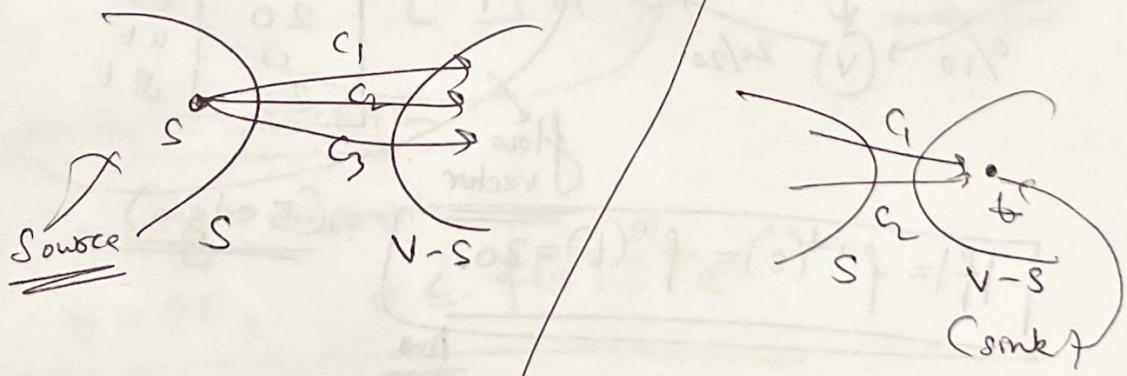
↳ capacity of the edges are the upper bounds!

2 upper bounds

$$|f| \le \sum_{e \text{ out of } s} c_e \quad \& \quad |f| \le \sum_{e \text{ into } t} c_e$$

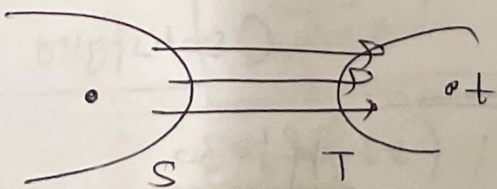
(1 more!)

Considering a bipartite graph of vertices of G into $S \& V-S$.



Def^n:- A s-t cut (S, T) is a bipartition of the vertices s.t. $T \supseteq V-S$

$s \in S$ & $t \in T$.



$$|f| \leq \sum_{\substack{e \text{ crosses} \\ \text{from } S \text{ to } T}} c_e \quad \text{(for any } S \& T\text{)}$$

$\triangleq c(S,T)$

(capacity of the cut (S,T))

for every (S,T), value of the flow cannot exceed the capacity of the cut,

$\forall$ s-t, cut (S,T)

$$|f| \leq c(S,T)$$

$$\therefore |f| \leq \min_{\substack{s\text{-}t \text{ cut} \\ (S,T)}} c(S,T)$$

min^m of all ~~cp~~capacity of the cuts formed.

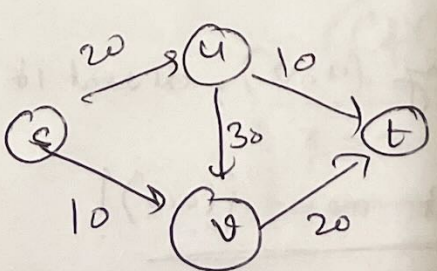
$$\therefore \max |f| = \min_{\substack{s\text{-}t \text{ cut} \\ (S,T)}} c(S,T)$$

max^m flow possible.

min-cut

we need an efficient algo. to find this ▷

s-t cuts?



(S,T)	$c(S,T)$	
$(\{s,u\},\{v,t\})$	50	
$(\{s\},\{u,v,t\})$	30	
$(\{s,u,v\},\{t\})$	30	
$(\{s,v\},\{u,t\})$	40	(only go from $S \to T$)

we are trying to find upper bound of the flow, thus, only consider flows from $s \to t$.

we always have, 2^{n-2} s-t cuts (for n vertices)

this is a lot of cuts to go-over to find max-flow!

Thus, we need an efficient algo. to compute!

Apps of

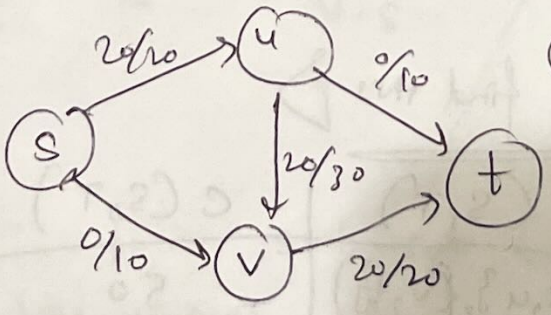
- → Bipartite matching
- → Airline sch.
- → Baseball elimination
- → Dist. of goods to cities
- → Image segmentation
- → Survey design

→ max-flow

→ min-cut

type of problem formulation

"Undoing" flow →



flow $f = 20$.

Goal:- undo 10 units of flow along (u, v), divert it along u-t.

(to get max-flow)!

To undo that (identify s-t path, s.t we can flow more amount, maybe from diff. dirns).

Identify an s-t path in G s.t — push flow forward to edges w. leftover capacity.

— push flow backward on edges that already carry some flow. ("Undo" flow)

Thus, we need diff. kind of Graph (G_f)!

Residual Graph ⇒

G_f w.r.t a flow network G and a flow f.

→ has the same vertices as G.

→ $\forall e \in G$, s.t. $f(e) < c_e$, introduce $e \in G_f$ with residual capacity $c_f(e) = c_e - f(e)$

leftover capacity.

(So, that we can saturate an edge).

It's strictly +ve number

FORWARD edges

→ $\forall e \in G$ s.t. $f(e) > 0$, introduce $e^r = (v, u)$ $\in G_f$ with residual capacity $c_f(e^r)$.

$c_f(e^r) = f(e)$ → to undo a flow.

$c_f(e^r) > 0$ → strictly +ve

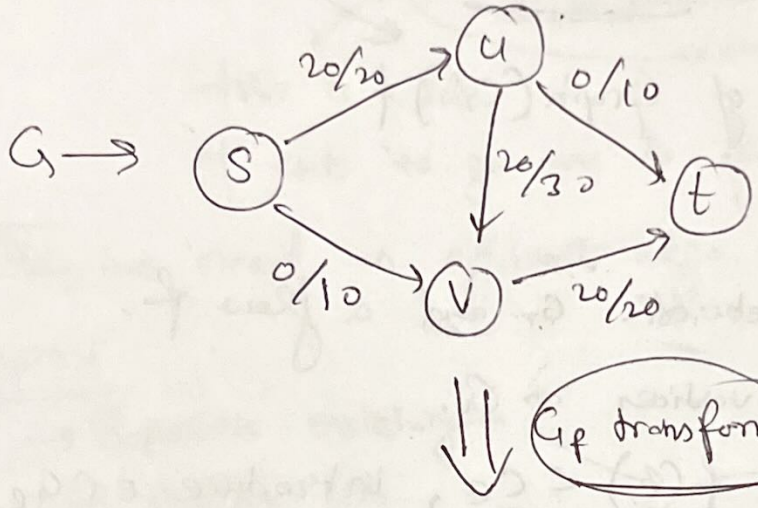
BACKWARD edges.

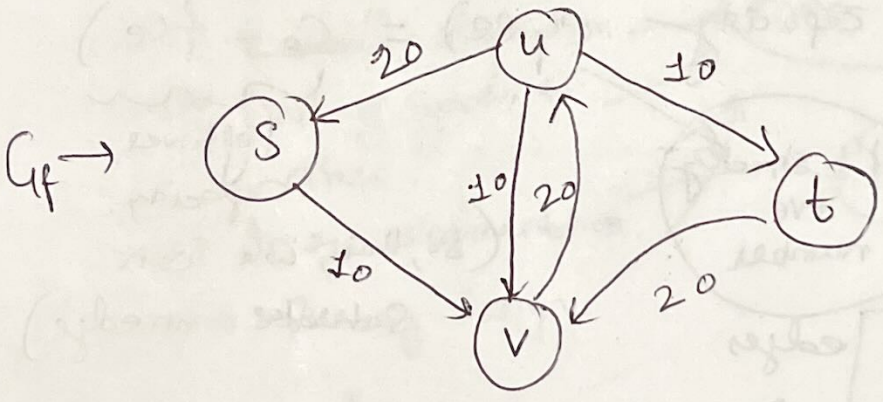
Thus, we have introduced both kinds of flow.

max^n no. of edges = $2|E|$ in G_f.

→ Thus, we generally don't take anti-parallel edges for analysis. (bcoz we need backward edges!)

Construction of G_f →





(not necessarily a flow N/w)

(there are anti-parallel edges here!)

(a roadmap to augment the flow).

→ We will use G_f as a roadmap to augment f in G.

Goal → Identify a simple s-t path in G_f.

— Then, we define the bottleneck capacity,

$$c(P) = \min_{e \in P} c_f(e)$$ → in residual graph.

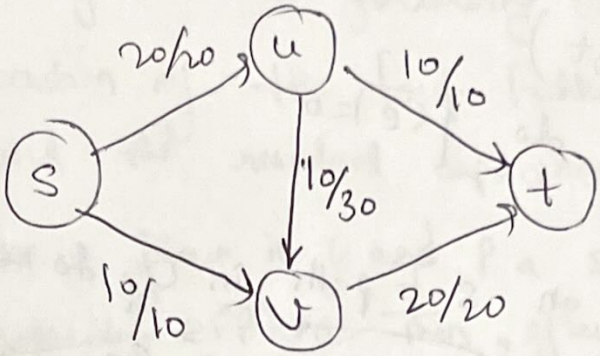
— Then, we update f in G as follows:

$\forall e \in P$, → If e is FORWARD, $f'(e) = f(e) + c(P)$

→ If e is BACKWARD, $f'(u,v) = f(v,u) - c(P)$

so, after updation,

G becomes G' ⇒



f' is the new flow

→ If $e \notin P$, $f'(e) = f(e)$.

Proof ⇒

we need to prove that both constraints are valid,

① capacity constraints ⇒ we don't exceed $c(e)$.

② flow conservation ⇒ $f^{in}(v) = f^{out}(v)$

Pseudocode of subroutine Augment

I/P ⇒ a flow f, and an augmenting path P in G_f

O/P ⇒ augmented flow f'.

Augment (f, P) ⇒

for each edge $(u,v) \in P$ do,

if $e = (u,v)$ is a forward edge then

$f(e) = f(e) + c(P)$

else

$f(v,u) = f(v,u) - c(P)$

end if

end for return f → new flow

ford-fulkerson Algo :)

FF ($G_1 = (V, E, C)$, s, t)

for all $e \in E$ do $f(e) = 0$

end for

while there is an s-t path in G_f do ←

Let P = simple s-t path in G_f

$f' =$ Augment (f, P)

Set $f = f'$

Set $G_f = G_{f'}$

end while

Return f.

jmp

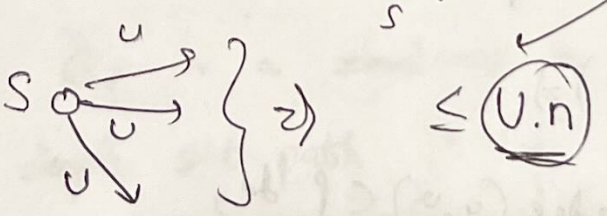
every vertex appears at-most once!

It ends in some finite steps!

⇒ Let $U = \max_{e \in E} c_e$

$\max |f| \leq$

$\therefore |f| = f^{out}(s) \leq \sum_{e \text{ out of } s} c_e \leq U$



$\max |f| \leq nU$ → this seems a finite upper bound?

(if we atleast inc. flow every time, then atmost nU (atleast 1) steps will be needed).

Thus, algo ends after finite # steps

increases by an integer amount after Augment(f, P).

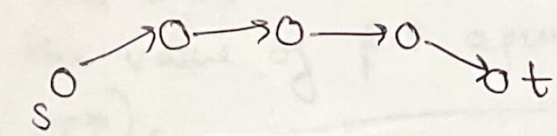
It follows from the following claims.

During execution of the ford-fulkerson algo., the flow values $\{f(e)\}$ and the residual capacities in G_f are all integers.

→ Let f be a flow in G and P a simple s-t path in G_f with residual capacity $c(P) > 0$. Then, after Augment(f, P)

$$|f'| = |f| + c(P) \geq |f| + 1 \quad \text{— (3)}$$

~~Let f be~~



(s, t) is FORWARD.

$f'(s,u) = f(s,u) + c(P)$ → capacity of the path "P".

$$|f'| = |f| - f(s,u) + f'(s,u)$$

$$\geq |f| + c(P) \geq |f| + 1$$

↳ bcoz capacity is strictly +ve.

(*) Running time of ford-fulkerson :-

(3) guarantees at most nU iterations.

- $O(m+n)$ to create G_f using adjacency list representation
- $O(m+n)$ to run BFS/DFS to find augmenting path.
- $O(n)$ for Augment(f, P) since P has at most $n-1$ edges.

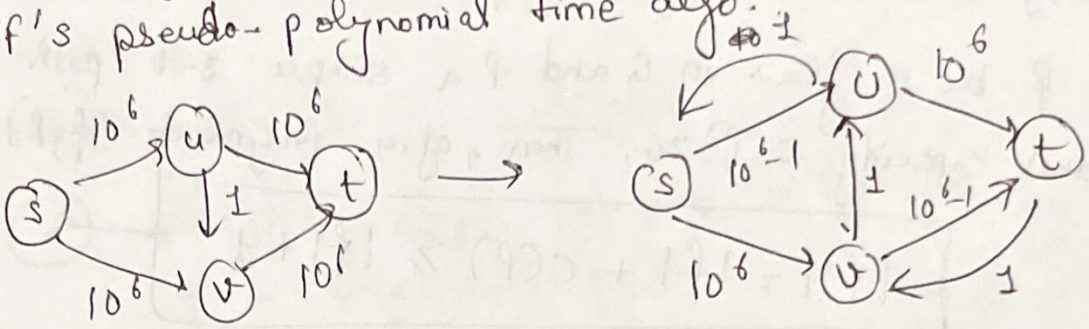
∴, one iteration requires $O(m)$ time.

∴, Running time of ford-fulkerson is $O(mnU)$. - Ans

Pseudo-polynomial algo ⇒

(*) An algo is pseudo-poly. algo if it is polynomial in the size of the I/P when the numeric part of the I/P is encoded in unary.

→ FF's pseudo-polynomial time algo.



U → max capacity

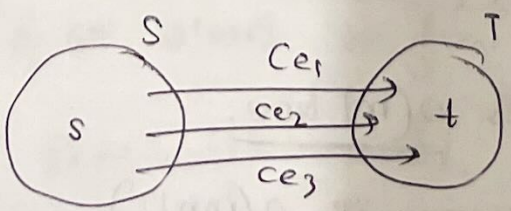
Here the degree of the capacity affects the running time in an exponential way.

U	Size of U (# digits to describe)	# iterations
1	0	2
10	1	2·10
100	2	$2 \cdot 10^2$
1000	3	$2 \cdot 10^3$
10000	4	$2 \cdot 10^4$

(b) A natural upper bound for max value of a flow?

→ An s-t cut (S, T) in G is a bipartition of the vertices into two sets S & T, s.t. $s \in S$ and $t \in T$.

⇒



The capacity $c(S,T)$ of s-t cut (S,T) is

$$\sum_{e \text{ out of } S} c_e$$

Then intuitively :-

$$\max_{f} |f| \leq \min_{(S,T) \text{ cut in } G} c(S,T)$$

(*) Roadmap to prove optimality of FF:-

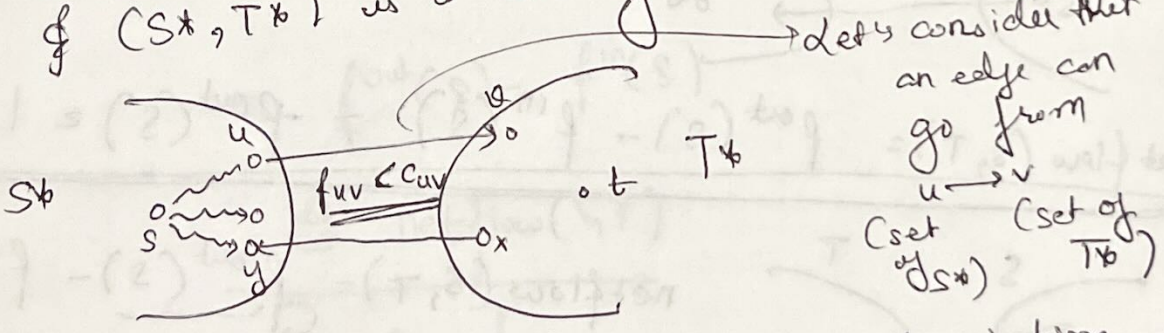
$|f| = f^{out}(S)$. We will prove the following:-

1. for any flow f, the value of the flow $|f|$ cannot exceed the capacity of any cut in G.
2. Let f^* be the flow upon termination of the FF algo. We will exhibit a specific cut (S^*, T^*) s.t the value of f^* equals the capacity of (S^*, T^*).

$$|f^*| = c(S^*, T^*)$$

So, we conclude that f^* is max flow.

& (S^*, T^*) is a cut of minimum capacity.



Let G_f^* be the residual graph upon termination of FF and f^* the flow upon termination of FF.

S^* = set of nodes reachable from S in G_f^*.

(S^*, T^*) is an s-t cut in G_f^* → both has some set of vertices!

$\Rightarrow \nexists (u,v)$ in G_{f^*} with $u \in S^*$, $v \in T^*$.

→ If $(u,v) \in G$ with $u \in S^*$, $v \in T^*$, then $f(u,v) = c_{uv}$

→ If $(x,y) \in G$, with $x \in T^*$, $y \in S^*$.

Date:- $\text{netflow}(S, T) = f^{out}(S) - f^{in}(S)$

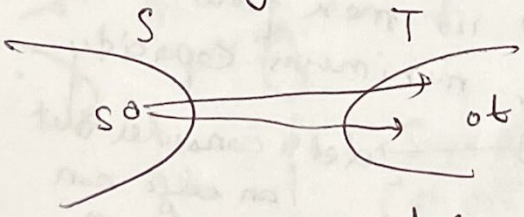
$$f^{out}(S) = \sum_{e \text{ out of } S} f(e)$$

$$f^{in}(S) = \sum_{e \text{ into } S} f(e)$$

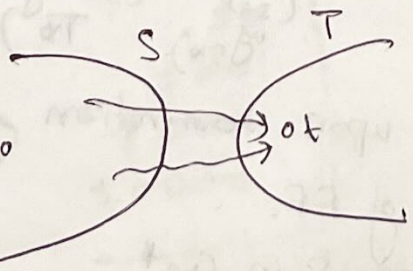
$$\text{netflow}(S^*, T^*) = \sum_{e \text{ out of } S^*} f(e) - \sum_{e \text{ into } S^*} f(e)$$

$$\Rightarrow \sum_{e \text{ out of } S^*} c_e \Rightarrow c(S^*, T^*)$$

flow is netflow (S, T) related to $|f|$,



$$\text{netflow}(S, T) = f^{out}(S) - f^{in}(S) = f^{out}(S) = |f|$$



$$\text{netflow}(S, T) = f^{out}(S) - f^{in}(S)$$
$$= f^{in}(t)$$
$$= |f|$$

LEMMA

netflow $(S, T) = |f|$

$\forall$ s-t cut (S, T), $\forall f$

COROLLARY

$$|f| = \text{netflow}(S, T) = f^{out}(S) - f^{in}(S) = \sum_{e \text{ out of } S} c_e = c(S, T)$$

Proof of Lemma:-

$\forall$ s-t cut (S, T) $\quad \forall f,$

$|f| = \text{netflow}(S, T)$ → To prove

Proof → $|f| = f^{out}(S) - f^{in}(S)$

$$= \sum_{v \in S} f^{out}(v) - f^{in}(v)$$

$$= \sum_{v \in S} \left(\sum_{(v,u) \in E} f(v,u) - \sum_{(u,v) \in E} f(u,v) \right)$$

$$= \sum_{\substack{(v,u) \in E \\ v \in S, u \in T}} f(v,u) - \sum_{\substack{(u,v) \in E \\ u \in T, v \in S}} f(u,v)$$

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

$$= f^{out}(S) - f^{in}(S)$$

$$\triangleq \text{netflow}(S, T)$$

Hence, proved

(*) The max-flow min-cut theorem:-

THEOREM:- If 'f' is an s-t flow such that there is no s-t path in G_f, then there is an s-t cut (S^*, T^*) in G such that $|f| = c(S^*, T^*)$,

$\therefore$ f is a max-flow and (S^*, T^*) is a cut of min capacity.

Max cut - min flow theorem :-

In every flow network, the max^m value of an s-t flow equals the min^m capacity of an s-t cut.

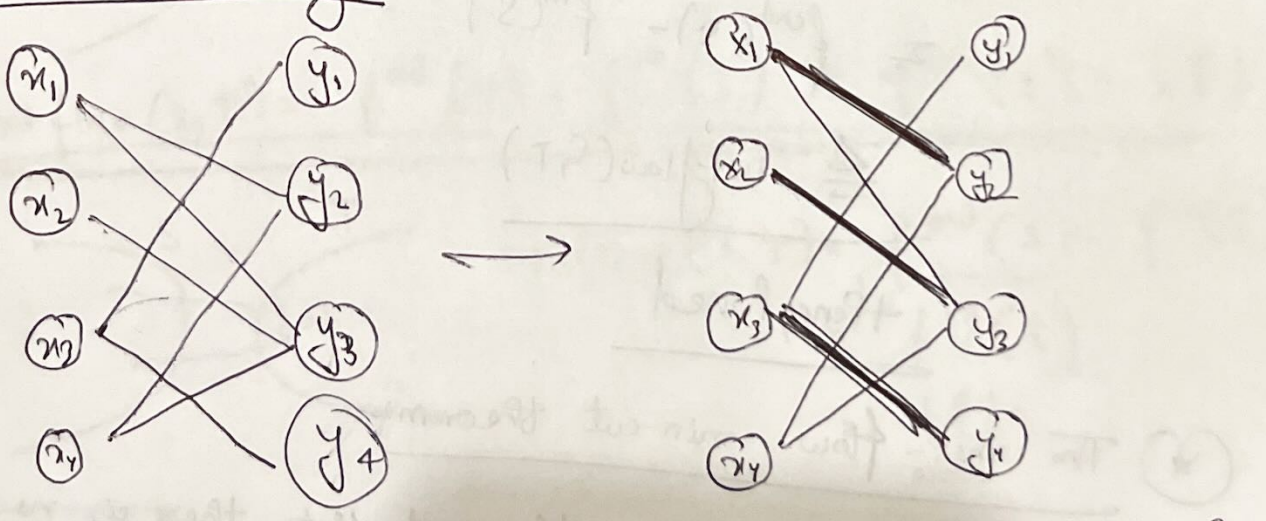
→ we can efficiently find a min path because we already have the residual graph G_f and can form S* using BFS/DFS.

(*) Integrality Theorem :-

→ In FF, the flow values & residual capacities in G_f are I.

If all capacities in a flow N/W are ints, then there is a max^m flow for which every flow value f(e) is an integer.

(*) Bipartite Matching :-



This is a matching of size 3
$M = \{(x_1, y_2), (x_2, y_3), (x_3, y_4)\}$

⇒ A matching M is a subset of edges where every vertex in X ∪ Y appears at most once.

we can get a matching of size ≤ 4 here, nothing more since these matchings are one-to-one.

Perfect Matching:- Every vertex in $X \cup Y$ appears exactly once.

Not always possible, eg:- $|X| = |Y|$.

Maximum Matching desirable in applications.

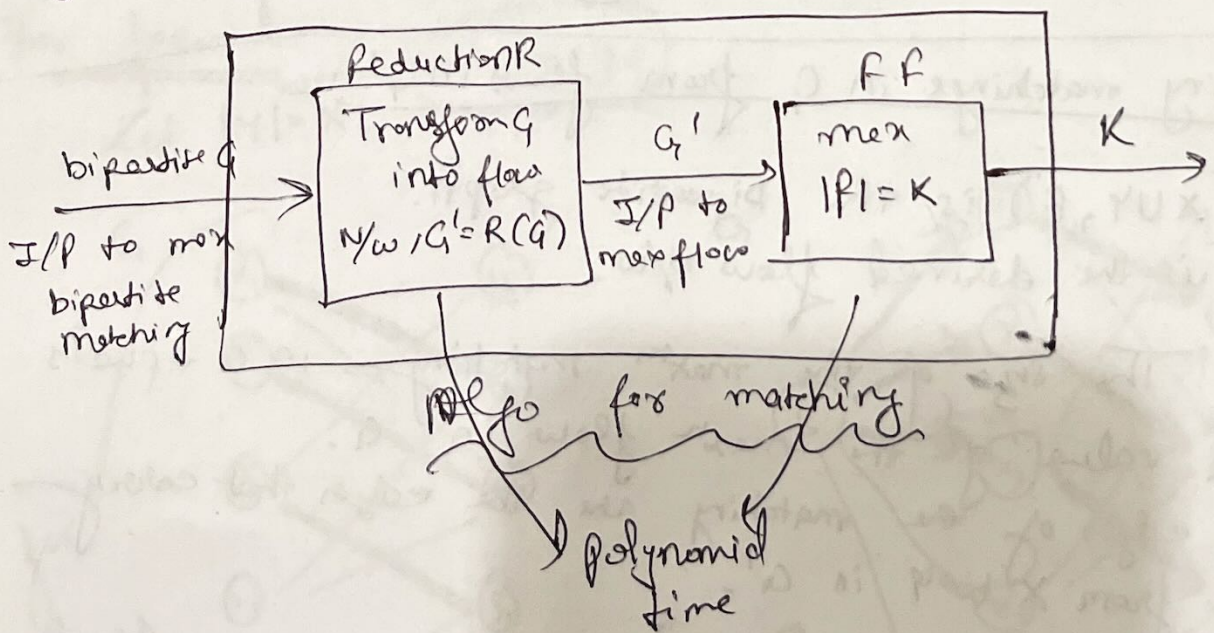
→ If we had an algo to find max^m matching then we could also find a perfect matching, if one exists.

(*) **finding max matchings in bipartite graphs:-**

IDEA:- Use FF algo to find max (or perfect) matchings.

STRATEGY:- reformulate the problem as a max-flow problem which we know how to solve (reduction).

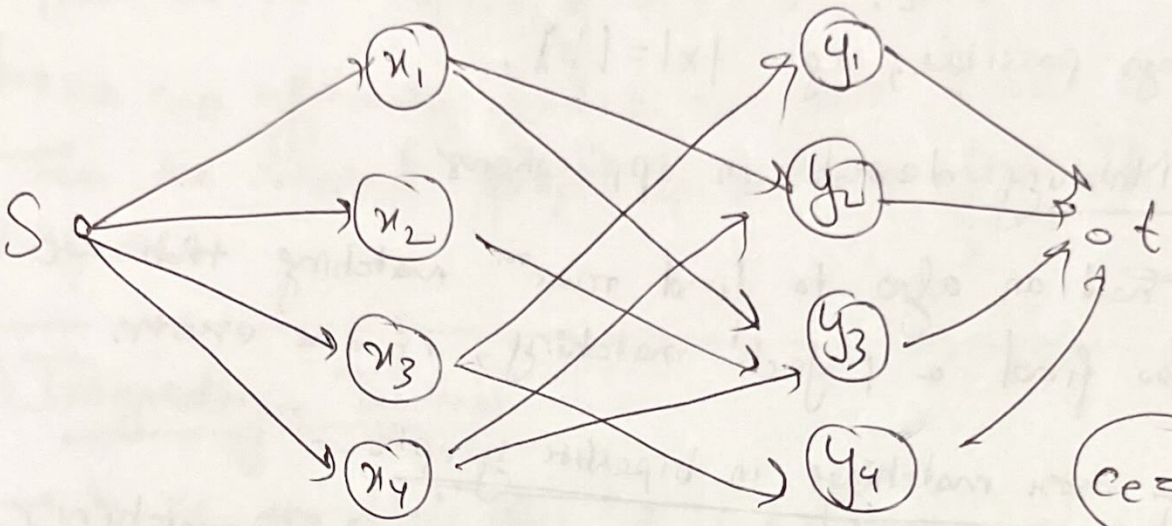
To this end, we need to transform our input bipartite graph into a flow N/w.



→ We need to prove the following for correctness:-

(1.) Reduction R must be efficient (polynomial in the size of G)

(2.) G and G' should be equivalent, in the sense that G has a max matching of size k iff the max flow in G' has value k.



$c_e = 1$ for all e.

bcoz it is the smallest value (int > 0) to use to keep it easy. Also, running FF depends on U, thus, it will be the most efficient.

(V) Computing matchings in G from flows in G' :-

→ $G = (X \cup Y, E)$ is the bipartite graph.
→ G' is the derived flow N/w.

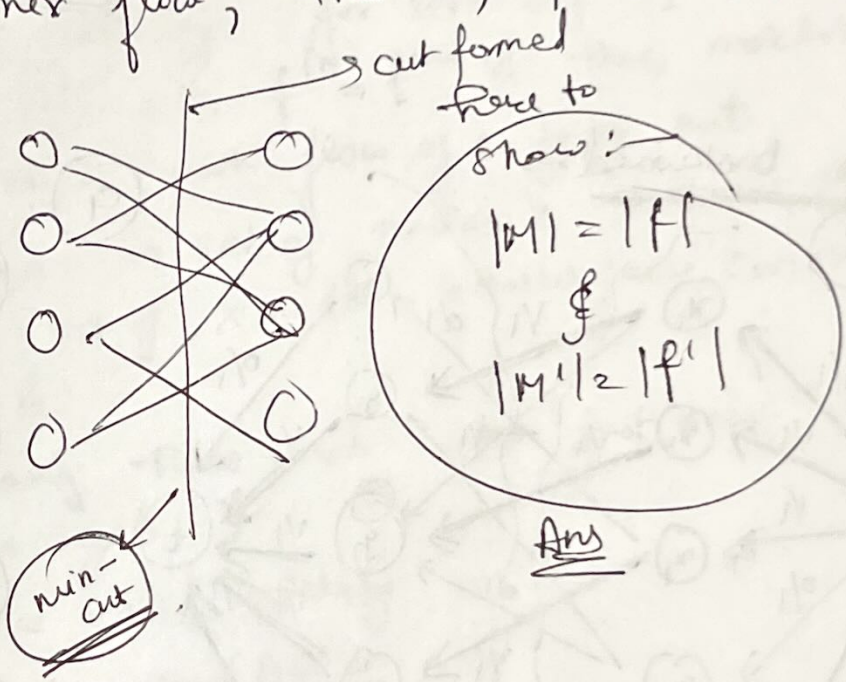
CLAIM :- The size of the max^m matching in G equals the value of the max^n flow in G.
The edges of the matching are the edges that carry flow from X to Y in G'.

(Equivalence of Instances)

Proof ⇒

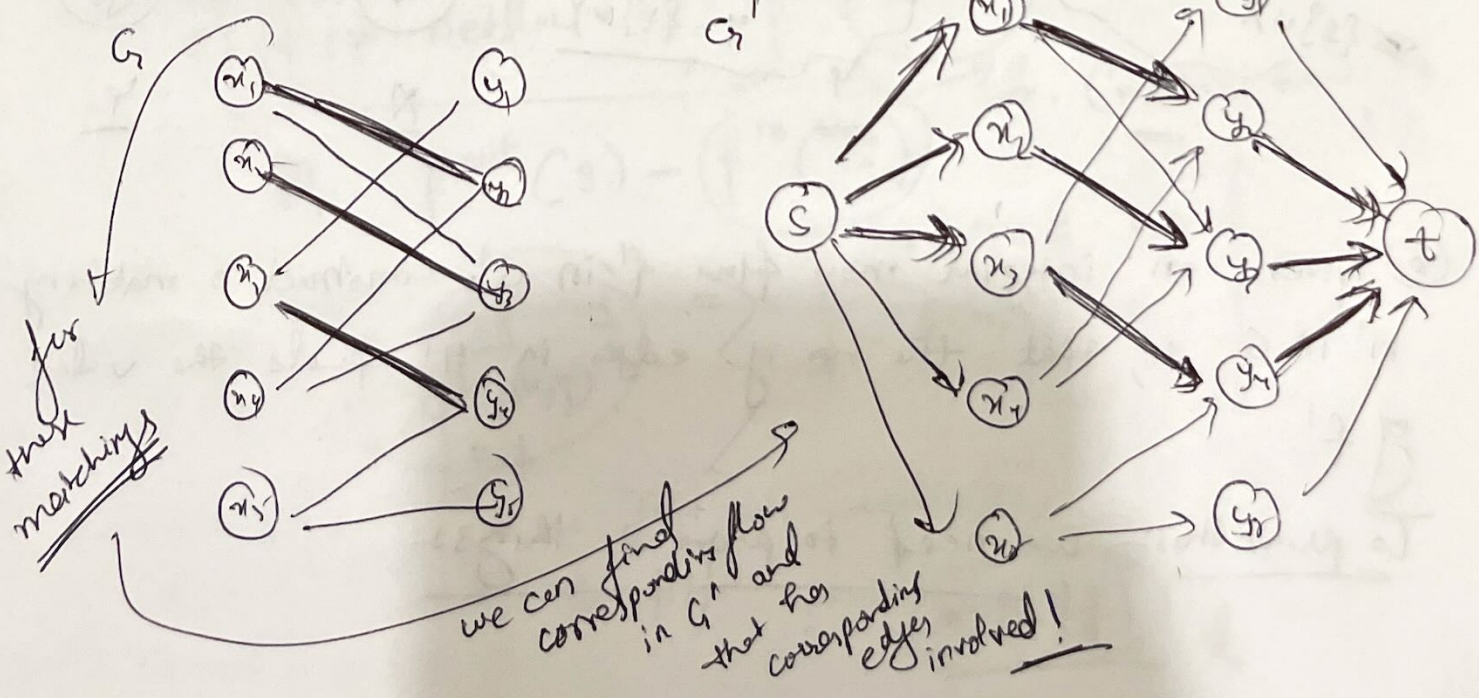
(1) Forward direction ⇒ For any matching M in G, we can construct a flow f in G' with value equal to the size of M, that is $|f| = |M|$.

(2) Reverse direction ⇒ Given a max flow f' in G', we can construct a matching M' in G, with size equal to the value of the max flow, that is, $|M'| = |f'|$.



(1) For forward → matching size

Let $|M| = k$,



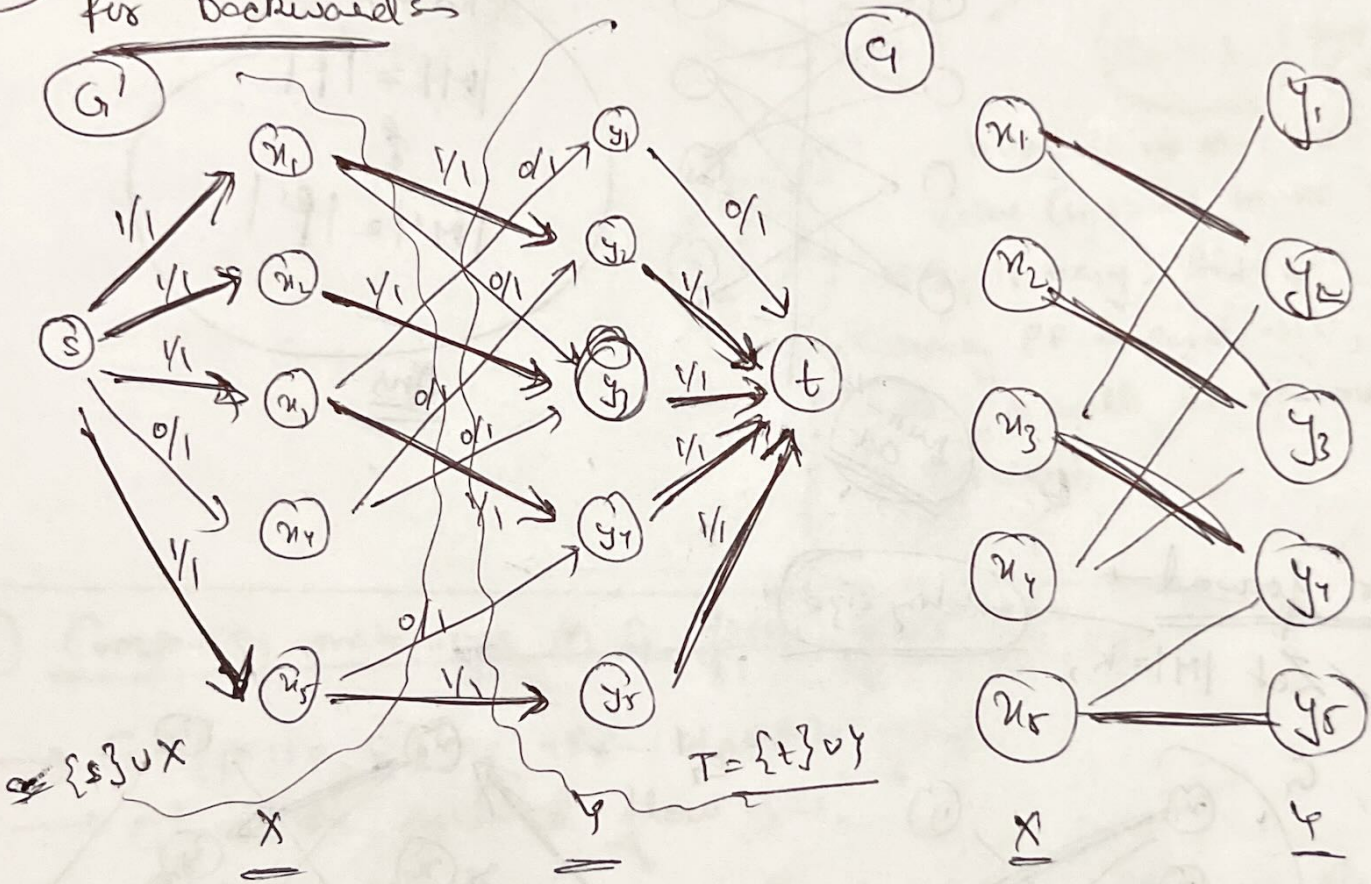
Thus, given a matching M (darkened edges in G), we can construct an integral flow f in G' s.t. the value of f equals the no. of edges in M.

$|f| = |M|$

Hence, forward dir^n equivalence of instance is proved ✓

Hence Proved

(2) For backwards:



(⁂) Given an integral max flow f' in G', construct a matching M' in G so, that the no. of edges in M' equals the value of f'.

To prove this:- we need to prove 2 things:-

First we need to prove that M would be a valid matching that we are getting thru. the flow network.

for that, M can be formulated as follows (to get a matching):-

$$M = \{ (x,y) \in E \quad s.t \quad f(x,y) = 1 \}$$

for this,

the matchings that we get, due to flow conservation, we can't have 2 matchings for lets say x_1 as a flow of 1 comes into x_1, so, $f^{in} = f^{out}$ & thus, matching found cannot have 2 edges with flow of 1 going out.

Similarly, any node in set of nodes y cannot have 2 matchings from set of x nodes otherwise flow conservation will be invalid.

Thus, we can only have each vertex just once in the matchings that we get.

∴ It is a valid matching.

(b) Now, we need to prove that :-

$$|M| = |f|$$

$$\rightarrow |f| = \text{netflow}(s,T) \rightarrow \text{Zero}$$

$$\Rightarrow f^{out}(s) - f^{in}(s) \quad (\text{nothing coming in } s).$$

$$\Rightarrow \sum_{(x,y) : f(x,y) = 1} 1 \Rightarrow \quad (x,y) \in E \text{ s.t } f(x,y) = 1$$

This was actually (M)

$$\Rightarrow |M| \quad \therefore |M| = |f| \quad \#$$

Hence, we have proved the equivalence!

This is a poly-time algo.

Reduction → $O(n+m)$

FF → $O(nm)$

$O(nm)$

Improved → $O(m\sqrt{n})$

⇒ If we convert this into a decision problem!

BPM :- Does a bipartite graph have a perfect matching?

I/P → a bipartite graph $G = (X \cup Y, E)$.

O/P → yes, iff G has a matching of size $|X| = |Y|$.

How to do this?

Terminology →

→ An instance of BPM is a specific I/P graph G.

→ we may think of an I/P instance G as a binary string x encoding the graph G, with length $|x|$ bits.

→ we assume reasonable encodings: e.g. a binary number b requires a logarithmic no. of bits to be encoded.

→ reasonable encodings are related polynomially.

→ An algo. that solves BPM admits:-

→ I/P :- a binary string x encoding a bipartite graph

Hashing, Bloom filters

→ A data structure maintaining a dynamic subset S of a huge universe U.

$$|S| << |U|$$

support:
- → efficient insertion
- → efficient deletion
- → efficient search

holds all these → { dictionary }

Opⁿˢ supported

(1) Create :- Initialize a dict with $S = \phi$.

(2) Insert :- add x to S, if $x \notin S$

→ Additional info about x might be stored in the dict as part of a record for x.

(3) Delete(x) :- delete x from S, if $x \in S$.

(4) Lookup(x) :- determine if $x \in S$.

(2), (3), (4) require a lookup!

for IP addresses →

$$|U| = 2^{32} \Rightarrow |S| = 250$$

↳ sample size of IP addresses.

(*) If we have an array for storing the bits as booleans, then time complexity is very good, but unrealistic to store it. (space considerations are huge!).

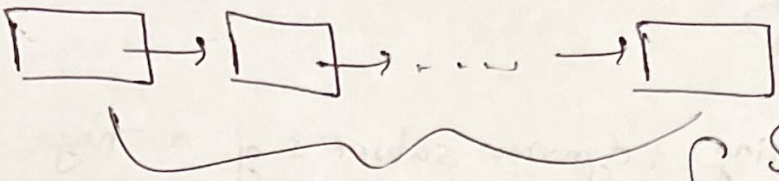
lookup → $O(1)$ ← insert

↑

Delete.

Space → $O(|U|)$ (X)

(*) If Linked List,



|S| nodes

Space → O(|S|)
Lookup → O(|S|)
Insert
Delete

So, space complexity gets better, but time goes higher!

Thus, we are trying to both optimize time & space.

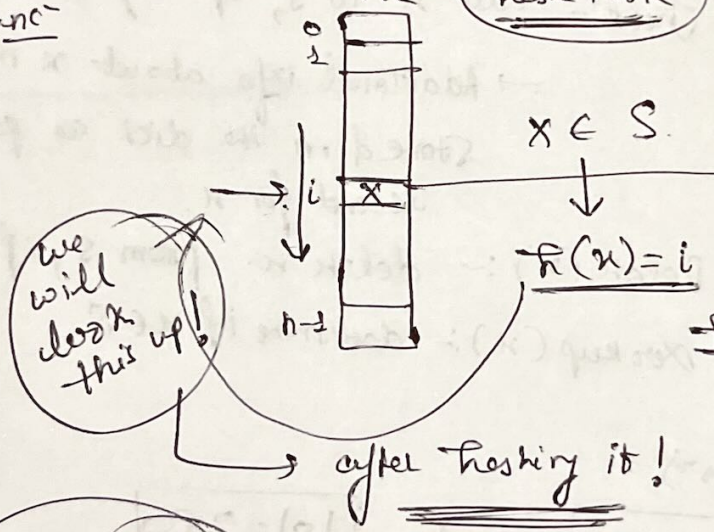
$|S| << |U|$

$|S| = n$

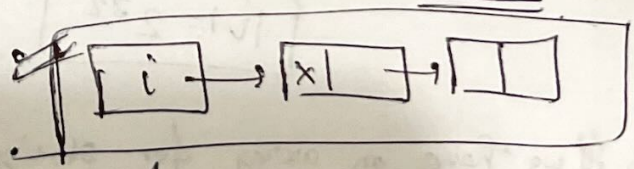
$h: U \to \{0, 1, \ldots, n-1\}$

hash func

H ← hash table



Hashing as chaining



Thus, what would be the time required:-

⇒ Lookup(x):-

(1) Compute $h(x)$ → O(1) time (might be diff!)

(2) Traverse the linked list at $h(x)$ to find x.

→ Time :- O(# elements in linked list at $h(x)$) ← it should spread out elements as much as possible

insert & delete are constrained by ZooKup!

⇒ Simple hash fns might not work!

(I) (2 simple hash func^{n} that hash an IP address x from $\{0, \ldots, 2^{32}-1\}$ to $\{0, \ldots, 255\}$.

→ assign the last 8 bits of x as $h(x)$.

→ assign the 1st 8 bits of x as $h(x)$.

⇒ Nothing is inherently wrong ⇒ the problem is that our 2 IP address might not be drawn uniformly at random from among 2^{32} possibilities.

(II) No single hash func^{n} can work well on all data sets.

FACT:- for any fixed (deterministic) $h: U \to \{0, 1, \ldots, n-1\}$ where $|U| \geq n^2$, there exists some set S of n elements that all map to some pos^{n}.

eg → h dest. into n names

$|U|$ → $\frac{|U|}{n}$ → map to some → $|S|$.

(lots of collisions possible!)

⇒ There are n^2 balls into n bins in any using:

→ what is lower bound on the max load of any bin?

Proof ⇒ (n collisions always!)

1 2 ... n ⇒ $\geq n$

to optimize max across all the bins! ⇐ Can atleast give n balls to all the bins.

for any bin, to optimize, if

→ we are going to introduce randomness here to decrease # collisions!

Completely random hash fn:

$\forall x \in U, \forall 0 \le j \le n-1$ independently.

$$\boxed{Pr[h(x) = j] = 1/n}$$

for $x, y \in U$,

$$Pr[h(x) = h(y)] \Rightarrow Pr[h(x) = i \wedge h(y) = i]$$

$$\sum_{i \ge 0}^{n-1} Pr[h(x) = i \wedge h(y) = i]$$

$$\Rightarrow \boxed{\sum_{i \ge 0}^{n-1} Pr[h(x) = i] \cdot Pr[h(y) = i].}$$

$$\Rightarrow \sum_{i \ge 0}^{n-1} \frac{1}{n} \cdot \frac{1}{n} \Rightarrow \sum_{i \ge 0}^{n-1} \frac{1}{n^2} = \boxed{1/n}$$

(when collision would happen)

probability for that?

⇒ Suppose $x \in S$,

How many other elements in S collide with x in expectation?

Let X = # other elements from S that collide with x.

$E[X] = ?$

$X_i = \begin{cases} 1, & \text{if element } i \in S, i \neq x, \text{ collides with } x \\ 0, & \text{otherwise.} \end{cases}$

Indicator func^n

$$\boxed{X = \sum_{i \ge 1}^{n-1} X_i}$$

$$E[X] = E\left[\sum_{i=1}^{n-1} X_i\right] = \sum_{i=1}^{n-1} E[X_i] = \sum_{i=1}^{n-1} Pr[X_i = 1]$$

$$\Rightarrow \frac{n-1}{n} < 1$$

only size of n elements.

$\therefore$ size of linked list is const. (w.r.t the collisions)

($\because$ it works in const time) → time is const. in expectation!

~~Linear space etc.~~ is.

(*) What completes the problem?

↳ $\because$ hash assigns a random value! (it's not a func^n as we don't know where to store it).

{ Randomness is valuable but is not feasible! }

→ So, we will randomize the choice of hash function from a suitable class of func^n into $[0, n-1]$.

[h must have a compact representation].

⇒ Universal Hash function ⇒ (does it span the whole universe?)

Idea:- choose h at random from a carefully selected class of functions H with the following properties:-

(1) h behaves almost like a completely random hash func^n.
→ for $x, y \in U$. The probability that a randomly chosen $h \in H$ satisfies $h(x) = h(y)$ is at most $1/n$.

→ Can select a random h(a) efficiently.

→ Given h, can compute $h(x)$ efficiently.

⇒ Such hash func^n are called universal; their design relies on number theoretical facts.

(8)

$U \Rightarrow \{a, b, c, d, e, f\}$

$n = 2$ (~~range~~ range of hash fn).

$k = 2$ (no. of hash function)

$H = \{ h_1, h_2 \}$ → universal class of ~~hash fns~~ ? Is this?

$h_1 : U \to \{0, 1\}$

$h_2 : U \to \{0, 1\}$

	a	b	c	d	e	f
h_1 :	0	0	0	1	1	1
h_2 :	0	1	0	1	0	1

⇒ Prob. of $h(x) = h(y)$ ⇒ $Pr[\;] \Rightarrow 1/n$

$Pr_h[h(x) = h(y)] \leq \frac{1}{n}$

fix a, b,

$Pr_h[h(a) = h(b)] \Rightarrow$

$Pr[h_1(a) = h_1(b) \mid h_1] \cdot Pr[h_1] + Pr[h_2(a) = h_2(b) \mid h_2] \cdot Pr[h_2]$

$\Rightarrow 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{2} \Rightarrow 1/2$

$Pr_h[h(a) = h(b)] \Rightarrow 1/2 \leq 1/n$ when $n = 2$,

$1/2 \leq 1/2$ → this holds!

⇒ This should hold for all pair of values!

ix a, c ⇒

$$Pr_{h}[h(a) = h(c)] \Rightarrow$$

$$Pr[h_1(a) = h_1(c)] \cdot Pr[h_1]$$

$$+ Pr[h_2(a) = h_2(c)] \cdot Pr[h_2]$$

$$\Rightarrow 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = 1 \leq 1/n \quad (X)$$

(doesn't hold)

⇒ This is not universal class of hash func^n.

Hence, Proved Ans.

(Imp) We will have to assume a completely random hash fn exists.

(bcoz this provides us a good rough idea of how hashing schemes perform in practice)

(in reality, there are no completely random hash fns).

Remark → $h(x)$ is fixed for every x: it just takes one of the n possible values with equal probability.

$(x, h(x))$ ↳ stored as pair!

⇒ Hashing modeled as a balls and bin problem ⇒

Q: How many elements can we insert in the hash table before it is more likely than not that there is a collision?

Ans. At least 1 entry in the table that has more than 1 ball (in any of the bins).

$\Rightarrow \boxed{\Omega(\sqrt{n})}$

(just an occupancy problem).

hashing

1. balls correspond to elements from U.
2. bins are slots in the hash table.
3. each ball falls into one of the n bins independently & with prob. $1/n$. (uniform distribution).

→ Analyzing time/space of hash table →

(Q) What is the expected time for lookup(x)?

$\Rightarrow \underbrace{O(1)}_{\text{calculate hash func}^n} + \underbrace{O(1)}_{\text{scan the linked list at that position.}} \Rightarrow \boxed{O(1)}$

(because we will take completely random hash fn).

(Q) What is the expected wasted space in the hashed table?

(how many entries are empty) $\Rightarrow \boxed{n/3}$ $(> n/3)$

(expected no. of empty bins).

(hashing with chaining).

(Q) worst case time for Lookup(x)?

$\Rightarrow \boxed{O(n)}$ (all items might be mapped at the same loc^n).

worst-case with highest prob. case, (w.h.p) $\geq 1 - 1/n$ (independent & uniformly at random!)

$\Rightarrow O\left(\frac{\ln n}{\ln(\ln n)}\right) \Rightarrow O(\ln n)$ Ans.

{ capacity of highest loaded bin!

Apps of Hashing to save space ⇒

⇒ **Password checker:-**

⇒ maintain a dictionary for a set S of 2^{16} bad passwords so, that when a user tries to set one, we check as quickly if it belongs to S and reject it.

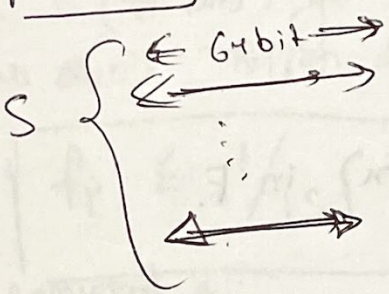
→ **Assume:-** p/w consist of 8 ASCII chars. (rep. 8 bytes (64 bits) to represent) password.

Q. how to store the dictionary?

Store them explicitly!

⇒ S of 2^{16} bad passwords

⇒ $x \in S$



↓

how to efficiently search?

1. Sort S.
2. Binary search for X in S.
3. Time:- $O(\log |S|)$
4. Space:- $|S|.64$ bits.

⇒ Now, we want to use less space!

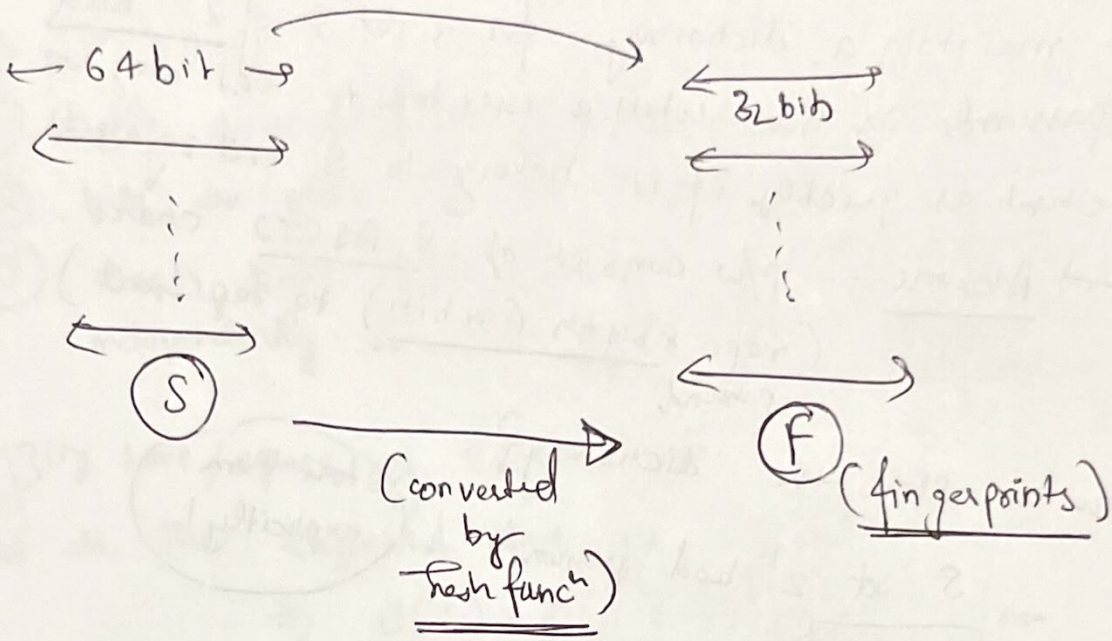
(we would potentially allow for some errors).

→ we might reject some good passwords, but never accept a bad password!

Okay to have FPs, certain about (FNs).

IDEA ⇒ Map passwords to fingerprints.

$h : \{0, \ldots, 2^{64}-1\} \to \{0, \ldots, 2^{32}-1\}$.



1. Sort F
2. Compute $h(x)$
3. Binary search for $h(x)$ in F.
4. Lets say $h(x) \in F$

How ~~to~~ answer is x in S?

⇒ $x \in S$ (we will find it in F and reject ~~not~~ accordingly)

$h(x) \notin F$

Answer ⇒ $x \notin S$

(correct), so we do not reject!

→ Time to answer the query?

Same as before!

$O(\log_2 |S|)$

→ Space :- $O(|S| \cdot 32)$ Ans (32 bits fingerprint)

Correctness:-

If $x \notin S$ as we do not find that,

(*) FNs are not possible! (as we are still storing all bits of x).

(*) FPs might happen. ⇒ (as smaller range of bit strings stored, collisions might be more).

(as info. is less).

$\therefore P(FPs) \neq 0$

Imp

Thus, we have reduced space without FNs and $P(FPs) \neq 0$.

(we want P(FPs) as less as possible as well, so, we set no. of bit in f accordingly).

Ans

Q. what is b (# bits) for each fingerprint s.t. FNs don't happen & prob. of FPs ≠ 0 but less.

$$fp \leq 1/|S|$$

Fix x → good password x.

$fp = Pr[h(x)$ collides with $h(y)$ for some "bad" $y]$

→ fix a "bad" y,

$Pr[h(x) = h(y)] = 1/n = 1/2^b$

Ans

(for completely random hash function)

$Pr[h(x) \neq h(y)] \Rightarrow 1 - \frac{1}{2^b}$

(differs from all those fingerprints.

$Pr[\forall \text{ bad } y\ h(x) \neq h(y)] \geq \left(1 - \frac{1}{2^b}\right)^{|S|}$

$fp \Rightarrow Pr[\exists \text{ bad } y, \text{ s.t. } h(x) = h(y)]$

$$\Rightarrow \boxed{1 - \left(1 - \frac{1}{2^b}\right)^{|S|}}$$

for,

$$\boxed{1 - \left(1 - \frac{1}{2^b}\right)^{|S|} \leq \frac{1}{|S|}}$$

↳ solving for b gives,

$$\boxed{b = 2 \log_2 |S|}$$ Ans

$$\boxed{\text{Space} \Rightarrow O(|S| \log |S|) \text{ bits}}$$ Ans

⇒ Improved fp ⇒

$$\boxed{b = \alpha \log_2 m}$$

Derivation

original fp ⇒ ⇒

$$\Rightarrow 1 - \left(1 - \frac{1}{2^b}\right)^m$$

$$\Rightarrow 1 - \left(1 - \frac{1}{m^2}\right)^m \leq 1 - \left(1 - \frac{1}{m}\right) = 1/m$$

Thus, if our dict has $|S| = m = 2^{16}$ bad passwords, using a hash func^n that maps each of the m passw to 32 bits yields a false pos. prob of about $1/2^{16}$

Ans

∴ we would use bloom filters (for checking bad URL)

I/P:- a large set S, and queries of the form "Is $x \in S$?"
we want a data structure that answers the queries:-

→ fast (faster than searching in S) O(1)

→ is small (smaller than the explicit repr^n provided by hash table). O(1)

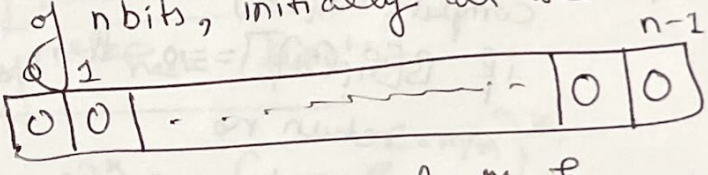
{ → FPS:- answer yes when $x \notin S$.
→ FNS:- answer no when $x \in S$. } → const fp probability.

O/P:- small prob. of FPs, no FNS. (we need this!)

→ Bloom filters ⇒ (used by Google for finding bad URLs).

Consists of:-

(1) an array B of n bits, initially all set to 0.



(2) K independent random hash func^{ns} $h_1, \ldots, h_k$ with range $\{0, 1, \ldots, n-1\}$

A basic bloom filter supports:-

→ Insert (x)

→ Lookup (x)

⇒ Pseudocode:-

```
SetupBloomfilter(S, h1, ..., hk):
    Initialize arr B of size n to all zeros.
    for i = 1 to m do:-
        Insert(xi)
    end for
```

Insert(x):

$|S| = m$

```
for i=1 to k:
    Compute h_i(x)
    set B[h_i(x)]=1
end for
```

Imp → An entry of B may be set multiple times; only the first change has an effect.

$h_i : U \to \{0, 1, \ldots n\}$

Lookup y

To check membership of an element x in S do:-

```
Lookup(x):-
    for i=1 to k do:
        Compute h_i(x)
        if B[h_i(x)]==0 then
            return no
    return yes
```

⇒ If $B[h_i(x)] \neq 1$, for some i, then $x \notin S$.

⇒ Otherwise, answer "$x \in S$".

↳ might be FP-s since we could get the same values in bloom filter mapped due to other elements which were actually not part of the same element x_i (could be comb^n of multiple x_i's).

⇒ false negative not possible since we never set 1 back to 0, so, we won't lose any info. & we have all the info. for the complete set S. (which covers all the info. for finding an x) → thus, a kind of a filter!

Probability of false positive:-

→ After all elements from S have been hashed into the Bloom filter, the probability that a specific bit is still 0 is:-

$$\left(1 - \frac{1}{n}\right)^{km} \approx e^{-km/n} = p$$

→ To simplify the analysis, assume that the prob. that a specific bit is still 0 is exactly p.

→ The prob. of a false positive is the probability that all k hashes evaluate to 1:-

$$f = (1-p)^k$$

Optimal number of hash func^{n}:-

$$f = (1-p)^k = \left(1 - e^{-km/n}\right)^k.$$

more hash fns ↓ ⇒ Trade-off b/w k & p:- using more hash functions:-
↑ k, fp less → gives us more chances to find a 0 when $x \notin S$,
but we need to reduce sparsity of the array B. (bloom filter) → but reduces the no of 0s in the array!

→ Compute optimal no. k^* of hash functions by minimizing f as a func^{n} of k:-

$n \to |B|$
$m \to |S|$

$$k^* = (n/m) \cdot \ln 2$$

Then the false positive probability is given by:

$$f = \left(\tfrac{1}{2}\right)^{k^*} \approx (0.6185)^{n/m}$$

Ans.

Big savings in space:-

n/m bits

for ex:- set $n = 8m$. Then $k^* = 6$ and $f \approx 0.02$

(thus, constant space gives fp quite less)

Δ no FNs (✓)

⇒ small constant false +ve prob. by using only 8 bits (1 byte) / element of S; independently of the size of S!

Ans. (space / input element → 8 bits (→ thus, const. space)) Ans.

$E[X] = n H_n = n \ln n + n\Theta(1)$

$= \boxed{\Theta(n \ln n)}$

Ans. (super-linear no.)

{ # balls needs to be thrown! }

Randomized algo. ⇒ worst-case (deterministic I/P)

⇒ On the same input, the algo. produces the same o/p but diff. executions may req diff. running times.

⇒ The latter depends on the random choices of the algorithm.

⇒ Random samples are assumed independent of each other.

→ worst-case I/P.

→ expec. run. time analysis of the running time of the rand. algo. on a worst case input.

(1) Las Vegas ⇒

— deterministically correct → always returns correct

— probabilistically fast

(2) Monte Carlo ⇒

— deterministically fast

→ probabilistically correct → some executions may fail to return the correct answer.

Ans.

⇒ Rand. algos may be:-

(1) simpler
(2) req less memory of the past
(3) useful for symmetry-breaking
(4) generally equally efficient as deterministic algos.

that's why used

Deterministic algos are a special case of randomized algos.

Randomized Quicksort:-

→ Can we use randomization so that quicksort works with an "avg" I/P even when it receives a worst-case I/P?

(1) Explicitly permute the I/P (there's no def. way to do that, tough to analyse!)

(2) Use random sampling to choose pivot: instead of using A[right] as pivot, select pivot randomly.

↙ we won't often pick the largest/smallest item as pivot (prob very less). Thus, most often the partitioning will be "balanced".

⇒ Random-Partition

b = random(left, right) → this is a random variable!
swap(A[b], A[right])

then we call "Partition"

random(i, j) → which is an O(1) op.

(n) → expected running time of Randomized Quicksort. (based on Quicksort & Partition).

Partition → $O(n)$

↳ can be called n times in worst-case! (# calls to partition) → n

(*) Total time spent in partition calls outside the for loop ⇒ $O(n)$

(*) Total time spent in partition calls outside the for loop?

(rv) → X_i comparisons in the i-th partition call

$\Rightarrow O(X_i)$

$X \Rightarrow$ total # comparisons in all partition calls.

$\Rightarrow O(X)$ time

Running time of Rand. QS ⇒ $\Rightarrow \boxed{O(n) + O(X)}$

(rv) → depends on random choices made in the algorithm!

$T(n) \Rightarrow$ exp. runtime of Rand-QS.

$$T(n) = E\left[O(n) + O(X)\right]$$

$$= O(n) + O(E[X])$$

(*) Fix X_i & X_j ⇒ each of the nos can be compared at most once → (either once/never)

∴ compared ≤ 1 throughout the algo.

Pairs of X_i & X_j ⇒ $\leq \binom{n}{2} \cdot 1 \Rightarrow O(n^2)$

↳ # comparisons

⇒ for better behavior, we need to know expected # comparisons before ⇒

— Assume all I/P items are distinct.

— Relabel the I/P as $\{z_1 < z_2 < \dots < z_n\}$

→ # comparisons (total)

$E[X] \Rightarrow$

$X_{ij} \Rightarrow \begin{cases} 1, & \text{if } z_i \text{ and } z_j \text{ are never compared} \\ 0, & \text{otherwise.} \end{cases}$

$$X = \sum_{1 \le i < j \le n} X_{ij}$$

→ dist. as many rv's.

$$E[X] \Rightarrow \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

$\Pr[X_{ij} = 1] = ?$

fixes pivot

$z_1 < z_2 < \dots [z_i < \dots < z_j] < \dots z_n$

z_{ij}

$z_i \& z_j$ won't be compared when z_{ij} is being considered. Otherwise, we don't know whether they would it be!

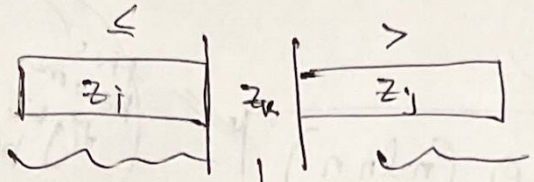
So, how can we know when $z_i \& z_j$ will be compared

↳ This will happen!

→ Suppose an item from z_{ij} is chosen as pivot.

→ z_i = pivot
→ z_j = pivot
} → then, they would be compared. ✓✓

⇒ $z_i < z_k < z_j$, z_k = pivot } → No, when pivot is being considered.
(so, never compared, here as well!).



(all comparisons w.r.t. pivot).

⇒

∃ comparisons between z_i & z_j,
→ the first item picked as pivot, item from z_{ij} is z_i / z_j.

So, $P_r[X_{ij} = 1]$ ⇒ $P_r[z_i / z_j$ is the 1st item picked as pivot from $z_{ij}]$.

NOTE: $P_r[E_1 \cup E_2] = P_r[E_1] + P_r[E_2]$ if $E_1 \cap E_2 = \phi$

$$P_r[X_{ij} = 1] = \frac{1}{|z_{ij}|} + \frac{1}{|z_{ij}|} = \frac{2}{|z_{ij}|} = \frac{2}{j-i+1}$$

$$= P_r[X_{ij} = 1]$$

$$E[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$\Rightarrow 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1/j-i+1.$$

Let $k = j-i+1$,

for $j = i+1 \Rightarrow k = i+1-i+1 = 2$.

for $j = n \Rightarrow k = n-i+1$.

$$\therefore \Rightarrow 2 \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} 1/k$$

$$\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} 1/k \quad \rightarrow H_n$$

$$\leq 2(n H_n) \Rightarrow \boxed{\Theta(n \ln n)}$$

Ans.

this becomes tight-bound.

$X_i = \begin{cases} 1, & \text{if } i\text{th coin flip is H} \\ 0, & \text{otherwise} \end{cases}$

$X = \sum_{i=1}^{n} X_i$ — how many coins come up as head.

$\Rightarrow E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} Pr[X_i = 1]$

Linearity of expectation.

$= np$

$n \rightarrow$ no. of heads

Occupancy problems:- dist. of balls into bins when m balls are thrown independently & uniformly at random into n bins.

Q How many balls can we throw before it is more likely than not that some bin contains at least two balls?

find K s.t.,

$Pr[\exists \text{ bin with } \geq 2 \text{ balls after K balls thrown}] > 1/2$

→ complementary event

find K s.t.

$Pr[\text{every bin has } \leq 1 \text{ ball after K balls thrown}] \leq 1/2$.

$\forall$ bin

Let $\mathcal{E}_k$ = event that the k-th ball lands in a diff. bin than the first k-1 balls.

$\Rightarrow \mathcal{E}_k \cap \mathcal{E}_{k-1} \cap \mathcal{E}_{k-2} \cap \dots \cap \mathcal{E}_1$

$Pr\left[\bigcap_{i=1}^{k} \mathcal{E}_i\right] = Pr\left[\mathcal{E}_k \mid \bigcap_{i=1}^{k-1} \mathcal{E}_i\right] \cdot Pr\left[\bigcap_{i=1}^{k-1} \mathcal{E}_i\right] =$

$Pr\left[\mathcal{E}_k \mid \bigcap_{i=1}^{k-1} \mathcal{E}_i\right] \cdot Pr\left[\mathcal{E}_{k-1} \mid \bigcap_{i=1}^{k-2} \mathcal{E}_i\right] \cdot Pr\left[\bigcap_{i=1}^{k-2} \mathcal{E}_i\right]$

$$= P_r\left[\mathcal{E}_k \mid \bigcap_{i=1}^{k-1} \mathcal{E}_i\right] \cdot P_r\left[\mathcal{E}_{k-1} \mid \bigcap_{i=1}^{k-2} \mathcal{E}_i\right] \cdots P_r[\mathcal{E}_1]$$

$$\Rightarrow P_r\left[\mathcal{E}_i \mid \bigcap_{j=1}^{i-1} \mathcal{E}_j\right]$$

If $(i-1)$ bins had already 1 ball, then prob. of failure if total # bins $= n$,

$\therefore \frac{i-1}{n}$

$\therefore$ Prob. of success (of putting a ball in no bin that already has a ball) :

$$\Rightarrow 1 - \frac{i-1}{n}$$

$$\Rightarrow \left(1 - \frac{k-1}{n}\right)\left(1 - \frac{k-2}{n}\right) \cdots \left(1 - \frac{1}{n}\right) \cdots 1$$

$$\Rightarrow \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right)$$

(after i bins have been filled, # bins total $= n$).

Ans

By taylor expansion,

$$1 + x \leq e^x, \quad \forall \text{ real } x$$

Expanding this,

$$\leq \prod_{i=1}^{k-1} e^{-i/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-\frac{1}{n} \cdot \frac{k(k-1)}{2}}$$

going back to the prob. eq^n,

$$e^{-\frac{1}{n}\frac{k(k-1)}{2}} \leq 1/2$$

$$2 \leq e^{1/n \cdot \frac{k(k-1)}{2}}$$

$$\ln 2 \leq \frac{1}{n} \cdot \frac{k(k-1)}{2}$$

$$2n \ln 2 \leq k(k-1) \quad \{\text{Birthday paradox}\}$$

$$2n \ln 2 \leq k^2$$

$$k \geq \sqrt{2n \ln 2}$$

$$\Rightarrow \boxed{k = \Omega(\sqrt{n})}$$

Ans.

Q2 what is expected load of a bin after m balls are thrown?

⇒ Let X = # balls in a bin.

we want $E[X]$,

Let $X_i = \begin{cases} 1, & \text{if ball } i \text{ falls in this bin.} \\ 0, & \text{otherwise.} \end{cases}$ } for a bin

$1 \leq i \leq m$.

$$X = \sum_{i=1}^{m} X_i$$

$$\Rightarrow E[X] = \sum_{i=1}^{m} E[X_i] = \sum_{i=1}^{m} 1 \cdot P_r[X=1] = m\left[\frac{1}{n}\right]$$

$= m/n$

A.

Q. What is expected # empty bins after m balls are thrown?

Ans Let Y = # empty bins.

we want $E[Y]$

Let Y_j = $\begin{cases} 1, & \text{if bin } j \text{ is empty} \\ 0, & \text{otherwise.} \end{cases}$ $\quad 1 \le j \le n$

$$\Rightarrow E[Y] = E\left[\sum_{j=1}^{n} Y_j\right] = \sum_{j=1}^{n} E[Y_j] = \sum_{j \ge 1} 1 \cdot Pr[Y_j = 1]$$

$\Rightarrow$ we want $Pr[Y_j = 1]$ after m balls are thrown!

↙

that means

{ ball 1 doesn't land in bin j, ball 2 doesn't in bin j, etc... {ball m doesn't land in bin j}

$\therefore \Rightarrow$ (∵ events are independent) → prod. of all events

$Pr[Y_j = 1] = Pr[$ ball 1 didn't lie in b_j $]$

$\cdot Pr[$ ball 2 didn't lie in b_j $]$

$\cdots$

$Pr[$ ball m didn't lie in b_j $]$.

$\Rightarrow Pr[$ ball i doesn't lie in b_j $]$.

$\Rightarrow \left(1 - 1/n\right)^m$

for $E[Y] \Rightarrow \boxed{n\left(1 - \frac{1}{n}\right)^m}$ Ans.

n → no. of bins

m → balls thrown

$E[Y] \leq n\, e^{-m/n}$ (by exp. series expansion).

If $m=n$,

$E[Y] \approx \frac{n}{e} > \frac{n}{3}$ (hence)

$E[Y] > n/3$ → more than $1/3^{rd}$ of the entries are expected to be empty

Ans

Q4. What is the load of the fullest bin with high probability?

[when n balls thrown ind. & uniformly random into n bins]

as $n \to \infty$, prob. happens to be 1.

that's high prob,

w.h.p means → $Pr \geq 1 - 1/n$

(with even stronger prob. for high enough n).

$Pr \geq 1 - 1/e^n$

(for weaker)

$Pr \geq 1 - 1/\sqrt{n}$

we will consider this one here!

find k^* such that:-

Pr [every bin has $\leq k^*$ balls] $\geq 1 - 1/n$

Complementary event :-

find k^* st.,

$$\Pr[\exists \text{ bin with } \geq k^* \text{ balls}] \leq 1/n$$

by boosting property

a particular bin

① $\Pr[\text{bin has } \geq k \text{ balls}] \Rightarrow \because$ max. no. of balls thrown $= n$.

$$\Rightarrow \sum_{j \geq k}^{n} \Pr[\text{bin has } j \text{ balls}]$$

$$\Rightarrow \sum_{j \geq k}^{n} \binom{n}{j} \left(\frac{1}{n}\right)^j \left(1-\frac{1}{n}\right)^{n-j}$$

Ans.

$$\leq \sum_{j \geq k}^{n} \binom{n}{j} \left(\frac{1}{n}\right)^j$$

[$\because$ this is less than 1, so, if we remove this, we can upper bound the left-out part].

$$\binom{n}{j} = \frac{n!}{j!(n-j)!} = \frac{(n-j+1)\ldots n}{j!}$$

$$\leq \frac{n^j}{j!}$$

$$j! \approx \sqrt{2\pi j} \cdot \left(\frac{j}{e}\right)^j \left(1 + O\left(\frac{1}{j}\right)\right)$$

$$j! \geq \left(\frac{j}{e}\right)^j \Rightarrow \frac{1}{j!} \leq \left(\frac{e}{j}\right)^j$$

$$\leq \sum_{j=1}^{n} \binom{n}{j} \left(\frac{1}{n}\right)^j = \sum_{j=k}^{n} \frac{1}{j!}$$

$$\leq \sum_{j=k}^{n} \left(\frac{e}{j}\right)^j \leq \sum_{j=k}^{n} \left(\frac{e}{k}\right)^j$$

$$\leq \sum_{j=k}^{\infty} \left(\frac{e}{k}\right)^j = \left(\frac{e}{k}\right)^k \left(\frac{1}{1 - e/k}\right)$$

$$Pr[\text{bin has } \geq k \text{ balls}] \leq \frac{(e/k)^k}{[1 - e/k]}$$

2) $Pr[\exists \text{ bin with } \geq k \text{ balls}]$ ⟶ ∴ there exist, there events will occur independently

$$Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} Pr[\mathcal{E}_i]$$

$Pr[\exists \text{ bin with } \geq k \text{ balls}]$

$$\leq \sum_{i=k}^{n} Pr[\text{bin has } \geq k \text{ balls}]$$

$$= \frac{n\,(e/k)^k}{1 - e/k}$$

Any

2) $\frac{n\,(e/k)^k}{1-e/k} \leq \frac{1}{n}$

Now, we solve this!

$k^* = \gamma \frac{\ln n}{\ln(\ln n)} \rightarrow \Theta(\ln n)$ Ans.

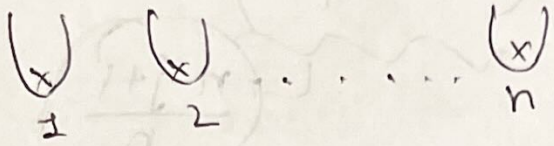
→ with fresh table, in worst-case you can have $\log n$ time (going by this).

(Applicable to randomized algos.)

what is the expected no. of balls until every bin has atleast 1 ball (Coupon Collectors Problem)?

↳ non-empty.

Ans. Lower bound 2) no. bins = (n).

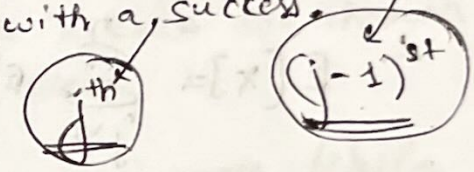


→ Initially, prob will be higher as there are lots of empty bins & then it goes smaller step by step.

Success:- ball that lands in an empty bin.

Epoch:- Sequence of balls starting after a success and ending with a success.

(j)



→ $X_j \Rightarrow$ # balls in epoch j.

Let $X \Rightarrow$ # balls until every bin has ≥ 1 ball.

$$X = \sum_{j=1}^{n} X_j$$

$$E[X] = E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E[X_j]$$

$X_1 = 1$ (in 1st epoch).

$X_2 = K \longrightarrow$ (no. of balls need to throw until we hit an empty bin).

don't know!

$$Pr[X_2 = k] \Rightarrow \left(\frac{1}{n}\right)^{k-1}\left(1-\frac{1}{n}\right)$$

failures

k-1 thrown in bin 1 &

kth time is a success. (this a diff. bin).

$Pr[X_3 = k] \Rightarrow \left(\frac{2}{n}\right)^{k-1} \cdot \left(1 - \frac{2}{n}\right)$

$\vdots$

$Pr[X_j = k] \Rightarrow \left(\frac{j-1}{n}\right)^{k-1} \cdot \left(1 - \frac{j-1}{n}\right)$

$\left(1 - \frac{j-1}{n}\right) \rightarrow \frac{n-j+1}{n}$

$\vdots$

$P[X_n = k] \Rightarrow \left(1 - \frac{1}{n}\right)^{k-1} \cdot \frac{1}{n}$

$\therefore E[X] = \sum_{j=1}^{n} E[X_j]$

X_j is geometrically distributed with success prob $\Rightarrow \frac{n-j+1}{n}$

$Pr[X = k] = \boxed{(1-p)^{k-1} \cdot p}$

$\boxed{E[X] \Rightarrow 1/p}$

$\Rightarrow \sum_{j=1}^{n} \frac{n}{n-j+1} \Rightarrow n \sum_{j=1}^{n} 1/n-j+1$

$\Rightarrow n \sum_{k=1}^{n} 1/k$

$\sum_{j=1}^{n} 1/j \triangleq H_n$ (nth harmonic no.)

$\boxed{H_n \approx \ln n + \Theta(1)}$

⇒ Run-time to find ⇒

$\binom{n}{2} \cdot n \Rightarrow O(n^3)$ (efficient algo.)

possible comb^n of vertices.

n → find that edge.

⇒ Verification algo. for Certifiers.

→ Input to $VC(D) :- \langle G, b, k \rangle$

→ Input to certifier for $VC(D) :- (x, t)$

→ What does the certifier do with its input?

(1) $\leq k$ nodes in b.

(2) t forms a VC.

↔ Can be solved in linear time to find the vertices set of t and for the possible sets $\binom{n}{2} \cdot n$

∴ $O(n^3)$

poly time → thus, efficient algo

∴ $X(D) \in NP$ if $\exists$ efficient (*) certifier (verification algo) B that takes TWO inputs, the instance x & the certificate t, and it is s.t.

$x \in X(D) \iff \exists$ short (**) t s.t. $B(x, t) = yes$.

Non-deterministic → guess a certificate!

* eff :- worst-case poly-time. (bcoz we want whole algo in poly time)

(*) short :- $|t| = poly(|x|)$

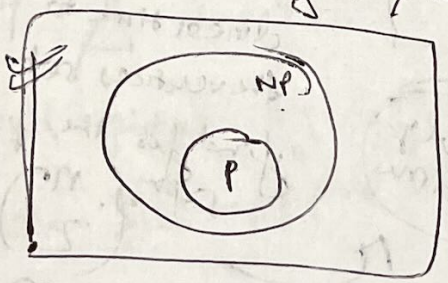
⇒ Note the existence of the certifier B does not provide us with any efficient way to solve $X(D)$!

Imp (bcoz certifiers should be efficient ~~to solve it~~ in polynomial time).

(*) $P \subseteq NP$

Is $P = NP$?

→ Not! finding a sol^n should be harder than checking one, especially for hard problems...



-hardest probs. in NP are least likely to be in P!

NP-complete problems :-

A problem $X(D)$ is NP-complete if

(1) $X(D) \in NP$ and

(2) for all $Y \in NP$, $Y \leq_p X(D)$.

$X(D)$ is at least as hard as Y.

FACT ⇒ Suppose X is NP-complete. Then X is solvable in polynomial time (i.e., $X \in P$) if & only if $P = NP$.

Sol^n to Boolean CSP, which

How do we show that a problem is NP-Complete?

Suppose we find an NP-complete problem X.

To show that another problem Y is NP-complete, we only need to show that:-

(1) $Y \in NP$ and

(2) $X \leq_p Y$

→ this provides an easier to prove more NP-complete problems.

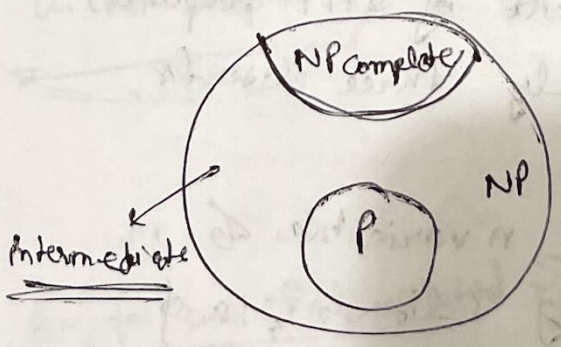
FACT → Transitivity of reductions.

If $X \leq_p Y$ and $Y \leq_p Z$, then $X \leq_p Z$.

→ We know that for all $\pi \in NP$, $\pi \leq_p X$. By fact $\pi \leq_p Y$. Hence, Y is NP-complete. Hw

(*) The first NP-Complete problem.

→ Circuit SAT is NP-complete!



CNF:—

formula ϕ in CNF if it consists of conjunction of clauses each of which is a disjunction of literals.

ϕ with m clauses is in CNF if:-

$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$

(each clause C_i is disjunction)

$C_i = (l_1 \vee l_2 \vee \ldots \vee l_k)$.

{∴ we will work in CNF now! }

⇒ A truth assignment satisfies a clause if it causes the clause to evaluate to 1.

⇒ A truth assignment satisfies a formula in CNF if it satisfies every clause in the formula.

⇒ A formula ϕ is satisfiable if it has a satisfying truth assignment.

Satisfiability (SAT) & 3 SAT →

Given a formula ϕ in CNF with n vars & m clauses, is ϕ satisfiable?

A convenient (& not easier) variant of SAT requires that every clause consists of exactly three literals.

3SAT ⇒

Given a formula ϕ in CNF with n variables & m clauses s.t. each clause has exactly 3 literals, is ϕ satisfiable?

⇒ These problems are NP-complete problems. (hardest problems in NP!).

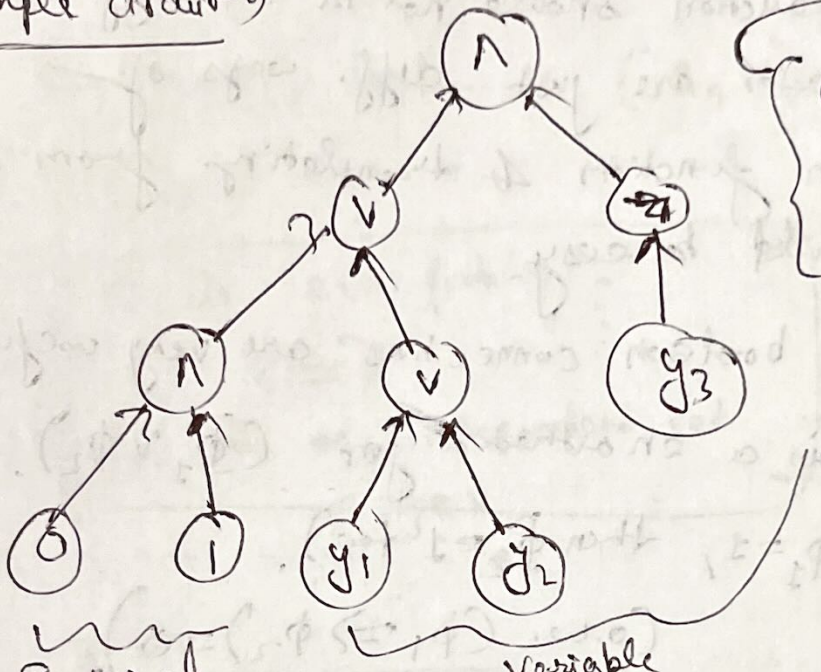
$B = (\phi, t)$

Circuit-SAT

Given a circuit C, is there an assignment of truth values to its inputs that causes the o/p to evaluate to 1.

Circuit SAT is NP. (also NP-complete).

Example circuit :)



hardwired source nodes.

Variable inputs.

→ this idea below sets up the size of ϕ.

Circuit SAT ≤p SAT

$C_1 \longrightarrow \phi$

Given a circuit C, is there an assignment of truth values to its I/P: that causes o/p to evaluate to 1?

(Inorder traversal of the tree)

CNF:- $((0 \wedge 1) \vee (y_1 \wedge \vee y_2)) \wedge (\neg y_3)$

Could be reduced to CNF if not using laws.

To prove SAT is NP-complete

So, for we have stated :)

→ Circuit-SAT is NP-complete

→ Circuit-SAT ≤p SAT

→ SAT ≤p 3-SAT

⇒ SAT & 3-SAT are NP-complete.

(**) Is IS(D) as "hard" as SAT?

Before doing this,

Lemma → Circuit-SAT $\leq_p$ SAT

Intuitively, this reduction should not be too difficult :- formulas and circuits are just diff. ways of representing boolean functions & translating from one to the other should be easy.

The following two boolean connectives are very useful.

1. $(\phi_1 \Rightarrow \phi_2)$ is a shorthand for $(\bar{\phi}_1 \vee \phi_2)$.

(**) Intuition :- if $\phi_1 = 1$, then $\phi_2 = 1$ too).
(o.w. $(\phi_1 \Rightarrow \phi_2) = 0$).

2. $(\phi_1 \Leftrightarrow \phi_2)$ is a shorthand for $((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))$,

which may be expended to $(\bar{\phi}_1 \vee \phi_2) \wedge (\phi_1 \vee \bar{\phi}_2)$.

(**) This clause evaluates to 1 iff. $\phi_1 = \phi_2$.

Transforming C into ϕ requires polynomial time (**)

This completes our construction of the clauses of Φ.

Construction → polynomial in the size of the I/P circuit.

Moreover, every clause consists of at most 3 literals once ϕ is in CNF.

(*) for independent set :-

IS(D) is hardy

$$3SAT \leq_p IS(D)$$

To prove: $IS(D) \in NP$.

$3SAT \leq_p IS(D)$

$\Phi\ (n, m) \longrightarrow (G, k)$

s.t.

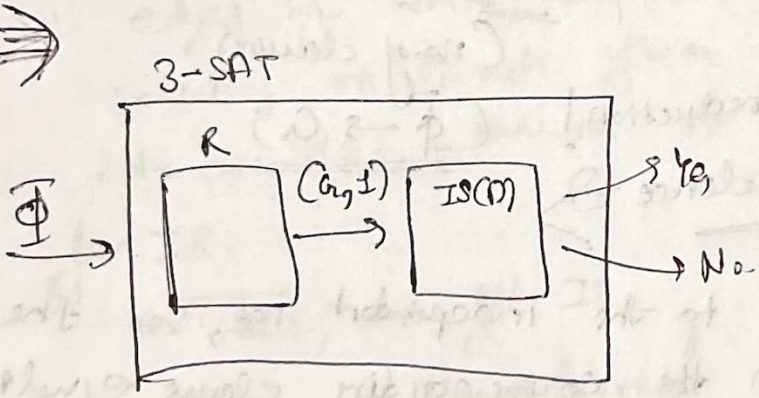
Φ is satisfiable

$\Leftrightarrow$

G has an independent set of size k.

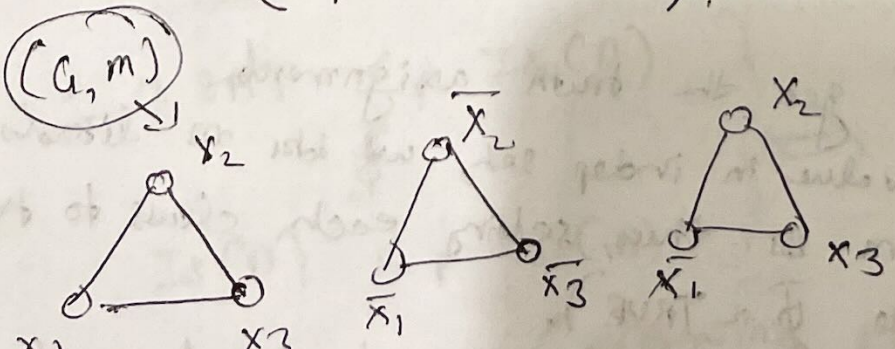
equivalence of instances

thats how it gives a "yes" instance!



$$\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3).$$

(G, m)



$x_1 = 1, \bar{x}_2 = 1, \bar{x}_3 = 1$

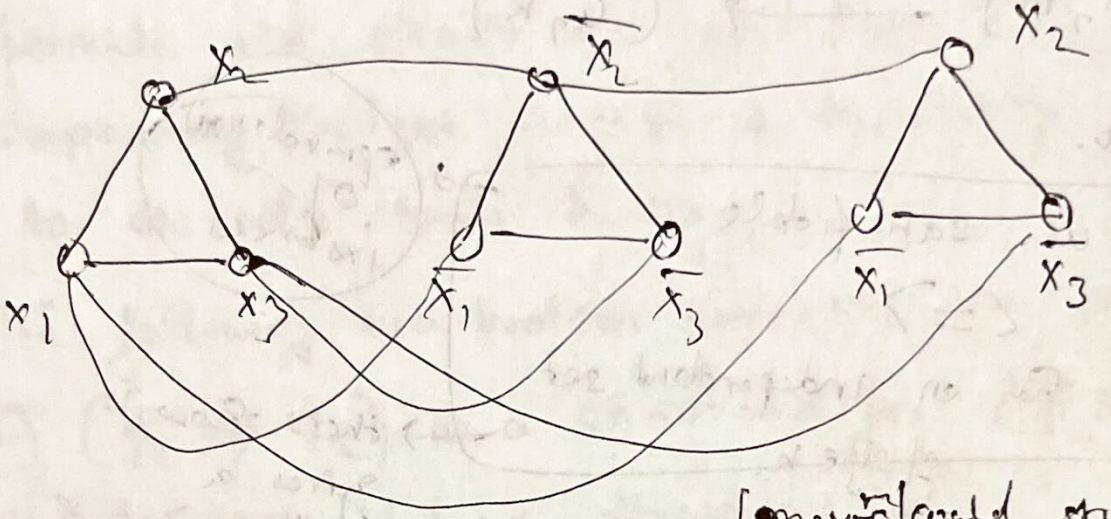
{we can get an independent set of size 3.

Thus independent set of size 3}.

→ There are so many ways to satisfy all clauses. (but each time at most we can get m literals as part of the independent set).

∴ we shouldn't have complemented literals in the same part of independent set.

for this, we will have to add edges b/w them:



|max IS| could still be m (no. of clauses).

⇒ This is a polynomial-time reduction! ($\Phi \rightarrow G$)

Now, we need to prove equivalence ⇒

Idea → If a node belongs to the independent set, set the corresponding literal in the corresponding clause equal to 1.

Proof ←

If G has an independent set of size m, then Φ is satisfiable.

Use idea to get the truth assignments.

(∵ we take m values in indep set, we take m literals from each of the m ∆s then, setting each clause to true which would make Φ = TRUE).

Also, the IS is valid as complemented literal, can't both belong to an IS bcoz of the edges b/w them in the graph.

thus, we have proved this in backward direction.

(FORWARD)

If ϕ is satisfiable, there exists ind set of size 'm' in G.

(∵ every clause should evaluate to 1, we need to ~~satisfy~~ set at least one of the literal values in each of the clauses as 1).

∴ we will select those m nodes to be a part of an independent set. Now, we need to prove that the set formed is an ind. set.

∵ we have complemented the literals and formed edges b/w them, we can't take those & also we are selecting m literals from each clause, thus, we take care that there is no edge b/w nodes considered for IS.

Thus, we get IS.

Hence, Proved

∴ 3-SAT problem is NP-complete.

∴ 3 SAT $\leq_p$ Is (D)

Is (D) $\leq_p$ VC (D)

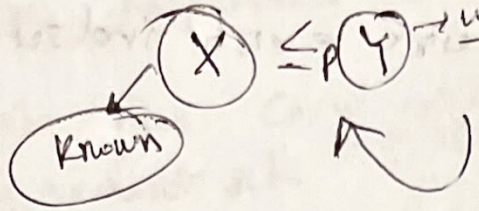
(by transitive property)

$\Rightarrow$ 3-SAT $\leq_p$ VC (D)

∴ IS(n) is atleast as hard as 3-SAT.

Any

Pitfalls ⇒



↘ Y is my prob.

this is a pit fall!

⇒ In absence of ideas, reduce from 3-SAT.

(o.w. we should see which prob is most suitable to reduce from).

Traveling Salesman Problem (TSP):-

Tour - a simple cycle that visits every vertex exactly once.

Defⁿ ⇒

Given n cities $\{1, \ldots, n\}$, a set of non-negative distances d_{ij} b/w every pair of cities & a budget B, is there a tour of length $\leq B$?

Equivalently, is there a permutation π s.t

(1) $\pi(1) = \pi(n+1) = 1$, that is, we start & end at city 1.

(2) total distance travelled satisfies

$$\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)} \leq B$$

Appⁿ's Google street view car.

ple instances of TSP :-

→ Asymmetric :- $d_{ij} \neq d_{ji}$ → directed graphs

→ Symmetric :- $d_{ij} = d_{ji}$ → undirected graphs

→ Metric :- satisfy the Δ inequality $d_{ij} \leq d_{ik} + d_{kj}$

→ Euclidean :- eg :- cities are in R^2 hence city i corresponds to pt. (x_i, y_i), then,

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

using approximate algo, we can solve for these type of instances but not for "asymmetric" Ans

⇒ A related problem & hardness of TSP(D) ⇒

Hamiltonian cycle :- given a graph $G = (V, E)$, is there a simple cycle that visits every vertex exactly once ?

Imp

Claim - 1

Hamiltonian cycle is NP-complete.

Proof :- Reduction from 3-SAT.

Claim - 2

Symmetric TSP(D) is NP-Complete.

Proof :- Red^n from undirected Hamiltonian cycle.

graph for Ham cycle is unweighted

Ham. cycle → tour of length n in unweighted graph!

Q How to prove TSP ∈ NP (define a certifier!)

→ B: (x, π) ?

↳ permutation of vertices.

This is a polynomial time certifier.

∴ TSP is in NP!

Now, to prove it is NP complete, Ham cycle $\leq_p$ TSP(D)

↓

To Prove

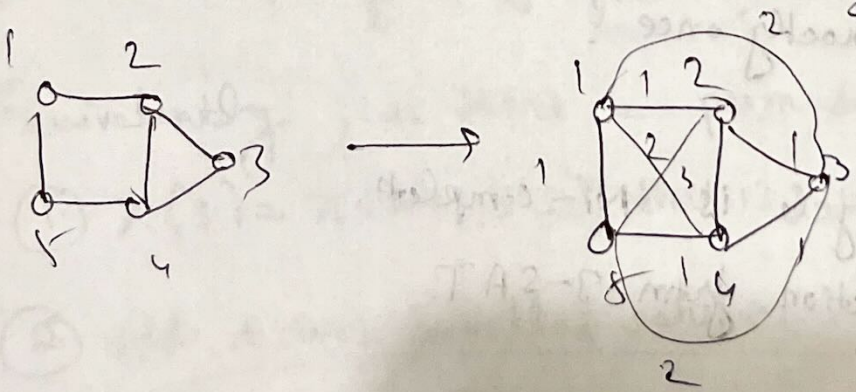
Ham cycle is already a decision problem, so, we don't need to convert it.

I/P →

$G = (V, E)$ → Convert to → $(G' = (V', E', w), B)$ ← budget

in polynomial time

S.t G has a ham cycle (⟺) G' has a tour of length B.



Complete Graph formation

The vertices in G' remain the same in G. The edges in G is replicated in G' with weight = 1

However, in the TSP, we need to know the weight (paths) of all possible vertices, ∴ we make all possible edges (even those not in G). Now G' becomes a complete graph. Further, we set the wts. of these new edges to an arbitrary large #. To keep it clean, we set it to 2.

We set the budget to be 'n', since a ham cycle is a tour of length n, ∴ $\boxed{B = n}$ → Red^n in polytime!

Equivalence of instances:

⟹ Forward dir^n:

If G has a ham cycle, G' has a tour of length 'n'! The edges of the tour in G' will be exactly the same as in ham cycle G. The weight of the tour would be 'n' since each of these edges also appear in G & the edge wts in G' = 1.

⟸ Reverse:

Given a tour of length 'n' in G', exhibit a ham cycle in G.

G' is a complete graph, ∴ it has at least one hamiltonian cycle. Given a tour of length 'n', it means that every edge wt in the ham cycle must be one, ∴ a tour of length 'n' in G' forms a ham cycle in G. (This is because all the 'n' edges in the tour in G' also appear in G.)

(*) Approximation algorithm

An α-approx. algo for an optimization problem is a poly-time algo that, for all instances of the problem, produces a soln whose value is within a factor of α of the value of the optimal soln.

Remark:) α is the approx ratio / factor.

→ For min. prob, $\alpha \geq 1$

→ For max prob, $\alpha < 1$.

To Prove: $\nexists$ 2 approx algo to prove TSP.

→ we'll prove by contradiction

Assume that $\exists$ 2 approx algo for TSP

If $A(G') \leq 2n$, then G has a ham cycle

If $A(G') \geq 2n+1$, then G does not have a ham cycle.

An approx. algo is an efficient algo! we observe the above is true, then we'll get some answer in poly time, which is not possible unless P = NP.

$\geq n-1+(n+1) \geq 2n+1$ ∴ we conclude that there does not exist a 2-approx. algo.

(*) It is possible to provide very good approximate soln to euclidean TSP, which is not for symmetric TSP

Linear Programming

LP describes a broad class of optimization problems where both the constraints and the objective function are linear functions.

Applications
1) Resource allocation
2) Production planning
3) military strategy forming
4) Error correction
5) graph theoretic problems.

It can establish existence of polynomial time (efficient) algos.
Guide the design of approximation algos for computationally hard problems.
Duality allows to unify abstract views of seemingly unrelated results and is useful in algo design

---

* GOALS

1. Structure of an LP.
2. Geometry of the solution in small dimensions.
3. LP solvers as a black box.
4. Integer LP and NP-completeness.

Q A boutique chocolatier has 2 products: Chocolates (C) & truffles (T)

their profit is: \$1/C , \$6/T

They can produce ≤ 400 boxes/day

daily demand is limited to

1) ≤ 200 boxes of C 2) ≤ 300 boxes of T

What are the optimal levels of production?

⇒ Variables: x_1 = # of boxes of chocolate

x_2 = # of boxes of truffles

$x_1 \geq 0$, $x_2 \geq 0$ (special constraints)

CONSTRAINTS: $x_1 + x_2 \leq 400$

$x_1 \leq 200$

$x_2 \leq 300$

Objective: Maximize $x_1 + 6x_2$

Maximize $x_1 + 6x_2$

$x_1 \geq 0, x_2 \geq 0$

subject to $x_1 \leq 200$

$x_2 \leq 300$

$x_1 + x_2 \leq 400$

(6)

e general problem

1. a set of variables
2. a set of linear constraints on the variables (equalities / inequalities ($\geq$, $\leq$ ones)
3. a linear objective function to maximize (or minimize)

O/P An assignment of real values to the variables such that
1. the constraints are satisfied.
2. the objective fn is maximized (or minimized)

---

* Terminology

1. Solution: An assignment of real values to the variables.
2. Feasible sol^n: sol^n that satisfies all the constraints (including the special ones)
3. Feasible region: the set of all feasible solutions
4. Optimal solution: a feasible sol^n that maximizes (or minimizes) the objective function if the LP is a maximization (minimization) LP
5. Cost/value of sol^n: the value of the objective fn for this sol^n.
6. optimal value of LP: the value of an optimal soln.

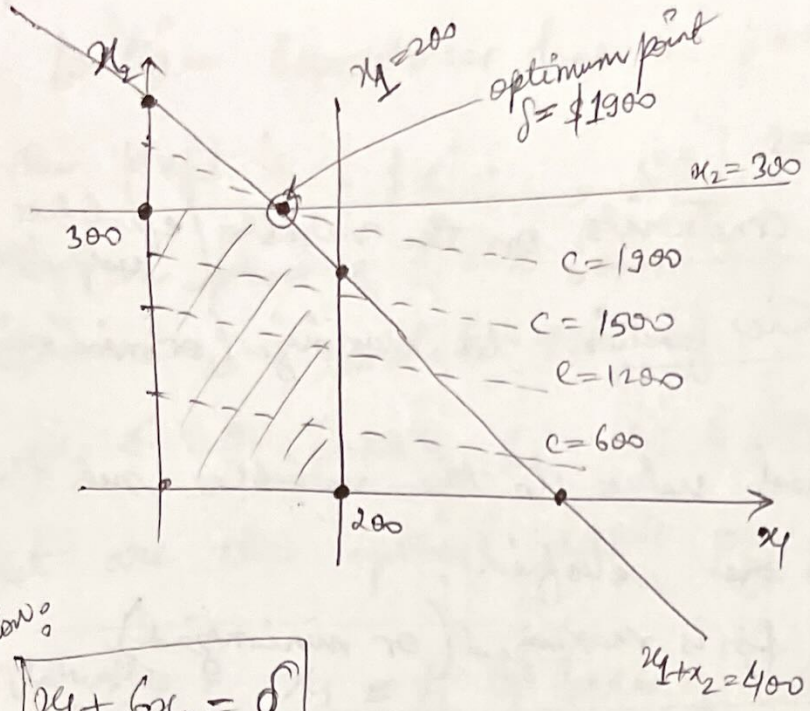
Let's find the feasible region now!

↓

It will always be convex! (by construction)

↓

means that if you draw a line segment, the line must fully lie in the set.



Now, we need to find the optimal sol^n in this feasible region.

objective function:

$$x_1 + 6x_2 = \delta$$

defines a line on the plane that contains all points that achieves a profit of δ.

---

* Theorem

The optimum is achieved at a vertex of the feasible region

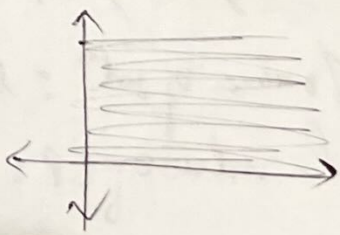
EXCEPTIONS

1. The linear program is infeasible (no solution)

eg: $x \leq 1, x \geq 2$

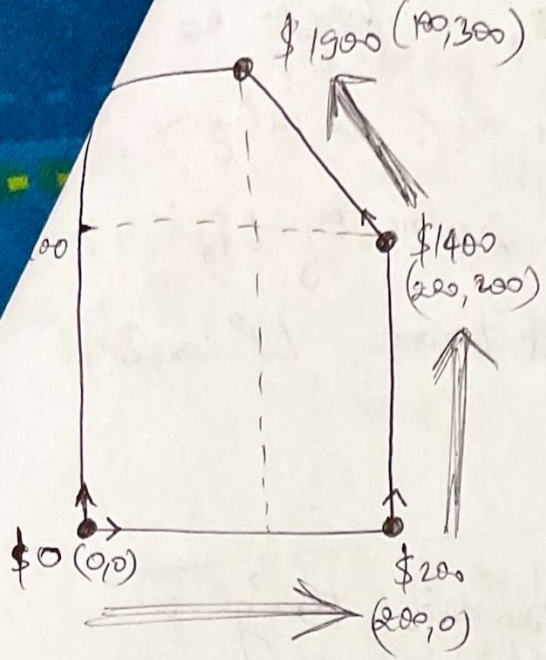
2. The optimum value is unbounded

$$\max_{x_1 \geq 0,\ x_2 \geq 0} x_1 + x_2$$



---

* Solving LPs : the simplex method [Dantzig]



algo.

It is a hill climbing ~~problem~~.

(1) Find the vertex to start with. "First vertex". Let's say (0,0) in the feasible region

(2) Move towards the direction of improvement, i.e, an adjacent vertex that ↑ the value of the current ~~vertex~~ vertex.

(3) Keep repeating this you're stuck. Nowhere else to go where the value of the objective fn can be maximized further.

The vertex at which the algorithm terminates, becomes the global optima because of the convex shape of the feasible region.

→ The Simplex algorithm always moves in the direction of improvement. It is the most IMP algo of the previous century!

---

* History on LP solvers

1. Simplex Method — fast in practice
   — exponential worst case performance

2. Ellipsoid Method — provably poly time algorithm
   — slow in practice

3. Interior point Method — polynomial time algorithm
   — fast in practice.

4. Interior point methods is a field of active research.

In the problem of chocolates & truffles, we forgot to put a constraint that the # of boxes must be integers. If we do so, we are no longer in the realm of LP, but we enter integer LP (ILP). This is because LP consists of solutions containing real values.

Also, this problem could have been solved using a greedy approach / without LP. Prof used this just to motivate the problem & draw a graph. (represent visually on the plane)

→ ILP problems are not solvable in polynomial time!

28th April 2022

(d)

alternative proof that $1900 is optimal

PRIMAL LP

max $x_1 + 6x_2$

s.t. $x_1, x_2 \geq 0$

subject to

$(x_1 \leq 200) \cdot 0$

$(x_2 \leq 300) \cdot 5$

$(x_1 + x_2 \leq 400) \cdot 1$

$0 \cdot x_1 + 5 \cdot x_2 + x_1 + x_2 \leq 200 \cdot 0 + 300 \cdot 5 + 400 \cdot 1$

$x_1 + 6x_2 \leq 1900$

How did we come up with these numbers?

① Multiply these constraints with ~~none~~ different non-negative real #s. so that they don't change the direction of the inequality

② Add the LHS and RHS of those constraints separately

③ Make the LHS similar to the objective function.

~~[illegible]~~

$(x_1 \leq 200) \cdot y_1$

$(x_2 \leq 300) \cdot y_2$

$(x_1 + x_2 \leq 400) \cdot y_3$

s.t. $y_1, y_2, y_3 \geq 0$

$y_1 \cdot x_1 + y_2 \cdot x_2 + y_3 x_1 + y_3 x_2 \leq 200 y_1 + 300 y_2 + 400 y_3$

$(y_1 + y_3) x_1 + (y_2 + y_3) x_2 \leq 200 y_1 + 300 y_2 + 400 y_3$ — ①

Now, we must upper bound our objective function.

If we can upper bound it by the LHS of ①, then we can also upper bound it by the RHS of ①.

So, $x_1 + 6x_2 \leq (y_1 + y_3) x_1 + (y_2 + y_3) x_2$

To make sure that this inequality holds:-

$y_1 + y_3 \geq 1$ and $y_2 + y_3 \geq 6$

This becomes:- **DUAL LP**

$$\min \quad 200y_1 + 800y_2 + 400y_3$$

$y_1 \geq 0$
$y_2 \geq 0$
$y_3 \geq 0$

s.t.

$$y_1 + y_3 \geq 1 \text{ and } y_2 + y_3 \geq 6$$

minimization

OBSERVATIONS

1) If the primal LP has m constraints, there are m variables in the dual.

2) ~~[illegible]~~ The # of variables in the primal = # of constraints in the dual.

∴ for every primal variable, we get a dual constraint.

3) If the primal is a maximization, then the dual is a minimization and vice versa.

Let v_p be the cost of any solution to the primal LP
Let v_D be the cost of any solution to the dual LP.

WEAK DUALITY → $v_p \leq v_D$ by construction

Cost of optimal Soln

If dual is maximization, then $v_p \geq v_D$