ANALYSIS OF ALGORITHMS NOTES

- BY ANISH ADNANI

PAGE No.
DATE / /

AOA Notes week 1

Stable Matching.

'N' men : $\{M_1, M_2, \ldots M_n\}$

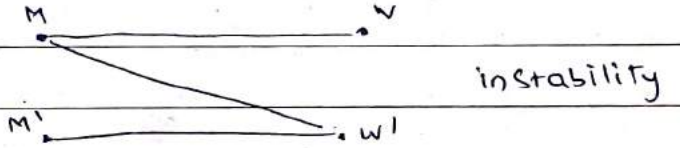
'N' women : $\{w_1, w_2, \ldots w_n\}$

Matching is a set of ordered pairs

perfect Matching: each member of M and each member of W only exist in one pair

All men rank women acc to their pref

similarly all women rank men acc to their pref



we need perfect and stable matching

Gale Shaplay Algorithm

$M = \{m_1, m_2, m_3\}$ $W = \{w_1, w_2, w_3\}$

Men proposing

$M_1 : \{w_1\ w_2\ w_3\}$ $w_1\ \{m_2\ m_1\ m_3\}$

$M_2\ \{w_1\ w_3\ w_2\}$ $w_2\ \{m_1\ m_2\ m_3\}$

$M_3\ \{w_2\ w_1\ w_3\}$ $w_3\ \{m_3\ m_2\ m_1\}$

STABLE MATCHING

w_1	~~M_1~~ M_2
w_2	~~M_3~~ M_1
w_3	M_3

side that does proposing in G-S algo ends with the best possible stable matching (from their perspective).

PAGE NO.
DATE / /

From women's perspective, she starts single, and once she gets engaged, she can only get into better engagements.

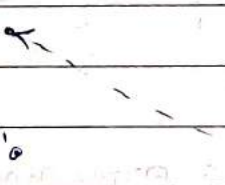
From mans perspective, he start single, gets engaged, and may get dropped repeatedly only to settle for lower ranking woman.

Soln terminates in atmost (n^2) iterations

Proof of correctness:

proof by contradiction

Assume instability exists in our soln involving two pairs (m,w) (m', w')



Say (mw') is instability

Q: Did m propose to w' at some point in the execution.

A: ~~No~~ If no, w must be higher than w' on his list → contradiction.

If yes, he must have been rejected in favor of m'' and due to ① either $m'' = m'$ or m' is better than m'' ↯ contradiction.

② Identify highest ranked woman to whom m has not yet proposed O(1)

Next [1 . . . N]

Next [m]

↳ Next women 'm' will be proposing to

PAGE No.
DATE / /

ManPref [1.. n, 1.. n]

ManPref [m, i] denotes i^{th} women on man 'm' list

(☆) To find next woman w to whom m will be proposing to — O(1)

w = ManPreF [m, Next[m]]

(☆) Determine w's status — O(1)

Current[1.. N]

initially current[1.. N] = null

once male 'm' assigned to women w3

current[3] = m

(☆) Determine which man is preferred by w — O(1)

Women ranking

mans	men10	. . ..
1	2	

Prep before entering GS Algorithm.

Create a ranking array where ranking[w,m] contains the rank of mem m based on w's pref

preparation + GS iteration

$O(n^2)$ + $O(n^2)$

$O(n^2)$

Women w is a valid partner of a man m if there is a stable matching that contains the pair (m,w)

Highest women is best valid partner of man m

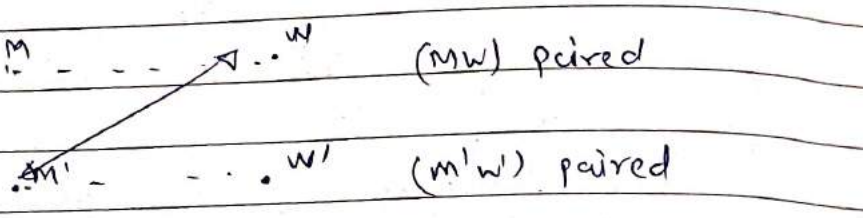
Every exec of GS algo (men proposing) results in same stable matching regardless the order in which men propose.

Best valid partner are invariant. with respect to the Algorithm

Proof by contradiction.

Say M is the first man rejected by a valid partner W in Favor of M'

IF M and W are valid partners in some pairing S' they are paired



But W prefers M' over M as ~~assumed~~ W rejected M for M'

But M' proposed w/ over W Hence pair (M'W) cant be formed.

IF M' proposed to W and M was first to get rejected (M'W) would be paired but M'W' paired

Hence Contradiction.

1924

$T(k) = 2T(k-1) + O(1)$

$T(k-1) = 2T(k-2)$

$T(1) = 2T(0)$

Discussion Week 1

$T(1) = 2T(0)$

$T(2) = 2T(1)$

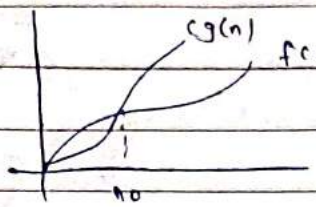
$T(3) = 2T(2)$

$T(k) = 2T(k-1) + O(1)$

$O(k)$

PAGE No
DATE / /

DOA Week 2 Notes



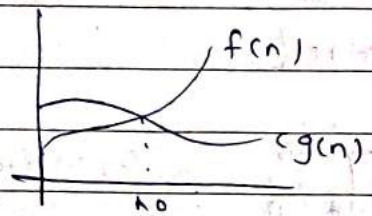
worst case

For all $n \geq n_0$

$0 \leq f(n) \leq c \cdot g(n)$

$f(n) = O(g(n))$

i.e $f(n) \in O(g(n))$

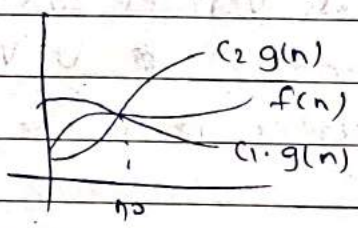


Best case

For all $n \geq n_0$

$0 \leq c \cdot g(n) \leq f(n)$

$f(n) = \Omega(g(n))$



For all $n \geq n_0$

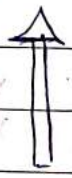
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

$f(n) = \Theta(g(n))$

Exponential growing

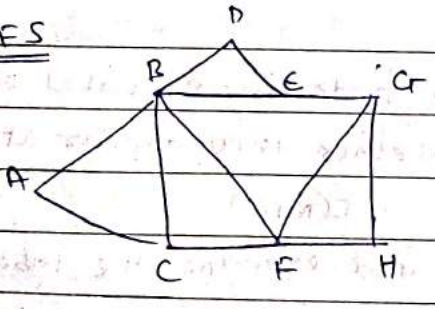
polynomial

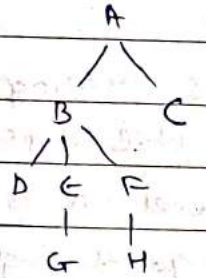
logarithmic



Fastest growing

BFS





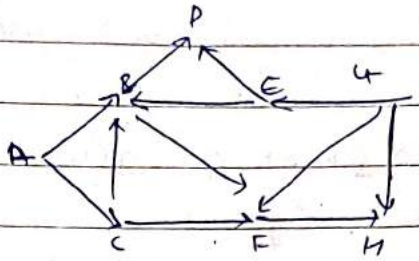
m: no. of edges

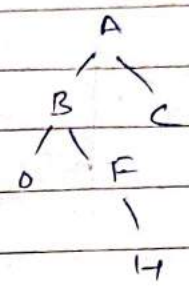
n: no. of nodes

$O(m+n)$

PAGE No.
DATE / /

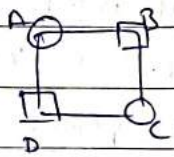
DFS

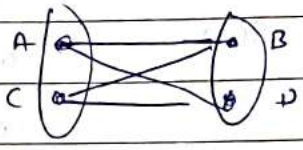




O(m+n)

Determine if graph is bipartite?

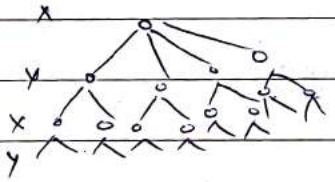




Bipartite ✓

Bipartite graph: Graph whose vertices can be divided into 2 sets (disjoint and independent) U, V ; such that every edge connects a vertex U to one in V.

If a graph G is bipartite it cannot have a odd cycle.



Find if bipartite

Method I

- Run BFS starting from any node say S : Label each node Red or Blue depending on whether they appear at an odd or even level on BFS tree O(m+n)
- Then go through all edges and examine the labels at the two ends of the edge. If all edges have a red end and a blue end, then graph is bipartite. O(m)

O(m+n)

PAGE NO.
DATE / /

(*) Directed graph is strongly connected if there is a path From any point to any other point in the graph.

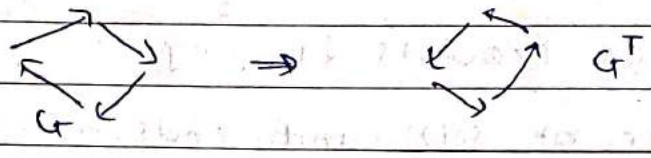
Find if DG is strongly connected?

Method 1

Run BFS/DFS 'n' times once for each node

$$O(n(m+n)) = O(n^2+mn)$$

Method 2



use BFS/DFS to Find reachable nodes from any point s. $O(m+n)$

IF some node not reachable:

return False.

else:

Find G^T $O(m+n)$

use BFS/DFS to Find all reachable nodes from point s in G^T $O(m+n)$

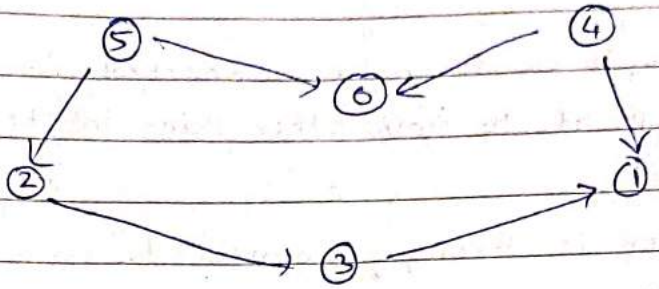
$O(m+n)$

(*) Topological sorting

Topological sorting For DAG is a linear ordering of vertices such that for every edge (u, v) vertex u comes before v in the ordering

DAG may have many topological sorting

PAGE No
DATE / /



"542310"

"452310"

To Find longest path in DAG Topological ordering is used.

(*) Interval scheduling problem.

Input: set of requests $\{1 \ldots n\}$

i^{th} req starts at s(i) and ends at f(i)

Objective: To Find the largest compatible subset of these requests.

Smallest Finish Time First

Algo

R ← Complete set of requests
A ← empty # answer
While R is not empty:
    choose a request $i \in R$ that has smallest Finish time

    Add req 'i' to A
    Delete all req from R that are not compatible with i

Return A.

Proof of correctness

(1) Show that A is a compatible set
(2) Show that A is optimal set

Algo we have written select req 'i' and removes all incompatible req.
Hence A is compatible set.

optimality proof

Say A is of size k
Say there is an optimal solution O

Req in A: $i_1, i_2, \ldots i_k$

Req in O: $j_1, j_2, \ldots j_k$

For all $r \leq k$

$$f(i_r) \leq f(j_r)$$

As our algo is always choosing arc to earliest Finish time

our schedule i_1 will end before or on same time as that of j_1

Similarly For i_2 j_2

i_n j_n

i_1 i_2 i_j

PAGE No.
DATE / /

i_1 i_2 . . . i_{k-1} i_k

j_1 j_2 . . . j_{k-1} j_k . .

Here i_{k-1} finishes at same time or before j_{k-1}

$\therefore$ i_{k-1} could choose i_k or j_k

But it choose i_k

$\therefore$ ~~Time~~ Finish-time(i_k) $\leq$ Finish Time(j_k)

Now To prove $|A| = |O|$

i.e our soln and optimal soln has same no. of schedules

A . . . i_k

O . . . . j_k j_{k+1}

i_k finishes before or at same time of j_k

~~If 'O' too~~

our algo is not able to fit 'k+1' in schedule

So Algo 'O' Finishing schedule K after our schedule 'K' cannot fit 'K+1'

Hence $|A| = |O|$

Hence A i.e our Algorithm is optimal.

PAGE NO.
DATE / /

~~Practical~~

Algo

- Sort req acc to finish Time $O(n \log n)$

- select req in order of inc f(i), always selecting the first.
Then iterate through the intervals in this order until reaching the first interval for which
$s(j) \geq f(i)$ $O(n)$

$O(n \log n)$

Fractional knapsack

knapsack weight → w

Given: set of n objects with their weight and val

Objective: To fill knapsack to its weight capacity such that the value of items in knapsack is maximized.

PAGE No.
DATE / /

AOA week 3 Notes

Goal : To Minimize the Maximum Lateness

Req can be scheduled at any time.
Each request has a deadline.

$$L_i = f(i) - d_i$$

L_i ← Lateness, $f(i)$ ← Finish Time, d_i ← deadline

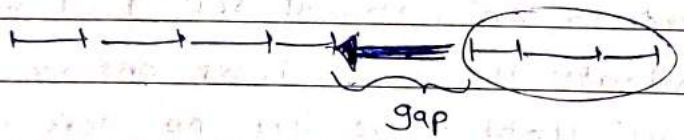
Sol 1 job1 late by 5 hrs ✓
job2 late by 4 hrs

Sol 2 job1 late by 7 hrs
job2 late by 0 hrs

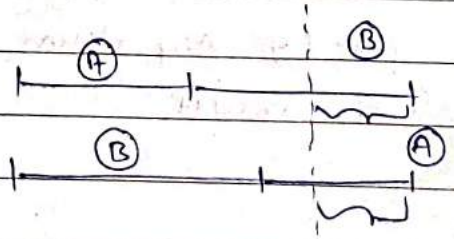
Solution:
Schedule jobs in order of their deadline without any gaps between jobs.

Proof of correctness

① There is an optimal soln with no gaps



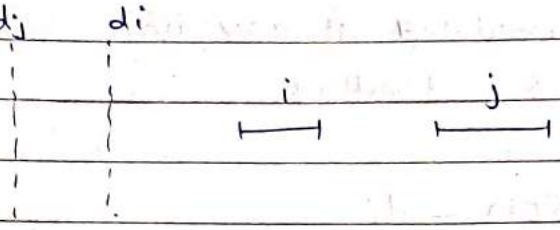
② Jobs with identical deadlines can be scheduled in any order without affecting maximum lateness



Lateness is same.

PAGE No.
DATE / /

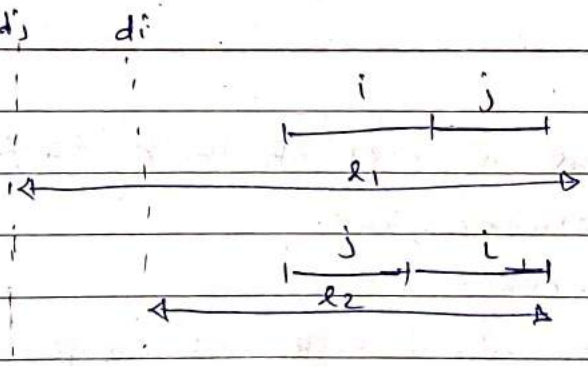
③ Schedule A' has an inversion if a job i with deadline d_i is scheduled before job j with earlier deadline d_j



Our soln has no inversions as tasks are scheduled acc to their deadline

④ All schedules with no inversions and no idle time have the same maximum lateness

⑤ There is an optimal ~~soln~~ schedule that has no inversions and no idle time.



So if there is an optimal soln that has inversions, we can eliminate the inversions one by one as shown above until there are no more inversions. This soln will also be optimal.

∴ our greedy Algorithm produces one of the optimal result.

PAGE No.
DATE / /

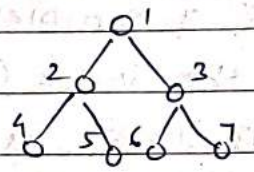
Priority Queues

A priority queue has to perform these two operations

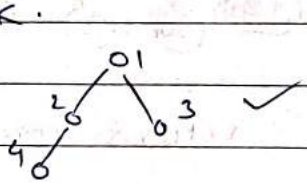
1. Insert an element into the set
2. Find the smallest element in the set.

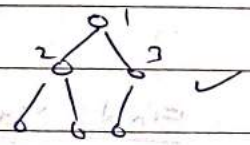
	insert	find smallest
Array implementation	O(1)	O(n)
Sorted "	O(n)	O(1)
Linked list	O(1)	O(n)
Sorted "	O(n)	O(1)

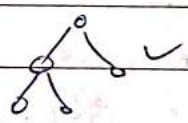
Binary Tree of depth K which has exactly $2^K - 1$ nodes is called Full Binary Tree.

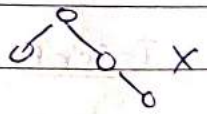


A Binary Tree with n nodes and of depth K is complete if its nodes correspond to the nodes which are numbered 1 to n in Full Binary Tree of depth K.









∴ A complete binary Tree is a binary Tree in which every level, except possibly last, is completely Filled, and all nodes are as far left as possible.

PAGE No.
DATE / /

Traversing a complete binary stored as an array

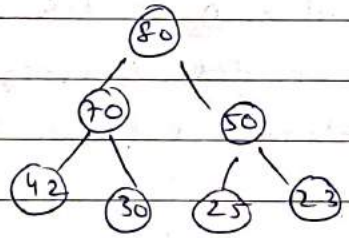
Parent (i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
if $i=1$, i is root

L child (i) is at $2i$ if $2i \leq n$
otherwise no left child

Rchild (i) is at $2i+1$ if $2i+1 \leq n$
otherwise no right child.

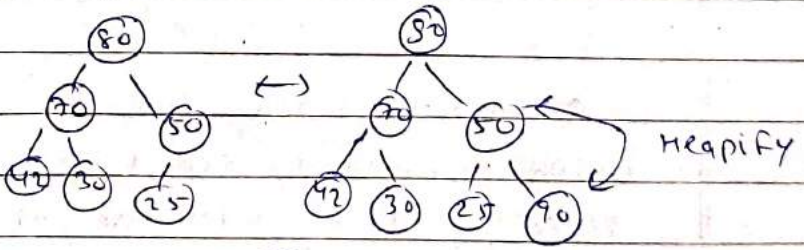
(*) Binary Heap

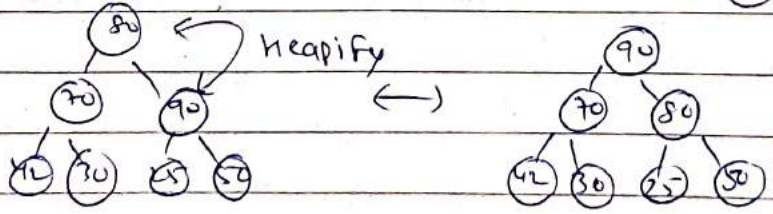
Binary Heap is a complete binary Tree with the property that the value (of the key) at each node is at least as large as the values at its children (Max Heap)



Find Max : return root O(1)

Insert Insert 90



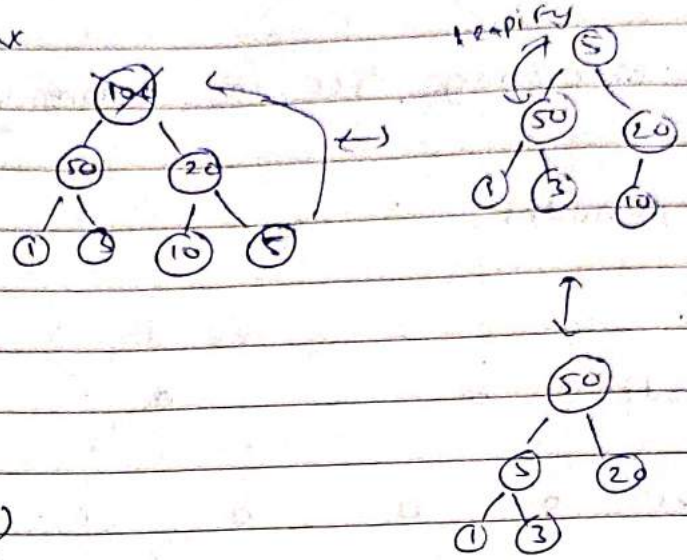


there are $\log n$ levels
Hence maximum of $\log n$ operations

$O(\log n)$

PAGE No
DATE / /

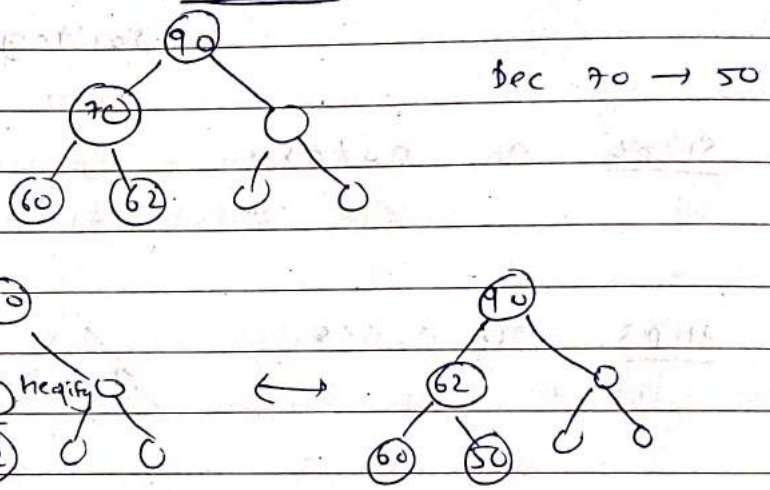
Extract max



O(logn)

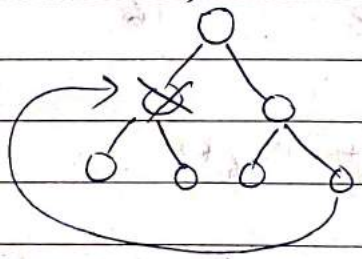
Decrease key

O(logn)



Delete

O(logn)



Then Heapify

Construction of Binary heap

O(n logn)

n * O(insertion)

∴ O(n logn)

How to reduce it ?

PAGE NO.
DATE / /

Start preparing Tree in Bottom up Fashion

$n/8$ nodes

$n/4$ nodes

$n/2$ nodes

Step1 $n/2$ nodes All heaps of size 1 so no sorting

Step2 $n/4$ nodes At max 1 operations $\log_2(2)$

Step3 $n/8$ nodes At max 2 operations. $\log_2(7) = 2$

Total cost

$$T = n/2 * 1 + \frac{n}{8} * 2 + \frac{n}{16} * 3 + \cdots$$

$$T/2 = n/8 * 1 + n/16 * 2 + \cdots$$

$$T - T/2 = T/2 = n/4 + n/8 + n/16 + \cdots$$

$$T/2 = n/2$$

$$T = O(n)$$

PAGE No.
DATE / /

Merge of 2 binary Heaps of size n
Takes linear time using linear time construction of the heap.

(*) Problem

i/p : unsorted array

o/p : top 'k' values

Constraints:

- No extra memory
- Time O(n log k) or less

Soln

Min Heap

put first k elements in min Heap O(k)

n-k elements

For every new element compare root O(1)

if element > root # not in smallest k

else

insert in min Heap → O(log k)

k

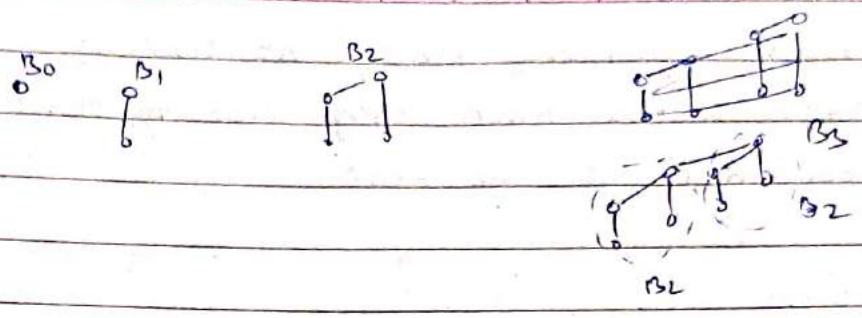
O(log k) + n-k O(log k)

O(n log k)

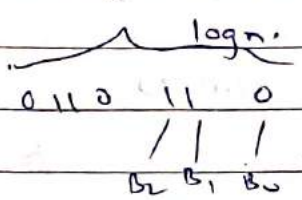
(*) Binomial ~~Heap~~ Tree.

Binomial Tree B_k is an ordered tree defined recursively

- Binomial Tree B_0 consists of one node
- B_k consists of $2 * B_{k-1}$ linked together such that root of one is the leftmost child of the root of the other



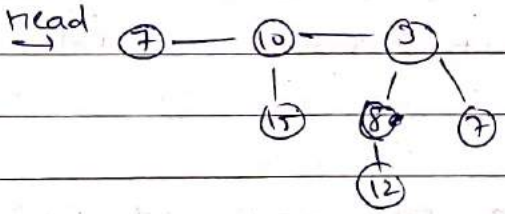
To construct heap of size $\frac{n}{}$



(*) Binomial Heap:

A Binomial Heap H is a set of binomial trees that satisfies the Foll:

1. Each Binomial tree in H obeys min Heap property.
2. For any non-negative integer k, there is at-most one binomial tree in H whose root has degree k.



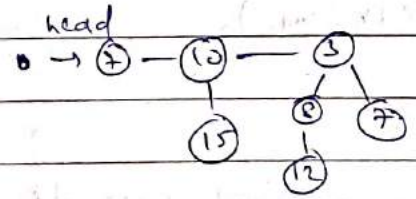
Find min

O(log(n))

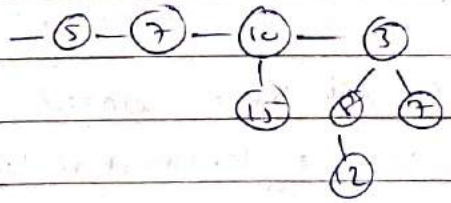
min (7, 10, 3)

3

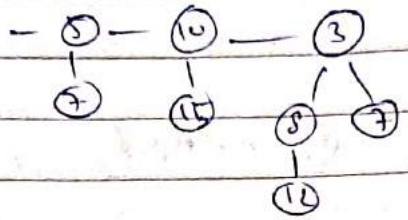
Insert O(logn)

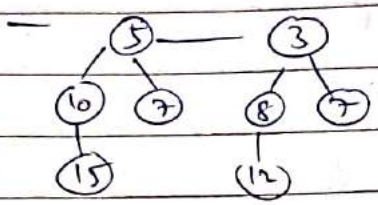


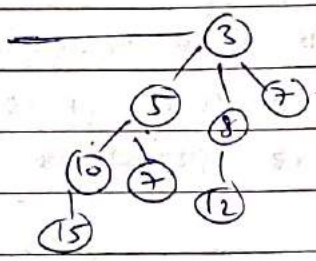
Insert (5)



PAGE No.
DATE / /







(*) Amortized Cost Analysis.

For i=1 to n

push or pop or multipop.

↓ ↓ ↓

O(1) O(1) O(n)

∴ worst case $O(n^2)$

Aggregate Analysis

We show that a sequence of n operations (for all n) takes worst-case Time T(n) Total.

So in worst case, the amortized cost (average cost) per operation will be $T(n)/n$

Multipop → O(n)

seq of n pushes O(n)

multipop O(n)

T(n) = O(n)

∴ amortized cost for n operations $O(n)/n = O(1)$

PAGE NO.
DATE / /

avg cost per operation = O(1)

for i=1 to n

push, pop, multipop

push O(1)
pop O(1)
multipop O(1)

∴ O(n)

(A) Accounting method.

- Assign diff charges to diff operation
- If charge exceeds actual cost, it is stored as credit and used in future operations

Total credit any time = Total amortized cost − Total actual cost

Total credit can never be -ve

push	− 2	O(1)
pop	− 0	O(1)
multipop	− 0	O(1)

op	charge	Actual Cost	Credit
push	2	1	1
push	2	1	2
multipop	0	2	0

for i=1 to n

push, pop, multipop: O(n)
↳ O(1) ↳ O(1) ↳ O(1)

PAGE NO.
DATE / /

(*) Fibonacci Heaps-

Fibonacci Heap is a collection of min-Heap. trees similar to Binomial Heaps. However, trees in a Fibonacci heap are not constrained to be binomial trees. Also, unlike binomial Heaps, trees in Fibonacci Heaps are not ordered.

PAGE NO:
DATE: / /

DAA notes week 4

Shortest path:

Given $G = (V, E)$ with $w(u, v) \geq 0$ for every edge,
Find shortest path from $s \in V$ to $V - s$.

Dijkstra's Algorithm

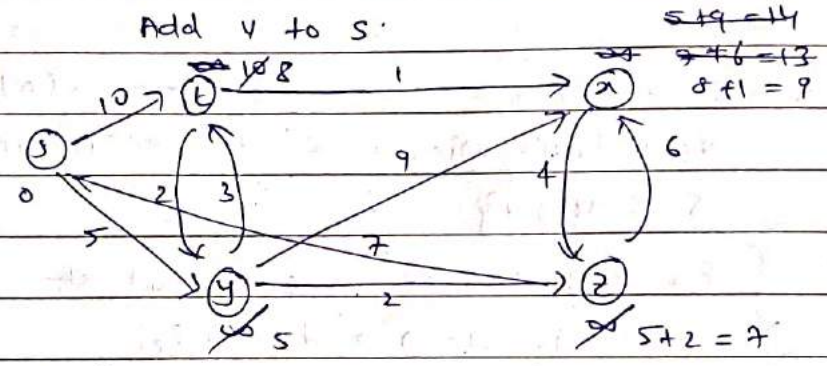
Initially $S = \{s\}$ and $d(s) = 0$

For all other nodes: $d(s) = \infty$

while $S \neq V$

select a node $v \notin S$ with atleast one edge from S for which

$$d(v) = \min(d(u) + l_e)$$



$S = \{s\}$

$S = \{s, y\}$

$S = \{s, y, z\}$

$S = \{s, y, z, t\}$

$S = \{s, y, z, t, x\}$

Pg4 Proof of correctness.

PAGE No.
DATE / /

Dijkstra's Time complexity

Considering there are as many edges as there are nodes.

Binary heap implementation → $o(m \log n)$
Binomial heap implementation → $o(m \log n)$
Fibonacci heap implementation → $o(m + n \log n)$

Dijkstra's Algo:

S = Null
Initialize priority queue Q with all nodes V where d(v) is key value
(All $d(v) = \infty$, except for s $d(s) = 0$) ← $o(n)$

while $S \neq V$ ← $o(n)$
  v = Extract-Min(Q) ← n operations
  $S = S \cup \{v\}$
  For each vertex $u \in Adj(v)$ ← $o(n)$
    if $d(u) > d(v) + l_{ei}$
      Decrease-key (Q, u, $d(v) + l_e$) ← $o(n)$

But here we can only call every edge once.
This gives $o(n^2)$ but if every edge is called only once $o(n^2) \Rightarrow o(m)$

Initialize priority queue: $o(n)$
max no. of Extract-min operations: $o(n)$
max no. of decrease key operations: $o(m)$.

PAGE No.
DATE / /

no cycles

Any Tree that covers all nodes of a graph is called spanning Tree.

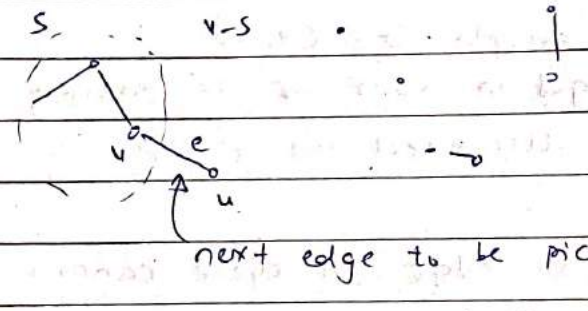
A spanning Tree with minimum total edge cost is a minimum spanning Tree (MST).

Method 1 Kruskals Algorithm.

Sort all edges in increasing order of cost.
Add edges to T as long as no cycle is formed

Fact: Let S be any subset of nodes that is neither empty nor equal to V, and let edge e = (v, w) be the min cost edge with one end in S and other end in V-S. Then every MST contain the edge e.

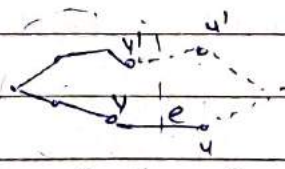
Kruskals proof of correctness:



From the above mentioned Fact
our Algo matches the Fact Hence works perfect everytime.

Hence Kruskals gives MST.

Proof of Fact



There are many edges from S to V-S
'e' is the smallest.

Proof by contradiction

∴ To avoid cycle we would remove v'u' because its cost is more than vu

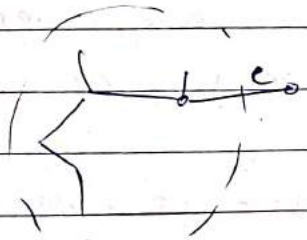
∴ we would take vu instead of v'u'

PAGE No.
DATE / /

Method 2 Prims Algorithm.

Similar to Dijkstra's Algorithm., start with a node S
initially S is only root node.
At each step grow S by one node, adding the
node V that minimizes the attachment cost.

Proof of correctness



Min cost between S and V-S
Fact used again
Hence prims is also
optimal.

Method 3 ~~Backward Delete~~ / Reverse-Delete

Backward version of Kruskals
Start with full graph $G = (V, E)$
Begin deleting edges in order of decreasing cost as long
as it does not disconnect the graph.

Fact: Highest cost edge in cycle cannot belong
to MST.

Proof of correctness

Rev Del is removing edge i.e useless high cost edge
not in MST
Hence it also finds MST.

Prims ~ Dijkstra's ⇒ $O(m \log n)$

Kruskall

	Array implementation	pointer imple
make-set	$O(1)$	$O(1)$
Find-set	$O(1)$	$O(\log n)$
Union	$O(\log n)$	$O(1)$

Find-set: Takes node and returns set it belongs to

Algo

```
A = NULL
For each v ∈ V          } O(n)
    make set(v)         }
sort edges acc to non-decreasing edge cost } O(m log m)
For each edge (u,v) ∈ E in this order → m
    if Find-set(u) ≠ Find-set(v) → O(1)
        A = A ∪ {(u,v)}
        Union(u,v) → log n
```

$O(n) + O(m \log m) + O(m \log n)$

$= O(m \log m)$

Prims vs Kruskals

$O(m \log n)$ $O(m \log m)$

worst case $m = n^2$

$O(m \log n^2) \Rightarrow O(m \log n)$

PAGE No.
DATE / /

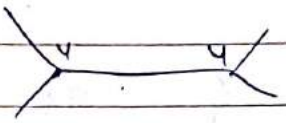
Reverse Delete

Sort edges in dec order ← $O(m \log m)$

For x in edges ← $O(m)$

$O(m+n)$ (if x does not disconnect graph.
remove(x)

$O(m^2)$



To remove edge (u, v)

we run BFS starting at v and see if we reach u.

if we reach u graph is not disconnected and we remove edge (v-u)

(*) Clustering

Given a set of n objects

$p_1 \ldots p_n$

where $d(p_i, p_i) = 0$

$d(p_i, p_j) > 0$

$d(p_i, p_j) = d(p_j, p_i)$

A 'k' clustering of u is partitions of u into k non-empty sets $C_1 \ldots C_k$

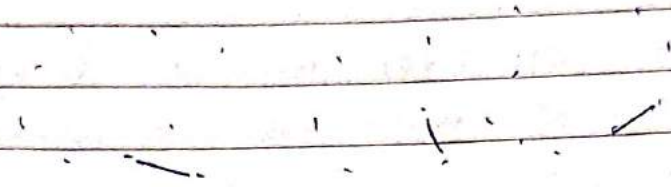
The spacing of a 'k' clustering is the minimum distance between any pair of points lying in different clusters.

PAGE NO.
DATE / /

Problem

Given a set of n objects

Find k-clustering with maximum spacing

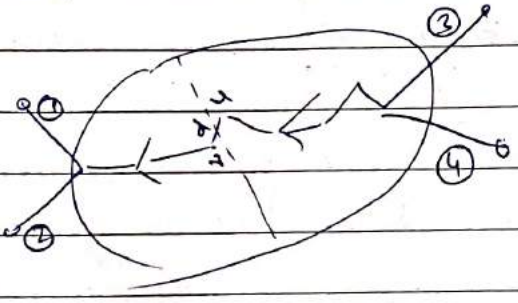


works like prims just leave out last 'k-1' edges.

dist btwn cluster ↑

dist of points in cluster ↓

Proof that there is maximum spacing



lets say cost of edge is 'd'

our spacing is 'd*'

↓ min cost

$d < d^*$

Kruskal drops last k-1 edges ① ② ③ ④

Kruskals choose d over ① ② ③ ④ this means

$d <$ ① ② ③ ④

d^*

PAGE NO.
DATE / /

ADA, week 5 Notes

Divide and conquer:

Divide into subproblems

Conquer i.e solve the subproblems recursively, or if trivial solve the problem itself.

Combine the solution to the subproblems.

Eg: merge-sort (A, p, r)

if p < r then

q = ⌊(p+r)/2⌋ → Divide O(1)

merge_sort (A, p, q) } ⇒ Conquer 2T(n/2)

merge_sort (A, q+1, r) }

merge (A, p, q, r) } ⇒ combine Θ(n).

$$T(n) = \begin{cases} 1 & \text{if } n == 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & \end{cases}$$

$2T(n/2)$ ← Conquer; $\Theta(n)$ ← merge; $\Theta(1)$ ← Divide

General recurrence equation:

$$T(n) = \begin{cases} \Theta(1) & n == 1 \\ aT(n/b) + D(n) + C(n) & \end{cases}$$

a ← No. of subproblems; n/b ← Size of subproblem; $D(n)$ ← Time to divide; $C(n)$ ← Time to conquer.

PAGE NO.
DATE / /

Master Method

$$T(n) = aT(n/b) + f(n)$$

$a \geq 1$ $b \geq 1$ are constants

1) If $F(n) = \Theta(n^{\log_b a - \varepsilon})$ For some $\varepsilon > 0$

$$T(n) = \Theta(n^{\log_b a})$$

2) If $f(n) = \Theta(n^{\log_b a})$ then

$$T(n) = \Theta(n^{\log_b a} \log n)$$

3) If $F(n) = \Omega(n^{\log_b a + \varepsilon})$ For some $\varepsilon > 0$

and if $af(n/b) \leq c \cdot f(n)$ For some constant $c < 1$

$$T(n) = \Theta(f(n))$$

Generalization

$$f(n) = \Theta(n^{\log_b a} \log^k n)$$

where $k \geq 0$.

Then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

PAGE No.
DATE / /

Eg: Merge sort

$$T(n) = 2T(n/2) + \theta(1) + \theta(n)$$

$$= 2T(n/2) + \theta(n)$$

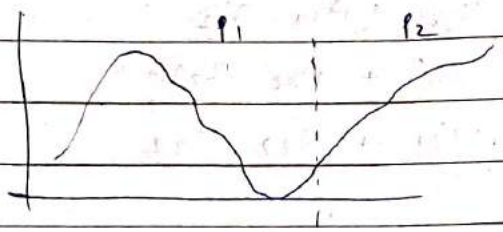
$$n^{\log_b a} = n^{\log_2 2} = n^1$$

$$f(n) = n$$

∴ Case # 2

$$\theta(n \log n)$$

Eg: Stock market problem.



Case 1 : Sell & Buy in P_1

Case 2 : Sell & Buy in P_2

Case 3 : Sell in P_2 , Buy in P_1

Case 3

$M = \text{Min}(M_1, M_2)$

$x = \text{Max}(x_1, x_2)$

$B = M_1$

$S = x_2$

$f(n) = C(n) + D(n)$

$= \theta(1) + \theta(1)$

$= \theta(1)$

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

Case #1 $T = \theta(n)$

PAGE No.
DATE / /

eg: Dense Matrix multiplication

$n\{[A]\ [B]\}_n = [C]\}_n$

($[A]$: n rows; $[B]$: n columns; $[C]$: n)

Brute Force $\Rightarrow \Theta(n^3)$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$ → subproblems

$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$

$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$

$C_{22} = A_{21} \cdot B_{21} + A_{22} \cdot B_{22}$

→ conQuer.

$f(n) = D(n) + C(n)$

$= O(1) + \Theta(n^2) = \Theta(n^2)$

$n^{\log_b a} = n^{\log_2 8} = n^3$

case #1 $\Theta(n^3)$

suppose initially $n=4$.

$\begin{bmatrix} 2\times2 & 2\times2 \\ 2\times2 & 2\times2 \end{bmatrix}$

$\therefore$ 2×2 matrix

8 multiplication

$\therefore$ Subproblems = 8

Size reduced by 2

PAGE NO.
DATE / /

Strassen's Matrix Multiplication

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

$f(n) = \Theta(1) + \Theta(n^2) = \Theta(n^2)$

$n^{\log_b 9} = n^{\log_2 7} = n^{2.81}$

Case #1 $\Theta(n^{2.81})$

pg 17 (*) Finding Min Max in unsorted array

pg 18 (*) closest pair of points problem (2D)

PAGE No.
DATE / /

ADA Week 7 Notes

Approach to DP

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a Bottom up Fashion.
4. Construct an optimal solution from computed information.

(*) Weighted interval scheduling problem.

- we have 1 resource
- we have n requests labeled 1...N
- Each req has start Time s_i
  Finish Time f_i
  weight w_i
- Goal: Subset $S \subseteq \{1..N\}$ of mutually compatible intervals so as to maximize $\sum_{i \in S} w_i$

There are 2 CASES either a req is in the soln set or not in the soln set.

CASE 1: req i belongs to soln set

OPT SOLN = w_i + OPT SOLN (All req compatible with i)

CASE 2: req i not in soln set

OPT SOLN = OPT SOLN (All req with job i)

PAGE No.
DATE / /

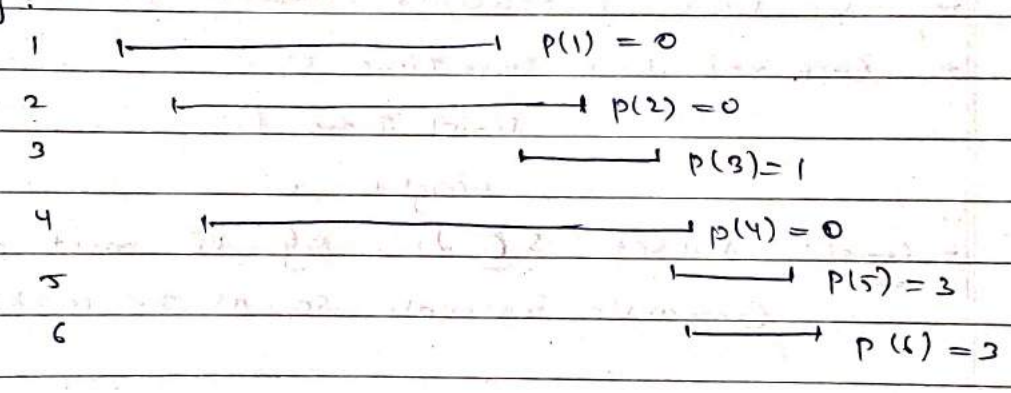
SOLUTION:

Sort req in order of non-decreasing Finish Time
$f_1 \leq f_2 \leq \ldots \leq f_n$.

P(j) → For an interval j to be largest index i<j such that interval i & j are disjoint.

i.e i is the leftmost interval that ends before j begins.

Eg:



O(j) → OPT SOLN TO problem consisting of requests {1..j}

let OPT(j) denote value of Oj

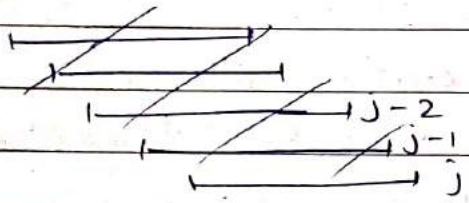
$O(3) = \{1, 3\}$ $OPT(3) = 6$

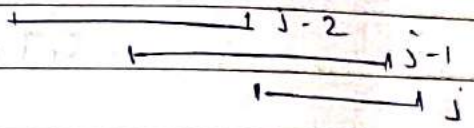
$$OPT(j) = \max \begin{cases} w_j + OPT(P(j)) & \rightarrow j \in O_j \\ OPT(j-1) & \rightarrow j \notin O_j \end{cases}$$

PAGE No.
DATE / /

```
Compute_opt(j)
    if j==0  ret 0
    else
        return max( wj + compute-opt(P(j)) , Compute-opt(j-1) )
```

Worst case





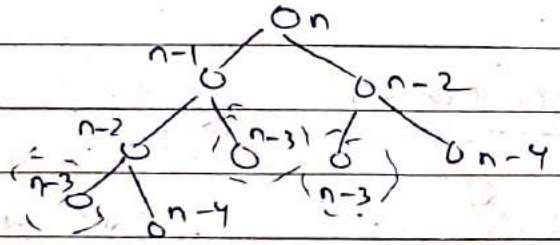
$$T(n) = T(n-1) + T(n-2)$$

$j \notin o_j$ — we reduce j by 1

$j \in o_j$ — $P(j) = j-2$

Worst case if each req only overlaps with one req above it and one req below it.

exponential time complexity.



There are various subproblems that repeat in the tree. And we are calculating them again and again.

Instead we should calculate it once and store, and just retrieve when called again

MEMOIZATION

PAGE No.
DATE / /

```
Compute-opt(j):
    if j == 0  return 0

    elif m[j] is not empty
        return m[j]

    else
        m[j] = max( compute_opt(j-1),
                    wj + compute_opt(p(j)) )
    return m[j]
```

O(n) As every subproblem called only once and it is reused when called again.

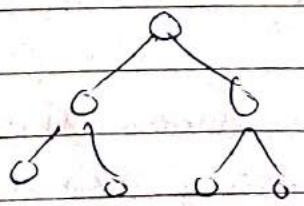
Initial sorting O(n log n)

making p[] O(n log n) USING BINARY SEARCH TO FIND THE p[] For n elements

n * log n
↑ element, ↑ each binary search count

Compute-opt O(n)

O(n log n)

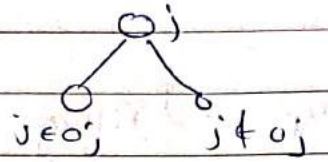


We do Bottom up to Find optimal value

Then do Top Down to Find the optimal path that was taken.

PAGE No.
DATE / /

Compute an optimal soln.



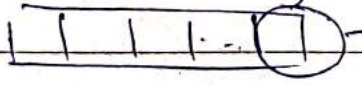
$w_j + OPT(P(j)) \geq OPT(j-1)$

j is in optimal solution if and only if

```
Find_solution.
    if j>0 then
        if w_j + M[P(j)] ≥ M[j-1] then.
            output j together with the results
            of Find_solution (P(j))
        else
            output the results of
                Find_solution (j-1)
        endif
    endif
```

O(n) worst case when subproblem size does not reduce much.

memoized array



what if we are able to build memorized array without recursion.

Try to build memoized array using iteration.

PAGE No.
DATE / /

Top Down {finding optimal path taken}

Bottom up {filling memo}

For i = 1 to n:

$$m[i] = \max(m[i-1] + w_i + m[p(i)])$$

Divide & Conquer	DYNAMIC
• independent subproblems	• dependent subproblems {reduce overlapping subproblems}
• we find optimal soln for each subproblem & combine	• we find optimal value then from that optimal solution.

(*) Video Game Problem:

GOAL: get home at level 'n'

G_0 ← initial energy.

we loose e_i energy when we land on stage i

choices

		Cost
1.	walk into stage	50 units
2.	Jump over a stage	150 units
3.	Jump over 2 stages	350 units

Reach home with maximum energy left.

PAGE No.
DATE / /

We have 'n' unique subproblems.

Stage $0 \to 1$

$0 \to 2$

$0 \to 3$

.

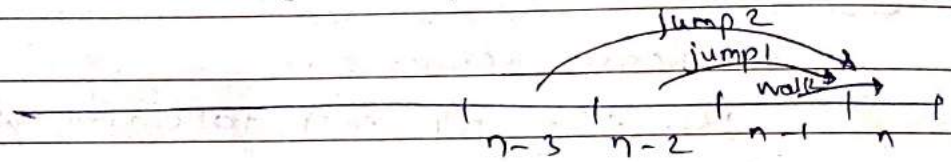
.

$0 \to n$.

$OPT(n)$ = OPT level of energy when we reach stage n.

Formula (1)

$$OPT(n) = MAX \begin{pmatrix} OPT(n-1) - 50 - e_n, \\ OPT(n-2) - 150 - e_n, \\ OPT(n-3) - 350 - e_n \end{pmatrix}$$



$OPT(0) = E_0$

$OPT(1) = E_0 - 50 - e_1$

$OPT(2) = MAX (\cdot \cdot \cdot)$

For $i = 3$ to n :

Formula (1)

end For.

(*) Austrian coin determination.

Goal : gives less number of coins.

1, 5, 10, 20, 25

$OPT(n)$ = min. no. of coins to pay for n schillings.

Formula (2)

$$OPT(n) = MIN \begin{pmatrix} 1 + OPT(n-1), \\ 1 + OPT(n-5), \\ 1 + OPT(n-10), \\ 1 + OPT(n-20), \\ 1 + OPT(n-25) \end{pmatrix}$$

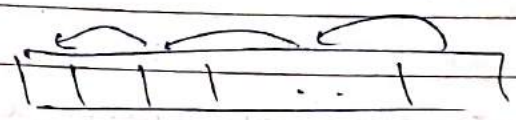
PAGE No.
DATE / /

Initialize OPT[1 . . 25]

For i = 25 to n:

Formula ②

End For.



Find OPT soln then Top Down to find number of each items.

④ 0-1 knapsack & subset sum.

→ single resource.

- Req {1 . . n} each take time w_i to process.
- can schedule jobs at any time between 0 to W

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

OPT(i, w) = value of the OPT solution using a subset of the items {1 . . i} with max allowed weight W

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

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

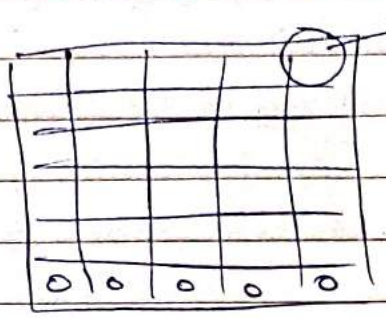
If $w < w_i$:

$$OPT(i, w) = OPT(i-1, w)$$

Formula ③

Else:

$$OPT(i, w) = MAX\left(\begin{matrix} OPT(i-1, w) \\ , \\ w_i + OPT(i-1, w - w_i) \end{matrix}\right)$$



For i = 1 to N
O(n)
For w = 0 to W
O(W)
use Rec formula (1)
O(1)

Pseudo Polynomial

$O(n \cdot W)$

number of req's

Int weight

Pseudo-Polynomial Time

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

Polynomial Time

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

PAGE No.
DATE / /

AOA week 8 Notes

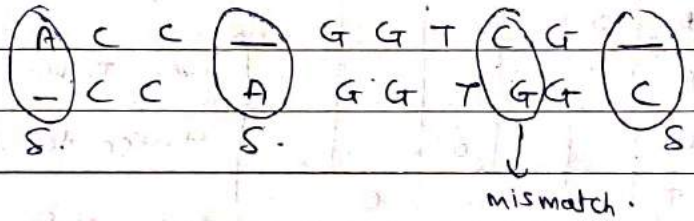
(*) Sequence Alignment Problem:

DNA Strand consists of 4 Bases {A, C, G, T}

Comparing DNA Strands

S_1 = A C C G G T C G

S_2 = C C A G G T G G C

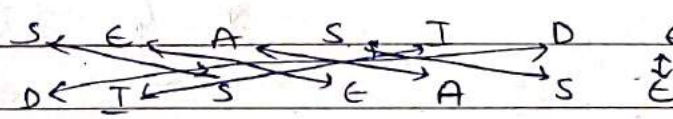


3 gaps 1 mismatch.

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

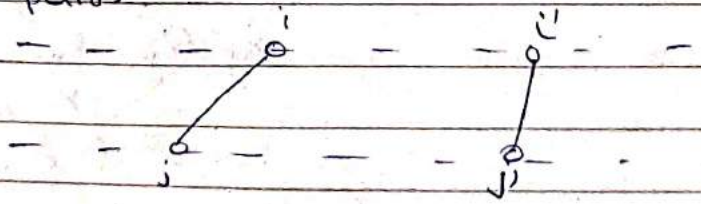
$Y = \{y_1, y_2 \ldots y_n\}$

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



perfect matching But they are not similar.

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



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

if $i < i' \Rightarrow j < j'$

PAGE No.
DATE / /

For an alignment M between X and Y.

1. we incur a "gap penalty" of δ (Fixed) for each gap

2. For each mismatch (of letters $p \neq q$) we incur a mismatch cost α_{pq} (not Fixed.)

	A	C	G	T
A	0			
C		0		
G			0	
T				0

α Table given to us.

$\alpha_{xx} = 0$

$\alpha_{xy} \Rightarrow$ +ve int.

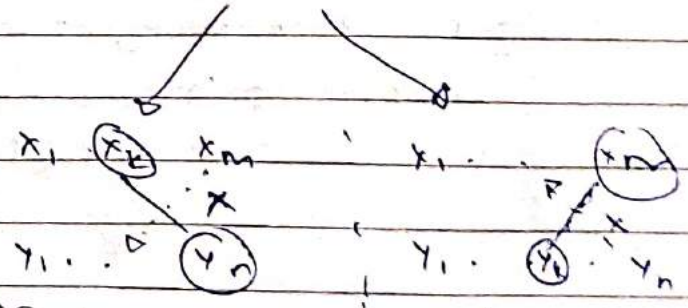
Defn Similarity between strings X and Y is the minimum cost of an alignment between X and Y.

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

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

Say M is OPT soln.

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



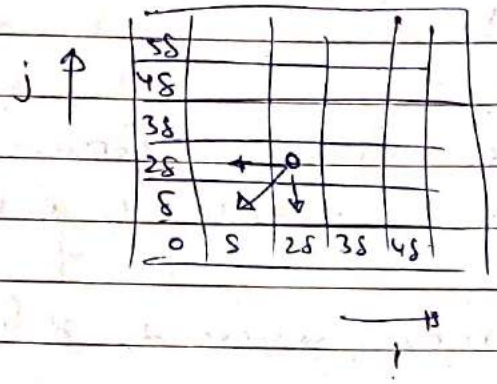
if (x_k, y_n) exist so x_m will not be paired as it will result in cross pair

if (x_m, y_k) exist so y_n will not be paired as it will result in cross pair.

PAGE NO.
DATE / /

Rec Formula ①

$$OPT(m,n) = \min \begin{pmatrix} OPT(m-1, n-1) + \alpha_{x_m y_n} \\ OPT(m-1, n) + \delta \\ OPT(m, n-1) + \delta \end{pmatrix}$$

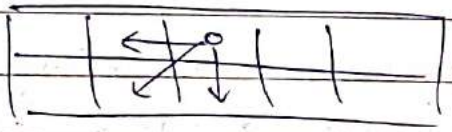


Alignment (X, Y)
Init $A[i, 0] = i\delta$ for each i
Init $A[0, j] = j\delta$ for each j

For j=1 to n
  For i=1 to m
    Rec Formula ①

Time : $O(m * n)$
Space : $O(m * n)$

Suppose DNA seq ≈ 3B size
Space Req : 3B * 3B

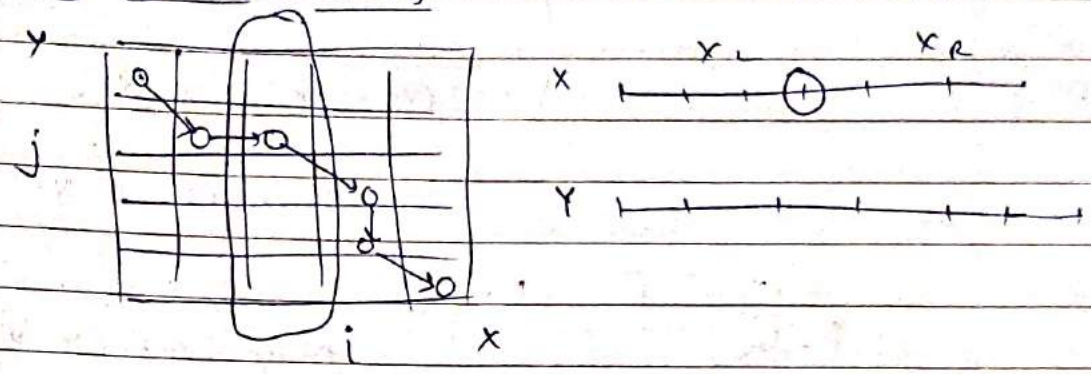


But for calculation of a particular row we only need 1 prev row.

But if we keep 2 rows to solve.
we will get the optimal ans but how will we find opt SOLN as we will not be able to backtrack.

PAGE No.
DATE / /

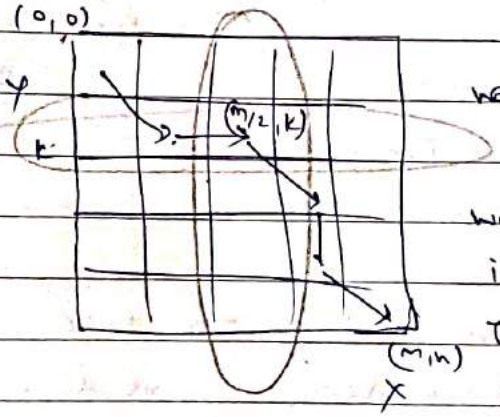
Memory Efficient way:



We will use divide & conquer

We will divide X in middle to get X_L and X_R

But how do we know where to divide Y



We have x in middle.

We now need to find index 'k' such that the optimal soln passes through that point

∴ OPT path passes through index k

where $x = m/2$

i.e $\left(\overset{q}{\left(m/2\right)}, k\right)$

let $f(q,k)$ $\quad$ $g(q,k)$

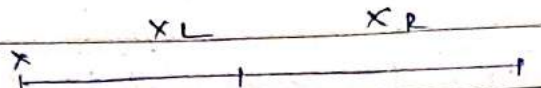
↓ $\quad$ ↓

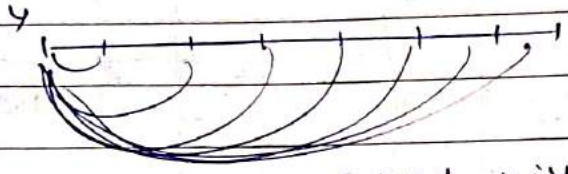
OPT $((0,0) \rightarrow (q,k))$ $\quad$ OPT $((q,k) \rightarrow (m,n))$

∴ k is the index which minimizes $\left(f(q,k) + g(q,k)\right)$

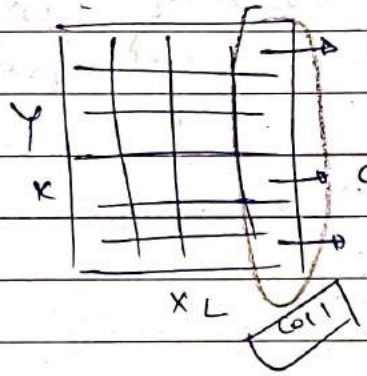
PAGE No.
DATE / /

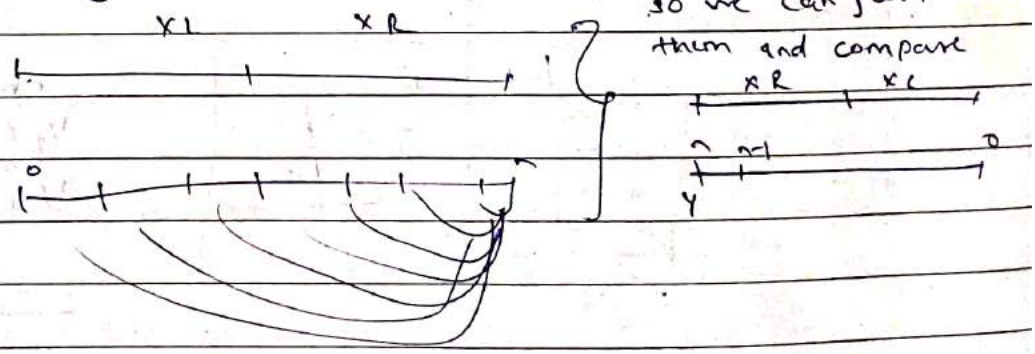
eg :



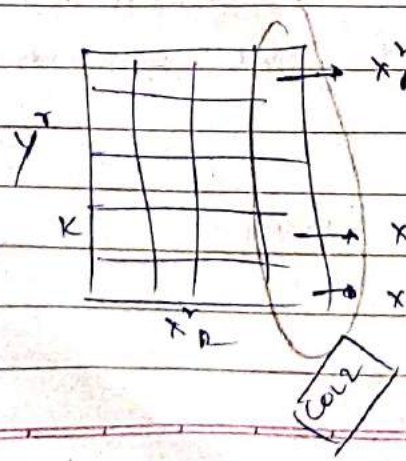


- x_L could be solved with y_0
  x_L " y_1
  x_L " y_n



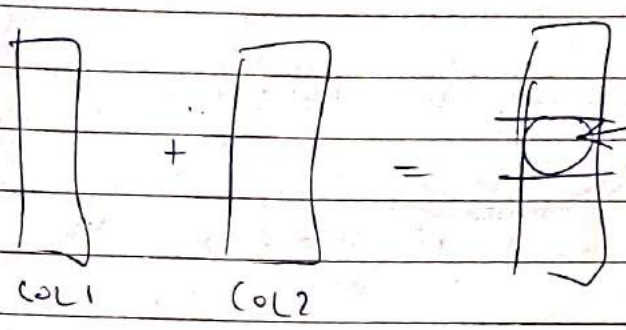


x_R could be aligned with y_n
" $y_{n-1..n}$
"
$y_{k..n}$.

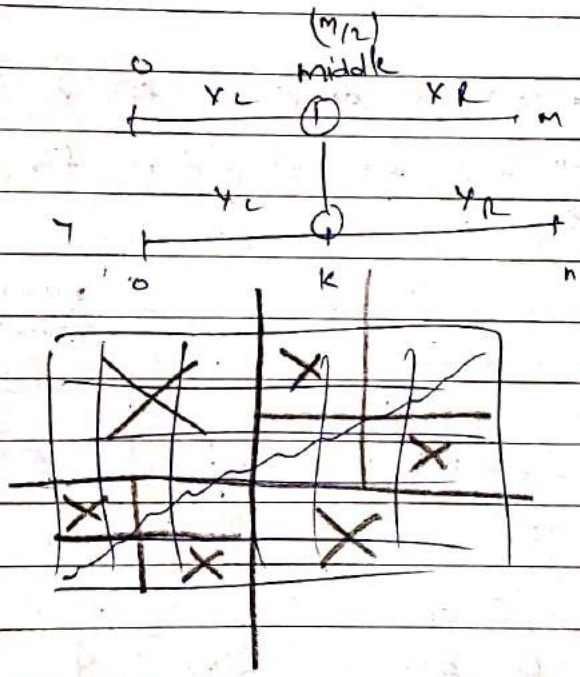


PAGE No.
DATE / /

after getting [col1] and [col2]



Find the index start and split 4 on that index give min ans



cmn
1/2 cmn
1/4 cmn
.
.
.
2 cmn

Time : o(2cmn)

As we are just intersted in final 2 columns of x_L y and x_R y we use memory efficient way

Space o(m+n) linear.

PAGE No.
DATE / /

memory efficient way

Divide ⇐ use Dynamic programming to find optimal split point for y

Solve

Conquer ⇐ just concatenate the ans of subproblems.

	memory wasteful way	memory efficient way
Time	$\Theta(mn)$	$O(2mn)$
Space	$O(mn)$	$O(m+n)$

In D&C we solve for OPT SOLN so we directly get OPT SOLN after concat

PAGE No
DATE / /

⑧ matrix chain Multiplication

$A = A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_n$

Eg

$B \cdot C \cdot D$

$B = 2 * 10$

$C = 10 * 50$

$D = 50 * 20$

$B \cdot C \cdot D = (B \cdot C) \cdot D \Rightarrow 3000$ operations

or

$B \cdot (C \cdot D) \Rightarrow 10400$ operations.

$$\left((A_i \ldots A_k)(A_{k+1} \cdot \cdot A_j) \right)$$

Finally $\left((1 \text{ matrix})(1 \text{ matrix}) \right)$

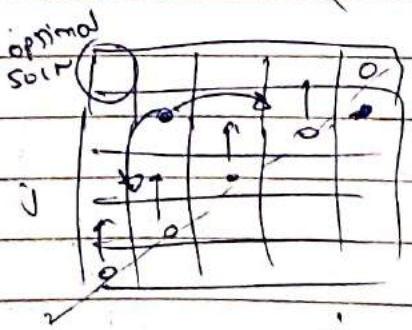
we need to find where to split

$OPT(i,j)$ = optimal cost of multiplying matrices i through j

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

Formula

final product cost



PAGE NO.
DATE / /

For i=1 to n OPT(i,i) = 0

~~For j=1 to~~

For j=2 to n
  For i = j-1 to 1 dec by -1
    Rec formula (2) → O(n)
  end for
end for

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

$O(n^3)$ Time

Efficient Soln.

(*) Shortest path problem Dynamic Programming.

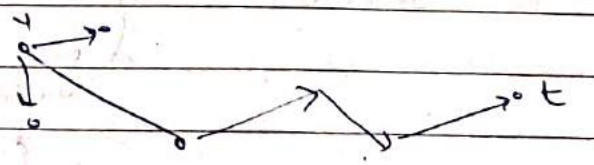
Assumption: No -ve cost cycles

There can be -ve cost edges.

If G has no -ve cycles, then there is a shortest path from s to t that is simple and hence atmost 'n-1' edges.

OPT(i, v) ⇒ min cost from v - t with atmost i edges

Problem statement: OPT(n-1, s)

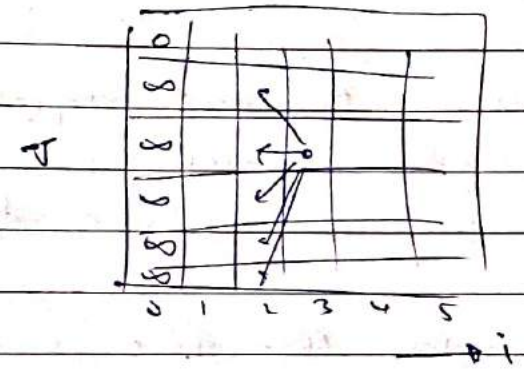


$$OPT(i,v) = \min_{\text{For } w \text{ in adj}(v)} \{ (c_{vw} + OPT(i-1, w)) \}$$

PAGE NO.
DATE / /

But by the recurrence relation we are only considering paths of exactly (or 'i')
Path are to be atmost 'i'
So they can also be less than 'i'

$$OPT(i,v) = \min \begin{pmatrix} OPT(i-1,v) \\ \min_{\substack{\text{for} \\ w \in Adj(v)}} \{ c_{vw} + OPT(i-1,w) \} \end{pmatrix}$$



Bellman Ford Algo.

```
Shortest-Path (G, s, t)
    n = no. of nodes in G
    define M[0, t] = 0      M[0, v] = ∞
    For i = 1 to n-1
        For v ∈ V in any order
            M[i, v] = min( M[i-1, v],
                           min_{w ∈ Adj(v)} { c_vw + OPT(i-1, w) } )
        End For
    End for
```

O(n) O(n) O(n) O(n²) O(mn)

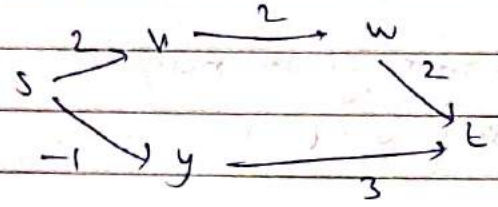
But every edge can be called only once (SAME AS THAT OF DIJKSTRA)

PAGE NO.
DATE / /

$O(m \cdot n)$

at max $m = n^2$

eg:



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

→ 2 col's same.

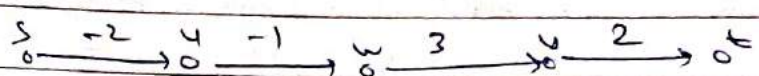
STOP HERE

1 COLUMN (Search t with almost 1 edge)		2 COLUMN (search t with almost 2 edge)	
t-t	0	t-t	0
y-t	3	y-t	3
w-t	2	w-t	2
v-t	∞	v-t	4
s-t	∞	s-t	2

At any time only last 2 col's are in consideration i.e col3 can be judged from col2 only.

PAGE No.
DATE / /

Trying to solve with only 1 column.



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

→ w looks at its neighbours ie y sets v - t = 2

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

filling ↑

$O(n-1)$ worst case

What is preference of filling?

BFS order from T

As its neighbours will convey info

then their neighbours

:

so on.

We found shortest dist

For shortest path we just keep a track of present

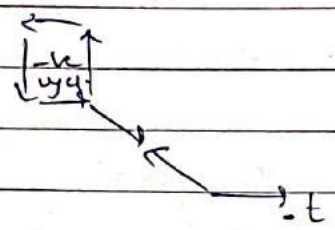
i.e neighbour that give shortest distance to t.

PAGE No.
DATE / /

∴ shortest path always found in 'n-1' iterations

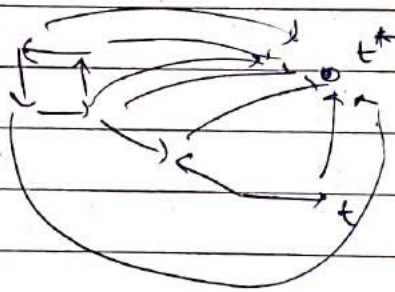
if after n-1 iterations value go down this indicates presence of -ve cycle.

Q: How to FIND -ve CYCLE



In this running it after n-1 times wont change column value as -ve cycle is not on path to t

To FIND CYCLE



Add new node t^* connect every node to t^* run Bellmanford. For t^*

Bellman Ford
$O(mn)$

Dijkstras
$O(m \log n)$

If -ve cost edges can only use Bellman-Ford

If no -ve cost edges
Can use Bellmanford & Dijkstras.

Some cases Bellmand ford faster than Dijkstras as it can be easily parallelized.

PAGE No.
DATE / /

AOA week 9 Notes

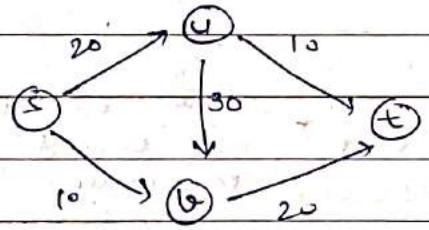
Network Flow:

Many types of problems can be solved from network flow problem.

Flow network is a directed graph $G = (V, E)$ with following features:

- Associated with each edge 'e' is a capacity, which is a nonnegative number that we denote c_e
- There is a single source $s \in V$
- There is a single sink $t \in V$

Nodes other than s and t are internal nodes.



Flow: s-t flow is a function f that maps each edge e to a nonnegative real number

$$f: E \rightarrow R^+$$

f(e): represents the ~~value~~ amount of flow carried by edge e.

f satisfies two properties.

(i) Capacity condition

For each $e \in E$, we have $0 \leq f(e) \leq c_e$

(ii) conservation condition.

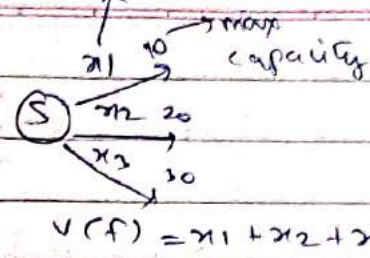
For each node v other than s and t, we have

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

PAGE No.
DATE Carrying Flow.

value of Flow $v(f)$

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



$v(f) = x_1 + x_2 + x_3$.

For making eqn's compact.

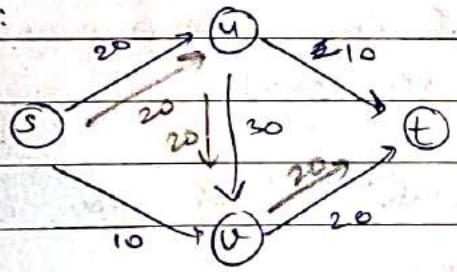
$$f^{out}(v) = \sum_{e \text{ out of } v} f(e)$$

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

Given a flow ntwk, find a flow of maximum possible value.

Maximum flow minimum cut — Cut of the graph puts a bound on maximum flow value.

Eg:



Flow = 20

we cannot pass anymore flow

But by looking at the figure we can say max flow that can be achieved is 30.

So there must be a way to undo some flow in case if we get stuck to move forward in the algorithm.

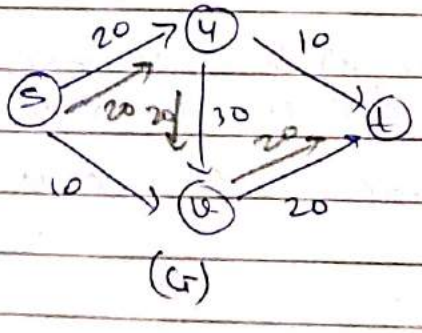
So we undo 10 units of flow on (v, u)
finally pushing 10 units from (v, t)
Hence increasing the flow to 30.

* we can push Forward on the edges with leftover capacity
we can push backward on the edges that are already carrying flow, to divert it in a different direction.

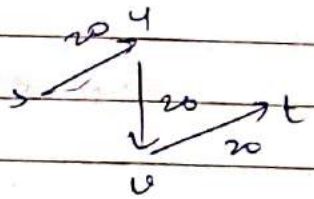
* Residual Graph.
Has same no. of nodes as that of G
For a given flow f on G

edge $e = (u, v)$ of G on which $f(e) < c_e$ there are $c_e - f(e)$ "leftover" units which we could try pushing forward.

There are $f(e)$ units of flow that can be undone, by pushing flow backward. So we include $e' = (v, u)$ in G_f with capacity $f(e)$

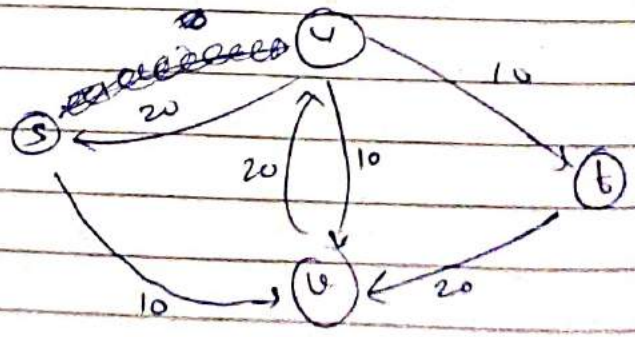


(G)



(f)

(G_f)



PAGE NO.
DATE / /

Thus G_f can have atmost twice as many edges as G.

Augmenting paths in a Residual Graph.

let P be a simple s-t path.

↳ no cycles
• visits any node only once

bottleneck(P, f) = minimum residual capacity of any edge of P, with respect to flow f.

augment (f, P) → yield new flow f' in G

```
augment(f, P)
    b = bottleneck(P, f)
    For each edge (u, v) ∈ P:
        if e = (u, v) is Forward edge:
            increase f(e) in G by b.
        else (u, v) is backward edge,
            and let e = (v, u):
            decrease f(e) in G by b.
    return (f).
```

PAGE No.
DATE / /

TRUE

(7.1) IF F is legal flow F' is also legal flow.?

F' should follow 2 conditions

(i) Capacity condition.

If $e = (u, v)$ is fwd edge with residual capacity $(e - f(e))$

$$0 \le f(e) \le f'(e)$$

$$f'(e) = f(e) + \text{bottleneck}(p, f) \quad \text{known.}$$

$$0 \le f(e) \le f(e) + \text{bottleneck}(p, f)$$

$$\text{bottleneck}(p, f) = c(e) - f(e)$$

$$0 \le f(e) \le f(e) + c(e) - f(e)$$

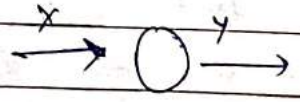
$$0 \le f(e) \le c(e) \quad \text{always True}$$

same can be proved for backward edge.

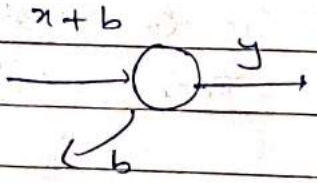
(ii) Conservation condition.

If f followed conservation condition

then in f' we increase fwd edge by b and dec back edge by b



How F
if $x = y$ (given)



F^1

$x + b = y + b$

$x = y$ (given)

Hence F^1 is also conserved.

PAGE NO.
DATE / /

(*) FORD FULKERSON ALGO

Max Flow.

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

While there is an s-t path in residual graph G_f:

Let P be simple s-t path in G_f

$f' =$ augmentation (f, P)

update f to f'

Updat residual graph G_f to $G_{f'}$

EndWhile

return f

(7.2) At every intermediate stage of the Ford Fulkerson algo, the Flow values $\{f(e)\}$ and residual capacities in G_f are integers.

Obvious

Starting with int

$b \Rightarrow$ int

int $\pm$ b $\Rightarrow$ int.

(7.3) Let f be a Flow in G, and let P be a simple s-t path in G_f. Then $v(f') = v(f) + \text{bottleneck}(P, f)$

But bottleneck $(P, f) > 0$

$\therefore$ $\boxed{v(f') > v(f)}$

PAGE No.
DATE / /

(7.4) Suppose, that all capacities in the flow ntwk G are integers. Then the Ford Fulkerson algo terminates in atmost C iterations of the while loop.

$$C = \sum_{e \text{ out of } s} c_e$$

TRUE
G_f cannot have value greater than C.

initially Flow = 0
in worst case Ford Fulkerson will increase flow by 1

Hence atmost C iterations

(7.5) $O(m+n) \Rightarrow O(m)$
Since all the nodes have atleast 1 incident edge.

Suppose from above all capacities in the flow network G are integers.
Ford Fulkerson runtime $O(C \cdot m)$

↳ pseudo polynomial

C

↑ while there is a s-t path in residual graph.

path found using BFS or DFS $O(m+n) = O(m)$ ← Let P be simple s-t path in G_f — path P will have atmost n-1 edges

f = augmentation (f, P) → O(n)
update f to f'
update G_f to $G_{f'}$ → O(m)

PAGE No.
DATE / /

Now we have to show Ford Fullerson returned max flow.

From (7.4) we know

Max-flow $\leq$ C

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

Sometimes this upperbound is very weak.

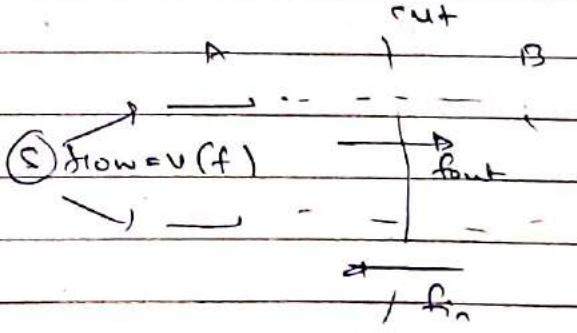
(*) A cut (A, B) has capacity = $c(A, B)$

$$c(A, B) = \sum_{e \text{ out of } A} c_e$$

Cut divides graph into two partitions A and B.

(7.6) Let f be any s-t flow, and (A, B) be any s-t cut.

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



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

PAGE No.
DATE / /

(7.7) Let f be any s-t flow, and (A, B) any s-t cut. Then $v(f) = f^{in}(B) - f^{out}(B)$

(7.8) Let f be any s-t flow, and (A, B) be st cut.

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

$$c(A, B) = f^{out}(A)$$

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

$$v(f) \leq f^{out}(A)$$

$$\boxed{v(f) \leq c(A, B)}$$

value of every flow is upper-bounded by capacity of every cut.

(*) Let $\bar{f}$ be flow returned by Ford Fulkerson. We want $\bar{f}$ to have max possible value

We exhibit a cut (A^*, B^*)

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

This immediately establishes that $\bar{f}$ has the maximum value of any flow, and that (A^*, B^*) has the minimum capacity of any s-t cut.

PAGE No.
DATE / /

(7.9) If f is ~~any~~ an st flow such that there is no s-t path in the residual graph G_f, then there is an s-t cut (A^*, B^*) in G for which $v(f) = c(A^*, B^*)$.
Consequently f has the maximum value of any flow in G, and (A^*, B^*) has minimum capacity of any s-t cut in G.

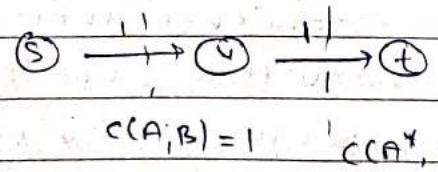
(7.10) Flow returned by Ford Fulkerson is max flow.

(7.11) Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.

Run BFS/DFS from s to find nodes reachable from s
they belong to A^*
$B^* = V - A^*$

→ $O(m+n) \Rightarrow O(m)$

Note: There may be many min capacity cuts



$c(A, B) = 1$ $c(A', B') = 1$

Our method returns the cut closest to S.

PAGE No.
DATE / /

This must be max flow

(7.12) In every flow network, there is a flow f and a cut (A, B) so that

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

(7.13) In every flow network, the maximum value of an s-t flow is equal to minimum capacity of an s-t cut.

(7.14) If all capacities in flow ntwk are integers, there there is a maximum flow f for which every flow value f(e) is an integer.

In every iteration

$$v(f') = v(f) + \text{bottleneck}(p, f)$$

As all numbers are int → ≥ 1

∴ while loop terminates in atmost C iteration.

But consider values in decimal.

$$v(f') = v(f) + \text{bottleneck}(p, f)$$

→ 0.000001... 1

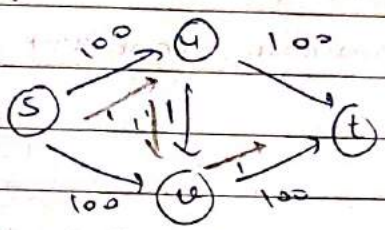
This loop wont end in finite time

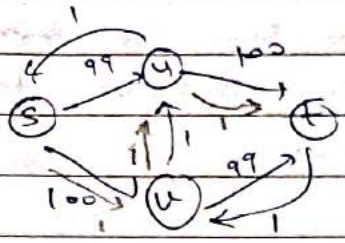
PAGE No.
DATE / /

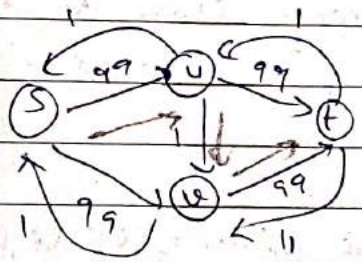
(A) Choosing good augmenting paths

As seen we just choose any path and augment it.

eg:







As seen if we keep picking this path it may take us 200 iterations

But we can do it in less number of iterations.

Idea: Recall that augmentation increase the value of the maximum flow by the bottleneck capacity of the selected path; so if we choose paths with large bottleneck capacity, we will be making a lot of progress.

Having to find such paths can slow down each individual iteration by quite a bit. we will also avoid this slowdown by not worrying about selecting the path that has exactly the largest bottleneck value.

PAGE No
DATE / /

Instead, we will maintain a so called scaling parameter Δ, and we will look for paths that have bottleneck capacity of atleast Δ.

Let $G_f(\Delta)$ be the subset of the residual graph consisting only of edges with residual capacity of atleast Δ.

We will work with values of Δ that are powers of 2.

Scaling Max Flow $O(m^2 \log_2 C)$

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

Initially set Δ to be the largest power of 2 that is no larger than maximum capacity out of s

$(\Delta \leq \max_{e \text{ out of } s} c_e)$

$\log_2 C$ while $\Delta \geq 1$ Ford Fulkerson

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

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

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

update f to be f' and update $G_f(\Delta)$

End while

$\Delta = \Delta/2$

end while

return f

(7.15) If the capacities are integer valued, then throughout the scaling max-flow algorithm the flow and the residual capacities remain integer-valued. This implies that when $\Delta = 1$ $G_f(\Delta)$ is the same as G_f, and hence when the algorithm terminates the flow f, is of maximum value.

PAGE No.
DATE / /

(7.16)

∴ Δ SHOULD BE LESS THAN 70 OR EQUAL TO

Δ = 64

The number of iterations of the outer while loop is atmost $1 + \log_2 C$

(7.17) During the Δ-scaling phase, each augmentation increases the flow by atleast Δ.

as all edges have capacity ≥ Δ

∴ BOTTLENECK ≥ Δ

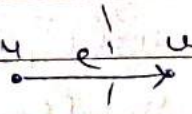
new flow = old flow + BOTTLENECK.

(7.18) Let f be the flow at the end of the Δ-scaling phase. There is an st cut (A, B) in G_f for which $C(A,B) \leq v(f) + m\Delta$, where m is the number of edges in the graph. Consequently, the maximum flow in the network has value atmost $v(f) + m\Delta$

Proof

let there be A → nodes reachable from s

B → V − A.

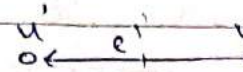


$c_e < f(e) + \Delta$

or else v would also have been reachable and be in set A.

PAGE No.
DATE / /

B A



IF $f(e') \geq \Delta$ then u' will belong to A

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in } A} f(e)$$

$$\geq \sum_{e \text{ out of } A} (c_e - \Delta) - \sum_{e \text{ in } A} \Delta$$

$$= \sum_{e \text{ out of } A} c_e - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ into } A} \Delta$$

$$\boxed{v(f) \geq c(A,B) - m\Delta}$$

(7.19) The number of augmentations in a scaling phase is atmost $2m$

new flow $\leq$ old flow + m(old delta)

old delta $= 2$ new delta

new flow $\leq$ old flow + $2m$ (new delta)

$2m$ → atmost $2m$ augmentations in scaling phase

PAGE No.
DATE / /

(7.20) The Scaling max-flow Algorithm in a graph with m edges and integer capacities finds a maximum flow in at most ~~$O(m^2 \log_2 ( )$~~ $2m(1 + \log_2 ( )$ augmentations.
It can be implemented to run in at most $O(m^2 \log_2 ( )$ time.

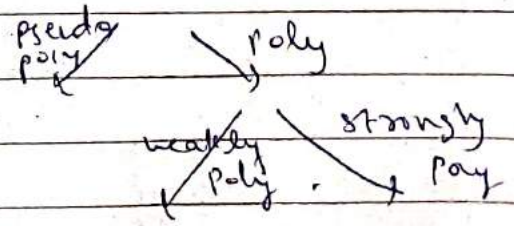
$O(m^2 \log_2 ( )$

efficient
dependent on no. of bits
weakly polynomial.

(*) strongly polynomial
(relevant if input consists of integers)

An algorithm runs in strongly polynomial time if the number of operations is bounded by a polynomial in the number of integers in the input

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



PAGE No.
DATE / /

Edmond's Karp

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

shortest path → least number of edge.

$O(nm^2)$

Ford Fulkerson	$O(C \cdot m)$	pseudo polynomial
Scaled Ford Fulkerson	$O(m^2 \log_2 C)$	weakly polynomial.
Edmonds Karp	$O(m^2 n)$	strongly polynomial.
Orlin + KTR	$O(nm)$	

Recently developed methods to solve maxflow in close to linear time wrt m

These FPTAS approximations of max flow not exact maxflow.

PAGE No.
DATE / /

DAA week 10 Notes

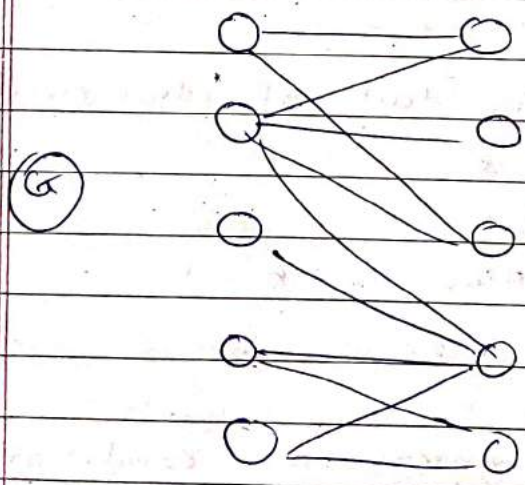
Applications of Max Flow Algo.

Network Flow

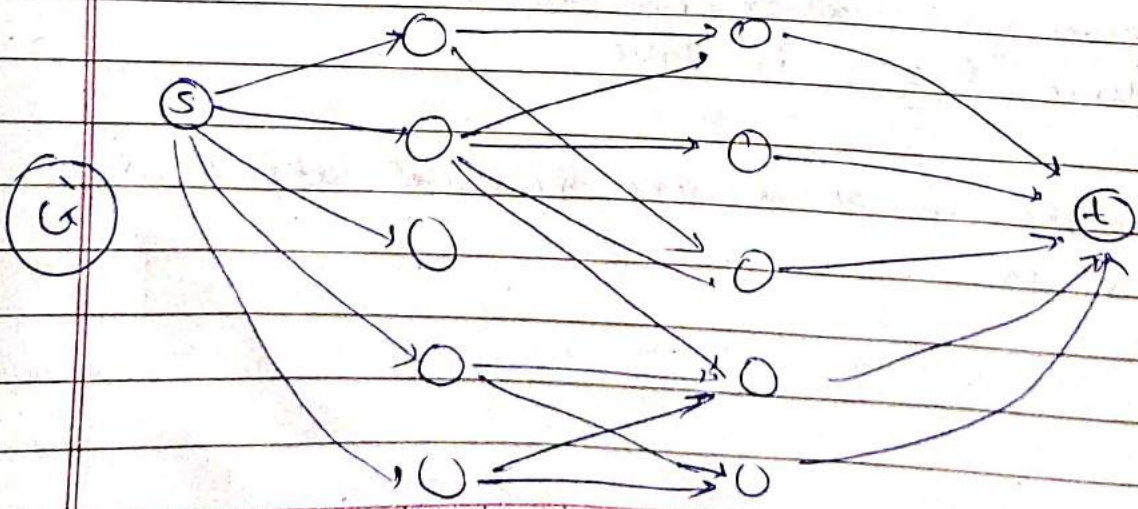
(+) Bipartite Matching problem.

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

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



Given graph G find matching M of largest possible size.



with all capacities equal to 1.

Now Find Max-Flow in G'

PAGE No.
DATE / /

suppose there is a matching in G consisting of k edges $(x_{i_1}, y_{i_1}) \cdots (x_{i_k}, y_{i_k})$.
Flow across each edge is 1

∴ By satisfying capacity and conservation condition
Flow = k

Conversely if there is a flow f' in G' of value k, there will be k edges.

(7.34) → M' contains k edges

(7.35) Each node in X is the tail of atmost one edge in M'

Proof: suppose x is tail of two nodes

But incoming flow to x = 1

Not satisfying conservation condition.

Hence assumption not true
Hence (7.35) is True.

(7.36) Each node in Y is the head of atmost one edge in M'

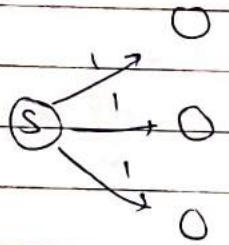
PAGE No.
DATE / /

(7.37) The size of maximum matching in G is equal to the value of the max flow in G', and the edges in such a matching in G are the edges that carry flow from X to Y in G'

(7.38) The Ford Fulkerson Algo can be used to find maximum matching in a bipartite graph in $O(mn)$ time

PROOF: $O(c \cdot m)$

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



$\therefore c = 1 * n$ (↑ no. of nodes.)

$O(n \cdot m)$

Augmenting paths are also called alternating paths in the context of finding a maximum matching.

What if there is no perfect matching?

- Till now we will find matching using max flow and then find if matching is perfect or not

- We can also see to find a cut of capacity less than n.

If capacity is less than 'n' then flow < 'n'

∴ No perfect Matching

PAGE No.
DATE / /

Consider a subset of nodes $A \subseteq X$

$\Gamma(A) \subseteq Y$ → denote all nodes that are adjacent to nodes in A.

(7.39) If a bipartite graph $G = (V, E)$ with two sides X and Y has a perfect matching, then for all $A \subseteq X$ we must have $|\Gamma(A)| \geq |A|$.

(7.40) Assume that the bipartite graph $G = (V, E)$ has two sides X and Y such that $|X| = |Y|$. Then the graph G either has a perfect matching or there is a subset $A \subseteq X$ such that $|\Gamma(A)| < |A|$. A perfect matching or an appropriate subset A can be found in $O(mn)$ time.

(*) Edge Disjoint Path problem. (directed Graphs)

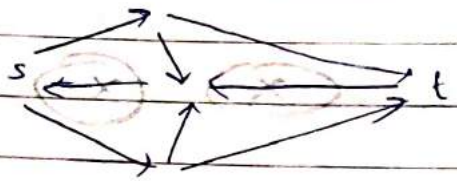
We say that a set of paths is edge-disjoint if their edge sets are disjoint, that is no two paths share an edge, though multiple paths may go through some of the same nodes.

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

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

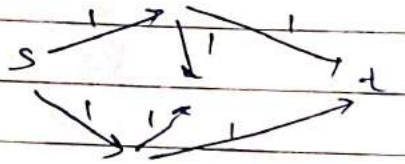
Moreover flow in G' should identify the set of edge-disjoint paths in G.

(G)



No incoming edge on source
No outgoing edge from sink

(G')



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

(7.41) If there are k edge disjoint paths in a directed graph G from s to t, then value of maximum s-t flow in G is atleast k.

(7.42) If F is a 0-1 valued flow of value v, then the set of edges with flow value $f(e) = 1$ contains a set of v edge-disjoint paths.

(7.43) There are k edge-disjoint paths in a directed graph G from s to t if and only if the value of the maximum value of an s-t flow in G is atleast k.

PAGE No.
DATE / /

(7.44) Ford Fulkerson Algo can be used to find a maximum set of edge-disjoint s-t paths in a directed graph G in $O(m \cdot n)$ time.

(7.45) In every directed graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal seperates s from t.

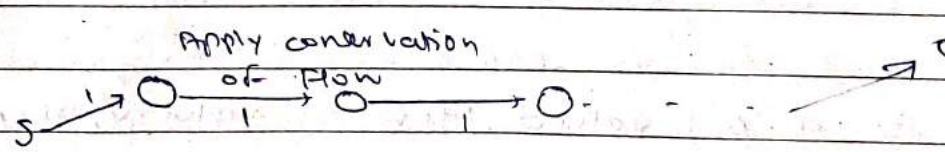
To prove: converting G to G' and Finding Max Flow in it gives edge disjoint path in G.

We will show that there are k edge disjoint paths in G if there is a Flow of value k in G'

Proof:

(A) If we have k edge disjoint s-t path in G, we can Find a Flow of value k in G'

(B) If we have a Flow of value k in G', we can Find k edge point s-t path in G.

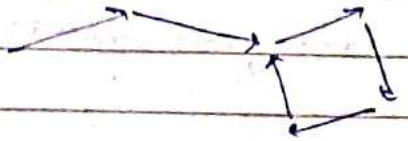


After reaching we have Flow 1

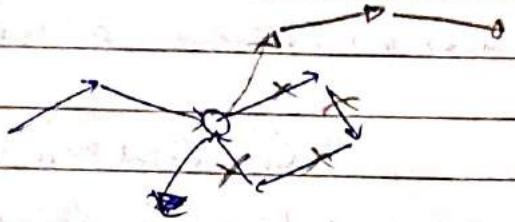
No same exploring is done for other $k-1$ paths.

PAGE NO.
DATE / /

problem in this kind of exploring



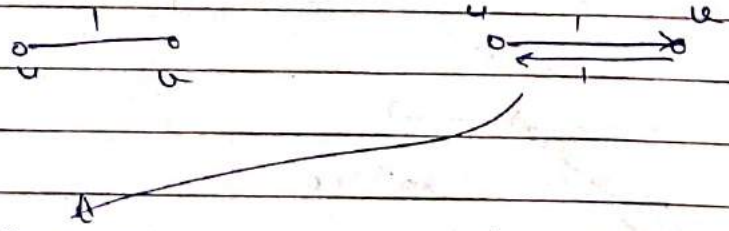
while expanding we land on a particular node again



For this node
incoming edge = 2
outgoing = 1
∴ it must have 1 more outgoing edge to conserve flow

So we will remove the cycle and explore the other edge.

(*) Disjoint Path problems (undirected Graphs)



But this may lead to problem as (u, v) and (v, u) are seperate edges in directed graphs and can be used in two diff paths, but in undirected graph it is a single edge

Hence we have to make sure only one of this edge is used.

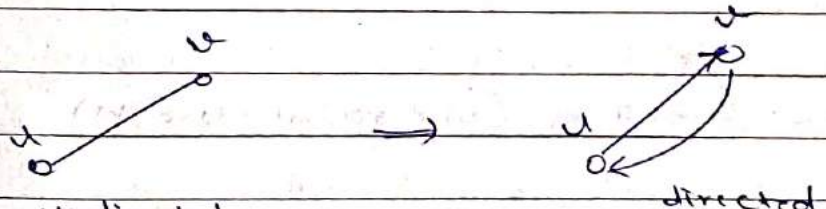
PAGE No.
DATE / /

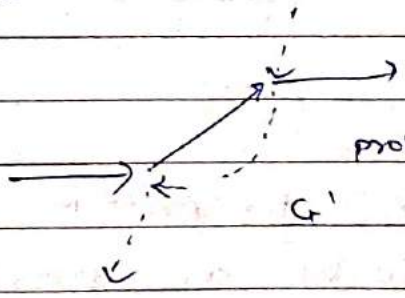
(7.46) In any Flow network, there is a maximum Flow f where for all opposite directed edges $e = (u, v)$ and $e' = (v, u)$, either $f(e) = 0$ or $f(e') = 0$. If the capacities of the Flow network are integral, then there also is such an integral maximum Flow.

(7.47) There are K edge-disjoint paths in an undirected graph G from S to T if and only if the maximum value of s-t Flow in directed version G' of G is atleast K.

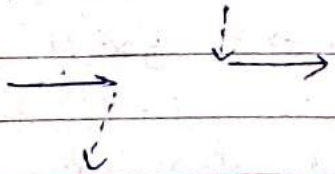
Further, Ford Fulkerson can be used to Find a maximum set of disjoint s-t paths in an undirected graph G in $O(m \cdot n)$ time.

(7.48) In every undirected graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal seperates s and t.





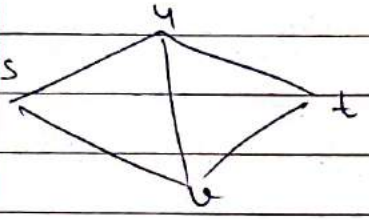
remove cycle before exploring starts

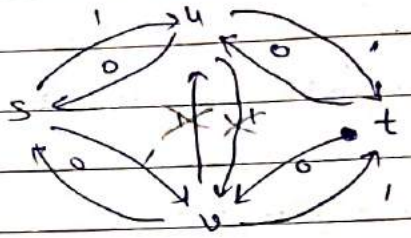


PAGE No.
DATE / /

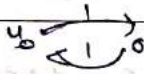
~~(*) Node disjoint s-t paths~~

Example

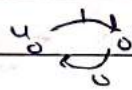




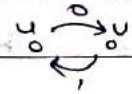
Tip: remove cycles corresponding to a edge



remove

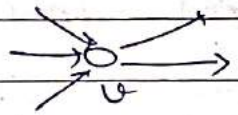


OR

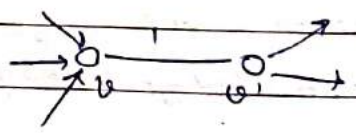


No need to remove

(*) Node disjoint s-t paths.
Given a directed graph G with $s, t \in V$, find the max number of node disjoint s-t paths in G.



G



G'

PAGE NO.
DATE / /

Extensions to Max-Flow problem

→ circulation with Demands

→ circulation with Demands and lower bounds.

(#) circulation with Demands.

$G = (V, E)$

d_v → demand associated to any node v

if $d_v > 0$: indicates node v has a demand of d_v for flow; node in a sink

if $d_v < 0$: indicates node v has a supply of d_v; node is a source

if $d_v = 0$: neither source nor sink.

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

1) capacity conditions

For each edge $e \in E$ $\quad 0 \le f(e) \le c_e$

2) Demand condition

For each node $v \in V$

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

PAGE No.
DATE / /

(7.49) If there exists a Feasible circulation with demands $\{d_v\}$

then $\sum_v d_v = 0$

Proof:

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

For some edge e

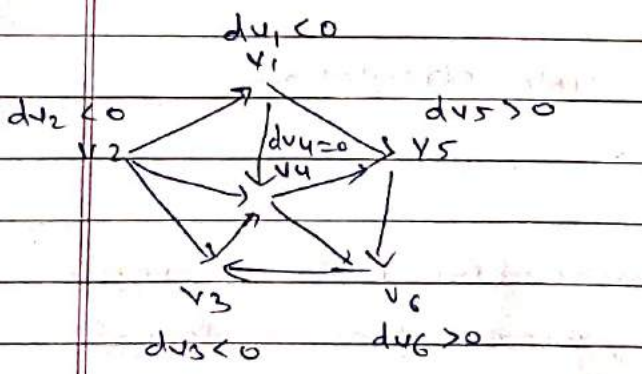
This edge will be $f^{out}(u)$ and $f^{in}(u)$

∴ each edge appears twice with +ve and -ve

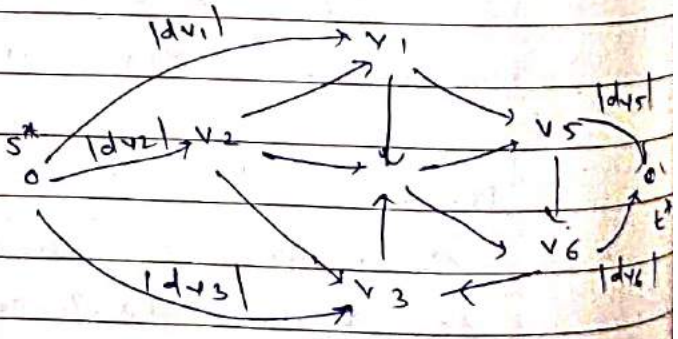
$$\therefore \sum_v d_v = \sum_v f^{in}(v) - \sum_v f^{out}(v) = 0.$$

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

Demand = Supply.



G



G'

G' in directed graph without demands

we added super source to satisfy demands for all sources

PAGE No.
DATE / /

(7.50) There is a Feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s^*-t^* Flow in G' has value D. If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.

Run Max Flow on G' ⟶ f

Say $v(f) = x$

Can $x < D$?

Yes, if no Feasible circulation

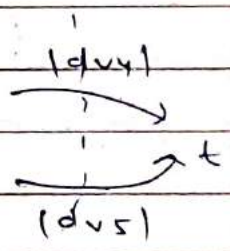
Can $x > D$?

Not possible

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

A

Any cut



$d_{v4} + d_{v5} = D$

$v(f) \leq D$.

Can $x = D$?

Yes, if Feasible circulation.

PROOF:

(A) If there is a Feasible circulation f with demand value $\{d_v\}$ in G, we can find a Max Flow in G' of value D.

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

PAGE NO.
DATE / /

(7.51) The graph G has a feasible circulation with demands $\{d_v\}$ if and only if for all cuts (A, B)

$$\sum_{v \in B} d_v \leq c(A, B)$$

(#) Circulation with Demands and Lower Bounds

$G = (V, E)$

$d_v \rightarrow$ demand associated with any node

1> capacity condition

For each $e \in E$ we have $\ell_e \leq f(e) \leq c_e$

2> Demand condition

For every $v \in V$ $\quad f^{in}(v) - f^{out}(v) = d_v$

Circulation with demands & lower bounds $\xrightarrow{\text{convert}}$ circulation with demands $\xrightarrow{\text{convert}}$ Max-flow

Solution in 2 steps

Pass #1 Find f_0 to satisfy all ℓ's

↳ removing capacity condition

Pass #2 use remaining capacity of the network to Find a feasible circulation f_1 (if it exists)

Combine two flows $(f) = f_0' + f_1$

PAGE No.
DATE / /

Solution :

1. Push flow f_0 through G where $f_0(e) = l_e$

2. Construct G' where $c_e' = c_e - l_e$

$d_v' = d_v - L_v$

3. Find feasible circulation in G'

call this f'

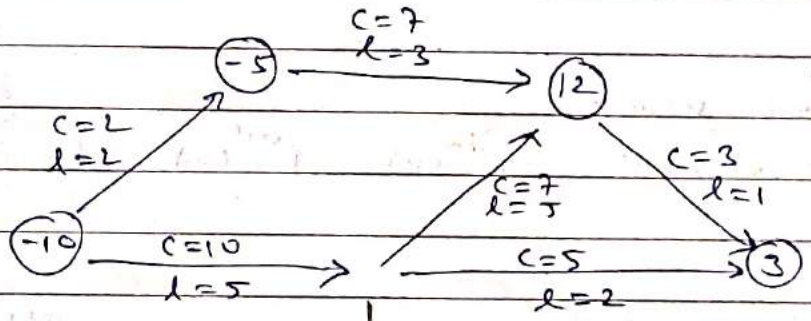
4. If no feasible circulation in G'

∴ no feasible circulation in G

otherwise, feasible circulation in G

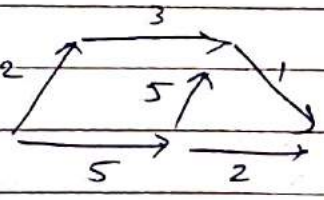
$= f_0 + f_1$

eg :

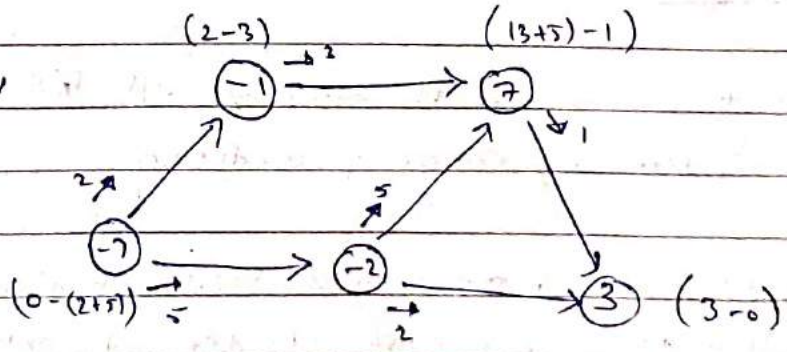


↳ nothing mentioned means $l=0$

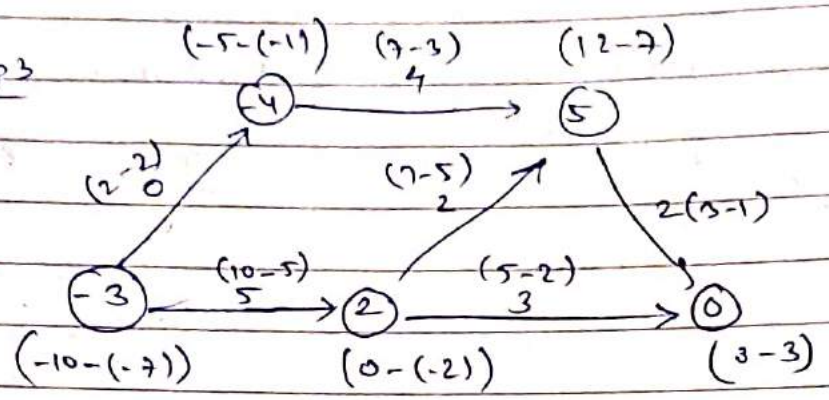
Step1 f_0



Step 2 L_v



Step 3



Now this is ~~Max Flow~~ Circulation with Demands

Solve this to find flow f_1

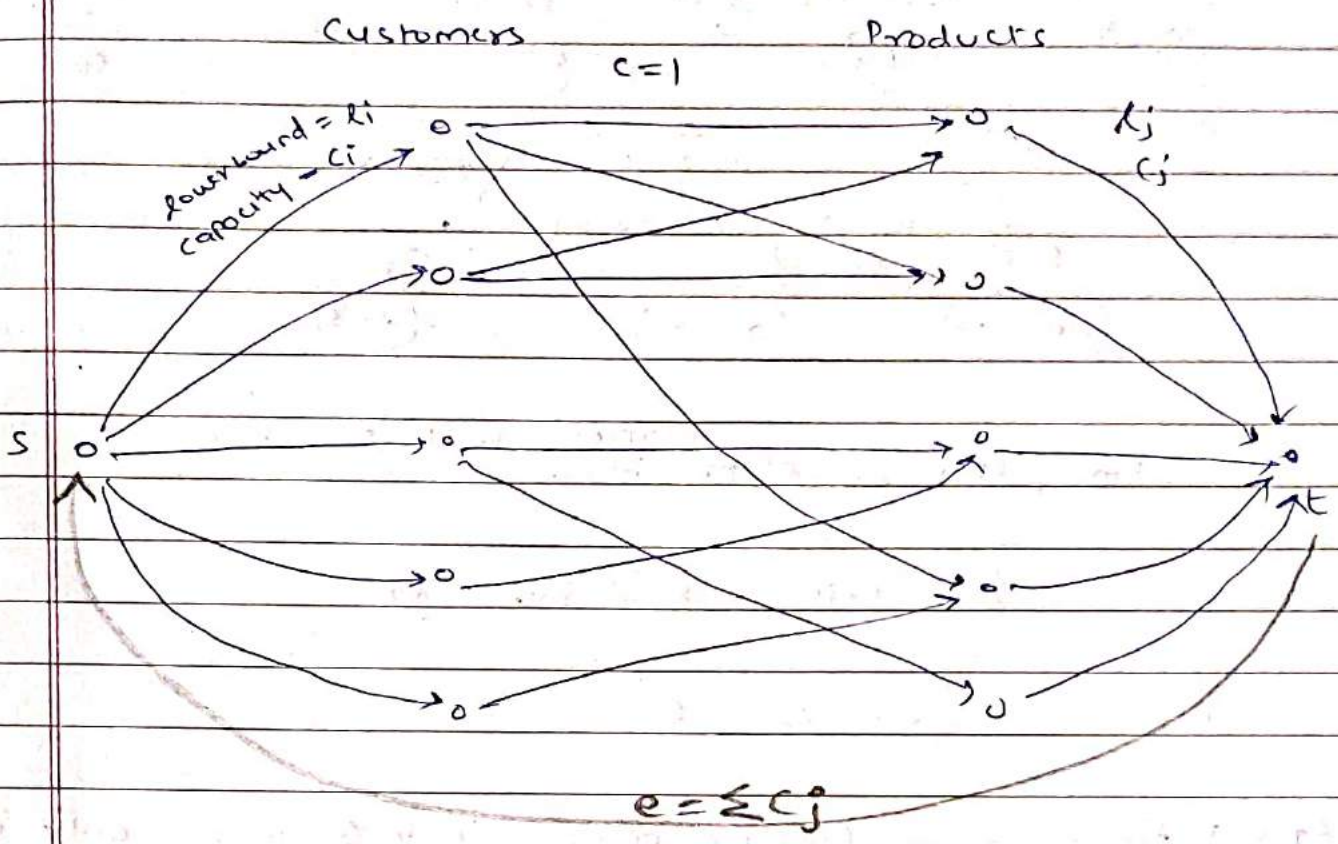
$$f = f_0 + f_1$$

(7.52) There is a feasible circulation in G if and only if there is a feasible circulation in G'. If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer-valued.

(#) survey design.

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

PAGE No.
DATE / /



lowerbound = l_i = min ques asked to customer i

Capacity = c_i = max ques asked to customer i

l_j → min ques asked about pdt j

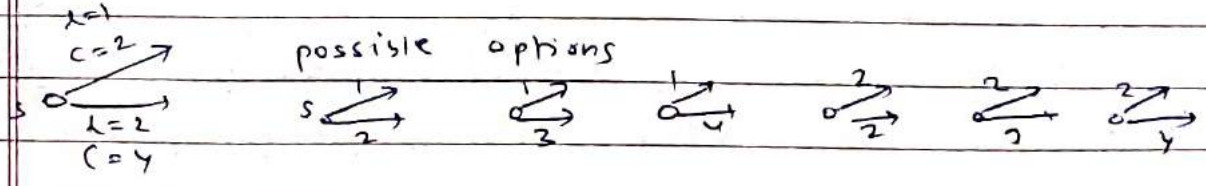
c_j → max ques asked about pdt j

This is ≠ Circulation with Demand & Lower Bound

But what is demand of source & sink

$$d_{source} = f^{in}(source) - f^{out}(source)$$

$$= - f^{out}(source)$$



PAGE No.
DATE / /

Hence we do not exactly how much flow is there ~~in~~ out from source

Similarly for sink

$$d_{sink} = f^{in}(sink) - f^{out}(sink)$$
$$= f^{in}(sink)$$

∴ To make demands of source & sink = 0

we already know

$$-d_{source} = d_{sink}$$

$$d_{sink} = \sum_{\substack{e \text{ in to} \\ sink}} c_e$$

∴ we add a backward edge from sink to source of capacity $= \sum_{j \text{ into } t} c_j$

(7.53) The graph G just constructed has a feasible circulation if and only if there is a feasible way to design the survey.

PAGE NO.
DATE / /

ADA Week 10 Notes continue

(*) Min Flow Problem:

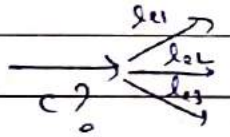
Input: Directed graph $G = (V, E)$
source $s \in V$
sink $t \in V$
l_e for each edge $e \in E$
There are no c_e's

Objective: Find a feasible flow of minimum possible value.

SOLUTION:

PASS #1

Satisfy min capacity condition.



$c = \sum l_e$ for all edges

Initially $c = \infty$ replace ∞ with $\sum l_e$

Now Find Feasible circulation.

Solution

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

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

excess flow going through every edge

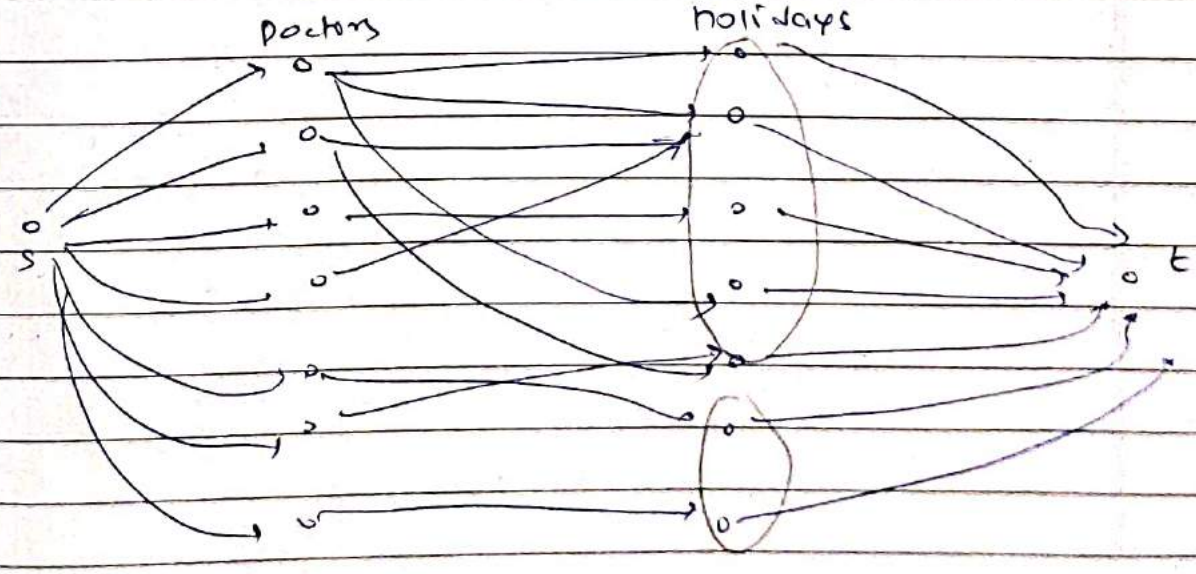
PAGE No.
DATE / /

3. Find maximum flow from t to s in G' — Find maximum flow of this excess flow

4. Min flow $= f - f'$ — minFlow = Flow from satisfying constraints − Max excess flow.

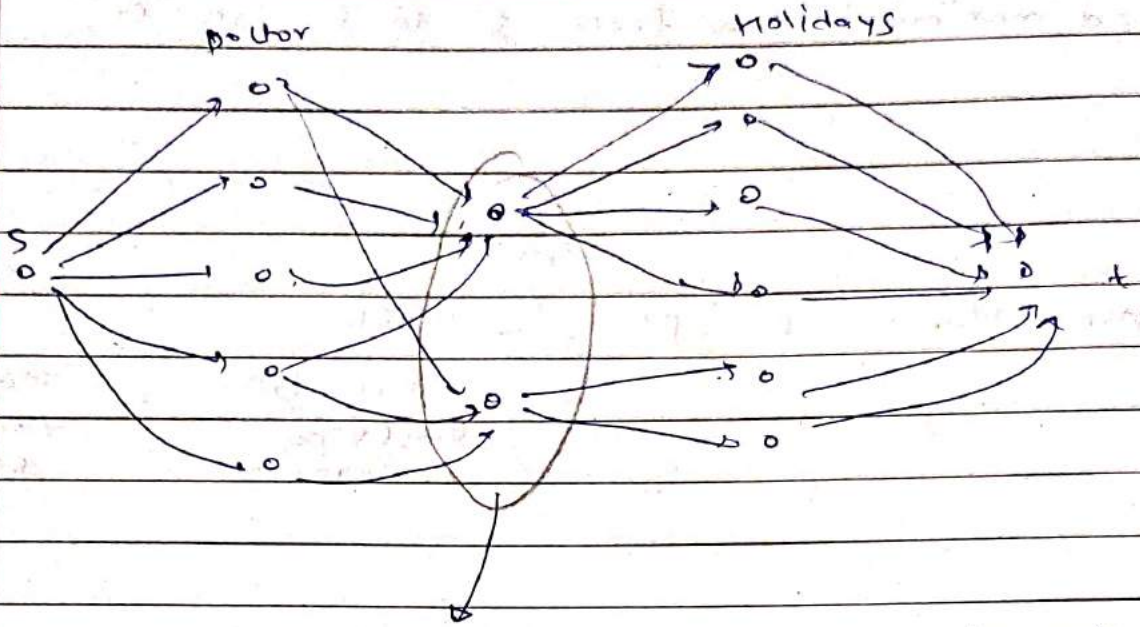
(#) Doctors without weekend problem. (using gadgets)

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



No doctor has working day more than 1 in a particular set

PAGE No.
DATE / /



These nodes physically do not correspond to anything

They are just to satisfy the constraint that no doctor should have more than 1 working day in a particular holiday set.

Hence they are called gadgets

PAGE No.
DATE / /

ADA, Week 12 Notes

(*) **Independent set Problem:**

Given $G = (V, E)$, we say a set of nodes $S \subseteq V$ is independent if no two nodes in S are joined by an edge.

(*) **Vertex Cover Problem:**

Given $G = (V, E)$, we say a set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has atleast one end in S.

(*) **Set Cover Problem:**

Given a set U of n elements, a collection $S_1, \ldots S_m$ of subsets of U, and a number K, does there exist a collection of atmost K of these sets whose union is equal to all of U.

(*) **3 SAT Problem:**

SAT problem: Given a set of clauses $C_1, \ldots C_k$ over a set of variables $X = \{x_1, \ldots x_n\}$ does there exist a satisfying truth assignment?

eg: $X = \{x_1, x_2, x_3\}$

$C_1 = (x_1 \vee \bar{x}_2)$

$C_2 = (\bar{x}_1 \vee \bar{x}_3)$

$C_3 = (x_2 \vee \bar{x}_3)$

$C_1 \wedge C_2 \wedge C_3$

does there exist truth assign of x_1, x_2, x_3 such that $C_1 \wedge C_2 \wedge C_3$ is True.

PAGE No.
DATE / /

(*) A large set of problems is in the "gray area". A poly time solution to anyone would imply poly time algo for all of them.

(*) Poly Time Reductions.

$Y \leq_p X$ (Y is poly time reducible to X)

if Y can be solved using poly number of standard computational steps plus a poly number of calls to a black box that solves X.

$Y \leq_p X$ (Problem X is atleast as hard as Problem Y)

In other words X is powerful enough to let us solve Y.

(8.1) Suppose $Y \leq_p X$. If X can be solved in poly time, then Y can be solved in poly time.

(8.2) Suppose $Y \leq_p X$. If Y cannot be solved in poly time, then X cannot be solved in poly time {As X is atleast as hard as Y}

PAGE No. DATE / /

(#) Reductions

#1 Indp Set and Vertex Cover

(b) Indp set :- Given graph G and number K does G contain an indp set of size atleast K?
(decision version)

Find largest indp set in graph G
(optimization version)

optimization prob is atleast as hard as decision prob.

(a) Vertex Cover :- Find smallest vertex cover in G
(opt version)

- Given graph G and number K, does G contain a vertex cover set of size atmost K
(decision version)

FACT : let $G=(V,E)$ be a graph, then S is an indp set if and only if its complement V-S is a vertex cover set.

Proof :

(A) Suppose S is an indp set

Consider an edge e

Case1 : u is in S and v is not
V-S will have v and not u.

Case 2 : v is in S and u is not
V-S will have u and not v.

Case 3 : Neither v and u in S
V-S will have both v and u.

PAGE No.
DATE / /

(B) Suppose V-S is a vertex cover set then S in indp set:

similar proof.

(8.4) **Indp set $\leq_p$ vertex cover**

proof: If we have a blackbox to solve vertex cover, then we can decide whether G has an indp set of size atleast K by asking the blackbox whether G has a vertex cover of size atmost n-k.

(8.5) **Vertex Cover $\leq_p$ Indp set**

Similar proof.

#2 Vertex Cover to Set Cover

Note: Indp set and Vertex cover are two diff types of prob

Indp set is a type of packing problem:
The goal is to "pack in" as many vertices as possible, subject to conflicts (the edges) that try to prevent one from doing this.

Vertex cover is a type of covering problem:
The goal is to cover all the edges in the graph using as few vertices as possible.

Set Cover is also a covering problem

PAGE No.
DATE / /

Intuitively it feels vertex cover is a special case of set cover

(8.6) **Vertex Cover $\leq_p$ set Cover**

Vertex Cover	Set Cover
$G=(V,E)$	$G=(V,E)$
Set $S \rightarrow$ vertex cover such that every edge has atleast one end in set S.	$S \rightarrow$ set cover such that union of all sets in S gives V.

∴ To convert vertex cover to set cover we need some number of sets, and select 'k' of the sets such that their union covers all nodes

eg: $U = \{x_1, x_2, x_3\}$

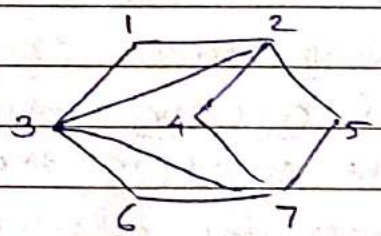
$S_1 = \{x_1, x_2\}$ $S_2 = \{x_3, x_1\}$

$S_3 = \{x_2, x_3\}$ $S_4 = \{x_1\}$

If we select (S_1 and S_2) $\boxed{S_1 \cup S_2 = U}$.

$\therefore K = 2$

min number of subsets chosen.



Forming sets

$S_1 = \{(1,2)\ (1,3)\}$

$S_2 = \{(2,1)\ (2,3)\ (2,4)\ (2,5)\}$

↑ all edges incident on 2

$S_7 = \{\quad\}$

↑ all edges incident on 7

PAGE No.
DATE / /

pg 458

Suppose vertex cover Ans = {2, 3, 7} (at most one)

∴ All edges in graph will have end from {2, 3, 7}

∴ we can say $S_2 \cup S_3 \cup S_7$ = All Edges = Y

∴ G has a vertex cover of size K, if the corresponding set cover instance has K sets whose union contains all edges in G.

Proof: (A) IF I have a vertex cover set of size K in G, I can find a collection of K sets whose union contains all edges in G.

Given {2, 3, 7} as vertex cover set.
$S_2 \cup S_3 \cup S_7$ covers all edges.

(B) IF I have K sets whose union contains all edges in G, I can find a vertex cover set of size K in G.

SAME AS ABOVE

(*) Set Packing Problem.

Given a set U of n elements, a collection $S_1, \ldots S_m$ of subsets of U, and a number K, does there exist a collection of atleast K of these sets with the property that no two of them intersect.

PAGE NO.
DATE / /

(8.7) Indp set $\leq_p$ Set Packing

Imp Note: we transforming $Y \leq_p X$
we transform a single instance of Y to single instance of X

(#) Reduction using Gadgets.

Mainly gadgets used ~~when~~ when we want to convert

Non graph problem ⟹ Graph problem.

SAT problem:
Given n Boolean variables
A clause is disjunction of terms $t_1 \vee t_2 \cdots \vee t_\ell$
where $t_i \in \{x_1, \bar{x}_1, x_2, \bar{x}_2, \cdots x_n, \bar{x}_n\}$

Find truth assignment for variables
$x_1, \cdots x_n$ such that
(Clause 1) $\wedge$ (Clause 2) $\cdots$ $\wedge$ (Clause k) = T

Example:
$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_3)$
$x_1 = 1 \quad x_2 = 1 \quad x_3 = 1$ (Not satisfying constraints)

$x_1 = 0 \quad x_2 = 0 \quad x_3 = 0$ (✓)

$x_1 = 1 \quad x_2 = 0 \quad x_3 = 0$ (✓)

PAGE No.
DATE / /

SAT problem: Given a set of clauses $C_1 \ldots C_K$ over a set of variables $X = \{x_1, \ldots x_n\}$ does there exist a satisfying truth assignment.

(8.8) **3 SAT $\leq_p$ Indp Set**

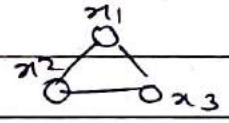
Non graph prob

graph prob

selects max nodes such that no two nodes from selected set form a pair.

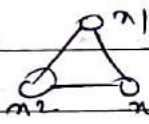
Plan: Given an instance of 3 SAT with K clauses, build a graph G that has an ~~edge~~ indp set of size K if the 3 SAT ~~probl~~ instance is satisfiable

$(x_1 \vee x_2 \vee x_3)$



To make this clause T we just need to make x_1 or x_2 or x_3 any one of them T.

So Indp Set will pick any one vertex from this



as if two vertex is selected they will have edge between them.

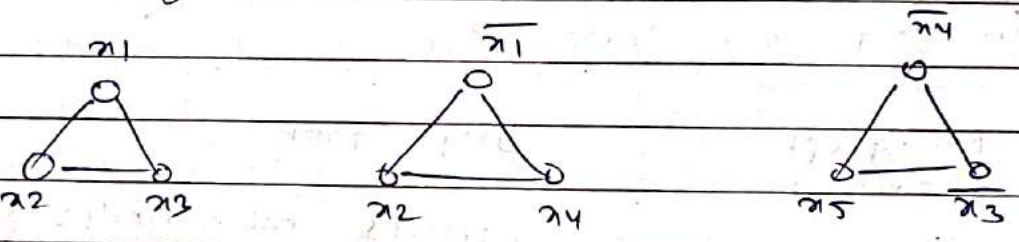
PAGE No.
DATE / /

So given K clauses there will K Δ's.

eg: $C_1 = (x_1 \vee x_2 \vee x_3)$

$C_2 = (\bar{x_1} \vee x_2 \vee x_4)$

$C_3 = (\bar{x_4} \vee x_5 \vee \bar{x_3})$

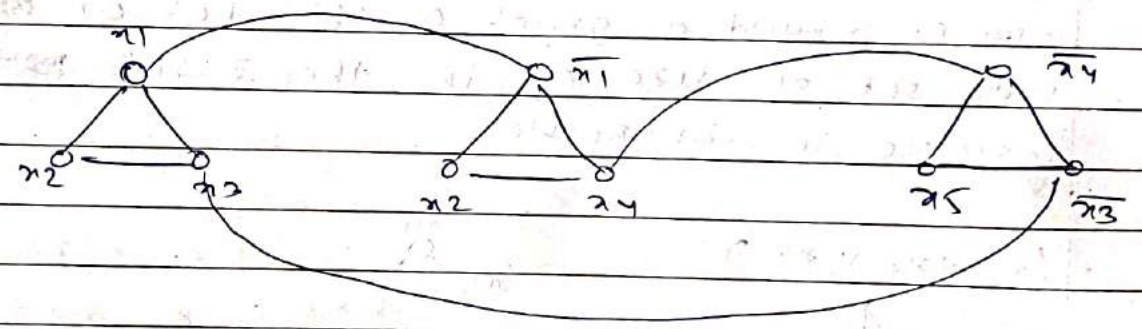


Problem: suppose indp set is

$\{x_1, \bar{x_1}, \bar{x_4}\}$

then $x_1 = T$ and $\bar{x_1} = T$ (Not possible)

Hence little modification is needed.



connecting all x_i and $\bar{x_i}$ makes sure that only one of them is picked.

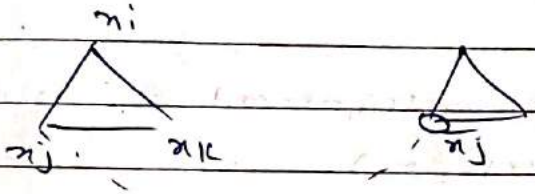
Claim: 3 SAT instance is satisfiable if the graph G has an indp set of size K

As from every Δ if indp set selects one node 3 SAT problem is solved

PAGE No.
DATE / /

Proof: (A) If the 3-SAT instance is satisfiable, then there is atleast one node label per triangle that evaluates to 1

Let S be a set containing one such node From each Δ.



If there is a satisfying truth assignment then atleast one of x_i, x_j, x_k would have value = 1

only problem could be if x_j and $\bar{x}_j$ both = 1 But by adding such edges we have solved that problem.

(B) Suppose G has an indp set S of size atleast K

- if x_i appears as a label in S then set $x_i = 1$
- if $\bar{x}_i$ appears as a label in S then set $x_i = 0$
- if neither x_i nor $\bar{x}_i$ appear as a label in S, then its value does not matter

(4) Transitivity Reductions

If $Z \leq_p Y$ and $Y \leq_p X$

Then $Z \leq_p X$

PAGE No.
DATE / /

pg 465

(A) Efficient certification.

1. Poly length certificate
2. Poly Time certificate

3-SAT

certificate t is an assignment of truth values ~~of~~ to variables (x_i)

certifier : evaluate the clauses if all of them evaluate to 1 then it answers yes.

Indp Set

certificate t in a set of nodes of size atleast k in G.

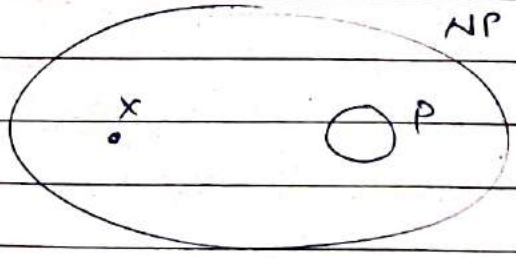
certifier: check each edge to make sure no edges have both ends in the set

check size of set $\geq k$

no repeating nodes.

(Class NP)

is a set of all problems for which there exists an efficient certifier



PAGE No.
DATE / /

if X is in NP

and for all $Y \in NP$

$Y \leq_p X$, then X must be hardest problem.

(3-SAT) is the hardest problem.

Such a problem is called NP-complete.

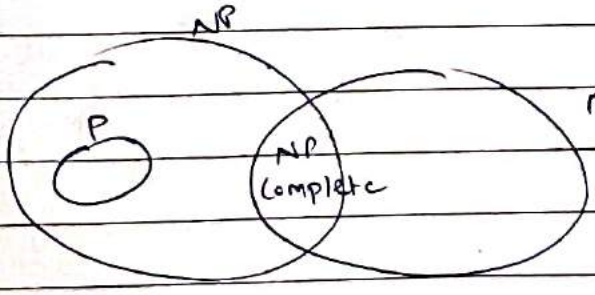
3-SAT $\leq_p$ Indp set $\leq_p$ vertex cover $\leq_p$ set cover

∴ 3SAT, Indp set, Vertex Cover, Set Covers are all NP complete.

How to show problem is NP-complete?

Basic strategy to prove problem X is NP-complete.

1. we need to prove X is in NP-class
2. Choose a problem Y know to be NP complete
3. Prove $Y \leq_p X$ } (This proves X is atleast as hard as Y and Y is NP-complete ∴ X is NP-complete)



NP Hard problems are atleast as hard as NP complete problems.

PAGE No.
DATE / /

If From given 3 steps
we only prove step (3)
i.e $Y \leq_p X$

we prove X is NP-Hard
only if X is (NP and NP-Hard)
↑ NP-complete.

PAGE No.
DATE / /

ADA, Week 13 Notes

④ Travelling Salesman problem and Hamiltonian cycle.

= Decision version of TSP
Given a set of distances on n cities and a bound D, is there a tour of length/cost atmost D?

Hamiltonian cycle
Hamiltonian cycle, if it visits each vertex exactly once.

Problem statement: Given an undirected graph G, is there a Hamiltonian cycle.

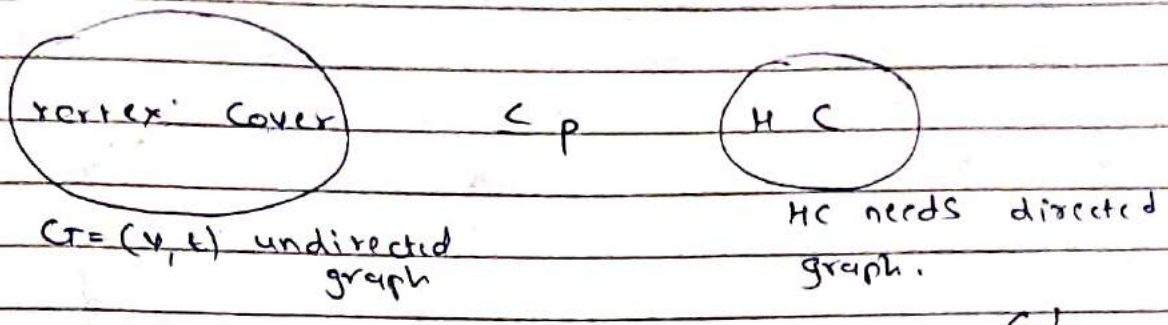
⟨Q⟩ Show that Hamiltonian cycle problem is NP-complete.

Step1: showing Hamiltonian cycle is NP
a. Certificate: ordered list of nodes on the Hamiltonian cycle.
b. Certifier: (i) check to make sure there is an edge between each pair of adjacent nodes in the list
(ii) All nodes are visited
(iii) No repeated nodes
(iv) edge between last and first nodes in the list.

Step2: Choose a problem already known to be NP-complete
Vertex Cover

PAGE No.
DATE / /

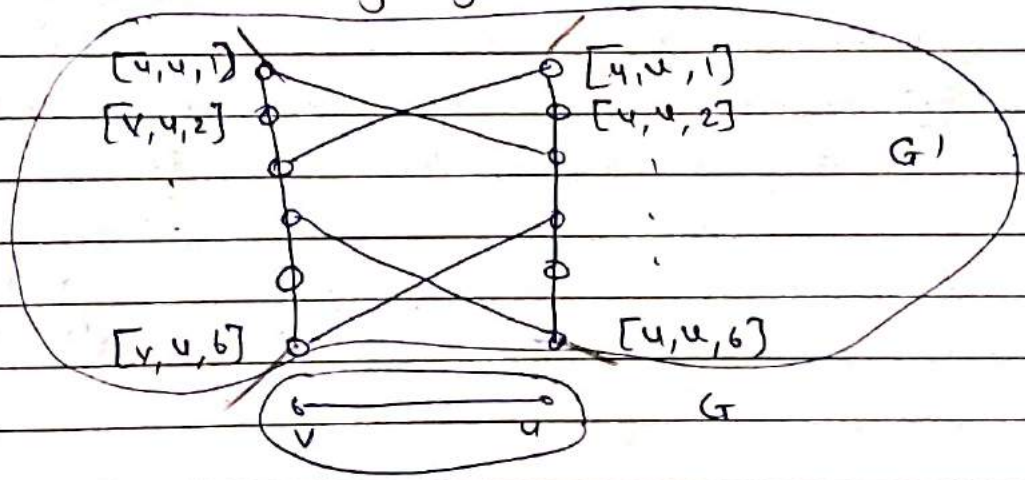
Step3 : Prove vertex cover $\leq_p$ Hamiltonian cycle.



Plan: Given an undirected graph $G = (V, E)$ and an integer K, we construct $G' = (V', E')$ that has a Hamiltonian cycle if G has a vertex cover of size atmost K

Construction of G'

For each edge (v, u) in G
G' will have one gadget w.r.t u.



This gadget can be connected to other gadgets only through 4 corner nodes

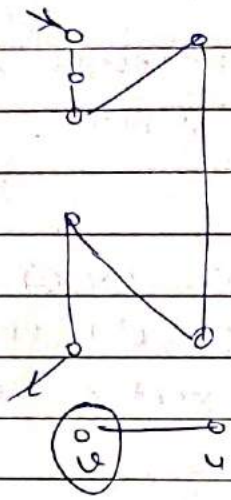
[u,u,1] [u,u,6] [u,u,1] [u,u,6]

Note $Y \leq_p X$

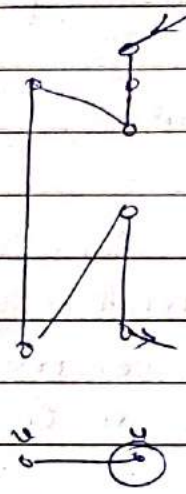
convert Y to prob type X → BlackBox solving X → Get Ans

PAGE NO.
DATE / /

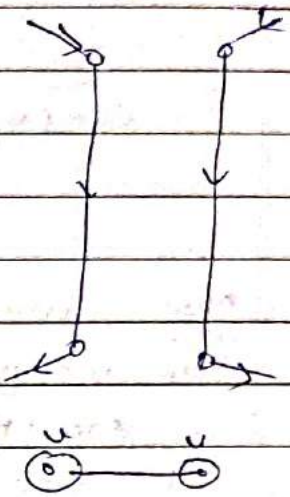
There are 3 ways for HC to visit all nodes of this edge.



HC enters the gadget from v.



HC enters the gadget from u



HC covers v and covers some other vertices then cover u.

There are some other vertices in G'

(*) selector vertices: There are k selector vertices in G' $s_1, s_2 \cdots s_k$

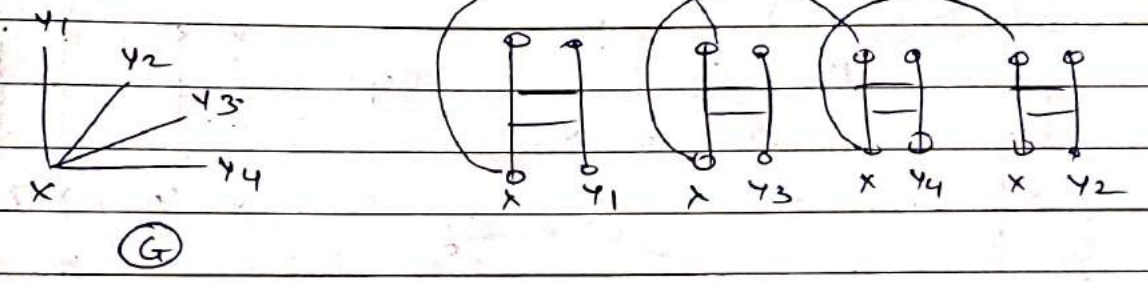
This k comes from vertex cover problem statement.

PAGE No.
DATE / /

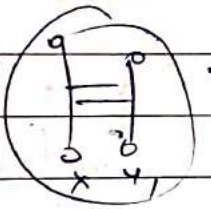
Explanation of how G' is formed.

1. For each vertex $v \in V$ we add edges to join pairs of gadgets in order to form a path going through all the gadgets corresponding to edges incident on v in G.

eg:

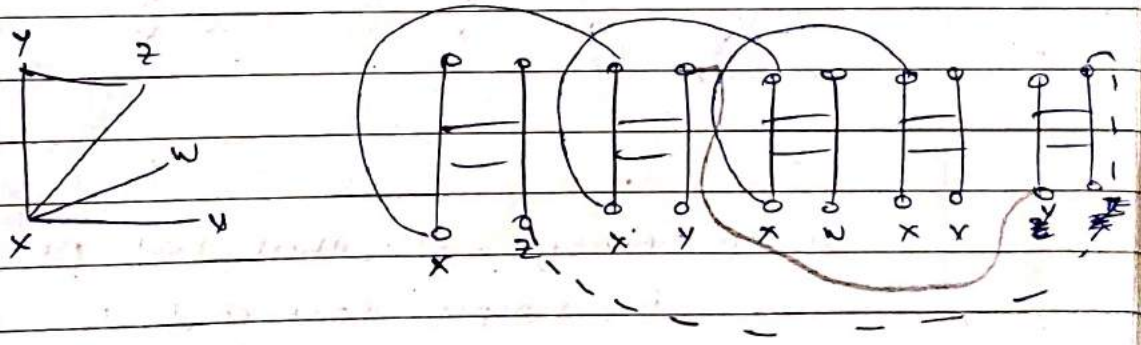


(G)



This represents a gadget for edge xy_1

eg:



Tips: • Connection is always Top ↓ Bottom
OR
Bottom ↓ Top.

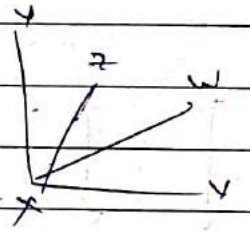
• For every vertex having more than 1 edge such connection is done

PAGE No.
DATE / /

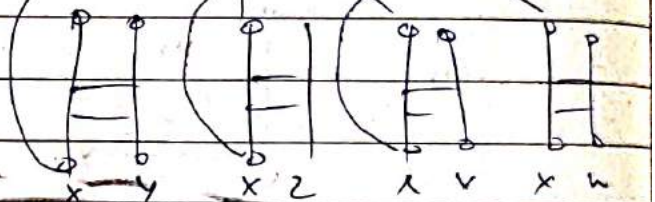
2. Final set of edges in G' join the first vertex [x, y, 1] and last vertex [x, y(deg(x)), 6] of each of these paths to each of the selector vertex.

Here only 1 selector vertex is added for every vertex in G

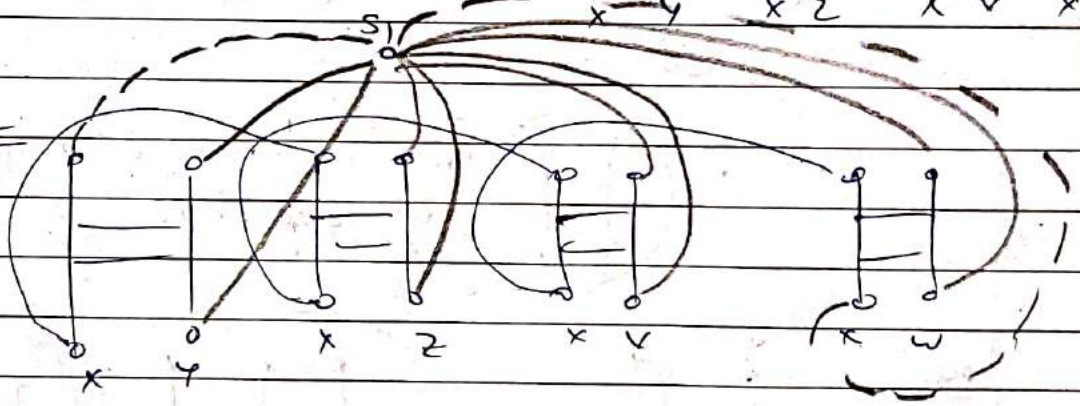
Eg: (1)



Step 1



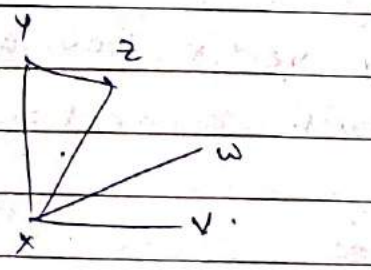
Step 2



Pro Tip: Every node that has no connectivity is simply connected to the selector vertex

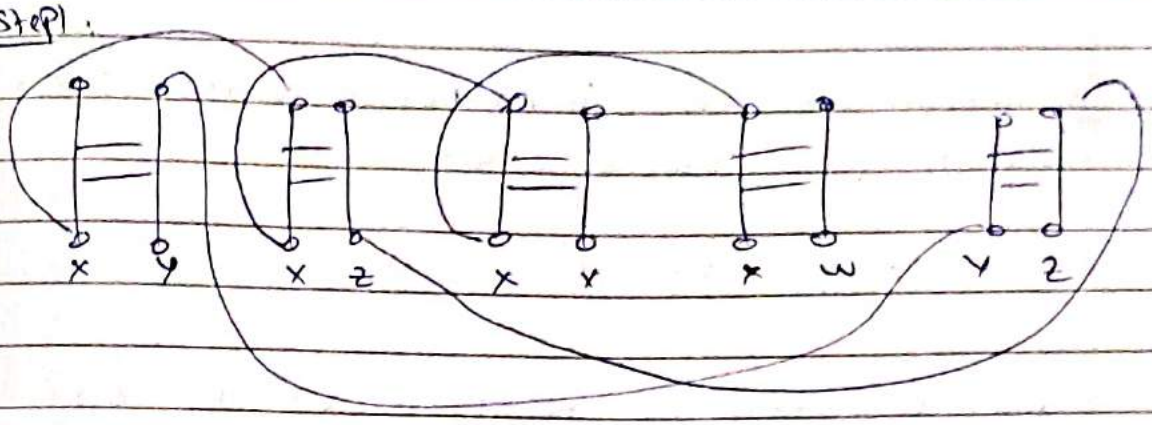
Eg (2):

G

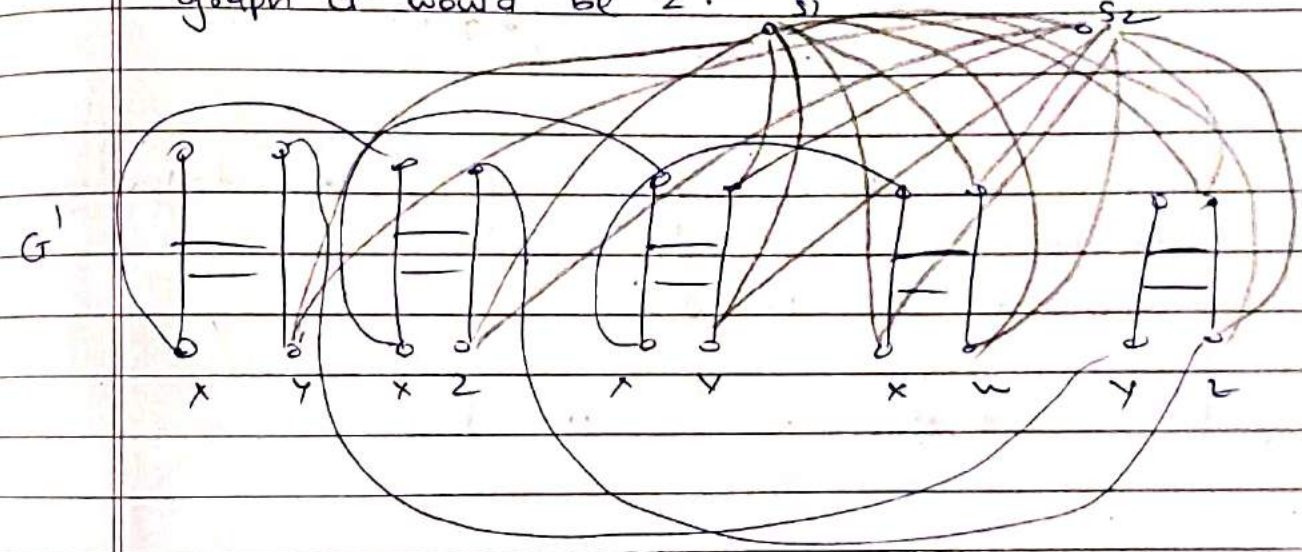


PAGE No.
DATE / /

Step1 :

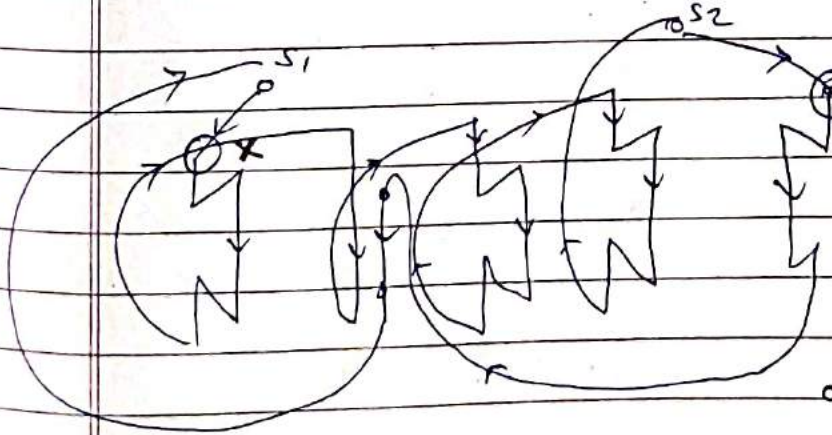


Step2 : Here $K = 2$ ~~are~~ as vertex cover in graph G would be 2.



S_1 and S_2 have same connectivity

Is there a Hamiltonian cycle in G' ? (If yes, then we will get vertex cover in G)



Yes there is Hamiltonian cycle and

and $\{x, z\}$ is vertex cover of graph G.

S_1 selected x to be in vertex cover
S_2 selected z to be in vertex cover

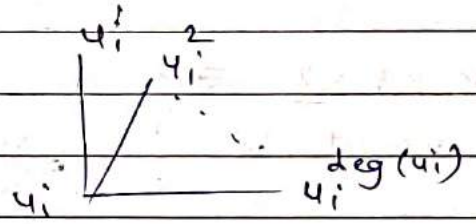
PAGE No.
DATE / /

Step 4: Proof

(A) Suppose that $G = (V, E)$ has a vertex cover of size k, let vertex cover set be

$$S = \{u_1, u_2 \ldots u_k\}$$

we will identify neighbors of u_i as shown here



Form a Hamiltonian cycle in G' by following the nodes in G in this order.

start at s_1 and go to

s_1 → select u_1 → $[u_1, u_1', 1] \ldots [u_1, u_1', 6]$

neigh of u_1

$[u_1, u_1^2, 1] \ldots [u_1, u_1^2, 6]$

$[u_1, u_1^{deg u_1}, 1] \ldots [u_1, u_1^{deg u_1}, 6]$

all neighbours of u_1 visited

s_2 → select u_2 → $[u_2, u_2', 1] \ldots [u_2, u_2', 6]$

$[u_2, u_2^{deg u_2}, 1] \ldots [u_2, u_2^{deg u_2}, 6]$

s_3 ...

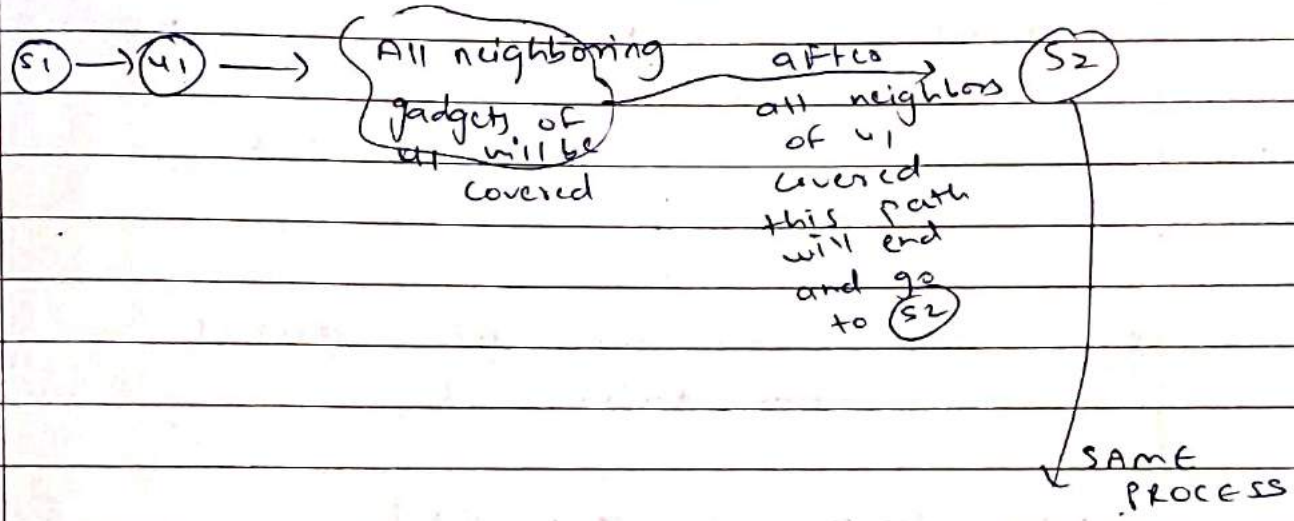
$[u_k, u_k^{deg u_k}, 6]$

PAGE No.
DATE / /

Explanation:

as u_1 is in vertex cover set in G

In G' the selected vertex (s_1) will select u_1 to be in vertex cover set.



(B) Suppose G' has a Hamiltonian cycle C, then the set

$$S = \{ v_j \in V : (s_j, [u_j, u'_j, 1]) \in C \text{ for some } 1 \le j \le t \}$$

will be a vertex cover set in G

∴ Proved Hamiltonian cycle is NP-complete.

PAGE No.
DATE / /

(Q) Now prove TSP is NP-complete

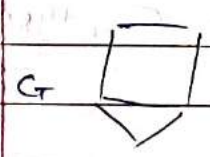
Step1: Prove TSP in NP

a. certificate: tour of cost no more than C

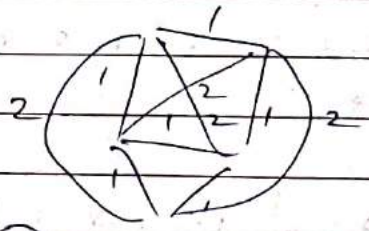
b. certifier: Everything in Hamiltonian cycle
+
check total cost $\leq C$

Step2: choose an NP complete problem
(Hamiltonian cycle)

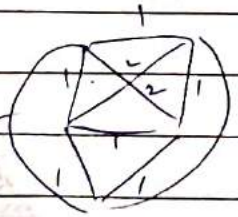
Step3: Prove $HC \leq_p TSP$



Instance of HC



G' · weighted
· directed
· fully connected



G'

Is there a TSP of size 'n' in G'

Black Box that solves TSP

→ Yes ?
No ?

There is a tour in G' of size n if and only if there is Hamiltonian cycle in G.

PAGE No
DATE / /

Step 4 : Proof of reduction

(A) If there is a HC in G then there is TSP in G' of cost 'n'

If there is HC in G
and in G' we put all that edges with cost = 1

∴ There will be a TSP of size 'n' in G'

(B) If there is a TSP of size n in G' then there is HC in G

If TSP cost = n and there are 'n' edges selected
∴ cost of each edge = 1

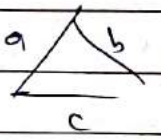
∴ all cost 1 edges selected
∴ HC in G.

PAGE NO.
DATE / /

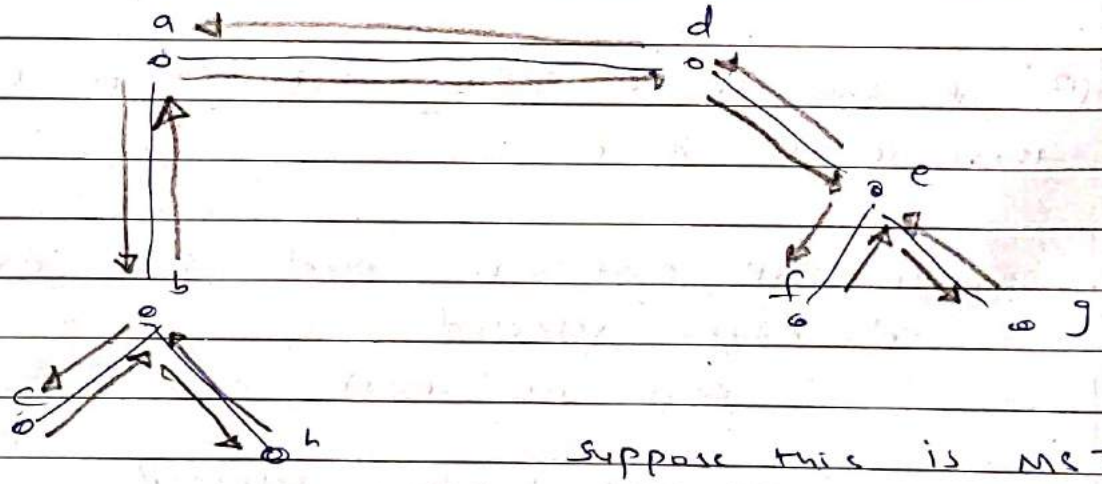
List of NP-complete problems-

* 3-SAT, Vertex Cover, Indp set, Set cover, Hamiltonian cycle, TSP
all knapsack, subset sum, graph 3-Coloring.

(*) Travelling salesman problem (with Δ inequality)



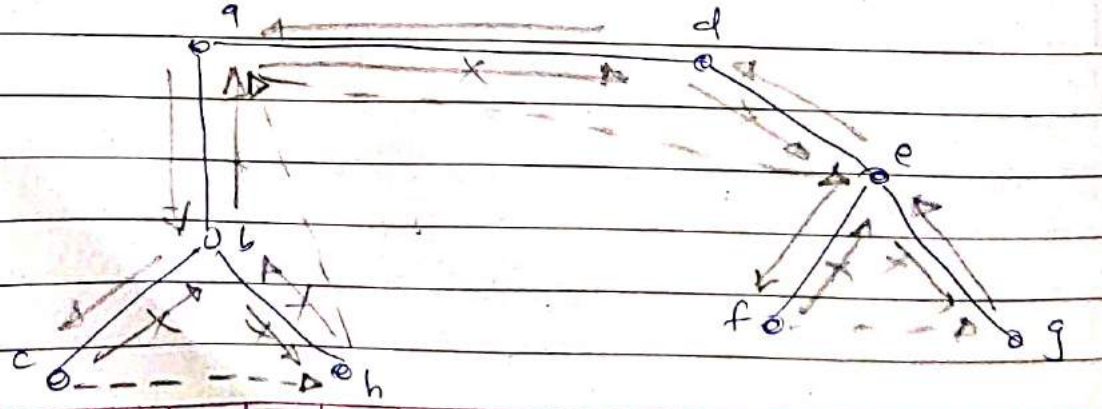
$(a+b) > c$



Suppose this is MST

Initial tour cost = 2 * cost of MST

Applying Δ inequalities



$(cb) + (bh) > (ch)$

PAGE NO.
DATE / /

My approx cost now

Cost $\leq$ 2 * Cost of MST

(Cost of optimal Tour) $\geq$ Cost of MST (As in optimal soln each path will be taken atleast once)

$\therefore$ Cost of our approx soln $\leq$ 2 * cost of optimal solution.

This is a 2-approximation → called "2-approximation" as we are in range of factor of 2 to the optimal soln.

read ppt

Note: As $P \neq NP$, then For any constant $\rho \geq 1$, there is no polynomial time approximation algorithm with approximation ratio ρ for the general TSP.

explanation: Since NP problems cannot be solved in polynomial Time.
we develop algorithms to give approximate solutions to these problems
But we can never say that approximate algo will give answer in (k^{th} factor) of optimal answer.

IF we say so then basically we have proved P=NP.

PAGE NO.
DATE / /

AOA week 14 Notes.

As we cannot find exact answers for NP-complete problems, we will try to find approximate answers for such problems.

* Load Balancing problems
  # Greedy Approach.

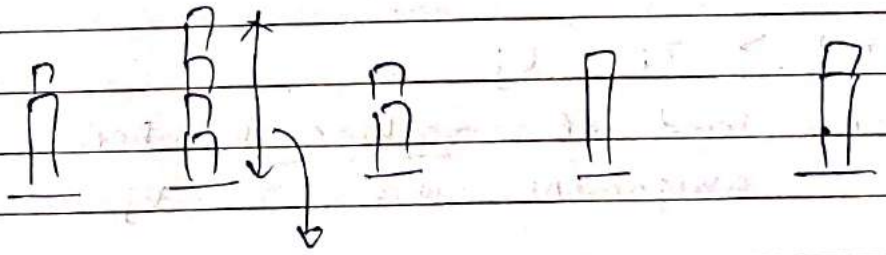
Input : 'm' resources with equal processing power

'n' jobs where job j takes tj to process

Objective: Assignment of jobs to resources such that the maximum load on any machine is minimized.

let Ti denote load on machine i

T* : value of the optimal solution.



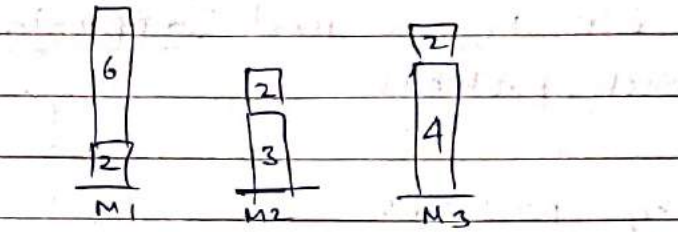
we have to minimize this

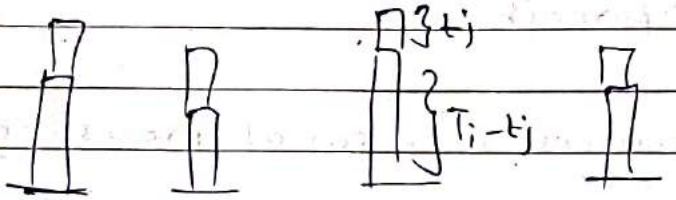
Greedy Balancing

Place the next job j with tj on machine that has least load currently.

PAGE No.
DATE / /

eg: Schedule 2, 3, 4, 6, 2, 2 on 3 machines





We know that

eq1 $T^* \geq t_j$ → load of any task j

load in optimal answer

Also if number of jobs n > number of machines 'm'

eq2 $T^* \geq T_i - t_j$

pg603 explanation: load of machine i before assignment was $T_i - t_j$

And the cost can only increase if we add a new task on same machine i it can never decrease.

$\therefore T^* \geq t_j$

$T^* \geq T_i - t_j$

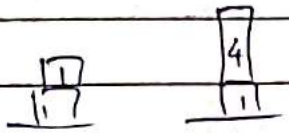
$2T^* \geq T_i$

$\boxed{T_i \leq 2T^*}$ 2-approximation.

PAGE No.
DATE / /

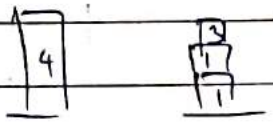
Consider a case

(1, 1, 1, 4) aligned on 2 machines



∴ T = 5

But we can have a better solution like



Here T = 4.

our previous greedy approach balances all resources among machines, and then a large resource comes and spoils the balancing

what can we do?

we can sort the resources in decreasing order and then apply greedy balancing

As first if we balance all large jobs, small jobs cannot cause much imbalance.

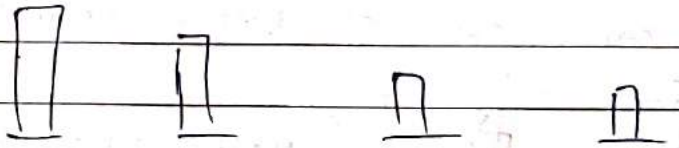
(#) Improved approximation to greedy Balancing

Initially sort jobs in dec order of length then use the same greedy balancing.

PAGE No.
DATE / /

if resources < machines
our solution is optimal.

if resources > machines



and $t_j \leq t_{m+1}$

$T^* \geq 2\, t_{m+1} \implies T^* \geq 2 t_j$ eq 1

$T^* \geq T_i - t_j$ eq 2

$T^* \geq 2\, T_j$

$2 \times \left(T^* \geq T_i - t_j\right)$

$3T^* \geq 2T_i$

$\therefore \boxed{T_i \leq 1.5\, T^*}$

1.5 approximation.

PAGE No.
DATE / /

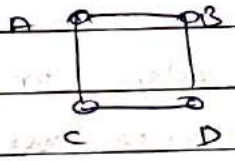
Vertex Covex Problem (Approximate Example)

problem statement: Find smallest vertex cover in G

Start with S = Null
while S is not a vertex cover:
select an edge e not covered by S
Add both ends of e to S

End while

example



Pick AB
S = {A, B}
covers edges AB, AC, BD

Pick CD
S = {A, B, C, D}
Covers all edges

2-approximate

An vertex cover has one end of each edge.

Our solution can have atmost two ends of each edge.

PAGE No.
DATE / /

Question

Since Indp set $\leq_p$ vertex cover

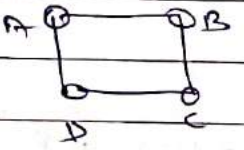
Can we use approximate soln of vertex cover to find a 5-approximate to indp set?

No

example:

we know Indp set = S

then vertex cover = V − S



running our algo on this gives vertex cover = {A, B, C, D}

∴ indp set = ϕ

Theorem

unless P = NP, there is no $\frac{1}{n^{1-\varepsilon}}$ approximation for the maximum independent set problem for any $\varepsilon > 0$ where n is the no. of nodes in the graph.

Question since vertex Cover $\leq_p$ set Cover

Can I use 2-approx algo for set cover to find a 2-approx for vertex cover?

Yes

PAGE NO.
DATE / /

MAX - 3SAT

Given a set of clauses of length 3, find a truth assignment that satisfies the largest number of clauses.

A. $\frac{7}{8}$-approximate to MAX-3SAT problem.

- set everything to true
  If less than 50% of clauses evaluate to true, then
- set everything to false

There are soln's that get $\frac{7}{8}$ factor of optimal solution.

(*) Linear equations:

$$[A][x] = [B]$$

linear programming

$$[A][x] \leq [B]$$

objective fnc: $[c^T][x]$

Goal: maximize the objective function subject to the above constraints

If $[A][x] \geq [B]$

The minimize the obj function.

PAGE No.
DATE / /

Pg 12
PPT

Linear programming standard Form

- All constraints are of Form (≤)
- All variables are non negative
- obj function is maximized.

eg: $x_1 - x_2 \geq 0 \quad x_1 \geq 0 \quad x_2 \geq 0 \quad x_1 + x_2 \leq 4$

maximize $x_1 + 2x_2$

(*) weighted vertex cover Problem

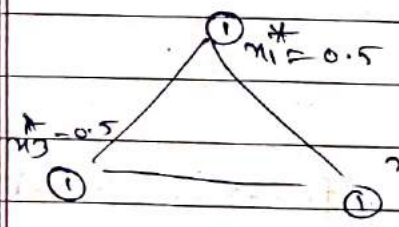
To find an approximate soln using LP, drop the requirement that $x_i \in \{0, 1\}$ and solve the LP in poly time to find $\{x_i^*\}$ between 0 & 1

$$w_{LP} = \sum_i w_i x_i^*$$

Assume $S' \leftarrow$ optimal vertex cover set

$w(S') \leftarrow$ weight of optimal solution

$$w_{LP} \leq w(S')$$



$w_{LP} = 1.5$

$w(S') = 2$

$x_i^* = 0 \longrightarrow i \notin S$

$x_i^* = 1 \longrightarrow i \in S$

$x_i^* + x_j^* \geq 1$

Say $S \geq 1/2 = \{ i \in V \; ; \; x_i^* \geq 1/2 \}$

$w(S) = 3.$

PAGE NO.
DATE / /

$w(S) \leq 2\, w_{LP}$

$w_{LP} \leq w(S')$

$w(S) \leq 2 \cdot w(S')$

2 - approx.

(*) Maxflow problem

Variables: Flow over edges

Maximize: $\sum_{e \text{ out of } s} f(e)$

Subject to:

$0 \leq f(e) \leq c_e$

$$\left(\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = 0 \right)$$

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

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

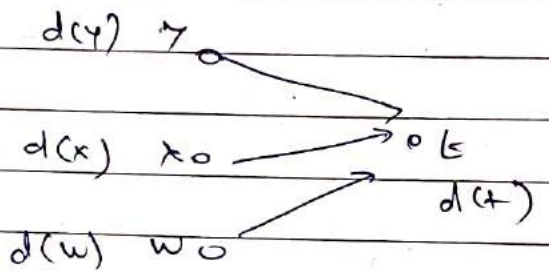
$A = B$ Then in LP $A \geq B$, $B \geq A$

PAGE No.
DATE / /

look ipt

(*) shortest path in LP:

Find shortest path from s to t



$$d(t) \le d(y) + c_{yt}$$
$$d(t) \le d(x) + c_{xt}$$
$$d(t) \le d(w) + c_{wt}$$

$$\begin{cases} d(v) \le d(u) + w(u,v) & \text{For each edge } (u,v) \in E \\ d(s) = 0 \end{cases}$$

objective

~~maxi~~ minimize $d(t)$

Maximize