Page No.
Date

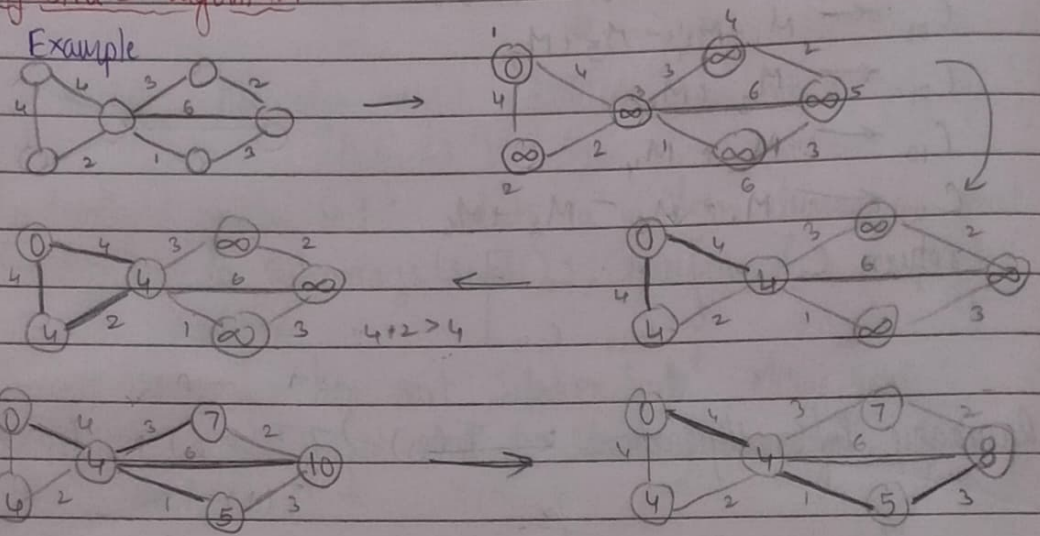
AA mod 3

- **Greedy Approach**
  Solve a problem by selecting the best option available at the moment
  Algo never reverses itself/earlier decision even if the choice is wrong
  Top down approach. continued later

- **Single Source Shortest Paths :**
  $G = (V, E)$ can be directed or undirected.
  $w : E \rightarrow R$: weight
  vertex : $s \in V$
  For all vertices $v \in V$ the minimum possible weight for path from s to v.
  Dijkstra's algo : positive edge weights only.

- **Dijkstra's algorithm**

Example



This works on basis that any subpath $B \rightarrow D$ of the shortest path $A \rightarrow D$ b/w vertices A and D is also the shortest path b/w B, D

```
Algorithm Dijkstra (Graph G, Vertex s);
    dist = array of distances, initialized to infinity
    dist[s] = 0                                          } initialize Θ(V)
    priority_queue = Min-Heap containing vertices, prioritized by dist[]
```

stores vertices
binary tree based DS
root has min value
priority determined by distance value. Min dist vertex will be at top of array

while priority_queue is not empty:
  u = dequeue vertex with minimum distance from priority_queue
  for each neighbor v of u:
    if dist[u] + weight(u,v) < dist[v]:
      dist[v] = dist[u] + weight(u,v)
      update priority_queue with new distance for v
return dist[]

} Relax $\Theta(1)$

Complexity:

$T_I(V,E) = \Theta(V)$: initialize single source. ← as vertex is set to infinity

$T_R(V,E) = \Theta(1)$: Relax

$T(V,E) = T_I(V,E) + \Theta(V) + V\Theta(\log V) + ET_R(V,E)$

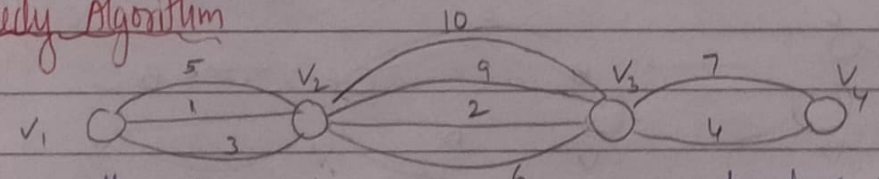
$= \Theta(V) + \Theta(V) + V\Theta(\log V) + E\,\Theta(1) = \Theta(E + V \log V)$

initialize ← extract min element from priority queue → relaxing edges

	1	2	3	4	5	6
1	0	4	4			
2	4	0	2			
3	4	2	0	3		1
4			3	0	2	
5				2	0	3
6			1		3	0

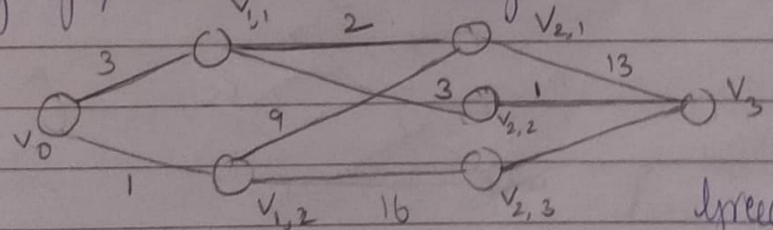
Diagonals are 0. as distance between the same point is always 0.
All entries not shown are $+\infty$
If the graph is directed,
ex. 2 → 1 but not 1 → 2
$A_{21} = 4$ $A_{12} = \infty$

- Greedy Algorithm



shortest path = 1 + 2 + 4 = 7 greedy = 1 + 2 + 4 = 7

Multi-stage graph = a node has many sub nodes

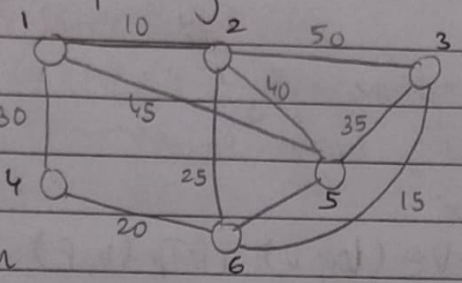


greedy: $V_0 V_{1,2} V_{2,1} V_3$
= 23 ≠ 7
Shortest: $V_0 V_{1,1} V_{2,2} V$
greedy doesn't work here

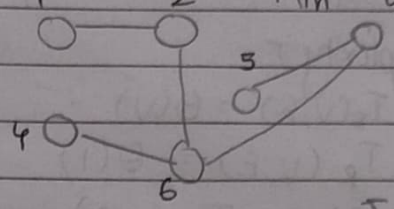
$d_{min}(i,j)$: distance b/w i and j (minimum)

$$d_{min}(V_0, V_3) = \min \begin{cases} 3 + d_{min}(V_{1,1}, V_3) \\ 1 + d_{min}(V_{1,2}, V_3) \end{cases}$$

Minimum Spanning Tree : Euclidean space points.
Is a spanning tree with the smallest total weight.



graph

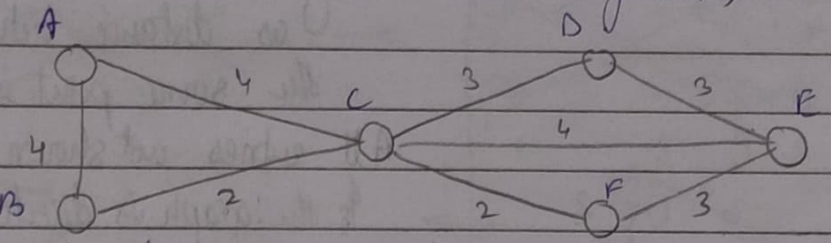


Min^m costs spanning tree

[I use this graph as example]

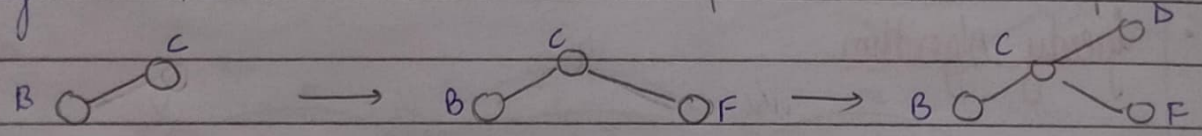
- **Kruskal's Algorithm [MST]**

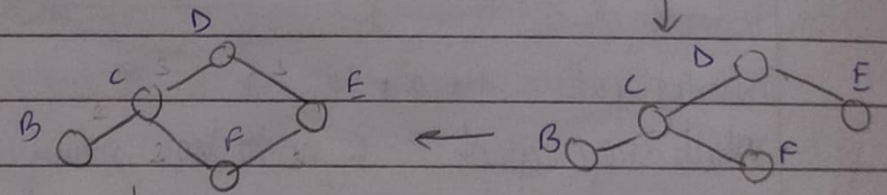
Takes a graph as input and finds the subset of edges of that graph which : form a tree that includes every vertex, has min^m sum of weights.



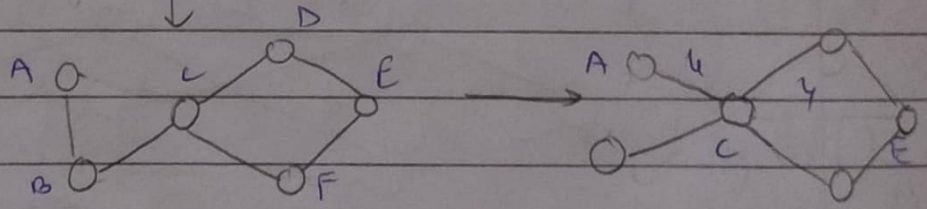
Weight of edges in ascending order :

Edge:	BC	CF	CD	DE	FE	AC	CE	AB
Weight:	2	2	3	3	3	4	4	4



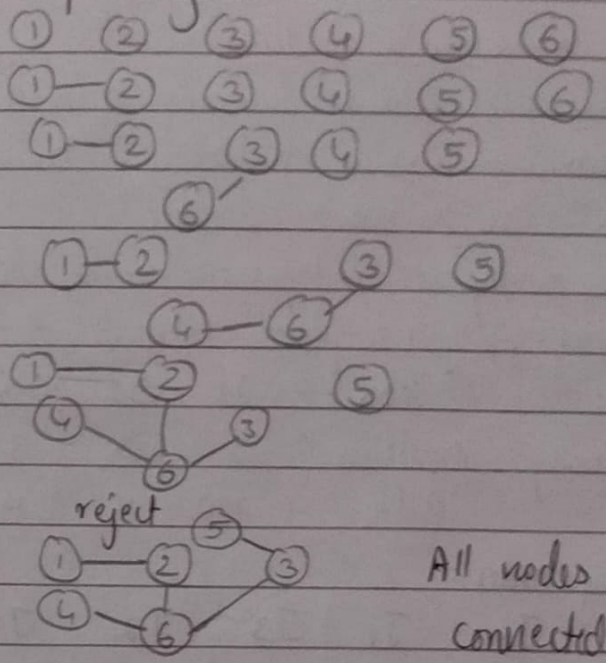


Cost : ~~4+4~~ 4+4
= AC + CE



[Trust the best examples refer next one]

Another Example :- [MST graph]

Edge	Cost	Spanning Forest
		
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	reject
(3,5)	35	

All nodes need to be connected.

Step 1 : Sort all edges in nondecreasing order
Step 2 : Add the next smallest weight edge to the forest if it won't cause a cycle.
Step 3 : Stop if n-1 edges . Otherwise go to step2

Time Complexity : $O(E \log E)$ → or $O(V \log V)$: step 1

proper algo :-

```
Kruskal (G) :
    T = {} // initialize MST
    Sort edges of G in non-decreasing order of weight in asc order
    for each edge (u,v) in sorted edges : //loop over edges
        if adding (u,v) to T doesn't create a cycle : //disjoint
            Add (u,v) to T
    return T
```

→

Job Sequencing

Job Sequencing with deadlines

Sort all given jobs in decreasing order of their profit.
Check maximum value of deadline. Draw Gantt Chart
(max^m time = max^m deadline)
Pick jobs one by one and put em on the chart as far as possible from 0 ensuring job gets completed before its deadline.

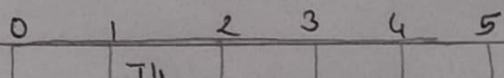
Example:

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

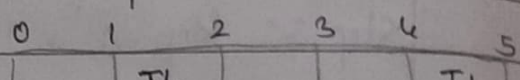
Decreasing order:-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

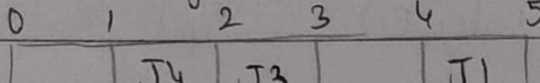
value of max^m deadline = 5, max^m time on Gnatt chart = 5



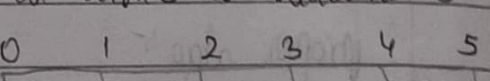
take J4, place it before deadline 2



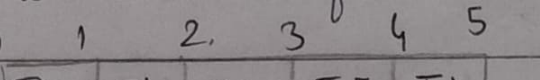
take J1, empty cell before 5



take J3, place it in empty cell before 3 as deadline = 3



J2 before empty cell of 1 as deadline = 3 and cells before 2,3 are filled.



J6's deadline is 2, but all slots are occupied, so J6 cannot be completed

Optimal schedule:-
J2, J4, J3, J5, J1 for max^m profit
All jobs aren't completed in the optimal schedule (J6)
Max^m profit = P2 + P4 + P3 + P5 + P1
= 180 + 300 + 190 + 120 + 200
= 990 units

Page No.
Date / /

Algorithm :-
1. Arrange all jobs in decreasing order of profit
2. For each Job (m,) do linear search to find particular slot in array of size (n), where n = max^m deadline, m = total jobs

Time complexity : $O(n^2)$

- Knapsack Problem [0/1 Knapsack]

↳ like a container or bag, in which we have to put items with weights and profits, which need to put in the knapsack in such a way that total value has max^m profit.

But, if we are greedy about weight, we choose lowest first.

Example :- Weights : {3, 4, 6, 5} max^m W = 8
Profits : {2, 3, 1, 4} n = 4

asc order W → (subsets)

i → item	0	1	2	3	4	5	6	7	8
P: W: 0	0	0	0	0	0	0	0	0	0
2 3 1	0	0	0	2	2	2	2	2	2
3 4 2	0	0	0	2	3	3	3	5	5
4 5 3	0	0	0	2	3	4	4	4	6
1 6 4	0	0	0	2	3	4	4	5	6 → max^m profit

4 - 4 = 0, check i=1 w=0

i=2 w=4 : max(3+0, 2) = 3

i=2 w=7 : max(3+2, 2) = 5

↳ 7-4=3, add value of i=1, w=3 with w.

i=4 w=6 : max(1+0, 4) = 4

m[i, w] = max(m[i-1, w], m[i-1, w-w[i]]+p[i])

Knapsack Algorithm :- (4+1 above, 8+1 above)

1. Create matrix K with dimensions (n+1) × (W+1), initialize all cells as 0, in the matrix

2 for i from 1 to n and for w from 1 to W (value from upper row)
if i's weight > current weight w, set K[i][w] as K[i-1][w]
else consider max^m of 2 cases:
including current item: value[i] + K[i-1][w - weight[i]]
excluding: K[i-1][w]
set K[i][w] to max^m of above cases.

3 Once table is filled, max^m value is K[n][W].
To find items which were selected, trace back from K[n][w] to K[1][1].
Include: if K[i][w] differs from K[i-1][w], where w is the remaining capacity after including i.

```
Pseudocode Knapsack(W, wt[], val[], n):
    Let K[0...n][0...W] with 0s
    for i from 0 to n:
        for w from 0 to W:
            if i=0 or w=0:
                K[i][w]=0
            else if wt[i-1] <= w:
                K[i][w]=max(val[i-1]+K[i-1][w-wt[i-1]],
                            K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
```

Time complexity: O(n × W)