UNIT-4

Greedy Method:

Feasible Soln :- A problem may consist of many solutions, but the solution which satisfy the constraint are known as feasible soln. (For a pblm we will have 1/more feasible soln)

Optimal Soln :- Solutions which are feasible and also satisfy the objective (best result) is known as optimal solution. (For a given pblm there can be only one optimal soln)

Greedy method is an algorithm designed technique used for solving optimization problem [which require either min (or) max result]

General Algorithm for Greedy Method:

```
Algm Greedy (a, n)
{
for i:=1 to n do
  {
    x=select(a);
    if feasible(x) then
    {
      soln=soln + x
    }
  }
```

Difference between greedy method and dynamic Programming:

Difference between greedy method and dynamic Programming

Greedy method	Dynamic programming
• It doesnot follow principle of optimality.	• It works based on principle of optimality.
• There is no memorization required	• Previous results are stored for finding solution.
• Space Complexity is less	• Space Complexity is more
• Final result is obtained as single sequence of solution	• Solution is obtained using multi stage process.
• Previous solution is not considered for solving current problem.	• Current problem is solved based upon previous output
• The final result may (or) may not be optimal	• Final result must be optimal always.

Applica

⇒ Greedy method says that problem should be solved in stages. In each stage, we have to consider input from the given problem. If that input is feasible, we will include it in soln. So, by combining all the feasible soln we get the optimal solution.

Camlin Page
Date / /

Applications:

- Job sequencing with dead-line.

Job	J_1	J_2	J_3	J_4	J_5
Profit	20	15	10	5	1
deadline	2	2	1	3	3

Pre-requisite:

→ There is only one processor to execute

→ Each process take same amount of time for completion (unit time)

→ No preemption is allowed.

On the above

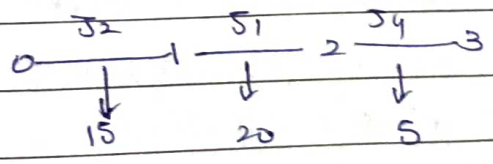
→ Arrange the jobs in the decreasing order of profits

⇒ On the above example, max deadline is 3. So, which means job 1 is ready to wait for 2 hours for attention of processor. Similarly job 2 also.

⇒ Job 3 is ready to waite only 1 hour

⇒ Job 4 and 5 are ready to waite for 3 hours

Job	Slot allocated	Sequence	Profit
J_1	[1, 2]	J_1	20
J_2	[0, 1] [1, 2]	J_2, J_1	15 + 20
J_3 (discard J_3)	[0, 1] [1, 2]	J_2, J_1	15 + 20
J_4	[0, 1] [1, 2] [2, 3]	J_2, J_1, J_4	15 + 20 + 5
J_5 (discard J_5)	[0, 1] [1, 2] [2, 3]	J_2, J_1, J_4	40.



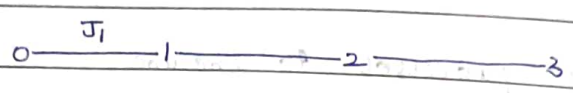
$J_2 - J_1 - J_4$

Camlin Page
Date / /

Consider Jobs J_1, J_2, J_3

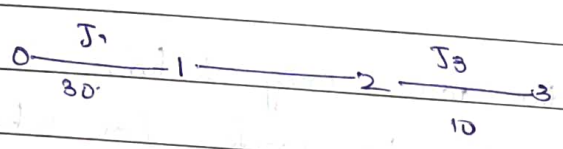
Job	J_1	J_2	J_3
Profit	30	20	10
deadline	2	1	3

In this problem J_1 is ready to waite for 2 hrs. If we allocate J_1 in the first slot itself. The sequence is as follows

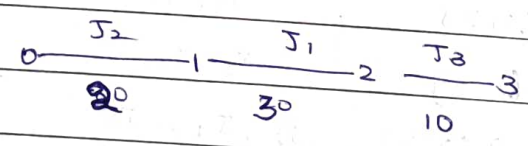


If we consider next input J_2 whose deadline is 1 But the first slot is already been allocated for Job_1, so, J_2 cannot be taken into account.

J_3 is ready to waite till 3 hrs. So the allocation can be either at 1-2 or 2-3



If we try to schedule the job as late as possible (but within the deadline). We can even include Job 2 ie.



max profit. 60

Algorithm

→ Sort the job in decreasing order of profit

→ Initialize the result sequence as the first from the sorted list (We will try to schedule the job as late as possible)

→ Do the following for remaining n-1 jobs

If the current job can fit in the current result sequence without missing the deadline add current job to result. Else discard the current job.

n = 7

Job	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	25	30	15	35	20	5	10
deadline	3	4	4	2	3	1	2

Arrange them in decreasing order

Job	J_4	J_2	J_1	J_5	J_3	J_7	J_6
Profit	35	30	25	20	15	10	5
deadline	2	4	3	3	4	2	1

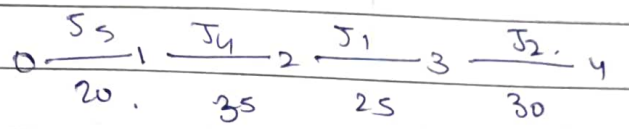
0 — J_5 (20) — 1 — J_4 (35) — 2 — J_1 (25) — 3 — J_2 (30) — 4

Job	Slot allocated	Sequence	Profit
J_1	[2,3]	J_1	25
J_2	[2,3] [3,4]	J_1, J_2	30+25
J_3 discard.	[2,3] [3,4]	J_1, J_2	30+25
J_4	[1,2] [2,3] [3,4]	J_4, J_1, J_2	25+30+35
J_5	[0,1] [1,2] [2,3] [3,4]	J_5, J_4, J_1, J_2	25+30+35+20.
J_6 discard J_6	[0,1] [1,2] [2,3] [3,4]	J_5, J_4, J_1, J_2	25+30+35+20
J_7 discard J_7	[0,1] [1,2] [2,3] [3,4]	J_5, J_4, J_1, J_2	25+30+35+20. = 110.

Max profit = 110

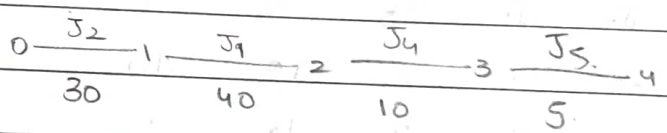
0 — J_5 — 1 — J_4 — 2 — J_1 — 3 — J_2 — 4

Job	Slot allocated	Sequence	Profit.
J_4	[1, 2]	J_4	35.
J_2	[1, 2] [3, 4]	J_4, J_2	35 + 30.
J_1	[1, 2] [3, 4] [2, 3]	J_4, J_1, J_2	35 + 30 + 25
J_5	[1, 2] [3, 4] [2, 3] [0, 1]	J_5, J_4, J_1, J_2	35 + 30 + 25 + 20
J_3 discard J_3	[1, 2] [3, 4] [2, 3] [0, 1]	J_5, J_4, J_1, J_2	35 + 30 + 25 + 20
J_7 discard J_7	[1, 2] [3, 4] [2, 3] [0, 1]	J_5, J_4, J_1, J_2	35 + 30 + 25 + 20
J_6 discard J_6	[1, 2] [3, 4] [2, 3] [0, 1]	J_5, J_4, J_1, J_2	35 + 30 + 25 + 20 = 110



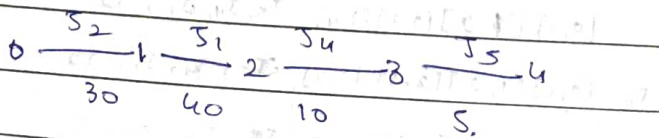
$n = 5$

Job	J_1	J_2	J_3	J_4	J_5
Profit	40	30	12	10	5
deadline	2	2	1	3	4



Job	Slot allocated	Sequence	Profit.
J_1	[1, 2]	J_1	40
J_2	[0, 1] [1, 2]	J_2, J_1	30 + 40
J_3 discard	[0, 1] [1, 2]	J_2, J_1	30 + 40.
J_4	[0, 1] [1, 2] [2, 3]	J_2, J_1, J_4	30 + 40 + 10.
J_5	[0, 1] [1, 2] [2, 3] [3, 4]	J_2, J_1, J_4, J_5	30 + 40 + 10 + 5

Max profit = 85.



FRACTIONAL KNAP SACK PROBLEM (0-1 knap sack using greedy method):

Problem Statement:

Including objects in bag, which gives max profit

Constraint: Total weight of the objects consider should not be greater than the weight of the bag [$\sum x_i w_i \leq m$] m = size of bag.

Objective:

$\sum x_i p_i$ should be maximum.

n = 7

m

w = 15

o :	1	2	3	4	5	6	7
p :	5	10	15	7	8	9	4
w :	1	3	5	4	1	3	2
p/w :	5	3.3	3	1.75	8	3	2

De Including the obj in decreasing order of profit

Object	Profit	Weight	Remaining weight (total size of bag given m = 15)
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
4	$7 \times \frac{3}{4} = 5.2$	3/4	3 - 3 = 0
	47.2		[Size of bag = 3 but for Obj 4 w = 4 So we are considering 3/4ths out of 4]

$x_1, x_2, x_3, x_4, x_5, x_6, x_7$

$x(0, 1, 1, \frac{3}{4}, 1, 1, 0)$

$\sum x_i w_i =$

* Including object based on weight [Lesser weight items first]

Object	Profit	weight	Remaing weight
1	5	1	15-1 = 14
5	8	1	14-1 = 13
7	4	2	13-2 = 11
2	10	3	11-3 = 8
6	9	3	8-3 = 5
4	7	4	5-4 = 1
3	$15 \times \frac{1}{5}$	1..	0.
	= 3		
	46		

$x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7$

$x(1, 1, 1/5, 1, 1, 1, 1)$ 5 — 5, ? — 1

$\sum x_i w_i = 1\times1 + 1\times3 + 1/5\times5 + 1\times4 + 1\times1 + 1\times3 + 1\times2 = 1+3+1+4+1+3+2 = 15$

* Based on Profit /weight ratio

Object	Profit	weight	Remaing weight
5	8	1	15-1 = 14
1	5	1	14-1 = 13
2	10.	3	13-3 = 10
3	15	5	10-5 = 5.
6.	9	3	5-3 = 2
7	4	2	2-2 = 0.
	51		

Max profit we will get in P/w ratio problem

$x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7$

$x(1, 1, 1, 0, 1, 1, 1)$

$\sum x_i w_i = 1\times1 + 1\times3 + 1\times5 + 0\times4 + 1\times1 + 1\times3 + 1\times2$

$= 1+3+5+1+3+2$

$= 15$ ⇒

$x_i P_i = 1\times5 + 1\times10 + 1\times15 + 0\times7 + 1\times8 + 1\times9 + 1\times4$

$= 5+10+15+0+8+9+4 = 51$ ⇒

Camlin Page
Date / /

Algorithm for Fractional knap-sack

```
Algo Greedyknapsack (m, n)
{
// m represent total size of the bag
// n represent no. of objects
// w[1:n] weight array which hold weight of n objects
// p[1:n] profit array hold profit of n object.
// u - hold updated weight of the bag.
// x[1:n] resultant array which hold included object.
for i:1 to n
        x[i]=0 // initialize x array with zero
u=m
for i:1 to n do
    {
     if (w[i]>0) then
        break;
     else
        x[i]=1 , u=u-w[i] // including the item complete
    }
    if (i<=n) then
        x[i]=u/w[i] // including fraction of item.
}// end.
```

6/6/22

m=12

m = 12

n = 6

O:	1	2	3	4	5	6
P:	10	5	15	7	6	18
W:	2	3	5	7	1	4
P/w:	5	1.6	3	1	6	4.5

profit

Object	Profit	weight	Remaining weight
6	18	4	12 - 4 = 8
3	15	5	8 - 5 = 3
1	10	2	3 - 2 = 1
4	$7 \times \frac{1}{7}$	1	1 - 1 = 0
5			
2	8. 44		

$x(1, 0, 1, \frac{1}{7}, 0, 1)$

weight

Object	Profit	weight	Remaining weight
5	6	1	12 - 1 = 11
1	10	2	11 - 2 = 9
2	5	3	9 - 3 = 6
6	18	4	6 - 4 = 2
3	$15 \times \frac{2}{5}$	2	
	45		

$x(1, 1, \frac{2}{5}, 0, 1, 1)$

S = 15

15

Object	Profit	weight	Remaining weight
5	6	1	12 - 1 = 11
1	10	2	11 - 2 = 9
6	18	4	9 - 4 = 5
3	15	5	5 - 5 = 0.
	49		

$x(1, 0, 1, 0, 1, 1)$

Application - 3.

Minimum Cost Spanning Tree :

Spanning Tree :

Spanning Tree is a subgraph of a graph which connect all the vertices of graph with min. no. of edges

Mathematically we can represent as, $S \subseteq G$

$S \rightarrow$ Subset of G

$G \rightarrow G$ is given graph $\quad S \subseteq G$

$S = (V', E')$

$V' = V.$

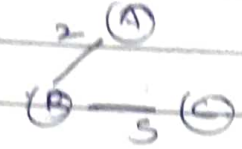
$E' = |V| - 1$

A Complete undirected graph max we can have n^{n-2} no. of spanning tree. [$n \rightarrow$ no. of vertices]

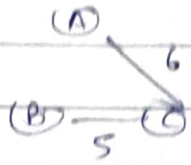
ex: Suppose given graph G is 

Draw the graph

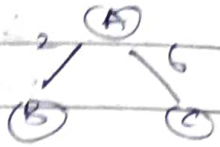
Possible spanning tree:



Cost = 7



Cost = 11



Cost = 8

In this ex, n = 3 [3 vertices]

$3^{3-2} = 3$ max spanning trees.

Properties of Spanning Tree:

1) No. of vertices in spanning tree is same as no. of vertices in given graph.

2) No. of edges is [v - 1]

3) Spanning Tree doesnot include cycles.

4) If we remove one edge from spanning tree, tree will be disconnected because spanning tree is minimally connected graph.

5) Adding one edge to the spanning tree form a cycle

Mainly 2 algorithms to find min. cost spanning tree.

1) Prim's algorithm
2) Kruskal's algorithm.

Camlin Page
Date / /

Prim's Algorithm

Procedure: (Pseudocode)

1) Remove all the loops and parallel edges [Remove the parallel edge having highest cost]

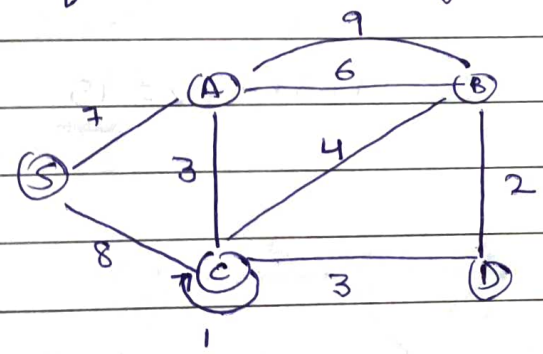
2) Choose any one vertex as starting node.

3) Check the outgoing edges from starting vertex and select one with minimum cost.

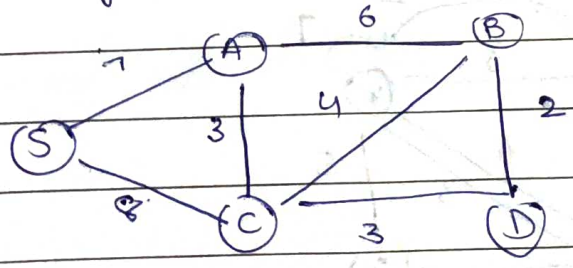
4) Repeat step 5 till all the vertices are included in resultant spanning tree.

5) Select the min cost edge which is already connected with selected edge. [During selection of an edge if it cost a cycle, we should avoid that edge]

(1) ex:-

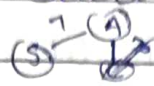


By avoiding the self loop and parallel edge, the resultant graph will be



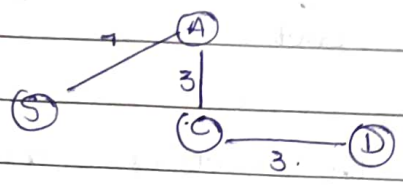
Let us consider S as starting node.

* 2 outgoing edges from S are S-A [cost = 7] S-C [cost = 8]. Select min. cost edge ie, S-A.

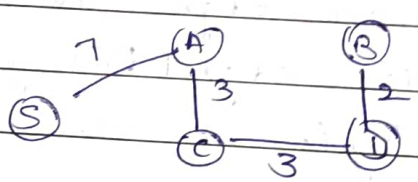


* Findout min outgoing cost edge from ~~A~~ 2S, ~~A-C~~. towards unvisited vertices

* Findout min outgoing cost edge from vertex C, A, S C-D(3), C-A(3), C-S(8). The vertex A, S are visited we will Consider C-D(3).



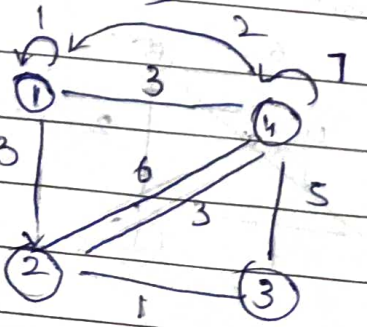
From D, min outgoing edge is D-B(2)



The cost of the tree is 7+3+3+2 = 15

(2).

	1	2	3	4
1	1	3	∞	2
2	∞	∞	1	6
3	∞	∞	∞	5
4	2	3	∞	7

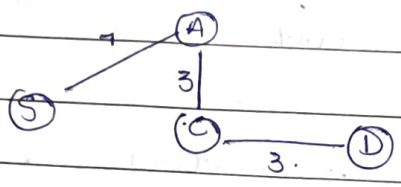


Let us Consider S as starting node.

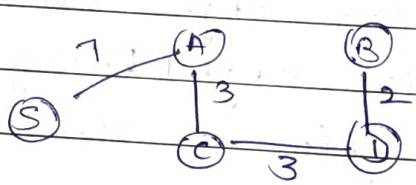
* 2 outgoing edges from S are S-A [Cost = 7] S-C [Cost = 8]. Select min. Cost edge ie., S-A.

* Findout min outgoing Cost edge from ~~A~~ S, ~~A-C~~ towards unvisited vertices

* Findout min outgoing Cost edge from vertex C, A, S C-D(3), C-A(3), C-S(8). The vertex A, S are visited Lee will Consider C-D(3).



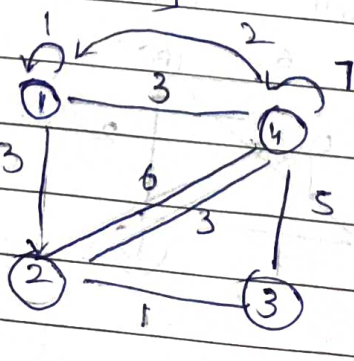
From D, min outgoing edge is D-B(2)

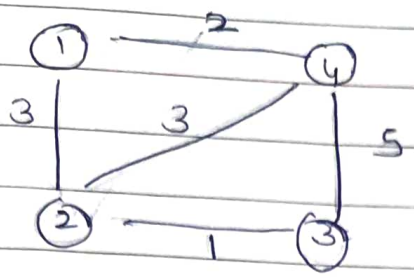


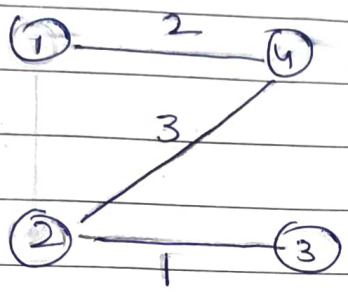
The Cost of the tree is 7+3+3+2 = 15

(2).

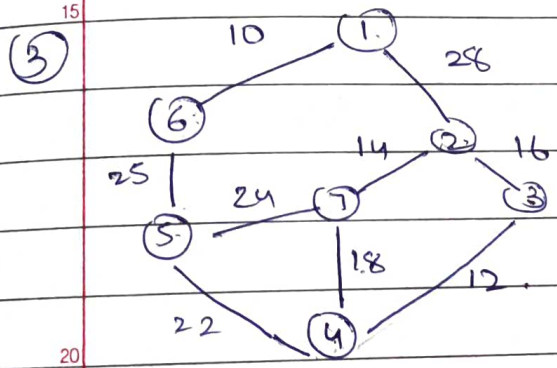
	1	2	3	4
1	1	3	∞	2
2	∞	∞	1	6
3	∞	∞	∞	5
4	2	3	∞	7

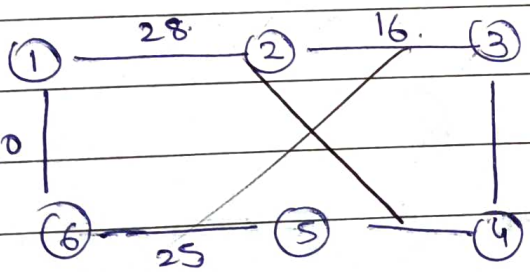


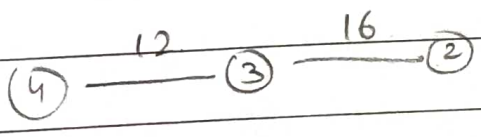


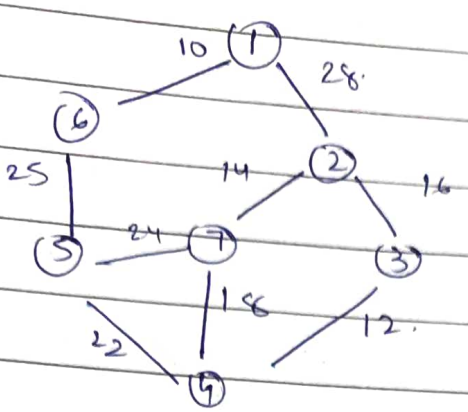


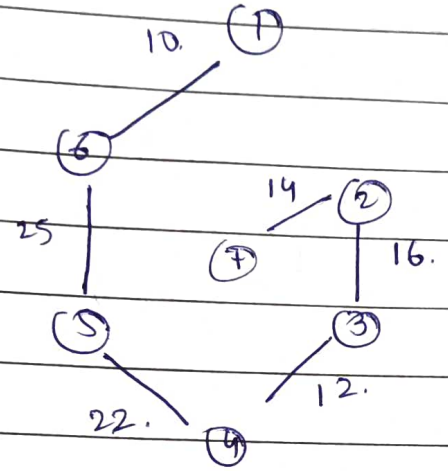
Min Cost = 2+3+1 = 6











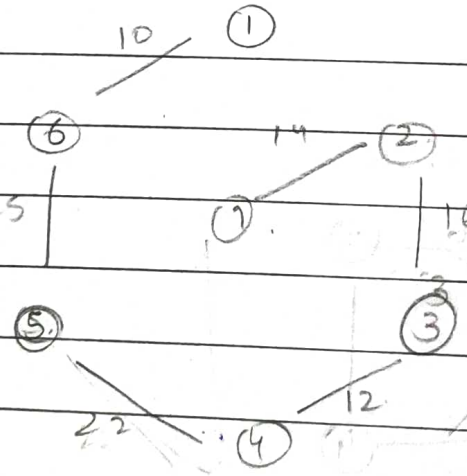
12
16
18
22

$= 10 + 25 + 14 + 16 + 12 + 22$

$= 99$

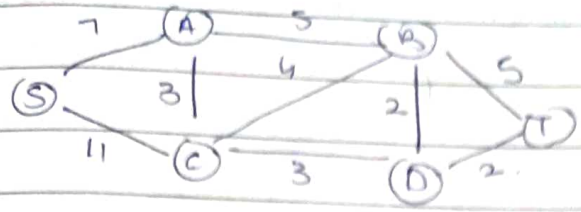
35.

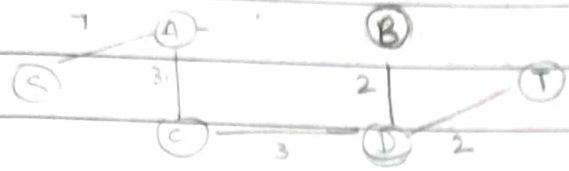
65
34



28
14
42
35
67
10
77
22
99

3)

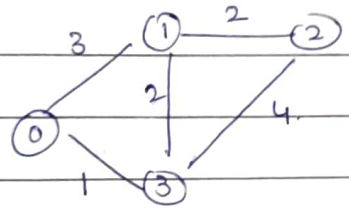




= 17

Implementation of Prim's algorithm:

Consider the given graph,



	0	1	2	3
0	∞	3	∞	1
1	3	∞	2	2
2	∞	2	∞	4
3	1	2	4	∞

Initialize parent array with -1

0	1	2	3
-1	-1	-1	-1

Initialize weight array with ∞.

0	1	2	3
∞	∞	∞	∞

Initialize visited array with false.

0	1	2	3
F	F	F	F

Select arbitarily any vertex as starting vertex.
Let us Consider 0^{th} vertex as starting. wt[0] = 0
v[0] = T

Consider all the neighbours of the vertex whose visiting value is 0 and update the weight array if this edge give a minimum value.

	0	1	2	3
Parent	-1	0	-1	0

	0	1	2	3
weight	0	3	∞	1

Now, identify the vertex with min. weight whose visited value is False (from weight array)

v = 3
v[3] = True.

Repeat the above step with vertex 3.

	0	1	2	3
Parent	-1	3	3	0

	0	1	2	3
weight	0	2	4	1

Next vertex to be visited is vertex 1
v[1] = True.

	0	1	2	3
Parent	-1	3	1	0

	0	1	2	3
weight	0	2	2	1

Next vertex to be visited is 2.

V[2] = T

All the neighbours of 2 is already visited, So no updation required.

We will get output edges from weight array

edge

root of 1st vertex is 3 — weight

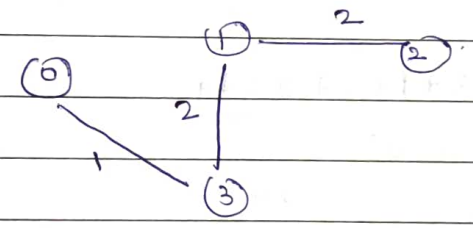
3 — 1 — weight of 1 is 2

root of 2nd vertex is 2. — weight of 2 is 2

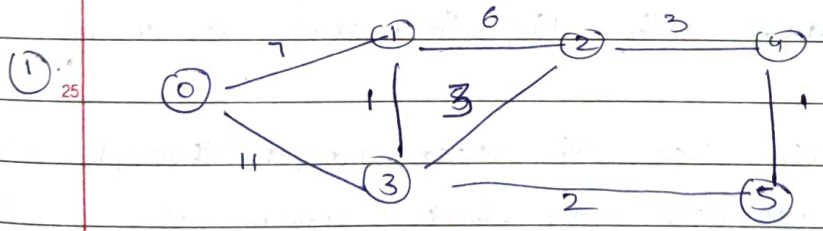
1 — 2

root of 3rd vertex is 0 — weight of 3 is 1.

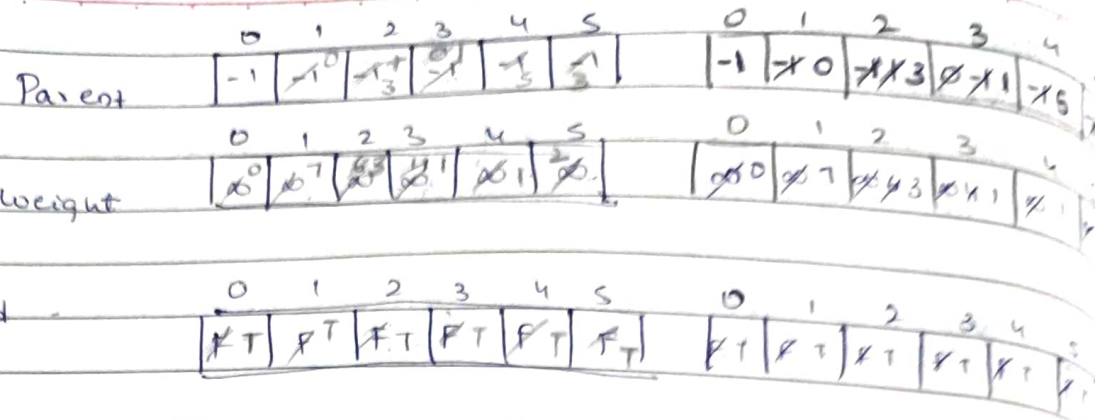
0 — 3



2 + 2 + 1 = 5



	0	1	2	3	4	5
0		7		11		
1	7		6	1		
2	·	6		3	3	
3	11	1	3			2
4		3	3			1
5			2	1		



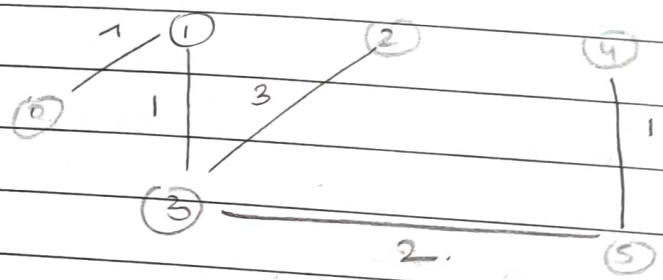
0 → 1 → 7

3 → 2 → 3

1 → 3 → 1

5 → 4 → 1

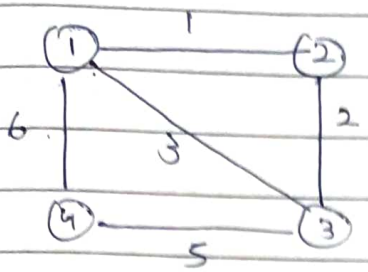
3 → 5 → 2



Cost: 7 + 3 + 1 + 1 + 2 = 14.

Kruskal's Algorithm:

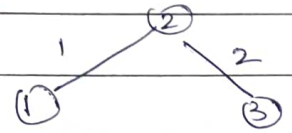
1) Remove all the loops and parallel edges.
2) Arrange all the edges in the increasing order of weight
3) Add the edge which has the least weight (through out this process, we need to check where the Spanning tree property remained impact or not ie, if inclusion of an edge causes a cycle we should not consider that edge in the Spanning tree.



Edges in increasing order of weight

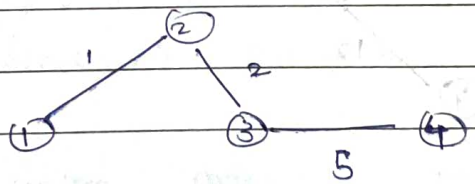
	order of weight
1 — 2	(1)
2 — 3	(2)
1 — 3	(3)
3 — 4	(5)
4 — 5	(6)

Include the edge 1 — 2 into the spanning tree.
Include the next min cost edge ie, 2 — 3

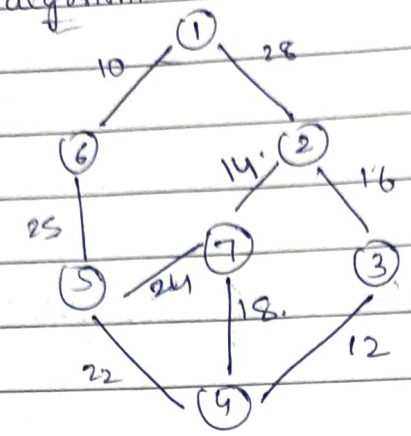


Next min. cost edge is 1 — 3 but inclusion of this edge will cause a cycle. 1 — 3 neglected (x)

Next included edge is 3 — 4. The process is repeated until we include (v-1) edges into the spanning tree.

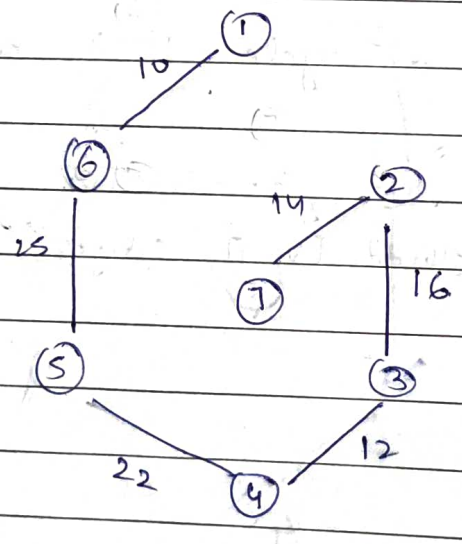


Kruskal's algorithm:

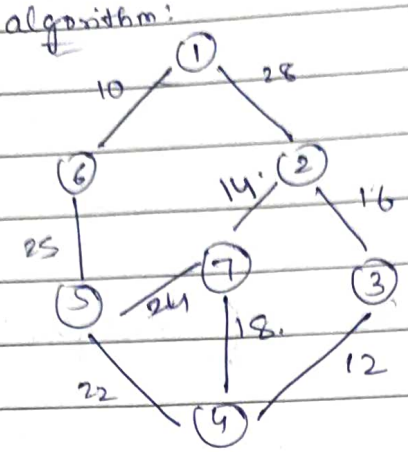


6 — 1	10
~~3 — 4~~	12
2 — 3	16
7 — 4	18
4 — 5	22
~~5 — 7~~	~~24~~
~~5 — 6~~	~~25~~
1 — 2	28

6 — 1	10
4 — 3	12
2 — 7	14
2 — 3	16
4 — 7	18 x
4 — 5	22
5 — 7	24 x
5 — 6	25
1 — 2	28 x

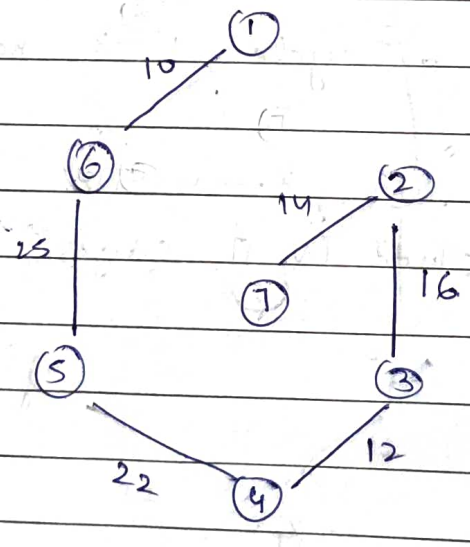


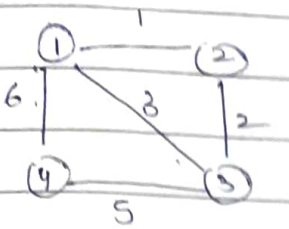
Kruskal's algorithm:



6 — 1	10.
~~3 — 4~~	12.
2 — 3	16.
7 — 4	18.
4 — 5	22
~~5 — 7~~	~~24~~
~~5 — 6~~	~~25~~
1 — 2	28.

6 — 1	10
4 — 3	12
2 — 7	14
2 — 3	16
4 — 7	18 x
4 — 5	22
5 — 7	24 x
5 — 6	25
1 — 2	28 x





	1	2	3	4
	999			
1	0	1	3	6
		999		999
2	1	0	2	0
			999	
3	3	2	0	5
4	6	0	5	0
		999		999

→ Since we need to find min cost edge, replace 0 with higher integer value (let us say 999)

→ Findout the min. weighted edge from the vertex.
In this ex, weight 1 (i, j) = (1, 2)

u = i , v = j

u = 1 , v = 2

→ Perform find operation on u & v to find the parent node. Initialize the parent array with 0.

1	2	3	4
0	0	0	0

Parent (1) & ie., find(1). P(u) = 1

P(2) ie., find(2). P(u) = 2

→ Perform union operation on Pu and Pv.

union(1, 2)

```
union Alg (i, j){
  if (i ≠ j)
  {
    p[j] = i ;
    return 1;
  }
```

else {

return 0;

}

union(1, 2)

1 ≠ 2

P[2] = 1

	1	2	3	4
Parent	0	1	0	0

function returns the value 1

union fn will return value 1 if there is no cycle. Otherwise, it returns 0. Based on this value, an edge will be included in the resultant tree.

① —— ②

And the weight is Cost[i][j] ie, = 2.

We include a count cnt which is incremented whenever we include an edge into spanning tree.

cnt = 1.

→ Since this min. Cost edge is already processed, set Cost[i][j] and [j][i] as 999. [Since we need to find next min cost edge in each iteration

→ Next min edge is between (2, 3) u = 2, v = 3 then

P_u = find(2) ie, 1

P_v = find(3) ie, 0

union(1, 3)

1 ≠ 3.

P[3] = 1

	1	2	3	4
parent	0	1	1	0

union fn returns the value 1.

Camlin Page
Date / /

cnt = 2

Include edge (2,3) & set cost of [2][3], [3][2] = 999

Count is incremented to 2.

→ u=1, v=3.

Pu = find(1) = 0

Pv = find(3) = 1

union(1,1)

i and j are equal

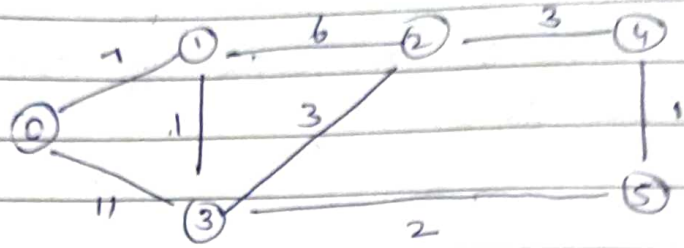
union to return a value 0.

Didnot include 1,3 coz it forms cycle.

Cnt = 2.

Cost 5 b/w 4 — 3.

u=4, v=3.



	0	1	2	3	4	5
0	0	7	0	11	0	0
1	7	0	0	999	0	0
2	0	0	0	999	0	0
3	11	999	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

⇒ A

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

1 →

$(i, j) = (1, 3)$

$u = 1, v = 3$

find (1) = 1 [∵ i ≠ j]

find (3) = 3

union fn returns value 1 - 3

P[j] = i

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

→ parent

$(i, j) = (3, 5)$

$u = 3, v = 5$

find (3) = 1

find (5) = 5

union fn P[j] = i

P[5] = 1

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

A[i,i] = A[j,j] = 999

P(i,j) = (2,3)

u = 2, v = 3

find(2) = 2

find(3) = 1

union → P[3] =

min(A)

1 min for i = 4, j = 5

u = i

v = j

pu = find(i) → find(4) | 4 = pu

pv = find(j) → find(5) | 5 = pv }

```
find(i)
{
 while(p[i]>0)
 {
   i = p[i];
 }
 return i;
}
```

5

union(pu, pv)

union(4,5)

```
{
 if(pu != pv)
 {
   parent[pv] = pu;
   return 1;
 }
 return 0;
}
```

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

parent

union(4,5)

replace, A[i,j] = A[j,i] = 999 inorder to find next minimum value.

⇒ min(A)

2 is minimum for i = 3 & j = 5

u = i

v = j

u = 3, v = 5

find(3) = 1, find(5) = 4

Union P[4] = 3

pu = find(3) ⇒ 1

pv = find(5) = 4

union(1,4)

```
{ if(1!=4)
 { parent(4) = 1;
   return 1
 }
}
```

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

A[3,5] = A[5,3] = 999

min(A) = 3 for i = 2 & j = 3

u = i ⇒ 2 ; pu = find(2) ⇒ 2

v = j ⇒ 3 ; pv = find(3) ⇒ 1

```
union(2,1)
{
   if (2 != 1)
   {
      parent[1] = 2;
      return 1;
   }
   return 0;
}
```

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

A[2,3] = A[3,2] = 999

min(A) = 3 for i = 2 & j = 4

u = i ⇒ 2 ; pu = find(2) ⇒ 2

v = j ⇒ 4 ; pv = find(4) ⇒ ~~8~~ 2

```
find(i)
{
   while (p[i] > 0)
   {
      i = p[i]
   }
   return i;
}
```

p[3] > 0 p[1] > 0

i = 3 i = p[3], i = 2 i = p[1], i = 2

union(2,4) return 0;

replace A[2,4] = A[4,2] = 999

Camlin Page
Date / /

Min(A) = 6 for i = 1 & j = 2

u = i = 1 ; pu = find(i) ⇒ find(1) = 2

v = j ⇒ 2 ; pv = find(j) ⇒ find(2) = 2

```
find(i)        p[i] > 0
{                2 > 0
   while(p[i] > 0)
   {  i = 2;
   }
   return i;
}
```

A[1,2] = A[2,1] = 999

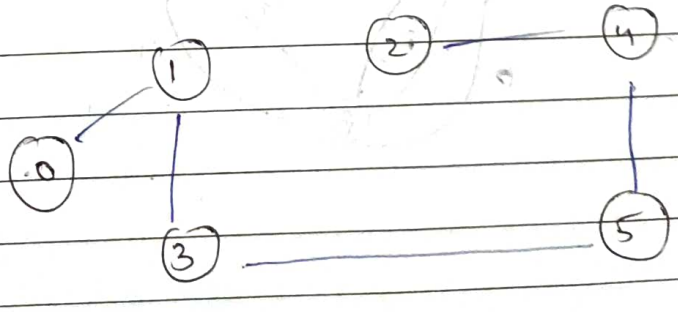
min(A) = 7 for i = 0 & j = 1

u = 0 ; pu = find(0) → 0

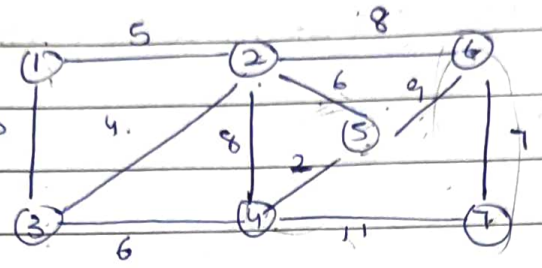
v = 1 ; pv = find(1) ⇒ 2

```
union(0,2)
{
   if(0 != 2)
   {
      parent[0] = 2;
      return 1
   }
   return 0
}
```

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



Optimal randomized algorithm:-



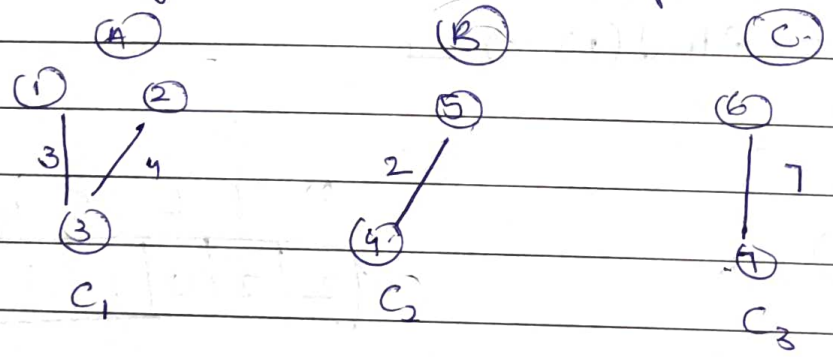
Boruvka Step

In this step for each node an incident edge with min weight is choosen.

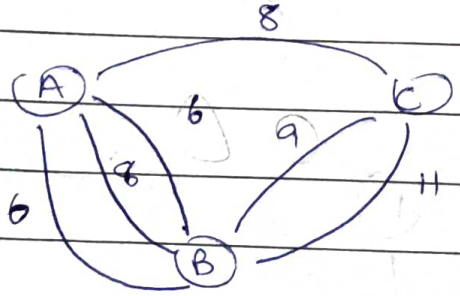
Edge 1—3, 2—3 as component 1

Edge 4—5 as component 2

Edge 6—7 as component 3.

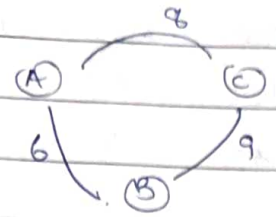


Now, identify the edges connecting diff components



Camlin Page
Date / /

Select the minimum cost edge.



Select A - B (6)
A - C (8)

Application - 4

Activity Selection Problem:

Given 'n' activities with their start and finish time. Select max no. of activities that can be performed by a single processor [Assuming the processor can perform one task at a time.

Algorithm:

1) Sort the activities according to their finishing time.
2) Select the 1st activity from the sorted array and print it.
3) Perform the below step for remaining activities in the sorted array
→ If start time of activity ≥ The finish time of previously selected activity then select the activity and print it.

Problem:

Ex:

Activity	A_1	A_2	A_3	A_4	A_5	A_6
Starting time (St)	0	3	1	5	5	8
f_i :	6	4	2	9	7	9

Activity :	A_3	A_2	A_1	A_5	A_4	A_6
St :	1	3	0	5	5	8
f_i :	2	4	6	7	9	9

Select the 1st activity ie., A_3. Since, the starting time of the next activity is greater than finishing time of A_3, we can select A_2.

Since starting time of next activity is less than finishy time of recently selected activity ie., A_2. We cannot include A_1.

Selected activity

A_3
↓
A_2
↓
A_5
↓
A_6

Starting time of next activity ie, A_5 is greater than finishing time of A_2. We can select A_5.

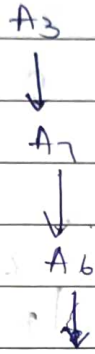
Starting time of next activity ie, A_4 is less than finishy time of activity A_5. So, we cant select A_4.

Starting time of A_6 greater than finishy time of activity A_5. So, we can include A_6.

Camlin Page
Date / /

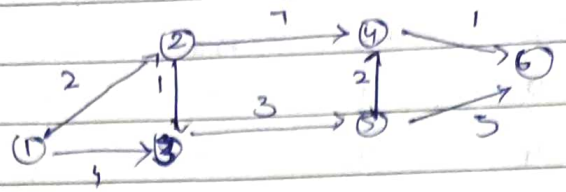
2) Activity | A_1 | A_2 | A_3 | A_4 | A_5 | A_6 | A_7
St : 1 0 1 4 2 5 3
fi : 3 4 2 6 9 8 5.

Activity : A_3 | A_1 | A_2 | A_7 | A_4 | A_6 | A_5
St : 1 1 0 3 4 5 2.
fi : 2 3 4 5 6 8 9



Application - 5:
Single Source Shortest path [Dijkstra's algorithm]

→ This algorithm helps to find shortest path between the nodes in a graph from a particular vertex. It is based on principle of relaxation. i.e., An approximation of the correct distance is gradually replaced by more accurate values until shortest distance is reached.



	1	2	3	4	5	6
1	0	2	4	0	0	0
2	0	0	1	7	0	0
3	0	0	0	0	3	0
4	0	0	0	0	0	1
5	0	0	0	2	0	5
6	0	0	0	0	0	0

Initialize select array as False. for all the vertices.

	1	2	3	4	5	6
Select	F	F	F	F	F	F

Initialize a distance array for all vertices as ∞
Let us consider starting vertex $v = 1$

∞	∞	∞	∞	∞	∞

Let us Consider Starting vertex $v = 1$
Selected of 1 make as True; distance (1) = 0.

	1	2	3	4	5	6
	T	F	F	F	F	F

1	2	3	4	5	6
0	∞	∞	∞	∞	∞

Select the neighbours of vertex 1 and update cost towards those neighbours in the distance array

1	2	3	4	5	6
0	2	4	∞	∞	∞

Find min distance vertex whose selected value is not True. So, vertex is 2.

S(2) = T

1	2	3	4	5	6
0	2	~~4~~ 3	~~9~~ 9	∞	∞

neighbours 3, 4

min edge vertex is 3

S(3) = T

1	2	3	4	5	6
0	2	3	9	~~9~~ 6	∞

neighbour 5
3 + 3

min edge vertex is 5.

S(5) = T

1	2	3	4	5	6
0	2	3	~~9~~ 8	6	~~∞~~ 11

neighbour 4, 6
6+2 6+

min edge vertex is 4

S(4) = T

neighbour 6.
8+1 =

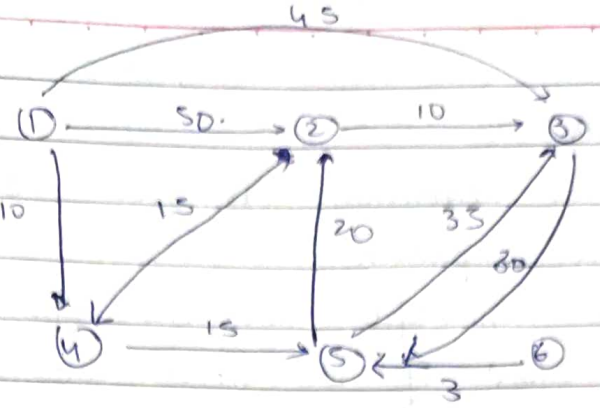
1	2	3	4	5	6
0	2	3	8	6	~~11~~ 9

min edge vertex is 6

S(6) = T

1	2	3	4	5	6
0	2	3	8	6	9

```
Algorithm Shortest path (v, Cost, distance, n)
{
// Distance[j] is set to length to shortest path from
   vertex v to vertex j in a graph G with n vertices
5
  for (i=1 to n){
  do
  {
    S[i] = False
    distance[j] = Cost[v, i]
  }
}
  S[v] = True
  distance[v] = 0
for (x=2 to n)
do {
// determine n-1 path from 'v'
  Choose u from those vertices which are not selected
  and distance of u is minimum //
20
  S[u] = true
  for each w adjacent to u with S[w] is False.
  do {
  if (distance[w] > distance[u] + Cost[u, w]) then
  distance[w] = distance[u] + Cost(u, w)
25
  }.
}
30
```



	1	2	3	4	5	6
1	0	50	45	10	0	0
2	0	0	10	15	0	0
3	0	0	0	0	30	0
4	0	0	0	0	15	0
5	0	20	35	0	0	0
6	0	0	0	0	3	0

Select array

1	2	3	4	5	6
T	F	F	F	F	F

distance

1	2	3	4	5	6
0	∞	∞	∞	∞	∞

S(1) = T.

1	2	3	4	5	6
0	50	45	10	∞	∞

S(4) = T

1	2	3	4	5	6
0	50	45	10	25	∞

S(5) = T

1	2	3	4	5	6
0	45	45	10	25	∞

2, 3

20 35

50 45

S(2) = T

1	2	3	4	5	6
0	45	45	10	25	∞

3, 4

10 15

25 50

S(3) = T

1	2	3	4	5	6
0	45	45	10	25	∞

5 → 30

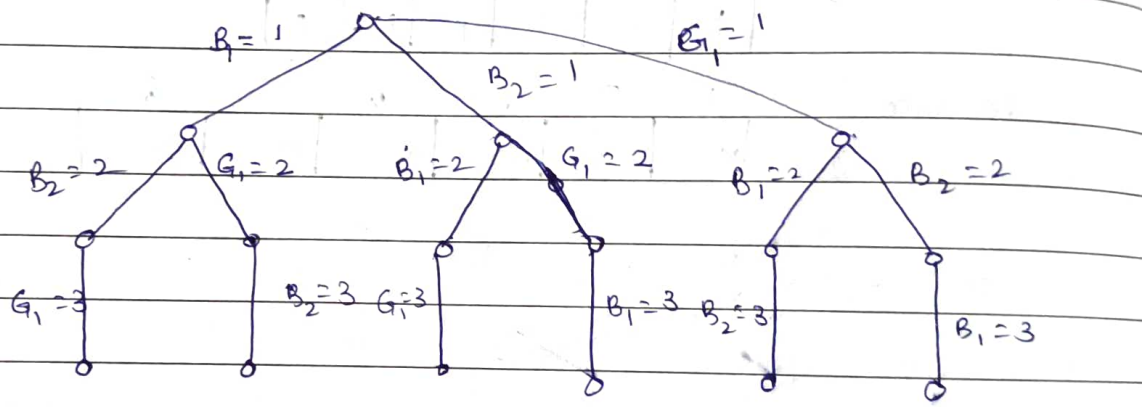
BACKTRACKING:

It is an algorithm designed strategy which uses Bruteforce approach.

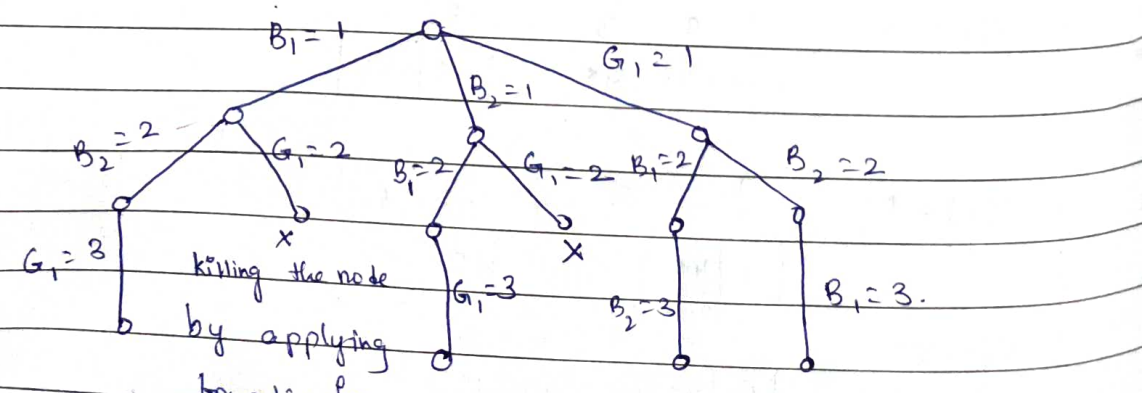
→ If we have multiple solutions for given problem and all the solns if we need we use this approach.

→ Output of back tracking problem is represented as n-tuple

Consider 3 students B_1, B_2, G_1. We need to allocate seat for these students when 3 chairs are given. We can arange them in 3! ways. The state space tree for this problem is



If we want to arange these 3 students such that G_1 shouldnot sit in middle, then the solution tree will be



So, 4 possible solutions can be represent as tuple.

B_1 B_2 G_1

(1, 2, 3)

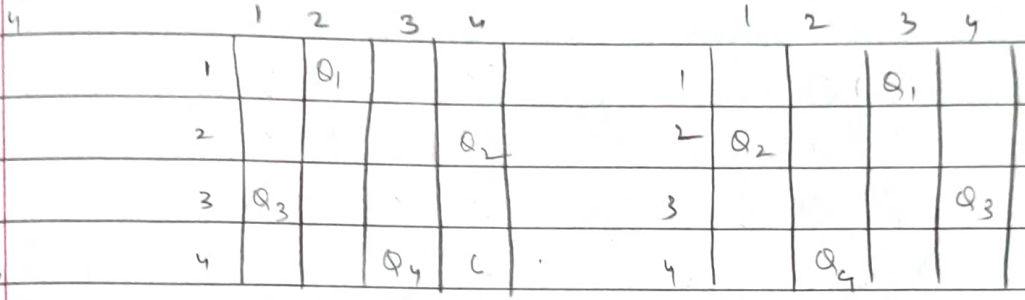
(2, 1, 3)

(2, 3, 1)

(3, 2, 1)

Applications:

1) n-queen problem.



Q_1 Q_2 Q_3 Q_4

[2 4 1 3] ←

[3 1 4 2]

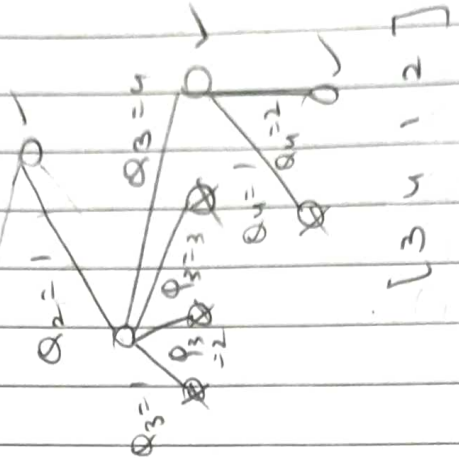
We will be representing the output as n-tuple which hold col. numbers for the respective queen. The above example is 4-queen problem. So, the output is

Q_1 Q_2 Q_3 Q_4

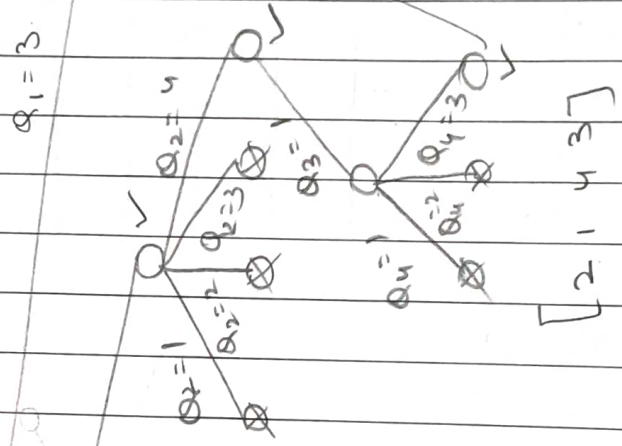
Sol-1 [2 4 1 3] ←

Sol-2 [3 1 4 2]

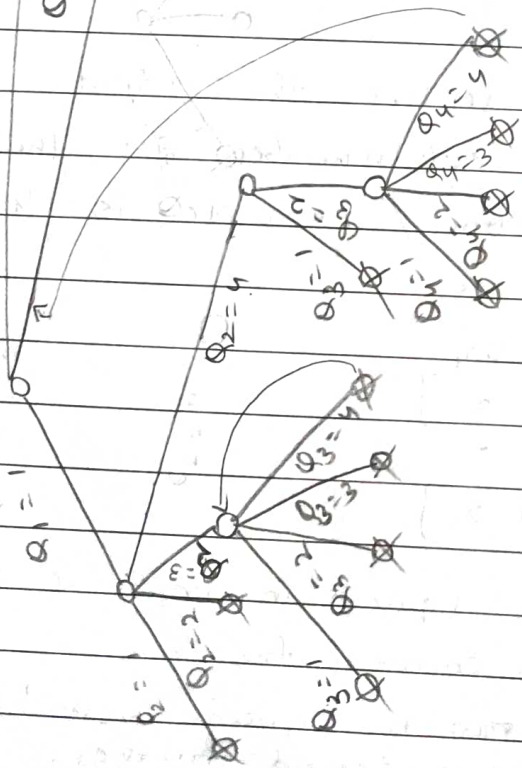
NOTE: We are considering 1 queen in 1 row. i.e., Q_1 1st row, Q_2 2nd row, ---. The two country fns are no two queens in same column, no two queens must be diagonal. If this fn is true we will not extend further.



[3 4 1 2]



[2 1 4 3]



The Soln hold the column no. of each queen. So, the tuple is [2, 4, 1, 3].

Similarly it will try for $Q_1 = 3$ which leads to a Solution [3, 1, 4, 2] and also $Q_1 = 4$ which will not give desired output.

→ In n-queen problem, in the above example we have considered n = 4. So, in 4×4 chess board we need to place, 4 queens in such a way that no two queens are under attack. [ie., no two queens should be placed in same row (or) same column (or) diagonal]

Algorithm for n queen problem:
There is two algorithms

```
Algm NQueen(k, n)
{
for i = 1 to n do // for each row for a queen all n cols
{                    are checked
if(place(k, i)) then
{
   x[k] = i
if (k == n) then write x[1 : n]
else
      NQueen(k+1, n)
} // if
} // for
} // Algm
```

```
Algm place(k, i)
{
for j:=1 to k-1 do //checking the Constraint with all the
{                    Queens possible placed in board.
if (x[j]==i) or Abs(x[j]-i)==Abs(j-k))
{
return false;
} //if
} //for
return True;
} //fn.
```

Let us Consider n=4. So, first time NQueen(1, 4), k represent the queen number, i represent Column number. The x[i, n] represent the output array which hold the ~~output~~ of respective queens. column no.

→ Inside this we call the fn place (1, 1) Place fn will return a value True, if the queen is not under attack.

→ X[j]=i is the Condition for checking whether 2 queens are in the same column.

→ Abs (x[j]-i) == Abs (j-k) is the Condition to check 2 queens are diagonal.

→ If we are at 'k' queen, we need to check the Constraint with all the queens which are already placed in the board.

→ Place(1, 1) will return a value True, so we update x[1] as 1.

1	2	3	4
1	0	0	0

Since k ≠ n, we again call the function n-queen

n-queen(2,4)

Place(2,1)

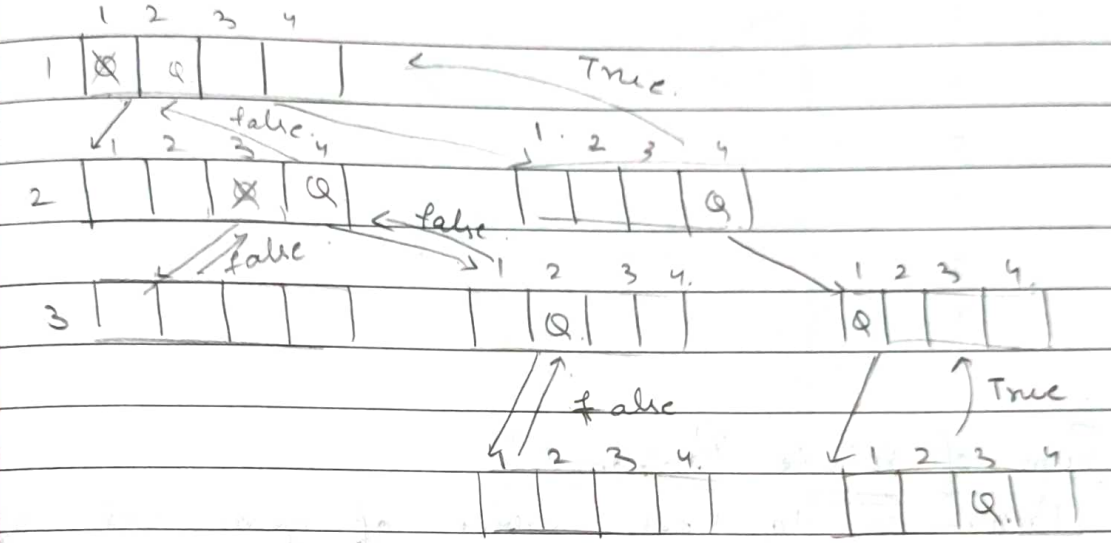
return false (∵ Column Constraint is True)

Place(2,2) (∵ Col Constraint is true)

return false

place(2,3)

X[2] = 3



* General Backtracking Algorithm:

Algm Backtrack(k)

{

for each x[k] ∈ [T(x[1], x[2] ---- x[k-1])] {

then

{

if B_k (bounding fn.) (x[1], x[2], ---- x[k-1] is true

then

{

if $x_1, x_2, \ldots x_k$ is a path to ans node then

write x[1:n]

}

if k<n

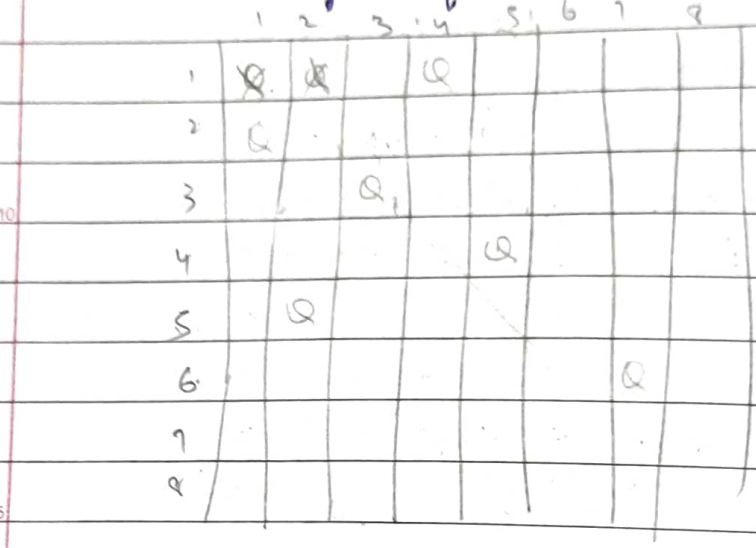
then Backtrack(k+1)

} // if

} // for

} // Algm

Time Complexity of N-Queen pblm is exponential.



Sum of Subsets Problem:

Problem is to find subset of given set of 'n' objects for $S = \{S_1, S_2, \ldots S_n\}$. We need to find Sum of these integers = a give value m.

We consider three parameters s is zero initially otherwise,

$$\sum_{i=1}^{n} w_i x_i$$

k takes value 1 to n

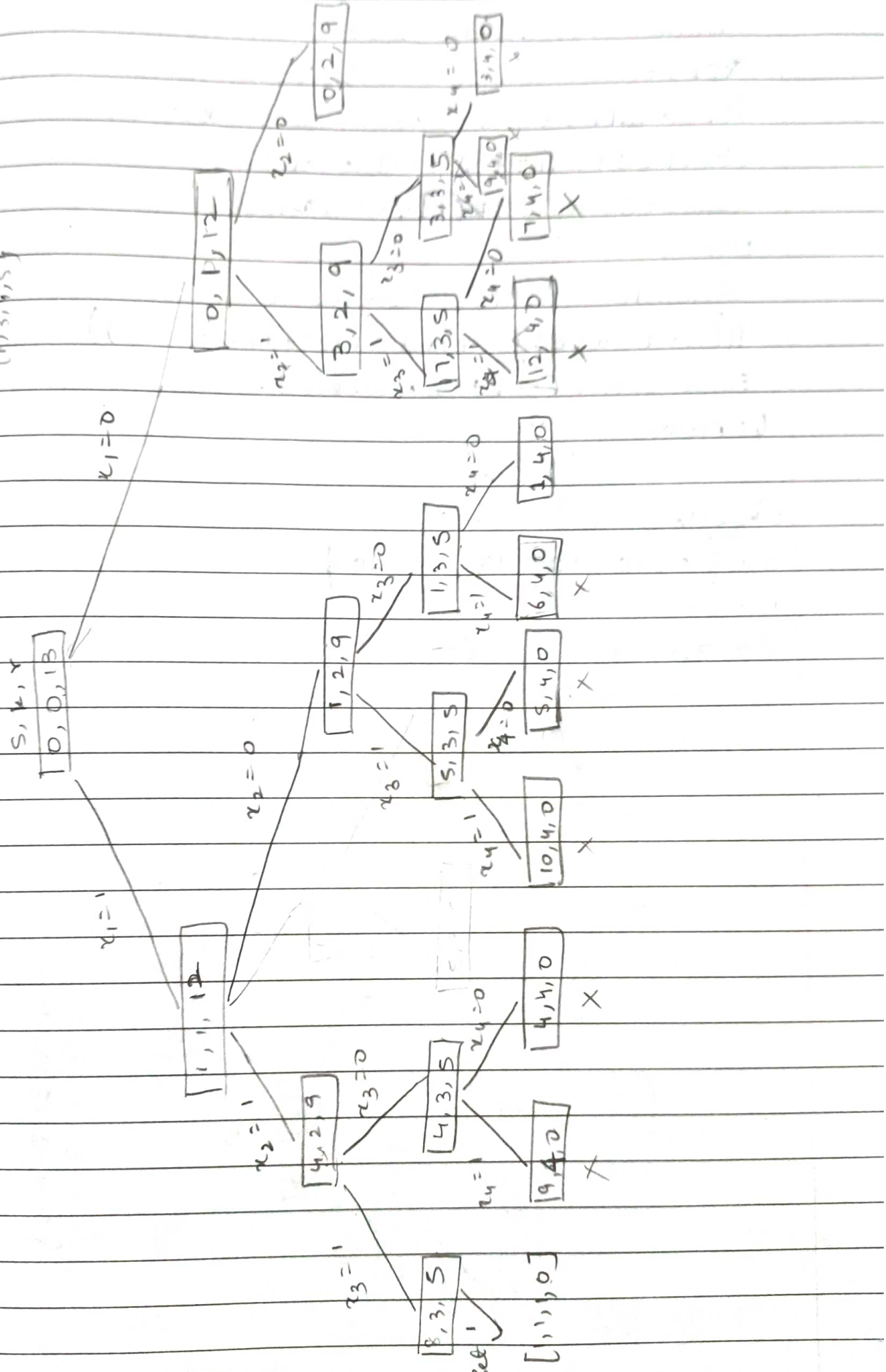
r initially it represent the total weight ie,

$$\sum_{i=1}^{n} w_i$$

In each level r value varies $i = k+1$ to n.

$$\sum_{i=k+1}^{n} w_i \text{ (remaing weight)}$$

Set S = {1, 3, 4, 5}, B = 8, n = 4.



Algorithm SumofSubset(S, k, r)

{

// Generate left child

x[k] = 1

if(s + w[k] == m) then write x[1:n]

else if (s + w[k] + w[k+1] <= m)

SumofSubset(s+w[k], k+1, r-w[k])

else

// Generate right Subtree

if((s + r - w[k] >= m) and s + w[k+1] <= m))

SumofSubset(s, k+1, r-w[k])

} // Algm.

Example

S = { 5, 10, 12, 13, 15, 18 }

n = 6

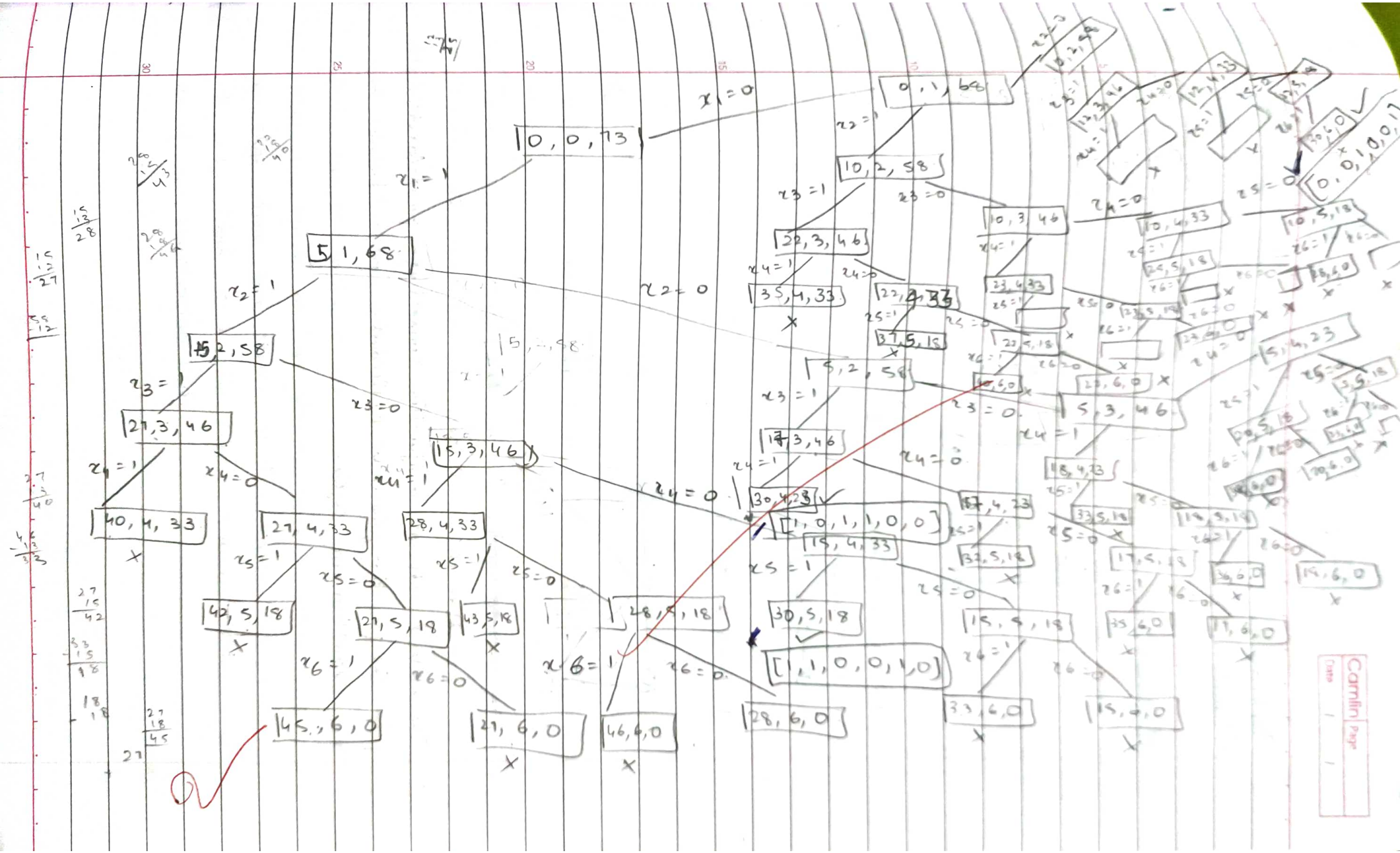
m = 30

s = 0 + 5 = 5 < 30

s = 5 + 10 = 15 < 30

s = 15 + 12 = 27 < 30

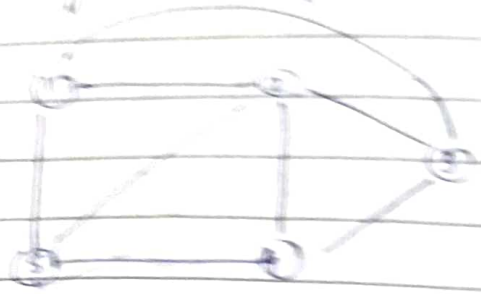
s = 27 + 13 = 40 > 30


0, 0, 73
x1 = 0
x1 = 1
1, 1, 68
x2 = 1
x2 = 0
2, 2, 58
x3 = 1
x3 = 0
3, 3, 46
x4 = 1
x4 = 0
4, 4, 33
x5 = 1
x5 = 0
5, 5, 18
x6 = 1
x6 = 0
6, 6, 0

Application 3

Hamiltonian cycle

It is defined as cycle that passes through all the vertices of graph exactly once & we end at starting vertex.



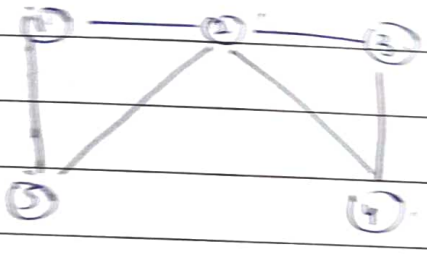
1-2-3-4-5-1

1-5-4-3-2-1

1-3-4-2-5-1

1-3-2-4-5-1

1-3-4-2-5-1



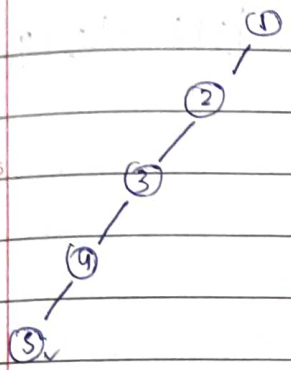
~~1-5-2-4-3~~

no Hamiltonian cycle.

- The node 2 acts as a junction connecting point (articulation point). Graph that include articulation point doesn't have any hamiltonian cycle.

Initialize the X array with Zero.
Consider 1 as the starting vertex Set X[1] as 1.
Mainly Three bounding functions
1) Each vertex should not take duplicate value
2) There should present an edge from current vertex to previous vertex
3) When we reach the last vertex check whether it is connecting to the starting vertex. If connected, then X array hold a hamiltonian cycle.

1	2	3	4	5
1	2	3	4	5



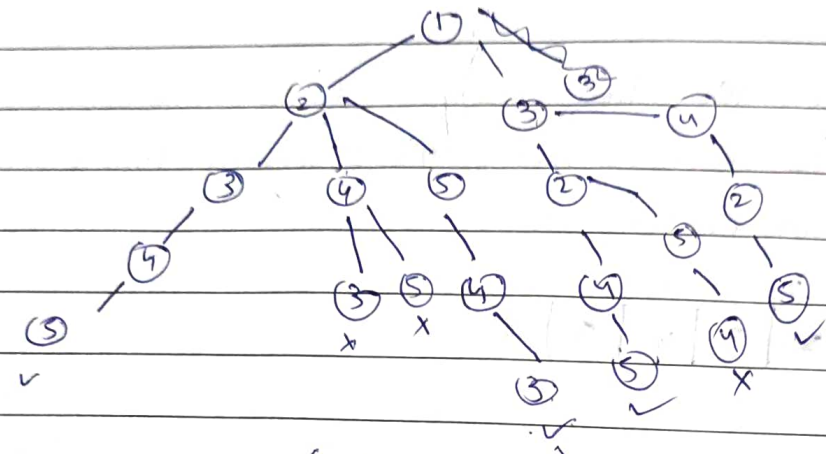
→ X[2] cannot take value 1 since 1 is already present in the array. So, X[2] = 2.

→ There exist an edge between 1 to 2. can be included into solution set. Move to next vertex

→ X[3] cannot take value 1, 2. Since already present in array. So, X[3] = 3. Check whether 2 to 3 edge is present. If exist then can be included into solution set and so, on.

→ When we reach at last vertex check whether there is an edge from last vertex to first vertex. Since, there is an edge from 5 - 1 the current X array value leads to one of the Hamiltonian cycle.

Now set x[5]=0 and back track to vertex 4. So, the other possible value for x[4] = 5. But there is no edge from previous vertex 3 to current vertex 5. So, set x[4]=0 and go back vertex 3.



(cont'd min 4).

1	2	3	4	5
1	2	3	4	3	0

3. 2 4 5.

1	2	3	4	5
1	3	2	4	5	0

5.

1	2	3	4	5
1	3	2	5	4	0

4. 2. 5.

```
Algorithm Hamiltonian(k)
{
 Repeat
 {
 Next value of [k] (Next value [k])
 if x[k] == 0 then
 return
 if k == n then
 write x[1:n]
 else
 Hamiltonian (k+1)
 } until (false)
 Algm Next value (k)
 {
 repeat
 {
 x[k] = (x[k] + 1) mod (n+1) // placing values for x[k]
 if (x[k] == 0) then return 1 ex: if n=5, then possible array
 if (G(x[k-1], x[k]) != 0) then values are 1, 2, 3, 4, 5, 0
 {
 for j = 1 to k-1 do
 if (x[j] == x[k])
 if (j == k) then
 if ((k < n) or (k == n) and G(x[n], x[1]) != 0)
 then return 1
 }
 } until (false)
```

x is the array which hold the result G is the graph (input graph)

If starting vertex (then x[1] is set to 1 and remaining all values set to zero initially

GREEDY METHOD

Greedy method

General algorithm a control structure / abstraction diff between.

Application:

→ Job Sequencing with deadline
Complexity $O(n^2)$

→ Fractional knapsack
Complexity $O(n^2)$ worst case

→ Minimum cost Spanning tree
* prim's algm - $O(n^2)$
* kruskal's algm - $O(n^2)$

→ Optimal Randomised Algorithm

→ Activity Selection - $O(n^2)$

→ Single Source shortest path (Dijkstra's) - $O(n^2)$

BACK-TRACKING

↳ should be explained using algm & statespace tree.

→ General method

→ Applications (Time Complexity is exponential)

→ N-Queen problem

→ Sum of Subsets

→ Hamiltonian Cycle.