ALGORITHM DESIGN AND ANALYSIS

UNIT III

GREEDY AND DYNAMIC PROGRAMMING

INTRODUCTION - GREEDY - HUFFMAN CODING - KNAPSACK PROBLEM - MINIMUM SPANNING TREE (KRUSKALS ALGORITHM) - INTRODUCTION - DYNAMIC PROGRAMMING - 0/1 KNAPSACK PROBLEM - TRAVELLING SALESMAN PROBLEM - MULTISTAGE GRAPH - FORWARD PATH AND BACKWARD PATH

3.1 INTRODUCTION TO GREEDY PROGRAMMING

- An optimization problem is one which finds just not a solution, but the best solution
- A greedy algorithm sometimes works well for optimization problems
- A Greedy algorithm works in phases. At each phase
  - (i) Take the best right now, without regard for future consequences
  - (ii) Hope that by choosing a local optimum at each step, it will end up at a global optimum
- Application of the Greedy Strategy
  - (i) Huffman codes
  - (ii) Minimum spanning Tree (MST)
  - (iii) Knapsack Problem
  - (iv) Simple Scheduling Problems
  - (v) Combinatorial optimization Problems
  - (vi) Single - source shortest paths

3.2 HUFFMAN CODING

- Luseless Data compression Algorithm
- IDEA :- To assign variable - length codes to input characters
    - Lengths of the assigned codes are based on the frequencies of the corresponding characters
    - The most frequent character gets the smallest code
    - The least frequent character gets the largest code

INFORMATION ENCODING

GOAL: To transmit information in the fewest bits possible in such a way that each encoding is unambigous

FIXED LENGTH ENCODING: Encoding for each symbol has the same number of bits

A	B	C	D
00	01	10	11

NOTE:- There are cases when the same binary string encodes different messages

VARIABLE LENGTH ENCODING: Symbols can be encoded with different number of bits

A	B	C	D
000	1	110	1111

NOTE:- Variable length encoding avoids ambiguity

TERMINOLOGIES

(i) MIN HEAP

- Min heap is a binary tree such that
  - The data contained in each node is less than (or equal to) the data in that node's children
  - The binary tree is complete

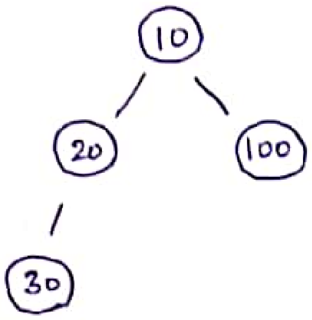


Fig (a)

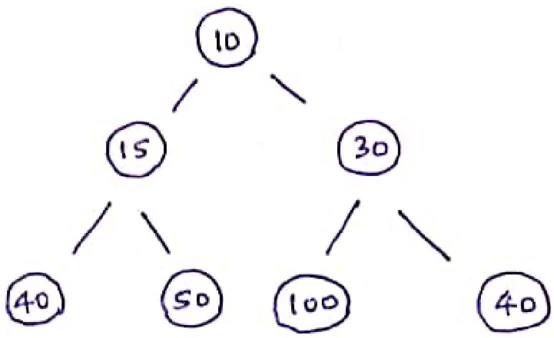
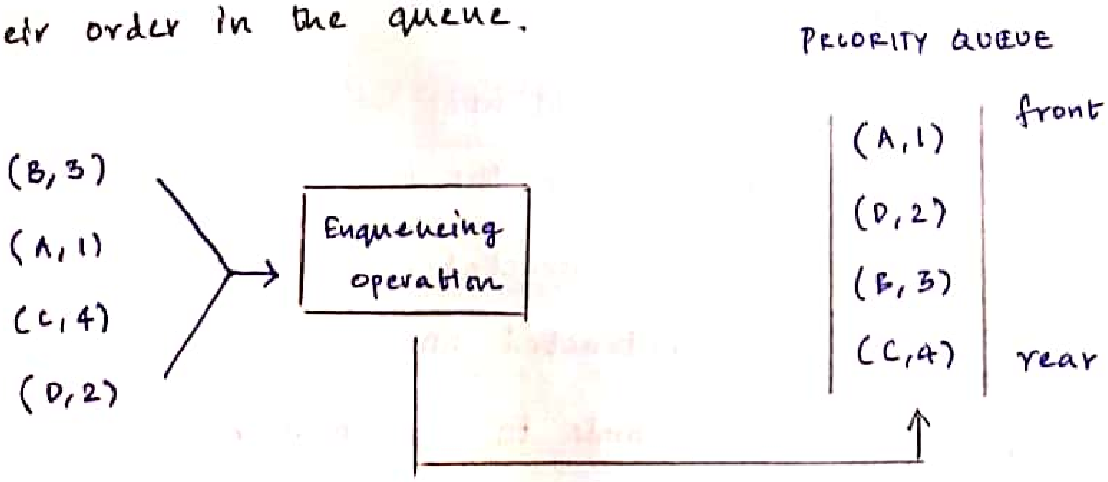


Fig. (b)

(ii) PRIORITY QUEUE

- Each element has a priority associated with it
- Element with higher priority is served before an element with lower priority
- Same priority elements will be served according to their order in the queue.



HOW HUFFMAN CODING WORKS ?

- There are mainly two parts in Huffman Coding
  - i) Build a Huffman Tree from Input characters
  - ii) Traverse the Huffman Tree and assign codes to characters

STEPS TO BUILD A HUFFMAN TREE

INPUT : Array of unique characters along with their frequency of occurences

OUTPUT : Huffman Tree

STEP 1 :- create a leaf node for each unique character and build a min heap of all leaf nodes

NOTE :-
- Min Heap is used as a Priority Queue
- The value of frequency field is used to compare two nodes in min heap
- Initially, the least frequent character is at the root

STEP 2 :- Extract two nodes with the minimum frequency from the min heap

STEP 3 :- create a new Internal node with frequency equal to the sum of the two node frequencies.
- Make the first extracted node as its left child & the other extracted node as its right child
- Add this new node to the min heap

STEP 4 :- Repeat Steps # 2 & Steps # 3 until the heap contains only one node

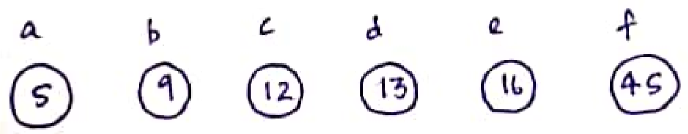
- The remaining node is the root node & the tree is complete.

EXAMPLE

CHARACTER	a	b	c	d	e	f
FREQUENCY	5	9	12	13	16	45

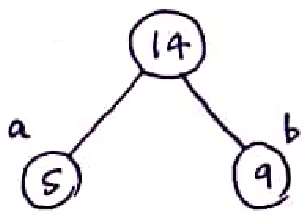
SOLUTION

STEP 1 : Build a min heap that contains 6 nodes where each node represents root of a tree with single node



STEP 2 :- Extract two minimum frequency nodes from min heap.

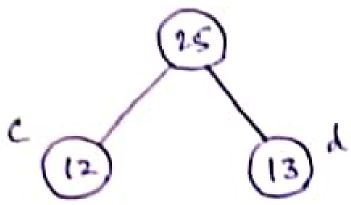
- Add a new internal node with frequency $5 + 9 = 14$



CHARACTER	c	d	INT NODE	e	f
FREQUENCY	12	13	14	16	45

STEP 3 :- Extract two minimum frequency nodes from heap

- Add a new Internal node with frequency $12 + 13 = 25$

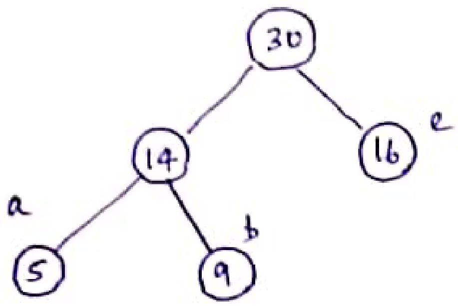


CHARACTER	INT NODE	e	INT NODE	f
FREQUENCY	14	16	25	45

STEP 4 :- + Extract two minimum frequency nodes

+ Add a new internal node with frequency

14 + 16 = 30

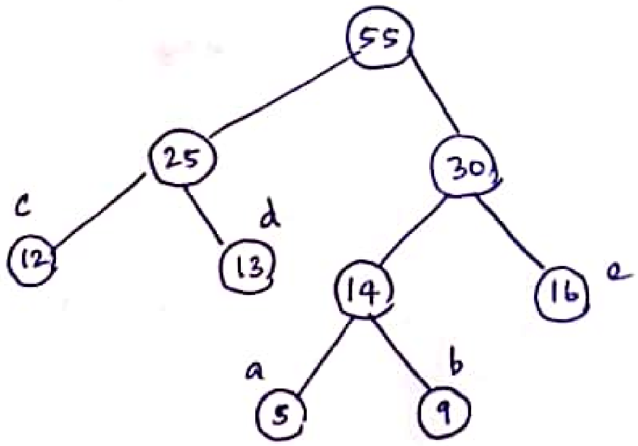


CHARACTER	INT NODE	INT NODE	f
FREQUENCY	25	30	45

STEP 5 :- + Extract two minimum frequency nodes

+ Add a new internal node with frequency

25 + 30 = 55

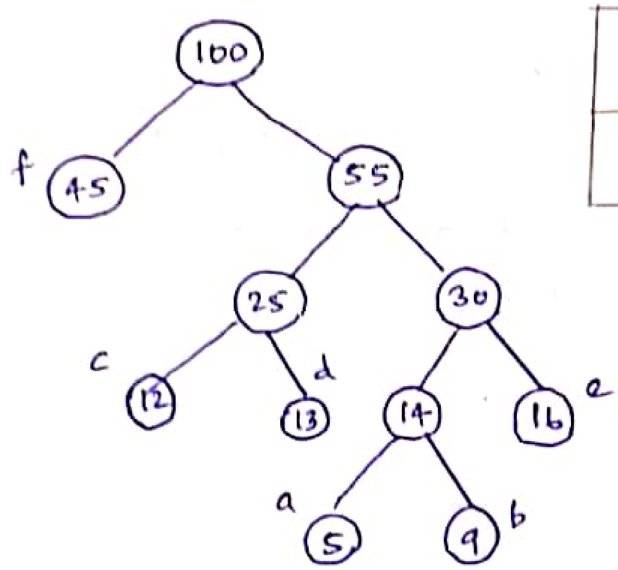


CHARACTER	f	INT NODE
FREQUENCY	45	55

STEP 6 :- + Extract two minimum frequency nodes

+ Add a new internal node with frequency

45 + 55 = 100

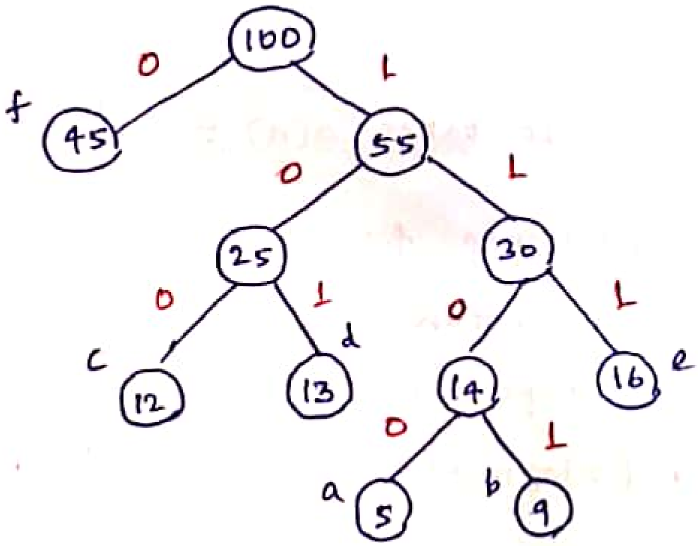


CHARACTER	INT NODE
FREQUENCY	100

- Now min heap contains only one node
- Since the heap contains only one node, the algorithm stops here.

STEPS TO PRINT CODES FROM HUFFMAN TREE

- Traverse the tree formed from the root
- Maintain an auxillary away
- While moving to the left child write 0 to the away
- While moving to the right child write 1 to the away
- Print the away when leaf node is encountered



CHARACTER	CODE WORD
f	0
c	100
d	101
a	1100
b	1101
e	111

HUFFMAN ALGORITHM

- C is a set of characters (n)
- Each character $c \in C$ is an object
  - Attribute c.freq gives the frequency of the character
- The algorithm builds the tree T in a bottom-up manner
- Q is a min priority queue

ALGORITHM : HUFFMAN (C)

```
n := |C|;
Q : = C;
for i = 1 to n-1 do
    allocate a new node z
    z.left = x : = EXTRACT-MIN (Q);
    z.right : = y : = EXTRACT-MIN (Q);
    z.freq = x.freq + y.freq ;
    INSERT (Q, z);
end for
return EXTRACT-MIN (Q); { return the root of the Tree}
```

TIME COMPLEXITY

STEP 1 :- create Minheap. It takes $O(n)$ time for 'n' keys

STEP 2 :- Delete two minimum freq from heap ($2\log n$)

Perform one Insertion ($\log n$)

Total no of steps = $(n-1)$

$\therefore (n-1) + (2\log n + \log n) = O(n\log n)$

Total time complexity of Huffman code is given by

$$O(n) + O(n \log n) = \underline{O(n \log n)}$$

3.3 KNAPSACK PROBLEM

PROBLEM SCENARIO

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are 'n' items available in the store & the weight of i^{th} item is W_i and its profit P_i. What items should the thief take?

Note:- The items should be selected in such a way that the thief will carry those items for which he will gain maximum profit

OBJECTIVE:- To maximize the profit

PROBLEM

Given a set of items, each with a weight & value, determine a subset of items to include in a collection so that the total weight is less than (or) equal to a given limit & the total value is as large as possible.

FRACTIONAL KNAPSACK

- Items can be broken into smaller pieces, hence the thief can select fraction of items
- The knapsack problem is a combinatorial optimization problem
- Applications
  - i) Finding least wasteful way to cut raw materials
  - ii) Portfolio optimization

GENERAL ALGORITHM

GIVEN :- A set of Items 'n' with their respective weights & Value

WEIGHT	w_1	w_2	...	w_n
VALUE	c_1	c_2	...	c_n

GOAL :- Let P be the problem of selecting items from 'n' with weight Limit 'W' such that the resulting cost value is maximum

STEP I :- Calculate $V_i = \frac{c_i}{w_i}$ for $i = 1, 2, \ldots, n$

STEP II :-
- Sort the items by decreasing V_i.
- Let the sorted item/sequence be $1, 2, \ldots i \ldots n$ and the corresponding 'v' & weight be V_i & W_i respectively

STEP III :-
- Let 'K' be the current weight limit.
- In each Iteration, choose item i
  - If $K >= W_i$, take item i & $K = K - W_i$. Then consider the next item.
  - If $K < W_i$, take a fraction f of item i ie.,

$$f = \frac{K}{W_i} \text{ of item } i$$

NOTE :- The Algorithm may take a fraction of an item, which weights exactly 'K'.

SUMMARY

(i) We have a knapsack that has a weight limit 'W'

(ii) There are 'n' items $i_1, i_2, \ldots i_n$ having weights $W_1, W_2, \ldots W_n$ with value $V_1, V_2, \ldots V_n$

(iii) calculate value density for each item

$$\boxed{\text{Value density} = (\text{value} / \text{weight})}$$

(iv) Sort the items as per the value density in descending order

(v) Pick items such that the benifit is maximum & total weight selected is exactly 'W'.

PSEUDOCODE

ALGORITHM : FRACTIONAL-KNAPSACK (W[1...n], P[1,...n], W)

```
for i = 1 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```

EXAMPLE

GIVEN :- Capacity of the knapsack |M| = 60

ITEM	A	B	C	D
WEIGHT	40	10	20	24
PROFIT	280	100	120	120

SOLUTION

STEP I : calculate value density for each item 'i'

ITEM	A	B	C	D
WEIGHT	40	10	20	24
PROFIT	280	100	120	120
VALUE DENSITY	7	10	6	5

STEP II : sort the items as per value density in descending order

ITEM	B	A	C	D
WEIGHT	10	40	20	24
PROFIT	100	280	120	120
VALUE DENSITY	10	7	6	5

STEP III :- From the Sorted table, start selecting items to put into the Knapsack.

(i) First choose B as weight of B is less than the capacity of Knapsack 'W'

(ii) Next item 'A' is chosen, as the Knapsack has enough space to accomodate A

(iii) Now 'C' is chosen as the next item. The whole 'C' cannot be chosen as the capacity of knapsack is less than the remaining Space in W.

Hence fraction of C is chosen i.e.,

$(60-50)/20$ of C is taken

STEP IV :- No more items are selected because the capacity of Knapsack is equal to the selected items.

STEP V :- The total weight of selected items is

$10 + 40 + 20 * (10/20) = 60$

The total profit is

$100 + 280 + 120 * (10/20) = 380 + 60$

$= 440$

This is the optimal solution. We cannot gain more profit selecting any different Combination of Items.

TRACING THE ALGORITHM

ITEM	A	B	C	D
WEIGHT	10	40	20	24
PROFIT	100	280	120	120
VALUE DENSITY	10	7	6	5

W

$\boxed{60}$

(I) for (i = 1 to n)

do x[i] = 0

n

$\boxed{4}$

x[i]

0	1
0	2
0	3
0	4

(II) weight = 0

weight

$\boxed{0}$

(III) $\boxed{i = 1}$

weight + w[i] ≤ W

0 + w[1] ≤ 60

0 + 10 ≤ 60

10 ≤ 60 (TRUE)

x[1] = 1

weight = weight + w[i]

= 0 + 10

$\boxed{\text{weight} = 10}$

w[i]

1	10
2	40
3	20
4	24

P[i]

1	100
2	280
3	120
4	120

when i = 1,

weight

$\boxed{10}$

x[i]

1	1
0	2
0	3
0	4

$\boxed{i = 2}$

weight + w[i] ≤ W

10 + w[2] ≤ 60

$10 + 40 \leq 60$

$50 \leq 60$ (TRUE)

$x[2] = 1$

$\text{weight} = \text{weight} + w[2]$

$= 10 + 40$

$\boxed{\text{weight} = 50}$

for $i = 2$,

weight: $\boxed{50}$

x[i]	
1	1
1	2
0	3
0	4

$\boxed{i = 3}$

$\text{weight} + w[i] \leq W$

$50 + w[3] \leq 60$

$50 + 20 \leq 60$

$70 \leq 60$ (FALSE)

Go to the else part of the program

$x[3] = (60 - 50) / w[3]$

$= (60 - 50) / 20$

$= 10/20$

$x[3] = 0.5$

$\text{weight} = W$

$\boxed{\text{weight} = 60}$

for $i = 3$,

weight: $\boxed{60}$

x[i]	
1	1
1	2
0.5	3
0	4

When weight becomes 60, it is implied that the knapsack cannot hold more than its capacity. Hence the algorithm terminates

CONCLUSION

(i) The total weight of selected items is

$B(10) + A(40) + C[20 * (10/20)]$

$= 10 + 40 + 10 = 60 = W$ (The capacity of knapsack)

(ii) The total profit is

$$100 + 280 + 120 * (10/20)$$

$$= 380 + 60$$

$$= 440$$

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

ANALYSIS OF KNAPSACK PROBLEM

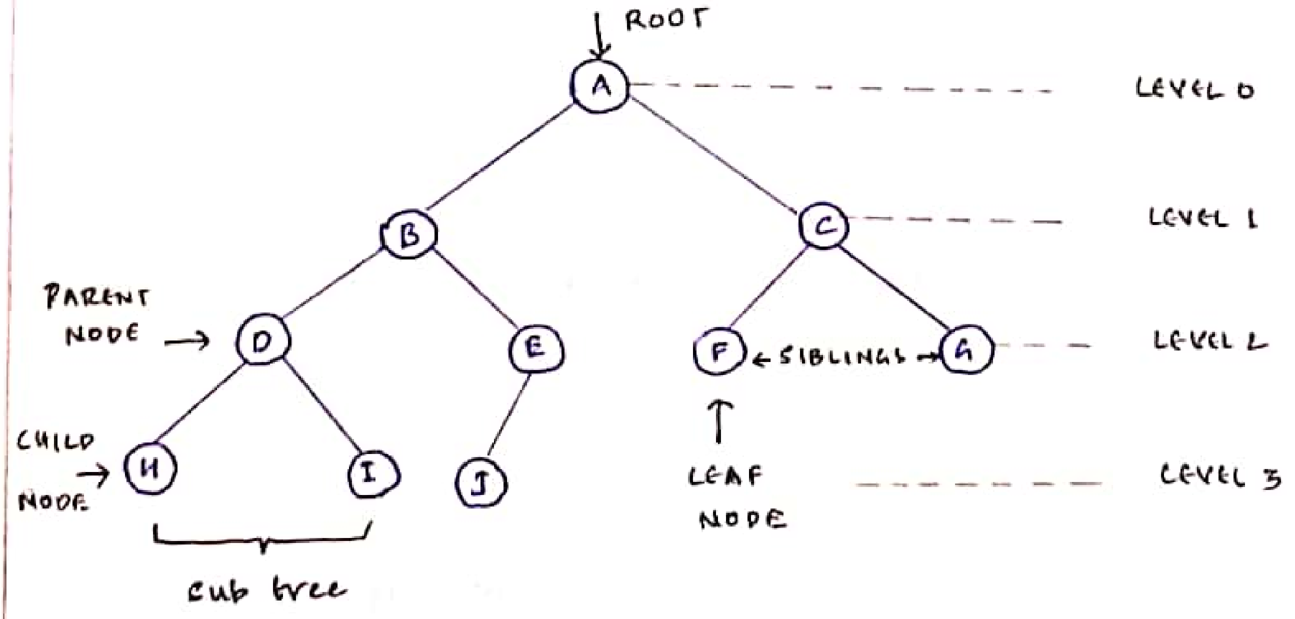
- If the items are already sorted into decreasing order of v_i/w_i then
  - The for loop takes a time $O(n)$

∴ The total time including the sort is $O(n \log n)$

3.4 MINIMUM SPANNING TREE

TREE

DEFINITION: A tree is a datastructure made up of nodes/vertices and edges without having any cycle.



(II) TERMINOLOGIES

(i) **PATH** :- Path refers to the sequence of nodes along the edges of a tree

(ii) **ROOT** :- The node at the top of the tree is called root. There is only one root per tree & one path from the root node to any node.

(iii) **PARENT** :- Any node except the root node has one edge upward to a node called parent

(iv) **CHILD** :- The node below a given node connected by its edge downward is called its child node.

(v) **LEAF** :- The node which does not have any child node is called the leaf node.

(vi) **SUBTREE** :- Represents the descendants of the node

(vii) **TRAVERSING** :- Traversing means passing through nodes in specific order

(viii) **LEVELS** :- Level represents the generation of the node.

(iii) SPANNING TREES

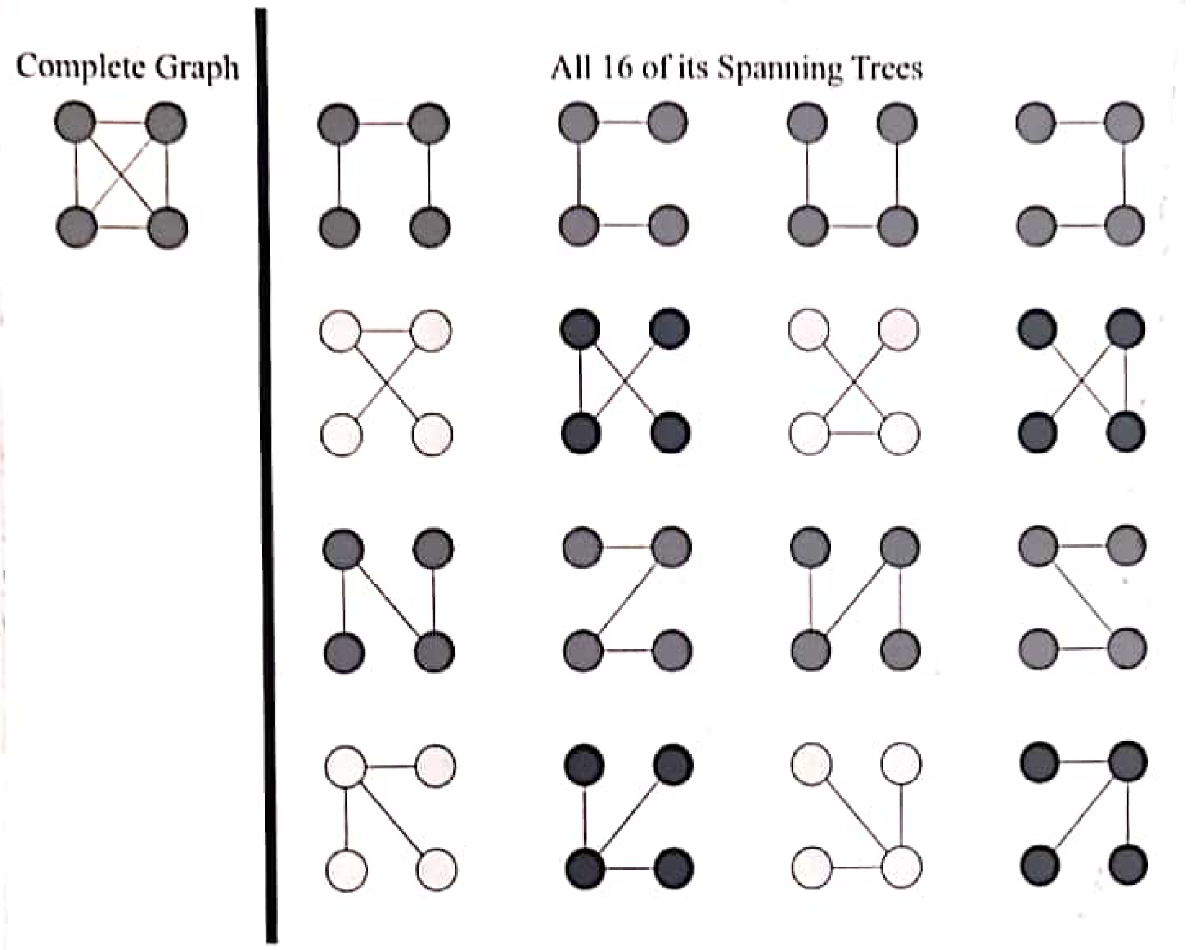
Trees & Graphs :- A tree is a connected acyclic undirected Graph.

GIVEN :- A connected undirected Graph

connected ↓ Every node is reachable from every other node

undirected ↓ edges do not have a associated direction

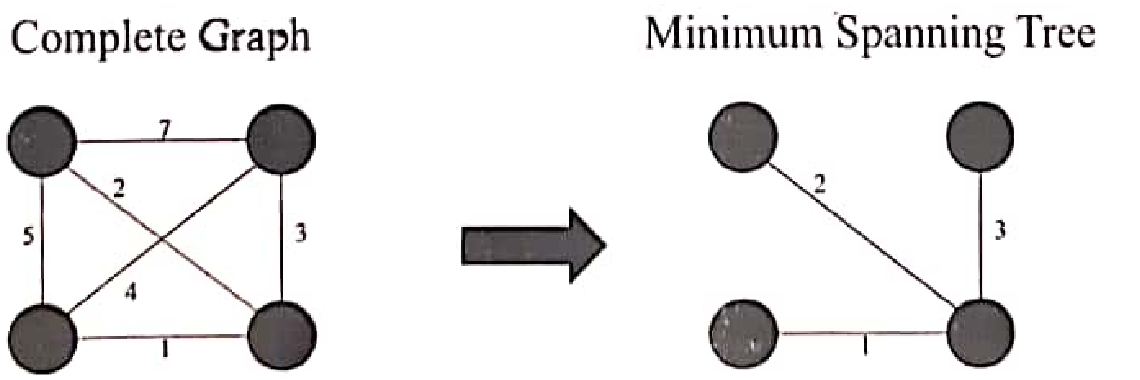
DEFINITION :- A spanning tree of the graph is a connected subgraph in which there are no cycles



NOTE :- A graph may have many spanning Trees

(iv) MINIMUM SPANNING TREES

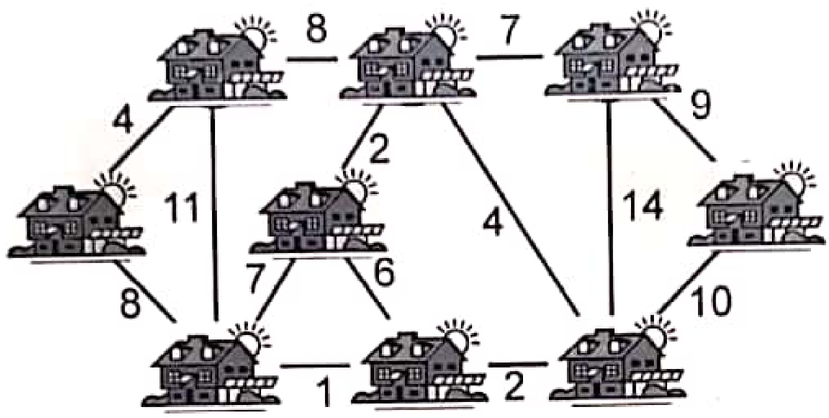
DEFINITION :- The minimum spanning tree for a given graph is the spanning tree of minimum cost for that graph.



EXAMPLE :-

PROBLEM :-

+ A house has a set of houses and a set of roads
+ A road connects 2 & only 2 houses
+ A road connecting houses U & V has a repair cost of $W(U,V)$



+ Vertices = houses
+ Edges = roads

GOAL :- Repair enough (and no more) roads such that :-

(i) Everyone stays Connected

ie., each house can be reached from all other houses

(ii) Total repair cost is minimum

GIVEN :- + A connected, undirected graph G where,

vertices = Houses & Edges = Roads

+ A weight $W(U,V)$ on each edge $(U,V) \in E$

TO FIND :- $T \subseteq E$ such that

i) T connects all vertices

ii) $W(T) = \sum_{(u,v) \in T} W(u,v)$ is minimized

NOTE :- (i) Minimum Spanning Tree is not unique

(ii) Minimum spanning Tree has no cycles

(iii) Number of edges in a Minimum spanning Tree is $|V| - 1$

KRUSKAL'S ALGORITHM

SELF LOOP :- An edge where end vertices are same / Equal

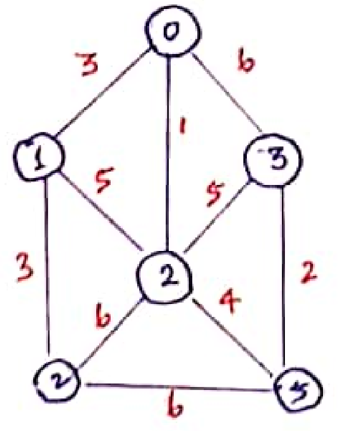
PARALLEL EDGES :- Two or more edges are called parallel edges when they have same end vertices

FOREST :- A tree is a connected graph with no cycles. A forest is a bunch of trees.

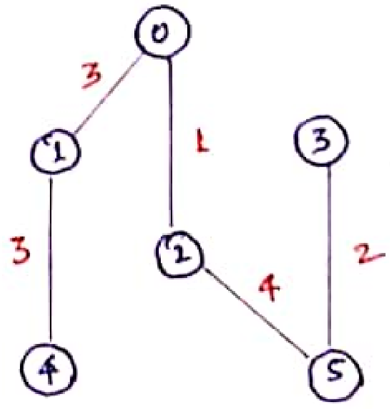
What is Kruskal's Algorithm ?

Kruskal's Algorithm is a Greedy Algorithm in graph theory that finds a minimum spanning Tree for a connected weighted graph.

- It finds the subset of the edges that forms a tree that includes every vertex
- The total weight of all the edges in the tree is minimized



A SIMPLE WEIGHTED GRAPH



MINIMUM COST SPANNING TREE

STEPS IN KRUSKAL'S ALGORITHM

STEP I : Remove all loops and parallel edges

STEP II : Sort all the edges in increasing order of their weights

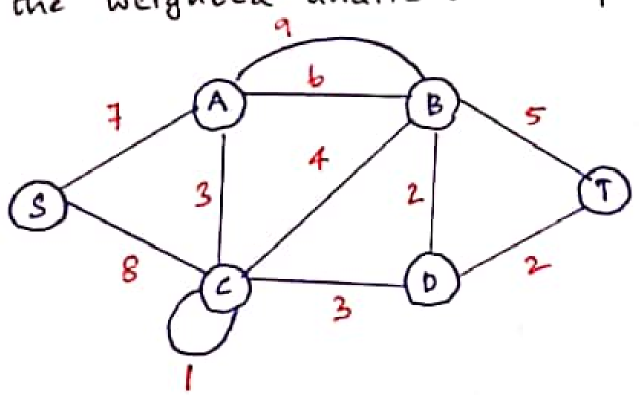
STEP III : (i) Take the edge with lowest weight and add it to the spanning tree

(ii) If adding the edge creates a cycle, reject the edge.

STEP IV : Keep adding vertices until all vertices are reached.

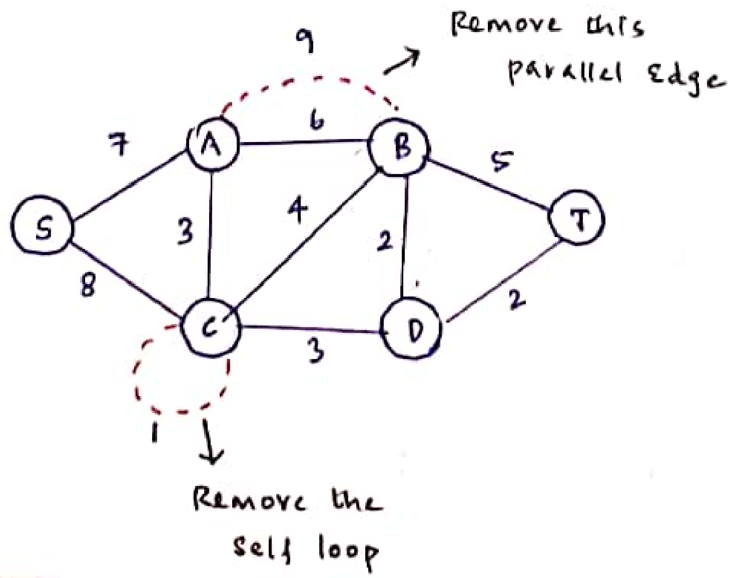
EXAMPLE

Consider the weighted undirected graph given below :-

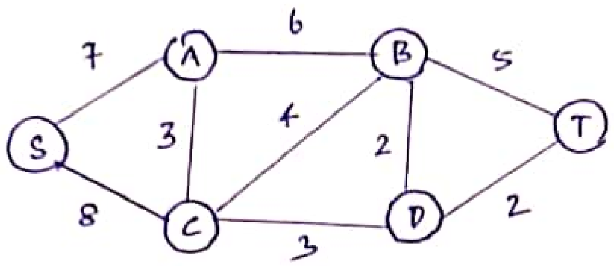


SOLUTION

STEP I : Remove all loops & parallel edges



+ After removing the loops & parallel edges we get :-



STEP II : Arrange all edges in increasing order of their weight.

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8
✓	✓	✓	✓	×	×	×	✓	×

NOTE :- + A ✓ indicates that the edge can be included in the spanning Tree

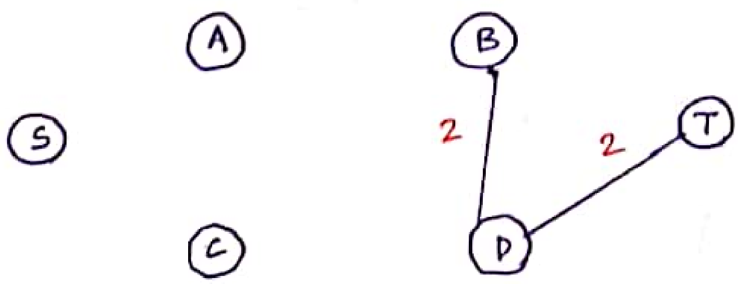
+ A × indicates that the edge forms an cycle.

STEP III : + Start adding edges to the graph beginning with the edge with least weight

+ Throughout, the spanning tree properties should remain intact

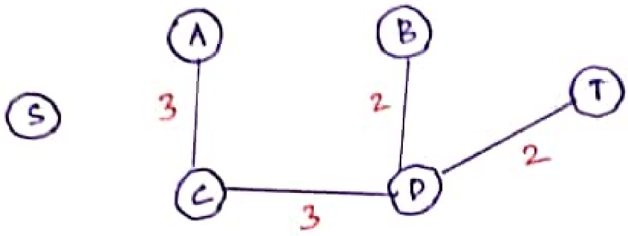
+ Do not add edges that do not satisfy the spanning Tree property.

(i)



+ Least cost is 2. Edges involved as B, D & D, T

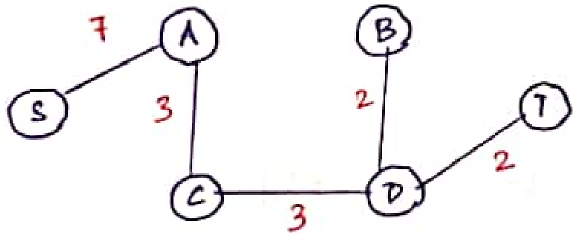
(ii) Next least cost is 3 & associated edges are A, C & C, D



(iii) Next cost in the table is 4. But edge C, B creates a cycle. So we omit the edge.

Similarly B, T & A, B create a cycle. So we omit these edges too.

(iv) The edge S, A with cost 7 does not create a cycle. Add S, A to the minimum spanning Tree



(v) Edge S, C creates a cycle. So we omit the edge S, C.

(vi) The spanning Tree is now complete. The total cost of the minimum spanning Tree is given by

cost (BD) + cost (DT) + cost (AC) + cost (SA) + cost (A, C)

= 2 + 2 + 3 + 7 + 3

= 17

NOTE :- This is the optimal cost of the weighted undirected graph given to construct the minimum spanning Tree.

Scanned by CamScanner

PSEUDO CODE - KRUSKAL'S ALGORITHM

MST - KRUSKAL (G, w)

```
1   A = ∅
2   for each vertex v ∈ G.V
3         MAKE-SET(v)
4   sort the edges of G.E into non-decreasing order by
                                             weight w
5   for each edge (u,v) ∈ G.E taken in non-increasing
                                        order by weight
6           if FIND-SET(u) ≠ FIND-SET(v)
7                  A = A ∪ {(u,v)}
8                  UNION(u,v)
9   return A
```

EXPLANATION OF PSEUDO-CODE

(i) Line 1 initializes A to an empty set

(ii) Lines 2 & 3 create V trees, each containing one vertex

(iii) Line 4 sorts all edges in the original graph by non-decreasing order of their weights

(iv) In lines 5 to 8

- pick edges one by one in the order of their weight (ie., lowest to highest)
- The loop checks for each edge (u, v) whether the endpoints u & v belong to the same tree
    - Adding such edges form a cycle. so it is discarded
- If no cycle is formed line 7 adds u,v to MST

Scanned by CamScanner

(v) Line 8 merges the 2 trees using u & v

(vi) Line 9 returns the set A containing all the edges belonging to the minimum spanning tree

ANALYSIS

- step 2 & 3 takes $O(V)$ time
- step 4 takes $O(E \log E)$ time
- step 5 to 8 takes $O(V \log V)$
- $\therefore$ overall Time complexity is given by

$$O(V) + O(E \log E) + O(V \log V)$$

$$\boxed{\text{COMPLEXITY} = O(V \log V + E \log E)}$$

3.5 DYNAMIC PROGRAMMING

- Powerful technique to solve a particular class of problems

IDEA: (i) If the problem has been solved with the given input then save the result for future reference

(ii) This avoids solving the same problem again in the future

CLUE: Problems with overlapping sub problems come under Dynamic Programming (DP)

METHODS

(i) MEMOIZATION (TOP-DOWN APPROACH)

- start solving the problem by breaking it down
- If the problem has been solved, then just return the answer

+ If the problem has not been solved, solve it and save the answer

(ii) DYNAMIC PROGRAMMING (BOTTOM-UP)

+ Analyse the problem

(a) Find the order in which sub-problems are solved

(b) Start solving from the trivial sub-problem & go up towards the given problem

(c) This guarantees that the subproblems are solved before solving the problem

APPLICATIONS OF DYNAMIC PROGRAMMING

(i) 0/1 knapsack problem

(ii) Travelling salesman problem

(iii) Dijikstra's Algorithm for the shortest path problem

(iv) Tower of Hanoi puzzle

(v) checkerboard

(vi) Sequence Alignment

(vii) Fibonacci Sequence

3.6 0/1 KNAPSACK PROBLEM

+ In 0/1 knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it.

+ 0/1 knapsack cannot be solved by Greedy Approach. Greedy approach does not ensure an optimal solution.

WHY GREEDY APPROACH IS NOT SUITABLE FOR 0/1 KNAPSACK ?

EXAMPLE

- consider the capacity of the knapsack as $W = 30$
- The items are given below :-

ITEM	A	B	C
PRICE	100	280	120
WEIGHT	10	40	20
RATIO	10	7	6

- The Items are selected based on ratio P_i / W_i
- using Greedy Approach,
  - Item A is selected
  - Then Item B is selected
  - Hence Total Profit = 100 + 280 = 380
- However, the optimal Solution of this instance can be achieved by selecting items B & C.
  - The Total profit is 280 + 120 = 400

CONCLUSION :- Greedy Approach may not give an optimal Solution

0/1 KNAPSACK PROBLEM

GIVEN :- A knapsack with a limited weight capacity & some items each of which have a weight and a value.

PROBLEM :- " Which items to place in the knapsack such that the weight limit is not exceeded and the total value of items is as large as possible ? "

HOW TO PROCEED ?

(i) Consider we are given a knapsack of weight 'W' and 'n' number of items with some weights

(ii) Proceed by drawing a table 'T' with $(n+1)$ number of rows & $(W+1)$ number of columns

(iii) Fill all the boxes of 0^{th} row & 0^{th} column with zero

	0	1	2	3	....	W
0	0	0	0	0	....	0
1	0					
2	0					
⋮	⋮					
n	0					

T-TABLE

(iv) Start completing the table row-wise from left to right using the following formula:-

$$T(i,j) = \max\{T(i-1,j),\ (\text{value})_i + T(i-1,\ j-(\text{weight})_i)\}$$

MEANING:- $T(i,j)$ = Maximum value of the selected items. One is allowed to take items 1 to i & have weight restrictions of j

EXAMPLE I

Find the optimal solution for the 0/1 knapsack problem by using the dynamic programming approach. Consider $n=4$, $W=5$, $(w_1, w_2, w_3, w_4) = (2,3,4,5)$ and $(b_1, b_2, b_3, b_4) = (3,4,5,6)$

SOLUTION :-

GIVEN :- Knapsack capacity $(W) = 5$

Number of Items $(n) = 4$

STEP I :- Start by drawing a table T with $(n+1) = 4+1 = 5$ rows & $(W+1) = (5+1) = 6$ columns

STEP II :- Fill all the boxes of 0^{th} row & 0^{th} column with 0

W →

n ↓

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

FINDING T(1,1)

$i = 1, \; j = 1$

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

Substituting we get,

$$T(1,1) = \max\{T(1-1,1),\; 3 + (T(1-1,\; 1-2)\}$$

$$= \max\{T(0,1),\; 3 + \underbrace{T(0,-1)}_{\text{Ignore}}\}$$

$$= T(0,1)$$

$\therefore \; \boxed{T(1,1) = 0}$

GIVEN

ITEM	WEIGHT	VALUE
1	2	3
2	3	4
3	4	5
4	5	6

FINDING T(1,2) :-

$i = 1,\ j = 2$

$(\text{value})_i = (\text{value})_1 = 3$

$(\text{weight})_i = (\text{weight})_1 = 2$

Substituting we get,

$$T(1,2) = \max\{T(1-1, 2),\ 3 + T(1-1, 2-2)\}$$
$$= \max\{T(0,2),\ 3 + T(0,0)\}$$
$$= \max\{0, 3\}$$

$$\therefore \boxed{T(1,2) = 3}$$

FINDING T(1,3) :-

We have,
$i = 1,\ j = 3$

$(\text{value})_i = (\text{value})_1 = 3$
$(\text{weight})_i = (\text{weight})_1 = 2$

Substituting we get,

$$T(1,3) = \max\{T(1-1,3),\ 3 + T(1-1, 3-2)\}$$
$$= \max\{T(0,3),\ 3 + T(0,1)\}$$
$$= \max\{0,\ 3+0\}$$
$$= \max\{0, 3\} \qquad \therefore \boxed{T(1,3) = 3}$$

FINDING T(1,4) :-

We have,
$i = 1,\ j = 4$

$(\text{value})_i = (\text{value})_1 = 3$
$(\text{weight})_i = (\text{weight})_1 = 2$

Substituting we get,

$$T(1,4) = \max\{T(0,4),\ 3 + T(0,2)\}$$

$$\therefore \boxed{T(1,4) = 3}$$

FINDING T(1,5)

We have,

$i = 1$ | $(value)_i = (value)_1 = 3$

$j = 5$ | $(weight)_i = (weight)_1 = 2$

Substituting we get,

$$T(1,5) = \max\{T(1-1,5),\ 3 + T(1-1,\ 5-2)\}$$
$$= \max\{T(0,5),\ 3 + T(0,3)\}$$
$$= \max\{0,\ 3+0\}$$
$$= \max\{0,3\}$$

$\therefore \boxed{T(1,5) = 3}$

- After I iteration the table T becomes,

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

- After all Iterations the table T becomes,

WEIGHT		0	1	2	3	4	5
	0	0	0	0	0	0	0
2	1	0	0	3	3	3	3
3	2	0	0	3	4	4	7
4	3	0	0	3	4	5	7
5	4	0	0	3	4	5	7

Represents the maximum possible value which can be put in the knapsack

Maximum value that can be put in the knapsack	= 7

FINDING ITEMS TO BE PUT IN A KNAPSACK TO GET MAXIMUM POSSIBLE VALUE 7

(i) consider the last column of the table and start scanning from bottom to top

(ii) If an entry whose value is not the same as the value stored in the previous entry, mark the label of row of that entry

(iii) the rows labelled '1' & '2' are marked.

(iv) The items to be put in a knapsack to obtain the maximum value 7 are

Item 1 & Item 2

PSEUDO-CODE - 0/1 KNAPSACK PROBLEM

* c is an $(n+1) \times (W+1)$ array;

$$c[0, \ldots n : 0 \ldots W]$$

NOTE :- Table is computed in row-major order

```
KNAPO-1(v, w, n, W)
    for w ← 0 to W do
        c[0,w] ← 0
    for i ← 1 to n do
        c[i,0] ← 0
```

Scanned by CamScanner

```
for i ← 1 to n do
    for w ← 1 to W do
        if w_i ≤ w then
            c[i,w] ← max {v_i + c[i-1, w-w_i], c[i-1,w]}
        else
            c[i,w] ← c[i-1,w]
return c[n, W]
```

TIME COMPLEXITY: $O(nW)$ where,

n is the number of items &

W is the knapsack capacity

3.7 TRAVELLING SALESMAN PROBLEM

PROBLEM STATEMENT

A traveller needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city ?

SOLUTION

(1) **BRUTE FORCE APPROACH**

- Evaluate every possible tour and select the best one.
- For 'n' number of vertices in a graph there are $(n-1)!$ number of possibilities
- Time complexity is $(n!)$

(ii) DYNAMIC PROGRAMMING APPROACH

+ consider a graph $G = (V, E)$ where,
  + V is a set of cities
  + E is a set of weighted edges
  + edge (u, v) represents that vertices u & v are connected
  + Distance between vertex u & v is $d(u, v)$
+ Let $G = (V, E)$ be a directed graph with edge costs c_{ij} where,
  + $c_{ij} >= 0$ for all i & j
  + $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$
+ Let $|V| = n$ & assume $n > 1$ where,
  + V is the set of vertices &
  + n is the number of vertices

TOUR : A tour of G is a directed simple cycle that includes every vertex of G

COST : The cost of a tour is the sum of the cost of the edges on the Tour

IDEA BEHIND TSP : To find a tour with minimum cost

CONDITION :- (i) All the nodes are to be traversed

(ii) only the starting node could be traversed twice

(iii) Except the first nodes all the nodes should be traversed & traversed only once

$$\boxed{g(i,s) = \min_{j \in s} \{ c_{ij} + g(j, s - \{j\}) \}}$$

where,

$i \rightarrow$ The salesperson is at the i^{th} node

$S \rightarrow$ The remaining set of vertices which have not been traversed yet

$j \rightarrow$ The next node to i & $j \in S$

$c_{ij} \rightarrow$ cost from i to j

$g(j, S-\{j\}) \rightarrow$ Recursive call of $g(i, S)$ for obtaining minimum cost

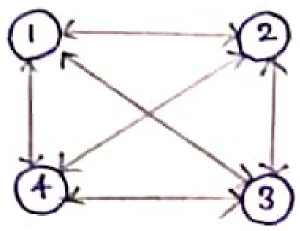
COMPARISON BETWEEN BRUTE FORCE & DYNAMIC PROGRAMMING

+ Time complexity of Brute force is $n!$
+ Time complexity of Dynamic programming is 2^n

NUMBER OF VERTICES 'n'	TIME COMPLEXITY	
	$n!$	2^n
1	1	2
2	2	4
3	6	8
4	24	16
5	120	32

CONCLUSION :- $\boxed{n! > 2^n}$, when $n > 3$

EXAMPLE



$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$

COST MATRIX

SOLUTION

STEP I : Let the starting vertex be 1

STEP II : Exclude vertex 1 take the remaining vertices 1, 2 & 3 & generate all possible subsets

VALUE OF S	SUBSET
ϕ	ϕ
1	{2} {3} {4}
2	{3, 4} {2, 4} {2, 3}
3	{2, 3, 4}

STEP III : Start with $\boxed{S = \phi}$

Consider $g(2, \phi)$. Here $j = 2$ & $S = \phi$.

- The travelling salesperson is at Node No : 2
- $S = \phi$ denotes there are no remaining untraversed nodes
- ∴ The cost from 2 to 1 should be computed ie., C_{21}

Thus $g(2, \phi) = C_{21} = 5$

$\therefore$ At $s = \phi$ we get,

FUNCTION	COST	PARENT NODE
$g(2, \phi) = c_{21}$	5	1
$g(3, \phi) = c_{31}$	6	1
$g(4, \phi) = c_{41}$	8	1

$\boxed{S = 1}$

consider $g(2, \{3\})$. Here $j = 2$ & $s = \{3\}$

- The travelling sales person is at Node No : 2
- $s = \{3\}$ denotes that there is one untraversed node 3.
- The cost from 2 to 3 should be computes ie.,
  $c_{23} = 9$

Thus $g(2, \{3\}) = c_{23} + g(3, \phi) = 9 + 6 = 15$

$\therefore$ At $s = 1$ we get,

FUNCTION	COST	PARENT NODE
$g(2, \{3\}) = c_{23} + g(3, \phi) = 9 + 16$	15	3
$g(2, \{4\}) = c_{24} + g(4, \phi) = 10 + 8$	18	4
$g(3, \{2\}) = c_{32} + g(2, \phi) = 13 + 5$	18	2
$g(3, \{4\}) = c_{34} + g(4, \phi) = 12 + 8$	20	4
$g(4, \{2\}) = c_{42} + g(2, \phi) = 8 + 5$	13	2
$g(4, \{3\}) = c_{43} + g(3, \phi) = 9 + 6$	15	3

$\boxed{S = 2}$

Now we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ & $i \notin S$

$\therefore$ At $s = 2$ we get,

FUNCTION	COST	PARENT NODE
$g(2, \{3,4\}) = \min \begin{cases} c_{13} + g(3, \{4\}) = 9 + 20 = 29 \\ c_{24} + g(4, \{3\}) = 10 + 15 = \underline{25} \end{cases}$	25	2
$g(3, \{2,4\}) = \min \begin{cases} c_{32} + g(2, \{4\}) = 13 + 18 = 31 \\ c_{34} + g(\{4\}, 2) = 12 + 13 = \underline{25} \end{cases}$	25	3
$g(4, \{2,3\}) = \min \begin{cases} c_{42} + g(2, \{3\}) = 8 + 15 = \underline{23} \\ c_{43} + g(3, \{2\}) = 9 + 18 = 27 \end{cases}$	23	2

$\boxed{S = 3}$

At $S = 3$ we get,

FUNCTION	COST	PARENT NODE
$g(1, \{2,3,4\}) = \min \begin{cases} c_{12} + g(2, \{3,4\}) = \underline{35} \\ c_{13} + g(3, \{2,4\}) = 40 \\ c_{14} + g(4, \{2,3\}) = 43 \end{cases}$	35	

<u>CONCLUSION</u> :- An optimal tour of the graph has length $\boxed{25}$

FINDING THE OPTIMAL TOUR

(i) start from cost $\{1, \{2,3,4\}\}$. we get minimum value for C_{12}.

$\therefore \quad 1 \xrightarrow{10} 2$

(ii) When $s = 2$, we get minimum value for C_{42}. $\therefore$

$1 \xrightarrow{10} 2 \xrightarrow{10} 4$

(iii) At $s = 1$, we get minimum value for C_{42}. But 2 & 4 are already selected.

- The next minimum value is for C_{43} & C_{23} ie., 15
- we select C_{43} because our last node is 4.

$1 \xrightarrow{10} 2 \xrightarrow{10} 4 \xrightarrow{9} 3$

(iv) And finally we select C_{31}. The optimal tour thus becomes

$$\boxed{1 \xrightarrow{10} 2 \xrightarrow{10} 4 \xrightarrow{9} 3 \xrightarrow{6} 1}$$

PSEUDO-CODE FOR TRAVELLING SALESMAN PROBLEM

ALGORITHM : TRAVELLING SALESMAN-PROBLEM

$C(\{1\}, 1) = 0$

for $s = 2$ to n do

for all subsets $S \in (1, 2, 3, \ldots n)$ of size s & containing 1

$C(S, 1) = \alpha$

for all $j \in S$ and $j \neq 1$

$C(S, j) = \min \{ c(S - \{j\}, i) + d(i, j)$ for $i \in S$ and $i \neq j \}$

Return $\min_j \; c(\{1, 2, 3, \ldots n\}, j) + d(j, i)$

TIME COMPLEXITY ANALYSIS

+ There are at most $O(n * 2^n)$ subproblems &

+ Each subproblem takes linear time to solve.

• The total running time is $\boxed{O(n^2 * 2^n)}$

3.8 MULTISTAGE GRAPHS

+ A multistage graph $G = (V, E)$ is a directed graph (usually weighted) in which vertices are partitioned into $K \geq 2$ disjoint set (V_i) where $[1 \leq I \leq K]$.

+ If (u, v) is an edge E then $u \in V_i$, $v \in V_{i+1}$

+ Stage 1 has only one vertex called S (source) & last stage has only one vertex called T (Target)

+ Let $c(i, j)$ be the cost of edge (i, j)

+ The cost of a path from S to T is the sum of costs of the edges on the path.

PROBLEM : To find the minimum cost path from S to T

(i) SOLUTION - FORWARD APPROACH

+ Memorize the costs in such a way that any stage depends on the stage next to it [FORWARD]

BASE CASE : To find the minimum cost of path from vertices in last but one stage to the last stage.

(a) **DEFINE SUB PROBLEMS**

+ Let cost (i,j) be the minimum cost of path from vertex j of stage i to the target vertex

(b) **RECURRENCE RELATING SUB-PROBLEMS**

$$\boxed{cost(i,j) = \min\{c(i,j) + cost(i+1, l)\}}$$

(c) **RECOGNIZE AND SOLVE BASE CASES**

+ If there are total of n stages, then target T is at stage n, for the previous stage ie, n-1, $cost(n-1, j) = c(j, T)$

(ii) **SOLUTION - BACKWARD APPROACH**

+ Memorize the costs in such a way that for any stage depends on the stage before it [BACKWARD]

BASE CASE :- To find the minimum cost of path from vertices in first stage to next stage.

(a) **DEFINE SUB PROBLEMS**

+ Let cost (i,j) be the minimum cost of path from source S to vertex j of stage i

(b) **RECURRENCE RELATING SUB PROBLEM**

$$\boxed{cost(i,j) = \min\{c(l,j) + cost(i-1, l)\}}$$

(c) RECOGNIZE & SOLVE BASE CASES

- If there are a total of 'n' stages, then source S is at stage 0 for the next stage ie., 1

ie., $cost(1, j) = c(s, j)$